Introduction to graph.

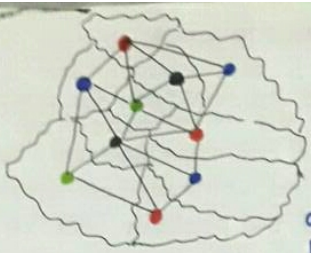
So when we design an algorithm for a problem, we need to represent the information of problem, in a way that we can manipulate it, this is called modelling. We need some kind of notation or structure to model the problem and very important class of structure is graph, which are immensily use in different context.

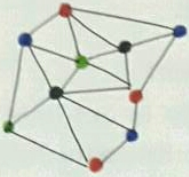
Let's start with a problem:

Map colouring: Principle is we have to colour the way so that no two boundary sharing state have same colour, because if so we will not able to distinguish states from each other. But those which are not sharing same boundary, might have different colour.

So question which may be ask is: How many colour we need to satisfy this criteria?

-Now clearly if we assign different colour to each state, then there will be even ~~two~~ states which are even not sharing same boundary will have same colour. we can do this with much less colour.





So if you look at this colour map problem, the actual map underlying these dot is not necessary any more.. So remove underlying picture and keep dot.. becz. dots keep all necessary information of graph.

So the solving this problem same as solving original map problem.

So this type of diagram is called as graph.

. The dots are vertices (one vertex, many vertices) and connection is edges.

So the problem we solve called graph colouring problem - So we used four colours, and we manage to colour whole graph with these four.

.So question might be- Is it property of this graph or this property of all graph. So infact it turn out that if you take any graph that we drew and convert to it what we did. 4 colours are always enough. This is mathematical fact about graph.

Now this is not an easy problem to solve. This is an open problem for many years and it was very celebrated theorem when it was actually proved

So one of the advantages to moving to representation like graph is that we have thrown out the inessential feature of the problem like structure, geometry Of State and so on... (we just have to know who are neighbour and not

More graph problem...

- Airline routing..

where question can be ask like connectivity.. Can I go from Shimla to ABC without changing airline.

So we donot need actual map (unnecessary things)

Graph formally:

$G=(V,E)$

- Set of vertices.
- Set of edges.
- E is a subset of pairs (v,v') : $E \subseteq V \times V$

- Undirected graph → (v,v') and (v',v) are same
- directed graph → (v,v') not always same (v',v)

- So graph coloring is a undirected graph problem
  - In which legal colour (v,v') ; $c(v) \neq c(v')$

So we can now take this graph as a mathematical object and describe the problem to be solve in a completely objective way in term of vertices and edge.

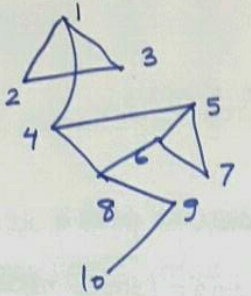
To represent graph- we have two ways.

- Adjacency matrix.
- Adjacency list.

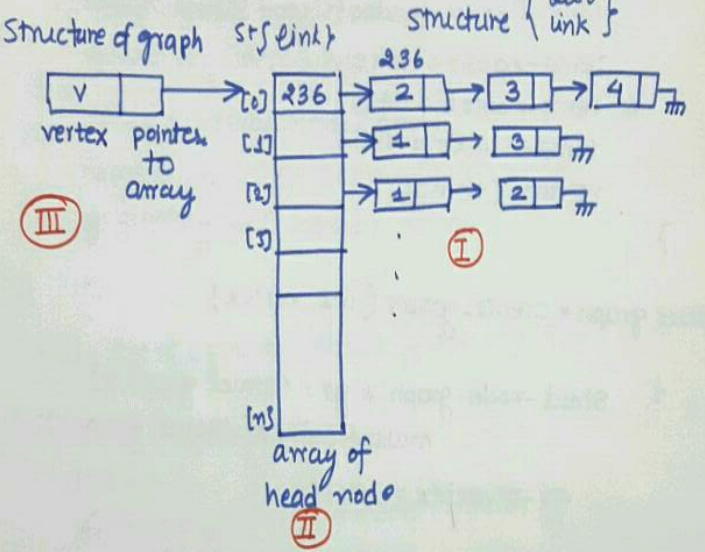
Comparing representations.

- Adjacency list typically required less space
- Is J a neighbour of i ?
  - Just check A[i][j] is 1 in adj. matrices O(1)
  - Need to scan whole list. O(m) /O(n)
- Which vertices are neighbours of i ?
  - Scan all n columns. O(n)
  - Take time proposional to neighbour in adjacency list. O(m)

Code for Adj. list representation.



1 { 2, 3, 4 }
2 { 1, 3 }
3 { 1, 2 }
4 { 5, 1, 8 }



(I)
```
struct node {
    int dest;
    struct node * next;
};
```

```
struct Adj_List {
    struct node * head;
};
```

```
struct graph {
    int vertex;
    struct Adj-List *array;
    };

struct node*create_node (int dest)
{
    struct node * temp = (struct node *)
                malloc(sizeof (struct node));
    temp->dest = dest;
   // temp->dest = dest;
    temp->next = NULL;
    return (temp);
}

struct graph* create_graph (int vertex)
{
    Stud ~~node~~ graph * gr = (struct graph *)
                malloc (sizeof (struct graph));
    gr->vertex = vertex;

    gr-> array = (struct Adj-List *)
            malloc ( vertex * sizeof (struct Adj-List));
    int i=0;
    for(i=0; i< vertex; i++)
    { gr->array[i].head = NULL; }
```

```
    return (gr);
}
void create_edge (struct graph * gr, int src,
                  int dest)
{
    struct node * temp = create_node (src);
    // gr->array [dest].head ;
    temp->next = gr->array [dest].head ;
    gr->array [dest].head = NULL ;
    temp = create_node(dest);
    temp->next = gr->array [src].head;
    gr->array [src].head = temp;
}
void print_graph (struct graph * gr)
{
    struct node * temp ;
    int i ;
    for(i=0 ; i< gr->vertex ; ++i)
    {   temp = gr->array[i].head ;
        while(temp)
        { printf ("%d", temp->dest);
          temp = temp->next;
    }   }
}
```

```
int main()
{
    int V=8;
    Struct graph * gr=  create_graph(v);
      create_edge (gr, 1, 2);
      Creati_edge (gr, 1, 3);
      creati_edge (gr, 1, 4);
       :
      print_graph( gr);
    return 0;
}
```

Breadth first traversal.

```
#include <stdio.h>
#include <stdlib.h>
# define size 40

Struct node {
      int dest;
      struct node * head;
};
Struct Adj_list {
      Struct node * head;
};
Struct graph { int v;
     struct adj_list * array;
     int * visited;
     int * parent;
};
```

```
struct queue {
        int items[size];
        int front, rear;
    };

struct node * cnode(int dest){
struct node * temp = (struct node *) malloc (sizeof
                                    (struct node));
    temp -> dest = dest;
    temp -> next = NULL;
    return (temp);
}
struct graph * cgraph(int v) {
struct graph * gr = (struct graph *) malloc (size of
                                    (struct graph));
    gr -> v = v;
    gr -> array = (struct node *) malloc (v *
                                    sizeof(node));
    gr -> visited = (int *) malloc(v*sizeof(int));
    gr -> parent = (int *) malloc (sizeof(int)*v);
    int i;
    for(i=0; i<v; ++i)
    { gr -> array[i].head = NULL;
      gr -> visited[i] = 0;
      gr -> parent[i] = -1;
    }
```

```
    return (gr);
  }

void add_edge (struct graph *gr, int src, int dest)
{
  struct node * temp1 = cnode(src);
  temp1 -> next = gr-> array [dest].head;
  gr-> array [dest].head = temp1;

  temp1 = cnode (dest);
  temp1-> next = gr-> array [src].head;
  gr-> array [src].head = temp1;
}

Struct queue * cqueue ()
  { struct queue *q = (struct queue *) malloc(sizeof
                                    (queue));
    q-> front = q-> rear = -1;
    return (q);
  }

int isEmpty (struct queue * q)
{ if (q-> rear == -1)
      return 1;
   else
      return 0;
}
```

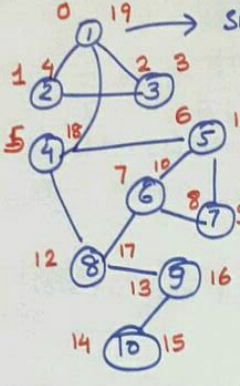
```
void enqueue (struct queue *q, int value)
{
    if (q->rear == SIZE-1)
        printf ("\n Queue is Full");
    else {
        if (q->front == -1)
            q->front = 0;
        q->rear++;
        q->item[rear=q->rear] = value;
    }
}
int dequeue (struct queue *q)
{
    int item;
    if (isEmpty (q))
    {   printf ("Queue is Empty");
        item = -1;
    }
    else {
        item = q->items[q->front];
        q->front++;
        if (q->front > q->rear)
            { q->front = q->rear = -1;
            }
    }
    return item;
}
```

```
void printQueue(struct queue *q)
{
    int i = q->front;
    if (isEmpty(q)) {
        printf("Queue is Empty");
    }
    else {
        printf("Queue contains ::");
        for (i = q->front; i<q->rear+1; i++) {
            printf("%d", q->items[i]+1);
```

```
void bfs(struct graph *gr, int start){
  struct queue *q = cqueue();
  gr->visited[start] = 1;
  enqueue(q, start);
  while (!isEmpty(q)) {
      printQueue(q);
      int currentVertex = dequeue(q);
    struct node *temp = gr->array[currentVertex].head
  while (temp)
    { int t = temp->dest;
      if (gr->visited[t] == 0) {
        gr->parent[t] = currentVertex+1;
        gr->visited[t] = 1;
        enqueue(q, t);
      }
```

```
            temp = temp -> next;
        }
    }
    printf ("Parent array is :");
    for(int i=0; i< gr->v; ++i)
  { printf ("|%d| is parent of |%d|\n",
                        gr-> parent[i], i+1);
  }
}

void path (struct graph * gr, int dest, int src)
{ int k= gr-> parent[dest-1];
  printf ("%d <--- %d ", k, dest);
  if(k==src){
    printf("\n YES! ... there is path --");
    return;
    }
  else{
    path( gr,k, src);
  }
}
int main()
{ char  // provide edge acc. to graph
    //call bfs
}   //Call path
```

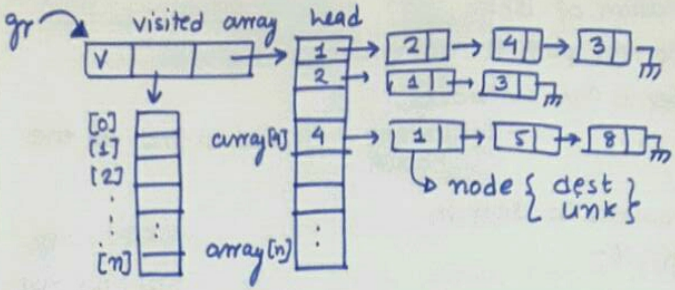
Depth First Search.



Start from the given starting point and take out the any neighbour of the 1 (S.P)

```
ALGO DFS ( S )
{
    VISITED[S] = true
    for each edge (S, u)
        If VISITED[u] == False
            DFS(u)
}
```

```
void DFS (struct graph * gr, int vertex)
{
  struct node * temp = gr -> array [vertex].head;
   gr -> visited [vertex] = 1;
  while (temp != NULL)
    { int k =     temp -> vertex;
      If ( gr -> array visited [k] == 0)
          { DFS ( gr, k);
          }
      temp : temp -> next;
}   }
```



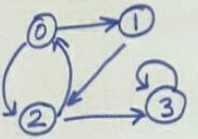
Use- Stack (no need to explicitly maintain stack)
- Recursive calls.

Applications of DFS
- DFS produce (minimum spanning tree) for an unweighted graph.
- Detecting a cycle: If you have back edge, then there is a cycle.
- Path Finding.
- Topological Sorting.
- To test graph is biparte or not.
- Finding Strongly connected components in graph.
- Solving puzzles with only one solution
  - Such as mazes

Application of BFS.

1) Shortest path in an unweighted graph.
2) Peer to Peer networks.
   - BitTorrent, It is used to find all the neighbour.
3) Crawlers in Search Engines-
4) Social Networking websites
5) GPS Navigation system
6) Broadcasting in Networks.
7) In Garbage Collection
8) Cycle detecting in an undirected graph.
9) Ford-Fulkerson algorithm. (maximum flow)
10) To Test graph is biparte or not.
11) Path Finding

CYCLE - USING DFS IN AN ~~UN~~DIRECTED GRAPH.



Logic - because it is undirected graph.

We will use simply a stack, which help us, suppose if i strart from so we put zero on stack and during DFS if we again meet zero, we will say there is a cycle in graph. and when we are done with zero we will put it out from stack. Similarly feel all above thing when you call a neighbour of 0 (from 0).

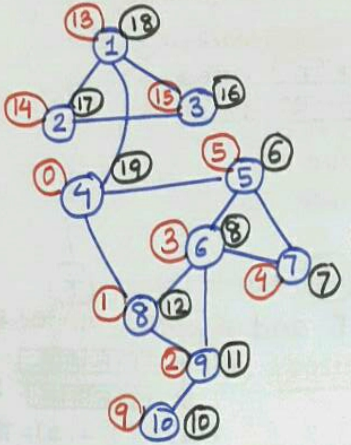
```
bool isCycleUtil(int v, bool visited[], bool Stack[])
{
    If (visited[v] == false)
    {   visited[v] = true;
        Stack[v] = true;
        struct node * temp = gr->array[v].head;
        while (temp != NULL)
        { int k = temp->dest;
          If (visited[k] == false && isCycleUtil
                                     (k, visited, stack))
          {
              return (true);
          }
```

```
        else if (stack [k] == true)
            { printf ("I got a cycle :::" );
              return(true) ;
            }
        temp= temp-> next;
    }
  Stack [v]= false ;
  return false;
}
}
bool  isCycle ( ){
     bool  visited [gr-> v], stack [gr->v];
    for(int i=0; i< gr->v ; ++i)
       { visited[i]= false;
         stack [i]= false;
       }
    for( int i=0; i< gr->v ; i++)
        if ( isCycleUtil (i, visited, stack))
             return (true);

    return(false);
}
```

Let's look by example also.

Application:

Recording Pre[] and Post[] of a node
- when we enter when we exixt!



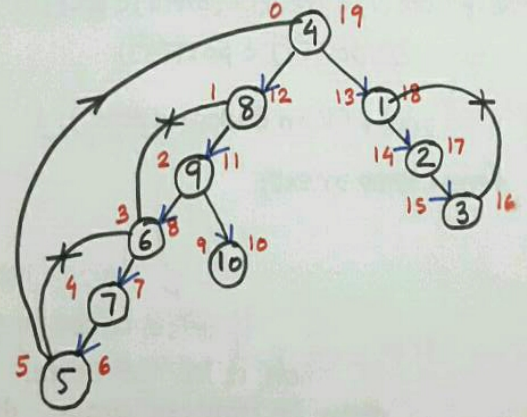
→ pre value.
→ post value.

So I need two array pre[size] and post[] to record when we enter and when we leave...
we can achiere-this during our Dfs so only thing that is extra is the introd. of these array.
So we put pre[] = 1 to some (static i) and then obiously increment that. and when we end with exploring we put it's post value also (when nothing remain to explore).

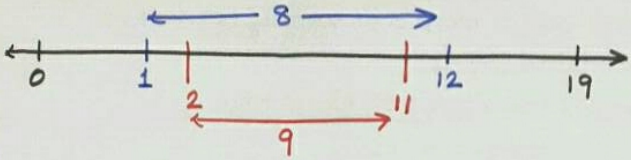
```
Static int m=0;
int pre[V];
int post[V];
```

```
void dfs (struct graph *gr, int s)
{
  struct node * temp = gr->array[s].head;
  gr->visited[s] = 1;
  pre[s] = m++; // pre[s] = m++;
  while (temp != NULL)
        int k = temp->dest;
    if (gr->visited[k] == 0)
    {   dfs (gr, k);
    }
    temp = temp->next;
  }
  post[s] = m++; // post[s] = m++;
}
```

we know DFS traversal explore DFS tree. let's see.



(8,9)



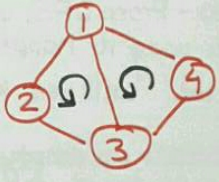

```
void edge ( struct graph * gr ) {
  for (int u=0 ; u<gr->v; u++)
     { struct node * temp= gr->array [u].head;
       while (temp!= NULL)
         { int v= temp->dest;
           if ( pre [u]< pre [v] && pre[v] <post [v]
                  && post [v] <post [u])
              { printf (" \n Forward edge ");}
           else if ( pre [v] < pre[u] && pre[u]< post [u]
                  && post [u] < post [v])
              {
                printf (" \n Backward edge");}
           temp= temp->next;
         }
     }
}
```

If it is a back edge then it is guaranted than in directed graph it is a back edge.

CYCLE IN AN UNDIRECTED GRAPH.

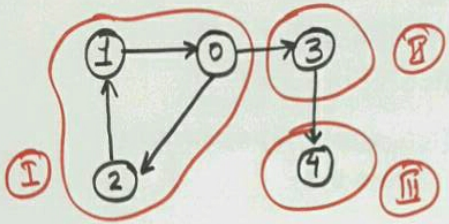
USING DFS.

```
bool isCycleUtil( int v, int parent, bool visited[] )
{
    struct node * temp = gr->array[v].head;
    visited[v] = true;
    while(temp)
    {
        int K = temp->dest;
        if(visited[K] == false)
        {
            if(isCycleUtil(K, v, visited))
                return true;
        }
        else if ( K != parent)
            return true;
        temp = temp->next;
    }
    return(false);
}

bool isCyclic() {
    bool visited[gr->V];
    for(int i=0; i<gr->V; ++i)
        visited[i] = false;
```



```
for(int u=0; u< gr->V ; u++)
    if ( visited[u]== false)
        if (isCycleUtil(u, -1, visited))
            return true;
return false;
```

Strongest connected components. (SCC)



Idea - From DFS using. So we do that but change the way it happen. Introduce a stack explicitly so we will use KOSARAJU_ALGORITHM.

1) Create an empty stack and when DFS(u) is called put it to stack when ~~all~~ nothing remain to explore put it to the stack.

2) Reverse the direction of all edges. (Transpose of gr)

3) Pop the element from stack and start DFS again for them.

- Stack (top, pop, push)

```
struct graph * tgraph(struct graph * gr)
{
    struct graph * g1 = cgraph(gr->v);
    for(int v=0; v< gr->V ; ++v)
    { struct node * temp= gr->array[v].head;
      while (temp!=NULL)
      { int k = temp->dest;
        addedge (gr1, k, v);
        temp = temp->next;
        printf("\n");
      }
    }
    printf ("\n GRAPH AFTER TRANSPOSE : \n");
    print(gr1);
    return(gr1);
}

void dfsUtil(struct graph * g , int v, bool visited[])
{   visited[v] = true;
    printf ( "%d", v);
    struct node * temp = g->array[v].head
    while (temp!= NULL)
      { int k= temp->dest;
        if(visited[k]== false)
```

```
            { dfsUtil(g, temp->dest, visited);
            }
        temp = temp->next;
    }
}

void fillOrder(int v, bool visited[], stack *s)
{   visited[v] = true;
    Struct node * temp = gr->array[v].head;
    while (temp != NULL)
      { int k = temp->dest;
        If (!visited[k])
          { fillOrder(k, visited, st);
          }
        temp = temp->next;
    }
   push(st, v);
}

void printscc(){
      Struct stack *st = cstack();
      bool visited[gr->v];
     for(int i=0; i<gr->v; ++i)
        { visited[i] = false;
        }
```

```
for(int i=0 ; i< gr->v ; ++i)
 {   if (!visited[i])
        { fillorder (i, visited, st);
        }
 }
for(int i=0; i<gr->v ; ++i)
  { visited[i]= false; }
Struct graph *gr1 = tgraph(gr);
while (isempty(st) == false)
    { int k = top(st);
      pop(st);
      If (visited[k]== false){
                dfsutil (gr1, k, visited);
          }
    }
}
```

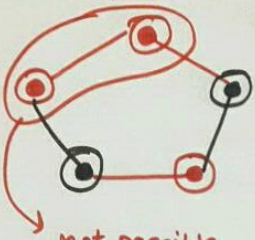
BIPARTE : CHECH FOR A GIVEN GRAPH.

Color the graph with just two colour : is it possible . if then biparte , if not then not biparte.

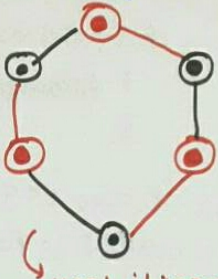
So Logic : DFS

Start colour from any node colour its adjacent

node opposite the colour it has e.g.



not possible to colour with red and black



yes! it is possible

So in original you need not to set the colour actually but (0 and 1) will work. mean colour (1) with 0 and (2) with 1 and so on if if the last does not appear with same colour then.

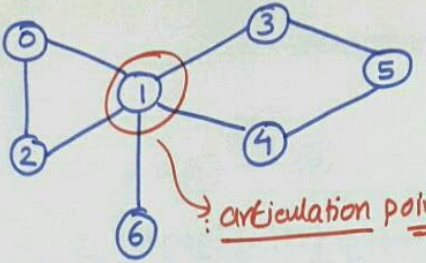
```
bool isbiparte(int v, bool visited[], int color[])
{ struct node * temp = gr->array[v].head;
  while (temp != NULL)
    { int k = temp->dest;
      If (visited[k] == false)
        { visited[k] = true;
          color[k] = !color[v];
          If (!isbiporte (k, visited, color)
              return(false);
```

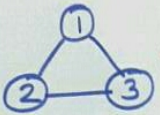
```
        }
        else if (!biparte(k,
      }
      else if (color[k] == color[v])
        { return (false); }
      temp = temp->next;
    }
  return (true);
}
int main()
{   bool visited[7];
    int color[7];
    set all visited =false
    and then start from any say 0
    visited[0]=true;
    set any colour (mean number (0,1) to start.
    color[0] = 1/0;
    if (biparte(0, visited, color))
    { printf("yes!");
    }
    else
    {printf("no");
}.
```

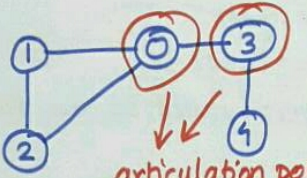
ARTICULATION POINT.

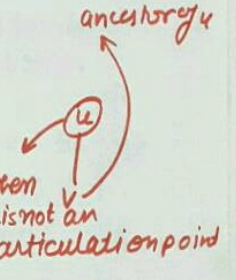
```
void apu (int v, int parent[], bool visited[],
                bool ap[], int low[], int disc[])
{
    int children = 0;
    Static int t = 0;
    disc[v] = low[v] = ++t;
    visited[v] = true;
    Struct node * temp = gr->array[v].head;
    while (temp != NULL)
    {   int k = temp->dest;
        if (visited[k] == false)
          {  children ++;
             parent[k] = v;
             apu(k, parent, visited, ap, low, disc);
             low[v] = min(low[v], low[k]);
          *  if (parent[v] == -1 && children > 1)
                { ap[v] = true;
                }
            if (parent[v] != -1 && low[k] > disc[v])
              { ap[v] = true;
              }
          }
        else if (k != parent[v])
          {
```

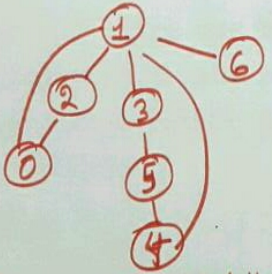
```
        low[v] = min(disc[k], low[v]);
      }
    temp = temp->next;
  }
}

void ap() {
  bool visited[gr->v], ap[gr->v];
  int parent[gr->v], disc[gr->v], low[gr->v];
  for(int i=0; i<gr->v; ++i)
  { parent[i] = -1;
    visited[i] = ap[i] = false;
  }
  for(int i=0; i<gr->v; ++i)
  { if(visited[i] == false)
    { apu(i, parent, visited, ap, low, disc);
    }
  }
  print articulation points;
}
```

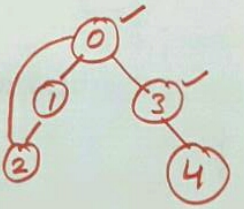

0
1
2
3
4
5
6
articulation point.


1
2
3
: no articulation point


1
0
3
2
4
articulation point.

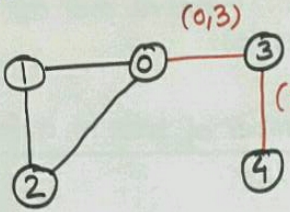

ancestor of u
u
then
u is not an
articulation point

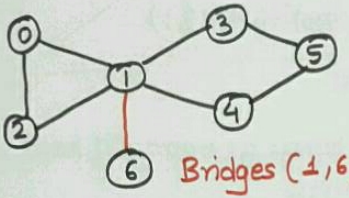

1
6
2
3
0
5
4
1 has more than 1 child so 1 is ap


0
1
3
2
4

BRIDGES IN A GRAPH.

- generally represent crucial wire in the network system.





Again it is same like the way we have find the articulation point just we have print the bridge mean (0,3) beside of $\binom{0}{3}$ ok that's just (3,4)

simple..

```
if (low[k] > disc[v])
    printf("%d ----- %d \n", v, k);
```

all other code is same... (remove the ap[] array, because that is of no need).

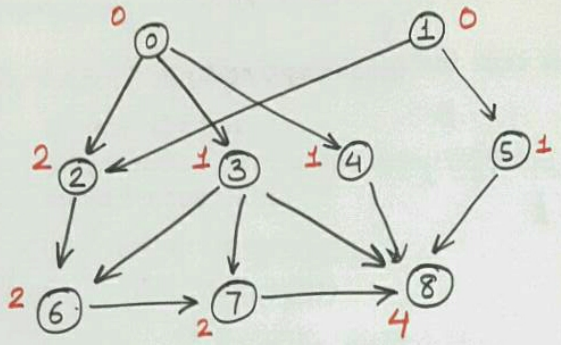
Very Simple: Count Number of tree in a forest.

Logic: DFS

```
void DFSutil(int u, bool visited[])
{   visited[u] = true;
    for(int
    struct node * temp = gr->array[v].head;
    while(temp != NULL)
    {  int k = temp->dest;
       if(visited[k] == false)
          { DFSutil(k, visited); }
       temp = temp->next;
    }
}
int Ctree(){  bool visited[gr->v]; //set them false
    int res = 0;
    for(int u=0; u<gr->V; ++u)
    {  if (visited[u] == false)
          { DFSUtil(u, visited);
            res++; }
    }
}
```

SELF-TOPOLOGICAL SORTING.

Topological sorting is actually done in Directed Acyclic Graph. (DAG's)



indegree [i] = no. of edges coming to the vertex i.

```
void toUtil (int degree[]){
    for (int v=1 ; v< gr->v ; v++)
        { if (indegree[v]==0)
          indegree[v] = -1;
          enqueue (q,v);
          while (isEmpty (q) == false)
          { int u= dequeue (q);
            indegree [u] = -1;
            printf ("%d ", u);
            struct node * temp= gr -> array[u].head;
```

```
                while (temp!= NULL)
                 { int k= temp→ dest;
                   indegree[k]= indegree[k]-1;
                   if (indegree[k] == 0)
                    { enqueue (q,k);
                     }
                   temp=temp→next;
                 }
              }
           }
        }
     }
```

```
void  topl_order(){
       int  indegree [gr→v];
    for(int i=0; i<gr→v; i++)
      { indegree [i]= 0; }
   for(int i=0 ; i<gr→v; i++)
  { struct node * temp= gr→array[i].head;
    while(temp!=NULL)
   { int k= temp→dest;
     indegree[k]++;
     temp= temp→next;
   }
}
```

```
    toUtil (indegree);
}
```

Another simple method - using DFS mech alone.

: Stack using.

```
void tsUtil (int v, bool visited[], struct stack *st)
{   visited [v] = true;
    struct node * temp = gr-> array [v]. head;
    while (temp! =NULL)
       { int k = temp-> dest;
         If ( !visited [k])
              { tsUtil(k, visited, st);
              }
         temp = temp-> next;
       }
   push(st, v);
}
void topological_sort() {
    structure stack * st = cstack();
    bool visited [gr->v];
    for (int i=0; i< gr-> v ; ++i)
       { visited [i] = false; }
    for (int i=0; i< gr-> v; ++i)
       { if (!visited [i])
```

```
            { tsUtil (i, visited, &st);
            }
    }
printf ("Topological order :");
 while (isempty(st) == false)
     { int k = top(st);
       printf ("%d -->", k+1);
       pop(st);
    }
}
```

WEIGHTED GRAPH.

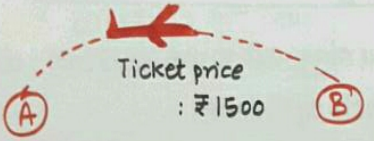
These are the graph which associate some cost with each edge. The most important thing that we can do with these graph is calculating the shortest path.

So far we have seen unweighted graph. and explore them using BFS and DFS. where we use BFS to compute shortest path in term of number of edges in the path.

DFS reveals a lot of structural property.

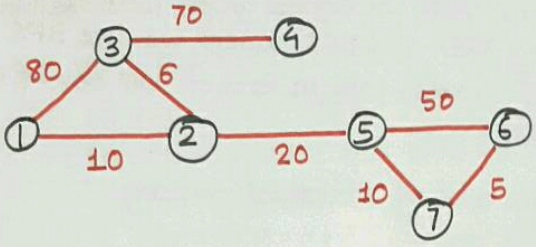
Adding edge weight.

- Ticket price on a flight sector.
- Tolls on highway segment.
- Distance b/w two stations.
- Typical time b/w location during peak hour traffic.



So in general weighted graph are just normal graph with extra information which tells us cost of each edge (consider that number $\in R$).

$w: E \rightarrow Reals.$



To Compute shortest path - Dijsktra's algorithm.

-First we will look a solution that ig not that much efficient but easy one to implement and understand.

we will use - Adjacency matrix representation.

and the algorithm turning out to be a $O(n^3)$ algorithm which is not that much efficient.

Dijsktra's Algorithm.

```
#include <stdio.h>
#include <limits.h>

#define V 9

int mindistance (int dist[], bool visited[])
{
      int min = INT_MAX, min_index;
     for (int v=0; v<V; ++v)
       if (visited[v] == false && dist[v] <= min)
           { min = dist[v];
             min_index = v;
           }
     return min_index;
}
```

bottleneck.
.. later we will see how to resolve this.

```
void dijkstra (int graph[V][V], int src)
{
      int dist[V];
      bool visited[V];
     for (int i=0 ; i<V ; ++i)
      { dist[i] = INT_MAX;
        visited[i] = false;
      }
     dist[src] = 0;
```

heaps and red-black tree.

Dijsktra's Algorithm.

```
#include <stdio.h>
#include <limits.h>

#define V 9

int mindistance (int dist[], bool visited[])
{
      int min = INT_MAX, min_index;
      for (int v=0; v<V; ++v)
        if (visited[v] == false && dist[v] <= min)
            { min = dist[v];
              min_index = v;
            }
      return min_index;
}

void dijkstra (int graph[V][V], int src)
{     int dist[V];
      bool visited[V];
      for (int i=0 ; i<V; ++i)
       { dist[i] = INT_MAX;
         visited[i] = false;
       }
      dist[src] = 0;
```

bottleneck.
.. later we will see how to resolve this.

heaps and red-black tree.

```
    for (int c=0 ; c < V-1 ; ++c)
    {
        int u = minDistance (dist, visited);
        visited[u] = true;
        for (int v=0 ; v<V ; ++v)
        {
            if (visited[v]== false && graph [u][v]
                    && dist[u] != INT_MAX
                        && dist[u] + graph[u][v] < dist[v])
                dist[v] = dist[u] + graph[u][v];
        }
    }
int print solution (int dist[], int n)
{ printf ( " vertex Distance from Source \n");
  for (int i=0 ; i<V ; ++i)
      printf ("%d --->%d", i , dist[i]);
}
int main(){
    int graph [v][v] = {{0,4,0,0,0,0,8,0}
                                . . };
    dijkstra (graph,0);
    return (0);
}
```

Limitations: No negative weight if that then algorithm fail

Analysis of Dijkstra's algorithm: (Greedy algorithm)

So in this we are using visited[] array, when we are selected the minimum edge weight, we immediately mark it visited[]: true.

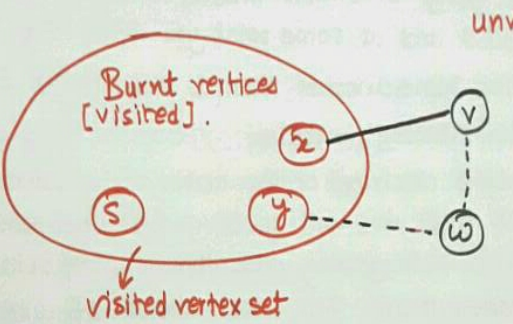
well before this discussion, it is a greedy algorithm and we have to justify the order in which this algorithm is picking vertex is correct, ~~at e~~. so we have to justify that choice that you are making is always going to remain undone ... that is you keep going forward and at some point you don't say ohh! I have pick some another then it was correct. So this type of strategy is called greedy strategy.

So we make decision on the basis of local choice and think that this will yield global optimum solution- for such greedy algorithm Its important to established that this local choice of next step give us global optimum because very often these local choice does not give correct algorithm.

So let's looks for the correctness of the Dijkstra's algorithm.

Correctness.

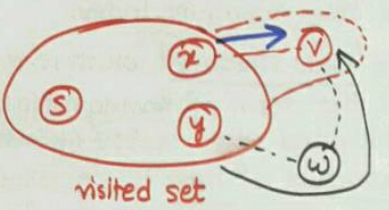
The key to prove correctness of Dijkstra's algorithm is what we call "Invariant". Invariant in terms of visited[] information and what we want to claim is that burn vertices are correctly solve. mean at any time if we look the set of visited[] vertex in the visited set then distance that is computed is actually the shortest distance blw two vertices. So we assume that this iterative "invariant" is true, which at begining is true because at that time, the only burn vertex that we have is starting vertex.



Now assume that we have extend our set to some other vertex (x,y) and then we pick up a vertex "v". So if we look the distance

$d(x) + w(x,y)$ is the smallest among all the vertices which are not visited.

Now the claim is that now if we add this v to visited set (red), then the distance of v (cannot be smaller than) what we have computed now i.e $(d(v)+w(x,v)$ because of later update. later update mean that suppose if we include w later on then there is no path from w that is shorter than what we have selected



So if we see

$d(y)+w(y,w)$ this must be either > or equal to $d(x)+w(v,x)$ because we choose at that time v not w so $d(y)+w(y,w)$ can be less than $d(x)+w(y,x)$

and so it can be same as $d(x)+w(x,v)$ but when you go from w to x it always become greater.

Complexity:- O(n) ; set visited [i]: false
Distance [i]: INFINITY

; another loop O(n) { O(n) - select minimum
{ O(m) - list
{ O(n) - matrix } depend on you

$O(n)$ ~ { $O(n)$ – finding minimum-index
$O(m)$ – adjacency list
$O(n)$ – adjacency matrix.

~ $O(n^2)$ < list, matrix both.

So bottle neck is only minimum-index.

So we need that "data structure" which is more sophisticated in the term of finding minimum and remove quickly and also update efficiently.

and that data structure is "heap" — $O(\log_2 n)$

insert min, find min, delete min

$O(n)$ - { $O(\log_2 n)$ ~ choosing a vertex
$O(m)$ ~ updating distance

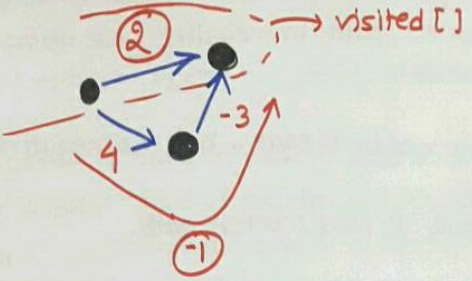
~ $O(n \log n + m \log n)$

~ $O((n+m)\log n)$ → more efficient than naïve $O(m^2)$ implementation.

Limitation

what if negative edge?

The criteria that we use for "the invariant". will fail.



:: Why negative weights ?

:: Weight represent money

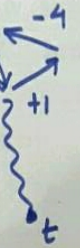
- Taxi driver earn from airport to city, travels empty to next pick-up point.
- Some segment earn money, some lose.

:: Chemistry -

- Nodes are compounds, edges are reactions
- weight are energy absorbed / released by reactions.

How to handle ?

- Negative cycle : negative edge are on one hand and having negative cycle on other



So I can make the distance from source to target arbitary small by running loop in negative cycle (-4)

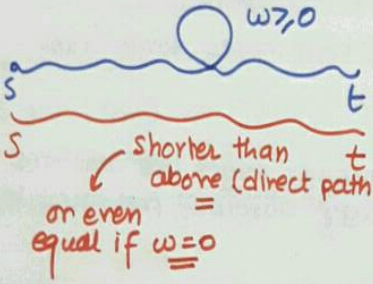
. So having shortest path in negative cycle do not make any sence.

· So I could have negative edge ... but no negative cycle.

- Bellman-Ford
- Floyd-warshall all pairs shortest path.

Before going to these algorithm, lets look some basic property of the "shortest path".

· Shortest path will never go through loop.



mean we are not visiting same vertex twice.

so if you have n vertex then this is fix that it is going to have n-1 edges

2) Every prefix of a shortest path itself is a shortest path.

$$s \to v_1 \to v_2 \to v_3 \ldots \to v_m \to t$$

Then path till v_m ($s \to v_m$) is a shorter path and if suppose there is another shorter path from $s \to v_m$

another path which is shorter than ($s \to v_m$) provided

$$s \to v_1 \to v_2 \to v_3 \ldots \to v_m \to t$$

then for $s \to t$ the path will be -

(new discovered red path ($s \to v_m$)) + ($v_m - t$)

example:

Suppose till (v_{m-1}) we know only shortest path from $s \to v_2$ via v_1 ($s \to v_1 \to v_2$) and later on we find some another shortest path.

new path

$$s \to v_1 \to v_2 \to v_3 \to \ldots v_m \to t$$

then we have to modify all other shortest path as previous is changed.

So new $s \to v_2 + v_3$ and soon...

So these two property are enough to arive us at Bellman Ford Algorithm.

Let notice what were things that miss out in Dijkstra's algorithm.

Update Distance()

So in Dijkstra's algorithm when we visited a vertex j. then we modify each edge (j,k) with

Distance[K] = min(Distance(K), Distance(j) + weight(j,K))

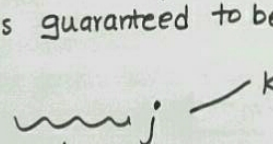
↓

update(j,k)

↓

In Dijkstra's algorithm we only do this update when we visited[j]

- when we compute update(j,k), Distance(j)
- Focus* Acc. to Dijkstra's algorithm Distance[j] is always guaranteed to be correct.



This is always right acc. to Dijkstra's

but it is onot always. Is it? No, in negative edge weight this is not true

So these two property are enough to arrive us at Bellman Ford Algorithm.

Let notice what were things that miss out in Dijkstra's algorithm.

Update Distance()

So in Dijkstra's algorithm when we visited a vertex j. then we modify each edge (j, k) with

Distance[K) = min (Distance (K), Distance (j) + weight (j, K))

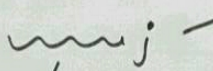
↓

update(j, k)

↓

In Dijkstra's algorithm we only do this update when we visited [j]

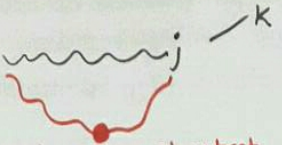
- when we compute update(j, k), Distance (j)
- Focus* Acc. to Dijkstra's algorithm Distance [j] is always guaranteed to be correct.



↓

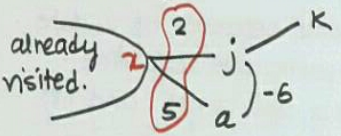
This is always right acc. to Dijksha's

but it is onot always. Is it? No, in negative edge weight this is not true



we might have another shortest path. (which include negative edge)

So when we be visited [j] , this is not always gauranted that this is the shortest path, because it is locally choose.



at local choice we select $d[j] = d(x) + w(x, j)$ (2)

but it is not correct, but if we take

$x \to a \to j$ (-1) but Dijkstra's select minimum distance from x (which is j not a) so the distance to k is also wrongly calculate. but on later on we know this thing from a there is another shortest path and thus we have to modify j - k as d[j] is modify..

But this update has some useful prop.

- Update give us upper point to the dist[K] so "invariant" that we are having, is the value to the distance is greater or equal to th "actual distance"

So if later on we find some path which is shorter to some j, then all edges coming out from j are also to be modify as distance to j is modified.

Important point:

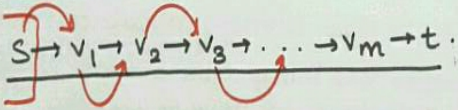
Update distance is safe.. because it will never bring ba the distance shorter than the actual it is.

- Redudant update cannot hurt.

So Dijkstra's algorithm choose some sequence of update and yield a shortest path. (a particular sequence - choose smaller vertex which is not burn). and it fail in case of negative.

- Is there something? another way.

Suppose we have to find SP from $s \to t$



if we do update in this order 1st S then v_1 ...so

then what do we know.. well we know that if we have computed distance till v_3 correctly then to v_1 we will always compute correctly.

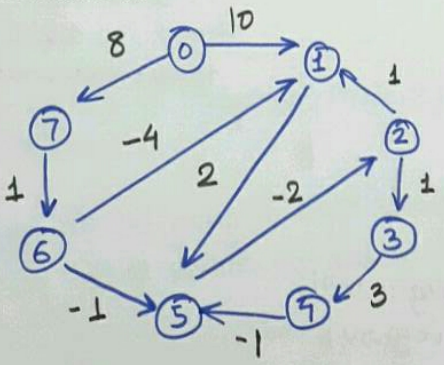
So Bellman-Ford algorithm say do not compute a particular sequence... just generally compute all possible distance.

```
#include <stdio.h>
#include <stdlib.h>
#include <limits.h>

int min(int a, int b)
{
  if (a<b) return a;
   else
     return b;
}
struct graph * gr= NULL;
struct node {
       int dest;
       int weight;
       struct node * next;
};
struct list {
     struct node *head;
};
struct graph {
     int V;
     struct list * array;
};
```

```
struct node * cnode (int dest, int w)
{ struct node * temp = (struct node *) malloc(sizeof (struct
                                                    node));
    temp->dest = dest;
    temp->weight = w;
    temp->next = NULL;
    return (temp);
}
struct graph * cgraph (int v) {
    struct graph *gr = (struct graph *) malloc (sizeof (struct
                                                    graph));
    gr->v = v;
    gr->array = (struct list *) malloc (sizeof (struct list) *v)
    for (int i=0; i<gr->v; ++i)
    { gr->array [i].head = NULL;
    }
    return (gr);
}
void addedge (struct graph *gr, int src, int weight, int dest
{ struct node * temp = cnode (dest, weight);
    temp->next = gr->array [src].head;
    gr->array [src].head = temp;
    temp = NULL;
    free (temp);
}
```

```
void bellman(int src) {
    int dist[gr->v];
    for(int i=0; i< gr->v; ++i)
      { dist[i] = INT_MAX;
      }
   dist[src] = 0;
  for( int j=0 ; j< gr->v ; ++j)
     {
       for(int i=0; i<gr->v; ++i)
          { if( dist[i] != INT_MAX)
              {
                struct node * temp = gr->array[i].head;
                 while(temp != NULL)
                   { int j = temp->dest;
                     int w = temp->weight;
                     dist[j] = min(dist[j],
                                   dist[i]+w);
                     temp = temp->next;
                   }
              }
          }
     }
print f ("printing :: \n");
 for( int i=0; i<gr->v; ++i)
```

```
{ printf ("\n %d| --->---|%d|-->--|%d|", src+1, dist[i,i+1]);
  }
printf ("\n ---------\n");
}
int main(){
    gr = cgraph(8);
    addedge(gr, 0, 10, 1);
    addedge(gr, 0, 8, 7);
    addedge(gr, 1, 2, 5);
    addedge(gr, 2, 1, 1);
    addedge(gr, 3, 3, 4);
    addedge(gr, 4, -1, 5);
    addedge(gr, 5, -2, 2);
    addedge(gr, 6, -1, 5);
    addedge(gr, 6, -4, 1);
    addedge(gr, 7, 1, 6);
    bellman(0);
}
```



bellman(0)

	↓
0	0
1	5
2	5
3	6
4	9
5	7
6	9
7	8

Till now we have seen shortest path b/w particular source and all remaining destination or either a pair {(src, v_1) (src, v_2) (src, v_3)}

But suppose if you are running travelling website or handling the air flight details and in that case beside of just shortest distance b/w a src to all other remaining is not enough, but b/w any pair actually you should have this information. Okay so we will look for some new algorithm which will compute all pair shortest path.

Okay so the way by which we generalized Dijkstra's to Bellman Ford algorithm we will generalized Floy Bellman Ford (For all pair shortest path).

Again we start from the fact what we know about shortest path.

- It is never a loop.
- At most n-1 length

- We will use this to inductively explore all possible path (shortest).

- Basically we are going to build "shortest path" in term of what vertices we allow or what not

so the simplest shortest path that you can imagine b/w any two pair is just the direct edge

But :: (imp) in general this may not be the shortest path .. there may be another way (because of negative edge)

But we do know because of this characterization of shortest path :: no vertex repeat

so $i \to v_1 \to v_2 \to v_3 \cdots \to v_m \to j$

no thing repeat (none even i or j) — say if this is actual shortest path. :: no loop

So what we do for this inductive thing, "Restrict" what can happen b/w i and j, what are vertices b/w i and j that you can include and we gradually increase our set

: as atmax $n-1$ vertices can came (if can also find shortest path before that .. but we have upper bound of $n-1$, that either that time or before we will have shortest path in our hand. So we will do

"Inductive exploration of shortest path"

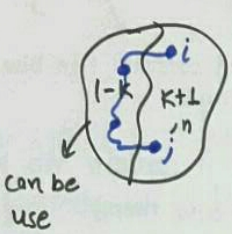
↳ first $\{v_1\}$

then second iteration $\{v_1, v_2\}$

- Say $w^k(i,j)$: weight of shortest path from i to j among paths that only go via $\{1,2,..k\}$

  - $\{k+1,...n\}$ cannot appear on path.
  - $\{i$ and $j\}$ themself need not on path b/w $\{1,2,..k\}$: having i again mean that you are on loop.

So



So in particular if k is zero then $w^0(i,j)$ mean from $\{0+1,..n\}$ nothing can appear on the path. so mean only direct edge is referred there no vertex $\{1,2,3,...n\}$ can appear on path. Now this is an inductive definition.

- From $w^{k-1}(i,j)$ to $w^k(i,j)$

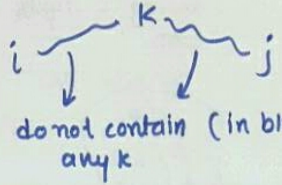
1) Case : I Shortest path via $\{1,2,...k\}$ does not use vertex k

  - $w^{k-1}(i,j) = w^k(i,j)$

Case 2: On the other hand including k give us some non trivial improvement. which mean path from $\{1, 2, 3, \ldots k\}$ includes k.

- k can appear once in the path.

So break the path.



Focus: from i to k (b/w them it contain no k) and also from k → j (no k b/w them).

focus it mean to say $\{1, 2, \ldots k-1\}$

So break the path:

$$w^k(i,j) = w^{k-1}(i,k) + w^{k-1}(k,j)$$

oh! and I know this distance. it is actually $w^{k-1}(i,k)$

So two cases: either we do not use k or use k so we have to pick minimum dist.

conclusion:

$$w^k(i,j) = \min\left(w^{k-1}(i,j), \left[w^{k-1}(i,k) + w^{k-1}(k,j)\right]\right)$$

So this give us immediately an algorithm which is Floyd's Warshall Algorithm

- $w^0(i,j) - w^0[i][j]$
  - w^0 is adjacency matrix with edge weights.

$$w^0[i][j] - \begin{cases} \text{weight}(i,j) \text{ of edge} \\ \infty \text{, otherwise ; because I am allowed direct edge (one)} \end{cases}$$

- For $k = 1, 2, \ldots n$, I basically repeat this n time
  - Compute $w^k(i,j)$

$$w^k(i,j) = \min\left(w^{k-1}(i,j), w^{k-1}(i,k) + w^{k-1}(k,j)\right)$$

- w^k contains weight of shortest path for all pairs
  - ↳ Imp:: This is no constrain case , I can include all edges.

Complexity - $O(n^3)$
- no need to move to Adj. list
- It is a Adj. matrices based algorithm.
- It sort of solve - Bellman Ford ~ $O(mn)$ $O(n^3)$ ~ - It is a Generalised solution for

Space complexity:: $O(n^2)$

As for $w^1(i,j)$ you just need to know $w^0(i,j)$ and similar for $w^k(i,j)$ only $w^{k-1}(i,j)$ nothing previous that okay so it just require two n^2 array.

Historical Remarks: See Floyd-Warshall - is a hybrid name

So originally there were two algorithm given alone.

Warshall propose - "transistive closure".

↙

It is like locating a path or computing a path using edge relations

- Like suppose in a group of people :: you know who is direct friend of whom

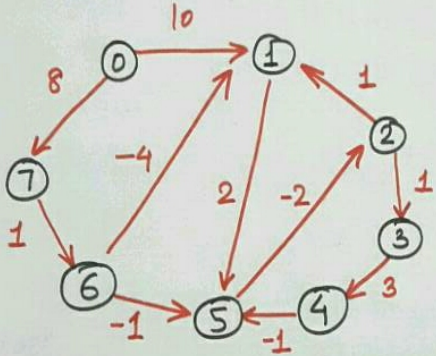
and suppose if I ask - who "knows" indirectly ? So I know someone indirectly if mine friend is friend of that person ~So it is a "transistive closure" of friend. function.

~ and Instagram / Facebook cleverly use this to recommend you friend that you may know

Floyd warshall
-All pair shortest path problem:

```
#include<stdio.h>
#include V 8
#define INF 999999

int min (int a, int b)
    { if (a<b) return a;
      return b;
    }
void printSolution ( int dist[V][V])
{
   for (int i=0; i<V; i++)
   { print
     for(int j=0; j<V; ++j)
       {
          if ( dist[i][j] == INF)
                printf ("%7s ", "INF");
          else
              printf( "%7d", dist[i][j]);
       }
   }
}
void Floyd_warshall (int graph[V][V]){
          int dist[V][V];                    //Setting w^(i,j)
          for(int i=0; i<V; ++i)
            { for (int j=0; j<V; ++j)
                 dist[i][j] = graph[i][j];
            }
```

```
for(k=0; k<V; ++k)
{   for(int i=0; i<V; ++i)
    {   for(int k=0; k<V; ++k)
        {   dist[i][j]= min(dist[i][j],
                                dist[i][k] + dist[k][j]);
        }
    }
}
printSolution(dist);
}
int main(){

    }
```



Minimum Spanning Tree.

· Prim's Algorithm:- Correctness of Prim's Algorithm.

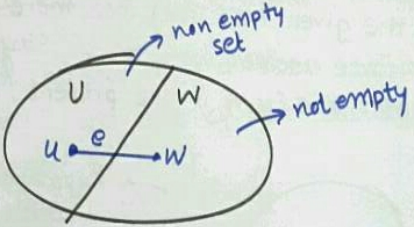
It is a greedy algorithm. since it is then we have to prove the correctness. Greedy in the sence at each point we have to decide how to extend the tree. So we look for the neighbourhood of the existing tree and add minimum cost edge to our tree.

"A local heuristic is used to decide which edge to add next to the tree."

Choice made are never reconsidered.

So we have to prove our claim!.

So in order to prove the prim's algorithm, we use a very useful lemma called "minimum separator lemma"

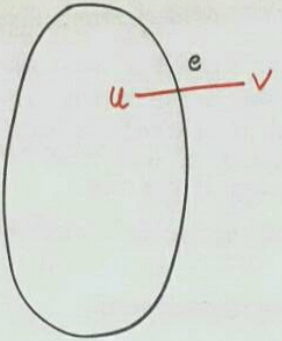


Now lets look for the smallest edge (u-w). Now claim is that every MCST must contain this edge .claim is very powerful claim

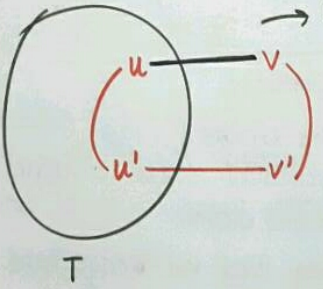
- Assume also that no to edge have same weight

Then every minimum cost spanning tree must include e

Let's assume we have these two parts



u-v smallest edge: and claim is it is in every minimum cost spanning tree. So suppose it is not then, there must be some another spanning tree (the time I say spanning tree mean all vertices in the graph are all present in the given MCS Tree.) Then there must be some how some another way to v. Say we have tree T in which ~~in~~ e (u, v) is not present. So in that case.



→ hypothetical tree T which do not contain u-v edge.

Now the claim is that If I take that particular tree remove the edge $(u'-v')$ and replace it with $(u''-v)$ and say that tree is T'.

$$T' = T - (u',v') + (u'',v)$$

Now by the assumption (u,v) is smallest edge

So $(u',v') > (u,v)$

$(u',v') - (u,v) > 0$

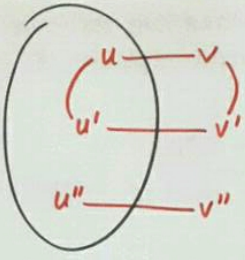
So $T' = T - ((u',v') - (u,v))$

always > 0

So $T' < T$ and also in T' every thing is connected - This is actually a imp point because $(u'-u-v-v')$ of this you can say T' is a "valid tree" or "valid spanning tree". and also of lower cost than T. and thus T cannot be MST.

So This is our Proof.

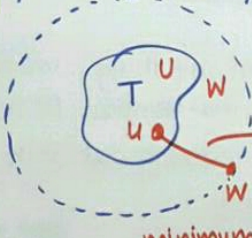
Imp :- We have note down one thing... we have to care when we are proving this lemma. It is true that among all the edges coming out from T(existing tree) u-v is smallest



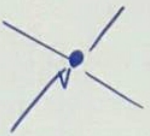
So one might do (u-v) is smallest and pick any vertex set (u", v") and disconnect them and replace by the u-v edge. Now Focus on one thing removing u"-v" and joining u-v will let you no way to v". because they are not connected like (u-u'-v'-v). So it is cructial we must choose right edge. (No Arbitary edge to pick).

So once we have this lemma the correctness of Prim's algorithm is very obious.

So at every stage in Prim's we construct a tree T (which have few edge) and every thing outside. and now we want to connect smallest edge



Further observe that : need not to start with smallest edge overall.



$U = \{v\}$ } every minimum cost edge must be include.
$W = V \setminus \{v\}$

So we can start from any node.

Complexity. Similar to Dijkstra's algorithm.

- Outer loop run n times::
  - In each iteration we add new ~~edge~~ vertex to tree
  - O(n) scan to find nearest vertex to add.
- Each time we add a vertex v, we have to scan all its neighbours to update distance
  - O(n) scan of adjacency matrix to find all neighbour. and update
- Overall $O(n^2)$ (adj matrixes)
  - If we move to adj list
    - O(m) scan to update

and if we have heap - $O((m+n)\log n)$

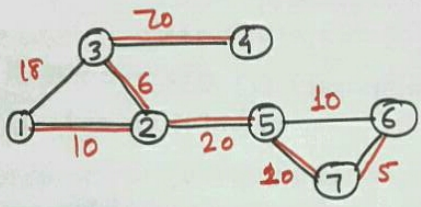
Prim's algorithm. (Simple)

```
#include <stdio.h>
#include <limits.h>
#define V 7
int min_edge ( int dist[], bool visited[])
{
    int min= INT_MAX ;
    int min_index;
    for (int i=0 ; i<V; ++i)
        { if (visited[i]==false && dist[i] < min)
            { min = dist[i];
              min_index=i;
            }
        }
    return (min_index);
}

int void printMST (int parent[], int n, int graph[V][V])
{ printf (" MST");
  for (int i=1; i<V; ++i )
     printf ("\n %d ---->--(%d) -->-- %d ",
                  parent[i]+1, graph[i][parent[i]], i+1);
}

void primTree ( int graph[][V])
  {   bool visited[V];
```

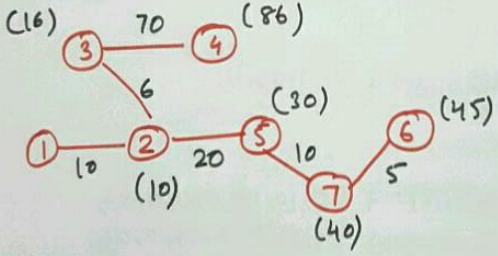
```
    int parent[V], dist[V];
    for(int i=0; i<V; ++i)
        { dist[i] = INT_MAX;
          visited[i] = false;
        }
    dist[0] = 0;
    parent[0] = -1;
    for (int i = 0; i<V-1; ++i)
        {
          int u = min_edge(dist, visited);
          visited[u] = true;
          for(int v=0; v<V; ++v)
            { if(graph[u][v] && !visited[u] &&
                        graph[u][v] < dist[v])
                { parent[v] = u;
                  dist[v] = graph[u][v];
                }
            }
        }
    print MST(parent, V, graph);
}
int main(){
    int graph[V][V] = { {0,10,18,0,0,0,0},
                        {10,0,6,0,20,0,0},
                        {18,6,0,70,0,0,0},
                        {0,0,70,0,0,0,0},
                        {0,20,0,0,0,10,10},
                        {0,0,0,0,10,0,5},
                        {0,0,0,0,10,5,0} };
    primsTree(graph); }
```

Let look's for another algorithm... i.e Krushkal's algorithm.

Spanning Tree. Krushkal's unlike Prim's donot form a tree from starting ...but it order the edges in the ascending order. Now if it can add edge and if that edge do not violate the property of the tree (mean it do not produce a cycle).



5, 6, 10, 10, 10, 18, 20, 70.

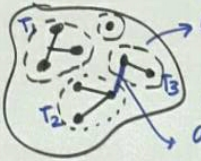


Correctness. Krushkal's algorithm is also a greedy algorithm. Unlike Dijkstra or Prim's where we make next choice based on what we have now.

Here in Krushkal's we actually say make choices at very advanced.

Minimum Separater lemma.

- At intermediate Unlike Prim's, at intermediate stages TE is not a tree.

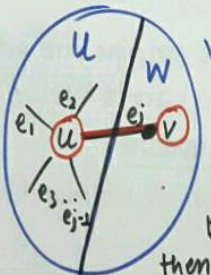


Important thing is when we add two (edge) vertex then then they do not belong to same component (because if that is tried it form a cycle). - that violate tree property.

- So when we join two end then they must be in different set or disjoint set. So in term of using "lemma" let call $e = (u, v)$

U = Component (u) $\quad$ W = V - Component (u)

- say this edge to $e_j (u, v)$



- No smaller weight edge in $(e_1, e_2, \ldots, e_{j-1})$

Connect $u - w$ component u to component w because if they are connected then we would not connect them. Connecting them would make a cycle.

- By minimum separator lemma, e_j must be in the minimum cost spanning tree - since it is smallest.

- To keep track of the property that it do not form a edge and if it does not form a cycle put it to

  Component [i] = i for each vertex i.

- e = (u, v) can be added if Component [u] is different from Component [v].

  Then • Merge two components (all which are connected to v (in component w) . so I have to scan all the vertices which are component of w or v.

Complexity.

- Initially sort edge - and sort m edges $O(m \log m)$
  - at most $m = n^2$, so this $O(m \log n)$
- Outer loop run upto m times
  - In each iteration, we examine one edge.
  - If we add edge, we have to merge component

Bottleneck → $O(n)$ scan to update.

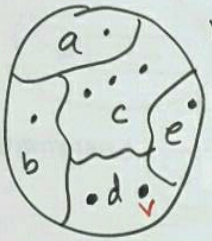
- This occur for n-1 times (so (n-1) updates)

- Overall $O(n^2)$

Bottleneck.

- Naïve strategy for labelling and merging component is inefficient.
- Just like we have think for heap in Prims algo and in Dijkstras.
- "Union- Find" data structure. implements the following operation efficiently.

↳ component structure.



Operation that I want

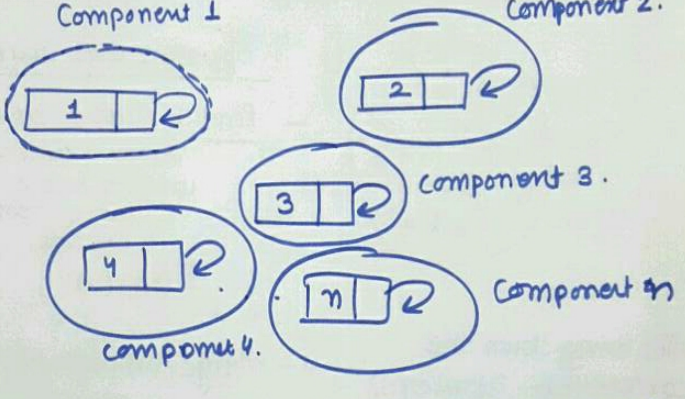
- find : given a name vertex v and you want to know which component it belongs.
  "find(v)"
- merge/union(u,v)
  - merge two component
  - so merge do not take component, it takes two representative like c,d.

- This bring down the complexity- O(mlogn)

So using efficient data structure or right selection of data structure is important to implement the algorithm efficiently. and Union-Find is one data structure which bring down the näive implementation ($O(m^2)$) for Krushkal's to $O((m+n)\log n)$.

way it implement :- 1) Think of array (a set of array)
2) Think of using pointer (more efficient than array).

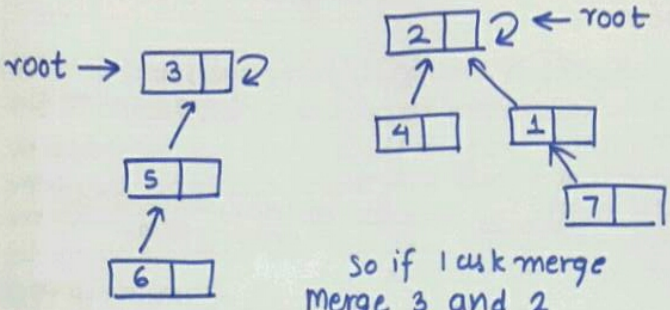
Make Union Find():



at initial only the node itself is in its component section, and second box is pointer with represent root. . $O(n)$

and when we have to merge two component we just have to modify pointer.

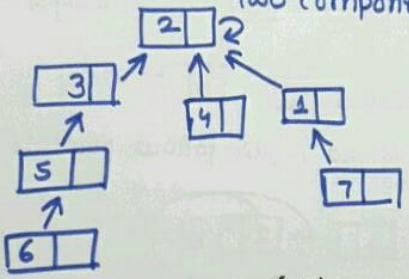
Time complexity of merge components:



So if I ask merge
Merge 3 and 2

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

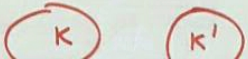
so if we modify pointer of 3 to point to 2 (which is just a O(1) opr) we merge two components.



: O(1)

Path Compression Technique and complexity of Find operation.

~ Each time we merge component, the size we are having is atleast twice the size of single component.

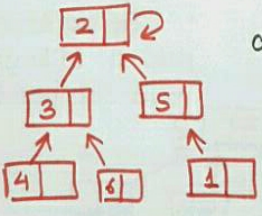


$K > K'$
merge K' to K

$K + K' \geq K' + K'$
New set $\geq 2K'$ (at least twice and even greater)

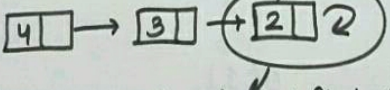
Now when we ask to find a node to which component it belong then in that case we have to find the root node 

Suppose in →



at some step we are having this stage.

Now if I ask find 4...so i just follow pointers.



and when i find this i immediately return 2 as the component which 4 belong

Now if you see we only scan the half from which we start. so In worst case finding :: $O(\log n)$ operation and for n opr. - $O(n \log n)$ time.

So what is Path Compression ?

~ This is something by which we want to bring down the complexity of Find to even lesser. So what we when we ask for a find(K) then each time we do not scan the whole path again but when we know that K belong to some K' then beside of again scanning we modify the pointer part of K to K'. directly. so after ~~f~~once we ~~find~~ apply find operation on all then the complexity -

$$O(4n \text{ alpha}(n)) \leq 4$$

$$\approx O(n\, \alpha(n)) \leq 4$$

$\alpha(n)$ - inverse of ackermann function ~ grows very slow

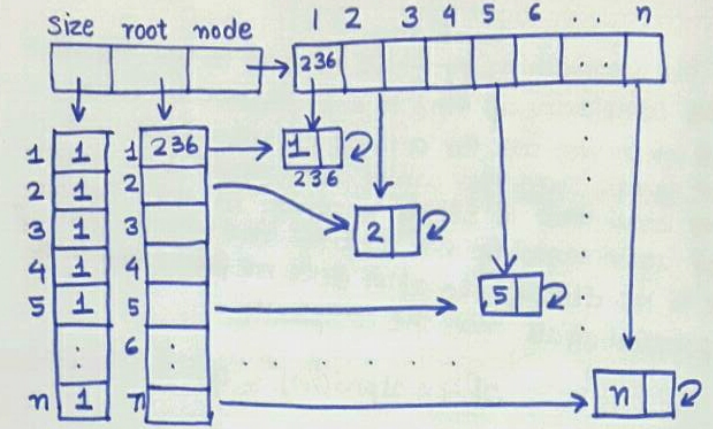
$\alpha(n) \leq 4$ (for any practical problem that we think) can

$$A(m,n) = \begin{cases} n+1 & \text{if } m=0 \\ A(m-1, 1) & \text{if } m>0 \text{ and } n=0 \\ A(m-1, A(m, n-1)) & \text{if } m>0, n>0 \end{cases}$$

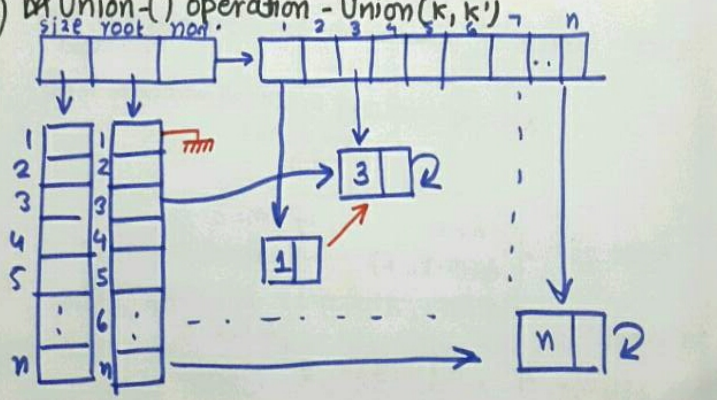
$A(1,2) = 4$, $A(4,3) = 2^{2^{65536}} - 3$

Let's look structure to feel it more...

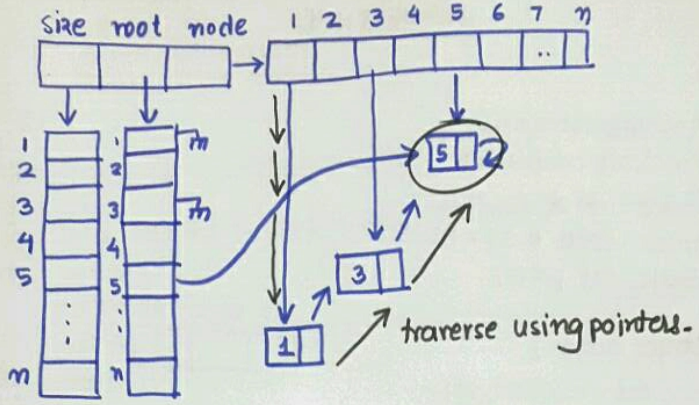
1) Make Union-Find ()



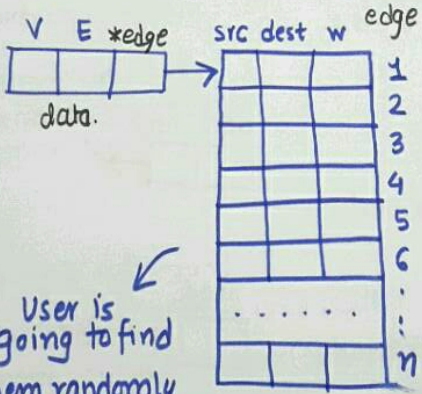
2) Union-() operation - Union(k, k')



3) Find( ) - Find(K). Uptrace the path e.g Find(1) 81



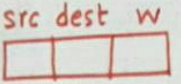
4) Structure design for graph information.

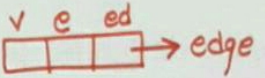


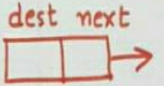
nd then using efficient algorithm we put them in ascending order. (quick sort).

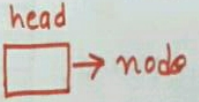
::: UNION-FIND IMPLEMENTATION OF KRUSHKAL ::: ALGORITHM.

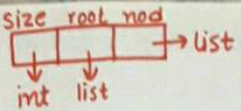
```
#include <stdio.h>
#include<stdlib.h>
struct uf *ds=NULL;
struct data *dt=NULL;
static int p=0;

struct edge {
    int src,dest,w;
};

struct data {
    int v,e;
    struct edge *ed;
};

struct node {
   int dest;
   struct node *next;
};
struct list {
   struct node *head;
};
```












```
struct uf {
    int * size ;
    struct list * root;
    struct list * nod ;
};
```

```
struct data *   cdata (int v, int e)
{   struct data *dt = (struct data*) malloc (size of (struct data));
    dt->v = v;
    dt->e = e;
    dt->ed =(struct edge *) malloc (e *sizeof (struct edge));
    return(dt);
}
void addedge ( int src, int dest, int w);
void node *   cnode (int d) {
{   struct node * temp = (struct node *)malloc (sizeof
                                        (struct node));
    temp->dest = d;
    temp->next = temp;
    return(temp);
}
```

```
struct uf *  cuf(int v) {
 struct uf * ds =(struct uf*) malloc(sizeof (struct uf));
  ds->size =(int *) malloc( v * sizeof(int)) ;
  ds-> root = (struct list*) malloc ( v *sizeof (struct list));
  ds-> node =(struct list *) malloc(v* sizeof( struct list));
```

```
    for(int i=0; i<v; ++i)
      { ds->size[i]= 1;
        ds-> root[i].head = ds-> nod[i]. head = cnode(i);
      }
   return (ds);
}

//Now quick sort.

void swap(int *a, int * b)
{   int t = *a;
    *a = *b;
    *b = t;
}

int partition ( int low, int high)
{ int pivot = dt->ed[high].w;
  int i= (low-1);
  for( int j = low; j<= high-1 ; ++j)
  {  if (dt->ed[j].w<= pivot)
     {  i++;
        swap( &dt->ed[i].w , &dt-> ed[j].w);
        swap (&dt->ed[i].src, &dt->ed[j].src);
        swap ( &dt->ed[i].dest, &dt-> ed[j].dest);
     }
  } swap( &dt->ed[i+1].w , &dt->ed [high].w);
   swap(&dt->ed[i+1].src, &dt->ed [high].src);
```

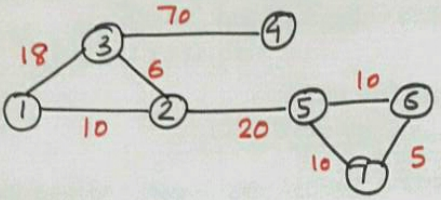
```
    swap(&dt->ed[i+1].dest, &dt->ed[high].dest);
    return(i+1);
}

void quicksort(struct edge * ar, int low, int high)
{
    if (low<high)
    {
        int pi= partition(low, high);
        quicksort(ar, low, pi-1);
        quicksort(ar, pi+1, high);
    }
}
void print();
int find(int k) {
    struct node *temp = ds->nod[k].head;
    while (temp->next!= temp)
    { temp= temp->next;
    }
    ds->node[k].head->next = temp;
    return(temp->dest);
}

void Union(struct uf *ds, int u, int v)
{
        int x= find(u);
        int y= find(v);
```

```
if (ds->size[x] < ds->size[y])
    { printf("%d ----> %d\n", u+1, v+1);
      struct node * temp = ds->node[x].head;
      temp->next = ds->nod[y].head;
      ds->root[x].head = NULL;
      ds->size[y] += ds->size[x];
      ds->size[x] = 0;
      print(); temp = NULL ; free(temp);
    }
  else {
      printf("%d ---> %d\n", u+1 , v+1);
       struct node * temp= ds->node[y].head;
       temp->next = ds->nod[x].head;
       ds->root[y].head = NULL;
       ds->size[x] += ds->size[y];
       ds->size[y]= 0;
       print();
    }
}
void Krushkals()
 {  int v= dt->v;
    int i=0;
    int e=0;
    quicksort(dt->ed, 0, dt->e-1);
```

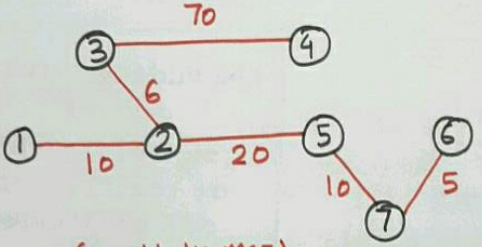
```
while (e< v-1)
 {
   struct edge dy = dt -> ed [i++];
   printf ("\n Call %d and %d \n",
                          dy.src+1, dy.dest+1);

   int x = find (dy.src);
   int y = find (dy.dest);
   if ( x != y)
     {
       Union (ds, dy.src, dy.dest);
       e++;
     }
 }
}
```

```
int main () {
   ds = cuf (7);
   dt = cdata (7, 8);
   addedge ( 2, 3, 70);
   addedge ( 4, 6, 10);
   addedge (5, 6, 5);
           (1, 2, 6);
           (1, 4, 20);
           (0, 1, 10);
           (4, 5, 10);
           (0, 2, 18);
  krushkal ();
}
```

```
void addedge (int src, int
        dest, w)
{ dt -> ed [p].dest = dest;
  dt -> ed [p].src = src;
  dt -> ed [p+1].w = w;
}
```



- { 5, 6, 10, 10, 10, 18, 20, 70 } → Quick Sort



(Kruskal's MST).

So using clever data structure like union-find can make Krushkal's algorithm more efficient.

Now we will also make Dijkstra's and Prims algorithm efficient - for that we need a data-structure called "priority queue."

"PRIORITY QUEUE"

Job Scheduler:- Jobs with highest priority are to be pick first.

Now obiously job comes dynamically (mean some job while running, new job arrive with new priority and its the job of scheduler to make sure new job with higher priority come ahead the job with lower priority.

Two basic operation:

- Need to maintain a list of jobs with priorities to optimise the following operations.
  - delete_max()
    - Identify and remove job with highest priority
    - Need not to be unique
  - insert()
    - Add a new job to the list.

So I can think to maintain Priority Queue: using

- Unsorted list
  - insert() – O(1)
  - delete_max() – O(n)

- Sorted list
  - insert() – O(n)
  - delete_max() – O(1)

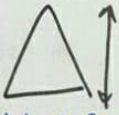
Processing sequence of n jobs – $O(n^2)$

So one dimensional structure is not efficient.

So lets goto 2D structure.

– also not efficient $O(N\sqrt{N})$

So efficient – Tree – Binary Tree – a balance Tree.



– n operation

$O(n \log_2 n)$

So we will look for
data structure – heap : Update value.

"Heaps"

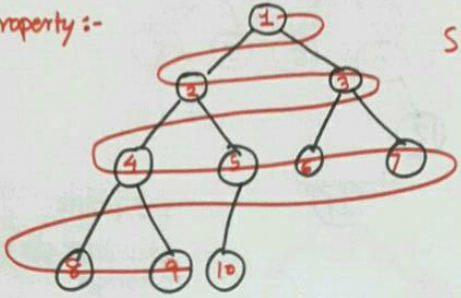
Heaps are tree implementation of priority queues

- insert and delete – O(LogN)
- heapify() build a heap – O(N) time
- Tree can be manipulated using array.

Heap : Heap is a "binary tree" which has very specific shape. where we add node level by level, left to right

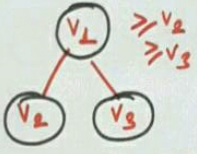
- At each node, value stored is bigger than both children

I property :-

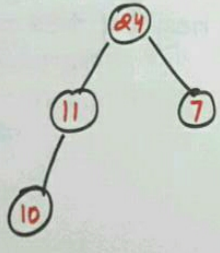


Shape is fixed for n node

II property

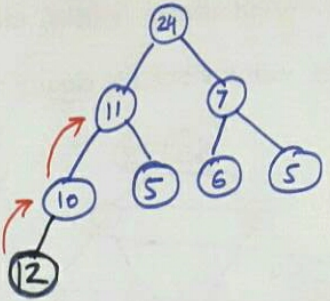


example : Every 4 node has same structure as shown

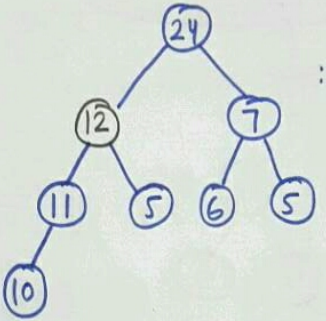


∴ Every leaf node satisfy heap property.

- insert on heap- Insert()
  - Insert 12.



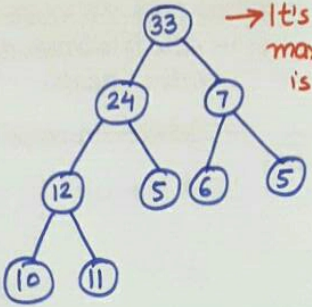
Final::



Imp point
:: Structure do not change

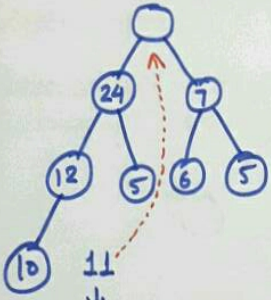
Complexity of insert() :- every time we add a node at the leaf and depending we might have to bow walk up to root. worst case - height of tree - n node
$H_n - (\log_2 n)$

delete_max():

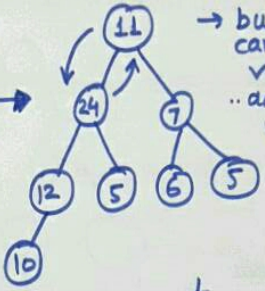


→ It's always that in max heap, maximum is always there.

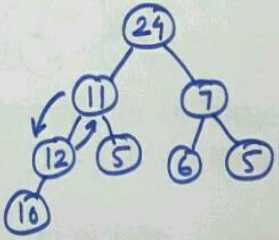
- Remove the value (not the node) it will leave a root with hole.
- Now reduce one node i.e last node.

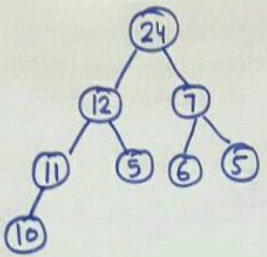


Why we use this:
because removing this will not violate heap property.



→ but there can be violation.. .. actualy violation will be always.

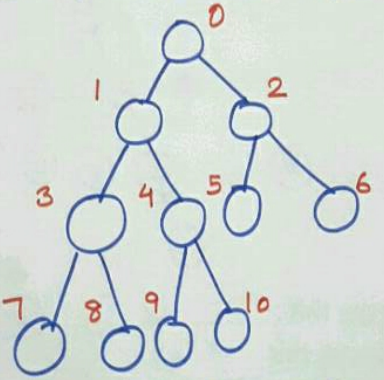




- now it is balanced and a valid heap.
- delete_max() - O(log n)

Implementation using array

Now another nice property about heap we do not need a very complex data structure like tree to implement heap ... but we can implement it using array



Represent as an array H[0 ... N-1]

- Children:- H[i] = H[2i+1], H[2i+2]
  H[1] → H[3], H[4]
- Parent:- H[j] is at H[floor(j - 1/2)] for j > 0
  take integer part - H[3] → H[1]

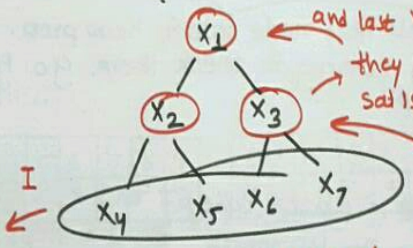
As we know that if we have index of array we can access element directly.

Build a heap : heapify().

- Given a set of value $[x_1, x_2, \ldots x_n]$, build a heap.
- Näive strategy
  - Start with empty heap
  - Insert each x_j
  - Overall $O(N \log N)$.

Better heapify.

- Because leaf node always satisfy heap property so they are already done.



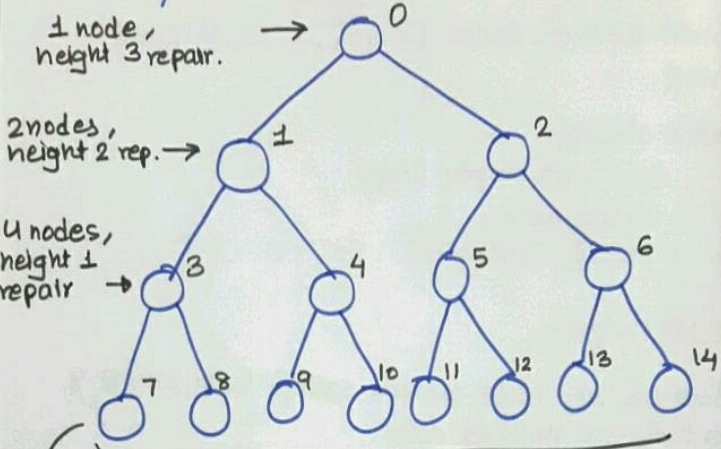
and last root node.

they may or may not satisfy but in worst case we have to 100/c for 1 step down to correct par

already satisfy heap property. so go one lerelup and

- And we do analysis, it come out that we can achiere this "O(N) time."

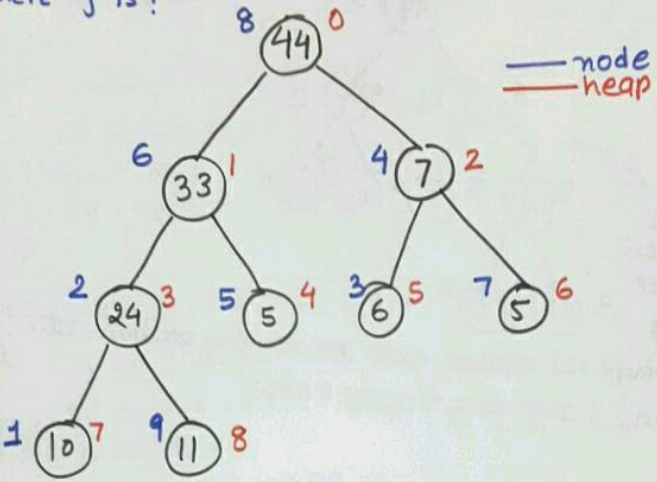
let's see by another example::



~ O(n) time :: (worst case) = 4×1 + 2×2 + 3×1
for 14 node ≥ 11 steps.
n ≥ no. of steps.

~ "smaller number have higher priority" when it comes to rank in a "competitive exam".

UPDATING VALUES In Dijkstra work - we take vertex j and say update its distance . . how to know where j is?



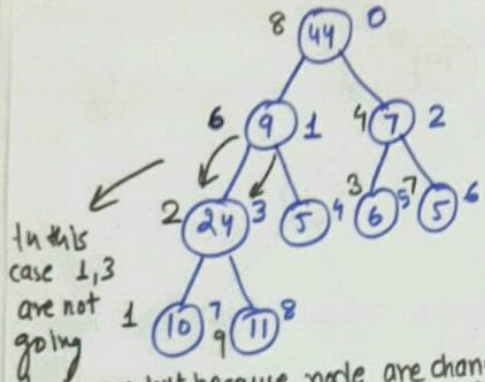
Node To Heap

1	2	3	4	5	6	7	8	9
7	3	6	2	5	1	6	p	8

Heap To Node.

0	1	2	3	4	5	6	7	8
8	6	4	2	5	3	7	1	9

Suppose if want to change our vertex 6 value in 98 graph to 9. then we have to modify



In this case 1,3 are not going to change. but because node are changing position, so... 2 and 6 are going to swap (imp)

NodeToHeap.

1	2	3	4	5	6	7	8	9
7	1	6	2	5	3	6	0	8

HeaptoNode

10	11	12	13	14	15	16	17	8
8	2	4	6	5	3	7	1	9

Implementation of Heap: → Implementing using array.

```
#include<stdio.h>
int n;
void swap (int *x, int *y)
{   int temp= *x;
    *x = *y;
    *y = temp;
}
void heapify ( int arr[], int i)
{
    int l= 2 *i +1;
    int r= 2 *i + 2;
    int small = i;
    if (l<n && arr[l] <arr[i])
        small = L;
    if (r<n && arr[r] < arr [small])
        small=r;

    if ( small != i)
        { swap (&arr[i], &arr [small]);
          heapify (arr, small);
        }
}
void reset (int arr[])
{ arr[0]= arr[n-1];
```

} O(log n).

```
    heapify(arr,0);
}

int min(int arr[])
{   int k=arr[0];
    reset(arr);
    n--;
    return k;
}
void print(int arr[7);
int main(){
    printf("Enter size of array :");
    scanf("%d", &n);
    int arr[n];
    for(int i=n-1; i>(n/2)-1; i--)
        { scanf("%d", &arr[i]); }
    for(int i=(n/2)-1; i>=0; i--)
        { scanf("%d", &arr[i]);
          heapify(arr,i);
        }
    int k=min(arr);
    print("%d", k);
    print(arr);
}
```

O(1)

O(n)

Divide and Conquer.

We have seen using divide and conquer alot of time provide a effective and efficient way to tackle problems. So what we do, we take our problem divide it to some subproblem and recursively solve them and return solved subproblem back to call placed.

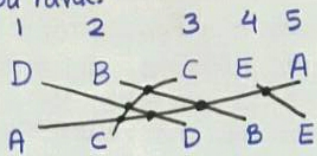
"RECOMMENDATION! SYSTEM"

You might have use alot of e-commerce website like Amazon, Flipkart .... Have you ever notice about the recommendation that appear on screen, when you look for some product. Saying people who bought this phone or this book or any thing also looked for this pair of headphone or nobels or any thing. You might amaze but most of things that you are "recommended" by that website. you actually like them. and may be willing to purchase them! Do you know how clever algorithm run behind the system to generate these recommendation. let Discuss how recommendation are provided.

~ Counting Inversion ~

- "Comparing Rank"
- Suppose If I provide you a list of movies and I asked to rank them depend upon ~~there~~ how you like them. same task I ask from your friend. and twice your and your friend rating.

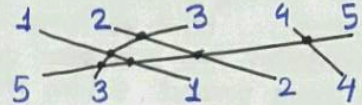
Suppose you rank.



- your rank
- your friend rank.

Inversion pair are :- no. of intersection pair i.e 6.

Let convert it more discrete data



∴ Inversion pair: (1,3), (1,5), (2,3), (3,5), (4,5) (2,5).

Rank = 6

Rank = 0 is perfect match with you... so I look for min rank pair so that i can recommend that to you.

- more formally rather than drawing line we can say any (i,j) pair is called inversion pair if

∵ $i<j$ and j appears before i in the list. then it is a inversion pair.

what any Inversion pair mean - for this pair the of movies the way you rank your friend rank exactly opposite to that. example:: (1,5) is an inversion pair mean you rank 1 before 5 and your friend do opposite.

- if you and your friend have rank same then there will be no inversion pair.

You — D B C E A
Friend — D B C E A

{ No intersection and also no such pair (i, j) such that $j > i$ and j appear before i.

~ So bruteforce implementation is like that you compare each (i, j) pair and for $n - \binom{n}{2}$ pairs so time $- O(n^2)$ which is high. actually very high.

Suppose if database of Amazon has - 1 billion user $\sim 10^8$ user. Then it take $O((10^8)^2)$ time.

if your system computer 10^9 operations/sec

then for $10^{16} - \frac{10^{16}}{10^9} = 10^7 \text{sec} = 115 \text{ days}$

- not a practical time to show you recommendation.

- we want something efficient ... and here comes divide and conquer strategy to solve this problem.

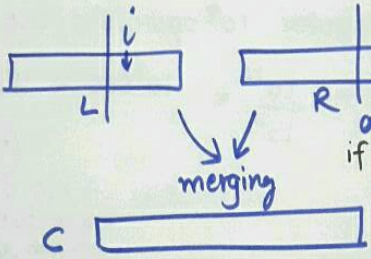
let look's for the algorithm.

- Consider your friends permutation $[i_1, i_2 \dots i_N]$
- Divide it to two parts (equal)

left → $[i_1, i_2 \dots i_{N/2}]$ right → $[i_{N/2+1} \dots i_N]$

Recursively count for this and right

end point for a single number there will be no permutation, so we can return zero. and also we do some thing clever in this - we are going to adapt divide and conquer, divide as we have seen and let see when we join them when we join them we also sort them. so focus.



and during merging:
if it is the case such that

$R[j] < L[i]$

then $R[j]$ is going to less than all item afterward i in the L so all pair are inversion if you see, R is right side and L is left

So any (i, j) such that $j \geq i$ and j appear before i is IP.

So number of inversion pair we will return from this particular step during merging:.

= + (size[L] - i) to the current count (→ inversion)

count = count + Lsize(L) - i)

- It also remove the problem when we are done with the element in L and we have to push remaining of R into C (as list are sorted so this must happen with any of L and R). but at that time nothing in L is greater than right so we run count still modify it but size(L) == i so we add nothing... Isn't amazing.. ok.

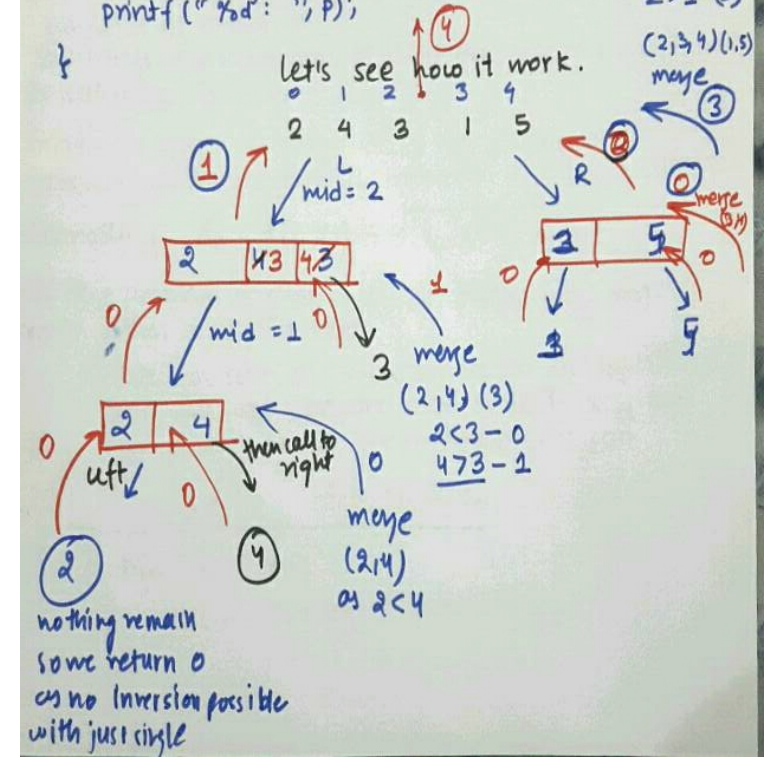
Code in C.

```
#include<stdio.h>
int merge(int arr[], int temp[], int left, int mid,
                                  int right)

{   int i, k, j, count = 0;
    i = left;
    j = mid;
    k = left; // for new array same index as i so
              // at filling back sorting to arr() it
              // become easy to iterate.

    while ((i <= mid-1) && (j <= right))

    {   if (arr[i] <= arr[j]) // Picking element from left.
                              then no problem.
```

```
        {  temp[k++] = arr[i++];
        }
      else{
            temp[k++] = arr[j++];
            count = count + (mid-i);
          }
    }
    while (i<= mid -1)   // when R get empty
        temp[k++] = arr[i++];
    while (j<= right)
        temp[k++] = arr[j++];

    // again put back to arr, so no need to return a
       new C from L and R merge.

    for(int i= left; i<=right; ++i)
          arr[i] = temp[i];

  return count;
}

int merge_count(int arr[],int temp[],int left, int right)
{ int mid,count=0;                                          )
  if(right > left)                                          ed
    { mid = (left+right)/2;
      count = merge_count(arr,temp, left,mid) - // L
      count += merge_count(arr, temp, mid+1, right) - // R
```

whatever count was earlier

```
        count += merge(arr, temp, left, mid+1, right);
    }
    return count;
}

int main(){
    int arr[5] = {2,4,3,1,5};
    int temp[5];
    int p = merge_count(arr, temp, 0, 4);
    printf("%d: ", p);
}
```



Closest pair of point...

Suppose if there are n points in a given plane and i ask you to compute the shortest pair point or in sense the pair of point with shortest distance.

Example :. Video Game. Suppose there are several objec on the screen. and the basic step is to find the closest pair of the object.

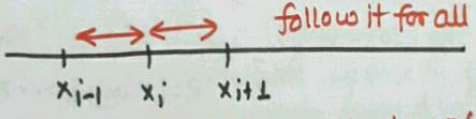
Now Näive implementation for this algorithm is $O(n^$ inwhich you just compute out distance btw all pair possible and for n points there can be $\binom{n}{2}$ pair which is of order n^2.

But for a very large value of n it may become inefficien to calculate distance.

There is a clever algorithm based on divide and conquer which compute it infact in $O(n \log_2 n)$ steps.

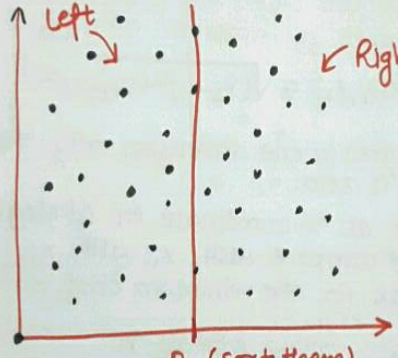
Formally.. $d_p = d(p_1, p_2) = \sqrt{(x_2-x_1)^2+(y_2-y_1)^2}$

- If the problem is given in one dimension only that mean either x or y is zero.
  - we can sort all x-coordinate in $O(n \log n)$ step and we just compare each x_i with x_{i-1} and x_{i+1} and look for the minimum dist pair.



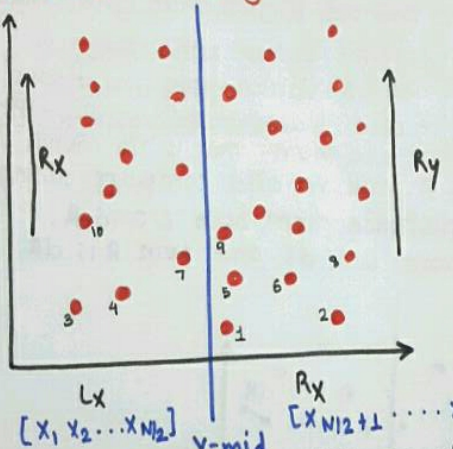
- So overall - $O(n \log_2 n + n) = O(n \log_2 n)$

But it is not as simple for the 2-Dimension. So we follow divide and conquer. So we split the set of points into two halves by vertical line. and we "recursively" compute closest pair in the two halves. and then we compare the closest pair from two halves and let's see how we can compute it effectively. So the vertical line that we have talk is not going to be any random line but it will be such that it divides two segment with equal number of points i.e $n/2$ and $n/2$. So simple way to do this is sort the point On the basis of there x coordinated and take the x[mid] to the line across which you can divide so that you got exact $n/2$ and $n/2$ left and right section.



So this much we have to done in $O(n \log_2 n)$ to sort and just a constant time $O(1)$ to compute middle element. in order to divide. Now because of recursive nature we again have to do this for left section and then to right section. we divide it until a case which is no more costlier to me to compute - Base case (say) $n = 3$ - so for three pair I just need $\left(\frac{3 \times 2}{2!}\right)$ $- {}^3C_2 - \frac{3!}{2!1!} = 3$ step - which we consider constant steps for each computation.

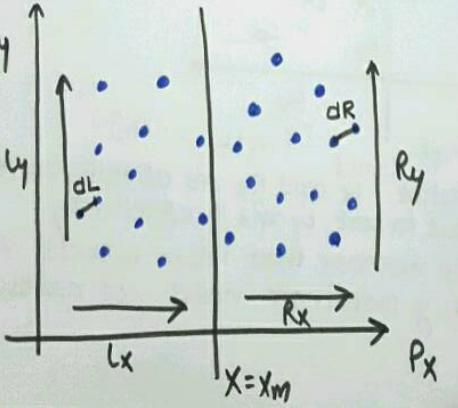
The important in this whole procedure is to compute the sorted L_x and R_x again let's see how to do that



So if you notice L_x and R_x are already sorted now we have to sort L_y and R_y efficiitrely

.So if you see the number that i have provide you can notice the y coordinate need not necessarily in pattern.

So they can appear like the way I have draw.
But imp point is that ... if you can notice is there are two set of y ... if with x coordinate less than x [mid] (letsay it x_m). and another. Point with x coordinate with $x > x_m$. Wow.. In saying these above two line we have find the solution to have sorted P_y and $R_y \rightarrow L_y$ consist of all y with $x < x_m$. and R_y consists of all y with $x > x_m$.
- and in one scan i.e O(n) scan of n point we can divide point in that order. Now after this step we can have the previous picture that I have drawed.
. Now we apply Recursive further split them till I find such L and R which have no. of point ≤ 3 and when it is the case I use BruteForce Algorithm to compute distance b/w them - that is no costier they are just steps ≤ 3. and we also compute along the shortest distance from both L and R.
say shortest from L = dL and from R is dR.



So you might say that the minimum of dL and dR is overall minimum distance pair... Isn't it... But wait we have check the smallest pair on leftside and also on separately right side too but have we check for any points that ~~go on~~ ~~fro~~ are on left and right side...No.....So we have to or we have also see for v points ~~b/w~~ going from distance b/w left to right . So is it that we again have to compute distance b/w each $n/2$ point with $n/2$ points on other side ?.. Well we can do some optimization in that because we do know that if i got a minimum pair from a point going from one side to another it should not be ~~ras~~ any bigger than the minimum