CSCI-GA.1170-003/004 Fundamental Algorithms October 10, 2019

Solutions to Problem 1 of Homework 4 (18 points)

Name: jingshuai jiang (jj2903) *Due: 5 pm on Thursday, October 3*

Collaborators: NetID1, NetID2

Given a maxheap 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. We can write a recursive procedure to compute this value.

(a) (5 points) Consider the following pseudocode. Fill in the blanks with the appropriate values. Remember that this is a recursive procedure.

```
1 PositiveCount(A, i, n)
2     if i > n return 0
3     if A[i] ≤ 0 return _____
4     left =PositiveCount(_____)
5     right =PositiveCount(_____)
6     pos = _____
7     return pos
```

Solution:

```
1 PositiveCount(A, i, n)
2     if i > n return 0
3     if A[i] ≤ 0 return 0
4     left =PositiveCount(A, left(i), n)
5     right =PositiveCount(A, right(i), n)
6     pos = left + right + 1
7     return pos
```

□

(b) (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 $O(pos)$, independent of n.

Solution:

line 2 means if the index of the node i is larger than the whole maxheap, then we return this node's positive number to be 0;

line 3 means if the current node value is less than or equal to 0,then all of the childrens can not be positive numbers than ,we return 0;

line 4 means we recurrsively get the positive numbers of the leftchild.

line 5 means we recurrsively get the positive numbers of the rightchild.

each time when we get a positive node we will do the procedure of "pos = left +right+1" this one here means we add one to the positive numbers. Then the total number will be the value of the whole pos. Since each node is added for one time.Then it is in time $O(pos)$. This algorithm is independent of n because for different i it is different running time, with no relation to the n,just related to pos.

□

(c) (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.

Solution:

```
1 kPositiveCount(A, i, n, k)
2     if k ≤ 0 return 0
3     if i > n return 0
4     if A[i] ≤ 0 return 0
5     left =kPositiveCount(A, left(i), n, k − 1)
6     right =kPositiveCount(A, right(i), n, k − left − 1)
7     pos = left + right + 1
8     return pos
```

proof : line 2 means if k is less than or equal to 0, then it means that the number of k has been reached,we won't count the numbers of the positive numbers of its child nodes. so we just return 0.

line 3 means if the index of the node i is larger than the whole maxheap, then we return this node's positive number to be 0;

line 4 means if the current node value is less than or equal to 0,then all of the childrens can not be positive numbers than ,we return 0;

line 5 means We recurrsively find positve numbers in its left child.This ith node is positive, then we should minus it from the k.Since we only have to count $min(pos, k)$, and we only need to find k-1 nodes in its left child subarrays.

line 6 means We recurrsively find positve numbers in its right child. We only need to find k-1-left positive numbers in its right child subarrays.

the algorithm will stop when k is reached or all the positive numbers have been counted. k is minus from k to 0 one at a time.Then it takes k times to do it.It is obvious $O(k)$.If all the positive numebrs have been counted, which means k is greater than *pos* ,then it is

$$O(k)$$

□

CSCI-GA.1170-003/004 Fundamental Algorithms October 10, 2019

Solutions to Problem 2 of Homework 4 (6 points)

Name: jingshuai jiang (jj2903) *Due: 5 pm on Thursday, October 3*

Collaborators: NetID1, NetID2

Recall that in the worst case the running time of (non-randomized) QuickSort and Insertion-Sort are $\Theta(n^2)$. Refer to the algorithm given on Page 171 for QuickSort implementation. Pay attention to the last line of the Partition procedure.

(a) (3 points) Give an example of array of length n where both take time $\Omega(n^2)$. Justify your answer.

Solution:

if the array of length n is decreasing sorted, then both of them take time $\Omega(n^2)$

proof for QuickSort : if it is decreasing sorted,each time after the partition precedure it will devide the following array into a $T(n-1)$ and a $T(0)$ then we get

$$T(n) = T(n-1) + O(n) = \Theta(n^2) = \Omega(n^2)$$

proof for InsertionSort : if it is decreasing sorted , then it will take j-1 time exchange to resort. j is from 2 to a.length, then it takes time $O(n^2) = \Omega(n^2)$ □

(b) (3 points) Give an example of array of length n where (non-randomized) QuickSort runs in $\Omega(n^2)$ but Insertion-Sort runs in time $O(n)$. Justify your answer.

Solution:

if the array of length n is already sorted, then quicksort takes time $\Omega(n^2)$, insertion sort takes time $O(n)$

proof for QuickSort : if it is already sorted,each time after the partition precedure it will devide the following array into a $T(n-1)$ and a $T(0)$ then we get

$$T(n) = T(n-1) + O(n) = \Theta(n^2) = \Omega(n^2)$$

proof for InsertionSort : if it is already sorted , then it only needs to traversing the array without doing anything which takes time $O(n)$

□

CSCI-GA.1170-003/004 Fundamental Algorithms October 10, 2019

Solutions to Problem 3 of Homework 4 (20+3 points)

Name: jingshuai jiang (jj2903) *Due: 5 pm on Thursday, October 3*

Collaborators: NetID1, NetID2

We now define the notion of *stability* of a sorting algorithm. A sorting algorithm is said to be *stable* if it preserves the original order of equal elements in the sorted array. For example when sorting the array $< 4', 2, 8, 5, 3, 4^* >$ a stable sorting algorithm returns $< 2, 3, 4', 4^*, 5, 8 >$ while an unstable algorithm returns $< 2, 3, 4^*, 4', 5, 8 >$.

(a) (3 Points) Argue that QUICKSORT (as on page 171) is not necessarily a stable sorting algorithm. In particular, give an example of an array where QUICKSORT is not stable. Make sure you justify your answer.

Solution: $< 2', 2^*, 3, 4, 5, 0 >$ after the algorithm it will output $< 0, 2^*, 2', 3, 4, 5 >$ □

(b) (3 Points) Remember, the PARTITION procedure places all elements less than or equal to the pivot $A[r]$ (whose rank is later determined as q) into array $B = A[p, \ldots, q-1]$, and all elements greater than the pivot $A[r]$ into array $C = A[q+1, \ldots, r]$. Which of the following three (mutually exclusive) statement is correct, after the first call to PARTITION:

1. Elements of B are always guaranteed to be in "stable" order, but elements of C may or may not be.
2. Elements of C are always guaranteed to be in "stable" order, but elements of B may or may not be.
3. Neither elements of B nor the elements of C are guaranteed to be in "stable" order.

If you answered 1. or 2., be sure to argue why the corresponding array is guaranteed to be "stable" (you can refer to your counter-example in part (a) to show why the other array is not stable). If you answered 3., make sure you give the example of this scenario as well.

Solution:

my answer is 1.

When doing the partition procedure, when j gose from p to r-1 if we meet a number that is smaller than A[r], we will insert it to the ith place. and i will move to the next place. This first come first insert rule guarantee the relative order of these elements. And finally makes it stable.

In the meantime everytime when it finishes the partition process. It will exchange the A[pivot] with A[r], which will destroy the relative order of A[pivot] to some other points. □

(c) (5 Points) Give a variant of the PARTITION procedure which still runs in time $O(n)$, but makes QUICKSORT stable. Notice, you are *no longer required* to sort the elements "in place".

Solution:

```
1 NewPartition(A, p, r)
2     smaller,greater = [],[]
3     for j = ptor − 1
4           if A[j] ≤ A[r]
5                 smaller.append(A[j])
6           else  greater.append(A[j])
7     for i = 0tosmaller.length − 1
8           A[p + i] = smaller[i]
9     A[p + smaller.length] = A[r]
10    for i = 0togreater.length − 1
11          A[p + i + smaller.length + 1] = greater[i]
12    return p + smaller.length
```

□

In the previous part we modified the PARTITION procedure described in the textbook to make it *stable*. Observe that a *stable* partition procedure is one which preserves the relative order of elements less than the pivot and the relative order of elements greater than pivot after the partition.

(d) (2 points) Prove that a stable partition on A where A is a random permutation, results in subarrays that are random permutations themselves. (**Hint**: Consider the two distributions: (a) running the stable partition on a random permutation A and taking the left half; (b) a random permutation over B where B consists of all elements in A less than the pivot.)

Solution:

Let we assume that the length of the left half of A after first partition is q (a) the left half of A after first partition is uniform distribution. Because it just move all the numbers smaller than A[r] to the left side without disturbing their raltive order. Then each possible permutaion will have a possibility of $\frac{1}{q!}$ which is uniform distribution.

(b) the random permutation over B where B consists of all elements in A less than the pivot is also a uniform distribution. Since b have q elements, then every possible permutation has the possibility of $\frac{1}{q!}$

Obviously these two are the same distribution, a stable partition on A results in subarrays that are random permutaions themselves. □

(e) (6 points) Use the above observation to write a recurrence for the expected running time of Quicksort on a random permutation A, and solve it. (**Hint**: Recall the recurrence for Randomized QUICKSORT from the lecture.)

Solution:

according to the question (4-d) the two parts of the A after the partition procedure are still random permutatuions themselves. Then we just divide the A array into two similar random array. Then we get the equation.

$$T(n) = \exp_{0\le q\le n-1}[T(q) + T(n-q-1) + n] = n + \frac{1}{n}[T(0) + ... + T(n-1)]$$
$$nT(n) = n^2 + 2[T(0) + ... + T(n-1)]$$
$$(n-1)T(n-1) = (n-1)^2 + 2[T(0) + ... + T(n-2)]$$
$$nT(n) = (n+1)T(n-1) + 2n - 1]$$
$$\frac{T(n)}{n+1} = \frac{T(n-1)}{n} + \frac{2n-1}{n(n+1)} \le \frac{T(n-1)}{n} + \frac{2}{n}$$
$$S(n) = S(n-1) + \frac{2}{n} = \Theta(logn)$$
$$T(n) = \Theta(nlogn)$$

□

(f) (1 point) Which of QuickSort and InsertionSort has better expected running time on a random permutation? Use the results from Problem 1-3 for expected running time of Insertion Sort.

Solution:

quicksort has beeter expected running time. □

(g) (3 points) [**Extra credit**] Prove that the expected running time of QuickSort computed in part (d) does not change if we use the standard (non-stable) Partition procedure from the book.

Solution: The unstable partition procedure will not change the relative order of the smaller parts. It do not change the relative order of the bigger parts except for the first number in the bigger part, which will be swiched to the end of this bigger part. and after this switch the bigger part and the smaller parts are still random permutations. Then we can get a similar procedure when dealing with a stable partition procedure. Then we get

$$T(n) = \exp_{0\le q\le n-1}[T(q) + T(n-q-1) + n] = n + \frac{1}{n}[T(0) + ... + T(n-1)]$$
$$nT(n) = n^2 + 2[T(0) + ... + T(n-1)]$$
$$(n-1)T(n-1) = (n-1)^2 + 2[T(0) + ... + T(n-2)]$$
$$nT(n) = (n+1)T(n-1) + 2n - 1]$$
$$\frac{T(n)}{n+1} = \frac{T(n-1)}{n} + \frac{2n-1}{n(n+1)} \le \frac{T(n-1)}{n} + \frac{2}{n}$$
$$S(n) = S(n-1) + \frac{2}{n} = \Theta(logn)$$
$$T(n) = \Theta(nlogn)$$

□