Team Reference Document

Team Dolores TUMbridge – TU München

1 General 1
1.1 Compilation 1
1.2 Debugging 1
1.3 Random 2
1.4 Language Specific Functionalities 2

2 Dynamic Programming 2
2.1 Longest Increasing Subsequence 2
2.2 Divide and Conquer Optimization 2
2.3 Knuths Optimization 2
2.4 Alien Trick 2
2.5 Knapsack 3

3 Graphs 3
3.1 Theorems 3
3.1.1 Euler's theorem 3
3.1.2 Vertex covers and independent sets 3
3.1.3 Erdős Gallai: Degree Sequence 3
3.1.4 Cayley's formula (Prüfer sequence) 3
3.2 Traversal 3
3.2.1 Articulation Points Bridges 3
3.2.2 Strongly Connected Components 3
3.3 Matching 4
3.3.1 Max Cardinality Bipartite Matching 4
3.3.2 Min Bipartite Vertex Cover 4
3.3.3 Min Cost Bipartite Matching 4
3.3.4 General Matching 4
3.4 Flow 5
3.4.1 Max Flow – Push Relabel 5
3.4.2 Max Flow Dinic 5
3.4.3 Min Cost Max Flow 6
3.4.4 Min Cost Max Flow Capacity Scaling 6
3.5 Lowest Common Ancestor 7
3.6 Edge Coloring 7
3.7 Centroid Decomposition 7

4 Data Structures 8
4.1 Union Find Disjoint Sets 8
4.2 Sparse Table 8
4.3 Fenwick Tree 8
4.4 Data 8
4.5 Segment Tree 9
4.6 Persistent Segment Tree 9
4.7 Link Cut Tree 10
4.8 Convex Hull Trick 10
4.8.1 Partially Dynamic 10
4.8.2 Dynamic 11
4.8.3 Li Chao Segment Tree 11
4.9 Treap 11
4.9.1 BST 11
4.10 Order Statistics Tree 12
4.11 Heavy Light Decomposition 12
4.12 Queue-like undoing 13

5 Strings 13
5.1 Basics 13
5.2 Trie 13
5.3 Suffix Array 14
5.4 String Matching (KMP) 14
5.5 Z-Algorithm 14
5.6 Longest Palindrome 15
5.7 Eertree 15
5.8 Suffix Automaton 15
5.9 Aho-Corasick 15

6 Geometry 16
6.1 2D Geometry 16
6.1.1 Triangles 16
6.1.2 Quadrilaterals (Vierecke) 16
6.2 Nearest pair of points 17
6.3 Half-plane Intersection 17
6.4 3D Geometry 18
6.4.1 Spherical coordinates 18
6.4.2 Spherical Distance 18
6.4.3 Point 18
6.4.4 Convex Hull 18
6.4.5 Volume of Polyhedron 19

7 Mathematics 19
7.1 Theorems 19
7.1.1 Fibonacci numbers 19
7.1.2 Series Formulas 19
7.1.3 Binomial coefficients 19
7.1.4 Catalan's number 19
7.1.5 Pentagonal Number theorem 19
7.1.6 Hook Length formula 19
7.1.7 Pick's theorem 19
7.1.8 Burnside's Lemma 19
7.1.9 Multinomial coefficients 19
7.1.10 Gray's code 19
7.2 Game theory 19
7.2.1 Grundy's function 19
7.3 2-SAT 19
7.4 Lattice Points below a line 20
7.5 Prime numbers 20
7.6 Algebra Basics 20
7.7 Modular Inverse 21
7.8 Euler's Totient Function and Theorem 21
7.9 Discrete Logarithm: Baby Gigant 21
7.10 Discrete Root 21
7.11 Primitive Root (Generator) 22
7.12 Rabin Miller 22
7.13 Pollard Rho 22
7.14 Fast Fourier Transformation 22
7.14.1 Non-Recursive Fast Fourier Transformation 22
7.14.2 Number Theoretic Transform 22
7.14.3 Multipoint Evaluation 23
7.14.4 Newton Iteration 23
7.14.5 Inverse Series 23
7.14.6 Polynom Division with remainder 23
7.14.7 Fast Walsh-Hadamard transform 23
7.14.8 Subset convolution 23
7.15 Linear Algebra 24
7.15.1 Gauss-Jordan 24
7.15.2 Characteristic Polynomial 24
7.16 Linear Recurrence 24
7.16.1 kth Term 24
7.16.2 Berlekamp-Massey 25
7.17 Simplex 25

1 General

1.1 Compilation

```
g++ -std=gnu++17 -Wall -Wextra -Wconversion -
    fsanitize=undefined,address -
    D_GLIBCXX_DEBUG program.cpp
./a.out < input.in > output.out
```

1.2 Debugging

```
g++ -std=gnu++17 -g program.cpp
gdb a.out
  break function # set a breakpoint at the
      beginning of the specified function
  break line-number # set a breakpoint at the
      specified line
  info break # show all breakpoints
  clear # remove all breakpoints
  clear function # remove the breakpoint at
      the specified function
  clear line-number # remove the breakpoint at
       the specified line
  run < input.in > output.out # run the
      program with input and output
  step # execute next line of code, enter into
       functions
  next # execute next line of code, do not
      enter into functions
  backtrace # show backtrace of the current
      position
  backtrace full # show backtrace and values
      of local variables
  print variable-name # show value of the
      specified variable
  ptype variable-name # show type of the
      specified variable
  continue # continue execution
  finish # continue after the current function
       returns
  kill # end the execution
  quit # quit gdb
```

1.3 Random

```
// select seed to avoid being hacked
unsigned seed = chrono::system_clock::now().
    time_since_epoch().count();

mt19937 rng(seed); // random generator
uniform_int_distribution<int> unii(0, 100);
int x = unii(rng); // x in [0, 100]

uniform_real_distribution<double> unir(0.0,
    1.0);
double y = unir(rng); // y in [0.0, 1.0]

bernoulli_distribution bern(0.7);
bool b = bern(rng); // true with prob. 0.7

// bin(n, p), geom(p), normal(Exp, Var^2)
binomial_distribution<int> bin(9, 0.5);
geometric_distribution<int> geom(0.3);
normal_distribution<double> normal(5.0, 2.0);

vector<int> r(10);
shuffle(r.begin(), r.end(), rng);
```

1.4 Language Specific Functionalities

```
// integer logarithm for positive int (rounded
    down)
#define log2(x) (31 - __builtin_clz(x))
// integer logarithm for positive long long (
    rounded down)
#define log2ll(x) (63 - __builtin_clzll(x))

//overflow checking
int a,b,c;
if (__builtin_saddll_overflow(a,b,&c))
  printf("a + b > INT_MAX");
else
  printf("%d + %d = %d",a,b,c);
long long d,e,f;
if (__builtin_smulll_overflow(d,e,&f))
  printf("a * b > LONG_LONG_MAX");
else
  printf("%lld * %lld = %lld",d,e,f);
// similar functions exist for subtraction
// and various data types (s = signed, u =
    unsigned
// e.g. __builtin_usubll_overflow for unsigned
    long long

// min/max values of data types, e.g:
numeric_limits<double>()::max();
numeric_limits<int>()::min();

//Hash map with the same API as unordered_map,
    but ~3x faster. Initial capacity must be
    a power of 2 (if provided).
#include <bits/extc++.h>
__gnu_pbds::gp_hash_table<ll, int> h
    ({},{},{},{}, {1 << 16});

//sets int x to the larger number with the
    same number of bits set
int c = x&-x, r = x+c;
x = (((r^x) >> 2)/c) | r;
```

2 Dynamic Programming

2.1 Longest Increasing Subsequence

- Input: n numbers
- Find a longest increasing subsequence: $O(n \log l)$ time and $O(n)$ space
- Output: size l of the longest increasing subsequence and its previous indices in p

```
struct lis {
  int n, l = 0;
  vector<int> v, e, p;
  lis(vector<int> & a) : n(a.size()), v(n), e(
      n), p(n) {
    for (int i = 0; i < n; i++) {
      int j = lower_bound(v.begin(), v.begin()
          + l, a[i]) - v.begin();
      v[j] = a[i]; e[j] = i; p[i] = (j > 0 ? e
          [j - 1] : -1);
      l = max(l, j + 1);
    }
  }
};
```

2.2 Divide and Conquer Optimization

```
struct divideOpt {
  static const int N = 1005;
  int dp[2][N];
  void dfs(int i, int l, int r, int oL, int oR
      , vector<vector<int>> & C) {
    if (r < l)
      return;
    int m = (l + r) / 2, opt = oL;
    int & v = (dp[i][m] = 1e9);
    for (int j = oL; j <= min(oR, m-1); j++)
      if (dp[i^1][j] + C[j+1][m] < v)
        v = dp[i^1][j] + C[j+1][m], opt = j;
    dfs(i, l, m - 1, oL, opt, C);
    dfs(i, m + 1, r, opt, oR, C);
  }
  void doDp(int n, vector<vector<int>> & C) {
    for (int i = 1; i < n; i++)
      dfs(i&1, 0, n-1, 0, n-1, C);
  }
};
```

2.3 Knuths Optimization

```
struct knuthOpt {
  static const int N = 1005;
  int dp[N][N], opt[N][N];
  void doDp(int n, vector<vector<int>> & C) {
    for (int i = 1; i <= n; i++)
        dp[i][i] = 0, opt[i][i] = i;
    for (int i = 1; i <= n; i++)
      for (int j = 1; j + i <= n; j++) {
        int oL = opt[j][j+i-1];
        int oR = opt[j+1][j+i];
        dp[j][j+i] = 1e9;
        for (int l = oL; l <= oR; l++) {
          int v = dp[j][l-1] + dp[l+1][j+i] +
              C[j][j+i];
          if (v < dp[j][j+i])
            opt[j][j+i] = l, dp[j][j+i] = v;
        }
      }
  }
};
```

2.4 Alien Trick

```
struct alien {
static const int N = 1005;
vector<vector<ll>> C;
int dp[N], cnt[N];
int n, k, q[N], par[N];
alien(int n, int k, vector<vector<ll>> & C) :
    n(n), k(k), C(c);
int gt(int i, int j, bool mi = 1) {
  int lo = j+1, hi = n;
  bool fl = mi || (cnt[i] > cnt[j]);
  while (lo <= hi) {
    int mi = (lo + hi) / 2;
    ll l = dp[i] + C[i+1][mi];
    ll r = dp[j] + C[j+1][mi];
    if (l > r || (l == r && fl))
      lo = mi + 1;
    else
      hi = mi - 1;
  }
  return lo;
}
int solve(ll x, bool mi = 1) {
  for (int i = 1, l = 0, r = 0; i <= n; i++) {
    while (l<r && gt(q[l], q[l+1], mi) <= i)
      l++;
    dp[i] = dp[q[l]] + C[q[l]+1][i] + x;
    cnt[i] = cnt[q[l]] + 1;
    par[i] = q[l];
    while (l<r && gt(q[r-1], q[r], mi) >= gt(q
        [r], i))
      r--;
    q[++r] = i;
  }
  return cnt[n];
}
```

```
vector<int> reconstruct(ll x) {
  vector<int> lo, hi, ret;
  solve(x, 1);
  for (int u = n; u != 0; u = par[u])
    lo.pb(u);
  if (sz(lo) == k) return lo;
  solve(x, 0);
  for (int u = n; u != 0; u = par[u])
    hi.pb(u);
  if (sz(hi) == k) return hi;
  lo.pb(0); reverse(lo.begin(), lo.end());
  hi.pb(0); reverse(hi.begin(), hi.end());
  for (int cl=1, ch=1; i < sz(hi); ch++) {
    while (cl < sz(lo) && lo[cl] < hi[ch])
      ret.pb(lo[cl++]);
    if (lo[cl-1] <= hi[ch-1] && sz(hi) - 1 - k
        == ch - cl) {
      for (; cl < sz(hi); cl++)
        ret.pb(hi[cl]);
      break;
    }
  }
  return ret;
}
// solves the maximization problem
void doDp() {
  ll lo = -3*inf, hi = 3*inf, ans = 0;
  while (lo <= hi) {
    ll mi = lo+(hi-lo) / 2;
    if (solve(mi) <= k)
      lo = mi + 1, ans = dp[n];
    else
      hi = mi - 1;
  }
  return ans - k * hi;
}
};
```

2.5 Knapsack

- Input: list with n elements and target sum
- Output: largest subset sum $\leq$ target sum
- Runtime: $\mathcal{O}(n \cdot \max(a_i))$

```
int balancing01(vector<int> a, int sum) {
  int n = (int)a.size();
  sort(a.rbegin(), a.rend());
  int cur = 0, s = -1, off = sum - a[0] + 1;
  while (cur < sum && s + 1 < n)
    cur += a[++s];
  if (cur == sum || s + 1 == n && cur < sum)
    return cur;
  cur -= a[s];
  vector<int> dp(2 * a[0], 0);
  for (int i = 0; i < a[0]; i++) // remove
    dp[i] = -1;
  dp[cur - off] = s;
  for (int b = s; b < n; b++) {
    vector<int> ndp = dp;
    for (int i = 0; i < a[0]; i++) //add
      ndp[i+a[b]] = max(ndp[i+a[b]], dp[i]);
    for (int i = 2*a[0]-1; i >= a[0]; i--)
      for (int j=max(0,dp[i]); j<ndp[i]; j++)
        ndp[i - a[j]] = max(ndp[i - a[j]], j);
    dp.swap(ndp);
  }
  for (int i = a[0] - 1; i >= 0; i--)
    if (dp[i] >= 0)
      return i + off;
  return cur;
}
```

3 Graphs

3.1 Theorems

3.1.1 Euler's theorem

For any planar graph, $V - E + F = 1 + C$, where V is the number of graph's vertices, E is the number of edges, F is the number of faces in graph's planar drawing, and C is the number of connected components. Corollary: $V - E + F = 2$ for a 3D polyhedron.

3.1.2 Vertex covers and independent sets

Let M, C, I be a max matching, a min vertex cover, and a max independent set. Then $|M| \leq |C| = N - |I|$, with equality for bipartite graphs. Complement of an MVC is always a MIS, and vice versa. Given a bipartite graph with partitions (A, B), build a network: connect source to A, and B to sink with edges of capacities, equal to the corresponding nodes' weights, or 1 in the unweighted case. Set capacities of the original graph's edges to the infinity. Let (S, T) be a minimum $s - t$ cut. Then a maximum(-weighted) independent set is $I = (A \cap S) \cup (B \cap T)$, and a minimum(-weighted) vertex cover is $C = (A - T) \cup (B \cap S)$.

3.1.3 Erdős Gallai: Degree Sequence

A sequence $d_1 \geq d_2 \cdots \geq d_n$ is a degree sequence of a simple graph if and only if $\sum_{i=1}^{n} d_i$ is even and $\sum_{i=1}^{k} d_i \leq k(k-1) + \sum_{i=k+1}^{n} \min\{d_i, k\}$ holds for all $1 \leq k \leq n$.

3.1.4 Cayley's formula (Prüfer sequence)

Theorem: There are exactly n^{n-2} trees on n labelled vertices.
Prüfer sequence: Bijection between labelled trees of size n and sequences of length $n - 2$.

- *Tree to sequence*: For $n-2$ times, remove the leaf with smallest label and add its neighbour to the sequence.
- *Sequence to tree*: From left to right, until the sequence is empty, connect the first element from the sequence v to the lowest-label element u that is unmarked and not in the sequence. Mark u and remove v from the sequence. In the end, connect the two remaining unmarked vertices.
- Note that vertex v appears $\deg(v) - 1$ times in the sequence.

Similar results:

- The number of spanning trees of a labelled complete bipartite graph $U \cup V$ is $|U|^{|V|-1} \cdot |V|^{|U|-1}$.
- The number of labelled rooted forests on n vertices is $(n+1)^{n-1}$ (simply add one virtual vertex).
- The number of labelled forests with k connected components such that $1, \ldots, k$ all belong to different components is kn^{n-k-1}.

3.2 Traversal

3.2.1 Articulation Points Bridges

- Input: undirected graph with v vertices and e edges
- Find all articulation points and bridges: $O(v + e)$ time and space
- Output: articulation points in `art` and bridges in `bri`

```
struct articulation_points_bridges {
  int n, v = 0;
  vector<int> num, low, art;
  vector<vector<int>>& e;
  vector<pair<int, int>> bri;
  articulation_points_bridges(vector<vector<
      int>> & e) : n(e.size()), num(n, -1),
      low(n), e(e) {
    for (int i = 0; i < n; i++)
      if (num[i] == -1)
        dfs(i);
  }
  void dfs(int i, int p = -1) {
    num[i] = low[i] = v++;
    int s = 0;
    bool a = false;
    for (int j : e[i]) {
      if (j == p) {
        p = -2;
        continue;
      }
      if (num[j] >= 0)
        low[i] = min(low[i], num[j]);
      else {
        dfs(j, i);
        if (low[j] > num[i])
          bri.push_back({i, j});
        a |= low[j] >= num[i];
        low[i] = min(low[i], low[j]);
        s++;
      }
    }
    if (p == -1 ? s > 1 : a)
      art.push_back(i);
  }
};
```

3.2.2 Strongly Connected Components

- Input: graph with v vertices and e edges
- Find all strongly connected components or biconnected components: $O(v + e)$ time and $O(v)$ space
- Output: number of sccs `c` and component of each vertex in `com`

```
struct strongly_connected_components {
  int n, v = 0, c = 0;
  vector<bool> ins;
  vector<int> s, num, low, com;
```

```
  vector<vector<int>>& e;
  strongly_connected_components(vector<vector<
      int>> & e) : n(e.size()), ins(n), num(n,
       -1), low(n), com(n), e(e) {
    for (int i = 0; i < n; i++)
      if (num[i] == -1)
        dfs(i);
  }
  // use commented lines for biconnected
      components in undirected graphs
  void dfs(int i) {
  // void dfs(int i, int p = -1) {
    num[i] = low[i] = v++;
    s.push_back(i); ins[i] = true;
    for (int j : e[i]) {
      // if (j == p) {
      //   p = -1;
      //   continue;
      // }
      if (num[j] == -1)
        dfs(j);
        // dfs(j, i);
      if (ins[j])
        low[i] = min(low[i], low[j]);
    }
    if (low[i] == num[i]) {
      int j;
      do {
        j = s.back(); s.pop_back(); ins[j] =
            false;
        com[j] = c;
      } while (j != i);
      c++;
    }
  }
};
```

3.3 Matching

3.3.1 Max Cardinality Bipartite Matching

- Input: bipartite graph with v vertices and e edges
- Find a maximum cardinality matching: $O(ve)$ time and $O(v)$ space
- Output: size of the matching `mbm` and matching in `ml` and `mr`

```
struct maximum_bipartite_matching {
  int nl, nr, mbm = 0;
  vector<bool> v;
  vector<int> ml, mr;
  vector<vector<int>>& e;
  maximum_bipartite_matching(int nl, int nr,
      vector<vector<int>> & e) : nl(nl), nr(nr
      ), ml(nl, -1), mr(nr, -1), e(e) {
    int prv = 0;
    do {
      prv = mbm;
      v.assign(nr, false);
      for (int i = 0; i < nl; i++)
        if (ml[i] == -1)
          mbm += findPath(i);
    } while (mbm > prv);
  }
  bool findPath(int i) {
    for (int j : e[i])
      if (!v[j]) {
        v[j] = true;
        if (mr[j] == -1 || findPath(mr[j])) {
          ml[i] = j; mr[j] = i;
          return true;
        }
      }
    return false;
  }
};
```

3.3.2 Min Bipartite Vertex Cover

- Input: bipartite graph with v vertices and e edges
- Find a minimum vertex cover: $O(ve)$ time and $O(v)$ space
- Output: size of the cover `mbcv` and cover in `cl` and `cr`

```
struct minimum_bipartite_vertex_cover {
  int nl, nr, mbvc;
  vector<bool> cl, cr;
  vector<vector<int>>& e;
  maximum_bipartite_matching mbm;
  minimum_bipartite_vertex_cover(int nl, int
      nr, vector<vector<int>> & e) : nl(nl),
      nr(nr), cl(nl, true), cr(nr, false), e(e
      ), mbm(nl, nr, e) {
    mbvc = mbm.mbm;
    for (int i = 0; i < nl; i++)
      if (mbm.ml[i] == -1)
        findPath(i);
  }
  void findPath(int i) {
    cl[i] = false;
    for (int j : e[i])
      if (!cr[j]) {
        cr[j] = true;
        findPath(mbm.mr[j]);
      }
  }
};
```

3.3.3 Min Cost Bipartite Matching

- Input: $n \times m$ cost matrix with (positive or negative) values
- The matrix has to be 1-indexed, so add a dummy row/column
- Find a minimal-cost perfect matching in $O(nm^2)$ time and $O(nm)$ space (matches column i with `mt`$[i]$)
- Also finds labels s, t, such that $t_i + s_j \leq A_{ij}$.
- Output: cost of the matching

```
struct minBPM {
  vector<ll> mi, s, t;
  vector<int> mt, id, vis;
  int n, m;
  minBPM(int n, int m) : n(n),m(m),mi(m+1),s(m
      +1),t(n+1),mt(m+1),id(m+1),vis(m+1) {}
  ll matching(vector<vector<ll>> & a) {
    for (int i = 1, x, nx, y; i <= n; i++) {
      fill(vis.begin(), vis.end(), 0);
      fill(mi.begin(), mi.end(), 1e18);
      mt[x=0] = i;
      do {
        vis[x] = 1, y = mt[x], nx = 0;
        ll d = 1e18;
        for (int j = 1; j <= m; j++) {
          if (vis[j]) continue;
          ll v = a[y][j] - s[j] - t[y];
          mi[j] = v<mi[j] ? id[j]=x,v : mi[j];
          d = mi[j] < d ? nx=j, mi[j] : d;
        }
        for (int j = 0; j <= m; mi[j++] -= d)
          if (vis[j])
            s[j] -= d, t[mt[j]] += d;
      } while (mt[x = nx] != 0);
      for (;x!=0; mt[x]=mt[id[x]], x=id[x]);
    }
    return -s[0];
  }
};
```

3.3.4 General Matching

Tutte's theorem: Given an undirected graph on n vertices, without self-loops. Consider an $n \times n$ matrix A, where $A_{i,j} = 0$, if there's no edge between i and j. Otherwise let $i < j$ and define $A_{i,j} = x_{i,j}$, $A_{j,i} = -x_{i,j}$, where $x_{i,j}$ is some variable. Tutte's theorem states, that G has a perfect matching iff $det(A) \neq 0$ (the 0 polynomial, in terms of $x_{i,j}$). This leads to a randomised $O(n^3)$ algorithm: Replace the $x_{i,j}$'s with random numbers and compute the determinant. This is supposedly slower than Edmond's blossom, but probably shorter to implement/still good to know.

Edmond's blossom: `matching(N, G)` computes the maximum matching in the given graph on N vertices (0-indexed), represented by the adjacency list G and returns it's size (also stored in ret). Vertex i is matched with `mate[i]` (or -1). `const static int N` denotes the maximum number of vertices.

Complexity: $O(n^3)$

```
struct Blossom {
  int n, ret;
  vector<int> mate, par;
  vector<int> nx, dsu, mrk, vis;
  queue<int> pq;
  vector<vector<int>> adj;
  Blossom() {}
  Blossom(int n) : n(n), par(n+5), nx(n+5),
      mate(n+5), dsu(n+5), mrk(n+5), vis(n+5),
       adj(n+2) {
    iota(par.begin(), par.end(), 0);
  }

  void add_edge(int u, int v) {
```

```
    adj[u].push_back(v);
    adj[v].push_back(u);
  }

  int qry(int x) { return x == par[x] ? x :
      par[x] = qry(par[x]); }
  void join(int x, int y) { par[qry(x)] = qry(
      y); }

  int lca(int x, int y) {
    static int t=0;
    for (t++; ;swap(x,y)) if (x != -1) {
      if (vis[x=qry(x)]==t) return x;
      vis[x] = t;
      x = (mate[x]!=-1)?nx[mate[x]]:-1;
    }
  }

  void group(int a, int p) {
    for (int b,c; a != p; join(a,b), join(b,c)
        ,a=c) {
      b=mate[a], c=nx[b];
      if (qry(c) != p) nx[c] = b;
      if (mrk[b] == 2) mrk[b] = 1, pq.push(b);
      if (mrk[c] == 2) mrk[c] = 1, pq.push(c);
    }
  }

  void aug(int s) {
    for (int i = 0; i <= n; i++)
      nx[i] = vis[i] = -1, par[i] = i, mrk[i]
          = 0;
    while (!pq.empty()) pq.pop();
    pq.push(s); mrk[s] = 1;
    while (mate[s] == -1 && !pq.empty()) {
      int x = pq.front(); pq.pop();
      for (int i = 0,y; i < sz(adj[x]); i++) {
        if ((y=adj[x][i]) != mate[x] && qry(x)
            !=qry(y)&&mrk[y]!=2) {
          if (mrk[y]==1) {
            int p = lca(x, y);
            if (qry(x)!=p) nx[x] = y;
            if (qry(y)!=p) nx[y] = x;
            group(x,p); group(y,p);
          } else if (mate[y]==-1) {
            nx[y]=x;
            for (int j=y,k,l; j != -1; j=l) {
              k=nx[j]; l = mate[k];
              mate[j] = k; mate[k] = j;
            }
            break;
          } else {
            nx[y] = x;
            pq.push(mate[y]);
            mrk[mate[y]] = 1;
            mrk[y] = 2;
          }
        }
      }
    }
  }

  int matching() {
    fill(mate.begin(), mate.end(), -1);
    for (int i = 1; i <= n; i++)
      if (mate[i] == -1)
        aug(i);
    int ret = 0;
    for (int i = 1; i <= n; i++) {
      ret += mate[i] > i;
    }
    return ret;
  }
};
```

3.4 Flow

3.4.1 Max Flow – Push Relabel

- Input: graph with v vertices and e edges
- Find a maximum flow: $O(v^3)$ time and $O(v+e)$ space
- Output: flow between `s` and `t` with values in `e`

```
typedef long long ll;

struct push_relabel {
  struct edge {
    ll j, f, c, r;
  };
  int n;
  vector<bool> a;
  vector<int> h, c;
  vector<ll> x;
  vector<vector<edge>> e;
  queue<int> q;
  push_relabel(int n) : n(n), a(n), h(2 * n),
      c(2 * n), x(n), e(n) {}
  void addEdge(int i, int j, ll c) {
    e[i].push_back({j, 0, c, e[j].size() + (i
        == j)});
    e[j].push_back({i, 0, 0, e[i].size() - 1})
        ;
  }
  void activate(int i) {
    if (!a[i] && x[i] > 0)
      a[i] = true, q.push(i);
  }
  void push(int i, int j) {
    ll f = min(x[i], e[i][j].c - e[i][j].f);
    if (h[i] <= h[e[i][j].j] || f == 0)
      return;
    e[i][j].f += f; x[i] -= f;
    e[e[i][j].j][e[i][j].r].f -= f; x[e[i][j].
        j] += f;
    activate(e[i][j].j);
  }
  void label(int i, int k) {
    c[h[i]]--; h[i] = k; c[h[i]]++;
    activate(i);
  }
  void relabel(int i) {
    int k = 2 * n - 1;
    for (edge & ed : e[i])
      if (ed.c > ed.f)
        k = min(k, h[ed.j] + 1);
    label(i, k);
  }
  void gap(int k) {
    for (int i = 0; i < n; i++)
      if (h[i] >= k)
        label(i, max(h[i], n + 1));
  }
  void push(int i) {
    for (int j = 0; j < e[i].size() && x[i] >
        0; j++)
      push(i, j);
    if (x[i] > 0)
      if (c[h[i]] == 1)
        gap(h[i]);
      else
        relabel(i);
  }
  ll maxFlow(int s, int t) {
    h[s] = n; c[0] = n - 1; c[n] = 1;
    a[s] = a[t] = true;
    for (int i = 0; i < e[s].size(); i++) {
      x[s] += e[s][i].c;
      push(s, i);
    }
    while (!q.empty()) {
      int i = q.front();
      q.pop();
      a[i] = false;
      push(i);
    }
    return x[t];
  }
};
```

3.4.2 Max Flow Dinic

- Input: graph with v vertices and e edges
- Find a maximum flow: $O(v^2e)$ time and $O(v+e)$ space
- For unit networks the runtime is bounded by $O(e\sqrt{v})$
- Output: flow between `s` and `t` with values in `e`

```
typedef long long ll;
struct dinic {
  struct edge {
    ll j, c, f;
  };
  vector<edge> e;
  vector<vector<int>> adj;
  vector<int> lvl, ptr;
  int n, m = 0;
  dinic(int n) : n(n), adj(n), lvl(n), ptr(n)
      {}
  void addEdge(int i, int j, ll c) {
```

```
    e.push_back({j, c, 0});
    e.push_back({i, 0, 0});
    adj[i].push_back(m++);
    adj[j].push_back(m++);
  }
  bool bfs(int s, int t) {
    fill(lvl.begin(), lvl.end(), -1);
    lvl[s] = 0;
    for (queue<int> q = {s}; !q.empty();) {
      int v = q.front(); q.pop();
      for (int i : adj[v])
        if (e[i].c > e[i].f && lvl[e[i].j] <
            0) {
          lvl[e[i].j] = lvl[v] + 1;
          q.push(e[i].j);
        }
    }
    return lvl[t] != -1;
  }
  ll dfs(int v, int t, ll push) {
    if (push == 0 || v == t)
      return push;
    for (; ptr[v] < (int)adj[v].size(); ptr[v
        ]++) {
      int id = adj[v][ptr[v]];
      if (lvl[v] + 1 != lvl[e[id].j] || e[id].
          c == e[id].f)
        continue;
      ll f = dfs(e[id].j, t, min(push, e[id].c
           - e[id].f));
      if (f != 0) {
        e[id].f += f;
        e[id ^ 1].f -= f;
        return f;
      }
    }
    return 0;
  }
  ll maxFlow(int s, int t) {
    ll ret = 0;
    while (bfs(s, t)) {
      fill(ptr.begin(), ptr.end(), 0);
      while (ll f = dfs(s, t, 1e18))
        ret += f;
    }
    return ret;
  }
};
```

3.4.3 Min Cost Max Flow

- Input: graph with v vertices, e edges and no negative cycle
- Find a minimal-cost maximum flow: avg. $O(e^2)$ (worst case $O(2^v)$) time and $O(v+e)$ space
- Output: flow between s and t and its costs with values in e

```
typedef long long ll;
const ll inf = LLONG_MAX / 4;

struct min_cost_max_flow {
  typedef __gnu_pbds::priority_queue<pair<ll,
      int>> prio;
  struct edge {
    ll j, f, c, p, r;
  };
  int n;
  vector<int> p;
  vector<ll> d, pi;
  vector<vector<edge>> e;
  vector<prio::point_iterator> its;
  min_cost_max_flow(int n) : n(n), p(n), d(n),
       e(n), its(n) {}
  void addEdge(int i, int j, ll c, ll p) {
    e[i].push_back({j, 0, c, p, e[j].size() +
        (i == j)});
    e[j].push_back({i, 0, 0, -p, e[i].size() -
         1});
  }
  void path(int s) {
    swap(d, pi);
    d.assign(n, inf);
    d[s] = 0;
    prio pq; its.assign(n, pq.end());
    its[s] = pq.push({0, s});
    while (!pq.empty()) {
      ll di = pq.top().first;
      int i = pq.top().second;
      pq.pop();
      if (-di != d[i] - pi[i])
        continue;
      for (edge & ed : e[i]) {
        ll v = d[i] + ed.p;
        if (ed.c > ed.f && v < d[ed.j]) {
          d[ed.j] = v; p[ed.j] = ed.r;
          if (its[ed.j] == pq.end())
            its[ed.j] = pq.push({-(d[ed.j] -
                pi[ed.j]), ed.j});
          else
            pq.modify(its[ed.j], {-(d[ed.j] -
                pi[ed.j]), ed.j});
        }
      }
    }
  }
  pair<ll, ll> minCostMaxFlow(int s, int t) {
    ll f = 0, c = 0;
    while (path(s), d[t] < inf) {
      ll w = inf;
      for (int i = t; i != s; i = e[i][p[i]].j
          ) {
        edge & ed = e[e[i][p[i]].j][e[i][p[i
            ]].r];
        w = min(w, ed.c - ed.f);
      }
      f += w;
      c += d[t] * w;
      for (int i = t; i != s; i = e[i][p[i]].j
          ) {
        edge & ed = e[e[i][p[i]].j][e[i][p[i
            ]].r];
        e[i][p[i]].f -= w;
        ed.f += w;
      }
    }
    return {f, c};
  }
  // for negative costs, call this function
      before min cost max flow
  void setPi(int s) {
    d.assign(n, inf);
    d[s] = 0;
    bool c = true;
    for (int i = 0; i < n && c; i++) {
      c = false;
      for (int j = 0; j < n; j++)
        for (edge & ed : e[j])
          if (ed.c > ed.f && d[j] + ed.p < d[
              ed.j])
            d[ed.j] = d[j] + ed.p, c = true;
    }
    assert(!c);
  }
};
```

3.4.4 Min Cost Max Flow Capacity Scaling

- Input: graph with v vertices and e edges
- Find a minimal-cost maximum flow: $O(e^2 \log e \log c)$ (c is the maximal capacity) time and $O(v+e)$ space
- Output: flow between s and t and its costs with values in e

```
typedef long long ll;
const ll inf = LLONG_MAX / 4;

struct min_cost_max_flow {
  struct edge {
    ll j, f, c, oc, p, r;
  };
  int n;
  ll mc = 0, mp = 0;
  vector<int> p;
  vector<ll> d, pi;
  vector<vector<edge>> e;
  min_cost_max_flow(int n) : n(n), p(n), d(n),
       pi(n), e(n) {}
  void addEdge(int i, int j, ll c, ll p) {
    mc = max(mc, c); mp = max(mp, abs(p));
    e[i].push_back({j, 0, 0, c, p, e[j].size()
         + (i == j)});
    e[j].push_back({i, 0, 0, 0, -p, e[i].size
        () - 1});
  }
  void path(int s) {
    d.assign(n, inf);
    d[s] = 0;
    priority_queue<pair<ll, int>> pq;
    pq.push({pi[s], s});
    ll md = 0;
```

```
    while (!pq.empty()) {
      ll di = pq.top().first;
      int i = pq.top().second;
      pq.pop();
      if (-di != d[i] - pi[i])
        continue;
      md = max(md, d[i]);
      for (edge & ed : e[i]) {
        ll v = d[i] + ed.p;
        if (ed.c > ed.f && v < d[ed.j]) {
          d[ed.j] = v; p[ed.j] = ed.r;
          pq.push({-(d[ed.j] - pi[ed.j]), ed.j
              });
        }
      }
    }
    for (int i = 0; i < n; i++)
      if (d[i] < inf)
        pi[i] += d[i] - md;
  }
  void augment(int s, int t) {
    for (int i = t; i != s; i = e[i][p[i]].j)
        {
      edge & ed = e[e[i][p[i]].j][e[i][p[i]].r
          ];
      e[i][p[i]].f -= 1;
      ed.f += 1;
    }
  }
  pair<ll, ll> minCostMaxFlow(int s, int t) {
    addEdge(t, s, 1LL << 60, - n * mp - 1);
    ll f = 0, c = 0;
    int b = 0;
    while ((1LL << b) < mc)
      b++;
    for (; b >= 0; b--) {
      c *= 2;
      for (int i = 0; i < n; i++)
        for (edge & ed : e[i])
          ed.c *= 2, ed.f *= 2;
      for (int i = 0; i < n; i++)
        for (edge & ed : e[i])
          if ((ed.oc >> b) & 1) {
            if (ed.c == ed.f) {
              path(ed.j);
              if (d[i] < inf && d[i] + ed.p <
                  0) {
                c += d[i] + ed.p;
                e[ed.j][ed.r].f -= 1;
                ed.f += 1;
                augment(ed.j, i);
              }
            }
            ed.c += 1;
          }
    }
    f = e[t].back().f;
    c -= f * e[t].back().p;
    return {f, c};
  }
};
```

3.5 Lowest Common Ancestor

- Input: tree with v vertices
- Preprocessing: $O(v \log v)$ time and space for `sparse_table` from 4.2
- Find the lowest common ancestor of two vertices: $O(1)$ time and space
- Output: the lowest common ancestor of the two vertices

```
struct lowest_common_ancestor {
  int n, m = 0;
  vector<int> a, v, h;
  vector<vector<int>>& e;
  sparse_table st;
  lowest_common_ancestor(vector<vector<int>> &
       e, int r) : n(e.size()), a(n), v(2 * n
      - 1), h(2 * n - 1), e(e) {
    dfs(r);
    st = sparse_table(h);
  }
  void dfs(int i, int p = -1, int d = 0) {
    a[i] = m; v[m] = i; h[m++] = d;
    for (int j : e[i]) {
      if (j == p)
        continue;
      dfs(j, i, d - 1);
      v[m] = i; h[m++] = d;
    }
  }
  // calculate the lowest common ancestor of x
      and y
  int lca(int x, int y) {
    return v[st.query(min(a[x], a[y]), max(a[x
        ], a[y]) + 1)];
  }
};
```

3.6 Edge Coloring

- Input: graph with v vertices, e edges and maximal degree D
- Find a $D+1$ edge coloring: $O(ve)$ (avg. $O(v^2)$) time and $O(v^2)$ space
- Output: coloring `col` in $[0, D]$

```
typedef vector<int> VI;
typedef vector<vector<int>> VVI;

struct edge_coloring {
  VVI color, adj, free; VI y, t;

  edge_coloring(int n, int D) : color(n, VI(n,
       -1)), adj(n, VI(D + 1, -1)), free(n, VI
      (D + 1)), y(n), t(D + 1) {
    for(int i = 0; i < n; ++i)
      for(int j = 0; j <= D; ++j)
        free[i][j] = D - j;
  }

  int find_common(int u, int v) {
    while(adj[v][free[v].back()] != -1)
      free[v].pop_back();
    if(adj[v][free[u].back()] == -1)
      return free[u].back();
    if(adj[u][free[v].back()] == -1)
      return free[v].back();
    return -1;
  }

  int   trace(int a, int b, int q, int r) {
    int s = adj[r][b];
    color[q][r] = color[r][q] = b;
    adj[q][b] = r; adj[r][b] = q;
    if(s != -1) return trace(b, a, r, s);
    adj[r][a] = -1;
    free[r].push_back(a);
    return r;
  }

  void add_edge(int u, int v) {
    while(adj[u][free[u].back()] != -1)
      free[u].pop_back();

    y[0] = v;
    int j = 0, c = find_common(u, v);

    while(c < 0) {
      c = free[y[j]].back();
      if(t[c] < j && free[y[t[c]]].back() == c
          ) {
        if(trace(c, free[u].back(), u, adj[u][
            c]) != y[t[c]]) j = t[c];
        break;
      }
      t[c] = j++; y[j] = adj[u][c];
      c = find_common(u, y[j]);
    }

    while(j >= 0) {
      int v = y[j], d = color[u][v];
      adj[u][c] = v; adj[v][c] = u;
      if(j > 0) {
        free[u].push_back(d);
        free[v].push_back(d);
        adj[u][d] = adj[v][d] = -1;
      }
      color[u][v] = color[v][u] = c;
      c = d; --j;
    }
  }
};
```

3.7 Centroid Decomposition

- Input: tree with v vertices
- Find a centroid decomposition: $O(v \log v)$ time and $O(v)$

space
- Output: centroid decomposition with subtree sizes in s, parents in cp and depths in cd

```
#define MAXN 100000

int n, cs, ms, cc, s[MAXN], cp[MAXN], cd[MAXN
    ];
bool u[MAXN];
vector<int> e[MAXN];

void findCentroid(int i, int p = -1) {
  int cms = 0;
  s[i] = 1; cp[i] = p;
  for (int j : e[i]) {
    if (u[j] || j == p)
      continue;
    findCentroid(j, i);
    cms = max(cms, s[j]);
    s[i] += s[j];
  }
  cms = max(cms, cs - s[i]);
  if (cms < ms)
    ms = cms, cc = i;
}

void findCentroidDecomposition(int i, int p =
    -1, int d = 0) {
  cs = s[i]; ms = cs + 1;
  findCentroid(i);
  i = cc; cd[i] = d; u[i] = true;
  if (cp[i] != -1)
    s[cp[i]] = cs - s[i];
  s[i] = cs; cp[i] = p;
  for (int j : e[i])
    if (!u[j])
      findCentroidDecomposition(j, i, d + 1);
}

void centroidDecomposition(int i = 0) {
  s[i] = n;
  findCentroidDecomposition(i);
}
```

4 Data Structures

4.1 Union Find Disjoint Sets

- Input: n elements
- Preprocessing: $O(n)$ time and space
- Requesting the set of an element, to merge two sets and the size of a set: $O(1)$ time and space

```
struct DSU {
  vector<int> hist, lst = {0}, par, s;
  DSU(int n) : par(n+1), s(n+1) {
    iota(par.begin(), par.end(), 0);
    fill(s.begin(), s.end(), 1);
  }
  int qry(int x) {
    return par[x] == x ? x : qry(par[x]);
  }
  void join(int x, int y) {
    if ((x=qry(x)) == (y=qry(y))) {
      hist.push_back(-1);
      return;
    }
    if (s[y] < s[x])
      swap(x, y);
    s[par[x] = y] += s[x];
    hist.push_back(x);
  }
  void snapshot() {
    lst.push_back((int)hist.size());
  }
  void rollback() {
    while (hist.size() != lst.back()) {
      int u = hist.back();
      if (0 <= u)
        s[par[u]] -= s[u], par[u] = u;
      hist.pop_back();
    }
    lst.pop_back();
  }
};
```

4.2 Sparse Table

- Input: n elements with an associative and absorbing combination
- Preprocessing: $O(n \log n)$ time and space
- Requesting the result of the combination of all elements in the range $[l, r[$: $O(1)$ time and space

```
#define log2(x) (31 - __builtin_clz(x))

struct sparse_table {
  int n;
  vector<int> a;
  vector<vector<int>> st;
  int combine(int dl, int dr) {
    return a[dl] > a[dr] ? dl : dr;
  }
  sparse_table() {}
  sparse_table(vector<int> & a) : n(a.size()),
      a(a), st(log2(n) + 1, vector<int>(n)) {
    for (int i = 0; i < n; i++)
      st[0][i] = i;
    for (int j = 1; 1 << j <= n; j++)
      for (int i = 0; i + (1 << j) <= n; i++)
        st[j][i] = combine(st[j - 1][i], st[j
            - 1][i + (1 << (j - 1))]);
  }
  // query the data on the range [l, r[
  int query(int l, int r) {
    int s = log2(r - l);
    return combine(st[s][l], st[s][r - (1 << s
        )]);
  }
};
```

4.3 Fenwick Tree

- Input: n elements with an associative and reversible combination
- Preprocessing: $O(n \log n)$ time and $O(n)$ space
- Requesting to change an element: $O(\log n)$ time and $O(1)$ space
- Requesting the result of the combination of all elements in the range $[l, r[$: $O(\log n)$ time and $O(1)$ space

```
struct fenwick_tree {
  int n;
  vector<int> a, f;
  fenwick_tree(int n = 0) : n(n), a(n), f(n +
      1) {}
  fenwick_tree(vector<int> & a) : fenwick_tree
      (a.size()) {
    for (int i = 0; i < n; i++)
      setValue(i, a[i]);
  }
  void changeValue(int i, int d) {
    for (a[i++] += d; i <= n; i += i & -i)
      f[i] += d;
  }
  void setValue(int i, int v) {
    changeValue(i, v - a[i]);
  }
  int getSum(int i) {
    int s = 0;
    for (i++; i; i -= i & -i)
      s += f[i];
    return s;
  }
  // get the sum of the range [l, r[
  int getSum(int l, int r) {
    return getSum(r - 1) - getSum(l - 1);
  }
};
```

4.4 Data

- Data with an associative combination
- Associative operation on the data which commutes with the combination

```
// data (sum and length of a segment)
struct data {
  int s = 0, l = 0;
};
// operation on the data (x -> a * x + b)
struct operation {
  int a = 1, b = 0;
};
// alternatively use typedefs for simpler data
    and operations
// combine the data from different segments
data combine(data dl, data dr) {
```

```
  return {dl.s + dr.s, dl.l + dr.l};
}
// calculate the result of an operation on the
   data
data calculate(operation o, data d) {
  return {o.a * d.s + o.b * d.l, d.l};
}
// merge an operation onto another operation
operation merge(operation ot, operation ob) {
  return {ot.a * ob.a, ot.b + ot.a * ob.b};
}
```

4.5 Segment Tree

- Input: n elements
- Preprocessing: $O(n)$ time and space
- Requesting to change an element or interval: $O(\log n)$ time and space
- Requesting the result of the combination on all elements in the range $[l, r[$: $O(\log n)$ time and space

```
struct segment_tree {
  struct data;
  struct operation;
  static data combine(data dl, data dr);
  static data calculate(operation o, data d);
  static operation merge(operation ot,
      operation ob);
  int n, h;
  vector<data> t;
  vector<operation> o;
  segment_tree(int n = 0) : n(n), h(32 -
      __builtin_clz(n)), t(2 * n), o(n) {}
  segment_tree(vector<data> & a) :
      segment_tree(a.size()) {
    for (int i = 0; i < n; i++)
      t[i + n] = a[i];
    for (int x = n - 1; x > 0; x--)
      t[x] = combine(t[x << 1], t[x << 1 | 1])
          ;
  }
  void apply(int x, operation op) {
    t[x] = calculate(op, t[x]);
    if (x < n)
      o[x] = merge(op, o[x]);
  }
  void push(int x) {
    for (int s = h; s > 0; s--) {
      int c = x >> s;
      apply(c << 1, o[c]);
      apply(c << 1 | 1, o[c]);
      o[c] = operation();
    }
  }
  void build(int x) {
    while (x >>= 1)
      t[x] = calculate(o[x], combine(t[x <<
          1], t[x << 1 | 1]));
  }
  // set the data at the position i
  void setValue(int i, data d) {
    i += n;
    push(i);
    t[i] = d;
    build(i);
  }
  // query the data on the range [l, r[
  data query(int l, int r) {
    l += n; r += n;
    push(l); push(r - 1);
    data dl, dr;
    for (; l < r; l >>= 1, r >>= 1) {
      if (l & 1)
        dl = combine(dl, t[l++]);
      if (r & 1)
        dr = combine(t[--r], dr);
    }
    return combine(dl, dr);
  }
  // apply an operation on the range [l, r[
  void apply(int l, int r, operation op) {
    l += n; r += n;
    push(l); push(r - 1);
    int xl = l, xr = r;
    for (; l < r; l >>= 1, r >>= 1) {
      if (l & 1)
        apply(l++, op);
      if (r & 1)
        apply(--r, op);
    }
    build(xl); build(xr - 1);
  }
};
```

4.6 Persistent Segment Tree

- Input: n elements
- Preprocessing: $O(n)$ time and space
- Requesting to change an element or interval: $O(\log n)$ time and space
- Requesting the result of the combination on all elements in the range $[l, r[$ for the version v of the segment tree: $O(\log n)$ time and space

```
struct persistent_segment_tree {
  struct data;
  struct operation;
  static data combine(data dl, data dr);
  static data calculate(operation o, data d);
  static operation merge(operation ot,
      operation ob);
  struct node {
    node *l, *r;
    data t;
    operation o;
    node(node *l, node *r) : l(l), r(r) {
      t = combine(l->t, r->t);
    }
    node(data t) : t(t) {}
    node(node *l, node *r, data t, operation o
        ) : l(l), r(r), t(t), o(o) {}
  };
  int n;
  vector<node*> t;
  persistent_segment_tree(vector<data> & a) :
      n(a.size()) {
    t.push_back(build(a, 0, n));
  }
  node* build(vector<data> & a, int l, int r)
      {
    if (l + 1 == r)
      return new node(a[l]);
    int m = (l + r) / 2;
    return new node(build(a, l, m), build(a, m
        , r));
  }
  data query(node *x, int l, int r, int xl,
      int xr) {
    if (l <= xl && xr <= r)
      return x->t;
    if (xr <= l || r <= xl)
      return data();
    int xm = (xl + xr) / 2;
    return calculate(x->o, combine(query(x->l,
         l, r, xl, xm), query(x->r, l, r, xm,
        xr)));
  }
  // query the data on the range [l, r[ for
      version v
  data query(int v, int l, int r) {
    return query(t[v], l, r, 0, n);
  }
  // query the data on the range [l, r[
  data query(int l, int r) {
    return query(t.back(), l, r, 0, n);
  }
  node* apply(node *x, int l, int r, operation
       o, int xl, int xr, operation xo) {
    if (l <= xl && xr <= r)
      return new node(x->l, x->r, calculate(
          merge(o, xo), x->t), merge(merge(o,
          xo), x->o));
    if (xr <= l || r <= xl)
      return new node(x->l, x->r, calculate(xo
          , x->t), merge(xo, x->o));
    int xm = (xl + xr) / 2;
    xo = merge(xo, x->o);
    return new node(apply(x->l, l, r, o, xl,
        xm, xo), apply(x->r, l, r, o, xm, xr,
        xo));
  }
  // apply an operation on the range [l, r[
      for version v
  void apply(int v, int l, int r, operation o)
       {
    t.push_back(apply(t[v], l, r, o, 0, n,
```

```
        operation()));
  }
  // apply an operation on the range [l, r[
  void apply(int l, int r, operation o) {
    t.push_back(apply(t.back(), l, r, o, 0, n,
         operation()));
  }
};
```

4.7 Link Cut Tree

- Input: v vertices
- Requesting to link or cut two nodes, query or change elements on a path, ...: amortized $O(\log n)$ time and space

```
struct link_cut_tree {
  struct data;
  struct operation;
  static data combine(data dl, data dr);
  static data calculate(operation o, data d);
  static operation merge(operation ot,
      operation ob);
  struct node {
    node *p = 0, *c[2] = {0, 0};
    bool r = false;
    data d, t;
    operation o;
  };
  vector<node> v;
  link_cut_tree(int n) : v(n) {}
  bool isRoot(node *x) {
    return !x->p || x->p->c[0] != x && x->p->c
        [1] != x;
  }
  int direction(node *x) {
    return x->p && x->p->c[1] == x;
  }
  data getData(node *x) {
    return x ? x->t : data();
  }
  void fix(node *x) {
    for (int i = 0; i < 2; i++)
      if (x->c[i])
        x->c[i]->p = x;
    x->t = combine(getData(x->c[0]), combine(x
        ->d, getData(x->c[1])));
  }
  void apply(node *x, bool r, operation o) {
    x->r ^= r;
    x->d = calculate(o, x->d);
    x->t = calculate(o, x->t);
    x->o = merge(o, x->o);
  }
  void push(node *x) {
    if (x->r)
      swap(x->c[0], x->c[1]);
    for (int i = 0; i < 2; i++)
      if (x->c[i])
        apply(x->c[i], x->r, x->o);
    x->r = false;
    x->o = operation();
  }
  void rotate(node *x) {
    node *p = x->p;
    int d = direction(x);
    p->c[d] = x->c[!d];
    if (!isRoot(p))
      p->p->c[direction(p)] = x;
    x->p = p->p;
    x->c[!d] = p;
    fix(p);
    fix(x);
  }
  void splay(node *x) {
    while (!isRoot(x) && !isRoot(x->p)) {
      push(x->p->p); push(x->p); push(x);
      direction(x) == direction(x->p) ? rotate
          (x->p) : rotate(x);
      rotate(x);
    }
    if (!isRoot(x))
      push(x->p), push(x), rotate(x);
    push(x);
  }
  node* outermost(node *x, int d) {
    push(x);
    while (x->c[d]) {
      x = x->c[d];
      push(x);
    }
    splay(x);
    return x;
  }
  node* expose(node *x) {
    node *r = 0;
    for (node *p = x; p; p = p->p) {
      splay(p);
      if (r) { /* TODO: remove r as virtual
          child from p->d */ }
      if (p->c[1]) p->d = combine(p->d, p->c
          [1]->t);
      p->c[1] = r;
      fix(p);
      r = p;
    }
    splay(x);
    return r;
  }
  // get the root of the tree containing x
  int findRoot(int x) {
    expose(&v[x]);
    return outermost(&v[x], 0) - &v[0];
  }
  // make x the root of the tree
  void makeRoot(int x) {
    expose(&v[x]);
    v[x].r ^= 1;
  }
  // get the parent of x
  int getParent(int x) {
    expose(&v[x]);
    return v[x].c[0] ? outermost(v[x].c[0], 1)
         - &v[0] : -1;
  }
  // set the parent of x to y
  void link(int x, int y) {
    makeRoot(x); push(&v[x]);
    expose(&v[y]);
    v[y].p = &v[x]; v[x].c[0] = &v[y];
    fix(&v[x]);
  }
  // cut the link between x and its parent
  void cut(int x) {
    expose(&v[x]);
    v[x].c[0]->p = 0;
    v[x].c[0] = 0;
    fix(&v[x]);
  }
  // cut the link between x and y
  void cut(int x, int y) {
    makeRoot(y);
    cut(x);
  }
  bool inSameComponent(int x, int y) {
    return findRoot(x) == findRoot(y);
  }
  // calculate the lowest common ancestor of x
      and y
  int lca(int x, int y) {
    if (x == y)
      return x;
    expose(&v[x]);
    node *z = expose(&v[y]);
    return v[x].p ? z - &v[0] : -1;
  }
  // query the data along the path from x to y
  data query(int x, int y) {
    makeRoot(x);
    expose(&v[y]);
    return v[y].t;
  }
  // apply an operation along the path from x
      to y
  void apply(int x, int y, operation o) {
    makeRoot(x);
    expose(&v[y]);
    apply(&v[y], false, o);
  }
};
```

4.8 Convex Hull Trick

4.8.1 Partially Dynamic

- Adding a line to the convex hull with increasing slope: amortized $O(1)$

- Requesting the maximal value at position x: amortized $O(1)$ with increasing positions (otherwise $O(\log n)$)

```
typedef long long ll;
const ll inf = LLONG_MAX;

ll divide(ll a, ll b) {
  return a / b - ((a ^ b) < 0 && a % b);
}
// for doubles, use inf = 1.0 / 0 and div(a, b
    ) = a / b

// for non-increasing queries, use commented
    lines
struct line {
  ll a, b, r;
  // bool operator<(ll x) const {
  //   return r < x;
  // }
};

struct convex_hull : vector<line> {
  int p = 0;
  bool isect(line & x, line & y) {
    if (x.a == y.a)
      x.r = x.b > y.b ? inf : -inf;
    else
      x.r = divide(y.b - x.b, x.a - y.a);
    return x.r >= y.r;
  }
  // add the line a * x + b to the convex hull
      , added lines must have increasing slope
  void add(ll a, ll b) {
    line l = {a, b, inf};
    if (size() - p > 0 && isect(back(), l))
      return;
    while (size() - p > 1 && (--(--end()))->r
        >= back().r)
      pop_back(), isect(back(), l);
    push_back(l);
  }
  // query the maximal value at position x
  ll query(ll x) {
    while (x > at(p).r)
      p++;
    return at(p).a * x + at(p).b;
    // auto l = *lower_bound(begin(), end(), x
        );
    // return l.a * x + l.b;
  }
};
```

4.8.2 Dynamic

- Adding a line to the convex hull: amortized $O(\log n)$
- Requesting the maximal value at position x: $O(\log n)$

```
typedef long long ll;
const ll inf = LLONG_MAX;
ll divide(ll a, ll b) {
  return a / b - ((a ^ b) < 0 && a % b);
}
// for doubles, use inf = 1.0 / 0 and div(a, b
    ) = a / b

struct line {
  mutable ll a, b, r;
  bool operator<(const line & o) const {
    return a < o.a;
  }
  bool operator<(ll x) const {
    return r < x;
  }
};

struct convex_hull : multiset<line, less<>> {
  bool isect(iterator x, iterator y) {
    if (y == end()) {
      x->r = inf;
      return false;
    }
    if (x->a == y->a)
      x->r = x->b > y->b ? inf : -inf;
    else
      x->r = divide(y->b - x->b, x->a - y->a);
    return x->r >= y->r;
  }
  // add the line a * x + b to the convex hull
  void add(ll a, ll b) {
    auto y = insert({a, b, 0}), x = y++;
    while (isect(x, y))
      y = erase(y);
    if ((y = x) != begin() && isect(--x, y))
      isect(x, erase(y));
    while ((y = x) != begin() && (--x)->r >= y
        ->r)
      isect(x, erase(y));
  }
  // query the maximal value at position x
  ll query(ll x) {
    auto l = *lower_bound(x);
    return l.a * x + l.b;
  }
};
```

4.8.3 Li Chao Segment Tree

- Adding a segment to the tree: $O(\log^2 C)$
- For lines the insertion works in $O(\log C)$
- Requesting the maximal value at position x: $O(\log C)$

```
struct Line {
  ll m, n;
  Line() : m(0), n(LLONG_MIN) {}
  Line(ll _m, ll _n) : m(_m), n(_n) {}
  ll get(ll x) { return m*x + n; }
  bool majorize(ll l, ll r, Line x) {
    return get(l) >= x.get(l) && get(r) >= x.
        get(r);
  }
};
struct LiChao {
  LiChao *c[2] = {0, 0}; Line d = Line();
  ll qry(ll l, ll r, ll x) {
    ll ret = d.get(x), m = l + (r - l) / 2;
    if (x <= m) {
      if (c[0]) ret = max(ret, c[0]->qry(l, m,
          x));
    } else {
      if (c[1]) ret = max(ret, c[1]->qry(m +
          1, r, x));
    }
    return ret;
  }
  void modify(ll l, ll r, Line v) {
    if (v.majorize(l, r, d)) swap(d, v);
    if (d.majorize(l, r, v)) return;
    if (d.get(l) < v.get(l)) swap(d, v);
    ll m = l + (r - l) / 2;
    if (d.get(m) < v.get(m)) {
      swap(d, v);
      if (!c[0]) c[0] = new LiChao();
      c[0]->modify(l, m, v);
    } else {
      if (!c[1]) c[1] = new LiChao();
      c[1]->modify(m + 1, r, v);
    }
  }
  void upd(ll l, ll r, ll x, ll y, Line v) {
    if (r < x || y < l) return;
    if (x <= l && r <= y) return modify(l, r,
        v);
    ll m = l + (r - l) / 2;
    if (x <= m) {
      if (!c[0]) c[0] = new LiChao();
      c[0]->upd(l, m, x, y, v);
    }
    if (y > m) {
      if (!c[1]) c[1] = new LiChao();
      c[1]->upd(m + 1, r, x, y, v);
    }
  }
};
```

4.9 Treap

4.9.1 BST

```
/* use arrays if the TL is tight! */
#define ep emplace_back
mt19937 rng(chrono::steady_clock::now().
    time_since_epoch().count());
template<class T> struct treap {
  vector<int> L, R, prio;
  vector<T> val;
  int root = 0;
  treap() {
    L.ep(), R.ep(), prio.ep(), val.ep();
```

```
}
ll genRnd() {
  return uniform_int_distribution<int>(0,1e9)(rng);
}
int create(const T& v, ll p) {
  val.ep(v), L.ep(), R.ep(), prio.ep(p);
  return sz(val)-1;
}
void pushdown(int x) {
  if (!x) return;
}
void cmb(int x) {
  if (!x) return;
  pushdown(L[x]), pushdown(R[x]);
  // combine data from left and right child
}
void split(int cur, int &l, int &r, const T& v) {
  pushdown(cur);
  if (!cur)
    l = r = 0;
  else if (val[cur] < v) {
    int x = R[l = cur];
    split(R[cur], x, r, v);
    R[l] = x;
    cmb(l);
  } else {
    int x = L[r = cur];
    split(L[cur], l, x, v);
    L[r] = x;
    cmb(r);
  }
}
void merge(int &cur, int l, int r) {
  pushdown(l), pushdown(r);
  if (!l || !r)
    cur = !l ? r : l;
  else if (prio[l] > prio[r]) {
    int x = R[cur = l];
    merge(x, R[l], r);
    R[cur] = x;
  } else {
    int x = L[cur = r];
    merge(x, l, L[r]);
    L[cur] = x;
  }
  cmb(cur);
}
void add(int &cur, const T& v, ll p) {
  pushdown(cur);
  if (!cur) {
    cur = create(v, p);
  } else if (p > prio[cur]) {
    int nx = create(v, p);
    split(cur, L[nx], R[nx], v);
    cur = nx;
  } else {
    int x = val[cur] < v ? R[cur] : L[cur];
    add(x, v, p);
    if (val[cur] < v) R[cur] = x; else L[cur] = x;
  }
  cmb(cur);
}
void rem(int& cur, const T& v) {
  pushdown(cur);
  if (!(v < val[cur]) && !(val[cur] < v)) {
    merge(cur, L[cur], R[cur]);
  } else if (val[cur] < v) {
    int x = R[cur];
    rem(x, v);
    R[cur] = x;
  } else {
    int x = L[cur];
    rem(x, v);
    L[cur] = x;
  }
  cmb(cur);
}
void ins(const T& v) {
  add(root, v, genRnd());
}
void del(const T& v) {
  rem(root, v);
}
T qry(int x) {
  pushdown(x);
  return !R[x] ? val[x] : qry(R[x]);
}
T qryMax() {
  return qry(root);
}
};
```

4.10 Order Statistics Tree

```
#include <bits/stdc++.h>
#include <ext/pb_ds/assoc_container.hpp>
#include <ext/pb_ds/tree_policy.hpp>

using namespace std;
using namespace __gnu_pbds;

typedef tree<int, null_type, less<int>,
    rb_tree_tag,
    tree_order_statistics_node_update> ost; // null_mapped_type instead of null_type in older versions

int main() {
  ost X;
  for(int i = 0; i < 100; i += 10)
    X.insert(i); // insert 0, 10,..., 90

  cout << X.order_of_key(30) << endl;
  // result: 3 (number of keys < 30)
  cout << X.order_of_key(31) << endl;
  // result: 4 (number of keys < 31)
  cout << *X.find_by_order(3) << endl;
  // result: 30 (3th element (0-based))

  // first >= 30 (lower) and > 30 (upper)
  cout << *X.lower_bound(30) << endl; // 30
  cout << *X.upper_bound(30) << endl; // 40

  X.erase(20); // remove element
  return 0;
}
```

4.11 Heavy Light Decomposition

- Heavy Light Decomposition of a tree. To update / query a path from u to v call work(u, v, operation).

```
struct HLD {
  vector<int> par, depth, root, heavy, pos;
  vector<vector<int>>& e;

  HLD(int n, vector<vector<int>> & e) : e(e),
      par(n+1), depth(n+1), root(n+1), heavy(n+1), pos(n+1) {
    fill(heavy.begin(), heavy.end(), -1);
    par[1] = -1, depth[1] = 0;
    dfs(1);
    for (int i = 1, cur = 0; i <= n; i++)
      if (par[i] == -1 || heavy[par[i]] != i)
        for (int j = i; j != -1; j = heavy[j])
          root[j] = i, pos[j] = cur++;
  }
  int dfs(int u) {
    int sz = 1, mx = 0;
    for (int v: e[u]) if (v != par[u]) {
      par[v] = u;
      depth[v] = depth[u] + 1;
      int sub = dfs(v);
      if (sub > mx)
        mx = sub, heavy[u] = v;
      sz += sub;
    }
    return sz;
  }
  template<class T> void work(int u, int v, T op) {
    for (; root[u] != root[v]; v = par[root[v]]) {
      if (depth[root[u]] > depth[root[v]])
        swap(u, v);
      op(root[v], pos[root[v]], pos[v]);
    }
    if (depth[u] > depth[v]) swap(u, v);
    op(root[u], pos[u], pos[v]);
  }
  int dist(int u, int v) {
    int ret = -1;
```

```
    work(u, v, [this, &ret](int x, int l, int
        r) {
      ret += r - l + 1;
    });
    return ret;
  }
};
```

4.12 Queue-like undoing

- Implements a queue using a single stack.
- TODO: Implement a data structure, which supports applying updates and undoing the last update in a commutative way, i.e., `undo()`, `apply(o)` has the same effect as `apply(o)`, `undo()`.
- Application: This trick allows for queue-like undoing of updates.
- If there are q `push` and `pop` operations, then this trick does $\mathcal{O}(q \log q)$ `apply` and `undo` operations on the data structure.

```
struct dsqueue {
  struct operation;
  struct dat {
    void undo();
    void apply(operation o);
  };
  dat ds;
  vector<pair<int, operation>> a;
  int cnt = 0;
  dsqueue() {}
  void pop() {
    if (!cnt) {
      cnt = (int)a.size();
      reverse(a.begin(), a.end());
      for (auto& [t, o]: a)
        ds.undo(), t = 0;
      for (auto& [t, o]: a)
        ds.apply(o);
    }
    deque<operation> b[2];
    for (int t : {1, 0}) {
      for (int i = 0; !t ? i < (cnt & -cnt) :
          a.back().st; i++) {
        b[t].push_front(a.back().nd);
        a.pop_back();
        ds.undo();
      }
    }
    for (int t : {1, 0}) {
      for (auto& o: b[t]) {
        ds.apply(o);
        a.emplace_back(t, o);
      }
    }
    cnt--;
    ds.undo();
    a.pop_back();
  }
  void push(operation o) {
    a.emplace_back(1, o);
    ds.apply(o);
  }
};
```

5 Strings

5.1 Basics

```
string s = "abc.xabc";

// copy string
string r(s.begin(), s.end()); // "abc.xabc"

// reverse string (2 ways)
reverse(s.begin(), s.end()); // "cbax.cba"
s = string(s.rbegin(), s.rend()); // "abc.xabc
    "

// find in string
size_t i = s.find('b'); // 1 (find character)
i = s.find("bc"); // 1 (substring in O(n*m))
i = s.find("bc", 2); // 6 (from position 2)
i = s.rfind("bc", 6); // 6 (backwards from 6)
i = s.find_first_of("xyz"); // 4 (first
    occurrence of x,y or z)
i = s.find_last_of("xyz"); // 4
i = s.find_first_not_of("abc"); // 3 ('.')

if(i != string::npos) cout << i; // found

// substrings
r = s.substr(s.find('x'), 2); // xa

// number conversion
r = to_string(42); // "42"
r = to_string(42.0);// "42.000000"
int x = stoi("42"); // 42
long long y = stoll("123456789123456789"); //
    123...
double z = stod("1e7"); // 1e+007

// alternative conversion approach
r = "42 123456789123456789 1e7";
sscanf(r.c_str(), "%d %lld %lf", &x, &y, &z);

// more complex transformations
transform(s.begin(), s.end(), s.begin(), ::
    toupper); // "ABC.XABC"
transform(s.begin(), s.end(), s.begin(), ::
    tolower); // "abc.xabc"
// where tolower takes and returns a char

// with stringstream
stringstream ss("This is a test.");
while(!ss.eof()) {
    string next;
    ss >> next;
}

// getline with custom delimiter
ss = stringstream("comma,separ\nated,text");
cout << ss.str() << endl; // "comma...text"
string token;
while(getline(ss, token, ',')) {
    cout << token << endl; // "comma", "separ\
        nated", "text"
}
```

5.2 Trie

- Input: n strings of combined length m
- Preprocessing: $O(m)$ time and space
- Requesting to insert or delete a string of length l from the trie: $O(l)$ time and space
- Requesting, whether a string of length l is in the trie: $O(l)$ time and space

```
struct trie {
  #define ep emplace_back
  vector<array<int,26>> nx;
  vector<int> isFin;
  int cnt;
  trie() : cnt(1) { add(); }
  trie(const vector<string>& s) : cnt(1) {
    add();
    for (auto x: s) ins(x);
  }
  void add() {
    cnt++, nx.ep(), isFin.ep();
  }
  void ins(const string& s) {
    int cur = 0;
    for (auto c: s) {
      if (!nx[cur][c-'a'])
        nx[cur][c-'a'] = cnt, add();
      cur = nx[cur][c-'a'];
    }
    isFin[cur]++;
  }
  void del(const string& s) {
    int cur = 0;
    for (auto c: s)
      cur = nx[cur][c-'a'];
    isFin[cur]--;
  }
  int find(const string& s) {
    int cur = 0;
    for (auto c: s) {
      if (!nx[cur][c-'a'])
        return 0;
      cur = nx[cur][c-'a'];
    }
    return isFin[cur] > 0;
  }
};
```

5.3 Suffix Array

- Input: string of length n
- Preprocessing: $O(n \log n)$ time and $O(n)$ space
- Requesting the matches of a pattern of length m in the string: $O(m \log n)$ time and $O(1)$ space
- Requesting the longest repeating substring: $O(n)$ time and space

```
struct suffix_array {
  int n;
  vector<int> rnk, c, suf, sra, tsu, lcp;
  sparse_table sp;
  string s;
  suffix_array() {}
  suffix_array(string& s) : s(s), n(sz(s)),
      rnk(n+1), c(26*n+1), suf(n+1), sra(n+1),
       tsu(n+1), lcp(n+1) {
    for(int i = 0; i < n; i++)
      suf[i] = i, sra[i] = s[i]-'a';
    for(int k = 1; k < n; k <<= 1) {
      countingSort(k);
      countingSort(0);
      tsu[suf[0]] = 0;
      for(int i = 1; i < n; i++)
        tsu[suf[i]] = tsu[suf[i - 1]] + ((sra[
            suf[i]] == sra[suf[i - 1]] && (suf
            [i] + k < n ? sra[suf[i] + k] :
            -1) == (suf[i - 1] + k < n ? sra[
            suf[i - 1] + k] : -1)) ? 0 : 1);
      for(int i = 0; i < n; i++)
        sra[i] = tsu[i];
      if(sra[suf[n - 1]] == n - 1)
        break;
    }
  }
  void countingSort(int k) {
    int mra = 0, sum = 0, tmp = 0;
    fill(c.begin(), c.end(), 0);
    for (int i = 0; i < n; i++)
      c[i + k < n ? sra[i + k] + 1 : 0]++, mra
          = max(mra, i + k < n ? sra[i + k] +
          1 : 0);
    for (int i = 0; i <= mra; i++)
      tmp = sum + c[i], c[i] = sum, sum = tmp;
    for (int i = 0; i < n; i++)
      tsu[c[suf[i] + k < n ? sra[suf[i] + k] +
          1 : 0]++] = suf[i];
    for (int i = 0; i < n; i++)
      suf[i] = tsu[i];
  }
  int findString(const string & p, bool eql) {
    int l = 0, r = n - 1;
    while(l < r) {
      int m = (l + r) / 2;
      int res = strncmp(& s.front() + suf[m],
          & p.front(), p.size());
      if(res > 0 || (eql && res == 0))
        r = m;
      else
        l = m + 1;
    }
    int res = strncmp(& s.front() + suf[l], &
        p.front(), p.size());
    if(res < 0 || (!eql && res == 0))
      l++;
    return l;
  }
  // get the indices of matches from p in s
  vector<int> findMatches(const string & p) {
    int l = findString(p, true), r =
        findString(p, false);
    vector<int> res;
    for(int i = l; i < r; i++)
      res.push_back(suf[i]);
    return res;
  }
  // initialize the longest common prefix, get
       the starting index and the length of
       the longest repeated substring
  pair<int, int> longestCommonPrefix() {
    int lrs = 0, rsp = -1;
    for (int i = 0; i < n; i++)
      rnk[suf[i]] = i;
    for (int i = 0, k = 0; i < n; i++) {
      if (rnk[i] == n - 1) {
        k = 0;
        continue;
      }
      int j = suf[rnk[i] + 1];
      while (max(i, j) + k < n && s[i + k] ==
          s[j + k])
        k++;
      lcp[rnk[i]] = k;
      if (k > lrs)
        lrs = k, rsp = i;
      k = max(k - 1, 0);
    }
    sp = sparse_table(lcp);
    return {rsp, lrs};
  }
  // get the length of the longest common
      prefix starting at i and j
  int getLongestCommonPrefix(int i, int j) {
    if (i == j)
      return n - i;
    i = rnk[i]; j = rnk[j];
    if (i > j)
      swap(i, j);
    return lcp[sp.query(i, j)];
  }
  // get the length of the longest common
      suffix ending at i and j
  int getLongestCommonSuffix(int i, int j) {
    int l = 1, r = min(i, j) + 1;
    while (l <= r) {
      int m = (l + r) / 2;
      if (getLongestCommonPrefix(i - m + 1, j
          - m + 1) >= m)
        l = m + 1;
      else
        r = m - 1;
    }
    return r;
  }
  int operator[](int i) {
    return suf[i];
  }
};
```

5.4 String Matching (KMP)

- Input: string of length n and a pattern of length m
- Find the matches of the pattern in the string: $O(n+m)$ time and $O(1)$ space
- Output: the matches of the pattern in the string

```
#define MAXN 1000000

int n, m, r[MAXN];
string s;

void preprocessPattern(string & p) {
  r[0] = -1;
  for (int i = 0, j = -1; i < m; i++) {
    while (j >= 0 && p[i] != p[j])
      j = r[j];
    r[i + 1] = ++j;
  }
}

// get the indices of matches from p in s
vector<int> findMatches(string & s, string & p
    ) {
  n = s.size(); m = p.size();
  preprocessPattern(p);
  vector<int> res;
  for (int i = 0, j = 0; i < n; i++) {
    while (j >= 0 && s[i] != p[j])
      j = r[j];
    j++;
    if (j == m)
      res.push_back(i - j + 1), j = r[j];
  }
  return res;
}
```

5.5 Z-Algorithm

- Runtime $O(n)$ for a string s of length n.
- For $1 \leq i < n$, $z[i]$ gives the length of the longest common prefix of s and $s[i \ldots n-1]$, and $z[0] = 0$.
- E.g. $s =$ "$aaabaab$" $\Rightarrow z = [0, 2, 1, 0, 2, 1, 0]$.

```
vector<int> z_function(string s) {
  int n = (int) s.length();
  vector<int> 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;
}
```

5.6 Longest Palindrome

- Input: string of length n
- Find the longest palindrome: $O(n)$ time and space
- Output: the longest palindrome

```
#define MAXN 1000000

int n, p[2 * MAXN + 1];

// get the starting index and the length of
    the longest palindrome
pair<int, int> LongestPalindrome(string & s) {
  n = s.size();
  int c = 1, r = 2;
  p[0] = 1; p[1] = 2;
  for (int i = 2; i < 2 * n + 1; i++) {
    if (i < r)
      p[i] = min(r - i, p[2 * c - i]);
    else
      p[i] = 0;
    while (i + p[i] < 2 * n + 1 && i - p[i] >=
         0 && ((i + p[i]) % 2 == 0 || s[(i + p
        [i]) / 2] == s[(i - p[i]) / 2]))
      p[i]++;
    if (i + p[i] > r)
      c = i, r = i + p[i];
  }
  int l = -1, s = 0;
  for (int i = 0; i < 2 * n + 1; i++) {
    p[i] /= 2;
    if (2 * p[i] - (i % 2) > s)
      s = 2 * p[i] - (i % 2), l = (i + 1) / 2
          - p[i];
  }
  return {l, s};
}
```

5.7 Eertree

```
struct eertree {
  string str;
  int cnt = 2, suf = 1;
  vector<int> lnk, len, dif, slnk;
  vector<array<int,26>> go;
  eertree(string& s, int n) : lnk(n+3), len(n
      +3), go(n+3), dif(n+3), slnk(n+3), str(s
      ) {
    len[1] = -1, len[2] = 0;
    lnk[1] = 1, lnk[2] = 1;
  }
  int walk(int i, int v) {
    while (i-1-len[v] < 0 || str[i-1-len[v]]
        != str[i])
      v = lnk[v];
    return v;
  }
  void add(int i) {
    int c = str[i]-'a', lst = walk(i, suf);
    if (!go[lst][c]) {
      go[lst][c] = ++cnt;
      len[cnt] = len[lst] + 2;
      lnk[cnt] = lst > 1 ? go[walk(i,lnk[lst])
          ][c] : 2;
      dif[cnt] = len[cnt] - len[lnk[cnt]];
      slnk[cnt] = dif[cnt] == dif[lnk[cnt]] ?
          slnk[lnk[cnt]] : lnk[cnt];
    }
    suf = go[lst][c];
  }
};
```

5.8 Suffix Automaton

```
struct SA {
  struct state {
    int len, link;
    map<char, int> next;
  };
  vector<state> st;
  int last = 0;

  SA() { st.push_back({0, -1}); }
  SA(string &s) : SA() {
    st.reserve(2*s.size());
    for (char c : s) append(c);
  }
  void append(char c) {
    int cur = st.size(), p = last;
    st.push_back({st[last].len + 1, 0});
    for (; p != -1 && !st[p].next.count(c); p
        = st[p].link)
      st[p].next[c] = cur;
    if (p != -1) {
      int q = st[p].next[c];
      if (st[p].len + 1 == st[q].len)
        st[cur].link = q;
      else {
        int clone = st.size();
        st.push_back({st[p].len + 1, st[q].
            link, st[q].next});
        st[q].link = st[cur].link = clone;
        for (; p != -1 && st[p].next[c] == q;
            p = st[p].link)
          st[p].next[c] = clone;
      }
    }
    last = cur;
  }
  bool contains(string &t) {
    int p = 0;
    for (char c : t) {
      if (!st[p].next.count(c))
        return false;
      p = st[p].next[c];
    }
    return true;
  }
};
string lcs(string &S, string &T) { //longest
    common substring
  SA s(S);
  int v = 0, l = 0, best = 0, bestpos = 0;
  for (size_t i = 0; i < T.size(); i++) {
    while (v && !s.st[v].next.count(T[i])) {
      v = s.st[v].link;
      l = s.st[v].len;
    }
    if (s.st[v].next.count(T[i]))
      v = s.st[v].next[T[i]], l++;
    if (l > best)
      best = l, bestpos = i;
  }
  return T.substr(bestpos - best + 1, best);
}
```

5.9 Aho-Corasick

```
struct AhoCorasick {
  #define ep emplace_back
  vector<array<int,26>> go;
  vector<int> fin, lnk;
  int cnt;
  AhoCorasick() : cnt(0) { add(); }
  AhoCorasick(const vector<string> &S) : cnt
      (0) {
    add();
    for (auto &s : S) {
      int cur = 0;
      for (auto c: s) {
        if (!go[cur][c-'a'])
          go[cur][c-'a'] = cnt, add(), lnk[cnt
              -1] = -1;
        cur = go[cur][c-'a'];
      }
      fin[cur]++;
    }
    lnk[0] = -1;
    for (queue<int> q({0}); !q.empty();) {
      int u = q.front(); q.pop();
      if (u) fin[u] += fin[lnk[u]];
      for (int i = 0; i < 26; i++) {
        int v = go[u][i];
        if (v) {
```

```
          lnk[v] = ~lnk[u] ? go[lnk[u]][i] :
              0;
          q.push(v);
        }
        if (u) go[u][i] = v ? v : go[lnk[u]][i
            ];
      }
    }
  }
  void add() {
    cnt++, go.ep(), fin.ep(), lnk.ep();
  }
  int query(const string &s) {
    int cur = 0, ans = 0;
    for (char c : s) {
      cur = go[cur][c-'a'];
      ans += fin[cur];
    }
    return ans;
  }
};
```

6 Geometry

6.1 2D Geometry

6.1.1 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): $s_a = \sqrt{bc\left[1 - \left(\dfrac{a}{b+c}\right)^2\right]}$
- 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: $\dfrac{a+b}{a-b} = \dfrac{\tan\frac{\alpha+\beta}{2}}{\tan\frac{\alpha-\beta}{2}}$

6.1.2 Quadrilaterals (Vierecke)

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

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

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

```
const long double inf = 1e100, eps = 1e-12, PI
    = acos(-1L);
struct point {
  long double x, y;
  bool operator<(point & b) {
    if(x != b.x)
      return x < b.x;
    return y < b.y;
  }
  bool operator==(point b) {
    return x == b.x && y == b.y;
  }
  point operator+(point b) {
    return {x + b.x, y + b.y};
  }
  point operator-(point b) {
    return {x - b.x, y - b.y};
  }
  point operator*(long double b) {
    return {x * b, y * b};
  }
  point operator/(long double b) {
    return {x / b, y / b};
  }
  long double length() {
    return sqrt(x * x + y * y);
  }
};
long double dot(point a, point b) {
  return a.x * b.x + a.y * b.y;
}
long double cross(point a, point b) {
  return a.x * b.y - a.y * b.x;
}
long double dist(point a, point b) {
  point d = a - b; return d.length();
}
long double angle(point a, point b) {
  return acos(dot(a, b) / a.length() / b.
      length());
}
bool collinear(point a, point b) {
  return fabs(cross(a, b)) < eps;
}
bool collinear(point p, point a, point b) {
  return collinear(a - p, b - p);
}
bool ccw(point a, point b) {
  return cross(a, b) > 0;
}
bool ccw(point p, point a, point b) {
  return ccw(a - p, b - p);
}
point rotateCCW90(point p) {
  return { -p.y, p.x};
}
point rotateCW90(point p) {
  return {p.y, -p.x};
}
point rotateCCW(point p, double t) {
  return {p.x * cos(t) - p.y * sin(t), p.x *
      sin(t) + p.y * cos(t)};
}
point projectPointLine(point p, point a, point
    b) {
  long double r = dot(b - a, b - a);
  if(fabs(r) < eps) return a;
  r = dot(p - a, b - a) / r;
  return a + (b - a) * r;
}
long double distancePointLine(point p, point a
    , point b) {
  return dist(p, projectPointLine(a, b, p));
}
point projectPointSegment(point p, point a,
    point b) {
  long double r = dot(b - a, b - a);
  if(fabs(r) < eps) return a;
  r = dot(p - a, b - a) / r;
  if(r < 0) return a;
  if(r > 1) return b;
  return a + (b - a) * r;
}
long double distancePointSegment(point p,
    point a, point b) {
  return dist(p, projectPointSegment(p, a, b))
      ;
}
bool linesParallel(point a, point b, point c,
    point d) {
  return fabs(cross(b - a, c - d)) < eps;
}
bool linesCollinear(point a, point b, point c,
     point d) {
  return linesParallel(a, b, c, d) && fabs(
      cross(a - b, a - c)) < eps && fabs(cross
      (c - d, c - a)) < eps;
}
bool segmentsIntersect(point a, point b, point
     c, point d) {
  if(linesCollinear(a, b, c, d)) {
    if(dist(a, c) < eps || dist(a, d) < eps ||
         dist(b, c) < eps || dist(b, d) < eps)
        return true;
    if(dot(c - a, c - b) > 0 && dot(d - a, d -
         b) > 0 && dot(c - b, d - b) > 0)
        return false;
    return true;
  }
  if(cross(d - a, b - a) * cross(c - a, b - a)
       > 0) return false;
  if(cross(a - c, d - c) * cross(b - c, d - c)
       > 0) return false;
  return true;
}
// Lines -a--b- and -c--d- can't be collinear
```

```
point computeLineIntersection(point a, point b
    , point c, point d) {
  b = b - a, d = c - d, c = c - a;
  if(dot(b, b) < eps || dot(d, d) < eps)
    return a;
  return a + b * cross(c, d) / cross(b, d);
}
point computeCircleCenter(point a, point b,
    point c) {
  b = (a + b) / 2;
  c = (a + c) / 2;
  return computeLineIntersection(b, b +
      rotateCW90(a - b), c, c + rotateCW90(a -
       c));
}
bool pointInPolygon(vector<point> & p, point q
    ) {
  bool c = false;
  for(int i = 0; i < p.size(); i++) {
    int j = (i + 1) % p.size();
    if((p[i].y <= q.y && q.y < p[j].y || p[j].
        y <= q.y && q.y < p[i].y) &&
        q.x < p[i].x + (p[j].x - p[i].x) * (q.
            y - p[i].y) / (p[j].y - p[i].y))
      c = !c;
  }
  return c;
}
bool pointOnPolygon(vector<point> & p, point q
    ) {
  for(int i = 0; i < p.size(); i++)
    if(dist(projectPointSegment(p[i], p[(i +
        1) % p.size()], q), q) < eps)
      return true;
  return false;
}
vector<point> circleLineIntersection(point a,
    point b, point c, double r) {
  vector<point> ret;
  b = b - a;
  a = a - c;
  double A = dot(b, b);
  double B = dot(a, b);
  double C = dot(a, a) - r * r;
  double D = B * B - A * C;
  if(D < -eps) return ret;
  ret.push_back(c + a + b * (-B + sqrt(D + eps
      )) / A);
  if(D > eps) ret.push_back(c + a + b * (-B -
      sqrt(D)) / A);
  return ret;
}
vector<point> circleCircleIntersection(point a
    , point b, double r, double R) {
  vector<point> ret;
  double d = dist(a, b);
  if(d > r + R || d + min(r, R) < max(r, R))
    return ret;
  long double x = (d * d - R * R + r * r) / (2
      * d);
  long double y = sqrt(r * r - x * x);
  point v = (b - a) / d;
  ret.push_back(a + v * x + rotateCCW90(v) * y
      );
  if(y > 0) ret.push_back(a + v * x -
      rotateCCW90(v) * y);
  return ret;
}
long double computeSignedArea(vector<point> &
    p) {
  long double area = 0;
  for(int i = 0; i < p.size(); i++) {
    int j = (i + 1) % p.size();
    area += p[i].x * p[j].y - p[j].x * p[i].y;
  }
  return area / 2.0;
}
long double computeArea(vector<point> & p) {
  return fabs(computeSignedArea(p));
}
point computeCentroid(vector<point> & p) {
  point c = {0, 0};
  double scale = 6.0 * computeSignedArea(p);
  for(int i = 0; i < p.size(); i++) {
    int j = (i + 1) % p.size();
    c = c + (p[i] + p[j]) * (p[i].x * p[j].y -
        p[j].x * p[i].y);
  }
  return c / scale ;
}
bool isSimple(vector<point> & p) {
  for(int i = 0; i < p.size(); i++) {
    for(int k = i + 1; k < p.size(); k++) {
      int j = (i + 1) % p.size();
      int l = (k + 1) % p.size();
      if(i == l || j == k) continue;
      if(segmentsIntersect(p[i], p[j], p[k], p
          [l])) return false;
    }
  }
  return true;
}
int n; point p[100000];
point s = {1000000000, 1000000000};
bool comp(point & a, point & b) {
  if(a == s) return true;
  if(b == s) return false;
  if(collinear(s, a, b))
    return dist(a, s) < dist(b, s);
  return ccw(s, a, b);
}
int main() {
  cin >> n;
  // no duplicates in p
  for(int i = 0; i < n; i++) {
    cin >> p[i].x >> p[i].y;
    if(p[i] < s) s = p[i];
  }
  sort(p, p + n, comp);
  p[n] = s;
  vector<int> res;
  res.push_back(0);
  res.push_back(1);
  for(int i = 2; i <= n; i++) {
    while(res.size() >= 2 && ccw(p[res[res.
        size() - 2]], p[i], p[res[res.size() -
         1]]))
      res.pop_back();
    if(i != n)
      res.push_back(i);
  }
  //the convex hull
  for(int i : res)
    //(p[i].x , p[i])
}
```

6.2 Nearest pair of points

```
double nearestPairOfPoints(vector<point>& a) {
  int n = a.size(), l = 0;
  set<point> cur;
  double ans = 1e18;
  sort(a.begin(), a.end());
  for (int i = 0; i < n; i++) {
    while (a[l].x < a[i].x && (a[l].x - a[i].x
        ) * (a[l].x - a[i].x) > ans)
      cur.erase(a[l++]);
    auto lo = cur.lower_bound({a[i].x - sqrt(
        ans) - eps, a[i].y});
    auto hi = cur.upper_bound({a[i].x + sqrt(
        ans) + eps, a[i].y});
    while (lo != hi) {
      ans = min(ans, dist(*lo, a[i]) * dist(*
          lo, a[i]));
      lo = next(lo);
    }
  }
  return ans;
}
```

6.3 Half-plane Intersection

- Input: A set of n halfplanes, given by two points a and b on the boundary. The halfplane is to the left of the line connecting a and b.
- Output: The intersection forms a convex polygon, whose vertices are returned.
- Complexity: $\mathcal{O}(n \log n)$.
- TODO: Implement a point struct pt.

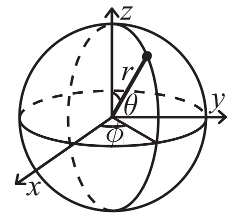
```
struct plane {
  pt p, d;
  ld phi;
  plane() {}
  plane(pt a, pt b) : p(a), d(b - a) {
```

```
    phi = atan2(d.y, d.x);
  }
  bool out(pt x) const {
    return cross(d, x - p) < -eps;
  }
  bool operator<(const plane& x) const {
    return abs(phi - x.phi) < eps ? out(x.p) :
         phi < x.phi;
  }
  bool operator==(const plane& x) const {
    return abs(phi - x.phi) < eps;
  }
  friend pt sect(const plane& x, const plane&
     y) {
    return x.p + x.d * (cross(y.p - x.p, y.d)
        / cross(x.d, y.d));
  }
};
vector<pt> solve(vector<plane> a) {
  a.pb(plane(pt(inf, inf), pt(-inf, inf)));
  a.pb(plane(pt(-inf, inf), pt(-inf, -inf)));
  a.pb(plane(pt(-inf, -inf), pt(inf, -inf)));
  a.pb(plane(pt(inf, -inf), pt(inf, inf)));
  sort(all(a));
  a.erase(unique(all(a)), a.end());
  int n = sz(a);
  deque<plane> pq;
  for (int i = 0; i < n; i++) {
    while (sz(pq) > 1 && a[i].out(sect(pq[sz(
        pq)-1], pq[sz(pq)-2])))
      pq.pop_back();
    while (sz(pq) > 1 && a[i].out(sect(pq[0],
        pq[1])))
      pq.pop_front();
    pq.push_back(a[i]);
  }
  while (sz(pq) > 2 && pq[0].out(sect(pq[sz(pq
      )-1], pq[sz(pq)-2])))
    pq.pop_back();
  while (sz(pq) > 2 && pq.back().out(sect(pq
      [0], pq[1])))
    pq.pop_front();
  if (sz(pq) < 3)
    return {};
  vector<pt> ret(sz(pq));
  for (int i = 0; i < sz(pq); i++)
    ret[i] = sect(pq[i], pq[(i+1) % sz(pq)]);
  return ret;
}
```

6.4 3D Geometry

6.4.1 Spherical coordinates

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



6.4.2 Spherical Distance

Returns the shortest distance on the sphere with radius radius between the points with azimuthal angles (longitude) f1 (ϕ_1) and f2 (ϕ_2) from x axis and zenith angles (latitude) t1 (θ_1) and t2 (θ_2) from z axis. All angles measured in radians. The algorithm starts by converting the spherical coordinates to cartesian coordinates so if that is what you have you can use only the two last rows. dx*radius is then the difference between the two points in the x direction and d*radius is the total distance between the points.

```
double sphericalDistance(double f1, double t1,
    double f2, double t2, double radius) {
  double dx = sin(t2)*cos(f2) - sin(t1)*cos(f1
      );
  double dy = sin(t2)*sin(f2) - sin(t1)*sin(f1
      );
  double dz = cos(t2) - cos(t1);
  double d = sqrt(dx*dx + dy*dy + dz*dz);
  return radius*2*asin(d/2);
}
```

6.4.3 Point

```
template<class T> struct Point3D {
  typedef Point3D P;
  typedef const P& R;
  T x, y, z;
  explicit Point3D(T x=0, T y=0, T z=0) : x(x)
      , y(y), z(z) {}
  bool operator<(R p) const {
    return tie(x, y, z) < tie(p.x, p.y, p.z);
        }
  bool operator==(R p) const {
    return tie(x, y, z) == tie(p.x, p.y, p.z);
         }
  P operator+(R p) const { return P(x+p.x, y+p
      .y, z+p.z); }
  P operator-(R p) const { return P(x-p.x, y-p
      .y, z-p.z); }
  P operator*(T d) const { return P(x*d, y*d,
      z*d); }
  P operator/(T d) const { return P(x/d, y/d,
      z/d); }
  T dot(R p) const { return x*p.x + y*p.y + z*
      p.z; }
  P cross(R p) const {
    return P(y*p.z - z*p.y, z*p.x - x*p.z, x*p
        .y - y*p.x);
  }
  T dist2() const { return x*x + y*y + z*z; }
  double dist() const { return sqrt((double)
      dist2()); }
  //Azimuthal angle (longitude) to x-axis in
      interval [-pi, pi]
  double phi() const { return atan2(y, x); }
  //Zenith angle (latitude) to the z-axis in
      interval [0, pi]
  double theta() const { return atan2(sqrt(x*x
      +y*y),z); }
  P unit() const { return *this/(T)dist(); }
      //makes dist()=1
  //returns unit vector normal to *this and p
  P normal(P p) const { return cross(p).unit()
      ; }
  //returns point rotated 'angle' radians ccw
      around axis
  P rotate(double angle, P axis) const {
    double s = sin(angle), c = cos(angle); P u
        = axis.unit();
    return u*dot(u)*(1-c) + (*this)*c - cross(
        u)*s;
  }
};
```

6.4.4 Convex Hull

- Computes the faces of the convex hull spanned by the points in A.
- Requirement: no four points must be coplanar!
- All faces will face outwards.
- Runtime $O\left(n^2\right)$.

```
#define rep(i,a,b) for (int i = (a); i < (b);
    ++i)

typedef Point3D<double> P3;

struct PR {
  void ins(int x) { (a == -1 ? a : b) = x; }
  void rem(int x) { (a == x ? a : b) = -1; }
  int cnt() { return (a != -1) + (b != -1); }
  int a, b;
};

struct F { P3 q; int a, b, c; }; // Direction
    and indices of involved vertices

vector<F> hull3d(const vector<P3>& A) {
  assert(A.size() >= 4);
  vector<vector<PR>> E(A.size(), vector<PR>(A.
      size(), {-1, -1}));
#define E(x,y) E[f.x][f.y]
  vector<F> FS;
  auto mf = [&](int i, int j, int k, int l) {
```

```
    P3 q = (A[j] - A[i]).cross((A[k] - A[i]));
    if (q.dot(A[l]) > q.dot(A[i]))
      q = q * -1;
    F f{q, i, j, k};
    E(a,b).ins(k); E(a,c).ins(j); E(b,c).ins(i
        );
    FS.push_back(f);
  };
  rep(i,0,4) rep(j,i+1,4) rep(k,j+1,4)
    mf(i, j, k, 6 - i - j - k);

  rep(i,4,(int)A.size()) {
    rep(j,0,(int)FS.size()) {
      F f = FS[j];
      if(f.q.dot(A[i]) > f.q.dot(A[f.a])) {
        E(a,b).rem(f.c);
        E(a,c).rem(f.b);
        E(b,c).rem(f.a);
        swap(FS[j--], FS.back());
        FS.pop_back();
      }
    }
    int nw = FS.size();
    rep(j,0,nw) {
      F f = FS[j];
#define C(a, b, c) if (E(a,b).cnt() != 2) mf(f
    .a, f.b, i, f.c);
      C(a, b, c); C(a, c, b); C(b, c, a);
    }
  }
  for (auto it : FS) if ((A[it.b] - A[it.a]).
      cross(
    A[it.c] - A[it.a]).dot(it.q) <= 0) swap(it
        .c, it.b);
  return FS;
}
```

6.4.5 Volume of Polyhedron

- `p` should be a list of the vertices and `trilist` a list of the triangular faces (facing outwards) of the polyhedron.

```
template<class V, class L>
double signed_poly_volume(const V& p, const L&
     trilist) {
  double v = 0;
  for (auto i : trilist) v += p[i.a].cross(p[i
      .b]).dot(p[i.c]);
  return v / 6;
}
```

7 Mathematics

7.1 Theorems

7.1.1 Fibonacci numbers

- Definition:
  - $f_0 = 0,\ f_1 = 1,\ f_i = f_{i-1} + f_{i-2}$
- Calculation:
  - Dynamic programming: $O(n)$
  - Fast matrix exponentiation: $O(\log n)$
  - $\begin{bmatrix}1 & 1\\ 1 & 0\end{bmatrix}^n = \begin{bmatrix}f_{n+1} & f_n\\ f_n & f_{n-1}\end{bmatrix}$
- $\sum_{k=0}^{n} \binom{n-k}{k} = F_{n+1}$
- Generating function $f(z) = \frac{1}{1-z-z^2}$

7.1.2 Series Formulas

$$\sum_{k=0}^{n} k = \frac{n(n+1)}{2} \qquad \sum_{k=0}^{n} k^4 = \frac{6n^5+15n^4+10n^3-n}{30}$$
$$\sum_{k=a}^{b} k = \frac{(a+b)(b-a+1)}{2} \qquad \sum_{k=0}^{n} k^5 = \frac{2n^6+6n^5+5n^4-n^2}{12}$$
$$\sum_{k=0}^{n} k^2 = \frac{n(n+1)(2n+1)}{6} \qquad \sum_{k=0}^{n} x^k = \frac{x^{n+1}-1}{x-1}$$
$$\sum_{k=0}^{n} k^3 = \frac{n^2(n+1)^2}{4} \qquad \sum_{k=0}^{n} kx^k = \frac{x-(n+1)x^{n+1}+nx^{n+2}}{(x-1)^2}$$

Geometric series:

- $\sum_{i=0}^{n} c^i = \frac{c^{n+1}-1}{c-1}$ for $c \neq 1$
- $\sum_{i=0}^{\infty} c^i = \frac{1}{1-c}$ and $\sum_{i=1}^{\infty} c^i = \frac{c}{1-c}$ for $|c| < 1$

7.1.3 Binomial coefficients

- Definition:
  - $\binom{n}{k} = \frac{n!}{k!(n-k)!}$
  - $\binom{n}{k} = \binom{n-1}{k-1} + \binom{n-1}{k}$

$$\sum_{m=k}^{n} \binom{m}{k} = \binom{n+1}{k+1} \qquad \sum_{k=0}^{m} \binom{n+k}{k} = \binom{n+m+1}{m}$$
$$\sum_{k=0}^{n} \binom{n}{k}^2 = \binom{2n}{n} \qquad \sum_{k=1}^{n} k\binom{n}{k} = n2^{n-1}$$
$$\sum_{k=1}^{n} k^2\binom{n}{k} = (n+n^2)2^{n-2} \qquad \sum_{j=0}^{k} \binom{m}{j}\binom{n}{k-j} = \binom{n+m}{k}$$
$$\sum_{m=0}^{n} \binom{m}{j}\binom{n-m}{k} = \binom{n+1}{k+j+1} \qquad \sum_{j=0}^{k} (-1)^j\binom{n}{j} = (-1)^k\binom{n-1}{k}$$
$$\sum_{k=q}^{n} \binom{n}{k}\binom{k}{q} = 2^{n-q}\binom{n}{q}$$
$$\sum_{k=-a}^{a} (-1)^k \binom{a+b}{a+k}\binom{b+c}{b+k}\binom{c+a}{c+k} = \frac{(a+b+c)!}{a!b!c!}$$

- There are exactly $\binom{|A|+|B|}{|A|}$ ways to select sets $A' \subseteq A$ and $B' \subseteq B$ such that $|A'| = |B'|$ (proof sketch: choose those not selected in A and those selected in B):
$$\sum_{k=0}^{\min(n,m)} \binom{n}{k}\binom{m}{k} = \binom{n+m}{n}$$

7.1.4 Catalan's number

- $1, 1, 2, 5, 14, 42, 132, 429, 1430, \ldots$
- $c_n = \sum_{k=0}^{n-1} c_k\, c_{n-1-k} = \frac{1}{n+1}\binom{2n}{n} = \binom{2n}{n} - \binom{2n}{n-1}$
- Number of correct bracket sequences consisting of n opening and n closing brackets.
- Generating function $f(x) = \frac{1-\sqrt{1-4x}}{2x}$

7.1.5 Pentagonal Number theorem

- $\prod_n (1-x^n) = \sum_k (-1)^k x^{k(3k-1)/2}$
- $\sum_n p(n)x^n = \prod_n (1-x^n)^{-1}$ where p is the partition function.

7.1.6 Hook Length formula

- Number of Young diagrams (filling of the cells with integers from $\{1, \cdots, n\}$ without repetitions) with shape $\lambda = (\lambda_1 \geq \cdots \geq \lambda_k)$ and $n = \sum_i \lambda_i$: $f^\lambda = \frac{n!}{\prod h_\lambda(i,j)}$ where $h_\lambda(i,j)$ is the hook length of cell (i,j). (number of cells below / right)

7.1.7 Pick's theorem

$I = A - B/2 + 1$, where A is the area of a lattice polygon, I is number of lattice points inside it, and B is number of lattice points on the boundary. Number of lattice points minus one on a line segment from $(0,0)$ and (x,y) is $gcd(x,y)$.

7.1.8 Burnside's Lemma

- $ClassesCount = \frac{1}{|G|}\sum_{g \in G} |X^g|$
- G: group of operations(invariant permutations)
- X^g: set of fixed points for operation g, i.e. $X^g = \{x \in X : g.x = x\}$
- special case: $ClassesCount = \frac{1}{|G|}\sum_{g \in G} k^{c(g)}$
- k: "number of colors"
- $c(g)$: number of cycles in permutation

7.1.9 Multinomial coefficients

$$(x_1 + \ldots + x_m)^n = \sum_{k_1+\ldots+k_m=n, k_i \geq 0} \binom{n}{k_1,\ldots,k_m} x_1^{k_1} \ldots x_m^{k_m},$$

where $\binom{n}{k_1,\ldots,k_m} = \frac{n!}{k_1!k_2!\ldots k_m!}$

in combinatorial sense $\binom{n}{k_1,\ldots,k_m}$ is equal to the number of ways of depositing n distinct objects into m distinct bins, with k_1 objects in the first bin, k_2 objects in the second bin ...

7.1.10 Gray's code

direct: $G(n) = n \oplus (n >> 1)$
recurrent: $G(n) = 0G(n-1) \cup 1G(n-1)^R$ and $G(n)^R = 1G(n-1) \cup 0G(n-1)^R$

7.2 Game theory

7.2.1 Grundy's function

For all transitions $v- > v_i$ compute the Grundy's function $f(v_i)$:
1. $v- > v_i$ transtion into one game, then compute $f(v_i)$ recursively
$v- > v_i$ transition into sum of several games, compute f for each game and take $\oplus$ sum of their values
2. $f(v) = mex\{f(v_1), \ldots, f(v_k)\}$ (mex returns minimal number not contained in the set)

7.3 2-SAT

2-SAT on n variables $0, \ldots n-1$. Negated variables are represented by bit-inversion (`~x`). Usage:

```
TwoSat ts(n);
ts.either(0, ~3) // var 0 true or var 3 false
ts.set_value(2); // var 2 true
ts.at_most_one({0,~1,2});
```

```
ts.solves() // true iff solveable
ts.values[0..n-1] holds the assignment
```

Complexity: $O(N+E)$ for N variables and E clauses.

```
#define rep(i, a, b) for(int i = a; i < (b);
    ++i)
#define sz(x) (int)(x).size()
typedef vector<int> vi;

struct TwoSat {
  int N;
  vector<vi> gr;
  vi values; // 0 = false, 1 = true

  TwoSat(int n = 0) : N(n), gr(2*n) {}

  int add_var() { // (optional)
    gr.emplace_back();
    gr.emplace_back();
    return N++;
  }
  void either(int f, int j) {
    f = max(2*f, -1-2*f);
    j = max(2*j, -1-2*j);
    gr[f].push_back(j^1);
    gr[j].push_back(f^1);
  }
  void set_value(int x) { either(x, x); }

  void at_most_one(const vi& li) { // (
      optional)
    if (sz(li) <= 1) return;
    int cur = ~li[0];
    rep(i,2,sz(li)) {
      int next = add_var();
      either(cur, ~li[i]);
      either(cur, next);
      either(~li[i], next);
      cur = ~next;
    }
    either(cur, ~li[1]);
  }

  vi val, comp, z; int time = 0;
  int dfs(int i) {
    int low = val[i] = ++time, x; z.push_back(
        i);
    for (auto &e : gr[i]) if (!comp[e])
      low = min(low, val[e] ?: dfs(e));
    ++time;
    if (low == val[i]) do {
      x = z.back(); z.pop_back();
      comp[x] = time;
      if (values[x>>1] == -1)
        values[x>>1] = x&1;
    } while (x != i);
    return val[i] = low;
  }

  bool solve() {
    values.assign(N, -1);
    val.assign(2*N, 0); comp = val;
    rep(i,0,2*N) if (!comp[i]) dfs(i);
    rep(i,0,N) if (comp[2*i] == comp[2*i+1])
        return 0;
    return 1;
  }
};
```

7.4 Lattice Points below a line

- Returns $\sum_{i=1}^{D} \left\lfloor \frac{A+Bi}{C} \right\rfloor$, which is the number of integer points below the line $\frac{A}{C} + \frac{B}{C} \cdot x$.
- Requires: $A, B, D \geq 0, C > 0$.
- Complexity: $\mathcal{O}(\log D)$

```
ll prog(ll a, ll b, ll c) {
  return c * a + b * c * (c + 1) / 2;
}
ll sum(ll a, ll b, ll c, ll d) {
  if ((a + b * d) / c == 0 || d < 1)
    return 0;
  ll r = prog(a < 0 ? (a - c + 1) / c : a / c,
      b / c, d);
  if (b % c != 0) {
    a = (a % c + c) % c, b = (b % c + c) % c;
    r += sum((a + b * d) % c + b - c, c, b, (a
        + b * d) / c);
  }
  return r;
}
```

7.5 Prime numbers

Selected prime numbers:

- The first 25: 2, 3, 5, 7, 11, 13, 17, 19, 23, 29, 31, 37, 41, 43, 47, 53, 59, 61, 67, 71, 73, 79, 83, 89, 97
- ≥ 100: 101, 211 (47th), 307 (63th), 503 (96th), 997
- $\geq 10^3$: 1009 (169th), 2003, 3001, 4001, 5003, 7001
- $\geq 10^4$: 10,007 (1230th), 10,009, 20,011, 50,021
- $\geq 10^5$: 100,003 (9593th), 100,019, 200,003, 500,009
- $\geq 10^6$: 1,000,003 (78,499th), 2,000,003, 5,000,011
- $\geq 10^7$: $10^7 + 19$ (664,580th), 20,000,003, 50,000,017
- $\geq 10^8$: $10^8 + 7$ (5,761,456th), 200,000,033, 500,000,003
- $\geq 10^9$: $10^9 + 7$ (50,847,535th), $10^9 + 9$, $10^{10} + 19$
- All Fermat primes (of form $2^{2^n} + 1$): 3, 5, 17, 257, 65,537

The enumeration algorithm:

- For a given n `sieve(n)` runs in $O(n)$ time and space
- For a given n `moebius(n)` runs in $O(n \log n)$ time and $O(n)$ space
- For $x \leq n$ `factorise(x))` then returns the prime factorisation of x in $O(\log x)$ time.
- `phi[x]` denotes the Euler phi function of x
- Useful identities:
  - $\sum_{d|n} \phi(d) = n$
  - $i \star \mu = \mu \star i = \epsilon$ or alternatively $g \star i = f \Leftrightarrow g = f \star \mu$
  - f, g are multiplicative $\Rightarrow f \cdot g, f \star g$ are multiplicative
  - $\left(\frac{a}{p}\right) = a^{\frac{p-1}{2}} \equiv \pm 1 \mod p$ for some prime $p > 2$. a is a quadratic residue if and only if $\left(\frac{a}{p}\right) = 1$
  - $\left(\frac{q}{p}\right) = \left(\frac{p}{q}\right) \cdot \begin{cases} +1 & p \equiv 1 \mod 4 \text{ or } q \equiv 1 \mod 4 \\ -1 & p \equiv q \equiv 3 \mod 4 \end{cases}$

```
int n, phi[N], lp[N], mu[N];
vector<int> p;
//x = [0].first^[0].second * ...
vector<pair<int, int>> factorise(int x) {
  vector<pair<int, int>> d;
  int y = lp[x], a = 1;
  x /= lp[x];
  while (x > 1) {
    if (lp[x] != y) {
      d.push_back({y, a});
      y = lp[x]; a = 0;
    }
    x /= lp[x], a++;
  }
  d.push_back({y, a});
  return d;
}
void sieve(int n) {
  phi[1] = 1;
  for (int i = 2; i <= n; i++) {
    if (lp[i] == 0) {
      lp[i] = i; phi[i] = i - 1;
      p.push_back(i);
    } else if (lp[i] == lp[i / lp[i]])
      phi[i] = phi[i / lp[i]] * lp[i];
    else
      phi[i] = phi[i / lp[i]] * (lp[i] - 1);
    for (int j = 0; j < (int) p.size() && p[j]
        <= lp[i] && i * p[j] <= n; j++)
      lp[i * p[j]] = p[j];
  }
}
void moebius(int n) {
  mu[1] = -1;
  for (int i = 1; i <= n; i++) {
    mu[i] *= -1;
    for (int j = 2*i; j <= n; j += i)
      mu[j] += mu[i];
  }
}
```

7.6 Algebra Basics

```
typedef long long ll;
typedef pair<ll, ll> PLL;

// return a % b (positive value)
ll mod(ll a, ll b) {
  return ((a % b) + b) % b;
}
```

```
// return a^b mod m
ll powmod(ll a, ll b, ll m) {
  ll res = 1;
  while(b > 0)
    if(b & 1) res = (res * a) % m, --b;
    else a = (a * a) % m, b >>= 1;
  return res % m;
}

// computes gcd(a,b)
ll gcd(ll a, ll b) {
  ll tmp;
  while(b) {a %= b; swap(a, b); }
  return a;
}

// computes lcm(a,b)
ll lcm(ll a, ll b) {
  return a / gcd(a, b) * b;
}

// returns d = gcd(a,b); finds x,y such that d
    = ax + by
int extended_euclid(int a, int b, int &x, int
    &y) {
  int xx = y = 0; int yy = x = 1;
  while (b) {
    int q = a / b; int t = b; b = a % b;
    a = t; t = xx; xx = x - q * xx; x = t;
    t = yy; yy = y - q * yy; y = t;
  }
  return a;
}

// finds all solutions to ax = b (mod n)
VI modular_linear_equation_solver(int a, int b
    , int n) {
  int x, y; VI solutions;
  int d = extended_euclid(a, n, x, y);
  if(!(b % d)) {
    x = mod(x * (b / d), n);
    for(int i = 0; i < d; i++)
      solutions.push_back(mod(x + i * (n / d),
          n));
  }
  return solutions;
}

// computes b such that ab = 1 (mod n),
    returns -1 on failure
int mod_inverse(int a, int n) {
  int x, y;
  int d = extended_euclid(a, n, x, y);
  if(d > 1) return -1;
  return mod(x, n);
}

// Chinese remainder theorem (special case):
    find z such that
// z % x = a, z % y = b.  Here, z is unique
    modulo M = lcm(x,y).
// Return (z,M).  On failure, M = -1.
PII chinese_remainder_theorem(int x, int a,
    int y, int b) {
  int s, t;
  int d = extended_euclid(x, y, s, t);
  if(a % d != b % d) return make_pair(0, -1);
  return make_pair(mod(s * b * x + t * a * y,
      x * y) / d, x * y / d);
}

// Chinese remainder theorem: find z such that
// z % x[i] = a[i] for all i.  Note that the
    solution is
// unique modulo M = lcm_i (x[i]).  Return (z,
    M).  On
// failure, M = -1.  Note that we do not
    require the a[i]'s
// to be relatively prime.
PII chinese_remainder_theorem(const VI &x,
    const VI &a) {
  PII ret = make_pair(a[0], x[0]);
  for(int i = 1; i < x.size(); i++) {
    ret = chinese_remainder_theorem(ret.second
        , ret.first, x[i], a[i]);
    if(ret.second == -1) break;
  }
  return ret;
}

// computes x and y such that ax + by = c; on
    failure, x = y =-1
void linear_diophantine(int a, int b, int c,
    int &x, int &y) {
  int d = gcd(a, b);
  if(c % d) x = y = -1;
  else {
    x = c / d * mod_inverse(a / d, b / d);
    y = (c - a * x) / b;
  }
}
```

7.7 Modular Inverse

- Precomputes *all* modular multiplicative inverse elements mod `mod` up to `n` in $O(n)$ time.

```
int inv[MAX];
void precompute_inverse(int n, int mod) {
  inv[1]=1;
  for (int i=2; i<=n; i++)
    inv[i] = (mod - (mod/i)*1LL*inv[mod%i] %
        mod) % mod;
}
```

7.8 Euler's Totient Function and Theorem

- $\phi(n)$ counts the number of integers between 1 and n inclusive, which are coprime to n
- Let $n = p_1^{a_1} \cdot \dots \cdot p_k^{a_k}$ be the prime decomposition of n, then $\phi(n) = \prod_{i=1}^{k} p_i^{a_i - 1}(p_i - 1)$
- if $gcd(a, m) = 1$, then $a^{\phi(m)} \equiv 1 \mod m$ and thus, $a^b \equiv a^{b \mod \phi(m)} \mod m$
- For arbitrary integers a, m and $b \geq \log_2 m$ it holds that $a^b \equiv a^{\phi(m) + [b \bmod \phi(m)]} \bmod m$

```
//O(sqrt(n))
//calculation for all n <= N in O(N) time
//see "Prime numbers" section
int euler_phi(int n) {
  int result = n;
  for(int i = 2; i * i <= n; i++) {
    if(n % i == 0) {
      while(n % i == 0) n /= i;
      result -= result / i;
    }
  }
  if(n > 1) result -= result / n;
  return result;
}
```

7.9 Discrete Logarithm: Baby Gigant

- Return x such that $a^x \equiv b \mod m$, or -1 otherwise.
- Let $n = \lfloor \sqrt{m} \rfloor$ and $x = np - q$. The algorithm stores all a^{np} and checks all $ba^q \Rightarrow$ Runtime $O\left(\sqrt{m} \log m\right)$.

```
int dlog(int a, int b, int m) {
  int n = sqrt((double)m) + 1;
  map<int, int> vals;
  for(int i = n; i >= 1; --i)
    vals[powmod(a, i * n, m)] = i;
  for(int i = 0; i <= n; ++i) {
    int cur = (powmod(a, i, m) * b) % m;
    if(vals.count(cur)) {
      int ans = vals[cur] * n - i;
      if(ans < m) return ans;
    }
  }
  return -1;
}
```

7.10 Discrete Root

- Return x such that $x^k \equiv a \mod m$.
- Let g be a primitive root modulo m. Then $g^y \equiv x \mod m$ for some y and hence, $\left(g^k\right)^y \equiv (g^y)^k \equiv x^k \equiv a \mod m$. Thus, it is sufficient to compute $y = \mathrm{dlog}_{(g^k)}\, a$ in order to find $x \equiv g^y \mod m$. The discrete log takes time $O(m \log m)$.
- A solution exists iff the discrete log exists. In this case, all solutions are of the form $x = g^{y + i\frac{\phi(n)}{\gcd(k, \phi(n))}}$ for some integer i.

7.11 Primitive Root (Generator)

- g is a primitive root modulo m if, for every a coprime to m, there exists some k such that $g^k \equiv a \mod m$.
- A primitive root modulo m exists iff $m \in \{1, 2, 4\}$ or $m = p^k$ or $m = 2 \cdot p^k$ for some prime $p \neq 2$ and $k \geq 1$. In this case, the number of primitive roots is $\phi(\phi(m))$.
- Runtime: $\mathcal{O}(\sqrt{m} + x \cdot \log^2(m))$, where x is the number of iterations until a root is found. In practice that should only be a couple of iterations.

```
int primitive_root(int m) {
  int phi = euler_phi(m); // m-1 if m prime
  int n = phi;
  vector<int> fact;
  for(int i = 2; i * i <= n; ++i)
    if(n % i == 0) {
      fact.push_back(i);
      while(n % i == 0) n /= i;
    }
  if(n > 1) fact.push_back(n);

  for(int res = 2; res < m; ++res) {
    // skip next line if m is prime
    if(gcd(res, m) != 1) continue;
    bool ok = true;
    for(int f : fact)
      if(powmod(res, phi / f, m) == 1) {
        ok = false; break; }
    if(ok) return res;
  }
  return -1; // no root exists
}
```

7.12 Rabin Miller

- Deterministic primality test for $n < 2^{32}$
- For $n < 2^{64}$ replace primes (2, 7, 61) with 2, 3, 5, 7, 11, 13, 17, 19, 23, 29, 31 and 37.
- For $n < 341 \cdot 10^{12}$ use all primes up to 17.

```
bool Miller(LL p, LL s, int a) {
  if(p == a) return 1;
  LL mod = expmod(a, s, p); //a^s
  for(; s - p + 1 && mod - 1 && mod - p + 1; s
      *= 2) mod = mulmod(mod, mod, p);  //mod
      ^2
  return mod == p - 1 || s % 2;
}
bool isprime(LL n) {
  if(n < 2) return 0;
  if(n % 2 == 0) return n == 2;
  LL s = n - 1;
  while(s % 2 == 0) s /= 2;
  return Miller(n, s, 2) && Miller(n, s, 7) &&
      Miller(n, s, 61);
}
```

7.13 Pollard Rho

- Expected time: $\mathcal{O}(N^{1/4})$

```
typedef unsigned long long ul;
ul pollard(ul n) { // return some nontrivial
    factor of n
  auto f = [n](ul x) { return x * x % n + 1;
      };
  ul x = 0, y = 0, t = 30, prd = 2, i = 1, q;
  while (t++ % 40 || gcd(prd, n) == 1) {
    if (x == y) x = ++i, y = f(x);
    if ((q = prd * (max(x,y)-min(x,y)) % n))
        prd = q;
    x = f(x), y = f(f(y));
  }
  return gcd(prd, n);
}
void factor_rec(ul n, map<ul,int>& cnt) {
  if (n == 1) return;
  if (is_prime(n)) { ++cnt[n]; return; }
  ul u = pollard(n);
  factor_rec(u,cnt), factor_rec(n/u,cnt);
}
vector<pair<ul,int>> factor(ul n) {
  map<ul,int> cnt; factor_rec(n,cnt);
  return vector<pair<ul,int>>(all(cnt));
}
```

7.14 Fast Fourier Transformation

7.14.1 Non-Recursive Fast Fourier Transformation

```
//for extra speed use follwoing custom complex
    type
struct cpx {
  double a=0,b=0;
  cpx(){}
  cpx(double a):a(a){}
  cpx(double a, double b):a(a),b(b){}
  double len() {
    return a * a + b * b;
  }
  cpx bar() {
    return cpx(a, -b);
  }
  cpx operator/=(int n) {
    a /= n, b /= n;
    return *this;
  }
};
cpx operator+(cpx a, cpx b) {
  return cpx(a.a + b.a, a.b + b.b);
}
cpx operator-(cpx a, cpx b) {
  return cpx(a.a - b.a, a.b - b.b);
}
cpx operator*(cpx a, cpx b) {
  return cpx(a.a * b.a - a.b * b.b, a.a * b.b
      + a.b * b.a);
}
cpx operator/(cpx a, cpx b) {
  cpx r = a * b.bar();
  return cpx(r.a / b.len(), r.b / b.len());
}
using cd = cpx;
//using cd = complex<double>;

void fft(vector<cd> & a, int inv) {
  int n = sz(a);
  for (int i = 1, j = 0; i < n; i++) {
    int bit = n >> 1;
    for (j ^= bit; !(j&bit); j ^= (bit>>=1));
    if (i < j)
      swap(a[i], a[j]);
  }
  for (int l = 2; l <= n; l *= 2) {
    double ang = 2 * pi / l * (inv ? -1 : 1);
    cd wl(cos(ang), sin(ang));
    for (int i = 0; i < n; i += l) {
      cd w(1);
      for (int j = i; j < i + l / 2; j++) {
        cd u = a[j], v = a[j + l / 2] * w;
        a[j] = u + v; a[j + l / 2] = u - v;
        w = w * wl;
      }
    }
  }
  if (inv)
    for (cd & x : a)
      x /= n;
}
```

7.14.2 Number Theoretic Transform

- Requirements:
  - MOD must be prime.
  - root must have order root_{pw} modulo MOD.
  - $n = a.\text{size}()$ must be a power of 2.
- E.g. if $\text{MOD} = c2^k + 1$, then g^c has order 2^k, where g is a primitive root modulo MOD.
- If inv $= 0$, the function evaluates the polynomial of degree $n-1$ (given by the n coefficients in a) at the n roots of unity (i.e. at $\text{root}^1, \text{root}^2, \ldots, \text{root}^n \equiv 1$).
- If inv $= 1$, the function computes the n coefficients of the polynomial determined by the points in a.
- Runtime $O(n \log n)$. In practice slower than regular FFT.

```
const int mod = 998244353; // 119 * 2^23 + 1
const int root = 15311432; // 3^119
const int iroot = 469870224; // 1 / root
const int root_pw = 1 << 23;

void fft(Poly& a, int inv = 0) {
  int n = sz(a);
  for(int i = 1, j = 0; i < n; i++) {
    int bit = n >> 1;
    for (j ^= bit; !(j&bit); j ^= (bit>>=1));
    if (i < j)
      swap(a[i], a[j]);
  }
```

```
  for(int l = 1; 2 * l <= n; l *= 2) {
    int wl = inv ? iroot : root;
    for (int i = l; 2 * i < root_pw; i *= 2)
      wl = wl * 1ll * wl % mod;
    for(int i = 0; i < n; i += 2 * l) {
      for (int j = i, w = 1; j < i + l; j++) {
        int u = a[j], v = a[j+l]*1ll*w % mod;
        a[j] = u+v < mod ? u+v : u+v-mod;
        a[j + l] = u-v < 0 ? u-v+mod : u-v;
        w = w * 1ll * wl % mod;
      }
    }
  }
  if (inv) {
    n = pw(n, mod - 2);
    for (int& i: a)
      i = n * 1ll * i % mod;
  }
}

Poly operator*(Poly x, Poly y) {
  int n = 2, s = sz(x) + sz(y) - 1;
  while (n / 2 < max(sz(x), sz(y))) n *= 2;
  x.resize(n);
  y.resize(n);
  fft(x), fft(y);
  for (int i = 0; i < n; i++)
    x[i] = x[i] * 1ll * y[i] % mod;
  fft(x, 1);
  x.resize(s);
  return x;
}
```

7.14.3 Multipoint Evaluation

- Evaluates a polynomial of degree N at the points $b_1, \cdots, b_M$ in $\mathcal{O}(N \log^2 N)$
- To use this code you have to implement the operator $\cdot$ and a function invert() which calculates the series $1/p(x)$.

```
Poly evaluate(const Poly& x, Poly b) {
  int n = max(sz(x) + 1, sz(b));
  vector<Poly> p(2 * n), q(2 * n);
  for (int i = 0; i < n; i++)
    q[i + n] = {i < sz(b) ? mod - b[i] : 0,
        1};
  for (int i = n - 1; i > 0; i--)
    q[i] = q[2 * i] * q[2 * i + 1];
  reverse(all(q[1])); q[1] = invert(q[1], sz(q
      [1]));
  reverse(all(q[1]));
  p[1] = range(x * q[1], sz(q[1]) - 1, 2 * n);
  p[1].resize(2 * n - sz(q[1]) + 1);
  for (int i = 2; i < sz(b) + n; i++)
    p[i] = range(p[i / 2] * q[i ^ 1], sz(q[i ^
      1]) - 1, sz(p[i / 2]));
  for (int i = 0; i < sz(b); i++)
    b[i] = p[i + n][0];
  return b;
}
```

7.14.4 Newton Iteration

- Solves the equation $f(A(x)) = 0$ (finds the first coefficients of $A(x)$)
- Start with some function $A_0(x)$ which solves the equation mod x
- $A_{k+1}(x) = A_k(x) - \frac{f(A_k(x))}{f'(A_k(x))}$ is a solution mod $x^{2^{k+1}}$
- Runtime $O(T(n))$ to evaluate the first n coefficients, if it takes $T(n)$ to evaluate $f(A(x))/f'(A(x))$
- Examples: (Q is given)
  - Find $A = Q^{-1}$, $f(A) = A^{-1} - Q \Rightarrow A_{k+1} = 2A_k - A_k^2 Q$
  - Find $A = \exp(Q)$, $f(A) = \ln(A) - Q \Rightarrow A_{k+1} = A_k(1 - \ln(A_k) + Q)$ Note that $\ln(A) = \int \frac{A'}{A} \mathrm{d}x$
  - Find $A = Q^\alpha$, $f(A) = A^{1/\alpha} - Q \Rightarrow A_{k+1} = A_k - \alpha(A_k^{1/\alpha} - Q)A_k^{1-1/\alpha}$ Note that you should be able to calculate $\sqrt[\alpha]{Q(0)}$. Alternatively you can calculate $\exp(\alpha \ln(Q))$.
- This can be extended to first order ODEs $x'(t) = f(x)$. Looking at the Taylorexpansion of f we obtain:

$$x'_{2n} \equiv f(x_n) + f'(x_n)(x_{2n} - x_n) \mod t^{2n}$$

This reduces with the integrating factor $\mu = e^{-\int f'(x_n)}$ to $(x_{2n}\mu)' \equiv (f(x_n) - f'(x_n)x_n)\mu \mod t^{2n}$

7.14.5 Inverse Series

- Calculates the first $2^{\lfloor \log n \rfloor + 1}$ coefficients of the series $\frac{1}{A(x)}$ where $A(x) = 1 + ..$ is a polynomial.
- Therefore the polynomials $B_k \equiv B_{k-1}(2 - AB_{k-1}) \mod x^{2^k}$ are calculated with $AB_k \equiv 1 \mod x^{2^k}$.
- Runtime $O(n \log n)$.

```
Poly invert(const Poly& x, int s) {
  Poly ret = {pw(x[0], mod - 2)};
  int k = 1;
  for (; k < s; k *= 2) {
    ret = ret + ret - (ret * ret) * range(x,
        0, 2 * k);
    ret.resize(2 * k);
  }
  ret.resize(s);
  return ret;
}
```

7.14.6 Polynom Division with remainder

- Calculates the coefficients of the two polynomials D and R with $A(x) = B(x) \cdot D(x) + R(x)$ and $\deg D = \deg A - \deg B$ for given A and B.
- Runtime $O(n \log n)$.

```
void divide(vector<int> a, vector<int> b,
    vector<int>& d, vector<int>& r) {
  int n = sz(a), m = sz(b);
  if (n-m+1 <= 0) { r = a; return; }
  reverse(a.begin(), a.end());
  reverse(b.begin(), b.end());
  d = mul(a, inv(b, m)); d.resize(n-m+1);
  reverse(a.begin(), a.end());
  reverse(b.begin(), b.end());
  reverse(d.begin(), d.end());
  r = sub(a, mul(b, d));
  r.resize(m);
}
```

7.14.7 Fast Walsh-Hadamard transform

- calculates the FWH transform (also known as xor-transform) of the polynomial A in $O(n \log n)$.
- This is the same as a multidimensional DFT of size $2 \times \cdots \times 2$. The DFT for a single dimension can be hardcoded.
- The DFT for a single dimension is the same as multiplying by the Hadamard Matrix $H = \begin{pmatrix} 1 & 1 \\ 1 & -1 \end{pmatrix}$. Applying the inverse transform is the same as multiplying by the inverse of this matrix.
- Use $\begin{pmatrix} 0 & 1 \\ 1 & 1 \end{pmatrix}$ for the **and** transform and $\begin{pmatrix} 1 & 1 \\ 1 & 0 \end{pmatrix}$ for the **or** transform. Remember that those two matrices are in $SL(2)$, hence you don't have to divide by n for the inverse transform.
- Application: Calculate $c_k = \sum_{i \oplus j = k} a_i b_j$ fast. Also the or transform is almost the same as sum over all submasks.
- We can also generalize this idea to addition mod m in base m (so $i \oplus_m j = k$). We therefore have to evaluate all the $\log_m n$ polynomials at all $m-$th primitive roots of unity.

```
void fwht(vector<int>& a, int inv = 0) {
  int n = sz(a); assert((n&-n) == n);
  for (int i = 2; i <= n; i *= 2)
    for (int j = 0; j < n; j += i)
      for (int k = j; k < j+i/2; k++) {
        int u = a[k], v = a[k+i/2];
        a[k] = u+v >= mod ? u+v : u+v-mod;
        a[k+i/2] = u-v < 0 ? u-v+mod : u-v;
      }
  if (inv) {
    n = pw(n, mod-2);
    for (int& i: a)
      i = i * 1ll * n % mod;
  }
}
```

7.14.8 Subset convolution

- Applies known operations like $\log, \cdot, \cdots$ to the set power series $f(x) = \sum_{s \subseteq S} a_s x^s$ in $O(n^2 2^n)$.
- The code shows how to multiply two such series. For other operations you only have to replace the multiplication part with the naive computation of this operation.

```
#define add(x, y) x = (x + y < mod ? x + y : x
    + y - mod)
const int N = 1 << 20;
int a[N], b[N], btc[N], ca[N][21], cb[N][21],
    cc[N][21];
// result is stored in cc[i][btc[i]]
void mul(int n) {
```

```
  for (int i = 0; i < 1 << n; i++)
    btc[i] = btc[i / 2] + (i & 1), ca[i][btc[i
        ]] = a[i], cb[i][btc[i]] = b[i];
  for (int i = 2; i <= 1 << n; i *= 2)
    for (int j = 0; j < 1 << n; j += i)
      for (int k = j; k < j+i/2; k++)
        for (int bt = 0; bt <= n; bt++)
          add(ca[k+i/2][bt], ca[k][bt]),
          add(cb[k+i/2][bt], cb[k][bt]);
  for (int msk = 0; msk < 1 << n; msk++)
    for (int i = 0; i <= n; i++) {
      unsigned long long v = 0;
      for (int j = 0; j <= i; j++)
        v += ca[msk][j] * (unsigned long long)
            cb[msk][i - j];
      cc[msk][i] = v % mod;
    }
  for (int i = 2; i <= 1 << n; i *= 2)
    for (int j = 0; j < 1 << n; j += i)
      for (int k = j; k < j+i/2; k++)
        for (int bt = 0; bt <= n; bt++)
          add(cc[k+i/2][bt], mod-cc[k][bt]);
}
```

7.15 Linear Algebra

7.15.1 Gauss-Jordan

```
// Gauss-Jordan elimination with full pivoting
    .
//
// Uses:
//   (1) solving systems of linear equations (
    AX=B)
//   (2) inverting matrices (AX=I)
//   (3) computing determinants of square
    matrices
//
// Running time: O(n^3)
//
// INPUT:    a[][] = an nxn matrix
//           b[][] = an nxm matrix
//
// OUTPUT:   X      = an nxm matrix (stored in
     b[][])
//           A^{-1} = an nxn matrix (stored in
     a[][])
//           returns determinant of a[][]

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;
}
```

7.15.2 Characteristic Polynomial

- Calculates $\det(A - xI)$ for an $N \times N$ matrix A
- Complexity: $\mathcal{O}(N^3)$

```
void transform(vector<vector<int>>& a) {
  int n = sz(a);
  for (int j = 0; j + 2 < n; j++) {
    int i = j+2; while (i < n && a[i][j] == 0)
        i++;
    if (i == n) continue;
    if (a[j+1][j] == 0) {
      swap(a[i], a[j+1]);
      for (int k = 0; k < n; k++)
        swap(a[k][i], a[k][j+1]);
    }
    int v = pw(a[j+1][j], mod-2);
    for (int k = j+2; k < n; k++) {
      int u = a[k][j] * 1ll * v % mod;
      for (int l = 0; l < n; l++) {
        a[k][l] = (a[k][l] - u * 1ll * a[j+1][
            l]) % mod;
        a[k][l] += a[k][l] < 0 ? mod : 0;
        a[l][j+1] = (a[l][j+1] + u * 1ll * a[l
            ][k]) % mod;
      }
    }
  }
}
vector<int> calc(vector<vector<int>>& a) {
  transform(a);
  int n = sz(a);
  vector<vector<int>> p(n+1); p[0] = {1};
  for (int k = 0; k < n; k++) {
    p[k+1] = vector<int>({!a[k][k] ? 0 : mod-a
        [k][k], 1}) * p[k];
    int v = 1;
    for (int i = 0; i < k; i++) {
      v = v * 1ll * a[k-i][k-i-1] % mod;
      p[k+1] = p[k+1] - (v * 1ll * a[k-i-1][k]
          % mod) * p[k-i-1];
    }
  }
  return p[n];
}
```

7.16 Linear Recurrence

7.16.1 kth Term

Given a linear recurrence relation, where each elements depends on the previous n elements, `kth(k)` computes the k-th term in $O(n^2 \log k)$. Initialisation: $a_0, \dots a_{n-1}$ are the n initial values. $p_1, \dots p_n$ describe the recurrence as $a_k = a_{k-1} \cdot p_1 + \dots a_{k-n} \cdot p_n$.

```
const int MAX_N = 2005; //a little larger
const int MOD = 1000000007;

int n,p[MAX_N],a[MAX_N];

// to improve constant factor compute mod MOD*
    MOD, if that fits in ll
void mul(ll *a, ll *b)  {
  static ll t[2*MAX_N];
  fill(t,t+2*n-1,0);
  for (int i=0; i<n; i++)
    for (int j=0; j<n; j++)
      t[i+j]=(t[i+j]+a[i]*b[j])%MOD;
  for (int i=2*n-2; i>=n; i--)
    for (int j=1; j<=n; j++)
      t[i-j]=(t[i-j]+t[i]*p[j])%MOD;
  copy(t,t+n,a);
}
int kth(ll k) {
  static ll r[MAX_N],t[MAX_N];
```

```
  fill(r,r+n,0),fill(t,t+n,0);
  for (r[0]=t[1]=1; k; k/=2,mul(t,t))
    if (k&1)
      mul(r,t);
  for (int i=0; i<n; i++)
    k=(k+r[i]*a[i])%MOD;
  return k;
}
```

7.16.2 Berlekamp-Massey

Recovers any n-order linear recurrence relation from the first $2n$ terms of the recurrence in $O(n^2)$. Useful for guessing linear recurrences after brute-forcing the first terms.

```
const int MOD = 1000000007; //prime!

//using fast exp ll fpow(ll a, ll b)
vector<ll> BerlekampMassey(vector<ll> s) {
  int n = s.size(), L = 0, m = 0;
  vector<ll> C(n), B(n), T;
  C[0] = B[0] = 1;
  ll b = 1;
  for (int i=0; i<n; i++) {
    ++m; ll d = s[i] % MOD;
    for (int j=1; j<=L; j++)
      d=(d + C[j] * s[i - j])%MOD;
    if (!d) continue;
    T = C; ll coef = d * fpow(b, MOD-2)%MOD;
    for (int j=m; j<n; j++)
      C[j] = (C[j] - coef * B[j - m])%MOD;
    if (2 * L > i) continue;
    L = i + 1 - L; B = T; b = d; m = 0;
  }
  C.resize(L + 1); C.erase(C.begin());
  for (auto &x : C)
    x=(MOD-x)%MOD;
  return C;
}
//BerlekampMassey({0, 1, 1, 3, 5, 11}) ->
    {1,2}
```

7.17 Simplex

Solves a general linear maximization problem: maximize $c^T x$ subject to $Ax \leq b$, $x \geq 0$. Returns `-inf` if there is no solution, `inf` if there are arbitrarily good solutions, or the maximum value of $c^T x$ otherwise. The input vector is set to an optimal x (or in the unbounded case, an arbitrary solution fulfilling the constraints). Numerical stability is not guaranteed. For better performance, define variables such that $x = 0$ is viable.

Usage:

```
vvd A = {{1,-1}, {-1,1}, {-1,-2}};
vd b = {1,1,-4}, c = {-1,-1}, x;
T val = LPSolver(A, b, c).solve(x);
```

Time: $O(NM \cdot \#pivots)$, where a pivot may be e.g. an edge relaxation. $O(2^n)$ in the general case.

```
typedef double T;
typedef vector<int> vi;
typedef vector<T> vd;
typedef vector<vd> vvd;

const T eps = 1e-8, inf = 1/.0;
#define MP make_pair
#define ltj(X) if(s == -1 || MP(X[j],N[j]) <
    MP(X[s],N[s])) s=j
#define rep(i, a, b) for(int i = a; i < (b);
    ++i)

struct LPSolver {
  int m, n;
  vi N, B;
  vvd D;

  LPSolver(const vvd& A, const vd& b, const vd
      & c) :
    m(b.size()), n(c.size()), N(n+1), B(m), D(
        m+2, vd(n+2)) {
      rep(i,0,m) rep(j,0,n) D[i][j] = A[i][j];
      rep(i,0,m) { B[i] = n+i; D[i][n] = -1; D
          [i][n+1] = b[i];}
      rep(j,0,n) { N[j] = j; D[m][j] = -c[j];
          }
      N[n] = -1; D[m+1][n] = 1;
    }

  void pivot(int r, int s) {
    T *a = D[r].data(), inv = 1 / a[s];
    rep(i,0,m+2) if (i != r && abs(D[i][s]) >
        eps) {
      T *b = D[i].data(), inv2 = b[s] * inv;
      rep(j,0,n+2) b[j] -= a[j] * inv2;
      b[s] = a[s] * inv2;
    }
    rep(j,0,n+2) if (j != s) D[r][j] *= inv;
    rep(i,0,m+2) if (i != r) D[i][s] *= -inv;
    D[r][s] = inv;
    swap(B[r], N[s]);
  }

  bool simplex(int phase) {
    int x = m + phase - 1;
    for (;;) {
      int s = -1;
      rep(j,0,n+1) if (N[j] != -phase) ltj(D[x
          ]);
      if (D[x][s] >= -eps) return true;
      int r = -1;
      rep(i,0,m) {
        if (D[i][s] <= eps) continue;
        if (r == -1 || MP(D[i][n+1] / D[i][s],
            B[i]) < MP(D[r][n+1] / D[r][s], B
           [r])) r = i;
      }
      if (r == -1) return false;
      pivot(r, s);
    }
  }

  T solve(vd &x) {
    int r = 0;
    rep(i,1,m) if (D[i][n+1] < D[r][n+1]) r =
        i;
    if (D[r][n+1] < -eps) {
      pivot(r, n);
      if (!simplex(2) || D[m+1][n+1] < -eps)
          return -inf;
      rep(i,0,m) if (B[i] == -1) {
        int s = 0;
        rep(j,1,n+1) ltj(D[i]);
        pivot(i, s);
      }
    }
    bool ok = simplex(1); x = vd(n);
    rep(i,0,m) if (B[i] < n) x[B[i]] = D[i][n
        +1];
    return ok ? D[m][n+1] : inf;
  }
};
```