Модуль 4. Алгоритмы и структуры данных

Тема 4.1. Массив как базовая структура данных

4 часа

Оглавление

4.1. Массив как базовая структура данных

4.1.1. Основные сведения о массивах

Массив - именованная совокупность элементов одинакового типа. При этом тип, на основе которого строится массив, называется ***базовым.***

Напомним, что базовыми могут являться как примитивные типы (int, char, boolean и др.), так и ссылочные типы.

Переменную массива объявляем с помощью квадратных скобок:

```
int[] a; // объявление целочисленного массива
```

и затем выделяем память под необходимое количество элементов массива (создаем массив):

```
a = new int [10];  // выделение памяти под массив из 10 элементов
```

При необходимости можем совместить объявление массива с его созданием:

```
int [] a = new int [10];
```

При создании массива происходит инициализация начальных значений элементов массива либо по умолчанию, либо с помощью списка выражений:

```
int[] a = new int[] {1, 9, 34, -9, -2, 45, 90, 0, -11};
```

В этом случае размер создаваемого массива будет равен длине списка инициализации.

Размер массива хранится в свойстве length, которое можно использовать при обработке элементов массива:

```
int sum = 0;
for (int i = 0; i < a.length; i++) {
   sum += a[i];
}
System.out.println("Сумма элементов = " + sum);
```

В тех случаях, когда в цикле элементы массива не меняются, можно воспользоваться циклом for each, который копирует в переменную цикла x поочередно все элементы массива a:

```
int sum = 0;
for (int x : a)
```

```
        sum += x;
    System.out.println("Сумма элементов = " + sum);
```

Полезным является метод клонирования (создания точной копии) массива:

```
int[] b = a.clone();
```

Также создание точной копии массива может быть осуществлено через метод **System.arraycopy()**

```
    int[] a = {3, 12, 4, 15, 4, 7, 47};
    int[] b = new int[a.length];
    System.arraycopy(a, 0, b, 0, b.length);
    for (int i : b) {
       System.out.println(i);
    }
```

Одной из сложностей при программировании на C и C++ является то, что массив может быть возвращен только неявным образом по адресу, переданному в одном из параметров функции, при этом программисту необходимо управлять временем жизни массива, что может привести к потерям памяти.

В Java также возвращается ссылка на массив, но программисту нет необходимости нести ответственность за этот массив, поскольку массив будет существовать столько, сколько нужно, а сборщик мусора очистит массив, когда программа закончит работу с ним, или когда ни одна переменная в программе не будет ссылаться на этот массив.

Упражнение 4.1.1

Создать массив из 10 элементов. Получить массив-копию и массив, каждый элемент которого будет равен произведению элемента исходного массива и его индекса. Реализовать статический метод вывода массива на экран. Вывести все массивы на экран.

```
public class Program {

   // метод возвращает новый массив
   public static int[] change(int[] a) {
       int[] results = new int[a.length];
       for (int i = 0; i < a.length; ++i) {
           results[i] = a[i] * i;
       }
       return results;
   }

   // метод вывода массива на экран
   public static void printArr(int[] a) {
       for (int x : a) {
           System.out.print("  " + x);
       }
       System.out.println();
   }

   public static void main(String[] args) {
       int[] a = new int[10]; //
       for (int i = 0; i < a.length; ++i) {
           a[i] = i;
       }
```

```
        printArr(a);
        int[] b = a.clone();
        printArr(b);
        int[] c = change(a);
        printArr(c);
    }
}
```

4.1.2 Библиотечный класс Arrays

В классе **Arrays** из пакета **java.util** собрано множество методов для работы с массивами. Их можно разделить на четыре группы:

- заполнение;
- сортировка;
- поиск;
- сравнение.

Отметим, что все эти методы являются статическими, а, значит, должны быть вызваны по имени класса Arrays. Массив, к которому применяют эти методы, является одним из параметров указанных методов и должен быть описан как type_[], где type_ может быть одним из примитивных типов byte, short, int, long, char, float, double, или тип Object, что, в свою очередь, означает возможность использования любого типа, являющегося наследником типа Object.

Заполнение массива

Первая группа методов - **fill ()** - заполняет указанным значением **value:**

а) массив целиком:

`static void fill(type_[] a, type_ value),`

Это метод удобно использовать, когда необходимо задать значение по умолчанию для всех элементов массива:

```
import java.util.Arrays;
...

int[] a = new int[10];
Arrays.fill(a, 0);          // обнуление всех элементов массива
```

б) часть массива от индекса **from** включительно до индекса **to** исключительно:

`static void fill(type_[] a, int from, int to, type_ value),`

где type_ может быть одним из примитивных типов byte, short, int, long, char, float, double или тип Object.

```
// заполнение единицами первой половины массива
Arrays.fill(a, 0, a.length / 2, 1);
```

Сортировка массива

Вторая группа методов - **sort ( )** - сортирует:

а) массив в порядке возрастания числовых значений или объекты в их естественном порядке:

```
Arrays.sort(b);
```

б) в порядке возрастания часть массива от индекса **from** включительно до индекса **to** исключительно:

```
Arrays.sort(b, a.length / 2, a.length);
```

в) массив или его часть с элементами типа **Object** по правилу, заданному объектом **c**, реализующим интерфейс **Comparator:**

```
static void sort(Object[] a, Comparator c)
```

Интерфейс **Comparator** *позволяет определить собственные методы сравнения:* ***compare****(Object obj1, Object obj2) и проверки на эквивалентность* ***equals****(Object obj).*

int ***compare****(Object obj1, object obj2) — возвращает:*

- отрицательное число, если obj1 в каком-то смысле меньше obj2;

- 0, если они считаются равными;

- положительное число, если obj1 больше obj2.

boolean ***equals*** *(Object obj) — сравнивает данный объект с объектом obj, возвращая true, если объекты совпадают в каком-либо смысле, заданном этим методом.*

или

```
static void sort(Object[] a, int from, int to, Comparator c).
```

Стоит заметить, что параметром метода сортировки с использованием компаратора должен быть наследник класса Object, а не примитивный тип.

Упражнение 4.1.2

Создать массив из 10 элементов. Отсортировать массив по возрастанию значений младшей цифры элементов массива.

```
import java.util.Arrays;
import java.util.Comparator;

public class Program {

   public static void printArrI(Integer[] a) {
       for (int x : a) {
           System.out.print("  " + x);
       }
       System.out.println();
   }
```

```
    public static void main(String[] args) {
        int maxRange = 100;

        System.out.println("Сортировка массива через компаратор");

        Integer[] aInt = new Integer[10];
        for (int i = 0; i < aInt.length; ++i) {
            aInt[i] = (int) (Math.random() * (maxRange + 1));
        }
        printArrI(aInt);


        class Comp implements Comparator<Integer> {

            @Override
            public int compare(Integer obj1, Integer obj2) {
                int m1 = obj1 % 10;
                int m2 = obj2 % 10;

                if (m1 < m2)
                    return -1;
                else if (m1 > m2)
                    return 1;
                else
                    return 0;
            }
        }
        Comp c = new Comp();
        Arrays.sort(aInt, c);
        printArrI(aInt);
    }
}
```

На экране появится (для массива случайных чисел):

```
Сортировка массива через компаратор
  45  45  26  7  61  98  19  98  22  84
  61  22  84  45  45  26  7  98  98  19
```

Изменяя условия в компараторе, мы можем менять порядок сортировки массива (например, поменяв, местами команды return -1 и return 1, мы изменим сортировку по возрастанию на сортировку по убыванию значений младшей цифры элементов массива).

Поиск в массиве

Третья группа методов - **binarySearch()** - организует бинарный поиск, при котором метод возвращает индекс найденного элемента массива. Подробно бинарный поиск рассматривается далее.

Если элемент не найден, то возвращается отрицательное число, означающее индекс, с которым элемент *был бы вставлен* в массив в заданном порядке, с обратным знаком.

Класс Arrays поддерживает:

а) поиск элемента **element** в упорядоченном по возрастанию массиве:

```
static int binarySearch(type_[] a, type_ element)
```

```
int[] a = {3, 5, 8, 9, 11};
int k = Arrays.binarySearch(a, 8);
System.out.println(" " + k);
```

В этом примере программа вернет число 2.

б) поиск элемента **element** в массиве, отсортированном в порядке, заданном объектом-компаратором c.

static int binarySearch(Object[] a, Object element, Comparator c)

```
Integer[] aInt = new Integer[10];
class Comp implements Comparator<Integer> {

   @Override
   public int compare(Integer obj1, Integer obj2) {
       int m1 = obj1 % 10;
       int m2 = obj2 % 10;

       if (m1 < m2)
           return -1;
       else if (m1 > m2)
           return 1;
       else return 0;
   }
}
Comp c = new Comp();
Integer key = 2;
int ind = Arrays.binarySearch(aInt, key, c);
System.out.println(" " + ind);
```

Сравнение массивов

Четвертая группа методов - **equals ()** - сравнивает массивы:

static boolean equals (type_[] al, type_[] a2)

Массивы считаются равными, если они имеют одинаковую длину и равные элементы массивов с одинаковыми индексами. В этом случае метод возвращает true.

```
Arrays.equals(aInt, bInt);
```

Если элементы класса не относятся ни к одному из примитивных типов, то в классе необходимо реализовать метод equals(), который будет сравнивать отдельные элементы массивов на равенство.

Пример программы, демонстрирующей методы класса Arrays:

```
public class Program {
   public static void printArr(int[] a) {
       for (int x : a) {
           System.out.print("  " + x);
       }
       System.out.println();
   }

   public static void main(String[] args) {
       int maxRange = 100;
       int[] a = new int[10];
```

```
        for (int i = 0; i < a.length; ++i) {
            a[i] = (int) (Math.random() * (maxRange + 1));
        }
        printArr(a);

        int[] b = a.clone();
        if (Arrays.equals(a, b))
            System.out.println("Массивы равны");
        else
            System.out.println("Массивы не равны");

        System.out.println("Массив b после сортировки: ");
        Arrays.sort(b);
        printArr(b);

        if (Arrays.equals(a, b))
            System.out.println("Массивы равны");
        else
            System.out.println("Массивы не равны");

        int k = Arrays.binarySearch(b, a[0]);
        System.out.println("Элемент " + a[0] + " b имеет индекс " + k);
        System.out.println("\nЗаполнение части массива b: ");
        Arrays.fill(b, 0, b.length / 2, 0);
        printArr(b);
        //поиск в массиве произвольного числа 35
        k = Arrays.binarySearch(b, 35);
        System.out.println("Элемент " + 35 + " имеет индекс " + k);
    }
}
```

4.1.3 Библиотечный класс ArrayList

Напомним, что обычный массив имеет фиксированную длину. После того, как он создан, изменение его свойства length невозможно (т.к. это свойство объявлено как **final**).

В отличие от Arrays класс java.util.ArrayList из пакета java.util является автоматически расширяемым массивом. При создании объекта типа ArrayList необязательно указывать его размерность. При работе с таким массивом используются специальные методы.

ArrayList (http://habrahabr.ru/post/128269/) - это список, реализованный на основе массива. Отсюда и преимущества ArrayList: в возможности доступа к произвольному элементу по индексу за постоянное время (так как это массив), минимум накладных расходов при хранении такого списка, вставка в конец списка в среднем производится так же за постоянное время.

*В среднем, потому что массив имеет определенный начальный размер n (в коде это параметр capacity), по умолчанию n = 10, при записи n+1 элемента, будет создан новый массив размером (n * 3) / 2 + 1, в него будут помещены все элементы из старого массива + новый, добавляемый элемент. В итоге получаем, что при добавлении элемента при необходимости расширения массива, время добавления будет значительно больше, нежели при записи элемента в готовую пустую ячейку. Тем не менее, в среднем время вставки элемента в конец списка является постоянным.*

	Удаление последнего элемента происходит за константное время. *Недостатки ArrayList проявляются при вставке/удалении элемента в середине списка — это взывает перезапись всех элементов размещенных «правее» в списке на одну позицию влево, кроме того, при удалении элементов размер массива не уменьшается, до явного вызова метода trimToSize().* *ArrayList. Итоги:* *— Быстрый доступ к элементам по индексу за константное время O(1);* *— Доступ к элементам по значению за линейное время O(n);* *— Медленный, когда вставляются и удаляются элементы из «середины» списка;* *— Позволяет хранить любые значения в том числе и null;* *— Не синхронизирован (не потокобезопасен) .* *Подробнее о списках в следующей теме.*

Создание массива и добавление в него элементов

выполняется с помощью метода **add()**:

```
import java.util.ArrayList;
…

ArrayList friendsList = new ArrayList();
friendsList.add("Антонов Сергей");
friendsList.add("Сергеев Антон");
```

При необходимости можно **добавить** новый элемент **в указанное место** (по индексу в диапазоне от 0 до size()-1 (см. ниже)):

```
friendsList.add(1, "Андреев Игорь");
```

При добавлении и удалении элементов ArrayList меняет свой размер, который можно узнать с помощью метода **size()**.

```
System.out.println(friendsList.size());
```

Обратиться к элементу можно по его индексу с помощью метода **get()**. Напомним, что индексация в ArrayList, как и в статических массивах, начинается с 0.

```
System.out.println(friendsList.get(0));
```

Соответственно, обращение к последнему элементу массива будет выглядеть так:

```
System.out.println(friendsList.get(friendsList.size()-1));
```

В **`friendsList`** можно добавить и целое число (а не строку с именем друга), при этом компилятор java не выдаст никаких сообщений об ошибках.

```
friendsList.add(5);
```

Но в дальнейшем, например, при попытке распечатать содержимое списка, возникнут проблемы, т.к. объекты из ArrayList будут иметь разный тип.

*С помощью специального синтаксиса (угловые скобки, внутри которых указан тип) в описании классов и методов можно указать параметры-типы, которые внутри описания могут использоваться в качестве типов полей, параметров и возвращаемых значений методов. Это так называемый специальный подход - **generics** (дженерик, обобщение) или **параметризация**. Эта тема рассмотрена в дополнительным материале 2.8.*

*С использованием **generics** отсутствует необходимость преобразования типов при работе с элементами (casting) и предотвращаются ошибки при работе (ClassCastException).*

Для корректного использования объектов типа ArrayList в угловых скобках необходимо указывать тип элементов этого массива.

```
ArrayList <String> friendsList = new ArrayList <String>();
```

Замена и удаление элементов

Для замены элемента в массиве нужно использовать метод **set()** с указанием индекса и новым значением:

```
friendsList.set(1, "Викторов Андрей");
```

Для удаления элемента из массива пользуются методом **remove()**. При этом элемент можно удалять как по индексу, так и по объекту:

```
friendsList.remove("Антонов Сергей");
friendsList.remove(0);
```

Очистка списка (удаление всех элементов) - это метод **clear()**

```
friendsList.clear();
```

При поиске элемента в массиве применяют **indexOf()**, который возвращает номер элемента в диапазоне от **0** до **size()-1**, или отрицательное число, если элемент не найден.

```
int ind = friendsList.indexOf("Сергеев Антон");
System.out.println("Индекс элемента = " + ind);
```

Просмотр и поиск элементов

Просмотр всего списка можно осуществить несколькими способами:

а) через цикл **for**

```
for (int i = 0; i < friendsList.size(); i++) {
   System.out.println(friendsList.get(i));
}
```

б) через цикл **for each**

```
for (String friend : friendsList) {
   System.out.println(friend);
}
```

в) через **Iterator**

Интерфейс **Iterator** (итератор) содержит обобщенную схему доступа ко всем элементам контейнера (коллекции объектов), которая не зависит от особенностей его организации. Итератор последовательности возвращает элементы в соответствии с линейным порядком их следования.

Основными методами **Iterator** являются:

- **hasNext()** — возвращает true, если следующий элемент существует в коллекции;
- **next()** — возвращает следующий элемент коллекции, при этом итератор переходит на следующий элемент;

```
import java.util.Iterator;
…
Iterator iterator = friendsList.iterator();
while (iterator.hasNext()) {
   System.out.println(iterator.next());
}
```

Чтобы узнать, есть в массиве какой-либо элемент, можно воспользоваться методом **contains()**, который вернёт true или false:

```
if (friendsList.contains("John Johnson")) {
   System.out.println("Есть такой друг!");
}
```

В массиве ArrayList значения вполне могут совпадать (как и в обычном статическом массиве). Например, среди друзей попадаются тезки и мы их можем добавить в ArrayList. Узнать, сколько раз повторяются одинаковые элементы, можно, если обратиться к методу **frequency()** класса - родителя **Collections.**

```
import java.util.Collections;
…

ArrayList<String> friendsList = new ArrayList<String>();
friendsList.add("Иванов");
friendsList.add("Петров");
friendsList.add("Сидоров");
friendsList.add("Иванов");
int count = Collections.frequency(friendsList, "Иванов");
System.out.println(count + " ");
```

Конвертация в массив

Также можно конвертировать список ArrayList в обычный статический массив. Конвертация в массив может понадобится, например, для ускорения некоторых операций или передачи массива в качестве параметра методам, которые требуют именно массив.

Например, после конвертации можно выполнить сортировку.

```
ArrayList<String> friendsList = new ArrayList<String>();
```

```
friendsList.add("Антонов Сергей");
friendsList.add("Сергеев Антон");
friendsList.add("Викторов Андрей");
friendsList.add("Костантинов Александр");

// конвертируем ArrayList в массив
String[] myArray = friendsList.toArray(new String[friendsList.size()]);
Arrays.sort(myArray);
for (String x : myArray) {
   System.out.println(x);
}
```

Упражнение 4.1.3

Разработать программу формирования списка игроков, включающего имя игрока и количество очков, набранных в игре. Предусмотреть:

- добавление нового результата игрока (фамилии могут повторяться);
- добавление лучшего результата игрока;
- редактирование записи об игроке;
- удаление записи об игроке.

```
import java.io.PrintStream;
import java.util.ArrayList;
import java.util.Iterator;
import java.util.Scanner;

public class Player {
   private String name;
   private int score;

   Player() {
       name = "Noname";
       score = 0;
   }

   Player(String name, int score) {
       this.name = name;
       this.score = score;
   }

   // просмотр списка через объект Iterator
   public static void printListPlayer(ArrayList<Player> list) {
       Iterator<Player> iterator = list.iterator();
       while (iterator.hasNext()) {
           Player p1 = iterator.next();
           System.out.println(p1.name + "  " + p1.score);
       }
   }

   // поиск индекса заданного игрока p по фамилии
   public static int searchBestList(ArrayList<Player> list, String name) {
       int ind = 0;
       while ((ind < (list.size()))
               && !(list.get(ind).name.equals(name))) {
           ind++;
       }
       if (ind == list.size())
```

```
            ind = -1;
        return ind;
    }

    public static void main(String[] args) {

        Scanner in = new Scanner(System.in);
        PrintStream out = new PrintStream(System.out);

        ArrayList<Player> list = new ArrayList<Player>();
        int choice = 0;

        do {
            Player p = new Player();
            int ind = -1;
            System.out.println("\nСписок игроков:");
            printListPlayer(list);

            out.println("\nВыберите:");
            out.println("1 - Добавить игрока");
            out.println("2 - Удалить игрока (по индексу)");
            out.println("3 - Изменить запись об игроке");
            out.println("4 - Добавление лучшего результата игрока");
            out.println("0 - Выход");
            out.print("Ваш выбор:");
            choice = in.nextInt();
            out.println("\n\n");

            switch (choice) {
                case 1: // Добавление нового результата
                    // (фамилия игрока может повторяться)
                    out.println("Добавление результата игрока ");
                    out.print("Введите фамилию:");
                    p.name = in.next();
                    out.print("Введите очки:");
                    p.score = in.nextInt();
                    list.add(p);
                    break;

                case 2: // Удаление игрока из списка по индексу
                    out.println("Удаление по индексу");
                    if (list.size() != 0) {
                        out.print("Введите индекс:");
                        ind = in.nextInt();
                        if ((ind >= 0) && (ind < list.size())) {
                            list.remove(ind);
                        } else
                            out.print("Введите индекс от 0 до " +
                                    (list.size() - 1));
                    } else
                        out.print("Список пуст. Удаление невозможно");
                    break;
                case 3: // Изменение объекта по индексу
                    out.println("Изменение данных об игроке");
                    if (list.size() != 0) {
                        out.print("Введите индекс:");
                        ind = in.nextInt();
                        if ((ind >= 0) && (ind < list.size())) {
                            out.print("Введите фамилию:");
                            p.name = in.next();
                            out.print("Введите очки:");
                            p.score = in.nextInt();
```

```
                            list.set(ind, p);
                        } else
                            out.print("Введите индекс от 0 до " +
                                    (list.size() - 1));
                    } else
                        out.print("Список пуст. Изменение невозможно");
                    break;

                case 4:
                    out.println("Добавление лучшего результата игрока");
                    out.print("Введите фамилию:");
                    p.name = in.next();
                    out.print("Введите очки:");
                    p.score = in.nextInt();
                    ind = searchBestList(list, p.name);
                    if (ind == -1) {
                        list.add(p);
                    } else if (list.get(ind).score < p.score)
                        list.set(ind, p);
                    break;
            }
        } while (choice != 0);
        in.close();
        out.println("Программа завершена");
    }
}
```

Необходимо помнить! При вызове метода **next()** итератор переходит на следующий элемент, и тогда при проходе через ArrayList могут возникнуть “накладки”: программа распечатает имя одного игрока, а количество очков - следующего игрока:

```
while (iterator.hasNext()) {
   System.out.println(iterator.next().name + "  " + iterator.next().score);
}
```

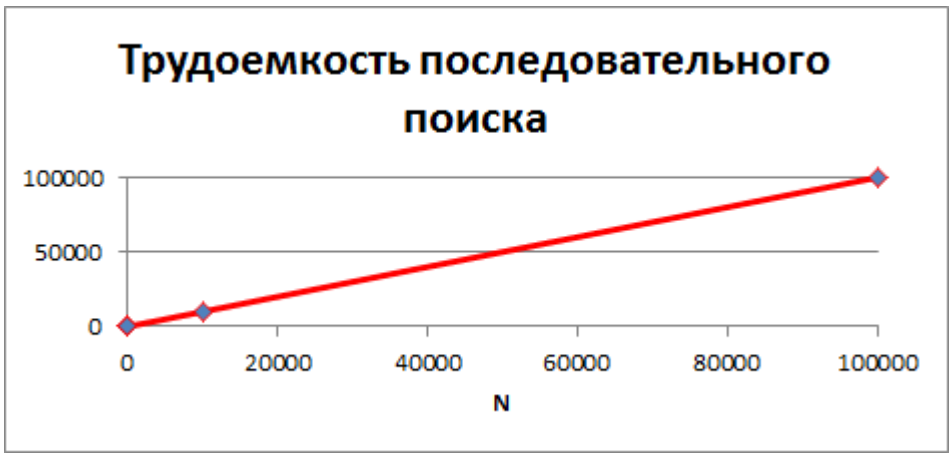
4.1.4. Алгоритмы поиска на масивах

Последовательный поиск

Представьте, что у вас есть некоторая последовательность значений и стоит задача определить, есть ли в этой последовательности некоторое заданное значение и на каком месте оно находится. Какой алгоритм необходим, чтобы решить такую задачу?

Конечно, первое, что приходит на ум - это просто перебрать последовательно все значения. Такой алгоритм называется “последовательным поиском”.

Примем количество сравнений за меру **трудоемкости алгоритма**. Если мы имеем массив длиной N, то при последовательном поиске в зависимости от того, на каком месте расположен искомый элемент, может понадобиться от 1 до N сравнений. Значит максимальная трудоемкость равна N. Таким образом получаем, что трудоемкость последовательного поиска линейно зависит от длины массива. Такую трудоемкость алгоритма обозначают O(n).



Для небольших массивов это не существенно, но для последовательностей длиной в сотни тысяч значений и более это становится проблематичным в связи со значительным увеличением времени поиска. Поэтому на практике используют более эффективные алгоритмы, и прежде всего, это двоичный поиск.

Подведем итог:

- достоинство последовательного поиска - простота в реализации;
- недостаток - низкая эффективность.

Двоичный поиск

Для поиска значений в упорядоченных последовательностях (в отсортированных по возрастанию или убыванию массивах) наиболее эффективно использовать алгоритм двоичного (бинарного) поиска.

Шаги алгоритма двоичного поиска на возрастающей последовательности:

1. Исходный диапазон поиска - это вся последовательность. Нижняя граница (lo) - это первый элемент, верхняя граница (hi) - это последний элемент последовательности.
2. Находим элемент, который расположен на средней позиции (mid) между границами текущего диапазона поиска. Найденный средний элемент сравнивается с искомым значением. Если они равны, то поиск завершен - элемент найден.
3. Если искомый элемент больше среднего, то новым диапазоном поиска становится верхняя половина текущего диапазона, поэтому нижняя граница передвигается на элемент, стоящий на одну позицию выше среднего. Переходим на шаг 5.
4. Если искомый элемент меньше среднего, то новым диапазоном поиска становится нижняя половина текущего диапазона, поэтому верхняя граница устанавливается на элемент, стоящий на одну позицию ниже от среднего. Переходим на шаг 5.
5. Если диапазон пустой, то поиска завершен - элемент не найден. Иначе - переходим на следующую итерацию с шага 2.

Пример готового алгоритма:

```
public class Binary {
    public static void main(String[] args) {
        int[] arr = {1, 12, 23, 34, 55, 61, 67, 88, 89, 101};
        System.out.println(rank(55, arr));
    }

    public static int rank(int val, int[] arr) {
        return rank(val, arr, 0, arr.length - 1);
    }
```

```
    private static int rank(int val, int[] arr, int lo, int hi) {
        if (lo > hi) return -1;

        int mid = lo + (hi - lo) / 2;

        if (val < arr[mid]) {
            return rank(val, arr, lo, mid - 1);
        } else if (val > arr[mid]) {
            return rank(val, arr, mid + 1, hi);
        } else {
            return mid;
        }
    }
}
```

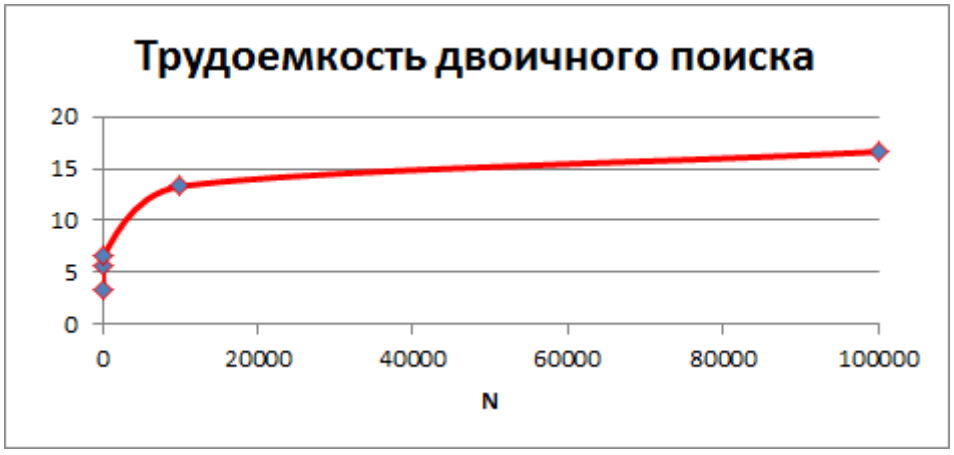
При реализации алгоритма важно не забывать о частных случаях, когда массив пуст или в нём один элемент. Полезно предусмотреть также обработку исключения в том случае, если массив не инициализирован.

Какова же трудоемкость алгоритма двоичного поиска? На каждой следующей итерации наш диапазон поиска сокращается в двое. Т.е. мы делим N на 2, потом еще раз на 2 и так до тех пор, пока диапазон не станет пустым. Значит максимальное количество сравнений, которое можно выполнить на последовательности N это $\log_2 N$, поэтому пишут, что трудоемкость двоичного поиска равна $O(\log_2 N)$.

На рисунке ниже приведена зависимость трудоемкости двоичного поиска от длины посдовательности. Сравните ее с трудоемкостью последовательного поиска при длине 100 тысяч!

Подведем итог:

- достоинство двоичного поиска - высокая эффективность;
- недостаток - т.к. работает только на упорядоченной последовательности, требует предварительной сортировки.



В Java алгоритм бинарного поиска уже реализован, например, статическими методами Arrays.binarySearch() — для массивов. В пакете java.util находится класс Arrays, который содержит методы манипулирования содержимым массива, а именно для поиска, заполнения, сравнения, преобразования в коллекцию и прочие.

int binarySearch(параметры) – перегруженный метод организации бинарного поиска значения в массивах примитивных и объектных типов. Возвращает позицию первого совпадения, для его корректной работы массив должен быть отсортирован.

Что является результатом binarySearch(...)? Этот метод возвращает индекс найденного элемента массива, если элемент был найден. Если же элемент не найден, то binarySearch(...) возвращает величину (-i-1), где i — это индекс элемента, после которого нужно вставить искомый элемент для того, чтобы сохранился порядок сортировки.

Пример поиска с классом Arrays:

```
import java.util.Arrays;

public class ArrayDemo {

   public static void main(String[] args) {
        // инициализируем массив
        int intArr[] = {30, 20, 5, 12, 55};

        // сортируем его
        Arrays.sort(intArr);

        // выводим все элементы массива
        System.out.println("Элементы сортированного массива:");
        for (int number : intArr) {
            System.out.println(number);
        }
        // зададим искомое значение
        int searchVal = 12;
        int retVal = Arrays.binarySearch(intArr, searchVal);

        System.out.println("Элемент с о значением 12 находится по адресу : "
+ retVal);
    }
}
```

Упражнение 4.1.4

Задана матрица, в которой элементы упорядочены при просмотре слева направо сверху вниз. Реализовать класс, который находит в такой матрице строку и столбец, в котором находится заданный элемент. Необходимо обработать ситуацию, когда элемент отсутствует.

Самый простой способ поиска в двумерной матрице - поиск по каждой отдельной подстроке. Конечно, в зависимости от условий задачи (например, поиск дубликатов) алгоритм может усложниться. Приведем пример для простейшего случая с квадратной матрицей чисел 4x4:

```
// инициализируем массив
int[] intArr = {30, 20, 5, 12, 55};
int[][] matrix = {{10, 20, 30, 40},
        {15, 25, 35, 45},
        {27, 29, 37, 48},
        {32, 33, 39, 50},};
int found = -1;
int x, y;
int number = 29;
int N = 4;
for (int i = 0; i < N; i++) {
   found = binarySearch(matrix[i], number, 0, N);
```

```
    if (found != -1) {
        x = i;
        y = found;
        System.out.printf("Значение найдено по адресу %d, %d", x, y);
    }
}
System.out.println("Значение не найдено");
```

Задание 4.1.1

Разработать класс “Телефонная книга”, содержащий информацию о фамилиях, именах, телефонах и датах рождения знакомых. Реализовать методы добавления информации в алфавитном порядке (по фамилии), печати списка контактов, удаления контакта, сортировки по дате рождения.

Задание 4.1.2

Напишите программу, реализующую алгоритм приближенного бинарного поиска. Программа на входе от пользователя получает два массива M и K натуральных чисел одинаковой длины. Требуется для каждого элемента массива M найти наиболее близкий ему по значению элемент массива K. Если таких элементов несколько, вывести наименьший из них.

Благодарности

Компания Samsung Electronics выражает благодарность за участие в подготовке данного материала преподавателям IT ШКОЛЫ SAMSUNG Сопченко Елене Вильевне, Фиониной Людмиле Евгеньевне и Петрушину Ивану Сергеевичу.