BACHELOR'S THESIS

Degree in Mathematics

Title: Analysis and solution of a collection of algorithmic problems

Author: Rafael Eusebio López Martínez

Advisor: Salvador Roura Ferret

Department: Computer Science

Academic year 2017/2018

UNIVERSITAT POLITÈCNICA DE CATALUNYA
BARCELONATECH
Facultat de Matemàtiques i Estadística

Universitat Politècnica de Catalunya
Facultat de Matemàtiques i Estadística

Degree in Mathematics

Bachelor's Degree Thesis

Analysis and solution of a collection of algorithmic problems

Rafael Eusebio López Martínez

Supervised by Salvador Roura Ferret

January, 2018

Thanks to the people who have been with me in the journey of competitive programming, including but not limited to Salvador Roura, Albert Martínez, Enrique Jiménez, Ángel García and Sergio Rodríguez.

Thanks to CFIS for bringing me to Barcelona and giving more support and opportunities than necessary.

Thanks to mom and dad for giving me everything.

Abstract

In competitive programming, one has to use knowledge in algorithms and data structures to find solutions to algorithmic problems, then put those a ideas into a correct computer program that solves the problem within given time and memory constraints. This activity involves learning about a wide range of complex data structures and algorithms, and many hours of training.

This project offers insight into the solutions for problems from different programming contests, using algorithms and data structures from different relevant topics in competitive programming. Most of these problems were solved while training with the UPC competitive programming teams, which have dominated their regional competition for more than one decade.

Keywords

algorithm, competitive programming, programming contest, graph, dynamic programming, data structure, C++

Contents

Introduction

Context

Competitive programming is a mind sport which involves participants programming, as quickly as possible, efficient solutions to given algorithmic problems. Often, the statements for the problems are written as to appear like real world problems. Solving these problems usually needs of previous knowledge of many algorithms and data structures.

The solutions to these problems consist in programs which read some input data, compute the solution for the problem and the given input, and write the answer asked by the problem. Solutions are evaluated by automatic judges that compile and execute the submitted program and compare the output against the official solution to determine its correctness. The solution does not only have to be correct but it also has to be efficient; programs that take too much time or use too much memory aren't accepted as valid solutions. For example, most problems in recent Codeforces rounds have time limits of 1 or 2 seconds of running time and a memory limit of 256 MB.

Competitive programming has a huge community. Several resources dedicated to competitive programming have been created and mantained by the community around the world.

There are some communities that hold programming contests regularly; the most famous being probably Codeforces [4], Topcoder [5] and Codechef [3]. These websites tend to have an ELO-like ranking system for their users based on their performance in the regular competitions.

There are also websites that do not usually run competitions, but provide a a large list of problems set up in an online judge to evaluate solutions. UVa Online Judge [10] has one of the biggest collections of problems, and the ACM-ICPC Live Archive [2] contains a list with most of the problems used in past ACM-ICPC Regional and World Finals contests. In this project there are also problems from Timus Online Judge [9] and Sphere Online Judge [8].

Some of the largest tech companies set up their own annual competitions, mainly as a way to look for skilled potential employees, since these companies are very interested in the abilities needed for competitive programming. The most popular of this kind of competitions are Google Code Jam by Google [7], Facebook HackerCup by Facebook [6] and Yandex Algorithm Contest by Yandex [11].

One of the oldest competitions is the ACM International Collegiate Programming Contest (ACM-ICPC) [1]. In it, teams of 3 students from universities from all over the world participate. The UPC holds some training activities with the goal of preparing its teams for the ACM-ICPC contests. The UPC has participated in this contest since 2002, most of the times qualifying for the World Finals.

Objective and scope

The work in this project could be considered part of the training for the ACM ICPC Regional contest. The product is a collection of problems from a wide range of topics very relevant in competitive programming with an analysis and C++ code of the solution.

In this project there are solutions for some problems created by others, obtained from the different resources mentioned before. The analysis of those problems will be done by the author of the project. Most of the code in the solutions is writen by the author.

Since we cannot cover all of the many topics that appear nowadays in competitve programing in a single project, we are forced to choose a number of topics, and at least one problem from each topic.

The project will try to cover important, recent and interesting issues in the topics of computational geometry, graph algorithms, square root decomposition and strings. Knowledge from part of the reader of the concept of dynamic programming might be assumed in some of the issues or problems.

Methodology and rigor

This project has been in part developed as the author took part in the training sessions for the UPC teams, where participants practiced solving problems as preparation for the ACM ICPC regional competitions. This has been done, for many years, by choosing problems from past contests available in one of the aforementioned online resources, and simulating a contest environment; where the contestants have to read the problems, understand them, sometimes discuss them, find a solution and write the code that performs it correctly and efficiently, all within limited time.

These training sessions have been taking place three times per week and five hours per session, during some months before the relevant competitions, every year. The director of this project is the coach of the UPC teams for the ACM ICPC, and he often gives advice not only about some of the harder problems, but also in how to approach the contest environment and how to achieve efficient team cooperation.

The director will watch over the analysis of the problems in this document to ensure their correctness.

Strings

Z-Algorithm

Given a string S, the Z-Algorithm computes an array Z such that $Z[i]$ is the length of the longest substring starting from $S[i]$ that is a prefix of S. It can be built in linear time.

The most common application is string matching; that is, you can search for a pattern T in a string S by running the Z-Algorithm in string $T + \Psi + S$ where Ψ is a character that matches nothing (and $+$ represents string concatenation). Then those positions i in which $Z[i]$ equals the length of T are matches.

Codeforces 471 - MUH and Cube Walls

Statement: Polar bears Menshykov and Uslada from the zoo of St. Petersburg and elephant Horace from the zoo of Kiev got hold of lots of wooden cubes somewhere. They started making cube towers by placing the cubes one on top of the other. They defined multiple towers standing in a line as a wall. A wall can consist of towers of different heights.

Horace was the first to finish making his wall. He called his wall an elephant. The wall consists of w towers. The bears also finished making their wall but they didn't give it a name. Their wall consists of n towers. Horace looked at the bears' tower and wondered: in how many parts of the wall can he "see an elephant"? He can "see an elephant" on a segment of w contiguous towers if the heights of the towers on the segment match as a sequence the heights of the towers in Horace's wall. In order to see as many elephants as possible, Horace can raise and lower his wall. He even can lower the wall below the ground level.

Your task is to count the number of segments where Horace can "see an elephant".

Solution: To be able to raise and lower the wall, we will only look at the differences between adjacent towers. This way, it becomes a simple pattern matching problem: you have to count the matches of the array of differences of the elephant wall inside the array of differences of the bear wall.

Listing 1: Solution for MUH and Cube Walls

```
#include <iostream>
#include <vector>
using namespace std;

typedef vector<int> vi;

int main()
{
  ios::sync_with_stdio(false);
  int n, w; cin >> n >> w;
  vi a(n); for (int& x : a) cin >> x;
  vi b(w); for (int& x : b) cin >> x;

  if (w < 2)
  {
    cout << n << endl;
    return 0;
  }

  int N = n+w-1;
  vi s(N), z(N);
  for (int i = 0; i+1 < w; ++i) s[i] = b[i+1]-b[i];
  s[w-1] = 1e9;
```

```
  for (int i = 0; i+1 < n; ++i) s[w+i] = a[i+1]-a[i];
  int res = 0;

  int L = 0, R = 0;
  for (int i = 1; i < N; i++)
  {
    if (i > R)
    {
      L = R = i;
      while (R < N && s[R-L] == s[R]) R++;
      z[i] = R-L; R--;
    }
    else
    {
      int k = i-L;
      if (z[k] < R-i+1) z[i] = z[k];
      else
      {
        L = i;
        while (R < N && s[R-L] == s[R]) R++;
        z[i] = R-L; R--;
      }
    }
    if (z[i]+1 >= w) ++res;
  }

  cout << res << endl;
}
```

KMP

The Knuth-Morris-Pratt (or KMP) algorithm is another string matching algorithm that works in linear time. When searching for pattern T in string S from a position p, and finding a mismatch $S[p+i] \neq T[i]$, instead of matching from the beginning of T in position $p+1$, we already know which other prefixes of T do match with the already matched portion of the pattern, and thus the pointer $p+i$ never goes back.

We do this by building an array that, similar to the Z-Algorithm but unlike it, has at position i the longest prefix of T that is also a suffix of $T[1 \dots i]$. Such a table can be easily built incrementally, and it can be seen that the total running cost of the algorithm is $\mathcal{O}(|S| + |T|)$[12, p.1002-1011].

UVa 11475 - Extend to Palindromes

Statement: Your task is, given an integer N, to make a palidrome (word that reads the same when you reverse it) of length at least N. Any palindrome will do. Easy, isn't it? That's what you thought before you passed it on to your inexperienced team-mate. When the contest is almost over, you find out that that problem still isn't solved. The problem with the code is that the strings generated are often not palindromic. There's not enough time to start again from scratch or to debug his messy code. Seeing that the situation is desperate, you decide to simply write some additional code that takes the output and adds just enough extra characters to it to make it a palindrome and hope for the best. Your solution should take as its input a string and produce the smallest palindrome that can be formed by adding zero or more characters at its end.

Solution: To make the result be the smallest possible, we need to use the largest suffix of the string (let it be S) that is already a palindrome. To find it, we build the KMP table for S reversed (let's call it S'), then search in S. The position in S' in which we are when finishing to traverse S is the length of such

largest palindrome suffix. After that, we just append the rest of S' to S to be left with the answer.

Listing 2: Solution for Extend to Palindromes

```
#include <iostream>
#include <string>
#include <vector>
#include <algorithm>
using namespace std;

typedef vector<int> vi;

int kmp(const string& s)
{
  int N = s.size();
  vi T(N+1);
  T[0] = -1;
  T[1] = 0;
  int x = 0;
  for (int i = 1; i < N; ++i)
  {
    while (x >= 0 && s[i] != s[x]) x = T[x];
    T[i+1] = ++x;
  }
  return T[N];
}

int main()
{
  for (string s; cin >> s;)
  {
    string t = s;
    int n = s.size();
    reverse(t.begin(), t.end());
    t += '#';
    t += s;
    int c = kmp(t);
    cout << s << t.substr(c, n-c) << endl;
  }
}
```

Codeforces 346 - Lucky Common Subsequence

Statement: You are given two strings s_1, s_2 and another string called *virus*($1 \leq |s_1|, |s_2|, |virus| \leq 100$). Your task is to find the longest common subsequence of s_1 and s_2, such that it doesn't contain *virus* as a substring.

Solution: We build the KMP table for *virus*, and then we run a dynamic programming solution. We have the subproblem $f(i, j, k)$ be the length of the longest common subsequence of $s_1[i \dots]$ and $s_2[j \dots]$ that does not match *virus* if the first k characters of *virus* are already matched (this is different from saying that it does not match $virus[k \dots]$ since there can be multiple prefixes of virus being a suffix of $virus[0 \dots k]$).

This is a typical problem of longest common subsequence (equivalent to the edit distance problem) but adding an additional dimension to our dynamic programming which is the status of the *virus* in the answer.

Listing 3: Solution for Lucky Common Subsequence

```
#include <iostream>
#include <string>
#include <vector>
#include <functional>
using namespace std;
```

```
typedef vector<int> vi;
typedef vector<vi> vvi;
typedef vector<vvi> vvvi;

vi kmp(const string& s)
{
  int N = s.size();
  vi T(N+1);
  T[0] = -1;
  T[1] = 0;
  int x = 0;
  for (int i = 1; i < N; ++i)
  {
    while (x >= 0 && s[i] != s[x]) x = T[x];
    T[i+1] = ++x;
  }
  return T;
}

int main() {
  string s1, s2, virus;
  cin >> s1 >> s2 >> virus;
  vi t = kmp(virus);
  vvvi dp(1+s1.size(), vvi(1+s2.size(), vi(t.size(), -1)));
  function<int(int, int, int)> f;
  function<int(int, int, int)> g;
  f = [&](int i, int j, int k)
  {
    int& res = dp[i][j][k];
    if (res != -1) return res;
    res = 0;
    if (i == s1.size() or j == s2.size()) return res;
    res = max(f(i, j+1, k), f(i+1, j, k));
    if (s1[i] == s2[j])
    {
      while (k >= 0 && s1[i] != virus[k]) k = t[k];
      if (k+1 < virus.size()) res = max(res,1+f(i+1, j+1, k+1));
    }
    return res;
  };
  g = [&](int i, int j, int k)
  {
    int res = f(i, j, k);
    if (res == 0) return 0;
    if (res == f(i, j+1, k)) return g(i, j+1, k);
    if (res == f(i+1, j, k)) return g(i+1, j, k);
    cout << s1[i];
    while (k >= 0 && s1[i] != virus[k]) k = t[k];
    return 1+g(i+1, j+1, k+1);
  };
  int l = g(0, 0, 0);
  if (!l) cout << "0";
  cout << endl;
}
```

Aho-Corasick

The Aho-Corasick algorithm is a matching algorithm that locates elements of a finite set of patterns within a text. It constructs a Deterministic Finite Automaton resembling a trie with some extra links using the patterns to match. Then it runs the automaton against the text.

Timus 1269 - Obscene Words Filter

Statement: There is a problem to check messages of web-board visitors for the obscene words. Your elder colleagues commit this problem to you. You are to write a program, which check if there is at least one obscene word from the given list in the given text as a substring.

Solution: It is a direct application of the Aho-Corasick algorithm for matching multiple patterns in a text. Due to strong memory restrictions in the problem, we split the set of patterns in several buckets and run the search for each of the buckets.

Listing 4: Solution for Obscene Words Filter

```
#include <iostream>
#include <sstream>
#include <string>
#include <vector>
#include <map>
#include <queue>
#include <cstdlib>
using namespace std;

struct AC
{
  struct node
  {
    int fail, match;
    int len;
    map<char, int> nxt;
    bool ids;
    node() {}
    node(int l) : fail(-1), match(-1), len(l), nxt(), ids(false) {}
  };
  vector<node> A;

  AC() : A(1, node(0)) {}

  void add(const string& s)
  {
    int u = 0, n = s.size();
    for (int i = 0; i < n; ++i)
    {
      char c = s[i];
      auto it = A[u].nxt.find(c);
      if (it == A[u].nxt.end())
      {
        it = A[u].nxt.emplace(c, A.size()).first;
        A.emplace_back(A[u].len+1);
      }
      u = it->second;
    }
    A[u].ids = true;
  }

  void aho_corasick()
  {
    queue<int> q; q.push(0);
    while (!q.empty())
    {
      int u = q.front(); q.pop();
      for (auto e : A[u].nxt)
      {
        char c = e.first; int v = e.second;
        if (A[v].fail != -1) continue;
        if (u == 0)
```

```
        {
          A[v].fail = A[v].match = 0;
        }
        else
        {
          int f = A[u].fail;
          while (f and !A[f].nxt.count(c)) f=A[f].fail;
          if (!A[f].nxt.count(c)) A[v].fail = 0;
          else A[v].fail = A[f].nxt[c];
          A[v].match = A[A[v].fail].ids ?
                       A[v].fail : A[A[v].fail].match;
        }
        q.push(v);
      }
    }
  }

  int match(int u)
  {
    if (!A[u].ids) u = A[u].match;
    while (u != -1)
    {
      if (A[u].ids) return A[u].len;
      u = A[u].match;
    }
    return -1;
  }

  int search(const string& s)
  {
    int n = s.size();
    int u = 0;
    int res = -1;
    for (int i = 0; i < n; ++i)
    {
      char c = s[i];
      while (u and !A[u].nxt.count(c)) u = A[u].fail;
      if (A[u].nxt.count(c)) u = A[u].nxt[c];
      int x = match(u);
      if (x != -1)
      {
        res = res == -1 ?
              i+1-x : min(res, i+1-x);
      }
    }
    return res;
  }
};

int main()
{
  string line;
  int n; cin >> n; getline(cin, line);
  vector<string> dict(n);
  for (int i = 0; i < n; ++i)
    getline(cin, dict[i]);

  int m; cin >> m; getline(cin, line);
  vector<string> text(m);
  for (int i = 0; i < m; ++i)
    getline(cin, text[i]);

  AC* ac = new AC();

  int rl = m-1, rc = -1;
  for (int i = 0; i < n; ++i)
```

```
  {
    ac->add(dict[i]);
    if (ac->A.size() > 30000)
    {
      ac->aho_corasick();
      for (int j = 0; j <= rl; ++j)
      {
        int x = ac->search(text[j]);
        if (x != -1 and (j < rl or rc == -1 or x < rc))
        {
          rl = j;
          rc = x;
        }
      }
      *ac = AC();
    }
  }
  ac->aho_corasick();
  for (int j = 0; j <= rl; ++j)
  {
    int x = ac->search(text[j]);
    if (x != -1 and (j < rl or rc == -1 or x < rc))
    {
      rl = j;
      rc = x;
    }
  }
  delete ac;
  if (rc != -1) cout << rl+1 << ' ' << rc+1 << endl;
  else cout << "Passed\n";
}
```

Suffix Tree

A suffix tree of a string S is a tree with characters in its edges, each suffix of S being a path from the root to one of the leafs. It is a structure similar to a trie, but paths of nodes with only one children are compressed as a single edge representing a substring. It can be built in $\mathcal{O}(|S|)$.

Codeforces 452 - Three strings

Statement: You are given three strings (s_1, s_2, s_3). For each integer $l (1 \leq l \leq \min |s1|, |s2|, |s3|)$ you need to find how many triples (i_1, i_2, i_3) exist such that three strings $s_k[i_k \ldots i_k + l - 1] (k = 1, 2, 3)$ are pairwise equal. Print all found numbers modulo $1000000007 (10^9 + 7)$.

Solution: We build the suffix tree for the three strings, and we store for each node how many times it appears in each of the three strings. Then we traverse the tree and compute the results.

The problem asks us for an answer for each length of substrings; to avoid doing operations for each character in a compressed edge, we store the results in differential form: if *ans* is the array where store the results, instead of adding x to each of $ans[t_0]$ to $ans[t_1]$, we add it to $ans[t_0]$ and substract it from $ans[t_1 + 1]$, and then we do a sum of the answers when outputting them.

Listing 5: Solution for Three strings

```
#include <iostream>
#include <vector>
#include <string>
#include <map>
```

```
#include <set>
using namespace std;

typedef long long ll;

typedef char ttype;

struct TR {
  int t0, t1;
  int node;
  TR() {}
  TR(int _t0, int _t1, int _node) : t0(_t0), t1(_t1), node(_node) {}
};
struct ND {
  int link;
  map<ttype, TR> edges;
  vector<int> dollars;
};
const int MAXN = 300050;
int N, M;
ttype T[MAXN + 17];
ND tree[2*MAXN + 17];

bool test_and_split(int s, int k, int p, ttype t, int& r) {
  r = s;
  if (p < k) return tree[s].edges.count(t) != 0;
  TR tr = tree[s].edges[T[k]];
  int sp = tr.node, kp = tr.t0, pp = tr.t1;
  if (t == T[kp + p - k + 1]) return true;
  r = M++;
  tree[s].edges[T[kp]] = TR(kp, kp + p - k, r);
  tree[r].edges[T[kp + p - k + 1]] = TR(kp + p - k + 1, pp, sp);
  return false;
}

void canonize(int s, int k, int p, int& ss, int& kk){
  if (p < k) { ss = s; kk = k; return; }
  TR tr = tree[s].edges[T[k]];
  int sp = tr.node, kp = tr.t0, pp = tr.t1;
  while (pp - kp <= p - k) {
    k += pp - kp + 1; s = sp;
    if (k <= p) {
      tr = tree[s].edges[T[k]];
      sp = tr.node; kp = tr.t0; pp = tr.t1;
    }
  }
  ss = s; kk = k;
}

void update(int& s, int& k, int i) {
  int oldr = 0, r;
  while (not test_and_split(s, k, i - 1, T[i], r)) {
    tree[r].edges[T[i]] = TR(i, MAXN, M++);
    if (oldr != 0) tree[oldr].link = r;
    oldr = r;
    canonize(tree[s].link, k, i - 1, s, k);
  }
  if (oldr != 0) tree[oldr].link = s;
}

int sts, stk;
void init() {
  for (int i = 0; i < M; ++i) {
    tree[i].link = 0; tree[i].edges.clear(); }
  M = 2; N = 0; tree[0].link = 1; sts = stk = 0;
}
```

```
void add(ttype c) {
  T[N] = c;
  tree[1].edges[T[N]] = TR(N, N, 0);
  update(sts, stk, N);
  canonize(sts, stk, N++, sts, stk);
}

void tallagespa() {
  vector<int> next(N + 1);
  next[N] = N;
  for (int i = N - 1; i >= 0; --i) {
    if (T[i] < 0) next[i] = i;
    else next[i] = next[i + 1];
  }
  for (int i = 0; i < M; ++i) {
    if (i == 1) continue;
    for (auto it = tree[i].edges.begin(); it != tree[i].edges.end(); ++it) {
      TR &tr = it->second;
      if (next[tr.t0] <= tr.t1) {
        tr.t1 = next[tr.t0];
        tree[tr.node].edges.clear();
      }
    }
  }
}

void compute_dollars() {
  vector<vector<int> > dollars(M, vector<int>(3, 0));
  vector<int> ord(1, 0);
  for (int i = 0; i < int(ord.size()); ++i) {
    int n = ord[i];
    for (auto it = tree[n].edges.begin(); it != tree[n].edges.end(); ++it) {
      TR tr = it->second;
      if (tree[tr.node].edges.size() == 0) {
        dollars[tr.node][-T[tr.t1]-1] = 1;
      }
      else ord.push_back(tr.node);
    }
  }
  for (int i = ord.size() - 1; i >= 0; --i) {
    int n = ord[i];
    for (auto& it : tree[n].edges) {
      TR tr = it.second;
      for (int i = 0; i < 3; ++i)
        dollars[n][i] += dollars[tr.node][i];
      dollars[tr.node].clear();
    }
    tree[n].dollars = dollars[n];
  }
}

const int MOD = 1000000007;

void add_product(int& r, int a, int b, int c)
{
  int p = ((ll)a*b)%MOD;
  p = ((ll)p*c)%MOD;
  r = ((ll)r+p)%MOD;
}

void count(vector<int>& dif, int n, int d, int maxd) {
  for (auto it = tree[n].edges.begin(); it != tree[n].edges.end(); ++it) {
    TR tr = it->second;
    if (tree[tr.node].edges.size() != 0) {
      add_product(dif[d],
                  tree[tr.node].dollars[0],
```

```
                    tree[tr.node].dollars[1],
                    tree[tr.node].dollars[2]);
      if (d+1+tr.t1-tr.t0 < maxd) {
        add_product(dif[d+tr.t1+1-tr.t0],
                    -tree[tr.node].dollars[0],
                    -tree[tr.node].dollars[1],
                    -tree[tr.node].dollars[2]);
        count(dif, tr.node, d+1+tr.t1-tr.t0, maxd);
      }
    }
  }
}

int main() {
  init();
  int maxl = 400000;
        for (int i = 1; i <= 3; ++i)
  {
    string s; cin >> s;
    for (char c : s) add(c);
    add(-i);
    maxl = min(maxl, (int)s.size());
  }
  tallagespa();
  compute_dollars();
  vector<int> dif(maxl, 0);
  count(dif, 0, 0, maxl);
  vector<int> res(maxl, 0);
  res[0] = dif[0];
  cout << res[0];
  for (int i = 1; i < maxl; ++i)
  {
    res[i] = res[i-1] + dif[i];
    if (res[i] < 0) res[i] += MOD;
    if (res[i] >= MOD) res[i] %= MOD;
    cout << ' ' << res[i];
  }
  cout << endl;
}
```

Eertree

The eertree or palindromic tree is a data structure that contains all distinct palindrome substrings of a string S, and it can be built in $\mathcal{O}(|S|)$ time and space.

It is built incrementally; initially, there are two nodes: the empty string of length 0, which will be the root to all palindromes of even length, and a "imaginary" string of length -1 for convenience, which is the root of the odd-lengthed palindromes.

Each edge is marked with a character: an edge from X to Y with character c means that the palindrome represented by Y is equal to the one represented by X adding c at each side. Every node has also a pointer called suffix link to the longest suffix of itself that is also a palindrome.

An interesting property that is not hard to see is that every letter added can add at most one new palindrome to the tree, and it will be the longest suffix of the string up to that point that is a palindrome. By keeping a pointer to such longest palindromic suffix, when adding a new character c we only need to check if cXc is a suffix for X in the suffix link path of the previous longest palindromic suffix.

Timus 1960 - Palindromes and Super Abilities

Statement: After solving seven problems on Timus Online Judge with a word "palindrome" in the problem name, Misha has got an unusual ability. Now, when he reads a word, he can mentally count the number of unique nonempty substrings of this word that are palindromes.

Dima wants to test Misha's new ability. He adds letters $s_1, \ldots, s_n$ to a word, letter by letter, and after every letter asks Misha, how many different nonempty palindromes current word contains as substrings. Which n numbers will Misha say, if he will never be wrong?

Solution: Building the eertree is a direct solution for this problem: after processing each letter, we print the current number of palindromes in the eertree, which is the number of nodes minus 2.

Listing 6: Solution for Palindromes and Super Abilities

```
#include <iostream>
#include <vector>
#include <string>
#include <cstring>
#include <cmath>
using namespace std;

typedef long long ll;

struct eertree
{
  struct node
  {
    node() : len(0), sufflink(1), num(0), cant(0)
    {
      memset(next, 0, sizeof(next));
    }
    int next[26], len, sufflink, num;
    ll cant;
  };
  string s;
  int len, num, suff;
  vector<node> tree;
  eertree() : num(2), suff(2) {}

  void initTree()
  {
    len = s.size();
    tree = vector<node>(len + 3);
    tree[1].len = -1; tree[1].sufflink = 1;
    tree[2].len = 0; tree[2].sufflink = 1;
  }

  bool addLetter(int pos)
  {
    int cur = suff, curlen = 0, let = s[pos]-'a';
    while (true)
    {
      curlen = tree[cur].len;
      if (pos - 1 - curlen >= 0
          and s[pos - 1 - curlen] == s[pos])
        break;
      cur = tree[cur].sufflink;
    }
    if (tree[cur].next[let])
    {
      suff = tree[cur].next[let];
      ++tree[suff].cant;
      return false;
```

```
    }
    num++;
    suff = num;
    tree[num].len = tree[cur].len + 2;
    tree[num].cant = 1;
    tree[cur].next[let] = num;
    if (tree[num].len == 1)
    {
      tree[num].sufflink = 2; tree[num].num = 1;
      return true;
    }
    while (true)
    {
      cur = tree[cur].sufflink;
      curlen = tree[cur].len;
      if (pos - 1 - curlen >= 0
          and s[pos - 1 - curlen] == s[pos])
      {
        tree[num].sufflink = tree[cur].next[let];
        break;
      }
    }
    tree[num].num = 1 + tree[tree[num].sufflink].num;
    return true;
  }
};

int main()
{
  ios::sync_with_stdio(false);
  eertree t;
  cin >> t.s;
  t.initTree();
  for (int i = 0; i < t.len; ++i)
  {
    t.addLetter(i);
    cout << (i?" ":"") << t.num-2;
  }
  cout << endl;
}
```

Geometry

Geometry is a common and interesting topic in competitive programming; most modern contests include at least one geometry problem. Sometimes, the solution to those problems can be easily derived without complex algorithmic knowledge, but usually it is not the case. Most common problems in computational geometry involve points in bidimensional space.

Codeforces 613 - Peter and Snow Blower

Statement: Peter got a new snow blower as a New Year present. Of course, Peter decided to try it immediately. After reading the instructions he realized that it does not work like regular snow blowing machines. In order to make it work, you need to tie it to some point that it does not cover, and then switch it on. As a result it will go along a circle around this point and will remove all the snow from its path.

Formally, we assume that Peter's machine is a polygon on a plane. Then, after the machine is switched on, it will make a circle around the point to which Peter tied it (this point lies strictly outside the polygon). That is, each of the points lying within or on the border of the polygon will move along the circular trajectory, with the center of the circle at the point to which Peter tied his machine.

Peter decided to tie his car to point P and now he is wondering what is the area of the region that will be cleared from snow. Help him.

Solution: We need to compute the minimum distance d and the maximum distance D from points in the polygon to P. After that, we can easily see that the area covered is the one of the circular ring centered in P with radiuses d and D, that is, $\pi(D^2 - d^2)$.

Since the distance to P is a convex function, the maximum will lay in one of the vertices of the polygon. It is easily computable by taking the maximum among all distances from P to the vertices. Nevertheless, the minimum may be in a point on a side that is not a vertex. It will not be in the inside since P is strictly outside of the polygon. We can compute the distance from P to each segment using basic algebra, and take the minimum of them.

This way, the problem is solved in $\mathcal{O}(n)$.

Listing 7: Solution for Peter and Snow Blower

```
#include <iostream>
#include <cmath>
#include <vector>
using namespace std;

typedef long long ll;
typedef long double ld;
constexpr ld pi =
3.14159265358979323846264338327950288419716939937510582097494459230781640628 6l;

ld dist_ps(ll px, ll py, ll qx, ll qy)
{
  ll vx = qx-px; ll vy = qy-py;
  if (px*vx+py*vy >= 0) return -1;
  if (qx*vx+qy*vy <= 0) return -1;
  ld t = -(px*vx + py*vy)/
      (ld)(vx*vx + vy*vy);
  return (px + t*vx)*(px + t*vx) +
         (py + t*vy)*(py + t*vy);
}
```

```
int main()
{
  ios::sync_with_stdio(false);
  cout.setf(ios::fixed);
  cout.precision(10);
  int n; cin >> n;
  vector<ll> x(n);
  vector<ll> y(n);

  {
    ll x0, y0;
    cin >> x0 >> y0;
    for (int i = 0; i < n; ++i)
    {
      cin >> x[i] >> y[i];
      x[i] -= x0; y[i] -= y0;
    }
  }

  ll RR = x[0]*x[0]+y[0]*y[0];
  ld rr = RR;
  ld dd = dist_ps(x[n-1], y[n-1], x[0], y[0]);
  if (dd > 0) rr = min(rr,dd);
  for (int i = 1; i < n; ++i)
  {
    ll a = x[i]*x[i]+y[i]*y[i];
    RR = max(RR, a);
    rr = min(rr, (ld)a);
    if((dd = dist_ps(x[i-1], y[i-1], x[i], y[i])) > 0)
      rr = min(rr, dd);
  }
  cout << pi*(RR-rr) << endl;
}
```

Convex hull

Often, it is of interest to find the convex hull of a finite set of points in the plane. One well-known algorithm for computing it is the Graham scan.

It is based on sorting the points by angle (e.g. counterclockwise) respect a given fixed point, and then iterating over them and discarding some of them when a "wrong" (e.g. clockwise) turn happens. It runs in $\mathcal{O}(n \log n)$ due to the sorting, but it can be made to run in linear time if the set of points is sorted in a convenient way.

It is described first in [14], although the more modern implementation can be found in [15, p. 289-291] and [12, p. 1030-1037] where there is also a proof of correctness.

UVa 811 - The Fortified Forest

Statement: There are n trees in certain points in 2D space, and you have been tasked to build a high fence around them. But the only material available to build the fence is the wood from the trees themselves. In other words, it's necessary to cut down some trees in order to build a fence around the remaining trees. You want to minimize the value of the trees that have to be cut.

For each case you are given n between 2 and 15, the locations (x_i, y_i) of the n trees, their values v_i and the length of fence l_i that can be built using the wood of each tree. Output which trees are to be cut

with minimum valueso that the fence is built successfully, and how much wood is leftover after building the fence.

Solution: The base of this problem is a convex hull problem: given a set of trees, we can know the minimum amount of fence required for surrounding it by computing the perimeter of the convex hull.

The problem is then to choose a subset of trees to cut, and the answer is simple: given the constraints of the problem ($n < 15$) it's affordable to check for each possible subset of trees if the perimeter of the convex hull of the remaining trees is lesser than or equal to the length of fence available from the cut trees. This can be done in $\mathcal{O}(2^n n \log n)$.

So for this problem, we compute the convex hull of every subset of points, then we choose the one such that the perimeter of the convex hull is lesser than or equal to the length available, and the sum of values of the trees cut is minimum. The leftover wood is then trivially computed as the difference between the length of the trees cut and the perimeter of the convex hull.

In my solution, when iterating over the subsets of cut trees, I computed the value of the subset and skipped if it was not lesser that the best already found. This optimization is probably not necessary but it did not add any complexity to the code.

Listing 8: Solution for The Fortified Forest

```
#include <iostream>
#include <cmath>
#include <vector>
#include <algorithm>
using namespace std;

typedef vector<int> vi;
typedef struct
{
  int x, y;
} point;
typedef vector<point> vp;

int det2(const point& a, const point&b)
{
  return a.x*b.y - a.y*b.x;
}

point operator-(const point& lhs, const point& rhs)
{
  return point{lhs.x - rhs.x, lhs.y - rhs.y};
}

int det3(const point&a, const point& b, const point& c)
{
  return det2(b-a, c-a);
}

int dist2(const point& a, const point& b)
{
  return (a.x-b.x)*(a.x-b.x) + (a.y-b.y)*(a.y-b.y);
}

double dist(const point& a, const point& b)
{
  return sqrt((double)dist2(a,b));
}

vi ch(vp& p)
```

```
{
  int n = p.size();
  vi r;
  if (n < 4)
  {
    for (int i = 0; i < n; ++i) r.push_back(i);
    return r;
  }
  nth_element(p.begin(), p.begin(), p.end(), [](const point& a, const point& b)
      { return a.x < b.x or (a.x == b.x and a.y < b.y); }
      );
  sort(p.begin()+1, p.end(), [&](const point& a, const point& b)
    { return det3(p[0], a, b) > 0; }
    );
  r.push_back(0);
  r.push_back(1);
  r.push_back(2);
  for (int i = 3; i < n; ++i)
  {
    while (r.size() > 1 and det3(p[r[r.size()-2]], p[r[r.size()-1]], p[i]) <= 0)
    {
      if (det3(p[r[r.size()-2]], p[r[r.size()-1]], p[i]) == 0
          and dist2(p[r[r.size()-2]], p[r[r.size()-1]]) > dist2(p[r[r.size()-2]], p[i])
         )
      {
        break;
      }
      r.pop_back();
    }
    r.push_back(i);
  }
  return r;
}

double chlen(const vp& p, int cut_mask)
{
  vp pp;
  for (int i = 0; i < (int)p.size(); ++i) if (not ((1<<i) & cut_mask)) pp.push_back(p[i]);
  vi ind = ch(pp);
  double len = 0;
  for (int i = 1; i < (int)ind.size(); ++i)
  {
    len += dist(pp[ind[i-1]], pp[ind[i]]);
  }
  len += dist(pp[ind.back()], pp[ind[0]]);
  return len;
}

int sumval(const vi& v, int mask)
{
  int r = 0;
  for (int b = 0; b < 15; ++b) if ((1<<b) & mask) r += v[b];
  return r;
}

int main()
{
  cout.setf(ios::fixed);
  cout.precision(2);
  for(int n = 0, t = 1; cin >> n, n; ++t)
  {
    vp p(n);
    vi v(n), l(n);
    int minval = 0;
    double extra = 0;
    for (int i = 0; i < n; ++i)
```

```
    {
      cin >> p[i].x >> p[i].y >> v[i] >> l[i];
      minval += v[i];
      extra += l[i];
    }
    int cutting = (1<<n) - 1;
    for (int mask = 1; mask < (1<<n); ++mask) {
      int newval = sumval(v, mask);
      if (newval < minval or
          (newval == minval and __builtin_popcount(mask) < __builtin_popcount(cutting))
         )
      {
        int len = sumval(l, mask);
        double needed = chlen(p, mask);
        if ((double)len >= needed)
        {
          minval = newval;
          extra = (double)len - needed;
          cutting = mask;
        }
      }
    }
    if (t > 1) cout << endl;
    cout << "Forest␣" << t << endl;
    cout << "Cut␣these␣trees:";
    for (int b = 0; b < n; ++b) if((1<<b) & cutting) cout << '␣' << b+1;
    cout << endl;
    cout << "Extra␣wood:␣" << extra << endl;
  }
}
```

Graphs

Graphs are a powerful tool, since they allow us to model a wide variety of real life problems. The most common operation in a graph is arguably calculating the shortest path, but there are many other kinds of problems such as the maximum flow problem or the minimum spanning tree of a connected weighted graph.

In this section, the set of vertices of a graph will often be referred to as V and the set of edges, as E.

Breadth-first search

Breadth-first search is a simple and useful way to traverse a graph; since it iterates over the vertices in order of increasing distance to the source of the search, it can be used to find the shortest path between two nodes in an unweighted graph. Its running time is $\mathcal{O}(|V| + |E|)$.

UVa 532 - Dungeon Master

Statement: You are trapped in a 3D dungeon and need to find the quickest way out! The dungeon is composed of unit cubes which may or may not be filled with rock. It takes one minute to move one unit north, south, east, west, up or down. You cannot move diagonally and the maze is surrounded by solid rock on all sides.

Is an escape possible? If yes, how long will it take?

Solution: If we think of the 3D maze as a graph, the solution is just one BFS to find the distance from the start to the exit. The complexity of the problem is then linear in the size of the map, since each vertex has at most 6 edges.

Listing 9: Solution for Dungeon Master

```
#include <iostream>
#include <vector>
#include <queue>
using namespace std;

typedef vector<int> vi;
typedef vector<vi> vvi;
typedef vector<vvi> vvvi;
typedef pair<int, int> ii;
typedef pair<int, ii> iii;
typedef queue<iii> qiii;

int dz[] = {1, -1, 0, 0, 0, 0};
int dx[] = {0, 0, 1, 0, -1, 0};
int dy[] = {0, 0, 0, 1, 0, -1};

int main()
{
  for (int L, R, C; cin >> L >> R >> C, L;)
  {
    vvvi d(L, vvi(R, vi(C, -1)));

    int sl, sr, sc, dl, dr, dc;

    for (int l = 0; l < L; ++l)
    {
      for (int r = 0; r < R; ++r)
```

```
      {
        for (int c = 0; c < C; ++c)
        {
          char x; cin >> x;
          if (x == 'S')
          {
            sl = l;
            sr = r;
            sc = c;
          }
          else if (x == 'E')
          {
            dl = l;
            dr = r;
            dc = c;
          }
          else if (x == '#')
          {
            d[l][r][c] = -2;
          }
        }
      }
    }

    int& res = d[dl][dr][dc];
    qiii q;
    d[sl][sr][sc] = 0;
    q.push(iii(sl, ii(sr, sc)));
    while (!q.empty() and res == -1)
    {
      int l = q.front().first;
      int r = q.front().second.first;
      int c = q.front().second.second;
      q.pop();
      for (int k = 0; k < 6; ++k)
      {
        int nl = l + dz[k];
        int nr = r + dx[k];
        int nc = c + dy[k];
        if (nl >= 0 and nr >= 0 and nc >= 0 and
            nl < L and nr < R and nc < C and d[nl][nr][nc] == -1)
        {
          d[nl][nr][nc] = d[l][r][c] + 1;
          q.push(iii(nl, ii(nr, nc)));
        }
      }
    }
    if (res < 0)
    {
      cout << "Trapped!\n";
    }
    else
    {
      cout << "Escaped in " << res << " minute(s).\n";
    }
  }
}
```

Dijkstra's algorithm

Dijkstra's algorithm[12, p.658-662] is the well-known efficient solution to find shortest paths with a single source on a weighted graph with positive costs. It is usually implemented using the `std::priority_queue`

from the C++ STL, and it runs in $\mathcal{O}(|E| \log |V|)$.

UVa 10801 - Lift Hopping

Statement: A skyscraper has no more than 100 floors, numbered from 0 to 99. It has $n(1 \leq n \leq 5)$ elevators which travel up and down at (possibly) different speeds. For each i in $1, 2, \ldots n$, elevator number i takes $T_i(1 \leq Ti \leq 100)$ seconds to travel between any two adjacent floors (going up or down). Elevators do not necessarily stop at every floor. What's worse, not every floor is necessarily accessible by an elevator.

You are on floor 0 and would like to get to floor k as quickly as possible. Assume that you do not need to wait to board the first elevator you step into and (for simplicity) the operation of switching an elevator on some floor always takes exactly a minute. Of course, both elevators have to stop at that floor. You are forbiden from using the staircase. No one else is in the elevator with you, so you don't have to stop if you don't want to. Calculate the minimum number of seconds required to get from floor 0 to floor k (passing floor k while inside an elevator that does not stop there does not count as "getting to floor k").

Solution: We can model this problem as a shortest path problem in a graph.

We have a vertex in the graph for each stop of each elevator. For each floor, if multiple elvators stop in that floor, we should add edges between every pair of stops of that floor, with cost 60 (a minute).

Besides, for each elevator i, every two stops of that elevator are also connected: if they're at floors a and b, the cost of the edge is $T_i|a - b|$. Nevertheless, it is enough to add edges for the contiguous stops in an elevator, since that way you can still reach any stop from any other with the same cost as before, going through the intermediate stops. This way, the number of edges in the graph is greatly reduced and so is the running time of the program.

Finally, we add an special source vertex to the graph which is connected to all stops in floor 0 with cost 0, and a destination vertex analogously in floor k. Now we look for the shortest path from the source vertex to the destination vertex with Dijkstra's algorithm, which will give us the answer to the problem.

Listing 10: Solution for Lift Hopping

```
#include <iostream>
#include <sstream>
#include <vector>
#include <unordered_map>
#include <queue>
using namespace std;

typedef pair<int, int> ii;
typedef vector<int> vi;

int v(int i, int f) { return 100*i+f; }

int main() {
  ios::sync_with_stdio(false);
  for (int n, k; cin >> n >> k;)
  {
    vi T(n); for (int& x : T) cin >> x;
    vector< unordered_map<int, int> > adj(100*n+2);
    vector<bool> E(100*n+2, false);
    int src = 100*n, dst = 100*n+1;

    string line; getline(cin, line);
    for (int i = 0; i < n; ++i)
    {
      getline(cin, line);
      istringstream sin(line);
```

```
      int prev = -1;
      for (int f; sin >> f;)
      {
        E[v(i,f)] = true;
        if (prev != -1)
        {
          int t = T[i]*(f-prev);
          adj[v(i,prev)][v(i,f)] = t;
          adj[v(i,f)][v(i,prev)] = t;
        }
        prev = f;
        if (f == 0)
        {
          adj[src][v(i,f)] = 0;
          adj[v(i,f)][src] = 0;
        }
        if (f == k)
        {
          adj[dst][v(i,f)] = 0;
          adj[v(i,f)][dst] = 0;
        }
        for (int j = 0; j < i; ++j) if (E[v(j,f)])
        {
          adj[v(i,f)][v(j,f)] = 60;
          adj[v(j,f)][v(i,f)] = 60;
        }
      }
    }
    vi d(100*n+2, -1);
    priority_queue<ii> q;

    q.push(ii(0, src));
    d[src] = 0;
    while(!q.empty())
    {
      int x = q.top().second;
      int dd = -q.top().first;
      q.pop();
      if (dd != d[x]) continue;
      if (x == dst) break;
      for (const auto& it : adj[x])
      {
        int y = it.first;
        int nd = dd+it.second;
        if (d[y] == -1 or d[y] > nd)
        {
          d[y] = nd;
          q.push(ii(-nd, y));
        }
      }
    }
    if (d[dst] == -1) cout << "IMPOSSIBLE\n";
    else cout << d[dst] << endl;
  }
}
```

Floyd-Warshall algorithm

Floyd-Warshall algorithm[12, p.693-700] is a dynamic programming algorithm to compute the distances between all pairs of vertices in a weighted graph. It computes the shortest path incrementally, at each step using only the first k vertices as intermediates in the path. The incremental computation can be described

with the formula

$$f_k(x, y) = \min(f_{k-1}(x, y), f_{k-1}(x, v_k) + f_{k-1}(v_k, y)).$$

This algorithm has a great advantadge in that it is very fast to implement, making it really useful in a lot of situations where the graph is not very big and we don't need a more efficient algorithm.

Codeforces 295 - Greg and Graph

Statement: Graph has a weighted directed graph, consisting of n vertices. In this graph any pair of distinct vertices has an edge between them in both directions. Greg loves playing with the graph and now he has invented a new game:

1. The game consists of n steps.
2. On the i-th step Greg removes vertex number x_i from the graph. As Greg removes a vertex, he also removes all the edges that go in and out of this vertex.
3. Before executing each step, Greg wants to know the sum of lengths of the shortest paths between all pairs of the remaining vertices. The shortest path can go through any remaining vertex. In other words, if we assume that $d(i, v, u)$ is the shortest path between vertices v and u in the graph that formed before deleting vertex x_i, then Greg wants to know the value of the following sum: $\sum_{v,u,v\neq u} d(i, v, u)$.

Help Greg, print the value of the required sum before each step.

Solution: We can modify the Floyd-Warshall algorithm so that it solves this problem: by choosing to update using the intermediates in reverse order of when will they be removed, we can use the partial results of the Floyd-Warshall algorithm to know the distances without the vertices that will be deleted before, and build the answer in reverse order.

Listing 11: Solution for Greg and Graph

```
#include <iostream>
#include <vector>
using namespace std;

typedef long long ll;
typedef vector<ll> vi;
typedef vector<vi> vvi;

int main()
{
  int n; cin >> n;

  vvi fw(n, vi(n)), s(n, vi(n));
  for (vi& v : fw) for (ll& x : v) cin >> x;

  vi order(n), rorder(n);
  for (int i = 0; i < n; ++i)
  {
    cin >> order[i]; --order[i];
    rorder[order[i]] = i;
  }

  vi result(n, 0);
  for (int k = 1; k <= n; ++k)
  {
```

```
    int p = order[n-k];
    for (int i = 0; i < n; ++i)
    {
      for (int j = 0; j < n; ++j)
      {
        s[i][j] = min(fw[i][j], fw[i][p] + fw[p][j]);
        if (min(rorder[i],rorder[j]) >= n-k) result[n-k] += s[i][j];
      }
    }
    swap(fw, s);
  }

  for (int k = 0; k < n; ++k)
    cout << (k?" ":"") << result[k];
  cout << endl;
}
```

Strongly Connected Components and 2-SAT

A directed graph is said to be *strongly connected* if, for every pair of vertices $u, v \in V$, v is reachable from u and u is reachable from v. The strongly connected components of a directed graph form a partition into maximal subgraphs that are strongly connected.

A directed graph can be partitioned into its strongly connected components in $\mathcal{O}(|V|+|E|)$ complexity using Kosaraju's algorithm[12, p. 615-620][13, p.98-100], which consits only in running a depth-first search and using some information from that while performing a depth-first search in the reversed graph.

An interesting application is 2-satisfiability, the special case of the Boolean satisfiability problem where each constraint has only 2 variables. It can be reduced to a strongly connected components problem by building the implication graph. The graph has a vertex for each literal and every clause $a \vee b$ in the formula is translated to an edge $(\neg a, b)$ and an edge $(\neg b, a)$ in the graph, which represent the implications $(\neg a \Rightarrow b) \equiv (\neg b \Rightarrow a)$ equivalent to $(a \vee b)$.

Then, if there is a variable x_i such that x_i and $\neg x_i$ are in the same strongly connected component of the implication graph, it would mean that $x_i \Rightarrow \neg x_i$ and $\neg x_i \Rightarrow x_i$ and thus the given 2-SAT formula is unsatisfiable. I can also be seen that the formula is satisfiable if such variable does not exist. Therefore, to solve the 2-SAT problem, we build the implication graph find its strongly connected components, then check for each variable if its negated is in the same component as it.

With Kosaraju's algorithm, it is also easy to find a feasible assignment of values for the variables, since you can find the strongly connected components in an order such that C_j is not reachable from C_i if $i < j$.

UVa 10319 - Manhattan

Statement: You are the mayor of a city with severe traffic problems. To deal with the situation, you have decided to make a new plan for the street grid. As it is impossible to make the streets wider, your approach is to make them one-way (only traffic in one direction is allowed on a street), thus creating a more efficient flow of traffic.

The streets in the city form an orthogonal grid – like on Manhattan avenues run in north-southdirection, while streets run in east-west-direction. Your mission is to make all the streets and avenues one-way, i.e. fix the direction in which traffic is allowed, while maintaining a short driving distance between some ordered pairs of locations. More specifically, a route in the city is defined by two streetavenue crossings, the start and goal location. On a one-way street grid, a route has a legal path if it is possible to drive from the start

location to the goal location along the path passing streets and avenues in their prescribed direction only. A route does not define a specific path between the two locations – there may be many possible paths for each route. A legal path in a one-way street grid is considered simple if it requires at most one turn, i.e. a maximum of one street and one avenue need to be used for the path.

When traveling by car from one location to another, a simple path will be preferred over a nonsimple one, since it is faster. However, as each street in the grid is one-way, there may always be routes for which no simple path exists. On your desk lies a list of important routes which you want to have simple paths after the re-design of the street grid.

Your task is to write a program that determines if it is possible to fix the directions of the one-way streets and avenues in such a way that each route in the list has at least one simple path.

Solution: We will turn the problem into a 2-SAT problem. We have a Boolean variable for each street and for each avenue representing the direction of it, and each route that needs to have a simple path translates into at most four 2-SAT clauses. For each route to have a simple path, we can only use the streets and avenues of origin and destiny of that route.

Let s_o and s_d be the literals that represent that the streets of origin and destiny respectively go in the direction favorable to the route, and a_o and a_d analogously for the avenues. Then the constraints for the route are $(s_o \vee s_d) \wedge (a_o \vee a_d) \wedge (s_o \vee a_o) \wedge (s_d \vee a_d)$. In the case the origin and destiny are already equal in one of the coordinates, we don't need the constraints that imply movement in that dimension, so we are left with just one, for example $(s_o \vee s_d)$, where s_o and s_d are actually the same literal.

Thus, we build the implication graph for the clauses described, and run Kosaraju's algorithm to find the strongly connected components and then check the satisfiability of the formula.

Listing 12: Solution for Manhattan

```
#include <iostream>
#include <vector>
#include <functional>
using namespace std;
using vi = vector<int>;
using vvi = vector<vi>;

vi scc(int n, const vvi& g, const vvi& gr)
{
  vi r(n, 0);
  vi post(n);
  int p = n;

  std::function<void(int)> dfs;
  dfs = [&](int x) -> void
  {
    r[x] = -1;
    for (int y : gr[x]) if (!r[y]) dfs(y);
    post[--p] = x;
  };

  for (int x = 0; x < n; ++x)
    if (!r[x])
      dfs(x);

  dfs = [&](int x) -> void
  {
    r[x] = p;
    for (int y : g[x]) if (r[y] == -1) dfs(y);
  };
```

```
  for (int i = 0; i < n; ++i)
  {
    if (r[post[i]] == -1)
    {
      dfs(post[i]);
      ++p;
    }
  }

  return r;
}

int main()
{
  int n; cin >> n;
  while (n--)
  {
    int S, A, m;
    cin >> S >> A >> m;

    int N = S + A;

    vvi imp(2*N);

    for (int i = 0; i < m; ++i)
    {
      int s1, a1, s2, a2;
      cin >> s1 >> a1 >> s2 >> a2;
      --s1; --a1; --s2; --a2;

      bool sp = a2 > a1;
      bool ap = s2 > s1;

      if (s1 == s2 and a1 == a2);
      else if (s1 == s2)
      {
        imp[sp*N + s1].push_back((!sp)*N + s1);
      }
      else if (a1 == a2)
      {
        imp[ap*N + S+a1].push_back((!ap)*N + S+a1);
      }
      else
      {
        imp[sp*N + s1].push_back((!sp)*N + s2);
        imp[sp*N + s2].push_back((!sp)*N + s1);

        imp[sp*N + s1].push_back((!ap)*N + S+a1);
        imp[ap*N + S+a1].push_back((!sp)*N + s1);

        imp[ap*N + S+a2].push_back((!sp)*N + s2);
        imp[sp*N + s2].push_back((!ap)*N + S+a2);

        imp[ap*N + S+a2].push_back((!ap)*N + S+a1);
        imp[ap*N + S+a1].push_back((!ap)*N + S+a2);
      }
    }

    vvi rev(2*N);
    for (int x = 0; x < 2*N; ++x)
      for (int y : imp[x])
        rev[y].push_back(x);

    vi c = scc(2*N, imp, rev);

    bool ok = true;
```

```
    for (int x = 0; x < N and ok; ++x)
      if (c[x] == c[N+x])
100     ok = false;

    cout << (ok ? "Yes\n" : "No\n");
  }
}
```

Maximum flow problem

The maxflow problem is, in a graph with capacities in the edges, to find an assignment of flow to the edges of the graph such that the flow from the vertex labeled as *source* to the vertex labeled as *sink* is maximum.

The minimum cut problem is, in a graph with weights in the edges, to find a set of edges that, if taken out, split two certain vertices into different connected components, such that the sum of the weights of the edges in the set is minimum.

It can be seen that the maximum amount flow is equal to the minimum weight necessary to split the source and the sink, if using the capacities as wights.

One common solution of the maxflow problem is the Edmonds-Karp algorithm, which is a particular implementation of the Ford-Fulkerson method that runs in $\mathcal{O}(VE^2)$ time. It consists of finding a path from the source to the sink by using breadth-first search, adding flow along that path and modifying the effective capacity of the edges used: subtracting to the capacity in the direction used but adding to it in the opposite direction. Since in the pathfinding we ignore edges with effective capacity 0, we repeat this process until there is no path left, and then we will have found the maximum flow possible.

UVa 10480 - Sabotage

Statement: The regime of a small but wealthy dictatorship has been abruptly overthrown by an unexpected rebellion. Because of the enormous disturbances this is causing in world economy, an imperialist military super power has decided to invade the country and reinstall the old regime.

For this operation to be successful, communication between the capital and the largest city must be completely cut. This is a difficult task, since all cities in the country are connected by a computer network using the Internet Protocol, which allows messages to take any path through the network. Because of this, the network must be completely split in two parts, with the capital in one part and the largest city in the other, and with no connections between the parts.

There are large differences in the costs of sabotaging different connections, since some are much more easy to get to than others.

Write a program that, given a network specification and the costs of sabotaging each connection, determines which connections to cut in order to separate the capital and the largest city to the lowest possible cost.

Solution: This is a typical minimum cut problem, and it asks us to explicit the set edges that form the cut. To obtain it, we run a search in the residual grap after running the Edmonds-Karp algorithm. All vertices that are reachable from the source vertex will form one side of the cut and unreachable vertices will form the other one. Now we just have to output the edges that connect vertices from the two sets.

Listing 13: Solution for Sabotage

```
#include <iostream>
#include <vector>
#include <queue>
using namespace std;

int main()
{
  for (int n, m; cin >> n >> m, n;)
  {
    vector< vector<bool> > adj(n, vector<bool>(n, false));
    vector< vector<int> > cap(n, vector<int>(n, 0));
    while (m--)
    {
      int a, b, c; cin >> a >> b >> c;
      --a; --b;
      cap[a][b] = cap[b][a] = c;
      adj[a][b] = adj[b][a] = true;
    }
    while (true)
    {
      vector<int> p(n, -1);
      vector<int> f(n, -1);

      queue<int> q;
      q.push(0);
      f[0] = 1000000000;
      p[0] = 0;
      while (!q.empty())
      {
        int x = q.front(); q.pop();
        for (int y = 0; y < n; ++y) if(p[y] == -1 and cap[x][y])
        {
          p[y] = x;
          f[y] = min(f[x], cap[x][y]);
          q.push(y);
        }
      }
      if (p[1] == -1)
      {
        for (int i = 0; i < n; ++i)
          for (int j = 0; j < n; ++j)
            if (p[i] != -1 and p[j] == -1 and adj[i][j])
              cout << i+1 << ' ' << j+1 << endl;
        cout << endl;
        break;
      }
      int flow = f[1];
      int x = 1;
      while(x)
      {
        cap[p[x]][x] -= flow;
        cap[x][p[x]] += flow;
        x = p[x];
      }
    }
  }
}
```

Square Root Decomposition

There is a number of problems over arrays in which one can come up with a solution that involves dividing the array in blocks over which we do some pre-computations to be able to solve queries efficiently later, and the optimal size for those blocks happens to be the square root of the size. It will better explained with an example.

SPOJ - Race Against Time

Statement: As another one of their crazy antics, the $N(1 \leq N \leq 10^5)$ cows want Farmer John to race against the clock to answer some of their pressing questions.

The cows are lined up in a row from 1 to N, and each one is holding a sign representing a number, $A_i(1 \leq Ai \leq 10^9)$. The cows need FJ to perform $Q(1 \leq Q \leq 50,000)$ operations, which can be either of the following:

- Modify cow i's number to $X(1 \leq X \leq 10^9)$.
- Count how many cows in the range $[P, Q](1 \leq P \leq Q \leq N)$ have $A_i \leq X(0 \leq X \leq 10^9)$.

Solution: Let us split the array in blocks $P_1, P_2, \ldots, P_m$ of size S. We will have a data structure for each of these blocks that allows us to solve the second type of queries efficiently for the whole block. This can be achieved in many ways, in our solution, we'll have a copy of the block with the contents sorted. This will have an initial cost of $\mathcal{O}(N \log S)$.

This way, we can solve efficiently both types of queries:

- When modifying a number in the original array, we need to look for it, modify it and re-position it in the sorted copy. This can all be done in $\mathcal{O}(S)$.
- When querying the count of numbers lesser or equal than X in a range $[P, Q]$, there are two steps. For each of the blocks which are completely contained in $[P, Q]$, we can count efficiently by binary searching X in the sorted copy; this will take $\mathcal{O}(\log S)$ for each block, and there are at most $\mathcal{O}(N/S)$ blocks. For the blocks in the extremes (the ones containing P and Q), we check the elements individually, there are $\mathcal{O}(S)$ elements to check like this.

This way the overall complexity of the solution is $\mathcal{O}(N \log S + Q(S + (N/S) \log S))$. Since we have both S and N/S terms, it is a good approximation to take $S = \sqrt{N}$ to minimize the number of operations needed. Thus the problem is solved in $\mathcal{O}((N + Q\sqrt{N}) \log N)$ time.

Listing 14: Solution for Race Against Time

```
#include <iostream>
#include <cmath>
#include <vector>
#include <algorithm>
using namespace std;

typedef vector<int> vi;

struct Data {
  int N;
```

```
  int S;
  vi A;
  vi B;

  Data(vi& a) : N(a.size()), S(sqrt(a.size())), A(a), B(move(a))
  {
    int p;
    for (p = 0; (p+1)*S < N; ++p)
      sort(B.begin() + (p*S), B.begin() + ((p+1)*S));
    sort(B.begin() + (p*S), B.end());
  }

  void replace(int old, int X, int part)
  {
    int l = part*S, r = min(N, (part+1)*S);
    int k = l; while (B[k] != old) ++k;
    B[k] = X;
    sort(B.begin()+l, B.begin()+r);
  }

  void modify(int I, int X)
  {
    int p = I/S;
    int old = A[I];
    A[I] = X;
    replace(old, X, p);
  }

  int num(int X, int part)
  {
    int l = part*S, r = min(N, (part+1)*S);
    return distance(B.begin()+l, upper_bound(B.begin()+l, B.begin()+r, X));
  }

  int query(int L, int R, int X)
  {
    int pl = L/S;
    int pr = R/S;
    int res = 0;
    for (int i = L; i < min(R, (pl+1)*S); ++i) if (A[i] <= X) ++res;
    if (pl == pr) return res;
    for (int p = pl+1; p < pr; ++p) res += num(X, p);
    for (int i = pr*S; i < R; ++i) if (A[i] <= X) ++res;
    return res;
  }
};

int main()
{
  ios::sync_with_stdio(false);
  int N, Q;
  cin >> N >> Q;
  vi v(N);
  for (int& x : v) cin >> x;
  Data d(v);
  for (int q = 0; q < Q; ++q)
  {
    char c; cin >> c;
    if (c == 'M')
    {
      int I, X;
      cin >> I >> X;
      d.modify(I-1, X);
    }
    else
    {
```

```
        int P, Q, X;
        cin >> P >> Q >> X;
        cout << d.query(P-1, Q, X) << endl;
80    }
  }
}
```

Mo's Algorithm

Mo's algorithm is a method to solve efficiently problems with range queries. The problems must be such that all queries can be known before starting, and the solution of a query can be constructed incrementally by adding or removing elements individually. It is most useful when the number of queries is close to the size of the data.

Let's say that the data has size n and we split it in buckets of size S, so that the i-th bucket represents the range $[iS, (i+1)S)$. This way there are $\lceil n/S \rceil$ buckets.

We sort the queries before answering them. If each query asks for a range $[l_i, r_i]$, we sort them first by index of the bucket in which l_i is, then by r_i.

While procressing the queries, we keep at all times two pointers, l and r, as well as the current answer to the query for the range $[l, r]$. We set them initially to $[0, 0]$; then, for each query, we move r and l one position at a time and recalculating the answer for range $[l, r]$ until $[l, r] = [l_i, r_i]$. Once $[l, r] = [l_i, r_i]$, we have computed the answer for that query. We repeat this process for every query, in the order specified before.

Let's say there are q queries.

Because of the way the queries are sorted, the pointer r only moves to the right within each bucket, and it only moves to the left one time when starting a new bucket of queries. Thus the right pointer moves a position $\mathcal{O}(n(n/S))$ times.

For the left pointer l, it will move at most $\mathcal{O}(S)$ positions between queries in the same bucket, since the l_i positions are in the same bucket. And the total sum of steps between queries of different buckets is $\mathcal{O}(n)$ since the queries are sorted by bucket increasingly.

Overall, the number of additions and deletions is $\mathcal{O}(n(n/S) + qS)$. By picking again $S = \sqrt{n}$, we get a complexity of $\mathcal{O}((n+q)\sqrt{n})$, which is best when $q \sim n$.

Codeforces 220 - Little Elephant and Array

Statement: The Little Elephant loves playing with arrays. He has array a, consisting of n positive integers, indexed from 1 to n. Let's denote the number with index i as a_i.

Additionally the Little Elephant has m queries to the array, each query is characterised by a pair of integers l_j and $r_j (1 \leq l_j \leq r_j \leq n)$. For each query l_j, r_j the Little Elephant has to count, how many numbers x exist, such that number x occurs exactly x times among numbers a_{l_j}, a_{l_j+1}, ... , a_{r_j}.

Help the Little Elephant to count the answers to all queries.

Solution: By application of the algorithm explained above:

We mantain a data structure with the frequencies f_x of numbers inside the range, and the count of numbers satisfying the property. When inserting or removing an element, we increment or decrement the

frequency and check if $x = f_x$ before and after modifying f_x to modify the count accordingly.

Complexity is $\mathcal{O}(m \log m + (n+m)\sqrt{n})$.

Listing 15: Solution for Little Elephant and Array

```
#include <iostream>
#include <cmath>
#include <vector>
#include <unordered_map>
#include <algorithm>
using namespace std;

typedef long long ll;
typedef vector<ll> vi;

struct DS
{
  vi a;
  ll res;
  unordered_map<int, int> K;
  int l, r;

  DS(vi&& v) : a(v), res(0), K(), l(0), r(0) {}

  void ins(int x)
  {
    if (!K.count(x)) K[x] = 0;
    if (x == K[x]) --res;
    ++K[x];
    if (x == K[x]) ++res;
  }

  void del(int x)
  {
    if (!K.count(x)) K[x] = 0;
    if (x == K[x]) --res;
    --K[x];
    if (x == K[x]) ++res;
  }

  void move(int l1, int l2)
  {
    if (l1 < l2)
      for (int x = l1; x < l2; ++x)
        del(a[x-1]);
    else
      for (int x = l2; x < l1; ++x)
        ins(a[x-1]);
  }

  void update(int nl, int nr)
  {
    move(l, nl);
    l = nl;
    move(nr, r);
    r = nr;
  }
};

struct Query
{
  int id;
  int l, r;
  ll res;
};
```

```
int main()
{
  ios::sync_with_stdio(false);
  int n, t, r;
  cin >> n >> t;
  r = sqrt(n);
  vi a(n);
  for (ll& x : a) cin >> x;
  DS ds(move(a));
  vector<Query> q(t);
  for (int i = 0; i < t; ++i) {
    q[i].id = i;
    cin >> q[i].l >> q[i].r;
  }
  sort(q.begin(), q.end(), [&](const Query& A, const Query& B)
      {
        if (A.l / r != B.l / r) return A.l / r < B.l / r;
        return A.r < B.r;
      });
  for (Query& x : q)
  {
    ds.update(x.l, x.r + 1);
    x.res = ds.res;
  }
  sort(q.begin(), q.end(), [&](const Query& A, const Query& B)
      { return A.id < B.id; });
  for (const Query& x : q) cout << x.res << endl;
}
```

Conclusions

This project covered many topics that can be used to improve at competitive programming. There are many more topics that could have been included, such as combinatorics and number theory, and more graph algorithms, such as Kruskal's or Bellman-Ford's, that are still present nowadays in programming contests but were not covered because of time constraints and desire for conciseness. The author hopes that the document assists to increase the interest of the reader towards competitive programming.

The problems in this document were solved in the same way as many others in the trainings at UPC for the ACM ICPC. This method has been working well for the UPC teams for years, since UPC has had many successes at ACM ICPC competitions. The author of this project himself has taken part in the ACM ICPC Southwestern European Regional Contest twice with the UPC teams, achieving a bronze medal and a silver medal in 2014 and 2015 respectively. The author has also achieved a world top 1000 position in the Google Code Jam multiple times.

For further reading, I suggest [15] for a practical approach on many competitive programming topics not covered in this document, and [13] for a more analytical approach on some of the basic algorithmic concepts that are used in this project and not only in the world of competitive programming but also outside of it.

Bibliography

References

[1] The acm-icpc international collegiate programming contest. https://icpc.baylor.edu/. Accessed: 2018-01-07.

[2] Acm-icpc live archive. https://icpcarchive.ecs.baylor.edu. Accessed: 2018-01-07.

[3] Codechef. http://codechef.com/. Accessed: 2018-01-07.

[4] Codeforces. http://codeforces.com. Accessed: 2018-01-07.

[5] Competitive programming - topcoder. https://www.topcoder.com/community/competitive-programming/. Accessed: 2018-01-07.

[6] Facebook hacker cup. https://www.facebook.com/hackercup/. Accessed: 2018-01-07.

[7] Google code jam. https://code.google.com/codejam/. Accessed: 2018-01-07.

[8] Sphere online judge (spoj). http://www.spoj.com/. Accessed: 2018-01-07.

[9] Timus online judge. http://acm.timus.ru/. Accessed: 2018-01-07.

[10] Uva online judge. http://uva.onlinejudge.org/. Accessed: 2018-01-07.

[11] Yandex algorithm contest. https://algorithm.contest.yandex.com/. Accessed: 2018-01-07.

[12] Thomas H. Cormen, Clifford Stein, Ronald L. Rivest, and Charles E. Leiserson. *Introduction to Algorithms, Third Edition*. The MIT Press, 3rd edition, 2009.

[13] Sanjoy Dasgupta, Christos H. Papadimitriou, and Umesh Vazirani. *Algorithms*. McGraw-Hill, Inc., New York, NY, USA, 1 edition, 2008.

[14] R.L. Graham. An efficient algorith for determining the convex hull of a finite planar set. *Information Processing Letters*, 1(4):132 – 133, 1972.

[15] S. Halim and F. Halim. *Competitive Programming 3: The New Lower Bound of Programming Contests*. Number v. 3. Lulu.com, 2013.