CSCI 570: Analysis of Algorithm HW 3

Name: Aagam Shah
USC ID: 5420023430

1. Suppose you are responsible for organizing and determining the results of an election. An election has a winner if one person gets at least half of the votes. For an election with n voters and k candidates, design an optimal algorithm to decide if the election has a winner and finds that winner. Analyze the time complexity of your algorithm.

Ans 1)

Approach: We use the approach of divide and conquer where we will divide the subproblem into multiple small problems and find the maximum number of votes that candidate has gotten if it's the maximum. We return the candidate's number. If the number of votes is < n/2 then we will print "No Winner in the Election" or if any candidate gets vote > n/2 then we will print his number.

Input: An Array containing the voters votes of size n, The number of candidates k
Output: The Candidate who won the election

Algorithm:

1. Create 2 variables left and right and set their values
    a. Left = 0
    b. Right = voters.size() // Size of Voters Array
2. Call the function findElectionWinner(voters, left, right)
    a. Winner = findElectionWinner(voters,left,right)
    b. If(winner == -1)
        i. Print(" No Winner of the Election"
    c. Else
        i. Print(Winner). // Candidate's Number

Function findElectionWinner(int voters [] , int left, int right)

1. Consider when we have only 1 vote than that candidate is the winner
    a. If(left == right)
        i. Return voters[left]
2. Initialize a variable mid and set its value
    a. Mid = (left + right) / 2;
3. Recursively find the winner in the left subarray
    a. LeftWinner = findElectionWinner(voters, left, mid)
4. Recursively find the winner in right subarray
    a. RightWinner = findElectionWinner(voters, mid+1 , right)
5. If both the winners are the same return any one
    a. If(LeftWinner == RightWinner)

        i. Return RightWinner
6. Find the number of votes leftWinner has achieved
    a. LeftWinnerVotes = CountVote(votes,leftWinner,left,right)
7. Find the number of votes RightWinner has achieved
    a. RightWinnerVotes = CountVote(votes,rightWinner,left,right)
8. If(LeftWinnerVotes > voters.size / 2)
    a. Return LeftWinner
9. If(RightWinnerVotes > voters.size / 2)
    a. Return RightWinner
10. Else
    a. Return -1

Function countVote(int [] votes, int candidate, int left, int right)

1. Initialize a variable count = 0
2. For (int i =left ; i <= right; i++)
    a. If(votes[i] == candidate)
        i. Count++
3. Return count

Time Complexity: Using this approach, we are going to divide the array into multiple sub array and calculate the number of times the winner of subarray has won taking n * log n. Log n in breaking the array in between and n to count the votes of the candidate. This approach should take time complexity $O(nlogn)$time complexity to divide the array and finding the maximum of each subarray. Thus, the net time complexity is $O(nlogn)$).

Space Complexity: Since we are just using some variables to find the max and the count thus the Space Complexity is O(1).

2. Alice has learned the sqrt function in her math class last week. She thinks that the method to compute sqrt function can be improved at least for perfect squared numbers. Please help her to find an optimal algorithm to find perfect squared numbers and their squared root.

Ans 2)
We use Divide and Conquer approach to solve this problem. Where we input the numbers from 1 to n in an array and then using divide and conquer, we will find the mid and check if mid*mid = number if not then we will reduce the search to half numbers. Using this we can find the square root of an element in O(logn) time complexity.

Input: Integer n whose square root we need to find where n is a perfect square of a number
Output: Integer sqrt which is the square root of integer n

Algorithm:

1. Initialize 3 variables to find the square root of a number
    a. Left = 1
    b. Right = n
2. We recursively check if the mid*mid == n which means the mid number is the sqrt of n, if not then we reduce the array to half and check in the half of the array.
    a. While (left <= right)
        i. Mid = (left + right) / 2
        ii. If (mid * mid == n)
            1. Return mid
        iii. Else
            1. If (mid * mid > n)
                a. Left = mid
            2. Else
                a. Right = mid - 1

Runtime Complexity: In this Algorithm we will split the array into 2 parts and find the mid and if mid*mid is not equal to the number required we reduce the array to 2 parts making the input size / 2 and we keep on doing this till we don't get the solution. Here we can split it at max log n number of times, therefore the time complexity: $O(\log n)$

Runtime Complexity: $O(\log n)$

3. Alice has recently started working on the Los Angeles beautification team. She found a street and is proposing to paint a mural on their walls as her first project. The buildings in this street are consecutive, have the same width but have different heights. She is planning to paint the largest rectangular mural on these walls. The chance of approving her proposal depends on the size of the mural and is higher for the larger murals. Suppose n is the number of buildings in this street and she has the list of heights of buildings. Propose a divide and conquer algorithm to help her find the size of the largest possible mural and analyze the complexity of your algorithm. The picture below shows a part of that street in her proposal and two possible location of the mural:

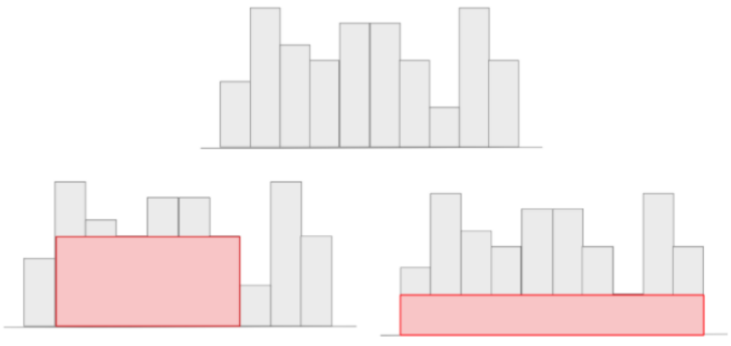


Figure 1: The example in Q9

Ans3) Approach: We will divide the problem into 3 subparts where we can say that the maximum area can lie in the left half, right half or in the span of the 2 subparts in the middle. We will do recursive call for the left and right half and then solve the span part using an iterative approach where we keep looking from the middle to left and right side of the buildings and include those whose height is greater than the other side trying to maximize the area between the set of buildings.

Input: Array of heights of building, the start index and ending index
Output: Area of Maximum largest mural

Algorithm:

function Area_Of_Mural (int [] height, int start, int end)
{

1. If we only have 1 element in the height array, then max area is the height itself Area = Height * Width = height[start] * 1 (Since width = 1)
    a. If (start == end)
        i. Return height[start]
2. We calculate the middle of the height array.
    a. Mid = (start + end) /2
3. Initialize Area variable to be set as 0
    a. Area = 0
4. We recursively find the area of Start to Middle
    a. Area_Left = Max (height, start, mid)
5. We recursively find the area of Middle + 1 to end
    a. Area_Right = Max (height, mid + 1, end)
6. We also calculate the area generated by considering the span / area from middle buildings.
    a. Area_span = SpanArea(height[], start, middle, end))
7. Return calculated max (Area_left, Area_right, Area_span)

}

Function SpanArea(int height[], int start, int middle, int end)
{

1. Initialize Variable i and j to expand the span in left and right setting i as mid and j as mid +1
    a. I = mid
    b. J = mid + 1
2. Initialize Area to be 0 for span
    a. Area = 0
3. Getting the minimum height of the 2 elements currently in the span i.e height[i] & height[j] to find their area.
    a. H = Min (height [i], height[j])

4. We keep on going in right and left adding elements to find the maximum permissible area using the buildings height.
    a. While (i >= start and j <=end)
        i. We find the minimum height since this will limit the area calculated.
            1. H = Min (h, Min (height [i], height[j])
        ii. Calculate area based on the location of mural and their height i.e., the (rightmost building - leftmost building) * minimum height.
            1. Area = Max (Area, (j - i +1) * h)
            2. If we reach the end of left buildings, then we just check the right side of the buildings from mid
                a. If (i == start)
                    i. J = j +1
            3. If we reach the end of right buildings, then we just check the left side of the buildings from mid.
                a. If(J == end)
                    i. I = i -1
            4. If we have buildings on both side of the mid then we find the one with max height and include in our span
                a. If(height[i-1] > height[j+1])
                    i. I = i -1
                b. Else
                    i. J = j + 1
5. Return the calculated max area.

Time Complexity: Here we can see that the recurrence relation generated will be like

$$T(n) = 2\,T\left(\frac{n}{2}\right) + \;\; O(n)$$

Where we are dividing the sub problem into 2 similar parts finding the max area in the left and right subarray and then we use O(n) to find the area of the span since we are looking from middle to the 2 ends of the array where at max we can traverse the entire array of n elements. On solving the above recurrence equation we know that the time complexity

$$\boldsymbol{T(n) = O(n * \log n)}$$

4. Solve the following recurrence using the master theorem.

a) $T(n) = 8T(\frac{n}{2}) + n^2 log(n)$

b) $T(n) = 4T(\frac{n}{2}) + n!$

c) $T(n) = 2T(\frac{n}{4}) + \sqrt{n}$

d) $T(2^n) = 4T(2^{n-1}) + 2^{\frac{n}{2}}$

Ans 4)

1) $\boldsymbol{T(n) = 8\,T\left(\frac{n}{2}\right) + n^2 \log n}$

Comparing the equation to $T(n) = a\,T\left(\frac{n}{b}\right) + f(n)$

Here, a = 8, b = 2 and $f(n) = n^2 \log n$

We can apply master's theorem here as we know that $a \geq 1$, $b > 1$, $f(n)\ is\ a\ positive\ function$

Finding the value of $c = \log_b a$

$$c = \log_b a = \log_2 8 = 3$$

Number of leaves = $O(n^c) = O(n^3)$

Applying **Case 1 of master's Theorem**:

$$if\ \ f(n) = O(n^{c-\epsilon})\ then\ T(n) = \ \theta(n^c)$$

$$Since:\ n^3 > \ n^2 \log n$$

$$\therefore \boldsymbol{T(n) = \ \theta(n^3)}$$

2) $\boldsymbol{T(n) = 4\,T\left(\frac{n}{2}\right) + n!}$

Comparing the equation to $T(n) = a\,T\left(\frac{n}{b}\right) + f(n)$

Here, a = 4, b = 2 and $f(n) = n!$

We can apply master's theorem here as we know that $a \geq 1$, $b > 1$, $f(n)\ is\ a\ positive\ function$

Finding the value of $c = \log_b a$

$$c = \log_b a = \log_2 4 = 2$$

Number of leaves = $O(n^c) = O(n^2)$

Applying **Case 3 of master's Theorem**:

$$if\ \ f(n) = O(n^{c+\epsilon})\ then\ T(n) = \ \theta(f(n))$$

$$Since:\ n! > \ n^2$$

$$\therefore \boldsymbol{T(n) = \ \theta(n!)}$$

3) $\boldsymbol{T(n) = 2\,T\left(\frac{n}{4}\right) + \sqrt{n}}$

Comparing the equation to $T(n) = a\,T\left(\frac{n}{b}\right) + f(n)$

Here, a = 2, b = 4 and $f(n) = \sqrt{n}$

We can apply master's theorem here as we know that $a \geq 1$, $b > 1$, $f(n)\ is\ a\ positive\ function$

Finding the value of $c = \log_b a$

$$c = \log_b a = \log_4 2$$

Applying change of base theorem for log

$$\log_b a = \frac{\log_c a}{\log_c b} \qquad \log_4 2 = \frac{\log_2 2}{\log_2 4} = \frac{1}{2}$$

Number of leaves = $O(n^c) = O(n^{1/2}) = O(\sqrt{n})$

Applying **Case 2 of master's Theorem**:

$$if\ \ f(n) = O(n^c \log^k n)\ then\ T(n) = \ \theta(n^c \log^{k+1} n)$$

$$\therefore \boldsymbol{T(n) = \ \theta\left(\sqrt{n}\log n\right)}$$

4) $\boldsymbol{T(2^n) = 4\,T(2^{n-1}) + 2^{n/2}}$

Consider that $\boldsymbol{k = \ 2^n}$ we can substitute value of k in the equation of T(n) to make it T(k)

$$\boldsymbol{T(2^n) = 4\,T\left(\frac{2^n}{2}\right) + (2^n)^{\frac{1}{2}}}$$

$$\boldsymbol{T(k) = 4T\left(\frac{k}{2}\right) + \sqrt{k}}$$

Solving the newly formed equation using master's theorem

Comparing the equation to $T(k) = a\,T\left(\frac{k}{b}\right) + f(k)$

Here, a = 4, b = 2 and $f(k) = \sqrt{n}$

We can apply master's theorem here as we know that $a \geq 1$, $b > 1$, $f(n)\ is\ a\ positive\ function$

Finding the value of $c = \log_b a$

$$c = \log_b a = \log_2 4 = 2$$

Number of leaves = $O(n^c) = O(n^2)$

Applying **Case 1 of master's Theorem**:

$$if\ \ f(n) = O(n^{c-\epsilon})\ then\ T(n) = \ \theta(n^c)$$

$$Since:\ k^2 > \sqrt{k}$$

$$\therefore \boldsymbol{T(k) = \ \theta(k^2)}$$

Resubstituting the value of $\boldsymbol{k = \ 2^n}$ we get

$$\boldsymbol{T(n) = \ \theta(2^{2n})}$$

5. Suppose you have a rod of length N, and you want to cut up the rod and sell the pieces in a way that maximizes the total amount of money you get. A piece of length i is worth p_i dollars. Devise a Dynamic Programming algorithm to determine the maximum amount of money you can get by cutting the rod strategically and selling the cut pieces.
   a. Define (in plain English) subproblems to be solved.
   b. Write a recurrence relation for the subproblems
   c. Using the recurrence formula in part b, write pseudocode to find the solution.
   d. Make sure you specify
      i. base cases and their values
      ii. where the final answer can be found
   e. What is the complexity of your solution?

Ans5)

Subproblem: Let Opt[k,x] be the max amount achievable using a cutting a rod of size x into k pieces.

Let's consider the choices we can take:

1. Piece k is Not Selected. If the piece is not selected then we can assure that it won't be chosen again in the solution making the number of items reduce by 1 making it k-1
   a. $X_k = 0$
      i. $Opt[k, x] = Opt[k-1, x]$

2. Piece k is selected. If the piece is selected we can select the item again as well so the k will remain k

a. $X_k = 1$
   i. $Opt[k, x] = Opt[k, x - w_k]$

2) **Recurrence Formula**

Consider having k = no of Pieces by cutting the rod and x = Size of the rod

$$Opt[k, x] = Max\ (\ Opt[k-1, x], V_k + \ Opt[k, x - w_k])$$

Where V_k is the profit, we achieve by selecting the kth piece into our final rod.

3) **Base Case**

i. If the No of piece or the total size of rod is 0

$$Opt[k, x] = 0\ , if\ k = 0\ or\ x = 0$$

4) **Pseudo - Code**

```
Int rodCutting (int N, int l[], int p[], int n)
{
    Int Opt[n+1][W+1];
    for (k = 0;  k <= n; k++)
        {
            for ( x=0; x<= n, x++)
                {
                    if (k == 0 || x == 0)
                        Opt[k][x] = 0
                    Else
                        Opt[k][x] = Max(Opt[k-1][x] , p[k] +
Opt[k][x - l[k-1]])
                }
        }
    return Opt[n][W];
}
```

5) Solution: The solution will be found at Opt[n][W].

6) Complexity:

Time Complexity = Table size * The Work we do
= $O(n * W) * O(1)$
= $O(n * W)$ -- Not in Input Size

Input Size = $O(n * \log W)$

Actual Runtime = $O\left(n * 2^{input\ size\ of\ w}\right)$

It has Exponential Runtime Complexity.

Space Complexity = Size of the Table = $O(n * W)$
It has Polynomial Runtime Complexity.

6. Tommy and Bruiny are playing a turn-based game together. This game involves N marbles placed in a row. The marbles are numbered 0 to N-1 from the left to the right. Marble i has a positive value m_i. On each player's turn, they can remove either the leftmost marble or the rightmost marble from the row and receive points equal to the sum of the remaining marbles' values in the row. The winner is the one with the higher score when there are no marbles left to remove.

   Tommy always goes first in this game. Both players wish to maximize their score by the end of the game.

Assuming that both players play optimally, devise a Dynamic Programming algorithm to return the difference in Tommy and Bruiny's score once the game has been played for any given input. Your algorithm must run in $O(N^2)$ time.

a. Define (in plain English) subproblems to be solved.
b. Write a recurrence relation for the subproblems
c. Using the recurrence formula in part b, write pseudocode to find the solution.
d. Make sure you specify
   i. base cases and their values
   ii. where the final answer can be found
e. What is the complexity of your solution?

Ans 6)

Subproblem:
Let OPT[i,j] be the maximum difference in score of Tommy and Bruiny achievable when the marbles from i to j are remaining and both of them play optimally. We use the concept of presum prefix array which can help us get the sum of the remaining elements of array in O(1)

We have two choices:

If the player selects the ith marble from the array, then the difference is given by.

$$Opt[i,j] = prefix\_arr[j+1] - prefix\_arr[i+1] - OPT[i+1][j]$$

If the player selects the jth marble from the array, then the difference is given by.

$$Opt[i,j] = prefix\_arr[j] - prefix\_arr[i] - OPT[i][j-1]$$

Recurrence relation:

We can define the recurrence relation for the subproblem as follows:

$$OPT[i,j] = max\,(prefix\_arr[j+1] - prefix\_arr[i+1] - OPT[i+1,j],\\ prefix\_arr[j] - prefix\_arr[i] - OPT[i,j-1])$$

Pseudo Code:

```
int maxDifferecefromGame(int marbles[], int n)
{
   int prefix_arr[] = new int[n];
   int OPT[][] = new int[n][n];

   // Intialize the OPT array with all 0 intial value
   for(int i = 0; i < n; i++) {
      for(int j = 0; j < n; j++) {
         OPT[i][j] = 0;
      }
   }

  // arr = {1,2,3,4}
  // Prefix_arr = { 0 , 1, 3, 6, 10} - Adding extra 0 at the
start
   prefix_arr[0] = 0;
   for(int i = 1; i < n + 1; i++) {
      prefix_arr[i] = prefix_arr[i - 1] + prefix_arr[i - 1];
   }

   // Dynamic Programming to solve the problem
   for(int i = n - 2; i >= 0; i--)
       {
            for(int j = i + 1; j <= n-1 ; j++)
                {
                OPT[i][j] = Math.max(prefix_arr[j + 1] -
prefix_arr[i + 1] - OPT[i + 1][j], prefix_arr[j] -
                          prefix_arr[i] - OPT[i][j - 1]);
            }
        }
```

```
    return OPT[0][n - 1]
}
```

Base Case:

1. If i > j then this will terminate the recursion

$$Opt[i][j] = 0$$

Solution Location:
The location of the final answer is in the last cell of the first row of the opt table, that is opt[0][n-1].

Complexity:

Time complexity = $O(\text{Table size} \times \text{Work done})$
= $O(\text{n} * \text{n} \times 1) = O(n^2)$

We fill the table which takes O (1). Additionally after calculating the prefix_arr we can find the sum in O(1) time

Space complexity = Table size = $O(n^2)$

7. The Trojan Band consisting of n band members hurries to lined up in a straight line to start a march. But since band members are not positioned by height the line is looking very messy. The band leader wants to pull out the minimum number of band members that will cause the line to be in a formation (the remaining band members will stay in the line in the same order as they were before). The formation refers to an ordering of band members such that their heights satisfy $\text{r}_1 < \text{r}_2 < \ldots < \text{r}_i > \ldots > \text{r}_n$, where $1 \leq \text{i} \leq \text{n}$.

   For example, if the heights (in inches) are given as

$$R = (67, 65, 72, 75, 73, 70, 70, 68)$$

   the minimum number of band members to pull out to make a formation will be 2, resulting in the following formation:

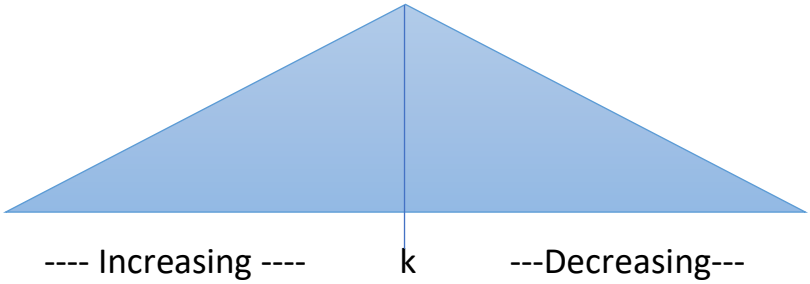
$$(67, 72, 75, 73, 70, 68)$$

   Give an algorithm to find the minimum number of band members to pull out of the line.

   Note: you do not need to find the actual formation. You only need to find the minimum number of band members to pull out of the line, but you need to find this minimum number in $\text{O}(\text{n}^2)$ time.

Ans 7)

Subproblem: Let Opt1[i] of size n represents the length of the longest increasing subsequence that ends at index i. and Opt2[j] of size n represents the length of the longest decreasing subsequence that starts at index j. Then we find the Maximum length of the increasing subsequence that ends at i and longest decreasing subsequence that starts from j+1, thus finding out the number of band members to be removed.



Recurrence relation:

We can define the recurrence relation for the subproblem as follows:

a. If height of band member at index j < Height of band member at index i and Opt1[j] >= Opt1[i]

$$Opt1[i] = Max(\ Opt1[i], Opt1[j] + 1)$$

b. If height of band member at index j < Height of band member at index i and Opt2[j] >= Opt2[i]

$$Opt2[i] = Max(\ Opt2[i], Opt2[j] + 1)$$

Pseudo Code :

```
function getMinMembers(int[] heights, int n){
    int[] Opt1 = new int[n];
    int[] Opt2 = new int[n];.
    // Intialize the Longest Increasing/Decresing
Subsequence Arrays by 1
    for int(i = 0;i < n; i++)
    {
        Opt1[i] = 1;
        Opt2[i] = 1;
    }
    // Find the Longest Incresing Subsequence from index i
    int max_length = 1;
    for (int i = 1; i < n; i++) {
        for (int j = 0; j < i; j++) {
            if (heights[j] < heights[i] && Opt1[j] >=
Opt1[i]) {
```

```
                Opt1[i] = Opt1[j] + 1;
            }
        }
        max_len = Math.max(max_len, dp1[i]);
    }
    // Find the Longest Decresing Subsequence from index i
    for (int i = n - 2; i >= 0; i--)
        {
        for (int j = i + 1; j < n; j++)
            {
            if (heights[j] < heights[i] && Opt2[j] >=
Opt2[i])
                {
                Opt2[i] = Opt2[j] + 1;
                }
            }
        }
        // Find the Maximum of Incresing SubSequence ending
at index k and
        // Decresing SubSequence starting at index k, '-1'
since the kth element is considered 2 times
    for(int k=0;k<n;k++)
        {
                max_len = Math.max(max_len, Opt1[k] +
Opt2[k] - 1);
        }
    }
    return n - max_len;
}
```

Base Case:
If we only have 1 member in the subsequence i.e, the length of the subsequence possible is of length 1

$$Opt1[i] = 1$$

$$Opt2[j] = 1$$

Solution Location:

The Final Answer is $n - max\_subsequence\_length$ where $max\_subsequence\_length = Opt1[k] + Opt2[k] - 1$ from 0 to n

Complexity:

Time Complexity: The Time complexity of this algorithm is $O(n^2)$,where n is the length of the input sequence. This is because we use **two** nested loops to fill out the dynamic

programming arrays Opt1 and Opt2, Making it $O(n^2)$, . Then we use another loop to iterate through all possible values of k and compute the maximum length of a subsequence that ends at index k and starts at index k, which takes O(n) time. Therefore, the overall time complexity is $O(n^2)$

Space Complexity: The Space complexity of this algorithm is $O(n)$, Since we are only using 3 arrays of size n which makes the total space complexity - O(n)

8. From the lecture, you know how to use dynamic programming to solve the 0-1 knapsack problem where each item is unique and only one of each kind is available. Now let us consider knapsack problem where you have in- finitely many items of each kind. Namely, there are n different types of items.

   All the items of the same type i have equal size w_i and value v_i. You are offered infinitely many items of each type. Design a dynamic programming algorithm to compute the optimal value you can get from a knapsack with capacity W.

   a. Define (in plain English) subproblems to be solved.
   b. Write a recurrence relation for the subproblems
   c. Using the recurrence formula in part b, write pseudocode to find the solution.
   d. Make sure you specify
      i. base cases and their values
      ii. where the final answer can be found
   e. What is the complexity of your solution?

Ans 8)

1) Statement: Let Opt[k,x] be the max value achievable using a knapsack of capacity x and k items.

Let's consider the choices we can take:

3. Item k is Not Selected by our Knapsack.
   a. $X_k = 0$
      i. $Opt[k, x] = Opt[k-1, x]$

4. Item k is selected by our Knapsack.
   a. $X_k = 1$
      i. $Opt[k, x] = Opt[k, x - w_k]$

2) Recurrence Formula

Consider having k = no of items and x = weight of knapsack

$$Opt[k,x] = Max\ (\ Opt[k-1,x], V_k + \ Opt[k, x - w_k])$$

Where V_k is the profit, we achieve by selecting the kth item into our knapsack.

3) Base Case

ii. If the No of items or the Weight of Knapsack is 0

$$Opt[k,x] = 0\ , if\ k = 0\ or\ x = 0$$

iii. If the weight of item selected is more than the capacity of knapsack, then we leave that item.

$$Opt[k,x] = Opt[k-1,x]\ , w_k > x$$

4) **Pseudo - Code**

```
Int knapsack (int W, int w[], int v[], int n)
{
    Int Opt[n+1][W+1];
    for (k = 0;  k <= n; k++)
        {
            for ( x=0; x<= W, x++)
                {
                    if (k == 0 || x == 0)
                        Opt[k][x] = 0
                    If (w[x] > x)
                        Opt[k][x] = Opt[k-1] [x];
                    Else
                        Opt[k][x] = Max(Opt[k-1][x] , v[k] +
Opt[k][x - x[k-1]])
                }
        }
    return Opt[n][W];
}
```

5) **Solution**: The solution will be found at Opt[n][W].

6) **Complexity**:

Time Complexity = Table size * The Work we do

$= O(n * W) * O(1)$
$= O(n * W)$ -- Not in Input Size

Input Size = $O(n * \log W)$

Actual Runtime = $O\left(n * 2^{input\ size\ of\ w}\right)$

It has Exponential Runtime Complexity.

Space Complexity = Size of the Table = $O(n * W)$
It has Polynomial Runtime Complexity.

9. Given n balloons, indexed from 0 to $n-1$. Each balloon is painted with a number on it represented by array nums. You are asked to burst all the balloons. If you burst balloon i you will get $nums[left] * nums[i] * nums[right]$ coins. Here left and right are adjacent indices of i. After the burst, the left and right then becomes adjacent. You may assume

$nums[-1] = nums[n] = 1$ and they are not real therefore you can not burst them. Design an dynamic programming algorithm to find the maximum coins you can collect by bursting the balloons wisely.

Here is an example. If you have the $nums$ array equals $[3, 1, 5, 8]$. The optimal solution would be 167, where you burst balloons in the order of 1, 5 3 and 8. The left balloons after each step is:

$$[3,\ 1,\ 5,\ 8] \rightarrow [3,\ 5,\ 8] \rightarrow [3,\ 8] \rightarrow [8] \rightarrow []$$

And the coins you get equals:

$$167 = 3 * 1 * 5 + 3 * 5 * 8 + 1 * 3 * 8 + 1 * 8 * 1.$$

a. Define (in plain English) subproblems to be solved.
b. Write a recurrence relation for the subproblems
c. Using the recurrence formula in part b, write pseudocode to find the solution.
d. Make sure you specify
   i. base cases and their values
   ii. where the final answer can be found
e. What is the complexity of your solution?

Ans 9)

1. **Sub problem**:

Let DP [i, j] be the maximum coins achievable by bursting balloons with left index as i and right index as j of the subarray.

2. **Recurrence Relationship:**

$$dp[i][j] = \max(dp[i][j], arr[i-1] * arr[k] * arr[j+1] + dp[i][k-1] + dp[k+1][j]); \quad i \le k \le j$$

Base Case:

- **I.** If we have only 1 single ballon then the cost will be equal to arr[1]
  - **a.** If(n == 1)
    - **i.** Return arr[1]
- **II.** If we have no balloon to burst
  - a. DP[i][i+1] = 0

3. **Pseudo Code:**

```
function maxCoins(int n, int arr[])
    {
        if(n == 1)
        {
           Return (1*arr[1]*1);
        }
        // This will auto initialize the dp[][] with all 0 in
Java
        int dp[][] = new int[n+2][n+2];

        for(int l=1;l<=n;l++)
            {
                for(int i=1;i<=n-l+1;i++)
                    {
                        int j=i+l-1;
                        for(int k=i;k<=j;k++)
                        {
                            int cost=arr[i-1]
*arr[k]*arr[j+1]+dp[i][k-1]+dp[k+1][j];
                            dp[i][j]=max(dp[i][j], cost);
                        }
                    }
            }
        return dp[1][n];
    }
```

4. **Solution:**

a. Solution can be found in the matrix at location dp[0][n] i.e., the 1st row's and last column since we compute the elements in the array diagonally and the solution will be the end of the matrix

5. Complexity:

a. Time Complexity

i. The Time complexity of the solution is $O(n^3)$ Considering the time complexity is $O(Table\ Size * Work\ Done\ )$ here since we are using a n*n matrix the table size becomes n^2 and we need to find the max score which takes n time thus making the runtime $O(Table\ Size * Work\ Done\ ) = O(n^2 * n) = O(n^3)$

b. Space Complexity

i. The Space complexity is $O(n^2)$ since we are using a table of size n*n to compute all the possible combinations.