INDEX

MATH

EXTENDED GCD

```
int egcd(int a,int b)
{               // a = t*q + res;
    int x=0 , px = 1 , y = 1, py = 0;
    while(b != 0)
    {
        int temp = b;
        int q = a/b;
        b = a%b;
        a = temp;      //end of normal gcd
        temp = x;      //find x (return py)
        x = px - q*x;
        px = temp;
        temp = y;      //find y (return px)
        y = py - q*y;
        py = temp;
    }
    return a;
}
```

SIEVE OF ERATHOSTHENES

```
#define MAXP 1000000000
#define SIZE (MAXP+31)/32
#define isprime(n) (sieve[n>>5]&(1<<(n&31))) //is the bit on
#define stprime(n) (sieve[n>>5]&=~(1<<(n&31))) //turn bit on
vector<int> Primes;
int sieve[SIZE];
void generate()
{
    memset(sieve,-1,sizeof(sieve));stprime(2);stprime(3);
    for(int i = 5, s = 0; i < MAXP ; i += (1<<(s+1)),s = 1-s)
    {
        if(isprime(i) == 0)continue;
        Primes.push_back(i);
        if(1LL*i*i > MAXP)continue;
        for(int j = i*i ; j < MAXP ; j += i)stprime(j);
    }
}
```

LUCAS THEOREM

```
Long NChooseK(int N,int K,int M)
{
        if( N<0 || K < 0 || K > N)return 0;
        K = min( K , N-K );
        Long ret = 1;
        for(int i = N , j = 1 ; j <= K ; ++j , --i)
        {
                Long div_inverse = egcd( j , M ); // (return py)
                ret =( (ret*div_inverse)%M * i)%M;
                if( ret < 0 ) ret += M;
        }
        return ret;
}

Long NChooseKmod(int N,int K,int M)
{
        Long ret = 1;
        for(int i = N, j = K; j > 0 ; j /= M , i /= M)
        {
                ret = (ret*NChooseK( i%M , j%M , M ))%M;
        }
        return ret;
}
```

STRING PROCESSING

KMP

```
void compute_T(string &A,vector<int> &T)
{
    int j = -1, i = 0;
    T[0] = -1;
    while(i < (int)A.size())
    {
        while(j>=0 && A[i] != A[j])j = T[j];
        T[++i] = ++j;
    }
}

void search(string &Hay , string &Needle,vector<int> &T)
{
```

```
    int j = 0, i = 0; //j -> Needle , i -> Hay
    while(i < (int)Hay.size())
    {
        while(j>=0 && Hay[i] != Needle[j])j = T[j];
        i++; j++;
        if(j == (int)Needle.size())
        { // found (do something)
            j = T[j];
        }
    }
}
```

Hash

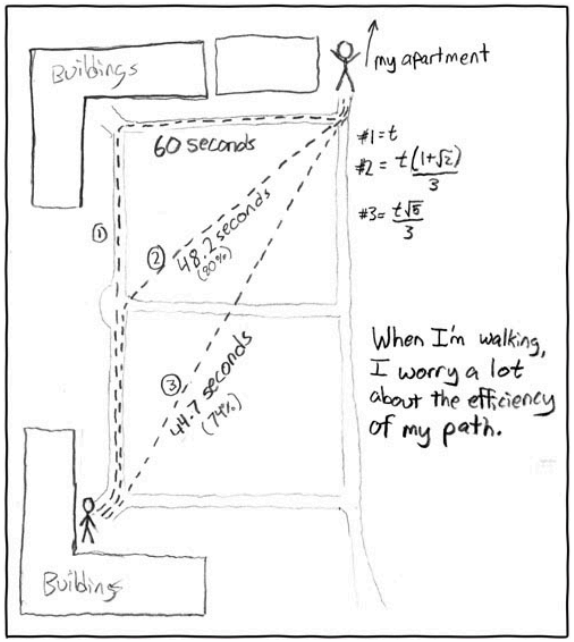
```
#include<ext/hash_map>
using __gnu_cxx::hash_map;
struct Hash //hash_map<string , int , Hash>
{
    size_t operator() (const string &K)const
    {
        size_t idx = 0;
        for(int i = 0 ; i < K.size(); ++i)
            idx = 33*idx + K[i];
        return idx;
    }
};
```

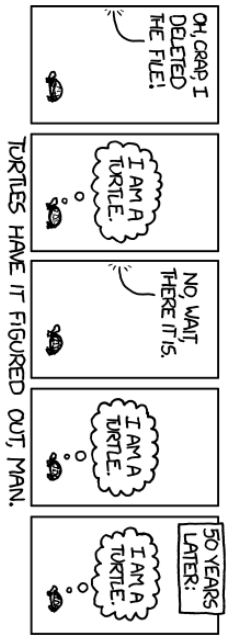
AHO CORASICK

```
struct State {
   int pat_id, outnxt, fail;
   int edges[26];
   State() : pat_id(-1), outnxt(-1), fail(0) {
      memset(edges, -1, sizeof(edges));
   }
};

struct AhoCorasick {
#define ROOT 0
   vector<State> nodes;
// vector<string> patterns;
   AhoCorasick(int npat, const char *pat[]) {
      nodes.push_back(State());   // root
      // 1. Construct keyword tree for each pattern
      for (int i = 0; i < npat; ++i) {
         int v = ROOT;
         for (const char *p = pat[i]; *p; ++p) {
            int k = *p-'a';
            if (nodes[v].edges[k] < 0) {
               nodes[v].edges[k] = nodes.size();
               nodes.push_back(State());
            }
            v = nodes[v].edges[k];
         }
         nodes[v].pat_id = i;  // set pattern id of terminating node
      }
      // 2. Complete the goto function for missing transitions from root
      for (int k = 0; k < 26; ++k)
         if (nodes[ROOT].edges[k] < 0)
            nodes[ROOT].edges[k] = ROOT;
      // 3. Compute failure and output functions in BFS order
      queue<int> q;
      for (int k = 0; k < 26; ++k) {
         int u = nodes[ROOT].edges[k];
         if (u != 0) {
            nodes[u].fail = ROOT;
            q.push(u);
         }
      }

      while (!q.empty()) {
         int r = q.front();
         q.pop();
         for (int k = 0; k < 26; ++k) {
            int u = nodes[r].edges[k];
            if (u < 0) continue;
            q.push(u);
            int v = nodes[r].fail;
            while (nodes[v].edges[k] < 0)
               v = nodes[v].fail;
            nodes[u].fail = nodes[v].edges[k];
            nodes[u].outnxt = nodes[nodes[u].fail].pat_id ?
                  nodes[u].fail : nodes[nodes[u].fail].outnxt;
         }
      }
   }
   void find(const char *S) {
      int q = ROOT;
```

```
      for (int i = 0; S[i]; ++i) {
         int k = S[i]-'a';
         while (nodes[q].edges[k] < 0)
            q = nodes[q].fail;
         q = nodes[q].edges[k];
         if (nodes[q].pat_id >= 0) {
            // output all matching patterns stored in link list from
outnxt
            printf("Found match at %d: ", i);
            for (int x = q; x > 0; x = nodes[x].outnxt) {
               if (nodes[x].pat_id >= 0)
                  printf(" %d", nodes[x].pat_id);
            }
            putchar('\n');
         }
      }
   }
};

const char* pat[] = {
   "he", "she", "his", "hers"
};
```

FILLER: --> COMIC





THIS IS NOT STRING -----> PROGRAMMING IDEAS

String
- KMP Matching
- Prefix Processing (for segment)
- + DP

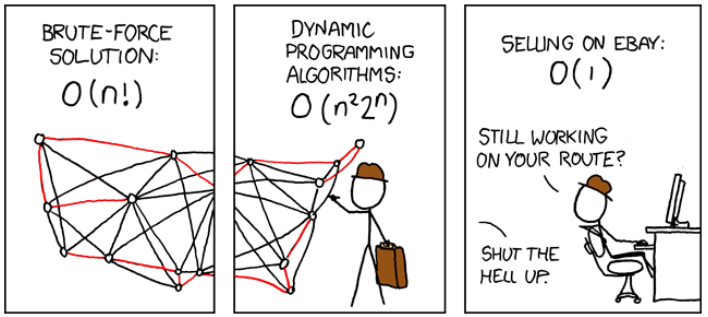
Tree
- Segment Tree (Prefix Sum , [maxsum,lo,hi])

Adhoc
- Double Ended Queue
- Offline Solution
- Subsequence accumulated + lower_bound
- Tracking Subsequence and updating next equal (DQUERY , GSS2)
- Bitmasking (one or more low constraint)
- Sorting + Binary Search
- Set + Lower Bound
- Brute Force (very low constraints)
- Formula (very high constraints)
- Head & Tail
- Entering & Exiting (events sorting)

Geometry
- Ordered Set for Counting
- Sweeps
- One coordinate sorting

Floating Point
- Small decimals written as INTEGERS

Tips
- Pigeon Whole principle (mod , limited spots)
- sprintf y sscanf (int -> str) (str -> int)



SUFFIX ARRAY

```
const int MAXN = 200010;
struct SuffixArray
{
    string A;
    int N;
    int SA[MAXN] , RA[MAXN] , LCP[MAXN];
    SuffixArray(string &B)
    {
        A = B; N = A.size();
        for(int i = 0; i < N; ++i)
            SA[i] = i , RA[i] = A[i];
    }
    void countingSort(int H)
    {
        int maxn = max(N,300)+N;
        int freq[MAXN*2+300] , nSA[N];  memset(freq,0,sizeof(freq));
        for(int i = 0; i < N; ++i)
            freq[ SA[i]+H<N ? RA[SA[i]+H]+N : N-1-SA[i] ]++;
        for(int i = 0,sum = 0,t; i < maxn; ++i)
            t = freq[i] , freq[i] = sum , sum += t;
        for(int i = 0,p,m; i < N ; ++i)
            nSA[ freq[ (SA[i]+H<N)?(RA[SA[i]+H]+N):(N-1-SA[i])]++ ] =SA
[i];
        memcpy(SA,nSA,sizeof(nSA));

    }
    void BuildSA()
    {
        for(int H = 1; H < A.size() ; H<<=1)
        {
            countingSort(H);
            countingSort(0);
            int nRA[N] , rank = nRA[ SA[0] ] = 0;
            for(int i = 1; i < N; ++i)
            {
                if(RA[ SA[i] ] != RA[ SA[i-1] ])rank++;
                else if(SA[i-1]+H >= N || SA[i]+H >= N)rank++;
                else if(RA[ SA[i]+H ] != RA[ SA[i-1]+H ])rank++;
                nRA[ SA[i] ] = rank;
            }
            memcpy(RA,nRA,sizeof(nRA));
        }
    }
    void BuildLCP()
    {
        int PLCP[MAXN];
        int PHI[MAXN];
        PHI[ SA[0] ] = -1;
        for(int i = 1; i < N; ++i)
            PHI[ SA[i] ] = SA[i-1];
        for(int i = 0, L = 0; i < N; ++i)
        {
            if(PHI[i]==-1){PLCP[i] = 0;continue;}
            while(PHI[i]+L < N && i+L < N && A[i+L] == A[ PHI[i]+L ])L+
+;
            PLCP[i] = L;
            L = max(L-1 , 0);
        }
        for(int i = 1; i < N; ++i)
            LCP[i] = PLCP[ SA[i] ];
    }
    pair<int,int> Match(string &B)
    {
        int lo = 0, hi = N-1;
        for(int idx = 0; idx < B.size(); ++idx)
        {
            while(SA[lo]+idx>=N || A[SA[lo]+idx]<B[idx])
            {
                if(lo<N && LCP[lo+1] > idx-1)lo++;
                else break;
            }
            while(SA[hi]+idx>=N || A[SA[hi]+idx]>B[idx])
            {
                if(hi>0 && LCP[hi] > idx-1)hi--;
                else break;
            }
        }
        return pair<int,int>(lo,hi);
    }
};
```

TREES

FENWICK TREE

```
void insert(int x,int val){
    while(x < 100000)FEN[x]
}
int query(int x){
    int ret = 0;
    while(x > 0)ret += FEN[x] ,
    return ret;
}
int find(int val)
{
    int bit = 0,cm = 100000 , ans
    while(cm)bit = cm, cm -= (cm
    while(bit)
    {
        int temp = ans+bit;
        if(temp<100000 && val >=
= bit;
        bit >>= 1;
    }
    return ans;
}
```

```
+=val , x += (x & -x);

x -= (x & -x);

= 0; //MAXN
& -cm);

FEN[temp])val -= FEN[temp], ans |
```

DISJOINT SET

```
struct NODE{
    int p,r,s;
    NODE(){par=-1 , rank=0 ,size
};
NODE set[100000]; //MAXN
int FIND(int x)
{
    if(set[x].p==-1)return x;
    return set[x].p = FIND(set
}
bool UNION(int x,int y)
{
    x=FIND(x), y=FIND(y);
    if(x==y)return false;
    if(set[x].r >= set[y].r)set
    if(set[x].r <  set[y].r)set
```

```
= 1;}

[x].p);

[y].p = x;
[x].p = y;
    if(set[x].r == set[y].r)set[x].r++;
    set[x].s = set[y].s = set[x].s+set[y].s;
    return true;
}
```

LOWEST COMMON ANCESTOR

```
//T -> Logarithmic Tree of Parents , L -> Array of Level
//Compute the level Tree with a DFS
int T[100000][20] , L[100000];
void create(int *P,int N)
{
    memset(T,-1,sizeof(T));
    for(int i = 0; i < N; ++i)
        T[i][0] = P[i];
    for(int lvl = 1; (1<<lvl) <= N; lvl++)
        for(int i = 0; i < N; ++i)
            if(T[i][lvl-1]!=-1)
                T[i][lvl] = T[T[i][lvl-1]][lvl-1];
}

int LCA(int x,int y)
{
    if(L[x] < L[y])swap(x,y);

    int log = 0;
    for(log = 1; (1<<log) <= L[x]; log++); log--;

    for(int i = log ; L[x] != L[y] ; i-- )
        if(L[x] - (1<<i) >= L[y])
            x = T[x][i];

    for(int i = log ; x!=y && i >= 0 ; i--)
        if( T[x][i] != T[y][i] )
            x = T[x][i] , y = T[y][i];

    return x==y ? x : T[x][0];
}
```

HEAVY LIGHT DECOMPOSTION

```
#define ROOT 0

struct NO
{
    int stChild,parent,path,pathIdx,size,lvl;
    NO(){stChild = -1 , parent = -1; lvl = 0;}
};
typedef NO Vertex;

int allEdges[10010];
int edgeV[10010];
VVI adj;
VVI edg;
Vertex List[10010];

struct Seg
{
    int id;
    VI Nodes, Edges, seg;
    Seg(){Nodes = VI();seg = VI();Edges = VI();}
    void createSegment(int node = 1, int b = 0,int e = -1)
    {
        if(e==-1)e = Edges.size() - 1;
        if(b == e)
            seg[node] = allEdges[Edges[b]];
        else
        {
            createSegment(2*node,b,(b+e)/2);
            createSegment(2*node+1,(b+e)/2+1,e);
            seg[node] = max(seg[2*node],seg[2*node+1]);
        }
    }
    int query(int i,int j,int node = 1, int b = 0,int e = -1)const
    {
        if(e==-1)e = Edges.size() - 1;
        if(i <= b && e <= j)
            return seg[node];
        else if(j < b || e < i)
            return 0;
        else
        {
            int f = query(i,j,2*node,b,(b+e)/2);
            int s = query(i,j,2*node+1,(b+e)/2+1,e);
            return max(s , f);
        }
    }
    void update(int i,int val,int node = 1, int b = 0,int e = -1)
    {
        if(e==-1)e = Edges.size() - 1;
        if(b==i && e==i)
            seg[node] = allEdges[Edges[b]] = val;
        else if(i < b || e < i)
            return;
        else
        {
            update(i,val,2*node,b,(b+e)/2);
            update(i,val,2*node+1,(b+e)/2+1,e);
            seg[node] = max(seg[2*node],seg[2*node+1]);
        }
    }
};
typedef Seg Segment;
vector<Segment> Paths;

void createS(int u,int d = 10006, int L = 0)
{
    int sz = 1, ma = 0 , maId = 0;
    List[u].lvl = L;
    for(int i = 0; i < adj[u].size(); ++i)
    {
        int v = adj[u][i];
        if(List[v].parent != -1)continue;
        List[v].parent = u;
        edgeV[edg[u][i]] = v;
        createS( v , edg[u][i] , L+1 );
        sz += List[v].size;
        if(List[v].size > ma)
            ma = List[v].size , maId = v;
    }
    if(2*ma >= sz)
    {
        List[u].stChild = maId;
    }
    if(List[u].stChild == -1)
    {
```

```
        List[u].path = Paths.size() , List[u].pathIdx = 0;
        Paths.push_back(Segment());
        Paths.back().Nodes.push_back(u);
        Paths.back().Edges.push_back(d);
    }
    else
    {
        int p = List[u].path = List[maId].path;
        List[u].pathIdx = Paths[p].Nodes.size();
        Paths[p].Nodes.push_back(u);
        Paths[p].Edges.push_back(d);
    }
    List[u].size = sz;
}

int maxEdge(int a,int b)
{
    if(b==a)return 0;
    int A,B;
    for(A=a, B=b ; (A!=ROOT || B!=ROOT) ;)
    {
        if(List[A].path == List[B].path)break;
        int na = Paths[List[A].path].Nodes.back();
        int nb = Paths[List[B].path].Nodes.back();
        if(List[na].lvl > List[nb].lvl || B==ROOT)
            A = Paths[List[A].path].Nodes.back();
        else
            B = Paths[List[B].path].Nodes.back();
    }
    int ma = 0;
    for(int i = a; i != A ; i = Paths[List[i].path].Nodes.back())
    {
        int lo = List[i].pathIdx;
        int p = List[i].path;
        int hi = Paths[p].Edges.size()-1;
        ma = max( ma ,  Paths[p].query(lo,hi) );
    }
    for(int i = b; i != B ; i = Paths[List[i].path].Nodes.back())
    {
        int lo = List[i].pathIdx;
        int p = List[i].path;
        int hi = Paths[p].Edges.size()-1;
        ma = max( ma ,  Paths[p].query(lo,hi) );
    }

    if(A!=B)
    {

        int lo = min(List[B].pathIdx,List[A].pathIdx);
        int hi = max(List[B].pathIdx,List[A].pathIdx);
        int p = List[A].path;
        ma = max( ma , Paths[p].query(lo,hi-1) );
    }
    return ma;

}

int main(int argc, char** argv) {
    int TC;
    scanf("%d",&TC);
    for(int tc = 1; tc<=TC ; ++tc)
    {
        int N;
        scanf("%d",&N);
        Paths.clear();
        adj = VVI(N);
        edg = VVI(N);
        for(int i = 0; i < N ; ++i)
            List[i] = Vertex();
        for(int i = 0; i < N-1 ; ++i)
        {
            int u ,v ,c;
            scanf("%d%d%d",&u,&v,&c);
            u--;v--;
            adj[u].push_back(v);
            adj[v].push_back(u);
            edg[u].push_back(i);
            edg[v].push_back(i);
            allEdges[i] = c;
        }

        List[ROOT].parent = -2;
        createS(ROOT);

        for(int i = 0; i < Paths.size() ; ++i)
        {
```

```
            if(Paths[i].Nodes.back()!=ROOT)
                Paths[i].Nodes.push_back(List[Paths[i].Nodes.back
()].parent);
            Paths[i].seg = VI(4*Paths[i].Edges.size()+10);
            if(Paths[i].Nodes[0]!=ROOT)Paths[i].createSegment();
        }

        string com;
        char line[30];
        while(scanf("%s",line))
        {
            com = line;
            if(com=="DONE")break;
            if(com=="CHANGE")
            {
                int idx, newcost;
                scanf("%d%d",&idx,&newcost);
                idx--;
                int p = List[edgeV[idx]].path;
                int pos= List[edgeV[idx]].pathIdx;
                Paths[p].update(pos,newcost);
            }
            else if(com=="QUERY")
            {
                int a,b;
                scanf("%d%d",&a,&b);
                int t = maxEdge(a-1,b-1);
                printf("%d\n",t);
            }
        }
    }
    return 0;
}
```

GEOMETRY

POINT STRUCTURE

```
template<class INT>
struct Poin
{
    INT x,y;
    Poin(INT a=0,INT b=0){
        x = a, y = b;
    }
    Poin operator+(const Poin &P)const{
        return Poin(x+P.x , y+P.y);
    }
    Poin operator-(const Poin &P)const{
        return Poin(x-P.x , y-P.y);
    }
    INT operator*(const Poin &P)const{
        return x*P.x + y*P.y;
    }
    INT operator^(const Poin &P)const{
        return x*P.y - y*P.x;
    }
    INT mag()const{
        return sqrt(x*x + y*y);
    }
    Poin scale(INT H,bool div=false)const{
        return div ? Poin(x/H , y/H) : Poin(x*H , y*H);
    }
    Poin unit()const{
        return scale(mag(),1);
    }
    bool operator<(const Poin &P)const{
        return x!=P.x ? x<P.x : y<P.y;
    }
    bool operator==(const Poin &P)const{
        return x==P.x && y==P.y;
    }
};
typedef Poin<double> Point;
```

CIRCLE WITH 3 POINTS

```
Point Circle3Points(Point A,Point B,Point C)
{
    //Find the Center of a Circle that has points A,B,C
    //The Circle3Points needs to be different else use Circle2Points
    Point M1 = (A+B).scale(2,true);
    double a1 = (A-B).y , b1 = (A-B).x , c1 = a1*M1.y + b1*M1.x;
    Point M2 = (C+B).scale(2,true);
    double a2 = (C-B).y , b2 = (C-B).x , c2 = a2*M2.y + b2*M2.x;
```

```
    double det = a1*b2 - b1*a2;
    double detY= c1*b2 - b1*c2;// <- Easy Bug here
    double detX= a1*c2 - c1*a2;
    return Point(detX/det , detY/det);
}
```

CIRCLE WITH 2 POINTS AND RADIUS

```
pair<Point,Point> Circle2Points(Point A,Point B,double R)
{
    //Find the Centers of the circles which touch A,B and have radius R
    Point M = (A+B).scale(2,true); // scale 2 for divide
    Point MA = A-M;
    if(R*R < MA*MA || MA*MA==0) // does not exist
        return pair<Point,Point>(Point(1e9,1e9),Point(1e9,1e9));
    double mag = sqrt(R*R - MA*MA);
    Point M1 = MA; swap(M1.x,M1.y); M1.x *= -1;
    M1 = M1.unit();
    M1 = M1.scale(mag);
    Point M2 = MA; swap(M2.x,M2.y); M2.y *= -1;
    M2 = M2.unit();
    M2 = M2.scale(mag);
    return pair<Point,Point>(M+M1,M+M2);
}
```

AREA OF A POLYGON

```
double AreaOfPolygon(vector<Point> &V) // Area of a Polygon O(N)
{
    //negative area means clockwise positive means anticlockwise
    double Area = 0;
    int N = V.size();
    for(int i = 1; i < N-1; ++i)
        Area += (V[i]-V[0])^(V[i+1]-V[0]);
    return Area/2.0;
}
```

COLLINEAR , BETWEEN & INSIDE OF TRIANGLE

```
inline bool Collinear(Point A,Point B,Point C)// O(1)
{
    return ((A-B)^(C-B)) == 0;
}

inline bool between(Point A,Point B,Point C) // A-B-C   O(1)
{
    // B lines in the segment AC
    if( A==B || C==B )return true;
    return (A-C)*(B-C) > 0 && (C-A)*(B-A) > 0 && Collinear(A,B,C);
}

bool InsideTriangle(Point A,Point B,Point C,Point P)
{
    // Is Point P insde triangle ABC
    bool bt = between(A,P,B) || between(B,P,C) || between(C,P,A);
    if( ((A-B)^(C-B)) == 0 )return bt;
    bool cp = ( (((A-B)^(P-B))>0) == (((B-C)^(P-C))>0) &&
                (((B-C)^(P-C))>0) == (((C-A)^(P-A))>0) );
    return bt || cp;
}

double LinePointD(Point A,Point B,Point C,bool seg = false)
{
    //Line is AB  the point is C
    if(seg && (A-B)*(C-B) < 0)return (B-C).mag();
    if(seg && (B-A)*(C-A) < 0)return (A-C).mag();
    return fabs((A-B)^(C-B))/(A-B).mag();
}
```

ROTATE A VECTOR

```
Point Rotate(Point A,double delta) // Rotating a Vector O(1)
{
       return Point( A.x*cos(delta)-A.y*sin(delta) ,
                     A.x*sin(delta)+A.y*cos(delta) );
}
```

LINE INTERSECTION

```
pair<Point,int> lineIntersection(Point A1, Point B1, Point A2,Point B2)
{
    // Intersection between two segments or two lines
    // ay + bx = c --> (x2-x1)y - (y2-y1)x = (x2-x1)y0-(y2-y1)x0
    double a1 = A1.x-B1.x , b1 = -(A1.y-B1.y), c1 = a1*A1.y + b1*A1.x;
    double a2 = A2.x-B2.x , b2 = -(A2.y-B2.y), c2 = a2*A2.y + b2*A2.x;
    double det = a1*b2 - b1*a2;
    double detY= c1*b2 - b1*c2;
    double detX= a1*c2 - c1*a2;
```

```
    if(det==0)return make_pair(A1,1);// Parallel Lines
    Point C(detX/det , detY/det);
    return pair<Point,int>(C,0); // Normal Intersection
}
```

SEGMENT INTERSECTION

```
pair<Point,int> segmentIntersection(Point A1, Point B1, Point A2,Point
B2,bool seg = false)
{
  // Intersection between two segments or two lines
  // ay + bx = c --> (x2-x1)y - (y2-y1)x = (x2-x1)y0-(y2-y1)x0
  double a1 = A1.x-B1.x , b1 = -(A1.y-B1.y), c1 = a1*A1.y + b1*A1.x;
  double a2 = A2.x-B2.x , b2 = -(A2.y-B2.y), c2 = a2*A2.y + b2*A2.x;
  double det = a1*b2 - b1*a2;
  double detY= c1*b2 - b1*c2;
  double detX= a1*c2 - c1*a2;

  if(det==0)
       if(between(A1,A2,B1) || between(A1,B2,B1) || between(A2,A1,B2))
            return make_pair(A1,3); // Segments or lines are conicident
       else if(!seg && Collinear(A1,B2,A2))
            return make_pair(A1,3); // Lines are coincident
       else
            return make_pair(A1,1); // Lines are parallells
  Point C(detX , detY);
  if(seg && (!between(A1.scale(det),C,B1.scale(det)) ||
          !between(A2.scale(det),C,B2.scale(det))))
       return make_pair(C,2); // Lines intersect out of the
  return pair<Point,int>( C.scale(det,true) ,0); // Normal Intersection
}
```

SEGMENT INTERSECT (BOOLEAN)

```
bool segmentIntersect(Point A, Point B, Point C, Point D)
{

    int d1 = (A-C)^(D-C);
    int d2 = (B-C)^(D-C);
    int d3 = (C-A)^(B-A);
    int d4 = (D-A)^(B-A);
    // if( d1 and d2 has different signs && d3 and d4 too)
    if( (d1^d2) < 0 && (d3^d4) < 0 ) return true;
    if( between(C,A,D) || between(C,B,D)
     || between(A,C,B) || between(A,D,B) )
        return true;
    return false;
}
```

CONVEX HULL

```
struct cmpCP
{
    Point P;
    cmpCP(Point &_P){P = _P;}
    bool operator()(const Point &A, const Point &B)const{
        double cp = (A-P)^(B-P);
        if(cp != 0)return cp > 0;
        return (A-P)*(A-P) < (B-P)*(B-P);
    }
};
vector<Point> ConvexHull(vector<Point> &V) // Convex Polygon O(NlogN)
{
    if(V.size()<=2)return V;
    sort(V.begin(),V.end()); //sort them by X then Y
    sort(V.begin()+1,V.end(),cmpCP(V[0]));
    vector<Point> R(V.begin(),V.begin()+2);
    int top = 1;
    for(int i = 2; i < (int)V.size(); ++i)
    {
        while(top >= 1 && ((V[i]-R[top])^(R[top-1]-R[top])) <= 0 )
        {
            R.pop_back(); top--;
        }
        R.push_back(V[i]); top++;
    }
    return R;
}
```

CLOSEST PAIR OF POINTS

```
double ClosestPoint(vector<Point> &V) // LineSweep O(NlogN)
{
    sort(V.begin(),V.end());
    set<Point,cmpYX> S;
    double D = 1e9; int tail = 0;
    for(int i = 0; i < (int)V.size(); ++i)
    {
        while(tail<i && (V[i].x-V[tail].x) > D)tail++;

        set<Point,cmpYX>::iterator it1,it2;
        it1 = S.lower_bound(Point(V[i].x,V[i].y-D));
        it2 = S.upper_bound(Point(V[i].x,V[i].y+D));
```

```
        for(set<Point,cmpYX>::iterator it = it1 ; it != it2 ; ++it)
            D = min( D , ((*it)-V[i]).mag() );
        S.insert(V[i]);
    }
    return D;
}
```

DIAMETER OF A POLYGON

```
double DiameterOfPolygon(vector<Point> &V) // Rotating Caliper O(N)
{
    // V --> has to be a ConvexPolygon, use Convex Hull
    int idx = 0 , N = V.size();
    double dist = 1e9;
    for(int i = 0; i < N; ++i)
    {
        Point A = V[i] , O = V[idx], NE = V[(i+1)%N];
        while(LinePointD(A,NE,V[(idx+1)%N]) > LinePointD(A,NE,O))
            O = V[idx = (idx+1)%N];
        dist = min( dist , LinePointD(V[i],V[(i+1)%N],V[idx]));
    }
    return dist;
}
```

FARTHEST PAIR OF POINTS

```
double FarthestPoints(vector<Point > &V) //Rotating Caliper O(N)
{
        // V --> has to be a ConvexPolygon, use Convex Hull
        int j = 0 , N = V.size();
        double dist = 0;
        for(int i = 0; i < N; ++i)
        {
                while( ((V[i]-V[(j+1)%N])*(V[i]-V[(j+1)%N])) >
                         ((V[i]-V[j])*(V[i]-V[j])) )
                        j = (j+1)%N;
                dist = max( dist , (V[i]-V[j])*(V[i]-V[j]) );
        }
        return sqrt(dist);
}
```

AREA OF RECTANGLES

```
struct Event
{
        int y, x1,x2 , id;
        Event(int a=0,int b=0,int c=0,int ac = 0){
                y = a, x1 = b , x2 = c , id = ac;
        }
        bool operator<(const Event &E)const{
                return (y != E.y) ? (y<E.y) : ((x1 != E.x1) ? (x1<E.x1) :
                                ((x2!=E.x2) ? (x2<E.x2) : (id < E.id)));
        }
};
int AreaOfRectangles(vector<pair<Point,Point> > &V)
{
        vector<Event> EV;
        for(int i = 0; i < (int)V.size(); ++i)
        {
                int x1 = min(V[i].first.x,V[i].second.x);
                int x2 = max(V[i].first.x,V[i].second.x);
                int y1 = min(V[i].first.y,V[i].second.y);
                int y2 = max(V[i].first.y,V[i].second.y);
                EV.push_back(Event(y1,x1,x2,i));
                EV.push_back(Event(y2,x1,x2,i));
        }
        sort(EV.begin(),EV.end());
        set< Event > S;
        int p = 0;
        int Area = 0;
        for(int i = 0; i < (int)EV.size(); ++i)
        {
                Event A = EV[i];
                int np = A.y;
                A.y = 0;
                int length = 0;
                int pr = -1,ma = -1;
                for(set< Event >::iterator it=S.begin();it!=S.end();++it)
                {
                        int x1 = it->x1 , x2 = it->x2;
                        if(ma < x1)
                        {
                                length += ma - pr;
                                pr = x1;
                                ma = x2;
                        }else
                                ma = max( ma , x2 );
                }
                length += ma-pr;
                Area += (np-p)*(length);
                if(S.count(A)==0)S.insert(A);
```

```
                else S.erase(A);
                p = np;
        }
        return Area;
}
```

INSIDE A CONVEX POLYGON

```
bool insideConvex(vector<Point> V, Point P,bool edge = false)
{
        int poscnt = 0 , N = V.size();
        for(int i = 0; i < V.size(); ++i)
        {
                poscnt += ((V[(i+1)%N]-V[i])^(P-V[i])) > 0;
                if(edge && between(V[i],P,V[(i+1)%N]))return true;
        }
        return poscnt == V.size() || poscnt == 0;
}
```

INSIDE A CONCAVE POLYGON

```
bool insideConcave(vector<Point> V, Point P)
{
        int N = V.size();
        double sum = 0;
        for(int i = 0; i < N; ++i)
        {
                int j = (i+1)%N;
                double m1 = (V[i]-P).mag() , m2 = (V[j]-P).mag();
                double dp = ((V[i]-P)*(V[j]-P))/(m1*m2);
                double cp = (V[i]-P)^(V[j]-P);
                if(cp==0 && between(V[i],P,V[j]))return true;
                if(dp>1)dp = 1;  if(dp<-1)dp = -1;
                double angle = cp > 0 ? acos(dp) : -acos(dp);
                sum += angle;
        }
        return fabs(sum) > M_PI;
}
```

CENTROID OF A POLYGON

```
Point Centroid(vector<Point> V)
{
        Point R(0,0); int N = V.size();
        for(int i = 0 ; i < N; ++i)
        {
                int j = (i+1)%N;
                R.x += (V[i].x+V[j].x)*(V[i]^V[j]);
                R.y += (V[i].y+V[j].y)*(V[i]^V[j]);
        }
        R.scale(6*AreaOfPolygon(V),true);
        return R;
}
```

THIRD POINT GIVEN 2

```
Point ThirdPoint(Point A,Point B,double ac,double bc)
{
        double ab = (A-B).mag();
        double s = (ac + bc + ab)/2;
        double AREA = sqrt(s*(s-ac))*sqrt((s-bc)*(s-ab));
        double h = 2*(AREA / ab);
        double proyA = sqrt(ac*ac - h*h);
        double proyB = sqrt(bc*bc - h*h);
        Point D = A-B , C;
        if(proyA > proyB)
        {
                D = D.scale(-1);
                D = (D.unit()).scale( proyA );
                Point D2 = (A-B).unit();
                swap(D2.x,D2.y); D2.x *= -1;
                D2 = D2.scale(h);
                C = A+D+D2;
        }else{
                D = (D.unit()).scale( proyB );
                Point D2 = (B-A).unit();
                swap(D2.x,D2.y); D2.x *= -1;
                D2 = D2.scale(h);
                C = B+D+D2;
        }
        return C;
}
```

GRAPH THEORY

STRONGLY CONNECTED COMPONENTS

```
VVI adj;
VVI radj;
bool vis[100000];
void scc_dfs(int u,stack<int> &S,bool rev = false)
{
    vis[u] = true;
    VI V = rev ? radj[u] : adj[u];
    for(int i = 0; i < (int)V.size(); ++i)
        if(!vis[V[i]])
            scc_dfs(V[i],S,rev);
    if(!rev)S.push(u);
}

VVI StronglyConnected()
{
    stack<int> STK;
    memset(vis,0,sizeof(vis));
    for(int i = 0; i < (int)adj.size(); ++i)
        if(!vis[i])
            scc_dfs(i,STK);
    memset(vis,0,sizeof(vis));
    VVI SCC;
    while(!STK.empty())
    {
        int u = STK.top();STK.pop();
        if(vis[u]){
            SCC.back().push_back(u);
        }else{
            scc_dfs(u,STK,true);
            SCC.push_back(VI(1,u));
        }
    }
    return SCC;
}
```

TWO COLORING OF A GRAPH

```
int color[100000];
bool TwoColoring(int u,int c)
{
        color[u] = c;
        for(int i = 0; i < adj[u].size(); ++i)
        {
                int v = adj[u][i];
                if(color[v] == c)return false;
                if(color[v] !=-1)continue;
                TwoColoring(v,1-c);
        }
        return true;
}
void ColorgGraph(int N)
{
        memset(color,-1,sizeof(color));
        for(int i = 0; i < N; ++i)
                if(color[i]==-1)
                        TwoColoring(i,0);
}
```

MAXIMUM BIPARTITE MATCHING

```
int par[1000];
bool FindMatch(int u)
{
        for(int i = 0; i < adj[u].size(); ++i)
        {
                int v = adj[u][i];
                if(vis[v])continue; vis[v] = true;
                if(par[v]==-1 || FindMatch(par[v]))
                {
                        par[v] = u;
                        return true;
                }
        }
        return false;
}
int MaximumMatching(int L)
{
        int ret = 0;
        memset(par,-1,sizeof(par));
        for(int i = 0; i < L ; ++i)
        {
                memset(vis,false,sizeof(vis));
                if(FindMatch(i)) ret++;
        }
        return ret;
}
```

MINIMUM VERTEX COVER

```
int VetexCoverBP(int N)
{
	int cover = MaximumMatching(N);
	for(int u = 0; u < N; ++u)
		for(int j = 0; j < adj[u].size(); ++j)
			if(par[adj[u][j]]==-1)
				cover++;
	return cover;
}
```

HOPCROFT-KARP BIPARTITE MATCHING

```
typedef vector<int> VI;
typedef vector<VI> VVI;
VVI adj , radj;
int parL[60000] , L , parR[60000] , R;
int prev[60000];
bool bfs()
{
	queue<int> Q;
	for(int u = 0; u < L ; ++u)
		if( parL[u] == -1 )
			Q.push(u);
	memset(prev,-1,sizeof(prev));
	bool found = false;
	while(!Q.empty())
	{
		int u = Q.front();Q.pop();
		for(int i = 0; i < (int)adj[u].size(); ++i)
		{
			int v = adj[u][i];
			if(parL[u] == v || prev[v]!=-1)continue;
			if(parR[v] == -1)found = true;
			prev[v] = u;
			if(!found)Q.push( parR[v] );
		}
	}
	return found;
}

bool dfs(int v)
{
	int u = prev[v];
	if(u==-1)return false;
	if( parL[u] == -1 || dfs(parL[u]) )
	{
		parL[u] = v , parR[v] = u , prev[v] = -1;
		return true;
	}
	return false;
}

int HopcroftKarp()
{
	int ret = 0;
	memset(parL,-1,sizeof(parL));
	memset(parR,-1,sizeof(parR));
	while(bfs()) for(int v = 0; v < R ; ++v)
		if(parR[v]==-1 && prev[v]!=-1)
		for(int i = 0; i < radj[v].size() && parR[v]==-1 ; ++i)
		{
			int u = radj[v][i];
			if(parL[u]==-1 || dfs( parL[u] ))
				parL[u] = v , parR[v] = u, ++ret,prev[v] = -1;
		}
	return ret;
}
```

DINITZ EDGE MATRIX

```
int cap[100][100];
int flow[100][100];
int dinitz(int s,int t)
{
    int f = 0;
    while(true)
    {
        int prev[100];
        queue<int> Q;
        memset(prev,-1,sizeof(prev));
        prev[s] = -2;
        Q.push(s);
        while(!Q.empty() && !prev[t])
        {
            int u = Q.front();Q.pop();
            for(int i = 0; i < (int)adj[u].size(); ++i)
            {
```

```
                int v = adj[u][i];
                if(prev[v]!=-1 || cap[u][v]-flow[u][v] == 0)continue;
                Q.push(v);
                prev[v] = u;
            }
        }
        if(prev[t] == -1)break;
        for(int i = 0; i < (int)adj.size(); ++i)
        {
            int z = adj[t][i];
            if(prev[z]==-1 || cap[z][t]-flow[z][t] == 0)continue;
            int mincap = 0;
            for(int u = z,v = t; v != s; v=u ,u = prev[v])
                mincap = min( mincap , cap[u][v]-flow[u][v] );
            for(int u = z,v = t; v != s; v=u ,u = prev[v])
                flow[u][v] = -(flow[v][u] -= mincap);
            f += mincap;
        }
    }
    return f;
}
```

DINITZ EDGE LIST

```
struct Edge
{
    int u,v,cap,flow,rev;
    Edge(int U = 0,int V = 0,int C = 0,int R = -1){
        u = U , v = V , cap = C , rev = R , flow = 0;
    }
};
vector<Edge> Edges;

void addEdge(int u,int v , int cap = 1)
{
    int pDir = Edges.size() , pRev = pDir + 1;
    adj[ u ].push_back(pDir);
    adj[ v ].push_back(pRev);
    Edges.push_back(Edge( u , v , cap , pRev));
    Edges.push_back(Edge( v , u , 0 , pDir));
}

int dinitzADJ(int s,int t)
{
    int flow = 0;
    while(true)
    {
        int prev[30000];
        memset(prev,-1,sizeof(prev));
        prev[s] = -2;
        queue<int> Q;
        Q.push(s);
        while(!Q.empty() && prev[t]==-1)
        {
            int u = Q.front();Q.pop();
            for(int i = 0; i < int(adj[u].size()); ++i)
            {
                int e = adj[u][i], v = Edges[e].v;
                int cap = Edges[e].cap , flo = Edges[e].flow;
                if(prev[v]!=-1)continue;
                if(cap - flo <= 0)continue;
                Q.push(v);
                prev[v] = e;
            }
        }
        if(prev[t] == -1)break;
        for(int i = 0; i < int(adj[t].size()); ++i)
        {
            int e = adj[t][i];
            if(prev[Edges[e].v]==-1 ||
               Edges[Edges[e].rev].cap == Edges[Edges[e].rev].flow)
                continue;
            int mincap = int(1e7);
            for(int v = t ; v != s ; v = Edges[prev[v]].u)
            {
                int tem = Edges[prev[v]].cap - Edges[prev[v]].flow;
                mincap = min( mincap , tem );
            }
            flow += mincap;
            for(int v = t ; v != s ; v = Edges[prev[v]].u)
            {
                Edges[prev[v]].flow += mincap;
                Edges[Edges[prev[v]].rev].flow = -Edges[prev[v]].flow;
            }
        }
    }
    return flow;
}
```

BELLMAN-FORD

```
int D[100000];
struct EdgeB
{
    int u,v,d;
};
EdgeB EdgesB[100000];
bool BellManFord(int st,int N,int E)
{
    // return true if there is no negative cycles
    // use this to preprocess the potentials of MCMF
    for(int i = 0; i < N; ++i)
        D[i] = INF;
    D[st] = 0;
    bool changed = true;;
    for(int kth = 0 ; kth < N && changed ; ++kth)
    {
        changed = false;
        for(int j = 0; j < E; ++j)
        {
            int u = EdgesB[j].u, v = EdgesB[j].v, d = EdgesB[j].d;
            if(D[v] > D[u]+d)
            {
                D[v] = D[u]+d;
                changed = true;
                if(kth == N)return false;
            }
        }
    }
    return true;
}
```

STOER-WAGNER MINCUT

```
#define INF 98765432
// edge-weighted graph in O( VE + V^2lg(V) )
class MinCut {
public:
   VVI W; //capacity adjacency matrix
   MinCut(int _N) : W(_N, VI(_N)) {}
   MinCut(const VVI& _W = VVI()) : W(_W) {}
   int process();
};
struct Node {
   int vertex , weight;
   Node(int _vertex, int _weight) : vertex(_vertex), weight(_weight) {}
   bool operator<(const Node& n) const {
      if (weight != n.weight) return weight > n.weight;
      return vertex < n.vertex;
   }
};
int MinCut::process() {
   int N = W.size();
   vector<int> V(N);
   for (int i = 0; i < N; ++i)
      V[i] = i;
   int res = INF;
   for (int n = N; n > 0; --n) {
      vector<bool> A(n);
      A[0] = true;
      vector<int> D(n);
      set<Node> pq;
      for (int i = 1; i < n; ++i)
         pq.insert(Node(i, D[i] = W[ V[0] ][ V[i] ]));
      int prev = V[0];
      for (int i = 1; i < n; ++i) {
         Node cur = *pq.begin();
         pq.erase(pq.begin());
         A[cur.vertex] = true;
         if (i == n-1) {
            res = min(res, cur.weight);
            for (int j = 0; j < n; ++j)
               W[ V[j] ][prev] =
                    (W[prev][ V[j] ] += W[ V[cur.vertex] ][ V[j] ]);
            V[cur.vertex] = V[n-1];
            break;
         }
         prev = V[cur.vertex];
         for (int j = 1; j < n; ++j) {
            if (A[j]) continue;
            set<Node>::iterator it = pq.find(Node(j, D[j]));
            pq.erase(it);
            pq.insert(Node(j, D[j] += W[ V[cur.vertex] ][ V[j] ]));
         }
      }
   }
   return res;
}
```

MAXIMUM FLOW MINIMUM COST

```
#define SOURCE 0
#define INF 100000000
#define EPS 1e-9

int cost[100][100];
int cap[100][100];
int prev[100];
int pot[100];
int flow[100][100];
bool ditra(int t,int N)
{
    typedef pair<int,int> PII;
    priority_queue<PII, vector<PII> , greater<PII> > PQ;
    PQ.push(PII(0,SOURCE));
    vector<int> dist( N , INF );
    dist[SOURCE] = 0;
    while(!PQ.empty())
    {
        PII curr = PQ.top();PQ.pop();
        int u = curr.second;
        int d = curr.first;
        if(d > dist[u]+EPS)continue;
        for(int v = 0; v < N ; ++v)
        {
            if(cap[u][v] > flow[u][v] &&
               dist[v] > (pot[u] - pot[v] + d + cost[u][v]))
            {
                PQ.push(PII(dist[v] = (pot[u]-pot[v]+d+cost[u][v]) ,
v));
                prev[v] = u;
            }
            if(flow[v][u] > 0 &&
               dist[v] > (pot[u] - pot[v] + d - cost[v][u]))
            {
                PQ.push(PII(dist[v] = (pot[u]-pot[v]+d-cost[v][u]) ,
v));
                prev[v] = u;
            }
        }
    }
    for(int i = 0; i < N; ++i)pot[i] = min( pot[i]+dist[i] , INF);
    return dist[t] < INF-EPS;
}

int mcmf(int t,int N)
{
    memset(flow,0,sizeof(flow));
    for(int i = 0; i < N; ++i)
        pot[i] = 0;
    int totcost = 0 , totflow = 0;
    while( ditra(t,N) )
    {
        int mincap = INT_MAX;
        for(int u = prev[t] , v = t; v!=SOURCE ; v=u , u = prev[u])
            mincap = min( mincap , (cap[u][v]>flow[u][v] ? cap[u][v]-
flow[u][v] : flow[v][u]) );
        int mincost = 0;
        for(int u = prev[t] , v = t; v!=SOURCE ; v=u , u = prev[u])
        {
            if( cap[u][v] > flow[u][v] )
            {
                flow[u][v] += mincap;
                mincost += mincap*cost[u][v];
            }else
            {
                flow[v][u] -= mincap;
                mincost -= mincap*cost[v][u];
            }
        }
        totflow += mincap;
        totcost += mincost;
    }
    return totcost;
}
```

ARTICULATION POINT

```
#define ROOT 0
bool ArtiPoint[10000];
int low[10000], num[10000] , idx;
void dfs_ArtiPoints(int u)
{
	num[u] = low[u] = idx++;
	int c = 0;
	bool ap = false;
	for(int i = 0; i < adj[u].size(); ++i)
	{
		int v = adj[u][i];
		if(num[v]==-1)
		{
			dfs_ArtiPoints(v);
			c++;
			if(low[v] >= num[u] && num[u] != ROOT)
				ap = true;
		}
		low[u] = min(low[u] , num[v]);
	}
	if(num[u]==ROOT && c>1)
		ap = true;
	ArtiPoint[u] = ap;
}

void buildArtiPoints(int N)
{
	idx = 0;
	memset(num,-1,sizeof(num));
	for(int i = 0; i < N ; ++i)
		if(num[i]==-1)
			dfs_ArtiPoints(i);
}
```

BRIDGES

```
typedef pair<int,int> PII;
vector<PII> Bridges;
int low[10000], num[10000] , idx;
void dfs_Bridges(int u,int p = -1)
{
	num[u] = low[u] = idx++;
	for(int i = 0; i < adj[u].size(); ++i)
	{
		int v = adj[u][i];
		if(v == p)continue;
		if(num[v]==-1)
		{
			dfs_Bridges(v , u);
			if(low[v] > num[u])
				Bridges.push_back(PII(u,v));
			low[u] = min(low[u] , low[v]);
		}
		else
			low[u] = min(low[u] , num[v]);
	}
}
void buildBridges(int N)
{
	idx = 0;
	Bridges = vector<PII>();
	memset(num,-1,sizeof(num));
	for(int i = 0; i < N ; ++i)
		if(num[i]==-1)
			dfs_Bridges(i);
}
```

BICONNECTED COMPONENTS

```
int low[10000], num[10000] , idx;
vector<vector<PII> > BCC;
stack<PII> S;
void dfs_BCC(int u,int p,int root)
{
	num[u] = low[u] = idx++;
	for(int i = 0; i < adj[u].size(); ++i)
	{
		int v = adj[u][i];
		if(v == p)continue;
		if(num[v]==-1)
		{
			S.push(PII(u,v));
			dfs_BCC(v, u,root);
			if(u==root || low[v] >= num[u]){
				BCC.push_back(vector<PII>());
				while(!S.empty())
				{
					PII n = S.top();S.pop();
					BCC.back().push_back(n);
```

```
                                if(n.first == u && n.second==v)
break;
                            }
                        }
                        low[u] = min(low[u] , low[v]);
                }
                else if(num[v]-1 < num[u])
                {
                        low[u] = min(low[u] , num[v]);
                        S.push(PII(u,v));
                }
        }
}
void BiconnectedComponents(int N)
{
        idx = 0;
        BCC.clear();
        memset(num,-1,sizeof(num));
        for(int i = 0; i < N; ++i)
                if(num[i]==-1)
                        dfs_BCC(i,-1,i);
}
```

KUHN-MUNKRE'S ASSIGNMENT ALGORITHM

```
using namespace std;

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

typedef vector<bool> VB;
typedef vector<VB> VVB;
class AssignmentProblem {
   const static int INF = 98765432;
   const static int PARENT_OF_ROOT = -2;
   const static int UNASSIGNED     = -1;
public:
   // INPUT:
   VVI W;          // W: profit matrix
                   //    set W[i][j] = -W[i][j] for min weighted matching
                   // each row => vertex in L; each col => vertex in R
   // OUTPUT:      // after algorithm => best matching (asgnLR & asgnRL)
   int N;          // max(nRows, nCols)
   VI asgnLR, asgnRL;
                   // asgnLR[i] assignment in R of vertex i of L
                   // asgnRL[j] assignment in L of vertex j of R
   AssignmentProblem(const VVI& _w = VVI()) : W(_w) {}
   AssignmentProblem(int nRows, int nCols) : W(VVI(nRows, VI(nCols))) {}
   int process();
};
int AssignmentProblem::process()
{
   int nRows = int(W.size());
   if (nRows == 0) return 0;
   int nCols = int(W[0].size());
   if (nCols == 0) return 0;
   N = max(nRows, nCols);
   if (nRows != nCols)
   {
      if (nRows < nCols)
         W.insert(W.end(), nCols-nRows, VI(nCols, 0));
      else
         for (int i = 0; i < nRows; ++i)
            W[i].insert(W[i].end(), nRows-nCols, 0);
   }
   asgnLR = VI(N, UNASSIGNED);
   asgnRL = VI(N, UNASSIGNED);
   VB S(N), T(N);
   VI par(N, UNASSIGNED);
   // initial labelling
   VI lblL(N, 0);
   VI lblR(N, 0);
   for (int i = 0; i < N; ++i)
      for (int j = 0; j < N; ++j)
         lblL[i] = max(lblL[i], W[i][j]);

   VI mingap(N), mingap_src(N);
   for (int rem_match = N; rem_match > 0; --rem_match) {
      // rem_match = number of vertices to match
      queue<int> Q;
      S.assign(N, false);  // S = {}
      T.assign(N, false);  // T = {}
      par.assign(N, UNASSIGNED);
      // Find root of the tree as first non-matched element in L
      int root = UNASSIGNED;
      for (int i = 0; i < N; ++i)
         if (asgnLR[i] == UNASSIGNED)
         {     root = i;      break;  }
```

```
S[root] = true;
Q.push(root);
par[root] = PARENT_OF_ROOT;
for (int j = 0; j < N; ++j)
{
   mingap[j] = lblL[root] + lblR[j] - W[root][j];
   mingap_src[j] = root;
}
int fi = -1, fj = -1;
bool augmenting_path_found;
while (true)
{
   augmenting_path_found = false;
   while (!Q.empty())
   {
      int cur = Q.front();
      Q.pop();
      for (int j = 0; j < N; ++j)
      {
         if (!T[j] && lblL[cur] + lblR[j] == W[cur][j])
         {
            int mi = asgnRL[j];
            if (mi == UNASSIGNED)
            {
               fi = cur, fj = j;
               augmenting_path_found = true;
               break;
            }
            T[j] = true;           // add j to set T
            if (!S[mi])
            {
               Q.push(mi);
               S[mi] = true;         // add mi to S
               par[mi] = cur;        // augmenting path
               for(int j2 = 0; j2 < N; j2++)
               {
                  if(lblL[mi] + lblR[j2]-W[mi][j2]<mingap[j2])
                  {
                     mingap[j2] =lblL[mi]+lblR[j2]-W[mi][j2];
                     mingap_src[j2] = mi;
                  }
               }
            }
         }
      }
   }

   if (augmenting_path_found)
      break;
   // No augmenting path found => improve labelling
   int delta = INF;
   for (int j = 0; j < N; ++j)
      if (!T[j])
         delta = min(delta, mingap[j]);

   for (int i = 0; i < N; ++i)
      if (S[i])
         lblL[i] -= delta;
   for (int j = 0; j < N; ++j) {
      if (T[j])
         lblR[j] += delta;
      else {
         mingap[j] -= delta;
         if (mingap[j] == 0)
            Q.push(mingap_src[j]);
      }
   }
}
// Inverse edges along augmenting path
if (augmenting_path_found) {
   for (; fi != PARENT_OF_ROOT; fi = par[fi]) {
      int tj = asgnLR[fi];
      asgnLR[fi] = fj;
      asgnRL[fj] = fi;
      fj = tj;
   }
}
}
// return weight of the optimal (maximum) matching
int ret = 0;
for (int i = 0; i < N; ++i)
   ret += W[i][asgnLR[i]];
return ret;
}
```

STABLE MARRIAGE PROBLEM

```
class StableMarriage {
public:
   // INPUT:
        VVI Mlist , Wpref;
   // Mlist[i] is list of women in order of decreasing
   // preference of each man
   // Wpref[i][j] is attractiveness of man j to woman i
   // the higher the value, the more attractive man j is to woman i
        VI M2W , W2M; // OUTPUT: engagements
   StableMarriage(int num_women, int num_men)
      : Mlist(num_men, VI(num_women)), Wpref(num_women, VI(num_men))  {}
   void run();
};
void StableMarriage::run() {
   int NM = int(Mlist.size());
   int NW = int(Wpref.size());
   W2M = VI(NW, -1);
   M2W = VI(NM, -1);
   VI idx(NM, 0);
   queue<int> Q;
   for (int i = 0; i < NM; ++i)
      Q.push(i);
   while (!Q.empty()) {
      int& m = Q.front(); // erm si hace referencia
      int w = Mlist[m][idx[m]++];
      // m proposes to w
      if (W2M[w] >= 0) { // w is already engaged
         if (Wpref[w][m] > Wpref[w][ W2M[w] ]) {
            int other = W2M[w];   // the other guy
            M2W[m] = w;           // engage w to m
            W2M[w] = m;           // engage m to w
            M2W[ other ] = -1;    // free the other guy
            m = other;            // and replace front of queue
         }
      }
      else {  // engage w to m and viceversa
         M2W[m] = w;
         W2M[w] = m;
         Q.pop();
      }
   }
}
```

SPLAY TREE

```
struct Node
{
   int val, freq;    // value to store in splay tree and its frequency
count
   int treeSz;       // size of subtree from this node
   Node *left, *right, *par;  // pointers to left, right, parent nodes
   Node(int v = 0) : val(v), freq(1), treeSz(1), left(0), right(0), par
(0) { }
// ~Node() { delete left;  delete right; }
   void _upd_subtree_size()
   {
      treeSz = freq + (left ? left->treeSz : 0) + (right ? right-
>treeSz : 0);
   }
   void _rotate()
   {
      Node *p = par;
      if (!p) return;      // no rotation since parent does not exist
      Node *g = p->par;  // grand-parent
      bool rotate_right = p->left == this;
      Node *T = ( rotate_right ? (p->left = right) : (p->right =
left) );
      ( rotate_right ? right : left ) = p;
      if (T) T->par = p;
      p->par = this;
      par = g;
      if (g) (g->left == p ? g->left : g->right) = this;
      p->_upd_subtree_size();
      _upd_subtree_size();
   }
   Node *splay(int keyval) {
      Node *x = this;
      while ( keyval != x->val ) // traverse down the tree comparing
with keyval
      {
         Node *nxt = ( keyval < x->val ? x->left : x->right );
         if (!nxt) break;
         x = nxt;
      }
      for (Node *p = x->par; p; p = x->par)
      {
         Node *g = p->par;
```

```
         if (!g)   // parent is root, perform zig step
         {
            x->_rotate();
            break;
         }
         if ((p->left == x) == (g->left == p)) // 2 lefts or rights, do
zig-zig
         {
            p->_rotate();
            x->_rotate();
         }
         else  // alternate left and right, do zig-zag
         {
            x->_rotate();
            x->_rotate();  // this is correct (do another rotation on x)
         }
      }
      return x;
   }
   Node *insert(int newval)
   {
      Node *x = this->splay(newval);
      if (x->val == newval) {
         ++x->freq;
         x->_upd_subtree_size();
         return x;
      }
      Node *n = new Node(newval);
      if (newval < x->val) {
         n->right = x;
         n->left  = x->left;
         if (x->left)
            x->left->par = n;
         x->left = NULL;
      }
      else {
         n->left  = x;
         n->right = x->right;
         if (x->right)
            x->right->par = n;
         x->right = NULL;
      }
      x->par = n;
      x->_upd_subtree_size();
      n->_upd_subtree_size();
      return n;
   }
   Node *erase(int oldval, bool all = false)
   {
      Node *x = this->splay(oldval);
      if (oldval != x->val) return x;
      x->freq = all ? 0 : x->freq-1;
      if (x->freq)
      {
         x->_upd_subtree_size();
         return x;
      }
      Node *l = x->left, *r = x->right;
      x->left = x->right = NULL;
      if (l)  // move l as root
      {
         l->par = NULL;
         l = l->splay(oldval);
         if (r) r->par = l;
         l->right = r;
         delete(x);
         l->_upd_subtree_size();
         return l;
      }
      else  // move r as root, if any
      {
         if (r) r->par = NULL;
         delete(x);
         return r;
      }
   }
   int* kth_element(size_t k)
   // return kth-element (1-based) in tree; NULL if kth-element does not
exist
   {
      for (Node *x = this; x; )
      {
         if (x->left)
         {
            if (x->left->treeSz >= k)
            {
```

```
            x = x->left;
            continue;
         }
         k -= x->left->treeSz;
      }
      if (x->freq >= k) return &x->val;
      k -= x->freq;
      x = x->right;
   }
   return NULL;
}
static size_t rank(Node *& root, int cmpval)
// return 0-based rank of cmpval; modifies root
{
   if (!root) return 0;
   root = root->splay(cmpval);
   size_t ret = 0;
   if (root->left)
      ret += root->left->treeSz;
   if (cmpval > root->val)
      ret += root->freq;
   return ret;
}
};
```

EXTRAS

MATRIX DETERMINANT

```
int det(VVI V)
{
    int N = V.size() , ret = 1 , div = 1;
    for(int j = 0; j < N; ++j)
    {
        ret *= V[j][j];
        for(int i = j+1; i< N ; ++i)
        {
            div *= V[j][j];
            for(int k = N-1; k >= j; --k)
                V[i][k] = V[i][k]*V[j][j] - V[j][k]*V[i][j];

        }
    }
    return ret/div;
}
```


BINARY SEARCH ---> [NO] YES & NO [YES] CASES

```
int binarySearch(int hi , int lo)
{
    //for(int i = 0; i < 100; ++i) --> for doubles
    while( lo > hi )
    {
        //for: no no [no] yes yes yes
        int mid = lo + (hi-lo+1)/2;
        //for: no no no [yes] yes yes
        //  mid = lo + (hi-lo)/2;
        if( eval(mid) == true )
            hi = mid - 1;
        //  hi = mid;
        else
            lo = mid;
        //  lo = mid+1;
    }
    if( eval(lo) == true  )return -1; //there is no NO
//  if( eval(lo) == false )return -1; // there is no YES
    return lo;
}
```

TIPS-TRICKS

- Number of Fixed Length Path between two vertices is the power of the adjacency matrix (without distance)
- For a shortest path of length K you need to compute SHP[K+1][i][j] = min(SHP[K][i][p] + SHP[1][p][j]) for al p, this could be fastened with matrix multiplication, instead of A[i][j] =sum(d[i][p]*d[p][j])
- Eulerian circuit exist only when degree of every node is EVEN or OUTdegree = INdegree, path exist when this condition holds and 2 of the vertex are missing one of the degrees to be equal.
- For finding if a graph has a cycle and finding one we perform a DFS, when we enter a node we paint it gray, and when exiting we paint it black, if during DFS we found a gray NODE it means there is a cycle, be careful edges to the parent doesn't count.
- For Number of minimum spanning trees in a graph diagonals should be nodes degree and an edge should be -1, then cover last row and las column and find the determinant of the resulting matrix.

- For finding an inscribed circle in a polygon just create a function R (x,y) which computes the maximum radius of an inscribed circle (minimum distance from the center to any edge) and iterate X and Y with binary search (x and y) should be inside the polygon.

- PICKS THEOREM: the area of a lattice polygon is the number of latties points included in it + the half of the number of points in its boundaries, this can be used to find the number of lattice points inside it. S = I + B/2 , then similarly I = S - B/2

- Grundy Function, the grundy function of a game gives the equivalent of having G(n) balls in a pile for the game of NIM.
- INCLUSION-EXCLUSION principle: sum of all sets - sum of intersections of two sets + sum of intersection of 3 sets - sum of intersections of 4 sets and so on.
- DERANGEMENT: !n = (n-1)*(!(n-1)+!(n-2)) --> !0 = 1 , !1 = 0
- EXTENDED GCD solves the ecuation ax + by = GCD(a,b) , solves ax = 1 (mod b) and by = 1 (mod a) for these last two do egcd(a,PRIME) and the x will be the inverse multiplicative mod that PRIME.
- CATALAN NUMBERS: having X and Y, catalan numbers gives the number of cominations of n letters so that for any prefix there are not more Ys than Xs. Number of paths in a nxn square for Cn is for nxn square.

$$C_0 = 1 \quad ; \quad C_n = \frac{2(2n+1)}{n+2} C_{n-1}$$

- PARTITION OF A NUMBER
  - p(k , n) = 0 | k > n
  - p(k , n) = 1 | k = n
  - p(k , n) = p(k+1 , n-k) + p(k , n-k) | otherwise
- the function p(k,n) gives the number of partitions of the number n with numbers less than or equal to k.
  - The number of partitions of n into no more than k parts is the same as the number of partitions of n into parts no larger than k.
  - The number of partitions of n into no more than k parts is the same as the number of partitions of n + k into exactly k parts.
- EULER FORMULA for EDGES, VERTICES and FACES
  - V - E + F = 2
- STIRLING NUMBERS
  - Is the number of ways of partition a set of n objects into k non-empty subsets
  - S(n+1, k) = k*S(n, k) + S(n , k-1) if(k > 0)
  - S(n,0) = S(0,k) = 0
  - S(0,0) = 1

DEFINES & TEMPLATES

```
#define DEBUG(x) cout << #x << ": " << x << endl
#define SZ(a) int((a).size())
#define FOREACH(it,c) for(typeof((c).begin()) it=(c).begin();it!=(c).end
();++it)
#define FOREACHR(it,c) for(typeof((c).rbegin()) it=(c).rbegin();it!=
(c).rend();++it)
#define ALL(c) (c).begin(),(c).end()
typedef __uint_128_t LONG;
// 64-bits
#define BITCOUNT(mask) ( __builtin_popcountll((mask)) )
#define LOWESTSETBIT(mask) ( __builtin_ctzll((mask)) )
#define HIGHESTSETBIT(mask) ( sizeof(long long)*8-1-__builtin_clzll
((mask)) )
```

JAVA EXAMPLE

```
import java.io.*;
import java.math.*;
import java.util.*;
public class Main {
   public static void main (String[] args) throws IOException {
      (new Main()).run();
   }
   BigInteger Choose(int n, int k) {
           if (n < 0 || k < 0 || k > n)  // invalid n or k
              return BigInteger.ZERO;
           if (k > n-k)k = n-k;
           BigInteger res = BigInteger.ONE;
           for (int i = 1; i <= k; ++i, --n)
              res = res.multiply(new BigInteger(String.valueOf(n)))
                       .divide(new BigInteger(String.valueOf(i)));
           return res;
   }
   public void run() throws IOException {
           BufferedReader in = new BufferedReader(new InputStreamReader
(System.in));
           String line = in.readLine();
           StringTokenizer tok = new StringTokenizer(line);
           int N = Integer.parseInt(tok.nextToken());
           int K = Integer.parseInt(tok.nextToken());
           System.out.println(Choose(N, K));
   }
}
```