CSCI-GA.1170-001/002 Fundamental Algorithms October 1, 2014

Solutions to Problem 1 of Homework 4 (16 points)

Name: Keeyon Ebrahimi *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:

Distinction: If $\frac{n}{3}$ is even. Then when calculating median p, we are going to be using the lesser of the two middle values, instead of adding them together and dividing by two for the median. We are doing this because we need to have p be an actual element from Array A.

With this distinction in mind we can come up with the lower bound of $A_{<p}$ and $A_{>p}$

Lower Bound of $A_{>p}$:
If $\frac{n}{3}$ is odd, lower bound is $\frac{n}{3}$.
If $\frac{n}{3}$ is even, lower bound is $\frac{n}{3}+1$

If $\frac{n}{3}$ is odd, this is how we can imagine the lower bound. In the ideal case, we can have $\frac{\frac{n}{3}}{2}$ subdivisions that have their midpoints less than p have all their values be less than p, so they are out.

We now examine the subdivision that has p. The least amount of elements this subdivision can have above p is one, for the p element and the one lower than the p element are out. So far we have 1.

Now we examine half the subdivisions that have a midpoint above p. There are a total of $\frac{\frac{n}{3}}{2}-1$ of these subdivisions. The -1 exists because that is the subdivision that contains p. These subdivisions have 2 elements each that are above p. This means that the minimum

amount of elements in $A_{<p}$ is $(\frac{\frac{n}{3}-1}{2}) * 2 + 1$ which equals $\frac{n}{3}$.

If $\frac{n}{3}$ is even, we use very similar logic. All the subdivisions with a midpoint less than p, we assume have all elements with values less than p, so they are not evaluated.

Once again the subdivision with p has 1 element with a value greater than p.

Because of our distinction stated at the start of this solution, if we have an even amount of subdivisions, then half of the subdivisions have a midpoint above p. These subdivisions all have 2 elements greater than p, which are the midpoint and the point above. This means that with even subdivisions, the minimum amount of elements is $A_{<p}$ is $(\frac{\frac{n}{3}}{2}) * 2 + 1$, which is just $\frac{n}{3} + 1$

Lower Bound of $A_{<p}$:
If $\frac{n}{3}$ is even, lower bound is $\frac{\frac{n}{3}}{2} - 1$.
If $\frac{n}{3}$ is odd, lower bound is $\frac{n}{3}$
We are using the same logic as the above example.

If $\frac{n}{3}$ is even, half the subdivisions with medians larger than p, we can assume have all their values above p, we we rule them out.

The subdivision that has p has one element less than p, meaning we currently have 1.

There are $\frac{\frac{n}{3}}{2} - 1$ divisions with a midpoint less than p. These elements have 2 values less than p. So in total, including the one element from the subdivision with p, we have $(\frac{\frac{n}{3}}{2} - 1)2 + 1$ subdivisions which reduces to

$$\frac{\frac{n}{3}}{2} - 1$$

If $\frac{n}{3}$ is odd, we have the exact same math as if it is odd for the $A_{<p}$ case, so refer to that to find the correct reasoning behind that.

□

(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: $T(n) = 3T(\frac{n}{3}) + n$

This is because we have 3 recursive calls of size $\frac{n}{3}$. The finding medians also takes n time, so we get $T(n) = 3T(\frac{n}{3}) + n$ □

[1] It is easy to see that this is the worst case, but you do not have to show this.

(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?

Solution:
We need to prove. $T(n) \leq n^{1+c}$ $T(n) \leq n^{1+c}$ and $T(n) = 3T(\frac{n}{3}) + n$ so If we us n^{1+c} as our guess, when replace everything from the first equation into the second equation, we get

1. $T(n) = 3(\frac{n^{1+c}}{3}) + n$
2. $T(n) = n^{1+c} + n$

   Unfortunately, this next statement is not true.
3. $n^{1+c} + n \leq n^{1+c}$

   This means we have to readjust our guess.
   Now lets try the guess. $an^{1+c} + bn$ where a, b, and c are all constants.
   This guess is appropriate too because in terms of Big O notation running time,
   $O(n^{1+c}) = O(an^{1+c} + bn)$
   so for proof, we need to prove $T(n) \leq an^{1+c} + bn$ which uses $an^{1+c} + bn$ as our guess.

1. $T(n) = 3(\frac{an^{1+c}+bn}{3}) + n$
2. $T(n) = an^{1+c} + bn + n$
3. $T(n) = an^{1+c} + bn$
4. $an^{1+c} + bn \leq an^{1+c} + bn$

□

(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: The resulting running time is slightly less than the running time of quicksort, but if we ignore the constants when we put them in Big O notation, it is the same. □

[2]I.e., the $n/3$ median findings on 3 elements and the run of the Partition procedure around p.

(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: Once again, the running time of set size 5 is less than the running time of set size 3, but in terms of Big O notation, it is the same.

□

CSCI-GA.1170-001/002 Fundamental Algorithms October 1, 2014

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

Name: Keeyon Ebrahimi *Due: Wednesday, October 1*

Assume we are given an array $A[1 \dots n]$ 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 $\{0 \dots k\}$, since $n = 2k + 1$ is odd. Given $0 \leq j \leq k$, what is the probability that $smallest(A) = j$?

Solution:

The probability is $\frac{1}{k+1}$

The range of $smallest(A)$ goes from possible values 0 to k

We can see this by examining the edge cases. When the pivot is at the min or max of the array, then $smallest(A) = 0$. When the pivot is the actual median of the array, then $smallest(A) = k$. So we can have all possible values of $0, 1, 2, ..., k$, which is $k + 1$ options

This means that our probability will be $\frac{1}{k+1}$

□

(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: The expected value of $smallest(A)$ is

$$\frac{\sum_{j=0}^{k} j}{k+1}$$

Here is an example. Lets say $n = 7$. This would make $k = 3$. We now have an expected value of $\frac{0+1+2+3}{4}$, which equals $\frac{3}{2}$ □

(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: $T(n) = 2T(\frac{n}{2}) + n$
If we plug in the Master Theorem to this recurrence equation, we get

$$O(n) = n\ log(n)$$

This is the same running time 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: This argument is incorrect because in actuality, at some iterations, especially when k is small, we will pick a pivot that will be the min/max of the array, which means that we will be having a recursive call that will do absolutely no work for us. Because of this, the equation is not justified. □

CSCI-GA.1170-001/002 Fundamental Algorithms October 1, 2014

Solutions to Problem 3 of Homework 4 (13 points)

Name: Keeyon Ebrahimi *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] ≤ 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:

On a high level, this algorithm first checks if root i is positive. If it is positive, need to check all of its children and count their positive values. To prove the correctness of this algorithm, I will explain what each line is doing.

Line 2:

This just this checks if $i > n$. This means that we are searching for an element that is past the last leaf of the heap, and because that doesn't exist, we just return 0.

Line 3:

We are now checking if the current node is positive or not. Because this is a heap, we know that all children of a node will be less than the value of the parent, so if $A[i] \leq 0$, we know that both $A[i]$ and all of its children will not be positive, so we just return 0.

Line 4:

We only get here if $A[i]$ is positive. This is why we have the 1 being added. That 1 represents the positive value in $A[i]$. We add this 1 to positive elements to under the left child of this subroot, which we will be solving recursively.

Line 5:

When we hit this line, pos now is all the positive numbers of the root and the right children. All we need to calculate now is the positive numbers under the right child. We add the the number of positive values of the root and the left children with the recursively solved positive

numbers under the right child to get the total positive numbers.

This runs in time $O(pos)$ because this algorithm never evaluates anything that is not positive. It immediately return out of all non positive values, and runs through all positive, so we know that this has a running time of $O(pos)$

□

(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] ≤ 0 Return 0
4     If k = 1 Return k
5     pos = 1 + kPositiveCount(A, Left(i), n, k − 1)
6     If pos ≥ k Return k
7     pos = pos + kPositiveCount(A, Right(i), n, k − pos)
8     If pos ≥ k Return k
9     Return pos
```

The logic of this algorithm is that we keep on adding values to the positive count, and whenever we recursively call *kPositiveCount*, we subtract the k input by the current calculated positive value. We know this will run in time $O(k)$ because we never evaluate more than k. On line 4, we check if $k = 1$ and return if so. We never do work more than k because of this return, so we can safely say that the running time is $O(k)$

□

CSCI-GA.1170-001/002 Fundamental Algorithms October 1, 2014

Solutions to Problem 4 of Homework 4 (12 points)

Name: Keeyon Ebrahimi *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:

(i) To put a bunch of numbers into an unsorted array, this will only take $O(n)$ time because we just have to iterate through all items and place them into an array, disregarding order.

(ii) To Initialize a sorted array. We first create an unsorted array which will take time $O(n)$ and then we use *Quicksort* to make this a sorted array, which then takes $O(n\ log(n)$ time to run, so total time we have $O(n\ log(n) + n) = O(n\ log(n))$

(ii) To initialize a heap, we take $O(n)$ time. In high level terms, this is because when we call heapify, potentially all elements have to move down all the way to the bottom, which then creates $O(n)$ time. Here is that process explained.

When doing this, we must build the heap from bottom up. All the leaves are trivial because they don't need any shifting down, so we are good with those elements. All of the elements on the 1st level from the bottom going up can possibly shift down 1 level. All of the elements on the 2nd level from the bottom going up can possibly shift down 2 levels, and so on and so forth. If we have a total of h levels, then the amount of nodes on the level one up level is 2^{h-1}. At level 2 from the bottom up, we have At level one, we have 2^{h-2} elements.

So in terms of levels from the bottom up, if j represents that level, each node can potentially move down j levels. We also have 2^{h-j} nodes at that level, so the total work is a summation of all $j * 2^{h-j}$ for all levels. So the worst time this will take is $O(n)$

□

(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:

(*i*) The Traverse here should take $O(n)$ time. This is because you have to iterate through the entire array when looking for these numbers, so running time is $O(n)$

(*ii*) The Traverse here should be constant time, $O(1)$ time. This is because the array is already sorted, so all we have to do is grab the first three elements of the array.

(*iii*) The Traverse here will take $O(n)$ time. We know the three minimum numbers will all be leaves, but we are not sure which ones. This means that we have to iterate through $\frac{n}{2}$ elements in order find the three minimum numbers, which is still a running time of $O(n)$

□

(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:

(*i*) The Insert here is $O(1)$ because all we have to do is insert the new element at the end. Order doesn't matter so inserting in the end always works.

(*ii*) The insert here takes time $O(n)$. This is because you have to iterate through the array to find the correct spot. Wherever this element does go, we then must shift all remaining elements over to the right. This means that everything will run in running time of $O(n)$

(*iii*) The insert here will take running time of $O(log\ n)$. This is because when we insert, what we do is just insert the new item as a leaf. We then compare the element to its parent. If the element is smaller than the parent, no more work is needed to happen. If the new element is larger than its parent, we then have to replace the new element with its parent. We keep comparing this new element's value with its parents value, and stop once we find a parent that is larger than this new element. This does not run through most elements, and only has an effect on a logrithmic value.

□