Contents

1 Contest

1.1 Header

```cpp
#pragma once

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

template <class TH>
void _dbg(const char *sdbg, TH h) { cerr << sdbg << '=' << h << endl; }

template <class TH, class... TA>
void _dbg(const char *sdbg, TH h, TA... a)
{
  while (*sdbg != ',')
    cerr << *sdbg++;
  cerr << '=' << h << ',';
  _dbg(sdbg + 1, a...);
}

template <class L, class R>
ostream &operator<<(ostream &os, pair<L, R> p)
{
  return os << "(" << p.first << ", " << p.second << ")";
}

template <class Iterable, class = typename enable_if<!is_same<string,
   Iterable>::value>::type>
auto operator<<(ostream &os, Iterable v) -> decltype(os << *begin(v))
{
  os << "[";
  for (auto vv : v)
    os << vv << ", ";
  return os << "]";
}

#define debug(...) _dbg(#__VA_ARGS__, __VA_ARGS__)

typedef pair<int, int> pii;
typedef long long ll;
typedef long double ld;

const int inf = 0x3f3f3f3f;
const long long infll = 0x3f3f3f3f3f3f3f3fll;

#define sz(x) ((int)(x).size())

// Return 1 if x > 0, 0 if x == 0 and -1 if x < 0.
template <class T>
```

```
int sign(T x) { return (x > 0) - (x < 0); }

template <class T>
T abs(const T &x) { return (x < T(0)) ? -x : x; }

// Pretty good compilation command:
// g++ -g a.cpp --std=c++14 -Wall -Wextra -Wno-unused-result -Wconversion -
   Wfatal-errors -fsanitize=undefined,address -o a.out

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

1.2 Sample Debug

```
1 32cfcc  #include "header.hpp"
2 d41d8c
3 13a4b1  int main(void)
4 f95b70  {
5 3e8410    int a = 11, b = 12, c = 13;
6 b9ee34    vector<vector<int>> v = {{a, b}, {c}, {0, 1}};
7 2a803c    set<int> s = {a, b};
8 6b3b13    map<double, int> m;
9 af2bd1    m[2.5] = 2;
10  37a428    m[-3.1] = 3;
11  d41d8c
12  632de9    map<string, int> tab;
13  88ec8d    tab["abc"] = (int) 'a' + 'b' + 'c';
14  69908e    tab["abz"] = (int) 'a' + 'b' + 'z';
15  bd6def    int array[3] = {1, 2, 5};
16  d41d8c
17  5939a6    debug(a, b, c);
18  fb9ee1    debug(v);
19  b45aab    debug(s, m);
20  3cf91d    debug(tab);
21  d95ee2    debug(array); // This one does not work.
22  cbb184  }
```

1.3 Hash Code

```
#!/bin/bash

for i in $(seq 1 `wc -l < $1`); do
  echo -en "$i\t";
  sed -n $i'p' $1 | cpp -dD -P -fpreprocessed | tr -d '[:space:]' | md5sum |
   cut -c-6 | tr -d '\n';
  echo -en '\t';
  sed -n $i'p' $1;
```

done

2 Data Structures

2.1 BIT

```
#include "../../contest/header.hpp"

/*
  BIT: element update, range sum query and sum lower_bound in O(log(N)).
  Represents an array of elements in range [1, N].
*/

template <class T>
struct bit
{
  int n, LOGN;
  vector<T> val;
  bit(int _n) : n(_n), LOGN(log2(n + 1)), val(_n + 1, 0) {}

  // val[pos] += x
  void update(int pos, T x)
  {
    for (int i = pos; i <= n; i += -i & i)
      val[i] += x;
  }

  // sum of range [1, pos]
  T query(int pos)
  {
    T retv = 0;
    for (int i = pos; i > 0; i -= -i & i)
      retv += val[i];
    return retv;
  }

  // min pos such that sum of [1, pos] >= sum, or n + 1 if none exists.
  int lower_bound(T x)
  {
    T sum = 0;
    int pos = 0;

    for (int i = LOGN; i >= 0; i--)
      if (pos + (1 << i) <= n && sum + val[pos + (1 << i)] < x)
        sum += val[pos += (1 << i)];

    return pos + 1; // pos will have position of largest value less than x.
  }
};
```

2.2 BIT2D

```
1 5d1131  #include "../../contest/header.hpp"
2 d41d8c
3 d41d8c  /*
4 caf843    BIT: element update, range sum query in O(log(n) * log(m)). This
   can also be generalized for 3d.
5 a6cfe6    Represents a matrix of elements in range [1 ... n][1 ... m].
6 c4c9bd  */
7 d41d8c
8 4fce64  template <class T>
9 f6f3a7  struct bit2d
10  f95b70  {
11  14e0a7    int n, m;
12  f7ea55    vector<vector<T>> val;
13  9c8214    bit2d(int _n, int _m) : n(_n), m(_m), val(_n + 1, vector<T>(_m
   + 1, 0)) {}
14  d41d8c
15  d41d8c    // val[i][j] += x
16  4460cb    void update(int r, int c, T x)
17  f95b70    {
18  9e45d9      for (int i = r; i <= n; i += -i & i)
19  13d333        for (int j = c; j <= m; j += -j & j)
20  ff237f          val[i][j] += x;
21  cbb184    }
22  d41d8c
23  d41d8c    // sum of positions (1 ... r, 1 ... c)
24  450f85    T query(int r, int c)
25  f95b70    {
26  56622d      T retv = 0;
27  bc7409      for (int i = r; i > 0; i -= -i & i)
28  d53722        for (int j = c; j > 0; j -= -j & j)
29  86df71          retv += val[i][j];
30  6272cf      return retv;
31  cbb184    }
32  d41d8c
33  d41d8c    // sum of positions (ri ... rf, ci ... cf). (1 <= ri <= rf <= n
   ) and (1 <= ci <= cf <= m). TODO: test me.
34  bdc664    T query_rect(int ri, int ci, int rf, int cf)
35  f95b70    {
36  6072bc      return query(rf, cf) - query(rf, ci - 1) - query(ri - 1, cf)
   + query(ri - 1, ci - 1);
37  cbb184    }
38  2145c1  };
```

2.3 Dynamic Seg

```
1 5d1131  #include "../../contest/header.hpp"
2 d41d8c
3 d41d8c  /*
```

```
4 811629    Segment tree with dynamic memory allocation and arbitrary
   interval.
5 91eb69      Every operation is O(log(r-l))
6 b5a53f      Uses O(min(r-l, n*log(r-l)) memory, where n is the number of
   insertions.
7 d41d8c
8 ca2095    Constraints:
9 3dcfba      Segment tree range [l, r] must be such that 0 <= l <= r.
10  d41d8c
11  3db72f    Author: Arthur Pratti Dadalto
12  c4c9bd  */
13  d41d8c
14  4fce64  template<class T>
15  e4accb  struct node
16  f95b70  {
17  f48ea0    T val;
18  af32d9    node *left, *right;
19  d41d8c
20  995125    T get(int l, int r, int a, int b)
21  f95b70    {
22  47234b      if (l == a && r == b)
23  d943f4        return val;
24  814ad2      int mid = (l + 0ll + r) / 2;
25  f890f2      if (b <= mid)
26  ac57ce        return left ? left->get(l, mid, a, b) : 0;
27  a54f0c      else if (a > mid)
28  1c7837        return right ? right->get(mid + 1, r, a, b) : 0;
29  2954e9      else
30  9b1cb1        return (left ? left->get(l, mid, a, mid) : 0) + (right ?
   right->get(mid + 1, r, mid + 1, b) : 0);
31  cbb184    }
32  d41d8c
33  14d5ea    void update(int l, int r, int a, T x)
34  f95b70    {
35  bd3398      if (l == r)
36  c43fe0        val = x;
37  2954e9      else
38  f95b70      {
39  814ad2        int mid = (l + 0ll + r) / 2;
40  a49729        if (a <= mid)
41  1ec55a          (left ? left : (left = new node()))->update(l, mid, a, x)
   ;
42  2954e9        else
43  92fe63          (right ? right : (right = new node()))->update(mid + 1, r
   , a, x);
44  d41d8c
45  dd51dd        val = (left ? left->val : 0) + (right ? right->val : 0);
46  cbb184      }
47  cbb184    }
48  2145c1  };
```

49 d41d8c

2.4 Linear Container

```
1 5d1131  #include "../../contest/header.hpp"
2 d41d8c
3 d41d8c  /*
4 1ee0c3    Line Container (most common for convex hull trick). Amortized O(
   log N) per operation.
5 48cf95      Container where you can add lines of the form kx+m, and query
   maximum values at points x.
6 dc45cd      Useful for dynamic programming.
7 d41d8c
8 1d1558    Source: https://github.com/kth-competitive-programming/kactl/blob
   /master/content/contest/template.cpp
9 c4c9bd  */
10  d41d8c
11  3fe318  struct line
12  f95b70  {
13  3e2604    mutable ll k, m, p;
14  889941    bool operator<(const line &o) const { return k < o.k; }
15  abfd1f    bool operator<(ll x) const { return p < x; }
16  2145c1  };
17  d41d8c
18  0c8ce5  struct line_container : multiset<line, less<>>
19  f95b70  {
20  d41d8c    // (for doubles, use inf = 1/.0, div(a,b) = a/b)
21  f5e3e7    const ll inf = LLONG_MAX;
22  d41d8c
23  9608c5    ll div(ll a, ll b)
24  f95b70    { // floored division
25  353cf0      return a / b - ((a ^ b) < 0 && a % b);
26  cbb184    }
27  d41d8c
28  9c092f    bool isect(iterator x, iterator y)
29  f95b70    {
30  f959d1      if (y == end())
31  f95b70      {
32  09a75e        x->p = inf;
33  d1fe4d        return false;
34  cbb184      }
35  3cca77      if (x->k == y->k)
36  83e301        x->p = x->m > y->m ? inf : -inf;
37  2954e9      else
38  b4284e        x->p = div(y->m - x->m, x->k - y->k);
39  870ec6      return x->p >= y->p;
40  cbb184    }
41  d41d8c
42  928f4b    void add(ll k, ll m)
43  f95b70    {
```

```
44  116e6c      auto z = insert({k, m, 0}), y = z++, x = y;
45  2d9d80      while (isect(y, z))
46  96cee5        z = erase(z);
47  d94b4e      if (x != begin() && isect(--x, y))
48  c07d21        isect(x, y = erase(y));
49  57dd20      while ((y = x) != begin() && (--x)->p >= y->p)
50  77462a        isect(x, erase(y));
51  cbb184    }
52  d41d8c
53  e8b5c2    ll query(ll x)
54  f95b70    {
55  229883      assert(!empty());
56  7d13b8      auto l = *lower_bound(x);
57  96a2bc      return l.k * x + l.m;
58  cbb184    }
59  2145c1  };
60  d41d8c
```

2.5 Min Queue

```
1 d41d8c  /*
2 958401    max(min) queue with O(1) get_max(min).
3 d41d8c
4 f67dcb    Tips:
5 c53808      - Useful for sliding window 1D and 2D.
6 af9dc1      - For 2D problems, you will need to pre-compute another matrix,
7 55e3e9      by making a row-wise traversal, and calculating the min/max
   value
8 79c288      beginning in each cell. Then you just make a column-wise
   traverse
9 b21db2      as they were each an independent array.
10  c4c9bd  */
11  d41d8c
12  8f0a66  struct max_queue
13  f95b70  {
14  84841a    queue<ll> q;
15  889d23    deque<ll> s;
16  d41d8c
17  dbb27b    int size()
18  f95b70    {
19  593f12      return (int)q.size();
20  cbb184    }
21  d41d8c
22  a1fe24    void push(ll val)
23  f95b70    {
24  d41d8c      // while (!s.empty() && s.back() > val) -> for a min_queue
25  1cb658      while (!s.empty() && s.back() < val) // for a max_queue
26  342ca4        s.pop_back();
27  fcc849      s.push_back(val);
28  d41d8c
```

```
29  380c99      q.push(val);
30  cbb184    }
31  d41d8c
32  d99fc4    void pop()
33  f95b70    {
34  7a8432      ll u = q.front();
35  833270      q.pop();
36  d41d8c
37  de7036      if (!s.empty() && s.front() == u)
38  784c93        s.pop_front();
39  cbb184    }
40  d41d8c
41  ba28bf    ll get_max()
42  f95b70    {
43  eccd4b      return s.front(); // same for min and max queue
44  cbb184    }
45  d41d8c
46  2145c1  };
```

2.6 Persistent Seg

```
1 5d1131  #include "../../contest/header.hpp"
2 d41d8c
3 d41d8c  /*
4 a032aa    Persistent Segment Tree:
5 115cd2      Segment tree that stores all previous versions of itself.
6 91eb69      Every operation is O(log(r-l))
7 ad671a      Uses O(n*log(r-l)) memory, where n is the number of updates.
8 d41d8c
9 b95cae    Usage:
10  aca1a7      A new root is created for every persistent update (p_update)
   and returned.
11  0d5abd      Queries can be performed on any root as if it were a usual
   segment tree.
12  61a20d      You should keep a list of roots. Something like:
13  072987        vector<node *> roots = {new node()};
14  bf8bc0        roots.push_back(p_update(roots.back(), 0, 2*MAXV, a[i] +
   MAXV, v + 1));
15  d41d8c
16  ca2095    Constraints:
17  3dcfba      Segment tree range [l, r] must be such that 0 <= l <= r.
18  d41d8c
19  3db72f    Author: Arthur Pratti Dadalto
20  c4c9bd  */
21  d41d8c
22  e4accb  struct node
23  f95b70  {
24  97f03f    int val;
25  af32d9    node *left, *right;
26  d41d8c
```

```
27  1f6b0f    node(int x=0) : val(x), left(NULL), right(NULL) {}
28  2f77b9    node(node *l, node *r) : left(l), right(r) { val = (left ? left
   ->val : 0) + (right ? right->val : 0); }
29  d41d8c
30  f219f1    int get(int l, int r, int a, int b)
31  f95b70    {
32  47234b      if (l == a && r == b)
33  d943f4        return val;
34  814ad2      int mid = (l + 0ll + r) / 2;
35  f890f2      if (b <= mid)
36  ac57ce        return left ? left->get(l, mid, a, b) : 0;
37  a54f0c      else if (a > mid)
38  1c7837        return right ? right->get(mid + 1, r, a, b) : 0;
39  2954e9      else
40  9b1cb1        return (left ? left->get(l, mid, a, mid) : 0) + (right ?
   right->get(mid + 1, r, mid + 1, b) : 0);
41  cbb184    }
42  2145c1  };
43  d41d8c
44  63f202  node *p_update(node *prev, int l, int r, int a, int x)
45  f95b70  {
46  bd3398    if (l == r)
47  13478f      return new node(x);
48  d41d8c
49  814ad2    int mid = (l + 0ll + r) / 2;
50  a49729    if (a <= mid)
51  b73799      return new node(p_update(prev ? prev->left : NULL, l, mid, a,
    x), prev ? prev->right : NULL);
52  2954e9    else
53  460332      return new node(prev ? prev->left : NULL, p_update(prev ?
   prev->right : NULL, mid + 1, r, a, x));
54  cbb184  }
55  d41d8c
```

2.7 Lazy Seg

```
1 2b74fa  #include <bits/stdc++.h>
2 ca417d  using namespace std;
3 d41d8c
4 d41d8c  /*
5 6f561b    Segment Tree with Lazy updates:
6 d8b1dc      Range update and range query in O(log(MAX_RANGE))
7 c329b0      Binary search on tree in O(log(MAX_RANGE))
8 05382c      Given as an example since it is not worth it to copy a generic
   tree during a contest.
9 d41d8c
10  e3c955    Solves: https://codeforces.com/contest/1179/problem/C
11  c4c9bd  */
12  d41d8c
13  ab0dbf  #define MAX_RANGE 1123456
```

```
14  d41d8c
15  fd87fe  int val[4 * MAX_RANGE];
16  802d92  int delta[4 * MAX_RANGE];
17  d41d8c
18  4ee394  #define left(i) ((i) << 1)
19  56e5cf  #define right(i) (((i) << 1) + 1)
20  d41d8c
21  0379af  void prop(int id, int l, int r)
22  f95b70  {
23  cfd4b4    if (l != r)
24  f95b70    {
25  d41d8c      // Updates need to be numerically stackable (e.g. not valid
    to have a list of updates).
26  df541b      delta[left(id)] += delta[id];
27  966351      delta[right(id)] += delta[id];
28  cbb184    }
29  d41d8c
30  21c2c8    val[id] += delta[id]; // Node value needs to be obtainable
    without propagating all the way to root.
31  0a8860    delta[id] = 0;
32  cbb184  }
33  d41d8c
34  d41d8c  // Sum x in all elements in range [a, b].
35  f2b4f2  void update(int id, int l, int r, int a, int b, int x)
36  f95b70  {
37  addc1f    if (a == l && b == r)
38  f95b70    {
39  d50197      delta[id] += x;
40  b62cfe      prop(id, l, r);
41  cbb184    }
42  2954e9    else
43  f95b70    {
44  b62cfe      prop(id, l, r);
45  ae007b      int mid = (l + r) / 2;
46  f890f2      if (b <= mid)
47  f95b70      {
48  6dbd37        update(left(id), l, mid, a, b, x);
49  384ec5        prop(right(id), mid + 1, r);
50  cbb184      }
51  a54f0c      else if (a > mid)
52  f95b70      {
53  859d13        update(right(id), mid + 1, r, a, b, x);
54  221ad0        prop(left(id), l, mid);
55  cbb184      }
56  2954e9      else
57  f95b70      {
58  fc79c7        update(left(id), l, mid, a, mid, x);
59  04c83e        update(right(id), mid + 1, r, mid + 1, b, x);
60  cbb184      }
61  d41d8c
```

```
62  caf644      val[id] = min(val[left(id)], val[right(id)]);
63  cbb184    }
64  cbb184  }
65  d41d8c
66  d41d8c  // Get the minimum value in range [a, b].
67  9fed20  int get(int id, int l, int r, int a, int b)
68  f95b70  {
69  b62cfe    prop(id, l, r);
70  addc1f    if (a == l && b == r)
71  a0328b      return val[id];
72  2954e9    else
73  f95b70    {
74  ae007b      int mid = (l + r) / 2;
75  f890f2      if (b <= mid)
76  c55f80        return get(left(id), l, mid, a, b);
77  a54f0c      else if (a > mid)
78  26dd34        return get(right(id), mid + 1, r, a, b);
79  2954e9      else
80  5e3fad        return min(get(left(id), l, mid, a, mid), get(right(id),
    mid + 1, r, mid + 1, b));
81  cbb184    }
82  cbb184  }
83  d41d8c
84  d41d8c  // Find index of rightmost element which is less than x. (works
    because this is a seg of min)
85  0529b3  int bsearch(int id, int l, int r, int x)
86  f95b70  {
87  b62cfe    prop(id, l, r);
88  d41d8c
89  bd3398    if (l == r)
90  f7d2ed      return (val[id] < x) ? l : -1;
91  2954e9    else
92  f95b70    {
93  ae007b      int mid = (l + r) / 2;
94  221ad0      prop(left(id), l, mid);
95  384ec5      prop(right(id), mid + 1, r);
96  f01b35      if (val[right(id)] < x)
97  018a94        return bsearch(right(id), mid + 1, r, x);
98  2954e9      else
99  bad725        return bsearch(left(id), l, mid, x);
100 cbb184    }
101 cbb184  }
102 d41d8c
103 1037bf  #define MAXN 312345
104 d41d8c
105 a58cd5  int a[MAXN];
106 c4b25f  int b[MAXN];
107 d41d8c
108 13a4b1  int main(void)
109 f95b70  {
```

```
110 b067b3    int n, m, q, tp, x, y;
111 d69917    scanf("%d %d", &n, &m);
112 5359f3    for (int i = 1; i <= n; i++)
113 f95b70    {
114 9376f3      scanf("%d", &a[i]);
115 49e934      update(1, 1, 1000000, 1, a[i], -1);
116 cbb184    }
117 d41d8c
118 8eae24    for (int i = 1; i <= m; i++)
119 f95b70    {
120 264aeb      scanf("%d", &b[i]);
121 472fcc      update(1, 1, 1000000, 1, b[i], 1);
122 cbb184    }
123 d41d8c
124 4aaeab    scanf("%d", &q);
125 a953ae    while (q--)
126 f95b70    {
127 960099      scanf("%d %d %d", &tp, &x, &y);
128 abc772      if (tp == 1)
129 f95b70      {
130 996a9b        update(1, 1, 1000000, 1, a[x], 1);
131 e603e6        a[x] = y;
132 28cfa0        update(1, 1, 1000000, 1, a[x], -1);
133 cbb184      }
134 2954e9      else
135 f95b70      {
136 8dbabe        update(1, 1, 1000000, 1, b[x], -1);
137 0464a9        b[x] = y;
138 bc18aa        update(1, 1, 1000000, 1, b[x], 1);
139 cbb184      }
140 d41d8c
141 584906      int tmp = bsearch(1, 1, 1000000, 0);
142 d41d8c
143 d41d8c      // Test of get and bsearch. Make sure all to the right are
    non-negative.
144 5a5bec      if (tmp != 1000000)
145 5df0f6        assert(get(1, 1, 1000000, tmp == -1 ? 1 : (tmp + 1),
    1000000) >= 0);
146 c3e568      if (tmp != -1)
147 1d95f2        assert(get(1, 1, 1000000, tmp, tmp) < 0);
148 d41d8c
149 b03a7a      printf("%d\n", tmp);
150 cbb184    }
151 cbb184  }
```

2.8 Key Treap

```
1 5d1131  #include "../../contest/header.hpp"
2 d41d8c
3 d41d8c  /*
```

```
4 1977a5    Treap:
5 3ca64f      This treap implements something like a c++ set with additional
   operations: find the k-th element and count elements less than a given
   value.
6 d41d8c
7 4c88cf    Time: O(log N) per operation.
8 d41d8c
9 3db72f    Author: Arthur Pratti Dadalto
10  c4c9bd  */
11  d41d8c
12  41c55a  namespace treap
13  f95b70  {
14  e4accb  struct node
15  f95b70  {
16  97f03f    int val; // node key.
17  ee1179    int p;   // node heap priority.
18  59afd1    int num; // node subtree size.
19  af32d9    node *left, *right;
20  d41d8c
21  71091e    node(int _val) : val(_val), p(rand()), num(1), left(NULL),
   right(NULL) {}
22  2145c1  };
23  d41d8c
24  48f3b4  int get_num(node *root)
25  f95b70  {
26  424a36    return (root == NULL) ? 0 : root->num;
27  cbb184  }
28  d41d8c
29  68f1eb  void update_num(node *root)
30  f95b70  {
31  47a6f1    root->num = get_num(root->left) + get_num(root->right) + 1;
32  cbb184  }
33  d41d8c
34  afdba0  node *rotate_left(node *root)
35  f95b70  {
36  d25f1b    node *a = root;
37  a95379    node *b = root->right;
38  d41d8c
39  b51426    a->right = b->left;
40  e7e30a    b->left = a;
41  a5e0c3    update_num(a);
42  2b11db    update_num(b);
43  73f89f    return b;
44  cbb184  }
45  d41d8c
46  f17a34  node *rotate_right(node *root)
47  f95b70  {
48  d25f1b    node *a = root;
49  eb0328    node *b = root->left;
50  d41d8c
```

```
51  a09684    a->left = b->right;
52  7352c4    b->right = a;
53  a5e0c3    update_num(a);
54  2b11db    update_num(b);
55  73f89f    return b;
56  cbb184  }
57  d41d8c
58  d41d8c  // Insert new node with key x in treap rooted at root if not
    already there.
59  960bce  node *insert(node *root, int x)
60  f95b70  {
61  0edbc9    if (root == NULL)
62  13478f      return new node(x);
63  6b2a0b    if (x > root->val)
64  34c9df      root->right = insert(root->right, x);
65  ba0dc8    else if (x < root->val)
66  12f5b5      root->left = insert(root->left, x);
67  d41d8c
68  622638    update_num(root);
69  d41d8c
70  4f4bcf    if (root->right && root->right->p > root->p)
71  04107a      root = rotate_left(root);
72  c93ea7    if (root->left && root->left->p > root->p)
73  3f3108      root = rotate_right(root);
74  e2fc54    return root;
75  cbb184  }
76  d41d8c
77  d41d8c  // Remove node with key x in treap rooted at root if present.
78  d0ba77  node *remove(node *root, int x)
79  f95b70  {
80  0edbc9    if (root == NULL)
81  ea9b0a      return NULL;
82  6b2a0b    if (x > root->val)
83  fed39a      root->right = remove(root->right, x);
84  ba0dc8    else if (x < root->val)
85  6cf773      root->left = remove(root->left, x);
86  fb8e77    else if (root->left == NULL)
87  4de2d2      root = root->right;
88  a15580    else if (root->right == NULL)
89  2d4ff4      root = root->left;
90  386129    else if (root->left->p > root->right->p)
91  f95b70    {
92  3f3108      root = rotate_right(root);
93  fed39a      root->right = remove(root->right, x);
94  cbb184    }
95  2954e9    else
96  f95b70    {
97  04107a      root = rotate_left(root);
98  6cf773      root->left = remove(root->left, x);
99  cbb184    }
```

```
100 e6a2b0    if (root)
101 622638      update_num(root);
102 e2fc54    return root;
103 cbb184  }
104 d41d8c
105 d41d8c  // Return the k-th smallest element in tree rooted at root.
106 3576ec  int kth(node *root, int k)
107 f95b70  {
108 f9e30a    if (get_num(root->left) >= k)
109 7473ee      return kth(root->left, k);
110 f3e79f    else if (get_num(root->left) + 1 == k)
111 ae0ddc      return root->val;
112 2954e9    else
113 235aa0      return kth(root->right, k - get_num(root->left) - 1);
114 cbb184  }
115 d41d8c
116 d41d8c  // Return the number of elements smaller than x in tree rooted at
    root.
117 194e12  int count(node *root, int x)
118 f95b70  {
119 0edbc9    if (root == NULL)
120 bb30ba      return 0;
121 83010a    if (x < root->val)
122 da7c4c      return count(root->left, x);
123 08e5c0    else if (x == root->val)
124 140f45      return get_num(root->left);
125 2954e9    else
126 b73a02      return get_num(root->left) + 1 + count(root->right, x);
127 cbb184  }
128 cbb184  } // namespace treap
```

2.9 Sequential Treap

```
1 5d1131  #include "../../contest/header.hpp"
2 d41d8c
3 d41d8c  /*
4 1977a5    Treap:
5 5c39c7      A short self-balancing tree. It acts as a sequential container
   with log-time splits/joins, and
6 df7261      is easy to augment with additional data.
7 d41d8c
8 4c88cf    Time: O(log N) per operation.
9 d41d8c
10  ca2095    Constraints:
11  c1b810      Acts as a vector of size N, with positions in range [0, N-1].
12  d41d8c
13  1d1558    Source: https://github.com/kth-competitive-programming/kactl/
   blob/master/content/data-structures/Treap.h
14  d41d8c
15  b95cae    Usage:
```

```
16  24eb84      To insert elements, create one node treaps. (e.g. treap::ins(
   root, new treap::node(x), i))
17  acfc60      To augment with extra data you should mostly add stuff to the
    recalc function. (e.g. to make it work like a seg tree)
18  03bb33      See applications for more usage examples.
19  c4c9bd  */
20  d41d8c
21  41c55a  namespace treap
22  f95b70  {
23  e4accb  struct node
24  f95b70  {
25  8f5901    node *l = 0, *r = 0;
26  97f03f    int val;   // Any value associated with node.
27  ee1179    int p;   // Node heap priority.
28  c6aff2    int c = 1; // Node subtree size.
29  674490    node(int val) : val(val), p(rand()) {}
30  86d631    void recalc();
31  2145c1  };
32  d41d8c
33  853943  int cnt(node *n) { return n ? n->c : 0; }
34  9af082  void node::recalc() { c = cnt(l) + cnt(r) + 1; }
35  d41d8c
36  d41d8c  // Apply function f on each tree node in order.
37  044d82  template <class F>
38  d5442c  void each(node *n, F f)
39  f95b70  {
40  f63660    if (n)
41  f95b70    {
42  cbc351      each(n->l, f);
43  ed31a5      f(n->val);
44  f5ab50      each(n->r, f);
45  cbb184    }
46  cbb184  }
47  d41d8c
48  d41d8c  // Split treap rooted at n in two treaps containing positions [0,
    k) and [k, ...)
49  de9c69  pair<node *, node *> split(node *n, int k)
50  f95b70  {
51  a020ba    if (!n)
52  e70a07      return {NULL, NULL};
53  9416bd    if (cnt(n->l) >= k) // "n->val >= k" for lower_bound(k)
54  f95b70    {
55  215a80      auto pa = split(n->l, k);
56  f3cfa7      n->l = pa.second;
57  2f09c0      n->recalc();
58  c05937      return {pa.first, n};
59  cbb184    }
60  2954e9    else
61  f95b70    {
62  7c23f0      auto pa = split(n->r, k - cnt(n->l) - 1); // and just "k"
```

```
63  d37e77      n->r = pa.first;
64  2f09c0      n->recalc();
65  7af31a      return {n, pa.second};
66  cbb184    }
67  cbb184  }
68  d41d8c
69  d41d8c  // Merge treaps l and r keeping order (l first).
70  7f5419  node *merge(node *l, node *r)
71  f95b70  {
72  0c92a8    if (!l)
73  4c1f3c      return r;
74  6bf95d    if (!r)
75  792fd4      return l;
76  a0ade2    if (l->p > r->p)
77  f95b70    {
78  ed7b68      l->r = merge(l->r, r);
79  bf6a1f      l->recalc();
80  792fd4      return l;
81  cbb184    }
82  2954e9    else
83  f95b70    {
84  654f23      r->l = merge(l, r->l);
85  cda92d      r->recalc();
86  4c1f3c      return r;
87  cbb184    }
88  cbb184  }
89  d41d8c
90  d41d8c  // Insert treap rooted at n into position pos of treap rooted at
    t.
91  3fc637  node *ins(node *t, node *n, int pos)
92  f95b70  {
93  ca9a9f    auto pa = split(t, pos);
94  cc8215    return merge(merge(pa.first, n), pa.second);
95  cbb184  }
96  d41d8c
97  d41d8c  // Remove node at position pos from treap rooted at t.
98  1e0b32  node *rem(node *t, int pos)
99  f95b70  {
100 abdf75    node *a, *b, *c;
101 cf9546    tie(a, b) = split(t, pos);
102 0052e9    tie(b, c) = split(b, 1);
103 d41d8c
104 625cf2    delete b;
105 a300e4    return merge(a, c);
106 cbb184  }
107 d41d8c
108 d41d8c  // Example application: do a query in range [l, r].
109 0475c8  node *query(node *t, int l, int r)
110 f95b70  {
111 abdf75    node *a, *b, *c;
```

```
112 a8341d    tie(a, b) = split(t, l);
113 89f194    tie(b, c) = split(b, r - l + 1);
114 d41d8c
115 d41d8c    // printf("%lld\n", b->tab);
116 d41d8c
117 53aa0f    return merge(merge(a, b), c);
118 cbb184  }
119 d41d8c
120 d41d8c  // Example application: move the range [l, r) to index k.
121 b51124  void move(node *&t, int l, int r, int k)
122 f95b70  {
123 abdf75    node *a, *b, *c;
124 a8341d    tie(a, b) = split(t, l);
125 e81a2b    tie(b, c) = split(b, r - l);
126 1527bb    if (k <= l)
127 eeb6c2      t = merge(ins(a, b, k), c);
128 2954e9    else
129 646d6a      t = merge(a, ins(c, b, k - r));
130 cbb184  }
131 cbb184  } // namespace treap
```

3 Geometry

3.1 2D

```
1 5d1131  #include "../../contest/header.hpp"
2 d41d8c
3 d41d8c  // 2D geometry operations. This file should not have algorithms.
4 d41d8c  // Author: some of it by Arthur Pratti Dadalto.
5 d41d8c  // Source: some of it from https://github.com/kth-competitive-
   programming/kactl/blob/master/content/geometry/.
6 d41d8c  // Usage: avoid int unless necessary.
7 d41d8c
8 d41d8c  // When increasing EPS, keep in mind that sqrt(1e9^2 + 1) = 1e9 + 5
   e-10.
9 22c921  const double EPS = 1e-12;
10  d41d8c
11  d41d8c  // Point struct implementation. Some methods are useful only when
    using this to represent vectors.
12  4fce64  template <class T>
13  4befb0  struct point
14  f95b70  {
15  5dcf91    typedef point<T> P;
16  645c5d    T x, y;
17  d41d8c
18  571f13    explicit point(T x = 0, T y = 0) : x(x), y(y) {}
19  0d0d56    bool operator<(P p) const { return tie(x, y) < tie(p.x, p.y); }
20  ec7475    bool operator==(P p) const { return tie(x, y) == tie(p.x, p.y);
    }
21  2798c7    P operator+(P p) const { return P(x + p.x, y + p.y); }
22  40d57e    P operator-(P p) const { return P(x - p.x, y - p.y); }
23  e03fa4    P operator*(T d) const { return P(x * d, y * d); }
24  0b99e8    P operator/(T d) const { return P(x / d, y / d); }
25  57bee4    T dot(P p) const { return x * p.x + y * p.y; }
26  460881    T cross(P p) const { return x * p.y - y * p.x; }
27  b3fab9    T cross(P a, P b) const { return (a - *this).cross(b - *this);
   } // product sign: right hand rule from a to b.
28  f681d2    T dist2() const { return x * x + y * y; }            //
   Distance squared to origin.
29  18b7a8    double dist() const { return sqrt((double)dist2()); }      //
   Vector norm (distance to origin).
30  9073ff    double angle() const { return atan2(y, x); }           // angle
    to x-axis in interval [-pi, pi]
31  6f5d42    point<double> unit() const { return *this / dist(); }      //
   makes dist()=1 (unit vector).
32  200c8f    P perp() const { return P(-y, x); }                // rotates
   +90 degrees around origin.
33  567be8    point<double> normal() const { return perp().unit(); }       //
    perpendicular unit vector.
34  82fcdd    point<double> rotate(double a) const               // returns
   point rotated 'a' radians ccw around the origin.
```

```
35  f95b70    {
36  80d6a0      return P(x * cos(a) - y * sin(a), x * sin(a) + y * cos(a));
37  cbb184    }
38  8ad6e5    double angle(P p) const { return p.rotate(-angle()).angle(); }
   // Angle between the vectors in interval [-pi, pi]. Positive if p is ccw
   from this.
39  2145c1  };
40  d41d8c
41  d41d8c  // Solves the linear system {a * x + b * y = e
42  d41d8c  //                          {c * x + d * y = f
43  d41d8c  // Returns {1, {x, y}} if solution is unique, {0, {0,0}} if no
   solution and {-1, {0,0}} if infinite solutions.
44  d41d8c  // If using integer function type, this will give wrong answer if
    answer is not integer.
45  d41d8c  // TODO: test me with integer and non-integer.
46  4fce64  template <class T>
47  562c39  pair<int, point<T>> linear_solve2(T a, T b, T c, T d, T e, T f)
48  f95b70  {
49  468cb9    point<T> retv;
50  256940    T det = a * d - b * c;
51  d41d8c
52  57f40d    if (det == 0) // Maybe do EPS compare if using floating point.
53  f95b70    {
54  cdd981      if (b * f == d * e && a * f == c * e)
55  3d7337        return {-1, point<T>()};
56  37dde3      return {0, point<T>()};
57  cbb184    }
58  d41d8c
59  d41d8c    // In case solution needs to be integer, use something like the
    line below.
60  d41d8c    // assert((e * d - f * b) % det == 0 && (a * f - c * e) % det
   == 0);
61  d41d8c
62  848480    return {1, point<T>((e * d - f * b) / det, (a * f - c * e) /
   det)};
63  cbb184  }
64  d41d8c
65  d41d8c  // Represents line segments defined by two points.
66  4fce64  template <class T>
67  4b2ec6  struct segment
68  f95b70  {
69  5dcf91    typedef point<T> P;
70  efb78f    P pi, pf; // Initial and final points.
71  d41d8c
72  a76c62    explicit segment(P a = P(), P b = P()) : pi(a), pf(b) {}
73  d41d8c
74  d41d8c    // Distance from this segment to a given point. TODO: test me.
75  325177    double dist(P p)
76  f95b70    {
77  58fd41      if (pi == pf)
```

```
78  adefd2        return (p - pi).dist();
79  96a4f0      auto d = (pf - pi).dist2();
80  486c32      auto t = min(d, max(.0, (p - pi).dot(pf - pi)));
81  5dab06      return ((p - pi) * d - (pf - pi) * t).dist() / d;
82  cbb184    }
83  d41d8c
84  d41d8c    // Checks if given point belongs to segment. Use dist(p) <= EPS
    instead when using point<double>.
85  0e3dba    bool on_segment(P p)
86  f95b70    {
87  50f719      return p.cross(pi, pf) == 0 && (pi - p).dot(pf - p) <= 0;
88  cbb184    }
89  d41d8c
90  d41d8c    // If a unique intersection point between the line segments
   exists then it is returned.
91  d41d8c    // If no intersection point exists an empty vector is returned.
92  d41d8c    // If infinitely many exist a vector with 2 elements is
   returned, containing the endpoints of the common line segment.
93  d41d8c    // The wrong position will be returned if P is point<ll> and
   the intersection point does not have integer coordinates.
94  d41d8c    // However, no problem in using it to check if intersects or
   not in this case (size of vector will be correct).
95  d41d8c    // Products of **three** coordinates are used in intermediate
   steps so watch out for overflow if using int or long long.
96  f3f800    vector<P> intersect(segment rhs)
97  f95b70    {
98  9b1730      auto oa = rhs.pi.cross(rhs.pf, pi), ob = rhs.pi.cross(rhs.pf,
    pf),
99  1d46ec           oc = pi.cross(pf, rhs.pi), od = pi.cross(pf, rhs.pf);
100 d41d8c
101 d41d8c      // Checks if intersection is single non-endpoint point.
102 288e4c      if (sign(oa) * sign(ob) < 0 && sign(oc) * sign(od) < 0)
103 655339        return {(pi * ob - pf * oa) / (ob - oa)};
104 d41d8c
105 4c122f      set<P> s;
106 0373dd      if (rhs.on_segment(pi))
107 f07e25        s.insert(pi);
108 6725fe      if (rhs.on_segment(pf))
109 3c93ab        s.insert(pf);
110 3ad8fc      if (on_segment(rhs.pi))
111 522b2f        s.insert(rhs.pi);
112 f425cd      if (on_segment(rhs.pf))
113 d1c5a5        s.insert(rhs.pf);
114 d2dd66      return vector<P>(s.begin(), s.end());
115 cbb184    }
116 2145c1  };
117 d41d8c
118 d41d8c  // Represents a line by its equation in the form a * x + b * y =
   c.
119 d41d8c  // Can be created from two points or directly from constants.
```

```
120 4fce64  template <class T>
121 3fe318  struct line
122 f95b70  {
123 5dcf91    typedef point<T> P;
124 52d831    T a, b, c; // line a * x + b * y = c
125 d41d8c
126 f4f0fd    explicit line(P p1, P p2) // TODO: test me.
127 f95b70    {
128 4c2f1e      assert(!(p1 == p2));
129 6a88e5      a = p2.y - p1.y;
130 82330e      b = p1.x - p2.x;
131 cfae8e      c = a * p1.x + b * p1.y;
132 d41d8c
133 d41d8c      // In case of int, it is useful to scale down by gcd (e.g to
    use in a set).
134 d41d8c      // Might be useful to normalize here.
135 cbb184    }
136 d41d8c
137 510551    explicit line(T _a, T _b, T _c) : a(_a), b(_b), c(_c) {}
138 d41d8c
139 d41d8c    // Distance from this line to a given point. TODO: test me.
140 325177    double dist(P p)
141 f95b70    {
142 d37216      return abs(a * p.x + b * p.y - c) / sqrt((double)(a * a + b *
     b));
143 cbb184    }
144 d41d8c
145 d41d8c    // Intersects this line with another given line. See
    linear_solve2 for usage. TODO: test me.
146 4a5d8e    pair<int, P> intersect(line rhs)
147 f95b70    {
148 6c76dc      return linear_solve2(a, b, rhs.a, rhs.b, c, rhs.c);
149 cbb184    }
150 d41d8c
151 d41d8c    // Normalize line to c >= 0, a*a + b*b == 1. Only use with
    double.
152 050345    line normalize()
153 f95b70    {
154 22b5e2      double d = P(a, b).dist() * (c < 0 ? -1 : 1);
155 7c9abe      return line(a / d, b / d, c / d);
156 cbb184    }
157 2145c1  };
158 d41d8c
159 d41d8c  // Represents a circle by its center and radius. Mostly only
    works with double.
160 4fce64  template <class T>
161 0b1113  struct circle
162 f95b70  {
163 5dcf91    typedef point<T> P;
164 1ab228    P center;
```

```
165 c3df30    T r;
166 d41d8c
167 d41d8c    // Intersects circle with a given line. This does not work with
    integer types.
168 d41d8c    // If there is no intersection, returns 0 and retv is whatever.
169 d41d8c    // If intersection is a single point, returns 1 and retv is a
   pair of equal points.
170 d41d8c    // If intersection is two points, return 2 and retv is the two
   intersection points.
171 d41d8c    // Assume points are given in no particular order. If you
   really need it, should be leftmost first when looking from center of the
   circle.
172 ec2c6b    int intersect(line<T> l, pair<P, P> &retv)
173 f95b70    {
174 800175      l = l.normalize();
175 f543ca      l.c -= l.a * center.x + l.b * center.y; // Recenter so that
   we can consider circle center in origin.
176 18b956      P v(l.a, l.b);
177 cf8231      P p0 = v * l.c; // p0 is the point in the line closest to
   origin.
178 d41d8c
179 2d9566      if (p0.dist() > r + EPS) // No intersection.
180 bb30ba        return 0;
181 40b0e2      else if (p0.dist() > r - EPS) // dist in [r - EPS, r + EPS]
   -> single point intersection at p0.
182 f95b70      {
183 de0c90        retv = {p0, p0};
184 6a5530        return 1;
185 cbb184      }
186 d41d8c
187 85b09c      double d = sqrt(r * r - l.c * l.c); // d is distance from p0
   to the intersection points.
188 c4bf3f      retv = {center + p0 + v.normal() * d, center + p0 - v.normal
   () * d};
189 18b932      return 2;
190 cbb184    }
191 d41d8c
192 d41d8c    // Intersects circle with another circle. This does not work
   with integer types.
193 d41d8c    // This assumes the circles do not have the same center. Check
   this case if needed, can have 0 or infinite intersection points.
194 d41d8c    // If there is no intersection, returns 0 and retv is whatever.
195 d41d8c    // If intersection is a single point, returns 1 and retv is a
   pair of equal points.
196 d41d8c    // If intersection is two points, return 2 and retv is the two
   intersection points.
197 d41d8c    // Assume points are given in no particular order. If you
   really need it, should be leftmost first when looking from center of the
   rhs circle.
198 f2bab0    int intersect(circle rhs, pair<P, P> &retv)
```

```
199 f95b70    {
200 db42cd      rhs.center = rhs.center - center;
201 2adf3a      int num = rhs.intersect(line<T>(2 * rhs.center.x, 2 * rhs.
   center.y, rhs.center.x * rhs.center.x + rhs.center.y * rhs.center.y + r *
   r - rhs.r * rhs.r), retv);
202 2a6a69      retv.first = retv.first + center;
203 e34010      retv.second = retv.second + center;
204 fcc01b      return num;
205 cbb184    }
206 d41d8c
207 d41d8c    // Returns a pair of the two points on the circle whose tangent
    lines intersect p.
208 d41d8c    // If p lies within the circle NaN-points are returned. P is
   intended to be Point<double>.
209 d41d8c    // The first point is the one to the right as seen from the
   point p towards the circle.
210 163627    pair<P, P> tangents(P p)
211 f95b70    {
212 75ad6b      p = p - center;
213 28b73b      double k1 = r * r / p.dist2();
214 f84c08      double k2 = sqrt(k1 - k1 * k1);
215 a64b03      return {center + p * k1 + p.perp() * k2, center + p * k1 - p.
   perp() * k2};
216 cbb184    }
217 d41d8c
218 d41d8c    // TODO: find pair of tangent lines passing two circles.
219 2145c1  };
220 d41d8c
221 d41d8c  // The circumcircle of a triangle is the circle intersecting all
   three vertices.
222 d41d8c  // Returns the unique circle going through points A, B and C (
   given in no particular order).
223 d41d8c  // This assumes that the triangle has non-zero area.
224 d41d8c  // TODO: test specifically.
225 11308f  circle<double> circumcircle(const point<double> &A, const point<
   double> &B, const point<double> &C)
226 f95b70  {
227 b10dc9    circle<double> retv;
228 6d2418    point<double> a = C - B, b = C - A, c = B - A;
229 1d9440    retv.r = a.dist() * b.dist() * c.dist() / abs(c.cross(b)) / 2;
230 0d1695    retv.center = A + (b * c.dist2() - c * b.dist2()).perp() / b.
   cross(c) / 2;
231 6272cf    return retv;
232 cbb184  }
```

3.2 Graham Scan (convex hull)

```
1 d41d8c  /*
2 6ccc59    Solution for convex hull problem (minimum polygon covering a set
   of points) based on ordering ponts by angle.
```

```
3 3248ac    * Finds the subset of points in the convex hull in O(Nlog(N)).
4 687c39    * This version works if you either want intermediary points in
   segments or not (see comments delimited by //)
5 01b744    * This version works when all points are collinear
6 246d86    * This version works for repeated points if you add a label to
   struct, and use this label in overloaded +, - and =.
7 d41d8c
8 1d1558    Source: https://github.com/kth-competitive-programming/kactl/blob
   /master/content/contest/template.cpp
9 c4c9bd  */
10  d41d8c
11  2b74fa  #include<bits/stdc++.h>
12  d41d8c
13  ad1153  typedef long long ll;
14  d41d8c
15  ca417d  using namespace std;
16  d41d8c
17  67a100  template<typename T>
18  4befb0  struct point
19  f95b70  {
20  5dcf91    typedef point<T> P;
21  645c5d    T x, y;
22  d41d8c
23  571f13    explicit point(T x = 0, T y = 0) : x(x), y(y) {}
24  d41d8c    //Double version: bool operator<(P p) const { return  fabs(x -
   p.x) < EPS ? y < p.y : x < p.x; }
25  0d0d56    bool operator<(P p) const { return tie(x, y) < tie(p.x, p.y); }
26  d41d8c    //Double version: bool operator==(P p) const { return fabs(x -
   p.x) < EPS && fabs(y - p.y) < EPS; }
27  ec7475    bool operator==(P p) const { return tie(x, y) == tie(p.x, p.y);
    }
28  2798c7    P operator+(P p) const { return P(x + p.x, y + p.y); }
29  f681d2    T dist2() const { return x*x + y*y; }
30  40d57e    P operator-(P p) const { return P(x - p.x, y - p.y); }
31  57bee4    T dot(P p) const { return x * p.x + y * p.y; }
32  460881    T cross(P p) const { return x * p.y - y * p.x; }
33  b3fab9    T cross(P a, P b) const { return (a - *this).cross(b - *this);
   }
34  5b4ebf    long double dist() const { return sqrt((long double)dist2()); }
35  2145c1  };
36  d41d8c
37  d41d8c  /*Compara primeiro por angulo em relacao a origem e depois por
   distancia para a origem*/
38  67a100  template<typename T>
39  d74eff  bool cmp(point<T> a, point<T> b){
40  a9b570    if(a.cross(b) != 0)
41  c33606      return a.cross(b) > 0;
42  ba7b3a    return a.dist2() < b.dist2();
43  cbb184  }
44  d41d8c
```

```
45  67a100  template<typename T>
46  3c7876  vector<point<T> > CH(vector<point<T> > points){
47  d41d8c    /*Encontra pivo (ponto extremos que com ctz faz parte do CH)*/
48  95b799    point<T> pivot = points[0];
49  e409fb    for(auto p : points)
50  e01c07      pivot = min(pivot, p);
51  d41d8c
52  d41d8c    /*Desloca conjunto para pivo ficar na origem e ordena potos
    pelo angulo e distancia do pivo*/
53  9ac126    for(int i = 0; i < (int) points.size(); i++)
54  3010bd      points[i] = points[i] - pivot;
55  d41d8c
56  e2c4e0    sort(points.begin(), points.end(), cmp<ll>);
57  d41d8c
58  9ac126    for(int i = 0; i < (int) points.size(); i++)
59  eda5a9      points[i] = points[i] + pivot;
60  d41d8c
61  d41d8c    /*Ponto extra para fechar o poligono*/
62  36b3da    points.push_back(points[0]);
63  d41d8c
64  620533    vector<point<T> > ch;
65  d41d8c
66  b7f960    for(auto p : points){
67  d41d8c      /*Enquanto o proximo ponto gera uma curva para a direita,
    retira ultimo ponto atual*/
68  d41d8c      /*Segunda comparaÃğÃčo serve para caso especial de pontos
    colineares quando se quer eliminar os intermediarios*/
69  d41d8c      //Trocar terceira comparacao pra <= para discartar pontos do
    meio de arestas no ch
70  d41d8c      //Double: trocar terceira comparaÃğÃčo por < EPS (descarta
    pontos em arestas) ou < -EPS (mantem pinto em aresta
71  29fcb4      while(ch.size() > 1 && !(p == ch[ch.size() - 2]) && ch[ch.
    size() - 2].cross(ch[ch.size() - 1] , p) < 0)
72  9d9654        ch.pop_back();
73  d2ebaf      ch.push_back(p);
74  cbb184    }
75  d41d8c
76  d41d8c    /*Elimina ponto extra*/
77  9d9654    ch.pop_back();
78  d41d8c
79  66cc3c    return ch;
80  cbb184  }
81  d41d8c
82  e8d76f  int main(){
83  1a88fd    int n;
84  f4c120    scanf("%d", &n);
85  76374e    vector<point<ll> > p(n);
86  d41d8c
87  d41d8c    /*Le poligono*/
88  83008c    for(int i = 0; i < n; i++)
```

```
89  3daa4c      scanf("%lld %lld", &p[i].x, &p[i].y);
90  d41d8c
91  d41d8c    /*Encontra CH*/
92  680587    vector<point<ll> > ch = CH(p);
93  d41d8c
94  d41d8c    /*Imorime resultado*/
95  3b846e    printf("%d\n", (int)ch.size());
96  c857e7    for(int i = 0; i < (int)ch.size(); i++)
97  fc3047      printf("%d% %d\n", ch[i].x, ch[i].y);
98  d41d8c
99  cbb184  }
```

3.3 Min Enclosing Circle (randomized)

```
1 ad578e  #include "../2d/2d.cpp"
2 d41d8c
3 d41d8c  /*
4 744027    Minimum Enclosing Circle:
5 2d38cc      Given a list of points, returns a circle of minimum radius such
    that all given
6 2602de      points are within the circle.
7 0c4a3b      Runs in O(n) expected time (in practice 200 ms for 10^5 points)
   .
8 d41d8c
9 ca2095    Constraints:
10  99b71a      Non-empty list of points.
11  d41d8c
12  3db72f    Author: Arthur Pratti Dadalto
13  c4c9bd  */
14  d41d8c
15  e89126  #define point point<double>
16  0f3aa0  #define circle circle<double>
17  d41d8c
18  41ee07  circle min_enclosing_circle(vector<point> p)
19  f95b70  {
20  b4da45    shuffle(p.begin(), p.end(), mt19937(time(0)));
21  2e09de    point o = p[0];
22  76160f    double r = 0, eps = 1 + 1e-8;
23  fe16ed    for (int i = 0; i < sz(p); i++)
24  197ee7      if ((o - p[i]).dist() > r * eps)
25  f95b70      {
26  ba37a5        o = p[i], r = 0;
27  c791cd        for (int j = 0; j < i; j++)
28  f5972f          if ((o - p[j]).dist() > r * eps)
29  f95b70          {
30  d2b545            o = (p[i] + p[j]) / 2;
31  0657ce            r = (o - p[i]).dist();
32  674051            for (int k = 0; k < j; k++)
33  355d4d              if ((o - p[k]).dist() > r * eps)
34  f95b70              {
```

```
7fb807                o = circumcircle(p[i], p[j], p[k]).center;
0657ce                r = (o - p[i]).dist();
cbb184              }
cbb184            }
cbb184          }
d41d8c
645c1d      return {o, r};
cbb184    }
d41d8c
```

3.4 Min Enclosing Circle (ternary search)

```
ad578e  #include "../2d/2d.cpp"
2729c3  #include "../../misc/ternary_search/ternary_search_continuous.cpp"
d41d8c
d41d8c  /*
744027    Minimum Enclosing Circle:
2d38cc      Given a list of points, returns a circle of minimum radius such
    that all given
2602de      points are within the circle.
a1c921      Runs in O(n * log^2((top - bot) / eps)) (in practice 2.5s at
   best for 10^5 points).
d41d8c
ca2095    Constraints:
99b71a      Non-empty list of points.
d41d8c
b95cae    Usage:
ca9f24      The coordinates of the circle's center must be in the range [
   bot, top].
a09e29      eps specifies the precision of the result, but set it to a
   higher value
53006e      than necessary since the error in x affects the y value.
d41d8c
3db72f    Author: Arthur Pratti Dadalto
c4c9bd  */
d41d8c
e89126  #define point point<double>
0f3aa0  #define circle circle<double>
d41d8c
e1710f  circle min_enclosing_circle(const vector<point> &p, double bot =
   -1e9, double top = 1e9, double eps = 1e-9)
f95b70  {
1841a9    circle retv;
d41d8c
0a37af    auto f1 = [&](double x) {
d9991d      auto f2 = [&](double y)
f95b70      {
996834        double r = 0;
fe16ed        for (int i = 0; i < sz(p); i++)
62adf4          r = max(r, (p[i].x - x)*(p[i].x - x) + (p[i].y - y)*(p[i
```

```
   ].y - y));
34  4c1f3c         return r;
35  2145c1       };
36  410f57       retv.center.y = ternary_search(f2, bot, top, eps);
37  50ac50       return f2(retv.center.y);
38  2145c1     };
39  d41d8c
40  596ad7     retv.center.x = ternary_search(f1, bot, top, eps);
41  3b2a60     retv.r = sqrt(f1(retv.center.x));
42  d41d8c
43  6272cf     return retv;
44  cbb184   }
```

4 Graph

4.1 Biconnected Components

```
1 5d1131  #include "../../contest/header.hpp"
2 d41d8c
3 d41d8c  /*
4 399f8b    Finding bridges, articulation points and biconnected components
   in O(V + E):
5 5d1e77      A bridge is an edge whose removal splits the graph in two
   connected components.
6 8dd98b      An articulation point is a vertex whose removal splits the
   graph in two connected components.
7 d41d8c
8 8254c5      A biconnected component (or 2VCC) is a maximal subgraph where
   the removal of any vertex doesn't
9 44c916          make the subgraph disconnected. In other words, it is a
   maximal 2-vertex-connected (2VC) subgraph.
10  d41d8c
11  deb2a1          A 2-connected graph is a 2VC one, except that a---b is
   considered 2VC but not 2-connected.
12  d41d8c
13  3a585b      Useful theorems:
14  d41d8c
15  45e89c          A 2-edge connected (2EC) graph is a graph without bridges
   . Any 2-connected graph is also 2EC.
16  d41d8c
17  20dbb2          Let G be a graph on at least 2 vertices. The following
   propositions are equivalent:
18  d2a064              âĂć (i) G is 2-connected;
19  f24b9a              âĂć (ii) any two vertices are in a cycle; (a cycle
   can't repeat vertices)
20  160c93              âĂć (iii) any two edges are in a cycle and Îť(G) âĽě
   2;
21  52612e              âĂć (iv) for any three vertices x,y et z, there is a
   (x,z)-path containing y.
22  561b74          Let G be a graph on at least 3 vertices. The following
   propositions are equivalent:
23  ed0575              âĂć (i) G is 2-edge-connected;
24  85aaec              âĂć (ii) any edge is in a cycle;
25  d15d6b              âĂć (iii) any two edges are in a tour and Îť âĽě 1;
26  dc847c              âĂć (iv) any two vertices are in a tour (a tour can
   repeat vertices)
27  d41d8c
28  8c2af3      If G is 2-connected and not bipartite, all vertices belong to
    some odd cycle. And any two vertices are in a odd cycle (not really
   proven).
29  d41d8c
30  bcc5c9          If G is 2-edge-connected (proof by AC):
31  33bf43              For any two vertices x, y and one edge e, there is a
```

```
   (x, y)-walk containing e without repeating edges.
32  d41d8c
33  ab50d8          A graph admits a strongly connected orientation if and
   only if it is 2EC.
34  d3ea79          A strong orientation of a given bridgeless undirected
   graph may be found in linear time by performing
35  0ec987          a depth first search of the graph, orienting all edges in
    the depth first search tree away from the
36  d494fc          tree root, and orienting all the remaining edges (which
   must necessarily connect an ancestor and a
37  030955          descendant in the depth first search tree) from the
   descendant to the ancestor.
38  d41d8c
39  ca2095    Constraints:
40  b9aa54      ***undirected*** graph.
41  80b2d0      Vertices are labeled from 0 to n (inclusive).
42  568d46      Graph is connected (but for unconnected just replace single
   dfs call with a loop).
43  d41d8c
44  b95cae    Usage:
45  4436dc      Create the struct setting the starting vertex (a), the
   maximum vertex label (n),
46  e1993f        the graph adjacency list (graph) and a callback f to apply
   on the biconnected components.
47  f8f25e      Afterwards, art[i] == true if i is an articulation point.
48  e9a79e      If the pair {a, i} is on the bridges list, then the edge {a,
   graph[a][i]} is a bridge.
49  ccfd29          The callback must receive a vector of edges {a, b} that
   are in the same biconnected component.
50  a32c3f      Remember that for a single vertex, the biconnected callback
   will not be called.
51  d41d8c
52  e152b4      Sample Usage:
53  0ec6ee          auto rdm = apb(1, n, graph, [&](vector<pii> v){
54  f4ecd5      set<int> s;
55  9ad08e      for (int i = 0; i < sz(v); i++)
56  f95b70      {
57  f19ef4        s.insert(v[i].first);
58  0858fa        s.insert(v[i].second);
59  cbb184      }
60  d41d8c
61  0fe299      ans = max(ans, sz(s));
62  c0c97e        });
63  c4c9bd  */
64  d41d8c
65  f117a6  struct apb
66  f95b70  {
67  9cf2b9  vector<int> *graph;
68  9cf143  vector<bool> art;
69  c9001f  vector<int> num /* dfs order of vertices starting at 1 */, low;
```

```
70  c83796  vector<pii> bridges;
71  91936b  vector<pii> st;
72  53e65f  int id;
73  d41d8c
74  044d82  template<class F>
75  09caad  apb(int a, int n, vector<int> graph[], const F &f) : graph(graph)
    , art(n + 1, false), num(n + 1), low(n + 1)
76  f95b70  {
77  0f6720    id = 1;
78  ccac4e    dfs(a, a, f);
79  cbb184  }
80  d41d8c
81  044d82  template<class F>
82  dc584b  void dfs(int a, int p, const F &f)
83  f95b70  {
84  7be506    low[a] = num[a] = id++;
85  34863b    int comp = 0;
86  d41d8c
87  1429ef    for (int i = 0; i < sz(graph[a]); i++)
88  f95b70    {
89  b7a810      if (num[graph[a][i]] == 0)
90  f95b70      {
91  d40410              int si = sz(st);
92  f309f5        comp++;
93  8ece2e              st.push_back({a, graph[a][i]}); // Tree edge.
94  d41d8c
95  fc5941        dfs(graph[a][i], a, f);
96  085d64        low[a] = min(low[a], low[graph[a][i]]);
97  d41d8c
98  bb63a0        if (low[graph[a][i]] >= num[a])
99  f95b70              {
100 558f81                  if (a != 1)
101 016392              art[a] = true;
102 d41d8c
103 b91456                  f(vector<pii>(st.begin() + si, st.end()));
104 901921                  st.resize(si);
105 cbb184              }
106 d41d8c
107 0e9ddb        if (low[graph[a][i]] > num[a])
108 b3cacb          bridges.push_back({a, i});
109 cbb184      }
110 624580      else if (graph[a][i] != p && num[graph[a][i]] < num[a]) //
    Back edge.
111 f95b70          {
112 066898              low[a] = min(low[a], num[graph[a][i]]);
113 8ece2e              st.push_back({a, graph[a][i]});
114 cbb184          }
115 cbb184    }
116 d41d8c
117 85e3a2    if (a == p && comp > 1)
```

```
118 016392      art[a] = true;
119 cbb184  }
120 2145c1  };
121 d41d8c
```

4.2 Bipartite Matching (Hopcroft Karp)

```
1 2b74fa  #include <bits/stdc++.h>
2 ca417d  using namespace std;
3 d41d8c
4 d41d8c  /*
5 ec23c9    Hopcroft-Karp:
6 eaeddf      Bipartite Matching O(sqrt(V)E)
7 d41d8c
8 ca2095    Constraints:
9 998cc9      Vertices are labeled from 1 to l + r (inclusive).
10  682ff0      DO NOT use vertex 0.
11  968b86      Vertices 1 to l belong to left partition.
12  a6a4c4      Vertices l + 1 to l + r belong to right partition.
13  d41d8c
14  b95cae    Usage:
15  d86132      Set MAXV if necessary.
16  70636b      Call init passing l and r.
17  0f3b71      Add edges to the graph from left side to right side.
18  5263f1      Call hopcroft to get the matching size.
19  a0da8e      Then, each vertex v has its pair indicated in p[v] (or 0 for
   not paired).
20  c4c9bd  */
21  d41d8c
22  dde07b  namespace hopcroft
23  f95b70  {
24  998014  const int inf = 0x3f3f3f3f;
25  ed5ed2  const int MAXV = 112345;
26  d41d8c
27  3098d4  vector<vector<int>> graph;
28  0a3d29  int d[MAXV], q[MAXV], p[MAXV], l, r;
29  d41d8c
30  4025e1  void init(int _l, int _r)
31  f95b70  {
32  0ebd66    l = _l, r = _r;
33  2213c3    graph = vector<vector<int>>(l + r + 1);
34  cbb184  }
35  d41d8c
36  6a1cf9  bool bfs()
37  f95b70  {
38  18753f    int qb = 0, qe = 0;
39  4f2bde    memset(d, 0x3f, sizeof(int) * (l + 1));
40  a89ba9    for (int i = 1; i <= l; i++)
41  8b3877      if (p[i] == 0)
42  248d2f        d[i] = 0, q[qe++] = i;
```

```
43  d41d8c
44  2caa87    while (qb < qe)
45  f95b70    {
46  e8e8a0      int a = q[qb++];
47  0087d7      if (a == 0)
48  8a6c14        return true;
49  c4fff3      for (int i = 0; i < graph[a].size(); i++)
50  68367c        if (d[p[graph[a][i]]] == inf)
51  a8cd28          d[q[qe++] = p[graph[a][i]]] = d[a] + 1;
52  cbb184    }
53  d41d8c
54  d1fe4d    return false;
55  cbb184  }
56  d41d8c
57  0752c9  bool dfs(int a)
58  f95b70  {
59  0087d7    if (a == 0)
60  8a6c14      return true;
61  c4fff3    for (int i = 0; i < graph[a].size(); i++)
62  7d85df      if (d[a] + 1 == d[p[graph[a][i]]])
63  a2f815        if (dfs(p[graph[a][i]]))
64  f95b70        {
65  460f0a          p[a] = graph[a][i];
66  51e040          p[graph[a][i]] = a;
67  8a6c14          return true;
68  cbb184        }
69  d41d8c
70  343737    d[a] = inf;
71  d1fe4d    return false;
72  cbb184  }
73  d41d8c
74  68fd9d  int hopcroft()
75  f95b70  {
76  9e3790    memset(p, 0, sizeof(int) * (l + r + 1));
77  fc833c    int matching = 0;
78  d594a7    while (bfs())
79  f95b70    {
80  a89ba9      for (int i = 1; i <= l; i++)
81  8b3877        if (p[i] == 0)
82  57e7a2          if (dfs(i))
83  730cbb            matching++;
84  cbb184    }
85  d41d8c
86  2afcbe    return matching;
87  cbb184  }
88  cbb184  } // namespace hopcroft
```

4.3 Bridges/Articulation Points

```
1 5d1131  #include "../../contest/header.hpp"
```

```
d41d8c
d41d8c  /*
62784d    Finding bridges and articulation points in O(V + E):
5d1e77      A bridge is an edge whose removal splits the graph in two connected components.
8dd98b      An articulation point is a vertex whose removal splits the graph in two connected components.
8b8ace      This can also be adapted to generate the biconnected components of a graph, since the
14a784      articulation points split components.
d41d8c
d41d8c
ca2095    Constraints:
b9aa54      ***undirected*** graph.
80b2d0      Vertices are labeled from 0 to n (inclusive).
1e6120      Graph is connected (otherwise it doesn't make sense).
d41d8c
b95cae    Usage:
668bef      Create the struct setting the starting vertex (a), the maximum vertex label (n)
8acebe      and the graph adjacency list (graph).
6ffc91      Aftewards, art[i] == true if i is an articulation point.
e9a79e      If the pair {a, i} is on the bridges list, then the edge {a, graph[a][i]} is a bridge.
c4c9bd  */
d41d8c
f117a6  struct apb
f95b70  {
9cf2b9  vector<int> *graph;
9cf143  vector<bool> art;
c9001f  vector<int> num /* dfs order of vertices starting at 1 */, low;
c83796  vector<pii> bridges;
53e65f  int id;
d41d8c
4dc736  apb(int a, int n, vector<int> graph[]) : graph(graph), art(n + 1, false), num(n + 1), low(n + 1)
f95b70  {
0f6720    id = 1;
bb407e    dfs(a, a);
cbb184  }
d41d8c
69c421  void dfs(int a, int p)
f95b70  {
7be506    low[a] = num[a] = id++;
34863b    int comp = 0;
d41d8c
c4fff3    for (int i = 0; i < graph[a].size(); i++)
f95b70    {
b7a810      if (num[graph[a][i]] == 0)
f95b70      {
```

```
46  f309f5        comp++;
47  783129        dfs(graph[a][i], a);
48  085d64        low[a] = min(low[a], low[graph[a][i]]);
49  d41d8c
50  b28b5f        if (a != 1 && low[graph[a][i]] >= num[a])
51  016392          art[a] = true;
52  d41d8c
53  0e9ddb        if (low[graph[a][i]] > num[a])
54  b3cacb          bridges.push_back({a, i});
55  cbb184      }
56  2cae7a      else if (graph[a][i] != p && num[graph[a][i]] < low[a])
57  ed0d8a        low[a] = num[graph[a][i]];
58  cbb184    }
59  d41d8c
60  85e3a2    if (a == p && comp > 1)
61  016392      art[a] = true;
62  cbb184  }
63  2145c1  };
64  d41d8c
```

4.4 Max Flow (Dinic)

```
1 2b74fa  #include <bits/stdc++.h>
2 ca417d  using namespace std;
3 d41d8c
4 d41d8c  /*
5 908d2f    Dinic:
6 67cbe4      Max-flow O(V^2E)
7 eaeddf      Bipartite Matching O(sqrt(V)E)
8 d41d8c
9 ca2095    Constraints:
10  80b2d0      Vertices are labeled from 0 to n (inclusive).
11  8f4ce8      Edge capacities must fit int (flow returned is long long).
12  d41d8c
13  b95cae    Usage:
14  d86132      Set MAXV if necessary.
15  148d9c      Call init passing n, the source and the sink.
16  2d6398      Add edges to the graph by calling put_edge(_undirected).
17  bb3825      Call max_flow to get the total flow. Then, individual edge
    flows can be retrieved in the graph.
18  22c3c2      Note that flow will be negative in return edges.
19  c4c9bd  */
20  d41d8c
21  82657b  namespace dinic
22  f95b70  {
23  729806  struct edge
24  f95b70  {
25  bf6256    int dest, cap, re, flow;
26  2145c1  };
27  d41d8c
```

```
28  998014  const int inf = 0x3f3f3f3f;
29  8550b5  const int MAXV = 312345;
30  d41d8c
31  8a367b  int n, s, t, d[MAXV], q[MAXV], next[MAXV];
32  d8f9f2  vector<vector<edge>> graph;
33  d41d8c
34  bc6f23  void init(int _n, int _s, int _t)
35  f95b70  {
36  c992e9    n = _n, s = _s, t = _t;
37  b72d19    graph = vector<vector<edge>>(n + 1);
38  cbb184  }
39  d41d8c
40  7c85eb  void put_edge(int u, int v, int cap)
41  f95b70  {
42  506964    graph[u].push_back({v, cap, (int)graph[v].size(), 0});
43  68ec95    graph[v].push_back({u, 0, (int)graph[u].size() - 1, 0});
44  cbb184  }
45  d41d8c
46  d6a592  void put_edge_undirected(int u, int v, int cap)
47  f95b70  {
48  506964    graph[u].push_back({v, cap, (int)graph[v].size(), 0});
49  fce495    graph[v].push_back({u, cap, (int)graph[u].size() - 1, 0});
50  cbb184  }
51  d41d8c
52  6a1cf9  bool bfs()
53  f95b70  {
54  18753f    int qb = 0, qe = 0;
55  3c6658    q[qe++] = s;
56  98fde3    memset(d, 0x3f, sizeof(int) * (n + 1));
57  d66185    d[s] = 0;
58  2caa87    while (qb < qe)
59  f95b70    {
60  e8e8a0      int a = q[qb++];
61  c9a55a      if (a == t)
62  8a6c14        return true;
63  3352c6      for (int i = 0; i < (int)graph[a].size(); i++)
64  f95b70      {
65  10e42b        edge &e = graph[a][i];
66  d948dd        if (e.cap - e.flow > 0 && d[e.dest] == inf)
67  f4063b          d[q[qe++] = e.dest] = d[a] + 1;
68  cbb184      }
69  cbb184    }
70  d41d8c
71  d1fe4d    return false;
72  cbb184  }
73  d41d8c
74  1a19d4  int dfs(int a, int flow)
75  f95b70  {
76  c9a55a    if (a == t)
77  99d2e8      return flow;
```

```
78  10647a    for (int &i = next[a]; i < (int)graph[a].size(); i++)
79  f95b70    {
80  10e42b      edge &e = graph[a][i];
81  c6fb85      if (d[a] + 1 == d[e.dest] && e.cap - e.flow > 0)
82  f95b70      {
83  5f308a        int x = dfs(e.dest, min(flow, e.cap - e.flow));
84  5f75db        if (x == 0)
85  5e2bd7          continue;
86  7f9751        e.flow += x;
87  4c55a5        graph[e.dest][e.re].flow -= x;
88  ea5659        return x;
89  cbb184      }
90  cbb184    }
91  d41d8c
92  343737    d[a] = inf;
93  bb30ba    return 0;
94  cbb184  }
95  d41d8c
96  afa2f7  long long max_flow()
97  f95b70  {
98  f013d3    long long total_flow = 0;
99  d594a7    while (bfs())
100 f95b70    {
101 ba90c2      memset(next, 0, sizeof(int) * (n + 1));
102 60616b      while (int path_flow = dfs(s, inf))
103 a0d8d9        total_flow += path_flow;
104 cbb184    }
105 d41d8c
106 793f63    return total_flow;
107 cbb184  }
108 cbb184  } // namespace dinic
```

4.5 Max Flow (Dinic w/ Scaling)

```
1 2b74fa  #include <bits/stdc++.h>
2 ca417d  using namespace std;
3 d41d8c
4 d41d8c  /*
5 3678e9    Dinic with Scaling:
6 476fd1      Max-flow O(VE * log(MAX_CAP)), but usually slower than regular
   Dinic.
7 d41d8c
8 ca2095    Constraints:
9 80b2d0      Vertices are labeled from 0 to n (inclusive).
10  8f4ce8      Edge capacities must fit int (flow returned is long long).
11  d41d8c
12  b95cae    Usage:
13  d86132      Set MAXV if necessary.
14  148d9c      Call init passing n, the source and the sink.
15  2d6398      Add edges to the graph by calling put_edge(_undirected).
```

```
16 bb3825     Call max_flow to get the total flow. Then, individual edge
   flows can be retrieved in the graph.
17 22c3c2     Note that flow will be negative in return edges.
18 c4c9bd */
19 d41d8c
20 82657b namespace dinic
21 f95b70 {
22 729806 struct edge
23 f95b70 {
24 bf6256   int dest, cap, re, flow;
25 2145c1 };
26 d41d8c
27 998014 const int inf = 0x3f3f3f3f;
28 8550b5 const int MAXV = 312345;
29 d41d8c
30 19c361 int n, s, t, lim, d[MAXV], q[MAXV], next[MAXV];
31 d8f9f2 vector<vector<edge>> graph;
32 d41d8c
33 bc6f23 void init(int _n, int _s, int _t)
34 f95b70 {
35 c992e9   n = _n, s = _s, t = _t;
36 b72d19   graph = vector<vector<edge>>(n + 1);
37 cbb184 }
38 d41d8c
39 7c85eb void put_edge(int u, int v, int cap)
40 f95b70 {
41 506964   graph[u].push_back({v, cap, (int)graph[v].size(), 0});
42 68ec95   graph[v].push_back({u, 0, (int)graph[u].size() - 1, 0});
43 cbb184 }
44 d41d8c
45 d6a592 void put_edge_undirected(int u, int v, int cap)
46 f95b70 {
47 506964   graph[u].push_back({v, cap, (int)graph[v].size(), 0});
48 fce495   graph[v].push_back({u, cap, (int)graph[u].size() - 1, 0});
49 cbb184 }
50 d41d8c
51 6a1cf9 bool bfs()
52 f95b70 {
53 18753f   int qb = 0, qe = 0;
54 3c6658   q[qe++] = s;
55 98fde3   memset(d, 0x3f, sizeof(int) * (n + 1));
56 d66185   d[s] = 0;
57 2caa87   while (qb < qe)
58 f95b70   {
59 e8e8a0     int a = q[qb++];
60 c9a55a     if (a == t)
61 8a6c14       return true;
62 3352c6     for (int i = 0; i < (int)graph[a].size(); i++)
63 f95b70     {
64 10e42b       edge &e = graph[a][i];
```

```
65  21aca9        if (e.cap - e.flow >= lim && d[e.dest] == inf)
66  f4063b          d[q[qe++] = e.dest] = d[a] + 1;
67  cbb184      }
68  cbb184    }
69  d41d8c
70  d1fe4d    return false;
71  cbb184  }
72  d41d8c
73  1a19d4  int dfs(int a, int flow)
74  f95b70  {
75  c9a55a    if (a == t)
76  99d2e8      return flow;
77  10647a    for (int &i = next[a]; i < (int)graph[a].size(); i++)
78  f95b70    {
79  10e42b      edge &e = graph[a][i];
80  cbf046      if (d[a] + 1 == d[e.dest] && e.cap - e.flow >= lim /* >= 1 ?
    */)
81  f95b70      {
82  5f308a        int x = dfs(e.dest, min(flow, e.cap - e.flow));
83  5f75db        if (x == 0)
84  5e2bd7          continue;
85  7f9751        e.flow += x;
86  4c55a5        graph[e.dest][e.re].flow -= x;
87  ea5659        return x;
88  cbb184      }
89  cbb184    }
90  d41d8c
91  343737    d[a] = inf;
92  bb30ba    return 0;
93  cbb184  }
94  d41d8c
95  afa2f7  long long max_flow()
96  f95b70  {
97  f013d3    long long total_flow = 0;
98  aab413      for (lim = (1 << 30); lim >= 1; lim >>= 1)
99  d594a7      while (bfs())
100 f95b70      {
101 ba90c2        memset(next, 0, sizeof(int) * (n + 1));
102 60616b        while (int path_flow = dfs(s, inf))
103 a0d8d9          total_flow += path_flow;
104 cbb184      }
105 d41d8c
106 793f63    return total_flow;
107 cbb184  }
108 cbb184  } // namespace dinic
```

4.6 Min Cost Max Flow

```
1 2b74fa  #include <bits/stdc++.h>
2 ca417d  using namespace std;
```

```
3 d41d8c
4 d41d8c  /*
5 dfc480    Min-Cost Max-Flow: O(V^2E^2)
6 078f0d      Finds the maximum flow of minimum cost.
7 d41d8c
8 ca2095    Constraints:
9 80b2d0      Vertices are labeled from 0 to n (inclusive).
10  247575      Edge cost and capacities must fit int (flow and cost returned
    are long long).
11  75ef18      Edge Cost must be non-negative.
12  d41d8c
13  b95cae    Usage:
14  d86132      Set MAXV if necessary.
15  148d9c      Call init passing n, the source and the sink.
16  909583      Add edges to the graph by calling put_edge.
17  553d3e      Call mincost_maxflow to get the total flow and its cost (in
   this order).
18  0e374d      Individual edge flows can be retrieved in the graph. Note
   that flow will be negative in return edges.
19  c4c9bd  */
20  d41d8c
21  ad1153  typedef long long ll;
22  d29b14  typedef pair<long long, long long> pll;
23  d41d8c
24  e3de19  namespace mcmf
25  f95b70  {
26  729806  struct edge
27  f95b70  {
28  60f183    int dest, cap, re, cost, flow;
29  2145c1  };
30  d41d8c
31  ed5ed2  const int MAXV = 112345;
32  6a4c6c  const ll infll = 0x3f3f3f3f3f3f3f3fLL;
33  998014  const int inf = 0x3f3f3f3f;
34  d41d8c
35  128c92  int n, s, t, p[MAXV], e_used[MAXV];
36  97e5bb  bool in_queue[MAXV];
37  c97378  ll d[MAXV];
38  d41d8c
39  d8f9f2  vector<vector<edge>> graph;
40  d41d8c
41  bc6f23  void init(int _n, int _s, int _t)
42  f95b70  {
43  c992e9    n = _n, s = _s, t = _t;
44  b72d19    graph = vector<vector<edge>>(n + 1);
45  cbb184  }
46  d41d8c
47  a4abfa  void put_edge(int u, int v, int cap, int cost)
48  f95b70  {
49  bd3e8b    graph[u].push_back({v, cap, (int)graph[v].size(), cost, 0});
```

```
50  2b8b8d    graph[v].push_back({u, 0, (int)graph[u].size() - 1, -cost, 0});
51  cbb184  }
52  d41d8c
53  b34984  bool spfa()
54  f95b70  {
55  664c61    memset(in_queue, 0, sizeof(bool) * (n + 1));
56  9eef50    memset(d, 0x3f, sizeof(ll) * (n + 1));
57  26a528    queue<int> q;
58  d66185    d[s] = 0;
59  e2828b    p[s] = s;
60  08bec3    q.push(s);
61  ee6bdd    while (!q.empty())
62  f95b70    {
63  0930a5      int a = q.front();
64  833270      q.pop();
65  e7249b      in_queue[a] = false;
66  d41d8c
67  c4fff3      for (int i = 0; i < graph[a].size(); i++)
68  f95b70      {
69  10e42b        edge &e = graph[a][i];
70  6fa321        if (e.cap - e.flow > 0 && d[e.dest] > d[a] + e.cost)
71  f95b70        {
72  3bf598          d[e.dest] = d[a] + e.cost;
73  6d6530          p[e.dest] = a;
74  183d83          e_used[e.dest] = i;
75  27788c          if (!in_queue[e.dest])
76  b34293            q.push(e.dest);
77  04f0f7          in_queue[e.dest] = true;
78  cbb184        }
79  cbb184      }
80  cbb184    }
81  d41d8c
82  d1cd45    return d[t] < infll;
83  cbb184  }
84  d41d8c
85  99658d  pll mincost_maxflow()
86  f95b70  {
87  f04b2a    pll retv = pll(0, 0);
88  d9383f    while (spfa())
89  f95b70    {
90  e98031      int x = inf;
91  c9b315      for (int i = t; p[i] != i; i = p[i])
92  d4a316        x = min(x, graph[p[i]][e_used[i]].cap - graph[p[i]][e_used[
    i]].flow);
93  c9b315      for (int i = t; p[i] != i; i = p[i])
94  dc731d        graph[p[i]][e_used[i]].flow += x, graph[i][graph[p[i]][
    e_used[i]].re].flow -= x;
95  d41d8c
96  75907a      retv.first += x;
97  1be465      retv.second += x * d[t];
```

```
98  cbb184    }
99  d41d8c
100 6272cf    return retv;
101 cbb184  }
102 cbb184  } // namespace mcmf
```

4.7 Heavy-Light Decomposition

```
1  2b74fa  #include<bits/stdc++.h>
2  d41d8c
3  ca417d  using namespace std;
4  d41d8c
5  eed838  #define ll long long
6  efe13e  #define pb push_back
7  d41d8c
8  3a6c63  typedef vector<ll> vll;
9  990dd5  typedef vector<int> vi;
10  d41d8c
11  e06cc0  #define MAXN 100010
12  d41d8c
13  d41d8c  //Vetor que guarda a arvore
14  698e25  vector<vi> adj;
15  d41d8c
16  9e6e6d  int subsize[MAXN], parent[MAXN];
17  d41d8c  //Inciar chainHead com -1; e chainSize e chainNo com 0.
18  080553  int chainNo = 0, chainHead[MAXN], chainPos[MAXN], chainInd[MAXN],
    chainSize[MAXN];
19  42a605  void hld(int cur){
20  cb42fb    if(chainHead[chainNo] == -1)
21  6591fe      chainHead[chainNo] = cur;
22  d41d8c
23  3a4605    chainInd[cur] = chainNo;
24  220e91    chainPos[cur] = chainSize[chainNo];
25  6f00fb    chainSize[chainNo]++;
26  d41d8c
27  89108d    int ind = -1, mai = -1;
28  9d9afd    for(int i = 0; i < (int)adj[cur].size(); i++){
29  9ff2fa      if(adj[cur][i] != parent[cur] && subsize[adj[cur][i]] > mai){
30  31fcc6        mai = subsize[adj[cur][i]];
31  b9b7e9        ind  = i;
32  cbb184      }
33  cbb184    }
34  d41d8c
35  27d206    if(ind >= 0)
36  f23581      hld(adj[cur][ind]);
37  d41d8c
38  e506c6    for(int i = 0; i < (int)adj[cur].size(); i++)
39  6f7286      if(adj[cur][i] != parent[cur] && i != ind){
40  959ef6        chainNo++;
41  270563        hld(adj[cur][i]);
```

```
42  cbb184      }
43  cbb184  }
44  d41d8c
45  d41d8c  //usar LCA para garantir que v eh pai de u!!
46  f179f7  ll query_up(int u, int v){
47  c20c7b    int uchain = chainInd[u], vchain = chainInd[v];
48  bdd5ea    ll ans = 0LL;
49  d41d8c
50  31e3cd    while(1){
51  f523c5      if(uchain == vchain){
52  d41d8c        //Query deve ir de chainPos[i] ate chainPos[v]
53  7d2150        ll cur = /*sum(chainPos[u], uchain) - (chainPos[u] == 0? 0
    LL : sum(chainPos[v] - 1, vchain))*/;
54  d133d8        ans += cur;
55  c2bef1        break;
56  cbb184      }
57  d41d8c
58  d41d8c      //Query deve ir de chainPos[i] ate o fim da estrutura
59  d41d8c      //ll cur = sum(chainPos[u], uchain);
60  d133d8      ans += cur;
61  a258cd      u = chainHead[uchain];
62  8039e1      u = parent[u];
63  cabc24      uchain = chainInd[u];
64  cbb184    }
65  ba75d2    return ans;
66  cbb184  }
67  d41d8c
68  b7aa64  int dfs0(int pos, int prev = -1){
69  c92501    int res  = 1;
70  97817b    for(int i = 0; i < (int)adj[pos].size(); i++){
71  ec49a3      int nx  = adj[pos][i];
72  773904      if(nx != prev){
73  3f20e3        res += dfs0(nx, pos);
74  522845        parent[nx] = pos;
75  cbb184      }
76  cbb184    }
77  a1881c    return subsize[pos] = res;
78  cbb184  }
79  d41d8c
80  0b8977  int main()
81  f95b70  {
82  d41d8c    //Salvar arvore em adj
83  d41d8c
84  d41d8c    //Inicializa estrutura de dados
85  b75143    memset(chainHead, -1, sizeof(chainHead));
86  d41d8c
87  d41d8c    //Ou 0, se for o no raiz
88  bf6cde    dfs0(1);
89  bac429    hld(1);
90  d41d8c
```

```
91  d41d8c    //Inicializar estruturas usadas
92  cbb184  }
```

4.8 Strongly Connected Components

```
1 5d1131  #include "../../contest/header.hpp"
2 d41d8c
3 d41d8c  /*
4 eaba86    Strongly connected components in O(V + E):
5 970b0b      Finds all strongly connected components of a graph.
6 3f48a9          A strongly connected component is a maximal set of vertices
    such that
7 de0185          every vertex can reach every other vertex in the component.
8 0a2f52          The graph where the SCCs are considered vertices is a DAG.
9 d41d8c
10  ca2095    Constraints:
11  000269      Vertices are labeled from 1 to n (inclusive).
12  d41d8c
13  b95cae    Usage:
14  ee0d06      Create the struct setting the maximum vertex label (n) and
   the graph adjacency list (graph).
15  862124      Aftewards, ncomp has the number of SCCs in the graph and scc[
   i] indicates the SCC i
16  67d138          belongs to (1 <= scc[i] <= ncomp).
17  d41d8c
18  38224f          sorted is a topological ordering of the graph, byproduct
   of the algorithm.
19  484a00          if edge a -> b exists, a appears before b in the sorted
   list.
20  c4c9bd  */
21  d41d8c
22  d41d8c
23  73e60a  struct scc_decomp
24  f95b70  {
25  9cf2b9      vector<int> *graph;
26  00b6a0      vector<vector<int>> tgraph;
27  1b013f      vector<int> scc;
28  1ee615      vector<bool> been;
29  8d35d1      int ncomp;
30  2035f3      list<int> sorted;
31  d41d8c
32  4875fb      scc_decomp(int n, vector<int> graph[]) : graph(graph), tgraph
   (n + 1), scc(n + 1, 0), been(n + 1, false), ncomp(0)
33  f95b70      {
34  5359f3          for (int i = 1; i <= n; i++)
35  6376e8              for (int j = 0; j < graph[i].size(); j++)
36  14234d                  tgraph[graph[i][j]].push_back(i);
37  d41d8c
38  5359f3          for (int i = 1; i <= n; i++)
39  018df6        if(!been[i])
```

```
40  1e5da3          dfs(i);
41  d41d8c
42  16ef86      for(int a : sorted)
43  f49735        if(scc[a] == 0)
44  f95b70        {
45  a8f1f2                  ncomp++;
46  4dd966          dfst(a);
47  cbb184        }
48  cbb184      }
49  d41d8c
50  0cbab3      void dfs(int a)
51  f95b70      {
52  1689c6          been[a] = true;
53  c4fff3          for(int i = 0; i < graph[a].size(); i++)
54  b0c443              if(!been[graph[a][i]])
55  fded7f                  dfs(graph[a][i]);
56  ddbf66          sorted.push_front(a);
57  cbb184      }
58  d41d8c
59  9b760a      void dfst(int a)
60  f95b70      {
61  1689c6          been[a] = true;
62  d28fcc          scc[a] = ncomp;
63  9c28b7          for(int i = 0; i < tgraph[a].size(); i++)
64  c480c5              if(scc[tgraph[a][i]] == 0)
65  caa482                  dfst(tgraph[a][i]);
66  cbb184      }
67  2145c1  };
```

5 Misc

5.1 DP Optimization - Binary Search

```
d41d8c  // https://codeforces.com/contest/321/problem/E
d41d8c
5d1131  #include "../../contest/header.hpp"
d41d8c
d41d8c  /*
4e8f0c    Binary Search Optimization for DP:
6aaf9d      Optimizes dp of the form (or similar) dp[i][j] = min_{k < i}(dp[k][j-1] + c(k + 1, i)).
78d4c1      The classical case is a partitioning dp, where k determines the break point for the next partition.
0467c8      In this case, i is the number of elements to partition and j is the number of partitions allowed.
 d41d8c
 537168       Let opt[i][j] be the values of k which minimize the function. (in case of tie, choose the smallest)
 765123       To apply this optimization, you need opt[i][j] <= opt[i+1][j].
 6c4b0a       That means the when you add an extra element (i + 1), your partitioning choice will not be to include more elements
 efc594       than before (e.g. will no go from choosing [k, i] to [k-1, i+1]).
 242554       This is usually intuitive by the problem details.
 d41d8c
 4d883e       Time goes from O(n^2m) to O(nm log n).
 d41d8c
 895144       To apply try to write the dp in the format above and verify if the property holds.
 d41d8c
 3db72f     Author: Arthur Pratti Dadalto
 c4c9bd  */
 d41d8c
 3494e9  #define MAXN 4123
 fc36c1  #define MAXM 812
 d41d8c
 14e0a7  int n, m;
 1590eb  int u[MAXN][MAXN];
 2bbe6d  int tab[MAXN][MAXM];
 d41d8c
 65a7b7  inline int c(int i, int j)
 f95b70  {
 229880    return (u[j][j] - u[j][i - 1] - u[i - 1][j] + u[i - 1][i - 1]) / 2;
 cbb184  }
 d41d8c
 d41d8c  // This is responsible for computing tab[l...r][j], knowing that opt[l...r][j] is in range [low_opt...high_opt]
```

```
37  30d71a  void compute(int j, int l, int r, int low_opt, int high_opt)
38  f95b70  {
39  c30a4b    int mid = (l + r) / 2, opt = -1; // mid is equivalent to i in
    the original dp.
40  d41d8c
41  7222d3    tab[mid][j] = inf;
42  0e2f2c    for (int k = low_opt; k <= high_opt && k < mid; k++)
43  6f6e42      if (tab[k][j - 1] + c(k + 1, mid) < tab[mid][j])
44  f95b70      {
45  451068        tab[mid][j] = tab[k][j - 1] + c(k + 1, mid);
46  613f3c        opt = k;
47  cbb184      }
48  d41d8c
49  d41d8c    // New bounds on opt for other pending computation.
50  42c8a1    if (l <= mid - 1)
51  c7dd31      compute(j, l, mid - 1, low_opt, opt);
52  8b4e40    if (mid + 1 <= r)
53  8aa379      compute(j, mid + 1, r, opt, high_opt);
54  cbb184  }
55  d41d8c
56  13a4b1  int main(void)
57  f95b70  {
58  d69917    scanf("%d %d", &n, &m);
59  5359f3    for (int i = 1; i <= n; i++)
60  947790      for (int j = 1; j <= n; j++)
61  f95b70      {
62  433ab9        getchar();
63  512e3d        u[i][j] = getchar() - '0';
64  cbb184      }
65  d41d8c
66  5359f3    for (int i = 1; i <= n; i++)
67  947790      for (int j = 1; j <= n; j++)
68  a10370        u[i][j] += u[i - 1][j] + u[i][j - 1] - u[i - 1][j - 1];
69  d41d8c
70  5359f3    for (int i = 1; i <= n; i++)
71  5c5410      tab[i][0] = inf;
72  d41d8c
73  d41d8c    // Original dp
74  d41d8c    // for (int i = 1; i <= n; i++)
75  d41d8c    //   for (int j = 1; j <= m; j++)
76  d41d8c    //   {
77  d41d8c    //     tab[i][j] = inf;
78  d41d8c    //     for (int k = 0; k < i; k++)
79  d41d8c    //       tab[i][j] = min(tab[i][j], tab[k][j-1] + c(k + 1,i);
80  d41d8c    //   }
81  d41d8c
82  2e2a5d    for (int j = 1; j <= m; j++)
83  fdaa69      compute(j, 1, n, 0, n - 1);
84  d41d8c
85  721eeb    cout << tab[n][m] << endl;
```

```
86  cbb184  }
```

5.2 DP Optimization - CHT

```
1 d41d8c  // https://codeforces.com/contest/319/problem/C
2 d41d8c
3 ad67d1  #include "../../data_structures/line_container/line_container.cpp"
4 d41d8c
5 d41d8c  /*
6 082e10    Convex Hull Trick for DP:
7 366d02      Transforms dp of the form (or similar) dp[i] = min_{j < i}(dp[j
   ] + b[j] * a[i]).
8 ab5453      Time goes from O(n^2) to O(n log n), if using online line
   container, or O(n) if
9 56c021      lines are inserted in order of slope and queried in order of x.
10  d41d8c
11  3ffb56      To apply try to find a way to write the factor inside
   minimization as a linear function
12  ac2c47      of a value related to i. Everything else related to j will
   become constant.
13  c4c9bd  */
14  d41d8c
15  69abfb  #define MAXN 112345
16  d41d8c
17  a58cd5  int a[MAXN];
18  c4b25f  int b[MAXN];
19  d41d8c
20  f80900  ll tab[MAXN];
21  d41d8c
22  13a4b1  int main(void)
23  f95b70  {
24  1a88fd    int n;
25  f4c120    scanf("%d", &n);
26  83008c    for (int i = 0; i < n; i++)
27  9376f3      scanf("%d", &a[i]);
28  83008c    for (int i = 0; i < n; i++)
29  264aeb      scanf("%d", &b[i]);
30  d41d8c
31  a447b8    tab[0] = 0;
32  79ab5f    line_container l;
33  c01116    l.add(-b[0], -tab[0]);
34  d41d8c
35  aa4866    for (int i = 1; i < n; i++)
36  f95b70    {
37  23bd61      tab[i] = -l.query(a[i]);
38  8fd447      l.add(-b[i], -tab[i]);
39  cbb184    }
40  d41d8c
41  d41d8c    // Original DP O(n^2).
42  d41d8c    // for (int i = 1; i < n; i++)
```

```
43  d41d8c    // {
44  d41d8c    //  tab[i] = inf;
45  d41d8c    //  for (int j = 0; j < i; j++)
46  d41d8c    //    tab[i] = min(tab[i], tab[j] + a[i] * b[j]);
47  d41d8c    // }
48  d41d8c
49  cf6e15    cout << tab[n - 1] << endl;
50  cbb184  }
```

5.3 DP Optimization - Knuth

```
1 d41d8c  // https://www.spoj.com/problems/BRKSTRNG/
2 d41d8c
3 5d1131  #include "../../contest/header.hpp"
4 d41d8c
5 d41d8c  /*
6 c97188    Knuth Optimization for DP:
7 a41b1d      Optimizes dp of the form (or similar) dp[i][j] = min_{i <= k <=
    j}(dp[i][k-1] + dp[k+1][j] + c(i, j)).
8 f833b2      The classical case is building a optimal binary tree, where k
   determines the root.
9 d41d8c
10  c8aa2c      Let opt[i][j] be the value of k which minimizes the function.
    (in case of tie, choose the smallest)
11  c472ed      To apply this optimization, you need opt[i][j - 1] <= opt[i][
   j] <= opt[i+1][j].
12  0eeb73      That means the when you remove an element form the left (i +
   1), you won't choose a breaking point more to the left than before.
13  b38eb7      Also, when you remove an element from the right (j - 1), you
   won't choose a breking point more to the right than before.
14  242554      This is usually intuitive by the problem details.
15  d41d8c
16  cbb42a      Time goes from O(n^3) to O(n^2).
17  d41d8c
18  895144      To apply try to write the dp in the format above and verify
   if the property holds.
19  f76c09      Be careful with edge cases for opt.
20  d41d8c
21  3db72f    Author: Arthur Pratti Dadalto
22  c4c9bd  */
23  d41d8c
24  dbf7e4  #define MAXN 1123
25  d41d8c
26  c4b25f  int b[MAXN];
27  1ee552  ll tab[MAXN][MAXN];
28  38ab0d  int opt[MAXN][MAXN];
29  ef864b  int l, n;
30  d41d8c
31  5a7750  int c(int i, int j)
32  f95b70  {
```

```
33  33e24b    return b[j + 1] - b[i - 1];
34  cbb184  }
35  d41d8c
36  13a4b1  int main(void)
37  f95b70  {
38  57a598    while (scanf("%d %d", &l, &n) != EOF)
39  f95b70    {
40  5359f3      for (int i = 1; i <= n; i++)
41  264aeb        scanf("%d", &b[i]);
42  665bd2      b[n + 1] = l;
43  00d08b      b[0] = 0;
44  d41d8c
45  da41df      for (int i = 1; i <= n + 1; i++)
46  d6bc61        tab[i][i - 1] = 0, opt[i][i - 1] = i;
47  d41d8c
48  586d50      for (int i = n; i > 0; i--)
49  5d4199        for (int j = i; j <= n; j++)
50  f95b70        {
51  639af9          tab[i][j] = infll;
52  823124          for (int k = max(i, opt[i][j - 1]); k <= j && k <= opt[i
    + 1][j]; k++)
53  9e9168            if (tab[i][k - 1] + tab[k + 1][j] + c(i, j) < tab[i][j
    ])
54  f95b70            {
55  680c31              tab[i][j] = tab[i][k - 1] + tab[k + 1][j] + c(i, j);
56  14da03              opt[i][j] = k;
57  cbb184            }
58  cbb184        }
59  d41d8c
60  ea7bd9      printf("%lld\n", tab[1][n]);
61  cbb184    }
62  cbb184  }
```

5.4 Ternary Search (continuous)

```
1 d41d8c  /*
2 0a2f9f    Ternary Search:
3 28ea2d      Finds x such that f(x) is minimum in range [bot, top] in O(lg((
    top - bot) / eps)).
4 6f01ba      Value is correct within the specified precision eps.
5 d41d8c
6 ca2095    Constraints:
7 7474cd      f(x) is strictly decreasing for some interval [bot, x1],
    constant in an interval [x1, x2]
8 60dab3      and strictly increasing in a interval [x2, top]. x1 <= x2 are
    arbitrary values where [x1, x2]
9 7523fb      is a plateau of optimal solutions.
10  d41d8c
11  b95cae    Usage:
12  5b60e3      Call the function passing a lambda expression or function f.
```

```
13  8bc9f4      If there are multiple possible solutions, assume that an
   arbitrary one in the plateau is returned.
14  d41d8c
15  3db72f    Author: Arthur Pratti Dadalto
16  c4c9bd  */
17  d41d8c
18  398727  template <typename F>
19  ca64a1  double ternary_search(const F &f, double bot = -1e9, double top =
    1e9, double eps = 1e-9)
20  f95b70  {
21  14d91b    while (top - bot > eps)
22  f95b70    {
23  8e37d7      double x1 = (0.55*bot + 0.45*top); // (2*bot + top) / 3 is
   more stable, but slower.
24  3f8318      double x2 = (0.45*bot + 0.55*top);
25  9482ba      if (f(x1) > f(x2))
26  443914        bot = x1;
27  2954e9      else
28  16b1b8        top = x2;
29  cbb184    }
30  d41d8c
31  05eb81    return (bot + top) / 2;
32  cbb184  }
```

6 Number Theory

6.1 Euclid

```
#include "../../contest/header.hpp"

/*
  Extended Euclidean Algorithm:
    Returns the gcd of a and b.
    Also finds numbers x and y for which a * x + b * y = gcd(a, b) (not unique).
    All pairs can be represented in the form (x + k * b / gcd, y - k * a / gcd) for k an arbitrary integer.
    If there are several such x and y, the function returns the pair for which |x| + |y| is minimal.
    If there are several x and y satisfying the minimal criteria, it outputs the pair for which X <= Y.

  Source: modified from https://cp-algorithms.com/algebra/extended-euclid-algorithm.html

  Usage:
    For non-extendend version, c++ has __gcd and __lcm.

  Constraints:
    Produces correct results for negative integers as well.
*/

template<class T>
T gcd(T a, T b, T &x, T &y)
{
  if (b == 0)
  {
    x = 1;
    y = 0;
    return a;
  }

  T x1, y1;
  T d = gcd(b, a % b, x1, y1);
  x = y1;
  y = x1 - y1 * (a / b);
  return d;
}
```

6.2 Pollard rho

```
/*
  Description: Pollard-rho randomized factorization algorithm. Returns prime
```

```
3 9849b1    factors of a number, in arbitrary order (e.g. 2299 -> {11, 19,
   11}).
4 c21e7b    Time: O(n^1/4) gcd calls, less for numbers with small factors.
5 d41d8c
6 1d1558    Source: https://github.com/kth-competitive-programming/kactl/blob
   /master/content/number-theory/Factor.h
7 c4c9bd    */
8 f4cf5b  typedef unsigned long long ull;
9 088cf4  typedef long double ld;
10  d41d8c
11  ae330e  ull mod_mul(ull a, ull b, ull M) {
12  053258    ll ret = a * b - M * ull(ld(a) * ld(b) / ld(M));
13  964402    return ret + M * (ret < 0) - M * (ret >= (ll)M);
14  cbb184  }
15  97f234  ull mod_pow(ull b, ull e, ull mod) {
16  c1a4a1    ull ans = 1;
17  4d1884    for (; e; b = mod_mul(b, b, mod), e /= 2)
18  0c1ecc      if (e & 1) ans = mod_mul(ans, b, mod);
19  ba75d2    return ans;
20  cbb184  }
21  d41d8c
22  da49ed  bool isPrime(ull n) {
23  32f8ec    if (n < 2 || n % 6 % 4 != 1) return n - 2 < 2;
24  43a246    ull A[] = {2, 325, 9375, 28178, 450775, 9780504, 1795265022},
25  c17dd6        s = __builtin_ctzll(n-1), d = n >> s;
26  d236d6    for(auto &a : A) {   // ^ count trailing zeroes
27  8a86e5      ull p = mod_pow(a, d, n), i = s;
28  274cbc      while (p != 1 && p != n - 1 && a % n && i--)
29  2cbb80        p = mod_mul(p, p, n);
30  e2871b      if (p != n-1 && i != s) return 0;
31  cbb184    }
32  6a5530    return 1;
33  cbb184  }
34  d41d8c
35  7eb30f  ull pollard(ull n) {
36  4de5da    auto f = [n](ull x) { return (mod_mul(x, x, n) + 1) % n; };
37  68eade    if (!(n & 1)) return 2;
38  7b6fa7    for (ull i = 2;; i++) {
39  e17462      ull x = i, y = f(x), p;
40  332fe8      while ((p = __gcd(n + y - x, n)) == 1)
41  b789c2        x = f(x), y = f(f(y));
42  c493bb      if (p != n) return p;
43  cbb184    }
44  cbb184  }
45  d41d8c
46  bc43a4  vector<ull> factorize(ull n) {
47  1b90e8    if (n == 1) return {};
48  6b5b32    if (isPrime(n)) return {n};
49  bc6125    ull x = pollard(n);
50  b3b29a    auto l = factorize(x), r = factorize(n / x);
```

```
51  7af87c    l.insert(l.end(), all(r));
52  792fd4    return l;
53  cbb184  }
```

6.3 Modular Inverse

```
1 771bdd  #include "../euclid/euclid.cpp"
2 d41d8c
3 d41d8c  /*
4 18a91e    Modular Inverse:
5 76e032      Returns an integer x such that (a * x) % m == 1.
6 4e4745      The modular inverse exists if and only if a and m are
   relatively prime.
7 ff1c03      Modular inverse is also equal to a^(phi(m) - 1) % m.
8 261f2c      In particular, if m is prime a^(-1) == a^(m-2), which might be
   faster to code.
9 d41d8c
10  3997db    Source: modified from https://cp-algorithms.com/algebra/module-
   inverse.html
11  c4c9bd  */
12  d41d8c
13  4fce64  template<class T>
14  b267c1  T mod_inverse(T a, T m)
15  f95b70  {
16  645c5d    T x, y;
17  6553c1    assert(gcd(a, m, x, y) == 1); // Or return something, if gcd is
    not 1 the inverse doesn't exist.
18  08ffd4    return (x % m + m) % m;
19  cbb184  }
```

6.4 Phi

```
1 5d1131  #include "../../contest/header.hpp"
2 d41d8c
3 d41d8c  /*
4 bf26c1    Euler's totient function (PHI):
5 fc5093      Euler's totient function, also known as phi-function PHI(n),
   counts the number of integers
6 d7ef61      between 1 and n inclusive, which are coprime to n. Two numbers
   are coprime if their greatest
7 bf3431      common divisor equals 1 (1 is considered to be coprime to any
   number).
8 d41d8c
9 d41d8c
10  3997db    Source: modified from https://cp-algorithms.com/algebra/phi-
   function.html
11  e1fde7      and https://github.com/kth-competitive-programming/kactl/blob
   /master/content/number-theory/phiFunction.h
12  d41d8c
13  b95cae    Usage:
14  a1b248      Some useful properties:
```

```
9f5d84      - If p is a prime number, PHI(p)=p-1.
d4d311      - If a and b are relatively prime, PHI(ab)=PHI(a)*PHI(b).
65a02c      - In general, for not coprime a and b, PHI(ab)=PHI(a)*PHI(b)*d/PHI(d), with d=gcd(a,b) holds.
417c3d      - PHI(PHI(m)) <= m / 2
037fb5      - Euler's theorem: a^PHI(m) === 1 (mod m), for a and m coprime.
bffb1c      - For a and m coprime: a^n === a^(n % PHI(m)) (mod m)
986ce1      - For arbitrary x,m and n >= log_2(m): x^n === x^(PHI(m)+[n % PHI(m)]) (mod m)
8d568b      The one above allows computing modular exponentiation for really large exponents.
ec5d4e      - If d is a divisor of n, then there are phi(n/d) numbers i <= n for which gcd(i,n)=d
137411      - sum_{d|n} phi(d) = n
c228b3      - sum_{1 <= k <= n, gcd(k,n)=1} k = n * phi(n) / 2, for n > 1
c4c9bd  */
d41d8c
d41d8c  // Use this one for few values of phi.
b5f6f9  int phi(int n)
f95b70  {
efa47a    int result = n;
83f497    for (int i = 2; i * i <= n; i++)
f95b70    {
775f6d      if (n % i == 0)
f95b70      {
49edb8        while (n % i == 0)
1358bf          n /= i;
21cd49        result -= result / i;
cbb184      }
cbb184    }
f3d362    if (n > 1)
e48781      result -= result / n;
dc8384    return result;
cbb184  }
d41d8c
4fee4d  namespace totient
f95b70  {
2d637a  const int MAXV = 1000001; // Takes ~0.03 s for 10^6.
6e559b  int phi[MAXV];
d41d8c
b2a56e  void init()
f95b70  {
9484bb    for (int i = 0; i < MAXV; i++)
ed1f90      phi[i] = i & 1 ? i : i / 2;
9be7d6    for (int i = 3; i < MAXV; i += 2)
a2252f      if (phi[i] == i)
8a4437        for (int j = i; j < MAXV; j += i)
a9ba36          phi[j] -= phi[j] / i;
cbb184  }
```

```
60  cbb184  } // namespace totient
```

6.5 Sieve

```
1 5d1131  #include "../../contest/header.hpp"
2 d41d8c
3 d41d8c  /*
4 d5cfbe    Sieve of Eratosthenes:
5 0a5343      Finds all primes in interval [2, MAXP] in O(MAXP) time.
6 7e51df      Also finds lp[i] for every i in [2, MAXP], such that lp[i] is
   the minimum prime factor of i.
7 6dbf8e      Particularly useful for factorization.
8 d41d8c
9 3997db    Source: modified from https://cp-algorithms.com/algebra/prime-
   sieve-linear.html
10  d41d8c
11  b95cae    Usage:
12  9290fb      Set MAXP and call init.
13  85265b      Sieve for 10^7 should run in about 0.2 s.
14  c4c9bd  */
15  d41d8c
16  2ca8b0  namespace sieve
17  f95b70  {
18  bace98  const int MAXP = 10000000; // Will find primes in interval [2,
   MAXP].
19  39b809  int lp[MAXP + 1]; // lp[i] is the minimum prime factor of i.
20  632f82  vector<int> p; // Ordered list of primes up to MAXP.
21  d41d8c
22  b2a56e  void init()
23  f95b70  {
24  008cd3    for (int i = 2; i <= MAXP; i++)
25  f95b70    {
26  d4a1cc      if (lp[i] == 0)
27  b6fab7        p.push_back(lp[i] = i);
28  d41d8c
29  9d854a      for (int j = 0; j < (int)p.size() && p[j] <= lp[i] && i * p[j
   ] <= MAXP; j++)
30  fb7d48        lp[i * p[j]] = p[j];
31  cbb184    }
32  cbb184  }
33  cbb184  } // namespace sieve
```

7 Numerical

7.1 Big Int

```
1 5d1131  #include "../../contest/header.hpp"
2 d41d8c
3 d41d8c  // This code is not meant to be written in icpc contests. This is
   just here to fill a void for now.
4 d41d8c  // Source: someone on CF
5 d41d8c
6 d41d8c  // NOTE:
7 d41d8c  // This code contains various bug fixes compared to the original
   version from
8 d41d8c  // indy256 (github.com/indy256/codelibrary/blob/master/cpp/
   numbertheory/bigint-full.cpp),
9 d41d8c  // including:
10  d41d8c  // - Fix overflow bug in mul_karatsuba.
11  d41d8c  // - Fix overflow bug in fft.
12  d41d8c  // - Fix bug in initialization from long long.
13  d41d8c  // - Optimized operators + - *.
14  d41d8c  //
15  d41d8c  // Tested:
16  d41d8c  // - https://www.e-olymp.com/en/problems/266: Comparison
17  d41d8c  // - https://www.e-olymp.com/en/problems/267: Subtraction
18  d41d8c  // - https://www.e-olymp.com/en/problems/271: Multiplication
19  d41d8c  // - https://www.e-olymp.com/en/problems/272: Multiplication
20  d41d8c  // - https://www.e-olymp.com/en/problems/313: Addition
21  d41d8c  // - https://www.e-olymp.com/en/problems/314: Addition/
   Subtraction
22  d41d8c  // - https://www.e-olymp.com/en/problems/317: Multiplication (
   simple / karatsuba / fft)
23  d41d8c  // - https://www.e-olymp.com/en/problems/1327: Multiplication
24  d41d8c  // - https://www.e-olymp.com/en/problems/1328
25  d41d8c  // - VOJ BIGNUM: Addition, Subtraction, Multiplication.
26  d41d8c  // - SGU 111: sqrt
27  d41d8c  // - SGU 193
28  d41d8c  // - SPOJ MUL, VFMUL: Multiplication.
29  d41d8c  // - SPOJ FDIV, VFDIV: Division.
30  d41d8c
31  d73a77  const int BASE_DIGITS = 9;
32  82e97b  const int BASE = 1000000000;
33  d41d8c
34  6acb6c  struct BigInt {
35  d65d12      int sign;
36  a9d078      vector<int> a;
37  d41d8c
38  d41d8c      // -------------------- Constructors --------------------
39  d41d8c      // Default constructor.
40  1acfca      BigInt() : sign(1) {}
41  d41d8c
```

```
42  d41d8c      // Constructor from long long.
43  ccf902      BigInt(long long v) {
44  324222          *this = v;
45  cbb184      }
46  235125      BigInt& operator = (long long v) {
47  ce6fc2          sign = 1;
48  ea2149          if (v < 0) {
49  6a74a9              sign = -1;
50  6fab41              v = -v;
51  cbb184          }
52  22838a          a.clear();
53  fefe2d          for (; v > 0; v = v / BASE)
54  c237f1              a.push_back(v % BASE);
55  357a55          return *this;
56  cbb184      }
57  d41d8c
58  d41d8c      // Initialize from string.
59  c710ec      BigInt(const string& s) {
60  e65d4a          read(s);
61  cbb184      }
62  d41d8c
63  d41d8c      // -------------------- Input / Output --------------------
64  6c30c4      void read(const string& s) {
65  ce6fc2          sign = 1;
66  22838a          a.clear();
67  bec7a6          int pos = 0;
68  a68fdf          while (pos < (int) s.size() && (s[pos] == '-' || s[pos]
    == '+')) {
69  dbe226              if (s[pos] == '-')
70  2b8bd1                  sign = -sign;
71  17dad0              ++pos;
72  cbb184          }
73  7959ef          for (int i = s.size() - 1; i >= pos; i -= BASE_DIGITS) {
74  c67d6f              int x = 0;
75  d343c4              for (int j = max(pos, i - BASE_DIGITS + 1); j <= i; j
    ++)
76  cfc7e4                  x = x * 10 + s[j] - '0';
77  7c6978              a.push_back(x);
78  cbb184          }
79  0ebb65          trim();
80  cbb184      }
81  bd2995      friend istream& operator>>(istream &stream, BigInt &v) {
82  ac0066          string s;
83  e0c759          stream >> s;
84  c4002a          v.read(s);
85  a87cf7          return stream;
86  cbb184      }
87  d41d8c
88  44647f      friend ostream& operator<<(ostream &stream, const BigInt &v)
    {
```

```
89  b5c525          if (v.sign == -1 && !v.isZero())
90  27bc2a              stream << '-';
91  4fda68          stream << (v.a.empty() ? 0 : v.a.back());
92  fce618          for (int i = (int) v.a.size() - 2; i >= 0; --i)
93  018b85              stream << setw(BASE_DIGITS) << setfill('0') << v.a[i
    ];
94  a87cf7          return stream;
95  cbb184      }
96  d41d8c
97  d41d8c      // -------------------- Comparison --------------------
98  7014c0      bool operator<(const BigInt &v) const {
99  eb909f          if (sign != v.sign)
100 603965             return sign < v.sign;
101 a2765e         if (a.size() != v.a.size())
102 f7d303             return a.size() * sign < v.a.size() * v.sign;
103 305fef         for (int i = ((int) a.size()) - 1; i >= 0; i--)
104 00d0de             if (a[i] != v.a[i])
105 2441c5                 return a[i] * sign < v.a[i] * sign;
106 d1fe4d         return false;
107 cbb184     }
108 d41d8c
109 426053     bool operator>(const BigInt &v) const {
110 54bd3a         return v < *this;
111 cbb184     }
112 65677c     bool operator<=(const BigInt &v) const {
113 0fe7a0         return !(v < *this);
114 cbb184     }
115 605209     bool operator>=(const BigInt &v) const {
116 d9c542         return !(*this < v);
117 cbb184     }
118 880606     bool operator==(const BigInt &v) const {
119 7f44a6         return !(*this < v) && !(v < *this);
120 cbb184     }
121 062171     bool operator!=(const BigInt &v) const {
122 6c55aa         return *this < v || v < *this;
123 cbb184     }
124 d41d8c
125 d41d8c     // Returns:
126 d41d8c     // 0 if |x| == |y|
127 d41d8c     // -1 if |x| < |y|
128 d41d8c     // 1 if |x| > |y|
129 ce6386     friend int __compare_abs(const BigInt& x, const BigInt& y) {
130 e78df5         if (x.a.size() != y.a.size()) {
131 c86c62             return x.a.size() < y.a.size() ? -1 : 1;
132 cbb184         }
133 d41d8c
134 a552ab         for (int i = ((int) x.a.size()) - 1; i >= 0; --i) {
135 a5b2df             if (x.a[i] != y.a[i]) {
136 b1ec3d                 return x.a[i] < y.a[i] ? -1 : 1;
137 cbb184             }
```

```
138 cbb184          }
139 bb30ba          return 0;
140 cbb184      }
141 d41d8c
142 d41d8c      // -------------------- Unary operator - and operators +=
    --------------------
143 1e3c00      BigInt operator-() const {
144 18bf1f          BigInt res = *this;
145 b9607c          if (isZero()) return res;
146 d41d8c
147 290faa          res.sign = -sign;
148 b5053e          return res;
149 cbb184      }
150 d41d8c
151 d41d8c      // Note: sign ignored.
152 d60e6f      void __internal_add(const BigInt& v) {
153 f7247c          if (a.size() < v.a.size()) {
154 2ce41c              a.resize(v.a.size(), 0);
155 cbb184          }
156 1addcf          for (int i = 0, carry = 0; i < (int) max(a.size(), v.a.
    size()) || carry; ++i) {
157 df4512              if (i == (int) a.size()) a.push_back(0);
158 d41d8c
159 85e77e              a[i] += carry + (i < (int) v.a.size() ? v.a[i] : 0);
160 49bff0              carry = a[i] >= BASE;
161 1791a8              if (carry) a[i] -= BASE;
162 cbb184          }
163 cbb184      }
164 d41d8c
165 d41d8c      // Note: sign ignored.
166 8b47dc      void __internal_sub(const BigInt& v) {
167 65cb2e          for (int i = 0, carry = 0; i < (int) v.a.size() || carry;
    ++i) {
168 a1437d              a[i] -= carry + (i < (int) v.a.size() ? v.a[i] : 0);
169 e0b1f1              carry = a[i] < 0;
170 da53a6              if (carry) a[i] += BASE;
171 cbb184          }
172 0e329b          this->trim();
173 cbb184      }
174 d41d8c
175 89fb6b      BigInt operator += (const BigInt& v) {
176 8ea459          if (sign == v.sign) {
177 570069              __internal_add(v);
178 9d9745          } else {
179 ae3659              if (__compare_abs(*this, v) >= 0) {
180 e9815a                  __internal_sub(v);
181 9d9745              } else {
182 dcc3fe                  BigInt vv = v;
183 3c5f43                  swap(*this, vv);
184 fe0d8d                  __internal_sub(vv);
```

```
185 cbb184              }
186 cbb184          }
187 357a55          return *this;
188 cbb184      }
189 d41d8c
190 6b1a22      BigInt operator -= (const BigInt& v) {
191 8ea459          if (sign == v.sign) {
192 ae3659              if (__compare_abs(*this, v) >= 0) {
193 e9815a                  __internal_sub(v);
194 9d9745              } else {
195 dcc3fe                  BigInt vv = v;
196 3c5f43                  swap(*this, vv);
197 fe0d8d                  __internal_sub(vv);
198 0db96d                  this->sign = -this->sign;
199 cbb184              }
200 9d9745          } else {
201 570069              __internal_add(v);
202 cbb184          }
203 357a55          return *this;
204 cbb184      }
205 d41d8c
206 d41d8c      // Optimize operators + and - according to
207 d41d8c      // https://stackoverflow.com/questions/13166079/move-
    semantics-and-pass-by-rvalue-reference-in-overloaded-arithmetic
208 f1e02d      template< typename L, typename R >
209 81c687          typename std::enable_if<
210 4eceb0              std::is_convertible<L, BigInt>::value &&
211 c0db24              std::is_convertible<R, BigInt>::value &&
212 061102              std::is_lvalue_reference<R&&>::value,
213 6b2030              BigInt>::type friend operator + (L&& l, R&& r) {
214 46b960          BigInt result(std::forward<L>(l));
215 fbef75          result += r;
216 dc8384          return result;
217 cbb184      }
218 f1e02d      template< typename L, typename R >
219 81c687          typename std::enable_if<
220 4eceb0              std::is_convertible<L, BigInt>::value &&
221 c0db24              std::is_convertible<R, BigInt>::value &&
222 bccc2f              std::is_rvalue_reference<R&&>::value,
223 6b2030              BigInt>::type friend operator + (L&& l, R&& r) {
224 5f09ae          BigInt result(std::move(r));
225 a5a040          result += l;
226 dc8384          return result;
227 cbb184      }
228 d41d8c
229 f1e02d      template< typename L, typename R >
230 81c687          typename std::enable_if<
231 4eceb0              std::is_convertible<L, BigInt>::value &&
232 6ca6cc              std::is_convertible<R, BigInt>::value,
233 1612ea              BigInt>::type friend operator - (L&& l, R&& r) {
```

```
234 46b960          BigInt result(std::forward<L>(l));
235 1d15a0          result -= r;
236 dc8384          return result;
237 cbb184      }
238 d41d8c
239 d41d8c      // -------------------- Operators * / % --------------------
240 a179f4      friend pair<BigInt, BigInt> divmod(const BigInt& a1, const
   BigInt& b1) {
241 872d46          assert(b1 > 0);  // divmod not well-defined for b < 0.
242 d41d8c
243 25f4e9          long long norm = BASE / (b1.a.back() + 1);
244 7c41dc          BigInt a = a1.abs() * norm;
245 ecd4f4          BigInt b = b1.abs() * norm;
246 da5ddc          BigInt q = 0, r = 0;
247 90ee93          q.a.resize(a.a.size());
248 d41d8c
249 72b5b8          for (int i = a.a.size() - 1; i >= 0; i--) {
250 79aca3              r *= BASE;
251 0caac0              r += a.a[i];
252 0eeb4e              long long s1 = r.a.size() <= b.a.size() ? 0 : r.a[b.a
   .size()];
253 bc1a99              long long s2 = r.a.size() <= b.a.size() - 1 ? 0 : r.a
   [b.a.size() - 1];
254 0ebba0              long long d = ((long long) BASE * s1 + s2) / b.a.back
   ();
255 5d4f85              r -= b * d;
256 612239              while (r < 0) {
257 bd3902                  r += b, --d;
258 cbb184              }
259 5898c8              q.a[i] = d;
260 cbb184          }
261 d41d8c
262 535024          q.sign = a1.sign * b1.sign;
263 a29af3          r.sign = a1.sign;
264 36a918          q.trim();
265 9a35fd          r.trim();
266 38a539          auto res = make_pair(q, r / norm);
267 458098          if (res.second < 0) res.second += b1;
268 b5053e          return res;
269 cbb184      }
270 547e4b      BigInt operator/(const BigInt &v) const {
271 ce8f7c          return divmod(*this, v).first;
272 cbb184      }
273 d41d8c
274 ee46c3      BigInt operator%(const BigInt &v) const {
275 7a671a          return divmod(*this, v).second;
276 cbb184      }
277 d41d8c
278 c2998e      void operator/=(int v) {
279 d1ee66          assert(v > 0);  // operator / not well-defined for v <=
```

```
   0.
280 dd9f94          if (llabs(v) >= BASE) {
281 85cc00              *this /= BigInt(v);
282 505b97              return ;
283 cbb184          }
284 8e679f          if (v < 0)
285 20198f              sign = -sign, v = -v;
286 8e5533          for (int i = (int) a.size() - 1, rem = 0; i >= 0; --i) {
287 cbe153              long long cur = a[i] + rem * (long long) BASE;
288 8d1e71              a[i] = (int) (cur / v);
289 cb35e0              rem = (int) (cur % v);
290 cbb184          }
291 0ebb65          trim();
292 cbb184      }
293 d41d8c
294 49658a      BigInt operator/(int v) const {
295 d1ee66          assert(v > 0);  // operator / not well-defined for v <=
   0.
296 d41d8c
297 dd9f94          if (llabs(v) >= BASE) {
298 ed0225              return *this / BigInt(v);
299 cbb184          }
300 18bf1f          BigInt res = *this;
301 37184f          res /= v;
302 b5053e          return res;
303 cbb184      }
304 3b4fa6      void operator/=(const BigInt &v) {
305 e51f70          *this = *this / v;
306 cbb184      }
307 d41d8c
308 54c35d      long long operator%(long long v) const {
309 d1ee66          assert(v > 0);  // operator / not well-defined for v <=
   0.
310 a1e888          assert(v < BASE);
311 cbed95          int m = 0;
312 947442          for (int i = a.size() - 1; i >= 0; --i)
313 95269a              m = (a[i] + m * (long long) BASE) % v;
314 9af577          return m * sign;
315 cbb184      }
316 d41d8c
317 a0b62a      void operator*=(int v) {
318 dd9f94          if (llabs(v) >= BASE) {
319 014cdd              *this *= BigInt(v);
320 505b97              return ;
321 cbb184          }
322 8e679f          if (v < 0)
323 20198f              sign = -sign, v = -v;
324 c6279c          for (int i = 0, carry = 0; i < (int) a.size() || carry;
   ++i) {
325 74ab7d              if (i == (int) a.size())
```

```
326 ddfb75                  a.push_back(0);
327 d09f08              long long cur = a[i] * (long long) v + carry;
328 98cd39              carry = (int) (cur / BASE);
329 861843              a[i] = (int) (cur % BASE);
330 d41d8c              //asm("divl %%ecx" : "=a"(carry), "=d"(a[i]) : "A"(
    cur), "c"(base));
331 d41d8c              /*
332 97f03f               int val;
333 ab8362               __asm {
334 bab6b5               lea esi, cur
335 6cd1f3               mov eax, [esi]
336 d5ad3f               mov edx, [esi+4]
337 378c50               mov ecx, base
338 d88250               div ecx
339 e3e615               mov carry, eax
340 6f8726               mov val, edx;
341 cbb184               }
342 26a9ce               a[i] = val;
343 c4c9bd               */
344 cbb184          }
345 0ebb65          trim();
346 cbb184      }
347 d41d8c
348 d1d185      BigInt operator*(int v) const {
349 dd9f94          if (llabs(v) >= BASE) {
350 42696e              return *this * BigInt(v);
351 cbb184          }
352 18bf1f          BigInt res = *this;
353 6b38f1          res *= v;
354 b5053e          return res;
355 cbb184      }
356 d41d8c
357 d41d8c      // Convert BASE 10^old --> 10^new.
358 ead252      static vector<int> convert_base(const vector<int> &a, int
    old_digits, int new_digits) {
359 943071          vector<long long> p(max(old_digits, new_digits) + 1);
360 c4bbd4          p[0] = 1;
361 85cf8d          for (int i = 1; i < (int) p.size(); i++)
362 7cc6c9              p[i] = p[i - 1] * 10;
363 02fb60          vector<int> res;
364 c6278d          long long cur = 0;
365 6427c9          int cur_digits = 0;
366 c0e004          for (int i = 0; i < (int) a.size(); i++) {
367 b28c31              cur += a[i] * p[cur_digits];
368 e4696c              cur_digits += old_digits;
369 5ebda5              while (cur_digits >= new_digits) {
370 6f203f                  res.push_back((long long)(cur % p[new_digits]));
371 1cec8a                  cur /= p[new_digits];
372 318982                  cur_digits -= new_digits;
373 cbb184              }
```

```
374 cbb184          }
375 a5eaaa          res.push_back((int) cur);
376 c5a021          while (!res.empty() && !res.back())
377 efcb65              res.pop_back();
378 b5053e          return res;
379 cbb184      }
380 d41d8c
381 009dfc      void fft(vector<complex<double> > & a, bool invert) const {
382 8ec808          int n = (int) a.size();
383 d41d8c
384 677a94          for (int i = 1, j = 0; i < n; ++i) {
385 4af5d7              int bit = n >> 1;
386 425aec              for (; j >= bit; bit >>= 1)
387 b39a0f                  j -= bit;
388 297413              j += bit;
389 9dcc5c              if (i < j)
390 33275d                  swap(a[i], a[j]);
391 cbb184          }
392 d41d8c
393 eb733a          for (int len = 2; len <= n; len <<= 1) {
394 2f82ea              double ang = 2 * 3.14159265358979323846 / len * (
   invert ? -1 : 1);
395 a0b444              complex<double> wlen(cos(ang), sin(ang));
396 6c8781              for (int i = 0; i < n; i += len) {
397 c2eaad                  complex<double> w(1);
398 876230                  for (int j = 0; j < len / 2; ++j) {
399 371eda                      complex<double> u = a[i + j];
400 0c0391                      complex<double> v = a[i + j + len / 2] * w;
401 6c3014                      a[i + j] = u + v;
402 273255                      a[i + j + len / 2] = u - v;
403 3e4104                      w *= wlen;
404 cbb184                  }
405 cbb184              }
406 cbb184          }
407 2111a0          if (invert)
408 6cb8cc              for (int i = 0; i < n; ++i)
409 b098a6                  a[i] /= n;
410 cbb184      }
411 d41d8c
412 0d5969      void multiply_fft(const vector<int> &a, const vector<int> &b,
    vector<int> &res) const {
413 58dd64          vector<complex<double> > fa(a.begin(), a.end());
414 249aaa          vector<complex<double> > fb(b.begin(), b.end());
415 43ec81          int n = 1;
416 727e5e          while (n < (int) max(a.size(), b.size()))
417 c149a4              n <<= 1;
418 c149a4          n <<= 1;
419 37aa6c          fa.resize(n);
420 870070          fb.resize(n);
421 d41d8c
```

```
422 3a13f2          fft(fa, false);
423 c76760          fft(fb, false);
424 6cb8cc          for (int i = 0; i < n; ++i)
425 940eb7              fa[i] *= fb[i];
426 959d01          fft(fa, true);
427 d41d8c
428 f38aa2          res.resize(n);
429 6e20af          long long carry = 0;
430 baeb9e          for (int i = 0; i < n; ++i) {
431 6e6901              long long t = (long long) (fa[i].real() + 0.5) +
   carry;
432 9e18f0              carry = t / 1000;
433 bb5b3b              res[i] = t % 1000;
434 cbb184          }
435 cbb184      }
436 d41d8c
437 d64466      BigInt mul_simple(const BigInt &v) const {
438 02a624          BigInt res;
439 325cfe          res.sign = sign * v.sign;
440 4bc9af          res.a.resize(a.size() + v.a.size());
441 7a7093          for (int i = 0; i < (int) a.size(); ++i)
442 b40a68              if (a[i])
443 761845                  for (int j = 0, carry = 0; j < (int) v.a.size()
   || carry; ++j) {
444 df3e98                      long long cur = res.a[i + j] + (long long) a[
   i] * (j < (int) v.a.size() ? v.a[j] : 0) + carry;
445 98cd39                      carry = (int) (cur / BASE);
446 ff01d5                      res.a[i + j] = (int) (cur % BASE);
447 cbb184                  }
448 d7ee6d          res.trim();
449 b5053e          return res;
450 cbb184      }
451 d41d8c
452 ad1556      typedef vector<long long> vll;
453 d41d8c
454 4d42f9      static vll karatsubaMultiply(const vll &a, const vll &b) {
455 94d5f8          int n = a.size();
456 1fb0e0          vll res(n + n);
457 44d3ec          if (n <= 32) {
458 83008c              for (int i = 0; i < n; i++)
459 f90a6b                  for (int j = 0; j < n; j++)
460 8dd9af                      res[i + j] += a[i] * b[j];
461 b5053e              return res;
462 cbb184          }
463 d41d8c
464 af0b16          int k = n >> 1;
465 f9fca2          vll a1(a.begin(), a.begin() + k);
466 72c0c7          vll a2(a.begin() + k, a.end());
467 48ebf6          vll b1(b.begin(), b.begin() + k);
468 88c9a6          vll b2(b.begin() + k, b.end());
```

```
469 d41d8c
470 03c868          vll a1b1 = karatsubaMultiply(a1, b1);
471 e56678          vll a2b2 = karatsubaMultiply(a2, b2);
472 d41d8c
473 40d6ad          for (int i = 0; i < k; i++)
474 c20ed7              a2[i] += a1[i];
475 40d6ad          for (int i = 0; i < k; i++)
476 b009cc              b2[i] += b1[i];
477 d41d8c
478 6a2f29          vll r = karatsubaMultiply(a2, b2);
479 be9bd2          for (int i = 0; i < (int) a1b1.size(); i++)
480 47fef2              r[i] -= a1b1[i];
481 cf04ec          for (int i = 0; i < (int) a2b2.size(); i++)
482 00a00c              r[i] -= a2b2[i];
483 d41d8c
484 5951a9          for (int i = 0; i < (int) r.size(); i++)
485 1bf61e              res[i + k] += r[i];
486 be9bd2          for (int i = 0; i < (int) a1b1.size(); i++)
487 d6cf88              res[i] += a1b1[i];
488 cf04ec          for (int i = 0; i < (int) a2b2.size(); i++)
489 ab9916              res[i + n] += a2b2[i];
490 b5053e          return res;
491 cbb184      }
492 d41d8c
493 287510      BigInt mul_karatsuba(const BigInt &v) const {
494 48c647          vector<int> a6 = convert_base(this->a, BASE_DIGITS, 6);
495 f64a05          vector<int> b6 = convert_base(v.a, BASE_DIGITS, 6);
496 e1cb30          vll a(a6.begin(), a6.end());
497 5ed74f          vll b(b6.begin(), b6.end());
498 1a813e          while (a.size() < b.size())
499 ddfb75              a.push_back(0);
500 0d118e          while (b.size() < a.size())
501 c40831              b.push_back(0);
502 634b60          while (a.size() & (a.size() - 1))
503 eed3fb              a.push_back(0), b.push_back(0);
504 16bf35          vll c = karatsubaMultiply(a, b);
505 02a624          BigInt res;
506 325cfe          res.sign = sign * v.sign;
507 6e20af          long long carry = 0;
508 7dbc9f          for (int i = 0; i < (int) c.size(); i++) {
509 dc97b8              long long cur = c[i] + carry;
510 cdf472              res.a.push_back((int) (cur % 1000000));
511 735fb2              carry = cur / 1000000;
512 cbb184          }
513 7b10c4          res.a = convert_base(res.a, 6, BASE_DIGITS);
514 d7ee6d          res.trim();
515 b5053e          return res;
516 cbb184      }
517 d41d8c
518 933d02      void operator*=(const BigInt &v) {
```

```
519 fa4bc1          *this = *this * v;
520 cbb184      }
521 24478f      BigInt operator*(const BigInt &v) const {
522 de6792          if (a.size() * v.a.size() <= 1000111) return mul_simple(v
    );
523 fec548          if (a.size() > 500111 || v.a.size() > 500111) return
    mul_fft(v);
524 a67c32          return mul_karatsuba(v);
525 cbb184      }
526 d41d8c
527 0f0ce5      BigInt mul_fft(const BigInt& v) const {
528 02a624          BigInt res;
529 325cfe          res.sign = sign * v.sign;
530 d1a018          multiply_fft(convert_base(a, BASE_DIGITS, 3),
    convert_base(v.a, BASE_DIGITS, 3), res.a);
531 74be5c          res.a = convert_base(res.a, 3, BASE_DIGITS);
532 d7ee6d          res.trim();
533 b5053e          return res;
534 cbb184      }
535 d41d8c
536 d41d8c      // -------------------- Misc --------------------
537 9f0aff      BigInt abs() const {
538 18bf1f          BigInt res = *this;
539 3ccc69          res.sign *= res.sign;
540 b5053e          return res;
541 cbb184      }
542 a0fac1      void trim() {
543 b03a9b          while (!a.empty() && !a.back())
544 4685a5              a.pop_back();
545 e28510          if (a.empty())
546 ce6fc2              sign = 1;
547 cbb184      }
548 d41d8c
549 88d324      bool isZero() const {
550 5c0518          return a.empty() || (a.size() == 1 && !a[0]);
551 cbb184      }
552 d41d8c
553 e7ccd6      friend BigInt gcd(const BigInt &a, const BigInt &b) {
554 183a15          return b.isZero() ? a : gcd(b, a % b);
555 cbb184      }
556 7977e6      friend BigInt lcm(const BigInt &a, const BigInt &b) {
557 8b81ac          return a / gcd(a, b) * b;
558 cbb184      }
559 d41d8c
560 2f7166      friend BigInt sqrt(const BigInt &a1) {
561 b25149          BigInt a = a1;
562 53b77e          while (a.a.empty() || a.a.size() % 2 == 1)
563 8a6b34              a.a.push_back(0);
564 d41d8c
565 0c5896          int n = a.a.size();
```

```
566 d41d8c
567 f9194d          int firstDigit = (int) sqrt((double) a.a[n - 1] * BASE +
    a.a[n - 2]);
568 3c7b49          int norm = BASE / (firstDigit + 1);
569 b65c20          a *= norm;
570 b65c20          a *= norm;
571 53b77e          while (a.a.empty() || a.a.size() % 2 == 1)
572 8a6b34              a.a.push_back(0);
573 d41d8c
574 8a28a4          BigInt r = (long long) a.a[n - 1] * BASE + a.a[n - 2];
575 4e5685          firstDigit = (int) sqrt((double) a.a[n - 1] * BASE + a.a[
    n - 2]);
576 97c0e8          int q = firstDigit;
577 02a624          BigInt res;
578 d41d8c
579 a1054f          for(int j = n / 2 - 1; j >= 0; j--) {
580 e63f29              for(; ; --q) {
581 592185                  BigInt r1 = (r - (res * 2 * BigInt(BASE) + q) * q
    ) * BigInt(BASE) * BigInt(BASE) + (j > 0 ? (long long) a.a[2 * j - 1] *
    BASE + a.a[2 * j - 2] : 0);
582 60f563                  if (r1 >= 0) {
583 01144f                      r = r1;
584 c2bef1                      break;
585 cbb184                  }
586 cbb184              }
587 d2c0d8              res *= BASE;
588 f2637e              res += q;
589 d41d8c
590 e79d0e              if (j > 0) {
591 febb34                  int d1 = res.a.size() + 2 < r.a.size() ? r.a[res.
    a.size() + 2] : 0;
592 baacce                  int d2 = res.a.size() + 1 < r.a.size() ? r.a[res.
    a.size() + 1] : 0;
593 78b193                  int d3 = res.a.size() < r.a.size() ? r.a[res.a.
    size()] : 0;
594 7d925d                  q = ((long long) d1 * BASE * BASE + (long long)
    d2 * BASE + d3) / (firstDigit * 2);
595 cbb184              }
596 cbb184          }
597 d41d8c
598 d7ee6d          res.trim();
599 28ae5c          return res / norm;
600 cbb184      }
601 2145c1  };
```

7.2 FFT

```
1 5d1131  #include "../../contest/header.hpp"
2 d41d8c
3 d41d8c  /*
```

```
4 18a91e    Modular Inverse:
5 f7b296      FFT allows multiplication of two polynomials in O(n log n).
6 420c7a      This can also be used to multiply two long numbers faster.
7 c00ff6      Other applications:
8 c35b73      - All possible sums of two arrays.
9 1da5a4      - Dot product of vector a with every cyclic shift of vector b.
10  49afe3     - Attaching two boolean stripes without two 1s next to each
   other.
11  52f6a3     - String matching.
12  d41d8c
13  b95cae    Usage:
14  afb0f5      long double is a lot slower. 3s with ld and 0.7 with double
   for 10^6 size vectors.
15  d41d8c
16  1d1558    Source: https://cp-algorithms.com/algebra/fft.html
17  c4c9bd  */
18  d41d8c
19  f4f8e6  using cd = complex<long double>;
20  c4f8de  const ld PI = acos(-1.0L);
21  d41d8c
22  9b5b94  void fft(vector<cd> &a, bool invert)
23  f95b70  {
24  94d5f8    int n = a.size();
25  d41d8c
26  d94885    for (int i = 1, j = 0; i < n; i++)
27  f95b70    {
28  4af5d7      int bit = n >> 1;
29  474fac      for (; j & bit; bit >>= 1)
30  53c7ca        j ^= bit;
31  53c7ca      j ^= bit;
32  d41d8c
33  9dcc5c      if (i < j)
34  33275d        swap(a[i], a[j]);
35  cbb184    }
36  d41d8c
37  2fe9ad    for (int len = 2; len <= n; len <<= 1)
38  f95b70    {
39  c19c97      ld ang = 2 * PI / len * (invert ? -1 : 1);
40  808a0b      cd wlen(cos(ang), sin(ang));
41  3dd9d3      for (int i = 0; i < n; i += len)
42  f95b70      {
43  8c3c80        cd w(1);
44  5594fb        for (int j = 0; j < len / 2; j++)
45  f95b70        {
46  cf0824          cd u = a[i + j], v = a[i + j + len / 2] * w;
47  6c3014          a[i + j] = u + v;
48  273255          a[i + j + len / 2] = u - v;
49  3e4104          w *= wlen;
50  cbb184        }
51  cbb184      }
```

```
52  cbb184    }
53  d41d8c
54  2111a0    if (invert)
55  f95b70    {
56  0b5665      for (cd &x : a)
57  b6d31b        x /= n;
58  cbb184    }
59  cbb184  }
60  d41d8c
61  d41d8c  // Input a[0] + a[1]x + a[2]x^2 ...
62  d41d8c  // Returns polynomial of size equal to the smallest power of two
    at least
63  d41d8c  // as large as a.size() + b.size(). This can have some extra
    zeros.
64  d41d8c  // Use long double if using long long.
65  4fce64  template <class T>
66  a3a2ed  vector<T> multiply(vector<T> const &a, vector<T> const &b)
67  f95b70  {
68  6fa6b9    vector<cd> fa(a.begin(), a.end()), fb(b.begin(), b.end());
69  43ec81    int n = 1;
70  cd5a64    while (n < a.size() + b.size())
71  c149a4      n <<= 1;
72  37aa6c    fa.resize(n);
73  870070    fb.resize(n);
74  d41d8c
75  3a13f2    fft(fa, false);
76  c76760    fft(fb, false);
77  83008c    for (int i = 0; i < n; i++)
78  940eb7      fa[i] *= fb[i];
79  959d01    fft(fa, true);
80  d41d8c
81  ebf3b6    vector<T> result(n);
82  83008c    for (int i = 0; i < n; i++)
83  4c9bb2      result[i] = round(fa[i].real()); // Remember to remove
    rounding if working with floats.
84  dc8384    return result;
85  cbb184  }
```

7.3 Fraction

```
1 5d1131  #include "../../contest/header.hpp"
2 d41d8c
3 d41d8c  /*
4 390211    Fraction representation:
5 4d1181      All operations run in O(gcd) = O(log).
6 d41d8c
7 b95cae    Usage:
8 70e7d7      Don't modify internal values, use constructor.
9 408ef0      Some nice things about the constructor: frac() = 0/1, frac(5) =
     5/1.
```

```
10  d41d8c
11  b8d28c      Be careful that the numerator and denominator might overflow
    if lcm is too big.
12  20fa30      In those cases, you can always do frac<big_int>, but that
    will be painful to code.
13  d41d8c
14  3db72f    Author: Arthur Pratti Dadalto
15  c4c9bd  */
16  d41d8c
17  4fce64  template <class T>
18  4cf1ca  struct frac
19  f95b70  {
20  e75828    T a, b; // b can't be negative, very important.
21  d41d8c
22  191fc6    explicit frac(T a = 0, T b = 1) : a(a), b(b) { simpl(); }
23  d41d8c
24  7d70f7    void simpl()
25  f95b70    {
26  8eb5bb      T g = __gcd(abs(a), abs(b)) * sign(b); // Make b positive.
27  fe7245      a /= g;
28  ee2d42      b /= g;
29  cbb184    }
30  d41d8c
31  d59b8a    bool operator<(const frac &rhs) const
32  f95b70    {
33  5c6427      return a * rhs.b < rhs.a * b;
34  cbb184    }
35  d41d8c
36  7ebf19    bool operator>(const frac &rhs) const
37  f95b70    {
38  2ab79c      return rhs < *this;
39  cbb184    }
40  d41d8c
41  d60bf3    bool operator==(const frac &rhs) const // TODO: untested.
42  f95b70    {
43  77c0b8      return !(*this < rhs) && !(rhs < *this);
44  cbb184    }
45  d41d8c
46  473b74    frac operator*(const frac &rhs) const
47  f95b70    {
48  f0117d      return frac(a * rhs.a, b * rhs.b);
49  cbb184    }
50  d41d8c
51  04b5a1    frac operator+(const frac &rhs) const
52  f95b70    {
53  3ff11f      T m = (b * rhs.b) / __gcd(b, rhs.b);
54  24edd6      return frac(a * (m / b) + rhs.a * (m / rhs.b), m);
55  cbb184    }
56  d41d8c
57  c8ca1d    frac operator-(void) const
```

```
58  f95b70    {
59  132fb3      return frac(-a, b);
60  cbb184    }
61  d41d8c
62  de243f    frac operator-(const frac &rhs) const
63  f95b70    {
64  111760      return (*this) + (-rhs);
65  cbb184    }
66  d41d8c
67  d63a85    frac operator/(const frac &rhs) const
68  f95b70    {
69  f5299b      return (*this) * frac(rhs.b, rhs.a);
70  cbb184    }
71  d41d8c
72  9e018a    friend ostream &operator<<(ostream &os, const frac &f)
73  f95b70    {
74  891d94      return os << f.a << "/" << f.b;
75  cbb184    }
76  2145c1  };
77  d41d8c
```

7.4 Integration

```
1 d41d8c  /*
2 f64ead    Numerical Integration:
3 49d5e8      Given a function f and an interval [a, b] estimates integral of
    f(x) dx from a to b.
4 bfe460      Error is in theory inversely proportional to n^4.
5 d41d8c
6 b95cae    Usage:
7 be1ead      n, the number of intervals must be even.
8 d41d8c
9 3db72f    Author: Arthur Pratti Dadalto
10  c4c9bd  */
11  d41d8c
12  044d82  template <class F>
13  7d9945  double simpsons(const F &f, int n /* even */, double a, double b)
14  f95b70  {
15  46af34    double retv = f(a) + f(b);
16  d025af    double h = (b - a) / n;
17  acfc81    for (int i = 1; i < n; i += 2)
18  900086      retv += 4 * f(a + i * h);
19  1c3900    for (int i = 2; i < n; i += 2)
20  6c1313      retv += 2 * f(a + i * h);
21  d41d8c
22  055fe5    retv *= h / 3;
23  6272cf    return retv;
24  cbb184  }
25  d41d8c
26  d41d8c  // Sample usage:
```

```
27  d41d8c  // int main(void)
28  d41d8c  // {
29  d41d8c  //  printf("%.20lf\n", simpsons([](double x) { return pow(sin(
   M_PI * x / 2.0), 3.2);}, 2000, 0, 2));
30  d41d8c  // }
```

7.5 linalg

```
1 5d1131  #include "../../contest/header.hpp"
2 d41d8c
3 d41d8c  /*
4 f92339    Vector and matrix operations:
5 687bbc      Details are given in each function.
6 3ab55f      vec inherits from vector<T>, so there is a lot you can do with
   it.
7 5ae524      Also, mat inherits from vector<vec<T>>.
8 d41d8c
9 3db72f    Author: Arthur Pratti Dadalto
10  d41d8c
11  1ef4c8    Source: some of it from https://github.com/kth-competitive-
   programming/kactl/blob/master/content/numerical/MatrixInverse.h
12  c4c9bd  */
13  d41d8c
14  4fce64  template <class T>
15  fe4002  struct vec : vector<T>
16  f95b70  {
17  469362    vec(int n) : vector<T>(n) {}
18  d41d8c
19  d41d8c    // c = a*x + b*y
20  e918cb    static void linear_comb(const vec &a, T x, const vec &b, T y,
   vec &c)
21  f95b70    {
22  8fe753      for (int i = 0; i < sz(a); i++)
23  75e753        c[i] = a[i] * x + b[i] * y;
24  cbb184    }
25  d41d8c
26  d41d8c    // return a*x + b*y
27  250f88    static vec linear_comb(vec a, T x, const vec &b, T y)
28  f95b70    {
29  4fec85      linear_comb(a, x, b, y, a);
30  3f5343      return a;
31  cbb184    }
32  2145c1  };
33  d41d8c
34  4fce64  template <class T>
35  dade1f  struct mat : vector<vec<T>>
36  f95b70  {
37  d41d8c    // Creates a zero-filled matrix of n rows and m columns.
38  2d2b5d    mat(int n, int m) : vector<vec<T>>(n, vec<T>(m)) {}
39  d41d8c
```

```
40  d41d8c    // c = a * x + b * y
41  762fbc    static void linear_comb(const mat &a, T x, const mat &b, T y,
   mat &c)
42  f95b70    {
43  8fe753      for (int i = 0; i < sz(a); i++)
44  f47ed7        for (int j = 0; j < sz(a[i]); j++)
45  4f844b          c[i][j] = a[i][j] * x + b[i][j] * y;
46  cbb184    }
47  d41d8c
48  d41d8c    // return a * x + b * y
49  08e6ea    static mat linear_comb(mat a, T x, const mat &b, T y)
50  f95b70    {
51  4fec85      linear_comb(a, x, b, y, a);
52  3f5343      return a;
53  cbb184    }
54  d41d8c
55  13fd2a    mat operator-(const mat &b) const { return linear_comb(*this, T
   (1), b, T(-1)); }
56  d41d8c
57  0138fa    mat operator+(const mat &b) const { return linear_comb(*this, T
   (1), b, T(1)); }
58  d41d8c
59  93d3e8    mat operator*(const T &x) { return linear_comb(*this, x, *this,
    T(0)); }
60  d41d8c
61  d41d8c    // Absolutely does not work for int.
62  72c1fd    mat operator/(const T &x) const { return linear_comb(*this, T
   (1) / x, *this, T(0)); }
63  d41d8c
64  d41d8c    // Returns inverse of matrix (assuming it is square and non-
   singular). Runs in O(n^3).
65  d41d8c    // Absolutely does not work for int.
66  14566d    mat inverse() // TODO: test singular.
67  f95b70    {
68  d23a72      int n = sz(*this);
69  bca455      mat a(n, 2 * n);  // A is Nx2N: X|I.
70  f7f2d1      vector<int> col(n); // Will be using column pivoting, so need
    to remember original columns.
71  83008c      for (int i = 0; i < n; i++)
72  f95b70      {
73  f90a6b        for (int j = 0; j < n; j++)
74  c1c7c0          a[i][j] = (*this)[i][j];
75  34ac5b        a[i][i + n] = T(1);
76  6dcd38        col[i] = i;
77  cbb184      }
78  d41d8c
79  83008c      for (int i = 0; i < n; i++)
80  f95b70      {
81  903ccf        int r = i, c = i;
82  775cab        for (int j = i; j < n; j++)
```

```
83  90f1d8          for (int k = i; k < n; k++)
84  f78c7f            if (abs(a[j][k]) > abs(a[r][c]))
85  d4c894              r = j, c = k;
86  d41d8c
87  d41d8c        // assert(abs(a[r][c]) > EPS); Uncomment to check singular
    matrix
88  a2fa24        swap(a[i], a[r]);
89  d41d8c
90  f90a6b        for (int j = 0; j < n; j++)
91  c8cc8f          swap(a[j][i], a[j][c]), swap(a[j][i + n], a[j][c + n]);
92  c1d48e        swap(col[i], col[c]);
93  d41d8c
94  b70d15        vec<T>::linear_comb(a[i], T(1) / a[i][i], a[i], T(0), a[i])
    ;
95  67830d        a[i][i] = T(1);
96  d41d8c
97  197ab1        for (int j = i + 1; j < n; j++)
98  3704dc          vec<T>::linear_comb(a[j], T(1), a[i], -a[j][i], a[j]);
99  cbb184      }
100 d41d8c
101 d41d8c      // Right now A is:
102 d41d8c      //
103 d41d8c      //  1 * *
104 d41d8c      //  0 1 *
105 d41d8c      //  0 0 1
106 d41d8c      //
107 d41d8c      // Next we remove non-1s from right to left.
108 d41d8c
109 917d8b      for (int i = n - 1; i > 0; i--)
110 c791cd        for (int j = 0; j < i; j++)
111 3704dc          vec<T>::linear_comb(a[j], T(1), a[i], -a[j][i], a[j]);
112 d41d8c
113 c70ad2      mat retv(n, n);
114 83008c      for (int i = 0; i < n; i++)
115 f90a6b        for (int j = 0; j < n; j++)
116 4eb40a          retv[col[i]][col[j]] = a[i][j + n];
117 6272cf      return retv;
118 cbb184    }
119 2145c1  };
```

7.6 Simplex

```
1 5d1131  #include "../../contest/header.hpp"
2 d41d8c
3 d41d8c  /*
4 458b90    Simplex:
5 6956ec      Optimizes a linear program of the form:
6 15b127        maximize c*x, s.t. a*x <ops> b, x >= 0.
7 7b88d6      Each constraint can use a different operator from {<= >= ==}.
8 8aa76d      Not polynomial, but got AC 150 ms with 4000 constraints and 200
```

```
    variables.
9 d41d8c
10  b95cae    Usage:
11  e8b3b7      Call run_simplex, with the number of constraints and
   variables, a, b, ops and c (as specified above).
12  34036d      Return value is ok if solution was found, unbounded if
   objective value can be infinitely large
13  eb42f2      or infeasible if there is no solution given the constraints.
14  d41d8c
15  2baa60      The value of each variable is returned in vector res.
   Objective function optimal value is also returned.
16  060dc4      Sample usage is commented below.
17  d41d8c
18  3db72f    Author: Arthur Pratti Dadalto
19  c4c9bd  */
20  d41d8c
21  4fce64  template <class T>
22  fe4002  struct vec : vector<T>
23  f95b70  {
24  469362    vec(int n) : vector<T>(n) {}
25  d41d8c
26  d41d8c    // c = a*x + b*y
27  e918cb    static void linear_comb(const vec &a, T x, const vec &b, T y,
   vec &c)
28  f95b70    {
29  8fe753      for (int i = 0; i < sz(a); i++)
30  75e753        c[i] = a[i] * x + b[i] * y;
31  cbb184    }
32  2145c1  };
33  d41d8c
34  4fce64  template <class T>
35  dade1f  struct mat : vector<vec<T>>
36  f95b70  {
37  d41d8c    // Creates a zero-filled matrix of n rows and m columns.
38  2d2b5d    mat(int n, int m) : vector<vec<T>>(n, vec<T>(m)) {}
39  d41d8c
40  d41d8c    // Erase row O(n^2).
41  82436c      void erase_row(int i)
42  f95b70      {
43  7c9f9f          this->erase(this->begin() + i);
44  cbb184      }
45  d41d8c
46  d41d8c    // Erase column O(n^2).
47  1b22c6      void erase_col(int j)
48  f95b70      {
49  798fc8          for (int i = 0; i < sz(*this); i++)
50  a7796a              (*this)[i].erase((*this)[i].begin() + j);
51  cbb184      }
52  2145c1  };
53  d41d8c
```

```
54  d3ff82  namespace simplex
55  f95b70  {
56  d41d8c  // Any value within [-EPS, +EPS] will be considered equal to 0.
57  05667a  const double EPS = 1e-6;
58  d41d8c
59  5e6f5b  enum op { ge, le, eq };
60  d41d8c
61  242dbb  enum optimization_status { ok, unbouded, infeasible };
62  d41d8c
63  4d9580  int get_entering_var(mat<double> &tab)
64  f95b70  {
65  d41d8c    // Get first non-artificial variable with negative objective
   coeficient. If none, return -1. (could instead return most negative, but
   that could cycle)
66  682f62    for (int i = 0; i < sz(tab[0]) - 1; i++)
67  72e0d2      if (tab[0][i] < -EPS)
68  d9a594        return i;
69  daa4d1    return -1;
70  cbb184  }
71  d41d8c
72  201003  int get_exiting_var_row(mat<double> &tab, int entering_var)
73  f95b70  {
74  d41d8c    // Get smallest value of val and first in case of tie. If none,
    return -1.
75  fcb2fc    int retv = -1;
76  6213b9    double val = -1.0;
77  a07064    for (int i = 1; i < sz(tab); i++)
78  f95b70    {
79  d41d8c      // If strictly positive, it bounds the entering var.
80  dcda72      if (tab[i][entering_var] > EPS)
81  f95b70      {
82  d41d8c        // Entering var will be bounded by tab[i][tab.size().second
    - 1] / tab[i][entering_var].
83  d41d8c        // val could be slightly negative if tab[i][tab.size().
   second - 1] = -0.
84  393d3f        if (val == -1.0 || tab[i][sz(tab[i]) - 1] / tab[i][
   entering_var] < val)
85  f95b70        {
86  78d87c          val = tab[i][sz(tab[i]) - 1] / tab[i][entering_var];
87  52cece          retv = i;
88  cbb184        }
89  cbb184      }
90  cbb184    }
91  d41d8c
92  6272cf    return retv;
93  cbb184  }
94  d41d8c
95  ed25d2  optimization_status solve_tab(mat<double> &tab, vector<int> &
   basic_var)
96  f95b70  {
```

```
97  d41d8c    // artificial_count is the number of variables at the end we
   should ignore.
98  a17ec7    int entering_var;
99  6b7846    while ((entering_var = get_entering_var(tab)) != -1)
100 f95b70    {
101 6c0a23      int exiting_var_row = get_exiting_var_row(tab, entering_var);
102 d41d8c
103 d41d8c      // If no exiting variable bounds the entering variable, the
   objective is unbounded.
104 813335      if (exiting_var_row == -1)
105 914a2e        return optimization_status::unbouded;
106 d41d8c
107 d41d8c      // Set new basic var coeficient to 1.
108 89c7a2      vec<double>::linear_comb(tab[exiting_var_row], (1.0 / tab[
   exiting_var_row][entering_var]), tab[exiting_var_row], 0.0, tab[
   exiting_var_row]);
109 d41d8c
110 d41d8c      // Gaussian elimination of the other rows.
111 c7a773      for (int i = 0; i < sz(tab); i++)
112 81c379        if (i != exiting_var_row)
113 ed2730          if (abs(tab[i][entering_var]) > EPS)
114 7ad878            vec<double>::linear_comb(tab[i], 1.0, tab[
   exiting_var_row], -tab[i][entering_var], tab[i]);
115 d41d8c
116 64dd6a      basic_var[exiting_var_row] = entering_var;
117 cbb184    }
118 d41d8c
119 c52f1c    return optimization_status::ok;
120 cbb184  }
121 d41d8c
122 d41d8c  // maximize c*x, s.t. a*x <ops> b. x >= 0.
123 f1a105  optimization_status run_simplex(int num_constraints, int num_vars
   , mat<double> a, vec<op> ops, vec<double> b, vec<double> c, vec<double> &
   res, double &obj_val)
124 f95b70  {
125 334f46    for (int i = 0; i < num_constraints; i++)
126 5f946c      if (ops[i] == op::ge)
127 f95b70      {
128 d41d8c        // Beyond this point "ge" constraints won't exist.
129 44438f        vec<double>::linear_comb(a[i], -1, a[i], 0, a[i]); // a[i]
   *= -1;
130 250b4d        b[i] *= -1;
131 1c38d4        ops[i] = op::le;
132 cbb184      }
133 d41d8c
134 0264da    int num_artificial_variables = 0;
135 371f2b    int num_slack_variables = 0;
136 334f46    for (int i = 0; i < num_constraints; i++)
137 f95b70    {
138 0ec40f      if (ops[i] == op::le)
```

```
139 f95b70      {
140 37acf9        num_slack_variables++;
141 cbb184      }
142 d41d8c
143 359aa4      if ((ops[i] == op::le && b[i] < -EPS) || ops[i] == op::eq)
144 f95b70      {
145 d41d8c        // If we have rhs strictly negative in a inequality or an
   equality constraint, we need an artificial val.
146 fc36e6        num_artificial_variables++;
147 cbb184      }
148 cbb184    }
149 d41d8c
150 854c33    mat<double> tab(num_constraints + 1, num_vars +
   num_slack_variables + num_artificial_variables + 1);
151 9a9a70    vector<int> basic_var(num_constraints + 1);
152 775265    vector<int> slack_cols, artificial_cols;
153 7f63aa    for (int i = num_vars; i < num_vars + num_slack_variables; i++)
154 10c71f      slack_cols.push_back(i);
155 e0b615    for (int i = num_vars + num_slack_variables; i < num_vars +
   num_slack_variables + num_artificial_variables; i++)
156 eafbfb      artificial_cols.push_back(i);
157 c70a50    int rhs_col = num_vars + num_slack_variables +
   num_artificial_variables;
158 d41d8c
159 d41d8c    // First objective will be to have artificial variables equal
   to 0.
160 017565    for (int i : artificial_cols)
161 b98201      tab[0][i] = 1;
162 d41d8c
163 9c49f5    for (int i = 0, k = 0, l = 0; i < num_constraints; i++)
164 f95b70    {
165 861a15      for (int j = 0; j < num_vars; j++)
166 e3832e        tab[i + 1][j] = a[i][j];
167 d41d8c
168 0ec40f      if (ops[i] == op::le)
169 141495        tab[i + 1][slack_cols[l++]] = 1;
170 d41d8c
171 142f37      tab[i + 1][rhs_col] = b[i];
172 d41d8c
173 359aa4      if ((ops[i] == op::le && b[i] < -EPS) || ops[i] == op::eq) //
    Basic var will be artificial
174 f95b70      {
175 2a6978        if (b[i] < -EPS)
176 009fda          vec<double>::linear_comb(tab[i + 1], -1, tab[i + 1], 0,
   tab[i + 1]); // a[i] *= -1;
177 d41d8c
178 86fab4        tab[i + 1][artificial_cols[k++]] = 1;
179 116454        basic_var[i + 1] = artificial_cols[k - 1];
180 d41d8c
181 06db08        vec<double>::linear_comb(tab[0], 1.0, tab[i + 1], -1.0, tab
```

```
   [0]);
182 cbb184       }
183 2954e9       else // Basic var will be slack var.
184 f95b70       {
185 ae77b6         basic_var[i + 1] = slack_cols[l - 1];
186 cbb184       }
187 cbb184     }
188 d41d8c
189 df8d17     assert(solve_tab(tab, basic_var) == optimization_status::ok);
190 d41d8c
191 d41d8c     // Best solution could not bring artificial variables to 0 (
   objective max Z = sum(-xa)).
192 fe0d64     if (tab[0][sz(tab[0]) - 1] < -EPS)
193 94b8a3       return optimization_status::infeasible;
194 d41d8c
195 d41d8c     // If we have an artificial variable on the base with xb = 0,
   we need to remove it.
196 e6411b     for (int i = 1; i < sz(basic_var); i++)
197 0778cb       if (basic_var[i] >= num_vars + num_slack_variables)
198 f95b70       {
199 d41d8c         // Find non-artificial replacement.
200 e2f213         for (int j = 0; j < sz(tab[i]) - 1 -
   num_artificial_variables; j++)
201 f95b70         {
202 d41d8c           // If non-zero value in row, we can replace.
203 a8880b           if (j != basic_var[i] && abs(tab[i][j]) > EPS)
204 f95b70           {
205 d41d8c             // Remove from the other rows.
206 b5fa44             vec<double>::linear_comb(tab[i], 1.0 / tab[i][j], tab[i
   ], 0, tab[i]);
207 d41d8c
208 443db5             for (int k = 0; k < sz(tab); k++)
209 635b4c               if (k != i)
210 f95b70               {
211 e76184                 if (abs(tab[k][j]) > EPS)
212 4b6b27                   vec<double>::linear_comb(tab[k], 1, tab[i], -tab[
   k][j], tab[k]);
213 cbb184               }
214 d41d8c
215 d41d8c             // Basic variable replacemente done, so proceed to next
    basic_var.
216 7e0f27             basic_var[i] = j;
217 c2bef1             break;
218 cbb184           }
219 cbb184         }
220 cbb184       }
221 d41d8c
222 ca2210     for (int i = sz(tab) - 1; i > 0; i--)
223 0778cb       if (basic_var[i] >= num_vars + num_slack_variables)
224 f95b70       {
```

```
225 d41d8c        // Could not replace basic var, so constraint is redundant.
226 2cd1fb        tab.erase_row(i);
227 fe14c7        basic_var.erase(basic_var.begin() + i);
228 cbb184      }
229 d41d8c
230 d41d8c    // Remove artificial variable columns.
231 5c3178    for (int i = sz(artificial_cols) - 1; i >= 0; i--)
232 9a226e      tab.erase_col(artificial_cols[i]);
233 d41d8c
234 1311b7    for (int i = 0; i < sz(tab[0]); i++)
235 d2677f      tab[0][i] = 0;
236 f17293    for (int i = 0; i < num_vars; i++)
237 94256d      tab[0][i] = -c[i];
238 d41d8c
239 a07064    for (int i = 1; i < sz(tab); i++)
240 b39526      vec<double>::linear_comb(tab[0], 1, tab[i], -tab[0][basic_var
    [i]], tab[0]);
241 d41d8c
242 54ad02    optimization_status status = solve_tab(tab, basic_var);
243 d41d8c
244 b68670    res = vec<double>(num_vars);
245 e6411b    for (int i = 1; i < sz(basic_var); i++)
246 047b20      if (basic_var[i] < num_vars)
247 81f54e        res[basic_var[i]] = tab[i][sz(tab[i]) - 1];
248 d41d8c
249 a3473e    obj_val = tab[0][sz(tab[0]) - 1];
250 d41d8c
251 62d3d5    return status;
252 cbb184  }
253 cbb184  } // namespace simplex
254 d41d8c
255 d41d8c  /*
256 13a4b1  int main(void)
257 f95b70  {
258 14e0a7    int n, m;
259 aa3380    cin >> n >> m;
260 d41d8c
261 37ce14    int num_constraints = m, num_vars = n;
262 d41d8c
263 d41d8c    // maximize c*x, s.t. a*x <ops> b. x >= 0.
264 2626bb    mat<double> a(num_constraints, num_vars);
265 84d434    vec<double> b(num_constraints);
266 01b2af    vec<simplex::op> ops(num_constraints);
267 dabb12    vec<double> c(num_vars);
268 40ca17    vec<double> res(num_vars);
269 d41d8c
270 83008c    for (int i = 0; i < n; i++)
271 a733f7      cin >> c[i];
272 d41d8c
273 94f72b    for (int i = 0; i < m; i++)
```

```
274 f95b70    {
275 7ba74c      int l, r, x;
276 15994b      cin >> l >> r >> x;
277 0dfebd      for (int j = l - 1; j <= r - 1; j++)
278 a21125        a[i][j] = 1;
279 df0b9d      b[i] = x;
280 80367f      ops[i] = simplex::op::le;
281 cbb184    }
282 d41d8c
283 1afc12    double ans;
284 dd6c28    simplex::run_simplex(num_constraints, num_vars, a, ops, b, c,
   res, ans);
285 d41d8c
286 530b75    cout << ((long long)(ans + 0.5)) << endl;
287 cbb184  }
288 c4c9bd  */
```

8 String

8.1 KMP

```
#include "../../contest/header.hpp"

/*
  Prefix Function and KMP:
    Computes prefix function for a given string in O(n).
    String matching in O(n + m).
    No need to be strings, you can use vector<int> since the
    algorithms don't depend on the alphabet size, they only perform
    equality comparisons.
    Usage is explained in each function.

    Author: Arthur Pratti Dadalto
*/

// Returns the prefix function for the given string.
// pi[i] for 0 <= i <= s.size() (s.size() + 1 elements).
// pi[i] considers the prefix of string s having size i.
// pi[i] is the size of its (the prefix's) largest proper prefix
  which is also a suffix.
// For "aabaaab", pi is is {0,0,1,0,1,2,2,3}
template <class T>
vector<int> prefix_function(T s)
{
  vector<int> pi(s.size() + 1, 0);
  for (int i = 2; i <= s.size(); i++)
  {
    int j = pi[i - 1];          // j is the size of the candidate
    prefix to expand.
    while (j > 0 && s[j] != s[i - 1]) // While we still have a
    candidate prefix and it can't be expanded.
      j = pi[j];            // Go to the next candidate prefix.

    // If candidate prefix can be expanded, do it. Otherwise,
    there is no prefix that is also a suffix.
    pi[i] = s[j] == s[i - 1] ? j + 1 : 0;
  }

  return pi;
}

// Returns a sorted list of all positions in the text string
  where begins an ocurrence of the key string.
// e.g. kmp("aabaaab", "aab") returns {0, 4}.
template <class T>
vector<int> kmp(T text, T key)
{
```

```
41  aeb888    vector<int> retv;
42  7fa638    vector<int> pi = prefix_function(key);
43  5d936d    for (int i = 0, match = 0; i < text.size(); i++) // There is no
    need to have the entire text in memory, you could do this char by char.
44  f95b70    {
45  d41d8c      // match stores the size of the prefix of the key which is a
   suffix of the current processed text.
46  9d984d      while (match > 0 && text[i] != key[match])
47  7eb4cc        match = pi[match];
48  db8319      if (text[i] == key[match])
49  24b638        match++;
50  d41d8c
51  dd8c14      if (match == key.size())
52  f95b70      {
53  7b8421        retv.push_back(i - match + 1);
54  7eb4cc        match = pi[match]; // To avoid access to key[key.size()] in
    next iteration.
55  cbb184      }
56  cbb184    }
57  d41d8c
58  6272cf    return retv;
59  cbb184  }
```

8.2 Acho Corasick

```
1 5d1131  #include "../../contest/header.hpp"
2 d41d8c
3 d41d8c  /*
4 30562e    Aho-Corasick: O(alpha_size * string_sum)
5 4e9057      In general, multiple pattern string matching tree/automaton.
6 d41d8c
7 fbc6b5      Keep in mind that find_all can be O(N*sqrt(N)) if no duplicate
   patterns. (N is total string length)
8 d41d8c
9 ca2095    Constraints:
10  00d37a      chars in the string are all in the interval [first, first +
   alpha_size - 1].
11  3da079      This will not free some memory on object destruction.
12  390590      Duplicate patterns are allowed, empty patterns are not.
13  d41d8c
14  b95cae    Usage:
15  df3a72      Set alpha_size and the first char in the alphabet.
16  e98cb2      Call constructor passing the list of pattern strings.
17  0a657b      Use one of find, find_all ... to process a text or do your
   own thing.
18  acdd39      To find the longest words that start at each position,
   reverse all input.
19  3439d1      Bottleneck in this code is memory allocation.
20  91a84c      For 10^6 total string size, memory usage can be up to 300 Mb.
21  d41d8c
```

```
22  b34145      You can save time:
23  93df09        list_node, match_list, match_list_last are only needed to
   list all matches.
24  57e4db        atm automaton table can be cut to reduce memory usage.
25  018d49        The text processing stuff is also optional.
26  02e3ad        Node memory can be one big array instead of vector.
27  d41d8c
28  3db72f    Author: Arthur Pratti Dadalto
29  c4c9bd  */
30  d41d8c
31  e7f92a  struct aho_corasick
32  f95b70  {
33  da45ec    enum
34  f95b70    {
35  033315      alpha_size = 26,  // Number of chars in the alphabet.
36  b3d02f      first = 'a'     // First char.
37  2145c1    };
38  d41d8c
39  fc487b    struct list_node    // Simple linked list node struct.
40  f95b70    {
41  53e65f      int id;
42  6ec94b      list_node *next;
43  ff56a7      explicit list_node(int id, list_node *next) : id(id), next(
   next) {}
44  2145c1    };
45  d41d8c
46  e4accb    struct node
47  f95b70    {
48  ca8b7e      int fail = -1;      // node failure link (aka suffix link).
49  2eb620      int nmatches = 0;   // Number of matches ending in this node.
50  9005b9      int next[alpha_size]; // Next node in trie for each letter.
   Replace with unordered_map or list if memory is tight.
51  c0f747      int atm[alpha_size];  // Optional: Automaton state transition
    table. Simpler text processing.
52  d41d8c
53  44edb6      list_node *match_list = nullptr;    // Pointer to first node
   in linked list of matches. List ends with null pointer.
54  01009c      list_node *match_list_last = nullptr; // Internal: pointer to
    last node in list of matches (before bfs), or null if empty list.
55  d41d8c
56  e6fb82      node() { memset(next, -1, sizeof(next)); } // Start with all
   invalid transitions.
57  2145c1    };
58  d41d8c
59  b9ea22    vector<node> nodes;
60  d41d8c
61  9b61f6    aho_corasick(const vector<string> &pats)
62  f95b70    {
63  225eb3      nodes.emplace_back(); // Make root node 0.
64  b5bf96      for (int i = 0; i < sz(pats); i++)
```

```
65  f95b70      {
66  b3da3c        int cur = 0;  // Start from root.
67  9f5c69        for (int j = 0; j < sz(pats[i]); j++)
68  f95b70        {
69  ec0388          int k = pats[i][j] - first;
70  d41d8c
71  10937b          if (nodes[cur].next[k] <= 0)  // Make new node if needed.
72  f95b70          {
73  976fa3            nodes[cur].next[k] = sz(nodes);
74  225eb3            nodes.emplace_back();
75  cbb184          }
76  d41d8c
77  47b49f          cur = nodes[cur].next[k];
78  cbb184        }
79  d41d8c
80  d41d8c        // Add logic here if additional data is needed on matched
    strings.
81  4daeea        nodes[cur].nmatches++;
82  45f177        nodes[cur].match_list = new list_node(i, nodes[cur].
    match_list);  // Add string to node list of matches.
83  fe38fe        if (nodes[cur].nmatches == 1)
84  947da5          nodes[cur].match_list_last = nodes[cur].match_list;
85  cbb184      }
86  d41d8c
87  26a528      queue<int> q;
88  6733a6      for (int i = 0; i < alpha_size; i++) // Define fail for first
     level.
89  f95b70      {
90  e8dc83        if (nodes[0].next[i] == -1) // Invalid transitions from 0
    now become valid self transitions.
91  fb628f          nodes[0].next[i] = 0;
92  d41d8c
93  7d3171        nodes[0].atm[i] = nodes[0].next[i]; // Automaton state
    transition table.
94  d41d8c
95  bc34bf        if (nodes[0].next[i] > 0) // Single letter nodes have fail
    = 0 and go in the queue.
96  f95b70        {
97  eded92          q.push(nodes[0].next[i]);
98  9b22e6          nodes[nodes[0].next[i]].fail = 0;
99  cbb184        }
100 cbb184      }
101 d41d8c
102 ee6bdd      while (!q.empty()) // Use bfs to compute fail for next level.
103 f95b70      {
104 69faa7        int cur = q.front();
105 833270        q.pop();
106 d41d8c
107 6733a6        for (int i = 0; i < alpha_size; i++)
108 af4a6e          if (nodes[cur].next[i] > 0) // Don't use -1 and don't use
```

```
    transition to root.
109 f95b70          {
110 3ecdd3            nodes[cur].atm[i] = nodes[cur].next[i]; // Unrelated to
    code below, filling automaton.
111 d41d8c
112 d41d8c            // Computing fail for next node and putting it in the
   queue.
113 3ae7da            int prox = nodes[cur].next[i];
114 53ef92            q.push(prox);
115 d41d8c
116 f252cb            int state = nodes[cur].fail;
117 c66324            while (nodes[state].next[i] == -1)
118 d712e2              state = nodes[state].fail;
119 d41d8c
120 7836db            nodes[prox].fail = nodes[state].next[i];
121 d41d8c
122 d41d8c            // Add logic here if additional data is needed on
   matched strings.
123 2940ed            nodes[prox].nmatches += nodes[nodes[prox].fail].
   nmatches;
124 d41d8c
125 d41d8c            // Add in O(1) list from fail link to next node's list.
    Operation: a->b->null c->null to a->b->c->null.
126 59ed4d            (nodes[prox].match_list_last ? nodes[prox].
   match_list_last->next : nodes[prox].match_list) = nodes[nodes[prox].fail].
   match_list;
127 cbb184          }
128 2954e9          else
129 f95b70          {
130 a04598            nodes[cur].atm[i] = nodes[nodes[cur].fail].atm[i];
131 cbb184          }
132 cbb184      }
133 cbb184    }
134 d41d8c
135 d41d8c    // Optional
136 d41d8c    // Returns a vector retv such that, for each text position i:
137 d41d8c    // retv[i] is the index of the largest pattern ending at
   position i in the text.
138 d41d8c    // If retv[i] == -1, no pattern ends at position i.
139 32246d    vector<int> find(const string &text)
140 f95b70    {
141 107323      vector<int> retv(sz(text));
142 b3da3c      int cur = 0;
143 d41d8c
144 77447e      for (int i = 0; i < sz(text); i++)
145 f95b70      {
146 13dae2        cur = nodes[cur].atm[text[i] - first];
147 29e58f        retv[i] = (nodes[cur].match_list ? nodes[cur].match_list->
   id : -1);
148 cbb184      }
```

```
149 d41d8c
150 6272cf      return retv;
151 cbb184    }
152 d41d8c
153 d41d8c    // Optional
154 d41d8c    // Returns a vector retv such that, for each text position i:
155 d41d8c    // retv[i] is the number of pattern matches ending at position
    i in the text.
156 48d0f2    vector<int> count(const string &text)
157 f95b70    {
158 107323      vector<int> retv(sz(text));
159 b3da3c      int cur = 0;
160 d41d8c
161 77447e      for (int i = 0; i < sz(text); i++)
162 f95b70      {
163 13dae2        cur = nodes[cur].atm[text[i] - first];
164 1a43d3        retv[i] = nodes[cur].nmatches;
165 cbb184      }
166 d41d8c
167 6272cf      return retv;
168 cbb184    }
169 d41d8c
170 d41d8c    // Optional
171 d41d8c    // Returns a vector retv such that, for each text position i:
172 d41d8c    // retv[i] is a list of indexes to the patterns ending at
    position i in the text.
173 d41d8c    // These lists will be sorted from largest to smallest pattern
    length.
174 d41d8c    // Keep in mind that find_all can be O(N*sqrt(N)) if no
    duplicate patterns. (N is total string length)
175 4e5a4c    vector<vector<int>> find_all(const string &text)
176 f95b70    {
177 77b54a      vector<vector<int>> retv(sz(text));
178 b3da3c      int cur = 0;
179 d41d8c
180 77447e      for (int i = 0; i < sz(text); i++)
181 f95b70      {
182 13dae2        cur = nodes[cur].atm[text[i] - first];
183 d82b0e        for (auto n = nodes[cur].match_list; n != nullptr; n = n->
    next)
184 4c4784          retv[i].push_back(n->id);
185 cbb184      }
186 d41d8c
187 6272cf      return retv;
188 cbb184    }
189 d41d8c
190 d41d8c    // Optional
191 d41d8c    // Returns a vector retv such that:
192 d41d8c    // retv is a list of indexes to the patterns ending at position
     pos in the text.
```

```
193 d41d8c    // This list will be sorted from largest to smallest pattern
   length.
194 251c66    vector<int> find_all_at_pos(const string &text, int pos)
195 f95b70    {
196 aeb888      vector<int> retv;
197 b3da3c      int cur = 0;
198 d41d8c
199 77447e      for (int i = 0; i < sz(text); i++)
200 f95b70      {
201 13dae2        cur = nodes[cur].atm[text[i] - first];
202 d41d8c
203 c57c6f        if (i == pos)
204 d82b0e          for (auto n = nodes[cur].match_list; n != nullptr; n = n
   ->next)
205 1ad617            retv.push_back(n->id);
206 cbb184      }
207 d41d8c
208 6272cf      return retv;
209 cbb184    }
210 2145c1  };
```

8.3 Suffix Array

```
1 d41d8c
2 5d1131  #include "../../contest/header.hpp"
3 d41d8c
4 d41d8c  /*
5 1f77e9    Suffix array:
6 be00ca      Build suffix array and LCP array in O((n + lim) log n) using O(
   n + lim) memory, where lim is the alphabet size.
7 d41d8c
8 643e15      sa[i] is the starting index of the suffix which is i-th in the
   sorted suffix array.
9 0d5e62      The returned vector is of size s.size()+1, and sa[0] == s.size
   (). The '\0' char at the end is considered
10  29ea73      part of the string, so sa[0] = "\0", the prefix starting at
   index s.size().
11  d41d8c
12  ee7035      The lcp array contains longest common prefixes for
   neighbouring strings
13  317e94      in the suffix array: lcp[i] = lcp(sa[i], sa[i-1]), lcp[0] =
   0.
14  d41d8c
15  81eeab    Example:
16  981b73      Computing the LCP and the SA of "GATAGACA"
17  d33e7b        i sa[i] lcp[i]  suffix
18  fd774f        0 8   0   ""
19  cac682        1 7   0   "A"
20  430b3a        2 5   1   "ACA"
21  d30cc0        3 3   1   "AGACA"
```

```
22  c895f5        4 1   1   "ATAGACA"
23  1a04b3        5 6   0   "CA"
24  b1b780        6 4   0   "GACA"
25  2999cd        7 0   2   "GATAGACA"
26  08e6dc        8 2   0   "TAGACA"
27  d41d8c
28  b95cae    Usage:
29  1c63e0      Important: the input string must not contain any zero values.
    Must use C++11 or above.
30  b847d4      You can use this for strings of integers, just change the
   alphabet size.
31  d41d8c
32  1d1558    Source: https://github.com/kth-competitive-programming/kactl/
   blob/master/content/strings/SuffixTree.h
33  c4c9bd  */
34  d41d8c
35  15a9b6  struct suffix_array
36  f95b70  {
37  71675a    vector<int> sa, lcp;
38  092958    suffix_array(const string &s, int lim = 256) // or basic_string
   <int> for integer strings.
39  f95b70    {
40  e72340      int n = sz(s) + 1, k = 0, a, b;
41  f6a0db      vector<int> x(s.begin(), s.end() + 1), y(n), ws(max(n, lim)),
    rank(n);
42  85469f      sa = lcp = y;
43  eb75f9      iota(sa.begin(), sa.end(), 0);
44  7707f7      for (int j = 0, p = 0; p < n; j = max(1, j * 2), lim = p)
45  f95b70      {
46  8dff9b        p = j;
47  00aec0        iota(y.begin(), y.end(), n - j);
48  83008c        for (int i = 0; i < n; i++)
49  e9b19c          if (sa[i] >= j)
50  d0873d            y[p++] = sa[i] - j;
51  450a8a        fill(ws.begin(), ws.end(), 0);
52  83008c        for (int i = 0; i < n; i++)
53  799bb0          ws[x[i]]++;
54  7d6bd3        for (int i = 1; i < lim; i++)
55  f256af          ws[i] += ws[i - 1];
56  5df399        for (int i = n; i--;)
57  d01b67          sa[--ws[x[y[i]]]] = y[i];
58  9dd20c        swap(x, y);
59  017be6        p = 1;
60  16ab1b        x[sa[0]] = 0;
61  d41d8c
62  aa4866        for (int i = 1; i < n; i++)
63  f95b70        {
64  fcb940          a = sa[i - 1];
65  2d820b          b = sa[i];
66  0cc036          x[b] = (y[a] == y[b] && y[a + j] == y[b + j]) ? p - 1 : p
```

```
   ++;
67  cbb184        }
68  cbb184      }
69  d41d8c
70  aa4866      for (int i = 1; i < n; i++)
71  2f33c5        rank[sa[i]] = i;
72  d41d8c
73  05cb2b      for (int i = 0, j; i < n - 1; lcp[rank[i++]] = k)
74  487069        for (k &&k--, j = sa[rank[i] - 1]; s[i + k] == s[j + k]; k
   ++)
75  9eecb7          ;
76  cbb184    }
77  2145c1  };
```