Всероссийская олимпиада школьников по информатике 2017/2018
Региональный этап, I тур
Разбор задач

подготовил: Андрианов И.А.,
Вологодский государственный университет

Вологда
2018 г.

Задача 1 - «Улучшение успеваемости»

Краткое условие:
Ученик получил a двоек, b троек, c четвёрок.
Сколько пятёрок ему надо получить, чтобы среднее арифметическое оценок стало >= 3.5?

Решение – провести ряд несложных преобразований и получить формулу для ответа:

$$(2a+3b+4c+5x) / (a+b+c+x) >= 3.5$$
$$(2a+3b+4c+5x) >= 3.5 * (a+b+c+x)$$
$$4a+6b+8c+10x >= 7a + 7b + 7c + 7x$$
$$3x >= 3a + b - c$$
$$x >= (3a + b - c) / 3$$

Отсюда:

x = (3a + b - c + 2) div 3

Задача 2 - «Квадраты и кубы»

Краткое условие:
Найти количество пар натуральных чисел x и y, таких, что x^2 и y^3 лежат в интервале от *a* до *b* включительно и при этом $|x^2–y^3| \le k$.
Ограничения: *a*, *b*, *k* до 10^{18}.

Решение:

- Перебираем все числа *y*, кубы которых попадают в интервал [a, b]
- Для каждого такого *y* берём интервал max(y^3-k,a) – min(y^3+k, b). Осталось узнать, сколько на этом интервале точных квадратов.
- Количество квадратов на интервале от p до q равно squares(q) – squares(p-1), где функция squares(x) возвращает количество точных квадратов от 0 до x. Её несложно реализовать с помощью операции sqrt.

Сложность алгоритма - $O(b^{1/3})$.

Пример реализации на C++:

```cpp
#include <iostream>
#include <cmath>

long long squares(long long q) {
  long long res = std::sqrt(q);
  if (res * res > q) res--;
  return res;
}

int main() {
 long long a, b, k, ans = 0, y;
 std::cin >> a >> b >> k;
 for (y = 1; y * y * y < a; y++);
 for (; y * y * y <= b; y++) {
   long long left = std::max(a, y * y * y - k);
   long long right = std::min(b, y * y * y + k);
   ans += squares(right) - squares(left - 1);
 }
 std::cout << ans;
}
```

Задача 3 - «Лифт»

Краткое условие:

В здании *m* этажей и *n* сотрудников. Сотрудники с разных этажей в разные моменты времени вызывают лифт, и все они хотят уехать на первый этаж.

Лифт работает так:

Если он приехал на первый этаж, и нет вызовов, то он ждёт вызова. Если есть вызовы, то он поедет к самому раннему из них (а если таких несколько – то к вызову сотрудника с меньшим номером).

На обратном пути лифт будет останавливаться на всех этажах, где к моменту его проезда есть ждущие сотрудники.

Нужно: для каждого сотрудника определить номер секунды, в которую он окажется на первом этаже.

Ограничения: n до 10^5, m до 10^9, моменты времени до 10^9.

Решение: аккуратно промоделируем работу лифта, используя подходящие структуры данных для нужной производительности

Структуры данных:

- массив *come* – сотрудники в порядке их прихода на этаж (а для одинакового времени – по возрастанию номеров)

- булевский массив *out* – какие сотрудники уже уехали

- ассоциативный массив *wait*, который для каждого этажа хранит список сотрудников, которые ждут лифта на данный момент времени (*не включая тех, что уехали, и тех, что ещё не подошли*)

Алгоритм:

В цикле делаем следующее:

Находим следующий активный этаж и обрабатываем очередной цикл лифта вверх-вниз:

- из *wait* извлекаем всех, кто ждёт на этажах не выше активного, отмечаем их как уехавших, считаем для них ответы
- обрабатываем людей, приходящих в этот интервал времени: закидываем в *wait* тех, кто приходит на этажи выше активного либо кто не успевает на этот рейс. А те, кто успевают на этот рейс, уедут - отмечаем их как уехавших, считаем для них ответы.

Важно: не забыть удалять ключ из *wait*, когда этаж становится пустым

Примечание: следующий активный этаж определяется с помощью двух массивов *come* и *out* и указателя на текущую позицию *cur*, который может только увеличиваться

Пример реализации на C++:

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

std::map<int, std::vector<int> > wait; // сотрудники, ждущие лифта на этажах

int main() {
  int n, m;
  scanf("%d %d", &n, &m);
  std::vector<int> t(n), a(n), come(n);
  std::vector<long long> ans(n);
  std::vector<bool> out(n, false);
  for (int i = 0; i < n; i++) {
    scanf("%d %d", &t[i], &a[i]);
    a[i]--;
    come[i] = i;
  }
  std::sort(come.begin(), come.end(), [&](int i, int j){
    return t[i] < t[j] || t[i] == t[j] && i < j;
  });
```

Окончание на следующем слайде

```
  long long T = 0; // текущее время
  int cur = 0, cur2 = 0;
  while (cur < n) {
    while (cur < n && out[come[cur]]) cur++;
    if (cur == n) break;
    T = std::max(T, (long long)t[come[cur]]);
    int floor = a[come[cur]];
    long long T2 = T + floor * 2;
    // обработаем тех, кто ждет лифта сейчас
    while (!wait.empty() && wait.begin()->first <= floor) {
      for (int w : wait.begin()->second) {
        ans[w] = T2;
        out[w] = true;
      }
      wait.erase(wait.begin());
    }
    // обработаем тех, кто приходит не позже окончания этого рейса
    for (; cur2 < n; cur2++) {
      if (out[come[cur2]]) continue;
      if (t[come[cur2]] > T2) break;
      if (a[come[cur2]] <= floor && t[come[cur2]] <= T2 - a[come[cur2]]) {
        ans[come[cur2]] = T2;
        out[come[cur2]] = true;
      } else { // иначе добавим его в перечень ждущих на соотв-м этаже
        wait[a[come[cur2]]].push_back(come[cur2]);
      }
    }
    T = T2;
  }
  for (int i = 0; i < n; i++)
    printf("%I64d\n", ans[i]);
}
```

Задача 4 - «Мониторинг труб»

Краткое условие:
Задано корневое дерево из n вершин, на каждом ребре которого написана буква, а также множество из m слов. Для каждого слова s_i дана его стоимость w_i.

Требуется покрыть все ребра дерева словами из заданного множества, причем слово покрывает путь вниз по дереву, буквы на котором при прочтении вдоль пути образуют это слово. При этом сумма стоимостей слов должна быть наименьшей.

Ограничения: *n* до 500, *m* до 10^5, суммарная длина слов не превышает 10^6, w_i до 10^9.

Предварительные действия:

Построим матрицу cost, где cost[i][j] - номер самого дешёвого слова, по которому можно от вершины i дойти до вершины j.

Для эффективного построения матрицы cost построим бор над всеми словами.

Затем выполним одновременный обход дерева и бора (переходя по ребру дерева, одновременно проходим по ребру в боре, соответствующего нашей букве).

Построение бора:

```
struct Node {
  int next[26], int parent, spec; // номер спецификации, которая кончается в этой вершине
  Node() : parent(0), spec(0) {
    for (int i = 0; i < 26; i++) next[i] = 0;
  }
};
std::vector<Node> trie(1);

void addToTrie(int node, char *s, int spec_id) {
  while (*s != 0) {
    if (trie[node].next[*s-'a'] == 0) {
      trie.push_back(Node());
      trie[node].next[*s-'a'] = trie.size() - 1;
      trie.back().parent = node;
    }
    node = trie[node].next[*s-'a'];
    s++;
  }
  if (trie[node].spec == 0 || w[trie[node].spec] > w[spec_id]) {
    trie[node].spec = spec_id;
  }
}
```

```
for(int i = 1; i <= m; i++) {
  scanf("%I64d %s", &w[i], s);
  if ((int)strlen(s) > n) continue;
  addToTrie(0, s, i);
}
```

Заполнение матрицы cost:

```
void dfs(int u0, int u, int node) { // вершина в графе, узел в луче
  if (trie[node].spec > 0) {
    cost[u0][u] = trie[node].spec;
  }
  for (int to : g[u]) {
    if (trie[node].next[ch[to] - 'a'] > 0) {
      dfs(u0, to, trie[node].next[ch[to] - 'a']);
    }
  }
}

// заполним матрицу cost
for (int i = 1; i <= n; i++) {
  dfs(i, i, 0);
}
```

Далее построим матрицу cost2, где cost2[i][j] - номер самого дешёвого слова, по которому можно от вершины i или её предка дойти до вершины j. Её можно построить из cost, имея также массив предков par:

```
for (int i = 1; i <= n; i++) {
  // проходим путь до корня
  std::stack<int> path;
  int u = par[i];
  while (u != 0) {
    path.push(u);
    u = par[u];
  }
  // идём обратно по этому пути и поддерживаем лучшую спецификацию,
  int best_spec = 0, best_from = 0;//по которой можно дойти ровно до i
  while (!path.empty()) {
    int u = path.top();
    path.pop();
    cost2[u][i] = cost[u][i];
    if (cost[u][i] > 0) {
      from2[u][i] = u;
    }
    if (cost[u][i] > 0 && (best_spec == 0 || w[cost[u][i]] < w[best_spec])) {
      best_spec = cost[u][i];
      best_from = u;
    }
    if (best_spec > 0 && (cost[u][i] == 0 || w[cost[u][i]] > w[best_spec])) {
      cost2[u][i] = best_spec;
      from2[u][i] = best_from;
    }
  }
}
```

Воспользуемся методом динамического программирования.

Пусть dp[u][v] - минимальная стоимость такого набора путей, что:

- они проходят через все вершины поддерева v
- они все кончаются в поддереве вершины v
- при этом покрыт и весь путь от u до v (u - предок v). *Из предыдущего пункта следует, что покрыт он может быть лишь одним словом*
- разрешается покрыть и какие-то вершины выше u

Если v – лист, то dp[u][v] = w[cost2[u][v]] (где w – стоимости слов)

Иначе: возьмём для каждого ребёнка w вершины v значение d[v][w] и просуммируем эти значения. Тем самым мы получим стоимость покрытия всего поддерева с корнем v. Добавим сюда ещё w[cost2[u][v]] – стоимость покрытия пути от u до v.

Однако, может быть ещё более дешёвый вариант. Возможно, выгоднее будет для одного из детей вместо dp[v][w] взять dp[u][w] – тогда не потребуется прибавлять w[cost2[u][v]].

Особенность реализации *dp*: для каждой вершины v поднимаемся вверх по её предкам *u* до корня дерева.

```
void dfs_dp(int v) {
  visited[v] = true;
  int u = v;
  while (u > 0) {
    if (g[v].size() == 0) { // v - лист
      dp[u][v] = w[cost2[u][v]];
    } else { // v - внутренняя вершина
      // сначала посчитаем dp в детях
      for  (int w : g[v])
        if (!visited[w])
          dfs_dp(w);
      long long sum = 0;
      for (int w : g[v]) {
          sum += dp[v][w];
          if (sum > (1LL << 60)) sum = 1LL << 60;
          };
       // попробуем покрыть от v в поддерево каждого ребёнка и плюс ещё
       dp[u][v] = w[cost2[u][v]] + sum;// путь от u (или его предка) до v
       type[u][v] = 1;
       // попробуем от u не до v вести, а пониже-в поддерево какого-то ребёнка
       for (int w : g[v])
         if (sum - dp[v][w] + dp[u][w] < dp[u][v]) {
           dp[u][v] = sum - dp[v][w] + dp[u][w];
           type[u][v] = 2;
           child[u][v] = w;
         }
    }
```

Такой способ можно реализовать за $O(n^2)$

Другой способ решения

Построим вспомогательный ориентированный граф. Вершины графа будут совпадать с узлами заданного во входном файле дерева, а ориентированные взвешенные ребра построим следующим образом.

Если, начав от вершины *u,* можно пройти вниз по дереву по буквам слова s_i и попасть в вершину *v*, то добавим в граф ребро *u*-*v* с весом w_i. Также для каждой вершины *u* добавим ребро веса 0 в её родителя в исходном дереве p_u. Для эффективной реализации этой части используем бор из слов – см. начало разбора.

Легко убедиться, что минимальная стоимость проверки всех труб равна стоимости минимального ориентированного остова в получившемся графе с корнем в корне исходного дерева.

Для поиска минимального остова можно использовать алгоритм Чу Йонджина и Лю Цзенхонга («алгоритм двух китайцев»). Его время работы в самой простой реализации составляет $O(n^3)$, что достаточно для этой задачи.

Алгоритм описан, например, в книге Романовского «Дискретный анализ»