CMSC351 - Fall 2014, Homework #3

Due on October 20th at the start of class

Name:
Section:

- Grades depend on neatness and clarity.
- Write your answers with enough detail about your approach and concepts used, so that the grader will be able to understand it easily. You should ALWAYS prove the correctness of your algorithms either directly or by referring to a proof in the book.
- Write your answers in the spaces provided. If needed, attach other pages.
- The grades would be out of 100. Four problems would be selected and everyones grade would be based only on those problems. You will also get 25 bonus points for trying to solve all problems.

Solution 1 The hash function output for the sequence is :

$1, 5, 3, 1, 4, 0, 4, 4, 0, 3, 1, 0$

92 98 4 85 86 17 23 79 10 11 99 80

(a)

0	:	$17, 10, 80$	(1)
1	:	$92, 85, 99$	
2	:		
3	:	$4, 11$	
4	:	$86, 23, 79$	
5	:	98	
6	:		

(b)

0	:	17	(2)
1	:	92	
2	:	85	
3	:	4	
4	:	86	
5	:	98	
6	:	23	

Solution 2 Here would be the heap in each step:
Step 1: (4)
Step 2: $(4, 1)$
Step 3: $(4, 1, 2)$
Step 4: $(4, 3, 2, 1)$
Step 5: $(9, 4, 2, 1, 3)$
Step 6: $(9, 4, 8, 1, 3, 2)$
Step 7: $(9, 4, 8, 1, 3, 2, 7)$
Step 8: $(9, 6, 8, 4, 3, 2, 7, 1)$
Step 9: $(9, 6, 8, 5, 3, 2, 7, 1, 4)$
Step 10: $(10, 9, 8, 5, 6, 2, 7, 1, 4, 3)$

You can see heap construction element by element here:

https://www.cs.usfca.edu/ galles/visualization/Heap.html

Solution 3 (a) Use a hash table of size n. Let h be the hash function. First insert all elements into the hash table. Then go over the elements of the array one by one. For each element x_i, check if the cell $h(k - x_i)$ is nonempty and if any of the elements hashed to this cell is equal to $k - x_i$. In case you find such element you are done. Otherwise, move to the next element. Since the table size is n, the expected number of elements mapped to any cell is 1. Therefore, searching for $k - x_i$ is $O(1)$. This means that the algorithm runs in $O(n)$.

(b) Sort the array. Take two pointers, one starting at the first element of the array moving forward, and the other starting at the last element of the array moving backward. Do the following while the pointers have not crossed each other.

- Move the pointer at the end backward if sum of the current 2 numbers is larger than k.
- Move the pointer at the start forward if sum of the current 2 numbers is smaller than k.
- Accept if sum is equal to k.

It can be seen that you never have to move the first pointer backward or the second pointer forward. The overall running time would be $O(n \log n)$ for sorting the array plus $O(n)$ for moving two pointer. Thus, the overall running time is $O(n \log n)$.

Solution 4 Solution with heap : Use a heap of size k. Initially, insert elements $A[1]$ to $A[k]$ into it. $B[1]$ would be the top member of the heap. To find $B[2]$ we need to delete $A[1]$ from the heap and insert $A[k+1]$ instead. Inserting $A[k+1]$ is easy in $O(\log k)$ but how can we delete $A[1]$ from the heap in $O(\log k)$ since $A[1]$ can be anywhere in the heap now. So for each element of the array we need to keep track of where that element moves in the heap. Use an array Location which keeps track of the elements' location in the heap. Each time you swap two elements in the heap, you need to update the location of the corresponding elements in the array. Now as we move forward to calculate $B[i]$ we need to delete $A[i-1]$ from the heap, and insert $A[i+k]$ to the heap. Both take $O(\log k)$. So the order of the algorithm would be $O(n \log k)$.

Removing an element from the middle of the heap is not trivial. When you remove something from somewhere in the heap you have to replace it with the last element of the heap (rightmost element in the heap). Then you need to first sift up and then sift down this element in the heap (think how you can maintain a heap by just one sift up and sift down.).

Solution with Binary Search Tree:

Keep a balanced BST of the first k elements. Finding minimum element of the BST is possible in $O(\log k)$ (Just go to the left child while possible). The minimum element of the BST is $B[1]$ from now on in order to find $B[i+1]$ remove $A[i]$ from BST and insert $A[i+k]$ to the BST. Finding $A[i]$ is possible in BST in $O(\log k)$ and so are deleting and adding elements. So here you do not have to have an array to keep track of location of the elements.

Solution 5 First forget about insert and delete. Suppose we have a balanced binary search tree such that for each node we have the number of all of its descendants (including itself). Finding the k-th smallest element in this tree is possible in $O(\log n)$ with a recursive algorithm. If you are at a node and you want to find the k-th smallest element simply check how many descendants the left child of the node has. There are three possible cases.

(a) If the left child has more than or equal to k descendants, you have to look for k-th smallest element in the left subtree recursively(You can completely ignore the right subtree in this case).

(b) If the left child has exactly $k-1$ descendants, the node you are currently observing is the k-th smallest element

(c) If the left child has $l < k-1$ descendants, you have to look for the $k-l-1$-th smallest element in the right subtree recursively (You can completely ignore the right subtree in this case.

Note that these three cases cover all possible cases. For example, if the node does not have left child then it can be assumed that its left child has 0 decendents.

Now, since we are working with a binary search tree, insert and delete can be done in $O(\log n)$. The only problem is how to update the number of descendants at each node while inserting and deleting in $O(\log n)$. To do this, add one to number of descendants of every node on the way while inserting a new element. If you need to delete an item, you should subtract one from all of the deleted nodes ascendants. Since the tree is balanced, the height of the tree is $O(\log n)$ and hence, all of these take $O(\log n)$ running time.

Solution 6 (a) Let PARTITION3 be the routine that partitions the array based on its first and second element and returns the new positions of the two pivots. First we make sure that the second element is bigger than the first element. Let PARTITION be the routine that partitions the array based on only one pivot (its first element) and returns the position of the pivot. This is similar to the routine used in normal Quicksort. In PARTITION3 we first partition the array based on the second element using Partition. So everything bigger than $A[1]$ and $A[2]$ would be placed after $A[2]$'s new position. Let q be $A[2]$'s new position in the array. Then we have to partition based on $A[1]$. However, since everything after q is bigger than both $A[1]$ and $A[2]$, we only need to partition the second time from 1 to $q-1$.

---

function PARTITION3($A[1,\ldots,n]$)
  if $A[1] > A[2]$ **then**
    $swap(A[1], A[2])$
  end if
  q = PARTITION($A[2,\ldots,n]$)
  p = PARTITION($A[1,\ldots,q-1]$)
  return (p,q)
end function

(b)

---

function QUICKSORT($A[1,\ldots,n]$)
  if $(n = 1)$ **then**
    return
  end if
  (p,q) = PARTITION3($A[1,\ldots,n]$)
  QUICKSORT($A[1,\ldots,p-1]$)
  QUICKSORT($A[p+1,\ldots,q-1]$)
  QUICKSORT($A[q+1,\ldots,n]$)
end function

(c) Suppose at the end of the routine PARTITION3, the pivots are at positions p and q respectively. First one comparison is done between $A[1]$ and $A[2]$. Then the first partition does $n-2$ comparisons, and the second partition does $q-2$ comparisons. Therefore, if the pivots are the p-th and q-th elements of the original array $n+q-3$ comparisons are done. The probability that the pivots are p-th and q-th elements of the original array is $\frac{1}{\binom{n}{2}}$ for any p, q. Let X denote the number of comparisons that PARTITION3 makes. To calculate the expectation we have to go over all possible p, q and sum up the value of X with probability of its happening. You can assume that it is like getting average over all possible situations (Since we have overall $\binom{n}{2}$ cases.

$$E(X) = 2\times\frac{1}{\binom{n}{2}}\sum_{p=1}^{n}\sum_{q=p+1}^{n} q+n-3 = 2\times\frac{1}{\binom{n}{2}}\frac{1}{6}n(5n^2-12n+7) = \frac{1}{3}(5n-7) \tag{3}$$

If the array is divided into equal sizes the first iteration uses $n-2$ comparisons and the second partition uses $\frac{2n}{3}-2$ number of comparisons. One comparison is done between the first two elements so overall $\frac{5n}{3}$ comparisons

(d) $T(n) = O(n) + 3T(\frac{n}{3})$.
By Master Theorem, $T(n) = O(n\log n)$.

Solution 7 For this problem we need to assume that all elements are distinct.

Let $k = \frac{n}{11}$ and $m_1, m_2, \ldots, m_k$ be the medians of the groups. Without loss of generality assume they are in the sorted order(We are not sorting them. This assumption is just for the sake of analysis!). $m_1, m_2, \ldots, m_{\frac{k}{2}}$ are less than M. Also they are medians of their own group. Therefore, in each of the groups number 1 to $\frac{k}{2}$, there are 5 elements less than the median. This means there are $5 \times \frac{k}{2} = 5 \times \frac{n}{11} \times \frac{1}{2} = 5 \times \frac{n}{22}$ elements less than or equal to M.

(b) Let $T(n)$ be the running time of the algorithm on an array of size n. Finding median in each group can be done by any sorting algorithm. For example, Bubblesort can sort a group of size 11 with $\frac{11\times10}{2} = 55$ comparisons. Once the array is sorted, finding median is easy. There are $\frac{n}{11}$ groups, so this will take $55 \times \frac{n}{11} = 5n$ comparisons.

After finding medians of the groups, we will run the selection algorithm on the medians. Again the number of medians is $\frac{n}{11}$ so this takes $T(\frac{n}{11})$ time.

Partitioning an array of size n based on any element takes n comparisons.

Finally, in (a) we proved that at least $\frac{5}{22}$ of the elements are less than M. Similarly, $\frac{5}{22}$ of the elements are bigger than M. So none of the two parts can have more than $\frac{17}{22}$ of the elements. So continuing Selection in either part does not take more than $T(\frac{17}{22}n)$

Therefore, the recurrence relation for this algorithm looks like this:

$T(n) \leq 5n + T(\frac{n}{11}) + n + T(\frac{17}{22}n)$.

(c) This can be proven by Substitution method. Just assume $T(m) \leq cm$ for any $m < n$ and for a constant $c > 0$. Then

$T(n) \leq 6n + c\frac{n}{11} + c\frac{17n}{22} = (6 + \frac{19}{22}c)n \leq cn$

Where the last inequality holds if we set the constant c greater than 35.

Note : It does not matter if you do not have the exact same constants. For example, you can estimate finding median in each group to take any constant number of comparisons and still prove that the running time is $O(n)$.