CSE 222 (ADA) Homework Assignment 1 (Theory)

Deadline : Feb 11 (Friday) 1:15 PM.

The theory assignment has to be done in a team of at most two members, as already selected by you. The solutions are to be typed either as a word document or latex-ed and uploaded as pdf on GC. We shall strictly not accept solutions written in any other form. Remember that both team members need to upload the HW solution on GC. Collaboration across teams or seeking help from any sources other than the lectures, notes and texts mentioned on the homepage will be considered an act of plagiarism

There are 3 problems in this homework.

Problem 1.

(a) (10 points) Describe an algorithm that sorts an input array $A[1 \cdots n]$ by calling a subroutine SQRTSORT(k), which sorts the subarray $A[k+1 \cdots k+\sqrt{n}]$ in place, given an arbitrary integer k between 0 and $n-\sqrt{n}$ as input. (To simplify the problem, assume that $\sqrt{n}$ is an integer.) Your algorithm is ***only*** allowed to inspect or modify the input array by calling SQRTSORT; in particular, your algorithm must not directly compare, move or copy array elements. How many times does your algorithm call SQRTSORT in the worst case? Give pseudocode. Remember that you cannot use anything other than calling the SQRTSORT routine and maybe some loops. No formal proof of correctness is mandatory. But write a few sentences justifying your approach.

Solution.

Algorithm 1 Algorithm for FoolishSort(Array A, length ℓ)

```
1: Let $m = \sqrt{n}$
2: if $\ell == 1$ then
3:     Return $A[1]$
4: else
5:     for $i = 1$ to $\ell - m/2$ in steps of $m/2$ do
6:         SQRTSORT(A, i)
7:     end for
8:     FoolishSort(Array $A$, length $\ell - m/2$)
9: end if
```

I have written the pseudocode recursively but surely one can implement the same thing iteratively by running an outer loop that runs from $\ell = n$ down to $m/2 + 1$ and decreases in steps of $m/2$.

Correctness. (Not required for grading :) We prove that at level ℓ of the recursion, where $\ell = n, n - m/2, n - m, \cdots m/2 + 1$, the execution of the for loop ensures that the elements

$\ell - m/2$ to ℓ are in their correct position. Note that this implies the correctness of the entire algorithm.

Consider the execution of the for-loop. We prove the following statement by induction

For all $i = 1, 1+m/2, 1+m, \cdots \ell - m/2$, after the iteration i, $A[im/2, im/2+1, \cdots (i+1)m/2]$ are the largest elements in the subarray $A[1, \cdots (i+1)m/2]$.

The base case is true for $i = 1$ since SQRTSORT(A, i) correctly sorts the first m elements and hence $A[m/2, \cdots m]$ are indeed the largest in $A[1, 2, \cdots m]$. Now assume an induction hypothesis that the statement is true for any $i = r$. By induction hypothesis, at the end of iteration r, $A[rm/2, (r+1)m/2)]$ were the largest $m/2$ elements in the subarray $A[1, 2, \cdots (r+1)m/2]$ - call this set of elements S. After execution of iteration $r+1$,let us call the set of elements $A[(r+1)m/2, \cdots (r+2)m/2]$ as S' . Now for any elements $s \in S, s' \in S'$, we know $s \leq s'$. Using this and the induction hypothesis, we conclude that s' is larger (or equal) to every element in $A[1, 2, \cdots (r+1)m/2]$.

Runtime. We first calculate the number of calls to SQRTSORT when the length of the array for the recursive call is ℓ. This number is clearly upper bounded by $2\ell/m$. Now we add up over all levels of recursive calls. The total number of calls to SQRTSORT will turn out to be at most (I a being slightly sloppy here)

$$\begin{aligned}
&\sum_{\ell:\ell=im/2, i=1,2,\cdots 2m} \frac{2\ell}{m} \\
&= \frac{2}{m} \sum_{i=1,2,\cdots 2m} \frac{im}{2} \\
&= \sum_{i=1,2,\cdots 2m} i \\
&= \frac{2m(2m+1)}{2} \leq 2m^2 + 1 = O(n)
\end{aligned}$$

(b) (10 points) Prove that your algorithm from part (a) is optimal up to constant factors. In other words, if $f(n)$ is the number of times your algorithm calls SqrtSort, prove that no algorithm can sort using $o(f(n))$ calls to SqrtSort. Note that here we are assuming that these algorithms **cannot do** anything other than calling SqrtSort repeatedly.

(Hint: Think of a typical worst case example for any sorting algorithm)

Solution. Consider any array which is sorted in the reverse order. There are about $\mathcal{O}(n^2)$ many inversions in such an array. Now, one invocation of SQRTSORT() can fix at most $\mathcal{O}(n)$ many inversions. Hence, any algorithm that is allowed to only use SQRTSORT will need $\omega(n)$ many such calls.

(c) (10 points) Now suppose SqrtSort is implemented recursively, by calling your sorting algorithm from part (a). For example, at the second level of recursion, the algorithm is sorting arrays roughly of size $n^{1/4}$. What is the worst-case running time of the resulting sorting algorithm? (To simplify the analysis, assume that the array size n has the form 2^{2^k}), so that repeated square roots are always integers.)

Solution. First some explanation and then the actual solution. This problem seems weird at first since it seems to be stuck in a cyclic argument. The trick is to again realize the power of recursion here. First think of FOOLISHSORT as blackbox and forget about how it is implemented. Well, I can definitely implement SQRTSORT(A,k) by calling FOOLISHSORT($A[k, k+1, k+\sqrt{n}], \sqrt{n}$). The confusing part is that FOOLISHSORT will again call SQRTSORT from inside as implemented in Algorithm 1. But hey ! Thats ok. Why ? Because it is calling SQSORT() recursively on a *smaller input size*, which is $\sqrt{\sqrt{n}} = n^{\frac{1}{4}}$. How many recursive calls ? Ahh ! We know that too from part (a). It's $2\sqrt{n}+1$ (please note that the input size here at the top level of recursion is just $\sqrt{n}$). The rest is easy peasy. The recursion for FoolishSort with input size n is $T(n) = 2nT(\sqrt{n})$ (I am ignoring any other work done outside the recursive calls since they do not matter in this case)

The solution of this recursion can be done by just opening it up and looking at the pattern. For example

$$T(n) = 2nT(\sqrt{n}) = 2n \cdot 2n^{1/2}T(n^{1/4}) = \cdots$$

The only thing that we need to figure out is the number of levels the recursion tree for the above (or equivalently the number of times you will open up the recurrence until you reach an array of size 1). The answer in this case is $\log\log n$ (can you see why?). Hence, the total contribution of the leading constant 2 is $2^{\log\log n} = \log n$ and the rest will boil down to $n^{1+1/2+1/4+1/8+\cdots} \leq n^2$. Hence the overall time complexity of this algorithm is $\mathcal{O}(n^2 \log n)$.

Problem 2. (10 points)

Let $n = 2^l - 1$ for some positive integer l. Suppose someone claims to hold an unsorted array $A[1 \cdots n]$ of *distinct* l-bit strings; thus, exactly one l-bit string does *not* appear in A. Suppose further that the ***only*** way we can access A is by calling the function FETCHBIT(i, j), which returns the j^{th} bit of the string $A[i]$ in $O(1)$ time. Describe an algorithm to find the missing string in A using only $O(n)$ calls to FETCHBIT. Again, give either pseucode or write a generic description. Demonstrating the algorithm on a specific example will not fetch any marks.

Solution: Without loss of generality, let us assume that all strings are binary (every bit is either 0 or 1). Then, we can observe the following types of strings. There are 2^l distinct binary strings of l bits.

Type (a) The number of strings with first bit 0 is 2^{l-1}.

Type (b) The number of strings with first bit 1 is 2^{l-1}.

Then, in the first stage, we call FETCHBIT(i, j) for all $i = 1, \ldots, n$ and for $j = 1$ (check the first bits of all the strings). As there are $2^l - 1$ strings, it must be that for exactly one of the cases above (either Type (a), or Type (b), but not both), there are $2^{l-1} - 1 = \frac{n+1}{2} - 1 \leq n/2$ strings. We focus our attention to only those type, i.e. Type (a) or Type(b) but not both such that there are $2^{l-1} - 1$ many of them. Next, we remove the first bit from all these $2^{l-1} - 1 \leq n/2$ strings and store the first bit of these strings which is either 0 for both or 1 for both. Hence, we have $2^{l-1} - 1$ strings each having $l - 1$ bits. We recursively call the algorithm on these $2^{l-1} - 1$ strings of length $l - 1$. Finally, when we reach a situation when there is only 1 string of 1-bit each, then we know which string is missing. If this particular 1-bit string is 1, then we say that the last bit of the missing string is 0. Otherwise, this particular 1-bit string is 0 and we say that the last bit of this missing string is 1. Finally, we keep collecting the missing bits in the reverse order of the stored values. In this process we identify the missing string.

Observe that first we perform n calls to FETCHBIT(i, j), identify the first bit of the missing string, keep track of it. Then, recursively call the algorithm for at most $n/2$ strings, find the next bit of the missing string, and so on. We reduce our search space from n to $n/2$ and outside the recursive call, we do $O(n)$ operations. Hence, the recurrence relation is

$$T(n) = T(n/2) + O(n)$$

Using Master's theorem, we see that $T(n) = O(n)$.

Problem 3. (10 points)

Use recursion tree to solve the following recurrence.

$$T(n) = T(n/15) + T(n/10) + 2T(n/6) + \sqrt{n}$$

Solution: As $f(n) = \sqrt{n}$, a (partial) recursion tree is given in the following figure for ease of understanding.

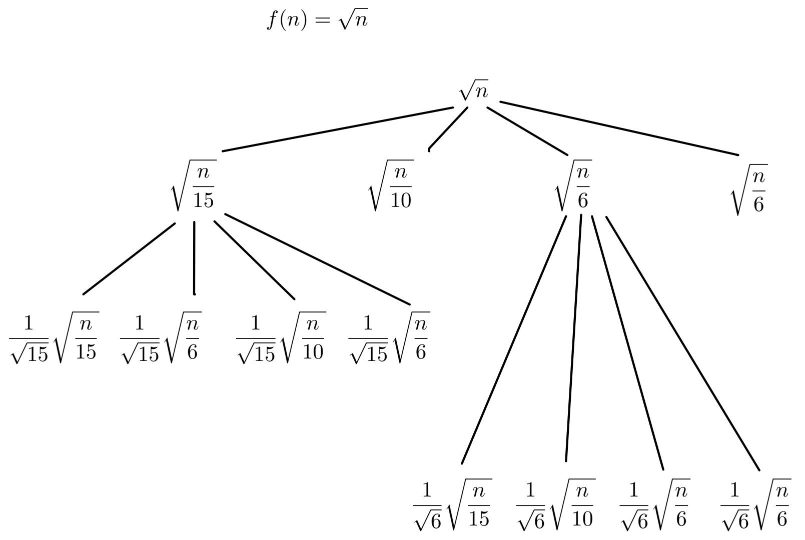
$f(n) = \sqrt{n}$
$\sqrt{n}$
$\sqrt{\frac{n}{15}}$
$\sqrt{\frac{n}{10}}$
$\sqrt{\frac{n}{6}}$
$\sqrt{\frac{n}{6}}$
$\frac{1}{\sqrt{15}}\sqrt{\frac{n}{15}}$
$\frac{1}{\sqrt{15}}\sqrt{\frac{n}{6}}$
$\frac{1}{\sqrt{15}}\sqrt{\frac{n}{10}}$
$\frac{1}{\sqrt{15}}\sqrt{\frac{n}{6}}$
$\frac{1}{\sqrt{6}}\sqrt{\frac{n}{15}}$
$\frac{1}{\sqrt{6}}\sqrt{\frac{n}{10}}$
$\frac{1}{\sqrt{6}}\sqrt{\frac{n}{6}}$
$\frac{1}{\sqrt{6}}\sqrt{\frac{n}{6}}$

Figure 1: Partial Recursion Tree for the recurrence relation

Work done at root level $\sqrt{n} = n^{1/2}$. Work done at Level-1 is

$$\sqrt{n}\left(\frac{1}{\sqrt{15}} + \frac{1}{\sqrt{10}} + \frac{2}{\sqrt{6}}\right)$$

Work done at Level-2 is

$$\sqrt{n}\left(\frac{1}{\sqrt{15}} + \frac{1}{\sqrt{10}} + \frac{2}{\sqrt{6}}\right)^2$$

Work done at Level-k is

$$\sqrt{n}\left(\frac{1}{\sqrt{15}} + \frac{1}{\sqrt{10}} + \frac{2}{\sqrt{6}}\right)^k$$

As the rate of shrinkage of the problem is at least 6 at each level, we can presume that n is a power of 6. Hence, the highest depth is $\log_6 n$. When $k = \log_6 n$, then the work done at level $\log_6 n$ is

$$\sqrt{n}\left(\frac{1}{\sqrt{15}} + \frac{1}{\sqrt{10}} + \frac{2}{\sqrt{6}}\right)^{\log_6 n}$$

Let $\frac{1}{\sqrt{15}} + \frac{1}{\sqrt{10}} + \frac{2}{\sqrt{6}} = a \approx 1.39$.

We have the following

$$\sqrt{n}(1 + a + a^2 + a^3 + \ldots + a^{\log_6 n}) = \sqrt{n}(1 + a + a^2 + \ldots + a^{\log_6 n})$$

$$= \sqrt{n}\left(\frac{a^{\log_6 n} - 1}{a - 1}\right) = \sqrt{n}n^{\log_6 a} \approx (\sqrt{n}/0.39)n^{\log_6 1.39} \approx n^{0.5}n^{0.1836} \approx n^{0.6836}$$

Hence, $T(n) \leq O(n^{0.6836})$.