# Поиск

## Полный перебор

Полный перебор – один из самых простых алгоритмов поиска. Чтобы выполнить поиск методом полного перебора, необходимо начать с начала массива и перебирать элементы последовательно, пока не будет найден искомый элемент. Поскольку этот алгоритм исследует массив по порядку, он ищет элементы в начале массива быстрее, чем элементы, находящиеся в конце. Очевидно, что в лучшем случае метод найдёт элемент с первой же попытки за время $O(1)$, а в худшем случае придётся просмотреть весь массив за время $O(N)$.

Если элемент содержится в массиве, то алгоритм должен в среднем исследовать $\frac{N}{2}$ элементов до того, как обнаружит искомый. Таким образом, в среднем поиск осуществляется за время порядка $O(N)$. Хотя алгоритмы, которые выполняются за время порядка $O(N)$, нельзя назвать быстрыми, этот алгоритм достаточно прост и даёт неплохие результаты на практике. Для небольших массивов этот алгоритм показывает приемлемую производительность. Преимущество этого метода ещё и в том, что он не требует предварительной сортировки.

**Листинг 1**

```
function LinearSearch(target: LongWord): LongWord;
var
i: LongWord;
fl: boolean;
begin
  i:=1;
  fl:=false;
  while (i<=N) and (not fl) do
  begin
    if A[i]=target then
      fl:=true
    else
      i:=i+1;
  end;
  if not fl then
    i:=0;
  LinearSearch:=i;
end;
```

## Двоичный поиск

Бинарный поиск – первый метод поиска, который мы рассмотрим, применимый только для отсортированных массивов. Алгоритм бинарного поиска сравнивает элемент в середине исследуемой части массива с искомым элементом. Если искомый элемент меньше, алгоритм продолжает поиск в левой половине массива, если он больше, чем найденный элемент, поиск продолжается во в правой половине массива. Ниже приведён пример поиска элемента со значением $44$.

**Табл. 1**

| 1 | 4 | 7 | 9 | 9 | 12 | 14 | 17 | 19 | 21 | 24 | 32 | 36 | 44 | 45 | 54 | 55 | 63 | 66 | 70 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|

| | | | | | | | | | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 1 | 4 | 7 | 9 | 9 | 12 | 14 | 17 | 19 | 21 | 24 | 32 | 36 | 44 | 45 | 54 | 55 | 63 | 66 | 70 |
| 1 | 4 | 7 | 9 | 9 | 12 | 14 | 17 | 19 | 21 | 24 | 32 | 36 | 44 | 45 | 54 | 55 | 63 | 66 | 70 |
| 1 | 4 | 7 | 9 | 9 | 12 | 14 | 17 | 19 | 21 | 24 | 32 | 36 | 44 | 45 | 54 | 55 | 63 | 66 | 70 |

Хотя этот алгоритм естественно рекурсивен, его довольно просто записать без использования рекурсии. Идея, положенная в основу этого алгоритма, проста, но детали её реализации достаточно сложны. Программа должна аккуратно отслеживать часть массива, которая может содержать искомый элемент. В противном случае искомый элемент может быть пропущен.

Для отслеживания минимального и максимального индекса записей части массива, которая может содержать искомый элемент, алгоритм использует две переменных – $min$ и $max$. Во время выполнения алгоритма индекс искомого элемента всегда будет находиться между значениями $min$ и $max$.

Во время каждого прохода алгоритма переопределяется значение переменной $middle = round(\frac{min+max}{2})$, и проверяется элемент массива с индексом $middle$. Если её значение равно искомому, то цель найдена, и алгоритм завершает свою работу.

Если искомый элемент меньше, то алгоритм устанавливает $max = middle - 1$ и продолжает поиск. Поскольку индексы элементов, которые могу содержать искомый элемент, находятся теперь в диапазоне от $min$ до $middle - 1$, программа продолжит исследовать левую половину массива.

Если искомый элемент больше, чем элемент с индексом $middle$, программа устанавливает значение $min = middle + 1$ и продолжает поиск. Диапазон индексов элементов, которые могут содержать искомый элемент, лежит теперь в пределах от $middle + 1$ до $max$, поэтому программа продолжает исследование правой половины массива.

В конце концов программа либо найдёт искомый элемент, либо наступит момент, когда значение переменной $min$ будет больше, чем значение переменной $max$. Значения $min$ и $max$ постоянно корректируются таким образом, чтобы индекс искомого элемента находился всегда между ними. Поскольку в данной точке больше нет индексов между $min$ и $max$, искомого элемента в массиве тоже нет.

Следующий код демонстрирует выполнение двоичного поиска[1]:

**Листинг 2**

```
function BinarySearch(target: LongWord): LongWord;
var
min,max,middle: LongWord;
fl: boolean;
begin
  fl:=false;
  min:=1; max:=N;
  while (min<=max) and (not fl) do
  begin
    middle:=round((max+min)/2);
```

[1] Недостаток данной реализации в том, что программа может выдать не первое появление элемента в массиве. Чтобы исправить этот недостаток необходимо выполнить следующий код перед тем, как вернуть результат:

```
while (middle>1) and (A[middle]=A[middle-1]) do
  middle:=middle-1;
```

```
    if A[middle]=target then
      fl:=true
    else
      if target<A[middle] then
        max:=middle-1
      else
        min:=middle+1;
  end;
  if not fl then
    middle:=0;
  BinarySearch:=middle;
end;
```

На каждом шаге алгоритм сокращает число элементов, которые всё ещё могут содержать искомый элемент, в два раза. Для массива размера $N$ алгоритму потребуется максимум $O(\log_2 N)$ шагов, чтобы найти любой элемент или определить, что его нет в массиве. При этом двоичный поиск работает намного быстрее, чем полный перебор: полный перебор из миллиона элементов занимает в среднем $500000$ шагов, алгоритму двоичного поиска потребуется максимум $\log_2 1000000 = 20$ шагов.

# Интерполяционный поиск

Двоичный поиск оптимизирует поиск полным перебором, так как исключает большие части массива, не проверяя значения пропускаемых элементов. Если известно, что значения распределены достаточно равномерно, то можно на каждом шаге исключить ещё большее количество элементов, используя интерполяционный поиск.

Интерполяция – это процесс предсказания неизвестных значений на основе имеющихся. В данном случае будут использоваться индексы известных значений в массиве, чтобы определить какой индекс должно иметь искомое значение. Предположим, что задан следующий массив:

**Табл. 2**

| 1 | 4 | 7 | 9 | 9 | 12 | 14 | 17 | 19 | 21 | 24 | 32 | 36 | 44 | 45 | 54 | 55 | 63 | 66 | 70 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|

Этот массив содержит $20$ элементов со значениями от $1$ до $70$. Допустим, мы хотим найти в этом массиве элемент со значением $44$. Значение $44$ составляет $64\%$ размера диапазона от $1$ до $70$. Если считать, что значения элементов распределены равномерно, то искомый элемент, предположительно, будет находится в позиции, составляющей $64\%$ от размера массива – то есть в позиции с индексом $13$.

Если найденная алгоритмом позиция неверна, то он сравнивает искомое значение со значением в выбранной позиции. Если искомое значение меньше, алгоритм продолжает искать в левой части массива, если больше, то поиск продолжается в правой части массива.

Двоичный поиск разбивает массив пополам. Интерполяционный поиск делит массив, пытаясь найти ближайший к искомому элемент в массиве, при этом точка разбиения определяется следующей формулой: $middle = min + round(\frac{(target - A[min]) * (max - min)}{A[max] - A[min]})$.

Эта операция помещает значение элемента $middle$ между $min$ и $max$, что соответствует положению искомого элемента между $A[min]$ и $A[max]$.

Если искомый элемент близок к $A[min]$, то разность $target - A[min]$ близка к $0$. Тогда значение $middle$ почти равно $min$. Можно ожидать, что индекс элемента будет близок к $min$, если его значение почти равно $A[min]$. Аналогично, если искомый элемент находится рядом с $A[max]$, то и его индекс практически равен $max$.

Когда программа вычислит значение $middle$, она сравнивает значение элемента в этой позиции с искомым так же, как и при двоичном поиске. Если элемент равен искомому, то программа завершает работу. Если искомый элемент меньше, то программа устанавливает $max = middle - 1$ и продолжает искать в массиве меньшие элементы. Если искомый элемент больше найденного, программа устанавливает $min = middle + 1$ и продолжает искать в массиве большие элементы.

Стоит обратить внимание на следующий нюанс. Соотношение, которое вычисляет очередное значение для $middle$, делится на $A[max] - A[min]$. Если $A[min] = A[max]$, то происходит попытка деления на $0$, и программа аварийно завершит работу. Это может случиться, если в массиве имеется два идентичных значения. Тогда может возникнуть ситуация, когда значение $min$ возросло, а $max$ уменьшилось до уровня $min = max$.

Чтобы справиться с этой проблемой, программа перед делением проверяет условие $A[min] = A[max]$. Если условие выполняется, значит осталось проверить только одно значение. Программа проверяет его на равенство искомому.

Ещё одна тонкость заключается в том, что вычисленное значение $middle$ не всегда находится между $min$ и $max$. Самый простой случай, когда это происходит, - это когда искомый элемент лежит за пределами диапазона значений массива. Для примера попробуем найти элемент $300$ в массиве $\{100; 150; 200\}$. Первый раз, когда программа вычисляет средний элемент, $min = 1, max = 3$. Тогда $middle = 1 + \frac{(300 - A[1]) * (3-1)}{A[3] - A[1]} = 1 + \frac{(300-100)*2}{200-100} = 5$. Индекс $5$ находится за пределами границ массива. Если программа попробует обратиться к элементу $A[5]$, то она аварийно завершит работу с сообщением об ошибке.

Подобная проблема может возникать и в других случаях. Например, если значения между $min$ и $max$ распределены крайне неравномерно. Предположим, что надо найти значение $100$ в массиве $\{0; 1; 2; 199; 200\}$. При первом вычислении значения $middle$ мы получаем $1 + \frac{(100-0)*(5-1)}{200-0} = 3$. Затем мы сравниваем $A[3]$ с искомым элементом $100$. Так как $A[3] = 2$ меньше, чем $100$, программа устанавливает $min = middle + 1 = 4$. Затем программа вычисляет новое значение $middle = 4 + \frac{(100-199)*(5-4)}{200-199} = -98$. Значение $-98$ лежит за пределами массива.

Если рассмотреть вычисление среднего значения, то можно увидеть два варианта, при которых новое значение может быть меньше $min$ или больше $max$.

Предположим, что $middle$ меньше $min$:

$$min + \frac{(target - A[min]) * (max - min)}{A[max] - A[min]} < min$$

Вычтем $min$ из обеих частей:

$$\frac{(target - A[min]) * (max - min)}{A[max] - A[min]} < 0$$

Поскольку $max \geq min$, разность $max - min \geq 0$, $A[max] \geq A[min]$ разность $A[max] - A[min] \geq 0$. Тогда единственный вариант, при котором всё значение может быть меньше $0$, – если разность $target - A[min] < 0$.

Проводя аналогичные рассуждения для случая $target > max$, замечаем, что $middle$ может выйти за пределы массива только в случае, когда $target > A[max]$.

Объединяя эти результаты, получим, что единственная ситуация, при которой значение $middle$ может быть вне диапазона $[min; \max]$, возникает, когда значение $target$ лежит вне диапазона $[A[min]; A[max]]$.

В алгоритме надо предусмотреть этот факт и использовать специальную проверку перед каждым вычислением нового значения $middle$. Перед вычислением необходимо проверить условие $A[min] \leq target \leq A[max]$. Если оно ложное, то искомого элемента нет в массиве, и работу алгоритма можно завершить.

Следующий код представляет вариант реализации интерполяционного поиска[2]:

**Листинг 3**

```
function InterpolationSearch(target: LongWord): LongWord;
var
min,max,middle: LongWord;
fl: boolean;
begin
  fl:=false;
  min:=1; max:=N;
  while (min<=max) and (not fl) do
  begin
    if A[min]=A[max] then
    begin
      fl:=true;
      if target<>A[min] then
        middle:=0;
    end
    else
    begin
        middle:=round(min+((target-A[min])*((max-min)/(A[max]-
A[min]))));
        if ((middle<min) or (middle>max)) then
        begin
          fl:=true;
          middle:=0;
        end
        else
        begin
            if target=A[middle] then
              fl:=true
            else
              if target<A[middle] then
                max:=middle-1
              else
```

[2] Интерполяционный поиск обладает тем же недостатком, что и бинарный поиск (может возвращать не первое вхождение искомого элемента в массив). Способ решения аналогичен.

```
                min:=middle+1;
        end;
    end;
  end;
  if not fl then
    middle:=0;
  InterpolationSearch:=middle;
end;
```

Интерполяционный поиск работает ещё быстрее, чем двоичный поиск. Эта разница становится более заметной, если данные хранятся не в оперативной памяти, а на жёстком диске или любом другом медленном устройстве. Хотя интерполяционный поиск требует гораздо больше времени на вычисления, за счёт меньшего количества обращений к медленной памяти он оказывается эффективнее.

# Сравнение алгоритмов поиска

Некоторая информация, позволяющая выбрать наиболее подходящий для конкретной ситуации алгоритм, приведена в Табл. 3:

**Табл. 3**

| Метод | Преимущества | Недостатки | Область применения |
|---|---|---|---|
| **Полный перебор** | Прост в реализации<br>Простота обработки строковых данных | Низкая скорость работы | Поиск в небольших или несортированных массивах |
| **Двоичный поиск** | Довольно высокая скорость поиска<br>Простота обработки строковых данных | Относительная сложность реализации | Поиск в больших отсортированных массивах, хранящихся в памяти |
| **Интерполяционный поиск** | Очень высокая скорость поиска<br>Высокая скорость поиска на медленных устройствах | Сложность реализации<br>Требуется равномерность распределения данных | Поиск в больших отсортированных массивах, хранящихся в памяти или на внешних устройствах |