|  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| |  |  |  | | --- | --- | --- | |  |  |  | | МИНОБРНАУКИ РОССИИ | | | | Федеральное государственное бюджетное образовательное учреждение  высшего образования  **«МИРЭА – Российский технологический университет»**  **РТУ МИРЭА** | | |   Институт Информационных технологий | |
|  | |
| Кафедра Математического обеспечения и стандартизации информационных технологий | |
|  | |
|  | |

|  |  |
| --- | --- |
| **ОТЧЕТ ПО ПРАКТИЧЕСКОЙ РАБОТЕ № 8** | |
| **по дисциплине** | |
| **«**Структуры и алгоритмы обработки данных**»**  **Тема: «Эффективные алгоритмы сортировки»** | |
|  | |
| Выполнил студент группы ИКБО-09-22 | Гришин А. В. |
| Принял преподаватель | Филатов А. С. |

|  |  |  |
| --- | --- | --- |
| Лабораторная работа выполнена | «\_\_»\_\_\_\_\_\_\_202\_\_ г. | *(подпись студента)* |
|  |  |  |
| «Зачтено» | «\_\_»\_\_\_\_\_\_\_202\_\_ г. | *(подпись руководителя)* |

Москва 2023

# **Цель работы**

Получить навыки по анализу вычислительной сложности нескольких алгоритмов сортировки и определение наиболее эффективного алгоритма.

1. **Постановка задачи**

Разработать три алгоритма сортировки, определенные вариантом. Провести анализ вычислительной и емкостной сложности алгоритма на массивах, заполненных случайно. Определить наиболее эффективный алгоритм.

1. Разработать алгоритм простой сортировки, определенной вариантом. Определить емкостную и временную сложность алгоритма.
2. Разработать алгоритм усовершенствованной сортировки, определенной вариантом. Определить емкостную и временную сложность алгоритма.
3. Разработать алгоритм сортировки методом простого слияния. Определить емкостную и временную сложность алгоритма.
4. Провести контрольные прогоны функций на уже отсортированных массивах, отсортированных в обратном порядке и сгенерированных с помощью генератора псевдослучайных чисел различных размеров (N>999). Рабочие прогоны функций должны проводиться на одинаковых массивах. Во время сортировки вычислять время её выполнения t. Провести эмпирическую (практическую) оценку вычислительной сложности алгоритмов для трех случаев, для чего предусмотреть в программе подсчет фактического количества операций сравнения С и количества операций перемещения М. Полученные результаты свести в сводные таблицы.
5. Представить графики зависимости С+М от N и t от N для анализируемых алгоритмов в трех случаях.
6. Провести анализ зависимости алгоритмов сортировок от размера и исходной упорядоченности массива по составленным таблицам. Определить эффективный алгоритм для каждого случая.

|  |  |  |  |
| --- | --- | --- | --- |
| **Вариант** | **Алгоритм простой сортировки** | **Алгоритм усовершенствованной сортировки** | **Алгоритм**  **слияния** |
| 7 | Простого выбора | Пирамидальная сортировка | Простое слияние |

# **. Решение**

* 1. **Краткая теория по теме**

Сортировка пузырьком с условием Айверсона — это алгоритм сортировки, который использует дополнительную проверку на выполнение каких-либо перестановок на текущей итерации. Если на текущей итерации не было произведено ни одной перестановки, то массив уже отсортирован и алгоритм завершается. Таким образом, сортировка пузырьком с условием Айверсона может быть более эффективной, чем обычная сортировка пузырьком, особенно для больших массивов данных, где количество обменов может быть значительным. Однако, несмотря на это улучшение, сортировка пузырьком с условием Айверсона все еще имеет квадратичную сложность времени в худшем случае, что может быть неэффективно для очень больших массивов данных. Поэтому для более эффективной сортировки часто используются другие алгоритмы, такие как сортировка слиянием, речь о которой пойдет далее.

Шейкерная сортировка с условием Айверсона — это улучшенный алгоритм сортировки пузырьком, который работает по принципу двунаправленной сортировки. Алгоритм начинает сравнивать первый и последний элементы массива и меняет их местами, если они стоят в неправильном порядке. Затем он продвигается к следующим элементам массива, сравнивая их и меняя местами, если это необходимо. Этот процесс продолжается до тех пор, пока не будет достигнут конец массива. Затем алгоритм повторяет этот процесс в обратном направлении, начиная с последнего элемента и двигаясь к первому. Это позволяет исправить любые ошибки, которые могли возникнуть в предыдущих итерациях.

Простое слияние — это алгоритм сортировки, который работает на основе принципа разделяй и властвуй. Он разбивает исходный массив на две половины, затем рекурсивно сортирует каждую половину и затем объединяет их в отсортированный массив. Процесс слияния начинается с создания нового массива, который будет содержать отсортированные элементы из двух половин исходного массива. Затем алгоритм сравнивает первый элемент каждой половины и помещает наименьший из них в новый массив. Этот процесс повторяется для каждой пары элементов до тех пор, пока не будет достигнут конец одной из половин. Если одна из половин закончилась, а другая еще содержит элементы, то оставшиеся элементы копируются в новый массив. В результате получается отсортированный массив, который содержит все элементы из исходного массива. Простое слияние имеет лучшую сложность времени, чем сортировка пузырьком или шейкерная, и может быть эффективным для больших массивов данных.

* 1. **. Дополнительные функции.**

Введем дополнительные функции Reverse - "переворачивает" массив для выполнения сортировки в худшем случае, и CopyArray – копирует значения из массива-буфера в основной массив.

|  |
| --- |
| void Reverse(int arr[], int size)  {  for (int i = 0; i < size / 2; i++) {  int temp = arr[i];  arr[i] = arr[size - i - 1];  arr[size - i - 1] = temp;  }  }  void CopyArray(int\* arr, int\* temp, int size)  {  for (int i = 0; i < size; i++)  {  arr[i] = temp[i];  }  } |

* 1. **. Выполнение первого задания****.**

Для выполнения первого задания необходимо разработать функцию, реализующую алгоритм сортировки пузырьком с условием Айверсона, и определить емкостную и временную сложность алгоритма.

|  |
| --- |
| void BubbleSort(int arr[], int size, long long& comps, long long& moves)  {  bool swapped = true;  int j = 0;  while (swapped)  {  swapped = false;  j++;  for (int i = 0; i < size - j; i++)  {  if (arr[i] > arr[i + 1])  {  swap(arr[i], arr[i + 1]);  swapped = true;  moves++;  }  comps++;  }  // Если не было перестановок на этом проходе, то сортировка завершается  if (!swapped)  {  break;  }  }  } |

Данная функция проходит по массиву и сравнивает каждую пару соседних элементов, меняя их местами, если они находятся в неправильном порядке. Этот процесс повторяется до тех пор, пока не будет достигнут конечный отсортированный массив. Переменная swapped отслеживает, были ли произведены перестановки на текущем проходе, чтобы определить, нужно ли продолжать сортировку или можно завершить ее. Переменная moves отслеживает количество перемещений элементов в процессе сортировки, а переменная comps - количество сравнений. Определим емкостную и временную сложность алгоритма.

Таблица 1. Расчет ёмкостной сложности алгоритма

|  |  |
| --- | --- |
| swapped = true; | 1 |
| j = 0; | 1 |
| while (swapped) | n |
| swapped = false; | 1 |
| j++; | 1 |
| for (int i = 0; i < size - j; i++) | n |
| if (arr[i] > arr[i + 1]) | n |
| swap(arr[i], arr[i + 1]); | n |
| swapped = true; | n |
| moves++; | n |
| comps++; | n |
| if (!swapped) | 1 |
| break; | 1 |
| Емкостная сложность: | 7n+6 |
| Лучший случай: | O(n) |
| Средний случай: | O(n^2) |
| Худший случай: | O(n^2) |

* 1. **Выполнение второго задания.**

Для выполнения второго задания необходимо разработать функцию, реализующую алгоритм шейкерной сортировки с условием Айверсона, и определить емкостную и временную сложность алгоритма.

|  |
| --- |
| void ShakerSort(int arr[], int size, long long& comps, long long& moves)  {  int left = 0, right = size - 1, k = 0;  bool swapped = true;  while (left < right && swapped)  {  swapped = false;  for (int i = left; i < right; i++)  {  if (arr[i] > arr[i + 1])  {  swap(arr[i], arr[i + 1]);  k = i;  swapped = true;  moves++;  }  comps++;  }  right = k;  for (int i = right; i > left; i--)  {  if (arr[i] < arr[i - 1])  {  swap(arr[i], arr[i - 1]);  k = i;  swapped = true;  moves++;  }  comps++;  }  left = k;  }  } |

Изначально устанавливаются границы сортируемой части массива: левая граница равна 0, правая граница равна размеру массива минус 1. Пока левая граница меньше правой и происходили перестановки элементов на предыдущей итерации, выполняется следующее:

1. swapped устанавливается в значение false.

2. Цикл проходится от левой границы до правой границы. Если текущий элемент больше следующего, то они меняются местами, устанавливается индекс последней перестановки k, и swapped устанавливается в true. Также увеличивается счетчик перемещений moves.

3. После прохода цикла, правая граница устанавливается в значение k.

4. Цикл проходится от правой границы до левой границы. Если текущий элемент меньше предыдущего, то они меняются местами, устанавливается индекс последней перестановки k, и swapped устанавливается в true. Счетчик перемещений moves увеличивается.

5. После прохода цикла, левая граница устанавливается в значение k.

Шаги 1-5 повторяются до тех пор, пока не будет выполнено условие выхода из цикла. Определим емкостную и временную сложность алгоритма.

Таблица 2. Расчет ёмкостной сложности алгоритма

|  |  |
| --- | --- |
| left = 0, right = size - 1, k = 0; | 1 |
| swapped = true | 1 |
| while (left < right && swapped) | n |
| swapped = false; | 1 |
| j++; | 1 |
| for (int i = left; i < right; i++) | n |
| if (arr[i] > arr[i + 1]) | n |
| swap(arr[i], arr[i + 1]); | n |
| k = i; | n |
| swapped = true; | n |
| moves++; | n |
| comps++; | n |
| right = k; | 1 |
| for (int i = right; i > left; i--) | n |
| if (arr[i] < arr[i - 1]) | n |
| swap(arr[i], arr[i - 1]); | n |
| k = i; | n |
| swapped = true; | n |
| moves++; | n |
| comps++; | n |
| left = k; | 1 |
| Емкостная сложность: | 15n+6 |
| Лучший случай: | O(n) |
| Средний случай: | O(n^2) |
| Худший случай: | O(n^2) |

* 1. **Выполнение третьего задания.**

Для выполнения последнего задания необходимо разработать функцию, реализующую сортировку методом простого слияния, и определить емкостную и временную сложность алгоритма. Однако данная сортировка состоит из двух функций - сортировки и слияния.

|  |
| --- |
| void Merge(int arr[], int left, int medium, int right, long long& comps, long long& moves)  {  int leftHalfSize = medium - left + 1;  int rightHalfSize = right - medium;  int\* leftHalf = new int[leftHalfSize];  int\* rightHalf = new int[rightHalfSize];  for (int i = 0; i < leftHalfSize; i++)  {  leftHalf[i] = arr[left + i];  }  for (int j = 0; j < rightHalfSize; j++)  {  rightHalf[j] = arr[medium + 1 + j];  }  int i = 0, j = 0, k = left;  // Перестановка элементов  while (i < leftHalfSize && j < rightHalfSize)  {  if (leftHalf[i] <= rightHalf[j])  {  arr[k++] = leftHalf[i++];  }  else  {  arr[k++] = rightHalf[j++];  }  moves++;  comps++;  }  // Выгрузка оставшихся элемнтов  while (i < leftHalfSize)  {  arr[k++] = leftHalf[i++];  moves++;  }  while (j < rightHalfSize)  {  arr[k++] = rightHalf[j++];  moves++;  }  }  void MergeSort(int arr[], int left, int right, long long& comps, long long& moves)  {  if (left < right)  {  int medium = left + (right - left) / 2;  MergeSort(arr, left, medium, comps, moves);  MergeSort(arr, medium + 1, right, comps, moves);  Merge(arr, left, medium, right, comps, moves);  }  } |

Функция Merge реализует операцию слияния двух отсортированных половин массива arr: от left до medium и от medium+1 до right. В результате работы данной функции элементы в указанном диапазоне будут отсортированы по возрастанию, а переменные comps и moves будут содержать количество сравнений и перемещений элементов соответственно.

Функция MergeSort реализует алгоритм сортировки слиянием в диапазоне от left до right. Алгоритм работает рекурсивно, разбивая массив на половины и сортируя каждую из них отдельно. Затем выполняется операция слияния двух отсортированных половин с помощью функции Merge, которая была описана выше. При каждом вызове функции MergeSort переменные comps и moves увеличиваются на количество сравнений и перемещений элементов соответственно, произведенных в процессе сортировки. Определим емкостную и временную сложность алгоритма.

Таблица 3. Расчет ёмкостной сложности алгоритма

|  |  |
| --- | --- |
| leftHalfSize = medium - left + 1; | 1 |
| rightHalfSize = right - medium; | 1 |
| leftHalf = new int[leftHalfSize]; | 1 |
| rightHalf = new int[rightHalfSize]; | 1 |
| for (int i = 0; i < leftHalfSize; i++) | n |
| leftHalf[i] = arr[left + i]; | n |
| for (int j = 0; j < rightHalfSize; j++) | n |
| rightHalf[j] = arr[medium + 1 + j]; | n |
| i = 0, j = 0, k = left; | 1 |
| while (i < leftHalfSize && j < rightHalfSize) | n |
| if (leftHalf[i] <= rightHalf[j]) | n |
| arr[k++] = leftHalf[i++]; | n |
| arr[k++] = rightHalf[j++]; | n |
| moves++; | n |
| comps++; | n |
| while (i < leftHalfSize) | n |
| arr[k++] = leftHalf[i++]; | n |
| moves++; | n |
| while (j < rightHalfSize) | n |
| arr[k++] = rightHalf[j++]; | n |
| moves++; | n |
| Емкостная сложность: | 16n+5 |
| Лучший случай: | O(nlogn) |
| Средний случай: | O(nlogn) |
| Худший случай: | O(nlogn) |

* 1. **Описание работы программы и её интерфейса**

В самом начале программы программа вывод количество элементов в массиве и результаты сортировки (рис. 1) – и так для каждой сортировки.

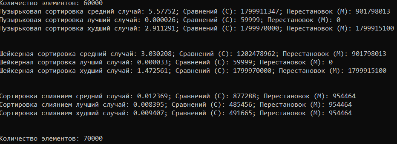


Рисунок 1. Интерфейс работы программы

1. **Тестирование**

Протестируем работу программы на одинаковых массивах случайных чисел размером с 60000 по 100000 с помощью трех сортировок (рис. 2, 3).

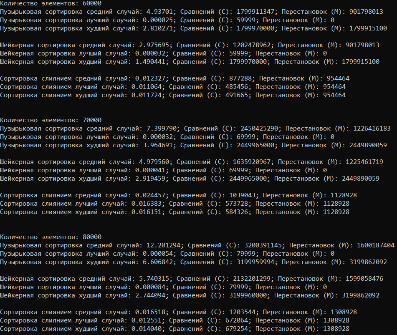


Рисунок 2. Тестирование работы трех алгоритмов  
на массивах размером с 60000 по 80000

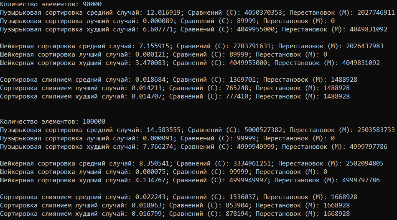


Рисунок 3. Тестирование работы трех алгоритмов  
на массивах размером 90000 и 100000

Программа работает корректно. Выпишем в таблицы 4-12 время выполнения, количество операций сравнения и перемещения элементов в массиве случайных чисел, а также представим графики (рис. 4-9) зависимости С+М от N и t от N для анализируемых алгоритмов в трех случаях.

Таблица 4. Таблица результатов тестирования пузырьковой сортировки с условием Айверсона для среднего случая

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **N** | **t, сек** | **Tа(n)=N2** | **Tэ=С+M** | **Tэ/Tа** |
| 60000 | 4.93701 | 3600000000 | 2701709360 | 0,7504748 |
| 70000 | 7.399790 | 4900000000 | 3676841473 | 0,7503758 |
| 80000 | 12.201294 | 6400000000 | 4800578549 | 0,7500904 |
| 90000 | 12.016919 | 8100000000 | 6078117264 | 0,7503848 |
| 100000 | 14.583555 | 10000000000 | 7504111115 | 0,7504111 |

Таблица 5. Таблица результатов тестирования пузырьковой сортировки с условием Айверсона для лучшего случая

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **N** | **t, сек** | **Tа(n)=N** | **Tэ=С+M** | **Tэ/Tа** |
| 60000 | 0.000025 | 60000 | 59999 | 0,999983 |
| 70000 | 0.000032 | 70000 | 69999 | 0,999986 |
| 80000 | 0.000054 | 80000 | 79999 | 0,999988 |
| 90000 | 0.000089 | 90000 | 89999 | 0,999989 |
| 100000 | 0.000091 | 100000 | 99999 | 0,99999 |

Таблица 6. Таблица результатов тестирования пузырьковой сортировки с условием Айверсона для худшего случая

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **N** | **t, сек** | **Tа(n)=N2** | **Tэ=С+M** | **Tэ/Tа** |
| 60000 | 2.810271 | 3600000000 | 3599885100 | 0,9999681 |
| 70000 | 3.964691 | 4900000000 | 4899855059 | 0,9999704 |
| 80000 | 6.606842 | 6400000000 | 6399822086 | 0,9999722 |
| 90000 | 6.607771 | 8100000000 | 8099786092 | 0,9999736 |
| 100000 | 7.766274 | 10000000000 | 9999747785 | 0,9999748 |

Таблица 7. Таблица результатов тестирования шейкерной сортировки с условием Айверсона для среднего случая

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **N** | **t, сек** | **Tа(n)=N2** | **Tэ=С+M** | **Tэ/Tа** |
| 60000 | 2.975695 | 3600000000 | 2104276975 | 0,5845214 |
| 70000 | 4.979560 | 4900000000 | 2861382686 | 0,5839557 |
| 80000 | 5.740315 | 6400000000 | 3731259775 | 0,5830093 |
| 90000 | 7.155915 | 8100000000 | 4730229614 | 0,5839790 |
| 100000 | 8.350541 | 10000000000 | 5837056056 | 0,5837056 |

Таблица 8. Таблица результатов тестирования шейкерной сортировки с условием Айверсона для лучшего случая

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **N** | **t, сек** | **Tа(n)=N** | **Tэ=С+M** | **Tэ/Tа** |
| 60000 | 0.000032 | 60000 | 59999 | 0,999983 |
| 70000 | 0.000041 | 70000 | 69999 | 0,999986 |
| 80000 | 0.000084 | 80000 | 79999 | 0,999988 |
| 90000 | 0.000121 | 90000 | 89999 | 0,999989 |
| 100000 | 0.000075 | 100000 | 99999 | 0,99999 |

Таблица 9. Таблица результатов тестирования шейкерной сортировки с условием Айверсона для худшего случая

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **N** | **t, сек** | **Tа(n)=N2** | **Tэ=С+M** | **Tэ/Tа** |
| 60000 | 1.490441 | 3600000000 | 3599885100 | 0,9999681 |
| 70000 | 2.919459 | 4900000000 | 4899855059 | 0,9999704 |
| 80000 | 2.744094 | 6400000000 | 6399822092 | 0,9999722 |
| 90000 | 3.470083 | 8100000000 | 8099786092 | 0,9999736 |
| 100000 | 4.134767 | 10000000000 | 9999747783 | 0,9999748 |

Таблица 10. Таблица результатов тестирования сортировки простым слиянием для среднего случая

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **N** | **t, сек** | **Tа(n)=Nlog(N)** | **Tэ=С+M** | **Tэ/Tа** |
| 60000 | 0.012327 | 286689 | 1831752 | 6,389335 |
| 70000 | 0.024457 | 339157 | 2167971 | 6,392234 |
| 80000 | 0.016518 | 392247 | 2512472 | 6,405331 |
| 90000 | 0.018684 | 445882 | 2858629 | 6,411178 |
| 100000 | 0.022243 | 500000 | 3204965 | 6,40993 |

Таблица 11. Таблица результатов тестирования сортировки простым слиянием для лучшего случая

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **N** | **t, сек** | **Tа(n)= Nlog(N)** | **Tэ=С+M** | **Tэ/Tа** |
| 60000 | 0.011064 | 286689 | 1439920 | 5,022585 |
| 70000 | 0.016383 | 339157 | 1702656 | 5,020259 |
| 80000 | 0.012551 | 392247 | 1980992 | 5,050369 |
| 90000 | 0.014213 | 445882 | 2254176 | 5,055544 |
| 100000 | 0.018963 | 500000 | 2522832 | 5,045664 |

Таблица 12. Таблица результатов тестирования сортировки простым слиянием для худшего случая

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **N** | **t, сек** | **Tа(n)= Nlog(N)** | **Tэ=С+M** | **Tэ/Tа** |
| 60000 | 0.011724 | 286689 | 1446129 | 5,044243 |
| 70000 | 0.016151 | 339157 | 1713254 | 5,051507 |
| 80000 | 0.014040 | 392247 | 1988182 | 5,068699 |
| 90000 | 0.014707 | 445882 | 2266338 | 5,08282 |
| 100000 | 0.016799 | 500000 | 2547122 | 5,094244 |

Рисунок 4. Зависимость C+M от N для среднего случая

Рисунок 5. Зависимость C+M от N для лучшего случая

Рисунок 6. Зависимость C+M от N для худшего случая

Рисунок 7. Зависимость t от N для среднего случая

Рисунок 8. Зависимость t от N для лучшего случая

Рисунок 9. Зависимость t от N для худшего случая

Как можно заметить пузырьковая и шейкерная имеют схожие результаты. Для лучшего случая пузырьковая и шейкерная превосходят сортировку слиянием по количеству сравнений/перемещений и затраченного времени. Однако в остальных случаях сортировка простым слиянием имеет лучшие показатели как по количеству сравнений/перемещений, так и по затраченному времени, поэтому можно сказать, что сортировка слиянием для больших массивов данных эффективнее, чем пузырьковая и шейкерная.

1. **Вывод**

В результате выполнения работы я получил навыки по анализу вычислительной сложности нескольких алгоритмов сортировки и определение наиболее эффективного алгоритма.

1. **Исходный код программы**

Исходный код файла main.cpp

|  |
| --- |
| #include <iostream>  #include <chrono>  using namespace std;  using namespace chrono;  void BubbleSort(int arr[], int size, long long& comps, long long& moves)  {  bool swapped = true;  int j = 0;  while (swapped)  {  swapped = false;  j++;  for (int i = 0; i < size - j; i++)  {  if (arr[i] > arr[i + 1])  {  swap(arr[i], arr[i + 1]);  swapped = true;  moves++;  }  comps++;  }  // Если не было перестановок на этом проходе, то сортировка завершается  if (!swapped)  {  break;  }  }  }  void ShakerSort(int arr[], int size, long long& comps, long long& moves)  {  int left = 0, right = size - 1, k = 0;  bool swapped = true;  while (left < right && swapped)  {  swapped = false;  for (int i = left; i < right; i++)  {  if (arr[i] > arr[i + 1])  {  swap(arr[i], arr[i + 1]);  k = i;  swapped = true;  moves++;  }  comps++;  }  right = k;  for (int i = right; i > left; i--)  {  if (arr[i] < arr[i - 1])  {  swap(arr[i], arr[i - 1]);  k = i;  swapped = true;  moves++;  }  comps++;  }  left = k;  }  }  void Merge(int arr[], int left, int medium, int right, long long& comps, long long& moves)  {  int leftHalfSize = medium - left + 1;  int rightHalfSize = right - medium;  int\* leftHalf = new int[leftHalfSize];  int\* rightHalf = new int[rightHalfSize];  for (int i = 0; i < leftHalfSize; i++)  {  leftHalf[i] = arr[left + i];  }  for (int j = 0; j < rightHalfSize; j++)  {  rightHalf[j] = arr[medium + 1 + j];  }  int i = 0, j = 0, k = left;  // Перестановка элементов  while (i < leftHalfSize && j < rightHalfSize)  {  if (leftHalf[i] <= rightHalf[j])  {  arr[k++] = leftHalf[i++];  }  else  {  arr[k++] = rightHalf[j++];  }  moves++;  comps++;  }  // Выгрузка оставшихся элемнтов  while (i < leftHalfSize)  {  arr[k++] = leftHalf[i++];  moves++;  }  while (j < rightHalfSize)  {  arr[k++] = rightHalf[j++];  moves++;  }  }  void MergeSort(int arr[], int left, int right, long long& comps, long long& moves)  {  if (left < right)  {  int medium = left + (right - left) / 2;  MergeSort(arr, left, medium, comps, moves);  MergeSort(arr, medium + 1, right, comps, moves);  Merge(arr, left, medium, right, comps, moves);  }  }  void Reverse(int arr[], int size)  {  for (int i = 0; i < size / 2; i++) {  int temp = arr[i];  arr[i] = arr[size - i - 1];  arr[size - i - 1] = temp;  }  }  void CopyArray(int\* arr, int\* temp, int size)  {  for (int i = 0; i < size; i++)  {  arr[i] = temp[i];  }  }  int main()  {  setlocale(LC\_ALL, "rus");  long long comps = 0, moves = 0;  for (int size = 60000; size <= 100000; size += 10000)  {  int\* arr = new int[size];  int\* temp = new int[size];  for (int j = 0; j < size; j++)  {  arr[j] = rand();  temp[j] = arr[j];  }  cout << "Количество элементов: " << size << endl;  auto start = steady\_clock::now();  BubbleSort(arr, size, comps, moves);  auto end = steady\_clock::now();  auto difference = duration\_cast<duration<float>>(end - start).count();  cout << "Пузырьковая сортировка средний случай: " << difference << "; Сравнений (C): " << comps << "; Перестановок (M): " << moves << endl;  comps = 0, moves = 0;  start = steady\_clock::now();  BubbleSort(arr, size, comps, moves);  end = steady\_clock::now();  difference = duration\_cast<duration<float>>(end - start).count();  cout << "Пузырьковая сортировка лучший случай: " << fixed << difference << "; Сравнений (C): " << comps << "; Перестановок (M): " << moves << endl;  comps = 0, moves = 0;  Reverse(arr, size);  start = steady\_clock::now();  BubbleSort(arr, size, comps, moves);  end = steady\_clock::now();  difference = duration\_cast<duration<float>>(end - start).count();  cout << "Пузырьковая сортировка худший случай: " << difference << "; Сравнений (C): " << comps << "; Перестановок (M): " << moves << endl << endl;  CopyArray(arr, temp, size);  comps = 0, moves = 0;  start = steady\_clock::now();  ShakerSort(arr, size, comps, moves);  end = steady\_clock::now();  difference = duration\_cast<duration<float>>(end - start).count();  cout << "Шейкерная сортировка средний случай: " << difference << "; Сравнений (C): " << comps << "; Перестановок (M): " << moves << endl;  comps = 0, moves = 0;  start = steady\_clock::now();  ShakerSort(arr, size, comps, moves);  end = steady\_clock::now();  difference = duration\_cast<duration<float>>(end - start).count();  cout << "Шейкерная сортировка лучший случай: " << fixed << difference << "; Сравнений (C): " << comps << "; Перестановок (M): " << moves << endl;  comps = 0, moves = 0;  Reverse(arr, size);  start = steady\_clock::now();  ShakerSort(arr, size, comps, moves);  end = steady\_clock::now();  difference = duration\_cast<duration<float>>(end - start).count();  cout << "Шейкерная сортировка худший случай: " << difference << "; Сравнений (C): " << comps << "; Перестановок (M): " << moves << endl << endl;  CopyArray(arr, temp, size);  comps = 0, moves = 0;  start = steady\_clock::now();  MergeSort(arr, 0, size - 1, comps, moves);  end = steady\_clock::now();  difference = duration\_cast<duration<float>>(end - start).count();  cout << "Сортировка слиянием средний случай: " << difference << "; Сравнений (C): " << comps << "; Перестановок (M): " << moves << endl;  comps = 0, moves = 0;  start = steady\_clock::now();  MergeSort(arr, 0, size - 1, comps, moves);  end = steady\_clock::now();  difference = duration\_cast<duration<float>>(end - start).count();;  cout << "Сортировка слиянием лучший случай: " << difference << "; Сравнений (C): " << comps << "; Перестановок (M): " << moves << endl;  comps = 0, moves = 0;  Reverse(arr, size);  start = steady\_clock::now();  MergeSort(arr, 0, size - 1, comps, moves);  end = steady\_clock::now();  difference = duration\_cast<duration<float>>(end - start).count();  cout << "Сортировка слиянием худший случай: " << difference << "; Сравнений (C): " << comps << "; Перестановок (M): " << moves << endl << endl;  cout << endl;  }  } |