## Conteúdo



# 1 Grafos

## 1.1 Árvore de Steiner

Complexidade: O(3^t), t = número de terminais
Descricao: Encontra o grafo de menor custo que conecta todos os vértices terminais, podendo utilizar os demais vértices.

```
#include <vector>
#include <cstring>
#include <algorithm>
using namespace std;

#define INF 0x3f3f3f3f
#define MAXN 120
#define MAXT 10

/* FILL ME */
int adj[MAXN][MAXN]; /* matriz de adj com custos */
int tt[MAXT]; /* vértices terminais */
int n, nt; /* número de vértices e de terminais */

int memo[1<<MAXT][MAXT];

vector<int> mask[MAXT];

/* FLOYD AQUI */

void getMask(int mask, int e, int& x, int& y, int n) {
  int j = 0;
  x = 0;
  y = 0;
  for (int i = 0; i < n; i++) {
    while (!(mask & (1 << j))) {
      j++;
    }
    if (e & (1 << i)) {
      x = x | (1 << j);
    }
    else {
      y = y | (1 << j);
    }
    j++;
  }
}

int minstree() {
  floyd();
  if (nt == 2) return d[tt[0]][tt[1]];
  for (int t = 0; t < nt-1; t++) {
    mask[t].clear();
    for (int j = 0; j < n; j++) {
      memo[(1<<t)][j] = d[j][tt[t]];
    }
  }
  for (int i = 1; i <= (1 << (nt-1)) - 1; i++) {
    int x = __builtin_popcount(i);
    if (x > 1) {
      mask[x].push_back(i);
    }
  }
  for (int m = 2; m <= nt-2; m++) {
    for (int k = 0; k < mask[m].size(); k++) {
      int msk = mask[m][k];
      for (int i = 0; i < n; i++) {
        memo[msk][i] = INF;
      }
      for (int j = 0; j < n; j++) {
        int u = INF;
        int e;
        for (e = 0; e < (1 << (m-1)) - 1; e++) {
          e = e | (1 << (m-1));
```

```
          int x, y;
          getMask(msk, e, x, y, m);
          u = min(u, memo[x][j] + memo[y][j]);
        }
        for (int i = 0; i < n; i++) {
          memo[msk][i] = min(memo[msk][i], d[i][j] + u);
        }
      }
    }
  }
  int v = INF;
  int q = tt[nt-1];
  for (int j = 0; j < n; j++) {
    int u = INF;
    for (int e = 1; e < (1 << (nt -1)) -1; e++) {
      u = min(u, memo[e][j] +
        memo[e ^ ((1 << (nt -1)) -1)][j]);
    }
    v = min(v, d[q][j] + u);
  }
  return v;
}
```

### 1.2 Árvore Geradora Mínima (Prim)

```
Complexidade: O(m*logn)
Descricao: Encontra Arvore Geradora Minima

#include <queue>
#include <limits.h>
#include <cstdio>
#include <cstring>

using namespace std;

#define MAXN 101 //numero maximo de vertices
#define INF INT_MAX //nao ha perigo de overflow

/* FILL ME */
int adj[MAXN][MAXN]; //lista de adj
int custo[MAXN][MAXN]; //tamanho das arestas de adj
int nadj[MAXN]; //grau de cada vertice

int pai[MAXN]; //para reconstruir o caminho
int dist[MAXN]; //distancia de cada vertice a arvore
bool used[MAXN];

/*
  n: numero de vertices, s: origem (opcional)
  retorna peso total da arvore
*/
int prim(int n, int s = 0) {
  priority_queue<pair<int, int> > q;
  int a,v;
  int cost, nv = 0;
  int ret = 0;
  memset(pai,-1,sizeof(pai));
  memset(used,0,sizeof(used));
  for (int i = 0; i < n; i++) dist[i] = INF;
  dist[s] = 0;
  pai[s] = s;
  q.push(make_pair(0,s));
  while(!q.empty() && nv < n) {
    a = q.top().second; q.pop();
    if (used[a]) continue;
    ret += dist[a];
    used[a] = true;
    nv++;
    for (int i = 0; i < nadj[a]; i++) {
      v = adj[a][i];
      if(!used[v]){
        cost = custo[a][i];
        if (cost >= dist[v]) continue;
        dist[v] = cost;
        q.push(make_pair(-1*cost,v));
        pai[v] = a;
      }
    }
  }
  return ret;
}
```

### 1.3 Bellman Ford

```
Complexidade: O(n*m)
Descricao: Caminho minimo com pesos negativos

#define MAXN 100 //Numero maximo de vertices
#define MAXM 10000 //Numero maximo de arestas
#define INF 0x3f3f3f3f

/* aresta (u,v) com peso w:
   orig[i] = u, dest[i] = v, peso[i] = w
   d[u], distancia da origem s ao vertice u
*/

/* FILL ME */
int orig[MAXM], dest[MAXM], peso[MAXM];

int d[MAXN], pai[MAXN];
/*
s: origem, n: numero de vertices, m: numero de arestas
retorna 1 se o grafo nao tem ciclo negativo alcancavel
a partir de s, 0 c.c
*/

int bellman_ford(int s, int n, int m) {
  int i, j;
  memset(pai,-1,sizeof(pai));
  pai[s] = s;
  for (i = 0; i < n; i++)
    d[i] = INF;
  d[s] = 0;
  for (i = 0; i < n-1; i++)
    for (j = 0; j < m; j++) {
      int u = orig[j], v = dest[j], w = peso[j];
      if (d[u] != INF && d[v] > d[u]+w) {
        d[v] = d[u]+w;
        pai[v] = u;
      }
    }
  for (j = 0; j < m; j++) {
    int u = orig[j], v = dest[j], w = peso[j];
    if (d[u] != INF && d[v] > d[u]+w) return 0;
  }
  return 1;
}
```

### 1.4 Circuito e Passeio de Euler

```
Complexidade: O(n+m)
Descricao: Verifica se há e encontra um circuito/passeio
de Euler em grafo não-direcionado contendo todas arestas
podendo ser paralelas ou laços e o grafo conexo ou não

#include <queue>
#include <cstring>
#include <vector>
#include <algorithm>
#include <stack>

using namespace std;
#define MAXM 100100
#define MAXN 100100

/* FILL ME */
int ea[MAXM], eb[MAXM], n, m;

vector<int> vtour; // resposta: lista de vértices
vector<int> g[MAXN];
int mrk[MAXM];

/* Retorna 1 se há circuito, 2 se há passeio ou 0 c.c */
int euler() {
  for (int i=0; i<n; i++) g[i].clear();

  for (int i=0; i<m; i++) {
    g[ea[i]].push_back(i);
    g[eb[i]].push_back(i);
    mrk[i] = 0;
  }

  int qi = 0, v0;
  for (int i=0; i<n; i++) {
    if (!qi && g[i].size()) v0 = i;
    if (g[i].size() % 2) v0 = i, qi++;
  }

  if (qi > 2) return 0;

  stack<int> st;
  st.push(v0);

  vtour.clear();
  while (!st.empty()) {
    int v = st.top();

    while (g[v].size() && mrk[g[v].back()]++)
      g[v].pop_back();
```

```
    if (g[v].empty()) {
      vtour.push_back(v);
      st.pop();
    }
    else {
      int k = g[v].back();
      st.push(v == ea[k] ? eb[k] : ea[k]);
    }
  }

  return (vtour.size() == m+1) ? (1+qi/2) : 0;
}
```

## 1.5 Cliques Maximais

```
Complexidade: O(3^(n/3))
Descricao: Acha todas as cliques maximais de um grafo.

#include <cstring>
#include <algorithm>
using namespace std;

typedef long long int int64;

#define MAXN 55
#define INF 0x3f3f3f3f

/* FILL ME */
int n;
/* matriz de adj representada por mascara de bits */
int64 adj[MAXN];

void clique(int64 r, int64 p, int64 x) {
  if (p == 0 && x == 0) {
    /* r é uma clique maximal */
    return;
  }
  int pivot = -1;
  int menor = INF;
  for (int i = 0; i < n; i++) {
    if ( ((1LL << i) & p) || ((1LL << i) & x) ) {
      int x = __builtin_popcountll(p & (~(adj[i])));
      if (x < menor) {
        pivot = i;
        menor = x;
      }
    }
  }

  for (int i = 0; i < n; i++) {
    if ((1LL << i) & p) {
      if (pivot != -1 && adj[pivot] & (1LL << i)) continue;
      clique(r | (1LL << i), p & adj[i], x & adj[i]);
      p = p ^ (1LL << i);
      x = x | (1LL << i);
    }
  }
}
```

## 1.6 Cobertura Mínima por Caminhos em DAG

```
Complexidade: O(n*m)
Descricao: Dado uma DAG encontra o menor número de
caminhos necessários para cobrir todos os vértices.
Cada vértice é coberto exatamente uma vez.

#include <vector>
#include <cstring>
#include <algorithm>
using namespace std;

#define MAXNDAG 130
#define MAXN 2*MAXNDAG

/* BPM AQUI */

/* FILL ME */
int n;
vector<int> dag[MAXNDAG];

int minpcover() {
  memset(nadj, 0, sizeof(nadj));
  nU = nV = n;

  for (int i = 0; i < n; i++) {
    for (int j = 0; j < (int) dag[i].size(); j++) {
      int v = dag[i][j];
      adj[i][nadj[i]++] = v+n;
      adj[v+n][nadj[v+n]++] = i;
    }
  }

  return n - maxbpm();
}

/* Abaixo apenas se for necessário imprimir a solução*/
vector<int> path[MAXNDAG];

void DFS(int u, int c) {
  path[c].push_back(u);
  if (conj[u+n] == -1) return;
  DFS(conj[u+n], c);
}

int getPaths() {
  int res = 0;
  for (int i = 0; i < n; i++) {
    if (conj[i] == -1) {
      path[res].clear();
      DFS(i, res);
      reverse(path[res].begin(), path[res].end());
      res++;
    }
  }
  return res;
}
```

## 1.7 Componentes Fortemente Conexas

```
Complexidade: O(n+m)
Descricao: Encontra as componentes fortemente conexas de um
grafo orientado. Componentes nomeadas de 1 à ncomp.

#include <cstring>
#include <algorithm>
#include <vector>
using namespace std;

#define MAXN 1024

/* Input - FILL ME */
vector<int> adj[MAXN];

/* Output */
int ncomp;      // numero de componentes fortemente conexas
int comp[MAXN]; // comp[i] = componente do vertice i

/* Variaveis auxiliares */
int vis[MAXN], stck[MAXN], t, high[MAXN];
int num;

void dfscc(int u) {
  int i, v;
  high[u] = vis[u] = num--;
  stck[t++] = u;
  for (i = 0; i < (int) adj[u].size(); i++) {
    v = adj[u][i];
    if (!vis[v]) {
      dfscc(v);
      high[u] = max(high[u], high[v]);
    } else if (vis[v] > vis[u] && !comp[v])
      high[u] = max(high[u], vis[v]);
  }
  if (high[u] == vis[u]) {
    ncomp++;
    do {
      v = stck[--t];
      comp[v] = ncomp;
    } while (v != u);
  }
}

void scc(int n) {
  ncomp = t = 0; num = n;
  memset(vis, 0, sizeof(vis));
  memset(comp, 0, sizeof(comp));
  for (int i = 0; i < n; i++)
    if (!vis[i]) dfscc(i);
}
```

## 1.8 Corte Mínimo Geral (Stoer-Wagner)

```
Complexidade: O(n^3)
Descricao: Algoritmo que encontra o valor do corte mínimo
dentre todos de um grafo nao-orientado com peso na aresta.

#include <algorithm>
using namespace std;

// Maximum number of vertices in the graph
```

```
#define MAXN 256

// Maximum edge weight (MAXW * NN * NN must fit into an int)
#define MAXW 1000

// Adjacency matrix and some internal arrays
int adj[MAXN][MAXN], v[MAXN], w[MAXN], na[MAXN];
bool a[MAXN];

int mincut(int n) {
  // init the remaining vertex set
  for(int i = 0; i < n; i++) v[i] = i;

  // run Stoer-Wagner
  int best = MAXW * n * n;
  while(n > 1) {
    // initialize the set A and vertex weights
    a[v[0]] = true;
    for( int i = 1; i < n; i++ ) {
      a[v[i]] = false;
      na[i - 1] = i;
      w[i] = adj[v[0]][v[i]];
    }

    // add the other vertices
    int prev = v[0];
    for(int i = 1; i < n; i++) {
      // find the most tightly connected non-A vertex
      int zj = -1;
      for(int j = 1; j < n; j++)
        if(!a[v[j]] && (zj < 0 || w[j] > w[zj]))
          zj = j;

      // add it to A
      a[v[zj]] = true;

      // last vertex?
      if(i == n - 1) {
        // remember the cut weight
        best = min(best, w[zj]);

        // merge prev and v[zj]
        for(int j = 0; j < n; j++)
          adj[v[j]][prev]=adj[prev][v[j]] += adj[v[zj]][v[j]];
        v[zj] = v[--n];
        break;
      }
      prev = v[zj];

      // update the weights of its neighbours
      for(int j = 1; j < n; j++)
        if(!a[v[j]])
          w[j] += adj[v[zj]][v[j]];
    }
  }
  return best;
}
```

## 1.9 Dijkstra

```
Complexidade: O(m*logn)
Descricao: Encontra caminho minimo em grafos com pesos >= 0

#include <queue>
#include <cstring>
using namespace std;

#define MAXN 101
#define INF INT_MAX //nao ha perigo de overflow

/* FILL ME */
int adj[MAXN][MAXN], nadj[MAXN]; //lista de adj
int cus[MAXN][MAXN]; //tamanho das arestas de adj

int dist[MAXN]; //distancia da origem ateh cada vertice
bool used[MAXN];
//int pai[MAXN]; //Caso queira reconstruir o caminho

// preenche o vetor de distancias dist
void dijkstra(int n, int s) {
  priority_queue<pair<int, int> > q;

  //memset(pai,-1,sizeof(pai));
  memset(used,0,sizeof(used));
  for (int i = 0; i < n; i++) dist[i] = INF;
  dist[s] = 0;
  //pai[s] = s;

  q.push(make_pair(0,s));
  while (!q.empty()) {
    int a = q.top().second;
    q.pop();

    if (used[a]) continue;
    used[a] = true;

    for (int i = 0; i < nadj[a]; i++) {
      int v = adj[a][i];
      int cost = dist[a] + cus[a][i];
      if (cost >= dist[v]) continue;
      dist[v] = cost;
      q.push(make_pair(-1*cost,v));
      //pai[v] = a;
    }
  }
}
```

## 1.10 Emparalhamento Bipartido de Custo Máximo

```
Complexidade: O(n^3)
Descricao: Encontra o emparelhamento maximo
de custo maximo, para custo minimo insira as
arestas com peso negativo. Se uma aresta nao
existe o valor na matriz deve ser -1*INF.
Cuidado: NAO UTILIZAR MEMSET PARA O -1*INF
necessariamente n <= m

#include <algorithm>
#include <vector>
#include <cstring>
using namespace std;

#define INF 0x3f3f3f3f
#define MAXN 351

/* FILL ME */
int n, m; //# de vertices em cada lado
int adj[MAXN][MAXN]; //Matriz de Adj

int labelx[MAXN], usedx[MAXN], lnk[MAXN];
int labely[MAXN], usedy[MAXN];
int mat; //Tamanho to match

//Auxiliar Caminho Aumentante
bool path(int i) {
  usedx[i] = 1;
  for (int j = 0; j < m; j++) {
    if (!usedy[j] && adj[i][j] != -INF &&
        !abs(adj[i][j] - labelx[i] - labely[j])) {
      usedy[j] = 1;
      if (lnk[j] == -1 || path(lnk[j])) {
        lnk[j] = i;
        return true;
      }
    }
  }
  return false;
}

//Apos preencher adj chamar match()
int match() {
  mat = 0;
  memset(lnk,-1,sizeof(lnk));
  memset(labely,0,sizeof(labely));
  for (int i = 0; i < n; i++) {
    labelx[i] = 0;
    for (int j = 0; j < m; j++)
      if (adj[i][j] > labelx[i]) labelx[i] = adj[i][j];
  }
  for (int k = 0; k < n; k++) {
    while (1) {
      memset(usedx,0,sizeof(usedx));
      memset(usedy,0,sizeof(usedy));
      if (path(k)) { mat++; break; }
      int del = INF;
      for (int i = 0; i < n; i++)
        if (usedx[i])
          for (int j = 0; j < m; j++)
            if (!usedy[j] && adj[i][j] != -INF)
              del = min(del,labelx[i]+labely[j]-adj[i][j]);
      if (del == 0 || del == INF) break;
      for (int i = 0; i < n; i++)
        if (usedx[i]) labelx[i] -= del;
      for (int j = 0; j < m; j++)
        if (usedy[j]) labely[j] += del;
    }
  }
  int sum = 0;
  for (int i = 0; i < n; i++) sum += labelx[i];
```

```
  for (int j = 0; j < m; j++) sum += labely[j];
  return sum;
}
```

### 1.11 Emparelhamento Máximo e Cobertura Mínima

```
Complexidade: O(n*m)
Descricao: Encontra um emparelhamento máximo e uma cobertura
de vértice mínima em grafo bipartido.
Para um grafo bipartido:
- |cobertura mínima de vertices| = |emparelhamento máximo|
- |conj. independente máximo| = n - |emparelhamento máximo|
- |cobertura mínima de arestas| = n - |emparelhamento máximo|

#include <cstring>
#include <vector>
using namespace std;
#define MAXN 2024

/* Input - FILL ME */
vector<int> adj[MAXN];
/* Output */
int conj[MAXN]; // conj[u] = v, se v for emparelhado com u
int cor[MAXN];  // cor[u] = partição do vertice u (0 ou 1)
/* Variavel auxiliar */
int vis[MAXN];

/* Função auxiliar */
bool DFS(int u, int c) {
  cor[u] = c;
  for (int i = 0; i < (int) adj[u].size(); i++) {
    int v = adj[u][i];
    if (cor[v] == c) return false;
    if (cor[v] == -1) {
      if (!DFS(v, c^1)) return false;
    }
  }
  return true;
}

/* Função auxiliar */
int aumenta(int u) {
  int i;
  for (i = 0; i < (int) adj[u].size(); i++) {
    int v = adj[u][i];
    if (vis[v]) continue; vis[v] = 1;
    if (conj[v] == -1 || aumenta(conj[v])) {
      conj[v] = u;
      conj[u] = v;
      return 1;
    }
  }
  return 0;
}

int maxbpm(int n) {
  int i;
  int res = 0;
  memset(cor, -1, sizeof(cor));
  for (int i = 0; i < n; i++) {
    if (cor[i] == -1)
      if (!DFS(i, 0)) return -1; // grafo não é bipartido
  }
  memset(conj, -1, sizeof(conj));
  for (i = 0; i < n; i++) {
    if (cor[i]) continue;
    memset(vis, 0, sizeof(vis));
    if (aumenta(i)) res++;
  }
  return res;
}

/* Código apenas para cobertura mínima de vertice */

/* Output */
bool cover[MAXN]; //cover[i] -> vertice i pertence a cobertura

/* Função auxiliar */
void reach(int u) {
  int i;
  vis[u] = 1;
  for (i = 0; i < (int) adj[u].size(); i++) {
    int v = adj[u][i];
    if (!vis[v] && ((conj[u] != v && !cor[u]) ||
                    (conj[u] == v && cor[u]))) reach(v);
  }
}

int minbpec(int n) {
  int i;
  int res = maxbpm(n);
  memset(vis, 0, sizeof(vis));
  for (i = 0; i < n; i++) {
    if (cor[i]) continue;
    if (!vis[i] && conj[i] == -1) reach(i);
  }
  /* C = (U \ Rm) \/ (V /\ Rm) */
  memset(cover, false, sizeof(cover));
  for (i = 0; i < n; i++) {
    if (!vis[i] && !cor[i]) cover[i] = true;
    if (vis[i] && cor[i]) cover[i] = true;
  }
  return res;
}
```

### 1.12 Emparelhamento Máximo Geral (Edmonds)

```
Complexidade: O(n^3)
Uso: CUIDADO - Nao utilizar o vertice 0
- Para cada aresta 'i' (sem direcao) u-v,
faca from[i] = u, to[i] = v e coloque i
na lista de adjacencia de ambos u e v.
- n e m devem ser utilizados obrigatoriamente.
- E() retorna o tamanho do emparalhamento (# de casais).
- mate[v] quando diferente de 0 indica que o vertice v esta
casado com mate[v]

#include <algorithm>
#include <queue>
#include <cstring>
using namespace std;

#define MAXN 110
#define MAXM MAXN*MAXN

/* FILL ME */
int n,m;
int adj[MAXN][MAXN], nadj[MAXN], from[MAXM], to[MAXM];

int mate[MAXN], first[MAXN], label[MAXN];
queue<int> q;

#define OUTER(x) (label[x] >= 0)

void L(int x, int y, int nxy) {
  int join, v, r = first[x], s = first[y];
  if (r == s) return;
  nxy += n + 1;
  label[r] = label[s] = -nxy;
  while (1) {
    if (s != 0) swap(r,s);
    r = first[label[mate[r]]];
    if (label[r] != -nxy) label[r] = -nxy;
    else {
      join = r;
      break;
    }
  }
  v = first[x];
  while (v != join) {
    if (!OUTER(v)) q.push(v);
    label[v] = nxy; first[v] = join;
    v = first[label[mate[v]]];
  }
  v = first[y];
  while (v != join) {
    if (!OUTER(v)) q.push(v);
    label[v] = nxy; first[v] = join;
    v = first[label[mate[v]]];
  }
  for (int i = 0; i <= n; i++) {
    if (OUTER(i) && OUTER(first[i])) first[i] = join;
  }
}

void R(int v, int w) {
  int t = mate[v]; mate[v] = w;
  if (mate[t] != v) return;
  if (label[v] >= 1 && label[v] <= n) {
    mate[t] = label[v];
    R(label[v],t);
    return;
  }
  int x = from[label[v]-n-1];
  int y = to[label[v]-n-1];
  R(x,y); R(y,x);
}

int E() {
  memset(mate,0,sizeof(mate));
```

```
  int r = 0;
  bool e7;
  for (int u = 1; u <= n; u++) {
    memset(label,-1,sizeof(label));
    while (!q.empty()) q.pop();
    if (mate[u]) continue;
    label[u] = first[u] = 0;
    q.push(u); e7 = false;
    while (!q.empty() && !e7) {
      int x = q.front(); q.pop();
      for (int i = 0; i < nadj[x]; i++) {
        int y = from[adj[x][i]];
        if (y == x) y = to[adj[x][i]];
        if (!mate[y] && y != u) {
          mate[y] = x; R(x,y);
          r++; e7 = true;
          break;
        }
        else if (OUTER(y)) L(x,y,adj[x][i]);
        else {
          int v = mate[y];
          if (!OUTER(v)) {
            label[v] = x; first[v] = y;
            q.push(v);
          }
        }
      }
    }
    label[0] = -1;
  }
  return r;
}
```

## 1.13 Floyd Warshall

```
Complexidade: O(n^3)
Descricao: Encontra o caminho
mínimo entre todos os pares de vértices

#include <cstdio>
#include <cstring>
using namespace std;

typedef long long int64;

#define MAXN 150
#define INF 0x3f3f3f3f

/* FIILL ME */
int adj[MAXN][MAXN]; // matriz de adj com os custos

int d[MAXN][MAXN];
int pai[MAXN][MAXN]; /* pai de j nos caminhos a partir de i */

void floyd(int n) {
  memset(d, 0x3f, sizeof(d));
  for (int i = 0; i < n; i++)
    for (int j = 0; j < n; j++)
      if (adj[i][j] < INF) {
        d[i][j] = adj[i][j];
        pai[i][j] = i;
      }

  for (int k = 0; k < n; k++)
    for (int i = 0; i < n; i++)
      for (int j = 0; j < n; j++)
        if (d[i][j] > d[i][k] + d[k][j]) {
          d[i][j] = d[i][k] + d[k][j];
          pai[i][j] = pai[k][j];
        }
}
```

## 1.14 Fluxo Máximo de Custo Mínimo (Uso Geral) - Corte Mínimo

```
Complexidade: O(m*Flow), em média, O(m*n*Flow) pior caso
Descricao: Calcula fluxo usando caminhos aumentantes usando
SPFA (um Bellman-Ford otimizado), suporta arestas múltiplas
e não direcionadas (usar add() 2x), usa lista de adjacências
eficiente em um único vetor e sem STL. Para max-cost, use ar-
estas de custo negativo (mas sem ciclo negativo). Se todos os
custos são iguais, o algoritmo equivale ao Edmonds Karp. Se
quiser obter o fluxo em cada aresta i, use re[2*i+1] e para
obter o residual use re[2*i]. Aresta i: ve[2*i+1] -> ve[2*i]

#include <algorithm>
using namespace std;

#define N 201
#define M (2*1010) // dobro do número de arestas
#define INF 0x3f3f3f3f

int vt, ve[M], re[M], ze[M], next[M];
int in[N], head[N], path[N], dis[N], qu[N], lim[N];

void init() {
  vt = 1;
  memset(head, 0, sizeof(head));
}

void add(int x, int y, int cap, int wei = 0) {
  // aresta x->y é armazenada em [vt+1] e [vt+2]
  ve[++vt] = y; re[vt] = cap; ze[vt] = wei;
  next[vt] = head[x]; head[x] = vt;
  ve[++vt] = x; re[vt] = 0; ze[vt] = -wei;
  next[vt] = head[y]; head[y] = vt;
}

int mfmc(int s, int t, int n, int &fcost) {
  int flow = fcost = 0;
  while (1) {
    int qt = 0, k = 0;
    qu[qt++] = s;
    for (int i = 0; i < n; i++)
      dis[i] = lim[i] = INF;
    dis[s] = 0;

    while (k != qt) {
      if (k == N) k = 0;
      int x = qu[k++];
      for (int i = head[x]; i; i = next[i]) // ve[i]: adjs de x
        if (re[i] && dis[x] + ze[i] < dis[ve[i]]){
          dis[ve[i]] = dis[x] + ze[i];
          path[ve[i]] = i;
          lim[ve[i]] = min(lim[x], re[i]);
          if (!in[ve[i]]) {
            if (qt == N) qt = 0;
            qu[qt++] = ve[i];
            in[ve[i]] = 1;
          }
        }
      in[x] = 0;
    }

    if (dis[t] == INF) break;
    int f = lim[t];
    for (int p = t; p != s; p = ve[path[p] ^ 1]) {
      re[path[p]] -= f; re[path[p] ^ 1] += f; // novo residual
    }
    fcost += f * dis[t];
    flow += f;
  }
  return flow;
}

/** Código abaixo apenas para Min Cut **/

/* mark[v] = true, se v esta no mesmo lado de u no corte */
bool mark[N];

void DFS(int u) {
  mark[u] = true;
  for (int i = head[u]; i; i = next[i]) {
    if (i % 2) continue;
    int v = ve[i];
    if (!mark[v] && re[i] > 0) {
      DFS(v);
    }
  }
 }

void mincut(int s, int t, int n) {
  memset(mark, 0, sizeof(mark));
  DFS(s);
  /* Arestas do corte */
  for (int i = 0; i < n; i++)
    for (int j = head[i]; j; j = next[j]) {
      if (j % 2) continue;
      int v = ve[j];
      if (mark[i] && !mark[v])
        printf("%d %d\n", i, v);
    }
}
```

## 1.15 Fluxo Mínimo

```
Complexidade: O(m*Flow)
Descricao: Calcula o fluxo mínimo viável entre s e t,
onde cada aresta e do grafo tem um capacidade mínima lb[e] e
máxima ub[e], de forma que o fluxo: lb[e] <= f[e] <= ub[e].
```

```
#include <algorithm>
#include <cstring>
using namespace std;

/* Fluxo Máximo SPFA AQUI */

/* FILL ME */
int lb[N][N], ub[N][N];

int f[N][N];

int minflow(int n, int s, int t) {
  lb[t][s] = 0;
  ub[t][s] = INF;
  init();
  for (int i = 0; i < n; i++)
    for (int j = 0; j < n; j++)
      if (ub[i][j])
        add(i, j, ub[i][j] - lb[i][j], 0);

  for (int i = 0; i < n; i++) {
    int b = 0;
    for (int j = 0; j < n; j++) {
      b += lb[j][i];
      b -= lb[i][j];
    }
    if (b >= 0)
      add(n, i, b);
    else
      add(i, n+1, -b);
  }
  int fcost;
  mfmc(n, n+1, n+2, fcost);

  for (int i = head[n]; i; i = next[i]) {
    if (i & 1) continue;
    if (re[i] > 0) return -1;
  }

  for (int i = 0; i < n; i++)
    for (int j = head[i]; j; j = next[j]) {
      if (j & 1 || ve[j] != n+1) continue;
      if (re[j] > 0) return -1;
    }

  memcpy(f, lb, sizeof(lb));
  mfmc(t, s, n, fcost);
  int res = 0;
  for (int i = 0; i < n; i++)
    for (int j = head[i]; j; j = next[j]) {
      if (j & 1) continue;
      int v = ve[j];
      f[i][v] += re[j+1];
      if (i == s) res += f[s][v];
    }
  return res;
}
```

## 1.16 Pontes, Pontos de Articulação e Componentes Biconexas

```
Complexidade: O(n+m)
Descricao: Encontra as pontes, os pontos de articulacao e as
componentes biconexas (comecando em 1) em um grafo
não direcionado. Não permite arestas múltiplas.

#include <cstring>
#include <stack>
#include <algorithm>
#include <vector>
using namespace std;

#define MAXN 1024
#define MAXM 1024*1024

#define VIZ(u, i) (orig[inc[u][i]] != (u) ? \
                   orig[inc[u][i]] : dest[inc[u][i]])

/* Input - FILL ME */
/* aresta i = (u, v); orig[i] = u, dest[i] = v,
   inc[u].push_back(i), inc[v].push_back(i) */
vector<int> inc[MAXN];
int orig[MAXM], dest[MAXM];

/* Output */
int ponte[MAXM]; // ponte[i] -> aresta i é ponte
int part[MAXN]; // part[u] -> vertice u é de articulação
int ncomp; // numero de componentes biconexas
int comp[MAXM]; // comp[i] = componente da aresta i

/* Variaveis auxiliares */
int low[MAXN], vis[MAXN], dt;
stack<int> stck;

/* Função auxiliar */
int dfsbcc(int u, int p) {
  int ch = 0;
  vis[u] = dt++;
  low[u] = vis[u];
  for (int i = 0; i < (int) inc[u].size(); i++) {
    int e = inc[u][i], v = VIZ(u, i);
    if (!vis[v]) {
      stck.push(e);
      dfsbcc(v, u); ch++;
      low[u] = min(low[u], low[v]);
      if (low[v] >= vis[u]) {
        part[u] = 1;
        ncomp++;
        while (stck.top() != e) {
          comp[stck.top()] = ncomp;
          stck.pop();
        }
        comp[stck.top()] = ncomp; stck.pop();
      }
      if (low[v] == vis[v]) ponte[e] = 1;
    } else if (v != p) {
      if (vis[v] < vis[u]) stck.push(e);
      low[u] = min(low[u], vis[v]);
    }
  }
  return ch;
}

void bcc(int n) {
  memset(low, 0, sizeof(low));
  memset(vis, 0, sizeof(vis));
  memset(part, 0, sizeof(part));
  memset(ponte, 0, sizeof(ponte));
  memset(comp, 0, sizeof(comp));
  dt = 1;
  ncomp = 0;
  for (int i = 0; i < n; i++)
    if (!vis[i])
      part[i] = dfsbcc(i, -1) >= 2;
}

#include <cstdio>

int main(){
  int i;
  int n, m;
  scanf("%d%d", &n, &m);

  for (int i = 0; i < m; i++) inc[i].clear();
  for (i = 0; i < m; i++) {
    int u, v;
    scanf("%d%d", &u, &v);
    orig[i] = u; dest[i] = v;
    inc[u].push_back(i);
    inc[v].push_back(i);
  }

  bcc(n);

  printf("Pontos de Articulacao:");
  for (i = 0; i < n; i++)
    if (part[i]) printf(" %d", i);
  printf("\n");

  printf("Pontes:");
  for (i = 0; i < m; i++)
      if (ponte[i])
        printf(" (%d %d)", orig[i], dest[i]);
  printf("\n");

  printf("Componentes:\n");
  for (i = 0; i < m; i++)
    printf("comp[%d] = %d\n", i, comp[i]);
  printf("\n");

  return 0;
}
```

## 1.17 Stable Marriage

```
Complexidade: O(m^2), m = numero de homens
Descricao:
Takes a set of m men and n women, where each person has
```

```
an integer preference for each of the persons of opposite
sex. Produces a matching of each man to some woman.
The matching will have the following properties:
- Each man is assigned a different woman
(n must be at least m).
- No two couples M1W1 and M2W2 will be unstable.
- The solution is man-optimal.
Two couples are unstable if
- M1 prefers W2 over W1 and
- W1 prefers M2 over M1.

#include <cstring>

#define MAXM 1024
#define MAXN 1024

int m, n; // number of men and women, n>=m
// the list of women in order of decreasing preference (man i)
int L[MAXM][MAXN];
int R[MAXN][MAXM]; // the attractiveness of man i to woman j
int L2R[MAXM]; // the mate of man i (always between 0 and n-1)
int R2L[MAXN]; // the mate of woman j (or -1 if single)

int p[MAXM];

void stableMarriage() {
  memset( R2L, -1, sizeof( R2L ) );
  memset( p, 0, sizeof( p ) );

  // Each man proposes...
  for( int i = 0; i < m; i++ ) {
    int man = i;
    while( man >= 0 ) {
      // to the next woman on his list in order
      // of decreasing preference, until one of them accepts;
      int wom;
      while( 1 ) {
        wom = L[man][p[man]++];
        if (R2L[wom] < 0 || R[wom][man] > R[wom][R2L[wom]])
          break;
      }

      // Remember the old husband of wom.
      int hubby = R2L[wom];

      // Marry man and wom.
      R2L[L2R[man] = wom] = man;

      // If a guy was dumped in the process, remarry him now.
      man = hubby;
    }
  }
}
```

## 1.18 Topological Sort

```
Complexidade: O(E + V)
Descricao: Ordena Topologicamente, ou verifica que não há
ordenação topológica. Para verificação de existência da
ordenação, implemente os trechos comentados do código;
além disso, as duas funções podem ser declaradas como void.

int n;
int adj[MAXN][MAXN]; /* lista de adj */
int nadj[MAXN]; /* grau de cada vertice */

int foi[MAXN], ip; /* auxiliar */
/* int foi2[MAXN]; */
int tops[MAXN]; /* resposta */

int DFS(int k) {
  int i,j;

  foi[k] = /* foi2[k] = */ 1;
  for(j=0 ; j<nadj[k] ; j++) {
    i = adj[k][j];
    /* if(foi2[i]) return 0; */
    if(!foi[i] && !DFS(i)) return 0;
  }

  tops[--ip] = k;
  /* foi2[k] = 0; */

  return 1;
}

/*
popular n: numero de vertices
apos chamar ord_top() "tops" tera a solucao
*/
int ord_top() {
  memset(foi, 0, n*sizeof(int));
  /* memset(foi2, 0, n*sizeof(int)); */
  ip = n;

  for(int i=0 ; i<n ; i++)
    if(!foi[i] && !DFS(i)) return 0;

  return 1;
}
```

## 1.19 Two Satisfiability

```
Complexidade: O(E + V)
Descricao:
Determina se existe uma atribuicao
que satisfaca a expressao (Xi v Xk)^(Xj v !Xl)...
Para cada clausula deve haver uma aresta
no grafo da forma !Xi -> Xk e !Xk -> Xi
A funcao "clau" gera as arestas automaticamente
dada a clausula, ver exemplo.

#define N(x) (2*x + 1)
#define Y(x) (2*x)
#define NEG(x) (x%2 == 1 ? x-1 : x+1)

/*n deve ser o numero total de literais p (nao 2*p))*/
bool two_sat(int n) {
  n *= 2;
  bool ok = true;
  scc(n);
  for (int i=0; i<n/2 && ok; i++)
    ok &= (comp[2*i] != comp[2*i+1]);
  return ok;
}

/*Solucao da literal x apos rodar two_sat() == true */
int getsol(int x) {
  return comp[2*x] < comp[2*x+1];
}

/*Insira a clausula como descrito na main*/
void clau(int x, int y) {
  int negx = NEG(x), negy = NEG(y);
  adj[negx][nadj[negx]++] = y;
  adj[negy][nadj[negy]++] = x;
}
```

## 1.20 Union Find e Árvore Geradora Mínima (Kruskal)

```
Complexidade: Union-Find: O(1), Kruskal: O(m*lgn)
Descricao: Estrutura de dados union find e uma aplicação
para encontrar a árvore geradora mínima

#include <algorithm>
using namespace std;

#define MAXN 1010

int id[MAXN], sz[MAXN]; //uf auxiliar

void ufinit(int n) {
  for (int i = 0; i < n; i++)
    id[i] = i, sz[i] = 1;
}

int uffind(int i) {
  if (i == id[i]) return i;
  return id[i] = uffind(id[i]);
}

void ufunion(int v, int w) {
  v = uffind(v); w = uffind(w);
  if (v == w) return;
  if (sz[v] > sz[w]) swap(v,w);
  id[v] = w;
  if (sz[v] == sz[w]) sz[w]++;
}

/* MST - Kruskal a partir daqui */
#define MAXM 10010

/* FILL ME */
struct edge {
  int u, v, w;
  int ind;
} ed[MAXM];

bool comp(edge a, edge b) {
```

```
  return a.w < b.w;
}

/* Para cada aresta i,
diz se está ou não na árvore */
bool used[MAXM];

int kruskal(int n, int m) {
  sort(ed, ed+m, comp);
  ufinit(n);
  int res = 0;
  for (int i = 0; i < m; i++) {
    int u = uffind(ed[i].u);
    int v = uffind(ed[i].v);
    if (u == v) {
      used[ed[i].ind] = false;
      continue;
    }
    used[ed[i].ind] = true;
    ufunion(u, v);
    res += ed[i].w;
  }
  return res;
}
```

# 2 Geométricos

## 2.1 Algoritmos Básicos para Circunferência

```
Descricao: Calcula interseção entre duas circunferência,
e a aréa dessa intersecção.

/**
Estrutura de ponto, circunferencia e reta aqui
**/

#include <cmath>
#include <algorithm>
#define F first
#define S second
using namespace std;

double sqr(double x) { return x*x; }

/* retorna 0, 1 ou 2 intersecções entre dois círculos e
 * se houver 1, em ia; se houver 2, em ia e ib */
int inter_circ(pt &ia, pt &ib, circ c1, circ c2) {
  int c = cmp(norma(c1.F-c2.F), c1.S+c2.S);
  if (c > 0) return 0;

  double dx=c2.F.x-c1.F.x;
  double dy=c2.F.y-c1.F.y;
  double d=sqr(dx)+sqr(dy);
  double r=sqrt((sqr(c1.S+c2.S)-d)*(d-sqr(c2.S-c1.S)));

  ia.x=ib.x=0.5*((c2.F.x+c1.F.x)+dx*(sqr(c1.S)-sqr(c2.S))/d);
  ia.y=ib.y=0.5*((c2.F.y+c1.F.y)+dy*(sqr(c1.S)-sqr(c2.S))/d);
  ia.x+=dy*r/(2*d); ib.x-=dy*r/(2*d);
  ia.y-=dx*r/(2*d); ib.y+=dx*r/(2*d);
  return 1-c;
}

/* Calcula a menor área quando o círculo é divido pela
corda dos pontos (a, b) */
double area_corda(circ c, pt a, pt b) {
  double d = norma(a - b);
  double r = c.S;
  double h = sqrt(r*r - (d*d)/4.0);
  double at = (d*h) / 2.0;
  double ang = 2 * acos(h / r);
  double ac = (ang * r * r) / 2.0;
  return ac - at;
}

/* Calcula a area da intersecção entre dois círculos */
double area_inter(circ c1, circ c2) {
  if (c1.S > c2.S) swap(c1, c2);
  c2.F.x = norma(c1.F - c2.F); c2.F.y = 0;
  c1.F.x = 0; c1.F.y = 0;
  pt ia, ib;
  int it = inter_circ(ia, ib, c1, c2);
  if (it == 1) return 0.0;
  if (it == 0) {
    if (cmp(c2.F.x, c1.S + c2.S) > 0) return 0.0;
    else return  pi * c1.S * c1.S;
  }

  double a1 = area_corda(c1, ia, ib);
  double a2 = area_corda(c2, ia, ib);
  if (ccw(ia, ib, c1.F) == ccw(ia, ib, c2.F)) {
    return a2 + pi * c1.S * c1.S - a1;
  }
  return a1 + a2;
}

/* Exemplo simples de uso */
int main() {
  circ c1, c2;
  pt ia, ib;
  int res;

  c1 = circ(pt(0, 0), 1);
  c2 = circ(pt(0, 2), 2);
  res = inter_circ(ia, ib, c1, c2);
  printf("%d (%lf %lf) (%lf %lf) %lf\n", res, ia.x, ia.y,
         ib.x, ib.y, area_inter(c1, c2));

  return 0;
}
```

## 2.2 Algoritmos Básicos para Geométricos

```
Descricao: Contem algoritmos simples para geometricos

/**
Estrutura de ponto e poligono aqui
**/

double polyarea(poly& p){ /* area com sinal */
  int i, n=p.size();
  double area = 0.0;

  for(i=0 ; i<n ; i++)
    area += p[i]%p[(i+1)%n];

  return area/2.0; /* area>0 = ccw ; area<0 = cw */
}

/* ponto p entre segmento [qr] */
int between3(pt p, pt q, pt r){
  if(cmp((q-p)%(r-p)) == 0) /* colinear */
    if(cmp((q-p)*(r-p)) <= 0) /* < para nao contar extremos */
      return 1;

  return 0;
}

/* rotaciona pt p em ang radianos, em torno do ponto q
   se q nao especificado, rotaciona em torno da origem */
pt rotate(pt p, double ang, pt q = pt(0,0)) {
  double s = sin(ang), c = cos(ang);
  p = p-q;
  return q + pt( p.x*c - p.y*s, p.x*s + p.y*c );
}
```

## 2.3 Algoritmos de Intersecções

```
- UVa 11068 [intersect] [acha] t=0.010s
- UVa 866 [intersect_seg] [intersect_seg_2] [acha] t=0.000s
- UVa 378 [intersect] [acha] t=0.010s
- UVa 191 [intersect_seg] [intersect_seg_2] t=0.000s
- POJ 3819 [inter_reta_circ]
- comparacoes na estrutura de ponto - soh intersect_seg()
- norma() - distPR(), inter_reta_circ()
- projecao() - distPR(), inter_reta_circ()
- between3() - distPR(), intersect_seg_2(), inter_reta_circ()
- ccw() - soh intersect_seg_2()
Descricao: Determina se há intersecção ou o ponto de
intersecção entre segmentos de reta ou retas. Acha inter-
secções entre segmento de reta ou reta e circunferência.
Também contém função que devolve a distância de um ponto
a uma reta.

/**
Estruturas aqui
**/

int intersect(reta p0, reta q0){ /*intersecção de retas*/
  eq_reta p(p0), q(q0);

  if(cmp(p.A*q.B , p.B*q.A)==0){ /*paralelos*/
    if(cmp(p.A*q.C , p.C*q.A)==0 &&
       cmp(p.B*q.C , p.C*q.B)==0) return 2; /*reta*/
    else return 0; /*nada*/
  }
  return 1; /*ponto*/
}

/* intersecção nos extremos dos segmentos tbm é contada! */
bool intersect_seg(pt p, pt q, pt r, pt s) {
```

```
  pt A = q - p, B = s - r, C = r - p, D = s - q;
  int a = cmp(A % C) + 2 * cmp(A % D);
  int b = cmp(B % C) + 2 * cmp(B % D);
  if (a == 3 || a == -3 || b == 3 || b == -3) return false;
  if (a || b || p==r || p==s || q==r || q==s) return true;
  int t = (p < r) + (p < s) + (q < r) + (q < s);
  return t != 0 && t != 4;
}

bool intersect_seg_2(pt p, pt q, pt r, pt s) {
  int a = ccw(p,q,r)*ccw(p,q,s);
  int b = ccw(r,s,p)*ccw(r,s,q);

  if(a<0 && b<0) return true;
  else return false;

  // tire o 'else' para verificar intersecção nos extremos
  if(a>0 || b>0) return false;

  return ( between3(p,r,s) ||
           between3(q,r,s) ||
           between3(r,p,q) ||
           between3(s,p,q) );
}

/*acha intersecção de duas retas*/
pt acha(pt a, pt b, pt c, pt d){
  /* pressupoe que haja intersecção! */
  eq_reta p(reta(a,b)), q(reta(c,d));
  pt k;

  k.x = (q.C*p.B - p.C*q.B)/(p.A*q.B - q.A*p.B);
  k.y = (q.C*p.A - p.C*q.A)/(p.B*q.A - q.B*p.A);
  return k;
}

/*acha intersecção de duas retas - da PUC*/
pt acha_(pt p, pt q, pt r, pt s){
  pt a = q-p, b = s-r, c = pt(p%q,r%s);
  return pt(pt(a.x, b.x)%c, pt(a.y, b.y)%c) / (a%b);
}

/* distância de um ponto a uma reta */
double distPR(pt p, reta r){
  pt v = p - r.ini;
  pt w = r.fim - r.ini;

  pt proj = projecao(v,w);
  /* (proj+r.ini) é o ponto mais proximo de p,
     e que pertence à reta r */

  /* para segmentos de reta
   *
   * if( !between3(proj+r.ini, r.ini, r.fim) )
   *   return min( norma(p-r.ini), norma(p-r.fim) );
   */

  return norma(v - proj);
}
```

```
/* retorna 0, 1 ou 2 intersecções entre segmento/reta e
 * círculo: se houver 1, em ia; se houver 2, em ia e ib */
int inter_reta_circ(pt &ia, pt &ib, reta r, circ c) {
  pt p = r.ini + projecao(c.first - r.ini, r.fim - r.ini);
  double d = norma(p - c.first);

  if (cmp(d, c.second) > 0) return 0;

  pt v = cmp(norma(r.ini - p)) ? r.ini : r.fim;
  v = versor(v - p) * sqrt(max(0.0,c.second*c.second - d*d));

  ia = p + v; ib = p - v;

  /* para segmentos de reta, descomente
   * int ba = between3(ia, r.ini, r.fim);
   * int bb = between3(ib, r.ini, r.fim);
   * if (!ba) {
   *   ia = ib;
   *   return bb;
   * }
   */
  return (cmp(norma(ia - ib))/* && bb*/) + 1;
}
```

## 2.4 Círculo Gerador Mínimo

```
Complexidade: O(n^3)
- SPOJbr ICPC (n<=100) t=0.35s
- UVa 10005 (n<=100) t=0.002s
- norma()
Descricao: O algoritmo devolve o circulo de raio minimo que
contem todos os pontos dados

bool in_circle(circ C, pt p){
  return cmp(norma(p - C.first), C.second) <= 0;
}

pt circumcenter(pt p, pt q, pt r) {
  pt a = p - r, b = q - r,
    c = pt(a * (p + r) / 2, b * (q + r) / 2);

  return pt(c % pt(a.y, b.y),pt(a.x, b.x)%c) / (a%b);
}

circ spanning_circle(vector<pt>& T) {
  int n = T.size();

  circ C(pt(), -INFINITY);
  for (int i = 0; i < n; i++) if (!in_circle(C, T[i])) {
    C = circ(T[i], 0);
    for (int j = 0; j < i; j++) if (!in_circle(C, T[j])) {
      C = circ((T[i] + T[j]) / 2, norma(T[i] - T[j]) / 2);
      for (int k = 0; k < j; k++) if (!in_circle(C, T[k])) {
        pt o = circumcenter(T[i], T[j], T[k]);
        C = circ(o, norma(o - T[k]));
      }
    }
  }

  return C;
}
```

## 2.5 Convex Hull (Graham Scan)

```
Complexidade: O(n*lg(n))
- norma()
- ccw()
Descricao: Algoritmo de Graham, para obter o Convex Hull de um
dado conjunto de pontos

/**
Estrutura de ponto e poligono aqui
**/

pt pivo;
/* ordena em sentido horario */
bool cmp_radial(pt a, pt b){
  int aux = ccw(pivo, a,b);
  return ((aux<0) || (aux==0 && norma(a-pivo)<norma(b-pivo)));
}
bool cmp_pivo(pt p, pt q){ /* pega o de menor x e y */
  int aux = cmp( p.x, q.x );
  return ((aux<0) || (aux==0 && cmp( p.y, q.y )<0));
}
/* usar poly& p reduz tempo, mas desordena o conj de pontos */
poly graham(poly p){
  int i,j,n = p.size();
  poly g;

  /* ordena e torna o conj de pontos um poligono estrelado */
  pivo = *min_element(p.begin(), p.end(), cmp_pivo);
  sort(p.begin(), p.end(), cmp_radial);

  /* para pegar colineares do final do poligono
   *
   * for(i=n-2 ; i>=0 && ccw(p[0], p[i], p[n-1])==0 ; i--);
   * reverse(p.begin()+i+1, p.end());
   */

  for(i=j=0 ; i<n ; i++) {
    /* trocar ccw>=0 por ccw>0 para pegar colineares */
    while( j>=2 && ccw(g[j-2], g[j-1], p[i]) >= 0 ){
      g.pop_back(); j--;
    }
    g.push_back(p[i]); j++;
  }

  return g;
}
```

## 2.6 Diâmetro de Pontos e Polígono

```
Complexidade: O(n) polígono convexo, O(n lg n) pontos
Descrição: Calcula a maior distância entre um par de pontos
de um polígono ou de um conjunto de pontos em posição geral

#include <cmath>
#include <vector>
#include <algorithm>
using namespace std;
```

```
typedef pair<int,int> pii;

int cmpa(poly &p, int i, int j) {
  int n = p.size();
  return cmp(triarea(p[i], p[(i+1) % n], p[(j+1) % n]),
    triarea(p[i], p[(i+1) % n], p[j]));
}

/* Retorna os O(n) pares de vértices de um polígono convexo
pelos quais passam um par de retas de suporte paralelas. O
polígono deve ser anti-horário e pode ter pontos colineares
*/
vector<pii> antipodals(poly &p) {
  int n = p.size();
  int i = n - 1, j;

  for (j = 0; cmpa(p, i, j) >= 0; j++) {}

  vector<pii> res(1, pii(i, j));
  int k = j;
  while (j) {
    i = (i+1) % n;
    res.push_back(pii(i, j));
    while (j && cmpa(p, i, j) >= 0) {
      j = (j+1) % n;
      if (i != k || j != 0)
        res.push_back(pii(i, j));
    }
    if (!cmpa(p, i, j) && (i != k || j != n-1))
      res.push_back(pii(i, (j+1) % n));
  }
  return res;
}

double diam_convex(poly p) { // p em sentido horário
  double res = 0;
  reverse(p.begin(),p.end());

  vector<pii> c = antipodals(p);
  for (int i = 0; i < c.size(); i++)
    res = max(res, norma(p[c[i].first]-p[c[i].second]));

  return res;
}

double diam_points(poly &p) {
  if (p.size() <= 1) return 0;
  if (p.size() == 2) return norma(p[0] - p[1]);
  return diam_convex(graham(p));
}
```

## 2.7 Distância Esférica

```
Complexidade: O(1)
Descricao: Calcula a distância entre 2 pontos em uma esfera

#include <cmath>

double torad;
double r = 6378;

struct geo {
  double lat, lon;
   geo(double lat1 = 0.0, double lon1 = 0.0) {
    lat = lat1 * torad;
    lon = lon1 * torad;
}};

double geoDist(geo a, geo b)
{
  return acos(sin(a.lat) * sin(b.lat) +
         cos(a.lat)*cos(b.lat)*cos(fabs(a.lon - b.lon)))*r;
}
```

## 2.8 Estrutura e Base para Geométricos

```
Descricao: Contem estrutura de ponto, reta, poligono e algumas
operacoes-base para os algoritmos geometricos

#include <cmath>
#include <vector>

using namespace std;

const double pi = acos(-1);

int cmp(double a, double b = 0){
  if (fabs(a-b)<1e-8) return 0;
  if (a<b) return -1;
  return 1;
}

struct pt {
  double x,y;
  explicit pt(double x = 0, double y = 0): x(x), y(y) {}

  pt operator +(pt q){ return pt(x + q.x, y + q.y); }
  pt operator -(pt q){ return pt(x - q.x, y - q.y); }
  pt operator *(double t){ return pt(x * t, y * t); }
  pt operator /(double t){ return pt(x / t, y / t); }
  double operator *(pt q){ return x * q.x + y * q.y; }
  double operator %(pt q){ return x * q.y - y * q.x; }

  int cmp(pt q) const {
    if (int t = ::cmp(x, q.x)) return t;
    return ::cmp(y, q.y);
  }
  bool operator ==(pt q) const { return cmp(q) == 0; }
  bool operator !=(pt q) const { return cmp(q) != 0; }
  bool operator < (pt q) const { return cmp(q) < 0; }
};

struct reta {
  pt ini,fim;
  reta(){}
  reta(pt ini, pt fim): ini(ini), fim(fim) {}
};

struct eq_reta {
  double A,B,C;  /* Ax + By + C = 0 */

  void init(reta p){
    pt aux = p.ini - p.fim;
    A = aux.y;
    B = -aux.x;
    C = -A*p.ini.x - B*p.ini.y;
  }
  eq_reta(reta p){ init(p); }
};

typedef vector<pt> poly;
typedef pair<pt,double> circ;

pt normal(pt v){ return pt(-v.y,v.x); }
double norma(pt v){ return hypot(v.x, v.y); }
pt versor(pt v){ return v/norma(v); }
double anglex(pt v){ return atan2(v.y, v.x); }
double angle(pt v1, pt v2){ /* angulo orientado ccw */
  return atan2(v1%v2 , v1*v2);
}
double triarea(pt a, pt b, pt c){ /* area c/ sinal */
  return ((b-a)%(c-a))/2.0; /* area>0 = ccw ; area<0 = cw */
}
int ccw(pt a, pt b, pt c){ /* b-a em relacao a c-a */
  return cmp((b-a)%(c-a)); /* ccw=1 ; cw=-1 ; colinear=0 */
  /* equivalente a cmp(triarea(a,b,c)), mas evita divisao */
}
pt projecao(pt v, pt w){ /* proj de v em w */
  double alfa = (v*w)/(w*w);
  return w*alfa;
}
```

## 2.9 Intersecção de Polígonos Convexos

```
Complexidade: O(n+m)
- ccw()
- between3()
- intersect_seg()
- acha()
- inpoly()
Descricao: O algoritmo devolve a interseccao de dois poligonos
convexos, orientados em sentido anti-horario.
Pode ser utilizado inpoly_convex(), sem verificacao de ponto
na borda do poligono, ja que os poligonos sao convexos.

#define all(x) (x).begin(),(x).end()

/* os poligonos P e Q devem estar orientados em
   sentido anti-horario! */
poly poly_intersect(poly& P, poly& Q) {
  int m = Q.size(), n = P.size();
  int a = 0, b = 0, aa = 0, ba = 0, inflag = 0;
  poly R;
  while ((aa < n || ba < m) && aa < 2*n && ba < 2*m) {
    pt p1 = P[a], p2 = P[(a+1) % n],
      q1 = Q[b], q2 = Q[(b+1) % m];
    pt A = p2 - p1, B = q2 - q1;
    int cross = cmp(A % B), ha = ccw(p2, q2, p1),
      hb = ccw(q2, p2, q1);
```

```
    if (cross == 0 && ccw(p1, q1, p2) == 0 && cmp(A*B) < 0) {
      if (between3(p1, q1, p2)) R.push_back(q1);
      if (between3(p1, q2, p2)) R.push_back(q2);
      if (between3(q1, p1, q2)) R.push_back(p1);
      if (between3(q1, p2, q2)) R.push_back(p2);
      if (R.size() < 2) return poly();
      inflag = 1; break;
    } else if (cross != 0 && intersect_seg(p1, p2, q1, q2)) {
      if (inflag == 0) aa = ba = 0;
      R.push_back(acha(p1, p2, q1, q2));
      inflag = (hb > 0) ? 1 : -1;
    }
    if (cross == 0 && hb < 0 && ha < 0) return R;
    bool t = cross == 0 && hb == 0 && ha == 0;
    if (t ? (inflag==1) : (cross>=0) ? (ha<=0) : (hb>0)) {
      if (inflag == -1) R.push_back(q2);
      ba++; b++; b %= m;
    } else {
      if (inflag == 1) R.push_back(p2);
      aa++; a++; a %= n;
    }
  }
  if (inflag == 0) {
    if (inpoly(P[0], Q)) return P;
    if (inpoly(Q[0], P)) return Q;
  }
  R.erase(unique(all(R)), R.end());
  if (R.size() > 1 && R.front() == R.back()) R.pop_back();
  return R;
}
```

### 2.10 Par de Pontos Mais Próximos

```
Complexidade: O(n*lg(n))
- UVa 10245 (n<=10000) t=0.290s
- norma()
Descricao: Obtem a menor distancia entre pontos de um conjunto
de pontos. Eh preciso que o conjunto contenha pelo menos 2
pontos para o algoritmo funcionar

#include <set>
#define foreach(it, a,b) for(typeof(a)it=(a) ; it!=(b) ; it++)
#define all(x) (x).begin(), (x).end()

bool ycmp(pt a, pt b) {
  if (a.y!=b.y) return a.y<b.y;
  return a.x<b.x;
}
double closest_pair (poly &P) {
  int n = P.size();
  double d = norma(P[0]-P[1]);
  set<pt, bool(*)(pt,pt)> s(&ycmp);

  sort(all(P));
  for(int i=0,j=0 ; i<n ; i++) {
    pt lower(0, P[i].y - d) , upper(0, P[i].y + d);
    while(P[i].x - P[j].x > d)
      s.erase(P[j++]);

    foreach(p, s.lower_bound(lower), s.upper_bound(upper))
      /* os pontos mais proximos sao tirados de P[i] e *p */
      d = min(d, norma(P[i] - *p));
    s.insert(P[i]);
  }
  return d;
}
```

### 2.11 Verificações de Ponto em Polígono

```
Complexidade: O(n), O(lg(n))
- inpoly(): uva.634
- inpoly_convex(): testes gerados na mão
- ccw()
- between3()
- intri() - soh inpoly_convex()

/**
Estrutura de ponto e poligono aqui
**/

int intri(pt k, pt a, pt b, pt c){
  int a1,a2,a3;

  a1 = ccw(a,k,b);
  a2 = ccw(b,k,c);
  a3 = ccw(c,k,a);

  if((a1*a2)>0 && (a2*a3)>0) return 1; /*dentro*/
  if(between3(k,a,b) || between3(k,b,c) || between3(k,c,a))
    return 2; /*borda*/
  return 0; /*fora*/
}

int inpoly(pt k, poly &p){
  int n = p.size();
  int cross = 0;

  for(int i=1; i<=n ; i++) {
    pt q=p[i-1], r=p[i%n];

    if( between3(k,q,r) ) return 2;
    if( q.y>r.y ) swap(q,r);
    if( q.y<k.y && r.y>=k.y  && ccw(k,q,r)>0 ) cross++;
  }

  return cross%2;
}

/* O(lg(n)) - só para polígonos convexos */
int inpoly_convex(pt k, poly& p){
  /* 'val' indica o sentido do polígono */
  int val = ccw(p[0],p[1],p[2]);
  /* tomar cuidado para o caso em que o polígono
     comeca com pontos colineares, 'val' receberá 0 */

  int esq,dir,meio, n = p.size();

  esq = 1; dir = n-1;
  while(dir>esq+1) {
    meio = (esq+dir)/2;
    if(ccw(p[0],p[meio],k) == val) esq = meio;
    else dir = meio;
  }

  return intri(k, p[0],p[esq],p[dir]);

  /* caso seja preciso verificar se está na borda,
   * substituir o return por:
   *
   * if(between3(k,p[esq],p[dir]) ||
   *    between3(k,p[0],p[1])     ||
   *    between3(k,p[0],p[n-1])) return 2;  //BORDA
   * return intri(k, p[0],p[esq],p[dir])?1:0;  //DENTRO:FORA
   */
}
```

## 3 Numéricos

### 3.1 Binomial Modular (e não modular)

```
Complexidade: binomial: O(n)*O(lg MOD);  binomial_: O(n)
Descricao: binomial(n,k,M) calcula o binomial C(n,k)%M sem
overflow, utilizando inverso modular;  binomial_(n,k) deter-
mina qualquer C(n,k) que caiba em um int (menor que 2^31-1)

#define MOD 1300031
typedef long long int64;

int64 fat(int64 n, int64 M = MOD) {
  int64 i, fat = 1;
  for(i=2 ; i<=n ; i++)
    fat = (fat*i)%M;
  return fat;
}

int64 binomial(int64 n, int64 k, int64 M = MOD) {
  int64 a = fat(n)*invmod(fat(k),M);
  int64 b = invmod(fat(n-k),M);
  return ( (a%M)*b )%M;
}

int64 binomial_(int n, int k) {
  if(n-k < k) k = n-k;
  if(k == 0) return 1;
  return (n-k+1)*binomial_(n,k-1)/k;
}
```

### 3.2 Crivo de Erastótenes

```
Complexidade: O(N log log N)
Descricao: Popula o array pr, de tal forma que pr[i] eh
verdadeiro se i eh primo.

#include <iostream>
#include <cstdio>

// Numero maximo a ser analisado
#define MAXN 1123123
```

```
// se pr[i] == true, i eh primo
bool pr[MAXN+1];

// algum divisor primo de i. Para fatoracao.
int divisor[MAXN+1];

// Analisa primalidade no intervalo [1,n]
void crivo(int n) {
  memset(pr, true, n * sizeof(bool));
  pr[0] = pr[1] = false;
  for(int i = 2; i*i <= n; i++){
    if( !pr[i] || !(i&1) && i > 2) continue;
    int k = i*i;
    divisor[i] = i;
    while(k <= n){
      pr[k] = false;
      divisor[k] = i;
      k += i;
    }
  }
}
```

### 3.3 Eliminação de Gauss

```
Complexidade: O(n^3)
Descricao: Calcula, se existir, a inversa mi da matriz
ma com pivoteamento, sendo inicialmente mi a identidade.
Pode ser usado colocando uma matriz-coluna de constantes
em mi para se obter a solução do sistema linear.

#include <cmath>
#include <algorithm>
using namespace std;
#define MAXN 100

double ma[MAXN][MAXN], mi[MAXN][MAXN];

bool invert(int n) {
  for (int k=0; k<n; k++) {
    int imax=k;
    for (int i=k+1; i<n; i++)
      if (fabs(ma[i][k]) > fabs(ma[imax][k])) imax=i;

    double p = ma[imax][k];
    if (fabs(p) < 1e-8) return false;

    for (int j=0; j<n; j++) {
      swap(ma[k][j], ma[imax][j]);
      swap(mi[k][j], mi[imax][j]);
      ma[k][j] /= p; mi[k][j] /= p;
    }

    for (int i=0; i<n; i++) {
      if (i == k) continue;
      double mul = ma[i][k];
      for (int j=0; j<n; j++) {
        ma[i][j] -= ma[k][j]*mul;
        mi[i][j] -= mi[k][j]*mul;
      }
    }
  }
  return true;
}
```

### 3.4 Estrutura de Polinômio

```
Descricao: Estrutura e operações com polinômios

#include <math.h>
#include <vector>
#include <algorithm>

using namespace std;

const double EPS = 1e-8;

typedef vector<double> vd;

struct polin {
  vd p; // expoentes decrescentes

  polin(){}
  polin(double x) { p.push_back(x); }
  polin(vd v): p(v) { fix(); }

  int grau() { return p.size()-1; }
  double& operator [](int i) { return p[i]; }

  void fix() {
    int i;
    for(i=0; i<=grau() && fabs(p[i])<EPS; i++) ;
    p.erase(p.begin(), p.begin()+i);
  }

  polin operator + (polin &q) {
    int k = q.grau() - grau();
    polin sum = (k>0)? q : p;

    for(int i=sum.grau(); i>=abs(k); i--)
        sum[i] += (k>0)? p[i-k] : q[i+k];

    sum.fix();
    return sum;
  }
  polin operator - (polin &q) {
    return q*(-1) + *this;
  }
  polin operator * (double x) {
    polin prod(p);
    for(int i=0; i<=grau(); i++)
      prod.p[i] *= x;
    return prod;
  }
  polin operator * (polin &q) {
    polin prod;

    for(int i=0; i<=grau(); i++) {
      polin aux = q * p[i];
      aux.p.resize(q.grau()+p.size()-i, 0);
      prod = prod + aux;
    }
    return prod;
  }
  pair<polin,polin> operator / (polin &d) {
    polin resto(p);
    polin q;
    int g = grau(), dg = d.grau();
    for(int i=0; i<=g-dg; i++) {
      double a = resto[i] / d[0];
      for(int j=0; j<=dg; j++)
        resto[i+j] -= a*d[j];
      q.p.push_back(a);
    }

    resto.fix();
    return make_pair(q, resto);
  }
  polin operator ~ () { // derivada
    polin dp;
    int g = grau();
    for(int i=0; i<g; i++)
      dp.p.push_back(p[i] * (g-i));
    return dp;
  }

  double eval(double x) {
    double res=0, pw=1;
    for (int i=grau(); i>=0; i--, pw*=x)
      res += pw*p[i];
    return res;
  }
};

polin mdc(polin &a, polin &b) {
  if(b.grau() == -1) return a;
  polin resto = (a/b).second;
  return mdc(b, resto);
}
```

### 3.5 Euclides Extendido

```
Complexidade: O(lg x)
Descricao: Calcula um par x,y tal que a*x+b*y=mdc(a,b)

#include <algorithm>
using namespace std;

typedef pair<int,int> pii;

pii mdc(int a, int b){
  if (b == 0) return pii(1,0);
  pii u = mdc (b,a%b);
  return pii(u.second, u.first - (a/b)*u.second);
}
```

### 3.6 Exponenciação modular rápida

```
Complexidade: O(log(b))
Descricao: Dado a, b e m, calcula a^b mod m
```

```
#include <cstdio>
#include <cstring>

using namespace std;

typedef long long int int64;

int64 expo(int64 a, int64 b, int64 m) {
  int64 y = a % m, x = 1;
  while (b > 0) {
    if (b % 2 == 1) {
      x = (x*y) % m;
    }
    y = (y*y) % m;
    b = b/2;
  }
  return x % m;
}
```

### 3.7 Inverso Modular

```
Complexidade: O(lg x)
Descricao: Calcula um x tal que a*x === 1 (mod M)
Para a e M coprimos, eh garantido que x eh unico
Nesse caso, pode ser usado para determinar
a divisao modular como exemplificado.

#include <algorithm>
using namespace std;

typedef pair<int,int> pii;

pii mdc(int a, int b){
  if (b == 0) return pii(1,0);
  pii u = mdc (b,a%b);
  return pii(u.second, u.first - (a/b)*u.second);
}

int invmod(int a, int M) {
  pii r=mdc(a,M);
  if (r.first * a + r.second * M == 1)
    return (r.first + M) % M;
  return 0;
}

#include <cstdio>
```

### 3.8 Log Discreto

```
Complexidade: O(sqrt(m)*lg m)
Descricao: Dados a, b e m (a e m devem ser coprimos),
calcula o menor x tal que a^x = b (mod m)

#include <cmath>
#include <map>
using namespace std;

int baby_giant(int a, int b, int m) {
  int n = ceil(sqrt(m));
  int an = 1;
  for (int i = 0; i < n; i++)
    an = (an * 1LL * a) % m;

  map<int, int> vals;
  for (int i = 0, cur = b; i <= n; i++) {
    vals[cur] = i;
    cur = (cur * 1LL * a) % m; // baby step
  }

  for (int i = 1, cur = an; i <= n; i++) {
    if (vals.count(cur)) {
      int ans = i*n - vals[cur];
      if (ans < m)
        return ans;
    }
    cur = (cur * 1LL * an) % m; // giant step
  }
  return -1;
}
```

### 3.9 Teorema Chinês do Resto

```
Complexidade: O(n * lg X)
Descricao: Resolve o conj de eqs: a[i]*x === b[i] (mod m[i])
para 0<=i<n com a restricao m[i]>1
Se a[i] == 1 para todo i, existe solucao sse
b[i]===b[j] (mod gcd(m[i],m[j])) para todo i e j

#include <algorithm>
#include <vector>
#define MAXN 1000
using namespace std;

int n,a[MAXN],b[MAXN],m[MAXN];
typedef pair<int,int> pii;

int chines() {
   int x = 0, M = 1;
   for (int i=0; i<n; i++) {
     int b2 = b[i] - a[i]*x;
     pii bizu = mdc(a[i]*M, m[i]);
     int g = a[i]*M * bizu.first + m[i] * bizu.second;

     if (b2 % g) return -1;
     x += M *(bizu.first * (b2/g) % (m[i]/g));
     M *= (m[i]/g);
   }
   return (x%M+M)%M;
}
```

## 4 Miscelânea

### 4.1 Árvore de Intervalos

```
Complexidade: O(lg n) por update/query
Descricao: Modelo de segtree que deve ser adaptado ao problema
desejado. Suporta query em intervalo e update em ponto. O código
abaixo é de RMQ, ou seja, dá o máximo elemento em um intervalo

#include <cstring>
#include <algorithm>
#define MAXN 100100
using namespace std;

int t[4*MAXN];

/* Obtém o RMQ em [a,b]; chamar com root=0, rl=0, rr=n-1 */
int query(int root, int rl, int rr, int a, int b) {
  if (a > b) return 0;
  if (rl==a && rr==b) return t[root];

  int rm = (rl+rr)/2;
  return max(query(2*root+1, rl, rm, a, min(b, rm)),
             query(2*root+2, rm+1, rr, max(a, rm+1), b));
}

/* Muda posição x para vx; chamar com root=0, rl=0, rr=n-1 */
void update(int root, int rl, int rr, int x, int vx) {
  if (rl==rr) t[root] = vx;
  else {
    int rm = (rl+rr)/2;
    if (x <= rm) update(2*root+1, rl, rm, x, vx);
    else update(2*root+2, rm+1, rr, x, vx);

    t[root] = max(t[2*root+1], t[2*root+2]);
  }
}
```

### 4.2 Árvore de Intervalos (c/ Lazy Propagation)

```
Complexidade: O(lg n) por update/query
Descricao: Modelo de segtree que deve ser adaptado ao problema
desejado. Suporta query e update em ponto ou intervalo. O código
abaixo representa um vetor de bits com update(a,b) sendo "toggle
bits entre a e b" e query(a,b) "quantos bits 1 entre a e b"

#include <algorithm>
#define MAXN 100100
using namespace std;

struct tr {
  int ql;   /* Qtd de bits 1 no intervalo */
  int sz;   /* Tam do intervalo */
  int prop; /* Qtd de updates a propagar nos filhos */
} tree[4*MAXN];

/* Aplica q vezes o update no nó t (q pode ser zero) */
void apply(tr &t, int q) {
  if (q % 2) t.ql = t.sz - t.ql;
}

/* Faz x updates (chamando acc=up=x) e retorna a query a
 * partir da sub-árvore root = [rl,rr] no intervalo [a,b]
 * Para obter apenas a query, use acc=up=0 */
int go(int root, int rl, int rr,
       int a, int b, int acc, int up) {
  /* acc = acúmulo de updates dos pais mais o up original */
  /* devemos aplicar acc updates na sub-árvore */
  tree[root].prop+=acc;
```

```
  if (a > b) { /* [a,b] não está nesse nó raiz */
    /* aplica na raiz e agenda para os filhos
     * apenas os updates dos pais (sem up) */
    tree[root].prop -= up;
    apply(tree[root], acc - up);
    return 0; /* elemento nulo */
  }
  if (rl == a && rr == b) { /* [a,b] == nó raiz */
    /* basta aplicar updates na raiz e devolvê-la
     * a propagação será feita posteriormente */
    apply(tree[root], acc);
    return tree[root].ql;
  }

  int rm = (rl + rr) / 2;
  int ls = 2*root + 1, rs = 2*root + 2;

  /* res = a combinacao das querys dos filhos (p ex soma) */
  int res = go(ls, rl, rm, a, min(b,rm), tree[root].prop, up)
  + go(rs, rm + 1, rr, max(a,rm+1), b, tree[root].prop, up);

  /* nova raiz é a combinação dos filhos atualizados */
  tree[root].ql = tree[ls].ql + tree[rs].ql;
  tree[root].prop=0; /* propagação feita na raiz */
  return res;
}

/* Inicializar árvore (ou pode usar memset se tudo for 0) */
void init(int root, int rl, int rr) {
  tree[root].ql = 0;
  tree[root].sz = rr-rl+1;
  tree[root].prop = 0;

  if (rl < rr) {
    int rm = (rl+rr) / 2;
    init(2*root+1, rl, rm);
    init(2*root+2, rm+1, rr);
  }
}
```

### 4.3 Binary Indexed Tree/Fenwick Tree

```
Complexidade: O(lg MAX) - atualizacao e consulta
Descricao: Dada um vetor inicial vazio, eh possivel fazer
incremento no conteudo v[x] e consultar a soma do
subvetor v[0..x] de forma eficiente. Para o calculo
da soma de um subvetor qualquer usa-se a relacao
sum(a..b) =  sum(1..b) - sum(1..a-1)

Observacao: Zerar o vetor tree[] antes de utilizar

#define MAXN 1000

int tree[MAXN+1];

int query(int x) {
  int sum=0;

  for (x++; x>0; x-=x & (-x))
    sum+=tree[x];
  return sum;
}

/*by representa um inc/decremento em x*/
void update(int x, int by) {
  if (x<0) return;

  for (x++; x<=MAXN; x+=x & (-x))
    tree[x]+=by;
}
```

### 4.4 Convex Hull Trick

```
Complexidade: O(lg n) - insert e query
Descrição: Estrutura de dados que permite inserir retas
não-verticais na forma y(x)=A*x+B e consultar, dado x*,
qual é o maior y(x*) entre as retas existentes. Para se
obter o menor no lugar do maior, troque os sinais de A,
B e do resultado da query.
Funciona para int e double.

#include <set>
#include <algorithm>
#include <cmath>
using namespace std;

// Caso queira double, troque as 2 linhas abaixo
typedef long long int T;
#define INF 0x3f3f3f3f3f3f3f3fLL

struct line {
  T a, b, xmax;
  line(T a, T b) : a(a), b(b) {};
};

bool compa(line a, line b) {
  return a.a < b.a;
}

bool compx(line a, line b) {
  return a.xmax < b.xmax;
}

bool(*fcompa)(line,line) = compa;
bool(*fcompx)(line,line) = compx;

set<line, bool(*)(line, line)> sa(fcompa);
set<line, bool(*)(line, line)> sx(fcompx);

set<line, bool(*)(line, line)>::iterator it;

/* Funcao auxiliar */
void add(line r) {
  sa.insert(r);
  sx.insert(r);
}

/* Funcao auxiliar */
void remove(line r) {
  sa.erase(r);
  sx.erase(r);
}

/* Funcao auxiliar */
T getxMax(line r, line s) {
   //return (s.b - r.b) / (r.a - s.a); // para double
  return floor((double) (s.b - r.b) / (double) (r.a - s.a));
}

/* Funcao auxiliar */
T gety(line r, T x) {
  return r.a * x + r.b;
}

void init() {
  sa.clear();
  sx.clear();
}

T query(T x) {
  if (sx.empty()) return -INF;
  line r(0, 0);
  r.xmax = x;
  it = sx.lower_bound(r);
  return gety(*it, x);
}

bool insert(T a, T b) {
  line r(a, b);
  it = sa.lower_bound(r);
  if (it != sa.end() && it->a == r.a) {
    if (it->b >= r.b) return false;
    remove(*it);
  }
  it = sa.lower_bound(r);
  if (it != sa.end() && it != sa.begin()) {
    line s = *it;
    it--;
    if (getxMax(r, s) <= getxMax(r, *it)) return false;
  }
  while (1) {
    it = sa.lower_bound(r);
    if (it == sa.end()) break;
    if (getxMax(r, *it) >= it->xmax) {
      remove(*it);
    }
    else {
      break;
    }
  }
  while (1) {
    it = sa.lower_bound(r);
    if (it == sa.begin()) break;
    it--;
    line s = *it;
    if (it == sa.begin()) {
      remove(s);
      s.xmax = getxMax(s, r);
      add(s);
```

```
      break;
    }
    it--;
    line t = *it;
    remove(s);
    if (getxMax(s, r) > t.xmax) {
      s.xmax = getxMax(s, r);
      add(s);
      break;
    }
  }
  it = sa.lower_bound(r);
  if (it == sa.end()) r.xmax = INF;
  else r.xmax = getxMax(r, *it);
  add(r);
  return true;
}
```

### 4.5 De Brujin Sequence

```
Descricao: a cyclic sequence of a given alphabet A with size
k for which every possible subsequence of length n in A
appears as a sequence of consecutive characters exactly once
OBS: sequence has length k^n

string seq;

int pw(int b,int a){
  int ans = 1;
  while( a ){
    if(a&1) ans *= b;
    b *= b;
    a /= 2;
  }
  return ans;
}

void debruijn( int n, int k ){
  seq = "";
  char s[n];
  if( n == 1 ){
    for( int i = 0; i < k; i++ )
    seq += char('0'+i);
  } else {
    for( int i = 0; i < n-1; i++ )
      s[i] = k-1;
    int kn = pw(k,n-1);
    char nxt[kn]; memset(nxt,0,sizeof(nxt));
    kn *= k;
    for( int h = 0; h < kn; h++ ){
      int m = 0;
      for( int i = 0; i < n-1; i++ ){
        m *= k;
        m += s[(h+i)%(n-1)];
      }
      seq += char('0'+nxt[m]);
      s[h%(n-1)] = nxt[m];
      nxt[m]++;
    }
  }
}
```

### 4.6 Decomposição Heavy Light

```
Complexidade: O(lg n) LCA / O(lg^2 n) queries
Descrição: Particiona os vértices de uma árvore em chains
(sequência de vértices ancestrais) de modo que qualquer
caminho usa um número logarítmico de chains, que podem ser
incrementadas para responder queries em caminhos (ver ex.)

#include <vector>
using namespace std;
#define MAXN 100100

vector<int> g[MAXN];
/* Vértice do topo da chain i, tam da chain i e qtd delas */
int head[MAXN], chsz[MAXN], nch;
/* Chain do vértice i e seu índice nela (cresce pra raiz) */
int chain[MAXN], chidx[MAXN];
/* Altura do vértice i, seu antecessor e tam da subárvore */
int depth[MAXN], pai[MAXN], size[MAXN];

/* Adiciona um vértice v no topo da chain c */
void chadd(int v, int c) {
  chidx[v] = chsz[c]++;
  chain[v] = c;
  head[c] = v;
}

/* Gera as chains e vetores associados */
void dfshl(int x) {
  size[x]=1;

  for (int i = 0; i < g[x].size(); i++) {
    int v = g[x][i];
    if (pai[x] != v) {
      depth[v] = depth[x]+1;
      pai[v] = x;

      dfshl(v);
      size[x] += size[v];
    }
  }

  chain[x] = -1;
  for (int i = 0; i < g[x].size(); i++)
    if (g[x][i] != pai[x] && size[g[x][i]] > size[x]/2)
      chadd(x, chain[g[x][i]]);

  if (chain[x] == -1) chadd(x, nch++);
}

/* Exemplo de LCA. Percorre as chains no caminho entre a e b
 Pode ser alterado para responder query usando uma estrutura
 de dados de intervalos por chain (por ex. BIT, segtree) */
int lca(int a, int b) {
  while (chain[a] != chain[b]) {
    if (depth[head[chain[a]]] > depth[head[chain[b]]])
      // query chain[a] em [chidx[a], chsz[chain[a]]-1]
      a = pai[head[chain[a]]];
    else
      // query chain[b] em [chidx[b], chsz[chain[b]]-1]
      b = pai[head[chain[b]]];
  }

  if (depth[a] < depth[b]) {
    // query chain[a] em [chidx[b], chidx[a]]
    return a;
  }
  // query chain[a] em [chidx[a], chidx[b]]
  return b;
}
```

### 4.7 Dinic Maximum Flow

```
int last_edge[MAXV], cur_edge[MAXV], dist[MAXV];
int prev_edge[MAXE], cap[MAXE], flow[MAXE], adj[MAXE];
int nedges;

void d_init() {
    nedges = 0;
    memset(last_edge, -1, sizeof last_edge);
}

void d_edge(int v, int w, int capacity, bool r = false) {
    prev_edge[nedges] = last_edge[v];
    cap[nedges] = capacity;
    adj[nedges] = w;
    flow[nedges] = 0;
    last_edge[v] = nedges++;

    if(!r) d_edge(w, v, 0, true);
}

bool d_auxflow(int source, int sink) {
    queue<int> q;
    q.push(source);

    memset(dist, -1, sizeof dist);
    dist[source] = 0;
    memcpy(cur_edge, last_edge, sizeof last_edge);

    while(!q.empty()) {
        int v = q.front(); q.pop();
        for(int i = last_edge[v]; i != -1;
        i = prev_edge[i]) {
            if(cap[i] - flow[i] == 0) continue;

            if(dist[adj[i]] == -1) {
                dist[adj[i]] = dist[v] + 1;
                q.push(adj[i]);

                if(adj[i] == sink) return true;
            }
        }
    }

    return false;
}
```

```
int d_augmenting(int v, int sink, int c) {
    if(v == sink) return c;

    for(int& i = cur_edge[v]; i != -1; i = prev_edge[i]) {
        if(cap[i] - flow[i] == 0 || dist[adj[i]]!=dist[v]+1)
            continue;

        int val;
        if(val = d_augmenting(adj[i], sink,
        min(c, cap[i] - flow[i]))) {
            flow[i] += val;
            flow[i^1] -= val;
            return val;
        }
    }

    return 0;
}

int dinic(int source, int sink) {
    int ret = 0;
    while(d_auxflow(source, sink)) {
        int flow;
        while(flow = d_augmenting(source, sink, 0x3f3f3f3f))
            ret += flow;
    }

    return ret;
}
```

### 4.8 Floyd Cycle Finding

```
pair<int, int> floyd(int x0) {
    int t = f(x0), h = f(f(x0)), start = 0, length = 1;
    while(t != h)
        t = f(t), h = f(f(h));

    h = t; t = x0;
    while(t != h)
        t = f(t), h = f(h), ++start;

    h = f(t);
    while(t != h)
        h = f(h), ++length;

    return make_pair(start, length);
}
```

### 4.9 Funções para Datas

Complexidade: O(1)

Zeller: Eh capaz de calcular o dia da semana para o calendario gregoriano (atual) - chame zeller() -, ou calendario juliano (antigo, considerava bissexto todo ano multiplo de 4, sem as regras de multiplo de 100 e 400) - chame zeller_julian(). Getdate: Retorna o numero de dias a partir do ano 0 ate a data

```
bool bissex(int y) { return (y%4==0 && (y%100 || y%400==0)); }

int zeller(int d, int m, int y) {
  if(m<3) --y, m+=12;
  return (d + ((m+1)*13)/5 + y + y/4 +
          6*(y/100) + y/400 + 6) % 7;
}

int zeller_julian(int d, int m, int y) {
  if(m<3) --y, m+=12;
  return (d + ((m+1)*13)/5 + y + y/4 + 4) % 7;
}

int getdate(int d, int m, int y) { //mes e dia a partir de 1
  int qm[]={0,31,28,31,30,31,30,31,31,30,31,30,31};
  int s=0;
  for (int i=1; i<m; i++) s+=qm[i];

  int res=365*y+s+d+(y/4-y/100+y/400);
  if (m<3 && bissex(y)) res--;
  return res;
}
```

### 4.10 Geometria 3D

```
using namespace std;
const double pi = acos(-1.0);
const double EPS = 1e-9;
const double INF = 1e50;
struct pt3;
struct line3;

int cmp(double a, double b = 0.0){
  if(fabs(a-b) < EPS) return 0;
  return a > b ? 1 : -1;
}

struct pt3{
  double x, y, z;
  pt3(double x = 0.0, double y = 0.0, double z = 0.0):
    x(x), y(y), z(z) {}
  double length(){ return sqrt(x*x + y*y + z*z); }
  double length2() { return x*x + y*y + z*z; }
  pt3 operator + (pt3 p) { return pt3(x+p.x,y+p.y,z+p.z); }
  pt3 operator - (pt3 p) { return pt3(x-p.x,y-p.y,z-p.z); }
  pt3 operator * (double k) { return pt3(x*k,y*k,z*k); }
  pt3 operator / (double k) { return pt3(x/k,y/k,z/k); }
  pt3 normalize() { return (*this)/length(); };
};

double dist(pt3 a, pt3 b){ return (b-a).length(); }

double dot(pt3 a, pt3 b) {
  return a.x*b.x + a.y*b.y + a.z* b.z;
}
pt3 cross(pt3 a, pt3 b) {
  return pt3(a.y*b.z - a.z*b.y,
  a.z*b.x - a.x*b.z, a.x*b.y - a.y*b.x);
}

struct line3{
  pt3 a, b;
  line3(pt3 a = pt3(), pt3 b = pt3()) : a(a), b(b) {}
  //direcao da reta - (nao normalizado)
  pt3 dir() { return (b-a); }
};

//ponto mais proximo de uma reta
//retorna o ponto da reta mais proximo de p
pt3 closest_point_line(pt3 p, line3 l){
  pt3 dir = l.dir();
  return l.a + dir*dot(p - l.a, dir)/dir.length2();
}

//distancia entre retas
//retorna a distancia minima entre duas retas
double dist(line3 r, line3 s){
  pt3 ort = cross(r.dir(), s.dir());
  if(!cmp(ort.length()))
  return dist(closest_point_line(r.a, s), r.a);
  return dot(s.a - r.a, ort)/ort.length();
}

//encontra o ponto mais proximo entre duas retas
//retorna o ponto em r mais proximo da resta s
//assume retas nao paralelas
bool closest_point_line_line(line3 r, line3 s, pt3& close){
  pt3 rdir = r.dir(), sdir = s.dir();
  double rr = dot(rdir, rdir);
  double ss = dot(sdir, sdir);
  double rs = dot(rdir, sdir);
  double t = dot(r.a - s.a, rdir)*ss
    - dot(r.a - s.a, sdir)*rs;
  //retas paralelas
  if(!cmp(rs*rs - rr*ss)) return false;
  t /= (rs*rs - rr*ss);
  close = r.a + rdir*t;
  return true;
}

//ponto mais proximo do segmento
//retorna o ponto do segmento mais proximo de p
pt3 closest_point_seg(pt3 p, line3 l){
  pt3 ldir = (l.b - l.a);
  double s = dot(l.a - p, ldir)/ldir.length2();
  if(s < -1.0) return l.b;
  if(s > 0.0) return l.a;
  return l.a - ldir*s;
}

//define um plano no 3D
//p eh um ponto no plano
//n eh a normal do plano a partir de p
//(representacao util para calculos algebricos)
struct plane{
  pt3 n, p;
  plane(pt3 n = pt3(), pt3 p = pt3()) : n(n), p(p) {}
  plane(pt3 a, pt3 b, pt3 c) : n(cross(b-a, c-a)), p(a) {};
  //produto misto
```

```
  double d() { return -dot(n , p); }
};

//ponto do plano mais proximo de p
pt3 closest_point_plane(pt3 p, plane pl){
  return p - pl.n*(dot(pl.n, p - pl.p))/pl.n.length2();
}

//ponto de intersecao entre uma reta e um plano
//assume que reta nao eh paralela ao plano
bool intersect(line3 l, plane pl, pt3& inter){
  pt3 ldir = l.dir();
  if(!cmp(dot(pl.n, ldir)))
    return false; //reta paralela ao plano
  inter = l.a - ldir*( (dot(pl.n, l.a) +
    pl.d())/dot(pl.n, ldir) );
  return true;
}
//intersecao de planos
//assume que os planos nao sao paralelos
bool intersect(plane u, plane v, line3& inter){
  pt3 p1 = u.n*(-u.d()), uv = cross(u.n, v.n);
  pt3 uvu = cross(uv, u.n);
  if(!cmp(dot(v.n, uvu))) return false; //planos paralelos
  pt3 p2 = p1 - uvu*((dot(v.n, p1)+v.d())/(dot(v.n, uvu)));
  inter.a = p2;
  inter.b = p2 + uv;
  return true;
}

//angulo entre dois vetores
double angle(pt3 a, pt3 b){
  return acos(dot(a, b)/(a.length()*b.length()));
}
//determina o volume formado pelo tetraedro delimitado
//pelos pontos a, b, c, d
double signed_volume(pt3 a , pt3 b, pt3 c, pt3 d){
  plane pl(b-a, c-a, d-a);
  return dot(cross(b-a, c-a), (d-a))/6.0;
  return pl.d()/6.0;
}
//area formada pelo triangulo a, b, c
double signed_area(pt3 a, pt3 b, pt3 c){
  double h = dist(a, closest_point_line(a , line3(b, c)));
  return dist(b, c)*h/2.0;
}
void print(pt3 p){
  cout << "(" << p.x << "," << p.y << "," << p.z << ")";
}

int main(){
  cout << fixed << setprecision(6);
  //declara os pontos de dois planos
  pt3 a1(100, 0, 0), b1(0, 100, 0), c1(0, 0 ,0);
  pt3 a(100, 0, 0), b(0, 100, 0), c(0, 0 ,100);
  //cria os dois planos
  plane u(a1, b1, c1), v(a, b, c);
  //cria duas retas
  line3 l(pt3(0, 0, 0), pt3(0, 0, 10)), s(pt3(0, 2, 2),
    pt3(2, 0, 2));
  pt3 close;
  //calcula o ponto mais proximo entre duas retas
  closest_point_line_line(l, s, close); print(close);
  cout << endl;
  //calcula o ponto mais proximo na reta s
  print(closest_point_line(pt3(0, 2, 0), s)); cout << endl;
  //calcula a distancia entre as duas retas
  cout << dist(l, s) << endl;
  //calcula a intersecao de dois planos
  cout << intersect(u, v, l) << endl;
  print(l.a); cout << " "; print(l.b); cout << endl;
  //verifica se a solucao eh ortogonal
  //as normais dos dois planos
  //dot = 0
  cout << dot(l.dir(), v.n) << endl;
  cout << dot(l.dir(), u.n) << endl;
  //calcula os angulos entre vetores coplanares
  //soma deve ser igual a pi neste caso (triangulo)
  cout << angle(a-c, b-c) << " " << angle(b-a, c-a)
  << " " << angle(c-b, a-b) << endl;
  cout << angle(a-c, b-c) + angle(b-a, c-a) +
    angle(c-b, a-b) << endl;
  //testa se os pontos da solucao estao certos
  //volume = 0 (pontos coplanares)
  cout << signed_volume(l.a, a, b, c) << endl;
  cout << signed_volume(l.b, a, b, c) << endl;
  cout << signed_volume(l.a, a1, b1, c1) << endl;
  cout << signed_volume(l.b, a1, b1, c1) << endl;
  //calcula a area do triangulo no espaco
  cout << signed_area(a, b, c) << endl;
}
```

### 4.11 Knight Distance

```
Complexidade: O(1)
Descricao: Determina em O(1) a distância (em movimentos de
cavalo) entre 2 pontos de um tabuleiro (infinito ou finito).
Se o tabuleiro for finito, deve ter tamanho n x m com
n >= 4 e m >= 4.

#include <algorithm>
using namespace std;

int knightdist_inf(int x1, int y1, int x2, int y2) {
  int dx=abs(x2-x1);
  int dy=abs(y2-y1);
  if (abs(dx)==1 && dy==0) return 3;
  if (abs(dy)==1 && dx==0) return 3;
  if (abs(dx)==2 && abs(dy)==2) return 4;

  int lb=max((dx+1)/2, (dy+1)/2);
  lb = max(lb, (dx+dy+2)/3);
  if ((lb%2)!=(dx+dy)%2) lb++;
  return lb;
}

int n,m; //tamanho do tabuleiro

int knightdist(int x1, int y1, int x2, int y2) {
  if(x1==n || x2==n) {
    x1 = n+1 - x1;
    x2 = n+1 - x2;
  }
  if(y1==m || y2==m) {
    y1 = m+1 - y1;
    y2 = m+1 - y2;
  }
  if((x1==1 && y1==1) || (x2==1 && y2==1)) {
    int a=abs(x1-x2), b=abs(y1-y2);
    if(a==0 && b==3 && m==4) return 5;
    if(b==0 && a==3 && n==4) return 5;
    if(a==1 && b==1) return 4;
  }
  return knightdist_inf(x1,y1,x2,y2);
}
```

### 4.12 Maior Retângulo em um Histograma

```
Complexidade: O(n)
Descricao: Dado um vetor que contem alturas (>=0) das
barras de um histograma de largura fixa = 1, calcula
a área do maior retangulo contido no histograma

#include <algorithm>
#define MAX 100100
using namespace std;

int sh[MAX], sp[MAX];

long long histogram(int *v, int n) {
  int qs=1, curh=0;
  long long res=0;

  sh[0]=-1; sp[0]=0;
  v[n]=-1;

  for (int i=0; i<n+1; i++) {
    if (i<n && v[i]>=curh) {
      sh[qs]=v[i];
      sp[qs++]=i;
    }
    else {
      while (sh[qs-1]>v[i]) {
        qs--;
        res=max(res, (long long) sh[qs]*(i-sp[qs]));
      }
      sh[qs++]=v[i];
    }
    curh=v[i];
  }

  return res;
}
```

### 4.13 Polynomial Roots

```
typedef complex<double> cdouble;
int cmp(cdouble x, cdouble y = 0) {
  return cmp(abs(x), abs(y));
```

```
}

const int TAM = 200;

struct poly {
  cdouble poly[TAM]; int n;
  poly(int n = 0): n(n) { memset(p, 0, sizeof(p)); }
  cdouble& operator [](int i) { return p[i]; }
  poly operator ~() {
    poly r(n-1);
    for (int i = 1; i <= n; i++)
    r[i-1] = p[i] * cdouble(i);
    return r;
  }
  pair<poly, cdouble> ruffini(cdouble z) {
    if (n == 0) return make_pair(poly(), 0);
    poly r(n-1);
    for (int i = n; i > 0; i--) r[i-1] = r[i] * z + p[i];
    return make_pair(r, r[0] * z + p[0]);
  }
  cdouble operator ()(cdouble z) { return ruffini(z).second; }
  cdouble find_one_root(cdouble x) {
    poly p0 = *this, p1 = ~p0, p2 = ~p1;
    int m = 1000;
    while (m--) {
      cdouble y0 = p0(x);
      if (cmp(y0) == 0) break;
      cdouble G = p1(x) / y0;
      cdouble H = G * G - p2(x) - y0;
      cdouble R = sqrt(cdouble(n-1) * (H * cdouble(n) - G*G));
      cdouble D1 = G + R, D2 = G - R;
      cdouble a = cdouble(n) / (cmp(D1, D2) > 0 ? D1 : D2);
      x -= a;
      if (cmp(a) == 0) break;
    }
    return x;
  }
  vector<cdouble> roots() {
    poly q = *this;
    vector<cdouble> r;
    while (q.n > 1) {
      cdouble z(rand() / double(RAND_MAX),
        rand() / double(RAND_MAX));
      z = q.find_one_root(z); z = find_one_root(z);
      q = q.ruffini(z).first;
      r.push_back(z);
    }
    return r;
  }
};
```

### 4.14 Range Minimum Query (RMQ)

```
Complexidade: O(n lg n)-preprocessamento e O(1)-consulta
Descricao: Apos o preprocessamento de um vetor v, o algoritmo
responde de forma eficiente o indice do elemento minimo em
v[a..b]. Em caso de empate, é devolvido o maior índice.
Caso queira o menor, descomente o = em pairmin()

using namespace std;

#define N 100100
#define LOG 16     // piso de log2(N)

int *va, Log2[N], p[LOG+1][N];

int pairmin(int i1, int i2) {
   return va[i1]</*=*/va[i2] ? i1:i2;
}

void init(int *a, int n) {
  va = a;
  for (int i=1,k=0; i<=n; i++) {
    Log2[i] = k;
    if (1 << (k+1) == i) k++;
  }

  int ln = Log2[n];
  for (int i=0; i<n; i++) p[0][i]=i;
  for (int i=1; i<=ln; i++)
    for (int j=0; j + (1 << i) - 1 < n; j++) {
      int i1 = p[i-1][j];
      int i2 = p[i-1][j+ (1 << i-1)];
      p[i][j] = pairmin(i1, i2);
    }
}

int query(int b, int e) {
  int ln = Log2[e - b + 1];
  int i1 = p[ln][b];
  int i2 = p[ln][e - (1 << ln) + 1];

  return pairmin(i1,i2);
}
```

### 4.15 Romberg - Integral

```
long double romberg(long double a, long double b,
                    long double(*func)(long double)) {
    long double R[16][16], div = (b-a)/2;

    R[0][0] = div * (func(a) + func(b));
    for(int n = 1; n <= 15; n++, div /= 2) {
        R[n][0] = R[n-1][0]/2;
        for(long double sample = a + div; sample < b;
         sample += 2 * div)
            R[n][0] += div * func(a + sample);
    }

    for(int m = 1; m <= 15; m++)
        for(int n = m; n <= 15; n++)
            R[n][m] = R[n][m-1] + 1/(pow(4, m)-1) *
             (R[n][m-1] - R[n-1][m-1]);

    return R[15][15];
}
```

### 4.16 Rope (via árvore cartesiana)

```
Complexidade: O(lg n) por operação
Descricao: Estrutura para manipular cadeias, suporta merge
e split em qualquer ponto. Implementada com árvore cartesiana
com Y's aleatórios, o que a torna balanceada.

#include <cstdio>
#include <cstring>
#include <algorithm>
#define MAXN 100100
using namespace std;

struct _node {
  int c;      // Contagem de nós na subárvore (inclui raiz)
  int v, sum; // "Valor" da raiz e a soma deles na subárvore
  int id;     // Índice do nó na cadeia original
  int y;
  _node *l,*r;
} mem[MAXN], nil;

typedef _node* node;

// Atualiza o nó T dado que seus filhos já o fizeram
// pode ser extendido se houver outras variáveis de interesse
void fix(node T) {
  T->sum = T->v+T->l->sum+T->r->sum;
  T->c = 1+T->l->c+T->r->c;
}

// Divide subárvore T em [L,R], deixando L com x elementos
void split(node T, int x, node &L, node &R) {
  if (T==&nil) L = R = &nil;
  else if (x <= T->l->c) {
    split(T->l, x, L, T->l);
    fix(T);
    R = T;
  }
  else {
    split(T->r, x-T->l->c-1, T->r, R);
    fix(T);
    L = T;
  }
}

node merge(node L, node R) {
  if (L == &nil) return R;
  if (R == &nil) return L;
  if (L->y > R->y) {
    L->r = merge(L->r, R);
    fix(L);
    return L;
  }
  R->l = merge(L, R->l);
  fix(R);
  return R;
}

node add(node T, node N) {
  if (T == &nil) return N;
  if (T->y < N->y) {
    split(T, N->id, N->l, N->r);
    fix(N);
```

```
    return N;
  }
  if (N->id < T->id) T->l = add(T->l,N);
  else T->r = add(T->r,N);

  fix(T);
  return T;
}

// Uso como árvore de segmentos da variável sum
int query(node T, int ll, int rr, int a, int b) {
  if (T == &nil || a > b) return 0;
  if (a == ll && b == rr) return T->sum;

  int me = ll+T->l->c;
  int res = query(T->l, ll, me-1, a, min(b, me-1))+
            query(T->r, me+1, rr, max(a, me+1), b);

  if (a<=me && b>=me) res += T->v;

  return res;
}

// Devolve o nó na x-ésima posição de T
node getid(node T, int x) {
  if (T->l->c == x) return T;
  if (T->l->c > x) return getid(T->l, x);
  return getid(T->r, x-T->l->c-1);
}

int main() {
  int n,x;

  while (scanf(" %d",&n)==1 && n) {
    node t = &nil;

    for (int i=0; i<n; i++) {
      scanf(" %d",&x);

      mem[i].y=rand()%123456789;
      mem[i].v=mem[i].sum=x;
      mem[i].c=1;
      mem[i].l=mem[i].r=&nil;
      mem[i].id=i;

      t=add(t, &mem[i]);
    }

    // troca de ordem as duas metades da sequência
    node p1, p2;
    split(t, n/2, p1, p2);
    t=merge(p2, p1);

    for (int i=0; i<n; i++)
      printf("%d\n",getid(t, i)->v);
  }
  return 0;
}
```

# 5 Programação Dinâmica

## 5.1 Longest Increasing Subsequence (LIS)

```
Complexidade: O(n*lg k), sendo k o tamanho da LIS
Descricao: Determina o tamanho da LIS do vetor v,
que pode ter numeros negativos, inclusive. Os trechos
de codigo comentados são  relativos apenas a parte
de reconstrucao de uma LIS. Esse algoritmo so funciona
quando a relacao entre dois elementos eh transitiva
(a < b e b < c => a < c), como acontece com
numeros, strings, etc.

#include <stdio.h>
#include <string.h>
#define MAXN 1000
#define INF 0x3f3f3f3f

int v[MAXN+1] /*,ant[MAXN+1],li[MAXN+1]*/ ;
int pd[MAXN+1] /*,ipd[MAXN+1]*/ ;
/*pd armazena o menor elemento que lide-
  ra uma IS de tamanho i ate o momento*/

int lis(int n) {
  int es,di,m,mx=0;

  memset(pd,0x3f,sizeof(pd));
  pd[0]=-INF;

  for (int i=0;i<n;i++) {
    es=0; di=i;
    while (es<di) {
      m=(es+di+1)/2;
      if (pd[m]<v[i]) es=m;
      else di=m-1;
    }

    if (pd[es]<v[i] && pd[es+1]>v[i]) {
      pd[es+1]=v[i];
      if (es+1>mx) mx=es+1;
      /* ipd[es+1]=i;
         ant[i]=ipd[es];*/
    }
  }
  return mx;
}

/*reconstroi uma IS de tamanho tam depois de chamar lis(n)*/
/*void build(int tam) {
  int p=ipd[tam];

  if (pd[tam]==INF) printf("-1\n");
  else if (tam>0) {
    for (int i=0;i<tam;i++) {
      li[i]=v[p];
      p=ant[p];
    }

    for (int i=tam-1;i>0;i--) printf("%d ",li[i]);
    printf("%d\n",li[0]);
  }
  else printf("\n");
}*/
```

# 6 Strings

## 6.1 Aho-Corasick

```
Complexidade: O(texto + padrões + ocorrências)
Descrição: Dado um conjunto de padrões (strings) e um texto,
encontra todas as ocorrências dos padrões no texto

#include <map>
#include <vector>
#include <queue>
using namespace std;
typedef pair<int,int> pii;

/* Tamanho total dos padrões */
#define MAXST (1000100)

struct No {
  vector<pii> out; // num e tamanho do pad
  map<char, int> lis;
  int fail;
  int nxt; // aponta para o próx. sufixo com out.size > 0
};
No t[MAXST];
int qNo, qPad;

void init() {
  t[0].fail = t[0].nxt = -1;
  t[0].lis.clear();
  t[0].out.clear();
  qNo = 1;
  qPad = 0;
}

void add(const char *pad) {
  int no = 0, len = 0;
  for (int i = 0; pad[i]; i++, len++) {
    if (t[no].lis.find(pad[i]) == t[no].lis.end()) {
      t[qNo].lis.clear(); t[qNo].out.clear();
      t[no].lis[pad[i]] = qNo;
      no = qNo++;
    }
    else no = t[no].lis[pad[i]];
  }
  t[no].out.push_back(pii(qPad++, len));
}

// Ativar aho-corasick, ajustando funções de falha
void preprocess() {
  int no, v, f, w;
  queue<int> fila;
  for (map<char,int>::iterator it = t[0].lis.begin();
    it != t[0].lis.end(); it++) {
    t[no = it->second].fail = 0;
    t[no].nxt = t[0].out.size() ? 0 : -1;
    fila.push(no);
```

```
  }
  while (!fila.empty()) {
    no = fila.front(); fila.pop();
    for (map<char,int>::iterator it = t[no].lis.begin();
      it != t[no].lis.end(); it++) {
      char c = it->first;
      v = it->second;
      fila.push(v);
      f = t[no].fail;
      while (t[f].lis.find(c) == t[f].lis.end()) {
        if (f == 0) { t[0].lis[c] = 0; break; }
        f = t[f].fail;
      }
      w = t[f].lis[c];
      t[v].fail = w;
      t[v].nxt = t[w].out.size() ? w : t[w].nxt;
    }
  }
}

// descomente p/ obter só 1 ocorrência por padrão (+rápido)
// int mark[MAXST];

// Busca em text devolve pares (índice do padrão, posição)
void find(const char *text, vector<pii> &res) {
  int v, no = 0;

  // memset(mark,0,sizeof(mark));
  for (int i = 0; text[i]; i++) {
    while (t[no].lis.find(text[i]) == t[no].lis.end()) {
      if (no == 0) { t[0].lis[text[i]] = 0; break; }
      no = t[no].fail;
    }
    for (v = no = t[no].lis[text[i]]; v != -1; v = t[v].nxt) {
      // if (mark[v]++) break;
      for (int k = 0 ; k < (int)t[v].out.size() ; k++) {
        // encontrado padrao t[v].out[k].first no
        // intervalo (i-t[v].out[k].second+1)..i
        res.push_back(pii(t[v].out[k].first,
          i - t[v].out[k].second + 1));
      }
    }
  }
}

int main(){
  char text[10010], pat[10010];
  int qpat;

  scanf(" %s %d", text, &qpat);
  init();

  for (int i=0; i<qpat; i++) {
    scanf(" %s",pat);
    add(pat);
  }

  preprocess();
  vector<pii> oc;
  find(text, oc);
  for (int i=0; i<(int)oc.size(); i++)
    printf("Padrão %d em %d\n", oc[i].first, oc[i].second);
  return 0;
}
```

## 6.2 Array de Sufixos n*lg(n)

```
Complexidade: O(n*lg(n))
Descricao: Gera o indice de cada sufixo quando ordenados
lexicograficamente em O(n*lg(n)). No entanto, só calcula
LCP de sufixos adjacentes.

#include <cstring>
#include <algorithm>

using namespace std;

#define N 150000

char str[N], inp[N];
int H, Bucket[N], nBucket[N], Rank[N], Height[N], c;

struct Suffix {
  int idx; // Suffix starts at idx, i.e. it's str[ idx .. L-1 ]
  bool operator<(const Suffix& sfx) const
  // Compares two suffixes based on their first 2H symbols,
  // assuming we know the result for H symbols.
  {
    if(H == 0) return str[idx] < str[sfx.idx];
    else if(Bucket[idx] == Bucket[sfx.idx])
      return (Bucket[idx+H] < Bucket[sfx.idx+H]);
    else
      return (Bucket[idx] < Bucket[sfx.idx]);
  }
  bool operator==(const Suffix& sfx) const {
    return !(*this < sfx) && !(sfx < *this);
  }
} Pos[N];

int UpdateBuckets(int L) {
  int start = 0, id = 0, c = 0;
  for(int i = 0; i < L; i++) {
    if(i != 0 && !(Pos[i] == Pos[i-1])) {
      start = i;
      id++;
    }
    if(i != start)
      c = 1;
    nBucket[Pos[i].idx] = id;
  }
  memcpy(Bucket, nBucket, 4 * L);
  return c;
}

void SuffixSort(int L) {
  H = 0;
  for(int i = 0; i < L; i++) Pos[i].idx = i;
  sort(Pos, Pos + L);
  c = UpdateBuckets(L);
  for(H=1;c;H *= 2) {
    // Sort based on first 2*H symbols, assuming
    // that we have sorted based on first H character
    sort(Pos, Pos+L);
    // Update Buckets based on first 2*H symbols
    c = UpdateBuckets(L);
  }
}

// Must compute the suffix array Pos first
void ComputeLCP(int L) {
  for (int i = 0; i < L; i++) Rank[Pos[i].idx] = i;
  int h = 0;
  for (int i = 0; i < L; i++)
    if (Rank[i] > 0) {
      int k = Pos[Rank[i] - 1].idx;
      while (str[i+h] == str[k+h])
        ++h;
      Height[Rank[i]] = h;
      if (h > 0) --h;
    }
}

int main() {
  scanf("%s",str);

  /* e necessario colocar o tamanho + 1 */
  int n = strlen(str) + 1;
  SuffixSort(n);

  ComputeLCP(n);

  /* Pos[i].idx guarda a posicao na string original */
  for (int i = 0; i < n; i++) {
    printf("%d\n", Pos[i].idx);
  }

  /* Height[i] tem o LCP da posicao i com a posicao i-1 */
  for (int i = 0; i < n; i++) {
    printf("%d\n", Height[i]);
  }

  return 0;
}
```

## 6.3 Busca de Strings (KMP)

```
Complexidade: O(n+m)
Descricao: Acha todas as ocorrencias do padrao p num texto t

#define MAXNP 1000

int fail[MAXNP+1];

void buildFail(char *p, int psize) {
    int j = fail[0] = -1;
    for (int i = 1; i <= psize; i++) {
        while (j >= 0 && p[j] != p[i-1]) j = fail[j];
        fail[i] = ++j;
    }
```

```
}

void kmp(char *text, char *pattern) {
    int m = strlen(pattern), n = strlen(text);
    buildFail(pattern, m);
    for (int i = 0, k = 0; i < n; i++) {
        while (k >=0 && pattern[k] != text[i]) k = fail[k];
        if (++k >= m){
            k = fail[k];
            printf("Achou em %d\n", i-m+1);
        }
    }
}
```

### 6.4 Hash de Strings

```
Complexidade: O(n)
Descricao: Após preprocessar uma string, calcula o hash
de qualquer substring sua em tempo constante.

#include <algorithm>
#define MAXN 100100
#define B 33
using namespace std;
typedef unsigned long long hash;

hash h[MAXN], pwr[MAXN];
char s[MAXN];

void gen(char *s) {
  h[0] = 0;
  pwr[0] = 1;
  for (int i = 0; s[i]; i++) {
    h[i+1] = h[i] * B + s[i];
    pwr[i+1] = pwr[i] * B;
  }
}

// Calcula o hash da substring s[a..b]
hash sect(int a, int b) {
  if (a > b) return 0;
  return h[b+1] - h[a] * pwr[b - a + 1];
}

// Maior prefixo comum das substrings s[a..n-1], s[b..n-1]
int lcp(int a, int b, int n) {
  int es = 0, di = min(n-b, n-a);

  while (es < di) {
    int me = (es+di+1)/2;
    if (sect(a, a+me-1) == sect(b, b+me-1)) es = me;
    else di = me-1;
  }
  return es;
}
```

## 7 Matemática

### 7.1 Geometria

**Matriz de rotação**

$$\begin{bmatrix} x_\theta \\ y_\theta \end{bmatrix} = \begin{bmatrix} cos\ \theta & -sen\ \theta \\ sen\ \theta & cos\ \theta \end{bmatrix} \begin{bmatrix} x \\ y \end{bmatrix}$$

**Fórmula de Brahmagupta** Sendo $a$, $b$, $c$, $d$ os lados do quadrilátero, $s = \frac{1}{2}(a+b+c+d)$:

$$A = \sqrt{(s-a)(s-b)(s-c)(s-d) - k}$$

E sendo $\theta$ a soma do ângulo de dois lados opostos, ou $p$ e $q$ os comprimentos das diagonais do quadrilátero, temos:

$$\begin{aligned} k &= abcd\cos^2\frac{\theta}{2} \\ &= \tfrac{1}{4}(ac+bd+pq)(ac+bd-pq) \end{aligned}$$

**Calota Esférica** Sendo $R$ o raio da esfera, $r$ o raio da base, e $h$ a altura da calota:

$$A_{calota} = 2\pi Rh$$

$$V_{calota} = \frac{\pi h}{6}\left(3r^2+h^2\right) = \frac{\pi h^2}{3}(3R-h)$$

**Área de Segmento Circular** Sendo $\alpha$ o ângulo formado pelo segmento circular, temos:

$$A_{segmento} = \frac{r^2}{2}(\alpha - sen\ \alpha)$$

Se tivermos $h$, a altura do segmento circular, ao invés de $\alpha$:

$$\alpha = 2acos\left(\frac{h}{r}\right)$$

**Centróide de um Polígono**

$$c_x = \frac{1}{6A}\sum_{i=0}^{n-1}(x_i + x_{i+1})(x_i y_{i+1} - x_{i+1}y_i)$$

$$c_y = \frac{1}{6A}\sum_{i=0}^{n-1}(y_i + y_{i+1})(x_i y_{i+1} - x_{i+1}y_i)$$

**Área de triângulo** Sendo $R$ o raio da circunferência circunscrita, e $r$ da inscrita, temos:

$$A_\triangle = \frac{abc}{4R} = \frac{(a+b+c)r}{2}$$

**Fórmula de Euler para Poliedros Convexos** $V$ vértices, $A$ arestas, $F$ faces: $V - A + F = 2$

**Teorema de Pick** Sendo $A$ a área de um polígono e $i$ e $b$ a quantidade de pontos de coordenadas inteiras no interior e na borda no polígono,respectivamente, temos:

$$A = i + b/2 - 1$$

**Quantidade de pontos de coordenas inteiras num segmento** Sendo $(x_1, y_1)$ e $(x_2, y_2)$ pontos de coordenadas inteiras nos extremos de um segmento:

$$q = mdc(|x_1 - x_2|, |y_1 - y_2|) + 1$$

### 7.2 Relações Binomiais

Relação de Stifel:

$$\binom{n}{k} = \binom{n-1}{k} + \binom{n-1}{k-1}$$

Absorções:

$$\binom{n}{k} = \frac{n-k+1}{k}\binom{n}{k-1} = \frac{n}{k}\binom{n-1}{k-1} = \frac{n}{n-k}\binom{n-1}{k}$$

Soma de quadrados de binomiais:

$$\sum_{k=0}^{n}\binom{n}{k}^2 = \binom{2n}{n}$$

### 7.3 Equações Diofantinas

Dados inteiros $a, b > 0$ e $c$, a equação $ax + by = c$ tem soluções sse $g = gcd(a,b)$ é divisor de $c$.

Sejam $x_g$ e $y_g$ a solução de $a \cdot x_g + b \cdot y_g = g$ obtida por Euclides. Então:

$$\begin{cases} x = x_g(c/g) + k \cdot b/g \\ y = y_g(c/g) - k \cdot a/g \end{cases} \quad k \in Z$$

### 7.4 Fibonacci

Fórmula em $lg(n)$:
$f(0) = 1$ e $f(1) = 1$

$$\begin{aligned} f(n) &= f(x)f(n-x) + f(x-1)f(n-x-1) \\ &= f(\lfloor \tfrac{n}{2} \rfloor)f(n - \lfloor \tfrac{n}{2} \rfloor) + f(\lfloor \tfrac{n}{2} \rfloor + 1)f(n - \lfloor \tfrac{n}{2} \rfloor - 1) \\ &= f(\lfloor \tfrac{n}{2} \rfloor)f(\lceil \tfrac{n}{2} \rceil) + f(\lfloor \tfrac{n}{2} \rfloor + 1)f(\lceil \tfrac{n}{2} \rceil - 1) \end{aligned}$$

Fórmula com potência de matrizes:

$$\begin{bmatrix} f(n+1) \\ f(n) \end{bmatrix} = \begin{bmatrix} 1 & 1 \\ 1 & 0 \end{bmatrix}^n \begin{bmatrix} f(1) \\ f(0) \end{bmatrix}$$

Propriedades:

- $f(n+1)f(n-1) - f(n)^2 = (-1)^n$
- $f(m)$ múltiplo de $f(n)$ sse $m$ múltiplo de $n$
- $mdc(f(m), f(n)) = f(mdc(m,n))$

## 7.5 Problemas clássicos

**Fila do cinema**: Sendo $n$ pessoas com \$5 e $m$ com \$10, temos:
$K_{0,m} = 0$ e $K_{n,0} = 1$
$K_{n,m} = K_{n-1,m} + K_{n,m-1}$

$$K_{n,m} = \binom{n+m}{n} - \binom{n+m}{n+1} = \frac{n-m+1}{n+1}\binom{n+m}{n}$$

**Números de Catalan**: É um caso do problema da *Fila de cinema*, com $n = m$.

$$C_n = \binom{2n}{n} - \binom{2n}{n+1} = \frac{1}{n+1}\binom{2n}{n}$$

*Aplicações*: 1) Número de expressões com $n$ pares de parênteses, todos abrindo e fechando corretamente. Exemplo: (()) ()(); 2) Número de maneiras de parentizar completamente $n+1$ fatores. Exemplo: $(ab)c$ $a(bc)$; 3) Número de árvores binárias completas com $n+1$ folhas; 4) Número de maneiras de triangularizar um polígono convexo de $n+2$ lados;

**Número de somas $x_1 + x_2 + \cdots + x_n = p$**
- Soluções não negativas: $CR_n^p = \binom{n+p-1}{p}$
- Soluções positivas $CP_n^p = \binom{p-1}{n-1}$

**Variáveis com restrições**: Quando alguns $x_i$ têm restrições do tipo $x_i \geq 3$, adotamos um $y_i$ tal que $x_i = 3 + y_i$.

Assim, seguindo a restrição de que $y_i \geq 0$, teremos $x_i \geq 3$. A soma fica, então:

$$\begin{aligned} x_1 + x_2 + \cdots + x_i + \cdots + x_n &= p \\ x_1 + x_2 + \cdots + y_i + \cdots + x_n &= p - 3 \end{aligned}$$

De forma geral, teremos:

$$CR_n^p = \binom{n + p - (b_1 + b_2 + \cdots + b_n) - 1}{p}$$

Sendo $b_i$ o decremento (pode ser negativo) na variável $x_i$.

**$x_1 + x_2 + \cdots + x_n \leq p$**

Definimos uma variável de $folga$, $f = p - (x_1 + x_2 + \cdots + x_n)$, e obtemos:

$$\begin{aligned} f &\geq 0 \\ x_1 + x_2 + \cdots + x_n + f &= p \end{aligned}$$

**Permutações Caóticas:** O número de permutações caóticas para $n$ elementos é dado por: $D_0 = 1; D_n = (-1)^n + nD_{n-1} = (n-1)(D_{n-1} + D_{n-2})$

**Triângulos de Lados em $\{1, 2, \cdots, n\}$**

$$f_{n+1} = f_n + \begin{cases} \frac{(n-2)^2}{2} & \text{, n par} \\ \left\lceil \frac{(n-2)(n-4)}{4} \right\rceil & \text{, n ímpar} \end{cases}$$

**Problema de Josephus:** Sendo $n$ pessoas em circulo, eliminando-se de $k$ em $k$, temos a recorrência:

$$\begin{aligned} f(1,k) &= 0 \\ f(n,k) &= (f(n-1,k) + k)(mod\ n) \end{aligned}$$

**Formas de Conectar um Grafo:** Seja um grafo com $k$ componentes com tamanhos $s_1, \cdots, s_k$. O número de maneiras de adicionar $k-1$ arestas de modo a conectá-lo é: $s_1 \cdots s_k n^{k-2}$

**Código de Gray:** $gray(i) = i \text{ xor } \frac{i}{2}$

**Código de Gray Invertido (n-bits):**

$$\overline{gray_n}(i) = (\frac{i}{2} \text{ or } (i \text{ and } ((n\%2)2^{n-1}))) \text{ xor } \begin{cases} i & \text{, i par} \\ \bar{i} & \text{, i ímpar} \end{cases}$$

## 7.6 Séries Numéricas

$$\sum_{i=1}^{n} i^2 = \frac{n(n+1)(2n+1)}{6}$$

$$\sum_{i=1}^{n} i^3 = \frac{n^2(n+1)^2}{4}$$

**PA de 2ª ordem:**

$$a_n = a_1 + b_1(n-1) + \frac{r}{2}(n-2)(n-1)$$

$$S_n = a_1 n + \frac{b_1 n(n-1)}{2} + \frac{r}{6}n(n-2)(n-1)$$

**PA de nª ordem:**

$$S_k = a_1\binom{k}{1} + \sum_{i=1}^{n} \Delta_i \binom{k}{i+1}$$

$\Delta_i$: Primeiro elemento considerando a i-ésima PA.
Exemplo: n = 5 ; seq = (1,32,243,1024,3125,7776,...)

$$\begin{aligned} \Delta_1 &= 32 - 1 = 31 \\ \Delta_2 &= 211 - 31 = 180 \\ \Delta_3 &= 570 - 180 = 390 \\ \Delta_4 &= 750 - 390 = 360 \\ \Delta_5 &= 480 - 360 = 120 \end{aligned}$$

Para a PA de 2ª ordem ficaria:

$$\begin{aligned} \Delta_1 &= b_1 \\ \Delta_2 &= b_2 - b_1 = r \end{aligned}$$

## 7.7 Matrizes e Determinantes

**Determinante de Vandermonde:**

$$V_n = \begin{vmatrix} 1 & 1 & \cdots & 1 \\ a_1 & a_2 & \cdots & a_n \\ a_1^2 & a_2^2 & \cdots & a_n^2 \\ \vdots & \vdots & \ddots & \vdots \\ a_1^{n-1} & a_2^{n-1} & \cdots & a_n^{n-1} \end{vmatrix} = \prod_{i>j}(a_i - a_j)$$

## 7.8 Probabilidades

**Probabilidade Condicional:**

$$P(B|A) = \frac{n(A \cap B)}{n(A)} = \frac{P(A \cap B)}{P(A)}$$

**Experimentos Repetidos:** Seja um experimento que se repete $n$ vezes, e em qualquer um deles temos $P(A) = p$ e, portanto, $P(\bar{A}) = 1 - p$. A probabilidade do evento $A$ ocorrer $k$ das $n$ vezes é:

$$P_k = \binom{n}{k} p^k (1-p)^{n-k}$$

## 7.9 Teoria dos Números

**Teorema de Fermat-Euler:** Se $p$ é primo, temos, para todo inteiro $a$: $a^{p-1} \equiv 1 (mod\ p)$. Se temos $a$ e $n$ coprimos: $a^{\phi(n)} \equiv 1 (mod\ n)$ onde $\phi(n) = n\prod_{p|n}\left(1 - \frac{1}{p}\right)$, $p$ é fator primo de $n$, é a quantidade de números entre 1 e $n$ que são coprimos com $n$.

**Teorema de Wilson:** $n$ é primo sse $(n-1)! \equiv -1 (mod\ n)$

**Soma dos Divisores:** A soma dos divisores de $n$ elevados à x-ésima potência, sendo $p_i$ os fatores primos e $a_i$ os expoentes correspondentes:

$$\sigma_x(n) = \prod_{i=1}^{r} \frac{p_i^{(a_i+1)x} - 1}{p_i^x - 1}$$

**Divisibilidade**
Considere o numéro como: $a_n a_{n-1} a_{n-2} ... a_2 a_1 a_0$

Por 3: A soma dos dígitos deve ser divisível por 3
Por 4: O número formado por $a_1 a_0$ deve ser divisível por 4
Por 7: A soma $a_2 a_1 a_0 - a_5 a_4 a_3 + a_8 a_7 a_6 - ...$ deve ser divisível por 7
Por 8: O número formado por $a_2 a_1 a_0$ deve ser divisível por 8
Por 9: A soma dos dígitos deve ser divisível por 9
Por 11: A soma $a_0 - a_1 + a_2 - a_3 + a_4 -$ deve ser divisível por 11
Por 13: A soma $a_2 a_1 a_0 - a_5 a_4 a_3 + a_8 a_7 a_6 - ...$ deve ser divisível por 13

**Equação Modular Linear:** Dada equação $ax \equiv b (mod\ m)$, se $b \equiv 0 (mod\ g)$ onde $g = \gcd(a, m)$, então as soluções são:

$$x = \frac{b}{g} * \text{invmod}(\frac{a}{g}, \frac{m}{g}) + k\frac{m}{g} \quad , k \in Z$$

### 7.10 Prime counting function ($\pi(x)$)

The prime counting function is asymptotic to $\frac{x}{\log x}$, by the prime number theorem.

| x | 10 | $10^2$ | $10^3$ | $10^4$ | $10^5$ | $10^6$ | $10^7$ | $10^8$ |
|---|---|---|---|---|---|---|---|---|
| $\pi(x)$ | 4 | 25 | 168 | 1.229 | 9.592 | 78.498 | 664.579 | 5.761.455 |

### 7.11 Partition function

The partition function $p(x)$ counts show many ways there are to write the integer $x$ as a sum of integers.

| x | 36 | 37 | 38 | 39 | 40 | 41 | 42 |
|---|---|---|---|---|---|---|---|
| p(x) | 17.977 | 21.637 | 26.015 | 31.185 | 37.338 | 44.583 | 53.174 |
| x | 43 | 44 | 45 | 46 | 47 | 100 | |
| p(x) | 63.261 | 75.175 | 89.134 | 105.558 | 125.754 | 190.569.292 | |

### 7.12 Catalan numbers

Catalan numbers are defined by the recurrence:

$$C_{n+1} = \sum_{i=0}^{n} C_i C_{n-i}$$

A closed formula for Catalan numbers is:

$$C_n = \frac{1}{n+1}\binom{2n}{n} = \binom{2n}{n} - \binom{2n}{n+1}$$

### 7.13 Stirling numbers of the first kind

These are the number of permutations of $I_n$ with exactly $k$ disjoint cycles. They obey the recurrence:

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

### 7.14 Stirling numbers of the second kind

These are the number of ways to partition $I_n$ into exactly $k$ sets. They obey the recurrence:

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

A "closed" formula for it is:

$$\begin{Bmatrix} n \\ k \end{Bmatrix} = \frac{1}{k!}\sum_{j=0}^{k}(-1)^{k-j}\binom{k}{j}j^n$$

### 7.15 Bell numbers

These count the number of ways to partition $I_n$ into subsets. They obey the recurrence:

$$\mathcal{B}_{n+1} = \sum_{k=0}^{n}\binom{n}{k}\mathcal{B}_k$$

| x | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 |
|---|---|---|---|---|---|---|---|---|
| $\mathcal{B}_x$ | 52 | 203 | 877 | 4.140 | 21.147 | 115.975 | 678.570 | 4.213.597 |

### 7.16 Turán's theorem

No graph with $n$ vertices that is $K_{r+1}$-free can have more edges than the Turán graph: A $k$-partite complete graph with sets of size as equal as possible.

### 7.17 Generating functions

A list of generating functions for useful sequences:

| | |
|---|---|
| $(1,1,1,1,1,1,\ldots)$ | $\frac{1}{1-z}$ |
| $(1,-1,1,-1,1,-1,\ldots)$ | $\frac{1}{1+z}$ |
| $(1,0,1,0,1,0,\ldots)$ | $\frac{1}{1-z^2}$ |
| $(1,0,\ldots,0,1,0,1,0,\ldots,0,1,0,\ldots)$ | $\frac{1}{1-z^2}$ |
| $(1,2,3,4,5,6,\ldots)$ | $\frac{1}{(1-z)^2}$ |
| $(1,\binom{m+1}{m},\binom{m+2}{m},\binom{m+3}{m},\ldots)$ | $\frac{1}{(1-z)^{m+1}}$ |
| $(1,c,\binom{c+1}{2},\binom{c+2}{3},\ldots)$ | $\frac{1}{(1-z)^c}$ |
| $(1,c,c^2,c^3,\ldots)$ | $\frac{1}{1-cz}$ |
| $(0,1,\frac{1}{2},\frac{1}{3},\frac{1}{4},\ldots)$ | $\ln\frac{1}{1-z}$ |

A neat manipulation trick is:

$$\frac{1}{1-z}G(z) = \sum_n \sum_{k\le n} g_k z^n$$

### 7.18 Polyominoes

How many free (rotation, reflection), one-sided (rotation) and fixed $n$-ominoes are there?

| n | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 |
|---|---|---|---|---|---|---|---|---|
| free | 2 | 5 | 12 | 35 | 108 | 369 | 1.285 | 4.655 |
| one-sided | 2 | 7 | 18 | 60 | 196 | 704 | 2.500 | 9.189 |
| fixed | 6 | 19 | 63 | 216 | 760 | 2.725 | 9.910 | 36.446 |

### 7.19 The twelvefold way (from Stanley)

How many functions $f\colon N \to X$ are there?

| $N$ | $X$ | Any $f$ | Injective | Surjective |
|---|---|---|---|---|
| dist. | dist. | $x^n$ | $(x)_n$ | $x!\left\{{n \atop x}\right\}$ |
| indist. | dist. | $\binom{x+n-1}{n}$ | $\binom{x}{n}$ | $\binom{n-1}{n-x}$ |
| dist. | indist. | $\left\{{n \atop 1}\right\} + \ldots + \left\{{n \atop x}\right\}$ | $[n \leq x]$ | $\left\{{n \atop k}\right\}$ |
| indist. | indist. | $p_1(n) + \ldots p_x(n)$ | $[n \leq x]$ | $p_x(n)$ |

Where $\binom{a}{b} = \frac{1}{b!}(a)_b$ and $p_x(n)$ is the number of ways to partition the integer $n$ using $x$ summands.

### 7.20 Common integral substitutions

And finally, a list of common substitutions:

| | | |
|---|---|---|
| $\int F(\sqrt{ax+b})dx$ | $u = \sqrt{ax+b}$ | $\frac{2}{a}\int uF(u)du$ |
| $\int F(\sqrt{a^2-x^2})dx$ | $x = a\sin u$ | $a\int F(a\cos u)\cos u du$ |
| $\int F(\sqrt{x^2+a^2})dx$ | $x = a\tan u$ | $a\int F(a\sec u)\sec^2 u du$ |
| $\int F(\sqrt{x^2-a^2})dx$ | $x = a\sec u$ | $a\int F(a\tan u)\sec u\tan u du$ |
| $\int F(e^{ax})dx$ | $u = e^{ax}$ | $\frac{1}{a}\int \frac{F(u)}{u}du$ |
| $\int F(\ln x)dx$ | $u = \ln x$ | $\int F(u)e^u du$ |

### 7.21 Table of non-trigonometric integrals

Some useful integrals are:

| | |
|---|---|
| $\int \frac{dx}{x^2+a^2}$ | $\frac{1}{a}\arctan\frac{x}{a}$ |
| $\int \frac{dx}{x^2-a^2}$ | $\frac{1}{2a}\ln\frac{x-a}{x+a}$ |
| $\int \frac{dx}{a^2-x^2}$ | $\frac{1}{2a}\ln\frac{a+x}{a-x}$ |
| $\int \frac{dx}{\sqrt{a^2-x^2}}$ | $\arcsin\frac{x}{a}$ |
| $\int \frac{dx}{\sqrt{x^2-a^2}}$ | $\ln\left(u+\sqrt{x^2-a^2}\right)$ |
| $\int \frac{dx}{x\sqrt{x^2-a^2}}$ | $\frac{1}{a}\operatorname{arcsec}\left\lvert\frac{u}{a}\right\rvert$ |
| $\int \frac{dx}{x\sqrt{x^2+a^2}}$ | $-\frac{1}{a}\ln\left(\frac{a+\sqrt{x^2+a^2}}{x}\right)$ |
| $\int \frac{dx}{x\sqrt{a^2+x^2}}$ | $-\frac{1}{a}\ln\left(\frac{a+\sqrt{a^2-x^2}}{x}\right)$ |

### 7.22 Table of trigonometric integrals

A list of common and not-so-common trigonometric integrals:

| | |
|---|---|
| $\int \tan x dx$ | $-\ln\lvert\cos x\rvert$ |
| $\int \cot x dx$ | $\ln\lvert\sin x\rvert$ |
| $\int \sec x dx$ | $\ln\lvert\sec x+\tan x\rvert$ |
| $\int \csc x dx$ | $\ln\lvert\csc x-\cot x\rvert$ |
| $\int \sec^2 x dx$ | $\tan x$ |
| $\int \csc^2 x dx$ | $\cot x$ |
| $\int \sin^n x dx$ | $\frac{-\sin^{n-1}x\cos x}{n} + \frac{n-1}{n}\int \sin^{n-2}x dx$ |
| $\int \cos^n x dx$ | $\frac{\cos^{n-1}x\sin x}{n} + \frac{n-1}{n}\int \cos^{n-2}x dx$ |
| $\int \arcsin x dx$ | $x\arcsin x + \sqrt{1-x^2}$ |
| $\int \arccos x dx$ | $x\arccos x - \sqrt{1-x^2}$ |
| $\int \arctan x dx$ | $x\arctan x - \frac{1}{2}\ln\lvert 1-x^2\rvert$ |

### 7.23 Centroid of a polygon

The x coordinate of the centroid of a polygon is given by $\frac{1}{3A}\sum_{i=0}^{n-1}(x_i+x_{i+1})(x_iy_{i+1}-x_{i+1}y_i)$, where $A$ is twice the signed area of the polygon.