Разбор задач Одиннадцатой Интернет-олимпиады

Автор: Максим Буздалов

Введение

В базовой номинации Одиннадцатой Интернет-олимпиады сезона 2006-2007 участникам было предложено для решения 8 задач. В олимпиаде приняло участие 55 команд, из них 50 решили хотя бы одну задачу.

Наиболее простой оказалась задача «F. Сравнение дробей» — ее решили 50 команд. Наиболее сложными — задачи «B. Автоспорт» и «G. Три поросенка» — их не решила ни одна команда.

Условия задач, результаты олимпиады, тесты и решения жюри можно найти на сайте интернет-олимпиад `http://neerc.ifmo.ru/school/io`.

Задача A. Задача о назначениях

При ограничениях, данных в условии задачи ($N \leq 10$), ее возможно решать полным перебором всех вариантов. Рассмотрим *перестановку* — упорядоченный набор различных целых чисел от 1 до N. Каждой перестановке однозначно соответствует некоторая схема назначения работ рабочим. А именно, назначим i-тому рабочему работу с номером P_i, где P_i — i-тый элемент перестановки.

Для решения задачи необходимо перебрать все возможные перестановки чисел $1 \ldots N$, для каждой такой перестановки P посчитать стоимость назначения и вывести в выходной файл минимальную из них. Стоимость назначения считается так:

$$C_P = \sum_{i=1}^{N} C_{i,P[i]}$$

Для такого решения существует множество способов реализации. Приведем два из них.

Первый способ существенно использует генерацию следующей перестановки.

Пусть значения C_{ij} хранятся в двумерном массиве `C`, а текущая перестановка хранится в одномерном массиве `P`. Также пусть существует функция `nextPermutation(K)`, которая по текущей перестановке длиной K в массиве `P` строит следующую в некотором порядке перестановку и записывает ее в этот же массив `P`. Эта функция будет возвращать `true`, если ей удалось построить следующую перестановку, и `false`, если данная перестановка была последней.

Кроме того, в переменной `best` будем хранить наименьшую достигнутую стоимость назначения, а в переменной `current` — текущую стоимость назначения. Тогда основной фрагмент программы будет выглядеть, например, так:

```
for i := 1 to N do begin
    P[i] := i;
end;
repeat
    current := 0;
    for i := 1 to N do begin
        current := current + C[i][P[i]];
    end;
    if current < best then begin
        best := current;
    end;
until not nextPermutation(N);
```

Саму процедуру `nextPermutation` можно организовать следующим образом. Условимся, что мы будем перебирать перестановки в лексикографическом порядке, то есть, каждая следующая перестановка будет лексикографически больше предыдущей. Это не обязательно при решении данной задачи, но может оказаться полезным при решении других задач.

Рассмотрим текущую перестановку P. В этой перестановке несколько последних элементов идут в порядке убывания. Если рассмотреть эти элементы, то можно убедиться, что следующей в указанном смысле перестановки невозможно добиться, переставляя каким-либо образом только эти элементы. Пусть рассматриваемые элементы имеют номера $P_k, P_{k+1} \dots P_N$.

Если $k = 1$, то не существует перестановки, лексикографически большей чем текущая, поэтому функция должна вернуть `false`.

Иначе, среди элементов $P_k \dots P_N$ существует минимальный элемент, превосходящий P_{k-1}. Пусть он имеет номер m. Переставим элементы P_{k-1} и P_m местами. Заметим, что первые $k-1$ элементов искомой перестановки соответствуют элементам $P_1 \dots P_{k-1}$. Остается отсортировать элементы $P_k \dots P_N$ в порядке возрастания, чтобы получить искомую перестановку.

Обратим внимание на то, что в данный момент они расположены в порядке убывания, и чтобы отсортировать их в порядке возрастания, достаточно перевернуть эту последовательность.

После завершения вышеперечисленных операций вернем `true`.

Приведем текст реализации процедуры `nextPermutation`:

```
function nextPermutation(N: integer): boolean;
var
    i, j, k, t: integer;
begin
    i := N;
    while (i > 1) and (P[i - 1] > P[i]) do begin
        dec(i);
    end;
    if i = 1 then begin
        result := false;
    end else begin
        k := i;
        for j := i + 1 to N do begin
            if (P[j] > P[i - 1]) and (P[j] < P[k]) then begin
                k := j;
            end;
        end;
        t := P[i - 1];
        P[i - 1] := P[k];
        P[k] := t;

        j := N;
        while i < j do begin
            t := P[i];
            P[i] := P[j];
            P[j] := t;
            inc(i);
            dec(j);
        end;
        result := true;
    end;
end;
```

Отметим, что в стандартной библиотеке языка `C++` существует функция `next_permutation`, вы-

полняющая вышеописанные действия.

Второй способ перебирает перестановки рекурсивно.

Заведем массив `W` логических величин. `W[i] = true` тогда и только тогда, когда в построенной части перестановки содержится число i. Напишем рекурсивную процедуру `go(m, k)`, в которой m — это длина построенной перестановки, а k — это стоимость назначения для построенной части перестановки. Процедура пытается добавить к перестановке в конец все возможные числа и вызывает сама себя.

```
procedure go(m, k: integer);
var
    i: integer;
begin
    if m = N then begin
        if best > k then begin
            best := k;
        end;
    end else begin
        for i := 1 to N do begin
            if not W[i] then begin
                W[i] := true;
                go(m + 1, k + C[m + 1][i]);
                W[i] := false;
            end;
        end;
    end;
end;
```

Чтобы перебрать все перестановки, нужно сделать вызов `go(0, 0)`.

Второй способ несколько проще для понимания и написания, чем первый. С другой стороны, он менее универсален, поскольку первый способ позволяет, например, перебрать несколько перестановок, начиная с данной, в то время как второй способ менее «управляем» в этом отношении.

Отметим особо то, что эта задача имеет решение, не сводящееся к полному перебору всех возможных вариантов. Данную задачу можно интерпретировать как поиск минимального по весу полного паросочетания в полном двудольном графе, если всех рабочих поместить в левую долю графа, все работы — в правую, а весом ребра между рабочим i и работой j назначить величину C_{ij}. Прочитать о подобных алгоритмах можно в [2].

Задача B. Автоспорт

Будем сначала пытаться решать эту задачу прямым моделированием. А именно, будем приписывать каждой машине ее координаты и вектор скорости и через малые промежутки времени изменять координаты всех машин в соответствии с их скоростями, а также изменять скорости машин в результате их взаимодействия. В результате некоторых взаимодействий некоторые машины сойдут с дистанции согласно правилам гонки.

Такое решение для данной задачи в чистом виде неприменимо. Действительно, если шаг по времени будет слишком большим, то мы можем «не заметить» некоторые столкновения или наоборот, принять близко проходящие машины за сталкивающиеся. Если же он будет меньше, то время работы решения будет чересчур большим.

Заметим, что скорость машины, а также присутствие ее в гонке, может изменяться только в результате одного из следующих событий:

- машина врезалась в борт трассы.
- две или более машин столкнулись в одной точке движения.

Это наводит на мысль о том, что можно моделировать изменения системы не через фиксированные промежутки времени, а от одного события к другому, в порядке возрастания времени события. Разовьем эту мысль в алгоритм решения задачи.

Каждое из возможных событий будет считаться независимым, то есть, если некоторые машины участвуют в событии, то остальные машины не рассматриваются. Это упрощает процесс нахождения возможных событий, но некоторые из них не будут осуществлены в реальности.

Так как для каждой машины заданы начальные координаты (x_i, y_i) и вектор скорости (vx_i, vy_i), то в любой момент времени можно определить положение машины на плоскости как функцию от времени:

$$x = x_i + vx_i \cdot t$$
$$y = y_i + vy_i \cdot t$$

Строго говоря, в общем случае эти уравнения описывают движение машины только на части временного интервала, в остальные моменты времени машина покоится после участия в столкновении. Но для упрощения расчетов мы пренебрегаем этим, считая при нахождении времени некоторого события, что иные события не происходили.

С помощью формул легко найти время событий «машина врезалась в борт трассы» из следующих уравнений:

$$0 = y_i + vy_i \cdot T_d$$
$$W = y_i + vy_i \cdot T_u$$

В этих уравнениях T_d — время столкновения с нижним бортом (прямой $Y = 0$), T_u — время столкновения с верхним бортом (прямой $Y = W$).

Вышеприведенные уравнения являются линейными. Такие уравнения могут иметь одно решение, ни одного решения или бесконечно много решений.

Заметим, что из условия задачи ни одно из уравнений не сможет иметь бесконечного числа решений. Действительно, это возможно только тогда, когда машина изначально находилась либо на борту трассы, либо на линии финиша.

Некоторые из этих уравнений могут не иметь решения, тогда соответствующее событие никогда не состоится.

Единственное решение уравнения может быть либо положительным, либо отрицательным (из условия задачи следует, что нулю оно равняться не может). В случае положительного решения событие вполне может состояться, тогда мы вводим его в рассмотрение. При отрицательном же решении событие состояться не может, так как ему соответствует отрицательное время события.

Для регистрации столкновений необходимо решить систему линейных уравнений для каждой пары машин:

$$\begin{cases} x_i + vx_i \cdot T_{ij} = x_j + vx_j \cdot T_{ij} \\ y_i + vy_i \cdot T_{ij} = y_j + vy_j \cdot T_{ij} \end{cases}$$

Если система имеет решение (оно будет единственным, так как по условию задачи машины в начале соревнований не совпадают), то соответствующие машины могут столкнуться. Введем все такие события в рассмотрение.

Будем хранить события как записи, в которых хранятся время и тип события, а также участвующие в событии машины (одна или две).

```
type Event = record
    time: extended;
    border: boolean;
    car1, car2: integer;
end;
```

В описанной выше записи `border = true`, если событие является столкновением с бортом, и `false` в случае столкновения двух машин.

С каждой машиной будем ассоциировать время, в которое она была уничтожена. Будем хранить эти величины в массиве `destroyTime` значений вещественного типа. Если машина не была уничтожена, то будем хранить в соответствующем элементе массива значение «плюс бесконечность».

Отсортируем все события по времени. Далее будем брать события по одному, в порядке возрастания времени, и обновлять время уничтожения машин, участвующих в событии. Если событие является столкновением с бортом, то время уничтожения машины делается равным минимуму из текущего времени ее уничтожения и времени события. Иначе, если время уничтожения *обеих* машин *не превышает* времени события, то уничтожаем обе машины:

```
if event.border then begin
    destroyTime[car1] := min(destroyTime[car1], event.time);
end else begin
    if not less(destroyTime[car1], event.time)
       and not less(destroyTime[car2], event.time) then begin
        destroyTime[car1] := event.time;
        destroyTime[car2] := event.time;
    end;
end;
```

Обратим внимание на то, что при сравнении величин вещественного типа категорически не рекомендуется употреблять обычные операции сравнения типа $<$, $\leq$, $>$, $\geq$, $=$, $<>$, так как арифметически одинаковые результаты, полученные различными путями, могут различаться в представлении в памяти компьютера.

Например, упомянутая в программе функция `less`, может быть реализована так:

```
const EPS = 1e-9;

function less(a, b: extended): boolean;
begin
    result := a + EPS < b;
end;
```

где `EPS` — точность сравнения.

После того, как мы перебрали все события, каждой машине будет ассоциировано время, равное времени ее уничтожения.

Переберем все машины и проверим, пересекла ли она линию финиша прямую. Это можно сделать, подставив время ее уничтожения в формулы для вычисления координат. Если машина при этом находится *строго* справа от финишной прямой, то она пересекла линию финиша.

Среди всех машин, пересекавших линию финиша, выберем все те, которые сделали это раньше всех, и выведем их в ответ. Время финиша можно определить из уравнения:

$$L = x_i + vx_i \cdot T_f$$

Заметим, что максимальное количество событий пропорционально квадрату числа машин, то есть, около миллиона для данных в задаче ограничений $N \leq 1000$. Следовательно, для сортировки событий по времени нельзя было использовать алгоритмы сортировки, работающие за квадратичное время, иначе время работы решения составляло бы $O(N^4)$.

Задача C. Скидка

Заметим, во-первых, что если число товаров n меньше m, то скидкой воспользоваться не удастся. В этом случае ответом будет являться суммарная стоимость всех товаров (обозначим ее S).

Иначе переберем все возможные тройки различных товаров. Для каждой такой тройки рассмотрим ее суммарную стоимость C, и если она не превосходит c, то можно воспользоваться скидкой. Стоимость покупки в этом случае будет равна $S - C/2$. Будем запоминать в отдельной переменной

наименьшее достигнутое значение стоимости покупки. По окончании перебора в этой переменной и будет хранится ответ на задачу.

Все числа, с которыми приходится оперировать в задаче, полуцелые. Действительно, сумма стоимостей товаров — целое число, а скидка всегда полуцелая. То есть, можно решать задачу в целых числах, храня в переменных не сами стоимости, а их удвоенные значения, и к вещественным числам прибегать только при выводе ответа.

Заметим, что суммарная стоимость товаров S может быть порядка 10^{10}, то есть, необходимо пользоваться либо 64-битными целочисленными типами (`int64` в Delphi, `long long` в C/C++, `long` в Java), либо вещественными числами не менее чем двойной точности (`extended` в Delphi, `double` в C, C++ и Java).

Задача D. Сумма делителей

Найдем разложение числа x на простые множители:

$$x = p_1^{d_1} \cdot p_2^{d_2} \cdot \ldots \cdot p_k^{d_k}$$

где $p_1 \ldots p_k$ — различные простые числа, $d_1 \ldots d_k$ — неотрицательные целые числа.

Пусть мы нашли сумму всех делителей, простыми делителями которых являются числа $p_1 \ldots p_{n-1}$, и она равна S_{n-1}. Тогда все делители числа x, имеющие простые делители $p_1 \ldots p_n$, будут иметь вид

$$D_n = D_{n-1} \cdot p_n^a$$

где D_{n-1} — некоторый делитель числа x, имеющий простые делители $p_1 \ldots p_{n-1}$, $0 \le a \le d_n$.

Тогда сумма S_n вычисляется по формуле

$$S_n = S_{n-1} \cdot (1 + p_n + p_n^2 + \ldots + p_n^{d_n}) = S_{n-1} \cdot (\frac{p_n^{d_n+1} - 1}{p_n - 1})$$

Согласно этой схеме вычислений, нетрудно подсчитать и сумму всех делителей.

О вычислении и свойствах сумм степеней делителей числа можно прочитать (к сожалению, только на английском языке) в [4].

Задача E. Поток в сети

Для решения этой задачи достаточно просто проверить, выполняются ли указанные в условии задачи свойства потока в сети для данной сети и функции f.

Выполнение свойства подчиненности пропускной способности можно проверить для каждой пары вершин, например, следующим образом:

```
for i := 1 to N do begin
    for j := 1 to N do begin
        if not (f[i][j] <= c[i][j]) then begin
            correct := false;
        end;
    end;
end;
```

Для проверки антисимметричности можно организовать такой цикл:

```
for i := 1 to N do begin
    for j := 1 to N do begin
        if not (f[i][j] = -f[j][i]) then begin
            correct := false;
        end;
    end;
end;
```

Для проверки свойства сохранения потока для вершины v нужно сложить по всем вершинам u значения функции потока $f[v][u]$ и проверить, равен ли результат нулю. Разумеется, это необходимо делать для всех вершин, кроме истока (его номер равен 1) и стока (его номер равен N).

```
for i := 2 to N - 1 do begin
    sum := 0;
    for j := 1 to N do begin
        sum := sum + f[i][j];
    end;
    if sum <> 0 then begin
        correct := false;
    end;
end;
```

Во всех вышеперечисленных примерах кода `f` — двумерный массив, хранящий значения функции f, `c` — двумерный массив, содержащий значения пропускной способности. `correct` — переменная логического типа, вначале хранящая значение `true` и определяющая, является ли функция f потоком.

Про сети, потоки и связанные с ними задачи можно прочитать в [1].

Задача F. Сравнение дробей

Приведем две данные обыкновенные дроби к общему знаменателю:

$$\frac{a_1}{b_1} = \frac{a_1 \cdot b_2}{b_1 \cdot b_2}$$

$$\frac{a_2}{b_2} = \frac{a_2 \cdot b_1}{b_1 \cdot b_2}$$

При сравнении дробей с одинаковым знаменателем достаточно сравнивать их числители. Например, условие

$$\frac{a_1}{b_1} < \frac{a_2}{b_2}$$

эквивалентно условию

$$a_1 \cdot b_2 < a_2 \cdot b_1$$

Реализующая эту логику программа может выглядеть следующим образом:

```
var
    a1, b1, a2, b2: integer;
begin
    reset(input, 'fraction.in');
    rewrite(output, 'fraction.out');
    readln(a1, b1);
    readln(a2, b2);
    if a1 * b2 < a2 * b1 then begin
        writeln('Less');
    end else if a1 * b2 = a2 * b1 then begin
        writeln('Equal');
    end else begin
        writeln('More');
    end;
end.
```

Задача G. Три поросенка

Рассмотрим несколько случаев входных данных для этой задачи, что приведет нас к лучшему ее пониманию.

Во-первых, если число N простое, т.е. делится только на себя и единицу, то подобрать такие $A, B \geq 2$, что $A \cdot B = N$, невозможно. Следовательно, в случае простого числа всегда выигрывает первый поросенок. Заметим, что числа 2 и 3 простые.

Во-вторых, если $N = 4$, то единственным вариантом подобрать нужные A и B будет $A = 2, B = 2$. Но $2 + 2 = 4$, следовательно, при $N = 4$ наступает ничья.

В-третьих, если $N > 4$ и не является простым, то для любых подходящих A и B $4 < A + B < N$. Докажем это.

Заметим, что $B = N/A$. Условие

$$A + \frac{N}{A} > 4$$

равносильно

$$A^2 - 4A + N > 0$$

Дискриминант этого квадратного относительно A уравнения отрицателен при $N > 4$, следовательно, эти условия всегда выполняются.

Для доказательства условия $A + B < N$ упомянем, что $A \geq 2$, а также положим $A \leq B$, то есть, $A^2 \leq N$. Условие

$$A + \frac{N}{A} < N$$

равносильно

$$A^2 < N \cdot (A - 1)$$

что верно при $A > 2$ для любых N, а при $A = 2$ верно для любых $N > 4$.

Следовательно, при $N > 4$ игра всегда будет заканчиваться, так как на каждом ходу N строго уменьшается и все получающиеся числа строго превосходят 4.

Для $N \leq 4$ задача уже решена. Решим ее для остальных N.

Будем хранить в массиве `winner` для каждого i номер поросенка, который выигрывает, если игра начинается с i. Определим для некоторого k значение в массиве `winner`. Если k простое, то `winner[k]` = 1. Иначе, будем перебирать все делители j числа k такие, что $j^2 <= k$. Заметим, что мы выполняем этот перебор за *второго* поросенка.

Предположим, второй поросенок выбрал некоторое j из делителей k. Заметим, что после совершения этого хода мы свели задачу для числа k к задаче для числа $j + k/j$, в которой номера поросят оказались сдвинутыми, то есть, поросенок с номером 2 стал поросенком с номером 1, третий поросенок получил номер 2, а первый поросенок сделался третьим.

Таким образом, если `winner[j + k / j]` = 1, то в задаче для числа k второй поросенок выигрывает, если выбирает число j. Так как любой поросенок играет так, как ему выгоднее, то если среди делителей числа k найдется j такой что `winner[j + k / j]` = 1, то `winner[k]` = 2. Если же такого j не найдется, то при любом ходе второго поросенка он не выиграет. Следовательно, он максимизирует своим ходом величину $j + k/j$. Несложно показать, что это достигается при минимальном j.

Вышеописанная логика решения может быть оформлена в виде следующей процедуры (она включает в себя и случай простого k):

```
procedure defineWinner(k: integer);
var
    j, minDiv: integer;
begin
    if k = 4 then begin
        {Особыйслучай ничья - .}
        exit;
```

```
    end;
    j := 2;
    minDiv := 0;
    while j * j <= k do begin
        if k mod j = 0 then begin
            if minDiv = 0 then begin
                minDiv := j;
            end;
            if winner[j + k div j] = 1 then begin
                {Второйпоросенок находитвыигрышныйход .}
                winner[k] := 2;
                break;
            end;
        end;
        inc(j);
    end;
    if winner[k] = 0 then begin
        if minDiv = 0 then begin
            {Число kпростое - .}
            winner[k] := 1;
        end else begin
            {Выбираемминимальный делитель .}
            winner[k] := winner[minDiv + k div minDiv] + 1;
            if winner[k] > 3 then begin
                winner[k] := 1;
            end;
        end;
    end;
end;
```

Для такой процедуры необходимо, чтобы необходимые значения массива `winner` были известны до момента вызова процедуры. Наиболее простым решением является вызов этой процедуры от всех необходимых аргументов в порядке их возрастания, например:

```
for i := 5 to N do begin
    defineWinner(i);
end;
```

Данная логика работы программы представляет собой пример динамического программирования «сверху вниз».

На практике, описанный подход для решения данной задачи работает чересчур долго для больших N. Действительно, мы перебираем все числа от 5 до N, и для каждого числа k перебираем все числа от 2 до $\sqrt{k}$ в качестве потенциальных делителей. Такая программа работает за время $O(N\sqrt{N})$, что для $N \leq 10^6$ является недопустимым.

Если применить несколько другой подход, по сути являющимся динамическим программированием «снизу вверх», то становится возможным неявно перебирать для каждого числа лишь его делители. Используя формулу Дирихле для числа делителей ([3]), можно показать, что такое решение будет работать за $O(NlogN)$. Реализации такого решения можно довести до приемлемого времени работы, используя различные нетривиальные приемы программирования, и здесь мы их приводить не будем.

Однако для данной задачи первый подход можно значительно ускорить, если вычислять победителя не для каждого возможного значения k, а только для необходимых. Для этого можно использовать приведенную выше процедуру с незначительными изменениями:

```
procedure defineWinner(k: integer);
```

```
var
    j, minDiv: integer;
begin
    if k = 4 then begin
        {Особыйслучай ничья - .}
        exit;
    end;
    j := 2;
    minDiv := 0;
    while j * j <= k do begin
        if k mod j = 0 then begin
            if minDiv = 0 then begin
                minDiv := j;
            end;
            {Вызовемпроцедуручтобы , определитьнеобходимоезначение .}
            defineWinner(j + k div j);
            if winner[j + k div j] = 1 then begin
                {Второйпоросенок находитвыигрышныйход .}
                winner[k] := 2;
                break;
            end;
        end;
        inc(j);
    end;
    if winner[k] = 0 then begin
        if minDiv = 0 then begin
            {Число kпростое - .}
            winner[k] := 1;
        end else begin
            {Выбираемминимальный делитель .}
            {Здесьвызывать процедурудополнительнонетнеобходимости .}
            winner[k] := winner[minDiv + k div minDiv] + 1;
            if winner[k] > 3 then begin
                winner[k] := 1;
            end;
        end;
    end;
end;
```

Такую процедуру достаточно вызвать один раз, чтобы получить ответ:

```
defineWinner(N);
```

Решение, основанное на этой процедуре, уже достаточно быстрое. Однако, можно заметить, что в течение работы программы существует возможность вызвать эту процедуру с одинаковым аргументом более одного раза. При этом процедура считает значение заново, что не требуется. Поэтому можно выходить из процедуры досрочно, если значение уже найдено.

```
procedure defineWinner(k: integer);
var
    j, minDiv: integer;
begin
    if (k = 4) or (winner[k] <> 0) then begin
        {Ничьяили ужеопределенноезначение .}
        exit;
```

```
end;
j := 2;
minDiv := 0;
...
```

Данный подход к реализации решения часто называют «ленивым» динамическим программированием, так как значения в состояниях, которые не нужны для вычисления ответа для задачи, никогда не вычисляются.

Задача N. Ладья в лабиринте

Данная задача может быть сведена к нахождению длины кратчайшего пути в невзвешенном неориентированном графе. Такой тип задач решается с помощью *поиска в ширину*. Об этом алгоритме и его применениях можно прочитать в [1].

Однако создание и хранение этого графа в явном виде для данной задачи неприменимо, так как при данных ограничениях в нем будет до 250000 вершин и до $2 \cdot 500 \cdot C_{500}^2$ ребер (на пустой доске). В излагаемом ниже решении мы будем использовать этот граф неявным образом.

Будем заполнять пустые клетки доски целыми неотрицательными числами. Если в некоторой клетке стоит число k, это будет обозначать, что до этой клетки ладья может дойти минимум за k ходов. В процессе работы алгоритма эти величины будут определяться, поэтому заполним изначально все пустые клетки доски некоторым очень большим числом (назовем его `INF`), превосходящим возможное число ходов. Это число будет выполнять роль своеобразной «бесконечности».

Число ходов заведомо не превышает площадь доски — иначе бы ладья побывала на одной клетке дважды, и соответствующий маршрут был бы неоптимален. Следовательно, сделаем `INF` $>= n \cdot m$, например `INF` $= 500000$.

Занятые пешками клетки доски пометим каким-нибудь другим числом, например, -1.

Организуем структуру данных «очередь», в которой будем хранить пары чисел — координаты полей на доске. Запишем в клетку, на которой изначально стоит ладья, число 0, а координаты этой клетки положим в очередь.

Алгоритм решения задачи заключается в следующем. Если очередь пуста, то алгоритм прекращает работу. Иначе, вынем координаты очередной клетки из очереди. Пусть значение, записанное в этой клетке, равно k. Будем двигаться в направлении возможного движения ладьи (вверх, вниз, влево и вправо), пока значения, записанные в посещаемых клетках, равны `INF`. По мере движения заполняем посещенные клетки значением $k + 1$ и кладем их координаты в очередь.

Рассмотрим числа, записанные в клетках, координаты которых в данный момент хранятся в очереди. Заметим, что все они отличаются друг от друга не более чем на единицу и не убывают в соответствии с порядком вынимания клеток из очереди. Это утверждение легко доказывается по индукции.

Заметим, что при таком движении мы можем остановиться в трех случаях:

- Мы вышли за край доски.
- Мы перешли на занятую клетку.
- Мы перешли на ранее посещенную клетку, в которой записано число k или $k + 1$.

Действительно, если мы переходим на клетку, в которой записано большее, чем $k+1$, число (но не бесконечность), то мы вытащили из очереди клетку, в которой записано число, большее чем k, раньше чем текущую клетку, что противоречит свойству, отмеченному ранее. Если же это число меньше k, то клетка, из которой мы движемся на текущем ходу, была бы уже ранее заполнена.

Заметим, что с помощью данного алгоритма пустые клетки, достижимые из начальной, заполняются корректно, то есть, число указывает минимальное расстояние в ходах ладьи до начальной точки. Недостижимые же из начальной клетки остаются заполненными значением `INF`.

С учетом сказанного, ответ равен числу, хранящемуся в конечной клетке, если оно не равно `INF`, и -1 в противном случае.

Время работы можно оценить как $O(m \cdot n)$, т.е. пропорционально площади доски. Действительно, каждая свободная клетка попадает в очередь не более одного раза, и любые другие действия с клетками также вовлекают каждую из них не более чем константное число раз.

С другой стороны, число ходов ладьи равно числу поворотов в пути плюс 1. Поэтому данная задача эквивалентна следующей: найти путь из начальной вершины в конечную с минимальным числом поворотов.

Если рассмотреть граф, в котором вершинами служат пары «поле доски, направление движения», а ребра бывают двух типов: ребра веса 0, ведущие в соседнюю вершину по направлению движения, и ребра веса 1, изменяющие направление движения, то задача сводится к поиску кратчайшего пути в графе с ребрами, имеющими веса 0 и 1. Этот путь можно найти с помощью алгоритма, напоминающего поиск в ширину. Отличие заключается в том, что вместо очереди в этом алгоритме используется дек, и при переходе по ребру с нулевым весом мы кладем новую вершину в начало дека, а при переходе по ребру с единичным весом — в конец. Доказательство корректности этого алгоритма мы оставим в качестве небольшого домашнего задания, отметив, что при его анализе полезно вспомнить *алгоритм Дейкстры*, о котором можно прочитать в [1].

Список литературы

[1] Кормен Т., Лейзерсон Ч., Ривест Р. Алгоритмы: построение и анализ. - М.: МЦНМО, 1999. - 960 с., 263 ил.

[2] Майника Э. Алгоритмы оптимизации на сетях и графах. - М.: Мир, 1981. - 323 с., ил.

[3] Формула Дирихле в Википедии: http://ru.wikipedia.org/wiki/Формула_Дирихле

[4] Divisor Function in Wikipedia: http://en.wikipedia.org/wiki/Divisor_function