CSE 4810: Algorithm Engineering Lab
Lab - 4

Md Farhan Ishmam (180041120)

Group - CSE 1A

March 19, 2023

Task 1

(a)

In the rod-cutting problem, the original rod of size n can be cut at n places resulting in 2^{n-1} possible ways to divide the rod. We need to cut the rod at all n places and find the maximum value to get the optimal value from cutting the rod. This can be formulated as follows as seen in [1]:

$$r_n = max(p_n, r_1 + r_{n-1}, r_2 + r_{n-2}, ..., r_{n-1} + r_1)$$

From the above equation of revenue from a rod of size n, we can say rod-cutting has an **optimal substructure** i.e. we can get the optimal value of size n from optimal values of size $n-1, n-2, ...$ For a rod of length n, we will cut at all possible places i which can take value from 1 to n. For each, cut we will again find the optimal value which will hence, be a sub-problem for the original rod of length n. In this way, the rod-cutting problem can be formulated using sub-problems.

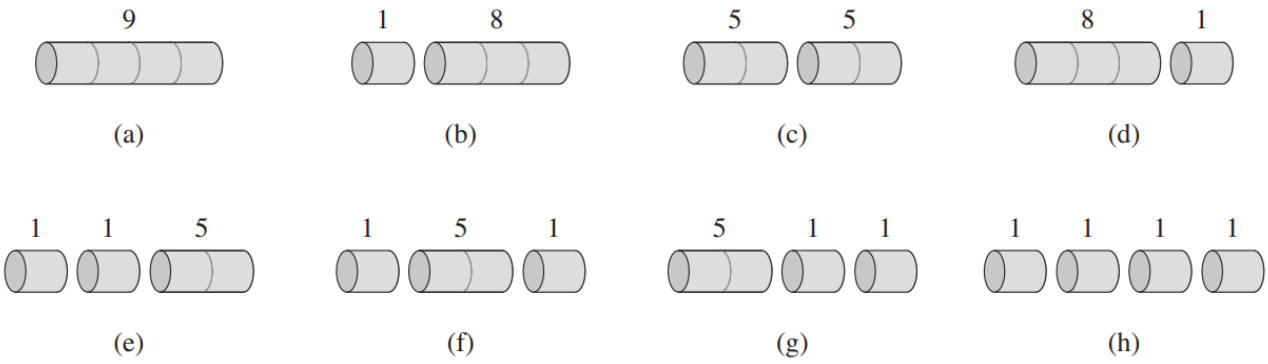

Figure 1: All the possible ways to cut the rod of size n. Diagram is taken from [1]

(b)

Yes, the sub-sub-problems overlap. In figure-1, for the rod of length 4, there are 8 possible ways to cut. If we cut according to (b), then we have a rod of length 1 and length 3 which will be our sub-problems. Again a rod of length 3 can be cut in 4 ways. One such cut will give us rods of length 1 and length 2 which will be our sub-sub-problem. Again, if we cut according to (d), we get sub-problems of lengths 1 and 3. The sub-problem of 3 will have sub-sub-problems of lengths 1 and 2. Hence, these sub-problems of lengths 1 and 2 will overlap if we cut according to (b) and (d). Thus, rod-cutting will have overlapping sub-sub-problems.

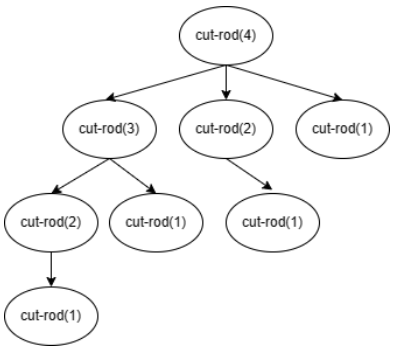

Figure 2: Recursion tree of the $cut - rod()$ function for a rod of length 4 shows that sub-sub-problems are overlapping. $cut - rod(1)$ is the overlapping sub-sub problem in this case.

(c)

Yes, we can find an optimal solution for the rod-cutting problem. As the rod-cutting problem has an optimal sub-structure, by solving the sub-problems and combining the results, we can get the optimal solution. The following algorithm can be used to generate the optimal solution for the problem instance given in the question:

```
cut-rod(p, n):
    r = array of size n
    for every element i in r:
        i = -1
    return cut-rod-func(p,n,r)

cut-rod-func(p,n,r):
    if r[n] >= 0:     #revenue is stored in the memory
        return r[n]
    if n==0:
        r[n] = 0
        return 0
    else:
        for i = 1 to n:
            r[n] = max(r[n], p[i] + cut-rod-func(p,n-i,r))
    return r[n]
```

Length, i	1	2	3	4	5	6	7	8	9	10
Revenue, r_i	1	5	8	10	14	16	19	22	24	28

Table 1: Dynamic programming table for a rod of length 10.

For the rod of length 10, we take the maximum revenue generated from a cut. In this case, cutting in the middle and getting two rods of length 5 will give us the maximum revenue of $14 + 14 = 28$. For the rod of length 5, keeping it uncut gives us the maximum revenue of 14 as given in the table.

(d)

We prove the given greedy strategy is **wrong** by providing a counter-example.

Length, i	1	2	3
Price, p_i	0	4	7
Density, p_i/i	0	2	2.33

Table 2: The length, associated price, and density for a sample rod-cutting problem.

For the given table, we find the optimal value for a rod of length 4. According to the greedy strategy, cutting the rod at length 3 should be optimal as the rod of length 3 has the highest density. So, the total revenue generated will be $7 + 0 = 7$ where 7 will be revenue from length 3 and 0 will be revenue from the remaining length 1. However, we can generate a revenue of 8 by cutting the rod in the middle which gives us two rods of length 2 and thus, generating the revenue of $4 + 4 = 8$. Hence, the greedy strategy fails for the given instance.

Task 2

(a)

The problem exhibits optimal substructure and overlapping subproblems. Hence, it is possible to engineer a dynamic programming solution to solve this problem. The equation for the sub-problems is given below:

```
coin_change(n, sum) = max(coin_change(n-1, sum), coin_change(n, sum - A[n]))
```

Here, the coin change value determines the number of ways the summation *sum* can be obtained for n coins each with a value defined in the A array. The coin change value is basically the maximum of two values. The first value $coinChange(n-1, sum)$ comes if the last coin isn't picked resulting in the number of coins decreasing by 1 and the sum remaining the same. The second value $coinChange(n, sum - A[n])$ comes if the last coin is picked. Since the coin can be picked again, the number of coins stays the same and the sum decreases by the value of the last coin which can be obtained by accessing the A array. Max is taken as the problem asks us to find whether it is possible or not to get the summation using the given set of coins. If the problem asked us to find the number of ways then we can simply take the summation instead of the max.

(b)

My DP table will be a 2D table $table[i][j]$ where i or rows will indicate the index of the last coin and j or columns will indicate the remaining summation value.

(c)

The code to generate the DP table is given below:

```
coin_change(A, sum):
    n = length(A)
    // initialize the 2D table with zeros
    table = new 2D array with n rows and sum+1 columns
    for i from 0 to n-1:
        // base case: when sum is 0, there is only 1 way to make change (using no coins)
        table[i][0] = 1
        for j from 1 to sum:
            // if the current coin is larger than the current sum, skip it
            if A[i] > j:
                table[i][j] = table[i-1][j]
            else:
                // use the dynamic programming formula to calculate the number of ways to
                    make change
                table[i][j] = max(table[i-1][j], table[i][j-A[i]])
    // return the value in the bottom-right corner of the table
    return table[n-1][sum]
```

The generated DP table is given below:

0	1	2	3	$\cdots$	249	250	251	252
2	1	0	1	$\cdots$	0	1	0	1
5	1	0	1	$\cdots$	0	1	0	1
10	1	0	1	$\cdots$	1	1	1	1
50	1	0	1	$\cdots$	1	1	1	1
100	1	0	1	$\cdots$	1	1	1	1
200	1	0	1	$\cdots$	1	1	1	1
501	1	0	1	$\cdots$	0	1	0	1
997	1	0	1	$\cdots$	0	1	0	1

Table 3: DP table to find the value of 252 in coin change

For the given table the bottom-right value of the table indicates whether the value 252 can be obtained or not. As the bottom-right value is 1, it means we can obtain 252 from the given set of coins.

(d)

Yes, using the same sets of coins, the value of 10 can be constructed. The table is given below:

0	0	1	2	3	4	5	6	7	8	9	10
2	1	0	1	0	1	0	1	0	1	0	1
5	1	0	1	0	1	1	1	1	1	1	1
10	1	0	1	0	1	1	1	1	1	1	1
50	1	0	1	0	1	1	1	1	1	1	1
100	1	0	1	0	1	1	1	1	1	1	1
200	1	0	1	0	1	1	1	1	1	1	1
501	1	0	1	0	1	1	1	1	1	1	1
997	1	0	1	0	1	1	1	1	1	1	1

Table 4: DP table to find the value of 10 in coin change

(e)

The asymptotic complexity of the coin change problem is $O(mn)$ where n is the number of coins and m is the amount to be made.

(f)

The DP table is given below:

0	1	2	3	$\cdots$	98	99	100	101
1	1	1	1	$\cdots$	1	1	1	1
2	1	1	1	$\cdots$	1	1	1	1
5	1	1	1	$\cdots$	1	1	1	1
10	1	1	1	$\cdots$	1	1	1	1
50	1	1	1	$\cdots$	1	1	1	1
100	1	1	1	$\cdots$	1	1	1	1
200	1	1	1	$\cdots$	1	1	1	1
500	1	1	1	$\cdots$	1	1	1	1
1000	1	1	1	$\cdots$	1	1	1	1

Table 5: DP table for the given set of coins

To construct 37, we take 3*10 + 1*5 + 1*2. To construct 72, we take 1*50 + 2*20 + 1*2. To construct 101, we take 1*100 + 1*1. To construct 7, we take 1*5 + 1*2.

(g)

For problem (f), to make 37, the biggest value coin will be 10. After taking the coin of value 10, 3 times then we can take the coin of value 5 once and the coin of value 2 once. Hence, the approach will work. The same approach will work in generating 72, 101, and 7 for the problem (f). However, for the problem (c), this will not work for making an amount like 251. If we take 200 and 50 once then we are left with 1, making the achievable amount impossible. However, if we can the coin 5, 5 times, the coin 200, 1 time, the coin 10 two times, and the coin 2 three times then we achieve then the sum of 251. Hence, 251 should produce the result, yes, but using the greedy approach doesn't produce a yes. The reason is that for problem (f), we have coins of values 1 and 2. In the worst-case scenario, if an amount of 1 remains then the coin of value 1 can be picked once. However, in problem (c), we do not have such a coin of value 1. Hence, we need to chose values more carefully instead of picking the largest one using the greedy approach.

Task 3

(a)

We reverse the string and then find the longest common substring between the original and the reversed string. The algorithm is given below:

```
longestCommonPalindrome(s):
    s_rev = s.reverse()
    m = len(s)
    n = len(s_rev)
    z = 0
    out = []
    for i = 1 to m:
        for j = 1 to n:
            if s[i] = s_rev[j]:
                if i=1 or j=1:
                    dp[i][j] = 1
                else:
                    dp[i][j] = dp[i-1][j-1] + 1
                if dp[i][j] > z:
                    z = dp[i][j]
                    out = s[i-z+1 ... i]
                else if dp[i][j] = z
                    out = out + s[i-z+1 ... i]
            else:
                dp[i][j] = 0
    return out
```

(b)

The table for the second input is given below:

		m	o	s	a	d	a	s	n	o
	0	0	0	0	0	0	0	0	0	0
o	0	0	1	0	0	0	0	0	0	1
n	0	0	0	0	0	0	0	1	0	0
s	0	0	0	1	0	0	0	1	0	0
a	0	0	0	0	2	0	1	0	0	0
d	0	0	0	0	0	3	0	0	0	0
a	0	0	0	0	1	0	4	0	0	0
s	0	0	0	1	0	0	0	5	0	0
o	0	0	1	0	0	0	0	0	0	1
m	0	1	0	0	0	0	0	0	0	0

Table 6: DP table for the second input

Task 4

The problem exhibits optimal substructure as the value at a current cell depends on two values from the previous row. The equation for a particular value can be written as:

$$p[i][j] = p[i-1][j] + p[i-1][j-1]$$

The following dynamic programming algorithm can be used to solve the value for the r-th row:

```
dp = table of size n,n #n indicates the number of rows of the pascal triangle
for every element i in dp:
    i = -1

pascal-cell(m,n):
    if dp[m][n]>0:
        return dp[m][n]
    if m==n | m=0 | n=0:
        return 1
    else
        return pascal(m-1,n) + pascal(m-1,n-1)

pascal-row(m):
    out = []
    for i=0 to m:
        out.append(pascal(m,i))
    return out
```

References

[1] Thomas H Cormen, Charles E Leiserson, Ronald L Rivest, and Clifford Stein. *Introduction to algorithms*. MIT press, 2022.