ICPC Handbook – 404 Team Not Found

1. STL

vector

```
vector<int> v(n, 0); //create vector of size n initialized to 0
a = new vector<int>(); //reset a
a.clear();
a = v; //copy v to a
auto it = v.rbegin(); //iterator to reverse beginning
*it; //value at iterator
int v.size();
void v.resize(n, 0); //resize to n and init to 0
bool v.empty();
reference v.front(); //get reference to front element
reference v.back();
v.swap(a); //swap vectors a & v
reverse(v.begin(),v.end()); //Reverse a vector
count(v.begin(), v.end(), val); //count occurences
find(v.begin(), v.end(), val); //iterator, O(n)
swap(v[a], v[b]);
rotate(v.begin(),v.begin()+3,v.end()); //v.begin() + 3 becomes first element after left rotate
random_shuffle(v.begin(), v.end());
bool cmp(int i,int j) { return i>j; }
sort(v.begin(), v.end(), cmp);
bool is_sorted(v.begin(), v.end());
lower_bound(v.begin(), v.end(), val); //1st occ
upper_bound(v.begin(), v.end(), val); //1st occ of next greater
bool binary_search (v.begin(), v.end(), val);
merge(a.begin(), a.end(), b.begin(), b.end(), v.begin()); //merge sorted
it = set_union(a.begin(), a.end(), b.begin(), b.end(), v.begin()); //it = end of union
it = set_intersection(a.begin(), a.end(), b.begin(), b.end(), v.begin());
it = set_difference(a.begin(), a.end(), b.begin(), b.end(), v.begin());
next_permutation(v.begin(), v.end());
prev_permutation(v.begin(), v.end());
```

string

Same as vector

```
s.substr(a, l); //substring from index a of length l
```

```
s.substr(a); //from index a to end
s.compare(b); //= 0 : equality
s.replace(a, l, b); //replace substring from index a of length l with b
s.replace(s.begin(), s.end()-3, b);
s += b; //append b to s
getline(cin, s); //Scan ans assign whole line to s
stoi(s); stoll(s); stof(s); stod(s); //string to int; lli; float; double
s = to_string(3.14); //Number to string
```

deque

Same as vector

```
deq.push_front(0); // constant
deq.pop_front(); // constant
```

queue

Same as vector

```
q.push(0);
q.pop();
```

stack & priority_queue

Same as vector

```
st.push(0);
st.pop();
st.top();
```

map

Same as vector

```
map<int, int> mp;
mp["name"] = "gaurav";
auto it = mp.find("a");
mp.find("a") != mymap.end(); //found
it->first; it->second; //key; value
```

heap

```
vector<int> v;
make_heap (v.begin(),v.end());
pop_heap(v.begin(),v.end());
v.pop_back();
v.push_back(val);
push_heap(v.begin(),v.end());
```

unordered_map is implemented as hash while map is implemented as bst

2. Number Theory

isprime

```
bool isprime(ll a) {
    ll i;
    if(a==2 || a==3)
    return true;
    if(a==1 || a==0)
    return false;
    for(i=2;i<=sqrt(a);i++) {
        if(isprime(i))
        if(a%i==0) return false;
    }
    return true;
}
```

GCD

```
int gcd(int a, int b) {
    if(!b) return a;
    return gcd(b, a%b);
}
```

Power

```
ll pw(ll a, ll b, ll M){
ll r;
if(b==0) return 1;
r = pw(a,b/2, M);
r = (r*r)%M;
if(b%2) r =
(r*a)%M;
return r;
}
```

prime factorization

```
// smallest prime factor of a number.
function factor(int n) {
   int a;
   if (n%2==0)
   return 2;
   for (a=3;a<=sqrt(n);a++++)  {
      if (n%a==0)
      return a;
   }
   return n;
}
// complete factorization
int r;
while (n>1) {
   r = factor(n);
   printf("%d", r);
   n /= r;
}
```

10-ary to m-ary

```
char a[16]={'0','1','2','3','4','5','6','7','8','9',
'A','B','C','D','E','F'};
string tenToM(int n, int m) {
   int temp=n;
   string result="";
   while (temp!=0)  {
      result=a[temp%m]+result;
      temp/=m;
   }
   return result;
}
```

M-ary to 10-ary

```
string num="0123456789ABCDE";
int mToTen(string n, int m) {
   int multi=1;
   int result=0;
   for (int i=n.size()-1;i>=0;i--) {
      result+=num.find(n[i])*multi;
      multi*=m;
```

Factorial Mod

```
ll fact(ll x) {
   if(x==1 || x==0) return 1;
   return ((x%M)*(fact(x-
1)%M))%M;
}
```

Combinations

```
ll comb(ll n,ll r) {
   if(n<r)
   return 0;
   return fact(n)/(fact(r)*fact(n-r));
}
```

Generate combinations

```
// n>=m, choose M numbers from 1 to
N.
void combination(int n, int m) {
   if (n<m) return ;
   int a[50]={0};
   int k=0;
   for (int i=1;i<=m;i++) a[i]=i;
   while (true) {
      for (int i=1;i<=m;i++)
         cout << a[i] << " ";
      cout << endl;
      k=m;
      while ((k>0) && (n-a[k]==m-k))
k--;
      if (k==0) break;
      a[k]++;
      for (int i=k+1;i<=m;i++)
         a[i]=a[i-1]+1;
   }
}
```

```
    }
    return result;
}
```

Segmented Sieve

```
ll a[100000];
ll sieve()
{
    ll i,j;
    for(i=2;i<100000;i++)
    a[i]=1;
    a[0]=0;
    a[1]=0;
    for(i=2;i<100000;i++)
    {
        if(a[i])
        {
            for(j=i*2;j<100000;j+=i)
            {
                a[j]=0;
            }
        }
    }
}
```

No. of Factors

```
ll phi (ll n) {
        ll result = n;
        for (ll i=2; i*i<=n; ++i)
                if (n % i == 0) {
                        while (n % i == 0)
                                n /= i;
                        result -= result / i;
                }
        if (n > 1) result -= result / n;
        return result;
}
```

```
ll segsieve(ll m,ll n)
{
    ll b[n-m+10],q,x,i,j;
    for(i=0;i<=n-m;i++)
    b[i]=1;
    if(m==1)
    b[0]=0;
    q=sqrt(n);
    for(i=2;i<=q;i++)
    {
        x=(m/i);
        x*=i;
        if(i>=x)
        x=i*2;
        for(j=x;j<=n;j+=i)
        {
          if(j>=m)
          b[j-m]=0;
        }
    }
    for(i=m;i<=n;i++)
    if(b[i-m])
    printf("%lld\n",i);
    printf("\n");
}
```

3. Basic Algos

Merge Sort

```
void merge(vector<ll> &a,ll lb,ll mid,ll ub) {
    vector<ll> c(ub-lb+1, -1);
    ll i=0,lp=lb,up=mid+1;
    while(lp<=mid&&up<=ub)
    {
        if(a[lp]<a[up]) c[i++]=a[lp++];
        else c[i++]=a[up++];
    }
    while(lp<=mid) c[i++]=a[lp++];
    while(up<=ub) c[i++]=a[up++];
    ll j;
    for(j=0;j<i;j++) a[j+lb]=c[j];
}
void mergesort(vector<ll> &a,ll lb,ll ub) {
    ll mid=(lb+ub)/2;
    if(lb<ub) {
        mergesort(a,lb,mid);
        mergesort(a,mid+1,ub);
        merge(a,lb,mid,ub);
    }
}
```

4. Trees

BST

```
struct node
{
    int data;
    node* left;
    node* right;
};
node* bstinsert(node* root, int value)
{
    struct node* temp = root;
    if(value<=temp->data){
        if(temp->left==NULL){
            struct node* new_node = new node();
            new_node->left = NULL;
```

Binary Search

```
int binarySearch(vector<int> a, int l,
int r, int val) {
    if(l>r) return -1;
    int m = (l + r)/2;
    if(a[m]==val) return m;
    if(a[m] > val) return
binarySearch(a, l, m-1, val);
    return binarySearch(a, m+1, r, val);
}
```

Heap

```
class MinHeap{
    int cap;
    int heapSize;
    int *arr;
    public  :
        MinHeap() {
        }
        void createHeap(int heapSize) {
            this->heapSize = 0;
            cap = heapSize;
            arr = new int[heapSize];
        }
        int parent(int i) {
            return (i-1)/2;
        }
        int leftChild(int i) {
            return 2*i + 1;
        }
        int rightChild(int i) {
            return 2*i + 2;
        }
        int getMin() {
            return arr[0];
        }
```

```
            new_node->right = NULL;
            new_node->data = value;
            temp->left = new_node;
        }
        else temp = bstinsert(temp->left, value);
    }else{
        if(temp->right==NULL){
            struct node* new_node = new node();
            new_node->left = NULL;
            new_node->right = NULL;
            new_node->data = value;
            temp->right = new_node;
        }
        else temp = bstinsert(temp->right, value);
    }
   return root;
}
node* bstsearch(node* root, int v)
{
    if(root->data==v)
        return temp;
    if(root->data>v)
        return bstsearch(root->left,v);
    return bstsearch(root->right, v);
}
struct node * minValueNode(struct node* node)
{
    struct node* current = node;
    while (current->left != NULL)
        current = current->left;
    return current;
}
struct node * maxValueNode(struct node* node)
{
    struct node* current = node;
    while (current->right != NULL)
        current = current->right;
    return current;
}
```

```
        void insertKey(int k) {
            int i;
            if (heapSize == cap) {
                cout << "Heap Overflow" <<
endl;
                return;
            }
            i = heapSize;
            heapSize++;
            arr[i] = k;
            while (i!=0 && arr[parent(i)] >
arr[i]) {
                int p = parent(i);
                swap(arr[i], arr[p]);
                i = parent(i);
            }
        }
        void minHeapify(int i) {
            int small = i;
            if (leftChild(i) < heapSize &&
arr[leftChild(i)] < arr[small]) {
                small = leftChild(i);
            }
            if (rightChild(i) < heapSize &&
arr[rightChild(i)] < arr[small]) {
                small = rightChild(i);
            }
            if (small != i) {
                swap(arr[i], arr[small]);
                minHeapify(small);
            }
        }
        int extractMin() {
            if (heapSize == 1) {
                heapSize--;
                return arr[0];
            }
            int root = arr[0];
            arr[0] = arr[heapSize-1];
            heapSize--;
            minHeapify(0);
            return root;
        }
};
```

```
void Predecessor(node* root, node*&pre, int v)
{
   if(root==NULL) return ;
   if(root->data==v)
   {
      if(root->left!=NULL) pre = maxValueNode(root->left);
   }
   else if(root->data<v)
   {
      pre = root;
      Predecessor(root->right, pre, v);
   }
   else Predecessor(root->left, pre, v);
}
void Successor(node* root, node*&suc, int v)
{
   if(root==NULL) return ;
   if(root->data==v)
   {
      if(root->right!=NULL)
          suc = minValueNode(root->right);
   }
   if(root->data>v)
   {
      suc = root;
      Successor(root->left, suc, v);
   }
   else Successor(root->right, suc, v);
}
node* bstdelete(node* root, int v)
{
   if(root==NULL)
      return root;
   if(root->data==v)
   {
      if(root->left==NULL)
      {
         struct node* temp=root->right;
         free(root);
```

```
            return temp;
        }
        else if(root->right==NULL)
        {
            struct node* temp=root->left;
            free(root);
            return temp;
        }
        struct node* temp = minValueNode(root->right);
        root->data=temp->data;
        root->right=bstdelete(root->right,temp->data);
    }
    if(root->data>v)
        return bstdelete(root->left,v);
    return bstdelete(root->right, v);
}
bool isBST(struct node* root)
{
    static struct node *prev = NULL;
    if (root)
    {
        if (!isBST(root->left))
          return false;
        if (prev != NULL && root->data <= prev->data)
          return false;
        prev = root;
        return isBST(root->right);
    }
    return true;
}
int isBSTUtil(struct node* node, int min, int max)
{
  if (node==NULL)
     return 1;
  if (node->data < min || node->data > max)
     return 0;
  return
    isBSTUtil(node->left, min, node->data-1)
&& isBSTUtil(node->right, node->data+1, max);
```

```
}
int isBSTmm(struct node* node)
{
  return(isBSTUtil(node, INT_MIN, INT_MAX));
}
node* lca(node* root, int v1,int v2)
{
    if(root==NULL) return NULL;
    if(root->data>max(v1,v2))
        return lca(root->left,v1,v2);
   else if(root->data<min(v1,v2))
       return lca(root->right, v1, v2);
   return root;
}
```

Prioirty Queue

```
class MinIndexedPQ {
    int NMAX, N, *heap, *index, *keys;

    void swap(int i, int j) {
        int t = heap[i]; heap[i] = heap[j]; heap[j] = t;
        index[heap[i]] = i; index[heap[j]] = j;
    }

    void bubbleUp(int k)   {
        while(k > 1 && keys[heap[k/2]] > keys[heap[k]])   {
            swap(k, k/2);
            k = k/2;
        }
    }

    void bubbleDown(int k)  {
        int j;
        while(2*k <= N) {
            j = 2*k;
            if(j < N && keys[heap[j]] > keys[heap[j+1]])
                j++;
            if(keys[heap[k]] <= keys[heap[j]])
                break;
            swap(k, j);
```

```
            k = j;
        }
    }

public:
    // Create an empty MinIndexedPQ which
can contain atmost NMAX elements
    MinIndexedPQ(int n)  {
        NMAX = n;
        N = 0;
        keys = new int[NMAX + 1];
        heap = new int[NMAX + 1];
        index = new int[NMAX + 1];
        for(int i = 0; i <= NMAX; i++)
            index[i] = -1;
    }

    // check if the PQ is empty
    bool isEmpty()  {
        return N == 0;
    }

    // check if i is an index on the PQ
    bool contains(int i)    {
        return index[i] != -1;
    }

    // return the number of elements in the PQ
    int size()  {
        return N;
    }

    // associate key with index i; 0 < i < NMAX
    void insert(int i, int key) {
        N++;
        index[i] = N;
        heap[N] = i;
        keys[i] = key;
        bubbleUp(N);
```

```
}

// returns the index associated with the minimal key
int minIndex()  {
   return heap[1];
}

// returns the minimal key
int minKey()   {
   return keys[heap[1]];
}

// delete the minimal key and return its associated index
// Warning: Don't try to read from this index after calling this function
int deleteMin() {
   int min = heap[1];
   swap(1, N--);
   bubbleDown(1);
   index[min] = -1;
   heap[N+1] = -1;
   return min;
}

// returns the key associated with index i
int keyOf(int i)   {
   return keys[i];
}

// change the key associated with index i to the specified value
void changeKey(int i, int key)  {
   keys[i] = key;
   bubbleUp(index[i]);
   bubbleDown(index[i]);
}

// decrease the key associated with index i to the specified value
void decreaseKey(int i, int key)   {
   keys[i] = key;
   bubbleUp(index[i]);
```

```
    }

    // increase the key associated with index i to the specified value
    void increaseKey(int i, int key)    {
        keys[i] = key;
        bubbleDown(index[i]);
    }

    // delete the key associated with index i
    void deleteKey(int i)   {
        int ind = index[i];
        swap(ind, N--);
        bubbleUp(ind);
        bubbleDown(ind);
        index[i] = -1;
    }
};
```

BIT

```
lli bit[1000];
// 1 based Bit Indexed Tree.
void init(lli n)
{
    for(lli i=1;i<=n;i++)
        bit[i]=0;
}
// to add val to arr_position 5, we add val to bit_position 5,6,8,16...
void add(lli pos,lli val,lli n)
{
    if(pos>n) return;
    bit[pos]+=val;
    add(pos+(pos&-pos),val,n);
}
// to find sum of values in arr upto 15, we add vals in bit at 15,14,12,8.
lli upto(lli pos)
{
    if(pos==0) return 0;
    return bit[pos]+upto(pos-(pos&-pos));
}
```

BIT (Point update range query)

```
int BIT[100000];
void initializeBIT(int a[],int n)
 {
    int i,j,k,l;
    for(i=0;i<=n;i++)
    {
            BIT[i]=0;
    }
    for(i=1;i<=n;i++)
    {
            int value_to_be_added=a[i-1];
            k=i;
            while(k<=n)
            {
            BIT[k]+=value_to_be_added;
            k+=(k&(-k));
            }
    }
 }
void update(int index,int value,int n)
 {
    int i,j,k;
    int index_to_modify=index+1;
    while(index_to_modify<=n)
    {
            BIT[index_to_modify]+=value;
            index_to_modify+=(index_to_modify&(-index_to_modify));
    }
 }
 int query(int i,int n)
 {
    int ans=0;
    int index_till=i+1;
    while(index_till>0)
    {
            ans+=BIT[index_till];
            index_till-=(index_till&(-index_till));
    }
    return ans;
```

```
}
```

Segment Tree

```
// A utility function to get the middle index from corner indexes.
int getMid(int s, int e) {  return s + (e -s)/2;  }

/*  A recursive function to get the sum of values in given range
   of the array. The following are parameters for this function.

   st    --> Pointer to segment tree
   si    --> Index of current node in the segment tree. Initially
             0 is passed as root is always at index 0
   ss & se  --> Starting and ending indexes of the segment represented
                by current node, i.e., st[si]
   qs & qe  --> Starting and ending indexes of query range */
int getSumUtil(int *st, int ss, int se, int qs, int qe, int si)
{
   // If segment of this node is a part of given range, then return
   // the sum of the segment
   if (qs <= ss && qe >= se)
      return st[si];

   // If segment of this node is outside the given range
   if (se < qs || ss > qe)
      return 0;

   // If a part of this segment overlaps with the given range
   int mid = getMid(ss, se);
   return getSumUtil(st, ss, mid, qs, qe, 2*si+1) +
          getSumUtil(st, mid+1, se, qs, qe, 2*si+2);
}

/* A recursive function to update the nodes which have the given
   index in their range. The following are parameters
   st, si, ss and se are same as getSumUtil()
   i    --> index of the element to be updated. This index is
            in input array.
   diff --> Value to be added to all nodes which have i in range */
void updateValueUtil(int *st, int ss, int se, int i, int diff, int si)
{
```

```
    // Base Case: If the input index lies outside the range of
    // this segment
    if (i < ss || i > se)
        return;

    // If the input index is in range of this node, then update
    // the value of the node and its children
    st[si] = st[si] + diff;
    if (se != ss)
    {
        int mid = getMid(ss, se);
        updateValueUtil(st, ss, mid, i, diff, 2*si + 1);
        updateValueUtil(st, mid+1, se, i, diff, 2*si + 2);
    }
}

// The function to update a value in input array and segment tree.
// It uses updateValueUtil() to update the value in segment tree
void updateValue(int arr[], int *st, int n, int i, int new_val)
{
    // Check for erroneous input index
    if (i < 0 || i > n-1)
    {
        printf("Invalid Input");
        return;
    }

    // Get the difference between new value and old value
    int diff = new_val - arr[i];

    // Update the value in array
    arr[i] = new_val;

    // Update the values of nodes in segment tree
    updateValueUtil(st, 0, n-1, i, diff, 0);
}

// Return sum of elements in range from index qs (quey start)
// to qe (query end).  It mainly uses getSumUtil()
```

```
int getSum(int *st, int n, int qs, int qe)
{
   // Check for erroneous input values
   if (qs < 0 || qe > n-1 || qs > qe)
   {
      printf("Invalid Input");
      return -1;
   }

   return getSumUtil(st, 0, n-1, qs, qe, 0);
}

// A recursive function that constructs Segment Tree for array[ss..se].
// si is index of current node in segment tree st
int constructSTUtil(int arr[], int ss, int se, int *st, int si)
{
   // If there is one element in array, store it in current node of
   // segment tree and return
   if (ss == se)
   {
      st[si] = arr[ss];
      return arr[ss];
   }

   // If there are more than one elements, then recur for left and
   // right subtrees and store the sum of values in this node
   int mid = getMid(ss, se);
   st[si] =  constructSTUtil(arr, ss, mid, st, si*2+1) +
             constructSTUtil(arr, mid+1, se, st, si*2+2);
   return st[si];
}

/* Function to construct segment tree from given array. This function
  allocates memory for segment tree and calls constructSTUtil() to
  fill the allocated memory */
int *constructST(int arr[], int n)
{
   // Allocate memory for segment tree
```

```
    //Height of segment tree
    int x = (int)(ceil(log2(n)));

    //Maximum size of segment tree
    int max_size = 2*(int)pow(2, x) - 1;

    // Allocate memory
    int *st = new int[max_size];

    // Fill the allocated memory st
    constructSTUtil(arr, 0, n-1, st, 0);

    // Return the constructed segment tree
    return st;
}
```

5. String

Suffix Array

```
int sa[50001],lcp[50001],p[25][50001],stp;
string s;
struct node
{
    int cm[2];
    int ind;
} N[50001],M[50001];


void myf( node u,node v )
{
    return (u.cm[0]==v.cm[0]) ?
(u.cm[1]<v.cm[1]) : (u.cm[0]<v.cm[0]);
}

void sorting(int k)
{
    int counti[50005]={0},flag;
    int i,j,l;
    for(i=0;i<k;i++)
```

KMP

```
void KMPSearch(char *pat, char *txt)
{
    int M = strlen(pat);
    int N = strlen(txt);
    int *lps = (int *)malloc(sizeof(int)*M);
    int j  = 0;  // index for pat[]
    int i = 0;  // index for txt[]
    while (i < N)
    {
        if (pat[j] == txt[i])
        {
            j++;
            i++;
        }
        if (j == M)
        {
        printf("Found at %d \n", i-j);
            j = lps[j-1];
        }
        else if (i < N && pat[j] != txt[i])
        {
```

```
    {
        M[i]=N[i];
        counti[M[i].cm[1]+1]+=1;
    }
    for(i=1;i<=50004;i++)
        counti[i]+=counti[i-1];
    for(i=k-1;i>=0;i--)
    {
        N[counti[M[i].cm[1]+1]-1]=M[i];
        counti[M[i].cm[1]+1] -= 1 ;
    }
    for(i=0;i<50005;i++)
        counti[i]=0;
    for(i=0;i<k;i++)
    {
        M[i]=N[i];
        counti[M[i].cm[0]+1]+=1;
    }
    for(i=1;i<=50004;i++)
        counti[i]+=counti[i-1];
    for(i=k-1;i>=0;i--)
    {
        N[counti[M[i].cm[0]+1]-1]=M[i];
        counti[M[i].cm[0]+1]-=1;
    }
}
void longest_common_prefix(int k)
{
    int i,j,l,m,x,y;
    int ans=0;
    lcp[0]=0;
    stp-=1;
    for(i=0,j=1;j<k;i+=1,j+=1)
    {
        l=stp;
        ans=0;
        x=sa[i];
        y=sa[j];
        while(l>=0&&x<k&&y<k)
                if (j != 0)
                        j = lps[j-1];
                else
                        i = i+1;
            }
        }
}

void computeLPSArray(char *pat, int M, int
*lps)
{
    int len = 0;
 // length of the previous longest prefix
suffix
    int i=1;
    lps[0] = 0;
    while (i < M)
    {
        if (pat[len] == pat[i])
        {
            len++;
            lps[i] = len;
            i++;
        }
        else if(len != 0)
        {
            len=lps[len-1];
        }
        else
        {
            lps[i] = 0;
            i++;
        }
    }
}
```

```
            {
                    if(p[l][x]==p[l][y])
                    {
                    ans+=1<<l;
                    x+=1<<l;
                    y+=1<<l;
                    }
                    l-=1;
            }
            lcp[j]=ans;
    }
}

void suffix_array(string s)
{
    int i,j,l,n,m,cnt;
    int k=s.length();
    for(i=0;i<k;i++)
    {
            p[0][i]=s[i];
    }
    for(cnt=1,stp=1;(cnt>>1)<k;cnt<<=1,stp++)
    {
            for(i=0;i<k;i++)
            {
                    N[i].cm[0]=p[stp-1][i];
                    N[i].cm[1]=(i+cnt)<k?p[stp-1][i+cnt]:-1;
                    N[i].ind=i;
            }
            sorting(k);
            //sort(N,N+k,myf);
            for(i=0;i<k;i++)
            {
                    p[stp][N[i].ind]=(i>0&&(N[i].cm[0]==N[i-1].cm[0])&&(N[i].cm[1]==N[i-1].cm[1])) ?
p[stp][N[i-1].ind] : i;
            }
    }
    for(i=0;i<k;i++)
            sa[p[stp][i]]=i;
```

```
}
```

6. Graph

```
class Graph
{
    int V;
    list<int> *adj;
    void DFSUtil
(int v, bool visited[]);
    bool isCyclicUtil
(int v, bool visited[], bool *rs);
    void TopologicalSortUtil
(int v, bool visited[], stack<int> &Stack);
public:
        Graph(int V);
        void addEdge(int v, int w);
        void BFS(int s);
        void DFS();
        bool isCyclic();
        void TopologicalSort();
};
Graph::Graph(int V)
{
    this->V = V;
    adj = new list<int>[V];
}
void Graph::addEdge(int v, int w)
{
    adj[v].push_back(w);
}
void Graph::BFS(int s)
{
    bool *visited = new bool[V];
    for(int i=0;i<V;i++)
        visited[i]=false;
    list<int> queue;
    visited[s] = true;
    queue.push_back(s);
    list<int>::iterator i;
```

Kruskal

```
void KruskalMST(struct Graph* graph)
{
    int V = graph->V;
    struct Edge result[V];
    int e = 0, i = 0;
    qsort(graph->edge, graph->E,
            sizeof(graph->edge[0]), myComp);
    struct subset *subsets =
    (struct subset*) malloc ( V * sizeof(struct subset) );

    for (int v = 0; v < V; ++v)

    {

        subsets[v].parent = v;

        subsets[v].rank = 0;

    }

    while (e < V - 1)

    {

        struct Edge next_edge = graph->edge[i++];

        int x = find(subsets, next_edge.src);

        int y = find(subsets, next_edge.dest);

        if (x != y)

        {

            result[e++] = next_edge;

            Union(subsets, x, y);

        }

    }

    printf("edges in MST\n");

    for (i = 0; i < e; ++i)

        printf("%d -- %d == %d\n",

                        result[i].src,
result[i].dest,result[i].weight);

    return;

}
```

```
    while(!queue.empty())
    {
        s = queue.front();
        cout << s << " ";
        queue.pop_front();
        for(i=adj[s].begin(); i!=adj[s].end();++i)
        {
            if(!visited[*i])
            {
                visited[*i]=true;
                queue.push_back(*i);
            }
        }
    }
}
void Graph::DFSUtil(int v, bool visited[])
{
    visited[v]=true;
    cout << v << " ";
    list<int>::iterator i;
    for(i=adj[v].begin();i!=adj[v].end();i++)
        if(visited[*i]==false)
            DFSUtil(*i, visited);
}
void Graph::DFS()
{
    bool *visited = new bool[V];
    int i;
    for(i=0;i<V;i++)
        visited[i]=false;
    for(i=0;i<V;i++)
        if(visited[i]==false)
            DFSUtil(i, visited);
}
bool Graph::isCyclicUtil
(int v, bool visited[], bool *recStack)
{
    if(visited[v] == false)
    {
```

Prim

```
void PrimMST(struct Graph* graph)

{

   int V = graph->V;

   int parent[V];

   int key[V];



   struct MinHeap* minHeap

         = createMinHeap(V);

   for (int v = 1; v < V; ++v)

   {

      parent[v] = -1;

      key[v] = INT_MAX;

      minHeap->array[v]

                  = newMinHeapNode(v, key[v]);

      minHeap->pos[v] = v;

   }

   key[0] = 0;

   minHeap->array[0] = newMinHeapNode(0, key[0]);

   minHeap->pos[0]   = 0;

   minHeap->size = V;

   while (!isEmpty(minHeap))

   {

      struct MinHeapNode* minHeapNode

                  = extractMin(minHeap);

      int u = minHeapNode->v;

      struct AdjListNode* pCrawl

                  = graph->array[u].head;
```

```
        visited[v] = true;
        recStack[v] = true;
        list<int>::iterator i;
        for(i = adj[v].begin(); i != adj[v].end(); ++i)
        {
            if (visited[*i]==false
&& isCyclicUtil(*i, visited, recStack) )
                return true;
            else if (recStack[*i])
                return true;
        }

    }
    recStack[v] = false;
    return false;
}
bool Graph::isCyclic()
{
    bool *visited = new bool[V];
    bool *recStack = new bool[V];
    for(int i = 0; i < V; i++)
    {
        visited[i] = false;
        recStack[i] = false;
    }
    for(int i = 0; i < V; i++)
        if (isCyclicUtil(i, visited, recStack))
            return true;
    return false;
}
void Graph::TopologicalSortUtil
(int v, bool visited[], stack<int> &Stack)
{
    visited[v]=true;
    list<int>::iterator i;
    for(i=adj[v].begin();i!=adj[v].end();i++)
        if(visited[*i]==false)
            TopologicalSortUtil(*i, visited, Stack);
    Stack.push(v);
```

```
    while (pCrawl != NULL)

    {

        int v = pCrawl->dest;

        if (isInMinHeap(minHeap, v)

                                && pCrawl->weight < key[v])

        {

            key[v] = pCrawl->weight;

            parent[v] = u;

            decreaseKey(minHeap, v, key[v]);

        }

        pCrawl = pCrawl->next;

    }

  }

  printArr(parent, V);

}
```

Bellman Ford

```
for ( i = 1 ; i < V ; i++ )
{
    for ( j = 0 ; j < E ; j++ )
    {
            int u = edge[j].src ;
            int v = edge[j].dest ;
            int weight = edge[j].weight ;
            if ( dist[u] != INT_MAX
            && dist[u] + weight < dist[v] )
                    dist[v] = dist[u] + weight ;
    }
}
```

```
}
void Graph::TopologicalSort()
{
   stack<int> Stack;
   bool *visited = new bool[V];
   int i;
   for(i=0;i<V;i++)
      visited[i]=false;
   for(i=0;i<V;i++)
      if(visited[i]==false)
         TopologicalSortUtil(i, visited, Stack);
   while(!Stack.empty())
   {
      cout << Stack.top() << " ";
      Stack.pop();
   }
}
```

Union Find

```
class UF
{
private:
   int *id;
public:
   UF(int n)
   {
      id=new int[n];
      for(int i=0;i<n;i++)
         id[i]=i;
   }
   int root(int i)
   {
      while(i!=id[i])
      {
         id[i]=id[id[i]];
         i=id[i];
      }
      return i;
   }
   bool iscon(int p, int q)
```

Floyd Warshall

```
void floydWarshell (int graph[][V])
{
   int dist[V][V], i, j, k;
   for (i = 0; i < V; i++)
      for (j = 0; j < V; j++)
         dist[i][j] = graph[i][j];
   for (k = 0; k < V; k++)
   {
      for (i = 0; i < V; i++)
      {
         for (j = 0; j < V; j++)
         {
            if (dist[i][k] + dist[k][j] < dist[i][j])
               dist[i][j] = dist[i][k] + dist[k][j];
         }
      }
   }
   printSolution(dist);
}
```

Connected Components

```
class CC
{
private:
   bool *marked;
   int *id;
   int cou=0;
   void dfs(graph G, int v)
   {
      marked[v]=true;
      id[v]=cou;
      for(int &w : G.con(v))
      {
         if(!marked[w])
            dfs(G,w);
      }
   }
```

```
   {
      return (root(p)==root(q));
   }
   void join(lli p, lli q)
 // join the two components (root to root)
   {
      id[root(p)]=id[root(q)];
   }
};
```

Dijkstra (V^2)

```
int minDistance(int dist[], bool sptSet[])
{
    int min = INT_MAX, min_index;
    for (int v = 0; v < V; v++)
            if (sptSet[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 sptSet[V] ;
    for ( int i = 0 ; i < V ; i++ ) {
            dist[i] = INT_MAX;
            sptSet[i] = false ;
    }
    dist[src] = 0 ;

    for ( int count = 0 ; count < V-1 ; count++ )
    {
            int u = minDistance (dist , sptSet ) ;
            sptSet[u] = true ;
            for (int v = 0; v < V; v++)
                    if ( ! sptSet[v] && graph[u][v] && dist[u] != INT_MAX && dist[u]+graph[u][v] <
dist[v] )
                            dist[v] = dist[u] + graph[u][v] ;
    }
}
```

```
public:
   CC(graph G)
   {
      marked = new bool[G.v()];
      id = new int[G.v()];
      for(int v=0;v<G.v();v++)
         marked[v]=false;
      for(int v=0;v<G.v();v++)
      {
         if(!marked[v])
         {
            dfs(G,v);
            cou++;
         }
      }
   }
   int total()
   {
      return cou;
   }
   int cid(int v)
   {
      return id[v];
   }
   bool connected(int v,int w)
   {
      return id[v]==id[w];
   }
};
```

Djikstra(ElogV)

```
typedef pair<LL,LL> PII;
int main()
{
long long N , s ;
    cin >> N >> s ;
    vector < vector < PII > > graph ( N ) ;
    for ( int i = 0 ; i < N ; i++ )
    {
            long long M ;
            cin >> M ;
            for ( int j =0 ; j < M ; j++ )
            {
            long long vertex , dist;
            cin >> vertex >> dist ;
            graph[i].push_back ( make_pair ( dist,vertex ) ) ;
            }
    }
    priority_queue < PII , vector<PII> , greater<PII> > Q;
    vector < long long > dist ( N , INF ) , dad ( N , -1 ) ;
    Q.push ( make_pair ( 0 , s ) ) ;
    dist [ s ] = 0 ;
    while ( ! Q.empty () )
    {
            PII p = Q.top() ;
            Q.pop() ;
            long long here = p.second ;
            for (auto it = graph[here].begin() ; it ! = graph[here].end() ; it++)
            {
                    if ( dist [ here ] + it -> first < dist [ it -> second ] )
                    {
                            dist [ it -> second ] = dist [ here ] + it -> first ;
                            dad [ it -> second ] = here ;
                            Q.push ( make_pair ( dist [ it -> second ],it -> second)) ;
                    }
            }
    }
    return 0;
}
```