Assignment-5

Design and Analysis of Algorithms

Trishan Mondal

⚠ **Disclaimer.** Consider the following set of students

$$\mathcal{P} = \{\text{Trishan, Eeshan Pandey, Soumya Dasgupta, Priyatosh Jana}\}$$

Discussion of solutions to the assignment problems are limited to the people of set $\mathcal{P}$ only. Most of the problems in this assignment has general solution. If any other person have same solution as mine is not my fault.

§ Problem 20

Problem. **(Greedy Scheduling)** There are n tasks, $T_1, T_2, \cdots, T_n$. We are given n pairs

$$(d_1, p_1), (d_2, p_2), ..., (d_n, p_n)$$

where $d_i \in \{1, 2, ..., n\}$ refers to the deadline of the i-th task T_i, and p_i is the penalty if T_i is not performed by the deadline. Each task needs one unit-length timeslot. We wish to assign to each task a different timeslot s_i in the range $\{1, 2, ..., n\}$. A task i is delayed under this assignment if $d_i < s_i$. The cost of the assignment is

$$\sum_{j:T_j \text{ is delayed}} p_j$$

Show that the following greedy strategy produces an optimal solution:

> Consider the tasks in the monotonically decreasing order of their penalties (tasks with higher penalty earlier). When considering task T_i, determine if some timeslot that helps it meet the deadline d_i is still available. If there is such a slot, set s_i to be the last slot that still allows it to meet the deadline. Otherwise, schedule T_i in the last available slot.

State your argument by carefully establishing that *at each stage, there is an optimal solution that extends the current partial assignment of tasks to slots.* Describe how you will implement this strategy as an algorithm and analyze the worst-case time complexity of your algorithm.

Solution. WLOG, $p_1 \geq p_2 \geq \cdots p_n$. We want to assing an array of time slots $s_1, \cdots, s_n$ to the tasks, $T_1, \cdots, T_n$ so that,

$$\sum_{j=1}^{n} p_j \mathbb{1}[d_j < s_j]$$

get minimized. We will prove the correctness of the greedy algorithm by induction. Here the induction hypothesis is: The greedy strategy gives us an optimal assignment that agrees upto i-th step. For the base case $i = 1$ there is nothing to prove. Let $\{s_k\}$ for $k \in \{1, \cdots, n\}$ be the optimal solution that agrees upto i-th step. We will show the greedy strategy will give us an optimal assignment $\{s'_k\}$ for $k \in \{1, \cdots, n\}$ that agrees upto $(i + 1)$-th step. To prove this we have to deal with different cases:

1. **The greedy solution fails to schedule T_{i+1} before its deadline d_{i+1} :** In this case we must have used all the slots from 1 to d_{i+1} while scheduling $T_1, \cdots, T_i$. The solution $\{s_k\}$ fails to assign T_{i+1} before d_{i+1} so in this case just take $\{s_k\} = \{s'_k\}$ which will give us optimal solution that agrees upto $(i+1)$-th step.

2. **The greedy solution schedules T_{i+1} before its deadline d_{i+1}:**

   – Assume that the greedy algorithm assigns T_{i+1} to slots $s < d_{i+1}$. The assignment $\{s_k\}$ has no choice but to schedule T_{i+1} within the slots from 1 to s. If $\{s_k\}$ indeed schedules T_{i+1} in slot s, we can simply set $\{s_k\} = \{s'_k\}$. However, if that's not the case, it implies that $\{s_k\}$ schedules some other task T_j in slots for some $j > i+1$, given that $\{s_k\}$ aligns with the greedy solution up to stage i. In this scenario, by exchanging the slots assigned to T_j and T_{i+1}, we observe that the resulting solution remains optimal. The cost can only decrease as T_{i+1} is scheduled in earlier slots than T_j, while the scheduling of the remaining tasks T_i remains unchanged. This process allows us to construct a valid assignment $\{s'_k\}$, thereby establishing the validity of induction hypothesis at $(i+1)$-th step.

   – If the greedy algorithm schedules T_k in slots $s < d_k$, it implies that in this scenario, $\{s_k\}$ allocates some slot to a task T_j with $j > i+1$ as $\{s_k\}$ follows the greedy solution up to stage i. However, we can exchange the slots assigned to T_j and T_{i+1}. Here, T_{i+1} is assigned before d_{i+1} but T_j can exceed deadline. By the assumption $p_j \leq p_{i+1}$, so the solution $\{s'_k\}$ is optimal too. Thus this gives us an assignment which is optimal upto $(i+1)$-th position.

   Thus the proof of correctness is complete.

Algorithms and Time complexity

- `Input`: Array of (d_i, p_i), where d_i is the deadline of task T_i and p_i is the penalty if the work hasn't done within d_i.
- `Output`: An assignment of tasks to slots s_i such that $\sum_{j=1}^{n} p_j \mathbb{1}[d_j < s_j]$ is minimized.

- Sort Tasks in decreasing order of penalties i.e. $p_1 \geq p_2 \geq \ldots \geq p_n$.
- Initialize an array s to represent time slots with all values set to `NULL`.
- For $i = 1$ to n:
  – **If:** d_i (corresponding to p_i in the sorted array) is available in the time slot s_i, then assign T_i to the time slot s_i.
  – **Else:** Assign task T_i to the last available time slot s_j where j is the largest such that $s_j =$ `NULL`.

  Mark time slot s_i as unavailable.
- **Return:** The assignment of tasks to time slots.

Time complexity. The above algorithm takes $O(n \log n)$ time to sort the array $P = [p_i]$ and the subsequent `for` loop takes $O(n^2)$ time (as each iteration takes $O(n)$ time to find the maximum j such that $s_j =$ `NUll`). So the time complexity is $O(n^2)$.

§ Problem 21

Part(I): The given algorithm produce the following codewords for the sequence $(\ell_1, \cdots, \ell_8) = (3, 4, 3, 4, 2, 3, 4, 3)$:

$1 : S = \{\Lambda\}$	
$2 : S = \{1, 01, 001\}$	$w_1 = 000$
$3 : S = \{1, 01, 0011\}$	$w_2 = 0010$
$4 : S = \{1, 0011, 011\}$	$w_3 = 010$
$5 : S = \{1, 01, 0011\}$	$w_4 = 0011$
$6 : S = \{011, 11\}$	$w_5 = 10$
$7 : S = \{11\}$	$w_6 = 011$
$8 : S = \{111, 1101\}$	$w_7 = 1100$
$9 : S = \{1101\}$	$w_8 = 111$

Part(II): Let, S_i be the set S at the i-th iteration step. And let there are total n number of iteration (i.e. there are given n positive integer $\ell_1, \cdots, \ell_n$). Considering the algorithm never stuck we will prove the algorithm is correct. In the next part we will show the algorithm never stuck (maybe we can use the hint given in the question). Before proving the correctness we want to prove the **Claim:** After each iteration S_i is prefix free and S_i contains no prefix of $\{w_1, \cdots, w_n\}$ (here i runs through 1 to n).

Proof of claim: We will prove this by induction on i. The base case is $S_0 = \{\Lambda\}$, in this case the statement is trivially true. Let S_k is prefix free which is the induction hypothesis.

If $|w| = \ell = \ell_k$: In this situation, we have $\{w_1, \cdots, w_k\} = \{w_1, \cdots, w_{k-1}\} \cup \{w\}$ and $S_k = S_{k-1} \setminus \{w\}$. If S_k were to contain a prefix of $\{w_1, \cdots, w_k\}$, that prefix would need to be part of w, which would violate the prefix-free property of S_{k-1}, leading to a contradiction. It is evident that $S_k = S_{k-1} \setminus w$ maintains the prefix-free property because it is a subset of S_{k-1}, which is guaranteed to be prefix-free by the induction hypothesis.

If $|w| = \ell < \ell_k$: In this scenario, we have $\{w_1, \cdots, w_k\} = \{w_1, \cdots, w_{k-1}\} \cup \{w\}$ and $S_k = (S_{k-1} \setminus \{w\}) \cup \{w1, w01, \ldots, w0 \ldots w\underbrace{000000..0}_{\ell_i - \ell - 1}1\}$. If S_k were to contain a prefix of an element x in $\{w_1, \cdots, w_k\}$, that prefix could not belong to $S_{k-1} \setminus \{w\}$ because if it did, and $x = w$, S_{k-1} would lose its prefix-free property. Additionally, if $x \in \{w_1, \cdots, w_{k-1}\}$, such a scenario would contradict the induction hypothesis. The prefix cannot belong to $\{w1, w01, \ldots, \underbrace{w0 \ldots 0}_{\ell_k - \ell - 1}1\}$ because, in that case, it implies that $x = w$. Consequently, $w\underbrace{0 \ldots 0}_{p}1$ would be a prefix of an element in $\{w_1, \cdots, w_{k-1}\}$ for some $0 \leq p \leq \ell_k - \ell - 1$. However, this would also mean that w is a prefix of that element, leading to a contradiction with S_{k-1} containing a prefix of $\{w_1, \cdots, w_{k-1}\}$, which contradicts the induction hypothesis. Thus, S_k must not include any prefixes of $\{w_1, \cdots, w_k\}$. Additionally, S_k must be prefix-free. To understand this, consider that $S_{k-1} \setminus \{w\}$ is prefix-free, being a subset of S_{k-1} which is prefix-free according to the induction hypothesis. It is also evident that $\{w_1, w_{01}, \ldots, w\underbrace{0 \ldots 0}_{\ell_k - \ell - 1}1\}$ is prefix-free. Furthermore, no element x from $S_{k-1} \setminus \{w\}$ can be a prefix of $\{w1, w01, \ldots, w\underbrace{0 \ldots 0}_{\ell_k - \ell - 1}1\}$. If it were, and assuming $\text{len}(x) < \text{len}(w)$, then x would be a prefix of w. Alternatively, if $\text{len}(x) \geq \text{len}(w)$, then w would be a prefix of x. In either scenario, S_{k-1} would lose its prefix-free property.

Lastly, no element within $\{w1, w01, \ldots, w_0 \ldots 0_{\ell_k - \ell - 1}1\}$ would be a prefix of an element in $S_{k-1} \setminus \{w\}$ because that would imply w is a prefix of the same element in S_{k-1}, which leads to a contradiction.

Thus be induction we have proved that $\{w_1, \cdots, w_n\}$ are prefix free coding of length $\ell_1 \cdots, \ell_n$ respectively. Now we will prove the algorithm never stuck i.e $\ell \leq \ell_i$ before each iteration step i. To prove this we will prove the following lemma (in the proof of lemma w and ℓ are the maximum length string in S_i and their length respectively):

Claim (a): All the strings in S_i have distinct length.

We will again proceed by induction. This statement holds for $i = 0$ since S_0 contains only the empty word. Let's assume that it holds for some $i \geq 0$. At the next step, we can have two scenarios. Either S_{i+1} is formed by excluding w from S_i, or it is formed by excluding w and adding words like $\{w1, \ldots, w0\ldots01\}$, where the last word added in the second case contains $\ell_{i+1} - \ell - 1$ trailing zeros. In the first case, S_{i+1} is a subset of S_i, which means it consists of words with distinct lengths. In the second case, we encounter two words of the same length in S_{i+1} if there exists a word x in S_i (after removing w) such that $|x| = \ell + s$ for some $1 \leq s \leq \ell_{i+1} - \ell$. In other words, $\ell + 1 \leq |x| \leq \ell_{i+1}$. This contradicts our assumption that ℓ is the maximum length below ℓ_{i+1} among the words in S_i. Therefore, S_{i+1} consists of words with distinct lengths. By induction, this holds for all i.

Claim (b): For all i the following inequality holds:

$$\sum_{j>i} 2^{-\ell_j} \leq \sum_{w' \in S_i} 2^{-|w'|}$$

We will again use induction to prove this (ᵕ̈). For $S = \{\Lambda\}$ it is true as $\sum 2^{-\ell_i} \leq 1$ according to the condition given in the question. Assume it holds for some $i \geq 0$. If $S_{i+1} = S_i \setminus w$, the inequality holds for $i+1$ as the same term is subtracted from both sides. Now assume $\ell < \ell_{i+1}$ so that $S_{i+1} = (S_i \setminus w_i') \cup w1, \ldots, w0\ldots01$. For the first case we have,

$$\sum_{j>i} 2^{-\ell_j} \leq \sum_{w' \in S_i} 2^{-|w'|} \implies \sum_{j>i+1} 2^{-\ell_j} \leq \sum_{w' \in S_i \setminus w} 2^{-|w'|} + \left(2^{-\ell} - 2^{-\ell_{i+1}}\right)$$

The later term is zeronegative as $\ell = \ell_{i+1}$. Thus, the inequality holds. For the second case where $S_{i+1} = (S_i \setminus w) \cup \{w1, \cdots, w000..1\}$ note that,

$$\sum_{w' \in S_{i+1}} 2^{-|w'|} = \sum_{s=1}^{\ell_{i+1}-\ell} 2^{-(\ell+s)} + \sum_{w' \in S_i \setminus w} 2^{-|w'|} = 2^{-\ell} - 2^{-\ell_{i+1}} + \sum_{w' \in S_i \setminus w} 2^{-|w'|} > \sum_{j>i+1} 2^{-\ell_j}$$

By induction we are done.

From the inequality in claim(b), we observe that $\sum_{w \in S_i} 2^{-w} \geq \sum_{j>i} 2^{-\ell_i}$. Due to (a), the words in S_i all possess distinct lengths. As a result, the sum on the left can be interpreted as a binary expansion. Hence, there exists $x \in S_i$ such that $|x| \leq \ell_{i+1}$ since the inequality cannot hold otherwise. Consequently, S contains w_i' for all $0 \leq i < n$, ensuring that the algorithm never encounters an impasse. ■

§ Problem 22

Problem. Suppose the edges of a graph on a vertex set $\{1, 2, \ldots, n\}$ are stored on a tape in the form $e_1, e_2, \ldots, e_m$. Design an algorithm that uses $O(n)$ space (assuming that vertices and pointers can be stored in one cell of memory) and, after performing one scan of the tape, determines if the graph is bipartite. (Hint: you might want to use a union-find data structure to keep track of the color classes of the graph as it is being built edge by edge.)

Solution. For simplicity we will work with connected graph. To determine if the graph is bipartite as the edges are being scanned, we can utilize a union-find data structure with $O(n)$ space. We will maintain two color classes, say 'red' and 'blue', to represent the bipartition.

Input: $G = (V, E)$.

output: If the graph G is bi-partite.

- **Step 1**: For each edge $e_i \in E$:

  we will check if the two vertices it connects are in the same color class. If they are, then adding e_i would create an odd-length cycle, making the graph non-bipartite. In such a case, we can conclude that the graph is not bipartite. And return FALSE.

- **Step 2**: If the two vertices of e_i are in different color classes, we can safely assign one of them to the 'red' class and the other to the 'blue' class. This ensures that no odd-length cycle is formed.

- **Finally**: By the end of the For loop, if we have not encountered any conflicts (i.e., vertices that should be in the same color class but are not), the graph is bipartite. Then we would return TRUE.

CORRECTNESS AND TIME COMPLEXITY. The way we described the algorithm, the correctness immediately follows. We can also see the algorithm terminates after checking all the edges. The for loop runs m-times and inside the loop every work can be done in constant time. So the time complexity is $O(m)$. We can implement the above algorithm in psudo-code:

```
function bipartite(G):
        # Input : G = (V,E)
        # Output : TRUE or FALSE according the graph is bipartite or not
    for v ∈ V:
       makeset(v)
       color[v] = Nil
    for {u,v} ∈ E:
     if Find(u) = Find(v):
       Return FALSE
     else:
       if color(u) = Nil:
         color[u]=v
       else:
         Union(color(u),v)
       if color(v) = Nil:
         color[v]=u
       else:
         Union(color(v),u)
    return TRUE
```

§ Problem 23

Solution. We will use dynamic programming method to find the optimal binary search tree as following:

Consider an array $W = \{w_1, w_2, \ldots, w_n\}$ of words in sorted order, along with an array $P = \{p_1, \cdots, p_n\}$ representing their corresponding frequencies. We will maintain two arrays: T, storing binary search trees, and C, storing their associated costs. For $i \leq j$, $T[i, j]$ represents the binary search tree with the minimum cost that contains words from w_i to w_j, or is an empty tree if $i > j$ (in such cases, we will not populate entries where $j < i$). Initially, we set $T[i, i]$ to singleton trees, each consisting solely of the word w_i. The corresponding tree costs are given by the respective frequencies p_i.

We will compute the remaining $T[i, j]$ values diagonally. Starting with $T[1, 1], \ldots, T[n, n]$, we continue with the super diagonal $T[1, 2], \ldots, T[n-1, n]$, and so on. To compute $T[i, j]$, we select a word w_k from the range w_i to w_j, where k varies from i to j. We calculate the cost of the tree with w_k as the root vertex, with its left subtree containing words from $w_i, .., w_{k-1}$, and the right subtree containing words from $w_{k+1}, \cdots, w_j$. The tree with the minimum cost is assigned as $T[i, j]$, and its cost is stored in $C[i, j]$.

We will write the above algorithm as a psudo-code as following, where we will give input $W = [w_1, \cdots, w_n]$ in sorted order with their frequencies $P = [p_1, \cdots, p_n]$ and out put will be an optimal binary search tree

```

function obst(W,P)

# Initialize T and C arrays
  T[i,j] = Null for 1 ≤ i ≤ j ≤ n        #Initialize such an array
    A[i,j] = 0 for 1 ≤ i ≤ j ≤ n           #Initialize such an array

# Initialize singleton trees and their costs
    for i in 1 to n:
                T[i,i] = w_i    # This will store root for the sub problem {w_i ··· , w_j}
    for i in 1 to n:
                C[i,i] = p_i     # This will store cost for the sub problem {w_i ··· , w_j}

# Loop to compute T and C values
for i in 1 to n-1:
    for j in 1 to n-i:
        sum_r = sum(P[i:j])
        min_cost = ∞
        min_root = Null

        for k in i to i+j:
           if(sum_r+C[j,k-1]+C[k+1,d]<min_cost):
                min_cost = C[j,k-1]+C[k+1,d]
                min_root = k
            T[j,j+i] = tree (leftt=T[j,min_root-1],root=min_r,rightt=T[min_root+1,i]) #tree(
    left,root,right) forms a tree with root, with ‘left’, ‘right’ subtree.
            C[j,j+i]=min_c+sum_r

   return T[1,n], C[1,n]
```

CORRECTNESS AND TIME COMPLEXITY: The correctness of the algorithm is evident through its recursive nature and the inherent property that the minimal binary search tree must have one of its elements as its root. Furthermore, the final tree generated is guaranteed to be a binary search tree due to the fact that the array W is sorted. Each iteration of the outermost 'for' loop runs at most 'n' times, and there are three nested 'for' loops. All other operations inside the loops have a time complexity of $O(1)$. Therefore, the algorithm exhibits a time complexity of $O(n^3)$, making it an efficient solution.

§ Problem 24

Problem. You are tasked with preparing a five-volume collection of articles on algorithms. The available articles are of varying lengths and are denoted as $\ell_1, \ell_2, \ell_3, \ldots, \ell_n$, where ℓ_1 corresponds to the first article published on the subject, ℓ_2 corresponds to the next article, and ℓ_n corresponds to the most recent article. The following constraints must be satisfied:

- No volume is allowed to have more than 300 pages.
- Every selected article must start on a fresh page and must appear completely in one volume.
- All articles in volume $i+1$ must have been published after all the articles in volume i.

Describe a dynamic programming-based method to determine a plan for publishing the maximum number of articles in the five-volume collection while adhering to the above constraints. The output should indicate which articles go into each volume. Provide an estimate of the time complexity of your algorithm.

Solution. Let's define $A[p,m]$ be the maximum number of article from $\{1, \cdots, m\}$ that can be accommodated in pages $1, \cdots, p$ while respecting the conditions, $A[p,m] = 0$ whenever $p \leq 0$ or $m \leq 0$. For $p, m \geq 1$ we can compute $A[p,m]$ by considering three cases

- We don't include article m

- We include article m and it's last page falls on p. In this case the page $p - \ell - 1$ and p must be in the same volume.
- We include article m but it falls before the page p

Combining these cases we can write,

$$A[p,m] = \max \begin{cases} A[p, m-1] \\ A[p - \ell_m, m-1] + 1 & \text{if } \lceil \frac{p-\ell_m}{300} \rceil = \lceil \frac{p-1}{300} \rceil \\ A[p-1, m] \end{cases}$$

The second term arises from the second case mentioned above and the fact that every volume can have at-most 300 pages. We can compute $A[p,m]$ by Initializeing two for loops:

for $m = 1, \cdots, n$: $A[0,m] = 0$

for $p = 1, \cdots, 1500$: $A[p, 0] = 0$

- Then compute $A[p,m]$ for other entries using the above recurrence

  for $m = 1, \cdots, n$:

  for $p = 1, \cdots, 1500$:

  compute $A[p,m]$ using the recurrence.

- Then return $A[1500, n]$.

- The above will give us $A[n, 1500]$ and adding those articles to volumes one by one as follows: Suppose some new article is encountered, of length ℓ. Let x be the sum modulo 300 of the lengths of all the articles encountered before this. We add the article encountered to the current volume if $x + \ell < 300$ and to a new volume otherwise.

CORRECTNESS AND TIME COMPLEXITY. The correctness follows from the description of the algorithm and it must terminate due to the recursive nature. While computing $A[1500, n]$ we have two loops and inside the loop, the works can be done in constant time. Thus, the time complexity for this case is $O(1500n)$. And the space complexity is also $O(1500n)$. ■