Student Name : Numan Salim Shaikh
STU ID : 40266934

Assignment 2 – COMP 6651

Q1: Rod-Cutting Modification

Modified Algorithm for Rod Cutting ->

ROD_CUT_MAXIMIZE(p, n, c):

```
let r[0..n] be a new array
r[0] = 0              # revenue of length 0 is 0
for j = 1 to n:
   q = p[j]           # Initialize q with no cut (whole rod is sold)

   for i = 1 to j - 1:    # iterate and try all possible first cuts (i < j)

      # Maximize the revenue considering a cut at i and subtracting cut cost
      q = max(q, p[i] + r[j - i] - c)

   r[j] = q           # Store the best revenue for a length j
return r[n]
```

Explanation :
The modified rod-cutting problem introduces a fixed cost c c for every cut, requiring a balance between higher revenue from smaller pieces and the cost of making cuts. The goal is to maximize net revenue by either selling the rod whole or dividing it optimally. Using dynamic programming, we compute the best revenue for each rod length incrementally.

For every length j j, the algorithm evaluates two options:

- No cuts: The entire rod is sold for $p\,[\,j\,]$ p[j].
- With cuts: For each possible first cut at $i < j$ i<j, the revenue is $p\,[\,i\,] + r\,[\,j - i\,] - c$ p[i]+r[j−i]−c.

The algorithm stores the maximum revenue for each length in an array, ensuring optimal solutions are reused to avoid redundant computation. By only subtracting c c when cuts are made, the solution accurately reflects the trade-off between cut costs and increased revenue.

This bottom-up approach ensures an efficient ***O(n²)*** solution, balancing profitability with cut penalties to deliver the maximum net revenue.

Q2: LCS without *b* table

Pseudocode for LCS

```
LCS (c, X, Y, i, j):
  if c[i, j] == 0
    return
  if X[i] == Y[j]
    LCS(c, X, Y, i - 1, j - 1)
    print X[i]
  else if c[i - 1, j] > c[i, j - 1]
    LCS(c, X, Y, i - 1, j)
  else
    LCS(c, X, Y, i, j - 1)
```

Explanation :

The modified algorithm reconstructs the **Longest Common Subsequence (LCS)** using only the c-table, which stores the LCS lengths for prefixes of two sequences $X = \langle x_1, x_2, \ldots, x_m \rangle$ and $Y = \langle y_1, y_2, \ldots, y_n \rangle$.
The key idea is to **trace back through the table** to identify which characters are part of the LCS. Depending on the comparison of characters and LCS values in neighboring cells, the code handles **three possible cases**

1. **Case 1: Characters Match**
    - If X[i]=Y[j] , the character is part of the LCS. The algorithm **moves diagonally** to c[i−1,j−1]c[i-1, j-1]c[i−1,j−1] and prints the character.
2. **Case 2: Move Up**
    - If the characters do not match, and the value in the **upper cell** c[i−1,j] is greater, the optimal solution excludes the character from XXX. The algorithm **moves up** to c[i−1,j].
3. **Case 3: Move Left**
    - If the value in the **left cell** c[i,j−1]c[i, j-1]c[i,j−1] is greater or equal, the optimal solution excludes the character from YYY. The algorithm **moves left** to c[i,j−1].

These cases ensure correct backtracking through the c-table to reconstruct the LCS efficiently in ***O(m + n)*** time.

Q3: Modified Activity Selection Algorithm to maximize activities

ACTIVITY-SELECTION-WITH-VALUES(activities) :

```
#Sort activities by finish time in ascending order
Let dp[0..n-1] be an array to store maximum values
Let p[0..n-1] be an array to store the index of the last compatible activity

for i = 0 to n-1
  p[i] = -1
  for j = i - 1 downto 0
    if activities[j].finish <= activities[i].start
      p[i] = j
      break

dp[0] = activities[0].value

for i = 1 to n-1
  if p[i] == -1
    dp[i] = max(dp[i-1], activities[i].value)
  else
    dp[i] = max(dp[i-1], activities[i].value + dp[p[i]])

return dp[n-1]
```

Explanation :

1. **Sorting Activities**: The activities are first sorted by their finish times. This step is crucial as it helps in efficiently finding compatible activities.
2. **Initializing Arrays**: Two arrays, `dp` and `p`, are initialized:
   - `dp[i]` stores the maximum value that can be achieved by considering activities up to index i.
   - `p[i]` stores the index of the last activity that is compatible with activity i.
3. **Finding Compatible Activities**: For each activity iii, the code checks the previous activities (in reverse order) to find the last compatible activity. This is done using a loop from i−1 to 100.
4. **Dynamic Programming Calculation**: The `dp` array is filled based on the following cases:
   - **Case 1**: If the last compatible activity index p[i] is -1, it means activity *i* cannot be combined with any previous activities. The maximum value for dp[i] will be the maximum of the previous value dp[i−1] and the value of the current activity activities[i].values
   - **Case 2**: If there is a compatible activity (i.e., p[i] is not -1), the value of dp[i] is calculated as the maximum between the previous value dp[i−1]

and the sum of the current activity's value and the value of the last compatible activity dp[p[i]]

- **Case 3**: If the maximum value is the same from both cases, it can be resolved arbitrarily, either including or excluding the current activity.

Time Complexity Analysis

- **Sorting**: The initial sort of activities takes ***O(nlog n)*** time.
- **Finding Compatible Activities**: The search for compatible activities is performed using a binary search, allowing for ***O(log n)*** complexity for each S_{i+1} calculation.
- **Total Complexity**: The overall time complexity for constructing all partial solutions S_i is ***O(nlog n)***

Summary

The algorithm effectively utilizes dynamic programming to achieve an optimal solution for the activity selection problem with values. By systematically building on previously computed solutions while ensuring efficient memory use, the algorithm maintains a time complexity of ***O(nlog n)*** .The consideration of different cases ensures that each activity is evaluated for its maximum possible contribution to the total value, leading to the correct final solution.

Q4 : BST Cost

Let T be a full binary tree with n leaves. We will apply the induction hypothesis to the number of leaves in T. For the base case where n=2 (the case where n=1 is trivially true), there are two leaves x and y that share the same parent z. The cost of T can be calculated as follows:

$$B(T) = f(x) \cdot d_T(x) + f(y) \cdot d_T(y) = f(x) + f(y)$$

This is because both $d_T(x)$ and $d_T(y)$ equal 1, resulting in:

$$B(T) = f(child\ 1\ of\ z) + f(child\ 2\ of\ z)$$

Therefore, the theorem holds for this case. Now, let's assume the theorem is true for trees with n−1 leaves, where n>2. Let c_1 and c_2 be two sibling leaves in T with the same parent p. By removing c_1 and c_2, we obtain a new tree T'. By the induction hypothesis, we can express the total cost as:

$$B(T) = \sum_{l' \in T'} f(l').d_T(l') = \sum_{i' \in T'} f(child\ 1\ of\ i') + f(child\ 2\ of\ i')$$

To calculate the cost of T:

$$B(T) = \sum_{l \in T} f(l).d_T(l)$$

This can be expanded as:

$$B(T) = \sum_{l \neq c_1, c_2} f(l).d_T(l) + f(c_1).d_T(c_1) - 1 + f(c_2).d_T(c_2) - 1 + f(c_1).f(c_2)$$

This simplifies to:

$$B(T) = \sum_{i' \in T'} \big(f(child\ 1\ of\ i') + f(child\ 2\ of\ i')\big) + f(c_1) + f(c_2)$$

Finally, we can represent it as:

$$B(T) = \sum_{i \in T} \big(f(child\ 1\ of\ i) + f(child\ 2\ of\ i)\big)$$

Thus, we conclude that the statement is true.

Q5 : Stack Amortization

To analyze the cost of performing a sequence of stack operations, including making backups, we can use amortized analysis to show that the total cost of n stack operations is ***O(n)***.We can achieve this by assigning suitable amortized costs to the various stack operations.

Amortized Cost Calculation

1. **Charge for Each Operation**:
    - For each stack operation, we charge an amortized cost of 3:
        - **Actual Cost**: The actual cost of performing the operation.
        - **Backup Cost**: An additional charge to account for the eventual copying of elements when we reach every ***k*** operations.
    - This means the cost of each operation is effectively treated as follows:
        - If the operation is not an exact power of 2, we pay 1 and store 2 as credit.
        - If the operation is an exact power of 2, we pay *i* (the index of the operation) using the stored credit.
2. **Example of Costs**:
    - Let's denote the cost of the i^{th} operation as c_i :

$$c_i = \begin{cases} i & \text{if } i \text{ is the exact power of } 2 \\ 1 & otherwise \end{cases}$$

    - The amortized cost, $\hat{c}_i$ for each operation is set to 3.

$$\hat{c}_i = 3$$

Credit and Actual Costs

- The process can be visualized in a table format:

Operation	Amortized $\hat{c}_i$	Actual Cost c_i	Credit Remaining
1	3	1	2
2	3	2	3
3	3	1	4
4	3	4	6
5	3	1	8
6	3	1	10
7	3	1	12
8	3	8	7
9	3	1	9
10	3	1	11

The credit stored is never negative, ensuring that any surplus is available to cover the actual costs of operations that have higher costs.

Total Cost Analysis

- The total amortized cost for n operations can be expressed as:

$$\sum_{i=1}^{n} \hat{c}_i = 3n$$

- Given that the total actual cost must be less than or equal to the total amortized cost:

$$\sum_{i=1}^{n} \hat{c}_i \geq \sum_{i=1}^{n} c_i$$

- The credit accumulated ensures that:

$$\text{Credit} = \text{Amortized Cost} - \text{Actual Cost} \geq 0$$

Conclusion

Since each operation has an amortized cost of ***O(1)*** and the credit balance never goes negative, the total cost of performing n operations, including the stack backups, is ***O(n)***. This amortized analysis effectively demonstrates that even with the backup process in place, the overall cost remains efficient.

Q6 : Binary SEARCH and INSERT

(A) : SEARCH

Description: The SEARCH operation in this data structure involves checking each of the k sorted arrays A_0, A_1, ..., A_{k-1} sequentially. Each array A_i has a length of 2^i and is individually sorted. To find an element, we perform a binary search within each array. To search one of them takes O(log m) time. In a worst-case scenario, we can assume that all arrays are full, where k = $\lceil \log_2(n+1) \rceil$, and an unsuccessful search is performed.
Analysis: The total time taken, denoted as T(n), can be expressed as:

$T(n) = \theta(\log 2^{k-1} + \log 2^{k-2} + \cdots log 2^1 + log 2^0)$

$= \theta((k-1)+(k-2)+\cdots+1+0)$

$= \theta((k(k-1)/2)$

$= \theta((\log(n+1)(\log(n+1)-1)/2)$

$T(n) = \theta(log^2 n)$

We sequentially traverse the lists and perform a binary search on each one, as we lack information about the relationship between the lists. In the worst case, every list is used. List *i* with length 2*i* is sorted, allowing us to search it in O(i) time. Since *i* varies from 0 to O(log n), the runtime of the SEARCH operation is $O(\log_2 n)$

(B) : INSERT

Description: To perform an INSERT operation, we first place the new element into the smallest array A_0 is already full, we move its contents to A_1, and if A_1 is full, we proceed to A_2, continuing this process up to A_{k-1}. If all arrays are full, the contents from the last filled array are merged into a new array, A_k . This process ensures that each insertion maintains the sorted order of the arrays.

1. **Insertion Process:** When a new element is inserted, it is added to A_0. If A_0 is full, we must merge it with the next array. The merging process for two sorted arrays takes linear time based on their total length.

2. **Merging Time Complexity:** If we consider the worst-case scenario where merging involves all filled arrays, the time taken can be expressed as:

   $O(2^m)$ where m = $\lceil \log_2(n+1) \rceil$

   In this case, the worst-case time complexity for the INSERT operation becomes O(n) when merging all filled arrays into A_k .

Analysis: To analyse the amortized cost of INSERT operations, we employ the accounting method. Each insertion incurs a cost of ***O(log n)***

By assigning this credit to each new element, we cover the potential future merging costs. Since a single element can only be merged into a larger list and the number of lists is bounded by O(log n), this strategy ensures that the credits collected sufficiently cover any merging costs incurred later.

Therefore, the amortized cost for the INSERT operation is: ***O(log n)***