Всероссийская олимпиада школьников по информатике
Вологодская область
II (муниципальный) этап
2017-2018 учебный год
9 - 11 классы
Методические рекомендации по разбору задач

Задача 1. Количество прямоугольников (100 баллов)

Левая граница прямоугольника может изменяться от 1 до *q*, верхняя граница - от 1 до p, правая - от q до N, нижняя - от p до M. Тогда общее количество вариантов равно $p \cdot q \cdot (m - p + 1) \cdot (n - q + 1)$. Заметим, что ответ может быть порядка 10^{16}, поэтому необходимо использовать 64-битный целый тип.

Пример решения на C++:

```
#include<iostream>

int main() {
  long long m, n, p, q;
  std::cin >> m >> n >> p >> q;
  std::cout << p * q * (m - p + 1) * (n - q + 1);
}
```

Задача 2. Пароль (100 баллов)

Входное число прочитаем как строку. Припишем справа к ней любой символ, больший '9' - например, символ 'A'. Он будет играть роль ограничителя, не позволяя при просмотре выходить за число.

Затем будем просматривать число слева направо, при каждом просмотре удаляя первый встреченный символ, который меньше своего правого соседа. Повторив процесс *K* раз и удалив из строки-результата символ-ограничитель, получим искомый ответ.

Такой алгоритм имеет квадратичную сложность, он позволяет решить первые две подзадачи из трёх и получить 65 баллов.

Пример решения на Pascal (подзадачи 1, 2):

```
var par:string;
    k,i,j:integer;
begin
  readln(par);
  readln(k);
  par:=par+'A';
  for i:=1 to k do
```

```
    begin
      j:=1;
      while par[j]>=par[j+1] do
        j:=j+1;
      delete(par,j,1);
    end;
  delete(par,length(par),1);
  writeln(par);
end.
```

Чтобы решить третью подзадачу, необходимо улучшить алгоритм, чтобы не бегать по строке много раз. Для этого можно использовать структуру данных "стек". Вначале положим в стек любой символ, больший '9'. Затем будем просматривать строку слева направо. Взяв очередную цифру, будет удалять элементы из стека до тех пор, пока текущая цифра больше цифры на вершине стека, а общее количество удалённых цифр меньше *K*. После этого текущую цифру добавим в стек.

Если после прохода по строке количество удалённых цифр оказалось меньше *K*, то выполним недостающее количество удалений из стека. Оставшиеся в стеке символы образуют ответ.

Пример решения на C++:

```
#include <iostream>
#include <algorithm>
#include <stack>

int main() {
  std::string s;
  int k;
  std::cin >> s >> k;
  std::stack<char> st;
  st.push('z');
  int len = s.length();
  for (int i = 0, k1 = k; i < len; i++) {
    while (k1 > 0 && s[i] > st.top()) {
      st.pop();
      k1--;
    }
    st.push(s[i]);
  }
  while ((int)st.size() > len - k + 1)
    st.pop();
  std::string ans;
  while (st.size() > 1) {
    ans.push_back(st.top());
```

```
    st.pop();
  }
  std::reverse(ans.begin(), ans.end());
  std::cout << ans << std::endl;
}
```

Задача 3. Сортировка по сумме цифр (100 баллов)

Для решения первой подзадачи достаточно выполнить операции, которые описываются в условии. Заводим массив на 10^N чисел, сортируем его согласно условию. На каждый запрос проходимся по всему массиву, пока не встретим заданное число *K*, по пути подсчитывая, сколько чисел имеют длину больше, чем число *K*.

Описанный подход не позволит решить вторую подзадачу, так как при большом количестве запросов мы не можем себе позволить для каждого запроса проходить по всему массиву. Для решения второй подзадачи требуется найти способ быстро отвечать на каждый запрос, не обходя массив целиком.

Можно заметить, что ответ на самом деле не зависит от самого числа, он зависит только от его длины и суммы цифр. Поскольку комбинаций <длина, сумма цифр> совсем немного, мы можем завести матрицу ответов для всех возможных таких комбинаций. Для каждого запроса проверяем, есть ли в нашей матрице ответ для заданного значения. Если ответ уже считали ранее, то просто выводим его, иначе пробегаемся по всему массиву, считаем ответ, выводим и сохраняем в матрицу для будущего использования.

.Пример решения на C++ (подзадачи 1, 2):

```
#include <stdio.h>
#include <vector>
#include <algorithm>

void sumlen(int x, int &sum, int &len) {
  sum = len = 0;
  while (x > 0) {
    sum += x % 10;
    len++;
    x /= 10;
  }
}

struct Elem {
  int value, sum, len;
  bool operator < (Elem s) {
    return sum < s.sum || sum == s.sum && value < s.value;
```

```
  }
};

int main() {
  int n, m;
  scanf("%d %d", &n, &m);
  int p = 1;
  for (int i = 1; i <= n; i++) {
    p *= 10;
  }
  std::vector<Elem> a(p);
  for (int i = 0; i < p; i++) {
    a[i].value = i + 1;
    sumlen(a[i].value, a[i].sum, a[i].len);
  }
  std::sort(a.begin(), a.end());

  // f[len][sum]
  std::vector<std::vector<int> >
     f(n + 2, std::vector<int>(9 * (n + 1) + 1, -1));
  for (int i = 0; i < m; i++) {
    int k, sum, len;
    scanf("%d", &k);
    sumlen(k, sum, len);
    if (f[len][sum] < 0){
      int ans = 0;
      for (int j = 0; a[j].value != k; j++)
        if (a[j].len > len)
          ans++;
      f[len][sum] = ans;
    }
    printf("%d ", f[len][sum]);
  }
}
```

Для решения задачи на максимальный балл предлагается использовать метод динамического программирования. Создадим матрицу, одна из размерностей которой – это длина числа, а вторая – сумма цифр. В ячейках матрицы будем хранить количество чисел с соответствующей длиной и суммой цифр.

Для получения ответа для любого *K* достаточно пройтись по всем ячейкам матрицы, где значение суммы цифр меньше суммы цифр в числе *K*, а длина больше, чем длина *K*, и сложить значения в таких ячейках. Даже для максимальных значений в ограничениях задачи, матрица будет размера 18 x (18 x 9), что позволяет быстро отвечать на каждый запрос.

Единственный тонкий момент остаётся с числом 10^N. Это единственное число длины N+1, его можно не учитывать в таблице, а обработать отдельно. Это число будет стоять раньше любого числа с суммой цифр больше 1, и позже любого другого числа с суммой цифр, равной 1. Поэтому, при выводе ответа просто проверяем сумму цифр числа K: если она больше 1, то добавляем к ответу 1, если она равна единице, то не добавляем к ответу ничего.

Осталось понять, как построить матрицу методом динамического программирования. В таблице в N-ом столбце в S-ой строке подсчитано количество чисел длины N, имеющих сумму цифр S (ведущие нули запрещены). Обозначим это количество через f(S,N). Пересчет такой таблицы очень прост: последняя цифра может быть любой от 0 до 9. Количество таких чисел с последней цифрой d, длинной N, суммой всех цифр S составляет f(S-d,N-1). Если просуммировать по всем d, как раз и получим ответ.

Начальное заполнение матрицы: существует только по одному числу длины 1 с суммой цифр 1, 2, …, 9 (это и есть числа 1, 2, …, 9).

Заметим, что необходимо использовать 64-разрядный тип данных для вычислений значений в ячейках матрицы.

Пример решения на C++:

```
#include <stdio.h>
#include <vector>

int main() {
  int n, m;
  scanf("%d %d", &n, &m);
 // f[len][sum]
  std::vector<std::vector<long long> >
    f(n + 1, std::vector<long long>(9 * n + 1));
  for (int j = 1; j <= 9; j++)
    f[1][j] = 1;
  for (int i = 2; i <= n; i++) {
    for (int j = 1; j <= 9 * n; j++) {
      for (int d = 0; d <= 9 && j - d >= 0; d++) {
        f[i][j] += f[i - 1][j - d];
      }
    }
  }
  for (; m > 0; m--) {
    long long k;
    scanf("%I64d", &k);
    int klen = 0, ksum = 0;
    while (k > 0) {
      klen++;
      ksum += k % 10;
```

```
      k /= 10;
    }
    long long ans = 0;
    for (int len = klen + 1; len <= n; len++) {
      for (int sum = 1; sum < ksum; sum++) {
        ans += f[len][sum];
      }
    }
    if (ksum > 1)
      ans++;
    printf("%I64d ", ans);
  }
}
```

Задача 4. Пропавший путь (100 баллов)

Рассмотрим два способа решения данной задачи.

Первый способ использует структуру данных "система непересекающихся множеств" (англ. - disjoint set union, DSU). Структура данных широко известна, поэтому мы не будем её подробно рассматривать. Прочитать о ней можно, например, здесь: http://e-maxx.ru/algo/dsu

Суть решения состоит в следующем. Будем идти с конца, то есть вначале возьмём поле, полученное после всех закрашиваний. Каждая пустая клетка в нём становится отдельным одноэлементным множеством в нашем DSU.

Затем переберём все пары смежных пустых клеток и выполним объединение соответствующих множеств. В итоге каждое множество будет представлять собой изолированную связную область пустых клеток.

Проверим, совпадают ли множества, содержащие клетку (1,1) и клетку (N, M). Если да, то это означает, что лабиринт остался проходимым после всех закрашиваний - то есть выводим ответ 0.

Если множества не совпадают, то будем идти в обратную сторону в порядке закрашивания и отменять закраску последней клетки, предпоследней и так далее. Отменив закраску очередной клетки, соединим появившуюся пустую клетку с областями выше её, ниже, левее и правее. Как только сольются множества, содержащие клетку (1,1) и клетку (N, M), ответ будет найден.

Вычислительная сложность решения составляет $O(K \cdot log(M \cdot N))$.

Пример решения на C++:

```
#include <stdio.h>
#include <vector>
#include <algorithm>
#include <utility>

struct DSU {
  std::vector<int> parent, size;

  DSU(int n) : parent(n), size(n, 1) {
    for (int i = 0; i < n; i++)
      parent[i] = i;
  }

  int getRoot(int u) {
    while (parent[u] != u)
      u = parent[u];
    return u;
  }

  void unionRoots(int r1, int r2) {
    if (size[r1] > size[r2])
      std::swap(r1, r2);
    size[r2] += size[r1];
    parent[r1] = r2;
  }

  void tryUnion(int u1, int u2) {
    int r1 = getRoot(u1);
    int r2 = getRoot(u2);
    if (r1 != r2)
      unionRoots(r1, r2);
  }

};

int main() {
  int n, m, k;
  scanf("%d %d %d", &n, &m, &k);
  std::vector<std::vector<bool> > a(n, std::vector<bool>(m));
  DSU dsu(n * m);
  std::vector<std::pair<int, int> > steps(k);
  for (int i = 0; i < k; i++) {
    scanf("%d %d", &steps[i].first, &steps[i].second);
    --steps[i].first;
    --steps[i].second;
    a[steps[i].first][steps[i].second] = true;
  }
  for (int i = 0; i < n - 1; i++)
    for (int j = 0; j < m; j++)
```

```
      if (!a[i][j] && !a[i + 1][j])
        dsu.tryUnion(i * m + j, (i + 1) * m + j);
  for (int i = 0; i < n; i++)
    for (int j = 0; j < m - 1; j++)
      if (!a[i][j] && !a[i][j + 1])
        dsu.tryUnion(i * m + j, i * m + j + 1);
  if (dsu.getRoot(0) == dsu.getRoot(m * n - 1)) {
    puts("0");
    return 0;
  }
  for (--k; k >= 0; --k) {
    int i = steps[k].first, j = steps[k].second;
    a[i][j] = false;
    for (int di = -1; di <= 1; di++) {
      for (int dj = -1; dj <= 1; dj++) {
        if (di*di+dj*dj==1 && i+di>=0 && i+di<n && j+dj>=0
           && j+dj<m && !a[i+di][j+dj]) {
          dsu.tryUnion(i * m + j, (i + di) * m + j + dj);
          if (dsu.getRoot(0) == dsu.getRoot(m * n - 1)) {
            printf("%d\n", k + 1);
            return 0;
          }
        }
      }
    }
  }
}
```

Второй способ решения использует метод двоичного поиска в сочетании с любым алгоритмом обхода графа (в глубину или в ширину). С помощью двоичного поиска подбирается номер шага, а с помощью обхода графа проверяется существование пути. Если путь есть, то смещается левая граница к середине (пытаемся взять большее количество закрашиваний), если пути нет - смещается правая граница.

Вычислительная сложность - $O(\log(K) \cdot M \cdot N)$.

Пример решения на C++:

```
#include <stdio.h>
#include <vector>
#include <algorithm>
#include <utility>

int n, m, k;
std::vector<std::vector<bool> > a(n, std::vector<bool>(m));

bool dfs(std::vector<std::vector<bool> > &a, int i, int j) {
```

```
  if (a[i][j]) return false;
  a[i][j] = true;
  if (i == n - 1 && j == m - 1) return true;
  for (int di = -1; di <= 1; di++) {
    for (int dj = -1; dj <= 1; dj++) {
      if (di*di+dj*dj==1 && i+di>=0 && i+di<n && j+dj>=0
          && j+dj<m && !a[i+di][j+dj]) {
        bool found = dfs(a, i + di, j + dj);
        if (found) return true;
      }
    }
  }
  return false;
}

bool pathExists(std::vector<std::vector<bool> > a) {
  return dfs(a, 0, 0);
}

int main() {
  scanf("%d %d %d", &n, &m, &k);
  a.resize(n, std::vector<bool>(m));
  std::vector<std::pair<int, int> > steps(k);
  for (int i = 0; i < k; i++) {
    scanf("%d %d", &steps[i].first, &steps[i].second);
    --steps[i].first;
    --steps[i].second;
  }
  int left = 0, right = k - 1, prev = -1, answer = -1;
  while (left <= right) {
    int mid = (left + right) / 2;
    if (mid > prev) {
      for (int i = prev + 1; i <= mid; i++)
        a[steps[i].first][steps[i].second] = true;
    } else {
      for (int i = prev; i > mid; i--)
        a[steps[i].first][steps[i].second] = false;
    }
    prev = mid;
    if (pathExists(a)) {
      left = mid + 1;
    } else {
      answer = mid;
      right = mid - 1;
    }
  }
  printf("%d\n", answer + 1);
}
```

Задача 5. Новогодний хоровод (100 баллов)

Для решения данной задачи нужно было придумать какой-нибудь способ генерации последовательностей, чтобы в них не было повторов. Возможный вариант, как это сделать, проиллюстрируем на примере N = 5:

```
1 10  2  9  3  8  4  7  5  6  11
2  1  3 10  4  9  5  8  6  7  11
3  2  4  1  5 10  6  9  7  8  11
4  3  5  2  6  1  7 10  8  9  11
5  4  6  3  7  2  8  1  9 10  11
```

Число $2 \cdot N+1$ всегда стоит в самом конце. Посмотрим на остальные числа.

В первой строчке числа чередуются: 1, $2 \cdot N$, 2, $2 \cdot N$-1,… Для перехода от предыдущей строчки к следующей увеличим каждое число в ней на единицу. Если при этом число превысит $2 \cdot N$, то запишем вместо него 1 (то есть числа будут меняться как бы по кругу). Несложно заметить, что при таком подходе у нас действительно не будет возникать повторов. Возможны и другие способы решения этой задачи.

Пример решения на C++:

```
#include <stdio.h>
#include <bits/stdc++.h>
 int main() {
  int n, m;
  scanf("%d", &n);
  m = 2 * n + 1;
  std::vector<int> a(m + 1);
  for (int i = 1, j = 1; i < m; i += 2, j++){
    a[i] = j;
    a[i + 1] = m - j;
  }
  a[m] = m;
  for (int i = 0; i < n; i++) {
    for (int i = 1; i < m; i++) {
      printf("%d ", a[i]);
        a[i]++;
       if (a[i] == m) a[i] = 1;
    }
    printf("%d\n", a[m]);
  }
}
```

В материалах олимпиады можно найти также примеры и других решений задач на языках C++, Java, Pascal.

Информация о тестах для задач 9-11 классов

Задача	Тестов из условия	Основ-ных тестов	Баллов за один основ-ной тест	Проверяющая программа
1. Количество прямоугольников	1	10	10	**check.exe**
2. Пароль	1	20	5	**check.exe**
3. Сортировка	1	20	5	**check.exe**
4. Пропавший путь	1	20	5	**check.exe**
5. Новогодний хоровод	1	20	5	**check.exe**

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