**Отчет по лабораторной работе №2**

На тему «Сортировки»

Вариант №17

Выполнил: ст.группы

2221121

Хайрутдинов А.А.

Проверил:

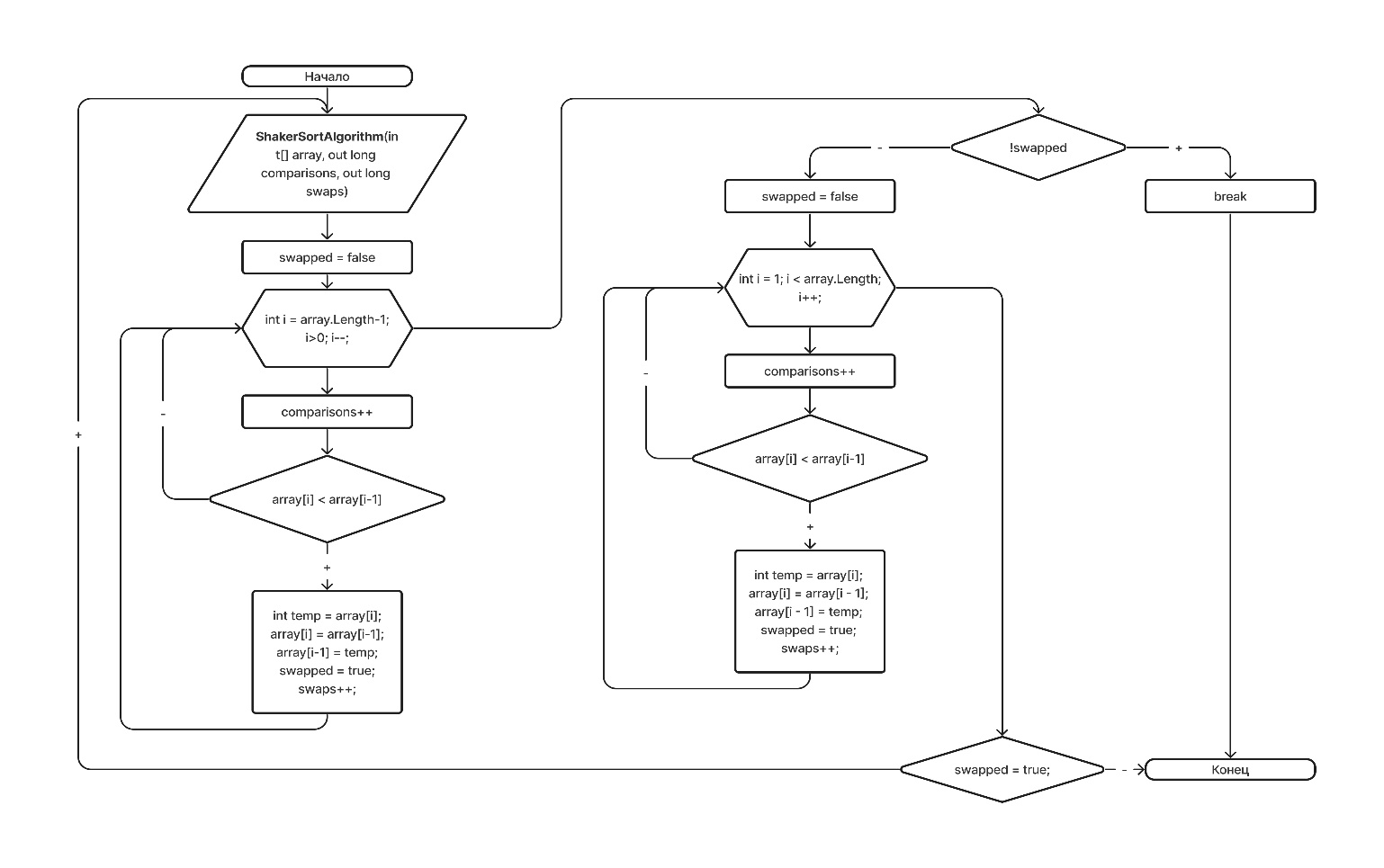
Хузятова Л.Б.

**1.Шейкерная сортировка / Shakersort**

**Описане алгоритма** Алгоритм шейкерной сортировки, также известный как алгоритм коктейльной сортировки или двунаправленная сортировка (Bidirectional Bubble Sort), является вариантом сортировки пузырьком. Он работает по тому же принципу, что и обычная сортировка пузырьком, но с движением в обе стороны. Это позволяет уменьшить количество итераций в случаях, когда большая часть массива уже отсортирована, и обеспечивает более эффективное уменьшение неотсортированных элементов на каждой итерации.

Вот общий алгоритм шейкерной сортировки:

1. Инициализация переменных: Устанавливаются переменные comparisons и swaps в ноль. Также создается переменная swapped, которая будет указывать, были ли выполнены обмены на текущей итерации.
2. Цикл сортировки: Алгоритм работает в цикле do-while, который будет повторяться, пока не будет выполнена ни одна перестановка на текущей итерации.
3. Шаг вверх по массиву: Начиная с конца массива, проходимся по элементам от конца к началу. На каждой итерации сравниваем текущий элемент с предыдущим. Если текущий элемент меньше предыдущего, меняем их местами и устанавливаем флаг swapped в true.
4. Проверка наличия обменов: Если в этой итерации не было выполнено ни одной перестановки (то есть swapped остается false), завершаем цикл.
5. Шаг вниз по массиву: После завершения шага вверх, начинаем шагать вниз по массиву, начиная с начала массива и идя к его концу. На каждой итерации сравниваем текущий элемент с предыдущим. Если текущий элемент меньше предыдущего, меняем их местами и устанавливаем флаг swapped в true.
6. Проверка наличия обменов: Если в этой итерации не было выполнено ни одной перестановки (то есть swapped остается false), завершаем цикл.
7. Завершение: После выполнения всех итераций возвращаем отсортированный массив.

****

**Код программы:**

public static void ShakerSortAlgorithm(int[] array, out long comparisons, out long swaps)

{

comparisons = 0;

swaps = 0;

bool swapped;

do

{

swapped = false;

for (int i = array.Length - 1; i > 0; i--)

{

comparisons++; // увеличиваем счетчик сравнений

if (array[i] < array[i - 1])

{

int temp = array[i];

array[i] = array[i - 1];

array[i - 1] = temp;

swapped = true;

swaps++; // увеличиваем счетчик обменов

}

}

if (!swapped)

break;

swapped = false;

for (int i = 1; i < array.Length; i++)

{

comparisons++; // увеличиваем счетчик сравнений

if (array[i] < array[i - 1])

{

int temp = array[i];

array[i] = array[i - 1];

array[i - 1] = temp;

swapped = true;

swaps++; // увеличиваем счетчик обменов

}

}

} while (swapped);

}

**Таблица измерений:**

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| **n** | **Параметр** | **Сгенерированные массивы** | | | | **Среднее значение** |
| **Массив а** | **Массив б** | **Массив в** | **Массив г** |
| 128 | Compare | 8636 | 11811 | 8001 | 1397 | 7461,25 |
|  | Swap | 3960 | 4327 | 3600 | 293 | 3045 |
|  | Time | 1679 | 391 | 283 | 36 | 597,25 |
| 1024 | Compare | 523776 | 696663 | 517638 | 99231 | 459327 |
|  | Swap | 255452 | 325344 | 252609 | 27033 | 215109,5 |
|  | Time | 31149 | 20464 | 16648 | 2293 | 17638,5 |
| 16384 | Compare | 133242939 | 179754276 | 133242939 | 25442799 | 117920738,3 |
|  | Swap | 66064890 | 82318937 | 66020588 | 6722960 | 55281843,75 |
|  | Time | 5566853 | 5971943 | 4215012 | 573837 | 4081911,25 |

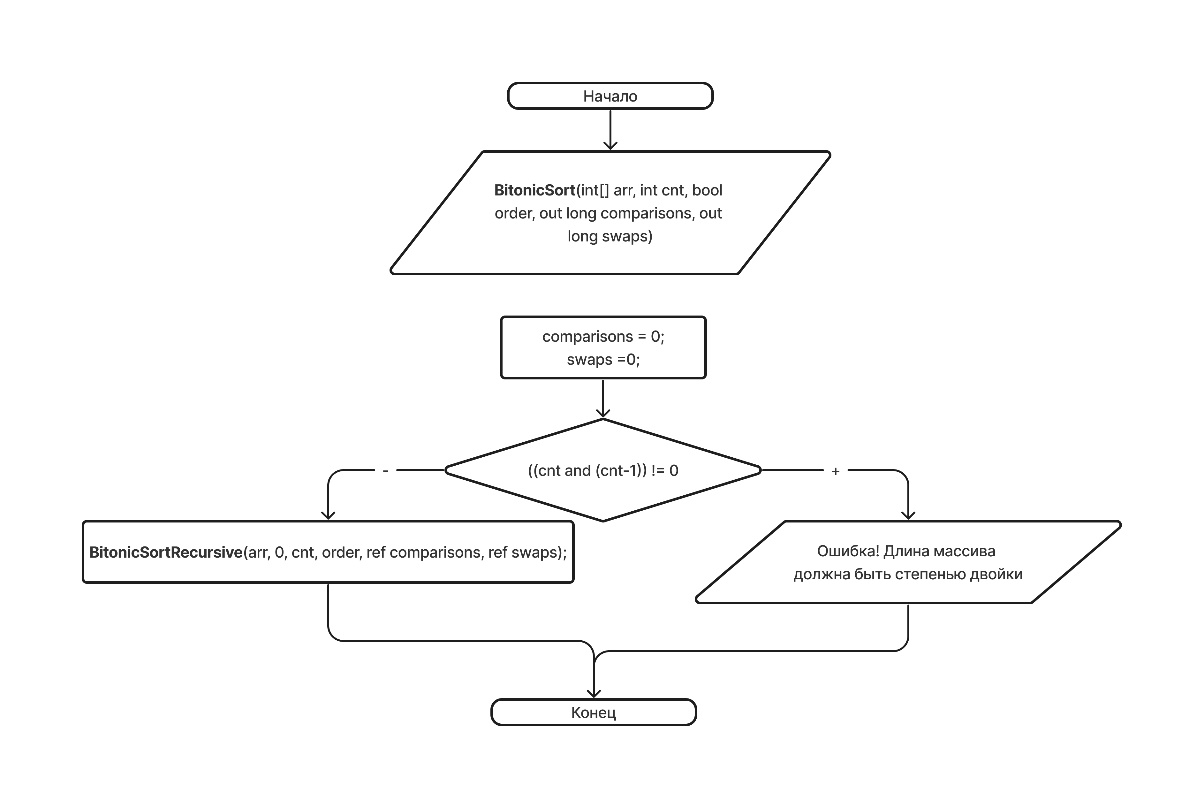
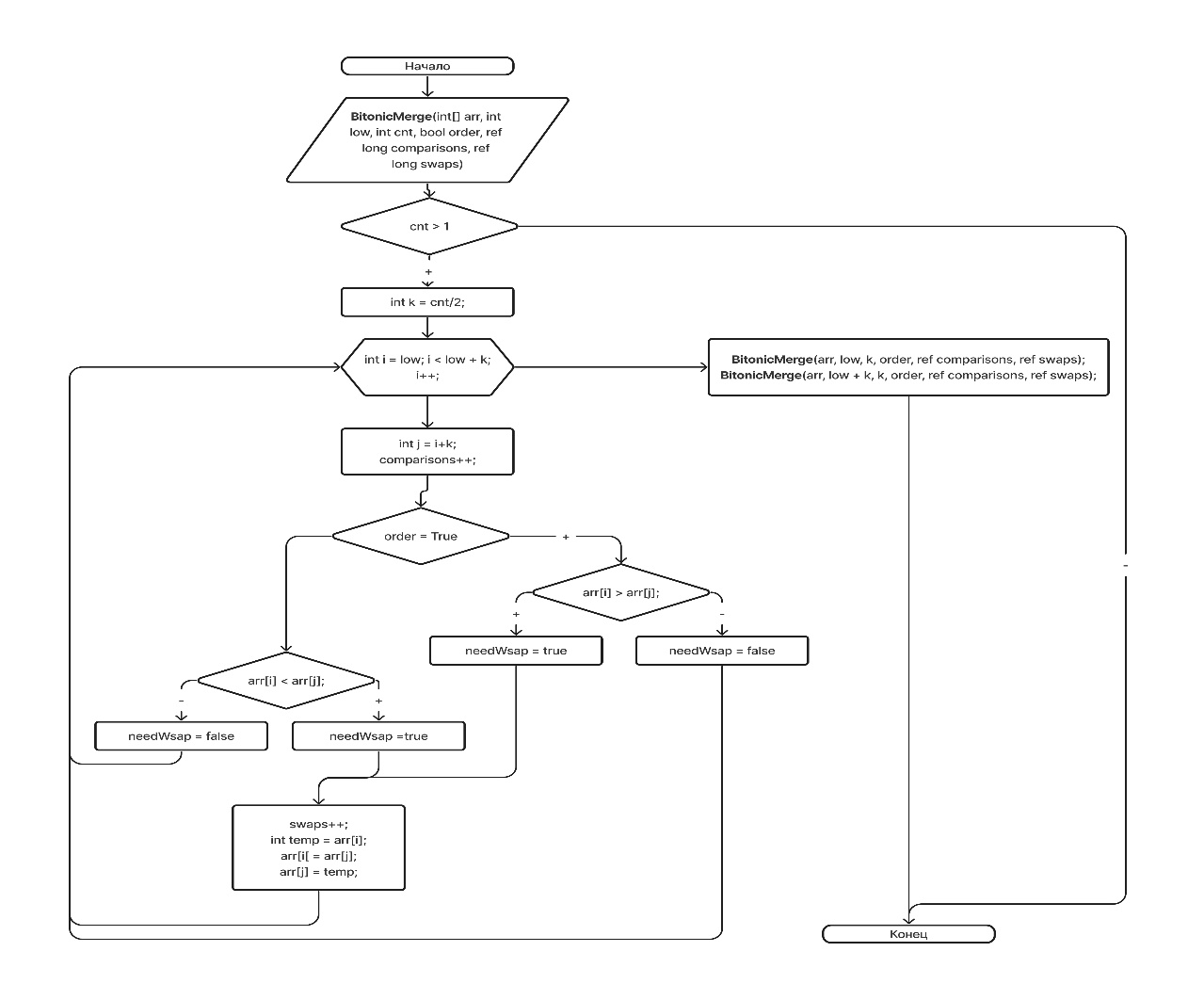
**2. Битонная сортировка / Bitonicsort**

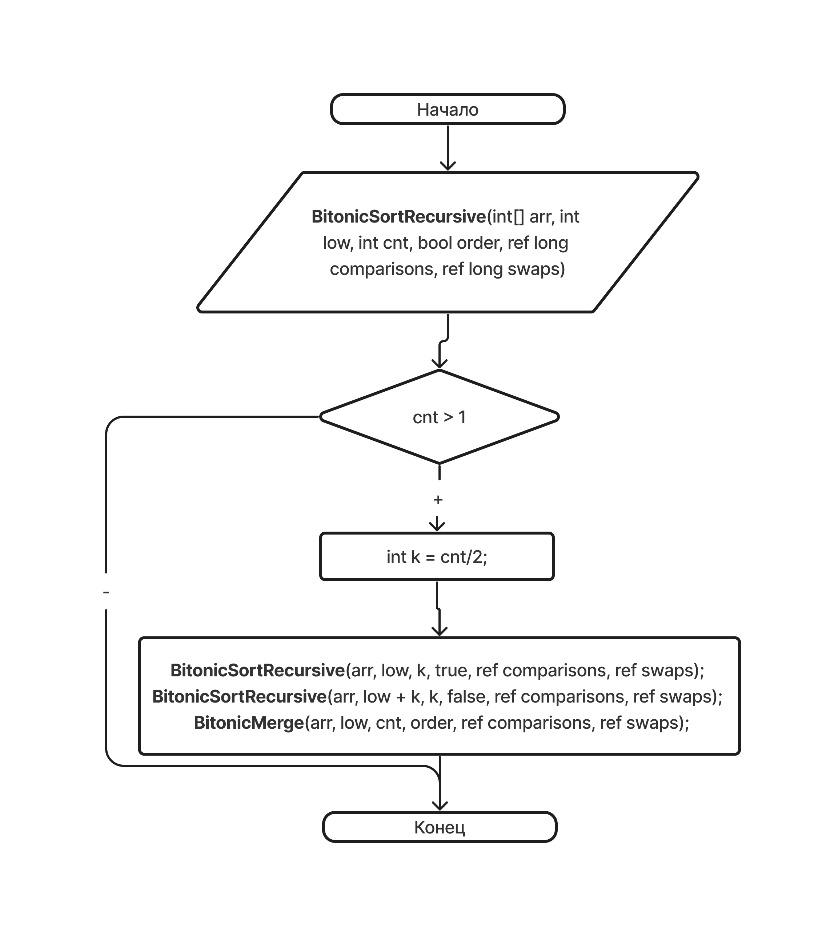
**Описание алгоритма**

Битонная сортировка (Bitonic sort) — это параллельный алгоритм сортировки, который работает на основе сети слияния-сортировки. Основная идея этого алгоритма заключается в том, чтобы разбить исходный массив на битонные последовательности, а затем объединить их в отсортированном порядке.

Вот описание алгоритма:

1. Проверка длины массива: Проверяем, является ли длина массива степенью двойки. Если нет, выбрасываем исключение, так как битонная сортировка требует, чтобы длина массива была степенью двойки.
2. Рекурсивная сортировка: Вызываем рекурсивную функцию BitonicSortRecursive для сортировки массива. На этом этапе массив делится на две части, каждая из которых сортируется отдельно.
3. Слияние: Выполняем слияние отсортированных подмассивов с помощью функции BitonicMerge.
4. Битонное слияние: Функция BitonicMerge выполняет битонное слияние для слияния двух отсортированных половин массива. На этом этапе каждая половина массива считается битонной последовательностью. Сначала сравниваются элементы с определенным порядком (возрастающим или убывающим), а затем происходит обмен элементами, если это необходимо для правильной сортировки.
5. Рекурсивное битонное слияние: После того как массив разделен на две части, каждая часть снова сортируется рекурсивно с использованием битонной сортировки. Это происходит до тех пор, пока каждая половина массива не будет отсортирована.

Алгоритм продолжает делить массив на подмассивы до тех пор, пока длина подмассивов не станет равной 1. Каждый шаг деления выполняется параллельно для различных частей массива, что делает битонную сортировку эффективной для многопроцессорных и параллельных систем. 



**Код программы:**

public static void BitonicSort(int[] arr, int cnt, bool order, out long comparisons, out long swaps)

{

comparisons = 0;

swaps = 0;

// Проверяем, является ли cnt степенью двойки

if ((cnt & (cnt - 1)) != 0)

{

throw new ArgumentException("Длина массива должна быть степенью двойки");

}

BitonicSortRecursive(arr, 0, cnt, order, ref comparisons, ref swaps);

}

private static void BitonicMerge(int[] arr, int low, int cnt, bool order, ref long comparisons, ref long swaps)

{

if (cnt > 1)

{

int k = cnt / 2;

for (int i = low; i < low + k; i++)

{

int j = i + k;

// Увеличиваем счетчик операций сравнения

comparisons++;

// Выбираем правильный порядок для сравнения

bool needSwap = order ? (arr[i] > arr[j]) : (arr[i] < arr[j]);

if (needSwap)

{

swaps++; // Увеличиваем счетчик операций обмена

int temp = arr[i];

arr[i] = arr[j];

arr[j] = temp;

}

}

// Рекурсивно сливаем обе части массива

BitonicMerge(arr, low, k, order, ref comparisons, ref swaps);

BitonicMerge(arr, low + k, k, order, ref comparisons, ref swaps);

}

}

private static void BitonicSortRecursive(int[] arr, int low, int cnt, bool order, ref long comparisons, ref long swaps)

{

if (cnt > 1)

{

int k = cnt / 2;

BitonicSortRecursive(arr, low, k, true, ref comparisons, ref swaps);

BitonicSortRecursive(arr, low + k, k, false, ref comparisons, ref swaps);

BitonicMerge(arr, low, cnt, order, ref comparisons, ref swaps);

}

}

**Таблица измерений:**

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| **n** | **Параметр** | **Сгенерированные массивы** | | | | **Среднее значение** |
| **Массив а** | **Массив б** | **Массив в** | **Массив г** |
| 128 | Compare | 1792 | 1792 | 1792 | 1792 | 1792 |
|  | Swap | 829 | 724 | 827 | 652 | 758 |
|  | Time | 2555 | 195 | 215 | 168 | 783,25 |
| 1024 | Compare | 28160 | 28160 | 28160 | 28160 | 28160 |
|  | Swap | 11947 | 10116 | 11965 | 7315 | 10335,75 |
|  | Time | 4917 | 2071 | 1971 | 1736 | 2673,75 |
| 16384 | Compare | 860160 | 860160 | 860160 | 860160 | 860160 |
|  | Swap | 309492 | 249678 | 308880 | 127880 | 248982,5 |
|  | Time | 48410 | 44403 | 44078 | 33607 | 42624,5 |

**3. Пирамидальная сортировка / Heapsort**

**Описание алгоритма**  
 Алгоритм пирамидальной сортировки (Heap Sort) является одним из эффективных алгоритмов сортировки, основанных на структуре данных "куча" (heap). Он основан на принципе преобразования массива в пирамиду (кучу) и последующем извлечении элементов из корня пирамиды, что обеспечивает сортировку элементов по возрастанию или убыванию.

Вот общий алгоритм пирамидальной сортировки:

1. Построение кучи (heapify):

Начиная с середины массива и двигаясь к началу, для каждого узла выполнить процедуру Heapify, чтобы убедиться, что для каждого узла поддеревья, корни которых больше узла, являются также пирамидой.

1. Сортировка:

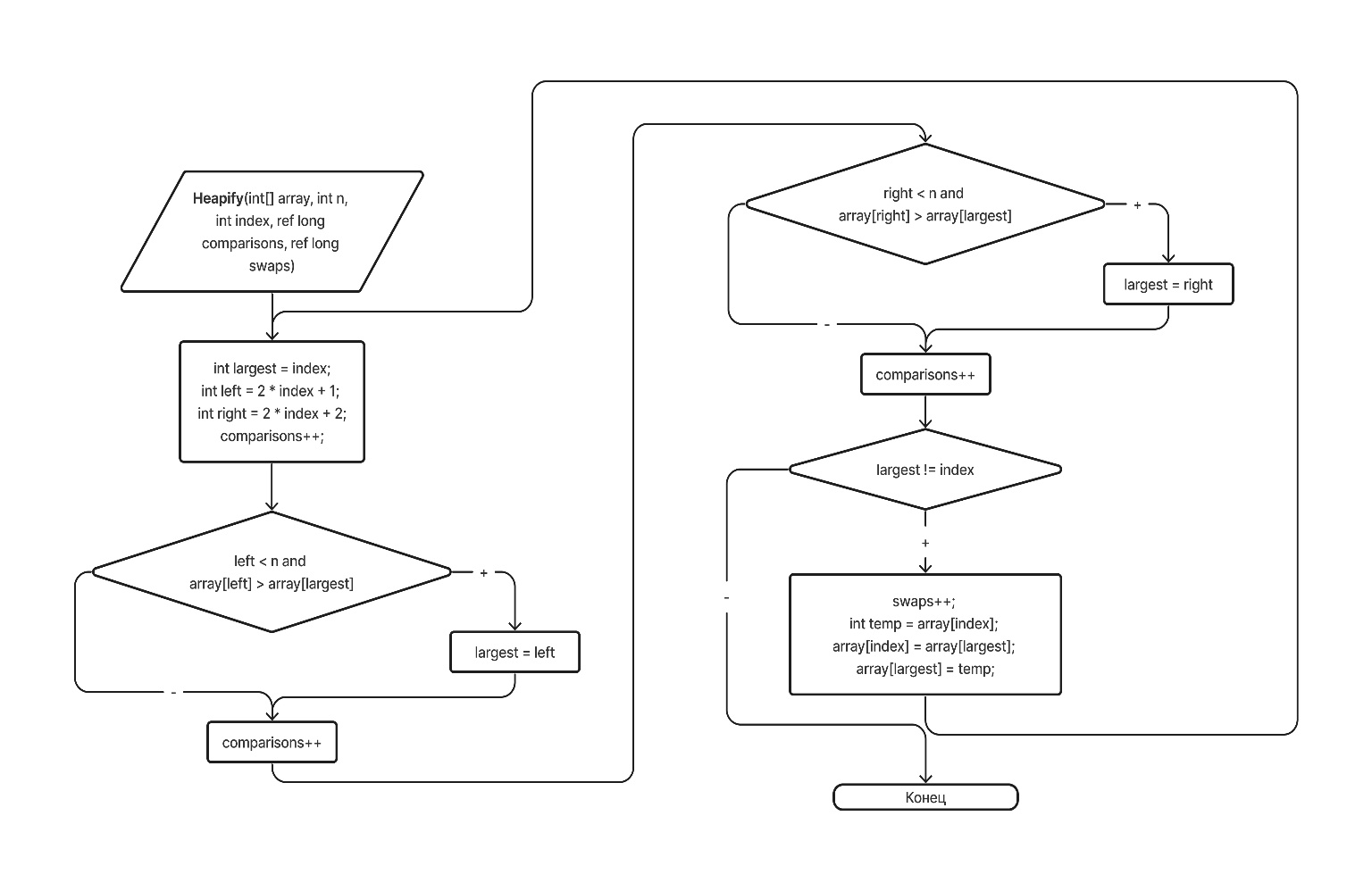
После построения кучи, корень пирамиды содержит наибольший (или наименьший) элемент.

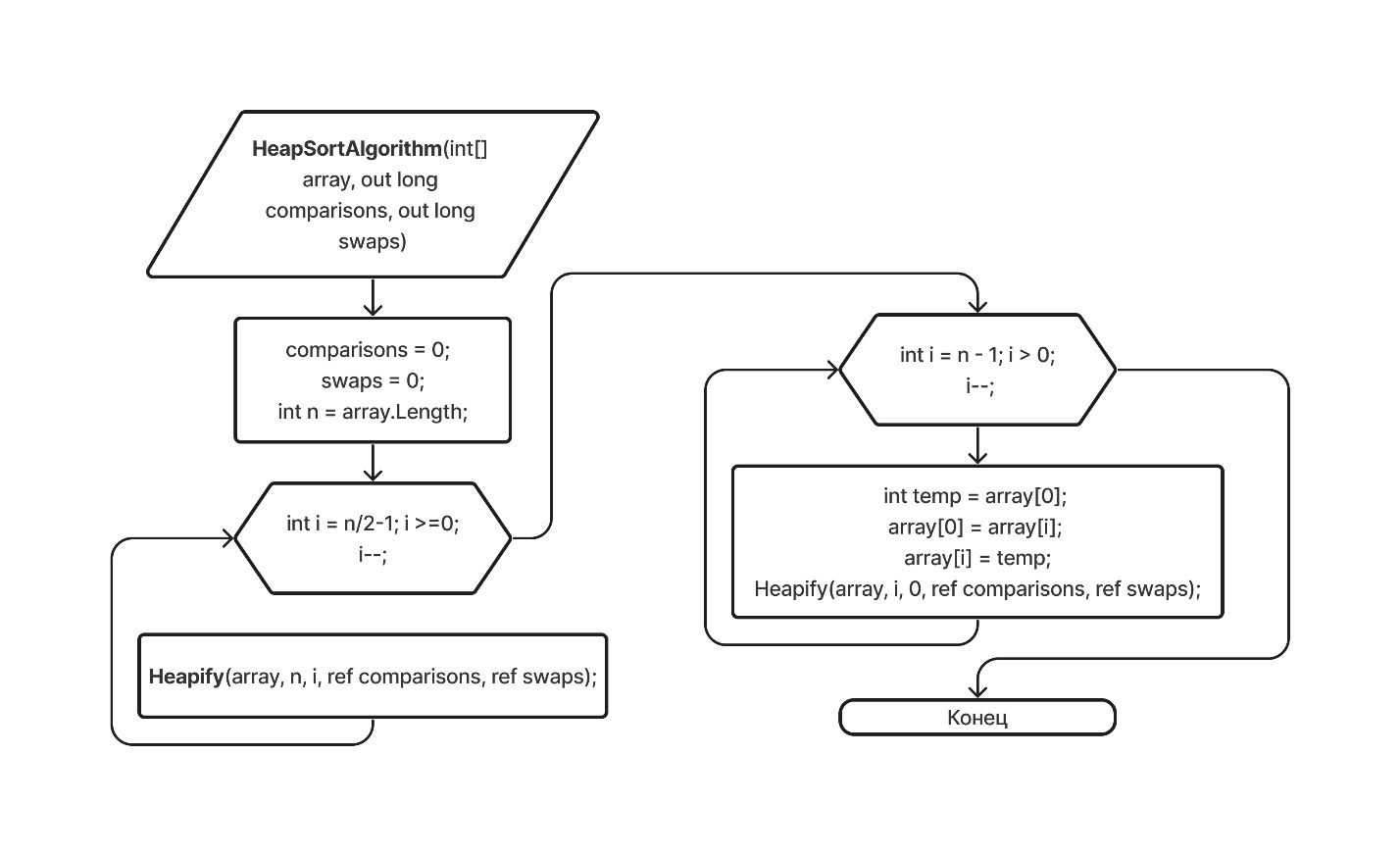
Последовательно извлекаем элементы из корня кучи и помещаем их в конец массива, меняя местами корень с последним элементом и затем уменьшая размер кучи.

После каждого извлечения элемента снова вызываем процедуру Heapify для поддержания свойства пирамиды.

1. Повторение:

Повторяем шаг 2 до тех пор, пока весь массив не будет отсортирован.

Алгоритм пирамидальной сортировки обладает временной сложностью O(n log n) в худшем, лучшем и среднем случае, что делает его одним из наиболее эффективных алгоритмов сортировки. Он также не требует дополнительной памяти, за исключением нескольких переменных, что делает его простым и эффективным в использовании. 



**Код программы:**

public static void HeapSortAlgorithm(int[] array, out long comparisons, out long swaps)

{

comparisons = 0; // Обнуляем счетчик операций сравнения

swaps = 0; // Обнуляем счетчик операций обмена

int n = array.Length;

// Построение кучи (heapify)

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

{

Heapify(array, n, i, ref comparisons, ref swaps);

}

// Последовательно извлекаем элементы из кучи

for (int i = n - 1; i > 0; i--)

{

// Перемещаем текущий корень в конец

int temp = array[0];

array[0] = array[i];

array[i] = temp;

// Вызываем процедуру heapify на уменьшенной куче

Heapify(array, i, 0, ref comparisons, ref swaps);

}

}

// Функция для поддержания свойства кучи в узле index в куче размером n

private static void Heapify(int[] array, int n, int index, ref long comparisons, ref long swaps)

{

int largest = index; // Инициализируем наибольший элемент как корень

int left = 2 \* index + 1; // Левый потомок узла index

int right = 2 \* index + 2; // Правый потомок узла index

// Увеличиваем счетчик операций сравнения

comparisons++;

// Если левый потомок больше родителя, то присваиваем largest ему

if (left < n && array[left] > array[largest])

{

largest = left;

}

// Увеличиваем счетчик операций сравнения

comparisons++;

// Если правый потомок больше родителя, то присваиваем largest ему

if (right < n && array[right] > array[largest])

{

largest = right;

}

// Увеличиваем счетчик операций сравнения

comparisons++;

// Если largest не равен index, то делаем обмен элементов и вызываем heapify рекурсивно

if (largest != index)

{

// Увеличиваем счетчик операций обмена

swaps++;

int temp = array[index];

array[index] = array[largest];

array[largest] = temp;

// Вызываем heapify рекурсивно для поддержания свойства кучи в largest

Heapify(array, n, largest, ref comparisons, ref swaps);

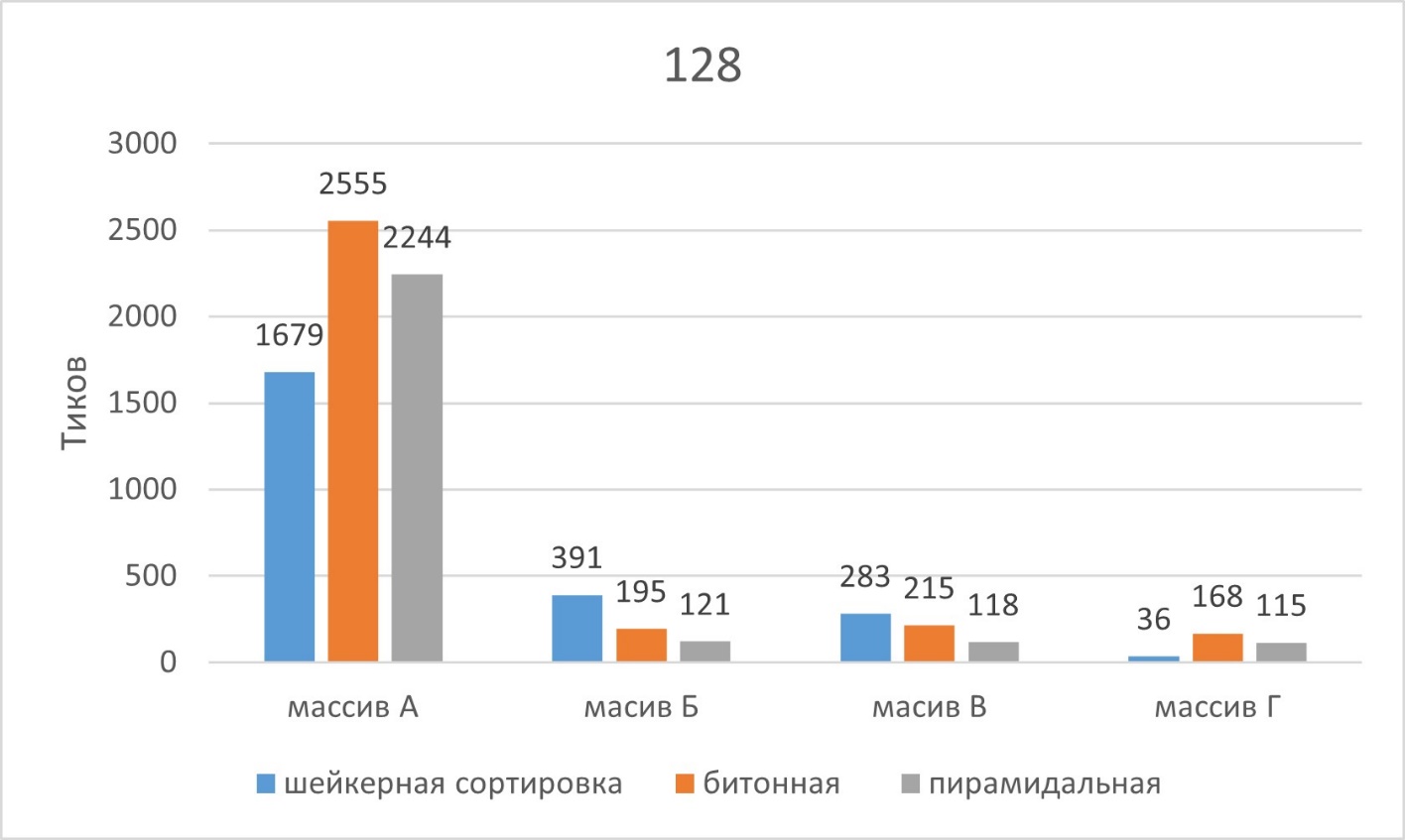
}

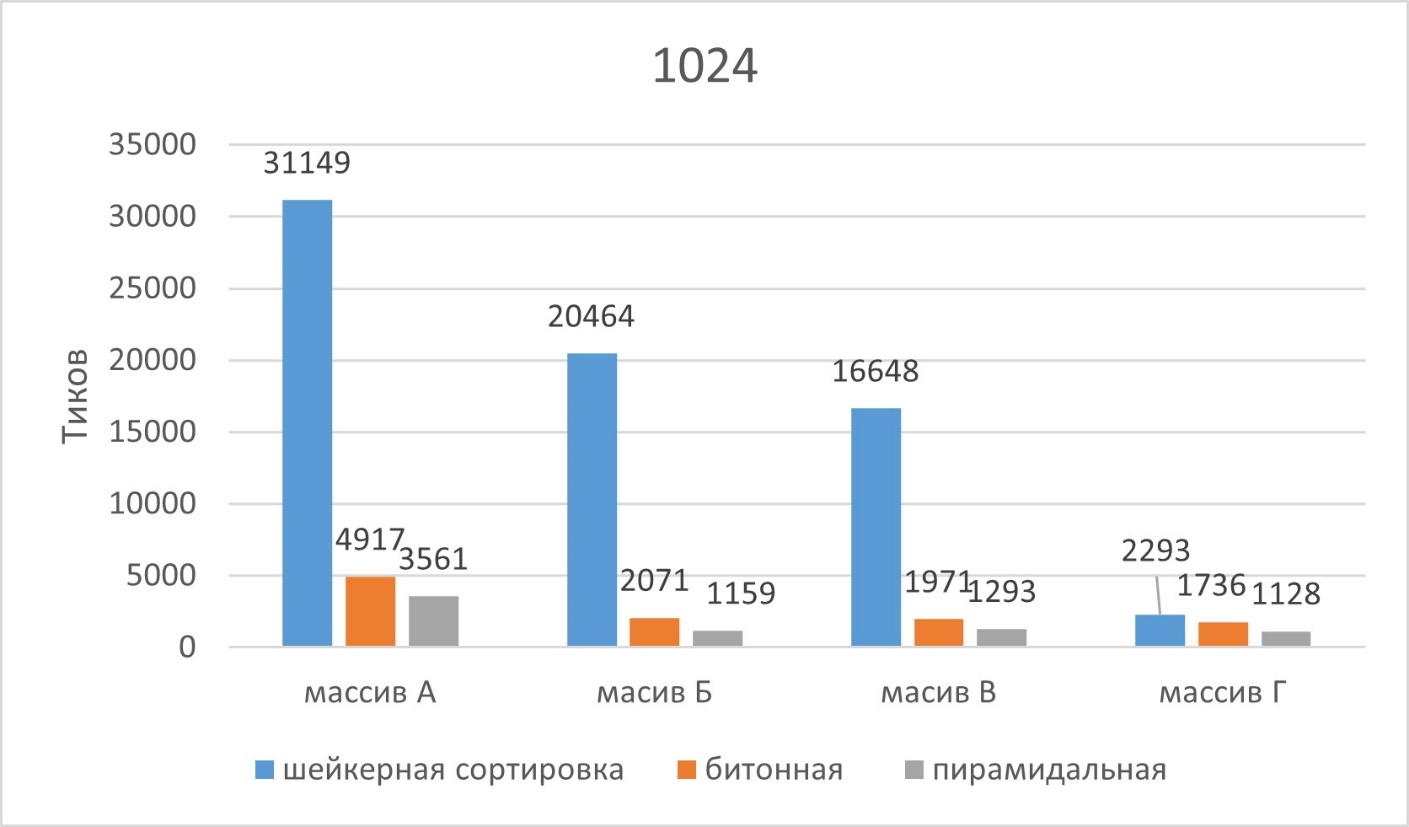
}

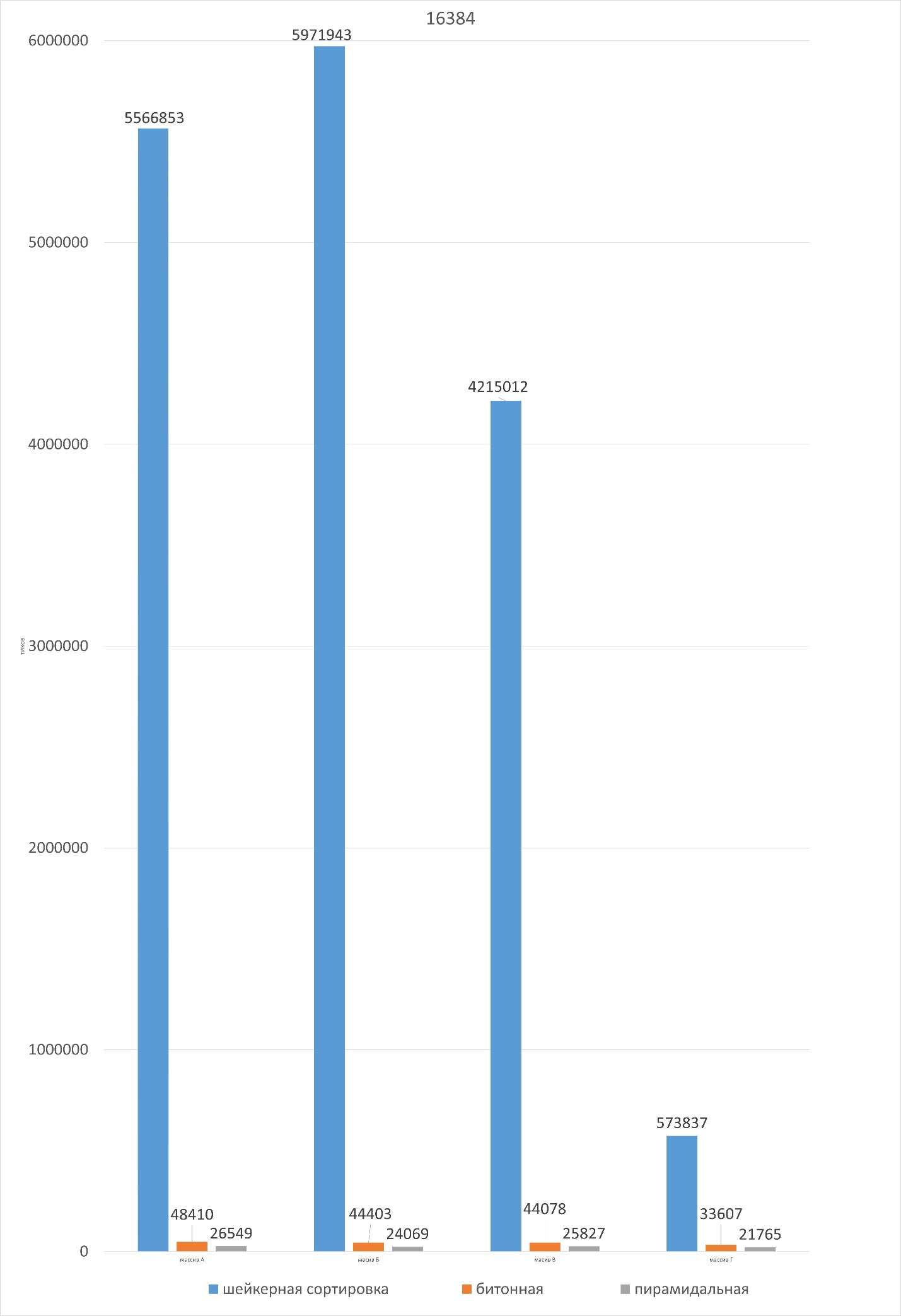
**Таблица измерений:**

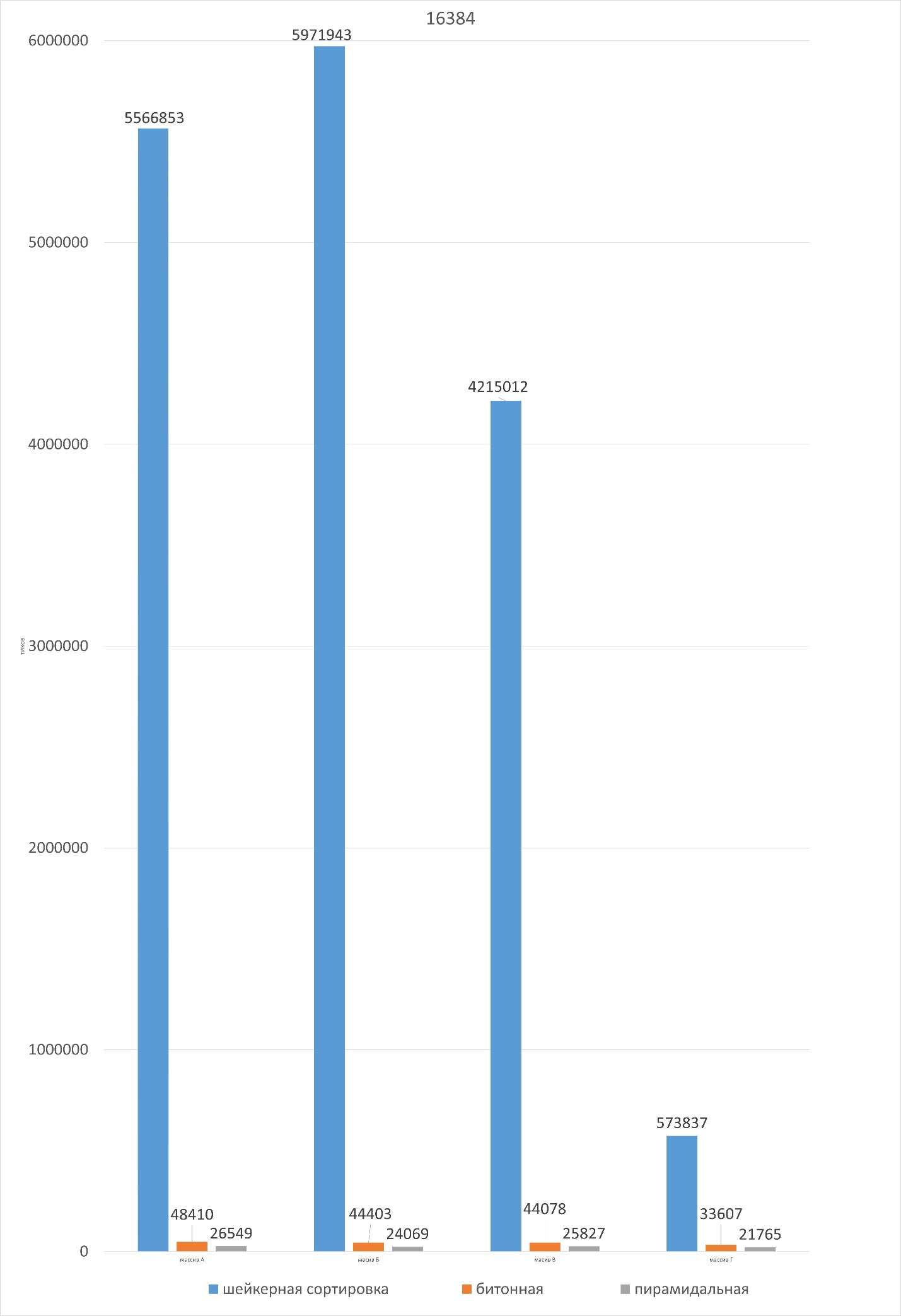
|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| **n** | **Параметр** | **Сгенерированные массивы** | | | | **Среднее значение** |
| **Массив а** | **Массив б** | **Массив в** | **Массив г** |
| 128 | Compare | 2520 | 2508 | 2529 | 2658 | 2553,75 |
|  | Swap | 649 | 645 | 652 | 695 | 660,25 |
|  | Time | 2244 | 121 | 118 | 115 | 649,5 |
| 1024 | Compare | 29358 | 28242 | 29373 | 28587 | 28890 |
|  | Swap | 8251 | 7879 | 8256 | 7994 | 8095 |
|  | Time | 3561 | 1159 | 1293 | 1128 | 1785,25 |
| 16384 | Compare | 664722 | 647757 | 663618 | 641661 | 654439,5 |
|  | Swap | 196999 | 191344 | 196631 | 189312 | 193571,5 |
|  | Time | 26549 | 24069 | 25827 | 21765 | 24552,5 |

**Вывод:**

****

****

****

****

**Сравнение операций сравнения и обмена:**

1)В целом, шейкерная сортировка и пирамидальная сортировка демонстрируют схожие значения среднего количества операций сравнения и обмена.

2)Битонная сортировка требует меньше операций сравнения и обмена на небольших массивах (128 и 1024 элемента), но наибольшее количество операций на массиве длиной 16384.

**Скорость выполнения:**

1)Время выполнения шейкерной сортировки значительно превышает время выполнения других двух сортировок, особенно на массивах большого размера (16384 элемента).

2)Пирамидальная сортировка обеспечивает более конкурентоспособное время выполнения на массивах среднего размера (1024 элемента), хотя на больших массивах ее производительность становится сравнимой со шейкерной сортировкой.

**Общие выводы:**

1)Шейкерная сортировка обладает наихудшей производительностью среди рассмотренных алгоритмов на всех размерах массивов.

2)Пирамидальная сортировка демонстрирует хорошие результаты на массивах среднего размера, но на больших массивах проигрывает битонной сортировке.

3)Битонная сортировка выгодна на небольших массивах, но при увеличении размера массива требует больше времени и ресурсов.

4)Для массивов небольшого размера можно использовать шейкерную сортировку или битонную сортировку в зависимости от специфики задачи.

5)При работе с массивами среднего размера рекомендуется использовать пирамидальную сортировку для обеспечения приемлемой производительности.

6)Для массивов большего размера наиболее эффективной оказывается битонная сортировка, но стоит учитывать, что на нее может быть наложено ограничение на допустимый размер массива.