Indian Institute of Technology Kharagpur

# AlooParatha

Aashirwaad Mishra, Sayandeep Bhowmick, Sujan Jain

2024-12-19

# Contents



**Contest (1)**

template.cpp

9 lines

```
// #pragma GCC optimize("O3,unroll-
    loops")
int32_t main(){
    ios_base::sync_with_stdio(0);
    cin.tie(0);
    cout.tie(NULL);
    #ifndef ONLINE_JUDGE
        freopen("input.txt","r",stdin
            );
        freopen("output.txt","w",
            stdout);
    #endif }
```

**Mathematics (2)**

## 2.1 Geometry

### 2.1.1 Triangles

Side lengths: $a,b,c$

Semiperimeter: $p = \frac{a+b+c}{2}$

Area: $A = \sqrt{p(p-a)(p-b)(p-c)}$

Circumradius: $R = \frac{abc}{4A}$

Inradius: $r = \frac{A}{p}$

Length of median (divides triangle into two equal-area triangles):
$m_a = \frac{1}{2}\sqrt{2b^2+2c^2-a^2}$

Length of bisector (divides angles in two):

$$s_a = \sqrt{bc\left[1-\left(\frac{a}{b+c}\right)^2\right]}$$

Law of sines:
$\frac{\sin\alpha}{a} = \frac{\sin\beta}{b} = \frac{\sin\gamma}{c} = \frac{1}{2R}$

Law of cosines: $a^2 = b^2+c^2-2bc\cos\alpha$

Law of tangents: $\frac{a+b}{a-b} = \frac{\tan\frac{\alpha+\beta}{2}}{\tan\frac{\alpha-\beta}{2}}$

### 2.1.2 Discrete distributions

**Binomial distribution**

The number of successes in $n$ independent yes/no experiments, each which yields success with probability $p$ is
$\mathrm{Bin}(n,p)$, $n = 1,2,\ldots$, $0 \le p \le 1$.

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

$$\mu = np, \sigma^2 = np(1-p)$$

$\mathrm{Bin}(n,p)$ is approximately $\mathrm{Po}(np)$ for small $p$.

**First success distribution**

The number of trials needed to get the first success in independent yes/no experiments, each which yields success with probability $p$ is $\mathrm{Fs}(p)$, $0 \le p \le 1$.

$$p(k) = p(1-p)^{k-1}, k = 1,2,\ldots$$

$$\mu = \frac{1}{p}, \sigma^2 = \frac{1-p}{p^2}$$

**Poisson distribution**

The number of events occurring in a fixed period of time $t$ if these events occur with a known average rate $\kappa$ and independently of the time since the last event is
$\mathrm{Po}(\lambda)$, $\lambda = t\kappa$.

$$p(k) = e^{-\lambda}\frac{\lambda^k}{k!}, k = 0,1,2,\ldots$$

$$\mu = \lambda, \sigma^2 = \lambda$$

### 2.1.3 Continuous distributions

**Exponential distribution**

The time between events in a Poisson process is $\mathrm{Exp}(\lambda)$, $\lambda > 0$.

$$f(x) = \begin{cases} \lambda e^{-\lambda x} & x \geq 0 \\ 0 & x < 0 \end{cases}$$

$$\mu = \frac{1}{\lambda}, \sigma^2 = \frac{1}{\lambda^2}$$

**Normal distribution**

Most real random values with mean $\mu$ and variance $\sigma^2$ are well described by $\mathcal{N}(\mu, \sigma^2)$, $\sigma > 0$.

$$f(x) = \frac{1}{\sqrt{2\pi\sigma^2}} e^{-\frac{(x-\mu)^2}{2\sigma^2}}$$

If $X_1 \sim \mathcal{N}(\mu_1, \sigma_1^2)$ and $X_2 \sim \mathcal{N}(\mu_2, \sigma_2^2)$ then

$$aX_1 + bX_2 + c \sim \mathcal{N}(\mu_1 + \mu_2 + c, a^2\sigma_1^2 + b^2\sigma_2^2)$$

### 2.2 General Random Walk

Let $a > 0$ and $b > 0$ be integers, and let $R_n$ denote a simple random walk with $R_0 = 0$. Let:

$$p(a) = P(R_n \text{ hits level } a \text{ before hitting level } -b).$$

By letting $a = N - i$ and $b = i$ (so that $N = a + b$), we can imagine a gambler who starts with $i = b$ and wishes to reach $N = a + b$ before going broke. So we can compute $p(a)$ by casting the problem into the framework of the gambler's ruin problem:

$$p(a) = P_i \quad \text{where } N = a + b, i = b.$$

The following equation holds:

$$p(a) = \begin{cases} \frac{1-\left(\frac{q}{p}\right)^b}{1-\left(\frac{q}{p}\right)^{a+b}} & \text{if } p \neq q, \\ \frac{b}{a+b} & \text{if } p = q = 0.5. \end{cases}$$

## Data structures (3)

### OrderStatisticTree.h

**Description:** A set (not multiset!) with support for finding the n'th element, and finding the index of an element. To get a map, change `null_type`.

**Time:** $\mathcal{O}(\log N)$

<ext/pb_ds/assoc_container.hpp>, <ext/pb_ds/tree_policy.hpp>

819d08, 13 lines

```
using namespace __gnu_pbds;
template<class T>
using Tree = tree<T, null_type, less<
    T>, rb_tree_tag,
 tree_order_statistics_node_update>;
void example() {
 Tree<int> t, t2; t.insert(8);
 auto it = t.insert(10).first;
 assert(it == t.lower_bound(9));
 assert(t.order_of_key(10) == 1);
 assert(t.order_of_key(11) == 2);
 assert(*t.find_by_order(0) == 8);
 t.join(t2); // assuming T < T2 or T
    > T2, merge t2 into t
}
```

### SegmentTree.h

**Description:** Zero-indexed max-tree. Bounds are inclusive to the left and exclusive to the right. Can be changed by modifying T, f and unit.

**Time:** $\mathcal{O}(\log N)$

0f4bdb, 19 lines

```
struct Tree {
 typedef int T;
 static constexpr T unit = INT_MIN;
 T f(T a, T b) { return max(a, b); }
    // (any associative fn)
 vector<T> s; int n;
 Tree(int n = 0, T def = unit) : s(2*
    n, def), n(n) {}
 void update(int pos, T val) {
 for (s[pos += n] = val; pos /= 2;)
  s[pos] = f(s[pos * 2], s[pos * 2 +
     1]);
 }
 T query(int b, int e) { // query [b,
     e)
 T ra = unit, rb = unit;
 for (b += n, e += n; b < e; b /= 2,
    e /= 2) {
  if (b % 2) ra = f(ra, s[b++]);
  if (e % 2) rb = f(s[--e], rb);
 }
 return f(ra, rb);
 }
};
```

# LazySegmentTree.h

**Description:** LazySegmentTree.h

0666ae, 62 lines

```
class LazySegmentTree {
private:
 vector<int> t, lazy;
 int n;
 void build(vector<int>& a, int v,
    int tl, int tr) {
  if (tl == tr) {
   t[v] = a[tl];
  } else {
   int tm = (tl + tr) / 2;
   build(a, v*2, tl, tm);
   build(a, v*2+1, tm+1, tr);
   t[v] = combine(t[v*2], t[v*2 + 1])
      ;
  }
 }
 void push(int v) {
  t[v*2] += lazy[v];
  lazy[v*2] += lazy[v];
  t[v*2+1] += lazy[v];
  lazy[v*2+1] += lazy[v];
  lazy[v] = 0;
 }
 void update(int v, int tl, int tr,
    int l, int r, int addend) {
  if (l > r)
   return;
  if (l == tl && tr == r) {
   t[v] += addend;
   lazy[v] += addend;
  } else {
   push(v);
   int tm = (tl + tr) / 2;
   update(v*2, tl, tm, l, min(r, tm),
       addend);
   update(v*2+1, tm+1, tr, max(l, tm
      +1), r, addend);
   t[v] = combine(t[v*2], t[v*2+1]);
  }
 }
 int query(int v, int tl, int tr, int
     l, int r) {
  if (l > r)
   return -INF;
  if (l == tl && tr == r)
   return t[v];
  push(v);
  int tm = (tl + tr) / 2;
  return combine(query(v*2, tl, tm, l
     , min(r, tm)),
       query(v*2+1, tm+1, tr, max(l,
          tm+1), r));
 }
 int combine(int a, int b) {
  return max(a, b); // Change this
      according to your requirement
 }
public:
 LazySegmentTree(vector<int>& a) {
  n = a.size();
  t.assign(4*n, 0);
  lazy.assign(4*n, 0);
  build(a, 1, 0, n-1);
 }
 void update(int l, int r, int addend
    ) {
  update(1, 0, n-1, l, r, addend);
 }
 int query(int l, int r) {
  return query(1, 0, n-1, l, r);
 }
};
```

# UnionFind.h

**Description:** UnionFind.h

3624b6, 17 lines

```
struct DSU
{
 vi par,size;
 DSU(int n) : par(n), size(n, 1) {
    iota(par.begin(), par.end(), 0);
    }
 int find(int x){return x == par[x] ?
     x : par[x] = find(par[x]);}
 void merge(int x, int y)
 {
  int nx = find(x);
  int ny = find(y);
  if(nx!=ny)
  {
   if(size[nx]<size[ny]) swap(nx,ny);
   par[ny] = nx;
   size[nx]+=size[ny];
  }
 }
};
```

# SubMatrix.h

**Description:** Calculate submatrix sums quickly, given upper-left and lower-right corners (half-open).

**Usage:** `SubMatrix<int> m(matrix);`
`m.sum(0, 0, 2, 2); // top left 4 elements`

**Time:** $\mathcal{O}\left(N^2+Q\right)$

c59ada, 13 lines

```
template<class T>
struct SubMatrix {
 vector<vector<T>> p;
 SubMatrix(vector<vector<T>>& v) {
 int R = sz(v), C = sz(v[0]);
 p.assign(R+1, vector<T>(C+1));
```

```
 rep(r,0,R) rep(c,0,C)
  p[r+1][c+1] = v[r][c] + p[r][c+1] +
      p[r+1][c] - p[r][c];
 }
 T sum(int u, int l, int d, int r) {
 return p[d][r] - p[d][l] - p[u][r] +
     p[u][l];
 }
};
```

# Matrix.h

**Description:** Matrix.h

5742e0, 32 lines

```
template<class T> struct Matrix {
 typedef Matrix M;
 vector<vector<T>> d;
 Matrix(int n){
  d.resize(n,vector<T>(n,0));
 };
 M operator*(const M& m) const {
 M a(m.d.size());
  int N = m.d.size();
 rep(i,0,N) rep(j,0,N)
  rep(k,0,N) {a.d[i][j] += (d[i][k]*m
     .d[k][j])%mod1;a.d[i][j]%=mod1;}
 return a;
 }
 vector<T> operator*(const vector<T>&
     vec) const {
  int N = this->d.size();
 vector<T> ret(N);
 rep(i,0,N) rep(j,0,N) {ret[i] += (d[
    i][j] * vec[j])%mod1;ret[i]%=mod1
    ;}
 return ret;
 }
 M operator^(ll p) const {
 assert(p >= 0);
 M a(this->d.size()), b(*this);
  int N = this->d.size();
 rep(i,0,N) a.d[i][i] = 1;
 while (p) {
  if (p&1) a = a*b;
  b = b*b;
  p >>= 1;
 }
 return a;
 }
};
```

# LineContainer.h

**Description:** Container where you can add lines of the form kx+m, and query maximum values at points x. Useful for dynamic programming ("convex hull trick").
**Time:** $\mathcal{O}(\log N)$

8ec1c7, 29 lines

```
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<>> {
 // (for doubles, use inf = 1/.0, div
    (a,b) = a/b)
 static const ll inf = LLONG_MAX;
 ll div(ll a, ll b) { // floored
    division
 return a / b - ((a ^ b) < 0 && a % b
    ); }
 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(ll k, ll m) {
 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;
 }
};
```

# Treap.h

**Description:** A short self-balancing tree. It acts as a sequential container with log-time splits/joins, and is easy to augment with additional data.
**Time:** $\mathcal{O}(\log N)$

9556fc, 49 lines

```
struct Node {
 Node *l = 0, *r = 0;
 int val, y, c = 1;
 Node(int val) : val(val), y(rand())
    {}
 void recalc();
};
int cnt(Node* n) { return n ? n->c :
   0; }
```

```
void Node::recalc() { c = cnt(l) + cnt(r) + 1; }
template<class F> void each(Node* n, F f) {
 if (n) { each(n->l, f); f(n->val); each(n->r, f); }
}
pair<Node*, Node*> split(Node* n, int k) {
 if (!n) return {};
 if (cnt(n->l) >= k) { // "n->val >= k" for lower_bound(k)
 auto pa = split(n->l, k);
 n->l = pa.second;
 n->recalc();
 return {pa.first, n};
 } else {
 auto pa = split(n->r, k - cnt(n->l) - 1); // and just "k"
 n->r = pa.first;
 n->recalc();
 return {n, pa.second};
 }
}
Node* merge(Node* l, Node* r) {
 if (!l) return r;
 if (!r) return l;
 if (l->y > r->y) {
 l->r = merge(l->r, r);
 l->recalc();
 return l;
 } else {
 r->l = merge(l, r->l);
 r->recalc();
 return r;
 }
}
Node* ins(Node* t, Node* n, int pos) {
 auto pa = split(t, pos);
 return merge(merge(pa.first, n), pa.second);
}
// Example application: move the range [l, r) to index k
void move(Node*& t, int l, int r, int k) {
 Node *a, *b, *c;
 tie(a,b) = split(t, l); tie(b,c) = split(b, r - l);
 if (k <= l) t = merge(ins(a, b, k), c);
 else t = merge(a, ins(c, b, k - r));
}
```

# RMQ.h

**Description:** Range Minimum Queries on an array. Returns min(V[a], V[a + 1], ... V[b - 1]) in constant time.
**Usage:** `RMQ rmq(values);`
`rmq.query(inclusive, exclusive);`
**Time:** $\mathcal{O}(|V|\log|V|+Q)$

510c32, 16 lines

```
template<class T>
struct RMQ {
 vector<vector<T>> jmp;
 RMQ(const vector<T>& V) : jmp(1, V) {
 for (int pw = 1, k = 1; pw * 2 <= sz(V); pw *= 2, ++k) {
  jmp.emplace_back(sz(V) - pw * 2 + 1);
  rep(j,0,sz(jmp[k]))
  jmp[k][j] = min(jmp[k - 1][j], jmp[k - 1][j + pw]);
 }
 }
 T query(int a, int b) {
 assert(a < b); // or return inf if a == b
 int dep = 31 - __builtin_clz(b - a);
 return min(jmp[dep][a], jmp[dep][b - (1 << dep)]);
 }
};
```

# MoQueries.h

**Description:** Answer interval or tree path queries by finding an approximate TSP through the queries, and moving from one query to the next by adding/removing points at the ends. If values are on tree edges, change `step` to add/remove the edge $(a,c)$ and remove the initial `add` call (but keep `in`).
**Time:** $\mathcal{O}\left(N\sqrt{Q}\right)$

436b77, 46 lines

```
class mo_algorithm
{
public:
 int n, q, block_size;
 vector<int> a;
 vector<pair<int, pii>> queries;
 vector<int> answers;
 int answer, val;
 mo_algorithm(int n, int q, vector<int> a, vector<pair<int, int>> queries)
 {
  this->n = n;
  this->q = q;
  this->a = a;
  for (int i = 0; i < q; i++)
   this->queries.push_back({queries[i].first, {queries[i].second, i}});
```

```
 block_size = sqrt(n);
 answers.resize(q);
 val = 0;
}
inline void add(int x) {val--;} // Try your best to keep this O(1) since n*root(n)*log(n) is too slow
inline void remove(int x) {val--;}
void process()
{
 sort(queries.begin(), queries.end(), [this](pair<int, pii> x, pair<int, pii> y) {
  int block_x = x.first / block_size;
  int block_y = y.first / block_size;
  if (block_x != block_y)
   return block_x < block_y;
  return x.second.first < y.second.first;
 });
 int l = 0, r = -1;
 for (auto z : queries)
 {
  int x = z.first, y = z.second.first;
  while (r < y)
   add(a[++r]);
  while (r > y)
   remove(a[r--]);
  while (l < x)
   remove(a[l++]);
  while (l > x)
   add(a[--l]);
   answers[z.second.second] = (val == 0);
  }
 }
};
```

## SegTree.h

**Description:** Segment tree implementation for range minimum query with count

1e12fc, 56 lines

```
struct node {
 int mini;
 int ct;
 node(int m=1e9, int c=0) {
  mini = m;
  ct = c;
 }
};
const int range = 1e5;
int arr[range];
node segment[4*range];
node merge(node& a,node& b)
{
 if(a.mini==b.mini)
 {
  node c(a.mini,a.ct+b.ct);
  return c;
 }
 else if(a.mini<b.mini) return a;
 else return b;
}
void build(int idx,int low,int high)
{
 if(low==high)
 {
  segment[idx] = node(arr[low],1);
  return;
 }
 int mid = low + (high - low)/2;
 build(2*idx,low,mid);
 build(2*idx+1,mid+1,high);
 segment[idx] = merge(segment[2*idx], segment[2*idx+1]);
}
node query(int idx,int low,int high, int l,int r)
{
 if(l<=low&&high<=r) return segment[idx];
 if (high<l||low>r) return node();
 int mid = low + (high-low)/2;
 node left = query(2*idx,low,mid,l,r);
 node right = query(2*idx+1,mid+1,high,l,r);
 return merge(left,right);
}
void pointUpdate(int idx,int low,int high,int pos_in_arr,int val)
{
 if(pos_in_arr<low||pos_in_arr>high) return;
 if(low==high)
 {
  segment[idx]=node(val,1);
  arr[low] = val;
  return;
 }
 int mid = low + (high - low)/2;
 pointUpdate(2*idx,low,mid,pos_in_arr,val);
 pointUpdate(2*idx+1,mid+1,high,pos_in_arr,val);
 segment[idx] = merge(segment[2*idx], segment[2*idx+1]);
```

```
}
```

## Numerical (4)

### 4.1 Matrices

#### Determinant.h

**Description:** Calculates determinant of a matrix. Destroys the matrix.
**Time:** $\mathcal{O}(N^3)$

bd5cec, 15 lines

```
double det(vector<vector<double>>& a)
    {
 int n = sz(a); double res = 1;
 rep(i,0,n) {
 int b = i;
 rep(j,i+1,n) if (fabs(a[j][i]) >
    fabs(a[b][i])) b = j;
 if (i != b) swap(a[i], a[b]), res *=
     -1;
 res *= a[i][i];
 if (res == 0) return 0;
 rep(j,i+1,n) {
  double v = a[j][i] / a[i][i];
  if (v != 0) rep(k,i+1,n) a[j][k] -=
      v * a[i][k];
 }
 }
 return res;
}
```

### 4.2 Fourier transforms

#### FastFourierTransform.h

**Description:** fft(a)
**Time:** $\mathcal{O}(N\log N)$ with $N = |A| + |B|$ ($\sim 1s$ for $N = 2^{22}$)

ccab8f, 102 lines

```
typedef long double ld;
#define mp make_pair
#define eprintf(...) fprintf(stderr,
   __VA_ARGS__)
#define sz(x) ((int)(x).size())
#define TASKNAME "text"
const ld pi = acos((ld)-1);
namespace FFT {
 struct com {
  ld x, y;
  com(ld _x = 0, ld _y = 0) : x(_x),
     y(_y) {}
  inline com operator+(const com &c)
     const {
   return com(x + c.x, y + c.y);
  }
  inline com operator-(const com &c)
     const {
   return com(x - c.x, y - c.y);
  }
  inline com operator*(const com &c)
     const {
   return com(x * c.x - y * c.y, x *
      c.y + y * c.x);
  }
  inline com conj() const {
   return com(x, -y);
  }
 };
 const static int maxk = 21, maxn =
    (1 << maxk) + 1;
 com ws[maxn];
 int dp[maxn];
 com rs[maxn];
 int n, k;
 int lastk = -1;
 void fft(com *a, bool torev = 0) {
  if (lastk != k) {
   lastk = k;
   dp[0] = 0;
   for (int i = 1, g = -1; i < n; ++i
      ) {
    if (!(i & (i - 1))) {
     ++g;
    }
    dp[i] = dp[i ^ (1 << g)] ^ (1 <<
       (k - 1 - g));
   }
   ws[1] = com(1, 0);
   for (int two = 0; two < k - 1; ++
      two) {
    ld alf = pi / n * (1 << (k - 1 -
       two));
    com cur = com(cos(alf), sin(alf))
       ;
    int p2 = (1 << two), p3 = p2 * 2;
    for (int j = p2; j < p3; ++j) {
     ws[j * 2 + 1] = (ws[j * 2] = ws[
        j]) * cur;
    }
   }
  }
  for (int i = 0; i < n; ++i) {
   if (i < dp[i]) {
    swap(a[i], a[dp[i]]);
   }
  }
  if (torev) {
   for (int i = 0; i < n; ++i) {
    a[i].y = -a[i].y;
   }
  }
  for (int len = 1; len < n; len <<=
     1) {
   for (int i = 0; i < n; i += len) {
    int wit = len;
```

```
      for (int it = 0, j = i + len; it
          < len; ++it, ++i, ++j) {
       com tmp = a[j] * ws[wit++];
       a[j] = a[i] - tmp;
       a[i] = a[i] + tmp;
      }
    }
  }
}
com a[maxn];
int mult(int na, int *_a, int nb,
    int *_b, long long *ans) {
 if (!na || !nb) {
  return 0;
 }
 for (k = 0, n = 1; n < na + nb - 1;
     n <<= 1, ++k);
 assert(n < maxn);
 for (int i = 0; i < n; ++i) {
  a[i] = com(i < na ? _a[i] : 0, i <
      nb ? _b[i] : 0);
 }
 fft(a);
 a[n] = a[0];
 for (int i = 0; i <= n - i; ++i) {
  a[i] = (a[i] * a[i] - (a[n - i] *
     a[n - i]).conj()) * com(0, (ld)
     -1 / n / 4);
  a[n - i] = a[i].conj();
 }
 fft(a, 1);
 int res = 0;
 for (int i = 0; i < n; ++i) {
  long long val = (long long)round(a
     [i].x);
  assert(abs(val - a[i].x) < 1e-1);
  if (val) {
    assert(i < na + nb - 1);
    while (res < i) {
     ans[res++] = 0;
    }
    ans[res++] = val;
   }
  }
  return res;
 }
};
```

# Number theory (5)

## 5.1 Modular arithmetic

### ModularArithmetic.h

**Description:** ModularArithmetic.h

247a4f, 25 lines

```
int ceilint(int a, int b) { return (a
     + b - 1) / b; }
int bp(int a, int b) {
    int res = 1;
    while (b > 0) {
        if (b % 2 == 1) res = res * a
            % mod;
        a = a * a % mod;
        b /= 2;
    }
    return res;
}
int fact[MAX], inv_fact[MAX];
void fact_init() {
    fact[0] = 1;
    for (int i = 1; i < MAX; i++) {
        fact[i] = fact[i - 1] * i %
            mod;
    }
    inv_fact[MAX - 1] = bp(fact[MAX -
        1], mod - 2);
    for (int i = MAX - 2; i >= 0; i
        --) {
        inv_fact[i] = inv_fact[i + 1]
            * (i + 1) % mod;
    }
}
int C(int n, int k) {
    if (k > n) return 0;
    return fact[n] * inv_fact[k] %
        mod * inv_fact[n - k] % mod;
}
```

## 5.2 Primality

### FastEratosthenes.h

**Description:** Prime sieve for generating all primes smaller than LIM.

**Time:** LIM=1e9 $\approx$ 1.5s

2ec31f, 12 lines

```
void all_prime_factors(int X){
 int sp[1000000+1]; int prime
     [1000000+1];
 const int range = 1e6;
 for (int i = 2; i <= range; i++){
  if(prime[i]==0){
   sp[i] = i;
   for (int j = i*i; j <= range; j+=i
       ){
    if(prime[j]==0)
    {
     prime[j]=1;
     sp[j] = i;
    }}}}}
```

## 5.3 Divisibility

### euclid.h

**Description:** Finds two integers $x$ and $y$, such that $ax+by = \gcd(a,b)$. If you just need gcd, use the built in `__gcd` instead. If $a$ and $b$ are coprime, then $x$ is the inverse of $a \pmod b$.

33ba8f, 5 lines

```
ll euclid(ll a, ll b, ll &x, ll &y) {
 if (!b) return x = 1, y = 0, a;
 ll d = euclid(b, a % b, y, x);
 return y -= a/b * x, d;
}
```

### CRT.h

**Description:** Chinese Remainder Theorem.

`crt(a, m, b, n)` computes $x$ such that $x \equiv a \pmod m$, $x \equiv b \pmod n$. If $|a| < m$ and $|b| < n$, $x$ will obey $0 \le x < \mathrm{lcm}(m,n)$. Assumes $mn < 2^{62}$.

**Time:** $\log(n)$

"euclid.h" 04d93a, 7 lines

```
ll crt(ll a, ll m, ll b, ll n) {
 if (n > m) swap(a, b), swap(m, n);
 ll x, y, g = euclid(m, n, x, y);
 assert((a - b) % g == 0); // else no
     solution
 x = (b - a) % n * x % n / g * m + a;
 return x < 0 ? x + m*n/g : x;
}
```

### phiFunction.h

**Description:** *Euler's* $\phi$ function is defined as $\phi(n) :=$ # of positive integers $\le n$ that are coprime with $n$. $\phi(1) = 1$, $p$ prime $\Rightarrow \phi(p^k) = (p-1)p^{k-1}$, $m,n$ coprime $\Rightarrow \phi(mn) = \phi(m)\phi(n)$. If $n = p_1^{k_1} p_2^{k_2} ... p_r^{k_r}$ then $\phi(n) = (p_1-1)p_1^{k_1-1}...(p_r-1)p_r^{k_r-1}$. $\phi(n) = n \cdot \prod_{p|n}(1-1/p)$. $\sum_{d|n}\phi(d) = n$, $\sum_{1\le k\le n, \gcd(k,n)=1} k = n\phi(n)/2, n > 1$

**Euler's thm**: $a,n$ coprime $\Rightarrow a^{\phi(n)} \equiv 1 \pmod n$.

**Fermat's little thm**: $p$ prime $\Rightarrow a^{p-1} \equiv 1 \pmod p$ $\forall a$.

cf7d6d, 7 lines

```
const int LIM = 5000000;
int phi[LIM];
void calculatePhi() {
 rep(i,0,LIM) phi[i] = i&1 ? i : i/2;
 for (int i = 3; i < LIM; i += 2) if(
     phi[i] == i)
 for (int j = i; j < LIM; j += i) phi
     [j] -= phi[j] / i;
}
```

## 5.4 Fractions

# Combinatorial (6)

## 6.1 Permutations

### 6.1.1 Cycles

Let $g_S(n)$ be the number of $n$-permutations whose cycle lengths all belong to the set $S$. Then

$$\sum_{n=0}^{\infty} g_S(n)\frac{x^n}{n!} = \exp\left(\sum_{n\in S}\frac{x^n}{n}\right)$$

### 6.1.2 Derangements

Permutations of a set such that none of the elements appear in their original position.

$$D(n) = (n-1)(D(n-1)+D(n-2)) = nD(n-1)+(-1)^n = \left\lfloor \frac{n!}{e} \right\rceil$$

### 6.1.3 Burnside's lemma

Given a group $G$ of symmetries and a set $X$, the number of elements of $X$ *up to symmetry* equals

$$\frac{1}{|G|}\sum_{g\in G}|X^g|,$$

where $X^g$ are the elements fixed by $g$ ($g.x = x$).

If $f(n)$ counts "configurations" (of some sort) of length $n$, we can ignore rotational symmetry using $G = \mathbb{Z}_n$ to get

$$g(n) = \frac{1}{n}\sum_{k=0}^{n-1} f(\gcd(n,k)) = \frac{1}{n}\sum_{k|n} f(k)\phi(n/k).$$

## 6.2 General purpose numbers

### 6.2.1 Stirling numbers of the second kind

Partitions of $n$ distinct elements into exactly $k$ groups.

$$S(n,k) = S(n-1,k-1) + kS(n-1,k)$$

$$S(n,1) = S(n,n) = 1$$

$$S(n,k) = \frac{1}{k!}\sum_{j=0}^{k}(-1)^{k-j}\binom{k}{j}j^n$$

# Graph (7)

## 7.1 Fundamentals

# BellmanFord.h

**Description:** Calculates shortest paths from $s$ in a graph that might have negative edge weights. Unreachable nodes get dist = inf; nodes reachable through negative-weight cycles get dist = -inf. Assumes $V^2 \max |w_i| < \sim 2^{63}$.

**Time:** $\mathcal{O}(VE)$

830a8f, 21 lines

```
const ll inf = LLONG_MAX;
struct Ed { int a, b, w, s() { return
    a < b ? a : -a; }};
struct Node { ll dist = inf; int prev
    = -1; };
void bellmanFord(vector<Node>& nodes,
    vector<Ed>& eds, int s) {
 nodes[s].dist = 0;
 sort(all(eds), [](Ed a, Ed b) {
    return a.s() < b.s(); });
 int lim = sz(nodes) / 2 + 2; //
    /3+100 with shuffled vertices
 rep(i,0,lim) for (Ed ed : eds) {
 Node cur = nodes[ed.a], &dest =
    nodes[ed.b];
 if (abs(cur.dist) == inf) continue;
 ll d = cur.dist + ed.w;
 if (d < dest.dist) {
  dest.prev = ed.a;
  dest.dist = (i < lim-1 ? d : -inf);
 }
 }
 rep(i,0,lim) for (Ed e : eds) {
 if (nodes[e.a].dist == -inf)
  nodes[e.b].dist = -inf;
 }
}
```

# FloydWarshall.h

**Description:** Calculates all-pairs shortest path in a directed graph that might have negative edge weights. Input is an distance matrix $m$, where $m[i][j] = \texttt{inf}$ if $i$ and $j$ are not adjacent. As output, $m[i][j]$ is set to the shortest distance between $i$ and $j$, `inf` if no path, or `-inf` if the path goes through a negative-weight cycle.

**Time:** $\mathcal{O}(N^3)$

531245, 12 lines

```
const ll inf = 1LL << 62;
void floydWarshall(vector<vector<ll
    >>& m) {
 int n = sz(m);
 rep(i,0,n) m[i][i] = min(m[i][i], 0
    LL);
 rep(k,0,n) rep(i,0,n) rep(j,0,n)
 if (m[i][k] != inf && m[k][j] != inf
    ) {
  auto newDist = max(m[i][k] + m[k][j
     ], -inf);
  m[i][j] = min(m[i][j], newDist);
 }
 rep(k,0,n) if (m[k][k] < 0) rep(i,0,
    n) rep(j,0,n)
 if (m[i][k] != inf && m[k][j] != inf
    ) m[i][j] = -inf;
}
```

# TopoSort.h

**Description:** Topological sorting. Given is an oriented graph. Output is an ordering of vertices, such that there are edges only from left to right. If there are cycles, the returned list will have size smaller than $n$ – nodes reachable from cycles will not be returned.

**Time:** $\mathcal{O}(|V|+|E|)$

d678d8, 8 lines

```
vi topoSort(const vector<vi>& gr) {
 vi indeg(sz(gr)), q;
 for (auto& li : gr) for (int x : li)
     indeg[x]++;
 rep(i,0,sz(gr)) if (indeg[i] == 0) q
    .push_back(i);
 rep(j,0,sz(q)) for (int x : gr[q[j
    ]])
 if (--indeg[x] == 0) q.push_back(x);
 return q;
}
```

## 7.2 Network flow

# Flows.h

**Description:** Flow algorithm. Use add and not Eadd.

2a679b, 50 lines

```
const int N = 1000;
template < int N, int Ne > struct
    flows {
 using F = int; // flow type
 F inf = 1e9;
 int n, s, t; // Remember to assign n
    , s and t !
 int ehd[N], cur[N], ev[Ne << 1], enx
    [Ne << 1], eid = 1;
 void clear() {
 eid = 1, memset(ehd, 0, sizeof(ehd))
    ;
 }
 F ew[Ne << 1], dis[N];
 void Eadd(int u, int v, F w) {
 ++eid, enx[eid] = ehd[u], ew[eid] =
    w, ev[eid] = v, ehd[u] = eid;
 }
 void add(int u, int v, F w) {
 Eadd(u, v, w), Eadd(v, u, 0);
 }
 bool bfs() {
 queue < int > q;
```

```
fr(i, 1, n+1) dis[i] = inf, cur[i] = ehd[i];
q.push(s), dis[s] = 0;
while(!q.empty()) {
 int u = q.front();
 q.pop();
 for(int i = ehd[u]; i; i = enx[i])
   if(ew[i] && dis[ev[i]] == inf) {
 dis[ev[i]] = dis[u] + 1, q.push(ev[i]);
 }
}
return dis[t] < inf;
}
F dfs(int x, F now) {
if(!now || x == t) return now;
F res = 0, f;
for(int i = cur[x]; i; i = enx[i]) {
 cur[x] = i;
 if(ew[i] && dis[ev[i]] == dis[x] + 1) {
 f = dfs(ev[i], min(now, ew[i])), ew[i] -= f, now -= f, ew[i ^ 1] += f, res += f;
 if(!now) break;
 }
}
return res;
}
F max_flow() {
F res = 0;
while(bfs())
 {
  res += dfs(s, inf);
 }
return res;
}
} ;
```

## 7.3 Matching

### hopcroftKarp.h

**Description:** Fast bipartite matching algorithm. Graph *g* should be a list of neighbors of the left partition, and *btoa* should be a vector full of -1's of the same size as the right partition. Returns the size of the matching. $btoa[i]$ will be the match for vertex *i* on the right side, or $-1$ if it's not matched.
**Usage:** `vi btoa(m, -1); hopcroftKarp(g, btoa);`
**Time:** $\mathcal{O}\left(\sqrt{V}E\right)$

f612e4, 41 lines

```
bool dfs(int a, int L, vector<vi>& g, vi& btoa, vi& A, vi& B) {
 if (A[a] != L) return 0;
 A[a] = -1;
 for (int b : g[a]) if (B[b] == L + 1) {
 B[b] = 0;
 if (btoa[b] == -1 || dfs(btoa[b], L + 1, g, btoa, A, B))
  return btoa[b] = a, 1;
 }
 return 0;
}
int hopcroftKarp(vector<vi>& g, vi& btoa) {
 int res = 0;
 vi A(g.size()), B(btoa.size()), cur, next;
 for (;;) {
 fill(all(A), 0);
 fill(all(B), 0);
 cur.clear();
 for (int a : btoa) if(a != -1) A[a] = -1;
 rep(a,0,sz(g)) if(A[a] == 0) cur.push_back(a);
 for (int lay = 1;; lay++) {
 bool islast = 0;
 next.clear();
 for (int a : cur) for (int b : g[a]) {
 if (btoa[b] == -1) {
  B[b] = lay;
  islast = 1;
 }
 else if (btoa[b] != a && !B[b]) {
  B[b] = lay;
  next.push_back(btoa[b]);
 }
 }
 if (islast) break;
 if (next.empty()) return res;
 for (int a : next) A[a] = lay;
 cur.swap(next);
 }
 rep(a,0,sz(g))
 res += dfs(a, 0, g, btoa, A, B);
 }
}
```

### BipartiteMatching.h

**Description:** bipartite matching

da1d4b, 67 lines

```
struct bipartite {
 int n, m;
 vector<vector<int>> g;
 vector<bool> paired;
 vector<int> match;
 bipartite(int n, int m): n(n), m(m), g(n), paired(n), match(m, -1) {}

 void add(int a, int b) {
```

```
  g[a].push_back(b);
 }
 vector<size_t> ptr;
 bool kuhn(int v) {
  for(size_t &i = ptr[v]; i < g[v].
      size(); i++) {
   int &u = match[g[v][i]];
   if(u == -1 || (dist[u] == dist[v]
       + 1 && kuhn(u))) {
    u = v;
    paired[v] = true;
    return true;
   }

  }
  return false;
 }
 vector<int> dist;
 bool bfs() {
  dist.assign(n, n);
  int que[n];
  int st = 0, fi = 0;
  for(int v = 0; v < n; v++) {
   if(!paired[v]) {
    dist[v] = 0;
    que[fi++] = v;
   }
  }
  bool rep = false;
  while(st < fi) {
   int v = que[st++];
   for(auto e: g[v]) {
    int u = match[e];
    rep |= u == -1;
    if(u != -1 && dist[v] + 1 < dist[
        u]) {
     dist[u] = dist[v] + 1;
     que[fi++] = u;
    }
   }
  }
  return rep;
 }

 auto matching() {
  while(bfs()) {
   ptr.assign(n, 0);
   for(int v = 0; v < n; v++) {
    if(!paired[v]) {
     kuhn(v);
    }
   }
  }
  vector<pair<int, int>> ans;
  for(int u = 0; u < m; u++) {
   if(match[u] != -1) {
    ans.emplace_back(match[u], u);
   }
  }
  return ans;
 }
};
```

# WeightedMatching.h

**Description:** Given a weighted bipartite graph, matches every node on the left with a node on the right such that no nodes are in two matchings and the sum of the edge weights is minimal. Takes cost[N][M], where cost[i][j] = cost for L[i] to be matched with R[j] and returns (min cost, match), where L[i] is matched with R[match[i]]. Negate costs for max cost. Requires $N \leq M$.
**Time:** $\mathcal{O}\left(N^2M\right)$

a8683a, 75 lines

```
typedef long double ld;
vector<int> hungarian(const vector<
    vector<ld>>& A, int n) {
 // Labels for workers (u) and jobs (
    v)
 vector<ld> u(n + 1, 0.0), v(n + 1,
    0.0);

 // p[j] - the worker assigned to job
    j
 vector<int> p(n + 1, 0);

 // way[j] - the previous job in the
    augmenting path for job j
 vector<int> way(n + 1, 0);

 for(int i = 1; i <= n; ++i){
  p[0] = i;
  int j0 = 0;
  // minv[j] - minimum reduced cost
     for job j
  vector<ld> minv(n + 1, inf);
  // used[j] - whether job j is used
     in the current augmenting path
  vector<bool> used(n + 1, false);

  int j1;
  while(true){
   used[j0] = true;
   int i0 = p[j0];
   ld delta = inf;
   j1 = 0;

   // Iterate over all jobs to find
      the minimum delta
   for(int j = 1; j <= n; ++j){
    if(!used[j]){
```

```
        ld cur = A[i0 - 1][j - 1] - u[i0
            ] - v[j];
        if(cur < minv[j]){
         minv[j] = cur;
         way[j] = j0;
        }
        if(minv[j] < delta){
         delta = minv[j];
         j1 = j;
        }
       }
      }

      // Update labels
      for(int j = 0; j <= n; ++j){
       if(used[j]){
        u[p[j]] += delta;
        v[j] -= delta;
       }
       else{
        minv[j] -= delta;
       }
      }

      j0 = j1;
      if(p[j0] == 0)
       break;
     }

     // Augmenting path: update the
        matching
     do{
      int j1 = way[j0];
      p[j0] = p[j1];
      j0 = j1;
     } while(j0 != 0);
    }

    // Construct the result: ans[i] = j
       means worker i is assigned to job
        j
    vector<int> ans(n, -1);
    for(int j = 1; j <= n; ++j){
     if(p[j] != 0){
      ans[p[j] - 1] = j - 1;
     }
    }

    return ans;
   }
```

## 7.4 DFS algorithms

### SCC.h

**Description:** SCC.h

5a2d60, 49 lines

```
vector<bool> visited; // keeps track
    of which vertices are already
    visited
// runs depth first search starting
    at vertex v.
// each visited vertex is appended to
     the output vector when dfs leaves
     it.
void dfs(int v, vector<vector<int>>
    const& adj, vector<int> &output) {
 visited[v] = true;
 for (auto u : adj[v])
  if (!visited[u])
   dfs(u, adj, output);
 output.push_back(v);
}
// input: adj -- adjacency list of G
// output: components -- the strongy
    connected components in G
// output: adj_cond -- adjacency list
    of G^SCC (by root vertices)
void strongly_connected_components(
    vector<vector<int>> const& adj,
          vector<vector<int>> &
             components,
          vector<vector<int>> &
             adj_cond) {
 int n = adj.size();
 components.clear(), adj_cond.clear()
    ;
 vector<int> order; // will be a
    sorted list of G's vertices by
    exit time
 visited.assign(n, false);
 // first series of depth first
    searches
 for (int i = 0; i < n; i++)
  if (!visited[i])
   dfs(i, adj, order);
 // create adjacency list of G^T
 vector<vector<int>> adj_rev(n);
 for (int v = 0; v < n; v++)
  for (int u : adj[v])
   adj_rev[u].push_back(v);
 visited.assign(n, false);
 reverse(order.begin(), order.end());
 vector<int> roots(n, 0); // gives
    the root vertex of a vertex's SCC
 // second series of depth first
    searches
 for (auto v : order)
  if (!visited[v]) {
   std::vector<int> component;
   dfs(v, adj_rev, component);
   components.push_back(component);
```

```
  int root = *min_element(begin(
     component), end(component));
  for (auto u : component)
   roots[u] = root;
 }
 // add edges to condensation graph
 adj_cond.assign(n, {});
 for (int v = 0; v < n; v++)
  for (auto u : adj[v])
   if (roots[v] != roots[u])
    adj_cond[roots[v]].push_back(
       roots[u]);
}
```

# BiconnectedComponents.h

**Description:** Finds all biconnected components in an undirected graph, and runs a callback for the edges in each. In a biconnected component there are at least two distinct paths between any two nodes. Note that a node can be in several components. An edge which is not in a component is a bridge, i.e., not part of any cycle.

**Usage:** `int eid = 0; ed.resize(N);`
`for each edge (a,b) {`
`ed[a].emplace_back(b, eid);`
`ed[b].emplace_back(a, eid++); }`
`bicomps([&](const vi& edgelist)`
`{...});`

**Time:** $\mathcal{O}(E+V)$

c6b7c7, 31 lines

```
vi num, st;
vector<vector<pii>> ed;
int Time;
template<class F>
int dfs(int at, int par, F& f) {
 int me = num[at] = ++Time, top = me;
 for (auto [y, e] : ed[at]) if (e !=
    par) {
 if (num[y]) {
  top = min(top, num[y]);
  if (num[y] < me)
  st.push_back(e);
 } else {
  int si = sz(st);
  int up = dfs(y, e, f);
  top = min(top, up);
  if (up == me) {
  st.push_back(e);
  f(vi(st.begin() + si, st.end()));
  st.resize(si);
  }
  else if (up < me) st.push_back(e);
  else { /* e is a bridge */ }
 }
 }
 return top;
}
template<class F>
void bicomps(F f) {
 num.assign(sz(ed), 0);
 rep(i,0,sz(ed)) if (!num[i]) dfs(i,
    -1, f);
}
```

# bridges.h

**Description:** Bridges and Articulation Points in a graph calculate low[v] for every vertex low[v] = min(tin[v], tin[to] such that (v,to) is a backedge, note that to is not parent of v, low[to] such that (v,to) is a tree edge, calculate after dfs call)

if(low[to] > tin[v]) then (v,to) is a bridge if(low[to] >= tin[v]) then v is a articulation point

add online bridges implementation

e7c1a5, 99 lines

```
vector<int> par, dsu_2ecc, dsu_cc,
   dsu_cc_size;
int bridges;
int lca_iteration;
vector<int> last_visit;
void init(int n) {
 par.resize(n);
 dsu_2ecc.resize(n);
 dsu_cc.resize(n);
 dsu_cc_size.resize(n);
 lca_iteration = 0;
 last_visit.assign(n, 0);
 for (int i=0; i<n; ++i) {
  dsu_2ecc[i] = i;
  dsu_cc[i] = i;
  dsu_cc_size[i] = 1;
  par[i] = -1;
 }
 bridges = 0;
}
int find_2ecc(int v) {
 if (v == -1)
  return -1;
 return dsu_2ecc[v] == v ? v :
    dsu_2ecc[v] = find_2ecc(dsu_2ecc[
    v]);
}
int find_cc(int v) {
 v = find_2ecc(v);
 return dsu_cc[v] == v ? v : dsu_cc[v
    ] = find_cc(dsu_cc[v]);
}
void make_root(int v) {
 int root = v;
 int child = -1;
 while (v != -1) {
  int p = find_2ecc(par[v]);
```

```
  par[v] = child;
  dsu_cc[v] = root;
  child = v;
  v = p;
 }
 dsu_cc_size[root] = dsu_cc_size[
    child];
}
void merge_path (int a, int b) {
 ++lca_iteration;
 vector<int> path_a, path_b;
 int lca = -1;
 while (lca == -1) {
  if (a != -1) {
   a = find_2ecc(a);
   path_a.push_back(a);
   if (last_visit[a] == lca_iteration
      ){
    lca = a;
    break;
    }
   last_visit[a] = lca_iteration;
   a = par[a];
  }
  if (b != -1) {
   b = find_2ecc(b);
   path_b.push_back(b);
   if (last_visit[b] == lca_iteration
      ){
    lca = b;
    break;
    }
   last_visit[b] = lca_iteration;
   b = par[b];
  }
 }
 for (int v : path_a) {
  dsu_2ecc[v] = lca;
  if (v == lca)
   break;
  --bridges;
 }
 for (int v : path_b) {
  dsu_2ecc[v] = lca;
  if (v == lca)
   break;
  --bridges;
 }
}
void add_edge(int a, int b) {
 a = find_2ecc(a);
 b = find_2ecc(b);
 if (a == b)
  return;
 int ca = find_cc(a);
 int cb = find_cc(b);
 if (ca != cb) {
  ++bridges;
  if (dsu_cc_size[ca] > dsu_cc_size[
     cb]) {
   swap(a, b);
   swap(ca, cb);
  }
  make_root(a);
  par[a] = dsu_cc[a] = b;
  dsu_cc_size[cb] += dsu_cc_size[a];
 } else {
  merge_path(a, b);
 }
}
```

## 2sat.h

**Description:** 2sat.h

3524b8, 60 lines

```
class TwoSAT {
private:
 int n;
 std::vector<std::vector<int>> adj,
    adj_t;
 std::vector<bool> used;
 std::vector<int> order, comp;
 std::vector<bool> assignment;
 void dfs1(int v) {
  used[v] = true;
  for (int u : adj[v]) {
   if (!used[u])
    dfs1(u);
  }
  order.push_back(v);
 }
 void dfs2(int v, int cl) {
  comp[v] = cl;
  for (int u : adj_t[v]) {
   if (comp[u] == -1)
    dfs2(u, cl);
  }
 }
public:
 TwoSAT(int size) : n(size), adj(2 *
    n), adj_t(2 * n), used(2 * n),
    comp(2 * n), assignment(n) {}
 bool solve() {
  order.clear();
  used.assign(2 * n, false);
  for (int i = 0; i < 2 * n; ++i) {
   if (!used[i])
    dfs1(i);
  }
  comp.assign(2 * n, -1);
  for (int i = 0, j = 0; i < 2 * n;
     ++i) {
   int v = order[2 * n - i - 1];
```

```
   if (comp[v] == -1)
    dfs2(v, j++);
  }
  assignment.assign(n, false);
  for (int i = 0; i < 2 * n; i += 2)
     {
   if (comp[i] == comp[i + 1])
    return false;
   assignment[i / 2] = comp[i] > comp
      [i + 1];
  }
  return true;
 }
 void add_disjunction(int a, bool na,
     int b, bool nb) {
  // na and nb signify whether a and
     b are to be negated
  a = 2 * a ^ na;
  b = 2 * b ^ nb;
  int neg_a = a ^ 1;
  int neg_b = b ^ 1;
  adj[neg_a].push_back(b);
  adj[neg_b].push_back(a);
  adj_t[b].push_back(neg_a);
  adj_t[a].push_back(neg_b);
 }
 std::vector<bool> get_assignment() {
  return assignment;
 }
};
```

# EulerWalk.h

**Description:** Eulerian undirected/directed path/cycle algorithm. Input should be a vector of (dest, global edge index), where for undirected graphs, forward/backward edges have the same index. Returns a list of nodes in the Eulerian path/cycle with src at both start and end, or empty list if no cycle/path exists. To get edge indices back, add .second to s and ret.
**Time:** $\mathcal{O}(V+E)$

780b64, 15 lines

```
vi eulerWalk(vector<vector<pii>>& gr,
     int nedges, int src=0) {
 int n = sz(gr);
 vi D(n), its(n), eu(nedges), ret, s
    = {src};
 D[src]++; // to allow Euler paths,
    not just cycles
 while (!s.empty()) {
 int x = s.back(), y, e, &it = its[x
    ], end = sz(gr[x]);
 if (it == end){ ret.push_back(x); s.
    pop_back(); continue; }
 tie(y, e) = gr[x][it++];
 if (!eu[e]) {
  D[x]--, D[y]++;
  eu[e] = 1; s.push_back(y);
 }}
 for (int x : D) if (x < 0 || sz(ret)
     != nedges+1) return {};
 return {ret.rbegin(), ret.rend()};
}
```

## 7.5 Trees

# BinaryLifting.h

**Description:** BinaryLifting.h

cf02b4, 106 lines

```
class Binary_lift{
 public:
  int n,l,timer;
 vector<vector<int>> adj;
 vector<vector<int>> up;
 vector<vector<int>> min_v;
 vector<int> depth;
 vector<int> tin;
 vector<int> tout;
 Binary_lift(int n){
  this->n = n;
  this->l = log2(n)+1;
  adj.resize(n);
  up.resize(n, vector<int>(l, -1));
  min_v.resize(n, vector<int>(l, inf
     ));
  depth.resize(n);tin.resize(n);tout
     .resize(n);
  timer = 0;
 }
 void set_min_v(vi& a){
  fr(i,0,n){
   min_v[i][0] = a[i];}
 }
 void add_edge(int u, int v){
  adj[u].push_back(v);adj[v].
     push_back(u);}
 void dfs(int u, int p, vi& a, int d
    =0){
  up[u][0] = p;
  depth[u] = d;
  tin[u] = timer++;
  for(int i=1;i<l;i++){
   if(up[u][i-1] != -1){
    up[u][i] = up[up[u][i-1]][i-1];
    min_v[u][i] = min(min_v[u][i-1],
         min_v[up[u][i-1]][i-1]);
   }
  }
  for(int v: adj[u]){
```

```
    if(v != p){
     dfs(v, u,a,d+1);}
   }
   tout[u] = timer;
  }

  int lift(int u, int k){
   for(int i=l-1;i>=0;i--){
    if(k >= (1<<i)){
     u = up[u][i];
     k -= (1<<i);
    }
   }
   return u;
  }

  int lca(int u, int v){
   if(depth[u] < depth[v]){
    swap(u,v);
   }
   u = lift(u, depth[u]-depth[v]);
   if(u == v){
    return u;
   }

   for(int i=l-1;i>=0;i--){
    if(depth[u]<(1<<i))
     continue;
    if(up[u][i] != up[v][i]){
     u = up[u][i];
     v = up[v][i];
    }
   }
   return up[u][0];
  }
  int get_kth_node_on_path(int u, int
      v, int k){
   int lca = this->lca(u,v);
   int dist = this->depth[u] + this->
      depth[v] - 2*this->depth[lca];
   if(k > dist){
    return -1;
   }
   if(k == 0){
    return u;
   }
   if(k == dist){
    return v;
   }
   if(this->depth[u] - this->depth[
      lca] >= k){
    return this->lift(u, k);
   }
   return this->lift(v, dist-k);
  }
  int get_min_on_path(int u, int v){
   int lca = this->lca(u,v);
   int ans = inf;
   for(int i=l-1;i>=0;i--){
    if(this->depth[u] - (1<<i) >=
       this->depth[lca]){
     ans = min(ans, this->min_v[u][i
        ]);
     u = this->up[u][i];
    }
   }
   for(int i=l-1;i>=0;i--){
    if(this->depth[v] - (1<<i) >=
       this->depth[lca]){
     ans = min(ans, this->min_v[v][i
        ]);
     v = this->up[v][i];
    }
   }
   ans = min(ans, this->min_v[u][0]);
   ans = min(ans, this->min_v[v][0]);
   return ans;
  }
```

# CentroidDecomposition.h

**Description:** Centroid Decomposition of a tree

2e2603, 59 lines

```
class CentroidDecomposition
{
 // 1 - based indexing
private:
 int n;
 vector<bool> vis;
 vector<int> sz;
 const vector<vector<int>> &tree;

 int find_size(int v, int p = -1)
 {
  if (vis[v])
   return 0;
  sz[v] = 1;
  for (const int &x : tree[v])
   if (x != p)
    sz[v] += find_size(x, v);
  return sz[v];
 }

 int find_centroid(int v, int p, int
    cur_sz)
 {
  for (const int &x : tree[v])
   if (x != p)
    if (!vis[x] && sz[x] > (cur_sz /
       2))
     return find_centroid(x, v,
        cur_sz);
  return v;
```

```
  }

  void init_centroid(int v, int p)
  {
   find_size(v);
   int c = find_centroid(v, -1, sz[v])
       ;
   vis[c] = true;
   centroid_par[c] = p;
   if (p == -1)
    root = c;
   else
    centorid_tree[p].push_back(c);
   for (const int &x : tree[c])
    if (!vis[x])
     init_centroid(x, c);
  }

public:
  vector<vector<int>> centorid_tree;
  vector<int> centroid_par;
  int root;
  CentroidDecomposition(vector<vector<
     int>> &_tree) : tree(_tree)
  {
   root = 1;
   n = tree.size();
   centorid_tree.resize(n);
   vis.resize(n, false);
   sz.resize(n, 0);
   centroid_par.resize(n, -1);
   init_centroid(1, -1);
  }
};
```

# HLD.h

**Description:** Decomposes a tree into vertex disjoint heavy paths and light edges such that the path from any leaf to the root contains at most log(n) light edges. fr(i, 0, n) b[pos[i]] = a[i];

85f97b, 67 lines

```
class HLD{
public:
 vector<int> parent, depth, heavy,
    head, pos;
 int cur_pos;
 vector<vector<int>> adj;

 int dfs(int v) {
  int size = 1;
  int max_c_size = 0;
  for (int c : adj[v]) {
   if (c != parent[v]) {
    parent[c] = v, depth[c] = depth[v
       ] + 1;
    int c_size = dfs(c);
    size += c_size;
    if (c_size > max_c_size)
     max_c_size = c_size, heavy[v] =
        c;
   }
  }
  return size;
 }

 void decompose(int v, int h) {
  head[v] = h, pos[v] = cur_pos++;
  if (heavy[v] != -1)
   decompose(heavy[v], h);
  for (int c : adj[v]) {
   if (c != parent[v] && c != heavy[v
      ])
    decompose(c, c);
  }
 }

 void build()
 {
  dfs(0);
  decompose(0, 0);
 }

 HLD(int n) {
  parent = vector<int>(n);
  depth = vector<int>(n);
  heavy = vector<int>(n, -1);
  head = vector<int>(n);
  pos = vector<int>(n);
  adj = vector<vector<int>>(n);
  cur_pos = 0;
 }

 void add_edge(int u, int v) {
  adj[u].push_back(v);
  adj[v].push_back(u);
 }

 vi query(int a, int b, int x,
    SegmentTree& st) {
  vi res;
  for (; head[a] != head[b]; b =
     parent[head[b]]) {
   if (depth[head[a]] > depth[head[b
      ]])
    swap(a, b);
   vi cur_heavy_path_max = st.query(
      pos[head[b]], pos[b], x);
   for(auto i: cur_heavy_path_max)
      res.pb(i);
  }
```

```
 if (depth[a] > depth[b])
  swap(a, b);
 vi last_heavy_path_max = st.query(
    pos[a], pos[b], x);
 for(auto i: last_heavy_path_max)
    res.pb(i);
 return res;
 }
};
```

## PRIMS.h

**Description:** DirectedMST.h

f4c895, 29 lines

```
class Solution
{
public:
 int spanningTree(int V, vector<
    vector<int>> adj[])
 {
 priority_queue<pair<int, int>,
      vector<pair<int, int> >,
          greater<pair<int, int>>> pq;
 vector<int> vis(V, 0);
 pq.push({0, 0});
 int sum = 0;
 while (!pq.empty()) {
  auto it = pq.top();
  pq.pop();
  int node = it.second;
  int wt = it.first;
  if (vis[node] == 1) continue;
  vis[node] = 1;
  sum += wt;
  for (auto it : adj[node]) {
  int adjNode = it[0];
  int edW = it[1];
  if (!vis[adjNode]) {
   pq.push({edW, adjNode});
  }
  }
 }
 return sum;
 }
};
```

# Geometry (8)

## 8.1 Geometric primitives

### Point.h

**Description:** Class to handle points in the plane. T can be e.g. double or long long. (Avoid int.)

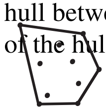
47ec0a, 28 lines

```
template <class T> int sgn(T x) {
   return (x > 0) - (x < 0); }
template<class T>
struct Point {
 typedef Point P;
 T x, y;
 explicit Point(T x=0, T y=0) : x(x),
     y(y) {}
 bool operator<(P p) const { return
    tie(x,y) < tie(p.x,p.y); }
 bool operator==(P p) const { return
    tie(x,y)==tie(p.x,p.y); }
 P operator+(P p) const { return P(x+
    p.x, y+p.y); }
 P operator-(P p) const { return P(x-
    p.x, y-p.y); }
 P operator*(T d) const { return P(x*
    d, y*d); }
 P operator/(T d) const { return P(x/
    d, y/d); }
 T dot(P p) const { return x*p.x + y*
    p.y; }
 T cross(P p) const { return x*p.y -
    y*p.x; }
 T cross(P a, P b) const { return (a
    -*this).cross(b-*this); }
 T dist2() const { return x*x + y*y;
    }
 double dist() const { return sqrt((
    double)dist2()); }
 // angle to x-axis in interval [-pi,
     pi]
 double angle() const { return atan2(
    y, x); }
 P unit() const { return *this/dist()
    ; } // makes dist()=1
 P perp() const { return P(-y, x); }
    // rotates +90 degrees
 P normal() const { return perp().
    unit(); }
 // returns point rotated 'a' radians
     ccw around the origin
 P rotate(double a) const {
 return P(x*cos(a)-y*sin(a),x*sin(a)+
    y*cos(a)); }
 friend ostream& operator<<(ostream&
    os, P p) {
 return os << "(" << p.x << "," << p.
    y << ")"; }
};
```

## 8.2 Polygons

### ConvexHull.h

**Description:**
Returns a vector of the points of the convex hull in counter-clockwise order. Points on the edge of the hull between two other points are not considered part of the hull.



**Time:** $\mathcal{O}(n \log n)$

f5a3ef, 27 lines

```
template <class T> int sgn(T x) {
   return (x > 0) - (x < 0); }
template<class T>
struct Point {
 typedef Point P;
 T x, y;
 explicit Point(T x=0, T y=0) : x(x),
     y(y) {}
 bool operator<(P p) const { return
    tie(x,y) < tie(p.x,p.y); }
 bool operator==(P p) const { return
    tie(x,y)==tie(p.x,p.y); }
 P operator-(P p) const { return P(x-
    p.x, y-p.y); }
 T cross(P a, P b) const { return (a
    -*this).cross(b-*this); }
 T cross(P p) const { return x*p.y -
    y*p.x; }
 friend ostream& operator<<(ostream&
    os, P p) {
 return os << "(" << p.x << "," << p.
    y << ")"; }
};
typedef Point<ll> P;
vector<P> convexHull(vector<P> pts) {
 if (sz(pts) <= 1) return pts;
 sort(all(pts));
 vector<P> h(sz(pts)+1);
 int s = 0, t = 0;
 for (int it = 2; it--; s = --t,
    reverse(all(pts)))
 for (P p : pts) {
  while (t >= s + 2 && h[t-2].cross(h
     [t-1], p) <= 0) t--;
  h[t++] = p;
 }
 return {h.begin(), h.begin() + t - (
    t == 2 && h[0] == h[1])};
}
```

## 8.3 Misc. Point Set Problems

### ClosestPair.h

**Description:** Finds the closest pair of points.

**Time:** $\mathcal{O}(n \log n)$

"Point.h" ac41a6, 17 lines

```
typedef Point<ll> P;
pair<P, P> closest(vector<P> v) {
 assert(sz(v) > 1);
 set<P> S;
 sort(all(v), [](P a, P b) { return a
    .y < b.y; });
 pair<ll, pair<P, P>> ret{LLONG_MAX,
    {P(), P()}};
 int j = 0;
 for (P p : v) {
 P d{1 + (ll)sqrt(ret.first), 0};
 while (v[j].y <= p.y - d.x) S.erase(
    v[j++]);
 auto lo = S.lower_bound(p - d), hi =
     S.upper_bound(p + d);
 for (; lo != hi; ++lo)
  ret = min(ret, {(*lo - p).dist2(),
     {*lo, p}});
 S.insert(p);
 }
 return ret.second;
}
```

# Strings (9)

### KMP.h

**Description:** pi[x] computes the length of the longest prefix of s that ends at x, other than s[0...x] itself (abacaba -> 0010123). Can be used to find all occurrences of a string.

**Time:** $\mathcal{O}(n)$

d4375c, 15 lines

```
vi pi(const string& s) {
 vi p(sz(s));
 rep(i,1,sz(s)) {
 int g = p[i-1];
 while (g && s[i] != s[g]) g = p[g
    -1];
 p[i] = g + (s[i] == s[g]);
 }
 return p;
}
vi match(const string& s, const
   string& pat) {
 vi p = pi(pat + '\0' + s), res;
 rep(i,sz(p)-sz(s),sz(p))
 if (p[i] == sz(pat)) res.push_back(i
     - 2 * sz(pat));
 return res;
}
```

### Zfunc.h

**Description:** z[i] computes the length of the longest common prefix of s[i:] and s, except z[0] = 0. (abacaba -> 0010301)

**Time:** $\mathcal{O}(n)$

ee09e2, 12 lines

```
vi Z(const string& S) {
 vi z(sz(S));
 int l = -1, r = -1;
 rep(i,1,sz(S)) {
 z[i] = i >= r ? 0 : min(r - i, z[i -
     l]);
```

```
 while (i + z[i] < sz(S) && S[i + z[i
    ]] == S[z[i]])
  z[i]++;
 if (i + z[i] > r)
  l = i, r = i + z[i];
 }
 return z;
}
```

## Manacher.h

**Description:** For each position in a string, computes p[0][i] = half length of longest even palindrome around pos i, p[1][i] = longest odd (half rounded down).

**Time:** $\mathcal{O}(N)$

e7ad79, 13 lines

```
array<vi, 2> manacher(const string& s
    ) {
 int n = sz(s);
 array<vi,2> p = {vi(n+1), vi(n)};
 rep(z,0,2) for (int i=0,l=0,r=0; i <
      n; i++) {
 int t = r-i+!z;
 if (i<r) p[z][i] = min(t, p[z][l+t])
    ;
 int L = i-p[z][i], R = i+p[z][i]-!z;
 while (L>=1 && R+1<n && s[L-1] == s[
    R+1])
  p[z][i]++, L--, R++;
 if (R>r) l=L, r=R;
 }
 return p;
}
```

## MinRotation.h

**Description:** Finds the lexicographically smallest rotation of a string.

**Usage:** `rotate(v.begin(), v.begin()+minRotation(v), v.end());`

**Time:** $\mathcal{O}(N)$

d07a42, 8 lines

```
int minRotation(string s) {
 int a=0, N=sz(s); s += s;
 rep(b,0,N) rep(k,0,N) {
 if (a+k == b || s[a+k] < s[b+k]) {b
    += max(0, k-1); break;}
 if (s[a+k] > s[b+k]) { a = b; break;
     }
 }
 return a;
}
```

## SuffixArray.h

**Description:** Builds suffix array for a string. `sa[i]` is the starting index of the suffix which is $i$'th in the sorted suffix array. The returned vector is of size $n+1$, and `sa[0] = n`. The `lcp` array contains longest common prefixes for neighbouring strings in the suffix array: `lcp[i] = lcp(sa[i], sa[i-1])`, `lcp[0] = 0`. The input string must not contain any zero bytes.

**Time:** $\mathcal{O}(n \log n)$

148d7d, 36 lines

```
struct SuffixArray {
 vi sa, lcp;
 SuffixArray(string& s, int lim=256)
    { // or basic_string<int>
 int n = sz(s) + 1, k = 0, a, b;
 vi x(all(s)), y(n), ws(max(n, lim));
 x.push_back(0), sa = lcp = y, iota(
    all(sa), 0);
 for (int j = 0, p = 0; p < n; j =
    max(1, j * 2), lim = p) {
  p = j, iota(all(y), n - j);
  fr(i,0,n) if (sa[i] >= j) y[p++] =
     sa[i] - j;
  fill(all(ws), 0);
  fr(i,0,n) ws[x[i]]++;
  fr(i,1,lim) ws[i] += ws[i - 1];
  for (int i = n; i--;) sa[--ws[x[y[i
     ]]]] = y[i];
  swap(x, y), p = 1, x[sa[0]] = 0;
  fr(i,1,n) a = sa[i - 1], b = sa[i],
      x[b] =
   (y[a] == y[b] && y[a + j] == y[b +
      j]) ? p - 1 : p++;
 }
 for (int i = 0, j; i < n - 1; lcp[x[
    i++]] = k)
  for (k && k--, j = sa[x[i] - 1];
   s[i + k] == s[j + k]; k++);
 }
};
int lower_bound(string& t,vector<int>
    &a,string &s){
 int l = 1,r=sz(a);
 while(l<r){
 int m = (l+r)/2;
 if(s.substr(a[m],min(sz(s)-a[m],sz(t
    )+1)) >= t) r = m;
 else l = m+1;}
 return l;}
int upper_bound(string& t,vector<int>
    &a,string &s){
 int l = 1,r=sz(a);
 while(l<r){
 int m = (l+r)/2;
 if(s.substr(a[m],min(sz(a)-a[m],sz(t
    ))) > t) r = m;
 else l = m+1;}
 return l;}
```

# Hashing.h

**Description:** Self-explanatory methods for string hashing.

2d2a67, 41 lines

```
// Arithmetic mod 2^64-1. 2x slower
    than mod 2^64 and more
// code, but works on evil test data
    (e.g. Thue-Morse, where
// ABBA... and BAAB... of length 2^10
    hash the same mod 2^64).
// "typedef ull H;" instead if you
    think test data is random,
// or work mod 10^9+7 if the Birthday
    paradox is not a problem.
typedef uint64_t ull;
struct H {
 ull x; H(ull x=0) : x(x) {}
 H operator+(H o) { return x + o.x +
    (x + o.x < x); }
 H operator-(H o) { return *this + ~o
    .x; }
 H operator*(H o) { auto m = (
    __uint128_t)x * o.x;
 return H((ull)m) + (ull)(m >> 64); }
 ull get() const { return x + !~x; }
 bool operator==(H o) const { return
    get() == o.get(); }
 bool operator<(H o) const { return
    get() < o.get(); }
};
static const H C = (ll)1e11+3; // (
    order ~ 3e9; random also ok)
struct HashInterval {
 vector<H> ha, pw;
 HashInterval(string& str) : ha(sz(
    str)+1), pw(ha) {
 pw[0] = 1;
 rep(i,0,sz(str))
  ha[i+1] = ha[i] * C + str[i],
  pw[i+1] = pw[i] * C;
 }
 H hashInterval(int a, int b) { //
    hash [a, b)
 return ha[b] - ha[a] * pw[b - a];
 }
};
vector<H> getHashes(string& str, int
    length) {
 if (sz(str) < length) return {};
 H h = 0, pw = 1;
 rep(i,0,length)
 h = h * C + str[i], pw = pw * C;
 vector<H> ret = {h};
 rep(i,length,sz(str)) {
 ret.push_back(h = h * C + str[i] -
    pw * str[i-length]);
 }
 return ret;
}
H hashString(string& s){H h{}; for(
    char c:s) h=h*C+c;return h;}
```

# Trie.h

**Description:** Trie.h

20c980, 131 lines

```
class Trie {
public:
 //N is number of possible characters
    in a string
 const static int N = 26;
 //baseChar defines the base
    character for possible characters
 //like '0' for '0','1','2'... as
    possible characters in string
 const static char baseChar = 'a';
 struct TrieNode
 {
 int next[N];
 //if isEnd is set to true , a string
    ended here
 bool isEnd;
 //freq is how many times this prefix
    occurs
  int freq;
 TrieNode()
 {
  for(int i=0;i<N;i++)
  next[i] = -1;
  isEnd = false;
  freq = 0;
 }
 };
 //the implementation is via vector
    and each position in this vector
 //is similar as new pointer in
    pointer type implementation
 vector <TrieNode> tree;
 //Base Constructor
 Trie ()
 {
 tree.push_back(TrieNode());
 }
 //inserting a string in trie
 void insert(const string &s)
 {
  int p = 0;
  tree[p].freq++;
  for(int i=0;i<s.size();i++)
  {
   // tree[]
   if(tree[p].next[s[i]-baseChar] ==
      -1)
   {
```

```
            tree.push_back(TrieNode());
            tree[p].next[s[i]-baseChar] =
                tree.size()-1;
        }
        p = tree[p].next[s[i]-baseChar];
        tree[p].freq++;
    }
    tree[p].isEnd = true;
}
//check if a string exists as prefix
bool checkPrefix(const string &s)
{
    int p = 0;
    for(int i=0;i<s.size();i++)
    {
    if(tree[p].next[s[i]-baseChar] ==
        -1)
        return false;
    p = tree[p].next[s[i]-baseChar];
    }
    return true;
}
//check is string exists
bool checkString(const string &s)
{
    int p = 0;
    for(int i=0;i<s.size();i++)
    {
    if(tree[p].next[s[i]-baseChar] ==
        -1)
        return false;
    p = tree[p].next[s[i]-baseChar];
    }
    return tree[p].isEnd;
}
//persistent insert
//returns location of new head
int persistentInsert(int head ,
    const string &s)
{
    int old = head;
    tree.push_back(TrieNode());
    int now = tree.size()-1;
    int newHead = now;
    int i,j;
    for(i=0;i<s.size();i++)
    {
    if(old == -1)
    {
        tree.push_back(TrieNode());
        tree[now].next[s[i]-baseChar] =
            tree.size() - 1;
        tree[now].freq++;
        now = tree[now].next[s[i]-baseChar
            ];
        continue;
    }
    for(j=0;j<N;j++)
        tree[now].next[j] = tree[old].next
            [j];
    tree[now].freq = tree[old].freq;
    tree[now].isEnd = tree[old].isEnd;
    tree[now].freq++;

    tree.push_back(TrieNode());
    tree[now].next[s[i]-baseChar] =
        tree.size()-1;
    old = tree[old].next[s[i]-baseChar
        ];
    now = tree[now].next[s[i]-baseChar
        ];
    }
    tree[now].freq++;
    tree[now].isEnd = true;
    return newHead;
}
//persistent check prefix
bool persistentCheckPrefix(int head,
    const string &s)
{
    int p = head;
    for(int i=0;i<s.size();i++)
    {
    if(tree[p].next[s[i]-baseChar] ==
        -1)
        return false;
    p = tree[p].next[s[i]-baseChar];
    }
    return true;
}
//persistent check string
bool persistentCheckString(int head,
    const string &s)
{
    int p = head;
    for(int i=0;i<s.size();i++)
    {
    if(tree[p].next[s[i]-baseChar] ==
        -1)
        return false;
    p = tree[p].next[s[i]-baseChar];
    }
    return tree[p].isEnd;
}
};
```

# Various (10)

## 10.1 Intervals

### IntervalContainer.h

**Description:** IntervalContainer.h

2b074b, 35 lines

```
struct non_overlapping_segment{
 set<pair<int,int>> seg;
 non_overlapping_segment()
 {
  seg.clear();
 }
 int insert(int lo, int hi)
 {
  auto it = seg.upper_bound({lo,0});
  int added = 0;
  if(it != seg.begin())
  {
   --it;
   if((*it).ss >= lo)
   {
    added -= (*it).ss - (*it).ff + 1;
    lo = (*it).ff;
    hi = max(hi,(*it).ss);
    seg.erase(it);
   }
  }
  while(true)
  {
   auto it = seg.lower_bound({lo,0});
   if(it == seg.end()) break;
   if((*it).ff > hi) break;
   hi = max(hi,(*it).ss);
   added -= (*it).ss - (*it).ff + 1;
   seg.erase(it);
  }
  added += hi - lo + 1;
  seg.insert({lo,hi});
  return added;
 }
};
```

### IntervalCover.h

**Description:** Compute indices of smallest set of intervals covering another interval. Intervals should be [inclusive, exclusive). To support [inclusive, inclusive], change (A) to add `|| R.empty()`. Returns empty set on failure (or if G is empty).
**Time:** $\mathcal{O}(N \log N)$

9e9d8d, 19 lines

```
template<class T>
vi cover(pair<T, T> G, vector<pair<T,
    T>> I) {
 vi S(sz(I)), R;
 iota(all(S), 0);
 sort(all(S), [&](int a, int b) {
    return I[a] < I[b]; });
 T cur = G.first;
 int at = 0;
 while (cur < G.second) { // (A)
 pair<T, int> mx = make_pair(cur, -1)
    ;
 while (at < sz(I) && I[S[at]].first
    <= cur) {
  mx = max(mx, make_pair(I[S[at]].
     second, S[at]));
  at++;
 }
 if (mx.second == -1) return {};
 cur = mx.first;
 R.push_back(mx.second);
 }
 return R;
}
```

### ConstantIntervals.h

**Description:** Split a monotone function on [from, to) into a minimal set of half-open intervals on which it has the same value. Runs a callback g for each such interval.
**Usage:** `constantIntervals(0, sz(v), [&](int x){return v[x];}, [&](int lo, int hi, T val){...});`
**Time:** $\mathcal{O}\left(k \log \frac{n}{k}\right)$

753a4c, 19 lines

```
template<class F, class G, class T>
void rec(int from, int to, F& f, G& g
   , int& i, T& p, T q) {
 if (p == q) return;
 if (from == to) {
 g(i, to, p);
 i = to; p = q;
 } else {
 int mid = (from + to) >> 1;
 rec(from, mid, f, g, i, p, f(mid));
 rec(mid+1, to, f, g, i, p, q);
 }
}
template<class F, class G>
void constantIntervals(int from, int
   to, F f, G g) {
 if (to <= from) return;
 int i = from; auto p = f(i), q = f(
    to-1);
 rec(from, to-1, f, g, i, p, q);
 g(i, to, q);
}
```

## 10.2 Misc. algorithms

### TernarySearch.h

**Description:** Find the smallest i in $[a,b]$ that maximizes $f(i)$, assuming that $f(a) < \dots < f(i) \geq \dots \geq f(b)$. To reverse which of the sides allows non-strict inequalities, change the $<$ marked with (A) to <=, and reverse the loop at (B). To minimize $f$, change it to $>$, also at (B).
**Usage:** `int ind = ternSearch(0,n-1,[&](int i){return a[i];});`
**Time:** $\mathcal{O}(\log(b-a))$

9155b4, 11 lines

```
template<class F>
int ternSearch(int a, int b, F f) {
 assert(a <= b);
 while (b - a >= 5) {
 int mid = (a + b) / 2;
 if (f(mid) < f(mid+1)) a = mid; // (A)
 else b = mid+1;
 }
 rep(i,a+1,b+1) if (f(a) < f(i)) a = i; // (B)
 return a;
}
```

# LIS.h

**Description:** Compute indices for the longest increasing subsequence.
**Time:** $\mathcal{O}(N \log N)$

2932a0, 17 lines

```
template<class I> vi lis(const vector<I>& S) {
 if (S.empty()) return {};
 vi prev(sz(S));
 typedef pair<I, int> p;
 vector<p> res;
 rep(i,0,sz(S)) {
 // change 0 -> i for longest non-decreasing subsequence
 auto it = lower_bound(all(res), p{S[i], 0});
 if (it == res.end()) res.emplace_back(), it = res.end()-1;
 *it = {S[i], i};
 prev[i] = it == res.begin() ? 0 : (it-1)->second;
 }
 int L = sz(res), cur = res.back().second;
 vi ans(L);
 while (L--) ans[L] = cur, cur = prev[cur];
 return ans;
}
```

# FastKnapsack.h

**Description:** Given N non-negative integer weights w and a non-negative target t, computes the maximum S <= t such that S is the sum of some subset of the weights.
**Time:** $\mathcal{O}(N \max(w_i))$

b20ccc, 16 lines

```
int knapsack(vi w, int t) {
 int a = 0, b = 0, x;
 while (b < sz(w) && a + w[b] <= t) a += w[b++];
 if (b == sz(w)) return a;
 int m = *max_element(all(w));
 vi u, v(2*m, -1);
 v[a+m-t] = b;
 rep(i,b,sz(w)) {
 u = v;
 rep(x,0,m) v[x+w[i]] = max(v[x+w[i]], u[x]);
 for (x = 2*m; --x > m;) rep(j, max(0,u[x]), v[x])
  v[x-w[j]] = max(v[x-w[j]], j);
 }
 for (a = t; v[a+m-t] < 0; a--) ;
 return a;
}
```

## 10.3 Dynamic programming

# KnuthDP.h

**Description:** When doing DP on intervals: $a[i][j] = \min_{i<k<j}(a[i][k] + a[k][j]) + f(i,j)$, where the (minimal) optimal $k$ increases with both $i$ and $j$, one can solve intervals in increasing order of length, and search $k = p[i][j]$ for $a[i][j]$ only between $p[i][j-1]$ and $p[i+1][j]$. This is known as Knuth DP. Sufficient criteria for this are if $f(b,c) \leq f(a,d)$ and $f(a,c) + f(b,d) \leq f(a,d) + f(b,c)$ for all $a \leq b \leq c \leq d$. Consider also: LineContainer (ch. Data structures), monotone queues, ternary search.
**Time:** $\mathcal{O}(N^2)$