Types of graph:
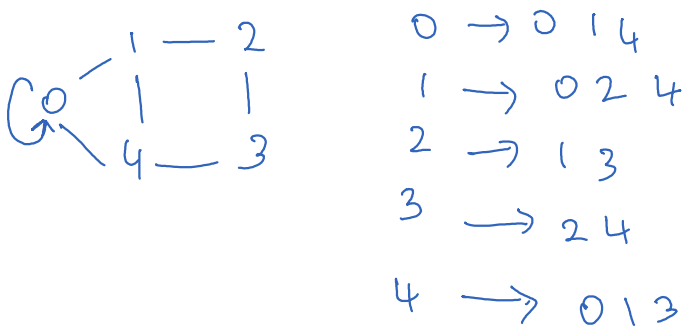
1) Null: No vertex

2) Trivial: single vertex

3) Complete: direct path b/w any two nodes

4) Regular : All nodes have same degree

5) Bipartite: divided into two graphs with vertices set v_1 & v_2 st $v_1 \cap v_2 = \emptyset$
   and there are no direct paths b/w vertices within v_1 & v_2

6) Weighted & Unweighted

7) Directed & Undirected

8) Cyclic : At least one cycle in it

9) Connected & Disconnected

Representation.

1) Adjancy List.

```
    1 — 2          0 → 0 1 4
  0 | |            1 → 0 2 4
    4 — 3          2 → 1 3
                   3 → 2 4
                   4 → 0 1 3
```

2) Matrix representation.

	0	1	2	3	4
0	1	1	0	0	1
1	1	0	1	0	1
2	0	1	0	1	0
3	0	0	1	0	1
4	1	1	0	1	0

✦ replace by weights if given

TRAVERSALS

BFS (Breadth First Search)

→ traverses at the same height first, then goes to next depth.

→ uses Queue & visited array

→ used for getting shortest distance from the starting node to any node.

→ $O(V+E)$ V = vertices   E = edges

colours :   white → not visited π → parent

gray → visited   d → dist from source

black → no more use.

BFS (G, S)

for each vertex u in G - {s}

u. colour = white

u. dist = ∞

u. π = nill

s. colour = gray

s. d = 0

s. π = nill

enqueue (Q,S)

while Q != empty

u = dequeue (Q)

for each v in G.adj [u]

if v. colour = white

$$v.\ Colour = gray$$
$$v.\ d = u.d+1$$
$$v.\pi = u$$
$$enqueue\ (Q,v)$$

$$u.\ Colour = black.$$

DFS (Depth First Search)

→ uses stack & visited array

→ first searches the entire depth of a node, then moves on to another node at same depth.

→ explore neighbours of neighbours before sibling

→ maynot give shortest path.

→ $O\ (V+E)$

```
DFS(G)                          // Main program
{   for each u in V             // Initialize
    {   color[u]=W; pred[u]=NIL; }
    time=0;
    for each u in V
        if(color[u]==W) DFSVisit(u); // Start new tree
}


DFSVisit(u)                     // Process vertex u
{   color[u]=G;                 // Vertex discovered
    d[u]=++time;                // Time of discovery
    for each v in adj[u]
        if(color[v]==W)         // Visit undiscovered
        {   pred[v]=u;  DFSVisit(v);} //   neighbours
    color[u]=B;                 // Vertex finished
    f[u]=++time;                // Time of finish
}
```

BFS   DFS

Web crawlers

topological sort

Shortest dist

GPS navigation

Detecting cycles in graph

finding a path

checking if graph is bipartite

back tracking

topological sort

if branches are many. BFS stuck.

if vertexes are may. DFS stuck

BFS gives a guarenteed sol^n. DFS doesn't.

Spanning Tree.

A tree which contains all the vertices of the graph but has no cycles in it.

edges = # vertices −1 $E = V-1$

spanning trees are acyclic

connected and contains all vertices

} props

Minimum Spanning tree = path with least weights. Maynot be unique.

KRUSKALS ALGO (greedy approach)

→ Sort all edges in increasing order

→ Pick the lowest cost edge and add it to the MST such that it doesnt create a cycle

Time complexity: Sorting $E \log E$

union find algo for a edge = $\log V$ to detect cycles

for E edges   UFA = $E \log V$

$$E \log V > E \log E$$

$$\therefore \quad O(E \log V)$$

Make-set (u) $\Rightarrow$ makes a set with element u.
find-set (u) $\Rightarrow$ finds a set with u as an element
union (x, y) $\Rightarrow$ union of sets x & y

Disjoint set
DS

```
KRUSKAL(G):
A = ø
For each vertex v ∈ G.V:
    MAKE-SET(v)
For each edge (u, v) ∈ G.E ordered by increasing order by weight(u, v):
    if FIND-SET(u) ≠ FIND-SET(v):
    A = A ∪ {(u, v)}
    UNION(u, v)
return A
```

PRIMS ALGO (greedy approach)

start with any random edge.
see to which all vertices the edge is connected to.
Add the min possible weight which doesn't create a cycle.
Now again repeat. Now select the min wt

Main idea: Always add the edge which will add a new vertice to the MST and has the least weight. At all times the tree is connected but not cyclic.

$$\rightarrow \quad O(E \log V)$$

SEARCHING ALGO

Linear Search.

Linear Search.
  best : $O(1)$
  worst: $O(n)$

Binary Search
  → requires sorted algo
  best: $O(1)$
  worst: $O(\log n)$

Ternary Search

Fibonacci Search

Exponential Search.

SORTING ALGO

Selection Sort.

find min element in array
put it on the first index
decrease size of array by one and repeat

time = $O(n^2)$ best / worst / avg

space = $O(1)$

swaps = $O(n)$

Stable = No

in place = yes.

```
for i = 1   to   i = n - 1
  for j = i+1  to  j = n
        if ( a[i] > a[j] )
            swap ( a[i], a[j] )
```

Bubble Sort.

Time: worst $O(n^2)$

best $O(n)$

Space: $O(1)$

in place: yes

stable: yes

```
for i=1 to i=n
    for j=0 to j=n-i-1
        if ( a[j] > a[j+1] )
            Swap ( a[j], a[j+1] )
```

INSERTION SORT

Time: worst $O(n^2)$

best $O(n)$

Space: $O(1)$

in place: yes

stable: yes

```
for i=2 to n
    Key = A[i]
    j = i-1
    while ( j>0 && A[i] > key )
        a[j+1] = a[j]
        j--
    a[j+1] = key
```

if our input is close to the best case, then we can use this

If our input data is small.

Basically our array is divided into two parts sorted & unsorted.

in each iteration we take an element from unsorted part and put it in

its correct position of Sorted array.

MERGE SORT

→ based on DNC

→ Divide array then combine in Sorted manner

→ merge sort better than heap sort

time: best $O(n\log n)$

avg $O(n\log n)$

worst $O(n\log n)$

space: $O(n)$

→ Divide array then combine in sorted manner

→ merge sort better than heap sort

worst $O(n \log n)$

space: $O(n)$

inplace: No

stable: yes.

Merge (A, start, mid, end)

   len 1 = mid - start - 1

   len 2 = end - mid

   left Arr [len 1]   right Arr [len 2]

   for i = 0 to i = len 1 - 1

      Left Arr [i] = A [start + i]

   for j = 0 to j = len 2 - 1

      right Arr [j] = A [mid + 1 + j]

   i = 0 j = 0   k = start

   while i < len1 && j < len 2

      if left Arr [i] <= right Arr [j]

         A [k] = left Arr [i]

         i++

      else

         A [k] = right Arr [j]

         j++

      k++

   while (i < len 1)

      A [k] = left Arr [i]

```
        i++
        k++
    while (j < len2)
        A[k] = rightArr[j]
        j++
        k++
```

Merge Sort (A, start, end)

```
    if start < end
        mid = start + (end - start)
                      ─────────────
                            2

        Merge Sort (A, start, mid)
        Merge Sort (A, mid+1, end)
        Merge        (A, start, mid, end)
```

QUICK SORT

Time: $O(n \log n)$ best

$O(n^2)$ worst

in-place = depending on version

Stable = No

Space: $O(1)$ for non stable

$O(n)$ for stable

pivot → first element
        last element
        median element

* An element (pivot) is put to its correct place in a single pass.

Partion (A, p, r)

i = p-1   j = p   pivot = A[r]

Quick Sort (A, p, r)

if (p < r)

```
i = p-1    j = p    pivot = A[r]
while (j < r)
    if ( A[j] < pivot)
        i++
        swap ( A[i], A[j])
    j++
i++
swap ( A[i], pivot)
return i
```

```
if (p < r)
    return
i = Partition ( A, p, r)
Quick Sort (A, p, i-1)
Quick Sort (A, i+1, r)

p = Start
r = end
```

General Stuff

→ comparison based algo have lower bound of $O(n \log n)$

because no. of permutations of input = n!

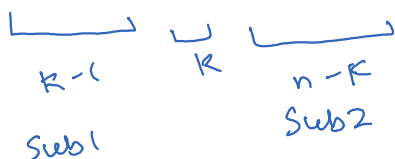$$2^h > n!$$ $$n! = c n^n$$ Sterling approx

$$\therefore h \geqslant n \log n$$

h = height of BT . h = no. of comparisons to be made.

Time Complexity of Quick Sort.

K = pivots correct position.

```
|____|  |_____|
 K-1   K   n-K
Sub1      Sub2
```

$$T(n) = T(n-k) + N$$ → for splitting the array
$\left(\begin{array}{c} \text{traversing in} \\ \text{partition fun}^c \end{array} \right)$

How to check whether a graph is bipartite using DFS

→ Assign a colour to a node

0 = not visited
1 = colour 1
2 = colour 2

→ Assign a colour to ...

1 = colour 1
2 = colour 2

→ if node colour = 0 ; give correct colour
→ if node colour ≠ 0 ; its parent & its own colour need to be diff
  else its not bipartite.

SORTING Rev

	Time			Space	in place	stable
	Best	Avg	Worst			
selection	n^2	n^2	n^2	1	Yes	No
bubble	n	n^2	n^2	1	Yes	Yes
insertion	n	n^2	n^2	1	yes	yes
Quick	$n\log n$	$n\log n$	n^2	1	Yes	No
Merge	$n\log n$	$n\log n$	n^2	n	No	Yes
Heap	$n\log n$	$n\log n$	$n\log n$	1	Yes	No

Time complexity of Merge Sort.

$$T(N) = 2 \, T\left(\frac{N}{2}\right) + O(N)$$

↳ for traversing

$$O(n \log n)$$

for merging   height of tree.

unstable = S Q H   not inplace = Merge.

unstable = S Q H   not inplace = Merge.

selection Quick Heap

Dijkstra Algorithm

```
int  minDist ( int  dist [ ], bool  fixed [ ])

        int  min_dist = INT_MAX
        int  min_index
        for ( int i = 0 ; i < V ; i++)
        {      if ( fixed [i] == false  && dist [i] <= min_dist )
                {        min_dist = dist[i] ;
                         min_index = i
                }
        }

    return  min_index;

void  Dijkstra ( int graph [][] , int source )

    int dist [v];
    bool fixed [v];
    for ( int i = 0 ; i < v ; i++)
    {
        dist [i] = INT_MAX;
        fixed [i] = false;
    }
    dist [source] = 0;
```

```
for ( int count = 0 ; count < V-1 ; count ++)
{
        int u = Min Dist ( dist, fined);

        fined[u] = true;
        for ( int i=0 ; i<V ; i++)
        {
            if  / graph[u][i] != 0 &&  fined[i] == false
               &&  dist[u] != INT_MAX  &&

                dist[u] + graph[u][i] < dist[i] /

                dist[i] =  dist[u] + graph[u][i];
        }
}
```

```cpp
// Returns the value of maximum profit
int knapSackRec(int W, int wt[], int val[], int index, int** dp)
{
    // base condition
    if (index < 0)
        return 0;
    if (dp[index][W] != -1)
        return dp[index][W];

    if (wt[index] > W) {

        // Store the value of function call
        // stack in table before return
        dp[index][W] = knapSackRec(W, wt, val, index - 1, dp);
        return dp[index][W];
    }
    else {
        // Store value in a table before return
        dp[index][W] = max(val[index]
                        + knapSackRec(W - wt[index], wt, val,
                                    index - 1, dp),
                        knapSackRec(W, wt, val, index - 1, dp));

        // Return value of table after storing
        return dp[index][W];
    }
}
```

```cpp
// Returns length of LCS for X[0..m-1],
// Y[0..n-1]
int lcs(char* X, char* Y, int m, int n,
            vector<vector<int> >& dp)
{
    if (m == 0 || n == 0)
        return 0;
    if (X[m - 1] == Y[n - 1])
        return dp[m][n] = 1 + lcs(X, Y, m - 1, n - 1, dp);

    if (dp[m][n] != -1) {
        return dp[m][n];
    }
    return dp[m][n] = max(lcs(X, Y, m, n - 1, dp),
                        lcs(X, Y, m - 1, n, dp));
}
```

```cpp
bool Graph::isCyclicUtil(int v, bool visited[],
                         bool* recStack)
{
    if (visited[v] == false) {
        // Mark the current node as visited
        // and part of recursion stack
        visited[v] = true;
        recStack[v] = true;

        // Recur for all the vertices adjacent to this
        // vertex
        list<int>::iterator i;
        for (i = adj[v].begin(); i != adj[v].end(); ++i) {
            if (!visited[*i]
                && isCyclicUtil(*i, visited, recStack))
                return true;
            else if (recStack[*i])
                return true;
        }
    }

    // Remove the vertex from recursion stack
    recStack[v] = false;
    return false;
}
```

```cpp
bool Graph::isCyclic()
{
    // Mark all the vertices as not visited
    // and not part of recursion stack
    bool* visited = new bool[V];
    bool* recStack = new bool[V];
    for (int i = 0; i < V; i++) {
        visited[i] = false;
        recStack[i] = false;
    }

    // Call the recursive helper function
    // to detect cycle in different DFS trees
    for (int i = 0; i < V; i++)
        if (!visited[i]
            && isCyclicUtil(i, visited, recStack))
            return true;

    return false;
}
```

```cpp
// function to check whether a graph is bipartite or not
bool isBipartite(vector<int> adj[], int v,
                 vector<bool>& visited, vector<int>& color)
{

    for (int u : adj[v]) {

        // if vertex u is not explored before
        if (visited[u] == false) {

            // mark present vertices as visited
            visited[u] = true;

            // mark its color opposite to its parent
            color[u] = !color[v];

            // if the subtree rooted at vertex v is not bipartite
            if (!isBipartite(adj, u, visited, color))
                return false;
        }

        // if two adjacent are colored with same color then
        // the graph is not bipartite
        else if (color[u] == color[v])
            return false;
    }
    return true;
}
```

```cpp
bool solveNQUtil(int board[N][N], int col)
{
    // base case: If all queens are placed
    // then return true
    if (col >= N)
        return true;

    // Consider this column and try placing
    // this queen in all rows one by one
    for (int i = 0; i < N; i++) {

        // Check if the queen can be placed on
        // board[i][col]
        if (isSafe(board, i, col)) {

            // Place this queen in board[i][col]
            board[i][col] = 1;

            // recur to place rest of the queens
            if (solveNQUtil(board, col + 1))
                return true;

            // If placing queen in board[i][col]
            // doesn't lead to a solution, then
            // remove queen from board[i][col]
            board[i][col] = 0; // BACKTRACK
        }
    }

    // If the queen cannot be placed in any row in
    // this column col  then return false
    return false;
}
```

```cpp
int BFS(int mat[][COL], Point src, Point dest)
{
    if (!mat[src.x][src.y] || !mat[dest.x][dest.y])
        return -1;

    bool visited[ROW][COL];
    memset(visited, false, sizeof visited);

    visited[src.x][src.y] = true;

    queue<queueNode> q;
    queueNode s = {src, 0};
    q.push(s);  // Enqueue source cell

    while (!q.empty())
    {
        queueNode curr = q.front();
        Point pt = curr.pt;
        if (pt.x == dest.x && pt.y == dest.y)
            return curr.dist;
        q.pop();

        for (int i = 0; i < 4; i++)
        {
            int row = pt.x + rowNum[i];
            int col = pt.y + colNum[i];
            if (isValid(row, col) && mat[row][col] &&
                !visited[row][col])
            {
                visited[row][col] = true;
                queueNode Adjcell = { {row, col},curr.dist + 1 };
                q.push(Adjcell);
            }
        }
    }
    return -1;
}
```

```cpp
/* problem */
int solveKTUtil(int x, int y, int movei, int sol[N][N],
                int xMove[8], int yMove[8])
{
    int k, next_x, next_y;
    if (movei == N * N)
        return 1;

    /* Try all next moves from
    the current coordinate x, y */
    for (k = 0; k < 8; k++) {
        next_x = x + xMove[k];
        next_y = y + yMove[k];
        if (isSafe(next_x, next_y, sol)) {
            sol[next_x][next_y] = movei;
            if (solveKTUtil(next_x, next_y, movei + 1, sol,
                            xMove, yMove)
                == 1)
                return 1;
            else

                // backtracking
                sol[next_x][next_y] = -1;
        }
    }
    return 0;
}
```

```cpp
// Function to get all valid paths
void findPath(int r, int c, vector<vector<int> >& maze,
              int n, vector<string>& ans,
              string& currentPath)
{

    // if we reach the bottom right cell of the matrix,
    // add the current path to ans and return
    if (r == n - 1 && c == n - 1) {
        ans.push_back(currentPath);
        return;
    }

    // Mark the current cell as blocked
    maze[r][c] = 0;

    for (int i = 0; i < 4; i++) {
        // find the next row based on the current row
        // (r) and the dr[] array
        int nextr = r + dr[i];
        // find the next column based on the current
        // column (c) and the dc[] array
        int nextc = c + dc[i];
        // check if the next cell is valid or not
        if (isValid(nextr, nextc, n, maze)) {
            currentPath += direction[i];
            // Recursively call the solve function for
            // the next cell
            findPath(nextr, nextc, maze, n, ans,
                     currentPath);
            currentPath.pop_back();
        }
    }
    // Mark the current cell as unblocked
    maze[r][c] = 1;
}
```

```cpp
bool solveSudoku(int grid[N][N], int row, int col)
{
    if (row == N - 1 && col == N)
        return true;

    if (col == N) {
        row++;
        col = 0;
    }

    if (grid[row][col] > 0)
```

```cpp
bool solveSudoku(int grid[N][N], int row, int col)
{
    if (row == N - 1 && col == N)
        return true;

    if (col == N) {
        row++;
        col = 0;
    }

    if (grid[row][col] > 0)
        return solveSudoku(grid, row, col + 1);

    for (int num = 1; num <= N; num++)
    {

        if (isSafe(grid, row, col, num))
        {

            grid[row][col] = num;

            if (solveSudoku(grid, row, col + 1))
                return true;
        }

        grid[row][col] = 0;
    }
    return false;
}
```