Тема 1.1

Массивы.
Поиск элемента в массиве

Учитывая текущее плачевное состояние наших программ, можно сказать, что программирование определенно все ещё черная магия и, пока, мы не можем называть его технической дисциплиной.

Bill Clinton

Содержание

1. Массивы
2. Строки
3. Записи и таблицы
4. Множества
5. Динамические массивы
6. Постановка задачи поиска элемента в массиве
7. Алгоритмы поиска:
   - последовательный поиск;
   - двоичный поиск

Рассмотрим *статические структуры данных:*

- массивы,
- записи,
- множества.

Цель описания типа данных и определения некоторых переменных, относящихся к статическим типам, состоит в том, чтобы зафиксировать

- ✔ диапазон значений, присваиваемых этим переменным,
- ✔ и соответственно размер выделяемой для них памяти.

Поэтому такие переменные и называются статическими.

1. Массивы

Понятие массива

Массив – это поименованная совокупность однотипных элементов, упорядоченных по индексам, определяющих положение элемента в массиве.

Следующее объявление задает

- ✔ имя для массива,
- ✔ тип для индекса
- ✔ и тип элементов массива:

```
имя: array[ТипИндекса] of ТипЭлемента;
```

Тип индекса, в общем случае, может быть любым порядковым. Но некоторые языки программирования поддерживают в качестве индексов массивов только последовательности целых чисел.

Понятие массива

Количество используемых индексов определяет размерность массива.

Массив может быть

- ✔ одномерным (вектор),
- ✔ двумерным (матрица),
- ✔ трехмерным (куб) и т. д.:

```
var
   Vector: array [1..100] of integer;
   Matrix: array [1..100, 1..100] of integer;
   Cube:  array   [1..100, 1..100, 1..100]
          of integer;
```

Операции над массивами

В Паскале определены такие операции над массивами в целом,

- как сравнение на равенство и неравенство массивов,
- а также операция присвоения для массивов с одинаковым типом индексов и одинаковым типом элементов.
- Доступ к массивам в этих операциях осуществляется через имя массива без указания индексов.

В некоторых языках программирования определен более мощный перечень операции, где в качестве операндов выступают целые массивы, это так называемые векторные вычисления.

Операции над массивами

Можно также выполнять операции над отдельными элементами массива.

- Перечень таких операций определяется типом элементов.
- Доступ к отдельным элементам массива осуществляется через имя массива и индекс (индексы) элемента:

```
Cube[0,0,10] := 25;
Matrix[10,30] := Cube[0,0,10] + 1;
```

В памяти ЭВМ элементы массива обычно располагаются непрерывно, в соседних ячейках.

Размер памяти, занимаемой массивом, есть суммарный размер элементов массива.

2. Строки

Строка – это последовательность символов (элементов символьного типа).

В Паскале количество символов в строке (длина строки) может динамически меняться от 0 до 255.

Рассмотрим пример описания строк:

```
Var
 TTxt: string;
 TWrd: string[10];
```

Здесь описаны

- ✔ строка ***TTxt***, максимальная длина которой 255 символов (по умолчанию)
- ✔ и строка ***TWrd***, максимальная длина которой ограничена 10 символами.

Каждый символ строки имеет свой индекс, принимающий значение от 1 до заданной длины строки.

Следует обратить внимание, что существует элемент строки с индексом 0:

- ✔ Он содержит текущее количество символов в строке.

Благодаря индексам, строки очень похожи на одномерные массивы символов:

- доступ к отдельным элементам строки можно получать с использованием этих индексов, выполняя операции, определенные для символьного типа данных;
- так же как и для массивов, определена операция присвоения строк в целом.

Однако есть ряд отличий:

- операций сравнения строк больше, чем аналогичных операций для массивов:

  <, >, >, <, =, <>;

- существует операция сцепления (конкатенации) строк «+».

В памяти ЭВМ символы строки располагаются непрерывно, в соседних ячейках.

Размер памяти, занимаемой строкой, есть суммарный размер элементов массива, включая элемент, содержащий длину строки.

3. Записи и таблицы

Записи

Запись – это агрегат, составляющие которого (поля) имеют имя и могут быть различного типа.

Рассмотрим пример простейшей записи:

```
Type
 TPerson = record
    Name:    string;
    Address: string;
    Index:  longint;
 end;
var    Person1: Tperson;
```

Запись описанного типа объединяет три поля:

- ✔ первые два из них символьного типа,
- ✔ а третье – целочисленного.

Записи

В Паскале определена операция присваивания для записей в целом (записи должны быть одного типа).

Доступ к записи осуществляется через ее имя.

Можно выполнять операции над отдельным полем записи. Перечень таких операций определяется типом поля.

Доступ к полям отдельной записи осуществляется через имя записи и имя поля:

```
Person1.Index := 190000;
Person1.Name := 'Иванов';
Person1.Adress := 'Новополоцк, ул. Кирова,
  д.1';
```

В памяти ЭВМ поля записи обычно располагаются непрерывно, в соседних ячейках.

Размер памяти, занимаемой записью, есть суммарный размер полей, составляющих запись.

Таблицы

Таблица представляет собой одномерный массив (вектор), элементами которого являются записи.

Отдельная запись массива называется ***строкой*** таблицы.

Чаще всего используется простая запись,

- ✔ т. е. поля которой являются элементарными данными.

Совокупность одноименных полей всех строк называется ***столбцом***, а конкретное поле отдельной строки – ***ячейкой***:

```
Type
  TPerson = record
      Name:    string;
      Address: string;
      Index:  longint;
  end;
  TTable = array[1..1000] of TPerson;
var   PersonList: TTable;
```

Таблицы

Характерной особенностью таблиц является то, что доступ к элементам таблицы производится не по индексу, а по ключу,

- ✔ т. е. по значению одного из полей записи.

Ключ таблицы (основной, первичный) – поле, значение которого может быть использовано для однозначной идентификации каждой записи таблицы.

- Ключ таблицы может быть составным – образовываться не одним, а несколькими полями данной таблицы.

Вторичный ключ – поле таблицы с несколькими ключами, не обеспечивающий (в отличие от первичного ключа) однозначной идентификации записей таблицы.

- В этот ключ могут входить все поля таблицы за исключением полей, составляющих первичный ключ.

Таблицы

Если последовательность записей упорядочена относительно какого-либо столбца (поля), то такая ***таблица*** называется ***упорядоченной***, иначе – ***таблица неупорядоченная***.

Основной операцией при работе с таблицами является операция доступа к записи по ключу. Она реализуется процедурой поиска.

- Получив доступ к конкретной записи (строке таблицы), с ней можно работать
  - ✔ как с записью в целом,
  - ✔ так и с отдельными полями (ячейками).

Таблицы

Перечень операций над отдельной ячейкой определяется типом ячейки:

```
PersonList[1].Index := 190000;
PersonList[1].Name := 'Иванов';
PersonList[1].Adress := 'Новополоцк,
      ул. Кирова, д.1';
```

В памяти ЭВМ ячейки таблицы обычно располагаются построчно, непрерывно, в соседних ячейках.

Размер памяти, занимаемой таблицей, есть суммарный размер ячеек.

4. Множества

Наряду с массивами и записями существует еще один структурированный тип – множество. Этот тип используется не так часто, хотя его применение в некоторых случаях является вполне оправданным.

Множество – совокупность каких-либо однородных элементов, объединенных общим признаком и представляемых как единое целое.

Тип множество соответствует математическому понятию множества в смысле операций, которые допускаются над структурами такого типа.

Множество допускает операции:

- объединения множеств «+»,
- пересечения множеств «*»,
- разности множеств «-»
- и проверки элемента на принадлежность к множеству «***in***».

Кроме того, определены операции сравнения множеств:

>, <, =, <>.

Множества, так же как и массивы, объединяют однотипные элементы. Поэтому в описании множества обязательно должен быть указан тип его элементов:

```
var
   RGB, YIQ, CMY: set of char;
```

Здесь приведено описание трех множеств, элементами которых являются символы.

В отличие от массивов и записей в множестве отсутствует возможность обращения к отдельным элементам.

- Операции выполняются по отношению ко всей совокупности элементов множества:

```
CMY := ['M', 'C', 'Y'];
RGB := ['R', 'G', 'B'];
YIQ := ['Y', 'Q', 'I'];
Writeln('пересечение цветовых систем
      RGB и CMY ', RGB*CMY);
Writeln('пересечение цветовых систем
      YIQ и CMY ', YIQ*CMY);
```

В Паскале в качестве типов элементов множества могут использоваться типы, максимальное количество значений которых не превышает 256.

В памяти ЭВМ элементы множества обычно располагаются непрерывно, в соседних ячейках.

5. Динамические массивы

При работе с массивами практически всегда возникает задача *настройки программы на фактическое количество элементов массива*.

В зависимости от применяемых средств решение этой задачи бывает различным.

Первый вариант – использование констант для задания размерности массива.

```
Program first;
const n = 10;
var a : array [1..n] of real;
    i : integer;
begin
  for i:=1 to n do
  readln (a[i]);
{ и далее все циклы работы с массивом используют n}
```

Такой способ требует перекомпиляции программы при каждом изменении числа обрабатываемых элементов.

Второй вариант – программист планирует некоторое условно максимальное (теоретическое) количество элементов, которое и используется при объявлении массива.

При выполнении программа запрашивает у пользователя фактическое количество элементов массива, которое должно быть не более теоретического.

На это значение и настраиваются все циклы работы с массивом.

```
Program second;
var a : array [1..25] of real;
    i, nf : integer;
begin
   writeln('введите фактич. число элементов
      массива <= 25 ');
   readln(nf);
   for i:=1 to nf do readln(a[i]);
{ и далее все циклы работы с массивом используют nf}
```

Этот вариант более гибок и технологичен по сравнению с предыдущим, т.к. не требуется постоянная перекомпиляция программы, но очень нерационально расходуется память, ведь ее объем для массива всегда выделяется по указанному максимуму. Используется же только часть ее.

Вариант третий – в нужный момент времени надо выделить динамическую память в требуемом объеме, а после того, как она станет не нужна, освободить ее.

```
Program dynam_memory;
type mas = array[1..2] of
<требуемый_тип_элемента>;
ms = ^mas;
var a : ms;
   i, nf : integer;
begin
   writeln('введите фактич. число элементов
        массива');
   readln(nf);
   getmem (a, sizeof(<требуемый_тип_элемента>)
         *nf);
   for i:=1 to nf do readln(a^[i]);
{ и далее все циклы работы с массивом используют nf}
...
   freemem(a);
end.
```

Где нужны динамические массивы?

Задача. Ввести размер массива, затем – элементы массива. Отсортировать массив и вывести на экран.

Проблема:

размер массива заранее неизвестен.

Пути решения:

1) выделить память «с запасом»;
2) выделять память тогда, когда размер стал известен.

Алгоритм:

3) ввести размер массива;
4) выделить память ;
5) ввести элементы массива;
6) отсортировать и вывести на экран;
7) удалить массив.

Использование указателей

какой-то массив целых чисел

```
program qq;
type intArray = array[1..1] of integer;
var A: ^intArray;
    i, N: integer;
begin
  writeln('Размер массива>');
  readln(N);
  GetMem(A, N*sizeof(integer));
  for i := 1 to N do
    readln(A^[i]);
  ... { сортировка }
  for i := 1 to N do
    writeln(A^[i]);
  FreeMem(A);
end.
```

выделить память

работаем так же, как с обычным массивом!

освободить память

Использование указателей

- для выделения памяти используют процедуру
  `GetMem( указатель, размер в байтах )`;
- указатель должен быть приведен к типу **`pointer`** – указатель без типа, просто адрес какого-то байта в памяти;
- с динамическим массивом можно работать так же, как и с обычным (статическим);
- для освобождения блока памяти нужно применить процедуру **`FreeMem ( указатель )`**;

6. Постановка задачи поиска элемента в массиве

Поиск в массиве

Одно из наиболее часто встречающихся в программировании действий – ***поиск данных***.

Существует несколько основных вариантов поиска, и для них создано много различных алгоритмов.

Постановка общей задачи поиска

Пусть $A = \{a_1, a_2, ...\}$ – последовательность однотипных элементов и ***b*** – некоторый элемент, обладающий свойством ***P***.

Найти место элемента ***b*** в последовательности ***A***.

Постановка общей задачи поиска

Поскольку представление последовательности в памяти может быть осуществлено в виде массива, задачи могут быть уточнены как одна из следующих задач поиска элемента в массиве *A*:

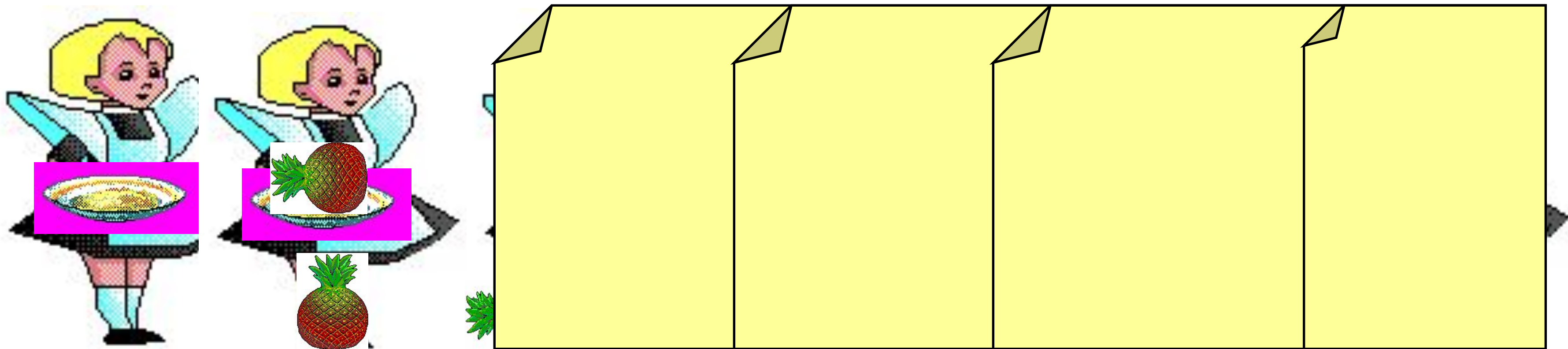
- найти максимальный (минимальный) элемент массива;
- найти заданный элемент массива;
- найти *k*–ый по величине элемент массива.

Наиболее простые и часто оптимальные алгоритмы основаны на последовательном просмотре массива ***A*** с проверкой свойства ***P*** на каждом элементе.

Максимальный элемент

Задача: найти в массиве максимальный элемент.

Алгоритм:



Псевдокод:

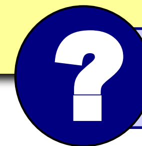
```
{ считаем, что первый элемент – максимальный }
for i:=2 to N do
  if a[i] > { максимального } then
    { запомнить новый максимальный элемент a[i] }
```

? Почему цикл от `i=2`?

Максимальный элемент

Дополнение: как найти номер максимального элемента?

```
                 { считаем, что первый – максимальный }
iMax := 1;
for i:=2 to N do       { проверяем все остальные }
  if a[i] > a[iMax] then { нашли новый максимальный }
  begin
                           { запомнить a[i] }
    iMax := i;             { запомнить i }
  end;
```



Как упростить?

По номеру элемента **iMax** всегда можно найти его значение **a[iMax]**. Поэтому везде меняем max на **a[iMax]** и убираем переменную **max**.

Максимальный элемент

```
program qq;
const N = 5;
var a: array [1..N] of integer;
    i, iMax: integer;
begin
  writeln('Исходный массив:');
  for i:=1 to N do begin
      a[i] := random(100) + 50;
      write(a[i]:4);
  end;
  iMax := 1; { считаем, что первый – максимальный }
  for i:=2 to N do      { проверяем все остальные }
    if a[i] > a[iMax] then { новый максимальный }
      iMax := i;           { запомнить i }
  writeln; {перейти на новую строку}
  writeln('Максимальный элемент a[', iMax, ']=', a[iMax]);
end.
```

случайные числа в интервале [50,150)

поиск максимального

Поиск заданного элемента в массиве

При дальнейшем рассмотрении делается принципиальное допущение:

- ✔ группа данных, в которой необходимо найти заданный элемент, фиксирована.

Будем считать, что множество из ***N*** элементов задано в виде такого массива

```
var a: array [1..N] of Item;
```

Обычно тип ***Item*** описывает запись с некоторым полем, играющим роль ключа.

Поиск заданного элемента в массиве

Задача заключается в поиске элемента, ключ которого равен заданному «аргументу поиска» ***x***.

Полученный в результате индекс ***i***, удовлетворяющий условию

a[i].key = x,

обеспечивает доступ к другим полям обнаруженного элемента.

Так как здесь рассматривается, прежде всего, сам процесс поиска, то мы будем считать (для простоты изложения материала), что

✔ тип ***Item*** включает только ключ.

Поиск заданного элемента в массиве

С точки зрения теории множеств поиск можно рассматривать, как

- ✔ отображение множества $X = \{x_1, x_2, ..., x_N\}$ на множество $X' = \{x_k\}$,
- ✔ X' является подмножеством X,
- ✔ и $x_k = x$, где x – аргумент поиска.

Если искомых данных в множестве X нет, то множество X' является пустым.

Поиск заданного элемента в массиве

Алгоритм поиска может

- ✔ возвратить элемент массива (или всю найденную запись массива)
- ✔ или чаще всего может возвратить некоторый указатель на этот элемент.

Поиск заданного элемента в массиве

Поиск, по завершении которого найден элемент массива с ключом, равным аргументу поиска, называется ***успешным*** или ***результативным***.

- Успешный поиск часто завершается ***извлечением*****.**

Поиск заданного элемента в массиве

Однако возможно, что поиск некоторого конкретного аргумента в массиве является ***неудачным*** (***безрезультатным***),

- ✔ т.е. в данном массиве не существует элемента с этим аргументом в качестве ключа.

- В этом случае такой алгоритм может возвратить
  - ✔ некоторый специальный «пустой элемент»
  - ✔ или некоторый пустой указатель.
- Однако чаще такое условие приводит
  - ✔ к появлению некоторой ошибки
  - ✔ или к установке во флаге некоторого конкретного значения, которое указывает, что искомый элемент отсутствует.

Поиск заданного элемента в массиве

Поиск требуемой информации применяется ко всем основным структурам данных с произвольным доступом:

- ✔массивам,
- ✔спискам (одно- и двусвязным),
- ✔деревьям,
- ✔графам,
- ✔таблицам.

Поиск заданного элемента в массиве

Задача – найти в массиве элемент, равный **x**, или установить, что его нет.

Решение: для произвольного массива: **линейный поиск** (перебор)

недостаток: **низкая скорость**

Как ускорить? – заранее подготовить массив для поиска

- как именно подготовить?
- как использовать «подготовленный массив»?

7. Алгоритмы поиска элемента в массиве

Линейный поиск

Нахождение информации в неотсортированной структуре данных, например в массиве, требует применения последовательного поиска.

Последовательный (или линейный) поиск – наиболее просто реализуемый метод поиска.

Последовательный поиск заключается в последовательном переборе элементов структуры данных (например, массива) от начального элемента до нахождения совпадения или до конца структуры данных.

Перебор элементов имеет линейный характер, поэтому такой поиск еще называют линейным.

Линейный поиск

nX – номер нужного элемента в массиве

```
nX := 0; { пока не нашли ...}
for i:=1 to N do { цикл по всем элементам }
  if A[i] = X then { если нашли, то ... }
    nX := i;      { ... запомнили номер}
if nX < 1 then writeln('Не нашли...')
else           writeln('A[', nX, ']=', X);
```

Улучшение: после того, как нашли **X**, выходим из цикла.

Что плохо?

```
nX := 0;
for i:=1 to N do
 if A[i] = X then begin
   nX := i;
   break; {выход из цикла}
 end;
```

```
nX := 0; i := 1;
while i <= N do begin
  if A[i] = X then begin
    nX := i; i := N;
  end;
  i := i + 1;
end;
```

Функции линейного поиска

Рассмотрим примеры последовательного поиска с циклами ***for*** и ***while***.

```
Type
  Item = record
     ... { здесь должно быть соответствующее описание
      типа элементов массива }
  end;
  Mas: array [1..N] of Item;
      { массив, в котором ищется элемент }
```

Функции линейного поиска

```
function search_s1(A: Mas; N: integer; key: Item):
                  integer;
var
  i: integer;
begin
  for i := 1 to N do
    if key = A[i] then begin
      search_s1 := i;
      break
    end
    else
      search_s1 := -1
end;
```

Функции линейного поиска

```
function search_s2(A: Mas; N: integer; key: Item):
                  integer;
var
  i: integer;
begin
  i:=0;
  search_s2 := -1;
  while (key <> A[i]) and ( i <= n) do begin
    if key = A[i] then
      search_s2 := i;
    i := i + 1
  end
end;
```

Линейный поиск

Последовательный поиск выполнит

- в среднем случае проверку N/2 элементов,
- в лучшем – 1 элемента,
- а в худшем – N элементов.

Таким образом, трудоемкость алгоритма выражается как *O(N)*.

Линейный поиск

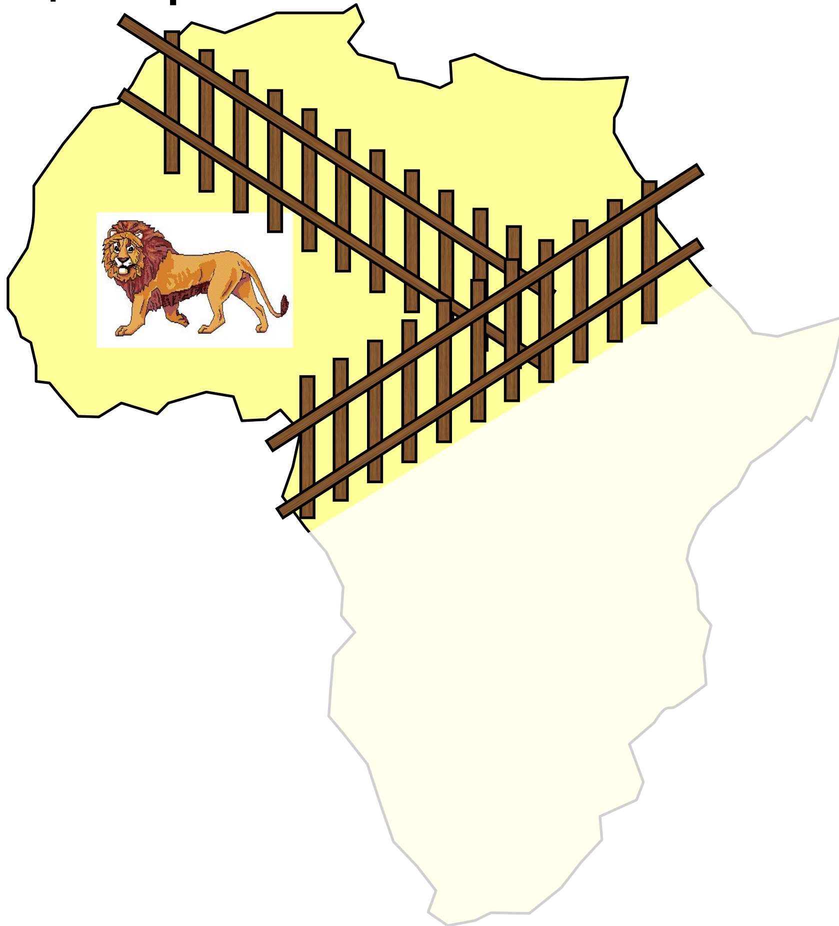
Недостаток этого поиска:

- медленное выполнение при большом объеме просматриваемого массива.

Но если данные не отсортированы, то должен использоваться только последовательный поиск.

Двоичный поиск

Двоичный (или **бинарный**) **поиск** основан на итерационном сравнении ключа поиска с центральным элементом текущей части массива.



1. Разделить область поиска на две равные части.
2. Определить, в какой половине находится нужный объект.
3. Перейти к шагу 1 для этой половины.
4. Повторять шаги 1-3 пока объект не будет «пойман».

Двоичный поиск

- При каждой итерации находится середина текущей анализируемой части,
  - ✔ т.е. интервал анализа делится пополам (на 2):

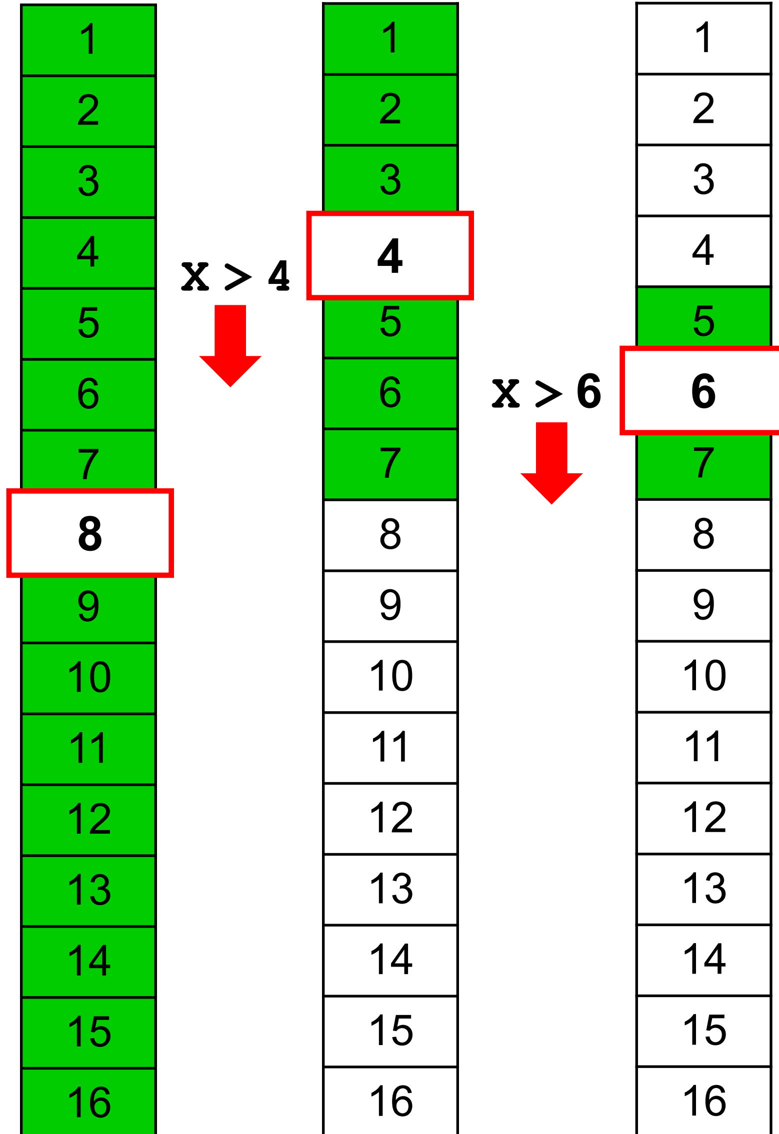
    1/2, 1/4, 1/8 и т.д.,
  - ✔ откуда этот метод поиска и получил свое название.
- При неудачном сравнении ключа поиска с текущими данными в зависимости от результата сравнения выбирается
  - ✔ нижний
  - ✔ или верхний полуинтервал.
- Процесс продолжается до тех пор, пока
  - ✔ не будет найдено совпадение
  - ✔ или длина интервала анализа не станет равной единице, и если при этом всё ещё нет совпадения, то фиксируется неудача поиска.

Двоичный поиск

Процесс поиска числа 7 в упорядоченном массиве целых чисел от 1 до 16:

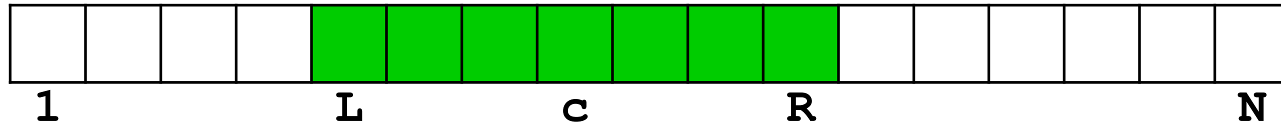
X = 7

X < 8



1. **Выбрать средний элемент `A[c]` и сравнить с X.**
2. **Если `X = A[c]`, нашли (выход).**
3. **Если `X < A[c]`, искать дальше в первой половине.**
4. **Если `X > A[c]`, искать дальше во второй половине.**

Двоичный поиск



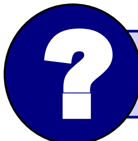

```
nX := 0;
L := 1; R := N; {границы: ищем от A[1] до A[N] }
while R >= L do begin
  c := (R + L) div 2;
  if X = A[c] then begin
    nX := c;
    R := L - 1; { break; }
  end;
  if X < A[c] then R := c - 1;
  if X > A[c] then L := c + 1;
end;
if nX < 1 then writeln('Не нашли...')
else           writeln('A[', nX, ']=', X);
```

номер среднего элемента

нашли

выйти из цикла

сдвигаем границы



Почему нельзя while R > L do begin … end; ?

Двоичный поиск

Этот метод поиска значительно эффективнее чем последовательный поиск, но требует, чтобы ***данные были предварительно упорядочены*** (отсортированы).

В худшем случае выполняется не более $\log_2 N$ сравнений, в связи с чем двоичный поиск ещё называется "логарифмическим поиском".

Трудоемкость его определяется как $O(\log_2 N)$.

Сравнение методов поиска

	Линейный	Двоичный
подготовка	нет	отсортировать
	число шагов	
N = 2	2	2
N = 16	16	5
N = 1024	1024	11
N= 1 048 576	1 048 576	21
N	$\leq N$	$\leq \log_2 N+1$