Problem set for the Algoritmica 2 class (2016/7)

Roberto Grossi
Dipartimento di Informatica, Università di Pisa
grossi@di.unipi.it

November 17, 2016

Abstract

This is the problem set assigned during class. What is relevant during the resolution of the problems is the reasoning path that leads to their solutions, thus offering the opportunity to learn from mistakes. This is why they are discussed by students in groups, one class per week, under the supervision of the teacher to guide the brainstorming process behind the solutions. The *wrong* way to use this problem set: accumulate the problems and start solving them alone, a couple of weeks before the exam. The correct way: solve them each week in groups, discussing them with classmates and teacher.

1. [Range updates] Consider an array C of n integers, initially all equal to zero. We want to support the following operations:

   - `update`(i, j, c), where $0 \leq i \leq j \leq n-1$ and c is an integer: it changes C such that $C[k] := C[k] + c$ for every $i \leq k \leq j$.
   - `query`(i), where $0 \leq i \leq n-1$: it returns the value of $C[i]$.
   - `sum`(i, j), where $0 \leq i \leq j \leq n-1$: it returns $\sum_{k=i}^{j} C[k]$.

   Design a data structure that uses $O(n)$ space, takes $O(n \log n)$ construction time, and implements each operation above in $O(\log n)$ time. Note that `query`$(i) =$ `sum`(i, i) but it helps to reason.

   [Hint: For the general case, use the segment tree seen in class, which uses $O(n \log n)$ space: prove that its space is actually $O(n)$ when it is employed for this problem.] [Hint to further save space in practice when the only changes are `update`(i, i, c): use an implicit tree such as the Fenwick tree (see wikipedia).]

2. [Depth of a node in a random search tree] A random search tree for a set S can be defined as follows: if S is empty, then the null tree is a random search tree; otherwise, choose uniformly at random a key $k \in S$: the random search tree is obtained by picking k as root, and the random search trees on $L = \{x \in S : x < k\}$ and $R = \{x \in S :$

$x > k\}$ become, respectively, the left and right subtree of the root k. Consider the randomized QuickSort discussed in class and analyzed with indicator variables [CLRS 7.3], and observe that the random selection of the pivots follows the above process, thus producing a random search tree of n nodes. Using a variation of the analysis with indicator variables, prove that the expected depth of a node (i.e. the random variable representing the distance of the node from the root) is nearly $2 \ln n$. Prove that the expected size of its subtree is nearly $2 \ln n$ too, observing that it is a simple variation of the previous analysis.

Prove that the probability that the expected depth of a node exceeds $c2 \ln n$ is small for any given constant $c > 1$. [Note: the latter point can be solved after we see Chernoff's bounds.]

3. [Karp-Rabin fingerprinting on strings] Given a string $S \equiv S[0 \dots n-1]$, and two positions $0 \leq i < j \leq n-1$, the longest common extension $\texttt{lce}_S(i,j)$ is the length of the maximal run of matching characters from those positions, namely: if $S[i] \neq S[j]$ then $\texttt{lce}_S(i,j) = 0$; otherwise, $\texttt{lce}_S(i,j) = \max\{\ell \geq 1 : S[i \dots i+\ell-1] = S[j \dots j+\ell-1]\}$. For example, if $S = \texttt{abracadabra}$, then $\texttt{lce}_S(1,2) = 0$, $\texttt{lce}_S(0,3) = 1$, and $\texttt{lce}_S(0,7) = 4$. Given S in advance for preprocessing, build a data structure for S based on the Karp-Rabin fingerprinting, in $O(n \log n)$ time, so that it supports subsequent online queries of the following two types:

   - $\texttt{lce}_S(i,j)$: it computes the longest common extension at positions i and j in $O(\log n)$ time.
   - $\texttt{equal}_S(i,j,\ell)$: it checks if $S[i \dots i+\ell-1] = S[j \dots j+\ell-1]$ in constant time.

   Analyze the cost and the error probability. The space occupied by the data structure can be $O(n \log n)$ but it is possible to use $O(n)$ space. [Note: in this exercise, a one-time preprocessing is performed, and then many online queries are to be answered on the fly.]

4. [Hashing sets] Your company has a database $S \subseteq U$ of keys. For this database, it uses a hash function h uniformly chosen at random from a universal family $\mathcal{H}$ (as seen in class); it also keeps a bit vector B_S of m entries, initialized to zeroes, which are then set $B_S[h(k)] = 1$ for every $k \in S$ (note that collisions may happen). Unfortunately, the database S has been lost, thus only B_S and h are known, and the rest is no more accessible. Now, given $k \in U$, how can you establish if k was in S or not? What is the probability of error? [Note: you are not choosing k and S randomly as the they are both given... randomization here is in the choice of $h \in \mathcal{H}$ performed when building B_S.]

   Under the hypothesis that $m \geq c\,|S|$ for some $c > 1$, find the expected number of 1s in B_S under a uniform choice at random of $h \in \mathcal{H}$.

5. [Family of uniform hash functions] The notion of pairwise independence says that, for any $x_1 \neq x_2$ and $c_1, c_2 \in Z_p$, we have that

$$\Pr_{h \in \mathcal{H}}[h(x_1) = c_1 \wedge h(x_2) = c_2] = \Pr_{h \in \mathcal{H}}[h(x_1) = c_1] \times \Pr_{h \in \mathcal{H}}[h(x_2) = c_2]$$

   In other words, the joint probability is the product of the two individual probabilities. Show that the family of hash functions $\mathcal{H} = \{h_{ab}(x) = ((ax + b) \texttt{ mod } p) \texttt{ mod } m : a \in Z_p^*, b \in Z_p\}$ (seen in class) is "pairwise independent", where p is a sufficiently large prime number ($m + 1 \leq p \leq 2m$).

6. [Deterministic data streaming] Consider a stream of n items, where items can appear more than once. The problem is to find the most frequently appearing item in the stream (where ties are broken arbitrarily if more than one item satisfies the latter). For any fixed integer $k \geq 1$, suppose that only k items and counters can be stored, one item per memory cell, where each counter can use only $O(\text{polylog}(n))$ bits (i.e. $O(\log^c n)$ for any fixed constant $c > 0$): in other words, only $b = O(k \,\text{polylog}(n))$ bits of space are available. (Note that, even though we call them counters, they can actually contain any kind of information as long as it does not exceed that amount of bits.)

   Show that the problem cannot be solved deterministically under the following rules: the algorithm can only use b bits, and read the next item of the stream, one item at a time. You, the adversary, have access to all the stream, and the content of the b bits stored by the algorithm: you cannot change those b bits and the past, namely, the items already read by the algorithm, but you can change the future, namely, the next item to be read. Since the algorithm must be correct for any input, you can use any amount of streams to be fed to the algorithm and as many distinct items as you want. [Hint: it is an adversarial argument based on the fact that, for many streams, there can be a tie on the items.]

7. [Special case of most frequent item in a stream] Suppose to have a stream of n items, so that one of them occurs $> n/2$ times in the stream. Also, the main memory is limited to keeping just $O(1)$ items and their counters, plus the knowledge of the value of n beforehand. Show how to find deterministically the most frequent item in this scenario (see Problem 6 for the general case). [Hint: since the problem cannot be solved deterministically if the most frequent item occurs $\leq n/2$ times, the fact that the frequency is $> n/2$ should be exploited.]

8. [Count-min sketch: extension to negative counters] Check the analysis seen in class, and discuss how to allow $F[i]$ to change by arbirary values read in the stream. Namely, the stream is a sequence of pairs of elements, where the first element indicates the item i whose counter is to be changed, and the second element is the amount v of that change (v can vary in each pair). In this way, the operation on the counter becomes $F[i] = F[i] + v$, where the increment and decrement can be now seen as $(i, 1)$ and $(i, -1)$.

9. [Count-min sketch: range queries] Show and analyze the application of count-min sketch to range queries (i, j) for computing $\sum_{k=i}^{j} F[k]$. Hint: reduce the latter query to the estimate of just $t \leq 2 \log n$ counters $c_1, c_2, \ldots, c_t$. Note that in order to obtain a probability at most δ of error (i.e. that $\sum_{l=1}^{t} c_l > \sum_{k=i}^{j} F[k] + 2\epsilon \log n ||F||$), it does not suffices to say that it is at most δ the probability of error of each counter c_l: while each counter is still the actual wanted value plus the residual as before, it is better to consider the sum V of these t wanted values and the sum X of these residuals, and apply Markov's inequality to V and X rather than on the individual counters.

10. [Space-efficient perfect hash] Consider the two-level perfect hash tables presented in [CLRS] and discussed in class. As already discussed, for a given set of n keys from the universe U, a random universal hash function $h : U \to [m]$ is employed where $m = n$, thus creating n buckets of size $n_j \geq 0$, where $\sum_{j=0}^{n-1} n_j = n$. Each bucket j uses a random universal hash function $h_j : U \to [m]$ with $m = n_j^2$. Key x is thus stored in position $h_j(x)$ of the table for bucket j, where $j = h(x)$.

    This problem asks to replace each such table by a bitvector of length n_j^2, initialized to all `0`s, where key x is discarded and, in its place, a bit `1` is set in position $h_j(x)$ (a similar thing was proposed in Problem 4 and thus we can have a one-side error). Design a space-efficient implementation of this variation of perfect hash, using a couple of tips. First, it can be convenient to represent the value of the table size in unary (i.e., x zeroes followed by one for size x, so `000001` represents $x = 5$ and `1` represents $x = 0$). Second, it can be useful to employ a rank-select data structure that, given any bit vector B of b bits, uses additional $o(b)$ bits to support in $O(1)$ time the following operations on B:

    - $\texttt{rank}_\texttt{1}(i)$: return the number of `1`s appearing in the first i bits of B.
    - $\texttt{select}_\texttt{1}(j)$: return the position i of the jth `1`, if any, appearing in B (i.e. $B[i] = \texttt{1}$ and $\texttt{rank}_\texttt{1}(i) = j$).

    Operations $\texttt{rank}_\texttt{0}(i)$ and $\texttt{select}_\texttt{0}(j)$ can be defined in the same way as above. Also, note that $o(b)$ stands for any asymptotic cost that is smaller than $\Theta(b)$ for $b \to \infty$.

11. [Bloom filters vs. space-efficient perfect hash] Recall that classic Bloom filters use roughly $1.44 \log_2(1/f)$ bits per key, as seen in class (where $f = (1 - p)^k$ is the failure probability minimized for $p \approx e^{-\frac{kn}{m}} = 1/2$). The problem asks to extend the implementation required in Problem 10 by employing an additional random universal hash function $s : U \to [m]$ with $m = \lceil 1/f \rceil$, called signature, so that $s(x)$ is also stored (in place of x, which is discarded). The resulting space-efficient perfect hash table T has now a one-side error with failure probability of roughly f, as in Bloom filters: say why. Design a space-efficient efficient implementation of T, and compare the number of bits per key required by T with that required by Bloom filters.

12. [MinHash sketches] As discussed in class, for a min-wise independent family $\mathcal{H}$, we can associate a sketch $s(X) = \langle \min h_1(X), \min h_2(X), \ldots, \min h_k(X) \rangle$ with each set X in the given data collection, where $h_1, h_2, \ldots, h_k$ are indepently chosen at random from

$\mathcal{H}$. Consider now any two sets A and B, with their sketches $s(A)$ and $s(B)$. Can you compute a sketch for $A \cup B$ using just $s(A)$ and $s(B)$ in $O(k)$ time? Can you prove that it is equivalent to compute $s(A \cup B)$ from stratch directly from $A \cup B$?

13. [Randomized min-cut algorithm] Consider the randomized min-cut algorithm discussed in class. We have seen that its probability of success is at least $1/\binom{n}{2}$, where n is the number of its vertices.

    - Describe how to implement the algorithm when the graph is represented by adjacency lists, and analyze its running time. In particular, a contraction step can be done in $O(n)$ time.
    - A weighted graph has a weight $w(e)$ on each edge e, which is a positive real number. The min-cut in this case is meant to be min-weighted cut, where the sum of the weights in the cut edges is minimum. Describe how to extend the algorithm to weighted graphs, and show that the probability of success is still $\geq 1/\binom{n}{2}$. [hint: define the weighted degree of a node]
    - Show that running the algorithm multiple times independently at random, and taking the minimum among the min-cuts thus produced, the probability of success can be made at least $1 - 1/n^c$ for a constant $c > 0$ (hence, with high probability).