UNIT-3

Design and Analysis of Algorithms

Design and Analysis of Algorithms is a fundamental aspect of computer science that involves creating efficient solutions to computational problems and evaluating their performance.

DSA focuses on designing algorithms that effectively address specific challenges and analyzing their efficiency in terms of time and space complexity.

The importance of design and analysis of algorithms lies in the following aspects:

Efficient problem-solving: Understanding how to design and analyze algorithms forms the foundation for creating better software and solving complex problems.

Scalability and trade-offs: Analyzing algorithms helps developers design more efficient and scalable systems, make informed decisions about algorithm selection, and understand trade-offs.

Algorithm discovery and improvement: It helps discover good algorithms, improve existing ones, and analyze their efficiency mathematically.

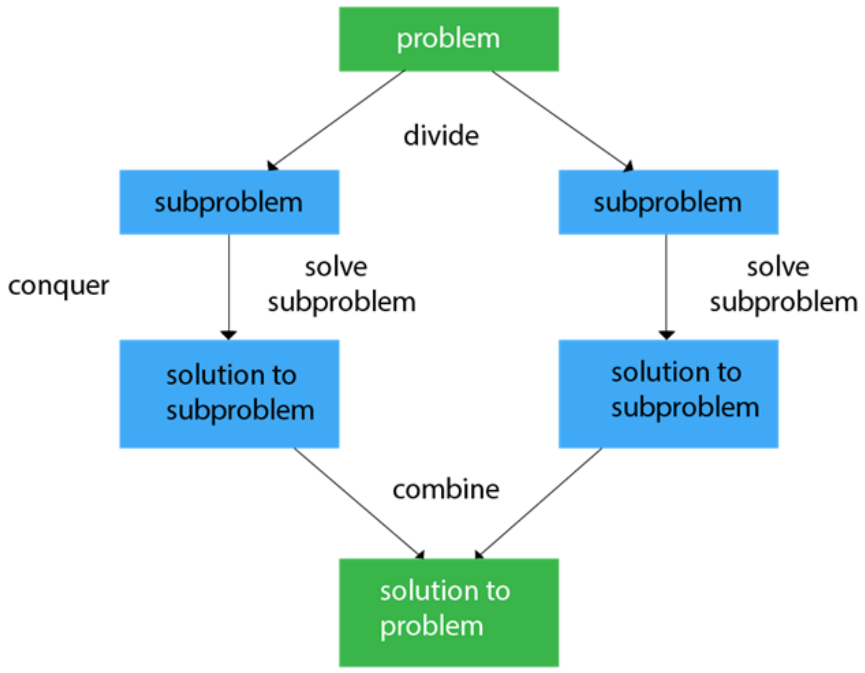
Logical design: It allows designing and analyzing the logic of how a program will work before developing the actual code

Divide and Conquer: The General Method

Divide and Conquer is an algorithmic pattern. In algorithmic methods, the design is to take a dispute on a huge input, break the input into minor pieces, decide the problem on each of the small pieces, and then merge the piecewise solutions into a global solution. This mechanism of solving the problem is called the Divide & Conquer Strategy.

Divide and Conquer algorithm consists of a dispute using the following three steps.

1. **Divide** the original problem into a set of subproblems.
2. **Conquer:** Solve every subproblem individually, recursively.
3. **Combine:** Put together the solutions of the subproblems to get the solution to the whole problem.



Generally, we can follow the **divide-and-conquer** approach in a three-step process.

Examples: The specific computer algorithms are based on the Divide & Conquer approach:

1. Maximum and Minimum Problem
2. Binary Search
3. Sorting (merge sort, quick sort)
4. Tower of Hanoi.

Fundamental of Divide & Conquer Strategy:

There are two fundamental of Divide & Conquer Strategy:

1. Relational Formula
2. Stopping Condition

1. Relational Formula: It is the formula that we generate from the given technique. After generation of Formula we apply D&C Strategy, i.e. we break the problem recursively & solve the broken sub problems.

2. Stopping Condition: When we break the problem using Divide & Conquer Strategy, then we need to know that for how much time, we need to apply divide & Conquer. So the condition where the need to stop our recursion steps of D&C is called as Stopping Condition.

Applications of Divide and Conquer Approach:

Following algorithms are based on the concept of the Divide and Conquer Technique:

1. **Binary Search:** The binary search algorithm is a searching algorithm, which is also called a half-interval search or logarithmic search. It works by comparing the target value with the middle element existing in a sorted array. After making the comparison, if the value differs, then the half that cannot contain the target will eventually eliminate, followed by continuing the search on the other half. We will again consider the middle element and compare it with the target value. The process keeps on repeating until the target value is met. If we found the other half to be empty after ending the search, then it can be concluded that the target is not present in the array.
2. **Quicksort:** It is the most efficient sorting algorithm, which is also known as partition-exchange sort. It starts by selecting a pivot value from an array followed by dividing the rest of the array elements into two sub-arrays. The partition is made by comparing each of the elements with the pivot value. It compares whether the element holds a greater value or lesser value than the pivot and then sort the arrays recursively.
3. **Merge Sort:** It is a sorting algorithm that sorts an array by making comparisons. It starts by dividing an array into sub-array and then recursively sorts each of them. After the sorting is done, it merges them back.
4. **Closest Pair of Points:** It is a problem of computational geometry. This algorithm emphasizes finding out the closest pair of points in a metric space,

given n points, such that the distance between the pair of points should be minimal.

5. **Strassen's Algorithm:** It is an algorithm for matrix multiplication, which is named after Volker Strassen. It has proven to be much faster than the traditional algorithm when works on large matrices.
6. **Cooley-Tukey Fast Fourier Transform (FFT) algorithm:** The Fast Fourier Transform algorithm is named after J. W. Cooley and John Turkey. It follows the Divide and Conquer Approach and imposes a complexity of O(nlogn).
7. **Karatsuba algorithm for fast multiplication:** It is one of the fastest multiplication algorithms of the traditional time, invented by Anatoly Karatsuba in late 1960 and got published in 1962. It multiplies two n-digit numbers in such a way by reducing it to at most single-digit.

Advantages of Divide and Conquer

- Divide and Conquer tend to successfully solve one of the biggest problems, such as the **Tower of Hanoi,** a mathematical puzzle. It is challenging to **solve complicated problems** for which you have no basic idea, but with the help of the divide and conquer approach, it has lessened the effort as it works on dividing the main problem into two halves and then solve them recursively. This algorithm is much faster than other algorithms.
- It efficiently uses **cache memory** without occupying much space because it solves simple subproblems within the cache memory instead of accessing the slower main memory.
- It is **more proficient** than that of its counterpart **Brute Force technique**.
- Since these algorithms inhibit parallelism, it does not involve any modification and is handled by systems incorporating **parallel processing**.

Disadvantages of Divide and Conquer

- Since most of its algorithms are designed by incorporating **recursion**, so it necessitates **high memory** management.
- An **explicit stack** may **overuse** the **space**.
- It may even **crash the system** if the recursion is performed rigorously greater than the stack present in the CPU.

Quick Sort

Quick sort is a sorting algorithm based on the Divide and Conquer that picks an element as a pivot and partitions the given array around the picked pivot by placing the pivot in its correct position in the sorted array.

How does Quick Sort Algorithm work?

Quicksort works on the principle of **divide and conquer**, breaking down the problem into smaller sub-problems.

There are mainly three steps in the algorithm:

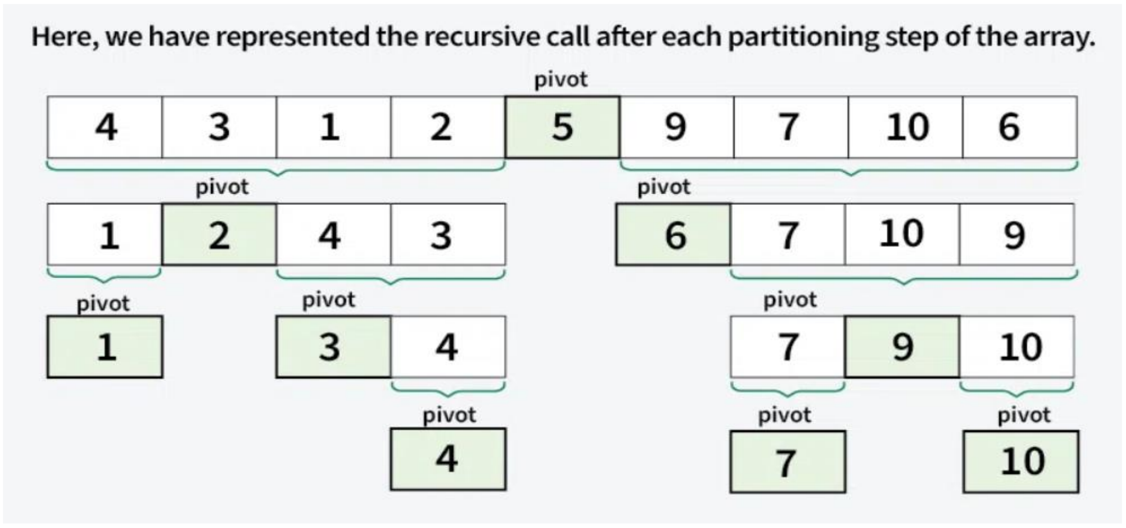
1. **Choose a Pivot:** Select an element from the array as the pivot. The choice of pivot can vary (e.g., first element, last element, random element, or median).

2. **Partition the Array:** Rearrange the array around the pivot. After partitioning, all elements smaller than the pivot will be on its left, and all elements greater than the pivot will be on its right. The pivot is then in its correct position, and we obtain the index of the pivot.

3. **Recursively Call:** Recursively apply the same process to the two partitioned sub-arrays (left and right of the pivot).

4. **Base Case:** The recursion stops when there is only one element left in the sub-array, as a single element is already sorted.

Here's a basic overview of how the Quick Sort algorithm works.



Choice of Pivot

There are many different choices for picking pivots.

- Always pick the first (or last) element as a pivot. The below implementation is picks the last element as pivot. The problem with this approach is it ends up in the worst case when array is already sorted.

- Pick a random element as a pivot. This is a preferred approach because it does not have a pattern for which the worst case happens.

- Pick the median element is pivot. This is an ideal approach in terms of time complexity as we can find median in linear time and the partition function will always divide the input array into two halves. But it is low on average as median finding has high constants.

Partition Algorithm

The key process in **quickSort** is a **partition().** There are three common algorithms to partition. All these algorithms have O(n) time complexity.

1. **Naive Partition**: Here we create copy of the array. First put all smaller elements and then all greater. Finally we copy the temporary array back to original array. This requires O(n) extra space.

2. **Lomuto Partition**: We have used this partition in this Example. This is a simple algorithm, we keep track of index of smaller elements and keep swapping. We have used it here in this example because of its simplicity.

3. **Hoare's Partition**: This is the fastest of all. Here we traverse array from both sides and keep swapping greater element on left with smaller on right while the array is not partitioned.

Working of Partition Algorithm with Illustration

The logic is simple, we start from the leftmost element and keep track of the index of smaller (or equal) elements as ***i*** *. While traversing, if we find a smaller element, we swap the current element with* ***arr[i]****. Otherwise, we ignore the current element.*

Let us understand the working of partition algorithm with the help of the following example:

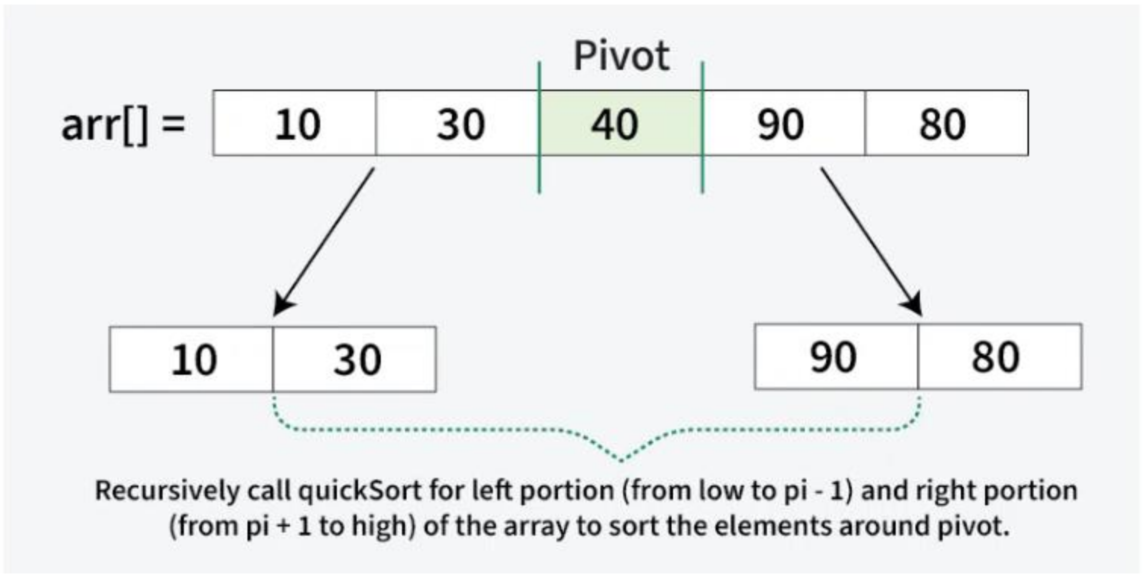
Illustration of Quick Sort Algorithm

In the previous step, we looked at how the **partitioning** process rearranges the array based on the chosen **pivot**.

Next, we apply the same method recursively to the smaller sub-arrays on the **left** and **right** of the pivot.

Each time, we select new pivots and partition the arrays again. This process continues until only one element is left, which is always sorted. Once every element is in its correct position, the entire array is sorted.

Below image illustrates how the recursive method calls for the smaller sub-arrays on the **left** and **right** of the **pivot**:


Pivot
arr[] =
10
30
40
90
80
10
30
90
80
Recursively call quickSort for left portion (from low to pi - 1) and right portion (from pi + 1 to high) of the array to sort the elements around pivot.

```
#include <stdio.h>

void swap(int* a, int* b);
```

```c
// Partition function
int partition(int arr[], int low, int high) {

    // Choose the pivot
    int pivot = arr[high];

    // Index of smaller element and indicates
    // the right position of pivot found so far
    int i = low - 1;

    // Traverse arr[low..high] and move all smaller
    // elements to the left side. Elements from low to
    // i are smaller after every iteration
    for (int j = low; j <= high - 1; j++) {
        if (arr[j] < pivot) {
            i++;
            swap(&arr[i], &arr[j]);
        }
    }

    // Move pivot after smaller elements and
    // return its position
    swap(&arr[i + 1], &arr[high]);
    return i + 1;
}

// The QuickSort function implementation
void quickSort(int arr[], int low, int high) {
    if (low < high) {

        // pi is the partition return index of pivot
        int pi = partition(arr, low, high);

        // Recursion calls for smaller elements
        // and greater or equals elements
        quickSort(arr, low, pi - 1);
        quickSort(arr, pi + 1, high);
    }
}

void swap(int* a, int* b) {
    int t = *a;
    *a = *b;
    *b = t;
}
```

```
int main() {
    int arr[] = {10, 7, 8, 9, 1, 5};
    int n = sizeof(arr) / sizeof(arr[0]);

    quickSort(arr, 0, n - 1);
    for (int i = 0; i < n; i++) {
        printf("%d ", arr[i]);
    }

    return 0;
}
```

Complexity Analysis of Quick Sort

Time Complexity:

- **Best Case:** (Ω(n log n)), Occurs when the pivot element divides the array into two equal halves.
- **Average Case** (θ(n log n)), On average, the pivot divides the array into two parts, but not necessarily equal.
- **Worst Case:** (O(n^2)), Occurs when the smallest or largest element is always chosen as the pivot (e.g., sorted arrays).

Auxiliary Space: O(n), due to recursive call stack

Advantages of Quick Sort

- It is a divide-and-conquer algorithm that makes it easier to solve problems.
- It is efficient on large data sets.
- It has a low overhead, as it only requires a small amount of memory to function.
- It is Cache Friendly as we work on the same array to sort and do not copy data to any auxiliary array.
- Fastest general-purpose algorithm for large data when stability is not required.
- It is **tail recursive** and hence all the tail call optimization can be done.

Disadvantages of Quick Sort

- It has a worst-case time complexity of O(n2), which occurs when the pivot is chosen poorly.
- It is not a good choice for small data sets.
- It is not a stable sort, meaning that if two elements have the same key, their relative order will not be preserved in the sorted output in case of quick sort, because here we are swapping elements according to the pivot's position (without considering their original positions).

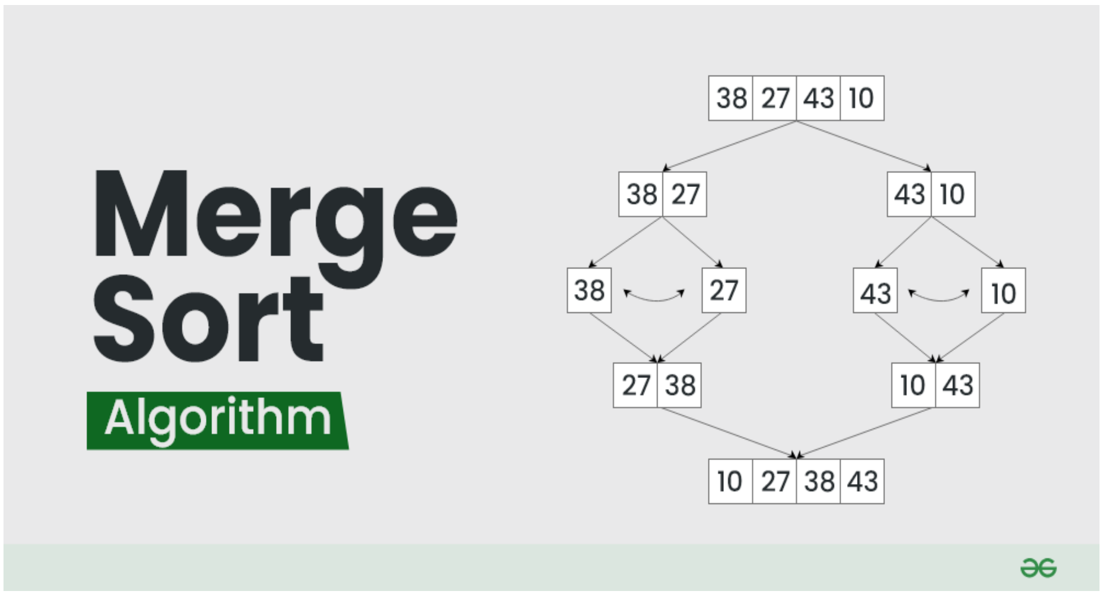
Applications of Quick Sort

- Efficient for sorting large datasets with O(nlogn) average-case time complexity.
- Used in partitioning problems like finding the kth smallest element or dividing arrays by pivot.
- Integral to randomized algorithms, offering better performance than deterministic approaches.
- Applied in cryptography for generating random permutations and unpredictable encryption keys.
- Partitioning step can be parallelized for improved performance in multi-core or distributed systems.
- Important in theoretical computer science for analyzing average-case complexity and developing new techniques.

Merge Sort

Merge sort is a sorting algorithm that follows the **divide-and-conquer** approach. It works by recursively dividing the input array into smaller subarrays and sorting those subarrays then merging them back together to obtain the sorted array.

In simple terms, we can say that the process of **merge sort** is to divide the array into two halves, sort each half, and then merge the sorted halves back together. This process is repeated until the entire array is sorted.



Merge Sort Algorithm

How does Merge Sort work?

Merge sort is a popular sorting algorithm known for its efficiency and stability. It follows the **divide-and-conquer** approach to sort a given array of elements.

Here’s a step-by-step explanation of how merge sort works:

1. **Divide:** Divide the list or array recursively into two halves until it can no more be divided.
2. **Conquer:** Each subarray is sorted individually using the merge sort algorithm.
3. **Merge:** The sorted subarrays are merged back together in sorted order. The process continues until all elements from both subarrays have been merged.

Illustration of Merge Sort:

Let’s sort the array or list **[38, 27, 43, 10]** using Merge Sort

Let’s look at the working of above example:

Divide:

- ***[38, 27, 43, 10]*** *is divided into* ***[38, 27*** *]* *and* ***[43, 10]*** *.*
- ***[38, 27]*** *is divided into* ***[38]*** *and* ***[27]*** *.*

- ***[43, 10]** is divided into **[43]** and **[10]** .*

Conquer:

- ***[38]** is already sorted.*
- ***[27]** is already sorted.*
- ***[43]** is already sorted.*
- ***[10]** is already sorted.*

Merge:

- *Merge **[38]** and **[27]** to get **[27, 38]** .*
- *Merge **[43]** and **[10]** to get **[10,43]** .*
- *Merge **[27, 38]** and **[10,43]** to get the final sorted list **[10, 27, 38, 43]***

*Therefore, the sorted list is **[10, 27, 38, 43]** .*

Implementation of Merge Sort:

```
// C program for Merge Sort
#include <stdio.h>
#include <stdlib.h>

// Merges two subarrays of arr[].
// First subarray is arr[l..m]
// Second subarray is arr[m+1..r]
void merge(int arr[], int l, int m, int r)
{
    int i, j, k;
    int n1 = m - l + 1;
    int n2 = r - m;

    // Create temp arrays
    int L[n1], R[n2];

    // Copy data to temp arrays L[] and R[]
    for (i = 0; i < n1; i++)
        L[i] = arr[l + i];
    for (j = 0; j < n2; j++)
```

```
        R[j] = arr[m + 1 + j];

    // Merge the temp arrays back into arr[l..r
    i = 0;
    j = 0;
    k = l;
    while (i < n1 && j < n2) {
        if (L[i] <= R[j]) {
            arr[k] = L[i];
            i++;
        }
        else {
            arr[k] = R[j];
            j++;
        }
        k++;
    }

    // Copy the remaining elements of L[],
    // if there are any
    while (i < n1) {
        arr[k] = L[i];
        i++;
        k++;
    }

    // Copy the remaining elements of R[],
    // if there are any
    while (j < n2) {
        arr[k] = R[j];
        j++;
        k++;
    }
}

// l is for left index and r is right index of the
// sub-array of arr to be sorted
void mergeSort(int arr[], int l, int r)
{
    if (l < r) {
        int m = l + (r - l) / 2;

        // Sort first and second halves
        mergeSort(arr, l, m);
        mergeSort(arr, m + 1, r);

        merge(arr, l, m, r);
    }
}

// Function to print an array
void printArray(int A[], int size)
{
    int i;
    for (i = 0; i < size; i++)
        printf("%d ", A[i]);
    printf("\n");
}

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

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

    mergeSort(arr, 0, arr_size - 1);

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

Output

```
Given array is

12 11 13 5 6 7
```

```
Sorted array is
5 6 7 11 12 13
```

Recurrence Relation of Merge Sort:

The recurrence relation of merge sort is:

$$T(n) = \begin{cases} \Theta(1) & \text{if } n = 1 \\ 2T\left(\frac{n}{2}\right) + \Theta(n) & \text{if } n > 1 \end{cases}$$

- T(n) Represents the total time time taken by the algorithm to sort an array of size n.

- 2T (n/2) represents time taken by the algorithm to recursively sort the two halves of the array. Since each half has n/2 elements, we have two recursive calls with input size as (n/2).

- O(n) represents the time taken to merge the two sorted halves

Complexity Analysis of Merge Sort:

- **Time Complexity:**
    - **Best Case:** O(n log n), When the array is already sorted or nearly sorted.
    - **Average Case:** O(n log n), When the array is randomly ordered.
    - **Worst Case:** O(n log n), When the array is sorted in reverse order.
- **Auxiliary Space:** O(n), Additional space is required for the temporary array used during merging.

Applications of Merge Sort:

- Sorting large datasets
- External sorting (when the dataset is too large to fit in memory)
- Inversion counting
- Merge Sort and its variations are used in library methods of programming languages.

For example its variation TimSort is used in Python, Java Android and Swift. The main reason why it is preferred to sort non-primitive types is stability which is not there in QuickSort.

For example Arrays.sort in Java uses QuickSort while Collections.sort uses MergeSort.

- It is a preferred algorithm for sorting **Linked lists**.
- It can be easily **parallelized** as we can independently sort subarrays and then merge.
- The merge function of merge sort to efficiently solve the problems like union and intersection of two sorted arrays.

Advantages of Merge Sort:

- **Stability** : Merge sort is a stable sorting algorithm, which means it maintains the relative order of equal elements in the input array.
- **Guaranteed worst-case performance:** Merge sort has a worst-case time complexity of **O(N logN)** , which means it performs well even on large datasets.
- **Simple to implement:** The divide-and-conquer approach is straightforward.
- **Naturally Parallel** : We independently merge subarrays that makes it suitable for parallel processing.

Disadvantages of Merge Sort:

- **Space complexity:** Merge sort requires additional memory to store the merged sub-arrays during the sorting process.
- **Not in-place:** Merge sort is not an in-place sorting algorithm, which means it requires additional memory to store the sorted data. This can be a disadvantage in applications where memory usage is a concern.
- S**lower than QuickSort in general**. QuickSort is more cache friendly because it works in-place.

Strassen’s Matrix Multiplication

Strassen’s Matrix Multiplication algorithm is a divide-and-conquer method for matrix multiplication, which reduces the number of scalar multiplications needed, making it more efficient than the conventional matrix multiplication method for large matrices. The traditional matrix multiplication has a time complexity of $O(n^3)$, while Strassen's algorithm reduces this to approximately $O(n^{2.81})$.

Here's a breakdown of the algorithm and how it works:

1. Problem Setup

Suppose we have two square matrices A and B, each of size $2^n \times 2^n$. If the matrices are not of size 2^n, we can pad them with zeros to make them a power of two. We want to compute their product matrix $C = A \times B$.

2. Divide the Matrices

Strassen’s algorithm involves dividing each matrix into four submatrices. For matrices A and B, we divide them as follows:

$$A = \begin{bmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{bmatrix}, \quad B = \begin{bmatrix} B_{11} & B_{12} \\ B_{21} & B_{22} \end{bmatrix}$$

Where:

- $A_{11}, A_{12}, A_{21}, A_{22}$ are submatrices of A.
- $B_{11}, B_{12}, B_{21}, B_{22}$ are submatrices of B.

Each submatrix is of size $n/2 \times n/2$.

3. Strassen's Formula

Instead of computing the eight matrix products required by standard matrix multiplication, Strassen’s algorithm reduces this to seven matrix products plus some additions and subtractions:

Define the following seven products:

1. $M_1 = (A_{11} + A_{22})(B_{11} + B_{22})$
2. $M_2 = (A_{21} + A_{22})B_{11}$
3. $M_3 = A_{11}(B_{12} - B_{22})$
4. $M_4 = A_{22}(B_{21} - B_{11})$
5. $M_5 = (A_{11} + A_{12})B_{22}$
6. $M_6 = (A_{21} - A_{11})(B_{11} + B_{12})$
7. $M_7 = (A_{12} - A_{22})(B_{21} + B_{22})$

Using these products, we can compute the resulting submatrices of C as follows:

$$C_{11} = M_1 + M_4 - M_5 + M_7$$

$$C_{12} = M_3 + M_5$$

$$C_{21} = M_2 + M_4$$

$$C_{22} = M_1 - M_2 + M_3 + M_6$$

4. Combine Results

After calculating C_{11}, C_{12}, C_{21}, and C_{22}, they are combined to form the resultant matrix C.

5. Recursive Application

Strassen's algorithm is applied recursively to the submatrices until the base case of 1×1 matrices is reached.

Complexity Analysis

Strassen's algorithm has a time complexity of approximately $O(n^{2.81})$, which is faster than the $O(n^3)$ of standard matrix multiplication for large matrices. However, the overhead of additions and subtractions and the recursive nature may not make it faster for very small matrices.

Advantages and Disadvantages

Advantages:

- Reduces the number of multiplications, which can be advantageous for large matrices.
- Asymptotically faster than conventional methods.

Disadvantages:

- More complex to implement.
- Overhead from addition/subtraction operations can outweigh the benefits for small matrices.
- Not optimal for sparse matrices where zeroes can be used to simplify computation.

Use Cases

Strassen's algorithm is beneficial for large, dense matrices and is used in various applications, such as scientific computing and large-scale data processing where matrix multiplication is common.

1. GENERAL METHOD

Greedy method: It is most straight forward method. It is popular for obtaining the optimized solutions.

Optimization Problem: An optimization problem is the problem of finding the best solution (optimal solution) from all the feasible solutions (practicable of possible solutions). In an optimization problem we are given a set of constraints and an optimization functions. Solutions that satisfy the constraints are called feasible solutions. A feasible solution for which the optimization function has the best possible value is called optimal solution.

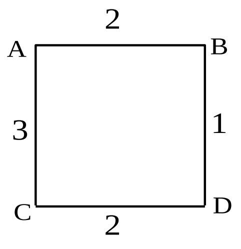
Ex: Problem: Finding a minimum spanning tree from a weighted connected directed graph G.

Constraints: Every time a minimum edge is added to the tree and adding of an edge does not form a simple circuit.

Feasible solutions: The feasible solutions are the spanning trees of the given graph G.

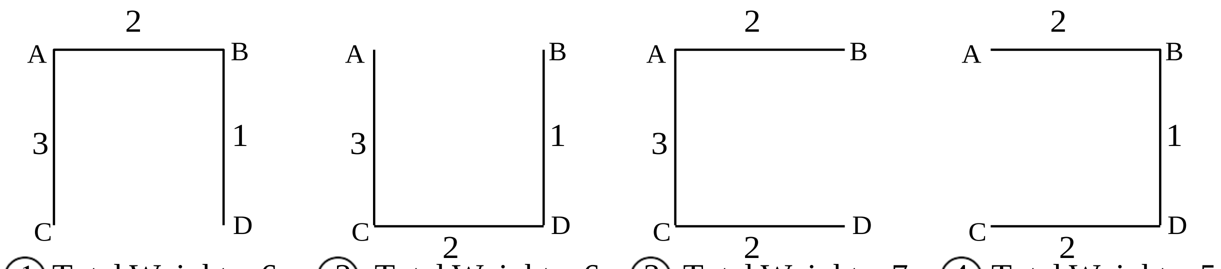
Optimal solution: An optimal solution is a spanning tree with minimum cost i.e. minimum spanning tree.

Q: Find the minimum spanning tree for the following graph.



Graph G

The feasible solutions are the spanning tree of the graph G. Those are



①.Total Weights=6 ② .Total Weights=6 ③.Total Weights=7 ④.Total Weights=5

From the above spanning tree the figure 4 gives the optimal solution, because it is the spanning tree with the minimum cost i.e. it is a minimum spanning tree of the graph G.

The greedy technique suggests constructing a solution to an optimization problem hrough a sequence of steps, each expanding a partially constructed solution obtained so far until a complete solution to the problem is reached to each step, the choice made must be feasible, locally optimal and irrecoverable.

Feasible: The choice which is made has to be satisfying the problems constraints.

Locally optimal: The choice has to be the best local choice among all feasible choices available on that step.

Irrecoverable: The choice once made cannot be changed on sub-sequent steps of the algorithm (Greedy method).

Control Abstraction for Greedy Method:

```
Algorithm GreedyMethod (a, n)
{
    // a is an array of n inputs
    Solution: =Ø;
    for i: =0 to n do
    {
            s: = select (a);
            if (feasible (Solution, s)) then
            {
                    Solution: = union (Solution, s);
            }
            else
                    reject (); // if solution is not feasible reject it.
    }
return solution;

}
```

In greedy method there are three important activities.

1. A selection of solution from the given input domain is performed, i.e. *s:= select(a)*.
2. The feasibility of the solution is performed, by using feasible '*(solution, s)*' and then all feasible solutions are obtained.
3. From the set of feasible solutions, the particular solution that minimizes or maximizes the given objection function is obtained. Such a solution is called optimal solution.

Change making example:

Suppose, we want to make change for an amount 'A' using fewest no of currency notes. Assume the available denominations are Rs 1, 2, 5, 10, 20, 50, 100, 200, 500, 1000.

To make a change for A=Rs 28, with the minimum number of notes, one would first choose anote of denomination Rs 20, 5, 2 and 1.

Denomination table					
for Rs 28		**for Rs 783**		**for Rs 3799**	
1000 X **0**	0	1000 X		1000 X	
500 X **0**	0	500 X		500 X	
200 X **0**	0	200 X		200 X	
100 X**0**	0	100 X		100 X	
50 X **0**	0	50 X		50 X	
20 X **1**	20	20 X		20 X	
10 X **0**	0	10 X		10 X	
5 X **1**	5	5 X		5 X	
2 X **1**	2	2 X		2 X	
1 X **1**	1	1 X		1 X	
Total	**28**	Total		Total	

```
Algorithm change making(denom_value[ ], TargetAmount)
{
    // denom={1000, 500, 200, 100, 50, 20, 10, 5, 2, 1}

    selected amount (SA) :=0;
    i :=1;
    while (SA < TargetAmount){
        if (SA + denom_value [i] <= TargetAmount ) {      // Select & Feasible

            denom_select[i] ++;  // Union
            SA := SA + denom_value[i];
        }
        else {
            i++;
        }
    }
    Print denom_select
}
```

Greedy Algorithm-General method

In Greedy method the problems have **'n'** inputs called as candidate set, from which a subset is selected to form a solution for the given problem. Any subset that satisfies the given constraintsis called a feasible solution. We need to find a feasible solution that maximizes or minimizes an objective function and such solution is called an optimal solution.

In the above ex currency notes denomination set { 1000 ….1000 ,500….500, 100….100,50…50, 20…20,10…10,5…5,2..2,1…1}is candidate set.

In the above ex constraint is our solution make the exact target amount of cash. Hence, any feasible solution i.e. sum of selected notes should be equal to target amount.

In the above ex objective function is our solution should consist of the fewest number of currency notes. Hence, any optimal solution which is one of the feasible solutions that optimizes the objective function. There can be more than one optimal solution.

Characteristics of Greedy:

1) These algorithms are simple and straightforward and easy to implement.

2) They take decisions on the basis of information at hand without worrying about the effect these decisions may have in the future.

3) They work in stages and never reconsider any decision.

APPLICATION - JOB SEQUENCING WITH DEADLINES

This problem consists of ***n*** jobs each associated with a deadline and profit and our objective is to earn ***maximum profit***. We will earn profit only when job is completed on or before deadline. We assume that each job will take ***unit time*** to complete.

Points to remember:

- In this problem we have n jobs j1, j2, … jn, each has an associated deadlines are d1, d2, … dn and profits are p1, p2, ... pn.
- Profit will only be awarded or earned if the job is completed on or before the deadline.
- We assume that each job takes unit time to complete.
- The objective is to earn maximum profit when only one job can be scheduled or processed at any given time.

Example: Consider the following 5 jobs and their associated deadline and profit.

index	1	2	3	4	5
JOB	j1	j2	j3	j4	j5
DEADLINE	2	1	3	2	1
PROFIT	60	100	20	40	20

Sort the jobs according to their profit in descending order.

Note! If two or more jobs are having the same profit then sorts them as per their entry in the job list.

index	1	2	3	4	5
JOB	j2	j1	j4	j3	j5
DEADLINE	1	2	2	3	1
PROFIT	100	60	40	20	20

Find the maximum deadline value

Looking at the jobs we can say the max deadline value is 3. So, dmax = 3

As dmax = 3 so we will have THREE slots to keep track of free time slots. Set the time slot status to EMPTY

time slot	1	2	3
status	EMPTY	EMPTY	EMPTY

Total number of jobs is 5. So we can write n = 5.
Note!

If we look at job j2, it has a deadline 1. This means we have to complete job j2 in time slot 1 if we want to earn its profit.

Similarly, if we look at job j1 it has a deadline 2. This means we have to complete job j1 on or before time slot 2 in order to earn its profit.

Similarly, if we look at job j3 it has a deadline 3. This means we have to complete job j3 on or before time slot 3 in order to earn its profit.

Our objective is to select jobs that will give us higher profit.

time slot	1	2	3
Job	J2	J1	J3
Profit	100	60	20

Total Profit is 180

Pseudo Code:

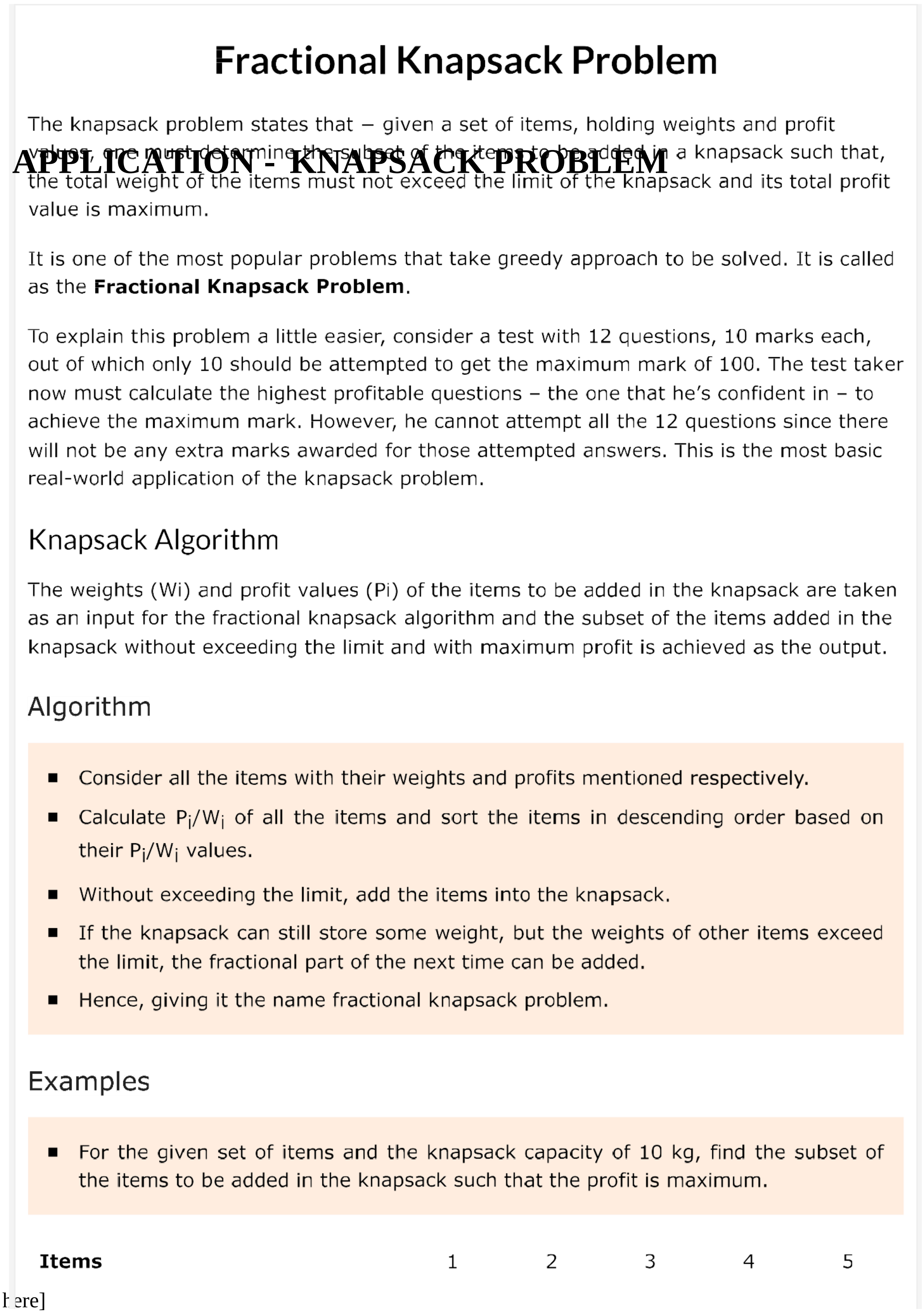
```
for i = 1 to n do
  Set k = min(dmax, DEADLINE(i))  //where DEADLINE(i) denotes deadline of ith job
  while k >= 1 do
    if timeslot[k] is EMPTY then
      timeslot[k] = job(i)
      break
    endif
        Set k = k - 1
        endwhile
        endfor
```

Algorithm:

```
int JS(int d[], int j[], int n)
// d[i]>=1, 1<=i<=n are the deadlines, n>=1. The jobs
// are ordered such that p[1]>=p[2]>= ... >=p[n]. J[i]
// is the ith job in the optimal solution, 1<=i<=k.
// Also, at termination d[J[i]]<=d[J[i+1]], 1<=i<k.
{
   d[0] = J[0] = 0; // Initialize.
   J[1] = 1; // Include job 1.
   int k=1;
   for (int i=2; i<=n; i++) {
   //Consider jobs in nonincreasing
   // order of p[i]. Find position for
   // i and check feasibility of insertion.
      int r = k;
      while ((d[J[r]] > d[i]) && (d[J[r]] != r)) r--;
      if ((d[J[r]] <= d[i]) && (d[i] > r)) {
         // Insert i into J[].
         for (int q=k; q>=(r+1); q--) J[q+1] = J[q];
         J[r+1] = i; k++;
      }
   }
   return (k);
}
```

Time Complexity = O (n^2)

APPLICATION - KNAPSACK PROBLEM



Fractional Knapsack Problem

The knapsack problem states that − given a set of items, holding weights and profit values, one must determine the subset of the items to be added in a knapsack such that, the total weight of the items must not exceed the limit of the knapsack and its total profit value is maximum.

It is one of the most popular problems that take greedy approach to be solved. It is called as the **Fractional Knapsack Problem**.

To explain this problem a little easier, consider a test with 12 questions, 10 marks each, out of which only 10 should be attempted to get the maximum mark of 100. The test taker now must calculate the highest profitable questions – the one that he's confident in – to achieve the maximum mark. However, he cannot attempt all the 12 questions since there will not be any extra marks awarded for those attempted answers. This is the most basic real-world application of the knapsack problem.

Knapsack Algorithm

The weights (Wi) and profit values (Pi) of the items to be added in the knapsack are taken as an input for the fractional knapsack algorithm and the subset of the items added in the knapsack without exceeding the limit and with maximum profit is achieved as the output.

Algorithm

- Consider all the items with their weights and profits mentioned respectively.
- Calculate P_i/W_i of all the items and sort the items in descending order based on their P_i/W_i values.
- Without exceeding the limit, add the items into the knapsack.
- If the knapsack can still store some weight, but the weights of other items exceed the limit, the fractional part of the next time can be added.
- Hence, giving it the name fractional knapsack problem.

Examples

- For the given set of items and the knapsack capacity of 10 kg, find the subset of the items to be added in the knapsack such that the profit is maximum.

Items	1	2	3	4	5

Weights (in kg)	3	3	2	5	1
Profits	10	15	10	12	8

Solution

Step 1

Given, n = 5

```
Wi = {3, 3, 2, 5, 1}
Pi = {10, 15, 10, 12, 8}
```

Calculate P_i/W_i for all the items

Items	1	2	3	4	5	
Weights (in kg)	3	3	2	5	1	
Profits	10	15	10	20	8	
P_i/W_i	3.3	5	5	4	8	

Step 2

Arrange all the items in descending order based on P_i/W_i

Items	5	2	3	4	1
Weights (in kg)	1	3	2	5	3
Profits	8	15	10	20	10
P_i/W_i	8	5	5	4	3.3

Step 3

Without exceeding the knapsack capacity, insert the items in the knapsack with maximum profit.

```
Knapsack = {5, 2, 3}
```

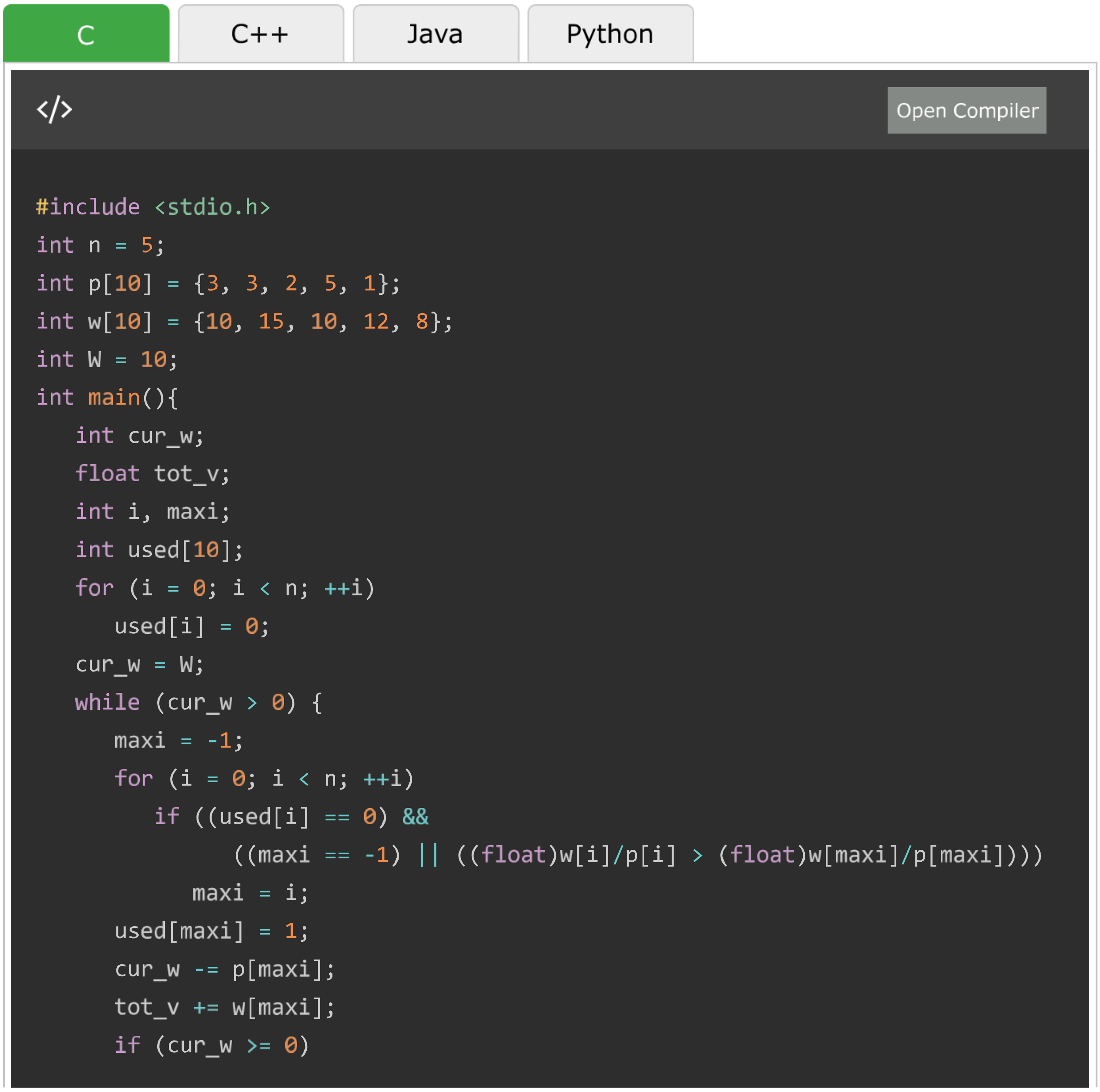
However, the knapsack can still hold 4 kg weight, but the next item having 5 kg weight will exceed the capacity. Therefore, only 4 kg weight of the 5 kg will be added in the knapsack.

Items	5	2	3	4	1
Weights (in kg)	1	3	2	5	3
Profits	8	15	10	20	10
Knapsack	1	1	1	4/5	0

Hence, the knapsack holds the weights = [(1 * 1) + (1 * 3) + (1 * 2) + (4/5 * 5)] = 10, with maximum profit of [(1 * 8) + (1 * 15) + (1 * 10) + (4/5 * 20)] = 37.

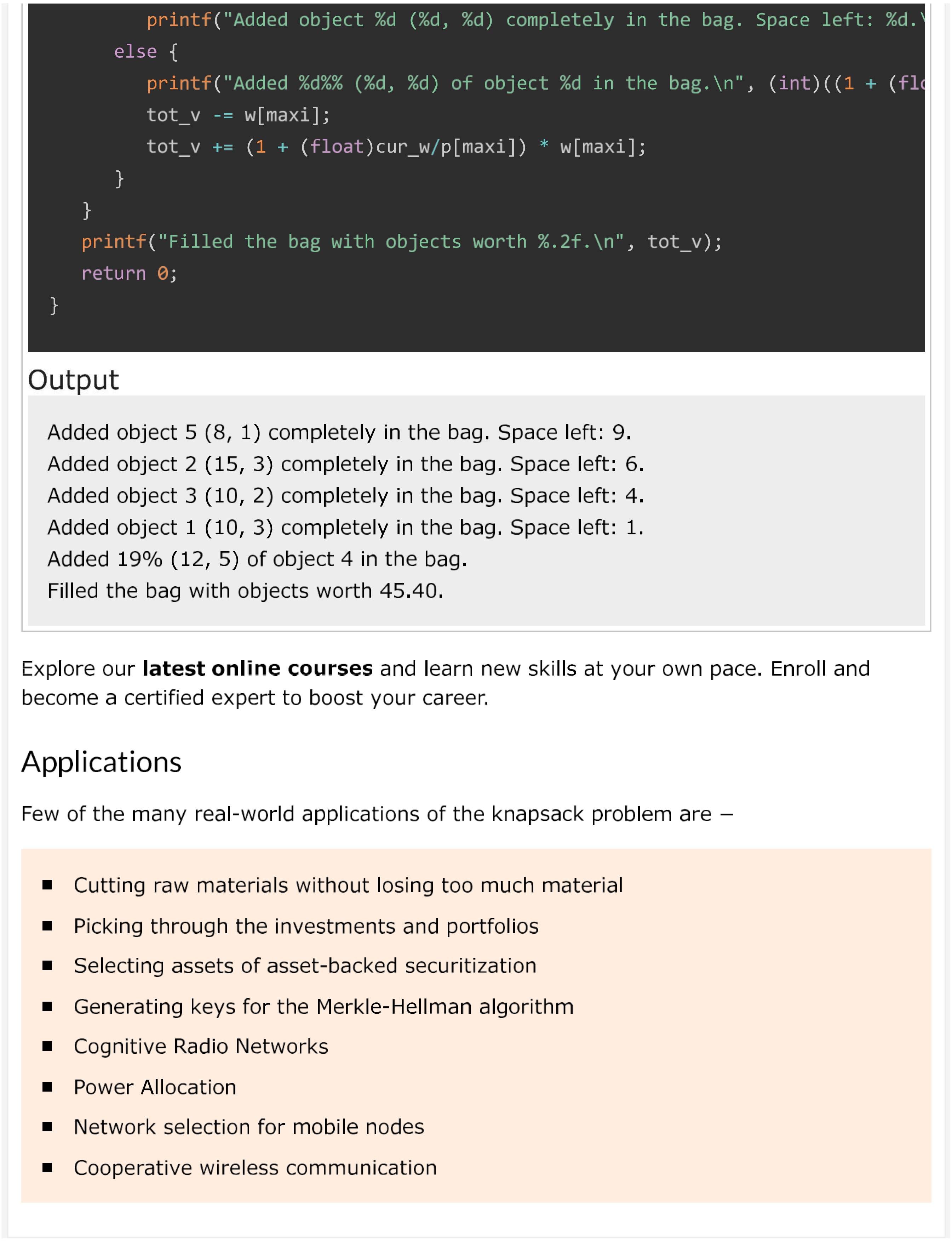
Example

Following is the final implementation of Fractional Knapsack Algorithm using Greedy Approach −



```c
#include <stdio.h>
int n = 5;
int p[10] = {3, 3, 2, 5, 1};
int w[10] = {10, 15, 10, 12, 8};
int W = 10;
int main(){
   int cur_w;
   float tot_v;
   int i, maxi;
   int used[10];
   for (i = 0; i < n; ++i)
      used[i] = 0;
   cur_w = W;
   while (cur_w > 0) {
      maxi = -1;
      for (i = 0; i < n; ++i)
         if ((used[i] == 0) &&
               ((maxi == -1) || ((float)w[i]/p[i] > (float)w[maxi]/p[maxi])))
            maxi = i;
      used[maxi] = 1;
      cur_w -= p[maxi];
      tot_v += w[maxi];
      if (cur_w >= 0)
```

[Type here]



```
         printf("Added object %d (%d, %d) completely in the bag. Space left: %d.
      else {
         printf("Added %d%% (%d, %d) of object %d in the bag.\n", (int)((1 + (fl
         tot_v -= w[maxi];
         tot_v += (1 + (float)cur_w/p[maxi]) * w[maxi];
      }
   }
   printf("Filled the bag with objects worth %.2f.\n", tot_v);
   return 0;
}
```

Output

```
Added object 5 (8, 1) completely in the bag. Space left: 9.
Added object 2 (15, 3) completely in the bag. Space left: 6.
Added object 3 (10, 2) completely in the bag. Space left: 4.
Added object 1 (10, 3) completely in the bag. Space left: 1.
Added 19% (12, 5) of object 4 in the bag.
Filled the bag with objects worth 45.40.
```

Explore our **latest online courses** and learn new skills at your own pace. Enroll and become a certified expert to boost your career.

Applications

Few of the many real-world applications of the knapsack problem are −

- Cutting raw materials without losing too much material
- Picking through the investments and portfolios
- Selecting assets of asset-backed securitization
- Generating keys for the Merkle-Hellman algorithm
- Cognitive Radio Networks
- Power Allocation
- Network selection for mobile nodes
- Cooperative wireless communication

This code is an implementation of a greedy algorithm for the fractional knapsack problem, where we aim to maximize the value of items placed in a bag with a weight limit. Here's a breakdown of the main steps in the program:

1. **Initialization of Values**:
   - n is the number of items (5 in this case).
   - Arrays p and w hold the weights and values of the items, respectively.
   - W is the weight capacity of the bag (10).

2. **Tracking Usage**:
   - used is an array that tracks whether an item has been fully or partially used.
   - cur_w keeps track of the current remaining weight capacity in the bag.
   - tot_v accumulates the total value of items in the bag.

3. **Greedy Selection**:
   - The algorithm iterates until cur_w (remaining capacity) is zero or negative.
   - For each step, it selects the item with the highest value-to-weight ratio that hasn't been fully used.
   - If the selected item can fit entirely within cur_w, it's fully added to the bag.
   - If it can't fit, a fraction of the item proportional to the remaining capacity is added.

4. **Output**:
   - After each addition, the program prints the status.
   - At the end, it outputs the total value of the items in the bag.

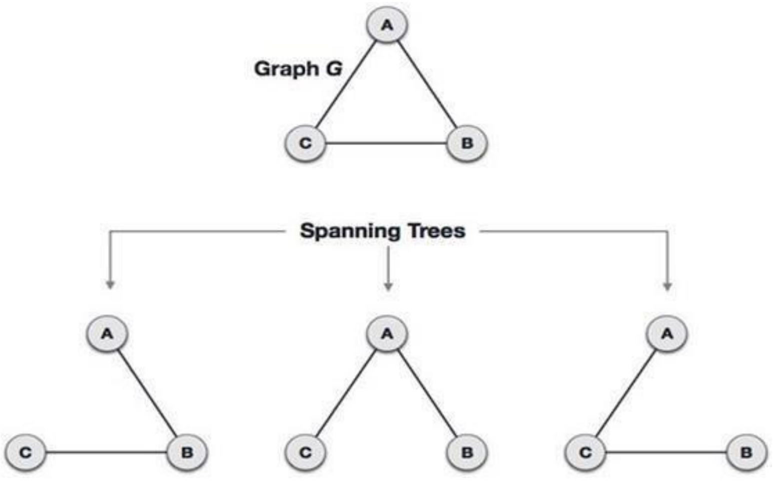
Output:

```
Added object 5 (8, 1) completely in the bag. Space left: 9.
Added object 2 (15, 3) completely in the bag. Space left: 6.
Added object 3 (10, 2) completely in the bag. Space left: 4.
Added object 1 (10, 3) completely in the bag. Space left: 1.
Added 20% (12, 5) of object 4 in the bag.
Filled the bag with objects worth 45.
```

MINIMUM SPANNING TREE

A spanning tree is a subset of Graph G, which has all the vertices covered with minimum possible number of edges. Hence, a spanning tree does not have cycles and it cannot be disconnected.

Note: Every connected and undirected Graph G has at least one spanning tree. A disconnected graph does not have any spanning tree.



We found three spanning trees off one complete graph. A complete undirected graph can have maximum n^{n-2} number of spanning trees, where n is the number of nodes. In the above addressed example, $3^{3-2} = 3$ spanning trees are possible.

General Properties of Spanning Tree

- A connected graph G can have more than one spanning tree.
- All possible spanning trees of graph G, have the same number of edges and vertices.
- The spanning tree does not have any cycle (loops).
- Removing one edge from the spanning tree will make the graph disconnected, i.e. the spanning tree is minimally connected.
- Adding one edge to the spanning tree will create a circuit or loop, i.e. the spanning tree is maximally acyclic.

Mathematical Properties of Spanning Tree

- Spanning tree has n-1 edges, where n is the number of nodes (vertices).
- From a complete graph, by removing maximum e - n + 1 edges, we can construct a spanning tree.
- A complete graph can have maximum nn-2 number of spanning trees.

Thus, we can conclude that spanning trees are a subset of connected Graph G and disconnected graphs do not have spanning tree.

Application of Spanning Tree

Spanning tree is basically used to find a minimum path to connect all nodes in a graph. Common applications of spanning trees are

- Civil Network Planning
- Computer Network Routing Protocol
- Cluster Analysis

Let us understand this through a small example. Consider, city network as a huge graph and now plans to deploy telephone lines in such a way that in minimum lines we can connect to all city nodes. This is where the spanning tree comes into picture.

Minimum Spanning Tree (MST)

In a weighted graph, a minimum spanning tree is a spanning tree that has minimum weight than all other spanning trees of the same graph. In real-world situations, this weight can be measured as distance, congestion, traffic load or any arbitrary value denoted to the edges.

Minimum Spanning-Tree Algorithm

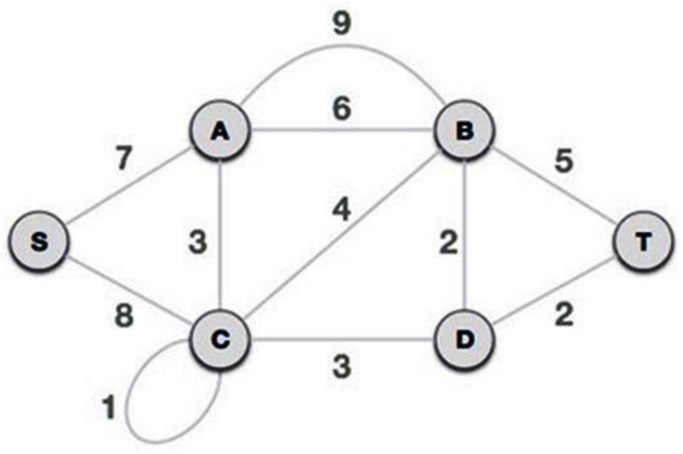
We shall learn about two most important spanning tree algorithms(greedy algorithms):

1. Kruskal's Algorithm
2. Prim's Algorithm

i. Kruskal's Algorithm

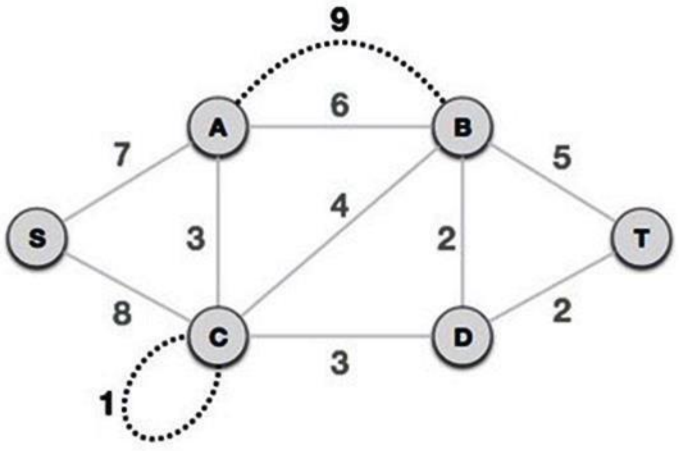
Kruskal's algorithm to find the minimum cost spanning tree uses the greedy approach. This algorithm treats the graph as a forest and every node it has as an individual tree. A tree connects to another only and only if, it has the least cost among all available options and does not violate MST properties.

To understand Kruskal's algorithm let us consider the following example:

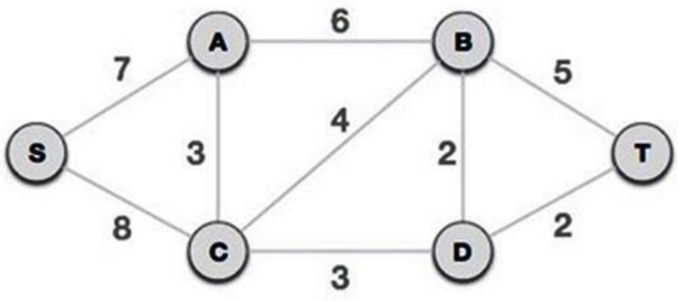


Step 1 - Remove all loops and Parallel Edges

Remove all loops and parallel edges from the given graph.



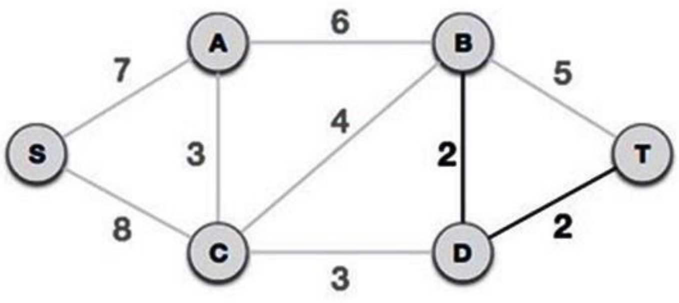
In case of parallel edges, keep the one which has the least cost associated and remove all others.



Step 2 - Arrange all edges in their increasing order of weight
The next step is to create a set of edges and weight, and arrange them in an ascending order of weightage (cost).

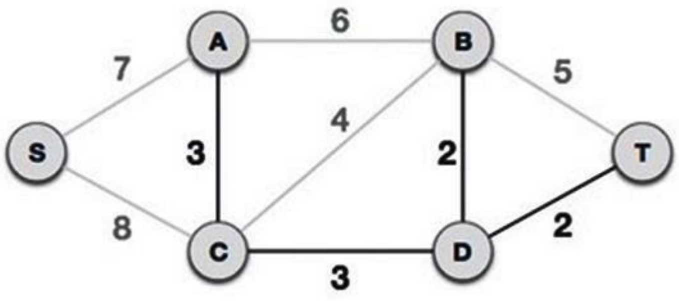
B, D	D, T	A, C	C, D	C, B	B, T	A, B	S, A	S, C
2	2	3	3	4	5	6	7	8

Step 3 - Add the edge which has the least weightage
Now we start adding edges to the graph beginning from the one which has the least weight. Throughout, we shall keep checking that the spanning properties remain intact. In case, by adding one edge, the spanning tree property does not hold then we shall consider not to include the edge in the graph.

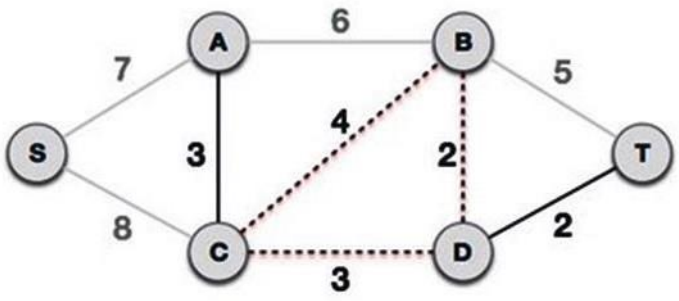


The least cost is 2 and edges involved are B,D and D,T. We add them. Adding them does not violate spanning tree properties, so we continue to our next edge selection.

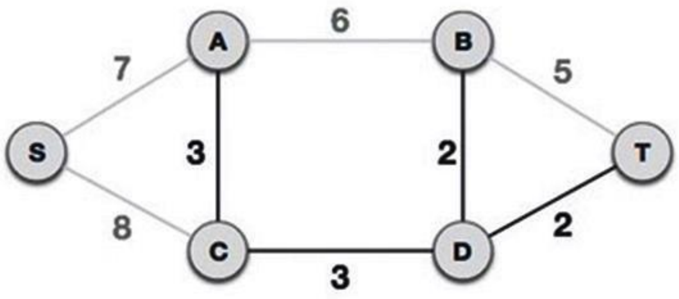
Next cost is 3, and associated edges are A,C and C,D. We add them again −



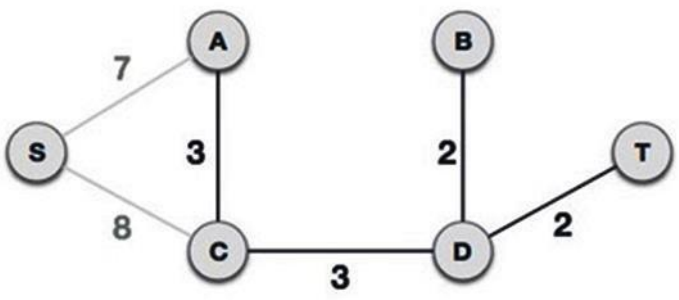
Next cost in the table is 4, and we observe that adding it will create a circuit in the graph. −



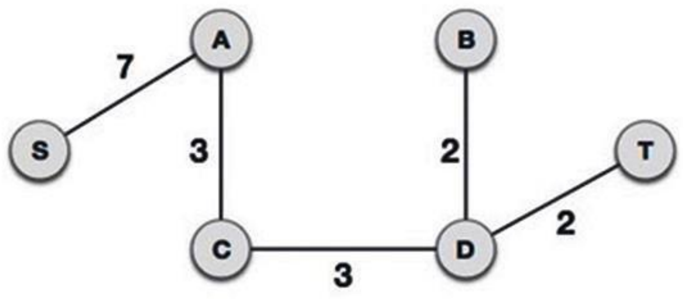
We ignore it. In the process we shall ignore/avoid all edges that create a circuit.



We observe that edges with cost 5 and 6 also create circuits. We ignore them and move on.



Now we are left with only one node to be added. Between the two least cost edges available 7 and 8, we shall add the edge with cost 7.



By adding edge S,A we have included all the nodes of the graph and we now have minimum cost spanning tree.

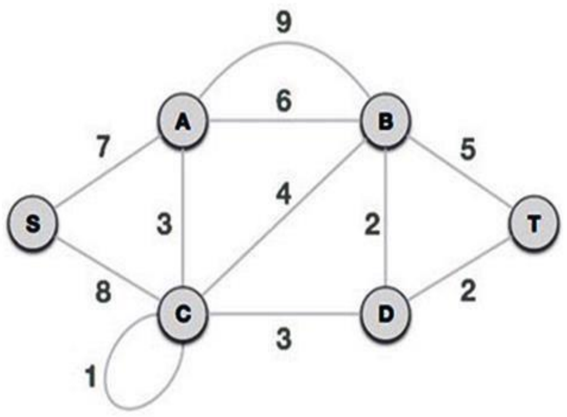
```
float Kruskal(int E[][SIZE],  float cost[][SIZE],  int n,  int t[][2])
{
    int parent[SIZE];
    construct a heap out of the edge costs using Heapify;
    for (int i=1; i<=n; i++)  parent[i] = -1;
    //  Each vertex is in a different set.
    i = 0; float mincost = 0.0;
    while ((i < n-1) && (heap not empty)) {
        delete a minimum cost edge (u,v) from the heap
        and reheapify using Adjust;
        int j = Find(u); int k = Find(v);
        if  (j != k)  {
              i++;
              t[i][1] = u;  y[i][2] = v;
              mincost += cost[u][v];
              Union(j, k);
        }
    }
     if  ( i !=  n-1) cout << "No spanning tree" << endl;
     else  return(mincost);
}
```

Time Complexity = O (|E| log |E|)

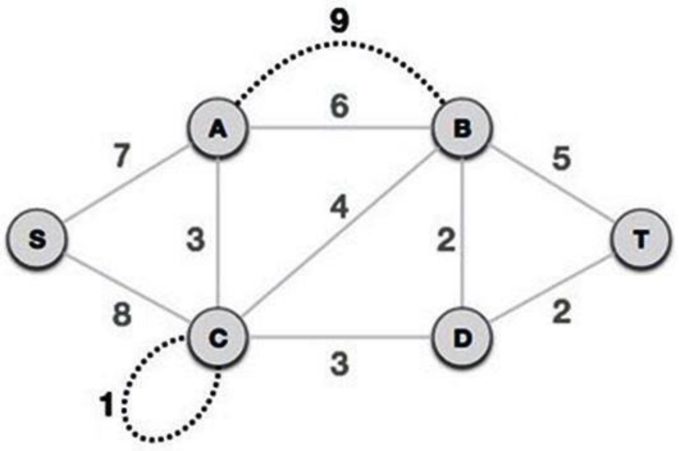
ii. Prim's Algorithm

Prim's algorithm to find minimum cost spanning tree (as Kruskal's algorithm) uses the greedy approach. Prim's algorithm shares a similarity with the shortest path first algorithms.

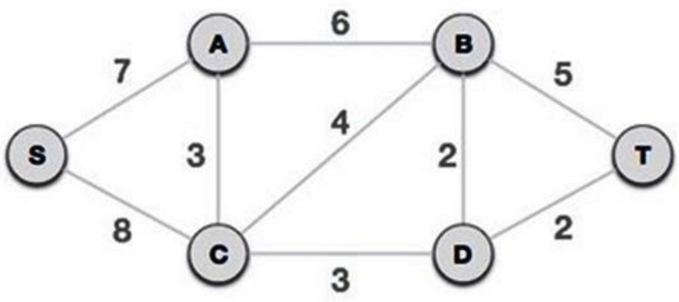
Prim's algorithm, in contrast with Kruskal's algorithm, treats the nodes as a single tree and keeps on adding new nodes to the spanning tree from the given graph. To contrast with Kruskal's algorithm and to understand Prim's algorithm better, we shall use the same example.



Step 1 - Remove all loops and parallel edges



Remove all loops and parallel edges from the given graph. In case of parallel edges, keep the one which has the least cost associated and remove all others.

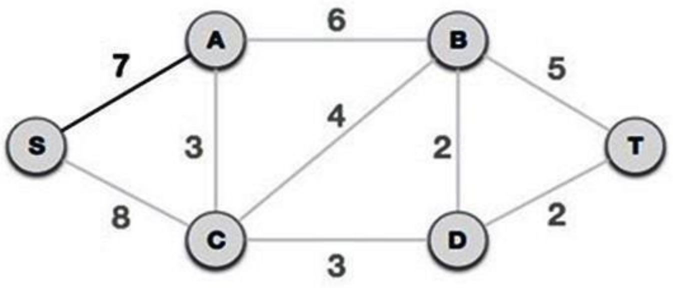


Step 2 - Choose any arbitrary node as root node

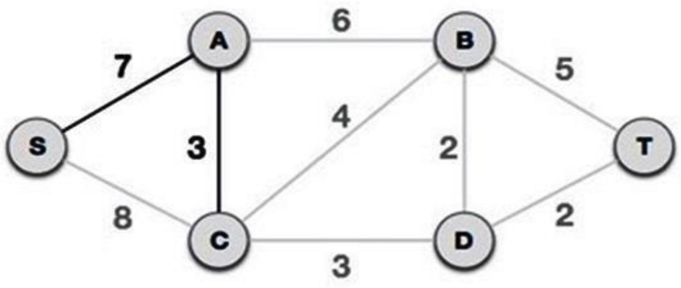
In this case, we choose S node as the root node of Prim's spanning tree. This node is arbitrarily chosen, so any node can be the root node. One may wonder why any video can be a root node. So the answer is, in the spanning tree all the nodes of a graph are included and because it is connected then there must be at least one edge, which will join it to the rest of the tree.

Step 3 - Check outgoing edges and select the one with less cost

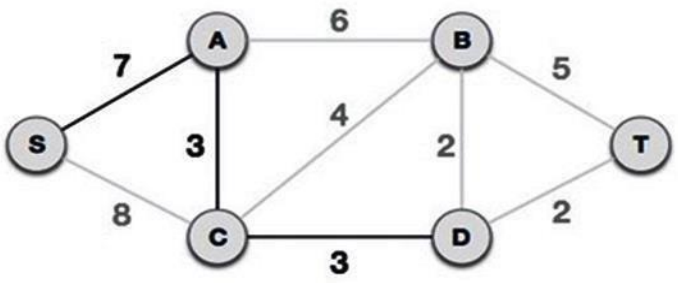
After choosing the root node S, we see that S,A and S,C are two edges with weight 7 and 8, respectively. We choose the edge S,A as it is lesser than the other.



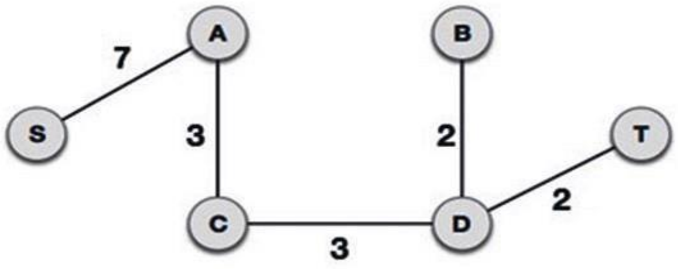
Now, the tree S-7-A is treated as one node and we check for all edges going out from it. We select the one which has the lowest cost and include it in the tree.



After this step, S-7-A-3-C tree is formed. Now we'll again treat it as a node and will check all the edges again. However, we will choose only the least cost edge. In this case, C-3-D is the new edge, which is less than other edges' cost 8, 6, 4, etc.



After adding node D to the spanning tree, we now have two edges going out of it having the same cost, i.e. D-2-T and D-2-B. Thus, we can add either one. But the next step will again yield edge 2 as the least cost. Hence, we are showing a spanning tree with both edges included.



We may find that the output spanning tree of the same graph using two different algorithms is same.

```
float Prim(int E[][SIZE], float cost[][SIZE],  int n, int t[][2])
{
  int near[SIZE], j, k, L;
  let (k,L) be an edge of minimum cost in E;
  float mincost = cost[k][L];
  t[1][1] = k; t[1][2] = L;
  for (int i=1; i<=n; i++)  // Initialize near.
    if (cost[i][L] < cost[i][k]) near[i] = L;
    else near[i] = k;
  near[k] = near[L] = 0;
  for (i=2; i <= n-1; i++) { // Find n-2 additional
                             // edges for t.
    let j be an index such that near[j]!=0 and
    cost[j][near[j]] is minimum;
    t[i][1] = j; t[i][2] = near[j];
    mincost = mincost + cost[j][near[j]];
    near[j]=0;
    for (k=1; k<=n; k++) // Update near[].
      if ((near[k]!=0) &&
          (cost[k][near[k]]>cost[k][j]))
        near[k] = j;
  }
   return(mincost);
}
```

Time Complexity = O (n^2)

single-source shortest path algorithm

Dijkstra's shortest path algorithm is similar to that of Prim's algorithm as they both rely on finding the shortest path locally to achieve the global solution. However, unlike prim's algorithm, the dijkstra's algorithm does not find the minimum spanning tree; it is designed to find the shortest path in the graph from one vertex to other remaining vertices in the graph. Dijkstra's algorithm can be performed on both directed and undirected graphs.

Since the shortest path can be calculated from single source vertex to all the other vertices in the graph, Dijkstra's algorithm is also called **single-source shortest path algorithm**. The output obtained is called **shortest path spanning tree**.

Dijkstra's Algorithm

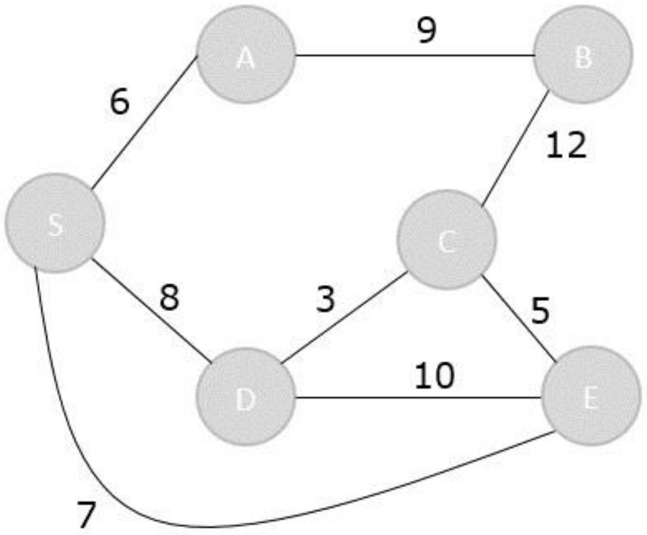
The dijkstra's algorithm is designed to find the shortest path between two vertices of a graph. These two vertices could either be adjacent or the farthest points in the graph. The algorithm starts from the source. The inputs taken by the algorithm are the graph G {V, E}, where V is the set of vertices and E is the set of edges, and the source vertex S. And the output is the shortest path spanning tree.

Algorithm

- Declare two arrays − *distance*[] to store the distances from the source vertex to the other vertices in graph and *visited*[] to store the visited vertices.
- Set distance[S] to '0' and distance[v] = ∞, where v represents all the other vertices in the graph.
- Add S to the visited[] array and find the adjacent vertices of S with the minimum distance.
- The adjacent vertex to S, say A, has the minimum distance and is not in the visited array yet. A is picked and added to the visited array and the distance of A is changed from ∞ to the assigned distance of A, say d_1, where $d_1 < \infty$.
- Repeat the process for the adjacent vertices of the visited vertices until the shortest path spanning tree is formed.

Examples

To understand the dijkstra's concept better, let us analyze the algorithm with the help of an example graph −



Step 1

Initialize the distances of all the vertices as ∞, except the source node S.

Vertex	S	A	B	C	D	E
Distance	0	∞	∞	∞	∞	∞

Now that the source vertex S is visited, add it into the visited array.

```
visited = {S}
```

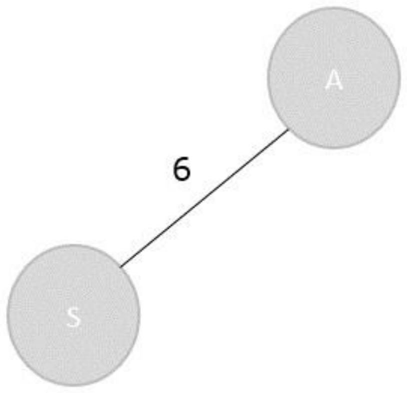
Step 2

The vertex S has three adjacent vertices with various distances and the vertex with minimum distance among them all is A. Hence, A is visited and the dist[A] is changed from ∞ to 6.

```
S → A = 6
S → D = 8
S → E = 7
```

Vertex	S	A	B	C	D	E
Distance	0	6	∞	∞	8	7

```
Visited = {S, A}
```



Step 3

There are two vertices visited in the visited array, therefore, the adjacent vertices must be checked for both the visited vertices.

Vertex S has two more adjacent vertices to be visited yet: D and E. Vertex A has one adjacent vertex B.

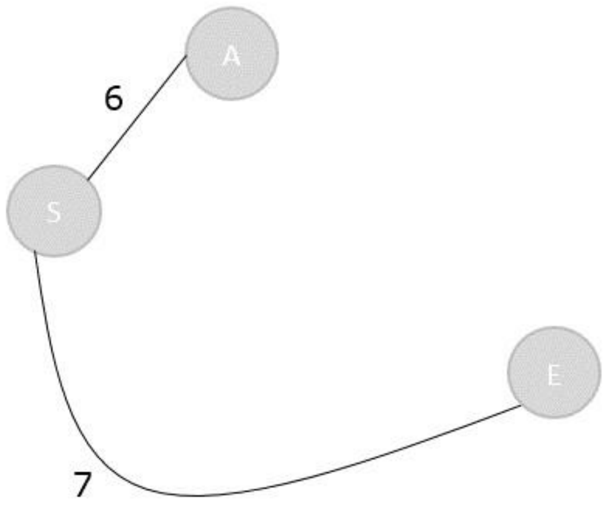
Calculate the distances from S to D, E, B and select the minimum distance −

$S \rightarrow D = 8$ and $S \rightarrow E = 7$.

$S \rightarrow B = S \rightarrow A + A \rightarrow B = 6 + 9 = 15$

Vertex	S	A	B	C	D	E
Distance	0	6	15	∞	8	7

Visited = {S, A, E}



Step 4

Calculate the distances of the adjacent vertices – S, A, E – of all the visited arrays and select the vertex with minimum distance.

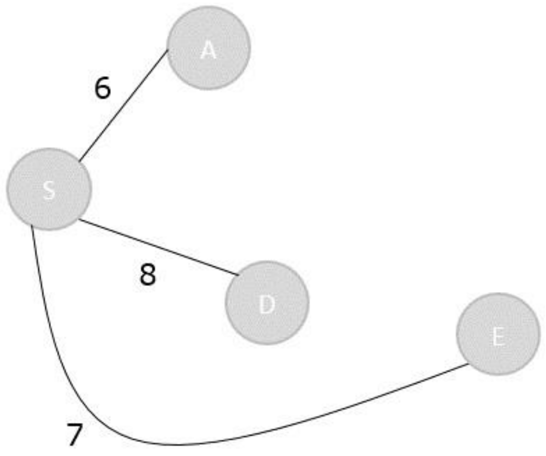
S → D = 8

S → B = 15

S → C = S → E + E → C = 7 + 5 = 12

Vertex	S	A	B	C	D	E
Distance	0	6	15	12	8	7

Visited = {S, A, E, D}



Step 5

Recalculate the distances of unvisited vertices and if the distances minimum than existing distance is found, replace the value in the distance array.

S → C = S → E + E → C = 7 + 5 = 12

S → C = S → D + D → C = 8 + 3 = 11

dist[C] = minimum (12, 11) = 11

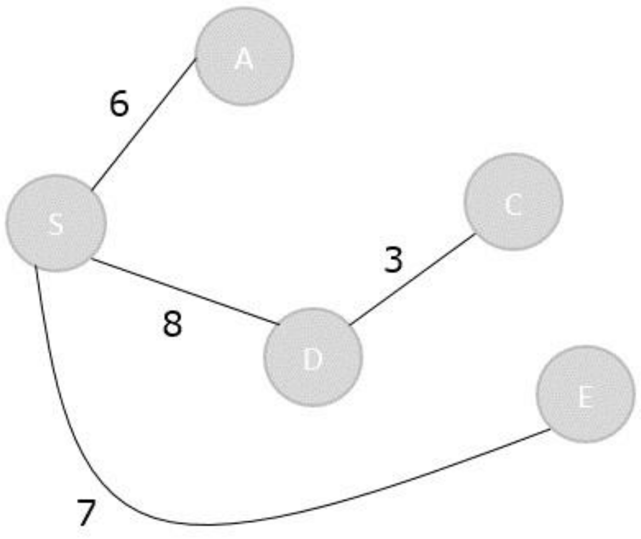
S → B = S → A + A → B = 6 + 9 = 15

S → B = S → D + D → C + C → B = 8 + 3 + 12 = 23

dist[B] = minimum (15,23) = 15

Vertex	S	A	B	C	D	E
Distance	0	6	15	11	8	7

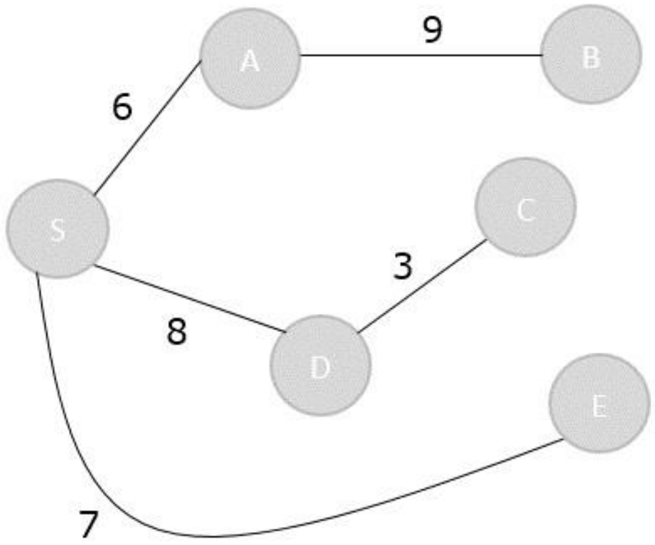
Visited = { S, A, E, D, C}



Step 6

The remaining unvisited vertex in the graph is B with the minimum distance 15, is added to the output spanning tree.

Visited = {S, A, E, D, C, B}



The shortest path spanning tree is obtained as an output using the dijkstra’s algorithm.

[Type here]

Example

The program implements the dijkstra’s shortest path problem that takes the cost adjacency matrix as the input and prints the shortest path as the output along with the minimum cost.

```
#include<stdio.h>

#include<limits.h>

#include<stdbool.h>

int min_dist(int[], bool[]);

void greedy_dijsktra(int[][6],int);

int min_dist(int dist[], bool visited[]){ // finding minimum dist

   int minimum=INT_MAX,ind;

   for(int k=0; k<6; k++) {

      if(visited[k]==false && dist[k]<=minimum) {

         minimum=dist[k];

         ind=k;

      }

   }

   return ind;

}

void greedy_dijsktra(int graph[6][6],int src){

   int dist[6];

   bool visited[6];

   for(int k = 0; k<6; k++) {

      dist[k] = INT_MAX;

      visited[k] = false;

   }
```

```
  dist[src] = 0; // Source vertex dist is set 0

  for(int k = 0; k<6; k++) {

    int m=min_dist(dist,visited);

    visited[m]=true;

    for(int k = 0; k<6; k++) {



      // updating the dist of neighbouring vertex

      if(!visited[k] && graph[m][k] && dist[m]!=INT_MAX &&
dist[m]+graph[m][k]<dist[k])

        dist[k]=dist[m]+graph[m][k];

    }

  }

  printf("Vertex\t\tdist from source vertex\n");

  for(int k = 0; k<6; k++) {

    char str=65+k;

    printf("%c\t\t\t%d\n", str, dist[k]);

  }

}

int main(){

  int graph[6][6]= {

    {0, 1, 2, 0, 0, 0},

    {1, 0, 0, 5, 1, 0},

    {2, 0, 0, 2, 3, 0},

    {0, 5, 2, 0, 2, 2},

    {0, 1, 3, 2, 0, 1},
    {0, 0, 0, 2, 1, 0}

  };

  greedy_dijsktra(graph,0);

  return 0;

}
```

Output

```
Vertex            dist from source vertex
A                           0
B                           1
C                           2
D                           4
E                           2
F                           3
```

GREEDY APPROACH	DIVIDE AND CONQUER
1.Many decisions and sequences areguaranteed and all the overlapping subinstancesare considered.	1.Divide the given problem into many subproblems.Find the individual solutions andcombine them to get the solution for themain problem
2. Follows Bottom-up technique	2. Follows top down technique
3.Split the input at every possible pointsrather than at a particular point	3.Split the input only at specific points (midpoint), each problem is independent.
4. Sub problems are dependent on the main Problem	4. Sub problems are independent on the main Problem
5. Time taken by this approach is not that much efficient when compared with DAC.	5. Time taken by this approach efficient when compared with GA.
6.Space requirement is less when compared DAC approach.	6.Space requirement is very much high when compared GA approach.