Занятие №12. Графы II.
Поиск в ширину

Тема заключительного занятия выбрана не случайно. Алгоритм поиска в ширину очень часто используется на практике. Важно помнить, что все, что мы изучили в этом Модуле и будем продолжать изучать в следующих, нужно, прежде всего, не для того, чтобы получить плюсик в таблице на informatics! Начнем сразу с практической задачи.

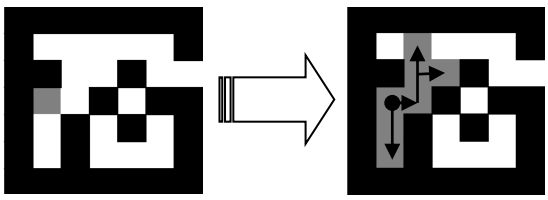
Путник находится в лабиринте. Выяснить, сможет ли он выйти из лабиринта, перемещаясь по его ходам. И, если да, подсказать ему план действий, желательно максимально короткий.

При этом предположим, что мы видим весь лабиринт, скажем, с вертолета.

(Что делать, если путник находится в подземном лабиринте и может ориентироваться потому, что видит непосредственно перед собой, мы обсудим на одном из занятий второго модуля.)

Лабиринт можно рассматривать в виде графа (доступные локации – вершины, ходы – ребра). На Занятии №10 мы говорили о том, что графы можно представлять по-разному. Для наглядности изложения представим лабиринт таблицей. Черная клетка будет означать преграду. Так же для простоты будем считать, что выход из лабиринта находится в одной заданной клетке. В нее и надо попасть.

Отметим положения путника, в которые он может попасть, серым цветом. Во-первых, это его начальное положение. На втором шаге отметим серым те клетки, которые рядом, но не черные. На третьем шаге – все те белые клетки, которые оказываются рядом с серыми (действительно, в них он может попасть за один шаг из уже серых, а значит, сможет попасть из начального положения). И так далее:

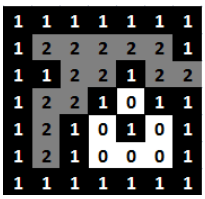


На каком-нибудь шаге мы не сможем покрасить ни одной клетки – можно останавливаться.

Сколько раз нужно повторять операцию раскрашивания в худшем случае? (См. Подсказку 12.1).

В клетки, которые остались белыми путник не сможет попасть, в серые – сможет. Если клетка выхода серая – путнику повезло.

При программировании можно использовать прямоугольный двумерный массив, черные клетки обозначить единицами, серые двойками:



Заполнять его очень просто. Пробегаемся по всему массиву и рядом с двойками нули (и только нули!) заменяем на двойки. Сколько раз? Это мы посчитали в Подсказке 12.1. Общая сложность получается $O(m^2)$, где m – это количество клеток. Достаточно медленно. Но важнее то, что мы так и не ответили на второй вопрос: как действовать путнику, чтобы выйти. Сколько шагов понадобится? Как передвигаться, чтобы выйти как можно быстрее?

На самом деле, можно легко модернизировать алгоритм, чтобы он отвечал и на эти вопросы. Достаточно в процессе ставить не только двойки, а постоянно увеличивать значения меток. На первом проходе рядом с двойками поставить тройки, на втором уже рядом с тройками ставить четверки и т.д. У нас получится такая картинка:

В клетке выхода стоит число 10. Значит путник сможет достигнуть выхода за ~~10~~ 8 шагов, мы же начинали с двойки.

Если поставить барьерные элементы – преграды - по периметру код пишется достаточно просто. Если размер лабиринта X на Y, то:

```
int k = 2;
for (int y = 1 y <=Y; y++)
{
     for (int x = 1 x <=X; x++)
     {
```

```
          if (a[y][x] == ___)
          {
               if (a[y][x - 1] == 0) {a[y][x - 1] = ___;}
               if (a[y - 1][x] == 0) {a[y - 1][x] = ___}
               if (a[y][x + 1] == 0) {a[y][x + 1] = ___}
               if (a[y + 1][x] == 0) {a[y + 1][x] = ___}
          }
     }
     ____;
}
```

Полный код приведен в Справочнике.

Такой алгоритм называют «волновым». От старта во все направления как бы распространяется волна, причем каждая пройденная волной клетка помечается как «пройденная». Волна, в свою очередь, не может проходить через клетки, уже помеченные как «пройденные» или «непроходимые». Волна движется, пока не достигнет точки финиша или пока не останется непройденных клеток. Если волна прошла все доступные клетки, но так и не достигла клетки финиша, значит, путь от старта до финиша проложить невозможно. Этот алгоритм можно использовать именно для заливки нарисованных фигур.

После достижения волной финиша, от финиша прокладывается путь до старта и сохраняется в массиве. Как его найти? **Пойдем с конца**. Найдем рядом с клеткой выхода число на единицу меньшее (в нашем случае левая клетка, в которой 9). Добавим ее в путь. Теперь ищем 8 (опять на единицу меньшее, у нас это клетка выше) и так далее, пока не дойдем до двойки. **Получаем путь в обратном порядке**. Но ведь легче искать в прямом!? Легче, но получается неверно… Почему? (см. Подсказку 12.2).

Попробуем ускорить работу алгоритма. Прежде всего, поймем причину плохой временной эффективности. Она в том, что мы каждый раз пробегаемся по всем клеткам-вешинам. Действительно, на первом шаге алгоритма серая клетка всего одна, а мы пробегаемся по всему графу в ее поисках. То же происходит, скажем, на 6-м шаге, когда мы ставим 8.

Можно хранить только нужные в данный момент вершины. Для этого используем очередь.

Запишем алгоритм поиска в ширину на естественном языке.

BFS (Breadth-first search)

Пометить стартовую вершину и добавить ее в очередь.

Пока очередь непуста

- **Взять очередную вершину из очереди**

- **Пометить все связанные с ней вершины меткой на единицу больше, чем ее собственная, и добавить их в очередь**

Этот алгоритм подходит для любого графа заданного любым способом. И он очень эффективен. Каждая вершина обрабатывается ровно дважды. При обработке вершины мы проходим по всем ребрам, связанным с ней. Суммарно за весь алгоритм просматриваются все ребра. Таким образом, сложность алгоритма поиска в ширину O(V+E). Конечно, если мы храним граф матрицей смежности, то пробегаемся в поисках соседних по всей строке. И сложность получается ____ (см. подсказку 12.2).

Запишем алгоритм программой на Java с использованием очереди, разработанной на предыдущем занятии.

Проще всего заполнить массив расстояний изначально большими числами или минус единицами – «бесконечностями» и начинать с метки 0.

BFS в графе, заданном матрицей смежности G

Количество вершин V. Массив расстояний d.

Подготовка

```
int INF = 1000*1000*1000;
for (int v = 0; v < V; v++)
{
     D[v] = INF;
}
```

Функции передается номер стартовой вершины `s` и она заполняет глобальный массив расстояний `d`.

```
static void bfs(int s)
{
     MyOueue q;
     d[s] = 0;
     q.offer(s);
     while(!q.empty())
     {
          int v = q.poll();
          for (int i = 0; i < V; i++)
          {
               if (____________________)
               {
                    d[i] = d[v] + 1;
               }
          }
     }
}
```

Применения алгоритма поиска в ширину

Прежде всего, это

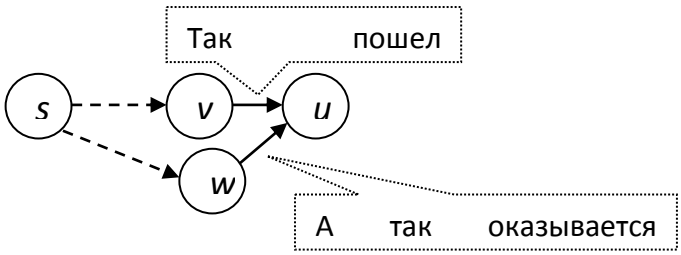
Поиск кратчайших путей из данной

Просто запустим bfs и массив d заполнится кратчайшими расстояниями.

Немного теории

Докажем, что числа в массиве **d** – это длины именно кратчайших путей. (Если в какой-то ячейке остается бесконечность – вершина недостижима.)

Допустим, что это не так. Выберем из вершин, для которых кратчайшие пути от *s* найдены некорректно, ту, настоящее расстояние до которой минимально. Пусть это вершина *u* и мы ее обработали как соседа вершины v, а предок в кратчайшем пути – вершина w (то есть надо было идти через *w*).



Так как *w* — предок *u*, то *d(s, u) = d(s, w) + 1 > d(s, w)*, а так как расстояние до *w* найдено верно, мы же выбирали u как ближайшую «неправильную», то *d(s, w) = d[w]*, и значит *d(s, u) = d[w] + 1*.

Так как *u* мы обрабатывали как соседа *v*, то *d[u] = d[v] + 1*;

Расстояние до u найдено некорректно, поэтому *d(s, u) < d[u]*. Подставляя сюда два последних равенства, получаем *d[w] + 1 < d[v] + 1*, то есть, *d[w] < d[v]*. Но это означает, что вершина *w* попала в очередь и была обработана раньше, чем *v*. Но она соединена с *u*, значит, *v* не может быть предком *u* при обходе в ширину.

Мы пришли к противоречию, следовательно, найденные расстояния до всех вершин являются кратчайшими

Поиск компонент связности

Если после заполнения массива расстояний функцией **bfs** какая-нибудь ячейка останется бесконечностью, то она недостижима из стартовой вершины, то есть принадлежит другой компоненте связности. Запустим bfs с ней в качестве стартовой вершины и обойдем другую компоненту связности и т.д. Действуя таким образом, мы обойдем уже весь граф, а в процессе можем посчитать компоненты связности. А если немного изменить функцию bfs, (как? см. Подсказку 12.4), то и получить элементы каждой из них.

```
int k = 0;// количество компонент связности
for (int i = 0; i < V; i++)
{
     if (d[i] != INF)
     {
          bfs(i);
          k++;
     }
```

```
}
```

Важно учитывать, что алгоритм поиска в ширину корректно работает только на невзвешенных графах, или, что то же, на взвешенных графах с ребрами одинакового веса.

Немного о задачах

Прежде всего, необходимо добиться «ОК» по задачам A и B. Конечно, лучше написать их самостоятельно, но можно изучить подробный разбор. Эти задачи очень важны в идейном плане. Не стоит их пропускать.

Также очень хорошо вернуться к задаче «Дерево ли?» из занятия по графам и, даже если вы ее уже решили, переписать решение с помощью поиска в ширину.

Задачи

Задача A. Длина пути в графе

В неориентированном невзвешенном графе (все ребра одинаковой длины) требуется найти длину кратчайшего пути между двумя вершинами.

Входные данные

Первым на вход поступает число N - количество вершин в графе ($1 \le N \le 100$). Затем записана матрица смежности (0 обозначает отсутствие ребра, 1 наличие ребра). Далее задаются номера двух вершин начальной и конечной.

Выходные данные

Выведите L длину кратчайшего пути (количество ребер, которые нужно пройти). Если пути не существует, выведите одно число - 1.

Пример

Входные данные	Выходные данные
5 0 1 0 0 1 1 0 1 0 0 0 1 0 0 0 0 0 0 0 0 1 0 0 0 0 3 5	3

Задача B. Путь в графе

В неориентированном графе требуется найти минимальный путь между двумя вершинами.

Формат входных данных

Первым на вход поступает число N – количество вершин в графе (1 ≤ N ≤ 100). Затем записана матрица смежности (0 обозначает отсутствие ребра, 1 – наличие ребра). Далее задаются номера двух вершин – начальной и конечной.

Формат выходных данных

Выведите сначала L – длину кратчайшего пути (количество ребер, которые нужно пройти), а затем L + 1 число – путь от одной вершины до другой, заданный своими вершинами. Если пути не существует, выведите одно число –1.

Пример

Входные данные	**Выходные данные**
5 0 1 0 0 1 1 0 1 0 0 0 1 0 0 0 0 0 0 0 0 1 0 0 0 0 3 5	3 3 2 1 5

Задача C. Задача из ЕГЭ

Вася, решая задачи демо-версии ЕГЭ, дошел до задачи B5, которая начиналась так:

«У исполнителя Калькулятор две команды:

- прибавь 3
- умножь на 4

Выполняя первую из них, Калькулятор прибавляет к числу на экране 3, а выполняя вторую, умножает его на 4.»

Далее в задаче требовалось получить из числа 3 число 57 не более, чем за 6 команд. Однако Вася заинтересовался, как можно для произвольных чисел a и b построить программу наименьшей длины получения числа b из числа a.

Напишите программу, которая по заданным числам a и b вычислит наименьшее количество команд Калькулятора, которое нужно, чтобы получить из a число b.

Формат входных данных

Вводятся два натуральных числа, не превышающих 1000 — a и b.

Формат выходных данных

Выведите наименьшее число команд, которое нужно, чтобы получить из a число b. Если число b получить нельзя, выведите –1 (минус 1).

Примеры

Входные данные	Выходные данные
3 57	5
43 57	-1
10 10	0

Задача D. Два коня

На стандартной шахматной доске (8x8) живут 2 шахматных коня: Красный и Зеленый. Обычно они беззаботно скачут по просторам доски, пощипывая шахматную травку, но сегодня особенный день: у Зеленого коня День Рождения. Зеленый конь решил отпраздновать это событие вместе с Красным. Но для осуществления этого прекрасного плана им нужно оказаться на одной клетке. Заметим, что Красный и Зеленый шахматные кони сильно отличаются от черного с белым: они ходят не по очереди, а одновременно, и если оказываются на одной клетке, никто никого не съедает. Сколько ходов им потребуется, чтобы насладиться праздником?

Формат входных данных

На вход программы поступают координаты коней, записанные по стандартным шахматным правилам (т.е. двумя символами - маленькая латинская буква (от a до h) и цифра (от 1 до 8), задающие столбец и строку соответственно).

Формат выходных данных

Требуется вывести наименьшее необходимое количество ходов, либо число -1, если кони не могут встретиться.

Пример

Входные данные	Выходные данные
a1 a3	1

Задача E. Получи дерево

Дан связный неориентированный граф без петель и кратных ребер. Разрешается удалять из него ребра. Требуется получить дерево.

Формат входных данных

Сначала вводятся два числа: n (от 1 до 100) и m – количество вершин и ребер графа соответственно. Далее идет m пар чисел, задающих ребра. Гарантируется, что граф связный.

Формат выходных данных

Выведите *N*-1 пару чисел – ребра, которые войдут в дерево. Ребра можно выводить в любом порядке.

Пример

Входные данные	Выходные данные
4 4 1 2 2 3 3 4 4 1	1 2 2 3 3 4

Задача F. Водостоки

Карту местности условно разбили на квадраты, и посчитали среднюю высоту над уровнем моря для каждого квадрата.

Когда идет дождь, вода равномерно выпадае т на все квадраты. Если один из четырех соседних с данным квадратом квадратов имеет меньшую высоту над уровнем моря, то вода с текущего квадрата стекает туда (и, если есть возможность, то дальше), если же все соседние квадраты имеют большую высоту, то вода скапливается в этом квадрате.

Разрешается в некоторых квадратах построить водостоки. Когда на каком-то квадрате строят водосток, то вся вода, которая раньше скапливалась в этом квадрате, будет утекать в водосток.

Если есть группа квадратов, имеющих одинаковую высоту и образующих связную область, то если хотя бы рядом с одним из этих квадратов есть квадрат, имеющий меньшую высоту, то вся вода утекает туда, если же такого квадрата нет, то вода стоит во всех этих квадратах. При этом достаточно построить водосток в любом из этих квадратов, и вся вода с них будет утекать в этот водосток.

Требуется определить, какое минимальное количество водостоков нужно построить, чтобы после дождя вся вода утекала в водостоки.

Формат входных данных

Во входном файле записаны сначала числа N и M, задающие размеры карты — натуральные числа, не превышающие 100. Далее идет N строк, по M чисел в каждой, задающих высоту квадратов карты над уровнем моря. Высота задается натуральным числом, не превышающим 10000. Считается, что квадраты, расположенные за пределами карты, имеют высоту 10001 (то есть вода никогда не утекает за пределы карты).

Формат выходных данных

В выходной файл выведите минимальное количество водостоков, которое необходимо построить.

Пример

Входные данные	Выходные данные
4 5 1 2 3 1 10 1 4 3 10 10 1 5 5 5 5 6 6 6 6 6	2

Задача G. Игрушечный лабиринт

Игрушечный лабиринт представляет собой прозрачную плоскую прямоугольную коробку, внутри которой есть препятствия и перемещается шарик. Лабиринт можно наклонять влево, вправо, к себе или от себя, после каждого наклона шарик перемещается в заданном направлении до ближайшего препятствия или до стенки лабиринта, после чего останавливается. Целью игры является загнать шарик в одно из специальных отверстий – выходов. Шарик проваливается в отверстие, если оно встречается на его пути.

Первоначально шарик находится в левом верхнем углу лабиринта. Гарантируется, что решение существует, и левый верхний угол не занят препятствием или отверстием.

Формат входного файла

В первой строке входного файла записаны числа *N* и *M* – размеры лабиринта (целые положительные числа, не превышающие 100). Затем идет *N* строк по *M* чисел в каждой – описание лабиринта. Число 0 в описании означает свободное место, число 1 – препятствие, число 2 – отверстие.

Формат выходного файла

Выведите единственное число – минимальное количество наклонов, которые необходимо сделать, чтобы шарик покинул лабиринт через одно из отверстий.

Примеры

Ввод	Вывод
4 5 0 0 0 0 1 0 1 1 0 2 0 2 1 0 0 0 0 1 0 0	3

Подсказки и решения. 12 Занятие.

12.1

Точно это посчитать достаточно сложно. Если преград нет вообще, то достаточно быстро – будет много соседних белых. Но можно поставить преграды так, что каждый раз будет краситься только одна клетка. Например, если расставить преграды спиралью: ???. Но тогда и клеток, которые нужно раскрашивать – белых – станет меньше. Но, скажем спираль, «съест» около половины клеток – таким образом, порядок количества белых клеток не уменьшится – и оценка будет *O(m)* раз, где *m* – это общее количество клеток.

12.2

Путь может получиться не туда. На рисунке занятия мы можем случайно уйти вниз – в нижнюю тройку.

12.3

$O(V^2)$. Действительно, при обработке каждой вершины мы просматриваем всю строку матрицы смежности в поиске соседних вершин.

12.4

Например, печатать номер вершины при добавлении ее в очередь.

12.5

Это означает, что в графе есть цикл. Действительно, оказывается, что до вершины **`B`** можно дойти из **`A`** несколькими путями. Любые два их них составляют цикл.

12.6

```
v + 3 <= 1000
```

чтобы не «вылететь» за пределы массива и не зациклиться. Вершины – числа большие тысячи по ограничениям задачи нас не интересуют – можно считать? что их нет.

12.7

Если граф связный, поиск в ширину обойдет все вершины. Для каждой будет добавлено одно ребро. Значит ребер в новом графе будет N-1. Граф, образованный этими ребрами, останется связным. По определению это будет дерево.

Разбор. Занятие 12

12A. Длина пути в графе

Подробно рассмотрим, как решать эту задачу поиском в ширину.

Во-первых, надо считать граф. Он задан стандартно – матрицей смежности. Поэтому делается это стандартно – как в занятии «Графы 1». Получаем заполненный матрицей смежности двумерный квадратный массив `m` размером `N`x`N`.

Так же объявляем массив кратчайших расстояний `d` на `N` ячеек и заполняем его минус единицами.

Чтобы не путаться с нумерацией, удобно создавать массивы на `N` *+ 1 и просто нулевые ячейки не использовать.*

Важно объявить переменную `N`, массив `m` и массив кратчайших расстояний `d` статическими на уровне класса, потому что с ними будет работать и функция `main` и функция `bfs`.

После считывания матрицы смежности считываем A и B – номера начальной и конечной вершин, помечаем в массиве расстояний, что расстояние до A из самой вершины A – ноль, и запускаем bfs

```
int A = in.nextInt();
int B = in.nextInt();
d[A] = 0;
bfs(A);
```

Эта функция фактически и сделает всю работу. То есть заполнит массив кратчайших расстояний от A до всех вершин. Нам останется только вывести ответ – значение `d[B]`.

Рассмотрим подробно функцию bfs.

```
static void bfs(int v)
{// ...
```

Она принимает номер стартовой вершины и ее задача заполнить массив кратчайших расстояний `d`.

Во-первых в ней создается очередь `q` и в нее закладывается стартовая вершина `v`.

```
MyQueue q = new MyQueue(100);
q.offer(v);
```

Естественно, для того, чтобы этот код заработал, надо добавить к программе класс MyQueue, разработанный на занятии «Стеки и очереди».

И начинаем цикл.

Пока очередь непуста вынимаем очередную вершину, и для всех соседних с ней и еще непросмотренных (в массиве расстояний для них лежит значение -1) ставим в массив расстояний значение на один большее и закладываем в очередь:

```
while (!q.empty())
{
     // берем вершину из очреди
     v = q.poll();
     // по всем вершинам в поисках соседних
     for (int i = 1; i <= N; i++)
     {
          // есть ребро – соседняя и непросмотренная
          if (m[v][i] == 1 && d[i] == -1)
          {
               // новая вершина дальше на одно ребро
               d[i] = d[v] + 1;
               // закладываем ее в очередь
               q.offer(i);
          }
     }
}
```

12B. Путь в графе

Эта задача – продолжение задачи A. Будем считать, что у нас есть уже массив d, заполенный кратчайшими расстояниями.

Выведем d[B], как и в задаче A. И, если это не минус единица, – путь.

Покажем, как вывести сам путь.

Во-первых, мы получим его в обратном порядке – значит надо его сохранить в массив, а потом вывести. Создадим массив way на d[B] + 1 ячейку. Именно столько будет вершин в пути.

```
int[] way = new int[d[B]+1];
```

Запомним вершину B как конец пути.

```
way[0] = B;
```

и пойдем циклом по соседним вершинам в любую сторону уменьшения кратчайших расстояний. Скажем, в `d[B]` стоит 5. Тогда ищем вершину `j` соседнюю с B, в которой

`d[j]` равно 4, то есть на единицу меньше. (Кстати, что означает, если таких вершин несколько? Подсказка 12.5). Дальше то же самое делаем с вершиной `j`, и т.д.

```
// начинаем с вершины B
int v = B;
// по всему пути
for (int i = d[B]; i >= 0; i--)
{
     // записываем в путь
     way[i] = v;
     // по всем вершинам в поисках соседних
     for (int j = 1; j <= N; j++)
     {
          // есть ребро – соседняя и на один ближе
          if (m[v][j] == 1 && d[j] == d[v] - 1)
          {
                // к новой вершине j
                v = j;
                // заканчиваем поиск
                break;
          }
     }
}
```

Путь получен. Остается его вывести:

```
for (int i = 0; i <= d[B]; i++)
{
     out.print(way[i] + " ");
}
```

12C. Задача из ЕГЭ

В этой задаче нужно только догадаться где же тут граф, что можно считать за вершины, а что за ребра. Дальше - практически стандартный bfs.

Вершины это числа. По ограничениям задачи можно считать, что в графе 1000 вершин. Ребро есть, когда одно число получается из другого прибавлением трех или умножением на четыре. Специально хранить граф стандартным образом нет необходимости. У каждой вершины с номером v две соседних с номерами v + 3 и v * 4.

Небольшая модификация bfs выглядит так: в цикле «пока очередь не пуста» обходим соседние:

```
v = q.poll();
if (_____________ && d[v + 3] == -1)
{
   d[v+3] = d[v] + 1;
   q.offer(v + 3);
}
```

```
// для вершины v * 4 аналогично
```

В программе пропущено важное условие. Какое? (Подсказка 12.6)

12D. Получи дерево

Сначала нужно считать граф, например, в матрицу смежности, как это делалось на первом занятии по графам. (задача «От списка ребер к матрице смежности»).

Обойдем наш граф в ширину из произвольной вершины. В процессе обхода будем выводить ребро из обрабатываемой вершины в «новую». Полученное дерево называется деревом обхода в ширину. Кстати, почему это будет дерево и почему оно будет содержать все вершины? (Подсказка 12.7)

Таким образом, практически единственное изменение в функции bfs такое

```
if (m[v][i] == 1 && d[i] == INF)
{
     // новая соседняя непросмотренная вершина.
     // Выводим ребро в нее.
     out.println(v + " " + i);
```

12E. Игрушечный лабиринт

Изюминка этой задачи в том, чтобы понять, что такое соседняя вершина. Здесь соседние вершины - это крайние в пути по лабиринту.

Читаем данные и заполняем барьерными значениями. Это сильно упростит «катание до бортика»:

```
N = in.nextInt();
M = in.nextInt();

m = new int[N+2][M+2];
d = new int[N+2][M+2];
for (int y = 0; y <= N+1; y++)
{
     for (int x = 0; x <= M+1; x++)
     {
          if (x == 0 || y == 0 || x == M+1 || y == N+1)
          {
               m[y][x] = 1;
          }
          else m[y][x] = in.nextInt();
          d[y][x] = -1;
     }
}
```

Ограничения в задаче позволяют решать ее без очереди, многократно проходя по массиву. Как только в процессе обхода находим 2 – отверстие, печатаем ответ и выходим из цикла при помощи переменной finish

```
d[1][1] = 0;
int k = 0;
boolean found = true;
boolean finish = false;
while (found && !finish)
{
     found = false;
     for (int y = 1; y <= N && !finish; y++)
     {
          for (int x = 1; x <= M && !finish; x++)
          {
               if (d[y][x] == k)
               {
                    int i = 0;
                    // катим до стенки или до отверстия
                    for (i = 1; m[y][x + i] == 0; i++){};
                    // если докатили до отверстия
                    if (m[y][x + i] == 2)
                    {
                         out.print(k + 1);
                         finish = true;
                    }
                    // в три другие стороны аналогично
                    for (i = 1; m[y][x - i] == 0; i++){};
                    // ...
               }
               k++;
          }
     }
}
```

12F. Два коня

В этой задаче две сложности. Идейная и техническая.

Во-первых, надо понять, как вообще решать. Поиск в ширину мы делали из одной клетки до всех, а в этой задаче *два* коня перемещаются одновременно. Поиск в ширину ищет кратчайший путь, а тут это необязательно. Например, один конь может несколько раз оказываться на одной и той же клетке, прежде чем до нее «доскачет» другой.

Вместо того, чтобы отправлять коней навстречу друг другу, будем считать, что второй стоит на месте, а первый прыгает к нему. Если в кратчайшем пути окажется четное количество звеньев – все хорошо – одну половину «пропрыгает» первый, вторую – второй. И это будет ближайшим моментом времени! Если нечетное – наоборот, встретиться они не смогут. Действительно, при прыжке конь каждый раз меняет цвет

клетки шахматной доски. Если количество нечетное – то начальные клетки разных цветов. За одинаковое количество шагов они обязательно окажутся на клетках разных цветов.

Итак, запустим bfs из первой клетки и выясним длину кратчайшего пути до второй. Если она нечетна – ответ -1, если четна – ответ половина длины.

Теперь о реализации

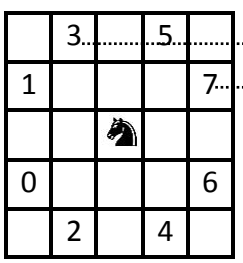
Первая задача – считать данные. Координаты задаются символами. Считаем две строки и символы переведем в координаты (будем нумеровать, начиная с нуля.)

```
int x1 = s1.charAt(0)-'a';
int y1 = s1.charAt(1)-'0' - 1;
int x2 = s2.charAt(0)-'a';
int y2 = s2.charAt(1)-'0' - 1;
```

Вторая техническая задача – не запутаться в координатах соседних вершин. Их много – восемь, при этом надо учитывать края доски.

Заведем для простоты массивы dx и dy на восемь ячеек, в них положим смещения соответствующие различным ходам:

```
int[] dx = new int[]{-2, -2, -1, -1,  1, 1,  2, 2};
int[] dy = new int[]{-1,  1, -2,  2, -2, 2, -1, 1};
```



Теперь можно обрабатывать все 8 вершин в цикле.

Покажем заодно, как обойтись без класса очереди. В этой задаче, как, впрочем, и практически во всех задачах на поиск в ширину, *вершина закладывается в очередь один раз и один раз вынимается из нее*. Достаточно просто завести массив на количество вершин (в этой задаче в два раза больше для хранения двух координат) и работать с ним. *Тогда очередь не успеет переполниться*:

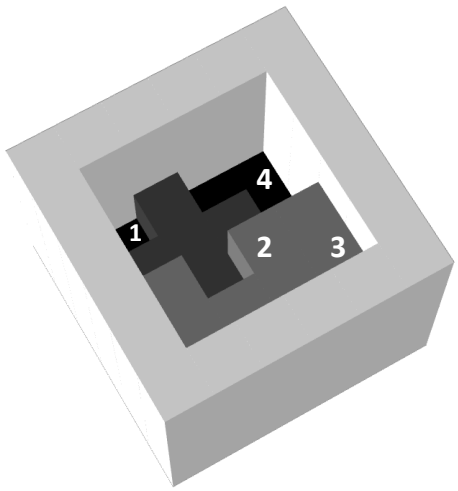
```
// d[8][8] – массив кратчайших расстояний, заполненный -1
d[y1][x1] = 0;
int[] q = new int[8*8*2];
int front = 0, rear = 0; // начало и конец очереди
// закладываем первую вершину
q[rear] = x1; rear++;
q[rear] = y1; rear++;
// пока очередь непуста
```

```
while (front < rear)
{
    // вынимаем очередную вершину
    int ux = q[front]; front++;
    int uy = q[front]; front++;
    // по соседним вершинам
    for (int i = 0; i < 8; i++)
    {
        int vx = ux + dx[i];
        int vy = uy + dy[i];
        // учитываем границы
        if ( vx >= 0 && vx < 8 && vy >= 0 &&
             vy < 8 && d[vy][vx] == -1)
        {
             d[vy][vx] = d[uy][ux] + 1;
             q[rear] = vx; rear++;
             q[rear] = vy; rear++;
        }
    }
}
```

Остается вывести ответ

```
if (d[y2][x2] % 2) out.println("-1");
else out.println(d[y2][x2] / 2);
```

12G. Водостоки



Для того, чтобы хорошо представить себе ситуацию, нарисуем 3D картинку.

Видно, что ответ для нее - 2. Две области, обозначенные черным. Как это определить?

Для квадратов 1 и 2 ситуация ясна сразу. Все клетки рядом квадратом 1 выше – там будет вода. Рядом с квадратом 2 есть квадрат с меньшей высотой – из области 2 вода будет сливаться. А вот с квадратами 3 и 4 непонятно. Например, рядом с квадратом 3 есть квадраты выше и *равные по высоте*. Будет ли вода сливаться? Это можно понять, «объединив» все светлосерые квадраты в одно «плато». И так как квадраты 2 и 3 в одной области, то и из 3 будет сток.

Объединять соседние квадраты с одинаковой высотой в области можно при помощи обхода в ширину. Вершинами графа будут сами квадраты, а ребра, будем считать, есть только между соседними квадратами одинаковой высоты. Таким образом, нам надо

будет посчитать количество компонент связности, в которых нет вершин-квадратов с соседями меньшей высоты.

Как и во многих других задачах на клетчатых полях нет необходимости хранить граф особо. Достаточно завести двумерный массив и считать данные в него.

Заведем также статический массив **used** (аналог **d**) – для этой задачи достаточно boolean – нам важно только просмотрен уже квадрат или нет, нас не интересует «расстояние» до него.

```
// Плюс 2 по каждой стороне для барьеров
static int[][] a = new int[102][102];
static boolean used[][] = new boolean[102][102];
```

и заполним их барьерами при чтении карты высот

```
for (int y = 0; y < N + 2; y++)
{
     for (int x = 0; x < M + 2; x++)
     {
          // ставим барьеры
          if (x == 0 || y == 0 || x == M + 1 || y == N + 1)
          {
               a[y][x] = 10000;
               used[y][x] = true;
          }
          else a[y][x] = in.nextInt();
     }
}
```

Вершина имеет две координаты, будем класть и вынимать из очереди вершины по парам.

```
static int bfs(int x, int y)
{
     // пока сток не найден
     int water = 1;
     // заводим очередь вдвое большего размера
     MyQueue q = new MyQueue(101*101*2);
     used[y][x] = true;
     q.offer(y); q.offer(x);
     while (!q.empty())
     {
          y = q.poll();
          x = q.poll();
```

Проверяем соседние вершины на низкую высоту – ищем сток.

```
          if (a[y-1][x] < a[y][x] || a[y+1][x] < a[y][x] ||
            a[y][x - 1] < a[y][x] || a[y][x + 1] < a[y][x])
           {
                 //сток найден!
                 water = 0;
           }
```

И стандартным образом соседние не просмотренные соседние вершины (в данной задаче соседние одинаковой высоты) помечаем и добавляем в очередь

```
          if (!used[y-1][x] && a[y-1][x] == a[y][x])
          {
               q.offer(y - 1);
               q.offer(x);
               used[y - 1][x] = true;
          }
          // ...аналогично для других трех соседних квадратов
     }
```

И возвращаем результат – обнаружен сток для очередного плато или нет.

```
     return water;
}
```

В `main` – почти стандартный подсчет количества компонент связности:

```
int answ = 0;
// по всем клеткам
for (int y = 1; y <= N; y++)
{
     for (int x = 1; x <= M; x++)
     {
          // если непросмотрена – начинаем новую компоненту
          if (!used[y][x])
          {
               answ += bfs(x, y);
          }
     }
}
```

Остается только вывести ответ.

Занятие 12.Справочник

Волновой алгорим без очереди

См. подробное условие в тексте Занятия №12.

```
int k = 2;
for (int i = 0 i <= X * Y; i++)
{
     for (int y = 1 y <= Y; y++)
     {
          for (int x = 1 x <= X; x++)
          {
               if (a[y][x] == k)
               {
                    if (a[y][x - 1] == 0) {a[y][x - 1] = k + 1;}
                    if (a[y - 1][x] == 0) {a[y - 1][x] = k + 1;}
                    if (a[y][x + 1] == 0) {a[y][x + 1] = k + 1;}
                    if (a[y + 1][x] == 0) {a[y + 1][x] = k + 1;}
               }
          }
          k++;
     }
}
```

BFS (Breadth-first search)

На естественном языке

Пометить стартовую вершину и добавить ее в очередь.

Пока очередь непуста.

- **Взять очередную вершину из очереди**
- **Пометить все связанные с ней вершины меткой на единицу больше, чем ее собственная и добавить их в очередь**

В графе, заданном матрицей смежности G

Количество вершин V. Массив расстояний d.

Подготовка

```
int INF = 1000*1000*1000;
for (int v = 0; v < V; v++)
{
     d[v] = INF;
}
```

Функции передается номер стартовой вершины **s** и она заполняет глобальный массив расстояний **d**.

```
static void bfs(int s)
{
     Oueue q;
     d[s] = 0;
     q.offer(s);
     while(!q.empty())
     {
          int v = q.poll();
          for (int i = 0; i < V; i++)
          {
               if (G[v][i] == 1 && d[i] == INF)
               {
                    d[i] = d[v] + 1;
               }
          }
     }
}
```

Подсчет количества компонент связности при помощи bfs

```
int k = 0;// количество компонент связности
for (int i = 0; i < V; i++)
{
     if (d[i] != INF)
     {
          bfs(i);
          k++;
     }
}
```