UNIT–III:

Divide and Conquer: The General Method, Quick Sort, Merge Sort, Strassen's matrix multiplication.

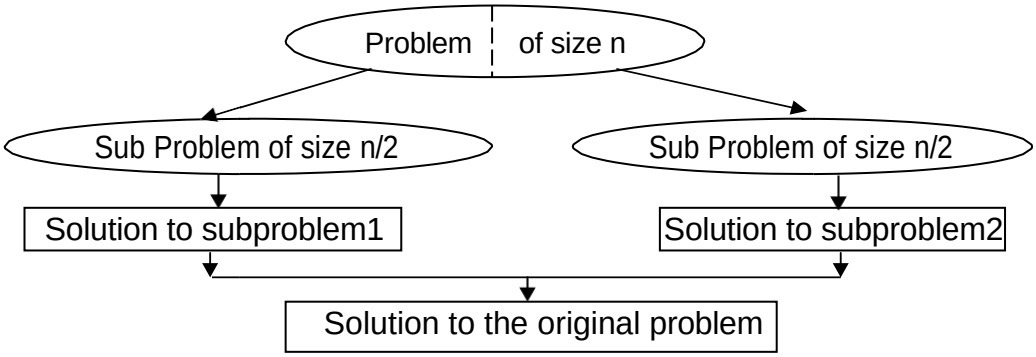
Greedy Method: General Method, Job Sequencing with deadlines, Knapsack Problem, Minimum cost spanning trees, Single Source Shortest Paths

GENERAL METHOD

Divide and Conquer is one of the best-known general algorithm design technique. It works according to the following general plan:

- Given a function to compute on 'n' inputs the divide-and-conquer strategy suggests splitting the inputs into 'k' distinct subsets, 1<k<=n, yielding 'k' sub problems.
- These sub problems must be solved, and then a method must be found to combine sub solutions into a solution of the whole.
- If the sub problems are still relatively large, then the divide-and-conquer strategy can possibly be reapplied.
- Often the sub problems resulting from a divide-and-conquer design are of the same type as the original problem. For those cases the reapplication of the divide-and- conquer principle is naturally expressed by a recursive algorithm.

A typical case with k=2 is diagrammatically shown below.



Control Abstraction for divide and conquer:

```
Algorithm DAndC(P)
{
    if Small(P) then return S(P);
    else
    {
        divide P into smaller instances P1, P2, ..., Pk, k ≥ 1;
        Apply DAndC to each of these subproblems;
        return Combine(DAndC(P1),DAndC(P2),...,DAndC(Pk));
    }
}
```

In the above specification,

- Initially ***DAndC(P)*** is invoked, where 'P' is the problem to be solved.
- ***Small (P)*** is a Boolean-valued function that determines whether the input size is small enough that the answer can be computed without splitting. If this so, the function '***S'*** is invoked. Otherwise, the problem P is divided into smaller sub problems. These sub problems P_1, P_2…P_k are solved by recursive application of ***DAndC***.
- ***Combine*** is a function that determines the solution to P using the solutions to the 'k' sub problems.

2. Recurrence equation for Divide and Conquer

If the size of problem 'p' is n and the sizes of the 'k' sub problems are $n_1, n_2 \ldots n_k$, respectively, then the computing time of divide and conquer is described by the recurrence relation

Where, $$T(n) = \begin{cases} g(n) & n \text{ small} \\ T(n_1) + T(n_2) + \cdots + T(n_k) + f(n) & \text{otherwise} \end{cases}$$

- T(n) is the time for divide and conquer method on any input of size n and
- g(n) is the time to compute answer directly for small inputs.
- The function f(n) is the time for dividing the problem 'p' and combining the solutions to sub problems.

For divide and conquer based algorithms that produce sub problems of the same type as the original problem, it is very natural to first describe them by using recursion.

More generally, an instance of size **n** can be divided into **b** instances of size **n/b**, with **a** of them needing to be solved. (Here, a and b are constants; **a>=1 and b > 1**.). Assuming that size **n** is a power of **b**(i.e. $n=b^k$), to simplify our analysis, we get the following recurrence for the running time T(n):.(1)

Where f(n) is a function t $T(n) = \begin{cases} T(1) & n = 1 \\ aT(n/b) + f(n) & n > 1 \end{cases}$ ıg the problem into smaller ones and on combining their sol

The recurrence relation can Be solved by i) substitution method or by using ii) master theorem.

1. **Substitution Method-**This method repeatedly makes substitution for each Occurrence of the function T in the right-hand side until all such occurrences disappears.
2. **Master Theorem**-The efficiency analysis of many divide-and-conquer algorithms is greatly simplified by the master theorem. It states that, in recurrence equation **T(n) = aT(n/b) + f (n)**, If f (n)$\in \Theta(n^d)$ where $d \geq 0$ then

$$T(n) \in \begin{cases} \Theta(n^d) & \text{if } a < b^d, \\ \Theta(n^d \log n) & \text{if } a = b^d, \\ \Theta(n^{\log_b a}) & \text{if } a > b^d. \end{cases}$$

Analogous results hold for the *O* and Ω notations, too.

For example, the recurrence for the number of additions A(n) made by the divide- and-conquer sum-computation algorithm (see above) on inputs of size $n = 2^k$ is

$$A(n) = 2A(n/2) + 1.$$

Thus, for this example, a=2, b =2, and d=0; hence, since $a>b^d$,

$$A(n) \in \Theta(n^{\log_b a}) = \Theta(n^{\log_2 2}) = \Theta(n).$$

Problems on Substitution method & Master theorem to solve the recurrence relation

Exercise 3.1 Horowitz

Solve following recurrence relation. $T(n) = 2T(n/2) + n$, $T(1) = 2$. Using substitution method

Soln: $T(n) = 2T(n/2) + n$.

$= 2[2 \cdot T(\frac{n}{4}) + \frac{n}{2}] + n = 4T(\frac{n}{4}) + 2n$

$= 4[2 \cdot T(\frac{n}{8}) + \frac{n}{4}] + 2n = 8T(\frac{n}{8}) + 3n$

$\vdots$

$= 2^i T(\frac{n}{2^i}) + in$; $1 \le i \le \log_2 n$

The maximum value of $i = \log_2 n$ [∵ then only we get $T(1)$].

$= 2^{\log_2 n} \cdot T(\frac{n}{2^{\log_2 n}}) + n \cdot \log_2 n$

$= n \cdot T(1) + n \log_2 n$

$= 2n + n \log_2 n$

$= \theta(n \log_2 n)$

Solution using Master theorem

Here $a = 2$, $b = 2$, $f(n) = n = \theta(n^1) \Rightarrow d = 1$

Also we see that $a = b^d$ [$2 = 2^1$]

∴ As per case-2 of master theorem.

$T(n) = \theta(n^d \cdot \log_2 n)$

$T(n) = \theta(n \log_2 n)$

Exercise 3.2: Solve by substitution method.

$a = 1$, $b = 2$, $f(n) = c$

Soln: $T(n) = T(\frac{n}{2}) + c$

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

$= [T(\frac{n}{8}) + c] + c = T(\frac{n}{8}) + 3c$

$\vdots$

$= T(\frac{n}{2^i}) + ic$; $1 \le i \le \log_2 n$.

$= T(\frac{n}{2^{\log_2 n}}) + \log_2 n \cdot c$

$= T(1) + c \cdot \log_2 n$

Assuming $T(1) = k$ some constant

$T(n) = c \log_2 n + k$.

$T(n) = \theta(\log_2 n)$.

Soln Using master theorem

Here $a = 1$, $b = 2$, $f(n) = c = \theta(1) = \theta(n^0)$.

$\Rightarrow d = 0$.

As $a = b^d$ [$1 = 2^0$], case-2 of master theorem is applied.

$T(n) = \theta(n^d \cdot \log_2 n)$

$T(n) = \theta(\log_2 n)$.

MERGE SORT

The Merge Sort algorithm is a divide-and-conquer algorithm that sorts an array by first breaking it down into smaller arrays, and then building the array back together the correct way so that it is sorted.

Divide: The algorithm starts with breaking up the array into smaller and smaller pieces until one such sub-array only consists of one element.

Conquer: The algorithm merges the small pieces of the array back together by putting the lowest values first, resulting in a sorted array.

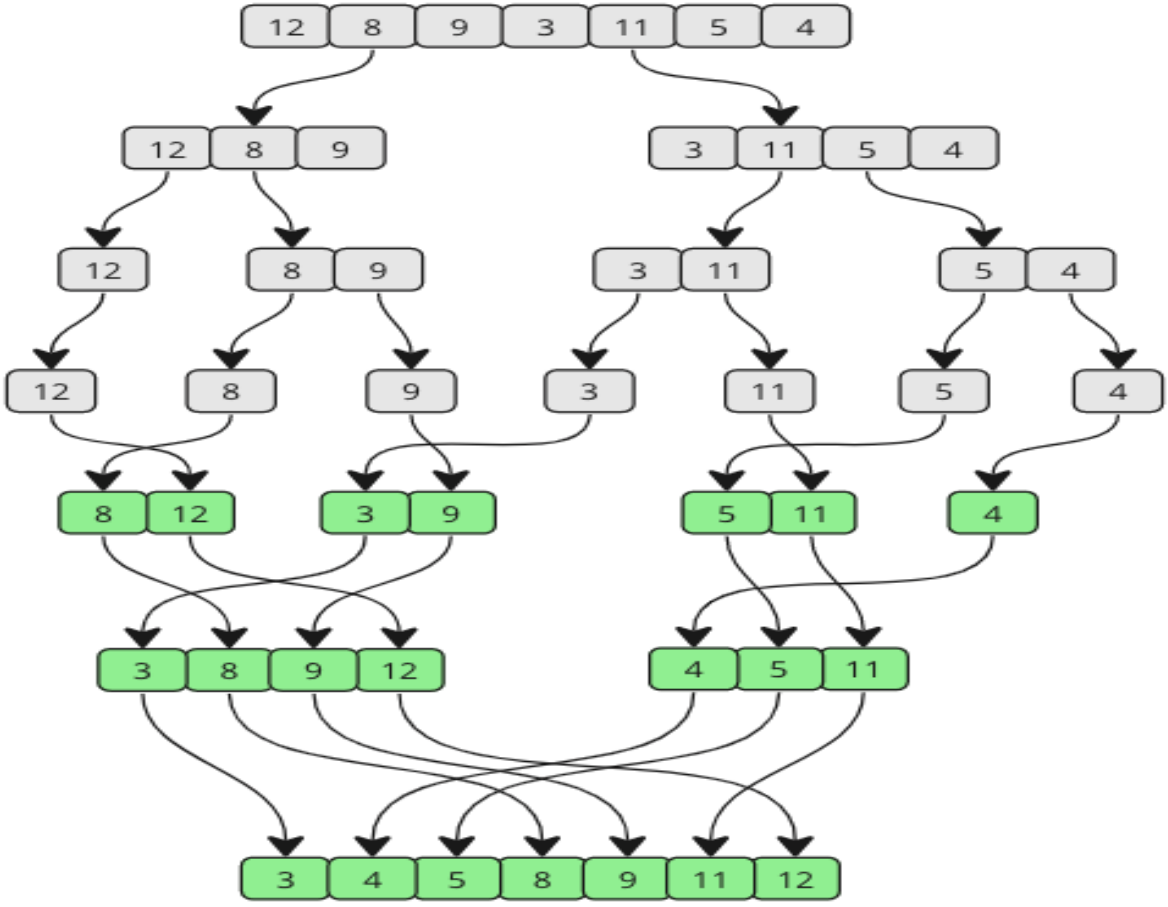
The breaking down and building up of the array to sort the array is done recursively.

In the animation above, each time the bars are pushed down represents a recursive call, splitting the array into smaller pieces. When the bars are lifted up, it means that two sub-arrays have been merged together.

The Merge Sort algorithm can be described as follows:

1. Divide the unsorted array into two sub-arrays, half the size of the original.
2. Continue to divide the sub-arrays as long as the current piece of the array has more than one element.
3. Merge two sub-arrays together by always putting the lowest value first.
4. Keep merging until there are no sub-arrays left.

Take a look at the drawing below to see how Merge Sort works from a different perspective. As you can see, the array is split into smaller and smaller pieces until it is merged back together. And as the merging happens, values from each sub-array are compared so that the lowest value comes first.



Do the sorting manually, just to get an even better understanding of how Merge Sort works before actually implementing it in a programming language.

Step 1: We start with an unsorted array, and we know that it splits in half until the sub-arrays only consist of one element. The Merge Sort function calls itself two times, once for each half of the array. That means that the first sub-array will split into the smallest pieces first.

[12, 8, 9, 3, 11, 5, 4]
[12, 8, 9] [3, 11, 5, 4]
[12] [8, 9] [3, 11, 5, 4]
[12] [8] [9] [3, 11, 5, 4]

Step 2: The splitting of the first sub-array is finished, and now it is time to merge. 8 and 9 are the first two elements to be merged. 8 is the lowest value, so that comes before 9 in the first merged sub-array.

[12] [8, 9] [3, 11, 5, 4]

Step 3: The next sub-arrays to be merged is [12] and [8, 9]. Values in both arrays are compared from the start. 8 is lower than 12, so 8 comes first, and 9 is also lower than 12.

[8, 9, 12] [3, 11, 5, 4]

Step 4: Now the second big sub-array is split recursively.

[8, 9, 12] [3, 11, 5, 4]
[8, 9, 12] [3, 11] [5, 4]
[8, 9, 12] [3] [11] [5, 4]

Step 5: 3 and 11 are merged back together in the same order as they are shown because 3 is lower than 11.

[8, 9, 12] [3, 11] [5, 4]

Step 6: Sub-array with values 5 and 4 is split, then merged so that 4 comes before 5.

[8, 9, 12] [3, 11] [5] [4]
[8, 9, 12] [3, 11] [4, 5]

Step 7: The two sub-arrays on the right are merged. Comparisons are done to create elements in the new merged array:

1. 3 is lower than 4
2. 4 is lower than 11
3. 5 is lower than 11
4. 11 is the last remaining value

[8, 9, 12] [3, 4, 5, 11]

Step 8: The two last remaining sub-arrays are merged. Let's look at how the comparisons are done in more detail to create the new merged and finished sorted array:

3 is lower than 8:

Before [8, 9, 12] [3, 4, 5, 11]

After: [3, 8, 9, 12] [4, 5, 11]

Step 9: 4 is lower than 8:

Before [3, 8, 9, 12] [4, 5, 11]
After: [3, 4, 8, 9, 12] [5, 11]

Step 10: 5 is lower than 8:

Before [3, 4, 8, 9, 12] [5, 11]
After: [3, 4, 5, 8, 9, 12] [11]

Step 11: 8 and 9 are lower than 11:

Before [3, 4, 5, 8, 9, 12] [11]
After: [3, 4, 5, 8, 9, 12] [11]

Step 12: 11 is lower than 12:

Before [3, 4, 5, 8, 9, 12] [11]
After: [3, 4, 5, 8, 9, 11, 12]

The sorting is finished!

We see that the algorithm has two stages: first splitting, then merging.

Although it is possible to implement the Merge Sort algorithm without recursion, we will use recursion because that is the most common approach.

We cannot see it in the steps above, but to split an array in two, the length of the array is divided by two, and then rounded down to get a value we call "mid". This "mid" value is used as an index for where to split the array.

After the array is split, the sorting function calls itself with each half, so that the array can be split again recursively. The splitting stops when a sub-array only consists of one element.

At the end of the Merge Sort function the sub-arrays are merged so that the sub-arrays are always sorted as the array is built back up. To merge two sub-arrays so that the result is sorted, the values of each sub-array are compared, and the lowest value is put into the merged array. After that the next value in each of the two sub-arrays are compared, putting the lowest one into the merged array.

To implement the Merge Sort algorithm we need:

1. An array with values that needs to be sorted.
2. A function that takes an array, splits it in two, and calls itself with each half of that array so that the arrays are split again and again recursively, until a sub-array only consist of one value.
3. Another function that merges the sub-arrays back together in a sorted way.

The time complexity for Merge Sort is $O(n \cdot \log n)$

Advantages:

- Number of comparisons performed is nearly optimal.

- For large n,the number of comparisons made by this algorithm in the average case turns out to be about 0.25n less and hence is also in $\Theta(n \log n)$.
- Merge sort will never degrade to $O(n^2)$
- Another advantage of merge sort over quick sort and heap sort is its **stability**. (A sorting algorithm is said to be stable if two objects with equal keys appear in the same order in sorted output as they appear in the input array to be sorted.)

Limitations:

- The principal shortcoming of merge sort is the linear amount [O(n)] of extra storage the algorithm requires. Though merging can be done in-place, the resulting algorithm is quite complicated and of theoretical interest only.

Variations of merge sort

1. The algorithm can be implemented bottom up by merging pairs of the array's elements, then merging the sorted pairs, and so on.(If n is not a power of 2,only slight book keeping complications arise.) This avoids the time and space overhead of using a stack to handle recursive calls.
2. We can divide a list to be sorted in more than two parts, sort each recursively, and then merge them together. This scheme, which is particularly useful for sorting files residing on secondary memory devices, is called multi way merge sort.

QUICK SORT

The Quick sort algorithm takes an array of values, chooses one of the values as the 'pivot' element, and moves the other values so that lower values are on the left of the pivot element, and higher values are on the right of it.

Quick sort is the widely used sorting algorithm that makes **n log n** comparisons in average case for sorting an array of n elements. It is a faster and highly efficient sorting algorithm. This algorithm follows the divide and conquer approach. Divide and conquer is a technique of breaking down the algorithms into sub problems, then solving the sub problems, and combining the results back together to solve the original problem.

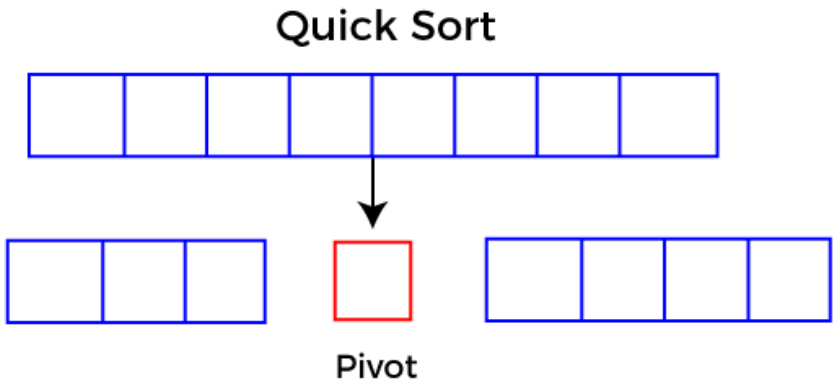
Divide: In Divide, first pick a pivot element. After that, partition or rearrange the array into two sub-arrays such that each element in the left sub-array is less than or equal to the pivot element and each element in the right sub-array is larger than the pivot element.

Conquer: Recursively, sort two sub arrays with Quick sort.

Combine: Combine the already sorted array.

Quick sort picks an element as pivot, and then it partitions the given array around the picked pivot element. In quick sort, a large array is divided into two arrays in which one holds values that are smaller than the specified value (Pivot), and another array holds the values that are greater than the pivot.

After that, left and right sub-arrays are also partitioned using the same approach. It will continue until the single element remains in the sub-array.



Choosing the pivot

Picking a good pivot is necessary for the fast implementation of quicksort. However, it is typical to determine a good pivot. Some of the ways of choosing a pivot are as follows -

- Pivot can be random, i.e. select the random pivot from the given array.
- Pivot can either be the rightmost element of the leftmost element of the given array.
- Select median as the pivot element.

Algorithm

Algorithm:

```
1. QUICKSORT (array A, start, end)
2. {
3.  1 if (start < end)
4.  2 {
5. 3 p = partition(A, start, end)
6. 4 QUICKSORT (A, start, p - 1)
7. 5 QUICKSORT (A, p + 1, end)
8. 6 }
9. }
```

Partition Algorithm:

The partition algorithm rearranges the sub-arrays in a place.

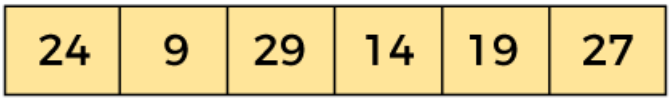
```
1. PARTITION (array A, start, end)
2. {
3.  1 pivot ? A[end]
4.  2 i ? start-1
5.  3 for j ? start to end -1 {
6.  4 do if (A[j] < pivot) {
7.  5 then i ? i + 1
8.  6 swap A[i] with A[j]
9.  7  }}
10. 8 swap A[i+1] with A[end]
11. 9 return i+1
12. }
```

Working of Quick Sort Algorithm

Now, let's see the working of the Quicksort Algorithm.

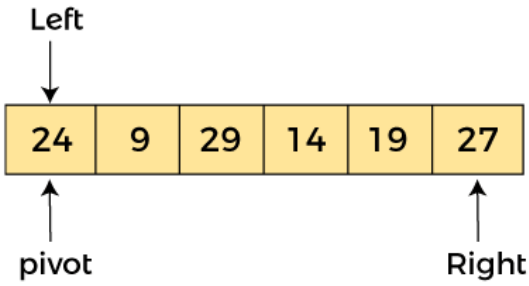
To understand the working of quick sort, let's take an unsorted array. It will make the concept more clear and understandable.

Let the elements of array are -

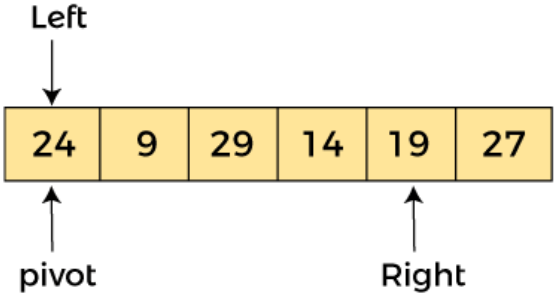


In the given array, we consider the leftmost element as pivot. So, in this case, a[left] = 24, a[right] = 27 and a[pivot] = 24.

Since, pivot is at left, so algorithm starts from right and move towards left.

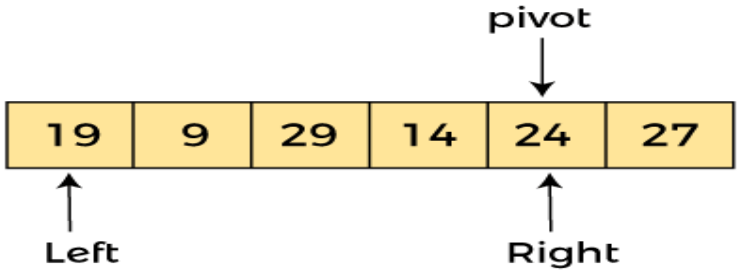


Now, a[pivot] < a[right], so algorithm moves forward one position towards left, i.e. -



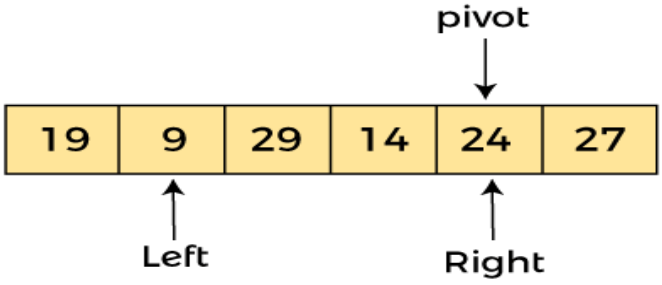
Now, a[left] = 24, a[right] = 19, and a[pivot] = 24.

Because, a[pivot] > a[right], so, algorithm will swap a[pivot] with a[right], and pivot moves to right, as -

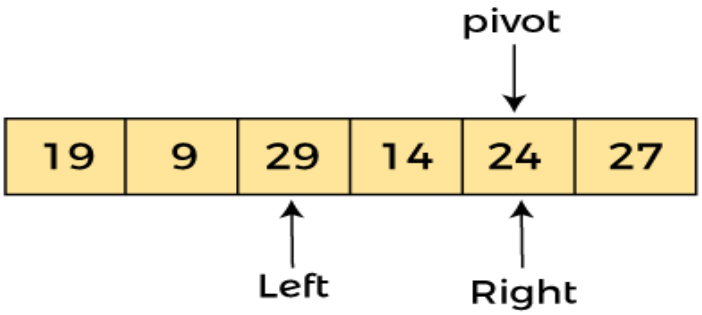


Now, a[left] = 19, a[right] = 24, and a[pivot] = 24. Since, pivot is at right, so algorithm starts from left and moves to right.

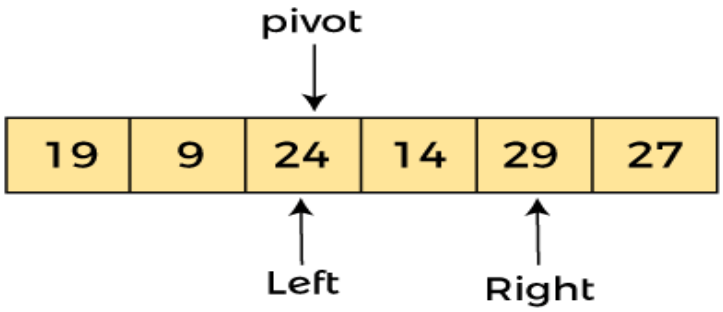
As a[pivot] > a[left], so algorithm moves one position to right as -



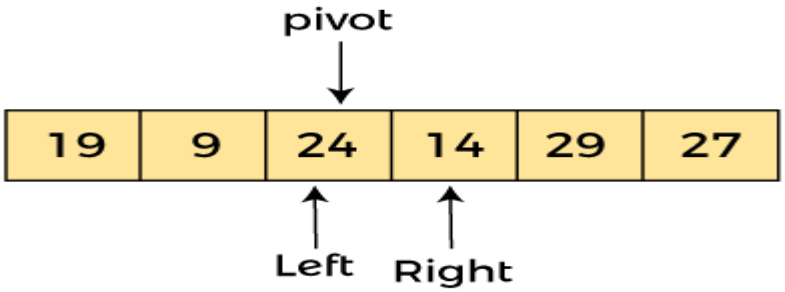
Now, a[left] = 9, a[right] = 24, and a[pivot] = 24. As a[pivot] > a[left], so algorithm moves one position to right as -



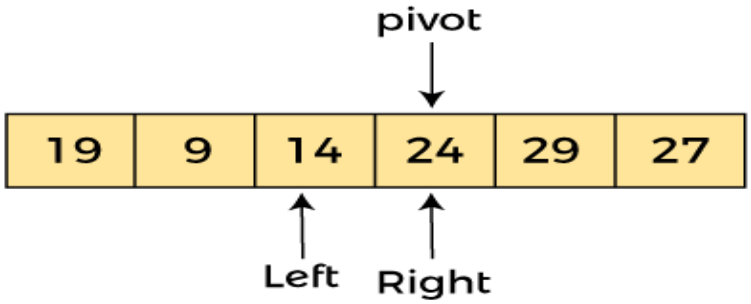
Now, a[left] = 29, a[right] = 24, and a[pivot] = 24. As a[pivot] < a[left], so, swap a[pivot] and a[left], now pivot is at left, i.e. -



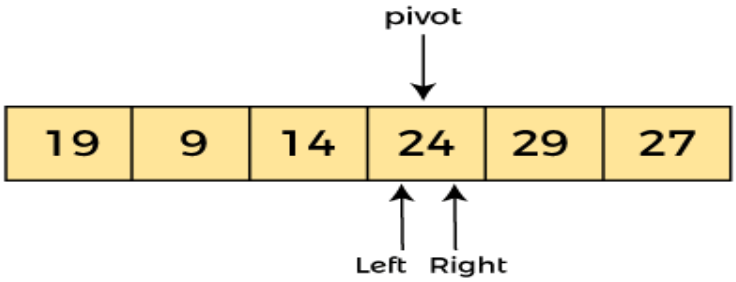
Since, pivot is at left, so algorithm starts from right, and move to left. Now, a[left] = 24, a[right] = 29, and a[pivot] = 24. As a[pivot] < a[right], so algorithm moves one position to left, as -



Now, a[pivot] = 24, a[left] = 24, and a[right] = 14. As a[pivot] > a[right], so, swap a[pivot] and a[right], now pivot is at right, i.e. -



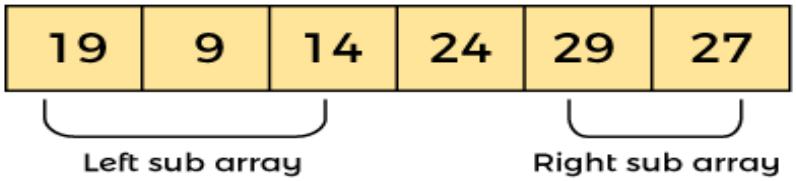
Now, a[pivot] = 24, a[left] = 14, and a[right] = 24. Pivot is at right, so the algorithm starts from left and move to right.



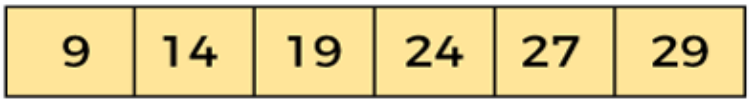
Now, a[pivot] = 24, a[left] = 24, and a[right] = 24. So, pivot, left and right are pointing the same element. It represents the termination of procedure.

Element 24, which is the pivot element is placed at its exact position.

Elements that are right side of element 24 are greater than it, and the elements that are left side of element 24 are smaller than it.



Now, in a similar manner, quick sort algorithm is separately applied to the left and right sub-arrays. After sorting gets done, the array will be -



Variations: Because of quick sort's importance, there have been persistent efforts over the years to refine the basic algorithm. Among several improvements discovered by researchers are:

- Betterpivotselectionmethodssuchasrandomizedquicksortthatusesarandom element or the median-of-three method that uses the median of the leftmost, rightmost, and the middle element of the array
- Switching to insertion sort on very small sub arrays (between 5 and 15 elements for most computer systems) or not sorting small sub arrays at all and finishing the algorithm with insertion sort applied to the entire nearly sorted array
- Modificationsofthepartitioningalgorithmsuchasthethree-waypartitioninto Segments smaller than, equal to, and larger than the pivot

Limitations: 1. It is not stable. 2. It requires a stack to store parameters of sub arrays that are yet to be sorted. 3. While Performance on randomly ordered arrays is known to be sensitive not only to the implementation details of the algorithm but also to both computer architecture and data type.

Strassen's Matrix Multiplication:

Strassen's algorithm, developed by Volker Strassen in 1969, is a fast algorithm for matrix multiplication. It is an efficient divide-and-conquer method that reduces the number of arithmetic operations required to multiply two matrices compared to the conventional matrix multiplication algorithm (the naive approach).

Classical Matrix Multiplication

The traditional matrix multiplication algorithm each element of the resulting matrix is computed by multiplying corresponding elements from a row of the first matrix and a column of the second matrix, then summing these products. This method has a time complexity of **$O(n^3)$** for multiplying two n×n matrices.
In this process,

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

so that we rewrite the equation $C = A \cdot B$ as

$$\begin{pmatrix} C_{11} & C_{12} \\ C_{21} & C_{22} \end{pmatrix} = \begin{pmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{pmatrix} \cdot \begin{pmatrix} B_{11} & B_{12} \\ B_{21} & B_{22} \end{pmatrix}.$$

$$\begin{aligned} C_{11} &= A_{11} \cdot B_{11} + A_{12} \cdot B_{21}, \\ C_{12} &= A_{11} \cdot B_{12} + A_{12} \cdot B_{22}, \\ C_{21} &= A_{21} \cdot B_{11} + A_{22} \cdot B_{21}, \\ C_{22} &= A_{21} \cdot B_{12} + A_{22} \cdot B_{22}. \end{aligned}$$

For multiplying two matrices of size n x n, we make 8 recursive calls above, each on a matrix/subproblem with size n/2 x n/2. Each of these recursive calls multiplies two n/2 x n/2 matrices, which are then added together. For the addition, we add two matrices of size $n^2/4$, so each addition takes $\Theta(n^2/4)$ time. We can write this recurrence in the form of the following equations (taken from Cormet et al.):

$$T(n) = \begin{cases} \Theta(1) & \text{if } n = 1, \\ 8T(n/2) + \Theta(n^2) & \text{if } n > 1. \end{cases}$$

Given two $n \times n$ matrices A and B, the classical algorithm computes the matrix product $C = A \times B$, where each element c_{ij} of the resulting matrix is given by:

$$c_{ij} = \sum_{k=1}^{n} a_{ik} \times b_{kj}$$

This requires n^3 scalar multiplications and $n^2(n-1)$ additions, leading to an overall time complexity of $O(n^3)$.

Strassen's Algorithm

Strassen's algorithm makes use of the same divide and conquer approach as above, but instead uses only 7 recursive calls rather than 8. However, Strassen's algorithm improves $O(n^{2.81})$, a significant improvement for large matrices.. The algorithm achieves this improvement by recursively breaking down the matrix multiplication into smaller subproblems and combining the results.

Strassen's insight was that matrix multiplication can be performed with fewer multiplications by dividing each matrix into smaller submatrices. It is a **divide-and-conquer** algorithm.

Breakdown of Strassen's Algorithm:

1. **Divide the matrices:** Given two matrices A and B, divide each into four smaller $n/2 \times n/2$ submatrices:

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

2. **Multiply submatrices with fewer multiplications:** Strassen found a way to compute the product using **seven** matrix multiplications (instead of eight) and additional additions and subtractions. The seven products, known as **Strassen's products**, are:

- $P = (A_{11} + A_{22})(B_{11} + B_{22})$
- $Q = (A_{21} + A_{22})B_{11}$
- $R = A_{11}(B_{12} - B_{22})$
- $S = A_{22}(B_{21} - B_{11})$
- $T = (A_{11} + A_{12})B_{22}$
- $U = (A_{21} - A_{11})(B_{11} + B_{12})$
- $V = (A_{12} - A_{22})(B_{21} + B_{22})$

3. **Recombine the results:** Use the seven products to compute the submatrices of the resulting matrix C:

$$C_{11} = P + S - T + V$$

$$C_{12} = R + T$$

$$C_{21} = Q + S$$

$$C_{22} = P - Q + R + U$$

Using the above steps, we get the recurrence of the following format:

$$T(n) = \begin{cases} \Theta(1) & \text{if } n = 1\,, \\ 7T(n/2) + \Theta(n^2) & \text{if } n > 1\,. \end{cases}$$

By dividing the matrices and recursively applying the same method to the submatrices, Strassen's algorithm reduces the number of scalar multiplications from 8 to 7 at each level of recursion.

Complexity

The recurrence relation for Strassen's algorithm is:

$$T(n) = 7T(n/2) + O(n^2)$$

Using the master theorem, this solves to $T(n) = O(n^{\log_2 7}) \approx O(n^{2.81})$.

This reduction in complexity makes Strassen's algorithm faster than the classical method for large matrices, though for small matrices, it can be slower due to overhead.

Example:

Consider two 2x2 matrices A and B that we want to multiply:

$$A = \begin{pmatrix} 1 & 3 \\ 7 & 5 \end{pmatrix}, \quad B = \begin{pmatrix} 6 & 8 \\ 4 & 2 \end{pmatrix}$$

We will use Strassen's algorithm to multiply them. For 2×2 matrices, the algorithm is simple to apply because they can be divided directly into four 1×1 submatrices. But first, let's recall the **Strassen's products**.

1. **Divide the matrices** into four submatrices. For these 2×2 matrices, each element is treated as a submatrix itself:

$$A = \begin{pmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{pmatrix} = \begin{pmatrix} 1 & 3 \\ 7 & 5 \end{pmatrix}$$

where:

$$A_{11} = 1, \quad A_{12} = 3, \quad A_{21} = 7, \quad A_{22} = 5$$

and similarly for B:

$$B = \begin{pmatrix} B_{11} & B_{12} \\ B_{21} & B_{22} \end{pmatrix} = \begin{pmatrix} 6 & 8 \\ 4 & 2 \end{pmatrix}$$

where:

$$B_{11} = 6, \quad B_{12} = 8, \quad B_{21} = 4, \quad B_{22} = 2$$

2. **Compute the seven products** using new names P, Q, R, S, T, U, and V:
    1. $P = (A_{11} + A_{22})(B_{11} + B_{22}) = (1+5)(6+2) = 6 \times 8 = 48$
    2. $Q = (A_{21} + A_{22})B_{11} = (7+5) \times 6 = 12 \times 6 = 72$
    3. $R = A_{11}(B_{12} - B_{22}) = 1 \times (8-2) = 1 \times 6 = 6$
    4. $S = A_{22}(B_{21} - B_{11}) = 5 \times (4-6) = 5 \times (-2) = -10$
    5. $T = (A_{11} + A_{12})B_{22} = (1+3) \times 2 = 4 \times 2 = 8$
    6. $U = (A_{21} - A_{11})(B_{11} + B_{12}) = (7-1) \times (6+8) = 6 \times 14 = 84$
    7. $V = (A_{12} - A_{22})(B_{21} + B_{22}) = (3-5) \times (4+2) = (-2) \times 6 = -12$
3. **Recombine the results** to compute the submatrices of the resulting matrix C:

$$C_{11} = P + S - T + V = 48 + (-10) - 8 + (-12) = 18$$

$$C_{12} = R + T = 6 + 8 = 14$$

$$C_{21} = Q + S = 72 + (-10) = 62$$

$$C_{22} = P - Q + R + U = 48 - 72 + 6 + 84 = 66$$

4. **Final result:**

The resulting matrix C is:

$$C = \begin{pmatrix} 18 & 14 \\ 62 & 66 \end{pmatrix}$$

3. Advantages and Disadvantages of Divide And Conquer

Advantages

- **Parallelism:** Divide and conquer algorithms tend to have a lot of inherent parallelism. Once the division phase is complete, the sub-problems are usually independent and can therefore be solved in parallel. This approach typically generates more enough concurrency to keep the machine busy and can be adapted for execution in multi- processor machines.
- **Cache Performance:** divide and conquer algorithms also tend to have good cache performance. Once a sub-problem fits in the cache, the standard recursive solution reuses the cached data until the sub-problem has been completely solved.
- It allows solving **difficult** and often impossible looking problems like the Tower of Hanoi. It reduces the degree of difficulty since it divides the problem into sub problems that are easily solvable, and usually runs faster than other algorithms would.
- Another advantage to this paradigm is that it often **plays a part in finding other efficient algorithms**, and in fact it was the central role in finding the quick sort and merge sort algorithms.

Disadvantages

- One of the most common issues with this sort of algorithm is the fact that the **recursion is slow**, which in some cases outweighs any advantages of this divide and conquer process.
- Another concern with it is the fact that sometimes it can become more **complicated than a basic iterative approach**, especially in cases with a large n. In other words, if someone wanted to add a large amount of numbers together, if they just create a
  Simple loop to add them together, it would turn out to be a much simpler approach than it would be to divide the numbers up into two groups, add these group recursively, and then add the sums of the two groups together.

- Another down fall is that sometimes once the problem is broken down into sub
  Problems, the same sub problem can occur many times. It is solved again. In cases like these, it can often be easier to identify and save the solution to the repeated sub problem, which is commonly referred to as memorization.

GREEDY METHOD

GENERAL METHOD

The greedy method is the straight forward design technique applicable to variety of applications.

The greedy approach suggests constructing a solution through a sequence of steps, each expanding a partially constructed solution obtained so far, until a complete solution to the problem is reached. On each step the choice made must be:

- ***feasible***, i.e., it has to satisfy the problem's constraints
- ***locally optimal***, i.e., it has to be the best local choice among all feasible choices available on that step
- ***irrevocable***, i.e., once made, it cannot be changed on subsequent steps of the algorithm

As a rule, greedy algorithms are both intuitively appealing and simple. Given an optimization problem, it is usually easy to figure out how to proceed in a greedy manner, possibly after considering a few small instances of the problem. What is usually more difficult is to prove that a greedy algorithm yields an optimal solution (when it does).

```
Algorithm Greedy(a, n)
// a[1 : n] contains the n inputs.
{
    solution := ∅; // Initialize the solution.
    for i := 1 to n do
    {
        x := Select(a);
        if Feasible(solution, x) then
            solution := Union(solution, x);
    }
    return solution;
}
```

Knapsack Problem (Fractional knapsack problem)

The fractional knapsack problem is also one of the techniques which are used to solve the knapsack problem. In fractional knapsack, the items are broken in order to maximize the profit. The problem in which we break the item is known as a Fractional knapsack problem.

This problem can be solved with the help of using two techniques:

- Brute-force approach: The brute-force approach tries all the possible solutions with all the different fractions but it is a time-consuming approach.
- Greedy approach: In Greedy approach, we calculate the ratio of profit/weight, and accordingly, we will select the item. The item with the highest ratio would be selected first.

There are basically three approaches to solve the problem:

- The first approach is to select the item based on the maximum profit.
- The second approach is to select the item based on the minimum weight.
- The third approach is to calculate the ratio of profit/weight.

Consider the below example:

Objects:	1	2	3	4	5	6	7
Profit (P):	5	10	15	7	8	9	4
Weight(w):	1	3	5	4	1	3	2

W (Weight of the knapsack): 15

n (no of items): 7

First approach:
First approach:

Object	Profit	Weight	Remaining weight
3	15	5	15 - 5 = 10
2	10	3	10 - 3 = 7
6	9	3	7 - 3 = 4
5	8	1	4 - 1 = 3
7	7 * ¾ = 5.25	3	3 - 3 = 0

The total profit would be equal to (15 + 10 + 9 + 8 + 5.25) = 47.25

Second approach:
The second approach is to select the item based on the minimum weight.

Object	Profit	Weight	Remaining weight
1	5	1	15 - 1 = 14
5	7	1	14 - 1 = 13
7	4	2	13 - 2 = 11
2	10	3	11 - 3 = 8
6	9	3	8 - 3 = 5
4	7	4	5 - 4 = 1
3	15 * 1/5 = 3	1	1 - 1 = 0

In this case, the total profit would be equal to (5 + 7 + 4 + 10 + 9 + 7 + 3) = 46

Third approach:
In the third approach, we will calculate the ratio of profit/weight.

Objects: 1 2 3 4 5 6 7

Profit (P): 5 10 15 7 8 9 4

Weight(w): 1 3 5 4 1 3 2

In this case, we first calculate the profit/weight ratio.

Object 1: 5/1 = 5

Object 2: 10/3 = 3. 33

Object 3: 15/5 = 3

Object 4: 7/4 = 1.7

Object 5: 8/1 = 8

Object 6: 9/3 = 3

Object 7: 4/2 = 2

P:w: 5 3.3 3 1.7 8 3 2

In this approach, we will select the objects based on the maximum profit/weight ratio. Since the P/W of object 5 is maximum so we select object 5.

Object	Profit	Weight	Remaining weight
5	8	1	15 - 8 = 7

After object 5, object 1 has the maximum profit/weight ratio, i.e., 5. So, we select object 1 shown in the below table:

Object	Profit	Weight	Remaining weight
5	8	1	15 - 1 = 14
1	5	1	14 - 1 = 13

After object 1, object 2 has the maximum profit/weight ratio, i.e., 3.3. So, we select object 2 having profit/weight ratio as 3.3.

Object	Profit	Weight	Remaining weight
5	8	1	15 - 1 = 14
1	5	1	14 - 1 = 13
2	10	3	13 - 3 = 10

After object 2, object 3 has the maximum profit/weight ratio, i.e., 3. So, we select object 3 having profit/weight ratio as 3.

Object	Profit	Weight	Remaining weight
5	8	1	15 - 1 = 14
1	5	1	14 - 1 = 13
2	10	3	13 - 3 = 10
3	15	5	10 - 5 = 5

After object 3, object 6 has the maximum profit/weight ratio, i.e., 3. So we select object 6 having profit/weight ratio as 3.

Object	Profit	Weight	Remaining weight
5	8	1	15 - 1 = 14
1	5	1	14 - 1 = 13
2	10	3	13 - 3 = 10
3	15	5	10 - 5 = 5
6	9	3	5 - 3 = 2

After object 6, object 7 has the maximum profit/weight ratio, i.e., 2. So we select object 7 having profit/weight ratio as 2.

Object	Profit	Weight	Remaining weight
5	8	1	15 - 1 = 14
1	5	1	14 - 1 = 13
2	10	3	13 - 3 = 10
3	15	5	10 - 5 = 5
6	9	3	5 - 3 = 2
7	4	2	2 - 2 = 0

As we can observe in the above table that the remaining weight is zero which means that the knapsack is full. We cannot add more objects in the knapsack. Therefore, the total profit would be equal to (8 + 5 + 10 + 15 + 9 + 4), i.e., 51.

In the first approach, the maximum profit is 47.25. The maximum profit in the second approach is 46. The maximum profit in the third approach is 51. Therefore, we can say that the third approach, i.e., maximum profit/weight ratio is the best approach among all the approaches.

Analysis:
Disregarding the time to initially sort the object, each of the above strategies use O(n) time,

JOB SEQUENCING WITH DEADLINES

The prime objective of the Job Sequencing with Deadlines algorithm is to complete the given order of jobs within respective deadlines, resulting in the highest possible profit. To achieve this, we are given a number of jobs, each associated with a specific deadline, and completing a job before its deadline earns us a profit. The challenge is to arrange these jobs in a way that maximizes our total profit.

GREEDY ALGORITHM-

Greedy Algorithm is adopted to determine how the next job is selected for an optimal solution.
The greedy algorithm described below always gives an optimal solution to the job sequencing problem-

Step-01:

Sort all the given jobs in decreasing order of their profit.

Step-02:

Check the value of maximum deadline.
Draw a Gantt chart where maximum time on Gantt chart is the value of maximum deadline.

Step-03:
Pick up the jobs one by one.
Put the job on Gantt chart as far as possible from 0 ensuring that the job gets completed before its deadline.
Problem-

Given the jobs, their deadlines and associated profits as shown-

Jobs	J1	J2	J3	J4	J5	J6
Deadlines	5	3	3	2	4	2
Profits	200	180	190	300	120	100

Solution-

Step-01:

Sort all the given jobs in decreasing order of their profit-

Jobs	J4	J1	J3	J2	J5	J6
Deadlines	2	5	3	3	4	2
Profits	300	200	190	180	120	100

Step-02:

Value of maximum deadline = 5.
So, draw a Gantt chart with maximum time on Gantt chart = 5 units as shown-

Now,
We take each job one by one in the order they appear in Step-01.
We place the job on Gantt chart as far as possible from 0.

Step-03:

We take job J4.
Since its deadline is 2, so we place it in the first empty cell before deadline 2 as-

Step-04:

We take job J1.
Since its deadline is 5, so we place it in the first empty cell before deadline 5 as-

Step-05:

We take job J3.
Since its deadline is 3, so we place it in the first empty cell before deadline 3 as-

Step-06:

We take job J2.
Since its deadline is 3, so we place it in the first empty cell before deadline 3.
Since the second and third cells are already filled, so we place job J2 in the first cell as-

Step-07:

Now, we take job J5.
Since its deadline is 4, so we place it in the first empty cell before deadline 4 as-

Now,
The only job left is job J6 whose deadline is 2.
All the slots before deadline 2 are already occupied.
Thus, job J6 can not be completed.

Now, the given questions may be answered as-

Part-01:

The optimal schedule is-
J2 , J4 , J3 , J5 , J1
This is the required order in which the jobs must be completed in order to obtain the maximum profit.

Part-02:

All the jobs are not completed in optimal schedule.
This is because job J6 could not be completed within its deadline.

Part-03:

Maximum earned profit
= Sum of profit of all the jobs in optimal schedule
= Profit of job J2 + Profit of job J4 + Profit of job J3 + Profit of job J5 + Profit of job J1
= 180 + 300 + 190 + 120 + 200
= 990 units

MINIMUM COST SPANNING TREES

Definition: A **spanning tree** of a connected graph is its connected acyclic sub graph (i.e., a tree) that contains all the vertices of the graph. A **minimum spanning tree** of a weighted connected graph is its spanning tree of the smallest weight, where the weight of a tree is defined as the sum of the **weights** on all its edges. The **minimum spanning tree problem** is the problem of finding a minimum spanning tree for a given weighted connected graph.

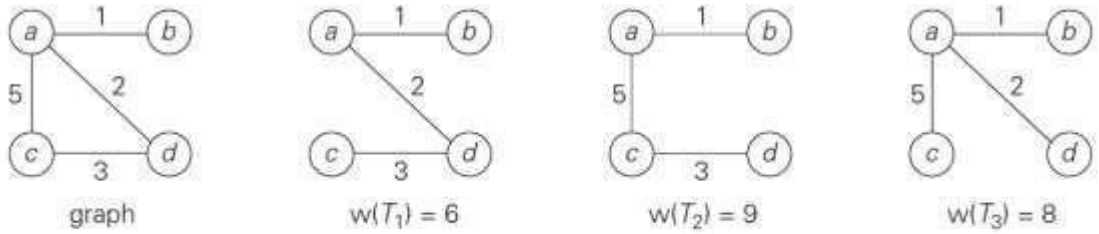
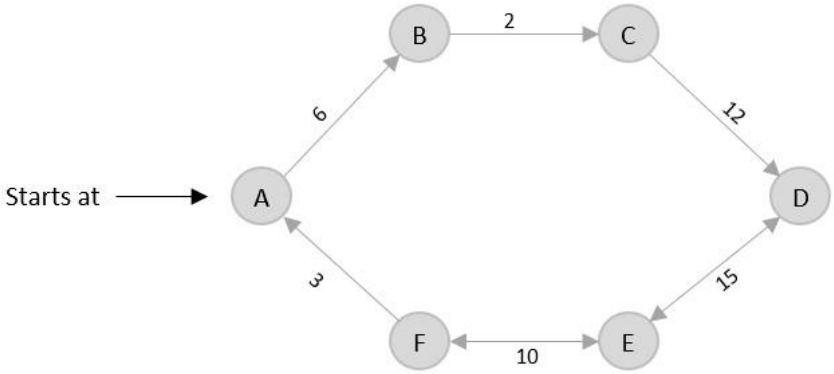


FIGURE 9.2 Graph and its spanning trees, with T_1 being the minimum spanning tree.

Prim's Algorithm

Prim's minimal spanning tree algorithm is one of the efficient methods to find the minimum spanning tree of a graph. A minimum spanning tree is a sub graph that connects all the vertices present in the main graph with the least possible edges and minimum cost (sum of the weights assigned to each edge).

The algorithm, similar to any shortest path algorithm, begins from a vertex that is set as a root and walks through all the vertices in the graph by determining the least cost adjacent edges.



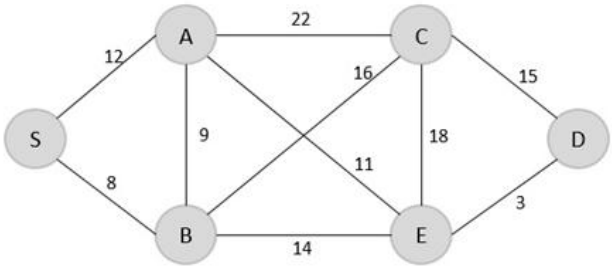
To execute the prim's algorithm, the inputs taken by the algorithm are the graph G {V, E}, where V is the set of vertices and E is the set of edges, and the source vertex S. A minimum spanning tree of graph G is obtained as an output.

Algorithm

- Declare an array *visited*[] to store the visited vertices and firstly, add the arbitrary root, say S, to the visited array.
- Check whether the adjacent vertices of the last visited vertex are present in the *visited*[] array or not.
- If the vertices are not in the *visited*[] array, compare the cost of edges and add the least cost edge to the output spanning tree.
- The adjacent unvisited vertex with the least cost edge is added into the *visited*[] array and the least cost edge is added to the minimum spanning tree output.
- Steps 2 and 4 are repeated for all the unvisited vertices in the graph to obtain the full minimum spanning tree output for the given graph.
- Calculate the cost of the minimum spanning tree obtained.

Examples

- Find the minimum spanning tree using prim's method (greedy approach) for the graph given below with S as the arbitrary root.



Solution

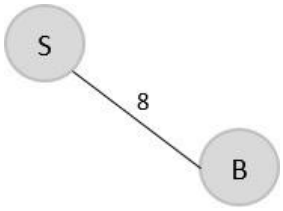
Step 1

Create a visited array to store all the visited vertices into it.

V = { }

The arbitrary root is mentioned to be S, so among all the edges that are connected to S we need to find the least cost edge.

S → B = 8
V = {S, B}



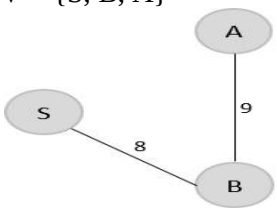
Step 2

Since B is the last visited, check for the least cost edge that is connected to the vertex B.

B → A = 9
B → C = 16
B → E = 14

Hence, B → A is the edge added to the spanning tree.

V = {S, B, A}



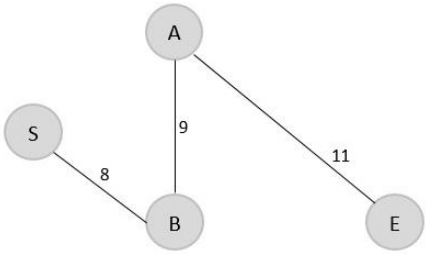
Step 3

Since A is the last visited, check for the least cost edge that is connected to the vertex A.

A → C = 22
A → B = 9
A → E = 11

But A → B is already in the spanning tree, check for the next least cost edge. Hence, A → E is added to the spanning tree.

V = {S, B, A, E}



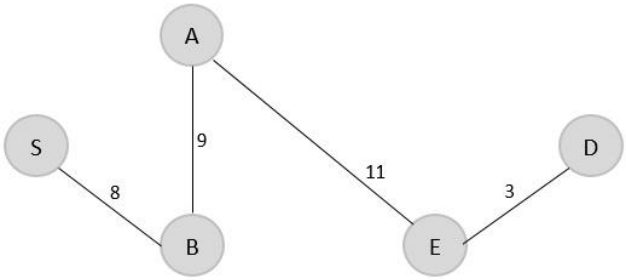
Step 4

Since E is the last visited, check for the least cost edge that is connected to the vertex E.

E → C = 18
E → D = 3

Therefore, E → D is added to the spanning tree.

V = {S, B, A, E, D}



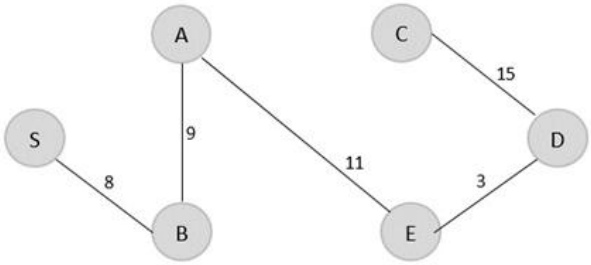
Step 5

Since D is the last visited, check for the least cost edge that is connected to the vertex D.

D → C = 15
E → D = 3

Therefore, D → C is added to the spanning tree.

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



The minimum spanning tree is obtained with the minimum cost = 46

Analysis of Efficiency

The efficiency of Prim's algorithm depends on the data structures chosen for the **graph** itself and for the **priority queue** of the set $V - V_T$ whose vertex priorities are the distances to the nearest tree vertices.

1. If a graph is represented by its **weight matrix** and the priority queue is implemented as an **unordered array**, the algorithm's running time will be in $\Theta(|V|^2)$. Indeed, on each of the $|V| - 1$iterations, the array implementing the priority queue is traversed to find and delete the minimum and then to update, if necessary, the priorities of the remaining vertices.

We can implement the priority queue as a **min-heap**. (A min-heap is a complete binary tree in which every element is less than or equal to its children.) Deletion of the smallest element from and insertion of a new element into a min-heap of size n are **O(log n)** operations.

2. If a graph is represented by its **adjacency lists** and the priority queue is implemented as a **min-heap**, the running time of the algorithm is in **O(|E| log |V |).**

This is because the algorithm performs $|V|-1$deletions of the smallest element and makes $|E|$ verifications and, possibly, changes of an element's priority in a min-heap of size not exceeding $|V|$. Each of these operations, as noted earlier, is a $O(\log |V|)$ operation. Hence, the running time of this implementation of Prim's algorithm is in

$(|V|-1+ |E|)\ O(\log |V|) = O(|E| \log|V|)$ because, in a connected graph, $|V|-1\leq|E|$.

Kruskal's Algorithm

Kruskal's minimal spanning tree algorithm is one of the efficient methods to find the minimum spanning tree of a graph. A minimum spanning tree is a subgraph that connects all the vertices present in the main graph with the least possible edges and minimum cost (sum of the weights assigned to each edge).

The algorithm first starts from the forest – which is defined as a subgraph containing only vertices of the main graph – of the graph, adding the least cost edges later until the minimum spanning tree is created without forming cycles in the graph.

Kruskal's algorithm has easier implementation than prim's algorithm, but has higher complexity.

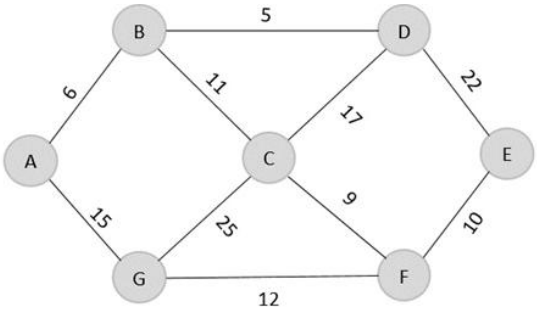
The inputs taken by the kruskal's algorithm are the graph G {V, E}, where V is the set of vertices and E is the set of edges, and the source vertex S and the minimum spanning tree of graph G is obtained as an output.

Algorithm

- Sort all the edges in the graph in an ascending order and store it in an array *edge*[].
- Construct the forest of the graph on a plane with all the vertices in it.
- Select the least cost edge from the edge[] array and add it into the forest of the graph. Mark the vertices visited by adding them into the visited[] array.
- Repeat the steps 2 and 3 until all the vertices are visited without having any cycles forming in the graph
- When all the vertices are visited, the minimum spanning tree is formed.
- Calculate the minimum cost of the output spanning tree formed.

Examples

Construct a minimum spanning tree using kruskal's algorithm for the graph given below −

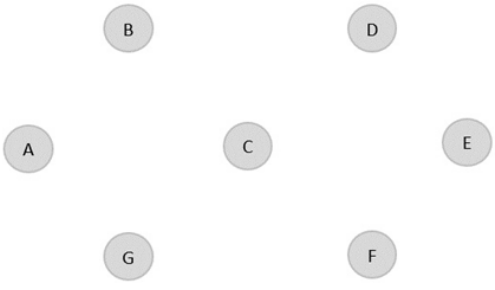


Solution

As the first step, sort all the edges in the given graph in an ascending order and store the values in an array.

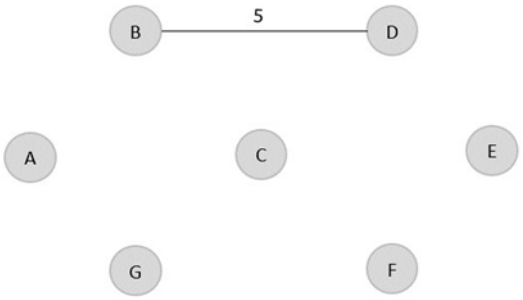
Edge	B→D	A→B	C→F	F→E	B→C	G→F	A→G	C→D	D→E	C→G
Cost	5	6	9	10	11	12	15	17	22	25

Then, construct a forest of the given graph on a single plane.



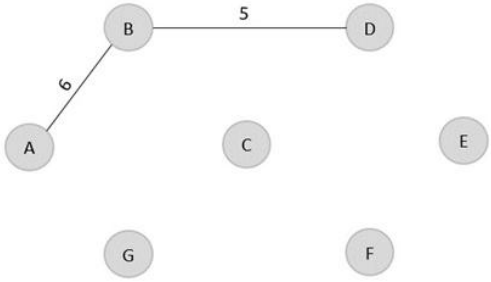
From the list of sorted edge costs, select the least cost edge and add it onto the forest in output graph.

B → D = 5
Minimum cost = 5
Visited array, v = {B, D}



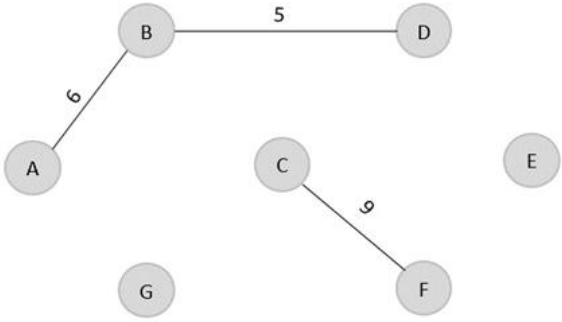
Similarly, the next least cost edge is B → A = 6; so we add it onto the output graph.

Minimum cost = 5 + 6 = 11
Visited array, v = {B, D, A}



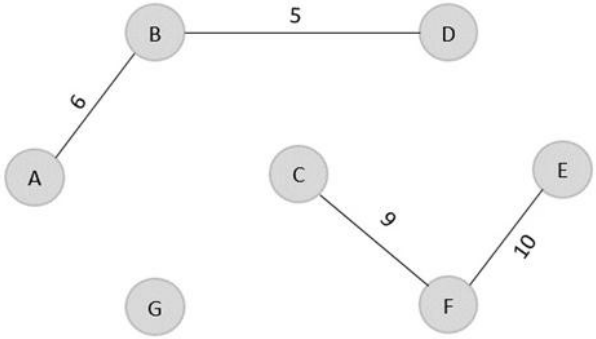
The next least cost edge is C → F = 9; add it onto the output graph.

Minimum Cost = 5 + 6 + 9 = 20
Visited array, v = {B, D, A, C, F}



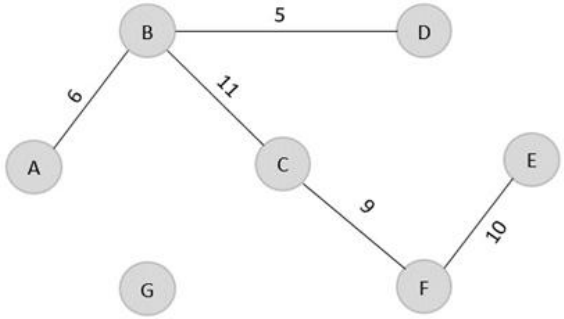
The next edge to be added onto the output graph is F → E = 10.

Minimum Cost = 5 + 6 + 9 + 10 = 30
Visited array, v = {B, D, A, C, F, E}



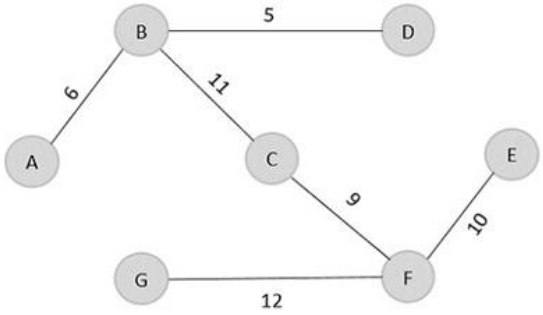
The next edge from the least cost array is B → C = 11, hence we add it in the output graph.

Minimum cost = 5 + 6 + 9 + 10 + 11 = 41
Visited array, v = {B, D, A, C, F, E}



The last edge from the least cost array to be added in the output graph is F → G = 12.

Minimum cost = 5 + 6 + 9 + 10 + 11 + 12 = 53
Visited array, v = {B, D, A, C, F, E, G}



The obtained result is the minimum spanning tree of the given graph with cost = 53.

Analysis of Efficiency

The crucial check whether two vertices belong to the same tree can be found out using **union- find algorithms.**

Efficiency of Kruskal's algorithm is based on the time needed for sorting the edge weights of a given graph. Hence, with an efficient sorting algorithm, the time efficiency of Kruskal's algorithm will be in O (|E| log |E|).

SINGLE SOURCE SHORTEST PATHS

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**.

In this chapter, we will learn about the greedy approach of the dijkstra's algorithm.

Dijkstra's Algorithm

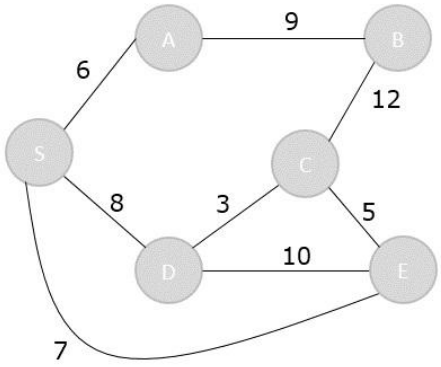
The dijkstra's algorithm is designed to find the shortest path between two vertices of a graph. These two vertices could either be adjacent or the farthest points in the graph. The algorithm starts from the source. The inputs taken by the algorithm are the graph G {V, E}, where V is the set of vertices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

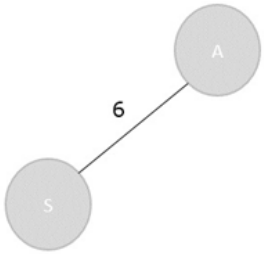
Step 2

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.

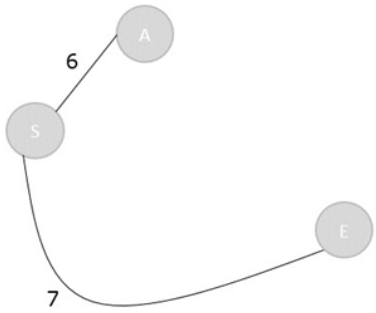
Vertex S has two more adjacent vertices to be visited yet: D and E. Vertex A has one adjacent vertex B.

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

S → D = 8 and S → E = 7.
S → B = S → A + A → B = 6 + 9 = 15

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

Visited = {S, A, E}



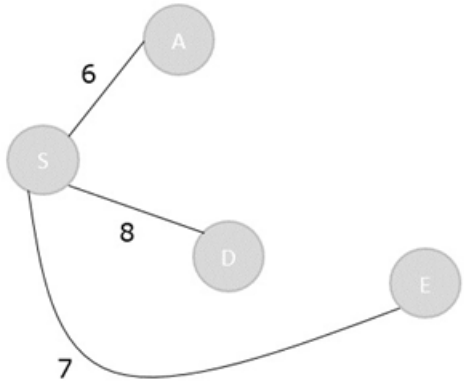
Step 4

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

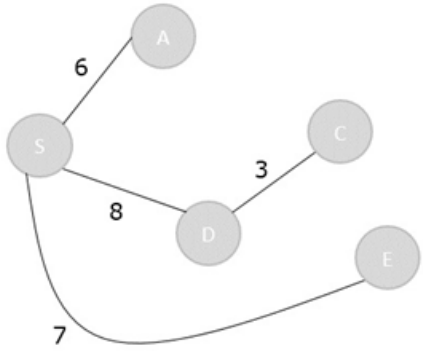
dist[C] = minimum (12, 11) = 11

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

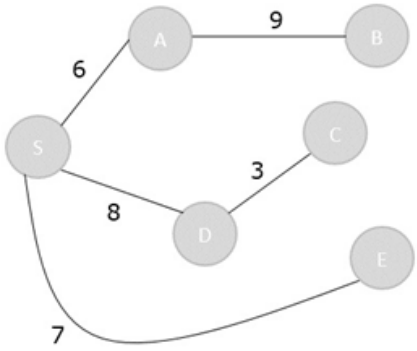
Visited = { S, A, E, D, C}



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.

Analysis:

The time efficiency of Dijkstra's algorithm depends on the data structures used for implementing the priority queue and for representing an input graph itself.

Efficiency is $\Theta(|V|^2)$ for graphs represented by their ***weight matrix*** and the priority queue implemented as an ***unordered array***.

For graphs represented by their ***adjacency lists*** and the priority queue implemented as a ***min- heap***, it is in **O (|E| log |V|)**

Applications

- Transportation planning and packet routing in communication networks, including the Internet
- Finding shortest paths in social networks, speech recognition, document formatting, robotics, compilers, and airline crew scheduling.