Universidade Federal de Pernambuco

las4s e pelados

Icaro Copo Papel Nunes, Joao Pou Grangeiro, Pedro Grisi

2025-11-20

Contest (1)

template.cpp 14 lines

```
#include <bits/stdc++.h>
using namespace std;

#define rep(i, a, b) for(int i = a; i < (b); ++i)
#define all(x) begin(x), end(x)
#define sz(x) (int)(x).size()
typedef long long ll;
typedef pair<int, int> pii;
typedef vector<int> vi;

int main() {
  cin.tie(0)->sync_with_stdio(0);
  cin.exceptions(cin.failbit);
}
```

.bashrc 2 lines

```
alias c='g++ -Wall -Wconversion -Wfatal-errors -g -std=c++17 \
  -fsanitize=undefined,address'
```

hash.sh 2 lines

```
# bash hash.sh file.cpp l1 l2
sed -n $2','$3' p' $1 | sed '/^#w/d' | cpp -dD -P -
    fpreprocessed | tr -d '[:space:]'| md5sum |cut -c-6
```

troubleshoot.txt 52 lines

```
Pre-submit:
Write a few simple test cases if sample is not enough.
Are time limits close? If so, generate max cases.
Is the memory usage fine?
Could anything overflow?
Make sure to submit the right file.

Wrong answer:
Print your solution! Print debug output, as well.
Are you clearing all data structures between test cases?
Can your algorithm handle the whole range of input?
Read the full problem statement again.
Do you handle all corner cases correctly?
Have you understood the problem correctly?
Any uninitialized variables?
Any overflows?
Confusing N and M, i and j, etc.?
Are you sure your algorithm works?
What special cases have you not thought of?
Are you sure the STL functions you use work as you think?
Add some assertions, maybe resubmit.
Create some testcases to run your algorithm on.
Go through the algorithm for a simple case.
Go through this list again.
Explain your algorithm to a teammate.
Ask the teammate to look at your code.
Go for a small walk, e.g. to the toilet.
Is your output format correct? (including whitespace)
Rewrite your solution from the start or let a teammate do it.

Runtime error:
Have you tested all corner cases locally?
Any uninitialized variables?
Are you reading or writing outside the range of any vector?
Any assertions that might fail?
Any possible division by 0? (mod 0 for example)
Any possible infinite recursion?
Invalidated pointers or iterators?
Are you using too much memory?
Debug with resubmits (e.g. remapped signals, see Various).

Time limit exceeded:
Do you have any possible infinite loops?
What is the complexity of your algorithm?
Are you copying a lot of unnecessary data? (References)
How big is the input and output? (consider scanf)
Avoid vector, map. (use arrays/unordered_map)
What do your teammates think about your algorithm?

Memory limit exceeded:
What is the max amount of memory your algorithm should need?
Are you clearing all data structures between test cases?
```

Mathematics (2)

2.1 Equations

$$ax^2 + bx + c = 0 \Rightarrow x = \frac{-b \pm \sqrt{b^2 - 4ac}}{2a}$$

The extremum is given by $x = -b/2a$.

$$\begin{aligned} ax + by = e \\ cx + dy = f \end{aligned} \Rightarrow \begin{aligned} x = \frac{ed - bf}{ad - bc} \\ y = \frac{af - ec}{ad - bc} \end{aligned}$$

In general, given an equation $Ax = b$, the solution to a variable x_i is given by

$$x_i = \frac{\det A'_i}{\det A}$$

where A'_i is A with the i'th column replaced by b.

2.2 Recurrences

If $a_n = c_1 a_{n-1} + \cdots + c_k a_{n-k}$, and $r_1, \ldots, r_k$ are distinct roots of $x^k - c_1 x^{k-1} - \cdots - c_k$, there are $d_1, \ldots, d_k$ s.t.

$$a_n = d_1 r_1^n + \cdots + d_k r_k^n.$$

Non-distinct roots r become polynomial factors, e.g. $a_n = (d_1 n + d_2) r^n$.

2.3 Trigonometry

$$\sin(v + w) = \sin v \cos w + \cos v \sin w$$
$$\cos(v + w) = \cos v \cos w - \sin v \sin w$$

$$\tan(v + w) = \frac{\tan v + \tan w}{1 - \tan v \tan w}$$
$$\sin v + \sin w = 2 \sin \frac{v + w}{2} \cos \frac{v - w}{2}$$
$$\cos v + \cos w = 2 \cos \frac{v + w}{2} \cos \frac{v - w}{2}$$

$$(V + W) \tan(v - w)/2 = (V - W) \tan(v + w)/2$$

where V, W are lengths of sides opposite angles v, w.

$$a \cos x + b \sin x = r \cos(x - \phi)$$
$$a \sin x + b \cos x = r \sin(x + \phi)$$

where $r = \sqrt{a^2 + b^2}, \phi = \operatorname{atan2}(b, a)$.

2.4 Geometry

2.4.1 Triangles

Side lengths: a, b, c

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

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

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

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

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: $\dfrac{\sin \alpha}{a} = \dfrac{\sin \beta}{b} = \dfrac{\sin \gamma}{c} = \dfrac{1}{2R}$

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

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

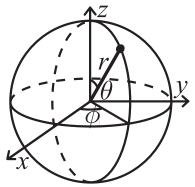
2.4.2 Quadrilaterals

With side lengths a, b, c, d, diagonals e, f, diagonals angle θ, area A and magic flux $F = b^2 + d^2 - a^2 - c^2$:

$$4A = 2ef \cdot \sin \theta = F \tan \theta = \sqrt{4e^2 f^2 - F^2}$$

For cyclic quadrilaterals the sum of opposite angles is $180°$, $ef = ac + bd$, and $A = \sqrt{(p - a)(p - b)(p - c)(p - d)}$.

2.4.3 Spherical coordinates



$$x = r\sin\theta\cos\phi \qquad r = \sqrt{x^2+y^2+z^2}$$
$$y = r\sin\theta\sin\phi \qquad \theta = \mathrm{acos}(z/\sqrt{x^2+y^2+z^2})$$
$$z = r\cos\theta \qquad \phi = \mathrm{atan2}(y,x)$$

2.4.4 Pick's Theorem

The area of a simple polygon whose vertices have integer coordinates is:

$$A = I + \frac{B}{2} - 1$$

where I is the number of interior integer points, and B is the number of integer points in the border of the polygon.

2.4.5 Centroid of a polygon

The coordites of the centroid of a non-self-intersecting closed polygon is:

$$\frac{1}{3A}\left(\sum_{i=0}^{n-1}(x_i+x_{i+1})(x_iy_{i+1}-x_{i+1}y_i), \sum_{i=0}^{n-1}(y_i+y_{i+1})(x_iy_{i+1}-x_{i+1}y_i)\right),$$

where A is twice the signed area of the polygon.

2.4.6 Two Ears Theorem

Every simple polygon with more than 3 vertices has at least two non-overlapping ears (a ear is a vertex whose diagonal induced by its neighbors which lies strictly inside the polygon). Equivalently, every simple polygon can be triangulated.

2.5 Derivatives/Integrals

$$\frac{d}{dx}\arcsin x = \frac{1}{\sqrt{1-x^2}} \qquad \frac{d}{dx}\arccos x = -\frac{1}{\sqrt{1-x^2}}$$
$$\frac{d}{dx}\tan x = 1+\tan^2 x \qquad \frac{d}{dx}\arctan x = \frac{1}{1+x^2}$$
$$\int \tan ax = -\frac{\ln|\cos ax|}{a} \qquad \int x\sin ax = \frac{\sin ax - ax\cos ax}{a^2}$$
$$\int e^{-x^2} = \frac{\sqrt{\pi}}{2}\mathrm{erf}(x) \qquad \int xe^{ax}dx = \frac{e^{ax}}{a^2}(ax-1)$$

Integration by parts:

$$\int_a^b f(x)g(x)dx = [F(x)g(x)]_a^b - \int_a^b F(x)g'(x)dx$$

2.6 Sums

$$c^a + c^{a+1} + \cdots + c^b = \frac{c^{b+1}-c^a}{c-1},\ c \neq 1$$

$$1^2 + 2^2 + \cdots + n^2 = \frac{n(2n+1)(n+1)}{6}$$
$$1^3 + 2^3 + \cdots + n^3 = \frac{n^2(n+1)^2}{4}$$
$$1^4 + 2^4 + \cdots + n^4 = \frac{n(n+1)(2n+1)(3n^2+3n-1)}{30}$$

$$\sum_{i=0}^{n} ic^i = \frac{nc^{n+2}-(n+1)c^{n+1}+c}{(c-1)^2},\ c \neq 1$$

2.7 Series

$$e^x = 1 + x + \frac{x^2}{2!} + \frac{x^3}{3!} + \ldots,\ (-\infty < x < \infty)$$
$$\ln(1+x) = x - \frac{x^2}{2} + \frac{x^3}{3} - \frac{x^4}{4} + \ldots,\ (-1 < x \leq 1)$$
$$\sqrt{1+x} = 1 + \frac{x}{2} - \frac{x^2}{8} + \frac{2x^3}{32} - \frac{5x^4}{128} + \ldots,\ (-1 \leq x \leq 1)$$
$$\sin x = x - \frac{x^3}{3!} + \frac{x^5}{5!} - \frac{x^7}{7!} + \ldots,\ (-\infty < x < \infty)$$
$$\cos x = 1 - \frac{x^2}{2!} + \frac{x^4}{4!} - \frac{x^6}{6!} + \ldots,\ (-\infty < x < \infty)$$
$$\sum_{i=0}^{\infty} ic^i = \frac{c}{(1-c)^2}, \quad |c| < 1$$

2.8 Probability theory

Let X be a discrete random variable with probability $p_X(x)$ of assuming the value x. It will then have an expected value (mean) $\mu = \mathbb{E}(X) = \sum_x xp_X(x)$ and variance $\sigma^2 = V(X) = \mathbb{E}(X^2) - (\mathbb{E}(X))^2 = \sum_x (x-\mathbb{E}(X))^2 p_X(x)$ where σ is the standard deviation. If X is instead continuous it will have a probability density function $f_X(x)$ and the sums above will instead be integrals with $p_X(x)$ replaced by $f_X(x)$.

Expectation is linear:

$$\mathbb{E}(aX+bY) = a\mathbb{E}(X) + b\mathbb{E}(Y)$$

For independent X and Y,

$$V(aX+bY) = a^2V(X) + b^2V(Y).$$

2.8.1 Discrete distributions

Binomial distribution

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

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

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

$\mathrm{Bin}(n,p)$ 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 $\mathrm{Fs}(p)$, $0 \leq p \leq 1$.

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

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

Poisson distribution

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

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

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

2.8.2 Continuous distributions

Uniform distribution

If the probability density function is constant between a and b and 0 elsewhere it is $\mathrm{U}(a,b)$, $a < b$.

$$f(x) = \begin{cases} \frac{1}{b-a} & a < x < b \\ 0 & \text{otherwise} \end{cases}$$

$$\mu = \frac{a+b}{2},\ \sigma^2 = \frac{(b-a)^2}{12}$$

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 μ and variance σ^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

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

2.9 Markov chains

A *Markov chain* is a discrete random process with the property that the next state depends only on the current state. Let $X_1, X_2, \ldots$ be a sequence of random variables generated by the Markov process. Then there is a transition matrix $\mathbf{P} = (p_{ij})$, with $p_{ij} = \Pr(X_n = i | X_{n-1} = j)$, and $\mathbf{p}^{(n)} = \mathbf{P}^n \mathbf{p}^{(0)}$ is the probability distribution for X_n (i.e., $p_i^{(n)} = \Pr(X_n = i)$), where $\mathbf{p}^{(0)}$ is the initial distribution.

π is a stationary distribution if $\pi = \pi\mathbf{P}$. If the Markov chain is *irreducible* (it is possible to get to any state from any state), then $\pi_i = \frac{1}{\mathbb{E}(T_i)}$ where $\mathbb{E}(T_i)$ is the expected time between two visits in state i. π_j/π_i is the expected number of visits in state j between two visits in state i.

For a connected, undirected and non-bipartite graph, where the transition probability is uniform among all neighbors, π_i is proportional to node i's degree.

A Markov chain is *ergodic* if the asymptotic distribution is independent of the initial distribution. A finite Markov chain is ergodic iff it is irreducible and *aperiodic* (i.e., the gcd of cycle lengths is 1). $\lim_{k\to\infty} \mathbf{P}^k = \mathbf{1}\pi$.

A Markov chain is an A-chain if the states can be partitioned into two sets $\mathbf{A}$ and $\mathbf{G}$, such that all states in $\mathbf{A}$ are absorbing ($p_{ii} = 1$), and all states in $\mathbf{G}$ leads to an absorbing state in $\mathbf{A}$. The probability for absorption in state $i \in \mathbf{A}$, when the initial state is j, is $a_{ij} = p_{ij} + \sum_{k\in\mathbf{G}} a_{ik} p_{kj}$. The expected time until absorption, when the initial state is i, is $t_i = 1 + \sum_{k\in\mathbf{G}} p_{ki} t_k$.

Number theory (3)

3.1 Modular arithmetic

ModularArithmetic.h

Description: Operators for modular arithmetic. You need to set `mod` to some number first and then you can use the structure.

"euclid.h" d41d8c, 19 lines

```
d41 const ll mod = 17; // change to something else
d41 struct Mod {
d41  ll x;
d41  Mod(ll xx) : x(xx) {}
d41  Mod operator+(Mod b) { return Mod((x + b.x) % mod); }
d41  Mod operator-(Mod b) { return Mod((x - b.x + mod) % mod);
    }
d41  Mod operator*(Mod b) { return Mod((x * b.x) % mod); }
d41  Mod operator/(Mod b) { return *this * invert(b); }
d41  Mod invert(Mod a) {
d41   ll x, y, g = euclid(a.x, mod, x, y);
d41   assert(g == 1); return Mod((x + mod) % mod);
d41  }
d41  Mod operator^(ll e) {
d41   if (!e) return Mod(1);
d41   Mod r = *this ^ (e / 2); r = r * r;
d41   return e&1 ? *this * r : r;
d41  }
d41 };
```

ModInverse.h

Description: Pre-computation of modular inverses. Assumes LIM $\leq$ mod and that mod is a prime.

d41d8c, 4 lines

```
d41 const ll mod = 1000000007, LIM = 200000;
d41 ll* inv = new ll[LIM] - 1; inv[1] = 1;
d41 rep(i,2,LIM) inv[i] = mod - (mod / i) * inv[mod % i] % mod
    ;
```

ModPow.h

d41d8c, 9 lines

```
d41 const ll mod = 1000000007; // faster if const

d41 ll modpow(ll b, ll e) {
d41  ll ans = 1;
d41  for (; e; b = b * b % mod, e /= 2)
d41   if (e & 1) ans = ans * b % mod;
d41  return ans;
d41 }
```

ModLog.h

Description: Returns the smallest $x > 0$ s.t. $a^x = b \pmod m$, or -1 if no such x exists. modLog(a,1,m) can be used to calculate the order of a.
Time: $\mathcal{O}\left(\sqrt{m}\right)$

d41d8c, 12 lines

```
d41 ll modLog(ll a, ll b, ll m) {
d41  ll n = (ll) sqrt(m) + 1, e = 1, f = 1, j = 1;
d41  unordered_map<ll, ll> A;
d41  while (j <= n && (e = f = e * a % m) != b % m)
d41   A[e * b % m] = j++;
d41  if (e == b % m) return j;
d41  if (__gcd(m, e) == __gcd(m, b))
d41   rep(i,2,n+2) if (A.count(e = e * f % m))
d41    return n * i - A[e];
d41  return -1;
d41 }
```

ModSum.h

Description: Sums of mod'ed arithmetic progressions.
`modsum(to, c, k, m)` $= \sum_{i=0}^{\mathrm{to}-1} (ki + c) \% m$. `divsum` is similar but for floored division.
Time: $\log(m)$, with a large constant.

d41d8c, 17 lines

```
d41 typedef unsigned long long ull;
d41 ull sumsq(ull to) { return to / 2 * ((to-1) | 1); }

d41 ull divsum(ull to, ull c, ull k, ull m) {
d41  ull res = k / m * sumsq(to) + c / m * to;
d41  k %= m; c %= m;
d41  if (!k) return res;
d41  ull to2 = (to * k + c) / m;
d41  return res + (to - 1) * to2 - divsum(to2, m-1 - c, m, k);
d41 }

d41 ll modsum(ull to, ll c, ll k, ll m) {
d41  c = ((c % m) + m) % m;
d41  k = ((k % m) + m) % m;
d41  return to * c + k * sumsq(to) - m * divsum(to, c, k, m);
d41 }
```

ModMulLL.h

Description: Calculate $a \cdot b \bmod c$ (or $a^b \bmod c$) for $0 \leq a, b \leq c \leq 7.2 \cdot 10^{18}$.
Time: $\mathcal{O}(1)$ for `modmul`, $\mathcal{O}(\log b)$ for `modpow`

d41d8c, 12 lines

```
d41 typedef unsigned long long ull;
d41 ull modmul(ull a, ull b, ull M) {
d41  ll ret = a * b - M * ull(1.L / M * a * b);
d41  return ret + M * (ret < 0) - M * (ret >= (ll)M);
d41 }
d41 ull modpow(ull b, ull e, ull mod) {
d41  ull ans = 1;
d41  for (; e; b = modmul(b, b, mod), e /= 2)
d41   if (e & 1) ans = modmul(ans, b, mod);
d41  return ans;
d41 }
```

ModSqrt.h

Description: Tonelli-Shanks algorithm for modular square roots. Finds x s.t. $x^2 = a \pmod p$ ($-x$ gives the other solution).
Time: $\mathcal{O}\left(\log^2 p\right)$ worst case, $\mathcal{O}(\log p)$ for most p

"ModPow.h" d41d8c, 25 lines

```
d41 ll sqrt(ll a, ll p) {
d41  a %= p; if (a < 0) a += p;
d41  if (a == 0) return 0;
d41  assert(modpow(a, (p-1)/2, p) == 1); // else no solution
d41  if (p % 4 == 3) return modpow(a, (p+1)/4, p);
     // a^(n+3)/8 or 2^(n+3)/8 * 2^(n-1)/4 works if p % 8 == 5
d41  ll s = p - 1, n = 2;
d41  int r = 0, m;
d41  while (s % 2 == 0)
d41   ++r, s /= 2;
d41  while (modpow(n, (p - 1) / 2, p) != p - 1) ++n;
d41  ll x = modpow(a, (s + 1) / 2, p);
d41  ll b = modpow(a, s, p), g = modpow(n, s, p);
d41  for (;; r = m) {
d41   ll t = b;
d41   for (m = 0; m < r && t != 1; ++m)
d41    t = t * t % p;
d41   if (m == 0) return x;
d41   ll gs = modpow(g, 1LL << (r - m - 1), p);
d41   g = gs * gs % p;
d41   x = x * gs % p;
d41   b = b * g % p;
d41  }
d41 }
```

3.2 Primality

FastEratosthenes.h

Description: Prime sieve for generating all primes smaller than LIM.
Time: LIM=1e9 $\approx$ 1.5s

d41d8c, 21 lines

```
d41 const int LIM = 1e6;
d41 bitset<LIM> isPrime;
d41 vi eratosthenes() {
d41  const int S = (int)round(sqrt(LIM)), R = LIM / 2;
d41  vi pr = {2}, sieve(S+1); pr.reserve(int(LIM/log(LIM)*1.1)
    );
d41  vector<pii> cp;
d41  for (int i = 3; i <= S; i += 2) if (!sieve[i]) {
d41   cp.push_back({i, i * i / 2});
d41   for (int j = i * i; j <= S; j += 2 * i) sieve[j] = 1;
d41  }
```

```
d41   for (int L = 1; L <= R; L += S) {
d41     array<bool, S> block{};
d41     for (auto &[p, idx] : cp)
d41       for (int i=idx; i < S+L; idx = (i+=p)) block[i-L] =
    1;
d41     rep(i,0,min(S, R - L))
d41       if (!block[i]) pr.push_back((L + i) * 2 + 1);
d41   }
d41   for (int i : pr) isPrime[i] = 1;
d41   return pr;
d41 }
```

MillerRabin.h

Description: Deterministic Miller-Rabin primality test. Guaranteed to work for numbers up to $7 \cdot 10^{18}$; for larger numbers, use Python and extend A randomly.
Time: 7 times the complexity of $a^b \mod c$.

"ModMulLL.h" d41d8c, 13 lines

```
d41 bool isPrime(ull n) {
d41  if (n < 2 || n % 6 % 4 != 1) return (n | 1) == 3;
d41  ull A[] = {2, 325, 9375, 28178, 450775, 9780504,
    1795265022},
d41      s = __builtin_ctzll(n-1), d = n >> s;
d41  for (ull a : A) {   // ^ count trailing zeroes
d41    ull p = modpow(a%n, d, n), i = s;
d41    while (p != 1 && p != n - 1 && a % n && i--)
d41      p = modmul(p, p, n);
d41    if (p != n-1 && i != s) return 0;
d41  }
d41  return 1;
d41 }
```

Factor.h

Description: Pollard-rho randomized factorization algorithm. Returns prime factors of a number, in arbitrary order (e.g. 2299 -> {11, 19, 11}).
Time: $\mathcal{O}\left(n^{1/4}\right)$, less for numbers with small factors.

"ModMulLL.h", "MillerRabin.h" d41d8c, 19 lines

```
d41 ull pollard(ull n) {
d41  ull x = 0, y = 0, t = 30, prd = 2, i = 1, q;
d41  auto f = [&](ull x) { return modmul(x, x, n) + i; };
d41  while (t++ % 40 || __gcd(prd, n) == 1) {
d41    if (x == y) x = ++i, y = f(x);
d41    if ((q = modmul(prd, max(x,y) - min(x,y), n))) prd = q;
d41    x = f(x), y = f(f(y));
d41  }
d41  return __gcd(prd, n);
d41 }
d41 vector<ull> factor(ull n) {
d41  if (n == 1) return {};
d41  if (isPrime(n)) return {n};
d41  ull x = pollard(n);
d41  auto l = factor(x), r = factor(n / x);
d41  l.insert(l.end(), all(r));
d41  return l;
d41 }
```

3.3 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 $a \pmod b$.

d41d8c, 6 lines

```
d41 ll euclid(ll a, ll b, ll &x, ll &y) {
d41  if (!b) return x = 1, y = 0, a;
d41  ll d = euclid(b, a % b, y, x);
d41  return y -= a/b * x, d;
d41 }
```

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 $0 \le x < \operatorname{lcm}(m, n)$. Assumes $mn < 2^{62}$.
Time: $\log(n)$

"euclid.h" d41d8c, 8 lines

```
d41 ll crt(ll a, ll m, ll b, ll n) {
d41  if (n > m) swap(a, b), swap(m, n);
d41  ll x, y, g = euclid(m, n, x, y);
d41  assert((a - b) % g == 0); // else no solution
d41  x = (b - a) % n * x % n / g * m + a;
d41  return x < 0 ? x + m*n/g : x;
d41 }
```

3.3.1 Bézout's identity

For $a \neq$, $b \neq 0$, then $d = gcd(a, b)$ is the smallest positive integer for which there are integer solutions to

$$ax + by = d$$

If (x, y) is one solution, then all solutions are given by

$$\left(x + \frac{kb}{\gcd(a,b)}, y - \frac{ka}{\gcd(a,b)}\right), \quad k \in \mathbb{Z}$$

phiFunction.h

Description: *Euler's* ϕ 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 $n = p_1^{k_1} p_2^{k_2} \ldots p_r^{k_r}$ then $\phi(n) = (p_1 - 1)p_1^{k_1 - 1} \ldots (p_r - 1)p_r^{k_r - 1}$. $\phi(n) = n \cdot \prod_{p|n}(1 - 1/p)$.
$\sum_{d|n} \phi(d) = n$, $\sum_{1 \le k \le n, \gcd(k,n)=1} k = n\phi(n)/2, n > 1$
Euler's thm: a, n coprime $\Rightarrow a^{\phi(n)} \equiv 1 \pmod n$.
Fermat's little thm: p prime $\Rightarrow a^{p-1} \equiv 1 \pmod p$ $\forall a$.

d41d8c, 9 lines

```
d41 const int LIM = 5000000;
d41 int phi[LIM];

d41 void calculatePhi() {
d41  rep(i,0,LIM) phi[i] = i&1 ? i : i/2;
d41  for (int i = 3; i < LIM; i += 2) if(phi[i] == i)
d41    for (int j = i; j < LIM; j += i) phi[j] -= phi[j] / i;
d41 }
```

3.4 Fractions

ContinuedFractions.h

Description: Given N and a real number $x \ge 0$, finds the closest rational approximation p/q with $p, q \le N$. It will obey $|p/q - x| \le 1/qN$.
For consecutive convergents, $p_{k+1}q_k - q_{k+1}p_k = (-1)^k$. ($p_k/q_k$ alternates between $> x$ and $< x$.) If x is rational, y eventually becomes ∞; if x is the root of a degree 2 polynomial the a's eventually become cyclic.
Time: $\mathcal{O}(\log N)$

d41d8c, 22 lines

```
d41 typedef double d; // for N ~ 1e7; long double for N ~ 1e9
d41 pair<ll, ll> approximate(d x, ll N) {
d41  ll LP = 0, LQ = 1, P = 1, Q = 0, inf = LLONG_MAX; d y = x
    ;
d41  for (;;) {
d41    ll lim = min(P ? (N-LP) / P : inf, Q ? (N-LQ) / Q : inf
    ),
d41       a = (ll)floor(y), b = min(a, lim),
d41       NP = b*P + LP, NQ = b*Q + LQ;
d41    if (a > b) {
      // If b > a/2, we have a semi-convergent that gives
          us a
      // better approximation; if b = a/2, we *may* have
          one.
      // Return {P, Q} here for a more canonical
          approximation.
d41      return (abs(x - (d)NP / (d)NQ) < abs(x - (d)P / (d)Q)
    ) ?
d41        make_pair(NP, NQ) : make_pair(P, Q);
d41    }
d41    if (abs(y = 1/(y - (d)a)) > 3*N) {
d41      return {NP, NQ};
d41    }
d41    LP = P; P = NP;
d41    LQ = Q; Q = NQ;
d41  }
d41 }
```

FracBinarySearch.h

Description: Given f and N, finds the smallest fraction $p/q \in [0, 1]$ such that $f(p/q)$ is true, and $p, q \le N$. You may want to throw an exception from f if it finds an exact solution, in which case N can be removed.
Usage: fracBS([](Frac f) { return f.p>=3*f.q; }, 10); // {1,3}
Time: $\mathcal{O}(\log(N))$

d41d8c, 26 lines

```
d41 struct Frac { ll p, q; };

d41 template<class F>
d41 Frac fracBS(F f, ll N) {
d41  bool dir = 1, A = 1, B = 1;
d41  Frac lo{0, 1}, hi{1, 1}; // Set hi to 1/0 to search (0, N
    ]
d41  if (f(lo)) return lo;
d41  assert(f(hi));
d41  while (A || B) {
d41    ll adv = 0, step = 1; // move hi if dir, else lo
d41    for (int si = 0; step; (step *= 2) >>= si) {
d41      adv += step;
d41      Frac mid{lo.p * adv + hi.p, lo.q * adv + hi.q};
d41      if (abs(mid.p) > N || mid.q > N || dir == !f(mid)) {
d41        adv -= step; si = 2;
d41      }
d41    }
d41    hi.p += lo.p * adv;
d41    hi.q += lo.q * adv;
d41    dir = !dir;
d41    swap(lo, hi);
d41    A = B; B = !!adv;
d41  }
d41  return dir ? hi : lo;
d41 }
```

3.5 Primes

$p = 962592769$ is such that $2^{21} \mid p - 1$, which may be useful. For hashing use 970592641 (31-bit number), 31443539979727 (45-bit), 3006703054056749 (52-bit). There are 78498 primes less than 1 000 000.

Primitive roots exist modulo any prime power p^a, except for $p = 2, a > 2$, and there are $\phi(\phi(p^a))$ many. For $p = 2, a > 2$, the group $\mathbb{Z}_{2^a}^\times$ is instead isomorphic to $\mathbb{Z}_2 \times \mathbb{Z}_{2^{a-2}}$.

3.6 Estimates

$\sum_{d|n} d = O(n \log \log n)$.

The number of divisors of n is at most around 100 for $n < 5e4$, 500 for $n < 1e7$, 2000 for $n < 1e10$, 200 000 for $n < 1e19$.

3.7 Mobius Function

$$\mu(n) = \begin{cases} 0 & n \text{ is not square free} \\ 1 & n \text{ has even number of prime factors} \\ -1 & n \text{ has odd number of prime factors} \end{cases}$$

Mobius Inversion:

$$g(n) = \sum_{d|n} f(d) \Leftrightarrow f(n) = \sum_{d|n} \mu(d) g(n/d)$$

Other useful formulas/forms:

$\sum_{d|n} \mu(d) = [n = 1]$ (very useful)

$g(n) = \sum_{n|d} f(d) \Leftrightarrow f(n) = \sum_{n|d} \mu(d/n) g(d)$

$g(n) = \sum_{1 \le m \le n} f(\lfloor \frac{n}{m} \rfloor) \Leftrightarrow f(n) = \sum_{1 \le m \le n} \mu(m) g(\lfloor \frac{n}{m} \rfloor)$

3.8 Theorems

Goldbach's conjecture: Every even integer $n > 2$ can be written as $n = a + b$ with a, b prime.

Legendre's conjecture: There is always at least one prime between n^2 and $(n+1)^2$.

Lagrange's four-square theorem: Every positive integer can be written as

$$n = a^2 + b^2 + c^2 + d^2.$$

Zeckendorf's theorem: Every integer $n \ge 1$ has a unique representation as a sum of non-consecutive Fibonacci numbers:

$$n = F_{i_1} + F_{i_2} + \cdots + F_{i_k}, \quad i_j - i_{j+1} \ge 2.$$

Euclid's formula (primitive Pythagorean triples): The Pythagorean triples are uniquely generated by

$$a = k \cdot (m^2 - n^2), \quad b = k \cdot (2mn), \quad c = k \cdot (m^2 + n^2),$$

with $m > n > 0$, $k > 0$, $m \perp n$, and either m or n even.

Wilson's theorem: n is prime iff

$$(n-1)! \equiv -1 \pmod{n}.$$

Chicken McNugget theorem: For coprime n, m, the largest integer not representable as $an + bm$ (with $a, b \ge 0$) is

$$nm - n - m.$$

There are $\frac{(n-1)(m-1)}{2}$ non-representable integers, and for each pair $(k,\ nm - n - m - k)$ exactly one is representable.

Combinatorial (4)

4.1 Binomial Identities

$$\binom{n-1}{k} - \binom{n-1}{k-1} = \frac{n-2k}{k}\binom{n}{k} \qquad \binom{n}{h}\binom{n-h}{k} = \binom{n}{k}\binom{n-k}{h}$$

$$\sum_{k=0}^{n} k\binom{n}{k} = n2^{n-1} \qquad \sum_{k=0}^{n} k^2\binom{n}{k} = (n+n^2)2^{n-2}$$

$$\sum_{j=0}^{k} \binom{m}{j}\binom{n-m}{k-j} = \binom{n}{k} \qquad \sum_{j=0}^{m} \binom{m}{j}^2 = \binom{2m}{m}$$

$$\sum_{m=0}^{n} \binom{m}{j}\binom{n-m}{k-j} = \binom{n+1}{k+1} \qquad \sum_{m=0}^{n} \binom{m}{k} = \binom{n+1}{k+1}$$

$$\sum_{r=0}^{m} \binom{n+r}{r} = \binom{n+m+1}{m} \qquad \sum_{k=0}^{n} \binom{n-k}{k} = \mathrm{Fib}(n+1)$$

$$\sum_{k=0}^{n} \binom{r}{k}\binom{s}{n-k} = \binom{r+s}{n}$$

4.2 Permutations

4.2.1 Factorial

n	1 2 3	4	5	6	7	8	9	10
$n!$	1 2 6	24	120	720	5040	40320	362880	3628800

n	11	12	13	14	15	16	17
$n!$	4.0e7	4.8e8	6.2e9	8.7e10	1.3e12	2.1e13	3.6e14

n	20	25	30	40	50	100	150	171
$n!$	2e18	2e25	3e32	8e47	3e64	9e157	6e262	>DBL_MAX

IntPerm.h

Description: Permutation -> integer conversion. (Not order preserving.) Integer -> permutation can use a lookup table.

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

d41d8c, 7 lines

```
d41  int permToInt(vi& v) {
d41   int use = 0, i = 0, r = 0;
d41   for(int x:v) r = r * ++i + __builtin_popcount(use & -(1<<
     x)),
d41     use |= 1 << x;                    // (note: minus, not
      ~!)
d41   return r;
d41  }
```

4.2.2 Cycles

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

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

4.2.3 Derangements

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

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

4.2.4 Burnside's lemma

Counts the number of distinct colorings of an object under symmetry.

$$\frac{1}{|G|}\sum_{g \in G} k^{\mathrm{cyc}(g)},$$

where G is the symmetry group, k the number of colors, and $\mathrm{cyc}(g)$ the number of cycles induced by g.

Example: number of ways to color a necklace with n beads using k colors (rotations only):

$$g(n) = \frac{1}{n}\sum_{i=0}^{n-1} k^{(\gcd(n,i))}$$

where rotation i shifts the necklace by i positions.

4.3 Partitions and subsets

4.3.1 Partition function

Number of ways of writing n as a sum of positive integers, disregarding the order of the summands.

$$p(0) = 1, \ p(n) = \sum_{k \in \mathbb{Z} \setminus \{0\}} (-1)^{k+1} p(n - k(3k-1)/2)$$

$$p(n) \sim 0.145/n \cdot \exp(2.56\sqrt{n})$$

n	0	1	2	3	4	5	6	7	8	9	20	50	100
$p(n)$	1	1	2	3	5	7	11	15	22	30	627	~2e5	~2e8

4.3.2 Lucas' Theorem

Let n, m be non-negative integers and p a prime. Write $n = n_k p^k + ... + n_1 p + n_0$ and $m = m_k p^k + ... + m_1 p + m_0$. Then $\binom{n}{m} \equiv \prod_{i=0}^{k} \binom{n_i}{m_i} \pmod{p}$.

4.3.3 Binomials

multinomial.h

Description: Computes $\binom{k_1 + \cdots + k_n}{k_1, k_2, \ldots, k_n} = \frac{(\sum k_i)!}{k_1! k_2! \ldots k_n!}$.

d41d8c, 6 lines

```
d41  ll multinomial(vi& v) {
d41   ll c = 1, m = v.empty() ? 1 : v[0];
d41   rep(i,1,sz(v)) rep(j,0,v[i]) c = c * ++m / (j+1);
d41   return c;
d41  }
```

4.4 General purpose numbers

4.4.1 Stirling numbers of the first kind

Number of permutations on n items with k cycles.

$$c(n,k) = c(n-1,k-1) + (n-1)c(n-1,k), \ c(0,0) = 1$$

$$\sum_{k=0}^{n} c(n,k)x^k = x(x+1)\ldots(x+n-1)$$

$c(8,k) = 8, 0, 5040, 13068, 13132, 6769, 1960, 322, 28, 1$

$c(n,2) = 0, 0, 1, 3, 11, 50, 274, 1764, 13068, 109584, \ldots$

4.4.2 Eulerian numbers

Number of permutations $\pi \in S_n$ in which exactly k elements are greater than the previous element. k j:s s.t. $\pi(j) > \pi(j+1)$, $k+1$ j:s s.t. $\pi(j) \geq j$, k j:s s.t. $\pi(j) > j$.

$$E(n,k) = (n-k)E(n-1,k-1) + (k+1)E(n-1,k)$$

$$E(n,0) = E(n,n-1) = 1$$

$$E(n,k) = \sum_{j=0}^{k} (-1)^j \binom{n+1}{j} (k+1-j)^n$$

4.4.3 Stirling numbers of the second kind

Partitions of n distinct elements into exactly k groups.

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

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

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

4.4.4 Bell numbers

Total number of partitions of n distinct elements. $B(n) = 1, 1, 2, 5, 15, 52, 203, 877, 4140, 21147, \ldots$. For p prime,

$$B(p^m + n) \equiv mB(n) + B(n+1) \pmod{p}$$

4.4.5 Labeled unrooted trees

- on n vertices: n^{n-2}
- on k existing trees of size n_i: $n_1 n_2 \cdots n_k n^{k-2}$
- with degrees d_i: $(n-2)!/((d_1-1)! \cdots (d_n-1)!)$

4.4.6 Catalan numbers

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

$$C_0 = 1,\ C_{n+1} = \frac{2(2n+1)}{n+2} C_n,\ C_{n+1} = \sum C_i C_{n-i}$$

$C_n = 1, 1, 2, 5, 14, 42, 132, 429, 1430, 4862, 16796, 58786, \ldots$

- sub-diagonal monotone paths in an $n \times n$ grid.
- strings with n pairs of parenthesis, correctly nested.
- binary trees with with $n+1$ leaves (0 or 2 children).
- ordered trees with $n+1$ vertices.
- ways a convex polygon with $n+2$ sides can be cut into triangles by connecting vertices with straight lines.
- permutations of $[n]$ with no 3-term increasing subseq.

Various (5)

5.1 Intervals

IntervalContainer.h

Description: Add and remove intervals from a set of disjoint intervals. Will merge the added interval with any overlapping intervals in the set when adding. Intervals are [inclusive, exclusive).
Time: $\mathcal{O}(\log N)$

d41d8c, 24 lines

```
d41  set<pii>::iterator addInterval(set<pii>& is, int L, int R)
      {
d41   if (L == R) return is.end();
d41   auto it = is.lower_bound({L, R}), before = it;
d41   while (it != is.end() && it->first <= R) {
d41     R = max(R, it->second);
d41     before = it = is.erase(it);
d41   }
d41   if (it != is.begin() && (--it)->second >= L) {
d41     L = min(L, it->first);
d41     R = max(R, it->second);
d41     is.erase(it);
d41   }
d41   return is.insert(before, {L,R});
d41  }

d41  void removeInterval(set<pii>& is, int L, int R) {
d41   if (L == R) return;
d41   auto it = addInterval(is, L, R);
d41   auto r2 = it->second;
d41   if (it->first == L) is.erase(it);
d41   else (int&)it->second = L;
d41   if (R != r2) is.emplace(R, r2);
d41  }
```

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

d41d8c, 20 lines

```
d41  template<class T>
d41  vi cover(pair<T, T> G, vector<pair<T, T>> I) {
d41   vi S(sz(I)), R;
d41   iota(all(S), 0);
d41   sort(all(S), [&](int a, int b) { return I[a] < I[b]; });
d41   T cur = G.first;
d41   int at = 0;
d41   while (cur < G.second) { // (A)
d41     pair<T, int> mx = make_pair(cur, -1);
d41     while (at < sz(I) && I[S[at]].first <= cur) {
d41       mx = max(mx, make_pair(I[S[at]].second, S[at]));
d41       at++;
d41     }
d41     if (mx.second == -1) return {};
d41     cur = mx.first;
d41     R.push_back(mx.second);
d41   }
d41   return R;
d41  }
```

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: $\mathcal{O}\left(k \log \frac{n}{k}\right)$

d41d8c, 20 lines

```
d41  template<class F, class G, class T>
d41  void rec(int from, int to, F& f, G& g, int& i, T& p, T q)
      {
d41   if (p == q) return;
d41   if (from == to) {
d41     g(i, to, p);
d41     i = to; p = q;
d41   } else {
d41     int mid = (from + to) >> 1;
d41     rec(from, mid, f, g, i, p, f(mid));
d41     rec(mid+1, to, f, g, i, p, q);
d41   }
d41  }
d41  template<class F, class G>
d41  void constantIntervals(int from, int to, F f, G g) {
d41   if (to <= from) return;
d41   int i = from; auto p = f(i), q = f(to-1);
d41   rec(from, to-1, f, g, i, p, q);
d41   g(i, to, q);
d41  }
```

5.2 Misc. algorithms

TernarySearch.h

Description: Find the smallest i in $[a,b]$ that maximizes $f(i)$, assuming that $f(a) < \ldots < f(i) \geq \cdots \geq f(b)$. 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: $\mathcal{O}(\log(b-a))$

d41d8c, 12 lines

```
d41  template<class F>
d41  int ternSearch(int a, int b, F f) {
d41   assert(a <= b);
d41   while (b - a >= 5) {
d41     int mid = (a + b) / 2;
d41     if (f(mid) < f(mid+1)) a = mid; // (A)
d41     else b = mid+1;
d41   }
d41   rep(i,a+1,b+1) if (f(a) < f(i)) a = i; // (B)
d41   return a;
d41  }
```

LIS.h

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

d41d8c, 18 lines

```
d41  template<class I> vi lis(const vector<I>& S) {
d41   if (S.empty()) return {};
d41   vi prev(sz(S));
d41   typedef pair<I, int> p;
d41   vector<p> res;
d41   rep(i,0,sz(S)) {
        // change 0 -> i for longest non-decreasing subsequence
d41     auto it = lower_bound(all(res), p{S[i], 0});
d41     if (it == res.end()) res.emplace_back(), it = res.end()
     -1;
d41     *it = {S[i], i};
d41     prev[i] = it == res.begin() ? 0 : (it-1)->second;
d41   }
d41   int L = sz(res), cur = res.back().second;
d41   vi ans(L);
d41   while (L--) ans[L] = cur, cur = prev[cur];
d41   return ans;
d41  }
```

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: $\mathcal{O}(N \max(w_i))$

d41d8c, 17 lines

```
d41  int knapsack(vi w, int t) {
d41   int a = 0, b = 0, x;
d41   while (b < sz(w) && a + w[b] <= t) a += w[b++];
d41   if (b == sz(w)) return a;
d41   int m = *max_element(all(w));
d41   vi u, v(2*m, -1);
d41   v[a+m-t] = b;
d41   rep(i,b,sz(w)) {
d41     u = v;
d41     rep(x,0,m) v[x+w[i]] = max(v[x+w[i]], u[x]);
d41     for (x = 2*m; --x > m;) rep(j, max(0,u[x]), v[x])
d41       v[x-w[j]] = max(v[x-w[j]], j);
d41   }
d41   for (a = t; v[a+m-t] < 0; a--) ;
d41   return a;
d41  }
```

5.3 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: $\mathcal{O}(N^2)$

d41d8c, 2 lines

DivideAndConquerDP.h

Description: Given $a[i] = \min_{lo(i) \leq k < hi(i)}(f(i,k))$ where the (minimal) optimal k increases with i, computes $a[i]$ for $i = L..R-1$.
Time: $\mathcal{O}((N + (hi - lo)) \log N)$

d41d8c, 19 lines

```
d41  struct DP { // Modify at will:
d41   int lo(int ind) { return 0; }
d41   int hi(int ind) { return ind; }
d41   ll f(int ind, int k) { return dp[ind][k]; }
d41   void store(int ind, int k, ll v) { res[ind] = pii(k, v);
     }

d41   void rec(int L, int R, int LO, int HI) {
d41     if (L >= R) return;
d41     int mid = (L + R) >> 1;
d41     pair<ll, int> best(LLONG_MAX, LO);
d41     rep(k, max(LO,lo(mid)), min(HI,hi(mid)))
d41       best = min(best, make_pair(f(mid, k), k));
d41     store(mid, best.second, best.first);
d41     rec(L, mid, LO, best.second+1);
d41     rec(mid+1, R, best.second, HI);
d41   }
d41   void solve(int L, int R) { rec(L, R, INT_MIN, INT_MAX); }
d41  };
```

5.4 Debugging tricks

- `signal(SIGSEGV, [](int) { _Exit(0); });` converts segfaults into Wrong Answers. Similarly one can catch SIGABRT (assertion failures) and SIGFPE (zero divisions). `_GLIBCXX_DEBUG` failures generate SIGABRT (or SIGSEGV on gcc 5.4.0 apparently).
- `feenableexcept(29);` kills the program on NaNs (`1`), 0-divs (`4`), infinities (`8`) and denormals (`16`).

5.5 Optimization tricks

`__builtin_ia32_ldmxcsr(40896);` disables denormals (which make floats 20x slower near their minimum value).

5.5.1 Bit hacks

- `x & -x` is the least bit in `x`.
- `for (int x = m; x; ) { --x &= m; ... }` loops over all subset masks of `m` (except `m` itself).
- `c = x&-x, r = x+c; (((r^x) >> 2)/c) | r` is the next number after `x` with the same number of bits set.
- `rep(b,0,K) rep(i,0,(1 << K))`
  `if (i & 1 << b) D[i] += D[i^(1 << b)];`
  computes all sums of subsets.

5.5.2 Pragmas

- `#pragma GCC optimize ("Ofast")` will make GCC auto-vectorize loops and optimizes floating points better.
- `#pragma GCC target ("avx2")` can double performance of vectorized code, but causes crashes on old machines.
- `#pragma GCC optimize ("trapv")` kills the program on integer overflows (but is really slow).

FastMod.h

Description: Compute $a\%b$ about 5 times faster than usual, where b is constant but not known at compile time. Returns a value congruent to a (mod b) in the range $[0, 2b)$.

d41d8c, 9 lines

```
d41  typedef unsigned long long ull;
d41  struct FastMod {
d41   ull b, m;
d41   FastMod(ull b) : b(b), m(-1ULL / b) {}
d41   ull reduce(ull a) { // a % b + (0 or b)
d41     return a - (ull)((__uint128_t(m) * a) >> 64) * b;
d41   }
d41  };
```

FastInput.h

Description: Read an integer from stdin. Usage requires your program to pipe in input from file.
Usage: `./a.out < input.txt`
Time: About 5x as fast as cin/scanf.

d41d8c, 18 lines

```
d41  inline char gc() { // like getchar()
d41   static char buf[1 << 16];
d41   static size_t bc, be;
d41   if (bc >= be) {
d41     buf[0] = 0, bc = 0;
d41     be = fread(buf, 1, sizeof(buf), stdin);
d41   }
d41   return buf[bc++]; // returns 0 on EOF
d41  }

d41  int readInt() {
d41   int a, c;
d41   while ((a = gc()) < 40);
d41   if (a == '-') return -readInt();
d41   while ((c = gc()) >= 48) a = a * 10 + c - 480;
d41   return a - 48;
d41  }
```

BumpAllocator.h

Description: When you need to dynamically allocate many objects and don't care about freeing them. "new X" otherwise has an overhead of something like 0.05us + 16 bytes per allocation.

d41d8c, 9 lines

```
     // Either globally or in a single class:
d41  static char buf[450 << 20];
d41  void* operator new(size_t s) {
d41   static size_t i = sizeof buf;
d41   assert(s < i);
d41   return (void*)&buf[i -= s];
d41  }
d41  void operator delete(void*) {}
```

SmallPtr.h

Description: A 32-bit pointer that points into BumpAllocator memory.

"BumpAllocator.h" d41d8c, 11 lines

```
d41  template<class T> struct ptr {
d41   unsigned ind;
d41   ptr(T* p = 0) : ind(p ? unsigned((char*)p - buf) : 0) {
d41     assert(ind < sizeof buf);
d41   }
d41   T& operator*() const { return *(T*)(buf + ind); }
d41   T* operator->() const { return &**this; }
d41   T& operator[](int a) const { return (&**this)[a]; }
d41   explicit operator bool() const { return ind; }
d41  };
```

BumpAllocatorSTL.h

Description: BumpAllocator for STL containers.
Usage: `vector<vector<int, small<int>>> ed(N);`

d41d8c, 15 lines

```
d41  char buf[450 << 20] alignas(16);
d41  size_t buf_ind = sizeof buf;

d41  template<class T> struct small {
d41   typedef T value_type;
d41   small() {}
d41   template<class U> small(const U&) {}
d41   T* allocate(size_t n) {
d41     buf_ind -= n * sizeof(T);
d41     buf_ind &= 0 - alignof(T);
d41     return (T*)(buf + buf_ind);
d41   }
d41   void deallocate(T*, size_t) {}
d41  };
```

SIMD.h

Description: Cheat sheet of SSE/AVX intrinsics, for doing arithmetic on several numbers at once. Can provide a constant factor improvement of about 4, orthogonal to loop unrolling. Operations follow the pattern `"_mm(256)?_name_(si(128|256)|epi(8|16|32|64)|pd|ps)"`. Not all are described here; grep for `_mm_` in `/usr/lib/gcc/*/4.9/include/` for more. If AVX is unsupported, try 128-bit operations, "emmintrin.h" and #define `__SSE__` and `__MMX__` before including it. For aligned memory use `_mm_malloc(size, 32)` or `int buf[N] alignas(32)`, but prefer loadu/storeu.

d41d8c, 44 lines

```
d41  #pragma GCC target ("avx2") // or sse4.1
d41  #include "immintrin.h"

d41  typedef __m256i mi;
d41  #define L(x) _mm256_loadu_si256((mi*)&(x))
```

```
// High-level/specific methods:
// load(u)?_si256, store(u)?_si256, setzero_si256, _mm_malloc
// blendv_(epi8|ps|pd) (z?y:x), movemask_epi8 (hibits of bytes)
// i32gather_epi32(addr, x, 4): map addr[] over 32-b parts of x
// sad_epu8: sum of absolute differences of u8, outputs 4 xi64
// maddubs_epi16: dot product of unsigned i7's, outputs 16 xi15
// madd_epi16: dot product of signed i16's, outputs 8xi32
// extractf128_si256(, i) (256->128), cvtsi128_si32 (128-> lo32)
// permute2f128_si256(x,x,1) swaps 128-bit lanes
// shuffle_epi32(x, 3*64+2*16+1*4+0) == x for each lane
// shuffle_epi8(x, y) takes a vector instead of an imm

// Methods that work with most data types (append e.g. _epi32):
// set1, blend (i8?x:y), add, adds (sat.), mullo, sub, and/or,
// andnot, abs, min, max, sign(1,x), cmp(gt|eq), unpack(lo|hi)

d41 int sumi32(mi m) { union {int v[8]; mi m;} u; u.m = m;
d41  int ret = 0; rep(i,0,8) ret += u.v[i]; return ret; }
d41 mi zero() { return _mm256_setzero_si256(); }
d41 mi one() { return _mm256_set1_epi32(-1); }
d41 bool all_zero(mi m) { return _mm256_testz_si256(m, m); }
d41 bool all_one(mi m) { return _mm256_testc_si256(m, one());
    }

d41 ll example_filteredDotProduct(int n, short* a, short* b) {
d41  int i = 0; ll r = 0;
d41  mi zero = _mm256_setzero_si256(), acc = zero;
d41  while (i + 16 <= n) {
d41    mi va = L(a[i]), vb = L(b[i]); i += 16;
d41    va = _mm256_and_si256(_mm256_cmpgt_epi16(vb, va), va);
d41    mi vp = _mm256_madd_epi16(va, vb);
d41    acc = _mm256_add_epi64(_mm256_unpacklo_epi32(vp, zero),
d41      _mm256_add_epi64(acc, _mm256_unpackhi_epi32(vp, zero)
    ));
d41  }
d41  union {ll v[4]; mi m;} u; u.m = acc; rep(i,0,4) r += u.v[
    i];
d41  for (;i<n;++i) if (a[i] < b[i]) r += a[i]*b[i]; // <-
    equiv
d41  return r;
d41 }
```

Techniques (A)

techniques.txt

159 lines

```
Recursion
Divide and conquer
  Finding interesting points in N log N
Algorithm analysis
  Master theorem
  Amortized time complexity
Greedy algorithm
  Scheduling
  Max contiguous subvector sum
  Invariants
  Huffman encoding
Graph theory
  Dynamic graphs (extra book-keeping)
  Breadth first search
  Depth first search
  * Normal trees / DFS trees
  Dijkstra's algorithm
  MST: Prim's algorithm
  Bellman-Ford
  Konig's theorem and vertex cover
  Min-cost max flow
  Lovasz toggle
  Matrix tree theorem
  Maximal matching, general graphs
  Hopcroft-Karp
  Hall's marriage theorem
  Graphical sequences
  Floyd-Warshall
  Euler cycles
  Flow networks
  * Augmenting paths
  * Edmonds-Karp
  Bipartite matching
  Min. path cover
  Topological sorting
  Strongly connected components
  2-SAT
  Cut vertices, cut-edges and biconnected components
  Edge coloring
  * Trees
  Vertex coloring
  * Bipartite graphs (=> trees)
  * 3^n (special case of set cover)
  Diameter and centroid
  K'th shortest path
  Shortest cycle
Dynamic programming
  Knapsack
  Coin change
  Longest common subsequence
  Longest increasing subsequence
  Number of paths in a dag
  Shortest path in a dag
  Dynprog over intervals
  Dynprog over subsets
  Dynprog over probabilities
  Dynprog over trees
  3^n set cover
  Divide and conquer
  Knuth optimization
  Convex hull optimizations
  RMQ (sparse table a.k.a 2^k-jumps)
  Bitonic cycle
  Log partitioning (loop over most restricted)
Combinatorics
  Computation of binomial coefficients
  Pigeon-hole principle
  Inclusion/exclusion
  Catalan number
  Pick's theorem
Number theory
  Integer parts
  Divisibility
  Euclidean algorithm
  Modular arithmetic
  * Modular multiplication
  * Modular inverses
  * Modular exponentiation by squaring
  Chinese remainder theorem
  Fermat's little theorem
  Euler's theorem
  Phi function
  Frobenius number
  Quadratic reciprocity
  Pollard-Rho
  Miller-Rabin
  Hensel lifting
  Vieta root jumping
Game theory
  Combinatorial games
  Game trees
  Mini-max
  Nim
  Games on graphs
  Games on graphs with loops
  Grundy numbers
  Bipartite games without repetition
  General games without repetition
  Alpha-beta pruning
Probability theory
Optimization
  Binary search
  Ternary search
  Unimodality and convex functions
  Binary search on derivative
Numerical methods
  Numeric integration
  Newton's method
  Root-finding with binary/ternary search
  Golden section search
Matrices
  Gaussian elimination
  Exponentiation by squaring
Sorting
  Radix sort
Geometry
  Coordinates and vectors
  * Cross product
  * Scalar product
  Convex hull
  Polygon cut
  Closest pair
  Coordinate-compression
  Quadtrees
  KD-trees
  All segment-segment intersection
Sweeping
  Discretization (convert to events and sweep)
  Angle sweeping
  Line sweeping
  Discrete second derivatives
Strings
  Longest common substring
  Palindrome subsequences
  Knuth-Morris-Pratt
  Tries
  Rolling polynomial hashes
  Suffix array
  Suffix tree
  Aho-Corasick
  Manacher's algorithm
  Letter position lists
Combinatorial search
  Meet in the middle
  Brute-force with pruning
  Best-first (A*)
  Bidirectional search
  Iterative deepening DFS / A*
Data structures
  LCA (2^k-jumps in trees in general)
  Pull/push-technique on trees
  Heavy-light decomposition
  Centroid decomposition
  Lazy propagation
  Self-balancing trees
  Convex hull trick (wcipeg.com/wiki/Convex_hull_trick)
  Monotone queues / monotone stacks / sliding queues
  Sliding queue using 2 stacks
  Persistent segment tree
```