SamuellH12 - ICPC Library

Conteúdo

1 Data Structures

1.1 BIT

```cpp
struct BIT {
  vector<int> bit;
  int N;

  BIT(){}
  BIT(int n) : N(n+1), bit(n+1){}

  void update(int pos, int val){
    for(; pos < N; pos += pos&(-pos))
      bit[pos] += val;
  }

  int query(int pos){
    int sum = 0;
    for(; pos > 0; pos -= pos&(-pos))
      sum += bit[pos];
    return sum;
  }
};
```

1.2 BIT2D

```
Complexity: O(Log^2 N)
Bit.update(x, y, v);  //Adiciona +v na posicao {x, y} da BIT
Bit.query(x, y);    //Retorna o somatorio do retangulo de
    inicio {1, 1} e fim {x, y}
Bit.queryArea(xi, yi, xf, yf);     //Retorna o somatorio do
    retangulo de inicio {xi, yi} e fim {xf, yf}
Bit.updateArea(xi, yi, xf, yf, v); //adiciona +v no retangulo
    de inicio {xi, yi} e fim {xf, yf}

IMPORTANTE! UpdateArea NAO atualiza o valor de todas as
    celulas no retangulo!!! Deve ser usado para Color Update
IMPORTANTE! Use query(x, y) Para acessar o valor da posicao
    (x, y) quando estiver usando UpdateArea
IMPORTANTE! Use queryArea(x, y, x, y) Para acessar o valor da
    posicao (x, y) quando estiver usando Update Padrao
```

```cpp
const int MAXN = 1e3 + 5;

struct BIT2D {
  int bit[MAXN][MAXN];

  void update(int X, int Y, int val){
    for(int x = X; x < MAXN; x += x&(-x))
      for(int y = Y; y < MAXN; y += y&(-y))
        bit[x][y] += val;
  }

  int query(int X, int Y){
    int sum = 0;
    for(int x = X; x > 0; x -= x&(-x))
      for(int y = Y; y > 0; y -= y&(-y))
        sum += bit[x][y];
    return sum;
  }

  void updateArea(int xi, int yi, int xf, int yf, int val){
    update(xi,   yi,    val);
    update(xf+1, yi,   -val);
    update(xi,   yf+1, -val);
    update(xf+1, yf+1,  val);
  }

  int queryArea(int xi, int yi, int xf, int yf){
    return query(xf, yf) - query(xf, yi-1) - query(xi-1, yf) +
        query(xi-1, yi-1);
  }
};
```

1.3 BIT2DSparse

```
Sparse Binary Indexed Tree 2D

Recebe o conjunto de pontos que serao usados para fazer os
    updates e as queries e cria uma BIT 2D esparsa que
    independe do "tamanho do grid".

Build: O(N Log N)  (N -> Quantidade de Pontos)
Query/Update: O(Log N)
IMPORTANTE! Offline!

BIT2D(pts);   // pts -> vecotor<pii> com todos os pontos em
    que serao feitas queries ou updates
```

```cpp
#define upper(v, x) (upper_bound(begin(v), end(v), x) - begin(
    v))

struct BIT2D {
  vector<int> ord;
  vector<vector<int>> bit, coord;

  BIT2D(vector<pii> pts){
    sort(begin(pts), end(pts));

    for(auto [x, y] : pts)
      if(ord.empty() || x != ord.back())
        ord.push_back(x);

    bit.resize(ord.size() + 1);
    coord.resize(ord.size() + 1);

    sort(begin(pts), end(pts), [&](pii &a, pii &b){
      return a.second < b.second;
    });

    for(auto [x, y] : pts)
      for(int i=upper(ord, x); i < bit.size(); i += i&-i)
        if(coord[i].empty() || coord[i].back() != y)
          coord[i].push_back(y);

    for(int i=0; i<bit.size(); i++) bit[i].assign(coord[i].
        size()+1, 0);
  }

  void update(int X, int Y, int v){
    for(int i = upper(ord, X); i<bit.size(); i += i&-i)
      for(int j = upper(coord[i], Y); j < bit[i].size(); j +=
          j&-j)
        bit[i][j] += v;
  }

  int query(int X, int Y){
    int sum = 0;
    for(int i = upper(ord, X); i > 0; i -= i&-i)
      for(int j = upper(coord[i], Y); j > 0; j -= j&-j)
        sum += bit[i][j];
```

```cpp
    return sum;
  }

  void updateArea(int xi, int yi, int xf, int yf, int val){
    update(xi,   yi,    val);
    update(xf+1, yi,   -val);
    update(xi,   yf+1, -val);
    update(xf+1, yf+1,  val);
  }

  int queryArea(int xi, int yi, int xf, int yf){
    return query(xf, yf) - query(xf, yi-1) - query(xi-1, yf) +
         query(xi-1, yi-1);
  }
};
```

1.4 PrefixSum2D

```cpp
const int MAXN = 1e3 + 5;
int ps [MAXN][MAXN];

void calcPS2d(){
  for (int i = 1; i < MAXN; i++) ps[0][i] += ps[0][i - 1];  //
     inicializo a 1a linha
  for (int i = 1; i < MAXN; i++) ps[i][0] += ps[i - 1][0];  //
     inicializo a 1a coluna

  for (int i = 1; i < MAXN; i++)
    for (int j = 1; j < MAXN; j++)
      ps[i][j] += ps[i - 1][j] + ps[i][j - 1] - ps[i - 1][j -
         1];
}
int queryPS2d(int xi, int yi, int xf, int yf){ return ps[xf][
    yf] - ps[xf][yi-1] - ps[xi-1][yf] + ps[xi-1][yi-1]; }
```

1.5 SegTree

```cpp
const int MAXN = 1e6 + 5;
int seg[4*MAXN];

int query(int no, int l, int r, int a, int b){
  if(b <  l || r <  a) return 0;
  if(a <= l && r <= b) return seg[no];

  int m=(l+r)/2, e=no*2, d=no*2+1;

  return query(e, l, m, a, b) + query(d, m+1, r, a, b);
}

void update(int no, int l, int r, int pos, int v){
  if(pos < l || r < pos) return;
  if(l == r){seg[no] = v; return; }

  int m=(l+r)/2, e=no*2, d=no*2+1;

  update(e, l,   m, pos, v);
  update(d, m+1, r, pos, v);

  seg[no] = seg[e] + seg[d];
}

void build(int no, int l, int r, vector<int> &lista){
  if(l == r){ seg[no] = lista[l]; return; }

  int m=(l+r)/2, e=no*2, d=no*2+1;

  build(e, l,   m, lista);
  build(d, m+1, r, lista);

  seg[no] = seg[e] + seg[d];
}
```

1.6 SegTreeLazy

```cpp
const int MAXN = 1e6 + 5;
int seg[4*MAXN];
int lazy[4*MAXN];

void unlazy(int no, int l, int r){
  if(lazy[no] == 0) return;

  int m=(l+r)/2, e=no*2, d=no*2+1;

  seg[no] += (r-l+1) * lazy[no];

  if(l != r){
    lazy[e] += lazy[no];
    lazy[d] += lazy[no];
  }

  lazy[no] = 0;
}

int query(int no, int l, int r, int a, int b){
  unlazy(no, l, r);
  if(b <  l || r <  a) return 0;
  if(a <= l && r <= b) return seg[no];

  int m=(l+r)/2, e=no*2, d=no*2+1;

  return query(e, l, m, a, b) + query(d, m+1, r, a, b);
}

void update(int no, int l, int r, int a, int b, int v){
  unlazy(no, l, r);
  if(b <  l || r <  a) return;
  if(a <= l && r <= b)
  {
    lazy[no]+= v;
    unlazy(no, l, r);
    return;
  }

  int m=(l+r)/2, e=no*2, d=no*2+1;

  update(e, l,   m, a, b, v);
  update(d, m+1, r, a, b, v);

  seg[no] = seg[e] + seg[d];
}

void build(int no, int l, int r, vector<int> &lista){
  if(l == r){ seg[no] = lista[l-1]; return; }

  int m=(l+r)/2, e=no*2, d=no*2+1;

  build(e, l,   m, lista);
  build(d, m+1, r, lista);

  seg[no] = seg[e] + seg[d];
}
-> Segment Tree - Lazy Propagation com:
  - Query em Range
  - Update em Range

build (1, 1, n, lista);
query (1, 1, n, a, b);
update(1, 1, n, a, b, x);

|   n    | o tamanho maximo da lista
| [a, b] | o intervalo da busca ou update
|   x    | o novo valor a ser somada no intervalo [a, b]
| lista  | o array de elementos originais

Build:  O(N)
Query:  O(log N)
Update: O(log N)
Unlazy: O(1)
```

1.7 SegTree Persistente

```
-> Segment Tree Persistente: (2x mais rapido que com ponteiro)
Build(1, N) -> Cria uma Seg Tree completa de tamanho N;
     RETORNA o NodeId da Raiz
Update(Root, pos, v) -> Soma +V em POS; RETORNA o NodeId da
     nova Raiz;
Query(Root, a, b) -> RETORNA o valor do range [a, b];
Kth(RootL, RootR, K) -> Faz uma Busca Binaria na Seg de
     diferenca entre as duas versoes.
[ Root -> No Raiz da Versao da Seg na qual se quer realizar a
     operacao ]

Build:  O(N)  !!! Sempre chame o Build
Query:  O(log N)
Update: O(log N)
Kth:  O(Log N)
```

```cpp
const int MAXN = 1e5 + 5;
const int MAXLOG = 31 - __builtin_clz(MAXN) + 1;
typedef int NodeId;
typedef int STp;

const STp NEUTRO = 0;
int IDN, LSEG, RSEG;
extern struct Node NODES[];

struct Node {
  STp val;
  NodeId L, R;
  Node(STp v = NEUTRO) : val(v), L(-1), R(-1) {}
  Node& l(){ return NODES[L]; }
  Node& r(){ return NODES[R]; }
};

Node NODES[4*MAXN + MAXLOG*MAXN]; //!!!CUIDADO COM O TAMANHO (
     aumente se necessario)
pair<Node&, NodeId> newNode(STp v = NEUTRO){ return {NODES[IDN
     ] = Node(v), IDN++}; }

STp join(STp lv, STp rv){ return lv + rv; }

NodeId build(int l, int r, bool root=true){
  if(root) LSEG = l, RSEG = r;
  if(l == r) return newNode().second;

  int m = (l+r)/2;
  auto [node, id] = newNode();
```

```cpp
  node.L = build(l,   m, false);
  node.R = build(m+1, r, false);
  node.val = join(node.l().val, node.r().val);

  return id;
}

NodeId update(NodeId node, int l, int r, int pos, int v){
  if( pos < l || r < pos ) return node;
  if(l == r) return newNode(NODES[node].val + v).second;

  int m = (l+r)/2;
  auto [nw, id] =newNode();

  nw.L = update(NODES[node].L, l,   m, pos, v);
  nw.R = update(NODES[node].R, m+1, r, pos, v);

  nw.val = join(nw.l().val, nw.r().val);

  return id;
}
NodeId update(NodeId node, int pos, STp v){ return update(node
     , LSEG, RSEG, pos, v); }

int query(Node& node, int l, int r, int a, int b){
  if(b <  l || r <  a) return NEUTRO;
  if(a <= l && r <= b) return node.val;

  int m = (l+r)/2;

  return join(query(node.l(), l, m, a, b), query(node.r(), m
      +1, r, a, b));
}
int query(NodeId node, int a, int b){ return query(NODES[node
    ], LSEG, RSEG, a, b); }

int kth(Node& Left, Node& Right, int l, int r, int k){
  if(l == r) return l;

  int sum =Right.l().val - Left.l().val;
  int m = (l+r)/2;

  if(sum >= k) return kth(Left.l(), Right.l(), l, m, k);
  return kth(Left.r(), Right.r(), m+1, r, k - sum);
}
int kth(NodeId Left, NodeId Right, int k){ return kth(NODES[
     Left], NODES[Right], LSEG, RSEG, k); }
```

1.8 SegTree Iterativa

```cpp
template<typename T> struct SegTree {
  int n;
  vector<T> seg;
  T join(T&l, T&r){ return l + r; }

  void init(vector<T>&base){
    n = base.size();
    seg.resize(2*n);
    for(int i=0; i<n; i++) seg[i+n] = base[i];
    for(int i=n-1; i>0; i--) seg[i] = join(seg[i*2], seg[i
         *2+1]);
  }

  T query(int l, int r){ //[L, R] & [0, n-1]
    T ans = 0; //NEUTRO //if order matters, change to l_ans,
         r_ans
    for(l+=n, r+=n+1; l<r; l/=2, r/=2){
      if(l&1) ans = join(ans, seg[l++]);
      if(r&1) ans = join(seg[--r], ans);
    }
    return ans;
  }

  void update(int i, T v){ // Set Value seg[i+=n] = v //
       change to += v to sum
    for(seg[i+=n] = v; i/=2;) seg[i] = join(seg[i*2], seg[i
         *2+1]);
  }
};
```

1.9 SegTree Lazy Iterativa

```cpp
template<typename T> struct SegTree {
  int n, h;
  vector<T> seg, lzy;
  vector<int> sz;
  T join(T&l, T&r){ return l + r; }

  void init(int _n){
    n = _n;
    h = 32 - __builtin_clz(n);
    seg.resize(2*n);
    lzy.resize(n);
    sz.resize(2*n, 1);
    for(int i=n-1; i; i--) sz[i] = sz[i*2] + sz[i*2+1];
    // for(int i=0; i<n; i++) seg[i+n] = base[i];
    // for(int i=n-1; i; i--) seg[i] = join(seg[i*2], seg[i
         *2+1]);
  }

  void apply(int p, T v){
    seg[p] += v * sz[p];
    if(p < n) lzy[p] += v;
  }
  void push(int p){
    for(int s=h, i=p>>s; s; s--, i=p>>s)
      if(lzy[i] != 0) {
        apply(i*2,   lzy[i]);
        apply(i*2+1, lzy[i]);
        lzy[i] = 0; //NEUTRO
      }
  }
  void build(int p) {
    for(p/=2; p; p/= 2){
      seg[p] = join(seg[p*2], seg[p*2+1]);
      if(lzy[p] != 0) seg[p] += lzy[p] * sz[p];
    }
  }

  T query(int l, int r){ //[L, R] & [0, n-1]
    l+=n, r+=n+1;
    push(l); push(r-1);

    T ans = 0; //NEUTRO
    for(; l<r; l/=2, r/=2){
      if(l&1) ans = join(seg[l++], ans);
      if(r&1) ans = join(ans, seg[--r]);
    }
    return ans;
  }

  void update(int l, int r, T v){
    l+=n, r+=n+1;
    push(l); push(r-1);
    int l0 = l, r0 = r;
    for(; l<r; l/=2, r/=2){
      if(l&1) apply(l++, v);
      if(r&1) apply(--r, v);
    }
    build(l0); build(r0-1);
  }
};
```

1.10 SparseTable

```cpp
const int MAXN = 1e5 + 5;
const int MAXLG = 31 - __builtin_clz(MAXN) + 1;

int table[MAXLG][MAXN];

void build(vector<int> &v){
  int N = v.size();
  for(int i=0; i<N; i++) table[0][i] = v[i];

  for(int p=1; p < MAXLG; p++)
    for(int i=0; i + (1 << p) <= N; i++)
      table[p][i] = min(table[p-1][i], table[p-1][i+(1 << (p
           -1))]);
}

int query(int l, int r){
  int p = 31 - __builtin_clz(r - l + 1);  //floor log
  return min(table[p][l], table[p][ r - (1<<p) + 1 ]);
}
Sparse Table for Range Minimum Query [L, R] [0, N)
build:  O(N log N)
Query:  O(1)
Value -> Original Array
```

2 dp

2.1 Digit DP

```
Digit DP - Sum of Digits

Solve(K)  ->  Retorna a soma dos digitos de todo numero X tal
     que: 0 <= X <= K
dp[D][S][f]  ->  D: Quantidade de digitos; S: Soma dos
     digitos; f: Flag que indica o limite.
int limite[D] -> Guarda os digitos de K.
```

Complexity: O(D^2 * B^2) (B = Base = 10)

```cpp
ll dp[2][19][170];

int limite[19];
ll digitDP(int idx, int sum, bool flag){
    if(idx < 0) return sum;
    if(~dp[flag][idx][sum]) return dp[flag][idx][sum];

    dp[flag][idx][sum] = 0;
  int lm = flag ? limite[idx] : 9;

    for(int i=0; i<=lm; i++)
      dp[flag][idx][sum] += digitDP(idx-1, sum+i, (flag && i
           == lm));
```

```
    return dp[flag][idx][sum];
}

ll solve(ll k){
    memset(dp, -1, sizeof dp);

  int sz=0;
  while(k){
    limite[sz++] = k % 10LL;
    k /= 10LL;
  }

  return digitDP(sz-1, 0, true);
}
```

2.2 LCS

```
LCS - Longest Common Subsequence

Complexity: O(N^2)

* Recursive: memset(memo, -1, sizeof memo); LCS(0, 0);
* Iterative: LCS_It();

* RecoverLCS O(N)
  Recover just one of all the possible LCS
```

```
const int MAXN = 5*1e3 + 5;
int memo[MAXN][MAXN];

string s, t;

inline int LCS(int i, int j){
  if(i == s.size() || j == t.size()) return 0;
  if(memo[i][j] != -1) return memo[i][j];

  if(s[i] == t[j]) return memo[i][j] = 1 + LCS(i+1, j+1);

  return memo[i][j] = max(LCS(i+1, j), LCS(i, j+1));
}

int LCS_It(){
  for(int i=s.size()-1; i>=0; i--)
    for(int j=t.size()-1; j>=0; j--)
      if(s[i] == t[j])
        memo[i][j] = 1 + memo[i+1][j+1];
      else
        memo[i][j] = max( memo[i+1][j], memo[i][j+1] );

  return memo[0][0];
}

string RecoverLCS(int i, int j){
  if(i == s.size() || j == t.size()) return "";

  if(s[i] == t[j]) return s[i] + RecoverLCS(i+1, j+1);

  if(memo[i+1][j] > memo[i][j+1]) return RecoverLCS(i+1, j);

  return RecoverLCS(i, j+1);
}
```

2.3 LIS

```
LIS - Longest Increasing Subsequence

Complexity: O(N Log N)
* For ICREASING sequence, use lower_bound()
* For NON DECREASING sequence, use upper_bound()
```

```
int LIS(vector<int>& nums){
  vector<int> lis;

  for(auto x : nums)
  {
    auto it = lower_bound(lis.begin(), lis.end(), x);

    if(it == lis.end()) lis.push_back(x);
    else *it = x;
  }

  return (int) lis.size();
}
```

2.4 SOS DP

```
SOS DP - Sum over Subsets

Dado que cada mask possui um valor inicial (iVal), computa
para cada mask a soma dos valores de todas as suas submasks.

N -> Numero Maximo de Bits
iVal[mask] -> initial Value / Valor Inicial da Mask
dp[mask]   -> Soma de todos os SubSets

Iterar por todas as submasks: for(int sub=mask; sub>0;
     sub=(sub-1)&mask)
```

```
const int N = 20;
ll dp[1<<N], iVal[1<<N];

void sosDP(){ // O(N * 2^N)
    for(int i=0; i<(1<<N); i++)
        dp[i] = iVal[i];

  for(int i=0; i<N; i++)
    for(int mask=0; mask<(1<<N); mask++)
      if(mask&(1<<i))
        dp[mask] += dp[mask^(1<<i)];
}

void sosDPsub(){ // O(3^N) //suboptimal
  for (int mask = 0, i; mask < (1<<N); mask++)
    for(i = mask, dp[mask] = iVal[0]; i>0; i=(i-1) & mask) //
         iterate over all submasks
      dp[mask] += iVal[i];
}
```

---

3 Geometry

3.1 ConvexHull

```
struct PT {
  ll x, y;
  PT(ll x=0, ll y=0) : x(x), y(y) {}

  PT operator- (const PT&a) const{ return PT(x-a.x, y-a.y); }
  ll operator% (const PT&a) const{ return  (x*a.y - y*a.x); }
      //Cross // Vector product

  bool operator==(const PT&a) const{ return x == a.x && y == a
      .y; }
  bool operator< (const PT&a) const{ return x != a.x ? x < a.x
       : y < a.y; }
};

// Colinear? Mude >= 0 para > 0 nos while
vector<PT> ConvexHull(vector<PT> pts, bool sorted=false) {
  if(!sorted) sort(begin(pts), end(pts));
  pts.resize(unique(begin(pts), end(pts)) - begin(pts));
  if(pts.size() <= 1) return pts;

  int s=0, n=pts.size();
  vector<PT> h (2*n+1);

  for(int i=0; i<n; h[s++] = pts[i++])
    while(s > 1 && (pts[i] - h[s-2]) % (h[s-1] - h[s-2]) >= 0
        )
      s--;

  for(int i=n-2, t=s; ~i; h[s++] = pts[i--])
    while(s > t && (pts[i] - h[s-2]) % (h[s-1] - h[s-2]) >= 0
        )
      s--;

  h.resize(s-1);
  return h;
}
// FOR DOUBLE POINT //
See Geometry - General
```

3.2 Geometry - General

```
#define ld long double

// !!! NOT TESTED !!! //

struct PT {
  ll x, y;
  PT(ll x=0, ll y=0) : x(x), y(y) {}

  PT operator+ (const PT&a) const{ return PT(x+a.x, y+a.y); }
  PT operator- (const PT&a) const{ return PT(x-a.x, y-a.y); }
  ll operator* (const PT&a) const{ return  (x*a.x + y*a.y); }
      //DOT product   // norm // lenght^2 // inner
  ll operator% (const PT&a) const{ return  (x*a.y - y*a.x); }
      //Cross // Vector product
  PT operator* (ll c)  const{ return PT(x*c, y*c); }
  PT operator/ (ll c)  const{ return PT(x/c, y/c); }

  bool operator==(const PT&a) const{ return x == a.x && y == a
      .y; }
  bool operator< (const PT&a) const{ return x != a.x ? x < a.x
       : y < a.y; }
  bool operator<<(const PT&a) const{ PT p=*this; return (p%a
      == 0) ? (p*p < a*a) : (p%a < 0);  } //angle(p) < angle(a
      )
};

// FOR DOUBLE POINT //
```

```cpp
const ld EPS = 1e-9;
bool eq(ld a, ld b){ return abs(a-b) < EPS; } // ==
bool lt(ld a, ld b){ return a + EPS < b;    } // <
bool gt(ld a, ld b){ return a > b + EPS;    } // >
bool le(ld a, ld b){ return a < b + EPS;    } // <=
bool ge(ld a, ld b){ return a + EPS > b;    } // >=
bool operator==(const PT&a) const{ return eq(x, a.x) && eq(y,
    a.y); }       // for double point
bool operator< (const PT&a) const{ return eq(x, a.x) ? lt(y, a
    .y) : lt(x, a.x); } // for double point
bool operator<<(PT&a){ PT&p=*this; return eq(p%a, 0) ? lt(p*p,
    a*a) : lt(p%a, 0);  } //angle(this) < angle(a)
//Change LL to LD and uncomment this
//Also, consider replacing comparisons with these functions

ld dist  (PT a, PT b){ return sqrtl((a-b)*(a-b)); }
                          // distance from A to B
ld angle (PT a, PT b){ return acos((a*b) / sqrtl(a*a) / sqrtl(
    b*b)); } //Angle between A and B
PT rotate(PT p, double ang){ return PT(p.x*cos(ang) - p.y*sin(
    ang), p.x*sin(ang) + p.y*cos(ang)); } //Left rotation.
    Angle in radian

ll Area(vector<PT>& p){
  ll area = 0;
  for(int i=2; i < p.size(); i++)
    area += (p[i]-p[0]) % (p[i-1]-p[0]);
  return abs(area) / 2LL;
}

PT intersect(PT a1, PT d1, PT a2, PT d2){
  return a1 + d1 * (((a2 - a1)%d2) / (d1%d2));
}

ld dist_pt_line(PT a, PT l1, PT l2){
  return abs( ((a-l1) % (l2-l1)) / dist(l1, l2)  );
}

ld dist_pt_segm(PT a, PT s1, PT s2){
  if(s1 == s2) return dist(s1, a);

  PT d = s2 - s1;
  ld t = max(0.0L, min(1.0L, ((a-s1)*d) / sqrtl(d*d)) );

  return dist(a, s1+(d*t));
}
```

3.3 LineContainer

```cpp
struct Line {
  mutable ll k, m, p;
  bool operator<(const Line& o) const { return k < o.k; }
  bool operator<(ll x) const { return p < x; }
};

struct LineContainer : multiset<Line, less<>> {
  static const ll inf = LLONG_MAX; // Double: inf = 1/.0, div(
      a,b) = a/b
  ll div(ll a, ll b) { return a / b - ((a ^ b) < 0 && a % b);
      } //floored division

  bool isect(iterator x, iterator y) {
    if(y == end()) return x->p = inf, 0;
    if(x->k == y->k) x->p = x->m > y->m ? inf : -inf;
    else x->p = div(y->m - x->m, x->k - y->k);
    return x->p >= y->p;
  }

  void add_line(ll k, ll m){ // kx + m  //if minimum k*=-1, m
      *=-1, query*-1
    auto z = insert({k, m, 0}), y = z++, x = y;
    while(isect(y, z)) z = erase(z);
    if(x != begin() && isect(--x, y)) isect(x, y = erase(y));
    while((y = x) != begin() && (--x)->p >= y->p) isect(x,
        erase(y));
  }

  ll query(ll x) {
    assert(!empty());
    auto l = *lower_bound(x);
    return l.k * x + l.m;
  }
};
```

4 Grafos

4.1 2-SAT

```
2 SAT - Two Satisfiability Problem

IMPORTANTE! o grafo deve estar 0-indexado!
Retorna uma valoracao verdadeira se possivel ou um vetor
    vazio se impossivel;
inverso de u = ~u
```

```cpp
struct TwoSat {
  int N;
  vector<vector<int>> E;

  TwoSat(int N) : N(N), E(2 * N) {}
  inline int eval(int u) const{ return u < 0 ? ((~u)+N)%(2*N)
      : u; }

  void add_or(int u, int v){
    E[eval(~u)].push_back(eval(v));
    E[eval(~v)].push_back(eval(u));
  }
  void add_nand(int u, int v) {
    E[eval(u)].push_back(eval(~v));
    E[eval(v)].push_back(eval(~u));
  }
  void set_true (int u){ E[eval(~u)].push_back(eval(u)); }
  void set_false(int u){ set_true(~u); }
  void add_imply(int u, int v){ E[eval(u)].push_back(eval(v));
      }
  void add_and  (int u, int v){ set_true(u);  set_true(v);
      }
  void add_nor  (int u, int v){ add_and(~u, ~v); }
  void add_xor  (int u, int v){ add_or(u, v); add_nand(u, v);
      }
  void add_xnor (int u, int v){ add_xor(u, ~v); }

  vector<bool> solve() {
    vector<bool> ans(N);
    auto scc = tarjan();

    for (int u = 0; u < N; u++)
      if(scc[u] == scc[u+N]) return {}; //false
      else ans[u] = scc[u+N] > scc[u];

    return ans; //true
  }
private:
  vector<int> tarjan() {
    vector<int> low(2*N), pre(2*N, -1), scc(2*N, -1);
    stack<int> st;
    int clk = 0, ncomps = 0;

    auto dfs = [&](auto&& dfs, int u) -> void {
      pre[u] = low[u] = clk++;
      st.push(u);

      for(auto v : E[u])
        if(pre[v] == -1) dfs(dfs, v), low[u] = min(low[u], low
            [v]);
        else
        if(scc[v] == -1) low[u] = min(low[u], pre[v]);

      if(low[u] == pre[u]){
        int v = -1;
        while(v != u) scc[v = st.top()] = ncomps, st.pop();
        ncomps++;
      }
    };

    for(int u=0; u < 2*N; u++)
      if(pre[u] == -1)
        dfs(dfs, u);

    return scc; //tarjan SCCs order is the reverse of topoSort
        , so (u->v if scc[v] <= scc[u])
  }
};
```

4.2 BlockCut Tree

```
Block Cut Tree - BiConnected Component

reset(n);
addEdge(u, v);
tarjan(Root);
buildBCC(n);

No fim o grafo da Block Cut Tree estara em _vector<int> tree
    []_
```

```cpp
const int MAXN = 1e6 + 5;
const int MAXM = 1e6 + 5;//Cuidado

vector<pii> grafo [MAXN];
int pre[MAXN], low[MAXN], clk=0, C=0;

vector<pii> edge;
bool visEdge[MAXM];
int edgeComponent[MAXM];
int vertexComponent[MAXN];

int cut[MAXN];
stack<int> s;

vector<int> tree [2*MAXN];
int componentSize[2*MAXN]; //vertex - cutPoints

void reset(int n){
  for(int i=0; i<=edge.size(); i++)
    visEdge[i] = edgeComponent[i] = 0;

  edge.clear();
```

```
  for(int i=0; i<=n; i++){
    pre[i] = low[i] = -1;
    cut[i] = false;
    vertexComponent[i] = 0;
    grafo[i].clear();
  }

  for(int i=0; i<=C; i++){
    componentSize[i] = 0;
    tree[i].clear();
  }

  while(!s.empty()) s.pop();

  clk = C = 0;
}

void newComponent(int i){
  C++;
  int j;

  do {
    j = s.top(); s.pop();
    edgeComponent[j] = C;

    auto [u, v]  = edge[j];
    if(!cut[u] && !vertexComponent[u]) componentSize[C]++,
        vertexComponent[u] = C;
    if(!cut[v] && !vertexComponent[v]) componentSize[C]++,
        vertexComponent[v] = C;

  } while(!s.empty() && j != i);
}

void tarjan(int u, bool root = true){
  pre[u] = low[u] = clk++;

  bool any = false;
  int chd = 0;

  for(auto [v, i] : grafo[u]){
    if(visEdge[i]) continue;
    visEdge[i] = true;

    s.emplace(i);

    if(pre[v] == -1)
    {
      tarjan(v, false);

      low[u] = min(low[v], low[u]);
      chd++;

      if(!root && low[v] >= pre[u]) cut[u] = true,
          newComponent(i);
      if( root && chd >= 2)       cut[u] = true, newComponent(
          i);
    }
    else
      low[u] = min(low[u], pre[v]);
  }

  if(root) newComponent(-1);
}

//ATENCAO: ESTA 1-INDEXADO
void buildBCC(int n){
  vector<bool> marc(C+1, false);

  for(int u=1; u<=n; u++)
  {
    if(!cut[u]) continue;

    C++;
    cut[u] = C;

    for(auto [v, i] : grafo[u])
    {
      int ec = edgeComponent[i];
      if(!marc[ec])
      {
        marc[ec] = true;
        tree[cut[u]].emplace_back(ec);
        tree[ec].emplace_back(cut[u]);
      }
    }

    for(auto [v, i] : grafo[u])
      marc[edgeComponent[i]] = false;
  }
}

void addEdge(int u, int v){
  int i = edge.size();
  grafo[u].emplace_back(v, i);
  grafo[v].emplace_back(u, i);
  edge.emplace_back(u, v);
}
```

4.3 CentroidDecomposition

```
Complexity: O(N*LogN)

dfsc() -> para criar a centroid tree

rem[u]    -> True se U ja foi removido (pra dfsc)
szt[u]    -> Size da subarvore de U (pra dfsc)
parent[u] -> Pai de U na centroid tree *parent[ROOT] = -1
distToAncestor[u][i] -> Distancia na arvore original de u para
seu i-esimo pai na centroid tree *distToAncestor[u][0] = 0

dfsc(u=node, p=parent(subtree), f=parent(centroid tree),
     sz=size of tree)
```

```
const int MAXN = 1e6 + 5;

vector<int> grafo[MAXN];
deque<int> distToAncestor[MAXN];

bool rem[MAXN];
int szt[MAXN], parent[MAXN];

void getDist(int u, int p, int d=0){
  for(auto v : grafo[u])
    if(v != p && !rem[v])
      getDist(v, u, d+1);
  distToAncestor[u].emplace_front(d);
}

int getSz(int u, int p){
  szt[u] = 1;
  for(auto v : grafo[u])
    if(v != p && !rem[v])
      szt[u] += getSz(v, u);
  return szt[u];
}

void dfsc(int u=0, int p=-1, int f=-1, int sz=-1){
  if(sz < 0) sz = getSz(u, -1); //starting new tree

  for(auto v : grafo[u])
    if(v != p && !rem[v] && szt[v]*2 >= sz)
      return dfsc(v, u, f, sz);

  rem[u] = true, parent[u] = f;
  getDist(u, -1, 0); //get subtree dists to centroid

  for(auto v : grafo[u])
    if(!rem[v])
      dfsc(v, u, u, -1);
}
```

4.4 Dijkstra

```
const int MAXN = 1e6 + 5;
#define INF 0x3f3f3f3f
#define vi vector<int>

vector<pii> grafo [MAXN];

vi dijkstra(int s){
  vi dist (MAXN, INF); // !!! Change MAXN to N

  priority_queue<pii, vector<pii>, greater<pii>> fila;
  fila.push({0, s});
  dist[s] = 0;

  while(!fila.empty())
  {
    auto [d, u] = fila.top();
    fila.pop();

    if(d > dist[u]) continue;

    for(auto [v, c] : grafo[u])
      if( dist[v] > dist[u] + c )
      {
        dist[v] = dist[u] + c;
        fila.push({dist[v], v});
      }
  }

  return dist;
}
```

```
Dijkstra - Shortest Paths from Source

caminho minimo de um vertice u para todos os
outros vertices de um grafo ponderado

Complexity: O(N Log N)

dijkstra(s)       ->  s : Source, Origem. As distancias serao
     calculadas com base no vertice s
grafo[u] = {v, c};    ->  u : Vertice inicial, v : Vertice
     final, c : Custo da aresta
priority_queue<pii, vector<pii>, greater<pii>> ->  Ordena pelo
     menor custo -> {d, v} -> d : Distancia, v : Vertice
```

4.5 Dinic

```
    Dinic - Max Flow Min Cut
Algoritmo de Dinitz para encontrar o Fluxo Maximo.
Casos de Uso em [Theorems/Flow]
IMPORTANTE! O algoritmo esta 0-indexado

Complexity:
O( V^2 * E )       ->  caso geral
O( sqrt(V) * E )  ->  grafos com cap = 1 para toda aresta //
    matching bipartido

* Informacoes:
  Crie o Dinic: Dinic dinic(n, source, sink);
  Adicione as Arestas: dinic.addAresta(u, v, capacity);
  Para calcular o Fluxo Maximo: dinic.maxFlow()
  Para saber se um vertice U esta no Corte Minimo:
     dinic.inCut(u)

* Sobre o Codigo:
  vector<Aresta> arestas;  -> Guarda todas as arestas do
     grafo e do grafo residual
  vector<vector<int>> adj; -> Guarda em adj[u] os indices de
     todas as arestas saindo de u
  vector<int> ptr;   ->  Pointer para a proxima aresta ainda
     nao visitada de cada vertice
  vector<int> level; -> Distancia em vertices a partir do
     Source. Se igual a N o vertice nao foi visitado.
  A BFS retorna se Sink e alcancavel de Source. Se nao e
     porque foi atingido o Fluxo Maximo
  A DFS retorna um possivel aumento do Fluxo
```

```cpp
struct Aresta {
  int u, v; ll cap;
  Aresta(int u, int v, ll cap) : u(u), v(v), cap(cap) {}
};

struct Dinic {

  int n, source, sink;
  vector<vector<int>> adj;
  vector<Aresta> arestas;
  vector<int> level, ptr; //pointer para a proxima aresta nao
      saturada de cada vertice

  Dinic(int n, int source, int sink) : n(n), source(source),
      sink(sink) { adj.resize(n); }

  void addAresta(int u, int v, ll cap)
  {
    adj[u].push_back(arestas.size());
    arestas.emplace_back(u, v, cap);

    adj[v].push_back(arestas.size());
    arestas.emplace_back(v, u, 0);
  }

  ll dfs(int u, ll flow = 1e9){
    if(flow == 0) return 0;
    if(u == sink) return flow;

    for(int &i = ptr[u];  i < adj[u].size();  i++)
    {
      int atual = adj[u][i];
      int v = arestas[atual].v;

      if(level[u] + 1 != level[v]) continue;

      if(ll got = dfs(v, min(flow, arestas[atual].cap)) )
      {
        arestas[atual].cap -= got;
        arestas[atual^1].cap += got;
        return got;
      }
    }

    return 0;
  }

  bool bfs(){
    level = vector<int> (n, n);
    level[source] = 0;

    queue<int> fila;
    fila.push(source);

    while(!fila.empty())
    {
      int u = fila.front();
      fila.pop();

      for(auto i : adj[u]){
        int v = arestas[i].v;

        if(arestas[i].cap == 0 || level[v] <= level[u] + 1 )
            continue;

        level[v] = level[u] + 1;
        fila.push(v);
      }
    }

    return level[sink] < n;
  }

  bool inCut(int u){ return level[u] < n; }

  ll maxFlow(){
    ll ans = 0;

    while( bfs() ){
      ptr = vector<int> (n+1, 0);

      while(ll got = dfs(source)) ans += got;
    }

    return ans;
  }
};
```

4.6 DSU Persistente

```
SemiPersistent Disjoint Set Union - O(Log n)
find(u, q) -> Retorna o pai de U no tempo q
* tim -> tempo em que o pai de U foi alterado
```

```cpp
struct DSUp {
  vector<int> pai, sz, tim;
  int t=1;
  DSUp(int n) : pai(n+1), sz(n+1, 1), tim(n+1) {
    for(int i=0; i<=n; i++) pai[i] = i;
  }

  int find(int u, int q = INT_MAX){
    if( pai[u] == u || q < tim[u] ) return u;
    return find(pai[u], q);
  }

  void join(int u, int v){
    u = find(u), v = find(v);

    if(u == v) return;
    if(sz[v] > sz[u]) swap(u, v);

    pai[v] = u;
    tim[v] = t++;
    sz[u] += sz[v];
  }
};
```

4.7 DSU Rollback

```
Disjoint Set Union with Rollback - O(Log n)
checkpoint() -> salva o estado atual
rollback() -> restaura no ultimo checkpoint
save another var? +save in join & +line in pop
```

```cpp
struct DSUr {
  vector<int> pai, sz, savept;
  stack<pair<int&, int>>  st;
  DSUr(int n) : pai(n+1), sz(n+1, 1) {
    for(int i=0; i<=n; i++) pai[i] = i;
  }

  int find(int u){ return pai[u] == u ? u : find(pai[u]); }

  void join(int u, int v){
    u = find(u), v = find(v);

    if(u == v) return;
    if(sz[v] > sz[u]) swap(u, v);

    save(pai[v]); pai[v] = u;
    save(sz[u]);  sz[u] += sz[v];
  }

  void save(int &x){ st.emplace(x, x); }
  void pop(){
    st.top().first = st.top().second; st.pop();
    st.top().first = st.top().second; st.pop();
  }

  void checkpoint(){ savept.push_back(st.size()); }
  void rollback(){
    while(st.size() > savept.back()) pop();
    savept.pop_back();
  }
};
```

4.8 DSU

```cpp
struct DSU {
  vector<int> pai, sz;
  DSU(int n) : pai(n+1), sz(n+1, 1) {
    for(int i=0; i<=n; i++) pai[i] = i;
  }
```

```cpp
  int find(int u){ return pai[u] == u ? u : pai[u] = find(pai[
      u]); }

  void join(int u, int v){
    u = find(u), v = find(v);

    if(u == v) return;
    if(sz[v] > sz[u]) swap(u, v);

    pai[v] = u;
    sz[u] += sz[v];
  }
};
```

4.9 Euler Path

```
Euler Path - Algoritmo de Hierholzer para caminho Euleriano

Complexity: O(V + E)

IMPORTANTE! O algoritmo esta 0-indexado

* Informacoes
  addEdge(u, v) -> Adiciona uma aresta de U para V
  EulerPath(n) -> Retorna o Euler Path, ou um vetor vazio se
     impossivel
  vi path -> vertices do Euler Path na ordem
  vi pathId -> id das Arestas do Euler Path na ordem

Euler em Undirected graph:
  - Cada vertice tem um numero par de arestas (circuito); OU
  - Exatamente dois vertices tem um numero impar de arestas
     (caminho);
Euler em Directed graph:
  - Cada vertice tem quantidade de arestas |entrada| ==
     |saida| (circuito); OU
  - Exatamente 1 tem |entrada|+1 == |saida| && exatamente 1
     tem |entrada| == |saida|+1 (caminho);
* Circuito -> U e o primeiro e ultimo
* Caminho -> U e o primeiro e V o ultimo
```

```cpp
#define vi vector<int>

const int MAXN = 1e6 + 5;
const bool BIDIRECIONAL = true;

vector<pii> grafo[MAXN];
vector<bool> used;

void addEdge(int u, int v){
  grafo[u].emplace_back(v, used.size()); if(BIDIRECIONAL && u
      != v)
  grafo[v].emplace_back(u, used.size());
  used.emplace_back(false);
}

pair<vi, vi> EulerPath(int n, int src=0){
  int s=-1, t=-1;
  vector<int> selfLoop(n*BIDIRECIONAL, 0);

  if(BIDIRECIONAL)
  {
    for(int u=0; u<n; u++) for(auto&[v, id] : grafo[u]) if(u==
        v) selfLoop[u]++;
    for(int u=0; u<n; u++)
      if((grafo[u].size() - selfLoop[u])%2)
      {
        if(t != -1) return {vi(), vi()};   // mais que 2 com
            grau impar
        else t = s, s = u;

    if(t == -1 && t != s) return {vi(), vi()}; // so 1 com
        grau impar
    if(s == -1 || t == src) s = src;           // se possivel,
         seta start como src
  }
  else
  {
    vector<int> in(n, 0), out(n, 0);

    for(int u=0; u<n; u++)
      for(auto [v, edg] : grafo[u])
        in[v]++, out[u]++;

    for(int u=0; u<n; u++)
      if(in[u] - out[u] == -1 && s == -1) s = u;  else
      if(in[u] - out[u] ==  1 && t == -1) t = u;  else
      if(in[u] !=out[u]) return {vi(), vi()};

    if(s == -1 && t == -1) s = t = src;          // se possivel
        , seta s como src
    if(s == -1 && t != -1) return {vi(), vi()}; // Existe S
        mas nao T
    if(s != -1 && t == -1) return {vi(), vi()}; // Existe T
        mas nao S
  }

  for(int i=0; grafo[s].empty() && i<n; i++) s =(s+1)%n; //
      evita s ser vertice isolado

  /////// DFS ///////
  vector<int> path, pathId, idx(n, 0);
  stack<pii> st; // {Vertex, EdgeId}
  st.push({s, -1});

  while(!st.empty())
  {
    auto [u, edg] = st.top();
    while(idx[u] < grafo[u].size() && used[grafo[u][idx[u]].
        second]) idx[u]++;

    if(idx[u] < grafo[u].size())
    {
      auto [v, id] = grafo[u][idx[u]];
      used[id] = true;
      st.push({v, id});
      continue;
    }

    path.push_back(u);
    pathId.push_back(edg);
    st.pop();
  }

  pathId.pop_back();
  reverse(begin(path),   end(path));
  reverse(begin(pathId), end(pathId));

  ///  Grafo conexo ? ///
  int edgesTotal = 0;
  for(int u=0; u<n; u++) edgesTotal += grafo[u].size() + (
      BIDIRECIONAL ? selfLoop[u] : 0);
  if(BIDIRECIONAL) edgesTotal /= 2;
  if(pathId.size() != edgesTotal) return {vi(), vi()};

  return {path, pathId};
}
```

4.10 HLD

```
Heavy-Light Decomposition

Complexity: O(LogN * (qry || updt))

Change qry(l, r) and updt(l, r) to call a query and update
     structure of your will

HLD hld(n); //call init
hld.add_edges(u, v); //add all edges
hld.build(); //Build everthing for HLD

tin[u] -> Pos in the structure (Seg, Bit, ...)
nxt[u] -> Head/Endpoint
IMPORTANTE! o grafo deve estar 0-indexado!
```

```cpp
const bool EDGE = false;
struct HLD {
public:
  vector<vector<int>> g; //grafo
  vector<int> sz, parent, tin, nxt;
  HLD(){}
  HLD(int n){ init(n); }
  void init(int n){
    t = 0;
    g.resize(n); tin.resize(n);
    sz.resize(n);nxt.resize(n);
    parent.resize(n);
  }
  void addEdge(int u, int v){
    g[u].emplace_back(v);
    g[v].emplace_back(u);
  }
  void build(int root=0){
    nxt[root]=root;
    dfs(root, root);
    hld(root, root);
  }

  ll query_path(int u, int v){
    if(tin[u] < tin[v]) swap(u, v);
    if(nxt[u] == nxt[v]) return qry(tin[v]+EDGE, tin[u]);
    return qry(tin[nxt[u]], tin[u]) + query_path(parent[nxt[u
        ]], v);
  }

  void update_path(int u, int v, ll x){
    if(tin[u] < tin[v]) swap(u, v);
    if(nxt[u] == nxt[v]) return updt(tin[v]+EDGE, tin[u], x);
    updt(tin[nxt[u]], tin[u], x); update_path(parent[nxt[u]],
        v, x);
  }

private:
  ll qry(int l, int r){ if(EDGE && l>r) return 0;/*NEUTRO*/ }
      //call Seg, BIT, etc
  void updt(int l, int r, ll x){ if(EDGE && l>r) return; }
      //call Seg, BIT, etc

  void dfs(int u, int p){
    sz[u] = 1, parent[u] = p;
    for(auto &v : g[u]) if(v != p) {
      dfs(v, u); sz[u] += sz[v];

      if(sz[v] > sz[g[u][0]] || g[u][0] == p)
```

```cpp
            swap(v, g[u][0]);
        }
    }

    int t=0;
    void hld(int u, int p){
        tin[u] = t++;
        for(auto &v : g[u]) if(v != p)
            nxt[v] = (v == g[u][0] ? nxt[u] : v),
            hld(v, u);
    }

    /// OPTIONAL ///
    int lca(int u, int v){
        while(!inSubtree(nxt[u], v)) u = parent[nxt[u]];
        while(!inSubtree(nxt[v], u)) v = parent[nxt[v]];
        return tin[u] < tin[v] ? u : v;
    }
    bool inSubtree(int u, int v){ return tin[u] <= tin[v] && tin
        [v] < tin[u] + sz[u]; }
    //query/update_subtree[tin[u]+EDGE, tin[u]+sz[u]-1];
};
```

4.11 LCA

```
  LCA - Lowest Common Ancestor - Binary Lifting
Algoritmo para encontrar o menor ancestral comum
entre dois vertices em uma arvore enraizada

IMPORTANTE! O algoritmo esta 0-indexado

Complexity:
buildBL() ->  O(N Log N)
lca()     ->  O(Log N)

* Informacoes
  -> chame dfs(root, root) para calcular o pai e a altura de
     cada vertice
  -> chame buildBL() para criar a matriz do Binary Lifting
  -> chame lca(u, v) para encontrar o menor ancestral comum
  bl[i][u] -> Binary Lifting com o (2^i)-esimo pai de u
  lvl[u]   -> Altura ou level de U na arvore
```

```cpp
const int MAXN = 1e4 + 5;
const int MAXLG = 16;

vector<int> grafo[MAXN];

int bl[MAXLG][MAXN], lvl[MAXN];

void dfs(int u, int p, int l=0){
    lvl[u] = l;
    bl[0][u] = p;

    for(auto v : grafo[u])
        if(v != p)
            dfs(v, u, l+1);
}

void buildBL(int N){
    for(int i=1; i<MAXLG; i++)
        for(int u=0; u<N; u++)
            bl[i][u] = bl[i-1][bl[i-1][u]];
}

int lca(int u, int v){
    if(lvl[u] < lvl[v]) swap(u, v);

    for(int i=MAXLG-1; i>=0; i--)
        if(lvl[u] - (1<<i) >= lvl[v])
            u = bl[i][u];

    if(u == v) return u;

    for(int i=MAXLG-1; i>=0; i--)
        if(bl[i][u] != bl[i][v])
            u = bl[i][u],
            v = bl[i][v];

    return bl[0][u];
}
```

4.12 MinCost MaxFlow

```cpp
struct Aresta {
    int u, v; ll cap, cost;
    Aresta(int u, int v, ll cap, ll cost) : u(u), v(v), cap(cap)
        , cost(cost) {}
};

struct MCMF {
    const ll INF = numeric_limits<ll>::max();
    int n, source, sink;
    vector<vector<int>> adj;
    vector<Aresta> edges;
    vector<ll> dist, pot;
    vector<int> from;

    MCMF(int n, int source, int sink) : n(n), source(source),
        sink(sink) { adj.resize(n); pot.resize(n); }

    void addAresta(int u, int v, ll cap, ll cost){
        adj[u].push_back(edges.size());
        edges.emplace_back(u, v, cap, cost);

        adj[v].push_back(edges.size());
        edges.emplace_back(v, u, 0, -cost);
    }

    queue<int> q;
    vector<bool> vis;
    bool SPFA(){
        dist.assign(n, INF);
        from.assign(n, -1);
        vis.assign(n, false);

        q.push(source);
        dist[source] = 0;

        while(!q.empty()){
            int u = q.front();
            q.pop();

            vis[u] = false;

            for(auto i : adj[u]){
                if(edges[i].cap == 0) continue;
                int v = edges[i].v;
                ll cost = edges[i].cost;

                if(dist[v] > dist[u] + cost + pot[u] - pot[v]){
                    dist[v] = dist[u] + cost + pot[u] - pot[v];
                    from[v] = i;
                    if(!vis[v]) q.push(v), vis[v] = true;
                }
            }
        }

        for(int u=0; u<n; u++)  //fix pot
            if(dist[u] < INF)
                pot[u] += dist[u];

        return dist[sink] < INF;
    }

    pair<ll, ll> augment(){
        ll flow = edges[from[sink]].cap, cost = 0; //fixed flow:
            flow = min(flow, remainder)

        for(int v=sink; v != source; v = edges[from[v]].u)
            flow = min(flow, edges[from[v]].cap),
            cost += edges[from[v]].cost;

        for(int v=sink; v != source; v = edges[from[v]].u)
            edges[from[v]].cap   -= flow,
            edges[from[v]^1].cap += flow;

        return {flow, cost};
    }

    bool inCut(int u){ return dist[u] < INF; }

    pair<ll, ll> maxFlow(){
        ll flow = 0, cost = 0;

        while( SPFA() ){
            auto [f, c] = augment();
            flow += f;
            cost += f*c;
        }
        return {flow, cost};
    }
};
```

4.13 SCC - Kosaraju

```
Kosaraju - Strongly Connected Component
Algoritmo de Kosaraju para encontrar Componentes Fortemente
     Conexas

Complexity: O(V + E)
IMPORTANTE! O algoritmo esta 0-indexado

* Variaveis e explicacoes *
int C    -> C e a quantidade de Componetes Conexas. As
     componetes estao numeradas de 0 a C-1
dag      -> Apos rodar o Kosaraju, o grafo das componentes
     conexas sera criado aqui
comp[u] -> Diz a qual componente conexa U faz parte
order    -> Ordem de saida dos vertices. Necessario para o
     Kosaraju
grafo    -> grafo direcionado
greve -> grafo reverso (que deve ser construido junto ao
     grafo normal) !!!

NOTA: A ordem que o Kosaraju descobre as componentes e uma
     Ordenacao Topologica do SCC
em que o dag[0] nao possui grau de entrada e o dag[C-1] nao
     possui grau de saida
```

```
#define vi vector<int>

const int MAXN = 1e6 + 5;

vi grafo[MAXN];
vi greve[MAXN];
vi dag[MAXN];
vi comp, order;
vector<bool> vis;
int C;

void dfs(int u){
  vis[u] = true;
  for(auto v : grafo[u])
    if(!vis[v])
      dfs(v);
  order.push_back(u);
}

void dfs2(int u){
  comp[u] = C;
  for(auto v : greve[u])
    if(comp[v] == -1)
      dfs2(v);
}

void kosaraju(int n){
  order.clear();
  comp.assign(n, -1);
  vis.assign(n, false);

  for(int v=0; v<n; v++)
    if(!vis[v])
      dfs(v);

  C = 0;
  reverse(begin(order), end(order));

  for(auto v : order)
    if(comp[v] == -1)
      dfs2(v), C++;

  //// Montar DAG  ////
  vector<bool> marc(C, false);

  for(int u=0; u<n; u++){
    for(auto v : grafo[u])
    {
      if(comp[v] == comp[u] || marc[comp[v]]) continue;

      marc[comp[v]] = true;
      dag[comp[u]].emplace_back(comp[v]);
    }

    for(auto v : grafo[u]) marc[comp[v]] = false;
  }
}
```

4.14 Tarjan

```
Tarjan - Pontes e Pontos de Articulacao
Algoritmo para encontrar pontes e pontos de articulacao.

Complexity: O(V + E)
IMPORTANTE! Lembre do memset(pre, -1, sizeof pre);

* Variaveis e explicacoes *
pre[u] = "Altura", ou, x-esimo elemento visitado na DFS.
     Usado para saber a posicao de um vertice na arvore de DFS
low[u] = Low Link de U, ou a menor aresta de retorno (mais
     proxima da raiz) que U alcanca entre seus filhos

chd = Children. Quantidade de componentes filhos de U. Usado
     para saber se a Raiz e Ponto de Articulacao.
any = Marca se alguma aresta de retorno em qualquer dos
     componentes filhos de U nao ultrapassa U. Se isso for
     verdade, U e Ponto de Articulacao.

if(low[v] >  pre[u]) pontes.emplace_back(u, v); -> se a mais
     alta aresta de retorno de V (ou o menor low) estiver
     abaixo de U, entao U-V e ponte
if(low[v] >= pre[u]) any = true;         -> se a mais alta
     aresta de retorno de V (ou o menor low) estiver abaixo de
     U ou igual a U, entao U e Ponto de Articulacao
```

```
const int MAXN = 1e6 + 5;
int pre[MAXN], low[MAXN], clk=0;
vector<int> grafo [MAXN];

vector<pair<int, int>> pontes;
vector<int> cut;

#warning "lembrar do memset(pre, -1, sizeof pre);"
void tarjan(int u, int p = -1){
  pre[u] = low[u] = clk++;

  bool any = false;
  int chd = 0;

  for(auto v : grafo[u]){
    if(v == p) continue;

    if(pre[v] == -1)
    {
      tarjan(v, u);

      low[u] = min(low[v], low[u]);

      if(low[v] >  pre[u]) pontes.emplace_back(u, v);
      if(low[v] >= pre[u]) any = true;

      chd++;
    }
    else
      low[u] = min(low[u], pre[v]);
  }

  if(p == -1 && chd >= 2) cut.push_back(u);
  if(p != -1 && any)      cut.push_back(u);
}
```

5 Strings

5.1 hash

```
const int MAXN = 1e6 + 5;

const ll MOD = 1e9 + 7; //WA? Muda o MOD e a base
const ll base = 153;

ll expb[MAXN];

void precalc(){
  expb[0] = 1;
  for(int i=1; i<MAXN; i++)
    expb[i] = (expb[i-1]*base)%MOD;
}

struct StringHash{
  vector<ll> hsh;

  StringHash(string &s){
    hsh.assign(s.size()+1, 0);
    for(int i=0; i<s.size(); i++)
      hsh[i+1] = (hsh[i] * base % MOD + s[i]) % MOD;
  }

  ll gethash(int l, int r){
    return (MOD + hsh[r+1] - hsh[l]*expb[r-l+1] % MOD ) % MOD;
  }
};
String Hash
precalc()    -> O(N)
StringHash() -> O(|S|)
gethash()    -> O(1)

StringHash hash(s); -> Cria uma struct de StringHash para a
     string s
hash.gethash(l, r); -> Retorna o hash do intervalo L R da
     string (0-Indexado)

IMPORTANTE! Chamar precalc() no inicio do codigo

const ll MOD  = 131'807'699; -> Big Prime Number
const ll base = 127;         -> Random number larger than the
     Alphabet
```

5.2 hash2

```
String Hash - Double Hash
precalc()    -> O(N)
StringHash() -> O(|S|)
gethash()    -> O(1)

StringHash hash(s); -> Cria o Hash da string s
hash.gethash(l, r); -> Hash [L,R] (0-Indexado)
```

```
const int MAXN = 1e6 + 5;

const ll MOD1 = 131'807'699;
const ll MOD2 = 1e9 + 9;
const ll base = 157;

ll expb1[MAXN], expb2[MAXN];

#warning "Call precalc() before use StringHash"
void precalc(){
    expb1[0] = expb2[0] = 1;

  for(int i=1;i<MAXN;i++)
        expb1[i] = expb1[i-1]*base % MOD1,
        expb2[i] = expb2[i-1]*base % MOD2;
}

struct StringHash{
```

```
    vector<pair<ll,ll>> hsh;
    string s; // comment S if you dont need it

    StringHash(string& s) : s(s){
        hsh.assign(s.size()+1, {0,0});

        for (int i=0;i<s.size();i++)
            hsh[i+1].first  = ( hsh[i].first *base % MOD1 + s[
                i] ) % MOD1,
            hsh[i+1].second = ( hsh[i].second*base % MOD2 + s[
                i] ) % MOD2;
    }

    ll gethash(int a,int b){
        ll h1 = (MOD1+ hsh[b+1].first  - hsh[a].first *expb1[b
            -a+1] % MOD1) % MOD1;
        ll h2 = (MOD2+ hsh[b+1].second - hsh[a].second*expb2[b
            -a+1] % MOD2) % MOD2;
        return (h1<<32) | h2;
    }
};

int firstDiff(StringHash& a, int la, int ra, StringHash& b,
     int lb, int rb)
{
  int l=0, r=min(ra-la, rb-lb), diff=r+1;
  while(l <= r){
    int m = (l+r)/2;
    if(a.gethash(la, la+m) == b.gethash(lb, lb+m)) l = m+1;
    else r = m-1, diff = m;
  }
  return diff;
}

int hshComp(StringHash& a, int la, int ra, StringHash& b, int
     lb, int rb){
  int diff = firstDiff(a, la, ra, b, lb, rb);
  if(diff > ra-la && ra-la == rb-lb) return 0; //equal
  if(diff > ra-la || diff  >  rb-lb) return ra-la < rb-lb ? -2
       : +2; //prefix of the other
  return a.s[la+diff] < b.s[lb+diff] ? -1 : +1;
}
```

5.3 KMP

```
vector<int> pi(string &t){
  vector<int> p(t.size(), 0);

  for(int i=1, j=0; i<t.size(); i++)
  {
    while(j > 0  && t[j] != t[i]) j = p[j-1];

    if(t[j] == t[i]) j++;

    p[i] = j;
  }

  return p;
}

vector<int> kmp(string &s, string &t){
  vector<int> p = pi(t), occ;

  for(int i=0, j=0; i<s.size(); i++)
  {
    while( j > 0 && s[i] != t[j]) j = p[j-1];
    if(s[i]==t[j]) j++;

    if(j == t.size()) occ.push_back(i-j+1), j = p[j-1];
  }

  return occ;
}
KMP - Knuth-Morris-Pratt Pattern Searching

Complexity: O(|S|+|T|)

S -> String
T -> Pattern
```

5.4 Manacher

```
vector<int> manacher(string &st){
  string s = "$_";
  for(char c : st){ s += c; s += "_"; }
  s += "#";

  int n = s.size()-2;

  vector<int> p(n+2, 0);
  int l=1, r=1;

  for(int i=1, j; i<=n; i++)
  {
    p[i] = max(0, min(r-i, p[l+r-i]) ); //atualizo o valor
        atual para o valor do palindromo espelho na string ou
        para o total que esta contido

    while( s[i-p[i]] == s[i+p[i]] ) p[i]++;

    if( i+p[i] > r ) l = i-p[i], r = i+p[i];
  }

  for(auto &x : p) x--; //o valor de p[i] e igual ao tamanho
      do palindromo + 1

  return p;
}

Manacher Algorithm
Find every palindrome in string
Complexidade: O(N)
```

5.5 trie

```
Trie - Arvore de Prefixos
insert(P) - O(|P|)
count(P)  - O(|P|)
MAXS  - Soma do tamanho de todas as Strings
sigma - Tamanho do alfabeto
```

```
const int MAXS = 1e5 + 10;
const int sigma = 26;

int trie[MAXS][sigma], terminal[MAXS], z = 1;

void insert(string &p){
  int cur = 0;

  for(int i=0; i<p.size(); i++){
    int id = p[i] - 'a';

    if(trie[cur][id] == -1 ){
      memset(trie[z], -1, sizeof trie[z]);
      trie[cur][id] = z++;
    }

    cur = trie[cur][id];
  }

  terminal[cur]++;
}

int count(string &p){
  int cur = 0;

  for(int i=0; i<p.size(); i++){
    int id = (p[i] - 'a');

    if(trie[cur][id] == -1) return 0;

    cur = trie[cur][id];
  }
  return terminal[cur];
}

void init(){
  memset(trie[0], -1, sizeof trie[0]);
  z = 1;
}
```

5.6 Z-Function

```
vector<int> Zfunction(string &s){ // O(N)
  int n = s.size();
  vector<int> z (n, 0);

  for(int i=1, l=0, r=0; i<n;  i++){
    if(i <= r) z[i] = min(z[i-l], r-i+1);

    while(z[i] + i < n && s[z[i]] == s[i+z[i]]) z[i]++;

    if(r < i+z[i]-1) l = i, r = i+z[i]-1;
  }

  return z;
}
```

6 others

6.1 Hungarian

Hungarian Algorithm - Assignment Problem
Algoritmo para o problema de atribuicao minima.

Complexity: O(N^2 * M)

hungarian(int n, int m); -> Retorna o valor do custo minimo
getAssignment(int m)   -> Retorna a lista de pares
     <linha, Coluna> do Minimum Assignment

n -> Numero de Linhas // m -> Numero de Colunas

```
IMPORTANTE! O algoritmo e 1-indexado
IMPORTANTE! O tipo padrao esta como int, para mudar para
     outro tipo altere | typedef <TIPO> TP; |
Extra: Para o problema da atribuicao maxima, apenas
     multiplique os elementos da matriz por -1
```

```cpp
typedef int TP;

const int MAXN = 1e3 + 5;
const TP INF = 0x3f3f3f3f;

TP matrix[MAXN][MAXN];
TP row[MAXN], col[MAXN];
int match[MAXN], way[MAXN];

TP hungarian(int n, int m){
  memset(row, 0, sizeof row);
  memset(col, 0, sizeof col);
  memset(match, 0, sizeof match);

  for(int i=1; i<=n; i++)
  {
    match[0] = i;
    int j0 = 0, j1, i0;
    TP delta;

    vector<TP> minv (m+1, INF);
    vector<bool> used (m+1, false);

    do {
      used[j0] = true;
      i0 = match[j0];
      j1 = -1;
      delta = INF;

      for(int j=1; j<=m; j++)
        if(!used[j]){
          TP cur = matrix[i0][j] - row[i0] - col[j];

          if( cur < minv[j] ) minv[j] = cur, way[j] = j0;
          if(minv[j] < delta) delta = minv[j], j1 = j;
        }

      for(int j=0; j<=m; j++)
        if(used[j]){
          row[match[j]] += delta,
          col[j] -= delta;
        }else
          minv[j] -= delta;

      j0 = j1;
    } while(match[j0]);


    do {
      j1 = way[j0];
      match[j0] = match[j1];
      j0 = j1;
    } while(j0);
  }

  return -col[0];
}

vector<pair<int, int>> getAssignment(int m){
  vector<pair<int, int>> ans;

  for(int i=1; i<=m; i++)
    ans.push_back(make_pair(match[i], i));

  return ans;
}
```

6.2 MO

```
Algoritmo de MO para query em range

Complexity: O( (N + Q) * SQRT(N) * F ) | F e a complexidade
     do Add e Remove

IMPORTANTE! Queries devem ter seus indices (Idx) 0-indexados!

Modifique as operacoes de Add, Remove e GetAnswer de acordo
     com o problema.
BLOCK_SZ pode ser alterado para aproximadamente SQRT(MAX_N)
```

```cpp
const int BLOCK_SZ = 700;

struct Query{
  int l, r, idx;

  Query(int l, int r, int idx) : l(l), r(r), idx(idx) {}

  bool operator < (Query q) const {
    if(l / BLOCK_SZ != q.l / BLOCK_SZ) return l < q.l;
    return (l / BLOCK_SZ &1) ? ( r < q.r ) : (r > q.r );
  }
};

void add(int idx);
void remove(int idx);
int getAnswer();

vector<int> MO(vector<Query> &queries){
  vector<int> ans(queries.size());

  sort(queries.begin(), queries.end());

  int L = 0, R = 0;
  add(0);

  for(auto [l, r, idx] : queries){
    while(l < L) add(--L);
    while(r > R) add(++R);
    while(l > L) remove(L++);
    while(r < R) remove(R--);

    ans[idx] = getAnswer();
  }

  return ans;
}

/* IF you want to use hilbert curves on MO
vector<ll> h(ans.size());
for (int i = 0; i < ans.size(); i++) h[i] = hilbert(queries[i
     ].l, queries[i].r);
sort(queries.begin(), queries.end(), [&](Query&a, Query&b) {
     return h[a.idx] < h[b.idx]; });   */
inline ll hilbert(int x, int y){
  static int N = 1 << (__builtin_clz(0) - __builtin_clz(MAXN))
     ;
  int rx, ry, s; ll d = 0;
  for(s = N/2; s > 0; s /= 2){
    rx = (x & s) > 0, ry = (y & s) > 0;
    d += s * (ll)(s) * ((3 * rx) ^ ry);
    if(ry == 0) { if(rx == 1) x = N-1 - x, y = N-1 - y; swap(x
        , y); }
  }
  return d;
}
```

6.3 MOTree

```
Algoritmo de MO para query de caminho em arvore
Complexity: O((N + Q) * SQRT(N) * F) | F e a complexidade do
     Add e Remove
IMPORTANTE! 0-indexado!
```

```cpp
const int MAXN = 1e5+5;
const int BLOCK_SZ = 500;
struct Query{int l, r, idx;}; //same of MO. Copy operator <

vector<int> g[MAXN];
int tin[MAXN], tout[MAXN];
int pai[MAXN], order[MAXN];

void remove(int u);
void add(int u);
int getAnswer();

void go_to(int ti, int tp, int otp){
  int u = order[ti], v, to;
  to = tout[u];
  while(!(ti <= tp && tp <= to)){ //subo com U (ti) ate ser
       ancestral de W
    v = pai[u];

    if(ti <= otp && otp <= to) add(v);
    else remove(u);

    u = v;
    ti = tin[u];
    to = tout[u];
  }

  int w = order[tp];
  to = tout[w];
  while(ti < tp){ //subo com W (tp) ate U
    v = pai[w];

    if(tp <= otp && otp <= to) remove(v);
    else add(w);

    w = v;
    tp = tin[w];
    to = tout[w];
  }
}

int TIME = 0;
void dfs(int u, int p){
  pai[u] = p;
  tin[u] = TIME++;
  order[tin[u]] = u;

  for(auto v : g[u])
    if(v != p)
      dfs(v, u);
```

```
  tout[u] = TIME-1;
}

vector<int> MO(vector<Query> &queries){
  vector<int> ans(queries.size());
  dfs(0, 0);

  for(auto &[u, v, i] : queries)
    tie(u, v) = minmax(tin[u], tin[v]);
  sort(queries.begin(), queries.end());

  add(0);
  int Lm = 0, Rm = 0;
  for(auto [l, r, idx] : queries){
    if(l < Lm) go_to(Lm, l, Rm), Lm = l;
    if(r > Rm) go_to(Rm, r, Lm), Rm = r;
    if(l > Lm) go_to(Lm, l, Rm), Lm = l;
    if(r < Rm) go_to(Rm, r, Lm), Rm = r;
    ans[idx] = getAnswer();
  }

  return ans;
}
```

7 Math

7.1 fexp

```
ll MOD = 1e9 + 7;

ll fexp(ll b, ll p){
  ll ans = 1;

  while(p){
    if(p&1) ans = (ans*b) % MOD;
    b = b * b % MOD;
    p >>= 1;
  }

  return ans % MOD;
}
// O(Log P) // b - Base // p - Potencia
```

7.2 CRT

```
#define ld long double

ll modinverse(ll a, ll b, ll s0 = 1, ll s1 = 0) { return b ==
     0 ? s0 : modinverse(b, a % b, s1, s0 - s1 * (a / b)); }
ll mul(ll a, ll b, ll m) {
    ll q = (long double) a * (long double) b / (long double) m
        ;
    ll r = a * b - q * m;
    return (r + m) % m;
}

struct Equation {
    ll mod, ans;
    bool valid;
    Equation() { valid = false; }
    Equation(ll a, ll m) { mod = m, ans = (a % m + m) % m,
         valid = true; }
    Equation(Equation a, Equation b){
        if(!a.valid || !b.valid){ valid = false; return; }
        ll g = gcd(a.mod, b.mod);
        if((a.ans - b.ans) % g != 0){ valid = false; return; }

        valid = true;
        mod = a.mod * (b.mod / g);
        ans = a.ans;
        ans += mul( mul(a.mod,  modinverse(a.mod, b.mod), mod)
             ,   (b.ans - a.ans) / g, mod);

        ans = (ans % mod + mod) % mod;
    }
    Equation operator+(const Equation& b) const { return
         Equation(*this, b); }
};
// Equation eq1(2, 3); // x = 2 mod 3
// Equation eq2(3, 5); // x = 3 mod 5
// Equation ans = eq1 + eq2;
```

8 Theorems

8.1 Propriedades Matemáticas

- **Conjectura de Goldbach:** Todo número par $n > 2$ pode ser representado como $n = a + b$, onde a e b são primos.
- **Primos Gêmeos:** Existem infinitos pares de primos p, $p + 2$.
- **Conjectura de Legendre:** Sempre existe um primo entre n^2 e $(n + 1)^2$.
- **Lagrange:** Todo número inteiro pode ser representado como soma de 4 quadrados.
- **Zeckendorf:** Todo número pode ser representado como soma de números de Fibonacci diferentes e não consecutivos.
- **Tripla de Pitágoras (Euclides):** Toda tripla pitagórica primitiva pode ser gerada por $(n^2 - m^2, 2nm, n^2 + m^2)$ onde n e m são coprimos e um deles é par.
- **Wilson:** n é primo se e somente se $(n-1)! \mod n = n-1$.
- **Problema do McNugget:** Para dois coprimos x e y, o número de inteiros que não podem ser expressos como $ax + by$ é $(x - 1)(y - 1)/2$. O maior inteiro não representável é $xy - x - y$.
- **Fermat:** Se p é primo, então $a^{p-1} \equiv 1 \mod p$. Se x e m são coprimos e m primo, então $x^k \equiv x^{k \bmod (m-1)} \mod m$. *Euler:* $x^{\varphi(m)} \equiv 1 \mod m$. $\varphi(m)$ é o totiente de Euler.
- **Teorema Chinês do Resto:** Dado um sistema de congruências:

$$x \equiv a_1 \mod m_1, \quad \ldots, \quad x \equiv a_n \mod m_n$$

com m_i coprimos dois a dois. E seja $M_i = \frac{m_1 m_2 \cdots m_n}{m_i}$ e $N_i = M_i^{-1} \mod m_i$. Então a solução é dada por:

$$x = \sum_{i=1}^{n} a_i M_i N_i$$

Outras soluções são obtidas somando $m_1 m_2 \cdots m_n$.

- **Números de Catalan:** Exemplo: expressões de parênteses bem formadas. $C_0 = 1$, e:

$$C_n = \sum_{i=0}^{n-1} C_i C_{n-1-i} = \frac{1}{n+1}\binom{2n}{n}$$

- **Bertrand (Ballot):** Com $p > q$ votos, a probabilidade de sempre haver mais votos do tipo A do que B até o fim é: $\frac{p-q}{p+q}$ Permitindo empates: $\frac{p+1-q}{p+1}$. Multiplicando pela combinação total $\binom{p+q}{q}$, obtém-se o número de possibilidades.
- **Linearidade da Esperança:** $E[aX+bY] = aE[X]+bE[Y]$
- **Variância:** $\text{Var}(X) = E[(X - \mu)^2] = E[X^2] - E[X]^2$
- **Progressão Geométrica:** $S_n = a_1 \cdot \frac{q^n - 1}{q - 1}$
- **Soma dos Cubos:** $\sum_{k=1}^{n} k^3 = \left(\sum_{k=1}^{n} k\right)^2$
- **Lindström-Gessel-Viennot:** A quantidade de caminhos disjuntos em um grid pode ser computada como o determinante da matriz do número de caminhos.
- **Lema de Burnside:** Número de colares diferentes (sem contar rotações), com m cores e comprimento n:

$$\frac{1}{n}\left(m^n + \sum_{i=1}^{n-1} m^{\gcd(i,n)}\right)$$

- **Inversão de Möbius:**

$$\sum_{d|n} \mu(d) = \begin{cases} 1, & n = 1 \\ 0, & \text{caso contrário} \end{cases}$$

- **Propriedades de Coeficientes Binomiais:**

$$\binom{N}{K} = \binom{N}{N-K} = \frac{N}{K}\binom{N-1}{K-1}$$
$$\sum_{k=0}^{m}(-1)^k\binom{n}{k} = (-1)^m\binom{n-1}{m}$$
$$\sum_{m=0}^{n}\binom{m}{k} = \binom{n+1}{k+1}$$
$$\sum_{k=0}^{m}\binom{n+k}{k} = \binom{n+m+1}{m}$$
$$\sum_{k=0}^{n}\binom{n}{k}^2 = \binom{2n}{n}$$
$$\sum_{k=0}^{n}\binom{n}{k} = 2^n$$
$$\sum_{k=0}^{n}k\binom{n}{k} = n \cdot 2^{n-1}$$
$$\sum_{k=0}^{n}\binom{n-k}{k} = F_{n+1}$$

- **Identidades Clássicas:**
  - **Hockey-stick:** $\sum_{i=r}^{n}\binom{i}{r} = \binom{n+1}{r+1}$
  - **Vandermonde:** $\binom{m+n}{r} = \sum_{k=0}^{r}\binom{m}{k}\binom{n}{r-k}$
- **Distribuições de Probabilidade:**
  - **Uniforme:** $X \in \{a, a+1, \ldots, b\}$, $E[X] = \frac{a+b}{2}$
  - **Binomial:** n tentativas com probabilidade p de sucesso:

$$P(X = x) = \binom{n}{x}p^x(1-p)^{n-x}, \quad E[X] = np$$

  - **Geométrica:** Número de tentativas até o primeiro sucesso:

$$P(X = x) = (1-p)^{x-1}p, \quad E[X] = \frac{1}{p}$$

8.2 Geometria

- **Fórmula de Euler:** Em um grafo planar ou poliedro convexo, temos: $V - E + F = 2$ onde V é o número de vértices, E o número de arestas e F o número de faces.
- **Teorema de Pick:** Para polígonos com vértices em coordenadas inteiras:

$$\text{Área} = i + \frac{b}{2} - 1$$

onde i é o número de pontos interiores e b o número de pontos sobre o perímetro.

- **Teorema das Duas Orelhas (Two Ears Theorem):** Todo polígono simples com mais de três vértices possui pelo menos duas "orelhas"— vértices que podem ser removidos sem gerar interseções. A remoção repetida das orelhas resulta em uma triangulação do polígono.
- **Incentro de um Triângulo:** É o ponto de interseção das bissetrizes internas e centro da circunferência inscrita. Se a, b e c são os comprimentos dos lados opostos aos vértices $A(X_a, Y_a)$, $B(X_b, Y_b)$ e $C(X_c, Y_c)$, então o incentro (X, Y) é dado por:

$$X = \frac{aX_a + bX_b + cX_c}{a+b+c}, \quad Y = \frac{aY_a + bY_b + cY_c}{a+b+c}$$

- **Triangulação de Delaunay:** Uma triangulação de um conjunto de pontos no plano tal que nenhum ponto está dentro do círculo circunscrito de qualquer triângulo. Essa triangulação:
  - Maximiza o menor ângulo entre todos os triângulos.
  - Contém a árvore geradora mínima (MST) euclidiana como subconjunto.
- **Fórmula de Brahmagupta:** Para calcular a área de um quadrilátero cíclico (todos os vértices sobre uma circunferência), com lados a, b, c e d:

$$s = \frac{a+b+c+d}{2}, \quad \text{Área} = \sqrt{(s-a)(s-b)(s-c)(s-d)}$$

Se $d = 0$ (ou seja, um triângulo), ela se reduz à fórmula de Heron:
$$\text{Área} = \sqrt{(s-a)(s-b)(s-c)s}$$

8.3 Grafos

- **Fórmula de Euler (para grafos planares):**

$$V - E + F = 2$$

onde V é o número de vértices, E o número de arestas e F o número de faces.

- **Handshaking Lemma:** O número de vértices com grau ímpar em um grafo é par.
- **Teorema de Kirchhoff (contagem de árvores geradoras):** Monte a matriz M tal que:

$$M_{i,i} = \deg(i), \quad M_{i,j} = \begin{cases} -1 & \text{se existe aresta } i - j \\ 0 & \text{caso contrário} \end{cases}$$

O número de árvores geradoras (spanning trees) é o determinante de qualquer co-fator de M (remova uma linha e uma coluna).

- **Condições para Caminho Hamiltoniano:**
  - **Teorema de Dirac:** Se todos os vértices têm grau $\geq n/2$, o grafo contém um caminho Hamiltoniano.
  - **Teorema de Ore:** Se para todo par de vértices não adjacentes u e v, temos $\deg(u) + \deg(v) \geq n$, então o grafo possui caminho Hamiltoniano.
- **Algoritmo de Borůvka:** Enquanto o grafo não estiver conexo, para cada componente conexa escolha a aresta de menor custo que sai dela. Essa técnica constrói a árvore geradora mínima (MST).
- **Árvores:**
  - Existem C_n árvores binárias com n vértices (C_n é o n-ésimo número de Catalan).
  - Existem C_{n-1} árvores enraizadas com n vértices.
  - **Fórmula de Cayley:** Existem n^{n-2} árvores com vértices rotulados de 1 a n.
  - **Código de Prüfer:** Remova iterativamente a folha com menor rótulo e adicione o rótulo do vizinho ao código até restarem dois vértices.
- **Fluxo em Redes:**
  - **Corte Mínimo:** Após execução do algoritmo de fluxo máximo, um vértice u está do lado da fonte se $\text{level}[u] \neq -1$.
  - **Máximo de Caminhos Disjuntos:**
    * **Arestas disjuntas:** Use fluxo máximo com capacidades iguais a 1 em todas as arestas.
    * **Vértices disjuntos:** Divida cada vértice v em v_{in} e v_{out}, conectados por aresta de capacidade 1. As arestas que entram vão para v_{in} e as que saem saem de v_{out}.
  - **Teorema de König:** Em um grafo bipartido:

$$\text{Cobertura mínima de vértices} = \text{Matching máximo}$$

  O complemento da cobertura mínima de vértices é o conjunto independente máximo.

- **Coberturas:**
  - **Vertex Cover mínimo:** Os vértices da partição X que **não** estão do lado da fonte no corte mínimo, e os vértices da partição Y que **estão** do lado da fonte.
  - **Independent Set máximo:** Complementar da cobertura mínima de vértices.
  - **Edge Cover mínimo:** É $N-$matching, pegando as arestas do matching e mais quaisquer arestas restantes para cobrir os vértices descobertos.
- **Path Cover:**
  - **Node-disjoint path cover mínimo:** Duplicar vértices em tipo A e tipo B e criar grafo bipartido com arestas de $A \to B$. O path cover é $N -$ matching.
  - **General path cover mínimo:** Criar arestas de $A \to B$ sempre que houver caminho de A para B no grafo. O resultado também é $N -$ matching.
- **Teorema de Dilworth:** O path cover mínimo em um grafo dirigido acíclico é igual à **antichain máxima** (conjunto de vértices sem caminhos entre eles).
- **Teorema do Casamento de Hall:** Um grafo bipartido possui um matching completo do lado X se:

$$\forall W \subseteq X, \quad |W| \leq |\text{vizinhos}(W)|$$

- **Fluxo Viável com Capacidades Inferiores e Superiores:** Para rede sem fonte e sumidouro:
  - Substituir a capacidade de cada aresta por $c_{\text{upper}} - c_{\text{lower}}$
  - Criar nova fonte S e sumidouro T
  - Para cada vértice v, compute:

$$M[v] = \sum_{\texttt{arestas entrando}} c_{\text{lower}} - \sum_{\texttt{arestas saindo}} c_{\text{lower}}$$

  - Se $M[v] > 0$, adicione aresta (S, v) com capacidade $M[v]$; se $M[v] < 0$, adicione (v, T) com capacidade $-M[v]$.
  - Se todas as arestas de S estão saturadas no fluxo máximo, então um fluxo viável existe. O fluxo viável final é o fluxo computado mais os valores de c_{lower}.

8.4 DP

- **Divide and Conquer Optimization:** Utilizada em problemas do tipo:

$$dp[i][j] = \min_{k<j}\{dp[i-1][k] + C[k][j]\}$$

  onde o objetivo é dividir o subsegmento até j em i segmentos com algum custo. A otimização é válida se:

$$A[i][j] \leq A[i][j+1]$$

  onde $A[i][j]$ é o valor de k que minimiza a transição.
- **Knuth Optimization:** Aplicável quando:

$$dp[i][j] = \min_{i<k<j}\{dp[i][k] + dp[k][j]\} + C[i][j]$$

  e a condição de monotonicidade é satisfeita:

$$A[i][j-1] \leq A[i][j] \leq A[i+1][j]$$

  com $A[i][j]$ sendo o índice k que minimiza a transição.
- **Slope Trick:** Técnica usada para lidar com funções lineares por partes e convexas. A função é representada por pontos onde a derivada muda, que podem ser manipulados com `multiset` ou `heap`. Útil para manter o mínimo de funções acumuladas em forma de envelopes convexos.
- **Outras Técnicas e Truques Importantes:**
  - **FFT (Fast Fourier Transform):** Convolução eficiente de vetores.
  - **CHT (Convex Hull Trick):** Otimização para DP com funções lineares e monotonicidade.
  - **Aliens Trick:** Técnica para binarizar o custo em problemas de otimização paramétrica (geralmente em problemas com limite no número de grupos/segmentos).
  - **Bitset:** Utilizado para otimizações de espaço e tempo em DP de subconjuntos ou somas parciais, especialmente em problemas de mochila.