Задача 2. Самый длинный возрастающий ряд

Дана числовая последовательность из N элементов ($1 \le N \le 10\,000$). Требуется выбрать из нее, не нарушая порядка, как можно больше элементов, образующих монотонно возрастающий ряд.

Чтобы условие стало более понятным, рассмотрим пример. Пусть задана такая последовательность из 14 элементов:

8	2	12	17	11	4	15	7	14	9	1	10	13	3

Если бы не требование сохранить порядок, мы получили бы классическую задачу сортировки и построили бы возрастающий ряд длины 14.

Если бы требовалось выбирать из исходной последовательности элементы, идущие подряд, мы бы построили однопроходный алгоритм и нашли ряд длины 3: 2, 12, 17 или 1, 10, 13.

Но в этой задаче можно включать в ряд не соседние элементы и нельзя менять их местами, поэтому для заданной последовательности правильным ответом будет ряд длины 6: 2, 4, 7, 9, 10, 13.

Решение 1

Будем выбирать из заданной последовательности все возможные комбинации элементов, проверять каждую комбинацию на монотонное возрастание и запоминать самый длинный подходящий ряд.

Вот полный текст программы на Паскале, реализующей это решение.

```
program effect_2_1;
{ Задача 2: выделить из заданной последовательности
            подпоследовательность максимальной длины
  Решение 1: перебор всех подмножеств }
  const NMAX = 10000;
```

14

```
var
    a: array[1..NMAX] of integer;
                          {данная последовательность}
    N: integer;    {количество элементов}
    subset: array[1..NMAX] of boolean;
                                {задание подмножества}
    best: array[1..NMAX] of integer;    {лучший ряд}
    lmax: integer;     {длина лучшего ряда}
 procedure input;
 { Ввести последовательность из файла.
   Определить количество элементов. }
    var f: text;
 begin
    assign(f, 'in.txt');
    reset(f);
    N := 0;
    while not eof(f) do begin
       inc(N);
       readln(f,a[N])
    end
 end;
 procedure sub_init;
 { Создать пустое подмножество }
    var i: integer;
 begin
    for i := 1 to N do subset[i] := false
 end;
 function sub_next: boolean;
 { !!! Функция с побочным эффектом !!!
    Переходит к следующему подмножеству.
    Возвращает true при успешном переходе. }
    var i: integer;
 begin
    i := N;
    while (i > 0) and subset[i] do begin
       subset[i] := false;
       dec(i)
    end;
    if i > 0 then subset[i] := true;
    sub_next := i > 0
 end;
 function check: integer;
 { Проверить текущее подмножество.
   Если получился возрастающий ряд, вернуть его длину.
   Если возрастающего ряда нет, вернуть ноль. }
```

```
 var last: integer; {последний элемент ряда}
     grow: boolean;
                  {подмножество задает возрастающий ряд}
     len, i: integer;
begin
  {Считаем, что все заданные элементы положительны.
   В общем случае можно взять любое значение, меньшее
   минимального элемента последовательности.}
   last := -1;
   len := 0;
   grow := true;
   i := 1;
   while (i <= N) and grow do begin
      if subset[i] then begin
         if a[i] > last then begin
            inc(len);
            last := a[i]
         end
         else grow := false
      end;
      inc(i)
   end;
   if grow then check := len
           else check := 0
end;

procedure copy_best;
{ Запомнить найденный ряд. }
   var i, k: integer;
begin
   k := 0;
   for i := 1 to N do begin
      if subset[i] then begin
         inc(k);
         best[k] := a[i]
      end
   end
end;
procedure process;
{ Найти самый длинный возрастающий ряд }
   var l: integer;
begin
   lmax := 0;
   sub_init;
```

```
        while sub_next do begin
           l := check;
           if l > lmax then begin
              lmax := l;
              copy_best
           end
        end
     end;
     procedure output;
     { Вывести полученные результаты }
        var i: integer;
     begin
       writeln('Наибольшая длина возрастающего ряда=',lmax);
       for i := 1 to lmax do write(best[i]:8);
       writeln
     end;
  begin
     input;
     process;
     output
  end.
```

Общее количество комбинаций равно количеству подмножеств множества из N элементов, то есть 2^N. Даже если не учитывать сложность организации перебора подмножеств и проверки каждого из них, получаем, что сложность алгоритма составляет не меньше, чем $O(2^N)$.

Попробуем понять, что это означает на практике. Предположим, что за секунду наш компьютер успевает перебрать миллион подмножеств. При такой скорости за секунду (точнее, чуть больше чем за секунду) можно решить задачу при $N = 20$, так как $2^{20} = 1\,048\,576$. Но мы ведь никуда не торопимся! Подождем минуту. За 60 секунд можно перебрать 60 миллионов комбинаций. Это позволит нам решить задачу для $N = 25$, а если подождать еще несколько секунд, то для $N = 26$. Не слишком обнадеживающий результат... Подождем час. За 60 минут можно перебрать 3,6 миллиарда комбинаций. Этого хватит для решения задачи при $N = 31$. Дальнейшее увеличение времени не дает возможности продвинуться существенно дальше. За сутки можно решить задачу для $N = 36$, за год — для $N = 44$.

Может быть, надо вооружиться более мощным компьютером? Предположим, что на нашем новом компьютере все вычисления выполняются в миллион раз быстрее, чем на старом. Увы, это не спасает положения: такое ускорение всего лишь позволит обрабатывать за то же время на 20 элементов больше. За секунду мы решим задачу для $N = 40$, а за год — для $N = 64$. Ясно, что до заданного в условии задачи $N = 10\,000$ за сколько-нибудь реальное время нам не добраться.

Что же получается? Задача вроде бы решена. Есть заведомо правильный алгоритм, есть программа, выдающая правильные результаты. Но время работы этой программы столь велико, что никакой практической пользы от нее нет! А значит, задачу можно фактически считать нерешенной.

Причина этой неприятности в том, что выбранный алгоритм требует порядка 2^N действий. Значит, как бы хорошо мы ни программировали, какой бы мощный компьютер ни взяли, время счета всегда будет пропорционально 2^N. Эта функция растет очень быстро: увеличение N всего ***на*** единицу приводит к росту ***в*** два раза. Чтобы удлинить последовательность на 10 элементов, нужно увеличить время работы или скорость компьютера в 1000 раз. Понятно, что чаще всего сделать это невозможно.

Про алгоритмы, выполнение которых требует порядка a^N действий ($a > 1$), говорят, что они имеют *экспоненциальную* сложность. На практике экспоненциальные алгоритмы можно использовать только при сравнительно небольших значениях N.

Решение 2

Перебор надо сокращать. Нет смысла рассматривать абсолютно все возможные подмножества, многие из них можно отсечь "на дальних подступах". Вернемся к примеру, на котором мы разбирали условие задачи. Если мы включили в комбинацию восьмерку, открывающую исходную последовательность, то стоящую за ней двойку можно уже не рассматривать. Тем самым мы сразу исключаем из рассмотрения 2^{12} заведомо неподходящих комбинаций, начинающихся с невозрастающего фрагмента (8, 2). При больших N и систематическом применении таких отсечений выигрыш может оказаться значительным.

Опишем соответствующий алгоритм более детально все на том же примере. Попробуем начать возрастающий ряд с первого элемента последовательности (8). Тогда 2 ставить на второе место нельзя, а 12 — можно. На третье место можно поставить 17, после этого продолжить ряд не удается. Фиксируем достигнутый результат (ряд из 3 элементов), убираем последний элемент (17), пробуем заменить его другим. 11 и 4 не подходят, 15 дает ряд длины 3, 7 не подходит, 14 тоже дает ряд длины 3, 9 и 1 не подходят, а вот после 10 можно поставить 13, получив ряд длины 4. Фиксируем этот результат, убираем последний элемент (13), заменить его нечем, убираем последний из оставшихся и т.д. Когда будут исчерпаны все варианты, начинающиеся с 8, поставим на первое место второй элемент исходной последовательности (2), затем третий и т.д.

Такой алгоритм называется *перебор с возвратом*, или *бектрекинг* (от англ. *backtracking*). Он позволяет перебрать все возрастающие ряды, кото-

рые можно получить из данной последовательности, не тратя время на рассмотрение заведомо неподходящих комбинаций.

Несмотря на сложное описание, программная реализация бектрекинга оказывается сравнительно короткой.

```
program effect_2_2;
{ Задача 2: выделить из заданной последовательности
  подпоследовательность максимальной длины.
  Решение 2: перебор с возвратом. }
   const NMAX = 10000;
   var
      a: array[0..NMAX] of integer;
                              {данная последовательность}
      N: integer;             {количество элементов}
      row: array[0..NMAX] of integer;{индексы ряда}
      best: array[1..NMAX] of integer;    {лучший ряд}
      lmax: integer;          {длина лучшего ряда}
   procedure input;{Такая же, как в решении 1}
   procedure copy_best;
   { Запомнить найденный ряд }
      var i: integer;
   begin
      for i := 1 to lmax do best[i] := a[row[i]];
   end;
   procedure process;
   {Найти самый длинный возрастающий ряд }
      var ia, ir: integer;
   begin
     lmax := 0;
     a[0] := -1; {число, заведомо меньшее всех элементов}
     row[0] := 0;
     ia := 1; ir := 1;
     while ir > 0 do begin
         if ia > N then begin {возврат}
            dec(ir);
            if ir > lmax then begin
               lmax := ir;
               copy_best
            end;
            ia := row[ir]
         end
         else if a[ia] > a[row[ir - 1]] then begin
            row[ir] := ia;
            inc(ir)
         end;
         inc(ia)
     end
   end;
```

```
   procedure output;
{Такая же, как в решении 1}
begin {Главная программа — такая же, как в решении 1}
…
end.
```

Опустим детальный анализ перебора с возвратом, приведем только окончательный результат. Вычислительная сложность этого алгоритма составляет $O((1 + p)^N)$, где p — величина от 0 до 1, характеризующая долю возрастающих рядов среди всех возможных подмножеств исходной последовательности.

Таким образом, это решение тоже экспоненциальное. Однако основание показательной функции теперь зависит от того, как расположены данные в исходной последовательности. В худшем случае (исходная последовательность упорядочена по возрастанию) $p = 1$, и мы фактически повторяем алгоритм первого решения. Во всех остальных случаях p будет меньше 1, и перебор с возвратом окажется быстрее полного перебора. Впрочем, выигрыш на самом деле оказывается не столь существенным. В таблице показано, для какого значения N удается решить задачу за фиксированное время в зависимости от значения p и скорости выполнения вычислений.

Время счета	p (доля подходящих комбинаций)							
	1		0,5		0,1		0,01	
	Количество комбинаций в секунду							
	10^6	10^9	10^6	10^9	10^6	10^9	10^6	10^9
1 сек.	19	29	34	51	144	217	1388	2082
1 мин.	25	35	44	61	187	260	1799	2494
1 час	31	41	54	71	230	303	2211	2905
1 сутки	36	46	62	79	264	336	2530	3225
1 неделя	39	49	66	83	284	357	2726	3420
1 год	44	54	76	93	326	398	3123	3817
1 век	51	61	88	105	374	446	3586	4280

Видно, что при небольших значениях p можно за то же самое время обработать существенно больше данных, чем при полном переборе. Эта разница часто оказывается достаточной, и для многих практических задач небольшого размера перебор с возвратом оказывается удовлетворительным решением.

Однако для больших наборов данных он непригоден. Наша задача все еще не решена: даже при самых оптимистичных предположениях для обработки последовательности из 10 тысяч элементов требуется больше ста лет.

Решение 3

При переборе с возвратом мы многократно выполняем одни и те же проверки. Каждый раз, когда некоторый элемент исходной последовательности вклю-

чается в итоговый ряд, мы сравниваем его со всеми последующими, выстраивая те же самые варианты окончания ряда. Вместо этого можно один раз сравнить каждый элемент со всеми последующими и запомнить результат.

Будем обрабатывать исходную последовательность с конца. Пусть в какой-то момент мы уже обработали все элементы с номерами, большими некоторого k, и для каждого элемента a_i ($i > k$) вычислили l_i — наибольшую возможную длину возрастающего ряда, начинающегося с a_i. Теперь сравним a_k со всеми уже обработанными элементами. Если $a_i > a_k$, значит, элемент a_i можно поставить после a_k, получив при этом ряд длины $l_i + 1$. Выбрав среди всех элементов, для которых $a_i > a_k$, имеющий максимальное значение l_i, окончательно определим l_k. Если элементов с $a_i > a_k$ нет, продолжить ряд после a_k нельзя, в этом случае $l_k = 1$.

Когда будет обработана вся последовательность, максимальное значение l_k покажет длину искомого ряда. Чтобы восстановить сам ряд, нужно в ходе обработки для каждого элемента запоминать не только длину ряда, но и индекс того элемента, который надо поставить после данного, чтобы обеспечить эту длину. Этот алгоритм решения основан на *методе динамического программирования*.

(Тот же результат можно получить, обрабатывая последовательность не с конца, а с начала, но в этом случае будет чуть сложнее не просто определить длину возрастающего ряда, но и построить этот ряд.)

Общая структура программы такая же, как и в предыдущих решениях. Приведем две основные процедуры:

```
procedure process;
{ Найти самый длинный возрастающий ряд }
   var i, j: integer;
begin
   lmax := 0;
   for i := N downto 1 do begin
      len[i] := 1;
      next[i] := 0;
      for j := i + 1 to N do begin
         if (a[j] > a[i]) and (len[j] >= len[i]) then begin
            len[i] := len[j] + 1;
            next[i] := j
         end;
      end;
      if len[i] > lmax then begin
         lmax := len[i];
         imax := i
      end
   end
end;
procedure output;
{ Вывести полученные результаты }
   var i: integer;
```

```
begin
  writeln('Наибольшая длина возрастающего ряда=',lmax);
 i := imax;
 while i > 0 do begin
    write(a[i]:8);
    i := next[i]
 end;
 writeln
end;
```

В таблице показаны результаты применения метода динамического программирования к тестовому примеру.

Индекс	1	2	3	4	5	6	7	8	9	10	11	12	13	14
Значение	8	2	12	17	11	4	15	7	14	9	1	10	13	3
Длина	3	6	2	1	2	5	1	4	1	3	3	2	1	1
Индекс следующего	3	6	4	0	7	8	0	10	0	12	12	13	0	0

Самая массовая операция в этом алгоритме — сравнение. Каждые два элемента исходной последовательности сравниваются между собой, общее количество сравнений элементов равно $\frac{N(N-1)}{2}$, вычислительная сложность алгоритма — $O(N^2)$.

Традиционно предположим, что за секунду наш компьютер выполняет миллион сравнений вместе с сопутствующими каждому сравнению операциями. Тогда за секунду удастся обработать 1414 элементов, а для 10 000 элементов потребуется 50 секунд. Такой результат можно считать практически удовлетворительным: минута — вполне допустимое время ожидания.

Решение 4

В предыдущем решении мы попарно сравнивали между собой все элементы исходной последовательности. На самом деле в этом нет необходимости: очередной элемент достаточно сравнить лишь с некоторыми из уже обработанных.

Рассмотрим один из заключительных моментов в ходе предыдущего решения, когда обработаны все элементы, кроме самого первого:

Индекс	1	2	3	4	5	6	7	8	9	10	11	12	13	14
Значение	8	2	12	17	11	4	15	7	14	9	1	10	13	3
Длина	?	6	2	1	2	5	1	4	1	3	3	2	1	1
Индекс следующего	?	6	4	0	7	8	0	10	0	12	12	13	0	0

Среди 13 обработанных элементов для 5 максимальная длина ряда равна 1. Сравнивая первый элемент с этими пятью, мы узнаем, может ли он стать началом ряда длины 2. Но для того, чтобы дать положительный ответ на этот вопрос, достаточно, если первый элемент будет меньше хотя бы одного из указанных 5 элементов, а значит, имеет смысл сравнивать его только с максимальным из них. Точно так же достаточно сравнить его лишь с максимальным из всех элементов, начинающих ряды длины 2, 3 и т.д. При этом, если результат очередного сравнения окажется отрицательным, процесс можно прекратить: если текущий элемент не может начать ряд длины m, он никак не сможет начать ряд большей длины.

Получается, что нужно хранить значение максимального начального элемента для каждой длины ряда и сравнивать очередной элемент только с этими максимумами. Если очередной элемент окажется меньше зафиксированного максимума для некоторой длины m, но больше максимума для длины $m + 1$, этот элемент может начать ряд длины $m + 1$, и мы заменяем этим (бóльшим!) элементом записанный для $m + 1$ максимум. Если же очередной элемент оказывается меньше всех найденных максимумов, мы фиксируем увеличение наибольшей длины ряда. В любом случае каждый очередной элемент обязательно попадает в текущий список максимумов, возможно, вытесняя оттуда один из ранее записанных элементов.

Продемонстрируем этот алгоритм, подробно рассмотрев все шаги для нашего тестового примера.

Индекс очередного элемента	Длина ряда					
	1	2	3	4	5	6
14	***3***					
13	***13***					
12	13	***10***				
11	13	10	***1***			
10	13	10	***9***			
9	***14***	10	9			
8	14	10	9	***7***		
7	***15***	10	9	7		
6	15	10	9	7	***4***	
5	15	***11***	9	7	4	
4	***17***	11	9	7	4	
3	17	***12***	9	7	4	
2	17	12	9	7	4	***2***
1	17	12	9	***8***	4	2

По результатам обработки можно сказать, что максимальный возрастающий ряд имеет длину 6, и начинаться он должен с элемента, равного 2. Однако построить весь ряд заполненная выше таблица не позволяет. Зафиксированная в ней последовательность не может быть получена из исходной. Для полного решения исходной задачи, включая построение ряда, нужно для каждого элемента запоминать индекс следующего, как это делалось в предыдущем решении.

Если не требовать построения ряда и ограничиться определением его длины, можно обрабатывать исходную последовательность не с конца, а с начала (и хранить при этом список не максимальных первых элементов рядов различной длины, а минимальных последних), читать данные из файла, а в памяти хранить не всю последовательность, а только упомянутый список. В этом случае удается решить задачу для очень больших значений N.

Оценим сложность алгоритма. Для каждого из N элементов выполняется не более L сравнений, где L — длина искомого возрастающего ряда. Сложность алгоритма составляет $O(NL)$. В наихудшем случае, когда вся последовательность возрастает, имеем $L = N$ и сложность $O\left(N^2\right)$, такую же, как и в предыдущем случае.

Заметим, однако, что список максимумов (или минимумов при обработке с начала исходной последовательности) представляет собой упорядоченный массив, в котором можно проводить не линейный, а двоичный поиск. Тогда количество сравнений для каждого элемента будет не более $\log L$, а сложность алгоритма составит $N \log L$, в худшем случае — $N \log N$.

С учетом этого улучшения получаем основную процедуру программы:

```
procedure process;
{ Найти самый длинный возрастающий ряд }
   var i, lt,rt,md: integer;
begin
   lmax := 0; imax[0] := 0;
   for i := N downto 1 do begin
      lt := 1; rt := lmax;
      while lt <= rt do begin
         md := (lt + rt) div 2;
         if a[i] < a[imax[md]]
            then lt := md + 1
            else rt := md - 1;
      end;
      if lt > lmax then inc(lmax);
      imax[lt] := i;
      next[i] := imax[lt - 1];
   end;
end;
```

При обычном предположении о миллионе сравнений и сопутствующих им операций в секунду получаем, что для $N = 10\,000$ достаточно одной десятой секунды, а за секунду можно обработать более 60 000 элементов.