Winter 2024 **Algorithm Design and Analysis: Midsem (Sec-B)** **Max: 70 Marks**
Feb 26, 2024 **120 minutes**

Roll No: _____ **Name:** _____

Question	1	2	3	4	5	6	Total
Marks							

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

All The Best!

---

1. **(3 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 subproblem and combines those solutions in $12n^2 + 16n$ time. If your input has size at most 4, then it solves your problem correctly using some brute-force approach with at most 20 steps (or time). 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)** Let $f(n)$ and $g(n)$ are two asymptotically positive functions. Prove or disprove the following claims.

   a. $f(n) + g(n) = \theta(\min\{f(n), g(n)\})$.

   Solution: If $f(n) + g(n) = \theta(\min\{f(n), g(n)\})$, then there is a positive integer n_0, two positive constants c_1 and c_2 such that for every integer $n \geq n_0$ we have the following relation

   $$c_1 \cdot (\min\{f(n), g(n)\}) \leq f(n) + g(n) \leq c_2 \cdot (\min\{f(n), g(n)\}).$$

   Alternatively, with n_0 and c_1, we have $f(n) + g(n) = \Omega(\min\{f(n), g(n)\})$ and with n_0 and c_2 we have $f(n) + g(n) = O(\min\{f(n), g(n)\})$.

   We can disprove the above claim with an example. Let $f(n) = 2n$ and $g(n) = 2^n$ then for $n \geq 3$ we get $\min\{f(n), g(n)\} = f(n)$, i.e., $\min\{2n, , 2^n\} = 2n$.

   For $n_0 = 4$ and $c_1 = 1$, we get $2n + 2^n = \Omega(\min\{2n, , 2^n\}) = \Omega(n)$. However, there is no integer n_0 and a constant c_2 such that $2n + 2^n = O(\min\{2n, , 2^n\}) = O(n)$, since 2^n grows exponentially compared to linear growth of $2n$. So $f(n) + g(n) \neq \theta(\min\{f(n), g(n)\})$.

   (**1 Marks** for the relation between θ and (Ω, O) or using the terms such as n_0, c_1 and c_2. No marks deduction if the $\geq$ is written as $>$. Do not deduct marks if this relation is stated in the next part of the question.)

   (**2 Marks** for showing $c_1 \cdot (\min\{f(n), g(n)\}) \leq f(n) + g(n)$ for every $n \geq n_0$ and a constant c_1. Or for showing $f(n) + g(n) = \Omega(\min\{f(n), g(n)\})$.)

   (**2 Marks** for showing $f(n) + g(n) \nleq c_2 \cdot (\min\{f(n), g(n)\})$ for every $n \geq n_0$ and a constant c_2. Or for showing $f(n) + g(n) \neq O(\min\{f(n), g(n)\})$.)

   b. $f(n) + g(n) = \theta(\max\{f(n), g(n)\})$.

   Solution: Similar as above, there is a positive integer n_0, two positive constants c_1 and c_2 such that for every integer $n \geq n_0$ we have the following relation $c_1 \cdot (\max\{f(n), g(n)\}) \leq f(n) + g(n) \leq c_2 \cdot (\max\{f(n), g(n)\})$. Alternatively, with n_0 and c_1, we have $f(n) + g(n) = \Omega(\max\{f(n), g(n)\})$ and with n_0 and c_2 we have $f(n) + g(n) = O(\max\{f(n), g(n)\})$.

   Without loss of generality assume $f(n) < g(n)$ for every n greater than some n_0, then we get $\max\{f(n), g(n)\} = g(n)$. So, for every $n \geq n_0$.

   $$\max\{f(n), g(n)\} \leq f(n) + g(n) \leq 2 \max\{f(n), g(n)\}$$

   (**1 Marks** for the relation between θ and (Ω, O) or using the terms such as n_0, c_1 and c_2. No marks deduction if the $\geq$ is written as $>$. Do not deduct marks if this relation is stated in the previous part of the question.)

   (**4 Marks** if the arguments are correct along with properly identifying the constants. **Deduct 1 mark** if constants are not identified.)

3. **(7 points)** Recall the algorithm to identify the k-th smallest element from an array A of n numbers done in the class. Fill in the blanks [(viii), (ix) & (x)] to design the following modified version of the algorithm.

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 $pivot$)** (**0.5 Mark**)

(ix) If $|A_L| < k - 1$, then (fill up written below)

- $r = |A_L|$;
- $a \leftarrow \mathsf{Select}(A_R, n - r - 1, k - r - 1)$;

(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)

(**1.5 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)

- $r = |A_L|$;
- $a \leftarrow \mathsf{Select}(A_L, r, k)$;

(since A_L has r elements and $r > k$, the recursive call has to find out k-th smallest element from A_L)

(**1 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.)

4. **(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 array 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.

   You will be awarded zero

   - if your algorithm runs in time linear in $|A|$ and $|B|$ and
   - if you use any direct library functions that returns the size of A or B.

   (a) **Write a pseudocode of your algorithm in plain-text or a brief description of your alg.**

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) **What is the recurrence of your algorithm? Compute the time complexity of your alg. Give a proper explanation of your answer**.

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.)

5. **(15 Marks)** Consider an ordered sequence of balls $b_0, b_1, b_2, \ldots, b_n, b_{n+1}$ and each ball is assigned a positive weight $\mathsf{wt}(b_i)$. Let $wt(b_0) = wt(b_{n+1}) = +\infty$. A subset of balls S is said to be a *competent* set if for every $i \in \{1, 2, \ldots, n\}$ either $b_i \in S$ or $b_{i-1} \in S$ or $b_{i+1} \in S$. A competent set S is of *minimum weight* if for every competent set S', $\sum_{b_i \in S} \mathsf{wt}(b_i) < \sum_{b_i \in S'} \mathsf{wt}(b_i)$. Design a polynomial-time algorithm to compute a minimum weight competent set of balls.

(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.

6. **(25 points)** Let A and B be two given words. Recall that "Edit Distance" computes the operational cost of transforming words A to B. Let the operations and their corresponding costs be as follows.

- If the i^{th} character of A is equal to the j^{th} character of B then, **do nothing** and transform the first $i-1$ characters of A into first $j-1$ characters of B. The operational cost of "do nothing" is 0.
- If the i^{th} character of A is not equal to the j^{th} character of B then,
  - **delete** the i^{th} character of A. Then, transform the first $i-1$ characters of A into the first j character of B. The operational cost of "delete" is 1.
  - **insert** the j^{th} character of B in between i^{th} and $(i+1)^{th}$ character of A. Then, transform the first i characters of A into the first $j-1$ character of B. The operational cost of "insert" is two times the cost of "delete".
  - **replace** the i^{th} character of A with the j^{th} character of B. Then, transform the first $i-1$ characters of A into the first $j-1$ character of B. The operational cost of "replace" is three times the cost of "delete".

Design an algorithm using dynamic programming that efficiently computes the minimum edit distance for transforming A into B.

(i) Definition of your subproblem: Given the words A and B, for every $i \in \{0, 1, \ldots, |A|\}$ and $j \in \{0, 1, \ldots, |B|\}$ we define the following function

- $\text{Cost}[A, B, i, j]$: It computes the cost of transforming the first i characters of A into the first j characters of B.

(**3 Marks** for subproblem definition. Do not deduct marks if A and B are not mentioned in $\text{Cost}(\cdot)$ and if $|A|$ and $|B|$ are assumed to be some values say m and n respectively.)

(ii) Recurrence of the subproblem and the subproblem that solves the final problem:

- **Base Cases:**
  - $\text{Cost}[A, B, i, 0] = i$ for every $i \in \{0, 1, \ldots, |A|\}$. These are the base case costs needed to transform the first i characters of A into 0^{th} character of B (i.e., an empty string). In this case, we delete the i characters in A to transform it into 0^{th} character of B.
  - $\text{Cost}[A, B, 0, j] = 2 \cdot j$ for every $j \in \{0, 1, \ldots, |B|\}$. These are the base case costs needed to transform the 0^{th} characters of A into the first j characters of B. In this case, we insert j characters to transform the 0^{th} character of A into the first j characters of B.

- General Cases:

$$\text{Cost}(A,B,i,j) = \begin{cases} \text{Cost}(A,B,i-1,j-1) & \text{if } A[i] == B[j] \\ \min \begin{cases} 1+\text{Cost}(A,B,i-1,j), \\ 2+\text{Cost}(A,B,i,j-1), \\ 3+\text{Cost}(A,B,i-1,j-1) \end{cases} & \text{else} \end{cases}$$

- Final Solution: $\text{Cost}(A,B,|A|,|B|)$

(**2 Marks** for the base cases. Again, do not deduct marks if A and B are not mentioned in $\text{Cost}(\cdot)$ and if $|A|$ and $|B|$ are assumed to be some values say m and n respectively.)

(**4 Marks** for the general cases. Here the important part is to realize that when the character matches, i.e., $A[i]$ same as $B[j]$ then the algorithm does nothing and computes diagonally above term $\text{Cost}(A,B,i-1,j-1)$. When, $A[i] \neq B[j]$, then there are three choices: delete (it costs 1), insert (costs 2) and replace (costs 3). It performs the algorithm that is least expensive along with its appropriate subproblem. One might have used a different indexing, as A are in rows and B are in columns.)

(**1 Mark** for defining the subproblem that solves the actual problem..)

(iii) **Algorithm Description:** We store the subproblem values Cost into a 2-d array M and the operations in D.

Algorithm 3: Finding final cost

```
Edit-Distance(A, B)
Initialize M = {0}^{|B|×|A|}
Initialize D = {∅}^{|B|×|A|}
for i = 0, 1, ..., |A| do
    Initialize M[0][i] = i
    Initialize D[0][i] = " ← "                    /*Delete operation
for j = 0, 1, ..., |B| do
    Initialize M[j][0] = 2 · j
    Initialize D[j][0] = " ↑ "                    /*Insert operation
for i = 1, ..., |A| do
    for i = 1, ..., |B| do
        if A[i] == B[j] then
            M[i][j] = M[i − 1][j − 1]
            D[i][j] = " ↖ "                       /*Do-nothing or Replace without any additional cost operation
        else
            delete = M[i − 1][j] + 1;
            insert = M[i][j − 1] + 2;
            replace = M[i − 1][j − 1] + 3;
            M[i][j] = min{delete, insert, replace}
            D[i][j] = " ← ", if delete smallest, or " ↑ ", if insert smallest, or " ↖ ", if replace smallest.
return M[|B|, |A|]
```

(**5 Marks** for the algorithm. If a student has written a memoization-based algorithm, then check for factors such as whether the very first recursive call solves the actual problem if the recursive function calls are based on the right condition. You may take small examples to verify, such as $A =$ "*he*" and $B =$ "*she*". If students have used additional cost for the keyword 'transformation', then check if their line of arguments is correct. Deduct marks only if the arguments are incorrect.)

(iv) **Explanation of the running time:** The above algorithm follows table filling method. The size of the table is $|A| \cdot |B|$ and to fill every cell of the table the algorithm takes constant time (O(1)). So the total running time of the algorithm is $O(|A| \cdot |B|)$. (**2 Marks** for the general cases.)

(v) **How much is the edit distance for $A =$ KITTEN and $B =$ SITTING? Where and what are the operations your algorithm made?**

Table 1: Table M

	∅	k	i	t	t	e	n
∅	0	1	2	3	4	5	6
s	2	3	4	5	6	7	8
i	4	5	3	4	5	6	7
t	6	7	5	3	4	5	6
t	8	9	7	5	3	4	5
i	10	11	9	7	5	6	7
n	12	13	11	9	7	8	6
g	14	15	13	11	9	10	8

The total cost is at $M[|B|][|A|] = 8$.

Table 2: Table D

	∅	k	i	t	t	e	n
∅	" "	←	←	←	←	←	←
s	↑	←	←	←	←	←	←
i	↑	←	↖	←	←	←	←
t	↑	←	↑	↖	↖	←	←
t	↑	←	↑	↖	↖	←	←
i	↑	←	↖	↑	↑	←	←
n	↑	←	↑	↑	↑	←	↖
g	↑	←	↑	↑	↑	←	↑

(**4 Marks**)

By following the table D from $D[|B|][|A|]$ to $D[0][0]$ the algorithm performs the following operations.

1. At $D[7][6]$ *insert* "g". Hence it results to "kitteng". It costs 2 and goes to $D[6][6]$.
2. At $D[6][6]$ *do-nothing*. Hence, we have "kitteng". It costs 0 and goes to $D[5][5]$.
3. At $D[5][5]$ *delete* "e". Hence it results to "kittng". It costs 1 and goes to $D[5][4]$.
4. At $D[5][4]$ *insert* "i". Hence it results to "kitting". It costs 2 and goes to $D[4][4]$.
5. At $D[4][4]$ *do-nothing*. Hence, we have "kitting". It costs 0 and goes to $D[3][3]$.
6. At $D[3][3]$ *do-nothing*. Hence, we have "kitting". It costs 0 and goes to $D[2][2]$.
7. At $D[2][2]$ *do-nothing*. Hence, we have "kitting". It costs 0 and goes to $D[1][1]$.
8. At $D[1][1]$ *delete* "k". Hence, it results to "itting". It costs 1 and goes to $D[1][0]$.
9. At $D[1][0]$ *insert* "s". Hence, it results to "sitting". It costs 2 and reaches to $D[0][0]$.

(**4 Marks**)