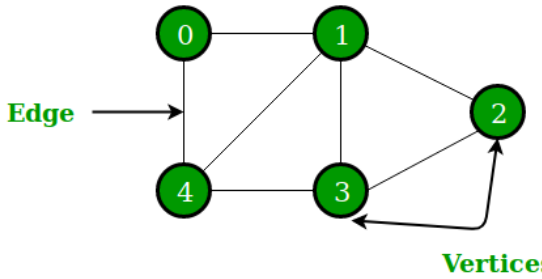
UNIT - II

Graphs – Terminology, Representations, Basic Search and Traversals, Connected Components and Biconnected Components, applications Divide and Conquer: The General Method, Quick Sort, Merge Sort, Strassen's matrix multiplication

Graphs – Terminology:

- **Graph:** A graph is a nonlinear data structure. A graph contains a set of points known as nodes or vertices and set of links known as edges or arcs which connects the vertices. A graph G is represented as G=(V, E) where V is the set of vertices and E is set of edges.
  Ex:

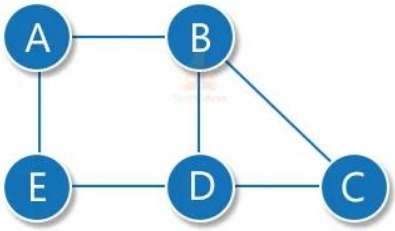


- **Vertex:** An individual data element of a graph is called a vertex. It is also known as node. In the above graph 0,1,2,3,4 are vertices.
- **Edge:** An edge is connecting link between to vertices. It is also known as Arc. An edge is represented as (Starting Vertex, Ending Vertex). In the above example (0,4), (4,3), (0,1), (1,3), (1,2), (2,3), (1,4) are the edges. Edges are three types;
  - **Undirected Edge:** It is a bidirectional edge. If there is an undirected edge between Vertices 0 and 1 then edge (0,1) is equal to the edge (1,0).
  - **Directed Edge:** It is a undirectional edge. If there is an directed edge between Vertices 0 and 1 then edge (0,1) is not equal to the edge (1,0).
  - **Weighted Edge:** It is an edge with cost on it.
- **Degree:** The number of edges connected to a vertex. In directed graphs;
  - **In-degree:** It is number of incoming edges to a vertex.
  - **Out-degree:** It is number of outgoing edges to a vertex.
- **Path:** A sequence of edges that connects a sequence of vertices is known as path.
- **Cycle:** A path that starts and end at the same vertex without repeating any edges or vertices (excepting starting/ending vertex).

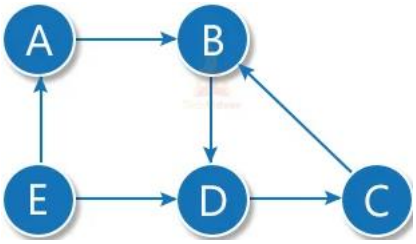
- **Sub Graph:** A graph formed from a subset of the vertices and edges of a larger graph is said to be sub graph.

Types of Graph:

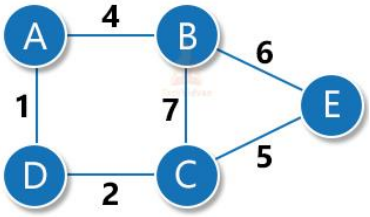
1. **Undirected Graph:** A graph with only undirected edges is said to undirected graph.



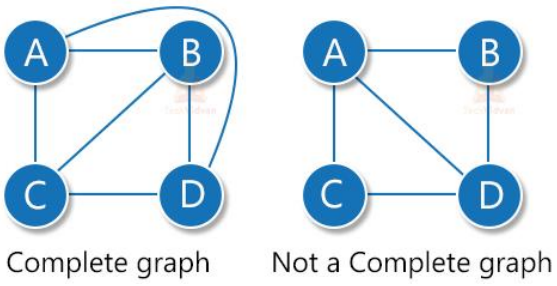
2. **Directed Graph:** A graph with directed edges is said to be directed graph.



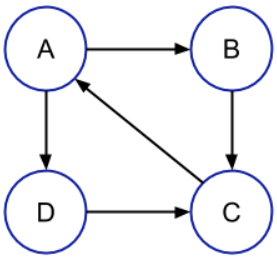
3. **Weighted Graph:** A weighted graph is the one in which we associate some weight with every edge.



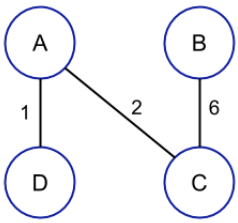
4. **Complete Graph:** A complete graph is a simple graph in which every node is adjacent to every other node. A complete graph has n(n-1)/2 edges.



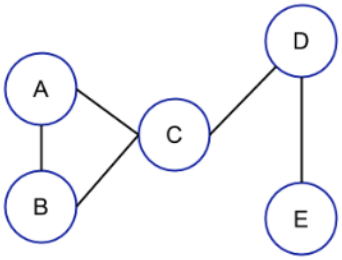
5. **Cyclic Graph:** A graph having a cycle is known as cyclic graph. A graph is said to have a cycle if you start from a node and after traversing some nodes, you come to the same node, then it is said to be graph is having a cycle**.** For a cyclic graph at lease one cycle is necessary.



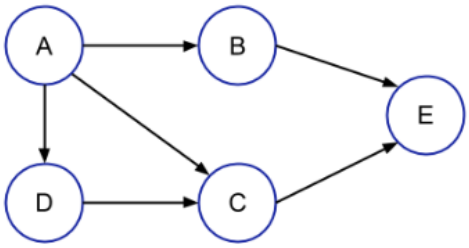
6. **Acyclic Graph:** A graph without cycle is called as acyclic graph.



7. **Connected Graph:** In this from each node, there is a path to all other nodes is existed.



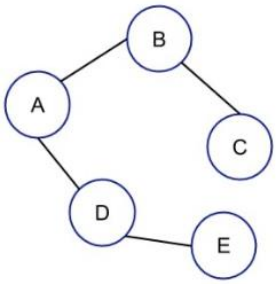
8. **Disconnected Graph:** In this we can't access all nodes from a particular node.



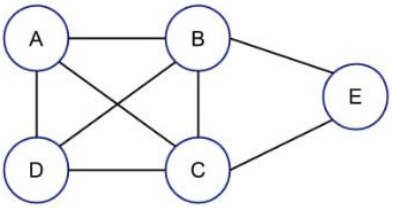
9. **Sparse Graph and Dense Graph:** If the number of edges of a graph is close to the total number of possible edges of that graph, then the graph is said to be Dense Graph otherwise, it is said to be a Sparse Graph.

   For example, if a graph is an undirected graph and there are 5 nodes, then the total number of possible edges will be n(n-1)/2 i.e. 5(5-1)/2 = 10. Now, if the graph contains

4 edges, then the graph is said to be Sparse Graph because 4 is very less than 10 and if the graph contains 8 nodes, then the graph is said to be Dense Graph because 8 is close to 10 i.e. total number of possible edges.



Sparse Graph



Dense Graph

Graph Representations:

A graph can be represented as;

(a) Adjacency Matrix

(b) Adjacency List

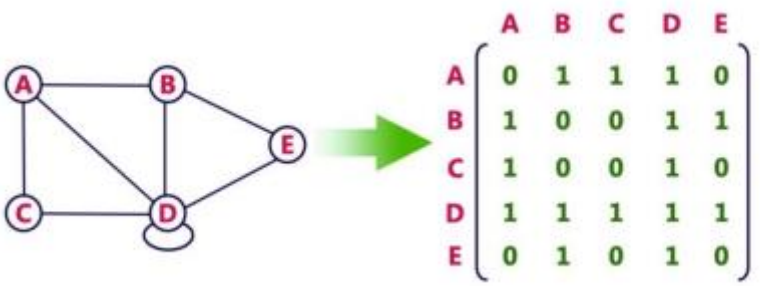
Adjacency Matrix:

In this representation, graph can be represented using a matrix of size total number of vertices by total number of vertices; means if a graph with 4 vertices can be represented using a matrix of 4X4 size. In this matrix, rows and columns both represent vertices.

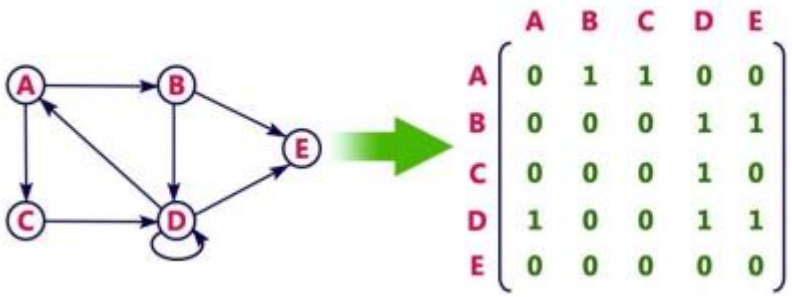
This matrix is filled with either 1 or 0. Here, 1 represents there is an edge from row vertex to column vertex and 0 represents there is no edge from row vertex to column vertex.

Adjacency Matrix : let G = (V, E) with n vertices, n ≥ 1. The adjacency matrix of G is a 2-dimensional n × n matrix, A, A(i, j) = 1 iff (vi , vj) ∈E(G) (⟨vi , vj⟩ for a diagraph), A(i, j) = 0 otherwise.

Ex: for undirected graph



Ex: for directed graph



Advantages:

- Simple to Implement

Disadvantages:

In an Adjacency matrix, we create a matrix of size *n*n* , where n is the number of nodes present in the graph. Now, suppose the graph is a Sparse graph i.e. total number of edges present in the graph is very less as compared to the total possible edges.

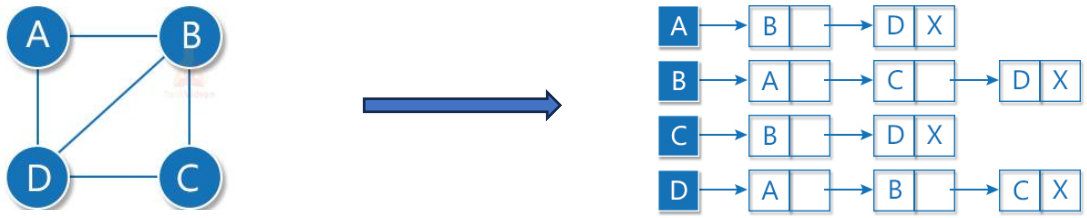
For example , if the total number of nodes is 5 and the number of edges is 4. Then we will make a matrix of size 5*5 = 25 and we are storing the value of only 4 edges. Rest of the 21 spaces are vacant. So, here there is a waste of memory.

In order to solve this problem of wastage of memory in case of a Sparse matrix, we use the Adjacency List.

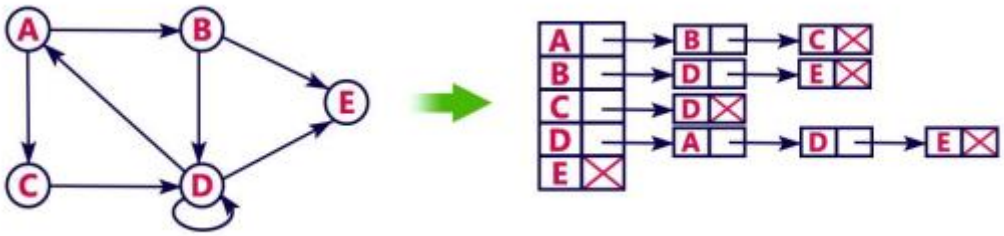
Adjacency List:

In this representation, every vertex of graph contains list of its adjacent vertices. The n rows of the adjacency matrix are represented as n chains. The nodes in chain I represent the vertices that are adjacent to vertex i. It can be represented in two forms. In one form, array is used to store n vertices and chain is used to store its adjacencies.

Ex 1: Undirected Graph using Linked List



Ex 2: Directed Graph using Linked List



Operations performed on a Graph:

- Graphs Traversal
- Display Vertex
- Add Vertex
- Add Edge
- Search Vertex

Basic Search and Traversals:

Graph traversal refers to the process of visiting each vertex in a graph. Traversals are classified by the order in which the vertices are visited.

There are two standard methods of graph traversal are there. They are;

1. Breadth First Search (BFS): It uses queue data structure to store nodes for further processing.
2. Depth First Search (DFS): It uses stack data structure to store nodes for further processing.

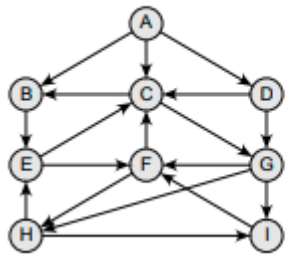
Breadth First Search (BFS):

BFS algorithm is a fundamental graph traversal technique used to explore nodes and edges of a graph in layers or levels. It starts from a given node (source) and explores all its neighbors (nodes at the same depth) before moving on to nodes at the next level.

Algorithm:

1. Start with any vertex, make it as visited and add it to the queue.
2. Deque the node from the queue, enque all its neighbours and mark as visited.
3. Continue the same process until the queue is empty.

Ex:



Adjacency lists
A: B, C, D
B: E
C: B, G
D: C, G
E: C, F
F: C, H
G: F, H, I
H: E, I
I: F

(a) Add A to QUEUE

A								

(b) Dequeue A and enqueue the neighbours of A.

FRONT = 1	QUEUE = A	B	C	D
REAR = 3	ORIG = \0	A	A	A

(c) Dequeue a node by setting FRONT = FRONT + 1 and enqueue the neighbours of B. Also, add B as the ORIG of its neighbours.

FRONT = 2	QUEUE = A	B	C	D	E
REAR = 4	ORIG = \0	A	A	A	B

(d) Dequeue a node by setting FRONT = FRONT + 1 and enqueue the neighbours of C. Also, add C as the ORIG of its neighbours. Note that C has two neighbours B and G. Since B has already been added to the queue and it is not in the Ready state, we will not add B and only add G.

FRONT = 3	QUEUE = A	B	C	D	E	G
REAR = 5	ORIG = \0	A	A	A	B	C

(e) Dequeue a node by setting FRONT = FRONT + 1 and enqueue the neighbours of D. Also, add D as the ORIG of its neighbours. Note that D has two neighbours C and G. Since both of them have already been added to the queue and they are not in the Ready state, we will not add them again.

FRONT = 4	QUEUE = A	B	C	D	E	G
REAR = 5	ORIG = \0	A	A	A	B	C

(f) Dequeue a node by setting FRONT = FRONT + 1 and enqueue the neighbours of E. Also, add E as the ORIG of its neighbours. Note that E has two neighbours C and F. Since C has already been added to the queue and it is not in the Ready state, we will not add C and add only F.

FRONT = 5	QUEUE = A	B	C	D	E	G	F
REAR = 6	ORIG = \0	A	A	A	B	C	E

(g) Dequeue a node by setting FRONT = FRONT + 1 and enqueue the neighbours of G. Also, add G as the ORIG of its neighbours. Note that G has three neighbours F, H, and I.

FRONT = 6	QUEUE = A	B	C	D	E	G	F	H	I
REAR = 9	ORIG = \0	A	A	A	B	C	E	G	G

Since F has already been added to the queue, we will only add H and I. As I is our final destination, we stop the execution of this algorithm as soon as it is encountered and added to the QUEUE. Now, backtrack from I using ORIG to find the minimum path P. Thus, we have P as A → C → G → I.

Applications of Breadth-First Search Algorithm

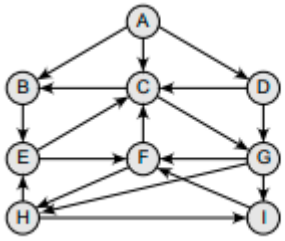
- Finding all connected components in a graph G.
- Finding all nodes within an individual connected component.
- Finding the shortest path between two nodes.

Depth First Search (DFS):

The depth-first search algorithm progresses by expanding the starting node of G and then going deeper and deeper until the goal node is found, or until a node that has no children is encountered. When a dead-end is reached, the algorithm backtracks, returning to the most recent node that has not been completely explored.

Algorithm:

Ex:



Adjacency lists
A: B, C, D
B: E
C: B, G
D: C, G
E: C, F
F: C, H
G: F, H, I
H: E, I
I: F

(a) Push H onto the stack.

STACK: H

(b) Pop and print the top element of the STACK, that is, H. Push all the neighbours of H onto the stack that are in the ready state. The STACK now becomes

PRINT: H

STACK: E, I

(c) Pop and print the top element of the STACK, that is, I. Push all the neighbours of I onto the stack that are in the ready state. The STACK now becomes

PRINT: I | STACK: E, F

(d) Pop and print the top element of the STACK, that is, F. Push all the neighbours of F onto the stack that are in the ready state. (Note F has two neighbours, C and H. But only C will be added, as H is not in the ready state.) The STACK now becomes

PRINT: F | STACK: E, C

(e) Pop and print the top element of the STACK, that is, C. Push all the neighbours of C onto the stack that are in the ready state. The STACK now becomes

PRINT: C | STACK: E, B, G

(f) Pop and print the top element of the STACK, that is, G. Push all the neighbours of G onto the stack that are in the ready state. Since there are no neighbours of G that are in the ready state, no push operation is performed. The STACK now becomes

PRINT: G | STACK: E, B

(g) Pop and print the top element of the STACK, that is, B. Push all the neighbours of B onto the stack that are in the ready state. Since there are no neighbours of B that are in the ready state, no push operation is performed. The STACK now becomes

PRINT: B | STACK: E

(h) Pop and print the top element of the STACK, that is, E. Push all the neighbours of E onto the stack that are in the ready state. Since there are no neighbours of E that are in the ready state, no push operation is performed. The STACK now becomes empty.

PRINT: E | STACK:

Since the STACK is now empty, the depth-first search of G starting at node H is complete and the nodes which were printed are:

H, I, F, C, G, B, E

These are the nodes which are reachable from the node H.

Features of DFS:

- The space complexity is lower than bfs.
- The time complexity of a dfs is proportional to the number of vertices plus the number of edges in the graphs that are traversed.
- Dfs is said to be a complete algorithm. If there is a solution, DFS will find it regardless of the kind of graph. But in case of an infinite graph, where there is no possible solution, it will diverge.

Applications of DFS:

- Finding path between two nodes.
- Finding whether a graph is connected or not.
- Computing spanning tree of a connected graph.

Connected Components and Biconnected Components:

Connected and biconnected components are used to analyze graph structures of various applications.

A connected component of an undirected graph is a subgraph in which any two vertices are connected to each other by a path, and which is connected to no additional vertices in the supergraph.

- Connected Graph: An undirected graph is said to be *connected* if there is a path between any two vertices.
- A graph may consist of multiple connected components. For example, if there are separate clusters of nodes that have no edges between them, each cluster is a connected component.

To find connected components, a simple Depth First Search (DFS) or Breadth First Search (BFS) can be used.

Applications of Graphs:

Minimum Spanning Trees:

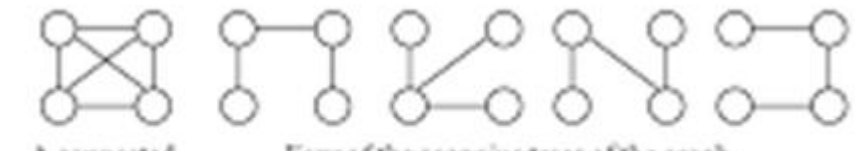
Spanning Tree:

Let G = (V, E) be an undirected connected graph. A sub-graph t = (V, E`) of G is a spanning tree of G if and only if t is a tree. A Subgraph 't' of a graph 'G' is called as a spanning tree if

(i) It includes all the vertices of 'G'

(ii) It is a tree

For a given graph 'G' there can be more than one spanning tree.



The graph with 'n' vertices, the number of spanning trees generated is 2nCn / n+1.

Minimum Cost Spanning Trees:

A tree is to be defined as an undirected, acyclic and connected graph. A graph has more than one spanning trees. A minimum spanning tree is a spanning tree, that has weights with the edges and the total weight of the tree is less. The minimum spanning tree is obtained by the use of two standard algorithms. They are;

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

Prim's Algorithm:

Prim's algorithm is a greedy algorithm that finds minimum cost spanning tree for a weighted undirected graph.

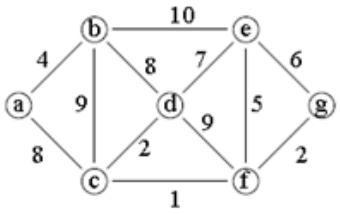
Steps:

1. Begin with any vertex which you think would be suitable and add to the tree.
2. Find an edge that connects to any vertex in the tree to any vertex that is not in the tree. Note that we don't have to form cycles and connect the vertex which has minimum weight.
3. Stop when n-1 edges have been added to the tree.

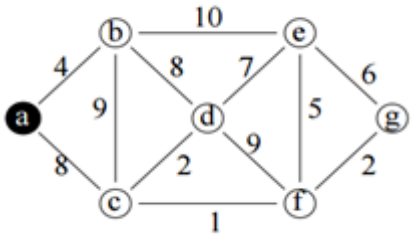
Analysis: -

The time required by the prince algorithm is directly proportional to the no. of vertices. If a graph 'G' has 'n' vertices then the time required by prim's algorithm is $0(n^2)$

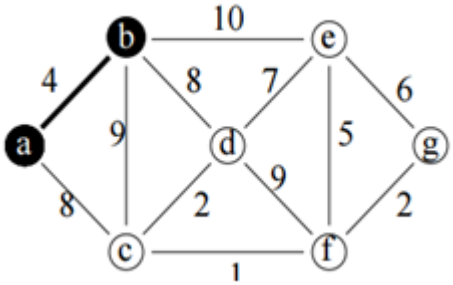
Ex: Consider the following graph and draw the minimum spanning tree.



Step1: Initially all vertices are unvisited i.e. S is empty.



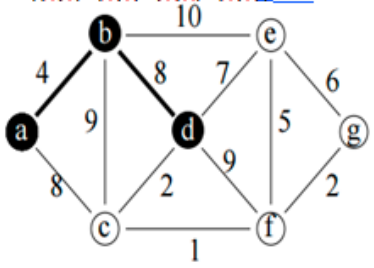
Step2: Now **a** contains two vertices b and c as neighbors. Among these edges distance from **a to b** is minimum. So include (a,b) into minimum spanning tree. Now S={a, b}. Repeat the same procedure until spanning tree contains (n-1) edges.



min((a,b) (a,c))

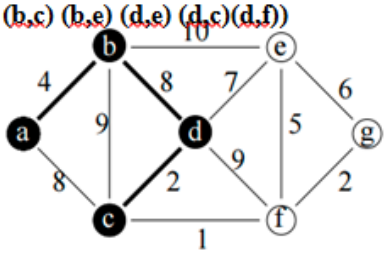
S={a, b}

3. min ((a,c) (b,c) (b,d) (b,e))



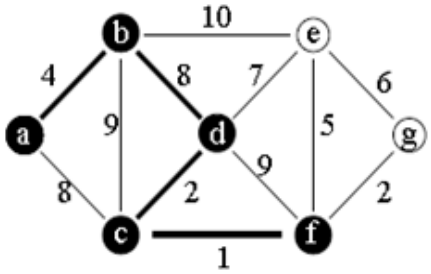
S= {{a, b} {b, d}}

4. min ((a,c), (b,c) (b,e) (d,e) (d,c)(d,f))



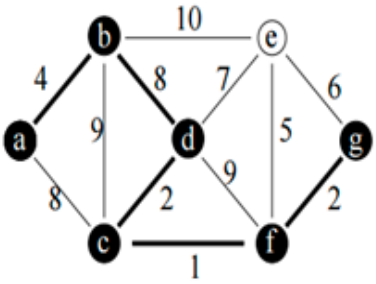
S= {{a, b} {b, d} {d, c}}

5. min ((b,e) (d,e) (d,f)(c,f))



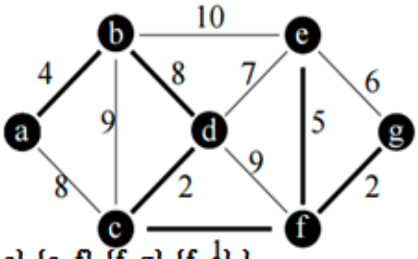
S= {{a, b} {b, d} {d, c} {c,f} }

6. min ((b,e) (d,e) (f,g), (f,e))



S= {{a, b} {b, d} {d, c} {c, f} {f, g}}

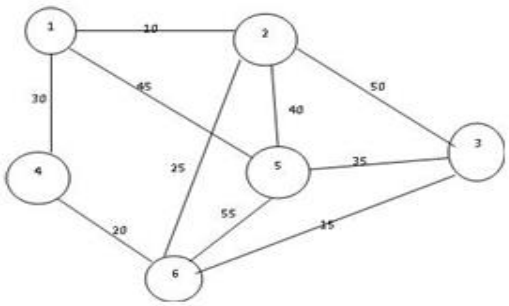
7. min ((b,e) (d,e) (f,e) (g,e))

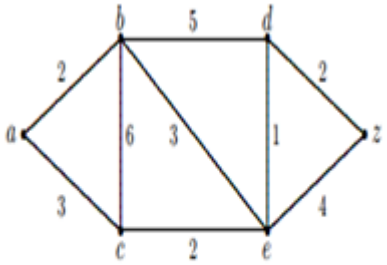


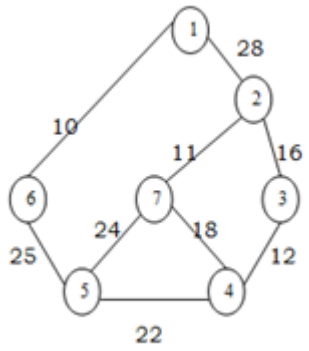
S= { {a, b} {b, d} {d, c} {c, f} {f, g} {f, e} }

Total Cost = 4 + 8 + 2+ 1 + 2 + 5 = 22

Exercise:







Kruskal's Algorithm:

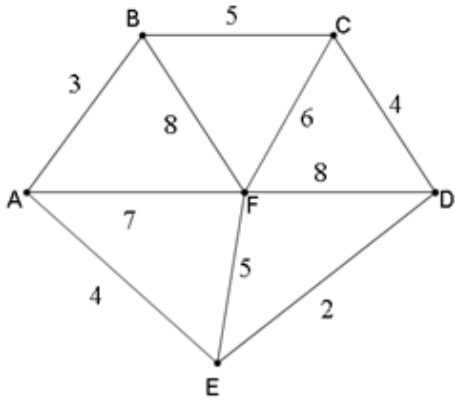
This algorithm constructs the minimum spanning tree of a graph by adding edges to the spanning tree one-by-one. This algorithm is conceptually quite simple. The edges are selected and added to the spanning tree in increasing order of their weights. An edge is added to the tree only if it does not create a cycle.

Steps:

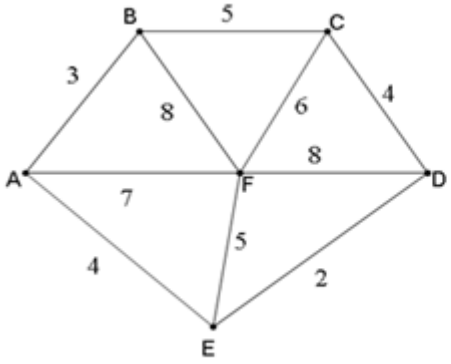
1. Arrange the edges by weight: least weight first and heaviest last.
2. Choose the lightest not examined edge from the diagram. Add this chosen edge to the tree, only if doing so will not make a cycle.
3. Stop the process whenever n-1 edges have been added to the tree.

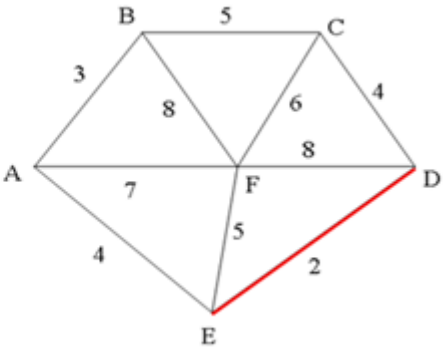
Analysis: - If the no: of edges in the graph is given by /E/ then the time for Kruskals algorithm is given by 0 (|E| log |E|).

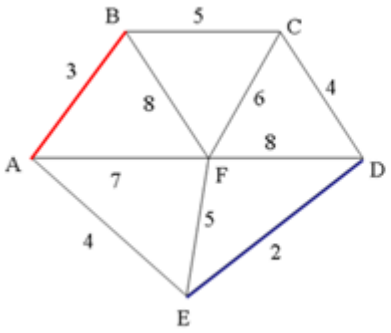
Ex:

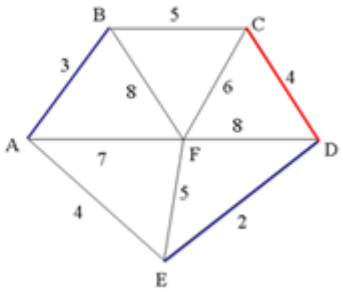


Edge	ED	AB	AE	CD	~~BC~~	EF	~~CF~~	~~AF~~	~~BF~~	~~FD~~
Weight	2	3	4	4	5	5	6	7	8	8

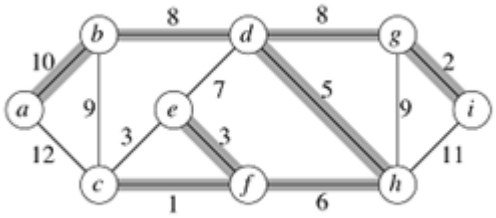


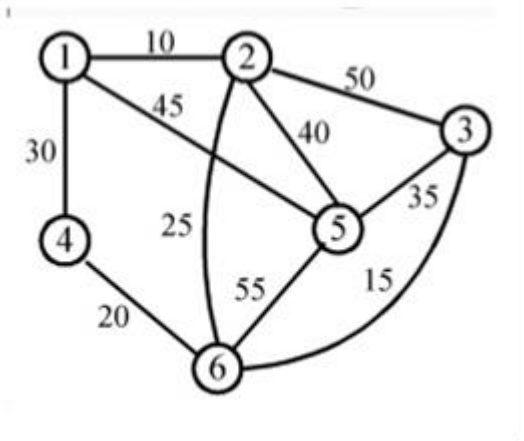


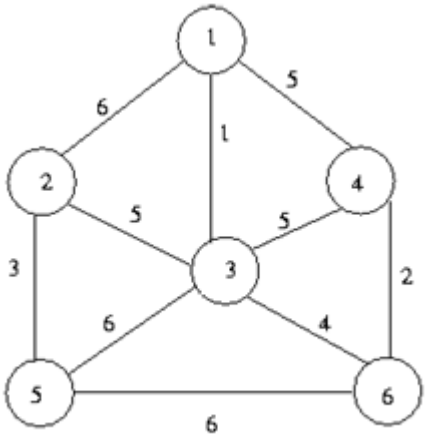




Exercise:







Differences between Prim's and Kruskal's algorithms:

S.No.	Prim's	Kruskal's
1	Select any vertex.	Select the shortest edge in a network.
2	Select the shortest edge connected to that vertex.	Select the next shortest edge which does not create a cycle.
3	Select the shortest edge connected to any vertex already connected.	Repeat step 2 until all vertices have been connected.
4	Repeat step 3 until all vertices have been connected.	

Dijkstra's shortest path: (Single Source Shortest Path)

In single source shortest path problem, the shortest distance from a single vertex is obtained. Let G(V,E) be a graph, then in single source shortest path problem, the shortest paths from vertex s to all remaining vertices is determined. The vertex S is then called a source.

Dijkstra's algorithm finds the shortest paths from a given node s to all other nodes of a weighted graph with positive weights. Node s is called a starting node.

Algorithm:

This algorithm follows Greedy approach that solves the single source shortest path problem for a directed graph with non-negative edge weights. Each node is labeled with its distance from the source node along with the best-known path. Initially all nodes are labeled with infinity since no paths are known initially. The algorithm will go on in simple steps. As algorithm processed, the paths would find, the labels will change, and reflects the better paths. The following is the stepwise procedure of algorithm.

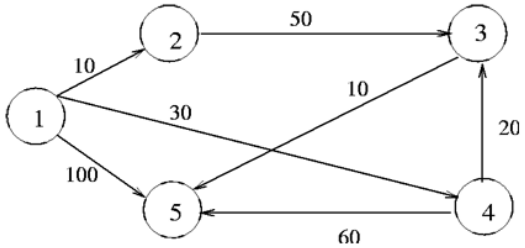
1. **Source node is initialized and can be indicated as a filled circle.**
2. **Initial path lost to neighboring nodes or link cost is computed and these nodes are re-labeled considering source node.**
3. **Examine all the adjacent nodes and find the smallest label and make it permanent.**
4. **The smallest label node is now working node, then step 2 & 3are repeated till the destination node reaches.**

Complexity of Dijkstra's Algorithm

With adjacency matrix representation, the running time is $O(n^2)$ By using an adjacency list representation and a partially ordered tree data structure for organizing the set V - S, the complexity can be shown to be O(elog n)

Where e is the number of edges and n is the number of vertices in the digraph.

Ex:



Initially:

$S = \{1\}$ $D[2] = 10$ $D[3] = \infty$ $D[4] = 30$ $D[5] = 100$

Iteration 1

Select w = 2, so that S = {1, 2}

$D[3] =$ $\min(\infty, D[2] + C[2, 3]) = 60$

$D[4] =$ $\min(30, D[2] + C[2, 4]) = 30$

$D[5] =$ $\min(100, D[2] + C[2, 5]) = 100$

Iteration 2

Select w = 4, so that S = {1, 2, 4}

$D[3] =$ $\min(60, D[4] + C[4, 3]) = 50$

$D[5] =$ $\min(100, D[4] + C[4, 5]) = 90$

Iteration 3

Select w = 3, so that S = {1, 2, 4, 3}

$D[5] =$ $\min(90, D[3] + C[3, 5]) = 60$

Iteration 4

Select w = 5, so that S = {1, 2, 4, 3, 5}

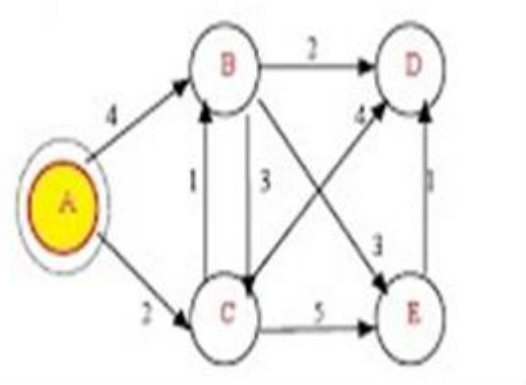
$D[2] = 10$

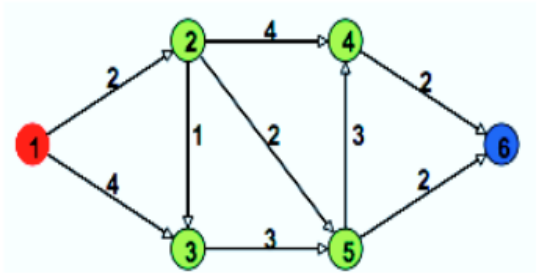
$D[3] = 50$

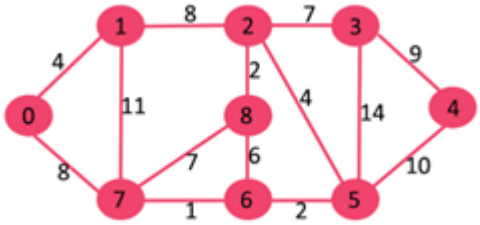
$D[4] = 30$

$D[5] = 60$

Exercise:


A
B
C
D
E
4
2
1
3
4
3
3
5


1
2
3
4
5
6
2
4
2
1
4
2
3
3
2


0
1
2
3
4
5
6
7
8
4
8
11
7
8
2
4
14
9
6
10
1
2

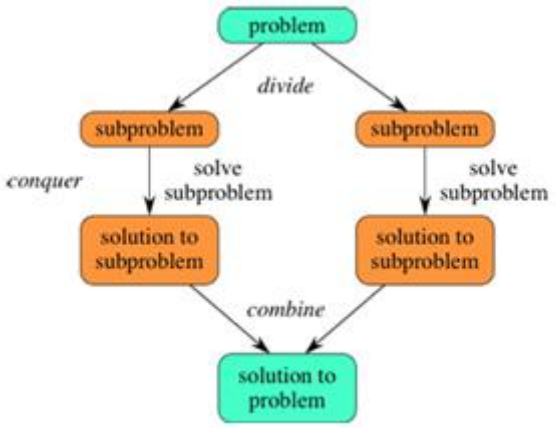
Divide and Conquer

The General Method:

A divide and conquer algorithm works by recursively breaking down a problem into two or more sub problems of similar type, until these problems become simple enough to be solved directly. The solutions to the sub-problems are then combined to give a solution to the original problem.

Control Abstraction:

```
Algorithm DAndC(P)
{
        if small (P) then
                return S(P);
        else
        {
                Divide p into small instances P1, P2,....Pn,  n>=1;
                Apply DAndC to each subproblem;
                return combine (DAndC(P1), DAndC(P2),....,DAndC(Pn));
        }
}
```



The computing time is;

$$T(n) = \begin{cases} g(n) & \text{if 'n' is small} \\ T(n1)+T(n2)+,,,,+T(nr)+Fn & \text{If 'n' is large} \end{cases}$$

Where,

T(n) - Time for divide and conquer size of 'n'

g(n) - Computing time required to small inputs

F(n) - Time required for dividing problem and combining solutions of sub problems

Applications of Divide and Conquer:

1. Defective Chess Board
2. Binary Search
3. Finding Maximum and Minimum
4. Quick Sort
5. Merge Sort
6. Strassen's Matrix Multiplication, etc.

Quick Sort:

Quick Sort is one of the sorting algorithms that uses the divide and conquer strategy. In this method division is dynamically carried out.

Steps:

1. **Divide:** Split the array into two sub arrays that each element in the left sub array is less than or equal to the middle element and each element in the right sub array is greater than the middle element. The middle element is called as pivot element.
2. **Conquer:** Recursively sort two sub arrays.
3. **Combine:** Combine all the sorted elements in a group to form a list of sorted elements.

In merge sort the division of array is based on the positions of array elements, but in quick sort the division is based on the actual value of the element.

- Quicksort is a divide-and-conquer algorithm that aims to sort an array by partitioning it into smaller subarrays, sorting them individually, and then combining them.
- The basic idea of quicksort is to select a pivot element from the array and partition the other elements into two subarrays, according to whether they are less than or greater than the pivot.
- After partitioning, the pivot is in its final sorted position, and the elements to the left of the pivot are smaller, while the elements to the right are greater.

- The process is then repeated recursively on the subarrays until the entire array is sorted.
- The choice of the pivot can significantly affect the performance of the algorithm. A good pivot choice can lead to more balanced partitions, improving the efficiency of the sorting process.
- Common strategies for choosing the pivot include selecting the first element, the last element, the middle element, or a randomly chosen element.
- Quicksort has an average and best-case time complexity of O(n log n), where n is the number of elements in the array. However, in the worst-case scenario where the pivot is consistently poorly chosen, the time complexity can degrade to O(n^2).
- Quicksort is an in-place sorting algorithm, meaning it does not require additional space proportional to the input size. It rearranges the elements within the original array.
- Quicksort is widely used in practice due to its efficiency and low space requirements.
- It is worth noting that quicksort is not a stable sorting algorithm, meaning it may change the relative order of elements with equal values.

Algorithm:

1. If the array has fewer than two elements, it is already considered sorted. Return the array.
2. Select a pivot element from the array. The choice of the pivot can vary (e.g., first element, last element, middle element, or random element).
3. Partition the array by rearranging the elements around the pivot. Move all elements smaller than the pivot to its left and all elements greater than the pivot to its right.
4. Recursively apply quicksort on the subarrays to the left and right of the pivot.
5. Combine the sorted subarrays along with the pivot element to obtain the final sorted array.

Ex: Sort the elements 50, 30, 10, 90, 80, 20, 40, 70 using Quick Sort.

Pivot

50-i, 30 10 90 80 20 40 70-j

We will now split the array into two parts. The left sublist will contains the elements less than the pivot and right sublist contains elements greater than pivot.

50 30-i 10 90 80 20 40 70-j

We will increment i. If A[i] <= pivotm we will continue to increment it until the element pointed by i is greater than pivot.

50 30 10-i 90 80 20 40 70-j

Increment i as A[i]<=piot

50 30 10 90-i 80 20 40 70-j

As A[i] > pivot we will stop incrementing i.

50 30 10 90-i 80 20 40 70-j

As A[i]>pivot we will decrement j. We will continue to decrement j until the element pointed by j is less than pivot.

50 30 10 90-i 80 20 40-j 70

Now we cannot decrement j because A[j]< pivot. Hence we will swap A[i] and A[j].

50 30 10 40-i 80 20 90-j 70

As A[i]<pivot and A[j] is greater than pivot we will continue incrementing i and decrementing j until the false conditions are occurred.

50 30 10 40 80-i 20 90-j 70

We will stop incrementing i because A[i]>pivot.

50 30 10 40 80-i 20-j 90 70

We will stop decrementing j because A[j] < pivot. Hence we will swap A[i] and A[j].

50 30 10 40 20-i 80-j 90 70

As A[i]<pivot and A[j] is greater than pivot we will continue incrementing i and decrementing j until the false conditions are occurred.

50 30 10 40 20-j 80-i 90 70

As A[j] < pivot and j has crossed i. i.e. j<i, So we will swap pivot and A[j].

20 30 10 40 **50 -P** 80 90 70

Now 20, 30, 10, 40 are less than the pivot and 80,90, 70 are greater than the pivot. Now sort the left sub list and right sub list recursively.

20-**P,i**	30	10	40-j	50	80	90	70
20	30-i	10	40-j				
20	30-j	10-j	40				
20	10-i	30-j	40				
20	10	30-i,j	40				
20	10-j	30-i	40				
10	**20-P**	30	40				
					80-P,i	**90**	70-j
					80	70-j	90-j
					80	70	90-i.j
					80	70-j	90-i
					70	**80-P**	90

Now merge two sublists for sorted list.

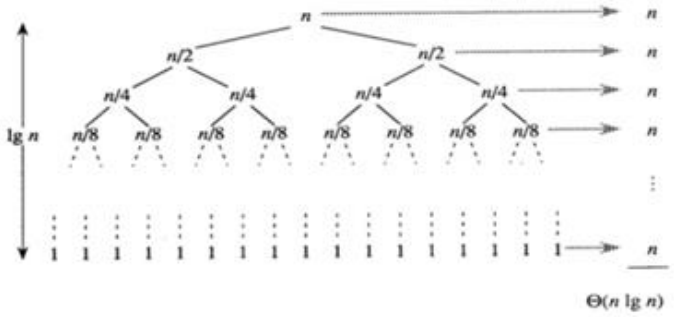
10	20	30	40	50	70	80	90

Time Complexity:

The running time of quick sort depends on whether partition is balanced or not. A very good partition splits an array into two equal sized arrays. A bad partition splits an array into two arrays of very different sizes. If the partition is balanced the quick sort runs faster otherwise it runs very slowly.

Best Case:

The best case of quick sort will happen if each portioning state divides the array exactly in half.



Thus we have k = n/2 and n − k = n/2 for the original array of size n. Consider, therefore, the recurrence:

T(n) = T(n/2) + T(n/2) + c.n if n>1

= 2T(n/2) + c.n

= 2 [2t(n/4)+ c.n/2) + c.n

= 4T (n/4) + 2c.n

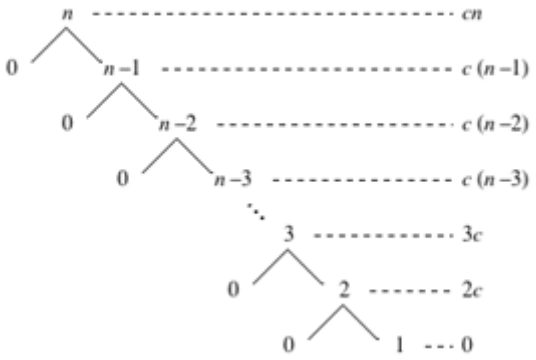
= 8T(n/8) + 3c.n

= $2^k T(n/2^k)$ + kc.n.

= n log n (Let $n/2^k = 1$ => k= $\log_2 n$)

i.e. T(n) = O(n log n)

Worst Case:



T(n) = T(n-1+ T(n-2)+ ....

= T(n-1) + c.n

= T(n-2)+ 2cn-c

= T(n-3)+ 3cn-3c

= T(n-4)+ 4cn-6c

= T(n-k)+ kcn-(1+2+3+.....+(k-1)c)

= T(n-k)+ ([k(k-1)]/2). c

= T(1) + cn^2 – ([n(n-1)]/2) C (n-k=1 => k=1)

= O(n^2)

Worst case of the quick sort is eliminated by using randomized quick sort.

The worst case for quick sort depends upon the selection of pivot element. The worst case occurs if we always select the first or last elements as the pivot.

Average Case: O(n log n)

The average case will occur if the pivot element is not a middle element or the first element or the last element.

Apply quick sort to the sort to sort the list E X A M P L E in alphabetical order. Draw the tree of recursive calls made.

0	1	2	3	4	5	6
E	X - i	A	M	P	L	E - j
E	E	A-j	M-i	P	L	X
A	E	E	M	P	L	X
A	E – i,j					
A – j	E-i					
A	E					
	E					
			M	P – i	L	X – j
			M	P – i	L - j	X
			M	L –i	P-j	X
			M	L –j	P-i	X
			L	M	P	X
			L			
					P	X – i, j
					P	X
						X

Exercise:

Show how the quick sorts the following sequence of keys in ascending order.

i. **22 55 33 11 99 77 55 66 54 21 32**

ii. **8 2 5 13 4 19 12 6 3 11 10 7 9**

iii. **65 70 75 80 85 60 55 50 45**

iv. **44 75 23 43 55 12 64 77 33**

Merge Sort:

The merge sort is a sorting algorithm that uses the divide and conquers strategy. In this method, division is carried out dynamically. We need an extra temporary array of the same size as the input array for merging.

1. Merge sort is a divide-and-conquer algorithm that aims to sort an array by dividing it into smaller subarrays, sorting them individually, and then merging them back together.
2. The basic idea of merge sort is to repeatedly divide the array in half until each subarray contains only one element, as a single element is considered sorted.
3. After dividing the array into single-element subarrays, the merge operation is performed. This operation involves merging two sorted subarrays into a single sorted array.
4. During the merge operation, two pointers are used to compare elements from the two subarrays and place them in the correct order in the merged array.
5. The merge operation continues until all the subarrays have been merged back into a single sorted array.
6. Merge sort has a time complexity of O(n log n) in all cases (worst, average, and best), where n is the number of elements in the array. This makes it one of the most efficient sorting algorithms.
7. Merge sort is a stable sorting algorithm, meaning it preserves the relative order of elements with equal values. This can be important in certain applications.
8. It is worth noting that merge sort requires additional space proportional to the size of the input array. This additional space is used for the temporary storage of subarrays during the merging process.
9. Merge sort is widely used in practice due to its efficiency and stability, especially for sorting large datasets or linked lists.

Steps:

1. **Divide:** Partition array into two sub arrays s1 and s2 with n/2 elements each.
2. **Conquer:** Recursively sort s1 and s2.
3. **Combine:** Merge s1 and s2 into a unique sorted group.

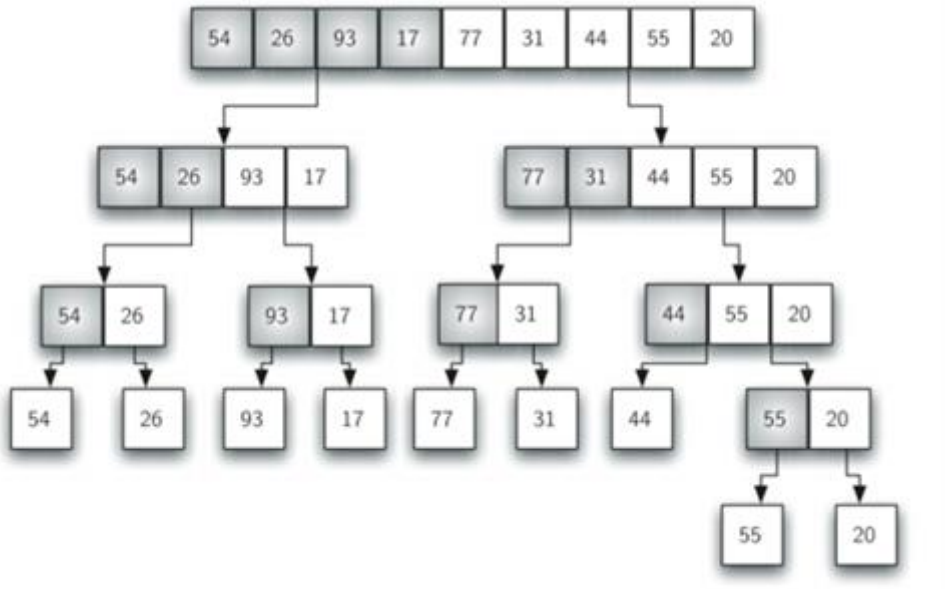
Algorithm:

1. If the array has fewer than two elements, it is already considered sorted. Return the array.
2. Divide the array into two halves. This can be done by finding the middle index of the array.

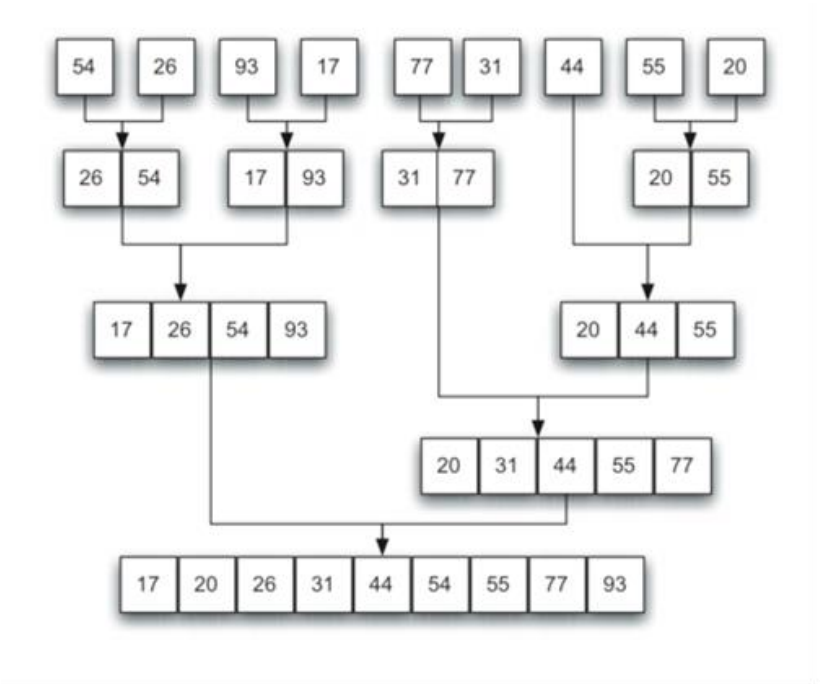
3. Recursively call merge sort on the first half of the array.
4. Recursively call merge sort on the second half of the array.
5. Merge the two sorted subarrays created from the previous steps.
    - Create an empty temporary array to store the merged subarrays.
    - Initialize two pointers, one for each subarray, pointing to the first element.
    - Compare the elements at the pointers of the subarrays and place the smaller (or larger) element into the temporary array.
    - Move the pointer of the subarray from which the element was taken.
    - Repeat the previous two steps until one of the subarrays is fully traversed.
    - Copy the remaining elements from the non-empty subarray into the temporary array.
    - Copy the elements from the temporary array back into the original array, replacing the corresponding elements.
6. Return the sorted array.

Ex: 54 26 93 17 77 31 44 55 20

i. **Splitting the list:** Merge sort is a recursive algorithm that continually splits a list in half. If the list is empty or has one element, it is sorted. If the list contains more than one item, we split the list and recursively invoke a merge sort on both halves.



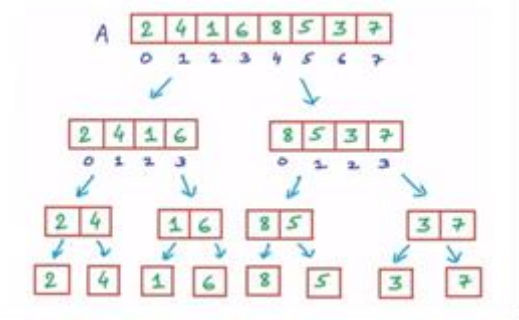
Merging the list:



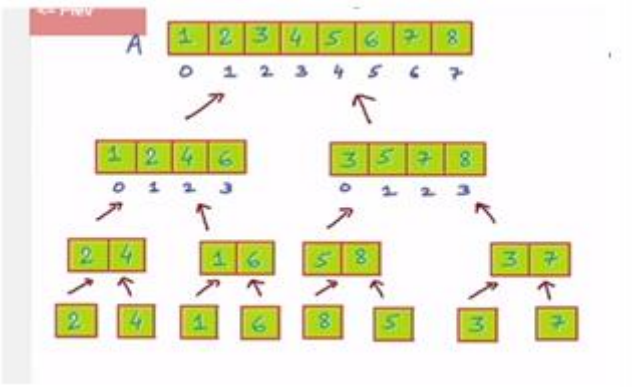
Example Problems

Sort the elements using merge sort. (2 4 1 6 8 5 3 7)

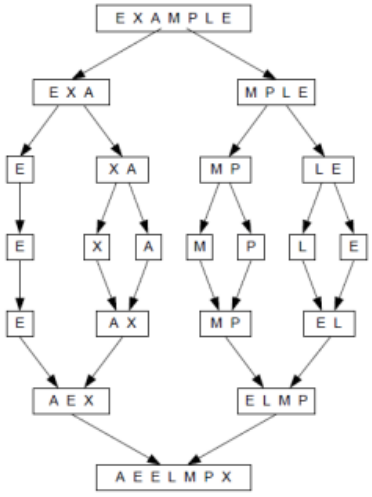
1. Split the array using recursion until the list contains single element.



2. Merging the list by performing sorting.



Draw the tree of recursive calls made by the merge sort. (EXAMPLE)



Exercise

1. **Draw the tree of calls of merge sort for the following set of elements.**
   20 30 10 45 5 60 90 45 35 25 15 55
2. **Draw the tree of calls of merge sort for the following set of elements.**
   35 25 15 10 45 75 85 65 55 5 20 18

Performance Analysis:

Time Complexity:

In merge sort algorithm the two recursive calls are made. Each recursive call focuses on n/2 elements of the list. After two recursive calls one call for combining two sublists. Hence the recurrence relation is;

$T(n) = T(n/2) + T(n/2) + cn$ for $n>1$

$T(n) = 2T(n/2)+cn$

$= 4T(n/4)+2cn$

$= 8T(n/8)+4cn$

$= 2^kT(n/2^k)+k.cn$ $((n/2^k) = 1 \Rightarrow n = 2^k \Rightarrow k = \log_2 n$

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

$= O(n \log n)$

Space Complexity:

Because of merge sort is not an in-placement algorithm, we need a stack for implementing recursion. So space complexity is O(log n)

Strassen's matrix multiplication:

Strassen's algorithm for matrix multiplication is an efficient algorithm for multiplying two matrices together. It was devised by Volker Strassen in 1969 and significantly reduces the number of arithmetic operations needed compared to the standard matrix multiplication algorithm, especially for large matrices.

The standard matrix multiplication algorithm has a time complexity of O(n^3), where n is the size of the matrices. Strassen's algorithm reduces this to approximately O(n^log 7), making it faster for large matrices.

The algorithm works by recursively breaking down the matrices into smaller submatrices and performing matrix multiplication using a set of seven multiplications instead of the usual eight. Although Strassen's algorithm is faster for large matrices, it has higher overhead due to the recursive decomposition, and it becomes less efficient for small matrices due to the constant factors involved.

Strassen's matrix multiplication follows Divide and conquer strategy and works on only square matrices, where 'n' is power of 2 and order of matrices are n x n.

- Divide: In this step each large matrix is divided into smaller square matrix.
- Conquer: Solve each and every smaller matrix.
- Merge: Merge all the results in conquer step to get final result.

In tradition method time complexity of matrix multiplication is $O(n^3)$. Time complexity of matrix multiplication is reduced by using Strassen's matrix multiplication method.

Algorithm:

1. Divide the given matrices into submatrices of n/2 x n/2. Until n=2.
2. Using scalar additions and subtractions compute the smaller matrices of size n/2.
3. Recursively compute the seven matrix products.
4. Compute the resultant matrix by using matrix products.

$$\begin{bmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{bmatrix} \begin{bmatrix} B_{11} & B_{12} \\ B_{21} & B_{22} \end{bmatrix} = \begin{bmatrix} C_{11} & C_{12} \\ C_{21} & C_{22} \end{bmatrix}$$

In traditional method C was calculated as,

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

If n=2 the traditional method requries 8 multiplcations and 4 additions. In this case time complexity is $O(n^3)$ and matrix multiplications are more expensive than matrix additions. In Strassen's matric multiplication it has only 7 multiplcations and 18 additions or subtractions. The following are the operations required for Strassen's matrix multiplication.

P = (A11 + A22) (B11 + B22)

Q = B11 (A21 + A22)

R = A11 (B12 – B22)

S = A22 (B21 – B11)

T = B22 (A11 + A12)

U = (B11 + B12) (A21 – A11)

V = (B21 + B22) (A12-A22)

C11 = P+S-T+V

C12 = R+T

C21 = Q+S

C22 = P+R-Q+U

The resulting recurrence relations for T(n) is

$$T(n) = \begin{cases} b & n \leq 2 \\ 7T(n/2) + an^2 & n > 2 \end{cases}$$

Where a and b are constants.

$$\begin{aligned} T(n) &= an^2[1 + 7/4 + (7/4)^2 + \cdots + (7/4)^{k-1}] + 7^k T(1) \\ &\leq cn^2(7/4)^{\log_2 n} + 7^{\log_2 n}, \ c \text{ a constant} \\ &= cn^{\log_2 4 + \log_2 7 - \log_2 4} + n^{\log_2 7} \\ &= O(n^{\log_2 7}) \approx O(n^{2.81}) \end{aligned}$$

Apply DandC on 4x4 matrix.

A=

A11	A12	A13	A14
A21	A22	A23	A24
A31	A32	A33	A34
A41	A42	A43	A44

4x4

B=

B11	B12	B13	B14
B21	B22	B23	B24
B31	B32	B33	B34
B41	B42	B43	B44

4x4

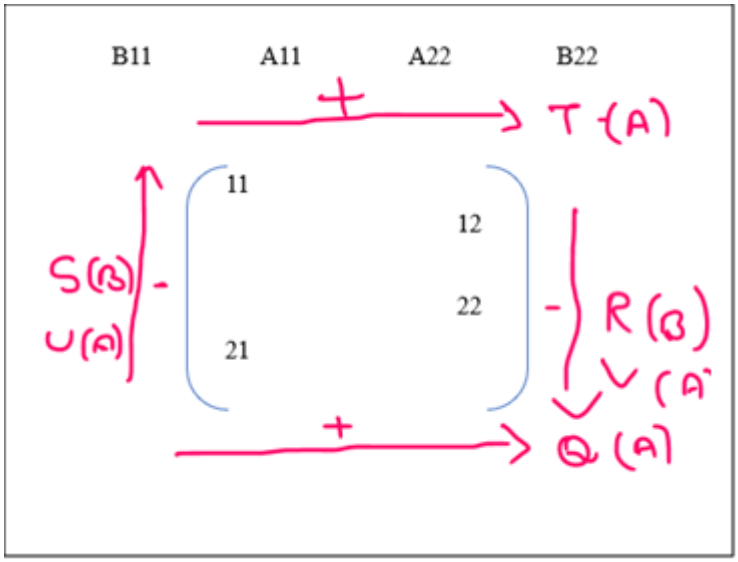
C=

C11	C12	C13	C14
C21	C22	C23	C24
C31	C32	C33	C34
C41	C42	C43	C44

Calculate C11 using A11 and B11 as follows;

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

Calculate:



P = (A11 + A22) (B11 + B22)

Q = B11 (A21 + A22)

R = A11 (B12 – B22)

S = A22 (B21 – B11)

T = B22 (A11 + A12)

U = (B11 + B12) (A21 – A11)

V = (B21 + B22) (A12-A22)

C11 = P+S-T+V

C12 = R+T

C21 = Q+S

C22 = P+R-Q+U

Recursively solve C12, C21 and C22 as per C11.

EX:

$$A = \begin{pmatrix} 1 & 2 \\ 3 & 4 \end{pmatrix} \qquad B = \begin{pmatrix} 2 & 2 \\ 3 & 1 \end{pmatrix}$$

CALCULATE C=AB.

SOLUTION:

A11 = 1 A12 = 2 A21 = 3 A22 = 4

B11 = 2 B12 = 2 B21 = 3 B22 = 1

P = (A11 + A22) (B11 + B22)

= (1+4) (2+1) = 15

Q = B11 (A21 + A22)

=2(3+4) = 14

R = A11 (B12 – B22)

= 1(2-1) =1

S = A22 (B21 – B11)

= 4(3-2) = 4

T = B22 (A11 + A12)

=1(1+2) =3

U = (B11 + B12) (A21 – A11)

= (2+2)(3-1) = 8

V = (B21 + B22) (A12-A22)

= (3+1) (2-4) = -8

C11 = P+S-T+V = 15+4-3-8 = 8

C12 = R+T = 1+3 = 4

C21 = Q+S = 14+4 = 18

C22 = P+R-Q+U = 15+1-14+8 = 10

$$C = \begin{pmatrix} 8 & 4 \\ 18 & 10 \end{pmatrix}$$

Important Questions

1. Explain the fundamental concepts of graph theory, including terminology and different representations. Discuss how the choice of representation can affect the performance of graph algorithms.
2. Apply DFS and BFS to the complete graph on four vertices. List the vertices in the order they would be visited.
3. Explain about Prim's and Kruskal's algorithms with an example.
4. Explain about Single Source Shortest Path algorithm with an example.
5. Discuss the concept of connected components in a graph. How are they identified, and what is their significance in applications such as network analysis and clustering?
6. Explain Divide and Conquer General method and analyze time complexity of recurrence relation.
7. Analyze quick sort algorithm with the help of an example and derive time complexity.
8. Show how the Quick sort sorts the following sequences of keys in ascending order.

   22, 55, 33, 11, 99, 77, 55, 66, 54, 21, 32
9. Explain merge sort with recursive algorithm and derive its time complexity.
10. Draw the tree of calls of merge for the following set of elements.
    (20, 30, 10, 40, 5, 60, 90, 45, 35, 25, 15, 55)
11. Derive the time complexity of Strassen's matrix multiplication using recurrence relation.

Short Answer Questions

1. Define Graph?
2. Define weakly connected graph.
3. Define strongly connected graph.
4. Define indegree and outdegree.
5. Define Disjoint Graphs.
6. How graphs are stored?
7. What is path, cycle, loop and Adjacency vertex of a graph.
8. What is a connected component in a graph?
9. How can you find the connected components of an undirected graph?
10. Define a biconnected component.
11. Give an example of a real-world problem that can be modeled using graphs.
12. Give an example of a problem that can be solved using the divide and conquer approach.
13. How is the pivot element chosen in Quick Sort?
14. Explain the time complexity of Quick Sort in the best and worst cases.
15. What is the time complexity of Merge Sort, and why is it consistent?
16. Compare Merge Sort with Quick Sort in terms of stability.
17. What problem does Strassen's algorithm solve, and how is it more efficient than standard matrix multiplication?
18. How does Strassen's algorithm reduce the number of multiplications required?
19. What is the time complexity of Strassen's matrix multiplication algorithm?
20. In what situations might the classical matrix multiplication algorithm be preferred over Strassen's algorithm?