RECursion, NIT Durgapur

Team Reference Material

1/47. Ranges:

char- -128 to 127
short- -32768 to 32767
int- <=2*10^9
long- **same as int
long long- <=9*10^18

2/47. Moduler Multiplication:

```
int mulmod(int a, int b, int c)
{
    int x=0,y=a%c;
    while(b>0)
    {
        if(b%2==1)
        {
            x=(x+y)%c;
        }
        y=(y*2)%c;
        b/=2;
    }
    return x%c;
}
```

3/47. Moduler Exponentiation:

```
int power ( int m , int n , int mod )
{
    int temp = m ;
    int ans = 1 ;
    while (n > 0)
    {
        if ( n%2 == 0 )
        {
            temp = ( temp * temp ) % mod ; n /= 2 ;
        }
        else
        {
            ans = ( ans * temp ) %mod ; n -= 1 ;
        }
    }
    return ans ;
}
```

4/47. Fast input:

```
inline int input()
 {
    char c = getchar() ;
    while (c < '0' || c > '9' ) c = getchar() ;
    int ret = 0 ;
     while ( c >= '0' && c <= '9' )
    {
```

```
        ret = 10 * ret + c – 48 ;
         c = getchar() ;
    }
     return ret ;
 }
```

5/47. nCr term:

```
s = 1 ;
for ( i=1 ; i<=r ; i++ )
{
    s = s * ( n - ( r – i ) ) / i ;
}
return s ;
```

6/47. Probability:

$$P(A|B) = \frac{P(A \cap B)}{P(B)}, \text{ if } P(B) \neq 0,$$

$$P(A|B) = \frac{P(B|A)\,P(A)}{P(B)}, \text{ if } P(B) \neq 0.$$

7/47. Bellman-Ford Algorithm:

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

8/47. Dijkstra's Algorithm [O (V^2)]:

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

void dijkstra ( int graph[V][V] , int src )
{
    int dist[V] ;           //dist[] will be the final array
    bool sptSet[V] ;   //to determine which node is yet to be visited
    for ( int i = 0 ; i < V ; i++ )
        dist[i] = INT_MAX , sptSet[i] = false ;
    dist[src] = 0 ;

    for ( int count = 0 ; count < V-1 ; count++ )
    {
        int u = minDistance (dist , sptSet ) ;
        sptSet[u] = true ;
```

```
        for (int v = 0; v < V; v++)
            if ( ! sptSet[v] && graph[u][v] && dist[u] != INT_MAX && dist[u]+graph[u][v] < dist[v] )
                dist[v] = dist[u] + graph[u][v] ;
    }
}
```

9/47. Euler totient function:

//it counts numbers less than or equal to n that are relatively //prime to n
//p1 , p2 ,…., pr are the prime factors of n

f(n) = n * (1 - 1/p1) * (1 - 1/p2) * (1 - 1/p3)…….(1 – 1/pr).

10/47. Dijkstra's Algorithm [O (ElogV)]:

```
typedef pair<LL,LL> PII;
int main()
{
    long long N , s ;
    cin >> N >> s ;
    vector < vector < PII > > graph ( N ) ;
    for ( int i = 0 ; i < N ; i++ )
    {
        long long M ;
        cin >> M ;
        for ( int j =0 ; j < M ; j++ )
        {
            long long vertex , dist;
                cin >> vertex >> dist ;
                graph[i].push_back ( make_pair ( dist,vertex ) ) ;
        }
    }

    priority_queue < PII , vector<PII> , greater<PII> > Q;

    vector < long long > dist ( N , INF ) , dad ( N , -1 ) ;

    Q.push ( make_pair ( 0 , s ) ) ;
    dist [ s ] = 0 ;
    while ( ! Q.empty () )
    {
        PII p = Q.top() ;
        Q.pop() ;
        long long here = p.second ;
        for ( vector<PII> : : iterator it = graph[here].begin() ; it ! = graph[here].end() ; it++)
        {
                if ( dist [ here ] + it -> first < dist [ it -> second ] )
                {
                    dist [ it -> second ] = dist [ here ] + it -> first ;
                    dad [ it -> second ] = here ;
                    Q.push ( make_pair ( dist [ it -> second ],it -> second)) ;
                }
        }
    }
    return 0;
}
```

11/47. Union by Rank and Path Compression:

```
int find ( struct subset subsets [ ] , int i )
{
```

```
    if ( subsets [ i ].parent != i )
        subsets [ i ].parent = find ( subsets , subsets[i].parent ) ;
    return subsets[i].parent ;
}

void Union ( struct subset subsets [ ] , int x , int y )
{
    int xroot = find ( subsets , x ) ;
    int yroot = find ( subsets , y ) ;
    if ( subsets [ xroot ].rank < subsets [ yroot ].rank)
        subsets [ xroot ].parent = yroot ;
    else if ( subsets [ xroot ].rank > subsets [ yroot ].rank )
        subsets [ yroot ].parent = xroot ;
    else
    {
        subsets [ yroot ].parent = xroot ;
        subsets [ xroot ].rank++ ;
    }
}
```

12/47. Bitwise operations:

-> if x & (x – 1) = 0 then number is power of 2
-> (x | 1 << n) returns the number x with nth bit set
-> x ^ (1 << n) toggles the state of the nth bit in the number x

13/47. Euler tour:

-> for undirected graph:
    all vertices having non-zero degree must be connected.
    Eulerian path- 0 or 2 vertices can have odd degree.
    Eulerian cycle- all vertices must have even degree.
-> for directed graph
    //these conditions are only for finding existence of Eulerian cycle
    *all vertices having non-zero degree belong to a single strongly connected components.
    *indegree and outdegree for every vertex must be same.

14/47. Tarjan's Algorithms for finding Articulation points:

```
#define NIL -1
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
```

```
}
 void Graph : : APUtil ( int u, bool visited [ ] , int disc [ ] , int low [ ] , int parent [ ] , bool ap [ ] )
{
    static int time = 0;
    int children = 0;
    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()
{
    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();
    return 0;
}
```

15/47. Subsets of a subset generation:

```
x=n ;    //n is representing the subset
while ( 1 )
{
    cout << x ;
    if ( x == 0 )
        break ;
    x = ( x -1 ) & n ;
}
```

16/47. Biconnected graph:

graph is biconnected if:
->graph is connected.
->there is no articulation point in the graph.

17/47. Matrix chain multiplication:

```
// Matrix Ai has dimension p[i-1] x p[i] for i = 1..n
int MatrixChainOrder(int p[], int n)
{
    int i, j, k, L, q;

   /* m[i,j] = Minimum no. of scalar multiplications needed to compute the matrix A[i]A[i+1]...A[j] = A[i..j] */
    for (i = 1; i < n; i++)
        m[i][i] = 0;
    // L is chain length.
    for (L=2; L<n; L++)
    {
        for (i=1; i<=n-L; i++)
        {
                j = i+L-1;
                m[i][j] = INT_MAX;
                for (k=i; k<=j-1; k++)
                {
                    // q = cost/scalar multiplications
                    q = m[i][k] + m[k+1][j] + p[i-1]*p[k]*p[j];
                    if (q < m[i][j])
                    m[i][j] = q;
                }
```

```
        }
    }
    return m[1][n-1];
}
```

18/47. Hamiltonian path:

```
bool isSafe(int v, bool graph[V][V], int path[], int pos)
{
    /* Check if this vertex is an adjacent vertex of the previously added vertex. */
    if (graph [ path[pos-1] ][ v ] == 0)
        return false;

    /* Check if the vertex has already been included.This step can be optimized by creating an array of size V */
    for (int i = 0; i < pos; i++)
        if (path[i] == v)
                return false;

    return true;
}

/* A recursive utility function to solve hamiltonian cycle problem */
bool hamCycleUtil(bool graph[V][V], int path[], int pos)
{
    /* base case: If all vertices are included in Hamiltonian Cycle */
    if (pos == V)
    {
        // And if there is an edge from the last included vertex to the first vertex
        if ( graph[ path[pos-1] ][ path[0] ] == 1 )
                return true;
        else
            return false;
    }

    // Try different vertices as a next candidate in Hamiltonian Cycle.
    // We don't try for 0 as we included 0 as starting point is in hamCycle()

    for (int v = 1; v < V; v++)
    {
        /* Check if this vertex can be added to Hamiltonian Cycle */
        if (isSafe(v, graph, path, pos))
        {
                path[pos] = v;
                /* recur to construct rest of the path */
                if (hamCycleUtil (graph, path, pos+1) == true)
                    return true;
                /* If adding vertex v doesn't lead to a solution,then remove it */
                path[pos] = -1;
        }
    }
   /* If no vertex can be added to Hamiltonian Cycle constructed so far then return false */
    return false;
}
```

19/47. KMP:

```
void KMPSearch(char *pat, char *txt)
{
    int M = strlen(pat);
    int N = strlen(txt);
```

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

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

20/47. Suffix Array [O(nlogn)]:

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

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

void sorting(int k)
{
    int counti[50005]={0},flag;
    int i,j,l;
    for(i=0;i<k;i++)
    {
        M[i]=N[i];
        counti[M[i].cm[1]+1]+=1;
    }
    for(i=1;i<=50004;i++)
        counti[i]+=counti[i-1];
    for(i=k-1;i>=0;i--)
    {
        N[counti[M[i].cm[1]+1]-1]=M[i];
        counti[M[i].cm[1]+1] -= 1 ;
    }
    for(i=0;i<50005;i++)
        counti[i]=0;
    for(i=0;i<k;i++)
    {
        M[i]=N[i];
        counti[M[i].cm[0]+1]+=1;
    }
    for(i=1;i<=50004;i++)
        counti[i]+=counti[i-1];
    for(i=k-1;i>=0;i--)
    {
        N[counti[M[i].cm[0]+1]-1]=M[i];
        counti[M[i].cm[0]+1]-=1;
    }
}



void longest_common_prefix(int k)
{
    int i,j,l,m,x,y;
    int ans=0;
    lcp[0]=0;
    stp-=1;
    for(i=0,j=1;j<k;i+=1,j+=1)
    {
        l=stp;
        ans=0;
        x=sa[i];
        y=sa[j];
        while(l>=0&&x<k&&y<k)
        {
                if(p[l][x]==p[l][y])
                {
                    ans+=1<<l;
                    x+=1<<l;
                    y+=1<<l;
                }
                l-=1;
        }
```

```
        lcp[j]=ans;
    }
}

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

21/47. Fermat's Little Theorem:

If p is a prime number and a is a positive number less than p
then

a^(p-1)=1(mod p)

22/47. Miller-Rabin primality test:

```
bool Miller(int p,int iteration)
{
    if(p<2)
    {
        return false;
    }
    if(p!=2 && p%2==0)
    {
        return false;
    }
    int s=p-1;
    while(s%2==0)
    {
        s/=2;
    }
    for(int i=0;i<iteration;i++)
    {
        int a=rand()%(p-1)+1,temp=s;
        int mod=power(a,temp,p);
        while(temp!=p-1 && mod!=1 && mod!=p-1)
        {
                mod=mulmod(mod,mod,p);
```

```
            temp *= 2;
        }
        if(mod!=p-1 && temp%2==0)
        {
            return false;
        }
    }
    return true;
}
```

23/47. BIT [point update-range query]:

```
int BIT[100000];
void initializeBIT(int a[],int n)
 {
    int i,j,k,l;
    for(i=0;i<=n;i++)
    {
        BIT[i]=0;
    }
    for(i=1;i<=n;i++)
    {
        int value_to_be_added=a[i-1];
        k=i;
        while(k<=n)
        {
            BIT[k]+=value_to_be_added;
            k+=(k&(-k));
        }
    }
 }



void update(int index,int value,int n)
 {
    int i,j,k;
    int index_to_modify=index+1;
    while(index_to_modify<=n)
    {
        BIT[index_to_modify]+=value;
        index_to_modify+=(index_to_modify&(-index_to_modify)          );
    }
 }

 int query(int i,int n)
 {
    int ans=0;
    int index_till=i+1;
    while(index_till>0)
    {
        ans+=BIT[index_till];
        index_till-=(index_till&(-index_till));
    }
    return ans;
 }
```

24/47. BIT [Range update-point query]:

Given an array A of N numbers, we need to support adding a value v to each element A[a...b] and querying the value of A[p], both operations in O(log N). Let ft[N+1] denote the underlying fenwick tree.

```
# Add v to A[p]
update(p, v):
   for (; p <= N; p += p&(-p))
      ft[p] += v

# Add v to A[a...b]
update(a, b, v):
   update(a, v)
   update(b + 1, -v)

# Return A[b]
query(b):
   sum = 0
   for(; b > 0; b -= b&(-b))
      sum += ft[b]
   return sum
```

25/47. BIT [range update-range query]:

Given an array A of N numbers, we need to support adding a value v to each element A[a...b] and querying the sum of numbers A[a...b], both operations in O(log N). This can be done by using two BITs B1[N+1], B2[N+1].

```
update(ft, p, v):
   for (; p <= N; p += p&(-p))
      ft[p] += v

# Add v to A[a...b]
update(a, b, v):
   update(B1, a, v)
   update(B1, b + 1, -v)
   update(B2, a, v * (a-1))
   update(B2, b + 1, -v * j)

query(ft, b):
   sum = 0
   for(; b > 0; b -= b&(-b))
      sum += ft[b]
   return sum

# Return sum A[1...b]
query(b):
   return query(B1, b) * b - query(B2, b)

# Return sum A[a...b]
query(a, b):
   return query(b) - query(a-1)
```

Explanation:

BIT B1 is used like in the earlier case with range updates/point queries such that query(B1, p) gives A[p].
Consider a range update query: Add v to [a...b]. Let all elements initially be 0. Now, Sum(1...p) for different p is as follows:
1 <= p < a : 0
a <= p <= b : v * (p – (a – 1))
b < p <= N : v * (b – (a – 1))
Thus, for a given index p, we can find Sum(1...p) by subtracting a value X from p * Sum(p,p) (Sum(p,p) is the actual value stored at index p) such that
1 <= p < a : Sum(1..p) = 0, X = 0
a <= p <= b : Sum(1...p) = (v * p) – (v * (a-1)), X = v * (a-1)
b < p <= N: Sum(1...p) = (v * b) – (v * (a-1)), X = -(v * b) + (v * (a-1))
To maintain this extra factor X, we use another BIT B2 such that
Add v to [a...b] -> Update(B2, a, v * (a-1)) and Update(B2, b+1, -v * b)
Query(B2, p) gives the value X that must be subtracted from A[p] * p

26/47. Lazy Propagation:

```
int update(int ind,int st,int se,int qs,int qe,int num)
{
    if(st>se)
        return 0;
    if(lazy[ind]!=0)
    {
        BIT[ind]+=(lazy[ind]*(se-st+1));
        if(st!=se)
        {
            lazy[2*ind+1]+=lazy[ind];
            lazy[2*ind+2]+=lazy[ind];
        }
        lazy[ind]=0;
    }

    if(qs>se||qe<st)
        return 0;
    if(qs<=st&&qe>=se)
    {
        BIT[ind]+=(num*(se-st+1));
        if(st!=se)
        {
            lazy[2*ind+1]+=num;
            lazy[2*ind+2]+=num;
        }
        return BIT[ind];
    }
    else
    {
        int mid=(st+se)/2,p,q;
        update(2*ind+1,st,mid,qs,qe,num);
        update(2*ind+2,mid+1,se,qs,qe,num);
        BIT[ind]=BIT[2*ind+1]+BIT[2*ind+2];
        return BIT[ind];
    }
}

int query(int ind,int st,int se,int qs,int qe)
{
    if(st>se)
        return 0;
    if(lazy[ind]!=0)
    {
            BIT[ind]+=(lazy[ind]*(se-st+1));
            if(st!=se)
            {
                lazy[2*ind+1]+=lazy[ind];
                lazy[2*ind+2]+=lazy[ind];
            }
            lazy[ind]=0;
    }
    if(qs>se||qe<st)
        return 0;
    if(qs<=st&&qe>=se)
    {
        return BIT[ind];
    }
    else
    {
        int mid=(st+se)/2,p,q;
```

```
        p=query(2*ind+1,st,mid,qs,qe);
        q=query(2*ind+2,mid+1,se,qs,qe);
        return p+q;
    }
}
```

27/47. Permutations of a string:

```
void permute(char* str,char *per,int ind,int len)
{
    if(ind==len)
    {
        cout<< per;
        return;
    }
    for(int i=0;i<len;i++)
    {
        int j,var=1;
        for(j=0;j<ind;j++)
        {
            if(per[j]==str[i])
            {
                var=0;
                break;
            }
        }
        if(var)
        {
            per[ind]=str[i];
            permute(str,per,ind+1);
        }
    }
}

int main()
{
    char str[]="ABCD";
    int len=strlen(str);
    char per[len+1];
    per[len]='\0';
    permute(str,per,0,len);
    getchar();
    return 0;
}
```

28/47. Sums:

$\sum_{k=0}^{n} k = n(n+1)/2$

$\sum_{k=0}^{n} k^2 = n(n+1)(2n+1)/6$

$\sum_{k=0}^{n} k^4 = (6n^5 + 15n^4 + 10n^3 - n)/30$

$\sum_{k=0}^{n} x^k = (x^{n+1} - 1)/(x - 1)$

$1 + x + x^2 + \cdots = 1/(1 - x)$

$\sum_{k=a}^{b} k = (a+b)(b-a+1)/2$

$\sum_{k=0}^{n} k^3 = n^2(n+1)^2/4$

$\sum_{k=0}^{n} k^5 = (2n^6 + 6n^5 + 5n^4 - n^2)/12$

$\sum_{k=0}^{n} kx^k = (x - (n+1)x^{n+1} + nx^{n+2})/(x-1)^2$

29/47. Harmonic Progression:

$$\sum_{k=1}^{n} 1/k = ln(n) + 0.577215664901532 + 1/(2n) - 1/(12n^2) + 1/(120n^4) \ldots$$

30/47. nCr:

$\binom{n}{r}$ mod p equals the product of $\binom{n_i}{r_i}$ mod p where
$n_1 n_2 \ldots$ and $r_1 r_2 \ldots$ are the digits of n and r when written in base p.
(e.g - $\binom{21}{3}$ mod 17 equals $\binom{1}{0} * \binom{4}{3}$ mod 17.)

31/47. Prime counting function:

f(n)=|{p<=n : p is prime}|
n/ln(n) < f(n) < 1.3n/ln(n)
eg. f(1000)=168, f(10^6)=78498
nth prime= nln(n)

32/47. Number of divisiors:

$$\tau(p_1^{a_1} \ldots p_k^{a_k}) = \prod_{j=1}^{k} (a_j + 1).$$

33/47. Sum of divisiors:

$$\sigma(p_1^{a_1} \ldots p_k^{a_k}) = \prod_{j=1}^{k} \frac{p_j^{a_j+1} - 1}{p_j - 1}.$$

34/47. Wilson's theorem:

p is prime iff (p-1)! = -1(mod p)

35/47. Postage stamps/McNuggets theorem:

Let a and b be relatively prime integers. There are exactly
(a-1)*(b-1)/2 numbers not of the form ax+by (x,y>=0), and the largest is ab-a-b.

36/47. Area of a polygon:

$$\frac{1}{2} \sum_{i=0}^{n-1} (x_i y_{i+1} - x_{i+1} y_i)$$

where Xn=Xo, Yn=Yo. Area is negative if boundry is oriented clockwise.

37/47. 3-D Figures:

Sphere	Volume $V = \frac{4}{3}\pi r^3$, surface area $S = 4\pi r^2$
	$x = \rho \sin\theta \cos\phi$, $y = \rho \sin\theta \sin\phi$, $z = \rho\cos\theta$, $\phi \in [-\pi, \pi]$, $\theta \in [0, \pi]$
Spherical section	Volume $V = \pi h^2 (r - h/3)$, surface area $S = 2\pi r h$
Pyramid	Volume $V = \frac{1}{3} h S_{base}$
Cone	Volume $V = \frac{1}{3}\pi r^2 h$, lateral surface area $S = \pi r \sqrt{r^2 + h^2}$

38/47. Fermat's two-squares theorem:

Odd prime p can be represented as a sum of two squares iff p=1(mod 4). A product of two sums of two squares is a sum of two squares. Thus, n is a sum of two squares i_ every prime of form p = 4k + 3 occurs an even number of times in n's factorization.

39/47. Manacher's Algorithm [Longest pallindromic substring]:

```
//S is the input string
// Transform S into T.
// For example, S = "abba", T = "^#a#b#b#a#$".
// ^ and $ signs are sentinels appended to each end to avoid bounds checking
string preProcess(string s)
{
    int n = s.length();
    if (n == 0) return "^$";
        string ret = "^";
    for (int i = 0; i < n; i++)
        ret += "#" + s.substr(i, 1);
    ret += "#$";
    return ret;
}

string longestPalindrome(string s)
{
    string T = preProcess(s);
    int n = T.length();
    int *P = new int[n];
    int C = 0, R = 0;
    for (int i = 1; i < n-1; i++)
    {
        int i_mirror = 2*C-i; // equals to i' = C - (i-C)
        P[i] = (R > i) ? min(R-i, P[i_mirror]) : 0;
        // Attempt to expand palindrome centered at i
        while (T[i + 1 + P[i]] == T[i - 1 - P[i]])
            P[i]++;
        // If palindrome centered at i expand past R,
        // adjust center based on expanded palindrome.
        if (i + P[i] > R)
        {
            C = i;
            R = i + P[i];
        }
    }

    // Find the maximum element in P.
    int maxLen = 0;
    int centerIndex = 0;
    for (int i = 1; i < n-1; i++)
    {
        if (P[i] > maxLen)
        {
            maxLen = P[i];
            centerIndex = i;
        }
    }
    delete[] P;
    return s.substr((centerIndex - 1 - maxLen)/2, maxLen);
}
```

40/47. Tarjan's Offline LCA Algorithm:

```
DFS(x):
    ancestor[Find(x)] = x
    for all children y of x:
        DFS(y); Union(x, y); ancestor[Find(x)] = x
    seen[x] = true
    for all queries {x, y}:
```

```
        if seen[y] then output "LCA(x, y) is ancestor[Find(y)]"
```

LCA:

```
void precomp(int N, int T[MAXN], int P[MAXN][LOGMAXN])
{
    int i, j;
    //we initialize every element in P with -1
    for (i = 0; i < N; i++)
        for (j = 0; 1 << j < N; j++)
            P[i][j] = -1;
    //the first ancestor of every node i is T[i]
    for (i = 0; i < N; i++)
        P[i][0] = T[i];
    //bottom up dynamic programing
    for (j = 1; 1 << j < N; j++)
        for (i = 0; i < N; i++)
            if (P[i][j - 1] != -1)
                P[i][j] = P[P[i][j - 1]][j - 1];
}

int query(int N, int P[MAXN][LOGMAXN], int T[MAXN], int L[MAXN], int p, int q)
{
    int tmp, log, i;
    //if p is situated on a higher level than q then we swap them
    if (L[p] < L[q])
        tmp = p, p = q, q = tmp;
    //we compute the value of [log(L[p)]
    for (log = 1; 1 << log <= L[p]; log++);
    log--;
    //we find the ancestor of node p situated on the same level with q using the values in P
    for (i = log; i >= 0; i--)
        if (L[p] - (1 << i) >= L[q])
            p = P[p][i];
    if (p == q)
        return p;
    //we compute LCA(p, q) using the values in P
    for (i = log; i >= 0; i--)
        if (P[p][i] != -1 && P[p][i] != P[q][i])
            p = P[p][i], q = P[q][i];
    return T[p];
}
```

41/47. Dinic's Blocking Flow Algorithm:

```
// Adjacency list implementation of Dinic's blocking flow algorithm.
// This is very fast in practice, and only loses to push-relabel flow.
//
// Running time:
//     O(|V|^2 |E|)
//
// INPUT:
//     - graph, constructed using AddEdge()
//     - source
//     - sink
//
// OUTPUT:
//     - maximum flow value
//     - To obtain the actual flow values, look at all edges with
//       capacity > 0 (zero capacity edges are residual edges).
```

```
using namespace std;
const int INF = 2000000000;
struct Edge
{
    int from, to, cap, flow, index;
    Edge(int from, int to, int cap, int flow, int index) :
        from(from), to(to), cap(cap), flow(flow), index(index) {}
};

struct Dinic
{
    int N;
    vector<vector<Edge> > G;
    vector<Edge *> dad;
    vector<int> Q;
    Dinic(int N) : N(N), G(N), dad(N), Q(N) {}
    void AddEdge(int from, int to, int cap)
    {
        G[from].push_back(Edge(from, to, cap, 0, G[to].size()));
        if (from == to) G[from].back().index++;
        G[to].push_back(Edge(to, from, 0, 0, G[from].size() - 1));
    }

    long long BlockingFlow(int s, int t)
    {
        fill(dad.begin(), dad.end(), (Edge *) NULL);
        dad[s] = &G[0][0] - 1;
        int head = 0, tail = 0;
        Q[tail++] = s;
        while (head < tail)
        {
            int x = Q[head++];
            for (int i = 0; i < G[x].size(); i++)
            {
                Edge &e = G[x][i];
                if (!dad[e.to] && e.cap - e.flow > 0)
                {
                    dad[e.to] = &G[x][i];
                    Q[tail++] = e.to;
                }
            }
        }
        if (!dad[t]) return 0;
        long long totflow = 0;
        for (int i = 0; i < G[t].size(); i++)
        {
            Edge *start = &G[G[t][i].to][G[t][i].index];
            int amt = INF;
            for (Edge *e = start; amt && e != dad[s]; e = dad[e->from])
            {
                if (!e)
                {
                    amt = 0; break;
                }
                amt = min(amt, e->cap - e->flow);
            }
            if (amt == 0) continue;
            for (Edge *e = start; amt && e != dad[s]; e = dad[e->from])
            {
                e->flow += amt;
                G[e->to][e->index].flow -= amt;
```

```
            }
            totflow += amt;
        }
        return totflow;
    }

    long long GetMaxFlow(int s, int t)
    {
        long long totflow = 0;
        while (long long flow = BlockingFlow(s, t))
            totflow += flow;
        return totflow;
    }
};
```

42/47. Min Cost Max Flow:

```
// Implementation of min cost max flow algorithm using adjacency
// matrix (Edmonds and Karp 1972).  This implementation keeps track of
// forward and reverse edges separately (so you can set cap[i][j] !=
// cap[j][i]).  For a regular max flow, set all edge costs to 0.
//
// Running time, O(|V|^2) cost per augmentation
//     max flow:           O(|V|^3) augmentations
//     min cost max flow:  O(|V|^4 * MAX_EDGE_COST) augmentations
//
// INPUT:
//     - graph, constructed using AddEdge()
//     - source
//     - sink
//
// OUTPUT:
//     - (maximum flow value, minimum cost value)
//     - To obtain the actual flow, look at positive values only.

typedef vector<int> VI;
typedef vector<VI> VVI;
typedef long long L;
typedef vector<L> VL;
typedef vector<VL> VVL;
typedef pair<int, int> PII;
typedef vector<PII> VPII;

const L INF = numeric_limits<L>::max() / 4;

struct MinCostMaxFlow
{
    int N;
    VVL cap, flow, cost;
    VI found;
    VL dist, pi, width;
    VPII dad;
    MinCostMaxFlow(int N) :
    N(N), cap(N, VL(N)), flow(N, VL(N)), cost(N, VL(N)),
    found(N), dist(N), pi(N), width(N), dad(N) {}

    void AddEdge(int from, int to, L cap, L cost)
    {
        this->cap[from][to] = cap;
        this->cost[from][to] = cost;
    }
```

```
    void Relax(int s, int k, L cap, L cost, int dir)
    {
        L val = dist[s] + pi[s] - pi[k] + cost;
        if (cap && val < dist[k])
        {
            dist[k] = val;
            dad[k] = make_pair(s, dir);
            width[k] = min(cap, width[s]);
        }
    }

    L Dijkstra(int s, int t)
    {
        fill(found.begin(), found.end(), false);
        fill(dist.begin(), dist.end(), INF);
        fill(width.begin(), width.end(), 0);
        dist[s] = 0;
        width[s] = INF;
        while (s != -1)
        {
            int best = -1;
            found[s] = true;
            for (int k = 0; k < N; k++)
            {
                if (found[k]) continue;
                Relax(s, k, cap[s][k] - flow[s][k], cost[s][k], 1);
                Relax(s, k, flow[k][s], -cost[k][s], -1);
                if (best == -1 || dist[k] < dist[best]) best = k;
            }
            s = best;
        }
        for (int k = 0; k < N; k++)
            pi[k] = min(pi[k] + dist[k], INF);
        return width[t];
    }

    pair<L, L> GetMaxFlow(int s, int t)
    {
        L totflow = 0, totcost = 0;
        while (L amt = Dijkstra(s, t))
        {
            totflow += amt;
            for (int x = t; x != s; x = dad[x].first)
            {
                if (dad[x].second == 1)
                {
                    flow[dad[x].first][x] += amt;
                    totcost += amt * cost[dad[x].first][x];
                }
                else
                {
                    flow[x][dad[x].first] -= amt;
                    totcost -= amt * cost[x][dad[x].first];
                }
            }
        }
        return make_pair(totflow, totcost);
    }
};
```

43/47. PushRelabel:

```
// Adjacency list implementation of FIFO push relabel maximum flow
// with the gap relabeling heuristic.  This implementation is
// significantly faster than straight Ford-Fulkerson.  It solves
// random problems with 10000 vertices and 1000000 edges in a few
// seconds, though it is possible to construct test cases that
// achieve the worst-case.
//
// Running time:
//     O(|V|^3)
//
// INPUT:
//     - graph, constructed using AddEdge()
//     - source
//     - sink
//
// OUTPUT:
//     - maximum flow value
//     - To obtain the actual flow values, look at all edges with
//       capacity > 0 (zero capacity edges are residual edges).

typedef long long LL;

struct Edge
{
    int from, to, cap, flow, index;
    Edge(int from, int to, int cap, int flow, int index) :
    from(from), to(to), cap(cap), flow(flow), index(index) {}
};

struct PushRelabel
{
    int N;
    vector<vector<Edge> > G;
    vector<LL> excess;
    vector<int> dist, active, count;
    queue<int> Q;
    PushRelabel(int N) : N(N), G(N), excess(N), dist(N), active(N), count(2*N) {}

    void AddEdge(int from, int to, int cap)
    {
        G[from].push_back(Edge(from, to, cap, 0, G[to].size()));
        if (from == to) G[from].back().index++;
        G[to].push_back(Edge(to, from, 0, 0, G[from].size() - 1));
    }

    void Enqueue(int v)
    {
        if (!active[v] && excess[v] > 0) { active[v] = true; Q.push(v); }
    }

    void Push(Edge &e)
    {
        int amt = int(min(excess[e.from], LL(e.cap - e.flow)));
        if (dist[e.from] <= dist[e.to] || amt == 0) return;
        e.flow += amt;
        G[e.to][e.index].flow -= amt;
        excess[e.to] += amt;
        excess[e.from] -= amt;
        Enqueue(e.to);
    }
```

```
    void Gap(int k)
    {
        for (int v = 0; v < N; v++)
        {
            if (dist[v] < k) continue;
            count[dist[v]]--;
            dist[v] = max(dist[v], N+1);
            count[dist[v]]++;
            Enqueue(v);
        }
    }

    void Relabel(int v)
    {
        count[dist[v]]--;
        dist[v] = 2*N;
        for (int i = 0; i < G[v].size(); i++)
            if (G[v][i].cap - G[v][i].flow > 0)
                dist[v] = min(dist[v], dist[G[v][i].to] + 1);
        count[dist[v]]++;
        Enqueue(v);
    }

    void Discharge(int v)
    {
        for (int i = 0; excess[v] > 0 && i < G[v].size(); i++) Push(G[v][i]);
        if (excess[v] > 0)
        {
            if (count[dist[v]] == 1)
                Gap(dist[v]);
            else
                Relabel(v);
        }
    }

    LL GetMaxFlow(int s, int t)
    {
        count[0] = N-1;
        count[N] = 1;
        dist[s] = N;
        active[s] = active[t] = true;
        for (int i = 0; i < G[s].size(); i++)
        {
            excess[s] += G[s][i].cap;
            Push(G[s][i]);
        }
        while (!Q.empty())
        {
            int v = Q.front();
            Q.pop();
            active[v] = false;
            Discharge(v);
        }
        LL totflow = 0;
        for (int i = 0; i < G[s].size(); i++) totflow += G[s][i].flow;
        return totflow;
    }
};
```

44/47. Maximum Bipartite Matching:

```
// This code performs maximum bipartite matching.
//
// Running time: O(|E| |V|) -- often much faster in practice
//
//   INPUT: w[i][j] = edge between row node i and column node j
//   OUTPUT: mr[i] = assignment for row node i, -1 if unassigned
//           mc[j] = assignment for column node j, -1 if unassigned
//           function returns number of matches made

typedef vector<int> VI;
typedef vector<VI> VVI;

bool FindMatch(int i, const VVI &w, VI &mr, VI &mc, VI &seen)
{
    for (int j = 0; j < w[i].size(); j++)
    {
        if (w[i][j] && !seen[j])
        {
            seen[j] = true;
            if (mc[j] < 0 || FindMatch(mc[j], w, mr, mc, seen))
            {
                mr[i] = j;
                mc[j] = i;
                return true;
            }
        }
    }
    return false;
}

int BipartiteMatching(const VVI &w, VI &mr, VI &mc)
{
    mr = VI(w.size(), -1);
    mc = VI(w[0].size(), -1);
    int ct = 0;
    for (int i = 0; i < w.size(); i++)
    {
        VI seen(w[0].size());
        if (FindMatch(i, w, mr, mc, seen)) ct++;
    }
    return ct;
}
```

45/47. Pollard Rho:

```
/* This is a pseudo random number genrator modulo n*/
LL f(LL x, LL n)
{
    LL temp = mulmod(x, x, n);
    temp = mulmod( alpha, temp, n);
    return ( temp + beta) % n;
}
/* Function returns a non trivial factor of the number N.
 Make sure before calling this that N is not a prime using miller rabin*/
LL pollardRho(LL N)
{
    LL x, y, d;
    while(true)
    {
        x = 2, y = 2, d = 1;
```

```
        alpha = (rand() % (N -1)) + 1;
        beta = (rand() % (N -1)) + 1;
        while( d == 1)
        {
            x = f(x,N);
            y = f( f(y,N), N);
            d = gcd( abs(x - y), N);
        }
        if( d != N) break;
    }
    assert(N % d == 0);
    // if this condition fails, consider increasing max iter and check if N is prime
    return d;
}
```

46/47. MinCut:

```
// Adjacency matrix implementation of Stoer-Wagner min cut algorithm.
// Running time:
//     O(|V|^3)
// INPUT:
//     - graph, constructed using AddEdge()
// OUTPUT:
//     - (min cut value, nodes in half of min cut)

typedef vector<int> VI;
typedef vector<VI> VVI;

const int INF = 1000000000;
pair<int, VI> GetMinCut(VVI &weights)
{
    int N = weights.size();
    VI used(N), cut, best_cut;
    int best_weight = -1;
    for (int phase = N-1; phase >= 0; phase--)
    {
        VI w = weights[0];
        VI added = used;
        int prev, last = 0;
        for (int i = 0; i < phase; i++)
        {
            prev = last;
            last = -1;
            for (int j = 1; j < N; j++)
                if (!added[j] && (last == -1 || w[j] > w[last])) last = j;
            if (i == phase-1)
            {
                for (int j = 0; j < N; j++) weights[prev][j] += weights[last][j];
                for (int j = 0; j < N; j++) weights[j][prev] = weights[prev][j];
                used[last] = true;
                cut.push_back(last);
                if (best_weight == -1 || w[last] < best_weight)
                {
                    best_cut = cut;
                    best_weight = w[last];
                }
            }
            else
            {
                for (int j = 0; j < N; j++)
                    w[j] += weights[last][j];
```

```
                    added[last] = true;
                }
            }
        }
        return make_pair(best_weight, best_cut);
}
```

47/47. Eulerian path:

```
int graph[100][100];
int n, x, y, steps;
list<int> path;

void walk(int pos)
{
    for(int i = 0; i < n; i++)
    {
        if(graph[pos][i] > 0)
        {
            graph[pos][i] --;
            graph[i][pos] --;
            walk(i);
            break;
        }
    }
    path.push_back(pos+1);
}

int main()
{
    cin >> n;
    for(int i = 0; i < n; i++)
    {
        cin >> x >> y;
        graph[x-1][y-1] ++; //we are using zero index
    }
    walk(0);
    while(!path.empty())
    {
        cout << path.back() << ' ';
        path.pop_back();
    }
}
```