Министерство образования и науки Российской Федерации

Государственное образовательное учреждение
высшего профессионального образования
«Алтайский государственный технический университет
им. И.И.Ползунова»

НАУКА И МОЛОДЕЖЬ – 2007

IV Всероссийская научно-техническая конференция студентов, аспирантов и молодых ученых

СЕКЦИЯ

ИНФОРМАЦИОННЫЕ И ОБРАЗОВАТЕЛЬНЫЕ ТЕХНОЛОГИИ

подсекция

СТРУКТУРЫ И АЛГОРИТМЫ ОБРАБОТКИ ДАННЫХ

Барнаул – 2007

ББК 784.584(2 Рос 537)638.1

IV Всероссийская научно-техническая конференция студентов, аспирантов и молодых ученых "Наука и молодежь – 2007". Секция «Информационные и образовательные технологии». Подсекция «Структуры и алгоритмы обработки данных». / Алт. гос. техн. ун-т им. И.И.Ползунова. – Барнаул: изд-во АлтГТУ, 2007. – 31 с.

В сборнике представлены работы научно-технической конференции студентов, аспирантов и молодых ученых, проходившей в апреле 2007 г.

СОДЕРЖАНИЕ

СУФФИКСНЫЕ АВТОМАТЫ В РЕШЕНИИ ЗАДАЧ ОБРАБОТКИ ТЕКСТОВ

Акиньшин А.А. – студент гр. ПОВТ– 62

Нередко в олимпиадном программировании встречаются задачи, решение которых заключает в себе стандартные алгоритмы, приемы и методы. Знание и углубленное изучение общеизвестных методов решения определенных задач является необходимым для профессионального программиста. Рассмотрим, например, задачу J из олимпиады "университеты Алтая – 2007" (http://neerc.secna.ru/ALTAI_U/2007).

Условие задачи:

Однажды Лосяш нашел древний манускрипт, содержащий карту пути к древней пещере, в которой лежали сокровища далёких предков Смешариков. Всем известно, что древние Смешарики были высокоинтеллектуальной народностью, поэтому дверь в пещеру была защищена паролем. В древнем манускрипте содержалась информация о том, как подобрать пароль. У древних Смешариков было магическое заклинание, которое они использовали для вызова злых духов. На двери в пещеру была выбита куча точечек, которые были соединены стрелочками. Над каждой стрелочкой была написана строчная буковка латинского алфавита. Древнее пророчество гласило, что пароль можно получить, пройдя по стрелочкам от некоторой конкретной точечки (волшебной) до какой-то другой. Более того, Лосяш выяснил, что пароль содержится в магическом заклинании для вызова злых духов как подстрока ровно k раз. Помогите Лосяшу найти пароль и открыть миру сокровища древней народности Смешариков. Если вариантов пароля, отвечающих данным условиям, несколько, то сгодится любой. Также учтите, что манускрипт Лосяшу могли подбросить враги. В этом случае пароля не существует.

Входные данные:

На первой строке находится заклинание. Длина магического заклинания $L \le 105$. *На второй строчке находятся числа N и M* ($N, M \le 105$) *– число точечек и число стрелочек. В следующих M строчках содержится информация о стрелочках: номер стартовой точечки, номер конечной точечки и строчная буковка латинского алфавита. На последней строчке содержится число k* ($1 \le k \le L$). *Внимательно анализируя картинку на двери в пещеру, Лосяш заметил, что количество различных путей из волшебной точечки в остальные не превышает* 105. *Зато Лосяш обрадовался, что все переходы детерминированы, то есть из каждой по любому символу существует не более одного пути.*

Начальной волшебной точкой является точка с номером 1. *Все точки нумеруются с единицы.*

Выходные данные:

Если пароли существуют, то Вы должны вывести в лексикографическом порядке все те из них, которые удовлетворяют следующим условиям:

пароли выводятся по одному в строке;

нельзя вывести пароль X , если в списке выведенных паролей есть другой пароль, для которого X является префиксом;

Если паролей не существует, то выходной файл остается пустым. Гарантируется, что длина выходного файла не превышает 120000 *символов.*

Решение задачи

Для решения исследуемой задачи необходимо знать много алгоритмов и уметь пользоваться развитыми структурами данных: суффиксные автоматы, сортировки, способы эффективного представления графов.

Рассмотрим сначала такое понятие, как суффиксный автомат. Пусть нам дана некоторая строка S длины N. Суффиксный автомат – это ориентированный граф с определёнными характеристиками:

- в этом графе одна из вершин называется начальной, а несколько других называются

конечными;

- на каждом ребре написан символ алфавита, который использовался для создания рассматриваемой строки;
- если мы рассмотрим произвольный путь от начальной вершины до какой-нибудь из конечных, то строка, образованная буквами на рёбрах этого пути, должна являться суффиксом строки *S*.

Очевидно, что любой подстроке строки *S* соответствует путь из начальной вершины до какой-то другой. Действительно, рассмотрим дополнение этой подстроки до суффикса. По определению автомата этот суффикс содержится в нашем графе как путь от начальной вершины до какой-нибудь из конечных. Заметим, что рассматриваемая подстрока является префиксом данного суффикса, следовательно, мы можем получить её, если взять начало вышеуказанного пути. Очевидно, что для каждой подстроки можно достаточно быстро подсчитать количество вхождений её в исходную строку *S*. Для этого нужно сделать топологическую сортировку данного графа, а затем пройти от последней вершины к первой и использовать метод динамического программирования: зная, количество вхождений более больших подстрок, мы можем подсчитать количество вхождений данной строки, как сумму вышеуказанных чисел. Полученное число назовём ключом данной вершины.

Теперь, когда мы знаем о том, что такое суффиксный автомат, обратимся к условию нашей задачи. Нам дана строка и граф, который по своей структуре напоминает суффиксный автомат. Требуется найти все такие подстроки, встречающиеся фиксированное количество раз, которые можно получить пройдя по данному графу определённым образом. Построим суффиксный автомат по данной строке. А теперь с помощью поиска в глубину будем синхронно обходить получившийся автомат и исходный граф. По определённому ребру мы будем переходить в том и только в том случае, если оно есть в обоих графах. При обнаружении вершины, ключ которой равен искомому, добавляем к списку ответов ещё один. В конце этот список необходимо отсортировать и вывести.

Но если просто реализовать эту схему, программа будет работать слишком долго. Для построения быстрого решения надо её немного оптимизировать. Для этого надо учесть ряд моментов. Во-первых, посмотрим на построение суффиксного автомата. Очевидно, что для одной и той же строки можно построить несколько автоматов, которые будут отличатся своими характеристиками. Мы заинтересованы в том, чтобы автомат имел как можно меньше вершин. Эта даст нам экономию в памяти и экономию во времени при обходе графа. Более того, автомат необходимо строить достаточно быстро, чтобы построение автомата не стало самым долгим элементом в решении задачи. Для этого следует приметь стандартный алгоритм, который строит автомат за линейное время от длины строки, и количество вершин в котором не превышает удвоенной длины строки.

Теперь, когда у нас есть быстро построенный небольшой автомат, обратимся к обходу в глубину. Здесь надо применить несколько эвристических оптимизаций, без которых программа не будет работать. Прежде всего, заметим, что с увеличением глубины обхода мы рассматриваем ключи вершин в невозрастающем порядке. Следовательно, если мы пришли в вершину, ключ которой меньше искомого, можем смело возвращаться назад: здесь мы уже ничего не найдём.

Рассмотрим теперь особенности реализации. Во-первых, при реализации следует обратить внимание на способ хранения пути. Здесь надо обязательно использовать массив символов char[ ], т.к. при использовании данных типа string задача не уложится в предел работы по времени.

Во-вторых, следует обратить внимание на способ представления графа. Если использовать какую-нибудь матрицу смежности или список смежных вершин, то опять-таки это будет не оптимально по времени и по памяти. В данной задачи следует использовать список рёбер, т.к. у нас есть ограничение на количество рёбер и мы знаем, что при большом количестве вершин графы у нас получаются достаточно с разреженной матрицей смежности.

И, наконец, при сортировке следует учесть, что возможных ответов могло получиться

очень много, а, значит, их сортировка может занять значительное время. Поэтому нужно использовать сортировку, которая не испортит нам работоспособность. Например, можно использовать быструю сортировку или сортировку кучей.

В завершении нужно отметить, что программисту следует обратить внимание на тот факт, что глубина обхода может быть очень велика, а значит нужно позаботиться, чтобы стековой памяти хватило для работы программы. Если учесть все эти замечания, то достаточно быстро можно реализовать правильный и эффективный алгоритм решения данной задачи.

Пример реализации на языке Java

```
import java.io.BufferedReader;
import java.io.File;
import java.io.FileReader;
import java.io.IOException;
import java.io.PrintWriter;
import java.io.StreamTokenizer;
import java.util.Arrays;

public class Main{

  public static void main(String[] args) throws IOException {
        new Thread() {
              public void run() {
                    try {
                          new Main().run();
                    } catch (Exception e) {
                    }
              }
        }.start();
  }

  StreamTokenizer st;
  BufferedReader br;
  PrintWriter pw;
  int n, m, uk, need;
  int[] l, id, nxt, first, e, w, s, par, f, count;
  int[][] next;
  boolean[] ends;
  String[] out;
  int countout;
  char[] path = new char[100000];
  int pathcount = 0;

  private void run() throws IOException {
        br = new BufferedReader(new FileReader("password.in"));
        st = new StreamTokenizer(br);
        pw = new PrintWriter("password.out");
        solve();
        int n = nextInt();
        int m = nextInt();
        first = new int[n + 1];
        nxt = new int[m + 1];
        e = new int[m + 1];
        w = new int[m + 1];
        out = new String[10000];
        countout = 0;
```

```
        for (int i = 0; i < m; i++)
            add(nextInt(), nextInt(), nextChar());
        need = nextInt();
        dfs(1, 1);
        Arrays.sort(out, 0, countout);
        for (int i = 0; i < countout; i++)
            pw.println(out[i]);
        pw.close();
    }

    private boolean dfs(int u, int q) {
        boolean ret = false;
        boolean ok = false;
        if (count[q] == need && q != 1)
            ok = true;
        if (count[q] < need)
            return false;
        int v = first[u];
        while (v != 0) {
            if (next[q][w[v]] != 0) {
                path[pathcount++] = (char) (w[v] + 'a' - 1);

                boolean ret1 = dfs(e[v], next[q][w[v]]);
                pathcount--;
                ret = ret || ret1;
            }
            v = nxt[v];
        }
        if (ok && !ret)
            out[countout++] = new String(path, 0, pathcount);

        return (ret || ok);
    }

    private void add(int a, int b, char c) {
        uk++;
        nxt[uk] = first[a];
        first[a] = uk;
        e[uk] = b;
        w[uk] = c;
    }

    private void solve() throws IOException {
        int n, t;
        t = 26;
        st.nextToken();
        String ss = st.sval;
        n = ss.length();
        s = new int[2 * n + 1];
        l = new int[2 * n + 1];
        par = new int[2 * n + 1];
        f = new int[2 * n + 1];
        next = new int[2 * n + 1][t + 1];
        ends = new boolean[2 * n + 1];
        id = new int[2 * n + 1];
        count = new int[2 * n + 1];
        for (int i = 1; i <= n; i++)
            s[i] = ss.charAt(i - 1) - 'a' + 1;
```

```
        int m = 1;
        l[m] = 0;
        par[m] = 0;
        f[m] = 0;
        Arrays.fill(next[m], 0);
        int last = m;
        for (int i = 1; i <= n; i++) {
              m++;
              int v = m;
              par[v] = last;
              l[v] = l[par[v]] + 1;
              int p = last;
              while (p != 0 && next[p][s[i]] == 0) {
                    next[p][s[i]] = v;
                    p = f[p];
              }
              if (p == 0)
                    f[v] = 1;
              else {
                    int q = next[p][s[i]];
                    if (l[q] - l[p] == 1)
                          f[v] = q;
                    else {
                          m++;
                          int r = m;
                          System.arraycopy(next[q],0, next[r], 0, t + 1);
                          par[r] = p;
                          l[r] = l[p] + 1;
                          f[r] = f[q];
                          f[v] = r;
                          f[q] = r;
                          while (p != 0 && next[p][s[i]] == q) {
                                next[p][s[i]] = r;
                                p = f[p];
                          }
                    }
              }
              last = v;
        }
        int u = last;
        while (u != 0) {
              ends[u] = true;
              u = f[u];
        }
        for (int i = 1; i <= m; i++)
              id[i] = i;
        heapsort(m);
        for (int i = m; i >= 1; i--) {
              u = id[i];
              count[u] = 0;
              if (ends[u])
                    count[u] = 1;
              for (int c = 1; c <= t; c++)
                    if (next[u][c] != 0)
                          count[u] += count[next[u][c]];
        }
}
```

```
  private void heapsort(int m) {
        int size = m;
        for (int i = size / 2; i >= 1; i--)
              heapify(size, i);
        while (size > 0) {
              int t = id[1];
              id[1] = id[size];
              id[size] = t;
              size--;
              heapify(size, 1);
        }

  }

  private void heapify(int size, int k) {
        while (true) {
              int q = k;
              if (2 * k <= size)
                    if (l[id[q]] < l[id[2 * k]])
                          q = 2 * k;
              if (2 * k + 1 <= size)
                    if (l[id[q]] < l[id[2 * k + 1]])
                          q = 2 * k + 1;
              if (k != q) {
                    int t = id[k];
                    id[k] = id[q];
                    id[q] = t;
                    k = q;
              } else
                    break;
        }
  }

  private char nextChar() throws IOException {
        st.nextToken();
        return (char) (st.sval.charAt(0) - 'a' + 1);
  }

  private int nextInt() throws NumberFormatException, IOException {
        st.nextToken();
        return (int) st.nval;
  }

}
```

МЕТОДИКА СОЗДАНИЯ НАБОРА ЗАДАЧ ДЛЯ ОЛИМПИАДЫ ПО ПРОГРАММИРОВАНИЮ

Гозман Д.М. – студент гр. ПОВТ– 32

Основными целями и задачами профессиональных олимпиад является развитие у студентов интереса у научной деятельности, активизация изучения спецкурсов, создание оптимальных условий для выявления одаренных студентов и их дальнейшего интеллектуального роста в профессиональной области, развитие умения работать в коллективе.

Для формирования более ясного представления о том, какой набор задач наиболее интересен для участников и жюри, необходимо проанализировать наборы заданий на известных соревнованиях, в частности ACM ICPC, NEERC, олимпиады на серверах acm.timus.ru, acm.zju.edu.cn, acm.sgu.ru, acm.uva.es.

Анализ показывает, что наиболее часто в набор включаются девять – двенадцать задач различной тематики:

1. теория графов и комбинаторика,
2. вычислительная геометрия,
3. структуры данных,
4. NP-полные задачи, методы перебора и отсечения,
5. динамическое программирование,
6. на логическое мышление,
7. дискретный анализ и теория игр.

Кроме того, в наборе всегда присутствуют задачи различной степени сложности: от простых задач до сложных или практически не решаемых за отведенное участникам время. Это объясняется самой технологией прохождения олимпиад, различным уровнем участников. Очень важен и психологический аспект: участнику должно быть интересно решать задачи, и даже самые сильные не должны решить все задачи до окончания олимпиады.

Рассмотрим, например набор задач, разработанный автором для Всероссийских тренировочных сборов студентов ведущих вузов страны для подготовки к финалу чемпионата мира по программированию (http://acm.timus.ru/monitor.aspx?id=58). В комплект включены девять задач направленные на развитие логического мышления и использование стандартных алгоритмов. Рассмотрим наиболее интересные из них.

Задача «Последовательность»

Известна рекуррентная формула для последовательности f:

$f(n) = 1 + f(1)g(1) + f(2)g(2) + ... + f(n\text{-}1)g(n\text{-}1)$*, где последовательность также задается рекуррентно:*

$g(n) = 1 + 2g(1) + 2g(2) + ... + 2g(n\text{-}1) – g(n\text{-}1)g(n\text{-}1)$.

Известно, что $f(1)=1$ *и* $g(1) = 1$.

Ваша задача найти $f(n)$ *по заданному модулю p.*

Решение

Методом математической индукции несложно доказать, что $g(n)=n$ и $f(n) = n!$. Таким образом, задача тривиальна в написании и направлена только на умение работать с рекуррентными формулами.

Задача «Марсианские тарелки»

В марсианском ресторане меню состоит из n блюд, а праздничный обед из l блюд. Некоторые блюда могут быть сервированы несколько раз в течение праздничного обеда, а другие ни разу. Во время обеда официант ставит тарелки одну на другую. В некоторые моменты времени он выносит новое блюдо и ставит его поверх стопки тарелок на столе, в другие моменты он уносит верхнюю пустую тарелку. При этом, в силу марсианских

традиций, некоторые блюда нельзя выносить, пока на столе стоят тарелки из-под других блюд.

Расписание официанта – это последовательность сервировки блюд и уноса тарелок со стола. В этом расписании t = 2l пунктов. Ваша задача – узнать, сколько существует различных расписаний для официанта по модулю p.

Решение

Переформулируем задачу в математических терминах: необходимо найти количество правильных скобочных последовательностей из *n* различных типов скобок, причем существуют ограничения на использование одних типов скобок внутри других. Совершенно очевидно, что при поставленных ограничениях задача не решается методом полного перебора, поскольку при максимальных входных данных количество различных скобочных последовательностей превышает 10^{20}.

В задачах такого типа чаще всего применим метод динамического программирования. Введем функцию *f*(l, M) – количество правильных скобочных последовательностей длины 2*l, состоящих из скобок множества M. Тогда *f*(t/2, {1 .. n}) – ответ на поставленную задачу. Пусть G(λ) – множество типов скобок, которые не могут находиться внутри пары соответствующих друг другу скобок типа λ. Очевидно, что для вычисления функции *f* можно воспользоваться формулами:

1) *f*(0, M) = 1

2) *f*(l, M) = sum {*f*(i, M\G(λ)) * *f*(l-1-i, M) | λ ∈ M, i = 0 .. l -1} для l = 1 .. t/2.

Теперь, закодировав множества битовыми масками, мы можем вести вычисления в двумерном массиве.

Задача «Плохие дороги»

Страна состоит из нескольких городов, соединенных дорогами. Из-за дефицита бюджета, на некоторых дорогах есть ямы, поэтому некоторые машины не могут проехать по некоторым дорогам. В виду этого, министерство присвоило каждой дороге некоторую величину – минимальную высоту посадки автомобиля, который может проехать по этой дороге. Кроме того, некоторые дороги платные – для проезда по ним необходимо заплатить одну условную единицу. И наконец, для каждой дороги известно время путешествия по ней.

Ваша задача состоит в определении минимальной посадки автомобиля, на котором можно добраться из города s в город t не более чем за maxtime времени, заплатив не более money условных единиц.

Решение

Это пример задачи на классические алгоритмы теории графов и двоичный поиск.

Построим новый ориентированный граф: его вершинами будут пары <u, d>, где u соответствует городу, а d – количеству условных единиц, потраченных водителем. Дуга (<u, d> → <v, e>) принадлежит графу, если дуга (u → v) принадлежала исходному и e = d + money(u → v). Длина новой дуги будет равна длине дуги (u → v). Теперь задача свелась к поиску пути в этом графе из вершины (s, 0) в вершины (t, d), d = 0..money длины не более maxtime с минимальной возможной посадкой автомобиля.

Теперь необходимо применить прием, помогающий в большом количестве задач по программированию - двоичный поиск по ответу задачи d. Для этого необходимо уметь проверять, есть ли в графе путь длины не более maxtime такой, что посадка автомобиля для всех дорог не более d. Это легко сделать, ограничив граф, оставив в нем только ребра с посадкой автомобиля не более d, и найдя в нем кратчайший путь (например, алгоритмом Дейкстры с использованием кучи). Если этот путь длины не более maxtime, то решение с посадкой d существует, в противном случае – нет.

Применение метода двоичного поиска в данном случае означает использование отрезка [a; b], ограничивающего правильный ответ. Первоначальные значения границ таковы, что

решение на отрезке существует (например [0; максимальная посадка]). На каждой итерации необходимо проверить середину отрезка d = (a + b)/2. Если для этого d существует решение (путь длины не более maxtime), то за новый отрезок принимается [d; b], иначе [a; d]. При равенстве границ отрезка решение найдено.

МЕТОД ДИНАМИЧЕСКОГО ПРОГРАММИРОВАНИЯ ПРИ РЕШЕНИИ ЗАДАЧ

Гоменюк Р.В. – студент гр. ПОВТ– 42

Метод динамического программирования является одним из ключевых методов при решении олимпиадных задач. Гибкость этого метода и простота реализации программы делает этот метод эффективным при решении олимпиадных задач.

Идея метода

Суть метода динамического программирования заключается в следующем:

Пусть нам необходимо решить задачу размерности N. Будем решать ее итеративно, т.е. последовательно строить решение, сначала для малой размерности, а потом будем увеличивать размерность и строить решение для данной размерности на основе более меньших размерностей. Для начала требуется сформировать базу динамического программирования, т.е. решение задачи для какой-либо малой размерности. Далее на основе этой базы будем строить решение задачи большей размерности. Например рассмотрим простую задачу нахождения факториала:

Пусть требуется посчитать N!, т.е. посчитать число 1*2*3*...*N. Пусть f(i) хранит решение для задачи размерности i, т.е. i факториал, база индукции - решение для задачи размерности 0, т.е. f(0) = 1, тогда для вычисления решения для задачи размерности i требуется воспользоваться решением задачи размерности (i-1):

f(i) = f(i-1)*i.

Данный пример полностью отражает главную суть метода динамического программ-мирования - решение задачи размерности N строится через решения задач размерности меньшей N.

Рассмотрим алгоритм решения задачи методом динамического программирования:

1) Построение базы динамического программирования (в нашем примере это f(0) = 1)
2) Построение динамического перехода, т.е. алгоритм перехода от меньших решений к большим (в нашем примере это f(i) = f(i-1)*i).

Применение метода

Рассмотрим одну из задач, которая предлагалась автором на прошедшей олимпиаде по программированию “Университеты Алтая – 2007”
(http://neerc.secna.ru/ALTAI_U/2007/res_2007.htm#Label_D).

Условие задачи. Задача D. Названия стихотворений

На недавно прошедшем референдуме смешарики приняли поправки к статье «Названия стихотворений» закона «О Защите Авторских Прав». Раньше закон требовал, чтобы называния стихотворений были последовательностями из 0 и 1. А сейчас необходимо, чтобы название каждого нового стихотворения не только состояло из 0 и 1, но и не содержало в себе названий других, уже опубликованных произведений.

После референдума, смешарики опубликовали на своем сайте список названий уже существующих произведений. И с появлением каждого нового стихотворения решили обновлять этот список.

Узнав о таких изменениях в законодательстве, Бараш решил называть все свои шедевры последовательностями длины K. Он зашел на сайт смешариков и увидел, что в списке уже есть N чужих произведений. Ему стало интересно, сколько еще стихотворений он сможет сочинить, не нарушая новый закон. Начав считать, Бараш понял, что это слишком сложно и ему с этим не справиться. Помогите Барашу определить, сколько стихотворений он сможет сочинить. Бараш подозревает, что таких стихотворений будет слишком много, поэтому он просит вывести не все число, а взятое по модулю P.

Входные данные

*В первой строке даны три числа N, K (K ≤ 1000) и P (P ≤ 2*109). В последующих N строках записан список названий. Каждое название представляет собой последовательность нулей и единиц. Длина слов не превышает 15. Известно, что до принятия закона некоторые названия стихотворений могли совпадать, но из списка их не изъяли.*

Выходные данные

Вывести число, равное количеству стихотворений по модулю P, которые сможет сочинить Бараш, не меняя своего правила о длине названия и не нарушая закон.

Решение задачи

Теперь постараемся применить метод динамического программирования для решения рассматриваемой задачи задачи. Пусть функция f(s, n) означает количество строк длины n, не содержащих заданных слов, таких что последние 15 символов образуют строку s.

8. Если строка s не содержит заданных слов, то f(s, 15)=1, иначе f(s, 15)=0.
9. Если строка s не содержит заданных слов, то f(s, n) = sum({f(x + s[1..14], n-1) | x = 0..1}), иначе f(s, n)=0.

Пример реализации

```
import java.io.File;
import java.io.FileNotFoundException;
import java.io.PrintWriter;
import java.util.Arrays;
import java.util.HashSet;
import java.util.Scanner;



public class Main {

  static int[][] a = new int[2][1 << 15];
  static int[] goodMask = new int[1 << 15];
  static int len = 0;
  static HashSet<String> words = new HashSet<String>();

  public static void main(String[] args) throws FileNotFoundException
{
        Scanner in = new Scanner(new File("forbidden.in"));
        PrintWriter out = new PrintWriter(new File("forbidden.out"));
        int n = in.nextInt();
        int k = in.nextInt();
        int p = in.nextInt();
        in.nextLine();
        for (int i=0;i<n;i++){
             String s = in.nextLine();
             s = s.trim();
             if (s.length() > len)  len = s.length();
             words.add(s);
        }

        if (len > k) len = k;
```

```
        Arrays.fill(goodMask, 0);
        for (int mask=0;mask < 1 << len;mask++){
             if (ok(mask)) {
                  goodMask[mask] = 1;
                  a[0][mask] = 1;
             }
        }


        int q=0;

        for (int cnt=len+1;cnt<=k;cnt++){
             for (int mask=0;mask < 1 << len;mask++){
                  int nm1 = mask << 1;
                  nm1 = nm1 & (~(1 << len));
                  int nm2 = nm1 + 1;
                  nm2 = nm2 & (~(1 << len));
                  if (goodMask[nm1] == 1)
                       a[1-q][nm1] = (a[1-q][nm1] + a[q][mask]) % p;
                  if (goodMask[nm2] == 1)
                       a[1-q][nm2] = (a[1-q][nm2] + a[q][mask]) % p;
             }
             q = 1 - q;
        }

        // count and write answer to file
        int ans = 0;
        for (int mask=0;mask < 1 << len;mask++){
             ans = (ans + a[q][mask]) % p;
        }
        out.println(ans % p);
        out.close();
        in.close();
   }

   private static boolean ok(int mask) {
        String s = "";
        for (int i=0;i<len;i++){
             if (mask % 2 == 0) s = "0"+s; else
                                     s = "1"+s;
             mask = mask >> 1;
        }
        for (int i=0;i<s.length();i++){
             for (int j=i+1;j<=s.length();j++){
                  String ss = s.substring(i, j);
                  if (words.contains(ss)) return false;
             }
        }
        return true;
   }

}
```

РЕШЕНИЕ ЗАДАЧ НА ОБРАБОТКУ ТЕКСТА

Колосовский М.А. – студент гр. ПОВТ– 52

Задачи обработки текстов, как правило, решаются на основе методов синтаксического анализа контекстно-свободных грамматик. Рассмотрим ,например, задачу B. "Макросы" предложенную автором на интернет-олимпиаду "Университеты Алтая 2007". Данная задача является представителем широкого класса задач на обработку текста. Зачастую, чтобы усложнить задание, в правила обработки включаются рекурсивные определения. Например, в задаче "Макросы" внутри одного макроопределения могут встретится другие макроопределения или внутри одного макровызова может встретиться другие макровызовы. Очевидно, что нагляднее всего такие задачи решаются рекурсией.

В условии задачи определено такое понятие, как блок (это текст, заключенный между фигурными скобками, весь текст также считается блоком). Каждый рекурсивный вызов обрабатывает точно один блок, после обработки текущего блока происходит выход из подпрограммы и происходит обработка блока, в котором содержался текущий блок. Такой подпрограмме передаются список уже определенных макроопределений, а также используется глобальная переменная, хранящая индекс первого необработанного символа во входной строке. Возвращает подпрограмма строчку, которую порождает данный блок. Внутри рекурсивной процедуры происходит поочередная обработка команд (или отдельных символов), составляющих текущий блок. В зависимости от типа следующей команды выполняются различные действия:

- если далее идет символ английского алфавита, то следует его добавить к результату, который вернет подпрограмма.
- если найдено новое макроопределение, то найти строчку, которую порождает это макроопределение, вызвав рекурсивно нашу подпрограмму. Запомнить это макроопределение в списке.
- если далее следует макровызов, то при наличии такого макроопределения в списке добавить строчку, которою оно порождает к результату.
- если нашли циклическое макроопределение, то найти строчку, которую порождает ее блок и добавить ее заданное число раз.

Отдельно стоит обратить внимание на то, как распознавать что следует дальше: отдельный символ, макроопределение, макровызов или циклическое макроопределение. Для этой цели используется служебный символ <#> (диез). Получается следующий алгоритм:

1. Если следующий символ <#> (диез), то это либо макроопределение (возможно циклическое), либо макровызов.
   1.1. если встретился еще один служебный символ (<#>), то это макровызов. Определяем его имя и вызываем, проверив существование такого в списке.
   1.2. если далее следует символ алфавита, то либо циклическое макроопределение, либо нет.
      1.2.1. если далее следует слово <rep>, то циклическое макроопределение.
      1.2.2. если какое-то другое, то это обычное макроопределение.
2. Если следующий символ - символ английского алфавита, то просто добавляем его к результату.

Для более сложных правил пишется конечный автомат, а не тождественный ему набор условий. Однако в данной задаче можно ограничиться таким алгоритмом. Как правило, в задачах на обработку текста все необходимые алгоритмы задаются уже в условии, остается лишь понять, как это все собрать вместе. В данной задаче нужно было только понять, что каждый блок должен быть обработан отдельным вызовом рекурсивной подпрограммы, и определить какие параметры передавать этой подпрограмме.

Рассмотрим вспомогательные функции

`private String find(String name, int c)` - найти элемент `name` в списке из c элементов,

`private int getNum()` - считать последовательность цифр,

`private String getName()` - считать последовательность букв.

Тогда рекурсивная функция `private String print(int cd)` будет выполнять основную работу по преобразованию текста.

```
import java.io.BufferedReader;
import java.io.FileReader;
import java.io.IOException;
import java.io.PrintWriter;

public class Main {
  private static final int MAXD = 10000;

  public static void main(String[] args) throws IOException {
       new Main().run();
  }

  char[] s; // входная последовательность символов
  int k = 0; // первый необработанный символ в s
  String[][] defs = new String[MAXD][2]; // список объявленных
                                         // макроопределений
  private void run() throws IOException {
       BufferedReader in = new BufferedReader(new
FileReader("macros.in"));
       PrintWriter out = new PrintWriter("macros.out");
       StringBuilder buf = new StringBuilder();
       while (in.ready())
            buf.append(((char)in.read()));
       s = (buf.toString()).toCharArray();
       out.print(print(0));
       out.close();
  }

  // основная подпрограмма, обрабатывающая один блок
  private String print(int cd) {
       // cd - число эелементов в списке defs
       StringBuilder res = new StringBuilder();
       while (k < s.length && s[k] != '}')
            if (s[k] == '#') {
                 k++;
                 if (s[k] == '#') { // макровызов
                      k++;
                      String name = getName();
                      res.append(find(name, cd));
                 } else {
                      String name = getName();
                      if (name.compareTo("rep") != 0) {
                       // циклич. макроопределение
                           k++;
                           defs[cd][0] = name;
                           defs[cd++][1] = print(cd);
                           k++;
                      } else { // макроопределение
                           int n = getNum();
```

```
                              k++;
                              String str = print(cd);
                              k++;
                              for (int i=0; i < n; i++)
                                   res.append(str);
                         }
                    }
               } else
                    res.append(s[k++]);
          return res.toString();
     }

     private String find(String name, int c) {
          // найти элемент name  в списке из с элементов
          for (int i=c - 1; i >= 0; i--)
               // обязательно с конца просмотр!
               if (defs[i][0].compareTo(name) == 0)
                    return defs[i][1];
          return new String();
     }

     private int getNum() { // считать последовательность цифр
          StringBuilder res = new StringBuilder();
          while (Character.isDigit(s[k]))
               res.append(s[k++]);
          k++;
          return Integer.parseInt(res.toString());
     }

     private String getName() { // считать последовательность букв
          StringBuilder res = new StringBuilder();
          while (Character.isLetter(s[k]))
               res.append(s[k++]);
          k++;
          return res.toString();
     }

}
```

СЛОЖНЫЕ ПРОБЛЕМЫ РЕШЕНИЯ ПРОСТЫХ ЗАДАЧ

Луценко Е.В., Закурдаев А.В. – студенты гр. ПОВТ–62

При решении казалось бы простых задач часто возникают проблемы, которые не всегда очевидны для неопытного программиста, но вызывают трудности при решении. Один из вопросов, который нужно рассмотреть в первую очередь– это вопрос выбора типов данных. На выбор влияют ограничения по значениям, ограничения по памяти. В качестве примера рассмотрим предложенные авторами задачи для интернет-олимпиады "Универ-ситеты Алтая 2007", которая проводилась на сайте http://neerc/secna.ru/ALTAI_U/2007. Рассмотрим, например, задачу "Центральное отопление".

Условие задачи

Кар Карыч с Пином восемнадцать часов подряд распивали холодные молочные коктейли и закусывали их мороженым. После этого Кар Карыч свалился со страшной простудой, а Пин решил провести в домик своему другу центральное отопление. Расчет количества отопительных приборов необходимо производить строго по ГОСТу 800333-90-06. Для простоты Пин решил купить простые батареи. Согласно таблице 14.1.3 этого ГОСТа, каждая батарея обогревает определённый объём воздуха - ровно K кубометров. Комната, которую собирается для своего друга обогреть Пин, имеет следующие размеры:*

- *высота H,*
- *ширина W,*
- *длина L.*

Определите, какое минимальное количество батарей Пину необходимо купить. Учтите только, что если в домике у Кар Карыча температура будет ниже, чем по ГОСТу, Кар Карыч никогда не поправится.

Входные данные

На единственной строке входного файла записано четыре целых числа: H, W, L, K. (*H, W, L*<=10^5, *K*<=$2*10^9$).

Выходные данные

На единственной строке выходного файла выведите ответ на задачу.

Исследование проблемы

Данная проблема представляет собой типичный пример простейшей задачи. Её решение, очевидно, заключается в перемножении трёх чисел и делении их произведения на четвёртое с последующим округлением результата вверх. Но главная трудность заключается не в нахождении алгоритма решения, а в правильности выбора типа данных для хранения переменных. Заметим, что из ограничений на входные данные, можно сделать вывод, что максимально возможное значение произведения равняется 10^{15}, из чего следует необходимость использования соответствующего типа данных, допускающего хранение 64 - разрядного числа. Можно предложить два варианта: целочисленный тип или тип с плавающей точкой. Если использовать целочисленный тип, то после выполнения деления для получения верного результата следует проверять, делилось ли число с остатком или без и в случае остатка прибавлять единицу к частному. Если использовать тип с плавающей точкой то важно учесть, что не во всех языках программирования есть встроенная функция округления вверх и поэтому при решении данной задачи может возникнуть необходимость написания её вручную. Заметим, что вариант прибавления единицы к целой части числа не всегда верен, поскольку он не работает для целых чисел. Поэтому следует проверять является ли число целым, и если нет, то применять описанный выше вариант.

Приведём тексты решения с использованием целочисленного типа и с использованием типа с плавающей точкой (оба решения написаны на языке C++)

Решение для данных с плавающей точкой

```
#include <fstream.h>
#include <math.h>
float eps=0.00000000001,x,y,z,n;

int RoundUp(float x){
 if (fabs(x-floor(x)>eps))
   return floor(x)+1;
 else
   return floor(x);
}

void main(){
 ifstream in("center.in");
 ofstream out("center.out");
 in>>x>>y>>z>>n;
 out<<RoundUp(x*y*z/n);
 out.close();
 in.close();
}
```

Решение с применением целочисленного типа

```
#include <fstream.h>
#include <math.h>
long long x,y,z,n;

void main(){
 ifstream in("center.in");
 ofstream out("center.out");
 in>>x>>y>>z>>n;
 out<<((x*y*z%n)?(x*y*z/n+1):(x*y*z/n));
 out.close();
 in.close();
}
```

Перейдем теперь к вопросу выбора алгоритма. Несмотря на простоту формулировки не всегда простая задача предполагает тривиальное решение типа полного перебора. Рассмотрим задачу ”Компотозаготовка” из вышеуказанной олимпиады

Условие задачи

Ежегодно Совунья заготавливает на зиму компоты для себя и своих друзей. Фрукты, которые она использует для этого, растут на том же дереве, где находится ее домик. Все было бы хорошо, если бы ее дерево не росло, а вместе с ним не увеличивались бы урожаи фруктов. В один прекрасный вечер, заготавливая компоты, она поняла, что просто не справляется с возросшим количеством фруктов.

К счастью, Пину в ближайшее время было совершенно нечего делать, так как ближайшее обновление Slackware Linux ожидалось нескоро, а предыдущую версию он уже изучил вдоль и поперек. Поэтому Пин помог Совунье с улучшением агрегатов для автоматизации процесса компотозакатывания с целью повысить производительность труда и поднять коэффициент полезного действия Совуньи в этом благом начинании.

Попутно Пин помог Совунье допить запасы забродившего компота за два предыдущих года. Так как Совунья была хозяйственная, она понимала, что из забродившего компота можно сделать настоящий Эликсир Вдохновения. А поэтому запасы такого компота у нее были всегда...

В итоге деятельности Пина агрегатов стало на K больше, чем было до этого, и общая производительность комплекса увеличилась. Все агрегаты имеют одинаковую производительность, выражаемую в банках в час. Если до реконструкции производство Совуньи в целом выдавало N банок компота в день, то после реконструкции оно стало выдавать в целом M банок компота в день. Впрочем, это не стало означать, что стал оставаться лишний компот– просто его стали быстрее выпивать.

Напишите программу, которая по входным данным, числам K, N, M определит, сколько компотных агрегатов могло быть до реконструкции, и выведет всевозможные варианты ответов в порядке возрастания в выходной файл. Считается, что возможные варианты есть всегда.

Входные данные

Последовательно в строках входного файла записаны целые числа K, N, M ($0 < K, N, M \leq$ 2000000000).

Выходные данные

Записать в выходной файл искомые числа в порядке возрастания по одному числу на строке.

Исследование проблемы

Первая проблема заключается в составлении алгоритма для решения задачи. Заметим, что как до, так и после реконструкции, производительность одного агрегата выражается целым числом. Таким образом, учитывая, что производительность всех агрегатов равна между собой, можно сделать вывод, что, если за X обозначить число агрегатов до реконструкции, то N/X – это производительность одного агрегата до реконструкции, а M/(X+K) – после реконструкции. Как уже было сказано ранее, производительность одного агрегата выражается целым числом, поэтому числа N/X и M/(X+K) должны быть целыми. Таким образом, задача сводится к нахождению таких значений X, чтобы числа N/X и M/(X+K) были целыми. Из того, что из того, что N/X – целое, следует, что X – делитель числа N, и, так как по условию задачи $N<=2*10^9$, то возникает вторая проблема – нахождение всех его делителей. Так N очень велико, то вариант нахождения всех его делителей путём перебора их от единицы до N/2, будет работать очень медленно (примерно 30 сек.). Предложим более быстрый вариант: будем перебирать делители не до N/2, а до округлённого вниз квадратного корня из N. Этот метод верен, так как для каждого делителя I из диапазона [1 $\sqrt{N}$] существует делитель N/I из диапазона [$\sqrt{N}$ N/2], поэтому для нахождения всех делителей числа N нет необходимости перебирать числа, большие чем округлённый вниз квадратный корень из N. Ещё один момент, который необходимо учесть, заключается в том, что, если N – полный квадрат, то при I равном квадратному корню из N, числа I и N/I совпадут, и мы получим два одинаковых делителя. Очевидно, что один из них нужно исключить.

Текст решения на языке C++

```
#include <fstream.h>
#include <math.h>
int n,m,k,kol;
int a[100000];

void init(){
```

```
 ifstream in("compot.in");
 in>>k>>n>>m;
 in.close();
}

void sort(int x){
 for (int i=1;i<=x-1;i++)
   for (int j=i+1;j<=x;j++)
     if (a[j-1]>a[j]){
       int t=a[j];
       a[j]=a[j-1];
       a[j-1]=t;
     }
}

void run(){
 kol=0;
 for (int i=1;i<=floor(sqrt(n));i++)
   if (n%i==0){
     kol++;
     a[kol]=i;
     kol++;
     a[kol]=n/i;
 }
 if (a[kol]==a[kol-1])
   kol--;
 sort(kol);
 ofstream out("compot.out");
 for (int i=1;i<=kol;i++)
   if (n%a[i]==0 && m%(a[i]+k)==0)
     out<<a[i]<<'\n';
 out.close();
}

void main(){
 init();
 run();
}
```

ПОТОКИ В СЕТЯХ

Могозов А.А. – студент гр. ПОВТ–42

Графы часто используются в качестве математической модели при решении многих прикладных задач. Поэтому такие задачи часто встречаются в заданиях олимпиад по программированию. Рассмотрим одну из задач, которая предлагалась автором на прошедшей олимпиаде по программированию Университеты Алтая – 2007 (http://neerc.secna.ru/ALTAI_U/2007/res_2007.htm#Label_C).

Условие задачи С. “К чему приводит защита от копирования”

Давным-давно, в далекой-далекой галактике, когда еще не вышел мультфильм про смешариков, никто не знал про Гарри Поттера и про Властелина Колец, на далекой-далекой планете жили-были полчища смешариков. Их технологии были настолько совершенны, что они создали машину времени и перенеслись на ней в будущее, на планету “Земля”, где одному из них совершенно случайно попалась первая серия “Смешариков”. Исследователей эта серия так потрясла, что они предприняли чрезвычайно опасный рейд, в ходе которого им удалось добыть полное собрание серий. Эти серии они увезли на родину, где они стали безумно популярными. К сожалению, мультфильмы были с системой защиты от копирования, а смешарики по своей законопослушной сущности не приспособлены к хакерской деятельности. Поэтому им пришлось обмениваться привезенными с Земли дисками.

Местная поп-звезда Билаш обиделся на такую популярность, к которой он не имел никакого отношения, и решил вернуть все в старое русло. Для этого Билаш хочет рассорить смешариков, чтобы они разделились на два не общающихся между собой лагеря. Для того, чтобы поссорить пару смешариков, Билашу требуется израсходовать 1 у.е. усилий. Но, так как Билаш жутко ленив, он хочет приложить минимум усилий для достижения своей цели. Помогите ему.

Входные данные

На первой строке два числа N (N≤100) и M - количество смешариков и количество пар смешариков, которые обмениваются мультфильмами. На последующих M строках перечисляются пары чисел U и V, означающих, что смешарик U и смешарик V знакомы друг с другом и обмениваются мультфильмами.

Выходные данные

Вывести минимальное число у.е., которое придется затратить Билашу на достижение своей цели.

Проанализируем условие задачи. Пусть каждый смешарик – вершина графа, а пара смешариков, знакомых друг с другом – ребро графа. Понятно, что требуется из полученного графа “удалить” минимальное число ребер, чтобы граф стал несвязным. Для решения задачи в новой формулировке, необходимы понятия сети и потока в сети.

Представим ориентированный граф, как сеть труб, по которым некоторое вещество движется от вершины, называемой истоком к вершине, называемой стоком. Естественно, что по каждой трубе не может передвигаться больше определенного количества вещества. Это количество назовем пропускной способностью трубы. Также будем предполагать, что вещество в вершинах, кроме истока и стока, не накапливается и ниоткуда не приходит. Это свойство назовем законом сохранения потока.

Задача о построении максимального потока в такой сети заключается в нахождении максимальной скорости производства вещества, при которой его можно доставить от истока к стоку, не нарушая закон сохранения потока и не превышая пропускных способностей труб.

Формальное описание задачи о построении максимального потока

Назовем сетью ориентированный граф G = (V, E), в котором каждому ребру сопоставлено некоторое число c(u, v) ≥ 0, u, v∈V, называемое пропускной способностью ребра. В случае (u,v)∉E, полагаем c(u,v)=0. В графе выделены две вершины s – исток и t – сток. Потоком в графе назовем функцию

$$f: V \times V \rightarrow R,$$

обладающую тремя свойствами:

1) $f(u, v) \le c(u, v)$ $u,v \in V$
2) $f(u, v) = - f(v, u)$ $u,v \in V$ – кососимметричность
3) $u \in V-\{s, t\}$ $\sum_{v} f(u,v) = 0$ – сохранение потока

Говоря неформально, описанные свойства означают следующее:

- первое говорит о том, что величина потока по ребру не может превышать его пропускной способности;
- второе означает, что поток по ребру в прямом направлении по абсолютной величине равен потоку в обратном направлении;
- третье означает, что для каждой вершины, кроме истока и стока сумма потоков во все вершины равна нулю.

Величина потока определяется, как сумма:

$$|f| = \sum_{u} f(s,u)$$

Задача о максимальном потоке заключается в поиске для данной сети G с истоком s и стоком t потока максимальной величины.

Для поиска максимального потока существует метод Форда-Фалкерсона, который заключается в следующем:

3) Положим поток f равным 0
4) Пока в графе существует дополняющий путь из s в t, дополним поток вдоль f

Одной из простейших реализаций метода Форда-Фалкерсона является алгоритм Эдмондса-Карпа, в котором поиск дополняющего пути осуществляется поиском в ширину.

Разрезом сети G = (V,E) назовем разбиение множества V на две части S и T = V \ S, причем s∈S и t∈T. Эти части назовем долями. Пропускной способностью разреза назовем сумму:

$$\sum c(u,v),\ u \in S,\ v \in T$$

Потоком через разрез назовем сумму:

$$\sum f(u,v),\ u \in S,\ v \in T$$

Минимальным разрезом называется разрез минимальной пропускной способности. Существует теорема, гласящая о том, что величина максимального потока равна пропускной способности минимального разреза.

Вернемся к решению рассматриваемой задачи. Для решения предлагается следующий метод. Построим по данному графу сеть. Пропускные способности всех ребер положим равными 1. Зафиксируем две вершины графа в качестве истока и стока. Найдем максимальный поток в данной сети. Его величина равна пропускной способности минимального разреза в этой сети. Поэтому, если удалить насыщенные ребра, пересекающие минимальный разрез, граф станет несвязным – пути между истоком и стоком в сети существовать не будет. Отсюда виден метод решения задачи: переберем вершины,

соответствующие истоку и стоку и найдем для каждого случая максимальный поток. Минимальное значение максимального потока и будет ответом на задачу.

На самом деле, достаточно зафиксировать исток один раз и перебирать только вершину, соответствующую стоку. В предлагаемом решении поиск максимального потока осуществляется методом масштабирования, который заключается в следующем:

1. Положим поток f, равным 0
2. Положим x равным бесконечности (достаточно взять большое число, например миллиард)
3. Пока x > 0
   Пока существует путь с пропускной способностью, не менее x, увеличим поток вдоль этого пути
   x = x / 2

Пример реализации

```
const
  maxn = 101;
  inf = 1000000000;

var
  c, f : array[1..2*maxn, 1..2*maxn] of longint;
  p, col : array[1..2*maxn] of longint;
  n, s, t : longint;

function dfs(u, x : longint) : boolean;
var
  v : longint;
begin
  col[u] := 1;
  if u = t then begin
    dfs := true;
    exit;
  end;
  dfs := false;
  for v := 1 to 2*n do
    if (c[u, v] - f[u, v] >= x) and (col[v] = 0) then
      if dfs(v, x) then begin
        p[v] := u;
        dfs := true;
        exit;
      end;
end;

procedure increase(x : longint);
var
  u, v : longint;
begin
  v := t;
  while v <> s do begin
    u := p[v];
    f[u, v] := f[u, v] + x;
    f[v, u] := -f[u, v];
    v := u;
  end;
end;
```

```
var
  u, v, m, i, flow, cflow, mflow : longint;

begin
  reset(input, 'bilash.in');
  rewrite(output, 'bilash.out');
  read(n, m);
  fillchar(c, sizeof(c), 0);
  for i := 1 to m do begin
    read(u, v);
    c[u + n, v] := 1;
    c[v + n, u] := 1;
  end;
  for u := 1 to n do
    c[u, u + n] := inf;

  s := 1 + n;                          //source
  mflow := inf;

  for t := 2 to n do begin             //sink
    fillchar(f, sizeof(f), 0);
    flow := inf;
    cflow := 0;
    while flow > 0 do begin            //scaling
      fillchar(col, sizeof(col), 0);
      while dfs(s, flow) do begin
        increase(flow);
        inc(cflow, flow);
        fillchar(col, sizeof(col), 0);
      end;
      flow := flow div 2;
    end;
    if cflow < mflow then
      mflow := cflow;
  end;

  writeln(mflow);
end.
```

ПРИМЕНЕНИЕ ПОВОРОТНОЙ ГЕОМЕТРИИ ПРИ РЕШЕНИИ ГЕОМЕТРИЧЕСКИХ ЗАДАЧ

Отморский С.А. – студент гр. ПОВТ– 43

Часто при решении геометрических задач вычисления становятся проще если систему координат повернуть на некоторый угол и перенести начало координат в некоторую точку. Поворотная геометрия позволяет получить координаты точек в новой системе координат. Тогда при известных координатах (x,y), угле поворота α и одновременном переносе начала координат в точку (x0,y0) новые координаты (x1,y1) вычисляются по формуле:

$$x1=x\cos(\alpha) - y\sin(\alpha) - x0$$
$$y1=y\cos(\alpha) + x\sin(\alpha) - y0$$

Рассмотрим применение метода при решении задачи из Интернет-олимпиады “Университеты Алтая 2006” (http://neerc.secna.ru/altai_u/2006)

Задача

Успешно пройдя предыдущее испытание, Винни Пух попал в следующую комнату, которая оказалась гробницей фараона Тутпуханета XVII. Это была волшебная комната бесконечных размеров. В гробнице располагались непересекающиеся зеркала двух типов: одни представляли собой цилиндр, а другие - прямоугольник. На полу на древнем языке было написано: "Оставь надежду всяк сюда входящий, ибо дверь откроется только тому, кто попадет из хитроумного и очень древнего лучевого пистолета прямо в правый глаз мумии. При этом допускается не более 10000 отражений пули от зеркал". Винни знал древний язык и прочитал это. Сначала задача показалась ему легкой, но потом Пух обнаружил, что стрелять можно только стоя на специальной подставке, иначе пистолет работать не будет. К счастью, методом проб и ошибок, Винни Пух понял, что луч пистолета отражается от зеркал по всем правилам физики. За всякое знание приходится платить: у Пуха остался только один патрон. Помогите Винни попасть в глаз Тутпуханета XVII. Мумия смотрит в одну точку, поэтому в глаз мумии можно попасть только с определенного направления.

Входные данные

В первой строке число K - количество зеркал. Далее даны три пары чисел: (Xp, Yp) - координаты Пуха, (Xm, Ym) - координаты мумии и (a, b) - вектор направления взгляда мумии. Далее, отдельно на каждой строчке, - типы и координаты оставшихся зеркал в формате: тип_зеркала (0 - прямоугольник, 1 - цилиндр), затем

1. *X, Y, R - координаты центра и радиус для цилиндра,*
2. *X1, Y1, X2, Y2 - координаты начала и конца для прямоугольника.*

Во входном файле все числа целые.

Выходные данные

В случае, если поразить глаз мумии можно - вывести сообщение "YES", иначе - вывести сообщение "NO" (без кавычек).

Решение задачи

Задача “сможет ли Пух попасть в глаз Мумии вдоль направления ее взгляда” эквивалентна задаче “видит ли Мумия пистолет Пуха”. Необходимо проследить за направлением взгляда Мумии.

Это типичная задача на поворотную геометрию. Действительно, система зеркал приводит к необходимости построения отражений от прямой и окружности. Для решения этой задачи можно вывести сложные формулы. Однако, существенно проще построить луч отражения при условии, что нормаль к поверхности в точке пересечения луча с поверхностью лежит на одной из осей (пример представлен на рисунке). Отраженный луч

будет проходить через точку (x,-y) для оси x или (-x,y) для оси y, если падающий луч проходил через точку (x,y). Очевидно что воспользовавшись формулами поворотной геометрии, можно изменить систему координат так, что найти луч отражения в новой системе будет просто. Совершив обратный поворот мы найдем направление луча в исходной системе координат.

Таким образом, задача решается следующим образом.

Находим точку пересечения луча с ближайшим объектом. Им может оказаться пистолет Пуха (тогда ответ “да”), зеркальная поверхность (тогда необходимо найти луч отражения повторить действия описанные в этом абзаце с ним) и может не существовать ни одного объекта с которым пересекается луч(тогда ответ “нет”).

МИНИМИЗАЦИЯ ВРЕМЕННОЙ СЛОЖНОСТИ АЛГОРИТМОВ ДЛЯ ЗАДАЧ БОЛЬШОЙ РАЗМЕРНОСТИ

Сикерин А.В. – студент гр. ПОВТ–42

Минимизация временной сложности алгоритмов достаточно важная проблема , возникающая при решении различного рода задач. Зачастую, разработанный алгоритм получает абсолютно верные результаты, однако при повышении размера входных данных, разработанная программа уже не укладывается во время, отведенное для ее работы. Перед программистом встает очередная проблема – оптимизация разработанной программы. Это процесс долгий и трудоемкий. К тому же для любой программы существует вполне конкретный предел оптимизации – это временная сложность алгоритма, положенная в основу программы. Поэтому существуют случаи, когда как бы тщательно не занимался программист процессом оптимизации, программа все равно не будет удовлетворять требованиям, определенным в спецификации ПО. В этом случае требуется разработка нового алгоритма, его реализация, тестирование, и т.д. Чтобы избежать подобных накладок, необходимо заранее анализировать требования быстродействия, предъявляемые к программе, и временную сложность разработанного алгоритма.

Рассмотрим данную проблему на примерах решения задач, представленных на олимпиаде «Университеты Алтая-2006» (http://neerc.secna.ru/ALTAI_U) под названиями «Монеты» (Задача G) и «Замок» (Задача I).

Задача G «Монеты»

Сформулируем математическую модель данной задачи:

Пусть $\exists S = \{a_1 : 1, a_2 : 2, \dots a_n : N\}$, т.е. S – набор, состоящий из a_1– единичек, a_2 – двоек ,…, a_n– n–ок, и $M = \sum_1^n a_i$ – количество элементов набора S.

Рассмотрим все перестановки набора S. Необходимо подсчитать количество перестановок, удовлетворяющих следующему условию

$$(b_1, b_2, \dots, b_m) : \forall j = 1..N \text{ количество } (b_i = j) = a_j \text{ и } \forall j : b_j = i\ (i <> 1)\ \exists k : (j > k) \& (b_k = i - 1)$$

Первое условие – данная перестановка является перестановкой набора S. Второе условие - для любого не равного единице элемента перестановки найдется в данной перестановке элемент со значением на единицу меньше, причем этот элемент стоит левее данного.

Напрашивается интуитивно понятное решение: перебрать все перестановки и посчитать количество перестановок, подходящих под условие. Однако размер входных данных ($M \le 150$) приводит к необходимости рассмотреть все перестановки из чисел от 1 до 150. В качестве примера возьмем входные данные такого вида: N = 150, a_i = 1. При этом придется перебрать все перестановки, но под ответ подходит только одна из них {1, 2, …, 150}.

Временная сложность данного алгоритма составит порядка $n!$ при любой оптимизации, что неприемлемо. Таким образом, реализация подобного алгоритма не приведет к решению поставленной задачи. Аналогично можно показать, что если мы будем генерировать только все перестановки, подходящие под условие, то мы так же получим неэффективный алгоритм. Значит, необходимо спроектировать алгоритм, который подсчитывает количество перестановок, без генерации самих перестановок. Для этого следует сделать следующее замечание.

Возьмем произвольную перестановку набора $S = (b_1, b_2, \dots, b_m)$. Рассмотрим множество позиций одного значения: $C_z = \{b_{i_1}, b_{i_2}, \dots, b_{i_l}\}\ \forall j = 1..l\ \ b_{i_j} = z\ \ \& i_1 < i_2 < \dots < i_z$. Тогда, если для самого левого элемента данного множества (b_{i_1}) найдется левее элемент со значением

z–1 (т.е. для b_{i_1} выполняется условие задачи), то для всех элементов множества C_z условие задачи так же выполняется (т.к. они стоят правее относительно b_{i_1}).

Это замечание и позволяет нам решить данную задачу. Будет считать количество перестановок, удовлетворяющих условию. Посмотрим, что стоит на первой позиции перестановки. Там может стоять только 1. В противном случае для данного элемента со значением z (z>1) не нашлось бы левее элемента со значением z-1. Но если 1 стоит на первой позиции, то для всех элементов множества C_2 условие задачи выполнено. Поэтому все остальные единицы можем расставить произвольно. Из курса комбинаторики известно, что это можно сделать $C_{m-1}^{a_1-1}$ способами.

Теперь осталось расставить элементы 2…N, на M-a_1 позициях. Решение последней задачи эквивалентно решению исходной задачи при $N = N$–1, $a_1 = a_2$, $a_2 = a_3$ … $a_{N-1} = a_N$. Действительно, достаточно перенумеровать элементы 2 в 1 , 3 в 2, … N в N-1. Таким образом получили рекуррентную формулу

$$ans(N, a_1, a_2, \ldots, a_n) = C_{m-1}^{a_1-1} \cdot ans(N-1, a_2, a_3, \ldots, a_n)$$
$$ans(1, a_1) = 1$$
$$m = \sum_1^n a_i$$

Если упростить данное выражение то получим следующее

$$ans = C_{m-1}^{a_1-1} * C_{m-a_1-1}^{a_2-1} * \ldots * C_{m-a_1-a_2-\ldots-a_{n-1}-1}^{a_n-1}.$$

Это выражение и является решением данной задачи. Следует отметить, что биноминальные коэффициенты можно подсчитать используя следующее равенство: $C_n^k = C_{n-1}^{k-1} + C_{n-1}^k$ (треугольник Паскаля). Таким образом, временная сложность предподсчета составит порядка O(m^2)*k действий, где k – действия с длинной арифметикой, подсчет ответа на задачу составит O(N)*k действий – это вполне приемлемый результат.

Задача I «Замок»

Математически эту задачу можно сформулировать следующим образом

Задана система уравнений вида

$$\begin{cases} x_1 + x_2 + \ldots + x_k = y_1 \\ x_2 + x_3 + \ldots + x_{k+1} = y_2 \\ \ldots \\ x_r + x_{r+1} + \ldots + x_n + x_1 + \ldots + x_{k-n+r-1} = y_r \\ \ldots \\ x_n + x_1 + \ldots + x_{k-1} = y_n \end{cases}$$

Необходимо в случае единственного решения вывести $x_1, x_2, \ldots x_n$. Иначе вывести "Impossible".

Данная система уравнений целиком и полностью решается методом Гаусса. Однако взглянем на размер исходных данных: $1 \le N \le 1000$. Учитывая, то, что метод Гаусса работает за O(n^3), где n- количество переменных, мы получаем очень медленный алгоритм решения. Необходимо разработать алгоритм решения с временной сложностью меньшей чем O(n^3). Сделать это поможет тот факт, что матрица системы линейных уравнений не произвольная, а строго определенного вида.

Если вычесть из 1-го уравнения 2-ое уравнение, мы получим выражение вида: $x_1 - x_{k+1} = y_1 - y_{k+1}$

Теперь вычтем из (k+1) уравнения (k+2) первое (если его нет, то перейдем по циклу на k+2 - n) мы получим: $x_{k+1} - x_{2k+1} = y_{k+1} - y_{2k+1}$. Не более, чем через N операций мы получим выражение вида: $x_i - x_1 = y_i - y_1$. Используя полученные равенства можно выразить все неизвестные, которые входят в полученный набор, через переменную x_1. Назовем x_1 независимой переменной.

Если какая-либо переменная x_i из набора $\{x_1, x_2, \ldots, x_n\}$ не выразилась через x_1, то описанный выше процесс можно применить, начиная с данной переменной. Процесс продолжается, до тех пор, пока все переменные не будут выражены через независимые. Далее рассмотрим равенство $x_1 + x_2 + \ldots + x_k = y_1$. Каждое x_i можно выразить через некоторую независимую переменную. Если в данном равенстве будет более одной независимой переменной, это означает, что у исходной системы существует бесконечное множество решений. Следует заметить, что данная ситуация возможна тогда и только тогда , когда число независимых переменных более одной. Дополнительно можно доказать следующее утверждение: система уравнений имеет единственное решение тогда и только тогда когда НОД(n, k) = 1. Если в данном равенстве только одна независимая переменная, то приведем подобные. Теперь если полученное равенство имеет единственное решение, то и система имеет единственное решение. Чтобы его получить достаточно подсчитать x_1, а затем и x_2, x_3, …,x_n.

Оценим временную сложность данного алгоритма. Алгоритм имеет два этапа: процесс получения независимых переменных, процесса получения решения. Оба этих этапа работают за O(n) , т.к. каждую переменную мы рассматриваем ровно один раз. Таким образом, мы понизили сложность алгоритма с O(n^3) до O(n).

В заключении хотелось бы отметить, что на сегодняшний день существуют стандартные классы, процедуры, функции, шаблоны, которые решают очень большое множество задач. Использование данных элементов существенно облегчает процесс программирования, избавляет от необходимости думать, ускоряет время создания программ. Однако не всегда и не везде следует применять стандартные элементы. Рассмотренные выше примеры убедительно доказывают это.