COMP 3711 – Design and Analysis of Algorithms
2022 Fall Semester – Written Assignment Solution # 2

Solution 1: [10 points]
For $i < j$ set

$$X_{i,j} = \begin{cases} 1 & \text{if } A[i] > A[j], \\ 0 & \text{otherwise.} \end{cases}$$

Now set $X = \sum_{i=1}^{n-1} \sum_{j=i+1}^{n} X_{ij}$ to be the total number of inversions.
For any fixed i, j, exactly half of the set of $n!$ permutations have $A[i] > A[j]$, so

$$E(X_{ij}) = Pr(X_{ij} = 1) = \frac{1}{2}.$$

Thus

$$\begin{aligned} E(X) &= E(\sum_{i=1}^{n-1} \sum_{j=i+1}^{n} X_{ij}) \\ &= \sum_{i=1}^{n-1} \sum_{j=i+1}^{n} E(X_{ij}) \\ &= \sum_{i=1}^{n-1} \sum_{j=i+1}^{n} \frac{1}{2} \\ &= C_n^2 \cdot \frac{1}{2} = \frac{n(n-1)}{2} \cdot \frac{1}{2} = \frac{n(n-1)}{4}. \end{aligned}$$

Solution 2: [10 points]

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 $\{a_1, a_2, \ldots, a_m\}$, the probability of returning $\{a_1, a_2, \ldots, a_m\}$ is

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

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 $\{a_1, \ldots, a_{m-1}\}$. For any such subset $\{a_1, \ldots, a_{m-1}\}$, $Pr(\text{RANDOM-SAMPLE}(m-1, n-1) = \{a_1, \ldots, a_{m-1}\}) = \frac{(m-1)! \cdot (n-m)!}{(n-1)!}$.
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)$\in \{a_1, \ldots, a_{m-1}\}$, then the algorithm appends n to the returning subset.
- If RANDOM(1,n)$\notin \{a_1, \ldots, a_{m-1}\}$, then the algorithm appends i to the returning subset.

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

- If $n \in \{a_1, \ldots, a_m\}$, without loss of generality, suppose $a_m = n$, then

$$\begin{aligned}
&Pr(\text{RANDOM-SAMPLE}(m,n) = \{a_1, \ldots, a_m\}) \\
=&Pr(\text{RANDOM-SAMPLE}(m-1,n-1) = \{a_1, \ldots, a_{m-1}\}) \cdot Pr(\text{RANDOM}(1,n) \in \{a_1, \ldots, a_{m-1}\}) \\
+&Pr(\text{RANDOM-SAMPLE}(m-1,n-1) = \{a_1, \ldots, a_{m-1}\}) \cdot Pr(\text{RANDOM}(1,n) = n) \\
=&\frac{(m-1)! \cdot (n-m)!}{(n-1)!} \cdot \frac{m-1}{n} + \frac{(m-1)! \cdot (n-m)!}{(n-1)!} \cdot \frac{1}{n} = \frac{m! \cdot (n-m)!}{n!}.
\end{aligned}$$

- If $n \notin \{a_1, \ldots, a_m\}$, then

$$\begin{aligned}
&Pr(\text{RANDOM-SAMPLE}(m,n) = \{a_1, \ldots, a_m\}) \\
=&m \cdot Pr(\text{RANDOM-SAMPLE}(m-1,n-1) = \{a_1, \ldots, a_{j-1}, a_{j+1}, \ldots, a_m\}) \cdot Pr(\text{RANDOM}(1,n) = \{a_j\}) \\
=&m \cdot \frac{(m-1)! \cdot (n-m)!}{(n-1)!} \cdot \frac{1}{n} = \frac{m! \cdot (n-m)!}{n!}.
\end{aligned}$$

Thus, we prove that for each possible subset $\{a_1, a_2, \ldots, a_m\}$, the probability of returning $\{a_1, a_2, \ldots, a_m\}$ is

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

□

Solution 3: [10 points]

(a) In the binary tree, the indices of the elements of A at level j are $[2^j, \ldots, 2^{j+1} - 1]$.
Since $h = L - l(i)$, the indices that have height h are $[2^{L-h}, \ldots, 2^{L-h+1} - 1]$.
The number of the elements that have height h is $2^{L-h+1} - 1 - 2^{L-h} + 1 = 2^{L-h}$.

(b) **Update-Subtree**(A, i): given that the subtrees rooted at the children of $A[i]$ are already binary min-heaps, update the subtree rooted at $A[i]$ so that this subtree becomes a binary min-heap.

```
Update-Subtree(A, i):
begin
    l = A[2i]; r = A[2i + 1];
    while A[i] > min(l, r) and i < 2^(log2(n+1)-1) do
        // if A[i] larger than a child and A[i] is not a leaf node, swap with min child
        if l < r then
            Swap A[i] with A[2i]; i = 2i;
        else
            Swap A[i] with A[2i + 1]; i = 2i + 1;
        l = A[2i]; r = A[2i + 1];
end
```

After each swap, the min-heap property is satisfied for all nodes except the node containing $A[i]$ (with respect to its children). So when the while loop ends, either $A[i]$ is smaller than both children, or it reaches the level L, the subtree becomes a binary min-heap.
Time complexity=O(height of $A[i]$).

(c) **Build-Binary-Min-Heap**(A, i):
if $i \geq 2^{\log_2(n+1)-1}$ **then** // $A[i]$ and $A[n]$ are in the same level

return
Build-Binary-Min-Heap$(A, 2i)$
Build-Binary-Min-Heap$(A, 2i+1)$
Update-Subtree(A, i)
First call: **Build-Binary-Min-Heap**$(A, 1)$

Correctness: For all leaf nodes, that are $A[i]$ where $i \geq 2^{\log_2(n+1)-1}$, the subtree rooted at $A[i]$ is a binary min-heap with only one node.
For other node $A[i]$, the algorithm builds the binary min-heaps rooted at the children of $A[i]$, then updates the subtree rooted at $A[i]$ so that it becomes a binary min-heap.
Thus the algorithm recursively builds a binary min-heap for $A[1 \ldots n]$.
Running time: $T(1) = 1$, $T(n) = 2T(n/2) + O(\log_2 n)$.
By master theorem, $c = \log_2 2 = 1$, $f(n) = O(\log_2 n) = O(n)$, so $T(n) = O(n)$.

Solution 4: [10 points]
For a group of time intervals, we call the group accessible if the intersection of all these time intervals is non-empty, which means these intervals can be hit by a single time instant in their intersection. A partition of $\{I_1, \ldots, I_n\}$ is optimal if each part of the partition is accessible and we cannot merge any two parts to get an accessible group. The underlying idea of the algorithm is simple, i.e., find the optimal partition.
First, we sort all time intervals in $\mathcal{I} = \{I_1, \ldots, I_n\}$ according to their finishing times. Second, scan from the first time interval. Once we find an integer j such that $\{I_1, \ldots, I_j\}$ is accessible and $\{I_1, \ldots, I_{j+1}\}$ is not accessible, we maintain $\{I_1, \ldots, I_j\}$ as an accessible group and start a new accessible group construction from I_{j+1}. The pseudocode is as follows.

Sort $\mathcal{I} = \{I_1, \ldots, I_n\}$ according to finishing times
$R \leftarrow \emptyset$ // R stores the maximal accessible groups
$G \leftarrow \emptyset$
for $i \leftarrow 1$ to n **do**
  if $G \cup \{I_i\}$ is accessible **then**
    $G \leftarrow G \cup \{I_i\}$
  else
    $R \leftarrow R \cup \{G\}$
    $G \leftarrow \{I_i\}$
return R

Since each element $G_j \in R$ is accessible, we can construct a set $\{t_1, \ldots, t_{|R|}\}$ of time instances such that t_j hits all time intervals in G_j, where $|R|$ is the cardinality of R. For $j \in [1, \ldots, |R|]$, we identify t_j as the latest time instance that hits all time intervals in G_j. Finally, $\{t_1, \ldots, t_{|R|}\}$ is returned as the solution.
Correctness: We assume that the smallest set of time instances that hit all intervals in $\mathcal{I}$ is $O = \{o_1, o_2, \ldots, o_k\}$, where $o_1 < o_2 < \cdots < o_k$. Then $k \leq |R|$.
If $o_i = t_i$ for $1 \leq i \leq k$, then $k = |R|$. Otherwise G_{k+1} cannot be hit by O since G_{k+1} cannot be hit by $\{t_1, \ldots, t_k\}$. Therefore, in this case, the greedy solution is the optimal solution.
If there exists r such that $o_r \neq t_r$, we look at the first such r where the two sets of time instances differ, i.e., $t_i = o_i$ for $1 \leq i \leq r-1$ and $t_r \neq o_r$.
We now modify O by replacing o_r with t_r.
Suppose $G_r = \{I_p, \ldots, I_q\}$. According to our greedy algorithm, the following properties are satisfied:

- $\bigcap_{p \leq i \leq q} I_i \neq \emptyset$.

- $\bigcap_{p \le i \le q+1} I_i = \emptyset$.

Suppose $\{o_1, \ldots, o_r\}$ hits $\{I_1, \ldots, I_m\}$ and cannot hit I_{m+1}. Since the maximum group of intervals that $\{t_1, \ldots, t_{r-1}\}$ (which is $\{o_1, \ldots, o_{r-1}\}$) can hit is $\{I_1, \ldots, I_{p-1}\}$, so o_r must hit $\{I_p, \ldots, I_m\}$. According the our greedy algorithm, we have $q \ge m$. Otherwise the intersetion of $\{I_p, \ldots, I_m\}$ is empty.

That means $\{o_1, \ldots, o_{r-1}, t_r\}$ also hits $\{I_1, \ldots, I_m\}$. So after the modification, the new O is still an optimal solution. Therefore, O's first index that differs from $\{t_1, \ldots, t_{|R|}\}$ is now at least $r+1$.
Repeating this process will eventually convert O into $\{t_1, \ldots, t_{|R|}\}$. At the end, $O = \{t_1, \ldots, t_{|R|}\}$. So the greedy algorithm is optimal.
Running time: To judge whether $G \cup \{I_i\}$ is accessible and to get t_j cost $O(i)$ since we need to compute the intersection of all intervals in G and I_i. So the algorithm costs $O(n^2)$ running time.

Solution 5: [10 points]

(a) The completion time of $A_{\sigma(j)} = \sum_{i=1}^{j} t_{\sigma(i)}$

(b) Let σ be the permuatition where $t_{\sigma(1)} < t_{\sigma(2)} < \cdots < t_{\sigma(n)}$. Assume the optimal permutation that minimizes the total completion time is σ'.
We find the largest value r such that $\sigma(i) = \sigma'(i)$ for $1 \le i \le r$ and $\sigma(r+1) \ne \sigma'(r+1)$.
The completion time of $A_{\sigma(r+1)} = \sum_{i=1}^{r+1} t_{\sigma(i)}$ is less than the completion time of $A_{\sigma'(r+1)}$ since $t_{\sigma(r+1)} < t_{\sigma'(r+1)}$.
Suppose $\sigma'(q) = \sigma(r+1)$ for some $q > r+1$. Then we create σ^* from σ' by swapping $\sigma'(r+1)$ and $\sigma'(q)$.
(The total completion time of σ')$-$ (the total completion time of σ^*) $= (t_{\sigma'(r+1)} - t_{\sigma(r+1)}) \cdot (q-r-1) > 0$. That is, σ^* has less total completion time than the optimal permutation σ'.
The contradiction shows that σ is the optimal permutation that minimizes the total completion time. Actually repeating the above process will eventually convert σ' into σ.