# Preamble

```cpp
#include <bits/stdc++.h>

using namespace std;

#define int long long
#define REP(i, a, b) for (int i = a; i < (b); ++i)

signed main() {
  cin.tie(NULL)->sync_with_stdio(false);
}
```

## Debug Memory Usage

```cpp
long long get_memory_usage() {
  struct rusage usage;
  getrusage(RUSAGE_SELF, &usage);
  return usage.ru_maxrss; // Maximum resident set size (in kilobytes on Linux, bytes on macOS)
}
```

## Output

```cpp
// Fixed precision.
cout << fixed << setprecision(6) << lf << '\n';
// Binary output
cout << format("{:06b}", b) << "fixed length binary";
cout << format("{:b}", b) << "variable length binary";
```

# Linear Algebra

## Gauss-Jordan

### Partial Pivot RREF - Rectangular

```cpp
const double EPSILON = 1e-10;
typedef double T;
typedef vector<T> VT;
typedef vector<VT> VVT;
tuple<int,double> rref(VVT &a) {
  int n = a.size();
  int m = a[0].size();
  int r = 0;
  double det = 1.;
  for (int c = 0; c < m && r < n; c++) {
    int j = r;
    for (int i = r + 1; i < n; i++)
      if (fabs(a[i][c]) > fabs(a[j][c])) j = i;
    if (fabs(a[j][c]) < EPSILON) continue;
    swap(a[j], a[r]);
    if (j != r) det *= -1.;
    det *= a[r][c];
    T s = 1.0 / a[r][c];
    for (int j = 0; j < m; j++) a[r][j] *= s;
    for (int i = 0; i < n; i++) if (i != r) {
      T t = a[i][c];
      for (int j = 0; j < m; j++) a[i][j] -= t * a[r][j];
    }
    r++;
  }
  return {r,det};
}
```

**Full Pivot - Inverse, Square, Solving** $(n \times n) \cdot (n \times m) = (n. \times m)$

- Solving systems of linear equations ($AX = B$)
- Inverting matrices ($AX = I$)
- Computing determinants of square matrices

Runs in $\mathcal{O}(n^3)$

Output:

- $X$ stored in b
- $A^{-1}$ stored in a

```cpp
const double EPS = 1e-10;
typedef vector<int> VI;
typedef double T;
typedef vector<T> VT;
typedef vector<VT> VVT;
T GaussJordan(VVT &a, VVT &b) {
  const int n = a.size();
  const int m = b[0].size();
  VI irow(n), icol(n), ipiv(n);
  T det = 1;
  for (int i = 0; i < n; i++) {
    int pj = -1, pk = -1;
    for (int j = 0; j < n; j++) if (!ipiv[j])
      for (int k = 0; k < n; k++) if (!ipiv[k])
  if (pj == -1 || fabs(a[j][k]) > fabs(a[pj][pk])) { pj = j; pk = k; }
    if (fabs(a[pj][pk]) < EPS) { cerr << "Matrix is singular." << endl; exit(0); }
    ipiv[pk]++;
    swap(a[pj], a[pk]);
    swap(b[pj], b[pk]);
    if (pj != pk) det *= -1;
    irow[i] = pj;
    icol[i] = pk;
    T c = 1.0 / a[pk][pk];
    det *= a[pk][pk];
    a[pk][pk] = 1.0;
    for (int p = 0; p < n; p++) a[pk][p] *= c;
    for (int p = 0; p < m; p++) b[pk][p] *= c;
    for (int p = 0; p < n; p++) if (p != pk) {
      c = a[p][pk];
      a[p][pk] = 0;
      for (int q = 0; q < n; q++) a[p][q] -= a[pk][q] * c;
      for (int q = 0; q < m; q++) b[p][q] -= b[pk][q] * c;
    }
  }
  for (int p = n-1; p >= 0; p--) if (irow[p] != icol[p]) {
    for (int k = 0; k < n; k++) swap(a[k][irow[p]], a[k][icol[p]]);
  }
  return det;
}
```

## XOR Basis

Small vectors

```cpp
vector<int> basis;
void add(int x) {
  for (int i = 0; i < basis.size(); i++) {  // reduce x using the current basis vectors
    x = min(x, x ^ basis[i]);
  }
  if (x != 0) { basis.push_back(x); }
}
```

Arbitrarily large vectors

```cpp
bool non_zero(const vector<uint64_t>& x) {
  bool non_zero = false;
  for(const auto& a : x) {
    non_zero |= (a != (uint64_t) 0);
  }
  return non_zero;
}
struct Basis {
  vector<vector<uint64_t>> basis;
  vector<uint64_t> reduce(vector<uint64_t> x) {
    for(int i = 0; i < basis.size(); i++) {
      int state = 0;
      for(int j = 0; j < x.size(); j++) {
        int cur = basis[i][j] ^ x[j];
        if (state == 0 and cur < x[j]) state = -1;
        if (state == 0 and cur > x[j]) state = 1;
        if (state <= 0) x[j] = cur;
      }
    }
    return x;
  }
  void add(vector<uint64_t> x) {
    x = reduce(x);
    if (non_zero(x)) basis.push_back(x);
```

```cpp
  }
  bool equal(const Basis& other) {
    if (other.basis.size() != basis.size()) return false;
    bool ans = true;
    for(const auto & v : other.basis) {
      ans &= !non_zero(reduce(v));
    }
    return ans;
  }
};
```

# Number Theory

## Extended Euclidean Algorithm

Finds $x$ and $y$ for which $ax + by = \gcd(a, b)$.
**Time:** $\mathcal{O}(\log n)$

```cpp
// Returns {x,y,gcd} where xa + yb = gcd
array<int,3> gcd_ext(int a,int b) {
  auto oa=a,ob=b;
  int x=0,y=1,u=1,v=0;
  while(a!=0) {
    auto q=b/a,r=b%a;
    auto m=x-u*q,n=y-v*q;
    b=a, a=r, x=u,y=v,u=m,v=n;
  }
  assert(oa*x+ob*y==b);
  return {x,y,b};
}
```

## Modular Inverse

Finds $x$ such that $ax = 1 \bmod m$.
**Time:** $\mathcal{O}(\log n)$

```cpp
int inv(int a, int m) {
  auto [x,y,g] = gcd_ext(a, m);
  if (g != 1) {
    // No solution!!!
    return -1;
  }
  else {
     // Inverse
     return (x % m + m) % m;
  }
}
```

TODO: All modular inverses in $\mathcal{O}(m)$: https://cp-algorithms.com/algebra/module-inverse.html

## Linear Congruence Equation

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

```cpp
// Returns {solution, modulo}
pair<int,int> linear_congruence(int a, int b, int n) {
  int d;
  if ((d = gcd(a,n)) != 1) {
    // No solution
    if (b % d != 0) return {-1, -1};
    a /= d; b /= d; n /= d;
  }
  int i = inv(a, n);
  return {(b * i) % n, n};
}
```

## Linear Prime Sieve

This calculates the minimum prime factor `pr[j]` for all all $j$ up to $n$. From this, we can calculate the prime factorisation of all these numbers.
**Time:** $\mathcal{O}(n)$

```cpp
const int N = 10000000;
vector<int> lp(N+1);
vector<int> pr;
for (int i=2; i <= N; ++i) {
  if (lp[i] == 0) { lp[i] = i; pr.push_back(i); }
  for (int j = 0; i * pr[j] <= N; ++j) {
    lp[i * pr[j]] = pr[j];
    if (pr[j] == lp[i]) break;
  }
}
```

## Extended Chinese Remainder Theorem

Works for non-coprime moduli

```cpp
struct ChineseRemainder {
  int a=0,m=0;
  void add(int b, int n) {
    b=(b%n+n)%n;
    if(m==-1) return;
    if(m==0) { a=b; m=n; return; }
    auto [u,v,g] = gcd_ext(m,n);
    if((a-b)%g!=0) { m=-1;return; }
    int lam = (a-b)/g;
    m=m/g*n;
    a = b + (lam*v)%m*n;
    a = (a%m+m)%m;
  }
  int get(int x) {return a+m*x;}
};
```

# Fast Fourier Transform

Useful for multiplying polynomials, or computing convolutions. $c[k] = \sum_i a[i]b[k-i]$. For sliding element-wise multiplication, reverse one of the arrays. Rounding is safe if $(\sum a_i^2 + \sum b_i^2)\log_2 N < 9 \cdot 10^{14}$. ($N = |A| + |B|$. In practice, with random inputs, bound is $10^{16}$).
**Time:** $\mathcal{O}(N \log N)$

```cpp

#define SZ(x) (int)(x).size()
#define ALL(x) begin(x), end(x)

typedef vector<int> vi;
typedef complex<double> C;
typedef vector<double> vd;
void fft(vector<C> &a) {
  int n = SZ(a), L = 31 - __builtin_clz(n);
  static vector<complex<long double>> R(2, 1);
  static vector<C> rt(2, 1);  // (^ 10% faster i f double )
  for (static int k = 2; k < n; k *= 2) {
    R.resize(n);
    rt.resize(n);
    auto x = polar(1.0L, acos(-1.0L) / k);
    REP(i, k, 2 * k) rt[i] = R[i] = i & 1 ? R[i / 2] * x : R[i /
    2];
  }
  vi rev(n);
  REP(i, 0, n) rev[i] = (rev[i / 2] | (i & 1) << L) / 2;
  REP(i, 0, n) if (i < rev[i]) swap(a[i], a[rev[i]]);
  for (int k = 1; k < n; k *= 2)
   for (int i = 0; i < n; i += 2 * k) REP(j, 0, k) {
     C z = rt[j + k] *
       a[i + j + k];  // (25% faster i f hand−r o l l e d )
     a[i + j + k] = a[i + j] - z;
     a[i + j] += z;
   }
}
vd conv(const vd &a, const vd &b) {
  if (a.empty() || b.empty()) return {};
  vd res(SZ(a) + SZ(b) - 1);
  int L = 32 - __builtin_clz(SZ(res)), n = 1 << L;
  vector<C> in(n), out(n);
  copy(ALL(a), begin(in));
  REP(i, 0, SZ(b)) in[i].imag(b[i]);
  fft(in);
  for (C &x : in) x *= x;
  REP(i, 0, n) out[i] = in[-i & (n - 1)] - conj(in[i]);
  fft(out);
  REP(i, 0, SZ(res)) res[i] = imag(out[i]) / (4 * n);
  return res;
}
```

# Geometry

## Preamble

This gives us vector addition, scalar and complex multiplication, angle `arg()`, and polar form initialisation `cis()`.

```cpp
typedef complex<double> C;
```

## Dot Product

```cpp
double dotp(C a , C b){return (conj(a)*b).real();}
double dist2(C a, C b){return dotp(a-b, a-b);}
```

$a_0 b_0 + a_1 b_1 = |a||b| \cos(\theta)$

## Cross Product

```cpp
double crossp(C a , C b){return (conj(a)*b).imag();}
double orient(C a, C b, C c){return crossp(b-c,b-a);}
```

$a_0 b_1 - a_1 b_0 = |a||b| \cos(\theta)$

## Ordering By Orientation

```cpp
bool topHalf(C a) {
  return (a.imag() > 0) || (a.imag() == 0 && a.real() >= 0);
}
bool cmp(const C &a, const C &b) {
  bool ha = topHalf(a);
  bool hb = topHalf(b);
  if (ha != hb) return ha;
  return orient(a, {0,0}, b) > 0;
}
```

# String Matching

## Z-Algorithm

```cpp
vector<int> z_algo(const string& s) {
  int n = s.size();
  vector<int> z(n);
  int l = 0, r = 0;
  for(int i = 1; i < n; i++) {
    if(i < r) z[i] = min(r - i, z[i - l]);
    while(i + z[i] < n && s[z[i]] == s[i + z[i]]) z[i]++;
    if(i + z[i] > r) { l = i; r = i + z[i]; }
  }
  return z;
}
```

## Aho-Curasick

Creates a string automaton for matching a dictionary of patterns. We hit a success state for each match of a pattern. Linear time on the total length of all patterns.

```cpp
struct Node {
  int par;
  char c;
  map<char, int> next;
  int link = -1;
  bool terminal = false;
  Node(int par, char c) : par(par), c(c) {}
};
vector<Node> nodes;
int new_node(int par, char c) {
  Node node = Node(par, c);
  nodes.push_back(node);
  return nodes.size() - 1;
}
int aho_curasick(const vector<string>& words) {
  // Root
  new_node(-1, '!');
  // Trie construction
  for (const auto& word : words) {
    int cur = 0;
    REP(i, 0, word.size()) {
      char c = word[i];
      if (nodes[cur].next.find(c) == nodes[cur].next.end()) {
        int nw = new_node(cur, c);
        nodes[cur].next[c] = nw;
      }
      cur = nodes[cur].next[c];
    }
    nodes[cur].terminal = true;
  }
  // Initialize root.
  deque<int> q;
  nodes[0].link = 0;
  for (char c = 'a'; c <= 'z'; c++) {
    if (nodes[0].next.find(c) == nodes[0].next.end()) {
      nodes[0].next[c] = 0;
    } else {
      q.push_back(nodes[0].next[c]);
    }
  }
  // BFS - initialise suffix links and failiure states.
  while (!q.empty()) {
    int i = q.front();
    q.pop_front();
    if (nodes[i].par == 0) {
      nodes[i].link = 0;
    } else {
      nodes[i].link = nodes[nodes[nodes[i].par].link].next[nodes[i].c];
    }
    for (char c = 'a'; c <= 'z'; c++) {
      if (nodes[i].next.find(c) == nodes[i].next.end()) {
        nodes[i].next[c] = nodes[nodes[i].link].next[c];
      } else {
        q.push_back(nodes[i].next[c]);
      }
    }
  }
  return 0;
}
```

## Ukkonen's

Linear time suffix tree construction. Useful for string matching.

```cpp
const int MAXN = 8000005;
string s;
int n;
struct Node {
  int l, r, par, link;
  vector<pair<char, int>> next;
  Node(int l = 0, int r = 0, int par = -1) : l(l), r(r), par(par), link(-1) {}
  int len() { return r - l; }
  // More space efficient than map, can use alternatively.
  int& get(char c) {
    for (auto& [a, b] : next)
      if (a == c) return b;
    next.push_back({c, -1});
    return next.back().second;
  }
};
Node t[MAXN];
int sz;
struct State {
  int v, pos;
  State(int v, int pos) : v(v), pos(pos) {}
};
State ptr(0, 0);
State go(State st, int l, int r) {
  while (l < r)
    if (st.pos == t[st.v].len()) {
      st = State(t[st.v].get(s[l]), 0);
      if (st.v == -1) return st;
    } else {
      if (s[t[st.v].l + st.pos] != s[l]) return State(-1, -1);
      if (r - l < t[st.v].len() - st.pos)
        return State(st.v, st.pos + r - l);
      l += t[st.v].len() - st.pos;
      st.pos = t[st.v].len();
    }
```

```
  return st;
}
int split(State st) {
  if (st.pos == t[st.v].len()) return st.v;
  if (st.pos == 0) return t[st.v].par;
  Node v = t[st.v];
  int id = sz++;
  t[id] = Node(v.l, v.l + st.pos, v.par);
  t[v.par].get(s[v.l]) = id;
  t[id].get(s[v.l + st.pos]) = st.v;
  t[st.v].par = id;
  t[st.v].l += st.pos;
  return id;
}
int get_link(int v) {
  if (t[v].link != -1) return t[v].link;
  if (t[v].par == -1) return 0;
  int to = get_link(t[v].par);
  return t[v].link = split(go(State(to, t[to].len()), t[v].l + (t[v].par == 0), t[v].r));
}
void tree_extend(int pos) {
  for (;;) {
    State nptr = go(ptr, pos, pos + 1);
    if (nptr.v != -1) {
      ptr = nptr;
      return;
    }
    int mid = split(ptr);
    int leaf = sz++;
    t[leaf] = Node(pos, n, mid);
    t[mid].get(s[pos]) = leaf;

    ptr.v = get_link(mid);
    ptr.pos = t[ptr.v].len();
    if (!mid) break;
  }
}

void build_tree() {
  sz = 1;
  for (int i = 0; i < n; ++i) tree_extend(i);
}
```

# Segment Trees!!!

## Basic

```cpp
struct BasicSegmentTree {
  using Value = int;
  Value identity = INT_MAX;
  Value binop(Value a, Value b) {return min(a, b);}
  vector<Value> arr;
  int size;
  BasicSegmentTree(int n) : arr(4*n + 2,identity), size(n) {};
  void update(int cur, int i, Value v, int l, int r) {
    if (l == r) {arr[cur] = v; return; }
    int mid = midpoint(l, r);
    if (i <= mid) update(2*cur, i, v, l, mid);
    else update(2*cur + 1, i, v, mid + 1, r);
    arr[cur] = binop(arr[2*cur],arr[2*cur + 1]);
  }
  void update(int i, int v) {update(1,i,v,0,size - 1);}
  Value query(int cur, int ql, int qr, int l, int r) {
    if (l == ql and r == qr) return arr[cur];
    int mid = midpoint(l,r);
    Value val = identity;
    if (ql <= mid) val = binop(val, query(2*cur,ql,min(mid,qr),l,mid));
    if (qr > mid) val = binop(val,query(2*cur + 1,max(mid + 1,ql),qr,mid+1,r));
    return val;
  }
  Value query(int ql, int qr) {return query(1,ql,qr,0,size - 1);}
};
```

## Lazy Update

```cpp
struct LazyUpdateTree {
  using Value = int;
  using Update = int;
  Value identity = LLONG_MIN;
  Value def = 0;
  Update idUpdate = 0;
  Value binop(Value a, Value b) {return max(a, b);}
  Value applyUpdate(Update a, Value u, int l, int r) {return u + a;}
  Update mergeUpdate(Update old, Update nw) {return old + nw;}
  vector<Value> arr;
  vector<Update> lazy;
  int size;
  LazyUpdateTree(int n) : arr(4*n + 2,def), lazy(4*n + 2, idUpdate), size(n) {};
  void push(int cur,int l, int r) {
    if (l != r) {
      int mid = midpoint(l,r);
      lazy[cur*2] = mergeUpdate(lazy[cur * 2], lazy[cur]);
      arr[cur * 2] = applyUpdate(lazy[cur],arr[cur*2],l,mid);
      lazy[cur*2 + 1] = mergeUpdate(lazy[cur * 2 + 1], lazy[cur]);
      arr[cur * 2 + 1] = applyUpdate(lazy[cur],arr[cur*2 + 1],mid + 1,r);
    }
    lazy[cur] = idUpdate;
  }
  void update(int cur, int ql,int qr, Update u, int l, int r) {
    if (l == ql and r == qr) {
      lazy[cur] = mergeUpdate(lazy[cur],u);
      arr[cur] = applyUpdate(u,arr[cur],l,r);
      return;
    }
    push(cur, l, r);
    int mid = midpoint(l, r);
    if (ql <= mid) update(2*cur,ql,min(mid,qr),u,l,mid);
    if (qr > mid) update(2*cur + 1,max(mid + 1,ql),qr,u,mid+1,r);
    arr[cur] = binop(arr[2*cur],arr[2*cur + 1]);
  }
  void update(int ql,int qr, Update u) {update(1,ql,qr,u,0,size-1);}
  Value query(int cur, int ql, int qr, int l, int r) {
    if (l == ql and r == qr) return arr[cur];
    push(cur,l,r);
    int mid = midpoint(l,r);
    Value val = identity;
    if (ql <= mid) val = binop(val, query(2*cur,ql,min(mid,qr),l,mid));
    if (qr > mid) val = binop(val,query(2*cur + 1,max(mid + 1,ql),qr,mid+1,r));
    return val;
  }
  Value query(int ql, int qr) {return query(1,ql,qr,0,size - 1);}
};
```

# DP Optimisations

## Convex Hull Trick

From a set of linear functions, finds the minimum value at a point.

- Adding Equation - Amortized $\mathcal{O}(1)$
- Finding minimum - $\mathcal{O}(\log n)$

Requires gradients to be increasing when inserted. Can use Li-Chao tree for online.

To find maximum, flip equations on insert, and flip answer.

```cpp
// Can use double
typedef int F;
typedef complex<F> P;
F dot(P a, P b) {
    return (conj(a) * b).real();
```

```
}
F cross(P a, P b) {
    return (conj(a) * b).imag();
}
struct Cht {
    vector<P> hull, vecs;
    // y= k x + b
    void add_line(F k, F b) {
        P nw = {k, b};
        while(!vecs.empty() && dot(vecs.back(), nw -
        hull.back()) < 0) {
            hull.pop_back();
            vecs.pop_back();
        }
        if(!hull.empty()) {
            vecs.push_back(P(0,1) * (nw - hull.back()));
        }
        hull.push_back(nw);
    }
    F get(F x) {
        P query = {x, 1};
        auto it = lower_bound(vecs.begin(), vecs.end(), query,
        [](F a, F b) {
            return cross(a, b) > 0;
        });
        return dot(query, hull[it - vecs.begin()]);
    }
};
```