Analysis of Algorithms Homework 3

USC ID: 3725520208
Name: Anne Sai Venkata Naga Saketh
Email: annes@usc.edu

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.**

Solution:

It is given that we are given an array of **'n'** voters, which has the details of to which candidate each voter has voted, and an election only has a winner if the number of votes received by a candidate is more than $\boldsymbol{n/2}$.

The optimal algorithm to decide if the election has a winner is by using a divide and conquer algorithm.

Step 1: Consider an array of voters; it has the details of which respective candidate each voter has voted for.

Step 2: (Divide Step): Divide the input array until we get a single element in each of the divided sub-arrays' (**Similar to merge sort**).

Step 3: (Conquer Step): Merging the individual sub-arrays (elements) until we form a single array at the end. **Compute/Check the winner (the candidate who has secured greater than or equal to the $\frac{array\_length}{2}$ votes)** in each of the individual sub-arrays. Assuming that, two candidates receive an equal number of votes, then the as per the question any one of them can be considered as the winner.

> **Case 1:** If the winner is the **same** in both the sub-arrays that are being merged, then the final **winner remains the same**.
>
> **Case 2:** If the winner is **different** in both the sub-arrays that are being merged, then count the votes received by each of the candidates in the merged array and then compute/decide the winner based on the formula: the candidate who has secured **greater than or equal to** $\frac{array\_length}{2}$ votes shall be the winner.

Step 4: By the end of execution, when all the individual sub-arrays are merged into a single array, the winner of the merged array shall be the winner of the election.

The time complexity of the above algorithm can be found using the below recurrence relation.

Here we are dividing the given input problem into **2** subproblems of input size $\boldsymbol{n/2}$.

After dividing the given problem into sub-problems, while joining the sub-arrays, we shall proceed to find the winner by comparing the number of votes received by the candidates in each of the sub-arrays. This operation takes $O(n)$ time.

Therefore, $\boldsymbol{T(n) = 2T(n/2) + O(n)}$**,** Here a = 2, b = 2.

Applying, $\boldsymbol{c = log_b^a}$ we get c = $\boldsymbol{log_2^2 = 1}$
Hence, $n^c = n^1$

Since $\theta(n^c) = \theta\big(f(n)\big)$ this falls under case 2 of the master theorem.

Therefore, the time complexity of the above recurrence relation is: $n^c log^{k+1} n$ where k = 0. i.e., $\boldsymbol{TC = \theta(nlog\ (n))}$

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.

Solution:

We can infer that a square root of a number shall always be less than the number **n** or equal to the number in the case of 1.

Here, we need to find if the given input number '**n**' is a perfect square and the respective square root of the number '**n.'**

Algorithm to find if the number 'n' is a perfect square and the square root of the number is.

Step 1: Consider a sorted array of integer numbers from 1 to n.

i.e., the array consists of 1, 2, 3……n.

Step 2: Initialize the following variables,

$low = 0,$

$high = size\ of\ the\ array - 1$, i.e., n-1

$$middle = \frac{low + high}{2}$$

Step 3: While the $low <= high$, do the following.

Case 1: If the **middle** element's square is equal to the given input number **'n,'** then 'n' is a perfect square, and its square root is **middle**. *End the execution of the algorithm***.**

Case 2: If the square of the middle element is greater than the input number 'n,' then update the value of $high = middle - 1$

Call the same algorithm with the updated low, high, and input array values.

Case 3: If the square of the middle element is less than the input number 'n,' then update the value of $low = middle + 1$

Call the same algorithm with the updated low, high, and input array values.

Step 4: Repeat Step 3. if by the end of the while loop, the square root of the number cannot be determined (i.e., the algorithm terminates unsuccessfully), then the given input number **'n'** is not a perfect square and does not have an integer square root value.

The time complexity of the above algorithm can be found using the below recurrence relation.

Here we are dividing the given input problem into **1** subproblem of input size $\boldsymbol{n/2}$

After dividing the given problem into sub-problems, we compare the square of the middle element with the given input number 'n' and proceed further with it, which takes $O(1)$ time.

Therefore, $\boldsymbol{T(n) = T(n/2) + O(1)}$**.** Here a = 1, b = 2.

Applying, $\boldsymbol{c = log_b^a}$ we get c = $\boldsymbol{log_2^1 = 0}$
Hence, $n^c = n^0$

Since $\theta(n^c) = \theta(f(n))$, this falls under case 2 of the master theorem.

Therefore, the time complexity of the above recurrence relation is: $n^c log^{k+1} n$ where k = 0 and c = 0. i.e., $\boldsymbol{TC = \theta(log\ (n))}$

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 locations of the mural:

Solution:

It is given that; we are given an input of buildings in a street of the same width but different heights.

It is said that the chance of approval for the mural will be higher if its height is greater.

An Optimal Divide and Conquer algorithm to find the largest possible mural is as follows:

Step 1: Consider that the input is an array of buildings with the same width but different heights.

Step 2: Divide Step: Split/Divide the array until we get to sub-arrays of size 1**(like merge sort)**

Step 3: Initialize four variables, height = 0, area = 0, fin_area = 0 and fin_height = 0

Step 4: Conquer Step: While joining each sub-array to form a complete array, find the largest possible mural using the steps below.

> **Step 4.1:** Initialize the variable **start1** and **end1** to the beginning of the first element in the subarray. Calculate the area of the mural, by using the height of the minimum building * width of the building * number of buildings and store the value of the computed area in **area1** variable.
>
> **Step 4.2:** Now, merge the subarray with the next element, and then calculate the area of the rectangle by finding the minimum height of the buildings and then multiply it with the width of the number of buildings to find the area and store it in the variable **area2**
>
> **Step 4.3:** Now compare the area1 and area2 variables and then proceed accordingly,
>
> If area1 <= area2, then add the element to the current subarray and repeat the **Step 4**, and then store the value of area1 to **area** variable.
>
> **Step 4.4:** If fin_area < area then fin_area = area and then start=start1, end=end1
>
> **Step 4.5:** If area1 > area2, in that case then update the end1 variable to the index of the last element subarray, and then store the value of area1 to **area** variable. Then go to **Step 5**.

Step 5: if area1 > area2, in that case we must stop the merging of the next subarrays and then start again from the new subarray which we have tried to merge. And then repeat the process from **Step 4**.

Step 6: Select the sub-arrays (a subset of buildings) that can give the maximum possible height and area so that the mural can have better chances of approval. i.e., from the start to end variable indices of the given array.

The time complexity of the above algorithm can be found using the below recurrence relation.

Here we are dividing the given input problem into 2 subproblems of input size $n/2$, where 'n' is the number of buildings in the street.

After dividing the given problem into sub-problems, we shall combine each sub-array and find the largest possible mural. This operation shall take $O(n)$ time.

Therefore, $T(n) = 2T(n/2) + O(n)$, Here a = 2, b = 2.

Applying, $\boldsymbol{c = log_b^a}$ we get c = $\boldsymbol{log_2^2 = 1}$
Hence, $n^c = n^1$

Since $\theta(n^c) = \theta\big(f(n)\big)$, this falls under case 2 of the master theorem.

Therefore, the time complexity of the above recurrence relation is: $n^c log^{k+1} n$ where k = 0. i.e., $\boldsymbol{TC = \theta(nlog\,(n))}$

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

Solution:

The definition of master theorem is if $T(n) = a \cdot T(n/b) + f(n)$
Case 1: if $f(n)=O(n^{c-\varepsilon})$, then $T(n)=\Theta(n^c)$
Case 2: if $f(n)=\Theta(n^c\log^k n)$, then $T(n)=\Theta(n^c\log^{k+1} n)$
Case 3: if $f(n)=\Omega(n^{c+\varepsilon})$, then $T(n) = \Theta(f(n))$

where $\boldsymbol{c} = \boldsymbol{log_b^a}$.

Using the above definition of the master theorem to solve the following problems.

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

Here as per the given question, a = 8, b = 2
Applying, $\boldsymbol{c = log_b^a}$ we get c = $\boldsymbol{log_2^8 = 3}$
Hence, $n^c = n^3$.

Since $\theta(n^c) > \theta(f(n))$, this falls under case 1 of the master theorem.

So, the time complexity for $\boldsymbol{T(n) = 8T(\frac{n}{2}) + n^2 log(n)}$ is $\boldsymbol{\theta(n^3)}$

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

Here as per the given question, a = 4, b = 2
Applying, $\boldsymbol{c = log_b^a}$ we get c = $\boldsymbol{log_2^4 = 3}$
Hence, $n^c = n^2$.

Since $\theta(n^c) < \theta(n!)$, because $n(n-1)(n-2) \ldots..1$, here the degree of 'n' shall be more than 'n^c'. So, this falls under the case 3 of master theorem.

So, the time complexity for $\boldsymbol{T(n) = 4T(\frac{n}{2}) + n!}$ is $\boldsymbol{\theta(f(n)), i.e., \theta(n!)}$

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

Here as per the given question, a = 2, b = 4, k= 0.
Applying, $\boldsymbol{c = log_b^a}$ we get c = $\boldsymbol{log_4^2 = \frac{log_2^2}{log_2^4} = \frac{1}{2}}$
Hence, $n^c = n^{\frac{1}{2}}, i.e., \sqrt{n}$.

Since $\theta(n^c) = \theta(f(n))$, this falls under the case 2 of master theorem.

So, the time complexity for $\boldsymbol{T(n) = 2T(\frac{n}{4}) + \sqrt{n}}$ is $\boldsymbol{\theta(f(n). log^{k+1}(n))}, here\ k = 0$
$$therefore, TC\ is\ \boldsymbol{\theta(\sqrt{n} log(n))}$$

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

Assume $\boldsymbol{2^n = M}$**, therefore the equation** $\boldsymbol{T(M) = 4T(M/2) + \sqrt{M}}$

Here as per the given question, a = 4, b = 2
Applying, $\boldsymbol{c = log_b^a}$ we get c = $\boldsymbol{log_2^4 = 2}$
Hence, $M^c = M^2$.

Since $\theta(M^c) > \theta(f(M))$, this falls under case 1 of the master theorem.

So, the time complexity for $\boldsymbol{T(2^n) = 4T(2^{n-1}) + 2^{\frac{n}{2}}}$ is $\boldsymbol{\theta(M^2)}$.

Now, substituting **M** back into the above equation, we get the $\boldsymbol{\theta(M^2) = \theta((2^n)^2) = \theta(2^{2n})}$.

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.**

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

Solution:

Here we are given a rod of length **'N'** that must be cut into multiple pieces, and each piece **'i'** is worth **'p_i'** dollars, and we must maximize the profit or the amount we earn by selling the pieces of the rod that we cut and sell.

In Dynamic Programming, the subproblems shall be overlapping; the greater the overlap between the subproblems, the better our algorithm/solution works.

A. Subproblem Definition:

Here, the **'L'** to be the length of the rod and X to be the length of the rod that shall be cut to make a $\mathbf{p_x}$ money by selling it.

Let OPT[L] be the equation that helps us to maximize our profit.

We need to select the length of the rod, such that it maximizes the money that we can earn by cutting the rod.

B. Recurrence Relation for the sub-problems:

The recurrence relation shall be: OPT[L] = *Max*(p_x + OPT[L-X]), where L is the length of the given rod, and X is the length of the rod that was cut giving us a profit of p_x.

C. Pseudo-Code:

Here, we need to use the tabulation method instead of memorization so that our algorithm shall perform better asymptomatically.

```
int max_profit(int N, int P[]) {

int OPT[N+1];

for(i=0; i<=N; i++) {

        if(i == 0){

                OPT[i] = 0;

        }

        else{

        for(j=0; j<=i; j++){

                OPT[i] = Max(OPT[i], P[j] + OPT[i-j])

        }

}

return OPT[N];

}
```

D. i. Base Cases:

As we have seen above that our function is OPT(), hence the base cases shall be as follows.

OPT[L] = 0 when L = 0

OPT[L] = p_L when L < X

ii. Final Answer:

The final answer can be found in the table that we have constructed OPT[N].

E. Time complexity of the Dynamic Programming Algorithm:

From the pseudo code, we can infer that there are two 'for' loops running from 1 to N, that are required to fill the values of the table**(Tabulation method).** Hence the Time complexity of this piece of code shall be $\boldsymbol{O}(\boldsymbol{N}^2)$.

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.**
      1. **base cases and their values**
      2. **where the final answer can be found**
   E. **What is the complexity of your solution?**

Solution:

Here we are given **N** number of marbles, and a marble 'i' has a positive value of $\mathbf{m_i}$. Whenever a marble mi is picked, the sum of the values of the remaining marbles is the score that the player receives.

Tommy and Bruiny take turns and can pick either the first or the last marble to maximize their score. In the game, it is given that Tommy always plays first.

A. Defining Subproblems to be solved using Dynamic Programming.

Let OPT[X,Y] be the given array of marbles such that m_x, m_{x+1},…….m_{y-1},m_y.

Here our choices can be,

- **Case 1:** As Tommy goes first, he can pick the leftmost marble, and in the next turn, Bruiny can also pick the leftmost marble.

i.e., if Tommy picks left most marble m_x, then his score earned shall be the sum of points of the remaining marbles m_{x+1},.......m_{y-1},m_y and in the next turn, if Bruiny picks the leftmost marble again, then his score shall be the sum of points of the remaining marbles m_{x+2},.......m_{y-1},m_y

Calculating the score difference = m_{x+1}

- **Case 2:** As Tommy goes first, he can pick the leftmost marble, and in the next turn, Bruiny can pick the rightmost marble.

  i.e., if Tommy picks left most marble m_x, then his score earned shall be the sum of points of the remaining marbles m_{x+1},.......m_{y-1},m_y and in the next turn, if Bruiny picks the right most marble, then his score shall be the sum of points of the remaining marbles m_{x+1},.......m_{y-1}.

  Calculating the score difference = m_y

- **Case 3:** As Tommy goes first, he can pick the rightmost marble, and in the next turn, Bruiny can also pick the rightmost marble.

  i.e., if Tommy picks the rightmost marble m_y, then his score earned shall be the sum of points of the remaining marbles m_x,m_{x+1},.......$m_{y-1,}$ and in the next turn, if Bruiny also picks the rightmost marble, then his score shall be sum of points of the remaining marbles m_x ,m_{x+1},.......m_{y-2}.

  Calculating the score difference = m_{y-1}

- **Case 4:** As Tommy goes first, he can pick the rightmost marble, and in the next turn, Bruiny can pick the leftmost marble.

  i.e., if Tommy picks the rightmost marble m_y, then his score earned shall be the sum of points of the remaining marbles m_x,m_{x+1}.......$m_{y-1,}$ and in the next turn, if Bruiny also picks the rightmost marble, then his score shall be the sum of points of the remaining marbles m_{x+1},.......m_{y-1}

  Calculating the score difference = m_x

B. Recurrence relationship:

The recurrence relation of the above dynamic programming shall be as follows.

OPT[X,Y] = *Max* (m_{x+1} + OPT[X+2,Y], m_y + OPT[X+1,Y-1], m_{y-1} + OPT[X,Y-2], m_x + OPT[X+1,Y-1])

C. Pseudo Code:

```
int MaxScore( int M[], int N){
        int OPT[N+1][N+1];
        for(int i = 0; i <= N; i ++){
                for(j = 0; j <= N; j++){
                        if (i == 0 | | j == 0){
                                OPT[i][j] = 0;
                        }
                        else if ( i == j){
                                OPT[i][j] = 0
                        }
                        else if (i > j){
                                OPT[i][j] = 0
                        }
                        else{
        OPT[i][j] =
Math.Max( M[i+1] + OPT[i+2][j], M[j] + OPT[i+1][j-1], M[j-1] + OPT[i][j-2], M[i] + OPT[i+1][j-1] )
                        }
                }
        }
        return OTP[0][N]; //The maximum difference value of the scores can be found here.
}
```

D. i. Base Cases:

- If X = 0 or Y = 0, then OPT[X,Y] = 0

- If X = Y, then OPT[X,Y] = 0, since there is only one marble, and when Tommy removes it his score shall be 0 and Bruiny has no marbles to pick, assuming his score to also be '0', the total difference shall be **'0'.**
- If X > Y, then OPT[X,Y] = 0

ii. Where the final answer can be found:

The final answer can be found in the array OPT[0][N]

E. Time Complexity of the above algorithm:

The above pseudo code is using two nested for loops that are running from 0 to N making the time complexity of the function $\boldsymbol{O(N^2)}$**.** Here, N is the size of the input given (number of marbles).

So, the time complexity shall be $\boldsymbol{O(N^2)}$. It is a polynomial in N of degree 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 $r_1 < r_2 < \ldots < r_i > \ldots > r_n$, where $1 \le i \le n$.

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 $O(n^2)$ 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?

Solution:

Given that there are **'n'** band members who are unorganized in a line, and the band leader must pull the **minimum** number of people to make the required formation such that, all the members before a person **'r_i'** should be in increasing order, and all the members after **'r_i'** should be in decreasing order.

We need to find the element **'r_i'** such that the number of pullouts from the left subarray(increasing subarray) and the number of pullouts from the right subarray(decreasing subarray) shall be the minimum possible.

A. Subproblem definition:

The subproblem to be solved is to find the minimum number of pullouts from the left subarray and the minimum number of pullouts from the right subarray for a given element $\mathbf{k_r}$.

Our final aim is to find the minimum possible number of pull outs such that the resulting array or the resulting set of heights shall be in the given formation.

B. Recurrence Relation for the above dynamic programming problem:

The recurrence relation to find the minimum number of pullouts is as follows.

1. For Left elements that are in the given order:

For an element $\boldsymbol{k_r}$ considered, let left_order() be the function that helps us determine the maximum number of elements that are in the given order from 0 to i-1.

if $\mathbf{k_{r-1} < k_r}$ then left_order(k_r) = left_order(k_{r-1})

if $\mathbf{k_{r-1} > k_r}$ then left_order(k_r) = *Max*(1 + left_order(k_{r-1}))

2. For Right elements that are in the given order:

For an element $\boldsymbol{k_r}$ considered, let right_order() be the function that helps us determine the maximum number of elements that are in the given order from i+1 to n-1.

if $\mathbf{k_r > k_{r+1}}$ then right_order(k_r) = right_order(k_{r+1})

if $\mathbf{k_r < k_{r+1}}$ then right_order(k_r) = *Max(*1 + right_ order(k_{r+1}))

Then the total minimum number of pullouts shall be $total\ number\ of\ element - left\_order(k_r) + right\_order(k_r) + 1$

C. Pseudo Code for the above dynamic programming problem using tabulation method:

```
int minimum_removals(int heights[]) {
        int len= sizeof(heights);

        int left_order[];

        int right_order[];

        int result=0;

\\ initialize all the elements in the left and the right array to be ‘0’

       for( i = 0; I < len; i++){

      left_order[i] = 0;

      right_order[i] = 0;

      }

        for (int i=1; i < len; i++)

        {

          for (int j=0; j < i; j++)

          {

              if (heights[i] > heights[j])

              left_ order[i]=Math.max(left_order[i], left_order[j]+1);

          }

        }

        for(int i= len-2; i>=0; i--)

        {

          for(int j= len-1; j>i; j--)

          {

               if (heights[i] > heights[j])

               right_ order[i]=Math.max(right_ order[i], right_ order[j]+1);
        }
      }
    for (int i=1; i<len; i++)
    {
        If(left_order[i] != 0 && right_order[i] != 0)
        result = Math.max(result, left_order[i] + right_order[i]);
    }
```

//for a given element ‘i’ this shall help to find the maximum number of elements that are in the given sequence.

```
    return len-(result+1);
}
```

D. i. Base Cases:

If L = 0, then left_order(L) = 0 since for the left most element there shall not be elements less than it to compare.

If R = Length-1, then right_order(R) = 0 since for the right most element, there shall not be elements less than it to compare.

ii. Where the final answer can be found: The final answer is the difference of the length of the array and the result value plus ‘1’.

E. Time Complexity:

The time complexity of the above pseudo code shall be $\boldsymbol{O}(\boldsymbol{n}^2)$, since there are two nested for loops running from 1 to n.

Since, **‘n’** is the given number of people in the band, that needs to be organized in a form, it represents the input size. So, the final time complexity of the above code shall be $\boldsymbol{O}(\boldsymbol{n}^2)$.

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 infinitely 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?**

Solution:

Here we are given a knapsack of capacity **'W',** which we need to fill up with **'i'** different items of different weights '$\mathbf{w_i}$' of the value of '$\mathbf{v_i}$' and make the most amount of profit.

In Dynamic Programming, the subproblems shall overlap. The more significant the overlap between the subproblems, our algorithm/solution works better.

A. Subproblem Definition:

Let OPT[K, X] be the maximum possible profit that is achievable using a knapsack of capacity X, where 0 <= X <= W, using k number of items that are given, where 0 < k <= n

Here our choices can be,

- **Case 1:** If we select **'kth item'** during the first iteration, then we get OPT[K, X] = V_k + OPT[K, X-W_k], we can choose the same weight again in the next iteration, so the number of items does not change, but the available capacity of the knapsack decreases.
- **Case 2:** If we do not select **'kth item'** during the first iteration, then we get OPT[K, X] = OPT[K, X]; we can choose the item **'kth item'** in later iterations, so neither the number of items available nor the knapsack capacity will change.

B. Recurrence relation of the above dynamic problem shall be:

OPT[K, X] = *Max*(V_k + OPT[K, X-W_k], OPT[K, X]), where V_k is the value associated with the 'kth' object.

C. Pseudo-Code based on the above recurrence relationship defined:

```
int max_profit(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;

                } else if ( w[k] > x ) {

                        OPT[k][x] = OPT[k-1][x];

                }

                else {

                        OPT[k][x] = Math.Max( V[k] + OPT[k][x-w[k]], OPT[k][x] );

                }

        }

}

return OPT[n][W]; //Maximum possible profit can be found here in the table.

}
```

D. i. Base Cases:

1. OPT[0,X] = 0

2. OPT[K,0] = 0

3. OPT[K,X] = OPT[K-1, X], if W_k > X

ii. Where the answer can be found:

The final answer with the maximum profit in the knapsack can be found at **OPT[n][W]**, where **n** is the number of items and **W** is the capacity of the knapsack.

E. Time Complexity of the solution:

The time complexity of the above code shall be $\boldsymbol{O(n * W)}$**,** since we have two for loops, one running from 1 to n, and the other running from 1 to W.

Here **'n'** is the input size or the number of items through which our algorithm should iterate, but 'W' is the total weight of the knapsack which we need to fill in and make the most profit. Hence 'W' is not the input size.

So, the time complexity of the above solution is $\boldsymbol{O}(\boldsymbol{n}.\boldsymbol{log}(\boldsymbol{W}))$ i.e., **Pseudo-polynomial time 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 cannot burst them. Design a 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 balloon after each step is:

   [3, 1, 5, 8] → [3, 5, 8] → [3, 8] → [8] → [] And the coins you get equal:

   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?**

Solution:

We are given that we have an array of **'n'** balloons that have a number painted on them. Our aim is to burst the balloons in such a way that we get the maximum coins that are possible.

A. Subproblem Definition:

The Subproblems to be solved is to find the way in which we can earn a maximum number of coins by bursting the balloons.

Assuming **'num[i]'** is the last balloon that shall be burst to earn a maximum number of coins, the subproblem is to find the maximum number of coins we can earn by bursting balloons in the left array of **‘i’** and the maximum number of coins we can earn by bursting balloons in the right array of **‘i’**.

B. Recurrence Relationship for the above dynamic programming problem.

OPT[L,R] = *Max*(nums[L-1] * nums[i] * nums[R+1] + OPT[L,i-1] + OPT[i+1, R], OPT[L,R])

Where ‘L’ is the index of the starting element in the array, and ‘R’ is the index of the ending element in the array.

Here ‘i’ is assumed to be the **last** element from the array that is to be deleted, and OPT[L,i-1] shall be the maximum coins that we can get from the left subarray, and OPT[i+1,R] be the maximum number of coins that we can get from the right subarray.

C. Pseudo Code:

```
int max_coins(int nums[]){

    int OPT[n][n];

    for(int i = 1; i <= n; i++){

        for(int j = 0; j <= n-i; j++){

            int m = j + i -1;

            for( int k = i; k <= m; k++){

                int left = 1;

                int right = 1;

                if( j != 0 ){

                    left = nums[j-1];

                }

                if( m != n-1 ){

                    right = nums[m+1];

                }

                int left_coins = 0;
```

```
                    int right_coins = 0;

                    if( j != k){

                              left_coins = OPT[j][k-1];

                    }

                    if( m != k){

                              right_coins = OPT[k+1][m];

                    }

                    OPT[j][m] = Math.Max(left * nums[k] * right + left_coins +
          right_coins, OPT[j][m]);

          }

     }

}

return OPT[0][n-1];

//The maximum coins we earn by bursting balloons can be found here in the 2-D array.

}
```

D. i. Base Cases:

- If L == R, the OPT[L,R] = 1* nums[L] * 1
- If the number of balloons is '0', i.e., L < 0 and R < 0, then OPT[L,R] = 0

ii. The final answer can be found at:

The final answer is the maximum number of coins that we can get by bursting an element in the end, and that can be found in the OPT 2-dimensional matrix at the location OPT[0][n-1].

E. Time Complexity:

The time complexity of the above pseudo-code shall be $\boldsymbol{O(n^3)}$**,** since we are using three nested for loops running from 0 to n.

Since **‘n’** is the given input size of the array of nums[], the final time complexity shall be $\boldsymbol{O}(\boldsymbol{n}^3)$, i.e., polynomial in ‘n’ of the degree of 3.