Starting template

```
/*-------property of the half blood prince-----*/

#include <bits/stdc++.h>
#define MIN(X,Y) X<Y?X:Y
#define MAX(X,Y) X>Y?X:Y
#define ISNUM(a) ('0'<=(a) && (a)<='9')
#define ISCAP(a) ('A'<=(a) && (a)<='Z')
#define ISSML(a) ('a'<=(a) && (a)<='z')
#define ISALP(a) (ISCAP(a) || ISSML(a))
#define MXX 10000000000
#define MNN -MXX
#define ISVALID(X,Y,N,M) ((X)>=1 && (X)<=(N) && (Y)>=1 && (Y)<=(M))
#define LLI long long int
#define VI vector<int>
#define VLLI vector<long long int>
#define MII map<int,int>
#define SI set<int>
#define PB push_back
#define MSI map<string,int>
#define PII pair<int,int>
#define PLLI pair<LLI,LLI>
#define FREP(i,I,N) for(int (i)=(I);(i)<=(N);(i)++)
#define eps 0.0000000001
#define RFREP(i,N,I) for(int (i)=(N);(i)>=(I);(i)--)
#define SORTV(VEC) sort(VEC.begin(),VEC.end())
#define SORTVCMP(VEC,cmp) sort(VEC.begin(),VEC.end(),cmp)
#define REVV(VEC) reverse(VEC.begin(),VEC.end())
using namespace std;
```

Data Structures

Segment tree (with lazy)

```
typedef pair<int,int> rng;

int seg[3000005];
vector<rng>post;

void init(int b, int e, int node, int K){
    if(b==e){
        seg[node]=K;
        return;
    }
    if(b>e)return;
    else{
  //  printf("in node %d, range %d to %d\n",node,b,e);
        int lt=node<<1;
        int rt=lt+1;
        int mid=(b+e)>>1;
        init(b,mid,lt,K);
        init(mid+1,e,rt,K);
       // seg[node].tot=seg[lt].tot+seg[rt].tot;
        seg[node]=K;
        return;
    }
}

void propagatebelow(int node,int b, int e){
```

```
    if(b<e){
        int ll=node<<1;
        int rr=ll+1;
        seg[ll]|=seg[node];
        seg[rr]|=seg[node];
    }
}

int update(int b,int e, int node, int load){
    if(b>e)return -1003;
    //propagatebelow(node,b,e);
    if(seg[node]<load)return -1003;
    if(b==e){
        seg[node]-=load;
        //propagatebelow(node,b,e);
        return b;
    }
    //if(b1>e1 || b2>e2)return;
    int lt=node<<1;
    int rt=lt+1;
    int mid=(b+e)>>1;
    int ans=update(b,mid,lt,load);
    if(ans<0)ans=update(mid+1,e,rt,load);
    seg[node]=max(seg[lt],seg[rt]);
    return ans;
}

int query(int i, int j, int b, int e, int node){
    if(b>e)return 0;
    propagatebelow(node,b,e);
    if(b>=i && e<=j){
        int mul=0;
        if(seg[node])mul=1;
        return (e-b+1)*mul;
    }
    if(e<i || b>j){
        return 0;
    }
    //int prop=seg[node].lazyval;
    int lt=node<<1;
    int rt=lt+1;
    int mid=(b+e)>>1;
    int lans=query(i,j,b,mid,lt);
    int rans=query(i,j,mid+1,e,rt);
    return lans+rans;
}
```

Femwick Tree

```
#include <iostream>
using namespace std;

/*            n  --> No. of elements present in input array.
    BITree[0..n] --> Array that represents Binary Indexed Tree.
    arr[0..n-1]  --> Input array for whic prefix sum is evaluated. */

// Returns sum of arr[0..index]. This function assumes
// that the array is preprocessed and partial sums of
// array elements are stored in BITree[].
int getSum(int BITree[], int index)
```

```
{
    int sum = 0; // Iniialize result

    // index in BITree[] is 1 more than the index in arr[]
    index = index + 1;

    // Traverse ancestors of BITree[index]
    while (index>0)
    {
        // Add current element of BITree to sum
        sum += BITree[index];

        // Move index to parent node in getSum View
        index -= index & (-index);
    }
    return sum;
}

// Updates a node in Binary Index Tree (BITree) at given index
// in BITree.  The given value 'val' is added to BITree[i] and
// all of its ancestors in tree.
void updateBIT(int BITree[], int n, int index, int val)
{
    // index in BITree[] is 1 more than the index in arr[]
    index = index + 1;

    // Traverse all ancestors and add 'val'
    while (index <= n)
    {
       // Add 'val' to current node of BI Tree
       BITree[index] += val;

       // Update index to that of parent in update View
       index += index & (-index);
    }
}

// Constructs and returns a Binary Indexed Tree for given
// array of size n.
int *constructBITree(int arr[], int n)
{
    // Create and initialize BITree[] as 0
    int *BITree = new int[n+1];
    for (int i=1; i<=n; i++)
        BITree[i] = 0;

    // Store the actual values in BITree[] using update()
    for (int i=0; i<n; i++)
        updateBIT(BITree, n, i, arr[i]);

    // Uncomment below lines to see contents of BITree[]
    //for (int i=1; i<=n; i++)
    //      cout << BITree[i] << " ";

    return BITree;
}


// Driver program to test above functions
int main()
{
```

```
    int freq[] = {2, 1, 1, 3, 2, 3, 4, 5, 6, 7, 8, 9};
    int n = sizeof(freq)/sizeof(freq[0]);
    int *BITree = constructBITree(freq, n);
    cout << "Sum of elements in arr[0..5] is "
         << getSum(BITree, 5);

    // Let use test the update operation
    freq[3] += 6;
    updateBIT(BITree, n, 3, 6); //Update BIT for above change in arr[]

    cout << "\nSum of elements in arr[0..5] after update is "
         << getSum(BITree, 5);

    return 0;
}
```

Trie

// Alphabet size (# of symbols)

```
#define ALPHABET_SIZE (2)
#define INDEX(c) ((int)c - (int)'0') //string kore pass korte hobe

#define FREE(p) \
    free(p);    \
    p = NULL;

// forward declration
using namespace std;
typedef struct trie_node trie_node_t;

// trie node
struct trie_node
{
    int value; // non zero if leaf
    trie_node_t *children[ALPHABET_SIZE];
};

// trie ADT
typedef struct trie trie_t;

struct trie
{
    trie_node_t *root;
    int count;
};

trie_node_t *getNode(void)
{
    trie_node_t *pNode = NULL;

    pNode = (trie_node_t *)malloc(sizeof(trie_node_t));

    if( pNode )
    {
        int i;

        pNode->value   = 0;
```

```
        for(i = 0; i < ALPHABET_SIZE; i++)
        {
            pNode->children[i] = NULL;
        }
    }

    return pNode;
}

void initialize(trie_t *pTrie)
{
    pTrie->root = getNode();
    pTrie->count = 0;
}

void insert(trie_t *pTrie, char key[])
{
    int level;
    int length = strlen(key);
    int index;
    trie_node_t *pCrawl;

    pTrie->count++;
    pCrawl = pTrie->root;

    for( level = 0; level < length; level++ )
    {
        index = INDEX(key[level]);

        if( pCrawl->children[index] )
        {
            // Skip current node
            pCrawl = pCrawl->children[index];
        }
        else
        {
            // Add new node
            pCrawl->children[index] = getNode();
            pCrawl = pCrawl->children[index];
        }
    }

    // mark last node as leaf (non zero)
    pCrawl->value = pTrie->count;
}

int search(trie_t *pTrie, char key[])
{
    int level;
    int length = strlen(key);
    int index;
    trie_node_t *pCrawl;

    pCrawl = pTrie->root;

    for( level = 0; level < length; level++ )
    {
        index = INDEX(key[level]);

        if( !pCrawl->children[index] )
        {
```

```
            return 0;
        }

        pCrawl = pCrawl->children[index];
    }

    return (0 != pCrawl && pCrawl->value);
}
char matched[33];
void xorsearch(trie_t *pTrie, char key[])
{
    int level;
    int length = strlen(key);
    int index;
    trie_node_t *pCrawl;

    pCrawl = pTrie->root;

    for( level = 0; level < length; level++ )
    {
        index = INDEX(key[level]);//either 0 or 1

        if( !pCrawl->children[index] )
        {
            pCrawl=pCrawl->children[1-index];
            matched[level]='0';
        }
        else{
            pCrawl = pCrawl->children[index];
            matched[level]='1';
        }
    }

    //return (0 != pCrawl && pCrawl->value);
}

int leafNode(trie_node_t *pNode)
{
    return (pNode->value != 0);
}

int isItFreeNode(trie_node_t *pNode)
{
    int i;
    for(i = 0; i < ALPHABET_SIZE; i++)
    {
        if( pNode->children[i] )
            return 0;
    }

    return 1;
}

bool deleteHelper(trie_node_t *pNode, char key[], int level, int len)
{
    if( pNode )
    {
        // Base case
        if( level == len )
        {
            if( pNode->value )
```

```
        {
            // Unmark leaf node
            pNode->value = 0;

            // If empty, node to be deleted
            if( isItFreeNode(pNode) )
            {
                return true;
            }

            return false;
        }
    }
    else // Recursive case
    {
        int index = INDEX(key[level]);

        if( deleteHelper(pNode->children[index], key, level+1, len) )
        {
            // last node marked, delete it
            FREE(pNode->children[index]);

            // recursively climb up, and delete eligible nodes
            return ( !leafNode(pNode) && isItFreeNode(pNode) );
        }
    }
}

return false;
}

void deleteKey(trie_t *pTrie, char key[])
{
    int len = strlen(key);

    if( len > 0 )
    {
        deleteHelper(pTrie->root, key, 0, len);
    }
}
```

Union Find

int representative[1003];

```
vector< pair<int,PII> >alledges;

int findrep(int a){
    if(representative[a]!=a){
        return representative[a]=findrep(representative[a]);
    }
    else{
        return a;
    }
}

void unionfind(int x, int y){
```

```
    int xp,yp;
    xp=findrep(x);
    yp=findrep(y);
    representative[yp]=xp;
}
```

Graph

Topological sort

```
#include<iostream>

#include <list>
#include <stack>
using namespace std;

// Class to represent a graph
class Graph
{
    int V;    // No. of vertices'

    // Pointer to an array containing adjacency listsList
    list<int> *adj;

    // A function used by topologicalSort
    void topologicalSortUtil(int v, bool visited[], stack<int> &Stack);
public:
    Graph(int V);   // Constructor

     // function to add an edge to graph
    void addEdge(int v, int w);

    // prints a Topological Sort of the complete graph
    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); // Add w to v’s list.
}

// A recursive function used by topologicalSort
void Graph::topologicalSortUtil(int v, bool visited[],
                                stack<int> &Stack)
{
    // Mark the current node as visited.
    visited[v] = true;

    // Recur for all the vertices adjacent to this vertex
    list<int>::iterator i;
    for (i = adj[v].begin(); i != adj[v].end(); ++i)
        if (!visited[*i])
            topologicalSortUtil(*i, visited, Stack);
```

```
    // Push current vertex to stack which stores result
    Stack.push(v);
}

// The function to do Topological Sort. It uses recursive
// topologicalSortUtil()
void Graph::topologicalSort()
{
    stack<int> Stack;

    // Mark all the vertices as not visited
    bool *visited = new bool[V];
    for (int i = 0; i < V; i++)
        visited[i] = false;

    // Call the recursive helper function to store Topological
    // Sort starting from all vertices one by one
    for (int i = 0; i < V; i++)
      if (visited[i] == false)
        topologicalSortUtil(i, visited, Stack);

    // Print contents of stack
    while (Stack.empty() == false)
    {
        cout << Stack.top() << " ";
        Stack.pop();
    }
}

// Driver program to test above functions
int main()
{
    // Create a graph given in the above diagram
    Graph g(6);
    g.addEdge(5, 2);
    g.addEdge(5, 0);
    g.addEdge(4, 0);
    g.addEdge(4, 1);
    g.addEdge(2, 3);
    g.addEdge(3, 1);

    cout << "Following is a Topological Sort of the given graph \n";
    g.topologicalSort();

    return 0;
}
```

Kahn's algo

```
#include<bits/stdc++.h>
using namespace std;

// Class to represent a graph
class Graph
{
    int V;    // No. of vertices'

    // Pointer to an array containing adjacency listsList
    list<int> *adj;
```

```
public:
    Graph(int V);   // Constructor

    // function to add an edge to graph
    void addEdge(int u, int v);

    // prints a Topological Sort of the complete graph
    void topologicalSort();
};

Graph::Graph(int V)
{
    this->V = V;
    adj = new list<int>[V];
}

void Graph::addEdge(int u, int v)
{
    adj[u].push_back(v);
}

// The function to do Topological Sort.
void Graph::topologicalSort()
{
    // Create a vector to store indegrees of all
    // vertices. Initialize all indegrees as 0.
    vector<int> in_degree(V, 0);

    // Traverse adjacency lists to fill indegrees of
    // vertices.  This step takes O(V+E) time
    for (int u=0; u<V; u++)
    {
        list<int>::iterator itr;
        for (itr = adj[u].begin(); itr != adj[u].end(); itr++)
             in_degree[*itr]++;
    }

    // Create an queue and enqueue all vertices with
    // indegree 0
    queue<int> q;
    for (int i = 0; i < V; i++)
        if (in_degree[i] == 0)
            q.push(i);

    // Initialize count of visited vertices
    int cnt = 0;

    // Create a vector to store result (A topological
    // ordering of the vertices)
    vector <int> top_order;

    // One by one dequeue vertices from queue and enqueue
    // adjacents if indegree of adjacent becomes 0
    while (!q.empty())
    {
        // Extract front of queue (or perform dequeue)
        // and add it to topological order
        int u = q.front();
        q.pop();
        top_order.push_back(u);
```

```
        // Iterate through all its neighbouring nodes
        // of dequeued node u and decrease their in-degree
        // by 1
        list<int>::iterator itr;
        for (itr = adj[u].begin(); itr != adj[u].end(); itr++)

            // If in-degree becomes zero, add it to queue
            if (--in_degree[*itr] == 0)
                q.push(*itr);

        cnt++;
    }

    // Check if there was a cycle
    if (cnt != V)
    {
        cout << "There exists a cycle in the graph\n";
        return;
    }

    // Print topological order
    for (int i=0; i<top_order.size(); i++)
        cout << top_order[i] << " ";
    cout << endl;
}

// Driver program to test above functions
int main()
{
    // Create a graph given in the above diagram
    Graph g(6);
    g.addEdge(5, 2);
    g.addEdge(5, 0);
    g.addEdge(4, 0);
    g.addEdge(4, 1);
    g.addEdge(2, 3);
    g.addEdge(3, 1);

    cout << "Following is a Topological Sort of\n";
    g.topologicalSort();

    return 0;
}
```

Articulation Point

#include<iostream>

```
#include <list>
#define NIL -1
using namespace std;

// A class that represents an undirected graph
class Graph
{
    int V;    // No. of vertices
    list<int> *adj;    // A dynamic array of adjacency lists
    void APUtil(int v, bool visited[], int disc[], int low[],
                int parent[], bool ap[]);
public:
```

```
    Graph(int V);   // Constructor
    void addEdge(int v, int w);   // function to add an edge to graph
    void AP();    // prints articulation points
};

Graph::Graph(int V)
{
    this->V = V;
    adj = new list<int>[V];
}

void Graph::addEdge(int v, int w)
{
    adj[v].push_back(w);
    adj[w].push_back(v);  // Note: the graph is undirected
}

// A recursive function that find articulation points using DFS traversal
// u --> The vertex to be visited next
// visited[] --> keeps tract of visited vertices
// disc[] --> Stores discovery times of visited vertices
// parent[] --> Stores parent vertices in DFS tree
// ap[] --> Store articulation points
void Graph::APUtil(int u, bool visited[], int disc[],
                                      int low[], int parent[], bool ap[])
{
    // A static variable is used for simplicity, we can avoid use of static
    // variable by passing a pointer.
    static int time = 0;

    // Count of children in DFS Tree
    int children = 0;

    // Mark the current node as visited
    visited[u] = true;

    // Initialize discovery time and low value
    disc[u] = low[u] = ++time;

    // Go through all vertices aadjacent to this
    list<int>::iterator i;
    for (i = adj[u].begin(); i != adj[u].end(); ++i)
    {
        int v = *i;  // v is current adjacent of u

        // If v is not visited yet, then make it a child of u
        // in DFS tree and recur for it
        if (!visited[v])
        {
            children++;
            parent[v] = u;
            APUtil(v, visited, disc, low, parent, ap);

            // Check if the subtree rooted with v has a connection to
            // one of the ancestors of u
            low[u]  = min(low[u], low[v]);

            // u is an articulation point in following cases

            // (1) u is root of DFS tree and has two or more chilren.
            if (parent[u] == NIL && children > 1)
```

```
                ap[u] = true;

            // (2) If u is not root and low value of one of its child is
more
            // than discovery value of u.
            if (parent[u] != NIL && low[v] >= disc[u])
                ap[u] = true;
        }

        // Update low value of u for parent function calls.
        else if (v != parent[u])
            low[u]  = min(low[u], disc[v]);
    }
}

// The function to do DFS traversal. It uses recursive function APUtil()
void Graph::AP()
{
    // Mark all the vertices as not visited
    bool *visited = new bool[V];
    int *disc = new int[V];
    int *low = new int[V];
    int *parent = new int[V];
    bool *ap = new bool[V]; // To store articulation points

    // Initialize parent and visited, and ap(articulation point) arrays
    for (int i = 0; i < V; i++)
    {
        parent[i] = NIL;
        visited[i] = false;
        ap[i] = false;
    }

    // Call the recursive helper function to find articulation points
    // in DFS tree rooted with vertex 'i'
    for (int i = 0; i < V; i++)
        if (visited[i] == false)
            APUtil(i, visited, disc, low, parent, ap);

    // Now ap[] contains articulation points, print them
    for (int i = 0; i < V; i++)
        if (ap[i] == true)
            cout << i << " ";
}

// Driver program to test above function
int main()
{
    // Create graphs given in above diagrams
    cout << "\nArticulation points in first graph \n";
    Graph g1(5);
    g1.addEdge(1, 0);
    g1.addEdge(0, 2);
    g1.addEdge(2, 1);
    g1.addEdge(0, 3);
    g1.addEdge(3, 4);
    g1.AP();

    cout << "\nArticulation points in second graph \n";
    Graph g2(4);
    g2.addEdge(0, 1);
```

```
    g2.addEdge(1, 2);
    g2.addEdge(2, 3);
    g2.AP();

    cout << "\nArticulation points in third graph \n";
    Graph g3(7);
    g3.addEdge(0, 1);
    g3.addEdge(1, 2);
    g3.addEdge(2, 0);
    g3.addEdge(1, 3);
    g3.addEdge(1, 4);
    g3.addEdge(1, 6);
    g3.addEdge(3, 5);
    g3.addEdge(4, 5);
    g3.AP();

    return 0;
}
```

Stable Marriage

```
int boyfree[1005];
int girlfree[1005];
vector<VI>boypref;
vector<VI>girlpref;
int girlproposed[1005][1005];
//int boyproposed[1005][1005];
int boypriority[1005][1005];
void init(int n){
    FREP(i,1,n){
        boyfree[i]=-1;
        girlfree[i]=-1;
        FREP(j,1,n){
            girlproposed[i][j]=-1;
           // boypriority[i][j]=0;
        }
    }
}



int main(){
    int t;
    scanf("%d",&t);
    int cs=1;
    while(t--){
        int n;
        scanf("%d",&n);
        init(n);
        boypref.clear();
        girlpref.clear();
        VI line;
        FREP(i,1,(n+3)){
            boypref.PB(line);
            girlpref.PB(line);
        }
        FREP(i,1,n){
            FREP(j,1,n){
```

```
                int a;
                scanf("%d",&a);
                girlpref[i].PB(a);
            }
        }
        FREP(i,1,n){
            FREP(j,1,n){
                int a;
                scanf("%d",&a);
                boypref[i].PB(a);
                boypriority[i][a]=j;
            }
        }
        while(true){
            int f=1;
            FREP(i,1,n){
                if(girlfree[i]==-1){
                    f=0;
                    FREP(j,0,(n-1)){
                        int curboy=girlpref[i][j];
                        int prop=0;
                        if(girlproposed[i][curboy]==-1){
                            girlproposed[i][curboy]=1;
                            prop=1;
                            if(boyfree[curboy]==-1){
                                boyfree[curboy]=i;
                                girlfree[i]=curboy;
                            }
                            else{
                                int boytakenby=boyfree[curboy];
                                if(boypriority[curboy][i]<boypriority[curboy][boytakenby]){
                                    girlfree[boytakenby]=-1;
                                    girlfree[i]=curboy;
                                    boyfree[curboy]=i;
                                }
                            }
                        }
                        if(prop){
                            break;
                        }
                    }
                }
            }
            if(f){
                break;
            }
        }
        if(cs>1){
            printf("\n");
        }
        FREP(i,1,n){
            printf("%d\n",girlfree[i]);
        }
        cs++;
    }
    return 0;
}
```

Maximum Bipartite Matching

```
#include <iostream>
#include <string.h>
using namespace std;

// M is number of applicants and N is number of jobs
#define M 6
#define N 6

// A DFS based recursive function that returns true if a
// matching for vertex u is possible
bool bpm(bool bpGraph[M][N], int u, bool seen[], int matchR[])
{
    // Try every job one by one
    for (int v = 0; v < N; v++)
    {
        // If applicant u is interested in job v and v is
        // not visited
        if (bpGraph[u][v] && !seen[v])
        {
            seen[v] = true; // Mark v as visited

            // If job 'v' is not assigned to an applicant OR
            // previously assigned applicant for job v (which is matchR[v])
            // has an alternate job available.
            // Since v is marked as visited in the above line, matchR[v]
            // in the following recursive call will not get job 'v' again
            if (matchR[v] < 0 || bpm(bpGraph, matchR[v], seen, matchR))
            {
                matchR[v] = u;
                return true;
            }
        }
    }
    return false;
}

// Returns maximum number of matching from M to N
int maxBPM(bool bpGraph[M][N])
{
    // An array to keep track of the applicants assigned to
    // jobs. The value of matchR[i] is the applicant number
    // assigned to job i, the value -1 indicates nobody is
    // assigned.
    int matchR[N];

    // Initially all jobs are available
    memset(matchR, -1, sizeof(matchR));

    int result = 0; // Count of jobs assigned to applicants
    for (int u = 0; u < M; u++)
    {
        // Mark all jobs as not seen for next applicant.
        bool seen[N];
        memset(seen, 0, sizeof(seen));

        // Find if the applicant 'u' can get a job
        if (bpm(bpGraph, u, seen, matchR))
            result++;
    }
    return result;
}
```

```
// Driver program to test above functions
int main()
{
    // Let us create a bpGraph shown in the above example
    bool bpGraph[M][N] = {  {0, 1, 1, 0, 0, 0},
                        {1, 0, 0, 1, 0, 0},
                        {0, 0, 1, 0, 0, 0},
                        {0, 0, 1, 1, 0, 0},
                        {0, 0, 0, 0, 0, 0},
                        {0, 0, 0, 0, 0, 1}
                      };

    cout << "Maximum number of applicants that can get job is "
         << maxBPM(bpGraph);

    return 0;
}
```

Max Flow

```
#include <iostream>
#include <limits.h>
#include <string.h>
#include <queue>
using namespace std;

// Number of vertices in given graph
#define V 6

/* Returns true if there is a path from source 's' to sink 't' in
  residual graph. Also fills parent[] to store the path */
bool bfs(int rGraph[V][V], int s, int t, int parent[])
{
    // Create a visited array and mark all vertices as not visited
    bool visited[V];
    memset(visited, 0, sizeof(visited));

    // Create a queue, enqueue source vertex and mark source vertex
    // as visited
    queue <int> q;
    q.push(s);
    visited[s] = true;
    parent[s] = -1;

    // Standard BFS Loop
    while (!q.empty())
    {
        int u = q.front();
        q.pop();

        for (int v=0; v<V; v++)
        {
            if (visited[v]==false && rGraph[u][v] > 0)
            {
                q.push(v);
                parent[v] = u;
                visited[v] = true;
            }
        }
    }
```

```
    // If we reached sink in BFS starting from source, then return
    // true, else false
    return (visited[t] == true);
}

// Returns tne maximum flow from s to t in the given graph
int fordFulkerson(int graph[V][V], int s, int t)
{
    int u, v;

    // Create a residual graph and fill the residual graph with
    // given capacities in the original graph as residual capacities
    // in residual graph
    int rGraph[V][V]; // Residual graph where rGraph[i][j] indicates
                     // residual capacity of edge from i to j (if there
                     // is an edge. If rGraph[i][j] is 0, then there is
not)
    for (u = 0; u < V; u++)
        for (v = 0; v < V; v++)
             rGraph[u][v] = graph[u][v];

    int parent[V];  // This array is filled by BFS and to store path

    int max_flow = 0;  // There is no flow initially

    // Augment the flow while tere is path from source to sink
    while (bfs(rGraph, s, t, parent))
    {
        // Find minimum residual capacity of the edhes along the
        // path filled by BFS. Or we can say find the maximum flow
        // through the path found.
        int path_flow = INT_MAX;
        for (v=t; v!=s; v=parent[v])
        {
            u = parent[v];
            path_flow = min(path_flow, rGraph[u][v]);
        }

        // update residual capacities of the edges and reverse edges
        // along the path
        for (v=t; v != s; v=parent[v])
        {
            u = parent[v];
            rGraph[u][v] -= path_flow;
            rGraph[v][u] += path_flow;
        }

        // Add path flow to overall flow
        max_flow += path_flow;
    }

    // Return the overall flow
    return max_flow;
}

// Driver program to test above functions
int main()
{
    // Let us create a graph shown in the above example
    int graph[V][V] = { {0, 16, 13, 0, 0, 0},
```

```
                        {0, 0, 10, 12, 0, 0},
                        {0, 4, 0, 0, 14, 0},
                        {0, 0, 9, 0, 0, 20},
                        {0, 0, 0, 7, 0, 4},
                        {0, 0, 0, 0, 0, 0}
                      };

    cout << "The maximum possible flow is " << fordFulkerson(graph, 0, 5);

    return 0;
}
```

Dijkstra

```
#include<bits/stdc++.h>
using namespace std;
# define INF 0x3f3f3f3f

// This class represents a directed graph using
// adjacency list representation
class Graph
{
    int V;    // No. of vertices

    // In a weighted graph, we need to store vertex
    // and weight pair for every edge
    list< pair<int, int> > *adj;

public:
    Graph(int V);  // Constructor

    // function to add an edge to graph
    void addEdge(int u, int v, int w);

    // prints shortest path from s
    void shortestPath(int s);
};

// Allocates memory for adjacency list
Graph::Graph(int V)
{
    this->V = V;
    adj = new list< pair<int, int> >[V];
}

void Graph::addEdge(int u, int v, int w)
{
    adj[u].push_back(make_pair(v, w));
    adj[v].push_back(make_pair(u, w));
}

// Prints shortest paths from src to all other vertices
void Graph::shortestPath(int src)
{
    // Create a set to store vertices that are being
    // prerocessed
    set< pair<int, int> > setds;

    // Create a vector for distances and initialize all
```

```
    // distances as infinite (INF)
    vector<int> dist(V, INF);

    // Insert source itself in Set and initialize its
    // distance as 0.
    setds.insert(make_pair(0, src));
    dist[src] = 0;

    /* Looping till all shortest distance are finalized
       then setds will become empty */
    while (!setds.empty())
    {
        // The first vertex in Set is the minimum distance
        // vertex, extract it from set.
        pair<int, int> tmp = *(setds.begin());
        setds.erase(setds.begin());

        // vertex label is stored in second of pair (it
        // has to be done this way to keep the vertices
        // sorted distance (distance must be first item
        // in pair)
        int u = tmp.second;

        // 'i' is used to get all adjacent vertices of a vertex
        list< pair<int, int> >::iterator i;
        for (i = adj[u].begin(); i != adj[u].end(); ++i)
        {
            // Get vertex label and weight of current adjacent
            // of u.
            int v = (*i).first;
            int weight = (*i).second;

            //  If there is shorter path to v through u.
            if (dist[v] > dist[u] + weight)
            {
                /*  If distance of v is not INF then it must be in
                    our set, so removing it and inserting again
                    with updated less distance.
                    Note : We extract only those vertices from Set
                    for which distance is finalized. So for them,
                    we would never reach here.  */
                if (dist[v] != INF)
                    setds.erase(setds.find(make_pair(dist[v], v)));

                // Updating distance of v
                dist[v] = dist[u] + weight;
                setds.insert(make_pair(dist[v], v));
            }
        }
    }

    // Print shortest distances stored in dist[]
    printf("Vertex   Distance from Source\n");
    for (int i = 0; i < V; ++i)
        printf("%d \t\t %d\n", i, dist[i]);
}

// Driver program to test methods of graph class
int main()
{
    // create the graph given in above fugure
```

```
    int V = 9;
    Graph g(V);

    //  making above shown graph
    g.addEdge(0, 1, 4);
    g.addEdge(0, 7, 8);
    g.addEdge(1, 2, 8);
    g.addEdge(1, 7, 11);
    g.addEdge(2, 3, 7);
    g.addEdge(2, 8, 2);
    g.addEdge(2, 5, 4);
    g.addEdge(3, 4, 9);
    g.addEdge(3, 5, 14);
    g.addEdge(4, 5, 10);
    g.addEdge(5, 6, 2);
    g.addEdge(6, 7, 1);
    g.addEdge(6, 8, 6);
    g.addEdge(7, 8, 7);

    g.shortestPath(0);

    return 0;
}
```

String

KMP

```
#include<bits/stdc++.h>

void computeLPSArray(char *pat, int M, int *lps);

// Prints occurrences of txt[] in pat[]
void KMPSearch(char *pat, char *txt)
{
    int M = strlen(pat);
    int N = strlen(txt);

    // create lps[] that will hold the longest prefix suffix
    // values for pattern
    int lps[M];

    // Preprocess the pattern (calculate lps[] array)
    computeLPSArray(pat, M, lps);

    int i = 0;  // index for txt[]
    int j  = 0;  // index for pat[]
    while (i < N)
    {
        if (pat[j] == txt[i])
        {
            j++;
            i++;
        }

        if (j == M)
        {
            printf("Found pattern at index %d \n", i-j);
            j = lps[j-1];
```

```
        }

        // mismatch after j matches
        else if (i < N && pat[j] != txt[i])
        {
            // Do not match lps[0..lps[j-1]] characters,
            // they will match anyway
            if (j != 0)
                j = lps[j-1];
            else
                i = i+1;
        }
    }
}

// Fills lps[] for given patttern pat[0..M-1]
void computeLPSArray(char *pat, int M, int *lps)
{
    // length of the previous longest prefix suffix
    int len = 0;

    lps[0] = 0; // lps[0] is always 0

    // the loop calculates lps[i] for i = 1 to M-1
    int i = 1;
    while (i < M)
    {
        if (pat[i] == pat[len])
        {
            len++;
            lps[i] = len;
            i++;
        }
        else // (pat[i] != pat[len])
        {
            // This is tricky. Consider the example.
            // AAACAAAA and i = 7. The idea is similar
            // to search step.
            if (len != 0)
            {
                len = lps[len-1];

                // Also, note that we do not increment
                // i here
            }
            else // if (len == 0)
            {
                lps[i] = 0;
                i++;
            }
        }
    }
}

// Driver program to test above function
int main()
{
    char *txt = "ABABDABACDABABCABAB";
    char *pat = "ABABCABAB";
    KMPSearch(pat, txt);
    return 0;
```

```
}
```

Manacher

```
// A C program to implement Manacher's Algorithm
#include <stdio.h>
#include <string.h>

char text[100];
int min(int a, int b)
{
    int res = a;
    if(b < a)
        res = b;
    return res;
}

void findLongestPalindromicString()
{
    int N = strlen(text);
    if(N == 0)
        return;
    N = 2*N + 1; //Position count
    int L[N]; //LPS Length Array
    L[0] = 0;
    L[1] = 1;
    int C = 1; //centerPosition
    int R = 2; //centerRightPosition
    int i = 0; //currentRightPosition
    int iMirror; //currentLeftPosition
    int maxLPSLength = 0;
    int maxLPSCenterPosition = 0;
    int start = -1;
    int end = -1;
    int diff = -1;

    //Uncomment it to print LPS Length array
    //printf("%d %d ", L[0], L[1]);
    for (i = 2; i < N; i++)
    {
        //get currentLeftPosition iMirror for currentRightPosition i
        iMirror  = 2*C-i;
        L[i] = 0;
        diff = R - i;
        //If currentRightPosition i is within centerRightPosition R
        if(diff > 0)
            L[i] = min(L[iMirror], diff);

        //Attempt to expand palindrome centered at currentRightPosition i
        //Here for odd positions, we compare characters and
        //if match then increment LPS Length by ONE
        //If even position, we just increment LPS by ONE without
        //any character comparison
        while ( ((i + L[i]) < N && (i - L[i]) > 0) &&
            ( ((i + L[i] + 1) % 2 == 0) ||
            (text[(i + L[i] + 1)/2] == text[(i - L[i] - 1)/2] )))
        {
            L[i]++;
        }
```

```
        if(L[i] > maxLPSLength)  // Track maxLPSLength
        {
            maxLPSLength = L[i];
            maxLPSCenterPosition = i;
        }

        //If palindrome centered at currentRightPosition i
        //expand beyond centerRightPosition R,
        //adjust centerPosition C based on expanded palindrome.
        if (i + L[i] > R)
        {
            C = i;
            R = i + L[i];
        }
        //Uncomment it to print LPS Length array
        //printf("%d ", L[i]);
    }
    //printf("\n");
    start = (maxLPSCenterPosition - maxLPSLength)/2;
    end = start + maxLPSLength - 1;
    printf("LPS of string is %s : ", text);
    for(i=start; i<=end; i++)
        printf("%c", text[i]);
    printf("\n");
}

int main(int argc, char *argv[])
{

    strcpy(text, "babcbabcbaccba");
    findLongestPalindromicString();

    strcpy(text, "abaaba");
    findLongestPalindromicString();

    strcpy(text, "abababa");
    findLongestPalindromicString();

    strcpy(text, "abcbabcbabcba");
    findLongestPalindromicString();

    strcpy(text, "forgeeksskeegfor");
    findLongestPalindromicString();

    strcpy(text, "caba");
    findLongestPalindromicString();

    strcpy(text, "abacdfgdcaba");
    findLongestPalindromicString();

    strcpy(text, "abacdfgdcabba");
    findLongestPalindromicString();

    strcpy(text, "abacdedcaba");
    findLongestPalindromicString();

    return 0;
}
```

Math

Inverse Mod 1

```
// Iterative C++ program to find modular inverse using
// extended Euclid algorithm
#include <stdio.h>

// Returns modulo inverse of a with respect to m using
// extended Euclid Algorithm
// Assumption: a and m are coprimes, i.e., gcd(a, m) = 1
int modInverse(int a, int m)
{
    int m0 = m, t, q;
    int x0 = 0, x1 = 1;

    if (m == 1)
      return 0;

    while (a > 1)
    {
        // q is quotient
        q = a / m;

        t = m;

        // m is remainder now, process same as
        // Euclid's algo
        m = a % m, a = t;

        t = x0;

        x0 = x1 - q * x0;

        x1 = t;
    }

    // Make x1 positive
    if (x1 < 0)
       x1 += m0;

    return x1;
}

// Driver program to test above function
int main()
{
    int a = 3, m = 11;

    printf("Modular multiplicative inverse is %d\n",
          modInverse(a, m));
    return 0;
}
```

Inverse Mod 2

```
// C++ program to find modular inverse of a under modulo m
// This program works only if m is prime.
```

```
#include<iostream>
using namespace std;

// To find GCD of a and b
int gcd(int a, int b);

// To compute x raised to power y under modulo m
int power(int x, unsigned int y, unsigned int m);

// Function to find modular inverse of a under modulo m
// Assumption: m is prime
void modInverse(int a, int m)
{
    int g = gcd(a, m);
    if (g != 1)
        cout << "Inverse doesn't exist";
    else
    {
        // If a and m are relatively prime, then modulo inverse
        // is a^(m-2) mode m
        cout << "Modular multiplicative inverse is "
             << power(a, m-2, m);
    }
}

// To compute x^y under modulo m
int power(int x, unsigned int y, unsigned int m)
{
    if (y == 0)
        return 1;
    int p = power(x, y/2, m) % m;
    p = (p * p) % m;

    return (y%2 == 0)? p : (x * p) % m;
}

// Function to return gcd of a and b
int gcd(int a, int b)
{
    if (a == 0)
        return b;
    return gcd(b%a, a);
}

// Driver Program
int main()
{
    int a = 3, m = 11;
    modInverse(a, m);
    return 0;
}
```

Totient phi

```
int phi(int n)
{
    int result = n;   // Initialize result as n

    // Consider all prime factors of n and subtract their
    // multiples from result
```

```
    for (int p=2; p*p<=n; ++p)
    {
        // Check if p is a prime factor.
        if (n % p == 0)
        {
            // If yes, then update n and result
            while (n % p == 0)
                n /= p;
            result -= result / p;
        }
    }

    // If n has a prime factor greater than sqrt(n)
    // (There can be at-most one such prime factor)
    if (n > 1)
        result -= result / n;
    return result;
}
```

Geometry

Convex hull Graham Scan

```
#include <iostream>
#include <stack>
#include <stdlib.h>
using namespace std;

struct Point
{
    int x, y;
};

// A globle point needed for  sorting points with reference
// to  the first point Used in compare function of qsort()
Point p0;

// A utility function to find next to top in a stack
Point nextToTop(stack<Point> &S)
{
    Point p = S.top();
    S.pop();
    Point res = S.top();
    S.push(p);
    return res;
}

// A utility function to swap two points
int swap(Point &p1, Point &p2)
{
    Point temp = p1;
    p1 = p2;
    p2 = temp;
}

// A utility function to return square of distance
// between p1 and p2
int distSq(Point p1, Point p2)
{
    return (p1.x - p2.x)*(p1.x - p2.x) +
```

```
          (p1.y - p2.y)*(p1.y - p2.y);
}

// To find orientation of ordered triplet (p, q, r).
// The function returns following values
// 0 --> p, q and r are colinear
// 1 --> Clockwise
// 2 --> Counterclockwise
int orientation(Point p, Point q, Point r)
{
    int val = (q.y - p.y) * (r.x - q.x) -
              (q.x - p.x) * (r.y - q.y);

    if (val == 0) return 0;  // colinear
    return (val > 0)? 1: 2; // clock or counterclock wise
}

// A function used by library function qsort() to sort an array of
// points with respect to the first point
int compare(const void *vp1, const void *vp2)
{
   Point *p1 = (Point *)vp1;
   Point *p2 = (Point *)vp2;

   // Find orientation
   int o = orientation(p0, *p1, *p2);
   if (o == 0)
     return (distSq(p0, *p2) >= distSq(p0, *p1))? -1 : 1;

   return (o == 2)? -1: 1;
}

// Prints convex hull of a set of n points.
void convexHull(Point points[], int n)
{
   // Find the bottommost point
   int ymin = points[0].y, min = 0;
   for (int i = 1; i < n; i++)
   {
     int y = points[i].y;

     // Pick the bottom-most or chose the left
     // most point in case of tie
     if ((y < ymin) || (ymin == y &&
         points[i].x < points[min].x))
        ymin = points[i].y, min = i;
   }

   // Place the bottom-most point at first position
   swap(points[0], points[min]);

   // Sort n-1 points with respect to the first point.
   // A point p1 comes before p2 in sorted ouput if p2
   // has larger polar angle (in counterclockwise
   // direction) than p1
   p0 = points[0];
   qsort(&points[1], n-1, sizeof(Point), compare);

   // If two or more points make same angle with p0,
   // Remove all but the one that is farthest from p0
   // Remember that, in above sorting, our criteria was
```

```
    // to keep the farthest point at the end when more than
    // one points have same angle.
    int m = 1; // Initialize size of modified array
    for (int i=1; i<n; i++)
    {
        // Keep removing i while angle of i and i+1 is same
        // with respect to p0
        while (i < n-1 && orientation(p0, points[i],
                                      points[i+1]) == 0)
           i++;


        points[m] = points[i];
        m++;  // Update size of modified array
    }

    // If modified array of points has less than 3 points,
    // convex hull is not possible
    if (m < 3) return;

    // Create an empty stack and push first three points
    // to it.
    stack<Point> S;
    S.push(points[0]);
    S.push(points[1]);
    S.push(points[2]);

    // Process remaining n-3 points
    for (int i = 3; i < m; i++)
    {
       // Keep removing top while the angle formed by
       // points next-to-top, top, and points[i] makes
       // a non-left turn
       while (orientation(nextToTop(S), S.top(), points[i]) != 2)
          S.pop();
       S.push(points[i]);
    }

    // Now stack has the output points, print contents of stack
    while (!S.empty())
    {
        Point p = S.top();
        cout << "(" << p.x << ", " << p.y <<")" << endl;
        S.pop();
    }
}

// Driver program to test above functions
int main()
{
    Point points[] = {{0, 3}, {1, 1}, {2, 2}, {4, 4},
                      {0, 0}, {1, 2}, {3, 1}, {3, 3}};
    int n = sizeof(points)/sizeof(points[0]);
    convexHull(points, n);
    return 0;
}
```

Convex Hull Monotonus chain

```
// Implementation of Andrew's monotone chain 2D convex hull algorithm.
```

```
// Asymptotic complexity: O(n log n).
// Practical performance: 0.5-1.0 seconds for n=1000000 on a 1GHz machine.
#include <algorithm>
#include <vector>
using namespace std;

typedef double coord_t;         // coordinate type
typedef double coord2_t;  // must be big enough to hold 2*max(|
coordinate|)^2

struct Point {
        coord_t x, y;

        bool operator <(const Point &p) const {
                return x < p.x || (x == p.x && y < p.y);
        }
};

// 2D cross product of OA and OB vectors, i.e. z-component of their 3D cross
product.
// Returns a positive value, if OAB makes a counter-clockwise turn,
// negative for clockwise turn, and zero if the points are collinear.
coord2_t cross(const Point &O, const Point &A, const Point &B)
{
        return (A.x - O.x) * (B.y - O.y) - (A.y - O.y) * (B.x - O.x);
}

// Returns a list of points on the convex hull in counter-clockwise order.
// Note: the last point in the returned list is the same as the first one.
vector<Point> convex_hull(vector<Point> P)
{
        int n = P.size(), k = 0;
        vector<Point> H(2*n);

        // Sort points lexicographically
        sort(P.begin(), P.end());

        // Build lower hull
        for (int i = 0; i < n; ++i) {
                while (k >= 2 && cross(H[k-2], H[k-1], P[i]) <= 0) k--;
                H[k++] = P[i];
        }

        // Build upper hull
        for (int i = n-2, t = k+1; i >= 0; i--) {
                while (k >= t && cross(H[k-2], H[k-1], P[i]) <= 0) k--;
                H[k++] = P[i];
        }

        H.resize(k-1);
        return H;
}
```

Polygon area

```
#include <bits/stdc++.h>
using namespace std;

// (X[i], Y[i]) are coordinates of i'th point.
double polygonArea(double X[], double Y[], int n)
```

```
{
    // Initialze area
    double area = 0.0;

    // Calculate value of shoelace formula
    int j = n - 1;
    for (int i = 0; i < n; i++)
    {
        area += (X[j] + X[i]) * (Y[j] - Y[i]);
        j = i;  // j is previous vertex to i
    }

    // Return absolute value
    return abs(area / 2.0);
}

// Driver program to test above function
int main()
{
    double X[] = {0, 2, 4};
    double Y[] = {1, 3, 7};

    int n = sizeof(X)/sizeof(X[0]);

    cout << polygonArea(X, Y, n);
}
```

```
int t r u n ( p o i n t p0 , p o i n t p1 , p o i n t p2 )
{
i n t r e s u l t =(p2 . x−p0 . x ) ∗ ( p1 . y−p0 . y ) −(p1 . x−p0 . x ) ∗ ( p2 . y−p0 . y ) ;
return r e s u l t ;
}
b o o l isConvex ( i n t n , v e c t o r <p o i n t >v )
{
i n t pos =0 , neg =0;
f o r ( i n t i = 0 ; i < n ; i ++)
{
i n t prev = ( i + n −1) % n , nex t = ( i + 1 ) % n ;
p o i n t A=v [ i ] ;
p o i n t B=v [ prev ] ;
p o i n t C=v [ n ext ] ;
i n t pv=t r u n (A, B, C) ;
i f ( pv > 0 ) pos++;
else
{
i f ( pv < 0 ) neg++;
}
}
r e t u r n ( pos == 0 ) | | ( neg == 0 ) ;
}
```

9.7

1

2

Closest Pair Algorithm

```
c o n s t i n t MAX=100005;
```

```
struct point
{
int x , y , i ;
} ;
p o i n t a r r [MAX] , s or ted Y [MAX] ;
b o o l f l a g [MAX] ;
t e m p l a t e < c l a s s T> i n t g e t d i s t (T a , T b )
{
r e t u r n max( abs ( a . x − b . x ) , abs ( a . y − b . y ) ) ;
}
b o o l compareX ( c o n s t p o i n t &a , c o n s t p o i n t &b )
{
return a . x < b . x ;
}
b o o l compareY ( c o n s t p o i n t &a , c o n s t p o i n t &b )
{
return a . y < b . y ;
}

int c l o s e s t p a i r ( point X[ ] , point Y[ ] , int n)
{
int l e f t c a l l , r i g h t c a l l , mindist ;
i f ( n == 1 ) r e t u r n i n f ;
i f ( n == 2 ) r e t u r n g e t d i s t (X [ 0 ] , X [ 1 ] ) ;
i n t n1 , n2 , ns , j , m = n / 2 , i ;
p o i n t xL [m + 1 ] , xR [m + 1 ] , yL [m + 1 ] , yR [m + 1 ] , Xm = X[m − 1 ] , yS [ n ] ;
f o r ( i = 0 ; i < m; i ++)
{
xL [ i ] = X[ i ] ;
f l a g [X[ i ] . i ] = 0 ;
}
f o r ( ; i < n ; i ++)
{
xR [ i − m] = X[ i ] ;
f l a g [X[ i ] . i ] = 1 ;
}
f o r ( i = n2 = n1 = 0 ; i < n ; i ++)
{
i f ( ! f l a g [Y[ i ] . i ] ) yL [ n1++] = Y[ i ] ;
e l s e yR [ n2++] = Y[ i ] ;
}
l e f t c a l l = c l o s e s t p a i r ( xL , yL , n1 ) ;
r i g h t c a l l = c l o s e s t p a i r (xR , yR , n2 ) ;
m i n d i s t = min ( l e f t c a l l , r i g h t c a l l ) ;
f o r ( i = ns = 0 ; i < n ; i ++)
i f ( (Y[ i ] . x − Xm. x ) < m i n d i s t )
yS [ ns++] = Y[ i ] ;
f o r ( i = 0 ; i < ns ; i ++)
f o r ( j = i + 1 ; j < ns && ( yS [ j ] . y − yS [ i ] . y ) < m i n d i s t ; j ++)
m i n d i s t = min ( m i n d i s t , g e t d i s t ( yS [ i ] , yS [ j ] ) ) ;
return mindist ;
}

int Find closestpair ( int n)
{
s o r t ( a r r , a r r + n , compareX ) ;
s o r t ( sortedY , sor te dY + n , compareY ) ;
i n t ans = c l o s e s t p a i r ( a r r , sortedY , n ) ;
r e t u r n ans ;
}
```