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.

1

Searching Strings by Prefix

This problem is experiencing renewed interest in the algorithmic community because of new applications spurring from Web search-engines. Think to the *auto-completion* feature currently offered by mayor search engines Google, Bing and Yahoo, in their search bars: It is a prefix search executed on-the-fly over millions of strings, using the query pattern typed by the user as the string to search. The dictionary typically consists of the most recent and the most frequent queries issued by other users. This problem is made challenging by the size of the dictionary and by the time constraints imposed by the patience of the users. In this chapter we will describe many different solutions to this problem of increasing sophistication and efficiency both in time, space and I/O complexities.

The prefix-search problem. *Given a dictionary $\mathcal{D}$ consisting of n strings of total length N, drawn from an alphabet of size σ, the problem consists of preprocessing $\mathcal{D}$ in order to retrieve (or just count) the strings of $\mathcal{D}$ that have P as a prefix.*

We mention two other typical string queries which are the *exact* search and the *substring* search within the dictionary strings in $\mathcal{D}$. The former is best addressed via hashing because of its simplicity and practical speed; Chapter **??** will detail several hashing solutions. The latter problem is more sophisticated and finds application in computational genomics and asian search engines, just to cite a few. It consists of finding all *positions* where the query-pattern P occurs as a substring of the dictionary strings. Surprisingly enough, it does exist a simple *algorithmic reduction* from substring search to prefix search over the set of *all suffixes of the dictionary strings*. This reduction will be commented in Chapter 2 where the Suffix Array and the Suffix Tree data structures will be introduced. As a result, we can conclude that the prefix search is the backbone of other important search problems on strings, so the data structures introduced in this chapter offer applications which go far beyond the simple ones discussed below.

1.1 Array of string pointers

We start with a simple, common solution to the prefix-search problem which consists of an array of pointers to strings stored in arbitrary locations on disk. Let us call $A[1, n]$ the array of pointers,

which are *indirectly* sorted according to the strings pointed to by its entries. We assume that each pointer takes w bytes of memory, typically 4 bytes (32 bits) or 8 bytes (64 bits). Several other representations of pointers are possible, as e.g. variable-length representations, but this discussion is deferred to Chapter **??**, where we will deal with the efficient encoding of integers.

Figure 1.1 provides a running example in which the dictionary strings are stored in an array S, according to an arbitrary order.

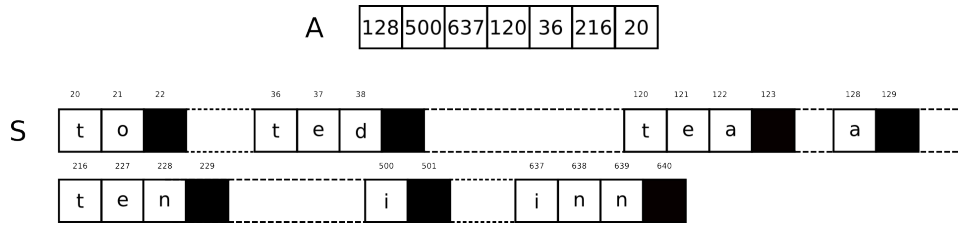


FIGURE 1.1: The array S of strings and the array A of (indirectly) sorted pointers to S's strings.

There are two crucial properties that the (sorted) array A satisfies:

- all dictionary strings prefixed by P occur contiguously if lexicographically sorted. So their pointers occupy a subarray, say $A[l, r]$, which may be possibly empty if P does not prefix any dictionary string.
- the string P is lexicographically located beyween $A[l-1]$ and $A[l]$.

Since the prefix search returns either the number of dictionary strings prefixed by P, hence the value $r - l + 1$, or visualizes these strings, the key problem is to identify the two extremes l and r efficiently. To this aim, we reduce the prefix search problem to the *lexicographic search* of a pattern Q in $\mathcal{D}$: namely, the search of the lexicographic position of Q among $\mathcal{D}$'s strings. The formation of Q is simple: Q is either the pattern P or the pattern $P\#$, where # is larger than any other alphabet character. It is not difficult to convince yourself that $Q = P$ will precede the string $A[l]$ (see above), whereas $Q = P\#$ will follow the string $A[r]$. This means actually that two lexicographic searches for patterns of length no more than $p + 1$ are enough to solve the prefix-search problem.

The lexicographic search can be implemented by means of an (indirect) binary search over the array A. It consists of $O(\log n)$ steps, each one requiring a string comparison between Q and the string pointed by the entry tested in A. The comparison is lexicographic and thus takes $O(p)$ time and $O(p/B)$ I/Os, because it may require in the worst case the scan of all $\Theta(p)$ characters of Q. The poor time and I/O-complexities derive from the *indirection*, which forces no locality in the memory/string accesses of the binary search. The inefficiency is even more evident if we wish to retrieve all n_occ strings prefixed by P, and not just count them. After that the range $A[l, r]$ has been identified, each string visualization elicits at least one I/O because contiguity in A does not imply contiguity of the pointed strings in S.

THEOREM 1.1 *The complexity of a prefix search over the array of string pointers is $O(p \log n)$ time and $O(\frac{p}{B} \log n)$ I/Os, the total space is $N + (1 + w)n$ bytes. Retrieving the n_{occ} strings prefixed by P needs $\Omega(n_{occ})$ I/Os.*

Proof Time and I/O complexities derive from the previous observations. For the space occu-

pancy, A needs n pointers, each taking a memory word w, and all dictionary strings occupy N bytes plus one-byte delimiter for each of them (commonly \0 in C). ■

The bound $\Omega(n_{occ})$ may be a major bottleneck if the number of returned strings is large, as it typically occurs in queries that use the prefix search as a preliminary step to select a *candidate set of answers* that have then to be refined via a proper post-filtering process. An example is the solution to the problem of *searching with wild-cards* which involves the presence in P of many special symbols *. The wild-card * matches any substring. In this case if $P = \alpha * \beta * \cdots$, where $\alpha, \beta, \ldots$ are un-empty strings, then we can implement the wild-card search by first performing a prefix-search for α in $\mathcal{D}$ and then checking brute-forcedly whether P matches the returned strings given the presence of the wild-cards. Of course this approach may be very expensive, especially when α is not a selective prefix and thus many dictionary strings are returned as candidate matches. Nevertheless this puts in evidence how much slow may be in a disk environment a wild-card query if solved with a trivial approach.

1.1.1 Contiguous allocation of strings

A simple trick to circumvent some of the previous limitations is to store the dictionary strings sorted lexicographically and contiguously on disk. This way (pointers) contiguity in A reflects into (string) contiguity in S. This has two main advantages:

- when the binary search is confined to few strings, they will be closely stored both in A and S, so probably they have been buffered by the system in internal memory (*speed*);
- some compression can be applied to contiguous strings in S, because they typically share some prefix (*space*).

Given that S is stored on disk, we can deploy the first observation by blocking strings into groups of B characters each and then *store* a pointer to the first string of each group in A. The sampled strings are denoted by $\mathcal{D}_B \subseteq \mathcal{D}$, and their number n_B is upper bounded by $\frac{N}{B}$ because we pick at most one string per block. Since A has been squeezed to index at most $n_B \leq n$ strings, the search over A must be changed in order to reflect the *two-level structure* given by the array A and the blocks of strings in S. So the idea is to decompose the lexicographic search for Q in a two-stages process: in the first stage, we search for the lexicographic position of Q within the sampled strings of $\mathcal{D}_B$; in the second stage, this position is deployed to identify the block of strings where the lexicographic position of Q lies in, and then the strings of this block are scanned and compared with Q for prefix match. We recall that, in order to implement the prefix search, we have to repeat the above process for the two strings P and $P\#$, so we have proved the following:

THEOREM 1.2 *Prefix search over $\mathcal{D}$ takes $O(\frac{p}{B} \log \frac{N}{B})$ I/Os. Retrieving the strings prefixed by P needs $\frac{N_{occ}}{B}$ I/Os, where N_{occ} is their length. The total space is $N + (1 + w)n_B$ bytes.*

Proof Once the block of strings $A[i, j]$ prefixed by P has been identified, we can report all of them in $O(\frac{N_{occ}}{B})$ I/Os; scanning the contiguous portion of S that contains those strings. The space occupancy comes from the observation that pointers are stored only for the n_B sampled strings. ■

Typically strings are shorter than B, so $\frac{N}{B} \leq n$, hence this solution is faster than the previous one, in addition it can be effectively combined with the technique called *Front-Coding compression* to further lowering the space and I/O-complexities.

1.1.2 Front Coding

Given a sequence of sorted strings is probable that adjacent strings share a common prefix. If ℓ is the number of shared characters, then we can substitute them with a proper variable-length binary encoding of ℓ thus saving some bits with respect to the classic fixed-size encoding based on 4- or 8-bytes. A following chapter will detail some of those encoders, here we introduce a simple one to satisfy the curiosity of the reader. The encoder pads the binary representation of ℓ with 0 until an integer number of bytes is used. The first two bits of the padding (if any, otherwise one more byte is added), are used to encode the number of bytes used for the encoding.[1] This encoding is prefix-free and thus guarantees unique decoding properties; its byte-alignment also ensures speed in current processors.

More efficient encoders are available, anyway this simple proposal ensures to replace the initial $\Theta(\ell \log_2 \sigma)$ bits, representing ℓ characters of the shared prefix, with $O(\log \ell)$ bits of the integer encoding, so resulting advantageous in space. Obviously its final impact depends on the amount of shared characters which, in the case of a dictionary of URLs, can be up to 70%.

Front coding is a *delta*-compression algorithm, which can be easily defined in an incremental way: given a sequence of strings $(s_1, \ldots, s_n)$, it encodes the string s_i using the couple $(\ell_i, \hat{s}_i)$, where ℓ_i is the length of the longest common prefix between s_i and its predecessor s_{i-1} (0 if $i = 1$) and $\hat{s}_i = s_i[\ell_i + 1, |s_i|]$ is the "remaining suffix" of the string s_i. As an example consider the dictionary $\mathcal{D} = \{$ `alcatraz, alcool, alcyone, anacleto, ananas, aster, astral, astronomy` $\}$; its front-coded representation is $(0, \texttt{alcatraz}), (3, \texttt{ool}), (3, \texttt{yone}), (1, \texttt{nacleto}), (3, \texttt{nas}), (1, \texttt{ster}), (3, \texttt{ral}), (4, \texttt{onomy})$.

Decoding a string a pair $(\ell, \hat{s})$ is symmetric, we have to copy ℓ characters from the previous string in the sequence and then append the remaining suffix $\hat{s}$. This takes $O(|s|)$ optimal time and $O(1 + \frac{|s|}{B})$ I/Os, provided that the preceding string is available. In general, the reconstruction of a string s_i may require to scan back the input sequence up to the first string s_1, which is available in its entirety. So we may possibly need to scan $(\hat{s}_1, \ell_1), \ldots, (\hat{s}_{i-1}, \ell_{i-1})$ and reconstruct $s_1, \ldots, s_{i-1}$ in order to decode $(\ell_i, \hat{s}_i)$. Therefore, the time cost to decode s_i might be much higher than the optimal $O(|s_i|)$ cost.[2]

To overcome this drawback, it is typical to apply front-coding to block of strings thus resorting the two-level scheme we introduced in the previous subsection. The idea is to restart the front-coding at the beginning of every block, so the first string of each block is stored *uncompressed*. This has two immediate advantages onto the prefix-search problem: (1) these uncompressed strings are the ones participating in the binary-search process and thus they do not need to be decompressed when compared with Q; (2) each block is compressed individually and thus the scan of its strings for searching Q can be combined with the decompression of these strings without incurring in any slowdown. We call this storage scheme "Front-coding with bucketing", and shortly denote it by FC_B. Figure 1.2 provides a running example in which the strings "alcatraz", "alcyone", "ananas", and "astral" are stored explicitly because they are the first of each block.

As a positive side-effect, this approach reduces the number of sampled strings because it can potentially increase the number of strings stuffed in one disk page: we start from s_1 and we front-compress the strings of $\mathcal{D}$ in order; whenever the compression of a string s_i overflows the current block, it starts a new block where it is stored *uncompressed*. The number of sampled strings lowers from about $\frac{N}{B}$ to about $\frac{FC_B(\mathcal{D})}{B}$ strings, where $FC_B(\mathcal{D})$ is the space required by FC_B to store all the dictionary strings. This impacts positively onto the number of I/Os needed for a prefix search in a obvious manner, given that we execute a binary search over the sampled strings. However space

[1] We are assuming that ℓ can be binary encoded in 30 bits, namely $\ell < 2^{30}$.

[2] A smarter solution would be to reconstruct only the first ℓ characters of the previous strings $s_1, s_2, \ldots, s_{i-1}$ because these are the ones interesting for s_i's reconstruction.

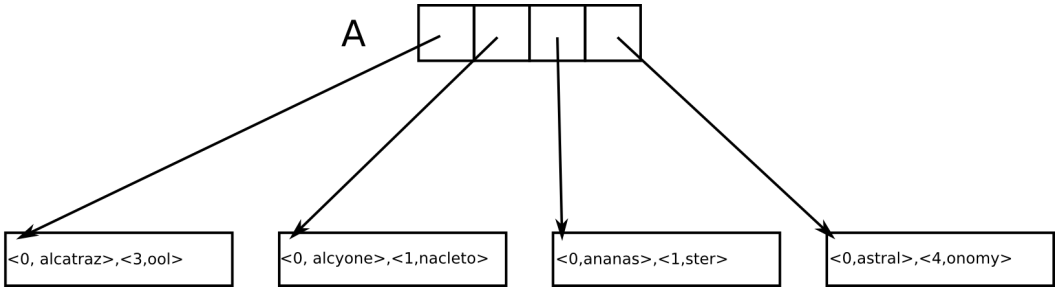


FIGURE 1.2: Two-level indexing of the set of strings $\mathcal{D}$ = { `alcatraz, alcool, alcyone, anacleto, ananas, aster, astral, astronomy`} are compressed with FC_B, where we assumed that each page is able to store two strings.

occupancy increases with respect to $FC(\mathcal{D})$ because $FC_B(\mathcal{D})$ forces the first string of each block to be stored uncompressed; nonetheless, we expect that this increase is negligible because $B \gg 1$.

THEOREM 1.3 *Prefix search over $\mathcal{D}$ takes $O(\frac{p}{B} \log \frac{FC_B(\mathcal{D})}{B}$ I/Os. Retrieving the strings prefixed by P needs $O(\frac{FC_B(\mathcal{D}_{occ})}{B})$ I/Os, where $\mathcal{D}_{occ} \subseteq \mathcal{D}$ is the set of strings in the answer set.*

So, in general, compressing the strings is a good idea because it lowers the space required for storing the strings, and it lowers the number of I/Os. However we must observe that FC-compression might increase the time complexity of the scan of a block from $O(B)$ to $O(B^2)$ because of the decompression of that block. In fact, take the sequence of strings $(a, aa, aaa, \ldots)$ which is front coded as $(0, a), (1, a), (2, a), (3, a), \ldots$. In one disk page we can stuff $\Theta(B)$ such pairs, which represent $\Theta(B)$ strings whose total length is $\sum_{i=0}^{B} \Theta(i) = \Theta(B^2)$ characters. Despite these pathological cases, in practice the space reduction consists of a *constant factor* so the time increase incurred by a block scan is negligible.

Overall this approach introduces a time/space trade-off driven by the block size B. As far as time is concerned we can observe that the longer is B, the better is the compression ratio but the slower is a prefix search because of a longer scan-phase; conversely, the shorter is B, the faster is the scan-phase but the worse is the compression ratio because of a larger number of fully-copied strings. As far as space is concerned, the longer is B, the smaller is the number of copied strings and thus the smaller is the storage space in internal memory needed to index their pointers; conversely, the shorter is B, the larger is the number of pointers thus making probably impossible to fit them in internal memory.

In order to overcome this trade-off we decouple search and compression issues as follows. We notice that the proposed data structure consists of *two* levels: the "upper" level contains references to the *sampled strings* $\mathcal{D}_B$, the "lower" level contains the strings themselves stored in a block-wise fashion. The choice of the algorithms and data structures used in the two levels are "orthogonal" to each other, and thus can be decided independently. It goes without saying that this 2-level scheme for searching-and-storing a dictionary of strings is suitable to be used in a hierarchy of two memory levels, such as the cache and the internal memory. This is typical in Web search, where $\mathcal{D}$ is the dictionary of terms to be searched by users and disk-accesses have to be avoided in order to support each search over $\mathcal{D}$ in few millisecs.

In the next three sections we propose three improvements to the 2-level solution above, two of them regard the first level of the sampled strings, one concerns with the compressed storage of

all dictionary strings. Actually, these proposals have an interest in themselves and thus the reader should not confine their use to the one described in these notes.

1.2 Interpolation search

Until now, we have used binary search over the array A of string pointers. But if $\mathcal{D}_B$ satisfies some statistical properties, there are searching schemes which support faster searches, such as the well known *interpolation search*. In what follows we describe a variant of classic interpolation search which offers some interesting additional properties (details in [3]). For simplicity of presentation we describe the algorithm in terms of a dictionary of integers, knowing that if items are strings, we can still look at them as integers in base σ. So for the prefix-search problem we can pad logically all strings at their end, thus getting to the same length, by using a character that we assume to be smaller than any other alphabet character. Lexicographic order among strings is turned in classic ordering of integers, so that the search for P and $P\#$ can be turned into a search for two proper integers.

So without loss of generality, assume that $\mathcal{D}_B$ is an array of integers $X[1, m] = x_1 \dots x_m$ with $x_i < x_{i+1}$ and $m = n_B$. We evenly subdivide the range $[x_1, x_m]$ into m bins $B_1, \dots, B_m$ (so we consider as many bins as integers in X), each bin representing a contiguous range of integers having length $b = \frac{x_m - x_1 + 1}{m}$. Specifically $B_i = [x_1 + (i-1)b, x_1 + ib)$. In order to guarantee the constant-time access to these bins we need to keep an additional array, say $I[1, m]$, such that $I[i]$ points to the first and last item of B_i in X.

Figure 1.3 reports an example where $m = 12$, $x_1 = 1$ and $x_{12} = 36$ and thus the bin length is $b = 3$.

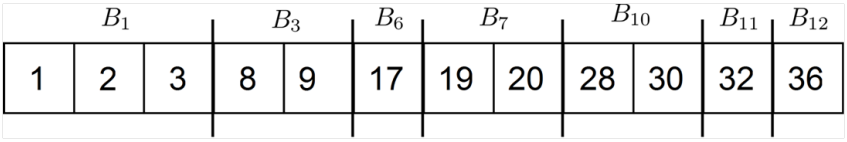


FIGURE 1.3: An example of use of interpolation search over an itemset of size 12. The bins are separated by bars; some bins, such as B_4 and B_8, are empty.

The algorithm searches for an integer y in two steps. In the first step it calculates j, the index of the candidate bin B_j where y could occur: $j = \lfloor \frac{y - x_1}{b} \rfloor + 1$. In the second step, it determines via $I[j]$ the sub-array of X which stores B_j and it does a binary search over it for y, thus taking $O(\log |B_i|) = O(\log b)$ time. The value of b depends on the magnitude of the integers present in the indexed dictionary. Surprisingly enough, we can get a better bound which takes into account the distribution of the integers of X in the range $[x_1, x_m]$.

THEOREM 1.4 *We can search for an integer in a dictionary of size m taking $O(\log \Delta)$ time in the worst case, where Δ is the ratio between the maximum and the minimum gap between two consecutive integers of the input dictionary. The extra space is $O(m)$.*

Proof Correctness is immediate. For the time complexity, we observe that the maximum of a series of integers is at least as large as their mean. Here we take as those integers the gaps $x_i - x_{i-1}$,

and write:

$$\max_{i=2\ldots m}(x_i - x_{i-1}) \geq \frac{\sum_{i=2}^{m} x_i - x_{i-1}}{m-1} \geq \frac{x_m - x_1 + 1}{m} = b \tag{1.1}$$

The last inequality comes from the following arithmetic property: $\frac{a'}{a''} \geq \frac{a'+1}{a''+1}$ whenever $a' \geq a''$, which can be easily proved by solving it.

Another useful observation concerns with the maximum number of integers that can belong to any bin. Since integers of X are spaced apart by $s = \min_{i=2,\ldots,m}(x_i - x_{i-1})$ units, every bin contains no more than b/s integers.

Recalling the definition of Δ, and the two previous observations, we can thus write:

$$|B_i| \leq \frac{b}{s} \leq \frac{\max_{i=2\ldots m}(x_i - x_{i-1})}{\min_{i=2,\ldots,m}(x_i - x_{i-1})} = \Delta$$

.

So the theorem follows due to the binary search performed within B_i. Space occupancy is optimal and equal to $O(m)$ because of the arrays $X[1,m]$ and $I[1,m]$. ∎

We note the following interesting properties of the proposed algorithm:

- The algorithm is oblivious to the value of Δ, although its complexity can be written in terms of this value.
- The worst-case search time is $O(\log m)$, when the whole X ends up in a single bin, and thus the precise bound should be $O(\log \min\{\Delta, m\})$. So it cannot be worst than the binary search.
- The space occupancy is $O(m)$ which is optimal asymptotically; however, it has to be notice that binary search is in-place, whereas interpolation search needs the extra array $I[1,m]$.
- The algorithm reproduces the $O(\log \log m)$ time performance of classic interpolation search on data drawn independently from the uniform distribution, as shown in the following lemma. We observe that, uniform distribution of the X's integers is uncommon in practice, nevertheless we can artificially enforce it by selecting a random permutation $\pi : U \longrightarrow U$ and shuffling X according to π before building the proposed data structure. Care must be taken at query time since we search not y but its permuted image $\pi(y)$ in $\pi(X)$. This way the query performance proved below holds with high probability whichever is the indexed set X. For the choice of π we refer the reader to [6].

LEMMA 1.1 If the m integers are drawn uniformly at random from $[1,U]$, the proposed algorithm takes $O(\lg \lg m)$ time with high probability.

Proof Say integers are uniformly distributed over $[0, U-1]$. As in bucket sort, every bucket B_i contains $O(1)$ integers on average. But we wish to obtain bounds with high probability. So let us assume to partition the integers in $r = \frac{m}{2 \log m}$ ranges. We have the probability $1/r$ that an integer belongs to a given range. The probability that a given range does not contain any integer is $(1 - \frac{1}{r})^m = (1 - \frac{2 \log m}{m})^m = O(e^{-2 \log m}) = O(1/m^2)$. So the probability that at least one range remains empty is smaller than $O(1/m)$; or, equivalently, with high probability every range contains at least one integer.

If this occurs with high probability, the maximum distance between two adjacent integers must be smaller than twice the range's length: namely $\max_i(x_i - x_{i-1}) \leq 2U/r = O(\frac{U \log m}{m})$.

Let us now take $r' = \Theta(m \log m)$ ranges, similarly as above we can prove that every adjacent pair of ranges contains at most one integer with high probability. Therefore if a range contains an integer, its two adjacent ranges (on the left and on the right) are empty with high probability. Thus we can lower bound the minimum gap with the length of one range: $\min_i(x_i - x_{i-1}) \geq U/r' = \Theta(\frac{U}{m \log m})$. Taking the ratio between the minimum and the maximum gap, we get the desired $\Delta = O(\log^2 m)$. ■

If this algorithm is applied to our string context, and strings are uniformly distributed, the number of I/Os required to prefix-search P in the dictionary $\mathcal{D}$ is $O(\frac{p}{B} \log \log \frac{N}{B})$. This is an exponential reduction in the search time performance according to the dictionary length.

1.3 Locality-preserving front coding

This is an elegant variant of front coding which provides a controlled trade-off between space occupancy and time to decode one string [2]. The key idea is simple, and thus easily implementable, but proving its guaranteed bounds is challenging. We can state the underlying algorithmic idea as follows: *a string is front-coded only if its decoding time is proportional to its length, otherwise it is written uncompressed*. The outcome in time complexity is clear: we compress only if decoding is optimal. But what appears surprising is that, even if we concentrated on the time-optimality of decoding, its "constant of proportionality" controls also the space occupancy of the compressed strings. It seems magic, indeed it is!

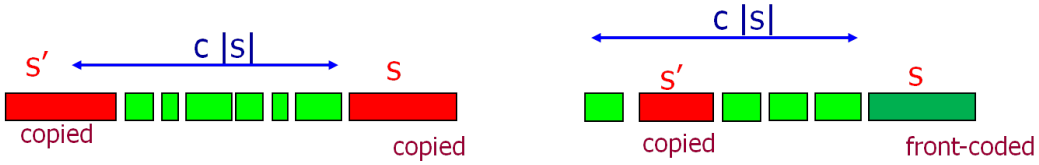


FIGURE 1.4: The two cases occurring in LPFC. Red rectangles are copied strings, green rectangles are front-coded strings.

Formally, suppose that we have front-coded the first $i-1$ strings $(s_1, \ldots, s_{i-1})$ into the compressed sequence $\mathcal{F} = (0, \hat{s}_1), (\ell_2, \hat{s}_2), \ldots, (\ell_{i-1}, \hat{s}_{i-1})$. We want to compress s_i so we scan backward at most $c|s_i|$ characters of $\mathcal{F}$ to check whether these characters are enough to reconstruct s_i. This actually means that an uncompressed string is included in those characters, because we have available the first character for s_i. If so, we front-compress s_i into $(\ell_i, \hat{s}_i)$; otherwise s_i is copied uncompressed in $\mathcal{F}$ outputting the pair $(0, s_i)$. The key difficulty here is to show that the strings which are left uncompressed, and were instead compressed by the classic front-coding scheme, have a length that can be controlled by means of the parameter c as the following theorem shows:

THEOREM 1.5 *Locality-preserving front coding takes at most* $(1+\epsilon)FC(\mathcal{D})$ *space, and supports the decoding of any dictionary string* s_i *in* $O(\frac{|s_i|}{\epsilon B})$ *optimal I/Os.*

Proof We call any uncompressed string s, a *copied* string, and denote the $c|s|$ characters explored during the backward check as the *left extent* of s. Notice that if s is a copied string, there can be

no copied string preceding s and beginning in its left extent (otherwise it would have been front-coded). Moreover, the copied string that precedes S may *end* within s's left extent. For the sake of presentation we call *FC-characters* the ones belonging to the output suffix $\hat{s}$ of a front-coded string s.

Clearly the space occupied by the front-coded strings is upper bounded by $FC(\mathcal{D})$. We wish to show that the space occupied by the copied strings, which were possibly compressed by the classic front-coding but are left uncompressed here, sums up to $\epsilon\ FC(\mathcal{D})$, where ϵ is a parameter depending on c and to be determined below.

We consider two cases for the copied strings depending on the amount of FC-characters that lie between two consecutive occurrences of them. The first case is called *uncrowded* and occurs when that number of FC-characters is at least $\frac{c|s|}{2}$; the second case is called *crowded*, and occurs when that number of FC-characters is at most $\frac{c|s|}{2}$. Figure 1.5 provides an example which clearly shows that if the copied string s is crowded then $|s'| \geq c|s|/2$. In fact, s' starts before the left extent of s but ends within the last $c|s|/2$ characters of that extent. Since the extent is $c|s|$ characters long, the above observation follows. If s is uncrowded, then it is preceded by at least $c|s|/2$ characters of front-coded strings (FC-characters).

FIGURE 1.5: The two cases occurring in LPFC. The green rectangles denote the front-coded strings, and thus their FC-characters, the red rectangles denote the two consecutive copied strings.

We are now ready to bound the total length of copied strings. We partition them into chains composed by one uncrowded copied-string followed by the maximal sequence of crowded copied-strings. In what follows we prove that the total number of characters in each chain is proportional to the length of its first copied-string, namely the uncrowded one. Precisely, consider the chain $w_1 w_2 \cdots w_x$ of consecutive copied strings, where w_1 is uncrowded and the following w_is are crowded. Take any crowded w_i. By the observation above, we have that $|w_{i-1}| \geq c|w_i|/2$ or, equivalently, $|w_i| \leq 2|w_{i-1}|/c = \cdots = (2/c)^{i-1}|w_1|$. So if $c > 2$ the crowded copied strings shrink by a constant factor. We have $\sum_i |w_i| = |w_1| + \sum_{i>1} |w_i| \leq |w_1| + \sum_{i>1}(2/c)^{i-1}|w_1| = |w_1| \sum_{i\geq 0}(2/c)^i < \frac{c|w_1|}{c-2}$.

Finally, since w_1 is uncrowded, it is preceded by at least $c|w_1|/2$ FC-characters (see above). The total number of these FC-characters is bounded by $FC(\mathcal{D})$, so we can upper bound the total length of the uncrowded strings by $(2/c)FC(\mathcal{D})$. By plugging this into the previous bound on the total length of the chains, we get $\frac{c}{c-2} \times \frac{2FC(\mathcal{D})}{c} = \frac{2}{c-2}FC(\mathcal{D})$. The theorem follows by setting $\epsilon = \frac{2}{c-2}$. ■

So locality-preserving front coding (shortly LPFC) is a compressed storage scheme for strings that can substitute their plain storage without introducing any asymptotic slowdown in the accesses to the compressed strings. In this sense it can be considered as a sort of *space booster* for any string indexing technique.

The two-level indexing data-structure described in the previous sections can benefit of LPFC as follows. We can use A to point to the copied strings of LPFC (which are uncompressed). This way the buckets delimited by these strings have variable length, but any string can be decompressed in

optimal time and I/Os (cfr. previous observation about the $\Theta(B^2)$ size of a bucket in classic FC_B). So the bounds are the ones stated in Theorem 1.3 but without the pathological cases commented next to its proof. This way the scanning of a bucket, identified by the binary-search step takes $O(1)$ I/O and time proportional to the returned strings, and hence it is optimal.

The remaining question is therefore how to speed-up the search over the array A. We foresee two main limitations: (i) the binary-search step has time complexity depending on N or n, (ii) if the pointed strings do not fit within the internal-memory space allocated by the programmer, or available in cache, then the binary-search step incurs many I/Os, and this might be expensive. In the next sections we propose a trie-based approach that takes full-advantage of LPFC by overcoming these limitations, resulting efficient in time, I/Os and space.

1.4 Compacted Trie

We already talked about tries in Chapter **??**, here we dig further into their properties as efficient data structures for string searching. In our context, the trie is used for the indexing of the sampled strings $\mathcal{D}_B$ in internal memory. This induces a speed up in the first stage of the prefix search from $O(\log(N/B))$ to $O(p)$ time, thus resulting surprisingly independent of the dictionary size. The reason is the power of the RAM model which allows to manage and address memory-cells of $O(\log N)$ bits in constant time.

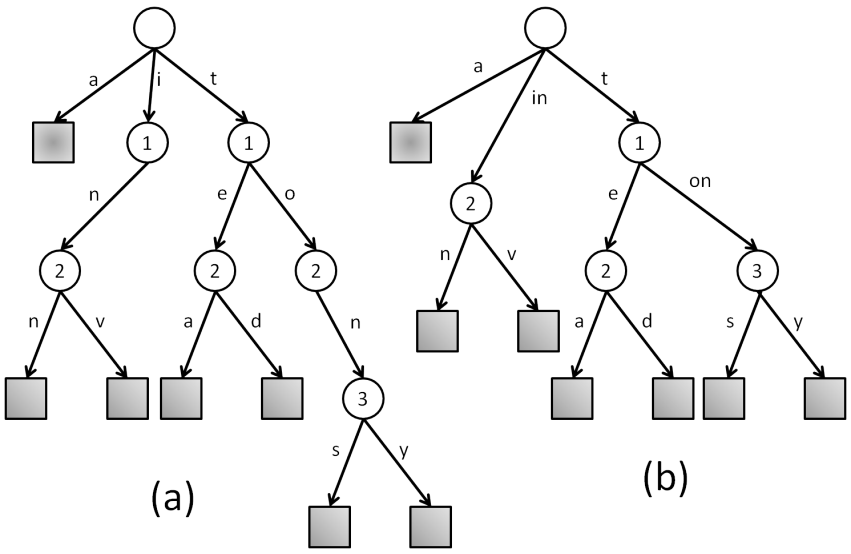


FIGURE 1.6: An example of uncompacted trie (a) and compacted trie (b) for $n = 7$ strings. The integer showed in each internal node u denotes the length of the string spelled out by u. In the case of uncompacted tries they are useless because they correspond to u's depth. Edge labels in compacted tries are substrings of variable length but they can be represented in $O(1)$ space with triples of integers: e.g. `on` could be encoded as $\langle 6, 2, 3\rangle$, since the 6-th string `tons` includes `on` from position 2 to position 3.

A trie is a multi-way tree whose edges are labeled by characters of the indexed strings. An internal

node u is associated with a string $s[u]$ which is indeed a *prefix* of a dictionary string. String $s[u]$ is obtained by concatenating the characters found on the downward path that connects the trie's root with the node u. A leaf is associated with a dictionary string. All leaves which descend from a node u are prefixed by $s[u]$. The trie has n leaves and at most N nodes, one per string character.[3] Figure 1.6 provides an illustrative example of a trie built over 6 strings. This form of trie is commonly called *uncompacted* because it can have *unary paths*, such as the one leading to string `inn`.[4]

If we want to check if a string P prefixes some dictionary string, we have just to check if there is a downward path spelling out P. All leaves descending from the reached node provide the correct answer to our prefix search. So tries do not need the reduction to the lexicographic search operation, introduced for the binary-search approach.

A big issue is how to efficiently find the "edge to follow" during the downward traversal of the trie, because this impacts onto the overall efficiency of the pattern search. The efficiency of this step hinges on a proper storage of the edges (and their labeling characters) outgoing from a node. The simplest data structure that does the job is the *linked list*. Its space requirement is optimal, namely proportional to the number of outgoing edges, but it incurs in a $O(\sigma)$ cost per traversed node. The result would be a prefix search taking $O(p\ \sigma)$ time in the worst case, which is too much for large alphabets. If we store the branching characters (and their edges) into a sorted array, then we could binary search it taking $O(\log \sigma)$ time per node. A faster approach consists of using a full-sized array of σ entries, the un-empty entries (namely the ones for which the pointer in not null) are the entries corresponding to the existing branching characters. In this case the time to branch out of a node is $O(1)$ and thus $O(p)$ time is the cost for searching the pattern Q. But the space occupancy of the trie grows up to $O(N\sigma)$, which may be unacceptably high for large alphabets. The best approach consists of resorting a *perfect hash table*, which stores just the existing branching characters and their associated pointers. This guarantees $O(1)$ branching time in the worst-case and optimal space occupancy, thus combining the best of the two previous solutions. For details about perfect hashes we refer the reader to Chapter **??**.

THEOREM 1.6 *The uncompacted trie solves the prefix-search problem in $O(p+n_{occ})$ time and $O(p + n_{occ}/B)$ I/Os, where n_{occ} is the number of strings prefixed by P. The retrieval of those strings prefixed by P takes $O(N_{occ})$ time, and it takes $O(N_{occ}/B)$ I/Os provided that leaves and strings are stored contiguously and alphabetically sorted on disk. The trie consists of at most N nodes, exactly n leaves, and thus takes $O(N)$ space. The retrieval of the result strings takes $O(N_{occ})$ time and $O(N_{occ}/B)$ I/Os, where N_{occ} is the total length of the retrieved strings.*

Proof Let u be the node such that $s[u] = P$. All strings descending from u are prefixed by P, and they can be visualized by visiting the subtree rooted in u. The I/O-complexity of the traversal is still $O(p)$ because of the jumps among trie nodes. The retrieval of the n_{occ} leaves descending from the node spelling Q takes optimal $O(n_{occ}/B)$ I/Os because we can assume that trie leaves are stored contiguously from left-to-right on disk. Notice that we have identified the strings (leaves) prefixed by Q but, in order to display them, we still need to retrieve them, this takes additional $O(N_{occ}/B)$ I/Os provided that the indexed strings are stored contiguously on disk. This is $O(n_{occ}/B)$ I/Os if we are interested only in the string pointers/IDs, provided that every internal node keeps a pointer to its leftmost descending leaf and all leaves are stored contiguously on disk. (These are the main reasons

[3] We say "at most" because some paths (prefixes) can be shared among several strings.

[4] The trie cannot index strings which are one the prefix of the other. In fact the former string would end up into an internal node. To avoid this case, each string is extended with a special character which is not present in the alphabet and is typically denoted by \$.

for keeping the pointers in the leaves of the uncompacted trie, which anyway stores the strings in its edges, and thus could allow to retrieve them but with more I/Os because of the difficulty to pack arbitrary trees on disk.) ■

A Trie can be wasteful in space if there are long strings with a short common prefix: this would induce a significant number of unary nodes. We can save space by *contracting* the unary paths into one single edge. This way edge labels become (possibly long) sub-strings rather than characters, and the resulting trie is named *compacted*. Figure 1.7 (left) shows an example of compacted trie. It is evident that each edge-label is a substring of a dictionary string, say $s[i, j]$, so it can be represented via a triple $\langle s, i, j\rangle$ (see also Figure 1.6). Given that each node is at least binary, the number of internal nodes and edges is $O(n)$. So the total space required by a compacted trie is $O(n)$ too.

Prefix searching is implemented similarly as done for uncompacted tries. The difference is that it alternates character-branches out of internal nodes, and sub-string matches with edge labels. If the edges spurring from the internal nodes are again implemented with perfect hast tables, we get:

THEOREM 1.7 *The compacted trie solves the prefix-search problem in $O(p + n_{occ})$ time and $O(p + n_{occ}/B)$ I/Os, where n_{occ} is the number of strings prefixed by P. The retrieval of those strings prefixed by P takes $O(N_{occ})$ time, and and it takes $O(N_{occ}/B)$ I/Os provided that leaves and strings are stored contiguously and alphabetically sorted on disk. The compacted trie consists of $O(n)$ nodes, and thus its storage takes $O(n)$ space. It goes without saying that the trie needs also the storage of the dictionary strings to resolve its edge labels, hence additional N space.*

At this point an attentive reader can realize that the compacted trie can be used also to search for the lexicographic position of a string Q among the indexed strings. It is enough to percolate a downward path spelling Q as much as possible until a mismatch character is encountered. This character can then be deployed to determine the lexicographic position of Q, depending on whether the percolation stopped in the middle of an edge or in a trie node. So the compacted trie is an interesting substitute for the array A in our two-level indexing structure and could be used to support the search for the candidate bucket where the string Q occurs in, taking $O(p)$ time in the worst case. Since each traversed edge can induce one I/O, to fetch its labeling substring to be compared with the corresponding one in Q, we point out that this approach is efficient if the trie and its indexed strings can be fit in internal memory. Otherwise it presents two main drawbacks: the linear dependance of the I/Os on the pattern length p, and the space dependance on the block-size B (influencing the the sampling) and the length of the sampled strings.

The *Patricia Trie* solves the former problem, whereas its combination with the LPFC solves both of them.

1.5 Patricia Trie

A Patricia Trie built on a string dictionary is a compacted Trie in which the edge labels consist just of their initial *single characters*, and the internal nodes are labeled with integers denoting the *lengths* of the associated strings. Figure 1.7 illustrates how to convert a Compacted Trie (left) into a Patricia Trie (right).

Even if the Patricia Trie strips out some information from the Compacted Trie, it is still able to support the search for the lexicographic position of a pattern P among a (sorted) sequence of strings, with the significant advantage (discussed below) that this search needs to access only one single string, and hence execute typically one I/O instead of the p I/Os potentially incurred by the edge-resolution step in compacted tries. This algorithm is called *blind search* in the literature [4].

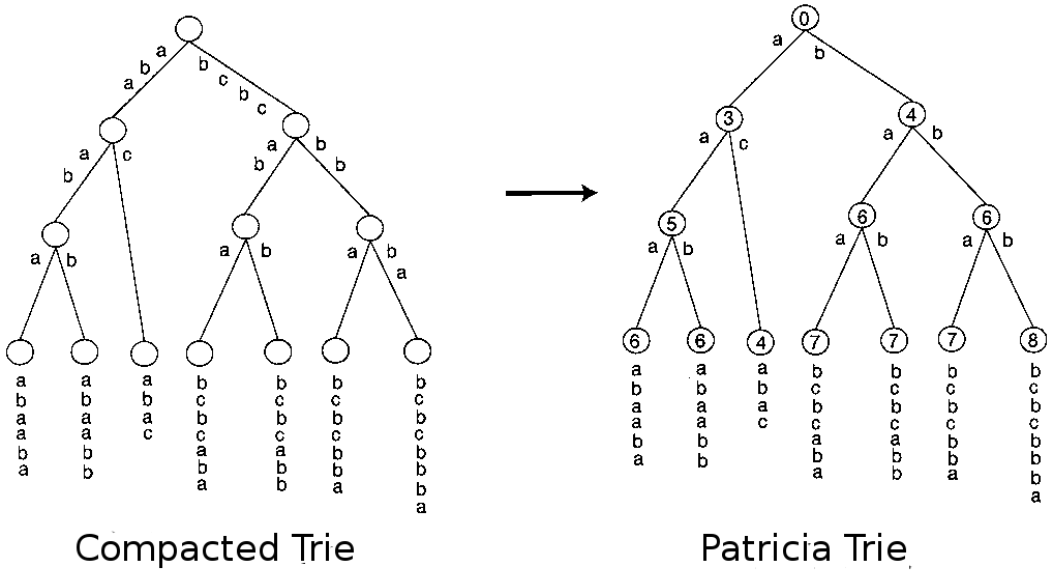


FIGURE 1.7: An example of Compacted Trie and the corresponding Patricia Trie.

It is a little bit more complicated than the prefix-search in classic tries, because of the presence of only one character per edge label, and in fact it consists of three stages:

- Trace a downward path in the Patricia Trie to locate a leaf l which points to an interesting string of the indexed dictionary. This string does not necessarily identify P's lexicographic position in the dictionary (which is our goal), but it provides *enough information* to find that position in the second stage. The retrieval of the interesting leaf l is done by traversing the Patricia Trie from the root and comparing the characters of P with the single characters which label the traversed edges until either a leaf is reached or no further branching is possible. In this last case, we choose l to be any descendant leaf from the last traversed node.
- Compare P against the string pointed by leaf l, in order to determine their longest common prefix. Let ℓ be the length of this shared prefix, then it is possible to prove that (see [4]) the leaf l stores one of the strings indexed by the Patricia Trie that shares the longest common prefix with P. Call s this pointed string. The length ℓ and the two mismatch characters $P[\ell + 1]$ and $s[\ell + 1]$ are then used to find the lexicographic position of P among the strings stored in the Patricia Trie.
- First the Patricia trie is traversed upward from l to determine the edge $e = (u, v)$ where the mismatch character $s[\ell + 1]$ lies; this is easy because each node on the upward path stores an integer that denotes the length of the corresponding prefix of s, so that we have $|s[u]| < \ell \leq |s[v]|$. If $s[\ell + 1]$ is a branching character (i.e. $\ell = |s[u]|$), then we determine the lexicographic position of $P[\ell + 1]$ among the branching characters of node u. Say this is the i-th child of u, the lexicographic position of P is therefore to the immediate left of the subtree descending from this child. Otherwise (i.e. $\ell > |s[u]|$), the character $s[\ell + 1]$ lies within e, so the lexicographic position of P is to the immediate right of the subtree descending from e, if $P[\ell + 1] > s[\ell + 1]$, otherwise it is to the immediate left of that subtree.

A running example is illustrated in Figure 1.8.

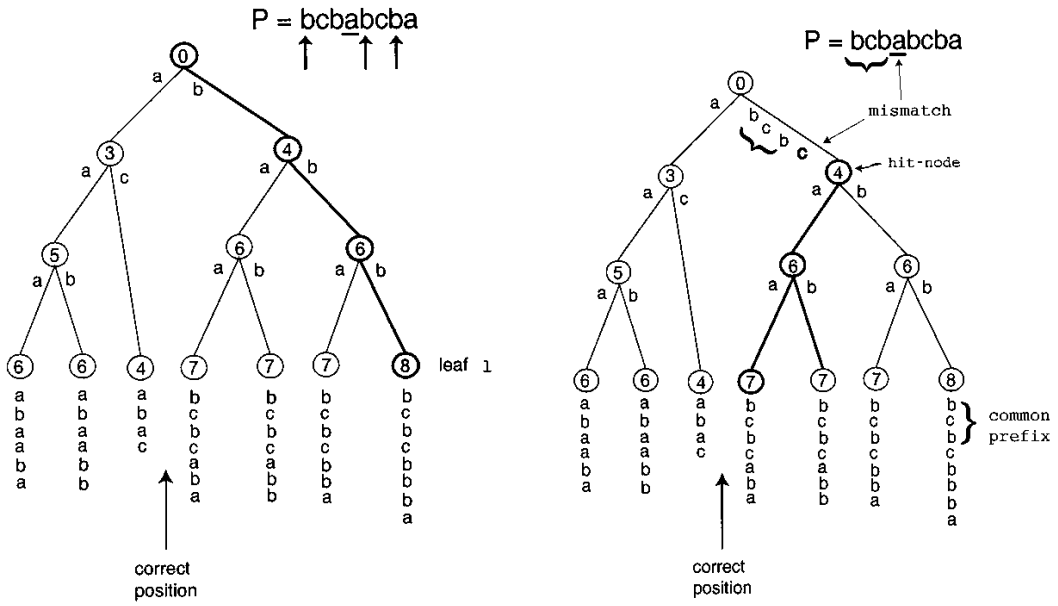


FIGURE 1.8: An example of the first (left) and second (right) stages of the blind search for P in a dictionary of 7 strings.

In order to understand why the algorithm is correct, let us take the path spelling out the string $P[1, \ell]$. We have two cases, either we reached an internal node u such that $|s[u]| = \ell$ or we are in the middle of an edge (u, v), where $|s[u]| < \ell < |s[v]|$. In the former case, all strings descending from u are the ones in the dictionary which share ℓ characters with the pattern, and this is the lcp. The correct lexicographic position therefore falls among them or is adjacent to them, and thus it can be found by looking at the branching characters of the edges outgoing from the node u. This is correctly done also by the blind search that surely stops at u, computes ℓ and finally determines the correct position of P by comparing u's branching characters against $P[\ell + 1]$.

In the latter case the blind search reaches v by skipping the mismatch character on (u, v), and possibly goes further down in the trie because of the possible match between branching characters and further characters of P. Eventually a leaf descending from v is taken, and thus ℓ is computed correctly given that all leaves descending from v share ℓ characters with P. So the backward traversal executed in the second stage of the Blind search reaches correctly the edge (u, v), which is above the selected leaf. There we deploy the mismatch character which allows to choose the correct lexicographic position of P which is either to the left of the leaves descending from v or to their right. Indeed all those leaves share $|s[v]| > \ell$ characters, and thus P falls adjacent to them, either to their left or to their right. The choice depends on the comparison between the two characters $P[\ell + 1]$ and $s[v][\ell + 1]$.

The blind search has excellent performance:

THEOREM 1.8 *A Patricia trie takes $O(n)$ space, hence $O(1)$ space per indexed string (in-*

dependent, therefore, of its length). The blind search for a pattern $P[1, p]$ requires $O(p)$ time to traverse the trie's structure (downward and upward), and $O(p/B)$ I/Os to compare the single string (possibly residing on disk) identified by the blind search. It returns the lexicographic position of P among the indexed strings. By searching for P and P#, as done in suffix arrays, the blind search determines the range of indexed strings prefixed by P, if any.

This theorem states that if $n < M$ then we can index in internal memory the whole dictionary, and thus build the Patricia trie over all dictionary strings and stuff it in the internal memory of our computer. The dictionary strings are stored on disk. The prefix search for a pattern P takes in $O(p)$ time and $O(p/B)$ I/Os. The total required space is the one needed to store the strings, and thus it is $O(N)$.

If we wish to compress the dictionary strings, then we need to resort front-coding. More precisely, we combine the Patricia Trie and LPFC as follows. We fit in the internal memory the Patricia trie of the dictionary $\mathcal{D}$, and store on disk the locality-preserving front coding of the dictionary strings. The two traversals of the Patricia trie take $O(p)$ time and no I/Os (Theorem 1.8), because use information stored in the Patricia trie and thus available in internal memory. Conversely the computation of the lcp takes $O(|s|/B + p/B)$ I/Os, because it needs to decode from its LPFC-representation (Theorem 1.5) the string s selected by the blind search and it also needs to compare s against P to compute their lcp. These information allow to identify the lexicographic position of P among the leaves of the Patricia trie.

THEOREM 1.9 *The data structure composed of the Patricia Trie as the index in internal memory ("upper level") and the LPFC for storing the strings on disk ("lower level") requires $O(n)$ space in memory and $O((1+\epsilon)FC(\mathcal{D}))$ space on disk. Furthermore, a prefix search for P requires $O(\frac{p}{B} + \frac{|s|}{B\epsilon})$ I/Os, where s is the "interesting string" determined in the first stage of the Blind search. The retrieval of the prefixed strings takes $O(\frac{(1+\epsilon)FC(\mathcal{D}_{occ})}{B})$ I/Os, where $\mathcal{D}_{occ} \subseteq \mathcal{D}$ is the set of returned strings.*

In the case that $n = \Omega(M)$, we cannot index in the internal-memory Patricia trie the whole dictionary, so we have to resort the bucketing strategy over the strings stored on disk and index in the Patricia trie only a sample of them. If $N/B = O(M)$ we can index in internal memory the first string of every bucket and thus be able to prefix-search P within the bounds stated in Theorem 1.9, by adding just one I/O due to the scanning of the bucket (i.e. disk page) containing the lexicographic position of P. The previous condition can be rewritten as $N = O(MB)$ which is pretty reasonable in practice, given the current values of $M \approx$ 4Gb and $B \approx$ 32Kb, which make $MB \approx$ 128Tb.

1.6 Managing Huge Dictionaries$^{\infty}$

The final question we address in this lecture is: What if $N = \Omega(MB)$? In this case the Patricia trie is too big to be fit in the internal memory of our computer. We can think to store the trie on disk without taking much care on the layout of its nodes among the disk pages. Unfortunately a pattern search could take $\Omega(p)$ I/Os in the two traversals performed by the Blind search. Alternatively, we could incrementally grow a root page and repeatedly add some node not already packed into that page, where the choice of that node might be driven by various criteria that either depend on some access probability or on the node's depth. When the root page contains B nodes, it is written onto disk and the algorithm recursively lays out the rest of the tree. Surprisingly enough, the obtained packing is far from optimality of a factor $\Omega(\frac{\log B}{\log\log B})$, but it is surely within a factor $O(\log B)$ from the optimal [1].

In what follows we describe two distinct optimal approaches to solve the prefix-search over dictionaries of huge size: the first solution is based on a data structure, called the *String B-Tree* [4], which boils down to a B-tree in which the routing table of each node is a Patricia tree; the second solution consists of applying proper *disk layouts of trees* onto the Patricia trie built over the entire dictionary.

1.6.1 String B-Tree

The key idea consists of dividing the big Patricia trie into a set of smaller Patricia tries, each fitting into one disk page. And then linking together all of them in a B-Tree structure. Below we outline a constructive definition of the String B-Tree, for details on this structure and the supported operations we refer the interested reader to the cited literature.

The dictionary strings are stored on disk contiguously and ordered. The pointers to these strings are partitioned into a set of smaller, equally sized chunks $\mathcal{D}_1, \ldots, \mathcal{D}_m$, each including $\Theta(B)$ strings independently of their length. This way, we can index each chunk $\mathcal{D}_i$ with a Patricia Trie that fits into one disk page and embed it into a leaf of the B-Tree. In order to search for P among those set of nodes, we take from each partition $\mathcal{D}_i$ its *first* and *last* (lexicographically speaking) strings s_{if} and s_{il}, defining the set $\mathcal{D}^1 = \left\{s_{1f}, s_{1l}, \ldots, s_{mf}, s_{ml}\right\}$.

Recall that the prefix search for P boils down to the lexicographic search of a pattern Q, properly defined from P. If we search Q within $\mathcal{D}^1$, we can discover one of the following three cases:

1. Q falls before the first or after the last string of $\mathcal{D}$, if $Q < s_{1f}$ or $Q > s_{ml}$.
2. Q falls among the strings of some $\mathcal{D}_i$, and indeed it is $s_{if} < Q < s_{il}$. So the search is continued in the Patricia trie that indexes $\mathcal{D}_i$;
3. Q falls between two chunks, say $\mathcal{D}_i$ and $\mathcal{D}_{i+1}$, and indeed it is $s_{il} < Q < s_{(i+1)f}$. So we found Q's lexicographic position in the whole $\mathcal{D}$, namely it is between these two adjacent chunks.

In order to establish which of the three cases occurs, we need to search efficiently in $\mathcal{D}^1$ for the lexicographic position of Q. Now, if $\mathcal{D}^1$ is small and can be fit in memory, we can build on it a Patricia trie ad we are done. Otherwise we repeat the partition process on $\mathcal{D}^1$ to build a smaller set $\mathcal{D}^2$, in which we sample, as before, two strings every B, so that $|\mathcal{D}^2| = \frac{2|\mathcal{D}^1|}{B}$. We continue this partitioning process for k steps, until it is $|\mathcal{D}^k| = O(B)$ and thus we can fit the Patricia trie built on $\mathcal{D}^k$ within one disk page[5].

We notice that each disk page gets an even number of strings when partitioning $\mathcal{D}^1, \ldots, \mathcal{D}^k$, and to each pair $\left(s_{if}, s_{il}\right)$ we associate a pointer to the block of strings which they delimit in the lower level of this partitioning process. The final result of the process is then a B-Tree over string pointers. The *arity* of the tree is $\Theta(B)$, because we index $\Theta(B)$ strings in each single node. The nodes of the String B-Tree are then stored on disk. The following Figure 1.9 provides an illustrative example for a String B-tree built over 7 strings.

A (prefix) search for the string P in a String B-Tree is simply the traversal of the B-Tree, which executes at each node a lexicographic search of the proper pattern Q in the Patricia trie of that node. This search discovers one of the three cases mentioned above, in particular:

- case 1 can only happen on the root node;
- case 2 implies that we have to follow the node pointer associated to the identified partition.

[5] Actually, we could stop as soon as $\left|\mathcal{D}^k\right| = O(M)$, but we prefer the former to get a standard B-Tree structure.

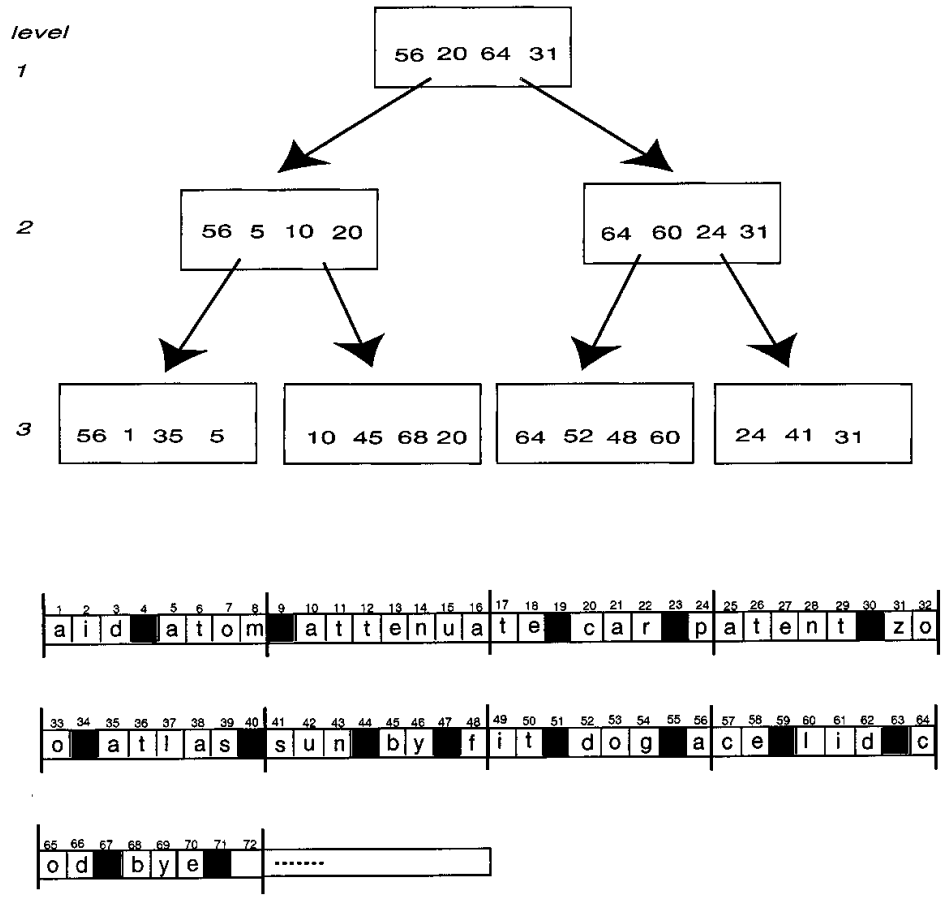


FIGURE 1.9: An example of an String B-tree on built on the suffixes of the strings in $\mathcal{D}$ = {*'ace'*, *'aid'*, *'atlas'*, *'atom'*, *'attenuate'*, *'by'*, *'bye'*, *'car'*, *'cod'*, *'dog'*, *'fit'*, *'lid'*, *'patent'*, *'sun'*, *'zoo'*}. The strings are stored in the B-tree leaves by means of their logical pointers 56, 1, 35, 5, 10, ..., 31. Notice that strings are not sorted on disk, nevertheless sorting improves their I/O-scanning, and indeed our theorems assume an ordered $\mathcal{D}$ on disk.

- case 3 has found the lexicographic position of Q in the dictionary $\mathcal{D}$, so the search in the B-tree stops.

The I/O complexity of the data structure just defined is pretty good: since the arity of the B-Tree is $\Theta(B)$, we have $\Theta(\log_B n)$ levels, so a search traverses $\Theta(\log_B n)$ nodes. Since on each node we need to load the node's page into memory and perform a Blind search over its Patricia trie, we pay $O(1 + \frac{p}{B})$ I/Os, and thus $O(\frac{p}{B} \log_B n)$ I/Os for the overall prefix search of P in the dictionary $\mathcal{D}$.

THEOREM 1.10 *A prefix search in the String B-Tree built over the dictionary $\mathcal{D}$ takes $O(\frac{p}{B} \log_B n + \frac{N_{occ}}{B})$ I/Os, where N_{occ} is the total length of the dictionary strings which are prefixed by P. The data structure occupies $O(\frac{N}{B})$ disk pages and, indeed, strings are stored uncompressed on disk.*

This result is good but not yet *optimal*. The issue that we have to resolve to reach optimality is

pattern rescanning: each time we do a Blind search, we compare Q and one of the strings stored in the currently visited B-Tree node starting from their first character. However, as we go down in the string B-tree we can capitalize on the characters of Q that we have already compared in the upper levels of the B-tree, and thus avoid the rescanning of these characters during the subsequent lcp-computations. So if f characters have been already matched in Q during some previous lcp-computation, the next lcp-computation can compare Q with a dictionary string starting from their $(f + 1)$-th character. The pro of this approach is that I/Os turn to be optimal, the cons is that strings have to be stored uncompressed in order to support the efficient access to that $(f + 1)$-th character. Working out all the details [4], one can show that:

THEOREM 1.11 *A prefix search in the String B-Tree built over the dictionary $\mathcal{D}$ takes $O(\frac{p+N_{occ}}{B} + \log_B n)$ optimal I/Os, where N_{occ} is the total length of the dictionary strings which are prefixed by P. The data structure occupies $O(\frac{N}{B})$ disk pages and, indeed, strings are stored uncompressed on disk.*

If we want to store the strings compressed on disk, we cannot just plug LPFC in the approach illustrated above, because the decoding of LPFC works only on full strings, and thus it does not support the efficient skip of some characters without wholly decoding the compared string. [2] discusses a sophisticated solution to this problem which gets the I/O-bounds in Theorem 1.11 but in the cache-oblivious model and guaranteeing LPFC-compressed space. We refer the interested reader to that paper for details.

1.6.2 Packing Trees on Disk

We point out that the advantage of finding a good layout for unbalanced trees among disk pages (of size B) may be unexpectedly large, and therefore, must not be underestimated when designing solutions that have to manage large trees on disk. In fact, while balanced trees save a factor $O(\log B)$ when mapped to disk (pack B-node balanced subtrees per page), the mapping of unbalanced trees grows with non uniformity and approaches, in the extreme case of a linear-height tree, a saving factor of $\Theta(B)$ over a naïve memory layout.

This problem is also known in the literature as the *Tree Packing* problem. Its goal is to find an allocation of tree nodes among the disk pages in such a way that the number of I/Os executed for a pattern search is minimized. Minimization may involve either the total number of loaded pages in internal memory (i.e. page faults) , or the number of distinct visited pages (i.e. working-set size). This way we model two extreme situations: the case of a one-page internal memory (i.e. a small buffer), or the case of an unbounded internal memory (i.e. an unbounded buffer). Surprisingly, the optimal solution to the tree packing problem is *independent* of the available buffer size because no disk page is visited twice when page faults are minimized or the working set is minimum. Moreover, the optimal solution shows a nice *decomposability property*: the optimal tree packing forms in turn a tree of disk pages. These two facts allow to restrict our attention to the page-fault minimization problem, and to the design of recursive approaches to the optimal tree decomposition among the disk pages.

In the rest of this section we present two solutions of increasing sophistication and addressing two different scenarios: one in which the goal is to *minimize the maximum number* of page faults executed during a downward root-to-leaf traversal; the other in which the goal is to *minimize the average number* of page faults by assuming an access distribution to the tree leaves, and thus to the possible tree traversals. We briefly mention that both solutions assume that B is known; the literature actually offers cache-oblivious solutions to the tree packing problem, but they are too much sophisticated to be reported in these notes. For details we refer the reader to [1, 5].

Min-Max Algorithm. This solution operates greedily and bottom up over the tree to be packed with

the goal of minimizing the maximum number of page faults executed during a downward traversal which starts from the root of the tree. The tree is assumed to be binary, this is not a restriction for Patricia Tries because it is enough to encode the alphabet characters with binary strings. The algorithm assigns every leaf to its own disk page and the height of this page is set to 1. Working upward, Algorithm 1.1 is applied to each processed node until the root of the tree is reached.

Algorithm 1.1 Min-Max Algorithm over binary trees (general step).

```
Let u be the currently visited node;
if If both children of u have the same page height d then
  if If the total number of nodes in both children's pages is < B then
    Merge the two disk pages and add u;
    Set the height of this new page to d;
  else
    Close off the pages of u's children;
    Create a new page for u and set its height to d + 1;
  end if
else
  Close off the page of u's child with the smaller height;
  If possible, merge the page of the other child with u and leave its height unchanged;
  Otherwise, create a new page for u with height d + 1 and close off the child's page;
end if
```

The final packing may induce a poor page-fill ratio, nonetheless several changes can alleviate this problem in real situations:

1. When a page is closed off, scan its children pages from the smallest to the largest and check whether they can be merged with their parent.
2. Design logical disk pages and pack many of them into one physical disk page; possibly ignore physical page boundaries when placing logical pages onto disk.

THEOREM 1.12 *The Min-Max Algorithm provides a disk-packing of a tree of n nodes and height H such that every root-to-leaf path traverses less than* $1 + \lceil \frac{H}{\sqrt{B}} \rceil + \lceil 2 log_B n \rceil$ *pages.*

Distribution-aware Packing. We assume that it is known an access distribution to the Patricia trie leaves. Since this distribution is often skewed towards some leaves, that are then accessed more frequently than others, the Min-Max algorithm may be significantly inefficient. The following algorithm is based on a Dynamic-Programming scheme, and optimizes the *expected* number of I/Os incurred by any traversal of a root-to-leaf path.

We denote by τ this optimal tree packing (from tree nodes to disk pages), so $\tau(u)$ denotes the disk page to which the tree node u is mapped. Let $w(f)$ be the probability to access a leaf f, we derive a distribution over all other nodes u of the tree by summing up the access probabilities of its descending leaves. We can assume that the tree root r is always mapped to a fixed page $\tau(r) = R$. Consider now the set V of tree nodes that descend from R's nodes but are not themselves in R. We observe that the optimal packing τ induces a tree of disk pages and consequently, if τ is optimal for the current tree T, then τ is optimal for all subtrees T_v rooted in $v \in V$.

This result allows to state a recursive computation for τ that first determines which nodes reside in R, and then continues recursively with all subtrees T_v for which $v \in V$. Dynamic programming

provides an efficient implementation of this idea, based on the following definition: An *i-confined* packing of a tree T is a packing in which the page R contains exactly i nodes (clearly $i \leq B$). Now, in the optimal packing τ, the root page R will contain i^* nodes from the left subtree $T_{left(r)}$ and $(B - i^* - 1)$ nodes from the right subtree $T_{right(r)}$, for some i^*. The consequence is that τ is both an optimal i^*-confined packing for $T_{left(r)}$ and an optimal $(B - i^* - 1)$-confined packing for $T_{right(r)}$. This property is at the basis of the Dynamic-Programming rule which computes $A[v, i]$, for a generic node v and integer $i \leq B$, as the cost of an optimal i-confined packing of the subtree T_v. In the paper [5] the authors showed that $A[v, i]$, for $i > 1$, can be computed as the access probability $w(v)$ plus the minimum among the following three quantities:

1. $A[left(v), i - 1] + w(right(v)) + A[right(v), B]$
2. $w(left(v)) + A[left(v), B] + A[right(v), i - 1]$
3. $\min_{1 \leq j < i-1}\{A[left(v), j] + A[right(v), i - j - 1]\}$

Rule (1) accounts for the (unbalanced) case in which the i-confined packing is obtained by storing $i - 1$ nodes from $T_{left(v)}$ into the v's page; Rule (2) is the symmetric of Rule (1); whereas Rule (3) accounts for the case in which j nodes from $T_{left(v)}$ and $i - j - 1$ nodes from $T_{right(v)}$ are stored into the page of v to form the optimal i-confined packing of T_v. The special case $i = 1$ is given by $A[v, 1] = w(T_v) + A[left(v), B] + A[right(v), B]$.

Algorithm 1.2 deploys these rules to compute the optimal tree packing in $O(nB^2)$ time and $O(nB)$ space.

Algorithm 1.2 Distribution-aware packing of trees on disk.

```
Initialize A[v, i] = w(v), for all leaves v and integers i ≤ B;
while there exist an unmarked node v do
    mark v;
    update A[v, 1] = w(v) + A[left(v), B] + A[right(v), B];
    for i = 2 to B do
        update A[v, i] according to the dyn-prog rule specified in the text.
    end for
end while
```

THEOREM 1.13 *An optimal packing for a f-ary tree of n nodes can be computed in $O(nB^2 \log f)$ time and $O(B \log n)$ space. The packing maps the tree into at most $2\lfloor \frac{n}{B} \rfloor$ disk pages. Optimality is with respect to the* expected *number of I/Os incurred by any root-to-leaf traversal.*

References

[1] Stefan Alstrup, Michael A. Bender, Erik D. Demaine, Martin Farach-Colton, Jan I. Munro, Theis Rauhe, and M. Thorup. *Efficient Tree Layout in a Multilevel Memory Hierarchy*, 2003. Personal Communication, corrected version of a paper appeared in the *European Symposium on Algorithms 2002*.

[2] Michael A. Bender, Martin Farach-Colton, and Bradley C. Kuszmaul. Cache-oblivious string B-trees. In *Procs ACM Symposium on Principles of Database Systems*, pages 223–242, 2006.

[3] Erik D. Demaine, Thouis Jones, and Mihai Pătraşcu. Interpolation search for non-independent data. In *Procs ACM-SIAM Symposium on Discrete algorithms*, pages 529–530, 2004.

[4] Paolo Ferragina and Roberto Grossi. The String B-tree: a new data structure for string search in external memory and its applications. *Journal of the ACM*, 46(2):236–280, 1999.

[5] Joseph Gil and Alon Itai. How to pack trees. *Journal of Algorithms*, 32(2):108–132, 1999.

[6] Michael Luby, Charles Rackoff. How to construct pseudorandom permutations from pseudorandom functions. *SIAM Journal on Computing*, 17, 373–386, 1988.

2

Searching Strings by Substring

In this lecture we will be interested in solving the following problem, known as *full-text searching* or *substring searching*.

The substring-search problem. *Given a text string $T[1,n]$, drawn from an alphabet of size σ, retrieve (or just count) all text positions where a query pattern $P[1,p]$ occurs as a substring of T.*

It is evident that this problem can be solved by brute-forcedly comparing P against every substring of T, thus taking $O(np)$ time in the worst case. But it is equivalently evident that this *scan*-based approach is unacceptably slow when applied to massive text collections subject to a massive number of queries, which is the scenario involving genomic databases or search engines. This suggests the usage of a so called *indexing* data structure which is built over T before that searches start. A setup cost is required for this construction, but this cost is amortized over the subsequent pattern searches, thus resulting convenient in a quasi-static environment in which T is changed very rarely.

In this lecture we will describe two main approaches to substring searching, one based on arrays and another one based on trees, that mimic what we have done for the prefix-search problem. The two approaches hinge on the use of two fundamental data structures: the *suffix array* (shortly *SA*) and the *suffix tree* (shortly *ST*). We will describe in much detail those data structures because their use goes far beyond the context of full-text search.

2.1 Notation and terminology

We assume that text T ends with a special character $T[n] = \$$, which is smaller than any other alphabet character. This ensures that text suffixes are prefix-free and thus no one is a prefix of another suffix. We use $suff_i$ to denote the i-th suffix of text T, namely the substring $T[i,n]$. The following observation is crucial:

If $P = T[i, i+p-1]$, then the pattern occurs at text position i and thus we can state that P is a prefix of the i-th text suffix, namely P is a prefix of the string $suff_i$.

As an example, if P ="siss" and T ="mississippi$", then P occurs at text position 4 and indeed it prefixes the suffix $suff_4 = T[4, 12]$ ="sissippi$". For simplicity of exposition, and for historical reasons, we will use this text as running example; nevertheless we point out that a text may be an arbitrary sequence of characters, hence not necessarily a single word.

Given the above observation, we can form with all text suffixes the dictionary $SUF(T)$ and state that *searching for P as a substring of T boils down to searching for P as a prefix of some string in* $SUF(T)$. In addition, since there is a bijective correspondence among the text suffixes prefixed by P and the pattern occurrences in T, then

1. the suffixes prefixed by P occur contiguously into the lexicographically sorted $SUF(T)$,
2. the lexicographic position of P in $SUF(T)$ immediately precedes the block of suffixes prefixed by P.

An attentive reader may have noticed that these are the properties we deployed to efficiently support prefix searches. And indeed the solutions known in the literature for efficiently solving the substring-search problem hinge either on array-based data structures (i.e. the Suffix Array) or on trie-based data structures (i.e. the Suffix Tree). So the use of these data structures in pattern searching is pretty immediate. What is challenging is the efficient construction of these data structures and their mapping onto disk to achieve efficient I/O-performance. These will be the main issues dealt with in this lecture.

Text suffixes	Indexes
mississippi$	1
ississippi$	2
ssissippi$	3
sissippi$	4
issippi$	5
ssippi$	6
sippi$	7
ippi$	8
ppi$	9
pi$	10
i$	11
$	12

Sorted Suffixes	*SA*	***Lcp***
$	12	0
i$	11	1
ippi$	8	1
issippi$	5	4
ississippi$	2	0
mississippi$	1	0
pi$	10	1
ppi$	9	0
sippi$	7	2
sissippi$	4	1
ssippi$	6	3
ssissippi$	3	-

FIGURE 2.1: *SA* and `lcp` array for the string T ="mississippi$".

2.2 The Suffix Array

The suffix array for a text T is the array of pointers to all text suffixes ordered lexicographically. We use the notation $SA(T)$ to denote the suffix array built over T, or just SA if the indexed text is clear from the context. Because of the lexicographic ordering, $SA[i]$ is the i-th smallest text suffix, so we have that $suff_{SA[1]} < suff_{SA[2]} < \cdots < suff_{SA[n]}$, where $<$ is the lexicographical order between strings. For space reasons, each suffix is represented by its starting position in T (i.e. an integer). SA consists of n integers in the range $[1, n]$ and hence it occupies $O(n \log n)$ bits.

Another useful concept is the *longest common prefix* between two consecutive suffixes $suff_{SA[i]}$ and $suff_{SA[i+1]}$. We use `lcp` to denote the array of integers representing the lengths of those lcps. Array `lcp` consists of $n-1$ entries containing values smaller than n. There is an optimal and non obvious linear-time algorithm to build the lcp-array which will be detailed in Section 2.2.3. The interest in `lcp` rests in its usefulness to design efficient/optimal algorithms to solve various search and mining problems over strings.

2.2.1 The substring-search problem

We observed that this problem can be reduced to a prefix search over the string dictionary $SUF(T)$, so it can be solved by means of a binary search for P over the array of text suffixes ordered lexicographically, hence $SA(T)$. Figure 2.1 shows the pseudo-code which coincides with the classic binary-search algorithm specialized to compare strings rather than numbers.

Algorithm 2.1 SUBSTRINGSEARCH($P, SA(T)$)

```
1:  L = 1, R = n;
2:  while (L ≠ R) do
3:      M = ⌊(L + R)/2⌋;
4:      if (strncmp(P, suff_M, p) > 0) then
5:          L = M + 1;
6:      else
7:          R = M;
8:      end if
9:  end while
10: if (strncmp(P, suff_L, p) = 0) then
11:     return L;
12: else
13:     return −1;
14: end if
```

A binary search in SA requires $O(\log n)$ string comparisons, each taking $O(p)$ time in the worst case.

LEMMA 2.1 Given the text $T[1,n]$ and its suffix array, we can count the occurrences of a pattern $P[1,p]$ in the text taking $O(p \log n)$ time and $O(\log n)$ memory accesses in the worst case. Retrieving the positions of these occ occurrences takes additional $O(occ)$ time. The total required space is $n(\log n + \log \sigma)$ bits, where the first term accounts for the suffix array and the second term for the text.

Figure 2.2 shows a running example, which highlights an interesting property: the comparison between P and $suff_M$ does not need to start from their initial character. In fact one could exploit the lexicographic sorting of the suffixes and skip the characters comparisons that have already been carried out in previous iterations. This can be done with the help of three arrays:

- the `lcp`$[1, n-1]$ array;
- two other arrays $Llcp[1, n-1]$ and $Rlcp[1, n-1]$ which are defined for every triple (L, M, R) that may arise in the inner loop of a binary search. We define $Llcp[M] =$

$\texttt{lcp}(suff_{SA[L]}, suff_{SA[M]})$ and $Rlcp[M] = \texttt{lcp}(suff_{SA[M]}, suff_{SA[R]})$, namely $Llcp[M]$ accounts for the prefix shared by the leftmost suffix $suff_{SA[L]}$ and the middle suffix $suff_{SA[M]}$ of the range currently explored by the binary search; $Rlcp[M]$ accounts for the prefix shared by the rightmost suffix $suff_{SA[R]}$ and the middle suffix $suff_{SA[M]}$ of that range.

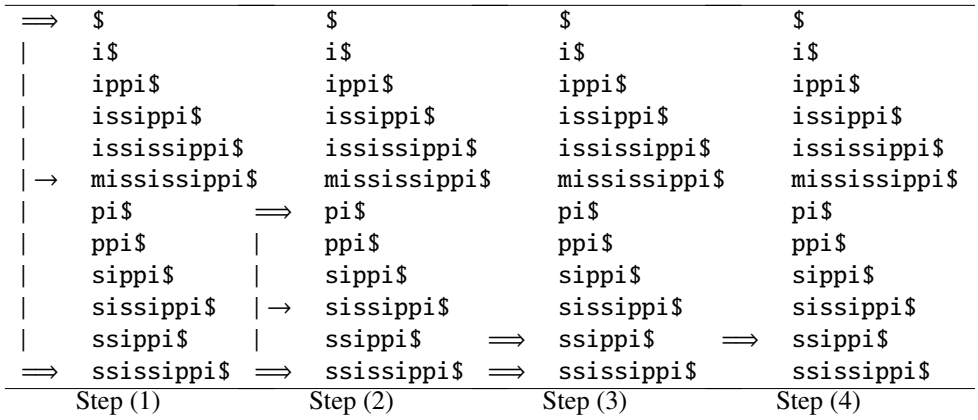


FIGURE 2.2: Binary search steps for the lexicographic position of the pattern P ="ssi" in "mississippi$".

We notice that each triple (L, M, R) is uniquely identified by its midpoint M because the execution of a binary search defines actually a hierarchical partition of the array SA into smaller and smaller sub-arrays delimited by (L, R) and thus centered in M. Hence we have $O(n)$ triples overall, and these three arrays occupy $O(n)$ space in total.

We can build arrays $Llcp$ and $Rlcp$ in linear time by exploiting two different approaches. We can deploy the observation that the $\texttt{lcp}[i, j]$ between the two suffixes $suff_{SA[i]}$ and $suff_{SA[j]}$ can be computed as the minimum of a range of lcp-values, namely $\texttt{lcp}[i, j] = \min_{k=i,\ldots,j-1} \texttt{lcp}[k]$. By associativity of the min we can split the computation as $\texttt{lcp}[i, j] = \min\{\texttt{lcp}[i, k], \texttt{lcp}[k, j]\}$ where k is any index in the range $[i, j]$, so in particular we can set $\texttt{lcp}[L, R] = \min\{\texttt{lcp}[L, M], \texttt{lcp}[M, R]\}$. This implies that the arrays $Llcp$ and $Rlcp$ can be computed via a bottom-up traversal of the triplets (L, M, R) in $O(n)$ time. Another way to deploy the previous observation is to compute $\texttt{lcp}[i, j]$ on-the-fly via a Range-Minimum Data structure built over the array lcp (see Section 2.4.1). All of these approaches take $O(n)$ time and space, and thus they are optimal.

We are left with showing how the binary search can be speeded up by using these arrays. Consider a binary-search iteration on the sub-array $SA[L, R]$, and let M be the midpoint of this range (hence $M = (L + R)/2$). A lexicographic comparison between P and $suff_{SA[M]}$ has to be made in order to choose the next search-range between $SA[L, M]$ and $SA[M, R]$. The goal is to compare P and $suff_{SA[M]}$ without starting necessarily from their first character, but taking advantage of the previous binary-search steps in order to infer, hopefully in constant time, their lexicographic comparison.

Surprisingly enough this is possible and requires to know, in addition to $Llcp$ and $Rlcp$, the values $l = \texttt{lcp}(P, suff_{SA[L]})$ and $r = \texttt{lcp}(P, suff_{SA[R]})$ which denote the number of characters the pattern P shares with the strings at the extremes of the range currently explored by the binary search. At the first step, in which $L = 1$ and $R = n$, these two values can be computed in $O(p)$ time by comparing character-by-character the involved strings. At a generic step, we assume that l and r are known

inductively, and show below how the binary-search step can preserve their knowledge after that we move onto $SA[L, M]$ or $SA[M, R]$.

So let us detail the implementation of a generic binary-search step. We know that P lies between $suff_{SA[L]}$ and $suff_{SA[R]}$, so P surely shares $\texttt{lcp}[L, R]$ characters with these suffixes given that any string (and specifically, all suffixes) in this range must share this number of characters (given that they are lexicographically sorted). Therefore the two values l and r are larger (or equal) than $\texttt{lcp}[L, R]$, as well as it is larger (or equal) to this value also the number of characters m that the pattern P shares with $suff_{SA[M]}$. We could then take advantage of this last inequality to compare P with $suff_{SA[M]}$ starting from their $(\texttt{lcp}[L, R] + 1)$-th character. But actually we can do better because we know r and l, and these values can be significantly larger than $\texttt{lcp}[L, R]$, thus more characters of P have been already involved in previous comparisons and so they are known.

We distinguish three main cases by assuming that $l \geq r$ (the other case $r > l$ is symmetric), and aim at not re-scanning the characters of P that have been already seen (namely characters in $P[1, l]$). We define our algorithm in such a way that the order between P and $suff_{SA[M]}$ can be inferred either comparing characters in $P[l + 1, n]$, or comparing the values l and $Llcp[M]$ (which give us information about $P[1, l]$).

- If $l < Llcp[M]$, then P is greater that $suff_{SA[M]}$ and we can set $m = l$. In fact, by induction, $P > suff_{SA[L]}$ and their mismatch character lies at position $l+1$. By definition of $Llcp[M]$ and the hypothesis, we have that $suff_{SA[L]}$ shares more than l characters with $suff_{SA[M]}$. So the mismatch between P and $suff_{SA[M]}$ is the same as it is between P and $suff_{SA[L]}$, hence their comparison gives the same answer— i.e. $P > suff_{SA[M]}$— and the search can thus continue in the subrange $SA[M, R]$. We remark that this case does not induce any character comparison.
- If $l > Llcp[M]$, this case is similar as the previous one. We can conclude that P is smaller than $suff_{SA[M]}$ and it is $m = Llcp[M]$. So the search continues in the subrange $SA[L, M]$, without additional character comparisons.
- If $l = Llcp[M]$, then P shares l characters with $suff_{SA[L]}$ and $suff_{SA[M]}$. So the comparison between P and $suff_{SA[M]}$ can start from their $(l+1)$-th character. Eventually we determine m and their lexicographic order. Here some character comparisons are executed, but the *knowledge* about P's characters advanced too.

It is clear that every binary-search step either advances the comparison of P's characters, or it does not compare any character but halves the range $[L, R]$. The first case can occur at most p times, the second case can occur $O(\log n)$ times. We have therefore proved the following.

LEMMA 2.2 Given the three arrays $\texttt{lcp}$, $Llcp$ and $Rlcp$ built over a text $T[1, n]$, we can count the occurrences of a pattern $P[1, p]$ in the text taking $O(p + \log n)$ time in the worst case. Retrieving the positions of these occ occurrences takes additional $O(occ)$ time. The total required space is $O(n)$.

Proof We remind that searching for all strings having the pattern P as a prefix requires two lexicographic searches: one for P and the other for $P\#$, where # is a special character larger than any other alphabet character. So $O(p + \log n)$ character comparisons are enough to delimit the range $SA[i, j]$ of suffixes having P as a prefix. It is then easy to count the pattern occurrences in constant time, as $occ = j - i + 1$, or print all of them in $O(occ)$ time. ∎

2.2.2 The LCP-array and its construction$^{\infty}$

Surprisingly enough the longest common prefix array $\texttt{lcp}[1, n-1]$ can be derived from the input string T and its suffix array $SA[1, n]$ in optimal linear time.[1] This time bound cannot be obtained by the simple approach that compares character-by-character the $n-1$ contiguous pairs of text suffixes in SA; as this takes $\Theta(n^2)$ time in the worst case. The optimal $O(n)$ time needs to avoid the re-scanning of the text characters, so some property of the input text has to be proved and deployed in the design of an algorithm that achieves this complexity. This is exactly what Kasai *et al* did in 2001 [8], their algorithm is elegant, deceptively simple, and optimal in time and space.

Sorted Suffixes	SA	SA `positions`
abc`def`...	$j-1$	$p-1$
abc`hi`...	$i-1$	p
.	.	.
.	.	.
.	.	.
bc`def`...	j	
.	.	.
.	.	.
.	.	.
bc`h`...	k	$q-1$
bc`hi`...	i	q

FIGURE 2.3: Relation between suffixes and `lcp` values in the Kasai's algorithm. Suffixes are shown only with their starting characters, the rest is indicated with ... for simplicity.

For the sake of presentation we will refer to Figure 2.3 which illustrates clearly the main algorithmic idea. Let us concentrate on two consecutive suffixes in the text T, say $suff_{i-1}$ and $suff_i$, which occur at positions p and q in the suffix array SA. And assume that we know inductively the value of $\texttt{lcp}[p-1]$, storing the longest common prefix between $SA[p-1] = suff_{j-1}$ and the next suffix $SA[p] = suff_{i-1}$ in the lexicographic order. Our goal is to show that $\texttt{lcp}[q-1]$ storing the longest common prefix between suffix $SA[q-1] = suff_k$ and the next ordered suffix $SA[q] = suff_i$, which interests us, can be computed without re-scanning these suffixes from their first character but can start where the comparison between $SA[p-1]$ and $SA[p]$ ended. This will ensure that re-scanning of text characters is avoided, precisely it is avoided the re-scanning of $suff_{i-1}$, and as a result we will get a linear time complexity.

We need the following property that we already mentioned when dealing with prefix search, and that we restate here in the context of suffix arrays.

FACT 2.1 *For any position $x < y$ it holds* $\texttt{lcp}(suff_{SA[y-1]}, suff_{SA[y]}) \geq \texttt{lcp}(suff_{SA[x]}, suff_{SA[y]})$.

Proof This property derives from the observation that suffixes in SA are ordered lexicographically, so that, as we go farther from $SA[y]$ we reduce the length of the shared prefix. ∎

[1] Recall that $\texttt{lcp}[i] = \texttt{lcp}(suff_{SA[i]}, suff_{SA[i+1]})$ for $i < n$.

Let us now refer to Figure 2.3, concentrate on the pair of suffixes $suff_{j-1}$ and $suff_{i-1}$, and take their next suffixes $suff_j$ and $suff_i$ in T. There are two possible cases: Either they share some characters in their prefix, i.e. $\texttt{lcp}[p-1] > 0$, or they do not. In the former case we can conclude that, since lexicographically $suff_{j-1} < suff_{i-1}$, the next suffixes preserve that lexicographic order, so $suff_j < suff_i$ and moreover $\texttt{lcp}(suff_j, suff_i) = \texttt{lcp}[p-1] - 1$. In fact, the first shared character is dropped, given the step ahead from $j-1$ (resp. $i-1$) to j (resp. i) in the starting positions of the suffixes, but the next $\texttt{lcp}[p-1] - 1$ shared characters (possibly none) remain, as well as remain their mismatch characters that drives the lexicographic order. In the Figure above, we have $\texttt{lcp}[p-1] = 3$ and the shared prefix is `abc`, so when we consider the next suffixes their lcp is `bc` of length 2, their order is preserved (as indeed $suff_j$ occurs before $suff_i$), and now they lie not adjacent in SA.

FACT 2.2 *If* $\texttt{lcp}(suff_{SA[y-1]}, suff_{SA[y]}) > 0$ *then:*

$$\texttt{lcp}(suff_{SA[y-1]+1}, suff_{SA[y]+1}) = \texttt{lcp}(suff_{SA[y-1]}, suff_{SA[y]}) - 1$$

By Fact 2.1 and Fact 2.2, we can conclude the key property deployed by Kasai's algorithm: $\texttt{lcp}[q-1] \geq \max\{\texttt{lcp}[p-1]-1, 0\}$. This algorithmically shows that the computation of $\texttt{lcp}[q-1]$ can take full advantage of what we compared for the computation of $\texttt{lcp}[p-1]$. By adding to this the fact that we are processing the text suffixes rightward, we can conclude that the characters involved in the suffix comparisons move themselves rightward and, since re-scanning is avoided, their total number is $O(n)$. A sketch of the Kasai's algorithm is shown in Figure 2.2, where we make use of the inverse suffix array, denoted by SA^{-1}, which returns for every suffix its position in SA. Referring to Figure 2.3, we have that $SA^{-1}[i] = p$.

Algorithm 2.2 LCP-Build(char *T, int n, char **SA)

```
1:  h = 0;
2:  for (i = 1; i ≤ n, i++) do
3:      q = SA^{-1}[i];
4:      if (q > 1) then
5:          k = SA[q − 1];
6:          if (h > 0) then
7:              h − −;
8:          end if
9:          while (T[k + h] == T[i + h]) do
10:             h++;
11:         end while
12:         lcp[q − 1] = h;
13:     end if
14: end for
```

Step 4 checks whether $suff_q$ occupies the first position of the suffix array, in which case the `lcp` with the previous suffix is undefined. The **for**-loop then scans the text suffixes $suff_i$ from left to right, and for each of them it first retrieves the position of $suff_i$ in SA, namely $i = SA[q]$, and its preceding suffix in SA, namely $k = SA[q-1]$. Then it extends their longest common prefix starting from the offset h determined for $suff_{i-1}$ via character-by-character comparison. This is the algorithmic application of the above observations.

As far as the time complexity is concerned, we notice that h is decreased at most n times (once per iteration of the for-loop), and it cannot move outside T (within each iteration of the for-loop),

so $h \leq n$. This implies that h can be increased at most $2n$ times and this is the upper bound to the number of character comparisons executed by the Kasai's algorithm. The total time complexity is therefore $O(n)$.

We conclude this section by noticing that an I/O-efficient algorithm to compute the lcp-array is still missing in the literature, some heuristics are known to reduce the number of I/Os incurred by the above computation but an optimal $O(n/B)$ I/O-bound is yet to come, if possible.

2.2.3 Suffix-array construction

Given that the suffix array is a sorted sequence of items, the most intuitive way to construct *SA* is to use an efficient comparison-based sorting algorithm and specialize the comparison-function in such a way that it computes the lexicographic order between strings. `Algorithm 2.3` implements this idea in C-style using the built-in procedure QSORT as sorter and a properly-defined subroutine Suffix_cmp for comparing suffixes:

Suffix_cmp(char $**$p, char $**$q){ `return` strcmp($*$p, $*$q) };

Notice that the suffix array is initialized with the pointers to the real starting positions in memory of the suffixes to be sorted, and not the integer offsets from 1 to n as stated in the formal description of *SA* of the previous pages. The reason is that in this way Suffix_cmp does not need to know T's position in memory (which would have needed a global parameter) because its actual parameters passed during an invocation provide the starting positions in memory of the suffixes to be compared. Moreover, the suffix array *SA* has indexes starting from 0 as it is typical of C-language.

Algorithm 2.3 COMPARISON_BASED_CONSTRUCTION(char *T, int n, char **SA)

```
1: for (i = 0; i < n; i ++) do
2:       SA[i] = T + i;
3: end for
4: QSORT(SA, n, sizeof(char *), Suffix_cmp);
```

A major drawback of this simple approach is that it is not I/O-efficient for two main reasons: the optimal number $O(n \log n)$ of comparisons involves now variable-length strings which may consists of up to $\Theta(n)$ characters; locality in *SA* does not translate into locality in suffix comparisons because of the fact that sorting permutes the string pointers rather than their pointed strings. Both these issues elicit I/Os, and turn this simple algorithm into a slow one.

THEOREM 2.1 *In the worst case the use of a comparison-based sorter to construct the suffix array of a given string $T[1,n]$ requires $O((\frac{n}{B})n \log n)$ I/Os, and $O(n \log n)$ bits of working space.*

In Section 2.2.3 we describe a Divide-and-Conquer algorithm— the *Skew* algorithm proposed by Kärkkäinen and Sanders [7]— which is elegant, easy to code, and flexible enough to achieve the optimal I/O-bound in various models of computations. In Section 2.2.3 we describe another algorithm— the *Scan-based* algorithm proposed by BaezaYates, Gonnet and Sniders [6]— which is also simple, but incurs in a larger number of I/Os; we nonetheless introduce this algorithm because it offers the positive feature of processing the input data in passes (streaming-like) thus forces pre-fetching, allows compression and hence it turns to be suitable for slow disks.

The Skew Algorithm

In 2003 Kärkkäinen and Sanders [7] showed that the problem of constructing suffix-arrays can be *reduced* to the problem of sorting a set of triplets whose components are integers in the range $[1, O(n)]$. Surprisingly this reduction takes *linear time and space* thus turning the complexity of suffix-array construction into the complexity of sorting atomic items, a problem about which we discussed deeply in the previous chapters and for which we know optimal algorithms for hierarchical memories and multiple disks. More than this, since the items to be sorted are integers bounded in value by $O(n)$, the sorting of the triplets takes $O(n)$ time in the RAM model, so this is the optimal time complexity of suffix-array construction in RAM. Really impressive!

This algorithm is named *Skew* in the literature, and it works in every model of computation for which an efficient sorting primitive is available: disk, distributed, parallel. The algorithm hinges on a divide&conquer approach that executes a $\frac{2}{3} : \frac{1}{3}$ split, crucial to make the final merge-step easy to implement. Previous approaches used the more natural $\frac{1}{2} : \frac{1}{2}$ split (such as [2]) but were forced to use a more sophisticated merge-step which needed the use of the suffix-tree data structure.

For the sake of presentation we use $T[1, n] = t_1 t_2 \dots t_n$ to denote the input string and we assume that the characters are drawn from an integer alphabet of size $\sigma = O(n)$. Otherwise we can sort the characters of T and rename them with integers in $O(n)$, taking overall $O(n \log \sigma)$ time in the worst-case. So T is a text of integers, taking $\Theta(\log n)$ bits each; this will be the case for all texts created during the suffix-array construction process. Furthermore we assume that $t_n = \$$, a special symbol smaller than any other alphabet character, and logically pad T with an infinite number of occurrences of \$.

Given this notation, we can sketch the three main steps of the Skew algorithm:

Step 1. Construct the suffix array $SA^{2,0}$ limited to the suffixes starting at positions $P_{2,0} = \{i : i \bmod 3 = 2, \text{ or } i \bmod 3 = 0\}$:

- Build a special string $T^{2,0}$ of length $(2/3)n$ which compactly encodes all suffixes of T starting at positions $P_{2,0}$.
- Build recursively the suffix-array SA' of $T^{2,0}$.
- Derive the suffix-array $SA^{2,0}$ from SA'.

Step 2 Construct the suffix array SA^1 of the remaining suffixes starting at positions $P_1 = \{i : i \bmod 3 = 1\}$:

- For every $i \in P_1$, represent suffix $T[i, n]$ with a pair $\langle T[i], \mathtt{pos}(i+1)\rangle$, where it is $i + 1 \in P_{2,0}$.
- Assume to have pre-computed the array $\mathtt{pos}[i+1]$ which provides the position of the $(i+1)$-th text suffix $T[i+1, n]$ in $SA^{2,0}$.
- Radix-sort the above $O(n)$ pairs.

Step 3. Merge the two suffix arrays into one:

- This is done by deploying the decomposition $\frac{2}{3} : \frac{1}{3}$ which ensures a constant-time lexicographic comparison between any pair of suffixes (see details below).

The execution of the algorithm is illustrated over the input string $T[1, 12] =$"mississippi\$" whose suffix array is $SA = (12, 11, 8, 5, 2, 1, 10, 9, 7, 4, 6, 3)$. In this example we have: $P_{2,0} = \{2, 3, 5, 6, 8, 9, 11, 12\}$ and $P_1 = \{1, 4, 7, 10\}$.

Step 1. The first step is the most involved one and constitutes the backbone of the entire recursive process. It lexicographically sorts the suffixes starting at the text positions $P_{2,0}$. The resulting

array is denoted by $SA^{2,0}$ and represents a *sampled* version of the final suffix array SA because it is restricted to the suffixes starting at positions $P_{2,0}$.

To efficiently obtain $SA^{2,0}$, we reduce the problem to the construction of the suffix array for a string $T^{2,0}$ of length about $\frac{2n}{3}$. This text consists of "characters" which are integers smaller than $\approx \frac{2n}{3}$. Since we are *again* in the presence of a text of integers, of length proportionally smaller than n, we can construct its suffix array by invoking *recursively* the construction procedure.

The key difficulty is how to define $T^{2,0}$ so that its suffix array may be used to derive easily $SA^{2,0}$, namely the sorted sequence of text suffixes starting at positions in $P_{2,0}$. The elegant solution consists of considering the two text suffixes $T[2,n]$ and $T[3,n]$, pad them with the special symbol \$ in order to have multiple-of-three length, and then decompose the resulting strings into triplets of characters $T[2,\cdot] = [t_2,t_3,t_4][t_5,t_6,t_7][t_8,t_9,t_{10}]\ldots$ and $T[3,\cdot] = [t_3,t_4,t_5][t_6,t_7,t_8][t_9,t_{10},t_{11}]\ldots$. The dot expresses the fact that we are considering the smallest integer, larger than n, that allows those strings to have length which is a multiple of three.
With reference to the previous example, we have:

$$T[2,\cdot] = \underset{2}{[\texttt{i s s}]}\,\underset{5}{[\texttt{i s s}]}\,\underset{8}{[\texttt{i p p}]}\,\underset{11}{[\texttt{i \$ \$}]} \qquad T[3,\cdot] = \underset{3}{[\texttt{ s s i }]}\,\underset{6}{[\texttt{ s s i }]}\,\underset{9}{[\texttt{ p p i }]}\,\underset{12}{[\texttt{ \$ \$ \$ }]}$$

We then construct the string $R = T[2,\cdot] \bullet T[3,\cdot]$, and thus we obtain:

$$R = \underset{2}{[\texttt{i s s}]}\,\underset{5}{[\texttt{i s s}]}\,\underset{8}{[\texttt{i p p}]}\,\underset{11}{[\texttt{i \$ \$}]}\underset{3}{[\texttt{ s s i }]}\,\underset{6}{[\texttt{ s s i }]}\,\underset{9}{[\texttt{ p p i }]}\,\underset{12}{[\texttt{ \$ \$ \$ }]}$$

The key property on which the first step of the Skew algorithm hinges on, is the following:

Property 2.2 *Every suffix $T[i,n]$ starting at a position $i \in P_{2,0}$, can be put in correspondence with a suffix of R consisting of an integral sequence of triplets. Specifically, if $i \bmod 3 = 0$ then the text suffix coincides exactly with a suffix of R; if $i \bmod 3 = 2$, then the text suffix prefixes a suffix of R which nevertheless terminates with special symbol \$.*

The correctness of this property can be inferred easily by observing that any suffix $T[i,n]$ starting at a position in $P_{2,0}$ is clearly a suffix of either $T[2,\cdot]$ or $T[3,\cdot]$, given that $i > 0$, and $i \bmod 3$ is either 0 or 2. Moreover, since $i \in P_{2,0}$, it has the form $i = 3 + 3k$ or $i = 2 + 3k$, for some $k \geq 0$, and thus $T[i,n]$ occurs within R aligned to the beginning of some triplet.

By the previous running example, take $i = 6 = 0 \bmod 3$, the suffix $T[6,12] = \texttt{ssippi\$}$ occurs at the second triplet of $T[3,\cdot]$, which is the sixth triplet of R. Similarly, take $i = 8 = 2 \bmod 3$, the suffix $T[8,12] = \texttt{ippi\$}$ occurs at the third triplet of $T[2,\cdot]$, which is the third triplet of R. Notice that, even if $T[8,12]$ is not a full suffix of R, we have that $T[8,12]$ ends with two \$s, which will constitute sort of end-delimiters.

The final operation is then to encode those triplets via integers, and thus squeeze R into a string $T^{2,0}$ of $\frac{2n}{3}$ integer-symbols, thus realizing the reduction in length we were aiming for above. This encoding must be implemented in a way that the lexicographic comparison between two triplets can be obtained by comparing those integers. In the literature this is called *lexicographic naming* and can be easily obtained by *radix sorting* the triplets in R and associating to each distinct triplet its *rank* in the lexicographic order. Since we have $O(n)$ triplets, each consisting of symbols in a range $[0,n]$, their radix sort takes $O(n)$ time.

In our example, the sorted triplets are labeled with the following ranks:

[\$ \$ \$]	[i \$ \$]	[i p p]	[i s s]	[i s s]	[p p i]	[s s i]	[s s i]	sorted triplets
0	1	2	3	3	4	5	5	sorted ranks

$R =$ [i s s]	[i s s]	[i p p]	[i \$ \$]	[[s s i]	[s s i]	[p p i]	[\$ \$ \$]	triplets
3	3	2	1	5	5	4	0	$T^{2,0}$ (string of ranks)

As a result of the naming of the triplets in R, we get the new text $T^{2,0} = 33215540$ whose length is $\frac{2n}{3}$. The crucial observation here is that we have a text $T^{2,0}$ which is again a text of integers as T, taking $O(\log n)$ bits per integer (as before), but $T^{2,0}$ has length shorter than T, so that we can invoke recursively the suffix-array construction procedure over it.

It is evident from the discussion above that, since the ranks are assigned in the same order as the lexicographic order of their triplets, the lexicographic comparison between suffixes of R (aligned to the triplets) equals the lexicographic comparison between suffixes of $T^{2,0}$.

Here Property 2.2 comes into play, because it defines a bijection between suffixes of R aligned to triplet beginnings, hence suffixes of $T^{2,0}$, with text suffixes starting in $P_{2,0}$. This correspondence is then deployed to derive $SA^{2,0}$ from the suffix array of $T^{2,0}$.

In our running example $T^{2,0} = 33215540$, the suffix-array construction algorithm is applied recursively thus deriving the suffix-array $(8, 4, 3, 2, 1, 7, 6, 5)$. We can turn this suffix array into $SA^{2,0}$ by turning the positions in $T^{2,0}$ into positions in T. This can be done via simple arithmetic operations, given the layout of the triplets in$T^{2,0}$, and obtains in our running example the suffix array $SA^{2,0} = (12, 11, 8, 5, 2, 9, 6, 3)$.

Before concluding the description of step 1, we add two notes. The first one is that, if all symbols in $T^{2,0}$ are different, then we do not need to recurse because suffixes can be sorted by looking just at their first characters. The second observation is for programmers that should be careful in turning the suffix-positions in $T^{2,0}$ into the suffix positions in T to get the final $SA^{2,0}$, because they must take into account the layout of the triplets of R.

Step 2. Once the suffix array $SA^{2,0}$ has been built (recursively), it is possible to sort lexicographically the remaining suffixes of T, namely the ones starting at the text positions $i \bmod 3 = 1$, in a simple way. We decompose a suffix $T[i, n]$ as composed by its first character $T[i]$ and its remaining suffix $T[i + 1, n]$. Since $i \in P_1$, the next position $i + 1 \in P_{2,0}$, and thus the suffix $T[i + 1, n]$ occurs in $SA^{2,0}$. We can then encode the suffix $T[i, n]$ with a pair of integers $\langle T[i], \mathtt{pos}(i + 1)\rangle$, where $\mathtt{pos}(i+1)$ denotes the lexicographic rank in $SA^{2,0}$ of the suffix $T[i+1, n]$. If $i+1 = n+1$ then we set $\mathtt{pos}(n + 1) = 0$ given that the character \$ is assumed to be smaller than any other alphabet character.

Given this observation, two text suffixes starting at positions in P_1 can then be compared in constant time by comparing their corresponding pairs. Therefore SA^1 can be computed in $O(n)$ time by radix-sorting the $O(n)$ pairs encoding its suffixes.

In our example, this boils down to radix-sort the pairs:

Pairs/suffixes:	$\langle \mathtt{m}, 4\rangle$	$\langle \mathtt{s}, 3\rangle$	$\langle \mathtt{s}, 2\rangle$	$\langle \mathtt{p}, 1\rangle$	
	1	4	7	10	starting positions in P_1
Sorted pairs/suffixes:	$\langle \mathtt{m}, 4\rangle$	$< \langle \mathtt{p}, 1\rangle$	$< \langle \mathtt{s}, 2\rangle$	$< \langle \mathtt{s}, 3\rangle$	
	1	10	7	4	SA^1

Step 3. The final step merges the two sorted arrays SA^1 and $SA^{2,0}$ in linear $O(n)$ time by resorting an interesting observation which motivates the split $\frac{2}{3} : \frac{1}{3}$. Let us take two suffixes $T[i, n] \in SA^1$ and $T[j, n] \in SA^{2,0}$, which we wish to lexicographically compare for implementing the merge-step. They belong to two different suffix arrays so we have no *lexicographic relation* known for them, and we cannot compare them character-by-character because this would incur in a very high cost. We deploy a decomposition idea similar to the one exploited in Step 2 above, which consists of looking at a suffix as composed by *one or two characters* plus the lexicographic rank of its remaining suffix. This decomposition becomes effective if the remaining suffixes of the compared ones lie in the same suffix array, so that their rank is enough to get their order in constant time. Elegantly enough this is possible with the split $\frac{2}{3} : \frac{1}{3}$, but it could not be possible with the split $\frac{1}{2} : \frac{1}{2}$. This observation is implemented as follows:

1. if $j \bmod 3 = 2$ then we compare $T[j,n] = T[j]T[j+1,n]$ against $T[i,n] = T[i]T[i+1,n]$. Both suffixes $T[j+1,n]$ and $T[i+1,n]$ occur in $SA^{2,0}$ (given that their starting positions are congruent 0 or 2 mod 3, respectively), so we can derive the above lexicographic comparison by comparing the pairs $\langle T[i], \mathtt{pos}(i+1)\rangle$ and $\langle T[j], \mathtt{pos}(j+1)\rangle$. This comparison takes $O(1)$ time, provided that the array pos is available.[2]
2. if $j \bmod 3 = 0$ then we compare $T[j,n] = T[j]T[j+1]T[j+2,n]$ against $T[i,n] = T[i]T[i+1]T[i+2,n]$. Both the suffixes $T[j+2,n]$ and $T[i+2,n]$ occur in $SA^{2,0}$ (given that their starting positions are congruent 0 or 2 mod 3, respectively), so we can derive the above lexicographic comparison by comparing the triples $\langle T[i], T[i+1], \mathtt{pos}(i+2)\rangle$ and $\langle T[j], T[j+1], \mathtt{pos}(j+2)\rangle$. This comparison takes $O(1)$ time, provided that the array pos is available.

In our running example we have that $T[8,11] < T[10,11]$, and in fact $\langle i,5\rangle < \langle p,1\rangle$. Also we have that $T[7,11] < T[6,11]$ and in fact $\langle s,i,5\rangle < \langle s,s,2\rangle$. In the following figure we depict all possible pairs of triples which may be involved in a comparison, where ($\star\star$) and ($\star\star\star$) denote the pairs for rule 1 and 2 above, respectively. Conversely ($\star$) denotes the starting position in T of the suffix. Notice that, since we do not know which suffix of $SA^{2,0}$ will be compared with a suffix of SA^1 during the merging process, for each of the latter suffixes we need to compute both representations ($\star\star$) and ($\star\star\star$), hence as a pair and as a triplet.[3]

SA^1				$SA^{2,0}$								
1	10	7	4	12	11	8	5	2	9	6	3	($\star$)
$\langle$m, 4$\rangle$	$\langle$p, 1$\rangle$	$\langle$s, 2$\rangle$	$\langle$s, 3$\rangle$		$\langle$i, 0$\rangle$	$\langle$i, 5$\rangle$	$\langle$i, 6$\rangle$	$\langle$i, 7$\rangle$				($\star\star$)
$\langle$m, i, 7$\rangle$	$\langle$p, i, 0$\rangle$	$\langle$s, i, 5$\rangle$	$\langle$s, i, 6$\rangle$	$\langle$\$, \$, $-1\rangle$					$\langle$p, p, 1$\rangle$	$\langle$s, s, 2$\rangle$	$\langle$s, s, 3$\rangle$	($\star\star\star$)

At the end of the merge step we obtain the final suffix array: $SA = (12, 11, 8, 5, 2, 1, 10, 9, 7, 4, 6, 3)$.

From the discussion above it is clear that every step can be implemented via the *sorting* or the scanning of a set of n atomic items, which are possibly triplets of integers, taking each triplet $O(\log n)$ bits, so one memory word. Therefore the proposed method can be seen as a *algorithmic reduction* of the suffix-array construction problem to the classic problem of sorting n-items. This problem has been solved optimally in several models of computation, for the case of the two-level memory model see Chapter **??**.

For what concerns the RAM model, the time complexity of the Skew algorithm can be modeled by the recurrence $T(n) = T(\frac{2n}{3}) + O(n)$, because Steps 2 and 3 cost $O(n)$ and the recursive call is executed over the string $T^{2,0}$ whose length is $(2/3)n$. This recurrence has solution $T(n) = O(n)$, which is clearly optimal. For what concerns the two-level memory model, the Skew algorithm can be implemented in $O(\frac{n}{B}\log_{M/B}\frac{n}{M})$ I/Os, that is the I/O-complexity of sorting n atomic items.

THEOREM 2.3 *The Skew algorithm builds the suffix array of a text string $T[1,n]$ in $O(Sort(n))$ I/Os and $O(n/B)$ disk pages. If the alphabet Σ has size polynomial in n, the CPU time is $O(n)$.*

The Scan-based Algorithm$^\infty$

Before the introduction of the Skew algorithm, the best known disk-based algorithm was the one proposed by Baeza-Yates, Gonnet and Sniders in 1992 [6]. It is also a divide&conquer algorithm

[2] Of course, the array pos can be derived from $SA^{2,0}$ in linear time, since it is its inverse.

[3] Recall that $\mathtt{pos}(n) = 0$, and for the sake of the lexicographic order, we can set $\mathtt{pos}(j) = -1$, for all $j > n$.

whose divide step is strongly unbalanced, thus it executes a quadratic number of suffix comparisons which induce a *cubic* time complexity. Nevertheless the algorithm is fast in practice because it processes the data into passes thus deploying the high throughput of modern disks.

Let $\ell < 1$ be a positive constant, properly fixed to build the suffix array of a text piece of $m = \ell M$ characters in internal memory. Then assume that the text $T[1, n]$ is logically divided into pieces of m characters each, numbered rightward: namely $T = T_1 T_2 \cdots T_{n/m}$ where $T_h = T[hm + 1, (h + 1)m]$ for $h = 0, 1, \ldots$. The algorithm computes *incrementally* the suffix array of T in $\Theta(n/M)$ stages, rather than the logarithmic number of stages of the Skew algorithm. At the beginning of stage h, we assume to have on disk *the array SA^h that contains the sorted sequence of the first hm suffixes of T*. Initially $h = 0$ and thus SA^0 is the empty array. In the generic h-th stage, the algorithm loads the next text piece T^{h+1} in internal memory, builds SA' as the sorted sequence of suffixes starting in T^{h+1}, and then computes the new SA^{h+1} by merging the two sorted sequences SA^h and SA'.

There are two main issues when detailing this algorithmic idea in a running code: how to efficiently construct SA', since its suffixes start in T^{h+1} but may extend outside that block of characters up to the end of T; and how to efficiently merge the two sorted sequences SA^h and SA', since they involve suffixes whose length may be up to $\Theta(n)$ characters. For the first issue the algorithm does not implement any special trick, it just compares pairs of suffixes character-by-character in $O(n)$ time and $O(n/B)$ I/Os. This means that over the total execution of the $O(n/M)$ stages, the algorithm takes $O(\frac{n}{B}\frac{n}{m} m \log m) = O(\frac{n^2}{B} \log m)$ I/Os to construct SA'.

For the second issue, we note that the merge between SA' with SA^h is executed in a smart way by resorting the use of an auxiliary array $C[1, m + 1]$ which counts in $C[j]$ the number of suffixes of SA^h that are lexicographically greater than the $SA'[j-1]$-th text suffix and smaller than the $SA'[j]$-th text suffix. Two special cases occur if $j = 1, m + 1$: in the former case we assume that $SA'[0]$ is the empty suffix, in the latter case we assume that $SA'[m + 1]$ is a special suffix larger than any string. Since SA^h is longer and longer, we process it streaming-like by devising a method that scans rightward the text T (from its beginning) and then searches each of its suffixes by binary-search in SA'. If the lexicographic position of the searched suffix is j, then the entry $C[j]$ is incremented. The binary search may involve a part of a suffix which lies outside the block T^{h+1} currently in internal memory, thus taking $O(n/B)$ I/Os per binary-search step. Over all the n/M stages, this binary search takes $O(\sum_{h=0}^{n/m-1} \frac{n}{B}(hm) \log m) = O(\frac{n^3}{MB} \log M)$ I/Os.

Array C is then exploited in the next substep to quickly merge the two arrays SA' (residing in internal memory) and SA^h (residing on disk): $C[j]$ indicates how many consecutive suffixes of SA^h lexicographically lie after $SA'[j-1]$ and before $SA'[j]$. Hence a disk scan of SA^h suffices to perform the merging process in $O(n/B)$ I/Os.

THEOREM 2.4 *The Scan-based algorithm builds the suffix array of a text string $T[1, n]$ in $O(\frac{n^3}{MB} \log M)$ I/Os and $O(n/B)$ disk pages.*

Since the worst-case number of total I/Os is cubic, a purely theoretical analysis would classify this algorithm as not interesting. However, in practical situations it is very reasonable to assume that each suffix comparison finds in internal memory all the characters used to compare the two involved suffixes. And indeed the practical behavior of this algorithm is better described by the formula $O(\frac{n^2}{MB})$ I/Os. Additionally, all I/Os in this analysis are sequential and the actual number of random seeks is only $O(n/M)$ (i.e., at most a constant number per stage). Consequently, the algorithm takes fully advantage of the large bandwidth of modern disks and of the high speed of current CPUs. As a final notice we remark that the suffix arrays SA^h and the text T are scanned sequentially, so some form of compression can be adopted to reduce the I/O-volume and thus further speed-up the underlying algorithm.

Before detailing a significant improvement to the previous approach, let us concentrate on the same running example used in the previous section to sketch the Skew algorithm.

	1	2	3	4	5	6	7	8	9	10	11	12
$T[1,12] =$	m	i	s	s	i	s	s	i	p	p	i	$

Suppose that $m = 3$ and that, at the beginning of stage $h = 1$, the algorithm has already processed the text block $T^0 = T[1,3] =$ `mis` and thus stored on disk the array $SA^1 = (2,1,3)$ which corresponds to the lexicographic order of the text suffixes which start in that block: namely, `mississippi$`, `ississippi$` and `ssissippi$`. During the stage $h = 1$, the algorithm loads in internal memory the next block $T^1 = T[4,6] =$ `sis` and lexicographically sorts the text suffixes which start in positions $[4,6]$ and extend to the end of T, see figure 2.4.

Text suffixes	`sissippi$`	`issippi$`	`ssippi$`
		⇓	
		Lexicographic ordering	
		⇓	
Sorted suffixes	`issippi$`	`sissippi$`	`ssippi$`
SA′	5	4	6

FIGURE 2.4: Stage 1, step 1, of the Scan-based algorithm.

The figure shows that the comparison between the text suffixes: $T[4,12] =$ `sissippi$` and $T[6,12] =$ `ssippi$` involves characters that lie outside the text piece $T[4,6]$ loaded in internal memory, so that their comparison induces some I/Os.

The final step merges $SA^1 = (2,1,3)$ with $SA' = (5,4,6)$, in order to compute SA^2. This step uses the information of the counter array C. In this specific running example, see Figure 2.5, it is $C[1] = 2$ because two suffixes $T[1,12] =$ `mississippi$` and $T[2,12] =$ `ississippi$` are between the $SA'[0]$-th suffix `issippi$` and the $SA'[1]$-th suffix `sissippi$`.

Suffix Arrays	$\underbrace{SA' = [5,4,6] \quad SA^1 = [\mathbf{2},\mathbf{1},\mathbf{3}]}$
Merge via C	$\Downarrow C=[0,2,0,1]$
	$SA^2 = [5,\mathbf{2},\mathbf{1},4,6,\mathbf{3}]$

FIGURE 2.5: Stage 1, step 3, of the Scan-based algorithm

The second stage is summarized in Figure 2.6 where the text substring $T^2 = T[7,9] =$ `sip` is loaded in memory and the suffix array SA' for the suffixes starting at positions $[7,9]$ is built. Then, the suffix array SA' is merged with the suffix array SA^2 residing on disk and containing the suffixes which start in $T[1,6]$.

The third and last stage is summarized in Figure 2.7 where the substring $T^3 = T[10,12] =$ `pi$` is loaded in memory and the suffix array SA' for the suffixes starting at positions $[10,12]$ is built.

Stage 2*:*

(1) Load into internal memory $T^2 = T[7, 9]$ = `sip`.

(2) Build SA' for the suffixes starting in $[7, 9]$:

Text suffixes	`sippi$`	`ippi$`	`ppi$`
		⇓	
		Lexicographic ordering	
		⇓	
Sorted suffixes	`ippi$`	`ppi$`	`sippi$`
SA′	8	9	7

(3) Merge SA' with SA^2 exploiting C:

Suffix Arrays	$\underbrace{SA' = [8, 9, 7] \quad SA^2 = [\mathbf{5}, \mathbf{2}, \mathbf{1}, \mathbf{4}, \mathbf{6}, \mathbf{3}]}$
Merge via C	$\Downarrow C=[0,3,0,3]$
	$SA^3 = [8, \mathbf{5}, \mathbf{2}, \mathbf{1}, 9, 7, \mathbf{4}, \mathbf{6}, \mathbf{3}]$

FIGURE 2.6: Stage 2 of the Scan-based algorithm.

Then, the suffix array SA' is merged with the suffix array on disk SA^3 containing the suffixes which start in $T[1, 9]$.

The performance of this algorithm can be improved via a simple observation [4]. Assume that, at the beginning of stage h, in addition to the SA^h we have on disk a bit array, called gt_h, such that $\mathsf{gt_h}[\mathrm{i}] = 1$ if and only if the suffix $T[(hm + 1) + i, n]$ is Greater Than the suffix $T[(hm + 1), n]$. The computation of gt can occur efficiently, but this technicality is left to the original paper [4] and not detailed here.

During the h-th stage the algorithm loads into internal memory the substring $t[1, 2m] = T^h T^{h+1}$ (so this is double in size with respect to the previous proposal) and the binary array $\mathsf{gt}_{h+1}[1, m - 1]$ (so it refers to the second block of text loaded in internal memory). The key observation is that we can build SA' by deploying the two arrays above without performing any I/Os, other than the ones needed to load $t[1, 2m]$ and $\mathsf{gt}_{h+1}[1, m - 1]$. This seems surprising, but it descends from the fact that any two text suffixes starting at positions i and j within T^h, with $i < j$, can be compared lexicographically by looking first at their characters in the substring t, namely at the strings $t[i, m]$ and $t[j, j + m - i]$. These two strings have the same length and are completely in $t[1, 2m]$, hence in internal memory. If these strings differ, their order is determined and we are done; otherwise, the order between these two suffixes is determined by the order of the remaining suffixes starting at the characters $t[m + 1]$ and $t[j + m - i + 1]$. This order is given by the bit stored in $\mathsf{gt}_{h+1}[j - i]$, also available in internal memory.

This argument shows that the two arrays t and gt_{h+1} contain all the information we need to build SA^{h+1} working in internal memory, and thus without performing any I/Os.

THEOREM 2.5 *The new variant of the Scan-based algorithm builds the suffix array of a string* $T[1, n]$ *in* $O(\frac{n^2}{MB})$ *I/Os and* $O(n/B)$ *disk pages.*

As an example consider stage $h = 1$ and thus load in memory the text substring $t = T^h T^{h+1} =$

Stage 3*:*

(1) Load into internal memory $T^3 = T[10, 12] = pi\$$.

(2) Build SA' for the suffixes starting in $[10, 12]$:

Text suffixes	`pi$`	`i$`	`$`
	⇓		
	Lexicographic ordering		
	⇓		
Sorted suffixes	`$`	`i$`	`pi$`
SA′	12	11	10

(3) Merge SA' with SA^3 exploiting C:

Suffix Arrays	$\underbrace{SA' = [12, 11, 10] \quad SA^3 = [\mathbf{8}, \mathbf{5}, \mathbf{2}, \mathbf{1}, \mathbf{9}, \mathbf{7}, \mathbf{4}, \mathbf{6}, \mathbf{3}]}$
Merge via C	$\Downarrow C=[0,0,4,5]$
	$SA^4 = [12, 11, \mathbf{8}, \mathbf{5}, \mathbf{2}, \mathbf{1}, 10, \mathbf{9}, \mathbf{7}, \mathbf{4}, \mathbf{6}, \mathbf{3}]$

FIGURE 2.7: Stage 3 of the Scan-based algorithm.

$T[4, 9] =$ `sis sip` and the array $\mathsf{gt}_2 = (1, 0)$. Now consider the positions $i = 1$ and $j = 3$ in t, we can compare the text suffixes starting at these positions by first taking the substrings $t[1, 3] = T[4, 6] =$ `sis` with $t[3, 5] = T[6, 9]$ `ssi`. The strings are different so we obtain their order without accessing the disk. Now consider the positions $i = 3$ and $j = 4$ in t, they would not be taken into account by the algorithm since the block has size 3, but let us consider them for the sake of explanation. We can compare the text suffixes starting at these positions by first taking the substrings $t[3, 3] =$ `s` with $t[4, 4] =$ `s`. The strings are not different so we use $\mathsf{gt}_2[j - i] = \mathsf{gt}_2[1] = 1$, hence the remaining suffix $T[4, n]$ is lexicographically greater than $T[5, n]$ and this can be determined again without any I/Os.

2.3 The Suffix Tree

The *suffix tree* is a fundamental data structure used in many algorithms processing strings [5]. In its essence it is a compacted trie that stores all suffixes of an input string, each suffix is represented by a (unique) path from the root of the trie to one of its leaves. We already discussed compacted tries in the previous chapter, now we specialize the description in the context of suffix trees and point out some issues, and their efficient solutions, that arise when the dictionary of indexed strings is composed by suffixes of one single string.

Let us denote the suffix tree built over an input string $T[1, n]$ as ST_T (or just ST when the input is clear from the context) and assume, as done for suffix arrays, that the last character of T is the special symbol \$ which is smaller than any other alphabet character. The suffix tree has the following properties:

1. Each suffix of T is represented by a *unique* path descending from root of ST to one of its leaves. So there are n leaves, one per text suffix, and each leaf is labeled with the starting position in T of its corresponding suffix.

2. Each internal node of ST has at least two outgoing edges. So there are less than n internal nodes and less than $2n - 1$ edges. Every internal node u spells out a text substring, denoted by $s[u]$, which prefixes everyone of the suffixes descending from u in the suffix tree. Typically the value $|s[u]|$ is stored as satellite information of node u, and we use $occ[u]$ to indicate the number of leaves descending from u.
3. The edge labels are non empty substrings of T. The labels of the edges spurring from any internal node start with different characters, called *branching characters*. Edges are assumed to be ordered alphabetically according to their branching characters. So every node has at most σ outgoing edges.[4]

In Figure 2.8 we show the suffix tree built over our exemplar text $T[1, 12]$ = `mississippi$`. The presence of the special symbol $T[12]$ = \$ ensures that no suffix is a prefix of another suffix of T and thus every pair of suffixes differs in some character. So the paths from the root to the leaves of two different suffixes coincide up to their common longest prefix, which ends up in an internal node of ST.

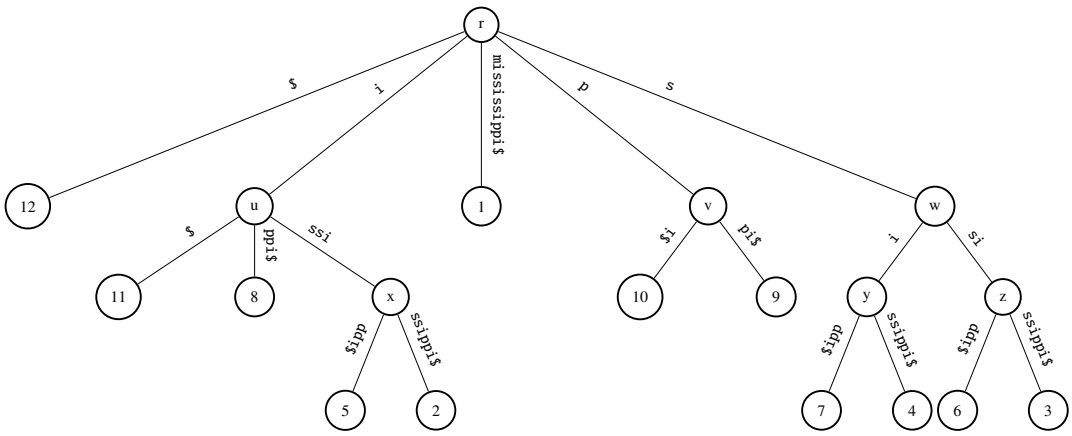


FIGURE 2.8: The suffix tree of the string `mississippi$`

It is evident that we cannot store explicitly the substrings labeling the edges because this would end up in a total space complexity of $\Theta(n^2)$. You can convince yourself by building the suffix tree for the string consisting of all distinct characters, and observe that the suffix tree consists of one root connected to n leaves with edges representing all suffixes. We can circumvent this space explosion by encoding the edge labels with pairs of integers which represent the starting position of the labeling substring and its length. With reference to Figure 2.8 we have that the label of the edge leading to leaf 5, namely the substring $T[9, 12]$ = `ppi$`, can be encoded with the integer pair $\langle 9, 4\rangle$, where 9 is the offset in T and 4 is the length of the label. Other obvious encodings could be possible — say the pair $\langle 9, 12\rangle$ indicating the starting and ending position of the label—, but we will not detail them here. Anyway, whichever is the edge encoding adopted, it uses $O(1)$ space, and thus the storage of all edge labels takes $O(n)$ space, independently of the indexed string.

[4]The special character \$ is included in the alphabet Σ.

FACT 2.3 *The suffix tree of a string $T[1,n]$ consists of n leaves, at most $n-1$ internal nodes and at most $2n-2$ edges. Its space occupancy is $O(n)$, provided that a proper edge-label encoding is adopted.*

As a final notation, we call *locus* of a text substring t the node v whose spelled string is exactly t, hence $s[v] = t$. We call *extended locus* of t' the locus of its shortest extension that has defined locus in ST. In other words, the path spelling the string t' in ST ends within an edge label, say the label of the edge (u, v). This way $s[u]$ prefixes t' which in turn prefixes $s[v]$. Therefore v is the extended locus of t'. Of course if t' has a locus in ST then this coincides with its extended locus. As an example, the node z of the suffix tree in Figure 2.8 is the locus of the substring `ssi` and the extended locus of the substring `ss`.

There are few important properties that the suffix-tree data structure satisfies, they pervade most algorithms which hinge on this powerful data structure. We summarize few of them:

Property 2.6 *Let α be a substring of the text T, then there exists an internal node u such that $s[u] = \alpha$ (hence u is the locus of α) iff they do exist at least two occurrences of α in T followed by distinct characters.*

As an example, take node x in Figure 2.8, the substring $s[x] =$ `issi` occurs twice in T at positions 2 and 5, followed by characters `i` and `p`, respectively.

Property 2.7 *Let α be a substring of the text T that has extended locus in the suffix tree. Then every occurrence of α is followed by the same character in T.*

As an example, take the substring `iss` that has node x as extended locus in Figure 2.8. This substring occurs twice in T at positions 2 and 5, followed always by character `i`.

Property 2.8 *Every internal node u spells out a substring $s[u]$ of T which occurs at the positions $occ[u]$ and is* maximal*, in the sense that it cannot be extended by one character and yet occur at these positions.*

Now we introduce the notion of *lowest common ancestor* (shortly, `lca`) in trees, which is defined for every pair of leaves and denotes the deepest node being ancestor of both leaves in input. As an example in Figure 2.8, we have that u is the `lca` of leaf 8 and 2. Now we turn `lca` between leaves into `lcp` between their corresponding suffixes.

Property 2.9 *Given two suffixes $T[i,n]$ and $T[j,n]$, say ℓ is the length of the longest common prefix between them. This value can be identified by computing the lowest common ancestor $a(i, j)$ between the leaves in the suffix tree corresponding to those two suffixes. Therefore, we have $s[a(i, j)] =$ `lcp`$(T[i,n], T[j,n])$.*

As an example, take the suffixes $T[11, 12] =$ `i$` and $T[5, 12] =$ `issippi$`, their `lcp` is the single character `i` and the `lca` between their leaves is the node u, which indeed spells out the string $s[u] =$ `i`.

2.3.1 The substring-search problem

The search for a pattern $P[1, p]$ as a substring of the text $T[1, n]$, with the help of the suffix tree ST, consists of a tree traversal which starts from its root and proceeds downward as pattern characters are matched against characters labeling the tree edges (see Figure 2.9). Note that, since the first character of the edges outgoing from each traversed node is distinct, the matching of P can follow only one downward path. If the traversal determines a mismatch character, the pattern P does not

occur in T; otherwise the pattern is fully matched, the extended locus of P is found, and all leaves of ST descending from this node identify all text suffixes which are prefixed by P. The text positions associated to these descending leaves are the positions of the occ occurrences of the pattern P in T. These positions can be retrieved in $O(occ)$ time by visiting the subtree that descends from the extended locus of P. In fact this subtree has size $O(occ)$ because its internal nodes have (at least) binary fan-out and consists of occ leaves.

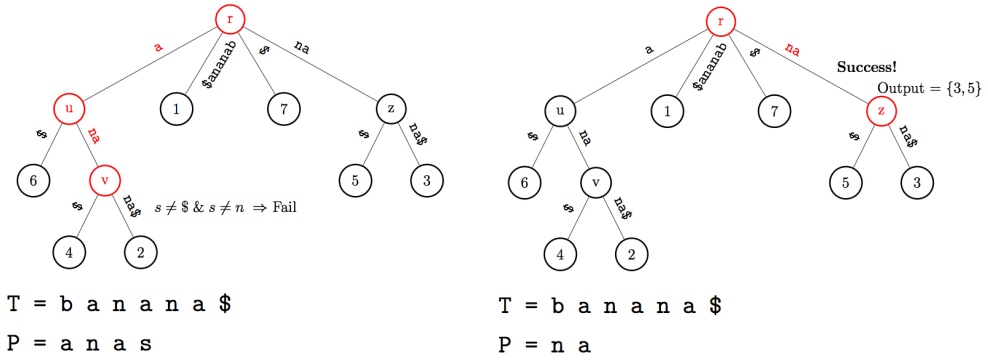


FIGURE 2.9: Two examples of substring searches over the suffix tree built for the text `banana$`. The search for the pattern P = `anas` fails, the other for the pattern P = `na` is successful.

In the running example of Figure 2.9, the pattern P = `na` occurs twice in T and in fact the traversal of ST fully matches P and stops at the node z, from which descend two leaves labeled 3 and 5. And indeed the pattern P occurs at positions 3 and 5 of T, since it prefixes the two suffixes $T[3,12]$ and $T[5,12]$. The cost of pattern searching is $O(pt_\sigma + occ)$ time in the worst case, where t_σ is the time to branch out of a node during the tree traversal. This cost depends on the alphabet size σ and the kind of data structure used to store the branching characters of the edges spurring from each node. We discussed this issue in the previous Chapter, when solving the prefix-search problem via compacted tries. There we observed that $t_\sigma = O(1)$ if we use a perfect-hash table indexed by the branching characters; it is $t_\sigma = O(\log \sigma)$ if we use a plain array and the branching is implemented by a binary search. In both cases the space occupancy is optimal, in that it is linear in the number of branching edges, and thus $O(n)$ overall.

FACT 2.4 *The occ occurrences of a pattern $P[1,p]$ in a text $T[1,n]$ can be found in $O(p + occ)$ time and $O(n)$ space by using a suffix tree built on the input text T, in which the branching characters at each node are indexed via a perfect hash table.*

2.3.2 Construction from Suffix Arrays and vice versa

It is not difficult to observe that the suffix array SA of the text T can be obtained from its suffix tree ST by performing an in-order visit: each time a leaf is encountered, the suffix-index stored in this leaf is written into the suffix array SA; each time an internal node u is encountered, its associated value is written into the array `lcp`.

FACT 2.5 *Given the suffix tree of a string $T[1,n]$, we can derive in $O(n)$ time and space the corresponding suffix array SA and the longest-common-prefix array* `lcp`.

Vice versa, we can derive the suffix tree ST from the two arrays SA and `lcp` in $O(n)$ time as follows. The algorithm constructs incrementally ST starting from a tree, say ST_1, that contains a root node denoting the empty string and one leaf labeled $SA[1]$, denoting the smallest suffix of T. At step $i > 1$, we have inductively constructed the partial suffix tree ST_{i-1} which contains all the $(i-1)$-smallest suffixes of T, hence the suffixes in $SA[1, i-1]$. During step i, the algorithm inserts in ST_{i-1} the i-th smallest suffix $SA[i]$. This requires the addition of one leaf labeled $SA[i]$ and, as we will prove next, at most one single internal node which becomes the father of the inserted leaf. After n steps, the final tree ST_n will be the suffix tree of the string $T[1,n]$.

The key issue here is to show how to insert the leaf $SA[i]$ into ST_{i-1} in constant amortized time. This will be enough to ensure a total time complexity of $O(n)$ for the overall construction process. The main difficulty consists in the detection of the node u father of the leaf $SA[i]$. This node u may already exist in ST_{i-1}, in this case $SA[i]$ is attached to u; otherwise, u must be created by splitting an edge of ST_{i-1}. Whether u exists or not is discovered by percolating ST_{i-1} upward (and not downward!), starting from the leaf $SA[i-1]$, which is the rightmost one in ST_{i-1} because of the lexicographic order, and stopping when a node x is reached such that $\texttt{lcp}[i] \leq |s[x]|$. Recall that $\texttt{lcp}[i]$ is the number of characters that the text suffix $suff_{SA[i-1]}$ shares with next suffix $suff_{SA[i]}$ in the lexicographic order. The leaves corresponding to these two suffixes are of course consecutive in the in-order visit of ST. At this point if $\texttt{lcp}[i] = |s[x]|$, the node x is the parent of the leaf labeled $SA[i]$, we connect them and the new ST_i is obtained. If instead $\texttt{lcp}[i] < |s[x]|$, the edge leading to x has to be split by inserting a node u that has two children: the left child is x and the right child is the leaf $SA[i]$ (because it is lexicographically larger than $SA[i-1]$). This node is associated with the value $\texttt{lcp}[i]$. The reader can run this algorithm over the string $T[1,12] = \texttt{mississippi\$}$ and convince herself that the final suffix tree ST_{12} is exactly the one showed in Figure 2.8.

The time complexity of the algorithm derives from an accounting argument which involves the edges traversed by the upward percolation of ST. Since the suffix $suff_{SA[i]}$ is lexicographically greater than the suffix $suff_{SA[i-1]}$, the leaf labeled $SA[i]$ lies to the right of the leaf $SA[i-1]$. So every time we traverse an edge, we either discard it from the next traversals and proceed upward, or we split it and a new leaf is inserted. In particular all edges from $SA[i-1]$ to x are never traversed again because they lie to the left of the newly inserted edge $(u, SA[i])$. The total number of these edges is bounded by the total number of edges in ST, which is $O(n)$ from Fact 2.3. The total number of edge-splits equals the number of inserted leaves, which is again $O(n)$.

FACT 2.6 *Given the suffix array and the longest-common-prefix array of a string $T[1,n]$, we can derive the corresponding suffix tree in $O(n)$ time and space.*

2.3.3 McCreight's algorithm$^{\infty}$

A naïve algorithm for constructing the suffix tree of an input string $T[1,n]$ could start with an empty trie and then iteratively insert text suffixes, one after the other. The algorithm maintains the property by which each intermediate trie is indeed a compacted trie of the suffixes inserted so far. In the worst case, the algorithm costs up to $O(n^2)$ time, take e.g. the highly repetitive string $T[1,n] = a^{n-1}\$$. The reason for this poor behavior is due to the *re-scanning* of parts of the text T that have been already examined during the insertion of previous suffixes. Interestingly enough do exist algorithms that construct the suffix tree directly, and thus without passing through the suffix- and lcp-arrays, and still take $O(n)$ time. Nowadays the space succinctness of suffix arrays and the existence of the Skew algorithm, drive the programmers to build suffix trees passing through suffix arrays (as explained in

the previous section). However, if the *average lcp* among the text suffixes is small then the direct construction of the suffix tree may be advantageous both in internal memory and on disk. We refer the interested reader to [3] for a deeper analysis of these issues.

In what follows we present the classic McCreight's algorithm [11], introduced in 1976. It is based on a nice technique that adds some special pointers to the suffix tree that allow to avoid the *rescanning* mentioned before. These special pointers are called *suffix links* and are defined as follows. The suffix link $SL(z)$ connects the node z to the node z' such that $s[z] = as[z']$. So z' spells out a string that is obtained by dropping the first character from $s[z]$. The existence of z' in ST is not at all clear: Of course $s[z']$ is a substring of T, given that $s[z]$ is, and thus there exists a path in ST that ends up into the extended locus of $s[z']$; but nothing seems to ensure that $s[z']$ has indeed a locus in ST, and thus that z' exists. This property is derived by observing that the existence of z implies the existence of at least 2 suffixes, say $suff_i$ and $suff_j$ that have the node z as their lowest common ancestor in ST, and thus $s[z]$ is their longest common prefix (see Property 2.9). Looking at Figure 2.8, we can take for node z the suffixes $suff_3$ and $suff_6$ (which are actually children of z). Now take the two suffixes following those ones, namely $suff_{i+1}$ and $suff_{j+1}$ (i.e. $suff_4$ and $suff_7$ in the figure). They will share $s[z']$ as their longest common prefix, given that we dropped just their first character, and thus they will have z' as their lowest common ancestor. In Figure 2.8, $s[z]$ = `ssi`, $s[z']$ = `si` and the node z' does exist and is indicated with y. In conclusion every node z has one suffix link correctly defined; more subtle is to observe that all suffix links form a tree rooted in the root of ST: just observe that $|s[z']| < |s[z]|$ so they cannot induce cycles and eventually end up in the root of the suffix tree (spelling out the empty string).

McCreight's algorithm works in n steps, it starts with the suffix tree ST_1 which consists of a root node, denoting the empty string, and one leaf labeled $suff_1 = T[1, n]$ (namely the entire text). In a generic step $i > 1$, the current suffix tree ST_{i-1} is the compacted trie built over all text suffixes $suff_j$ such that $j = 1, 2, \ldots, i-1$. Hence suffixes are inserted in ST from the longest to the shortest one, and at any step ST_{i-1} indexes the $(i-1)$ longest suffixes of T.

To ease the description of the algorithm we need to introduce the notation $head_i$ which denotes the longest prefix of suffix $suff_i$ which occurs in ST_{i-1}. Given that ST_{i-1} is a partial suffix tree, $head_i$ is the longest common prefix between $suff_i$ and any of its previous suffixes in T, namely $suff_j$ with $j = 1, 2, \ldots, i-1$. Given $head_i$ we denote by h_i the (extended) locus of that string in the current suffix tree: actually h_i is the extended locus in ST_{i-1} because $suff_i$ has not yet been inserted. After its insertion, we will have that $head_i = s[h_i]$ in ST_i, and indeed h_i is set as the parent of the leaf associated to the suffix $suff_i$. As an example, consider the suffix $suff_5$ = `byabz$` in the partial suffix trees of Figure 2.10. We have that this suffix shares only the character `b` with the previous four suffixes of T, so $head_5$ = `b` in ST_4, and $head_5$ has extended locus in ST_4, which is the leaf 2. But, after its insertion, we get the suffix tree ST_5 in which $h_5 = v$ in ST_5.

Now we are ready to describe the McCreight's algorithm in detail. To produce ST_i, we must locate in ST_{i-1} the (extended) locus h_i of $head_i$. If it is an extended locus, then the edge incident in this node is split by inserting an internal node, which corresponds to h_i, and spells out $head_i$, to which the leaf for $suff_i$ is attached. In the naïve algorithm, $head_i$ and h_i were found tracing a downward path in ST_{i-1} matching $suff_i$ character-by-character. However this induced a quadratic time complexity in the worst case. Instead McCreight's algorithm determines $head_i$ and h_i by using the information inductively available for string $head_{i-1}$, and its locus h_{i-1}, and the *suffix links* which are already available in ST_{i-1}.

FACT 2.7 *In ST_{i-1} the suffix link $SL(u)$ is defined for all nodes $u \neq h_{i-1}$. It may be the case that $SL(h_{i-1})$ is defined too, because that node was already present in ST_{i-1} before the insertion of $suff_{i-1}$.*

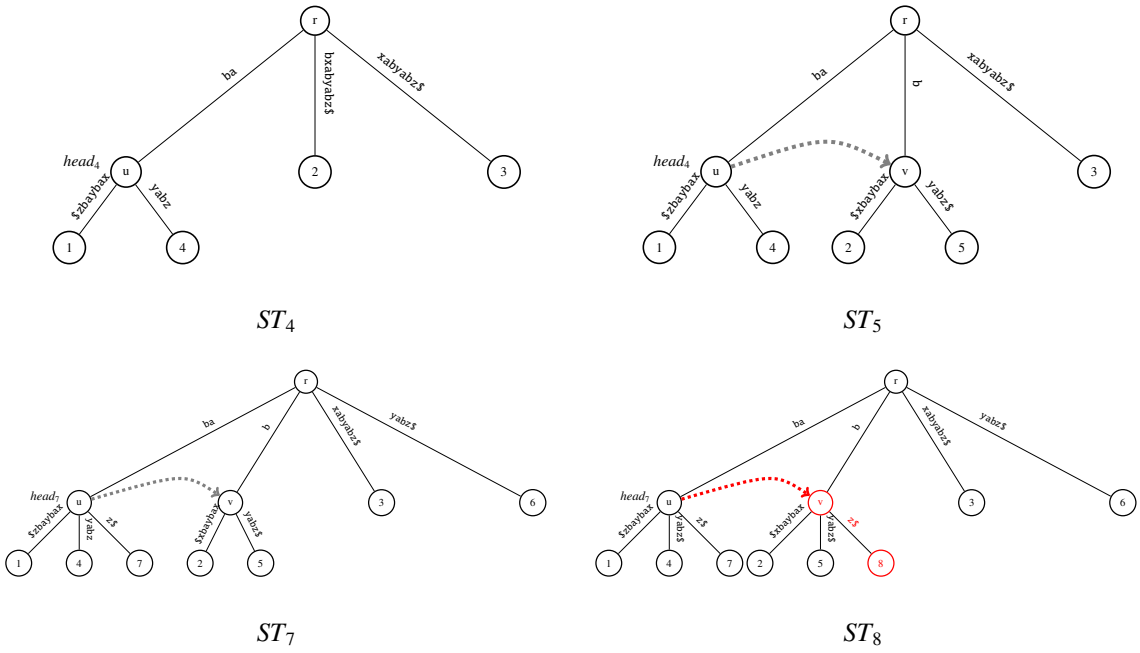


FIGURE 2.10: Several steps of the McCreight's algorithm for the string T =abxabyabz$.

Proof Since $head_{i-1}$ prefixes $suff_{i-1}$, the second suffix of $head_{i-1}$ starts at position i and thus prefixes the suffix $suff_i$. We denote this second suffix with $head^-_{i-1}$. By definition $head_i$ is the longest prefix shared between $suff_i$ and anyone of the previous text suffixes, so that $|head_i| \geq |head_{i-1}| - 1$ and the string $head^-_{i-1}$ prefixes $head_i$. ■

McCreight's algorithm starts with ST_1 that consists of two nodes: the root and the leaf for $suff_1$. At step 1 we have that $head_1$ is the empty string, h_1 is the root, and $SL(root)$ points to the root itself. At a generic step $i > 1$, we know $head_{i-1}$ and h_{i-1} (i.e. the parent of $suff_{i-1}$), and we wish to determine $head_i$ and h_i, in order to insert the leaf for $suff_i$ as a child of h_i. These data are found via the following three sub-steps:

1. if $SL(head_{i-1})$ is defined, we set $w = SL(head_{i-1})$ and we go to step 3;
2. Otherwise we need to perform a **rescanning** whose goal is to find/create the locus w of $head^-_{i-1}$ and consequently set the suffix link $SL(h_{i-1}) = w$. This is implemented by taking the parent f of $head_{i-1}$, jumping via its suffix link $f' = SL(f)$ (which is defined according to Fact 2.7), and then tracing a downward path from f' starting from the $(|s[f']| + 1)$-th character of $suff_i$. Since we know that $head^-_{i-1}$ occurs in T and it prefixes $suff_i$, this downward tracing to find w can be implemented by comparing only the branching characters of the traversed edges with $head^-_{i-1}$. If the landing node of this traversal is the locus of $head^-_{i-1}$, then this landing node is the searched w; otherwise the landing node is the extended locus of $head^-_{i-1}$, so we split the last traversed edge and insert the node w such that $s[w] = head^-_{i-1}$. In all cases we set $SL(h_{i-1}) = w$;
3. Finally, we locate $head_i$ starting from w and **scanning** the rest of $suff_i$. If the locus of $head_i$ does exist, then we set it to h_i; otherwise the scanning of $head_i$ stopped within some edge, and so we split it by inserting h_i as the locus of $head_i$. We conclude the

process by installing the leaf for $suff_i$ as a child of h_i.

Figure 2.10 shows an example of the advantage induced by suffix links. As step 8 we have the partial suffix tree ST_7, $head_7 = \texttt{ab}$, its locus $h_7 = u$, and we need to insert the suffix $suff_8 = \texttt{bz\$}$. Using McCreight's algorithm, we find that $SL(h_7)$ is defined and equal to v, so we reach that node following the suffix link (without rescanning $head_{i-1}^-$). Subsequently, we scan the rest of $suff_8$, namely $\texttt{z\$}$, searching for the locus of $head_8$, but we find that actually $head_8 = head_7^-$, so $h_8 = v$ and we can attach there the leaf 8.

From the point of view of time complexity, we observe that the rescanning and the scanning steps perform two different types of traversals: the former traverses edges by comparing only the branching characters, since it is rescanning the string $head_{i-1}^-$ which is already known from the previous step $i-1$; the latter traverses edges by comparing their labels in their entirety because it has to determine $head_i$. This last type of traversal always advances in T so the cost of the scanning phase is $O(n)$. The difficulty is to evaluate that the cost of rescanning is $O(n)$ too. The proof comes from an observation on the structure of suffix links and suffix trees: if $SL(u) = v$ then all ancestors of u point to a distinct ancestor of v. This comes from Fact 2.7 (all these suffix links do exist), and from the definition of suffix links (which ensures ancestorship). Hence the tree-depth of $v = SL(u)$, say $d[v]$, is larger than $d[u] - 1$ (where -1 is due to the dropping of the first character). Therefore, the execution of rescanning can decrease the current depth at most by 2 (i.e., one for reaching the father of h_{i-1}, one for crossing $SL(h_{i-1})$). Since the depth of ST is most n, and we loose at most two levels per SL-jump, then the number of edges traversed by rescanning is $O(n)$, and each edge traversal takes $O(1)$ time because only the branching character is matched.

The last issue to be considered regards the cost of branching out of a node during the re-scanning and the scanning steps. Previously we stated that this costs $O(1)$ by using perfect hash-tables built over the branching characters of each internal node of ST. In the context of suffix-tree construction the tree is dynamic and thus we should adopt dynamic perfect hash-tables, which is a pretty involved solution. A simpler approach consists of keeping the branching characters and their associated edges within a binary-search tree thus supporting the branching in $O(\log \sigma)$ time. Practically, programmers relax the requirement of worst-case complexity and use either hash tables with chaining, or dictionary data structures for integer values (such as the Van Emde-Boas tree, whose search complexity is $O(\log \log \sigma)$ time) because characters can be looked at as sequences of bits and hence integers.

THEOREM 2.10 *McCreight's algorithm builds the suffix tree of a string $T[1,n]$ in $O(n \log \sigma)$ time and $O(n)$ space.*

This algorithm is inefficient in an external-memory setting because it may elicit one I/O per each tree-edge traversal. Nevertheless, as we observed before, of the distribution of the lcps is skewed towards small values, then this construction might be I/O-efficient in that the top part of the suffix tree could be cached in the internal memory, and thus do not elicit any I/Os during the scanning and re-scanning steps. We refer the reader to [3] for details on this issue.

2.4 Some interesting problems

2.4.1 Approximate pattern matching

The problem of approximate pattern matching can be formulated as: *finding all substrings of a text $T[1,n]$ that match a pattern $P[1,p]$ with at most k errors*. In this section we restrict our discussion to the simplest type of errors, the ones called *mismatches* or *substitutions* (see Figure 2.11). This way the text substrings which "k-mismatch" the searched pattern P have length p and coincide with

the pattern in all but at most k characters. The following figure provides an example by considering two DNA strings formed over the alphabet of four nucleotide bases $\{A, T, G, C\}$. The reason for this kind of strings is that Bio-informatics is the context which spurred interest around the approximate pattern-matching problem.

```
C C G T A C G A T C A G T A
C C G A A C T
      ⇑     ⇑
```

FIGURE 2.11: An example of matching between T (top) and P (bottom) with $k = 2$ mismatches.

The naïve solution to this problem consists of trying to match P with every possible substring of T, having length p, counting the mismatches and returning the positions were their number is at most k. This would take $O(pn)$ time, independently of k. The inefficiency comes from the fact that each pattern-substring comparison starts from the beginning of P, thus taking $O(p)$ time. In what follows we describe a sophisticated solution which hinges on an elegant data structure that solves an apparently un-related problem formulated over an array of integers, and called *Range Minimum Query* (shortly, RMQ). This data structure is the backbone of many other algorithmic solutions in problems arising in Data Mining, Information Retrieval, and so on.

The following Algorithm 2.4 solves the k-mismatches problem in $O(nk)$ time by making the following basic observation. If P occurs in T with $j \leq k$ mismatches, then we can align the pattern P with a substring of T so that j or $j-1$ substrings coincide and j characters mismatch. Actually equal substrings and mismatches interleave each other. As an example consider again Figure 2.11, the pattern occurs at position 1 in the text T with 2 mismatches, and in fact two substrings of P match their corresponding substrings of T. This means that if we could compare pattern and text substrings for equality in constant time, then we could execute the naïve-approach taking $O(nk)$ time, instead of $O(np)$ time. To be operational, this observation can be rephrased as follows: if $T[i, i+\ell] = P[j, j+\ell]$ is one of these matching substrings, then ℓ is the longest common prefix between the pattern and the text suffixes starting at the matching positions i and j. Algorithm 2.4 deploys this rephrasing to code a solution which takes $O(nk)$ time provided that `lcp`-computations take $O(1)$ time.

If we run the Algorithm 2.4 over the strings showed in Figure 2.11, we perform two `lcp`-computations and find that P occurs at text position 1 with 2-mismatches:

- $\texttt{lcp}(T[1,14], P[1,7]) = \texttt{lcp}(\texttt{CCGTACGATCAGTA}, \texttt{CCGTACG}) = \texttt{CCG}$.
- $\texttt{lcp}(T[5,14], P[5,7]) = \texttt{lcp}(\texttt{ACGATCAGTA}, \texttt{ACG}) = \texttt{AC}$.

How do we compute $\texttt{lcp}(T[i+j-1, n], P[j, p])$ in constant time? We know that suffix trees and suffix arrays have built-in some `lcp`-information, but we similarly recall that these data structures were built on one single string, namely the text T. Here we are talking of suffixes of P and T together. Nonetheless we can easily circumvent this difficulty by constructing the suffix array, or the suffix tree, over the string $X = T\#P$, where # is a new character not occurring elsewhere. This way each computation of the form $\texttt{lcp}(T[i+j-1, n], P[j, p])$ can now be turned into an `lcp`-computation between suffixes of X, precisely $\texttt{lcp}(T[i+j-1, n], P[n+1+j, n+1+p])$. We are therefore left with showing how these `lcp`-computations can be performed in constant time, whichever is the pair of compared suffixes. This is the topic of the next subsection.

Algorithm 2.4 Approximate-pattern matching based on LCP-computations

```
matches = {}
for i = 1 to n do
    m = 0, j = 1;
    while m ≤ k and j ≤ p do
        ℓ = lcp(T[i + j − 1, n], P[j, p];
        j = j + ℓ;
        if j ≤ p then
            m = m + 1; j = j + 1;
        end if
    end while
    j = 1;
    if m ≤ k then
        matches = matches ∪ {T[i, i + p − 1]};
    end if
end for
return matches;
```

Lowest Common Ancestor, Range Minimum Query and Cartesian Tree

Let us start from an example, by considering the suffix tree ST_X and the suffix array SA_X built on the string $X = CCGTACGATCAGTA$. This string is not in the form $X = T\#P$ because we wish to stress the fact that the considerations and the algorithmic solutions proposed in this section apply to any string X, not necessarily the ones arising from the Approximate Pattern-Matching problem.

The key observation, whose correctness spurs immediately from Figure 2.12, is that there is a strong relation between the `lcp`-problem over X's suffixes and the computation of *lowest common ancestors* (`lca`) in the suffix tree ST_X. Consider the problem of finding the longest common prefix between suffixes $X[i, x]$ and $X[j, x]$) where $x = |X|$. It is not difficult to convince yourself that the node $u = \texttt{lca}(X[i, x], X[j, x])$ in the suffix tree ST_X spells out their `lcp`, and thus the value $|s[u]|$ stored in node u is exactly the `lcp`-value we are searching for. Notice that this property holds independently of the lexicographic sortedness of the edge labels, and thus of the suffix tree leaves.

Equivalently, the same value can be derived by looking at the suffix array SA_X. In particular take the lexicographic positions i_p and j_p where those two suffixes occur in SA_X, say $SA_X[i_p] = i$ and $SA_X[j_p] = j$ (we are assuming for simplicity that $i_p < j_p$). It is not difficult to convince yourself that the *minimum value* in the sub-array $\texttt{lcp}[i_p, j - 1]$[5] is exactly equal to $|s[u]|$ since the values contained in that sub-array are the values stored in the suffix-tree nodes of the subtree that descends from u. Actually the order of these values is the one given by the in-order visit of u's descendants. Anyway, this order is not important for our computation which actually takes the smallest value, because it is interested in the shallowest node (namely the root u) of that subtree.

Figure 2.12 provides a running example which clearly shows these two strong properties, which actually do not depend on the order of the children of suffix-tree nodes. As a result, we have two approaches to compute `lcp`in constant time, either through `lca`-computations over ST_X or through `RMQ`-computations over $\texttt{lcp}_X$. For the sake of presentation we introduce an elegant solution for the latter, which actually induces in turn an elegant solution for the former, given that their are strongly related.

[5]Recall that $\texttt{lcp}[q]$ stores the length of the longest common prefix between suffix $SA[i]$ and its next suffix $SA[i + 1]$.

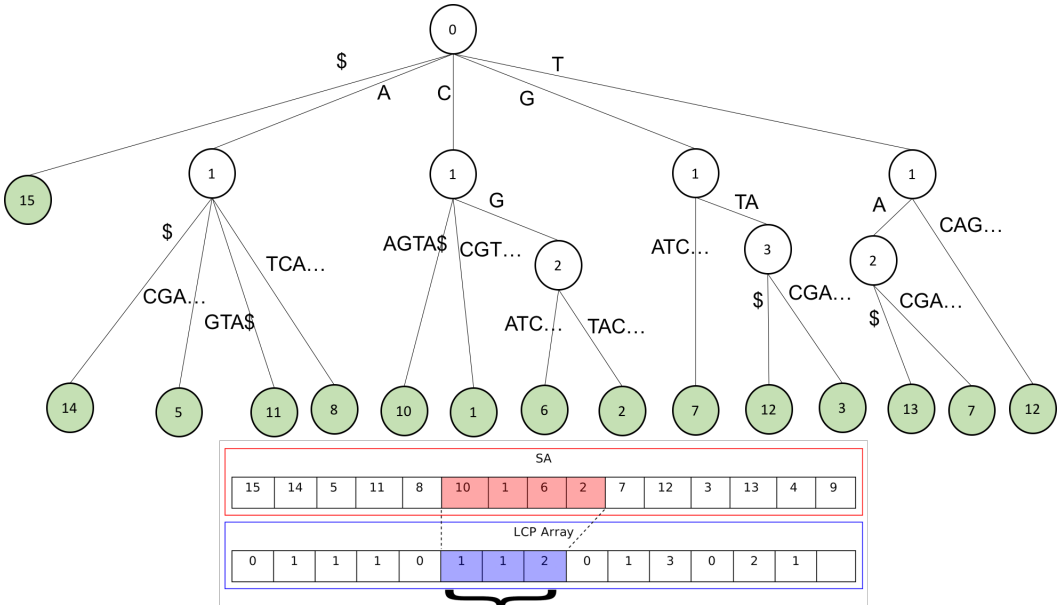


FIGURE 2.12: An example of suffix tree, suffix array, `lcp`-array for the string X = `CCGTACGATCAGTA`. In the suffix tree we have indicated only a prefix of few characters of the long edge labels. The figure highlights that the computation of $\texttt{lcp}(X[2,16], X[10,16])$ boils down to finding the depth of the `lca`-node in ST_X between the leaf 2 and the leaf 10, as well as to solve a range minimum query on the sub-array $\texttt{lcp}[6,8]$ since $SA_X[6] = 10$ and $SA_X[9] = 2$.

In general terms the `RMQ` problem can be stated as follows:

> **The range-minimum-query problem.** *Given an array $A[1,n]$ of elements drawn from an ordered universe, build a data structure $\texttt{RMQ}_A$ that is able to compute efficiently the position of a smallest element in $A[i,j]$, for any given queried range (i,j). We say "a smallest" because the array may contain many minimum elements.*

We underline that this problem asks for the *position* of a minimum element in the queried sub-array, rather than its value. This is more general because the value of the minimum can be obviously retrieved from its position in A by accessing this array, which is available.

In this lecture we aim for constant-time queries [1]. The simplest solution achieves this goal via a table that stores the index of a minimum entry for each possible range (i,j). Such table requires $O(n^2)$ space and $O(n^2)$ time to be built. A better solution hinges on the following observation: any range (i,j) can be decomposed into two (possibly overlapping) ranges whose size is a power of two, namely $(i, i+2^L)$ and $(j-2^L, j)$ where $L = \lfloor \log(j-i+1) \rfloor$. This allows us to *sparsify* the previous quadratic-sized table by storing only ranges whose size is a power of two. This way, for each position i we store the answers to the queries $\texttt{RMQ}_A(i, i+2^L)$, thus occupying a total space of $O(n \log n)$ without impairing the time complexity of the query which is still constant and corresponds to return $\texttt{RMQ}_A(i,j) = \texttt{argmin}_{i,j}\{\texttt{RMQ}_A(i, i+2^L), \texttt{RMQ}_A(j-2^L, j)\}$.

In order to get the optimal $O(n)$ space occupancy, we need to dig into the structure of the `RMQ`-problem and make a twofold reduction which goes back-and-forth from `RMQ`-computations to `lca`-computations: namely, we reduce (1) the `RMQ`-computation over the `lcp`-array to an `lca`-computation over Cartesian Trees (that we define next); we then reduce (2) the `lca`-computation over Cartesian Trees to an `RMQ`-computation over a binary array. This last problem will then be

solved in $O(n)$ space and constant query time. Clearly reduction (2) can be applied to any tree, and thus can be applied to Suffix Trees in order to solve `lca`-queries over them.

First reduction step: from `RMQ` to `lca`. We transform the $\texttt{RMQ}_A$-problem "back" into an `lca`-problem over a special tree which is known as *Cartesian Tree* and is built over the entries of the array $A[1, n]$. The *Cartesian Tree* C_A is a binary tree of n nodes, each labeled with one of A's entries (i.e. value and position in A). The labeling is defined recursively as follows: the root of C_A is labeled by the minimum entry in $A[1, n]$, say this is $\langle A[m], m\rangle$. Then the left subtree of the root is recursively defined as the Cartesian Tree of the subarray $A[1, m-1]$, and the right subtree is recursively defined as the Cartesian Tree of the subarray $A[m+1, n]$. See Figure 2.13 for an example.

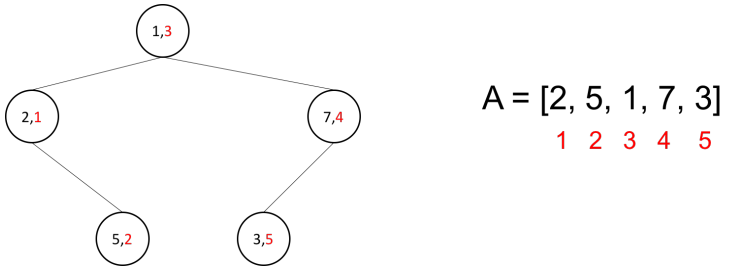


FIGURE 2.13: Cartesian Tree built over the array $A[1, 5] = \{2, 5, 1, 7, 3\}$. Observe that nodes of C_A store as first (black) number A's values, and as second (red) number their positions in A.

The following Figure 2.14 shows the Cartesian tree built on the `lcp`-array depicted in Figure 2.12. Given the construction process, we can state that ranges in the `lcp`-array correspond to subtrees of the Cartesian tree. Therefore computing $\texttt{RMQ}_A(i, j)$ boils down to compute an `lca`-query between the nodes of C_A associated to the entries i and j. Differently of what occurred for `lca`-queries on ST_X, where the arguments were leaves of that suffix tree, the queried nodes in the Cartesian Tree may be internal nodes, and actually it might occur that one node is ancestor of the other node. For example, executing $\texttt{RMQ}_{\texttt{lcp}}(6, 8)$ equals to executing $\texttt{lca}(6, 8)$ over the Cartesian Tree $C_{\texttt{lcp}}$ of Figure 2.14. The result of this query is the node $\langle \texttt{lcp}[7], 7\rangle = \langle 1, 7\rangle$. Notice that we have another minimum value in $\texttt{lcp}[6, 8]$ at $\texttt{lcp}[6] = 1$; the answer provided by the `lca` is one of the existing minima in the queried-range.

Second reduction step: from `lca` to `RMQ`. We transform the `lca`-problem over the Cartesian Tree $C_{\texttt{lcp}}$ "back" into an `RMQ`-problem over a special binary array $\Delta[1, 2e]$, where e is the number of edges in the Cartesian Tree (of course $e = O(n)$). It seems strange this "circular" sequence of reductions that now has turned us back into an `RMQ`-problem. But the current `RMQ`-problem, unlike the original one, is formulated on a binary array and thus admits an optimal solution in $O(n)$ space.

To build the binary array $\Delta[1, 2e]$ we need first to build the array $D[1, 2e]$ which is obtained as follows. Take the *Euler Tour* of Cartesian Tree C_A, visiting the tree in pre-order and writing down each node everytime the visit passes through it. A node can be visited multiple times, precisely it is visited/written as many times as its number of incident edges; except for the root which is written the number of incident edges plus 1.

Given the Euler Tour of the Cartesian Tree C_A, we build the array $D[1, 2e]$ which stores the depths of the visited nodes in the order stated by the Euler Tour (see Figure 2.14). Given D and the way the Euler Tour is built, we can conclude that query $\texttt{lca}(i, j)$ in C_A boils down to compute the node of minimum depth in the sub-array $D[i', j']$ where i' (resp. j') is the position of the first (resp. last)

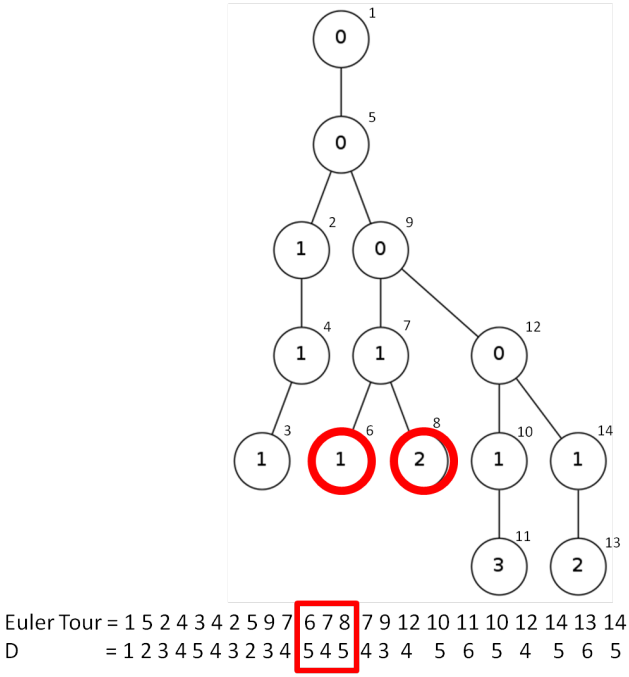


FIGURE 2.14: Cartesian tree built on the `lcp`-array of Figure 2.12: inside the nodes we report the LCP's values, outside the nodes we report the corresponding position in the LCP array (in case of ties in the LCP's values we make an arbitrary choice). On the bottom part are reported the Euler Tour of the Cartesian Tree and the array D of the depths of the nodes according to the Euler-Tour order.

occurrence of the node i (resp. j) in the Euler Tour. In fact, the range $D[i', j']$ corresponds to the part of the Euler Tour that starts at node i and ends at node j. The node of minimum depth encountered in this Euler sub-Tour is properly the $\texttt{lca}(i, j)$.

So we reduced an `lca`-query over the Cartesian Tree into an `RMQ`-query over node depths. In our running example on Figure 2.14 this reduction transforms the query $\texttt{lca}(6, 8)$ into a query $\texttt{RMQ}_D(11, 13)$, which is highlighted by a red rectangle. Turning nodes into ranges can be done in constant time by simply storing two integers per node of the Cartesian Tree, denoting their first/last occurrence in the Euler Tour, thus taking $O(n)$ space.

We are again "back" to an `RMQ`-query over an integer array. But the current array D is special because its consecutive entries differ by 1 given that they refer to the depths of consecutive nodes in an Euler Tour. And in fact, two consecutive nodes in the Euler Tour are connected by an edge and thus one node is the parent of the other, and hence their depths differ by one unit. The net result of this property is that we can solve the `RMQ`-problem over $D[1, 2e]$ in $O(n)$ space and $O(1)$ time as follows. (Recall that $e = O(n)$.) Solution is based on two data structures which are detailed next.

First, we split the array D into $\frac{2e}{d}$ subarrays D_k of size $d = \frac{1}{2}\log e$ each. Next, we find the minimum element in each subarray D_k, and store its position at the entry $M[k]$ of a new array whose size is therefore $\frac{2e}{d}$. We finally build on the array M the sparse-table solution indicated above which takes superlinear space (in the size of M) and solves `RMQ`-queries in constant time. The key point here is that M's size is sublinear in e, and thus in n, so that the overall space taken by array M and its sparse-table is $O((\frac{e}{\log e}) * \log \frac{e}{\log e}) = O(e) = O(n)$.

The second data structure is built to efficiently answer `RMQ`-queries in which i and j are in the

same block D_k. It is clear that we cannot tabulate all answers to all such possible pairs of indexes because this would end up in $\Theta(n^2)$ space occupancy. So the solution we describe here spurs from two simple, deep observations whose proof is immediate and left to the reader:

Binary entries: Every block D_k can be transformed into a pair that consists of its first element $D_k[1]$ and a binary array $\Delta_k[i] = D_k[i] - D_k[i-1]$ for $i = 2, \ldots, d$. Entries of Δ_k are either -1 or $+1$ because of the unit difference between adjacent entries of D.

Minimum location: The position of the minimum value in D_k depends only on the content of the binary sequence Δ_k and does not depend on the starting value $D_k[1]$.

Nicely, the possible configurations that every block D_k can assume are infinite, given that infinite is the number of ways we can instantiate the input array A on which we want to issue the `RMQ`-queries; but the possible configurations of the image Δ_k is finite and equal to 2^d. This suggests to apply the so called *Four Russians trick* to the binary arrays by tabulating all possible binary sequences Δ_k and, for each of them, storing the position of the minimum value. Since the blocks Δ_k have length $d = \frac{\log e}{2}$, the total number of possible binary sequences is $2^d = O(2^{\frac{\log e}{2}}) = O(\sqrt{e}) = O(\sqrt{n})$. Moreover, since both query-indexes i and j can take at most $d = \frac{\log e}{2}$ possible values, being internal in a block D_k, we can have at most $O(\log^2 e)$ queries of this third type. Consequently, we build a lookup table $T[i_o, j_o, \Delta_k]$ that is indexed by the possible query-offsets i_o and j_o within the block D_k and its binary configuration Δ_k. Table T stores at that entry the position of the minimum value in D_k. We also assume that, for each k, we have stored Δ_k so that the binary representation Δ_k of D_k can be retrieved in constant time. Each of these indexing parameters takes $O(\log e) = O(\log n)$ bits of space, hence one memory word, and thus can be managed in $O(1)$ time and space. In summary, the whole table T consists of $O(\sqrt{n}(\log n)^2) = o(n)$ entries. The time needed to build T is $O(n)$. The power of transforming D_k into Δ_k is evident now, every entry of $T[i_o, j_o, \Delta_k]$ is actually encoding the answer for an infinite number of blocks D_k, namely the ones that can be turned to the same binary configuration Δ_k.

At this point we are ready to design an algorithm that, using the three data structures illustrated above, answers a query $\texttt{RMQ}_D(i, j)$ in constant time. If i, j are inside the same block D_k then the answer is retrieved in two steps: first we compute the offsets i_o and j_o with respect to the beginning of D_k and determine the binary configuration Δ_k from k; then we use this triple to access the proper entry of T. Otherwise the range (i, j) spans at least two blocks and can thus be decomposed in three parts: a suffix of some block $D_{i'}$, a consecutive sequence of blocks $D_{i'+1} \cdots D_{j'-1}$, and finally the prefix of block $D_{j'}$. The minimum for the suffix of $D_{i'}$ and the prefix of $D_{j'}$ can be retrieved from T, given that these ranges are inside two blocks. The minimum of the range spanned by $D_{i'+1} \cdots D_{j'-1}$ is stored in M. All this information can be accessed in constant time and the final minimum-position can be retrieved by comparing these three minimum values, in constant time too.

THEOREM 2.11 *Range-minimum queries over an array $A[1, n]$ of elements drawn from an ordered universe can be answered in constant time using a data structure that occupies $O(n)$ space.*

Given the stream of reductions we illustrated above, we can conclude that Theorem 2.11 applies also to computing `lca` in generic trees: it is enough to take the input tree in place of the Cartesian Tree.

THEOREM 2.12 *Lowest-common-ancestor queries over a generic tree of size n can be answered in constant time using a data structure that occupies $O(n)$ space.*

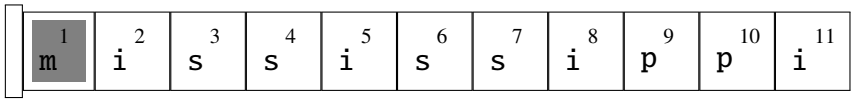
2.4.2 Text Compression

Data compression will be the topic of one of the following chapters; nonetheless in this section we address the problem of compressing a text via the simple algorithm which is at the core of the well known `gzip` compressor, named *LZ77* from the initials of its inventors (Abraham Lempel and Jacob Ziv [9]) and from the year of its publication (1977). We will show that there exists an optimal implementation of the LZ77-algorithm taking $O(n)$ time and using suffix trees.

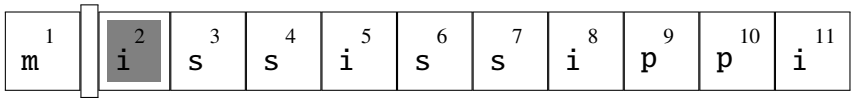
Given a text string $T[1,n]$, the algorithm LZ77 produces a parsing of T into substrings that are defined as follows. Assume that it has already parsed the prefix $T[1,i-1]$ (at the beginning this prefix is empty), then it decomposes the remaining text suffix $T[i,n]$ in three parts: the longest substring $T[i,i+\ell-1]$ which starts at i and repeats before in the text T, the next character $T[i+\ell]$, and the remaining suffix $T[i+\ell+1,n]$. The next substring to add to the parsing of T is $T[i,i+\ell]$, and thus corresponds to the shortest string that is *new* in $T[1,i-1]$. Parsing then continues onto the remaining suffix $T[i+\ell+1,n]$, if any.

Compression is obtained by succinctly encoding the triple of integers $\langle d,\ell,T[i+\ell]\rangle$, where d is the distance (in characters) from i to the previous copy of $T[i,i+\ell-1]$; ℓ is the length of the copied string; $T[i+\ell]$ is the appended character. By saying "previous copy" of $T[i,i+\ell-1]$, we mean that its copy starts before position i but it might extend after this position, hence it could be $d<\ell$; furthermore, the previous copy can be any previous occurrence of $T[i,i+\ell-1]$, although space-efficiency issues suggest us to take the closest copy (and thus the smallest d). Finally we observe that the reason for adding the character $T[i+\ell]$ to the emitted triple is that this character behaves like an *escape*-mechanism; in fact it is useful when no-copy is possible and thus $\ell=0$ (this occurs when a new character is met in T).[6]

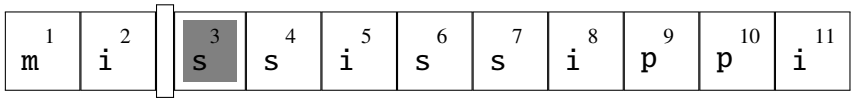
Before digging into an efficient implementation of the LZ77-algorithm let us consider our example string $T=$ `mississippi`. Its LZ77-parsing is computed as follows:



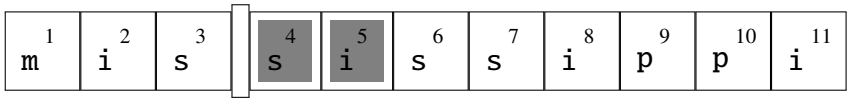
Output: $<0,0,m>$



Output: $<0,0,i>$

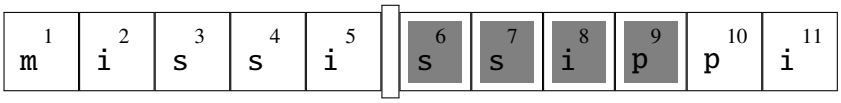


Output: $<0,0,s>$

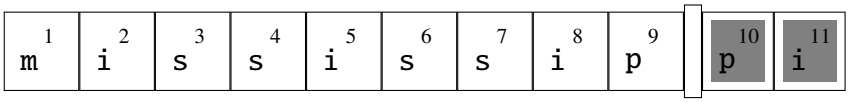


Output: $<1,1,i>$

[6] We are not going to discuss the integer-encoding issue, since it will be the topic of a next chapter, we just mention here that efficiency is obtained in `gzip` by taking the rightmost copy and by encoding the values d and ℓ via a Huffman coder.



Output: $< 3, 3, p >$



Output: $< 1, 1, i >$

We can compute the LZ77-parsing in $O(n)$ time via an elegant algorithm that deploys the suffix tree ST. The difficulty is to find π_i, the longest substring that occurs at position i and repeats before in the text T. Say d is the distance of the previous occurrence. Given our notation above we have that $\ell = |\pi_i|$. Of course π_i is a prefix of $suff_i$ and a prefix of $suff_{i-d}$; actually, it is the longest common prefix of these two suffixes, and by maximality, there is no other previous suffix $suff_j$ (with $j < i$) that shares a longer prefix with $suff_i$. By properties of suffix trees, the lowest-common-ancestor of leaves i and j spells out π_i. However we cannot compute $\texttt{lca}(i, j)$ by issuing a query to the data structure of Theorem 2.12 because we do not know j, which is exactly the information we wish to compute. Similarly we cannot trace a downward path from the root of ST trying to match $suff_i$ because all suffixes of T are indexed in the suffix tree and thus we could detect a longer copy which follows position i, instead of preceding it.

To circumvent these problems we preprocess ST via a post-order visit that computes for every internal node u its minimum leaf $\min(u)$. Clearly $\min(u)$ is the leftmost position from which we can copy the substring $s[u]$. Given this information we can determine easily π_i, just trace a downward path from the root of ST scanning $suff_i$ and stopping as soon as the reached node v is such that $\min(v) = i$. At this point we take u as the parent of v and set $\pi_i = s[u]$, and $d = i - \min(u)$. Clearly, the chosen copy of π_i is the farthest one and not the closest one: this does not impact in the number of phrases in which T is parsed by LZ77, but possibly influences the magnitude of these distances and thus their succinct encoding. Devising an LZ77-parser that efficiently determines the closest copy of each π_i is non trivial and needs much more sophisticated data structures.

Take $T = \texttt{mississippi}$ as the string to be parsed (see above) and consider its suffix tree ST in Figure 2.8. Assume that the parsing is at the suffix $suff_2 = \texttt{ississippi}$. Its tracing down ST stops immediately at the root of the suffix tree because the node to be visited next would be u, for which $\min(u) = 2$ which is not smaller than the current suffix position. Then consider the parsing at suffix $suff_6 = \texttt{ssippi}$. We trace down ST and actually exhaust $suff_6$, so reaching the leaf 6, for which min is 3. So the selected node is its parent z, for which $s[z] = \texttt{ssi}$. The emitted triple is correctly $< 3, 3, p >$.

The time complexity of this implementation of the LZ77-algorithm is $O(n)$ because the traversal of the suffix tree advances over the string T, and this may occur only n times. Branching out of suffix-tree nodes can be implemented in $O(1)$ time via perfect hash tables, as observed for the substring-search problem. The construction of the suffix tree costs $O(n)$ time, by using one of the algorithms we described in the previous sections. The computation of the values $\min(u)$, over all nodes u, takes $O(n)$ time via a post-order visit of ST.

THEOREM 2.13 *The LZ77-parsing of a string $T[1, n]$ can be computed in $O(n)$ time and space. Each substring of the parsing is copied from its farthest previous occurrence.*

2.4.3 Text Mining

In this section we briefly survey two examples of uses of suffix arrays and `lcp`-arrays in the solution of sophisticated text mining problems.

Let us consider the following question: *Check whether there exists a substring of $T[1,n]$ that repeats at least twice and has length L.* Solving this problem in a brute-force way would mean to take every text substring of length L, and count its number of occurrences in T. These substrings are $\Theta(n)$, searching each of them takes $O(nL)$ time, hence the overall time complexity of this trivial algorithm would be $O(n^2L)$. A smarter and faster, actually optimal, solution comes from the use either of the suffix tree or of the lcp-array `lcp`, built on the input text Ts.

The use of suffix tree is simple. Let us assume that such a string does exist, and it occurs at positions x and y of T. Now take the two leaves in the suffix tree which correspond to $suff_x$ and $suff_y$ and compute their lowest common ancestor, say $a(x,y)$. Since $T[x, x+L-1] = T[y, y+L-1]$, it is that $|s[a(x,y)]| \geq L$. We write "greater or equal" because it could be the case that a longer substring is shared at positions x and y, in fact L is just fixed by the problem. The net result of this argument is that it does exist an *internal* node in the suffix tree whose label is greater or equal than L; a visit of the suffix tree is enough to search for any node such this one, thus taking $O(n)$ time.

The use of the suffix array is a little bit more involved, but follows a similar argument. Recall that suffixes in SA are lexicographically ordered, so the longest common prefix shared by suffix $SA[i]$ is with its adjacent suffixes, namely either with suffix $SA[i-1]$ or with suffix $SA[i+1]$. The length of these lcps is stored in the entries $\texttt{lcp}[i-1, i]$. Now, if the repeated substring of length L does exist, and it occurs e.g. at text positions x and y, then we have $\texttt{lcp}(T[x,n], T[y,n]) \geq L$. These two suffixes not necessarily are contiguous in SA (this is the case when the substring occurs more than twice), nonetheless all suffixes occurring among them in SA will surely share a prefix of length L, because of their lexicographic order. Hence, if suffix $T[x,n]$ occurs at position q of the suffix array, i.e. $SA[q] = x$, then we have that either $\texttt{lcp}[q-1] \geq L$ or $\texttt{lcp}[q] \geq L$, depending on the fact that $T[y,n] < T[x,n]$ or vice versa, respectively. Hence we can solve the question stated above by scanning `lcp` and searching for an entry $\geq L$. This takes $O(n)$ optimal time.

Let us now ask a more sophisticated question: *Check whether there exists a text substring that repeats at least C times and has length L.* This is the typical query in a text mining scenario, where we are interested not just in a repetitive event but in an event occurring with some *statistical evidence.* We can again solve this problem by trying all possible substrings and counting their occurrences. Again, a faster solution comes from the use either of the suffix tree or of the array `lcp`. Following the argument provided in the solution of the previous question we note that, if a substring of length L occurs (at least) C times, then it does exist (at least) C text suffixes that share (at least) L characters. So it does exist a node u in the suffix tree such that $|s[u]| \geq L$ and the number of descending leaves $occ[u] \geq C$. Equivalently, it does exist a sub-array in `lcp` of length $\geq C-1$ that consists of entries $\geq L$. Both approaches provide an answer to the above question in $O(n)$ time.

Let us conclude this section by asking a query closer to a search-engine scenario: *Given two patterns P and Q, and a positive integer k, check whether there exists an occurrence of P whose distance from an occurrence of Q in an input text T is at most k.* This is also called *proximity search* over a text T which is given in advance to be preprocessed. The solution passes through the use of any search data structure, being it a suffix tree or a suffix array built over T, plus some sorting/merging steps. We search for P and Q in T and derive their occurrences, say O. Both suffix arrays and suffix trees return these occurrences unsorted. Therefore we sort them, in order to produce the ordered list of occurrences of P and Q. At this point it is easy to determine whether the question above admits a positive answer; if it does, then there do exist two *consecutive* occurrences of P and Q whose distance is at most k. To detect this situation it is enough to scan the sorted Sequence O and check, for every *consecutive* pair of positions which are occurrences of P and Q, whether the difference is at most k. This takes overall $O(|P| + |Q| + |O| \log |O|)$ time, which is clearly

advantageous whenever the set O of candidate occurrences is small, and thus the queries P and Q are sufficiently selective.

References

[1] Michael A. Bender and Martin Farach-Colton. The LCA Problem Revisited. In *Procs of the Latin American Symposium on Theoretical Informatics (LATIN)*, 88-94, 2000.

[2] Martin Farach-Colton, Paolo Ferragina, S. Muthukrishnan. On the sorting-complexity of suffix tree construction. *Journal of the ACM*, 47(6): 987-1011, 2000.

[3] Paolo Ferragina. String search in external memory: algorithms and data structures. *Handbook of Computational Molecular Biology*, edited by Srinivas Aluru. Chapman & Hall/CRC Computer and Information Science Series, chapter 35, Dicembre 2005.

[4] Paolo Ferragina and Travis Gagie and Giovanni Manzini. Lightweight data indexing and compression in external memory. *Algorithmica: Special issue on selected papers of LATIN 2010*, 63(3): 707-730, 2012.

[5] Dan Gusfield. *Algorithms on Strings, Trees and Sequences: Computer Science and Computational Biology.* University Press, 1997.

[6] Gaston H. Gonnet, Ricardo A. Baeza-Yates, and Tim Snider. New indices for text: PAT trees and PAT arrays. In B. Frakes and R. A. Baeza-Yates, editors, *Information Retrieval: Data Structures and Algorithms*, chapter 5, pages 66–82, Prentice-Hall, 1992.

[7] Juha Kärkkäinen and Peter Sanders. Simple linear work suffix array construction. In *Procs of the International Colloquium on Automata, Languages and Programming (ICALP)*, Lecture Notes in Computer Science vol. 2791, Springer, 943–955, 2003.

[8] Toru Kasai, Gunho Lee, Hiroki Arimura, Setsuo Arikawa, and Kunsoo Park. Linear-time longest-common-prefix computation in suffix arrays and its applications. In *Procs of the Symposium on Combinatorial Pattern Matching (CPM)*, Lecture Notes in Computer Science vol. 2089, Springer, 181–192, 2001.

[9] Jacob Ziv and Abraham Lempel. A Universal Algorithm for Sequential Data Compression. *IEEE Transactions on Information Theory*, 23(3): 337-343, 1977.

[10] Udi Manber and Gene Myers. Suffix arrays: a new method for on-line string searches. *SIAM Journal on Computing*, 22(5):935–948, 1993.

[11] Edward M. McCreight. A space-economical suffix tree construction algorithm. *Journal of the ACM*, 23(2): 262-272, 1976.