Assignment 3

Algorithmic Paradigms

Name: Dipam Sen
Roll No: 24CS10059

Note: Sets and arrays are assumed to be 1-indexed, unless otherwise mentioned.

3.6.1 **(Heaviest Sparse Set)** Let S be an ordered set of n elements, with each one having a positive weight. A subset $S' \subseteq S$ is called a **sparse set** if it contains no two consecutive elements of S; its weight is defined as the sum of the weights of its elements.
Design an algorithm to compute the weight of a heaviest sparse subset of S. Prove its correctness, and analyze its time and space complexities. Show a demonstration of your algorithm to obtain the output for the input set $S = \{1, 5, 3, 3, 5, 7, 3\}$.

Sol. Denote the weights of elements of S as $w_1, w_2, \ldots, w_n$.

Optimal Substructure: Consider S' to be a heaviest sparse set of $S[1..i]$. This subset either contains the ith element, or it doesn't.

- If ith element is contained in it, then the $(i-1)$th element cannot be contained in it. Other than the ith element, the remaining elements form a subset of $S[1..i-2]$. This must be a heaviest sparse set of $S[1..i-2]$. (Otherwise, we could use a heavier sparse set of $S[1..i-2]$ to get a heavier sparse set of $S[1..i]$, a contradiction).
- If ith element is not contained in it, the set S' is a subset of elements of $S[1..i-1]$. Then S' must be a heaviest sparse set of $S[1..i-1]$. (Otherwise, we could use a heavier sparse set of $S[1..i-1]$ to get a heavier sparse set of $S[1..i]$, a contradiction).

Denote the weight of a heaviest sparse set of $S[1..i]$ as $f[i]$. Then,

$$\begin{aligned} f[i] &= \max(f[i-1], f[i-2] + w_i) && i \geq 2 \\ &= w_1 && i = 1 \\ &= 0 && i = 0 \end{aligned}$$

We can compute $f[i]$ using bottom-up DP:

Algorithm 1: Heaviest Sparse Set

```
1  procedure HEAVIEST-SPARSE-SET(S)
2      f[0] ← 0
3      f[1] ← S[0]
4
5      i ← 2
6      for i ≤ n do
7          f[i] ← max(f[i − 1], f[i − 2] + S[i])
8
9      return f[n]
10
```

Proof of Correctness: We can inductively prove correctness of the algorithm.

[Basis]

- For $n = 1$, there is only one element so the weight of the heaviest sparse set is the weight of the element itself. Our algorithm gives the correct result since it defines $f[1] = w_1$.
- For $n = 2$, there are two elements, but they cannot be simultaneously be present in a sparse set. Thus the weight of the heaviest sparse set is $\max(w_1, w_2)$. Our algorithm will give $f[2] = \max(f[1], f[0] + w_2) = \max(w_1, w_2)$, which is correct.

[Inductive Step] Assume the algorithm is correct for $n = i - 1, i - 2$. By the optimal substructure property, we know the optimal value for $n = i$ is $\max(f[i-1], f[i-2] + w_i)$, which matches our algorithm.

Time and Space Complexity

Time complexity: $O(n)$, since we loop once through n, each iteration taking constant time.

Space complexity: $O(n)$, for storing f.

Since we don't need the entire array for computation, we may only store the last two elements of the array. Then we only need $O(1)$ additional space.

Demonstration

Consider $S = \{1, 5, 3, 3, 5, 7, 3\}$.

i	$S[i]$	$f[i]$	S'	$W(S')$	S^*	$W(S^*) = f[i]$
0		0	{}	0	{}	0
1	1	1	{1}	1	{1}	1
2	5	$f[1]$	{1}	1	{5}	5
		$f[0] + 5$	{5}	5		
3	3	$f[2]$	{5}	5	{5}	5
		$f[1] + 3$	{1, 3}	4		
4	3	$f[3]$	{5}	5	{5, 3}	8
		$f[2] + 3$	{5, 3}	8		
5	5	$f[4]$	{5, 3}	8	{5, 5}	10
		$f[3] + 5$	{5, 5}	10		
6	7	$f[5]$	{5, 5}	10	{5, 3, 7}	15
		$f[4] + 7$	{5, 3, 7}	15		
7	3	$f[6]$	{5, 3, 7}	15	{5, 3, 7}	15
		$f[5] + 3$	{5, 5, 3}	13		

Figure 1: Calculations for finding the weight of a heaviest sparse subset of the set $\{1, 5, 3, 3, 5, 7, 3\}$

Note that our algorithm does not keep track of the actual subsets, nor does it find it by backtracking. The algorithm works just on the weights of subsets. The sets themselves are shown here just for clarity.

3.6.2 **(Minimize recharge)** Mr. Singh drives his electric car from Srinagar to Kanyakumari along the highway NH-44. His car's battery, when fully charged, can travel k kilometers. The built-in map of his car indicates the distances between charging stations on his route. Mr. Singh started with a full charge from Srinagar and wishes to make as few charges as possible along the way.
Design an efficient algorithm to determine at which charging stations he should stop, and prove that your strategy yields an optimal solution. Justify its time complexity.

Sol. Let there be n charging stations along the route from Srinagar to Kanyakumari, numbered 1 to n. Thus there are $n+1$ segments of the path divided by the charging stations. Lets denote these as $a_0, a_1, ..., a_n$. The segment that comes after charging station i is a_i.

Consider the greedy strategy: Choose to charge at a charging station i if the next segment (a_i) cannot be covered by the current charge.

Algorithm 2: Minimize Recharge

```
1  procedure MINIMIZE-RECHARGE-GREEDY(A, k)
2      stations ← []
3      charge ← k
4
5      for i ← 0 to n do
6          if length(a_i) > k then
7              return ∞
8
9
10         if length(a_i) > charge then
11             append i to stations
12             charge ← k
13
14         charge ← charge − length(a_i)
15
16     return stations
17
```

Proof of Correctness

1. **Validity:** By definition, the greedy strategy is a valid one, since it simulates the charge of the car's battery at every station, and it ensures that at no point will the charge be ≤ 0.
2. **Optimality:**

Let us show that there must exist a optimal solution with the same choice as the greedy strategy for its first halt.

From the starting point (Srinagar), the greedy strategy will choose to halt at the farthest reachable station (given the current charge $= k$). Say this station is station r. If any solution chooses to stop and charge at some station $r' < r$, then pushing the halt to r does not increase the number of charges required. Thus an optimal solution must exist with the first choice the same as the greedy choice.

Now, note that after the first choice of halting, the car is fully charged, and the remaining problem is equivalent to the original problem, with a smaller input size. Thus we have an

optimal substructure, and by induction it follows that the greedy approach is optimal for the entire problem.

Time Complexity

We loop over every path segment and in each iteration we chose to charge or not charge in $O(1)$ time. Thus the overall time complexity of the algorithm is $O(n)$.

3.6.3 **(Min-max red-blue pairing)** There are n red points and n blue points placed along a horizontal line. Their positions are given in arrays $R[1..n]$ and $B[1..n]$ respectively. Suggest an algorithm to pair up each red point with exactly one blue point, such that the **maximum pair-gap** is minimized. Prove its correctness and write its time complexity. Note: The **pair-gap** between two points refers to the absolute difference of their positions.

Sol. **Observation**

Consider $r_1 < r_2$ and $b_1 < b_2$ to be 2 pairs of points. There exists two red-blue pairings for these points:

1. r_1-b_1 and r_2-b_2: maximum pair gap $= \max(|r_1 - b_1|, |r_2 - b_2|)$
2. r_1-b_2 and r_2-b_1: maximum pair gap $= \max(|r_1 - b_2|, |r_2 - b_1|)$

We claim that pairing 1 is always optimal, i.e. the maximum pair gap for pairing 1 is less than or equal to that of pairing 2.

We can prove the claim by considering each case for r_1, r_2, b_1, b_2:

- $r_1 < r_2 \leq b_1 < b_2$: $\max(|r_1 - b_1|, |r_2 - b_2|) \leq \max(|r_1 - b_2|, |r_2 - b_1|) = b_2 - r_1$
- $b_1 < b_2 \leq r_1 < r_2$: Equivalent to the previous case.
- $r_1 \leq b_1 < b_2 \leq r_2$: $|r_1 - b_1| < |r_1 - b_2|$ and $|r_2 - b_2| < |r_2 - b_1|$, thus $\max(|r_1 - b_1|, |r_2 - b_2|) \leq \max(|r_1 - b_2|, |r_2 - b_1|)$
- $b_1 \leq r_1 < r_2 \leq b_2$: Equivalent to previous case.
- $r_1 \leq b_1 \leq r_2 \leq b_2$: $\max(|r_1 - b_1|, |r_2 - b_2|) \leq \max(|r_1 - b_2|, |r_2 - b_1|) = b_2 - r_1$
- $b_1 \leq r_1 \leq b_2 \leq r_2$: Equivalent to previous case.

Thus we have shown that for two pairs of points, it is optimal to pair them in a 'non-crossing' manner, to minimize the maximum pair gap.

This leads to the following greedy algorithm: Sort both the arrays, and pair the ith red point with the ith blue point.

Algorithm 3: Min-Max Red-blue Pairing

```
1  procedure GREEDY-REDBLUE-PAIRING(R, B)
2      sort(R)
3      sort(B)
4      pairings ← []
5      for i ← 1 to n do
6          append into pairings (R_i, B_i)
7
8      return pairings
9
```

Proof of Correctness

Consider a pairing with some i where R_i is paired with B_j, $j \neq i$. WLOG assume that $j > i$. There must exist some $k > i$ such that R_k is paired with some B_m, $m < j$. (Pigeonhole principle)

We say, this pairing has a crossing pair of points. We can always get a better (or equivalent) pairing by uncrossing the pairs, i.e. by removing pairs $\left(R_i, B_j\right)$ and (R_k, B_m), and replacing them with the pairs (R_i, B_m) and $\left(R_k, B_j\right)$. We have shown this above.

Thus from any pairing if we repeatedly do the uncrossing process, we get a non-crossing pairing, finally reaching the pairing $P, (R_i, B_i) \in P \ \forall \ i$; which is what our greedy algorithm returns.

Time Complexity

Sorting dominates the complexity, $O(n \lg n)$.

3.6.4 **(Minimum operating cost)** Suppose you are running a lightweight consulting business. Your clients are distributed between the East Coast and the West Coast, and this leads to the following question.

Each month, you can either run your business from an office in New York (NY) or from an office in San Francisco (SF). In month i, you'll incur an operating cost of N_i if you run the business out of NY or an operating cost of S_i if you run it out of SF. However, if you run the business out of one city in month i, and then out of the other city in month $i+1$, then you incur a fixed moving cost of M to switch base offices.

Given a sequence of n months, a plan is a sequence of n locations—each one equal to either NY or SF—such that the ith location indicates the city in which you will be based in the ith month. The cost of a plan is the sum of the operating costs for each of the n months, plus a moving cost of M for each time you switch cities. The plan can begin in either city. Given a value for the moving cost M, and sequences of operating costs $N_1, \ldots, N_n$ and $S_1, \ldots, S_n$, find a plan of minimum cost.

(**a**) Show that the following algorithm does not solve this problem, by giving an instance on which it does not return the correct answer.

```
for i = 1 to n
  if Ni < Si then choose NY in Month i
  else choose SF in Month i
end for
```

In your example, say what the correct answer is and also what the algorithm above finds.

Sol. Consider the following instance: $n = 3$, $M = 100$, $N_i = [2, 5, 1]$, $S_i = [3, 2, 7]$.

The given algorithm will return the following plan:

$$[\text{NY}, \text{SF}, \text{NY}]$$

which has a cost of $2 + 2 + 1 + 100 + 100 = 205$.

An optimal plan is

$$[\text{NY}, \text{NY}, \text{NY}]$$

which has a cost of $2 + 5 + 1 = 8$.

The algorithm does not consider the moving cost; thus it does not always give the correct answer.

(**b**) Give an example of an instance in which every optimum plan must move (i.e. change locations) at least three times. Provide a brief explanation, saying why your example has this property.

Sol. Consider the following instance: $n = 4$, $M = 2$, $N_i = [1, 20, 1, 20]$, $S_i = [20, 1, 20, 1]$.

An optimal plan is

$$[\text{NY}, \text{SF}, \text{NY}, \text{SF}]$$

which has a cost of $1 + 1 + 1 + 1 + 2 + 2 + 2 = 10$.

This instance has the property that any optimal plan must move at least three times. This is because if a plan moves less than three times, we can show that the cost of that plan will be ≥ 20 as it must choose a month and a place whose operating cost is 20.

(c) Give an efficient algorithm that finds an optimal plan. Justify its correctness and runtime.

Sol. **Optimal Substructure**: Consider plan P_i to be an optimal plan for months 1 to i. Without loss of generality, assume that plan P_i operates at NY for month i. There are two cases:

- P_i operates at NY for month $i-1$. Then, $P_i[1..i-1]$ itself must be an optimal plan for $i-1$ months, ending with NY.
- P_i operates at SF for month $i-1$. Again, $P_i[1..i-1]$ must be an optimal plan for $i-1$ ending at SF.

This is because if the substructure is not optimal, if we replace it with an optimal substructure, we will get a better solution, which is a contradiction.

Similarly, the same applies even if P_i operates at SF for month i.

Overlapping Subproblems

$$f_N[i] = \min(f_N[i-1] + N_i, \quad f_S[i-1] + M + N_i)$$
$$f_S[i] = \min(f_S[i-1] + S_i, \quad f_N[i-1] + M + S_i)$$

This naturally leads to a DP solution:

Algorithm 4: Minimum Operating Cost

```
1  procedure MIN-OPERATING-COST(n, S, N, M)
2      f_S[1] ← S_1
3      f_N[1] ← N_1
4      for j ← 2 to n do
5          if f_S[j − 1] + S_j ≤ f_N[j − 1] + M + S_j then
6              f_S[j] ← f_S[j − 1] + S_j
7              trace_S[j] ← SF
8          else
9              f_S[j] ← f_N[j − 1] + M + S_j
10             trace_S[j] ← NY
11
12
13         if f_N[j − 1] + N_j ≤ f_S[j − 1] + M + N_j then
14             f_N[j] ← f_N[j − 1] + N_j
15             trace_N[j] ← NY
16         else
17             f_N[j] ← f_S[j − 1] + M + N_j
18             trace_S[j] ← SF
19
20
21     return min(f_S[n], f_N[n])
22
```

Algorithm 5: Backtracking to retrieve plan

```
1  procedure OPTIMAL-PLAN(n, f_N, f_S, trace_N, trace_S)
2      plan ← []
3      if f_N[n] < f_S[n] then
4          curr ← NY
5      else
6          curr ← SF
7
8      for j ← n to 1 do
9          prepend curr to plan
10         if j = 1 then
11             BREAK
12
13         if curr = NY then
14             curr ← trace_N[j]
15         else
16             curr ← trace_S[j]
17
18
19     return plan
20
```

Correctness: Inductive proof. For $n = 1$, the algorithm correctly gives $\min(N_1, S_1)$. Assuming it correctly finds the minimum cost for $n - 1$ months, by the optimal substructure property it must also get the minimum cost for n months.

Runtime:

- Algorithm 4: Time complexity: $O(n)$, Space complexity: $O(n)$
- Algorithm 5: Time complexity: $O(n)$, Space complexity: $O(n)$

3.6.5 (Optimal Strategy for a Game) There is a sequence of n coins with values $v_i > 0$ for $i = 1, 2, \ldots, n$. There are two players, Player 1 and Player 2. Each player chooses either the first coin or the last coin from the remaining coins alternately, starting with Player 1. Once a player chooses a coin, he removes it from the sequence of coins. The goal of Player 1 is to maximize his earning. The goal of Player 2 is to maximize his earning, or equivalently to minimize the earning of Player 1.
Give an efficient algorithm to determine the maximum possible earning of Player 1 irrespective of the strategy of Player 2. Justify its correctness, time and space complexities, and provide a demonstration for the input $4, 7, 3, 8, 2, 5$.

Sol. Interestingly, a greedy choice of picking the more valuable coin fails here. This is shown by this example: $7, 20, 1, 3$. Here, player 1 should choose 3 instead of 7, in order to guarantee to get 20 and win the game.

Since we cannot greedily make choices, we must find some other property of the problem.

Optimal Substructure: The optimal choices for player 1 on a sequence of n coins, must contain within it optimal choices for the resulting sequence of $n - 2$ coins (after the first move).

Overlapping subproblems: To compute the best choices for some state $[i..j]$, we need to compute for smaller subproblems $[i + 2..j]$, $[i + 1..j - 1]$, $[i..j - 2]$. These subproblems overlap, thus we can memoize the result by doing a bottom-up DP.

Formalising the DP

Let $f[i, j]$ = maximum possible earning of the first player given the state of coins i through j.

- If the first player chooses the first coin:
  - The other player may choose the second coin or the last coin.
- If the first player chooses the last coin:
  - The other player may choose the first coin or the second last coin.

We assume that the other player chooses his choice so as to minimise the first player's earnings.

Thus, we can define the recurrence

$$f[i, j] = \max\Big(v_i + \min(f[i + 2, j], f[i + 1, j - 1]), v_j + \min(f[i, j - 2], f[i + 1, j - 1])\Big)$$
$$f[i, i] = v_i$$
$$f[i, i + 1] = \max(v_i, v_{i+1})$$

Order of computation

We must go in order of increasing size $(j - i)$ of the range, since larger subproblems depend upon smaller subproblems.

We also observe that to compute $f[i, j]$, we only require subproblems of size $j - i - 2$. Since we finally need $f[1, n]$, if $n - 1$ is odd, we only need to compute for ranges with odd sizes, similarly if $n - 1$ is even, we only need to compute for ranges with even sizes.

Algorithm 6: Optimal strategy for a game

```
1  procedure MAX-EARNING(n, v)
2      parity ← (n − 1) mod 2
3      for size ← parity to (n − 1) step 2 do
4          for i ← 1 to n − size do
5              j ← i + size
6              if size = 0 then
7                  f[i, j] ← v_i
8              else if size = 1 then
9                  f[i, j] ← max(v_i, v_j)
10             else
11                 f[i, j] ← max(
                       v_i + min(f[i + 2, j], f[i + 1, j − 1]),
                       v_j + min(f[i, j − 2], f[i + 1, j − 1])
                   )
12
13
14
15     return f[1, n]
16
```

Correctness

For $n = 0, 1$ the algorithm gives the correct answer (trivial). By the optimal substructure property, we can construct the solution for n coins by finding the solution for $n - 2$ coins. By induction, this proves correctness.

Complexity

- Time Complexity: $O(n^2)$
- Space Complexity: $O(n^2)$

We only require optimal values for ranges of size $k - 2$ to compute optimal values for ranges of size k. So it is possible to optimize the storage to be $O(n)$.

Demonstration

Let the set of coins be $4, 7, 3, 8, 2, 5$. The algorithm is demonstrated in Figure 2 and Figure 3 below.

size	i	j	choice of coin	$f[i,j]$
1	1	2	$v_1 = 4$	7
			$v_2 = 7$	
	2	3	$v_2 = 7$	7
			$v_3 = 3$	
	3	4	$v_3 = 3$	8
			$v_4 = 8$	
	4	5	$v_4 = 8$	8
			$v_5 = 2$	
	5	6	$v_5 = 2$	5
			$v_6 = 5$	
3	1	4	$v_1 + \min(f[2,3], f[3,4]) = 4 + 7 = 11$	15
			$v_4 + \min(f[1,2], f[2,3]) = 8 + 7 = 15$	
	2	5	$v_2 + \min(f[3,4], f[4,5]) = 7 + 8 = 15$	15
			$v_5 + \min(f[2,3], f[3,4]) = 2 + 7 = 9$	
	3	6	$v_3 + \min(f[4,5], f[5,6]) = 3 + 5 = 8$	13
			$v_6 + \min(f[3,4], f[4,5]) = 5 + 8 = 13$	
5	1	6	$v_1 + \min(f[2,5], f[3,6]) = 4 + 13 = 17$	20
			$v_6 + \min(f[1,4], f[2,5]) = 5 + 15 = 20$	

Figure 2: Computational steps to fill up the DP Table

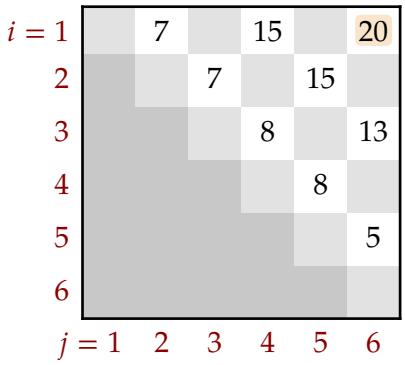

Figure 3: DP Table for finding max earning with $v = \{4, 7, 3, 8, 2, 5\}$. We only use cells where $i \leq j$ and $j - i$ is odd.