Всероссийская олимпиада школьников по информатике 2018/2019
Региональный этап

Разбор задач

подготовил: Андрианов И.А.,
Вологодский государственный университет

Вологда
2019 г.

Задача 1 - «Два измерения»

Даны целые числа *l*, *r*, *a*. Найти количество таких пар *i*, *j*, что *l* ≤ i < j ≤ *r* и величина (*j* - *i*) кратна a. Ограничения: l, r, a до 10^9

Возможное решение.

Пусть *len = r - l + 1* – длина интервала *l..r.*

Выпишем количество вариантов размещения интервала *i..j* на отрезке *l..r* в зависимости от возможных значений (j-i):

(*j-i*) число_вариантов

---

a len-a *пример (len=5, a=2):* ***.. .***. ..***

2a len-2a

3a len-3a

... ...

n·a len-n·a

Найдём n. Поскольку len-n·a ≥ 1, то n = (len - 1) div a.

Просуммируем все варианты, используя сумму арифметической прогрессии.

Ответ равен **n · len - a · (1 + n) · n div 2**

Задача 2 - «Полные квадраты»

Дано целое k. Найти минимальное целое неотрицательное число, квадрат которого встречается в последовательности k, k+1, k+1+3, k+1+3+5, … Ограничения: $-10^9 \le k \le 10^9$

Возможное решение.

- При k=0 ответ равен 0.

- При k > 0, чтобы число было точным квадратом, k должно равняться "хвосту" одной из следующих сумм (поскольку любой квадрат выражается суммой такого вида):
0+1, 0+1+3, 0+1+3+5, …
Например, для k = 21 имеем 0+1+3+(**5+7+9**) = 25 = 5^2.
Подходящих сумм может быть несколько, нам нужна самая маленькая.

- При k < 0 рассуждения похожи: -k должно равняться хвосту одной из сумм, а ответ определяется как сумма элементов до "хвоста":
-25+**0+1+3**+(5+7+9) = 4 = 2^2.

Задача 2 - продолжение

Итак, задача свелась к следующей: найти такое минимальное нечетное f, что $f + (f+2) + (f+4) + \ldots + (f + 2 \cdot x) = k$ для некоторого $x >= 0$.
Выразим f через x: $f = k / (1 + x) - x$
Отсюда видно, что **(x + 1) должен быть делителем числа k**.

Алгоритм решения.
Переберём делители числа k по убыванию (достаточно начать от корня квадратного из k, так как при бо́льших делителях f будет получаться отрицательным).
Для очередного делителя (x + 1) вычислим $f = k / (x + 1) – x$.
Если f получился нечетным, то выводим ответ (и завершаем работу):
при $k > 0$ ответ = $(f + 2 \cdot x + 1)$ div 2, для $k < 0$ ответ = f div 2.
Если ответ не был найден, выводим *none*.

В дополнение.
При желании можно доказать, что если k mod 4 = 2, то ответа не существует, в остальных случаях он найдётся.

Задача 3 - «Автоматизация склада»

Способ решения 1.

Пусть робот вынул верхнюю карту из отсека. Посмотрим, когда впервые встретится дверь для этой карты, и какие двери встретятся раньше её.

Основная идея: надо положить карту в отсек сразу после всех карт, которые потребуются для открывания дверей, что встретятся раньше.

Рассмотрим пример:

двери: **4** 1 2 **4** 4
карты: **4** 2 1 3

Вынимаем карту 4, открываем дверь. Видим, что снова нам 4-я дверь понадобится после дверей 1 и 2. Значит, надо засунуть карту на ближайшую позицию после карт 1 и 2 в отсеке:

двери: 4 1 2 4 4
карты: 2 1 4 3

Какие образом это делать эффективно?

Задача 3 – способ 1 - продолжение

▫ Построим вспомогательный массив *next*, где *next*[i] – позиция следующего вхождения двери в позиции *i*.
▫ Построим массив *pos,* где *pos*[*c*] – номер ближайшей двери среди ещё неоткрытых, которая открывается картой *c*. По мере открывания дверей массив *pos* меняется, а с помощью массива *next* эти изменения легко делать за константное время.
▫ Чтобы выбрать позицию для вставки карты *f*, нужно найти в отсеке самую правую карту *c* такую, что *pos[c] < pos[f]*.

Итак, нам нужна структура данных для представления отсека робота, которая умеет:

- Извлекать первый элемент
- Искать самый правый элемент со значением меньше заданного и возвращать его порядковый номер
- Вставлять элемент в заданную позицию (или после указанного)

Варианты такой структуры:

- sqrt – декомпозиция
- декартово дерево по неявному ключу

Задача 3 – способ 1 – продолжение

SQRT-декомпозиция устроена так:

- отсек робота представляется связным списком из √n блоков
- каждый блок хранит дек с номерами карт, а также значение функции $\min(pos[c_i])$ для всех карт c_i этого блока
- поиск места вставки выполняется справа налево – сначала ищется блок, затем нужная позиция в нём. Если при вставке блок стал слишком большим, он расщепляется на два.

Вычислительная сложность каждой операции – O(√n), в итоге общая сложность составляет O(m·√n). Такое решение набирает 74 балла. Вероятно, можно оптимизировать до 100 баллов (*на ноутбуке автора разбора разница в скорости работы данного решения и одного из решений, набирающих 100 баллов, составила всего 12%*).

Альтернативная структура данных - декартово дерево по неявному ключу. Позволяет выполнять операции за O(log(n)) вместо корня. Такое решение будет иметь сложность O(m·log(n)) и способно набрать до 100 баллов в зависимости от реализации.

Задача 3 – способ решения 2

Вначале определим последовательность, в которой карты достаются из отсека.

Ключевая идея: если карты мы кладём в отсек оптимально, то: каждая карта *только в первый раз* может извлекаться не для того, чтобы сразу открыть дверь. Второй и все следующие разы номер карты на вершине отсека будет как раз совпадать с номером текущей двери.
При желании можно доказать по индукции *(но не нужно)*.

Теперь легко получить, в каком порядке извлекаются карты.

- Перебираем двери.
- Если карта для очередной двери уже извлекалась раньше, то кидаем её в конец массив k.
- Если не извлекалась, то достаём из отсека карты до тех пор, пока не достанем нужную, и добавляем в конец массива k. Для всех извлекаемых карт при этом отмечаем, что она извлечена.

Задача 3 – способ решения 2 – продолжение

Пример кода для определения порядка, в котором достаются карты.

- вектор a содержит номера дверей,
- очередь b – карты в отсеке робота,
- вектор k – результат.

```
std::vector<int> used(n + 1);
std::vector<int> k;
for (int i = 0; i < m; i++) {
  if (used[a[i]]) {
    k.push_back(a[i]);
  } else {
    int card = 0;
    while (card != a[i]) {
      card = b.front();
      b.pop();
      used[card] = true;
      k.push_back(card);
    }
  }
}
```

Задача 3 – способ решения 2 – продолжение

Теперь определим, на какие позиции карты будет возвращаться. Берём в массиве k очередную карту *c* и найдём там её следующую позицию (чтобы делать это быстро, вначале можно провести простой преподсчёт).

Искомая позиция карты в отсеке – это **количество различных элементов** между текущей позицией карты *c* и следующей её позицией в массиве k.

Пояснение: то, сколько различных карт мы достали, перед тем как снова достали карту *c*, как раз и говорит о том, сколько карт было перед ней в тот момент, когда её засовывали. *Почему именно различных? Какие-то карты перед ней могли несколько раз вынимать, снова засовывать перед ней и опять вынимать...*

В итоге пришли к подзадаче ***поиска количества различных элементов на отрезке офлайн*** («офлайн» означает, что все отрезки массива заранее известны, и массив между запросами не меняется).

Задача 3 – способ решения 2 – продолжение

Подзадача поиска количества различных элементов на отрезке.
Дан массив k и набор запросов вида (l_i, r_i). Для каждого запроса нужно узнать, сколько уникальных элементов лежит на отрезке массива с индексами от l_i до r_i. Массив между запросами не меняется.

Один из способов решения. Отсортируем интервалы по возрастанию r_i. Пусть r – правая граница очередного интервала.

Введём вспомогательный массив p, где p[i] = 1, если для элемента k[i] не существует элемента с таким же значением правее *i* вплоть до позиции r.

Будем поддерживать массив p в актуальном состоянии при смещении границы *r* вправо (для этого можно использовать ещё один вспомогательный массив pos, где pos[s] – последнее на данный момент вхождение элемента s).

Построим над массивом *p* **дерево отрезков** для операции «сумма» (собственно, сам массив p в явном виде даже не нужен). Тогда число уникальных элементов – это просто запрос суммы на интервале.

Возможны другие способы решения задачи 4, например:
https://codeforces.com/blog/entry/64816?locale=ru#comment-487849

Задача 5 - «Неисправный марсоход»

Даны целые числа a, b, c. Нужно попасть из a в b, прибавляя 1 или 2 и не попадая на числа, кратные c. Ограничения: все числа до 10^9.

Решение. Числа, кратные c, разбивают интервал на отрезки одинаковой длины:

a ... x·c ..... (x+1)·c ..... (x+2)·c ..... (x+3)·c .. b

Нам нужно:

- определить минимальное число шагов, чтобы дойти от *a* до начала очередного целого отрезка (точней, числа перед ним)
- добавить минимальное число шагов внутри целого отрезка, умноженное на количество таких отрезков
- добавить минимальное число шагов после последнего целого отрезка до *b*.

Все формулы легко выводятся, см. пример кода далее.

Задача 5 – продолжение

Пример решения.

Здесь first – первое число после *a*, кратное *c*; next – второе; last – последнее (на интервале *a* – *b*); nsegments – число полных сегментов.

```
int steps(int x, int y) {
  return (y - x + 1) / 2;
}

int main() {
  long long a, b, c;
  std::cin >> a >> b >> c;
  int first = a + (c - a % c);
  if (first > b) {
    std::cout << steps(a, b);
  } else {
  long long next = first + c;
  long long last = b - b % c;
  long long nsegments = (last - first) / c;
  std::cout << steps(a, first - 1)
     + (1 + steps(first + 1, next - 1)) * nsegments
     + steps(last - 1, b);
  }
}
```

Задача 6 - «Интервальные тренировки»

Требуется определить количество «пилообразных» последовательностей из целых положительных чисел с первым членом = k и суммой всех элементов = n.

Идея – динамическое программирование.

Пусть f[s][last][p] - число последовательностей, где:

- сумма элементов = *s*,
- последний член = *last*,
- p = 0, если предыдущий член был меньше last
- p = 1, если предыдущий член был больше last

Чтобы вычислить f[s][last][p], переберём возможные значения предыдущего члена (интервал перебора разный при p=0 и при p=1).

Задача 6 – продолжение

Заполнение массива f и вычисление ответа на задачу:

```
f[k][k][0] = f[k][k][1] = 1;
for (int s = k + 1; s <= n; s++) {
  for (int last = 1; last <= s; last++) {
    // 1). prev < last
    for (int prev = 1; prev <= last - 1 && prev + last <= s; prev++) {
      f[s][last][0] += f[s - last][prev][1];
      f[s][last][0] %= 1000000007;
    }
    // 2). prev > last
    for (int prev = last + 1; prev + last <= s; prev++) {
      f[s][last][1] += f[s - last][prev][0];
      f[s][last][1] %= 1000000007;
    }
  }
}
int ans = 0;
for (int last = 1; last <= n; last++) {
  ans = (ans + f[n][last][0]) % 1000000007;
  ans = (ans + f[n][last][1]) % 1000000007;
}
std::cout << ans;
```

Проблема: решение работает за куб, и последняя подзадача не проходит.

Задача 6 – продолжение

Заметим, что во внутренних циклах просто прибавляется *сумма непрерывного отрезка одной из предыдущих строк* матрицы f. Избавимся от внутренних циклов с помощью преподсчёта **префиксных сумм** для каждой строки f. Для этого введём вспомогательную матрицу *c*:

```
for (int s = k + 1; s <= n; s++) {
  for (int last = 1; last <= s; last++) {
    // 1). prev < last
    f[s][last][0] += c[s - last][std::min(last - 1, s - last)][1];
    f[s][last][0] %= 1000000007;
    // 2). prev > last
    if (s - last >= last) {
      f[s][last][1] += c[s - last][s - last][0] - c[s - last][last][0];
      if (f[s][last][1] < 0) f[s][last][1]  += 1000000007;
      f[s][last][1] %= 1000000007;
    }
  }
  for (int j = 1; j <= n; j++) {
    c[s][j][0] = (c[s][j - 1][0] + f[s][j][0]) % 1000000007;
    c[s][j][1] = (c[s][j - 1][1] + f[s][j][1]) % 100000007;
  }
}
```

Такое решение работает за квадрат и набирает 100 баллов.

Примечание: этот приём ещё потребуется в задаче 4 первого тура.

Задача 7 - «Экспедиция»

Общая идея решения.

Построим неориентированный граф, где вершины – это кандидаты, а рёбра показывают, что данные два кандидата не должны лететь вместе.

Заметим, что граф имеет специфический вид:

- он состоит из одной или более компонент связности,
- каждая компонента связности – это либо дерево, либо «почти дерево» (имеет ровно один цикл)

Ответ на задачу равен сумме ответов для всех компонентов связности.

Ответ для компоненты-дерева можно найти с помощью ДП на дереве.

Ответ для компоненты-«почти дерева» можно найти тоже с помощью ДП, но чуть сложнее.

Задача 7 – продолжение

Поиск ответа для компоненты связности – дерева.

Возьмём любую вершину и назначим её корнем.
Будем для каждой вершины *u* находить пару чисел (skip, take), где:
▫ skip – ответ для поддерева с корнем в вершине *u* при условии, что вершину *u* мы не берём,
▫ take – ответ для поддерева с корнем в вершине *u* при условии, что вершину *u* берём

Как их находить? Сначала рекурсивно найдём ответы для всех детей, попутно вычислив следующие суммы:
▫ sum_skip – это сумма значений skip во всех детях
▫ sum_any – это сумма значений max(skip, take) во всех детях

Тогда ответ для корня u:

skip = sum_any
take = 1 + sum_skip

Задача 7 – продолжение

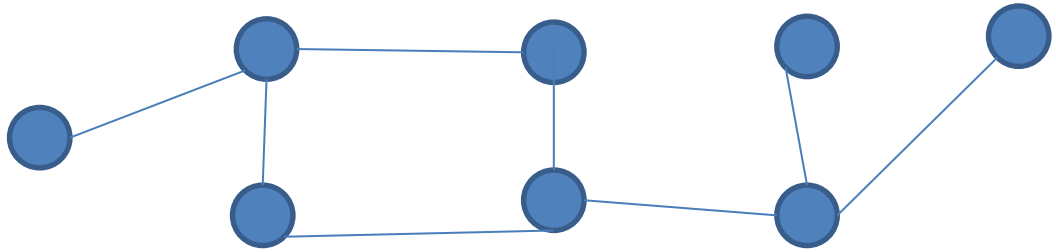
ДП на дереве – пример реализации:

```
struct Pair {
  int skip;
  int take;
};


Pair walk_tree(int u, int prev = -1) {
  color[u] = RED;
  int sum_skip = 0, sum_any = 0;
  for (int v : g[u]) {
    if (v == prev || v < 0 || color[v] == RED) continue;
    color[v] = RED;
    Pair p = walk_tree(v, u);
    sum_skip += p.skip;
    sum_any += std::max(p.skip, p.take);
  }
  return Pair{sum_any, 1 + sum_skip};
}
```

Задача 7 – продолжение

ДП на «почти дереве» (есть ровно один цикл)



К каждой вершине цикла прицеплено дерево. Выполним для каждого дерева вычисления, как на предыдущих двух слайдах (корнем дерева будет его вершина на цикле). *Для удобства можно вначале удалить рёбра цикла из графа.*

Возьмём любую вершину цикла в качестве начальной. Рассмотрим два случая: эта вершина берётся или не берётся.
Для каждого случая пройдём по циклу и вычислим пару (skip, take) для каждой вершины цикла, используя пару (skip, take) из предыдущей вершины цикла, а также данные из прицепленного поддерева.

Дойдя до последней вершины цикла, возьмём max(skip, take) – это будет ответ. В конце нужно взять max из двух ответов для обоих случаев.

Задача 7 – продолжение

ДП на “почти дереве” – пример реализации

```
erase_cycle_edges();
std::vector<Pair> p;
for (int v : cycle) {
  p.push_back(walk_tree(v));
}
// try to take the first vertex
std::vector<Pair> cost(p.size());
cost[0] = p[0];
for (int i = 1; i < (int) p.size(); i++) {
  cost[i].skip = std::max(cost[i - 1].skip, cost[i - 1].take) + p[i].skip;
  cost[i].take = cost[i - 1].skip + p[i].take;
}
int best1 = cost.back().skip;
// try to not take the first vertex
cost[0].skip = p[0].skip;
cost[0].take = 0; // no profit
for (int i = 1; i < (int) p.size(); i++) {
  cost[i].skip = std::max(cost[i - 1].skip, cost[i - 1].take) + p[i].skip;
  cost[i].take = cost[i - 1].skip + p[i].take;
}
int best2 = std::max(cost.back().skip, cost.back().take);
ans += std::max(best1, best2);
```

Общая вычислительная сложность решения – O(n)

Задача 8 - «Разбиение на пары»

Рассмотрим искомое разбиение на пары. Для каждого j есть $n/2$ индексов i, таких что $a_{i,j} \leqslant b_j$ и $n/2$ индексов i, таких что $a_{i,j} \geqslant b_j$. Таким образом, если решение существует, то в качестве b_j можно выбрать **медиану** чисел $a_{1,j}$, a_{2j}, … ,a_{nj} (медиана — это среднее арифметическое n/2-й и n/2+1-й порядковых статистик)

Подзадача 1: можно перебрать все разбиения на пары и для каждого проверить корректность.

Подзадача 2: параметр всего один, поэтому артефакты можно разбить на пары жадно: первый и последний, второй и предпоследний, и так далее. Решение всегда существует.

Подзадача 3: все значения каждого параметра различны. Для каждого артефакта построим битовый вектор длины k. Его j-я координата равна 0 или 1 в зависимости от того, больше или меньше значение j-го параметра, чем соответствующее медианное значение.

Артефакты надо разбить на пары, которым соответствуют битовые вектора, являющиеся *отрицанием* друг друга. Это можно сделать жадным алгоритмом. Если решение не найдено, то его не существует.

Задача 8 – продолжение

Трудность решения оставшихся подзадач: все параметры, кроме первого, могут принимать равные значения, в том числе равные медиане. Тогда предыдущий способ решения не годится.

Подзадача 4: параметров два. Разделим артефакты по первому параметру на два множества, условно назовём их красным и синим: первые n/2 и вторые n/2 (по возрастанию значения первого параметра). В каждую пару должен войти один красный и один синий артефакт.

Сопоставим каждому артефакту значение 0, 1 или -1 в зависимости от того, равно, больше или меньше значение его второго параметра медианному. Получаем 6 множеств: красные R_0, R_1, R_{-1} и, соответственно, синие B_0, B_1, B_{-1}.

Артефакты из R_1 должны войти в пары с B_{-1}, из R_{-1} — в пары с B_1. Артефакты из R_0 могут войти в пары с любыми синими, а из B_0 с любыми красными. Их можно распределить жадно.

Альтернативное решение: отсортируем артефакты по второму параметру (при равенстве произвольным образом), и первые n/2 назовём малыми, а последние n/2 большими. Тогда красные большие входят в пары с синими малыми, а красные малые с синими большими.

Задача 8 – продолжение

Подзадача 5: параметров больше двух, *n* до 400.

Для каждой пары артефактов проверим, можно ли создать из них пару. Проведём ребро, если можно.

Как и прежде, разделим артефакты на красные и синие по первому параметру. Заметим, что ребра соединяют артефакты разного цвета – следовательно, граф двудольный.

Найдём в получившемся графе полное паросочетание алгоритмом Куна.

В графе $O(n^2)$ рёбер, поэтому решение работает за $O(n^3)$.

Задача 8 – продолжение

Подзадача 6: параметров больше двух, *n* до $2 \cdot 10^5$.

Сопоставим каждому артефакту строку длины k из 0, 1 и -1: j-е число равно -1, если $a_{i,j} < b_j$, 1, если $a_{i,j} > b_j$, и 0, если $a_{i,j} = b_j$. Заметим, что первое число в строке не может быть равно 0. Таким образом, точки разбиваются на $2 \cdot 3^{k-1}$ групп.

Заведём вершину для каждой группы. Вершины, которые соответствуют точкам, у которых $a_{i,1} < b_1$ — это первая доля, остальные — вторая. Также заведём исток и сток.

В вершины первой доли проведём рёбра из истока с весом, равным числу точек в соответствующей группе. Из вершин второй доли в сток — аналогично. Между вершинами из разных долей проведём рёбра веса $+\infty$, если соответствующие им точки можно объединить в пары.

Найдём в таком графе максимальный поток. Зная величину потока по всем рёбрам графа, несложно восстановить ответ.

Задача 4 - «Машинное обучение»

Немного переформулируем условие эффективности плана:
(a_i and a_{i+1}) = a_i для любого i<n.
Отсюда видно, что любое решение – это массив, разбитый на блоки из одинаковых чисел, где числа в последующих блоках являются *надмасками* предыдущих (т.е. биты могут добавляться, но не могут удаляться).
Количество блоков – порядка $\log_2(k)$ – число битов в числе k.
Метод решения – **динамическое программирование**.

Подзадача 1. *n* до 500, m=0 (нет дополнительных требований).
dp[i][j] - количество искомых массивов длины i, таких что последний элемент равен j. Несложно пересчитывается через предыдущие.

Подзадача 2. *n* до 50, m до 50. В ДП нужно добавить ещё параметр:
dp[i][j][k] – количество искомых массивов длины i, таких что последний элемент равен j, а k — индекс последнего элемента, не равного j.
Тогда при пересчёте новых значений через предыдущие можно будет учесть и требования неравенства некоторых пар элементов.

Задача 4 – продолжение

Для решения остальных подзадач нужно перестать хранить значение последнего элемента в явном виде.

Ключевая идея: в ДП вместо последнего числа достаточно знать количество единиц в его битовом представлении:
dp[i][j] - количество искомых массивов длины i, где количество единиц в битовом представлении последнего числа равно j.

Чтобы решить искомую задачу, достаточно перебрать x в диапазоне от 0 до k - последнее число в искомом массиве, и прибавить к ответу значение dp[n][cnt(x)], где функция cnt(x) возвращает количество единиц в битовом представлении x.

Задача 4 – продолжение

```
dp[i][j] = 0;
if (check(1, i)) {
    dp[i][j] += 1;
}
for (int p = 1; p < i; p++) {
    if (check(p + 1, i) {
        for (int h = 0; h < j; h++) {
            dp[i][j] += dp[p][h] * C(j, h);
        }
    }
}
```

Функция check(l, r) проверяет, можно ли сделать все элементы отрезка [l; r] равными, не нарушив требований неравенства некоторых пар.

Функция *C*(j, *h*) возвращает число способов выбрать *h* элементов из *j*.

Изначально проверяется, можно ли сделать все элементы с 1-го по *i*-й равными. Если да, то прибавим 1 к *dp*[*i*][*j*]. Таким образом мы учли массивы, состоящие из одного блока одинаковых элементов.

Теперь считаем, что массив состоит из нескольких блоков и переберем *p* - конец предыдущего блока. С помощью функции *check*(*p* + 1, *i*) проверим, может ли конец предыдущего блока быть в позиции *p*, и если да, переберем *h* — количество единиц в битовом представлении числа предыдущего блока и добавим к ответу величину *dp*[*p*][*h*] · *C*(*j, h*).

И правда, при фиксированном числе с *j* единицами в битовом представлении существует ровно *C(j, h)* чисел, которые содержат *h* единиц в битовом представлении и при этом являются подмаской данного числа.

Задача 4 – продолжение

Предыдущее решение всё ещё не проходит все подзадачи – надо ускорять ещё.

Пусть *dp*[*i*][*j*][*c*] - количество массивов длины *i*, удовлетворяющих требованиям и состоящих из *c* блоков одинаковых элементов, где элемент последнего блока имеет *j* единиц в битовом представлении. Формулы пересчета почти не поменяются: если раньше мы пересчитывали значение *dp*[*i*][*j*] через *dp*[*p*][*h*], то теперь будем пересчитывать *dp*[*i*][*j*][*c*] через *dp*[*p*][*h*][*c* -1].

А теперь просто ***выкинем параметр j*** из ДП и перестанем при пересчете *dp*[*i*][*c*] домножать на *C*(*j, h*):

```
for (int i = 1; i <= n; i++) {
    if (check(1, i)) {
        dp[i][1] += 1;
    }
    for (int j = 2; j <= B; j++) {
        for (int p = 1; p < i; p++) {
            if (check(p + 1, i)) {
                dp[i][j] += dp[p][j - 1];
            }
        }
    }
}
```

Данный код теперь считает не окончательные ответы, а количество разбиений массива на заданное число блоков, чтобы выполнялись все требования.

Их ещё надо на что-то домножать. На что?...

Задача 4 – продолжение

Пусть массив составлен из *b* блоков с количеством единичных битов c_1, c_2, …, c_b. Тогда к ответу для него нужно будет добавить $dp[n][b] \cdot C_{c_2}^{c_1} \cdot C_{c_3}^{c_2} \cdot \dots \cdot C_{c_b}^{c_{b-1}}$ (где каждое *C* – это число способов распределить единичные биты предыдущего блока среди единичных битов следующего).

Посчитаем массив *cnt*[*j*][*b*]: сумму $C_{c_2}^{c_1} \cdot C_{c_3}^{c_2} \cdot \dots \cdot C_{c_b}^{c_{b-1}}$ по всем возможным массивам (c_1, c_2, …, c_b), где элементы возрастают, и при этом c_b=j. Это можно сделать ***ещё одним ДП***.
Теперь количество эффективных массивов, состоящих из b блоков, у которых в последнем блоке стоит зафиксированное число с j единицами в битовом представлении, равно dp[n][b] · cnt[j][b].

Ещё посчитаем массив cntBits[j] - количество чисел от 0 до k с j единицами в битовом представлении.
Это можно сделать ***ещё одним ДП*** ;)

Задача 4 – продолжение

Теперь ответ может быть сосчитан так:

```
int ans = 0;
for (int j = 0; j <= B; j++) {
    for (int b = 1; b <= min(n, j + 1); b++) {
        ans += dp[n][b] * cnt[j][b] * cntBits[j];
    }
}
```

Чтобы получить полное решение, нужно:

- быстро считать массив cntBits
- быстро вычислять функцию check(l, r)
- избавиться от вложенного цикла по *p* в вычислении dp[i][j] - *по аналогии с тем, как это делали в задаче №6.*

В итоге можно получить решение со сложностью $O(n \cdot log(k) + \log^2 k)$

Более подробный разбор от авторов задач ищите здесь:
http://neerc.ifmo.ru/school/archive/2018-2019.html