CSCI-GA.1170-001/002 Fundamental Algorithms October 3, 2014

Solutions to Problem 1 of Homework 4 (16 points)

Name: Sahil Goel *Due: Wednesday, October 1*

We give the following procedure StrangeSort to sort an array A of n distinct elements.

- Divide A into $n/3$ subsets of size 3 each
- Compute the median of each set to get $n/3$ elements
- Sort these $n/3$ elements recursively using StrangeSort and get p as the median of these $n/3$ elements
- Use p as a pivot in the Partition procedure of Quicksort, i.e., divide A into sets with elements less than p (call $A_{<p}$), those with elements equal to p, and those with elements greater than p (call $A_{>p}$)
- Recursively call StrangeSort on $A_{<p}$ and $A_{>p}$.

(a) (4 points) Give a lower bound on the number of elements in $A_{<p}$ and $A_{>p}$.

Solution: At the end, we" have n/3 elements, medians from every n/3 groups. When we take median of these n/3 elements, there will be atleast n/6 elements lesser than the median and n/6 greater than the median values. Further these n/6 elements lesser than median had each one more element less than them(because we took median at the first step). Therefore total number of elements that are atleast less than pivot are equal to $n/6 + n/6 = n/3$
Similarly total number of elements atleast greater than pivot will be equal to n/3

$A_{<p} = n/3$ and $A_{>p} = n/3$

□

(b) (2 points) What is the recurrence relation you obtain for the time complexity $T(n)$ of StrangeSort assuming that the position of p corresponds to the lower bound found in part (a)?[1]

Solution: $T(n) = T(n/3) + T(partition) + T(n/3) + T(2n/3)$
$T(n) = 2 * T(n/3) + T(2n/3) + O(n)$

□

(c) (4 points) Try to use induction to show that $T(n) \leq n^{1+c}$ for some (yet undetermined) value of $c > 0$. For simplicity, you may ignore the term corresponding to the (non-recursive) divide-and-conquer step[2] in your recurrence relation, when you do your inductive step. (In other words, only keep all the "T-terms" and drop the "$f(n)$"-terms when your prove your inductive step.) What is the inequality that c should satisfy in order for the induction to work?

[1]It is easy to see that this is the worst case, but you do not have to show this.
[2]I.e., the $n/3$ median findings on 3 elements and the run of the Partition procedure around p.

Solution: To prove $T(n) = O(n^{1+c})$
and $T(n) = 2 * T(n/3) + T(2 * n/3)$

for n=1
$T(1) = 2 * T(0) + T(0) = 0 \leq n^{1+c}$
$0 \leq 1^{1+c}$
Therefore it is true for every value of c.
Now assuming it is true upto n-1, we have to prove it is true for n
$T(n) = 2 * T(n/3) + T(2 * n/3) \leq n^{1+c}$
$2 * ((n/3)^{1+c}) + ((2n/3)^{1+c}) \leq n^{1+c}$
$2 * n^{1+c}/3^{1+c} + 2^{1+c} * n^{1+c}/3^{1+c} \leq n^{1+c}$
$2/3^{1+c} + 2^{1+c}/3^{1+c} \leq 1$
$2 + 2^{1+c} \leq 3^{1+c}$
$2 \leq 3^{1+c} - 2^{1+c}$
This is the inequality that c should satisfy

□

(d) (2 points) Use `http://www.wolframalpha.com` to find the smallest possible c (upto two decimal places) for which the "nasty" inequality in part (c) is satisfied. How does the resulting running time of StrangeSort compare to the worst and average-case running times of QuickSort?

Solution: $c \geq .394955$
Therefore $T(n) = O(n^{1.394955})$
Worst case time of Quicksort = n^2 and Average time =$n \log n$ Therefore T(n) performs much better than worst case time of Quicksort but since $n \log n < n^{1.394955}$ for large n, therefore T(n) performs slower than the average case time for Quicksort
$T(Strangesort) < T(WorstQuicksort)$
$T(Strangesort) > T(AvgQuicksort)$

□

(e) (4 points) How does your answer from part (d) change if you replace sets of size 3 by sets of size 5 in the StrangeSort algorithm.

Solution: if we replace the size of sets from 3 to 5, then the new inequality for c will be
$2 \leq 5^{1+c} - 2^{1+c}$
$C = -.1752$
$T(n) = O(n^{.8248})$
Now $T(Strangesort) < T(WorstQuicksort)$
$T(Strangesort) < T(AvgQuicksort)$

□

CSCI-GA.1170-001/002 Fundamental Algorithms October 3, 2014

Solutions to Problem 2 of Homework 4 (8 (+1) points)

Name: Sahil Goel *Due: Wednesday, October 1*

Assume we are given an array $A[1 \ldots n]$ of n *distinct* integers and that $n = 2k+1$ is *odd*.

(a) (3 points) Let $pivot(A)$ denote the rank of the pivot element at the end of the partition procedure, and assume that we choose a random element $A[i]$ as a pivot, so that $pivot(A) = i$ with probability $1/n$, for all i. Let $smallest(A)$ be the length of the smaller sub-array in the two recursive subcalls of the QUICKSORT. Notice, $smallest(A) = \min(pivot(A) - 1, n - pivot(A))$ and belongs to $\{0 \ldots k\}$, since $n = 2k+1$ is odd. Given $0 \le j \le k$, what is the probability that $smallest(A) = j$?

Solution: After finding out the pivot rank, 2 recursive calls can have subarrays elements split like

Subarray1	Subarray2
0	2k
1	2k-1
2	2k-2
.	.
.	.
.	.
.	.
k	k
k+1	k-1
.	.
.	.
.	.
2k	0

There are 2k+1 total rows and every row has probability of 1/(2k+1)
Smallest varies from $0, 1, 2, \ldots, k-1, k$ respectively
Every element till 0 to k-1 appears exactly in 2 rows whereas k only appears in one row
That's why
if $j = k$ then $P = \frac{1}{2k+1}$
otherwise $P = \frac{2}{2k+1}$ □

(b) (2 points) Compute the *expected value* of $smallest(A)$; i.e., $\sum_{j=0}^{k} \Pr(smallest(A) = j) \cdot j$. (**Hint**: If you solve part (a) correctly, no big computation is needed here.)

Solution: $Expectedvalue = [\frac{2}{2k+1} * 0 + \frac{2}{2k+1} * 1 + \ldots + \frac{2}{2k+1} * (k-1) + \frac{1}{2k+1} * k]$

$E = [\frac{2}{2k+1} * [1 + 2 + 3 + \ldots + k - 1] + \frac{1}{2k+1} * [k]$

$E = [\frac{2}{2k+1} * [k * (k-1)/2] + \frac{k}{2k+1}]$

$E = [\frac{k*(k-1)}{2k+1} + \frac{k}{2k+1}]$

$E = \frac{k^2}{2k+1}$
Therefore expected value $= \frac{k^2}{2k+1}$
for very large k, expected value $E = k/2$

$\square$

(c) (3 points) Write a recurrence equation for the running time $T(n)$ of Quicksort, assuming that at every level of the recursion the corresponding sub-arrays of A are partitioned *exactly* in the ratio you computed in part (b). Solve the resulting recurrence equation. Is it still as good as the average case of randomized Quicksort?

Solution: $T(2k+1) = T(\frac{k^2}{2k+1}) + T(2k - \frac{k^2}{2k+1}) + (2k+1)$

Replacing $2k + 1 = n$

$T(n) = T(\frac{(n-1)^2}{4n}) + T(\frac{3*n^2-2*n-1}{4n}) + n$

$T(n) = T(\frac{n^2+1-2n}{4n}) + T(\frac{3*n^2-2n-1}{4n}) + n$

For very large n, $n^2 >> 1 - 2n$ and $3 * n^2 >> -2n - 1$

Therefore $T(n) = T(\frac{n^2}{4n}) + T(\frac{3n^2}{4n}) + n$

$T(n) = T(\frac{n}{4}) + T(\frac{3n}{4}) + n$
Using recursion tree

$n \log_4 n < T(n) < n \log_{4/3} n$
$\frac{n \log n}{\log 4} < T(n) < \frac{n \log n}{\log 4/3}$
Therefore $T(n) = O(n \log n)$
It is as good as the average case of randomized Quicksort

□

(d)* (**Extra Credit**; 1 point) The analysis in parts (a)-(c) was much simpler than the analysis of randomized QUICKSORT we did in class. On the other hand, it *seems* that it also computes the expected running time of QUICKSORT when the pivot is chosen at random. Indeed, it seems that since we expect the array to be split according the ratio found in part (b) in every subcall, the recursive equation in part (c) was justified. Why is this argument incorrect?

Solution: The array is not split according to part c every time. There will be sub arrays which may have split perfectly evenly or perfectly unevenly (like n,0 and n/2,n/2), therefore we can't use the equation from part c to calculate the exact running time of algorithm and always divide n as 3n/4 and n/4. Whereas in class, we considered all the cases and then calculated the expected average time.

□

CSCI-GA.1170-001/002 Fundamental Algorithms October 3, 2014

Solutions to Problem 3 of Homework 4 (13 points)

Name: Sahil Goel *Due: Wednesday, October 1*

Given a heap A of size n, let $pos = pos(A, i, n)$ denote the number of positive elements in the sub-heap of A rooted at i. Consider the following recursive procedure that computes pos:

```
1 PositiveCount($A, i, n$)
2       If $i > n$ Return 0
3       If $A[i] \leq 0$ Return 0
4       $pos = 1 +$ PositiveCount($A$, Left($i$), $n$)
5       $pos = pos +$ PositiveCount($A$, Right($i$), $n$)
6       Return $pos$
```

(a) (5 points) Prove correctness of the above algorithm. Make sure to explain the meaning of each line 2-5. Then argue that the algorithm above runs in time $O(pos)$.

Solution:
Line 2: It checks whether i is greater than total number of elements in the heap. If it is then heap processing is complete and it has to return, that's why it has returned 0.

Line 3: It checks if the element at index i is positive or not. if it is not positive, then all of its children will also be negative because of property of heap(i.e. All of children are smaller than the parent). Therefore, after finding the first non positive element, rest of that branch should be ignored along with the element at index i and hence it returned 0.

Line 4: If the element is positive, this line is executed. It increments the total number of positive elements by 1(to count the element at index i) and then recursively call the procedure by passing its left child to the function (i.e. left subtree) to calculate number of positive elements in left subtree and add it to the current pos value.

Line 5: This line is also executed only if the element at index i was positive. This calls the procedure recursively for the right branch by passing the right child to the function. the function will eventually return the positive elements in the right sub tree and then it is added to the previous total number of positive elements to give total number of positive elements.

Tracing the algorithm:

Initially the parent node is checked if it is positive or not. If it is positive, pos(number of pos. elements) is incremented by 1 and then procedure is called recursively for left and right subtree.
Suppose T is a tree having P as Parent(root) node, L as left subtree and R as right subtree. If P is positive then pos=1 for P node. Total number of positive elements will be 1 + (Total

number of elements in left subtree) + (Total number of elements in right subtree). To calculate the total number of elements in left subtree, we have called this procedure recursively by passing the left child of P(which will now be parent for next round). Similar thing is done for right subtree. Since i is always increasing, it will eventually become greater than n or the elements will become negative, therefore the function will return 0 at some point hence the program will exit.

Now to calculate the complexity, we can see that all the positive elements are at least traversed once. Lets call the total number of positive elements as pos. There will be atleast pos number of comparisons to check if its positive or not. Therefore total complexity will be O(pos)

□

(b) (8 points) Assume now that we do not really care about the exact value of *pos* when $pos > k$; i.e., if the heap contains more than k positive elements, for some parameter k. More formally, you wish to write a procedure kPositiveCount(A, i, n, k) which returns the value $\min(pos, k)$, where $pos =$ PositiveCount(A, i, n).

Of course, you can implement kPositiveCount(A, i, n, k) by calling PositiveCount(A, i, n) first, but this will take time $O(pos)$, which could be high if $pos \gg k$. Show how to (slightly) tweak the pseudocode above to *directly* implement kPositiveCount(A, i, n, k) (instead of PositiveCount(A, i, n)) so that the running time of your procedure is $O(k)$, irrespective of *pos*. Make sure you explicitly write the pseudocode of you new recursive algorithm (which should be similar to the one given above), prove its correctness, and argue the $O(k)$ run time. (**Hint**: There is a reason we did not consolidate lines 4. and 5. of our pseudocode to a single line 4.5: $pos = 1 +$ PositiveCount$(A,$ Left$(i), n) +$ PositiveCount$(A,$ Right$(i), n)$.)

Solution:

```
1 KPositiveCount($A, i, n, k$)
2     **If** $i > n$ **Return** 0
3     **If** $A[i] \leq 0$ **Return** 0
4     $pos = 1 +$ KPositiveCount($A$, Left($i$), $n, k$)
5     $if(pos >= k)$ **Return** $k$
6     $pos = pos +$ KPositiveCount($A$, Right($i$), $n, k$)
7     **Return** $min(pos, k)$
```

Tracing the algorithm
Initially pos is zero and is incremented by 1 whenever a positive element is encountered. Then recursively left subtree is called. In line 5 of code, we check if the value of pos has become greater than k or not. If it has become more than k, then we don't need to check other elements and we can return k at this time because pos $\geq$ k and min(pos,k) = k. Otherwise we have called the recursive procedure to the right side of the tree (i.e. right subtree). After processing right subtree, it checks if vale of pos has become greater or not. If value of pos

$\geq$ k then it will return k otherwise pos. As we can observe, if k¡¡pos, whole tree will not be traversed and the function will return very early (because of line 5) and the number of elements traversed will be O(k) as compared to O(pos). If there is a case when pos is comparable to k, then O(k) = O(pos). If $k >> pos$ then the total number of elements processed will be pos and the complexity will be O(pos).

$\square$

CSCI-GA.1170-001/002 Fundamental Algorithms October 3, 2014

Solutions to Problem 4 of Homework 4 (12 points)

Name: Sahil Goel *Due: Wednesday, October 1*

We wish to implement a data structure D that maintains the k smallest elements of an array A. The data structure should allow the following procedures:

- $D \leftarrow$ Initialize(A, n, k) that initializes D for a given array A of n elements.
- Traverse(D), that returns the k smallest elements of A *in sorted order*.
- Insert(D, x), that updates D when an element x is inserted in the array A.

We can implement D using one of the following data structures: (i) an unsorted array of size k; (ii) a sorted array of size k; (iii) a max-heap of size k.

(a) (4 points) For each of the choices (i)-(iii), show that the Initialize procedure can be performed in time $O(n + k \log n)$.

Solution:

Steps :-
1. Find the kth minimum element in array.
2. Copy all the elements which are <= k to D.
3_1. (For Sorted arrays). Sort these k elements.
3_2. (For Heap). Buildheap for these k elements

```
Findkminimum(A,k)
{
$Pivot = A[r]$                where r is randomly chosen between 1 to n

for i=1:n
if($A[i] < Pivot$)
Append A[i] to A1
else if($A[i] > Pivot$)
Append A[i] to A2
else
Do nothing
end Loop

% Now, all elements in A1 are less than Pivot and all elements in A2 are greater than pivot

lenA1=length(A1);
lenA2=length(A2);
if($lenA1 > k$)                  %kth minimum element must lie in A1
Findminimum(A1,k);
else if($lenA1 < k - 1$)          %kth minimum element must lie in A2 (-1 because of pivot)
Findminimum(A2,k-lenA1-1)       % lenA1 smaller elements in A1 and 1 pivot
else
return Pivot                    %kth minimum element is pivot
}
```

To calculate the complexity of this procedure
$T(n) = T(max(q, n - q)) + (n)$
Second term i.e. T(max(q,n-q)) will vary between T(n-1), T(n-2), T(n-3) , ..., T(n/2).
To calculate the expected value

$$T(n) = 2 * \frac{T(n-1)+T(n-2)+T(n-3)+\ldots+T(n/2)}{n} + (n)$$

$$T(n) = \frac{2}{n} * [T(n-1) + T(n-2) + T(n-3) + \ldots + T(n/2)] + (n)$$

$$n * T(n) = 2 * [T(n-1) + T(n-2) + T(n-3) + \ldots + T(n/2)] + (n)$$

$$(n-1) * T(n-1) = 2 * [T(n-2) + T(n-3) + \ldots + T((n)/2)] + (n-1)$$

Subtracting two equations
$n * T(n) - (n-1) * T(n-1) = 2 * [T(n-1)] + 1$

$n * T(n) = (n+1) * T(n-1) + 1$
Dividing equation by (n)*(n+1)

$T(n)/n + 1 = T(n-1)/n + 1/[(n)(n+1)]$

Let $S(n) = T(n)/n + 1$

$S(n) = S(n-1) + \frac{1}{n*(n+1)}$

$S(n) = \frac{1}{1*2} + \frac{1}{2*3} + \ldots + \frac{1}{n*(n+1)}$

$S(n) = [1 - 1/2] + [1/2 - 1/3] + [1/3 - 1/4] + \ldots + [1/n - 1/(n+1)]$
$S(n) = \frac{n}{n+1}$
$T(n) = S(n) * (n+1)$
$T(n) = n$
therefore T(n) takes O(n) time

Step 2
Traverse through the whole array and copy all elements which are less than kth minimum to new array D. It takes O(n) time.

Therefore total time for unsorted array of size k will be O(n) + O(n) = O(n)

Step 3_1
Apply quicksort or mergesort algorithm on these k elements array, which will take $O(k \log k)$ time.
Therefore total time taken will be $O(n) + O(n) + O(k \log k)$
$T(n) = O(n + k \log k)$

Step 3_2
Call buildmaxheap for these k elements
Buildheap will take time $k \log k$
$T(n) = O(n) + O(k \log k)$
$T(n) = O(n + k \log k)$

□

(b) (3 points) For each of the choices (i)-(iii), compute the best running time for the TRAVERSE procedure you can think of. (In particular, tell your procedure.)

Solution:
Unsorted array
Traverse procedure will take $k * \log k$ time. To return all the elements in sorted order, we will first have to sort these k elements and then return them which will take $O(k * \log k)$ time

Sorted array
Since the array is already sorted, returning them will take O(1) time, since we just have to pass the pointer to the first address of array

MaxHeap
For max heap time taken will be $O(k * \log k)$
Procedure:

```
1. $Temparray[size - k] = 0$
2. For i=1:k
Temparray[k+1-i] = ExtractMax(D,k+1-i)
3. end loop
return Temparray;
```

ExtractMax function returns the maximum value and call maxHeapify to update the heap again

□

(c) (5 points) For each of the choices (i)-(iii), compute the best running time for the INSERT procedure you can think of. (In particular, tell your procedure.)

Solution:

Unsorted array :- O(k)
For every new element inserted in A, we will find out the maximum of D(unsorted array), if the new element is less than the maximum element of D, then we replace max element of D by new elements. To find out the maximum it takes O(k) time.

Sorted array :- O(k)
For every new elements inserted in A, we will find out if element is greater than maximum of D or not. If its greater then nothing to do else we have to find the correct position for new element. We can do this in 2 ways (1) By insertion sort, which will take O(k) time, (2) By binary search first and then shifting the corresponding number of elements to right which will again take O(k) time, just to shift those numbers.

Maxheap :- $O(\log k)$
For every new element inserted in A, we will first call Max(D) and compare it with the new element. If it is greater than Max(D) then nothing has to be changed otherwise, we will insert it into top of heap and reorder heap(MAXHEAPIFY) to shift it to its correct position, which will take $O(\log k)$ time. $\square$