DESIGN AND ANALYSIS OF ALGORITHMS LABORATORY

Lab Manual

Submitted in partial fulfilment of the requirements for the award of the degree of

BACHELOR OF TECHNOLOGY

(Information Technology)

Submitted by:

Shivay Bhandari (D3 IT-B1)

CRN:1921142

URN: 1905398

Submitted to:

Er. Manjot Kaur

Assistant Professor

Department of Information Technology,

Guru Nanak Dev Engineering College,

Ludhiana-141006

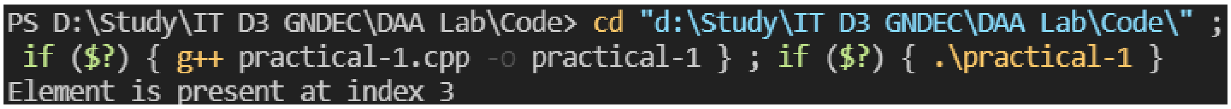
INDEX

Practical - 1

Implement binary search algorithm and compute its time complexity

```cpp
#include <bits/stdc++.h>
using namespace std;

int binarySearch(int arr[], int l, int r, int x)
{
    if (r >= l)
    {
        int mid = l + (r - l) / 2;
        if (arr[mid] == x)
            return mid;
        if (arr[mid] > x)
            return binarySearch(arr, l, mid - 1, x);
        return binarySearch(arr, mid + 1, r, x);
    }

    return -1;
}

int main(void)
{
    int arr[] = {2, 3, 4, 10, 40};
    int x = 10;
    int n = sizeof(arr) / sizeof(arr[0]);
    int result = binarySearch(arr, 0, n - 1, x);
    (result == -1)
        ? cout << "Element is not present in array"
        : cout << "Element is present at index " << result;
    return 0;
}
```

Output –



```
PS D:\Study\IT D3 GNDEC\DAA Lab\Code> cd "d:\Study\IT D3 GNDEC\DAA Lab\Code\" ;
 if ($?) { g++ practical-1.cpp -o practical-1 } ; if ($?) { .\practical-1 }
Element is present at index 3
```

Time Complexity –

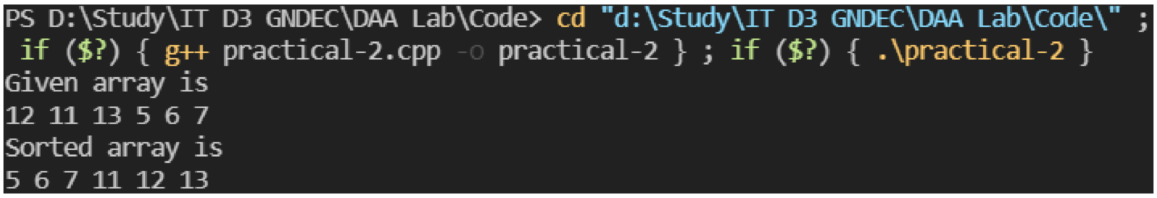
O(log(n))

Practical 2

Implement merge sort algorithm and demonstrate divide and conquer technique

```
#include <iostream>
using namespace std;

void merge(int array[], int const left, int const mid, int const right)
{
    auto const subArrayOne = mid - left + 1;
    auto const subArrayTwo = right - mid;
    auto *leftArray = new int[subArrayOne],
         *rightArray = new int[subArrayTwo];
    for (auto i = 0; i < subArrayOne; i++)
        leftArray[i] = array[left + i];
    for (auto j = 0; j < subArrayTwo; j++)
        rightArray[j] = array[mid + 1 + j];

    auto indexOfSubArrayOne = 0,
         indexOfSubArrayTwo = 0;
    int indexOfMergedArray = left;

    while (indexOfSubArrayOne < subArrayOne && indexOfSubArrayTwo <
subArrayTwo)
    {
        if (leftArray[indexOfSubArrayOne] <= rightArray[indexOfSubArrayTwo])
        {
            array[indexOfMergedArray] = leftArray[indexOfSubArrayOne];
            indexOfSubArrayOne++;
        }
        else
        {
            array[indexOfMergedArray] = rightArray[indexOfSubArrayTwo];
            indexOfSubArrayTwo++;
        }
        indexOfMergedArray++;
    }

    while (indexOfSubArrayOne < subArrayOne)
    {
        array[indexOfMergedArray] = leftArray[indexOfSubArrayOne];
        indexOfSubArrayOne++;
        indexOfMergedArray++;
    }

    while (indexOfSubArrayTwo < subArrayTwo)
    {
        array[indexOfMergedArray] = rightArray[indexOfSubArrayTwo];
        indexOfSubArrayTwo++;
        indexOfMergedArray++;
    }
}

void mergeSort(int array[], int const begin, int const end)
{
    if (begin >= end)
        return;

    auto mid = begin + (end - begin) / 2;
    mergeSort(array, begin, mid);
    mergeSort(array, mid + 1, end);
    merge(array, begin, mid, end);
}

void printArray(int A[], int size)
{
    for (auto i = 0; i < size; i++)
        cout << A[i] << " ";
}

int main()
{
    int arr[] = {12, 11, 13, 5, 6, 7};
    auto arr_size = sizeof(arr) / sizeof(arr[0]);

    cout << "Given array is \n";
    printArray(arr, arr_size);

    mergeSort(arr, 0, arr_size - 1);

    cout << "\nSorted array is \n";
    printArray(arr, arr_size);
    return 0;
}
```

Output –



```
PS D:\Study\IT D3 GNDEC\DAA Lab\Code> cd "d:\Study\IT D3 GNDEC\DAA Lab\Code\" ;
 if ($?) { g++ practical-2.cpp -o practical-2 } ; if ($?) { .\practical-2 }
Given array is
12 11 13 5 6 7
Sorted array is
5 6 7 11 12 13
```

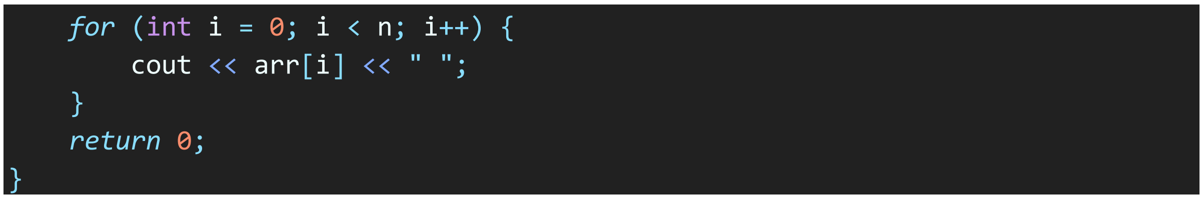
Time Complexity -

O(n*log(n))

PRACTICAL 3

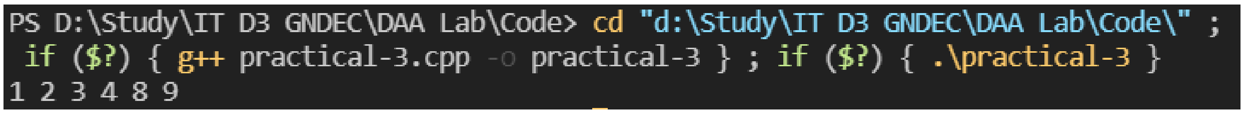
Analyze the time complexity of Quick-sort algorithm

```cpp
#include <iostream>
using namespace std;
int partition(int arr[], int start, int end)
{
    int pivot = arr[start];
    int count = 0;
    for (int i = start + 1; i <= end; i++) {
        if (arr[i] <= pivot)
            count++;
    }
    int pivotIndex = start + count;
    swap(arr[pivotIndex], arr[start]);
    int i = start, j = end;
    while (i < pivotIndex && j > pivotIndex) {
        while (arr[i] <= pivot) {
            i++;
        }
        while (arr[j] > pivot) {
            j--;
        }
        if (i < pivotIndex && j > pivotIndex) {
            swap(arr[i++], arr[j--]);
        }
    }
    return pivotIndex;
}
void quickSort(int arr[], int start, int end)
{
    if (start >= end)
        return;
    int p = partition(arr, start, end);
    quickSort(arr, start, p - 1);
    quickSort(arr, p + 1, end);
}
int main()
{
    int arr[] = { 9, 3, 4, 2, 1, 8 };
    int n = 6;
    quickSort(arr, 0, n - 1);
```



```
    for (int i = 0; i < n; i++) {
        cout << arr[i] << " ";
    }
    return 0;
}
```

Output –



```
PS D:\Study\IT D3 GNDEC\DAA Lab\Code> cd "d:\Study\IT D3 GNDEC\DAA Lab\Code\" ;
 if ($?) { g++ practical-3.cpp -o practical-3 } ; if ($?) { .\practical-3 }
1 2 3 4 8 9
```

Recurrence Relation –

<u>Worst Case Analysis</u>

Recurrence Relation:
$T(0) = T(1) = 0$ *(base case)*
$T(N) = N + T(N-1)$

Solving the RR:

$T(N) = N + T(N-1)$
$T(N-1) = (N-1) + T(N-2)$
$T(N-2) = (N-2) + T(N-3)$

...

$T(3) = 3 + T(2)$
$T(2) = 2 + T(1)$
$T(1) = 0$

Hence,

$T(N) = N + (N-1) + (N-2) \dots + 3 + 2$

$$\approx \frac{N^2}{2}$$

which is $O(N^2)$

Best Case Analysis

Recurrence Relation:

T(0) = T(1) = 0 *(base case)*
T(N) = 2T(N/2) + N

Solving the RR:

$$\frac{T(N)}{N} = \frac{N}{N} + \frac{2T(N/2)}{N}$$ Note: Divide both side of recurrence relation by N

$$\frac{T(N)}{N} = 1 + \frac{T(N/2)}{N/2}$$

$$\frac{T(N/2)}{N/2} = 1 + \frac{T(N/4)}{N/4}$$

$$\frac{T(N/4)}{N/4} = 1 + \frac{T(N/8)}{N/8}$$

...

$$\frac{T(\frac{N}{N/2})}{\frac{N}{N/2}} = 1 + \frac{T(\frac{N}{N})}{\frac{N}{N}} = 1 + \frac{T(1)}{1}$$

same as

$$\frac{T(2)}{2} = 1 + \frac{T(1)}{1}$$ *Note: T(1) = 0*

Hence,

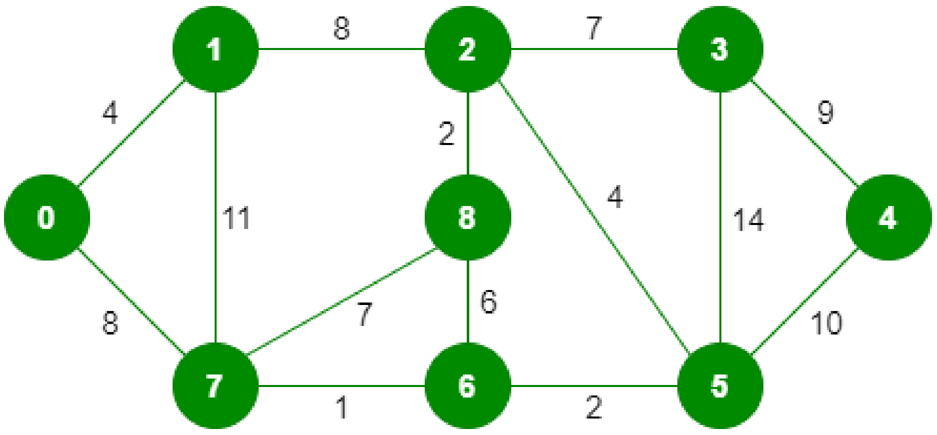
$$\frac{T(N)}{N} = 1 + 1 + 1 + \ldots 1$$ *Note: log(N) terms*

$$\frac{T(N)}{N} = \log N$$

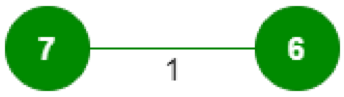
$T(N) = N \log N$ *which is O(N logN)*

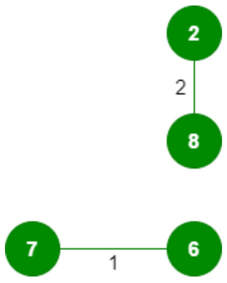
PRACTICAL - 4

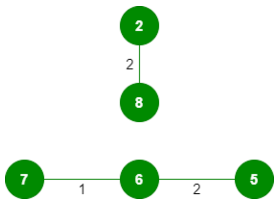
Solve minimum cost spanning tree problem using greedy method



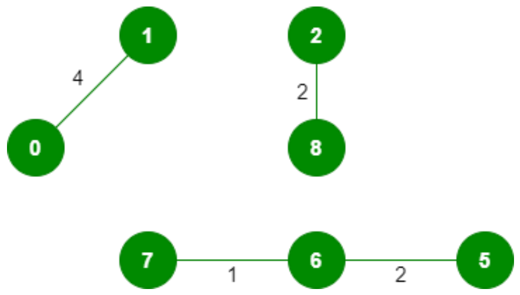
The graph contains 9 vertices and 14 edges. So, the minimum spanning tree formed will be having (9 – 1) = 8 edges.

1. Pick edge 7-6: No cycle is formed, include it.
   a. 

2. Pick edge 8-2: No cycle is formed, include it.
   a. 

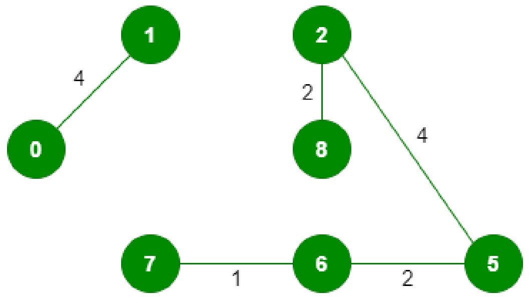
3. Pick edge 6-5: No cycle is formed, include it.
   a. 

4. Pick edge 0-1: No cycle is formed, include it.



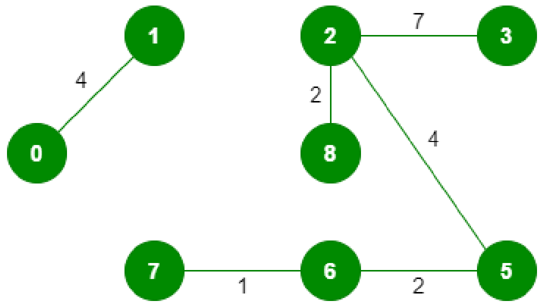
a.

5. Pick edge 2-5: No cycle is formed, include it.



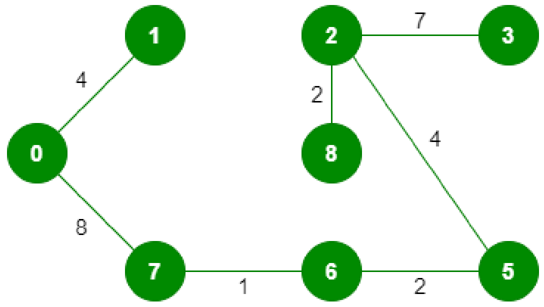
a.

6. Pick edge 8-6: Since including this edge results in the cycle, discard it.
7. Pick edge 2-3: No cycle is formed, include it.



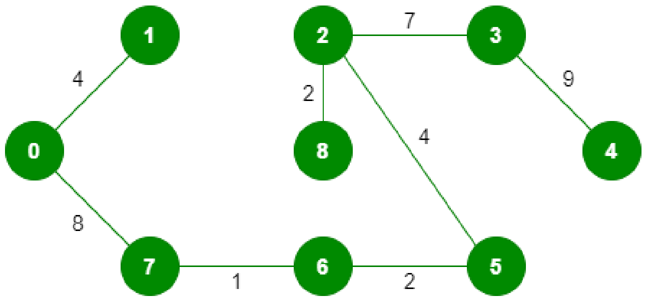
a.

8. Pick edge 7-8: Since including this edge results in the cycle, discard it.
9. Pick edge 0-7: No cycle is formed, include it.



a.

10. Pick edge 1-2: Since including this edge results in the cycle, discard it.
11. Pick edge 3-4: No cycle is formed, include it.

a. 

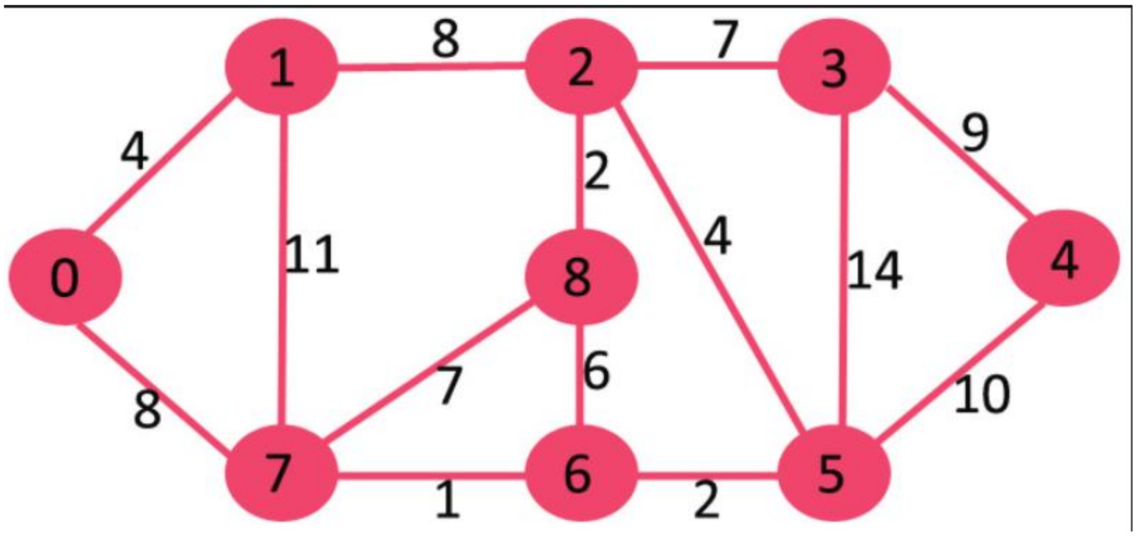
12. Since the number of edges included equals (V – 1), the algorithm stops here.

Time Complexity –

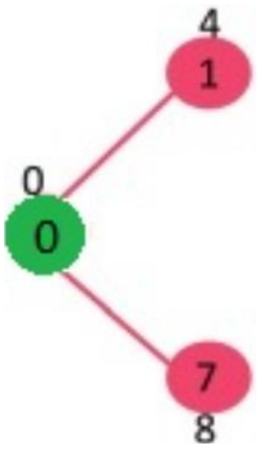
O(E(log(V)))

PRACTICAL - 5

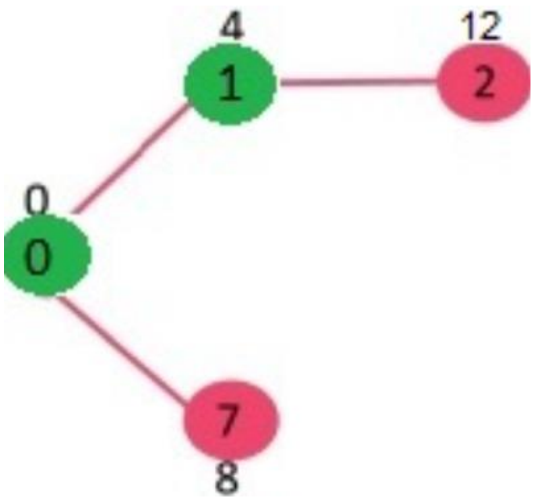
Implement greedy algorithm to solve single-source shortest path problem



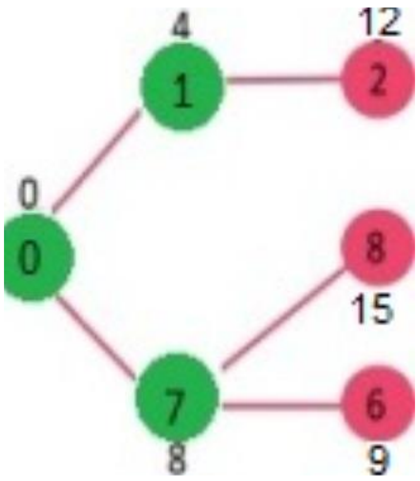
The set *sptSet* is initially empty and distances assigned to vertices are {0, INF, INF, INF, INF, INF, INF, INF} where INF indicates infinite. Now pick the vertex with a minimum distance value. The vertex 0 is picked, include it in *sptSet*. So *sptSet* becomes {0}. After including 0 to *sptSet*, update distance values of its adjacent vertices. Adjacent vertices of 0 are 1 and 7. The distance values of 1 and 7 are updated as 4 and 8. The following subgraph shows vertices and their distance values, only the vertices with finite distance values are shown. The vertices included in SPT are shown in green colour.



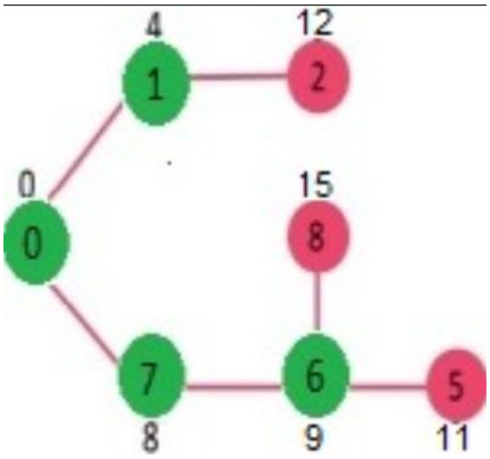
Pick the vertex with minimum distance value and not already included in SPT (not in sptSET). The vertex 1 is picked and added to sptSet. So sptSet now becomes {0, 1}. Update the distance values of adjacent vertices of 1. The distance value of vertex 2 becomes 12.



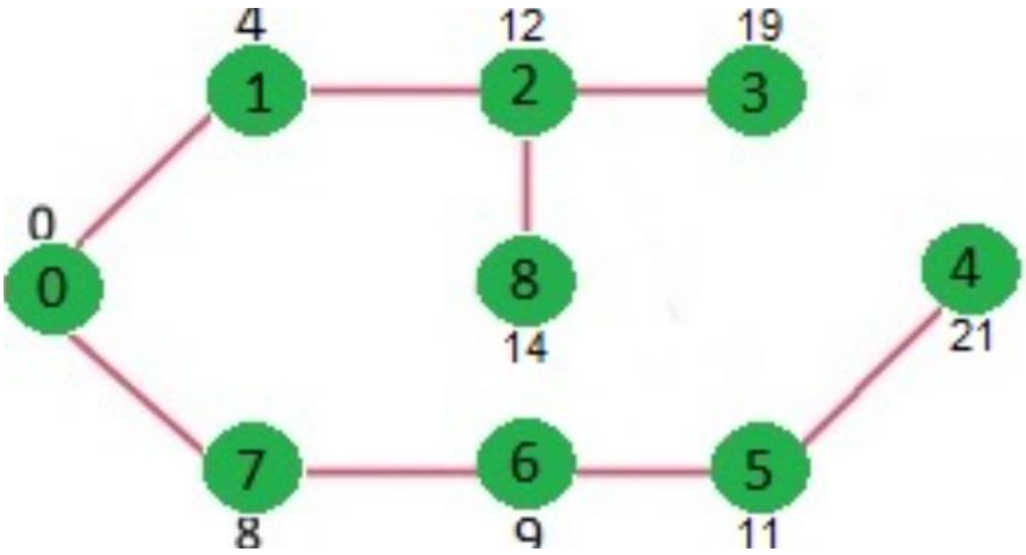
Pick the vertex with minimum distance value and not already included in SPT (not in sptSET). Vertex 7 is picked. So sptSet now becomes {0, 1, 7}. Update the distance values of adjacent vertices of 7. The distance value of vertex 6 and 8 becomes finite (15 and 9 respectively).



Pick the vertex with minimum distance value and not already included in SPT (not in sptSET). Vertex 6 is picked. So sptSet now becomes {0, 1, 7, 6}. Update the distance values of adjacent vertices of 6. The distance value of vertex 5 and 8 are updated.



We repeat the above steps until *sptSet* includes all vertices of the given graph. Finally, we get the following Shortest Path Tree (SPT).



Practical 6

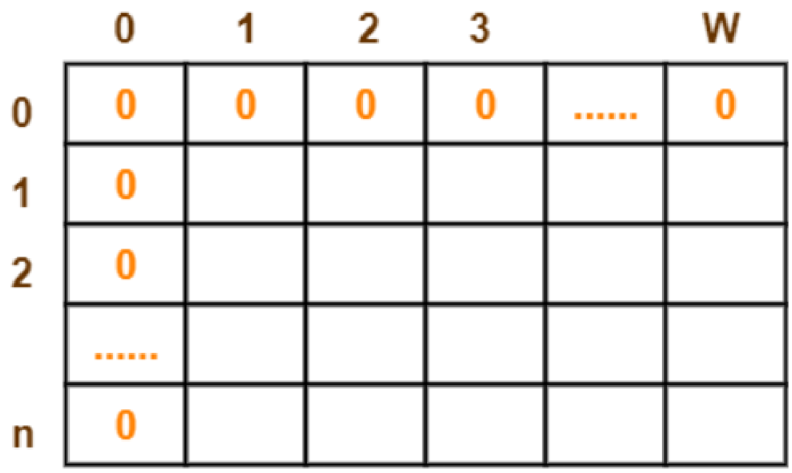
Use dynamic programming to solve Knapsack problem

Consider –

- Knapsack weight capacity = w
- Number of items each having some weight and value = n

0/1 knapsack problem is solved using dynamic programming in the following steps-

- Draw a table say ‘T’ with (n+1) number of rows and (w+1) number of columns.
- Fill all the boxes of 0th row and 0th column with zeroes as shown-



	0	1	2	3		W
0	0	0	0	0	……	0
1	0					
2	0					
	……					
n	0					

T-Table

- Start filling the table row wise top to bottom from left to right.
- Use the following formula-
    - $T(i, j) = \max \{ T(i-1, j), value_i + T(i-1, j - weight_i) \}$

Here, T(i , j) = maximum value of the selected items if we can take items 1 to i and have weight restrictions of j.

- This step leads to completely filling the table.
- Then, value of the last box represents the maximum possible value that can be put into the knapsack.

To identify the items that must be put into the knapsack to obtain that maximum profit-

- Consider the last column of the table.
- Start scanning the entries from bottom to top.
- On encountering an entry whose value is not same as the value stored in the entry immediately above it, mark the row label of that entry.
- After all the entries are scanned, the marked labels represent the items that must be put into the knapsack.

Time Complexity –

- Each entry of the table requires constant time θ(1) for its computation.
- It takes θ(nw) time to fill (n+1)(w+1) table entries

- It takes θ(n) time for tracing the solution since tracing process traces the n rows.
- Thus, overall θ(nw) time is taken to solve 0/1 knapsack problem using dynamic programming.

Practical 7

To

Practical 8

To

Practical 9

To