S. Jayanth Kumar
21BCE1218

① Design of Algorithm :

Understand the problem : Taking a problem of finding the minimum no. of coins to make a given change. We are going to illustrate how to apply the different stages involved in design of a dynamic Programming.

Different stages involved : (Illustration)

(i) **Define the problem :** The problem can be defined as finding the min no. of coins needed to make a given change, given a set of coins of different denominations.

(ii) **Identify the Optimal Substructure :** To find the Min no. of coins of different denominations to make the given change, the subproblems are the no. of coins required to make change for smaller amounts.

(iii) **Define Recurrence Relation :** The Recurrence relation can be defined as the no. of coins required to make change for the amount i, as the min no. of

coins required to make the change for the amount i, as the min of the no. of coins required to make change for i-j, where j is denomination of coin.

(iv) Initialize the Table : The table can be initialized with base cases which are the no. of coins required to make change for 0, which is 0.

(v) Fill the Table, Extract the Solution : Compute the no. of coins required for each amt, starting from base cases. The soln can be extracted by finding the value of the last cell in the table, which represents the min no. of coins required to make change for given Amt.

Write the pseudocode :

```
min coins (coins, n, amount)
1. table [amt + 1];
2. table [0] = 0;
3. for i = 1 to amt
4.     for j = 0 to n-1.
5.         if (coins[i] <= i)
6.             sub_res = table [i - coins[j]];
7.             if (sub_res != INT_MAX and sub_res + 1
                                < table [i])
8.                 table[i] = sub_res + 1;
9. return table[amt];
```

Analysis of Algorithm :

Proof of Correctness (PoC) :

Initialization : Before the first iteration of the outer loop, the table is initialized with base case of table[0] = 0 which represents the min no. of coins required to make change for 0, which is 0.

Maintenance : In each iteration of outer loop, the the value of table[i] is updated based on recurrence relation defined in the code. The recurrence relation defines the no. of coins required to make change for the amt i as min no. of coins required to make change for i-j, This ensures that value of table[i] is always updated.

Termination : After last iteration, the value of table[amt] is returned, This value is guaranteed to be the min no. of coins required to make change for given Amt, as it is the result of updating the table based on the recurrence relation, and loop invariant holds true for all iterations of the loop.

Time Complexity :-

$T(n) = O(n)$ for all the cases as it takes $O(n)$ time to calculate each subproblem and there are amount subproblems.

Space complexity is $O(amt)$

(2) The following properties need to be exhibited by a problem in order for a dynamic programming based pseudocode to be a feasible solution for solving it.

(i) Overlapping subproblems

(ii) Optimal substructure.

(iii) Memoization

Understand the problem : Taking the problem of finding the Longest Increasing subsequence of a given sequence. The problem is characterized by the fact that it requires finding a subsequence such that its elements are in increasing order.

Illustration :

10	22	9	33	21	50	41	60	80

Dynamic programming steps :

Overlapping subproblems : To find the LIS of the given sequence, we need to solve several subproblems. For example, we need to find the LIS of a subarray, starting from the first element and ending at the current element. This means that we're solving multiple subproblems that are overlapping i.e, they share common elements.

Optimal Substructure : The optimal sol^n of a subproblem can be used to find the optimal sol^n of a larger subproblem. For example, the LIS of a subarray ending at the current element can be used to find the LIS of the next element.

Memoization : We can store the solutions of subproblems in an array to avoid recalculating them.

Based on the above thought-process, it can be concluded that the problem of finding the LIS is solvable by DP.

Write the pseudocode :

```
LIS (sequence, n)
1. dp[n]
2. for i = 0 to n-1
3.     dp[i] = 1;
4.     for 1 = 1 to n-1
5.         for j = 0 to i-1
6.             if (sequence[i] > sequence[i])
7.                 dp[i] = max (dp[i], dp[j]+1);
8. maxLIS = 0
9. for i=0 to n-1
10. max LIS = max.(maxLIS, dp[i]);
11. return max LIS;
```

Analysis of Algorithm :

Proof of Correctness (POC) :

Intialization : dp[i] is initialized to 1 for all elements in the sequence. This satisfies the loop Invariant because dp[i] represents the length of the longest increasing subsequence ending at the i^{th} element in the sequence, and since there's only one element, the length of the longest increasing subsequence is 1

Maintenance : In each iteration of the outer loop, we compare the i^{th} element with all elements in the sequence [0 to i-1]. If the i^{th} element is greater than an element j, then the length of the LIS ending at the i^{th} element can be updated as dp[i] = max (dp[i], dp[j] + 1). This ensures that at the end of inner loop, dp[i] represents the length of the longest increasing subsequence ending at i^{th} element in the sequence.

Termination : The final answer is the max value of dp[i] for all elements in the sequence. This represents the length of the longest increasing subsequence in the entire sequence.

Time Complexity : $T(n) = O(n^2)$. The outer loop iterates n times, and the inner loop iterates i times for each iteration of the outer loop. The total no. of iterations is $\Rightarrow O\left(\frac{n(n+1)}{2}\right)$

Asymptotic Notations :

$\Theta(n^2)$, $O(n^2)$, $\Omega(n^2)$, $o(n^2)$ and $\omega(n^2)$

Write the pseudocode :-

```
fiba(n, dp)
1. if (n<=0)
2.    return 0;
3. if(n==1)
4.    return 1;
5. if(n==2)
6.    return 2;
7. if (dp[n]!=-1)
8.       return dp[n]
9.       dp[n] = fiba(n-1,dp) + fiba(n-2, dp)+fiba(n-3,dp)
10.      return dp[n];
11.   int dp[n+1];
12. for i=0 to n
13.         dp[i] = -1
14.         return fiba(n,dp);
```

Proof of Correctness (POC) :

Initialization : The value of dp[i] is set to -1 for all $0 <= i <= n$. This means fibo[i] is not yet computed.

Maintenance : At each iteration, the value of fibo(i) is computed and stored in dp[i] if it hasn't already been computed. The value of fibo(i) is computed using the values of fibo(i-1), fibo(i-2) and fibo(i-3) which has already been stored in the dp array.

Termination : At end of each iteration, the value of fibo(i) is stored in dp[i], so the loop invariant is true before and after each iteration.

Finally after all iterations, the value of fibo(n) is stored in dp[n].

Time Complexity : $T(n) = O(n)$. In each iteration, we compute fibo(i) and store in dp array.

Asymptotic Notations :

$\Theta(n)$, $O(n)$, $o(n)$, $\Omega(n)$, $\omega(n)$

④ Design of Algorithm :

Understand the problem :

In the given Rod-cutting problem, we are going to return the max-revenue that is obtainable by selling the rod of length n after cutting them into smaller pieces.

Input : An Array price[] of different prices of different rod lengths and length of the rod.

Output : The Maximum revenue that is obtainable by selling the rod of length n.

Validate the logic with simple Illustration :

Let us consider,

price[] = [1, 5, 8, 9, 10, 17, 17, 20, 24, 30]

* First cut at 1 unit length ⇒ Max revenue ⇒ 1unit
Then recursively consider rest of length 9units.
* Similarly, second cut at 2 unit length ⇒
Max revenue from selling two pieces is 5units.
Then recursively consider rest of length 8units.

⇒ At end of all possible combinations, the max revenue obtainable by selling rod of length 10 units ⟹ 30 units

Write the pseudocode :

```
RodCutting (price, n)
1. max-revenue = 0
2. for i = 1 to n.
3.     for j = i to n
4.         max-revenue = max(max-revenue, price[j-i],
                             RodCutting(price, n-i))
5. return max-revenue.
```

Analysis of Algorithm

proof of correctness (POC) :

Initialization : The loop Invariant holds true at the start of the first recursive call, where the length of the rod is n units and the max revenue obtainable by selling the rod isn't Known yet.

Maintenance : At each step of recursive call, the Algorithm calculates the max revenue obtainable by selling the rest of the rod after cutting it into smaller pieces. The calculation is based on the information from previous step and the current price of the rod. This algorithm updates the max revenue obtained by selling the cut pieces.

Termination : When the length of the rod becomes 0, the max revenue obtainable by selling the cut pieces is returned. The Algorithm terminates and the final answer is max revenue obtainable by selling the rod of length n.

Time Complexity : $T(n) = O(2^n)$. Because, for each length of the rod, the algorithm performs two recursive calls to calculate the max revenue obtainable by selling the cut pieces.

Asymptotic Notations

$O(2^n)$, $o(2^n)$, $\Omega(2^n)$, $\Theta(2^n)$, $\omega(2^n)$

(5) Design of Algorithm :

Understand the problem : Given a Rod-cutting problem, that uses DP. (top-down). We have to return the max revenue that is obtainable by selling the rod of length n after cutting them into smaller pieces.

Validate the Logic with simple Illustration :

Let us consider a rod of length n = 4 and prices = [0, 2, 5, 7, 8]

The revenue_table is initialized with '-1' for each length 'i' where 0 <= i <= n.

First call => max-revenue(price, 4, revenue-table) will compute max revenue for a rod of length 4 by considering the possible cuts and selecting the one that gives the highest revenue.

Write the pseudocode :

```
max - revenue (price, n, revenue - table)
1.    if revenue - table[n] != -1
2.        return revenue - table[n]
3.    if n == 0
4.        return 0
5.    max_val = -inf
6.    for i = 1 to n
7.        max_val = max (max_val, price[i] +
                        max - revenue(price, n-i,
                                      revenue-table))
8.    revenue - table[n] = max_val
9.    return max_val
```

Analysis of Algorithm :

proof of correctness (POC) :

Initialization : At the start of each iteration of the 'for' loop, revenue - table[i] stores the max revenue that can be obtained by selling a rod of length i after cutting it into smaller pieces.

Maintenance : revenue - table[i] is initialized with -1, for each i, meaning that the max revenue for each length hasn't computed yet.

Termination : revenue - table[n] with max revenue that can be obtained by selling a rod of length 'n' after cutting it into smaller pieces which is calculated by considering the possible cuts and selecting one that gives highest revenue.

Time complexity : $T(n) = O(n^2)$

Asymptotic Notations

$O(n^2)$, $o(n^2)$, $\Omega(n^2)$, $\omega(n^2)$, $\Theta(n^2)$

⑥ Design of Algorithm :

Understand the problem : In the given rod-cutting problem, that uses DP (bottom-up). We are returning the max-revenue that is obtainable by selling the rod of length n after cutting them into smaller pieces.

Validate the Logic with simple Illustration :

Consider n = 8

prices = [0, 1, 5, 8, 9, 10, 17, 17, 20]

max-revenue = max-revenue-bottomup (prices, 8)

revenue-table[0] = 0

for diff values of i iterate through the loop.

Finally we set,

max revenue that is obtained by selling a rod of length 'n' after cutting it into smaller pieces

max-revenue = revenue-table[8] = 22

Write the pseudocode :-

```
max_revenue_bottomup(prices, n)
1.  revenue_table[0] = 0
2.  for i = 1 to n
3.      revenue = -1
4.      for j = 1 to i
5.          revenue = max(revenue, price_table[i]
                + revenue_table[i-j])
6.      revenue_table[i] = revenue
7.  return revenue_table[n]
```

Analysis of Algorithm :-

Proof of correctness (POC) :-

Initialization :- The 'revenue_table' is initialized to -1 for all lengths except 0 which is initialized to 0. This ensures that the revenue table is initialized properly.

Maintenance :- The loop iterates from i = 1 to n. For each iteration i it calculates the max revenue that can be obtained by selling a rod length i by considering the possible cuts and selecting the one that gives highest revenue. That's why it maintains loop Invariant.

Termination : The loop terminates when i=n at which point the revenue-table has been filled with max revenue that can be obtained for each length i from 1 to n. The final value in revenue-table[n] is then returned.

Time Complexity : $T(n) = O(n^2)$

Asymptotic Notations :

$O(n^2)$, $o(n^2)$, $\Omega(n^2)$, $\omega(n^2)$, $\theta(n^2)$

④ Design of Algorithm :

Understand the problem :- Given the length of rod n and prices p_i, we will return the max revenue along with lengths of the smaller pieces.

Validate the logic with simple Illustration :

rod-length = 5, prices = [1, 5, 8, 9, 10]

Initializing r and decomposition. We iterate the outer loop i=1, i=2, i=3 . . . . .

After final iterations,

i = 5 ; r = [0, 1, 5, 8, 9, 10], decomposition = [0, 1, 2, 3, 4, 5]

So the max revenue for entire rod is 10 & the decomposition of the rod is [5].

This means that we can cut the rod into one piece with length 5 to get max revenue of 10.

Write the Pseudocode :

```
function rod-cut (n, prices)
1.    r = array of size n+1, filled with 0
2.    decomposition = array of size n+1 filled with -1.
3.    for i = 1 to n
4.        q = -1
5.        for j = 1 to i
6.            if q < prices[j] + r[i-j]
7.                q = prices[j] + r[i-j]
8.                decomposition[i] = j
9.    r[i] = q
10.   return (r[n], decomposition)
```

Analysis of the Algorithm :

Initialisation : At the start of each of the outer loop, the r[i] array holds the max revenue that can obtained for a rod of length j using prices p[i] for rods of lengths 1, 2 . . . . i.

Maintenance : For each j in inner loop, r[j] is updated as max revenue that can be obtained by cutting the rod of length 'j' into 2 pieces and taking max of revenues obtained. This ensures that r[j] array holds max revenue that can be obtained for rod of length j using prices p[i].

Termination : At end of outer loop, r[n] array holds max revenue that can be obtained for length n using prices p[i].

Time Complexity

$O(n^2)$ as it consists of two Nested loops.

Asymptotic Notations

$O(n^2)$, $o(n^2)$, $\Omega(n^2)$, $\Theta(n^2)$, $\omega(n^2)$

(8)

Input : n → length of rod

p[] → prices for rod of length 1 to n

Output : max_revenue → Max revenue that can be obtained.

cut_lengths → lengths of smaller pieces.

Pseudocode

1. Declare array r[n+1] to store max revenue
2. Declare array cut_lengths[n+1] to store decomposition of 'n'
3. r[0] = 0
4. cut_lengths[0] = 0
5. for i=1 to n
6. max_revenue = -∞
7. for j=1 to 5
8. if (i>=j)
9. if (r[i-j] + p[j] > max_revenue)
10. max_revenue = r[i-j] + p[j]
11. cut_lengths[i] = j
12. r[i] = max_revenue
13. if n%3!=0 and n%5!=0
14. max_revenue = max_revenue - (n%3)
15. return max_revenue and cut_lengths.

(9) Understand the problem: To compute the max revenue that is obtainable by selling the rod of length n after cutting them into k smaller pieces. For each possible value of k we should return the max_revenue obtainable by selling k pieces.

Validate the Logic with simple Illustration:

Length = 8 units, prices = [1, 5, 8, 9, 10, 17, 17, 20]

The max revenue that can be obtained by selling the rod of length 8 cut into 3 pieces is 30. The lengths of smaller pieces would be [2, 3, 3].

Write the pseudocode:

```
1. Declare array r[n+1][k+1]
2. for i=1 to n
3.     for j=1 to k
4.         max_revenue = -∞
5.         for l=1 to min(j,i)
6.             if (r[i-l][j-1] + p[l] > max_revenue)
7.                 max_revenue = r[i-l][j-1] + p[l]
8.         r[i][j] = max_revenue
9. return r[n][k]
```

Analysis of Algorithm

Proof of Correctness (POC) :

Initialization : Before the first iteration of the outer loop, j is 1, which corresponds to cutting the rod into 1 piece. In this case, r[i][1] stores the max revenue that can be obtained by selling rod of length 'i' as a single piece, which is equal to p[i].

Maintenance : During each iteration of outer loop, the algorithm computes the max revenue that can be obtained by selling the rod of length 'i' cut into 'j' pieces by considering all possible combinations of cutting the rod into j pieces of lengths l = 1, 2 . . . . . min(j,i) and choosing the one with max revenue.

Termination : After the kth iteration of the outer loop, j = k and r[n][k] stores the max revenue that can be obtained by selling the rod of length 'n' cut into 'k' pieces.

Time Complexity

O(n*k^2) as each rod of length i from 1 to n we are checking for j pieces from 1 to k.

(10) The DCC strategy and DP are two techniques used to solve optimization problems. Both techniques have their own advantages & disadvantages, and choosing one over the other depends on specific problem.

Longest Common Subsequence of two strings

DCC : Involves dividing the strings into subproblems, finding the LCS of subproblems and combining the solutions to subproblems to find LS of original strings

Algorithm

```
LCS (x, y, m, n)
1. if (m == 0 || n == 0)
2.     return 0
3. if (x[m-1] == x[n-1])
4.     return 1 + LCS (x, y, m-1, n-1)
6. else
7.     return max(LCS (x, y, m, n-1); LCS (x, y, m-1, n))
```

The DCC Approach for LS involves solving same subproblems multiple times, leading to large amt of redundant computations.

DP : Involves breaking the problem down into smaller subproblems and solving each only once, storing the results in a table to be reused later. The DP approach has time complexity of $O(m*n)$ where m and n are lengths of two strings.

Algorithm

```
LCS (A,B,m,n)
1. Create 2D dp of size (m+1) x (n+1)
2. Initialize all elements of dp as 0
3. for i=1 to m
4.     for j = 1 to n
5.         if A[i-1] == B[j-1]
6.             dp[i][j] = dp[i-1][j-1] + 1
7.         else
8.             dp[i][j] = max(dp[i-1][j], dp[i].[j-1])
9. return dp[m][n]
```

In conclusion, DCC is well suited for problems that can be broken down into subproblems. DP is well suited for problems where subproblems have an inherent overlap and can be solved more efficiently by reusing previous computed solutions.