Team Notebook

December 17, 2018

Contents

39 suffixarray	**26**	

1 2-SAT

```
struct TwoSAT
{
 static const int MAXV=1e5+5;
 int n, cnt;
 vector<int> g[MAXV], rg[MAXV]; //g=forward, rg=backward
 bool vis[MAXV];
 int order[MAXV], comp[MAXV];
 void init(int curn)
 {
  n=curn;
  for(int i=0;i<n;i++)
  {
   g[i].clear();
   rg[i].clear();
  }
 }
 void add(int u, int v)
 {
  g[u].push_back(v);
  rg[v].push_back(u);
 }
 void dfs1(int u)
 {
  vis[u] = true;
  for(auto it:g[u])
   if(!vis[it])
    dfs1(it);
  order[cnt++] = u;
 }
 void dfs2(int u, int c)
 {
  comp[u] = c;
  for(auto it:rg[u])
   if(comp[it]==-1)
    dfs2(it, c);
 }
 int solve(vector<int> &ans)
 {
  cnt=0;
  memset(vis, 0, sizeof(vis));
  for(int i=0;i<n;i++)
   if(!vis[i])
    dfs1(i);
  memset(comp, -1, sizeof(comp));
  int grp=0;
  for(int i=n-1;i>=0;i--)
  {
   int u=order[i];
   if(comp[u] == -1)
    dfs2(u, grp++);
  }
  for(int i=0;i<n;i+=2)
   if(comp[i]==comp[i^1])
    return 0;

  ans.clear();
  for(int i=0;i<n;i+=2)
  {
   int choose = (comp[i] > comp[i^1]) ? i : (i^1);
   ans.push_back(choose);
  }
  return 1;
 }
};
```

2 $2_{C}losest_{P}oints_{i}n_{2}D_{P}lane_{(}N_{l}og_{N}^{2})$

```
struct Point
{
 int x, y;
 Point operator -(Point p)
 {
  return {x-p.x, y-p.y};
 }
 int dist()
 {
  return x*x + y*y;
 }
};
bool by_x(Point &a, Point &b)
{
 return a.x < b.x;
}
bool by_y(Point &a, Point &b)
{
 return a.y < b.y;
}
int n, ans=1e18;
int a[N], pref[N];
Point pt[N];
int solve(int L, int R)
{
 if(L==R)
  return 1e18;
 int M=(L+R)/2;
 sort(pt+L, pt+R+1, by_x);
 int d=min(solve(L, M), solve(M+1, R));
 int midx=pt[L+(R-L+1)/2].x;
 vector<Point> v;
 for(int i=L;i<=R;i++)
 {
  if(Point{pt[i].x-midx, 0}.dist()<d)
  {
   v.push_back(pt[i]);
  }
 }
 sort(v.begin(), v.end(), by_y);
 for(int i=0;i<v.size();i++)
 {
  for(int j=i+1;j<v.size();j++)
  {
   if(Point{0, v[i].y-v[j].y}.dist()>d)
    break;
   d=min(d, (v[i]-v[j]).dist());
```

```
  }
 }
 return d;
}
```

3 Bridge$_{Tree_in_Graph}$

```
int tim=0, grp=1;
int u[N], v[N], comp[N];
bool vis[N], vis2[N], isBridge[M];
int tin[N], tout[N], minAncestor[N];
queue<int> Q[N];
vector<pair<int, int> > g[N];
vector<int> tree[N], vertices[N]; //Tree stores
    Bridge Tree, vertices stores the nodes in
    each component
void dfs(int k, int par)//bridges
{
 vis[k]=1;
 tin[k]=++tim;
 minAncestor[k]=tin[k];
 for(auto it:g[k])
 {
  if(it.first==par)
   continue;
  if(vis[it.first])
  {
   minAncestor[k]=min(minAncestor[k], tin[it.
       first]);
   continue;
  }
  dfs(it.first, k);
  minAncestor[k]=min(minAncestor[k], minAncestor[
      it.first]);
  if(minAncestor[it.first]>tin[k])
   isBridge[it.second]=1;
 }
 tout[k]=tim;
}
void dfs2(int k)
{
 int comp=grp;
 Q[comp].push(k);
 vis2[k]=1;
 while(!Q[comp].empty())
 {
  int u=Q[comp].front();
  Q[comp].pop();
  vertices[comp].push_back(u);
  for(auto it:g[u])
  {
   int v=it.first;
   int edgeidx=it.second;
   if(vis2[v])
    continue;
   if(isBridge[edgeidx])
   {
    grp++;
    tree[comp].push_back(grp);
    tree[grp].push_back(comp);
    dfs2(v);
   }
   else
   {
    Q[comp].push(v);
    vis2[v]=1;
   }
  }
 }
}
```

4 Centroid$_{Decomposition}$

```
int subtree[N], parentcentroid[N];
set<int> g[N];
void dfs(int k, int par)
{
 nodes++;
 subtree[k]=1;
 for(auto it:g[k])
 {
  if(it==par)
   continue;
  dfs(it, k);
  subtree[k]+=subtree[it];
 }
}
int centroid(int k, int par)
{
 for(auto it:g[k])
 {
  if(it==par)
   continue;
  if(subtree[it]>(nodes>>1))
   return centroid(it, k);
 }
 return k;
}
void decompose(int k, int par)
{
 nodes=0;
 dfs(k, k);
 int node=centroid(k, k);
 parentcentroid[node]=par;
 for(auto it:g[node])
 {
  g[it].erase(node);
  decompose(it, node);
 }
}
```

5 Convex$_{Hull_(Dynamic)}$

```
struct ConvexHullDynamic
```

```
{
 static const int INF=1e18;
 struct Line
 {
  int a, b; //y = ax + b
  double xLeft; //Stores the intersection wiith
      previous line in the convex hull. First
      line has -INF

  enum Type {line, maxQuery, minQuery} type;
  int val;

  explicit Line(int aa=0, int bb=0): a(aa), b(bb)
      , xLeft(-INF), type(Type::line), val(0) {}

  int valueAt(int x) const
  {
   return a*x + b;
  }
  friend bool isParallel(const Line &l1, const
      Line &l2)
  {
   return l1.a == l2.a;
  }
  friend double intersectX(const Line &l1, const
      Line &l2)
  {
   return isParallel(l1, l2)?INF:1.0*(l2.b-l1.b)
       /(l1.a-l2.a);
  }
  bool operator<(const Line& l2) const
  {
   if(l2.type == line)
    return this->a < l2.a;
   if(l2.type == maxQuery)
    return this->xLeft < l2.val;
   if(l2.type == minQuery)
    return this->xLeft > l2.val;
  }
 };
bool isMax;
set<Line> hull;
bool hasPrev(set<Line>::iterator it)
{
 return it!=hull.begin();
}
bool hasNext(set<Line>::iterator it)
{
 return it!=hull.end() && next(it)!=hull.end();
}
bool irrelevant(const Line &l1, const Line &l2,
    const Line &l3)
{
 return intersectX(l1, l3) <= intersectX(l1, l2)
     ;
}
bool irrelevant(set<Line>::iterator it)
{
 return hasPrev(it) && hasNext(it) && (
  (isMax && irrelevant(*prev(it), *it, *next(it)
      ))
  || (!isMax && irrelevant(*next(it), *it, *prev
      (it))));
}
//Updates xValue of line pointed by it
set<Line>::iterator updateLeftBorder(set<Line>::
    iterator it)
{
 if(isMax && !hasPrev(it) || !isMax && !hasNext(
     it))
  return it;
 double val=intersectX(*it, isMax?(*prev(it)):(*
     next(it)));
 Line temp(*it);
 it=hull.erase(it);
 temp.xLeft=val;
 it=hull.insert(it, temp);
 return it;
}
explicit ConvexHullDynamic(bool isMax): isMax(
    isMax) {}
void addLine(int a, int b) //Add ax + b in logN
    time
{
 Line l3=Line(a, b);
 auto it=hull.lower_bound(l3);

 //If parallel liune is already in set, one of
     the lines becomes irrelevant
 if(it!=hull.end() && isParallel(*it, l3))
 {
  if(isMax && it->b<b || !isMax && it->b>b)
   it=hull.erase(it);
  else
   return;
 }

 it=hull.insert(it, l3);
 if(irrelevant(it))
 {
  hull.erase(it);
  return;
 }
 //Remove lines which became irrelevant after
     inserting
 while(hasPrev(it) && irrelevant(prev(it)))
  hull.erase(prev(it));
 while(hasNext(it) && irrelevant(next(it)))
  hull.erase(next(it));
 //Update xLine
 it=updateLeftBorder(it);
 if(hasPrev(it))
  updateLeftBorder(prev(it));
 if(hasNext(it))
  updateLeftBorder(next(it));
}
int getBest(int x)
{
 Line q;
```

```
  q.val=x;
  q.type = isMax?Line::Type::maxQuery : Line::
      Type::minQuery;
  auto bestLine=hull.lower_bound(q);
  if(isMax)
   --bestLine;
  return bestLine->valueAt(x);
 }
};
```

6 Convex$_{H}ull_{(}Graham's_{S}can)$

```
struct point //Replace double with int if not
    required
{
 double x, y;
 point () {}
 point(int x, int y) : x(x), y(y) {}
 void operator =(const point &p)
 {
  x=p.x, y=p.y;
 }
 bool operator <(const point&p)
 {
  if(x==p.x)
   return y<p.y;
  return x<p.x;
 }
 point operator +(const point&p) const
 {
  point pt(x + p.x, y + p.y);
  return pt;
 }
 point operator -(const point&p) const
 {
  point pt(x - p.x, y - p.y);
  return pt;
 }
 double crossProduct(const point &p) const
 {
  return x * p.y - y * p.x;
 }
 int dotProduct(const point &p) const
 {
  return x * p.x + y * p.y;
 }
 double dist()
 {
  return x*x + y*y;
 }
};


bool comp(point &p1, point &p2)
{
 if(p1.x!=p2.x)
  return p1.x<p2.x;
 return p1.y<p2.y;
}

bool cw(point &a, point &b, point &c)
{
 int area=a.x*(b.y-c.y) + b.x*(c.y-a.y) + c.x*(a.
     y-b.y);
 return area<0;
}

bool ccw(point &a, point &b, point &c)
{
 int area=a.x*(b.y-c.y) + b.x*(c.y-a.y) + c.x*(a.
     y-b.y);
 return area>0;
}

vector<point> convex_hull(vector<point> &v)
{
 if(v.size()==1)
  return v;
 sort(v.begin(), v.end(), comp);
 point p1=v[0], p2=v.back();
 vector<point> up, down;
 up.push_back(p1);
 down.push_back(p1);
 for(int i=1;i<v.size();i++)
 {
  if(i==v.size()-1 || cw(p1, v[i], p2))
  {
   while(up.size()>=2 && !cw(up[up.size()-2], up[
       up.size()-1], v[i]))
    up.pop_back();
   up.push_back(v[i]);
  }
  if(i==v.size()-1 || ccw(p1, v[i], p2))
  {
   while(down.size()>=2 && !ccw(down[down.size()
       -2], down[down.size()-1], v[i]))
    down.pop_back();
   down.push_back(v[i]);
  }
 }
 for(int i=down.size()-2;i>0;i--)
  up.push_back(down[i]);
 return up;
}
```

7 DFS$_{_C}ycle_{D}etection_{i}n_{D}irected_{G}ra$

```
bool findLoop(int v)
{
 if(vis[v]==1)
  return 1;
 if(vis[v]==2)
  return 0;
 vis[v]=1;
 for(auto &it:g[v])
 {
```

```
  if(findLoop(it))
   return 1;
 }
 vis[v]=2;
 return 0;
}
bool checkLoop()
{
 fill(vis+1, vis+n+1, 0);
 for(int i=1;i<=n;i++)
 {
  if(!vis[i] && findLoop(i))
   return 1;
 }
 return 0;
}
```

8 DSU$_{OnTrees}$

```
int col[N], cnt[N], f[N], subtree[N], big[N], ans
    [N];
vector<int> g[N];
multiset<int> active;
void getsz(int v, int p)
{
 subtree[v]=1;
 for(auto u:g[v])
 {
  if(u==p)
   continue;
  getsz(u, v);
  subtree[v]+=subtree[u];
 }
}
void add(int v, int p, int x) //Function changes
    as per question,
{
 active.erase(cnt[col[v]]);
 f[cnt[col[v]]]-=col[v];
 cnt[col[v]]+=x;
 active.insert(cnt[col[v]]);
 f[cnt[col[v]]]+=col[v];
 for(auto u:g[v])
 {
  if(u!=p && !big[u])
   add(u, v, x);
 }
}
void computeans(int v)
{
 int maxf=*(--active.end());
 ans[v]=f[maxf];
}
void dfs(int v, int p, int keep)
{
 int mx = -1, bigChild = -1;
 for(auto u:g[v])
 {
  if(u!=p && subtree[u]>mx)
   mx=subtree[u], bigChild=u;
 }
 for(auto u:g[v])
 {
  if(u!=p && u!=bigChild)
   dfs(u, v, 0); //Run DFS on small children and
       clear them
 }
 if(bigChild!=-1)
 {
  dfs(bigChild, v, 1);
  big[bigChild]=1;
 }
 add(v, p, 1);
 //Now we have the information of subtree of v
 computeans(v);
 if(bigChild!=-1)
  big[bigChild]=0;
 if(keep==0)
  add(v, p, -1);
}
```

9 Dijkstra

Dijkstra with Path:

```
int arrival[N], departure[N], vis[N], parent[N];
vector<pair<int, int> > g[N];

void dijkstra(int source, int destination)
{
 for(int i=1;i<=n;i++)
 {
  arrival[i]=1e18;
  departure[i]=1e18;
  vis[i]=0;
 }
 arrival[source]=0;
 set<pair<int, int> > s;
 s.insert({0, source});
 while(!s.empty())
 {
  auto x = *(s.begin());
  s.erase(x);
  vis[x.second]=1;
  departure[x.second]=arrival[x.second];
  for(auto it:g[x.second])
  {
   if(arrival[it.first] > departure[x.second] +
       it.second)
   {
    s.erase({arrival[it.first], it.first});
    arrival[it.first]=departure[x.second] + it.
        second;
    s.insert({arrival[it.first], it.first});
    parent[it.first]=x.second;
   }
```

```
  }
 }
 if(!vis[destination])
 {
  cout<<"-1";
  return;
 }
 int v=destination;
 vector<int> ans;
 while(parent[v])
 {
  ans.push_back(v);
  v=parent[v];
 }
 ans.push_back(source);
 reverse(ans.begin(), ans.end());
 for(auto it:ans)
  cout<<it<<" ";
}
```

10 Extended$_Euclidean_Algorithm_(Extensive)$

```
int xgcd(int a, int b, int &x, int &y) //Returns
    GCD of A, B
{
 if(a==0)
 {
  x=0;
  y=1;
  return b;
 }
 int x1, y1;
 int d = xgcd(b % a, a, x1, y1);
 x = y1 - (b/a)*x1;
 y = x1;
 return d;
}
int modular_inverse(int a, int m)
{
 int x, y;
 int g=xgcd(a, m, x, y);
 if(g!=1)
  return -1;
 else
 {
  x=(x%m + m)%m;
  return x;
 }
}

void shift_solution(int &x, int &y, int a, int b,
     int cnt)
{
 x+=cnt*b;
 y-=cnt*a;
}

bool find_any_solution(int a, int b, int c, int &
    x0, int &y0)
{
 int g=xgcd(abs(a), abs(b), x0, y0);
 if(c%g!=0)
  return false;
 x0 *= c/g;
 y0 *= c/g;
 if(a<0)
  x0*=-1;
 if(b<0)
  y0*=-1;
 return true;
}

int find_all_solutions(int a, int b, int c, int
    minx, int maxx, int miny, int maxy) //Returns
     number of solutions with x[minx, maxx], y[
    miny, maxy]
{
 int x, y, g;
 if(!find_any_solution(a, b, c, x, y, g))
  return 0;
 a /= g;
 b /= g;

 int sign_a = a>0 ? +1 : -1;
 int sign_b = b>0 ? +1 : -1;

 shift_solution(x, y, a, b, (minx - x) / b);
 if (x < minx) shift_solution(x, y, a, b, sign_b)
     ;
 if (x > maxx) return 0;
 int lx1 = x;

 shift_solution(x, y, a, b, (maxx - x) / b);
 if (x > maxx) shift_solution(x, y, a, b, -sign_b
     );
 int rx1 = x;

 shift_solution(x, y, a, b, - (miny - y) / a);
 if (y < miny) shift_solution(x, y, a, b, -sign_a
     );
 if (y > maxy) return 0;
 int lx2 = x;

 shift_solution(x, y, a, b, - (maxy - y) / a);
 if (y > maxy) shift_solution(x, y, a, b, sign_a)
     ;
 int rx2 = x;

 if (lx2 > rx2)
  swap (lx2, rx2);
 int lx = max (lx1, lx2);
 int rx = min (rx1, rx2);

 return (rx - lx) / abs(b) + 1;
}
```

11 FFT(*Iterative*)

4 Call FFT with MOD:

```
typedef complex<double> base;

const double PI = acos(-1.0l);
const int N = 8e5+5;
const int Maxb = 19;
const int Maxp = 450;
const int MOD=13313;

vector<int> rev;
vector<base> omega;

void calc_rev(int n, int log_n) //Call this
    before FFT
{
 omega.assign(n, 0);
 rev.assign(n, 0);
 for(int i=0;i<n;i++)
 {
  rev[i]=0;
  for(int j=0;j<log_n;j++)
  {
   if((i>>j)&1)
    rev[i] |= 1<<(log_n-j-1);
  }
 }
}

void fft(vector<base> &A, int n, bool invert)
{
 for(int i=0;i<n;i++)
 {
  if(i<rev[i])
   swap(A[i], A[rev[i]]);
 }
 for(int len=2;len<=n;len<<=1)
 {
  double ang=2*PI/len * (invert?-1:+1);
  int half=(len>>1);

  base curomega(cos(ang), sin(ang));
  omega[0]=base(1, 0);

  for(int i=1;i<half;i++)
   omega[i]=omega[i-1]*curomega;

  for(int i=0;i<n;i+=len)
  {
   base t;
   int pu = i,
    pv = i+half,
    pu_end = i+half,
    pw = 0;
   for(; pu!=pu_end; pu++, pv++, pw++)
   {
    t=A[pv] * omega[pw];
    A[pv] = A[pu] - t;
    A[pu] += t;
   }
  }
 }

 if(invert)
  for(int i=0;i<n;i++)
   A[i]/=n;
}

void multiply(int n, vector<base> &A, vector<base
    > &B, vector<int> &C)
{
 fft(A, n, false);
 fft(B, n, false);
 for(int i=0;i<n;i++)
  A[i] *= B[i];
 fft(A, n, true);
 for(int i=0;i<n;i++)
 {
  C[i] = (int)(A[i].real() + 0.5);
  C[i] %= MOD;
 }
}

void Solve(int n, vector<int> &coeffA, vector<int
    > &coeffB, vector<int> &result, bool big1,
    bool big2) //Call 4 times: 00, 01, 10, 11
{
 vector<base> A(n), B(n);
 for(int i=0;i<n;i++)
 {
  A[i]=big1?coeffA[i]/Maxp : coeffA[i]%Maxp;
  B[i]=0;
 }
 for(int i=0;i<n;i++)
 {
  B[i]=big2?coeffB[i]/Maxp : coeffB[i]%Maxp;
 }
 vector<int> C(n);
 multiply(n, A, B, C);
 for(int i=0;i<n;i++)
 {
  int add=C[i];
  if(big1)
   add*=Maxp;
  if(big2)
   add*=Maxp;
  add%=MOD;
  result[i]+=add;
  result[i]%=MOD;
 }
}

void do_FFT(vector<int> &A, vector<int> &B,
    vector<int> &result)
{
```

```
 int n=1, bits=0;
 while(n<2*A.size() || n<2*B.size())
  n<<=1, bits++;
 result.assign(n, 0);
 calc_rev(n, bits);
 vector<int> tempA(A.begin(), A.end());
 vector<int> tempB(B.begin(), B.end());
 tempA.resize(n);
 tempB.resize(n);
 for(int i=0;i<2;i++)
 {
  for(int j=0;j<2;j++)
  {
   Solve(n, tempA, tempB, result, i, j);
  }
 }
}
```

Single Call without MOD:

```
typedef complex<double> base;

const double PI = acos(-1.0l);
const int N = 8e5+5;
const int Maxb = 19;
const int Maxp = 450;
const int MOD=13313;

vector<int> rev;
vector<base> omega;

void calc_rev(int n, int log_n) //Call this
    before FFT
{
 omega.assign(n, 0);
 rev.assign(n, 0);
 for(int i=0;i<n;i++)
 {
  rev[i]=0;
  for(int j=0;j<log_n;j++)
  {
   if((i>>j)&1)
    rev[i] |= 1<<(log_n-j-1);
  }
 }
}

void fft(vector<base> &A, int n, bool invert)
{
 for(int i=0;i<n;i++)
 {
  if(i<rev[i])
   swap(A[i], A[rev[i]]);
 }
 for(int len=2;len<=n;len<<=1)
 {
  double ang=2*PI/len * (invert? -1:+1);
  int half=(len>>1);

  base curomega(cos(ang), sin(ang));
  omega[0]=base(1, 0);

  for(int i=1;i<half;i++)
   omega[i]=omega[i-1]*curomega;

  for(int i=0;i<n;i+=len)
  {
   base t;
   int pu = i,
    pv = i+half,
    pu_end = i+half,
    pw = 0;
   for(; pu!=pu_end; pu++, pv++, pw++)
   {
    t=A[pv] * omega[pw];
    A[pv] = A[pu] - t;
    A[pu] += t;
   }
  }
 }

 if(invert)
  for(int i=0;i<n;i++)
   A[i]/=n;
}

void multiply(int n, vector<base> &A, vector<base
    > &B, vector<int> &C)
{
 fft(A, n, false);
 fft(B, n, false);
 for(int i=0;i<n;i++)
  A[i] *= B[i];
 fft(A, n, true);
 for(int i=0;i<n;i++)
 {
  C[i] = (int)(A[i].real() + 0.5);
 }
}


void Solve(int n, vector<int> &coeffA, vector<int
    > &coeffB, vector<int> &result)
{
 vector<base> A(n), B(n);
 for(int i=0;i<n;i++)
 {
  A[i]=coeffA[i];
  B[i]=0;
 }
 for(int i=0;i<n;i++)
 {
  B[i]=coeffB[i];
 }
 vector<int> C(n);
 multiply(n, A, B, C);
 for(int i=0;i<n;i++)
 {
```

```
  int add=C[i];
  result[i]+=add;
 }
}

void do_FFT(vector<int> &A, vector<int> &B,
    vector<int> &result)
{
 int n=1, bits=0;
 while(n<2*A.size() || n<2*B.size())
  n<<=1, bits++;
 result.assign(n, 0);
 calc_rev(n, bits);
 vector<int> tempA(A.begin(), A.end());
 vector<int> tempB(B.begin(), B.end());
 tempA.resize(n);
 tempB.resize(n);
 Solve(n, tempA, tempB, result);
}
```

12 Gaussian$_Elimination$

```
//Logic: https://math.stackexchange.com/questions
    /48682/maximization-with-xor-operator
struct Gaussian
{
 int no_of_bits = 20;
 vector<int> v;
 int set, origsize=0, redsize=0;

 void push(int val)
 {
  origsize++;
  if(val)
   v.push_back(val);
 }

 void clear()
 {
  v.clear();
  set=0, redsize=0;
 }

 void eliminate()
 {
  set = redsize = 0;
  for(int bit=0;bit<=no_of_bits;bit++)
  {
   bool check=false;
   for(int i=redsize;i<v.size();i++)
   {
    if((v[i]>>bit)&1)
    {
     swap(v[i], v[redsize]);
     check=true;
     break;
    }
   }
   if(check)
   {
    for(int i=redsize+1;i<v.size();i++)
    {
     if((v[i]>>bit)&1)
      v[i]^=v[redsize];
    }
    redsize++;
   }
  }
  v.resize(redsize);
  for(auto it:v)
   set|=it;
 }

 Gaussian& operator =(Gaussian &orig)
 {
  v = orig.v;
  set = orig.set;
  redsize = orig.redsize;
  origsize = orig.origsize;
  return *this;
 }
};
```

13 Geometry

```
struct point
{
 int x, y, idx;
};

//Finds squared euclidean distance between two
    points
int dist(point &a, point &b)
{
 return (a.x-b.x)*(a.x-b.x) + (a.y-b.y)*(a.y-b.y)
     ;
}

//Checks if angle ABC is a right angle
int isOrthogonal(point &a, point &b, point &c)
{
 return (b.x-a.x) * (b.x-c.x) + (b.y-a.y) * (b.y-
     c.y) == 0;
}

//Checks if ABCD form a rectangle (in that order)
int isRectangle(point &a, point &b, point &c,
    point &d)
{
 return isOrthogonal(a, b, c) && isOrthogonal(b,
     c, d) && isOrthogonal(c, d, a);
}

//Checks if ABCD form a rectangle, in any
    orientation
```

```
int isRectangleAnyOrder(point &a, point &b, point
    &c, point &d)
{
 return isRectangle(a, b, c, d) || isRectangle(b,
     c, a, d) | isRectangle(c, a, b, d);
}

//Checks if ABCD form a square (in that order)
int isSquare(point &a, point &b, point &c, point
    &d)
{
 return isRectangle(a, b, c, d) && dist(a, b) ==
     dist(b, c);
}

//Checks if ABCD form a square, in any
    orientation
int isSquareAnyOrder(point &a, point &b, point &c
    , point &d)
{
 return isSquare(a, b, c, d) || isSquare(b, c, a,
     d) | isSquare(c, a, b, d);
}
```

14 HLD

```
void init() {
    chainNo = 1;
    pos = 0;
    for (int i = 1; i <= n; i++) {
        chainHead[i] = -1;
        g[i].clear();
    }
}
void HLD(int k, int cost, int par) {
    if (chainHead[chainNo] == -1)
        chainHead[chainNo] = k;
    pos++;
    chainInd[k] = chainNo;
    basePos[k] = pos;
    a[pos] = cost;
    int special = -1, maxval = -1;
    for (auto it: g[k]) {
        if (it.ff == par)
            continue;
        if (subtree[it.ff] > maxval) {
            maxval = it.ss;
            special = it.ff;
        }
    }
    if (special != -1)
        HLD(special, maxval, k);
    for (auto it: g[k]) {
        if (it.ff == par || it.ff == special)
            ICPC Notebook
        continue;
        chainNo++;
        HLD(it.ff, it.ss, k);
    }
}
int queryUp(int u, int v) {
    int left = chainInd[u];
    int right = chainInd[v];
    int ans = 0;
    while (1) {
        if (u == v)
            break;
        if (left == right) {
            ans = max(ans, query(1, 1, pos,
                basePos[v] + 1, basePos[u]));
            break;
        }
        ans = max(ans, query(1, 1, pos, basePos[u
            ], basePos[chainHead[left]]));
        u = chainHead[left];
        u = parent[0][u];
        left = chainInd[u];
    }
    return ans;
}
int querypath(int u, int v) {
    int lca = LCA(u, v);
    int ans1 = queryUp(u, lca);
    int ans2 = queryUp(v, lca);
    return max(ans1, ans2);
}
void rearrange()
for (int i = 1; i <= n - 1; i++)
    if (level[edge[i].ss] < level[edge[i].ff])
        swap(edge[i].ff, edge[i].ss);
void change(int pos1, int val) {
    int node = edge[pos1].ss;
    update(1, 1, pos, basePos[node], val);
}
```

15 Hashing(Strings)

```
struct Hashs
{
 vector<int> hashs;
 vector<int> pows;
 int P;
 int MOD;
 Hashs() {}
 Hashs(string &s, int P, int MOD) : P(P), MOD(MOD
    )
 {
  int n = s.size();
  pows.resize(n+1, 0);
  hashs.resize(n+1, 0);
  pows[0] = 1;
  for(int i=n-1;i>=0;i--)
  {
   hashs[i]=(1LL * hashs[i+1] * P + s[i] - 'a' +
       1) % MOD;
   pows[n-i]=(1LL * pows[n-i-1] * P) % MOD;
```

```
 }
 pows[n] = (1LL * pows[n-1] * P)%MOD;
 }
 int get_hash(int l, int r)
 {
 int ans=hashs[l] + MOD - (1LL*hashs[r+1]*pows[r
    -l+1])%MOD;
 ans%=MOD;
 return ans;
 }
};
```

16 Intervals$_Handling$

```
map<int, int> active;

void init()
{
 active[-1] = -1;
 active[2e9] = 2e9;
 active[1] = n;
}

void add(int L, int R) //Always remove [L, R]
    before adding
{
 active[L]=R;
 ans+=R-L+1;
}

void remove(int L, int R)
{
 int removed=0;
 auto it = active.lower_bound(L);
 it--;
 if(it->second>=L)
 {
  active[L] = it->second;
  it->second = L-1;
 }
 it++;
 while(it->first <= R)
 {
  if(it->second > R)
  {
   removed+=R + 1 - it->first;
   active[R+1] = it->second;
  }
  else
   removed+= it->second - it->first + 1;
  auto it2=it;
  it++;
  active.erase(it2);
 }
 ans-=removed;
}
//submission
for(int l=1;l<=n;l++)
 {
  int r=l;
  while(r+1<=n && a[r+1]==a[l])
   r++;
  s.insert({l, r-l+1});
  rem.insert({-(r-l+1), l});
  l=r;
 }
 while(rem.size())
 {
  ans++;
  auto it=*rem.begin();
  int idx=it.second;
  rem.erase(it);
  auto it2=s.lower_bound(make_pair(idx, 0));
  auto L=it2, R=it2;
  if(L!=s.begin() && R!=s.end())
  {
   L--;
   R++;
   if(R!=s.end())
   {
    if(a[L->first]==a[R->first])
    {
     rem.erase({-L->second, L->first});
     rem.erase({-R->second, R->first});
     pair<int, int> cur={L->first, L->second + R
         ->second};
     s.erase(L);
     s.erase(R);
     s.insert(cur);
     rem.insert({-cur.second, cur.first});
    }
   }
  }
  s.erase(it2);
 }
```

17 KMP

```
String:
vector<int> prefix_function(string &s)
{
 int n = (int)s.length();
 vector<int> pi(n);
 for (int i = 1; i < n; i++)
 {
  int j = pi[i-1];
  while (j > 0 && s[i] != s[j])
   j = pi[j-1];
  if (s[i] == s[j])
   j++;
  pi[i] = j;
 }
 return pi;
}
vector<int> find_occurences(string &text, string
    &pattern)
```

```
{
 string cur=pattern + '#' + text;
 int sz1=text.size(), sz2=pattern.size();
 vector<int> v;
 vector<int> lps=prefix_function(cur);
 for(int i=sz2+1;i<=sz1+sz2;i++)
 {
  if(lps[i]==sz2)
   v.push_back(i-2*sz2);
 }
 return v;
}
Vector:
vector<int> prefix_function(vector<int> &v)
{
 int n = (int)v.size();
 vector<int> pi(n);
 for (int i = 1; i < n; i++)
 {
  int j = pi[i-1];
  while (j > 0 && v[i] != v[j])
   j = pi[j-1];
  if (v[i] == v[j])
   j++;
  pi[i] = j;
 }
 return pi;
}
vector<int> find_occurences(vector<int> &text,
    vector<int> &pattern)
{
 vector<int> v=pattern;
 v.push_back(-1);
 for(auto &it:text)
  v.push_back(it);
 int sz1=text.size(), sz2=pattern.size();
 vector<int> lps=prefix_function(v);
 vector<int> store;
 for(int i=sz2+1;i<=sz1+sz2;i++)
 {
  if(lps[i]==sz2)
   store.push_back(i-sz*2);
 }
 return v;
}
```

18 LineSweep-AreaofRectangles

```
inline bool intersect1d ( double l1, double r1,
    double l2, double r2 ) {
if ( l1 > r1 ) swap ( l1, r1 ) ;
if ( l2 > r2 ) swap ( l2, r2 ) ;
return max ( l1, l2 ) <= min ( r1, r2 ) + EPS ;
}
inline int vec ( const pt & a, const pt & b,
    const pt & c ) {
double s = ( b. x - a. x ) * ( c. y - a. y ) - (
    b. y - a. y ) * ( c. x - a. x ) ;
return abs ( s ) < EPS ? 0 : s > 0 ? + 1 : - 1 ;}
bool intersect ( const seg & a, const seg & b ) {
return intersect1d ( a. p . x , a. q . x , b. p .
    x , b. q . x )
&& intersect1d ( a. p . y , a. q . y , b. p . y ,
    b. q . y )
&& vec ( a. p , a. q , b. p ) * vec ( a. p , a. q
    , b. q ) <= 0
&& vec ( b. p , b. q , a. p ) * vec ( b. p , b. q
    , a. q ) <= 0 ;}
bool operator < ( const seg & a, const seg & b )
    {
double x = max ( min ( a. p . x , a. q . x ) ,
    min ( b. p . x , b. q . x ) ) ;
return a. get_y ( x ) < b. get_y ( x ) - EPS ;}
#define MAX 1000
struct event
{
int ind; // Index of rectangle in rects
bool type; // Type of event: 0 = Lower-left ; 1 =
     Upper-right
event() {};
event(int ind, int type) : ind(ind), type(type)
    {};
};
point rects [MAX][12]; // Each rectangle consists
     of 2 points: [0] = lower-left ; [1] = upper-
    right
bool compare_x(event a, event b) { return rects[a
    .ind][a.type].x<rects[b.ind][b.type].x; }
bool compare_y(event a, event b) { return rects[a
    .ind][a.type].y<rects[b.ind][b.type].y; }
int union_area(event events_v[],event events_h[],
    int n,int e)
{
//n is the number of rectangles, e=2*n , e is the
     number of points (each rectangle has two
    points as described in
declaration of rects)
bool in_set[MAX]={0};int area=0;
sort(events_v, events_v+e, compare_x); //Pre-sort
     of vertical edges
sort(events_h, events_h+e, compare_y); // Pre-
    sort set of horizontal edges
in_set[events_v[0].ind] = 1;
for (int i=1;i<e;++i)
{ // Vertical sweep line
event c = events_v[i];
int cnt = 0; // Counter to indicate how many
    rectangles are currently overlapping
// Delta_x: Distance between current sweep line
    and previous sweep line
int delta_x = rects[c.ind][c.type].x - rects[
    events_v[i-1].ind][events_v[i-1].type].x;
int begin_y;
if (delta_x==0){
in_set[c.ind] = (c.type==0);
continue;
}
```

```
for (int j=0;j<e;++j)
if (in_set[events_h[j].ind]==1)
//Horizontal sweep line for active rectangle
{
if (events_h[j].type==0)
//If it is a bottom edge of rectangle
{
if (cnt==0) begin_y = rects[events_h[j].ind][0].y
    ; // Block starts
++cnt;
//incrementing number of overlapping rectangles
}
else
//If it is a top edge
{
--cnt;
//the rectangle is no more overlapping, so remove
     it
if (cnt==0)
//Block endsICPC Notebook
{
int delta_y = (rects[events_h[j].ind][13].y-
    begin_y);//length of the vertical sweep line
    cut by
rectangles
area+=delta_x * delta_y;
}
}
}
in_set[c.ind] = (c.type==0);//If it is a left
    edge, the rectangle is in the active set else
     not
}
return area;
}
```

19 LineSweepClosestPair

```
#define px second
#define py first
typedef pair<long long, long long> pairll;
pairll pnts [MAX];
int compare(pairll a, pairll b)
{
return a.px<b.px;
}
double closest_pair(pairll pnts[],int n)
{
sort(pnts,pnts+n,compare);
double best=INF;
set<pairll> box;
box.insert(pnts[0]);
int left = 0;
for (int i=1;i<n;++i)
{
while (left<i && pnts[i].px-pnts[left].px > best)
box.erase(pnts[left++]);
for(typeof(box.begin()) it=box.lower_bound(
    make_pair(pnts[i].py-best, pnts[i].px-best));
    it!=box.end() &&
pnts[i].py+best>=it->py;it++)
best = min(best, sqrt(pow(pnts[i].py - it->py,
    2.0)+pow(pnts[i].px - it->px, 2.0)));
box.insert(pnts[i]);
}
return best;
}
```

20 Matching$_{(}Hopcroft$ — $Karp)_inBipartite_Graph$

```
//1 indexed Hopcroft-Karp Matching in O(E sqrtV)

struct Hopcroft_Karp
{
 static const int inf = 1e9;

 int n;
 vector<int> matchL, matchR, dist;
 vector<vector<int> > g;

 Hopcroft_Karp(int n) :
  n(n), matchL(n+1), matchR(n+1), dist(n+1), g(n
      +1) {}

 void addEdge(int u, int v)
 {
  g[u].push_back(v);
 }

 bool bfs()
 {
  queue<int> q;
  for(int u=1;u<=n;u++)
  {
   if(!matchL[u])
   {
    dist[u]=0;
    q.push(u);
   }
   else
    dist[u]=inf;
  }
  dist[0]=inf;

  while(!q.empty())
  {
   int u=q.front();
   q.pop();
   for(auto v:g[u])
   {
    if(dist[matchR[v]] == inf)
    {
     dist[matchR[v]] = dist[u] + 1;
```

```
    q.push(matchR[v]);
   }
  }
 }

 return (dist[0]!=inf);
}

bool dfs(int u)
{
 if(!u)
  return true;
 for(auto v:g[u])
 {
  if(dist[matchR[v]] == dist[u]+1 &&dfs(matchR[v
      ]))
  {
   matchL[u]=v;
   matchR[v]=u;
   return true;
  }
 }
 dist[u]=inf;
 return false;
}

int max_matching()
{
 int matching=0;
 while(bfs())
 {
  for(int u=1;u<=n;u++)
  {
   if(!matchL[u])
    if(dfs(u))
     matching++;
  }
 }
 return matching;
}
```

```
};
```

21 Matrix$_{struct}$

```
 int add(int a, int b)
{
 int res = a + b;
 if(res >= MOD)
  return res - MOD;
 return res;
}

int mult(int a, int b)
{
 long long res = a;
 res *= b;
 if(res >= MOD)
  return res % MOD;
 return res;
}

struct matrix
{
 int arr[SZ][SZ];

 void reset()
 {
  memset(arr, 0, sizeof(arr));
 }

 void makeiden()
 {
  reset();
  for(int i=0;i<SZ;i++)
  {
   arr[i][i] = 1;
  }
 }

 matrix operator + (const matrix &o) const
 {
  matrix res;
  for(int i=0;i<SZ;i++)
  {
   for(int j=0;j<SZ;j++)
   {
    res.arr[i][j] = add(arr[i][j], o.arr[i][j]);
   }
  }
  return res;
 }

 matrix operator * (const matrix &o) const
 {
  matrix res;
  for(int i=0;i<SZ;i++)
  {
   for(int j=0;j<SZ;j++)
   {
    res.arr[i][j] = 0;
    for(int k=0;k<SZ;k++)
    {
     res.arr[i][j] = add(res.arr[i][j] , mult(arr
         [i][k] , o.arr[k][j]));
    }
   }
  }
  return res;
 }
};

matrix power(matrix a, int b)
{
 matrix res;
 res.makeiden();
 while(b)
 {
  if(b & 1)
```

```
 {
  res = res * a;
 }
 a = a * a;
 b >>= 1;
}
return res;
}
```

22 MaxFlow$_{-}$ $push_Relabel_[V^3]$

```
//Push-Relabel Algorithm for Flows - Gap
    Heuristic, Complexity: O(V^3)
//To obtain the actual flow values, look at all
    edges with capacity > 0
//Zero capacity edges are residual edges

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

struct PushRelabel
{
 int n;
 vector<vector<edge> > g;
 vector<long long> excess;
 vector<int> height, active, count;
 queue<int> Q;

 PushRelabel(int n):
  n(n), g(n), excess(n), height(n), active(n),
      count(2*n) {}

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

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

void push(edge &e)
{
 int amt=(int)min(excess[e.from], (long long)e.
     cap - e.flow);
 if(height[e.from]<=height[e.to] || amt==0)
  return;
 e.flow += amt;
 g[e.to][e.index].flow -= amt;
 excess[e.to] += amt;
 excess[e.from] -= amt;
 enqueue(e.to);
}

void relabel(int v)
{
 count[height[v]]--;
 int d=2*n;
 for(auto &it:g[v])
 {
  if(it.cap-it.flow>0)
   d=min(d, height[it.to]+1);
 }
 height[v]=d;
 count[height[v]]++;
 enqueue(v);
}

void gap(int k)
{
 for(int v=0;v<n;v++)
 {
  if(height[v]<k)
   continue;
  count[height[v]]--;
  height[v]=max(height[v], n+1);
  count[height[v]]++;
  enqueue(v);
 }
}

void discharge(int v)
{
 for(int i=0; excess[v]>0 && i<g[v].size(); i++)
  push(g[v][i]);
 if(excess[v]>0)
 {
  if(count[height[v]]==1)
   gap(height[v]);
  else
   relabel(v);
 }
}

long long max_flow(int source, int dest)
{
 count[0] = n-1;
 count[n] = 1;
 height[source] = n;
 active[source] = active[dest] = 1;
 for(auto &it:g[source])
 {
```

```
  excess[source]+=it.cap;
  push(it);
 }

 while(!Q.empty())
 {
  int v=Q.front();
  Q.pop();
  active[v]=false;
  discharge(v);
 }

 long long max_flow=0;
 for(auto &e:g[source])
  max_flow+=e.flow;

 return max_flow;
}
};
```

23 Min_Cost_Max_Flow_Dijkstra

```
//Works for negative costs, but does not work for
     negative cycles
//Complexity: O(min(E^2 *V log V, E logV * flow))

struct edge
{
 int to, flow, cap, cost, rev;
};

struct MinCostMaxFlow
{
 int nodes;
 vector<int> prio, curflow, prevedge, prevnode, q
     , pot;
 vector<bool> inqueue;
 vector<vector<edge> > graph;
MinCostMaxFlow() {}

MinCostMaxFlow(int n): nodes(n), prio(n, 0),
    curflow(n, 0),
prevedge(n, 0), prevnode(n, 0), q(n, 0), pot(n,
    0), inqueue(n, 0), graph(n) {}

void addEdge(int source, int to, int capacity,
    int cost)
{
 edge a = {to, 0, capacity, cost, (int)graph[to
     ].size()};
 edge b = {source, 0, 0, -cost, (int)graph[
     source].size()};
 graph[source].push_back(a);
 graph[to].push_back(b);
}

void bellman_ford(int source, vector<int> &dist)
{
 fill(dist.begin(), dist.end(), INT_MAX);
 dist[source] = 0;
 int qt=0;
 q[qt++] = source;
 for(int qh=0;(qh-qt)%nodes!=0;qh++)
 {
  int u = q[qh%nodes];
  inqueue[u] = false;
  for(auto &e : graph[u])
  {
   if(e.flow >= e.cap)
    continue;
   int v = e.to;
   int newDist = dist[u] + e.cost;
   if(dist[v] > newDist)
   {
    dist[v] = newDist;
    if(!inqueue[v])
    {
     inqueue[v] = true;
     q[qt++ % nodes] = v;
    }
   }
  }
 }
}

pair<int, int> minCostFlow(int source, int dest,
     int maxflow)
{
 bellman_ford(source, pot);
 int flow = 0;
 int flow_cost = 0;
 while(flow < maxflow)
 {
  priority_queue<pair<int, int>, vector<pair<int
      , int> >, greater<pair<int, int> > > q;
  q.push({0, source});
  fill(prio.begin(), prio.end(), INT_MAX);
  prio[source] = 0;
  curflow[source] = INT_MAX;
  while(!q.empty())
  {
   int d = q.top().first;
   int u = q.top().second;
   q.pop();
   if(d != prio[u])
    continue;
   for(int i=0;i<graph[u].size();i++)
   {
    edge &e=graph[u][i];
    int v = e.to;
    if(e.flow >= e.cap)
     continue;
    int newPrio = prio[u] + e.cost + pot[u] -
        pot[v];
    if(prio[v] > newPrio)
    {
     prio[v] = newPrio;
     q.push({newPrio, v});
```

```
      prevnode[v] = u;
      prevedge[v] = i;
      curflow[v] = min(curflow[u], e.cap - e.flow
          );
     }
    }
   }
   if(prio[dest] == INT_MAX)
    break;
   for(int i=0;i<nodes;i++)
    pot[i]+=prio[i];
   int df = min(curflow[dest], maxflow - flow);
   flow += df;
   for(int v=dest;v!=source;v=prevnode[v])
   {
    edge &e = graph[prevnode[v]][prevedge[v]];
    e.flow += df;
    graph[v][e.rev].flow -= df;
    flow_cost += df * e.cost;
   }
  }
  return {flow, flow_cost};
 }
};
```

24 Mo's$_Algorithm$

```
const int N=2e5+5;
const int M=1e6+5;
struct data
{
 int l;
 int r;
 int idx;
 long long store_ans;
};
int n, q, blocksz=1000;
int a[N];
data queries[N];
long long freq[M];
long long ans=0;
bool comp(data &d1, data &d2)
{
 int blocka=d1.l/blocksz;
 int blockb=d2.l/blocksz;
 if(blocka<blockb)
  return true;
 else if(blocka==blockb)
  return (d1.r<d2.r)^(blocka%2);
 else
  return false;
}
bool comp2(data &d1, data &d2)
{
 return d1.idx<d2.idx;
}
void update(long long k, int sign) //Sign 1 = Add
    , -1 = Remove
{
 if(sign==1)
 {
  ans-=freq[k]*freq[k]*k;
  freq[k]++;
  ans+=freq[k]*freq[k]*k;
 }
 else
 {
  ans-=freq[k]*freq[k]*k;
  freq[k]--;
  ans+=freq[k]*freq[k]*k;
 }
}
void calcmo()
{
 int moleft=1;
 int moright=0;
 for(int i=1;i<=q;i++)
 {
  int r=queries[i].r;
  int l=queries[i].l;
  while(moright<r)
  {
   moright++;
   update(a[moright], 1);
  }
  while(moright>r)
  {
   update(a[moright], -1);
   moright--;
  }
  while(moleft<l)
  {
   update(a[moleft], -1);
   moleft++;
  }
  while(moleft>l)
  {
   moleft--;
   update(a[moleft], 1);
  }
  queries[i].store_ans=ans;
 }
}
```

25 Ordered$_Statistic_Tree_(PBDS)$

```
#include <bits/stdc++.h>
using namespace std;
#include <ext/pb_ds/assoc_container.hpp>
#include <ext/pb_ds/tree_policy.hpp>
using namespace __gnu_pbds;
#define IOS ios::sync_with_stdio(0); cin.tie(0);
    cout.tie(0);
#define endl "\n"
#define int long long
const int N=2e5+5;
```

```
#define T pair<int, int>
#define ordered_set tree<T, null_type, less<T>,
    rb_tree_tag,
    tree_order_statistics_node_update>
int getless(ordered_set &os, int R, int index)
{
 return os.order_of_key({R, index});
}
ordered_set os1;
//there are two new features  it is find_by_order
    () and order_of_key().The first returns an
    iterator to the k-th
//largest element (counting from zero),the second
      the number of items in a set that are
    strictly smaller than our item
```

26 Random$_{G}enerator$

```
mt19937 rng(chrono::steady_clock::now().
    time_since_epoch().count());
int getRand(int l, int r)
{
 uniform_int_distribution<int> uid(l, r);
 return uid(rng);
}
```

27 Sparse$_{M}atrix_{(}RMQ)$

```
int RMQ[20][N];
void precompute()
{
 for(int i=0;(1<<i)<N;i++)
 {
  for(int j=(1<<i);j<N && j<(1<<(i+1)); j++)
   floorlog[j]=i;
 }
 for(int i=n;i>=1;i--)
 {
  RMQ[0][i]=dp[i];
  int mxj=floorlog[n-i+1]; //2^j <= n-i+1
  int pw=1;
  for(int j=1;j<=mxj;j++)
  {
   RMQ[j][i]=max(RMQ[j-1][i], RMQ[j-1][i+pw]);
   pw<<=1;
  }
 }
}
int getMax(int L, int R)
{
 int k=floorlog[R-L+1]; //2^k <= R-L+1
 return max(RMQ[k][L], RMQ[k][R - (1<<k) +1]);
}
```

28 Strongly$_{C}onnected_{C}omponent$

```
vector<int> g[N], newg[N], rg[N], todo;
int comp[N], indeg[N];
bool vis[N];
vector<int> gr[N];

void dfs(int k)
{
 vis[k]=1;
 for(auto it:g[k])
 {
  if(!vis[it])
   dfs(it);
 }
 todo.push_back(k);
}

void dfs2(int k, int val)
{
 comp[k]=val;
 for(auto it:rg[k])
 {
  if(comp[it]==-1)
   dfs2(it, val);
 }
}

void sccAddEdge(int from, int to)
{
 g[from].push_back(to);
 rg[to].push_back(from);
}

void scc()
{
 for(int i=1;i<=n;i++)
  comp[i]=-1;

 for(int i=1;i<=n;i++)
 {
  if(!vis[i])
   dfs(i);
 }

 reverse(todo.begin(), todo.end());

 for(auto it:todo)
 {
  if(comp[it]==-1)
  {
   dfs2(it, ++grp);
  }
 }
}
```

29 Tree$_{C}onstruction_{w}ith_{S}pecific$ Vertices

```
int tim=0;
int parent[LG][N];
int tin[N], tout[N], level[N], vertices[N];
vector<int> g[N], tree[N];
void dfs(int k, int par, int lvl)
{
 tin[k]=++tim;
 parent[0][k]=par;
 level[k]=lvl;
 for(auto it:g[k])
 {
  if(it==par)
   continue;
  dfs(it, k, lvl+1);
 }
 tout[k]=tim;
}
void precompute()
{
 for(int i=1;i<LG;i++)
  for(int j=1;j<=n;j++)
   if(parent[i-1][j])
    parent[i][j]=parent[i-1][parent[i-1][j]];
}
int LCA(int u, int v)
{
 if(level[u]<level[v])
  swap(u,v);
 int diff=level[u]-level[v];
 for(int i=LG-1;i>=0;i--)
 {
  if((1<<i) & diff)
  {
   u=parent[i][u];
  }
 }
 if(u==v)
  return u;
 for(int i=LG-1;i>=0;i--)
 {
  if(parent[i][u] && parent[i][u]!=parent[i][v])
  {
   u=parent[i][u];
   v=parent[i][v];
  }
 }
 return parent[0][u];
}
bool isancestor(int u, int v) //Check if u is an
    ancestor of v
{
 return (tin[u]<=tin[v]) && (tout[v]<=tout[u]);
}
int work()
{
 sort(vertices+1, vertices+k+1, [](int a, int b)
 {
  return tin[a]<tin[b];
 });
 int idx=k;
 for(int i=1;i<idx;i++)
  vertices[++k]=LCA(vertices[i], vertices[i+1]);
 sort(vertices+1, vertices+k+1);
 k=unique(vertices+1, vertices+k+1) - vertices -
     1;
 sort(vertices+1, vertices+k+1, [](int a, int b)
 {
  return tin[a]<tin[b];
 });
 stack<int> s;
 s.push(vertices[1]);
 for(int i=2;i<=k;i++)
 {
  while(!isancestor(s.top(), vertices[i]))
   s.pop();
  tree[s.top()].push_back(vertices[i]);
  s.push(vertices[i]);
 }
 for(int i=1;i<=k;i++)
  tree[vertices[i]].clear();
}
```

30 Z$_{A}lgorithm$

```
//The Z-function for this string is an array of
    length n where the i-th element is equal to
    the greatest number
//of characters starting from the position i that
     coincide with the first characters of s.
vector<int> z_function(string &s)
{
 int n=s.size();
 vector<int> z(n);
 for(int i=1,l=0,r=0;i<n;i++)
 {
  if(i<=r)
   z[i]=min(r-i+1, z[i-l]);
  while(i+z[i]<n && s[z[i]]==s[i+z[i]])
   z[i]++;
  if(i+z[i]-1>r)
   l=i, r=i+z[i]-1;
 }
 return z;
}
```

31 blockcuttree

```
struct graph
{
  int n;
  vector<vector<int>> adj;
  graph(int n) : n(n), adj(n) {}
  void add_edge(int u, int v)
  {
```

```
    adj[u].push_back(v);
    adj[v].push_back(u);
  }
  int add_node()
  {
    adj.push_back({});
    return n++;
  }
  vector<int>& operator[](int u) { return adj[u];
      }
};
vector<vector<int>> biconnected_components(graph
    &adj)
{
  int n = adj.n;
  vector<int> num(n), low(n), art(n), stk;
  vector<vector<int>> comps;
  function<void(int, int, int&)> dfs = [&](int u,
      int p, int &t)
  {
    num[u] = low[u] = ++t;
    stk.push_back(u);
    for (int v : adj[u]) if (v != p)
    {
      if (!num[v])
      {
        dfs(v, u, t);
        low[u] = min(low[u], low[v]);
        if (low[v] >= num[u])
        {
          art[u] = (num[u] > 1 || num[v] > 2);

          comps.push_back({u});
          while (comps.back().back() != v)
            comps.back().push_back(stk.back()),
                stk.pop_back();
        }
      }
      else low[u] = min(low[u], num[v]);
    }
  };

  for (int u = 0, t; u < n; ++u)
    if (!num[u]) dfs(u, -1, t = 0);
  // build the block cut tree
  function<graph()> build_tree = [&]()
  {
    graph tree(0);
    vector<int> id(n);
    for (int u = 0; u < n; ++u)
      if (art[u]) id[u] = tree.add_node();

    for (auto &comp : comps)
    {
      int node = tree.add_node();
      for (int u : comp)
        if (!art[u]) id[u] = node;
        else tree.add_edge(node, id[u]);
    }
    return tree;
  };
  return comps;
}
```

32 boruvkamst

```
int V = graph - > V, E = graph - > E;
Edge * edge = graph - > edge;
struct subset * subsets = new subset[V];
int * cheapest = new int[V];
for (int v = 0; v < V; ++v) {
    subsets[v].parent = v;
    subsets[v].rank = 0;
    cheapest[v] = -1;
}
int numTrees = V, MSTweight = 0;
while (numTrees > 1) {
    for (int i = 0; i < E; i++) {
        int set1 = find(subsets, edge[i].src);
        int set2 = find(subsets, edge[i].dest);
        if (set1 == set2)
            continue;
        else {
            if (cheapest[set1] == -1 ||
                edge[cheapest[set1]].weight > edge
                    [i].weight)
                cheapest[set1] = i;
            if (cheapest[set1] == -1 ||
                edge[cheapest[set2]].weight > edge
                    [i].weight)
                cheapest[set2] = i;
        }
    }
    for (int i = 0; i < V; i++) {
        if (cheapest[i] != -1) {
            int set1 = find(subsets, edge[cheapest
                [i]].src);
            int set2 = find(subsets, edge[cheapest
                [i]].dest);
            if (set1 == set2)
                continue;
            MSTweight += edge[cheapest[i]].weight;
            Union(subsets, set1, set2);
            numTrees--;
        }
    }
}
```

33 dpoptimization

```
// Divide and conquer optimization:
// Original Recurrence
// dp[i][j] = min(dp[i-1][k] + C[k][j]) for k < j
// Sufficient condition:
// A[i][j] <= A[i][j+1]
```

```
// where A[i][j] = smallest k that gives optimal
    answer
// How to use:
// // compute i-th row of dp from L to R. optL <=
     A[i][L] <= A[i][R] <= optR
// compute(i, L, R, optL, optR)
//
1. special
case L == R
//
2.
let M = (L + R) / 2. Calculate dp[i][M] and opt[i
    ][M] using O(optR - optL + 1)
//
3. compute(i, L, M - 1, optL, opt[i][M])
//
4. compute(i, M + 1, R, opt[i][M], optR) ICPC
    Notebook
void compute(int i, int L, int R, int optL, int
    optR) {
    if (L > R) return;
    int mid = (L + R) >> 1, savek = optL;
    dp[i][mid] = inf;
    FOR(k, optL, min(mid - 1, optR)) {
        int cur = dp[i - 1][k] + getCost(k + 1,
            mid);
        if (cur < dp[i][mid]) {
            dp[i][mid] = cur;
            savek = k;
        }
    }
    compute(i, L, mid - 1, optL, savek);
    compute(i, mid + 1, R, savek, optR);
}
Knuth Optimisation
// Original Recurrence:
// dp[i][j] = min(dp[i][k] + dp[k][j]) + C[i][j]
    for k = i+1..j-1
// Necessary & Sufficient Conditions:
// A[i][j-1] <= A[i][j] <= A[i+1][j]
// with A[i][j] = smallest k that gives optimal
    answer
// Also applicable if the following conditions
    are met:
// 1. C[a][c] + C[b][d] <= C[a][d] + C[b][c] (
    quadrangle inequality)
// 2. C[b][c] <= C[a][d]
(monotonicity)
// for all a <= b <= c <= d
// To use:
// Calculate dp[i][i] and A[i][i]
//
// FOR(len = 1..n-1)
// FOR(i = 1..n-len) {
//
j = i + len
//
FOR(k = A[i][j - 1]..A[i + 1][j])
//
update(dp[i][j])
// }
const int MN = 2011;
int a[MN], dp[MN][MN], C[MN][MN], A[MN][MN];
int n;
void solve() {
    cin >> n;
    FOR(i, 1, n) {
        cin >> a[i];
        a[i] += a[i - 1];
    }
    FOR(i, 1, n) FOR(j, i, n) C[i][j] = a[j] - a[
        i - 1];
    FOR(i, 1, n) dp[i][i] = 0, A[i][i] = i;
    FOR(len, 1, n - 1)
    FOR(i, 1, n - len) {
        int j = i + len;
        ICPC Notebook
        dp[i][j] = 2000111000;
        FOR(k, A[i][j - 1], A[i + 1][j]) {
            int cur = dp[i][k - 1] + dp[k][j] + C[
                i][j];
            if (cur < dp[i][j]) {
                dp[i][j] = cur;
                A[i][j] = k;
            }
        }
    }
    cout << dp[1][n] << endl;
}
//SOS DP - Initialize base case of nums[i] - Sum
    of all subsets
rep(i, 0, 17) rep(j, 0, N) if ((j >> i) & 1) nums
    [j] += nums[j ^ (1 << i)];

void neg(int f[], int upto) {
    for (int i = 0; i <= upto; i++)
        if (pc[i] % 2 == 0) f[i] = -f[i];
}
void sos(int f[], int upto, int t) {
    for (int i = 0; i < t; ++i)
        for (int mask = 0; mask <= upto; ++mask)
            {
            if (mask & (1 << i))
                f[mask] += f[mask ^ (1 << i)];
        }
}
neg(cnt, upto);
sos(cnt, upto, t);
```

34 euler

```
//Start with an empty stack and an empty circuit
    (eulerian path).
//- If all vertices have even degree - choose any
     of them.
//- If there are exactly 2 vertices having an odd
     degree - choose one of them.
```

```
//- Otherwise no euler circuit or path exists.
//If current vertex has no neighbors - add it to
    circuit, remove the last vertex from the
    stack and set it as the current one.
//Otherwise (in case it has neighbors) - add the
    vertex to the stack, take any of its
    neighbors, remove the edge between
//selected neighbor and that vertex, and set that
     neighbor as the current vertex.
//Repeat step 2 until the current vertex has no
    more neighbors and the stack is empty.
//Note that obtained circuit will be in reverse
    order - from end vertex to start vertex.
//Code stores the Euler Circuit path in Circuit
void EulerTour(int k) //Heirholzer's Algorithm
{
    int cur = k;
    stack < int > temp;
    while (true) {
        int ct = g[cur].size();
        if (!ct) {
            viscircuit[cur] = 1;
            circuit.push_back(cur);
            if (temp.size() == 0)
                break;
            cur = temp.top();
            temp.pop();
        } else {
            pair < int, int > next = * (g[cur].
                begin());
            g[cur].erase(next);
            g[next.ff].erase(mp(cur, next.ss));
            if (next.ss == 1) { in [next.ff]++;
                out[cur]++;
                ans.pb(mp(cur, next.ff));
            }
            temp.push(cur);
            cur = next.ff;
        }
    }
}
```

35 implicitsegtree

```
//Implicit Segment Tree :
//For assigning a unique number to every visitor,
     such that the unique number >=x, and has not
     been taken. Input:
//1 x = Tourist enters, 2 x = Tourist leaves.
struct nd {
    ll t, x;
};
struct segmenttree {
    int val;
    segmenttree * left, * right;
    segmenttree() {
        val = 0;
        left = NULL;
        right = NULL;
    }
};
int n, num = 1e6;
nd queries[N];
void update(segmenttree * root, int L, int R, int
     pos, int type) {
    if (L == R) {
        if (type == 0) {
            root - > val = 0;
        }
        elseICPC Notebook {
            root - > val = 1;
        }
        return;
    }
    int M = (L + R) >> 1;
    if (pos <= M)
        update(root - > left, L, M, pos, type);
    else
        update(root - > right, M + 1, R, pos,
            type);
    root - > val = (root - > left) - > val * (
        root - > right) - > val;
}
int query(segmenttree * root, int L, int R, int
    pos) {
    if (L == R) {
        if (root - > val == 0)
            return L;
        else
            return -1;
    }
    int M = (L + R) >> 1;
    if (root - > left == NULL) {
        root - > left = new segmenttree();
        root - > right = new segmenttree();
    }
    if (pos <= M) {
        if ((root - > left) - > val == 1) {
            return query(root - > right, M + 1, R,
                pos);
        } else {
            int store = query(root - > left, L, M,
                pos);
            if (store == -1) {
                return query(root - > right, M +
                    1, R, pos);
            } else {
                return store;
            }
        }
    } else {
        return query(root - > right, M + 1, R,
            pos);
    }
}
```

36 manacher

```
void Manachers() {
    int N = strlen(text);
    N = 2 * N + 1; //Position count
    int L[N]; //LPS Length Array
    L[0] = 0;
    L[1] = 1;
    int C = 1, R = 2, i = 0;
    int iMirror, maxLPSLength = 0,
        maxLPSCenterPosition = 0;
    int start = -1, end = -1, diff = -1;
    for (i = 2; i < N; i++) {
        iMirror = 2 * C - i;
        L[i] = 0;
        diff = R - i;
        if (diff > 0)
            L[i] = min(L[iMirror], diff);
        while (((i + L[i]) < N && (i - L[i]) > 0)
             &&
            (((i + L[i] + 1) % 2 == 0) ||
                (text[(i + L[i] + 1) / 2] == text
                    [(i - L[i] - 1) / 2])))
            L[i]++;
        if (L[i] > maxLPSLength) {
            maxLPSLength = L[i];
            maxLPSCenterPosition = i;
        }
        if (i + L[i] > R) {
            C = i;
            R = i + L[i];
        }
    }
    start = (maxLPSCenterPosition - maxLPSLength)
         / 2;
    ICPC Notebook
    end = start + maxLPSLength - 1;
    printf("LPS of string is %s : ", text);
    for (i = start; i <= end; i++)
        printf("%c", text[i]);
    printf("\n");
}
```

37 maths

```
   //1) Sum of values of totient functions of all
      divisors of n is equal to n.
//4)    Bell Number:
//In combinatorial mathematics, the Bell numbers
    count the possible partitions of a set. The
    nth of these numbers,
//Bn,counts the number of different ways to
    partition a set that has exactly n elements,
    or equivalently, the number of
//equivalence relations on it.
void processbell()
{
    bell[0]=1;
    bell[1]=1;
    for(int i=2;i<=5000;i++)
    {
        for(int j=0;j<=i-1;j++)
        {
            bell[i]+=nCr(i-1, j) * bell[j];
            bell[i]%=MOD;
        }
    }
}
//5)   Stirling number of the second kind:
//In mathematics, particularly in combinatorics,
    a Stirling number of the second kind (or
    Stirling partition number) is the
//number of ways to partition a set of n objects
    into k non-empty subsets and is denoted by S(
    n,k).
//Equivalently, they count the number of
    different equivalence relations with
    precisely k equivalence classes that can be
//defined on an n element set.
//Value of S(n, k) can be defined recursively as,
     S(n+1, k) = k*S(n, k) + S(n, k-1)
//with S(0, 0) = 1 and S(n, 0) = S(0, n) = 0 for
    n>0


//8)    Modular Inverse modulo N (General) :
int modInverse(int a, int m)
{
    int m0 = m;
    int y = 0, x = 1;
    if (m == 1)
      return 0;
    while (a > 1)
    {
        int q = a / m;
        int t = m;
        m = a % m, a = t;
        t = y;
        y = x - q * y;
        x = t;
    }
    if (x < 0)
       x += m0;
    return x;
}
9    Chinese Remainder Theorem :
int findMinX(int num[], int rem[], int k)
{
 int prod = 1;
 for (int i = 0; i < k; i++)
  prod *= num[i];
 int result = 0;
 for (int i = 0; i < k; i++)
 {
  int pp = prod / num[i];ICPC Notebook
  result += rem[i] * inv(pp, num[i]) * pp;
 }
 return result % prod;
```

```
}
//nCr%m when n,r ~ 10^18, m ~10^6
#define N 1000005
lld maxp[N];
lld extended_euclid(lld a,lld b,lld &x,lld &y) {
    lld xx = y = 0;
    lld 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;
}
lld mod(lld a,lld b) {
    return ((a%b)+b)%b;
}
lld inversemod(lld a,lld n) {
    lld x, y;
    lld d = extended_euclid(a, n, x, y);
    if(d > 1) return -1LL;
    return mod(x,n);
}
plld chinese_remainder_theorem(lld x,lld a,lld y,
    lld b) {
  lld s, t;
  lld d = extended_euclid(x, y, s, t);
  if(a%d != b%d) return make_pair(0LL, -1LL);
  return make_pair(mod(s*b*x+t*a*y,x*y)/d, x*y/d)
      ;
}
plld chinese_remainder_theorem(const vlld &x,
    const vlld &a) { // x are the modules and a
    are remainders
    plld ret = make_pair(a[0], x[0]);
    lld tmp=x.size();
    rep(i,0,tmp) {
        ret = chinese_remainder_theorem(ret.
            second, ret.first, x[i], a[i]);
        if (ret.second == -1) break;
    }
    return ret;
}
lld countFact(lld n,lld p)
{
    lld k=0;
    while(n>=p) k+=n/p,n/=p;
    return k;
}
lld factorial_mod(lld n,lld m)
{
    lld res=1;
    while(n>0)
    {
        for(lld i=2, m=n%MOD; i<=m; i++)
            res=(res*i) % MOD;
        if ((n/=MOD)%2 > 0)
            res = MOD - res;
    }
    return res;
}
lld nCk_get_prime_pow(lld n,lld k,lld p) {
    lld res=countFact(n,p)-countFact(k,p)-
        countFact(n-k,p);
    return res;
}
lld nCk_get_non_prime_part(lld n,lld k,lld p,lld
    e) {
    lld pe=powm(p,e,LLINF);
    lld r=n-k,acc=1;
    vlld fact_pe;
    fact_pe.pb(1LL);
    rep(x,1,pe) {
        if(x%p == 0) fact_pe.pb(acc);
        else acc=(acc*x)%pe,fact_pe.pb(acc);
    }
    lld top=1,bottom=1,is_neg=0,digits=0;
    while(n!=0) {
        if(acc!=1) {
            if(digits>=e) {
                is_neg ^= (n&1);
                is_neg ^= (r&1);
                is_neg ^= (k&1);
            }
        }
        top=(top*fact_pe[n%pe])%pe;
        bottom=(bottom*fact_pe[r%pe])%pe;
        bottom=(bottom*fact_pe[k%pe])%pe;
        n/=p,r/=p,k/=p;
        digits+=1;
    }
    lld res=(top*inversemod(bottom,pe))%pe;
    if(p!=2 or e<3)
        if(is_neg) res=pe-res;

    return res;
}

//Sum of GP in LogN
lld solve(lld x,lld n,lld m){
  if(n==0) return 1LL;
  if(n==1) return (1LL+x)%m;
  if(n%2==0){
    lld t1=solve((x*x)%m,n/2LL-1LL,m);
    t1=(t1*(1LL+x))%m;
    t1=(t1+power(x,n,m))%m;
    return t1;
  }
  else{
    lld t1=solve((x*x)%m,n/2LL,m);
    t1=(t1*(1LL+x))%m;
    return t1;
  }
}
```

38 persistent

```
int build(int L, int R) {
    int node = ++ct;
    if (L == R) {
        return node;
    }
    int M = (L + R) >> 1;
    lc[node] = build(L, M);
    rc[node] = build(M + 1, R);
    return node;
}
int update(int onode, int L, int R, int pos) {
    int node = ++ct;
    if (L == R) {
        st[node] = st[onode] + 1;
        return node;
    }
    int M = (L + R) >> 1;
    lc[node] = lc[onode];
    rc[node] = rc[onode];
    if (pos <= M)
        lc[node] = update(lc[onode], L, M, pos);
    else
        rc[node] = update(rc[onode], M + 1, R,
            pos);
    st[node] = st[lc[node]] + st[rc[node]];
    return node;
}
int query(int nodeu, int nodev, int L, int R, int
    pos) {
    if (L == R) {
        return L;
    }
    int M = (L + R) >> 1;
    int leftval = st[lc[nodev]] - st[lc[nodeu]];
    int rightval = st[rc[nodev]] - st[rc[nodeu]];
    if (leftval >= pos) {
        return query(lc[nodeu], lc[nodev], L, M,
            pos);
    } else {
        return query(rc[nodeu], rc[nodev], M + 1,
            R, pos - leftval);
    }
}
```

39 suffixarray

```
int suffixRank[20][int(1E6)];
struct myTuple {
    int originalIndex; // stores original index
        of suffix
    int firstHalf; // store rank for first half
        of suffix
    int secondHalf;
    // store rank for second half of suffix
};
int cmp(myTuple a, myTuple b) {
    if (a.firstHalf == b.firstHalf) return a.
        secondHalf < b.secondHalf;
    else return a.firstHalf < b.firstHalf;
}
int N = s.size();
for (int i = 0; i < N; ++i)
    suffixRank[0][i] = s[i] - 'a';
myTuple L[N];
for (int cnt = 1, stp = 1; cnt < N; cnt *= 2, ++
    stp) {
    for (int i = 0; i < N; ++i) {
        L[i].firstHalf = suffixRank[stp - 1][i];
        L[i].secondHalf = i + cnt < N ?
            suffixRank[stp - 1][i + cnt] : -1;
        L[i].originalIndex = i;
    }
    sort(L, L + N, cmp);
    suffixRank[stp][L[0].originalIndex] = 0;
    for (int i = 1, currRank = 0; i < N; ++i) {
        if (L[i - 1].firstHalf != L[i].firstHalf
            || L[i - 1].secondHalf != L[i].
            secondHalf)
            ++currRank;
        suffixRank[stp][L[i].originalIndex] =
            currRank;
    }
}
//KASAI
vector < int > kasai(string txt, vector < int >
    suffixArr) {
        int n = suffixArr.size();
        vector < int > lcp(n, 0);
        vector < int > invSuff(n, 0);
        for (int i = 0; i < n; i++)
            invSuff[suffixArr[i]] = i;
        int k = 0;
        for (int i = 0; i < n; i++) {
            if (invSuff[i] == n - 1) {
                k = 0;
                continue;
            }
            int j = suffixArr[invSuff[i] + 1];
            while (i + k < n && j + k < n && txt[i
                 + k] == txt[j + k])
                k++;
            lcp[invSuff[i]] = k; // lcp for the
                present suffix.
            if (k > 0)
                k--;
        }
        return lcp;
```