CS21003 - Tutorial 3 Solution Sketch

August 27, 2018

1. The algorithm maintains the loop invariant that at the start of each iteration of the outer **for** loop, the subarray *A[1 . . j − 1]* consists of the *j − 1* smallest elements in the array *A[1 . . n]*, and this subarray is in sorted order. After the first *n − 1* elements, the subarray *A[1 . . n − 1]* contains the smallest *n − 1* elements, sorted, and therefore element *A[n]* must be the largest element. The running time of the algorithm is $\Theta(n^2)$ for all cases.

```
SELECTION-SORT(A)
n ← length[A]
for j ← 1 to n − 1
    do smallest ← j
       for i ← j + 1 to n
           do if A[i] < A[smallest]
                 then smallest ← i
       exchange A[j] ↔ A[smallest]
```

2. Done in the class -- stable (counting) sort from the least significant digit to the most significant digit.
3. Stable (counting) sort on dd, mm and yyyy, respectively.
4. We need to find n buckets with equal probability, thus each should have an area of $\pi r^2/n$. Each point can be mapped to the bucket as follows:

   Let d denote the distance of the point from the center of the circle, $d \in (0,r)$ and can be computed in O(1) time. Compute $\text{floor}(d^2n/r^2)$ to map to the bucket index.

   Once the elements have been inserted in n buckets in O(n) time, each bucket can now be sorted using a simple algorithm such as insertion sort, and the results can be merged in O(n) expected time.

5. Let S be a sequence of n elements divided into *n/k* subsequences each of length *k* where all of the elements in any subsequence are larger than all of the elements of a preceding subsequence and smaller than all of the elements of a succeeding subsequence.

   Claim: Any comparison-based sorting algorithm to sort s must take worst case. $\Omega(n \lg k)$ time in the worst case.

   Proof First notice that, as pointed out in the hint, we cannot prove the lower bound by multiplying together the lower bounds for sorting each subsequence. That would only prove that there is no faster algorithm that sorts the subsequences independently. This was not what we are asked to prove; we cannot introduce any extra assumptions.

Now, consider the decision tree of height *h* for any comparison sort for *S*. Since the elements of each subsequence can be in any order, any of the *k!* permutations correspond to the final sorted order of a subsequence. And, since there are *n/k* such subsequences, each of which can be in any order, there are $(k!)^{n/k}$ permutations of *S* that could correspond to the sorting of some input order. Thus, any decision tree for sorting *S* must have at least $(k!)^{n/k}$ leaves. Since a binary tree of height *h* has no more than 2^h leaves, we must have $2^h \geq (k!)^{n/k}$ or $h \geq \lg((k!)^{n/k})$. We therefore obtain

$$\begin{aligned} h &\geq \lg((k!)^{n/k} \\ &= (n/k)\lg(k!) \\ &\geq (n/k)\lg((k/2)^{k/2}) \\ &= (n/2)\lg(k/2)\ . \end{aligned}$$

The third line comes from k! having its k/2 largest terms being at least k/2 each. (We implicitly assume here that k is even. We could adjust with ßoors and ceilings if k were odd.)

Since there exists at least one path in any decision tree for sorting S that has length at least (n/2)lg(k/2), the worst-case running time of any comparison-based sorting algorithm for S is Ω(n lg k).

6. Solution
    a. Compare each red jug with each blue jug. Since there are n red jugs and n blue jugs, that will take $\Theta(n^2)$ comparisons in the worst case.
    b. Assume that the red jugs are labeled with numbers 1, 2, . . . , n and so are the blue jugs. The numbers are arbitrary and do not correspond to the volumes of jugs, but are just used to refer to the jugs in the algorithm description. Moreover, the output of the algorithm will consist of n distinct pairs (i, j), where the red jug i and the blue jug j have the same volume.

```
MATCH-JUGS(R, B)
if |R| = 0                                  ▷ Sets are empty
  then return
if |R| = 1                                  ▷ Sets contain just one jug each
  then let R = {r} and B = {b}
       output "(r, b)"
       return
  else r ← a randomly chosen jug in R
       compare r to every jug of B
       B_< ← the set of jugs in B that are smaller than r
       B_> ← the set of jugs in B that are larger than r
       b ← the one jug in B with the same size as r
       compare b to every jug of R − {r}
       R_< ← the set of jugs in R that are smaller than b
       R_> ← the set of jugs in R that are larger than b
       output "(r, b)"
       MATCH-JUGS(R_<, B_<)
```

The procedure MATCH-JUGS takes as input two sets representing jugs to be matched: $R \subseteq \{1, \ldots, n\}$, representing red jugs, and $B \subseteq \{1, \ldots, n\}$, representing blue jugs. We will call the procedure only with inputs that can be matched; one necessary condition is that $|R| = |B|$.

The running time then obeys the recurrence $T(n) = T(n-1) + \Theta(n)$, and the number of comparisons we make in the worst case is $T(n) = \Theta(n^2)$.

7. Solution
   a. QUICKSORT' does exactly what QUICKSORT does; hence it sorts correctly.

      QUICKSORT and QUICKSORT' do the same partitioning, and then each calls itself with arguments A, p, q − 1. QUICKSORT then calls itself again, with arguments A, q + 1, r. QUICKSORT instead sets p ← q + 1 and performs another iteration of its **while** loop. This executes the same operations as calling itself with A, q + 1, r, because in both cases, the Þrst and third arguments (A and r) have the same values as before, and p has the old value of q + 1.

   b. The stack depth of QUICKSORT' will be $\Theta(n)$ on an n-element input array if there are $\Theta(n)$ recursive calls to QUICKSORT'. This happens if every call to PARTITION(A, p, r) returns q = r. The sequence of recursive calls in this scenario is

$$\begin{aligned}
&\textsc{Quicksort}'(A, 1, n)\ , \\
&\textsc{Quicksort}'(A, 1, n-1)\ , \\
&\textsc{Quicksort}'(A, 1, n-2)\ , \\
&\qquad\qquad \vdots \\
&\textsc{Quicksort}'(A, 1, 1)\ .
\end{aligned}$$

      Any array that is already sorted in increasing order will cause QUICKSORT' to behave this way.

   c. The problem demonstrated by the scenario in part (b) is that each invocation of QUICKSORT' calls QUICKSORT' again with almost the same range. To avoid such behavior, we must change QUICKSORT' so that the recursive call is on a smaller interval of the array. The following variation of QUICKSORT' checks which of the two subarrays returned from PARTITION is smaller and recurses on the smaller subarray, which is at most half the size of the current array. Since the array size is reduced by at least half on each recursive call, the number of recursive calls, and hence the stack depth, is $\Theta(\lg n)$ in the worst case. Note that this method works no matter how partitioning is performed (as long as the PARTITION procedure has the same functionality as the procedure given in Section 7.1).

The expected running time is not affected, because exactly the same work is done as before: the same partitions are produced, and the same subarrays are sorted.

```
QUICKSORT''(A, p, r)
while p < r
    do ▷ Partition and sort the small subarray first
       q ← PARTITION(A, p, r)
       if q − p < r − q
         then QUICKSORT''(A, p, q − 1)
              p ← q + 1
         else QUICKSORT''(A, q + 1, r)
              r ← q − 1
```