Разбор задачи «Ожерелье»

Правильное решение

Задача всегда имеет решение. Достаточно с нужной стороны подогнать колечко N к колечку N-1, а дальше действовать рекурсивно, считая, что колечки N и N-1 склеены. Такое решение идейно простое, но не самое легкое для реализации, так как при протаскивании «склеенного колечка» необходимо делать сразу несколько перемещений.

Другое решение отдаленно напоминает метод сортировки пузырьком. Главное его преимущество заключается в том, что это решение очень легко придумать и запрограммировать. Кажется, что это решение не является верным, а представляет собой всего лишь некоторую эвристику, но доказать это не просто.

Решение заключается в том, что мы находим пару (x, y) соседних правильно упорядоченных колечек $(x < y)$ таких, что их можно протащить друг через друга $(|y-x| > 1)$, т.е. $y-x > 1$. Перетаскиваем колечки, ищем новую пару и так далее пока не получим полное упорядочивание колечек. Данное решение очень легко запрограммировать, но совершенно не понятно, почему оно всегда выдает правильный ответ.

Чтобы это понять, рассмотрим частный случай алгоритма: берем колечко с номером 1 и перемещаем его по часовой стрелке до тех пор, пока это возможно, т.е. пока не встретим колечко с номером 2. Затем аналогично перемещаем колечко с номером два, и так далее. В какой-то момент мы дойдем до последнего колечка. После этого повторяем проделанные операции заново и вновь пытаемся переместить колечко с номером 1. Интуитивно понятно, что в результате таких операций происходит постепенное упорядочивание колечек. Когда все колечки будут упорядочены по часовой стрелке, алгоритм завершит свое выполнение, так как ни одно из них невозможно будет переместить.

Докажем корректность работы алгоритма. Рассмотрим функцию $Dist(x, y)$, которая возвращает расстояние от колечка x до колечка y, если двигаться по часовой стрелке. Дополнительно рассмотрим сумму $S = 1 \cdot Dist(1, 2) + 2 \cdot Dist(2, 3) + \ldots + n \cdot Dist(n, 1)$. Очевидно, что при смене колечек x и y местами в сумме S изменяются только слагаемые $Dist(x, x+1)$ и $Dist(y, y+1)$. Первое слагаемое при такой транспозиции увеличивается на 1, а второе – уменьшается на 1, значит, а вся сумма изменяется на величину $\Delta S = x-y$. Учитывая, что $x < y$, получаем, что в результате перестановки колечек x и y сумма S уменьшается, а так как S все время остается положительным, то наш алгоритм рано или поздно закончит работу. Для завершения необходимо заметить, что в любой позиции, кроме правильного упорядочивания, существует «правильная» пара колечек, которые алгоритм должен поменять местами. Значит, после завершения работы алгоритма получится расстановка 1, 2, …, N.

Можно доказать, что оба алгоритма выдают минимально возможное количество перестановок и в случае расположения колечек в обратном порядке общее количество транспозиций будет $(N-2)(N-1)N/6$. Получается, что верное решение задачи имеет кубическое время выполнения и линейный расход по памяти, а максимальное количество перестановок получается на тесте 50, 49, …, 1 и равно 19600.

Эвристические решения и критерий их оценивания

Правильное решение набирает 100 баллов.

При небольших значениях N ($N \leq 11$) верный ответ можно получить с помощью такого алгоритма: создаем граф, вершинами которого являются всевозможные конфигурации ожерелья. Затем при помощи поиска в ширину находим нужную последовательность транспозиций. Такое решение набирало 40-50 баллов.

Помимо этого оценивались решения, которые разбирали случаи маленьких значений N, прямой и обратный порядок расположения колечек и оценивалось примерно в 30 баллов. Решения, которые выдавали только 0 и –1 или проходили только тривиальные тесты на прямой порядок оценивались в 0 баллов.

Учитывая, что информация о номерах колечек хранится в массиве, можно было допустить ряд ошибок, связанных с тем, что массив – линейный объект, а ожерелье замкнуто и представляет циклическую структуру. Естественно такие решения проходили не все тесты и за счет ошибок реализации набирали меньшее количество баллов.

Система тестов

Номер теста	На что тест (описание)	Размерность теста	Полный порядо	Полный порядо	Обратный порядо	Обратный порядо	Вид теста
1	Тест “1 2”	N = 2	+	+	+	+	Маленькие тесты
2	Тест “2 1”	N = 2	+	+	+	+	
3	Тест “1 2 3”	N = 3	+	+			
4	Тест “2 3 1”	N = 3		+			
5	Тест “3 1 2”	N = 3		+			
6	Тест “3 2 1”	N = 3			+	+	
7	Тест “2 1 3”	N = 3				+	
8	Тест “1 3 2”	N = 3				+	
9	Тест “1 2 3 4 5 6 7 8 9”	N = 9	+	+			Средние тесты
10	Тест “7 8 1 2 3 4 5 6”	N = 8		+			
11	Тест “9 8 7 6 5 4 3 2 1”	N = 9			+	+	
12	Тест “1 8 7 6 5 4 3 2”	N = 8				+	
13	Тест “1 2 4 3”	N = 4					
14	Случайный тест	N = 4					
15	Случайный тест	N = 5					
16	Случайный тест	N = 6					
17	Случайный тест	N = 7					
18	Случайный тест	N = 8					
19	Случайный тест	N = 9					
20	Случайный тест	N = 10					
21	Тест “1 2 ... 50”	N = 50	+	+			Большие тесты
22	Тест “10 11 ... 49 1 2 ... 9”	N = 49		+			
23	Тест “50 49 ... 1”	N = 50			+	+	
24	Тест “20 19 ... 1 49 48 ... 21”	N = 49				+	
25	Тест “2 1 4 3 6 5 8 7 ... 50 49”	N = 50					
26	Тест “3 2 1 6 5 4 9 8 7 ... 48 47 46	N = 48					
27	Случайный тест	N = 11					
28	Случайный тест	N = 15					
29	Случайный тест	N = 20					
30	Случайный тест	N = 25					
31	Случайный тест	N = 30					
32	Случайный тест	N = 35					
33	Случайный тест	N = 40					
34	Случайный тест	N = 45					
35	Случайный тест	N = 50					
36	Транспозиция “7 8”	N = 12					
37	Транспозиция “10 11”	N = 17					
38	Транспозиция “1 2”	N = 24					
39	Транспозиция “30 31”	N = 31					
40	Транспозиция со смещением	N = 33					
41	Транспозиция со смещением	N = 34					
42	Транспозиция со смещением	N = 36					
43	Транспозиция со смещением	N = 37					
44	Транспозиция со смещением	N = 38					
45	Тест “1 2 ... 25 50 49 ... 26”	N = 50					
46	Тест “49 50 47 48 ….”, shl 5	N = 50					
47	Случайный «гладкий» тест	N = 43					
48	Тест “50 1 49 2 …”, shl 17	N = 50					
49	Тест “49 1 48 2 …”, shl 32	N = 49					
50	Тест “50 2 3 4 5 …. 48 49 1”	N = 50					

Пример правильного решения задачи:

```
Const
  MAX = 50;
Var
  n: Integer;
  rings: Array[0..MAX-1] Of Integer;

Procedure ReadInput;
Var
  i: Integer;
Begin
  Assign(input, 'Necklace.in');
  Reset(input);
  Read(n);
  For i := 0 To n-1 do
    Read(rings[i]);

  Close(input);
End;

Function Swap(x, y: Integer): Boolean;
Var
  rx, ry: Integer;
Begin
  x := x Mod n;
  y := y Mod n;
  rx := rings[x];
  ry := rings[y];

  If (ry-rx > 1) Then
    Begin
      rings[x] := ry;
      rings[y] := rx;
      WriteLn(rx, ' ', ry);
      Swap := True;
    End
  Else
    Swap := False;
End;

Procedure Solve;
Var
  terminate: Boolean;
  i: Integer;
Begin
  Assign(output, 'Necklace.out');
  Rewrite(output);
  terminate := false;
  While (Not(terminate)) Do
    Begin
      terminate := True;
      For i := 1 To n do
        If (Swap(i, i+1)) Then
          terminate := False;
    End;
  WriteLn(0);
  Close(output);
End;

Begin
  ReadInput;
  Solve;
End.
```

Разбор задачи «Менеджер памяти»

Разобьем всю память на отрезки, каждый из которых состоит либо из свободных, либо из занятых (выделенных в ответ на запросы) ячеек.

Будем некоторым образом хранить свободные и занятые отрезки. Важное ограничение, упомянутое в условии, состоит в том, что выделенный кусок памяти может находиться либо в начале памяти, либо непосредственно после занятого фрагмента памяти. В связи с этим потребуем, чтобы свободные отрезки не могли следовать друг за другом.

Нам необходимо уметь:

1) по заданному объёму памяти находить свободный отрезок, из которого данный объём можно выделить;
2) по номеру запроса находить занятый отрезок, выделенный этим отрезком;
3) по занятому отрезку находить свободные отрезки, находящиеся непосредственно перед и после него.

Поскольку по условию оптимальность не требуется, мы можем всегда выделять память из самого длинного имеющегося свободного отрезка.

Будем использовать следующие структуры данных:

1) двусвязный список свободных отрезков;
2) массив запросов, где индексом является номер запроса, а значением — выделенный при выполнении этого запроса отрезок (или некоторое специальное значение вроде «nil» или «NULL», если память выделена не была или запрос не был запросом на выделение);
3) кучу (heap) из указателей на свободные отрезки; куча строится по их длинам.

(Куча и двусвязный список должны содержать одни и те же свободные отрезки; этого можно добиться, используя указатели или классы Delphi и Visual Basic.)

Куча — это массив из N чисел $H_1, \ldots, H_N$, таких, что для любого i, $1 < i \le N$, выполняется условие $H_{[i/2]} \ge H_i$. Первый элемент в куче всегда не меньше остальных ее элементов, поэтому операция нахождения максимума в куче выполняется за время $O(1)$. Можно реализовать операции добавления элементов в кучу, удаления из нее и изменения элемента за время $O(\log N)$; как это сделать, описано, например, в книге Т. Кормена, Ч. Лейзерсона, Р. Ривеста «Алгоритмы: построение и анализ», М.. МЦНМО, 2000.

Обработка запроса на выделение памяти производится следующим образом. Рассмотрим самый длинный свободный отрезок A: с помощью кучи мы можем найти его в списке за $O(1)$. Если длины отрезка A недостаточно, чтобы вместить требуемый блок памяти, то такой блок невозможно разместить. Иначе мы размещаем данный блок, вставляя соответствующий занятый отрезок в список и укорачивая отрезок A.

Обработка запроса на освобождение памяти производится следующим образом. Мы удаляем соответствующий занятый отрезок A из списка, после чего разбираем 4 случая:

1. Непосредственно слева и справа от A не располагается свободного отрезка. В этом случае мы делаем A свободным отрезком и добавляем его в кучу.
2. Непосредственно слева от A не располагается свободного отрезка, непосредственно справа от A располагается свободный отрезок B. В этом случае мы удаляем отрезок A из списка и удлиняем отрезок B.

3. Непосредственно справа от A не располагается свободного отрезка, непосредственно слева от A располагается свободный отрезок. Действуем аналогично случаю 2.
4. Непосредственно слева от A располагается свободный отрезок B, непосредственно справа от A располагается свободный отрезок C. В этом случае мы удаляем отрезки A и C из списка и удлиняем отрезок B.

Разбор задачи “Казино”

Заметим, во-первых, что не любая стратегия игрока приводит к нужному результату. Например, если в примере, разбиравшемся в условии задачи (**rrrgggbbb** и т.д.), стоимость буквы **b** будет больше стоимости буквы **r**, то только один из путей приводит к оптимальному результату: три раза забрать буквы **gb**. После некоторого размышления можно понять, что всевозможные т.н. «жадные» решения — делать самый выгодный ход, делать самый длинный ход, делать самый левый ход, и т.п. — также в большинстве случаев неверны. Перебирать же все варианты, для каждого из них — опять все варианты, и так далее — оказывается слишком медленным подходом.

Для решения этой задачи применим динамическое программирование. В первой половине решения мы для каждого куска начальной последовательности фишек определим, можем ли мы забрать его целиком, не затрагивая при этом остальные фишки. Иными словами, для всех l и r найдем величину $a[l,r]$, равную 1, если фишки с номерами (то есть позициями в начальном расположении, считая слева) с l по r игрок может за несколько действий все забрать себе, и 0 — в противном случае. Затем по этим данным во второй половине решения мы восстановим ответ к задаче.

Во избежание путаницы будем называть последовательности, объявленные крупье (в отличие от начальной последовательности) *словами*.

Вторая половина решения проще первой, поэтому начнём с неё. Пусть нам известны величины $a[l,r]$. Пусть тогда $b[l,r]$ — это максимальная сумма денег, которую можно получить, играя только на отрезке с l-ой фишки по r-ую (то есть, не трогая остальных фишек). Ясно, что если $a[l,r]=1$, то $b[l,r]$ — это просто сумма стоимостей всех фишек с l-ой по r-ую. Если же $a[l,r]=0$, то уже не удастся забрать все фишки, соответственно, какая-то из фишек должна остаться — пусть это фишка с номером k. Тогда заметим, что задача распадается на две — для фишек слева от k-ой и для фишек справа от неё, т.е. что $b[l,r]=b[l,k-1]+b[k+1,r]$ (здесь и далее мы считаем, что для любого m $b[\mathrm{m},\mathrm{m}-1]=0$). Но так как номер оставшейся фишки не фиксирован, то в действительности $b[l,r]$ равно максимуму этих величин по всем k:

$$b[l,r]=\max_{k=l,l+1,\ldots,r}\left(b[l,k-1]+b[k+1,r]\right)\quad(*)$$

(напомним, что эта формула верна при $a[l,r]=0$). Теперь мы видим, что для нахождения величин $b[l,r]$ можно применить динамическое программирование: если мы будем перебирать r от 1 до длины начальной последовательности, а затем l от r до 1, то к моменту вычисления величины $b[l,r]$ все величины, которые необходимы для её нахождения по формуле (*), уже вычислены. Таким образом, зная величины $a[l,r]$ мы можем найти $b[l,r]$. Легко видеть, что на этом шаге требуется порядка L^3 действий (напомним, что L — это длина начальной последовательности).

Теперь вернёмся к первой половине решения, а именно — нахождению величин $a[l,r]$. Как узнать, можно ли забрать себе данную последовательность (в нашем случае — кусок начальной) целиком? Во-первых, возможно, эту последовательность можно разделить на две части, каждую из которых можно забрать себе целиком. То есть, если найдётся k такое, что $a[l,k]=1$ и $a[k+1,r]=1$, то и $a[l,r]=1$. Если же так сделать не удаётся, то можно заметить, что если мы всё-таки заберём всю последовательность целиком, то её первую и последнюю фишки мы заберём только на последнем шаге. Пусть на последнем шаге мы заберём некоторое слово S. Тогда задача сводится к тому, чтобы выкинуть из последовательности некоторые куски (можно считать, что для этих кусков величина a уже посчитана, поэтому мы знаем, какие из них можно выкинуть, а какие — нет), оставив первую и последнюю фишки на месте, так, чтобы осталось слово S.

Для решения этой задачи опять применим динамическое программирование. А именно, пусть $c[l,r,p,q]=1$, если можно из отрезка с l-ой фишки по r-ю можно так выкинуть несколько кусков, чтобы остались первые q фишек слова номер p, и при этом l-я и r-я

фишки остались на месте, и 0 — иначе (тогда нас интересует величина $c[l,r,p,l\text{ength}(p)]$, где $l$ength(p) — длина слова номер p). Во-первых, ясно, что если первая фишка слова номер p не совпадает с l-ой фишкой начальной последовательности, или q-я — с r-ой, то $c[l,r,p,q]=0$. В противном случае, пусть (q-1)-я фишка слова получится из k-ой фишки последовательности. Тогда, во-первых, должно быть $c[l,k,p,q-1]=1$ (чтобы получить первые q-1 фишек), а, во-вторых, $a[k+1,r-1]=1$ (чтобы выкинуть кусок между (q-1)-ой фишкой и q-ой). Но так как число k не фиксировано, то получаем, что $c[l,r,p,q]=1$ тогда и только тогда, когда найдётся такое k. Тем самым получен алгоритм для подсчета c. Оценим время его работы. На подсчёт одного элемента массива c требуется порядка L действий (по количеству возможных k). Нам необходимо подсчитать элементы вида $c[l,r,p,l\text{ength}(p)]$. Из вышеприведённых рассуждений легко заметить, что для их подсчёта нам потребуется подсчёт только элементов вида $c[l,\dots,\dots,\dots]$, а элементов такого вида порядка $L\cdot S$ (где S — сумма длин всех слов), так как для параметра r есть порядка L вариантов, а для пары параметров (p,q) — порядка S вариантов. Тем самым мы для любых l и r можем найти $a[l,r]$ за порядка $L\cdot L\cdot S= L^2\cdot S$ действий, а общее время работы получается порядка $L^2\cdot L^2\cdot S=L^4\cdot S$.

Однако легко заметить, что если подсчитать все величины $c[l,r,p,q]$ сначала, а потом лишь использовать найденные значения, то время работы будет меньше. Действительно, всего в массиве c $L^2\cdot S$ элементов, и на подсчёт каждого из них уйдёт порядка L действий, тем самым общее время работы будет порядка $L^3\cdot S$.

Разбор задачи «Красивые числа»

Рассмотрим искомое равенство:

$$n = a_1 + a_2 + ... + a_m,$$

где каждый a_i состоит только из цифр 0 и k.

Разделим обе части на k. Понятно, что при этом в числах a_i все цифры k заменятся на единицы, а нули останутся нулями.

Получается новая формулировка задачи: представить число n / k как сумму чисел, состоящих только из цифр 0 и 1. Утверждается, что ответ на нее таков: минимальное количество слагаемых равно максимальной цифре в десятичной записи числа n / k.

Докажем сначала, что меньшим числом слагаемых обойтись нельзя. Будем действовать от противного: предположим, что можно обойтись меньшим числом слагаемых. При этом, так как максимальная цифра в n / k не превышает 9, то мы предполагаем, что можно обойтись не более чем восемью слагаемыми. Выпишем эти слагаемые друг под другом (даже если они бесконечные) и сложим «в столбик». При этом переносов не будет (все цифры 0 или 1, и цифр в каждом столбце не больше 8). Рассмотрим тот столбец, в котором при суммировании должна получиться максимальная цифра из десятичной записи n / k. Даже если во всех слагаемых в этом столбце стоят единицы, все равно нужная сумма не набирается. Получили противоречие; утверждение доказано.

Теперь, пожалуй, понятно, как добиться описанного выше ответа: надо просто в каждом столбце расставить столько единиц, сколько нужно получить в сумме – это соответствующая цифра из n / k.

Пример. Пусть $n = 15$, $k = 7$.

```
15/7 = 2,(142857)
```

Максимальная цифра — 8. Подберем соответствующие восемь слагаемых.

```
1,(111111)
1,(011111)
0,(010111)
0,(010111)
0,(000111) +
0,(000101)
0,(000101)
0,(000100)
----------
2,(142857)
```

Теперь умножим эти числа на 7, и получим следующее верное равенство:

```
15 = 7,(7) + 7,(077777) + 0,(070777) + 0,(070777) + 0,(000777) + 0,
(000707) + 0,(000707) + 0,(000700)
```

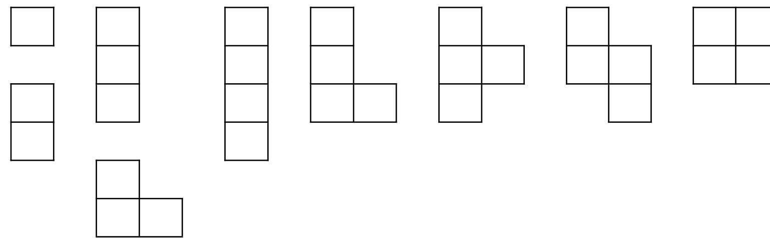
Таким образом, правильное решение состоит из следующих этапов: представление числа n / k в удобном для обработки формате (с аккуратным учетом периода), составление искомых слагаемых и их вывод с учетом имеющихся требований.

Разбор задачи «Сетевая игра».

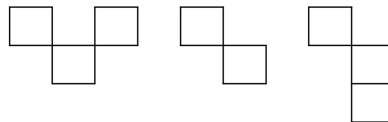
В качестве вводного замечания скажем, что игра всегда завершается победой либо первого, либо второго игрока. По правилам игры ничья никогда получиться не может.

Уровень сложности 1 (40 баллов)

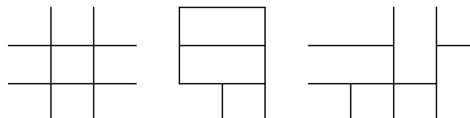
В данной задаче было выделено несколько уровней сложности. Сначала рассмотрим задачу в самой простой формулировке, в которой количество веревочек ограничено 13. Из-за такого ограничения на количество верёвочек оказывается, что возможно небольшое число конфигураций. Самое большое число ячеек сети, которые могут быть образованы 13 верёвочками, равно 4. Эти ячейки могут образовывать фигуры, хорошо известные по игре «Тетрис», и их подфигуры. Все они приведены ниже.



Эти фигуры являются одной компонентной связности каждая. Но поскольку, например, одиночный квадратик использует всего 4 верёвочки, мы можем комбинировать несколько одиночных квадратиков и фигур из двух клеток, связывая их верёвочками, которые не образуют новых ячеек сети. Таким образом, мы получим ещё несколько конфигураций игры, каждая из которых приведена ниже.



Все конфигурации приведены здесь без учёта возможных поворотов и отражений. Естественно, если в какой-либо конфигурации у нас остались верёвочки, мы можем их потратить, «подвешивая» их к верёвочкам, образующим ячейки сети. В частности, мы можем получить следующие конфигурации.



Третья конфигурация на рис. выше использует все 13 верёвочек, которые не образуют ни одной ячейки сети.

Построив все возможные конфигурации верёвочек можно вручную посчитать исход игры для каждой из возможных конфигураций. Получается, что если рассматривать фигурки тетриса, то единственной проигрышной конфигурацией является квадрат 2x2. Из «составных» конфигураций проигрышной является конфигурация из 2 одиночных клеток и без клеток вообще.
Теперь, вручную просчитав исход игры для всех возможных конфигураций, мы можем написать программу, которая будет отождествлять конфигурацию верёвочек во входном файле с одной из конфигураций, описанных выше. Жюри предполагало, что правильно написанное подобное решение должно набирать 40 баллов.

Уровень сложности 2 (70 баллов)

Очевидно, что с ростом количества ниточек, ручной разбор возможных конфигураций становиться практически невозможным, и нам требуются способы алгоритмического вычисления исходов игры.
Попробуем находить решение перебором по верёвочкам. Любая конфигурация, появляющаяся в процессе игры, отличается от начальной конфигурации только отсутствием некоторых верёвочек, разрезанных в процессе игры. Мы можем занумеровать все верёвочки числами от 0 до N-1. Тогда, поскольку нам известно начальное расположение верёвочек, любое состояние верёвочек, возникающее в процессе игры, описывается набором из N нулей и единиц, то есть N-битным числом. Начальное состояние описывается числом из всех 1.
Мы можем организовать перебор с возвратом от начальной конфигурации. На каждом уровне перебора мы пробуем отрезать каждую из оставшихся верёвочек, и если для некоторой верёвочки такой ход является допустимым, запускается рекурсивный поиск решений для новой конфигурации.
На каждом шаге перебора нам необходимо знать, какой из игроков ходит. На ходе первого игрока он выигрывает, если существует хотя бы один ход, который переводит игру в конфигурацию, проигрышную для второго игрока, первый игрок проигрывает, если при любом его ходе получается конфигурация, выигрышная для второго игрока. Аналогичные правила действуют и для хода второго игрока. Сформулированные правила выигрыша первого игрока позволяют нам сформулировать эвристику сокращения перебора, известную как αβ-отсечение, а именно, как только на ходе первого игрока мы находим ход, на котором он выигрывает, нам совсем не требуется перебирать оставшиеся ходы, вместо этого мы можем завершить работу функции немедленно, сформировав возвращаемое значение, соответствующее выигрышу первого игрока. Точно также мы сокращаем перебор и для второго игрока.
Такая эвристика очень просто программируется (буквально добавлением одного оператора if и break в тело цикла), но для данной игры даёт очень хороший результат, позволяя сократить количество просматриваемых конфигураций примерно в 2 раза.
Мы можем немного ускорить работу программы, если перед началом перебора удалим все верёвочки, которые не являются границей никакой ячейки сети. Для этого проще всего построить ячейки сети на координатной плоскости (благо, что ограничения на координаты это позволяют), а затем посмотреть, является ли каждая верёвочка границей какой либо ячейки. В цикле по всем верёвочкам для каждой вертикальной верёвочки мы прибавляем 1 к клеткам, расположенной слева и справа от неё, а для каждой горизонтальной верёвочки мы прибавляем 1 клеткам, расположенным сверху и снизу от верёвочки. Те клетки, в которых после цикла записано число 4, являются целыми ячейками сети.
Особой аккуратности требует работа с верёвочками, координаты которых находятся на границе диапазона допустимых координат. Например, если мы рассмотрим вертикальную верёвочку с координатой x, равной 0, то для неё не существует клетки слева. Чтобы избежать излишних проверок координат, можно создать массив размера 52x52 вместо 50x50. Для программ на Паскале этого можно добиться, объявив массив как `array [-`

`1..51] of integer`. Такой приём недоступен для программирующих на Си, однако в можно добиться того же эффекта, объявив массив как `int m[52][52]`, но прибавляя 1 к каждой координате при чтении конфигурации верёвочек из входного файла. Кстати, это даже будет более эффективно и для программ на Паскале, чем первый подход.
Тем не менее, вернёмся к организации перебора. Если на каждом шаге рекурсии после разрезания верёвочки сразу же удалять верёвочки, которые мы больше не можем использовать, так как они больше не являются границей целой ячейки сети, мы также получим некоторое ускорение.
Но самое главное улучшение следующее. Если мы рассмотрим ход работы алгоритма перебора, мы заметим, что перебор много раз попадает в одну и ту же конфигурацию. Возникает идея сохранить результат вычисления результата игры в данном состоянии для того чтобы, когда мы снова вернёмся в это же состояние, мы сможем взять уже вычисленное значение, а не выполнять весь рекурсивный обход по новой. Для каждого состояния верёвочек достаточно сохранять исход игры для первого игрока: 1 – если первый игрок выигрывает, 2 – если он проигрывает. Добавляя сюда 0 для ситуации, когда исход игры в данной конфигурации ещё неизвестен, мы получаем, что для каждого состояния игры достаточно хранить 1 байт информации (реально в нём будут использоваться только 2 бита). Если в некоторую конфигурацию мы попадаем на ходе второго игрока, исход игры для него будет противоположным чем тот, который сохранён в таблице, то есть 0 для 0 в таблице, 1 для 2 в таблице, 2 для 1 в таблице. Как уже было замечено выше, общее количество возможных состояний, которые могут возникать в процессе игры, равно 2^N, и, следовательно, для хранения всех состояний нам потребуется массив этого размера. Поскольку ограничение памяти для данной задачи установлено в 64 мб, под массив мы можем занять не более чем 32 мб памяти, что соответствует значению N равному 25.
Итак, если нам дано количество верёвочек, не превышающее 25, то мы можем реализовать рекурсивный поиск решения с отсечениями и запоминанием промежуточных результатов в таблице. Мы можем пытаться поднять это ограничение на N за счёт хранения информации от нескольких состояниях в одном байте, так мы получим максимальное N равное 27. С другой стороны, массив информации о состояниях игры достаточно разрежен, то есть большое количество конфигураций игры никогда не посещаются. Это позволяет нам использовать различные техники для хранения разреженных массивов, например, хэш-таблицы. В результате мы можем значительно поднять ограничение на максимальное количество верёвочек, но тогда мы столкнёмся с проблемой нехватки времени из-за того, что работа с разреженными массивами требует значительно больше времени, чем работа с обычными массивами.

Уровень сложности 3 (100 баллов)

Все соображения, изложенные выше, подталкивают нас к одному простому наблюдению. Игру можно рассматривать не только в терминах верёвочек, но в терминах целых ячеек сети. Мы можем за шаг игры удалить либо две смежных ячейки сети, либо одну ячейку при условии, что у неё меньше 4 соседей. Максимальное число ячеек в фигуре, которая состоит не более из 50 верёвочек равно всего 20. Это будет фигура прямоугольника 5x4, которая использует 49 верёвочек. Состоянием такой игры будет множество ещё целых ячеек сети. Если в начале игры было K ячеек, то всего состояний у такой игры не более чем 2^K. При максимальном значении K равном 20 мы легко можем создать массив требуемого размера.
Перебор по ячейкам сети организуется следующим образом: для каждой ячейки сети мы сначала пытаемся удалить сразу две ячейки: её и соседнюю с каждой из четырёх сторон ячейку. Если у ячейки меньше четырёх соседних ячеек, мы пытаемся удалить и саму ячейку. После каждого из возможных удалений мы запускаем рекурсию для новой конфигурации ячеек.

Некоторую сложность представляет восстановление номера верёвочки, которая является выигрышной для первого игрока, поскольку в переборе по ячейкам верёвочки никогда не возникают. Один из способов решить данную проблему – запускать перебор не по ячейкам, а по верёвочкам, а на втором и последующих шагах работать только с ячейками. Как и в случае перебора по верёвочкам, αβ-отсечение позволяет существенно ускорить перебор. Именно такое решение считалось полным и проходило все тесты.

Резюме полного решения

Итак, полное решение данной задачи должно состоять из следующих этапов:

- Построение ячеек по входному множеству верёвочек.
- Перебор по ячейкам с запоминанием результатов вычисления и αβ-отсечением.
- Восстановление номера верёвочки выигрышного хода.

Динамическое программирование

В данной задаче мы можем попробовать реализовать вычисление результата игры с помощью метода динамического программирования. В этом случае мы начинаем расчёт с всех конфигураций, состоящих из 1 клетки, затем переходим к конфигурации из двух клеток и так далее, пока не дойдем до конфигурации с K клетками. Проблема динамического программирования в том, что при таком расчёте исходов игры мы попадаем в большое число состояний, которые никогда бы не были достигнуты при оптимальной игре обоих игроков. Кроме того, переход от шага i к шагу $i + 1$ при динамическом программировании в данном случае достаточно сложен. Поэтому решение, использующее принцип динамического программирования, не должно проходить несколько самых сложных тестов.

Разбор задачи “Сталкер”

Сведем задачу к задаче поиска кратчайшего пути в графе с длинами ребер, равными 0 или 1. Затем покажем, как решать эту задачу со сложностью O(E), где E — число ребер в графе.

Рассмотрим граф состояний сталкера. Состояние — это пара (x, y), где x — это номер здания, в котором находится сталкер, а y — номер загруженной карты. Сталкер может бесплатно перейти из одного здания в другое, если на загруженной карте есть такое ребро. В этом графе такому действию соответствует ребро (x_1, y) — (x_2, y). Такие ребра имеют вес 0. Также сталкер может загрузить другую карту. Этому действию соответствует ребро (x, y_1) — (x, y_2). Такие ребра имеют вес 1. Таким образом, общее количество ребер равно $M + N * K * (K — 1) / 2$, где M — общее число дорог на всех картах. Но NK^2 — это слишком много. Чтобы уменьшить количество ребер, применим следующий трюк:

Большую часть ребер составляют всевозможные ребра (x, y_1) — (x, y_2). Добавим для каждого x вершину (x, 0), и проведем для каждого y дуги $(x, y) \rightarrow (x, 0)$ с весом 0 и $(x, 0) \rightarrow (x, y)$ с весом 1. Теперь между любыми двумя вершинами (x, y_1) и (x, y_2) есть путь $(x, y_1) \rightarrow (x, 0) \rightarrow (x, y_2)$ с весом 1. В модифицированном графе количество ребер равно $E = M + 2 * N * K$.

Теперь в полученном графе нужно найти кратчайший путь от вершины (1, 0) до вершины (N, 0). Покажем, как решать эту задачу за время O(E).

Идея похожа на обычный поиск в ширину. Но мы будем добавлять вершины не только в конец очереди, но и в начало (то есть, мы используем вместо очереди дек, но для удобства будем и дальше называть его очередью). У каждой вершины будет пометка, показывающая, была ли вершина обработана. Также будем хранить текущее расстояние d_i до каждой вершины i. Вначале положим в начало очереди первую вершину, и установим расстояние до нее $d_1 = 0$, а расстояние до остальных $d_i = +\infty$, $i > 1$.

На каждой итерации вынимаем вершину i из начала очереди, если она уже была обработана, то пропускаем ее, иначе проделываем с ней следующее: перебираем все ребра $i \rightarrow j$, выходящие из этой вершины и пытаемся по ним пройти. Тогда новое расстояние до вершины j будет равно $d' = d_i + \mathrm{dist}(i, j)$, где $\mathrm{dist}(i, j)$ — вес ребра $i \rightarrow j$. Если $d' < d_j$, то обновляем расстояние и кладем вершину j в очередь. Причем, если ребро $i \rightarrow j$ веса 1, то добавляем вершину в конец очереди, а если веса 0, то в начало очереди. При завершении итерации помечаем вершину i как обработанную. Итерации продолжаем до тех пор, пока очередь не станет пустой.

Доказательство того, что алгоритм находит кратчайшие расстояния аналогично доказательству обхода в ширину. Так же, как в поиске в ширину доказывается, что вершины обрабатываются в порядке увеличения расстояния. Поэтому вершина может быть добавлена в очередь не более двух раз. Таким образом, общее время работы с очередью O(V) и общее время обработки вершин O(E).