**Daftar Isi**

[**1.** **LCA Binary Lifting** 2](#_Toc529547642)

[**2.** **Knuth-Morris-pratt (Precompute & Checking)** 2](#_Toc529547643)

[**3.** **Const Big Prime Number** 3](#_Toc529547644)

**[4.](#_Toc529547645)****[Miller Rabin Big Primality test](#_Toc529547645)** [3](#_Toc529547645)

[**5.** **Extended Euclidean Algorithm** 3](#_Toc529547646)

[**6.** **FFT biasa & FFT versi modular arithmetic (perkalian polinom)** 3](#_Toc529547647)

[**7.** Gaussian Elimination 5](#_Toc529547648)

[**8.** **Maxflow Dinic** 5](#_Toc529547649)

[**9.** **Minimum Cost Max Flow** 6](#_Toc529547650)

[**10.** **MinCost MaxFlow with Potential** 7](#_Toc529547651)

[**11.** **Mincost MaxFlow with Negative Cost** 8](#_Toc529547652)

[**12.** **Maximum Cardinality Bipartite Matching** 9](#_Toc529547653)

[**13.** **Finding Cut Vertices & Cut Edges** 10](#_Toc529547654)

[**14.** **Rumus-rumus kombin** 10](#_Toc529547655)

[**15.** **Template Geometri** 12](#_Toc529547656)

[**16.** **Find two center of same size circle from its intersection and their radius** 13](#_Toc529547657)

[**17.** **The Great-Circle Distance (SPHERES)** 13](#_Toc529547658)

[**18.** **Cutting Polygon with a Straight Line** 13](#_Toc529547659)

[**19.** **Convex hull (Graham’s Scan & Andrew’s Monotone Chain)** 13](#_Toc529547660)

[**20.** **Pick’s Theorem** 15](#_Toc529547661)

[**21.** **Strongly Connected Component** 16](#_Toc529547662)

[**22.** **Suffix Array + LCP** 16](#_Toc529547663)

[**23.** **Manacher Algorithm (Palindrom)** 16](#_Toc529547664)

[**24.** **Implicit Treap** 17](#_Toc529547665)

[**25.** **Convex Hull Trick** 18](#_Toc529547666)

[**26.** **Z Algorithm** 19](#_Toc529547667)

[**27.** **Aho Corassick** 19](#_Toc529547668)

[**28.** Blossom 21](#_Toc529547669)

[**29.** Minimum Cut Stoer - Wagner 22](#_Toc529547670)

[**30.** Chinese Remainder Theorem 23](#_Toc529547671)

[**31.** Simplex 23](#_Toc529547672)

[**32.** Persistent Segment tree 25](#_Toc529547673)

# **LCA Binary Lifting**

|  |
| --- |
| int n, l;  vector<vector<int>> adj;  int timer;  vector<int> tin, tout;  vector<vector<int>> up;  void dfs(int v, int p)  {  tin[v] = ++timer;  up[v][0] = p;  for (int i = 1; i <= l; ++i)  up[v][i] = up[up[v][i-1]][i-1];  for (int u : adj[v]) {  if (u != p)  dfs(u, v);  }  tout[v] = ++timer;  }  bool is\_ancetor(int u, int v)  {  return tin[u] <= tin[v] && tout[u] >= tout[v];  }  int lca(int u, int v)  {  if (is\_ancetor(u, v))  return u;  if (is\_ancetor(v, u))  return v;  for (int i = l; i >= 0; --i) {  if (!is\_ancetor(up[u][i], v))  u = up[u][i];  }  return up[u][0];  }  void preprocess(int root) {  tin.resize(n);  tout.resize(n);  timer = 0;  l = ceil(log2(n));  up.assign(n, vector<int>(l + 1));  dfs(root, root);  } |

# **Knuth-Morris-pratt (Precompute & Checking)**

Precompute :

|  |
| --- |
| Arrays.fill(a, 0);  for(int i = 1; i < n; i++) {  int j = a[i - 1];  while(j > 0 && s[i] != s[j]) j = a[j - 1];  if(s[i] == s.[j]) a[i] = j + 1;  } |

Checking :

|  |
| --- |
| int[] b = computeKMP(pattern);  int j = 0;  for(int i = 0; i < text.length();) {  if(pattern.charAt(j) == text.charAt(i)) {  i++; j++;  } else if(j > 0) {  j = b[j - 1];  } else {  i++;  }  if(j == pattern.length()) {  return i - pattern.length();  }  }  return NOT\_FOUND; |

# **Const Big Prime Number**

|  |
| --- |
| 1e9 + 9, 1e9 + 87, 1e9 + 4207, 2e9 + 89, 2e9 + 143, 2e9 + 11, 2e9 + 1851, 2e9 + 2153,  252097800623, 1e15 - 11, 1e15 + 37, |

# **Miller Rabin Big Primality test**

|  |
| --- |
| vector<long long> A({2, 3, 5, 7, 11, 13, 17, 19, 23});  // if n < 3,825,123,056,546,413,051, it is enough to test a = 2, 3, 5, 7, 11, 13, 17, 19, and 23.  long long fastmul(long long a, long long b, long long n) {  long long ret = 0;  while (b) {  if (b & 1)  ret = (ret + a) % n;  a = (a + a) % n;  b >>= 1;  }  return ret;  }  long long fastexp(long long a, long long b, long long n) {//compute (a^b) mod n  long long ret = 1;  while (b) {  if (b & 1)  ret = fastmul(ret, a, n);  a = fastmul(a, a, n);  b >>= 1;  }  return ret;  }  bool mrtest(long long n)  {  if(n == 1) return false;  long long d = n-1;  long long s = 0;  while(d % 2 == 0)  {  s++;  d /= 2;  }  for(long long j=0;j<(long long)A.size();j++)  {  if(A[j] > n-1) continue;  long long ad = fastexp(A[j], d, n);  if(ad % n == 1) continue;  bool notcomp = false;  for(long long r=0;r<=max(0LL,s-1);r++)  {  long long rr = fastexp(2,r,n);  long long ard = fastexp(ad, rr, n);  if(ard % n == n-1) {notcomp = true; break;}  }  if(!notcomp)  {  return false;  }  }  return true;  } |

# **Extended Euclidean Algorithm**

|  |
| --- |
| long long x, y, d; // ax + by = d  void extendedEuclidean(long long a, long long b) {  if(b == 0) { x = 1; y = 0; d = a; return; }  extendedEuclidean(b, a % b);  long long xx, yy;  xx = y;  yy = x - (a/b)\*y;  x = xx; y = yy;  } |

# **FFT biasa & FFT versi modular arithmetic (perkalian polinom)**

|  |
| --- |
| /\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\* FFT dengan complex \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/  typedef complex<double> cd;  typedef vector< cd > vcd;  // asumsi ukuran as = 2^k, dengan k bilangan bulat positif  vcd fft(const vcd &as) {  int n = (int)as.size();  int k = 0;  while((1<<k) < n) k++;  vector< int > r(n);  r[0] = 0;  int h = -1;  for(int i = 1; i<n; i++) {  if((i & (i-1)) == 0)  h++;  r[i] = r[i ^ (1 << h)];  r[i] |= (1<<(k-h-1));  }  vcd root(n);  for(int i = 0; i<n; i++) {  double ang = 2.0\*M\_PI\*i/n;  root[i] = cd(cos(ang), sin(ang));  }  vcd cur(n);  for(int i = 0; i<n; i++)  cur[i] = as[r[i]];  for(int len = 1; len < n; len <<= 1 ) {  vcd ncur(n);  int step = n/(len << 1);  for(int pdest = 0; pdest <n;) {  for(int i = 0; i<len; i++) {  cd val = root[i\*step]\*cur[pdest + len];  ncur[pdest] = cur[pdest] + val;  ncur[pdest + len] = cur[pdest] - val;  pdest++;  }  pdest += len;  }  cur.swap(ncur);  }  return cur;  }  vcd inv\_fft(const vcd& fa) {  vcd res = fft(fa);  for(int i = 0; i<nn; i++) {  res[i] /= nn;  }  reverse(res.begin() + 1, res.end());  return res;  }  /\*\*\*\*\*\*\*\*\*\*\*\*\*\*\* FFT dengan Modular Aritmetic \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/  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);  }  } |

# Gaussian Elimination

|  |
| --- |
| vector<double> gauss(vector< vector<double> >& A) {  int n = A.size();  for (int i=0; i<n; i++) {  double maxEl = abs(A[i][i]);  double maxRow = i;  for (int k=i+1; k<n; k++) {  if (abs(A[k][i]) > maxEl) {  maxEl = abs(A[k][i]);  maxRow = k;  }  }  for (int k=i; k<n+1;k++) {  double tmp = A[maxRow][k];  A[maxRow][k] = A[i][k];  A[i][k] = tmp;  }  for (int k=i+1; k<n; k++) {  double c = -A[k][i]/A[i][i];  for (int j=i; j<n+1; j++) {  if (i==j) {  A[k][j] = 0;  } else {  A[k][j] += c \* A[i][j];  }  }  }  }  vector<double> x(n);  for (int i=n-1; i>=0; i--) {  x[i] = A[i][n]/A[i][i];  for (int k=i-1;k>=0; k--) {  A[k][n] -= A[k][i] \* x[i];  }  }  return x;  } |

# **Maxflow Dinic**

|  |
| --- |
| struct Edge {  int from, to, cap, flow, index;  Edge(int from, int to, int cap, int flow, int index) :  from(from), to(to), cap(cap), flow(flow), index(index) {}  };  struct Dinic {  int N;  vector<vector<Edge> > G;  vector<Edge \*> dad;  vector<int> Q;    Dinic(int N) : N(N), G(N), dad(N), Q(N) {}    void AddEdge(int from, int to, int cap) {  G[from].push\_back(Edge(from, to, cap, 0, G[to].size()));  if (from == to) G[from].back().index++;  G[to].push\_back(Edge(to, from, 0, 0, G[from].size() - 1));  }  long long BlockingFlow(int s, int t) {  fill(dad.begin(), dad.end(), (Edge \*) NULL);  dad[s] = &G[0][0] - 1;    int head = 0, tail = 0;  Q[tail++] = s;  while (head < tail) {  int x = Q[head++];  for (int i = 0; i < G[x].size(); i++) {  Edge &e = G[x][i];  if (!dad[e.to] && e.cap - e.flow > 0) {  dad[e.to] = &G[x][i];  Q[tail++] = e.to;  }  }  }  if (!dad[t]) return 0;  long long totflow = 0;  for (int i = 0; i < G[t].size(); i++) {  Edge \*start = &G[G[t][i].to][G[t][i].index];  int amt = INF;  for (Edge \*e = start; amt && e != dad[s]; e = dad[e->from]) {  if (!e) { amt = 0; break; }  amt = min(amt, e->cap - e->flow);  }  if (amt == 0) continue;  for (Edge \*e = start; amt && e != dad[s]; e = dad[e->from]) {  e->flow += amt;  G[e->to][e->index].flow -= amt;  }  totflow += amt;  }  return totflow;  }  long long GetMaxFlow(int s, int t) {  long long totflow = 0;  while (long long flow = BlockingFlow(s, t))  totflow += flow;  return totflow;  }  }; |

# **Minimum Cost Max Flow**

|  |
| --- |
| const int inf = 1e8;  struct Edge {  int from, to, cap, flow, cost;  Edge(int from, int to, int cap, int flow, int cost) :  from(from), to(to), cap(cap), flow(flow), cost(cost) {}  };  struct MCMF {  int n, s, t;  vector< vector< int > > adj;  vector< int > par;  vector< Edge > vEdge;  vector< long long > dist;  MCMF(int \_n, int \_s, int \_t) : n(\_n), adj(n), s(\_s), t(\_t) {  }  void addEdge(int from, int to, int cap, int cost) {  adj[from].push\_back(vEdge.size());  adj[to].push\_back(vEdge.size());  vEdge.push\_back(Edge(from, to, cap, 0, cost));  }  long long augment(int v, int minflow = inf) {  if(v == s) {  return minflow;  }  if(par[v] < 0) {  return 0;  }  long long flow;  Edge &e = vEdge[par[v]];  if(v == e.from) {  flow = augment(e.to, min(minflow, e.flow));  e.flow -= flow;  }  else {  flow = augment(e.from, min(minflow, e.cap - e.flow));  e.flow += flow;  }  return flow;  }  long long findingPath() {  //dijkstra  set< pair< long long , int > > st;  dist.assign(n, inf);  par.assign(n, -1);  dist[s] = 0;  st.insert(make\_pair(dist[s], s));  while(!st.empty()) {  set< pair< long long, int > >::iterator begin = st.begin();  int v = begin->second;  st.erase(begin);  for(int i = 0; i<adj[v].size(); i++) {  Edge &e = vEdge[adj[v][i]];  if(e.from == v) {  if(e.cap > e.flow && dist[e.to] > dist[v] + e.cost) {  st.erase(make\_pair(dist[e.to], e.to));  dist[e.to] = dist[v] + e.cost;  st.insert(make\_pair(dist[e.to], e.to));  par[e.to] = adj[v][i];  }  }  else {  if(e.flow > 0 && dist[e.from] > dist[v] - e.cost) {  st.erase(make\_pair(dist[e.from], e.from));  dist[e.from] = dist[v] - e.cost;  st.insert(make\_pair(dist[e.from], e.from));  par[e.from] = adj[v][i];  }  }  }  }  return augment(t, inf);  }  pair< long long, long long > EdmondKarp() {  long long maxflow = 0, mincost = 0;  long long flow;  while(flow = findingPath()) {  maxflow += flow;  mincost += flow \* dist[t];  }  return make\_pair(mincost, maxflow);  }  }; |

# **MinCost MaxFlow with Potential**

|  |
| --- |
| #include <bits/stdc++.h>  using namespace std;  #define INF 1000000000  #define MAXN 500  struct Edge  {  int t, c, w, r;  Edge(int \_t, int \_c, int \_w, int \_r): t(\_t), c(\_c), w(\_w), r(\_r) {};  };  int pot[MAXN+5], prv[MAXN+5], dist[MAXN+5], vis[MAXN+5];  vector<Edge> edge[MAXN+5];  pair<int, int> mcmf(int n, int s, int t)  {  fill(pot, pot+n, 0);  int mf = 0, mc = 0;  while (true) {  priority\_queue<pair<int, int> > pq;  fill(dist, dist+n, INF);  fill(vis, vis+n, 0);  pq.push(make\_pair(0, s));  dist[s] = 0;  while (!pq.empty()) {  pair<int, int> top = pq.top();  pq.pop();  int v = top.second, c = -top.first;  if (vis[v]) continue;  vis[v] = 1;  for (int i = 0; i < edge[v].size(); ++i) {  Edge &e = edge[v][i];  int u = e.t;  if (e.c == 0) continue;  int ndist = dist[v] + e.w + pot[v]-pot[u];  if (ndist < dist[u]) {  dist[u] = ndist;  prv[u] = e.r;  pq.push(make\_pair(-ndist, u));  }  }  }  int v = t;  if (dist[t] == INF) break;  int flow = INF;  for (int i = 0; i < n; ++i) pot[i] += dist[i];  while (v != s) {  Edge &r = edge[v][prv[v]], &e = edge[r.t][r.r];  flow = min(flow, e.c);  v = r.t;  }  mf += flow;  v = t;  while (v != s) {  Edge &r = edge[v][prv[v]], &e = edge[r.t][r.r];  e.c -= flow;  r.c += flow;  mc += e.w \* flow;  v = r.t;  }  }  return make\_pair(mf, mc);  }  int main()  {  int n, m, s, t;  scanf("%d%d%d%d", &n, &m, &s, &t);  for (int i = 0; i < m; ++i) {  int u, v, c, w;  scanf("%d%d%d%d", &u, &v, &c, &w);  Edge a(v,c,w,edge[v].size()), b(u,0,-w,edge[u].size());  edge[u].push\_back(a);  edge[v].push\_back(b);  }  pair<int, int> ret = mcmf(n, s, t);  printf("%d %d\n", ret.first, ret.second);  return 0;  } |

# **Mincost MaxFlow with Negative Cost**

|  |
| --- |
| /\*\* Max Flow Min Cost \*\*/  /\* complexity: O(min(E^2 V log V, E log V F)) \*/  const int maxnodes = 2010;  int nodes = maxnodes;  int prio[maxnodes], curflow[maxnodes], prevedge[maxnodes], prevnode[maxnodes], q[maxnodes], pot[maxnodes];  bool inqueue[maxnodes];    struct Edge {  int to, f, cap, cost, rev;  };  vector<Edge> graph[maxnodes];  void addEdge(int s,int t,int cap,int cost){  Edge a ={t,0, cap, cost, graph[t].size()};  Edge b ={s,0,0,-cost, graph[s].size()};  graph[s].push\_back(a);  graph[t].push\_back(b);  }  void bellmanFord(int s,int dist[]){  fill(dist, dist + nodes,1000000000);  dist[s]=0;  int qt =0;  q[qt++]= s;  for(int qh =0;(qh - qt)% nodes !=0; qh++){  int u = q[qh % nodes];  inqueue[u]=false;  for(int i =0; i <(int) graph[u].size(); i++){  Edge &e = graph[u][i];  if(e.cap <= e.f)continue;  int v = e.to;  int ndist = dist[u]+ e.cost;  if(dist[v]> ndist){  dist[v]= ndist;  if(!inqueue[v]){  inqueue[v]=true;  q[qt++% nodes]= v;  }  }  }  }  }  pair<int, int> minCostFlow(int s,int t,int maxf){  // bellmanFord can be safely commented if edges costs are non-negative  bellmanFord(s, pot);  int flow =0;  int flowCost =0;  while(flow < maxf){  priority\_queue<ll, vector<ll>, greater<ll>> q;  q.push(s);  fill(prio, prio + nodes,1000000000);  prio[s]=0;  curflow[s]=1000000000;  while(!q.empty()){  ll cur = q.top();  int d = cur >>32;  int u = cur;  q.pop();  if(d != prio[u])continue;  for(int i =0; i <(int) graph[u].size(); i++){  Edge &e = graph[u][i];  int v = e.to;  if(e.cap <= e.f)continue;  int nprio = prio[u]+ e.cost + pot[u]- pot[v];  if(prio[v]> nprio){  prio[v]= nprio;  q.push(((ll) nprio <<32)+ v);  prevnode[v]= u;  prevedge[v]= i;  curflow[v]= min(curflow[u], e.cap - e.f);  }  }  }  if(prio[t]==1000000000)break;  for(int i =0; i < nodes; i++) pot[i]+= prio[i];  int df = min(curflow[t], maxf - flow);  flow += df;  for(int v = t; v != s; v = prevnode[v]){  Edge &e = graph[prevnode[v]][prevedge[v]];  e.f += df;  graph[v][e.rev].f -= df;  flowCost += df \* e.cost;  }  }  return make\_pair(flow, flowCost);  }  /\* usage example:  \* addEdge (source, target, capacity, cost)  \* minCostFlow(source, target, INF) -><flow, flowCost>  \*/ |

# **Maximum Cardinality Bipartite Matching**

The code below finds a augmenting path:

|  |
| --- |
| bool dfs(int v){// v is in X, it reaturns true if and only if there is an augmenting path starting from v  if(mark[v])  return false;  mark[v] = true;  for(auto &u : adj[v])  if(match[u] == -1 or dfs(match[u])) // match[i] = the vertex i is matched with in the current matching, initially -1  return match[v] = u, match[u] = v, true;  return false; } |

An easy way to solve the problem is:

|  |
| --- |
| for(int i = 0;i < n;i ++)if(match[i] == -1){  memset(mark, false, sizeof mark);  dfs(i); } |

But there is a faster way:

|  |
| --- |
| while(true){  memset(mark, false, sizeof mark);  bool fnd = false;  for(int i = 0;i < n;i ++) if(match[i] == -1 && !mark[i])  fnd |= dfs(i);  if(!fnd)  break; } |

# 

# **Finding Cut Vertices & Cut Edges**

|  |
| --- |
| // Tarjan version again  void dfs(int v) {  low[v]= num[v] = ++cntr;  for(auto u : adj[v]) {  if(num[u] == -1) {  par[u] = v;  if(v == Root) rootChild++;    dfs(u);    if(low[u] >= num[v])  articulation\_vertex[v] = true;  if(low[u] > num[v])  printf("Edge (%d %d) is a bridge\n", v, u);    low[v] = min(low[v], low[u]);  }  else if(u != parent[v])  low[v] = min(low[v],num[u]);  }  }  // Inside Main  cntr = 0;  num.assign(n, -1);  low.assign(n, 0);  par.assign(n, -1);  articulation\_vertex.assign(n, 0);  for(int i = 0; i<n; i++) if(num[i] == -1) {  Root = i;  rootChild = 0;  dfs(i);  articulation\_vertex[i] = (rootChild > 1);  } |

# **Rumus-rumus kombin**





















Dixon Identity:



where *a*, *b*, and *c* are non-negative integers





**Lucas’ Theorem :**

For non-negative integers *m* and *n* and a prime *p*, the following [congruence relation](https://en.wikipedia.org/wiki/Modular_arithmetic) holds:



where

m=m_kp^k+m_{k-1}p^{k-1}+\cdots +m_1p+m_0,

and

n=n_kp^k+n_{k-1}p^{k-1}+\cdots +n_1p+n_0

are the base *p* expansions of *m* and *n* respectively. This uses the convention that \tbinom{m}{n} = 0 if *m* < *n*.

Example : (combinatrics in small mod wheren mod < n && mod < k)

|  |
| --- |
| int comb[mod][mod];  int c(int n, int k) {  return n == 0? 1 : comb[n%mod][k%mod] \* c(n/mod, k/mod) % mod;  } |

**Faulhaber’s Formula**

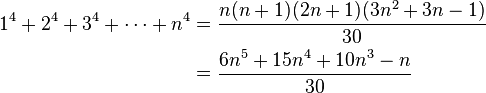


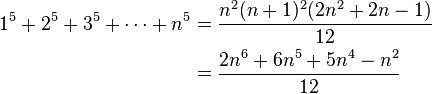
Examples:

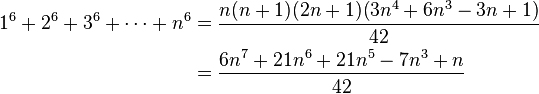
1 + 2 + 3 + \cdots + n = {n(n+1) \over 2} = {n^2 + n \over 2} (the [triangular numbers](https://en.wikipedia.org/wiki/Triangular_number))

1^2 + 2^2 + 3^2 + \cdots + n^2 = {n(n+1)(2n+1) \over 6} = {2n^3 + 3n^2 + n \over 6} (the [square pyramidal numbers](https://en.wikipedia.org/wiki/Square_pyramidal_number))

1^3 + 2^3 + 3^3 + \cdots + n^3 = \left({n(n+1) \over 2}\right)^2 = {n^4 + 2n^3 + n^2 \over 4} (the [squared triangular numbers](https://en.wikipedia.org/wiki/Squared_triangular_number))







# **Template Geometri**

|  |
| --- |
| // rotate p by theta degrees CCW w.r.t origin(0, 0)  point rotate(point p, double tetha) {  // rotate matrix R(theta0 = [cos(theta) -sin(theta)]  // [sin(theta) cos(theta)]  double rad = tehta \* PI / 180.0;  return point(p.x\*cos(rad) - p.y\*sin(rad), p.x\*sin(rad) + p.y\*cos(rad));  }  // (LINE)  struct line { double a,b,c; };  void pointToLine(point p1, point p2, line \*l) {  if(p1.x == p2.x) {  l->a = 1.0; l->b = 0.0; l->c = -p1.x;  }  else {  l->a = -(double)(p1.y-p2.y)/(p1.x-p2.x)l  l->b = 1.0;  l->c = -(double)(l->a \* p1.x) - (l->b \* p1.y);  }  }  bool areIntersect(line l1, line l2, point \* p) {  if(areSame(l1, l2)) return false;  if(areParallel(l1, l2)) return false;    p->x = (l2.b\*l1.c - l1.b\*l2.c)/(l2.a\*l1.b - l1.a\*l2.b);  if(fabs(l1.b) > EPS)  p->y = (l1.a\*p->x + l1.c)/l1.b;  else  p->y = (l2.a\*p->x + l2.c)/l2.b;  return true;  }  double area(const vector< point > & P) {  double result = 0.0;  for(int i = 0; i< (int)P.size()-1; i++) {  result += (P[i].x \* P[i+1].y - P[i].y\*P[i+1].x);  }  return fabs(result)/2.0;  }  // calculate angle between BA and BC  double angle(point a, point b, point c) {  double ux = a.x - b.x, uy = a.y - b.y;  double vx = c.x - b.x, vu = c.y - b.y;  return acos(ux\*vx + uy\*vy)/sqrt((ux\*ux + uy\*uy)\*(vx\*vx + vy\*vy)); }  // check if point p inside (CONVEX/CONCAVE) polygon vp  int inPolygon(point p, const vector< point >& vp) {  int wn = 0, n = (int)vp.size() - 1;  for(int i = 0; i<n; i++) {  long long cs = cross(vp[i+1], vp[i], p);  if(cs == 0 && 1LL \* (vp[i].x - p.x) \* (vp[i+1].x - p.x ) <= 0 && 1LL \* (vp[i].y - p.y) \* (vp[i+1].y - p.y ) <= 0)  return 1;  if(vp[i].y <= p.y) {  if(vp[i+1].y > p.y && cs > 0)  wn++;  }  else {  if(vp[i+1].y <= p.y && cs < 0)  wn--;  }  }  return wn;  } |

# 

# **Find two center of same size circle from its intersection and their radius**

|  |
| --- |
| bool circle2PtsRad(point p1, point p2, double r, point \*c) { // answer at \*c  double d2 = (p1.x - p2.x) \* (p1.x - p2.x) + (p1.y - p2.y)\*(p1.y - p2.y);  double det = r \* r / d2 - 0.25;  if(det<0.0) return false;  double h = sqrt(det);  c->x = (p1.x + p2.x) \* 0.5 + (p1.y-p2.y)\*h;  c->y = (p1.y + p2.y) \* 0.5 + (p2.x-p1.x)\*h;  return true;  } |

# **The Great-Circle Distance (SPHERES)**

|  |
| --- |
| double gcDistance(double plat, double plong, double qlat, double, qlong ,double radius) {  plat \*= PI/180; plong \*= PI/180;  qlat \*= PI/180; qlong \*= PI/180;  return radius \* acos(cos(plat)\*cos(plong)\*cos(qlat)\*cos(qlong) + cos(plat)\*sin(plong)\*cos(qlat)\*sin(qlong) +  sin(plat)\*sin(qlat));  } |

# 

# **Cutting Polygon with a Straight Line**

|  |
| --- |
| const long double EPS = 1e-7;  struct point {  double x, y;  point(double x, double y) : x(x), y(y) {} };  double cross(point p, point q, point r) {  return (p.x - q.x) \* (r.y - q.y) - (p.y - q.y) \* (r.x - q.x); }  // line segment p-q intersect with line A-B  point lineIntersectSeg(point p, point q, point A, point B) {  double a = B.y - A.y;  double b = A.x - B.x;  double c = B.x \* A.y - A.x \* B.y;  double u = fabs(a \* p.x + b \* p.y + c);  double v = fabs(a \* q.x + b \* q.y + c);  return point((p.x\*v + q.x\*u)/(u+v), (p.y\*v + q.y\*u)/(u+v));  }  // cuts polygon Q along the line formed by point a-> point b  // (note: the last point must be the same as the first point)  vector<point> cutPolygon(point a, point b, vector<point> Q) {  vector<point> P;  for(int i = 0; i<(int)Q.size(); i++) {  double left1 = cross(a, b, Q[i]), left2 = 0.0;  if(i != (int)Q.size()-1) left2 = cross(a, b, Q[i+1]);  if(left1 > -EPS) P.push\_back(Q[i]);  if(left1 \* left2 < -EPS)  P.push\_back(lineIntersectSeg(Q[i], Q[i+1], a, b));  }  if(P.empty()) return P;  if(fabs(P.back().x - P.front().x) > EPS ||  fabs(P.back().y - P.front().y) > EPS)  P.push\_back(P.front());  return P;  } |

# **Convex hull (Graham’s Scan & Andrew’s Monotone Chain)**

|  |
| --- |
| typedef pair<long long,long long> point;  #define x first  #define y second  // (p-q) x (r-q)  long long cross(point p, point q, point r) {  return (p.x - q.x) \* (r.y - q.y) - (p.y - q.y) \* (r.x - q.x);  }  bool collinear(point a, point o, point b) {  return cross(a, o, b) == 0;  }  // true if point r is on the left side of line pq  bool ccw(point p, point q, point r) {  return cross(p, q, r) > 0;  }  point pivot;  long long dist2(point a, point b) {  return (a.x - b.x) \* (a.x - b.x) + (a.y - b.y) \* (a.y - b.y);  }  bool angle\_cmp(point a, point b) {  if(collinear(pivot, a, b)) {  return dist2(a, pivot) < dist2(b, pivot);  }  return ccw(pivot, a, b);  }  bool cmp(point a, point b) {  return a.y < b.y || (a.y == b.y && a.x < b.x);  }  // P tidak siklik, P[0] tidak mengulang di P.back()  // return convex hull siklik, P[0] mengulang di P.back()  vector<point> ConvexHull(vector<point> P) {  int i, j, n = (int) P.size();  if(n < 3)  return P;  int PO = 0;  for(i = 1; i < n; i++) {  if(cmp(P[P0], P[P[i]])) {  PO = i;  }  }  swap(P[0], P[PO]);  pivot = P[0];  if(collinear(P.back(), P[0], P[1])) {  vector< point > S;  S.push\_back(P[0]);  S.push\_back(P.back());  return S;  }  sort(++P.begin(), P.end(), angle\_cmp);  int k = P.size() - 1;  while(k && collinear(P[0], P[k-1], P[k])) k--;  reverse(P.begin() + k, P.end());  vector<point> S;  S.push\_back(P[n-1]);  S.push\_back(P[0]);  S.push\_back(P[1]);  i = 2;  while(i < n) {  j = (int) S.size() - 1;  if(ccw(S[j-1], S[j], P[i])) S.push\_back(P[i++]);  else S.pop\_back();  }  S.pop\_back();  return S;  }  int main(void)  {  int n;  scanf("%d", &n);  vector<point> p;  for(int i = 0; i < n; i++) {  int a, b;  scanf("%d %d", &a, &b);  p.push\_back(point(a, b));  }  vector<point> ch = ConvexHull(p);  cout << ch.size() << endl;  for(auto it : ch) {  printf("%I64d %I64d\n", it.x, it.y);  }  return 0;  } |

|  |
| --- |
| // Andrew’s Monotone Chain  struct Point {  int x, y;  Point() {}  Point(int x, int y): x(x), y(y) {}   Point operator + (const Point& a) const {  return Point(x+a.x, y+a.y);  }  Point operator - (const Point& a) const {  return Point(x-a.x, y-a.y);  }  int operator % (const Point& a) const {  return x\*a.y - y\*a.x;  }  bool operator<(const Point &rhs) const { return make\_pair(y,x) < make\_pair(rhs.y,rhs.x); }  bool operator==(const Point &rhs) const { return make\_pair(y,x) == make\_pair(rhs.y,rhs.x); } }; int ccw(Point a, Point b, Point c) {  int t = (b - a) % (c - a);  if (t == 0) return 0;  if (t < 0) return -1;  return 1; }  typedef vector< Point > Polygon; int area2(Point a, Point b, Point c) { return a%b + b%c + c%a; } bool between(const Point &a, const Point &b, const Point &c) {  return (area2(a,b,c) == 0 && (a.x-b.x)\*(c.x-b.x) <= 0 && (a.y-b.y)\*(c.y-b.y) <= 0); } void ConvexHull(vector<Point> &pts) {  sort(pts.begin(), pts.end());  pts.erase(unique(pts.begin(), pts.end()), pts.end());  vector<Point> up, dn;  for (int i = 0; i < pts.size(); i++) {  while (up.size() > 1 && area2(up[up.size()-2], up.back(), pts[i]) >= 0) up.pop\_back();  while (dn.size() > 1 && area2(dn[dn.size()-2], dn.back(), pts[i]) <= 0) dn.pop\_back();  up.push\_back(pts[i]);  dn.push\_back(pts[i]);  }  pts = dn;  for (int i = (int) up.size() - 2; i >= 1; i--) pts.push\_back(up[i]);    if (pts.size() <= 2) return;  dn.clear();  dn.push\_back(pts[0]);  dn.push\_back(pts[1]);  for (int i = 2; i < pts.size(); i++) {  if (between(dn[dn.size()-2], dn[dn.size()-1], pts[i])) dn.pop\_back();  dn.push\_back(pts[i]);  }  if (dn.size() >= 3 && between(dn.back(), dn[0], dn[1])) {  dn[0] = dn.back();  dn.pop\_back();  }  pts = dn; } |

# **Pick’s Theorem**

Given a [simple polygon](https://en.wikipedia.org/wiki/Simple_polygon) constructed on a grid of equal-distanced points (i.e., points with [integer](https://en.wikipedia.org/wiki/Integer) coordinates) such that all the polygon's vertices are grid points, **Pick's theorem** provides a simple [formula](https://en.wikipedia.org/wiki/Formula) for calculating the [area](https://en.wikipedia.org/wiki/Area) *A* of this polygon in terms of the number *i* of *lattice points in the interior*located in the polygon and the number *b* of *lattice points on the boundary* placed on the polygon's perimeter:[[1]](https://en.wikipedia.org/wiki/Pick%27s_theorem#cite_note-1)

A = i + \frac{b}{2} - 1.

# **Strongly Connected Component**

|  |
| --- |
| /\*\*\*\*\*\* Tarjan’s SCC \*\*\*\*\*\*\*/  vector< int > num, low, S, vis;  int cntr, numCC;  void tarjanSCC(int v) {  low[v] = num[v] = ++cntr;  vis[v] = 1;  S.push\_back(v);  for(auto u : adj[v]) {  if(num[u] == -1)  tarjanSCC(u);  if(vis[u])  low[v] = min(low[v], low[u]);  }  if(low[v] == num[v]) {  printf("SCC %d :", ++numCC);  while(1) {  int u = S.back(); S.pop\_back(); vis[u] = 0;  printf(" %d", u);  if(u == v)  break;  }  }  }  // In MAIN();  num.assign(n, -1);  low.assign(n, 0);  vis.assign(n, 0);  cntr = numCC = 0;  for(int i = 0; i<n; i++))  if(num[i] == -1)  tarjanSCC(i); |

# **Suffix Array + LCP**

|  |
| --- |
| // suffix array  const int N = 1e5 + 5;  string s;  int sa[N], pos[N], lcp[N], tmp[N], gap, n;  bool cmp\_sa(int a, int b) {  if(pos[a] - pos[b])  return pos[a] < pos[b];  a += gap; b += gap;  return (a < n && b < n) ? pos[a] < pos[b] : a > b;  }  void build\_sa() {  n = s.size();  for(int i = 0; i<n; i++)  sa[i] = i, pos[i] = s[i];  for(gap = 1;; gap <<= 1) {  sort(sa, sa + n, cmp\_sa);  for(int i = 1; i<n; i++) tmp[i] = tmp[i-1] + cmp\_sa(sa[i-1], sa[i]);  for(int i = 0; i<n; i++) pos[sa[i]] = tmp[i];  if(tmp[n-1] == n-1) break;  }  }  void build\_lcp() {  for(int i = 0, k = 0; i<n; i++) if(pos[i] - n + 1) {  for(int j = sa[pos[i] + 1]; s[j + k] == s[i + k]; k++);  lcp[pos[i]] = k;  if(k) k--;  }  } |

# **Manacher Algorithm (Palindrom)**

|  |
| --- |
| Sumber : <http://e-maxx.ru/algo/palindromes_count>  vector<int> d1 (n); int l=0, r=-1; for (int i=0; i<n; ++i) {  int k = (i>r ? 0 : min (d1[l+r-i], r-i)) + 1;  while (i+k < n && i-k >= 0 && s[i+k] == s[i-k]) ++k;  d1[i] = k--;  if (i+k > r)  l = i-k, r = i+k; } vector<int> d2 (n); l=0, r=-1; for (int i=0; i<n; ++i) {  int k = (i>r ? 0 : min (d2[l+r-i+1], r-i+1)) + 1;  while (i+k-1 < n && i-k >= 0 && s[i+k-1] == s[i-k]) ++k;  d2[i] = --k;  if (i+k-1 > r)  l = i-k, r = i+k-1; }  Sumber : <http://codeforces.com/blog/entry/12143>  vector< vector<int> > p(2, vector<int>(n,0)); //p[1][i] even, p[0][i] odd palindrom center i for (int z=0, l=0, r=0; z < 2; z++, l=0, r=0)  for (int i = 0; i < n; i++) {  if (i < r) p[z][i] = min(r-i+!z, p[z][l+r-i+!z]);  int L = i-p[z][i], R = i+p[z][i]-!z;  while (L-1 >= 0 && R+1 < n && s[L-1] == s[R+1]) p[z][i]++, L--, R++;  if(R > r) l = L,r = R;  } |

# **Implicit Treap**

|  |
| --- |
| /\*\*  \* Treap uses implicit key  \* This Implementation : maintain array, can insert and delete in any position, can reverse interval  \*/    #include <bits/stdc++.h>  using namespace std;    typedef struct item \* pitem;    struct item  {  int cnt, value, prior;  bool rev;  pitem l, r;  item(int prior, int value) : cnt(1), rev(false), prior(prior), value(value), l(NULL), r(NULL) {}  };    int cnt(pitem t) {  return t ? t->cnt : 0;  }  void upd\_cnt(pitem it) {  if (it)  it->cnt = cnt(it->l) + cnt(it->r) + 1;  }    void push(pitem it) {  if (it && it->rev) {  it->rev = false;  swap(it->l, it->r);  if (it->l) it->l->rev ^= true;  if (it->r) it->r->rev ^= true;  }  }    void merge(pitem & t, pitem l, pitem r) {  push(l);  push(r);  if (!l || !r)  t = l ? l : r;  else if (l->prior > r->prior)  merge(l->r, l->r, r), t = l;  else  merge(r->l, l, r->l), t = r;  upd\_cnt(t);  }    void split(pitem t, pitem & l, pitem & r, int key, int add = 0) {  if (!t)  return void (l = r = 0);  int cur\_key = cnt(t->l) + add;  if (key <= cur\_key)  split(t->l, l, t->l, key, add), r = t;  else  split(t->r, t->r, r, key, add + cnt(t->l) + 1), l = t;  upd\_cnt(t);  }    void reverse(pitem t, int l, int r) {  pitem t1, t2, t3;  split(t, t1, t2, l);  split(t2, t2, t3, r-l+1);  t2->rev ^= true;  merge(t, t1, t2);  merge(t, t, t3);  }  void output (pitem t) {  if (!t) return;  push (t);  output (t->l);  printf ("%d ", t->value);  output (t->r);  }    int main() {  int n;  scanf("%d", &n);  srand(time(NULL));  pitem root = NULL;  for (int i = 0; i < n; i++) {  int a;  scanf("%d", &a);  pitem cur = new item(rand(), a);  if (root)  merge(root, root, cur);  else  root = cur;  }  int m;  scanf("%d", &m);  for (int i = 0; i < m; i++) {  int l, r;  scanf("%d %d", &l, &r);  reverse(root, l, r);  output(root);  printf("\n");  }  return 0;  } |

# **Convex Hull Trick**

|  |
| --- |
| #include <bits/stdc++.h>    using namespace std;  struct line {  long long a, b, get(long long x) {  return a\*x + b;  }  long double getd(long double x) {  return x \* a + b;  }  };    struct convex\_hull\_trick {  line \* hull;  int size;  convex\_hull\_trick(int sz) : size(0) {  hull = new line[sz+1];  }  bool isbad(line prev, line cur, line next) {  return (prev.b - cur.b) \* (next.a - cur.a) >= (cur.b - next.b) \* (cur.a - prev.a);  }  void add(line nl) {  hull[size++] = nl;  while(size > 2 && isbad(hull[size-3], hull[size-2], hull[size-1]))  hull[size-2] = nl, size--;  }    long long query(long long x) {  int l, r;  l = 0; r = size-1;  while(l < r) {  int m = (l + r) >> 1;  if(hull[m].get(x) <= hull[m+1].get(x))  l = m+1;  else  r = m;  }  return hull[l].get(x);  }  };  const int N = 2e5 + 5;  long long sum[N];  int a[N];    int main() {  int n;  scanf("%d", &n);  long long ans = 0, add = 0;  sum[0] = 0;  for(int i = 1; i<=n; i++) {  scanf("%d", a+i);  sum[i] = sum[i-1] + a[i];  ans += a[i] \* (long long)i;  }  convex\_hull\_trick hull(n);  hull.size = 0;  for(int i = 1; i <= n; i++) {  hull.add((line){i, -sum[i-1]});  add = max(add, hull.query(a[i]) + sum[i-1] - a[i]\*(long long)i);  }  hull.size = 0;  for(int i = n; i > 0; i--) {  hull.add((line){-i, -sum[i]});  add = max(add, hull.query(-a[i]) + sum[i] - a[i]\*(long long)i);  }  cout << ans + add << endl;  return 0;  } |

# **Z Algorithm**

|  |
| --- |
| string s;  cin >> s;  int L = 0, R = 0;  int n = s.size();  for (int i = 1; i < n; ++i) {  if (i > R) {  L = R = i;  while (R < n && s[R] == s[R-L]) ++R;  Z[i] = R-L; --R;  }  else {  int k = i-L;  if (Z[k] < R-i+1) Z[i] = Z[k];  else {  L = i;  while (R < n && s[R] == s[R-L]) ++R;  Z[i] = R-L; --R;  }  }  } |

# **Aho Corassick**

|  |
| --- |
| /\*\* Aho-Corasick Dictionary Matching \*\*/  constint NALPHABET **=**26**;**    struct Node **{**  Node**\*\*** children**,** go**;**  bool leaf**;**  char charToParent**;**  Node**\*** parent**,** suffLink**,** dictSuffLink**;**  int count**,** value**;**    Node**(){**  children **=new** Node**\*[**NALPHABET**];**  go **=new** Node**\*[**NALPHABET**];**  **for(**int i **=**0**;** i **<** NALPHABET**;++**i**){**  children**[**i**]=** go**[**i**]=NULL;**  **}**  parent **=** suffLink **=** dictSuffLink **=NULL;**  leaf **=false;**  count **=**0**;**  **}**  **};**    Node**\*** createRoot**(){**  Node**\*** node **=new** Node**();**  node**->**suffLink **=** node**;**  **return** node**;**  **}**    void addString**(**Node**\*** node**,**const string**&** s**,**int value **=-**1**){**  **for(**int i **=**0**;** i **<** s**.**length**();++**i**){**  int c **=** s**[**i**]-**'a'**;**  **if(**node**->**children**[**c**]==NULL){**  Node**\*** n **=new** Node**();**  n**->**parent **=** node**;**  n**->**charToParent **=** s**[**i**];**  node**->**children**[**c**]=** n**;**  **}**  node **=** node**->**children**[**c**];**  **}**  node**->**leaf **=true;**  node**->**count**++;**  node**->**value **=** value**;**  **}**    Node**\*** suffLink**(**Node**\*** node**);**  Node**\*** dictSuffLink**(**Node**\*** node**);**  Node**\*** go**(**Node**\*** node**,**char ch**);**  int calc**(**Node**\*** node**);**    Node**\*** suffLink**(**Node**\*** node**){**  **if(**node**->**suffLink **==NULL){**  **if(**node**->**parent**->**parent **==NULL){**  node**->**suffLink **=** node**->**parent**;**  **}else{**  node**->**suffLink **=** go**(**suffLink**(**node**->**parent**),**node**->**charToParent**);**  **}**  **}**  **return** node**->**suffLink**;**  **}**    Node**\*** dictSuffLink**(**Node**\*** node**){**  **if(**node**->**dictSuffLink **==NULL){**  Node**\*** n **=** suffLink**(**node**);**  **if(**node **==** n**){**  node**->**dictSuffLink **=** node**;**  **}else{**  **while(!**n**->**leaf **&&** n**->**parent **!=NULL){**  n **=** dictSuffLink**(**n**);**  **}**  node**->**dictSuffLink **=** n**;**  **}**  **}**  **return** node**->**dictSuffLink**;**  **}**    Node**\*** go**(**Node**\*** node**,**char ch**){**  int c **=** ch **-**'a'**;**  **if(**node**->**go**[**c**]==NULL){**  **if(**node**->**children**[**c**]!=NULL){**  node**->**go**[**c**]=** node**->**children**[**c**];**  **}else{**  node**->**go**[**c**]=** node**->**parent **==NULL?** node **:** go**(**suffLink**(**node**),** ch**);**  **}**  **}**  **return** node**->**go**[**c**];**  **}**    int calc**(**Node**\*** node**){**  **if(**node**->**parent **==NULL){**  **return**0**;**  **}else{**  **return** node**->**count **+** calc**(**dictSuffLink**(**node**));**  **}**  **}**    int main**(){**  Node**\*** root **=** createRoot**();**  addString**(**root**,**"a"**,**0**);**  addString**(**root**,**"aa"**,**1**);**  addString**(**root**,**"abc"**,**2**);**    string s**(**"abcaadc"**);**  Node**\*** node **=** root**;**  **for(**int i **=**0**;** i **<** s**.**length**();++**i**){**  node **=** go**(**node**,** s**[**i**]);**  Node**\*** temp **=** node**;**  **while(**temp **!=** root**){**  **if(**temp**->**leaf**){**  printf**(**"string (%d) occurs at position %d\n"**,** temp**->**value**,** i**);**  **}**  temp **=** dictSuffLink**(**temp**);**  **}**  **}**  **return**0**;**  **}** |

# Blossom

|  |
| --- |
| /\*\* Maximum Matching on General Graph \*\*/  /\* Blossom | O(V^3) \*/    int lca**(**vector**<**int**>&**match**,** vector**<**int**>&**base**,** vector**<**int**>&**p**,**int a**,**int b**){**  vector**<**bool**>** used**(**SZ**(**match**));**  **while(true){**  a **=** base**[**a**];**  used**[**a**]=true;**  **if(**match**[**a**]==-**1**)break;**  a **=** p**[**match**[**a**]];**  **}**  **while(true){**  b **=** base**[**b**];**  **if(**used**[**b**])return** b**;**  b **=** p**[**match**[**b**]];**  **}**  **return-**1**;**  **}**    void markPath**(**vector**<**int**>&**match**,** vector**<**int**>&**base**,** vector**<**bool**>&**blossom**,** vector**<**int**>&**p**,**int v**,**int b**,**int children**){**  **for(;** base**[**v**]!=** b**;** v **=** p**[**match**[**v**]]){**  blossom**[**base**[**v**]]=** blossom**[**base**[**match**[**v**]]]=true;**  p**[**v**]=** children**;**  children **=** match**[**v**];**  **}**  **}**    int findPath**(**vector**<**vector**<**int**>>&**graph**,** vector**<**int**>&**match**,** vector**<**int**>&**p**,**int root**){**  int n **=** SZ**(**graph**);**  vector**<**bool**>** used**(**n**);**  FORIT**(**it**,** p**)\***it **=-**1**;**  vector**<**int**>** base**(**n**);**  **for(**int i **=**0**;** i **<** n**;++**i**)** base**[**i**]=** i**;**    used**[**root**]=true;**  int qh **=**0**;**  int qt **=**0**;**  vector**<**int**>** q**(**n**);**  q**[**qt**++]=** root**;**  **while(**qh **<** qt**){**  int v **=** q**[**qh**++];**  FORIT**(**it**,** graph**[**v**]){**  int to **=\***it**;**  **if(**base**[**v**]==** base**[**to**]||** match**[**v**]==** to**)continue;**  **if(**to **==** root **||** match**[**to**]!=-**1**&&** p**[**match**[**to**]]!=-**1**){**  int curbase **=** lca**(**match**,** base**,** p**,** v**,** to**);**  vector**<**bool**>** blossom**(**n**);**  markPath**(**match**,** base**,** blossom**,** p**,** v**,** curbase**,** to**);**  markPath**(**match**,** base**,** blossom**,** p**,** to**,** curbase**,** v**);**  **for(**int i **=**0**;** i **<** n**;++**i**){**  **if(**blossom**[**base**[**i**]]){**  base**[**i**]=** curbase**;**  **if(!**used**[**i**]){**  used**[**i**]=true;**  q**[**qt**++]=** i**;**  **}**  **}**  **}**  **}elseif(**p**[**to**]==-**1**){**  p**[**to**]=** v**;**  **if(**match**[**to**]==-**1**)return** to**;**  to **=** match**[**to**];**  used**[**to**]=true;**  q**[**qt**++]=** to**;**  **}**  **}**  **}**  **return-**1**;**  **}**    int maxMatching**(**vector**<**vector**<**int**>>** graph**){**  int n **=** SZ**(**graph**);**  vector**<**int**>** match**(**n**,-**1**);**  vector**<**int**>** p**(**n**);**  **for(**int i **=**0**;** i **<** n**;++**i**){**  **if(**match**[**i**]==-**1**){**  int v **=** findPath**(**graph**,** match**,** p**,** i**);**  **while(**v **!=-**1**){**  int pv **=** p**[**v**];**  int ppv **=** match**[**pv**];**  match**[**v**]=** pv**;**  match**[**pv**]=** v**;**  v **=** ppv**;**  **}**  **}**  **}**    int matches **=**0**;**  **for(**int i **=**0**;** i **<** n**;++**i**){**  **if(**match**[**i**]!=-**1**){**  **++**matches**;**  **}**  **}**  **return** matches **/**2**;**  **}** |

# Minimum Cut Stoer - Wagner

|  |
| --- |
| // Adjacency matrix implementation of Stoer-Wagner min cut algorithm.  //  // Running time:  // O(|V|^3)  //  // INPUT:  // - graph, constructed using AddEdge()  //  // OUTPUT:  // - (min cut value, nodes in half of min cut)    #include <cmath>  #include <vector>  #include <iostream>    **usingnamespace** std**;**    **typedef** vector**<**int**>** VI**;**  **typedef** vector**<**VI**>** VVI**;**    const int INF **=**1000000000**;**    pair<int, VI> GetMinCut(VVI &weights){  int N = weights.size();  VI used(N), cut, best\_cut;  int best\_weight =-1;  for(int phase = N-1; phase >=0; phase--){  VI w = weights[0];  VI added = used;  int prev, last =0;  for(int i =0; i < phase; i++){  prev = last;  last =-1;  for(int j =1; j < N; j++)  if(!added[j]&&(last ==-1|| w[j]> w[last])) last = j;  if(i == phase-1){  for(int j =0; j < N; j++) weights[prev][j]+= weights[last][j];  for(int j =0; j < N; j++) weights[j][prev]= weights[prev][j];  used[last]=true;  cut.push\_back(last);  if(best\_weight ==-1|| w[last]< best\_weight){  best\_cut = cut;  best\_weight = w[last];  }  }else{  for(int j =0; j < N; j++)  w[j]+= weights[last][j];  added[last]=true;  }  }  }  return make\_pair(best\_weight, best\_cut);  } |

# 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**;**  **}** |

# 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>    **usingnamespace** std**;**    **typedef**longdouble 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**)returntrue;**  int r **=-**1**;**  **for(**int i **=**0**;** i **<** m**;** i**++){**  **if(**D**[**i**][**s**]<=**0**)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**)returnfalse;**  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**];**  **}**  **};**    int main**(){**  const int m **=**4**;**  const int n **=**3**;**  DOUBLE \_A**[**m**][**n**]={**  **{**6**,-**1**,**0**},**  **{-**1**,-**5**,**0**},**  **{**1**,**5**,**1**},**  **{-**1**,-**5**,-**1**}**  **};**  DOUBLE \_b**[**m**]={**10**,-**4**,**5**,-**5**};**  DOUBLE \_c**[**n**]={**1**,-**1**,**0**};**    VVD A**(**m**);**  VD b**(**\_b**,** \_b **+** m**);**  VD c**(**\_c**,** \_c **+** n**);**  **for(**int i **=**0**;** i **<** m**;** i**++)** A**[**i**]=** VD**(**\_A**[**i**],** \_A**[**i**]+** n**);**    LPSolver solver**(**A**,** b**,** c**);**  VD x**;**  DOUBLE value **=** solver**.**Solve**(**x**);**  cerr **<<**"VALUE: "**<<** value **<<** endl**;**  cerr **<<**"SOLUTION:"**;**  **for(**size\_t i **=**0**;** i **<** x**.**size**();** i**++)** cerr **<<**" "**<<** x**[**i**];**  cerr **<<** endl**;**  **return**0**;**  **}** |

# Persistent Segment tree

|  |
| --- |
| #include <bits/stdc++.h>  using namespace std;  struct Node {  int cnt;  Node \*left, \*right;  Node(int cnt, Node \*left, Node \*right) {  this->cnt = cnt;  this->left = left;  this->right = right;  }  Node \*insert(int l, int r, int k) {  if (!(l <= k && k <= r)) {  return this;  }  Node \*node = new Node(this->cnt + 1, this->left, this->right);  if (l == r) {  return node;  }  int m = (l + r) / 2;  node->left = node->left->insert(l, m, k);  node->right = node->right->insert(m + 1, r, k);  return node;  }  static int query(Node \*a, Node \*b, int l, int r, int k) {  if (l == r) {  return l;  }  int m = (l + r) / 2;  int cnt = b->left->cnt - a->left->cnt;  if (cnt >= k + 1) {  return query(a->left, b->left, l, m, k);  }  return query(a->right, b->right, m + 1, r, k - cnt);  }  };  int main() {  int n, m;  scanf("%d %d", &n, &m);  map<int, int> M, RM;  int a[n];  for (int i = 0; i < n; i++) {  scanf("%d", &a[i]);  M[a[i]];  }  int cntr = 0;  for (auto it = M.begin(); it != M.end(); it++) {  M[it->first] = cntr;  RM[cntr] = it->first;  cntr++;  }  cntr--;  Node \*null = new Node(0, NULL, NULL);  null->left = null, null->right = null;  Node \*roots[n];  for (int i = 0; i < n; i++) {  roots[i] = (i == 0 ? null : roots[i - 1])->insert(0, cntr, M[a[i]]);  }  int u, v, k;  while (m--) {  scanf("%d %d %d", &u, &v, &k);  u--, v--, k--;  Node \*a = (u == 0 ? null : roots[u - 1]);  Node \*b = roots[v];  int res = Node::query(a, b, 0, cntr, k);  printf("%d\n", RM[res]);  }  return 0;  } |