Всероссийская олимпиада школьников по информатике
Вологодская область
II (муниципальный) этап
2019-2020 учебный год
9 - 11 классы
Методические рекомендации по разбору задач

Задача 1. Красивые номера (100 баллов)

Первую подзадачу можно решить, перебрав все числа от 1 до N и проверив для каждого, состоит ли оно из одинаковых цифр.

Возможное решение на максимальный балл – строить в цикле все числа вида 1, 11, 111 и так далее, пока не превысим N. Затем аналогично строить числа 2, 22, 222 …, и так далее.

При реализации нужно учесть, что при $N = 10^{18}$ возможно переполнение знакового 64-битного типа: если число из 18 девяток ещё меньше N, то число из 19 девяток уже даёт переполнение. У участников, пишущих на Питоне, этой проблемы нет, а на других языках этот момент нужно было учитывать (например, использовать беззнаковый тип либо написать дополнительное условие).

Пример решения на C++:

```
#include <iostream>

int main() {
  unsigned long long n;
  std::cin >> n;
  int ans = 0;
  for (int d = 1; d <= 9; d++) {
    unsigned long long cur = d;
    while (cur <= n) {
      ans++;
      cur = cur * 10 + d;
    }
  }
  std::cout << ans;
}
```

Задача 2. Умножения в массиве (100 баллов)

Если в массиве есть нули, то очевидно, ответом будет количество ненулевых элементов.

Рассмотрим теперь случай, когда в массиве нулей нет. Найдём в массиве максимальное по модулю число, обозначим его за X. Проверим, верно ли, что X делится на все элементы массива. Если это

не так, то ответ равен N (так как каждый элемент придётся на что-то домножать, чтобы он стал равен наименьшему общему кратному). Если же X делится на все элементы, то посмотрим, каких элементов в массиве больше – равных X, либо равных $-X$. Ответ будет равен $N - max$(количество X, количество $-X$).

Пример решения на C++:

```
#include <iostream>
#include <vector>
#include <algorithm>

int main() {
  std::ios_base::sync_with_stdio(0); std::cin.tie(0);
  int n;
  std::cin >> n;
  std::vector<int> a(n);
  int zeroes = 0;
  for (int i = 0; i < n; i++) {
    std::cin >> a[i];
    if (a[i] == 0) {
      zeroes++;
    }
  }
  if (zeroes > 0) {
    std::cout << n - zeroes;
    return 0;
  }
  int x = a[0];
  for (int p : a) {
    if (std::abs(p) > std::abs(x)) {
      x = p;
    }
  }
  for (int p : a) {
    if (x % p != 0) {
      std::cout << n;
      return 0;
    }
  }
  int cnt_x = 0, cnt_minusx = 0;
  for (int p : a) {
    if (p == x) cnt_x++;
    if (p == -x) cnt_minusx++;
  }
  std::cout << n - std::max(cnt_x, cnt_minusx);
}
```

Задача 3. Дизъюнкция (100 баллов)

Первую подзадачу можно решить полным перебором.

Для решения второй подзадачи можно использовать динамическое программирование (по аналогии с известной задачей о рюкзаке). Однако, поскольку полное решение задачи намного проще, чем данный способ, то подробно на нём останавливаться не будем.

На максимальный балл данная задача решается очень легко. В ответ включим все входные числа, удовлетворяющие следующему условию: если у числа *X* в каком-то бите стоит ноль, то и у рассматриваемого числа в этом бите тоже должен быть ноль. В противном случае мы получим лишний единичный бит, который никакими следующими дизъюнкциями уже не обнулится. Осталось только посмотреть, даёт ли дизъюнкция всех выбранных чисел значение *X*. Если нет, то решения не существует.

Пример решения на C++:

```
#include <iostream>
#include <vector>

int main() {
  std::ios_base::sync_with_stdio(0); std::cin.tie(0);
  int n, x, cur = 0;
  std::cin >> n >> x;
  std::vector<int> ans;
  for (; n > 0; --n) {
    int a;
    std::cin >> a;
    if ( (~(cur | a) | x) == -1) {
      cur |= a;
      ans.push_back(a);
      if (cur == x) break;
    }
  }
  if (cur == x) {
    for (int v : ans) {
      std::cout << v << ' ';
    }
  } else {
     std::cout << -1;
  }
}
```

Задача 4. Полоска (100 баллов)

Первую подзадачу можно решить перебором – на каждом шаге выбирать, куда ставить очередное число (влево или вправо). Сложность такого решения составит $O(2^N)$.

Для полного решения воспользуемся методом динамического программирования. Пусть функция *sol*(*left*, *filled*) возвращает

количество способов так заполнить часть полоски, что количество пустых клеток слева равно *left*, а количество занятых клеток равно *filled*. Рекуррентное соотношение имеет следующий вид:

sol(left, filled) = sol(left + 1, filled - 1) + sol(left, filled - 1)

Пояснение: на предыдущем шаге количество заполненных клеток было на одну меньше, а лишняя пустая клетка была либо слева, либо справа.

Чтобы данная рекурсивная функция не вычисляла одно и то же много раз, создадим матрицу, в которую будем записывать уже сосчитанные значения для каждой пары аргументов. Если значение уже было вычислено ранее, то функция будет просто возвращать его вместо того, чтобы вызывать себя рекурсивно.

Примечание: при ограничениях задачи ответ не помещается в 32-битный целый тип, поэтому надо использовать 64-битный.

Пример решения на C++:

```
#include <iostream>
#include <vector>

std::vector<std::vector<long long> > f;
int n, k;

long long sol(int left, int filled) {
  if (left + filled > n){
    return 0;
  }
  if (filled == 1) {
    return (left == k - 1) ? 1 : 0;
  }
  if (f[left][filled] < 0) {
    f[left][filled] = sol(left+1, filled-1)+sol(left, filled-1);
  }
  return f[left][filled];
}

int main() {
  std::cin >> n >> k;
  f.resize(n + 1, std::vector<long long>(n + 1, - 1));
  std::cout << sol(0, n);
}
```

Также можно было решить данную задачу, выведя комбинаторную формулу – ниже приведено такое решение на Python:

```
from math import factorial
n = int(input())
k = int(input())
print(factorial(n - 1) // factorial(k - 1) // factorial(n - k))
```

Задача 5. Камеры (100 баллов)

Для решения первой подзадачи вначале найдём три числа $d1$, $d2$ и $d3$ – три минимальных расстояния между парами соседних камер (замечание: при N=3 можно либо считать, что d_3 равно бесконечности, либо рассмотреть этот случай особо).

Попробуем удалить каждую камеру, за исключением первой и последней (несложно понять, что при K=1 первую или последнюю камеру удалять смысла нет). Пусть a – расстояние до соседа слева, b – до соседа справа. Пусть $s_1=min(a,b)$, $s_2 = max(a, b)$. Обозначим через $dmin$ минимальное расстояние, которое получится после удаления текущей камеры. Рассмотрим несколько случаев.

Если $s_1=d_0$ и $s_2=d_1$, то $dmin = min(s_1+s_2, d_2)$, поскольку расстояния d_0 и d_1 исчезнут при удалении камеры, а расстояние s_1+s_2 добавится.

Если $s_1=d_0$, а $s_2\neq d_1$, то $dmin = d_1$, так как это расстояние не исчезло, и оно не превышает s_1+s_2.

В остальных случаях $dmin = d_0$.

Осталось выбрать такую камеру, при удалении которой значение $dmin$ получается максимальным.

Пример решения первой подзадачи на C++:

```
#include<iostream>
#include<vector>
#include<algorithm>
const int INF = 2000000000;

int main() {
  std::ios_base::sync_with_stdio(0); std::cin.tie(0);
  int n, k;
  std::cin >> n >> k;
  std::vector<int> a(n), d;
  std::cin >> a[0];
  for (int i = 1; i < n; i++) {
    std::cin >> a[i];
    d.push_back(a[i] - a[i-1]);
  }
  std::sort(d.begin(), d.end());
  d.push_back(INF);
  int ans_len = 0, ans_pos = -1;
  for (int i = 1; i < n - 1; i++) {
    int s1 = a[i] - a[i - 1], s2 = a[i + 1] - a[i];
    if (s1 > s2) std::swap(s2, s1);
    int dmin;
    if (s1 == d[0] && s2 == d[1]) {
      dmin = std::min(s1 + s2, d[2]);
    } else if (s1 == d[0]) {
```

```
      dmin = d[1];
    } else {
      dmin = d[0];
    }
    if (dmin > ans_len) {
      ans_len = dmin;
      ans_pos = i;
    }
  }
  std::cout << ans_len << "\n" << ans_pos + 1 << "\n";
}
```

Вторую подзадачу можно решить, просто написав полный перебор всех вариантов удаления камер.

Для решения третьей задачи воспользуемся двоичным поиском. Пусть зафиксировано некоторое значение *d*, и мы хотим проверить, возможно ли удалить *K* камер, чтобы максимальное расстояние между оставшимися было больше или равно *d*. *(*)*

Самую левую камеру удалять смысла нет. Действительно, представим, что в оптимальном решении она удалена. Тогда возьмём самую левую не удалённую, и её удалим, а первую восстановим. Такое решение будет, как минимум, не хуже предыдущего.

Итак, пусть *i* – номер последней камеры, которую мы решили не удалять. Вначале *i* = 1. Найдём *j* – номер ближайшей после *i* камеры, находящейся на расстоянии больше *d* от неё. Все камеры, лежащие между *i*-й и *j*-й, придётся удалить, чтобы не нарушилось условие *(*)*. А вот камеру *j* удалять не нужно – по тем же соображениям, что и камеру 1. И наша задача свелась к аналогичной: присваиваем *i* := *j*, и продолжаем в цикле делать то же самое.

По окончании цикла посмотрим, сколько камер нам пришлось удалить. Если больше *K*, то значение *d* слишком велико, и его надо уменьшить, а в противном случае запомним текущее *d* и попробуем его увеличить.

Пример решения на C++:

```
#include <iostream>
#include <vector>

struct Camera {
  int x;
  bool deleted;
};

int main() {
  std::ios_base::sync_with_stdio(0); std::cin.tie(0);
  int n, k;
```

```
  std::cin >> n >> k;
  std::vector<Camera> s(n);
  for (int i = 0; i < n; i++) {
    std::cin >> s[i].x;
    s[i].deleted = false;
  }
  int left = 0, right = s.back().x - s.front().x, ans = 0;
  while (left <= right) {
    int mid = (left + right) / 2, rem = 0, prev = 0;
    for (int i = 1; i < n; i++) {
      if (s[i].x - s[prev].x < mid) {
        rem++;
        if (rem > k) break;
      } else {
        prev = i;
      }
    }
    if (rem <= k) {
      ans = mid;
      left = mid + 1;
    } else {
      right = mid - 1;
    }
  }
  std::cout << ans << '\n';
  int rem = 0, prev = 0;
  for (int i = 1; i < n; i++) {
    if (s[i].x - s[prev].x < ans) {
      rem++;
      std::cout << i + 1 << ' ';
      s[i].deleted = true;
    } else {
      prev = i;
    }
  }
  for (int i = 0; i < n && rem < k; i++) {
    if (!s[i].deleted) {
      std::cout << i + 1 << ' ';
      rem++;
    }
  }
}
```

Информация о тестах для задач 9-11 классов

Задача	Тестов из условия	Основ-ных тестов	Баллов за один основ-ной тест	Проверяющая программа
1.Красивые номера	1	20	5	**check.exe**
2. Умножения в массиве	2	20	5	**check.exe**
3. Дизъюнкция	2	20	5	**check.exe**
4. Полоска	2	20	5	**check.exe**
5. Камеры	2	20	5	**check.exe**

Информация об использовании системы автоматической проверки решений приведена в Требованиях к организации и проведению муниципального этапа.