Курс лекций по олимпиадной информатике (C)

# Лекция 6. Задачи на анализ таблиц

Михаил Густокашин, 2009

Конспекты лекций подготовлены для системы дистанционной подготовки, действующей на сайте **informatics.mccme.ru**.

При нахождении ошибок или опечаток просьба сообщать по адресу

**gustokashin@gmail.com**

Версия C от 24.11.2009

## 1 Введение

В олимпиадных задачах довольно часто встречаются задачи, где входные данные заданы в виде таблицы. Под таблицей в данном случае понимается двумерный массив, но приведённые алгоритмы также подходят для трёх и более мерных массивов.

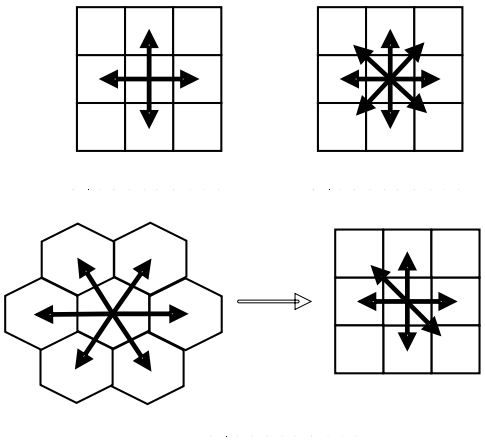

Рис. 1: Примеры связности

Введём понятие связности, т.е. определим, какие клетки являются «соседними». Если клетки считаются соседними, когда у них существует общая сторона, то это называется «4-связностью». Если клетки называются соседними, когда у них существует хотя бы одна общая точка, то это «8-связность». Также существуют более сложные способы определения соседей, например, если каждая ячейка представляет собой правильный шестиугольник или треугольник (6- и 3-связность). Иногда «соседи» могут определяться и совсем по другим параметрам, не по общим точкам или сторонам. Например, соседними могут называться клетки, на которые можно перейти ходом шахматного коня.

Почти все типы связности могут быть реализованы на квадратной матрице.

Аналогично можно определить тип связности для трёх и более мерных массивов.

Напомним о ещё одной особенности хранения массивов, которая заключается в том, что в памяти компьютера все многомерные массивы разворачиваются в одномерные. Этот факт мы уже использовали для ускорения работы функции умножения матриц, также его можно использовать для ускорения работы функций работы с таблицами. В общем случае метод выглядит так: если нам необходимо пройти по всем элементам таблицы, то наиболее внешний цикл должен идти по более раннему индексу. Для двумерной таблицы это выглядит так:

```
for (i=0; i<n; i++)
  for (k=0; k<n; k++)
   // какие-то действия с table[i][k]
```

В данной лекции мы рассмотрим несколько характерных задач, которые возникают при обработке таблиц.

## 2 Вычисление по локальным данным

Существует широкий класс задач, где по таблице необходимо подсчитать какие-либо значения для каждой ячейки. Чаще всего значения вычисляются только по данным из текущей и соседних ячеек.

Одна из часто возникающих проблем при решении задач с таблицами — нехватка памяти, для хранения всей таблицы. Таким образом, задачи, которые должны решаться только с использованием локальных данных, обладают тремя характерными признаками:

1. требуется вычисление параметра для каждой ячейки (т.е. ответ также должен быть таблицей или в качестве ответа должна выводиться ячейка, содержащая, например, максимум или минимум),
2. если таблица квадратная размером $N$, то алгоритмы со сложностью большей, чем $O(N^2)$, не проходят по времени,
3. вся таблица ($N^2$ ячеек) не укладывается в памяти.

Существует множество самых разнообразных задач на вычисление значений в ячейках по локальным данным, но можно выделить несколько приёмов, позволяющих облегчить их решение.

Первый приём — «барьерный» метод. Суть его заключается в том, что мы «окружаем» таблицу рамкой, содержащей какие-либо допустимые или специальные значения. Этот метод предназначен для того, чтобы избежать обработки частных случаев, связанных с выходом за границы массива. При этом ширина рамки зависит от того, сколько соседей мы используем для подсчёта значения в данной ячейке (чаще всего рамка имеет ширину всего в одну клетку). Рамка никак не должна влиять на решение (т.е. если, например, мы суммируем значения соседей, то рамка должна быть нулевой и т.п.).

Следующий приём является не совсем спортивным. На большинстве олимпиад ввод и вывод осуществляется с помощью файлов. Иногда бывает очень удобно просматривать таблицу несколько раз и накапливать некоторые значения. При этом если таблица не помещается в памяти, мы можем закрывать входной файл и открывать его заново (при этом порядок обхода таблицы изменить нельзя — только так, как она даётся во входных данных). Однако, во многих тестирующих системах программе передаётся уже открытый поток ввода, а следовательно такой приём применить нельзя.

Также во многих задачах требуется знать часть уже вычисленных значений соседей, и исходя из этих данных получать ответ. Здесь довольно важен порядок обхода - обычно он делается сверху вниз построчно, а начинают обход, чаще всего, с какой-либо угловой ячейки.

Рассмотрим примеры задач, где требуется вычисление значений по локальным данным.

### Программистика

Всероссийская олимпиада 2002

В Перми становится популярной игра «Программистика».

Для игры требуются плоские квадратные фишки 4-х видов, представляющие собой поле $3 \times 3$ с вырезанной центральной клеткой. В остальных клетках каждой фишки записаны числа от 1 до 8. Все виды фишек показаны на рисунке. Количество фишек каждого вида не ограничено.

| | | |
|---|---|---|
| 1 | 2 | 3 |
| 8 | | 4 |
| 7 | 6 | 5 |

| | | |
|---|---|---|
| 7 | 8 | 1 |
| 6 | | 2 |
| 5 | 4 | 3 |

| | | |
|---|---|---|
| 5 | 6 | 7 |
| 4 | | 8 |
| 3 | 2 | 1 |

| | | |
|---|---|---|
| 3 | 4 | 5 |
| 2 | | 6 |
| 1 | 8 | 7 |

Игра проводится на поле размером $N \times N$. Первоначально все клетки поля заполнены единицами.

В начале игры Магистр несколько раз случайным образом помещает произвольные фишки на игровое поле так, что фишка попадает на поле целиком, а ее центральная клетка совпадает с одной из клеток поля. После помещения очередной фишки все числа в восьми клетках игрового поля, которые перекрывает фишка, умножаются на соответствующие числа в клетках фишки, и результаты становятся новыми значениями этих клеток игрового поля.

Таким образом, после окончания процесса размещения фишек игровое поле оказывается заполненным полученными произведениями. Далее Магистр передаёт получившееся поле игроку, которому необходимо установить для каждой клетки поля, сколько раз Магистр в неё помещал центральные клетки фишек.

Требуется для каждого входного файла, содержащего полученное Магистром поле, сформировать соответствующий ему выходной файл, в $N$ строках которого содержится по $N$ чисел, показывающих, сколько раз в соответствующую клетку помещались центральные клетки фишек.

Рассмотрим верхнюю левую клетку игрового поля $(0, 0)$ — обозначим число, записанное в ней, за $X$. На неё могут оказать влияние только карточки с центром в ячейке $(1, 1)$, причём только своим левым верхним углом (т.е. числами 1, 7, 5 и 3 соответственно). Заметим, что числа 3, 5 и 7 — взаимно просты (их НОД равен 1), а это означает, что разложение числа $X$ на множители 3, 5 и 7 однозначно. Таким образом, мы можем определить количество карточек 2, 3 и 4 типа в клетке $(1, 1)$ как максимальные степени 7, 5 и 3 на которые делится число $X$. При этом следует не забывать изменять и другие клетки, которые накрывает карточка с центром в ячейке $(1, 1)$.

Чтобы окончательно вычислить ответ в ячейке $(1, 1)$ следует также определить число карточек 1 типа. Заметим, что ячейка $(1, 0)$ (обозначим ее содержимое за $Y$) изменяется только под воздействием карточек с центром в $(1, 1)$ (умножается на чётное число) и карточек с центром в $(2, 1)$ (умножается на нечётное число). После удаления карточек 2, 3 и 4 типа с центром в $(1, 1)$ умножать на чётное число эту ячейку может только карточка 1 типа с центром в $(1, 1)$ — карточки в $(2, 1)$ чётность изменить не могут. Таким образом, количество карточек 1 типа с центром в $(1, 1)$ определяется как максимальная степень 2, на которую делится число $Y$ (при этом, опять же, следует не забывать пересчитывать значения во всех ячейках, накрываемой данной карточкой).

Тот же метод можно использовать для последовательного вычисления ячеек, двигаясь во внешнем цикле по строкам, а во внутреннем — по столбцам. Предыдущая часть будет уже полностью вычислена и не окажет никакого влияния.

# 3 Кратчайшие пути в лабиринте

Ещё один большой класс задач, где данные задаются таблицей, являются задачи на поиск путей в лабиринте. При этом обычно лабиринт состоит из проходимых клеток, стен (непроходимых клеток), начальной и конечной позиции.

Рассмотрим метод поиска кратчайшего пути, который называется «волновым алгоритмом» или обходом в ширину. Такое название он получил за то, что движение происходит «волной» от начальной позиции (т.е. равномерно во все стороны) и как только волна доходит до конечной позиции, мы прекращаем работу алгоритма и восстанавливаем ответ.

Для использования алгоритма нам понадобится массив $N \times N$ для хранения длины пути в каждой ячейке (на самом деле, во многих случаях можно воспользоваться тем же массивом, в котором задаётся сам лабиринт). Кроме того, нам понадобится очередь, в которой будут храниться координаты клеток текущего шага «волны».

Поместим в начальную клетку 0 (т.е. мы можем добраться в неё за 0 шагов), проходимые клетки будем помечать, например, $-1$, а непроходимые $-2$. Занесём в очередь координаты начальной клетки и начнём выполнение алгоритма.

Алгоритм состоит в следующем: пока очередь не пуста, берём клетку из ее начала и рассматриваем ее соседей. Если соседняя клетка помечена как проходимая и ещё не посещённая (т.е. в ней находится $-1$), то записываем в неё число на 1 большее, чем в текущей клетке (нам понадобился ещё один шаг) и заносим ее в очередь.

Собственно, это и есть весь алгоритм. В конечной клетке будет записано число шагов на пути до неё от начала. Если же нам необходимо восстановить ещё и путь, то необходимо перемещаться из конечной клетки в любую соседнюю, в которой записан номер на 1 меньший, чем в текущей.

| | | | | |
|---|---|---|---|---|
| 0 | | 6 | 7 | 8 |
| 1 | | 5 | | 9 |
| 2 | 3 | 4 | | 8 |
| 3 | | 5 | 6 | 7 |

Т.е. путь будет восстановлен в обратном порядке.

Возможный вид таблицы и очереди после работы алгоритма (строка $N$ — номер шага, в самой очереди не используется, т.к. номер шага хранится в таблице и приведена для пояснения):

| $X$ | 0 | 0 | 0 | 0 | 1 | 2 | 2 | 2 | 3 | 2 | 4 | 3 | 4 | 4 | 4 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| $Y$ | 0 | 1 | 2 | 3 | 2 | 2 | 3 | 1 | 3 | 0 | 3 | 0 | 2 | 0 | 1 |
| $N$ | 0 | 1 | 2 | 3 | 3 | 4 | 5 | 5 | 6 | 6 | 7 | 7 | 8 | 8 | 9 |

В этом алгоритме мы не учитывали, что у граничных клеток нет части соседей. Этот частный случай легко обойти, если создать вокруг лабиринта барьер из непроходимых клеток. В некоторых задачах, наоборот, можно обходить лабиринт снаружи — тогда барьер должен быть проходимым.

Ещё один вариант задач с лабиринтами — когда у нас нет непроходимых клеток, но есть тонкие стенки между клетками (у одной клетки максимум 4 окружающие ее стенки). В таком случае нам нужно для каждой клетки помнить, какие у неё есть стенки. Удобнее всего делать это с помощью 4 битов — каждый бит отвечает за одну стенку:

Обычно такого рода лабиринты задаются списком стен. Чтобы получить готовое к употреблению описание лабиринта необходимо: во-первых «обнести» весь лабиринт стенками (установить для каждой граничной ячейки соответствующий бит, а для угловых — 2 бита), а во-вторых — расставить стенки (изменить по одному биту в двух

ячейках, которые разделяет данная стенка). Аналогично можно реализовать и любые другие лабиринты (с 6, 8-связностью и даже более сложные) — необходимо просто увеличить количество бит для описания стен.

Теперь на каждом шаге волнового алгоритма нам надо проверять, что соседняя ячейка не занята и две ячейки не разделяет стенка (смотреть соответствующий соседней ячейке бит). В остальной части алгоритм останется без изменений. При восстановлении пути также надо следить за тем, чтобы не ходить через стены.

Существует класс задач, в которых необходимо найти не кратчайший путь от одной клетки до остальных, а наоборот — кратчайшие пути от многих клеток до одной. Здесь можно воспользоваться инвертированием, т.е. найти пути от одной клетки до всех одним обходом в ширину, а потом просто развернуть пути в обратную сторону. Аналогично можно поступить и в случае если существует много начальных клеток и много конечных — волновые алгоритмы следует запускать из тех клеток, количества которых меньше (т.е. если начальных клеток меньше — запускаем волны оттуда и наоборот в противном случае), затем для каждой клетки следует выбрать минимум.

# 4 Система непересекающихся множеств

Перед рассмотрением следующего алгоритма работы с таблицами необходимо отвлечься и освоить одну несложную структуру данных — систему непересекающихся множеств.

В олимпиадных задачах довольно часто требуется разбить набор объектов на непересекающиеся множества (т.е. каждый объект может лежать только в одном множестве, но в одном множестве может находиться несколько объектов). Пусть объекты пронумерованы от 0 до $N-1$. В качестве идентификатора множества будем использовать также числа от 0 до $N-1$. При инициализации, обычно, все множества состоят из одного элемента, т.е. объект с номером $i$ лежит во множестве с номером $i$.

Для системы непересекающихся множеств определены две операции: `fset(x)`, возвращающая номер множества, в котором лежит элемент  и операция `sunion(x, y)`, объединяющая множества, содержащие элементы x и y в одно. Первая операция используется, обычно, для проверки того, лежат ли два элемента в одном множестве или в разных.

Эффективность различных структур будем оценивать как количество операций, необходимое для объединения $N$ множеств в одно в наихудшем для данной структуры случае.

Самым простым способом реализации является одномерный массив, где индекс задаёт номер объекта, а значение - номер множества, в котором этот объект находится. Проверка будет просто возвращать значение из запрошенной ячейки, а объединение должно проходить по всему массиву и заменять все числа  на  (или наоборот). В худшем случае каждый раз мы будем добавлять по одному элементу, таким образом, нам потребуется $N$ проходов по массиву и сложность составит $O(N^2)$.

Рассмотрим более эффективную реализацию, где кроме массива будет храниться также список элементов множества (для списка мы будем хранить указатели на начало и конец). Этот способ требует $O(N)$ дополнительной памяти. При проверке мы будем также возвращать значение из массива, а при модификации проходить по одному из списков и менять значения в массиве, а потом прикреплять этот список к концу

другого. Если мы будем делать это бездумно, то сложность также составит $O(N^2)$ — это случай, когда каждый раз длинный список будет прикрепляться к концу списка длины 1. Если ввести для каждого списка такое поле, как длина списка (оно легко пересчитывается при объединении) и приписывать каждый раз более короткий список, то для последовательного объединения всех списков длины 1 сложность получится $O(N)$. Однако, если каждый раз длины списков будут равны (худший для нас случай), мы получаем сложность $O(N \log N)$. На каждом шаге будут объединяться все пары списков равной длины, каждый раз длина списка будет увеличиваться вдвое, следовательно, общее количество шагов будет составлять $\log N$.

Рассмотрим также реализацию на стягивающихся корневых деревьях. Для каждого множества выберем элемент, который назовём «представителем». Представитель указывает сам на себя, и сначала все элементы указывают сами на себя. Для каждого дерева введём такое понятие, как высота, которое сродни длине списка и более «низкое» дерево следует прикреплять к более высокому, а в случае, если высоты были равны, высота результирующего дерева увеличится на единицу. Требования к дополнительной памяти составят $O(N)$.

Функция поиска должна идти наверх, пока не дойдёт до представителя, а функция объединения вызывает две функции поиска, а затем уже объединяет деревья. Казалось бы, такой способ не даёт никаких особых преимуществ перед списками, однако можно модифицировать функцию поиска так, чтобы при проходе по всем элементам пути до представителя она, получив указатель на представителя, для всех элементов на пути устанавливала указатель непосредственно на представителя. Это требует ещё $O(\log N)$ дополнительной памяти (эта оценка сильно завышена). Приведём реализацию системы непересекающихся множеств:

```
typedef struct {
  int rank, p;
} syst;

void init(syst *a, int n) {
  int i;
  for (i = 0; i<n; i++) {
    a[i].p = i;
    a[i].rank = 0;
  }
}

int fset(syst *a, int x) {
  if (x != a[x].p)
    a[x].p = fset(a, a[x].p);
  return a[x].p;
}
void sunion(syst *a, int x, int y) {
  if (a[fset(a, x)].rank < a[fset(a, y)].rank)
    a[fset(a, x)].p = a[fset(a, y)].p;
  else {
    a[fset(a, y)].p = a[fset(a, x)].p;
```

```
    if (a[fset(a, x)].rank == a[fset(a, y)].rank)
      a[fset(a, x)].rank++;
  }
}
```

Сложность объединения всех множеств в одно практически линейна. Оставим без доказательства этот факт и точную оценку, однако интуитивно понятно, почему операция `fset` будет выполняться с каждым разом все быстрее.

# 5 Выделение связных областей

На олимпиадах достаточно часто предлагаются задачи, в которых необходимо выделить в таблице связные области (т.е. каждой группе соприкасающихся «закрашенных» клеток присвоить уникальный номер) и модификации таких задач. Существует несколько способов решения данной задачи, которые различаются по сложности реализации и требованиям к памяти.

Простейший способ — «обход в глубину», когда мы используем рекурсивную функцию, которая помечает клетку, как уже просмотренную и вызывает себя для всех ещё не просмотренных закрашенных соседей. Пишется такая функция очень просто, но в худшем случае (когда закрашено все поле), количество рекурсивных вызовов будет $N^2$, что очень много. В целом алгоритм выделения связных областей обходом в глубины выглядит так: мы проходим всю таблицу и если встречаем закрашенную не помеченную клетку, то вызываем для неё рекурсивную функцию с новым номером (который будет номером данной области). Сложность этого алгоритма будет составлять $O(N^2)$.

Следующий способ — использование обхода в ширину, модификации волнового алгоритма. Отличается от предыдущего он только тем, что вместо вызова рекурсивной функции, новая клетка добавляется в очередь. При этом помечать клетку следует вместе с помещением в очередь (чтобы не возникало ситуаций, когда одна и та же клетка попала в очередь 2 или более раз). Требования по дополнительной памяти (для хранения очереди) у этого алгоритма будут $O(N)$.

Рассмотрим ещё одну реализацию поиска связных областей с помощью последовательного сканирования. В этом алгоритме мы будем просматривать таблицу сверху вниз, на каждом шаге нам известна уже размеченная строка (на первом шаге — строка, состоящая из не закрашенных клеток, барьер) и по ней мы будем размечать следующую, ещё не размеченную строку. Процесс разметки происходит следующим образом: мы идём по уже размеченной строке, если нам попалась клетка, относящаяся к какой-то области ($X$), то помечаем все не размеченные клетки под ней в обе стороны, как относящиеся к той же области. Если в процессе разметки клеток нижней строки над какой-либо клеткой оказалась клетка,

| | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|
| 0 | 1 | 0 | 0 | 2 | 0 | 3 | 0 | 0 | 0 |
| 0 | −1 | −1 | −1 | −1 | 0 | −1 | 0 | −1 | 0 |
| Шаг 1: нижнаяя строка не размечена | | | | | | | | | |
| 0 | 1 | 0 | 0 | 2 | 0 | 3 | 0 | 0 | 0 |
| 0 | *1* | *1* | *1* | *1* | 0 | −1 | 0 | −1 | 0 |
| Шаг 2: размечаем область 1 в новой строке. Области 1 и 2 объединяются | | | | | | | | | |
| 0 | 1 | 0 | 0 | 2 | 0 | 3 | 0 | 0 | 0 |
| 0 | 1 | 1 | 1 | 1 | 0 | *3* | 0 | −1 | 0 |
| Шаг 3: размечаем область 3 в новой строке | | | | | | | | | |
| 0 | 1 | 0 | 0 | 2 | 0 | 3 | 0 | 0 | 0 |
| 0 | 1 | 1 | 1 | 1 | 0 | 3 | 0 | *4* | 0 |
| Шаг 4: неразмеченная ячейка попадает в новую область 4 | | | | | | | | | |

помеченная, как относящаяся к другой области ($Y$), то области $X$ и $Y$ следует объединить в одну (эта ситуация возникает, например, если мы размечаем перевёрнутую букву П — в верхних строках у нас будут две разные области, а затем они объединяться перемычкой). После того, как мы выполнили проход по верхней строке, в нижней могут остаться ещё не размеченные клетки — для них следует создать новые области и разметить их. При этом смежные клетки следует помещать в одну область.

Для реализации областей (проверки принадлежности одной области и объединения областей) удобно использовать систему непересекающихся множеств, реализованную с помощью стягивающихся корневых деревьев.

```
int main(void)
{
  syst regions[MAXN*MAXN/2];
  int table[MAXN+2][MAXN+2];
  int i, j, k, n;
  int nowreg=1;
  scanf("%d", &n);
//Инициализация системы множеств
  init(regions, n*n/2);
//Создание барьера
  for (i=0; i<n+1; i++)
    table[i][0] = table[i][n+1] = table[0][i] = table[n+1][i] = 0;
//Считывание и преобразование входных данных
  for (i=1; i<=n; i++)
    for (j=1; j<=n; j++) {
      scanf("%d", &table[i][j]);
      if (table[i][j] == 1) table[i][j] = -1;
    }
//Обход таблицы
  for (i=0; i < n+1; i++) {
    for (j=0; j<n+1; j++) {
      if (table[i][j] > 0 && table[i+1][j] == -1) {
        k=j;
//Размечаем нижнюю строку налево
        while (table[i+1][k] == -1) {
          table[i+1][k] = fset(regions, table[i][j]);
          if (table[i][k] > 0 && fset(regions, table[i][k])
          != table[i+1][k])
            sunion(regions, table[i][k], table[i+1][k]);
          k++;
        }
        k=j-1;
//... и направо
        while (table[i+1][k] == -1) {
          table[i+1][k] = fset(regions, table[i][j]);
          if (table[i][k] > 0 && fset(regions, table[i][k])
          != table[i+1][k])
```

```
            sunion(regions, table[i][k], table[i+1][k]);
          k--;
        }
      }
    }
//Размечаем новые области
    for (j=0; j<n+1; j++)
      if (table[i+1][j] == -1) {
        k = j;
        while (table[i+1][k] == -1)
        table[i+1][k++] = nowreg;
        nowreg++;
      }
    }
//Вывод результатов разметки
  for (i=1; i <= n; i++) {
    for (j=1; j <= n; j++)
      if (table[i][j] != 0)
        printf("%d ", fset(regions, table[i][j]));
      else
        printf("0 ");
      printf("\n");
  }
  return 0;
}
```

Приведённая выше программа основывается на использовании системы непересекающихся множеств.

Отметим, что на самом деле, нам не нужно хранить всю таблицу для использования метода сканирования — достаточно только 2 строк. Однако это нельзя использовать даже для непосредственного вывода результатов (опять же пример с перевёрнутой П). Пользоваться тем, что мы можем не хранить всю таблицу можно только в задачах, где необходимо подсчитать некоторые параметры связных областей (например, количество элементов в максимальном и минимальном множестве), но не при непосредственном выводе разметки.

Максимальное количество связных множеств на таблице $N \times N$ составляет $\lceil N/2 \times N/2 \rceil$, но при использовании сканирования нам важны только множества в двух строках, т.е. достаточно поддерживать список свободных номеров множеств и использовать только $2 \times \lceil N/2 \rceil$ дополнительной памяти.

Метод сканирования можно также использовать и для решения других задач, которые, на первый взгляд, должны решаться другим методом. В частности, можно проверять, связаны ли две клетки (т.е. между ними существует путь в лабиринте). Для этого достаточно проверить, что они лежат в одной и той же связной области. Метод сканирования применим там, где невозможно поместить в память всю таблицу — в таком случае следует искать его модификацию, решающую конкретную задачу.