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

подготовил: Андрианов И.А.,
Вологодский государственный университет

Вологда

2018 г.

Задача 5 - «Удаление чисел»

Краткое условие:

Дан ряд чисел 1, 2, 3, … n.

На каждом шаге из ряда удаляются числа от начала с шагом k

Определить, на котором шагу будет удалено n.

Ограничения: *n* до 10^{18}, k до 100

Пример: n = 13, k = 2.

Ряд вначале и после каждого шага:

1 2 3 4 5 6 7 8 9 10 11 12 13

1 3 5 7 9 11 13

1 5 9 13 – *сейчас n=13 удалится*

1 9

1

Решение:

Пусть *pos* – позиция числа *n*. Вначале *pos* = *n*.

На каждом шаге делаем следующее:

- Если *pos* делится нацело на *k*, то число *n* удалится как раз на этом шаге. Выводим ответ и заканчиваем.
- Иначе *pos* уменьшится на *pos* div *k*, так как столько чисел удалится левее от pos.

```
long long n, k;
std::cin >> n >> k;
long long pos = n;
int step = 1;
while (pos >= k) {
  if (pos % k == 0) {
    std::cout << step;
    return 0;
  }
  pos -= pos / k;
  step++;
}
std::cout << 0;
```

Задача 6 - «Старая книга»

Краткое условие:

- В книге неизвестное число страниц
- Страница может содержать либо текст, либо рисунок
- Первые *k* страниц содержат рисунки
- Сумма номеров страниц с текстом равняется *s*

Найти, какое наименьшее количество рисунков могло быть в книге

Ограничения: *k* до 10^9, *s* до 10^{12}

Решение:
Отдельно решим подзадачи с k=0

Будем подряд ставить страницы с текстом (1, 2, 3…) и считать сумму номеров *sum*, пока она не станет >= *s*. Затем проверим:

- Если *sum* = *s*, то ответ – ноль (рисунков вообще нет)
- Если *sum* > *s*, то ответ – единица (можно заменить ровно одну текстовую страницу на рисунок, чтобы sum уменьшился до s)

Решение при $k > 0$ основано на следующей идее:

Пусть n – число страниц с текстом, (k+z) – с рисунками (где k рисунков стоят в начале, а z - можем двигать). Какую минимальную и максимальную сумму номеров текстовых страниц можно набрать?

- минимальная сумма получится, если все z рисунков сдвинуть в конец. Например, при k=2, z=3, n=4:

1 2 3 4 5 6 7 8 9

Р Р Т Т Т Т Р Р Р   сумма = 3+4+5+6=18

- максимальная сумма получится, если все z рисунков сдвинуть в начало:

1 2 3 4 5 6 7 8 9

Р Р Р Р Р Т Т Т Т   сумма = 6+7+8+9=30

Важно! Можно набрать любую сумму между мин. и макс., сдвигая рисунки влево: перемещение рисунка влево на один шаг (на текстовую страницу) увеличивает сумму на 1:

. . . Т Р . . . → . . . Р Т . . .

Алгоритм:

Перебираем n – количество текстовых страниц.
Интервал перебора – от 1 до условия (k+1)+(k+2)…+(k+n) <= s (пока минимально возможная сумма не превышает s).
Быстро проверять эту формулу можно через сумму арифметической прогрессии.

Для каждого *n* сосчитаем *z* – наименьшее количество дополнительных рисунков, чтобы максимально возможная сумма номеров текстовых страниц стала >= *s*:
(k+z+1)+(k+z+2)+...(k+z+n) >= s.
Воспользовавшись формулой суммы арифметической прогрессии и проведя простые преобразования, получаем:
$z = (2s – n^2 - n - 2*k*n + 2 * n - 1) \text{ div } (2 * n)$

Среди всех просмотренных вариантов запомним наилучший

Число итераций – порядка корня квадратного из *s*.

Пример реализации на C++:

```
long long k, s;
std::cin >> k >> s;
if (k == 0) {
  long long sum = 0;
  for (int i = 1; sum < s; sum += i, i++);
  std::cout << (sum > s ? 1 : 0);
} else {
  long long ans = s - 1;
  // перебираем, сколько будет страниц с текстом
  for (long long n = 1; (k + 1 + k + n) * n <= 2 * s; n++) {
    long long z = (2*s - n*n - n - 2*k*n + 2 * n - 1) / (2 * n);
    ans = std::min(ans, k + z);
  }
  std::cout << ans;
}
```

Дополнительные замечания:

Авторское решение этой задачи перебирает z – количество рисунков. Это неправильный подход, так как z может быть достаточно большим.
Например, для теста 1000000000 2000000000 ответом будет 1999999999, т.е. z = 999999999.

Однако, такое решение проходит все официальные тесты.

Задача 7 - «Красота фейерверка»

Краткое условие:

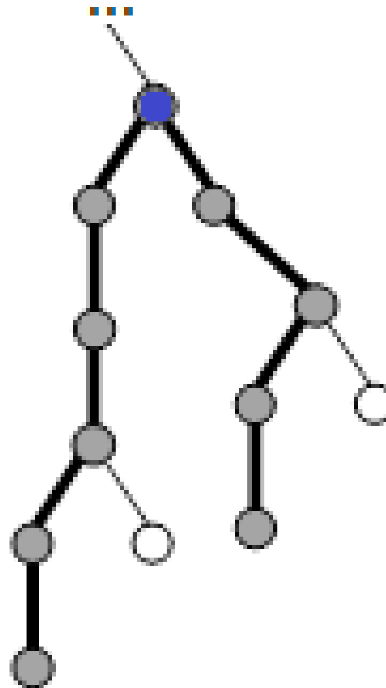
- Дано корневое дерево с n вершинами
- Вводится операция возведения в степень m - к каждому листу дерева подвесить за корень копию изначального дерева, повторить этот процесс (m-1) раз.

Найти самый длинный путь в результирующем дереве (то есть диаметр дерева).

Ограничения: *n* и *m* до 200 000.

Решение:

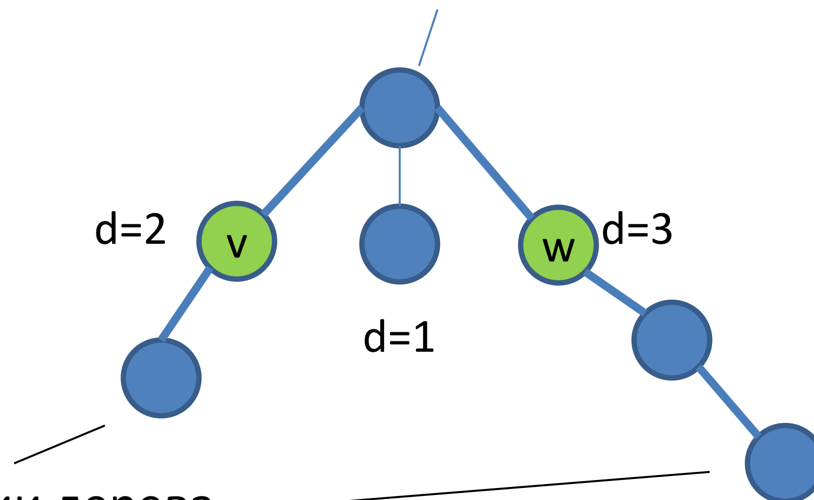
Результирующий путь выглядит так: начинается в некотором листе, вначале идёт вверх, в некоторой вершине «ломается» и идёт вниз до другого листа:



Несложно понять, что вершина перелома пути будет лежать в изначальном дереве (ещё до возведения его в степень). Однако, она не обязана быть его корнем.

Алгоритм: Построим вначале вспомогательный массив *d*, где *d*[*u*] – это расстояние от вершины исходного дерева *u* до самого дальнего листа в поддереве с корнем в *u*. Это можно сделать одним обходом в глубину.

Будем перебирать все внутренние вершины исходного дерева, имеющие хотя бы двух детей – это кандидаты на вершину перелома.
Возьмём два сына v и w с максимальным значением *d* (то есть первый и второй максимум). Оптимальный путь идёт от самого дальнего листа в поддереве v к точке перелома и спускается к самому дальнему листу в поддереве w.



Сюда цеплять копии дерева

При возведении в степень к этим двум листьям будем прицеплено по копии исходного дерева (а в них самый длинный путь до листа равен d[1]). К этим листьям снова будет прицеплено по копии исходного дерева, и так далее.

При этом длина искомого пути определится так:

$$d[v] + d[w] + 1 + 2 * (m-1) * d[1]$$

Пример реализации. Первая часть - заполнение массива d обходом в глубину:

```
std::vector<std::vector<int> > g;
std::vector<int> d;

// находим для каждой вершины, насколько длинный путь существует от неё вниз
void dfs(int u, int prev) {
  for (int v: g[u]) {
    if (v != prev) {
      dfs(v, u);
      d[u] = std::max(d[u], d[v] + 1);
    }
  }
  // отсортируем детей в списке смежности по убыванию глубины
  std::sort(g[u].begin(), g[u].end(), [&](int i, int j) {
    return d[i] > d[j];
  });
}
```

Оставшаяся часть реализации:

```
int main() {
  int n, m;
  scanf("%d %d", &n, &m);
  g.resize(n + 1);
  for (int i = 2; i <= n; i++) {
    int p;
    scanf("%d", &p);
    g[p].push_back(i);
  }
  d.resize(n + 1, 1);
  dfs(1, -1);
  long long ans = 0;
  // попробуем каждую нелистовую вершину сделать корнем
  for (int r = 1; r <= n; r++)
    if (g[r].size() > 1)
      ans = std::max(ans, 1 + d[g[r][0]] + 2LL * (m - 1) * d[1] + d[g[r][1]]);
  std::cout << ans;
}
```

Задача 8 - «Обработка больших данных»

Краткое условие:

- Дан массив с индексами от 0 до 2^k-1, вначале заполненный нулями
- Его надо заполнить подряд так: c_1 значений v_1, потом c_2 значений v_2, …, c_n значений v_n (сумма всех c_i равна 2^k).
- За одну операцию STORE можно заполнить любым значением любой отрезок [L, R] такой, что его длина = 2^i (для какого-то *i*), и при этом L делится на 2^i.

Определить наименьшее число операций STORE, чтобы заполнить весь массив нужными значениями

Ограничения: *k* до 30, n до 10^5, v_i до 10^9.

Решение. Структура корректных отрезков, описанная в условии, напоминает дерево отрезков.
Мысленно построим такое дерево отрезков на числах от 0 до 2^k-1, при этом не будем дальше разбивать те вершины, которые соответствуют отрезкам, целиком находящимся во входных отрезках c_i (поскольку ясно, что вызывать STORE для более мелких отрезков смысла нет).
Примечание: явно строить такое дерево не обязательно.

Утверждение. Пусть вершина u дерева соответствует отрезку [L, R]. Чтобы заполнить вест отрезок [L,R] нужными числами, можно в качестве первой операции выполнить STORE для **всего** отрезка [L,R] (а потом уже делать STORE для каких-то подотрезков). Какое именно значение писать первой операцией STORE - отдельный вопрос.

Пример: из 00000000 надо получить 22111111. Можно так: 00000000 → 11111111 → 22111111 (первым шагом заполнили весь отрезок). *Доказать можно по индукции*.

Используем динамическое программирование на дереве.

Пусть opt[u][c] - минимальное кол-во операций STORE, которое нужно, чтобы заполнить отрезок ячейки от L до R нужными значениями, если первой операцией будет STORE для всего отрезка [L, R] и значения c.

1. Если у вершины *u* нет детей, то opt[u][c] = 1 либо 2 (в зависимости от того, c – это нужное значение, или нет)

2. Пусть у вершины *u* есть дети *v* и w. Тогда opt[u][c] - это минимум среди:

- opt[v][c] + opt[w][c] - 1 (здесь -1=-2+1, потому что теперь для детей не нужно первой операцией заполнять их значениями *c* (два STORE в детях ушло, один в родителе добавился)
- min (opt[v][c] + opt[w][d]) по всем возможным значениями d. Здесь для *v* убирается первый STORE, так как его заменяет STORE для *u*. А правый подотрезок, возможно, вначале выгодно целиком заполнить значением d.
- min (opt[v][d] + opt[w][c]) по всевозможным d - аналогичноv

Такое решение работает за O($nm^2\log n$) и способно пройти подзадачи 1, 2, 3.

Заметим, что min (opt[v][c] + opt[w][d]) = opt[v][c] + min(opt[w][d]) – то есть перебирать d можно один раз, а не для каждого *c*. Получаем решение за O($nm \log n$), которое проходит подзадачи 1, 2, 3 и 4.

Для полного решения заметим, что величины opt[u][c] для разных *c* отличаются не больше чем на 1.

Действительно, пусть $opt[u][c_2] = t$. Значит, если первым действием мы заполняем весь отрезок, соответствующий вершине u, значениями c_2, то все значения могут быть получены за t операций. Но тогда если первым действием заполнить весь отрезок значениями c_1, а затем применить последовательность операций из оптимального решения для c_2, мы получим решение за $t + 1$ операцию.

Следовательно, вместо хранения значений *opt*[*u*][*c*] для каждой вершины и каждого возможного значения *c*, достаточно хранить:

- *best*[*u*], равное минимуму *opt*[*u*][*c*] по всем значениям *c*
- множество значений *bestv* [*u*] = {c_1, c_2, …, c_s}, для которых *opt*[*u*][c_i] минимально и равно *best*[*u*] (для остальных значений *d* значение *opt*[*u*][*d*] равно *best*[*u*] + 1).

● Если вершина *u* является листом, который должен быть заполнен значениями *c*, то *best*[*u*] = 1, *bestv*[*u*] = {*c*}.

● Иначе пусть *v* и *w* — дети *u*.

- Если есть значение *c*, которое входит одновременно в *bestv*[*v*] и *bestv*[*w*], то *best*[*u*] = *best*[*v*] + *best*[*w*] – 1, *bestv*[*u*] = *bestv*[*v*] ∩ *bestv*[*w*]
- Если же *bestv*[*v*] и *bestv*[*w*] не пересекаются, то

  best[*u*] = *best*[*v*] + *best*[*w*], *bestv*[*u*] = *bestv*[*v*] ∪ *bestv*[*w*]

Оценка сложности. Каждое значение попадает в *bestv* от листа, и далее может попасть только в *bestv* вершин на пути от листа до корня дерева отрезков, которых O(*k*). Если пересечение и объединение множеств осуществляется за время, пропорциональное их размеру, то время работы решения равно O(*kn* log *n*).

Пример реализации. Начало - объявления и функция main:

```
int k, n, m;
std::vector<int> c, v; // в c - префиксные суммы от исходного c
struct Node {
  int best;
  std::vector<int> bestv;
};


int main() {
  scanf("%d %d %d", &k, &n, &m);
  c.resize(n + 1);
  v.resize(n + 1);
  for (int i = 1; i <= n; i++) {
    scanf("%d %d", &c[i], &v[i]);
    c[i] += c[i - 1];
  }
  Node node = solve(0, (1 << k) - 1);
  std::cout << node.best;
}
```

Оставшаяся часть реализации - функция solve:

```
Node solve(int L, int R) {
  Node u;
  // проверим, лист ли это
  int pos = std::upper_bound(c.begin(), c.end(), L) - 1 - c.begin();
  if (R < c[pos + 1]) { // это лист
    u.best = 1;
    u.bestv.push_back(v[pos + 1]);
  } else { // это не лист
    int M = (L + R) / 2;
    Node v = solve(L, M), w = solve(M + 1, R);
    // пересекаются ли множества v.bestv и w.bestv
    std::set_intersection(v.bestv.begin(), v.bestv.end(), w.bestv.begin(),
                          w.bestv.end(), std::back_inserter(u.bestv));
    if (u.bestv.size() != 0) {
      u.best = v.best + w.best - 1;
    } else {
      u.best = v.best + w.best;
      u.bestv.resize(0);
      std::set_union(v.bestv.begin(), v.bestv.end(), w.bestv.begin(),
                     w.bestv.end(), std::back_inserter(u.bestv));
    }
  }
  return u;
}
```