Задача 1. Перетасуем колоду карт

Рассмотрим задачу, которая довольно часто возникает при программировании игр (и не только игр). Необходимо расположить некоторый набор объектов в случайном порядке. Это может быть колода карт, набор костей домино или любые другие предметы, которые надо перемешать. Очевидно, что если мы научимся перемешивать просто числа от 1 до N, то сможем решить и любую другую подобную задачу.

Дополнительно потребуем, чтобы любая перестановка получалась в результате работы нашего алгоритма с одинаковой вероятностью и предположим, что в нашем распоряжении имеется идеальный датчик случайных чисел, которому можно задать произвольный интервал и получить с равной вероятностью любое целое число из этого интервала. (В стандартных библиотеках большинства реализаций языков программирования есть функция для получения случайного числа. Строго говоря, идеальными эти датчики не являются, но для наших целей их точность вполне удовлетворительна.)

Мы рассмотрим несколько решений этой задачи и проведем теоретический анализ их эффективности.

Решение 1

Будем получать от датчика случайные числа от 1 до N. Первое полученное число запишем. Каждое следующее будем по очереди сравнивать с уже записанными. Если это число уже встречалось, его надо отбросить и дальнейшие сравнения прекратить, а если еще нет — записать. Через некоторое время записанными окажутся все числа от 1 до N в случайном порядке, это и есть нужная последовательность.

Это решение гарантирует равную вероятность любой перестановки, потому что на каждом очередном шаге у всех еще не выписанных чисел равные шансы занять следующее место в последовательности.

На Паскале основной фрагмент программы, реализующей этот алгоритм, можно записать так:

```
k := 0;
while k < N do begin
   x := random (N) + 1;
   i := 1;
   while (i <= k) and (a[i] <> x) do i := i + 1;
   if i > k then begin
      k := k + 1;
      a[k] := x
   end
end;
```

Теперь займемся анализом. Главные действия в этом алгоритме — обращение к датчику и сравнение нового числа с уже записанными. Количест-

во этих действий случайно, оно зависит от того, какие результаты будет давать обращение к датчику. Вообще говоря, алгоритм может работать сколь угодно долго: теоретически возможно, что при любом числе обращений к датчику какое-то число ни разу не выпадет. На практике после какого-то (быть может, довольно большого) количества обращений последовательность будет сформирована.

Подсчитаем количество действий в среднем. Рассмотрим момент, когда для построения полной последовательности длины N не хватает еще k элементов. Пусть $d_k(N)$ — среднее количество обращений к датчику, необходимое в этот момент для получения очередного еще не использованного числа.

Найдем $d_k(N)$ для произвольных k и N. С вероятностью $q = \frac{k}{N}$ при очередном обращении к датчику выпадет новое число, с вероятностью $p = 1 - q$ — число, которое уже есть в списке.

Ясно, что вероятность обойтись одним обращением равна q, двумя — pq (первое обращение неудачно, второе — удачно), тремя — p^2q (первые два обращения неудачны) и т.д. Среднее число обращений получим, умножив каждую вероятность на соответствующее ей число обращений и сложив все эти произведения. Сумма получается бесконечной, но вычисляется она очень легко, так как после несложных преобразований сводится к бесконечной геометрической прогрессии:

$$
\begin{aligned}
d_k(N) &= q + 2pq + 3p^2q + 4p^3q + \ldots = \\
&= (1-p) + 2p(1-p) + 3p^2(1-p) + 4p^3(1-p) + \ldots = \\
&= 1 - p + 2p - 2p^2 + 3p^2 - 3p^3 + 4p^3 - 4p^4 + \ldots = \\
&= 1 + p + p^2 + p^3 + p^3 + p^4 + \ldots = \\
&= \frac{1}{1-p} = \frac{1}{q} = \frac{N}{k}
\end{aligned}
$$

Чтобы получить общее количество обращений к датчику, надо сложить значения d_k для всех k от N (последовательность пуста) до 1 (остался последний элемент):

$$
\begin{aligned}
D(N) &= d_1(N) + d_2(N) + \ldots + d_{N-1}(N) + d_N(N) = \\
&= \frac{N}{1} + \frac{N}{2} + \cdots + \frac{N}{N-1} + \frac{N}{N} = N\left(1 + \frac{1}{2} + \frac{1}{3} + \cdots + \frac{1}{N-1} + \frac{1}{N}\right)
\end{aligned}
$$

Ряд в скобках встречается в математике и в теоретической информатике довольно часто, и у него есть специальное название: *гармонический ряд*. Сумма первых N слагаемых гармонического ряда называется N-м гармоническим числом и обозначается H_N.

В математике доказывается, что при больших N соблюдается приближенное равенство $H_N \approx \ln N + \gamma$, где $\gamma = 0{,}5772...$ — число, называемое константой Эйлера. Это равенство становится более понятным, если заметить, что гармонический ряд — это не что иное, как интегральная сумма для функции $\frac{1}{x}$, а первообразная этой функции есть $\ln x$.

Окончательно получаем $D(N) = NH_N \approx N \ln N$.

Теперь подсчитаем количество сравнений $T(N)$. Как мы уже знаем, когда в последовательности не хватает k элементов из N, среднее количество обращений к датчику для получения одного нового элемента равно $\frac{N}{k}$. После каждого из этих обращений полученное число сравнивается с уже записанными. При последнем обращении полученное число оказывается новым; чтобы убедиться в этом, его необходимо сравнить со всеми уже записанными, то есть провести $N - k$ сравнений. При всех обращениях, кроме последнего, число сравнений может с равной вероятностью оказаться любым — от 1 до $N - k$, так как вновь полученное число с равной вероятностью совпадает с любым из уже записанных. Среднее число сравнений при этом равно $\frac{N-k+1}{2}$. Получаем

$$t_k(N) = (N-k) + \left(\frac{N}{k} - 1\right)\frac{N-k+1}{2} = \frac{(N-k)(N+k+1)}{2k} = \frac{N^2+N}{2k} - \frac{k+1}{2}$$

Просуммируем эти значения для всех k от 1 до N. Первое слагаемое из полученной формулы даст при суммировании уже знакомый нам гармонический ряд, а второе — обычную арифметическую прогрессию:

$$T(N) = t_1(N) + t_2(N) + \ldots + t_{N-1}(N) + t_N(N) =$$

$$= \left(\frac{N^2+N}{2} - \frac{2}{2}\right) + \left(\frac{N^2+N}{2\cdot 2} - \frac{3}{2}\right) + \cdots + \left(\frac{N^2+N}{2(N-1)} - \frac{N}{2}\right) + \left(\frac{N^2+N}{2N} - \frac{N+1}{2}\right) =$$

$$= \frac{N^2+N}{2}\left(1 + \frac{1}{2} + \cdots + \frac{1}{N}\right) - \frac{1}{2}(2+3+\ldots+N+1) = \frac{N^2+N}{2}H_N - \frac{N(N+3)}{4} =$$

$$= \frac{N^2}{2}\left(H_N - \frac{1}{2}\right) + \frac{N}{2}\left(H_N - \frac{3}{2}\right)$$

При больших N значение этой функции будет в основном определяться первым слагаемым. Можно сказать, что при больших N $T(N) \approx \frac{1}{2}N^2 \ln N$.

При анализе алгоритмов далеко не всегда удается построить точную формулу, определяющую количество тех или иных действий. Реальный интерес представляют оценки, показывающие, как растет количество дей-

ствий с ростом параметра. При этом чаще всего пренебрегают даже числовыми коэффициентами. Например, говорят, что $T(N)$ растет как $N^2 \ln N$, и записывают это так: $T(N) = O(N^2 \ln N)$.

Точный смысл понятия O (читается "о большое") объясняется в курсе математического анализа. Для нас сейчас важно не строгое математическое определение, а содержательная сущность полученной формулы: функция $T(N)$, которую мы исследуем, при больших значениях N ведет себя примерно так же, как функция $N^2 \ln N$, легко поддающаяся анализу.

Итак, мы выяснили, что данное решение требует в худшем случае бесконечного количества действий, а в среднем — порядка $N \ln N$ обращений к датчику случайных чисел и порядка $N^2 \ln N$ сравнений. Наш анализ показывает, что сравнений выполняется примерно в N раз больше, чем обращений к датчику. Поэтому, хотя сравнение — операция значительно более быстрая, при больших N именно сравнения будут определять реальную скорость работы алгоритма. С ростом N время работы будет расти как $N^2 \ln N$. Эта функция называется *вычислительной сложностью* алгоритма.

Решение 2

Как выяснилось, основное время при реализации решения 1 тратится на проверку, не записано ли уже в последовательности полученное от датчика число. Попробуем сократить количество проверок. Заведем таблицу, в которой будем отмечать, какие числа уже есть в последовательности. В программе этой таблице будет соответствовать дополнительный массив из N элементов. В начальный момент таблица пустая — никаких чисел в последовательности еще нет. Записывая в последовательность число i, мы отмечаем его использование в i-м элементе таблицы. Тогда каждое обращение к датчику потребует ровно одной проверки: соответствующий элемент таблицы покажет, встречалось ли ранее полученное число. В остальном алгоритм остается тем же, что и в решении 1. Вот соответствующий фрагмент программы:

```
for i := 1 to N do ready[i] := true;
k := 0;
while k < N do begin
   x := random(N) + 1;
   if ready[x] then begin
      k := k + 1;
      a[k] := x;
      ready[x] := false
   end
end;
```

Анализ этого алгоритма очень прост. Количество обращений к датчику, очевидно, будет таким же, как в решении 1, оно равно NH_N. Количе-

ство сравнений будет точно таким же, поскольку на каждое обращение приходится теперь ровно одно сравнение. Таким образом, вычислительная сложность этого решения $N \ln N$. Это намного лучше, чем в предыдущем случае, но нельзя забывать, что мы вычисляем среднее количество действий. А в худшем случае оно по-прежнему бесконечно.

Решение 3

Почему предыдущие решения требуют большого числа обращений к датчику? Дело в том, что на каждое следующее место в последовательности претендует на один элемент меньше, чем на предыдущее, а мы каждый раз запрашиваем у датчика один из N вариантов и вынуждены делать повторные обращения, если результат окажется неподходящим.

Чтобы решить эту проблему, будем запрашивать у датчика числа от 1 до k, где k — количество оставшихся мест в последовательности. При этом результат, полученный от датчика, надо рассматривать не как число, которое следует записать, а как порядковый номер числа среди тех, которые еще не записаны.

Поясним этот алгоритм на примере. Пусть $N = 7$, в начале последовательности записаны числа 3, 6 и 2. Таблица, описанная в решении 2, приобретает такой вид:

1	2	3	4	5	6	7
	+	+			+	

Необходимо записать в последовательность еще 4 элемента, поэтому при очередном обращении к датчику мы запрашиваем число от 1 до 4. Предположим, получилось 3. Это означает, что записать надо третье из еще не использованных чисел. Будем последовательно проверять все элементы таблицы, подсчитывая при этом пустые клетки. В третью пустую клетку поставим плюс, а ее номер покажет очередное число в последовательности. В нашем примере это будет 5.

Основной фрагмент Паскаль-программы:

```
for i := 1 to N do ready[i] := true;
for k := N downto 1 do begin
   x := random(k) + 1;
   i := 0; cnt := 0;
   while cnt < x do begin
      i := i + 1;
      if ready[i] then cnt := cnt + 1
   end;
   a[k] := i;
   ready[i] := false
end;
```

Займемся анализом. Все обращения к датчику в этом алгоритме результативны. Значит, их количество всегда будет равно N (или $N - 1$, если сообразить, что при последнем обращении результат заранее известен). Но зато опять увеличилось количество сравнений — ведь после каждого обращения к датчику надо пройти по таблице в поисках нужного элемента.

На первый взгляд подсчитать количество сравнений очень трудно. В самом деле, количество сравнений после каждого обращения к датчику зависит и от результата этого обращения, и от предыстории, то есть от результатов предыдущих обращений. Но вычислить общее число сравнений при выполнении алгоритма все-таки можно. Заметим, что каждый раз мы останавливаем проход по таблице на том элементе, номер которого заносим в последовательность. Поскольку все числа от 1 до N должны попасть в последовательность ровно по 1 разу, мы будем один раз делать 1 сравнение, один раз — 2 сравнения и т.д. Предсказать порядок здесь невозможно, он случаен и совпадает с порядком элементов полученной случайной перестановки, но общее количество сравнений легко находится:

$$T(N) = 1 + 2 + \ldots + N = \frac{N(N+1)}{2} = O(N^2)$$

Итак, алгоритм в решении 3 имеет вычислительную сложность N^2. Такие алгоритмы называют *квадратичными*.

Заметим, что $N^2 > N \ln N$, то есть третий алгоритм медленнее второго. Но зато у третьего найденное количество действий постоянно, а у второго достигается только в среднем, а в худшем случае может быть сколь угодно большим.

На практике это означает, что в среднем второй алгоритм быстрее, но в отдельных случаях он может работать очень долго. Третий алгоритм в среднем медленнее, но зато гарантированно завершается за фиксированное время.

Подобная ситуация довольно типична. Одно решение оказалось быстрее в среднем, другое — в худшем случае. Какое выбрать? Это зависит от того, где применяется программа. Например, если нужно многократно выполнять какие-то расчеты, важна бывает средняя скорость и не страшно, если иногда единичный расчет займет много времени. А для программ, которые должны работать в экстренных ситуациях при управлении космическими кораблями или атомными станциями, важна скорость именно в худшем случае: надо гарантировать, что результат будет получен вовремя.

Впрочем, в нашей задаче подобный выбор на самом деле не возникает. Ведь мы рассмотрели еще не все решения.

Решение 4

В решении 3 время уходило на поиск нужного элемента в таблице. Сделаем так, чтобы результат, полученный от датчика, сразу показывал номер этого элемента. Для этого будем заносить в таблицу не плюсики, а

сами числа. В начальный момент в таблице находятся все числа от 1 до N, каждый раз, когда очередное число заносится в выходную последовательность, будем сдвигать элементы таблицы так, чтобы все неиспользованные значения группировались в начале.

Снова рассмотрим пример. Пусть $N = 7$, в начале итоговой последовательности записаны числа 3, 6 и 2. Таблица в этот момент выглядит так:

1	2	3	4	5	6	7
1	4	5	7			

Содержимое элементов с 5-го по 7-й нас уже не интересует. Если теперь при обращении к датчику будет получено значение 2, в выходную последовательность попадет второй элемент таблицы, то есть число 4, а в таблице произойдет очередной сдвиг: 5 и 7 переместятся, соответственно, во вторую и третью клетки.

Программа получается такая:

```
for i := 1 to N do list[i] := i;
for k := N downto 1 do begin
   x := random(k) + 1;
   a[k] := list[x];
   for i := x to k - 1 do begin
      list[i] := list[i + 1]
   end
end;
```

Очевидно, что, как и в предыдущем алгоритме, количество обращений к датчику равно N. Никаких сравнений здесь нет, но появились сдвиги. Количество сдвигаемых элементов на каждом шаге может быть различным: от нуля (если выпал последний из оставшихся элементов) до числа оставшихся элементов $k - 1$ (если выпал первый элемент). Поскольку все элементы выпадают с равной вероятностью, среднее число сдвигов равно $\frac{k-1}{2}$. В ходе работы k изменяется от N до 1, общее количество сдвигов равно

$$V(N) = \frac{N-1}{2} + \frac{N-2}{2} + \cdots + \frac{1}{2} + 0 = \frac{N(N-1)}{4} = O(N^2)$$

Алгоритм опять получился квадратичный. В вычислительной сложности мы ничего не выиграли. Сравнение точных формул для количества сравнений в решении 3 и количества сдвигов в решении 4 показывает, что коэффициент при N^2 уменьшился, так что, возможно, какой-то выигрыш мы все же получим. Правда, мы фактически считали количество других операций (сдвигов, а не сравнений), так что многое будет зависеть от того, как реализованы эти операции, сколько времени занимает каждая из них.

Решение 5

Попробуем обойтись без сдвигов. Они нужны только для того, чтобы избавиться от дыр в таблице. Но ведь сохранять при этом порядок элементов совершенно не обязательно! Поскольку все числа при построении случайной перестановки равноправны, они могут располагаться в таблице в любом порядке, важно только, чтобы у них сохранялись одинаковые шансы на очередное место в последовательности. Поэтому вместо сдвига можно просто переносить на освободившееся место в таблице ее последний элемент. (Конечно же имеется в виду не абсолютно последний N-й элемент, а текущий последний, k-й.)

Программа:

```
for k := 1 to N do list[k] := k;
for k := N downto 1 do begin
   x := random(k) + 1;
   a[k] := list[x];
   list[x] := list[k]
end;
```

Теперь на каждое из N обращений к датчику приходится единственный перенос элемента. Вычислительная сложность алгоритма — N, такие алгоритмы называют *линейными*.

Решение 6

Очевидно, что дальнейшее снижение сложности невозможно. Получить последовательность из N элементов меньше чем за N действий просто нельзя.

Но, кроме времени, есть еще и память. Для реализации решения 5 требуется $2N$ элементов памяти: для полученной перестановки и для вспомогательной таблицы. Оказывается, N элементов вполне достаточно.

В решении 5 мы выводим элемент из таблицы в итоговую последовательность, одновременно переносим на освободившееся место последний элемент, а последнее место в дальнейшем не используем.

Давайте поместим на это последнее место тот элемент, который мы выбрали для включения в перестановку. Тем самым все сводится к обмену последнего элемента со случайно выбранным, а случайная перестановка будет постепенно формироваться с конца таблицы.

Вот окончательный, наиболее эффективный, вариант программы случайной перестановки:

```
for k := 1 to N do a[k] := k;
for k := N downto 1 do begin
    x := random(k) + 1;
    t := a[k];
    a[k] := a[x];
    a[x] := t
end;
```

Эффективность и правильность

В погоне за эффективностью нельзя забывать о том, что программа должна в первую очередь давать правильный результат. Если задача решается неверно, нет никакой пользы в том, что это делается быстро.

В задаче о случайной перестановке критерием правильности служит равномерность полученного распределения: у всех возможных перестановок должны быть равные шансы. Специфика задачи не позволяет оценить достижение этой цели по результатам однократного запуска: мы получим какую-то перестановку, но ничего не сможем сказать о том, насколько случайно она выбрана. Несколько запусков подряд позволят увидеть явные перекосы (например, если забыть инициализировать датчик случайных чисел, будет все время выдаваться одна и та же перестановка), но не помогут оценить общий уровень равномерности.

Конечно, случайность выбора перестановки во всех рассмотренных решениях гарантирована самим построением алгоритмов: в любой момент все еще не включенные в перестановку элементы имеют равные шансы занять очередное место. Но хочется получить и экспериментальное подтверждение этого факта.

Понятно, что при больших N проанализировать равномерность сложно: количество различных перестановок столь велико, что получить каждую из них даже по одному разу практически невозможно. А вот при сравнительно небольших N оценить равномерность вполне возможно.

Напишем для начала небольшую функцию, которая по заданной перестановке вычисляет ее номер в общем лексикографическом списке. Более точно, эта функция вычисляет количество перестановок, расположенных в списке раньше данной (этот показатель можно считать номером перестановки, если начинать нумерацию с нуля).

Детальный разбор алгоритма этой функции выходит за рамки данной работы, поэтому ограничимся текстом на Паскале:

```
function lexNnum (a: array of integer; N: integer) : integer;
   var i, j: integer;
       count: integer;
       res: integer;
begin
   res := 0;
   for i := 0 to N - 1 do begin
      count := 0;
      for j := i + 1 to N - 1 do begin
         if a[j] < a[i] then inc(count)
      end;
      res := res*(N - i) + count
   end;
   lexNum := res
end;
```

Теперь можно завести массив счетчиков, в котором будет $N!$ элементов (вот где важно требование, что N для проведения анализа должно быть невелико), многократно генерировать случайную перестановку с помощью любого из разобранных алгоритмов и подсчитать, сколько раз выпала каждая конкретная перестановка. Вот основной фрагмент программы, реализующий этот анализ:

```
{ Nfact = N!, количество перестановок
  NRepeat — заданное среднее количество выпадений каждой
            перестановки }
for i := 1 to Nfact do c[i] := 0;
for i := 1 to NRepeat*Nfact do begin
  shuffle(a, N);{вызов процедуры построения перестановки}
  num := lexNum(a, N) + 1;
  c[num] := c[num] + 1
end;
cmin := c[1]; cmax := c[1];
for i := 2 to Nfact do begin
   if c[i] < cmin then cmin := c[i];
   if c[i] > cmax then cmax := c[i]
end;
writeln (N, cmin:6, cmax:6, cmax/cmin:7:3);
```

С помощью построенной на основе этого фрагмента программы было проведено тестирование всех вышеприведенных решений при $N = 5$ и `Nrepeat` = 10 000. Наибольшее зарегистрированное отношение `cmax/cmin` составило 1,066, что свидетельствует об очень хорошей равномерности распределения перестановок.

Для сравнения рассмотрим еще один алгоритм решения этой задачи, описанный в одном (в целом очень хорошем) учебнике, посвященном конкретному языку программирования.

В этом решении каждый элемент исходной упорядоченной перестановки по очереди обменивается со случайно выбранным элементом. Алгоритм имеет линейную сложность, и авторы справедливо указывают на его высокую эффективность, особенно в сравнении с другим разобранным в этой же книге алгоритмом, аналогичным нашему решению 2.

Решение, предложенное авторами учебника, очень похоже на наше решение 6 и отличается от него только в одном пункте: при каждом обращении к датчику случайных чисел выбирается число из постоянного интервала от 1 до N. Вот соответствующий фрагмент программы:

```
for k := 1 to N do a[k] := k;
for k := 1 to N do begin
   x := random (N) + 1;
   t := a[k];
   a[k] := a[x];
   a[x] := t
end;
```

К сожалению, это решение нельзя признать верным, так как оно не обеспечивает равномерности. Это можно доказать даже теоретически. В программе происходит N обращений к датчику, каждое из которых может вернуть любое из N значений. Таким образом, всего возможно N^N различных исходов. Каждому исходу соответствует ровно одна полученная в результате перестановка. Поскольку N^N не делится на число возможных перестановок $N!$, различные перестановки будут получаться с неодинаковой частотой.

Экспериментальная проверка по описанной выше схеме подтверждает этот факт: отношение `cmax/cmin` в проведенном эксперименте составило 3,021, то есть одни перестановки выпадали втрое чаще, чем другие.