Depth first

- You are going through every row in the matrix one by one
- marking the row as a visited row
- int the loop
- -> you are visiting every vertex in that particular row if it is already not visited
- you are priting the vertex
-

Adjacency List

```
class GraphAdjList{
  private:
    int vertex_count;
    vector<int> *  adjlist;

  public:

    GraphAdjList(int total_vertex_count){
      vertex_count = total_vertex_count;
      adjlist = new vector<int>[vertex_count];
    }

    void addEdge(int src, int des){
      adjlist[src].push_back(des);
      adjlist[des].push_back(src); // do this if for every (a,b) you dont want to add (b,a)
    }

    void print(){
      for(int i = 0; i < vertex_count; i++){
        for(auto a: adjlist[i]){
          cout << a << " ";
        }
        cout << endl;
      }
    }

    void bfs(int start_vertex){
      queue<int> q;
      vector<bool>visited(vertex_count, false); // in the queue or not

      q.push(start_vertex);
      visited[start_vertex] = true;

      while(!q.empty()){
        int current_vertex = q.front(); q.pop();
```

```
        cout << current_vertex << " ";

        for(auto a: adjlist[current_vertex]){
          if(!visited[a]){
            q.push(a);
            visited[a] = true;
          }
        }

      }
    }


    void dfs(int start_vertex){
      static vector<bool>visited(vertex_count, false);

      if(!visited[start_vertex]){
        cout << start_vertex << " ";
        visited[start_vertex] = true;
      }

      for(auto a: adjlist[start_vertex]){
        if(!visited[a]){
          dfs(a);
        }
      }
    }

};

int main(){

  GraphAdjList g(8);
  g.addEdge(1,6);
  g.addEdge(6,5);
  g.addEdge(5,4);
  g.addEdge(4,3);
  g.addEdge(3,2);
  g.addEdge(2,1);
  g.addEdge(5,7);
  g.addEdge(7,4);
  g.addEdge(7,2);

  g.dfs(3);

  return 0;
}
```

Adjacency Matrix

```
// breadth first search
void bfs(vector<vector<int>> g, int start_vertex){

  // visited to mark visist, queue to hold future visits
  vector<bool> visited(g.size(), false);
  queue<int> q;

  // push first vertex to queue
  q.push(start_vertex);
  visited[start_vertex] = true;

  while(!q.empty()){

    // visit the current vertex
    int currentVertex = q.front(); q.pop();
    cout << currentVertex << " ";

    for(int j = 0; j < g[currentVertex].size(); j++){
      // if current vertex's neighbours not visited push to queue
      if(g[currentVertex][j] == 1 && !visited[j]){
        q.push(j);
        visited[j] = true;
      }
    }
  }

}


// depth first search

void dfs(vector<vector<int>> g, int start_vertex){
  static vector<bool> visited(g.size(), false);

  // if not visited? visit! mark visited!
  if(!visited[start_vertex]){
    cout << start_vertex<< " ";
    visited[start_vertex] = true;
  }

  for(int j = 0; j < g[start_vertex].size(); j++){
    // if current vertex's neighbours not visited visit each of them
    if(g[start_vertex][j] == 1 && !visited[j]){
      dfs(g, j);
    }
  }

}


int main(){
```

```
  vector<vector<int>> g = {
    {0,0,0,0,0,0,0},
    {0,0,1,1,0,0,0},
    {0,1,0,0,1,0,0},
    {0,1,0,0,1,0,0},
    {0,0,1,1,0,1,1},
    {0,0,0,0,1,0,0}
  };

  bfs(g, 2);
  dfs(g, 1);

  return 0;
}
```

2D array / matrix

```
void bfsMatrix(vector<vector<int>> mat){
  vector<vector<bool>> visited(mat.size(), vector<bool>(mat[0].size(), false));

  queue<pair<int,int>> q;

  q.push(make_pair(0,0));

  while(!q.empty()){
    pair<int, int> a = q.front(); q.pop();

    if(a.first < 0 || a.first >= mat.size() || a.second < 0 || a.second >=
mat[a.first].size()){
      continue;
    }

    if(visited[a.first][a.second]) {
      continue;
    }

    cout << a.first << " " << a.second << ": "<< mat[a.first][a.second] << endl;
    visited[a.first][a.second] = true;

    q.push(make_pair(a.first, a.second-1));  // left
    q.push(make_pair(a.first, a.second+1));  // right
    q.push(make_pair(a.first - 1, a.second));  // top
    q.push(make_pair(a.first + 1, a.second));  // bottom

  }

}


void dfsMatrix(vector<vector<int>> mat){
```

```
  vector<vector<bool>> visited(mat.size(), vector<bool>(mat[0].size(), false));

  stack<pair<int,int>> s;

  s.push(make_pair(0,0));

  while(!s.empty()){
    pair<int, int> a = s.top(); s.pop();


    if(a.first < 0 || a.first >= mat.size() || a.second < 0 || a.second >=
mat[a.first].size()){
      continue;
    }

    if(visited[a.first][a.second]) {
      continue;
    }

    cout << a.first << " " << a.second << ": "<< mat[a.first][a.second] << endl;
    visited[a.first][a.second] = true;

    s.push(make_pair(a.first, a.second-1));  // left
    s.push(make_pair(a.first, a.second+1));  // right
    s.push(make_pair(a.first - 1, a.second));  // top
    s.push(make_pair(a.first + 1, a.second));  // bottom

  }

}
```

Binary Search tree

```
// iterative
class Solution {
public:
    TreeNode* searchBST(TreeNode* root, int val) {
        if(root == nullptr){
          return nullptr;
        }

        queue<TreeNode *> q;
        q.push(root);

        while(!q.empty()){
          TreeNode * current = q.front(); q.pop();
          if(current->val == val){
            return current;
          }
          if(current->left != nullptr){
```

```
            q.push(current->left);
          }

          if(current->right != nullptr){
            q.push(current->right);
          }
        }
        return nullptr;
    }
};
```

Union -find

Disjoint-set

- A disjoint–set is a data structure that keeps track of a set of elements partitioned into several disjoint (non-overlapping) subsets
- a disjoint set is a group of sets where no item can be in more than one set.
- It is also called a union–find data structure as it supports union and find operation on subsets.

union-find

- We can determine whether two elements are in the same subset by comparing the result of two *Find* operations
- If the two elements are in the same set, they have the same representation; otherwise, they belong to different sets. If the union is called on two elements, merge the two subsets to which the two elements belong.
-