## Question 1

(a) Algorithm description

The basic idea is quite similar to what we did in class for a bounded 0/1 Knapsack. In this problem, we can take a particular item any number of times to form a subset of items that exactly match M.

To convert this problem into a simplified problem as asked in the question, we can just determine whether the solution exists or not.
To determine this we need to reduce the problem size into subproblems.
Consider the Problem size as [n, M]. Assume there exists a solution subset S whose total size = M.

(a) if A[n] does not belong to S then there exists a subset of items A[1...n-1] whose total sum = M. So the Problem size becomes [n-1, M]
Sub-problem = Find a subset of items from A[1..n-1] whose sum = M.

(b) if A[n] belong to S then there exists a subset of items A[1...n] (not n-1 as we can select that item again) whose total sum = M - A[n]. So the Problem size becomes [n, M - A[n]]
Sub-problem = Find a subset of items from A[1..n] whose sum = M - A[n].

Solution:
f[n, M] = max { f(n-1, M) , f(n, M - A[n] }
f[n, M] = 1 then "Yes"
f[n, M] = 0 then "No"

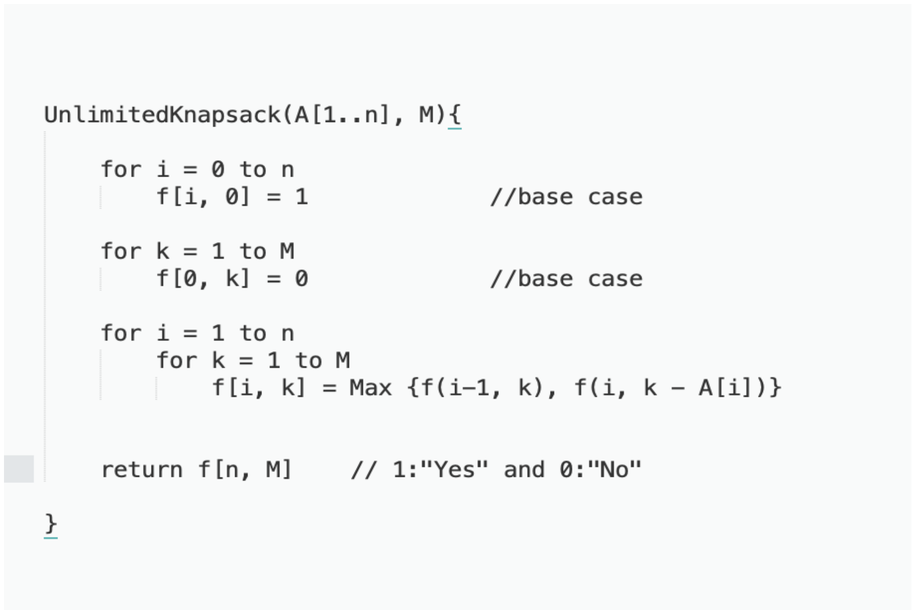
Subproblems: f[i, k], 0 $<= i <= n$ and 0 $<= k <= M$
Goal: f[n, M]
Dependency relation:
f[i, k] = max { f(i-1, k) , f(i, k - A[i] }
Base case:
f[i, 0] = 1 0 $<= i <= n$
f[0, k] = 0 1 $<= k <= M$

(b) Consider the following algorithm.

**Algorithm 1:** Determine if a Solution exists for the unlimited Knapsack problem

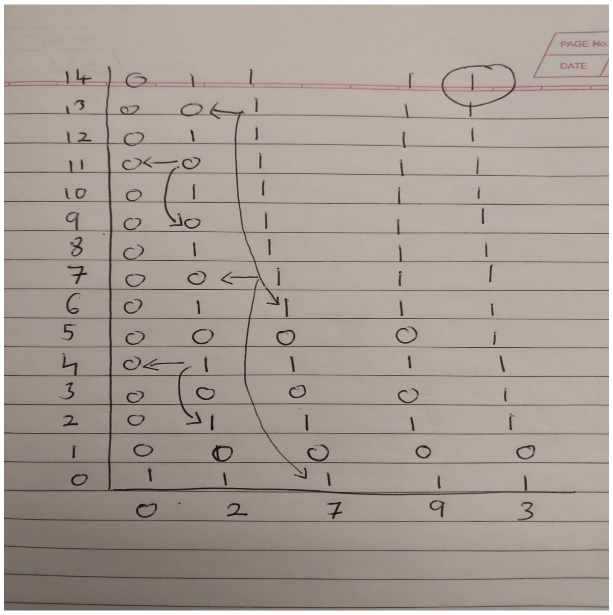
**Input:** An array A[1..n], containing n items
Knapsack of size M
**Output:** Is a solution possible or not (1 is yes, 0 is No)

```
/* Here's my algorithm. */
```



```
UnlimitedKnapsack(A[1..n], M){

    for i = 0 to n
        f[i, 0] = 1             //base case

    for k = 1 to M
        f[0, k] = 0             //base case

    for i = 1 to n
        for k = 1 to M
            f[i, k] = Max {f(i−1, k), f(i, k − A[i])}


    return f[n, M]    // 1:"Yes" and 0:"No"

}
```

(c) Correctness:
Here the dependency relation makes sure that at each iteration it checks the maximum of the previous i-1st iteration for that k or k - A[i] for the same ith iteration.
For the example provided in the question { 2, 7, 9, 3} and M = 14, We have the following table formed



| | | | | | |
|---|---|---|---|---|---|
| 14 | 0 | 1 | 1 | 1 | 1 |
| 13 | 0 | 0 | 1 | 1 | 1 |
| 12 | 0 | 1 | 1 | 1 | 1 |
| 11 | 0 | 0 | 1 | 1 | 1 |
| 10 | 0 | 1 | 1 | 1 | 1 |
| 9 | 0 | 0 | 1 | 1 | 1 |
| 8 | 0 | 1 | 1 | 1 | 1 |
| 7 | 0 | 0 | 1 | 1 | 1 |
| 6 | 0 | 1 | 1 | 1 | 1 |
| 5 | 0 | 0 | 0 | 0 | 1 |
| 4 | 0 | 1 | 1 | 1 | 1 |
| 3 | 0 | 0 | 0 | 0 | 1 |
| 2 | 0 | 1 | 1 | 1 | 1 |
| 1 | 0 | 0 | 0 | 0 | 0 |
| 0 | 1 | 1 | 1 | 1 | 1 |
| | 0 | 2 | 7 | 9 | 3 |

(d) Running time analysis.
It is quite evident from the algo that we have two for loops, till n and till M.
Time Complexity: O(nM)

## Question 2

(a) Algorithm description
The basic idea is quite similar to what we did in class for a bounded 0/1 Knapsack. In this problem, we also have values associated with items along with weight and we need to fill the knapsack by maximizing value.

To convert this problem into a simplified problem as asked in the question, we calculate the sum of values of all elements to be included in the knapsack.
To determine this we need to reduce the problem size into subproblems.
Consider the Problem size as [n, M]. Assume there exists a solution subset S whose total size = M and each element has Value V[1..n].

(a) if A[n] does not belong to S then there exists a subset of items A[1...n-1] whose total sum <= M. So the Problem size becomes [n-1, M]
Sub-problem = Find a subset of items from A[1..n-1] whose sum <= M Maximizing the value.

(b) if A[n] belong to S then there exists a subset of items A[1...n-1] whose total sum <= M - A[n]. So the Problem size becomes [n-1, M - A[n]]
Sub-problem = Find a subset of items from A[1..n] whose sum <= M - A[n] maximizing the value (adding the value in the solution) V becomes V[1..n-1].

Solution:
f[n, M] = max { f(n-1, M) , f(n-1, M - A[n]) + V[n] }
f[n, M] = the final value sum

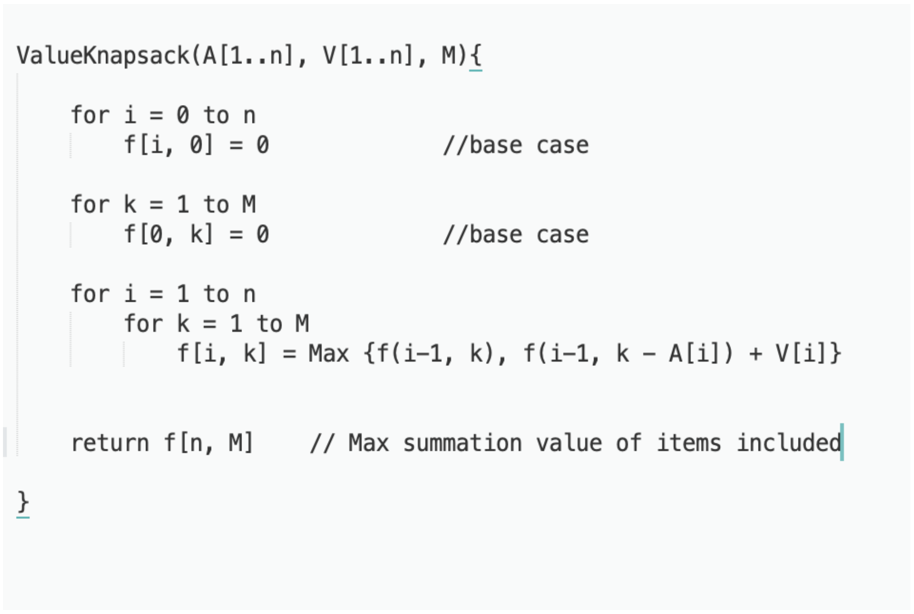
Subproblems: f[i, k], $0 <= i <= n$ and $0 <= k <= M$
Goal: f[n, M]
Dependency relation:
f[i, k] = max { f(i-1, k) , f(i-1, k - A[i]) + V[i] }
Base case:
f[i, 0] = 0 $0 <= i <= n$
f[0, k] = 0 $1 <= k <= M$

(b) Consider the following algorithm.

**Algorithm 2:** Determine the max value sum for the bounded Knapsack problem

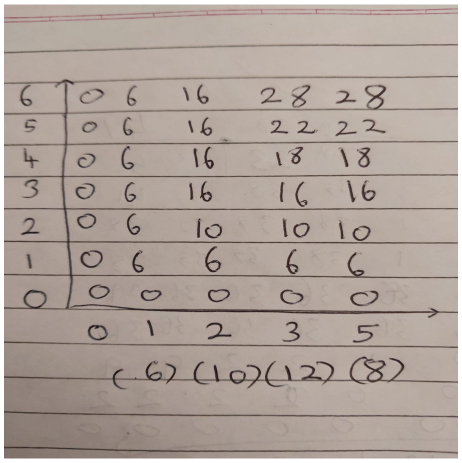
**Input:** An array A[1..n], containing n items
An array V[1..n], containing value of the n items
Knapsack of size M
**Output:** Sum of max possible values of elements whose weight <= M

```
/* Here's my algorithm.  */
```



```
ValueKnapsack(A[1..n], V[1..n], M){

    for i = 0 to n
        f[i, 0] = 0             //base case

    for k = 1 to M
        f[0, k] = 0             //base case

    for i = 1 to n
        for k = 1 to M
            f[i, k] = Max {f(i-1, k), f(i-1, k - A[i]) + V[i]}


    return f[n, M]    // Max summation value of items included

}
```

(c) Correctness:
Here the dependency relation makes sure that at each iteration it checks the maximum of the previous i-1st iteration for that k or k - A[i] and its current value for the prev i-1st iteration. For explanation see the below example { 1, 2, 3, 5} and M = 6 and Values = { 6, 10, 12, 8}, We have the following table formed
Here you can see that there are two possible combinations (1 + 2 + 3 and 5 +1), Since On



| | | | | | |
|---|---|---|---|---|---|
| 6 | 0 | 6 | 16 | 28 | 28 |
| 5 | 0 | 6 | 16 | 22 | 22 |
| 4 | 0 | 6 | 16 | 18 | 18 |
| 3 | 0 | 6 | 16 | 16 | 16 |
| 2 | 0 | 6 | 10 | 10 | 10 |
| 1 | 0 | 6 | 6 | 6 | 6 |
| 0 | 0 | 0 | 0 | 0 | 0 |
| | 0 | 1 | 2 | 3 | 5 |
| | | (6) | (10) | (12) | (8) |

taking 1 + 2 + 3, we get the max value number, our algo proves the same.

(d) Running time analysis.
It is quite evident from the algo that we have two for loops, till n and till M.
Time Complexity: O(nM)

## Question 3

(a) Algorithm description
The basic idea is quite similar to what we did in class for the longest common subsequence problem. In this problem, we have to find maximum sum for common sequences, keeping in mind the negative number as well

To convert this problem into a simplified problem as asked in the question, we calculate and return the max sum of common sub sequence.
To determine this we need to reduce the problem size into subproblems.
Suppose Z = Z1,Z2,Z3...Zk is the max sum common sequence

(a) if Z[k] not equal to X[n] then Z is an MCS of X1..Xn-1 and Y
Sub-problem = Find MCS of X1..Xn-1 and Y.

(b) if Z[k] not equal to Y[m] then Z is an MCS of Y1..Ym-1 and X
Sub-problem = Find MCS of Y1..Yn-1 and X.

(c) if Z[k] = X[n] = Y[m] Z1 to Z[k-1] is MCS of X1..Xn-1 and Y1..Ym-1 unless negative common elements that reduce the sum. Also, if all negative elements we maintain the least negative common number and return that.

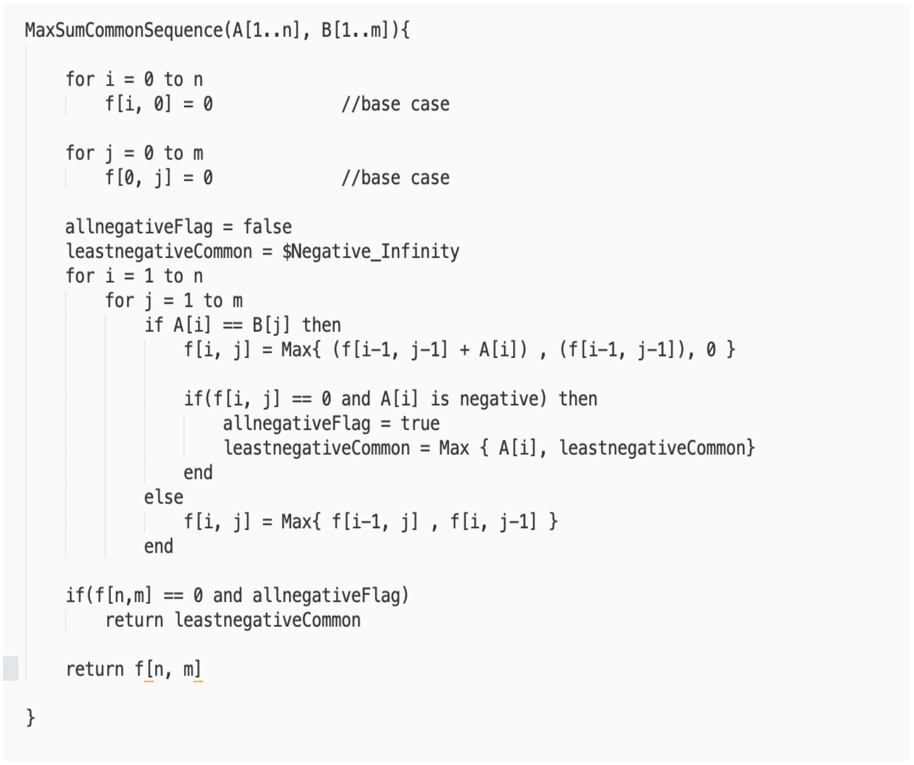
Solution:
if(X[n] = Y[m]) then
f[n, m] = max { (f[n-1, m-1] + X[n]) , (f[n-1, m-1]), 0 } (Also maintain least negative number if all numbers negative)
else
f[n, m] = max { f[n-1, m] , f[n, m-1] }

Subproblems: f[i, k], $0 <= i <= n$ and $0 <= j <= m$
Goal: f[n, m]
Dependency relation:
if(X[i] = Y[j]) then
f[i, j] = max { (f[i-1, j-1] + X[i]) , (f[i-1, j-1]), 0 } (Also maintain least negative number if all numbers negative)
else
f[i, j] = max { f[i-1, j] , f[i, j-1] }
Base case:
f[i, 0] = 0 $0 <= i <= n$
f[0, j] = 0 $0 <= j <= m$

(b) Consider the following algorithm.

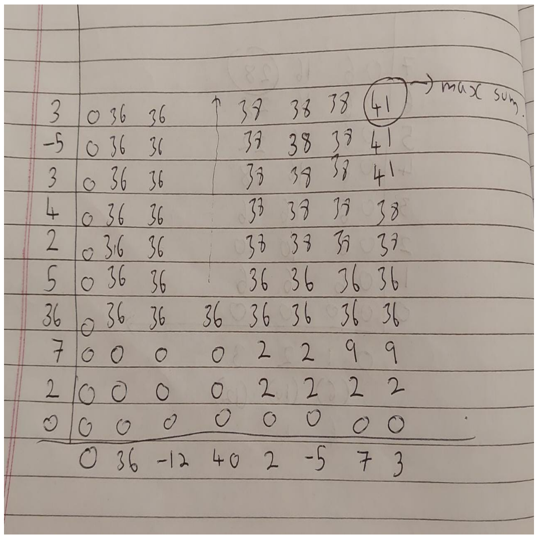
**Algorithm 3:** Determine the max sum for all the common sequences
**Input:** Two arrays A[1..n] and B[1..m], containing n and m items
**Output:** Return Max Sum of all the Common sequences

```
/* Here's my algorithm. */
```



```
MaxSumCommonSequence(A[1..n], B[1..m]){

    for i = 0 to n
        f[i, 0] = 0            //base case

    for j = 0 to m
        f[0, j] = 0            //base case

    allnegativeFlag = false
    leastnegativeCommon = $Negative_Infinity
    for i = 1 to n
        for j = 1 to m
            if A[i] == B[j] then
                f[i, j] = Max{ (f[i-1, j-1] + A[i]) , (f[i-1, j-1]), 0 }

                if(f[i, j] == 0 and A[i] is negative) then
                    allnegativeFlag = true
                    leastnegativeCommon = Max { A[i], leastnegativeCommon}
                end
            else
                f[i, j] = Max{ f[i-1, j] , f[i, j-1] }
            end

    if(f[n,m] == 0 and allnegativeFlag)
        return leastnegativeCommon

    return f[n, m]

}
```

(c) Correctness:
Here the dependency relation makes sure that if we find a common element then we check if by adding it do we get a greater number, if not then we just ignore that common element. If we don't find a common element we check max from left and down.
Consider the example given in the question: A = 36, 12, 40, 2, 5, 7, 3 and B = 2, 7, 36, 5, 2, 4, 3, 5, 3, The dependency table can be found in the image below:
Note: if all values are negative, in that case we store the least negative common value and



| | | | | | | | | |
|---|---|---|---|---|---|---|---|---|
| 3 | 0 | 36 | 36 | ↑ | 38 | 38 | 38 | 41 |
| -5 | 0 | 36 | 36 | | 38 | 38 | 38 | 41 |
| 3 | 0 | 36 | 36 | | 38 | 38 | 38 | 41 |
| 4 | 0 | 36 | 36 | | 38 | 38 | 38 | 38 |
| 2 | 0 | 36 | 36 | | 38 | 38 | 38 | 38 |
| 5 | 0 | 36 | 36 | | 36 | 36 | 36 | 36 |
| 36 | 0 | 36 | 36 | 36 | 36 | 36 | 36 | 36 |
| 7 | 0 | 0 | 0 | 0 | 2 | 2 | 9 | 9 |
| 2 | 0 | 0 | 0 | 0 | 2 | 2 | 2 | 2 |
| 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| | 0 | 36 | -12 | 40 | 2 | -5 | 7 | 3 |

→ max sum.

return that.

(d) Running time analysis.
It is quite evident from the algo that we have two for loops, till n and till m.
Time Complexity: O(nm)

## Question 4

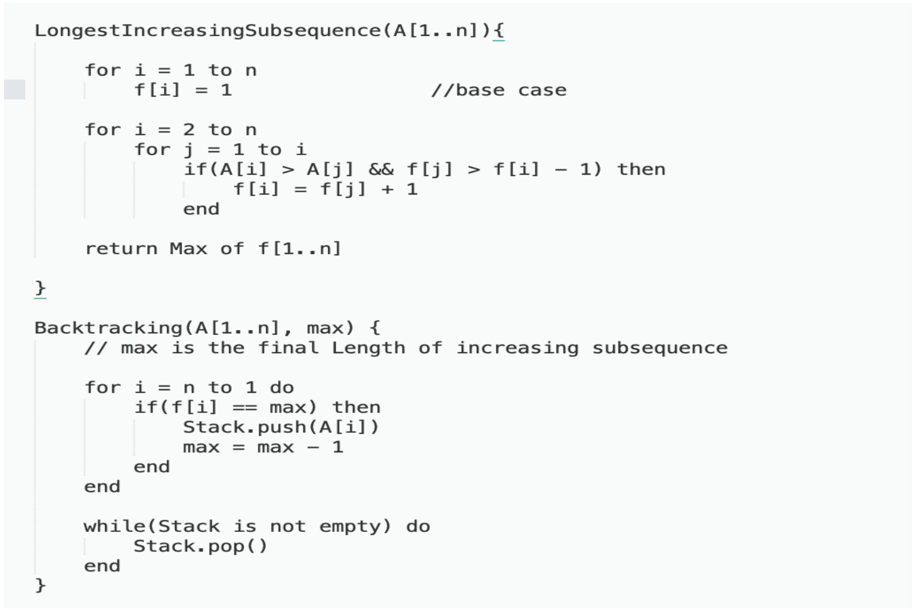
(a) Algorithm description
The basic idea here is we keep on tracking the increasing sequence based on the results from the previous iterations. Hence we use dynamic programming to use the already computed sequence length.

To convert this problem into a simplified problem, we calculate and return the length of the longest increasing sub sequence.
To determine this we need to reduce the problem size into subproblems.
for each j, $1 <= j <= n$
f[j] = the length of longest subsequence ending at j
Goal: Max (f[1],f[2] to f[n])
Dependency relation:
For each i upto n if A[i] > A[j] (increasing sequence) and f[i] + 1 > f[j] (adding 1 to solution increases the length) then
f[i] = f[j] + 1
Base case:
f[i] = 0 $1 <= i <= n$
On solving the simplified problem and getting max length we can backtrack and get the sequence using stack as shown in the Algo.

(b) Consider the following algorithm.

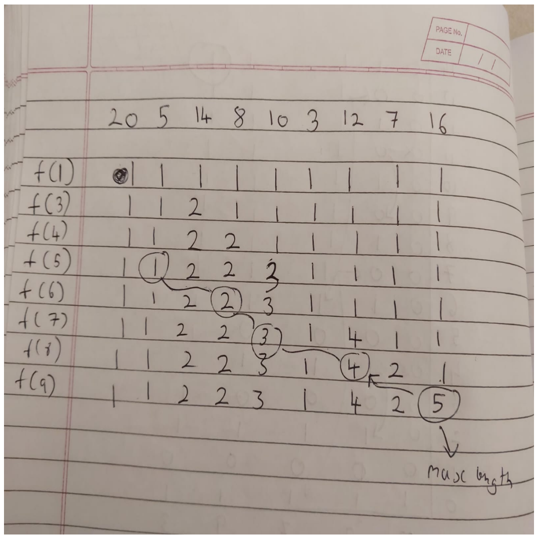
**Algorithm 4:** Determine the longest increasing sequence among all the sequences

**Input:** One array A[1..n] containing n elements
**Output:** Return Longest increasing Subsequence of all the sequences
`/* Here's my algorithm. */`



```
LongestIncreasingSubsequence(A[1..n]){

    for i = 1 to n
        f[i] = 1                //base case

    for i = 2 to n
        for j = 1 to i
            if(A[i] > A[j] && f[j] > f[i] - 1) then
                f[i] = f[j] + 1
            end

    return Max of f[1..n]

}

Backtracking(A[1..n], max) {
    // max is the final Length of increasing subsequence

    for i = n to 1 do
        if(f[i] == max) then
            Stack.push(A[i])
            max = max - 1
        end
    end

    while(Stack is not empty) do
        Stack.pop()
    end
}
```

(c) Correctness:
Here the dependency relation makes sure that Each iteration computes the longest subsequence until j and then using that value we add it to the next iteration.
In the end, the max of all solutions gives us the final goal.
Using the max length we can backtrack the solution from the original array as shown. Considering the example from the question: A = 20, 5, 14, 8, 10, 3, 12, 7, 16
We can see all dependence computations in the diagram below.



| | 20 | 5 | 14 | 8 | 10 | 3 | 12 | 7 | 16 |
|---|---|---|---|---|---|---|---|---|---|
| f(1) | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 |
| f(3) | 1 | 1 | 2 | 1 | 1 | 1 | 1 | 1 | 1 |
| f(4) | 1 | 1 | 2 | 2 | 1 | 1 | 1 | 1 | 1 |
| f(5) | 1 | 1 | 2 | 2 | 3 | 1 | 1 | 1 | 1 |
| f(6) | 1 | 1 | 2 | 2 | 3 | 1 | 1 | 1 | 1 |
| f(7) | 1 | 1 | 2 | 2 | 3 | 1 | 4 | 1 | 1 |
| f(8) | 1 | 1 | 2 | 2 | 3 | 1 | 4 | 2 | 1 |
| f(9) | 1 | 1 | 2 | 2 | 3 | 1 | 4 | 2 | 5 |

(d) Running time analysis.
It is quite evident from the algo that we have two for loops, till n.
Time Complexity: O($n^2$).