TCR

git diff solution (Jens Heuseveldt, Ludo Pulles, Pim Spelier)

Contents

At the start of a contest, type this in a terminal:

```
printf "set nu sw=4 ts=4 sts=4 noet ai hls shellcmdflag=-ic\nsy on
    colo slate" > .vimrc
printf "\nalias gsubmit='g++ -Wall -Wshadow -std=c++11'" >> .bashrc
printf "\nalias g11='gsubmit -DLOCAL -g'" >> .bashrc
. .bashrc
mkdir contest; cd contest
```

template.cpp

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

// Order statistics tree (if supported by judge!):
#include <ext/pb_ds/assoc_container.hpp>
#include <ext/pb_ds/tree_policy.hpp>
using namespace __gnu_pbds;

template<class TK, class TM>
using order_tree = tree<TK, TM, less<TK>, rb_tree_tag,
    tree_order_statistics_node_update>;
// iterator  find_by_order(int r) (zero based)
// int    order_of_key(TK v)
template<class TV> using order_set = order_tree<TV,
    null_type>;

#define x first
#define y second
#define pb push_back
#define eb emplace_back
#define rep(i,a,b) for(auto i=(a);i!=(b); ++i)
#define all(v) (v).begin(), (v).end()
#define rs resize

typedef long long ll;
typedef pair<int, int> pii;
typedef vector<int> vi;
typedef vector<vi> vvi;
template<class T> using min_queue = priority_queue<T,
    vector<T>, greater<T>>;

const int INF = 2147483647; // (1 << 30) - 1 + (1 << 30)
const ll LLINF = (1LL << 62) - 1 + (1LL << 62); // =
    9.223.372.036.854.775.807
const double PI = acos(-1.0);

#ifdef LOCAL
#define DBG(x) cerr << __LINE__ << ": " << #x << " = " << (x)
    << endl
#else
#define DBG(x)
const bool LOCAL = false;
#endif

void Log() { if(LOCAL) cerr << "\n\n"; }
template<class T, class... S>
void Log(T t, S... s) { if(LOCAL) cerr << t << "\t",
    Log(s...); }

// lambda-expression: [] (args) -> retType { body }
int main() {
  ios_base::sync_with_stdio(false); // fast IO
  cin.tie(NULL); // fast IO
  cerr << boolalpha; // print true/false
  (cout << fixed).precision(10); // adjust precision

  return 0;
}
```

Prime numbers: *982451653*, *81253449*, $10^3+\{-9,-3,9,13\}$, $10^6+\{-17,3,33\}$, $10^9+\{7,9,21,33,87\}$

0.1. De winnende aanpak.

- Goed slapen & een vroeg ritme hebben
- Genoeg drinken & eten voor en tijdens de wedstrijd
- Een lijst van alle problemen met info waar het over gaat, en wie het goed kan oplossen
- Ludo moet **ALLE** opgaves **goed** lezen
- Test de kleine voorbeeldgevallen
- Houd na 2 uur een pauze en overleg waar iedereen mee bezig is
- Maak zelf wat test-cases
- Typ de dingen uit de TCR, die je zeker nodig hebt, alvast in
- Als iemand niks te doen heeft, kan hij nodige dingen uit de TCR typen.
- We moeten ook een voorbeeld test-case voor TCR algoritmes hebben om te testen of het goed overgetypt is
- Bij geometrie moeten we om kunnen gaan met meerdere input manieren (voor bv. lijnen)
- Gebruik veel long long's

0.2. Wrong Answer.

(1) Print de oplossing om te debuggen! Kijk ook naar andere (mogelijk makkelijkere) problemen.
(2) Bedenk zelf test-cases met **randgevallen**!
(3) Controleer op **overflow** (gebruik **OVERAL** long long, long double).
Kijk naar overflows in tussenantwoorden bij modulo.
(4) Controleer de **precisie**.
(5) Controleer op **typo's**.
(6) Loop de voorbeeldinput accuraat langs.
(7) Controller op off-by-one-errors (in indices of lus-grenzen)?

0.3. Detecting overflow.

These are GNU builtins, detect both over- and underflow. Returns a boolean upon failure, otherwise the result is present in `ref`. Follow the template:

```cpp
bool isOverflown = __builtin_[add|mul|sub]_overflow(a, b, \&res);
```

0.4. Covering problems.

Minimum edge cover $\Longleftrightarrow$ *Maximum independent set*

Matching: A set of edges without common vertices *(Maximum is the* ***largest*** *such set, maximal is a set which you cannot add more edges to without breaking the property).*

Minimum Vertex Cover: A set vertices (cover) such that each edge in the graph is incident to at least one vertex of the set.

Minimum Edge Cover: A set of edges (cover) such that every vertex is incident to at least one edge of the set.

Maximum Independent Set: A set of vertices in a graph such that no two of them are adjacent.

König's theorem: In any bipartite graph, the number of edges in a maximum matching equals the number of vertices in a minimum vertex cover

0.5. **Game theory.** A game can be reduced to Nim if it is a finite impartial game. Nim and its variants include:

Nim: Let $X = \bigoplus_{i=1}^n x_i$, then $(x_i)_{i=1}^n$ is a winning position iff $X \neq 0$. Find a move by picking k such that $x_k > x_k \oplus X$.

Misère Nim: Regular Nim, except that the last player to move *loses*. Play regular Nim until there is only one pile of size larger than 1, reduce it to 0 or 1 such that there is an odd number of piles.

Staricase Nim: Stones are moved down a staircase and only removed from the last pile. $(x_i)_{i=1}^n$ is an L-position if $(x_{2i-1})_{i=1}^{n/2}$ is (i.e. only look at odd-numbered piles).

Moore's Nim$_k$: The player may remove from at most k piles (Nim $=$ Nim$_1$). Expand the piles in base 2, do a carry-less addition in base $k+1$ (i.e. the number of ones in each column should be divisible by $k+1$).

Dim$^+$: The number of removed stones must be a divisor of the pile size. The Sprague-Grundy function is $k+1$ where 2^k is the largest power of 2 dividing the pile size.

Aliquot game: Same as above, except the divisor should be proper (hence 1 is also a terminal state, but watch out for size 0 piles). Now the Sprague-Grundy function is just k.

Nim (at most half): Write $n+1 = 2^m y$ with m maximal, then the Sprague-Grundy function of n is $(y-1)/2$.

Lasker's Nim: Players may alternatively split a pile into two new non-empty piles. $g(4k+1) = 4k+1$, $g(4k+2) = 4k+2$, $g(4k+3) = 4k+4$, $g(4k+4) = 4k+3$ $(k \geq 0)$.

Hackenbush on trees: A tree with stalks $(x_i)_{i=1}^n$ may be replaced with a single stalk with length $\bigoplus_{i=1}^n x_i$.

A useful identity: $\bigoplus_{x=0}^{a-1} x = \{0, a-1, 1, a\}[a \bmod 4]$.

1. Mathematics

```
int abs(int x) { return x > 0 ? x : -x; }
int sign(int x) { return (x > 0) - (x < 0); }

// greatest common divisor
ll gcd(ll a, ll b) { while (b) a %= b, swap(a, b); return a; }
// least common multiple
ll lcm(ll a, ll b) { return a / gcd(a, b) * b; }
ll mod(ll a, ll b) { return (a %= b) < 0 ? a + b : a; }

// safe multiplication (ab % m) for m <= 4e18 in O(log b)
ll mulmod(ll a, ll b, ll m) {
  ll r = 0;
  while (b) {
    if (b & 1) r = (r + a) % m; a = (a + a) % m; b >>= 1;
  }
  return r;
}

// safe exponentation (a^b % m) for m <= 2e9 in O(log b)
ll powmod(ll a, ll b, ll m) {
  ll r = 1;
  while (b) {
    if (b & 1) r = (r * a) % m; // r = mulmod(r, a, m);
    a = (a * a) % m; // a = mulmod(a, a, m);
    b >>= 1;
  }
  return r;
}

// returns x, y such that ax + by = gcd(a, b)
ll egcd(ll a, ll b, ll &x, ll &y) {
  ll xx = y = 0, yy = x = 1;
  while (b) {
    x -= a / b * xx; swap(x, xx);
    y -= a / b * yy; swap(y, yy);
    a %= b; swap(a, b);
  }
  return a;
}

// Chinese remainder theorem
const pll NO_SOLUTION(0, -1);
// Returns (u, v) such that x = u % v <=> x = a % n and x = b % m
pll crt(ll a, ll n, ll b, ll m) {
  ll s, t, d = egcd(n, m, s, t), nm = n * m;
  if (mod(a - b, d)) return NO_SOLUTION;
  return pll(mod(s * b * n + t * a * m, nm) / d, nm / d);
  /* when n, m > 10^6, avoid overflow:
  return pll(mod(mulmod(mulmod(s, b, nm), n, nm)
             + mulmod(mulmod(t, a, nm), m, nm), nm) / d, nm / d); */
}

// phi[i] = #{ 0 < j <= i | gcd(i, j) = 1 }
vi totient(int N) {
  vi phi(N);
  for (int i = 0; i < N; i++) phi[i] = i;
  for (int i = 2; i < N; i++)
    if (phi[i] == i)
      for (int j = i; j < N; j += i) phi[j] -= phi[j] / i;
  return phi;
}

// calculate nCk % p (p prime!)
ll lucas(ll n, ll k, ll p) {
  ll ans = 1;
  while (n) {
    ll np = n % p, kp = k % p;
    if (np < kp) return 0;
    ans = mod(ans * binom(np, kp), p); // (np C kp)
    n /= p; k /= p;
  }
  return ans;
}

// returns if n is prime for n < 3e24 ( > 2^64)
bool millerRabin(ll n){
  if (n < 2 || n % 2 == 0) return n == 2;
  ll d = n - 1, ad, s = 0, r;
  for (; d % 2 == 0; d /= 2) s++;
  for (int a : { 2, 3, 5, 7, 11, 13, 17, 19, 23, 29, 31, 37, 41 }) {
    if (n == a) return true;
    if ((ad = powmod(a, d, n)) == 1) continue;
    for (r = 0; r < s && ad + 1 != n; r++)
      ad = mulmod(ad, ad, n);
    if (r == s) return false;
  }
  return true;
}
```

2. Datastructures

2.1. Standard segment tree $\mathcal{O}(\log n)$.

```
typedef /* Tree element */ S;
const int n = 1 << 20; S t[2 * n];

// required axiom: associativity
S combine(S l, S r) { return l + r; } // sum segment tree
S combine(S l, S r) { return max(l, r); } // max segment tree

void build() { for (int i = n; --i; ) t[i] = combine(t[2 * i], t[2 * i + 1]); }

// set value v on position i
void update(int i, S v) { for (t[i += n] = v; i /= 2; ) t[i] = combine(t[2 * i], t[2 * i + 1]);}
}

// sum on interval [l, r)
S query(int l, int r) {
  S resL, resR;
  for (l += n, r += n; l < r; l /= 2, r /= 2) {
    if (l & 1) resL = combine(resL, t[l++]);
    if (r & 1) resR = combine(t[--r], resR);
  }
  return combine(resL, resR);
}
```

2.2. Binary Indexed Tree $\mathcal{O}(\log n)$. Use one-based indices $(i > 0)$!

```
int bit[MAXN + 1];

// arr[i] += v
void update(int i, int v) {
  while (i <= MAXN) bit[i] += v, i += i & -i;
}

// returns sum of arr[i], where i: [1, i]
int query(int i) {
  int v = 0; while (i) v += bit[i], i -= i & -i; return v;
}
```

2.3. Disjoint-Set / Union-Find $\mathcal{O}(\alpha(n))$.

```
int par[MAXN], rnk[MAXN];

void uf_init(int n) {
  fill_n(par, n, -1); fill_n(rnk, n, 0);
}

int uf_find(int v) { return par[v] < 0 ? v : par[v] = uf_find(par[v]); }
```

```
void uf_union(int a, int b) {
  if ((a = uf_find(a)) == (b = uf_find(b))) return;
  if (rnk[a] < rnk[b]) swap(a, b);
  if (rnk[a] == rnk[b]) rnk[a]++;
  par[b] = a;
}
```

3. Graph Algorithms

3.1. Maximum matching $\mathcal{O}(nm)$. This problem could be solved with a flow algorithm like Dinic's algorithm which runs in $\mathcal{O}(\sqrt{V}E)$, too.

```
const int sizeL = 1e4, sizeR = 1e4;

bool vis[sizeR];
int par[sizeR]; // par : R -> L
vi adj[sizeL]; // adj : L -> (N -> R)

bool match(int u) {
  for (int v : adj[u]) {
    if (vis[v]) continue; vis[v] = true;
    if (par[v] == -1 || match(par[v])) {
      par[v] = u;
      return true;
    }
  }
  return false;
}

// perfect matching iff ret == sizeL == sizeR
int maxmatch() {
  fill_n(par, sizeR, -1); int ret = 0;
  for (int i = 0; i < sizeL; i++) {
    fill_n(vis, sizeR, false);
    ret += match(i);
  }
  return ret;
}
```

3.2. Strongly Connected Components $\mathcal{O}(V+E)$.

```
vvi adj, comps; vi tidx, lnk, cnr, st; vector<bool> vis; int
  age, ncomps;

void tarjan(int v) {
  tidx[v] = lnk[v] = ++age; vis[v] = true; st.pb(v);

  for (int w : adj[v]) {
    if (!tidx[w]) tarjan(w), lnk[v] = min(lnk[v], lnk[w]);
    else if (vis[w]) lnk[v] = min(lnk[v], tidx[w]);
  }

  if (lnk[v] != tidx[v]) return;

  comps.pb(vi()); int w;
  do {
    vis[w = st.back()] = false; cnr[w] = ncomps;
      comps.back().pb(w);
    st.pop_back();
  } while (w != v);
  ncomps++;
}

void findSCC(int n) {
  age = ncomps = 0; vis.assign(n, false); tidx.assign(n, 0);
    lnk.resize(n);
  cnr.resize(n); comps.clear();

  for (int i = 0; i < n; i++)
    if (tidx[i] == 0) tarjan(i);
}
```

3.2.1. *2-SAT $\mathcal{O}(V+E)$.* Include findSCC.

```
void init2sat(int n) { adj.assign(2 * n, vi()); }

// vl, vr = true -> variable l, variable r should be negated.
void imply(int xl, bool vl, int xr, bool vr) {
  adj[2 * xl + vl].pb(2 * xr + vr); adj[2 * xr +!vr].pb(2 * xl
    +!vl); }

void satOr(int xl, bool vl, int xr, bool vr) { imply(xl, !vl,
    xr, vr); }
void satConst(int x, bool v) { imply(x, !v, x, v); }
void satIff(int xl, bool vl, int xr, bool vr) {
  imply(xl, vl, xr, vr); imply(xr, vr, xl, vl);}

bool solve2sat(int n, vector<bool> &sol) {
  findSCC(2 * n);
  for (int i = 0; i < n; i++)
    if (cnr[2 * i] == cnr[2 * i + 1]) return false;
  vector<bool> seen(n, false); sol.assign(n, false);
  for (vi &comp : comps) {
    for (int v : comp) {
      if (seen[v / 2]) continue;
      seen[v / 2] = true; sol[v / 2] = v & 1;
    }
  }
  return true;
}
```

3.3. Cycle Detection $\mathcal{O}(V+E)$.

```
vvi adj; // assumes bidirected graph, adjust accordingly

bool cycle_detection() {
  stack<int> s; vector<bool> vis(MAXN, false); vi par(MAXN,
    -1); s.push(0);
  vis[0] = true;
  while(!s.empty()) {
    int cur = s.top(); s.pop();
    for(int i : adj[cur]) {
      if(vis[i] && par[cur] != i) return true;
      s.push(i); par[i] = cur; vis[i] = true;
    }
  }
  return false;}
```

3.4. Shortest path.

3.4.1. *Dijkstra $\mathcal{O}(E+V\log V)$.*

3.4.2. *Floyd-Warshall $\mathcal{O}(V^3)$.*

```
int n = 100; ll d[MAXN][MAXN];
for (int i = 0; i < n; i++) fill_n(d[i], n, 1e18);
// set direct distances from i to j in d[i][j] (and d[j][i])
for (int i = 0; i < n; i++)
  for (int j = 0; j < n; j++)
    for (int k = 0; k < n; k++)
      d[j][k] = min(d[j][k], d[j][i] + d[i][k]);
```

3.4.3. *Bellman Ford $\mathcal{O}(VE)$.* This is only useful if there are edges with weight $w_{ij} < 0$ in the graph.

```
vector< pair<pii, ll> > edges; // ((from, to), weight)
vector<ll> dist;

// when undirected, add back edges
bool bellman_ford(int V, int source) {
  dist.assign(V, 1e18); dist[source] = 0;

  bool updated = true; int loops = 0;
  while (updated && loops < n) {
    updated = false;
    for (auto e : edges) {
      int alt = dist[e.x.x] + e.y;
      if (alt < dist[e.x.y]) {
        dist[e.x.y] = alt; updated = true;
      }
    }
  }
  return loops < n; // loops >= n: negative cycles
}
```

3.5. Max-flow min-cut.

3.5.1. *Dinic's Algorithm $\mathcal{O}(V^2E)$.*

```
struct edge {
  int to, rev; ll cap, flow;
  edge(int t, int r, ll c) : to(t), rev(r), cap(c), flow(0) {}
};

int s, t, level[MAXN]; // s = source, t = sink
vector<edge> g[MAXN];

void add_edge(int fr, int to, ll cap) {
  g[fr].pb(edge(to, g[to].size(), cap)); g[to].pb(edge(fr,
    g[fr].size() - 1, 0));
}

bool dinic_bfs() {
  fill_n(level, MAXN, 0); level[s] = 1;

  queue<int> q; q.push(s);
  while (!q.empty()) {
    int cur = q.front(); q.pop();
    for (edge e : g[cur]) {
      if (level[e.to] == 0 && e.flow < e.cap) {
        level[e.to] = level[cur] + 1; q.push(e.to);
      }
```

```
    }
  }
  return level[t] != 0;
}

ll dinic_dfs(int cur, ll maxf) {
  if (cur == t) return maxf;

  ll f = 0; bool isSat = true;
  for (edge &e : g[cur]) {
    if (level[e.to] != level[cur] + 1 || e.flow >= e.cap)
      continue;
    ll df = dinic_dfs(e.to, min(maxf - f, e.cap - e.flow));
    f += df; e.flow += df; g[e.to][e.rev].flow -= df; isSat &=
↪   e.flow == e.cap;
    if (maxf == f) break;
  }
  if (isSat) level[cur] = 0;
  return f;
}

ll dinic_maxflow() {
  ll f = 0;
  while (dinic_bfs()) f += dinic_dfs(s, LLINF);
  return f;
}
```

3.6. **Min-cost max-flow.** Find the cheapest possible way of sending a certain amount of flow through a flow network.

```
struct edge {
  // to, rev, flow, capacity, weight
  int t, r; ll f, c, w;
  edge(int _t, int _r, ll _c, ll _w) : t(_t), r(_r), f(0),
↪   c(_c), w(_w) {}
};

int n, par[MAXN]; vector<edge> adj[MAXN]; ll dist[MAXN];

bool findPath(int s, int t) {
  fill_n(dist, n, LLINF); fill_n(par, n, -1);

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

  while (!q.empty()) {
    int d = q.top().x, v = q.top().y; q.pop();
    if (d > dist[v]) continue;

    for (edge e : adj[v]) {
      if (e.f < e.c && d + e.w < dist[e.t]) {
        q.push(pii(dist[e.t] = d + e.w, e.t)); par[e.t] = e.r;
      }
    }
  }
  return dist[t] < INF;
}

pair<ll, ll> minCostMaxFlow(int s, int t) {
  ll cost = 0, flow = 0;
  while (findPath(s, t)) {
    ll f = INF, c = 0; int cur = t;
    while (cur != s) {
      const edge &rev = adj[cur][par[cur]], &e =
↪     adj[rev.t][rev.r];
      f = min(f, e.c - e.f); cur = rev.t;
    }
    cur = t;
    while (cur != s) {
      edge &rev = adj[cur][par[cur]], &e = adj[rev.t][rev.r];
      c += e.w; e.f += f; rev.f -= f; cur = rev.t;
    }
    cost += f * c; flow += f;
  }
  return pair<ll, ll>(cost, flow);
}

inline void addEdge(int from, int to, ll cap, ll weight) {
  adj[from].pb(edge(to, adj[to].size(), cap, weight));
  adj[to].pb(edge(from, adj[from].size() - 1, 0, -weight));
}
```

3.7. **Minimal Spanning Tree.**

3.7.1. *Kruskal* $\mathcal{O}(E \log V)$.

4. String algorithms

4.1. **Trie.**

```
const int SIGMA = 26;

struct trie {
  bool word; trie **adj;

  trie() : word(false), adj(new trie*[SIGMA]) {
    for (int i = 0; i < SIGMA; i++) adj[i] = NULL;
  }

  void addWord(const string &str) {
    trie *cur = this;
    for (char ch : str) {
      int i = ch - 'a';
      if (!cur->adj[i]) cur->adj[i] = new trie();
      cur = cur->adj[i];
    }
    cur->word = true;
  }

  bool isWord(const string &str) {
    trie *cur = this;
    for (char ch : str) {
      int i = ch - 'a';
      if (!cur->adj[i]) return false;
      cur = cur->adj[i];
    }
    return cur->word;
  }
};
```

4.2. **Z-algorithm** $\mathcal{O}(n)$.

```
// z[i] = length of longest substring starting from s[i] which
↪  is also a prefix of s.
vi z_function(const string &s) {
  int n = (int) s.length();
  vi z(n);
  for (int i = 1, l = 0, r = 0; i < n; ++i) {
    if (i <= r) z[i] = min (r - i + 1, z[i - l]);
    while (i + z[i] < n && s[z[i]] == s[i + z[i]]) ++z[i];
    if (i + z[i] - 1 > r) l = i, r = i + z[i] - 1;
  }
  return z;
}
```

4.3. **Suffix array** $\mathcal{O}(n \log^2 n)$. This creates an array $P[0], P[1], \ldots, P[n-1]$ such that the suffix $S[i \ldots n]$ is the $P[i]^{th}$ suffix of S when lexicographically sorted.

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

const int maxlogn = 17, int maxn = 1 << maxlogn;

tii make_triple(int a, int b, int c) { return tii(pii(a, b),
↪  c); }

int p[maxlogn + 1][maxn]; tii L[maxn];

int suffixArray(string S) {
  int N = S.size(), stp = 1, cnt = 1;
  for (int i = 0; i < N; i++) p[0][i] = S[i];
  for (; cnt < N; stp++, cnt <<= 1) {
    for (int i = 0; i < N; i++)
      L[i] = tii(pii(p[stp-1][i], i + cnt < N ? p[stp-1][i +
↪     cnt] : -1), i);
    sort(L, L + N);
    for (int i = 0; i < N; i++)
      p[stp][L[i].y] = i > 0 && L[i].x == L[i-1].x ?
↪     p[stp][L[i-1].y] : i;
  }
  return stp - 1; // result is in p[stp - 1][0 .. (N - 1)]
}
```

4.4. **Longest Common Subsequence** $\mathcal{O}(n^2)$. Substring: *consecutive characters* !!!

```
int dp[STR_SIZE][STR_SIZE]; // DP problem

int lcs(const string &w1, const string &w2) {
  int n1 = w1.size(), n2 = w2.size();
  for (int i = 0; i < n1; i++) {
    for (int j = 0; j < n2; j++) {
      if (i == 0 || j == 0) dp[i][j] = 0;
      else if (w1[i - 1] == w2[j - 1]) dp[i][j] = dp[i - 1][j
↪     - 1] + 1;
      else dp[i][j] = max(dp[i - 1][j], dp[i][j - 1]);
    }
  }
  return dp[n1][n2];
```

```
}

// backtrace
string getLCS(const string &w1, const string &w2) {
  int i = w1.size(), j = w2.size(); string ret = "";
  while (i > 0 && j > 0) {
    if (w1[i - 1] == w2[j - 1]) ret += w1[--i], j--;
    else if (dp[i][j - 1] > dp[i - 1][j]) j--;
    else i--;
  }
  reverse(ret.begin(), ret.end());
  return ret;
}
```

4.5. **Levenshtein Distance** $\mathcal{O}(n^2)$. Also known as the 'Edit distance'.

```
int dp[MAX_SIZE][MAX_SIZE]; // DP problem

int levDist(const string &w1, const string &w2) {
  int n1 = w1.size(), n2 = w2.size();
  for (int i = 0; i <= n1; i++) dp[i][0] = i; // removal
  for (int j = 0; j <= n2; j++) dp[0][j] = j; // insertion
  for (int i = 1; i <= n1; i++)
    for (int j = 1; j <= n2; j++)
      dp[i][j] = min(
        1 + min(dp[i - 1][j], dp[i][j - 1]),
        dp[i - 1][j - 1] + (w1[i - 1] != w2[j - 1])
      );
  return dp[n1][n2];
}
```

4.6. **Knuth-Morris-Pratt algorithm** $\mathcal{O}(N + M)$.

```
int kmp_search(const string &word, const string &text) {
  int n = word.size();
  vi T(n + 1, 0);
  for (int i = 1, j = 0; i < n; ) {
    if (word[i] == word[j]) T[++i] = ++j; // match
    else if (j > 0) j = T[j]; // fallback
    else i++; // no match, keep zero
  }
  int matches = 0;
  for (int i = 0, j = 0; i < text.size(); ) {
    if (text[i] == word[j]) {
      i++;
      if (++j == n) { // match at interval [i - n, i)
        matches++; j = T[j];
      }
    } else if (j > 0) j = T[j];
    else i++;
  }
  return matches;
}
```

4.7. **Aho-Corasick Algorithm** $\mathcal{O}(N + \sum_{i=1}^{m} |S_i|)$. All given P must be unique!

```
const int MAXP = 100, MAXLEN = 200, SIGMA = 26, MAXTRIE = MAXP * MAXLEN;

int nP;
string P[MAXP], S;

int pnr[MAXTRIE], to[MAXTRIE][SIGMA], sLink[MAXTRIE], dLink[MAXTRIE], nnodes;

void ahoCorasick() {
  fill_n(pnr, MAXTRIE, -1);
  for (int i = 0; i < MAXTRIE; i++) fill_n(to[i], SIGMA, 0);
  fill_n(sLink, MAXTRIE, 0); fill_n(dLink, MAXTRIE, 0);
  nnodes = 1;
  // STEP 1: MAKE A TREE
  for (int i = 0; i < nP; i++) {
    int cur = 0;
    for (char c : P[i]) {
      int i = c - 'a';
      if (to[cur][i] == 0) to[cur][i] = nnodes++;
      cur = to[cur][i];
    }
    pnr[cur] = i;
  }
  // STEP 2: CREATE SUFFIX_LINKS AND DICT_LINKS
  queue<int> q; q.push(0);
  while (!q.empty()) {
    int cur = q.front(); q.pop();
    for (int c = 0; c < SIGMA; c++) {
      if (to[cur][c]) {
        int sl = sLink[to[cur][c]] = cur == 0 ? 0 : to[sLink[cur]][c];
        // if all strings have equal length, remove this:
        dLink[to[cur][c]] = pnr[sl] >= 0 ? sl : dLink[sl];
        q.push(to[cur][c]);
      } else to[cur][c] = to[sLink[cur]][c];
    }
  }
  // STEP 3: TRAVERSE S
  for (int cur = 0, i = 0, n = S.size(); i < n; i++) {
    cur = to[cur][S[i] - 'a'];
    for (int hit = pnr[cur] >= 0 ? cur : dLink[cur]; hit; hit = dLink[hit]) {
      cerr << P[pnr[hit]] << " found at [" << (i + 1 - P[pnr[hit]].size()) << ", " << i << "]" << endl;
    }
  }
}
```

5. Geometry

```
const double EPS = 1e-7, PI = acos(-1.0);

typedef long long NUM; // EITHER double OR long long
typedef pair<NUM, NUM> pt;
#define x first
#define y second

pt operator+(pt p, pt q) { return pt(p.x + q.x, p.y + q.y); }
pt operator-(pt p, pt q) { return pt(p.x - q.x, p.y - q.y); }

pt& operator+=(pt &p, pt q) { return p = p + q; }
pt& operator-=(pt &p, pt q) { return p = p - q; }

pt operator*(pt p, NUM l) { return pt(p.x * l, p.y * l); }
pt operator/(pt p, NUM l) { return pt(p.x / l, p.y / l); }

NUM operator*(pt p, pt q) { return p.x * q.x + p.y * q.y; }
NUM operator^(pt p, pt q) { return p.x * q.y - p.y * q.x; }

istream& operator>>(istream &in, pt &p) { return in >> p.x >> p.y; }
ostream& operator<<(ostream &out, pt p) { return out << '(' << p.x << ", " << p.y << ')'; }

NUM lenSq(pt p) { return p * p; }
NUM lenSq(pt p, pt q) { return lenSq(p - q); }
double len(pt p) { return hypot(p.x, p.y); } // more overflow safe
double len(pt p, pt q) { return len(p - q); }

typedef pt frac;
typedef pair<double, double> vec;
vec getvec(pt p, pt dp, frac t) { return vec(p.x + 1. * dp.x * t.x / t.y, p.y + 1. * dp.y * t.x / t.y); }

// square distance from pt a to line bc
frac distPtLineSq(pt a, pt b, pt c) {
  a -= b, c -= b;
  return frac((a ^ c) * (a ^ c), c * c);
}

// square distance from pt a to linesegment bc
frac distPtSegmentSq(pt a, pt b, pt c) {
  a -= b; c -= b;
  NUM dot = a * c, len = c * c;
  if (dot <= 0) return frac(a * a, 1);
  if (dot >= len) return frac((a - c) * (a - c), 1);
  return frac(a * a * len - dot * dot, len);
}

// projects pt a onto linesegment bc
frac proj(pt a, pt b, pt c) { return frac((a - b) * (c - b), (c - b) * (c - b)); }
vec projv(pt a, pt b, pt c) { return getvec(b, c - b, proj(a, b, c)); }

bool collinear(pt a, pt b, pt c) { return ((a - b) ^ (a - c)) == 0; }

bool pointOnSegment(pt a, pt b, pt c) {
  NUM dot = (a - b) * (c - b), len = (c - b) * (c - b);
  return collinear(a, b, c) && 0 <= dot && dot <= len;
}

// true => 1 intersection, false => parallel, so 0 or \infty solutions
bool linesIntersect(pt a, pt b, pt c, pt d) { return ((a - b) ^ (c - d)) != 0; }
```

```
vec lineLineIntersection(pt a, pt b, pt c, pt d) {
  double det = (a - b) ^ (c - d); pt ret = (c - d) * (a ^ b) - (a - b) * (c ^ d);
  return vec(ret.x / det, ret.y / det);
}

// dp, dq are directions from p, q
// intersection at p + t_i dp, for 0 <= i < return value
int segmentIntersection(pt p, pt dp, pt q, pt dq, frac &t0, frac &t1){
  if (dp * dp == 0) swap(p, q), swap(dp, dq); // dq = 0
  if (dp * dp == 0) { t0 = t1 = frac(0, 1); return p == q; } // dp = dq = 0
  pt dpq = (q - p); NUM c = dp ^ dq, c0 = dpq ^ dp, c1 = dpq ^ dq;
  if (c == 0) { // parallel, dp > 0, dq >= 0
    if (c0 != 0) return 0; // not collinear
    NUM v0 = dpq * dp, v1 = v0 + dq * dp, dp2 = dp * dp;
    if (v1 < v0) swap(v0, v1);
    t0 = frac(v0 = max(v0, (NUM) 0), dp2);
    t1 = frac(v1 = min(v1, dp2), dp2);
    return (v0 <= v1) + (v0 < v1);
  } else if (c < 0) c = -c, c0 = -c0, c1 = -c1;
  t0 = t1 = frac(c1, c);
  return 0 <= min(c0, c1) && max(c0, c1) <= c;
}

// Returns TWICE the area of a polygon to keep it an integer
NUM polygonTwiceArea(const vector<pt> &pts) {
  NUM area = 0;
  for (int N = pts.size(), i = 0, j = N - 1; i < N; j = i++)
    area += pts[i] ^ pts[j];
  return abs(area); // area < 0 <=> pts ccw
}

bool pointInPolygon(pt p, const vector<pt> &pts) {
  double sum = 0;
  for (int N = pts.size(), i = 0, j = N - 1; i < N; j = i++) {
    if (pointOnSegment(p, pts[i], pts[j])) return true; // boundary
    double angle = acos((pts[i] - p) * (pts[j] - p) / len(pts[i], p) / len(pts[j], p));
    sum += ((pts[i] - p) ^ (pts[j] - p)) < 0 ? angle : -angle;}
  return abs(abs(sum) - 2 * PI) < EPS;
}
```

5.1. Convex Hull $\mathcal{O}(n\log n)$.

```
// points are given by: pts[ret[0]], pts[ret[1]], ... pts[ret[ret.size()-1]]
vi convexHull(const vector<pt> &pts) {
  if (pts.empty()) return vi();
  vi ret;
  // find one outer point:
  int fsti = 0, n = pts.size(); pt fstpt = pts[0];
  for(int i = n; i--; ) if (pts[i] < fstpt) fstpt = pts[fsti = i];
  ret.pb(fsti); pt refr = pts[fsti];
  vi ord; // index into pts
  for (int i = n; i--; ) if (pts[i] != refr) ord.pb(i);
  sort(ord.begin(), ord.end(), [&pts, &refr] (int a, int b) -> bool {
    NUM cross = (pts[a] - refr) ^ (pts[b] - refr);
    return cross != 0 ? cross > 0 : lenSq(refr, pts[a]) < lenSq(refr, pts[b]);
  });
  for (int i : ord) {
    // NOTE: > INCLUDES points on the hull-line, >= EXCLUDES
    while (ret.size() > 1 &&
        ((pts[ret[ret.size()-2]]-pts[ret.back()]) ^ (pts[i]-pts[ret.back()])) >= 0)
      ret.pop_back();
    ret.pb(i);
  }
  return ret;
}
```

5.2. Rotating Calipers $\mathcal{O}(n)$.

Finds the longest distance between two points in a convex hull.

```
NUM rotatingCalipers(vector<pt> &hull) {
  int n = hull.size(), a = 0, b = 1;
  if (n <= 1) return 0.0;
  while (((hull[1] - hull[0]) ^ (hull[(b + 1) % n] - hull[b])) > 0) b++;
  NUM ret = 0.0;
  while (a < n) {
    ret = max(ret, lenSq(hull[a], hull[b]));
    if (((hull[(a + 1) % n] - hull[a % n]) ^ (hull[(b + 1) % n] - hull[b])) <= 0) a++;
    else if (++b == n) b = 0;
  }
  return ret;
}
```

5.3. Closest points $\mathcal{O}(n\log n)$.

```
int n;pt pts[maxn];

struct byY {
  bool operator()(int a, int b) const { return pts[a].y < pts[b].y; }
};

inline NUM dist(pii p) {
  return hypot(pts[p.x].x - pts[p.y].x, pts[p.x].y - pts[p.y].y);
}

pii minpt(pii p1, pii p2) { return (dist(p1) < dist(p2)) ? p1 : p2;}

// closest pts (by index) inside pts[l ... r], with sorted y values in ys
pii closest(int l, int r, vi &ys) {
  if (r - l == 2) { // don't assume 1 here.
    ys = { l, l + 1 };
    return pii(l, l + 1);
  } else if (r - l == 3) { // brute-force
    ys = { l, l + 1, l + 2 };
    sort(ys.begin(), ys.end(), byY());
    return minpt(pii(l, l + 1), minpt(pii(l, l + 2), pii(l + 1, l + 2)));
  }
  int m = (l + r) / 2; vi yl, yr;
  pii delta = minpt(closest(l, m, yl), closest(m, r, yr));
  NUM ddelta = dist(delta), xm = .5 * (pts[m-1].x + pts[m].x);
  merge(yl.begin(), yl.end(), yr.begin(), yr.end(), back_inserter(ys), byY());
  deque<int> q;
  for (int i : ys) {
    if (abs(pts[i].x - xm) <= ddelta) {
      for (int j : q) delta = minpt(delta, pii(i, j));
      q.pb(i);
      if (q.size() > 8) q.pop_front(); // magic from Introduction to Algorithms.
    }
  }
  return delta;
}
```

6. Miscellaneous

6.1. Binary search $\mathcal{O}(\log(hi - lo))$.

```
bool test(int n);

int search(int lo, int hi) {
  // assert(test(lo) && !test(hi));
  while (hi - lo > 1) {
    int m = (lo + hi) / 2;
    (test(m) ? lo : hi) = m;
  }
  // assert(test(lo) && !test(hi));
  return lo;
}
```

6.2. Fast Fourier Transform $\mathcal{O}(n\log n)$.

Given two polynomials $A(x) = a_0 + a_1x + \cdots + a_{n/2}x^{n/2}$ and $B(x) = b_0 + b_1x + \cdots + b_{n/2}x^{n/2}$, FFT calculates all coefficients of $C(x) = A(x)\cdot B(x) = c_0 + c_1x + \ldots c_nx^n$, with $c_i = \sum_{j=0}^{i} a_jb_{i-j}$.

```
typedef complex<double> cpx;
const int logmaxn = 20, maxn = 1 << logmaxn;

cpx a[maxn] = {}, b[maxn] = {}, c[maxn];

void fft(cpx *src, cpx *dest) {
  for (int i = 0, rep = 0; i < maxn; i++, rep = 0) {
    for (int j = i, k = logmaxn; k--; j >>= 1) rep = (rep << 1) | (j & 1);
    dest[rep] = src[i];
  }
  for (int s = 1, m = 1; m <= maxn; s++, m *= 2) {
    cpx r = exp(cpx(0, 2.0 * PI / m));
    for (int k = 0; k < maxn; k += m) {
```

```
      cpx cr(1.0, 0.0);
      for (int j = 0; j < m / 2; j++) {
        cpx t = cr * dest[k + j + m / 2]; dest[k + j + m / 2] = dest[k + j] - t;
        dest[k + j] += t; cr *= r;
      }
    }
  }
}

void multiply() {
  fft(a, c); fft(b, a);
  for (int i = 0; i < maxn; i++) b[i] = conj(a[i] * c[i]);
  fft(b, c);
  for (int i = 0; i < maxn; i++) c[i] = conj(c[i]) / (1.0 * maxn);
}
```

6.3. Minimum Assignment (Hungarian Algorithm) $\mathcal{O}(n^3)$.

```
int a[MAXN + 1][MAXM + 1]; // matrix, 1-based

int minimum_assignment(int n, int m) { // n rows, m columns
  vi u(n + 1), v(m + 1), p(m + 1), way(m + 1);

  for (int i = 1; i <= n; i++) {
    p[0] = i;
    int j0 = 0;
    vi minv(m + 1, INF);
    vector<char> used(m + 1, false);
    do {
      used[j0] = true;
      int i0 = p[j0], delta = INF, j1;
      for (int j = 1; j <= m; j++)
        if (!used[j]) {
          int cur = a[i0][j] - u[i0] - v[j];
          if (cur < minv[j]) minv[j] = cur, way[j] = j0;
          if (minv[j] < delta) delta = minv[j], j1 = j;
        }
      for (int j = 0; j <= m; j++) {
        if(used[j]) u[p[j]] += delta, v[j] -= delta;
        else minv[j] -= delta;
      }
      j0 = j1;
    } while (p[j0] != 0);
    do {
      int j1 = way[j0]; p[j0] = p[j1]; j0 = j1;
    } while (j0);
  }

  // column j is assigned to row p[j]
  // for (int j = 1; j <= m; ++ j) ans[p[j]] = j;
  return -v[0];
}
```

6.4. Partial linear equation solver $\mathcal{O}(N^3)$.

```
typedef double NUM;

#define MAXN 110
#define EPS 1e-5

NUM mat[MAXN][MAXN + 1], vals[MAXN]; bool hasval[MAXN];

bool is_zero(NUM a) { return -EPS < a && a < EPS; }
bool eq(NUM a, NUM b) { return is_zero(a - b); }

int solvemat(int n){ //mat[i][j] contains the matrix A, mat[i][n] contains b
  int pivrow = 0, pivcol = 0;
  while (pivcol < n) {
    int r = pivrow, c;
    while (r < n && is_zero(mat[r][pivcol])) r++;
    if (r == n) { pivcol++; continue; }

    for (c = 0; c <= n; c++) swap(mat[pivrow][c], mat[r][c]);

    r = pivrow++; c = pivcol++;
    NUM div = mat[r][c];
    for (int col = c; col <= n; col++) mat[r][col] /= div;
    for (int row = 0; row < n; row++) {
      if (row == r) continue;
      NUM times = -mat[row][c];
      for (int col = c; col <= n; col++) mat[row][col] += times * mat[r][col];
    }
  } // now mat is in RREF

  for (int r = pivrow; r < n; r++)
    if (!is_zero(mat[r][n])) return 0;

  fill_n(hasval, n, false);
  for (int col = 0, row; col < n; col++) {
    hasval[col] = !is_zero(mat[row][col]);
    if (!hasval[col]) continue;
    for (int c = col + 1; c < n; c++) {
      if (!is_zero(mat[row][c])) hasval[col] = false;
    }
    if (hasval[col]) vals[col] = mat[row][n];
    row++;
  }

  for (int i = 0; i < n; i++)
    if (!hasval[i]) return 2;
  return 1;
}
```

7. Useful Information

8. Misc

8.1. Debugging Tips.

- Stack overflow? Recursive DFS on tree that is actually a long path?
- Floating-point numbers
  - Getting `NaN`? Make sure `acos` etc. are not getting values out of their range (perhaps `1+eps`).
  - Rounding negative numbers?
  - Outputting in scientific notation?
- Wrong Answer?
  - Read the problem statement again!
  - Are multiple test cases being handled correctly? Try repeating the same test case many times.
  - Integer overflow?
  - Think very carefully about boundaries of all input parameters
  - Try out possible edge cases:
    * $n = 0, n = -1, n = 1, n = 2^{31} - 1$ or $n = -2^{31}$
    * List is empty, or contains a single element
    * n is even, n is odd
    * Graph is empty, or contains a single vertex
    * Graph is a multigraph (loops or multiple edges)
    * Polygon is concave or non-simple
  - Is initial condition wrong for small cases?
  - Are you sure the algorithm is correct?
  - Explain your solution to someone.
  - Are you using any functions that you don't completely understand? Maybe STL functions?
  - Maybe you (or someone else) should rewrite the solution?
  - Can the input line be empty?
- Run-Time Error?
  - Is it actually Memory Limit Exceeded?

8.2. Solution Ideas.

- Dynamic Programming
  - Parsing CFGs: CYK Algorithm
  - Drop a parameter, recover from others
  - Swap answer and a parameter
  - When grouping: try splitting in two
  - 2^k trick
  - When optimizing
    * Convex hull optimization
      · $\mathrm{dp}[i] = \min_{j<i}\{\mathrm{dp}[j] + b[j] \times a[i]\}$
      · $b[j] \geq b[j+1]$
      · optionally $a[i] \leq a[i+1]$
      · $O(n^2)$ to $O(n)$
    * Divide and conquer optimization
      · $\mathrm{dp}[i][j] = \min_{k<j}\{\mathrm{dp}[i-1][k] + C[k][j]\}$
      · $A[i][j] \leq A[i][j+1]$
      · $O(kn^2)$ to $O(kn \log n)$
      · sufficient: $C[a][c] + C[b][d] \leq C[a][d] + C[b][c]$, $a \leq b \leq c \leq d$ (QI)
    * Knuth optimization
      · $\mathrm{dp}[i][j] = \min_{i<k<j}\{\mathrm{dp}[i][k] + \mathrm{dp}[k][j] + C[i][j]\}$
      · $A[i][j-1] \leq A[i][j] \leq A[i+1][j]$
      · $O(n^3)$ to $O(n^2)$
      · sufficient: QI and $C[b][c] \leq C[a][d]$, $a \leq b \leq c \leq d$
- Greedy
- Randomized
- Optimizations
  - Use bitset (/64)
  - Switch order of loops (cache locality)
- Process queries offline
  - Mo's algorithm
- Square-root decomposition
- Precomputation
- Efficient simulation
  - Mo's algorithm
  - Sqrt decomposition
  - Store 2^k jump pointers
- Data structure techniques
  - Sqrt buckets
  - Store 2^k jump pointers
  - 2^k merging trick
- Counting
  - Inclusion-exclusion principle
  - Generating functions
- Graphs
  - Can we model the problem as a graph?
  - Can we use any properties of the graph?
  - Strongly connected components
  - Cycles (or odd cycles)
  - Bipartite (no odd cycles)
    * Bipartite matching
    * Hall's marriage theorem
    * Stable Marriage
  - Cut vertex/bridge
  - Biconnected components
  - Degrees of vertices (odd/even)
  - Trees
    * Heavy-light decomposition
    * Centroid decomposition
    * Least common ancestor
    * Centers of the tree
  - Eulerian path/circuit
  - Chinese postman problem
  - Topological sort
  - (Min-Cost) Max Flow
  - Min Cut
    * Maximum Density Subgraph
  - Huffman Coding
  - Min-Cost Arborescence
  - Steiner Tree
  - Kirchoff's matrix tree theorem
  - Prüfer sequences
  - Lovász Toggle
  - Look at the DFS tree (which has no cross-edges)
  - Is the graph a DFA or NFA?
    * Is it the Synchronizing word problem?
- Mathematics
  - Is the function multiplicative?
  - Look for a pattern
  - Permutations
    * Consider the cycles of the permutation
  - Functions
    * Sum of piecewise-linear functions is a piecewise-linear function
    * Sum of convex (concave) functions is convex (concave)
  - Modular arithmetic
    * Chinese Remainder Theorem
    * Linear Congruence
  - Sieve
  - System of linear equations
  - Values too big to represent?
    * Compute using the logarithm
    * Divide everything by some large value
  - Linear programming
    * Is the dual problem easier to solve?
  - Can the problem be modeled as a different combinatorial problem? Does that simplify calculations?
- Logic
  - 2-SAT
  - XOR-SAT (Gauss elimination or Bipartite matching)
- Meet in the middle
- Only work with the smaller half ($\log(n)$)
- Strings
  - Trie (maybe over something weird, like bits)
  - Suffix array
  - Suffix automaton (+DP?)
  - Aho-Corasick
  - eerTree
  - Work with $S + S$
- Hashing
- Euler tour, tree to array
- Segment trees
  - Lazy propagation
  - Persistent
  - Implicit
  - Segment tree of X
- Geometry
  - Minkowski sum (of convex sets)
  - Rotating calipers
  - Sweep line (horizontally or vertically?)
  - Sweep angle
  - Convex hull
- Fix a parameter (possibly the answer).
- Are there few distinct values?
- Binary search
- Sliding Window (+ Monotonic Queue)
- Computing a Convolution? Fast Fourier Transform
- Computing a 2D Convolution? FFT on each row, and then on each column
- Exact Cover (+ Algorithm X)
- Cycle-Finding
- What is the smallest set of values that identify the solution? The cycle structure of the permutation? The powers of primes in the factorization?
- Look at the complement problem
  - Minimize something instead of maximizing
- Immediately enforce necessary conditions. (All values greater than 0? Initialize them all to 1)
- Add large constant to negative numbers to make them positive
- Counting/Bucket sort

9. Formulas

- **Legendre symbol:** $\left(\frac{a}{b}\right) = a^{(b-1)/2} \pmod b$, b odd prime.
- **Heron's formula:** A triangle with side lengths a, b, c has area $\sqrt{s(s-a)(s-b)(s-c)}$ where $s = \frac{a+b+c}{2}$.
- **Pick's theorem:** A polygon on an integer grid strictly containing i lattice points and having b lattice points on the boundary has area $i + \frac{b}{2} - 1$. (Nothing similar in higher dimensions)
- **Euler's totient:** The number of integers less than n that are coprime to n are $n \prod_{p|n} \left(1 - \frac{1}{p}\right)$ where each p is a distinct prime factor of n.
- **König's theorem:** In any bipartite graph $G = (L \cup R, E)$, the number of edges in a maximum matching is equal to the number of vertices in a minimum vertex cover. Let U be the set of unmatched vertices in L, and Z be the set of vertices that are either in U or are connected to U by an alternating path. Then $K = (L \setminus Z) \cup (R \cap Z)$ is the minimum vertex cover.
- A minumum Steiner tree for n vertices requires at most $n-2$ additional Steiner vertices.
- The number of vertices of a graph is equal to its minimum vertex cover number plus the size of a maximum independent set.
- **Lagrange polynomial** through points $(x_0, y_0), \dots, (x_k, y_k)$ is $L(x) = \sum_{j=0}^{k} y_j \prod_{\substack{0 \le m \le k \\ m \ne j}} \frac{x - x_m}{x_j - x_m}$
- **Hook length formula:** If λ is a Young diagram and $h_\lambda(i,j)$ is the hook-length of cell (i,j), then then the number of Young tableux $d_\lambda = n!/\prod h_\lambda(i,j)$.
- **Möbius inversion formula:** If $f(n) = \sum_{d|n} g(d)$, then $g(n) = \sum_{d|n} \mu(d) f(n/d)$. If $f(n) = \sum_{m=1}^{n} g(\lfloor n/m \rfloor)$, then $g(n) = \sum_{m=1}^{n} \mu(m) f(\lfloor \frac{n}{m} \rfloor)$.
- #primitive pythagorean triples with hypotenuse $< n$ approx $n/(2\pi)$.
- **Frobenius Number:** largest number which can't be expressed as a linear combination of numbers $a_1, \dots, a_n$ with non-negative coefficients. $g(a_1, a_2) = a_1 a_2 - a_1 - a_2$, $N(a_1, a_2) = (a_1 - 1)(a_2 - 1)/2$. $g(d \cdot a_1, d \cdot a_2, a_3) = d \cdot g(a_1, a_2, a_3) + a_3(d-1)$. An integer $x > (\max_i a_i)^2$ can be expressed in such a way iff. $x \mid \gcd(a_1, \dots, a_n)$.

9.1. Physics.

- **Snell's law:** $\frac{\sin\theta_1}{v_1} = \frac{\sin\theta_2}{v_2}$

9.2. Markov Chains.

A Markov Chain can be represented as a weighted directed graph of states, where the weight of an edge represents the probability of transitioning over that edge in one timestep. Let $P^{(m)} = (p_{ij}^{(m)})$ be the probability matrix of transitioning from state i to state j in m timesteps, and note that $P^{(1)}$ is the adjacency matrix of the graph. **Chapman-Kolmogorov:** $p_{ij}^{(m+n)} = \sum_k p_{ik}^{(m)} p_{kj}^{(n)}$. It follows that $P^{(m+n)} = P^{(m)} P^{(n)}$ and $P^{(m)} = P^m$. If $p^{(0)}$ is the initial probability distribution (a vector), then $p^{(0)} P^{(m)}$ is the probability distribution after m timesteps.

The return times of a state i is $R_i = \{m \mid p_{ii}^{(m)} > 0\}$, and i is *aperiodic* if $\gcd(R_i) = 1$. A MC is aperiodic if any of its vertices is aperiodic. A MC is *irreducible* if the corresponding graph is strongly connected.

A distribution π is stationary if $\pi P = \pi$. If MC is irreducible then $\pi_i = 1/\mathbb{E}[T_i]$, where T_i is the expected time between two visits at i. π_j/π_i is the expected number of visits at j in between two consecutive visits at i. A MC is *ergodic* if $\lim_{m\to\infty} p^{(0)} P^m = \pi$. A MC is ergodic iff. it is irreducible and aperiodic.

A MC for a random walk in an undirected weighted graph (unweighted graph can be made weighted by adding 1-weights) has $p_{uv} = w_{uv}/\sum_x w_{ux}$. If the graph is connected, then $\pi_u = \sum_x w_{ux}/\sum_v \sum_x w_{vx}$. Such a random walk is aperiodic iff. the graph is not bipartite.

An *absorbing* MC is of the form $P = \begin{pmatrix} Q & R \\ 0 & I_r \end{pmatrix}$. Let $N = \sum_{m=0}^{\infty} Q^m = (I_t - Q)^{-1}$. Then, if starting in state i, the expected number of steps till absorption is the i-th entry in $N1$. If starting in state i, the probability of being absorbed in state j is the (i,j)-th entry of NR.

Many problems on MC can be formulated in terms of a system of recurrence relations, and then solved using Gaussian elimination.

9.3. Burnside's Lemma.

Let G be a finite group that acts on a set X. For each g in G let X^g denote the set of elements in X that are fixed by g. Then the number of orbits

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

$$Z(S_n) = \frac{1}{n} \sum_{l=1}^{n} a_l Z(S_{n-l})$$

9.4. Bézout's identity.

If (x, y) is any solution to $ax + by = d$ (e.g. found by the Extended Euclidean Algorithm), then all solutions are given by

$$\left(x + k\frac{b}{\gcd(a,b)}, y - k\frac{a}{\gcd(a,b)}\right)$$

9.5. Misc.

9.5.1. *Determinants and PM.*

$$det(A) = \sum_{\sigma \in S_n} \operatorname{sgn}(\sigma) \prod_{i=1}^{n} a_{i,\sigma(i)}$$

$$perm(A) = \sum_{\sigma \in S_n} \prod_{i=1}^{n} a_{i,\sigma(i)}$$

$$pf(A) = \frac{1}{2^n n!} \sum_{\sigma \in S_{2n}} \operatorname{sgn}(\sigma) \prod_{i=1}^{n} a_{\sigma(2i-1),\sigma(2i)}$$

$$= \sum_{M \in \mathrm{PM}(n)} \operatorname{sgn}(M) \prod_{(i,j) \in M} a_{i,j}$$

9.5.2. *BEST Theorem.*

Count directed Eulerian cycles. Number of OST given by Kirchoff's Theorem (remove r/c with root) $\#\mathrm{OST}(G,r) \cdot \prod_v (d_v - 1)!$

9.5.3. *Primitive Roots.*

Only exists when n is $2, 4, p^k, 2p^k$, where p odd prime. Assume n prime. Number of primitive roots $\phi(\phi(n))$ Let g be primitive root. All primitive roots are of the form g^k where $k, \phi(p)$ are coprime.
k-roots: $g^{i \cdot \phi(n)/k}$ for $0 \le i < k$

9.5.4. *Sum of primes.*

For any multiplicative f:

$$S(n,p) = S(n,p-1) - f(p) \cdot (S(n/p, p-1) - S(p-1, p-1))$$

9.5.5. *Floor.*

$$\lfloor \lfloor x/y \rfloor / z \rfloor = \lfloor x/(yz) \rfloor$$
$$x\%y = x - y \lfloor x/y \rfloor$$

Practice Contest Checklist

- How many operations per second? Compare to local machine.
- What is the stack size?
- How to use printf/scanf with long long/long double?
- Are `__int128` and `__float128` available?
- Does MLE give RTE or MLE as a verdict? What about stack overflow?
- What is `RAND_MAX`?
- How does the judge handle extra spaces (or missing newlines) in the output?
- Look at documentation for programming languages.
- Try different programming languages: C++, Java and Python.
- Try the submit script.
- Try local programs: i?python[23], factor.
- Try submitting with `assert(false)` and `assert(true)`.
- Return-value from `main`.
- Look for directory with sample test cases.
- Make sure printing works.
- Remove this page from the notebook.