Contents

1 Data structures

1.1 Order Statistics Tree

```
#include <ext/pb_ds/assoc_container.hpp>
#include <ext/pb_ds/tree_policy.hpp>
#include <ext/pb_ds/detail/standard_policies.hpp> // test me
using namespace __gnu_pbds; // pb_ds
typedef tree <pii , null_type , less <pii >, rb_tree_tag ,
  tree_order_statistics_node_update > tree;
tree.find_by_order(k) // iterator to the kth element
tree.order_of_key(k) // how many strictly less than k
```

1.2 Implicit Treap

```
typedef node * pnode;

struct node {
  int p, v, cnt;
  pnode l, r;
};

int cnt(pitem it){
  return it ? it->cnt : 0;
}

void upd_cnt(pitem it){
  if (it) it->cnt = cnt(it->l) + cnt(it->r) + 1;
}

void merge(pnode &t, pnode l, pnode r){
  if (!l || !r)
    t = l ? l : r;
  else if(l->p > r->p)
    merge(l->r, l->r, r), t = l;
  else
    merge(r->l, l, r->l), t = r;
  upd_cnt(t);
}

void split(pnode t, pnode &l, pnode &r, int key, int add = 0){
  if (!t) return void( l = r = 0 );
  int cur_key = add + cnt(t->l);
  if (key <= cur_key)
    split(t->l, l, t->l, key, add), r = t;
  else
    split(t->r, t->r, r, key, add + 1 + cnt(t->l)), l = t;
  upd_cnt(t);
}
```

1.3 Persistent Segment Tree

```
struct node{
  int sz,x,y;
  node* l;
  node* r;
  node():sz(0),x(0),y(0),l(NULL),r(NULL){};
  node(int _x,int _y,int _sz,node* _l,node* _r):x(_x),y(_y),sz(_sz),l(_l),r(_r
      ){};
  node(int _x,int _y,int _sz):x(_x),y(_y),sz(_sz){};
  node* L(){
    if(l)return l;
    return l=new node(x,(x+y)/2,0);
  }
  node* R(){
    if(r)return r;
    return r=new node((x+y)/2,y,0);
  }
  int szL(){
    return (l?l->sz:0);
  }
  int szR(){
    return (r?r->sz:0);
  }
};
int arr[MaxN],backH[MaxN];
pair<int,int> vec[MaxN];
node* ST[MaxN];
int query(node* t,node* _t,int k){
  if(t->x+1==t->y)
    return t->x;
  if(t->szL()-_t->szL()<k)
    return query(t->R(),_t->R(),k-(t->szL()-_t->szL()));
  else
    return query(t->L(),_t->L(),k);
}
node* upd(node* t,int k){
  if(t->x+1==t->y)
    return new node(t->x,t->y,t->sz+1);
  if(k<(t->x+t->y)/2)
    return new node(t->x,t->y,t->sz+1,upd(t->L(),k),t->r);
```

```
  else
    return new node(t->x,t->y,t->sz+1,t->l,upd(t->R(),k));
}
```

1.4 DP Divide and Conquer

Name	Original Recurrence	Sufficient Condition of Applicability	Original Complexity	Optimized Complexity
Convex Hull Optimization1	$dp[i] = min_{j<i}\{dp[j] + b[j] \star a[i]\}$	$b[j] \geq b[j+1]$ ~~optionally~~ $a[i] \leq a[i+1]$	$O(n^2)$	$O(n)$
Convex Hull Optimization2	$dp[i][j] = min_{k<j}\{dp[i-1][k] + b[k] * a[j]\}$	$b[k] \geq b[k+1]$ ~~optionally~~ $a[j] \leq a[j+1]$	$O(kn^2)$	$O(kn)$
Divide and Conquer Optimization	$dp[i][j] = min_{k<j}\{dp[i-1][k] + C[k][j]\}$	$A[i][j] \leq A[i][j+1]$	$O(kn^2)$	$O(knlogn)$
Knuth Optimization	$dp[i][j] = min_{i<k<j}\{dp[i][k] + dp[k][j]\} + C[i][j]$	$A[i,j-1] \leq A[i,j] \leq A[i+1,j]$	$O(n^3)$	$O(n^2)$

1.5 DP Convex Hull Trick

```
//dp[i]=min{dp[j]+b[j]*a[i]|0<=j<i}
struct line{
  lli m,c,x,id;
} CH[MaxN], X;

double intersect(line A,line B){
  return (double) (B.c-A.c)/(A.m-B.m);
}

line CH[MaxN],X;
lint H,dp[MaxN],B[MaxN],C[MaxN];
void add(line A){
  lli i, j, p, q;
  while (H){
    A.x = intersect(A,CH[H-1]);
    if(A.x<=CH[H-1].x) H--;
    else break;
  }
  CH[H++]=A;
}
lli query(lint x){
  lli i, j, p, q, a, b, m;
  a=0, b=H;
  while (a+1 < b){
    m=(a+b)/2;
    if (CH[m].x <= x) a=m;
    else b=m;
  }
  return dp[CH[a].id]+B[CH[a].id]*x;
}
```

1.6 Sparse Table

```
const int maxN = 5e5, maxPot = 20;
int n, a[maxN], sparse[maxN][maxPot], logg[maxN];
void fillSparse(){
    logg[1] = 0;
    for(int i = 0; i < maxN; ++i)
        logg[i] = logg[i/2]+1;

    int mid;
    for(int indx = 0; indx < n; ++indx)
        sparse[indx][0] = indx;
    for(int i = 1; i < maxPot; ++i){
        for(int indx = 0; i < n; ++i){
            mid = min(n-1, indx+(1<<(i-1)));
            sparse[indx][i] = a[sparse[mid][i-1]] > a[sparse[indx][i-1]]?
                sparse[mid][i-1] : sparse[indx][i-1];
        }
    }
}

//regresa el indice del maximo en el rango
//si hay varios, regresa el indice del primero
int leftMostQuery(int l, int r){
    int size = logg[r-l+1];
    int maxiL = sparse[l][size], maxiR = sparse[r-(1<<size)+1][size];
    return a[maxiR] > a[maxiL]? maxiR : maxiL;
}
```

2 Geometry

2.1 Miscellany

```
struct PT {
  double x, y;
  PT() {}
  PT(double x, double y) : x(x), y(y) {}
  PT(const PT &p) : x(p.x), y(p.y)    {}
  PT operator + (const PT &p)  const { return PT(x+p.x, y+p.y); }
```

```
  PT operator - (const PT &p)  const { return PT(x-p.x, y-p.y); }
  PT operator * (double c)     const { return PT(x*c,   y*c  ); }
  PT operator / (double c)     const { return PT(x/c,   y/c  ); }
  bool operator < (const PT &p) const { return x<p.x||(x==p.x&&y<p.y); }
};

double dot(PT p, PT q)     { return p.x*q.x+p.y*q.y; }
double dist2(PT p, PT q)   { return dot(p-q,p-q); }
double cross(PT p, PT q)   { return p.x*q.y-p.y*q.x; }

PT RotateCCW90(PT p)   { return PT(-p.y,p.x); }
PT RotateCW90(PT p)    { return PT(p.y,-p.x); }
PT RotateCCW(PT p, double t){
  return PT(p.x*cos(t)-p.y*sin(t), p.x*sin(t)+p.y*cos(t));
}

// project point c onto line through a and b
PT ProjectPointLine(PT a, PT b, PT c) {
  return a + (b-a)*dot(c-a, b-a)/dot(b-a, b-a);
}
// project point c onto line segment through a and b
PT ProjectPointSegment(PT a, PT b, PT c) {
  double r = dot(b-a,b-a);
  if (fabs(r) < EPS) return a;
  r = dot(c-a, b-a)/r;
  if (r < 0) return a;
  if (r > 1) return b;
  return a + (b-a)*r;
}

// compute distance from c to segment between a and b
double DistancePointSegment(PT a, PT b, PT c) {
  return sqrt(dist2(c, ProjectPointSegment(a, b, c)));
}

// compute distance between point (x,y,z) and plane ax+by+cz=d
double DistancePointPlane(double x, double y, double z,
                          double a, double b, double c, double d)
{
  return fabs(a*x+b*y+c*z-d)/sqrt(a*a+b*b+c*c);
}

// determine if lines from a to b and c to d are parallel or collinear
bool LinesParallel(PT a, PT b, PT c, PT d) {
  return fabs(cross(b-a, c-d)) < EPS;
}

bool LinesCollinear(PT a, PT b, PT c, PT d) {
  return LinesParallel(a, b, c, d)
      && fabs(cross(a-b, a-c)) < EPS
      && fabs(cross(c-d, c-a)) < EPS;
}

// determine if line segment from a to b intersects with
// line segment from c to d
bool SegmentsIntersect(PT a, PT b, PT c, PT d) {
  if (LinesCollinear(a, b, c, d)) {
    if (dist2(a, c) < EPS || dist2(a, d) < EPS ||
      dist2(b, c) < EPS || dist2(b, d) < EPS) return true;
    if (dot(c-a, c-b) > 0 && dot(d-a, d-b) > 0 && dot(c-b, d-b) > 0)
      return false;
    return true;
  }
  if (cross(d-a, b-a) * cross(c-a, b-a) > 0) return false;
  if (cross(a-c, d-c) * cross(b-c, d-c) > 0) return false;
  return true;
}

bool segmentLineIntersection(PT a, PT b, PT c, PT d){
    return cross(d-c, a-c)*cross(d-c, b-c) < EPS;
}

PT ComputeLineIntersection(PT a, PT b, PT c, PT d) {
  b=b-a; d=c-d; c=c-a;
  return a + b*cross(c, d)/cross(b, d);
}

PT ComputeCircleCenter(PT a, PT b, PT c) {
  b=(a+b)/2;
  c=(a+c)/2;
  return ComputeLineIntersection(b, b+RotateCW90(a-b), c, c+RotateCW90(a-c));
}

vector<PT> CircleLineIntersection(PT a, PT b, PT c, double r) {
  vector<PT> ret;
  b = b-a;
  a = a-c;
  double A = dot(b, b);
```

```
  double B = dot(a, b);
  double C = dot(a, a) - r*r;
  double D = B*B - A*C;
  if (D < -EPS) return ret;
  ret.push_back(c+a+b*(-B+sqrt(D+EPS))/A);
  if (D > EPS)
    ret.push_back(c+a+b*(-B-sqrt(D))/A);
  return ret;
}

vector<PT> CircleCircleIntersection(PT a, PT b, double r, double R) {
  vector<PT> ret;
  double d = sqrt(dist2(a, b));
  if (d > r+R || d+min(r, R) < max(r, R)) return ret;
  double x = (d*d-R*R+r*r)/(2*d);
  double y = sqrt(r*r-x*x);
  PT v = (b-a)/d;
  ret.push_back(a+v*x + RotateCCW90(v)*y);
  if (y > 0)
    ret.push_back(a+v*x - RotateCCW90(v)*y);
  return ret;
}

//Given 2 points and radius get the two center of circles determined by them
vector<PT> circle2PtsRad(PT p, PT q, double r) {
  vector<PT> ret;
  PT m = (p + q)/2;
  double d2 = dist2(p, q);
  p = p - q;
  double det = r * r / d2 - 0.25;
  if (det < 0.0) return ret;
  double h = sqrt(det);
  ret.push_back(PT(m.x + p.y * h, m.y - p.x * h));
  ret.push_back(PT(m.x - p.y * h, m.y + p.x * h))
  return ret;
}

double ComputeSignedArea(const vector<PT> &p) {
  double area = 0;
  for(int i = 0; i < p.size(); i++) {
    int j = (i+1) % p.size();
    area += p[i].x*p[j].y - p[j].x*p[i].y;
  }
  return area / 2.0;
}

double ComputeArea(const vector<PT> &p) {
  return fabs(ComputeSignedArea(p));
}

PT ComputeCentroid(const vector<PT> &p) {
  PT c(0,0);
  double scale = 6.0 * ComputeSignedArea(p);
  for (int i = 0; i < p.size(); i++){
    int j = (i+1) % p.size();
    c = c + (p[i]+p[j])*(p[i].x*p[j].y - p[j].x*p[i].y);
  }
  return c / scale;
}

// tests whether or not a given polygon (in CW or CCW order) is simple
bool IsSimple(const vector<PT> &p) {
  for (int i = 0; i < p.size(); i++) {
    for (int k = i+1; k < p.size(); k++) {
      int j = (i+1) % p.size();
      int l = (k+1) % p.size();
      if (i == l || j == k) continue;
      if (SegmentsIntersect(p[i], p[j], p[k], p[l]))
        return false;
    }
  }
  return true;
}

//Determine if point q is in possibly non-convex polygon. return 1 for
    strictly interior points. 0 for strictly interior points, and 0 or 1 for
    the remaining points.
bool PointInPolygon(vector<PT> &p, PT q){
    bool c = 0;
    for(int i = 0; i < p.size(); i++){
        int j = (i +1) % p.size();
        if(((p[i].y > q.y) != (p[j].y > q.y)) && q.x < p[i].x + (p[j].x - p[i
            ].x) * (q.y - p[i].y)/(p[j].y - p[i].y)
            c = !c;
}
return c;
}
```

```
bool PointOnPolygon(const vector<PT> &p, PT q) {
  for (int i = 0; i < p.size(); i++)
    if (dist2(ProjectPointSegment(p[i], p[(i+1)%p.size()], q), q) < EPS)
      return true;
    return false;
}

//cutting polygon
pair<vector<PT>, vector<PT> > CutPolygon(const vector<PT> &p, PT r, PT s){
    vector<PT> pleft, pright;
    PT q;
    int i;
    double sidei, sidej;
    for(i = 0; i < p.size(); i++){
        int j = (i + 1) % p.size();
        sidei = cross(s - r, p[i] - r);
        sidej = cross(s - r, p[j] - r);
        if(fabs(sidei) < EPS){
            pleft.push_back(p[i]);
            pright.push_back(p[i]);
        }
        else if(sidei > 0)
            pleft.push_back(p[i]);
        else
            pright.push_back(p[i]);
        if(sidei*sidej < EPS){
            if(LinesCollinear(r, s, p[i], p[j])) continue;
            q = ComputeLineIntersection(r, s, p[i], p[j]);
            //if(!(sqrt(dist2(q, p[i])) < EPS || sqrt(dist2(q, p[j])) < EPS)){
                pleft.push_back(q);
                pright.push_back(q);
            //}
        }
    }
    return make_pair(pleft, pright);
}

//Todavia no se ha testeado. Usar bajo su propio riesgo
vector<PT> convexPolygonIntersection(vector<PT> &p, vector<PT> &q){
    int i, j, k, l;
    vector<PT> intersection;
    for(i = 0; i < p.size(); i++)
        if(pointInPolygon(q, p[i]) || pointOnPolygon(q, p[i]))
            intersection.push_back(p[i]);
    for(i = 0; i < q.size(); i++)
        if(pointInPolygon(p, q[i]) || pointOnPolygon(p, q[i]))
            intersection.push_back(q[i]);
    for(i = 0; i < p.size(); i++){
        j = (i+1) % p.size();
        for(k = 0; k < q.size(); k++){
            l = (k+1) % q.size();
            if(!linesParallel(p[i], p[j], q[k], q[l]) && segmentsIntersect(p[i
                ], p[j], q[k], q[l]))
                intersection.push_back(computeLineIntersection(p[i], p[j], q[k
                    ], q[l]));
        }
    }
    sort(intersection.begin(), intersection.end());
    intersection.erase(unique(intersection.begin(), intersection.end()),
        intersection.end());

    PT mc = accumulate(intersection.begin(), intersection.end(), PT(0,0)) /
        intersection.size();

    sort(intersection.begin(), intersection.end(), [&](const PT &a, const PT &
        b){return atan2((a-mc).y, (a-mc).x) < atan2((b-mc).y, (b-mc).x); } );
    return intersection;
}
```

2.2 Convex Hull

```
vector < PT > ConvexHull(vector < PT > &P){
  sort(P.begin(), P.end());
  vector < PT > U, L;
  for(int i = 0; i < P.size(); i++){
    while(L.size() > 1 && cross(L[L.size()-1]-L[L.size()-2], P[i]-L[L.size()
        -2]) > 0)
      L.pop_back();
    L.push_back(P[i]);
  }
  if(L.size() > 1) L.pop_back();
  for(int i = P.size()-1; i >= 0; i--){
    while(U.size() > 1 && cross(U[U.size()-1]-U[U.size()-2], P[i]-U[U.size()
        -2]) > 0)
      U.pop_back();
    U.push_back(P[i]);
  }
  if(U.size() > 1) U.pop_back();
  L.insert(L.end(), U.begin(), U.end());
```

```
  return L;
}
```

2.3 Pick's Theorem

Si P es un polígono (sin lados que se corten) con vértices de coordenadas enteras entonces su área está dada por: $Area(P) = I + F/2 - 1$ donde I es el número de puntos de coordenadas enteras dentro de P y F es el número de puntos de coordenadas sobre la frontera de P.

3 Strings

3.1 KMP

```
vector<int> build_PI(string &P) { ///Visualizando indexado en 1
    ///PI[i] guarda el tamanio del prefijo mas grande de P que es sufijo de P
        [0...i]
    vector<int> PI(P.size());
    PI[0]=0;
    int k=0;
    int i;
    for(i=1;i<P.size();i++) {
        while(k>0 && P[k]!=P[i]) k=PI[k-1];
        if(P[k]==P[i]) k++;
        PI[i]=k;
    }
    return PI;
}

void KMP_Match(string &T , string &P) {
    vector<int> PI=build_PI(P);
    int q=0;
    for(int i=0;i<T.size();i++) {
        while(q>0 && P[q]!=T[i]) q=PI[q-1];
        if(P[q]==T[i]) q++;
        if(q==P.size()) {
            cout<<"Match con shift "<<i<<"\n";
            q=PI[q-1];
        }
    }
}
```

3.2 Suffix Array, LCP

```
// O(nlogn)
vector<int> suffix_array(string& s) {
    s.push_back('$');
    int N = s.size();
    vector<int> p(N), c(N);
    {  // k = 0
        vector<pair<char, int>> a(N);
        for (int i = 0; i < N; ++i)
            a[i] = {s[i], i};
        sort(a.begin(), a.end());
        for (int i = 0; i < N; ++i)
            p[i] = a[i].second;
        for (int i = 1; i < N; ++i)
            c[p[i]] = a[i].first == a[i - 1].first ? c[p[i - 1]] : c[p[i - 1]]
                + 1;
    }
    for (int k = 0; (1 << k) < N; ++k) {
        for (int i = 0; i < N; ++i)
            p[i] = (p[i] - (1 << k) + N) % N;
        {  // Counting sort
            vector<int> cnt(N), pos(N), p_new(N);
            for (auto x : c)
                cnt[x]++;
            for (int i = 1; i < N; ++i)
                pos[i] = pos[i - 1] + cnt[i - 1];
            for (auto x : p)
                p_new[pos[c[x]]++] = x;
            p = p_new;
        }
        vector<int> c_new(N);
        for (int i = 1; i < N; ++i) {
            pii prev = {c[p[i - 1]], c[(p[i - 1] + (1 << k)) % N]};
            pii now = {c[p[i]], c[(p[i] + (1 << k)) % N]};
            c_new[p[i]] = now == prev ? c_new[p[i - 1]] : c_new[p[i - 1]] + 1;
        }
        c = c_new;
    }
    s.pop_back();
    return vector<int>(p.begin() + 1, p.end());
}

// lcp[i] = lcp(i, i+1)
// lcp(x, y) = min(lcp[x], lcp[x+1], ... lcp[y-1])
vector<int> lcp_array(string& s, vector<int>& p) {
    int N = s.size();
    vector<int> c(N), lcp(N);
    for (int i = 0; i < N; ++i)
        c[p[i]] = i;
    int k = 0;
```

```
    for (int i = 0; i < N; ++i) {
        if (c[i] < N - 1) {
            int j = p[c[i] + 1];
            while (max(i, j) + k < N && s[i + k] == s[j + k])
                k++;
            lcp[c[i]] = k;
            k = max(k - 1, 0);
        }
    }
    return lcp;
}
```

3.3 Rolling Hash

Mod: 1e9+7, prime=2003, inv=137793311
Mod: 32416188191, prime=2003, inv=12623378327
Mod: 1e9+7, prime=2011, inv=76081552
Mod: 32416188191, prime=2011, inv=17811978096

```
struct hasher {
    int b = 311, m;
    vector<int> h, p;
    hasher(string s, int _m) : m(_m), h(s.size()+1), p(s.size()+1) {
        p[0] = 1; h[0] = 0;
        for (int i = 0; i < s.size(); ++i) {
            p[i+1] = (ll)p[i] * b % m;
            h[i+1] = ((ll)h[i] * b + s[i]) % m;
        }
    }
    int hash(int l, int r) {
        return (h[r+1] + m - (ll)h[l] * p[r-l+1] % m) % m;
    }
};
```

3.4 Aho-Corasick

Importante: ¿Qué pasa en el problema si un string está contenido en otro dentro de tu set base de strings?

```
const int K = 26;

struct Vertex {
    int next[K];
    bool leaf = false;
    int p = -1;
    char pch;
    int link = -1;
    int go[K];

    Vertex(int p=-1, char ch='$') : p(p), pch(ch) {
        fill(begin(next), end(next), -1);
        fill(begin(go), end(go), -1);
    }
};

vector<Vertex> t(1);

void add_string(string const& s) {
    int v = 0;
    for (char ch : s) {
        int c = ch - 'a';
        if (t[v].next[c] == -1) {
            t[v].next[c] = t.size();
            t.emplace_back(v, ch);
        }
        v = t[v].next[c];
    }
    t[v].leaf = true;
}

int go(int v, char ch);

int get_link(int v) {
    if (t[v].link == -1) {
        if (v == 0 || t[v].p == 0)
            t[v].link = 0;
        else
            t[v].link = go(get_link(t[v].p), t[v].pch);
    }
    return t[v].link;
}

int go(int v, char ch) {
    int c = ch - 'a';
    if (t[v].go[c] == -1) {
        if (t[v].next[c] != -1)
            t[v].go[c] = t[v].next[c];
        else
            t[v].go[c] = v == 0 ? 0 : go(get_link(v), ch);
    }
    return t[v].go[c];
}
```

```

//para el caso en que uno esta contenido en otro
for(int i = 0; i < t.size(); ++i){
    if(t[get_link(i)].leaf)
        t[i].leaf = true;
}
```

3.5 Manacher (Palindromes)

```
vector<int> manacher(string s) {
    string t;
    for(auto c: s) {
        t += string("#") + c;
    }
    auto res = manacher_odd(t + "#");
    return vector<int>(begin(res) + 1, end(res) - 1);
}

vector<int> manacher_odd(string s) {
    int n = s.size();
    s = "$" + s + "^";
    vector<int> p(n + 2);
    for(int i = 1; i <= n; i++) {
        while(s[i - p[i]] == s[i + p[i]]) {
            p[i]++;
        }
    }
    return vector<int>(begin(p) + 1, end(p) - 1);
}
```

In computer science, the **longest common prefix array** (**LCP array**) is an auxiliary data structure to the suffix array. It stores the lengths of the longest common prefixes (LCPs) between all pairs of consecutive suffixes in a sorted suffix array.

For example, if *A* := [aab, ab, abaab, b, baab] is a suffix array, the longest common prefix between *A*[1] = aab and *A*[2] = ab is *a* which has length 1, so *H*[2] = 1 in the LCP array *H*. Likewise, the LCP of *A*[2] = ab and *A*[3] = abaab is ab, so *H*[3] = 2.

Max common prefix of substrings starting at indices I and j

]. Then the answer to this query will be $\min(lcp[i],\ lcp[i+1],\ \dots,\ lcp[j-1])$.

Number of different substrings

$$\sum_{i=0}^{n-1}(n-p[i]) - \sum_{i=0}^{n-2}\mathrm{lcp}[i] = \frac{n^2+n}{2} - \sum_{i=0}^{n-2}\mathrm{lcp}[i]$$

4 Graphs

4.1 Lowest Common Ancestor

```
// Constant Query
int N,val[MaxN],C,fstT[MaxN],H[MaxN],pot2[MaxN*2],pot[40];
int sparse[MaxN*2][logN];
lli subtree[MaxN],total;
vector<int> ady[MaxN];

void dfs(int x,int past){
  int i,j,p,q,h;
  fstT[x]=C;
  H[x]=H[past]+1;
  subtree[x]=val[x];
  sparse[C++][0]=x;
  for(i=0;i<ady[x].size();i++){
    if(ady[x][i]==past)continue;
    dfs(ady[x][i],x);
    subtree[x]+=subtree[ady[x][i]];
    sparse[C++][0]=x;
  }
}

void doSparse(){
  int i,j,p,q;
  total=subtree[1];
  for(i=0,j=1;j<=C;j*=2,i++) pot2[j]=i;
  for(i=1,j=0;j<logN;j++,i*=2) pot[j]=i;
  for(i=3;i<=C;i++) if(!pot2[i])
    pot2[i]=pot2[i-1];
  for(i=1,p=1;i<logN;i++,p*=2)for(j=0;j+2*p<=C;j++)
    sparse[j][i]=(H[sparse[j][i-1]]<H[sparse[j+p][i-1]]?sparse[j][i-1]:sparse[
        j+p][i-1]);
}

int LCA(int x,int y){
  x=fstT[x],y=fstT[y];
  if(x>y)swap(x,y);
  int h = pot2[y-x+1];
  return (H[sparse[x][h]]<H[sparse[y-pot[h]+1][h]]?sparse[x][h]:sparse[y-pot[h
      ]+1][h]);
}

// inside int main(), before any LCA query
dfs(1,0)
doSparse()
```

4.2 Bipartite Matching

```
// Hopcroft Karp BPM
// O(E sqrt V) near-linear in random graphs
// define variables A and B equal to the cardinality of both sets.
int pair_A[MAXA], pair_B[MAXB];
bool adj[MAXA][MAXB];

int queue[MAXN];
int qs, qe;
#define resetQueue() qs = qe = 0
#define queueNotEmpty (qs < qe)
#define push(x) queue[qe++] = x
#define pop() queue[qs++]

int dist[MAXA];

bool matching_BFS(){
  resetQueue();
  for (int i = 0; i < A; i++){
    if (pair_A[i] == NIL){
      dist[i] = 0;
      push(i);
    } else {
      dist[i] = INF;
    }
  }
  dist[NIL] = INF;
  while (queueNotEmpty){
    int curr = pop();
    if (dist[curr] < dist[NIL])
      for (int i = 0; i < B; i++)
        if (adj[curr][i] && dist[pair_B[i]] == INF){
          dist[pair_B[i]] = dist[curr] + 1;
          push(pair_B[i]);
        }
  }
  return dist[NIL] < INF;
}

bool matching_DFS(int x){
  if (x == NIL) return true;
  for (int i = 0; i < B; i++)
    if (adj[x][i] && dist[pair_B[i]] == dist[x] + 1 && matching_DFS(pair_B[i])
        ){
      pair_B[i] = x;
      pair_A[x] = i;
      return true;
    }
  dist[x] = INF;
  return false;
}

int matching(void){
  int size = 0;
  fill(pair_A, pair_A + MAXA, NIL);
  fill(pair_B, pair_B + MAXB, NIL);
  while (matching_BFS())
    for (int i = 0; i < A; i++)
      if (pair_A[i] == NIL && matching_DFS(i))
        size++;
  return size;
}
```

4.3 Max Flow - Dinic's Algorithm

```
struct edge {
  int node, next, cap, flow;
};

edge g[MAXE*2];
int start[MAXV], nextEdge; // init start to 0s and nextEdge to 2

int addEdge(int a, int b, int c){
  g[nextEdge] = {b, start[a], c, 0};
  start[a] = nextEdge++;
  g[nextEdge] = {a, start[b], 0, 0};
  start[b] = nextEdge++;
}

// s->source, t->sink, n->total no. nodes
int maxFlow(){
  int tot = 0;
  static int q[MAXV], z[MAXV], d[MAXV], p[MAXV], qs, qe, curr;

  while (true){
    fill(d, d + n, MAXV);
    d[s] = qs = qe = 0;
    q[qe++] = s;

    while (qs < qe){
```

```
      curr = q[qs++];
      z[curr] = start[curr];
      if (d[curr] == d[t]) continue;
      for (int i = start[curr]; i; i = g[i].next)
        if (g[i].cap - g[i].flow > 0 &&
            d[g[i].node] > d[curr] + 1){
          d[g[i].node] = d[curr] + 1;
          q[qe++] = g[i].node;
        }
    }

    if (d[t] == MAXV) return tot;

    curr = s;
    while (true){
      while (z[curr] && (g[z[curr]].cap - g[z[curr]].flow <= 0 ||
             d[g[z[curr]].node] != d[curr] + 1))
        z[curr] = g[z[curr]].next;

      if (!z[curr]){
        if (curr == s) break;
        curr = g[p[d[curr]-1]^1].node;
        d[g[p[d[curr]]].node] = -INF;
        continue;
      }

      p[d[curr]] = z[curr];
      curr = g[z[curr]].node;

      if (curr == t){
        int m = INF;
        for (int i = 0; i < d[t]; i++)
          m = min(m, g[p[i]].cap - g[p[i]].flow);
        for (int i = 0; i < d[t]; i++){
          g[p[i]].flow += m;
          g[p[i]^1].flow -= m;
        }
        tot += m;
        curr = s;
      }
    }
  }
}
```

4.4 Strongly Connected Components

```
vi dfs_num, dfs_low, S, visited; // global variables

void tarjanSCC(int u) {
  dfs_low[u] = dfs_num[u] = dfsNumberCounter++;
  S.push_back(u);
  visited[u] = 1;
  for (int j = 0; j < (int)AdjList[u].size(); j++) {
    ii v = AdjList[u][j];
    if (dfs_num[v.first] == UNVISITED)
      tarjanSCC(v.first);
    if (visited[v.first]) // condition for update
      dfs_low[u] = min(dfs_low[u], dfs_low[v.first]);
  }

  if (dfs_low[u] == dfs_num[u]) {
    printf("SCC %d:", ++numSCC);
    while (1) {
      int v = S.back(); S.pop_back(); visited[v] = 0;
      printf(" %d", v);
      if (u == v) break;
    }
    printf("\n");
  }
}

// inside int main()
dfs_num.assign(V, UNVISITED);
dfs_low.assign(V, 0);
visited.assign(V, 0);
dfsNumberCounter = numSCC = 0;
for (int i = 0; i < V; i++)
  if (dfs_num[i] == UNVISITED)
    tarjanSCC(i);
```

4.5 SCC2

```
vector<int> adj[maxN], transpose[maxN];
bool visited[maxN];
stack<int> next;
void dfs1(int v){
    visited[v] = true;
    for(int son: adj[v])
        if(!visited[son])
            dfs1(son);
```

```
    next.push(v);
}

void dfs2(int v){
    visited[v] = true;
    for(int son: transpose[v])
        if(!visited[son])
            dfs2(son);
}

void scc(){
    //supose nodes are from 0 to v-1
    //fill adj and transpose
    stack<int> next;
    fill(visited, visited+v, false);
    for(int i = 0; i < v; ++i){
        if(!visited[i])
            dfs1(i);
    }

    fill(visited, visited+v, false);
    cout<<"stringly connected components are:"<<endl;
    while(!next.empty()){
        int aux = next.top(); next.pop();
        if(!visited[aux]){
            dfs2(aux, transpose, visited);
            cout<<endl;
        }
    }
```

4.6 Articulation Points / Bridges

```
void articulationPointAndBridge(int u) {
  dfs_low[u] = dfs_num[u] = dfsNumberCounter++;
  for (int j = 0; j < (int)AdjList[u].size(); j++) {
    ii v = AdjList[u][j];

    if (dfs_num[v.first] == UNVISITED) {
      dfs_parent[v.first] = u;
      if (u == dfsRoot) rootChildren++;
      articulationPointAndBridge(v.first);

      if (dfs_low[v.first] >= dfs_num[u])
            articulation_vertex[u] = true;

      if (dfs_low[v.first] > dfs_num[u])
        printf(" Edge (%d, %d) is a bridge\n", u, v.first);

      dfs_low[u] = min(dfs_low[u], dfs_low[v.first]);
    } else if (v.first != dfs_parent[u])
      dfs_low[u] = min(dfs_low[u], dfs_num[v.first]);
  }
}

// inside int main()
dfsNumberCounter = 0; dfs_num.assign(V, UNVISITED); dfs_low.assign(V, 0);
dfs_parent.assign(V, 0); articulation_vertex.assign(V, 0);
printf("Bridges:\n");
for (int i = 0; i < V; i++)
  if (dfs_num[i] == UNVISITED) {
    dfsRoot = i; rootChildren = 0; articulationPointAndBridge(i);
    articulation_vertex[dfsRoot] = (rootChildren > 1);
  } // special case
printf("Articulation Points:\n");
for (int i = 0; i < V; i++)
  if (articulation_vertex[i])
    printf(" Vertex %d\n", i);
```

4.7 Matrix exponetiation

```
struct matrix{
    vector< vector<ll> > m;
    ll mod, sz;
    ll mod2;

    matrix (ll n,ll modc) : sz(n),m(n), mod(modc) {
        for(int i=0;i<n;i++)
            m[i].resize(n);
        mod2=mod*mod;
    }

    matrix operator*(matrix b)
    {
        matrix ans(sz,mod);
        for(int i=0;i<sz;i++)
        for(int j=0;j<sz;j++)
            for(int u=0;u<sz;u++)
            {
                ans.m[i][u]+=m[i][j]*b.m[j][u];
                if(ans.m[i][u]>=mod2)
```

```
                    ans.m[i][u]-=mod2;
            }
        for(int i=0;i<sz;i++)
            for(int j=0;j<sz;j++)
                ans.m[i][j]%=mod;
        return ans;
    }

    matrix pow(ll e)
    {
        if(e==1)
            return *this;
        matrix x =pow(e/2);
        x=(x*x);
        if(e&1)
            x=(x*(*this));
        return x;
    }
};
```

4.8 Heavy Light Decomposition

```
#include <bits/stdc++.h>
#define pb(x) push_back(x)
using namespace std;
typedef long long int lli;
const int MaxN=100001,logN=17;
struct BIT{
    vector<lli> B;
    BIT(int n):B(n+2){}
    lli query(int x){
        lli S=0;
        for(x++;x;x-=x&-x)S+=B[x];
        return S;
    }
    void up(int x,lli v){
        for(;x<B.size();x+=x&-x)B[x]+=v;
    }
    void update(int x,int y,lli v){
        up(++x,v);
        up(++y+1,-v);
    }
};
vector<BIT> HLD;
vector<int> ady[MaxN];
int Cact,Cadena[MaxN],CDad[MaxN],Cnum[MaxN],sz[MaxN],lca[MaxN][logN],H[MaxN],
    vis[MaxN];
int dfs(int x,int past){
    lca[x][0]=past,H[x]=H[past]+1;
    for(int i=1;i<logN;i++)
        lca[x][i]=lca[lca[x][i-1]][i-1];
    for(int d:ady[x]){
        if(d==past)continue;
        sz[x]+=dfs(d,x);
    }
    return ++sz[x];
}
void doHLD(int x,int past){
    int i,j,p,q;
    vis[x]=1,Cadena[x]=Cact;
    Cnum[x]=Cnum[past]+1;
    for(i=p=q=0;i<ady[x].size();i++){
        if(vis[ady[x][i]])continue;
        if(p<sz[ady[x][i]])p=sz[ady[x][i]],q=i;
    }
    if(!p){
        HLD.pb(BIT(Cnum[x]+1));
        return;
    }
    doHLD(ady[x][q],x);
    for(i=0;i<ady[x].size();i++){
        if(vis[ady[x][i]])continue;
        CDad[++Cact]=x;
        doHLD(ady[x][i],0);
    }
}
int LCA(int x,int y){
    if(H[x]<H[y])swap(x,y);
    for(int i=logN-1;i>=0;i--)
        if(H[x]-(1<<i)>=H[y])x=lca[x][i];
    if(x==y)return x;
    for(int i=logN-1;i>=0;i--)
        if(lca[x][i]!=lca[y][i])
            x=lca[x][i],y=lca[y][i];
    return lca[x][0];
}
void update(int x,int y){
    int v=LCA(x,y),i,j,p,q;
    p=Cadena[x];
```

```
    while(Cadena[v]!=p){
        HLD[p].update(0,Cnum[x],1);
        x=CDad[p];
        p=Cadena[x];
    }
    HLD[p].update(Cnum[v],Cnum[x],1);
    p=Cadena[y];
    while(Cadena[v]!=p){
        HLD[p].update(0,Cnum[y],1);
        y=CDad[p];
        p=Cadena[y];
    }
    HLD[p].update(Cnum[v],Cnum[y],1);
    HLD[p].update(Cnum[v],Cnum[v],-2);
}
lli query(int x,int y){
    if(H[x]<H[y])swap(x,y);
    return HLD[Cadena[x]].query(Cnum[x]);
}
int main(){
    lli N,Q,i,j,p,q,t,T,M,a,b;
    char op;
    Cnum[0]=-1;
    cin>>N>>Q;
    for(i=1;i<N;i++){
        cin>>p>>q;
        ady[p].pb(q);
        ady[q].pb(p);
    }
    dfs(1,0);
    doHLD(1,0);
    for(i=0;i<Q;i++){
        cin>>op>>p>>q;
        if(op=='P')update(p,q);
        if(op=='Q')cout<<query(p,q)<<"\n";
    }
}
```

4.9 König's theorem

Let $G = (V, E)$ be a bipartite graph, and let the vertex set V be partitioned into left set L and right set R. Suppose that M is a maximum matching for G.
Let U be the set of unmatched vertices in L (possibly empty), and let Z be the set of vertices that are either in U by alternating paths (pahts that alternate between edges that are in the matching and edges that are not in the matching).
Then, $K = (L \setminus Z) \cup (R \cap Z)$, is a minimum vertex cover of cardinality equal to M.

5 Mathematics

5.1 Miller-Rabin / Pollard's Rho

```
vector<ll> toTest = {2, 3, 5, 7, 11, 13, 17, 19, 23, 29, 31, 37};

ll modPow(ll a, ll m, ll n){
  ll ans = 1;
  for (; m; m /= 2){
    if(m % 2)
      ans = (ans * a) % n;
    a = (a * a) % n;
  }
  return ans;
}

bool isPrimeMR(ll n){
  if (n == 2) return true;
  if (n % 2 == 0 || n <= 1) return false;
  ll m;
  int k = 0;
  for(m = n-1; m % 2 == 0; m /= 2) k++;
  for(auto a: toTest){
    if (a >= n) break;
    ll x = modPow(a, m, n);
    if (x == 1 || x == n-1) continue;
    int j;
    for(j = 0; j < k-1; j++){
      x = (x * x) % n;
      if(x == n-1) break;
    }
    if (j == k - 1) return false;
  }
  return true;
}

ll factor(ll n){
  ll A = 2 + rand() % (n - 2);
  ll B = 2 + rand() % (n - 2);
  auto f = [&](ll x){ return x * (x + A) % n + B; };
  ll d, x = 2, y = 2;
  do{
    x = f(x);
    y = f(f(y));
    d = __gcd(x >= y ? x - y : y - x, n);
```

```
  } while (d == 1);
  return d;
}

void factorize(ll n, vector<ll> &v){
  if (n == 1) return;
  if (isPrimeMR(n)){
    v.push_back(n);
    return;
  }
  ll f;
  do f = factor(n);
  while (f == n);
  factorize(f, v);
  factorize(n/f, v);
}
```

5.2 Tonelli-Shanks

```
/* Takes as input an odd prime p and n < p and returns r
 * such that r * r = n [mod p]. */
long tonelli_shanks(long n, long p) {
  long s = 0;
  long q = p - 1;
  while ((q & 1) == 0) { q /= 2; ++s; }
  if (s == 1) {
    long r = pow_mod(n, (p+1)/4, p);
    if ((r * r) % p == n) return r;
    return 0;
  }
  // Find the first quadratic non-residue z by brute-force search
  long z = 1;
  while (pow_mod(++z, (p-1)/2, p) != p - 1);
  long c = pow_mod(z, q, p);
  long r = pow_mod(n, (q+1)/2, p);
  long t = pow_mod(n, q, p);
  long m = s;
  while (t != 1) {
    long tt = t;
    long i = 0;
    while (tt != 1) {
      tt = (tt * tt) % p;
      ++i;
      if (i == m) return 0;
    }
    long b = pow_mod(c, pow_mod(2, m-i-1, p-1), p);
    long b2 = (b * b) % p;
    r = (r * b) % p;
    t = (t * b2) % p;
    c = b2;
    m = i;
  }
  if ((r * r) % p == n) return r;
  return 0;
}
```

5.3 Extended GCD

```
int mod(int a, int b) {
  return ((a%b)+b)%b;
}
// returns d = gcd(a,b); finds x,y such that d = ax + by.
//Solution of ax + by = c are given by
//(x, y) = (x0 + b*t/d, y0 + a*t/d)

int extended_euclid(int a, int b, int &x, int &y) {
  int xx = y = 0;
  int yy = x = 1;
  while (b) {
    int q = a/b;
    int t = b; b = a%b; a = t;
    t = xx; xx = x-q*xx; x = t;
    t = yy; yy = y-q*yy; y = t;
  }
  return a;
}

// finds all solutions to ax = b (mod n)
VI modular_linear_equation_solver(int a, int b, int n) {
  int x, y;
  VI solutions;
  int d = extended_euclid(a, n, x, y);
  if (!(b%d)) {
    x = mod (x*(b/d), n);
    for (int i = 0; i < d; i++)
      solutions.push_back(mod(x + i*(n/d), n));
  }
  return solutions;
}

```

```
// computes b such that ab = 1 (mod n), returns -1 on failure
int mod_inverse(int a, int n) {
  int x, y;
  int d = extended_euclid(a, n, x, y);
  if (d > 1) return -1;
  return mod(x,n);
}
```

5.4 Chinese Remainder Theorem

```
// Chinese remainder theorem (special case): find z such that
// z % x = a, z % y = b.  Here, z is unique modulo M = lcm(x,y).
// Return (z,M).  On failure, M = -1.
PII chinese_remainder_theorem(int x, int a, int y, int b) {
  int s, t;
  int d = extended_euclid(x, y, s, t);
  if (a%d != b%d) return make_pair(0, -1);
  return make_pair(mod(s*b*x+t*a*y,x*y)/d, x*y/d);
}

// Chinese remainder theorem: find z such that
// z % x[i] = a[i] for all i.  Note that the solution is
// unique modulo M = lcm_i (x[i]).  Return (z,M).  On
// failure, M = -1.  Note that we do not require the a[i]'s
// to be relatively prime.
PII chinese_remainder_theorem(const VI &x, const VI &a) {
  PII ret = make_pair(a[0], x[0]);
  for (int i = 1; i < x.size(); i++) {
    ret = chinese_remainder_theorem(ret.second, ret.first, x[i], a[i]);
    if (ret.second == -1) break;
  }
  return ret;
}
```

5.5 Linear Sieve

```
const int maxP = 1e5 + 5, maxN = 100 +5;

vector<ll> prime;
bool primo[maxP];

void sieve () {
    fill (primo, primo + maxP, true);
    primo[0] = primo[1] = false;
    for (int i = 2; i < maxP; ++i) {
        if (primo[i]) prime.push_back(i);
        for (int j = 0; j < prime.size() && i * prime[j] < maxP; ++j) {
            primo[i * prime[j]] = false;
            if (i % prime[j] == 0) break;
        }
    }
}
```

6 Combinatorics

6.1 Inclusion-Exclusion principle

$$E_m = \sum_{k=0}^{N-m} \left((-1)^k \binom{m+k}{k} S_{m+k} \right)$$

$$L_m = \sum_{k=0}^{N-m} \left((-1)^k \binom{m+k-1}{k} S_{m+k} \right)$$

6.2 Conteo

- $\binom{m+n}{n} = \sum_{i=0}^{n} \binom{m}{i} * \binom{n}{n-i}$, $n * \binom{2n-1}{n-1} = \sum_0^n \binom{n}{k}^2 k$
- $(-1)^0\binom{2n}{0}^2 + (-1)^1\binom{2n}{1}^2 + (-1)^2\binom{2n}{2}^2 + ... + (-1)^{2n}\binom{2n}{2n}^2 = (-1)^n\binom{2n}{n}$

Notas de conteo: El producto de dos cosas podemos relacionarlo con la elección de dos decisiones distintas. Cuando "falta un elemento" (ie. formula de pascal) podemos agregar un elemento y revisar todas sus posibilidades. Para contar de otra forma podemos interpretar la situación de elección, coloración, caminos, etc.

6.3 Games

El Grundy number de una juego es igual a 0 si es un posición donde se pierde inmediatamente, en caso contrario, es el MEX de los subjuegos a los que se puede llegar en una transición. Se recomienda usar sets para encontrar rápidamente el MEX.

```
int T,N, dp[35][35]; /// DP(i, j) is mex for substring S[i..j]
string s,w[35];
int DP(int x, int y){
  if(x > y) return 0;
  if(dp[x][y] != -1) return dp[x][y];
  set<int> mex;
  for(int i=0,lw=w[i].length(); i<N; i++, lw=w[i].length())
    for(int j=x; j+lw-1 <= y; j++)
      if(s.substr(j, lw) == w[i])
        mex.insert(DP(x, j-1)^DP(j+lw, y));
  for(int m = 0; ; m++)
    if(mex.find(m) == mex.end()){
        dp[x][y] = m; break;
    }
  return dp[x][y] = max(dp[x][y], 0);
```

```
}///answer DP(0,s.size()-1) 1 wins 0 lose
```

El teorema de Sprague grundy dice que sí un juego es partido en subjuegos, se calcula el Grundy number de cada uno, y se hace el xor de los valores, si da 0, entonces es posicion perdera, sino, es ganadora.

6.4 Probability

6.4.1 Cálculo de funciones de distribución dadas otras

$f(x|y) = \frac{f(x,y)}{f(y)}$

Técnica de la transformación: Si $z = g(x,y)$, hacemos $x = h(z,y)$ $f(z,y) = f(xoh(z,y),y) * |J|$ donde J es la derivada parcial de x respecto a z.

Si $z = g(x,y)$ y $w = h(x,y)$ $f(z,w) = f(x,y) * |J|$, donde J es el Jacobiano con la primer fila para las x's y la 2da fila para las y's.

6.4.2 Distribución Poisson

La variable Poisson mide la cantidad de eventos que ocurren en un intervalo de tiempo dado, área dada, longitud, étc. Su función de distribución de probabilidad es $f(x) = \frac{(\Lambda)^x * e^{-\Lambda}}{x!}, x = 0, 1, \ldots$. Tiene Media y Varianza igual a λ.

6.4.3 Distribución exponencial

La función de distribución es: $f(x) = \frac{e^{-\frac{x}{\theta}}}{\theta}, x > 0, \theta > 0$, tiene Media igual a θ y varianza igual a θ^2

6.4.4 Distribucion Uniforme

Esta distribución ocurre cuando todos los eventos tienen la misma probabilidad. Si el intervalo de resultados está entre a y b, entonces $f(x) = \frac{1}{b-a}, a \le x \le b$. La Media es $\frac{b-a}{2}$ y la varianza $\frac{(b-a)^2}{12}$

6.4.5 Esperanza

$$E[g(x)] = \begin{cases} \sum_{\forall x} g(x) f(x) & \text{si x es discreta} \\ \int_{\forall x} g(x) f(x) dx & \text{si x es continua} \end{cases}$$

La esperanza multivariada se comporta de forma similar La esperanza de una suma es la suma de las esperanzas. **Teorema de la Varianza**: $V(x) = E(x^2) - E(x)^2$. La varianza condicional se comporta de forma similar.

$$E(g(x)|y) = \begin{cases} \sum_{\forall x} g(x) f(x|y), \text{ si } x, y \text{ son discretas} \\ \int_{\forall x} g(x) f(x|y) dx \text{ , si } x, y \text{ son continuas} \end{cases}$$

6.4.6 Métodos

Para obtener números con cierta distribución:

Generamos una lista de números aleatorios(distribución continua), si F es la función de distribución hacia la cual queremos que se distribuyan los números basta con hacer $F(u)$ para todo número u de la lista original.

6.5 Catalan Numbers

$$C_n = \frac{1}{n+1}\binom{2n}{n} = \frac{(2n)!}{(n+1)!\,n!} = \prod_{k=2}^{n} \frac{n+k}{k} = \binom{2n}{n} - \binom{2n}{n+1}$$

- C_n counts the number of expressions containing n pairs of parentheses which are correctly matched.
- C_n is the number of different ways $n+1$ factors can be completely parenthesized.
- C_n is the number of full binary trees with $n+1$ leaves.
- C_n is the number of non-isomorphic ordered trees with n vertices.
- C_n is the number of monotonic lattice paths along the edges of a grid with $n \times n$ square cells, which do not pass above the diagonal.
- C_n is the number of monotonic lattice paths along the edges of a grid with $n \times n$ square cells, which do not pass above the diagonal.
- C_n is the number of different ways a convex polygon with $n+2$ sides can be cut into triangles by connecting vertices with straight lines.

1, 1, 2, 5, 14, 42, 132, 429, 1430, 4862, 16796, 58786, 208012, 742900, 2674440, 9694845, 35357670, 129644790, 477638700, 1767263190, 6564120420, 24466267020, 91482563640, 343059613650, 1289904147324, 4861946401452

6.6 Stirling Numbers

$$\begin{bmatrix} n+1 \\ k \end{bmatrix} = n \begin{bmatrix} n \\ k \end{bmatrix} + \begin{bmatrix} n \\ k-1 \end{bmatrix}$$

Counts the number of permutations of n elements with k cycles.

$$\begin{Bmatrix} n+1 \\ k \end{Bmatrix} = k \begin{Bmatrix} n \\ k \end{Bmatrix} + \begin{Bmatrix} n \\ k-1 \end{Bmatrix}$$

Counts the number of partitions of n elements with k subsets.

7 Other Algorithms

7.1 BIT Hacks

```
// Next higher number with same number of set bits
int snoob(int x) {
  int y = x & -x, z = x + y;
  return z | ((x ^ z) >> 2) / y;
}
```

7.2 Longest Increasing Subsequence

```
vector<int> LIS(vector<int> &x){
  vector<int> m, p(x.size());
  m.push_back(-1);
  for (int i = 0; i < x.size(); i++){
    int lo = 1, hi = m.size()-1, mid;
    while (lo <= hi){
      mid = (lo + hi) / 2;
      if (x[m[mid]] < x[i])
        lo = mid + 1;
      else
        hi = mid - 1;
    }
    p[i] = m[lo-1];
    if (lo >= m.size()) m.push_back(0);
    m[lo] = i;
  }
  vector<int> l(m.size()-1);
  for (int i = m[m.size()-1], j = m.size()-2; i != -1; j--, i = p[i])
    l[j] = x[i];
  return l;
}
```

7.3 2-SAT

```
//We have a vertex representing a var and other for his negation.
//Every edge stored in G represents an implication. To add an equation of the form a||b, use addor(a, b)
//MAX=max cant var, n=cant var
#define addor(a, b) (G[neg(a)].pb(b), G[neg(b)].pb(a))
vector<int> G[MAX*2];
//idx[i]=index assigned in the dfs
//lw[i]=lowest index(closer from the root) reachable from i
int lw[MAX*2], idx[MAX*2], qidx;
stack<int> q;
int qcmp, cmp[MAX*2];
//verdad[cmp[i]]=valor de la variable i
bool verdad[MAX*2+1];

int neg(int x) { return x>=n? x-n : x+n;}
void tjn(int v){
  lw[v]=idx[v]=++qidx;
  q.push(v), cmp[v]=-2;
  forall(it, G[v]){
    if(!idx[*it] || cmp[*it]==-2){
      if(!idx[*it]) tjn(*it);
      lw[v]=min(lw[v], lw[*it]);
    }
  }
  if(lw[v]==idx[v]){
    int x;
    do{x=q.top(); q.pop(); cmp[x]=qcmp;}while(x!=v);
    verdad[qcmp]=(cmp[neg(v)]<0);
    qcmp++;
  }
}
//remember to CLEAR G!!!
bool satisf(){//O(n)
  memset(idx, 0, sizeof(idx)), qidx=0;
  memset(cmp, -1, sizeof(cmp)), qcmp=0;
  forn(i, n){
    if(!idx[i]) tjn(i);
    if(!idx[neg(i)]) tjn(neg(i));
  }
  forn(i, n) if(cmp[i]==cmp[neg(i)]) return false;
  return true;
}
```

7.4 Gauss Jordan

```
// Gauss-Jordan elimination with full pivoting.
//
// Uses:
//   (1) solving systems of linear equations (AX=B)
//   (2) inverting matrices (AX=I)
//   (3) computing determinants of square matrices
//
// Running time: O(n^3)
//
// INPUT:    a[][] = an nxn matrix
//           b[][] = an nxm matrix
//
// OUTPUT:   X      = an nxm matrix (stored in b[][])
//           A^{-1} = an nxn matrix (stored in a[][])
//           returns determinant of a[][]

#include <iostream>
#include <vector>
#include <cmath>

using namespace std;
```

```

const double EPS = 1e-10;

typedef vector<int> VI;
typedef double T;
typedef vector<T> VT;
typedef vector<VT> VVT;

T GaussJordan(VVT &a, VVT &b) {
  const int n = a.size();
  const int m = b[0].size();
  VI irow(n), icol(n), ipiv(n);
  T det = 1;

  for (int i = 0; i < n; i++) {
    int pj = -1, pk = -1;
    for (int j = 0; j < n; j++) if (!ipiv[j])
      for (int k = 0; k < n; k++) if (!ipiv[k])
  if (pj == -1 || fabs(a[j][k]) > fabs(a[pj][pk])) { pj = j; pk = k; }
    if (fabs(a[pj][pk]) < EPS) { cerr << "Matrix is singular." << endl; exit
        (0); }
    ipiv[pk]++;
    swap(a[pj], a[pk]);
    swap(b[pj], b[pk]);
    if (pj != pk) det *= -1;
    irow[i] = pj;
    icol[i] = pk;

    T c = 1.0 / a[pk][pk];
    det *= a[pk][pk];
    a[pk][pk] = 1.0;
    for (int p = 0; p < n; p++) a[pk][p] *= c;
    for (int p = 0; p < m; p++) b[pk][p] *= c;
    for (int p = 0; p < n; p++) if (p != pk) {
      c = a[p][pk];
      a[p][pk] = 0;
      for (int q = 0; q < n; q++) a[p][q] -= a[pk][q] * c;
      for (int q = 0; q < m; q++) b[p][q] -= b[pk][q] * c;
    }
  }

  for (int p = n-1; p >= 0; p--) if (irow[p] != icol[p]) {
    for (int k = 0; k < n; k++) swap(a[k][irow[p]], a[k][icol[p]]);
  }

  return det;
}
```

7.5 Simplex

```
// Two-phase simplex algorithm for solving linear programs of the form
//
//     maximize     c^T x
//     subject to   Ax <= b
//                  x >= 0
//
// INPUT: A -- an m x n matrix
//        b -- an m-dimensional vector
//        c -- an n-dimensional vector
//        x -- a vector where the optimal solution will be stored
//
// OUTPUT: value of the optimal solution (infinity if unbounded
//         above, nan if infeasible)
//
// To use this code, create an LPSolver object with A, b, and c as
// arguments.  Then, call Solve(x).

#include <iostream>
#include <iomanip>
#include <vector>
#include <cmath>
#include <limits>

using namespace std;

typedef long double DOUBLE;
typedef vector<DOUBLE> VD;
typedef vector<VD> VVD;
typedef vector<int> VI;

const DOUBLE EPS = 1e-9;

struct LPSolver {
  int m, n;
  VI B, N;
  VVD D;

  LPSolver(const VVD &A, const VD &b, const VD &c) :
    m(b.size()), n(c.size()), N(n + 1), B(m), D(m + 2, VD(n + 2)) {
```

```
    for (int i = 0; i < m; i++) for (int j = 0; j < n; j++) D[i][j] = A[i][j];
    for (int i = 0; i < m; i++) { B[i] = n + i; D[i][n] = -1; D[i][n + 1] = b[
        i]; }
    for (int j = 0; j < n; j++) { N[j] = j; D[m][j] = -c[j]; }
    N[n] = -1; D[m + 1][n] = 1;
  }

  void Pivot(int r, int s) {
    for (int i = 0; i < m + 2; i++) if (i != r)
      for (int j = 0; j < n + 2; j++) if (j != s)
        D[i][j] -= D[r][j] * D[i][s] / D[r][s];
    for (int j = 0; j < n + 2; j++) if (j != s) D[r][j] /= D[r][s];
    for (int i = 0; i < m + 2; i++) if (i != r) D[i][s] /= -D[r][s];
    D[r][s] = 1.0 / D[r][s];
    swap(B[r], N[s]);
  }

  bool Simplex(int phase) {
    int x = phase == 1 ? m + 1 : m;
    while (true) {
      int s = -1;
      for (int j = 0; j <= n; j++) {
        if (phase == 2 && N[j] == -1) continue;
        if (s == -1 || D[x][j] < D[x][s] || D[x][j] == D[x][s] && N[j] < N[s])
            s = j;
      }
      if (D[x][s] > -EPS) return true;
      int r = -1;
      for (int i = 0; i < m; i++) {
        if (D[i][s] < EPS) continue;
        if (r == -1 || D[i][n + 1] / D[i][s] < D[r][n + 1] / D[r][s] ||
          (D[i][n + 1] / D[i][s]) == (D[r][n + 1] / D[r][s]) && B[i] < B[r]) r
              = i;
      }
      if (r == -1) return false;
      Pivot(r, s);
    }
  }

  DOUBLE Solve(VD &x) {
    int r = 0;
    for (int i = 1; i < m; i++) if (D[i][n + 1] < D[r][n + 1]) r = i;
    if (D[r][n + 1] < -EPS) {
      Pivot(r, n);
      if (!Simplex(1) || D[m + 1][n + 1] < -EPS) return -numeric_limits<DOUBLE
          >::infinity();
      for (int i = 0; i < m; i++) if (B[i] == -1) {
        int s = -1;
        for (int j = 0; j <= n; j++)
          if (s == -1 || D[i][j] < D[i][s] || D[i][j] == D[i][s] && N[j] < N[s
              ]) s = j;
        Pivot(i, s);
      }
    }
    if (!Simplex(2)) return numeric_limits<DOUBLE>::infinity();
    x = VD(n);
    for (int i = 0; i < m; i++) if (B[i] < n) x[B[i]] = D[i][n + 1];
    return D[m][n + 1];
  }
};
```

7.6 Numeric Integration - Romberg's Method

```
#define eps 1e-7
#define N 12

double R[N + 1][N + 1];

// a, b: limits of integration
// F: function pointer to function to be integrated.
double romberg(double a, double b, double (*F)(double)){
    int i,j,k;
    double h = (b - a);

    R[0][0] = ( (*F)(a) + (*F)(b) ) * h / 2;

    for(i = 1 ; i <= N ; i++){
        h = h / 2;
        double sum = 0;
        for(k = 1 ; k < (1<<i) ; k += 2 )
            sum += (*F)(a + k * h);
        R[i][0] = R[i - 1][0] / 2 + sum * h;
        for(j = 1 ; j <= i ; j++)
            R[i][j] = R[i][j - 1] + ( R[i][j - 1] - R[i - 1][j - 1]) /
                ((1<<(2*j)) - 1);
    }
    return R[N][N];
}
```

7.7 Fast Fourier Transform

```
typedef complex<double> base;

void fft (vector<base> & a, bool invert) {
  int n = (int) a.size();

  for (int i=1, j=0; i<n; ++i) {
    int bit = n >> 1;
    for (; j>=bit; bit>>=1)
      j -= bit;
    j += bit;
    if (i < j)
      swap (a[i], a[j]);
  }

  for (int len=2; len<=n; len<<=1) {
    double ang = 2*(M_PI)/len * (invert ? -1 : 1);
    base wlen (cos(ang), sin(ang));
    for (int i=0; i<n; i+=len) {
      base w (1);
      for (int j=0; j<len/2; ++j) {
        base u = a[i+j],  v = a[i+j+len/2] * w;
        a[i+j] = u + v;
        a[i+j+len/2] = u - v;
        w *= wlen;
      }
    }
  }
  if (invert)
    for (int i=0; i<n; ++i)
      a[i] /= n;
}

const int mod = 7340033;
const int root = 5;
const int root_1 = 4404020;
const int root_pw = 1<<20;

void fft (vector<int> & a, bool invert) {
  int n = (int) a.size();

  for (int i=1, j=0; i<n; ++i) {
    int bit = n >> 1;
    for (; j>=bit; bit>>=1)
      j -= bit;
    j += bit;
    if (i < j)
      swap (a[i], a[j]);
  }

  for (int len=2; len<=n; len<<=1) {
    int wlen = invert ? root_1 : root;
    for (int i=len; i<root_pw; i<<=1)
      wlen = int (wlen * 1ll * wlen % mod);
    for (int i=0; i<n; i+=len) {
      int w = 1;
      for (int j=0; j<len/2; ++j) {
        int u = a[i+j],  v = int (a[i+j+len/2] * 1ll * w % mod);
        a[i+j] = u+v < mod ? u+v : u+v-mod;
        a[i+j+len/2] = u-v >= 0 ? u-v : u-v+mod;
        w = int (w * 1ll * wlen % mod);
      }
    }
  }
  if (invert) {
    int nrev = reverse (n, mod);
    for (int i=0; i<n; ++i)
      a[i] = int (a[i] * 1ll * nrev % mod);
  }
}

void multiply (const vector<int> & a, const vector<int> & b, vector<int> & res
    ) {
  vector<base> fa (a.begin(), a.end()),  fb (b.begin(), b.end());
  size_t n = 1;
  while (n < max (a.size(), b.size()))  n <<= 1;
  n <<= 1;
  fa.resize (n),  fb.resize (n);

  fft (fa, false),  fft (fb, false);
  for (size_t i=0; i<n; ++i)
    fa[i] *= fb[i];
  fft (fa, true);

  res.resize (n);
  for (size_t i=0; i<n; ++i)
    res[i] = (int) round(fa[i].real());
}
```

7.8 Fast Fourier Transform 2

```
typedef long double ld;
const int maxn = 1e5 + 10;
const ld PI = acos(-1);
typedef complex<ld> base;
base wlen_pw[2*maxn];
void fft (vector<base> & a, bool invert) {
  int n = (int) a.size();

  for (int i=1, j=0; i<n; ++i) {
    int bit = n >> 1;
    for (; j>=bit; bit>>=1)
      j -= bit;
    j += bit;
    if (i < j)
      swap (a[i], a[j]);
  }

  for (int len=2; len<=n; len<<=1) {
    ld ang = 2*PI/len * (invert ? -1 : 1);
    int len2 = len>>1;
    base wlen (cos(ang), sin(ang));
    wlen_pw[0] = base (1, 0);
    for (int i=1; i<len2; ++i)
      wlen_pw[i] = wlen_pw[i-1] * wlen;

    for (int i=0; i<n; i+=len) {
      for (int j=0; j<len2; ++j) {
        base u = a[i+j],  v = a[i+j+len2] * wlen_pw[j];
        a[i+j] = u + v;
        a[i+j+len2] = u - v;
      }
    }
  }
  if (invert)
    for (int i=0; i<n; ++i)
      a[i] /= n;
}

void multiply (const vector<int> & a, const vector<int> & b, vector<ll> & res)
     {
  vector<base> fa (a.begin(), a.end()),  fb (b.begin(), b.end());
  size_t n = 1;
  while (n < max (a.size(), b.size()))  n <<= 1;
  n <<= 1;
  fa.resize (n),  fb.resize (n);

  fft (fa, false),  fft (fb, false);
  for (size_t i=0; i<n; ++i)
    fa[i] *= fb[i];
  fft (fa, true);

  res.resize (n);
  for (size_t i=0; i<n; ++i){
    res[i] = ll (fa[i].real() + 0.5 * (fa[i].real() >= 0 ? 1 : -1));
  }
}
```

7.9 Hungarian Algorithm

```
vector<int> u (n+1), v (m+1), p (m+1), way (m+1);
for (int i=1; i<=n; ++i) {
  p[0] = i;
  int j0 = 0;
  vector<int> minv (m+1, INF);
  vector<char> used (m+1, false);
  do {
    used[j0] = true;
    int i0 = p[j0],  delta = INF,  j1;
    for (int j=1; j<=m; ++j)
      if (!used[j]) {
        int cur = a[i0][j]-u[i0]-v[j];
        if (cur < minv[j])
          minv[j] = cur,  way[j] = j0;
        if (minv[j] < delta)
          delta = minv[j],  j1 = j;
      }
    for (int j=0; j<=m; ++j)
      if (used[j])
        u[p[j]] += delta,  v[j] -= delta;
      else
        minv[j] -= delta;
    j0 = j1;
  } while (p[j0] != 0);
  do {
    int j1 = way[j0];
    p[j0] = p[j1];
    j0 = j1;
  } while (j0);
```

```
}

vector<int> ans (n+1);
for (int j=1; j<=m; ++j)
  ans[p[j]] = j;

int cost = -v[0];
```

8 Tricks de novatos

8.1 Ideas/estrategias

Si buscas obtener una estructura o construir algo con cierta propiedad, obtenerlo de forma random en varios intentos puede ser la mejor opción aunque la probabilidad de conseguirlo en 1 intento sea muy baja, revisa que los intentos necesarios para que la probabilidad de conseguirlo en algún momento tienda a 0 entre dentro del tiempo

8.2 Consejos para revisar el codigo

Revisa que los índices que están escritos en el código en verdad sean los índices que quieres

Revisa que no accedas a posiciones inexistentes(por ejemplo el punto anterior), o revisa si estás eliminando elementos de una queue o un vector que está vacío.Puede ser que en la máquina no se genere ningún error de ejecución

Revisa que los nombres de los vectores coincidan con el que quieres(para evitar hacer referencia a un vector, queue, etc distinto al que quieres)

Asigna nombres significativos para evitar el punto anterior, si vas a hacer un parche del codigo revisa si es que las variables a las que antes hacias referencia cambian o no.

Cuidado con dividir entre 0, o exceder la memoria del tipo de dato

9 Memes