Theory Assignment 3: CS 452/652/752 Advanced Algorithms and Applications

I, Baba Avinash Puppala (bpuppala), declare that I have completed this assignment completely and entirely on my own, without any consultation with others. I understand that any breach of the UAB Academic Honor Code may result in severe penalties.

Solution to Question 1

We will solve this problem by creating a **memoization table** to store the number of distinct paths to reach each cell from the start position. The approach involves filling each cell by summing paths from cells that can reach it: the cell above (if moving down) and the cell to the left (if moving right).

Step-by-Step Solution

1. **Initialize the Grid:** Create a table representing the grid layout where ‘1‘ indicates an open cell and ‘0‘ indicates a blocked cell.

2. **Dynamic Programming Table Setup:** Create a memoization table with the same dimensions as the grid, initialized to zeros. Set the start cell (top-left) to ‘1‘ because there’s only one way to be at the start.

3. **Fill the Table:** - For each cell in the table, if it is not blocked: - Add the value from the cell above (if within bounds). - Add the value from the cell to the left (if within bounds). - This fills the table with the number of distinct paths to each cell from the start.

4. **Result:** The value in the bottom-right cell represents the total number of distinct paths from the start to the end position.

Memoization Table

The following table shows the computed number of paths for each cell in the grid:

1	1	0	1	2	3	3	6
1	0	1	2	0	3	0	9
1	1	2	4	4	7	7	16
1	2	0	4	0	7	0	23
1	3	3	7	7	14	14	37
1	0	3	0	7	0	14	51
1	1	4	4	11	11	0	62
0	1	0	5	16	27	27	**31**

Answer

The total number of distinct paths from the start to the end position, avoiding the blocked cells, is:

$$\text{Total Paths} = 31$$

Solution to Question 2

Dynamic Programming Approach

To solve this problem, we use dynamic programming. We define:

- $m[i][j]$: Minimum cost of multiplying matrices from M_i to M_j.
- $s[i][j]$: Index k where the optimal split occurs for matrices from M_i to M_j.

The recursive formula is:

$$m[i][j] = \min_{i \leq k < j} \{m[i][k] + m[k+1][j] + p_{i-1} \times p_k \times p_j\}$$

where p_i represents the dimension of matrix M_i.

Memoization Tables

Below are the tables showing the minimum cost and the multiplication order.

Cost Table $m[i][j]$

	M_1	M_2	M_3	M_4	M_5	M_6	M_7	M_8
M_1	0	8000	40000	120000	130000	150000	186000	290000
M_2		0	32000	80000	90000	130000	150000	210000
M_3			0	80000	130000	150000	186000	246000
M_4				0	400000	210000	330000	510000
M_5					0	200000	130000	300000
M_6						0	160000	260000
M_7							0	320000
M_8								0

Order Table $s[i][j]$

	M_1	M_2	M_3	M_4	M_5	M_6	M_7
M_1	0	1	1	1	3	3	5
M_2		0	2	3	3	5	5
M_3			0	3	3	5	5
M_4				0	4	5	5
M_5					0	6	7
M_6						0	7
M_7							0

Optimal Order and Minimum Cost

Using the order table, the optimal parenthesized order of multiplication is:

$$((M_1 \times M_2) \times ((M_3 \times M_4) \times ((M_5 \times M_6) \times (M_7 \times M_8))))$$

The minimum cost of multiplications is:

$$\text{Minimum Cost} = 290000$$

Problem Statement

Given 10 items with the following weight and profit values and a knapsack with weight capacity $K = 10$, find the set of items that maximizes the total profit. The weight and profit values are:

$$\text{Weight} = [1, 4, 2, 3, 2, 5, 2, 6, 1, 6]$$
$$\text{Profit} = [20, 50, 30, 30, 40, 20, 10, 40, 10, 30]$$

Solution to Question 3

Dynamic Programming Approach

To solve this problem, we use dynamic programming. Define:

- $dp[i][w]$: Maximum profit achievable with the first i items and weight capacity w.
- K: Total weight capacity of the knapsack, which is 10 in this case.

Recursive Formula

For each item i, we have two choices: 1. **Exclude the item** i: Profit remains as $dp[i-1][w]$. 2. **Include the item** i (if weight weight$[i-1] \leq w$): Profit becomes profit$[i-1] + dp[i-1][w - \text{weight}[i-1]]$.

Thus, the formula is:

$$dp[i][w] = \max(dp[i-1][w], \text{profit}[i-1] + dp[i-1][w - \text{weight}[i-1]])$$

Memoization Tables

Maximum Profit Table $dp[i][w]$

Below is the table showing the maximum profit for each item and weight capacity:

Items \ Capacity	0	1	2	3	4	5	6	7	8	9	10
0	0	0	0	0	0	0	0	0	0	0	
1	0	20	20	20	20	20	20	20	20	20	
2	0	20	20	20	50	70	70	70	70	70	
3	0	20	30	50	50	70	80	100	100	100	
4	0	20	30	50	50	70	80	100	100	100	
5	0	20	40	60	70	80	110	120	140	140	
6	0	20	40	60	70	80	110	120	140	150	
7	0	20	40	60	70	80	110	120	140	150	
8	0	20	40	60	70	80	110	120	140	150	
9	0	20	40	60	70	80	110	120	140	150	
10	0	20	40	60	70	80	110	120	140	150	

The maximum profit achievable is **150**.

Selected Items Table

To find the items selected, trace back from $dp[10][10]$: 1. If $dp[i][w] \neq dp[i-1][w]$, include item i and update $w = w - \text{weight}[i-1]$. 2. Continue this until $w = 0$.

The selected items are:

$$\text{Items} = \{1, 2, 3, 5, 9\}$$

Answer

The maximum profit is:

$$\text{Maximum Profit} = 150$$

The items selected to achieve this maximum profit are:

$$\text{Selected Items} = \{1, 2, 3, 5, 9\}$$

Solution to Question 4

To prove that the 0/1 Knapsack Problem has the **optimal substructure property**, we will use a proof by contradiction.

Definition of Optimal Substructure

A problem is said to have the **optimal substructure property** if an optimal solution to the problem contains optimal solutions to its subproblems. For the 0/1 Knapsack Problem, this means that if we have an optimal selection of items that maximizes the profit for a given weight limit K, then any subset of this selection should also be optimal for its respective weight limit.

Proof by Contradiction

Assume, for contradiction, that the 0/1 Knapsack Problem does not have the optimal substructure property. This would mean that there exists an optimal solution for a given capacity K and items that includes a subset of items that is not optimal for a smaller capacity K'.

1. Let S be the set of items chosen in the optimal solution for capacity K that maximizes the profit. 2. Suppose $S' \subset S$ is a subset of S corresponding to a smaller knapsack capacity $K' < K$. 3. According to our assumption, S' is not an optimal solution

for capacity K'. Therefore, there exists another set S'' with total weight $\leq K'$ and higher profit than S'.

However, if we replace S' in S with S'', we would achieve a higher profit for the original capacity K, contradicting our initial assumption that S is the optimal solution for K.

Thus, our assumption that the optimal substructure property does not hold is false. Therefore, the 0/1 Knapsack Problem must indeed have the **optimal substructure property**.

Solution to Question 5

A sorting algorithm is called **stable** if it preserves the relative order of records with equal keys. In other words, if two items have the same key or value, a stable sort will ensure that their original order is maintained in the sorted output.

An **unstable sort**, on the other hand, does not guarantee to maintain the relative order of records with equal keys.

Examples of Stable and Unstable Sorts

1. **Stable Sort Example: Merge Sort** - Suppose we have a list of tuples $(3,'a'), (2,'b'), (3,'c'), (1,'d')$, where the first element is the sorting key. - Using a stable sort like **Merge Sort**, the sorted output would be: $(1,'d'), (2,'b'), (3,'a'), (3,'c')$. - Notice that the relative order of $(3,'a')$ and $(3,'c')$ is preserved, as $(3,'a')$ appears before $(3,'c')$ in both the input and output.

2. **Unstable Sort Example: Quick Sort** - Using Quick Sort on the same list $(3,'a'), (2,'b'), (3,'c'), (1,'d')$, the output might be $(1,'d'), (2,'b'), (3,'c'), (3,'a')$. - In this case, the relative order of $(3,'a')$ and $(3,'c')$ is not preserved, as $(3,'c')$ appears before $(3,'a')$ in the sorted output.

Summary

Stable sorts like Merge Sort and Bubble Sort maintain the order of equal elements, which can be important in applications where the original order has significance. Unstable sorts like Quick Sort and Heap Sort do not guarantee the order of equal elements, which can lead to variations in order for elements with equal keys.

Solution to Question 6

Improving Quick Sort to Make it Stable with a Linear Time Partition Operation

The standard Quick Sort algorithm is not stable because the partition operation does not preserve the order of elements with the same key. Here, we will design an improved **linear-time partition** method that preserves the order, enabling a **stable version of Quick Sort**. We may use additional memory to accomplish this.

Improved Partition Operation Design

To make Quick Sort stable, we can use an additional array (or list) to perform the partition operation in a way that maintains the order of elements with the same key. Here's how the improved partition works:

1. **Initialize two arrays:** Create two additional lists, 'left' and 'right', where 'left' will hold elements less than the pivot, and 'right' will hold elements greater than or equal to the pivot.

2. **Partitioning Elements:** Iterate through the original array. For each element: - If the element is less than the pivot, append it to the 'left' array. - If the element is greater than or equal to the pivot, append it to the 'right' array.

3. **Combine Arrays:** After partitioning, combine 'left', the pivot, and 'right' back into the original array in that order. This step maintains the order of elements in 'left' and 'right', ensuring stability.

Why It Runs in Linear Time

This partition operation runs in linear time because: - Each element is processed exactly once in the iteration, giving a time complexity of $O(n)$ for partitioning. - The combining step, where we concatenate 'left', the pivot, and 'right' back into the original array, also takes $O(n)$ time.

Thus, the overall time complexity of this partitioning operation is $O(n)$, and it enables a stable version of Quick Sort by preserving the relative order of elements with equal keys.

Solution to Question 7

Using Radix Sort to Sort a Sequence with Intermediate Steps

The Radix Sort algorithm sorts numbers by processing each digit from the least significant to the most significant. We'll apply Radix Sort to the given list of numbers and show intermediate results after sorting by each digit.

Input

$$\text{Input: } [3453, 2675, 1211, 7642, 1352, 121, 8642, 6263, 135, 12]$$

Radix Sort Steps

Radix Sort processes each digit from the least significant (units place) to the most significant digit, sorting at each step. Numbers with fewer digits are padded with leading zeros for alignment.

1. Sorting by Units (1s) Digit

Consider the last digit of each number:

- 3453 (3), 2675 (5), 1211 (1), 7642 (2), 1352 (2), 121 (1), 8642 (2), 6263 (3), 135 (5), 12 (2)

After sorting by units digits:

$$[1211, 121, 7642, 1352, 8642, 12, 3453, 6263, 2675, 135]$$

2. Sorting by Tens (10s) Digit

Now consider the second-to-last digit (tens place) of each number:

- 1211 (1), 121 (2), 7642 (4), 1352 (5), 8642 (4), 12 (1), 3453 (5), 6263 (6), 2675 (7), 135 (3)

After sorting by tens digits:

$$[1211, 12, 121, 135, 7642, 8642, 1352, 3453, 6263, 2675]$$

3. Sorting by Hundreds (100s) Digit

Now consider the third-to-last digit (hundreds place) of each number:

- 1211 (2), 12 (0), 121 (1), 135 (1), 7642 (6), 8642 (6), 1352 (3), 3453 (4), 6263 (2), 2675 (6)

After sorting by hundreds digits:

$$[12, 121, 135, 1211, 1352, 6263, 3453, 2675, 7642, 8642]$$

4. Sorting by Thousands (1000s) Digit

Now consider the fourth-to-last digit (thousands place) of each number:

- 12 (0), 121 (0), 135 (0), 1211 (1), 1352 (1), 6263 (6), 3453 (3), 2675 (2), 7642 (7), 8642 (8)

After sorting by thousands digits:

$$[12, 121, 135, 1211, 1352, 2675, 3453, 6263, 7642, 8642]$$

Final Sorted Result

The sorted sequence after all steps is:

$$[12, 121, 135, 1211, 1352, 2675, 3453, 6263, 7642, 8642]$$

Solution to Question 8

Max-Heap Property Check and Fix with MAX-HEAPIFY

The question requires us to identify the node that does not satisfy the max-heap property and fix it using the MAX-HEAPIFY operation.

Identifying the Incorrect Node

In a max-heap, each parent node must be greater than or equal to its child nodes. Observing the heap structure:

- The root node is 43, which is greater than its children 2 and 14, so it satisfies the max-heap property. - The node with value 2 has children 7 and 15. Here, 2 is less than 15, which violates the max-heap property.

Therefore, the node with value **2** does not satisfy the max-heap property.

Step-by-Step MAX-HEAPIFY Operation

To fix this violation, we perform the MAX-HEAPIFY operation on the node with value 2. Here is the process in detail:

1. **Compare Node 2 with Its Children**: - Node 2 has two children: 7 and 15. - Among these, 15 is the largest.

2. **Swap Node 2 with Node 15**: - Swap the values of Node 2 and Node 15. - After the swap, 15 becomes the parent, and 2 moves down to the position previously occupied by 15.

3. **Check the New Position of Node 2**: - The new position of Node 2 (now a child of 7) has children 6 and 5. - Since 2 is smaller than both 6 and 5, another swap is required to maintain the max-heap property.

4. **Swap Node 2 with Node 6**: - Swap the values of Node 2 and Node 6. - Now, 6 is in the parent position, and 2 is moved down further as its child.

5. **Final Check**: - The new position of Node 2 has no children, so no further comparisons are needed.

After these steps, the entire tree satisfies the max-heap property.

Answer Summary

- The node that did not satisfy the max-heap property was the node with value 2. - We applied the MAX-HEAPIFY operation, swapping 2 with 15, and then with 6, to restore the max-heap property.

Solution to Question 9

Building a Max-Heap from an Unordered Array

Given an unordered input array:

$$A = [9, 2, 3, 12, 1, 8, 11, 10, 4, 7, 6, 5]$$

We will use the 'BUILD-MAX-HEAP' operation to transform this array into a max-heap. The process starts from the last non-leaf node and applies 'MAX-HEAPIFY' from the bottom up.

Initial Array Indexing

For reference, here is the indexed version of the array:

Index	1	2	3	4	5	6	7	8	9	10	11	12
Value	9	2	3	12	1	8	11	10	4	7	6	5

The array has 12 elements, so the last non-leaf node is at index $\lfloor \frac{n}{2} \rfloor = 6$.

Step-by-Step Build-Max-Heap Process

Starting from index 6 and moving up to index 1, we apply 'MAX-HEAPIFY' at each node. Here are the steps:

1. **MAX-HEAPIFY at index 6**: - Value: 8 - Children: 6 (index 11) and 5 (index 12) - Since 8 is already larger than both children, no changes are needed.

2. **MAX-HEAPIFY at index 5**: - Value: 1 - Children: 7 (index 10) and 6 (index 11) - Largest value is 7 at index 10. Swap 1 and 7. - New array: $[9, 2, 3, 12, 7, 8, 11, 10, 4, 1, 6, 5]$

3. **MAX-HEAPIFY at index 4**: - Value: 12 - Children: 10 (index 8) and 4 (index 9) - Since 12 is already larger than both children, no changes are needed.

4. **MAX-HEAPIFY at index 3**: - Value: 3 - Children: 8 (index 6) and 11 (index 7) - Largest value is 11 at index 7. Swap 3 and 11. - New array: $[9, 2, 11, 12, 7, 8, 3, 10, 4, 1, 6, 5]$

5. **MAX-HEAPIFY at index 2**: - Value: 2 - Children: 12 (index 4) and 7 (index 5) - Largest value is 12 at index 4. Swap 2 and 12. - New array: $[9, 12, 11, 2, 7, 8, 3, 10, 4, 1, 6, 5]$ - Now, apply 'MAX-HEAPIFY' on the subtree rooted at index 4: - Value: 2 - Children: 10 (index 8) and 4 (index 9) - Largest value is 10 at index 8. Swap 2 and 10. - New array: $[9, 12, 11, 10, 7, 8, 3, 2, 4, 1, 6, 5]$

6. **MAX-HEAPIFY at index 1**: - Value: 9 - Children: 12 (index 2) and 11 (index 3) - Largest value is 12 at index 2. Swap 9 and 12. - New array: $[12, 9, 11, 10, 7, 8, 3, 2, 4, 1, 6, 5]$ - Now, apply 'MAX-HEAPIFY' on the subtree rooted at index 2: - Value: 9 - Children: 10 (index 4) and 7 (index 5) - Largest value is 10 at index 4. Swap 9 and 10. - New array: $[12, 10, 11, 9, 7, 8, 3, 2, 4, 1, 6, 5]$

Final Max-Heap

After applying 'BUILD-MAX-HEAP', the array is transformed into a max-heap:

$$A = [12, 10, 11, 9, 7, 8, 3, 2, 4, 1, 6, 5]$$

Answer Summary

- The initial array was $[9, 2, 3, 12, 1, 8, 11, 10, 4, 7, 6, 5]$. - After applying 'BUILD-MAX-HEAP', the final max-heap is $[12, 10, 11, 9, 7, 8, 3, 2, 4, 1, 6, 5]$.