COMP 3711 – Design and Analysis of Algorithms
2023 Fall Semester – Written Assignment Solution # 2
Distributed: 9:00 on September 30, 2023 Due: 23:59 on October 15, 2023

Problem 1: [20 pts]

(a) Construct data structure C by sorting array A in non-decreasing order. The sorting can be done by an optimal sorting algorithm in $O(n \log n)$ time. Clearly, the array C uses $\Theta(n)$ space as there are n elements in the array.

We use a variant of binary search Findnext$(C, 1, n, x)$ that returns the position of the smallest integer y in the subarray $C[1..n]$ that is larger than x. If there are multiple occurrences of y in $C[1..n]$, the position of the leftmost occurrence of y is reported. If x is the largest in $C[1..n]$, then $n + 1$ is reported.

A range query for $(x, y]$ can be answered by calling the following procedure:

```
Range (C, x, y)
  X ← Findnext(C, 1, n, x)
  Y ← Findnext(C, 1, n, y)
  return  Y − X
```

The running time is $O(\log n)$ because we are basically calling binary search twice on the array $C[1..n]$.

(b) We need to sum the number of elements in A that are greater than x but at most y. We introduce an array $D[1..k]$. For every $i \in [1, k]$, $D[i]$ should store the number of occurrences of i in $A[1..n]$. This can be done as follows in $O(n + k)$ time. First, initialize $D[i] = 0$ for every $i \in [1, k]$. Then, we scan the elements $A[i]$ for $i = 1, 2, \cdots, n$, and for every $A[i]$ encountered, we increment the element $D[A[i]]$.

Next, we compute an array $C[1..k]$ such that for every $i \in [1, k]$, $C[i]$ stores the sum of elements in A that are at less than or equal to i. This can be done in $O(k)$ time inductively as follows. We first compute $C[1] := D[1]$. Then, for $i = 2, 3, \cdots, k$, we compute $C[i] := i \cdot D[i] + C[i-1]$. Due to the inductive computation, we could have replaced the array $D[1..k]$ by $C[1..k]$ without affecting correctness.

Finally, given a range-sum query $(x, y]$, we return $C[y] - C[x]$ in $O(1)$ time.

***Problem 2:** [20 pts]
(This is a bonus question. The extra points gained by problem 2 will be treated as bonus points.)
Let D be the device that we try to build for 2^{m+1} inputs, as shown in Figure 1. In D, there should be a column of 2^m switches that connect the 2^{m+1} inputs of D to the 2^m inputs of D_0 and the 2^m inputs of D_1. Then, there should be another column of 2^m switches that connect the 2^m outputs of D_0 and the 2^m outputs of D_1 to the 2^{m+1} outputs of D. The constructions of D_0 and D_1 are recursive.
Consider the first column of switches. For all k, input k and input $k+2^m$ of D are connected to the two inputs of a switch S_k in the first column. The outputs of S_k are connected to input k of D_0 and input k of D_1.
Consider the last column of switches. For all k, we connect output k of D_0 and output k of D_1 to the inputs of a switch Z_k in the last column. The outputs of Z_k are connected to output k and output $k+2^m$ of D.
Given a permutation p_i, we show that we can set the switches so that p_i can be realised. We want to color the inputs of D by two colors, red and blue, such that the red inputs are sent to D_0 and the blue inputs are sent to D_1, and the red and blue inputs can be recursively permuted in D_0 and D_1 without conflicts. For every input k of D, we say that another input j of D is in conflict with k if:

- Type 1 conflict: $k = 2^m + j$, or $j = 2^m + k$.
- Type 2 conflict: $p_i(k) = 2^m + p_i(j)$, or $p_i(j) = 2^m + p_i(k)$.

Note that every input k is in type 1 conflict with exactly one other input, and in type 2 conflict with exactly one other input.
When inputs k and j have type 1 conflict, we do not want to send k and j to the same D_0 or D_1. In fact, we cannot due to the connections made by the first column of switches. When inputs k and j have type 2 conflict, we do not want $p_i(j)$ and $p_i(k)$ to appear in the outputs of the same D_0 or D_1. The reason is that D_0 is supposed to send j and k to an output that matches the binary representations of $p_i(j)$ and $p_i(k)$ without the distinct most significant bits (MSB). So type 2 conflict would mean that both j and k would be sent to the same output of D_0, which is impossible to be done.
We color the inputs of D as follows. Color input 0 red. Then, color input 2^m blue (in type 1 conflict with 0). Then, find the input k that is in type 2 conflict with input 2^m. If k is not equal to 0, color input k red and repeat the above. Note that if k is not equal to 0, we must not have coloured k before; otherwise, we must have already discovered two conflicts involving k, one type 1 and one type 2, meaning that there is no other conflict that allows k to be seen again. It is also impossible to assign a blue color to a vertex k that is already red because doing so means that k is involved in two type-1 conflicts, an impossibility. Eventually, we must return to 0, completing a cycle.
If we have not coloured all inputs yet, we pick an arbitrary uncoloured input, color it red, and repeat the above again. In the end, we coloured exactly 2^m inputs red and 2^m inputs blue such that inputs with type 1 or type 2 conflicts receive different colors.
So we can send the red inputs to D_0 and the blue inputs to D_1. Finally, for all b, check the output b of D_0 and output b of D_1. Suppose that inputs j and k of D are sent to outputs b of D_0 and D_1, respectively. So the binary representation of b is the common binary presentation of $p_i(j)$ and $p_i(k)$ without the MSBs. If $p_i(j) = b$ and $p_i(k) = 2^m + b$, set the switch Z_b in the last column so that j is sent to output b of D and k is sent to output $2^m + b$ of D. If $p_i(j) = 2^m + b$ and $p_i(k) = b$, we set that switch Z_b the other way.

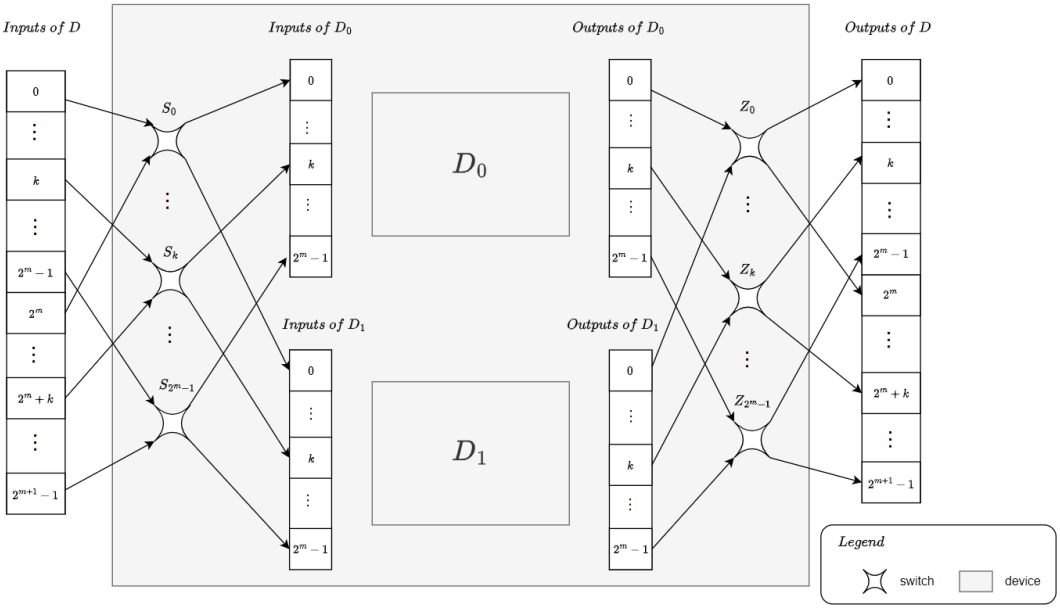

Figure 1: the device for 2^{m+1} inputs

Let $T(n)$ be the number of switches needed to generate n! permutations from n inputs:

$$T(2) = 1; T(n) = \frac{n}{2} + 2T(\frac{n}{2}) + \frac{n}{2} = 2T(\frac{n}{2}) + n \text{ for } n > 2.$$

To count the number of switches, we may use one of the following methods:

- Using the master theorem, $c = \log_b a = \log_2 2 = 1$ and $f(n) = n$, thus $T(n) = O(n \log n)$.
- Using the expansion method, expand the recurrence until a pattern emerges.

$$\begin{aligned} T(n) &= 2T(\frac{n}{2}) + n = 2(2T(\frac{n}{2^2}) + \frac{n}{2}) + n = \\ &= 2^2 T(\frac{n}{2^2}) + 2 \cdot \frac{n}{2} + n = 2^2(2T(\frac{n}{2^3}) + \frac{n}{2^2}) + 2 \cdot \frac{n}{2} + n \\ &= 2^i T(\frac{n}{2^i}) + i \cdot n = ... \\ &= \frac{n}{2} T(2) + (\log_2 n - 1) \cdot n = \frac{n}{2} \cdot 1 + (\log_2 n - 1) \cdot n = O(n \log n). \end{aligned}$$

- Using the recursion tree, sum up the overall workloads at each level.

Hence, the device uses $O(n \log n)$ switches.
Marking Note: Students referencing materials(e.g., websites) should cite the source properly.

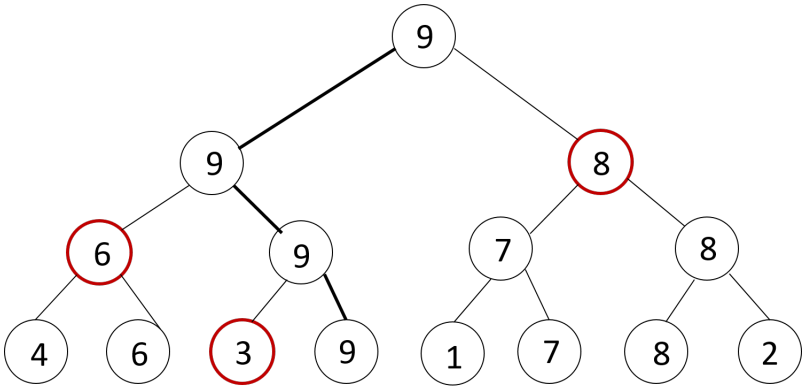

Figure 2: Binary tree for comparison of 8 numbers

Problem 3: [20 pts]
Divide n numbers into $\frac{n}{2}$ pairs, each pair has 2 numbers. Compare the 2 numbers for each pair and select the larger number. For the next round, use the larger number selected from the last round to formulate $\frac{n}{4}$ pairs and select the larger number within each pair again. Iterate this process until we find the largest number among n. The whole comparison process for finding the largest number among n will thus formulate a binary tree with $height = \log_2 n$ from bottom to top. The total number of comparisons required for this process would be the sum of the following geometric series: $\sum_{i=1}^{\log_2 n} \frac{n}{2^i} = n - 1$. Figure 1 is an example of finding the largest number among 8 numbers. The binary tree has a height of 3, and it takes 7 comparisons to find the largest number, which is 9 in this case.

Let's continue to find the second largest number. The second largest number must be smaller than the largest number, but greater than any other number. One important observation is that all the candidate second largest number must have been directly compared with the largest number in the process of finding the largest number, but was beaten by the largest number. In our example, only 3, 6, and 8 (in the red circle in Figure 2) are possible candidates for the second largest number as they have directly compared with 9. The number of candidates' second largest number must be exactly the height of the binary tree which is $\log_2 n$, because for each level of the tree, the largest number will be compared with exactly 1 number. We can thus extract all $\log_2 n$ numbers of candidates' second largest number (we can do this by maintaining a data structure that stores all the comparison pairs along the process), and repeat the process of finding the largest number among all candidate numbers, which will take exactly $\log_2 n - 1$ times of comparisons. Thus, the total number of comparisons for finding the largest and the second-largest number would be exactly $(n - 1) + (\log_2 n - 1) = n + log_2 n - 2$.

It is the same to consider this problem from top to bottom with a divide-and-conquer approach. To find the largest number, we equally divide all numbers into 2 parts and find the largest number among each part respectively. Then an extra comparison is required to find the largest number. Let $C(n)$ be the total number of comparisons required for finding the largest number among n, then by divide-and-conquer, $C(n) = 2C(n/2) + 1$, with the base case $C(2) = 1$. We can derive the same result $C(n) = n - 1$. The rest part for finding the second largest number is the same.

Problem 4: [20 pts]

Proof. We prove that RANDOM-SAMPLE(m,n) returns a subset of $[1, n]$ of size m drawn uniformly at random by proving that for each possible subset $\{a_1, a_2, ..., a_m\}$, the probability of returning $\{a_1, a_2, ..., a_m\}$ is

$$Pr(\text{RANDOM-SAMPLE}(m, n) = \{a_1, a_2, ..., a_m\}) = \frac{1}{C_n^m} = \frac{m! \cdot (n-m)!}{n!}.$$

We will prove this by induction.
When $m = 1$, RANDOM-SAMPLE(1,n) calls RANDOM(1,n) to draw an integer uniformly from $[1, n]$ then for any $a \in [1, n]$, $Pr(\text{RANDOM-SAMPLE}(1, n) = \{a\}) = \frac{1}{n} = \frac{(n-1)!}{n!}$.
When $1 < m \leq n$, RANDOM-SAMPLE(m,n) first calls RANDOM-SAMPLE(m-1,n-1) to return a subset of $[1, n-1]$ of size $m-1$, suppose it is $\{a_1, ..., a_{m-1}\}$. For any such subset $\{a_1, ..., a_{m-1}\}$,
$Pr(\text{RANDOM-SAMPLE}(m-1, n-1) = \{a_1, ..., a_{m-1}\}) = \frac{(m-1)! \cdot (n-m)!}{(n-1)!}$.
Then RANDOM-SAMPLE(m,n) calls RANDOM(1,n) to draw an integer uniformly from $[1, n]$. There are two cases:

- If RANDOM(1,n)$\in \{a_1, ..., a_{m-1}\}$, then the algorithm appends n to the returning subset.
- If RANDOM(1,n)$\notin \{a_1, ..., a_{m-1}\}$, then the algorithm appends i to the returning subset.

Therefore, for any possible subset $\{a_1, ..., a_m\}$ of $[1, n]$:

- If $n \in \{a_1, ..., a_m\}$, without loss of generality, suppose $a_m = n$, then
  $Pr(\text{RANDOM-SAMPLE}(m, n) = \{a_1, ..., a_m\})$
  $= Pr(\text{RANDOM-SAMPLE}(m-1, n-1) = \{a_1, ..., a_{m-1}\}) \cdot Pr(\text{RANDOM}(1, n) \in \{a_1, ..., a_{m-1}\})$
  $+Pr(\text{RANDOM-SAMPLE}(m-1, n-1) = \{a_1, ..., a_{m-1}\}) \cdot Pr(\text{RANDOM}(1, n) = n)$
  $= \frac{(m-1)! \cdot (n-m)!}{(n-1)!} \cdot (\frac{m-1}{n} + \frac{1}{n}) = \frac{m! \cdot (n-m)!}{n!}$.
- If $n \notin \{a_1, ..., a_m\}$, then
  $Pr(\text{RANDOM-SAMPLE}(m, n) = \{a_1, ..., a_m\})$
  $= m \cdot Pr(\text{RANDOM-SAMPLE}(m-1, n-1) = \{a_1, ..., a_{j-1}, a_{j+1}, ..., a_m\})$
  $\cdot Pr(\text{RANDOM}(1, n) = a_j)$
  $= m \cdot \frac{(m-1)! \cdot (n-m)!}{(n-1)!} \cdot (\frac{1}{n}) = \frac{m! \cdot (n-m)!}{n!}$.

Thus, for each possible subset $\{a_1, a_2, ..., a_m\}$, the probability of returning $\{a_1, a_2, ..., a_m\}$ is

$$Pr(\text{RANDOM-SAMPLE}(m, n) = \{a_1, a_2, ..., a_m\}) = \frac{m! \cdot (n-m)!}{n!}.$$

□

Problem 5: [20 pts]

(a) $E[X_i] = Pr(X_i = 1) = \frac{1}{n}$.

(b) *Proof.* If the ith smallest number in the array is chosen as the pivot, i.e., $X_i = 1$, then the algorithm would partition the array to two sub-arrays, one with $i-1$ numbers and another with $n-i$ numbers. The partition costs $\Theta(n)$ and the algorithm continues to do quicksort for the two sub-arrays. Therefore, $E[T(n)] = E[\sum_{i=1}^{n} X_i \cdot (T(i-1) + T(n-i) + \Theta(n))]$. □

(c) *Proof.*

$$\begin{aligned}
E[T(n)] &= E[\sum_{i=1}^{n} X_i \cdot (T(i-1) + T(n-i) + \Theta(n))] \\
&= \sum_{i=1}^{n} Pr(X_i = 1) \cdot E[T(i-1) + T(n-i) + \Theta(n)] \\
&= \frac{1}{n} \cdot (\sum_{i=1}^{n} E[T(i-1)] + \sum_{i=1}^{n} E[T(n-i)]) + \Theta(n) \\
&= \frac{2}{n} \cdot \sum_{i=2}^{n-1} E[T(i)] + \Theta(n)
\end{aligned}$$

□

(d) *Proof.* Assume n is even,

$$\begin{aligned}
\sum_{k=2}^{n-1} k \log k &= \sum_{k=2}^{n/2-1} k \log k + \sum_{k=n/2}^{n-1} k \log k \leq \sum_{k=2}^{n/2-1} k \log \frac{n}{2} + \sum_{k=n/2}^{n-1} k \log n \\
&= \frac{(\frac{n}{2} - 2)(\frac{n}{2} + 1)}{2} \log \frac{n}{2} + \frac{\frac{n}{2}(\frac{3}{2}n - 1)}{2} \log n \\
&= \frac{n^2}{8} \log \frac{n}{2} + \frac{3n^2}{8} \log n - (\frac{n}{4} + 1) \log \frac{n}{2} - \frac{n}{4} \log n \\
&\leq \frac{n^2}{8} (\log n - \log 2) + \frac{3n^2}{8} \log n = \frac{n^2}{2} \log n - \frac{n^2}{8}.
\end{aligned}$$

□

Assume n is odd, it can be proved similarly.

(e) We show $E[T(n)] \leq cn\log n$ for some positive constant c by induction.
First, when $n = 2$, it is true. Suppose $E[T(k)] \leq ck \log k$ for $k = 2, ..., n-1$, then by (c),

$$E[T(n)] = \frac{2}{n} \cdot \sum_{i=2}^{n-1} E[T(i)] + \Theta(n) \leq \frac{2}{n} \cdot \sum_{i=2}^{n-1} ck \log k + \Theta(n).$$

By (d), we further have

$$E[T(n)] \leq \frac{2c}{n} \cdot \sum_{i=2}^{n-1} k \log k + \Theta(n) \leq \frac{2c}{n} \cdot (\frac{n^2}{2} \log n - \frac{n^2}{8}) + \Theta(n) = cn\log n - \frac{cn}{4} + \Theta(n).$$

When n is sufficiently large, $E[T(n)] \leq cn \log n$.

Then, we show $E[T(n)] \geq mn\log n$ for a positive constant m.
When $n = 2$, it is true. Suppose $E[T(k)] \geq mk \log k$ for $k = 2, ..., n-1$,

$$\sum_{k=2}^{n-1} k \log k > \sum_{k=\lceil \frac{n}{2} \rceil}^{n-1} k \log k \geq \frac{(\lceil \frac{n}{2} \rceil + n - 1) * \lceil \frac{n}{2} \rceil}{2} log \lceil \frac{n}{2} \rceil \geq (\frac{3n^2}{8} - \frac{n}{4}) log \frac{n}{2}.$$

By (c), we have

$$E[T(n)] = \frac{2}{n} \sum_{i=2}^{n-1} E[T(i)] + \Theta(n) \geq \frac{2}{n} * (\frac{3n^2}{8} - \frac{n}{4}) * log \frac{n}{2} + \Theta(n) \geq (\frac{3}{4}n - \frac{1}{2}) log \frac{n}{2} + \Theta(n).$$

When n is sufficiently large, $E[T(n)] \geq mn\log n$. Thus, $E[T(n)] = \Theta(n \log n)$.