Лабораторная работа 2

Тема: Работа с массивами.

Цель: Изучение принципов работы массивов на примере алгоритмов сортировки.

1. Массивы в Java

Массив — это структура данных, в которой хранятся элементы одного типа.

В случае с Java массив однороден, то есть во всех его ячейках будут храниться элементы одного типа. Так, массив целых чисел содержит только целые числа (например, типа int), массив строк — только строки, массив из элементов класса будет содержать только объекты данного класса.

Объявление массива, Java-синтаксис:

```
dataType[] arrayName;
```

Как и любой другой объект, создать массив Java, то есть зарезервировать под него место в памяти, можно с помощью оператора new. Делается это так:

```
new typeOfArray [length];
```

Либо используя сокращенный синтаксис:

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

После создания массива в его ячейках записаны будут значения по умолчанию. Для числвых типов это будет 0, для boolean — false, для ссылочных типов — null.

Получить доступ к длине массива можно с помощью поля length. Пример:

```
int[] myArray = new int[10];
System.out.println(myArray.length);
```

Для быстрой инициализации элементов массива можно использовать указания значений элементов сразу при объявлении:

```
int numbers [] = {7, 12, 8, 4, 33, 79, 1, 16, 2};
```

В таком случае не нужно и явно указывать размер массива — поле length при быстрой инициализации заполнится автоматически.

Класс Java Arrays

Для работы с массивами в Java есть класс java.util.Arrays

Основные методы класса Arrays:

Метод `void sort(int[] myArray, int fromIndex, int toIndex)` сортирует массив целых чисел или его подмассив по возрастанию.

`int binarySearch(int[] myArray, int fromIndex, int toIndex, int key)`. Этот метод ищет элемент key в уже отсортированном массиве myArray или подмассиве, начиная с fromIndex и до toIndex. Если элемент найден, метод возвращает его индекс, если нет - (-fromIndex)-1.

Метод `String toString(int[] myArray)` преобразовывает массив к строке. В Java массивы не переопределяют `toString()`. Это значит, что если вывести целый массив на экран непосредственно по его имени (`System.out.println(myArray)`), то получим имя класса и шестнадцатеричный хэш-код массива. Данный метод унаследован от `Object.toString()`).

Пример:

```
class Main{
  public static void main(String[] args){
    int[] array = {1, 5, 4, 3, 7};
    System.out.println(array);
    System.out.println(array.toString());
    System.out.println(Arrays.toString(array));
    Arrays.sort(array, 0, 4);
    System.out.println(Arrays.toString(array));
    int key = Arrays.binarySearch(array, 5);
    System.out.println(key);
    System.out.println(Arrays.binarySearch(array, 0));
    }
}
```

Так же, как с методом toString(), сами по себе массивы не переопределяют метод equals(). Поэтому если сравнить их так:

```
int[] numbers = {1, 2, 3};
int[] numbers2 = {1, 2, 3};
System.out.println(numbers.equals(numbers2));
```

получим результат `false`, так как будет вызван метод `Object.equals()`, который сравнивает ссылки.

Класс Arrays содержит переопределенный метод `equals()`, который выполняет поэлементное сравнение массивов:

```
System.out.println(Arrays.equals(numbers, numbers2));
```

2. Работа с массивами на примере алгоритмов сортировки

2.1. Сортировка пузырьком

Алгоритм просматривает массив и сравнивает каждую пару соседних элементов. Когда он встречает пару элементов, расположенных не по порядку, происходит замена двух элементов местами.

Для реализации алгоритма можно воспользоваться оператором цикла `while`, в котором просматривается весь массив и меняет элементы местами при необходимости.

Массив в алгоритме считается отсортированным. При первой замене доказывается обратное и запускается еще одна итерация.

Цикл останавливается, когда все пары элементов в массиве пропускаются без замен:

```
boolean sorted = false;
int temp;
while(!sorted) {
  sorted = true;
  for (int i = 0; i < array.length - 1; i++) {
    if (array[i] > array[i+1]) {
      temp = array[i];
      array[i] = array[i+1];
      array[i+1] = temp;
      sorted = false;
    }
  }
}
```

2.2. Сортировка вставками

Данный алгоритм разделяет оригинальный массив на сортированный и несортированный подмассивы.

Длина сортированной части равна 1 в начале и соответствует первому (левому) элементу в массиве. После этого остается итерировать массив и расширять отсортированную часть массива одним элементом с каждой новой итерацией.

После расширения новый элемент помещается на свое место в отсортированном подмассиве. Это происходит путём сдвига всех элементов вправо, пока не встретится элемент, который не нужно двигать.

```
for (int i = 1; i < array.length; i++) {
        int current = array[i];
```

```
        int j = i - 1;
        while(j >= 0 && current < array[j]) {
            array[j+1] = array[j];
            j--;
        }
        array[j+1] = current;
    }
```

2.3. Сортировка выбором

Сортировка выбором тоже разделяет массив на сортированный и несортированный подмассивы. Но на этот раз сортированный подмассив формируется вставкой минимального элемента не отсортированного подмассива в конец сортированного, заменой.

В каждой итерации предполагается, что первый неотсортированный элемент минимален и перебирается по всем оставшимся элементам в поисках меньшего.

```
for (int i = 0; i < array.length; i++) {
        int min = array[i];
        int minId = i;
        for (int j = i+1; j < array.length; j++) {
            if (array[j] < min) {
                min = array[j];
                minId = j;
            }
        }
        // замена
        int temp = array[i];
        array[i] = min;
        array[minId] = temp;
    }
```

2.4. Сортировка слиянием

Сортировка слиянием эффективнее, чем примеры алгоритмов сортировки, представленные выше, благодаря использованию рекурсии и подходу «разделяй и властвуй».

Массив делится на два подмассива, а затем происходит:

- Сортировка левой половины массива (рекурсивно)
- Сортировка правой половины массива (рекурсивно)
- Слияние

В главную функцию передаются left и right – индексы подмассивов для сортировки, крайние слева и справа. Изначально они имеют значения 0 и array.length-1, в зависимости от реализации.

Основа рекурсии гарантирует, что она закончится после перебора массива, или когда left и right встретятся друг с другом. Сначала находится среднюю точку mid и рекурсивно сортируются подмассивы слева и справа от середины, в итоге объединяя результаты.

Достаточно следовать индексам не нарушая логики дерева рекурсии

```
public static void mergeSort(int[] array, int left, int
right) {
    if (right <= left) return;
    int mid = (left+right)/2;
    mergeSort(array, left, mid);
    mergeSort(array, mid+1, right);
    merge(array, left, mid, right);
}
```

Для сортировки двух подмассивов в один нужно вычислить их длину и создать временные массивы, в которые будем копировать.

После копирования проходим по результирующему массиву и назначаем текущий минимум. Теперь нужно выбрать наименьший из двух элементов, которые еще не были выбраны, и двигать итератор для этого массива вперед:

```
void merge(int[] array, int left, int mid, int right) {
    int lengthLeft = mid - left + 1;
    int lengthRight = right - mid;
    int leftArray[] = new int [lengthLeft];
    int rightArray[] = new int [lengthRight];
    // копируем отсортированные массивы во временные
    for (int i = 0; i < lengthLeft; i++)
        leftArray[i] = array[left+i];
    for (int i = 0; i < lengthRight; i++)
        rightArray[i] = array[mid+i+1];
    // итераторы содержат текущий индекс временного подмассива
    int leftIndex = 0;
    int rightIndex = 0;
    // копируем из leftArray и rightArray обратно в массив
    for (int i = left; i < right + 1; i++) {
    // если остаются нескопированные элементы в R и L, копируем
минимальный
        if (leftIndex < lengthLeft && rightIndex < lengthRight) {
            if (leftArray[leftIndex] < rightArray[rightIndex]) {
                array[i] = leftArray[leftIndex];
                leftIndex++;
            }
```

```
            else {
                array[i] = rightArray[rightIndex];
                rightIndex++;
            }
        }
        // если все элементы были скопированы из rightArray,
скопировать остальные из leftArray
        else if (leftIndex < lengthLeft) {
            array[i] = leftArray[leftIndex];
            leftIndex++;
        }
        // если все элементы были скопированы из leftArray, то
скопировать остальные из rightArray
        else if (rightIndex < lengthRight) {
            array[i] = rightArray[rightIndex];
            rightIndex++;
        }
    }
}
```

2.5. Быстрая сортировка

Данный алгоритм также реализует технику «разделяй и властвуй». Он выбирает один элемент массива в качестве «стержня» и сортирует остальные элементы вокруг (меньшие элементы налево, большие направо).

Так соблюдается правильная позиция самого «стержня». Затем алгоритм рекурсивно повторяет сортировку для правой и левой частей.

```
static int partition(int[] array, int begin, int end) {
    int pivot = end;
    int counter = begin;
    for (int i = begin; i < end; i++) {
        if (array[i] < array[pivot]) {
            int temp = array[counter];
            array[counter] = array[i];
            array[i] = temp;
            counter++;
        }
    }
    int temp = array[pivot];
    array[pivot] = array[counter];
    array[counter] = temp;
    return counter;
}

public static void quickSort(int[] array, int begin, int end) {
    if (end <= begin) return;
```

```
    int pivot = partition(array, begin, end);
    quickSort(array, begin, pivot-1);
    quickSort(array, pivot+1, end);
}
```

3. Генерация случайных чисел

Очень часто бывает нужно создать некоторые случайные данные, которые могут быть полезны как в работе алгоритмов, так и при их тестировании. Для этих целей в языке Java существует генератор случайных чисел `Random`.

Данный класс имеет два конструктора: по умолчанию, который использует текущую дату для своей инициализации и конструктор, который принимает на вход некоторое число типа `long`. Очевидно, если использовать второй конструктор с одинаковым значением параметра, то в результате будут генерироваться одинаковые случайные значения, поэтому на практике в основном применяют первый.

Рассмотрим методы классы `Random`:

- `nextBoolean()`
- `nextInt()`
- `nextLong()`
- `nextFloat()`
- `nextDouble()`

Вещественные числа генерируются только в промежутке с 0 до 1, а целочисленные по всему спектру значений. Кроме того, целые числа можно генерировать в диапазоне с 0 до max-1: `nextInt(max)`.

Пример. Заполним массив байт случайными значениями:

```
Random r = new Random();
byte[] arr = new byte[100];
r.nextBytes(arr);
for(int i = 0; i < arr.length; i++){
    System.out.println(arr[i]);
```

Аналогов функции `nextBytes` нет для других типов, поэтому придется явно инициализировать каждое значение:

```
Random r = new Random();
int[] arr = new int[100];
for(int i = 0; i < arr.length; i++){
    arr[i] = r.nextInt();
    System.out.println(arr[i]);
}
```

4. Измерение времени работы фрагмента кода

В Java существует функция `System.currentTimeMillis()` - возвращает количество миллисекунд прошедших с полуночи 1 января 1970 года, это называется UNIX-время.

Чтобы посчитать сколько времени выполнялся какой-то фрагмент кода, нужно посчитать разницу:

```
long time = System.currentTimeMillis();
someMethod();
System.out.println(System.currentTimeMillis() - time);
```

Если код выполняется очень быстро, тогда надо замерить за сколько времени выполнится большое количество повторений этого кода и разделить на количество повторений.

Если получаемое время не всегда стабильно, в этом случае надо провести несколько замеров и посчитать среднее арифметическое.

Также можно воспользоваться методом `System.nanoTime()`. - возвращает текущее значение самого точного доступного системного таймера в наносекундах.

Принцип измерения времени такой же, как и в случае с `currentTimeMillis()`:

```
long startTime = System.nanoTime();
someMethod();
long estimatedTime = System.nanoTime() - startTime;
```

Небезопасно сравнивать результаты `System.nanoTime()` вызовов между различными потоками. Даже если события потоков происходят в предсказуемом порядке, разница в наносекундах может быть положительной или отрицательной. `System.currentTimeMillis()` является безопасным для использования между потоками.

`CurrentTimeMillis()` - это часовое время, которое меняется из-за перехода на летнее время, изменения пользователями настроек времени, високосных секунд и синхронизации времени интернета. Если приложение зависит от монотонно увеличивающихся значений прошедшего времени, то лучше предпочесть `nanoTime()` вместо этого.

Клонирование массивов

Присваивание переменной значения другой переменной ссылочного типа приведет только к копированию ссылки, а не самого объекта.

В случае с массивом можно было бы осуществить поэлементное копирование:

```
int[] a = { 4, 1, 3, 2 };
```

```
int[] b = new int[a.length];
for (int i = 0; i < a.length; i++) {
    b[i] = a[i];
}
```

Более удобным вариантом является метод клонирования `clone()`, определенный в классе Object.

Так как массивы наследуют класс Object, то можно использовать `Object.clone()`:

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

Так же, в утилитном классе Arrays определен метод `Arrays.copyOf()`.

`Arrays.copyOf(a, a.length)` – первым параметром указывается массив, вторым количество копируемых элементов.

Задание на лабораторную работу:

I. Изучение времени работы алгоритмов

1. Вынести каждый из рассмотренных методов в отдельные функции.
2. Сгенерировать три массива с количеством элементов не менее 10 000:
   - Полностью отсортированный массив значений
   - Полностью случайных набор значений
   - Отсортированный массив, в котором первые 10% от общего числа элементов случайные
3. Замерить время сортировки для каждого из массивов используя подготовленные функции.
4. Замерить время сортировки массивов методом `sort()` класса `Arrays`
5. Сравнить результаты и сделать выводы.

II. Реализация алгоритмов работы с массивами:

1. Напишите функцию `int[] removeDuplicates(int[] array)`, которая возвращает массив, в котором удалены повторяющиеся элементы из массива.
2. Напишите функцию `int[] getFirst(int[] array, int n)`, которая возвращает фрагмент массива, содержащий первые ‘n’ элементов массива.
3. Напишите функцию `int[] getLast(int[] array, int n)`, которая возвращает фрагмент массива, содержащий последние ‘n’ элементов массива.
4. Напишите функцию `int countIdentic(int[] array)`, которая возвращает количество повторяющихся элементов в массиве.

- При реализации методов не использовать классы коллекций и их методов, а также сторонних библиотек классов.
- Продемонстрировать работу разработанных методов.