Поиск в массиве

Линейный поиск

Рассмотрим следующую задачу.

Пусть дан одномерный массив a из N элементов. Найти индекс элемента, равного заданному числу k.

Пусть для определённости нумерация элементов начинается с единицы. Существенного влияния на алгоритмы решения задачи это не оказывает.

Если никакой дополнительной информации о массиве и разыскиваемом элементе нет, то вполне оправдан следующий подход. Возьмём первый элемент массива и сравним его с k. Если равенства не обнаружено, повторим сравнение с k для второго элемента, третьего и т.д., пока или не найдём требуемый элемент, или не просмотрим весь массив, так и не встретив искомого элемента. Такой

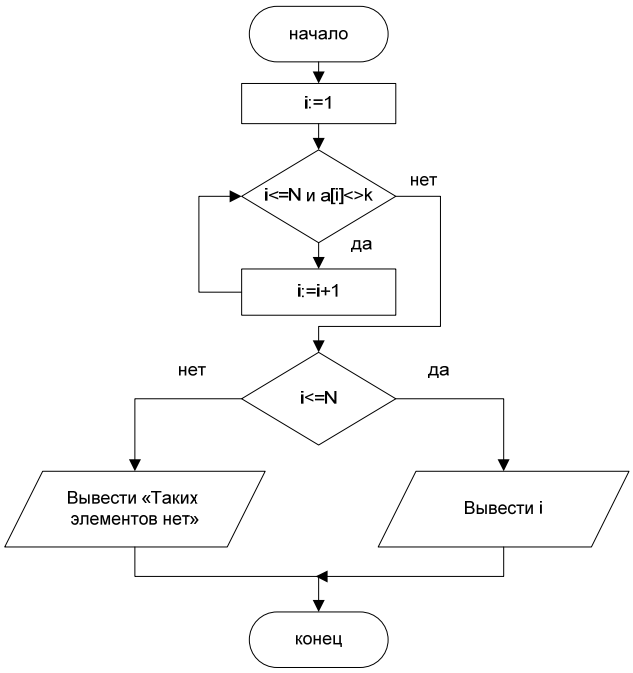

Рисунок 1. Линейный поиск

метод называют линейным, или последовательным, поиском.
Заметим, что если в массиве есть несколько элементов, равных k, то с помощью этого алгоритма мы найдём первый из них.

При написании программы будем придерживаться следующего соглашения о формате входных данных. Данные вводятся из файла input.txt, в первой строке которого находятся числа N ($0 < N \leq 1000$) и k ($|k| \leq 10^9$), а начиная со второй N целых чисел, по модулю не превышающих 10^9. Тем не менее, предпримем меры для быстрой перестройки программы на работу с другими исходными данными и их иным количеством. С этой целью для типа элементов массива, как и в ранее написанных нами программах, введём пользовательский тип TElem, а ограничение на количество элементов массива зададим с использованием константы MAXN. В выходной файл output.txt запишем индекс найденного элемента или “Not found”, если элемент не найден.

```
{$mode delphi}
Program SeqSearch;
Const MAXN = 1000;

Type TElem = Integer;
     TMyArray = array[1..MAXN] Of TElem;
```

```
Var a    : TMyArray;
    k    : TElem;
    i, N : Integer;
    f    : Text;
Begin
  Assign(f,'input.txt');
  Reset(f);
  Read(f,N,k);
  If N > MAXN Then
  Begin
    WriteLn('Количество элементов массива слишком велико: ', N ,'>',
MAXN);
    Halt
  End;

  For i:=1 To N Do
  Read(f,a[i]);

  Close(f);
  Assign(f,'output.txt');
  Rewrite(f);

  (* Собственно поиск *)
  {$B-}
  i:=1;
  While (i<=N) and (a[i]<>k) Do
  i:=i+1;

  If i<=N Then
    Write(f,i)
  Else
    Write(f,'Not found');
  Close(f)
End.
```

Обратите внимание, что в цикле While важен порядок операций сравнения `i<=N` и `a[i]<>k`. По умолчанию и FreePascal, и Delphi продолжают выполнение логических операций *and* и *or* лишь до тех пор, пока не ясен результат. Представьте теперь, что ни один элемент массива не равен k. Тогда в последнем проходе цикла значение `i` будет равно `N+1`. В случае

```
While (i<=N) and (a[i]<>k) Do
```

результатом логического выражения `i<=N` будет false. Этого достаточно, чтобы определить значение всего выражения `(i<=N) and (a[i]<>k)`. Это тоже false вне зависимости от результата второго сравнения. Поэтому значение выражения `a[i]<>k` вычисляться не будет. Если же в программе написать

```
While (a[i]<>k) and (i<=N) Do
```

то есть опасность выхода за пределы массива. Если N=MAXN, пришлось бы обращаться к элементу массива с индексом MAXN+1, а такого элемента в массиве нет. Поведение программы в такой ситуации зависит от настроек компилятора.

У компилятора есть опция «Range checking». В IDE FreePascal ищите её в меню Options – Compiler – Generated code, а в Delphi в меню Project – Options – Compiler в группе Runtime errors. Если эта опция включена, компилятор генерирует код проверки выхода за пределы массива. Если при выполнении программы есть попытка доступа к несуществующему элементу массива, то программа завершается с ошибкой. Так случилось бы и в описанном выше случае. Если эту опцию не включать, выход за пределы массива не проверяется. В нашем случае это и не нужно, но часто указанная проверка бывает полезной. Эту опцию можно также включить, написав в тексте программы {$R+}, а выключить — {$R-}.

Другая опция компилятора «Complete boolean eval» управляет вычислением логических выражений. Если она выключена, то сложные логические выражения вычисляются лишь до тех пор, пока не ясен их результат. Если включена, то вычисление идёт до конца, то есть значения

всех выражений, объединённых при помощи операций *and* и *or*, находятся обязательно. В IDE Delphi эта опция находится там же, где и Range checking, но в группе «Syntax options». В IDE FreePascal я эту опцию не нашёл, но её можно задать в тексте программы, написав {$B+} (включить) или {$B-} (выключить). Так можно поступить и в Delphi. Опции, заданные в тексте программы при помощи {$...}, имеют преимущество над заданными другим путём. Если программа будет компилироваться не вами, например, на олимпиаде, то задавайте настройки компилятора именно этим способом. По умолчанию (т.е. если их специально не задавать) обе описанные опции и в FreePascal, и в Delphi выключены. Поэтому в нашей программе их можно было и не задавать.

Теперь для увеличения скорости работы программы вообще избавимся от вычисления сложного логического выражения. Этого можно добиться так же, как мы делали в сортировке прямым включением. Используем барьерный элемент. То есть количество элементов массива увеличим на 1, а на место последнего элемента массива поставим k. В этом случае существует $i \in [1, N+1]$ такое, что условие `a[i]<>k` будет ложным. Если это случится при i=N+1, значит элемента, равного k, в исходном массиве (т.е. без добавленного элемента) нет. Получается следующая программа.

```
{$mode delphi}
Program SeqSearchSent;
Const MAXN = 1000;

Type TElem = Integer;
     TMyArray = array[1..MAXN+1] Of TElem;

Var a    : TMyArray;
    k    : TElem;
    i, N : Integer;
    f    : Text;
Begin
  Assign(f,'input.txt');
  Reset(f);
  Read(f,N,k);
  If N > MAXN Then
  Begin
    WriteLn('Количество элементов массива слишком велико: ', N ,'>',
MAXN);
    Halt
  End;

  For i:=1 To N Do
  Read(f,a[i]);

  Close(f);
  Assign(f,'output.txt');
  Rewrite(f);

  {$B-}
  (* Собственно поиск *)
  i:=1;
  a[N+1]:=k;
  While a[i]<>k Do
  i:=i+1;

  If i<=N Then
    Write(f,i)
  Else
    Write(f,'Not found');
  Close(f)
End.
```

Двоичный поиск

Пусть теперь массив, в котором осуществляется поиск, упорядочен. Для определённости будем считать его упорядоченным по возрастанию. Случай упорядоченного по убыванию массива рассматривается аналогично.

В этом случае время поиска можно значительно сократить. Вместо того чтобы на каждом шаге сокращать диапазон поиска на 1 элемент, можно сразу отбрасывать половину элементов непросмотренной части массива.

Будем сравнивать искомый элемент k с элементом, стоящим в середине непросмотренной части массива. Эту часть будем отмечать двумя переменными: l — индексом её левого конца и r — индексом правого конца. В начале поиска $l = 1$ и $r = N$. Индекс элемента в середине массива $m = (l + r)\ div\ 2$. Сравнивая $a[m]$ и k, мы получим один из трёх случаев:

- $k > a[m]$. Тогда индекс искомого элемента больше m, и можно уже не искать в части массива от его начала до m включительно. Полагаем $l = m + 1$.
- $k < a[m]$. Тогда индекс искомого элемента меньше m, и можно при следующем проходе цикла считать $r = m - 1$.
- $k = a[m]$. Элемент найден.

На каждом шаге разница между l и r уменьшается. Выполнение цикла закончится, если пока поиск увенчается успехом или l окажется больше r.

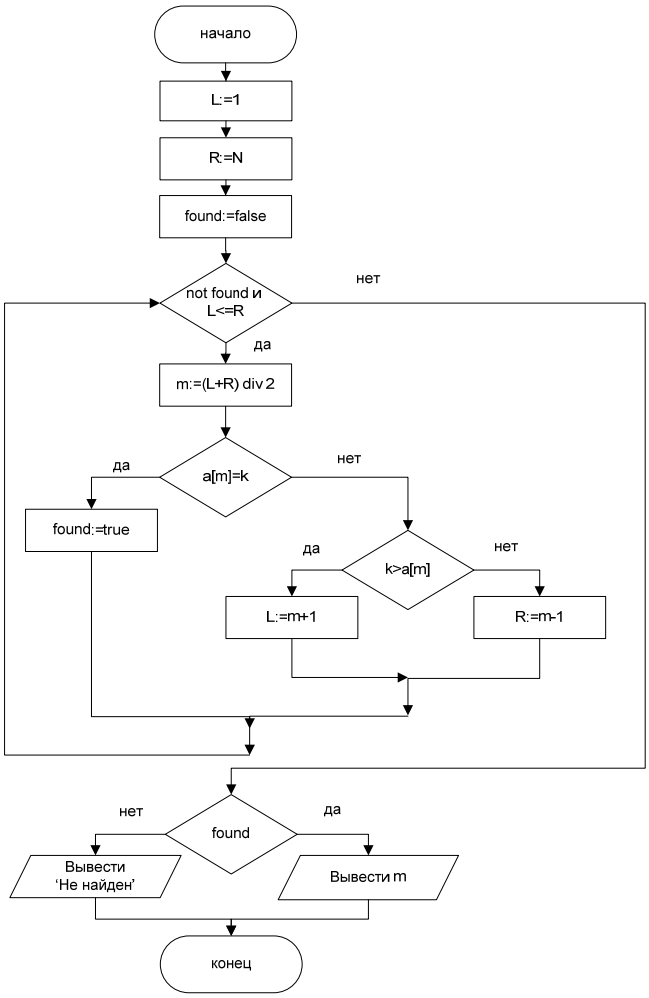

Рисунок 2. Двоичный поиск

И в этом алгоритме можно сократить количество сравнений. Не будем отдельно проверять случай $a[m] = k$. Объединим его со случаем $k < a[m]$. При $k \le a[m]$ будем считать $r = m$, а условие продолжения цикла заменим на $l < r$. Таким образом мы просто будем сужать диапазон поиска до одного элемента. После выхода из цикла окажется $l = r$. Если $a[r] = k$, то искомый элемент найден, иначе такого элемента в массиве просто нет.

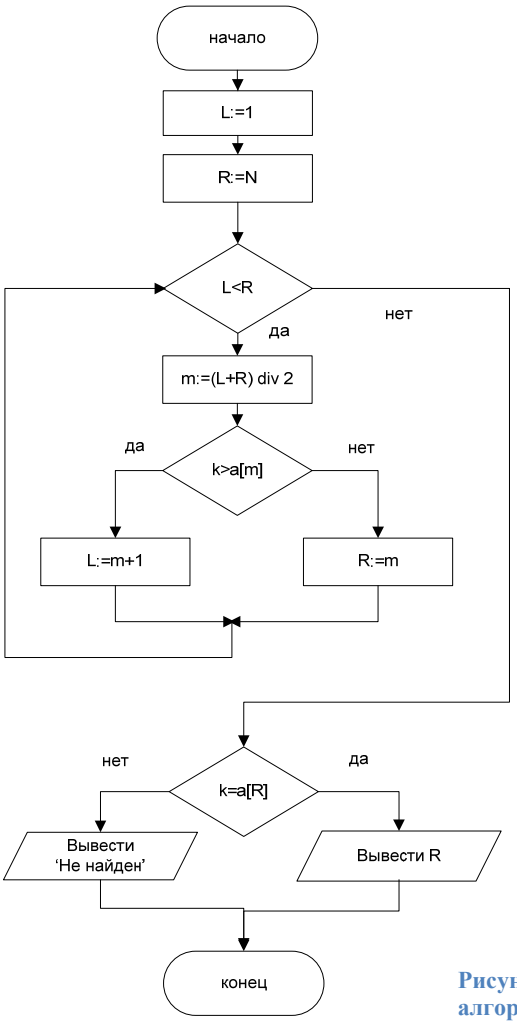

Рисунок 3. Двоичный поиск. Быстрый алгоритм

Проиллюстрируем на примере данный алгоритм. Пусть в массиве

10	20	30	40	50	60	70	80

требуется найти индекс элемента $k = 50$. Выполним цикл вручную.

Номер прохода цикла (итерации)	*Значения в начале цикла*		m
	L	R	
1	1	8	4
2	5	8	6
3	5	6	5
4	5	5	

Четвёртого прохода цикла (четвёртой итерации) не будет, т.к. условие $l < r$ не выполнено. Поиск в массиве из 8 элементов завершился за 3 итерации.

В каждой итерации этого алгоритма количество сравнений меньше, чем в алгоритме рис. 2. Но ввиду того, что отсутствует проверка $k = a[m]$, возможны лишние итерации, пока просматриваемая часть массива не стянется к одному элементу. Если элементов в массиве много, выигрыш в эффективности на каждом шаге превзойдёт потери от нескольких дополнительных сравнений[1].

Вот программа, реализующая этот алгоритм.

```
{$mode delphi}
Program BinSearch;
Const MAXN = 1000;
```

[1] См. [2] в списке литературы

```
Type TElem = Integer;
     TMyArray = array[1..MAXN] Of TElem;

Var a         : TMyArray;
    k         : TElem;
    i,l,r,m,N : Integer;
    f         : Text;
Begin
  Assign(f,'input.txt');
  Reset(f);
  Read(f,N,k);
  If N > MAXN Then
  Begin
    WriteLn('Количество элементов массива слишком велико: ', N ,'>',
MAXN);
    Halt
  End;

  For i:=1 To N Do
  Read(f,a[i]);

  Close(f);
  Assign(f,'output.txt');
  Rewrite(f);

  l:=1;
  r:=N;
  While l<r Do
  Begin
    m:=(l+r) div 2;
    If k>a[m]
    Then l:=m+1
    Else r:=m
  End;

  If k=a[r] Then
    Write(f,r)
  Else
    Write(f,'Not found');
  Close(f)
End.
```

Вопросы и задания

1. Двоичный поиск по алгоритму рис. 3 в массиве из 8 элементов завершился за 3 итерации. А сколько итераций потребуется для массива из 16 элементов? из 32? 1024? А в случае линейного поиска?
2. Пусть массив отсортирован по убыванию. Напишите программу двоичного поиска элемента этого массива. Предусмотрите в программе подсчёт количества итераций и вывод этого числа. Соответствует ли данный вами ответ на вопрос 1 полученному результату?
3. Пусть в отсортированном по возрастанию массиве существует несколько элементов, равных искомому значению k. Индекс какого из них мы получим, пользуясь ускоренным алгоритмом двоичного поиска (рис. 3)?
4. В работе [1] (см. список литературы) Д. Кнут описывает интерполяционный поиск, применимый к отсортированному массиву. Вот очень близко к тексту.

   Представьте, что вы ищете слово в словаре. Вероятно, вы не станете открывать словарь в середине и идти к странице, расположенной на $\frac{1}{4}$ или $\frac{3}{4}$ от начала, т.е. выполнять алгоритм бинарного поиска. Если слово начинается с буквы А, вероятно, вы ищете его в начале

книги. Если вы заметите, что искомое слово находится в конце группы слов на заданную букву, вы просто пропустите изрядное количество страниц. Эти рассуждения приводят нас к алгоритму, который может быть назван интерполяционным поиском: если мы знаем, что k находится между $a[l]$ и $a[r]$, следующую проверку мы будем делать примерно на расстоянии $\frac{k-a[l]}{a[r]-a[l]}(r-l)$ от l (в предположении, что элементы массива представляют собой числовые значения, близкие к арифметической прогрессии).

Попробуйте написать программу интерполяционного поиска.

Литература

1. Д. Кнут. Искусство программирования для ЭВМ. Том 3. Сортировка и поиск. Издание 3. — Издательский дом «Вильямс», 2005.
2. Н. Вирт. Алгоритмы и структуры данных. — СПб.: Невский диалект, 2008.