Разбор задачи «Сортировка вагонов»

Лига А, задача В. Лига В, задача D

Автор задачи и разбора – И. Григорьев, автор формулировки условия – В. Матюхин

Идея задачи взята из книги Д. Кнута[1], где подобный железнодорожный разъезд используется для иллюстрации работы стека[2].

Напомним, что условия задачи в разных лигах отличались не только ограничениями (до 100 вагонов в лиге В и до 2000 вагонов в лиге А), но и тем, что в лиге В требовалось лишь ответить, возможно ли правильно расставить вагоны, а в лиге А необходимо было ещё и определить, какие действия для этого нужны.

Первый способ – прямое моделирование

По-видимому, один из самых очевидных и эффективных способов решения, пригодный для обеих лиг – это прямое моделирование процесса перемещения вагонов. (Технически более простое решение, годящееся лишь для лиги В, будет упомянуто в самом конце разбора). Для моделирования тупика потребуется массив (или стек). Вагоны, прибывающие с пути 1, можно хранить в другом массиве (но в некоторых решениях можно без этого обойтись). Про путь 2 достаточно лишь помнить, какой вагон должен идти туда следующим.

Удобно представить себе, что мы играем в такую игру для одного игрока: за один ход можно либо завести один вагон в тупик с пути 1, либо вывести один вагон из тупика на путь 2, не нарушая порядка на этом пути. Цель игры – собрать все вагоны на пути 2 в правильном порядке. Неважно, что по условию задачи разрешается перемещать сразу несколько вагонов – такое перемещение удобно заменить несколькими «одновагонными» перемещениями. (Удобно, разумеется, не железнодорожнику, которому пришлось бы расцеплять состав в нескольких местах, но программисту).

Заметим, что делать ход первого типа – выводить вагон из тупика – можно (и нужно) лишь в том случае, если это именно тот вагон, которого ждут следующим на пути 2. В любом другом случае у нас просто нет другого выхода, кроме как заводить в тупик очередной вагон с пути 1. Если при этом окажется, что на пути 1 вагонов уже не осталось, то это значит, что наши усилия совсем зашли в тупик, и, следовательно, выстроить по порядку вагоны невозможно. Если же расстановка по порядку была возможна, то рано или поздно мы выиграем – все вагоны окажутся на пути 2. (Ведь каждый раз мы делали единственно возможный ход).

Таким образом, выигрыш – это расстановка вагонов на пути 2 в правильном порядке, проигрыш – невозможность сделать очередной ход, когда выигрыш не достигнут.

Теперь подумаем, как можно представить этот процесс в программе.

Введем основные переменные. (Используем здесь Паскаль; код на Си/Си++ – ниже.)

```
const
  max_n = 100; { - для лиги B; для лиги A max_n=2000 }
var
  N, expected, top, next : longint[3];
  st : array[1..max_n] of longint;
  way : array[1..max_n] of longint;
```

[1] Д. Кнут. Искусство программирования, том 1, § 2.2.1, упражнения 2, 3.

[2] Стек – это структура данных для хранения набора элементов, в которой поступление и уход элементов подчиняются правилу «последним пришёл – первым ушёл».

[3] Здесь и далее можно было бы использовать тип `integer`, но на современных компьютерах лучше всегда, кроме ситуаций явной нехватки памяти, использовать `longint`.

`N` – общее число вагонов;
`expected` – номер вагона, который ждут следующим на пути 2; значение *N*+1 будет означать, что все вагоны уже успешно выведены;
`st[1]` … `st[top]` – вагоны в тупике, перечисленные снизу вверх, то есть `st[1]` – дальний от выхода; когда тупик пуст, `top=0`;
`way[next]` … `way[n]` – вагоны на пути 1, перечисленные от головы к хвосту поезда; когда путь пуст, `next=N+1`.

Зададим начальные значения переменным:

Ввод данных: число вагонов – в `N`, их номера – в массив `way`.
```
top := 0;
next := 1;
expected := 1;
```
То есть тупик и путь 2 пусты, исходный состав находится на пути 1.

Далее напишем цикл `while`, за каждую итерацию[4] которого будем совершать ровно один ход, то есть перемещать ровно один вагон.

Сначала нужно решить, каким будет условие цикла. Возможны две ситуации завершения цикла: «выигрыш» или «проигрыш». Но обе эти ситуации объединяет невозможность сделать очередной ход.

Поэтому строим программу по такой схеме:
```
while можно сделать очередной ход do begin
   делаем ход
end;
проверяем, выиграли или проиграли.
```

Очередной ход можно сделать, если не пуст путь 1 (next ≠ N+1) или если сверху в тупике находится как раз нужный сейчас вагон ((top ≠ 0) and (st[top]=expected))[5]. Отсюда получаем условие цикла, а построение тела цикла должно быть понятно из предыдущих объяснений.

(Код программы – на следующей странице).

[4] Итерацией называется однократное выполнение тела цикла.

[5] Тут есть некоторая тонкость. Если при `top` = 0 будет проверяться часть составного условия, стоящая справа от `and`, то вычисление несуществующего элемента массива `st[0]` может привести к аварийному завершению программы. Как этого надёжно избежать? Если нет желания вникать в тонкости работы операции `and`, достаточно просто объявить массив с длиной на единицу больше, чем надо: `array[0..max_n]`, а лишний элемент `st[0]` никак не использовать.

Но на самом деле это не обязательно. Будет ли проверяться условие, стоящее справа от `and`, зависит от того, как выполняется операция `and` в данном языке программирования. При «полном» вычислении составного условия `X and Y` вычисляются всегда оба логических выражения `X` и `Y`, а при его «сокращённом» вычислении (short-circuit evaluation) `Y` будет вычисляться только в том случае, если `X` оказалось равно `true`. Ведь в противном случае и так ясно, что `X and Y = false`, и `Y` можно не вычислять. Таким образом, нам требуется «сокращённое» вычисление.

В языках Free Pascal, Delphi, C, C++, Java обычные логические операции `and` и `or` (`&&` и `||`) вычисляются всегда «сокращённо», а в языке Borland/Turbo Pascal – по-разному, в зависимости от настройки компилятора. Использованная нами далее директива компиляции `{$B-}` как раз включает режим «сокращенного» вычисления в Borland/Turbo Pascal. (Другие версии языка Паскаль просто не обратят на неё внимания).

```
{$B-}  { – см. сноску на предыдущей странице }
while (next <> N+1) or ((top <> 0) and (st[top]=expected)) do begin
  if (top <> 0) and (st[top]=expected) then begin { из тупика – на путь 2: }
    top := top - 1;
    expected := expected + 1;
  end else begin { с пути 1 – в тупик: }
    top := top + 1;
    st[top] := way[next];
    next := next + 1;
  end;
end;
```

Как отличить выигрыш от проигрыша? Когда выиграли, `expected=N+1`.

Приведём решение на Си/Си++; оно отличается лишь тем, что массивы нумеруются начиная с 0, а не с 1.

```
int top=-1; /*  st[0] ... st[top] – вагоны в тупике; когда тупик пуст top=-1  */
int next=0; /* way[next] ... way[N-1] – вагоны на пути 1; когда путь пуст, next=N */
int expected = 1; /*  номер вагона, который ждут следующим на пути 2
                  (значение N+1 будет означать, что все вагоны уже успешно выведены)  */

/* здесь нужно ввести исходные данные */

while (next!=N || (top!=-1 && st[top]==expected)) {
    if (top!=-1 && st[top]==expected) {
        top--;
        expected++;
    } else {
        top++;
        st[top] = way[next];
        next++;
    }
}
if (expected == N+1) printf("YES");
else printf("NO");
```

Это всё, что требовалось сделать в лиге B. В лиге A нужно было дополнительно к этому записывать ходы, чтобы в случае выигрыша вывести последовательность действий. Заметим, что при нашем решении K (число вагонов, перемещаемых за одно действие) всегда равно 1, поэтому запоминать нужно лишь типы действий (обозначаемые числами 1 и 2). Для этого достаточно одного массива размера $2N$.

Другие способы решения и оценка времени работы

Поскольку в лиге A требовалось записывать перемещения вагонов, то трудно было придумать какое-либо решение, принципиально отличающееся от моделирования их движения. Впрочем, решение это весьма эффективно: количество итераций не превышает $2N$, время одной итерации ограничено константой, поэтому время работы пропорционально N. (Или, как принято записывать в математике, – $O(N)$). Это решение проходило бы и при гораздо больших значениях N, чем указано в условии (например, при N около 10^5).

В лиге B выбор способов решения богаче. Например, можно доказать, что вагоны невозможно упорядочить в том и только в том случае, когда в первоначальном их расположении найдётся хотя бы одна такая тройка, что входящие в неё вагоны не могли бы быть упорядочены даже при отсутствии всех остальных вагонов. (К примеру, в последовательности 5, 1, 2, 3, 8, 4, 6, 7 такая «плохая» тройка – это 5, 8, 4). Было сдано несколько решений, основанных на поиске «плохих» троек. Заметим, что при $N \le 100$ проходит даже работающий за время $O(N^3)$ простой перебор всех троек (тремя циклами, вложенными друг в друга). Некоторыми ухищрениями можно уменьшить время работы до $O(N^2)$.