Winter 2024
Feb 26, 2024

Algorithm Design and Analysis: Midsem (Sec-A)

Max: 70 Marks
120 minutes

Roll No: _____ **Name:** _____

Please write solutions independent of each other. This is a closed book test. You can not use books or lecture notes. Please note that your solution must fit in the space provided. Extra sheet will be provided only *for roughwork . So try to be precise and brief. Meaningless blabber fetches negative credit.*

Part A

Question	1	2	3	Total
Marks				

Part B

Question	1	2	3	Total
Marks				

Total Marks:

Part A

1. **(5 Marks)** Suppose that your algorithm divides an input instance of size n into 2 different subproblems each of size $\lfloor n/2 \rfloor$ and 2 additional subproblems each of size $\lceil n/2 \rceil$. Then, recursively solves each of these subproblems and combines those solutions in $12n^2 + 16n$-time. If your input has size at most 4, then it solves your problem correctly in at most 20 steps. Then, what is the running time of your algorithm? Give a short explanation of your claimed running time.

Solution: The recurrence relation for the subproblem is $T(n) \leq \begin{cases} 4T(n/2) + (12n^2 + 16n) & \text{if } n \geq 5 \\ O(1) & \text{otherwise} \end{cases}$

(**3 Marks** for the recurrence. No marks deduction if the $\leq$ is written as $=$. Do not deduct marks if $12n^2 + 16n$ is replaced by $O(n^2)$.)

Using master's theorem, we have $T(n) = O(n^2 \log_2 n)$ (or $\Theta(n^2 \log_2 n)$).
(**2 Marks** for this. Use of Θ or BigOh has no issue with marking. No marks deduction if they omit the base of the logarithm. This is simple use of Master's theorem. If they write time complexity that is asymptotically worse than $O(n^2 \log n)$ or something that is asymptotically better than $O(n^2 \log n)$, then give 0 marks.)

2. **(10 Marks)** Recall that an algorithm to count the number of inversions in an array A of n distinct numbers that was discussed in the class. We used a subroutine **Count-Split-Inversion**(X, Y, n_x, n_y) that takes two sorted arrays X of n_x numbers and Y of n_y numbers as input and returns the number of split-inversions, i.e. $|\{(a, b) \mid a \in X, b \in Y\}|$ between X and Y in $O(n_x + n_y)$-time and outputs a sorted array Z of size $n_x + n_y$ that contains all the numbers of X and Y. We modify that algorithm and propose a different divide and conquer algorithm.

Count-Inversion(A, n)

(i) If$(n = 1)$ then (fill up here) **Return** $(0, A)$; (**1 Mark**)

(ii) If$(n = 2)$ then (fill up written below)

- **If** $(A[1] > A[2])$ **then**
- Swap(A[1], A[2]);
- return $(1, A)$;
- **Else return** $(0, A)$;

(**2 Marks**. There are precisely two instructions. Deduct 1 mark if some instruction is incorrect.)

(iii) If$(n \geq 3)$ then (fill up written below)

- $t_1 \leftarrow \lfloor n/3 \rfloor$ and $t_2 \leftarrow \lceil n/3 \rceil$;
- $A_L \leftarrow A[1, \ldots, t_1]$; $A_M \leftarrow A[t_1 + 1, \ldots, t_1 + t_2]$; $A_R \leftarrow A[t_1 + t_2 + 1, \ldots, n]$;
- $(d_L, A_L) \leftarrow$ Count-Inversion(A_L, t_1);
- $(d_M, A_M) \leftarrow$ Count-Inversion(A_M, t_2);
- $(d_R, A_R) \leftarrow$ Count-Inversion$(A_R, n - t_1 - t_2)$;
- (written below; some instruction directly invoke the subroutine **Count-Split-Inversion**. No need to describe it)

  - $(h_1, B) \leftarrow$**Count-Split-Inversion**(A_L, A_M, t_1, t_2);

  - $(h_2, A) \leftarrow$**Count-Split-Inversion**$(B, A_R, t_1 + t_2, n - (t_1 + t_2))$;

  - $total = d_L + d_M + d_R + h_1 + h_2$;

  - Return $(total, A)$;

(**4 Marks**. Deduct 2 marks if the some instructions are correct but the overall instructions do not lead to what it is supposed to. The instructions should lead to the same as this set of instructions achieve.)

(iv) Since the algorithm above is recursive, what is the the running time of the above algorithm? Explain with recurrence relation. Just write down the recurrence relation and the running time.

Solution: Since all $|A_L|, |A_M|, |A_R| \leq \lceil n/3 \rceil$, the recurrence relation is

$$T(n) = \begin{cases} 3T(n/3) + \Theta(n) & \text{if } n \geq 3 \\ \leq \Theta(1) & \text{otherwise} \end{cases}$$

Using master's theorem, $T(n) = O(n \log_2 n)$.

(**2 Marks** for recurrence and **1 Mark** for the master's theorem use. The explanation is just used for justification and not for marking.)

3. **(10 Marks)** Recall the algorithm to identify the k-th smallest element from an array A of n distinct numbers that was done in the class. Suppose that the algorithm is modified and designed using the following approach.

Function $\mathsf{Select}(A, n, k)$ where $n = |A|$ and $k \leq n$.

(i) If $n < 10$, then use brute-force approach and find the k-th smallest element.

(ii) Divide A into $\lfloor n/7 \rfloor$ groups of 7 elements each. Additional elements are placed in their own group of size r such that r is the remainder after you divide n by 7.

(iii) Find median of each of the groups (sort the elements of each group and pick the median). If there are r elements in a group, them median is the $\lfloor r/2 \rfloor$-th smallest element in that group.

(iv) $B \leftarrow$ the array containing the medians found in Step (ii).

(v) Recursively compute $pivot \leftarrow \mathsf{Select}(B, |B|, \lfloor |B|/2 \rfloor)$.

(vi) Compute the array A_L that contains all elements that are smaller than *pivot*.

(vii) Compute the array A_R that contains all elements that are larger than *pivot*.

(viii) If $|A_L| = k - 1$, then (fill up here) **Return $A[k]$ (or return *pivot*)**
(**1 Mark**)

(ix) If $|A_L| < k - 1$, then (fill up written below)
- $r = |A_L|$;
- $a \leftarrow \mathsf{Select}(A_R, n - r - 1, k - r - 1)$;
- Return a;

(since $|A_L| = r < k$, there are $n - r - 1$ elements in A_R. The recursive calls have to find out $(k - r - 1)$-th smallest number from A_R)
(**2 Marks**. Check the instructions carefully. Deduct 1 mark if some instructions are correct while some instructions are incorrect.)

(x) If $|A_L| \geq k$, then (fill up written below)
- $r = |A_L|$;
- $a \leftarrow \mathsf{Select}(A_L, r, k)$;
- Return a;

(since A_L has r elements and $r > k$, the recursive call has to find out k-th smallest element from A_L)
(**2 Marks**. Check the instructions carefully. Deduct 1 mark if some instructions are correct while some instructions are incorrect.)

(a) **What is the recurrence relation of the recursive algorithm described above?**
Note that in the array B, there are at least $n/14$ elements that are smaller than *pivot*. Additionally, there are three elements in $n/14$ different subarrays, that are smaller than *pivot*. Hence, there are at least $4n/14 = 2n/7$ elements that are smaller than the *pivot*. Hence, $\max(|A_L|, |A_R|) \leq 5n/7$. Hence, the recurrence is

$$T(n) \leq \begin{cases} T(n/7) + T(5n/7) + O(n) & \text{if } n \geq 10 \\ c & \text{otherwise} \end{cases}$$

(**2 Mark**. The first part of the recurrence is the most important. The explanation is for understanding the answer and not for marking. If somebody writes $O(1)$ or some specific constant instead of c, then do not deduct any marks.)

(b) **Explain the running time of your algorithm**.
Using substitution method, the solution can be obtained to $T(n) = O(n)$.
(**3 Marks**. Deduct 1 mark if not mentioned substitution medhod or recursion tree method. Just mentioning which method he/she uses is sufficient. Detailed explanations are not required.)

Part B

1. (10 **Marks**) You are given two sorted arrays $A[]$ and $B[]$ of positive integers. The sizes of the arrays are not given. Accessing any index beyond the last element of the arrays returns -1. The elements in each arrays are distinct but the two arrays may have common elements. An intersection point between two arrays is an element that is common to both, i.e. p be an intersection point if there is i and j such that $A[i] = B[j] = p$. Given an integer x, design an algorithm (in pseudocode) to check if x is an intersection point of A and B.

   (a) **Pseudocode of your algorithm. The instructions of your pseudocode can be in plain text if required**

Algorithm 1: Finding the Length an array $X[]$

```
1  Function Length(X)
2  Initialize index ← 1;
3  if X[i] = −1 then
4      Return 0;
5  while do
6      if X[index] > 0 and X[2 ∗ index] > 0 then
7          index ← 2 ∗ index;
8      else
9          low ← index;
10         high ← 2 ∗ index;
11         Break;
12 while low < high do
13     mid ← ⌊(low + high)/2⌋;
14     if X[mid] = −1 then
15         high ← mid − 1;
16     else
17         low ← mid + 1;
18     if X[high] > 0 then
19         Return high;
20     if X[low] = −1 then
21         Return (low − 1);
```

This completes the description of the algorithm to compute the number of elements. Our main algorithm uses the above as subroutine as follows.

Algorithm 2: Main algorithm.

```
1  Function Main-Algo(A, B, x)
2  L_A ← Length(A);
3  L_B ← Length(B);
4  (the above two instructions find the number of elements in both the arrays)
5  i ← BinarySearch(A, L_A, x);
6  j ← BinarySearch(B, L_B, x);
7  (the above two instructions find the position of x in A and B if exists)
8  if i = null or j = null then
9      Return No;
10 Return Yes;
```

- Algorithm-1 has **3 marks** and the Algorithm-2 has **3 Marks**.
- Give marks 0 if the algorithm uses $A.size()$ or some such similar subroutine irrespective of the written answer in (b). If the overall running time takes $O(\max(|A|, |B|))$-time (or asymptotically worse running time), then give 0 marks irrespective of the answer in (b).
- If the Algorithm-1 is not at all described and just the Main-Algorithm is described without any justification, then give only 1 marks out of 6 marks. No more than that.

(b) **Give an explanation of the running time of your algorithm**.

First the Main Algorithm invokes the Algorithm-1 for the arrays A and B.

- Algorithm-1 first identifies two indices k and $2k$ such that $A[k] > 0$ but $A[2k] = -1$ such that $k = 2^i$. This procedure takes $O(i)$ operations to find out k and $2k$.
- Since $k = 2^i$ and $2k = 2^{i+1}$, observe that the number of elements in A is at most $2k$.
- This procedure takes $O(\log |A|)$-time.
- Similarly, the same procedure is used to compute the size of B. Hence, that procedure takes $O(\log |B|)$-time.
- Subsequently algorithm checks if x is an element in A as well as an element in B using $O(\log |A|)$ steps and $O(\log |B|)$ steps respectively.
- Hence, the running time of the algorithm is $O(\log |A| + \log |B|)$.

Hence, identifying $|A|$ and $|B|$ take $O(\log |A| + \log |B|)$-time.

(**2 Marks** for this explanation - first 4 bullet points in summary.)

After that the algorithm checks if x belongs to the array A and x belongs to the array B. This takes additional $O(\log |A| + \log |B|)$-time. Hence, the running time of the algorithm is sublinear in $|A| + |B|$.

(**2 Marks** for this explanation. This consists of the last two bullet points.)

2. **(15 Marks)** Consider an ordered sequence of balls $b_1, b_2, \ldots, b_n$ and each ball is assigned a positive weight $\mathsf{wt}(b_i)$. A subset of balls S is said to be a *competent* set if for every $i \in \{2, \ldots, n-1\}$ either $b_i \in S$ or $b_{i-1} \in S$ or $b_{i+1} \in S$. Similarly, for the ball b_1 either $b_1 \in S$ or $b_2 \in S$. Moreover, for the ball b_n either $b_n \in S$ or $b_{n-1} \in S$. The *weight* of a set S of balls is defined as $\sum_{b_i \in S} \mathsf{wt}(b_i)$. A competent set S is *optimal* if the weight of S is the smallest among all possible competent set of balls. Design an algorithm that runs in time polynomial in n and computes the weight of a competent set of balls that is optimal.

(a) A brief description of your algorithm (DP is recommended)

Since it is recommended to use DYNAMIC PROGRAMMING paradigm, we describe the solution using this paradigm. For a non-DP solution, the marking has to be carried out rationally.

Subproblem: For every $k \in \{0, 1, \ldots, n\}$, we denote the following two subproblems.

- $\mathsf{MCS}[k, 0]$ the weight of an optimal competent subset $S \subseteq \{b_1, \ldots, b_k\}$ that does not contain b_k.
- $\mathsf{MCS}[k, 1]$ the weight of an optimal competent subset $S \subseteq \{b_1, \ldots, b_k\}$ that contains b_k.
- $\mathsf{DS}[k] = \min(\mathsf{MCS}[k, 0], \mathsf{MCS}[k, 1])$, i.e. the size of an optimal competent set $S \subseteq \{b_1, \ldots, b_k\}$ of balls.

(**Total 3 marks**)

Recurrence of Subproblem:

- **Base Cases:** $\mathsf{MCS}[1, 1] = \mathsf{wt}(b_1)$ and $\mathsf{MCS}[1, 0] = \infty$ (because no such solution exist, we use ∞ to denote the malformed subproblem definition)
  Hence, $\mathsf{DS}[1] = \mathsf{wt}(b_1)$.

  Additionally, $\mathsf{MCS}[0, 0] = \mathsf{MCS}[0, 1] = \mathsf{DS}[0] = 0$.
- For every $k \geq 2$, the followings hold true.
  $\mathsf{MCS}[k, 0] = \mathsf{MCS}[k-1, 1]$.
  $\mathsf{MCS}[k, 1] = \mathsf{wt}(b_k) + \min(\mathsf{DS}[k-1], \mathsf{DS}[k-2])$

(**5 Marks**)

Subproblem that solves the actual problem: $\mathsf{DS}[n]$.

(**1 Marks**)

Algorithm Description:

We store the subproblem values MCS into an array X and the subproblem values in an array Y.

- Initialize $X[0][0] \leftarrow X[0][1] \leftarrow Y[0] \leftarrow 0$.
- Initialize $X[1][0] \leftarrow \infty$, $X[1][1] \leftarrow \mathsf{wt}(b_1)$ and $Y[1] \leftarrow X[1][1]$.
- For $k = 2, \ldots, n$ in this order, fill-up the rest of the table as follows.
  - Set $X[k][0] \leftarrow X[k-1][1]$;
  - If $Y[k-1] > Y[k-2]$, then set $X[k][1] \leftarrow \mathsf{wt}(b_k) + Y[k-2]$.
    Otherwise set $X[k][1] \leftarrow \mathsf{wt}(b_k) + Y[k-1]$.
  - Finally, set $Y[k] \leftarrow \min(X[k][0], X[k][1])$.
- Return $Y[n]$.

(**4 Marks**)

(b) **Explanation of the running time of your algorithm**:

Observe that the algorithm fills up two different arrays, one of dimension $n \times 2$ and the other of dimension $n \times 1$. Computing each table entry takes $O(1)$-time. Hence, the running time of the algorithm is $O(n)$.

(**2 Marks**)

- For a non-DP solution, if it uses Divide and Conquer, then it has to be rationally checked if there is any mistake. In such a case, algorithm description goes 9 Marks, and running time explanation has 6 marks including recurrence.
- If it uses a greedy approach, then 10 marks for algorithm description and explanation of correctness. Check carefully why a particular greedy solution is wrong. For a wrong greedy solution, no more than 2 marks should be given for algorithm part (out of 10). The running time explanation has 5 marks.

3. **(20 Marks)** Suppose that you are working on a consulting business, just you, two associates and a rented equipment. Your clients are distributed in two parts of the country, north part and south part. In each month, you either run your business from an office in Delhi (DEL) or from an office in Bangalore (BLR). In the i-th month, if you run your business from DEL, then you will incur an operating cost of d_i; if you run your business from BLR, then you will incur an operating cost of r_i. However, if you run the business in one city in the i-th month and move it to another city in the $(i+1)$-th month, then you incur a fixed moving cost M to switch the base offices. Given a sequence of n months, a *plan* is a sequence of n locations - each one either to DEL or BLR - such that the i-th location indicates a city in which you will be based in the i-th month. The cost of the plan is the sum of the operating cost in each month, plus a moving cost each time you switch the city. The plan can begin in any of the two cities. A plan is *optimal* if it has minimum cost among all possible plans. Design a dynamic programming based algorithm that runs in time polynomial in n and outputs the cost of an optimal plan.

Example: Suppose that $n = 4, M = 10$ and the operating costs are given by the following example. The values are $d_1 = 1, d_2 = 3, d_3 = 20, d_4 = 30, r_1 = 50, r_2 = 20, r_3 = 2$ and $r_4 = 2$. The plan of a minimum cost would be the sequence of locations (DEL, DEL, BLR, BLR) with total cost $d_1 + d_2 + M + r_3 + r_4 = 18$. A different plan (DEL, BLR, BLR, DEL) has cost $d_1 + M + r_2 + r_3 + M + d_4 = 73$ since this plan requires moving between city two times.

(i) **Definition of your subproblem:**

For every $k \in \{1, \ldots, n\}$, we define the following two subproblems.

- $\mathsf{Cost}[k, 1]$ denotes the cost of an optimal plan for the months $\{1, 2, \ldots, k\}$ such that the k-th month is operated in DEL.
- $\mathsf{Cost}[k, 2]$ denotes the cost of an optimal plan for the months $\{1, 2, \ldots, k\}$ such that the k-th month is operated from BLR.

(**4 Marks**)

(ii) **Recurrence of the subproblem:**

- **Base Case:** $\mathsf{Cost}[k, 1] = d_1$ and $\mathsf{Cost}[k, 2] = r_1$.
- For every $k \geq 2$, the following is the recurrence

$$\mathsf{Cost}[k, 1] = d_k + \min \begin{cases} \mathsf{Cost}[k-1, 1] \\ M + \mathsf{Cost}[k-1, 2] \end{cases}$$

$$\mathsf{Cost}[k, 2] = r_k + \min \begin{cases} \mathsf{Cost}[k-1, 2] \\ M + \mathsf{Cost}[k-1, 1] \end{cases}$$

(**7 Marks** out of which for the base case, **2 marks** and for the other part of recurrence, **5 marks**)

(iii) **The subproblem that solves the final problem:** $\min(\mathsf{Cost}[n, 1], \mathsf{Cost}[n, 2])$.

(**2 Marks**)

(iv) **Algorithm Description:**

The algorithm works as follows. A two dimensional array B of size $n \times 2$ stores the values.

- Initialize $B[1][1] \leftarrow d_1$ and $B[1][2] \leftarrow r_1$.
- For every $k = 2, \ldots, n$ in this order

  - Set $B[k][1] \leftarrow d_k + \min(B[k][1], M + B[k][2])$.
  - Set $B[k][2] \leftarrow r_k + \min(B[k][2], M + B[k][1])$.
- Return $\min(B[k][1], B[k][2])$.

(**5 Marks**)

(v) **Explanation of the running time:**

Observe that the algorithm fills up a $2D$-array of size $n \times 2$. Computing every array entry takes $O(1)$-time. Hence, the running time of the algorithm is $O(n)$.

(**2 Marks**)