Computational Methods in Operations Research: Assignment 2

Raul Astudillo

March 14, 2014

Problem 2.

Consider the following algorithm. We start with the original input array and, while the current array has length at least 2 and its number of a's and z's are different, we obtain a new array as follows: from the two arrays consisting of dropping each of the extreme elements of our current array, keep the one with smallest difference between the number of a's and number of z's (with ties broken arbitrarily). After this process ends, we return the final array whenever its length is greater than 2; if its length is 2, then we return this array if it has exactly one a and one z, and the empty array, otherwise.

We claim that the algorithm described above indeed finds a (contiguous) subarray of maximum size with the same number of a's and z's. In order to see this, let $v_0, \ldots, v_r$ be the sequence of arrays generated by our algorithm and let $\hat{v}$ be a maximal subarray. If $\hat{v} = \emptyset$, the claim is clearly true. Thus, we can assume that $\hat{v} \neq \emptyset$ and let $2m = \text{len}(\hat{v})$.

We will prove by induction that v_i contains a contiguous subarray with at least m a's and m z's; we refer to this condition as $\star$. The statement is clearly true for $i = 0$. Now, assume it is true for some $i \geq 0$. If $\text{len}(v_i) = 2m$, then v_i is itself a maximal subarray, which necessarily implies $i = r$ and we are done; otherwise, at least one of the two subarrays obtained by dropping an extreme element of v_i satisfies $\star$. If one of these subarrays does not satisfy $\star$, one can easily verify that the difference between the number of a's ands z's of this subarray is strictly greater than the one of the other subarray. Hence, v_{i+1} must satisfy $\star$ as well, and this finishes the proof.

Finally, we prove that this algorithm has time complexity $O(n)$. In order to see this, first count the total number of a's and z's in the original subarray; this has complexity $O(n)$. Then, for every step, deducing the number of a's and z's in each extreme subarray has complexity $O(1)$ provided that we know the number of a's and z's in the current array; since there are at most n steps, the complexity of the algorithm is $O(n)$.

Problem 3.

Without loss of generality, we assume that the original array is sorted in increasing order. A slightly modified version of the algorithm we are about to consider, would handle the other case as well.

Algorithm 1

Require: array of the specified form, v; number to be searched, a.

```
l ← 0 and r ← n − 1
while 2 ≤ r − l do
    m ← ⌊(l+r)/2⌋
    if v[m] > v[r] then
        if v[l] ≤ a ≤ v[m − 1] then
            return l + BinarySearch(v[l : m], a)
        else
            l ← m
        end if
    else
        if v[m + 1] ≤ a ≤ v[r] then
            return m + 1 + BinarySearch(v[m + 1 : r + 1], a)
        else
            r ← m
        end if
    end if
end while
for k from l to r do
    if v[k] = a then
        return k
        return
    end if
end for
print 'a is not contained in v.'
```

Now we prove the correctnes of this algorithm, but before we establish some notation: we refer to the first $n - i - 1$ indices of v as the first part of v, the index $n - i$ as the second part, and the last i indices as the third part.

First, assume a is in v and let us analyze the first time the algorithm enters the while loop (assuming it enters at least once). If $v[m] > v[r]$, then m is either in the first or second part of v, and thus the array $v[l : m]$ is sorted in increasing order. Furthermore, if $v[l]] \leq a \leq v[m - 1]$, then necessarilly a is in $v[l : m]$ and we can find it using standard binary search. If, on the other hand, the condition $v[l]] \leq a \leq v[m - 1]$ does not hold, we know for sure that a is not in $v[l : m]$ and we can safely ignore this part of v by making $l \leftarrow m$.

Now suppose $v[m] \leq v[r]$. Then, $v[l] \leq v[m]$ cannot hold, as this would imply that $v[l] = v[r]$, contradicting that v has no duplicated elements. Therefore, $v[l] > v[m]$ and thus m lies in either the second or third part of v. An analogous analysis to the one in the previous shows that either we can find a in $v[m + 1 : r + 1]$ using binary search or we can safely ignore this part of v by making $r \leftarrow m$.

Finally, note that after every iteration, the list $v[l : r+1]$ has either less than 3 elements or the same estructure of the original input list, v. Thus, our analysis remains valid at every iteration of the while loop. Moreover, if a is not found first, the search eventually reduces to a list of length at most 2 in which we check element by element.

The complexity of this algorithm is clearly $O(\log n)$.

Problem 4.

In order to get some intution about choosing an suitable value of K, note that, at least intuitively, small random arrays should be close to be sorted already. Thus, for small arrays, it migth be more appropriate to

consider best-case performance insted of average performance when choosing a sorting algorithm. Said that, recall that the best-case complexity of quick sort and merge sort are $O(n \log n)$ and $O(n)$, respectively. Thus, we should choose K not so large and such that $K < K \log K$. After empirical evaluation, we obtain that for $K = 8$, our algorithm achieves a good performance. See implementation details in repository.

Problem 5.

Consider the slightly modified version of the Bellman-Ford algorithm learned in class:

Algorithm 2 Bellman-Ford

Require: graph $G = (V, E)$ with weights.
  $s = v_0$ `# set vertex source arbitrarily`
  `# Stage 1`
  for v in V **do**
    v.distance $\leftarrow \infty$
  end for
  s.distance $\leftarrow 0$
  `# Stage 2`
  for i from 1 to $|V| - 1$ **do**
    for (u, v) in E **do**
      if v.distance $>$ u.distance + weigth(u, v) **then**
        v.distance $\leftarrow$ u.distance + weigth(u, v)
      end if
    end for
  end for
  `# Stage 3`
  for (u, v) in E **do**
    if v.distance $>$ u.distance + weigth(u, v) **then**
      raise 'A negative cycle was found.'
      return
    end if
  end for
  print 'There are no negative cycles.'

Before proving the correctness of this algorithm, first we prove the following lemma.

Lemma 4.1. *Let $d_v^{(i)}$ be equal to the value of v.distance after iteration i. Then, $d_v^{(i)}$ is at most the minimum of the lengths of a path from s to v with at most i edges. Moreover, if $d_v^{(i)} < \infty$, then it is equal to the length of a path from s to v.*

Proof. The statement is trivially true for $i = 0$. Now suppose it is true for some $i \geq 0$. Note that

$$d_v^{(i+1)} = \min_{u \in N(v)} \{d_u^{(i)} + \text{weigth}(u, v)\},$$

and thus $d_v^{(i+1)}$ is at most the minimum of the lengths of a path from s to v with at most $i + 1$ edges. Furthermore, if the minimum on the right is finite, attained say at node $\bar{u}$, then, since $d_{\bar{u}}^{(i)}$ is the length of a path from s to u, $d_v^{(i+1)} = d_{\hat{u}}^{(i)} + \text{weigth}(\hat{u}, v)$ is the length of a path from s to v. □

Theorem 4.1. *Algorithm 2 determines correctly whether a negative cycle exists.*

Proof. By our previous lemma, we know that exactly before stage 3 of the algorithm, v.distance is at most the minimum of the lengths of a path from s to v with at most $|V| - 1$ edges. In particular, v.distance is at most the minimum of the lengths of any *simple* path from s to v.

Now suppose there are no negative cycles. Then, one can easily verify that there is a simple shortest path, and thus in this case $v.distance$ is indeed at most the minimum of the lengths of *any* path from s to v. In particular, this implies that the condition $v.\text{distance} > u.\text{distance} + \text{weigth}(u, v)$ cannot hold for any edge (u, v), and thus algorithm 3 correctly reports that no negative cycles were found.

Conversely, if the condition $v.\text{distance} > u.\text{distance} + \text{weigth}(u, v)$ holds for some edge, then there is a path from s to v strictly shorter than any simple path from s to v, which, in turn implies that a negative cycle exists. □

Finally, note that the time complexity of this algorithm is determined by the double nested for loop that iterates through all edges, $|V| - 1$ times each, where each iteration has complexity $O(1)$; thus, the time complexity of this algorithm is $O(|V||E|)$. Since we only need store $v.distance$ for every vertex, v, the space complexity is $O(|V|)$.