UNIT – II

Graphs – Terminology, Representations, Basic Search and Traversals, Connected Components and Biconnected Components, applications

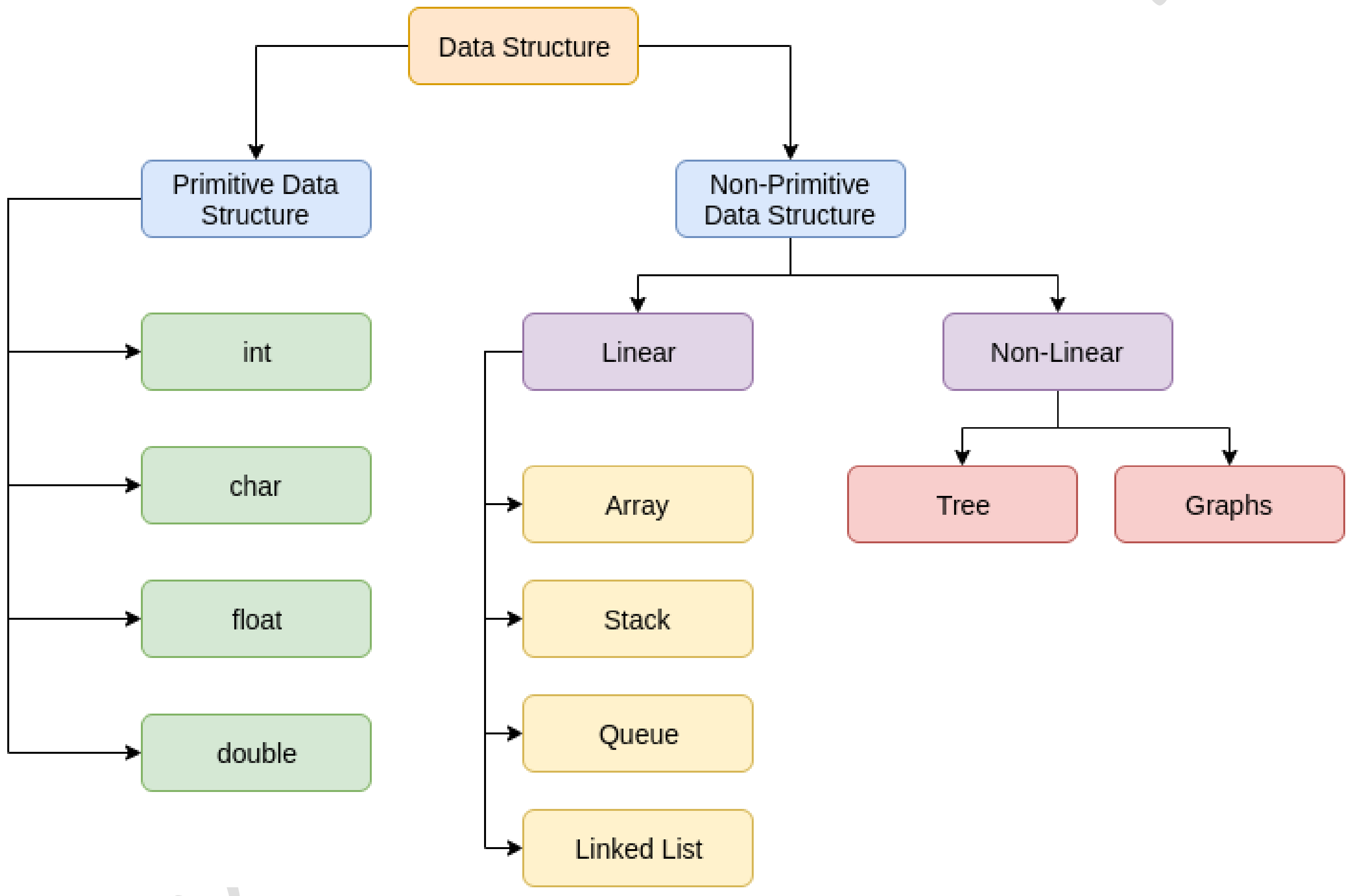
Divide and Conquer: The General Method, Quick Sort, Merge Sort, Strassen's matrix multiplication

....

Data structures are the ways of organizing and storing data in a computer so that we can perform several operations efficiently on it. It is widely used in every aspect of computer science.

Data structures are primarily categorized into two parts:

1. Primitive Data Structures
2. Non-Primitive Data Structures

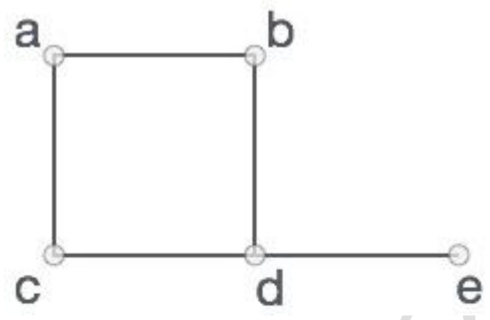


1. What is Graph?

A Graph is a non-linear data structure consisting of vertices and edges. The vertices are sometimes also referred to as nodes and the edges are lines or arcs that connect any two nodes in the graph. More formally a Graph is composed of a set of vertices(V) and a set of edges(E). The graph is denoted by G(V, E)

Components of a Graph

- **Vertices:** Vertices are the fundamental units of the graph. Sometimes, vertices are also known as vertex or nodes. Every node/vertex can be labeled or unlabelled.
- **Edges:** Edges are drawn or used to connect two nodes of the graph. It can be ordered pair of nodes in a directed graph. Edges can connect any two nodes in any possible way. There are no rules. Sometimes, edges are also known as arcs. Every edge can be labelled/unlabelled.



In the above graph,

V = {a, b, c, d, e}

E = {ab, ac, bd, cd, de}

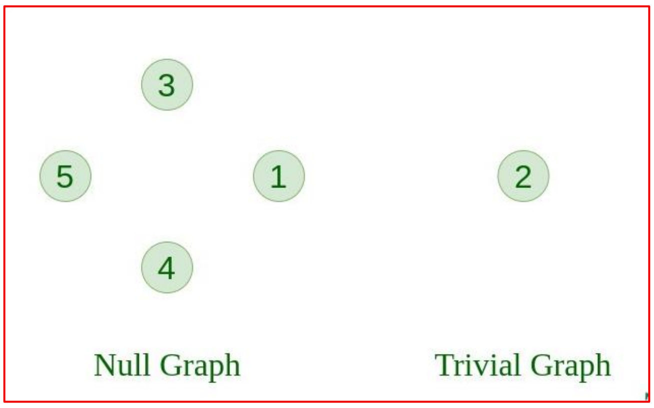
2. Types of Graphs

1. Null Graph

A graph is known as a null graph if there are no edges in the graph.

2. Trivial Graph

Graph having only a single vertex, it is also the smallest graph possible.

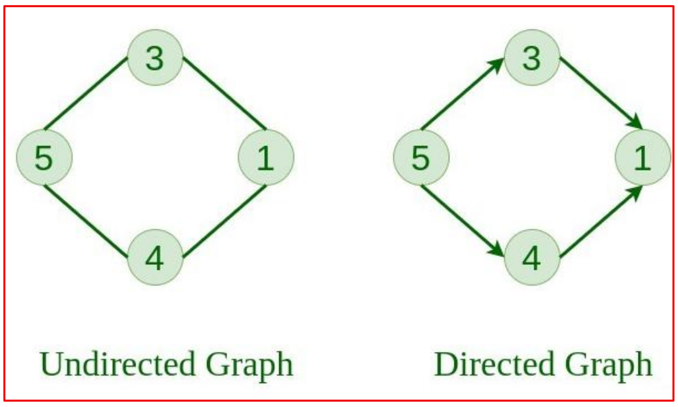


3. Undirected Graph

A graph in which edges do not have any direction. That is the nodes are unordered pairs in the definition of every edge.

4. Directed Graph

A graph in which edge has direction. That is the nodes are ordered pairs in the definition of every edge.

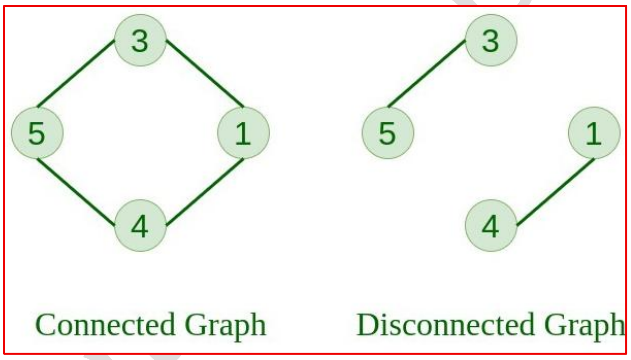


5. Connected Graph

The graph in which from one node we can visit any other node in the graph is known as a connected graph.

6. Disconnected Graph

The graph in which at least one node is not reachable from a node is known as a disconnected graph.

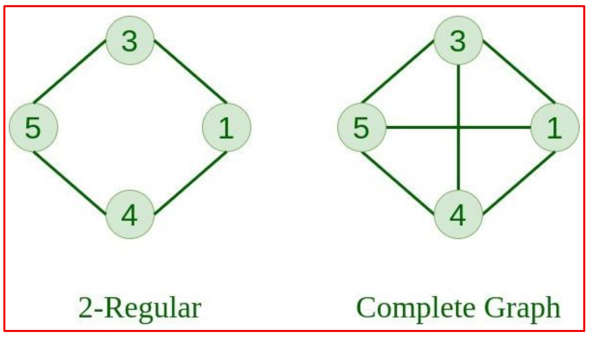


7. Regular Graph

The graph in which the degree of every vertex is equal to K is called K regular graph.

8. Complete Graph

The graph in which from each node there is an edge to each other node.

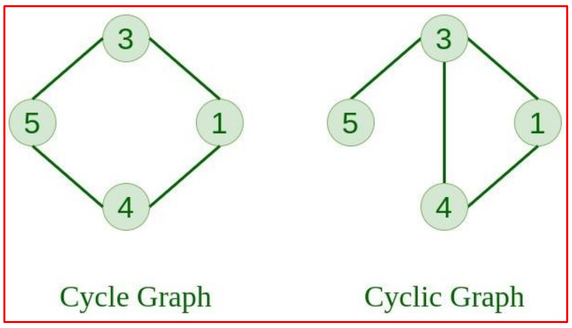


9. Cycle Graph

The graph in which the graph is a cycle in itself, the degree of each vertex is 2.

10. Cyclic Graph

A graph containing at least one cycle is known as a Cyclic graph.

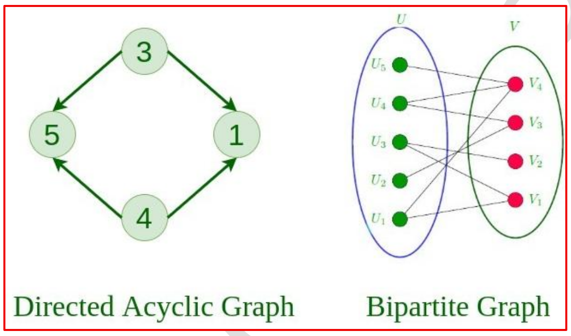


11. Directed Acyclic Graph

A Directed Graph that does not contain any cycle.

12. Bipartite Graph

A graph in which vertex can be divided into two sets such that vertex in each set does not contain any edge between them.

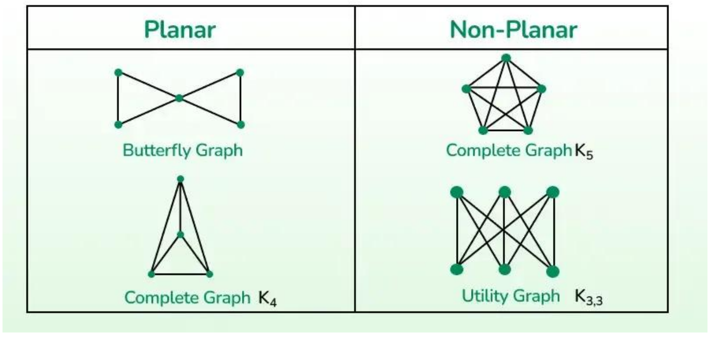


13. Weighted Graph

- A graph in which the edges are already specified with suitable weight is known as a weighted graph.
- Weighted graphs can be further classified as directed weighted graphs and undirected weighted graphs.

14. Planar graph

A planar graph is a graph that can be drawn in a plane or on a sphere so that no two edges intersect, except possibly at a vertex where they meet. In other words, it can be drawn without any edge crossings.



Faces: A planar graph divides the plane into regions, which are called faces.

In graph theory, a **utility graph** refers to the graph K3,3. It's a graph with six vertices, divided into two sets of three, and nine edges connecting each vertex in one set to every vertex in the other set. The utility graph is also known as the Thomsen graph and is relevant to the Three Utilities Problem, which asks if it's possible to connect three houses to three utilities (water, electricity, gas) without any crossings.

15. **Dense and Sparse graphs**

a dense graph is a graph in which the number of edges is close to the maximal number of edges (where every pair of vertices is connected by one edge). The opposite, a graph with only a few edges, is a sparse graph.

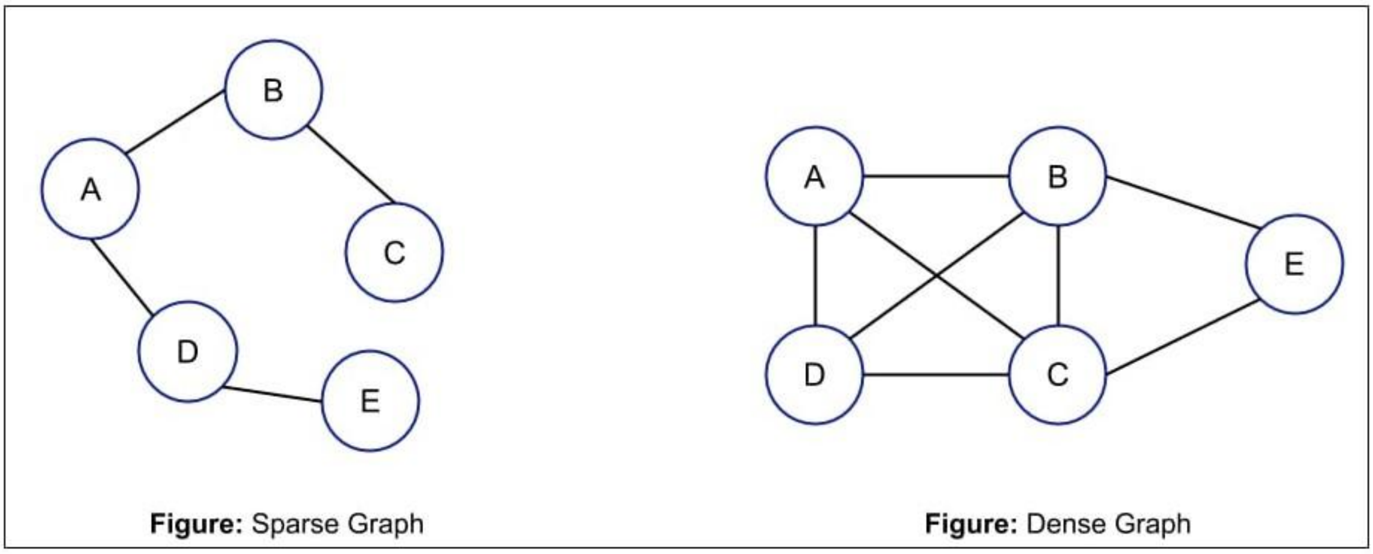


Figure: Sparse Graph **Figure:** Dense Graph

15. Clique: In an undirected graph G = (V, E), a clique is a subset of vertices in a graph where every vertex is connected to every other vertex in that subset. Essentially, all vertices within a clique are directly adjacent to each other.

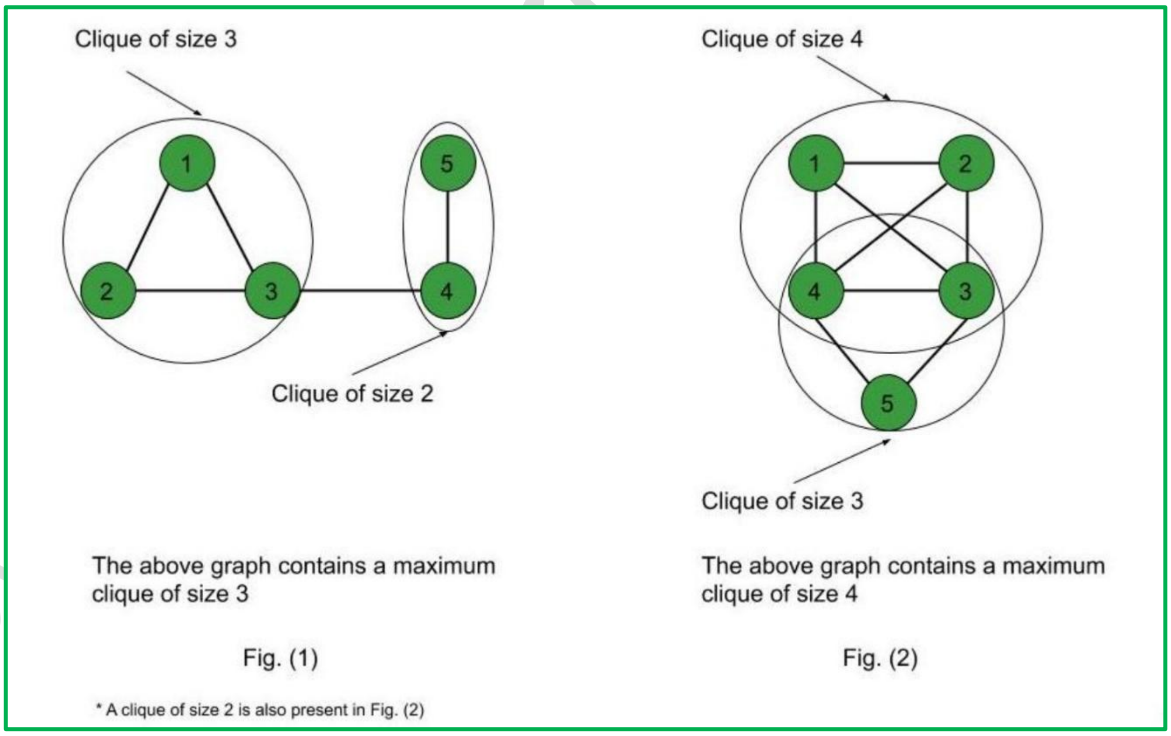

Clique of size 3
1
2
3
5
4
Clique of size 2
Clique of size 4
1
2
4
3
5
Clique of size 3
The above graph contains a maximum clique of size 3
The above graph contains a maximum clique of size 4

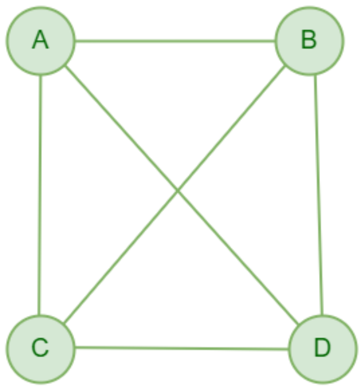
Fig. (1) Fig. (2)

* A clique of size 2 is also present in Fig. (2)

To find the number of cliques in a graph, graph traversal algorithms such as depth-first search (DFS) or breadth-first search (BFS) can be used to visit all the vertices and check for cliques at each vertex.

Example 1:

Consider a simple undirected graph with 4 vertices and 6 edges, as shown below:



Number of cliques = n * (n - 1) / 2 - m + 1 where n is the number of vertices in the graph and m is the number of edges. Plugging in the values for this graph, we get:

Number of cliques = 4 * (4 - 1) / 2 - 6 + 1 = 2

16. Size of a graph: The size of a graph is the total number of edges in it.

Directed Graphs

A directed graph G, also known as a *digraph*, is a graph in which every edge has a direction assigned

to it. An edge of a directed graph is given as an ordered pair (u, v) of nodes in G. For an edge (u, v),

- The edge begins at u and terminates at v.
- u is known as the origin or initial point of e. Correspondingly, v is known as the destination or terminal point of e.
- u is the predecessor of v. Correspondingly, v is the successor of u.
- Nodes u and v are adjacent to each other.

13.3.1 Terminology of a Directed Graph

☞ ***Out-degree of a node*** The out-degree of a node u, written as outdeg(u), is the number of edges that originate at u.
☞ ***In-degree of a node*** The in-degree of a node u, written as indeg(u), is the number of edges that terminate at u.
☞ ***Degree of a node*** The degree of a node, written as deg(u), is equal to the sum of in-degree and
☞ out-degree of that node. Therefore, deg(u) = indeg(u) + outdeg(u).
☞ ***Isolated vertex*** A vertex with degree zero. Such a vertex is not an end-point of any edge.
☞ ***Pendant vertex*** (also known as leaf vertex) A vertex with degree one.
☞ ***Cut vertex*** A vertex which when deleted would disconnect the remaining graph.
☞ ***Source*** A node u is known as a source if it has a positive out-degree but a zero in-degree.
☞ ***Sink*** A node u is known as a sink if it has a positive in-degree but a zero out-degree.
☞ ***Reachability*** A node v is said to be reachable from node u, if and only if there exists a (directed)
☞ path from node u to node v.
☞ ***Strongly connected directed graph*** A digraph is said to be strongly connected if and only if there exists a path between every pair of nodes in G. That is, if there is a path from node u to v, then there must be a path from node v to u.
☞ ***Unilaterally connected graph*** A digraph is said to be unilaterally connected if there exists a path between any pair of nodes u, v in G such that there is a path from u to v or a path from v to u, but not both.

☞ **Weakly connected digraph**: A directed graph is said to be weakly connected if it is connected by ignoring the direction of edges. That is, in such a graph, it is possible to reach any node from any other node by traversing edges in any direction (may not be in the direction they point). The nodes in a weakly connected directed graph must have either out-degree or in-degree of at least 1.

Transitive Closure of a Directed Graph

A transitive closure of a graph is constructed to answer reachability questions. That is, is there a path from a node A to node E in one or more hops? A binary relation indicates only whether the node A is connected to node B, whether node B is connected to node C, etc. But once the transitive closure is constructed as shown in Fig. 13.6, we can easily determine in O(1) time whether node E is reachable from node A or not.

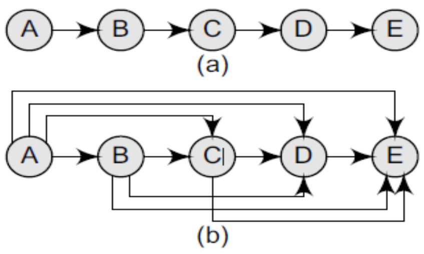


Figure 13.6 (a) A graph G and its (b) transitive closure G*

3. Graph Representations

Here are the two most common ways to represent a graph.

1. Adjacency Matrix
2. Adjacency List

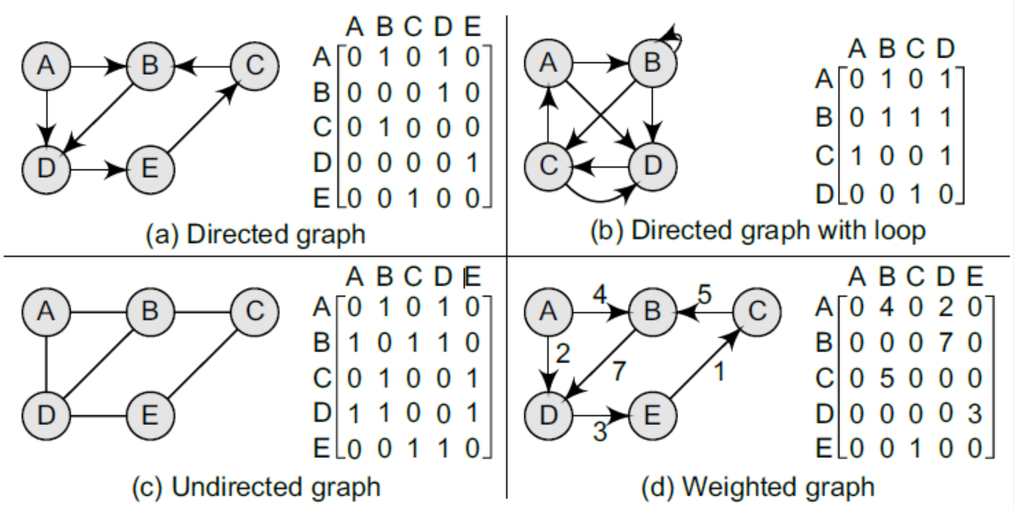
3.1 A adjacency Matrix Representation

An adjacency matrix is used to represent which nodes are adjacent to one another. By definition, two nodes are said to be adjacent if there is an edge connecting them.

In a directed graph G, if node v is adjacent to node u, then there is definitely an edge from u to v. That is, if v is adjacent to u, we can get from u to v by traversing one edge. For any graph G having n nodes, the adjacency matrix will have the dimension of n x n.

In an adjacency matrix, the rows and columns are labelled by graph vertices. An entry a_{ij} in the adjacency matrix will contain 1, if vertices vi and vj are adjacent to each other. However, if the nodes are not adjacent, a_{ij} will be set to zero.

$$a_{ij} \begin{cases} 1 & \text{[if } v_i \text{ is adjacent to } v_j \text{, that is there is an edge } (v_i, v_j)\text{]A} \\ 0 & \text{[otherwise]} \end{cases}$$



$$\begin{array}{c|ccccc} & A & B & C & D & E \\ \hline A & 0 & 1 & 0 & 1 & 0 \\ B & 0 & 0 & 0 & 1 & 0 \\ C & 0 & 1 & 0 & 0 & 0 \\ D & 0 & 0 & 0 & 0 & 1 \\ E & 0 & 0 & 1 & 0 & 0 \end{array} \qquad \begin{array}{c|cccc} & A & B & C & D \\ \hline A & 0 & 1 & 0 & 1 \\ B & 0 & 1 & 1 & 1 \\ C & 1 & 0 & 0 & 1 \\ D & 0 & 0 & 1 & 0 \end{array}$$

$$\begin{array}{c|ccccc} & A & B & C & D & E \\ \hline A & 0 & 1 & 0 & 1 & 0 \\ B & 1 & 0 & 1 & 1 & 0 \\ C & 0 & 1 & 0 & 0 & 1 \\ D & 1 & 1 & 0 & 0 & 1 \\ E & 0 & 0 & 1 & 1 & 0 \end{array} \qquad \begin{array}{c|ccccc} & A & B & C & D & E \\ \hline A & 0 & 4 & 0 & 2 & 0 \\ B & 0 & 0 & 0 & 7 & 0 \\ C & 0 & 5 & 0 & 0 & 0 \\ D & 0 & 0 & 0 & 0 & 3 \\ E & 0 & 0 & 1 & 0 & 0 \end{array}$$

Figure 13.14 Graphs and their corresponding adjacency matrices

From the above examples, we can draw the following conclusions:

- For a simple graph (that has no loops), the adjacency matrix has 0s on the diagonal.
- The adjacency matrix of an undirected graph is symmetric.
- The memory use of an adjacency matrix is $O(n^2)$, where n is the number of nodes in the graph.
- Number of 1s (or non-zero entries) in an adjacency matrix is equal to the number of edges in the graph.
- The adjacency matrix for a weighted graph contains the weights of the edges connecting the nodes.

13.5.2 Adjacency List Representation

An adjacency list is another way in which graphs can be represented in the computer's memory. This structure consists of a list of all nodes in G. Furthermore, every node is in turn linked to its own list that contains the names of all other nodes that are adjacent to it.

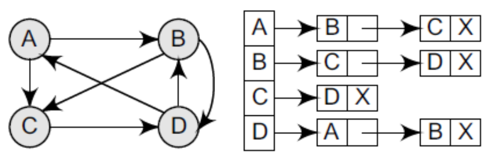


Figure 13.17 Graph G and its adjacency list

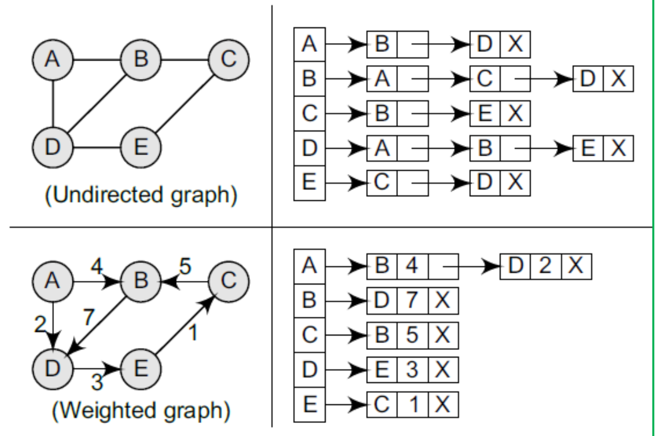


Figure 13.18 Adjacency list for an undirected graph and a weighted graph

The key advantages of using an adjacency list are:

- It is easy to follow and clearly shows the adjacent nodes of a particular node.
- It is often used for storing graphs that have a small-to-moderate number of edges. That is,
- an adjacency list is preferred for representing sparse graphs in the computer's memory; otherwise, an adjacency matrix is a good choice.
- Adding new nodes in G is easy and straightforward when G is represented using an adjacency list. Adding new nodes in an adjacency matrix is a difficult task, as the size of the matrix needs to be changed and existing nodes may have to be reordered.

4. Graph Traversal Algorithms

By traversing a graph, we mean the method of examining the nodes and edges of the graph. There are two standard methods of graph traversal. These two methods are:

1. Breadth-first search Traversal

2. Depth-first search Traversal

While breadth-first search uses a queue as an auxiliary data structure to store nodes for further processing, the depth-first search scheme uses a stack.

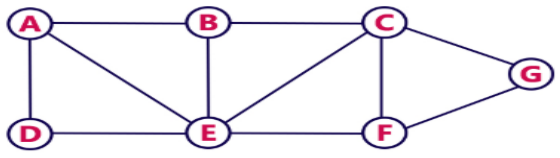
1. Breadth-first search Traversal

BFS traversal of a graph produces a **spanning tree** as final result. **Spanning Tree** is a graph without loops. We use **Queue data structure** with maximum size of total number of vertices in the graph to implement BFS traversal.

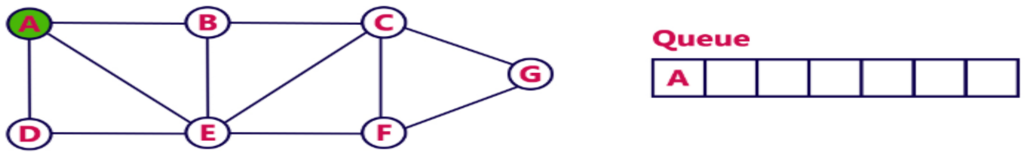
We use the following steps to implement BFS traversal...

- **Step 1 -** Define a Queue of size total number of vertices in the graph.
- **Step 2 -** Select any vertex as starting point for traversal. Visit that vertex and insert it into the Queue.
- **Step 3 -** Visit all the non-visited adjacent vertices of the vertex which is at front of the Queue and insert them into the Queue.
- **Step 4 -** When there is no new vertex to be visited from the vertex which is at front of the Queue then delete that vertex.
- **Step 5 -** Repeat steps 3 and 4 until queue becomes empty.
- **Step 6 -** When queue becomes empty, then produce final spanning tree by removing unused edges from the graph

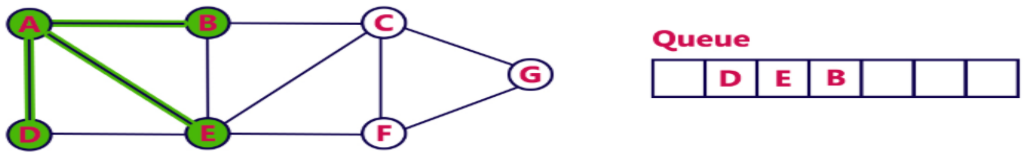
Consider the following example graph to perform BFS traversal



Step 1:
- Select the vertex **A** as starting point (visit **A**).
- Insert **A** into the Queue.

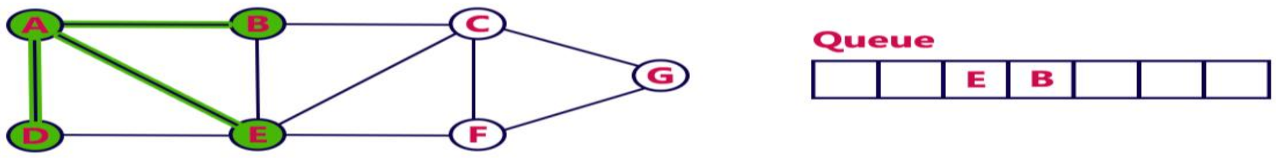


Step 2:
- Visit all adjacent vertices of **A** which are not visited (**D**, **E**, **B**).
- Insert newly visited vertices into the Queue and delete A from the Queue..



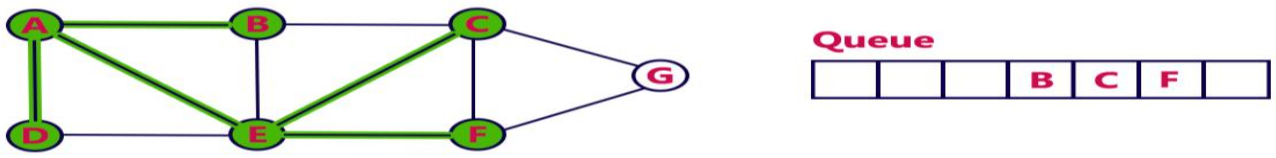
Step 3:

- Visit all adjacent vertices of **D** which are not visited (there is no vertex).
- Delete D from the Queue.



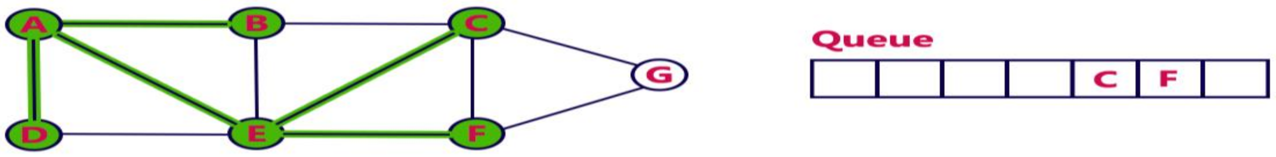
Step 4:

- Visit all adjacent vertices of **E** which are not visited (**C**, **F**).
- Insert newly visited vertices into the Queue and delete E from the Queue.



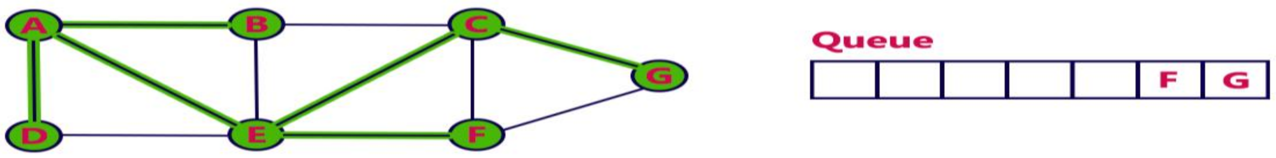
Step 5:

- Visit all adjacent vertices of **B** which are not visited (**there is no vertex**).
- Delete **B** from the Queue.



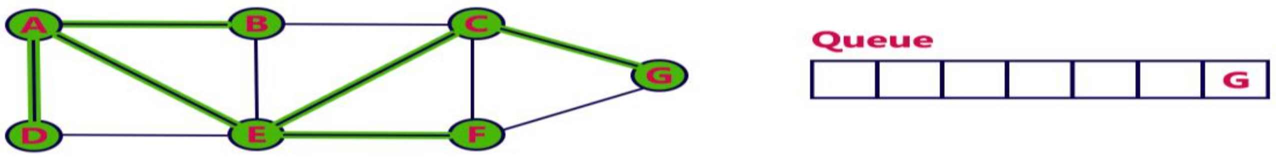
Step 6:

- Visit all adjacent vertices of **C** which are not visited (**G**).
- Insert newly visited vertex into the Queue and delete **C** from the Queue.



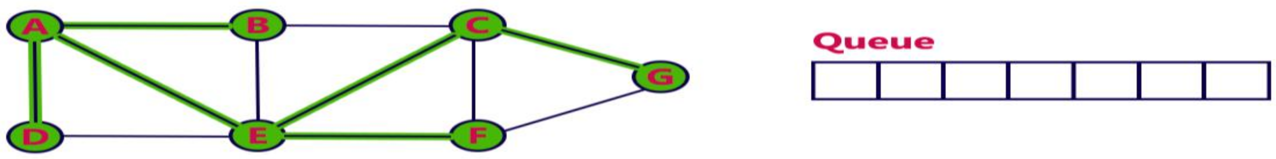
Step 7:

- Visit all adjacent vertices of **F** which are not visited (**there is no vertex**).
- Delete **F** from the Queue.

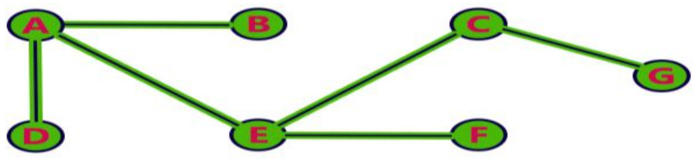


Step 8:

- Visit all adjacent vertices of **G** which are not visited (**there is no vertex**).
- Delete **G** from the Queue.



- Queue became Empty. So, stop the BFS process.
- Final result of BFS is a Spanning Tree as shown below...



Features of Breadth-First Search Algorithm

Space complexity: In the breadth-first search algorithm, all the nodes at a particular level must be saved until their child nodes in the next level have been generated.

The space complexity is therefore, proportional to the number of nodes at the deepest level of the graph. Given a graph with branching factor b (number of children at each node) and depth d, the asymptotic space complexity is the number of nodes at the deepest level $O(b^d)$.

If the number of vertices and edges in the graph are known ahead of time, the space complexity can also be expressed as O (| E | + | V |), where | E | is the total number of edges in G and | V | is the number of nodes or vertices.

Time complexity: In the worst case, breadth-first search has to traverse through all paths to all possible nodes, thus the time complexity of this algorithm asymptotically approaches O(bd).

However, the time complexity can also be expressed as O(| E | + | V |), since every vertex and every edge will be explored in the worst case.

Completeness: Breadth-first search is said to be a complete algorithm because if there is a solution, breadth-first search 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.

Optimality: Breadth-first search is optimal for a graph that has edges of equal length, since it always returns the result with the fewest edges between the start node and the goal node. But generally, in real-world applications, we have weighted graphs that have costs associated with each edge, so the goal next to the start does not have to be the cheapest goal available.

Applications of Breadth-First Search Algorithm

Breadth-first search can be used to solve many problems such as:

- Finding all connected components in a graph G.
- Finding all nodes within an individual connected component.
- Finding the shortest path between two nodes, u and v, of an unweighted graph.
- Finding the shortest path between two nodes, u and v, of a weighted graph.

2. Depth-first search Traversal

DFS traversal of a graph produces a **spanning tree** as final result. **Spanning Tree** is a graph without loops. We use **Stack data structure** with maximum size of total number of vertices in the graph to implement DFS traversal.

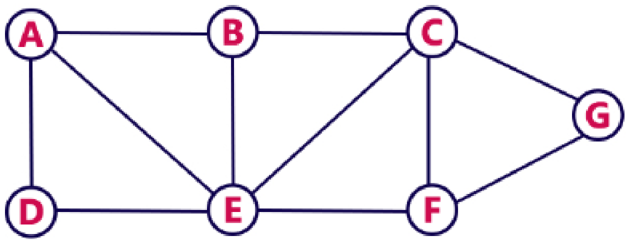
We use the following steps to implement DFS traversal...

- **Step 1 -** Define a Stack of size total number of vertices in the graph.
- **Step 2 -** Select any vertex as starting point for traversal. Visit that vertex and push it on to the Stack.
- **Step 3 -** Visit any one of the non-visited adjacent vertices of a vertex which is at the top of stack and push it on to the stack.
- **Step 4 -** Repeat step 3 until there is no new vertex to be visited from the vertex which is at the top of the stack.
- **Step 5 -** When there is no new vertex to visit then use **back tracking** and pop one vertex from the stack.
- **Step 6 -** Repeat steps 3, 4 and 5 until stack becomes Empty.

- **Step 7 -** When stack becomes Empty, then produce final spanning tree by removing unused edges from the graph

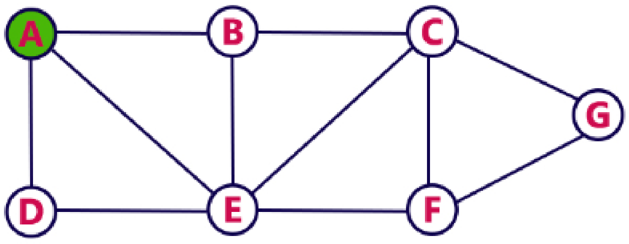
Note: Back tracking is coming back to the vertex from which we reached the current vertex.

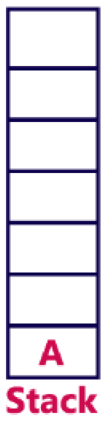
Consider the following example graph to perform DFS traversal



Step 1:

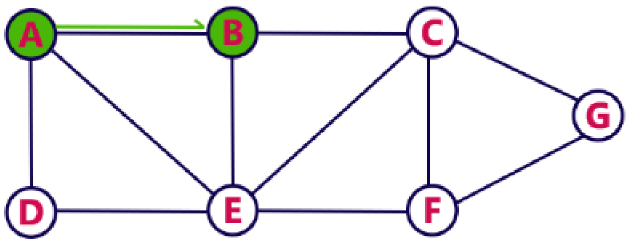
- Select the vertex **A** as starting point (visit **A**).
- Push **A** on to the Stack.

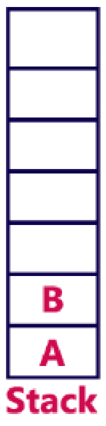




Step 2:

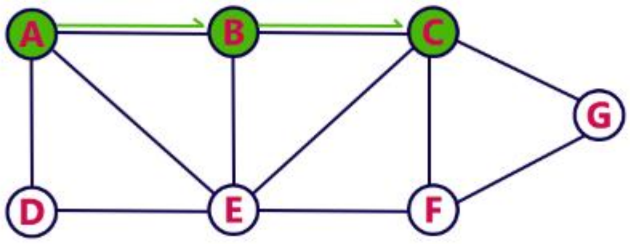
- Visit any adjacent vertex of **A** which is not visited (**B**).
- Push newly visited vertex B on to the Stack.

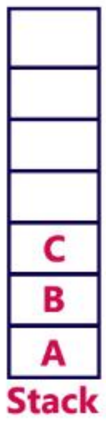




Step 3:

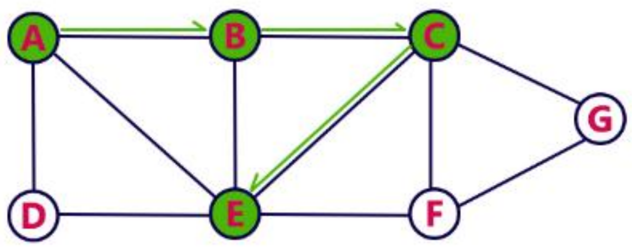
- Visit any adjacent vertext of **B** which is not visited (**C**).
- Push C on to the Stack.

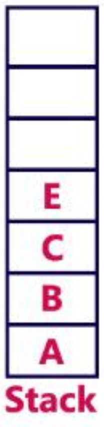




Step 4:

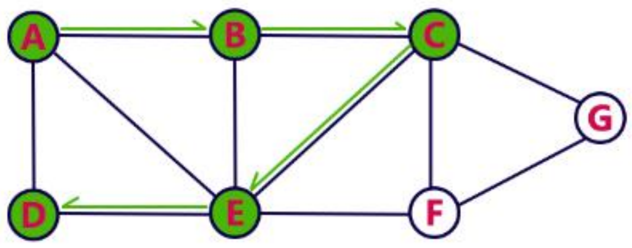
- Visit any adjacent vertext of **C** which is not visited (**E**).
- Push E on to the Stack

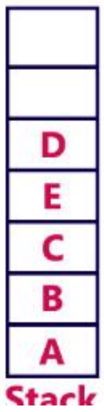




Step 5:

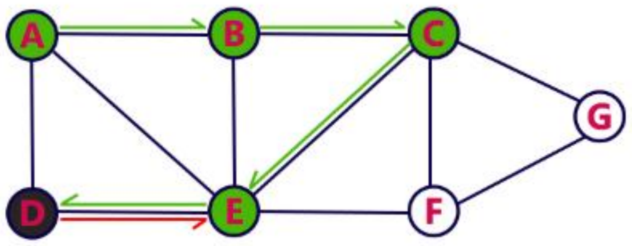
- Visit any adjacent vertext of **E** which is not visited (**D**).
- Push D on to the Stack

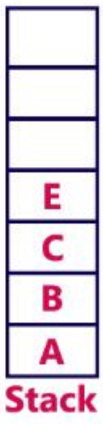




Step 6:

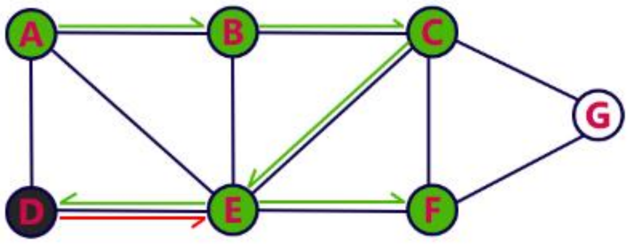
- There is no new vertiex to be visited from D. So use back track.
- Pop D from the Stack.

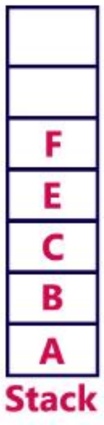




Step 7:

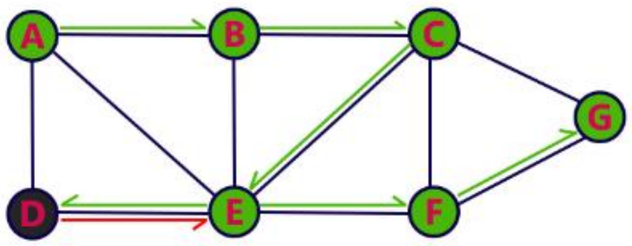
- Visit any adjacent vertex of **E** which is not visited (**F**).
- Push **F** on to the Stack.

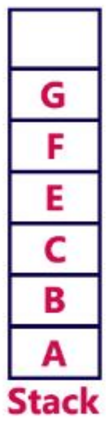




Step 8:

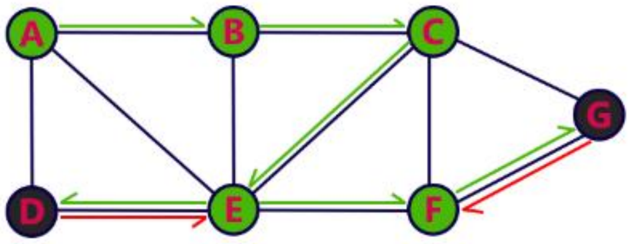
- Visit any adjacent vertex of **F** which is not visited (**G**).
- Push **G** on to the Stack.

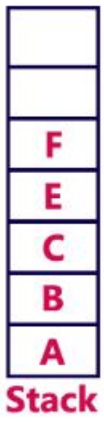




Step 9:

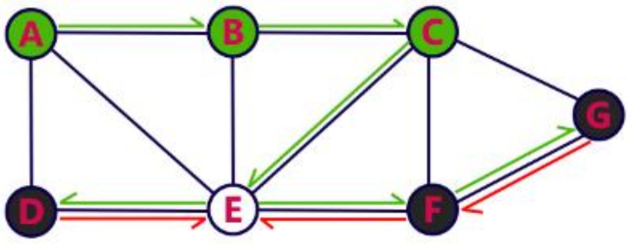
- There is no new vertiex to be visited from G. So use back track.
- Pop G from the Stack.

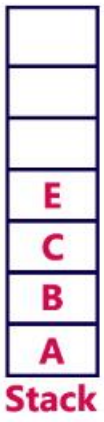




Step 10:

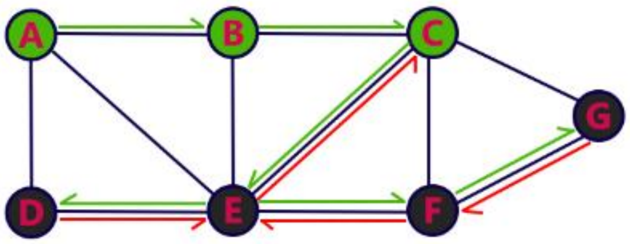
- There is no new vertiex to be visited from F. So use back track.
- Pop F from the Stack.

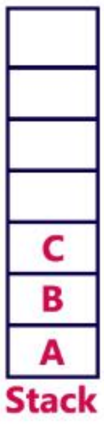




Step 11:

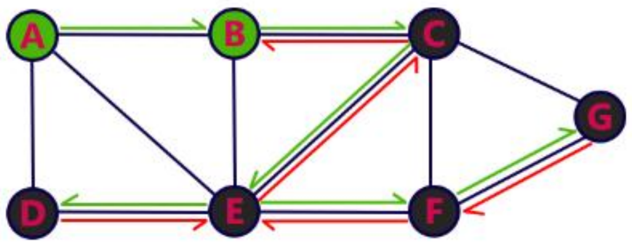
- There is no new vertiex to be visited from E. So use back track.
- Pop E from the Stack.

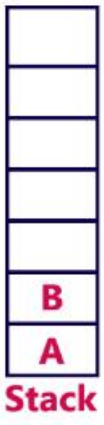




Step 12:

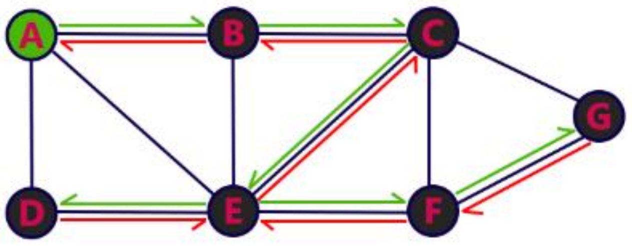
- There is no new vertiex to be visited from C. So use back track.
- Pop C from the Stack.

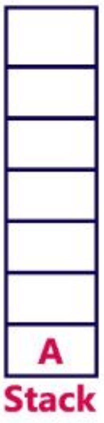




Step 13:

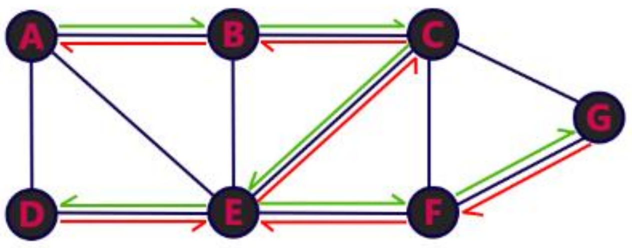
- There is no new vertiex to be visited from B. So use back track.
- Pop B from the Stack.

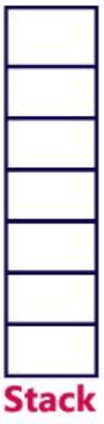




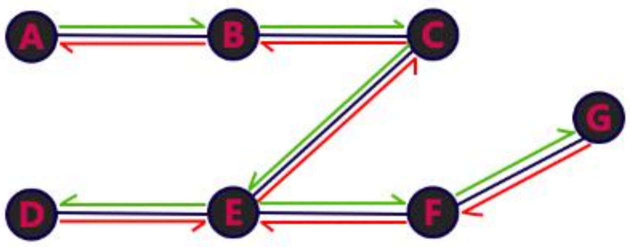
Step 14:

- There is no new vertiex to be visited from A. So use back track.
- Pop A from the Stack.





- Stack became Empty. So stop DFS Treversal.
- Final result of DFS traversal is following spanning tree.



Features of Depth-First Search Algorithm

Space complexity The space complexity of a depth-first search is lower than that of a breadth first search.

Time complexity The time complexity of a depth-first search is proportional to the number of vertices plus the number of edges in the graphs that are traversed. The time complexity can be given as (O(|V| + |E|)).

Completeness Depth-first search is said to be a complete algorithm. If there is a solution, depth first search 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 Depth-First Search Algorithm

Depth-first search is useful for:

- Finding a path between two specified nodes, u and v, of an unweighted graph.
- Finding a path between two specified nodes, u and v, of a weighted graph.
- Finding whether a graph is connected or not.
- Computing the spanning tree of a connected graph.

4. Connected Components and Biconnected Components

4.1.BI-CONNECTED components

A vertex v of G is called an articulation point, if removing v along with the edges incident on v, results in a graph that has at least two connected components.

A bi-connected graph (shown in Fig. 13.10) is defined as a connected graph that has no articulation vertices. That is, a bi-connected graph is connected and non-separable in the sense that even if we remove any vertex from the graph, the resultant graph is still connected. By definition,

A bi-connected undirected graph is a connected graph that cannot be broken into disconnected pieces by deleting any single vertex.

In a bi-connected directed graph, for any two vertices v and w, there are two directed paths from v to w which have no vertices in common other than v and w.

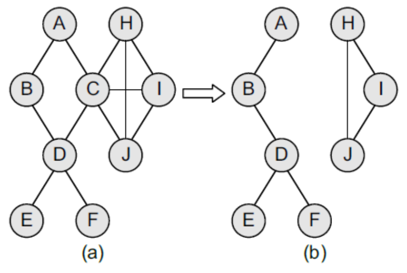


Figure 13.9 Non bi-connected graph

Note that the graph shown in Fig. 13.9(a) is not a bi-connected graph, as deleting vertex C from the graph results in two disconnected components of the original graph (Fig. 13.9(b)).

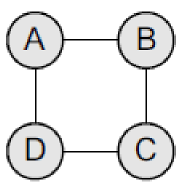


Figure 13.10 Bi-connected graph

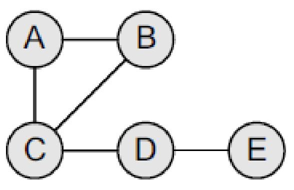


Figure 13.11 Graph with bridges

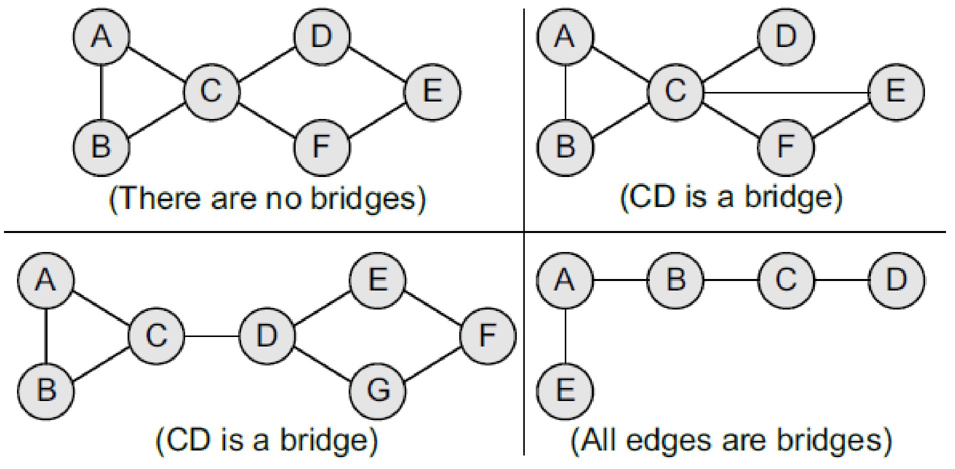


Figure 13.12 Graph with bridges

As for vertices, there is a related concept for edges.

An edge in a graph is called a *bridge* if removing that edge results in a disconnected graph. Also, an

edge in a graph that does not lie on a cycle is a bridge. This means that a bridge has at least one articulation point at its end, although it is not necessary that the articulation point is linked to a bridge. Look at the graph shown in Fig. 13.11. In the graph, CD and DE are bridges. Consider some more examples shown in Fig. 13.12.

Connected Component:

A connected component is a subgraph of a graph where a path exists between any two of its vertices. A graph can have multiple connected components if there are disjoint sets of vertices where no path exists between them.

5. APP LICATIONS OF GRAPHS

Graphs are constructed for various types of applications such as:

- ☞ In circuit networks where points of connection are drawn as vertices and component wires become the edges of the graph.
- ☞ In transport networks where stations are drawn as vertices and routes become the edges of the graph.
- ☞ In maps that draw cities/states/regions as vertices and adjacency relations as edges.
- ☞ In program flow analysis where procedures or modules are treated as vertices and calls to these procedures are drawn as edges of the graph.
- ☞ Once we have a graph of a particular concept, they can be easily used for finding shortest paths, project planning, etc.
- ☞ In flowcharts or control-flow graphs, the statements and conditions in a program are represented as nodes and the flow of control is represented by the edges.

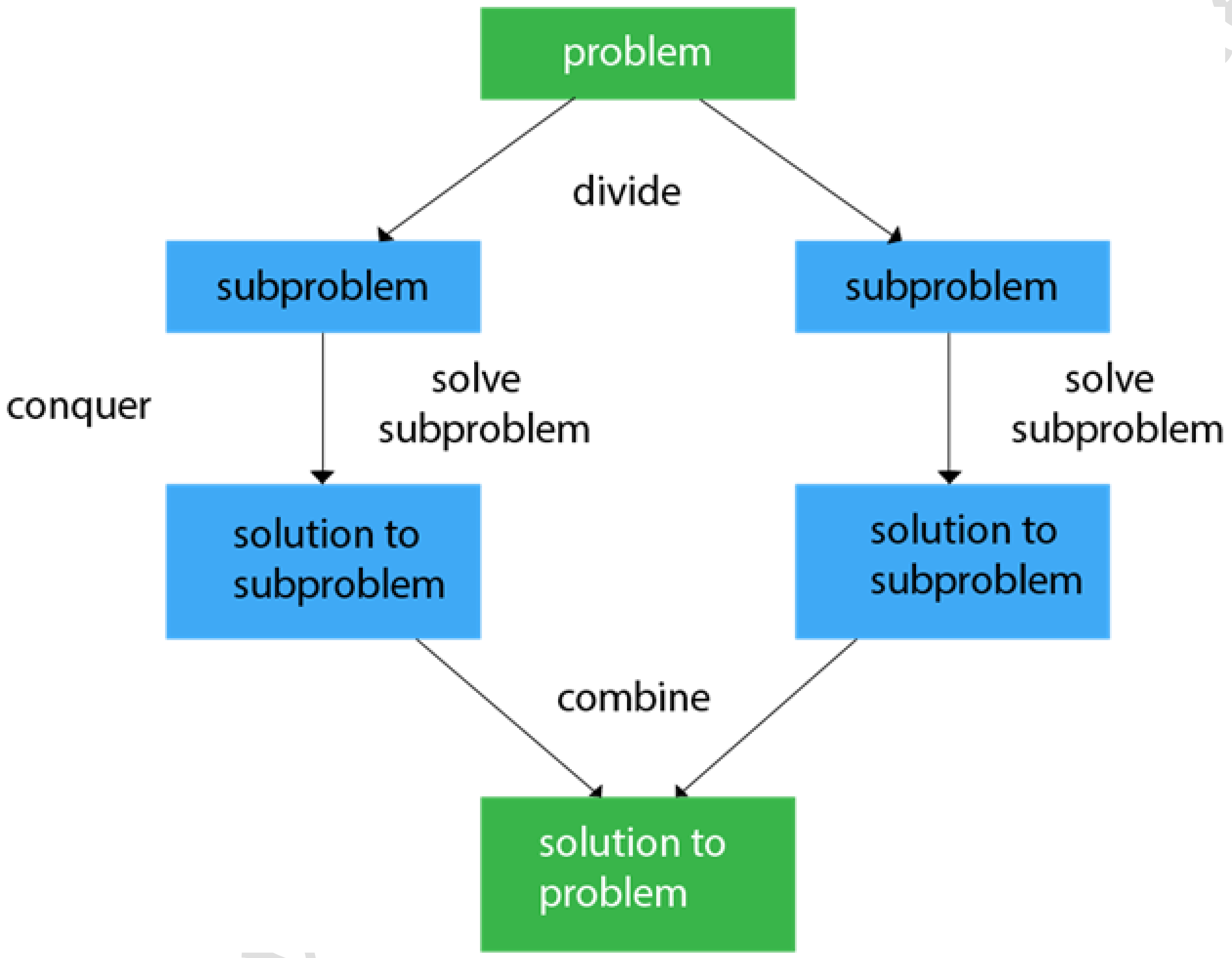
Chapter-II

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

....

Introduction to divide and conquer approach

In divide and conquer approach, the problem in hand, is divided into smaller sub-problems and then each problem is solved independently. When we keep on dividing the subproblems into even smaller sub-problems, we may eventually reach a stage where no more division is possible. Those "atomic" smallest possible sub-problem (fractions) are solved. The solution of all sub-problems is finally merged in order to obtain the solution of an original problem.



Broadly, we can understand **divide-and-conquer** approach in a three-step process.

Divide/Break

This step involves breaking the problem into smaller sub-problems. Sub-problems should represent a part of the original problem. This step generally takes a recursive approach to divide the problem until no sub-problem is further divisible. At this stage, sub-problems become atomic in nature but still represent some part of the actual problem.

Conquer/Solve

This step receives a lot of smaller sub-problems to be solved. Generally, at this level, the problems are considered 'solved' on their own.

Merge/Combine

When the smaller sub-problems are solved, this stage recursively combines them until they formulate a solution of the original problem. This algorithmic approach works recursively and conquer & merge steps works so close that they appear as one.

Examples

The following computer algorithms are based on **divide-and-conquer** programming approach –

- Merge Sort
- Quick Sort
- Binary Search
- Strassen's Matrix Multiplication
- Closest pair (points)

There are various ways available to solve any computer problem, but the mentioned are a good example of divide and conquer approach.

Advantages of Divide and Conquer

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

Disadvantages of Divide and Conquer

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

1. General Method

Divide and conquer is a design strategy which is well known to breaking down efficiency barriers. When the method applies, it often leads to a large improvement in time complexity. For example, from O (n2) to O (n log n) to sort the elements.

Divide and conquer strategy is as follows: divide the problem instance into two or more smaller instances of the same problem, solve the smaller instances recursively, and assemble the solutions to form a solution of the original instance. The recursion stops when an instance is reached which is too small to divide. When dividing the instance, one can either use whatever division comes most easily to hand or invest time in making the division carefully so that the assembly is simplified.

Control Abstraction of Divide and Conquer

A control abstraction is a procedure whose flow of control is clear but whose primary operations are specified by other procedures whose precise meanings are left undefined. The control abstraction for divide and conquer technique is DANDC(P), where P is the problem to be solved.

```
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 sub problems;
return (COMBINE (DANDC (p1) , DANDC (p2),…., DANDC (pk));
}
}
```

SMALL (P) is a Boolean valued function which determines whether the input size is small enough so that the answer can be computed without splitting. If this is so function 'S' is invoked otherwise, the problem 'p' into smaller sub problems. These sub problems p1, p2, . . . , pk are solved by recursive application of DANDC.

If the sizes of the two sub problems are approximately equal then the computing time of DANDC is:

$$T(n) = \begin{cases} g(n) & n \text{ small} \\ 2T(n/2)+f(n) & n \text{ otherwise} \end{cases}$$

Were,

T (n) is the time for DANDC on 'n' inputs

g (n) is the time to complete the answer directly for small inputs and f(n) is the time for Divide and Combine

2. Merge sort

The merge sort is a sorting algorithm that uses the divide and conquer strategy. In this method division is dynamically carried out.

Sorting by merging is a recursive strategy, divide-and-conquer strategy. In the base case, we have a sequence with exactly one element in it. Since such a sequence is already sorted, there is nothing to be done. To sort a sequence of elements (n>1)

- Divide the sequence into two sequences of length n/2 and n/2
- Recursively sort each of the two subsequence's; and then
- Merge the sorted subsequence's to obtain the final list.

Example : Consider the elements as

70, 20, 30, 40, 10, 50, 60

Now we will split this list into two sublists.

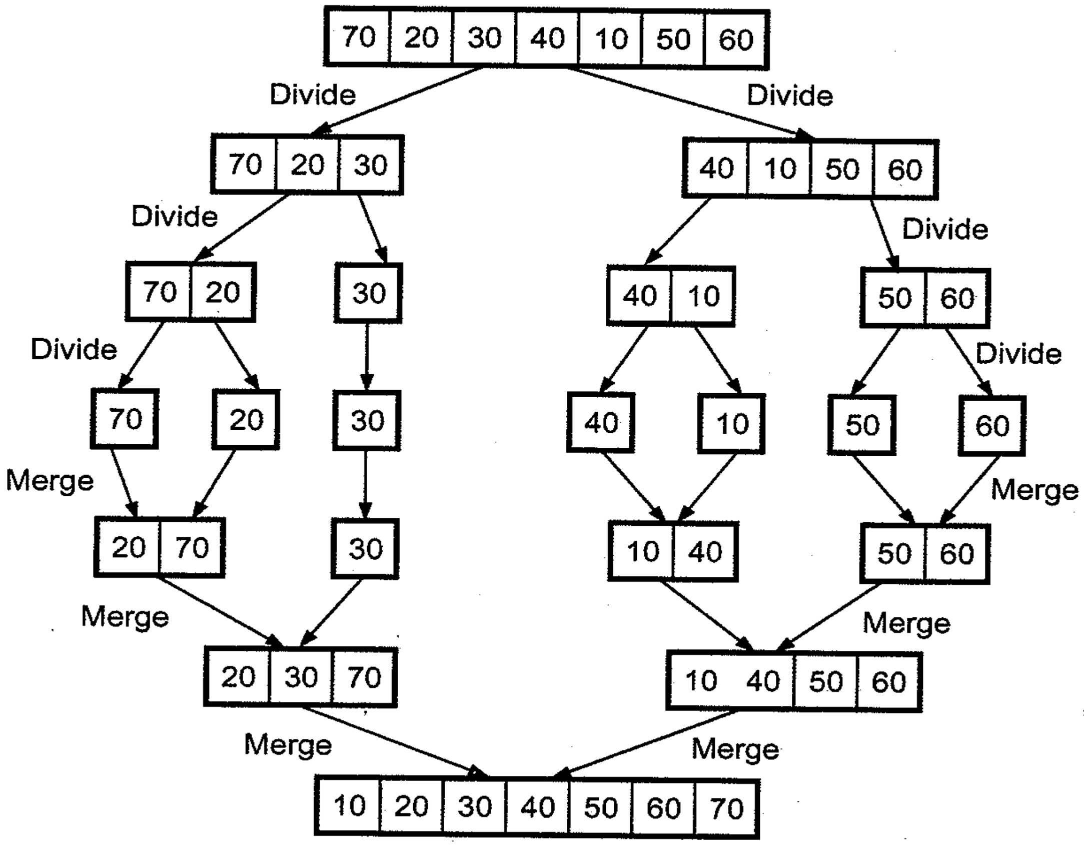


Fig. 3.7.1

Divide: The divide step just computes the middle of the subarray. Which takes constant time, O(1)

Conquer: We recursively solve two subproblems, each size n/2, which contributes T(n2/)+T(n/2)

Combine: The merge procedure on an n-element subarray takes time O(n)

```
void merge_sort (int A[ ] , int start , int end )
  {
        if( start < end ) {
        int mid = (start + end ) / 2 ;          // defines the current array in 2 parts .
        merge_sort (A, start , mid ) ;              // sort the 1st part of array .
        merge_sort (A,mid+1 , end ) ;            // sort the 2nd part of array.

       // merge the both parts by comparing elements of both the parts.
        merge(A,start , mid , end );
  }
}
```

Merge sort is a stable sorting algorithm. A sorting is said to be stable if it preserves the ordering of similar elements after applying sorting method. And merge sort is a method preserves this kind of ordering. Hence merge sort is a stable sorting algorithm.

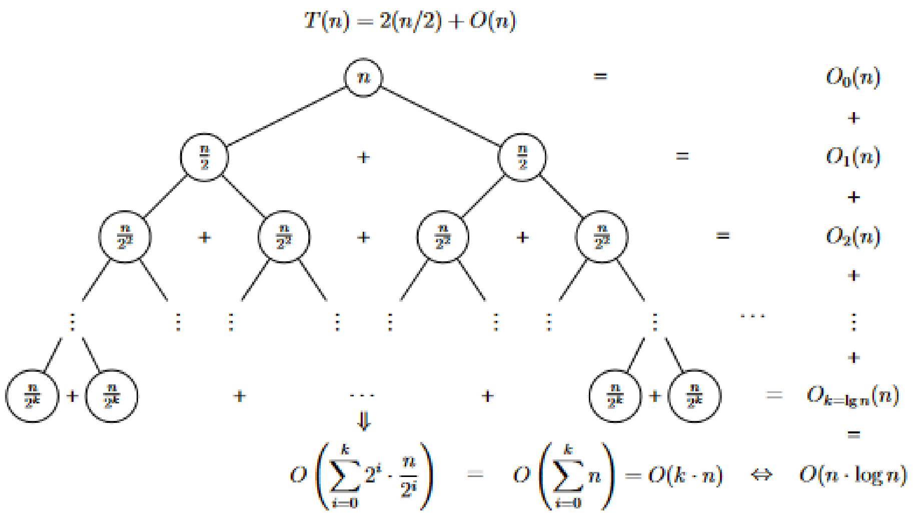
Drawbacks:

- This algorithm requires extra storage to execute this method
- This method is slower than the quick sort method
- This method is complicated to code.

Time Complexity: O(n log(n)), Sorting arrays on different machines. Merge Sort is a recursive algorithm and time complexity can be expressed as following recurrence relation.

T(n) = 2T(n/2) + θ(n)

Example: Merge Sort Recursion Tree


$T(n) = 2(n/2) + O(n)$
n
$\frac{n}{2}$
$\frac{n}{2^2}$
$\frac{n}{2^k}$
$O_0(n)$
$O_1(n)$
$O_2(n)$
$O_{k=\lg n}(n)$
$O\left(\sum_{i=0}^{k} 2^i \cdot \frac{n}{2^i}\right) = O\left(\sum_{i=0}^{k} n\right) = O(k \cdot n) \Leftrightarrow O(n \cdot \log n)$

Analysis

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 is made to combine two sublists i.e. to merge all n elements. Hence we can write recurrence relation as

$$T(n) = \underbrace{T(n/2)}_{\text{Time taken by left sublist to get sorted}} + \underbrace{T(n/2)}_{\text{Time taken by right sublist to get sorted}} + \underbrace{cn}_{\text{Time taken for combining two sublists}}$$

where n > 1 T (1) = 0

We can obtain the time complexity of Merge Sort using two methods

1. Master theorem 2. Substitution method

1. Using master theorem :

Let, the recurrence relation for merge sort is

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

i.e. $$T(n) = 2T(n/2) + cn \quad \text{... (3.7.1)}$$

$$T(1) = 0 \quad \text{... (3.7.2)}$$

As per Master theorem

$$T(n) = \Theta(n^d \log n) \qquad \text{if } a = b$$

As in equation (3.6.1),

$$a = 2, b = 2 \text{ and } f(n) = cn \text{ i.e. } n^d \text{ with } d = 1.$$

and $$a = b^d$$

i.e. $$2 = 2^1$$

This case gives us

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

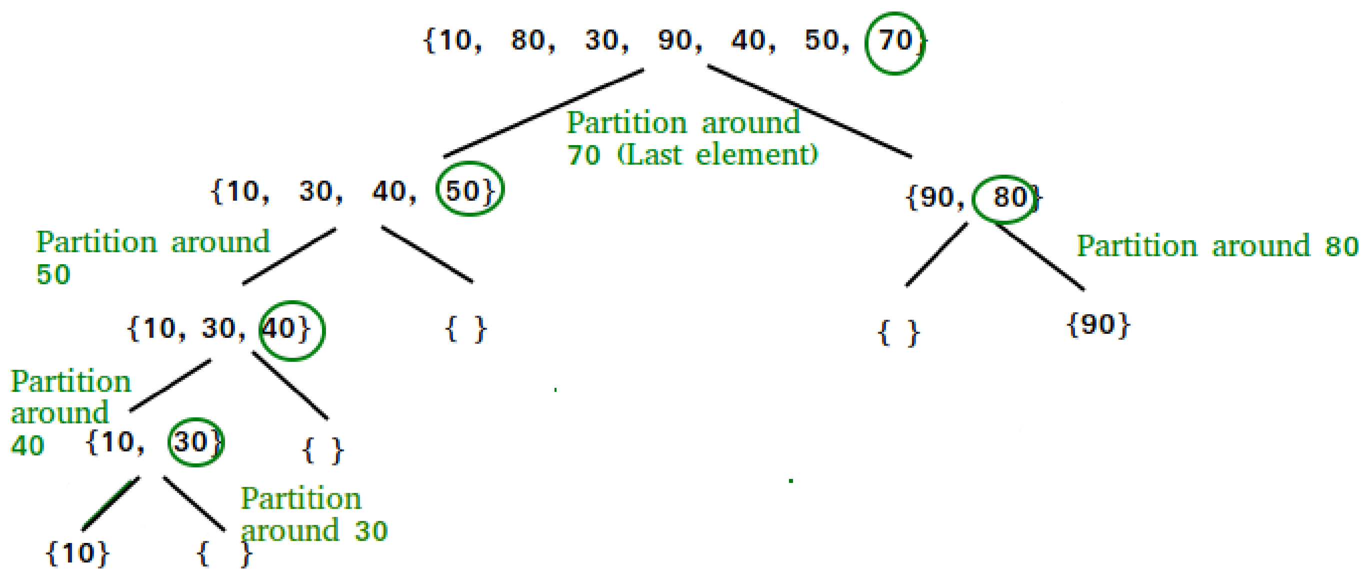
Hence the average and worst case time complexity of merge sort is $\Theta(n \log_2 n)$.

3. Quick sort

Like Merge Sort, QuickSort is a Divide and Conquer algorithm. It picks an element as a pivot and partitions the given array around the picked pivot. There are many different versions of quickSort that pick pivot in different ways.

- Always pick the first element as a pivot.
- Always pick the last element as a pivot (implemented below)
- Pick a random element as a pivot.
- Pick median as the pivot.

The key process in quickSort is a partition(). The target of partitions is, given an array and an element x of an array as the pivot, put x at its correct position in a sorted array and put all smaller elements (smaller than x) before x, and put all greater elements (greater than x) after x. All this should be done in linear time.



Partition Algorithm:

There can be many ways to do partition, following pseudo-code adopts the method given in the CLRS book. The logic is simple, we start from the leftmost element and keep track of the index of smaller (or equal to) elements as i. While traversing, if we find a smaller element, we swap the current element with arr[i]. Otherwise, we ignore the current element.

Pseudo Code for recursive QuickSort function:

```
/* low  –> Starting index,  high  –> Ending index */
quickSort(arr[], low, high) {
   if (low < high) {
      /* pi is partitioning index, arr[pi] is now at right place */
      pi = partition(arr, low, high);
      quickSort(arr, low, pi – 1);  // Before pi
      quickSort(arr, pi + 1, high); // After pi
   }
}
```

Pseudo code for partition()

```
/* This function takes last element as pivot, places the pivot element at its correct position in sorted array, and places all smaller (smaller than pivot) to left of pivot and all greater elements to right of pivot */
partition (arr[], low, high)
{
   // pivot (Element to be placed at right position)
pivot = arr[high];
```

```
 i = (low – 1)  // Index of smaller element and indicates the
// right position of pivot found so far

for (j = low; j <= high- 1; j++){

 // If current element is smaller than the pivot
if (arr[j] < pivot){
i++;    // increment index of smaller element
 swap arr[i] and arr[j]
     }
 }

    swap arr[i + 1] and arr[high])
return (i + 1)
}
```

Analysis of quick sort

Worst Case Analysis: It is the case when items are already in sorted form and we try to sort them again. This will takes lots of time and space.

Equation:

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

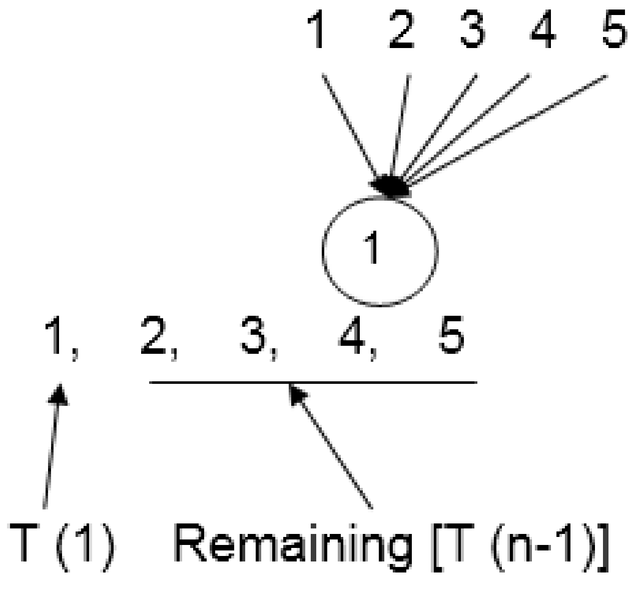
T (1) is time taken by pivot element.

T (n-1) is time taken by remaining element except for pivot element.

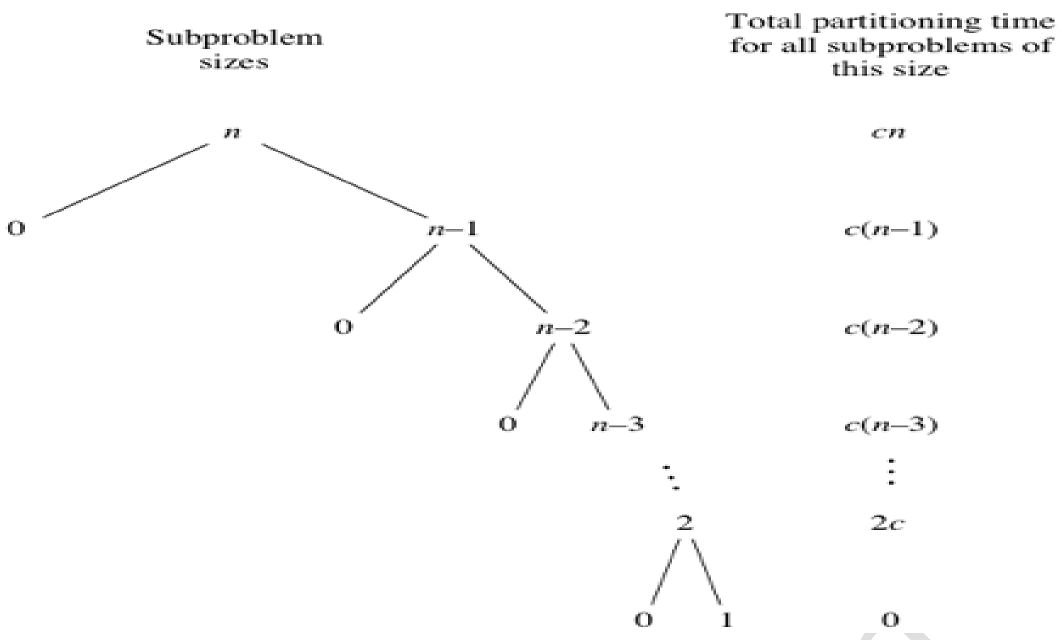
N: the number of comparisons required to identify the exact position of itself (every element)

If we compare first element pivot with other, then there will be 5 comparisons.

It means there will be n comparisons if there are n items.



Here's a tree of the subproblem sizes with their partitioning times:

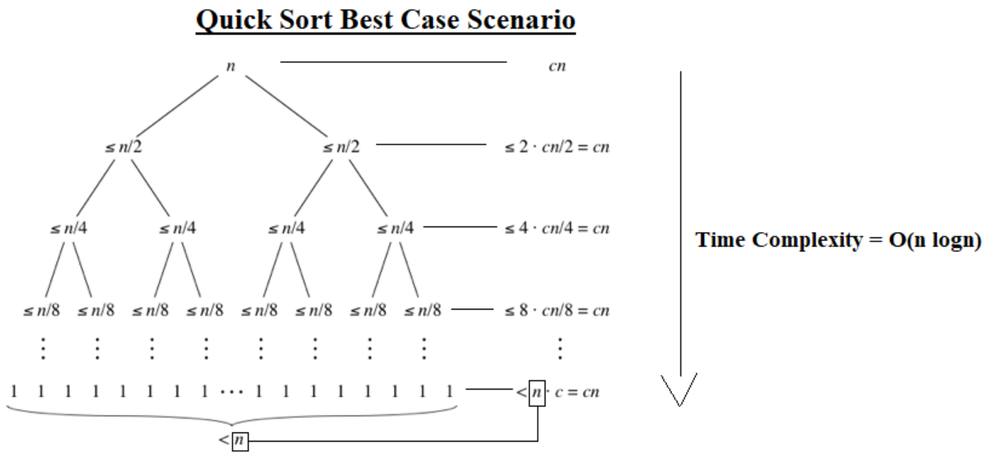


Best Case Complexity - In Quicksort, the best-case occurs when the pivot element is the middle element or near to the middle element. The best-case time complexity of quicksort is O(n*log n).

Average Case Complexity - It occurs when the array elements are in jumbled order that is not properly ascending and not properly descending. The average case time complexity of quicksort is O(n*log n).

- **Best case scenario:** The best case scenario occurs when the partitions are as evenly balanced as possible, i.e their sizes on either side of the pivot element are either are equal or are have size difference of 1 of each other.
    - Case 1: The case when sizes of sublist on either side of pivot becomes equal occurs when the subarray has an odd number of elements and the pivot is right in the middle after partitioning. Each partition will have (n-1)/2 elements.
    - Case 2: The size difference of 1 between the two sublists on either side of pivot happens if the subarray has an even number, n, of elements. One partition will have n/2 elements with the other having (n/2)-1.

In either of these cases, each partition will have at most n/2 elements, and the tree representation of the subproblem sizes will be as below:



The best-case complexity of the quick sort algorithm is O(n logn)

4. Strassen's matrix multiplication

Why Strassen's matrix algorithm is better than normal matrix multiplication and How to multiply two matrices using Strassen's matrix multiplication algorithm?

So the main idea is to use the divide and conquer technique in this algorithm – **divide matrix A & matrix B into 8 submatrices and then recursively compute the submatrices of C.**

$$\underset{A}{\begin{bmatrix} a & b \\ c & d \end{bmatrix}} \times \underset{B}{\begin{bmatrix} e & f \\ g & h \end{bmatrix}} = \underset{C}{\begin{bmatrix} ae + bg & af + bh \\ ce + dg & cf + dh \end{bmatrix}}$$

A, B and C are square metrices of size N x N
a, b, c and d are submatrices of A, of size N/2 x N/2
e, f, g and h are submatrices of B, of size N/2 x N/2

The above strategy is the basic $O(N^3)$ strategy

Using the Master Theorem with **T(n) = 8T(n/2) + O(n^2)** we still get a runtime of **O(n^3)**.

But Strassen came up with a solution where we don't need 8 recursive calls but can be done in only 7 calls and some extra addition and subtraction operations.

Strassen's 7 calls are as follows:

Simple Divide and Conquer also leads to O(N^3), can there be a better way?

In the above divide and conquer method, the main component for high time complexity is 8 recursive calls.

The idea of **Strassen's method** is to reduce the number of recursive calls to 7. Strassen's method is similar to above simple divide and conquer method in the sense that this method also divide matrices to sub-matrices of size N/2 x N/2 as shown in the above diagram, but in Strassen's method, the four sub-matrices of result are calculated using following formulae.

p1 = a(f - h) **p2 = (a + b)h**
p3 = (c + d)e **p4 = d(g - e)**
p5 = (a + d)(e + h) **p6 = (b - d)(g + h)**
p7 = (a - c)(e + f)

The A x B can be calculated using above seven multiplications.
Following are values of four sub-matrices of result C

$$\underset{A}{\begin{bmatrix} a & b \\ c & d \end{bmatrix}} \times \underset{B}{\begin{bmatrix} e & f \\ g & h \end{bmatrix}} = \underset{C}{\begin{bmatrix} p5 + p4 - p2 + p6 & p1 + p2 \\ p3 + p4 & p1 + p5 - p3 - p7 \end{bmatrix}}$$

A, B and C are square metrices of size N x N
a, b, c and d are submatrices of A, of size N/2 x N/2
e, f, g and h are submatrices of B, of size N/2 x N/2
p1, p2, p3, p4, p5, p6 and p7 are submatrices of size N/2 x N/2

If $A = \begin{bmatrix} 5 & 3 & 0 & 2 \\ 4 & 3 & 2 & 6 \\ 7 & 8 & 1 & 4 \\ 9 & 4 & 6 & 7 \end{bmatrix}$, $B = \begin{bmatrix} 3 & 2 & 4 & 7 \\ 2 & 5 & 2 & 9 \\ 3 & 9 & 0 & 3 \\ 7 & 6 & 2 & 1 \end{bmatrix}$

Implement Strassen's matrix multiplication on A and B.

Solution : The given matrix is of order 4 × 4. Hence we will subdivide it in 2 × 2 submatrices. Hence we will compute S_1, S_2, S_3, S_4, S_5, S_6 and S_7.

Let,

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

$$A = \begin{bmatrix} \begin{bmatrix} 5 & 3 \\ 4 & 3 \end{bmatrix} & \begin{bmatrix} 0 & 2 \\ 2 & 6 \end{bmatrix} \\ \begin{bmatrix} 7 & 8 \\ 9 & 4 \end{bmatrix} & \begin{bmatrix} 1 & 4 \\ 6 & 7 \end{bmatrix} \end{bmatrix} \qquad B = \begin{bmatrix} \begin{bmatrix} 3 & 2 \\ 2 & 5 \end{bmatrix} & \begin{bmatrix} 4 & 7 \\ 2 & 9 \end{bmatrix} \\ \begin{bmatrix} 3 & 9 \\ 7 & 6 \end{bmatrix} & \begin{bmatrix} 0 & 3 \\ 2 & 1 \end{bmatrix} \end{bmatrix}$$

Now, $S_1 = (A_{11} + A_{22})(B_{11} + B_{22})$

$$= \left[\begin{bmatrix} 5 & 3 \\ 4 & 3 \end{bmatrix} + \begin{bmatrix} 1 & 4 \\ 6 & 7 \end{bmatrix}\right]\left[\begin{bmatrix} 3 & 2 \\ 2 & 5 \end{bmatrix} \begin{bmatrix} 0 & 3 \\ 2 & 1 \end{bmatrix}\right] = \begin{bmatrix} 6 & 7 \\ 10 & 10 \end{bmatrix}\begin{bmatrix} 3 & 5 \\ 4 & 6 \end{bmatrix}$$

$$= \begin{bmatrix} 18+28 & 30+42 \\ 30+40 & 50+60 \end{bmatrix} = \begin{bmatrix} 46 & 72 \\ 70 & 110 \end{bmatrix}$$

$S_2 = (A_{21} + A_{22})\, B_{11}$

$$= \left[\begin{bmatrix} 7 & 8 \\ 9 & 4 \end{bmatrix} + \begin{bmatrix} 1 & 4 \\ 6 & 7 \end{bmatrix}\right]\begin{bmatrix} 3 & 2 \\ 2 & 5 \end{bmatrix} = \begin{bmatrix} 8 & 12 \\ 15 & 11 \end{bmatrix}\begin{bmatrix} 3 & 2 \\ 2 & 5 \end{bmatrix}$$

$$= \begin{bmatrix} 24+24 & 16+60 \\ 45+22 & 30+55 \end{bmatrix}$$

$$S_2 = \begin{bmatrix} 48 & 76 \\ 67 & 85 \end{bmatrix}$$

$S_3 = A_{11}(B_{12} - B_{22})$

$$= \begin{bmatrix} 5 & 3 \\ 4 & 3 \end{bmatrix} \times \begin{bmatrix} 4 & 7 \\ 2 & 9 \end{bmatrix} - \begin{bmatrix} 0 & 3 \\ 2 & 1 \end{bmatrix} = \begin{bmatrix} 5 & 3 \\ 4 & 3 \end{bmatrix} \times \begin{bmatrix} 4 & 4 \\ 0 & 8 \end{bmatrix}$$

$$= \begin{bmatrix} 20+0 & 20+24 \\ 16+0 & 16+24 \end{bmatrix}$$

$$S_3 = \begin{bmatrix} 20 & 44 \\ 16 & 40 \end{bmatrix}$$

$$S_4 = A_{22} \times (B_{21} - B_{11})$$

$$= \begin{bmatrix} 1 & 4 \\ 6 & 7 \end{bmatrix} \times \begin{bmatrix} 3 & 9 \\ 7 & 6 \end{bmatrix} - \begin{bmatrix} 3 & 2 \\ 2 & 5 \end{bmatrix} = \begin{bmatrix} 1 & 4 \\ 6 & 7 \end{bmatrix} \times \begin{bmatrix} 0 & 7 \\ 5 & 1 \end{bmatrix}$$

$$= \begin{bmatrix} 0+20 & 7+4 \\ 0+35 & 42+7 \end{bmatrix}$$

$$S_4 = \begin{bmatrix} 20 & 11 \\ 35 & 49 \end{bmatrix}$$

$$S_5 = (A_{11} + A_{12}) \times B_{22}$$

$$= \begin{bmatrix} 5 & 3 \\ 4 & 3 \end{bmatrix} + \begin{bmatrix} 0 & 2 \\ 2 & 6 \end{bmatrix} \times \begin{bmatrix} 0 & 3 \\ 2 & 1 \end{bmatrix} = \begin{bmatrix} 5 & 5 \\ 6 & 9 \end{bmatrix} \times \begin{bmatrix} 0 & 3 \\ 2 & 1 \end{bmatrix}$$

$$= \begin{bmatrix} 0+10 & 15+5 \\ 0+18 & 18+9 \end{bmatrix} = \begin{bmatrix} 10 & 20 \\ 18 & 27 \end{bmatrix}$$

$$S_6 = (A_{21} - A_{11}) \times (B_{11} + B_{12})$$

$$= \begin{bmatrix} 7 & 8 \\ 9 & 4 \end{bmatrix} - \begin{bmatrix} 5 & 3 \\ 4 & 3 \end{bmatrix} \times \begin{bmatrix} 3 & 2 \\ 2 & 5 \end{bmatrix} + \begin{bmatrix} 4 & 7 \\ 2 & 9 \end{bmatrix}$$

$$= \begin{bmatrix} 2 & 5 \\ 5 & 1 \end{bmatrix} \times \begin{bmatrix} 7 & 9 \\ 4 & 14 \end{bmatrix} = \begin{bmatrix} 14+20 & 18+70 \\ 35+4 & 45+14 \end{bmatrix}$$

$$S_6 = \begin{bmatrix} 34 & 88 \\ 39 & 49 \end{bmatrix}$$

$$S_7 = (A_{12} - A_{22}) \times (B_{21} + B_{22})$$

$$= \begin{bmatrix} 0 & 2 \\ 2 & 6 \end{bmatrix} - \begin{bmatrix} 1 & 4 \\ 6 & 7 \end{bmatrix} \times \begin{bmatrix} 3 & 9 \\ 7 & 6 \end{bmatrix} + \begin{bmatrix} 0 & 3 \\ 2 & 1 \end{bmatrix}$$

$$= \begin{bmatrix} -1 & -2 \\ -4 & -1 \end{bmatrix} \times \begin{bmatrix} 3 & 12 \\ 9 & 7 \end{bmatrix}$$

$$S_7 = \begin{bmatrix} -21 & -26 \\ -21 & -55 \end{bmatrix}$$

$$C_{11} = S_1 + S_4 - S_5 + S_7$$

$$= \begin{bmatrix} 46 & 72 \\ 70 & 110 \end{bmatrix} + \begin{bmatrix} 20 & 11 \\ 35 & 49 \end{bmatrix} - \begin{bmatrix} 10 & 20 \\ 18 & 27 \end{bmatrix} + \begin{bmatrix} -21 & -26 \\ -21 & -55 \end{bmatrix}$$

$$C_{11} = \begin{bmatrix} 35 & 37 \\ 66 & 77 \end{bmatrix}$$

$$C_{12} = S_2 + S_4$$

$$= \begin{bmatrix} 20 & 44 \\ 16 & 40 \end{bmatrix} + \begin{bmatrix} 10 & 20 \\ 18 & 27 \end{bmatrix}$$

$$C_{12} = \begin{bmatrix} 30 & 64 \\ 34 & 67 \end{bmatrix}$$

$$C_{21} = S_2 + S_4$$

$$= \begin{bmatrix} 48 & 76 \\ 67 & 85 \end{bmatrix} + \begin{bmatrix} 20 & 11 \\ 35 & 49 \end{bmatrix} = \begin{bmatrix} 68 & 87 \\ 102 & 134 \end{bmatrix}$$

$$C_{22} = S_1 + S_3 - S_2 + S_6$$

$$\begin{bmatrix} 46 & 72 \\ 70 & 110 \end{bmatrix} + \begin{bmatrix} 20 & 44 \\ 16 & 40 \end{bmatrix} - \begin{bmatrix} 48 & 76 \\ 67 & 85 \end{bmatrix} + \begin{bmatrix} 34 & 88 \\ 39 & 49 \end{bmatrix}$$

$$= \begin{bmatrix} 66 & 116 \\ 86 & 150 \end{bmatrix} - \begin{bmatrix} 48 & 76 \\ 67 & 85 \end{bmatrix} - \begin{bmatrix} 34 & 88 \\ 39 & 49 \end{bmatrix}$$

$$= \begin{bmatrix} 52 & 128 \\ 58 & 124 \end{bmatrix}$$

Thus the final product matrix C will be -

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

$$= \begin{bmatrix} \begin{bmatrix} 35 & 37 \\ 66 & 77 \end{bmatrix} & \begin{bmatrix} 30 & 64 \\ 34 & 67 \end{bmatrix} \\ \begin{bmatrix} 68 & 87 \\ 102 & 134 \end{bmatrix} & \begin{bmatrix} 52 & 128 \\ 58 & 124 \end{bmatrix} \end{bmatrix} = \begin{bmatrix} \mathbf{35} & \mathbf{37} & \mathbf{30} & \mathbf{64} \\ \mathbf{66} & \mathbf{77} & \mathbf{34} & \mathbf{67} \\ \mathbf{68} & \mathbf{87} & \mathbf{52} & \mathbf{128} \\ \mathbf{102} & \mathbf{134} & \mathbf{58} & \mathbf{124} \end{bmatrix}$$

```
Algorithm St_Mul(int *A, int *B, int *C, int n)
{
 if (n == 1)then
 {
     (*C) =(*C)+ (*A) * (*B);
 }
 else
 {
     St_Mul(A,B,C,n/4);
     St_Mul(A, B+(n/4), C+(n/4), n/4);
     St_Mul(A+2 *(n/4), B, C+2 *(n/4), n/4);
     St_Mul(A+2 *(n/4), B+(n/4), C+3 *(n/4), n/4);
     St_Mul(A+(n/4), B+2*(n/4), C, n/4);
     St_Mul(A+(n/4), B+3*(n/4), C+(n/4), n/4);
     St_Mul(A+3*(n/4), B+2*(n/4), C+2*(n/4), n/4);
     St_Mul(A+3*(n/4), B+3*(n/4), C+3*(n/4), n/4);
 }
}
```

3.9.2 Analysis of Algorithm

$$T(1) = 1 \qquad \text{assume } n = 2^k$$

$$T(n) = 7\,T(n/2)$$

$$T(n) = 7^k\,T(n/2^k)$$

$$T(n) = 7^{\log n}$$

$$T(n) = n^{\log 7} = n^{2.81}$$

Thus divide and conquer is an algorithmic strategy having with the principle idea of dividing the problem into subproblems. Then solution to these subproblems is obtained in order to get the final solution for the given problem.

Generally, Strassen's Method is not preferred for practical applications for following reasons.

1. The constants used in Strassen's method are high and for a typical application Naive method works better.
2. For Sparse matrices, there are better methods especially designed for them.
3. The submatrices in recursion take extra space.
4. Because of the limited precision of computer arithmetic on noninteger values, larger errors accumulate in Strassen's algorithm than in Naive Method

UNIT-II Questions

Graphs

1. Explain how graphs are represented in memory.

2. Explain various graph travel techniques

3. Explain connected and bi-connected graphs with examples.

Divide and conquer

1. Write and explain the control abstraction for divide and conquer.

2. Explain the divide and conquer method with an example.

3. Write an algorithm to sort n numbers in ascending order using merge sort, and compute its tine complexity.

4. Trace the Quick sort algorithm to sort the list C, O, L, L, E, G, E in alphabetical order. Give an instance where the quicksort algorithm has worst case complexity.

5. Explain the Strassen's matrix multiplication. Use Strassen's matrix multiplication algorithm to multiply the following two matrices

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

2 Marks Questions

Graphs – Terminology, Representations, and Traversals

1. Define a graph and differentiate between a directed and undirected graph.
2. What is the degree of a vertex? How is it calculated in a directed graph?
3. List two differences between an adjacency matrix and an adjacency list.
4. Define a connected component in an undirected graph.
5. What is a biconnected component?
6. Write two real-life applications of graphs.
7. What is depth-first search (DFS)? Mention one application.
8. Mention the time complexity of BFS and DFS in terms of vertices (V) and edges (E).
9. Explain the term "path" in graph theory.
10. What is a spanning tree?

Divide and Conquer – General Method and Algorithms

11. What is the basic idea of the Divide and Conquer strategy?
12. State the recurrence relation for Merge Sort.
13. What is the worst-case time complexity of Quick Sort?
14. Write the base condition in Strassen's matrix multiplication method.
15. How many recursive calls are made in Strassen's matrix multiplication?

16. Write one advantage of Merge Sort over Quick Sort.
17. List the steps involved in the Divide and Conquer approach.
18. What is the best-case time complexity of Quick Sort?
19. Name any two problems that can be solved using the Divide and Conquer technique.
20. Define pivot in the context of Quick Sort.