CS403
Programming Assignment 2
Report

Rashika Rathi
B18081

PAGE NO.
DATE: / /

Pseudo code:

Q1: procedure maxStolen (n, values[])
{

define dp[n+1]
dp[0] ← 0
dp[1] ← values[1]
for i = 2 to n
    dp[i] = max(dp[i-1], dp[i-2] + values[i]);
Return dp[n].
}

Complexity analysis:

→ Time complexity : O(N)
Since, we process n recursive calls, and we use memoization technique to store values in array.

→ Space complexity : O(n)
Memory occupied by dp[n+1].

Algorithm Analysis:

Maximum stolen values from first i houses can be either the maximum stolen value from first i-1 houses of the line or max val stolen value from i-2 houses of the line plus value in ith house. So, we'll choose max of these. We'll calculate the values in bottom up manner.

Based on the problem statement, we'll set the initial conditions:

1) If there's no house, the robber can't steal the money.
2) If there's only one house, the robber can only steal that house.
3) If there're two or more houses, the robber'll try to steal the max amount of money without stealing from the two adjacent houses.

We'll subset the base condition to solve it with dp. let's say the input is below:

[1, 3, 4, 4, 3, 3, 7, 2, 3, 4, 5, 1]

We'll compare the amount the robber can steal from the first house and the second house. 1 or 3? 3 is bigger, so the robber will go to stead 3.

On the third house, the robber will think about

whether it's better to steal only 3 or 1+4=5. The robber will steal 5 as it's a bigger no. Then the robber will decide if he should steal 5 or 3+4=7. The robber will go for 7 and so on. This notation can be written as below:

1 3 5 7 8 10 15 15 18 19 23 23,

[1, 3, 4, 4, 3, 3, 7, 2, 3, 4, 5, 1]

Notice that since the third house, we've compared the nums[i] + nums[i-2] and nums[i-1] in order to see which choice could be maximum.

PAGE NO.
DATE:

22 pseudo code:

```
procedure Award ( grade[], size)
{
    values [ ]
        values [0] ← 1   // array to store maxm

for i = 0 to size :   // left to right array
{ if (grade[i] > grade [i-1])
        values [i] ← values [i-1] +1
    else
        values [i] = 1
}
for i = size - 2 to 0   // scanning from right
}                          to left array.
if (grade[i] > grade[i+1])
    tmp ← values [i+1] +1
else
    tmp ← 1
};
ans ← ans + max (tmp, values[i]);
values [i] = tmp
return ans
}
```

Complexity Analysis:
Time complexity : O(n) for traversing the array twice.
Space Complexity : O(n) for values[]

proof :-
Let classify the children into 4 kinds
i) if $grade_{i-1} \geq grade_i \leq grade_{i+1}$
then $child_i$ is valley
ii) If $grade_{i-1} < grade_i \leq grade_{i+1}$
then $child_i$ is fall.
iii) If $grade_{i-1} \geq grade_i > grade_{i+1}$
then $child_i$ is fall.
iv) If $grade_{i-1} < grade_i > grade_{i+1}$.
then child i is peak

Now, we can distribute awards as follows -
Let A_i → No. of awards given to i^{th} child.
for each valley child → 1 Award.
for each rise child → $A_{i-1} + 1$ Awards.
For each fall child → $A_{i+1} + 1$ Awards.
for each peak child → $\max(A_{i-1}, A_{i+1})$ Awards.
Considering two neighbouring children i & i+1 with different grades.
Case 1 :- $grad_i < grade_{i+1}$
Then,
child i is valley & child i+1 is peak.

child i is valley & child i+1 is rise.
child i is rise & child i+1 is peak.
child i is rise & child i+1 is rise.
we'll give awards from left to right and thus child i is given award earlier than child i+1.

Case 2:- $grade_i > grade_{i+1}$

Then,
child i is peak & child i+1 is valley.
child i is fall & child i+1 is valley.
child i is peak & child i+1 is fall.
child i is fall & child i+1 is fall.

we'll give awards from right to left and thus child i+1 is given award earlier than child i.

This'll give minimum no. of awards.

∴ for each valley child i, $A_i \geq 1$.
for each rise child i, $A_i \geq A_{i-1} + 1$
for each fall child i, $A_i \geq A_{i+1} + 1$
for each peak child i, $A_i \geq \max(A_{i-1}, A_{i+1}) + 1$

This easily follows from the constraints of the problem:
Since, Algorithm above satisfies each inequality in best possible way, this means overall no. of awards is minimized.

PAGE NO.
DATE: / /

23 Minimum steps to one problem.

DP approach.

$f(n) = 1 + f(n-1)$

$f(n) = 1 + f(n/2)$ // if n is divisible by 2

$f(n) = 1 + f(n/3)$ // if n is divisible by 3.

pseudo code:

```
procedure getMinSteps (n)
{
    memo table[n+1]
for i = 0 to n
memo table[i] = -1.
for procedure getMinSteps (n, *memo)
{
if n == 1          // Base Case
  return 0
if (memo[n] != -1)
    return memo[n]
res ← getMinSteps (n-1, memo)
if (n % 2 == 0)
    res ← min (res, getMinSteps (n/2, memo))
if (n % 3 == 0)
    res ← min (res, getMinSteps (n/3, memo))
memo[n] ← 1+res
return memo[n]
}
```

Teacher's Signature

Complexity Analysis:

Time complexity: O(n)

~~i is~~ i = 1 to n is traversed once.

space complexity: O(n) for memo[n+1].

Proof using induction:

Base case:

n = 1	0 steps	
n = 2	1 steps	(∵ 2/2 = 1)
n = 3	1 steps	(∵ 3/3 = 1)

Let induction holds true for $n' < n$

Now, we can transform n to either $n-1$, $n/2$, $n/3$.

By induction hypothesis,

table[i], table[n/2], table[n/3] gives min steps ~~to~~ for the transformation.

Thus,

$$table[n] = min(table[n-1], table[n/2], table[n/3]) + 1$$

Hence, given recurrence is true for all $n \in N$.

Q1

pseudo code:

```
procedure cutRod ( price[], n)
{
    val[n+1];
    val[0] ← 0
    define integer i, j
    for i from 1 to n
    {
        max_val ← INT_MIN
        for j from 0 to i-1
        {
            max_val ← max(max_val, price[j] +
                          val[i-j-1])
        val[i] ← max_val
        }
    return val[n]
}
```

Complexity Analysis:

Time Complexity: O(length x len(price array))

$\Rightarrow O(n^2)$

Space Complexity: O(n) i.e., val[n+1].

Algorithm Analysis:

The problem is being solved by dp.

→ Optimal Substructure.

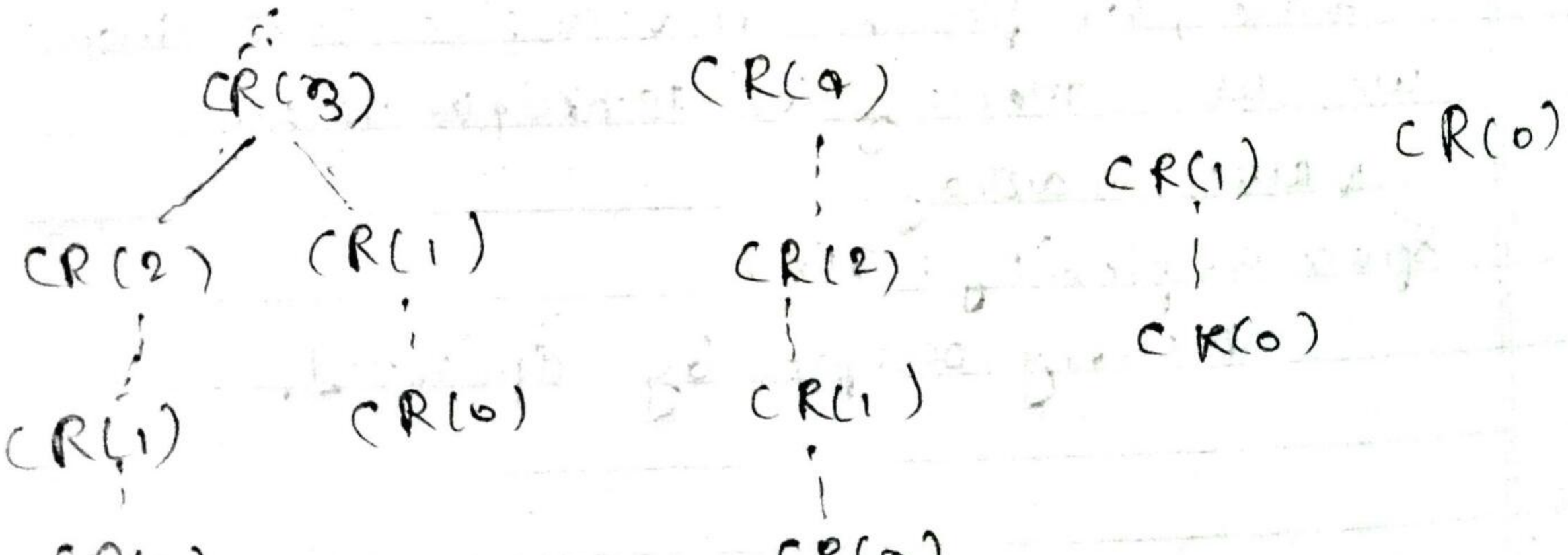
We can get the best price by making a cut at different positions and comparing the values obtained after a cut.

Let cutRod(n) be required value for a rod of length n.

cutRod(n) = max{price[i] + cutRod(n-i-1)
for all i in (0, 1, ..., n-1)}

→ Overlapping subproblems.

We can see that there're many subproblems which are solved again and again. For ex.:



Thus, max-value q will give optimal solution for the given problem.

25

Pseudo code:

```
procedure solve (a[][3], n, i, j)
{
    if n<0
        return 0.
    if dp[n][i]==0
    dp[n][i] ← a[n][i] + solve (a, n-1, (i+1)%.3,
                                (i+2) %.3);
    if dp[n][j]==0
    dp[n][j] ← a[n][j] + solve (a, n-1, (j+1)%.3,
                                (j+2) %.3);

    Return minimum of dp[n][i] and dp[n][j]
}

inside driver function
{
    x ← solve (arr, n-1, 0, 1)
    y ← minimum of x and solve (arr, n-1, 1, 2)
    ans ← minimum of y and solve (arr, n-1, 0, 2)
}
```

→ Complexity Analysis:

→ Time Complexity : O(n)

Here, we use ~~to~~ top-down approach, as the customer will've to minimize the previous cost so as to minimize the next fruit cost.

Also, we can say that Rahul tries all combination of buying fruits and choose the minimum one. As their are overlapping subproblems we use dp. So to buy the $(x+1)^{th}$ fruit with optimized approach, he'll have to optimize x^{th} fruit approach first.

→ Space complexity: O(n) that matrix dp to store the calculated values.

Q6 pseudo code:

```
INVEST(d, n)
    let I[1...10] and R[1...10] be new tables
    for K = 10 downto 1
        q = 1
        for i = 1 to n
            if r[i, K] > r[q, K] // i now holds the investment which looks best for a
                given year.
                q = i
        if R[K+1] + dr_{I[K+1]K} - f[1] > R[K+1]
    + dr[q, K] - f[2] // If revenue is greater
    when money isn't moved.
            R[K] = R[K+1] + dr_{I[K+1]K} - f[1]
            I[K] = I[K+1]
        else
            R[K] = R[K+1] + dr[q, K] - f[2]
            I[K] = q
    return I as an optimal strategy with
        Return R[1]
```

Teacher's Signature

Algorithm:

The algorithm works as follows: We build tables I and R of size 10 such that I[i] tells which investment should be made (with all money) in year i, and R[i] gives the total return on the investment strategy in years i through 10.

Proof of correctness:

Without loss of Generality, suppose that there exists an optimal solution S which involves investing d_1 dollers into investment K and d_2 dollars into investment m in year 1. Further, suppose in this optimal solution, you don't move your money for the first j years. If $r_{K1} + r_{K2} + \ldots + r_{Kj} > r_{m_1} + r_{m_2} + \ldots + r_{mj}$ then we can perform the usual cut-and-paste maneuver and instead invest $d_1 + d_2$ dollars into investment K for j years. Keeping all other investments the same, this results in a strategy which is at least as profitable as S, but has reduced the no. of different investments in a given span of years by 1. Continuing in this way, we can reduce the optimal strategy to consist of only a single investment each year.

Q8 Pseudo code:

```
procedure shortestPalindrome(s)
    n ← length of s
    reverse string s
    s_new ← s + "#" + rev;
    n_new ← length of s_new
    f ← vector of size n_new initialised
        with 'zero'.
for i = 1 to n_new - 1
{
    t ← f[i-1]
    while (t > 0 and s_new[i] != s_new[t])
        t ← f[t-1];
    if s_new[i] == s_new[t]
        ++t
    f[i] ← t
}
return reverse of substring i.e.,
    rev.substr(0, n - f[n_new - 1]) + s
```

Algorithm:

→ ~~We~~ I'm using the KMP lookup table generation
→ Create new-S as S + "#" + reverse(s) and use the string in the lookup-generation algorithm.
- The "#" in the middle is required, since without the #, the 2 strings could mix with each other, producing wrong answer. For example, take the string "aaaa". Had we not inserted "#" in the middle, the new string would be "aaaaaaaa" and the largest prefix size would be 7 corresponding to "aaaaaaa" which ~~wo~~ would be obviously wrong. Hence, a delimeter is required at the middle.
- Return reversed string after the largest palindrome from beginning length (give by n - f[n-new-1]) + original string s.

KMP Overview:

It's a string matching algorithm that runs in O(n+m) times, where n & m are sizes of the text and string to be searched respectively. The key component of KMP is the failure function lookup table, say f(s). The purpose of the lookup table is to store the length of the proper prefix of the string $b_1 b_2 \ldots b_s$ that's also

a suffix of $b_1 b_2 \ldots b_s$. This table is important because if I'm trying to match a text string for $b_1 b_2 \ldots b_n$, and I've matched the first s positions, but when we fail, then the value of lookup table for s is the longest prefix of $b_1 b_2 \ldots b_n$ that could possibly match the text string upto the point we're at. Thus, we don't need to start all over again, and can resume searching from the matching prefix.

The algorithm to generate the lookup table is

```
f(0) = 0
for ( i = 1 ; i < n ; i++ )
{
    t = f(i-1)
    while ( t > 0 && b[i] != b[t] )
        t = f(t-1)
    if ( b[i] == b[t] )
        t++
    f(i) = t
}
```

- Here, I first set $f(0) = 0$ since, no proper prefix is available.
- Next, iterate over i from 1 to n-1:
  - Set $t = f(i-1)$
  - while $t > 0$ and char at i doesn't match the char at t position, set $t = f(t)$, which essentially means that I've problem matching and must consider a shorter prefix, which will be $b_{f(t-1)}$, until I find a match or t becomes 0.
  - If $b_i == b_t$, add 1 to t.
  - set $f(i) = t$.

→ Complexity Analysis.

- Time complexity : O(n)
  - In every iteration of the inner while loop, t decreases until it reaches 0 or until it matches. After that, it's incremented by 1. Therefore, in the worst case, t can only be decreased up to n times and increased up to n times.
  - Hence, the algorithm is linear with maximum (2*n)*2 iterations.
- Space Complexity : O(n). Additional space for the reverse string and the concatenated string.

PAGE NO.
DATE:

q7

Pseudo code:

```
procedure func(s)
{
    str ← s + "#" + t
    t ← str
```

pseudo code:

```
main function.
{   string S, T, s;
    s = S + "#" + T;
    ans ← 0
    index ← 0
    for i = 0 to length of s.
    {
        temp ← func(s)
        if (ans < temp)
        { index ← i
          ans ← temp
        }
    }
for function (s)
  { n ← s.size()
    prefix-function[n]
    for i=1, j to n
    { j ← prefix-function[i-1]
      while (j and s[i] != s[j])
        j = prefix-function
```

Teacher's Signature

```
    if(s[i] == s[j]) j ← j+1.
      prefix-function[i] ← j
  }
  mx ← 0
  for (i=n-1 to s[i] != '#')
      mx = max(mx, prefix-function[i])
  return mx
}
```

Algorithm Analysis:

Longest common Substring using KMP.

String S, String P

$|S| = n$, $|P| = m$ $\quad n > m$

$P = P_1 P_2 P_3 \dots P_m$

→ T.C = $O(n * m)$

```
for i in range(m) : ——— I
String np = p[i:n] = P_i P_{i+1} ... P_n // Creating
                                  Substring of P[i:n]
ans ← max(find_kmp(s, p), ans)
find_kmp(string s, string p) ——— II O(n)
{ // trivial KMP algorithm
we keep track of length of prefix matched,
and update our answer.
  i=0, j=0 cons=0 Calculate_lps(p)
  while(i<n)
      if s[i] == p[j]
          i++; j++;
```

```
        cons ← max (cons, i)
    elif (i<n . s[i] != p[j])
        if j > 0
            j = lps[j-1]
        else
            i++
    return cons.
}
```

→ Time Complexity

find_kmp() return in O(m+n) time

∴ n > m ∴ O(n) (Calculate lps [work in O(m) time)

→ loop I, runs in O(m) time.

→ So finally, we've O(n * m) time complexity

→ Space complexity ∴ = O(m)