The magic of Algorithms!

Lectures on some algorithmic pearls

Paolo Ferragina, Università di Pisa

These notes should be an advise for programmers and software engineers: no matter how much smart you are, the so called *"5-minutes thinking"* is not enough to get a reasonable solution for your real problem, unless it is a toy one! Real problems have reached such a large size, machines got so complicated, and algorithmic tools became so sophisticated that you cannot improvise to be an *algorithm designer*: you should be trained to be one of them!

These lectures provide a witness for this issue by introducing challenging problems together with elegant and efficient algorithmic techniques to solve them. In selecting their topics I was driven by a twofold goal: from the one hand, provide the reader with an *algorithm engineering toolbox* that will help him/her in attacking programming problems over massive datasets; and, from the other hand, I wished to collect the stuff that I would have liked to see when I was a master/phd student!

The style and content of these lectures is the result of many hours of highlighting and, sometime hard and fatiguing, discussions with many fellow researchers and students. Actually some of these lectures composed the courses in Information Retrieval and/or Advanced Algorithms that I taught at the University of Pisa and in various International PhD Schools, since year 2004. In particular, a preliminary draft of these notes were prepared by the students of the *"Algorithm Engineering"* course in the Master Degree of Computer Science and Networking in Sept-Dec 2009, done in collaboration between the University of Pisa and Scuola Superiore S. Anna. Some other notes were prepared by the Phd students attending the course on *"Advanced Algorithms for Massive DataSets"* that I taught at the BISS International School on Computer Science and Engineering, held in March 2010 (Bertinoro, Italy). I used these drafts as a seed for some of the following chapters.

My ultimate hope is that reading these notes you'll be pervaded by the same pleasure and excitement that filled my mood when I met these algorithmic solutions for the first time. If this will be the case, please read more about Algorithms to find inspiration for your work. It is still the time that *programming is an Art*, but you need the good *tools* to make itself express at the highest beauty!

P.F.

Contents

1 Prologo

"This is a rocket science but you don't need to be a rocket scientist to use it"
The Economist, September 2007

The main actor of this book is *the Algorithm* so, in order to dig into the beauty and challenges that pertain with its ideation and design, we need to start from one of its many possible definitions. The OXFORD ENGLISH DICTIONARY reports that an algorithm is, informally, *"a process, or set of rules, usually one expressed in algebraic notation, now used esp. in computing, machine translation and linguistics"*. The modern meaning for Algorithm is quite similar to that of *recipe, method, procedure, routine* except that the word Algorithm in Computer Science connotes something more precisely described. In fact many authoritative researchers have tried to pin down the term over the last 200 years [3] by proposing definitions which became more complicated and detailed nonetheless, hopefully in the minds of their proponents, more precise and elegant. As algorithm designers and engineers we will follow the definition provided by Donald Knuth at the end of the 60s [7, pag 4]: an Algorithm is *a finite, definite, effective procedure, with some output.* Although these five features may be intuitively clear and are widely accepted as requirements for a sequence-of-steps to be an Algorithm, they are so dense of significance that we need to look into them with some more detail, even because this investigation will surprisingly lead us to the scenario and challenges posed nowadays by algorithm design and engineering, and to the motivation underling these lectures.

Finite: *"An algorithm must always terminate after a finite number of steps ... a very finite number, a reasonable number."* Clearly, the term "reasonable" is related to the *efficiency* of the algorithm: Knuth [7, pag. 7] states that *"In practice, we not only want algorithms, we want good algorithms"*. The "goodness" of an algorithm is related to the use that the algorithm makes of some precious *computational resources* such as: time, space, communication, I/Os, energy, or just simplicity and elegance which both impact onto the coding, debugging and maintenance costs!

Definite: *"Each step of an algorithm must be precisely defined; the actions to be carried out must be rigorously and unambiguously specified for each case"*. Knuth made an effort in this direction by detailing what he called the "machine language" for his *"mythical MIX...the world's first polyunsaturated computer"*. Today we know of many other programming languages such as `C/C++`, `Java`, `Python`, etc. etc. All of them specify a set of instructions that the programmer may use to describe the procedure underlying his/her algorithm in an unambiguous way: "unambiguity" here is granted by the formal semantics that researchers have attached to each of these instructions. This eventually means that anyone reading the algorithm's description will interpret it in a precise way: nothing will be left to personal mood!

Effective: *"... all of the operations to be performed in the algorithm must be sufficiently basic that they can in principle be done exactly and in a finite length of time by a man using paper and pencil".* Therefore the notion of "step" invoked in the previous item implies that one has to dig into a complete and deep understanding of the problem to be solved, and then into logical well-definite structuring of a step-by-step solution.

Procedure: *"... the sequence of specific steps arranged in a logical order".*

Input: *"... quantities which are given to it initially before the algorithm begins. These inputs are taken from specified sets of objects".*

Output: *"... quantities which have a specified relation to the inputs".*

In this booklet we will not use a *formal* approach to algorithm description, because we wish to concentrate on the theoretically elegant and practically efficient ideas which underlie the algorithmic solution of some interesting problems, without being lost in the maze of programming technicalities. So in every lecture we will take an interesting *problem* coming out from a *real/useful application* and then propose *deeper and deeper* solutions of increasing sophistication and improved efficiency, taking care that this will not necessarily correspond to increasing the complexity of algorithm's description. Actually, problems were selected to admit *surprisingly* elegant solutions that can be described in few lines of code! So we will opt for the current practice of algorithm design and describe our algorithms either *colloquially* or by using *pseudo-code* that mimics the most famous `C` and `Java` languages. In any case we will not renounce to be as much rigorous as it needs an algorithm description to match the five features above.

Elegance will not be the only feature of our algorithm design, of course, we will also aim for *efficiency* which commonly relates to the *time/space complexity* of the algorithm. Traditionally time complexity has been evaluated as a function of the input size n by counting the (maximum) number of steps, say $T(n)$, an algorithm takes to complete its computation over an input of n items. Since the maximum is taken over all inputs of that size, the time complexity is named *worst case* because it concerns with the input that induces the worst behavior in time for the algorithm. Of course, the larger is n the larger is $T(n)$, which is therefore non decreasing and positive. In a similar way we can define the (worst-case) *space complexity* of an algorithm, as the maximum number of memory cells it uses for its computation over an input of size n. This approach to the *design* and *analysis* of algorithms assumes a very simple model of computation, known as *model of Von Neumann* (aka Random Access Machine, *RAM* model). This model consists of a CPU and a memory of infinite size and constant-time access to each one of its cells. Here we argue that every step takes a fixed amount of time on a PC, which is the same for any operation: being it arithmetic, logical, or just a memory access (read/write). Here one postulates that it is enough to *count* the number of steps executed by the algorithm in order to have an "accurate" estimate of its execution time on a real PC. Two algorithms can then be *compared* according to the *asymptotic behavior* of their time-complexity functions as $n \longrightarrow +\infty$, the faster is growing the time complexity over inputs of larger and larger size, the worse is its corresponding algorithm. The robustness of this approach has been debated for a long time but, eventually, the RAM model dominated the algorithmic scene for decades (and is still dominating it!) because of its simplicity, which impacts on algorithm design and evaluation, and its ability to estimate the algorithm performance "quite accurately" on (old) PCs. Therefore it is not surprising that most introductory books on Algorithms take the RAM model as a reference.

But in the last ten years things have changed significantly, thus highlighting the need for a *shift* in algorithm design and analysis! Two main changes occurred: the architecture of modern PCs became more and more sophisticated (not just one CPU and one monolithic memory!), and input data have exploded in size ("$n \longrightarrow +\infty$" does not live only in the theory world!) because they are abundantly generated by many sources: such as DNA sequencing, bank transactions, mobile communications, Web navigation and searches, auctions, etc. etc.. The first change turned the RAM model into an unsatisfactory abstraction of modern PCs; whereas the second change made the design

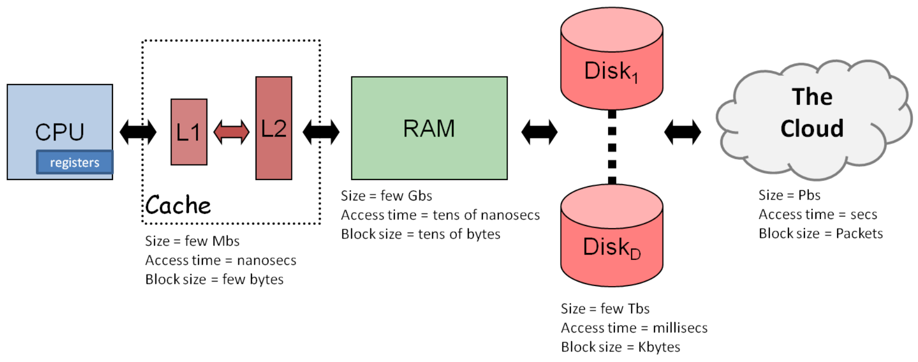


FIGURE 1.1: An example of memory hierarchy of a modern PC.

of *asymptotically good* algorithms ubiquitous and fruitful not only for dome-headed mathematicians but also for a much larger audience because of their impact on business [2], society [1] and science in general [4]. The net consequence was a revamped scientific interest in algorithmics and the spreading of the word “Algorithm” to even colloquial speeches!

In order to make algorithms effective in this new scenario, researchers needed new models of computation able to abstract in a better way the features of modern computers and applications and, in turn, to derive more accurate estimates of algorithm performance from their complexity analysis. Nowadays a modern PC consists of one or more CPUs (multi-core?) and a very complex hierarchy of memory levels, all with their own technological specialties (Figure 1.1): L1 and L2 caches, internal memory, one or more mechanical or SSDisks, and possibly other (hierarchical-)memories of multiple hosts distributed over a (possibly geographic) network, the so called “Cloud”. Each of these memory levels has its own cost, capacity, latency, bandwidth and access method. The closer a memory level is to the CPU, the smaller, the faster and the more expensive it is. Currently nanoseconds suffice to access the caches, whereas milliseconds are yet needed to fetch data from disks (aka I/O). This is the so called *I/O-bottleneck* which amounts to the astonishing factor of $10^5 - 10^6$, nicely illustrated by Tomas H. Cormen with his quote:

“The difference in speed between modern CPU and (mechanical) disk technologies is analogous to the difference in speed in sharpening a pencil using a sharpener on one’s desk or by taking an airplane to the other side of the world and using a sharpener on someone else’s desk”.

Engineering research is trying nowadays to improve the input/output subsystem to reduce the impact of the I/O-bottleneck onto the efficiency of applications managing large datasets; but, on the other hand, we are aware that the improvements achievable by means of a good algorithm design abundantly surpass the best expected technology advancements. Let us see the why with a simple example![1]

Assume to take three algorithms having increasing I/O-complexity: $C_1(n) = n$, $C_2(n) = n^2$ and $C_3(n) = 2^n$. Here $C_i(n)$ denotes the number of disk accesses executed by the ith algorithm to process n input data (stored e.g. in n/B disk pages). Notice that the first two algorithms execute a *polynomial* number of I/Os (in the input size), whereas the last one executes an *exponential* number of I/Os. Moreover we note that the above complexities have a very simple (and thus unnatural) mathematical form because we want to simplify the calculations without impairing our final conclusions. Let us

[1]This is paraphrased from [8], now we talk about I/Os instead of steps.

now ask for how many data each of these algorithms is able to process in a fixed time-interval of size t, given that each I/O takes c time. The answer is obtained by solving the equation $C_i(n) \times c = t$ with respect to n: so we get t/c data are processed by the first algorithm in t time, $\sqrt{t/c}$ data are processed by the second algorithm, and only $\log_2(t/c)$ data are processed by the third algorithm. These values are already impressive by themselves, and provide a robust understanding of why polynomial-time algorithms are called *efficient*, whereas exponential-time algorithms are called *inefficient*: a large change in the length t of the time-interval induces just a tiny change in the amount of data that exponential-time algorithms can process. Of course, this distinction admits many exceptions when the problem instances have limited size or have distributions which favor efficient executions (think e.g. to the simplex algorithm). But, on the other hand, these examples are quite rare, and the much more stringent bounds on execution time satisfied by polynomial-time algorithms make them considered *provably* efficient and the preferred way to solve problems. Algorithmically speaking, most exponential-time algorithms are merely implementations of the approach based on *exhaustive* search, whereas polynomial-time algorithms are generally made possible only through the gain of some deeper insight into the structure of a problem. So polynomial-time algorithms are the *right* choice from many points of view.

Let us now assume to run the above algorithms with a better I/O-subsystem, say one that is k times faster, and ask: How many data can be managed by this new PC? To address this question we solve the previous equations with the time-interval set to the length $k \times t$, thus implicitly assuming to run the algorithms over the old PC but providing itself with k times more time. We get that the first algorithm perfectly scales by a factor k, the second algorithm scales by a factor $\sqrt{k}$, whereas the last algorithm scales *only of an additional term* $\log_2 k$. Noticeably the improvement induced by a k-times more powerful PC for an exponential-time algorithm is totally negligible even in the presence of impressive (and thus unnatural) technology advancements! Super-linear algorithms, like the second one, are positively affected by technology advancements but their performance improvement decreases as the degree of the polynomial-time complexity grows: more precisely, if $C(n) = n^\alpha$ then a k-times more powerful PC induces a speed-up of a factor $\sqrt[\alpha]{k}$. Overall, it is not hazardous to state that the impact of a good algorithm is far beyond any optimistic forecasting for the performance of future (mechanical or SSD) disks.[2]

Given this *appetizer* about the "Power of Algorithms", let us now turn back to the problem of analyzing the performance of algorithms in modern PCs by considering the following simple example: Compute the sum of the integers stored in an array $A[1, n]$. The simplest idea is to scan A and accumulate in a temporary variable the sum of the scanned integers. This algorithm executes n sums, accesses each integer in A once, and thus takes n steps. Let us now generalize this approach by considering a family of algorithms, denoted by $\mathcal{A}_{s,b}$, which differentiate themselves according to the pattern of accesses to A's elements, as driven by the parameters s and b. In particular $\mathcal{A}_{s,b}$ looks at array A as logically divided into blocks of b elements each, say $A_j = A[j * b + 1, (j + 1) * b]$ for $j = 0, 1, 2, \ldots, n/b - 1$.[3] Then it sums all items in one block before moving to the next block that is s blocks apart on the right. Array A is considered cyclic so that, when the next block lies out of A, the algorithm wraps around it starting again from its beginning. Clearly not all values of s allow to take into account all of A's blocks (and thus sum all of A's integers). Nevertheless we know that if s is co-prime with n/b then $[s \times i \mod (n/b)]$ generates a permutation of the integers $\{0, 1, \ldots, n/b - 1\}$, and thus $\mathcal{A}_{s,b}$ touches all blocks in A and hence sums all of its integers. But the specialty of this parametrization is that by varying s and b we can sum according to different patterns of memory accesses: from the sequential scan indicated above (setting $s = b = 1$), to a block-wise

[2]See [11] for an extended treatment of this subject.

[3]For the sake of presentation we assume that n and b are powers of two, so b divides n.

access (set a large b) and/or a random-wise access (set a large s). Of course, all algorithms $\mathcal{A}_{s,b}$ are equivalent from a computational point of view, since they read exactly n integers and thus take n steps; but from a practical point of view, they have different time performance which becomes more and more significant as the array size n grows. The reason is that, for a growing n, data will be spread over more and more memory levels, each one with its own capacity, latency, bandwidth and access method. So the "equivalence in efficiency" derived by adopting the RAM model, and counting the number-of-steps executed by $\mathcal{A}_{s,b}$, is not an accurate estimate of the real time required by that algorithms to sum A's elements.

We need a different model that grasps the essence of real computers and is simple enough to not jeopardize algorithm design and analysis. In a previous example we already argued that the number of I/Os is a good estimator for the time complexity of an algorithm, given the large gap existing between disk- and internal-memory accesses. This is indeed what is captured by the so called *2-level memory model* (aka. disk-model, or external-memory model [11]) which abstracts the computer as composed by only *two memory levels*: the internal memory of size M, and the (unbounded) disk memory which operates by reading/writing data via blocks of size B (called *disk pages*). Sometimes the model consists of D disks, each of unbounded size, so that each I/O reads or writes a total of $D \times B$ items coming from D pages, each one residing on a different disk. For the sake of clarity we remark that the *two-level view* must not suggest to the reader that this model is restricted to abstracts disk-based computations; in fact, we are actually free to choose any two levels of the memory hierarchy, with their M and B parameters properly set. The algorithm performance is evaluated in this model by counting: (a) the number of accesses to disk pages (hereafter *I/O*s), (b) the internal running time (CPU time), and (c) the number of disk pages used by the algorithm as its working space. This suggests *two golden rules* for the design of "good" algorithms operating on large datasets: they must exploit *spatial locality* and *temporal locality*. The former imposes a data organization onto the disk(s) that makes each accessed disk-page as much useful as possible; the latter imposes to execute as much useful work as possible onto data fetched in internal memory, before they are written back to disk.

In the light of this new model, let us re-analyze the time complexity of algorithms $\mathcal{A}_{s,b}$ by taking into account I/Os, given that the CPU time is still n and the space occupancy is n/B pages. We start from the simplest settings for s and b in order to gain some intuitions about the general formulas. The case $s = 1$ is obvious, algorithms $\mathcal{A}_{1,b}$ scan A rightward by taking n/B I/Os, independently of the value of b. As s and b change the situation complicates, but by not much. Fix $s = 2$ and pick some $b < B$ that, for simplicity, is assumed to divide the block-size B. Every block of size B consists of B/b smaller (logical) blocks of size b, and the algorithm $\mathcal{A}_{2,b}$ examines only half of them because of the jump $s = 2$. This actually means that each B-sized page is half utilized in the summing process, thus inducing a total of $2n/B$ I/Os. It is then not difficult to generalize this formula by writing a cost of $\min\{s, B/b\} \times (n/B)$ I/Os, which correctly gives n/b for the case of large jumps over array A. This formula provides a better approximation of the real time complexity of the algorithm, although it does not capture all features of the disk. In fact, it considers all I/Os as *equal*, independently of their distribution. This is clearly unprecise because on real disks the *sequential* I/Os are faster than the *random* I/Os.[4] Referring to the previous example, the algorithms $\mathcal{A}_{s,B}$ have still I/O-complexity n/B, independently of s, although their behavior is rather different if executed on a (mechanical) disk because of the disk seeks induced by larger and larger s. As a result, we can conclude that even the 2-level memory model is an approximation of the behavior of

[4]Conversely, this difference will be almost negligible in an (electronic) memory, such as the DRAM or the modern Solid-State disks, where the distribution of the memory accesses does not significantly impact onto the throughput of the memory/SSD.

algorithms on real computers, although it results sufficiently good that it has been widely adopted in the literature to evaluate their performance on massive datasets. So that, in order to be as much precise as possible, we will evaluate in these notes our algorithms by specifying not only the number of executed I/Os but also characterizing their *distribution* (random vs contiguous) over the disk.

At this point one could object that given the impressive technological advancements of the last years, the internal-memory size M is so large that most of the working set of an algorithm (roughly speaking, the set of pages it will reference in the near future) can be fit into it, thus reducing significantly the case of an I/O-fault. We will argue that an even small portion of data resident to disk makes the algorithm slower than expected, and so, data organization cannot be neglected even in these extremely favorable situations.

Let us see why, by means of a "back of the envelope" calculation! Assume that the input size $n = (1+\epsilon)M$ is larger than the internal-memory size of a factor $\epsilon > 0$. The question is how much ϵ impacts onto the average cost of an algorithm step, given that it may access a datum located either in internal memory or on disk. To simplify our analysis, without renouncing to a meaningful conclusion, we assume that $p(\epsilon)$ is the probability of an I/O-fault. As an example, if $p(\epsilon) = 1$ then the algorithm always accesses its data on disk (i.e. one of the ϵM items); if $p(\epsilon) = \frac{\epsilon}{1+\epsilon}$ then the algorithm has a fully-random behavior in accessing its input data (since, from above, we can rewrite $\frac{\epsilon}{1+\epsilon} = \frac{\epsilon M}{(1+\epsilon)M} = \frac{\epsilon M}{n}$); finally, if $p(\epsilon) = 0$ then the algorithm has a working set smaller than the internal memory size, and thus it does not execute any I/Os. Overall $p(\epsilon)$ measures the *un-locality* of the memory references of the analyzed algorithm.

To complete the notation, let us indicate with c the time needed for one I/O wrt one internal-memory access— we have $c \approx 10^5 - 10^6$, see above— and we set a to be the fraction of steps that induce a memory access in the running algorithm (this is typically 30% – 40%, according to [6]). Now we are ready to estimate the *average cost of the step* for an algorithm working in this scenario:

$$1 \times \mathcal{P}(\text{computation step}) + t_m \times \mathcal{P}(\text{memory-access step}),$$

where t_m is the average cost of a memory access. To compute t_m we have to distinguish two cases: an in-memory access (occurring with probability $1 - p(\epsilon)$) or a disk access (occurring with probability $p(\epsilon)$). So we have $t_m = 1 \times (1 - p(\epsilon)) + c \times p(\epsilon)$. Observing that $\mathcal{P}(\text{memory-access step}) + \mathcal{P}(\text{computation step}) = 1$, and plugging the fraction a of memory accesses into $\mathcal{P}(\text{memory-access step})$, we derive the final formula:

$$(1-a) \times 1 + a \times [1 \times (1 - p(\epsilon)) + c \times p(\epsilon)] = 1 + a \times (c-1) \times p(\epsilon) \geq 3 \times 10^4 \times p(\epsilon).$$

This formula clearly shows that, even for algorithms exploiting locality of references (i.e. a small $p(\epsilon)$), the slowdown may be significant and actually it turns out to be four order of magnitudes larger than what might be expected (i.e. $p(\epsilon)$). Just as an example, take an algorithm that exploits locality of references in its memory accesses, say 1 out of 1000 memory accesses is on disk (i.e. $p(\epsilon) = 0.001$). Then, its performance on a massive dataset that is stored on disk would be slowed down by a factor > 30 with respect to a computation executed completely in internal memory.

It goes without saying that this is just the tip of the iceberg, because the larger is the amount of data to be processed by an algorithm, the higher is the number of memory levels involved in the storage of these data and, hence, the more variegate are the types of "memory faults" (say cache-faults, memory-faults, etc.) to cope with for achieving efficiency. The overall message is that neglecting questions pertaining to the cost of memory references in a hierarchical-memory system may *prevent* the use of an algorithm on large input data.

Motivated by these premises, these notes will provide few examples of challenging problems which admit elegant algorithmic solutions whose efficiency is crucial to manage the large datasets that occur in many real-world applications. Algorithm design will be accompanied by several comments on the difficulties that underlie the *engineering* of those algorithms: how to turn a "theoretically efficient" algorithm into a "practically efficient" code. Too many times, as a theoretician, I got the observation that "your algorithm is far from being amenable to an efficient implementation!". By following the recent surge of investigations in *Algorithm Engineering* [10] (to be not confused with the "practice of Algorithms"), we will also dig into the deep computational features of some algorithms by resorting few other successful models of computations— mainly the streaming model [9] and the cache-oblivious model [5]. These models will allow us to capture and highlight some interesting issues of the underlying computation: such as disk passes (streaming model), and universal scalability (cache-oblivious model). We will try our best to describe all these issues in their simplest terms but, nonetheless to say, we will be unsuccessful in turning this "rocket science for non-boffins" into a "science for dummies" [2]. In fact lots of many more things have to fall into place for algorithms to work: top-IT companies (like Google, Yahoo, Microsoft, IBM, Oracle, Facebook, Twitter, etc.) are perfectly aware of the difficulty to find people with the right skills for developing and refining "good" algorithms. This booklet will scratch just the surface of Algorithm Design and Engineering, with the main goal of spurring inspiration into your daily job as software designer or engineer.

References

[1] Person of the Year. *Time Magazine*, 168(27–28), December 2006.

[2] Business by numbers. *The Economist*, September 2007.

[3] Wikipedia's entry: "Algorithm characterizations", 2009. At `http://en.wikipedia.org/wiki/Algorithm_characterizations`

[4] Declan Butler. *2020 computing: Everything, everywhere*, volume 440, chapter 3, pages 402–405. Nature Publishing Group, March 2006.

[5] Rolf Fagerberg. Cache-oblivious model. In Ming-Yang Kao, editor, *Encyclopedia of Algorithms*. Springer, 2008.

[6] John L. Hennessy and David A. Patterson. *Computer Architecture, Fourth Edition: A Quantitative Approach*. Morgan Kaufmann, September 2006.

[7] Donald Knuth. *The Art of Computer Programming: Fundamental Algorithms*, volume 1. Addison-Wesley, 1973.

[8] Fabrizio Luccio. *La struttura degli algoritmi*. Boringhieri, 1982.

[9] S. Muthukrishnan. Data streams: Algorithms and applications. *Foundations and Trends in Theoretical Computer Science*, 1(2), 2005.

[10] Peter Sanders. Algorithm engineering - an attempt at a definition. In Susanne Albers, Helmut Alt, and Stefan Näher, editors, *Efficient Algorithms, Essays Dedicated to Kurt Mehlhorn on the Occasion of His 60th Birthday*, volume 5760 of *Lecture Notes in Computer Science*, pages 321–340. Springer, 2009.

[11] Jeffrey S. Vitter. External memory algorithms and data structures. *ACM Computing Surveys*, 33(2):209–271, 2001.

2
A warm-up!

"Everything should be made as simple as possible, but not simpler."
Albert Einstein

Let us consider the following problem, surprisingly simple in its statement but not that much for what concerns the design of its optimal solution.

Problem. *We are given the performance of a stock at* NYSE *expressed as a sequence of* day-by-day differences *of its quotations. We wish to determine the best buy-&-sell strategy for that stock, namely the pair of days $\langle b, s\rangle$ that would have maximized our revenues if we would have bought the stock at (the beginning of) day b and sold it at (the end of) day s.*

The specialty of this problem is that it has a simple formulation, which finds many other useful variations and applications. We will comment on them at the end of this lecture, now we content ourselves by mentioning that we are interested in this problem because it admits a sequence of algorithmic solutions of increasing sophistication and elegance, which imply a significant reduction in their time complexity. The ultimate result will be a linear-time algorithm, i.e. linear in the number n of stock quotations. This algorithm is *optimal* in terms of the number of executed steps, because all day-by-day differences must be looked at in order to determine if they must be included or not in the optimal solution, actually, one single difference could provide a one-day period worth of investment! Surprisingly, the optimal algorithm will exhibit the simplest pattern of memory accesses— it will execute a single scan of the available stock quotations— and thus it will offer a *streaming behavior*, particularly useful in a scenario in which the granularity of the buy-&-sell actions is not restricted to full-days and we must possibly compute the optimal time-window *on-the-fly* as quotations oscillate. More than this, as we commented in the previous lecture, this algorithmic scheme is optimal in terms of I/Os and *uniformly* over all levels of the memory hierarchy. In fact, because of its streaming behavior, it will execute n/B I/Os independently of the disk-page size B, which may be thus unknown to the underlying algorithm. This is the typical feature of the so called *cache-oblivious algorithms* [4], which we will therefore introduce at the right point of this lecture.

This lecture will be the prototype of what you will find in the next pages: a simple problem to state, with few elegant solutions and challenging techniques to teach and learn, together with several intriguing extensions that can be posed as exercises to the students or as puzzles to tempt your mathematical skills!

Let us now dig into the technicalities, and consider the following example. Take the case of 11 days of exchange for a given stock, and assume that $D[1, 11] = [+4, -6, +3, +1, +3, -2, +3, -4, +1,$

$-9, +6]$ denotes the day-by-day differences of quotations of that stock. It is not difficult to convince yourself that the gain of buying the stock at day x and selling it at day y is equal to the sum of the values in the sub-array $D[x, y]$, namely the sum of all its fluctuations. As an example, take $x = 1$ and $y = 2$, the gain is $+4 - 6 = -2$, and indeed we would loose 2 dollars in buying the morning of the first day and selling the stock at the end of the second day. Notice that the starting value of the stock is not crucial for determining the best time-interval of our investment, what is important are its variations. In other words, the problem stated above boils down to determine the sub-array of $D[1, n]$ which maximizes the sum of its elements. In the literature this problem is indeed known as the *maximum sub-array sum* problem.

Problem Abstraction. *Given an array $D[1, n]$ of positive and negative numbers, we want to find the sub-array $D[b, s]$ which maximizes the sum of its elements.*

It is clear that if all numbers are positive, then the optimal sub-array is the entire D: this is the case of an always increasing stock price, and indeed there is no reason to sell it before the last day! Conversely, if all numbers are negative, then we can pick the one-element window containing the largest (negative) value: if you are imposed to buy this poor stock, then do it in the day it looses the smallest value and sell it soon! In all other cases, it is not at all clear where the optimum sub-array is located. In the example, the optimum spans $D[3, 7] = [+3, +1, +3, -2, +3]$ and has gain $+8$ dollars. This shows that the optimum neither includes the best exploit of the stock (i.e. +6) nor it consists of positive values only. The *structure* of the optimum sub-array is not simple but, surprisingly enough, not very complicated as we will show in Section 2.3.

2.1 A cubic-time algorithm

We start by considering an inefficient solution which translates in pseudo-code the formulation of the problem given above. This algorithm is detailed in Figure 2.1, where the pair of variables $<b_o, s_o>$ identifies the current sub-array of maximum sum, whose value is stored in `MaxSum`. Initially `MaxSum` is set to the dummy value $-\infty$, so that it is immediately changed whenever the algorithm executes Step 8 for the first time. The core of the algorithm examines all possible sub-arrays $D[b, s]$ (Steps 2-3) computing for each of them the sum of their elements (Steps 4-7). If a sum larger than the current maximal value is found (Steps 8-9), then `TmpSum` and its corresponding sub-array are stored in `MaxSum` and $<b_o, s_o>$, respectively.

Algorithm 2.1 The cubic-time algorithm

```
1:  $MaxSum = -\infty$
2:  for ($b = 1$; $b \leq n$; $b{+}{+}$) do
3:      for ($s = b$; $s \leq n$; $s{+}{+}$) do
4:          $TmpSum = 0$
5:          for ($i = b$; $i \leq s$; $i{+}{+}$) do
6:              $TmpSum += D[i]$;
7:          end for
8:          if ($MaxSum < TmpSum$) then
9:              $MaxSum = TmpSum$; $b_o = b$; $s_o = s$;
10:         end if
11:     end for
12: end for
13: return $\langle MaxSum, b_o, s_o \rangle$;
```

The correctness of the algorithm is immediate, since it checks all possible sub-arrays of $D[1, n]$ and selects the one whose sum of its elements is the largest (Step 8). The time complexity is cubic, i.e. $\Theta(n^3)$, and can be evaluated as follows. Clearly the time complexity is upper bounded by $O(n^3)$ because we can form no more than $\frac{n^2}{2}$ pairs `<b,s>` out of n elements,[1] and n is an upper-bound to the cost of computing the sum of each sub-array. Let us now show that the time cost is also $\Omega(n^3)$, so concluding that the time complexity is strictly cubic. To show this lower bound, we observe that $D[1, n]$ contains $(n - L + 1)$ sub-arrays of length L, and thus the cost of computing the sum for all of their elements is $(n - L + 1) \times L$. Summing over all values of L, we would get the exact time complexity. But here we are interested in a lower bound, so we can evaluate that cost just for the subset of sub-arrays whose length L is in the range $[n/4, n/2]$. For each such L, we have that $n - L + 1 > n/2$ and $L \geq n/4$, so the cost above is $(n - L + 1) \times L > n^2/8$. Since we have $\frac{n}{2} - \frac{n}{4} + 1 > n/4$ of those Ls, the total cost for analysing that subset of sub-arrays is lower bounded by $n^3/32 = \Omega(n^3)$.

It is natural now to ask ourselves how much fast in practice is the designed algorithm. We implemented it in Java and tested on a commodity PC. As n grows, its time performance reflects in practice its cubic time complexity, evaluated in the RAM model. More precisely, it takes about 20 seconds to solve the problem for $n = 10^3$ elements, and about 30 hours for $n = 10^5$ elements. Too much indeed if we wish to scale to very large sequences (of quotations), as we are aiming for in these lectures.

2.2 A quadratic-time algorithm

The key inefficiency of the cubic-time algorithm resides in the execution of Steps 4-7 of the pseudo-code in Figure 2.1. These steps re-compute from scratch the sum of the sub-array $D[b, s]$ each time its extremes change in Steps 2-3. But if we look carefully at the **for**-cycle of Step 3 we notice that the size s is incremented by one unit at a time from the value b (one element sub-array) to the value n (the longest possible sub-array that starts at b). Therefore, from one execution to the next one of Step 3, the sub-array to be summed changes from $D[b, s]$ to $D[b, s + 1]$. It is thus immediate to conclude that the new sum for $D[b, s + 1]$ does not need to be recomputed from scratch, but can be computed *incrementally* by just adding the value of the new element $D[s + 1]$ to the current value of `TmpSum` (which inductively stores the sum of $D[b, s]$). This is exactly what the pseudo-code of Figure 2.2 implements: its two main changes with respect to the cubic algorithm of Figure 2.1 are in Step 3, that nulls `TmpSum` every time b is changed (because the sub-array starts again from length 1, namely $D[b, b]$), and in Step 5, that implements the incremental update of the current sum as commented above. Such small changes are worth of a saving of $\Theta(n)$ additions per execution of Step 2, thus turning the new algorithm to have quadratic-time complexity, namely $\Theta(n^2)$.

More precisely, let us concentrate on counting the number of additions executed by the algorithm of Figure 2.2; this is the prominent operation of this algorithm so that its evaluation will give us an estimate of its total number of steps. This number is[2]

$$\sum_{b=1}^{n}(1 + \sum_{s=b}^{n} 1) = \sum_{b=1}^{n}(1 + (n - b + 1)) = n \times (n + 2) - \sum_{b=1}^{n} b = n^2 + 2n - \frac{n(n-1)}{2} = O(n^2).$$

This improvement is effective also in practice. Take the same experimental scenario of the previous section, this new algorithm requires less than 1 second to solve the problem for $n = 10^3$

[1]For each pair $< b, s >$, with $b \leq s$, $D[b, s]$ is a possible sub-array, but $D[s, b]$ is not.

[2]We use below the famous formula, discovered by the young Gauss, to compute the sum of the first n integers.

Algorithm 2.2 The quadratic-time algorithm

```
1:  $MaxSum = -\infty$;
2:  for ($b = 1$; $b \leq n$; $b$++) do
3:      $TmpSum = 0$;
4:      for ($s = b$; $s \leq n$; $s$++) do
5:          $TmpSum$ += $D[s]$;
6:          if ($MaxSum < TmpSum$) then
7:              $MaxSum = TmpSum$; $b_o = b$; $s_o = s$;
8:          end if
9:      end for
10: end for
11: return $\langle MaxSum, b_o, s_o \rangle$;
```

elements, and about 28 minutes to manage 10^6 elements. This means that the new algorithm is able to manage more elements in "reasonable" time. Clearly, these timings and these numbers could change if we use a different programming language (Java, in the present example), operating system (Windows, in our example), and processor (the old Pentium IV, in our example). Nevertheless we believe that they are interesting anyway because they provide a concrete picture of what it does mean a theoretical improvement like the one we showed in the above paragraphs on a real situation. It goes without saying that the life of a coder is typically not so easy because theoretically-good algorithms many times hide so many details that their engineering is difficult and big-O notation often turn out to be not much "realistic". Do not worry, we will have time in these lectures to look at these issues in more detail.

2.3 A linear-time algorithm

The final step of this lecture is to show that the maximum sub-array sum problem admits an elegant algorithm that processes the elements of $D[1, n]$ is a streaming fashion and takes the *optimal* $O(n)$ time. We could not aim for more!

To design this algorithm we need to dig into the structural properties of the optimal sub-array. For the purpose of clarity, we refer the reader to Figure 2.1 below, where the optimal sub-array is assumed to be located at two positions $b_o \leq s_o$ in the range $[1, n]$.

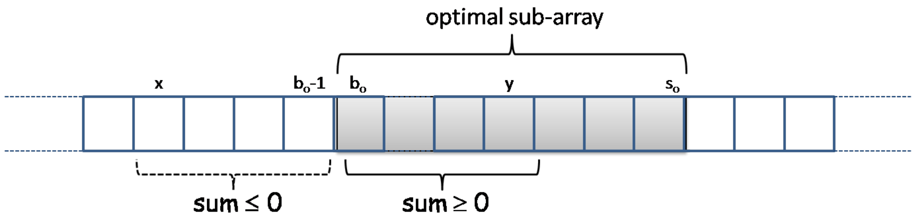


FIGURE 2.1: An illustrative example of Properties 1 and 2.

Let us now take a sub-array that starts before b_o and ends at position $b_o - 1$, say $D[x, b_o - 1]$. The sum of the elements in this sub-array cannot be positive because, otherwise, we could merge it with the (adjacent) optimal sub-array and thus get the longer sub-array $D[x, s_o]$ having sum *larger than* the one obtained with the (claimed) optimal $D[b_o, s_o]$. So we can state the following:

Property 1. The sum of the elements in a sub-array $D[x, b_o - 1]$, with $x < b_o$, cannot be (strictly) positive.

Via a similar argument, we can consider a sub-array that is a prefix of the optimal $D[b_o, s_o]$, say $D[b_o, y]$ with $y \leq s_o$. This sub-array cannot have negative sum because, otherwise, we could drop it from the optimal solution and get a shorter array, namely $D[y + 1, s_o]$ having sum larger than the one obtained by the (claimed) optimal $D[b_o, s_o]$. So we can state the following other property:

Property 2. The sum of the elements in a sub-array $D[b_o, y]$, with $y \leq s_o$, cannot be (strictly) negative.

We remark that any one of the sub-arrays considered in the above two properties might have sum equal to zero. This would not affect the optimality of $D[b_o, s_o]$, it could only introduce other optimal solutions being either longer or shorter than $D[b_o, s_o]$.

Let us illustrate these two properties on the array $D[1, 11] = [+4, -6, +3, +1, +3, -2, +3, -4, +1, -9, +6]$. Here the optimum sub-array is $D[3, 7] = [+3, +1, +3, -2, +3]$. We note that $D[x, 2]$ is always negative (Prop. 1), in fact for $x = 1$ the sum is $+4 - 6 = -2$ and for $x = 2$ the sum is -6. On the other hand the sum of all elements in $D[3, y]$ is positive for all prefixes of the optimum sub-array (Prop. 2), namely $y \leq 7$. We also point out that the sum of $D[3, y]$ is positive even for some $y > 7$, take for example $D[3, 8]$ for which the sum is 4 and $D[3, 9]$ for which the sum is 5. Of course, this does not contradict Prop. 2.

Algorithm 2.3 The linear-time algorithm

```
1:  $MaxSum = -\infty$
2:  $TmpSum = 0$; $b = 1$;
3:  for ($s = 1$; $s \leq n$; $s$++) do
4:      $TmpSum$ += $D[s]$;
5:      if ($MaxSum < TmpSum$) then
6:          $MaxSum = TmpSum$; $b_o = b$; $s_o = s$;
7:      end if
8:      if ($TmpSum < 0$) then
9:          $TmpSum = 0$; $b = s + 1$;
10:     end if
11: end for
12: return $\langle MaxSum, b_o, s_o \rangle$;
```

The two properties above lead to the simple Algorithm 2.3. It consists of one unique **for**-cycle (Step 3) which keeps in `TmpSum` the sum of a sub-array ending in the currently examined position s and starting at some position $b \leq s$. At any step of the **for**-cycle, the candidate sub-array is extended one position to the right (i.e. s++), and its sum `TmpSum` is increased by the value of the current element $D[s]$ (Step 4). Since the current sub-array is a candidate to be the optimal one, its sum is compared with the current optimal value (Step 5). Then, according to Prop. 1, if the sub-array sum is negative, the current sub-array is discarded and the process “restarts” with a new sub-array beginning at the next position $s + 1$ (Steps 8-9). Otherwise, the current sub-array is extended to the right, by incrementing s. The tricky issue here is to show that the optimal sub-array is checked in Step 5, and thus stored in $< b_o, s_o >$. This is not intuitive at all because the algorithm is checking n sub-arrays out of the $\Theta(n^2)$ possible ones, and we want to show that this (minimal) subset of candidates actually contains the optimal solution. This subset is *minimal* because these sub-arrays form a *partition* of $D[1,n]$ so that every element belongs to one, and only one checked sub-array. Moreover, since every element must be analyzed, we cannot discard any sub-array of this partition without checking its sum!

Before digging into the formal proof of correctness, let us follow the execution of the algorithm over the array $D[1,11] = [+4,-6,+3,+1,+3,-2,+3,-4,+1,-9,+6]$. Remember that the optimum sub-array is $D[3,7] = [+3,+1,+3,-2,+3]$. We note that $D[x,2]$ is negative for $x = 1,2$, so the algorithm surely zeroes the variable TmpSum when $s = 2$ in Steps 8-9. At that time, b is set to 3 and TmpSum is set to 0. The subsequent scanning of the elements $s = 3,\ldots,7$ will add their values to TmpSum which is always positive (see above). When $s = 7$, the examined sub-array coincides with the optimal one, we thus have TmpSum = 8, so Step 5 stores the optimum location in $< b_o, s_o >$. It is interesting to notice that, in this example, the algorithm does not re-start the value of TmpSum at the next position $s = 8$ because it is still positive (namely, TmpSum = 4); this means that the algorithm will examine sub-arrays longer than the optimal one, but all having a smaller sum, of course. The next re-starting will occur at position $s = 10$ where TmpSum = -4.

It is easy to realize that the time complexity of the algorithm is $O(n)$ because every element is examined just once. More tricky is to show that the algorithm is correct, which actually means that Steps 4 and 5 eventually compute and check the optimal sub-array sum. To show this, it suffices to prove the following two facts: (i) when $s = b_o - 1$, Step 8 resets b to b_o; (ii) for all subsequent positions $s = b_o,\ldots,s_o$, Step 8 never resets b so that it will eventually compute in TmpSum the sum of all elements in $D[b_o, s_o]$, whenever $s = s_o$. It is not difficult to see that Fact (i) derives from Property 1, and Fact (ii) from Property 2.

This algorithm is very fast in the same experimental scenario mentioned in the previous sections, it takes less than 1 second to process millions of quotations. A truly scalable algorithm, indeed, with many nice features that make it appealing also in a hierarchical-memory setting. In fact, this algorithm scans the array D from left to right and examines each of its elements just once. If D is stored on disk, these elements are fetched in internal memory one page at a time. Hence the algorithm executes n/B I/Os, which is *optimal*. It is interesting also to note that the design of the algorithm does not depend on B (which indeed does not appear in the pseudo-code), but we can evaluate its I/O-complexity in terms of B. Hence the algorithm takes n/B optimal I/Os independently of the the page size B, and thus subtly on the hierarchical-memory levels interested by the algorithm execution. Decoupling the use of the parameter B between algorithm design and algorithm analysis is the key issue of the so called *cache-oblivious algorithms*, which are a hot topic of algorithmic investigation nowadays. This feature is achieved here in a basic (trivial) way by just adopting a scan-based approach. The literature [4] offers more sophisticated results regarding the design of cache-oblivious algorithms and data structures.

2.4 Another linear-time algorithm

There exists another optimal solution to the maximum sub-array sum problem which hinges on a different algorithm design. For simplicity of exposition, let us denote by $Sum_D[y', y'']$ the sum of the elements in the sub-array $D[y', y'']$. Take now a selling time s and consider all sub-arrays that end at s: namely we are interested in sub-arrays having the form $D[x, s]$, with $x \leq s$. The value $Sum_D[x, s]$ can be expressed as the difference between $Sum_D[1, s]$ and $Sum_D[1, x-1]$. Both of these sums are indeed *prefix*-sums over the array D and can be computed in linear time. As a result, we can rephrase our maximization problem as follows:

$$\max_s \max_{b \leq s} \mathrm{Sum}_D[b, s] = \max_s \max_{b \leq s} (\mathrm{Sum}_D[1, s] - \mathrm{Sum}_D[1, b-1]).$$

We notice that if $b = 1$ the second term refers to the empty sub-array $D[1, 0]$; so we can assume that $Sum_D[1, 0] = 0$. This is the case in which $D[1, s]$ is the sub-array of maximum sum among all the sub-arrays ending at s (so no prefix sub-array $D[1, b-1]$ is dropped from it).

The next step is to pre-compute all prefix sums $P[i] = Sum_D[1, i]$ in $O(n)$ time and $O(n)$ space via a scan of the array D: Just notice that $P[i] = P[i-1] + D[i]$, where we set $P[0] = 0$ in order to manage the special case above. Hence we can rewrite the maximization problem in terms of the array P, rather than Sum_D: $\max_{b \le s}(P[s] - P[b-1])$. The cute observation now is that we can decompose the max-computation into a max-min calculation over the two variables b and s

$$\max_s \max_{b \le s}(P[s] - P[b-1]) = \max_s(P[s] - \min_{b \le s} P[b-1]).$$

The key idea is that we can move $P[s]$ outside the inner max-calculation because it does not depend on the variable b, and then change a max into a min because of the negative sign. The final step is then to pre-compute the minimum $\min_{b \le s} P[b-1]$ for all positions s, and store it in an array $M[0, n-1]$. We notice that, also in this case, the computation of $M[i]$ can be performed via a single scan of P in $O(n)$ time and space: set $M[0] = 0$ and then derive $M[i]$ as $\min\{M[i-1], P[i]\}$. Given M, we can rewrite the formula above as $\max_s(P[s] - M[s-1])$ which can be clearly computed in $O(n)$ time given the two arrays P and M. Overall this new approach takes $O(n)$ time and $O(n)$ extra space.

As an illustrative example, consider again the array $D[1, 11] = [+4, -6, +3, +1, +3, -2, +3, -4, +1, -9, +6]$. We have that $P[0, 11] = [0, +4, -2, +1, +2, +5, +3, +6, +2, +3, -6, 0]$ and $M[0, 10] = [0, 0, -2, -2, -2, -2, -2, -2, -2, -2, -6]$. If we compute the difference $P[s] - M[s-1]$ for all $s = 1, \ldots, n$, we obtain the sequence of values $[+4, -2, +3, +4, +7, +5, +8, +4, +5, -4, +6]$, whose maximum (sum) is $+8$ that occurs (correctly) at the (final) position $s = 7$. It is interesting to note that the left-extreme b_o of the optimal sub-array could be derived by finding the position $b_o - 1$ where $P[b_o - 1]$ is minimum: in the example, $P[2] = -2$ and thus $b_o = 3$.

Algorithm 2.4 Another linear-time algorithm

```
1:  $MaxSum = -\infty$; $b_o = 1$;
2:  $TmpSum = 0$; $MinTmpSum = 0$;
3:  for ($s = 1$; $s \le n$; $s$++) do
4:      $TmpSum$ += $D[s]$;
5:      if ($MaxSum < TmpSum - MinTmpSum$) then
6:          $MaxSum = TmpSum - MinTmpSum$; $s_o = s$; $b_o = b_{tmp}$;
7:      end if
8:      if ($TmpSum < MinTmpSum$) then
9:          $MinTmpSum = TmpSum$; $b_{tmp} = s + 1$;
10:     end if
11: end for
12: return $\langle MaxSum, b_o, s_o \rangle$;
```

We conclude this section by noticing that the proposed algorithm executes three passes over the array D, rather than the single pass of Algorithm 2.3. It is not difficult to turn this algorithm to make *one-pass* too. It suffices to deploy the associativity of the min/max functions, and use two variables that inductively keep the values of $P[s]$ and $M[s-1]$ as the array D is scanned from left to right. Algorithm 2.4 implements this idea by using the variable `TmpSum` to store $P[s]$ and the variable `MinTmpSum` to store $M[s-1]$. This way the formula $\max_s(P[s] - M[s-1])$ is evaluated incrementally for $s = 1, \ldots, n$, thus avoiding the two passes for pre-calculating the arrays P and

M and the extra-space needed to store them. One pass over D is then enough, and so we have re-established the nice algorithmic properties of Algorithm 2.3 but with a completely different design!

2.5 Few interesting variants$^\infty$

As we promised at the beginning of this lecture, we discuss now few interesting variants of the maximum sub-array sum problem. For further algorithmic details and formulations we refer the interested reader to [1, 2]. Note that this is a challenging section, because it proposes an algorithm whose design and analysis are sophisticated!

Sometimes in the bio-informatics literature the term "sub-array" is substituted by "segment", and the problem takes the name of "maximum-sum segment problem". In the bio-context the goal is to identify segments which occur inside DNA sequences (i.e. strings of four letters A, T, G, C) and are *rich* of G or C nucleotides. Biologists believe that these segments are biologically significant since they predominantly contain genes. The mapping from DNA sequences to *arrays of numbers*, and thus to our problem abstraction, can be obtained in several ways depending on the objective function that models the *GC-richness* of a segment. Two interesting mappings are the following ones:

- Assign a penalty $-p$ to the nucleotides A and T of the sequence, and a reward $1-p$ to the nucleotides C and G. Given this assignment, the sum of a segment of length l containing x occurrences of C+G is equal to $x - p \times l$. Biologists think that this function is a good measure for the CG-richness of that segment. Interestingly enough, all algorithms described in the previous sections can be used to identify the CG-rich segments of a DNA sequence in linear time, according to this objective function. Often, however, biologists prefer to define a cutoff range on the length of the segments for which the maximum sum must be searched, in order to avoid the reporting of extremely short or extremely long segments. In this new scenario the algorithms of the previous sections cannot be applied, but yet linear-time optimal solutions are known for them (see e.g. [2]).
- Assign a value 0 to the nucleotides A and T of the sequence, and a value 1 to the nucleotides C and G. Given this assignment, the density of C+G nucleotides in a segment of length l containing x occurrences of C and G is x/l. Clearly $0 \leq x/l \leq 1$ and every single occurrence of a nucleotide C or G provides a segment with maximum density 1. Biologists consider this as an interesting measure of CG-richness for a segment, provided that a cutoff range on the length of the searched segments is imposed. This problem is more difficult than the one stated in the previous item, nevertheless it posses optimal (quasi-)linear time solutions which are much sophisticated and for which we refer the interested reader to the pertinent bibliography (e.g. [1, 3, 5]).

These examples are useful to highlight a *dangerous trap* that often occurs when abstracting a real problem: apparently small changes in the problem formulation lead to big jumps in the complexity of designing efficient algorithms for them. Think for example to the density function above, we needed to introduce a cutoff lower-bound to the segment length in order to avoid the trivial solution consisting of *single* nucleotides C or G! With this "small" change, the problem results more challenging and its solutions sophisticated.

Other subtle traps are more difficult to be discovered. Assume that we decide to circumvent the single-nucleotide outcome by searching for the the *longest* segment whose density is *larger than* a fixed value d. This is, in some sense, a complementary formulation of the problem stated in the second item above, because maximization is here on the segment length and a (lower) cut-off is imposed on the density value. Surprisingly it is possible to *reduce* this density-based problem to a sum-based problem, in the spirit of the one stated in the first item above, and solved in the previous sections. Algorithmic reductions are often employed by researchers to re-use known solutions and thus do not re-discover again and again the "hot water". To prove this reduction it is enough to notice that:

$$\frac{\mathrm{Sum}_D[x,y]}{y-x+1} = \sum_{k=x}^{y} \frac{D[k]}{y-x+1} \geq t \iff \sum_{k=x}^{y}(D[k]-t) \geq 0.$$

Therefore, subtracting to all elements in D the density-threshold t, we can turn the problem stated in the second item above into the one that asks for the *longest segment that has sum larger or equal than* 0. Be careful that if you change the request from the *longest segment* to the *shortest one* whose density is larger than a threshold t, then the problem becomes trivial again: Just take the single occurrence of a nucleotide C or G. Similarly, if we fix an upper bound S to the segment's sum (instead of a lower bound), then we can change the sign to all D's elements and thus turn the problem again into a problem with a lower bound $t = -S$. So let us stick on the following general formulation:

Problem. *Given an array $D[1,n]$ of positive and negative numbers, we want to find the* longest *segment in D whose sum of its elements is* larger or equal than *a fixed threshold t.*

We notice that this formulation is in some sense a complement of the one given in the first item above. Here we maximize the segment length and pose a lower-bound to the sum of its elements; there, we maximized the sum of the segment provided that its length was within a given range. It is nice to observe that the structure of the algorithmic solution for both problems is similar, so we detail only the former one and refer the reader to the literature for the latter.

The algorithm proceeds inductively by assuming that, at step $i = 1, 2, \ldots, n$, it has computed the longest sub-array having sum larger than t and occurring within $D[1, i-1]$. Let us denote the solution available at the beginning of step i with $D[l_{i-1}, r_{i-1}]$. Initially we have $i = 1$ and thus the inductive solution is the empty one, hence having length equal to 0. To move from step i to step $i+1$, we need to compute $D[l_i, r_i]$ by possibly taking advantage of the currently known solution.

It is clear that the new segment is either inside $D[1, i-1]$ (namely $r_i < i$) or it ends at position $D[i]$ (namely $r_i = i$). The former case admits as solution the one of the previous iteration, namely $D[l_{i-1}, r_{i-1}]$, and so nothing has to be done: just set $r_i = r_{i-1}$ and $l_i = l_{i-1}$. The latter case is more involved and requires the use of some special data structures and a tricky analysis to show that the total complexity of the solution proposed is $O(n)$ in space and time, thus turning to be optimal!

We start by making a simple, yet effective, observation:

FACT 2.1

If $r_i = i$ then the segment $D[l_i, r_i]$ must be strictly longer than the segment $D[l_{i-1}, r_{i-1}]$. This means in particular that l_i occurs to the left of position $L_i = i - (r_{i-1} - l_{i-1})$.

The proof of this fact follows immediately by the observation that, if $r_i = i$, then the current step i has found a segment that improves the previously known one. Here "improved" means "longer" because the other constraint imposed by the problem formulation is boolean since it refers to a lower-bound on the segment's sum. This is the reason why we can discard all positions within the range $[L_i, i]$, in fact they originate intervals of length shorter or equal than the previous solution $D[l_{i-1}, r_{i-1}]$.

Reformulated Problem. *Given an array $D[1,n]$ of positive and negative numbers, we want to find at every step the* smallest *index $l_i \in [1, L_i)$ such that $\mathrm{Sum}_D[l_i, i] \geq t$.*

We point out that there could be many such indexes l_i, here we wish to find the *smallest* one because we aim at determining the *longest* segment.

At this point it is useful to recall that $\text{Sum}_D[l_i, i]$ can be re-written in terms of prefix-sums of array D, namely $\text{Sum}_D[1, i] - \text{Sum}_D[1, l_i - 1] = P[i] - P[l_i - 1]$ where the array P was introduced in Section 2.4. So we need to find the smallest index $l_i \in [1, L_i)$ such that $P[i] - P[l_i - 1] \geq t$. The array P can be pre-computed in linear time and space.

It is worth to observe that the computation of l_i could be done by scanning $P[1, L_i - 1]$ and searching for the *leftmost* index x such that $P[i] - P[x] \geq t$. We could then set $l_i = x + 1$ and have been done. Unfortunately, this is inefficient because it leads to scan over and over again the same positions of P as i increases, thus leading to a quadratic-time algorithm! Since we aim for a linear-time algorithm, we need to spend constant time "on average" per step i. We used the quotes because there is no *stochastic* argument here to compute the average, we wish only to capture syntactically the idea that, since we want to spend $O(n)$ time in total, our algorithm has to take constant time *amortized* per steps. In order to achieve this performance we first need to show that we can avoid the scanning of the whole prefix $P[1, L_i - 1]$ by identifying a *subset* of *candidate positions* for x. Call $C_{i,j}$ the candidate positions for iteration i, where $j = 0, 1, \ldots$. They are defined as follows: $C_{i,0} = L_i$ (it is a dummy value), and $C_{i,j}$ is defined inductively as the *leftmost minimum* of the sub-array $P[1, C_{i,j-1} - 1]$ (i.e. the sub-array to the left of the current minimum and/or to the left of L_i). We denote by $c(i)$ the number of these candidate positions for the step i, where clearly $c(i) \leq L_i$ (equality holds when $P[1, L_i]$ is decreasing).

For an illustrative example look at Figure 2.2, where $c(i) = 3$ and the candidate positions are connected via leftward arrows.

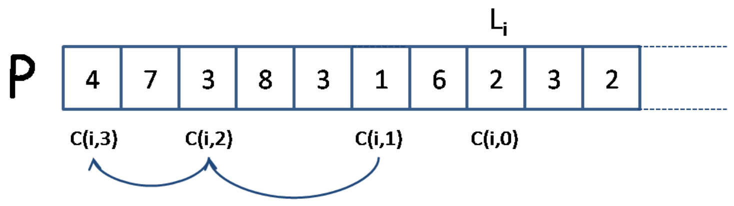


FIGURE 2.2: An illustrative example for the candidate positions $C_{i,j}$, given an array P of prefix sums. The picture is generic and reports only L_i for simplicity.

Looking at Figure 2.2 we derive three key properties whose proof is left to the reader because it immediately comes from the definition of $C_{i,j}$:

Property a. The sequence of candidate positions $C_{i,j}$ occurs within $[1, L_i)$ and moves leftward, namely $C_{i,j} < C_{i,j-1} < \ldots < C_{i,1} < C_{i,0} = L_i$.

Property b. At each iteration i, the sequence of candidate values $P[C_{i,j}]$ is increasing with $j = 1, 2, \ldots, c(i)$. More precisely, we have $P[C_{i,j}] > P[C_{i,j-1}] > \ldots > P[C_{i,1}]$ where the indices move leftward according to Property (a).

Property c. The value $P[C_{i,j}]$ is smaller than any other value on its left in P, because it is the leftmost minimum of the prefix $P[1, C_{i,j-1} - 1]$.

It is crucial now to show that the index we are searching for, namely l_i, can be derived by looking only at these candidate positions. In particular we can prove the following:

FACT 2.2

At each iteration i, the largest index j^* such that $Sum_D[C_{i,j^*}+1, i] \geq t$ (if any) provides us with the longest segment we are searching for.

By Fact 2.1 we are interested in segments having the form $D[l_i, i]$ with $l_i < L_i$, and by properties of prefix-sums, we know that $\mathrm{Sum}_D[C_{i,j}+1, i]$ can be re-written as $P[i] - P[C_{i,j}]$. Given this and Property (c), we can conclude that all segments $D[z, i]$, with $z < C_{i,j}$, have a sum *smaller* than $\mathrm{Sum}_D[C_{i,j}+1, i]$. Consequently, if we find that $\mathrm{Sum}_D[C_{i,j}+1, i] < t$ for some j, then we can discard all positions z to the left of $C_{i,j}+1$ in the search for l_i. Therefore the index j^* characterized in Fact 2.2 is the one giving correctly $l_i = C_{i,j^*} + 1$.

There are two main problems in deploying the candidate positions for the efficient computation of l_i: (1) How do we compute the $C_{i,j}$s as i increases, (2) How do we search for the index j^*. To address issue (1) we notice that the computation of $C_{i,j}$ depends only on the position of the previous $C_{i,j-1}$ and *not* on the indices i or j. So we can define an auxiliary array $LMin[1, n]$ such that $LMin[i]$ is the leftmost position of the minimum within $P[1, i-1]$. It is not difficult to see that $C_{i,1} = LMin[L_i]$, and that according to the definition of C it is $C_{i,2} = LMin[\, LMin[L_i]\,] = LMin^2[L_i]$. In general, it is $C_{i,k} = LMin^k[L_i]$. This allows an incremental computation:

$$LMin[x] = \begin{cases} 0 & \text{if } x = 0 \\ x-1 & \text{if } P[x-1] < P[\, LMin[x-1]\,] \\ LMin[x-1] & \text{otherwise} \end{cases}$$

The formula above has an easy explanation. We know inductively $LMin[x-1]$ as the leftmost minimum in the array $P[1, x-2]$: initially we set $LMin[0]$ to the dummy value 0. To compute $LMin[x]$ we need to determine the leftmost minimum in $P[1, x-1]$: this is located either in $x-1$ (with value $P[x-1]$) or it is the one determined for $P[1, x-2]$ of position $LMin[x-1]$ (with value $P[LMin[x-1]]$). Therefore, by comparing these two values we can compute $LMin[x]$ in constant time. Hence the computation of all candidate positions $LMin[1, n]$ takes $O(n)$ time.

We are left with the problem of determining j^* efficiently. We will not be able to compute j^* in constant time at each iteration i but we will show that, if at step i we execute $s_i > 1$ steps, then we are advancing in the construction of the longest solution. Specifically, we are extending the length of that solution by $\Theta(s_i)$ units. Given that the longest segment cannot be longer than n, the sum of these extra-costs cannot be larger than $O(n)$, and thus we are done! This is called *amortized argument* because we are, in some sense, charging the cost of the expensive iterations to the cheapest ones. The computation of j^* at iteration i requires the check of the positions $LMin^k[L_i]$ for $k = 1, 2, \ldots$ until the condition in Fact 2.2 is satisfied; in fact, we know that all the other $j > j^*$ do not satisfy Fact 2.2. This search takes j^* steps and finds a new segment whose length is *increased* by at least j^* units, given Property (a) above. This means that either an iteration i takes constant time, because the check fails immediately at $LMin[L_i]$ (so the current solution is not better than the one computed at the previous iteration $i-1$), or the iteration takes $O(j^*)$ time but the new segment $D[L_i, r_i]$ has been extended by j^* units. Since a segment cannot be longer than the entire sequence $D[1, n]$, we can conclude that the total extra-time cannot be larger than $O(n)$.

We leave to the diligent reader to work out the details of the pseudo-code of this algorithm, the techniques underlying its elegant design and analysis should be clear enough to approach it without any difficulties.

References

[1] Kun-Mao Chao. Maximum-density segment. In Ming-Yang Kao, editor, *Encyclopedia of Algorithms*. Springer, 2008.

[2] Kun-Mao Chao. Maximum-scoring segment with length restrictions. In Ming-Yang Kao, editor, *Encyclopedia of Algorithms*. Springer, 2008.

[3] Chih-Huai Cheng, Hsiao-Fei Liu, and Kun-Mao Chao. Optimal algorithms for the average-constrained maximum-sum segment problem. *Information Processing Letters*, 109(3):171–174, 2009.

[4] Rolf Fagerberg. Cache-oblivious model. In Ming-Yang Kao, editor, *Encyclopedia of Algorithms*. Springer, 2008.

[5] Takeshi Fukuda, Yasuhiko Morimoto, Shinichi Morishita, and Takeshi Tokuyama. Mining optimized association rules for numeric attributes. *Journal of Computer System Sciences*, 58(1):1–12, 1999.

3 Random Sampling

"So much of life, it seems to me, is determined by pure randomness."
Sidney Poitier

This lecture attacks a simple-to-state problem which is the backbone of many randomized algorithms, and admits solutions which are algorithmically challenging to design and analyze.

Problem. *Given a sequence of items $S = (i_1, i_2, \ldots, i_n)$ and a positive integer $m \leq n$, the goal is to select a subset of m items from S uniformly at random.*

Uniformity here means that any item in S has to be sampled with probability $1/n$. Items can be numbers, strings or general objects either stored in a file located on disk or streaming through a channel. In the former scenario, the input size n is known and items occupy n/B pages, in the latter scenario, the input size may be even unknown yet the *uniformity* of the sampling process must be guaranteed. In this lecture we will address both scenarios aiming at efficiency in terms of I/Os, extra-space required for the computation (in addition to the input), and amount of randomness deployed (expressed as number of randomly generated integers). Hereafter, we will make use of a built-in procedure `Rand(a,b)` that randomly selects a number within the range $[a, b]$. The number, being either real or integer, will be clear from the context. The design of a good `Rand`-function is a challenging task, however we will not go into its details because we wish to concentrate in this lecture on the sampling process rather than on the generation of random numbers; though, the interested reader can refer to the wide literature about (pseudo-)random number generators.

Finally we notice that it is desirable to have the positions of the sampled items in *sorted* order because this speeds up their extraction from S both in the disk setting (less seek time) and in the stream-based setting (less passes over the data). Moreover it reduces the working space because it allows to extract the items efficiently via a scan, rather than using an auxiliary array of pointers to items. We do not want to detail further the sorting issue here, which gets complicated whenever $m > M$ and thus these positions cannot fit into internal memory. In this case we need a disk-based sorter, which is indeed an advanced topic of a subsequent lecture. If instead $m \leq M$ we could deploy the fact that positions are integers in a fixed range and thus use radix sort or any other faster routine available in the literature.

3.1 Disk model and known sequence length

We start by assuming that the input size n is known and that $S[1, n]$ is stored in a file on disk which cannot be modified because it may be the input of a more complicated problem that includes the current one as a sub-task. The first algorithm we propose is very simple, and allows us to arise some issues that will be attacked in the subsequent solutions.

Algorithm 3.1 Drawing from all un-sampled positions

```
1: Initialize the auxiliary array $S'[1, n] = S[1, n]$;
2: for $s = 0, 1, \ldots, m - 1$ do
3:     $p = \mathtt{Rand}(1, n - s)$;
4:     select the item (pointed by) $S'[p]$;
5:     swap $S'[p]$ with $S'[n - s]$.
6: end for
```

At each step s the algorithm maintains the following invariant: *the subarray* $S'[n - s + 1, n]$ *contains the items that have been already sampled, the rest of the items of* S *are contained in* $S'[1, n-s]$. Initially (i.e. $s = 0$) this invariant holds because $S'[n-s+1, n] = S'[n+1, n]$ is empty. At a generic step s, the algorithm selects randomly one item from $S'[1, n - s]$, and replaces it with the last item of that sequence (namely, $S'[n - s]$). This preserves the invariant for $s + 1$. At the end (i.e. $s = m$), the sampled items are contained in $S'[n - m + 1, n]$. We point out that S' cannot be a *pure* copy of S but it must be implemented as an *array of pointers* to S's items. The reason is that these items may have variable length (e.g. strings) so their retrieval in constant time could not be obtained via arithmetic operations, as well as the replacement step might be impossible due to difference in length between the item at $S'[p]$ and the item at $S'[n - s]$. Pointers avoid these issues but occupy $\Theta(n \log n)$ bits of space, which might be a non negligible space when n gets large and might turn out even larger than S if the average length of S's objects is shorter than $\log n$.[1] Another drawback of this approach is given by its pattern of memory accesses, which acts over $O(n)$ cells in purely random way, taking $\Theta(m)$ I/Os. This may be slow when $m \approx n$, so in this case we would like to obtain $O(n/B)$ I/Os which is the cost of scanning the whole S.

Let us attack these issues by proposing a series of algorithms that incrementally improve the I/Os and the space resources of the previous solution, up to the final result that will achieve $O(m)$ extra space, $O(m)$ average time and $O(\min\{m, n/B\})$ I/Os. We start by observing that the swap of the items in Step 5 of `Algorithm 3.1` guarantees that every step generates one distinct item, but forces to duplicate S and need $\Omega(m)$ I/Os whichever is the value of m. The following `Algorithm 3.2` improves the I/O- and space-complexities by avoiding the item-swapping via the use of an auxiliary data structure that keeps track of the selected positions in sorted order and needs only $O(m)$ space.

Algorithm 3.2 Dictionary of sampled positions

```
1: Initialize the dictionary $\mathcal{D} = \emptyset$;
2: while $(|\mathcal{D}| < m)$ do
3:     $p = \mathtt{Rand}(1, n)$;
4:     if $p \notin \mathcal{D}$ insert it;
5: end while
```

[1]This may occur only if S contains duplicate items, otherwise a classic combinatorial argument applies.

`Algorithm 3.2` stops when $\mathcal{D}$ contains m (distinct) integers which constitute the positions of the items to be sampled. According to our observation made at the beginning of the lecture, $\mathcal{D}$ may be sorted before S is accessed on disk to reduce the seek time. In any case, the efficiency of the algorithm mainly depends on the implementation of the dictionary $\mathcal{D}$, which allows to detect the presence of duplicate items. The literature offers many data structures that efficiently support membership and insert operations, based either on hashing or on trees. Here we consider only an hash-based solution which consists of implementing $\mathcal{D}$ via a hash table of size $\Theta(m)$ with collisions managed via chaining and a universal hash function for table access [1]. This way each membership query and insertion operation over $\mathcal{D}$ takes $O(1)$ time on average (the load factor of this table is $O(1)$), and total space $O(m)$. Time complexity could be forced to be worst case by using more sophisticated data structures, such as dynamic perfect hashing, but the final time bounds would always be *in expectation* because of the underlying re-sampling process.

However this algorithm may generate *duplicate* positions, which must be discarded and *re-sampled*. Controlling the cost of the re-sampling process is the main drawback of this approach, but this induces a constant-factor slowdown on average, thus making this solution much appealing in practice. In fact, the probability of having extracted an item already present in $\mathcal{D}$ is $|\mathcal{D}|/n \leq m/n < 1/2$ because, without loss of generality, we can assume that $m < n/2$ otherwise we can solve the *complement* of the current problem and thus randomly select the positions of the items that are *not* sampled from S. So we need an average of $O(1)$ re-samplings in order to obtain a new item for $\mathcal{D}$, and thus advancing in our selection process. Overall we have proved the following:

FACT 3.1 `Algorithm 3.2` *based on hashing with chaining requires $O(m)$ average time and takes $O(m)$ additional space to select uniformly at random m positions in $[1, n]$. The average depends both on the use of hashing and the cost of re-sampling. An additional sorting-cost is needed if we wish to extract the sampled items of S in a streaming-like fashion. In this case the overall sampling process takes $O(\min\{m, n/B\})$ I/Os.*

If we substitute hashing with a (balanced) search tree and assume to work in the RAM model (hence we assume $m < M$), then we can avoid the sorting step by performing an *in-visit* of the search tree in $O(m)$ time. However, `Algorithm 3.2` would still require $O(m \log m)$ time because each insertion/membership operation would take $O(\log m)$ time. We could do better by deploying an *integer*-based dictionary data structure, such as a van Emde-Boas tree, and thus take $O(\log \log n)$ time for each dictionary operation. The two bounds would be incomparable, depending on the relative magnitudes of m and n. Many other trade-offs are possible by changing the underlying dictionary data structure, so we leave to the reader this exercise.

The next step is to avoid a dictionary data structure and use *sorting* as a basic block of our solution. This could be particularly useful in practice because comparison-based sorters, such as `qsort`, are built-in in many programming languages. The following analysis will have also another side-effect which consists of providing a more clean evaluation of the average time performance of `Algorithm 3.2`, rather than just saying re-sample each item at most $O(1)$ times on average.

The cost of `Algorithm 3.3` depends on the number of times the sorting step is repeated and thus do exist duplicates in the sampled items. We argue that a small number of re-samplings is needed. So let us compute the probability that `Algorithm 3.3` executes just one iteration: this means that the m sampled items are all distinct. This analysis is well known in the literature and goes under the name of *birthday problem*: how many people do we need in a room in order to have a probability larger than $1/2$ that at least two of them have the same birthday. In our context we have that people = items and birthday = position in S. By mimicking the analysis done for the birthday problem, we

Algorithm 3.3 Sorting

```
1: $\mathcal{D} = \emptyset$;
2: while ($|\mathcal{D}| < m$) do
3:     $\mathcal{X}$ = randomly draw $m$ positions from $[1, n]$;
4:     Sort $\mathcal{X}$ and eliminate the duplicates;
5:     Set $\mathcal{D}$ as the resulting $\mathcal{X}$;
6: end while
```

can estimate the probability that a duplicate among m randomly-drawn items does not occur as:

$$\frac{m!\binom{n}{m}}{n^m} = \frac{n(n-1)(n-2)\cdots(n-m+1)}{n^m} = 1 \times (1 - \frac{1}{n}) \times (1 - \frac{2}{n}) \times \cdots (1 - \frac{m-1}{n})$$

Given that $e^x \geq 1 + x$, we can upper bound the above formula as:

$$e^0 \times e^{-1/n} \times e^{-2/n} \times \cdots e^{-(m-1)/n} = e^{-(1+2+\cdots+(m-1))/n} = e^{-m(m-1)/2n}$$

So the probability that a duplicate *does not* occur is at most $e^{-m(m-1)/2n}$ and, in the case of the birthday paradox in which $n = 365$, this is slightly smaller than one-half already for $m = 23$. In general we have that $m = \sqrt{n}$ elements suffices to make the probability of a duplicate at least $1 - \frac{1}{\sqrt{e}} \approx 0.4$, thus making our algorithm need re-sampling. On the positive side, we notice that if $m \ll \sqrt{n}$ then e^x can be well approximated with $1 + x$, so $e^{-m(m-1)/2n}$ is not only an upper-bound but also a reasonable estimate of the collision probability and it could be used to estimate the number of re-samplings needed to complete `Algorithm 3.3`.

FACT 3.2 `Algorithm 3.3` *requires a constant number of sorting steps on average, and $O(m)$ additional space, to select uniformly at random m items from the sequence $S[1, n]$. This is $O(m \log m)$ average time and* $\min\{m, n/B\}$ *worst-case I/Os if $m \leq M$ is small enough to keep the sampled positions in internal memory. Otherwise an external-memory sorter is needed. The average depends on the re-sampling, integers are returned in sorted order for streaming-like access to the sequence S.*

Sorting could be speeded up by deploying the specialty of our problem, namely, that items to be sorted are m random integers in a fixed range $[1, n]$. Hence we could use either radix-sort or, even better for its simplicity, bucket sort. In the latter case, we can use an array of m slots each identifying a range of n/m positions in $[1, n]$. If item i_j is randomly selected, then it is inserted in slot $\lceil jm/n \rceil$. Since the m items are randomly sampled, each slot will contain $O(1)$ items on average, so that we can sort them in constant time per bucket by using insertion sort or the built-in `qsort`.

FACT 3.3 `Algorithm 3.3` *based on bucket-sort requires $O(m)$ average time and $O(m)$ additional space, whenever $m \leq M$. The average depends on the re-sampling. Integers are returned in sorted order for streaming-like access to the sequence S.*

We conclude this section by noticing that all the proposed algorithms, except `Algorithm 3.1`, generate the set of sampled positions using $O(m)$ space. If $m \leq M$ the random generation can occur within main memory without incurring in any I/Os. Sometimes this is useful because the randomized algorithm that invokes the random-sampling subroutine does not need the corresponding items, but rather their positions.

3.2 Streaming model and known sequence length

We next turn to the case in which S is flowing through a channel and the input size n is known and big (e.g. Internet traffic or query logs). We will turn to the more general case in which n is unknown at the end of the lecture, in the next section. This stream-based model imposes that no preprocessing is possible (as instead done above where items' positions were re-sampled and/or sorted), every item of S is considered once and the algorithm must immediately and irrevocably take a decision whether or not that item must be included or not in the set of sampled items. Possibly future items may kick out that one from the sampled set, but no item can be re-considered again in the future. Even in this case the algorithms are simple in their design but their probabilistic analysis is a little bit more involved than before. The algorithms of the previous section offer an *average* time complexity because they are faced with the re-sampling problem: possibly some samples have to be eliminated because duplicated. In order to avoid re-sampling, we need to ensure that each item is not considered more than once. So the algorithms that follow implement this idea in the simplest possible way, namely, they scan the input sequence S and consider each item once for all. This approach brings with itself two main difficulties which are related with the guarantee of both conditions: uniform sample from the range $[1, n]$ and sample of size m.

We start by designing an algorithm that draws just one item from S (hence $m = 1$), and then we generalize it to the case of a subset of $m > 1$ items. This algorithm proceeds by selecting the item $S[j]$ with probability $\mathcal{P}(j)$ which is properly defined in order to guarantee both two properties above.[2] In particular we set $\mathcal{P}(1) = 1/n, \mathcal{P}(2) = 1/(n-1), \mathcal{P}(3) = 1/(n-2)$ etc. etc., so the algorithm selects the item j with probability $\mathcal{P}(j) = \frac{1}{n-j+1}$, and if this occurs it stops. Eventually item $S[n]$ is selected because its drawing probability is $\mathcal{P}(n) = 1$. So the proposed algorithm guarantees the condition on the sample size $m = 1$, but more subtle is to prove that the probability of sampling $S[j]$ is $1/n$, independently of j, given that we defined $\mathcal{P}(j) = 1/(n - j + 1)$. The reason derives from a simple probabilistic argument because $n - j + 1$ is the number of remaining elements in the sequence and all of them have to be drawn uniformly at random. By induction, the first $j - 1$ items of the sequence have uniform probability $1/n$ to be sampled; so it is $1 - \frac{j-1}{n}$ the probability of not selecting anyone of them. As a result,

$$\mathcal{P}(\text{Sample } i_j) = \mathcal{P}(\text{Not sampling } i_1 \cdots i_{j-1}) \times \mathcal{P}(\text{Picking } i_j) = (1 - \frac{j-1}{n}) \times \frac{1}{n-j+1} = 1/n$$

Algorithm 3.4 Scanning and selecting

```
s = 0;
for (j = 1; j ≤ n; j++) do
    p = Rand(0, 1);
    if (p ≤ (m−s)/(n−j+1)) then
        select S[j];
        s++;
    end if
end for
```

[2]In order to draw an item with probability p, it suffices to draw a random real $r \in [0, 1]$ and then compare it against p. If $r \leq p$ then the item is selected, otherwise it is not.

`Algorithm 3.4` works for an arbitrarily large sample $m \geq 1$. The difference with the previous algorithm lies in the probability of sampling $S[j]$ which is now set to $\mathcal{P}(j) = \frac{m-s}{n-j+1}$ where s is the number of items already selected before $S[j]$. Notice that if we already got all samples, it is $s = m$ and thus $\mathcal{P}(j) = 0$, which correctly means that `Algorithm 3.4` does not generate more than m samples. On the other hand, it is easy to convince ourselves that `Algorithm 3.4` cannot generate less than m items, say y, given that the last $m - y$ items of S would have probability 1 to be selected and thus they would be surely included in the final sample (according to Step 4). As far as the uniformity of the sample is concerned, we show that $\mathcal{P}(j)$ equals the probability that $S[j]$ is included in a random sample of size m given that s samples lie within $S[1, j-1]$. We can rephrase this as the probability that $S[j]$ is included in a random sample of size $m - s$ taken from $S[j, n]$, and thus from $n - j + 1$ items. This probability is obtained by counting how many such combinations include $S[j]$, i.e. $\binom{n-j}{m-s-1}$, and dividing by the number of all combinations that include or not $S[j]$, i.e. $\binom{n-j+1}{m-s}$. Substituting to $\binom{b}{a} = \frac{b!}{a!\,(b-a)!}$ we get the formula for $\mathcal{P}(j)$.

FACT 3.4 `Algorithm 3.4` *takes $O(n/B)$ I/Os, $O(n)$ time, n random samples, and $O(m)$ additional space to sample uniformly m items from the sequence $S[1, n]$ in a streaming-like way.*

We conclude this section by pointing out a sophisticated solution proposed by Jeff Vitter [2] that reduces the amount of randomly-generated numbers from n to m, and thus speeds up the solution to $O(m)$ time and I/Os. This solution could be also fit into the framework of the previous section (random access to input data), and in that case its specialty would be the avoidance of re-sampling. Its key idea is not to generate random *indicators*, which specify whether or not an item $S[j]$ has to be selected, but rather generate random *jumps* that count the number of items to skip over before selecting the next item of S. Vitter introduces a random variable $G(v, V)$ where v is the number of items remaining to be selected, and V is the total number of items left to be examined in S. According to our previous notation, we have that $v = m - s$ and $V = n - j + 1$. The item $S[G(v, V) + 1]$ is the next one selected to form the uniform sample. It goes without saying that this approach avoids the generation of duplicate samples, but yet it incurs in an average bound because of the cost of generating the jumps according to the following distribution:

$$\mathcal{P}(G = g) = \binom{V - g - 1}{v - 1} \Big/ \binom{V}{v}$$

In fact the key problem here is that we cannot tabulate (and store) the values of all binomial coefficients in advance, because this would need space exponential in $V = \Theta(n)$. Surprisingly Vitter solved this problem in $O(1)$ average time, by adapting in an elegant way the von Neumann's rejection-acceptance method to the discrete case induced by G's jumps. We refer the reader to [2] for further details.

3.3 Streaming model and unknown sequence length

It goes without saying that the knowledge of n was crucial to compute $\mathcal{P}(j)$ in `Algorithm 3.4`. If n is unknown we need to proceed differently, and indeed the rest of this lecture is dedicated to detail two possible approaches.

The first one is pretty much simple and deploys a min-heap $\mathcal{H}$ of size m plus a real-number random generator, say `Rand(0,1)`. The key idea underlying this algorithm is to associate a random key to each item of S and then use the heap $\mathcal{H}$ to select the items corresponding to the top-m keys. The pseudo-code below implements this idea, we notice that $\mathcal{H}$ is a min-heap so it takes $O(1)$ time

Algorithm 3.5 Heap and random keys

```
1: Initialize the heap $\mathcal{H}$ with $m$ dummy pairs $\langle -\infty, \emptyset \rangle$;
2: for each item $S[j]$ do
3:     $r_j$ = Rand(0, 1);
4:     $m$ = the minimum key in $\mathcal{H}$;
5:     if ($r_j > m$) then
6:         extract the minimum key;
7:         insert $\langle r_j, S[j] \rangle$ in $\mathcal{H}$;
8:     end if
9: end for
10: return $\mathcal{H}$
```

to detect the minimum key among the current top-m ones. This is the key compared with r_j in order to establish whether or not $S[j]$ must enter the top-m set.

Since the heap has size m, the final sample will consists of m items. Each item takes $O(\log m)$ time to be inserted in the heap. So we have proved the following:

FACT 3.5 `Algorithm 3.5` *takes $O(n/B)$ I/Os, $O(n \log m)$ time, generates n random numbers, and uses $O(m)$ additional space to sample uniformly at random m items from the sequence $S[1, n]$ in a streaming-like way and without the knowledge of n.*

We conclude the lecture by introducing the elegant *reservoir sampling* algorithm, designed by Knuth in 1997, which improves `Algorithm 3.5` both in time and space complexity. The idea is similar to the one adopted for `Algorithm 3.4` and consists of properly defining the probability with which an item is selected. The key issue here is that we cannot take an irrevocable decision on $S[j]$ because we do not know how long the sequence S is, so we need some freedom to *change* what we have decided so far as the scanning of S goes on.

Algorithm 3.6 Reservoir sampling

```
1: Initialize array $R[1, m] = S[1, m]$;
2: for each next item $S[j]$ do
3:     $h$ = Rand(1, $j$);
4:     if $h \leq m$ then
5:         set $R[h] = S[j]$;
6:     end if
7: end for
8: return array $R$;
```

The pseudo-code of `Algorithm 3.6` uses a "reservoir" array $R[1, m]$ to keep the candidate samples. Initially R is set to contain the first m items of the input sequence. At any subsequent step j, the algorithm makes a choice whether $S[j]$ has to be included or not in the current sample. This choice occurs with probability $\mathcal{P}(j) = m/j$, in which case some previously selected item has to be kicked out from R. This item is chosen at random, hence with probability $1/m$. This double-choice is implemented in `Algorithm 3.6` by choosing an integer h in the range $[1, j]$, and making the substitution only if $h \leq m$. This event has probability m/j: exactly what we wished to set for $\mathcal{P}(j)$.

For the correctness, it is clear that `Algorithm 3.6` selects m items, it is less clear that these items are drawn uniformly at random from S, which actually means with probability m/n. Let's see why by assuming inductively that this property holds for a sequence of length $n-1$. The base case in which $n = m$ is obvious, every item has to be selected with probability $m/n = 1$, and indeed this is what Step 1 does by selecting all $S[1, m]$ items. To prove the inductive step (from $n-1$ to n items), we notice that the uniform-sampling property holds for $S[n]$ since by definition that item is inserted in R with probability $\mathcal{P}(n) = m/n$ (Step 4). Computing the probability of being sampled for the other items in $S[1, n-1]$ is more difficult to see. An item belongs to the reservoir R at the n-th step of `Algorithm 3.6` iff it was in the reservoir at the $(n-1)$-th step and it is not kicked out at the n-th step. This latter may occur either if $S[n]$ is not picked (and thus R is untouched) or if $S[n]$ is picked and $S[j]$ is not kicked out from R (being these two events independent of each other). In formulas,

$$\begin{aligned}\mathcal{P}(\text{item } S[j] \in R \text{ after } n \text{ items}) &= \mathcal{P}(S[j] \in R \text{ after } n-1 \text{ items}) \times [\mathcal{P}(S[n] \text{ is not picked}) \\ &\quad + \mathcal{P}(S[n] \text{ is picked}) \times \mathcal{P}(S[j] \text{ is not removed from } R)]\end{aligned}$$

Now, each of these items has probability $m/(n-1)$ of being in the reservoir R, by the inductive hypothesis, before that $S[n]$ is processed. Item $S[j]$ remains in the reservoir if either $S[n]$ is not picked (which occurs with probability $1 - \frac{m}{n}$) or if it is not kicked out by the picked $S[n]$ (which occurs with probability $\frac{m-1}{m}$). Summing up these terms we get

$$\mathcal{P}(\text{item } S[j] \in R \text{ after } n \text{ items}) = \frac{m}{n-1} \times [(1 - \frac{m}{n}) + (\frac{m}{n} \times \frac{m-1}{m})] = \frac{m}{n-1} \times \frac{n-1}{n} = \frac{m}{n}$$

To understand this formula assume that we have a reservoir of 1000 items, so the first 1000 items of S are inserted in R by Step 1. Then the item 1001 is inserted in the reservoir with probability 1000/1001, the item 1002 with probability 1000/1002, and so on. Each time an item is inserted in the reservoir, a random element is kicked out from it, hence with probability 1/1000. After n steps the reservoir R contains 1000 items, each sampled from S with probability $1000/n$.

FACT 3.6 `Algorithm 3.6` *takes $O(n/B)$ I/Os, $O(n)$ time, n random numbers, and exactly m additional space, to sample uniformly at random m items from the sequence $S[1, n]$ in a streaming-like way and without the knowledge of n. Hence it is time, space and I/O-optimal in this model of computation.*

References

[1] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest and Clifford Stein. *Introduction to Algorithms*. Chapter 11: "Hashing", The MIT press, third edition, 2009.

[2] Jeffrey Scott Vitter. Faster methods for random sampling. *ACM Computing Surveys*, 27(7):703–718, 1984.

4 List Ranking

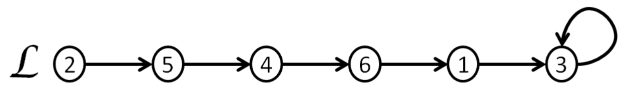
"Pointers are dangerous in disks!"

This lecture attacks a simple problem over lists, the basic data structure underlying the design of many algorithms which manage interconnected items. We start with an easy to state, but inefficient solution derived from the optimal one known for the RAM model; and then discuss more and more sophisticated solutions that are elegant, efficient/optimal but still simple enough to be coded with few lines. The treatment of this problem will allow also us to highlight a subtle relation between *parallel computation* and *external-memory computation*, which can be deployed to derive efficient disk-aware algorithms from efficient parallel algorithms.

Problem. *Given a (mono-directional) list $\mathcal{L}$ of n items, the goal is to compute the distance of each of those items from the tail of $\mathcal{L}$.*

Items are represented via their ids, which are integers from 1 to n. The list is encoded by means of an array $\texttt{Succ}[1, n]$ which stores in entry $\texttt{Succ}[i]$ the id j if item *i points to* item j. If t is the id of the tail of the list $\mathcal{L}$, then we have $\texttt{Succ}[t] = t$, and thus the link outgoing from t forms a *self-loop*. The following picture exemplifies these ideas by showing a graphical representation of a list (left), its encoding via the array Succ (right), and the output required by the list-ranking problem, hereafter encoded in the array $\texttt{Rank}[1, n]$.

This problem can be solved easily in the RAM model by exploiting the constant-time access to its internal memory. We can foresee three solutions. The first one scans the list from its head and computes the number n of its items, then re-scans the list by assigning to its head the rank $n-1$ and to



id	Succ	Rank
1	3	1
2	5	5
3	3	0
4	6	3
5	4	4
6	1	2

FIGURE 4.1: An example of input and output for the List Ranking problem.

every subsequent element in the list a value decremented by one at every step. The second solution computes the array of predecessors as Pred[Succ[i]] = i; and then scans the list backward, starting from its tail t, setting Rank[t] = 0, and then incrementing the Rank's value of each item as the percolated distance from t. The third way to solve the problem is *recursively*, without needing (an explicit) additional working space, by defining the function ListRank(i) which works as follows: Rank[i] = 0 if Succ[i] = i (and hence $i = t$), else it sets Rank[i] = ListRank(Succ[i]) + 1; at the end the function returns the value Rank[i]. The time complexity of both algorithms is $O(n)$, and obviously it is optimal since all list's items must be visited to set their n Rank's values.

If we execute this algorithm over a list stored on disk (via its array Succ), then it could elicit $\Theta(n)$ I/Os because of the arbitrary distribution of links which might induce an irregular pattern of disk accesses to the entries of arrays Rank and Succ. This I/O-cost is significantly far from the lower-bound $\Omega(n/B)$ which can be derived by the same argument we used above for the RAM model. Although this lower-bound seems very low, we will come in this lecture very close to it by introducing a bunch of sophisticated techniques that are general enough to find applications in many other, apparently dissimilar, contexts.

The moral of this lecture is that, in order to achieve I/O-efficiency on *linked* data structures, you need to avoid the *percolation* of pointers as much as possible; and possibly dig into the wide parallel-algorithms literature (see e.g. [2]) because efficient parallelism can be turned surprisingly into I/O-efficiency.

4.1 The pointer-jumping technique

There exists a well-known technique to solve the list-ranking problem in the parallel setting, based on the so called *pointer jumping* technique. The algorithmic idea is pretty much simple, it takes n processors, each dealing with one item of $\mathcal{L}$. Processor i initializes Rank[i] = 0 if $i = t$, otherwise it sets Rank[i] = 1. Then executes the following two instructions: Rank[i] += Rank[Succ[i]], Succ[i] = Succ[Succ[i]]. This update actually maintains the following invariant: Rank[i] *measures the distance (number of items) between i and the current* Succ[i] *in the original list.* We skip the formal proof that can be derived by induction, and refer the reader to the illustrative example in Figure 4.2.

In that Figure the red-dashed arrows indicate the new links computed by one pointer-jumping step, and the table on the right of each list specifies the values of array Rank[$1, n$] as they are recomputed after this step. The values in bold are the final/correct values. We notice that distances do not grow linearly (i.e. $1, 2, 3, \ldots$) but they grow as a power of two (i.e. $1, 2, 4, \ldots$), up to the step in which the next jump leads to reach t, the tail of the list. This means that the total number of times the parallel algorithm executes the two steps above is $O(\log n)$, thus resulting an exponential improvement with respect to the time required by the sequential algorithm. Given that n processors are involved, pointer-jumping executes a total of $O(n \log n)$ operations, which is inefficient if we compare it to the number $O(n)$ operations executed by the optimal RAM algorithm.

LEMMA 4.1 The parallel algorithm, using n processors and the pointer-jumping technique, takes $O(\log n)$ time and $O(n \log n)$ operations to solve the list-ranking problem.

Optimizations are possible to further improve the previous result and come close the optimal number of operations; for example, by *turning off* processors, as their corresponding items reach the end of the list, could be an idea but we will not dig into these details (see e.g. [2]) because they pertain to a course on parallel algorithms. Here we are interested in *simulating* the pointer-jumping technique in our setting which consists of one single processor and a 2-level memory, and show that deriving an I/O-efficient algorithm is very simple whenever an efficient parallel algorithm is

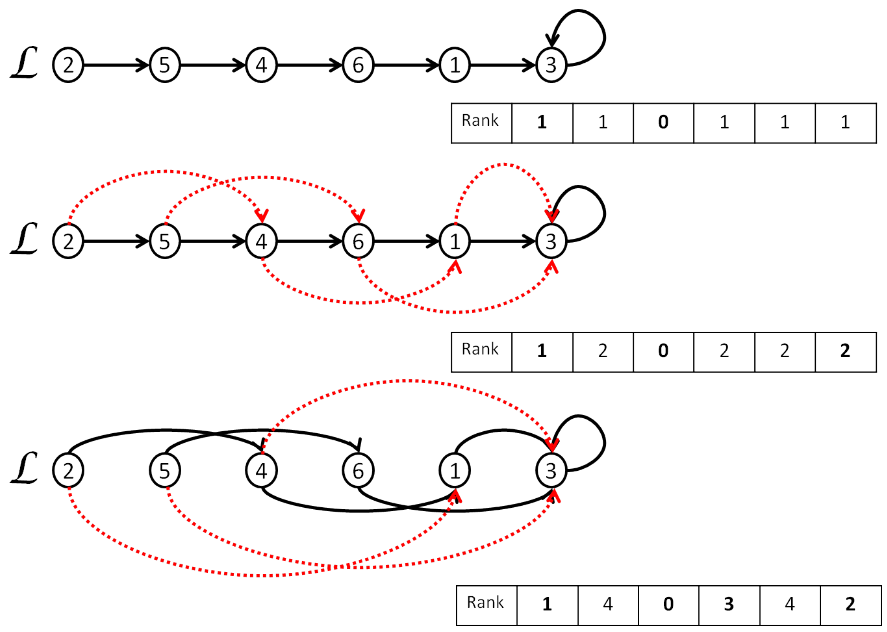


FIGURE 4.2: An example of pointer jumping applied to the list $\mathcal{L}$ of Figure 4.1. The dotted arrows indicate one pointer-jumping step applied onto the solid arrows, which represent the current configuration of the list.

available. The simplicity hinges onto an algorithmic scheme which deploys two basic primitives— Scan and Sort a set of triples— nowadays available in almost every distributed platforms, such as Apache Hadoop.

4.2 Parallel algorithm simulation in a 2-level memory

The key difficulty in using the pointer-jumping technique within the 2-level memory framework is the arbitrary layout of the list on disk, and the consequent arbitrary pattern of memory accesses to update `Succ`-pointers and `Rank`-values, which might induce many I/Os. To circumvent this problem we will describe how the two key steps of the pointer-jumping approach can be simulated via a constant number of `Sort` and `Scan` primitives over n triples of integers. Sorting is a basic primitive which is very much complicated to be implemented I/O-efficiently, and indeed will be the subject of the entire Chapter 5. For the sake of presentation, we will indicate its I/O-complexity as $\widetilde{O}(n/B)$ which means that we have hidden a logarithmic factor depending on the main parameters of the model, namely M, n, B. This factor is negligible in practice, since we can safely upper bound it with 4 or less, and so we prefer now to *hide it* in order to avoid jeopardizing the reading of this chapter. On the other hand, `Scan` is easy and takes $O(n/B)$ I/Os to process a contiguous disk portion occupied by the n triples.

We can identify a common algorithmic structure in the two steps of the pointer-jumping technique: each of them consists of an operation (either copy or sum) between two entries of an array (either `Succ` or `Rank`). For the sake of presentation we will refer to a generic array A, and model the parallel operation to be simulated on disk as follows:

Assume that a parallel step has the following form: $A[a_i]$ `op` $A[b_i]$, where `op` is the operation executed in parallel over the two array entries $A[a_i]$ and $A[b_i]$ by all processors $i = 1, 2, \ldots, n$ which actually read $A[b_i]$ and use this value to update the content of $A[a_i]$.

The operation `op` is a sum and an assignment for the update of the `Rank`-array (here $A =$ `Rank`), it is a copy for the update of the `Succ`-array (here $A =$ `Succ`). As far as the array indices are concerned they are, for both steps, $a_i = i$ and $b_i =$ `Succ`$[i]$. The key issue is to show that $A[a_i]$ `op` $A[b_i]$ can be implemented, simultaneously over all $i = 1, 2, 3, \ldots, n$, by using a constant number of `Sort` and `Scan` primitives, thus taking a total of $\widetilde{O}(n/B)$ I/Os. The simulation consists of 5 steps:

1. `Scan` the disk and create a sequence of triples having the form $\langle a_i, b_i, 0\rangle$. Every triple brings information about the source address of the array-entry involved in `op` (b_i), its destination address (a_i), and the value that we are moving (the third component, initialized to 0).
2. `Sort` the triples according to their second component (i.e. b_i). This way, we are "aligning" the triple $\langle a_i, b_i, 0\rangle$ with the memory cell $A[b_i]$.
3. `Scan` the triples and the array A to create the new triples $\langle a_i, b_i, A[b_i]\rangle$. Notice that not all memory cells of A are referred as second component of any triple, nevertheless their coordinated order allows to copy $A[b_i]$ into the triple for b_i via a coordinated scan.
4. `Sort` the triples according to their first component (i.e. a_i). This way, we are aligning the triple $\langle a_i, b_i, A[b_i]\rangle$ with the memory cell $A[a_i]$.
5. `Scan` the triples and the array A and, for every triple $\langle a_i, b_i, A[b_i]\rangle$, update the content of the memory cell $A[a_i]$ according to the semantics of `op` and the value $A[b_i]$.

I/O-complexity is easy to derive since the previous algorithm is executing 2 `Sort` and 3 `Scan` involving n triples. Therefore we can state the following:

THEOREM 4.1 *The parallel execution of n operations $A[a_i]$ `op` $A[b_i]$ can be simulated in a 2-level memory model by using a constant number of `Sort` and `Scan` primitives, thus taking a total of $\widetilde{O}(n/B)$ I/Os.*

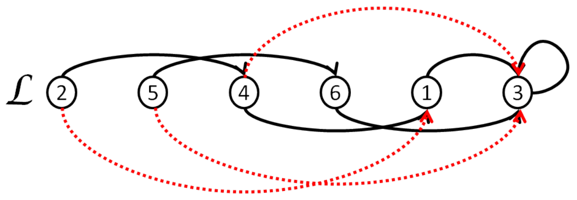
In the case of the parallel pointer-jumping algorithm, this parallel assignment is executed for $O(\log n)$ steps, so we have:

THEOREM 4.2 *The parallel pointer-jumping algorithm can be simulated in a 2-level memory model taking $\widetilde{O}((n/B)\log n)$ I/Os.*

This bound turns to be $o(n)$, and thus better than the direct execution of the sequential algorithm on disk, whenever $B = \omega(\log n)$. This condition is trivially satisfied in practice because $B \approx 10^4$ bytes and $\log n \leq 80$ for any real dataset size (being 2^{80} the number of atoms in the Universe[1]).

Figure 4.3 reports a running example of this simulation over the list at the top of the Figure 4.3. Table on the left indicates the content of the arrays `Rank` and `Succ` encoding the list; table on the right indicates the content of these two arrays after one step of pointer-jumping. The five columns of triples correspond to the application of the five `Scan`/`Sort` phases. This simulation is related to the update of the array `Rank`, array `Succ` can be recomputed similarly. Actually, the update

[1]See e.g. `http://en.wikipedia.org/wiki/Large_numbers`



Item	1	2	3	4	5	6
Rank	1	2	0	2	2	2
Succ	3	4	3	1	6	3

Item	1	2	3	4	5	6
Rank	1	4	0	3	4	2
Succ	3	1	3	3	3	3

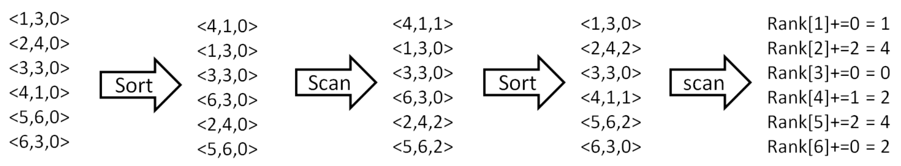


FIGURE 4.3: An example of simulation of the basic parallel step via Scan and Sort primitives, relative to the computation of the array Rank, with the configuration specified in the picture and tables above.

can be done simultaneously by using a quadruple instead of a triple which brings both the values of Rank[Succ[i]] and the value of Succ[Succ[i]], thus deploying the fact that both values use the same source and destination address (namely, i and Succ[i]).

The first column of triples is created as $\langle i, \text{Succ}[i], 0\rangle$, since $a_i = i$ and $b_i = \text{Succ}[i]$. The third column of triples is sorted by the second component, namely Succ[i], and so its third component is obtained by Scanning the array Rank and creating $\langle i, \text{Succ}[i], \text{Rank}[\text{Succ}[i]]\rangle$. The fourth column of triples is ordered by their first component, namely i, so that the final Scan-step can read in parallel the array Rank and the third component of those triples, and thus compute correctly Rank[i] as $\text{Rank}[i] + \text{Rank}[\text{Succ}[i]] = 1 + \text{Rank}[\text{Succ}[i]]$.

The simulation scheme introduced in this section can be actually generalized to every parallel algorithm thus leading to the following important, and useful, result (see [1]):

THEOREM 4.3 *Every parallel algorithm using n processors and taking T steps can be simulated in a 2-level memory by a disk-aware sequential algorithm taking $\widetilde{O}((n/B)\ T)$ I/Os and $O(n)$ space.*

This simulation is advantageous whenever $T = o(B)$, which implies a sub-linear number of I/Os $o(n)$. This occurs in all cases in which the parallel algorithm takes a low *poly-logarithmic* time-complexity. This is exactly the situation of parallel algorithms developed over the so called P-RAM model of computation which assumes that all processors work independently of each other and they can access in constant time an unbounded shared memory. This is an ideal model which was very famous in the '80s-'90s and led to the design of many powerful parallel techniques, which have been then applied to distributed as well as disk-aware algorithms. Its main limit was to do not account for conflicts among the many processors accessing the shared memory, and a simplified communication among them. Nevertheless this simplified model allowed researchers to concentrate onto the algorithmic aspects of parallel computation and thus design precious parallel schemes as the ones described below.

4.3 A Divide&Conquer approach

The goal of this section is to show that the list-ranking problem can be solved more efficiently than pointer-jumping on a list. The algorithmic solution we describe in this section relies on an interesting application of the *Divide&Conquer paradigm*, here specialized to work on a (mono-directional) list of items.

Before going into the technicalities related to this application, let us briefly recall the main ideas underlying the design of an algorithm, say $\mathcal{A}_{dc}$, based on the Divide&Conquer technique which solves a problem $\mathcal{P}$, formulated on n input data. $\mathcal{A}_{dc}$ consists of three main phases:

Divide. $\mathcal{A}_{dc}$ creates a set of k *subproblems*, say $\mathcal{P}_1, \mathcal{P}_2, \ldots, \mathcal{P}_k$, having sizes $n_1, n_2, \ldots, n_k$, respectively. They are identical to the original problem $\mathcal{P}$ but are formulated on smaller inputs, namely $n_i < n$.

Conquer. $\mathcal{A}_{dc}$ is invoked *recursively* on the subproblems $\mathcal{P}_i$, thus getting the solution s_i.

Recombine. $\mathcal{A}_{dc}$ recombines the solutions s_i to obtain the solution s for the original problem $\mathcal{P}$. s is *returned* as output of the algorithm.

It is clear that the Divide&Conquer technique originates a *recursive* algorithm $\mathcal{A}_{dc}$, which needs a *base case* to terminate. Typically, the base case consists of stopping $\mathcal{A}_{dc}$ whenever the input consists of few items, e.g. $n \leq 1$. In these small-input cases the solution can be computed easily and directly, possibly by enumeration.

The time complexity $T(n)$ of $\mathcal{A}_{dc}$ can be described as a recurrence relation, in which the base condition is $T(n) = O(1)$ for $n \leq 1$, and for the other cases it is:

$$T(n) = D(n) + R(n) + \sum_{i=1,\ldots,k} T(n_i)$$

where $D(n)$ is the cost of the Divide step, $R(n)$ is the cost of the Recombination step, and the last term accounts for the cost of all recursive calls. These observations are enough for these notes; we refer the reader to Chapter 4 in [3] for a deeper and clean discussion about the Divide&Conquer technique and the Master Theorem that provides a mathematical solution to recurrence relations, such as the one above.

We are ready now to specialize the Divide&Conquer technique over the List-Ranking problem. The algorithm we propose is pretty simple and starts by assigning to each item i the value $\texttt{Rank}[i] = 0$ for $i = t$, otherwise $\texttt{Rank}[i] = 1$. Then it executes three main steps:

Divide. We identify a set of items $I = \{i_1, i_2, \ldots, i_h\}$ drawn from the input list $\mathcal{L}$. Set I must be an *independent set*, which means that the successor of each item in I does not belong to I. This condition clearly guarantees that $|I| \leq n/2$, because at most one item out of two consecutive items may be selected. The algorithm will guarantee also that $|I| \geq n/c$, where $c > 2$, in order to make the approach effective.

Conquer. Form the list $\mathcal{L}^* = \mathcal{L} - I$, by pointer-jumping only on the predecessors of the removed items I: namely, for every $\texttt{Succ}[x] \in I$ we set $\texttt{Rank}[x] += \texttt{Rank}[\texttt{Succ}[x]]$ and $\texttt{Succ}[x] = \texttt{Succ}[\texttt{Succ}[x]]$. This way, at any recursive call, $\texttt{Rank}[x]$ accounts for the number of items of the original input list that lie between x and the current $\texttt{Succ}[x]$. Then solve recursively the list-ranking problem over $\mathcal{L}^*$. Notice that $n/2 \leq |\mathcal{L}^*| \leq (1 - 1/c)n$, so that the recursion acts on a list which is a fractional part of $\mathcal{L}$. This is crucial for the efficiency of the recursive calls.

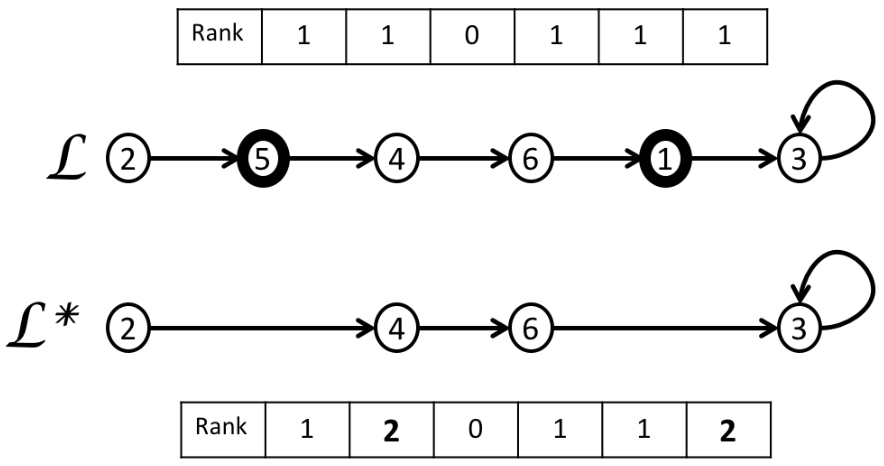


FIGURE 4.4: An example of reduction of a list due to the removal of the items in an Independent Set, here specified by the bold nodes. The new list on the bottom is the one resulting from the removal, the Rank-array is recomputed accordingly to reflect the missing items. Notice that the Rank-values are 1, 0 for the tail, because we assume that $\mathcal{L}$ is the initial list.

Recombine. At this point we can assume that the recursive call has computed correctly the list-ranking of all items in $\mathcal{L}^*$. So, in this phase, we derive the rank of each item $x \in I$ as Rank[x] = Rank[x] + Rank[Succ[x]], by adopting an update rule which reminds the one used in pointer jumping. The correctness of Rank-computation is given by two facts: (i) the independent-set property about I ensures that Succ[x] $\notin I$, thus Succ[x] $\in \mathcal{L}^*$ and so its Rank is available; (ii) by induction, Rank[Succ[x]] accounts for the distance of Succ[x] from the tail of $\mathcal{L}$ and Rank[x] accounts for the number of items between x and Succ[x] in the original input list (as observed in Conquer's step). In fact, the removal of x (because of its selection in I, may occur at any recursive step so that x may be far from the current Succ[x] when considered the original list; this means that it might be the case of Rank[x] $\gg 1$, which the previous summation-step will take into account. As an example, Figure 4.4 depicts the starting situation in which all ranks are 1 except the tail's one, so the update is just Rank[x] = 1 + Rank[Succ[x]]. In a general recursive step, Rank[x] ≥ 1 and so we have to take care of this when updating its value. As a result all items in $\mathcal{L}$ have their Rank-value correctly computed and, thus, induction is preserved and the algorithm may return to its invoking caller.

Figure 4.4 illustrates how an independent set (denoted by bold nodes) is removed from the list $\mathcal{L}$ and how the Succ-links are properly updated. Notice that we are indeed pointer-jumping only on the predecessors of the removed items (namely, the predecessors of the items in I), and that the other items leave untouched their Succ-pointers. It is clear that, if the next recursive step selects $I = \{6\}$, the final list will be constituted by three items $\mathcal{L} = (2, 4, 3)$ whose final ranks are $(6, 3, 0)$, respectively. The Recombination-step will re-insert 6 in $\mathcal{L} = (2, 4, 3)$, just after 4, and compute Rank[6] = Rank[6] + Rank[3] = 2 + 0 = 2 because Succ[6] = 3 in the current list. Conversely, if one would have not taken into account the fact that item 6 may be far from its current Succ[6] = 3, when referred to the original list, and summed 1 it would have made a wrong calculation for Rank[6].

This algorithm makes clear that its I/O-efficiency depends onto the Divide-step. In fact, Conquer-step is recursive and thus can be estimated as $T((1 - \frac{1}{c})n)$ I/Os; Recombine-step executes all reinsertions simultaneously, given that the removed items are not contiguous (by definition of independent set), and can be implemented by Theorem 4.1 in $\widetilde{O}(n/B)$ I/Os.

THEOREM 4.4 *The list-ranking problem formulated over a list $\mathcal{L}$ of length n, can be solved via a Divide&Conquer approach taking $T(n) = I(n) + \widetilde{O}(n/B) + T((1 - \frac{1}{c})n)$ I/Os, where $I(n)$ is the I/O-cost of selecting an independent set from $\mathcal{L}$ of size at least n/c (and, of course, at most $n/2$).*

Deriving a large independent set is trivial if a scan of the list $\mathcal{L}$ is allowed, just pick one every two items. But in our disk-context the list scanning is I/O-inefficient and this is exactly what we want to avoid: otherwise we would have solved the list-ranking problem!

In what follows we will therefore concentrate on the problem of identifying a *large* independent set within the list $\mathcal{L}$. The solution must deploy only local information within the list, in order to avoid the execution of many I/Os. We will propose two solutions: one is simple and randomized, the other one is deterministic and more involved. It is surprising that the latter technique (called *deterministic coin tossing*) has found applications in many other contexts, such as data compression, text similarity, string-equality testing. It is a very general and powerful technique that, definitely, deserves some attention in these notes.

4.3.1 A randomized solution

The algorithmic idea, as anticipated above, is simple: toss a fair coin for each item in $\mathcal{L}$, and then select those items i such that $\texttt{coin}(i) = \texttt{H}$ but $\texttt{coin}(\texttt{Succ}[i]) = \texttt{T}$.[2]

The probability that the item i is selected is $\frac{1}{4}$, because this happens for one configuration (HT) out of the four possible configurations. So the average number of items selected for I is $n/4$. By using sophisticated probabilistic tools, such as Chernoff bounds, it is possible to prove that the number of selected items is strongly concentrated around $n/4$. This means that the algorithm can repeat the coin tossing until $|I| \geq n/c$, for some $c > 4$. The strong concentration guarantees that this repetition is executed a (small) constant number of times.

We finally notice that the check on the values of `coin`, for selecting I's items, can be simulated by Theorem 4.1 via few `Sort` and `Scan` primitives, thus taking $I(n) = \widetilde{O}(n/B)$ I/Os on average. So, by substituting this value in Theorem 4.4, we get the following recurrence relation for the I/O-complexity of the proposed algorithm: $T(n) = \widetilde{O}(n/B) + T(\frac{3n}{4})$. It can be shown by means of the Master Theorem (see Chapter 4 in [3]) that this recurrence relation has solution $\widetilde{O}(n/B)$.

THEOREM 4.5 *The list-ranking problem, formulated over a list $\mathcal{L}$ of length n, can be solved with a randomized algorithm in $\widetilde{O}(n/B)$ I/Os on average.*

4.3.2 Deterministic coin-tossing$^\infty$

The key property of the randomized process was the *locality* of I's construction which allowed to pick an item i by just looking at the results of the coins tossed for i itself and for its successor $\texttt{Succ}[i]$. In this section we try to simulate *deterministically* this process by introducing the so called *deterministic coin-tossing* strategy that, instead of assigning two coin values to each item (i.e. `H` and `T`), it starts by assigning n coin values (hereafter indicated with the integers $0, 1, \ldots, n-1$) and

[2]The algorithm works also in the case that we exchange the role of head (H) and tail (T); but it does not work if we choose the configurations HH or TT. Why?

eventually reduces them to *three* coin values (namely $0, 1, 2$). The final selection process for I will then pick the items whose coin value is minimum among their adjacent items in $\mathcal{L}$. Therefore, here, three possible values and three possible items to be compared, still a constant execution of `Sort` and `Scan` primitives.
The pseudo-code of the algorithm follows.

Initialization. Assign to each item i the value $\texttt{coin}(i) = i - 1$. This way all items take a different coin value, which is smaller than n. We represent these values in $b = \lceil \log n \rceil$ bits, and we denote by $\texttt{bit}_b(i)$ the binary representation of $\texttt{coin}(i)$ using b bits.

Get 6-coin values. Repeat the following steps until $\texttt{coin}(i) < 6$, for all i:

- Compute the position $\pi(i)$ where $\texttt{bit}_b(i)$ and $\texttt{bit}_b(\texttt{Succ}[i])$ differ, and denote by $z(i)$ the bit-value of $\texttt{bit}_b(i)$ at that position.
- Compute the new coin-value for i as $\texttt{coin}(i) = 2\pi(i) + z(i)$ and set the new binary-length representation as $b = \lceil \log b \rceil + 1$.

Get just 3-coin values. For each element i, such that $\texttt{coin}(i) \in \{3, 4, 5\}$, do $\texttt{coin}(i) = \{0, 1, 2\} - \{\texttt{coin}(\texttt{Succ}[i]), \texttt{coin}(\texttt{Pred}[i])\}$.

Select *I*. Pick those items i such that $\texttt{coin}(i)$ is a local minimum, namely it is smaller than $\texttt{coin}(\texttt{Pred}[i])$ and $\texttt{coin}(\texttt{Succ}[i])$.

Let us first discuss the correctness of the algorithm. At the beginning all coin values are distinct, and in the range $\{0, 1, \ldots, n-1\}$. By distinctness, the computation of $\pi(i)$ is sound and $2\pi(i) + z(i) \leq 2(b-1) + 1 = 2b - 1$ since $\texttt{coin}(i)$ was represented with b bits and hence $\pi(i) \leq b - 1$ (counting from 0). Therefore, the new value $\texttt{coin}(i)$ can be represented with $\lceil \log b \rceil + 1$ bits, and thus the update of b is correct too.

A key observation is that the new value of $\texttt{coin}(i)$ is still different of the coin value of its adjacent items in $\mathcal{L}$, namely $\texttt{coin}(\texttt{Succ}[i])$ and $\texttt{coin}(\texttt{Pred}[i])$. We prove it by contradiction. Let us assume that $\texttt{coin}(i) = \texttt{coin}(\texttt{Succ}[i])$ (the other case is similar), then $2\pi(i) + z(i) = 2\pi(\texttt{Succ}[i]) + z(\texttt{Succ}[i])$. Since z denotes a bit value, the two coin-values are equal iff it is both $\pi(i) = \pi(\texttt{Succ}[i])$ and $z(i) = z(\texttt{Succ}[i])$. But if this condition holds then the two bit sequences $\texttt{bit}_b(i)$ and $\texttt{bit}_b(\texttt{Succ}[i])$ cannot differ at bit-position $\pi(i)$.

Easily it follows the correctness of the step which allows to go from 6-coin values to 3-coin values, as well as it is immediate the proof that the selected items form an independent set because of the minimality of $\texttt{coin}(i)$ and distinctness of adjacent coin values.

As far as the I/O-complexity is concerned, we start by introducing the function $\log^* n$ defined as $\min\{j \mid \log^{(j)} n \leq 1\}$, where $\log^{(j)} n$ is the repeated application of the logarithm function for j times to n. As an example[3] take $n = 16$ and compute $\log^{(0)} 16 = 16, \log^{(1)} 16 = 4, \log^{(2)} 16 = 2, \log^{(3)} 16 = 1$; thus $\log^* 16 = 3$. It is not difficult to convince yourselves that $\log^* n$ grows very much slowly, and indeed its value is 5 for $n = 2^{65536}$.

In order to estimate the I/O-complexity, we need to bound the number of iterations needed by the algorithm to reduce the coin-values to $\{0, 1, \ldots, 5\}$. This number is $\log^* n$, because at each step the reduction in the number of possible coin-values is logarithmic ($b = \lceil \log b \rceil + 1$). All single steps can be implemented by Theorem 4.1 via few `Sort` and `Scan` primitives, thus taking $\widetilde{O}(n/B)$ I/Os. So the construction of the independent set takes $I(n) = \widetilde{O}((n/B) \log^* n) = \widetilde{O}(n/B)$ I/Os, by definition of $\widetilde{O}()$. The size of I can be lower bounded as $|I| \geq n/4$ because the distance between two consecutive

[3]Recall that logarithms are in base 2 in these lectures.

selected items (local minima) is maximum when the coin-values form a *bitonic sequence* of the form $\ldots, 0, 1, 2, 1, 0, \ldots$.

By substituting this value in Theorem 4.4 we get the same recurrence relation of the randomized algorithm, with the exception that now the I/O-bound is worst case and deterministic: $T(n) = \widetilde{O}(n/B) + T(\frac{3n}{4})$.

THEOREM 4.6 *The list-ranking problem, formulated over a list $\mathcal{L}$ of length n, can be solved with a deterministic algorithm in $\widetilde{O}(n/B)$ worst-case I/Os.*

A comment is in order to conclude this chapter. The logarithmic term hidden in the $\widetilde{O}()$-notation has the form $(\log^* n)(\log_{M/B} n)$, which can be safely assumed to be smaller than 15 because, in practice, $\log_{M/B} n \leq 3$ and $\log^* n \leq 5$ for n up to 1 petabyte.

References

[1] Yi-Jen Chiang, Michael T. Goodrich, Edward F. Grove, Roberto Tamassia, Darren Erik Vengroff, Jeffrey Scott Vitter. External-Memory Graph Algorithms. *ACM-SIAM Symposium on Algorithms (SODA)*, 139-149, 1995.

[2] Joseph JaJa. *An Introduction to Parallel Algorithms*. Addison-Wesley, 1992.

[3] Tomas H. Cormen, Charles E. Leiserson, Ron L. Rivest, Clifford Stein. *Introduction to Algorithms*. The MIT Press, third edition, 2009.

[4] Jeffrey Scott Vitter. Faster methods for random sampling. *ACM Computing Surveys*, 27(7):703–718, 1984.

5 Sorting Atomic Items

This lecture will focus on the very-well known problem of sorting a set of *atomic* items, the case of *variable-length* items (aka strings) will be addressed in the following chapter. Atomic means that they occupy a constant-fixed number of memory cells, typically they are integers or reals represented with a fixed number of bytes, say 4 (32 bits) or 8 (64 bits) bytes each.

The sorting problem. *Given a sequence of n atomic items $S[1, n]$ and a total ordering $\leq$ between each pair of them, sort S in increasing order.*

We will consider two complemental sorting paradigms: the *merge-based* paradigm, which underlies the design of Mergesort, and the *distribution-based* paradigm which underlies the design of Quicksort. We will adapt them to work in the 2-level memory model, analyze their I/O-complexities and propose some useful tools that can allow to speed up their execution in practice, such as the Snow Plow technique and Data compression. We will also demonstrate that these disk-based adaptations are *I/O-optimal* by proving a sophisticated lower-bound on the number of I/Os any external-memory sorter must execute to produce an ordered sequence. In this context we will relate the Sorting problem with the so called *Permuting* problem, typically neglected when dealing with sorting in the RAM model.

The permuting problem. *Given a sequence of n atomic items $S[1, n]$ and a permutation $\pi[1, n]$ of the integers $\{1, 2, \ldots, n\}$, permute S according to π thus obtaining the new sequence $S[\pi[1]], S[\pi[2]], \ldots, S[\pi[n]]$.*

Clearly Sorting includes Permuting as a sub-task: to order the sequence S we need to determine its sorted permutation and then implement it (possibly these two phases are intricately intermingled). So Sorting should be more difficult than Permuting. And indeed in the RAM model we know that sorting n atomic items takes $\Theta(n \log n)$ time (via Mergesort or Heapsort) whereas permuting them takes $\Theta(n)$ time. The latter time bound can be obtained by just moving one item at a time according to what indicates the array π. Surprisingly we will show that this *complexity gap* does not exist in the disk model, because these two problems exhibit the same I/O-complexity under some reasonable conditions on the input and model parameters n, M, B. This elegant and deep result was obtained by Aggarwal and Vitter in 1998 [1], and it is surely the result that spurred the huge amount of algorithmic literature on the I/O-subject. Philosophically speaking, AV's result formally proves the intuition that *moving* items in the disk is the real *bottleneck*, rather than *finding* the sorted permutation. And indeed researchers and software engineers typically speak about the *I/O-bottleneck* to characterize this issue in their (slow) algorithms.

We will conclude this lecture by briefly mentioning at two solutions for the problem of sorting items on D-disks: the disk-striping technique, which is at the base of RAID systems and turns any efficient/optimal 1-disk algorithm into an efficient D-disk algorithm (typically loosing its optimality, if any), and the Greed-sort algorithm, which is specifically tailored for the sorting problem on D-disks and achieves I/O-optimality.

5.1 The merge-based sorting paradigm

We recall the main features of the external-memory model introduced in Chapter 1: it consists of an internal memory of size M and allows blocked-access to disk by reading/writing B items at once.

Algorithm 5.1 The binary merge-sort: MergeSort(S, i, j)

```
1: if $(i < j)$ then
2:     $m = (i + j)/2$;
3:     MergeSort($S, i, m - 1$);
4:     MergeSort($S, m, j$);
5:     Merge($S, i, m, j$);
6: end if
```

Mergesort is based on the Divide&Conquer paradigm. Step 1 checks if the array to be sorted consists of at least two items, otherwise it is already ordered and nothing has to be done. If items are more than two, it splits the input array S into two halves, and then recurses on each part. As recursion ends, the two halves $S[i, m - 1]$ and $S[m, j]$ are ordered so that Step 5 fuses them in $S[i, j]$ by invoking procedure Merge. This merging step needs an auxiliary array of size n, so that MergeSort is not an *in-place* sorting algorithm (unlike Heapsort and Quicksort) but needs $O(n)$ extra working space. Given that at each recursive call we halve the size of the input array to be sorted, the total number of recursive calls is $O(\log n)$. The Merge-procedure can be implemented in $O(j - i + 1)$ time by using two pointers, say x and y, that start at the beginning of the two halves $S[i, m - 1]$ and $S[m, j]$. Then $S[x]$ is compared with $S[y]$, the smaller is written out in the fused sequence, and its pointer is advanced. Given that each comparison advances one pointer, the total number of steps is bounded above by the total number of pointer's advancements, which is upper bounded by the length of $S[i, j]$. So the time complexity of MergeSort($S, 1, n$) can be modeled via the recurrence relation $T(n) = 2T(n/2) + O(n) = O(n \log n)$, as well known from any basic algorithm course.[1]

Let us assume now that $n > M$, so that S must be stored on disk and I/Os become the most important resource to be analyzed. In practice every I/O takes 5ms on average, so one could think that every item comparison takes one I/O and thus one could estimate the running time of Mergesort

[1] In all our lectures when the base of the logarithm is not indicated, it means 2.

on a massive S as: 5ms $\times$ $\Theta(n \log n)$. If n is of the order of few Gigabytes, say $n \approx 2^{30}$ which is actually not much massive for the current memory-size of commodity PCs, the previous time estimate would be of about $5 \times 2^{30} \times 30 > 10^8$ms, namely more than 1 day of computation. However, if we run Mergesort on a commodity PC it completes in few hours. This is not surprising because the previous evaluation totally neglected the existence of the internal memory, of size M, and the sequential pattern of memory-accesses induced by Mergesort. Let us therefore analyze the Mergesort algorithm in a more precise way within the disk model.

First of all we notice that $O(z/B)$ I/Os is the cost of merging two ordered sequences of z items in total. This holds if $M \geq 2B$, because the Merge-procedure in Algorithm 5.1 really keeps in internal memory the 2 pages that contain the two pointers scanning $S[i, j]$ where $z = j - i + 1$. Every time a pointer advances into another disk page, an I/O-fault occurs, the page is fetched in internal memory, and the fusion continues. Given that S is stored contiguously on disk, $S[i, j]$ occupies $O(z/B)$ pages and this is the I/O-bound for merging two sub-sequences of total size z. Similarly, the I/O-cost for writing the merged sequence is $O(z/B)$ because it occurs sequentially from the smallest to the largest item of $S[i, j]$ by using an auxiliary array. As a result the recurrent relation for the I/O-complexity of Mergesort can be written as $T(n) = 2T(n/2) + O(n/B) = O(\frac{n}{B} \log n)$ I/Os.

But this formula does not explain completely the good behavior of Mergesort in practice, because it does not account for the memory hierarchy yet. In fact as Mergesort recursively splits the sequence S, smaller and smaller sub-sequences are generated that have to be sorted. So when a subsequence of length z fits in internal memory, namely $z = O(M)$, then it is entirely cached by the underlying operating system using $O(z/B)$ I/Os and thus the subsequent sorting steps do not incur in any I/Os. The net result of this simple observation is that the I/O-cost of sorting a sub-sequence of $z = O(M)$ items is no longer $\Theta(\frac{z}{B} \log z)$, as accounted for in the previous recurrence relation, but it is $O(z/B)$ I/Os which accounts only the cost of loading the subsequence in internal memory. This saving applies to all S's subsequences of size $\Theta(M)$ on which Mergesort is recursively run, which are $\Theta(n/M)$ in total. So the overall saving is $\Theta(\frac{n}{B} \log M)$, which leads us to re-formulate the Mergesort's complexity as $\Theta(\frac{n}{B} \log \frac{n}{M})$ I/Os. This bound is particularly interesting because relates the I/O-complexity of Mergesort not only to the disk-page size B but also to the internal-memory size M, and thus to the *caching* available at the sorter. Moreover this bounds suggests three immediate optimizations to the classic pseudocode of Algorithm 5.1 that we discuss below.

5.1.1 Stopping recursion

The first optimization consists of introducing a threshold on the subsequence size, say $j - i < cM$, which triggers the stop of the recursion, the fetching of that subsequence entirely in internal-memory, and the application of an internal-memory sorter on this sub-sequence (see Figure 5.1). The value of the parameter c depends on the space-occupancy of the sorter, which must be guaranteed to work entirely in internal memory. As an example, c is 1 for in-place sorters such as Insertionsort and Heapsort, it is much close to 1 for Quicksort (because of its recursion), and it is less than 0.5 for Mergesort (because of the extra-array used by Merge). As a result, we should write cM instead of M into the I/O-bound above, because recursion is stopped at cM items: thus obtaining $\Theta(\frac{n}{B} \log \frac{n}{cM})$. This substitution is useless when dealing with asymptotic analysis, given that c is a constant, but it is important when considering the real performance of algorithms. In this setting it is desirable to make c as closer as possible to 1, in order to reduce the logarithmic factor in the I/O-complexity thus preferring in-place sorters such as Heapsort or Quicksort. We remark that Insertionsort could also be a good choice (and indeed it is) whenever M is small, as it occurs when considering the sorting of items over the 2-levels: L1 and L2 caches, and the internal memory. In this case M would be few Megabytes.

5.1.2 Snow Plow

Looking at the I/O-complexity of mergesort, i.e. $\Theta(\frac{n}{B}\log\frac{n}{M})$, is clear that the larger is M the smaller is the number of merge-passes over the data. These passes are clearly the bottleneck to the efficient execution of the algorithm especially in the presence of disks with low bandwidth. In order to circumvent this problem we can either buy a larger memory, or try to deploy as much as possible the one we have available. As algorithm engineer we opt for the second possibility and thus propose two techniques that can be combined together in order to enlarge (virtually) M.

The first technique is based on data compression and builds upon the observation that the runs are increasingly sorted. So, instead of representing items via a fixed-length coding (e.g. 4 or 8 bytes), we can use *integer compression* techniques that squeeze those items in fewer bits thus allowing us to pack more of them in internal memory. A following lecture will describe in detail several approaches to this problem (see Chapter **??**), here we content ourselves mentioning the names of some of these approaches: γ-code, δ-code, Rice/Golomb-coding, etc. etc.. In addition, since the smaller is an integer the fewer bits are used for its encoding, we can enforce the presence of small integers in the sorted runs by encoding not just their absolute value but the *difference* between one integer and the previous one in the sorted run (the so called *delta*-coding). This difference is surely non negative (equals zero if the run contains equal items), and smaller than the item to be encoded. This is the typical approach to the encoding of integer sequences used in modern search engines, that we will discuss in a following lecture (see Chapter **??**).

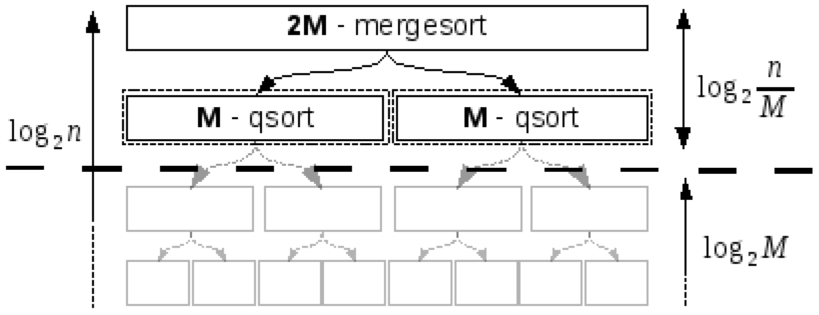


FIGURE 5.1: When a run fits in the internal memory of size M, we apply `qsort` over its items. In gray we depict the recursive calls that are executed in internal memory, and thus do not elicit I/Os. Above there are the calls based on classic Mergesort, only the call on $2M$ items is shown.

The second technique is based on an elegant idea, called the `Snow Plow` and due to D. Knuth [3], that allows to *virtually* increase the memory size of a factor 2 *on average*. This technique scans the input sequence S and generates sorted runs whose length has variable size longer than M and $2M$ on average. Its use needs to change the sorting scheme because it first creates these sorted runs, of variable length, and then applies repeatedly over the sorted runs the MERGE-procedure. Although runs will have different lengths, the MERGE will operate as usual requiring an optimal number of I/Os for their merging. Hence $O(n/B)$ I/Os will suffice to halve the number of runs, and thus a total of $O(\frac{n}{B}\log\frac{n}{2M})$ I/Os will be used on average to produce the totally ordered sequence. This corresponds to a saving of 1 pass over the data, which is non negligible if the sequence S is very long.

For ease of description, let us assume that items are transferred one at a time from disk to memory, instead that block-wise. Eventually, since the algorithm scans the input items it will be apparent that the number of I/Os required by this process is linear in their number (and thus optimal). The algorithm proceeds in phases, each phase generates a sorted run (see Figure 5.2 for an illustrative

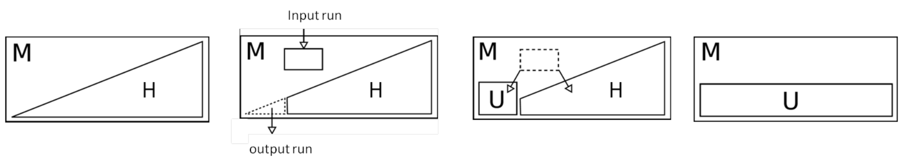


FIGURE 5.2: An illustration of four steps of a phase in Snow Plow. The leftmost picture shows the starting step in which $\mathcal{U}$ is heapified, then a picture shows the output of the minimum element in $\mathcal{H}$, hence the two possible cases for the insertion of the new item, and finally the stopping condition in which $\mathcal{H}$ is empty and $\mathcal{U}$ fills entirely the internal memory.

example). A phase starts with the internal-memory filled of M (unsorted) items, stored in a heap data structure called $\mathcal{H}$. Since the array-based implementation of heaps requires no additional space, in addition to the indexed items, we can fit in $\mathcal{H}$ as many items as we have memory cells available. The phase scans the input sequence S (which is unsorted) and at each step, it writes to the output the minimum item within $\mathcal{H}$, say `min`, and loads in memory the next item from S, say `next`. Since we want to generate a sorted output, we cannot store `next` in $\mathcal{H}$ if `next` < `min`, because it will be the new heap-minimum and thus it will be written out at the next step thus destroying the property of ordered run. So in that case `next` must be stored in an auxiliary array, called $\mathcal{U}$, which stays unsorted. Of course the total size of $\mathcal{H}$ and $\mathcal{U}$ is M over the whole execution of a phase. A phase stops whenever $\mathcal{H}$ is empty and thus $\mathcal{U}$ consists of M unsorted items, and the next phase can thus start (storing those items in a new heap $\mathcal{H}$ and emptying $\mathcal{U}$). Two observations are in order: (i) during the phase execution, the minimum of $\mathcal{H}$ is non decreasing and so it is non-decreasing also the output run, (ii) the items in $\mathcal{H}$ at the beginning of the phase will be eventually written to output which thus is longer than M. Observation (i) implies the correctness, observation (ii) implies that this approach is not less efficient than the classic Mergesort.

Algorithm 5.2 A phase of the Snow-Plow technique

```
Require: $\mathcal{U}$ is an unsorted array of $M$ items
 1: $\mathcal{H}$ = build a min-heap over $\mathcal{U}$'s items;
 2: Set $\mathcal{U} = \emptyset$;
 3: while ($\mathcal{H} \neq \emptyset$) do
 4:     min = Extract minimum from $\mathcal{H}$;
 5:     Write min to the output run;
 6:     next = Read the next item from the input sequence;
 7:     if (next < min) then
 8:         write next in $\mathcal{U}$;
 9:     else
10:         insert next in $\mathcal{H}$;
11:     end if
12: end while
```

Actually it is more efficient than that on average. Suppose that a phase reads τ items in total from S. By the while-guard in Step 3 and our comments above, we can derive that a phase ends when $\mathcal{H}$ is empty and $|\mathcal{U}| = M$. We know that the read items go in part in $\mathcal{H}$ and in part in $\mathcal{U}$. But since items are added to $\mathcal{U}$ and never removed during a phase, M of the τ items end-up in $\mathcal{U}$.

Consequently $(\tau - M)$ items are inserted in $\mathcal{H}$ and eventually written to the output (sorted) run. So the length of the sorted run at the end of the phase is $M + (\tau - M) = \tau$, where the first addendum accounts for the items in $\mathcal{H}$ at the beginning of a phase, whereas the second addendum accounts for the items read from S and inserted in $\mathcal{H}$ during the phase. The key issue now is to compute the average of τ. This is easy if we assume a random distribution of the input items. In this case we have probability $1/2$ that `next` is smaller than `min`, and thus we have equal probability that a read item is inserted either in $\mathcal{H}$ or in $\mathcal{U}$. Overall it follows that $\tau/2$ items go to $\mathcal{H}$ and $\tau/2$ items go to $\mathcal{U}$. But we already know that the items inserted in $\mathcal{U}$ are M, so we can set $M = \tau/2$ and thus we get $\tau = 2M$.

FACT 5.1 *Snow-Plow builds $O(n/M)$ sorted runs, each longer than M and actually of length $2M$ on average. Using Snow-Plow for the formation of sorted runs in a Merge-based sorting scheme, this achieves an I/O-complexity of $O(\frac{n}{B}\log_2\frac{n}{2M})$ on average.*

5.1.3 From binary to multi-way Mergesort

Previous optimizations deployed the internal-memory size M to reduce the number of recursion levels by increasing the size of the initial (sorted) runs. But then the merging was *binary* in that it fused two input runs at a time. This binary-merge impacted onto the base 2 of the logarithm of the I/O-complexity of Mergesort. Here we wish to increase that base to a much larger value, and in order to get this goal we need to deploy the memory M also in the merging phase by enlarging the number of runs that are fused at a time. In fact the merge of 2 runs uses only 3 blocks of the internal memory: 2 blocks are used to cache the current disk pages that contain the compared items, namely $S[x]$ and $S[y]$ from the notation above, and 1 block is used to cache the output items which are flushed when the block is full (so to allow a block-wise writing to disk of the merged run). But the internal memory contains a much larger number of blocks, i.e. $M/B \gg 3$, which remain unused over the whole merging process. The third optimization we propose, therefore consists of deploying all those blocks by designing a k-way merging scheme that fuses k runs at a time, with $k \gg 2$. Let us set $k = (M/B) - 1$, so that k blocks are available to read block-wise k input runs, and 1 block is reserved for a block-wise writing of the merged run to disk. This scheme poses a challenging merging problem because at each step we have to select the minimum among k candidates items and this cannot be obviously done brute-forcedly by iterating among them. We need a smarter solution that again hinges onto the use of a min-heap data structure, which contains k pairs (one per input run) each consisting of two components: one denoting an item and the other denoting the origin run. Initially the items are the minimum items of the k runs, and so the pairs have the form $\langle R_i[1], i\rangle$, where R_i denotes the ith input run and $i = 1, 2, \ldots, k$. At each step, we extract the pair containing the current smallest item in $\mathcal{H}$ (given by the first component of its pairs), write that item to output and insert in the heap the next item in its origin run. As an example, if the minimum pair is $\langle R_m[x], m\rangle$ then we write in output $R_m[x]$ and insert in $\mathcal{H}$ the new pair $\langle R_m[x+1], m\rangle$, provided that the mth run is not exhausted, in which case no pair replaces the extracted one. In the case that the disk page containing $R_m[x+1]$ is not cached in internal memory, an I/O-fault occurs and that page is fetched, thus guaranteeing that the next B reads from run R_m will not elicit any further I/O. It should be clear that this merging process takes $O(\log_2 k)$ time per item, and again $O(z/B)$ I/Os to merge k runs of total length z.

As a result the merging-scheme recalls a k-way tree with $O(n/M)$ leaves (runs) which can have been formed using any of the optimizations above (possibly via Snow Plow). Hence the total number of merging levels is now $O(\log_{M/B}\frac{n}{M})$ for a total volume of I/Os equal to $O(\frac{n}{B}\log_{M/B}\frac{n}{M})$. We observe that sometime we will also write the formula as $O(\frac{n}{B}\log_{M/B}\frac{n}{B})$, as it typically occurs in the literature, because $\log_{M/B} M$ can be written as $\log_{M/B}(B \times (M/B)) = (\log_{M/B} B) + 1 = \Theta(\log_{M/B} B)$. This makes no difference asymptotically given that $\log_{M/B} \frac{n}{M} = \Theta(\log_{M/B} \frac{n}{B})$.

THEOREM 5.1 *Multi-way Mergesort takes $O(\frac{n}{B} \log_{M/B} \frac{n}{M})$ I/Os and $O(n \log n)$ comparisons/-time to sort n atomic items in a two-level memory model in which the internal memory has size M and the disk page has size B. The use of Snow-Plow or integer compressors would virtually increase the value of M with a twofold advantage in the final I/O-complexity, because M occurs twice in the I/O-bound.*

In practice the number of merging levels will be very small: assuming a block size $B = 4$KB and a memory size $M = 4$GB, we get $M/B = 2^{32}/2^{12} = 2^{20}$ so that the number of passes is $1/20$th smaller than the ones needed by binary Mergesort. Probably more interesting is to observe that one pass is able to sort $n = M$ items, but two passes are able to sort M^2/B items, since we can merge M/B-runs each of size M. It goes without saying that in practice the internal-memory space which can be dedicated to sorting is smaller than the *physical* memory available (typically MBs versus GBs). Nevertheless it is evident that M^2/B is of the order of Terabytes already for $M = 128$MB and $B = 4$KB.

5.2 Lower bounds

At the beginning of this lecture we commented on the relation existing between the Sorting and the Permuting problems, concluding that the former one is more difficult than the latter in the RAM model. The gap in time complexity is given by a logarithmic factor. The question we address in this section is whether this gap does exist also when measuring I/Os. Surprisingly enough we will show that Sorting is *equivalent* to Permuting in terms of I/O-volume. This result is amazing because it can be read as saying that the I/O-cost for sorting is not in the *computation* of the sorted permutation but rather the *movement* of the data on the disk to realize it. This is the so called *I/O-bottleneck* that has in this result the mathematical proof and quantification.

Before digging into the proof of this lower bound, let us briefly show how a sorter can be used to permute a sequence of items $S[1, n]$ in accordance to a given permutation $\pi[1, n]$. This will allow us to derive an upper bound to the number of I/Os which suffice to solve the Permuting problem on any $\langle S, \pi\rangle$. Recall that this means to generate the sequence $S[\pi[1]], S[\pi[2]], \ldots, S[\pi[n]]$. In the RAM model we can jump among S's items according to permutation π and create the new sequence $S[\pi[i]]$, for $i = 1, 2, \ldots, n$, thus taking $O(n)$ optimal time. On disk we have actually two different algorithms which induce two incomparable I/O-bounds. The first algorithm consists of mimicking what is done in RAM, paying one I/O per moved item and thus taking $O(n)$ I/Os. The second algorithm consists of generating a proper set of tuples and then sort them. Precisely, the algorithm creates the sequence $\mathcal{P}$ of pairs $\langle i, \pi[i]\rangle$ where the first component indicates the position i where the item $S[\pi[i]]$ must be stored. Then it sorts these pairs according to the π-component, and via a parallel scan of S and $\mathcal{P}$ substitutes $\pi[i]$ with the item $S[\pi[i]]$, thus creating the new pairs $\langle i, S[\pi[i]]\rangle$. Finally another sort is executed according to the first component of these pairs, thus obtaining a sequence of items correctly permuted. The algorithm uses two scan of the data and two sorts, so it needs $O(\frac{n}{B} \log_{M/B} \frac{n}{M})$ I/Os.

THEOREM 5.2 *Permuting n items takes $O(\min\{n, \frac{n}{B} \log_{M/B} \frac{n}{M}\})$ I/Os in a two-level memory model in which the internal memory has size M and the disk page has size B.*

In what follows we will show that this algorithm, in its simplicity, is I/O-optimal. The two upper-bounds for Sorting and Permuting equal each other whenever $n = \Omega(\frac{n}{B} \log_{M/B} \frac{n}{M})$. This occurs when

$B > \log_{M/B} \frac{n}{M}$ that holds always in practice because that logarithm term is about 2 or 3 for values of n up to many Terabytes. So programmers should not be afraid to find sophisticated strategies for moving their data in the presence of a permutation, just sort them, you cannot do better!

	time complexity (RAM model)	I/O complexity (two-level memory model)
Permuting	$O(n)$	$O(\min\{n, \frac{n}{B} \log_{M/B} \frac{n}{M}\})$
Sorting	$O(n \log_2 n)$	$O(\frac{n}{B} \log_{\frac{M}{B}} \frac{n}{M})$

TABLE 5.1 Time and I/O complexities of the Permuting and Sorting problems in a two-level memory model in which M is the internal-memory size, B is the disk-page size, and $D = 1$ is the number of available disks. The case of multi-disks presents the multiplicative term n/D in place of n.

5.2.1 A lower-bound for Sorting

There are some subtle issues here that we wish to do not investigate too much, so we hereafter give only the intuition which underlies the lower-bounds for both Sorting and Permuting.[2] We start by resorting the comparison-tree technique for proving comparison-based lower bounds in the RAM model. An algorithm corresponds to a family of such trees, one per input size (so infinite in number). Every node is a comparison between two items. The comparison has two possible results, so the fan-out of each internal node is two and the tree is binary. Each leaf of the tree corresponds to a solution of the underlying problem to be solved: so in the case of sorting, we have one leaf per permutation of the input. Every root-to-leaf path in the comparison-tree corresponds to a computation, so the longest path corresponds to the worst-case number of comparisons executed by the algorithm. In order to derive a lower bound, it is therefore enough to compute the depth of the shallowest binary tree having that number of leaves. The shallowest binary tree with ℓ leaves is the (quasi-)perfectly balanced tree, for which the height h is such that $2^h \geq \ell$. Hence $h \geq \log_2 \ell$. In the case of sorting $\ell = n!$ so the classic lower bound $h = \Omega(n \log_2 n)$ is easily derived by applying logarithms at both sides of the equation and using the Stirling's approximation for the factorial.

In the two-level memory model the use of comparison-trees is more sophisticated. Here we wish to account for I/Os, and exploit the fact that the information available in the internal memory can be used for free. As a result every node corresponds to one I/O, the number of leaves equals still to $n!$, but the fan-out of each internal node equals to the *number of comparison-results* that this single I/O can generate among the items it reads from disk (i.e. B) and the items available in internal memory (i.e. $M - B$). These B items can be distributed in at most $\binom{M}{B}$ ways among the other $M - B$ items present in internal memory, so one I/O can generate no more than $\binom{M}{B}$ different results for those comparisons. But this is an incomplete answer because we are not considering the permutations among those items! However, some of these permutations have been already counted by some previous I/O, and thus we have not to recount them. These permutations are the ones concerning with items that have already passed through internal memory, and thus have been fetched by some previous I/O. So we have to count only the permutations among the *new* items, namely the ones

[2]There are two assumptions that are typically introduced in those arguments. One concerns with *item indivisibility*, so items cannot be broken up into pieces (hence hashing is not allowed!), and the other concerns with the possibility to *only move items* and not create/destroy/copy them, which actually implies that exactly one copy of each item does exist during their sorting or permuting.

that have never been considered by a previous I/O. We have n/B input pages, and thus n/B I/Os accessing new items. So these I/Os generate $\binom{M}{B}(B!)$ results by comparing those new B items with the $M - B$ ones in internal memory.

Let us now consider a computation with t I/Os, and thus a path in the comparison-tree with t nodes. n/B of those nodes must access the input items, which must be surely read to generate the final permutation. The other $t - \frac{n}{B}$ nodes read pages containing already processed items. Any root-to-leaf path has this form, so we can look at the comparison tree as having the new-I/Os at the top and the other nodes at its bottom. Hence if the tree has depth t, its number of leaves is at least $\binom{M}{B}^t \times (B!)^{n/B}$. By imposing that this number is $\geq n!$, and applying logarithms to both members, we derive that $t = \Omega(\frac{n}{B} \log_{M/B} \frac{n}{M})$. It is not difficult to extend this argument to the case of D disks thus obtaining the following.

THEOREM 5.3 *In a two-level memory model with internal memory of size M, disk-page size B and D disks, a comparison-based sorting algorithm must execute* $\Omega(\frac{n}{DB} \log_{M/B} \frac{n}{DB})$ *I/Os.*

It is interesting to observe that the number of available disks D does not appear in the denominator of the base of the logarithm, although it appears in the denominator of all other terms. If this would be the case, instead, D would somewhat penalize the sorting algorithms because it would reduce the logarithm's base. In the light of Theorem 5.1, multi-way Mergesort is I/O and time optimal on one disk, so D linearly boosts its performance thus having more disks is *linearly* advantageous (at least from a theoretical point of view). But Mergesort is no longer optimal on multi-disks because the simultaneous merging of $k > 2$ runs, should take $O(n/DB)$ I/Os in order to be optimal. This means that the algorithm should be able to fetch D pages per I/O, hence one per disk. This cannot be guaranteed, at every step, by the current merging-scheme because whichever is the distribution of the k runs among the D disks, and even if we know which are the next DB items to be loaded in the heap $\mathcal{H}$, it could be the case that more than B of these items reside on the same disk thus requiring more than one I/O from that disk, hence preventing the parallelism in the read operation.

In the following Section 5.4 we will address this issue by proposing the *disk striping* technique, that comes close to the I/O-optimal bound via a simple data layout on disks, and the *Greedsort* algorithm that achieves full optimality by devising an elegant and sophisticated merging scheme.

5.2.2 A lower-bound for Permuting

Let us assume that at any time the memory of our model, hence the internal memory of size M and the unbounded disk, contains a permutation of the input items possibly interspersed by empty cells. No more than n blocks will be non empty during the execution of the algorithm, because n steps (and thus I/Os) is an obvious upper bound to the I/O-complexity of Permuting (obtained by mimicking on disk the Permuting algorithm for the RAM model). We denote by P_t the number of permutations generated by an algorithm with t I/Os, where $t \leq n$ and $P_0 = 1$ since at the beginning we have the input order as initial permutation. In what follows we estimate P_t and then set $P_t \geq n!$ in order to derive the minimum number of steps t needed to realize any possible permutation given in input. Permuting is different from Sorting because the permutation to be realized is provided in input, and thus we do not need any computation. So in this case we distinguish three types of I/Os, which contribute differently to the number of generated permutations:

Write I/O: This may increase P_t by a factor $O(n)$ because we have at most $n+1$ possible ways to write the output page among the at most n not-empty pages available on disk. Any written page is "touched", and they are no more than n at any instant of the permuting process.

Read I/O on an untouched page: If the page was an input page never read before, the read operation imposes to account for the permutations among the read items, hence $B!$ in number, and to account also for the permutations that these B items can realize by distributing them among the $M - B$ items present in internal memory (similarly as done for Sorting). So this read I/O can increase P_t by a factor $O(\binom{M}{B}(B!))$. The number of input (hence "untouched") pages is n/B. After a read I/O, they become "touched".

Read I/O on a touched page: If the page was already read or written, we already accounted in P_t for the permutations among its items, so this read I/O can increase P_t only by a factor $O(\binom{M}{B})$ due to the shuffling of the B read items with the $M - B$ ones present in internal memory. The number of touched pages is at most n.

If t_r is the number of reads and t_w is the number of writes executed by a Permuting algorithm, where $t = t_r + t_w$, then we can bound P_t as follows (here big-Oh have been dropped to ease the reading of the formulas):

$$P_t \;\leq\; (\frac{n}{B}\binom{M}{B}(B!))^{n/B} \times (n\binom{M}{B})^{t_r - n/B} \times n^{t_w} \;\leq\; (n\binom{M}{B})^{t}(B!)^{\frac{n}{B}}$$

In order to generate every possible permutation of the n input items, we need that $P_t \geq n!$. We can thus derive a lower bound on t by imposing that $n! \leq (n\binom{M}{B})^t\ (B!)^{\frac{n}{B}}$ and resolving with respect to t:

$$t = \Omega(\frac{n \log \frac{n}{B}}{B \log \frac{M}{B} + \log n})$$

We distinguish two cases. If $B \log \frac{M}{B} \leq \log n$, then the above equation becomes $t = \Omega(\frac{n \log \frac{n}{B}}{\log n}) = \Omega(n)$; otherwise it is $t = \Omega(\frac{n \log \frac{n}{B}}{B \log \frac{M}{B}}) = \Omega(\frac{n}{B} \log_{\frac{M}{B}} \frac{n}{M})$. As for sorting, it is not difficult to extend this proof to the case of D disks.

THEOREM 5.4 *In a two-level memory model with internal memory of size M, disk-page size B and D disks, permuting n items needs* $\Omega(\min\{\frac{n}{D}, \frac{n}{DB} \log_{M/B} \frac{n}{DB}\})$ *I/Os.*

Theorems 5.2–5.4 prove that the I/O-bounds provided in Table 5.1 for the Sorting and Permuting problems are optimal. Comparing these bounds we notice that they are asymptotically different whenever $B \log \frac{M}{B} < \log n$. Given the current values for B and M, respectively few KBs and few GBs, this inequality holds if $n = \Omega(2^B)$ and hence when n is much more than Yottabytes ($= 2^{80}$). This is indeed an unreasonable situation to deal with one CPU and few disks. Probably in this context it would be more reasonable to use a *cloud* of PCs, and thus analyze the proposed algorithms via a *distributed* model of computation which takes into account many CPUs and more-than-2 memory levels. It is therefore not surprising that researchers typically assume `Sorting = Permuting` in the I/O-setting.

5.3 The distribution-based sorting paradigm

Like Mergesort, Quicksort is based on the divide&conquer paradigm, so it proceeds by dividing the array to be sorted into two pieces which are then sorted recursively. But unlike Mergesort, Quicksort does not explicitly allocate *extra*-working space, its *combine*-step is absent and its *divide*-step is sophisticated and impacts onto the overall efficiency of this sorting algorithm. Algorithm 5.3

Algorithm 5.3 The binary quick-sort: QuickSort(S, i, j)

```
1: if ($i < j$) then
2:     $r$ = pick the position of a "good pivot";
3:     swap $S[r]$ with $S[i]$;
4:     $p$ = Partition($S, i, j$);
5:     QuickSort($S, i, p-1$);
6:     QuickSort($S, p+1, j$);
7: end if
```

reports the pseudocode of Quicksort, this will be used to comment on its complexity and argue for some optimizations or tricky issues which arise when implementing it over hierarchical memories.

The key idea is to partition the input array $S[i, j]$ in two pieces such that one contains items which are *smaller (or equal)* than the items contained in the latter piece. This partition is order preserving because no subsequent steps are necessary to recombine the ordered pieces after the two recursive calls. Partitioning is typically obtained by selecting one input item as a *pivot*, and by distributing all the other input items into two sub-arrays according to whether they are smaller/greater than the pivot. Items equal to the pivot can be stored anywhere. In the pseudocode the pivot is forced to occur in the first position $S[i]$ of the array to be sorted (steps 2–3): this is obtained by swapping the real pivot $S[r]$ with $S[i]$ before that procedure Partition(S, i, j) is invoked. We notice that step 2 does not detail the selection of the pivot, because this will be the topic of a subsequent section.

There are two issues for achieving efficiency in the execution of Quicksort: one concerns with the implementation of Partition(S, i, j), and the other one with the ratio between the size of the two formed pieces because the more *balanced* they are, the more Quicksort comes closer to Mergesort and thus to the optimal time complexity of $O(n \log n)$. In the case of a totally unbalanced partition, in which one piece is possibly empty (i.e. $p = i$ or $p = j$), the time complexity of Quicksort is $O(n^2)$, thus recalling in its cost the Insertion sort. Let us comment these two issues in detail in the following subsections.

5.3.1 From two- to three-way partitioning

The goal of Partition(S, i, j) is to divide the input array into two pieces, one contains items which are smaller than the pivot, and the other contains items which are larger than the pivot. Items equal to the pivot can be arbitrarily distributed among the two pieces. The input array is therefore permuted so that the smaller items are located before the pivot, which in turn precedes the larger items. At the end of Partition(S, i, j), the pivot is located at $S[p]$, the smaller items are stored in $S[i, p-1]$, the larger items are stored in $S[p+1, j]$. This partition can be implemented in many ways, taking $O(n)$ optimal time, but each of them offers a different cache usage and thus different performance in practice. We present below a tricky algorithm which actually implements a *three-way* distribution and takes into account the presence of items equal to the pivot. They are detected and stored aside in a "special" sub-array which is located between the two smaller/larger pieces.

It is clear that the central sub-array, which contains items equal to the pivot, can be discarded from the subsequent recursive calls, similarly as we discard the pivot. This reduces the number of items to be sorted recursively, but needs a change in the (classic) pseudo-code of Algorithm 5.3, because Partition must now return the pair of indices which delimit the central sub-array instead of just the position p of the pivot. The following Algorithm 5.4 details an implementation for the three-way partitioning of $S[i, j]$ which uses three pointers that move rightward over this array and maintain the following invariant: P is the pivot driving the three-way distribution, $S[c]$ is the item currently compared against P, and $S[i, c-1]$ is the part of the input array already scanned and three-way partitioned in its elements. In particular $S[i, c-1]$ consists of three parts: $S[i, l-1]$ contains items smaller than P, $S[l, r-1]$ contains items equal to P, and $S[r, c-1]$ contains items larger than P. It may be the case that anyone of these sub-arrays is empty.

Algorithm 5.4 The three-way partitioning: PARTITION(S, i, j)

```
1:  $P = S[i]; l = i; r = i + 1;$
2:  for ($c = r; c \leq j; c{+}{+}$) do
3:      if ($S[c] == P$) then
4:          swap $S[c]$ with $S[r]$;
5:          $r{+}{+}$;
6:      else if ($S[c] < P$) then
7:          swap $S[c]$ with $S[l]$;
8:          swap $S[c]$ with $S[r]$;
9:          $r{+}{+}; l{+}{+}$;
10:     end if
11: end for
12: return $\langle l, r-1 \rangle$;
```

Step 1 initializes P to the first item of the array to be partitioned (which is the pivot), l and r are set to guarantee that the smaller/greater pieces are empty, whereas the piece containing items equal to the pivot consists of the only item P. Next the algorithm scans $S[i+1, j]$ trying to maintain the invariant above. This is easy if $S[c] > P$, because it suffices to extend the part of the larger items by advancing r. In the other two cases (i.e. $S[c] \leq P$) we have to insert $S[c]$ in its correct position among the items of $S[i, r-1]$, in order to preserve the invariant on the three-way partition of $S[i, c]$. The cute idea is that this can be implemented in $O(1)$ time by means of at most two swaps, as described graphically in Figure 5.3.

The three-way partitioning algorithm takes $O(n)$ time and offers two positive properties: (i) stream-like access to the array S which allows the pre-fetching of the items to be read; (ii) the items equal to the pivot can then be eliminated from the following recursive calls. A last note concerns with the pair of indices $\langle l, r-1 \rangle$ returned by PARTITION(S, i, j): they delimit the part of S which consists of elements equal to P, and thus they can be dropped from the subsequent recursive calls, being them in the final correct positions.

5.3.2 Pivot selection

The selection of the pivot is crucial to get balanced partitions, reduce the number of recursive calls, and achieve optimal $O(n \log n)$ time complexity. The pseudo-code of Algorithm 5.3 does not detail the way the pivot is selected because this may occur in many different ways, each offering pros/cons. As an example, if we choose the pivot as the first item of the input array (namely $r = i$), the selection is fast but it is easy to instantiate the input array in order to induce un-balanced partitions: just take S to be an increasing or decreasing ordered sequence of items. Worse than this, it is the observation that any deterministic choice incurs in this drawback.

One way to circumvent bad inputs is to select the pivot *randomly* among the items in $S[i, j]$. This prevents the case that a given input is bad for Quicksort, but makes the behavior of the algorithm un-predictable in advance and dependant on the random selection of the pivot. We can show that the *average* time complexity is the optimal $O(n \log_2 n)$, with an hidden constant small and equal to 1.39. This fact, together with the in-place nature of Quicksort, makes this approach much appealing in practice (cfr `qsort` below).

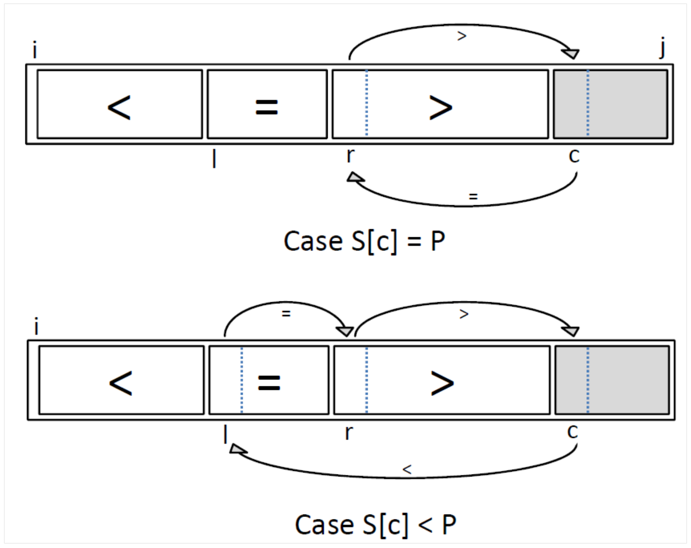


FIGURE 5.3: The two cases and the corresponding swapping. On the arrow we specify the value of the moved item with respect to the pivot.

THEOREM 5.5 *The random selection of the pivot drives Quicksort to compare no more than* $2n \ln n$ *items, on average.*

Proof The proof is deceptively simple if attacked from the correct angle. We wish to compute the number of comparisons executed by Partition over the sequence S. Let $X_{u,v}$ be the random binary variable which indicates whether $S[u]$ and $S[v]$ are compared by Partition, and denote by $p_{u,v}$ the probability that this event occurs. The average number of comparisons executed by Quicksort can then be computed as $E[\sum_{u,v} X_{u,v}] = \sum_u \sum_{v>u} 1 \times p_{u,v} + 0 \times (1 - p_{u,v}) = \sum_{u=1}^{n} \sum_{v=u+1}^{n} p_{u,v}$ by linearity of expectation.

To estimate $p_{u,v}$ we concentrate on the random choice of the pivot $S[r]$ because two items are compared by Partition only if one of them is the pivot. So we distinguish three cases. If $S[r]$ is smaller or larger than both $S[u]$ and $S[v]$, then the two items $S[u]$ and $S[v]$ are not compared to each other and they are passed to the same recursive call of Quicksort. So the problem presents itself again on a smaller subset of items containing both $S[u]$ and $S[v]$. This case is therefore not interesting for estimating $p_{u,v}$, because we cannot conclude anything at this recursive-point about the execution or not of the comparison between $S[u]$ and $S[v]$. In the case that either $S[u]$ or $S[v]$ is the pivot, then they are compared by Partition. In all other cases, the pivot is taken among the items of S whose value is between $S[u]$ and $S[v]$; so these two items go to two different partitions (hence two different recursive calls of Quicksort) and will never be compared.

As a result, to compute $p_{u,v}$ we have to consider as interesting pivot-selections the two last situations. Among them, two are the ”good” cases, but how many are all these interesting pivot-selections? We need to consider the sorted S, denoted by S'. There is an obvious bijection between pairs of items in S' and pairs of items in S. Let us assume that $S[u]$ is mapped to $S'[u']$ and $S[v]$ is mapped to $S'[v']$, then it is easy to derive the number of interesting pivot-selections as $v' - u' + 1$ which corresponds to the number of items (hence pivot candidates) whose value is between $S[u]$ and $S[v]$ (extremes included). So $p_{u,v} = 2/(v' - u' + 1)$.

This formula may appear complicate because we have on the left u, v and on the right u', v'. Given the bijection between S and S', We can rephrase the statement "considering all pairs (u, v) in S" as "considering all pairs (u', v') in S'", and thus write:

$$E[\sum_{u,v} X_{u,v}] = \sum_{u=1}^{n} \sum_{v=u+1}^{n} p_{u,v} = \sum_{u'=1}^{n} \sum_{v'>u'}^{n} \frac{2}{v' - u' + 1} = 2 \sum_{u'=1}^{n} \sum_{k=2}^{n-u'+1} \frac{1}{k} \leq 2 \sum_{u'=1}^{n} \sum_{k=2}^{n} \frac{1}{k} \leq 2n \ln n$$

where the last inequality comes from the properties of the n-th harmonic number, namely $\sum_{k=1}^{n} \frac{1}{k} \leq 1 + \ln n$. ■

The next question is how we can enforce the average behavior. The natural answer is to sample more than one pivot. Typically 3 pivots are randomly sampled from S and the central one (i.e. the median) is taken, thus requiring just two comparisons in $O(1)$ time. Taking more than 3 pivots makes the selection of a "good one" more robust, as proved in the following theorem [2].

THEOREM 5.6 *If Quicksort partitions around the median of $2s+1$ randomly selected elements, it sorts n distinct elements in $\frac{2nH_n}{H_{2s+2}-H_{s+1}} + O(n)$ expected comparisons, where H_z is the z-th harmonic number $\sum_{i=1}^{z} \frac{1}{i}$.*

By increasing s, we can push the expected number of comparisons close to $n \log n + O(n)$, however the selection of the median incurs a higher cost. In fact this can be implemented either by sorting the s samples in $O(s \log s)$ time and taking the one in the middle position $s + 1$ of the ordered sequence; or in $O(s)$ worst-case time via a sophisticated algorithm (not detailed here). Randomization helps in simplifying the selection still guaranteeing $O(s)$ time on average. We detail this approach here because its analysis is elegant and its structure general enough to be applied not only for the selection of the median of an unordered sequence, but also for selecting the item of any rank k.

Algorithm 5.5 Selecting the k-th ranked item: RandSelect(S, k)

```
1:  $r$ = pick a random item from $S$;
2:  $S_<$ = items of $S$ which are smaller than $S[r]$;
3:  $S_>$ = items of $S$ which are larger than $S[r]$;
4:  $n_< = |S_<|$;
5:  $n_= = |S| - (|S_<| + |S_>|)$;
6:  **if** $(k \leq n_<)$ **then**
7:      **return** RandSelect($S_<, k$);
8:  **else if** $(k \leq (n_< + n_=))$ **then**
9:      **return** $S[r]$;
10: **else**
11:     **return** RandSelect($S_>, k - n_< - n_=$);
12: **end if**
```

Algorithm 5.5 is randomized and selects the item of the unordered S having rank k. It is interesting to see that the algorithmic scheme mimics the one used in the Partitioning phase of Quicksort: here the selected item $S[r]$ plays the same role of the pivot in Quicksort, because it is used to partition the input sequence S in three parts consisting of items smaller/equal/larger than $S[r]$. But unlike Quicksort, RandSelect recurses only in one of these three parts, namely the one containing the k-th ranked item. This part can be determined by just looking at the sizes of those parts, as done in Steps 6 and 8. There are two specific issues that deserve a comment. We do not need to recurse on $S_=$ because it consists of items equal to $S[r]$. If recursion occurs on $S_>$, we need to update the rank k because we are dropping from the original sequence the items belonging to the set $S_< \cup S_=$. Correctness is therefore immediate, so we are left with computing the average time complexity of this algorithm which turns to be the optimal $O(n)$, given that S is unsorted and thus all of its n items have to be examined to find the one having rank k among them.

THEOREM 5.7 *Selecting the k-th ranked item in an unordered sequence of size n takes $O(n)$ average time in the RAM model, and $O(n/B)$ I/Os in the two-level memory model.*

Proof Let us call "good selection" the one that induces a partition in which $n_<$ and $n_>$ are not larger than $2n/3$. We do not care of the size of $S_=$ since, if it contains the searched item, that item is returned immediately as $S[r]$. It is not difficult to observe that $S[r]$ must have rank in the range $[n/3, 2n/3]$ in order to ensure that $n_< \leq 2n/3$ and $n_> \leq 2n/3$. This occurs with probability 1/3, given that $S[r]$ is drawn uniformly at random from S. So let us denote by $\hat{T}(n)$ the average time complexity of RandSelect when run on an array $S[1, n]$. We can write

$$\hat{T}(n) \leq O(n) + \frac{1}{3} \times \hat{T}(2n/3) + \frac{2}{3} \times \hat{T}(n),$$

where the first term accounts for the time complexity of Steps 2-5, the second term accounts for the average time complexity of a recursive call on a "good selection", and the third term is a crude upper bound to the average time complexity of a recursive call on a "bad selection" (that is actually assumed to recurse on the entire S again). This is not a classic recurrent relation because the term $\hat{T}(n)$ occurs on both sides; nevertheless, we observe that this term occurs with different constants in the front. Thus we can simplify the relation by subtracting those terms, so getting $\frac{1}{3}\hat{T}(n) \leq O(n) + \frac{1}{3}\hat{T}(2n/3)$, which gives $\hat{T}(n) = O(n) + \hat{T}(2n/3) = O(n)$. If this algorithm is executed in the two-level memory model, the equation becomes $\hat{T}(n) = O(n/B) + \hat{T}(2n/3) = O(n/B)$ given that the construction of the three subsets can be done via a single pass over the input items. ■

We can use RandSelect in many different ways within Quicksort. For example, we can select the pivot as the median of the entire array S (setting $k = n/2$) or the median among an over-sampled set of $2s + 1$ pivots (setting $k = s + 1$, where $s \ll n/2$), or finally, it could be subtly used to select a pivot that generates a balanced partition in which the two parts have different sizes both being a fraction of n, say αn and $(1 - \alpha)n$ with $\alpha < 0.5$. This last choice $k = \lfloor \alpha n \rfloor$ seems meaningless because the three-way partitioning still takes $O(n)$ time but increases the number of recursive calls from $\log_2 n$ to $\log_{\frac{1}{1-\alpha}} n$. But this observation neglects the sophistication of modern CPUs which are parallel, pipelined and superscalar. These CPUs execute instructions in parallel, but if there is an event that impacts on the instruction flow, their parallelism is *broken* and the computation slows down significantly. Particularly slow are *branch mispredictions*, which occur in the execution of Partition(S, i, j) whenever an item smaller than or equal to the pivot is encountered. If we reduce these cases, then we reduce the number of branch-mispredictions, and thus deploy the full parallelism of modern CPUs. Thus the goal is to properly set α in a way that the reduced number of mispredictions balances the increased number of recursive calls. The right value for α is clearly architecture dependent, recent results have shown that a reasonable value is 0.1.

5.3.3 Bounding the extra-working space

QuickSort is frequently named as an *in-place* sorter because it does not use extra-space for ordering the array S. This is true if we limit ourself to the pseudocode of Algorithm 5.3, but it is no longer true if we consider the cost of managing the recursive calls. In fact, at each recursive call, the OS must allocate space to save the local variables of the caller, in order to retrieve them whenever the recursive call ends. Each recursive call has a space cost of $\Theta(1)$ which has to be multiplied by the number of nested calls Quicksort can issue on an array $S[1, n]$. This number can be $\Omega(n)$ in the worst case, thus making the extra-working space $\Theta(n)$ on some bad inputs (such as the already sorted ones, pointed out above).

Algorithm 5.6 The binary quick-sort with bounded recursive-depth: BoundedQS(S, i, j)

```
1: while ($j - i > n_0$) do
2:     $r$ = pick the position of a "good pivot";
3:     swap $S[r]$ with $S[i]$;
4:     $p$ = Partition($S, i, j$);
5:     if ($p \leq \frac{i+j}{2}$) then
6:         BoundedQS($S, i, p - 1$);
7:         $i = p + 1$;
8:     else
9:         BoundedQS($S, p + 1, j$);
10:        $j = p - 1$;
11:    end if
12: end while
13: InsertionSort($S, i, j$);
```

We can circumvent this behavior by restructuring the pseudocode of Algorithm 5.3 as specified in Algorithm 5.6. This algorithm is cryptic at a first glance, but the underlying design principle is pretty smart and elegant. First of all we notice that the while-body is executed only if the input array is longer than n_0, otherwise Insertion-sort is called in Step 13, thus deploying the well-known efficiency of this sorter over very small sequences. The value of n_0 is typically chosen of few tens of items. If the input array is longer than n_0, a modified version of the classic binary Quicksort is executed that mixes one single recursive call with an iterative while-loop. The ratio underlying this code re-factoring is that the correctness of classic Quicksort does not depend on the order of the two recursive calls, so we can reshuffle them in such a way that the first call is always executed on the smaller part of the two/three-way partition. This is exactly what the IF-statement in step 5 guarantees. In addition to that, the pseudo-code above drops the recursive call onto the larger part of the partition in favor of another execution of the body of the while loop in which we properly changed the parameters i and j to reflect the new extremes of that larger part. This "change" is well-known in the literature of compilers with the name of *elimination of tail recursion*. The net result is that the recursive call is executed on a sub-array whose size is no more than the half of the input array. This guarantees an upper bound of $O(\log_2 n)$ on the number of recursive calls, and thus on the size of the extra-space needed to manage them.

THEOREM 5.8 BoundedQS *sorts n atomic items in the RAM model taking $O(n \log n)$ average time, and using $O(\log n)$ additional working space.*

We conclude this section by observing that the C89 and C99 ANSI standards define a sorting algorithm, called `qsort`, whose implementation encapsulates most of the algorithmic tricks detailed above.[3] This witnesses further the efficiency of the distribution-based sorting scheme over the 2-levels: cache and DRAM.

5.3.4 From binary to multi-way Quicksort

Distribution-based sorting is the *dual* of merge-based sorting in that the first proceeds by splitting sequences according to pivots and then ordering them recursively, while the latter merges sequences which have been ordered recursively. Disk-efficiency was obtained in Multi-way Mergesort by managing (fusing) multiple sequences together. The same idea is applied to design the Multi-way Quicksort which splits the input sequence into $k = \Theta(M/B)$ sub-sequences by using $k - 1$ pivots. Given that $k \gg 1$ the selection of those pivots is not a trivial task because it must ensure that the k partitions they form, are *balanced* and thus contain $\Theta(n/k)$ items each. Section 5.3.2 discussed the difficulties underlying the selection of one pivot, so the case of selecting many pivots is even more involved and needs a sophisticated analysis.

We start with denoting by $s_1, \ldots, s_{k-1}$ the pivots used by the algorithm to split the input sequence $S[1, n]$ in k parts, also called *buckets*. For the sake of clarity we introduce two dummy pivots $s_0 = -\infty$ and $s_k = +\infty$, and denote the i-th bucket by $B_i = \{S[j] : s_{i-1} < S[j] \leq s_i\}$. We wish to guarantee that $|B_i| = \Theta(n/k)$ for all the k buckets. This would ensure that $\log_k \frac{n}{M}$ partitioning phases are enough to get sub-sequences shorter than M, which can thus be sorted in internal-memory without any further I/Os. Each partitioning phase can be implemented in $O(n/B)$ I/Os by using a memory organization which is the dual of the one employed for Mergesort: namely, 1 input block (used to read from the input sequence to be partitioned) and k output blocks (used to write into the k partitions under formation). By imposing $k = \Theta(M/B)$, we derive that the number of partitioning phases is $\log_k \frac{n}{M} = \Theta(\log_{M/B} \frac{n}{M})$ so that the Multi-way Quicksort takes the optimal I/O-bound of $\Theta(\frac{n}{B} \log_{M/B} \frac{n}{M})$, provided that each partitioning step *distributes evenly* the input items among the k buckets.

To find efficiently k good pivots, we deploy a fast and simple randomized strategy based on *oversampling*, whose pseudocode is given in Algorithm 5.7 below. Parameter $a \geq 0$ controls the amount of oversampling and thus impacts onto the robustness of the selection process as well as on the cost of Step 2. The latter cost is $O((ak) \log(ak))$ if we adopt an optimal in-memory sorter, such as Heapsort or Mergesort, to sort the $\Theta(ak)$ sampled items.

Algorithm 5.7 Selection of $k - 1$ good pivots via oversampling

```
1: Take $(a + 1)k - 1$ samples at random from the input sequence;
2: Sort them into an ordered sequence $A$;
3: For $i = 1, \ldots, k - 1$, pick the pivot $s_i = A[(a + 1)i]$;
4: **return** the pivots $s_i$;
```

The main idea is to select $\Theta(ak)$ candidate pivots from the input sequence and then pick k among them, namely the ones which are evenly spaced and thus $(a + 1)$ far apart from each other. We are arguing that those $\Theta(ak)$ samples provide a faithful picture of the distribution of the items in the

[3] Actually `qsort` is based on a different two-way partitioning scheme that uses two iterators, one moves forward and the other one moves backward over S; a swap occurs whenever two un-sorted items are encountered. The asymptotic time complexity does not change, but practical efficiency can spur from the fact that the number of swaps is reduced since equal items are not moved.

entire input sequence, so that the balanced selection $s_i = A[(a+1)i]$ should provide us with "good pivots". The larger is a the closer to $\Theta(n/k)$ should be the size of all buckets, but the higher would be the cost of sorting the samples. At the extreme case of $a = n/k$, the samples could not be sorted in internal memory! On the other hand, the closer a is to zero the faster would be the pivot selection but more probable is to get unbalanced partitions. As we will see in the following Lemma 5.1, choosing $a = \Theta(\log k)$ is enough to obtain balanced partitions with a pivot-selection cost of $O(k \log^2 k)$ time. We notice that the buckets will be not perfectly balanced but quasi-balanced, since they include no more than $\frac{4n}{k} = O(n/k)$ items; the factor 4 will nonetheless leave unchanged the aimed asymptotic time complexity.

LEMMA 5.1 Let $k \geq 2$ and $a + 1 = 12 \ln k$. A sample of size $(a+1)k - 1$ suffices to ensure that all buckets receive less than $4n/k$ elements, with probability at least 1/2.

Proof We provide an upper bound of 1/2 to the probability of the complement event stated in the Lemma, namely that there exists one bucket whose size is larger than $4n/k$. This corresponds to a *failure* sampling, which induces an un-balanced partition. To get this probability estimate we will introduce a cascade of events that are implied by this one and thus have larger and larger probabilities to occur. For the last one in the sequence we will be able to fix an explicit upper-bound of 1/2. Given the implications, this upper bound will also hold for the original event. And so we will be done.

Let us start by considering the sorted version of the input sequence S, which hereafter we denote by S'. We logically split S' in $k/2$ segments of length $2n/k$ each. The event we are interested in is that there exists a bucket B_i with at least $4n/k$ items assigned to it. As illustrated in Figure 5.4 this large bucket completely spans at least one segment, say t_2 in the Figure below, because the former contains $\geq 4n/k$ items whereas the latter contains $2n/k$ items.

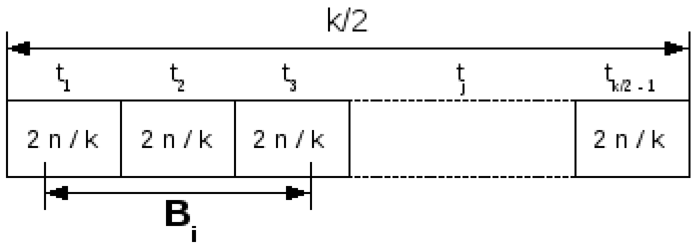


FIGURE 5.4: Splitting of the sorted sequence S' into segments.

By definition of the buckets, the pivots s_{i-1} and s_i which delimit B_i fall outside t_2. Hence, by Algorithm 5.7, less that $(a+1)$ samples fall in the segment overlapped by B_i. In the figure it is t_2, but it might be any segment of S'. So we have that:

$$\begin{aligned} \mathcal{P}(\exists B_i : \ |B_i| \geq 4n/k) &\leq \mathcal{P}(\exists t_j : \ t_j \text{ contains } < (a+1) \text{ samples}) \\ &\leq \frac{k}{2} \times \mathcal{P}(\text{a specific segment contains } < (a+1) \text{ samples}) \end{aligned} \quad (5.1)$$

where the last inequality comes from the *union bound*, given that $k/2$ is the number of segments constituting S'. So we will hereafter concentrate on providing an upper bound to the last term.

The probability that one sample ends in a given segment is equal to $\frac{(2n/k)}{n} = \frac{2}{k}$ because they are assumed to be drawn uniformly at random from S (and thus from S'). So let us call X the number of those samples, we are interested in computing $\mathcal{P}(X < a+1)$. We start by observing that $E[X] = ((a+1)k-1) \times \frac{2}{k} = 2(a+1) - \frac{2}{k}$. The Lemma assumes that $k \geq 2$, so $E[X] \geq 2(a+1) - 1$ which is $\geq \frac{3}{2}(a+1)$ for all $a \geq 1$. We can thus state that $a+1 \leq (2/3)E[X] = (1-\frac{1}{3})E[X]$. This form is useful to resort the Chernoff bound:

$$\mathcal{P}(X < (1-\delta)E[X]) \leq e^{-\frac{\delta^2}{2}E[X]}$$

By setting $\delta = 1/3$, we derive

$$\begin{aligned}\mathcal{P}(X < a+1) &\leq \mathcal{P}(X < (1-\frac{1}{3})E[X]) \leq e^{-(E[X]/2)(1/3)^2} = e^{-E[X]/18} \\ &\leq e^{-(3/2)(a+1)/18} = e^{-(a+1)/12} = e^{-\ln k} = \frac{1}{k} \qquad (5.2)\end{aligned}$$

where we used the inequality $E[X] \geq (3/2)a+1$ and the lemma's assumption that $a+1 = 12\ln k$. By plugging this value in Eqn 5.1, we get the statement of the Lemma. ∎

5.3.5 The Dual Pivot Quicksort

Very recently, in 2012, a new Quicksort variant due to Yaroslavskiy was chosen as the standard sorting method for Oracle's Java 7 runtime library.[4] The decision for the change was based on empirical studies showing that, on average, his new algorithm is faster than the formerly used classic Quicksort. The improvement was achieved by using a new three-way partition strategy based on a pair of pivots *properly moved* over the input sequence S. Surprisingly, this algorithmic scheme was considered not promising by several theoretical studies in the past. Instead, authors of [7] showed that this improvement is due to a reduction in the number of comparisons, about $1.9n\ln n$, at the expenses of an increase in the number of swaps, about $0.6n\ln n$. Despite this trade-off, the dual-pivot strategy results more than 10% faster, probably because branch mispredictions are more costly than memory accesses in modern PC architectures, as we commented earlier in this chapter.[5]

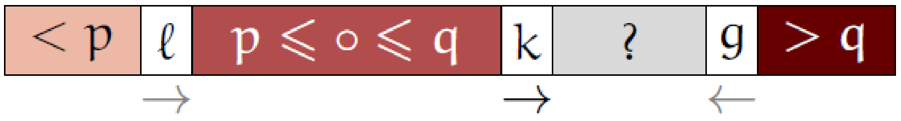


FIGURE 5.5: The invariant guaranteed by the Dual Pivot Quicksort

Figure 5.5 provides a pictorial representation of the Invariant maintained by the Dual Pivot Quicksort during the partitioning step. The algorithm uses two pivots– namely, p and q–, and three iterators– namely, ℓ, k and g– which partition the input sequence in four pieces.

[4]The discussion is archived at `http://permalink.gmane.org/gmane.comp.java.openjdk.core-libs.devel/2628`
[5]Studies about the selection of the pivots and the reason for this improvement are yet underway.

- The leftmost piece, delimited by ℓ, includes all items which are smaller that p;
- the mid-left piece, delimited between positions ℓ and k, includes items that are larger than p and smaller than q;
- the mid-right piece, delimited between positions k and g, includes items that have still to be examined by the algorithm;
- the rightmost piece, after position g, includes items that are larger than q.

This invariant is similar to the invariant maintained in Section 5.3.1, but there the three-way partition was obtained by using one single pivot, treating in a special way the items equal to the pivot, and all iterators were moving towards the right; here the algorithm uses two pivots, items equal to these pivots are not treated separately, and two pivots move rightward (i.e. ℓ and k) whereas the third one (i.e. g) moves leftward. This way the items larger than q are correctly located at the end of the input sequence, and the items smaller than p are correctly located at the beginning of the input sequence.

The partitioning proceeds in rounds until $k \geq g$ occurs. Each round first compares $S[k] < p$ and, if this is not true, it compares $S[k] > q$. In the former case it swaps $S[k]$ with $S[\ell]$, and then advances the pointers. In the latter case, the algorithm enters in a loop that moves g leftward up to the position where $S[g] \leq q$; here, it swaps $S[k]$ with $S[g]$. So it is the comparison with $S[k]$ which drives the phase, possibly switching to a long g shifting to the left. This nesting of the comparisons is the key for the efficiency of the algorithm which is able to move towards "better" comparisons which reduce the branch mispredictions. We cannot go into the sophisticate details of the analysis, so we refer the reader to [7].

This example is illustrative of the fact that classic algorithms and problems, known for decades and considered antiquated, may be harbingers of innovation and deep/novel theoretical analysis. So do not ever leave the curiosity to explore and analyze new algorithmic schemes!

5.4 Sorting with multi-disks$^{\infty}$

The bottleneck in disk-based sorting is obviously the time needed to perform an I/O operation. In order to mitigate this problem, we can use D disks working in parallel so to transfer DB items per I/O. On the one hand this increases the bandwidth of the I/O subsystem, but on the other hand, it makes the design of I/O-efficient algorithms particularly difficult. Let's see why.

The simplest approach to manage parallel disks is called *disk striping* and consists of looking at the D disks as *one single* disk whose page size is $B' = DB$. This way we gain simplicity in algorithm design by just using *as-is* any algorithm designed for one disk, now with a disk-page of size B'. Unfortunately, this simple approach pays an un-negligible price in terms of I/O-complexity:

$$O(\frac{n}{B'} \log_{M/B'} \frac{n}{M}) = O(\frac{n}{DB} \log_{M/DB} \frac{n}{M})$$

This bound is not optimal because the base of the logarithm is D times smaller than what indicated by the lower bound proved in Theorem 5.3. The ratio between the bound achieved via disk-striping and the optimal bound is $1 - \log_{M/B} D$, which shows disk striping to be less and less efficient as the number of disks increases $D \longrightarrow M/B$. The problem resides in the fact that we are not deploying the *independency* among disks by using them as a monolithic sub-system.

On the other hand, deploying this independency is tricky and it took several years before designing fully-optimal algorithms running over multi-disks and achieving the bounds stated in Theorem 5.3. The key problem with the management of multi-disks is to guarantee that every time we access the disk sub-system, we are able to read or write D pages each one coming from or going to a different disk. This is to guarantee a throughput of DB items per I/O. In the case of sorting, such a difficulty arises both in the case of distributed-based and merge-based sorters, each with its specialties given the duality of those approaches.

Let us consider the multi-way Quicksort. In order to guarantee a D-way throughput in reading the input items, these must be distributed evenly among the D disks. For example they could be striped circularly as indicated in Figure 5.6. This would ensure that a scan of the input items takes $O(n/DB)$ optimal I/Os.

	Block 1	Block 2	Block 3	Block 4	Block 5	
Disk 1	1 2	9 10	17 18	25 26	33 34	...
Disk 2	3 4	11 12	19 20	27 28	35 36	...
Disk 3	5 6	13 14	21 22	29 30	37 38	...
Disk 4	7 8	15 16	23 24	31 32	39 40	...

FIGURE 5.6: An example of striping a sequence of items among $D = 4$ disks, with $B = 2$.

This way the subsequent distribution phase can read the input sequence at that I/O-speed. Nonetheless problems occur when writing the output sub-sequences produced by the partitioning process. In fact that writing should guarantee that each of these sub-sequences is circularly striped among the disks in order to maintain the invariant for the next distribution phase (to be executed independently over those sub-sequences). In the case of D disks, we have D output blocks that are filled by the partitioning phase. So when they are full these D blocks must be written to D distinct disks to ensure full I/O-parallelism, and thus one I/O. Given the striping of the runs, if all these output blocks belong to the same run, then they can be written in one I/O. But, in general, they belong to different runs so that conflicts may arise in the writing process because blocks of different runs could have to be written onto the same disks. An example is given in Figure 5.7 where we have illustrated a situation in which we have three runs under formation by the partitioning phase of Quicksort, and three disks. Runs are striped circularly among the 3 disks and shadowed blocks correspond to the prefixes of the runs that have been already written on those disks. Arrows point to the next free-blocks of each run where the partitioning phase of Quicksort can append the next distributed items. The figure depicts an extremely bad situation in which all these blocks are located on the same disk D_2, so that an I/O-conflict may arise if the next items to be output by the partitioning phase go to these runs. This practically means that the I/O-subsystem must *serialize* the write operation in $D = 3$ distinct I/Os, hence loosing all the I/O-parallelism of the D-disks. In order to avoid these difficulties, there are known *randomized* solutions that ensure optimal I/Os in the average case [6].

In what follows we sketch a deterministic multi-disk sorter, known as Greed Sort [5], which solves the difficulties above via an elegant merge-based approach which consists of two stages: first, items are *approximately* sorted via an I/O-efficient Multi-way Merger that deals with $R = \Theta(\sqrt{M/B})$ sorted runs in an independent way (thus deploying disks in parallel), and then it *completes* the sorting of the input sequence by using an algorithm (aka ColumnSort, due to T. Leighton in 1985) that takes a linear number of I/Os when executed over *short* sequences of length $O(M^{\frac{3}{2}})$. Correctness comes from the fact that the distance of the un-sorted items from their correct sorted position, after the first stage, is smaller than the size of the sequences manageable by Columsort. Hence the second

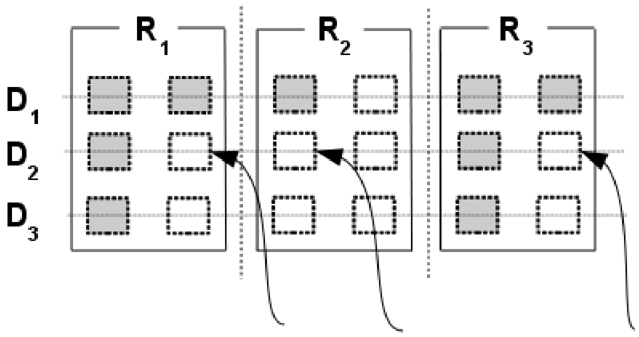


FIGURE 5.7: An example of an I/O-conflict in writing $D = 3$ blocks belonging to 3 distinct runs.

stage can correctly turn the approximately-sorted sequence into a totally-sorted sequence by a single pass.

How to get the approximately sorted runs in I/O-efficient way is the elegant algorithmic contribution of GreedSort. We sketch its main ideas here, and refer the interested reader to the corresponding paper [5] for further details. We assume that sorted runs are stored in a striped way among the D disks, so that reading D consecutive blocks from each of them takes one I/O. As we discussed for Quicksort, also in this Merge-based approach we could incur in I/O-conflicts when reading these runs. GreedSort avoids this problem by operating independently on each disk: in a parallel read operation, GreedSort fetches the two *best* available blocks from each disk. These two blocks are called "best" because they contain the *smallest minimum item*, say m_1, and the *smallest maximum item*, say m_2, currently present in blocks stored on that disk (possibly these two blocks are the same). It is evident that this selection can proceed independently over the D disks, and it needs a proper data structure that keeps track of minimum/maximum items in disk-blocks. Actually [5] shows that this data structure can fit in internal memory, thus not incurring any further I/Os for this selection operations.

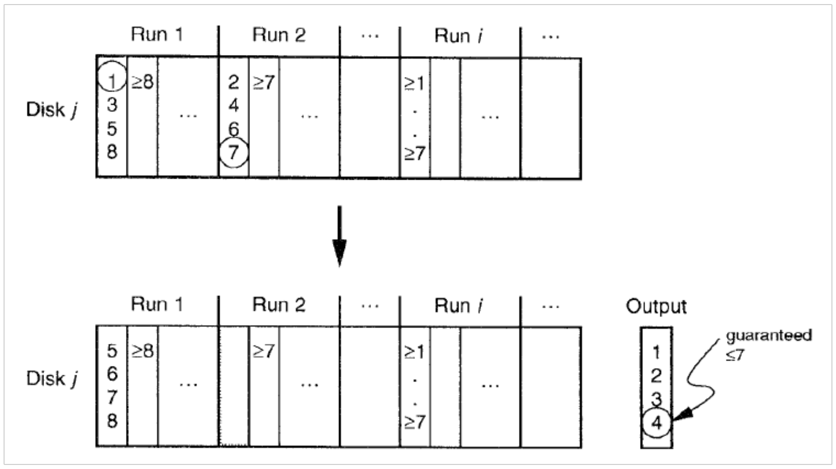


FIGURE 5.8: Example taken from the GreedSort's paper [5].

Figure 5.8 shows an example on disk j, which contains the blocks of several runs because of the striping-based storage. The figure assumes that run 1 contains the block with the smallest minimum item (i.e. 1) and run 2 contains the block with the smallest maximum item (i.e. 7). All the other blocks which come from run 1 contain items larger than 8 (i.e. the maximum of the first block), and all the other blocks which come from run 2 contain items larger than 7. All blocks coming form other runs have minimum larger than 1 and maximum larger than 7. Greedsort then merges these blocks creating two new sorted blocks: the first one is written to output (it contains the items $\{1, 2, 3, 4\}$), the second one is written back to the run of the smallest minimum m_1, namely run 1 (it contains the items $\{5, 6, 7, 8\}$). This last write back into run 1 does not disrupt that ordered subsequence, because the second block contains surely items smaller than the maximum of the block of m_1.

We notice that the items written in output are not necessarily the four smallest items of all runs. In fact it could exist a block in another run (different from runs 1 and 2) which contains a value within $[1, 4]$, say 2.5, and whose minimum is larger than 1 and whose maximum is larger than 7. So this block is compatible with the selection we did above from run 1 and 2, but it contains items that should be stored in the first block of the sorted sequence. So the selection of the "two-best blocks" proceeds independently over all disks until all runs have been examined and written in output. The final sequence produced by this merging process is *not* sorted, but if we read it in a striped-way along all D disks, then it results *approximately* sorted as stated in the following lemma (proved in [5]).

LEMMA 5.2 A sequence is called *L-regressive* if any pair of un-sorted records, say $\ldots y \ldots x \ldots$ with $y > x$, has distance less than L in the sequence. The previous sorting algorithm creates an output that is L-regressive, with $L = RDB = D\sqrt{MB}$.

The application of ColumnSort over the L-regressive sequence, by sliding a window of $2L$ items which moves L steps forward at each phase, allows to produce a merged sequence which is totally sorted. In fact $L = D\sqrt{MB} \leq DB\sqrt{M} \leq M^{3/2}$ and thus ColumnSort is effective in producing the entirely sorted sequence. We notice that at this point this sorted sequence is striped along all D disks, thus the invariant for the next merging phase is preserved and the merge can thus start over a number of runs that has been reduced by a factor R. The net result is that each merging takes $O(n/DB)$ I/Os, the total number of merging stages is $\log_R \frac{n}{M} = O(\log_{M/B} \frac{n}{M})$, and thus the optimal I/O-bound follows.

References

[1] Alok Aggarwal and Jeffrey S. Vitter. The Input/Output complexity of Sorting and Related Problems. *Communication of the ACM*, 31(9): 1116-1127, 1988.

[2] Jon L. Bentley and Robert Sedgewick. Fast algorithms for sorting and searching strings. In *Procs of the 8th ACM-SIAM Symposium on Discrete Algorithms*, 360–369, 1997.

[3] Donald E. Knuth. *The Art of Computer Programming: volume 3*. Addison-Wesley, 2nd Edition, 1998.

[4] Kurt Mehlhorn and Peter Sanders. *Algorithms and Data Structures: The basic toolbox.* Springer, 2009.

[5] Mark H. Nodine and Jeffrey S. Vitter. Greed Sort: Optimal Deterministic Sorting on Parallel Disks. *Journal of the ACM*, 42(4): 919-933, 1995.

[6] Jeffrey S. Vitter. External memory algorithms and data structures. *ACM Computing Surveys*, 33(2):209–271, 2001.

[7] Sebastian Wild and Markus E. Nebel Average case analysis of Java 7's Dual Pivot Quicksort. In *European Symposium on Algorithms (ESA)*, Lecture Notes in Computer Science 7501, Springer, 826–836, 2012.

6 Sorting Strings

In the previous chapter we dealt with sorting *atomic items*, namely items that either occupy $O(1)$ space or have to be managed in their entirety as atomic objects, and thus without possibly deploying their constituent parts. In the present chapter we will generalize those algorithms, and introduce new ones, to deal with the case of *variable-length items* (aka *strings*). More formally, we will be interested in solving efficiently the following problem:

The string-sorting problem. *Given a sequence $S[1, n]$ of strings, having total length N and drawn from an alphabet of size σ, sort these strings in increasing lexicographic order.*

The first idea to attack this problem consists of deploying the power of *comparison-based* sorting algorithms, such as QuickSort or MergeSort, and implementing a proper comparison function between pair of strings. The obvious way to do this is to compare the two strings from their beginning, character-by-character, find their mismatch and then use this character to derive their lexicographic order. Let $L = N/n$ be the average length of the strings in S, an optimal comparison-based sorter would take $O(L\, n \log n) = O(N \log n)$ average time on RAM, because every string comparison may involve $O(L)$ characters on average.

Apart from the time complexity, which is not optimal (see next), the key limitation of this approach in a memory-hierarchy is that S is typically implemented as an *array of pointers* to strings which are stored elsewhere, possibly on disk if N is very large or spread in the internal-memory of the computer. Figure 6.1 shows the two situations via a graphical example. Whichever is the allocation your program chooses to adopt, the sorter will *indirectly* order the strings of S by moving their pointers rather than their characters. This situation is typically neglected by programmers, with a consequent slowness of their sorter when executed on large string sets. The motivation is clear, every time a string comparison is executed between two items, say $S[i]$ and $S[j]$, these two pointers are resolved by accessing their corresponding strings, so that two cache misses or I/Os are elicited in order to fetch and then compare their characters. As a result, the algorithm might incur $\Theta(n \log n)$ I/Os. As we noticed in the first chapter of these notes, the Virtual Memory of the OS will provide an help by buffering the most recently compared strings, and thus possibly reducing the number of incurred I/Os. Nevertheless, two arrays are here competing for that buffering space— i.e. the array of pointers and the strings— and time is wasted by *re-scanning* over and over again string prefixes which have been already compared because of their brute-force comparison.

The rest of this lecture is devoted to propose algorithms which are optimal in the number of executed character comparisons, and possibly offer I/O-conscious patterns of memory accesses which make them efficient also in the presence of a memory hierarchy.

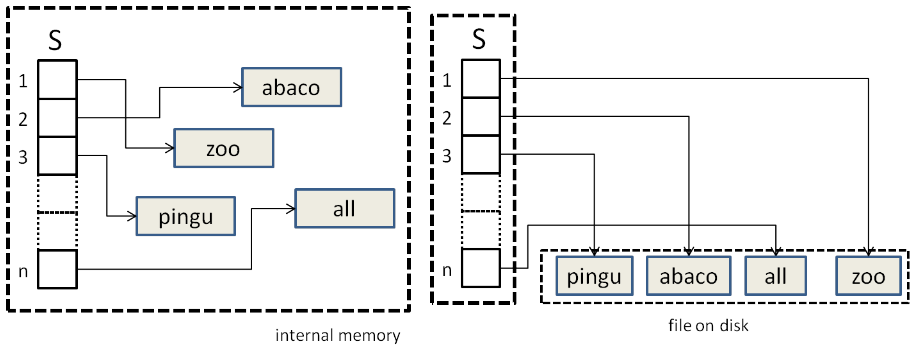


FIGURE 6.1: Two examples of string allocation, spread in the internal memory (left) and written contiguously in a file (right).

6.1 A lower bound

Let d_s be the length of the shortest prefix of the string $s \in S$ that distinguishes it from the other strings in the set. The sum $d = \sum_{s \in S} d_s$ is called the *distinguishing prefix* of the set S. Referring to Figure 6.1, and assuming that S consists of the 4 strings shown in the picture, the distinguishing prefix of the string `all` is `al` because this substring does not prefixes any other string in S, whereas `a` does.

It is evident that any string sorter must compare the initial d_s characters of s, otherwise it would be unable to distinguish s from the other strings in S. So $\Omega(d)$ is a term that must appear in the string-sorting lower bound. However, this term does not take into account the cost to compute the sorted order among the n strings, which is $\Omega(n \log n)$ string comparisons.

LEMMA 6.1 Any algorithm solving the string-sorting problem must execute $\Omega(d + n \log n)$ comparisons.

This formula deserves few comments. Assume that the n strings of S are binary, share the initial ℓ bits, and differ for the rest $\log n$ bits. So each string consists of $\ell + \log n$ bits, and thus $N = n(\ell + \log n)$ and $d = \Theta(N)$. The lower bound in this case is $\Omega(d + n \log n) = \Omega(N + n \log n) = \Omega(N)$. But string sorters based on Mergesort or Quicksort (as the ones detailed above) take $\Theta((\ell + \log n) n \log n) = \Theta(N \log n)$ time. Thus, for any ℓ, those algorithms may be far from optimality of a factor $\Theta(\log n)$.

One could wonder whether the upper-bound can be turned to be smaller than the input size N. This is possible because the string sorting could be implemented without looking at the entire content of strings, provided that $d < N$. And indeed, this is the reason why we introduced the parameter d, which allows a finer analysis of the following algorithms.

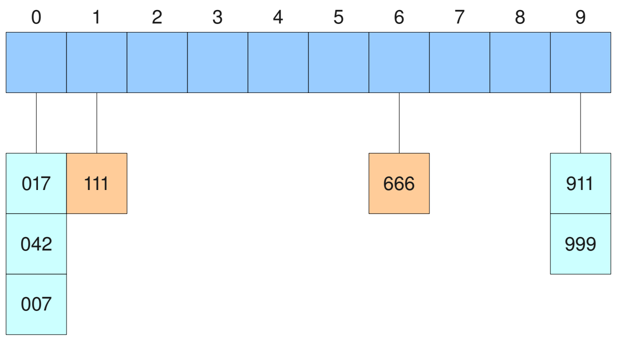


FIGURE 6.2: First distribution step in MSD-first RadixSort.

6.2 RadixSort

The first step to get a more competitive algorithm for string sorting is to look at strings as *sequence of characters* drawn from an integer alphabet $\{0, 1, 2, \ldots, \sigma - 1\}$ (aka *digits*). This last condition can be easily enforced by sorting in advance the characters occurring in S, and then assigning to each of them an integer (rank) in that range. This is typically called *naming* process and takes $O(N \log \sigma)$ time because we can use a binary-search tree built over the at most σ distinct characters occurring in S. After that, all strings can be sorted by considering them as sequence of σ-bounded digits.

Hereafter we assume that strings in S have been drawn from a integer alphabet of size σ and keep in mind that, if this is not the case, a term $O(N \log \sigma)$ has to be added to the time complexity. Moreover, we observe that each character can be encoded in $\lceil \log(\sigma + 1) \rceil$ bits; so that the input size is $\Theta(N \log \sigma)$ whenever it is measured in bits.

We can devise two main variants of RadixSort that differentiate each other according to the order in which the digits of the strings are processed: MSD-first processes the strings rightward starting from the Most Significant Digit, LSD-first processes the strings leftward starting from the Least Significant Digit.

6.2.1 MSD-first

This algorithm follows a divide&conquer approach which processes the strings character-by-character from their beginning, and distributes them into σ buckets. Figure 6.2 shows an example in which S consists of seven strings which are distributed according to their first (most-significant) digit in 10 buckets, because $\sigma = 10$. Since buckets `1` and `6` consist of one single string each, their ordering is known. Conversely, buckets `0` and `9` have to be sorted recursively according to the second digit of the strings contained into them. Figure 6.3 shows the result of the recursive sorting of those two buckets. Then, to get the ordered sequence of strings, all groups are concatenated in left-to-right order.

We point out that, the distribution of the strings among the buckets can be obtained via a comparison-based approach, namely binary search, or by the usage of CountingSort. In the former case the distribution cost is $O(\log \sigma)$ per string, in the latter case it is $O(1)$.

It is not difficult to notice that distribution-based approaches originate search trees. The classic Quicksort originates a binary search tree. The above MSD-first RadixSort originates a σ-ary tree because of the σ-ary partition executed at every distribution step. This tree takes in the literature

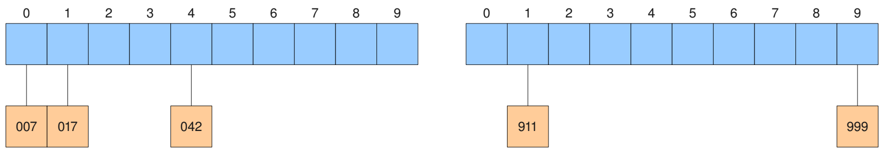


FIGURE 6.3: Recursive sort of bucket 0 (left) and bucket 9 (right) according to the second digit of their strings.

the name of *trie*, or *digital* search tree, and its main use is in string searching (as we will detail in the Chapter ??). An example of trie is given in Figure 6.4.

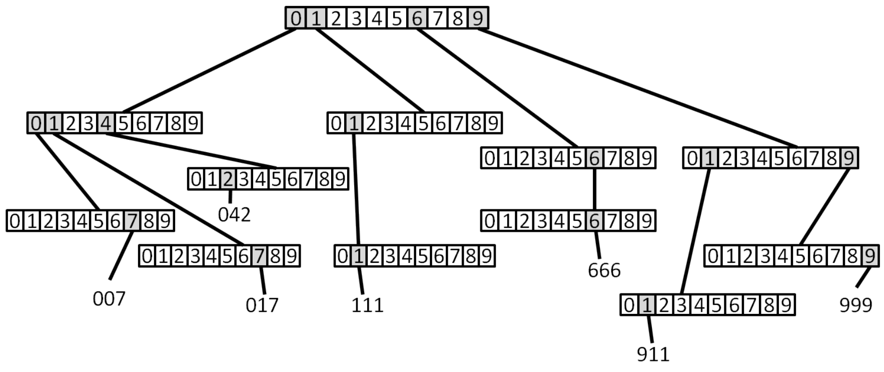


FIGURE 6.4: The trie-based view of MSD-first RadixSort for the strings of Figure 6.2

Every node is implemented as a σ-sized array, one entry per possible alphabet character. Strings are stored in the leaves of the trie, hence we have n leaves. Internal nodes are less than N, one per character occurring in the strings of S. Given a node u, the downward path from the root of the trie to u spells out a string, say $s[u]$, which is obtained by concatenating the characters encountered in the path traversal. For example, the path leading to the leaf 017 traverses three nodes, one per possible prefix of that string. Fixed a node u, all strings that descend from u share the same prefix $s[u]$. For example, $s[\text{root}]$ is the empty string, and indeed all strings of S do share no prefix. Take the leftmost child of the root, it spells the string 0 because it is reached form the root by traversing the edge spurring from the 0-entry of the array. Notice that the trie may contain *unary* nodes, namely nodes that have one single child. All the other internal nodes that have at least two children are called *branching* nodes. In the figure we have 9 unary nodes and 3 branching nodes, where $n = 7$ and $N = 21$. In general the trie can have no more than n branching nodes, and $O(N)$ unary nodes. Actually the unary nodes which have a descending branching node are at most $O(d)$. In fact, these unary nodes correspond to characters occurring in the distinguishing prefixes and the lowest descending branching nodes correspond to the characters that end the distinguishing prefixes; on the other hand, the unary paths which spur from the lowest branching nodes in the trie and lead to leaves correspond to the string suffixes which follow those distinguishing prefixes. In algorithmic terms, the unary nodes correspond to buckets formed by items all sharing the same compared-character in the distribution of MSD-first RadixSort, the branching nodes correspond to buckets formed by items with distinct compared-characters in the distribution of MSD-first RadixSort.

If edge labels are alphabetically sorted, as in Figure 6.4, reading the leaves according to the pre-order visit of the trie gets the sorted S. This suggests a simple trie-based string sorter. The idea is to start with an empty trie, and then insert one string after the other into it. Inserting a string $s \in S$ in the current trie consists of tracing a downward path until s's characters are matched with edge labels. As soon as the next character in s cannot be matched with any of the edges outgoing from the reached node u,[1] then we say that the mismatch for s is found. So a *special* node is appended to the trie at u with that branching character. This special node points to s. The specialty resides in the fact that we have dropped the not-yet-matched suffix of s, but the pointer to the string keeps implicitly track of it for the subsequent insertions. In fact, if inserting another string s' we encounter the special-node u, then we resort the string s (linked to it) and create a (unary) path for the other characters constituting the common prefix between s and s' which descends from u. The last node in this path branches to s and s', possibly dropping again the two paths corresponding to the not-yet-matched suffixes of these two strings, and introducing for each of them one special character.[2]

Every time a trie node is created, an array of size σ is allocated, thus taking $O(\sigma)$ time and space. So the following theorem can be proved.

THEOREM 6.1 *Sorting strings over an (integer) alphabet of size σ can be solved via a trie-based approach in $O(d\,\sigma)$ time and space.*

Proof Every string s spells out a path of length $O(d_s)$, before that the special node pointing to s is created. Each node of those paths allocates $O(\sigma)$ space and takes that amount of time to be allocated. Moreover $O(1)$ is the time cost for traversing a trie-node. Therefore $O(\sigma)$ is the time spent for each traversed/created node. The claim then follows by visiting the constructed trie and listing its leaves from left to right (given that they are lexicographically sorted, because the naming of characters is lexicographic and thus reflects their order). ■

The space occupancy is significant and should be reduced. An option is to replace the σ-sized array into each node u with an hash table (with chaining) of size proportional to the number of edges spurring out of u, say e_u. This guarantees $O(1)$ average time for searching and inserting one edge in each node. The construction time becomes in this case: $O(d)$ to insert all strings in the trie (here every node access takes constant time), $O(\sum_u e_u \log e_u) = O(\sum_u e_u \log \sigma) = O(d \log \sigma)$ for the sorting of the trie edges over all nodes, and $O(d)$ time to scan the trie leaves rightward via a pre-order visit of the trie, and thus get the dictionary strings in lexicographic order.

THEOREM 6.2 *Sorting strings drawn from an integer alphabet of size σ, by using the trie-based approach with hashing, takes $O(d \log \sigma)$ average time and $O(d)$ space.*

When σ is small we cannot conclude that this result is *better than the lower bound* provided in Lemma 6.1 because that applies to comparison-based algorithms and thus it does not apply to hashing or integer sorting.

[1]This actually means that the slot in the σ-sized array of u corresponding to the next character of s is `null`.
[2]We are assuming that allocating a σ-sized array cost $O(1)$ time.

The space allocated for the trie can be further reduced to $O(n)$, and the construction time to $O(d + n \log \sigma)$, by using *compacted* tries, namely tries in which the unary paths have been compacted into single edges whose length is equal to the length of the compacted unary path. The discussion of this data structure is deferred to Chapter **??**.

6.2.2 LSD-first

The next sorter was discovered by Herman Hollerith more than a century ago and led to the implementation of a card-sorting machine for the 1890 U.S. Census. It is curious to find that he was the founder of a company that then became IBM [4]. The algorithm is counter-intuitive because it sorts strings digit-by-digit starting from the least-significant one and using a *stable* sorter as black-box for ordering the digits. We recall that a sorter is *stable* iff equal keys maintain in the final sorted order the one they had in the input. This sorter is typically the CountingSort (see e.g. [5]), and this is the one we use below to describe the LSD-first RadixSort. We assume that all strings have the same length L, otherwise they are *logically* padded to their front with a special digit which is assumed to be *smaller than* any other alphabet digit. The ratio is that, this way, the LSD-first RadixSort correctly obtains an ordered lexicographic sequence.

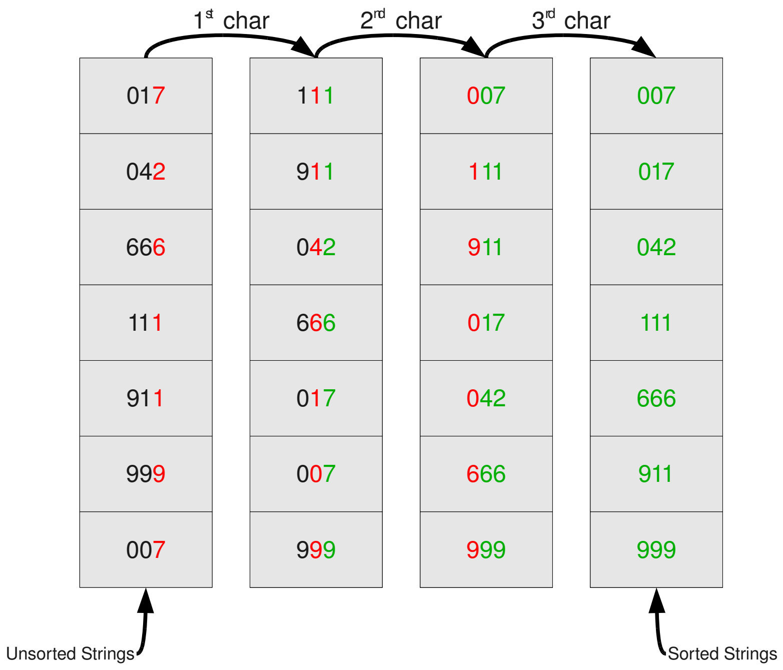


FIGURE 6.5: A running example for LSD-first RadixSort.

The LSD-first RadixSort consists of L phases, say $i = 1, 2, \ldots, L$, in each phase we stably sort all strings according to their i-th least significant digit. A running example of LSD-first RadixSort is given in Figure 6.5: the red digits (characters) are the ones that are going to be sorted in the current phase, whereas the green digits are the ones already sorted in the previous phases. Each phase produces a new sorted order which deploys the order in the input sequence, obtained from the previous phases, to resolve the ordering of the strings which show equal digits in the currently compared position i. As an example let us consider the strings `111` and `017` in the 2nd phase of Figure 6.5. These strings present the same second digit so their ordering in the second phase poses `111` before `017`, just because this was the ordering after the first sorting step. This is clearly a wrong order which will be nonetheless correctly adjusted after the last phase which operates on the third digit, i.e. `1` vs `0`. The time complexity can be easily estimated as L times the cost of executing CountingSort over n integer digits drawn from the range $[1, \sigma]$. Hence it is $O(L\,(n + \sigma))$. A nice property of this sorter is that it is in-place whenever the sorting black-box is in-place, namely $\sigma = O(1)$.

LEMMA 6.2 LSD-first Radixsort solves the string-sorting problem over an integer alphabet of size σ in $O(L\,(n+\sigma)) = O(N + L\sigma)$ time and $O(N)$ space. The sorter is in-place iff an in-place digit sorter is adopted.

Proof Time and space efficiency follow from the previous observations. The correctness is proved by deploying the stability of the Counting Sort. Let α and β be two strings of S, and assume that $\alpha < \beta$ according to the lexicographic order. Since we assumed that S's strings have the same length we can decompose these two strings into three parts: $\alpha = \gamma a \alpha_1$ and $\beta = \gamma b \beta_1$, where γ is the longest common prefix between α and β (possibly it is empty), $a < b$ are the first mismatch characters, α_1 and β_1 are the two remaining suffixes (which may be empty).

Let us now look at the history of comparisons between the digits of α and β. We can identify three stages, depending on the position of the compared digit within the three-way decomposition above. Since the algorithm starts from the least-significant digit, it starts comparing digits in α_1 and β_1. We do not care about the ordering after the first $|\alpha_1| = |\beta_1|$ phases, because at the immediately next phase, α and β are sorted in accordance to characters a and b. Since $a < b$ this sorting places α before β. All other $|\gamma|$ sorting steps will compare the digits falling in γ, which are equal in both strings, so their order will not change because of the stability of the digit-sorter. At the end we will correctly have $\alpha < \beta$. Since this holds for any pair of strings in S, the final sequence produced by LSD-first RadixSort will be lexicographically ordered. ■

The previous time bound deserves few comments. LSD-first RadixSort processes all digits of all strings, so it seems not appealing when $d \ll N$ with respect to MSD-first RadixSort. But the efficiency of LSD-first RadixSort hinges onto the observation that nobody prevents a phase to sort *groups of digits* rather than a single digit at a time. Clearly the longer is this group, the larger is the time complexity of a phase, but the smaller is the number of phases. We are in the presence of a trade-off that can be tuned by investigating deeply the relation that exists between these two parameters. Without loss of generality, we simplify our discussion by assuming that the strings in S are *binary* and have equal-length of b bits, so $N = bn$ and $\sigma = 2$. Of course, this is not a limitation in practice because any string is encoded in memory as a bit sequence; in theory, we can assume to pre-sort the alphabet-characters in $O(N \log \sigma)$ time and then encode them via bit-sequences of $\lceil \log \sigma \rceil$ bits reflecting the digit order.

LEMMA 6.3 LSD-first RadixSort takes $\Theta(\frac{b}{r}(n + 2^r))$ time and $O(nb) = O(N)$ space to sort n strings of b bits each. Here $r \leq b$ is a positive integer fixed in advance.

Proof We decompose each string in $g = \frac{b}{r}$ groups of r bits each. Each phase will order the strings according to a group of r bits. Hence CountingSort is asked to order n integers between 0 and $2^r - 1$ (extremes included), so it takes $O(n + 2^r)$ time. As there are $g = \frac{b}{r}$ phases, the total time is $O(g\,(n + 2^r)) = O((\frac{b}{r})(n + 2^r))$. ■

Given n and b, we need to choose a proper value for r such that the time complexity is minimized. We could derive this minimum via analytic calculus (i.e. first-order derivatives) but, instead, we argue for the minimum as follows. Since the CountingSort uses $O(n + 2^r)$ time to sort each group of r digits, it is useless to use groups shorter than $\log n$, given that $\Omega(n)$ time has to be payed in any case. So we have to choose r in the interval $[\log n, b]$. As r grows larger than $\log n$, the time complexity in Lemma 6.3 also increases because of the ratio $2^r/r$. So the best choice is $r = \Theta(\log n)$ for which the time complexity is $O(\frac{bn}{\log n})$.

THEOREM 6.3 *LSD-first Radixsort sorts n strings of b bits each in* $O(\frac{bn}{\log n})$ *time and* $O(bn)$ *space, by using CountingSort on groups of* $\Theta(\log n)$ *bits. The algorithm is not in-place because it needs* $\Theta(n)$ *space for the Counting Sort.*

We finally observe that bn is the total length in bits of the strings in S, so we can express that number as $N \log \sigma$ since each character takes $\log \sigma$ bits to be represented.

COROLLARY 6.1 LSD-first Radixsort solves the string-sorting problem on strings drawn from an arbitrary alphabet in $O(\frac{N \log \sigma}{\log n})$ time and $O(N \log \sigma)$ bits of space.

If $d = \Theta(N)$ and σ is a constant, the comparison-based lower-bound (Lemma 6.1) becomes $\Omega(d + n \log n) = \Omega(N)$. So LSD-first Radixsort equals or even beats that lower bound; but this is not surprising because this sorter operates on an *integer* alphabet and uses CountingSort, so it is *not* a comparison-based string sorter.

Comparing the trie-based construction (Theorems 6.1–6.2) and the LSD-first RadixSort algorithm we conclude that the former is always better than the latter for $d = O(\frac{N}{\log n})$, which is true for most practical cases. In fact LSD-first RadixSort needs to scan the whole string set whichever are the string compositions, whereas the trie-construction may skip some string suffixes whenever $d \ll N$. However the LSD-first approach avoids the dynamic memory allocation, incurred by the construction of the trie, and the extra-space due to the storage of the trie structure. This additional space and work is non negligible in practice and could impact unfavorably on the real performance of the trie-based sorter, or even prevent its use over large string sets because the internal memory has bounded size M.

6.3 Multi-key Quicksort

This is a variant of the well-known Quicksort algorithm extended to manage items of variable length. Moreover it is a comparison-based string sorter which matches the lower bound of $\Omega(d + n \log n)$ stated in Lemma 6.1. For a recap about Quicksort we refer the reader to the previous chapter. Here it is enough to recall that Quicksort hinges onto two main ingredients: the pivot-selection procedure and the algorithm to partition the input array according to the selected pivot. In Chapter 5 we discussed widely these issues, for the present section we fix ourselves to a pivot-selection based on a *random* choice and to a *three-way* partitioning of the input array. All other variants discussed in Chapter 5 can be easily adapted to work in the string setting too.

The key here is that items are not considered as atomic, but they are strings to be split into their constituent characters. Now the pivot is a character, and the partitioning of the input strings is done according to the single character that occupies a given position within them. Figure 6.1 details the pseudocode of Multi-key Quicksort, in which it is assumed that the input set R is *prefix free*, so no string in R prefixes any other string. This condition can be easily guaranteed by assuming that strings of R are distinct and logically padded with a dummy character that is smaller than any other character in the alphabet. This guarantees that any pair of strings in R admits a bounded longest-common-prefix (shortly, *lcp*), and that the mismatch character following the lcp does exist in both strings.

Algorithm 6.1 MULTIKEYQS(R, i)

```
if $|R| \leq 1$ then
    return R;
else
    choose a pivot-string $p \in R$;
    $R_< = \{s \in R \mid s[i] < p[i]\}$;
    $R_= = \{s \in R \mid s[i] = p[i]\}$;
    $R_> = \{s \in R \mid s[i] > p[i]\}$;
    $A = MultikeyQS(R_<, i)$;
    $B = MultikeyQS(R_=, i + 1)$;
    $C = MultikeyQS(R_>, i)$;
    return the concatenated sequence $A, B, C$;
end if
```

The algorithm receives in input a sequence R of strings to be sorted and an integer parameter $i \geq 0$ which denotes the offset of the character driving the three-way partitioning of R. The pivot-character is $p[i]$ where p is randomly chosen string within R. The real implementation of this three-way partitioning can follow the PARTITION procedure of Chapter 5. MULTIKEYQS(R, i) assumes that the following pre-condition holds on its input parameters: All strings in R are lexicographically sorted up to their $(i - 1)$-long prefixes. So the sorting of a string sequence $R[1, n]$ is obtained by invoking MULTIKEYQS$(R, 1)$, which ensures that the invariant trivially holds for the initial sequence R. Steps 5-7 partitions R in three subsets whose notation is explicative of their content. All these three subsets are recursively sorted and their ordered sequences are eventually concatenated in order to obtain the ordered R. The tricky issue here is the invocation of the three recursive calls:

- the sorting of the strings in $R_<$ and $R_>$ has still to reconsider the ith character, because we just checked that it is smaller/greater than $p[i]$ (and this is not sufficient to order those strings). So recursion does not advance i, but it hinges on the current validity of the invariant.
- the sorting of the strings in $R_=$ can advance i because, by the invariant, these strings are sorted up to their $(i - 1)$-long prefixes and, by construction of $R_=$, they share the i-th character. Actually this character is equal to $p[i]$, so $p \in R_=$ too.

These observations make correctness immediate. We are therefore left with the problem of computing the average time complexity of MULTIKEYQS$(R, 1)$. Let us concentrate on a single string, say $s \in R$, and count the number of comparisons that involve one of its characters. There are two cases, either $s \in R_< \cup R_>$ or $s \in R_=$. In the first case, s is compared with the pivot-string p and then included in a smaller set $R_< \cup R_> \subset R$ with the offset i unchanged. In the other case s is compared with p but, since the i-th character is found equal, it is included in a smaller set *and* offset i is advanced. If the pivot selection is *good* (see Chapter 5), the three-way partitions are balanced and thus

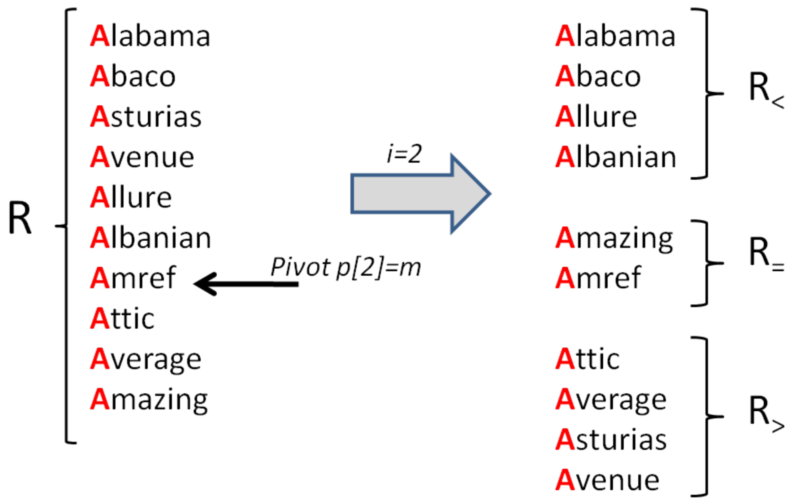


FIGURE 6.6: A running example for MULTIKEYQS(R, 2). In red we have the 1-long prefix shared by all strings in R.

$|R_< \cup R_>| \leq \alpha\, n$, for a proper constant $\alpha < 1$. As a result, both cases cost $O(1)$ time but one reduces the string set by a constant factor, while the other increases i. Since the initial set R has size n, and i is bounded above by the string length $|s|$, we have that the number of comparisons involving s is $O(|s| + \log n)$. Summing up over all strings in R we get the time bound $O(N + n \log n)$. A finer look at the second case shows that i can be bounded above by the number of characters that belong to s's distinguishing prefix, because these characters will led s to be located in a singleton set.

THEOREM 6.4 *Multi-key Quicksort solves the string-sorting problem by performing $O(d + n \log n)$ character comparisons on average. The bound can be turned into a worst-case bound by adopting a worst-case linear-time algorithm to select the pivot as the median of R. This is optimal.*

Comparing this result against what was obtained for the MSD-first RadixSort (Theorem 6.2) we observe that in that Theorem we got $\log \sigma$ instead of $\log n$, nevertheless the succinct space occupancy make Multi-key Quicksort very appealing in practice. Moreover, similarly as done for Quicksort, it is possible to prove that if the partition is done around the median of $2t + 1$ randomly selected pivots, Multi-key Quicksort needs at most $\frac{2nH_n}{H_{2t-2}-H_{t+1}} + O(d)$ average comparisons. By increasing the sample size t, one can reduce the time near to $n \log n + O(d)$. This bound is similar to the one obtained with the trie-based sorter (see Theorem 6.2, where the log-argument was σ instead of n), but the algorithm is much simpler, it does not use additional data structures (i.e. hash tables), and in fact it is the typical choice in practice.

We conclude this section by noticing an interesting parallel between Multikey Quicksort and *ternary* search trees, as discussed in [6]. These are search data structures in which each node contains a *split character* and pointers to low and high (or left and right) children. In some sense a ternary search tree is obtained from a trie by collapsing together the children whose leading edges are smaller/greater than the split character. If a given node splits on the character in position i, its low and high children also split on i-th character. Instead, the equal-child splits on the next $(i + 1)$-th character. Ternary search trees may be balanced by either inserting elements in random order

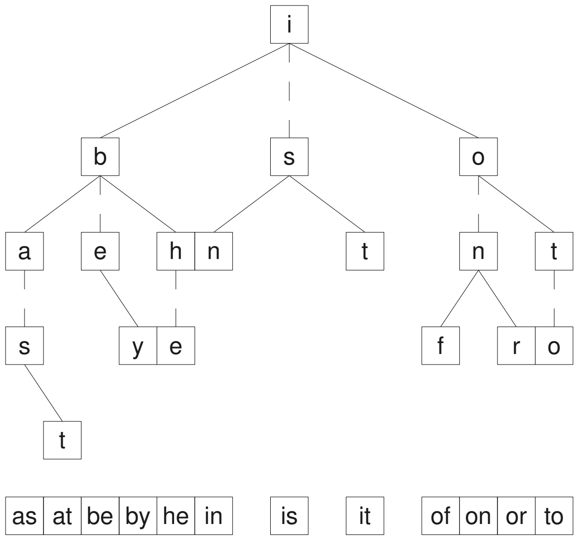


FIGURE 6.7: A ternary search tree for 12 two-letter strings. The low and high pointers are shown as solid lines, while the pointers to the equal-child are shown as dashed lines. The split character is indicated within the nodes.

or applying a variety of known schemes. Searching proceeds by following edges according to the split-characters of the encountered nodes. Figure 6.7 shows an example of a ternary search tree. The search for the word P =”ir” starts at the root, which is labeled with the character `i`, with the offset $x = 1$. Since $P[1] = \mathtt{i}$, the search proceeds down to the equal-child, increments x to 2, and thus reaches the node with split-character `s`. Here $P[2] = \mathtt{r} < \mathtt{s}$, so the search goes to the low/left child which is labeled `n`, and keeps x unchanged. At that node the search stops, because the split character is different and no (low or high) children do exist. So the search concludes that P does not belong to the string set indexed by the ternary search tree.

THEOREM 6.5 *A search for a pattern $P[1, p]$ in a perfectly-balanced ternary search tree representing n strings takes at most $\lfloor \log n \rfloor + p$ comparisons. This is optimal when P is drawn from a general alphabet (or, equivalently, for a comparison-based search algorithm).*

6.4 Some observations on the I/O-model$^\infty$

Sorting strings on disk is not nearly as simple as it is in internal memory, and a bunch of sophisticated string-sorting algorithms have been introduced in the literature which achieve I/O-efficiency (see e.g. [1, 3]). The difficulty is that strings have variable length and their brute-force comparisons over the sorting process may induce a lot of I/Os. In the following we will use the notation: n_s is the number of strings shorter than B, whose total length is N_s, n_l is the number of strings longer than B, whose total length is N_l. Clearly $n = n_s + n_l$ and $N = N_s + N_l$,

The known algorithms can be classified according to the way strings are managed in their sorting process. We can devise mainly three models of computations [1]:

***Model A*:** Strings are considered *indivisible* (i.e., they are moved in their entirety and cannot be broken into characters), except that long strings can be divided into blocks of size B.

***Model B*:** Relaxes the indivisibility assumption of Model A by allowing strings to be divided into single characters, but this may happen *only in internal memory*.

***Model C*:** Waives the indivisibility assumption by allowing division of strings *in both internal and external memory*.

Model A forces to use Mergesort-based sorters which achieve the following optimal bounds:

THEOREM 6.6 *In Model A, string sorting takes* $\Theta(\frac{N_s}{B}\log_{M/B}\frac{N_s}{B} + n_l \log_{M/B} n_l + \frac{N_s+N_l}{B})$ *I/Os.*

The first term in the bound is the cost of sorting the short strings, the second term is the cost of sorting the long strings, and the last term accounts for the cost of reading the whole input. The result shows that sorting short strings is as difficult as sorting their individual characters, which are N_s, while sorting long strings is as difficult as sorting their first B characters. The lower bound for small strings in Theorem 6.6 is proved by extending the technique used in Chapter 5 and considering the special case where all n_s small strings have the same length N_s/n_s. The lower bound for the long strings is proved by considering the n_l small strings obtained by looking at their first B characters. The upper bounds in Theorem 6.6 are obtained by using a special Multi-way MergeSort approach that takes advantage of a *lazy trie* stored in internal memory to guide the merge passes among the strings.

Model B presents a more complex situation, and leads to handle long and short strings separately.

THEOREM 6.7 *In Model B, sorting long strings takes* $\Theta(n_l \log_M n_l + \frac{N_l}{B})$ *I/Os, whereas sorting short strings takes* $O(\min\{n_s \log_M n_s, \frac{N_s}{B}\log_{M/B}\frac{N_s}{B}\})$ *I/Os.*

The first bound for long strings is optimal, the second for short strings is not. Comparing the optimal bound for long strings with the corresponding bound in Theorem 6.6, we notice that they differ in terms of the base of the logarithm: the base is M rather than M/B. This shows that breaking up long strings in internal memory is provably helpful for external string-sorting. The upper bound is obtained by combining the String B-tree data structure (described in Chapter **??**) with a proper buffering technique. As far as short strings are concerned, we notice that the I/O-bound is the same as the cost of sorting all the characters in the strings when the average length N_s/n_s is $O(\frac{B}{\log_{M/B} M})$. For the (in practice) narrow range $\frac{B}{\log_{M/B} M} < \frac{N_s}{n_s} < B$, the cost of sorting short strings becomes $O(n_s \log_M n_s)$. In this range, the sorting complexity for Model B is lower than the one for Model A, which shows that breaking up short strings in internal memory is provably helpful.

Surprisingly enough, the best deterministic algorithm for Model C is derived from the one designed from Model B. However, since Model C allows to split strings on disk too, we can use randomization and hashing. The main idea is to shrink strings by hashing some of their pieces. Since hashing does not preserve the lexicographic order, these algorithms must orchestrate the selection of the string pieces to be hashed with a carefully designed sorting process so that the correct sorted order may be eventually computed. Recently [3] proved the following result (which can be extended to the more powerful cache-oblivious model):

THEOREM 6.8 *In Model C, the string-sorting problem can be solved by a randomized algorithm using* $O(\frac{n}{B}(\log^2_{M/B}\frac{N}{M})(\log n) + \frac{N}{B})$ *I/Os, with arbitrarily high probability.*

References

[1] Lars Arge, Paolo Ferragina, Roberto Grossi, and Jeff S. Vitter. On sorting strings in external memory. In *Procs of the ACM Symposium on Theory of Computing (STOC)*, pp. 540–548, 1997.

[2] Jon L. Bentley and Doug McIlroy. Engineering a sort function. *Software-Practice and Experience*, pages 1249–1265, 1993.

[3] Rolf Fagerberg, Anna Pagh, Rasmus Pagh. External String Sorting: Faster and Cache-Oblivious. In *Procs of the Symposium on Theoretical Aspects of Computer Science (STACS)*, LNCS 3884, Springer, pp. 68-79, 2006.

[4] Herman Hollerith. Wikipedia's entry at `http://en.wikipedia.org/wiki/Herman_Hollerith`.

[5] Tomas H. Cormen, Charles E. Leiserson, Ron L. Rivest and Cliff Stein. *Introduction to Algorithms*. The MIT Press, third edition, 2009.

[6] Robert Sedgewick and Jon L. Bentley. Fast algorithms for sorting and searching strings. *Eight Annual ACM-SIAM Symposium on Discrete Algorithms*, 360-369, 1997.

7

The Dictionary Problem

In this lecture we present *randomized* and simple, yet smart, data structures that solve efficiently the classic **Dictionary Problem**. These solutions will allow us to propose *algorithmic fixes* to some *issues* that are typically left untouched or only addressed via "hand waiving" in basic courses on algorithms.

Problem. *Let $\mathcal{D}$ be a set of n objects, called the* dictionary*, uniquely identified by* `keys` *drawn from a* universe U*. The dictionary problem consists of designing a data structure that efficiently supports the following three basic operations:*

- `Search(`k`):` *Check whether $\mathcal{D}$ contains an object o with key k =* `key`$[o]$*, and then return* `true` *or* `false`*, accordingly. In some cases, we will ask to return the object associated to this key, if any, otherwise return* `null`.
- `Insert(`x`):` *Insert in $\mathcal{D}$ the object x indexed by the key k =* `key`$[x]$*. Typically it is assumed that no object in $\mathcal{D}$ has key k, before the insertion takes place; condition which may easily be checked by executing a preliminary query* `Search(`k`)`.
- `Delete(`k`):` *Delete from $\mathcal{D}$ the object indexed by the key k, if any.*

In the case that all three operations have to be supported, the problem and the data structure are named *dynamic*; otherwise, if only the query operation has to be supported, the problem and the data structure are named *static*.

We point out that in several applications the structure of an object x typically consists of a pair $\langle k, d\rangle$, where $k \in U$ is the key indexing x in $\mathcal{D}$, and d is the so called *satellite data* featuring x. For the sake of presentation, in the rest of this chapter, we will drop the satellite data and the notation $\mathcal{D}$ in favor just of the key set $S \subseteq U$ which consists of all keys indexing objects in $\mathcal{D}$. This way we will simplify the discussion by considering dictionary search and update operations only on those

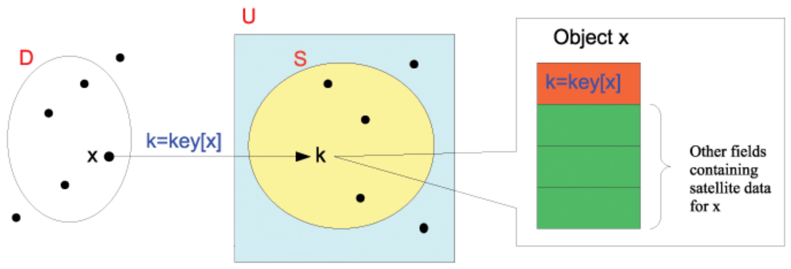


FIGURE 7.1: Illustrative example for U, S, $\mathcal{D}$ and an object x.

keys rather than on (full) objects. But if the context will require also satellite data, we will again talk about objects and their implementing pairs. See Figure 7 for a graphical representation.

Without any loss of generality, we can assume that keys are non-negative integers: $U = \{0, 1, 2, ...\}$. In fact keys are represented in our computer as binary strings, which can thus be interpreted as natural numbers.

In the following sections, we will analyze three main data structures: direct-address tables (or arrays), hash tables (and some of their sophisticated variants) and the Bloom Filter. The former are introduced for teaching purposes, because several times the dictionary problem can be solved very efficiently without resorting involved data structures. The subsequent discussion on hash tables will allow us, first, to fix some issues concerning with the design of a *good* hash function (typically flied over in basic algorithm courses), then, to design the so called *perfect* hash tables, that address *optimally and in the worst case* the static dictionary problem, and then move to the elegant *cuckoo* hash tables, that manage dictionary updates efficiently, still guaranteing constant query time in the *worst case*. The chapter concludes with the *Bloom Filter*, one of the most used data structures in the context of large dictionaries and Web/Networking applications. Its surprising feature is to guarantee query and update operations in constant time, and, more surprisingly, to take space depending on the number of keys n, but not on their lengths. The reason for this impressive *"compression"* is that keys are dropped and only a *fingerprint of few bits* for each of them is stored; the incurred cost is a *one-side error* when executing `Search(`k`)`: namely, the data structure answers in a correct way when $k \in S$, but it may answer un-correctly if k is not in the dictionary, by returning answer *true* (a so called *false positive*). Despite that, we will show that the probability of this error can be mathematically bounded by a function which exponentially decreases with the space m reserved to the Bloom Filter or, equivalently, with the number of bits allocated per each key (i.e. its fingerprint). The nice thing of this formula is that it is enough to take m a constant-factor slightly more than n and reach a negligible probability of error. This makes the Bloom Filter much appealing in several interesting applications: crawlers in search engines, storage systems, P2P systems, etc..

7.1 Direct-address tables

The simplest data structure to support all dictionary operations is the one based on a binary table T, of size $u = |U|$ bits. There is a one-to-one mapping between keys and table's entries, so that entry $T[k]$ is set to 1 *iff* the key $k \in S$. If some satellite data for k has to be stored, then T is implemented as a table of *pointers to* these satellite data. In this case we have that $T[k] \neq$ `NULL` *iff* $k \in S$ and it points to the memory location where the satellite data for k are stored.

Dictionary operations are trivially implemented on T and can be performed in *constant (optimal) time* in the worst case. The main issue with this solution is that table's occupancy depends on the

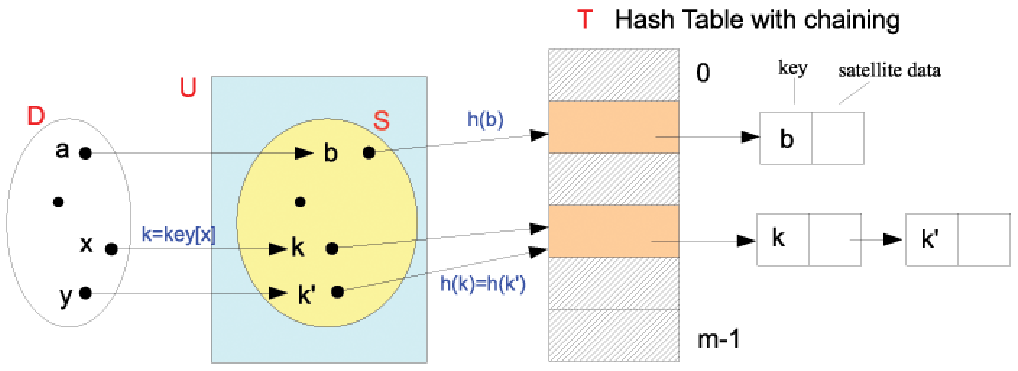


FIGURE 7.2: Hash table with chaining.

universe size u; so if $n = \Theta(u)$, then the approach is optimal. But if the dictionary is small compared to the universe, the approach wastes a lot of space and becomes unacceptable. Take the case of a university which stores the data of its students indexed by their IDs: there can be even million of students but if the IDs are encoded with integers (hence, 4 bytes) then the universe size is 2^32, and thus of the order of billions. Smarter solutions have been therefore designed to reduce the sparseness of the table still guaranteeing the efficiency of the dictionary operations: among all proposals, hash tables and their many variations provide an excellent choice!

7.2 Hash Tables

The simplest data structure for implementing a dictionary are arrays and lists. The former data structure offers constant-time access to its entries but linear-time updates; the latter offers opposite performance, namely linear-time to access its elements but constant-time updates whenever the position where they have to occur is given. Hash tables combine the best of these two approaches, their simplest implementation is the so called *hashing with chaining* which consists of an *array of lists*. The idea is pretty simple, the hash table consists of an array T of size m, whose entries are either `NULL` or they point to lists of dictionary items. The mapping of items to array entries is implemented via an *hash function* $h : U \to \{0, 1, 2 \ldots, m-1\}$. An item with key k is appended to the list pointed to by the entry $T[h(k)]$. Figure 7.2 shows a graphical example of an hash table with chaining; as mentioned above we will hereafter interchange the role of items and their indexing keys, to simplify the presentation, and imply the existence of some satellite data.

Forget for a moment the implementation of the function h, and assume just that its computation takes constant time. We will dedicate to this issue a significant part of this chapter, because the overall efficiency of the proposed scheme strongly depends on the efficiency and efficacy of h to *distribute* items evenly among the table slots.

Given a *good* hash function, dictionary operations are easy to implement over the hash table because they are just turned into operations on the array T and on the lists which spur out from its entries. Searching for an item with key k boils down to a search for this key in the list $T[h(k)]$. Inserting an item x consists of appending it at the front of the list pointed to by $T[h(\mathtt{key}[x])]$. Deleting an item with key k consists of first searching for it in the list $T[h(k)]$, and then removing the corresponding object from that list. The running time of dictionary operations is constant for `Insert`(x), provided that the computation of $h(k)$ takes constant time, and it is linear in the length of the list pointed to by $T[h(k)]$ for both the other operations, namely `Search`(k) and `Delete`(k).

Therefore, the efficiency of hashing with chaining depends on the ability of the hash function h to *evenly distribute* the dictionary items among the m entries of table T, the more evenly distributed they are the shorter is the list to scan. The worst situation is when all dictionary items are hashed to the same entry of T, thus creating a list of length n. In this case, the cost of searching is $\Theta(n)$ because, actually, the hash table boils down to a single linked list!

This is the reason why we are interested in *good* hash functions, namely ones that distribute items among table slots uniformly at random (aka *simple uniform hashing*). This means that, for such hash functions, every key $k \in S$ is *equally likely* to be hashed to everyone of the m slots in T, *independently* of where other keys are hashed. If h is such, then the following result can be easily proved.

THEOREM 7.1 *Under the hypotheses of simple uniform hashing, there exists a hash table with chaining, of size m, in which the operation* `Search(`k`)` *over a dictionary of n items takes* $\Theta(1+n/m)$ *time on average. The value* $\alpha = n/m$ *is often called the* load factor *of the hash table.*

Proof In case of unsuccessful search (i.e. $k \notin S$), the average time for operation `Search(`k`)` equals the time to perform a full scan of the list $T[h(k)]$, and thus it equals its length. Given the uniform distribution of the dictionary items by h, the average length of a generic list $T[i]$ is $\sum_{x \in S} p(h(\texttt{key}[x]) = i) = |S| \times \frac{1}{m} = n/m = \alpha$. The "plus 1" in the time complexity comes from the constant-time computation of $h(k)$.

In case of successful search (i.e. $k \in S$), the proof is less straightforward. Assume x is the i-th item inserted in T, and let the insertion be executed at the tail of the list $\mathcal{L}(x) = T[h(\texttt{key}[x])]$; we need just one additional pointer per list keep track of it. The number of elements examined during `Search(key`$[x]$`)` equals the number of items which were present in $\mathcal{L}(x)$ plus 1, i.e. x itself. The average length of $\mathcal{L}(x)$ can be estimated as $n_i = \frac{i-1}{m}$ (given that x is the i-th item to be inserted), so the cost of a successful search is obtained by averaging $n_i + 1$ over all n dictionary items. Namely,

$$\frac{1}{n}\sum_{i=1}^{n}\left(1 + \frac{i-1}{m}\right) = 1 + \frac{\alpha}{2} - \frac{1}{2m}$$

Therefore, the total time is $O(2 + \frac{\alpha}{2} - \frac{1}{2m}) = O(1 + \alpha)$. ■

The space taken by the hash table can be estimated very easily by observing that list pointers take $O(\log n)$ bits, because they have to index one out of n items, and the item keys take $O(\log u)$ bits, because they are drawn from a universe U of size u. It is interesting to note that the key storage can dominate the overall space occupancy of the table as the universe size increases (think e.g. to URL as keys). It might take even more space than what it is required by the list pointers and the table T (aka, the indexing part of the hash table). This is a simple but subtle observation which will be exploited when designing the Bloom Filter in Section 7.7. To be precise on the space occupancy, we state the following corollary.

COROLLARY 7.1 Hash table with chaining occupies $(m + n)\log_2 n + n\log_2 u$ bits.

It is evident that if the dictionary size n is known, the table can be designed to consists of $m = \Theta(n)$ cells, and thus obtain a constant-time performance over all dictionary operations, on average. If n is unknown, one can resize the table whenever the dictionary gets too small (many deletions), or too large (many insertions). The idea is to start with a table size $m = 2n_0$, where n_0 is the initial number of dictionary items. Then, we keep track of the current number n of dictionary items present in T.

If the dictionary gets too small, i.e. $n < n_0/2$, then we halve the table size and rebuild it; if the dictionary gets too large, i.e. $n > 2n_0$, then we double the table size and rebuild it. This scheme guarantees that, at any time, the table size m is proportional to the dictionary size n by a factor 2, thus implying that $\alpha = m/n = O(1)$. Table rebuilding consists of inserting the current dictionary items in a new table of proper size, and drop the old one. Since insertion takes $O(1)$ time per item, and the rebuilding affects $\Theta(n)$ items to be deleted and $\Theta(n)$ items to be inserted, the total rebuilding cost is $\Theta(n)$. But this cost is paid at least every $n_0/2 = \Omega(n)$ operations, the worst case being the one in which these operations consist of all insertions or all deletions; so the rebuilding cost can be spread over the operations of this sequence, thus adding a $O(1 + m/n) = O(1)$ *amortized cost* at the actual cost of each operation. Overall this means that

COROLLARY 7.2 Under the hypothesis of simple uniform hashing, there exists a *dynamic* hash table with chaining which takes constant time, expected and amortized, for all three dictionary operations, and uses $O(n)$ space.

7.2.1 How do we design a "good" hash function ?

Simple uniform hashing is difficult to guarantee, because one rarely knows the probability distribution according to which the keys are drawn and, in addition, it could be the case that the keys are not drawn independently. Let us dig into this latter feature. Since h maps keys from a universe of size u to a integer-range of size m, it induces a partition of those keys in m subsets $U_i = \{k \in U \ : \ h(k) = i\}$. By the *pigeon principle* it does exist at least one of these subsets whose size is larger than the average load factor u/m. Now, if we reasonably assume that the universe is sufficiently large to guarantee that $u/m = \Omega(n)$, then we can choose the dictionary S as that subset of keys and thus force the hash table to offer its worst behavior, by boiling down to a single linked list of length $\Omega(n)$.

This argument is independent of the hash function h, so we can conclude that no hash function is robust enough to guarantee *always* a "good" behavior. In practice heuristics are used to create hash functions that perform well sufficiently often: The design principle is to compute the hash value in a way that it is expected to be independent of any regular pattern that might exist among the keys in S. The two most famous and practical hashing schemes are based on division and multiplication, and are briefly recalled below (for more details we refer to any classic text in Algorithms, such as [3].

Hashing by division. The hash value is computed as the remainder of the division of k by the table size m, that is: $h(k) = k \,\texttt{mod}\, m$. This is quite fast and behaves well as long as $h(k)$ does not depend on few bits of k. So power-of-two values for m should be avoided, whereas prime numbers not too much close to a power-of-two should be chosen. For the selection of large prime numbers do exist either simple, but slow (exponential time) algorithms (such as the famous Sieve of Eratosthenes method); or fast algorithms based on some (randomized or deterministic) *primality test*.[1] In general, the cost of prime selection is $o(m)$; and thus turns out to be negligible with respect to the cost of table allocation.

Hashing by multiplication. The hash value is computed in two steps: First, the key k is multiplied by a constant A, with $0 < A < 1$; then, the fractionary part of kA is multiplied by m and the integral part of the result is taken as index into the hash table T. In formula: $h(k) = \lfloor m \,\texttt{frac}(kA) \rfloor$. An

[1]The most famous, and randomized, primality test is the one by Miller and Rabin; more recently, a deterministic test has been proposed which allowed to prove that this problem is in $\mathcal{P}$. For some more details look at `http://en.wikipedia.org/wiki/Prime_number`.

advantage of this method is that the choice of m is not critical, and indeed it is usually chosen as a power of 2, thus simplifying the multiplication step. For the value of A, it is often suggested to take $A = (\sqrt{5} - 1)/2 \cong 0.618$.

It goes without saying that none of these practical hashing schemes surpasses the problem stated above: it is always possible to select a bad set of keys which makes the table T to boil down to a single linked list, e.g., just take multiples of m to disrupt the hashing-by-division method. In the next section, we propose an hashing scheme that is robust enough to guarantee a "good" behavior on average, whichever is the input dictionary.

7.3 Universal hashing

Let us first argue by a counting argument why the uniformity property, we required to *good* hash functions, is computationally hard to guarantee. Recall that we are interested in hash functions which map keys in U to integers in $\{0, 1, ..., m-1\}$. The total number of such hash functions is $m^{|U|}$, given that each key among the $|U|$ ones can be mapped into m slots of the hash table. In order to guarantee uniform distribution of the keys and independence among them, our hash function should be anyone of those ones. But, in this case, its representation would need $\Omega(\log_2 m^{|U|}) = \Omega(|U| \ \log_2 m)$ bits, which is really too much in terms of space occupancy and in the terms of computing time (i.e. it would take at least $\Omega(\frac{|U| \log_2 m}{\log_2 |U|})$ time to just read the hash encoding).

Practical hash functions, on the other hand, suffer of several *weaknesses* we mentioned above. In this section we introduce the powerful Universal Hashing scheme which overcomes these drawbacks by means of *randomization* proceeding similarly to what was done to make *more robust* the pivot selection in the Quicksort procedure (see Chapter 5). There, instead of taking the pivot from a fixed position, it was chosen *uniformly at random* from the underlying array to be sorted. This way no input was bad for the pivot-selection strategy, which being unfixed and randomized, allowed to spread the risk over the many pivot choices guaranteeing that most of them led to a good-balanced partitioning.

Universal hashing mimics this algorithmic approach into the context of hash functions. Informally, we do not set the hash function in advance (cfr. fix the pivot position), but we will choose the hash function *uniformly at random* from a *properly defined* set of hash functions (cfr. random pivot selection) which is defined in a way that it is very probable to pick a *good* hash for the current input set of keys S (cfr. the partitioning is balanced). Good function means one that minimizes the number of *collisions* among the keys in S, and can be computed in constant time. Because of the randomization, even if S is fixed, the algorithm will behave differently on various executions, but the nice property of Universal Hashing will be that, on average, the performance will be the expected one. It is now time to formalize these ideas.

DEFINITION 7.1
Let $\mathcal{H}$ be a finite collection of hash functions which map a given universe U of keys into integers in $\{0, 1, ..., m-1\}$. $\mathcal{H}$ is said to be **universal** if, and only if, for all pairs of distinct keys $x, y \in U$ it is:

$$|\{h \in \mathcal{H} : h(x) = h(y)\}| \leq \frac{|\mathcal{H}|}{m}$$

In other words, the class $\mathcal{H}$ is defined in such a way that a randomly-chosen hash function h from this set has a chance to make the distinct keys x and y to collide *no more than* $\frac{1}{m}$. This is exactly the basic property that we deployed when designing hashing with chaining (see the proof of Theorem 7.1). Figure 7.3 pictorially shows this concept.

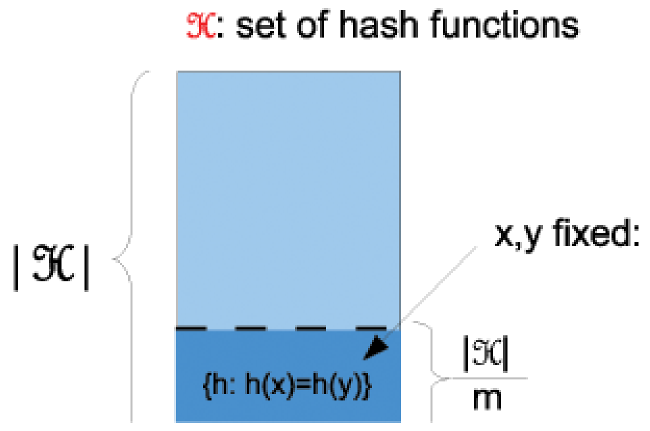


FIGURE 7.3: A schematic figure to illustrate the Universal Hash property.

It is interesting to observe that the definition of Universal Hashing can be extended with some slackness into the guarantee of probability of collision.

DEFINITION 7.2 Let c be a positive constant and $\mathcal{H}$ be a finite collection of hash functions that map a given universe U of keys into integers in $\{0, 1, ..., m-1\}$. $\mathcal{H}$ is said to be c**-universal** if, and only if, for all pairs of distinct keys $x, y \in U$ it is:

$$|\{h \in \mathcal{H} \ : \ h(x) = h(y)\}| \leq \frac{c|\mathcal{H}|}{m}$$

That is, for each pair of distinct keys, the number of hash functions for which there is a collision between this keys-pair is c times larger than what is guaranteed by universal hashing. The following theorem shows that we can use a universal class of hash functions to design a good hash table with chaining. This specifically means that Theorem 7.1 and its Corollaries 7.1–7.2 can be obtained by substituting the ideal Simple Uniform Hashing with Universal Hashing. This change will be effective in that, in the next section, we will define a real Universal class $\mathcal{H}$, thus making concrete all these mathematical ruminations.

THEOREM 7.2 *Let $T[0, m-1]$ be an hash table with chaining, and suppose that the hash function h is picked at random from a universal class $\mathcal{H}$. The expected length of the chaining lists in T, whichever is the input dictionary of keys S, is still no more than $1 + \alpha$, where α is the load factor n/m of the table T.*

Proof We note that the expectation here is over the choices of h in $\mathcal{H}$, and it does not depend on the distribution of the keys in S. For each pair of keys $x, y \in S$, define the indicator random variable I_{xy} which is 1 if these two keys collide according to a given h, namely $h(x) = h(y)$, otherwise it assumes the value 0. By definition of universal class, given the random choice of h, it is $P(I_{xy} = 1) = P(h(x) = h(y)) \leq 1/m$. Therefore we can derive $E[I_{xy}] = 1 \times P(I_{xy} = 1) + 0 \times P(I_{xy} = 0) = P(I_{xy} = 1) \leq 1/m$, where the average is computed over h's random choices.

Now we define, for each key $x \in S$, the random variable N_x that counts the number of keys other than x that hash to the slot $h(x)$, and thus collide with x. We can write N_x as $\sum_{\substack{y \in S \\ y \neq x}} I_{xy}$. By averaging, and applying the linearity of the expectation, we get $\sum_{\substack{y\in S\\ y\neq x}} E[I_{xy}] = (n-1)/m < \alpha$. By adding 1, because of x, the theorem follows. ■

We point out that the time bounds given for hashing with chaining are in expectation. This means that the average length of the lists in T is small, namely $O(\alpha)$, but there could be one or few lists which might be very long, possibly containing up to $\Theta(n)$ items. This satisfies Theorem 7.2 but is of course not a nice situation because it might occur sadly that the *distribution of the searches* privileges keys which belong to the very long lists, thus taking significantly more that the "average" time bound! In order to circumvent this problem, one should guarantee also a small upper bound on the length of the *longest list* in T. This can be achieved by putting some care when inserting items in the hash table.

THEOREM 7.3 *Let T be an hash table with chaining formed by m slots and picking an hash function from a universal class $\mathcal{H}$. Assume that we insert in T a dictionary S of $n = \Theta(m)$ keys, the expected length of the longest chain is $O(\frac{\log n}{\log\log n})$.*

Proof Let h be an hash function picked uniformly at random from $\mathcal{H}$, and let $Q(k)$ be the probability that exactly k keys of S are hashed by h to a particular slot of the table T. Given the universality of h, the probability that a key is assigned to a fixed slot is $\leq \frac{1}{m}$. There are $\binom{n}{k}$ ways to choose k keys from S, so the probability that a slot gets k keys is:

$$Q(k) = \binom{n}{k}\left(\frac{1}{m}\right)^k\left(\frac{m-1}{m}\right)^{n-k} < \frac{e^k}{k^k}$$

where the last inequality derives from Stirling's formula $k! > (k/e)^k$. We observe that there exists a constant $c < 1$ such that, fixed $m \geq 3$ and $k_0 = c\log m/\log\log m$, it holds $Q(k_0) < 1/m^3$.

Let us now introduce M as the length of the longest chain in T, and the random variable $N(i)$ denoting the number of keys that hash to slot i. Then we can write

$$\begin{aligned} P(M = k) &= P(\exists i : N(i) = k \text{ and } N(j) \leq k \text{ for } j \neq i) \\ &\leq P(\exists i : N(i) = k) \;\leq\; m\,Q(k) \end{aligned}$$

where the two last inequalities come, the first one, from the fact that probabilities are ≤ 1, and the second one, from the union bound applied to the m possible slots in T.

If $k \leq k_0$ we have $P(M = k_0) \leq mQ(k_0) < m\frac{1}{m^3} \leq 1/m^2$. If $k > k_0$, we can pick c large enough such that $k_0 > 3 > e$. In this case $e/k < 1$, and so $(e/k)^k$ decreases as k increases, tending to 0. Thus, we have $Q(k) < (e/k)^k \leq (e/k_0)^{k_0} < 1/m^3$, which implies again that $P(M = k) < 1/m^2$.
We are ready to evaluate the expectation of M:

$$\begin{aligned} E[M] &= \sum_{k=0}^{n} k \times P(M = k) = \sum_{k=0}^{k_0} k \times P(M = k) + \sum_{k=k_0+1}^{n} k \times P(M = k) && (7.1) \\ &\leq \sum_{k=0}^{k_0} k \times P(M = k) + \sum_{k=k_0+1}^{n} n \times P(M = k) && (7.2) \\ &\leq k_0 \sum_{k=0}^{k_0} P(M = k) + n \sum_{k=k_0+1}^{n} P(M = k) && (7.3) \\ &= k_0 \times P(M \leq k_0) + n \times P(M > k_0) && (7.4) \end{aligned}$$

We note that $P(M \leq k_0) \leq 1$ and

$$Pr(M > k_0) = \sum_{k=k_0+1}^{n} P(M = k) < \sum_{k=k_0+1}^{n} (1/m^2) < n(1/n^2) = 1/n.$$

By combining together all these inequalities we can conclude that $E[M] \leq k_0 + n(1/n) = k_0 + 1 = O(\log m/\log\log m)$, which is the thesis since we assumed $m = \Theta(n)$. ■

Two observations are in order at this point. The condition on $m = \Theta(n)$ can be easily guaranteed by applying the *doubling method* to the table T, as we showed in Theorem 7.2. The bound on the maximum chain length is on average, but it can be turned into worst case via a simple argument. We start by picking a random hash function $h \in \mathcal{H}$, hash every key of S into T, and see whether the condition on the length of the longest chain is at most twice the expected length $\log m/\log\log m$. If so, we use T for the subsequent search operations, otherwise we pick a new function h and re-insert all items in T. A constant number of trials suffice to satisfy that bound[2], thus taking $O(n)$ construction time in expectation.

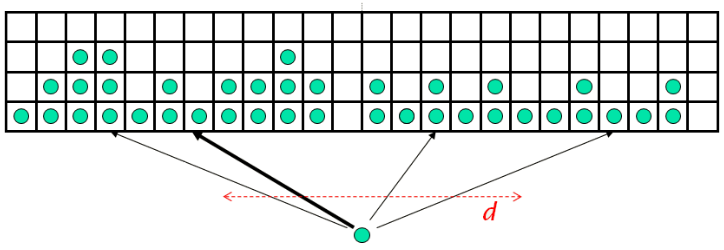


FIGURE 7.4: Example of d-left hashing with four subtables, four hash functions, and each table entry consisting of a bucket of a 4 slots.

Surprisingly enough, this result can be further improved by using two or more, say d, hash functions and d sub-tables $T_1, T_2, ..., T_d$ of the same size m/d, for a total space occupancy equal to the classic single hash table T. Each table T_i is indexed by a different hash function h_i ranging in $\{0, 1, \ldots, m/d - 1\}$. The specialty of this scheme resides in the implementation of the procedure `Insert(`k`)`: it tests the loading of the d slots $T_i[h_i(k)]$, and inserts k in the sparsest one. In the case of a tie, about slots' loading, the algorithm chooses the leftmost table, i.e. the one with minimum index i. For this reason this algorithmic scheme is also known as *d-left hashing*. The implementation of `Search(`k`)` follows consequently, we need to search all d lists $T_i[h_i(k)]$ because we do not know which were their loading when k was inserted, if any. The time cost for `Insert(`k`)` is $O(d)$ time, the time cost of `Search(`k`)` is given by the total length of the d searched lists. We can upper bound this length by d times the length of the longest list in T which, surprisingly, can be proved to be $O(\log\log n)$, when $d = 2$, and it is $\frac{\log\log n}{\log d} + O(1)$ for larger $d > 2$. This result has to be compared against the bound $O(\frac{\log n}{\log\log n})$ obtained for the case of a single hash function in Theorem 7.3. So, by just using one more hash function, we can get an exponential reduction in the search time: this sur-

[2]Just use the Markov bound to state that the longest list longer than twice the average may occur with probability $\leq 1/2$.

prising result is also known in the literature as *the power of two choices*, exactly because choosing between two slots the sparsest one allows to reduce exponentially the longest list.

As a corollary we notice that this result can be used to design a better hash table which does not use chaining-lists to manage collisions, thus saving the space of pointers and increasing locality of reference (hence less cache/IO misses). The idea is to allocate *small and fixed-size* buckets per each slot, as it is illustrated in Figure 7.4. We can use two hash functions and buckets of size $c \log\log n$, for some small $c > 1$. The important implication of this result is that even for just two hash functions there is a large reduction in the maximum list length, and thus search time.

7.3.1 Do universal hash functions exist?

The answer is positive and, surprisingly enough, universal hash functions can be easily constructed as we will show in this section for three classes of them. We assume, without loss of generality, that the table size m is a prime number and keys are integers represented as bit strings of $\log_2 |U|$ bits.[3] We let $r = \frac{\log_2 |U|}{\log_2 m}$ and assume that this is an integer. We decompose each key k in r parts, of $\log_2 m$ bits each, so $k = [k_0, k_1, ...k_{r-1}]$. Clearly each part k_i is an integer smaller than m, because it is represented in $\log_2 m$ bits. We do the same for a generic integer $a = [a_0, a_1, ..., a_{r-1}] \in [0, |U| - 1]$ used as the parameter that defines the universal class of hash functions $\mathcal{H}$ as follows: $h_a(k) = \sum_{i=0}^{r-1} a_i k_i \mod m$. The size of $\mathcal{H}$ is $|U|$, because we have one function per value of a.

THEOREM 7.4 *The class $\mathcal{H}$ that contains the following hash functions: $h_a(k) = \sum_{i=0}^{r-1} a_i k_i$* mod *m, where m is prime and a is a positive integer smaller than $|U|$, is universal.*

Proof Suppose that x and y are two distinct keys which differ, hence, on at least one bit. For simplicity of exposition, we assume that a differing bit falls into the first part, so $x_0 \neq y_0$. According to Definition 7.1, we need to count how many hash functions make these two keys collide; or equivalently, how many a do exist for which $h_a(x) = h_a(y)$. Since $x_0 \neq y_0$, and we operate in arithmetic modulo a prime (i.e. m), the inverse $(x_0 - y_0)^{-1}$ must exist and it is an integer in the range $[1, |U| - 1]$, and so we can write:

$$\begin{aligned} h_a(x) = h_a(y) &\Leftrightarrow \sum_{i=0}^{r-1} a_i x_i \equiv \sum_{i=0}^{r-1} a_i y_i \pmod m \\ &\Leftrightarrow a_0(x_0 - y_0) \equiv -\sum_{i=1}^{r-1} a_i(x_i - y_i) \pmod m \\ &\Leftrightarrow a_0 \equiv \left(-\sum_{i=1}^{r-1} a_i(x_i - y_i)\right)(x_0 - y_0)^{-1} \pmod m \end{aligned}$$

The last equation actually shows that, whatever is the choice for $[a_1, a_2, ..., a_{r-1}]$ (and they are m^{r-1}), there exists only one choice for a_0 (the one specified in the last line above) which causes x and y to collide. As a consequence, there are $m^{r-1} = |U|/m = |\mathcal{H}|/m$ choices for $[a_1, a_2, ..., a_{r-1}]$ that cause x and y to collide. So the Definition 7.1 of Universal Hash class is satisfied. ∎

[3]This is possible by pre-padding the key with 0, thus preserving its integer value.

It is possible to turn the previous definition holding for any table size m, thus not only for prime values. The key idea is to make a *double modular computation* by means of a large prime $p > |U|$, and a generic integer $m \ll |U|$ equal to the size of the hash table we wish to set up. We can then define an hash function parameterized in two values $a \geq 1$ and $b \geq 0$: $h_{a,b}(k) = ((ak + b) \bmod p) \bmod m$, and then define the family $\mathcal{H}_{p,m} = \bigcup_{a>0,b\geq 0}\{h_{a,b}\}$. It can be shown that $\mathcal{H}_{p,m}$ is a universal class of hash functions.

The above two definitions require r multiplications and r modulo operations. There are indeed other universal hashing which are faster to be computed because they rely only on operations involving power-of-two integers. As an example take $|U| = 2^h$, $m = 2^l < |U|$ and a be an odd integer smaller than $|U|$. Define the class $\mathcal{H}_{h,l}$ that contains the following hash functions: $h_a(k) = (ak \bmod 2^h)\ \mathtt{div}\ 2^{h-l}$. This class contains 2^{h-1} distinct functions because a is odd and smaller than $|U| = 2^h$. The following theorem presents the most important property of this class:

THEOREM 7.5 *The class $\mathcal{H}_{h,l} = \{\, h_a(k) = (ak \mod 2^h)\,\mathtt{div}\,2^{h-l}\}$, with $|U| = 2^h$ and $m = 2^l < |U|$ and a odd integer smaller than 2^h, is 2-universal because for any two distinct keys x and y, it is $P(h_a(x) = h_a(y)) \leq \dfrac{1}{2^{l-1}} = \dfrac{2}{m}$.*

Proof Without loss of generality let $x > y$ and define A as the set of possible values for a (i.e. a odd integer smaller than $2^h = |U|$). If there is a collision $h_a(x) = h_a(y)$, then we have:

$$\begin{aligned} ax \mod 2^h\,\mathtt{div}\,2^{h-l} - ay \mod 2^h\,\mathtt{div}\,2^{h-l} &= 0 \\ |ax \mod 2^h\,\mathtt{div}\,2^{h-l} - ay \mod 2^h\,\mathtt{div}\,2^{h-l}| &< 1 \\ |ax \mod 2^h - ay \mod 2^h| &< 2^{h-l} \\ |a(x-y) \mod 2^h| &< 2^{h-l} \end{aligned}$$

Set $z = x - y > 0$ (given that keys are distinct and $x > y$) and $z < |U| = 2^h$, it is $z \not\equiv 0(\mod 2^h)$ and $az \not\equiv 0(\mod 2^h)$ because a is odd, so we can write:

$$az \mod 2^h \in \{1, ..., 2^{h-l} - 1\} \cup \{2^h - 2^{h-l} + 1, ..., 2^h - 1\} \tag{7.5}$$

In order to estimate the number of $a \in A$ that satisfy this condition, we write z as $z'2^s$ with z' odd and $0 \leq s < h$. The odd numbers $a = 1, 3, 7, ..., 2^h - 1$ create a mapping $a \mapsto az' \mod 2^h$ that is a permutation of A because z' is odd. So, if we have the set $\{a2^s \mid a \in A\}$, a possible permutation is so defined: $a2^s \mapsto az'2^s \mod 2^h = az \mod 2^h$. Thus, the number of $a \in A$ that satisfy Eqn. 7.5 is the same as the number of $a \in A$ that satisfy:

$$a2^s \mod 2^h \in \{1, ..., 2^{h-l} - 1\} \cup \{2^h - 2^{h-l} + 1, ..., 2^h - 1\}$$

Now, $a2^s \mod 2^h$ is the number represented by the $h - s$ least significant bits of a, followed by s zeros. For example:

- If we take $a = 7$, $s = 1$ and $h = 3$:
  $7 * 2^1 \mod 2^3$ is in binary $111_2 * 10_2 \mod 1000_2$ that is equal to $1110_2 \mod 1000_2 = 110_2$. The result is represented by the $h - s = 3 - 1 = 2$ least significant bits of a, followed by $s = 1$ zeros.
- If we take $a = 7$, $s = 2$ and $h = 3$:
  $7*2^2 \mod 2^3$ is in binary $111_2*100_2 \mod 1000_2$ that is equal to $11100_2 \mod 1000_2 = 100_2$. The result is represented by the $h-s = 3-2 = 1$ least significant bits of a, followed by $s = 2$ zeros.

- If we take $a = 5$, $s = 2$ and $h = 3$:
  $5{*}2^2 \mod 2^3$ is in binary $101_2{*}100_2 \mod 1000_2$ that is equal to $10100_2 \mod 1000_2 = 100_2$. The result is represented by the $h-s = 3-2 = 1$ least significant bits of a, followed by $s = 2$ zeros.

So if $s \geq h - l$ there are no values of a that satisfy Eqn. 7.5, while for smaller s, the number of $a \in A$ satisfying that equation is at most 2^{h-l}. Consequently the probability of randomly choosing such a is at most $2^{h-l}/2^{h-1} = 1/2^{l-1}$. Finally, the universality of $\mathcal{H}_{h,l}$ follows immediately because $\frac{1}{2^{l-1}} < \frac{1}{m}$. ■

7.4 Perfect hashing, minimal, ordered!

The most known algorithmic scheme in the context of hashing is probably that of hashing with chaining, which sets $m = \Theta(n)$ in order to guarantee an average constant-time search; but that optimal time-bound is on average, as most students forget. This forgetfulness is not erroneous in absolute terms, because do indeed exist variants of hash tables that offer a constant-time *worst-case* bound, thus making hashing a competitive alternative to *tree-based* data structures and the *de facto* choice in practice. A crucial concept in this respect is *perfect hashing*, namely, a hash function which avoids collisions among the keys to be hashed (i.e. the keys of the dictionary to be indexed). Formally speaking,

DEFINITION 7.3 A hash function $h : U \to \{0, 1, \ldots, m - 1\}$ is said to be *perfect* with respect to a dictionary S of keys if, and only if, for any pair of distinct keys $k', k'' \in S$, it is $h(k') \neq h(k'')$.

An obvious counting argument shows that it must be $m \geq |S| = n$ in order to make perfect-hashing possible. In the case that $m = n$, i.e. the minimum possible value, the perfect hash function is named *minimal* (shortly `MPHF`). A hash table T using an `MPHF` h guarantees $O(1)$ worst-case search time as well as no waste of storage space, because it has the size of the dictionary S (i.e. $m = n$) and keys can be directly stored in the table slots. Perfect hashing is thus a sort of "perfect" variant of direct-address tables (see Section 7.1), in the sense that it achieves constant search time (like those tables), but optimal linear space (unlike those tables).

A (minimal) perfect hash function is said to be *order preserving* (shortly `OP(MP)HF`) iff, $\forall k_i < k_j \in S$, it is $h(k_i) < h(k_j)$. Clearly, if h is also minimal, and thus $m = n$; then $h(k)$ returns the *rank* of the key in the ordered dictionary S. It goes without saying that, the property `OP(MP)HF` strictly depends onto the dictionary S upon which h has been built: by changing S we could destroy this property, so it is difficult, even if not impossible, to maintain this property under a *dynamic* scenario. In the rest of this section we will confine ourselves to the case of *static* dictionaries, and thus a fixed dictionary S.

The design of h is based upon three auxiliary functions h_1, h_2 and g, which are defined as follows:

- h_1 and h_2 are two universal hash functions from strings to $\{0, 1, \ldots, m' - 1\}$ picked randomly from a universal class (see Section 7.3). They are not necessarily perfect, and so they might induce collisions among S's keys. The choice of m' impacts onto the efficiency of the construction, typically it is taken $m' = cn$, where $c > 1$, so spanning a range which is larger than the number of keys.
- g is a function that maps integers in the range $\{0, \ldots, m' - 1\}$ to integers in the range $\{0, \ldots, n - 1\}$. This mapping cannot be perfect, given that $m' \geq n$, and so some output

(a)	Term t	$h_1(t)$	$h_2(t)$	$h(t)$	(b)	x	$g(x)$
	body	1	6	0		0	0
	cat	7	2	1		1	5
	dog	5	7	2		2	0
	flower	4	6	3		3	7
	house	1	10	4		4	8
	mouse	0	1	5		5	1
	sun	8	11	6		6	4
	tree	11	9	7		7	1
	zoo	5	3	8		8	0
						9	1
						10	8
						11	6

TABLE 7.1 An example of an OPMPHF for a dictionary S of $n = 9$ strings which are in alphabetic order. Column $h(t)$ reports the lexicographic rank of each key; h is minimal because its values are in $\{0, \ldots, n-1\}$ and is built upon three functions: (a) two random hash functions $h_1(t)$ and $h_2(t)$, for $t \in S$; (b) a properly derived function $g(x)$, for $x \in \{0, 1, \ldots, m'\}$. Here $m' = 11 > 9 = n$.

values could be repeated. The function g is designed in a way that it *properly combines* the values of h_1 and h_2 in order to derive the OP(MP)HF h:

$$h(t) = (\, g(h_1(t)) + g(h_2(t))\,) \bmod n$$

The construction of g is obtained via an elegant randomized algorithm which deploys *paths in acyclic random graphs* (see below).

Examples for these three functions are given in the corresponding columns of Table 7.1. Although the values of h_1 and h_2 are randomly generated, the values of g are derived by a proper algorithm whose goal is to guarantee that the formula for $h(t)$ maps a string $t \in S$ to its lexicographic rank in S. It is clear that the evaluation of $h(t)$ takes constant time: we need to perform accesses to arrays h_1 and h_2 and g, plus two sums and one modular operation. The total required space is $O(m' + n) = O(n)$ whenever $m' = cn$. It remains to discuss the choice of c, which impacts onto the efficiency of the randomized construction of g and onto the overall space occupancy. It is suggested to take $c > 2$, which leads to obtain a successful construction in $\sqrt{\frac{m}{m-2n}}$ trials. This means about two trials by setting $c = 3$ (see [4]).

THEOREM 7.6 *An* OPMPHF *for a dictionary S of n keys can be constructed in $O(n)$ average time. The hash function can be evaluated in $O(1)$ time and uses $O(n)$ space (i.e. $O(n \log n)$ bits); both time and space bounds are worst case and optimal.*

Before digging into the technical details of this solution, let us make an important observation which highlights the power of OPMPHF. It is clear that we could assign the rank to each string by deploying a trie data structure (see Theorem **??**), but this would incur two main limitations: (i) rank assignment would need a trie search operation which would incur $\Theta(s)$ I/Os, rather than $O(1)$; (ii) space occupancy would be linear in the total dictionary length, rather than in the total dictionary cardinality. The reason is that, the OPMPHF's machinery does not need the string storage thus allowing to pay space proportional to the number of strings rather than their length; but on the other hand, OPMPHF does not allow to compute the rank of strings *not* in the dictionary, an operation which is instead supported by tries via the so called *lexicographic search* (see Chapter **??**).

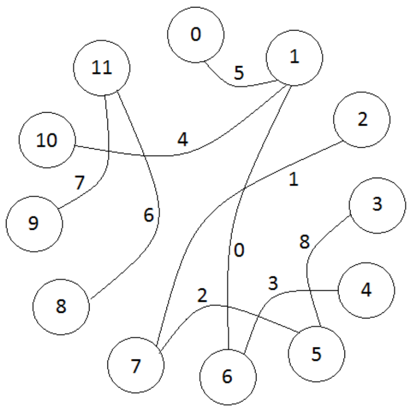


FIGURE 7.5: Graph corresponding to the dictionary of strings S and to the two functions h_1 and h_2 of Table 7.1.

We are left with detailing the algorithm that computes the function g, which will be actually implemented as an array of m' positions storing values bounded by n (see above). So g-array will occupy a total of $O(m' \log_2 n)$ bits. This array must have a quite peculiar feature: its entries have to be defined in a way that the computation

$$h(t) = (\, g(h_1(t)) + g(h_2(t))\,) \bmod n$$

returns the rank of term t in S. Surprisingly enough, the computation of g is quite simple and consists of building an undirected graph $G = (V, E)$ with m' nodes labeled $\{0, 1, \ldots, m' - 1\}$ (the same range as the co-domain of h_1 and h_2, and the domain of g) and n edges (as many as the keys in the dictionary) defined according to $(h_1(t), h_2(t))$ labeled with the *desired* value $h(t)$, for each dictionary string $t \in S$. It is evident that the topology of G depends only on h_1 and h_2, and it is exactly what it is called a *random graph*.[4]

Figure 7.5 shows an example of graph G constructed according to the setting of Table 7.1. Take the string t = body for which $h_1(t) = 1$ and $h_2(t) = 6$. The lexicographic rank of body in the dictionary S is $h(t) = 0$ (it is the first string), which is then the value labeling edge $(1, 6)$. We have to derive $g(t)$ so that 0 must be turn to be equal to $(\, g(1) + g(6)\,) \bmod 9$. Of course there are some correct values for entries $g(1)$ and $g(6)$ (e.g. $g(1) = 0$ and $g(6) = 0$), but these values should be correct for all terms t whose edges are incident onto the nodes 1 and 6. Because these edges refer to terms whose g's computation depends on $g(1)$ or $g(6)$. In this respect, the structure of G is useful to drive the instantiation of g's values, as detailed by the algorithm LabelAcyclicGraph(G).

The key algorithmic idea is that, if the graph originated by h_1 and h_2 is acyclic, then it can be decomposed into paths. If we assume that the first node, say v_0, of every path takes value 0, as indeed we execute LabelFrom(v_0, 0) in LabelAcyclicGraph(G), then all other nodes v_i subsequently traversed in this path will have value undef for $g(v_i)$ and thus this entry can be easily set by solving the following equation with respect to $g(v_i)$:

$$h(v_{i-1}, v_i) = (\, g(v_{i-1}) + g(v_i)\,) \bmod n$$

[4]The reader should be careful that the role of letters n and m' is exchanged here with respect to the traditional graph notation in which n refers to number of nodes and m' refers to number of edges.

Algorithm 7.1 Procedure `LabelAcyclicGraph`(G)

```
1: for $v \in V$ do
2:     $g[v] = \mathtt{undef}$
3: end for
4: for $v \in V$ do
5:     if $g[v] = \mathtt{undef}$ then
6:         LabelFrom$(v, 0)$
7:     end if
8: end for
```

which actually means to compute:

$$g(v_i) = (\, h(v_{i-1}, v_i) - g(v_{i-1})\,) \bmod n$$

Algorithm 7.2 Procedure `LabelFrom`(v, c)

```
1: if $g[v] \neq \mathtt{undef}$ then
2:     if $g[v] \neq c$ then
3:         return - the graph is cyclic; STOP
4:     end if
5: end if
6: $g[v] = c$
7: for $u \in \mathtt{Adj}[v]$ do
8:     LabelFrom$(u, h(v, u) - g[v] \mod n)$
9: end for
```

This is exactly what the algorithm `LabelFrom`(v, c) does. It is natural at this point to ask whether these algorithms always find a good assignment to function g or not. It is not difficult to convince ourselves that they return a solution only if the input graph G is acyclic, otherwise they stop. In this latter case, we have to rebuild the graph G, and thus the hash functions h_1 and h_2 by drawing them from the universal class.

What is the likelihood of building an acyclic graph ? This question is quite relevant since the technique is useful only if this *probability of success* is large enough to need few rebuilding steps. According to Random Graph Theory [4], if $m' \leq 2n$ this probability is almost equal to 0; otherwise, if $m' > 2n$ then it is about $\sqrt{\frac{m'-2n}{m'}}$, as we mentioned above. This means that the average number of graphs we will need to build before finding an acyclic one is: $\sqrt{\frac{m'}{m'-2n}}$; which is a constant number if we take $m' = \Theta(n)$. So, on average, the algorithm `LabelAcyclicGraph`(G) builds $\Theta(1)$ graphs of $m' + n = \Theta(n)$ nodes and edges, and thus it takes $\Theta(n)$ time to execute those computations. Space usage is $m' = \Theta(n)$.

In order to reduce the space requirements, we could resort multi-graphs (i.e. graphs with multiple edges between a pair of nodes) and thus use three hash functions, instead of two:

$$h\,(t) = (g\,(h_1\,(t)) + g\,(h_2\,(t)) + g\,(h_3\,(t))) \bmod n$$

We conclude this section by a running example that executes `LabelAcyclicGraph`(G) on the graph in Figure 7.5.

1. Select node 0, set $g(0) = 0$ and start to visit its neighbors, which in this case is just the node 1.
2. Set $g(1) = (h(0,1) - g(0))\texttt{mod}9 = (5 - 0)\texttt{mod}9 = 5$.
3. Take the unvisited neighbors of 1: 6 and 10, and visit them recursively.
4. Select node 6, set $g(6) = (h(1,6) - g(1))\texttt{mod}9 = (0 - 5)\texttt{mod}9 = 4$.
5. Take the unvisited neighbors of 6: 4 and 10, the latter is already in the list of nodes to be explored.
6. Select node 10, set $g(10) = (h(1,10) - g(1))\texttt{mod}9 = (4 - 5)\texttt{mod}9 = 8$.
7. No unvisited neighbors of 10 do exist.
8. Select node 4, set $g(4) = (h(4,6) - g(6))\texttt{mod}9 = (3 - 4)\texttt{mod}9 = 8$.
9. No unvisited neighbors of 4 do exist.
10. No node is left in the list of nodes to be explored.
11. Select a new starting node, for example 2, set $g(2) = 0$, and select its unvisited neighbor 7.
12. Set $g(7) = (h(2,7) - g(2))\texttt{mod}9 = (1 - 0)\texttt{mod}9 = 1$, and select its unvisited neighbor 5.
13. Set $g(5) = (h(7,5) - g(7))\texttt{mod}9 = (2 - 1)\texttt{mod}9 = 1$, and select its unvisited neighbor 3.
14. Set $g(3) = (h(3,5) - g(5))\texttt{mod}9 = (8 - 1)\texttt{mod}9 = 7$.
15. No node is left in the list of nodes to be explored.
16. Select a new starting node, for example 8, set $g(8) = 0$, and select its unvisited neighbor 11.
17. Set $g(11) = (h(8,11) - g(8))\texttt{mod}9 = (6 - 0)\texttt{mod}9 = 6$, and select its unvisited neighbor 9.
18. Set $g(9) = (h(11,9) - g(11))\texttt{mod}9 = (7 - 6)\texttt{mod}9 = 1$.
19. No node is left in the list of nodes to be explored.
20. Since all other nodes are isolated, their g's value is set to 0;

It goes without saying that, if S undergoes some insertions, we possibly have to rebuild $h(t)$. Therefore, all of this works for a static dictionary S.

7.5 A simple perfect hash table

If ordering and minimality (i.e. $h(t) < n$) is not required, then the design of a (static) perfect hash function is simpler. The key idea is to use a *two-level hashing scheme* with universal hashing functions at each level. The first level is essentially hashing with chaining, where the n keys are hashed into m slots using a universal hash function h; but, unlike chaining, every entry $T[j]$ points to a **secondary hash table** T_j which is addressed by another specific universal hash function h_j. By choosing h_j carefully, we can guarantee that there are no collisions at this secondary level. This way, the search for a key k will consist of two table accesses: one at T according to h, and one at some T_i according to h_i, given that $i = h(k)$. For a total of $O(1)$ time complexity in the worst case.

The question now is to guarantee that: (i) hash functions h_j are perfect and thus elicit no collisions among the keys mapped by h into $T[j]$, and (ii) the total space required by table T and all sub-tables T_j is the optimal $O(n)$. The following theorem is crucial for the following arguments.

THEOREM 7.7 *If we store q keys in a hash table of size $w = q^2$ using a universal hash function, then the expected number of collisions among those keys is less than* $1/2$. *Consequently, the probability to have a collision is less than* $1/2$.

Proof In a set of q elements there are $\binom{q}{2} < q^2/2$ pairs of keys that may collide; if we choose the function h from a universal class, we have that each pair collides with probability $1/w$. If we set $w = q^2$ the expected number of collisions is $\binom{q}{2}\frac{1}{w} < q^2/(2q^2) = \frac{1}{2}$. The probability to have at least one collision can be upper bounded by using the Markov inequality (i.e. $P(X \geq tE[X]) \leq 1/t$), with X expressing the number of collisions and by setting $t = 2$. ∎

We use this theorem in two ways: we will guarantee (i) above by setting the size m_j of hash table T_j as n_j^2 (the square of the number of keys hashed to $T[j]$); we will guarantee (ii) above by setting $m = n$ for the size of table T. The former setting ensures that every hash function h_j is perfect, by just two re-samples on average; the latter setting ensures that the total space required by the sub-tables is $O(n)$ as the following theorem formally proves.

THEOREM 7.8 *If we store n keys in a hash table of size $m = n$ using a universal hash function h, then the expected size of all sub-tables T_j is less than $2n$: in formula, $E[\sum_{j=0}^{m-1} n_j^2] < 2n$ where n_j is the number of keys hashing to slot j and the average is over the choices of h in the universal class.*

Proof Let us consider the following identity: $a^2 = a + 2\binom{a}{2}$ which is true for any integer $a > 0$. We have:

$$\begin{aligned} E[\sum_{j=0}^{m-1} n_j^2] &= E[\sum_{j=0}^{m-1}(n_j + 2\binom{n_j}{2})] \\ &= E[\sum_{j=0}^{m-1} n_j] + 2E[\sum_{j=0}^{m-1}\binom{n_j}{2}] \\ &= n + 2E[\sum_{j=0}^{m-1}\binom{n_j}{2}] \end{aligned}$$

The former term comes from the fact that $\sum_{j=0}^{m-1} n_j$ equals the total number of items hashed in the secondary level, and thus it is the total number n of dictionary keys. For the latter term we notice that $\binom{n_j}{2}$ accounts for the number of collisions among the n_j keys mapped to $T[j]$, so that $\sum_{j=0}^{m-1}\binom{n_j}{2}$ equals the number of collisions induced by the primary-level hash function h. By repeating the argument adopted in the proof of Theorem 7.7 and using $m = n$, the expected value of this sum is at most $\binom{n}{2}\frac{1}{m} = \frac{n(n-1)}{2m} = \frac{n-1}{2}$. Summing these two terms we derive that the total space required by this two-level hashing scheme is bounded by $n + 2\frac{n-1}{2} = 2n - 1$. ∎

It is important to observe that every hash function h_j is independent on the others, so that if it generates some collisions among the n_j keys mapped to $T[j]$, it can be re-generated without influencing the other mappings. Analogously if at the first level h generates more than zn collisions, for some large constant z, then we can re-generate h. These theorems ensure that the average number of re-generations is a small constant per hash function. In the following subsections we detail insertion and searching with perfect hashing.

Create a perfect hash table

In order to create a perfect hash table from a dictionary S of n keys we proceed as indicated in the pseudocode of Figure 7.3. Theorem 7.8 ensures that the probability to extract a good function from the family $H_{p,m}$ is at least $1/2$, so we have to repeat the extraction process (at the first level) an average number of 2 times to guarantee a successful extraction, which actually means $L < 2n$. At the second level Theorem 7.7 and the setting $m_j = n_j^2$ ensure that the probability to have a collision in table T_j because of the hash function h_j is lower than $1/2$. SO, again, we need on average two extractions for h_j to construct a perfect hash table T_j.

Algorithm 7.3 Creating a Perfect Hash Table

```
Choose a universal hash function $h$ from the family $H_{p,m}$;
for $j = 0, 1, \ldots, m-1$ do
    $n_j = 0, S_j = \emptyset$;
end for
for $k \in S$ do
    $j = h(k)$;
    Add $k$ to set $S_j$;
    $n_j = n_j + 1$;
end for
Compute $L = \sum_{j=0}^{m-1} (n_j)^2$;
if $L \geq 2n$ then
    Repeat the algorithm from Step 1;
end if
for $j = 0, 1, \ldots, m-1$ do
    Construct table $T_j$ of size $m_j = (n_j)^2$;
    Choose hash function $h_j$ from class $H_{p,m_j}$;
    for $k \in S_j$ do
        $i = h_j(k)$;
        if $T_j[i] \neq$ NULL then
            Destroy $T_j$ and repeat from step 15;
        end if
        $T_j[i] = k$;
    end for
end for
```

Searching keys with perfect hashing

Searching for a key k in a perfect hash table T takes just two memory accesses, as indicated in the pseudocode of Figure 7.4.

With reference to Figure 7.6, let us consider the successful search for the key $98 \in S$. It is $h(98) = ((2 \times 98 + 42) \bmod 101) \bmod 11 = 3$. Since T_3 has size $m_3 = 4$, we apply the second-level hash function $h_3(98) = ((4 \times 98 + 42) \bmod 101) \bmod 4 = 2$. Given that $T_3(2) = 98$, we have found the searched key.

Let us now consider the unsuccessful search for the key $k = 8 \notin S$. Since it is $h(8) = ((2 \times 8 + 42) \bmod 101) \bmod 11 = 3$, we have to look again at the table T_3 and compute $h_3(8) = ((4 \times 8 + 42) \bmod 101) \bmod 4 = 2$. Given that $T_3(2) = 19$, we conclude that the key $k = 8$ is not in the dictionary.

7.6 Cuckoo hashing

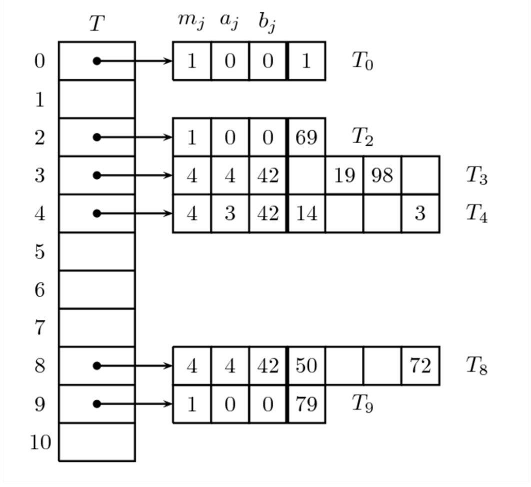


FIGURE 7.6: Creating a perfect hash table over the dictionary $S = \{98, 19, 14, 50, 1, 72, 79, 3, 69\}$, with $h(k) = ((2k + 42) \bmod 101) \bmod 11$. Notice that $L = 1 + 1 + 4 + 4 + 4 + 1 = 15 < 2 \times 9$ and, at the second level, we use the same hash function changing only the table size for the modulo computation. There no collision occur.

When the dictionary is dynamic a different hashing scheme has to be devised, an efficient and elegant solution is the so called **cuckoo hashing**: it achieves constant time in updates, on average, and constant time in searches, in the worst case. The only drawback of this approach is that it makes use of two hash functions that are $O(\log n)$-independent (new results in the literature have significantly relaxed this requirement [1] but we stick on the original scheme for its simplicity). In pills, cuckoo hashing combines the multiple-choice approach of d-left hashing with the ability to move elements. In its simplest form, cuckoo hashing consists of two hash functions h_1 and h_2 and one table T of size m. Any key k is stored either at $T[h_1(k)]$ or at $T[h_2(k)]$, so that searching and deleting operations are trivial: we need to look for k in both those entries, and eventually remove it. Inserting a key is a little bit more tricky in that it can trigger a *cascade* of key moves in the table. Suppose a new key k has to be inserted in the dictionary, according to the Cuckoo scheme it has to be inserted either at position $h_1(k)$ or at position $h_2(k)$. If one of these locations in T is empty (if both are, $h_1(k)$ is chosen), the key is stored at that position and the insertion process is completed. Otherwise, both entries are occupied by other keys, so that we have to create room for k by evicting one of the two keys stored in those two table entries. Typically, the key y stored in $T[h_1(k)]$ is evicted and replaced with k. Then, y plays the role of k and the insertion process is repeated.

There is a warning to take into account at this point. The key y was stored in $T[h_1(k)]$, so that $T[h_i(y)] = T[h_1(k)]$ for either $i = 1$ or $i = 2$. This means that if both positions $T[h_1(y)]$ and $T[h_2(y)]$ are occupied, the key to be evicted cannot be chosen from the entry that was storing $T[h_1(k)]$ because it is k, so this would induce a trivial infinite cycle of evictions over this entry between keys k and y. The algorithm therefore is careful to always avoid to evict the previously inserted key. Nevertheless cycles may arise (e.g. consider the trivial case in which $\{h_1(k), h_2(k)\} = \{h_1(y), h_2(y)\}$) and they can be of arbitrary length, so that the algorithm must be careful in defining an *efficient escape condition* which detects those situations, in which case it re-sample the two hash functions and re-hash all dictionary keys. The key property, proved in Theorem 7.9, will be to show that cycles occurs with

Algorithm 7.4 Procedure `Search(k)` in a Perfect Hash Table T

```
1: Let $h$ and $h_j$ be the universal hash functions defining $T$;
2: Compute $j = h(k)$;
3: if $T_j$ is empty then
4:     Return false; // $k \notin S$
5: end if
6: Compute $i = h_j(k)$;
7: if $T_j[i] =$ NULL then
8:     Return false; // $k \notin S$
9: end if
10: if $T_j[i] \neq k$ then
11:     Return false; // $k \notin S$
12: end if
13: Return true; // $k \in S$
```

bounded probability, so that the $O(n)$ cost of re-hashing can be amortized, by charging $O(1)$ time per insertion (Corollary 7.4).

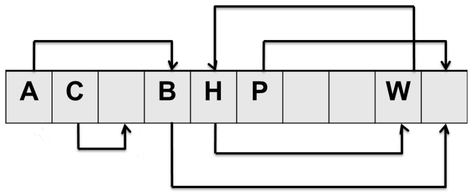


FIGURE 7.7: Graphical representation of cuckoo hashing.

In order to analyze this situation it is useful to introduce the so called *cuckoo graph* (see Figure 7.7), whose nodes are entries of table T and edges represent dictionary keys by connecting the two table entries where these keys can be stored. Edges are directed to keep into account where a key is stored (source), and where a key could be alternatively stored (destination). This way the cascade of evictions triggered by key k traverses the nodes (table entries) laying on a directed path that starts from either node (entry) $h_1(k)$ or node $h_2(k)$. Let us call this path the *bucket* of k. The bucket reminds the list associated to entry $T[h(k)]$ in hashing with chaining (Section 7.2), but it might have a more complicated structure because the cuckoo graph can have cycles, and thus this path can form loops as it occurs for the cycle formed by keys `W` and `H`.

For a more detailed example of insertion process, let us consider again the cuckoo graph depicted in Figure 7.7. Suppose to insert key `D` into our table, and assume that $h_1(D) = 4$ and $h_2(D) = 1$, so that D evicts either `A` or `B` (Figure 7.8). We put `D` in table entry 1, thereby evicting `A` which tries to be stored in entry 4 (according to the directed edge). In turn, `A` evicts `B`, stored in 4, which is moved to the last location of the table as its possible destination. Since such a location is free, `B` goes there and the insertion process is successful and completed. Let us now consider the insertion of key `F`, and assume two cases: $h_1(F) = 2$ and $h_2(F) = 5$ (Figure 7.9.a), or $h_1(F) = 4$ and $h_2(F) = 5$ (Figure 7.9.b). In the former case the insertion is successful: `F` causes `C` to be evicted, which in turn finds the third location empty. It is interesting to observe that, even if we check first $h_2(F)$, then the insertion is still successful: F causes H to be evicted, which causes W to be evicted, which in turn causes again F to be evicted. We found a cycle which nevertheless does not induces an infinite loop, in fact in this case the eviction of F leads to check its second possible location, namely $h_1(F) = 2$, which evicts C that is then stored in the third location (currently empty). Consequently the existence of a cycle does not imply an unsuccessful search; said this, in the following, we will compute the probability of the existence of a cycle as an upper bound to the probability of an unsuccessful search. Conversely, the case of an unsuccessful insertion occurs in Figure 7.9.b where there are two cycles F-H and A-B which gets glued together by the insertion of the key F. In this case the keys flow from one cycle to the other without stopping. As a consequence, the insertion algorithm must be designed in order to check whether the traversal of the cuckoo graph ended up in an infinite loop: this is done *approximately* by bounding the number of eviction steps.

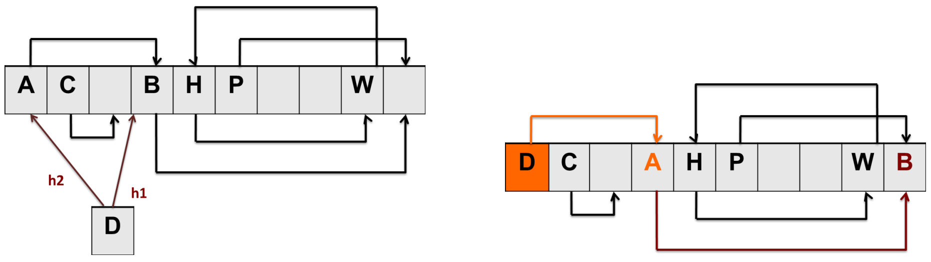


FIGURE 7.8: Inserting the key D: (left) the two entry options, (right) the final configuration.

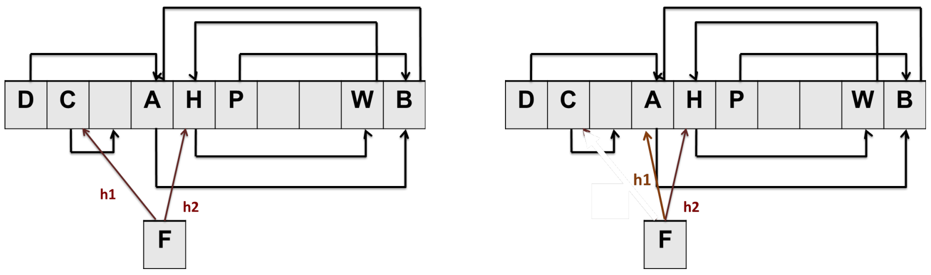


FIGURE 7.9: Inserting the key F: (left) successful insertion, (right) unsuccessful insertion.

7.6.1 An analysis

Querying and deleting a key k takes constant time, only two table entries have to be checked. The case of insertion is more complicated because it has necessarily to take into account the formation of paths in the (random) cuckoo graph. In the following we consider an *undirected* cuckoo graph, namely one in which edges are not oriented, and observe that a key y is in the bucket of another key x only if there is a path between one of the two positions (nodes) of x and one of the two positions (nodes) of y in the cuckoo graph. This relaxation allows to easy the bounding of the probability of the existence of these paths (recall that m is the table size and n is the dictionary size):

THEOREM 7.9 *For any entries i and j and any constant $c > 1$, if $m \geq 2cn$, then the probability that in the undirected cuckoo graph there exists a shortest path from i to j of length $L \geq 1$ is at most c^{-L}/m.*

Proof We proceed by induction on the path length L. The base case $L = 1$ corresponds to the existence of the undirected edge (i, j); now, every key can generate that edge with probability no more than $\leq 2/m^2$, because edge is undirected and $h_1(k)$ and $h_2(k)$ are uniformly random choices among the m table entries. Summing over all n dictionary keys, and recalling that $m \geq 2cn$, we get the bound $\sum_{k \in S} 2/m^2 = 2n/m^2 \leq c^{-1}/m$.

For the inductive step we must bound the probability that there exists a path of length $L > 1$, but no path of length less than L connects i to j (or vice versa). This occurs only if, for some table entry h, the following two conditions hold:

- there is a shortest path of length $L - 1$ from i to z (that clearly does not go through j);
- there is an edge from z to j.

By the inductive hypothesis, the probability that the first condition is true is bounded by $c^{-(L-1)}/m = c^{-L+1}/m$. The probability of the second condition has been already computed and it is at most $c^{-1}/m = 1/cm$. So the probability that there exists such a path (passing through z) is $(1/cm) * (c^{-L+1}/m) = c^{-L}/m^2$. Summing over all m possibilities for the table entry z, we get that the probability of a path of length L between i and j is at most c^{-L}/m. ■

In other words, this Theorem states that if the number m of nodes in the cuckoo graph is sufficiently large compared to the number n of edges (i.e. $m \geq 2cn$), there is a low probability that any two nodes i and j are connected by a path, thus fall in the same bucket, and hence participate in a cascade of evictions. Very significant is the case of a constant-length path $L = O(1)$, for which the probability of occurrence is $O(1/m)$. This means that, even for this restricted case, the probability of a large bucket is small and thus the probability of a not-constant number of evictions is small. We can related this probability to the *collision probability* in hashing with chaining. We have therefore proved the following:

THEOREM 7.10 *For any two distinct keys x and y, the probability that x hashes to the same bucket of y is $O(1/m)$.*

Proof If x and y are in the same bucket, then there is a path of some length L between one node in $\{h_1(x), h_2(x)\}$ and one node in $\{h_1(y), h_2(y)\}$. By Theorem 7.9, this occurs with probability at most $4 \sum_{L=1}^{\infty} c^{-L}/m = \frac{4}{c-1}/m = O(1/m)$, as desired. ■

What about rehashing? How often do we have to rebuild table T? Let us consider a sequence of operations involving ϵn insertions, where ϵ is a small constant, e.g. $\epsilon = 0.1$, and assume that the table size is sufficiently large to satisfy the conditions imposed in the previous theorems, namely $m \geq 2cn + 2c(\epsilon n) = 2cn(1 + \epsilon)$. Let S' be the final dictionary in which all ϵn insertions have been performed. Clearly, there is a re-hashing of T only if some key insertion induced an infinite loop in the cuckoo graph. In order to bound this probability we consider the final graph in which all keys S' have been inserted, and thus all cycles induced by their insertions are present. This graph consists of m nodes and $n(1 + \epsilon)$ keys. Since we assumed $m \geq 2cn(1 + \epsilon)$, according to the Theorem 7.9, any given position (node) is involved in a cycle (of any length) if it is involved in a path (of any length) that starts and end at that position: the probability is at most $\sum_{L=1}^{\infty} c^{-L}/m$. Thus, the probability that there is a cycle of any length involving any table entry can be bounded by summing over all m

table entries: namely, $m \sum_{L=1}^{\infty} c^{-L}/m = \frac{1}{c-1}$. As we observed previously, this is an upper bound to the probability of an unsuccessful insertion given that the presence of a cycle does not necessarily imply an infinite loop for an insertion.

COROLLARY 7.3 By setting $c = 3$, and taking a cuckoo table of size $m \geq 6n(1+\epsilon)$, the probability for the existence of a cycle in the cuckoo graph of the final dictionary S' is at most $1/2$.

Therefore a constant number of re-hashes are enough to ensure the insertion of ϵn keys in a dictionary of size n. Given that the time for one rehashing is $O(n)$ (we just need to compute two hashes per key), the expected time for all rehashing is $O(n)$, which is $O(1/\epsilon)$ per insertion.

COROLLARY 7.4 By setting $c > 2$, and taking a cuckoo table of size $m \geq 2cn(1+\epsilon)$, the cost for inserting ϵn keys in a dictionary of size n by cuckoo hashing is constant expected amortized. Namely, expected with respect to the random selection of the two universal hash functions driving the cuckoo hashing, and amortised over the $\Theta(n)$ insertions.

In order to make the algorithm works for every n and ϵ, we can adopt the same idea sketched for hashing with chaining and called *global rebuilding technique*. Whenever the size of the dictionary becomes too small compared to the size of the hash table, a new, smaller hash table is created; conversely, if the hash table fills up to its capacity, a new, larger hash table is created. To make this work efficiently, the size of the hash table is increased or decreased by a constant factor (larger than 1), e.g. doubled or halved.

The cost of rehashing can be further reduced by using a very small amount (i.e. constant) of extra-space, called a *stash*. Once a failure situation is detected during the insertion of a key k (i.e. k incurs in a loop), then this key is stored in the stash (without rehashing). This reduces the rehashing probability to $\Theta(1/n^{s+1})$, where s is the size of the stash. The choice of parameter s is related to some structural properties of the cuckoo graph and of the universal hash functions, which are too involved to be commented here (for details see [1] and refs therein).

7.7 Bloom filters

There are situations in which the universe of keys is very large and thus every key is long enough to take a lot of space to be stored. Sometimes it could even be the case that the storage of table pointers, taking $(n+m)\log n$ bits, is much smaller than the storage of the keys, taking $n \log_2 |U|$ bits. An example is given by the dictionary of URLs managed by crawlers in search engines; maintaining this dictionary in internal memory is crucial to ensure the fast operations over those URLs required by crawlers. However URLs are thousands of characters long, so that the size of the indexable dictionary in internal memory could be pretty much limited if whole URLs should have to be stored. And, in fact, crawlers do not use neither cuckoo hashing nor hashing with chaining but, rather, employ a simple and randomised, yet efficient, data structure named *Bloom filter*. The crucial property of Bloom filters is that keys are *not explicitly stored*, only a small fingerprint of them is, and this induces the data structure to make a *one-side error* in its answers to membership queries whenever the queried key is not in the currently indexed dictionary. The elegant solution proposed by Bloom filters is that those errors can be controlled, and indeed their probability *decreases exponentially* with the size of the fingerprints of the dictionary keys. Practically speaking tens of bits (hence, few bytes) per fingerprint are enough to guarantee tiny error probabilities[5] and succinct space occupancy, thus making this solution much appealing in a big-data context. It is useful at this point recall the *Bloom filter principle*: "Wherever a list or set is used, and space is a consideration, a Bloom filter should be considered. When using a Bloom filter, consider the potential effects of false positives".

Let $S = \{x_1, x_2, ..., x_n\}$ be a set of n keys and B a bit vector of length m. Initially, all bits in B are set to 0. Suppose we have r universal hash functions $h_i : U \longrightarrow \{0, ..., m-1\}$, for $i = 1, ..., r$. As anticipated above, every key k is not represented explicitly in B but, rather, it is fingerprinted by setting r bits of B to 1 as follows: $B[h_i(k)] = 1, \forall\, 1 \le i \le r$. Therefore, inserting a key in a Bloom filter requires $O(r)$ time, and sets at most r bits (possibly some hashes may collide). For searching, we claim that a key y is in S if $B[h_i(y)] = 1, \forall\, 1 \le i \le r$. Searching costs $O(r)$, as well as inserting. In the example of Figure 7.10, we can assert that $y \notin S$, since three bits are set to 1 but the last checked bit $B[h_4(y)]$ is zero.

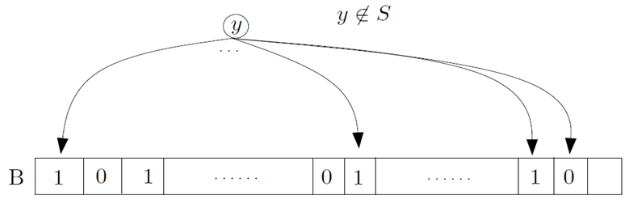


FIGURE 7.10: Searching key y in a Bloom filter.

Clearly, if $y \in S$ the Bloom filter correctly detects this; but it might be the case that $y \notin S$ and nonetheless all r bits checked are 1 because of the setting due to other hashes and keys. This is called *false positive* error, because it induces the Bloom filter to return a positive but erroneous answer to a membership query. It is therefore natural to ask for the probability of a false-positive error, which can be proved to be bounded above by a surprisingly simple formula.

The probability that the insertion of a key $k \in S$ has left null a bit-entry $B[j]$ equals the probability that the r independent hash functions $h_i(k)$ returned an entry different of j, which is $\left(\frac{m-1}{m}\right)^r \approx e^{-\frac{r}{m}}$. After the insertion of all n dictionary keys, the probability that $B[j]$ is still null can be then bounded by $p_0 \approx \left(e^{-\frac{r}{m}}\right)^n = e^{-\frac{rn}{m}}$ by assuming independencies among those hash functions.[6] Hence the probability of a false-positive error (or, equivalently, the false positive rate) is the probability that all r bits checked for a key not in the current dictionary are set to 1, that is:

$$p_{err} = (1 - p_0)^r \approx \left(1 - e^{-\frac{rn}{m}}\right)^r$$

Not surprisingly the error probability depends on the three parameters that define the Bloom filter's structure: the number r of hash functions, the number n of dictionary keys, and the number m of bits in the binary array B. It's interesting to notice that the fraction $f = m/n$ can be read as the average number of bits per dictionary key allocated in B, hence the fingerprint size f. The larger is

[5]One could object that, errors anyway might occur. But programmers counteract by admitting that these errors can be made smallers than hardware/network errors in data centers or PCs. So they can be neglected!

[6]A more precise analysis is possible, but much involved and without changing the final result, so that we prefer to stick on this simpler approach.

f the smaller is the error probability p_{err}, but the larger is the space allocated for B. We can optimize p_{err} according to m and n, by computing the first-order derivative and equalling it to zero: this gets $r = \frac{m}{n} \ln 2$. It is interesting to observe that for this value of r the probability a bit in B gets null value is $p_0 = 1/2$; which actually means that the array is half filled by 1s and half by 0s. And indeed this result could not be different: a larger r induces more 1s in B and thus a larger probability of positive errors, a lower r induces more 0s in B and thus a larger probability of correct answers: the correct choice of r falls in the middle! For this value of $r = \frac{m}{n} \ln 2$, we have $p_{err} = (0.6185)^{m/n}$ which decreases exponentially with the fingerprint size $f = m/n$. Figure 7.11 reports the false positive rate as a function of the number r of hashes for a Bloom-filter designed to use $m = 32n$ bits of space, hence a fingerprint of $f = 32$ bits per key. By using 22 hash functions we can minimize the false positive rate to less than $1.E - 6$. However, we also note that adding one more hash function does not significantly decreases the error rate when $r \geq 10$.

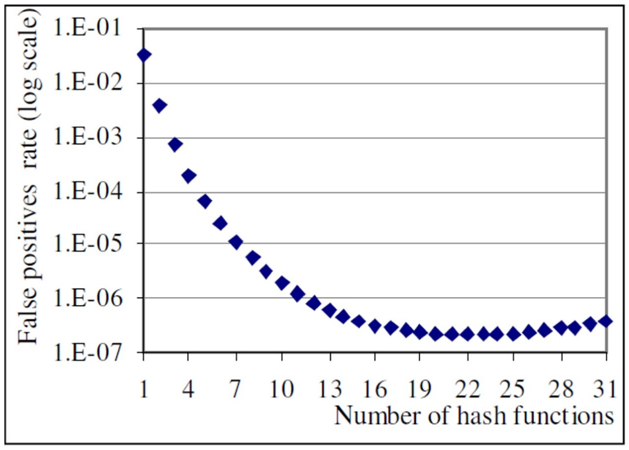


FIGURE 7.11: Plotting p_{err} as a function of the number of hashes.

It is natural now to derive the size m of the B-array whenever r is fixed to its optimal value $\frac{m}{n} \ln 2$, and n is the number of current keys in the dictionary. We obtain different values of p_{err} depending on the choice of m. For example, if $m = n$ then $p_{err} = 0.6185$, if $m = 2n$ then $p_{err} = 0.38$, and if $m = 5n$ we have $p_{err} = 0.09$. In practice $m = cn$ is a good choice, and for $c > 10$ the error rate is interestingly small and useful for practical applications. Figure 7.12 compares the performance of hashing (with chaining) to that of Bloom filters, assuming that the number r of used hash functions is the one which minimizes the false-positive error rate. In hashing with chaining, we need $\Theta(n(\log n + \log u))$ bits to store the pointers in the chaining lists, and $\Theta(m \log n$ is the space occupancy of the table, in bits. Conversely the Bloom filter does not need to store keys and thus it incurs only in the cost of storing the bit array $m = fn$, where f is pretty small in practice as we observed above.

7.7.1 A lower bound on space

The question is how small can be the bit array B in order to guarantee an given error rate ϵ for a dictionary of n keys drawn from a universe U of size u. This lower bound will allow us to prove that space-wise Bloom filters are within a factor of $\log_2 e \approx 1.44$ of the asymptotic lower bound.

The proof proceeds as follows. Let us represent any data structure solving the membership query

$\diamond$	Hash Tables	Bloom Filters
build time	$\Theta(n)$	$\Theta(n)$
space needed	$\Theta((m+n)\log n + n\log\lvert U\rvert)$	$\Theta(m)$
search time	$O(1)$	$(m/n)\ln 2$
ε value	0	$(0.6185)^{m/n}$

FIGURE 7.12: Hashing vs Bloom filter

on a dictionary $X \subseteq U$ with those time/error bounds with a m-bit string $F(X)$. This data structure must work for every possible subset of n elements of U, they are $\binom{u}{n}$. We say that an m-bit string s accepts a key x if $s = F(X)$ for some X containing x, otherwise we say that s rejects x. Now, let us consider a specific dictionary X of n elements. Any string s that is used to represent X must accept each one of the n elements of X, since no false negatives are admitted, but it may also accept at most $\epsilon(u-n)$ other elements of the universe, thus guaranteeing a false positive rate smaller than ϵ. Each string s therefore accepts at most $n+\epsilon(u-n)$ elements, and can thus be used to represent any of the $\binom{n+\epsilon(u-n)}{n}$ subsets of size n of these elements, but it cannot be used to represent any other set. Since we are interested into data structures of m bits, they are 2^m, and we are asking ourselves whether they can represent all the $\binom{u}{n}$ possible dictionaries of U of n keys. Hence, we must have:

$$2^m \times \binom{n+\epsilon(u-n)}{n} \geq \binom{u}{n}$$

or, equivalently:

$$m \geq \log_2 \binom{u}{n} / \binom{n+\epsilon(u-n)}{n} \;\geq\; \log_2 \binom{u}{n} / \binom{\epsilon u}{n} \;\geq\; \log_2 \epsilon^{-n} = n \log_2(1/\epsilon)$$

where we used the inequalities $(\frac{a}{b})^b \leq \binom{a}{b} \leq (\frac{ae}{b})^b$ and the fact that in pratice it is $u \gg n$. If we consider a Bloom filter with the same configuration— namely, error rate ϵ and space occupancy m bits—, then we have $\epsilon = (1/2)^r \geq (1/2)^{m \ln 2/n}$ by setting r to the optimal number of hash functions. After some algebraic manipulations, we find that:

$$m \geq n \frac{\log_2(1/\epsilon)}{\ln 2} \approx 1.44\, n\; \log_2(1/\epsilon)$$

This means that Bloom filters are asymptotically optimal in space, the constant factor is 1.44 more than the minimum possible.

7.7.2 Compressed Bloom filters

In many Web applications, the Bloom filter is not just an object that resides in memory, but it is a data structure that must be transferred between proxies. In this context it is worth to investigate whether Bloom filters can be *compressed* to save bandwidth and transfer time [3]. Suppose that we optimize the false positive rate of the Bloom filter under the constraint that the number of bits to be sent after compression is $z \leq m$. As compression tool we can use Arithmetic coding (see Chapter **??**), which well approximates the entropy of the string to be compressed: here simply expressed as $-(p(0)\log_2 p(0) + p(1)\log_2 p(1))$ where $p(b)$ is the frequency of bit b in the input string. Surprisingly enough it turns out that using a larger, but sparser, Bloom filter can yield the same false positive rate with a smaller number of transmitted bits. Said in other words, one can transmit the same number of bits but reduce the false positive rate. An example is given in Table 7.2, where the goal is to obtain small false positive rates by using less than 16 transmitted bits per element. Without compression, the optimal number of hash functions is 11, and the false positive rate is 0.000459. By making a sparse Bloom filter using 48 bits per element but only 3 hash functions, one can compress the result down to less than 16 bits per item (with high probability) and decrease the false positive rate by roughly a factor of 2.

Array bits per element	m/n	16	28	48
Transmission bits per element	z/n	16	15,846	15,829
Hash functions	k	11	4	3
False positive probability	f	0,000459	0,000314	0,000222

TABLE 7.2 Using at most sixteen bits per element after compression, a bigger but sparser Bloom filter can reduce the false positive rate.

Compressing a Bloom filter has benefits: (i) it uses a smaller number of hash functions, so that the lookups are more efficient; (ii) it may reduce the false positive rate for a desired compressed size, or reduce the transmited size for a fixed false positive rate. However the size m of the uncompressed Bloom filter increases the memory usage at running time, and comes at the computational cost of compressing/decompressing it. Nevertheless, some sophisticate approaches are possible which allow to access directly the compressed data without incurring in their decompression. An example was given by the FM-index in Chapter **??**, which could built over the bit-array B.

7.7.3 Spectral Bloom filters

A *spectral bloom filter* (SBF) is an extension of the original Bloom filter to multi-sets, thus allowing the storage of multiplicities of elements, provided that they are below a given threshold (a *spectrum* indeed). SBF supports queries on the multiplicities of a key with a small error probability over its estimate, using memory only slightly larger than that of the original Bloom filter. SBF supports also insertions and deletions over the data set.

Let S be a multi-set consisting of n distinct keys from U and let f_x be the multiplicity of the element $x \in S$. In a SBF the bit vector B is replaced by an array of counters $C[0, m-1]$, where $C[i]$ is the sum of f_x-values for those elements $x \in S$ mapping to position i. For every element x, we add the value f_x to the counters $C[h_1(x)], C[h_2(x)], ..., C[h_r(x)]$. Due to possible conflicts among hashes for different elements, the $C[i]$s provide approximated values, specifically upper bounds (given that $f_x > 0$.

The multi-set S can be dynamic. When inserting a new item s, its frequency f_s increases by one, so that we increase by one the counters $C[h_1(s)], C[h_2(s)], \ldots, C[h_r(s)]$; deletion consists symmetrically in decreasing those counters. In order to search for the frequency of element x, we simply return the minimum value $m_x = \min_i C[h_i(x)]$. Of course, m_x is a biased estimator for f_x. In particular, since all $f_x \leq C[h_i]$ for all i, the case in which the estimate is wrong (i.e. $m_x < f_x$) corresponds to the event "all counters $C[h_i(x)]$ have a collision", which in turn corresponds to a "false positive" event in the classical Bloom filter. So, the probability of error in a SBF is the error rate probability for a Bloom filter with the same set of parameters m, n, r.

7.7.4 A simple application

Bloom filters can be used to approximate the intersection of two sets, say A and B, stored in two machines M_A and M_B. We wish to compute $A \cap B$ distributively, by exchanging a small number of bits. Typical applications of the problem are data replication check and distributed search engines. The problem can be efficiently solved by using Bloom filters $BF(A)$ and $BF(B)$ stored in M_A and

M_B, respectively. The algorithmic idea to compute $A \cap B$ is as follows:

1. M_A sends $BF(A)$ to M_B, using $r_{opt} = (m_A ln2)/|A|$ hash functions and a bit-array $m_A = \Theta(|A|)$;
2. M_B checks the existence of elements B into A by deploying $BF(A)$ and sends back explicitly the set of found elements, say Q. Note that, in general, $Q \supseteq A \cap B$ because of false positives;
3. M_A computes $Q \cap A$, and returns it.

Since Q contains $|A \cap B|$ keys plus the number of false positives (elements belonging only to A), we can conclude that $|Q| = |A \cap B| + |B|\epsilon$ where $\epsilon = 0.6185^{m_A/|A|}$ is the error rate for that design of $BF(A)$. Since we need $\log |U|$ bits to represent each key, the total number of exchanged bits is $\Theta(|A|) + (|A \cap B| + |B| 0.6185^{m_A/|A|}) \log |U|$ which is much smaller than $|A| \log |U|$ the number of bits to be exchanged by using a plain algorithm that sends the whole A's set to M_B.

References

[1] Martin Aumüller, Martin Dietzfelbinger, and Philipp Woelfel. Explicit and Efficient Hash Families Suffice for Cuckoo Hashing with a Stash. In Proceedings of European Symposium on Algorithms (ESA), Lecture Notes in Computer Science 7501, Springer, 108–120, 2012.

[2] Andrei Z. Broder and Michael Mitzenmacher. Survey: Network Applications of Bloom Filters: A Survey. *Internet Mathematics*, 1(4): 485-509, 2003.

[3] Michael Mitzenmacher. Compressed bloom filters. *IEEE/ACM Transactions on Networks*, 10(5): 604-612, 2002.

[4] Ian H. Witten, Alistair Moffat and Timothy C. Bell. Managing Gigabytes: Compressing and Indexing Documents and Images, Second Edition. Morgan Kaufmann, 1999.

8 Set Intersection

"Sharing is caring!"

This lecture attacks a simple problem over sets, it constitutes the backbone of every query resolver in a (Web) search engine. A search engine is a well-known tool designed to search for information in a collection of documents $\mathcal{D}$. In the present chapter we restrict our attention to search engines for *textual* documents, meaning with this the fact that a document $d_i \in \mathcal{D}$ is a book, a news, a tweet or any file containing a sequence of linguistic tokens (aka, *words*). Among many other auxiliary data structures, a search engine builds an *index* to answer efficiently the queries posed by users. The user query Q is commonly structured as a *bag of words*, say $w_1 w_2 \cdots w_k$, and the goal of the search engine is to retrieve the *most relevant* documents in $\mathcal{D}$ which contain all query words. The people skilled in this art know that this is a very simplistic definition, because modern search engines search for documents that contain possibly *most* of the words in Q, the verb *contain* may be fuzzy interpreted as *contain synonyms or related words*, and the notion of *relevance* is pretty subjective and time-varying so that it cannot be defined precisely. In any case, this is not a chapter of an Information Retrieval book, so we refer the interested reader to the Information Retrieval literature, such as [4, 7]. Here we content ourselves to attack the most generic algorithmic step specified above.

Problem. *Given a sequence of words $Q = w_1 w_2 \cdots w_k$ and a document collection $\mathcal{D}$, find the documents in $\mathcal{D}$ that contain all words w_i.*

An obvious solution is to scan each document in $\mathcal{D}$ searching for all words specified by Q. This is simple but it would take time proportional to the whole length of the document collection, which is clearly too much even for a supercomputer or a data-center given the Web size! And, in fact, modern search engines build a very simple, but efficient, data structure called *inverted index* that helps in speeding up the flow of bi/million of daily user queries.

The inverted index consists of three main parts: the dictionary of words w, one list of occurrences per dictionary word (called *posting list*, below indicated with $\mathcal{L}[w]$), plus some additional information indicating the importance of each of these occurrences (to be deployed in the subsequent phases where the relevance of a document has to be established). The term "inverted" refers to the fact that word occurrences are not sorted according to their position in the document, but according to the alphabetic ordering of the words to which they refer. So inverted indexes remind the classic *glossary* present at the end of books, here extended to represent occurrences of *all* the words present into a collection of documents (and so, not just the most important words of them).

Each posting list $\mathcal{L}[w]$ is stored contiguously in a single array, eventually on disk. The names of the indexed documents (actually, their identifying URLs) are placed in another table and are succinctly identified by integers, called *docID*s, which we may assume to have been assigned arbitrarily by the search engine.[1] Also the dictionary is stored in a table which contains some satellite information plus the pointers to the posting lists. Figure 8.1 illustrates the main structure of an inverted index.

Dictionary	Posting list
...	...
abaco	50, 23, 10
abiura	131, 100, 90, 132
ball	20, 21, 90
mathematics	15, 1, 3, 23, 30, 7, 10, 18, 40, 70
zoo	5, 1000
...	...

FIGURE 8.1: An example of inverted (unsorted) index for a part of a dictionary.

Coming back to the problem stated above, let us assume that the query Q consists of two words `abaco mathematics`. Finding the documents in $\mathcal{D}$ that contain both two words of Q boils down to finding the docIDs shared by the two inverted lists pointed to by `abaco` and `mathematics`: namely, 10 and 23. It is easy to conclude that this means to solve a *set intersection* problem between the two sets represented by $\mathcal{L}$[`abaco`] and $\mathcal{L}$[`mathematics`], which is the key subject of this chapter.

Given that the integers of two posting lists are arbitrarily arranged, the computation of the intersection might be executed by comparing each docID $a \in \mathcal{L}$[`abaco`] with all docIDs $b \in \mathcal{L}$[`mathematics`]. If $a = b$ then a is inserted in the result set. If the two lists have length n and m, this brute-force algorithm takes $n \times m$ steps/comparisons. In the real case that n and m are of the order of millions, as it typically occurs for common words in the modern Web, then that number of steps/comparisons is of the order of $10^6 \times 10^6 = 10^{12}$. Even assuming that a PC is able to execute one billion comparisons per second (10^9 cmp/sec), this trivial algorithm takes 10^3 seconds to process a bi-word query (so about ten minutes), which is too much even for a patient user!

The bad news is that the docIDs occurring in the two posting lists cannot be arranged *arbitrarily*, but we must impose some proper structure over them in order to speed up the identification of the common integers. The key idea here is to *sort* the posting lists as shown in Figure 8.2.
It is therefore preferable, from a computational point of view, to reformulate the intersection problem onto two *sorted* sets $A = \mathcal{L}$[`abaco`] and $B = \mathcal{L}$[`mathematics`], as follows:

(Sorted) Set Intersection Problem. *Given two sorted integer sequences $A = a_1 a_2 \cdots a_n$ and $B = b_1 b_2 \cdots b_m$, such that $a_i < a_{i+1}$ and $b_i < b_{i+1}$, compute the integers common to both sets.*

The *sortedness* of the two sequences allows to design an intersection algorithm that is deceptively simple, elegant and fast. It consists of scanning A and B from left to right by comparing at each step a pair of docIDs from the two lists. Say a_i and b_j are the two docIDs currently compared, initially $i = j = 1$. If $a_i < b_j$ the iterator i is incremented, if $a_i > b_j$ the iterator j is incremented, otherwise $a_i = b_j$ and thus a common docID is found and both iterators are incremented. At each

[1]To be precise, the docID assignment process is a crucial one to save space in the storage of those posting lists, but its solution is too much sophisticated to be discussed here and thus it is deferred to the scientific literature [6].

Dictionary	Posting list
...	...
abaco	10, 23, 50
abiura	90, 100, 131, 132
ball	20, 21, 90
mathematics	1, 3, 7, 10, 15, 18, 23, 30, 40, 70
zoo	5, 1000
...	...

FIGURE 8.2: An example of inverted (sorted) index for a part of a dictionary.

step the algorithm executes one comparison and advances at least one iterator. Given that $n = |A|$ and $m = |B|$ are the number of elements in the two sequences, we can deduct that i (resp. j) can advance at most n times (resp. m times), so we can conclude that this algorithm requires no more than $n + m$ comparisons/steps; we write *no more* because it could be the case that one sequence is exhausted much before the other one, so that many elements of the latter may be not compared. This time cost is significantly smaller than the one mentioned above for the unsorted sequences (namely $n \times m$), and its real advantage in practice is strikingly evident. In fact, by considering our running example with n and m of the order of 10^6 docIDs and a PC performing 10^9 comparisons per second, we derive that this new algorithm takes 10^{-3} seconds to compute $A \cap B$, which is in the order of milliseconds, exactly what occurs in modern search engines.

An attentive reader may have noticed this algorithm mimics the merge-procedure used in Merge-Sort, here adapted to fing the common elements of the two sets A and B rather than merging them.

FACT 8.1 *The intersection algorithm based on the merge-based paradigm solves the sorted set intersection problem in $O(m + n)$ time.*

In the case that $n = \Theta(m)$ this algorithm is optimal, and thus it cannot be improved; moreover it is based on the scan-based paradigm that it is optimal also in the disk model because it takes $O(n/B)$ I/Os. To be more precise, the scan-based paradigm is optimal whichever is the memory hierarchy underlying the computation (the so called *cache-oblivious model*). The next question is what we can do whenever m is much different of n, say $m \ll n$. This is the situation in which one word is much more selective than the other one; here, the classic *binary search* can be helpful, in the sense that we can design an algorithm that binary searches every element $b \in B$ (they are few) into the (many) sorted elements of A thus taking $O(m \log n)$ steps/comparisons. This time complexity is better than $O(n + m)$ if $m = o(n/\log n)$ which is actually less stringent that the condition $m \ll n$ we imposed above.

FACT 8.2 *The intersection algorithm based on the binary-search paradigm solves the sorted set intersection problem in $O(m \log n)$ time.*

The next question is whether an algorithm can be designed that combines the best of both merge-based and search-based approaches. In fact, there is an inefficacy in the binary-search approach which becomes apparent when m is of the order of n. When we search item b_i in A we possibly re-check over and over the same elements of A. Surely this is the case for its middle element, say $a_{n/2}$, which is the first one checked by any binary search. But if $b_i > a_{n/2}$ then it is useless to compare b_{i+1} with $a_{n/2}$ because for sure it is larger, since $b_{i+1} \geq b_i > a_{n/2}$. And the same holds for all subsequent elements of B. A similar argument applies possibly to other elements in A checked by the binary search; so the next challenge we address is how to avoid this *useless comparisons*.

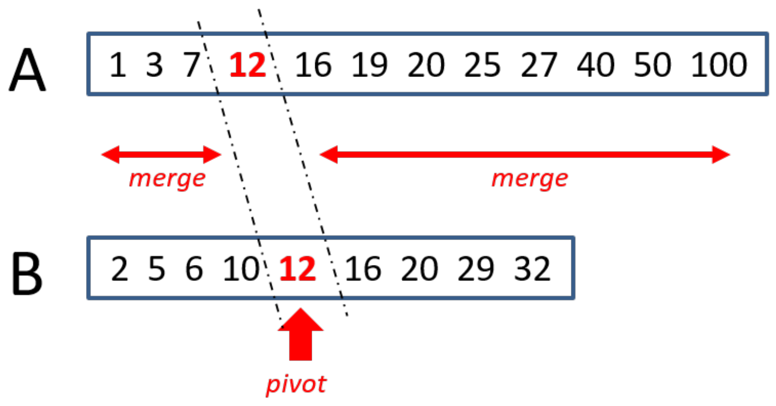


FIGURE 8.3: An example of the Intersection paradigm based on Mutual Partitioning: the pivot is 12, the median element of B.

This is achieved by adopting another classic algorithmic paradigm, called *partitioning*, which is the one we used to design the Quicksort, and here applied to split repeatedly and mutually two sequences. Formally, let us assume that $m \leq n$ and be both even numbers, we pick the *median* element $b_{m/2}$ of the shortest sequence B as a *pivot* and search for it into the longer sequence A. Two cases may occur: (i) $b_{m/2} \in A$, say $b_{m/2} = a_j$ for some j, and thus $b_{m/2}$ is returned as one of the elements of the intersection $A \cap B$; or (ii) $b_{m/2} \notin A$, say $a_j < b_{m/2} < a_{j+1}$ (where we assume that $a_0 = -\infty$ and $a_{n+1} = +\infty$). In both cases the intersection algorithm proceeds *recursively* in the two parts in which each sequence A and B has been split by the choice of the pivot, thus computing recursively $A[1, j] \cap B[1, m/2 - 1]$ and $A[j + 1, n] \cap B[m/2 + 1, n]$. A small optimization consists of discarding from the first recursive call the element $b_{m/2} = a_j$ (in case (i)). The pseudo-code is given in Figure 8.1, and a running example is illustrated in Figure 8.3. There the median element of B used as the pivot for the mutual partitioning of the two sequences is 12, and it splits A into two unbalanced parts (i.e. $A[1, 4]$ and $A[5, 12]$) and B into two almost-halves (i.e. $B[1, 5]$ and $B[6, 9]$) which are recursively intersected; since the pivot occurs both in A and B it is returned as an element of the intersection. Moreover we notice that the first part of A is shorter than the first part of B and thus in the recursive call their role will be exchanged.

In order to evaluate the time complexity we need to identify the worst case. Let us begin with the simplest situation in which the pivot falls outside A (i.e. $j = 0$ or $j = n$). This means that one of the two parts in A is empty and thus the corresponding halve of B can be discarded from the subsequent recursive calls. So one binary search over A, costing $O(\log n)$, has discarded an half of B. If this occurs at any recursive call, the total number of calls will be $O(\log m)$ thus inducing an overall cost for the algorithm equal to $O(\log m \; \log n)$. That is, an *unbalanced* partitioning of A induces indeed a very good behavior of the intersection algorithm; this is something opposite to what stated typically about recursive algorithms. On the other hand, let us assume that the pivot $b_{m/2}$ falls inside the sequence A and consider the case that it coincides with the median element of A, say $a_{n/2}$. In this specific situation the two partitions are balanced in both sequences we are intersecting, so the time complexity can be expressed via the following recurrent relation $T(n, m) = O(\log n) + 2T(n/2, m/2)$,

Algorithm 8.1 Intersection based on Mutual Partitioning

```
1: Let $m = |B| \leq n = |A|$, otherwise exchange the role of $A$ and $B$;
2: Pick the median element $p = b_{\lfloor m/2 \rfloor}$ of $B$;
3: Binary search for the position of $p$ in $A$, say $a_j \leq p < a_{j+1}$;
4: if $p = a_j$ then
5:     print $p$;
6: end if
7: Compute recursively the intersection $A[1, j] \cap B[1, m/2]$;
8: Compute recursively the intersection $A[j+1, n] \cap B[m/2+1, n]$.
```

with the base case of $T(n, m) = O(1)$ whenever $n, m \leq 1$. It can be proved that this recurrent relation has solution $T(n, m) = O(m(1 + \log \frac{n}{m}))$ for any $m \leq n$. It is interesting to observe that this time complexity subsumes the ones of the previous two algorithms (namely the one based on merging and the one based on binary searching). In fact, when $m = \Theta(n)$ it is $T(n, m) = O(n)$ (á la merging); when $m \ll n$ it is $T(n, m) = O(m \log n)$ (á la binary searching). As we will see in Chapter **??**, about Statistical compression, the term $m \log \frac{n}{m}$ reminds an entropy cost of encoding m items within n items and thus induces to think about something that cannot be improved (for details see [1]).

FACT 8.3 *The intersection algorithm based on the mutual-partitioning paradigm solves the sorted set intersection problem in $O(m(1 + \log \frac{n}{m}))$ time.*

We point out that the bound $m \log \frac{n}{m}$ is optimal in the comparison model because it follows from the classic binary decision-tree argument. In fact, they do exist at least $\binom{n}{m}$ solutions to the set intersection problem (here we account only for the case in which $B \subseteq A$), and thus every comparison-based algorithm computing anyone of them must execute $\Omega(\log \binom{n}{m})$ steps, which is $\Omega(m \log \frac{n}{m})$ by definition of binomial coefficient.

Algorithm 8.2 Intersection based on Doubling Search

```
1: Let $m = |B| \leq n = |A|$, otherwise exchange the role of $A$ and $B$;
2: $i = 1$;
3: for $j = 1, 2, \ldots, m$ do
4:     $k = 0$;
5:     while $(i + 2^k \leq n)$ and $(B[j] > A[i + 2^k])$ do
6:         $k = k + 1$;
7:     end while
8:     $i'$ = Binary search $B[j]$ into $A[i+1, \min\{i + 2^k, n\}]$;
9:     if $(a_{i'} = b_j)$ then
10:        print $b_j$;
11:    end if
12:    $i = i'$.
13: end for
```

Although this time complexity is appealing, the previous algorithm is heavily based on recursive calls and binary searching which are two paradigms that offer poor performance in a disk-based setting when sequences are long and thus the number of recursive calls can be large (i.e. many dynamic memory allocations) and large is the number of binary-search steps (i.e. random memory accesses). In order to partially compensate with these issues we introduce another approach to ordered set intersection which allows us to discuss another interesting algorithmic paradigm: the so called *doubling search* or *galloping search* or also *exponential search*. It is a mix of merging and binary searching, which is clearer to discuss by means of an inductive argument. Let us assume that we have already checked the first $j-1$ elements of B for their appearance in A, and assume that $a_i \leq b_{j-1} < a_{i+1}$. To check for the next element of B, namely b_j, it suffices to search it in $A[i+1, n]$. However, and this is the bright idea of this approach, instead of binary searching this sub-array, we execute a *galloping search* which consists of checking elements of $A[i+1, n]$ at distances which grow as a power of two. This means that we compare b_j against $A[i+2^k]$ for $k = 0, 1, \ldots$ until we find that either $b_j < A[i+2^k]$, for some k, or it is $i + 2^k > n$ and thus we jumped out of the array A. Finally we perform a binary search for b_j in $A[i+1, \min\{i+2^k, n\}]$, and we return b_j if the search is successful. In any case, we determine the position of b_j in that subarray, say $a_{i'} \leq b_j < a_{i'+1}$, so that the process can be repeated by discarding $A[1, i']$ from the subsequent search for the next element of B, i.e. b_{j+1}. Figure 8.4 shows a running example, whereas Figure 8.2 shows the pseudo-code of the doubling search algorithm.

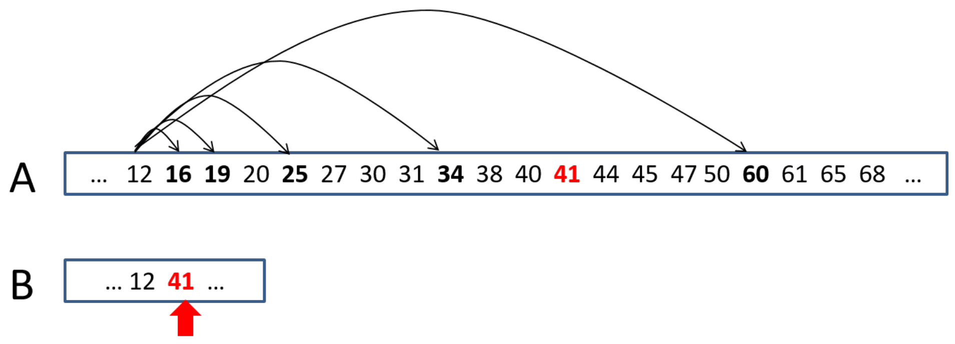


FIGURE 8.4: An example of the Doubling Search paradigm: the two sequences A and B are assumed to have been intersected up to the element 12. The next element in B, i.e. 41, is taken to be exponentially searched in the *suffix* of A following 12. This search checks A's elements at distances which are a power of two— namely $1, 2, 4, 8, 16$— until it finds the element 60 which is larger than 41 and thus delimits the portion of A within which the binary search for 41 can be confined. We notice that the searched sub-array has size 16, whereas the distance of 41 from 12 in A is 11 thus showing, on this example, that the binary search is executed on a sub-array whose size is smaller than twice the real distance of the searched element.

As far as the time complexity is concerned, we observe that the parameter k satisfies the property that $A[i+2^{k-1}] < b_j \leq A[i+2^k]$. So the position $i' - i$ of b_j in $A[i+1, \min\{i+2^k, n\}]$ is not much smaller than the size of this sub-array, because it is $2^{k-1} < i' - i \leq 2^k$ and so $2^k < 2(i' - i)$. Let us therefore denote with Δ_j the size of the sub-array where the binary search of b_j is executed, and let us denote with $i_j = i'$ as the position where b_j occurs in A. For the sake of presentation we set $i_0 = 0$. Clearly $i_j - i_{j-1} \leq \Delta_j \leq 2^k$ and thus, from before, we have $\Delta_j \leq 2(i_j - i_{j-1})$. These sub-arrays may be overlapping but by not much, as indeed we have $\sum_{j=1}^{m} \Delta_j \leq \sum_{j=1}^{m} 2(i_j - i_{j-1}) = 2n$ because this is a telescopic sum in which consecutive terms in the summation cancel out. For every j, the algorithm in Figure 8.2 executes $O(\log \Delta_j)$ steps because of the while-statement and because of the binary search. Summing for $j = 1, 2, \ldots, m$ we get a total time complexity of $O(\sum_{j=1}^{m} \log \Delta_j) = O(m \log \sum_{j=1}^{m} \frac{\Delta_j}{m}) = O(m \log \frac{n}{m})$.

FACT 8.4 *The intersection algorithm based on the doubling-search paradigm solves the sorted set intersection problem in $O(m(1 + \log \frac{n}{m}))$ time. This is the same time complexity of the intersection algorithm based on the mutual-partitioning paradigm but without incurring in the costs due to the recursive partitioning of the two sequences A and B. The time complexity is optimal in the comparison model.*

Although the previous approach avoids some of the pitfalls due to the recursive partitioning of the two sequences A and B, it still needs to jump over the array A because of the doubling scheme; and we know that this is inefficient when executed in a hierarchical memory. In order to avoid this issue, programmers adopt a *two-level organization of the data*, which is a very frequent scheme of efficient data structures for disk. The main idea of this storage scheme is to *logically* partition the sequence A into blocks A_i of size L each, and copy the first element of each block (i.e. $A_i[1] = A[iL + 1]$) into an auxiliary array A' of size $O(n/L)$. For the simplicity of exposition, let us assume that $n = hL$ so that the blocks A_i are h in number. The intersection algorithm then proceeds in two main phases. Phase 1 consists of merging the two sorted sequences A' and B, thus taking $O(n/L + m)$ time. As a result, the elements of B are interspersed among the element of A'. Let us denote by B_i the elements of B which fall between $A_i[1]$ and $A_{i+1}[1]$ and thus may occur in the block A_i. Phase 2 then consists of executing the merge-based paradigm of Fact 8.1 over all pairs of sorted sequences A_i and B_i which are non empty. Clearly, these pairs are no more than m. The cost of one of these merges is $O(|A_i| + |B_i|) = O(L + |B_i|)$ and they are at most m because this is the number of unempty blocks B_i. Moreover $B = \cup_i B_i$, consequently this intersection algorithm takes a total of $O(\frac{n}{L} + mL)$ time. For further details on this approach and its performance in practice the reader can look at [5].

FACT 8.5 *The intersection algorithm based on the two-level storage paradigm solves the sorted set intersection problem in $O(\frac{n}{L} + mL)$ time and $O(\frac{n}{LB} + \frac{mL}{B} + m)$ I/Os, because every merge of two sorted sequences A_i and B_i takes at least 1 I/O and they are no more than m.*

The two-level storage paradigm is suitable to adopt a compressed storage for the docIDs in order to save space. Let $a'_1, a'_2, \ldots, a'_L$ be the L docIDs stored ordered in some block A_i. These integers can be squeezed by adopting the so called Δ-scheme which consists of setting $a'_0 = 0$ and then representing a'_j as its difference with the preceding docID a'_{j-1} for $j = 1, 2, \ldots, L$. Then each of these differences can be stored somewhat compressed by using $\lceil \log_2 \max_i \{a'_i - a'_{i-1}\} \rceil$ bits, instead of the full-representation of four bytes!

This is a clear advantage whenever the differences are much smaller than the universe size from which the docIDs are taken. But this is not necessarily the case in practice, because the partitioning of the sequences is done according to L and not according to values of the docIDs. In general, given a sequence of n docIDs spread in a universe of $U = \{1, 2, \ldots, u\}$, the distribution which guarantees the smallest-maximum gap is the uniform one: for which it is $\max_i \{a'_i - a'_{i-1}\} \leq \frac{u}{n}$. In order to force this situation we preliminary shuffle the docIDs via a random permutation $\pi : U \longrightarrow U$, and assume that the possible sets over which the intersection problem can be invoked have been given in advance, and they are not larger than N. This is not a restriction in the Search Engine scenario because the dictionary and its posting lists are fixed in advance. Said this, we propose a solution which combines speed (as in the two-level scheme) and compressed space (unlike other solutions) by distinguishing between a preprocessing phase and a query phase.

In the preprocessing phase, we logically spit the universe U into M/L buckets of size uL/M each, denoted by U_i, where M is set as the size of the longest list. Then, we permute A according to the random permutation π and assign its permuted elements to the buckets U_i: namely, for each $x \in A$ we compute $\pi(x)$, take its $\ell = \lceil \log_2 \frac{uL}{M} \rceil$ most significant bits and denote by $\pi_\ell(x)$ their value. We then assign x to the bucket $U_{\pi_\ell(x)}$. We denote by A_i the sub-list of A's elements that have been mapped to U_i and are sorted according to their π-values. To implement the following query phase we need to have available π^{-1} so that we can retrieve the original element from its π-image.

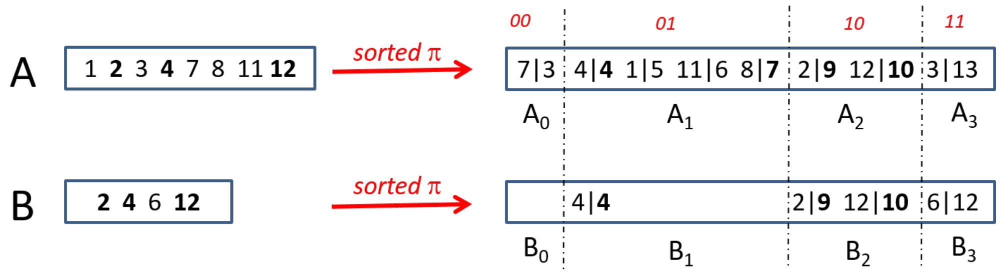


FIGURE 8.5: An example of the Random Permuting and Splitting paradigm. We assume universe $U = \{1, \ldots, 13\}$, set $L = 2$ and $M = 8$, and consider the permutation $\pi(x) = 1 + (4x \mod 13)$. So U is partitioned in $M/L = 4$ buckets identified by the MSB $\ell = \lceil \log_2 UL/M \rceil = \lceil \log_2 13 * 2/8 \rceil = 2$ bits of the π-image of each element. Recall that every π-image is represented in $\log_2 u = 4$ bits, so that $\pi(1) = 5 = (0101)_2$ and its 2 MSB are 01. The figure shows in bold the elements of $A \cap B$, moreover it depicts for the sake of exposition each docID as the pair $x|\pi(x)$ and, on top of every sublist, shows the 2 MSBs. In the example only three buckets of B are unempty, so we intersect only them with the corresponding ones of A, so that we drop the sublist A_0 without scanning it. The result is $\{4|4, 2|9, 12|10\}$, that gives $A \cap B$ by dropping the second π-component: namely, $\{2, 4, 12\}$.

In the query phase, let us assume that we wish to compute the intersection of two sets A and B which have been preprocessed above, and that $n = |A| > m = |B|$. The intuition is that, since the permutation π is the same for all sets, if element $z \in A \cap B$ then $\pi(z)$ will be routed to some sublists A_j and B_j with $j = \pi_\ell(z)$ according to its ℓ most significant bits. Therefore intersecting $A_j \cap B_j$ will correctly detect $\pi(z)$; moreover, since we have available π^{-1}, we can recover the original shared item z after having matched $\pi(z)$. Given these premises the intersection algorithm can be easily designed: for each B_j, we compute $B_j \cap A_j$ via the merge-based approach (Fact 8.1) and return the π^{-1}-image of the intersected elements. The average time complexity is therefore $O(m + \min\{n, mL\})$ because the number of unempty sublists B_j is at most m and each pair of intersected sets contains $O(L)$ elements on average. In fact the random permutation π maps A's elements in U's buckets of size $uL/M \le uL/n$ so that, on average, each of those buckets contains $|A| * (L/M) \le |A| * (L/n) = L$. This time complexity improves the one of the two-level storage scheme (Fact 8.5) whenever $m < \frac{n}{L}$, as it typically occurs in practice. A running example is shown in Figure 8.6.

As far as the space occupancy is concerned we notice that there are $\Theta(M/L)$ buckets, and the largest difference between two bucket entries can be bounded in two ways: the bucket width $O(uL/M)$ and the largest difference between any two consecutive list entries (after π-mapping). The latter quantity is well known from balls-and-bins problem: here having n balls and u bins. It can be shown that the largest difference is $O(\frac{u}{n} \log n)$ with high probability. The two bounds can be combined to a bound of $O(\log_2 \min\{\frac{uL}{M}, \frac{u}{n} \log n\}) = \log_2 \frac{u}{n} + \log_2 \min\{L, \log n\} + O(1)$ bits per list element (after π-mapping). The first term is unavoidable since it already shows up in the information theoretic lower bound. To be precise, an additional $O(n \frac{\log n}{L})$ bits have to be considered for A

in order to account for the cost of the $O(\log n)$-bits pointer to (the beginning of) each sublist of A. This term can be made smaller and smaller by increasing (not much!) the value of L.

FACT 8.6 *The intersection algorithm based on the random-permuting and splitting paradigm solves the sorted set intersection problem in $O(m + \min\{n, mL\})$ time and $O(m + \min\{\frac{n}{B}, \frac{mL}{B}\})$ I/Os, because every merge of two sublists A'_j and B'_j takes at least 1 I/O and they are no more than m. The space cost for storing a list of length n is $n(\log_2 \frac{u}{n} + \log_2 \min\{L, \log n\} + O(1 + \frac{\log n}{L}))$ bits with high probability.*

By analyzing the algorithmic structure of this last solution we notice few further advantages. First, we do not need to sort the original sequences, because the sorting is required only within the individual sublists which have average length L; this is much shorter than the lists' length so that we can use an internal-memory sorting algorithm over each π-permuted sublist. A second advantage is that we can avoid the checking of some sublists during the intersection process, without looking at them; this allows to drop the term $\frac{n}{L}$ occurring in Fact 8.5. Third, the choice of L can be done according to the hierarchical memory in which the algorithm is run; this means that if sublists are stored on disk, then $L = \Theta(B)$ can be the right choice.

The authors of [5, 3, 2] discuss some variants and improvements over all previous algorithms, some really sophisticate, we refer the interested reader to this literature. Here we report a picture taken from [5] that compares various algorithms with the following legenda: *zipper* is the merge-based algorithm (Fact 8.1), *skipper* is the two-level algorithm (Fact 8.5, with $L = 32$), *Baeza-Yates* is the mutual-intersection algorithm (Fact 8.3, 32 denotes the bucket size for which recursion is stopped), *lookup* is our last proposal (Fact 8.6, $L = 8$).

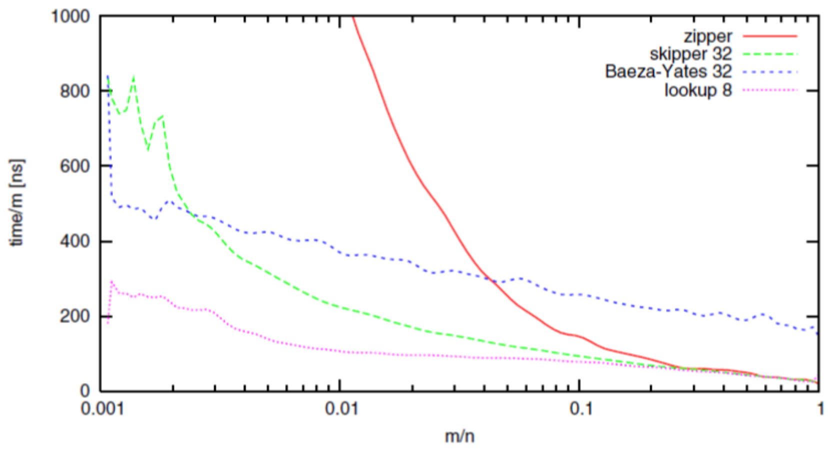


FIGURE 8.6: An experimental comparison among four sorted-set intersection algorithms.

We notice that *lookup* is the best algorithm up to a length ratio close to one. For lists of similar length all algorithms are very good. Still, it could be a good idea to implement a version of *lookup* optimized for lists of similar length. It is also interesting to notice that *skipper* improves *Baeza-Yates* for all but very small length ratios. For compressed lists and very different list lengths, we can claim that *lookup* is considerably faster over all other algorithms. Randomization allows interesting performance guarantees on both time and space performance. The experimented version of *skipper* uses a compressed first-level array; probably by dropping compression from the first-level would not increase much the space, but it would induce a significant speedup in time. The only clear looser is *Baeza-Yates*, for every list lengths there are other algorithms that improve it. It is pretty much clear that a good asymptotic complexity does not reflect onto a good time efficiency whenever recursion is involved.

References

[1] Ricardo Baeza-Yates. A fast set intersection algorithm for sorted sequences. In *Procs of Annual Symposium on Combinatorial Pattern Matching (CPM)*, Lecture Notes in Computer Science 3109, pp. 400-408, 2014.

[2] Jérémy Barbay, Alejandro López-Ortiz, Tyler Lu, Alejandro Salinger. An experimental investigation of set intersection algorithms for text searching. *ACM Journal of Experimental Algorithmics*, 14, 2009.

[3] Bolin Ding, Arnd Christian König. Fast set intersection in memory. *PVLDB*, 4(4): 255-266, 2011.

[4] Christopher D. Manning, Prabhakar Raghavan and Hinrich Schütze. *Introduction to Information Retrieval*. Cambridge University Press, 2008.

[5] Peter Sanders, Frederik Transier. Intersection in integer inverted indices. In *Procs of Workshop on Algorithm Engineering and Experiments (ALENEX)*, 2007.

[6] Hao Yan, Shuai Ding, Torsten Suel. Inverted index compression and query processing with optimized document ordering. In *Procs of WWW*, pp. 401-410, 2009.

[7] Ian H. Witten, Alistair Moffat, Timoty C. Bell. *Managing Gigabytes*. Morgan Kauffman, second edition, 1999.