Algorithmization

***1. Дайте определение массиву. Как осуществляется индексация элементов массива. Как необходимо обращаться к i-му элементу массива?***

Массив — это структура данных, в которой хранятся элементы одного типа. Его можно представить, как набор пронумерованных ячеек, в каждую из которых можно поместить какие-то данные (один элемент данных в одну ячейку). Доступ к конкретной ячейке осуществляется через её номер. Номер элемента в массиве также называют индексом. Нумерация массива начинается с нуля. Обращаться к элементу массива необходимо по его индексу:

a[1] = 5; // *элементу массива a с индексом 1 было присвоено значение 5.*

***2. Приведите способы объявления и инициализации одномерных и двумерных массивов примитивных и ссылочных типов. Укажите разницу, между массивами примитивных и ссылочных типов.***

Объявление массива:

int [] a; //*объявляем тип массива* ***int****,* ***[]*** *– означают, что это массив,* ***a*** *– имя массива (этот способ объявления предпочтительней)*

int a [];

Инициализация массива:

int[] a = new int[5]; // *оператор* ***new*** *выделяет в памяти место под массив с 5 элементами, по умолчанию все элементы имеют значение 0, для ссылок null, для boolean – false.*

String [] stA;

stA = new String[]{“one”, “two”, “three”};

или

stA = {“one”, “two”, “three”}; // *сразу происходит инициализация массива*

инициализация элементов массива осуществляется по его индексу:

stA [2] = “four”; // *элементу массива с индексом 2 присвоено значение four*

Двумерный массив в Java — это массив массивов, то есть в каждой его ячейке находится ссылка на некий массив. Но гораздо проще его представить в виде таблицы, у которой задано количество строк (первое измерение) и количество столбцов (второе измерение). Двумерный массив, у которого все строки имеют равное количество элементов, называется прямоугольным. Двумерные массивы объявляются и инициализируются следующим образом:

int [][] a = new int[3][4];//был объявлен и создан двумерный массив с типом данных int с количеством элементов в строке 3 и столбце 4. Массив содержит данные по умолчанию, т.е. все 0.

Или можно сразу проинициализировать массив 3х4:

int[][] a={{1, 2, 3, 4}, {5, 6, 7, 8}, {9, 10, 11, 12}};

Массивы объектов и массивы примитивов по-разному хранятся в памяти. В случае с примитивами массив хранит множество конкретных значений. В случае с объектами – массив хранит множество ссылок. Элементы массива хранятся в едином блоке, а уже индекс указывает на конкретный адрес. Массив сам является объектом.

***3. Объясните, что значит клонирование массива, как в Java можно клонировать массив, в чем состоит разница в клонировании массивов примитивных и ссылочных типов.***

Метод ***clone*** (клонирование) для примитивных типом возвращает ссылку на новый массив, который содержит те же элементы, что и исходный массив. Исходный массив и клонированные не равны между собой, это отдельные объекты, только одинаковые по содержанию.

Если исходный массив содержит ссылки, то метод ***clone*** вернет ссылку на новый массив, который ссылается на те же объекты, что и исходный массив. Если модифицировать объект, доступ к которому осуществляется через клонированный массив, изменения будут отражаться при обращении к одному и тому же объекту в исходном массиве, поскольку они указывают на туже ссылку. Однако изменения в самом массиве затронут только этот массив.

***4. Объясните, что представляет собой двумерных массив в Java, что такое “рваный массив”. Как узнать количество строк и количество элементов в каждой строке для “рваного” массива?***

Двумерный массив в Java — это массив массивов, то есть в каждой его ячейке находится ссылка на некий массив. Но гораздо проще его представить в виде таблицы, у которой задано количество строк (первое измерение) и количество столбцов (второе измерение). Двумерный массив, у которого все строки имеют равное количество элементов, называется прямоугольным.

int [][] a = new int[5][7];

или

int [][] a = {{2, 1},{3, 5},{4, 8}}; // массив [3][2]

Рваный массив имеет разную длину строк

int [][] a = new int [5][];

или

int [][] a = {{2, 5}, {3, 5, 6}, {5, 6, 8, 9}};

Длину каждой строки в рваном массиве можно узнать с помощью переменной ***length***. Для этого вызываем строку массива, длину которой мы хотим узнать:

a [5].length; //для строки массива под индексом 5.

***5. Объясните ситуации, когда в java-коде могут возникнуть следующие исключительные ситуации java.lang.AIndexOutOfBoundsException и java.lang.AStoreException.***

java.lang.AIndexOutOfBoundsException – исключение возникает, когда программа пытается обратиться к элементу (его индексу) за пределами массива, т.е. если в массиве 5 элементов, а мы обратимся к 6, то возникнет ошибка java.lang.AIndexOutOfBoundsException.

java.lang.AStoreException – исключение возникает, когда попытаться записать в ячейку массива ссылку на объект неправильного типа, т.е. если массив содержит элементы типа String, а в него пытаются записать элемент типа Integer, то возникнет ошибка java.lang.AStoreException.

***6. Объясните, зачем при кодировании разделять решаемую задачу на методы. Поясните, как вы понимаете выражение: “Один метод не должен выполнять две задачи”.***

Метод — это именованный блок кода, который объявляется внутри класса и может быть использован многократно. Хорошо написанный метод решает одну практическую задачу.

Метод возвращает одно значение определенного типа или не возвращает вообще (void). При решении задачи часто возникает ситуация, когда для ее решения необходимо решить промежуточные задачи (подзадачи) или же сложную задачу необходимо разбить на более простые и мелкие подзадачи. Поэтому для улучшения читабельности кода и более быстрому поиску ошибок в коде необходимо разбивать весь кода на методы. Каждый метод должен выполнять одну задачу. В этому случае главный метод (main) будет выглядеть более наглядно и читабельно, а внутри метод будет обращаться к другим методам, которые написаны отдельно.

***7. Объясните, как в Java передаются параметры в методы, в чем особенность передачи в метод значения примитивного типа, а в чем ссылочного.***

Java передает параметры в метод ВСЕГДА по значению, т.е. скопировать значение и передать.

Передача примитивных типов:

void sum (int x, int y){

System.out.println (x + y);

}

int a = 5;

int b = 6;

sum (a, b);

При передаче ссылки на объект в метод, передается копия ссылки. Объект по-прежнему находится внутри кучи и ему не важна какая из ссылок к нему обращается – первоисточник или его копия.

void voice(A){

System.out.println (“Гав!”);

}

Dog B = new Dog();

voice(B);

Для примитивных типов — передаете копию текущего значения, для ссылок на объекты — передаете копию ссылки на объект. Все объекты всегда находятся в куче.

***8. Объясните, как в метод передать массив. И как массив вернуть из метода. Можно ли в методе изменить размер переданного массива.***

Также как можно передать значение примитивного типа в метод, можно также передать массив в метод. Например:

public static void printArr(int[] arr) {

for (int i = 0; i < arr.length; i++) {

System.out.print(a[i] + " "); } }

Метод можно вызвать путем передачи в него массива:

printArr(new int[]{1, 3, 52, 32, 5});

Результатом работы метода будет выводи на экран содержания массива.

Метод может также возвращать массив. Например:

public static int[] reverse(int[] list) {

int[] result = new int[list.length];

for (int i = 0, j = result.length - 1; i < list.length; i++, j--) {

result[j] = list[i]; }

return result; }

Результатом работы метода будет новый массив result.

После создания массива его размер не может быть изменен. Вместо этого массив может быть только «изменен», создав новый массив с соответствующим размером и скопировав элементы из существующего массива в новый

***9. Поясните, что означает выражение ‘вернуть значение из метода’. Как можно вернуть значение из метода. Есть ли разница при возврате значений примитивного и ссылочного типов.***

Для возврата значений из метода используется оператор return. После оператора return указывается возвращаемое значение, которое является результатом метода. Это может быть литеральное значение, значение переменной или какого-то сложного выражения.

В методе в качестве типа возвращаемого значения вместо void используется любой другой тип. Если метод возвращает значение типа int, то этот тип указывается перед названием метода. Причем если в качестве возвращаемого типа для метода определен любой другой, отличный от void, то метод обязательно должен использовать оператор return для возвращения значения.

При этом возвращаемое значение всегда должно иметь тот же тип, что значится в определении функции. И если функция возвращает значение типа int, то после оператора return стоит целочисленное значение, которое является объектом типа int.

При возврате ссылочных типов возвращается ссылка на объект, а при возврате примитивных – возвращается значение примитивных типов.

***10. Перечислите известные вам алгоритмы сортировки значений, приведите код, реализующий это алгоритмы.***

**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.

Функция входит в цикл while, в котором проходит 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:

public static void bubbleSort(int[] a) {

boolean sorted = false;

int temp;

while(!sorted) {

sorted = true;

for (int i = 0; i < a.length - 1; i++) {

if (a[i] > a[i+1]) {

temp = a[i];

a[i] = a[i+1];

a[i+1] = temp;

sorted = false;

}

}

}

}

**Сортировка вставками**

Этот 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.

public static void insertionSort(int[] a) {

for (int i = 1; i < a.length; i++) {

int current = a[i];

int j = i - 1;

for (; j >= 0; j--) {

if (current < a[j]) {

a[j + 1] = a[j];

} else break;

}

a[j+1] = current;

}

}

**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 вы предполагаете, что первый неотсортированный элемент минимален и итерируете по всем оставшимся элементам в поисках меньшего.

После нахождения текущего минимума неотсортированной части массива вы меняете его местами с первым элементом, и он уже часть отсортированного массива:

public static void selectionSort(int[] a) {

for (int i = 0; i < a.length; i++) {

int min = a[i];

int minId = i;

for (int j = i+1; j < a.length; j++) {

if (a[j] < min) {

min = a[j];

minId = j;

}

}

// замена

int temp = a[i];

a[i] = min;

a[minId] = temp;

}

**Сортировка слиянием**

Сортировка слиянием эффективнее, чем примеры алгоритмов сортировки, представленные выше, благодаря использованию рекурсии и подходу [«разделяй и властвуй»](https://ru.wikipedia.org/wiki/%D0%A0%D0%B0%D0%B7%D0%B4%D0%B5%D0%BB%D1%8F%D0%B9_%D0%B8_%D0%B2%D0%BB%D0%B0%D1%81%D1%82%D0%B2%D1%83%D0%B9_(%D0%B8%D0%BD%D1%84%D0%BE%D1%80%D0%BC%D0%B0%D1%82%D0%B8%D0%BA%D0%B0)).

Массив делится на два подмассива, а затем происходит:

1. Сортировка левой половины массива (рекурсивно)
2. Сортировка правой половины массива (рекурсивно)
3. Слияние

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

Основа нашей рекурсии гарантирует, что мы выйдем, когда закончим, или когда left и right встретятся друг с другом. Мы находим среднюю точку mid и рекурсивно сортируем подмассивы слева и справа от середины, в итоге объединяя наши решения.

public static void mergeSort(int[] a, int left, int right) {

if (right <= left) return;

int mid = (left+right)/2;

mergeSort(a, left, mid);

mergeSort(a, mid+1, right);

merge(a, left, mid, right);

}

Для сортировки двух подмассивов в один нужно вычислить их длину и создать временные массивы, в которые будем копировать. Так можно свободно изменять главный массив.

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

void merge(int[] a, int left, int mid, int right) {

*// вычисляем длину*

int lengthLeft = mid - left + 1;

int lengthRight = right - mid;

*// создаем временные подмассивы*

int leftA[] = new int [lengthLeft];

int rightA[] = new int [lengthRight];

*// копируем отсортированные массивы во временные*

for (int i = 0; i < lengthLeft; i++)

leftA[i] = a[left+i];

for (int i = 0; i < lengthRight; i++)

rightA[i] = a[mid+i+1];

*// итераторы содержат текущий индекс временного подмассива*

int leftIndex = 0;

int rightIndex = 0;

*// копируем из leftA и rightA 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 (leftA[leftIndex] < rightA[rightIndex]) {

a[i] = leftA[leftIndex];

leftIndex++;

}

else {

a[i] = rightA[rightIndex];

rightIndex++;

}

}

*// если все элементы были скопированы из rightA, скопировать остальные из leftA*

else if (leftIndex < lengthLeft) {

a[i] = leftA[leftIndex];

leftIndex++;

}

*// если все элементы были скопированы из leftA, скопировать остальные из rightA*

else if (rightIndex < lengthRight) {

a[i] = rightA[rightIndex];

rightIndex++;

}

}

}

**Челночная сортировка (Shuttle Sort)**

Среди простых сортировок есть ещё одна — челночная сортировка. Суть алгоритма в том, что мы итерируемся слева направо, при этом при выполнении swap элементов мы выполняем проверку всех остальных элементов, которые остались позади, не нужно ли повторить swap.

int[] a = {10, 2, 10, 3, 1, 2, 5};

System.out.println(As.toString(a));

for (int i = 1; i < a.length; i++) {

if (a[i] < a[i - 1]) {

swap(a, i, i - 1);

for (int z = i - 1; (z - 1) >= 0; z--) {

if (a[z] < a[z - 1]) {

swap(a, z, z - 1);

} else {

break;

}

}

}

}

System.out.println(As.toString(a));

**Сортировка Шелла**

Ещё одной простой сортировкой является сортировка Шелла. Суть её похожа на сортировку пузырьком, но каждую итерацию мы имеем разный промежуток между сравниваемыми элементами. Каждую итерацию он уменьшается вдвое.

int[] a = {10, 2, 10, 3, 1, 2, 5};

for (int step = a.length / 2; step > 0; step /= 2) {

for (int i = step; i < a.length; i++) {

for (int j = i - step; j >= 0 && a[j] > a[j + step]; j -= step) {

int temp = a[j];

a[j] = a[j + step];

a[j + step] = temp;

}

}

}

System.out.println(As.toString(a));

**Пирамидальная сортировка**

Для понимания работы пирамидального алгоритма сортировки нужно понять структуру, на которой он основан – пирамиду.

Пирамида или двоичная куча – это дерево, в котором каждый узел состоит в отношениях с дочерними узлами. Добавление нового узла начинается с левой позиции нижнего неполного уровня.

По мере движения вниз по дереву значения уменьшаются (min-heap) или увеличиваются (max-heap).

Чтение графа сверху вниз здесь представлено слева направо. Мы добились того, что позиция дочернего элемента по отношению к k-ому элементу в массиве – 2\\*k+1 и 2\\*k+2 (при условии, что индексация начинается с 0).

И наоборот, для k-го элемента дочерняя позиция всегда равна (k-1)/2.

С этими знаниями вы сделаете max-heap из любого массива! Для этого проверьте каждый элемент на условие, что каждый из его дочерних элементов имеет меньшее значение.

Условие верно? Тогда меняйте местами один из дочерних элементов с родительским и повторяйте рекурсию с новым родительским элементом (он может всё ещё быть больше другого дочернего).

static void heapify(int[] a, int length, int i) {

int leftChild = 2\*i+1;

int rightChild = 2\*i+2;

int largest = i;

*// если левый дочерний больше родительского*

if (leftChild < length && a[leftChild] > a[largest]) {

largest = leftChild;

}

*// если правый дочерний больше родительского*

if (rightChild < length && a[rightChild] > a[largest]) {

largest = rightChild;

}

*// если должна произойти замена*

if (largest != i) {

int temp = a[i];

a[i] = a[largest];

a[largest] = temp;

heapify(a, length, largest);

}

}

public static void heapSort(int[] a) {

if (a.length == 0) return;

*// Строим кучу*

int length = a.length;

// проходим от первого без ответвлений к корню

for (int i = length / 2-1; i >= 0; i--)

heapify(a, length, i);

for (int i = length-1; i >= 0; i--) {

int temp = a[0];

a[0] = a[i];

a[i] = temp;

heapify(a, i, 0);

}

}

**Быстрая сортировка**

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[] a, int begin, int end) {

int pivot = end;

int counter = begin;

for (int i = begin; i < end; i++) {

if (a[i] < a[pivot]) {

int temp = a[counter];

a[counter] = a[i];

a[i] = temp;

counter++;

}

}

int temp = a[pivot];

a[pivot] = a[counter];

a[counter] = temp;

return counter;

}

public static void quickSort(int[] a, int begin, int end) {

if (end <= begin) return;

int pivot = partition(a, begin, end);

quickSort(a, begin, pivot-1);

quickSort(a, pivot+1, end);

}