Solutions - Problem Set #3

Alice Bob (uni). Collaborators: A. Turing. - uni@columbia.edu

October 11, 2019

1 Problem 1

1. We modify the Quicksort algorithm to design a randomized algorithm for fuzzy-sorting intervals. Specifically, suppose we choose interval $[a_k, b_k]$ to be the "pivot". We can then segment the intervals into three groups based on this pivot. Let these groups be:

$$\begin{aligned} A &= \{[a_j, b_j] \mid b_j < a_k\} \\ B &= \{[a_j, b_j] \mid a_j \leq a_k \leq b_j\} \\ C &= \{[a_j, b_j] \mid a_k < a_j\} \end{aligned}$$

   Note that the set B contains the interval $[a_k, b_k]$. It is clear that all intervals in the set B have the point a_k in common. Also, each interval in the set A contains a point smaller than a_k and each interval in the set C contains a point larger than a_k. This suggests that we recursively solve the problem for the sets A and C (the divide phase). The conquer phase consists of the following steps:

   (a) Output the permutation for the solution to recursive subproblem A.

   (b) Output the intervals in B in any arbitrary order.

   (c) Output the permutation for the solution to recursive subproblem C.

2. The analysis is similar to the analysis for quicksort discussed in class since the algorithm reduces to quicksort on the a_i's when the intervals are pairwise disjoint (meaning there is no overlap between any two intervals).

   Note that we asked for "expected" time here, where the expectation is taken over all choices of the input, i.e., all input permutations of the a_i's are equally likely. This is different from the expectation taken over the random choices made by the algorithm (none since the algorithm is deterministic).

As we showed in class, on a randomly selected permutation of the input, quicksort runs in $\Theta(n \lg n)$, which means that our algorithm for fuzzy sorting also runs in expected $\Theta(n \lg n)$ time.

Next, we analyze the case when all the intervals overlap. If this is the case, then the set A is empty. Also, set C will be of size $n/2$ on average, so that the recurrence becomes

$$T(n) = T(n/2) + \Theta(n).$$

By case 3 of the Master Theorem, we have expected time $T(n) = \Theta(n)$. As before, the expectation is taken over all choices of the input.

2 Problem 2

Consider the following algorithm, which recursively builds a 26-ary tree holding the strings. We start with a list of words L, as well as an index $i \geq 1$, where we assume recursively that every word in L has the same characters in all indices before i. Then, at a particular node v, we execute the following, corresponding to the subroutine SORT(v, L, i):

1. Consider 26 child nodes $v_a, v_b, \ldots, v_z$ of node v.
2. Scan through the list L, and place a word w into $v_{w[i]}$ (if a word has less than i characters, append the word to a global list G).
3. Starting at v_a and ending at v_z, recurse down the non-empty lists with the list of words that were placed into $v_a, \ldots, v_z$ and incrementing i by 1.

We note that the above algorithm will sort the list of words and place them in the global list G. In order to see why the algorithm is correct, note eventually, every word is placed in G, since the length of each word is finite. In addition, if a word comes before another word, and these words disagree first at the ith index, then we will have the first word be placed in G before the recursive call at the ith depth of the tree.

In order to see this algorithm takes $O(m)$ time, we note that the runtime of the sum over all words of the number of times a word is scanned, and since a word of length ℓ is scanned ℓ times (until $i = \ell + 1$), the algorithm will run in $O(m)$ time.

3 Problem 3

1. We draw a comparison decision tree for the problem. The decision tree contains $n + 1$ distinct leafs (each element of A plus the answer "no"). Thus, a lower bound on the number of comparisons any algorithm must make for outputting the position of x in A is $\log(n + 1) = \Omega(\log n)$.

2. We will use the adversarial lower bound argument instead of a decision tree argument as for part 3a (which would give a trivial log(2) lower bound in this case). We can prove that for the decision problem (yes vs. no), we cannot output "yes" without having looked at/found the element x. To prove this, suppose we did. Then an adversary can always change the element x to be some other real-valued element whose position does not change in the array. In this case, the algorithm would still output "yes" since it sees the same comparisons even though the answer is now "no". This means that the lower bound on part (a) (using decision tree argument) is a lower bound for part (b) since we must find the element x in $\Omega(\log n)$ time to say "yes" or output "no".

4 Problem 4

Our first step will be to identify the subproblems that satisfy the optimal-substructure property. Before we frame the subproblem, we make two simplifying modifications to the input:

- We sort L so that the indices in L are in ascending order.
- We prepend the index 0 to the beginning of L and append n to the end of L.

Let $L[i\,..\,j]$ denote a subarray of L that starts from index i and ends at index j. Define the subproblem denoted by (i, j) as "What is the cheapest sequence of breaks to break the substring $S[L[i] + 1\,..\,L[j]]$?" Note that the first and last elements of the subarray $L[i\,..\,j]$ define the ends of the substring, and we have to worry about only the indices of the subarray $L[i + 1\,..\,j - 1]$.

For example, let $L =< 20, 17, 14, 11, 25 >$ and $n = 30$. First, we sort L. Then, we prepend 0 and append n as explained to get $L =< 0, 11, 14, 17, 20, 25, 30 >$. Now, what is the subproblem $(2, 6)$? We obtain a substring by breaking S after character $L[2] = 11$ and character $L[6] = 25$. We ask "What is the cheapest sequence of breaks to break the substring $S[12\,..\,25]$?" We have to worry about only indices in the subarray $L[3\,..\,5] =< 14, 17, 20 >$, since the other indices are not present in the substring.

At this point, the problem looks similar to matrix-chain multiplication (see Section **??**). We can make the first break at any element of $L[i + 1\,..\,j - 1]$.

Suppose that an optimal sequence of breaks σ for subproblem (i, j) makes the first break at $L[k]$, where $i < k < j$. This break gives rise to two subproblems:

- The "prefix" subproblem (i, k), covering the subarray $L[i + 1\,..\,k - 1]$,
- The "suffix" subproblem (k, j), covering the subarray $L[k + 1\,..\,j - 1]$.

The overall cost can be expressed as the sum of the length of the substring, the prefix cost, and the suffix cost.

We show optimal substructure by claiming that the sequence of breaks in σ for the prefix subproblem (i, k) must be an optimal one. Why? If there were a less costly

way to break the substring $S[L[i]+1\,..\,L[k]]$ represented by the subproblem (i,k), then substituting that sequence of breaks in σ would produce another sequence of breaks whose cost is lower than that of σ, which would be a contradiction. A similar observation holds for the sequence of breaks for the suffix subproblem (k,j): it must be an optimal sequence of breaks.

Let $cost[i,j]$ denote the cost of the cheapest solution to subproblem (i,j). We write the recurrence relation for *cost* as

$$cost[i,j] = \begin{cases} 0 & \text{if } j-i \le 1\ , \\ \min\limits_{i<k<j}\Big\{cost[i,k]+cost[k,j]+(L[j]-L[i])\Big\} & \text{if } j-i>1\ . \end{cases}$$

Thus, our approach to solving the subproblem (i,j) will be to try to split the respective substring at all possible values of k and then choosing a break that results in the minimum cost. We need to be careful to solve smaller subproblems before we solve larger subproblems. In particular, we solve subproblems in increasing order of the length $j-i$.

```
BREAK-STRING(n, L)
 1  prepend 0 to the start of L and append n to the end of L
 2  m = L.length
 3  sort L into increasing order
 4  let cost[1..m, 1..m] and break[1..m, 1..m] be new tables
 5  for i = 1 to m − 1
 6        cost[i, i] = cost[i, i + 1] = 0
 7  cost[m, m] = 0
 8  for len = 3 to m
 9        for i = 1 to m − len + 1
10              j = i + len − 1
11              cost[i, j] = ∞
12              for k = i + 1 to j − 1
13                    if cost[i, k] + cost[k, j] < cost[i, j]
14                          cost[i, j] = cost[i, k] + cost[k, j]
15                          break[i, j] = k
16              cost[i, j] = cost[i, j] + L[j] − L[i]
17  print "The minimum cost of breaking the string is " cost[1, m]
18  PRINT-BREAKS(L, break, 1, m)
```

After sorting L, we initialize the base cases, in which $i=j$ or $j=i+1$.

The nested **for** loops represent the main computation. The outermost **for** loop runs for $len=3$ to m, which means that we need to consider subarrays of L with length at least 3, since the first and the last element define the substring, and we need at least one more element to specify a break. The increasing values of *len* also ensures that

we solve subproblems with smaller length before we solve subproblems with greater length.

The inner **for** loop on i runs from 1 to $m - len + 1$. The upper bound of $m - len + 1$ is the largest value that the start index i can take such that $i + len - 1 \le m$.

In the innermost **for** loop, we try each possible location k as the place to make the first break for subproblem (i, j). The first such place is $L[i+1]$, and not $L[i]$, since $L[i]$ represents the start of the substring (and thus not a valid place for a break). Similarly, the last valid place is $L[j-1]$, because $L[j]$ represents the end of the substring.

The **if** condition tests whether k is the best place for a break found so far, and it updates the best value in $cost[i, j]$ if so. We use $break[i, j]$ to record that the best place for the first break is k. Specifically, if $break[i, j] = k$, then an optimal sequence of breaks for (i, j) makes the first break at $L[k]$.

Finally, we add the length of the substring $L[j] - L[i]$ to $cost[i, j]$ because, irrespective of what we choose as the first break, it costs us a price equal to the length of the substring to make a break.

The lowest cost for the original problem ends up in $cost[1, m]$. By our initialization, $L[1] = 0$ and $L[m] = n$. Thus, $cost[1, m]$ will hold the optimum price of cutting the substring from $L[1] + 1 = 1$ to $L[m] = n$, which is the entire string.

The running time is $\Theta(m^3)$, and it is dictated by the three nested **for** loops. They fill in the entries above the main diagonal of the two tables, except for entries in which $j = i + 1$. That is, they fill in rows $i = 1, 2, \ldots, m-2$, entries $j = i+2, i+3, \ldots, m$. When filling in entry $[i, j]$, we check values of k running from $i+1$ to $j-1$, or $j-i-1$ entries. Thus, the total number of iterations of the innermost **for** loop is

$$\begin{aligned}
\sum_{i=1}^{m-2} \sum_{j=i+2}^{m} (j - i - 1) &= \sum_{i=1}^{m-2} \sum_{d=1}^{m-i-1} d \quad (d = j - i - 1) \\
&= \sum_{i=1}^{m-2} \Theta((m-i)^2) \quad \text{(equation ??)} \\
&= \sum_{h=2}^{m-1} \Theta(h^2) \quad (h = m - i) \\
&= \Theta(m^3) \quad \text{(equation ??)}
\end{aligned}$$

Since each iteration of the innermost **for** loop takes constant time, the total running time is $\Theta(m^3)$. Note in particular that the running time is independent of the length of the string n.

PRINT-BREAKS$(L, break, i, j)$

```
1  if j − i ≥ 2
2      k = break[i, j]
3      print "Break at " L[k]
4      PRINT-BREAKS(L, break, i, k)
5      PRINT-BREAKS(L, break, k, j)
```

PRINT-BREAKS uses the information stored in *break* to print out the actual sequence of breaks.