### **Macros and Preprocessors**

#### Typedef

**typedef** vector<**int**> vi;

**typedef** vector<vi> vvi;

**typedef** pair<**int**,**int**> ii;

**typedef** vector<ii> vii;

**typedef** set<**int**> si;

#### #Define

#define lb lower\_bound

#define ub upper\_bound

#define pb push\_back

#define pf push\_front

#define mp make\_pair

#define ll long long

#define F(i,a,n) for(typeof(a) i=(a);i<(n);++i)

#define R(i,a,n) for(typeof(a) i=(a);i>=(n);--i)

#define tr(it,container) for(typeof(container.begin()) it = container.begin(); it != container.end(); ++it)

#define S(x) scanf("%d",&x)

#define M(x,i) memset(x,i,sizeof(x)) // useful to clear array of integer

#define fr first

#define se second

#define sz(a) ((int)a.size())

#define len(a) ((int)a.length())

#define INF 2147483647

#define all(x) x.begin(),x.end()

#define SORT(x) sort(all(x))

### **Linked List**

**struct** node {

**int** value;

node \*next;

};

#### // Print LinkedList

**void** display(node \*head) {

node \*tmp=head;

**while**(tmp != **NULL**) {

printf("%d->",tmp->value);

tmp=tmp->next;

}

printf("NULL\n");

}

#### // Add a Node at the beginning of LinkedList

node \*addatbeg(node \*head, **int** data) {

node \*tmp = (node \*)malloc(**sizeof**(node));

tmp->value=data;

tmp->next=head;

**return** tmp;

}

#### // Add a Node at the end of LinkedList

node \*addatend(node \*head, **int** data) {

node \*tmp = (node \*)malloc(**sizeof**(node));

tmp->value=data;

tmp->next=**NULL**;

**if** (head == **NULL**) {

head=tmp;

}

**else** {

node \*it=head;

**while**(it->next != **NULL**) {

it=it->next;

}

it->next=tmp;

}

**return** head;

}

#### // Count total nodes in LinkedList

**int** count(node \*head) {

**int** ct=0;

node \*tmp=head;

**while**(tmp != **NULL**) {

tmp=tmp->next;

ct++;

}

**return** ct;

}

#### // Search for an element in LinkedList

node \*search(node \*head, **int** data) {

node \*tmp = head;

**while**(tmp != **NULL**) {

**if** (tmp->value == data) **break**;

tmp = tmp->next;

}

**return** tmp;

}

#### // Delete an element in LinkedList

node \*del(node \*head,**int** data) {

node \*tmp = head;

**if** (tmp == **NULL**) **return** **NULL**;

**if**(tmp->value==data) {

head = tmp->next;

free(tmp);

**return** head;

}

**while**(tmp->next != **NULL** && tmp->next->value != data) {

tmp = tmp->next;

}

**if** (tmp->next==**NULL**) **return** head;

node \*it=tmp->next;

tmp->next=it->next;

free(it);

**return** head;

}

#### // Create a LinkedList

node \*createlist(node \*head) {

**int** n,x;

S(n);

F(i,0,n) {

S(x);

head=addatend(head,x);

}

**return** head;

}

#### // Reverse a LinkedList

node \*reverse (node \*head) {

node \*prev,\*ptr,\*next;

prev=**NULL**;

ptr=head;

**while**(ptr != **NULL**) {

next=ptr->next;

ptr->next=prev;

prev=ptr;

ptr=next;

}

**return** prev;

}

#### // Recursively Reverse a LinkedList

node \*reverser (node \*head) {

node \*temp;

**if** (head->next == **NULL**) {

**return** head;

}

temp = reverser(head->next);

head->next->next=head;

head->next=**NULL**;

**return** temp;

}

#### // Selection Sort on a LinkedList

**void** selection\_sort(node \*head) {

node \*p,\*q,\*min;

**int** temp;

**for**(p = head; p->next != **NULL**; p = p->next) {

min=p;

**for**(q = p->next; q != **NULL**; q = q->next) {

**if**(min->value > q->value) {

min=q;

}

}

**if**(min != p) {

temp=p->value;

p->value=min->value;

min->value=temp;

}

}

}

### **Doubly LinkedList**

**struct** node {

node \*prev;

**int** value;

node \*next;

};

#### // Display Doubly LinkedList

**void** display(node \*head) {

node \*ptr=head;

**while**(ptr!=**NULL**) {

printf("%d<->",ptr->value);

ptr=ptr->next;

}

printf("NULL\n");

}

#### // Add a Node at the beginning of Doubly LinkedList

node \*addatbeg(node \*head,**int** data) {

node \*tmp = **new** node;

tmp->value=data;

tmp->prev=**NULL**;

tmp->next=head;

**if** (head!=**NULL**) head->prev=tmp;

**return** tmp;

}

#### // Add a Node at the end of Doubly LinkedList

node \*addatend(node \*head, **int** data) {

node \*ptr=head;

node \*tmp = **new** node;

tmp->value=data;

tmp->prev=**NULL**;

tmp->next=**NULL**;

**if** (ptr==**NULL**) **return** tmp;

**while**(ptr->next!=**NULL**) ptr=ptr->next;

ptr->next=tmp;

tmp->prev=ptr;

**return** head;

}

#### // Delete an element in Doubly LinkedList

node \*del(node \*head,**int** data) {

**if**(head==**NULL**) **return** head;

node \*ptr=head;

node \*tmp;

**if** (ptr->value==data) {

tmp=ptr;

ptr=ptr->next;

**if**(ptr!=**NULL**) {

ptr->prev=**NULL**;

}

free(tmp);

**return** ptr;

}

**while**(ptr->next != **NULL** && ptr->next->value!=data) {

ptr=ptr->next;

}

**if**(ptr->next==**NULL**) **return** head;

tmp=ptr->next;

ptr->next=ptr->next->next;

**if** (tmp->next!=**NULL**)

tmp->next->prev=ptr;

free(tmp);

**return** head;

}

#### // Reverse a Doubly LinkedList

node \*reverse(node \*head) {

node \*ptr,\*next;

ptr=head;

**if** (ptr==**NULL** || ptr->next==**NULL**) **return** ptr;

next=ptr->next;

ptr->next=**NULL**;

ptr->prev=next;

**while**(next!=**NULL**) {

next->prev=next->next;

next->next=ptr;

ptr=next;

next=next->prev;

}

**return** ptr;

}

#### // Recursively Reverse a Doubly LinkedList

node \*reverser(node \*head) {

node \*tmp;

**if** (head->next==**NULL**) **return** head;

tmp=reverser(head->next);

head->next->prev=head->next->next;

head->next->next=head;

head->next=**NULL**;

**return** tmp;

}

### **Binary Tree**

**struct** node {

**int** value;

node \*lchild;

node \*rchild;

};

#### // Preorder

**void** preorder(node \*root) {

**if**(root == **NULL**) **return**;

printf("%d ", root->value);

preorder(root->lchild);

preorder(root->rchild);

}

#### // Postorder

**void** postorder(node \*root) {

**if**(root == **NULL**) **return**;

postorder(root->lchild);

postorder(root->rchild);

printf("%d ",root->value);

}

#### // Inorder

**void** inorder(node \*root) {

**if**(root == **NULL**) **return**;

inorder(root->lchild);

printf("%d ",root->value);

inorder(root->rchild);

}

#### // Iterative Preorder

///////// root-left-right

**void** nrec\_pre(node \*root) {

stack<node\*> st;

node \*cur;

**if**(root==**NULL**) **return**;

st.push(root);

**while**(!st.empty()) {

cur=st.top();st.pop();

printf("%d ",cur->value);

**if**(cur->rchild != **NULL**) {

st.push(cur->rchild);

}

**if**(cur->lchild != **NULL**) {

st.push(cur->lchild);

}

}

}

#### // Iterative Postorder

///////// left-right-root

////////////// can be implemented using preorder also = root-left-right change it to root-right-left and reverse it left-right-root

**void** nrec\_post(node \*root) {

stack<node\*> st;

node \*cur,\*q;

**if**(root==**NULL**) **return**;

cur=root;

q=**NULL**;

**while**(1) {

**while**(cur->lchild != **NULL**) {

st.push(cur);

cur=cur->lchild;

}

**while**(cur->rchild == **NULL** || cur->rchild == q) { // if left && right both child are null or right child is

printf("%d ", cur->value); // already covered.

q=cur;

**if**(st.empty()) **return**;

cur=st.top();st.pop();

}

st.push(cur);

cur = cur->rchild;

}

printf("\n");

}

#### // Iterative Inorder

///////// left-root-right

vector<**int**> nrec\_in(node \*root) {

vector<**int**> vector;

stack<node \*> stack;

node \*cur = root;

**while**(!stack.empty() || cur) {

**if**(cur) {

stack.push(cur);

cur = cur->left;

} **else** {

node \*curNode = stack.top();

vector.push\_back(curNode->val);

stack.pop();

cur = curNode->right;

}

}

**return** vector;

}

#### // Levelorder

**void** levelorder(node \*root) {

queue<node\*> q;

node \*cur;

**if**(root==**NULL**) **return**;

q.push(root);

**while**(!q.empty()) {

cur=q.front();q.pop();

printf("%d ",cur->value);

**if**(cur->lchild != **NULL**) q.push(cur->lchild);

**if**(cur->rchild != **NULL**) q.push(cur->rchild);

}

printf("\n");

}

#### // Build Tree from Inorder and Preorder

TreeNode \*buildTreeTmp(vector<**int**>::iterator prel, vector<**int**>::iterator prer, vector<**int**>::iterator inl, vector<**int**>::iterator inr){

**if** (prel >= prer) **return** **NULL**;

**int** val = \*prel;

TreeNode \*root = **new** TreeNode(val);

vector<**int**>::iterator rootIndex = find(inl, inr, val);

vector<**int**>::size\_type lsize = rootIndex - inl;

root->left = buildTreeTmp(prel + 1, prel + 1 + lsize, inl, rootIndex);

root->right = buildTreeTmp(prel + 1 + lsize, prer, rootIndex + 1, inr);

**return** root;

}

TreeNode \*buildTree(vector<**int**> &preorder, vector<**int**> &inorder) {

**return** preorder.size() == 0 ? **NULL** : buildTreeTmp(preorder.begin(), preorder.end(), inorder.begin(), inorder.end());

}

#### // Build Tree from Inorder and Postorder

TreeNode \*buildTreeTmp(vector<**int**>::iterator postl, vector<**int**>::iterator postr, vector<**int**>::iterator inl, vector<**int**>::iterator inr){

**if** (postl >= postr) **return** **NULL**;

**int** val = \*(postr-1);

TreeNode \*root = **new** TreeNode(val);

vector<**int**>::iterator rootIndex = find(inl, inr, val);

vector<**int**>::size\_type lsize = rootIndex - inl;

root->left = buildTreeTmp(postl, postl + lsize, inl, rootIndex);

root->right = buildTreeTmp(postl + lsize, --postr, rootIndex + 1, inr);

**return** root;

}

TreeNode \*buildTree(vector<**int**> &postorder, vector<**int**> &inorder) {

**return** postorder.size() == 0 ? **NULL** : buildTreeTmp(postorder.begin(), postorder.end(), inorder.begin(),

inorder.end());

}

#### // Height of binary tree

**int** height(node \*ptr) {

**int** hleft,hright;

**if**(ptr==**NULL**) **return** 0;

hleft=height(ptr->lchild);

hright=height(ptr->rchild);

**return** 1+max(hleft, hright);

}

#### // Search for an element in Binary Search Tree

node \*search(node \*ptr,**int** data) {

**while**(ptr != **NULL**) {

**if**(ptr->value==data) **return** ptr;

**else** **if**(ptr->value < data) ptr=ptr->rchild;

**else** ptr=ptr->rchild;

}

**return** **NULL**;

}

#### // Find minimum element in Binary Search Tree

node \*min(node \*ptr) {

**if**(ptr != **NULL**) {

**while**(ptr->lchild != **NULL**) ptr=ptr->lchild;

}

**return** ptr;

}

#### // Find maximum element in Binary Search Tree

node \*max(node \*ptr) {

**if**(ptr != **NULL**) {

**while**(ptr->rchild != **NULL**) ptr=ptr->rchild;

}

**return** ptr;

}

#### // Insert an element in Binary Search Tree - Iterative

node \*insert\_nrec(node \*root, **int** data) {

node \*tmp = **new** node;

node \*ptr=root,\*par=**NULL**;

tmp->value=data;

tmp->lchild=tmp->rchild=**NULL**;

**while**(ptr!=**NULL**) {

par=ptr;

**if** (ptr->value<data) ptr=ptr->lchild;

**else** **if**(ptr->value>data) ptr=ptr->rchild;

**else** **return** root;

}

**if**(par==**NULL**) root=tmp;

**else** **if**(par->value<data) par->rchild=tmp;

**else** par->lchild=tmp;

**return** root;

}

#### // Insert an element in Binary Search Tree - Recursive

node \*insert(node \*root, **int** data) {

**if**(root==**NULL**) {

node \*tmp = **new** node;

tmp->value=data;

tmp->lchild=tmp->rchild=**NULL**;

**return** tmp;

}

**else** **if**(root->value<data) {

root->rchild=insert(root->rchild,data);

}

**else** **if**(root->value>data) {

root->lchild=insert(root->lchild,data);

}

**return** root;

}

#### // Delete an element in Binary Search Tree - Iterative

node \*del\_nrec(node \*root,**int** data) {

node \*ptr,\*par=**NULL**,\*parsuc,\*suc,\*child;

ptr=root;

**while**(ptr != **NULL**) {

**if** (ptr->value==data) **break**;

par=ptr;

**if**(ptr->value<data)ptr=ptr->rchild;

**else** ptr=ptr->rchild;

}

**if**(ptr==**NULL**) **return** root;

**if**(ptr->lchild!=**NULL**&&ptr->rchild!=**NULL**) {

parsuc=ptr;

suc=ptr->rchild;

**while**(suc->lchild!=**NULL**) {

parsuc=suc;

suc=suc->lchild;

}

ptr->value=suc->value;

ptr=suc;

par=parsuc;

}

child=**NULL**;

**if**(ptr->lchild!=**NULL**) child=ptr->lchild;

**else** child=ptr->rchild;

**if**(par==**NULL**) root=child;

**else** **if**(par->lchild==ptr) par->lchild=child;

**else** par->rchild=child;

free(ptr);

**return** root;

}

#### // Delete an element in Binary Search Tree - Recursive

node \*del(node \*root,**int** data) {

**if**(root==**NULL**) **return** root;

**else** **if**(root->value<data) {

root->rchild=del(root->rchild,data);

}

**else** **if**(root->value>data) {

root->lchild=del(root->lchild,data);

}

**else** {

**if** (root->lchild != **NULL** && root->rchild != **NULL**) {

node \*suc=root->rchild;

**while**(suc->lchild!=**NULL**) {

suc=suc->lchild;

}

root->value=suc->value;

root->rchild=del(root->rchild,suc->value);

}

**else** {

node \*ptr=root;

**if**(root->lchild!=**NULL**) root=root->lchild;

**else** **if**(root->rchild!=**NULL**) root=root->rchild;

**else** root=**NULL**;

free(ptr);

}

}

**return** root;

}

#### // BST Iterator

TreeNode \* cur1, \* cur2;

stack<TreeNode \*> mys1, mys2;

BST(TreeNode \*root) {

cur1=cur2=root;

}

**bool** hasNext() {

**if**(cur1==**NULL** **and** mys1.empty())**return** **false**;

**return** **true**;

}

**bool** hasPrev() {

**if**(cur2==**NULL** **and** mys2.empty())**return** **false**;

**return** **true**;

}

**int** next() {

**if**(cur1!=**NULL**) {

mys1.push(cur1);

cur1=cur1->left;

next();

}

**else**{

**if**(**not** mys1.empty()) {

cur1=mys1.top();

**int** ans=cur1->val;

mys1.pop();

cur1=cur1->right;

**return** ans;

}

}

}

**int** prev() {

**if**(cur2!=**NULL**) {

mys2.push(cur2);

cur2=cur2->right;

prev();

}

**else**{

**if**(**not** mys2.empty()) {

cur2=mys2.top();

**int** ans=cur2->val;

mys2.pop();

cur2=cur2->left;

**return** ans;

}

}

}

### **AVL tree**

// AVL TREE => difference between height of left subtree and right subtree can not be greater than 1 for all node

**struct** node {

**int** value;

node \*lc;

node \*rc;

**int** height;

};

#### // Create a new node

node \*newnode(**int** key) {

node \*tmp=**new** node;

tmp->value=key;

tmp->lc=tmp->rc=**NULL**;

tmp->height=1; // new node is initially added at leaf

**return** tmp;

}

#### // Height of AVL tree

**int** height(node \*tmp) {

**if**(tmp==**NULL**) **return** 0;

**return** tmp->height;

}

#### // Balance of AVL tree node

**int** getbalance(node \*tmp) {

**return** (tmp==**NULL**) **?** 0 : height(tmp->lc) - height(tmp->rc);

}

#### // Right Rotation

node \*rotateright(node \*y) {

node \*x=y->lc;

y->lc=x->rc;

x->rc=y;

y->height=max(height(y->lc),height(y->rc))+1;

x->height=max(height(x->lc),height(x->rc))+1;

**return** x;

}

#### // Left Rotation

node \*rotateleft(node \*y) {

node \*x=y->rc;

y->rc=x->lc;

x->lc=y;

y->height=max(height(y->lc),height(y->rc))+1;

x->height=max(height(x->lc),height(x->rc))+1;

**return** x;

}

#### // Insert an element in AVL Tree

node \*insert(node \*root,**int** data) {

**if**(root==**NULL**) **return**(newnode(data));

**else** **if**(root->value < data) root->rc=insert(root->rc,data);

**else** **if**(root->value > data) root->lc=insert(root->lc,data);

**else** **return** root;

root->height=max(height(root->lc),height(root->rc))+1;

**int** balance=getbalance(root);

**if**(balance > 1 && root->lc->value > data) { // Left Left Case

**return** rotateright(root);

}

**if**(balance > 1 && root->lc->value < data) { // Left Right Case

root->lc=rotateleft(root->lc);

**return** rotateright(root);

}

**if**(balance < -1 && root->rc->value < data) { // Right Right Case

**return** rotateleft(root);

}

**if**(balance < -1 && root->rc->value > data) { // Right Left Case

root->rc=rotateright(root->rc);

**return** rotateleft(root);

}

**return** root;

}

#### // Delete an element in AVL Tree

node \*del(node \*root,**int** data) {

**if**(root==**NULL**) **return** root;

**else** **if**(root->value < data) root->rc=del(root->rc,data);

**else** **if**(root->value > data) root->lc=del(root->lc,data);

**else** {

**if** (root->lc != **NULL** && root->rc != **NULL**) {

node \*suc=root->rc;

**while**(suc->lc != **NULL**) suc=suc->lc;

root->value=suc->value;

root->rc=del(root->rc,suc->value);

}

**else** {

node \*ptr=root;

**if**(root->lc != **NULL**) root=root->lc;

**else** **if**(root->rc != **NULL**) root=root->rc;

**else** root=**NULL**;

free(ptr);

}

}

**if**(root==**NULL**) **return** **NULL**;

root->height = max(height(root->lc),height(root->rc))+1;

**int** balance = getbalance(root);

**if**(balance > 1 && getbalance(root->lc) >= 0) { // Left Left Case

**return** rotateright(root);

}

**else** **if**(balance > 1 && getbalance(root->lc) < 0) { // Left Right Case

root->lc=rotateleft(root->lc);

**return** rotateright(root);

}

**else** **if**(balance < -1 && getbalance(root->rc) <= 0) { // Right Right Case

**return** rotateleft(root);

}

**else** **if**(balance < -1 && getbalance(root->rc) > 0) { // Right Left Case

root->rc=rotateright(root->rc);

**return** rotateleft(root);

}

**return** root;

}

### **Sorting**

#### **Heap Sort**

**void** restoreup(**int** \*arr,**int** i);

**void** restoredown(**int** \*arr,**int** hsize,**int** i);

// HEAP

// arr should be from index 1

// arr[0] = INF

#### // Insert an element in heap

**void** insert(**int** \*arr,**int** \*hsize,**int** num) {

(\*hsize)++;

arr[\*hsize]=num;

restoreup(arr,\*hsize);

}

**void** restoreup(**int** \*arr,**int** i) {

**int** k=arr[i];

**int** par=i/2;

**while**(arr[par] < k) {

arr[i]=arr[par];

i=par;

par=i/2;

}

arr[i]=k;

}

#### // Delete an element from heap

**int** del(**int** \*arr,**int** \*hsize) {

**if**((\*hsize)==0) **return** -INF;

**int** max=arr[1];

arr[1]=arr[\*hsize];

(\*hsize)--;

restoredown(arr,\*hsize,1);

**return** max;

}

#### // Maintaining the heap property for subtree from node i

**void** restoredown(**int** \*arr,**int** hsize,**int** i) {

**int** lc=i<<1,rc=lc+1;

**int** num=arr[i];

**while**(rc<=hsize) {

**if** (num >= arr[lc] && num >= arr[rc]) {

arr[i]=num;

**return**;

}

**else** **if** (arr[lc] > arr[rc]) {

arr[i]=arr[lc];

i=lc;

}

**else** {

arr[i]=arr[rc];

i=rc;

}

lc=i<<1;

rc=lc+1;

}

**if**(lc==hsize&&arr[lc]>num) {

arr[i]=arr[lc];

i=lc;

}

arr[i]=num;

}

#### // Building Heap

**void** buildheap(**int** \*a, **int** size) {

R(i,size/2,1) {

restoredown(a,size,i);

}

}

#### // Heap Sort

**void** heap\_sort(**int** \*a, **int** size) {

**int** max;

buildheap(a,size);

**while**(size>1) {

max = del(a,&size);

a[size+1]=max;

}

}

#### **Merge Sort**

**void** merge(**int** \*a,**int** p,**int** q,**int** r);

**void** merge\_pass(**int** \*a,**int** size,**int** n);

#### // Merge Sort - Recursive

**void** merge\_sort(**int** \*a, **int** p, **int** r) { //top-down

**int** q;

**if**(p<r) {

q=(p+r)>>1;

merge\_sort(a,p,q);

merge\_sort(a,q+1,r);

merge(a,p,q,r);

}

}

**void** merge(**int** \*a,**int** p,**int** q,**int** r) {

**int** n1,n2,i,j;

n1=q-p+1;

n2=r-q;

**int** L[n1+1],R[n2+1];

F(i,0,n1) {

L[i]=a[p+i];

}

F(i,0,n2) {

R[i]=a[q+1+i];

}

L[n1]=R[n2]=INT\_MAX;

i=j=0;

F(k,p,r+1) {

**if** (L[i]<=R[j]) {

a[k]=L[i];

i++;

}

**else** {

a[k]=R[j];

j++;

}

}

}

#### // Merge Sort - Iterative

**void** iterative\_merge\_sort(**int** \*a, **int** n) { //bottom-up

**int** size=1;

**while**(size<n) {

merge\_pass(a,size,n);

size=size<<1;

}

}

**void** merge\_pass(**int** \*a,**int** size,**int** n) {

**int** p=0,q,r;

**while**(p+size<n) {

q=p+size-1;

r=p+2\*size-1;

**if**(r>=n) {

r=n-1;

}

merge(a,p,q,r);

p=r+1;

}

}

#### **Quick Sort**

**int** partition1(**int** \*a, **int** p, **int** r);

**int** randomized\_partition1(**int** \*a, **int** p, **int** r);

#### // Quick Sort - Recursive

**void** quick\_sort(**int** \*a, **int** p, **int** r) {

**if** (p < r) {

**int** q = partition1(a,p,r);

quick\_sort(a,p,q-1);

quick\_sort(a,q+1,r);

}

}

**int** partition1(**int** \*a, **int** p, **int** r) {

**int** pivot = a[r];

**int** i,j;

i=p;

**for**(j = p; j < r; j++) {

**if** (a[j] <= pivot) {

swap(a[i],a[j]);

i++;

}

}

swap(a[i],a[r]);

**return** i;

}

**int** randomized\_partition1(**int** \*a, **int** p, **int** r) {

**int** i = rand()%(r-p+1)+p;

swap(a[i],a[r]);

**return** partition1(a,p,r);

}

#### // Quick Sort - Iterative

**void** iterative\_quick\_sort(**int** \*arr,**int** l,**int** h)

{

// Create an auxiliary stack

**int** stack[h-l+1];

// initialize top of stack

**int** top = -1;

// push initial values of l and h to stack

stack[++top] = l;

stack[++top] = h;

// Keep popping from stack while is not empty

**while** (top >= 0) {

// Pop h and l

h = stack[top--];

l = stack[top--];

// Set pivot element at its correct position in sorted array

**int** p = partition1(arr, l, h);

// If there are elements on left side of pivot, then push left //more than 1 element

// side to stack

**if** ( p-1 > l ) {

stack[++top] = l;

stack[++top] = p - 1;

}

// If there are elements on right side of pivot, then push right //more than 1 element

// side to stack

**if** ( p+1 < h ) {

stack[++top] = p + 1;

stack[++top] = h;

}

}

}

#### **Counting Sort**

// only for elements in range 0-k where k <= 10000000 k is maximum element in array a.

**void** counting\_sort(**int** \*a, **int** \*b, **int** n, **int** k) {

**int** c[k+1];

M(c,0);

F(i,1,n+1) c[a[i]]++; //c[i] contains number of elements equal to i

F(i,1,k+1) c[i]+=c[i-1]; //c[i] contains number of elements less than or equal to i

R(i,n,1) {

b[c[a[i]]]=a[i];

c[a[i]]--;

}

}

#### **Radix Sort**

**void** radix\_sort(**int** \*a, **int** n, **int** k) {

**int** b[n+1];

**int** c[10];

**int** mod=10,div=1,num\_dig=0;

**while**(k) {

k/=10;

num\_dig++;

}

F(i,0,num\_dig) {

M(c,0);

F(j,1,n+1) c[(**int**)((a[j]%mod)/div)]++;

F(j,1,10) c[j]+=c[j-1];

R(j,n,1) {

b[c[(**int**)((a[j]%mod)/div)]]=a[j];

c[(**int**)((a[j]%mod)/div)]--;

}

F(j,1,n+1) a[j]=b[j];

div=mod;

mod=mod\*10;

}

}

#### **Selection Sort**

**void** selection\_sort(**int** \*a, **int** n) {

**int** min;

F(i,0,n-1) {

min=i;

F(j,i+1,n) {

**if**(a[min]>a[j]) {

min=j;

}

}

**if** (min!=i) {

swap(a[min],a[i]);

}

}

}

#### **Bubble Sort**

**void** bubble\_sort(**int** \*a, **int** n) {

**int** xchanges;

F(i,0,n-1) {

xchanges=0;

F(j,0,n-1-i) {

**if** (a[j]>a[j+1]) {

swap(a[j],a[j+1]);

xchanges++;

}

}

**if**(xchanges==0) **break**;

}

}

#### **Insertion Sort**

**void** insertion\_sort(**int** \*a,**int** n) {

**int** j,k;

F(i,1,n) {

k=a[i];

**for**(j = i-1; j >= 0 && k < a[j]; j--) {

a[j+1]=a[j];

}

a[j+1]=k;

}

}

### **KMP**

#### // Compute Prefix array

**void** prefix(string p, vector<**int**> &pre) {

**int** n = p.size();

**int** q = -1;

pre[0] = -1;

**for**(**int** i = 1; i < n; ++i) {

**while**(q >= 0 && p[q+1] != p[i]) q = pre[q];

**if**(p[q+1] == p[i]) q++;

pre[i] = q;

}

}

#### // KMP

**int** kmp(string tr, string p) {

**int** n = tr.size(), m = p.size();

vector<**int**> pre(m, 0);

prefix(p, pre);

**int** q = -1;

**for**(**int** i = 0; i < n; i++) {

**while**(q >= 0 && p[q+1] != tr[i]) q = pre[q];

**if**(p[q+1] == tr[i]) q++;

**if**(q+1 == m) {

**return** i;

}

}

**return** -1;

}

### **Sieve**

#### // Segmented Sieve

// finding primes between [ l....u ]

**void** segmented\_sieve(**int** l,**int** u) {

**int** i,j,d,sqlmt;

d=u-l+1;

**bool** flag[d]; //flag[i] shows l+i is prime or not

M(flag,0);

**for**(i = (l%2!=0); i < d; i+=2) flag[i]=1; //we can skip this line

sqlmt=sqrt(u);

**for**(i = 3; i <= sqlmt; i+=2) {

**if**(i > l && flag[i-l]) **continue**;

j = (l/i)\*i;

**if**(j < l) j += i;

**if**(j == i) j += i;

**for**(j=j-l;j<d;j+=i) flag[j]=1;

}

**if**(l<=1) flag[1-l]=1;

**if**(l<=2) flag[2-l]=0;

}

#### // Sieve

#define MAX 10050

#define LMT 105

//not removing multiples of 2 consider it urself

**bool** flag1[MAX];

ll prime[MAX];

**void** sieve1() {

ll i,j,k;

flag1[0]=flag1[1]=1;

**for**(i = 3; i < LMT; i+=2) {

k=i<<1;

**for**(j = i\*i; j < MAX; j+=k) {

flag1[j]=1;

}

}

prime[0]=2;

cout<<"2,";

j=1;

**for**(i = 3; i < MAX; i+=2)

**if**(!flag1[i]) {

cout<<i<<",";

prime[j++] = i;

}

}

#### // Sieve - Memory Optimized

**unsigned** flag[MAX/64];

**unsigned** primes[MAX/10];

**unsigned** primelen;

#define ifC(n) (flag[n>>6]&(1<<((n>>1)&31)))

#define isC(n) (flag[n>>6]|=(1<<((n>>1)&31)))

#define chk(n) (n==2||(n>2&&(n&1)&&!ifC(n))) //tells if a number is prime or not

**inline** **void** sieve() {

**register** **unsigned** i, j, k;

/\*

declare l

for(j = 9; j < MAX; j+=6) {

isC(j);

}

for(i = 5; i < LMT; i+=6) {

l=i+2;

if(!ifC(i)) {

for(j = i\*i, k=2\*i; j < MAX; j+=k) {

isC(j);

}

}

if(!ifC(l)) {

for(j = l\*l, k=2\*l; j < MAX; j+=k) {

isC(j);

}

}

}

\*/

**for**(i = 3; i < LMT; i+=2)

**if**(!ifC(i))

**for**(j = i\*i, k=2\*i; j < MAX; j+=k)

isC(j);

/\* primes[0]=2;

for(i = 3, j = 1; i < MAX; i+=2)

if(!ifC(i))

primes[j++] = i;

primelen = j;

\*/

primes[0]=2;

primes[1]=3;

**for**(k = 5, j = 1; k < MAX; k+=6) {

**if**(!ifC(k))

primes[++j] = k;

**if**(!ifC(k+2))

primes[++j] = k+2;

}

}

### **String Library**

#### // Split a String

vector<string> split(**const** string &s, **char** delim)

{

vector<string> elems;

stringstream ss(s);

string item;

**while** (getline(ss, item, delim))

elems.push\_back(item);

**return** elems;

}

**void** append\_zero(string &value1, string &value2) {

ll i,x,y;

x = sz(value1);

y = sz(value2);

**if** (x < y) {

F(i,x,y) value1 = "0" + value1;

}

**else** {

F(i,y,x) value2 = "0" + value2;

}

}

#### // Integer to String

string tostring(ll x)

{

string str;

**while**(x) {

str = (**char**)((x%10)+'0') + str;

x /= 10;

}

**return** str;

}

#### // Adding two Strings

string add(string &value1, string &value2) {

append\_zero(value1,value2);

string sum;

ll i,tmp = 0, x = sz(value1);

R(i,x-1,0) {

tmp += (value1[i]-'0')+(value2[i]-'0');

sum = (**char**)((tmp%10)+'0') + sum;

tmp /= 10;

}

**if** (tmp) {

sum = (**char**)(tmp+'0') + sum;

}

**return** sum;

}

#### // Subtracting two Strings

string subtract(string &value1, string &value2) {

ll x = sz(value1),y = sz(value2);

**bool** sign = 0;

**if** (x < y || (x == y && value1.compare(value2) < 0)) {

value1.swap(value2);

swap(x,y);

sign = 1;

}

append\_zero(value1,value2);

string sub;

ll i,diff,hold = 0;

x = sz(value1);

R(i,x-1,0) {

diff = (value1[i]-'0')-(value2[i]-'0') - hold;

**if** (diff < 0) {

hold = 1;

sub = **char**(diff+10+'0') + sub;

}

**else** {

sub = **char**(diff+'0') + sub;

hold = 0;

}

}

F(i,0,x) {

**if** (sub[i] != '0') **break**;

}

sub = sub.substr(i);

**if** (sign) sub = '-'+sub;

**return** sub;

}

#### // Multiplying two Strings

string multiply(string &value1, string &value2) {

string product;

ll x =sz(value1),y = sz(value2);

ll i,j,tmp,k = 0;

**for**(i = x-1; i >= 0; i--) {

string tmp\_product;

tmp = 0;

**for**(j = y-1; j >= 0; j--) {

tmp += (value1[i] - '0')\*(value2[j] - '0');

tmp\_product = (**char**)((tmp%10)+'0') + tmp\_product;

tmp /= 10;

}

**if** (tmp) {

tmp\_product = (**char**)(tmp+'0') + tmp\_product;

}

**for**(j = 0; j < k; j++) {

tmp\_product += '0';

}

product = add(product,tmp\_product);

k++;

}

**return** product;

}

#### // Multiply a String by Integer

string multiplyint(string a, ll b) {

string ans;

ll carry = 0,i,x = len(a);

**for**(i = x-1; i >= 0; i--) {

carry += (a[i]-'0')\*b;

ans = (**char**)((carry%10)+'0') + ans;

carry /= 10;

}

**while**(carry) {

ans = (**char**)((carry%10)+'0') + ans;

carry /= 10;

}

**return** ans;

}

#### // Divide a String by Integer

string divideint(string a, ll b) {

string ans;

ll i,sum=0,x = len(a);

F(i,0,x) {

sum = sum\*10+(a[i]-'0');

ans += (**char**)(sum/b+'0');

sum -= (sum/b)\*b;

}

x = sz(ans);

F(i,0,x) {

**if** (ans[i] != '0') **break**;

}

ans = ans.substr(i);

**if** (x == 0) ans = "0";

**return** ans;

}

### **Trie Tree**

#define ALPHABET\_SIZE 26

#define FREE(ptr) \

free(ptr); \

ptr=**NULL**

**struct** trie\_node {

**bool** value;

trie\_node \*child[ALPHABET\_SIZE];

};

**struct** trie {

**int** count;

trie\_node \*root;

};

#### // Create a new Trie Tree Node

trie\_node \*newtrie\_node() {

trie\_node \*tmp = **new** trie\_node;

**if**(tmp) {

tmp->value=0;

F(i,0,ALPHABET\_SIZE) {

tmp->child[i]=**NULL**;

}

}

**return** tmp;

}

**void** initialize\_trie(trie \*tmp) {

tmp->count=0;

tmp->root=newtrie\_node();

}

#### // Insert a String in Trie Tree

**void** insert(trie \*root,**char** \*key) {

**int** len=strlen(key);

root->count++;

trie\_node \*tmp=root->root;

F(i,0,len) {

**if**(!tmp->child[key[i]-'a']) {

tmp->child[key[i]-'a'] = newtrie\_node();

}

tmp=tmp->child[key[i]-'a'];

}

tmp->value=1;

}

#### // Search for a String in Trie Tree

**bool** search(trie \*ptree, **char** \*key) {

**int** len = strlen(key);

trie\_node \*tmp=ptree->root;

F(i,0,len) {

**if**(!tmp->child[key[i]-'a']) {

**return** 0;

}

tmp = tmp->child[key[i]-'a'];

}

**if**(tmp && tmp->value) **return** 1;

**return** 0;

}

**bool** isitfreenode(trie\_node \*tmp) {

F(i,0,ALPHABET\_SIZE) {

**if**(tmp->child[i]) **return** 0;

}

**return** 1;

}

**bool** delhelp(trie\_node \*tmp,**char** \*key,**int** level,**int** len) {

**if**(tmp) {

**if**(level == len) {

**if**(tmp->value) {

tmp->value=0;

**if**(isitfreenode(tmp)) {

**return** **true**;

}

**return** **false**;

}

}

**else** {

**int** idx=key[level]-'a';

**if**(delhelp(tmp->child[idx],key,level+1,len)) {

FREE(tmp->child[idx]);

**return** (isitfreenode(tmp)&&tmp->value==0);

}

}

}

**return** **false**;

}

#### // Delete a String in Trie Tree

**void** del(trie \*ptree, **char** \*key) {

**int** len=strlen(key);

**if**(len>0) {

delhelp(ptree->root,key,0,len);

}

}

### **Graph**

vector< vector<ii> > G;

vvi g; //graph

**int** n; //number of vertices

vector<**bool**> used(MAX);

**int** p[MAX],d[MAX];

**int** time\_in[MAX],time\_out[MAX];

**int** dfs\_timer;

**int** parent[MAX];

**bool** recstack[MAX];

vi topological;

stack<**int**> vertices;

**int** low[MAX]; //low[v] indicates earliest visited vertex reachable from subtree rooted with v

**bool** ap[MAX]; //if ap[i] is 1 then i is an articulation point otherwise not

**void** printg() {

F(i,0,n) {

cout<<"g["<<i<<"] => ";

tr(it,g[i]) cout<<\*it<<" ";

cout<<endl;

}

}

**void** printG() {

F(i,0,n) {

cout<<"G["<<i<<"] => ";

tr(it,G[i]) cout<<it->fr<<"("<<it->se<<") ";

cout<<endl;

}

}

**void** rmedge(**int** u,**int** v) {

tr(it,g[u]) {

**if**(\*it==v) {

\*it=-1;

**break**;

}

}

tr(it,g[v]) {

**if**(\*it==u) {

\*it=-1;

**break**;

}

}

}

**void** addedge(**int** u,**int** v) {

g[u].pb(v);

}

**int** dx[4]={0,0,-1,1};

**int** dy[4]={1,-1,0,0};

#### // BFS

//s is the source from where the bfs is started

//initialize used array and d array by 0 and p array by -1

**void** bfs(**int** s) {

F(i,0,n) {

used[i]=0;

p[i]=-1;

d[i]=0;

}

**int** v;

queue<**int**> q;

q.push(s);

used[s]=1;

p[s]=-1;

d[s]=0;

**while**(!q.empty()) {

v = q.front();q.pop();

tr(it,g[v]) {

**if**(!used[\*it]) {

q.push(\*it);

used[\*it]=1;

p[\*it]=v;

d[\*it]=d[v]+1;

}

}

}

}

**void** printpath(**int** s,**int** v) {

**if**(!used[v]) printf("No path\n");

vi path;

**while**(s!=v&&p[v]!=-1) {

path.pb(v);

v = p[v];

}

**if**(p[v]==-1) printf("no path from s to v");

**else** {

reverse(all(path));

tr(it,path) {

printf("%d--",\*it);

}

printf("\n");

}

}

#### // DFS

//initialise used array by 0

**void** dfsr1(**int** s) {

used[s]=1;

tr(it,g[s]) {

**if**(!used[\*it]) {

dfsr1(\*it);

}

}

}

**void** dfsr2(**int** s) {

**if**(!used[s]) {

cout<<s<<" ";

used[s]=1;

for\_each(all(g[s]),dfsr2);

}

}

**void** dfsrec(**int** s) {

time\_in[s]=dfs\_timer++;

used[s]=1;

tr(it,g[s]) {

**if**(!used[\*it]) {

dfsrec(\*it);

}

}

time\_out[s]=dfs\_timer++;

}

**void** dfs(**int** s) {

F(i,0,n) used[i]=0;

stack<**int**> st;

**int** v;

used[s]=1;

time\_in[s]=dfs\_timer++;

st.push(s);

**while**(!st.empty()) {

v=st.top();st.pop();

time\_out[v]=dfs\_timer++;

tr(it,g[v]) {

**if**(!used[\*it]) {

used[\*it]=1;

time\_in[\*it]=dfs\_timer++;

st.push(\*it);

}

}

}

}

#### // Check if Undirected Graph is Connected

**bool** chk\_undir\_graph\_conn() {

F(i,0,n) used[i]=0;

dfs(0); // bfs(0) use dfs or bfs

**return** (find(all(used),0)==used.end());

}

#### // Disjoint Set

**void** init\_set() {

F(i,0,n) {

parent[i]=i;

}

}

**int** findset(**int** i) {

**return** ((parent[i]==i) ? i : (parent[i]=findset(parent[i]))); //path compression

}

**void** unionset(**int** i,**int** j) {

parent[findset(i)]=findset(j);

}

**bool** join(u a,u b) {

a=findset(a);b=findset(b);

**if**(a==b) **return** 0;

**if**(size[b]>size[a]) swap(a,b);

parent[b]=a;

size[a]+=size[b];

**return** 1;

}

#### // Check for a cycle in Undirected Graph

//cycle => just check for back edge for undirected graph

**bool** iscyclicutil\_undir(**int** s,**int** parent) {

used[s]=1;

tr(it,g[s]) {

**if**(!used[\*it]) {

**if**(iscyclicutil\_undir(\*it,s)) **return** 1;

}

**else** **if**(\*it!=parent) **return** 1;

}

**return** 0;

}

**bool** iscyclic\_undir() {

F(i,0,n) used[i]=0;

F(i,0,n) {

**if**(!used[i] && iscyclicutil\_undir(i,-1)) {

**return** 1;

}

}

**return** 0;

}

#### // Check for a cycle in Directed Graph

**bool** iscyclicutil\_dir(**int** i) {

recstack[i]=1;

used[i]=1;

tr(it,g[i]) {

**if**(!used[\*it]) {

**if**(iscyclicutil\_dir(\*it)) **return** 1;

}

**else** **if**(recstack[\*it]) **return** 1;

}

recstack[i]=0;

**return** 0;

}

**bool** iscyclic\_dir() {

F(i,0,n) {

used[i]=recstack[i]=0;

}

F(i,0,n) {

**if**(!used[i]) {

**if**(iscyclicutil\_dir(i)) **return** 1;

}

}

**return** 0;

}

// Recursive function to find if there is back edge

// in DFS subtree tree rooted with 'u'

**bool** DFSUtil(**int** u, **int** color[])

{

// GRAY : This vertex is being processed (DFS

// for this vertex has started, but not

// ended (or this vertex is in function

// call stack)

color[u] = GRAY;

// Iterate through all adjacent vertices

list<**int**>::iterator i;

**for** (i = adj[u].begin(); i != adj[u].end(); ++i)

{

**int** v = \*i; // An adjacent of u

// If there is

**if** (color[v] == GRAY)

**return** **true**;

// If v is not processed and there is a back

// edge in subtree rooted with v

**if** (color[v] == WHITE && DFSUtil(v, color))

**return** **true**;

}

// Mark this vertex as processed

color[u] = BLACK;

**return** **false**;

}

// Returns true if there is a cycle in graph

**bool** isCyclic()

{

// Initialize color of all vertices as WHITE

**int** \*color = **new** **int**[V];

**for** (**int** i = 0; i < V; i++)

color[i] = WHITE;

// Do a DFS traversal beginning with all

// vertices

**for** (**int** i = 0; i < V; i++)

**if** (color[i] == WHITE)

**if** (DFSUtil(i, color) == **true**)

**return** **true**;

**return** **false**;

}

#### // Topological Sort

//topological sorting only for directed acyclic graph (DAG) there should be no backedge

**void** topologicalsortutil(**int** i) {

used[i]=1;

tr(it,g[i]) {

**if**(!used[\*it]) {

topologicalsortutil(\*it);

}

}

topological.pb(i);

}

**void** topologicalsort() {

F(i,0,n) used[i]=0;

F(i,0,n) {

**if**(!used[i]) {

topologicalsortutil(i);

}

}

reverse(all(topological));

}

#### // Minimum Spanning Tree (Kruskal)

//SPARSE GRAPH

**void** kruskal\_mst() {

priority\_queue<pair <**int**,ii>,vector<pair<**int**,ii> >,greater<pair<**int**,ii> > > Q; //fill priority\_queue Q.fr =

// weight of edge Q.se.fr and Q.se.se are the vertices

vector< pair< ii,**int** > > edges;

**int** count=0;

**int** x,y,wt,s1,s2;

init\_set();

**while**(!Q.empty() && count<n-1) {

x=Q.top().se.fr;

y=Q.top().se.se;

wt=Q.top().fr;

Q.pop();

//using union and findset

s1=findset(x);s2=findset(y);

**if**(s1!=s2) {

count++;

edges.pb(mp(mp(x,y),wt));

unionset(x,y);

}

}

}

#### // Minimum Spanning Tree (Prim)

//DENSE GRAPH

**void** prim\_mst() {

set< ii > s;

ii x;

F(i,0,n) {

used[i]=0;

p[i]=-1;

d[i]=INF;

}

d[0]=0; //find minimum spanning subtree from root node here root == 0

F(i,0,n) s.insert(mp(d[i],i));

**int** ct=n;

**while**(ct--) {

x=\*(s.begin());

s.erase(s.begin());

**if**(d[x.se]==INF) {

printf("Graph is not connected\n");

**return**;

}

used[x.se]=1;

tr(it,G[x.se]) {

**if**(!used[it->fr]) {

**if**(it->se<d[it->fr]) {

**if**(d[it->fr]!=INF) s.erase(mp(d[it->fr],it->fr));

d[it->fr]=it->se;

s.insert(mp(d[it->fr],it->fr));

p[it->fr]=x.se;

}

}

}

}

}

//applying idea of dijkstra via priority queue //faster than using set

**void** prim\_mst1() {

priority\_queue< ii,vii,greater<ii> > pq;

ii x;

F(i,0,n) {

used[i]=0;

p[i]=-1;

d[i]=INF;

}

d[0]=0; //find minimum spanning subtree from root node here root == 0

F(i,0,n) pq.push(mp(d[i],i));

**while**(!pq.empty()) {

x=pq.top();pq.pop();

**if**(d[x.se]==INF) {

printf("Graph is not connected\n");

**return**;

}

used[x.se]=1;

**if**(x.fr<=d[x.se]) { //concept (this check is very important we analyze each vertex only once the other occurence of it in queue (added earlier) will have greater distance

tr(it,G[x.se]) {

**if**(!used[it->fr]) {

**if**(it->se<d[it->fr]) {

d[it->fr]=it->se;

pq.push(mp(d[it->fr],it->fr));

p[it->fr]=x.se;

}

}

}

}

}

}

#### // Dijkstra (Priority Queue)

// Shortest path, only for non-negative weights, O(E(log(V)))

**void** dijkstra() {

F(i,0,n) d[i]=INF;

priority\_queue< ii,vii,greater<ii> > pq;

ii x;

**int** v,v2,cost;

d[0]=0; //find shortest distance from root node here root == 0

pq.push(mp(d[0],0));

**while**(!pq.empty()) {

x=pq.top();pq.pop();

v=x.se;

**if**(x.fr <= d[v]) {

tr(it,G[v]) {

v2=it->fr;

cost=it->se;

**if**(d[v2] > d[v]+cost) {

d[v2]=d[v]+cost;

pq.push(mp(d[v2],v2));

}

}

}

}

}

#### // Dijkstra (Set)

**void** dijkstra1() {

F(i,0,n) d[i]=INF;

set< ii > s;

ii x;

**int** v,v2,cost;

d[0]=0; //find shortest distance from root node here root == 0

s.insert(mp(d[0],0));

**while**(!s.empty()) {

x=\*(s.begin());

s.erase(s.begin());

v=x.se;

tr(it,G[v]) {

v2=it->fr;

cost=it->se;

**if**(d[v2] > d[v]+cost) {

**if**(d[v2]!=INF) s.erase(s.find(mp(d[v2],v2)));

d[v2]=d[v]+cost;

s.insert(mp(d[v2],v2));

}

}

}

}

#### // Floyd Warshall

//graph should not have negative cycles

**int** adj[MAX][MAX];

**int** D[MAX][MAX],pred[MAX][MAX]; // d[i][j] shortest path from i to j//pred[i][j] predecessor of j on shortest

// path from i to j

**void** floydwarshall() {

F(i,0,n) {

F(j,0,n) {

**if**(adj[i][j]) {

D[i][j]=adj[i][j];

pred[i][j]=i;

}

**else** {

D[i][j]=INF;

pred[i][j]=-1;

}

}

}

F(k,0,n) {

F(i,0,n) {

F(j,0,n) {

**if**(D[i][k]+D[k][j]<D[i][j]) {

D[i][j]=D[i][k]+D[k][j];

pred[i][j]=pred[k][j];

}

}

}

}

F(i,0,n) {

**if**(D[i][i]<0) {

printf("error negative cycle\n");

**break**;

}

}

}

**void** findpath(**int** s,**int** d) {

vi path;

**if**(D[s][d]==INF) {

printf("No path");

**return**;

}

**while**(d!=s){

path.pb(d);

d=pred[s][d];

}

path.pb(s);

reverse(all(path));

}

#### // Warshall

//(adj\_matrix)^k shows number of paths of path length k from i to j

**bool** path[MAX][MAX]; //path[i][j] shows is there a path from i to j

**void** warshall() {

F(i,0,n) {

F(j,0,n) {

**if**(adj[i][j]) path[i][j]=1;

**else** path[i][j]=0;

}

}

F(k,0,n) {

F(i,0,n) {

F(j,0,n) {

path[i][j]=path[i][j]||(path[i][k]&&path[k][j]);

}

}

}

}

#### // Bellman Ford

// O(V\*E), graph should not have negative cycles, There can be maximum |V| – 1 edges in any simple path

**void** bellmanford(**int** s) {

vector< pair<**int**,ii> > edge;

**int** u,v,wt,e; //number of edges

F(i,0,n) {

d[i]=INF;

}

d[s]=0;

//first

F(i,1,v) {

F(j,0,e) {

u=edge[j].se.fr;v=edge[j].se.se;

wt=edge[j].fr;

**if**(d[u]!=INF&&d[u]+wt<d[v]) {

d[v]=d[u]+wt;

}

}

}

//second

//if at some stage nothing has happened, then the algorithm can be stopped

/\*

for(;;) {

bool any=false;

F(j,0,e) {

u=edge[j].se.fr;v=edge[j].se.se;

wt=edge[j].fr;

if(d[u]!=INF&&d[u]+wt<d[v]) {

d[v]=d[u]+wt;

any=true;

}

}

if(!any) break;

}

\*/

//checking negative cycles

F(i,0,e) {

u=edge[i].se.fr;v=edge[i].se.se;

wt=edge[i].fr;

**if**(d[u]!=INF&&d[u]+wt<d[v]) {

printf("graph contains negative cycle\n");

**break**;

}

}

}

//shortest path in a directed acyclic graph O(V+E) linear using topological sorting

**void** shortestpathdag(**int** s) {

F(i,0,n) {

d[i]=INF;

}

d[s]=0;

topologicalsort();

F(j,0,n) {

i = topological[j];

**if**(d[i]!=INF) {

tr(it,G[i]) {

**if**(d[i]+it->se<d[it->fr]) {

d[it->fr]=d[i]+it->se;

}

}

}

}

}

#### // Kosaraju - SCC

//strongly connected => if there is a path from any two pair of vertices

//undirected BFS/DFS | directed Kosaraju

//kosaraju(strongly connected or not) => if every node can be reached from a vertex v, and every node can reach v, then the graph is strongly connected

**bool** kosaraju() {

dfs(0);

F(i,0,n) {

**if**(!used[i]) **return** 0;

}

//taking transpose

vi x;

x.resize(n);

F(i,0,n) {

tr(it,g[i]) {

x[\*it].pb(i);

}

}

g=x;

dfs(0);

F(i,0,n) {

**if**(!used[i]) **return** 0;

}

**return** 1;

}

//\*\*\*

//kosaraju prints the SCC in topological order AND tarjan prints the SCC in reverse topological order.

//to find all the SCC we have to start dfs from a sink vertex then move to other vertex which is sink in the remaining set

// if we do DFS of graph and store vertices according to their finish times,

// \*\*\*\*\*\*\*\*\*\* IMPORTANT LINES \*\*\*\*\*\*\*\*\*\*\*\*

//we make sure that the finish time of a vertex that connects to other SCCs (other that its own SCC),

//will always be greater than finish time of vertices in the other SCC

**void** fillorder(**int** s) {

used[s]=1;

tr(it,g[s]) {

**if**(!used[\*it]) {

fillorder(\*it);

}

}

vertices.push(s);

}

**void** printallSCC() {

F(i,0,n) used[i]=0;

F(i,0,n) {

**if**(!used[i]) {

fillorder(i);

}

}

//taking transpose

vvi x;

x.resize(n);

F(i,0,n) {

tr(it,g[i]) {

x[\*it].pb(i);

}

}

g=x;

F(i,0,n) used[i]=0;

**while**(!vertices.empty()) {

**if**(!used[vertices.top()]) {

dfsr2(vertices.top());

cout<<endl;

}

vertices.pop();

}

}

#### // Tarjan - (articulation points & bridges)

//articulation points (cut vertices) || bridge => vertex || edge removing which increases the number of connected components

//u is a parent of v , u and v are adjacent , v is discovered by u

//1.=> u is root of dfs tree and it has atleast two children

//2.=> u is not root and it has a child v such that no vertex in the subtree rooted with v has a back edge to one of the ancestors(in DFS tree) of u

//for every node we should have the earliest visited vertex (the vertex with minimum discovery time) that can be reached from the subtree rooted with that node

//low[v] indicates earliest visited vertex reachable from subtree rooted with v

//if ap[i] is 1 then i is an articulation point otherwise not

**void** tarjanutil(**int** u) {

**int** children=0; //don't make children and v global ???

used[u]=1;

low[u]=time\_in[u]=++dfs\_timer;

tr(it,g[u]) {

**int** v=\*it;

**if**(!used[v]) {

children++;

parent[v]=u;

tarjanutil(v);

//updating low value of u from its subtree

low[u]=min(low[u],low[v]);

//conditions for articulation point

**if**(parent[u]==-1&&children>1) ap[u]=1;

**if**(parent[u]!=-1&&(low[v]>=time\_in[u])) ap[u]=1;

//condition for bridge (lowest vertex reachable from subtree under v is below u in DFS

// tree) if(low[v]>time\_in[u]) then u-v is a bridge

}

**else** **if**(v!=parent[u]) {

//Update low value of u for parent function calls

low[u]=min(low[u],time\_in[v]);

}

}

}

**void** tarjan() {

dfs\_timer=0;

F(i,0,n) {

used[i]=0;

ap[i]=0;

parent[i]=-1;

}

F(i,0,n) {

**if**(!used[i]) {

tarjanutil(i);

}

}

}

#### // Tarjan - SCC

//to find all SCC we use => DFS search produces DFS tree | SCC form subtrees of the dfs tree

//if we can find head of such subtrees, store all nodes in that subtree || there is no back edge from one SCC to another

//a node is a head if time\_in[u]==low[u]

//SCC using tarjan => If there is a back edge or cross edge from every subtree to it's ancestor in DFS tree then the graph would be strongly connected.

**void** tarjansccutil(**int** u) {

time\_in[u]=low[u]=++dfs\_timer;

vertices.push(u);

recstack[u]=1;

tr(it,g[u]) {

**int** v = \*it;

**if**(time\_in[v]==-1) {

tarjansccutil(v);

low[u]=min(low[u],low[v]);

}

**else** **if**(recstack[v]) {

low[u]=min(low[u],time\_in[v]);

}

}

**int** w=0;

**if**(low[u]==time\_in[u]) {

**while**(vertices.top()!=u) {

w=vertices.top();

cout<<w<<" ";

recstack[w]=0;

vertices.pop();

}

w=vertices.top();

cout<<w<<"\n";

recstack[w]=0;

vertices.pop();

}

}

**void** tarjanscc() {

dfs\_timer=0;

F(i,0,n) {

time\_in[i]=-1;

low[i]=-1;

recstack[i]=0;

}

F(i,0,n) {

**if**(time\_in[i]==-1) {

tarjansccutil(i);

}

}

}

#### // Euler

//eulerian => visits each edge exactly once

//directed graph

//All vertices with nonzero degree belong to a single strongly connected component

//if in degree and out degree of every vertex is same than eulerian circuit exists

//if in degree and out degree of every vertex is same except 2 vertices and one of those 2 vertices has out-degree with one greater than in-degree (this is the start vertex), and the other vertex has in-degree with one greater than out-degree (this is the end vertex) than eulerian path exists

//undirected graph

//graph with non-zero degree should be connected

//if there are 0 odd vertices eulerian circuit iseulerian return 2

//if there are 2 odd vertices eulerian path iseulerian return 1

//one vertex with odd degree is not possible in an undirected graph (sum of all degrees is always even in an undirected graph)

**bool** isconnected() {

**int** s=-1;

F(i,0,n) {

**if**(sz(g[i])!=0) {

s=i;

**break**;

}

}

**if**(s==-1) **return** 1;

dfs(s);

F(i,0,n) {

**if**(sz(g[i])>0&&!used[i]) **return** 0;

}

**return** 1;

}

**int** iseulerian() {

**if**(!isconnected()) **return** 0;

**int** odd\_deg=0;

F(i,0,n) {

**if**(sz(g[i])&1) odd\_deg++;

}

**if**(odd\_deg>2) **return** 0;

**if**(odd\_deg) **return** 1;

**return** 2;

}

#### // Fleury

**int** dfsct(**int** s) {

**int** count=1;

used[s]=1;

tr(it,g[s]) {

**if**(\*it!=-1&&!used[\*it]) {

count+=dfsct(\*it);

}

}

**return** count;

}

//if number of reachable vertices (dfsct) are reduced after removing edge u-v, then it is a bridge

**bool** isvalidnextedge(**int** u,**int** v) {

**int** ct=0,ct1;

tr(it,g[u]) {

**if**(\*it!=-1) ct++;

}

**if**(ct==1) **return** 1;

ct=ct1=0;

F(i,0,n) used[i]=0;

ct=dfsct(u);

rmedge(u,v);

F(i,0,n) used[i]=0;

ct1=dfsct(u);

addedge(u,v);

//bridge condition

**if**(ct>ct1) **return** 0;

**return** 1;

}

**void** fleuryutil(**int** s) {

stack<**int**> st;

st.push(s);

**while**(!st.empty()) {

s=st.top();st.pop();

tr(it,g[s]) {

**int** v=\*it;

**if**(v!=-1&&isvalidnextedge(s,v)) {

printf("%d-%d\n",s,v);

rmedge(s,v);

st.push(v);

**break**;

}

}

}

}

//main idea => don't burn the bridges (so that we can come back to the vertex and traverse remaining edges)

**void** fleury() {

**int** s,x=iseulerian();

**if**(x==0) **return**;

**else** **if**(x==1) {

F(i,0,n) {

**if**(sz(g[i])&1) {

s=i;

**break**;

}

}

}

**else** {

F(i,0,n) {

**if**(sz(g[i])>0) {

s=i;

**break**;

}

}

}

fleuryutil(s);

}

### **BIT**

#### // 1D BIT

ll read(**int** idx){

ll sum = 0;

**while**(idx>0) {

sum += tree[idx];

idx -= (idx & -idx);

}

**return** sum;

}

**void** update(**int** idx, **int** val){

++idx;

**while**(idx <= n) {

tree[idx] += val;

idx += (idx & -idx);

}

}

#### // 2D BIT

**int** n;

**int** T[1026][1026];

**void** set1(**int** x, **int** y, **int** val) {

**int** i,j;

**for**(i = x; i <= n; i += (i&-i)) {

**for**(j = y; j <= n; j += (j&-j)) {

T[i][j] += val;

}

}

}

**int** sum(**int** x, **int** y) {

// if (x == 0 || y == 0) return 0;

**int** sum = 0;

**int** i,j;

**for**(i = x; i > 0; i -= (i&-i)) {

**for**(j = y; j > 0; j -= (j&-j)) {

sum += T[i][j];

}

}

**return** sum;

}

// Set matrix value a,b to c: mat[a][b] = c

// s1 = sum(a,b), s2 = sum(a-1,b-1), s3 = sum(a-1,b), s4 = sum(a,b-1) -- set1(a,b,c-(s1+s2-s3-s4))

// sum from a,b to c,d

// s1 = sum(c,d), s2 = sum(a-1,b-1), s3 = sum(a-1,d), s4 = sum(c,b-1) -- s1+s2-s3-s4

### **Matrix Exponentiation**

#define dim 3

**struct** matrix {

ll a[dim][dim];

};

matrix mul(matrix x, matrix y) {

matrix res;

F(a,0,dim) F(b,0,dim) res.a[a][b] = 0;

F(a,0,dim) F(b,0,dim) F(c,0,dim) {

res.a[a][b] += (x.a[a][c] \* (y.a[c][b]));

}

**return** res;

}

matrix power(matrix m, ll n) {

**if**(n == 1) **return** m;

matrix u = mul(m, m);

u = power(u, n/2);

**if**(n&1) u = mul(u, m);

**return** u;

}

### Segment Tree

//generally we have to store start and end for lazy segment tree

**struct** data {

// int start,end;

ll total;

**bool** pendingupdate;

};

data segtree[400020];

ll a[100005];

**void** assignleaf(data &node,ll value) {

node.total = value;

node.pendingupdate = 0;

}

**void** combine(data &result,data lc, data rc) {

result.total = lc.total + rc.total;

}

//build, query and update for single updates

#### // Building Segment Tree

//stindex=1,start=0,end=n-1

**void** build1(**int** stindex,**int** start,**int** end) {

**if**(start == end) {

assignleaf(segtree[stindex],a[start]);

}

**else** {

**int** lc,rc,mid = (start+end)/2;

lc = 2\*stindex;

rc = lc+1;

build1(lc,start,mid);

build1(rc,mid+1,end);

combine(segtree[stindex],segtree[lc],segtree[rc]);

}

}

#### // Getting value from left to right index

//stindex=1,start=0,end=n-1

data getvalue(**int** stindex,**int** start,**int** end,**int** left,**int** right) {

**if**(left==start && right==end) {

**return** segtree[stindex];

}

**int** mid = (start+end)/2;

**if**(right <= mid) **return** getvalue(2\*stindex,start,mid,left,right);

**else** **if**(left > mid) **return** getvalue(2\*stindex+1,mid+1,end,left,right);

data leftresult,rightresult,result;

leftresult = getvalue(2\*stindex,start,mid,left,mid);

rightresult = getvalue(2\*stindex+1,mid+1,end,mid+1,right);

combine(result,leftresult,rightresult);

**return** result;

}

#### // Update value of index to value

**void** update1(**int** stindex,**int** start,**int** end,**int** idx,ll new\_val) {

**if**(start==end) {

assignleaf(segtree[stindex],new\_val);

}

**else** {

**int** lc,rc,mid = (start+end)/2;

lc = 2\*stindex;

rc = lc+1;

**if**(idx <= mid) update1(lc,start,mid,idx,new\_val);

**else** update1(rc,mid+1,end,idx,new\_val);

combine(segtree[stindex],segtree[lc],segtree[rc]);

}

}