DAA HW 6

Pranav Tushar Pradhan N18401944 pp3051@nyu.edu

Udit Milind Gavasane N16545381 umg215@nyu.edu

Rohan Mahesh Gore N19332535 rmg9725@nyu.edu

Saavy Singh N16140420 ss19170@nyu.edu

Sashank Badri Narayan N14628839 sb10192@nyu.edu

Page No.
Date.

Q1) Cost function-
r(n, k) be maximum price obtained from cutting rod of length n.
We can only use lengths upto k ($1 \leq k \leq n$)

Recursion-
n: remaining length of rod to cut
k: maximum length piece to currently consider

2 choices { ① Dont use length k
② Use length k exactly once if $n \geq k$

$$\therefore r(n,k) = \max \begin{cases} r(n, k-1), & \text{① dont use k} \\ p_k + r(n-k, k-1) & \text{②} \end{cases}$$

[if $n \geq k$]

Base cases:
$r(n,0) = 0$ for all n as no lengths are available hence revenue is 0
$r(0,k) = 0$ for all k as no rod length hence revenue is 0

Pseudo code:

```
def rod_cutting(n, prices):
    #create 2D DP table
    dp = [ ]
    for i in range(n+1):
        row = [0 for _ in range(n+1)]
        dp.append(row)
# fill table bottom up
    for i in range(1, n+1):                    # length of rod
        for k in range(1, n+1):                # max length to consider
            # Don't use length k
```

```
dp[i][k] = dp[i][k-1]
# use length k if possible
if k <= i
    dp[i][k] = max(dp[i][k], prices[k-1] +
                   dp[i-k][k-1])
return dp[n][n]
```

Time & space complexity
$O(n^2)$
- a table of size n×n
- filling the table and each cell take $O(1)$ time

03) Cost function -
Let $V_c(v, u)$ be the minimum weight of a vertex cover for the subtree rooted at v,.
$u=0 \longrightarrow$ v not in cover
$u=1 \longrightarrow$ v in cover

Recursion -
For any vertex v,
$V_c(v, 0) = \sum V_c(u, 1)$ for all children u of v
$V_c(v, 1) = w(v) + \sum \min(V_c(u,0), V_c(u,1))$ for all children u of v

Base cases -
For leaf node v,
$V_c(v, 0) = 0$
$V_c(v, 1) = w(v)$

Pseudo code:

```
def min_weight_cover(tree, weights, root):
    dp = { }          # dictionary to store (node, state)
                      solutions
```

```
def solve(v, parent, must_include):
    if (v, must_include) in dp:
        return dp[(v, must_include)]
    children = [u for u in tree[v] if u != parent]
    if not children:    # leaf node
        dp[(v, must_include)] = weights[v] if
                                must_include else 0
        return dp[(v, must_include)]
    if must_include:
        # if v is included, children can be either
          includer or not
        result = weights[v]
        for child in children:
            result += min(solve(child, v, 0), solve
                          (child, v, 1))
    else:
        # if v is not included, all chidrey must be
          included
        result = 0
        for child in children:
            result += solve(child, v, 1)
    dp[(v, must_include)] = result
    return result
# final result is minimum of including or not including root
return min(solve(root, None, 0), solve(root, None, 1))
```

Time & space complexity

O(|V|)

- Each vertex is processed exactly twice (include/not incly
- Each edge is considered exactly once
- Total number of subproblems is O(|V|)
- We store 2 values for each vertex in the dp table

Question 2

To find longest palindromic subsequence (LPS) of seq $X = \{x_1, x_2 \cdots x_n\}$

Approach:

① Reverse the sequence.

The problem transforms into finding the longest common subsequence (LCS) of X and reverse(X).

Eg A C T A
A T C A
] ⟶ A C A or A T A

Y
these are palindromes

② We will use DP[i][j] table to store the length of LCS of X[1,...i] and Y[1...j].

If X[i] = Y[j],

DP[i][j] = DP[i-1][j-1] + 1

The character is a part of LCS so we add 1 to the previous length.

If X[i] ≠ Y[j]

DP[i][j] = max(DP[i-1][j] , DP[i][j-1])

(Taking max. LCS length excluding one character from either X or Y)

Base case:

DP[i][0] = 0 and DP[0][j] = 0

③ Reconstruct the LPS using back tracking.

DP[n, n] = length of LPS

If X[i] = X[j]

add X[i] to LPS and move diagonally → (i-1, j-1)

If DP[i-1][j] > DP[i][j-1]

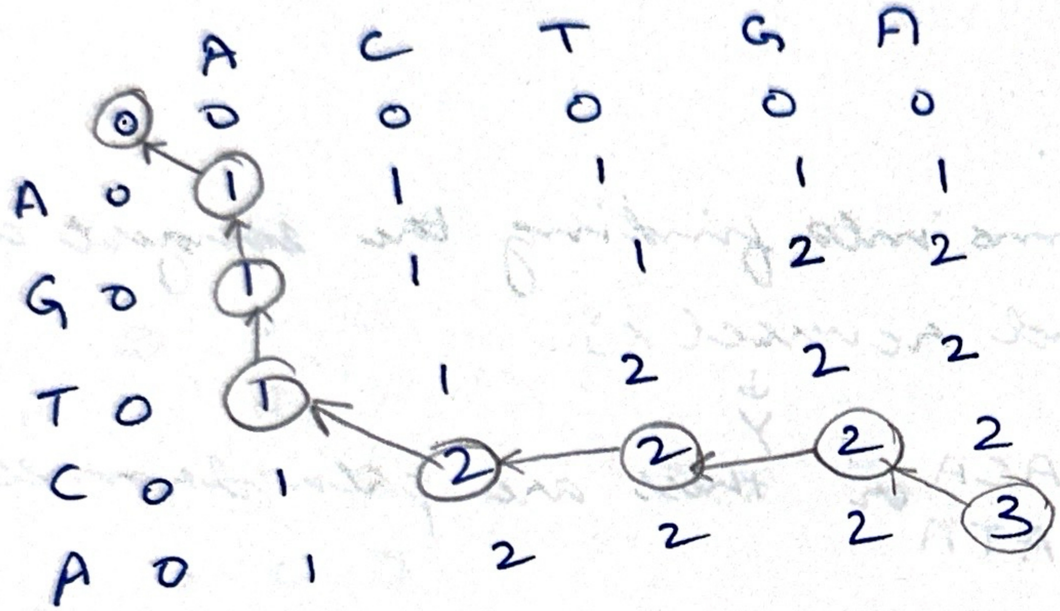
Move → (i-1, j)

else

Move → (i, j-1)

Example

$X = ACTGA \quad Y = AGTCA$



$dp[5][5] = 3 \quad : A$

$dp[4][2] = 2 \quad : C$

~~$dp[3][1] = 1 \quad : T$~~

$dp[1][1] = 1 \quad : A$

$\rightarrow ACA$

Time complexity:

① Filling $n \times n$ table acquires $O(n^2)$ time

② Table entry computation: $O(1)$

Overall Time complexity

$\hookrightarrow O(n^2)$

Assignment 6

Q4: Given a rectangular piece of cloth X and Y.

Sol^n: dimensions.

For each product $i \in \{1, 2, \ldots, n\}$, the dimensions are a_i, b_i s.t the final selling price is c_i.

a_i & b_i are positive integers

$c_i > 0$.

Dynamic Programming Approach.

Let $P(x,y)$ be the maximum selling price of all the cloth pieces that can be made from a cloth of dimensions $x \times y$.

$\therefore P(x,y) = \sum$ (maximum selling price)

We need to build a recurrence relation to compute $P(x,y)$ considering all the cases.

Case 1: Using a cloth of $a_i \times b_i$ in final solution

$\therefore P(x,y) = \max(P(x,y), c_i + P(x-a_i, y-b_i))$

This will make sure that we choose the max value of selling price with or without including $a_i \times b_i$ piece with price c_i in the final solution.

Case 2: We cut the cloth horizontally at position r.

∴ Dimensions of top part $r \times y$

—"— bottom part $(x-r) \times y$

∴ Max price in this case is:

$$P(x,y) = \max\left(P(x,y), P(r,y) + P(x-r,y)\right)$$

Case 3: Vertically cut at position t.

∴ Dimensions of right part $t \times x$.

Dimensions of left part $x \times (y-t)$.

Max price in this case is:

$$P(x,y) = \max\left(P(x,y), P(x,t) + P(x, y-t)\right)$$

Case 4: If the cloth is too small to produce any product then

$$P(x,y) = 0$$

as x and y are not positive integers in length.

Now that we have the cases ready, we need to build the DP table P of size $(x+1) \times (y+1)$ to store the results for all possible cloth dimensions.

Here will start with small pieces (1×1) and build large sizes $P(x, y)$

Finally the value $P(x, y)$ will give us the maximum selling price.

→ Algorithm: Define $P[X][Y]$

Let n = no. of products

for x in $(1, X+1)$:

for y in $(1, Y+1)$:

for a, b, c in d:

if $x \geq a$ & $y \geq b$

$P[x][y] = \max(P[x][y], c + P[x-a][y-b])$

for k in $(1, x)$:

$P[x][y] = \max(P[x][y], P[k][y] + P[x-k][y])$

for t in $(1, y)$:

$P[x][y] = \max(P[x][y], P[x][t] + P[x][y-t])$

Finally, we return $P[X][Y]$

→ Lets analyse the time complexity of the algorithm.

1) There are $X \times Y$ subproblems as the DP table contains $X \times Y$ entries.

For each sub-problem we compute $P(x,y)$ for all $1 \le x \le X$ and $1 \le y \le Y$.

Hence, time complexity to calculate the subproblems is $O(X \times Y)$.

2) Calculations per subproblem.

a.] $\forall$ n product we check if it can produce $a_i \times b_i$ with c_i price.

∴ This step is $O(n)$

b.] Horizontal cuts step takes $O(n)$ as we check for cuts $i = 1, 2, \ldots, n-1$

c.] Vertical cuts step takes $O(y)$ as we check for cuts $t = 1, 2, \ldots, y-1$

∴ Time complexity per subproblem = $O(n+xy)$.

Now, in the worst case n and y can be as large as X and Y respectively.

$\therefore$ T.C. per subproblem $= O(n+X+Y)$.

$\therefore$ Total T.C. $= O(X \times Y) \times O(n+X+Y)$

$= O(X \times Y \times (n+X+Y))$

for large inputs n dominate $(X+Y)$ as no. of pieces are usually larger than no. of cutting choices

$\therefore$ Total time complexity comes down : $O(n \times X \times Y)$

Thus the worst case time complexity is $O(n \times Y)$.