Всероссийская олимпиада школьников по информатике 2016-17
Региональный этап
Вологодская область
Разбор задач

Подготовили:

Андрианов И.А.

Стрешнев Е. Е.

Вологда

2017 г.

Задача 1 «Кампус»

1) **Вычислим суммарное количество комнат в одном подъезде**

$$p = (n \text{ div } k) * x + (n - (n \text{ div } k)) * y$$

2) **Заменим в каждом запросе числа a_i на $(a_i - 1) \bmod p$. Теперь можно считать, что все комнаты в первом подъезде, и нумерация комнат начинается с 0**

3) **Разобьём все этажи на блоки по k подряд идущих этажей (последний блок может содержать менее k этажей). Каждый такой блок содержит комнат:**

$$b = x + (k - 1) * y$$

4) **Значит, ниже искомой комнаты находится $a_i \text{ div } b$ полных блоков. А в блоке, в котором она находится, ниже неё находится этажей:**

$$\min((a_i \bmod b) \text{ div } y, k - 1)$$

5) **Суммируем значения и получаем ответ**

$$a_i \text{ div } b * k + \min((a_i * b) \text{ div } y, k - 1) + 1$$

1) **Если бы все этажи имели по y комнат, то этаж, на котором находится комната, был бы равен**

$$a_i \text{ div } y$$

Это решение подходит для третьей подзадачи

2) **Для получения полного балла по этой задаче надо не забыть использовать 64-битный тип данных**

Задача 2 - «Калькулятор»

Основная идея

Динамическое программирование

Заметим, что все три описанные в условии операции являются монотонными, то есть для большего значения результат не меньше

1) **Рассмотрим значения dp[*i*][*j*][*k*] – минимальное число, которое можно получить из числа n, нажав на кнопку A *i* раз, на кнопку B - *j* раз и на кнопку C - *k* раз. Тогда dp[0][0][0] = n, а ответ задачи находится в значении dp[a][b][c]**

2) **Для всех значений *i*, *j*, *k* значение dp[*i*][*j*][*k*] позволяет получить числа, которыми можно потенциально улучшить значения dp[*i + 1*][*j*][*k*], dp[*i*][*j + 1*][*k*] и dp[*i*][*j*][*k + 1*], применив операции A, B или C соответственно**

3) **То есть, создаём трёхмерный массив dp[1 + a][1 + b][1 + c], заполняем начальные значения dp[0][0][0] = n. Остальные элементы массива следует заполнить начальными значениями, которые гарантированно не меньше, чем возможное в них значение, так как мы ищем минимум - например, числом n.**

Фрагмент кода на Java

```
for (int i = 0; i <= a; i++) {
    for (int j = 0; j <= b; j++) {
        for (int k = 0; k <= c; k++) {
            if (i < a) {
                dp[i + 1][j][k] = Math.min(dp[i + 1][j][k], dp[i][j][k] / 2);
            }
            if (j < b) {
                dp[i][j + 1][k] = Math.min(dp[i][j + 1][k], (dp[i][j][k] + 1) / 2);
            }
            if (k < c) {
                dp[i][j][k + 1] = Math.min(dp[i][j][k + 1], (dp[i][j][k] - 1) / 2);
            }
        }
    }
}
```

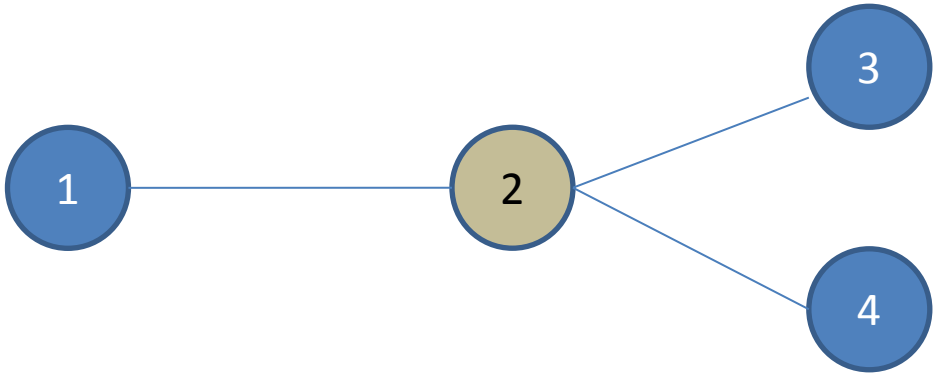
Подзадачи

1) **Для решения подзадачи 1 можно использовать полный перебор вариантов**

2) **Возможно построение жадных алгоритмов, ряд из которых работает лишь при b = 0 или c = 0 и позволяет решить подзадачи 2 и 3**

Задача 3 -
«Размещение данных»

Рассмотрим вначале подзадачу 2 – граф является деревом



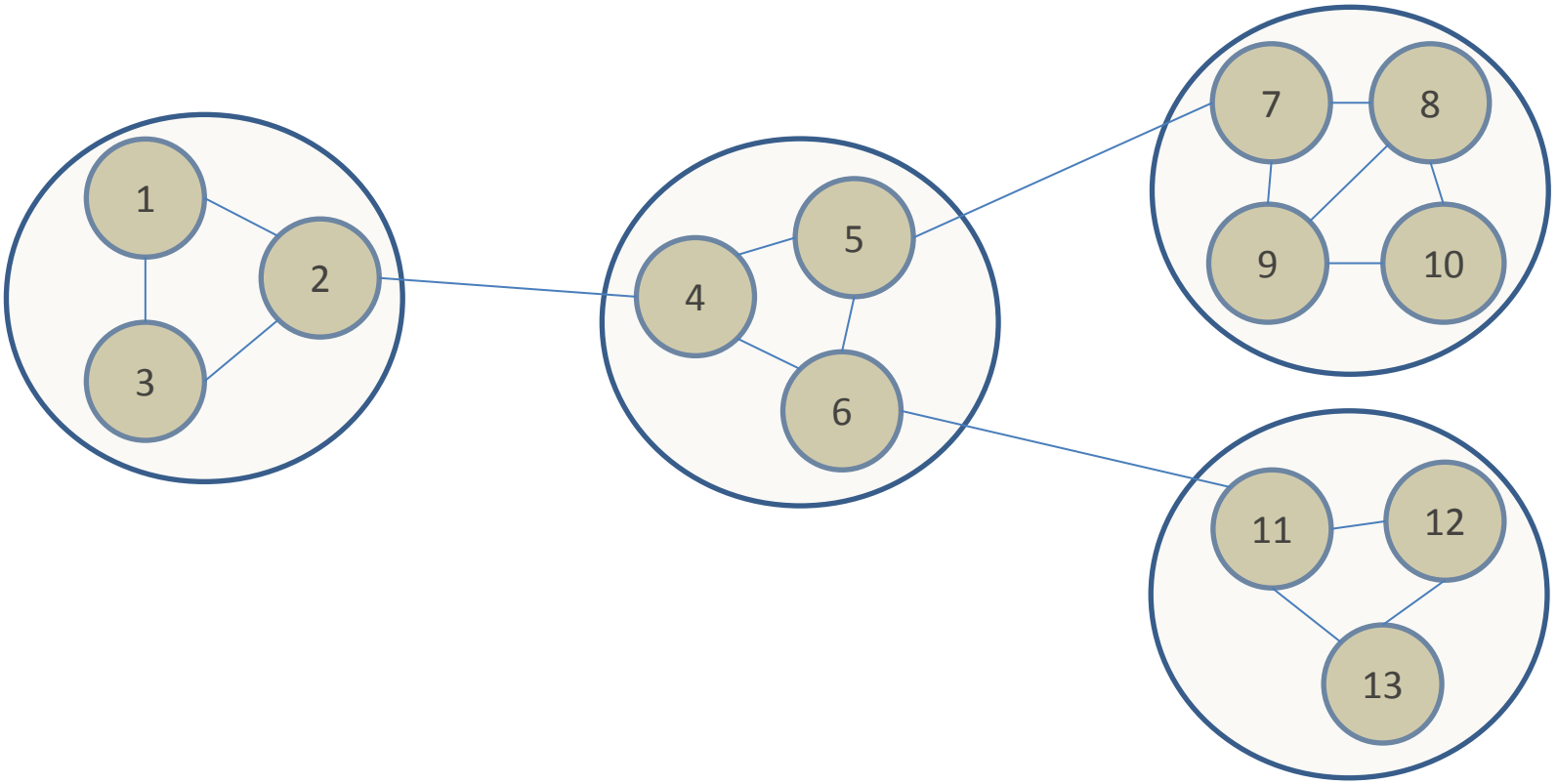
Решение: нужно разместить сервера во всех листьях дерева.

Обоснование: если не поместить сервер в лист, то при удалении ребра, ведущего в лист, пути от этой вершины никуда не будет.

С другой стороны, при удалении любого ребра в обеих получившихся компонентов остаётся хотя бы один лист, в котором стоит сервер

Общий случай – произвольный граф

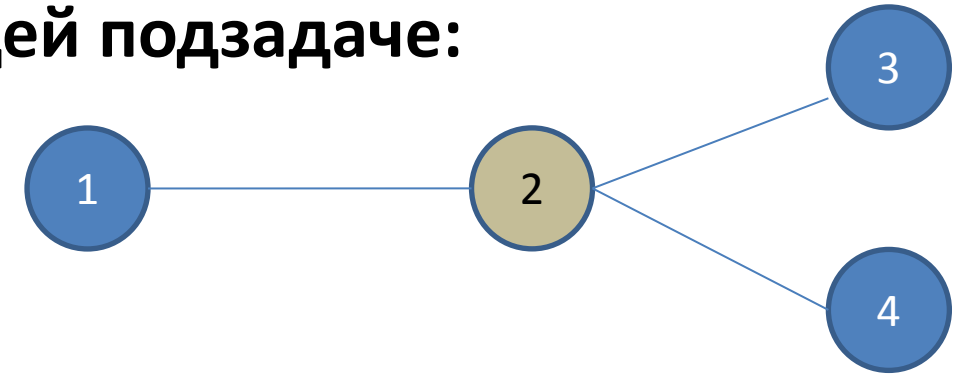
Из всех рёбер графа интерес представляют только мосты – рёбра, при удалении которых граф теряет связность. Граф можно рассматривать как набор двусвязных компонент, соединённых мостами.



Найдём в графе все мосты. Алгоритмы поиска мостов на основе поиску в глубину см. например, в:

- ***Кормен, Лейзерсон, Ривест. «Алгоритмы: Построение и анализ». – М.: МЦНМО, 1999. – 960 с.***
- ***http://e-maxx.ru/algo/bridge_searching***

Сожмём каждую двусвязную компоненту в одну вершину. В результате получим дерево, как в предыдущей подзадаче:



Ответом будет:

k – количество листьев в этом дереве

c – произведение размеров двусвязных компонентов – листьев дерева, взятое по модулю 10^9+7

Оценка вычислительной сложности

Поиск мостов выполняется за O(m), где m – количество рёбер.

Обход двусвязных компонент, определение их размеров и количества входящих мостов также можно сделать за O(m), если хранить мосты в списках смежности (так же, как и все рёбра графа) и использовать метод двух указателей (однонаправленных), идущих по рёбрам, выходящим из вершины, и по мостам, выходящим из неё.

Тогда общая сложность – O(m).

Если хранить мосты, например, в std::set на C++ или TreeSet в Java, то сложность составит $O(m \cdot \log(m))$.

Задача 4 -
«Полезные ископаемые»

Число групп, а также число роботов в последней группе подбираем двоичным поиском

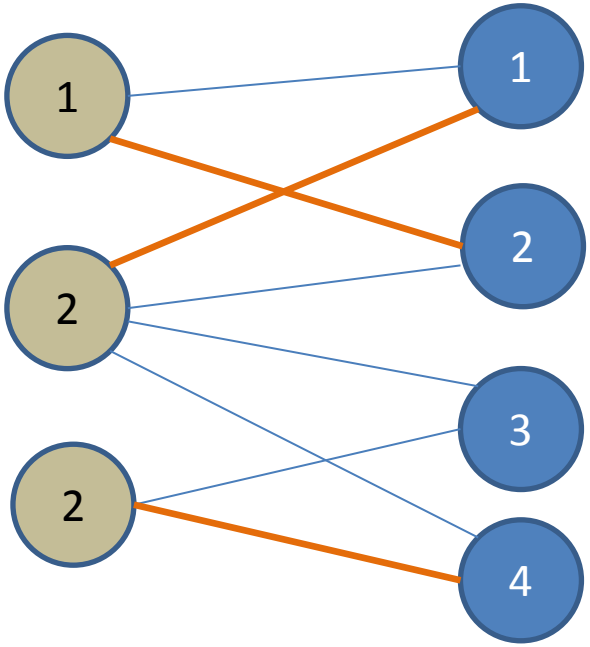
Можно делать в два последовательных этапа:

1. **Двоичным поиском подбираем количество полных групп**

2. **Двоичным поиском подбираем количество роботов в последней (неполной) группе**

Пусть зафиксировано число групп и количество роботов в них.

Рассмотрим сначала подзадачу с q = 1 (в каждой клетке не более одного робота). Построим двудольный граф, где вершины в левой доле – роботы, в правой доле – клетки, рёбра – может ли робот дойти до клетки.



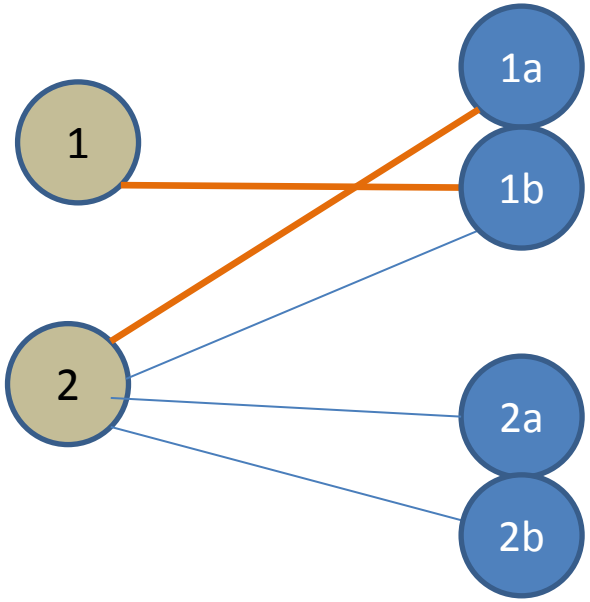
Нужно найти паросочетание, насыщающее левую долю.

Вариант решения – найти максимальное паросочетание (например, алгоритмом Куна)

Такой способ решает подзадачи 1- 3

Рассмотрим теперь случай, когда q > 1.

Он сводится к предыдущему: для каждой клетки в правую долю графа добавим не одну вершину, а q:



Оранжевые рёбра иллюстрируют пример, когда два робота оказались в одной клетке

Проблема: в подзадачах 4-6 количество вершин слишком велико, и алгоритм поиска максимального паросочетания не позволит их решить.

Что делать?

Полное решение – применение леммы Холла

Лемма Холла: рассмотрим двудольный граф с долями X и Y.
Пусть A - множество вершин из доли X, N(A) – множество их соседей (из доли Y).

Утверждается: в графе существует насыщающее долю X паросочетание тогда и только тогда, когда для любого A множество N(A) содержит не меньше вершин, чем A.

Доказательство можно найти в:

- Липский В. Комбинаторика для программистов. – М.: Мир, 1988.

Как применить лемму Холла?

Рассматривать все подмножества роботов не получится – их слишком много.

Заметим, что:

- имеет смысл рассматривать только целые группы роботов на базах *(если для всей группы условие леммы выполняется, то для части - выполняется тем более)*

- если рассматривается группа с мобильностью *m*, то можно сразу включить в подмножество и все группы на той же базе с мобильностью ≤ m, так как это увеличивает размер A, но не меняет размер N(A)
(то есть сразу проверять более строгое условие)

Алгоритм решения

Отсортируем на каждой базе партии роботов по неубыванию мобильности

Будем перебирать варианты количеств первых партий роботов, взятых на каждой базе. Например, если у нас две базы, на первой – 2 партии, на второй – 3, то варианты перебора будут такими:

- 1-я база – ничего, 2-я база – ничего
- 1-я база – ничего, 2-я база – партия 1
- 1-я база – ничего, 2-я база – партии 1, 2
- 1-я база – ничего, 2-я база – партии 1, 2, 3
- 1-я – партия 1, 2-я база – ничего
- 1-я база – партия 1, 2-я база – партия 1
- 1-я база – партия 1, 2-я база – партии 1, 2 и т.д.

Для каждого из вариантов сравним количество роботов и общее количество клеток, в которые они могут добраться.

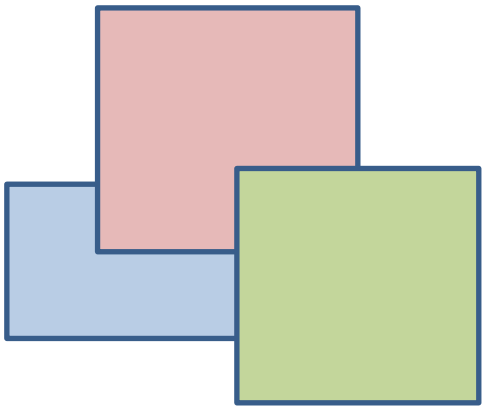
Если количество клеток, умноженное на q, меньше общего количества роботов в выбранных партиях, то мы нашли контрпример к лемме Холла, и распределить роботов нельзя.

Если для всех множеств контрпример не найден, то искомое распределение возможно

Как находить количество клеток, куда могут добраться роботы выбранных партий?

Для каждой базы множество клеток представляет собой прямоугольник, достижимый самой мобильной группой с этой базы.

Нужно найти площадь объединения прямоугольников для всех баз:



Как находить площадь объединения прямоугольников?

Можно использовать формулу включения – исключения: перебираются все подмножества прямоугольников, берётся их пересечение и учитывается со знаком ‘+’, если количество нечётно, и со знаком ‘-’, если количество чётно.

Например, для трёх прямоугольников на C++ это выглядит так:

```
S = r0.place()+r1.place()+r2.place()
-(r0*r1).place()-(r0*r2).place()
-(r1*r2).place()+(r0*r1*r2).place()
```

Здесь place – функция вычисления площади, ‘’ – перегруженный оператор, возвращающий пересечение двух прямоугольников*

Особенности реализации.

Поскольку количество баз ограничено 4, то можно выписать формулу включения-исключения явно для каждого случая, а не реализовывать в общем виде

Оценка сложности:

Пусть на i-й базе g_i групп роботов. Тогда необходимо перебрать $(g_1+1) * (g_2+1)...*(g_s+1)$ вариантов выбора групп роботов, а для каждого варианта - 2^s вариантов в формуле включения-исключения.

Наибольшее количество действий будет при $g_1+g_2+...+g_s=t$. Произведение при фиксированной сумме максимально при равных сомножителях, поэтому получается порядка $O((t/s)^s)$ действий. Это ещё нужно умножить на 2^s и на сложность двоичного поиска, получаем $O((t/s)^s \cdot 2^s \cdot \log(w \cdot h \cdot q))$

Задача 5 – «Автоматизированное управление доставкой»

1) **Заметим, что отправляемый пакет может иметь любой вес от *x* до (*x + k – 1)*. Вес *x + i* может получиться, например, после приёма *(a – 1)* пакета по *1* кг и пакета весом *(i + 1)* кг.**
2) **Рассмотрим максимальное количество пакетов, которое могло быть перевезено перед отправкой контейнера. Это количество *n = y div x*.**
3) **Минимальный суммарный вес этих пакетов равен *n * x*, максимальный равен *n * (x + k – 1)*, причём все промежуточные значения достижимы. Значит, если необходимый для отправки контейнера вес лежит между этими значениями, то мы можем добиться веса контейнера ровно *y*, что и будет являться ответом.**
4) **В противном случае доставка *n* пакетов, даже если они все имеют максимальный вес (a + k – 1), не приводит к отправке контейнера. Значит, для его отправки потребуется минимум *(n + 1)* пакет. Минимальный вес *(n + 1)* пакета равен *(n + 1) * x*.**

Следует использовать 64-битный тип данных при таком решении, так как промежуточные значения могут переполнять 32-битный тип, хотя сам ответ представим в 32-битном типе.

Например, тест:

k = 1000000000
x = 1
y = 1000000000

Ответом будет являться число 1000000000, которое не переполняет 32-битный тип, но при этом в момент вычисления значения максимального суммарного веса пакетов в формуле *n * (x + k – 1)* получается значение 1000000000000000000 ($1*10^{18}$), которое не помещается в 32-битный тип.

Задача 6 -
«Большой линейный коллайдер»

1) **Для удобства рассуждений представим частицы и их положение на прямой как скобочную последовательность. Положительно заряженной частице сопоставим открывающую скобку, отрицательно заряженной – закрывающую. Частицы взаимно уничтожаются тогда и только тогда, когда они соответствуют парным скобкам.**

2) **Найдём для каждой частицы момент времени, когда она будет уничтожена.**

3) **Теперь совместно отсортируем все моменты времени, когда происходит уничтожение частиц, и времена запросов на количество частиц. При этом запросы о количестве частиц при равенстве времени, в соответствии с условием, должны идти после уничтожений.**

4) **После сортировки рассматриваем все моменты времени по очереди. Начальное количество частиц – n. Если встречаем запрос о количестве – выводим текущее количество частиц, если происходит уничтожение в этот момент времени – уменьшаем текущее число частиц на 2.**

1) **Так как все координаты частиц во входных данных идут в строго возрастающем порядке, то можно находить моменты уничтожения частиц параллельно со чтением данных.**

2) **Читаем координату и тип текущей частицы. Если тип соответствует открывающей скобке – помещаем её координату в стек. Если закрывающей, то смотрим, пуст ли стек. Если стек пуст – ничего не делаем, у этой скобки нет парной, то есть частица никогда не уничтожится. Иначе достаём с вершины стека координату парной ей частицы, она будет ближайшей из возможных, и мы точно знаем момент времени их встречи, когда они уничтожатся. Хотя координаты идут в порядке возрастания, не забываем, что могут быть и отрицательные координаты, поэтому момент времени вычисляем, как модуль разности расстояния, делённый пополам, так как частицы движутся с одинаковой скоростью навстречу друг другу.**

$$|x1 - x2| \text{ div } 2$$

Таким образом, мы будем иметь все моменты времени уничтожения пар частиц. Их следует занести в структуру данных, которая содержит два поля – время события и тип события (уничтожение или запрос на количество). Добавлять моменты уничтожения можно также во время чтения.

```
final int DESTRUCTION_EVENT = 1;
final int REQUEST_EVENT = 2;

final int CLOSE_BRACKET = -1;
final int OPEN_BRACKET = 1;

for (int i = 1; i <= n; i++) {
    int x = in.nextInt();
    int type = in.nextInt();
    if (type == OPEN_BRACKET) {
        stack.push(x);
    } else {
        if (!stack.isEmpty()) {
            int match = stack.poll();
            Event event = new Event(Math.abs(x - match) / 2, DESTRUCTION_EVENT);
            events.add(event);
        }
    }
}
```

Далее читаем все моменты времени, в которые приходит запрос на количество частиц, и добавляем в ту же самую структуру, в которой храним информацию об уничтожениях. После чего сортируем все события так, чтобы при совпадении времени уничтожения и запроса на количество запрос шёл позже.

```
class Event implements Comparable<Event> {
    long time;
    int type;

    Event(long time, int type) {
        this.time = time;
        this.type = type;
    }

    @Override
    public int compareTo(Event second) {
        if (this.time != second.time) {
            return (int) (this.time - second.time);
        } else {
            return this.type - second.type;
        }
    }
}
```

Теперь наш список полностью соответствует требованиям. Все события происходят в порядке возрастания их времени и события на запрос в случае совпадения времени стоят после уничтожений, чтобы уничтожающиеся частицы не учитывались.

Начальное количество частиц – n. Если встречаем событие с типом «запрос о количестве» – выводим текущее количество частиц, если происходит уничтожение – уменьшаем текущее число частиц на 2.

```
Collections.sort(events);
int curCount = n;
for (Event event : events) {
    if (event.type == DESTRUCTION_EVENT) {
        curCount -= 2;
    } else {
        out.println(curCount);
    }
}
```

В решении присутствует деление на 2. Чтобы избежать использования вещественных типов, можно умножить все координаты и времена в запросах на 2.

Но при этом имеются тесты, в которых может получиться переполнение 32-битного типа при вычислении модуля расстояния между координатами перед делением пополам.

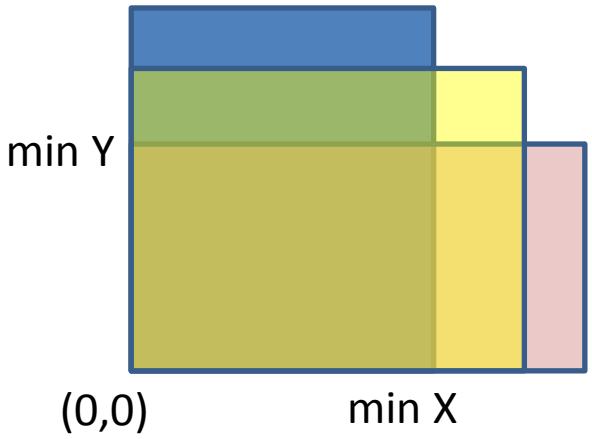
```
Math.abs(x - match) / 2
```

Задача 7 - «Силовые поля»

Нужно выбрать ровно k прямоугольников так, чтобы их пересечение дало максимальную площадь

<u>Вопрос:</u> если даны конкретные k прямоугольников, у которых левый нижний угол в точке (0,0), то как найти их пересечение?

<u>Ответ</u>: взять среди них минимальную координату *X* и минимальную координату Y и перемножить.



Идея решения:

Ясно, что координата *Y* прямоугольника, площадь которого будет в ответе, совпадает с координатой *Y* какого-то исходного прямоугольника.

Будем перебирать все прямоугольники по убыванию координаты *Y* и поддерживать мультимножество *S* из координат *X* всех уже рассмотренных прямоугольников.
Наилучшим выбором *X* для заданного *Y* и для уже рассмотренных прямоугольников будет *K*-й максимум из мультимножества *S*.
Заметим, что прямоугольники с высотой < *Y* на ответ для заданного *Y* не влияют, а с высотой = Y – будут рассмотрены на следующих шагах.

Особенности реализации

Заметим, что для получения k-го максимума в S не нужно хранить X-координаты всех предыдущих прямоугольников – достаточно только K самых больших. Поэтому в качестве структуры данных удобно использовать очередь с приоритетами (в C++ - класс std::priority_queue).
Также можно использовать мультимножество (std::multiset) и некоторые другие структуры.

Сложность - $O(n \cdot \log(n))$

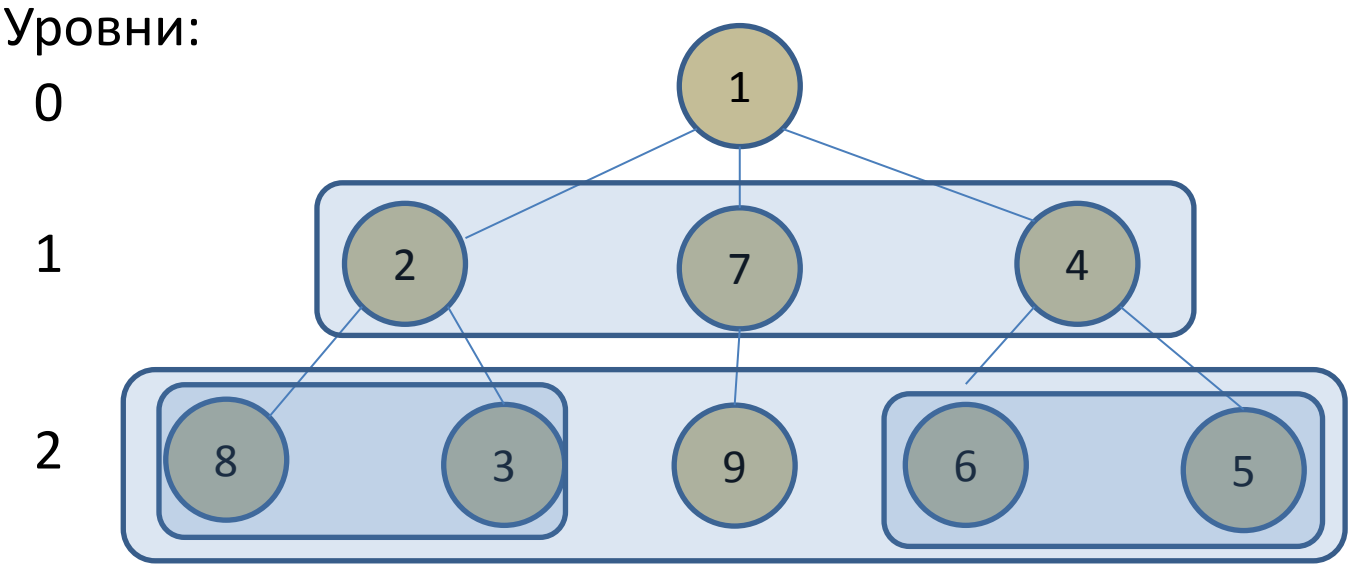
Полный текст решения:

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

struct Rect {
  int y, x;
  bool operator < (const Rect &p) const {
    return y > p.y;
  }
};

int main() {
  freopen("power.in", "r", stdin);
  freopen("power.out", "w", stdout);
  int n, k;
  scanf("%d %d", &n, &k);
  std::vector<Rect> rects(n);
  for (int i = 0; i < n; i++)
    scanf("%d %d", &rects[i].x, &rects[i].y);
  std::sort(rects.begin(), rects.end());
  long long answer = -1;
  std::priority_queue<int, std::vector<int>, std::greater<int> > q;
  for (Rect r : rects) {
      q.push(r.x);
    if ((int)q.size() > k)
      q.pop();
    if ((int)q.size() == k)
      answer = std::max(answer, r.y * (long long)q.top());
  }
  printf("%I64d", answer);
}
```

Задача 8 -
«Повышение квалификации»

Формализация задачи:

Дано дерево с корнем. Каждая заявка определяет интервал на одном из уровней дерева. Интервалы не пересекаются, но могут быть вложены один в другой. Нужно найти отрезок [L, R] минимальной длины, чтобы в каждом интервале нашелся хотя бы один номер вершины, принадлежащий [L, R].



Овалами показаны заявки:
(1, 1), (1,2),
(2, 1), (4,1)
Ответ: отрезок 3-5

Если один интервал содержится в другом, то внешний можно удалить. В итоге останутся интервалы без общих вершин. Тогда их суммарная длина не превышает n.

Задача свелась к следующей: дано несколько непересекающихся множеств целых чисел. Найти отрезок [*L*, *R*] наименьшей длины, содержащий хотя бы по одному элементу каждого множества.

Как это сделать?

1. Создадим вектор *w*, в котором каждый элемент – это пара чисел <v, from>, где v – значение, from – из какого множества.

2. Отсортируем вектор w по возрастанию v.

3. Воспользуемся методом двух однонаправленных указателей. Пусть *l* – индекс начала интервала в векторе w, *r* – индекс конца интервала (левый и правый указатели).
Создадим вектор *c*, где *c*[*i*] – это количество элементов из множества с номером *i* среди w[*l*], w[*l*+1],…, w[*r*].
Пусть *total* – это количество разных множеств, к которым относятся элементы w[*l*], w[*l*+1],…,w[*r*].

Двигая правый указатель, мы будем увеличивать значения в *c*. Если перед увеличением в каком-то элементе был 0, то увеличиваем total.
Двигая правый указатель, делаем наоборот – уменьшаем значения в *c* и, возможно, *total*.

Примечание: в авторском решении использовался несколько другой способ

Рассмотрим пример (с рисунка выше). Имеется 3 множества:
S1={2, 7, 4}, S2 = {8, 3}, S3 = {6, 5}
Вектор *w* будет выглядеть так (значение, из какого множества):

0	1	2	3	4	5	6
2,1	3,2	4,1	5,3	6,3	7,1	8,2
	l=1		*r=3*			

Пусть указатели *l* и *r* стоят как на рисунке. Тогда *total* = 3 (задействованы три разных множества), в каждом множестве использовано по одному элементу, то есть *c*[1]=*c*[2]=*c*[3]=1.

Если двинуть левый указатель на шаг вправо, то *c*[2] обратится в ноль, а *total* станет равен 2.

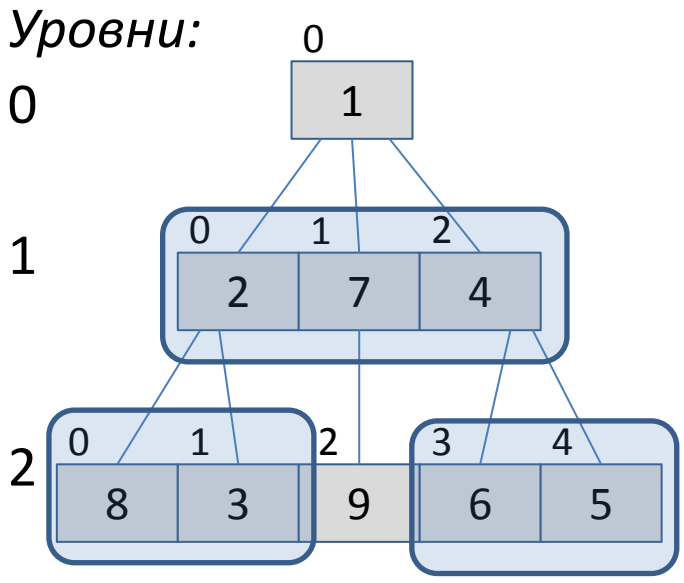
Если затем двинуть правый указатель на шаг вправо, то *c*[3] увеличится до двух, но *total* не изменится.

Сложность определяется сортировкой массива *q*, сам проход двумя указателями делается за линейное время.

```
for (;i <= j;) {
  // двигаем второй указатель, пока не удовлетворим все заявки
  for (;;) {
    if (total == rangesCount) break; // заявки удовлетворились
    j++;
    if (j >= (int) w.size()) break;
    if (c[w[j].second] == 0) total++;
    c[w[j].second]++;
  }
  if (j >= (int)w.size()) break;
  if (w[j].first - w[i].first < R - L) {
    L = w[i].first;
    R = w[j].first;
  }
  // двигаем первый указатель
  if (c[w[i].second] == 1) {
    total--;
  }
  c[w[i].second]--;
  i++;
}
```

Как эффективно построить интервалы для заявок, отбросив при этом лишние (внутри которых содержатся более мелкие)?

Все вершины на каждом уровне дерева удобно занести в отдельный вектор для этого уровня (в порядке их посещения при обходе в ширину или глубину). Тогда каждый интервал можно задавать всего двумя числами – позициями начала и конца в векторе.



Пояснение к рисунку:

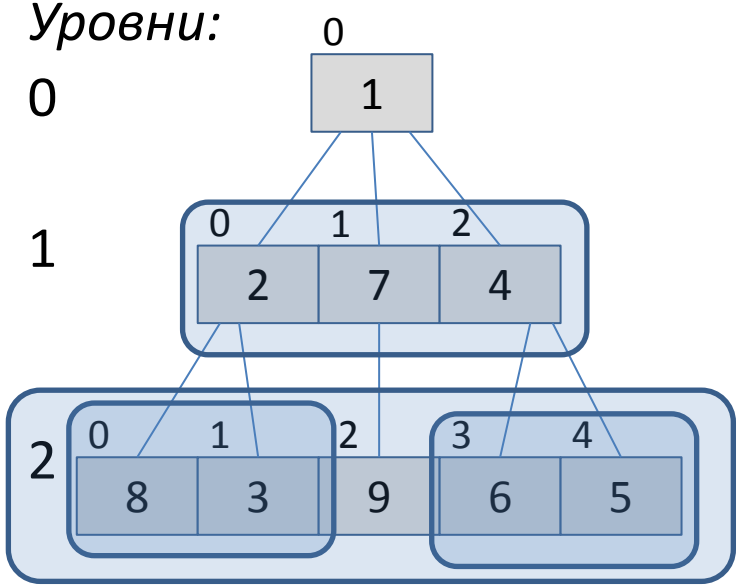
На каждом уровне создан вектор, куда занесены все вершины в порядке обхода. Убран лишний интервал. Оставшиеся интервалы задаются парой позиций в векторе своего уровня:

Уровень 1: [0, 2]

Уровень 2: [0, 1], [3, 4]

Как получить левую и правую позиции интервалов для заявок, а также выбросить лишние?
Воспользуемся обходом в глубину.

- Для каждого уровня в дереве будем хранить пару чисел <*first*, *second*> - индекс самой левой и самой правой вершины, которую мы посетили на этом уровне, а также признак *req*, что какая-то вершина, куда мы зашли и ещё не вышли, имеет заявку, интервал которой приходится как раз на этот уровень
- Зайдя в очередную вершину, обновим значения *first* и *second* для уровня, на котором она лежит
- Затем пройдём по всем заявкам вершины. Для каждой из них на соответствующем уровне поставим *first*=∞, *second*=-1.
- Рекурсивно запустим поиск в глубину для всех детей.
- После этого ответы для заявок из нашей вершины должны быть сформированы. Если для какой-то заявки ответа нет, то, значит, была найден ответ для заявки, интервал которой вложен в наш. Для уровня каждой заявки ставим *req=false*, чтобы отменить вышестоящие заявки.



Овалами показаны заявки:
(1, 1), (1,2), (2, 1), (4,1)

- Вначале мы зашли в вершину 1. Заявка (1,1) говорит о том, что нужно поставить req[1] = true, first[1] = ∞, last[1] = -1. Заявка (1,2) говорит о том, что нужно поставить req[2] = true, first[2] = ∞, last[2] = -1.
- Далее пошли в вершину 2. Заявка (2,1) говорит о том, что нужно поставить req[2] = true, first[2] = ∞, last[2] = -1.
- После того как посетили вершины 8 и 9, будет first[2]=0, last[2]=1 (значения - индексы в векторе вершин на уровне 2).
- Вернувшись в вершину 2, запоминаем для заявки (2, 1) first=0, last=1. Ставим req=false, чтобы, вернувшись в вершину 1, её заявку (1,2) проигнорировать, так как её интервал полностью содержит интервал заявки (2,1).

```cpp
// для каждой заявки определим границы её интервала на соотв. уровне
void dfs(int u) {
  // возможно, каким-то заявкам нужна информация об уровне вершины u
  st[levels[u]].first = std::min(st[levels[u]].first, pos[u]);
  st[levels[u]].second = std::max(st[levels[u]].second, pos[u]);
  // а какие заявки от этой вершины требуется обработать?
  for (int k : orders[u]) {
    int lev = levels[u] + k; // отрезок будет на этом уровне
    st[lev] = std::make_pair(1000000000, -1);
    req[lev] = true;
  }
  for (int v : g[u])
    dfs(v);
  // информация для наших заявок уже готова
  for (int k : orders[u]) {
    int lev = levels[u] + k; // отрезок заявки лежит на этом уровне
    if (!req[lev]) continue; // видимо, нашли вложенную заявку
    std::pair<int, int> range = st[lev];
    // вышестоящие заявки не обрабатываем
    req[lev] = false;
    ranges[lev].push_back(range); // очередной отрезок найден
  }
}
```

Оценка сложности: $O(n \cdot \log(n))$