- Set of vertices and Edges.

Simple —— Graphs

Path → sequences of vertices to reach from 1 node to another.

↓

Simple → if → each vertex is distinct.

Q.) Write a code to print each component.

- Cyclic → if contains a cycle. , else acyclic.
- Connected if you can go from any node

↓

Strongly connected

↳ If from any node to another in one edge

'k' graph (complete).

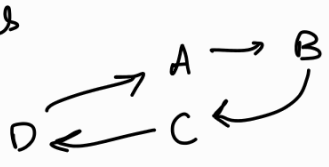
Weighted Graph:

each edge has an associated weight

Directed Graph → Complete → it has $n(n-1)$ edges

↓ ↳ non complete → $\frac{n(n-1)}{2}$ edges

has directions for edges



Connected ACyclic graph is a tree.

## Degree

- Max edges connected to a node.

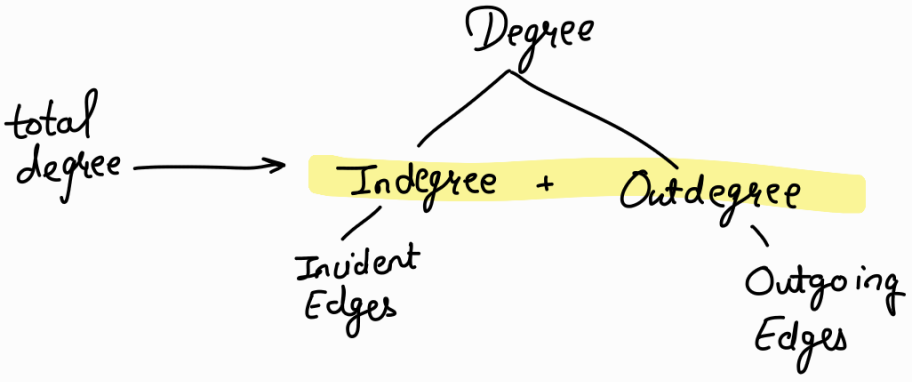
**Q Convert Graph to tree**

keep Removing edges causing a cycle

Code to:

- Convert a Graph to tree
- Convert tree to graph.

**Q. Display degree of Graph** | Degree of all nodes.
(Max degree)




- IF G is Digraph (Directed) with m edges, then.

Digraph → Directed Graph.

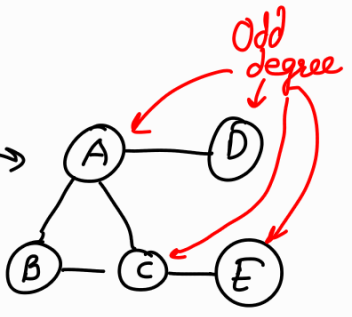
$$\sum \text{indeg}(v) = \sum \text{outdeg}(v) = m = |E|$$

$|E|$ → Edges.

- IF G is a graph with m edges,

$$\sum \deg(v) = 2m = 2|E|$$

- No. of odd degree Nodes is Even




$\sum \deg(G) = 6 = 2m$

$m = \frac{6}{2}$

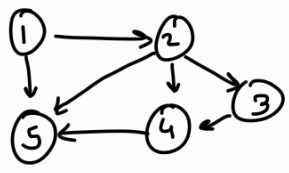
$m = 3$

$\therefore E = 3$

## Representations:

### Adjacency Matrix:

- V x V matrix
- Boolean values (adjacent or not) or Edge weights.



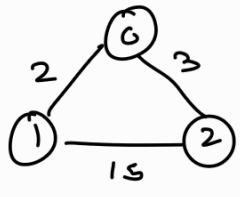

| | 1 | 2 | 3 | 4 | 5 |
|---|---|---|---|---|---|
| 1 | 0 | 1 | 0 | 0 | 1 |
| 2 | 0 | 0 | 1 | 1 | 1 |
| 3 | 0 | 0 | 0 | 1 | 0 |
| 4 | 0 | 0 | 0 | 0 | 1 |
| 5 | 0 | 0 | 0 | 0 | 0 |

Adjacency Matrix (Non-weighted).

Q. Representation → How are you going to store in computer's memory.

### Adjacency Matrix for weighted

Matrix(i,j) will have the weight

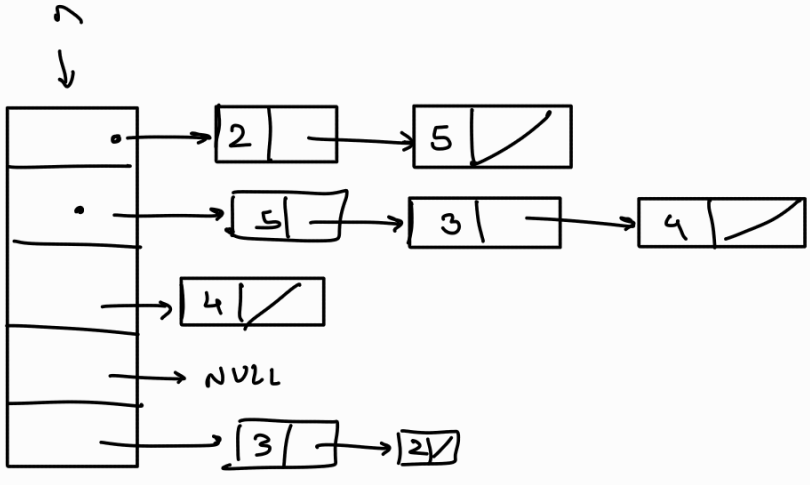



| | 0 | 1 | 2 |
|---|---|---|---|
| 0 | 0 | 2 | 3 |
| 1 | 2 | 0 | 15 |
| 2 | 3 | 15 | 0 |

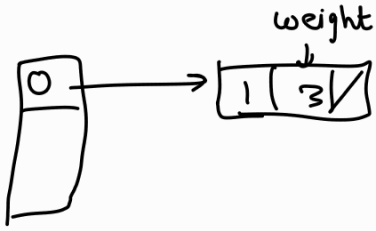
← Replace 0's as INT_MIN.

### Adjacency-list Representation:

n sized array of pointers to linked list.




Undirected




Adv. of Adjacency Matrix

→ Saves space for → dense graph.
- Small unweight graphs using 1 bit for edge.

→ • Check for existence of edge in $O(1)$

Disadvantage:

→ Traverse all edges that start at v, in $O(|V|)$

Adjacency Lists:

More 0's than 1's.

Adv:

- Saves space for sparse graphs. Most graphs are sparse.
- Traverse all edges that start at v, in $O(\text{degree}(v))$

Disadvantage:

- Check of existence of edge in $O(\deg(v))$ worst case.

Storage of Adjacency List

Directed Graph:

$\Sigma(\text{out-degree}(v)) = |E|$

So we need $O(V+E)$ for space.

Vertical List ↘ ↙ List of nodes

For undirected graph number of items are

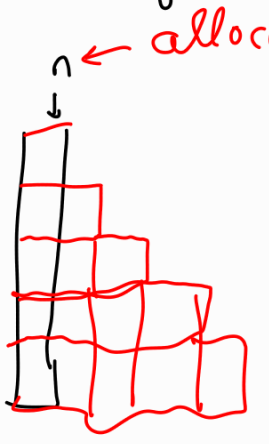
$\Sigma(\text{out degree}(v)) = 2|E|$

we need $O(V+E)$ for space with $c=2$.

Storage for Matrix Representation.

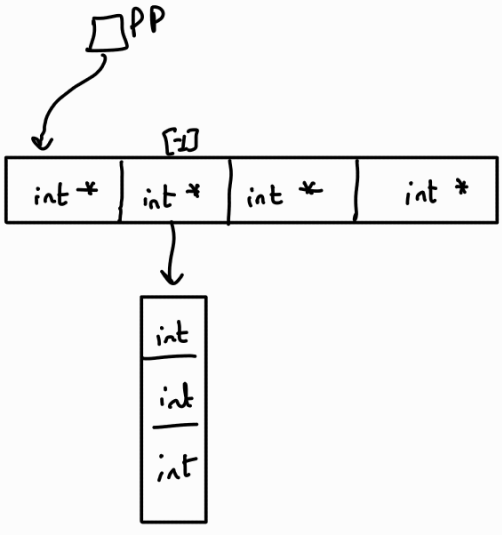
- $O(|V|^2)$ for directed.
- $O(\frac{1}{2} V^2)$ for undirected graph as undirected graphs are symmetric

↳ get array of size n:




## Pointer to Pointer:

```
int **p ,*p ,p;
pp = (int **) malloc(sizeof(int *) *4);
pp[i] (int *) malloc (sizeof(int) *3);
```




```
main() {
    pp = (int**) malloc (size of (int *) * size);

    for (i<n) {
        pp[i] = malloc(size of (int) * size);
```

Representation of:

ADT Graph:

```
typedef struct Graph{
  int **a;
```

Array representation

PS

These are my class notes.
Empty pages indicate I was absent
or highly confused/sleepy.
Read relevant topics from book for the same.

also: [find my codes here]
(github.com/PratyayDhond/DSA-2)

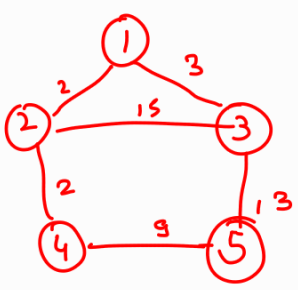
graph.h

```
typedef struct Graph{
    int **A;
    int n ;     // vertices in a graph. //size
}Graph;

void initGraph (Graph *g, char * filename);
void display( Graph g);
```

Graph will always be a square matrix ✗

← Read Data from file.

Test Case 1:




size → 5

| | 1 | 2 | 3 | 4 | 5 |
|---|---|---|---|---|---|
| 1 | 0 | 2 | 3 | 0 | 0 |
| 2 | 2 | 0 | 15 | 2 | 0 |
| 3 | 3 | 15 | 0 | 0 | 13 |
| 4 | 0 | 2 | 0 | 0 | 9 |
| 5 | 0 | 0 | 13 | 9 | 0 |

graph.c

```
#include "graph.h"
-11-
-11-
-11-

void initGraph (Graph *g, char *fileName){
    FILE *f = fopen(filename ,"r");
    if (!f) return;
    fscanf(fp, "%d",&g->n);

    g->A = (int**) malloc (sizeof(int*) * g->n);
    for(int i=0; i<g->n ; i++){
        g->A[i] = (int*) malloc (sizeof (int) * g->n);
        if (!g->A[i]) {
            for(int k = i-1 ;k>=0 ; k--) {
                //free k wala array;
            }
            free g->A ; return .
        }
```

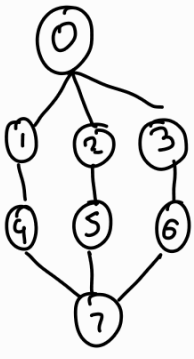
```
        for(j=0; j<g->n; j++){
            fscanf(fp, "%d", g->A[i][j]);
        }
    }
```

Traversals:

1] Breadth First Search

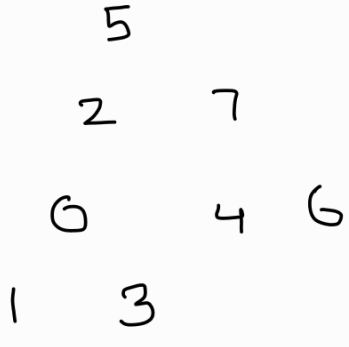
2] Depth First Search

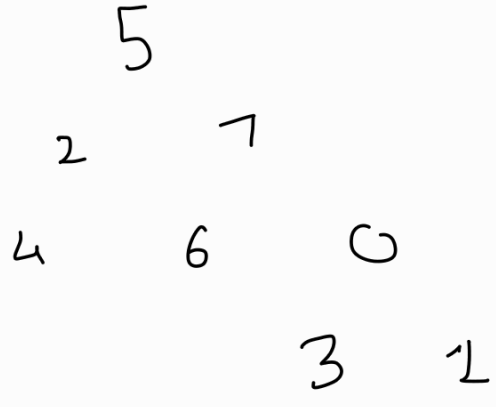
Init array of size V, initialize all to 0.



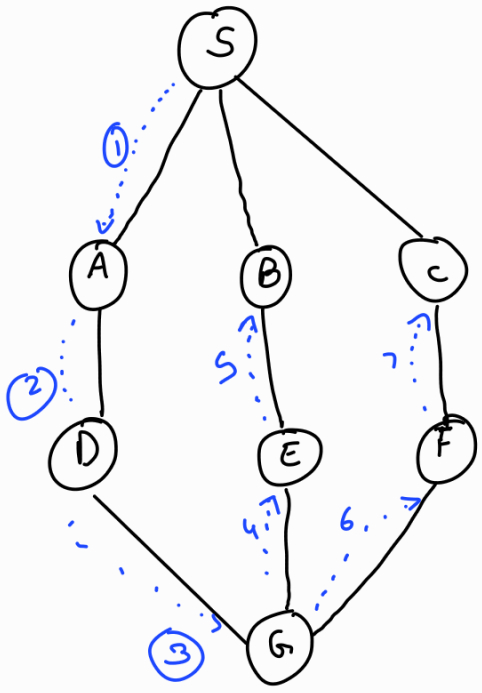

Queue

```
init visited
s -> visited
while (Q not empty) {
    n = dqueue(Q)
    print(n)
    enque all adjacent of not visited.  -> while enquing mark as visited also.
```



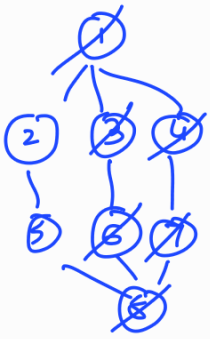



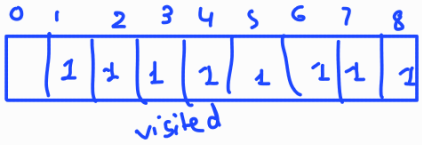
```
void bFS(Graph g, int s){
    int *visited = (int*) calloc (G.n, sizeof(int));
    Queue Q;
     initQ(&Q);
     enqueue(&Q, s);
     visited [s] = 1;
      int v; int n = G.n;
    while ( !isEmptyQ(Q)){

        v = deQueue(&Q);
         printf("%d ",v );
        for(int i=0; i<n ;i++) {
          if(G.A[v][i]  && !visited [i]) {

              enQueue (&Q,i);
              visited[i]=1;
           }
       }
    }
```

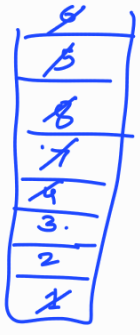



```
dfs(Graph g, int s){

    int *visited = (int*) calloc (G.n, sizeof(int));
    Queue Q; initQueue(&q);
    Stack st; initStack (&st);
    push(&st, s);
```

```
visited[s] = 1;
while (!isEmpty(st)){
    v = pop(&st);
    printf("%d ", v);
    for(int i=0; i<n; i++){
        if (G.A[v][i] != 0 && !visited[i]){
            push(&st, i);
            visited[i] == 1;
        }
    }
}
```





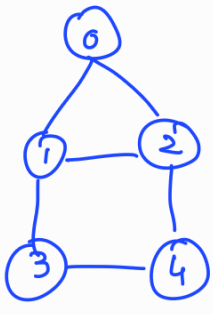

dp: 1 4 7 8 6 3 5 2

o/p: 0 3 6 7 5 2 4 1

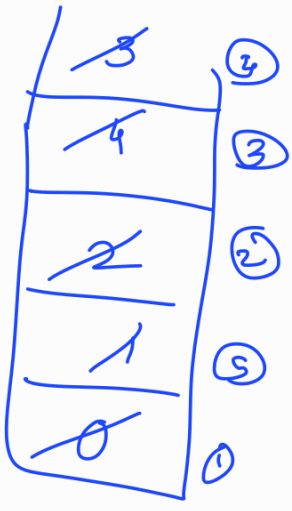
## check if graph is connected.

use counter = 0.

if at end of dfs, counter == n-1, then graph is connected else not connected.

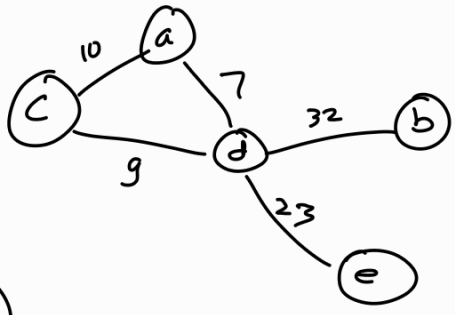


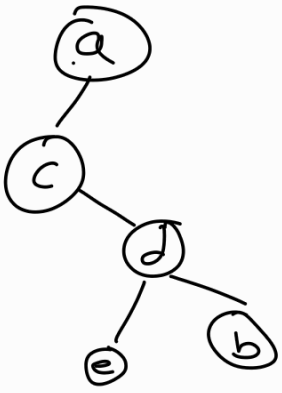

DFS

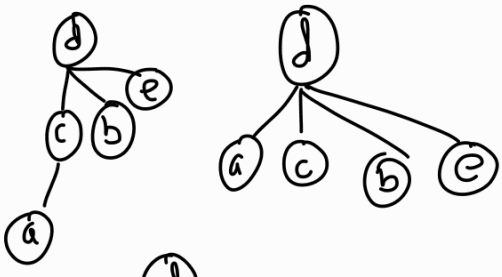


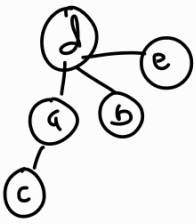

0 2 4 3 1

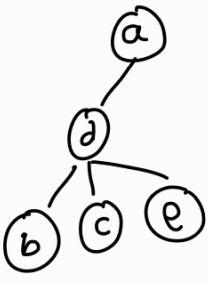
Spanning Tree of a Graph

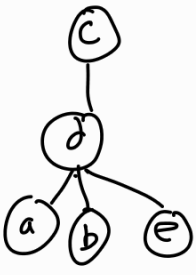


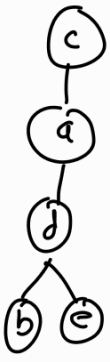



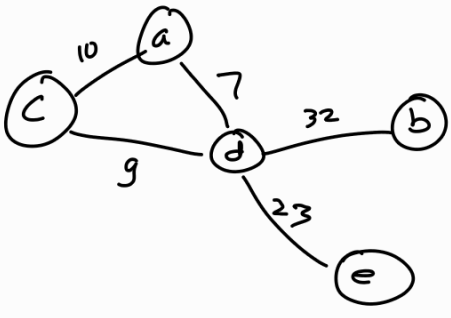










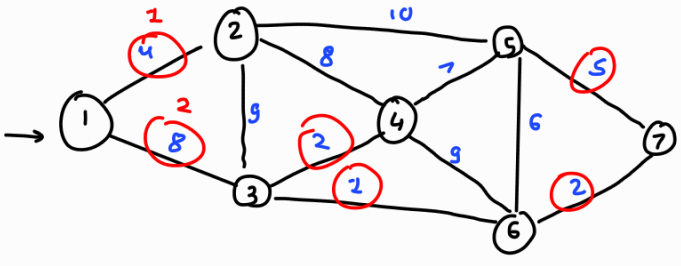



Min Span tree:

an undirected connected weighted graph.

Prim's Algorithm:

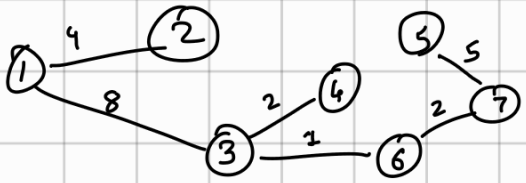
- Start from 1, pick the adjacent edge with minimum weight
- Mark the above as visited and add to tree.




A = {1, 2, 3, 6, 4, 7, 5}

V = {~~1~~, ~~2~~, ~~3~~, ~~4~~, ~~5~~, ~~6~~, ~~7~~}

| 1 | 2 | 3 | 6 | 4 | 7 | 5 |
|---|---|---|---|---|---|---|

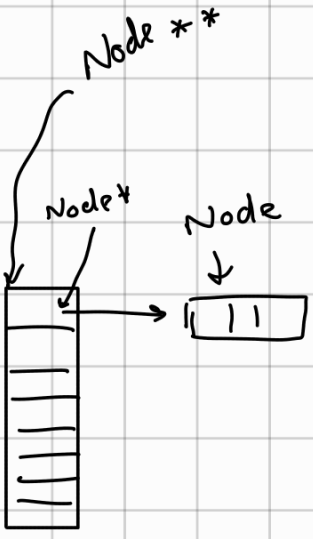



W= 22

Pseudo:

| | 1 | 2 | 3 | 4 | 5 | 6 | 7 |
|---|---|---|---|---|---|---|---|
| 1 | 0 | 4 | 8 | 0 | 0 | 0 | 0 |
| 2 | 4 | 0 | 9 | 8 | 10 | 0 | 0 |
| 3 | 8 | 9 | 0 | | | | |
| 4 | 0 | 8 | | 0 | | | |
| 5 | 0 | 10 | | | 0 | | |
| 6 | 0 | 0 | | | | 0 | |
| 7 | 0 | 0 | | | | | 0 |

```
typedef struct Node {
    int j, w;
    struct Node;
} Node;

typedef Node ** sp_tree;
```




```
Sp_tree Prims (Graph g, int s){
    Sp_tree t = (Node **) malloc (sizeof (Node *) * G.n);
    if(!t)
        return NULL;
    int * visited = (int *) calloc ( G.n , sizeof(int));
    if(!visited) {
        free(t);
        return NULL;
    }

for (int i=0; i< G.n ; i++)
        t [i] = NULL;
```

```
visited [s] = 1 ;
int min = INT_MAX;
prev_p = s;
for (int i=0; i< G.n-1; i++) {
    t[i] = NULL;   min = INT_MAX;   int minV = 0;
    for (int p=0; p < G.n; p++) {
        if (visited [p]) {
            for (int j=0; j< G.n ; j++) {
                if (G.A [p] [j]  && visited [p]  && ! visited [j]) {
                    if (G.A[p] [j] < min) {
                        min = G.A[p] [j];
                        minv = j;  prev_p = p;
                    }
                }
            }
        }
    }
// minv is the min edge

    Node *nn = (Node*) malloc (sizeof (Node));
    if (!nn) return NULL;
    nn->j = minV;
    nn->w = min;
    nn->next = t[p]
    t[p] = nn
    visited [minv] = 1;
}
return t;
}
```

```
void printT( Sp_tree t, int size) {
    if(!t) return;

    for (int i=0; i< size; i++) {
        if ( t[i]){
            Node *temp = t[i];
            while ( temp) {
                printf(" %d -> %d -> %d ", i, temp->j, temp->w);
                temp= temp -> next;
            }
        }
    }

    return;
}
```

| | 0 | 1 | 2 | 3 | 4 |
|---|---|---|---|---|---|
| 0 | 0 | 1 | 1 | 0 | 0 |
| 1 | 1 | 0 | 1 | 1 | 0 |
| 2 | 1 | 1 | 0 | 0 | 1 |
| 3 | 0 | 1 | 0 | 0 | 1 |
| 4 | 0 | 0 | 1 | 1 | 0 |

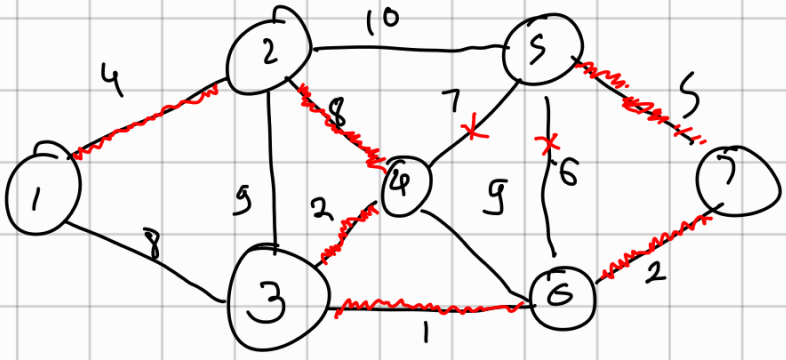
May 29

Greedy Algo's

- work incrementally
- Tries to maximize or minimize the result

Kruskal:

- Pick smallest Edge




| Vertex | : | Set |
|---|---|---|
| 1 | | {1} |
| 2 | | {2} |
| 3 | | {3} |
| 4 | | {4} |
| 5 | | {5} |
| 6 | | {6} |
| 7 | | {7} |

⇒

| Vertex | : | Set |
|---|---|---|
| 1 | | {1} |
| 2 | | {2} |
| 3 | | {3,6} |
| 4 | | {4} |
| 5 | | {5} |
| 7 | | {7} |

⇓

| | | |
|---|---|---|
| 3 | | {3,6,4} |

{1,2} , {3,6,4,7,5}

Vertex - Set Array Mapping.

1 2 3 4 5 6 7

1 2 3 4 5 6 7

1 2 3 4 5 6 7

1 2 3 4 5 3 7 ← (3,6) same set

1 2 3 4 5 6 7

1 2 1 1 1 1 1

till all become part of one set.

push to min heap.

Edge { i, j, weight }

Time (e log e)

every edge.

Implement heap for edge.

// Vertices → Set of Nodes

// Edges → Relation between Nodes.

# Shortest Path Algorithm:

Dijkstra's algorithm:
Solves the problem of non-negative costs.

Bellman - Ford → Applicable for problems with arbitrary cost.
Floyd warshall →

Dijkstra's algorithm:

visited array:

create and initialise cost array to the row

visited array update → update cost → relaxation.
↓
cost matrix.

$O(n^2)$

```
int *Dijkstra(Graph g, int start){
    int *cost = (int*)malloc(sizeof(int) * g.size);
    if(!cost)
        return;
    int *vertex = (int*) calloc(g.size, sizeof(int));
    if(!vertex){
        free(cost);
        return;
    }
```

```
        for (int i=0 ; i< g.size; i++)
            cost[i] = g.arr[s][i];
        visited[s]=1;

        int min = INT_MAX;
        int min_i;
    for(int j=0;j<g.size ; j++) {   min = INT_MAX;
        for(int i=0; i<g.size ; i++) {
            if(!visited[i] && cost[i] < min) {
                min =cost[i];
                min_i = i ;
            }
        }
        visited[min_i]=1;

        //relaxation;
        for(int i=0; i< g.size ; i++){
          if(!visited[i]) {
              if(cost[i] > min+ g.arr[min_i][i])
                  cost[i] = min + g.arr[min_i][i];
          }

    }

  return cost;
}

// root is the cause of my problems (imbalance) :)
```

// we are setting or updating cost value on each step.

Time complexity Analysis:

| | Linked - Representation | | | Adjecency Matrix | | |
|---|---|---|---|---|---|---|
| | Best | Avg | Worst | Best | Avg | Worst |
| BFS (Queue) | $O(n)$ | $O(v+e)$ | $O(n^2)$ | $O(n^2)$ | $O(n^2)$ | $O(n^2)$ |
| DFS (Stack) | $O(n)$ | $O(v+e)$ | $O(n^2)$ | $O(n^2)$ | $O(n^2)$ | $O(n^2)$ |
| Prims: | | $O(E\log(v))$ | $O(E\log(v))$ | | | |
| without heap | $O(n^3)$ | $O(n^3)$ | $O(n^3)$ | | | |
| Kruskal | | $v\log(e)$ | $v\log(e)$ | | | |
| Dijkistra | | | $O(n^2)$ | $O(n^2)$ | $O(n^2)$ | $O(n^2)$ |

Prims with Heap:

Push all adjacent to start node:
→ pop from heap, visit that.
(*) Push all adjacent to selected node.
→ Pop again and repeat (*).

Kruskal:

Insertion $e \log e$
Removal $v \log e$ ⇒ total $(v+e)\log e \Rightarrow v\log e$

We need not iterate anything to check for edges