UNIVERSITY OF TORONTO, DEPARTMENT OF ECE

ECE1762 - Homework 2

Xinyun Lv(1001091178), Yang Wang(1001319227)

October 15, 2015

1 PROBLEM I

(a) **Solution:**

To get the expected number of recursive calls, we let x_k be the indicator random variable associated with the event in which the kth element is chosen as x^*. Thus,

$$x_k = \begin{cases} 1 & \text{if the } k\text{th smallest element is chosen as } x^* \\ 0 & \text{otherwise} \end{cases}$$

Let $T(n)$ denotes the expected number of recursive calls of a set of numbers with size of n. Thus,

$$T(n) = \begin{cases} T(0) & \text{if the largest element is chosen as } x^* \\ T(1) + T(0) & \text{if the second largest element is chosen as } x^* \\ T(2) + T(0) & \text{if the third largest element is chosen as } x^* \\ . \\ . \\ . \\ T(n-1) + T(0) & \text{if the smallest element is chosen as } x^* \end{cases}$$

Then we have,

$$T(n) = x_0 T(0) + \sum_{k=1}^{n-1} x_k (T(k) + T(0))$$

$$\begin{aligned}
E(T(n)) &= \frac{1}{n} \cdot T(0) + E(\sum_{k=1}^{n-1} x_k(T(k) + T(0))) \\
&= \frac{1}{n} \cdot T(0) + \sum_{k=1}^{n-1} \{[E(x_k T(k))] + E[x_k T(0)]\} \\
&= \frac{1}{n} \cdot T(0) + \frac{1}{n} \sum_{k=1}^{n-1} E[T(k)] + \frac{n-1}{n} \cdot T(0) \\
&= T(0) + \frac{1}{n} \sum_{k=1}^{n-1} E[T(k)]
\end{aligned}$$

Where $T(0) = 1$. So,

$$E(T(n)) = T(0) + \frac{1}{n} \sum_{k=1}^{n-1} E[T(k)]$$

(b) **Solution:**
According to the result we got from (a), we have:

$$\sum_{k=1}^{n-1} E[T(k)] = n \cdot (E(T(n)) - T(0))$$

Also, we have,

$$\sum_{k=1}^{n-2} E[T(k)] = (n-1) \cdot (E(T(n-1)) - T(0))$$

Substract two equations above we have:

$$\begin{aligned}
E(T(n-1)) &= n \cdot (E(T(n)) - T(0)) - (n-1) \cdot (E(T(n-1) - T(0)) \\
nE(T(n)) - nE(T(n-1)) &= T(0) = 1 \\
E(T(n)) &= E(T(n-1)) + \frac{1}{n}
\end{aligned}$$

Thus, we have,

$$\begin{aligned}
E(T(n)) &= E(T(n-1)) + \frac{1}{n} \\
E(T(n-1)) &= E(T(n-2)) + \frac{1}{n-1} \\
... &= ... \\
E(T(2)) &= E(T(1)) + \frac{1}{2}
\end{aligned}$$

Which could be simplified as,

$$E(T(n)) = E(T(1)) + \sum_{i=2}^{n} \frac{1}{i}$$

Since $E(T(1)) = 1$,

$$E(T(n)) = \sum_{i=1}^{n} \frac{1}{i} \leq lnn + 1 = O(logn)$$

(c) **Solution:**
Similar with (a), let $T(n)$ denotes the expected running time of a set of numbers with size of n. Thus,

$$T(n) = \begin{cases} \theta(n) & \text{if the largest element is chosen as } x^* \\ T(1) + \theta(n) & \text{if the second largest element is chosen as } x^* \\ T(2) + \theta(n) & \text{if the third largest element is chosen as } x^* \\ . \\ . \\ . \\ T(n-1) + \theta(n) & \text{if the smallest element is chosen as } x^* \end{cases}$$

Then we have,

$$T(n) = x_0\theta(n) + \sum_{k=1}^{n-1} x_k(T(k) + \theta(n))$$

$$\begin{aligned} E(T(n)) &= \frac{1}{n} \cdot \theta(n) + E(\sum_{k=1}^{n-1} x_k(T(k) + \theta(n))) \\ &= \frac{1}{n} \cdot \theta(n) + \sum_{k=1}^{n-1} \{[E(x_k T(k))] + E[x_k \theta(n)]\} \\ &= \frac{1}{n} \cdot \theta(n) + \frac{1}{n} \sum_{k=1}^{n-1} E[T(k)] + \frac{n-1}{n} \cdot \theta(n) \\ &= \theta(n) + \frac{1}{n} \sum_{k=1}^{n-1} E[T(k)] \end{aligned}$$

So,

$$E(T(n)) = \theta(n) + \frac{1}{n} \sum_{k=1}^{n-1} E[T(k)]$$

(d) **Solution:**
Here, we can make a guess, $E(T(n)) \le cn = O(n)$, where c is a constant number. By induction we have,

$$\begin{aligned} E(T(n)) &\le \theta(n) + \frac{1}{n} \sum_{k=1}^{n-1} ck \\ &\le \theta(n) + \frac{c}{n} \sum_{k=1}^{n-1} k \\ &\le \theta(n) + \frac{cn}{2} = O(n) \end{aligned}$$

(e) **Solution:**
To compute the probability that Find-Max(X) compares the i-th and the j-th largest

items in X, we can think about when two items are not compared. For ease of analysis, we rename the elements of X as $z_n, z_{n-1}, ..., z_1$, with z_i being the ith largest element. We can also define the set $Z_{ij} = \{min(z_i, z_j), ..., z_2, z_1\}$ to be the set of elements between z_1 and minimum element of z_i and z_j , inclusive. because we assume that element values are distinct, once a x^* is chosen with $z_j < x^* < z_i$ or $z_i < x^* < z_1$(we can suppose that $z_i > z_j$ for ease of analysis), we know that z_i and z_j cannot be compared at any subsequent time. If, on the other hand, z_i is chosen as x^* before any other item in Z_{ij}, then z_i will be compared to each item in Z_{ij} including z_j, except for itself. Similarly, if z_j is chosen as x^* before any other item in Z_{ij}, then z_j will be compared to each item in Z_{ij} including z_i, except for itself. Thus z_i and z_j are compared if and only if the first element to be chosen as x^* from Z_{ij} is either z_i or z_j. Meanwhile, prior to the point at which an element from Z_{ij} has been chosen as x^*, the whole Z_{ij} is together in the same partition, so any element of Z_{ij} is equally likely to be the first one chosen as x^*. Because the set Z_{ij} has j elements, and because x^* is chosen randomly and independently, the probability that any given element is the first one chosen as x^* is $1/j$. Thus, in general case, we have

$$Pr(z_i \text{ compared with } z_j) = Pr(z_i \text{ or } z_j \text{ is chosen as the first element from } Z_{ij}) = \frac{2}{max\{i,j\}}$$

2 Problem II

Solution:

To show that number of comparisons is at least $n-k+log\begin{pmatrix} n \\ k-1 \end{pmatrix}$ is equivalent to showing that number of outcome boxes is at least $2^{n-k} \times \begin{pmatrix} n \\ k-1 \end{pmatrix}$.

Let $V(n,k)$ denote the number of comparisons of finding the kth largest element of n-element set.

When $k=1$, $V(n,1)$ represent the number of comparisons for finding the largest element of n-element set. We could easily observe that $V(n,1) \geq n-1$, since every element except the largest must lose at lease one comparison. This observation implies that in any comparison tree to find the largest element, every leaf has depth at least $n-1$, which implies that there must be at least 2^{n-1} leaves. We can generalize this argument to prove a lower bound for $V(n,k)$ for arbitrary values of k.

Let T be a comparison tree that identifies the kth largest element $x_{(k)} \in X$.

Suppose we are at some outcome box for determining the kth largest element $x_{(k)}$. Assume there are some elements not yet directly known to be bigger or smaller than $x_{(k)}$.

- Set U is set of elements not yet directly known to be bigger or smaller than $x_{(k)}$.
- Set L is set of those known to be larger than $x_{(k)}$
- Set S is set of those known to be smaller than $x_{(k)}$

Suppose that there is an element $x_* \in U$ such that $x_* > x_{(k)}$, which implies that $| L |= k-1+1 = k$ and $x_{(k)}$ become the $k+1$ largest element. However, we have known that $x_{(k)}$ is the kth largest element which is a contradiction. So we can conclude that when a decision reaching the right outcome of finding the kth largest element also decide correctly which the $k-1$ larger elements are.

Now suppose that the set of $k-1$ largest elements of X:

$$L = \{x_{(1)}, x_{(2)}, ..., x_{(k-1)}\}$$

Since those elements in set L must bigger than those elements in set S, we could remove those comparisons from T. Call the reduced tree R. Since the reduced tree R also identifies the largest element of S, based on the conclusion we got from base case $k = 1$, R must have at least 2^{n-k} leaves. Since there are $\begin{pmatrix} n \\ k-1 \end{pmatrix}$ choices for set L, we conclude that T has at least

$$\begin{pmatrix} n \\ k-1 \end{pmatrix} \cdot 2^{n-k}$$

leaves which also implies that the number of comparisons is at least

$$n - k + log\begin{pmatrix} n \\ k-1 \end{pmatrix}$$

Thus we proved the argument.

3 Problem III

Solution:

Since finding the kth largest element in the union of the two arrays is equivalent of finding the $n-k+1$th smallest element. To easier analyze the algorithm we are going to apply, we denote $k^* = n-k+1$.

Binary search is a good example of achieving logarithmic complexity by halving its search space in each iteration. As a good hint, to achieve complexity of $O(logn)$ running time, we must halved the search space of A and B in each iteration.

Base case is pretty strait forward: If length of one of the array is 0, then the answer is $k*$th smallest element of the second array.

Array A and array B could be described as follows:

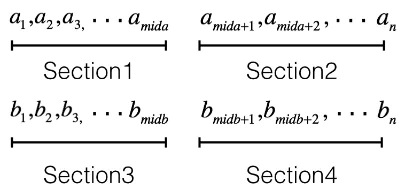
$a_1, a_2, a_3, \cdots a_{mida}$
$a_{mida+1}, a_{mida+2}, \cdots a_n$
Section1
Section2
$b_1, b_2, b_3, \cdots b_{midb}$
$b_{midb+1}, b_{midb+2}, \cdots b_n$
Section3
Section4

Figure 3.1: Picture of putting balls into bins

where $mida$ is the mid index of the first array A and $midb$ is the mid index of the second array B.

If $mida + midb < k^*$, then it means that the k^* smallest element is at {Section1, Section2, Section4} or {Section2, Section3, Section4}, the reason is as follows. If $A[mida] > B[midb]$, we can safely discard section3 since every element of section3 is smaller than $A[mida]$ and cannot be the k^*th smallest element of the union array. Similarly, If $A[mida] < B[midb]$ then we can discard section1.

However, if $mida + midb > k^*$, then it means that the k^* smallest element is at {Section1, Section2, Section3} or {Section1, Section3, Section4}, the reason is as follows. If $A[mida] < B[midb]$, we can safely discard section4 since every element of section4 is bigger than $A[mida]$ and cannot be the k^*th smallest element of the union array. Similarly, If $A[mida] > B[midb]$ then we can discard section2.

The pseudo code is as follows:

Algorithm 1 Find(A, B, k^*), $k^* = n - k + 1$

```
if length(A) = 0 then return $B[k^*]$
if length(B) = 0 then return $A[k^*]$
$mida \leftarrow length(A)/2$
$midb \leftarrow length(B)/2$
if $mida + midb < k^*$ then
    if $A[mida] > B[midb]$ then $Find(A, B[midb+1:], k^* - midb - 1)$
// B[midb + 1:] means subarray after index of midb
    else$Find(A[mida+1:], B, k^* - mida - 1)$
// A[mida + 1:] means subarray after index of mida
else
    if $A[mida] > B[midb]$ then $Find(A[:mida], B, k^*)$
// A[:mida] means subarray before index of mida
    else$Find(A, B[:midb], k^*)$
// B[:midb] means subarray before index of midb
```

Since we halved the search space of A and B in each iteration, in the worst case we may halved both A and B down to empty which will cost $log(n_1) + log(n_2) = 2logn$. so the time complexity is $O(logn)$

4 Problem IV

(a) **Solution:**
To show the rank of the root is $O(logn)$, we could first alternatively show that total nodes of a leftist heap with length of the rank of root is k is at least $2^{k+1} - 1$.
Proof:
Proof: By induction on the height h of L. Basis ($h = 0$): Then L consists of a single node and its right path has length $k = 0$. Indeed, L has $1 \geq 2^1 - 1$ nodes, as wanted.
Inductive Step ($h > 0$): Suppose the theorem holds for all leftist heaps that have height $< h$ and let L be a leftist heap of height h. Further, let k be the length of L's right path and n be the number of nodes in L. Consider two cases:
Case 1: $k = 0$ (i.e. L has no right subtree). But then clearly $n \geq 1 = 2^1 - 1$ as wanted.
Case 2: $k > 0$. Let L_L, L_R be the left and right subtrees of L; n_L, n_R be the number of nodes in L_L and L_R; and k_L, k_R be the lengths of the right paths in L_L and L_R respectively. Because the left and right subtrees of a leftist heap are leftist heaps and the distance of a leftist heap's root is equal to the length of the tree's right path. Then we have $k_R = k - 1$, and by definition of leftist heap $k_L \geq k_R$. Since L_L, L_R have height $< h$ we get, by induction hypothesis, $n_R \geq 2^k - 1$ and $n_L \geq 2^k - 1$. But $n = n_L + n_R + 1$ and

thus, $n \geq 2^k - 1 + 2^k - 1 + 1 = 2^{k+1} - 1$. Therefore $n \geq 2^{k+1} - 1$, as wanted.

$$\begin{aligned} n &\geq 2^{rank(x)+1} - 1 \\ n+1 &\geq 2^{rank(x)+1} \\ log(n+1) &\geq rank(x) + 1 \\ rank(x) &\leq lg(n+1) - 1 = O(logn) \end{aligned}$$

(b) **Solution:**

If one of the two leftist heap is empty, we are done. Otherwise we want to merge two non-empty leftist heaps h_1 and h_2. We can assume that without loss of generality, that the key in the root of h_1 is less than or equal to the key in the root of h_2. If $key(h_1) > key(h_2)$ we swap h_1 and h_2, so h_1 has the smaller root. Recursively we merge h_2 with the right subtree of h_1, and we make the resulting leftist tree into the right subtree of h_1. If this has made the rank of the right subtree's root longer than the rank of the left subtree's root, we simply interchange the left and right children of h_1's root(thereby making what used to be the right subtree of h_1 into its left subtree and vice-versa). Finally, we update the rank of h_1's root . The pseudo code below gives more details.
 The complexity of the algorithm is proportional to the number of recursive calls to

Algorithm 2 MERGE(h_1, h_2)

```
if $h_1 = \phi$ then return $h_2$
else if $h_2 = \phi$ then return $h_1$
else
    if $key(h_1) > key(h_2)$ then
        $swap(h_1, h_2)$
    $right(h_1) \leftarrow MERGE(right(h_1), h_2)$
// merge $h_2$ on right
end MERGE
```

MERGE. It is easy to see that, in the worst case, this will be equal to rank of h_1's root plus rank of h_2's root. Let the number of nodes in these trees be n_1 and n_2. By the proof procedure in question(a), we have $rank(h_1) \leq log(n_1)$, and $rank(h_2) \leq log(n_2)$. Thus $rank(h_1)+rank(h_2) \leq logn_1 + logn_2$. Let $n = \max(n_1, n_2)$. Then $rank(h_1)+rank(h_2) \leq 2logn$. Therefore, MERGE is called at most $2logn$ times, and the complexity of the algorithm is $O(log(n)$ in the worst case.
Meanwhile, because we maintain the key(h_1) is smaller than key(h_2) by swapping h_1 and h_2 if hey(h_1) < key(h_2) and then we let the rightChildOf(h_1) equals to the merge result of rightChildOf(h_1) and h_2. So we will always let the node with smaller key merge earlier than the node with larger key. That is, the order invariant is maintained.

(c) **Solution:**

We can suppose that x is not on the rightmost path of the given leftist heap and the rank of x changed after MERGE operation. Since every MERGE operation happens on the right child of every node, so for those node not on the right most path will still follow

$rank(x) = 1 + rank(right(x))$ without any changes with their rank in the new merged tree which is a contradiction.
So, the rank of a node x might change if and only if x is on the rightmost path.

(d) **Solution:**
If one of the two leftist heap is empty, we are done. Otherwise we want to merge two non-empty leftist heaps h_1 and h_2. We can assume that without loss of generality, that the key in the root of h_1 is less than or equal to the key in the root of h_2. If $key(h_1) > key(h_2)$ we swap h_1 and h_2, so h_1 has the smaller root. Recursively we merge h_2 with the right subtree of h_1, and we make the resulting leftist tree into the right subtree of h_1. If this has made the rank of the right subtree's root longer than the rank of the left subtree's root, we simply interchange the left and right children of h_1's root(thereby making what used to be the right subtree of h_1 into its left subtree and vice-versa). Finally, we update the rank of h_1's root . The pseudo code below gives more details.

Algorithm 3 MERGE(h_1, h_2)

```
if $h_1 = \phi$ then return $h_2$
else if $h_2 = \phi$ then return $h_1$
else
    if $key(h_1) > key(h_2)$ then
        $swap(h_1, h_2)$
    $right(h_1) \leftarrow MERGE(right(h_1), h_2)$
// merge $h_2$ on right and swap if needed

    if $right(h_1) \neq \phi$ and $(left(h_1) = \phi$ or $rank(right(h_1)) > rank(left(h_2)))$ then
        $swap(right(h_1), left(h_1))$
// swap children to make leftist

    if $right(h_1) = \phi$ then
        $rank(h_1) = 0$
    else
        $rank(h_1) \leftarrow rank(right(h_1)) + 1$
// update rank value

    return $h_1$
end MERGE
```

The order invariant is maintained which is proved in question(b). As shown in the pseudo code above, we can note that if node x with $rank(left(x)) < rank(right(x))$, then its children $left(x)$ and $right(x)$ will be swapped in updating.

The running time analysis is pretty similar with question(b), observe that there is a constant number of steps that must be executed before and after each recursive call to MERGE. Thus the complexity of the algorithm is proportional to the number of recursive calls to MERGE. It is easy to see that, in the worst case, this will be equal to rank of h_1's root plus rank of h_2's root. Let the number of nodes in these trees be n_1 and n_2. By the proof procedure in question(a), we have $rank(h_1) \le log(n_1)$, and $rank(h_2) \le log(n_2)$. Thus $rank(h_1) + rank(h_2) \le log n_1 + log n_2$. Let $n = \max(n_1, n_2)$. Then $rank(h_1) + rank(h_2) \le 2 log n$. Therefore, MERGE is called at most $2 log n$ times, and the complexity of the algorithm is $O(log(n)$ in the worst case.

(e) **Solution:**
Using the MERGE algorithm in problem(b) we can write algorithm for Insert and DeleteMin:

Insert(e, r) where e is an element, r is the root of tree.

```
1. Let $r'$ be a the leftist tree containing only $e$.
2. MERGE($r'$, $r$).
```

As is described above, we could treat the element we want to insert to the given tree as a leftist heap contains only one element as root node. Then we could merge this leftist heap with the given leftist heap as we did before. The running time of merging and rank updates is $O(log n)$ as we discussed in previous steps.

DeleteMin(r)

```
1. $min \leftarrow$ element stored at $r$ (root of the given leftist heap)
2. $r \leftarrow MERGE(left(r), right(r))$
3. **return** $min$.
```

To delete the minimum element in the give leftist heap, we can start with removing the root of the given leftist heap. Then, we could merge and update the rank of its left and right subtrees, which has a running time of $O(log n)$ as we discussed above.