|  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| |  |  |  | | --- | --- | --- | |  |  |  | | 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-08-22 | Сенькевич Г.Д. |
| 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.

# **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 №7. Условие задания:

|  |  |
| --- | --- |
| Упражнение 1 | Разработать алгоритм сортировки простого выбора |
| Упражнение 2 | Разработать алгоритм пирамидальной сортировки |
| Упражнение 3 | Разработать алгоритм сортировки слиянием |

# **Решение**

Сортировка — это процесс упорядочивания набора данных в определенном порядке. Она широко используется в программировании и информатике для обработки больших объемов информации и оптимизации производительности алгоритмов.

Существует множество алгоритмов сортировки, каждый из которых имеет свои уникальные особенности и применяется в различных ситуациях в зависимости от характеристик данных и требований к производительности. Некоторые из наиболее распространенных алгоритмов сортировки включают в себя сортировку пузырьком, сортировку вставками, сортировку выбором, быструю сортировку, сортировку слиянием и т.д.

Важными факторами при выборе алгоритма сортировки являются время выполнения, потребляемая память и стабильность сортировки. Время выполнения обычно измеряется в количестве операций сравнения и перемещения элементов данных, а потребляемая память зависит от количества элементов, которые необходимо отсортировать.

Сортировка выбором (selection sort) — это алгоритм сортировки, который работает путем нахождения наименьшего элемента в неотсортированной части массива и перемещения его в начало этой части. Этот процесс повторяется для каждого элемента неотсортированной части массива до тех пор, пока не будет отсортирована вся последовательность.

Алгоритм сортировки выбором начинается с первого элемента массива. Далее, находится наименьший элемент в неотсортированной части массива, затем меняется местами с первым элементом массива. Теперь первый элемент является отсортированным. Затем поиск наименьшего элемента начинается с второго элемента, и процесс повторяется, пока все элементы не будут отсортированы.

Пирамидальная сортировка (heap sort) — это алгоритм сортировки, который использует структуру данных пирамиду (heap) для поиска и выбора наименьшего (или наибольшего) элемента из неотсортированной части массива. Этот алгоритм основан на принципе сортировки внутренним выбором (selection sort), но вместо линейного поиска наименьшего элемента, используется более эффективный алгоритм нахождения минимального элемента в пирамиде.

Для пирамидальной сортировки сначала создается пирамида из неотсортированной части массива. Затем наименьший элемент (для сортировки по возрастанию) перемещается в начало массива, и он удаляется из пирамиды. Этот процесс повторяется для каждого элемента неотсортированной части массива, пока все элементы не будут отсортированы.

Пирамидальная сортировка может быть реализована как вариант сортировки восходящей пирамиды (min-heap), где каждый узел имеет меньшее значение, чем его потомки, или как вариант сортировки нисходящей пирамиды (max-heap), где каждый узел имеет большее значение, чем его потомки.

Сортировка слиянием (merge sort) — это алгоритм сортировки, который основан на принципе "разделяй и властвуй". Он разбивает неотсортированный массив на две половины, каждую из которых сортирует рекурсивно, а затем объединяет эти половины в отсортированный массив.

Алгоритм сортировки слиянием начинается с разделения неотсортированного массива на две равные половины. Затем каждая половина сортируется рекурсивно с помощью того же алгоритма сортировки слиянием. Этот процесс продолжается до тех пор, пока не будет достигнут массив из одного элемента, который считается отсортированным.

Затем отсортированные половины массива объединяются в один отсортированный массив. Для этого каждый элемент сравнивается с элементом на соответствующей позиции в другой половине массива, и наименьший элемент помещается в новый массив. Этот процесс продолжается до тех пор, пока все элементы не будут добавлены в новый массив.

Для решения первого упражнения была написана функция selection\_sort, выполняющая сортировку массива методом сортировки выбором. Функция принимает на вход целочисленный массив (std::vector<int>) и возвращает данные о количестве сравнений и перемещений элементов в ходе сортировки (экземпляр структуры sorting\_data).

Ёмкостная сложность: O(1).

Временная сложность: O(n2).

|  |
| --- |
| sorting\_data selection\_sort(std::vector<int>& vec)  {  sorting\_data data;  for (int i = 0; i < vec.size(); i++)  {  int min\_index = i;  for (int j = i + 1; j < vec.size(); j++)  {  data.comparisons++;  if (vec[j] < vec[min\_index])  {  min\_index = j;  }  }  if (min\_index != i)  {  data.moves++;  std::swap(vec[i], vec[min\_index]);  }  }  return data;  } |

Для решения второго упражнения была написаны функции heapify и heap\_sort, первая из которых нужна для построения кучи (heap), а вторая для выполнения сортировки массива методом пирамидальной сортировки. Функция принимает на вход целочисленный массив (std::vector<int>) и возвращает данные о количестве сравнений и перемещений элементов в ходе сортировки (экземпляр структуры sorting\_data).

Ёмкостная сложность: O(1).

Временная сложность: O(n \* log(n)).

|  |
| --- |
| sorting\_data heapify(std::vector<int>& vec, int head, int N = -1)  {  sorting\_data data;    if (N == -1)  {  N = vec.size();  }  int largest = head;  int left = 2 \* head + 1;  int right = 2 \* head + 2;  data.comparisons++;  if (left >= 0 && left < N && vec[left] > vec[largest])  {  largest = left;  }  data.comparisons++;  if (right >= 0 && right < N && vec[right] > vec[largest])  {  largest = right;  }  if (largest != head)  {  data.moves++;  std::swap(vec[head], vec[largest]);  sorting\_data returned = heapify(vec, largest, N);  return sorting\_data(data.comparisons + returned.comparisons,  data.moves + returned.moves);  }  return data;  }  sorting\_data heap\_sort(std::vector<int>& vec)  {  sorting\_data data;  for (int i = vec.size() / 2 - 1; i >= 0; i--)  {  sorting\_data returned = heapify(vec, i);  data.comparisons += returned.comparisons;  data.moves += returned.moves;  }  for (int i = vec.size() - 1; i > 0; i--)  {  data.moves++;  std::swap(vec[i], vec[0]);  sorting\_data returned = heapify(vec, 0, i);  data.comparisons += returned.comparisons;  data.moves += returned.moves;  }  return data;  } |

Для решения второго упражнения была написана функция merge\_sort, выполняющая сортировку массива методом сортировки слиянием. Функция принимает на вход целочисленный массив (std::vector<int>) и возвращает данные о количестве сравнений и перемещений элементов в ходе сортировки (экземпляр структуры sorting\_data).

Ёмкостная сложность: O(n).

Временная сложность: O(n \* log(n)).

|  |
| --- |
| sorting\_data merge\_sort\_with\_borders(std::vector<int>& vec, int left, int right)  {  sorting\_data data;  if (left <= -1 || left >= vec.size() ||  right <= -1 || right >= vec.size() ||  left == right)  {  return data;  }  int middle = left + (right - left) / 2;  sorting\_data returned\_left = merge\_sort\_with\_borders(vec, left, middle);  sorting\_data returned\_right = merge\_sort\_with\_borders(vec, middle + 1, right);  data.comparisons += returned\_left.comparisons + returned\_right.comparisons;  data.moves += returned\_left.moves + returned\_right.moves;  std::vector<int> tmp(right - left + 1);  int curr\_left = left;  int curr\_right = middle + 1;  for (int i = 0; i < tmp.size(); i++)  {  if (curr\_left <= middle && curr\_right <= right)  {  data.comparisons++;  if (vec[curr\_left] < vec[curr\_right])  {  tmp[i] = vec[curr\_left];  curr\_left++;  }  else  {  tmp[i] = vec[curr\_right];  curr\_right++;  }  }  else if (curr\_left <= middle)  {  tmp[i] = vec[curr\_left];  curr\_left++;  }  else if (curr\_right <= right)  {  tmp[i] = vec[curr\_right];  curr\_right++;  }  }  for (int i = left; i <= right; i++)  {  vec[i] = tmp[i - left];  }  data.moves += right - left + 1;  return data;  }  sorting\_data merge\_sort(std::vector<int>& vec)  {  return merge\_sort\_with\_borders(vec, 0, vec.size() - 1);  } |

При запуске программы пользователь видит сообщения о процессе тестирования сортировок на различных массивах, результаты тестирования сохраняются в текстовые файлы.

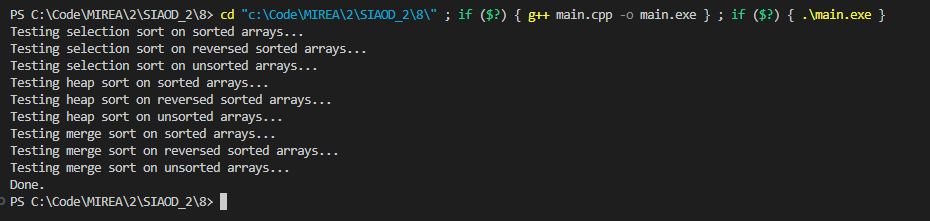


Рисунок 1. Интерфейс программы

# **Тестирование**

Таблица 1. Рабочий прогон сортировки выбором на отсортированных по возрастанию массивах

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **N** | **t, мс** | **Ta(n) = N2** | **Tэ=C+M** | **Tэ / Ta** |
| 10000 | 285 | 100000000 | 49995000 | 0,49995 |
| 20000 | 1208 | 400000000 | 2E+08 | 0,5 |
| 30000 | 3799 | 900000000 | 4,5E+08 | 0,5 |
| 40000 | 5814 | 1600000000 | 8E+08 | 0,5 |
| 50000 | 8291 | 2500000000 | 1,25E+09 | 0,5 |
| 60000 | 14678 | 3600000000 | 1,8E+09 | 0,5 |
| 70000 | 20289 | 4900000000 | 2,45E+09 | 0,5 |
| 80000 | 25534 | 6400000000 | 3,2E+09 | 0,5 |
| 90000 | 31614 | 8100000000 | 4,05E+09 | 0,5 |
| 100000 | 38984 | 10000000000 | 5E+09 | 0,5 |

Таблица 2. Рабочий прогон сортировки выбором на отсортированных по убыванию массивах

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **N** | **t, мс** | **Ta(n) = N2** | **Tэ=C+M** | **Tэ / Ta** |
| 10000 | 915 | 100000000 | 50000554 | 0,50000554 |
| 20000 | 2579 | 400000000 | 2E+08 | 0,5 |
| 30000 | 3570 | 900000000 | 4,5E+08 | 0,5 |
| 40000 | 10183 | 1600000000 | 8E+08 | 0,5 |
| 50000 | 14941 | 2500000000 | 1,25E+09 | 0,5 |
| 60000 | 22692 | 3600000000 | 1,8E+09 | 0,5 |
| 70000 | 28332 | 4900000000 | 2,45E+09 | 0,5 |
| 80000 | 36713 | 6400000000 | 3,2E+09 | 0,5 |
| 90000 | 45521 | 8100000000 | 4,05E+09 | 0,5 |
| 100000 | 55500 | 10000000000 | 5E+09 | 0,5 |

Таблица 3. Рабочий прогон сортировки выбором на неотсортированных массивах

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **N** | **t, мс** | **Ta(n) = N2** | **Tэ=C+M** | **Tэ / Ta** |
| 10000 | 282 | 100000000 | 50004991 | 0,50004991 |
| 20000 | 1676 | 400000000 | 2E+08 | 0,5 |
| 30000 | 4644 | 900000000 | 4,5E+08 | 0,5 |
| 40000 | 6534 | 1600000000 | 8E+08 | 0,5 |
| 50000 | 9852 | 2500000000 | 1,25E+09 | 0,5 |
| 60000 | 14272 | 3600000000 | 1,8E+09 | 0,5 |
| 70000 | 20070 | 4900000000 | 2,45E+09 | 0,5 |
| 80000 | 26419 | 6400000000 | 3,2E+09 | 0,5 |
| 90000 | 33284 | 8100000000 | 4,05E+09 | 0,5 |
| 100000 | 40866 | 10000000000 | 5E+09 | 0,5 |

Таблица 4. Рабочий прогон пирамидальной сортировки на отсортированных по возрастанию массивах

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **N** | **t, мс** | **Ta(n)=N\*log(N)** | **Tэ=C+M** | **Tэ / Ta** |
| 10000 | 4 | 92103,40 | 404062 | 4,39 |
| 20000 | 9 | 198069,75 | 868109 | 4,38 |
| 30000 | 14 | 309268,58 | 1351785 | 4,37 |
| 40000 | 18 | 423865,39 | 1857958 | 4,38 |
| 50000 | 24 | 540988,91 | 2372147 | 4,39 |
| 60000 | 28 | 660125,99 | 2886585 | 4,37 |
| 70000 | 33 | 780937,54 | 3413110 | 4,37 |
| 80000 | 39 | 903182,55 | 3957131 | 4,38 |
| 90000 | 44 | 1026680,85 | 4500543 | 4,38 |
| 100000 | 51 | 1151292,55 | 5044675 | 4,38 |

Таблица 5. Рабочий прогон пирамидальной сортировки на отсортированных по убыванию массивах

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **N** | **t, мс** | **Ta(n)=N\*log(N)** | **Tэ=C+M** | **Tэ / Ta** |
| 10000 | 3 | 92103,40 | 359944 | 3.91 |
| 20000 | 8 | 198069,75 | 782834 | 3.95 |
| 30000 | 11 | 309268,58 | 1227750 | 3,97 |
| 40000 | 16 | 423865,39 | 1682452 | 3.97 |
| 50000 | 22 | 540988,91 | 2146574 | 3.97 |
| 60000 | 26 | 660125,99 | 2624637 | 3.98 |
| 70000 | 31 | 780937,54 | 3108145 | 3.98 |
| 80000 | 36 | 903182,55 | 3601631 | 3.99 |
| 90000 | 42 | 1026680,85 | 4095291 | 3.99 |
| 100000 | 48 | 1151292,55 | 4593223 | 3.99 |

Таблица 6. Рабочий прогон пирамидальной сортировки на неотсортированных массивах

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **N** | **t, мс** | **Ta(n)=N\*log(N)** | **Tэ=C+M** | **Tэ / Ta** |
| 10000 | 4 | 92103,40 | 382540 | 4.15 |
| 20000 | 9 | 198069,75 | 824768 | 4.16 |
| 30000 | 14 | 309268,58 | 1289634 | 4.17 |
| 40000 | 19 | 423865,39 | 1770232 | 4.18 |
| 50000 | 24 | 540988,91 | 2263442 | 4.18 |
| 60000 | 31 | 660125,99 | 2758812 | 4.18 |
| 70000 | 35 | 780937,54 | 3264571 | 4.18 |
| 80000 | 41 | 903182,55 | 3780407 | 4.19 |
| 90000 | 46 | 1026680,85 | 4300737 | 4.19 |
| 100000 | 52 | 1151292,55 | 4824958 | 4.19 |

Таблица 7. Рабочий прогон сортировки слиянием на отсортированных по возрастанию массивах

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **N** | **t, мс** | **Ta(n)=N\*log(N)** | **Tэ=C+M** | **Tэ / Ta** |
| 10000 | 3 | 92103,40 | 203177 | 2.21 |
| 20000 | 7 | 198069,75 | 437324 | 2.21 |
| 30000 | 12 | 309268,58 | 680144 | 2.20 |
| 40000 | 15 | 423865,39 | 937624 | 2.21 |
| 50000 | 20 | 540988,91 | 1195176 | 2.21 |
| 60000 | 23 | 660125,99 | 1457508 | 2.21 |
| 70000 | 31 | 780937,54 | 1725335 | 2.21 |
| 80000 | 49 | 903182,55 | 2002587 | 2.22 |
| 90000 | 34 | 1026680,85 | 2277315 | 2.22 |
| 100000 | 39 | 1151292,55 | 2550989 | 2.22 |

Таблица 8. Рабочий прогон сортировки слиянием на отсортированных по убыванию массивах

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **N** | **t, мс** | **Ta(n)=N\*log(N)** | **Tэ=C+M** | **Tэ / Ta** |
| 10000 | 3 | 92103,40 | 198224 | 2.15 |
| 20000 | 7 | 198069,75 | 426448 | 2.15 |
| 30000 | 10 | 309268,58 | 666736 | 2.15 |
| 40000 | 14 | 423865,39 | 912896 | 2.15 |
| 50000 | 18 | 540988,91 | 1166976 | 2.16 |
| 60000 | 22 | 660125,99 | 1423472 | 2.16 |
| 70000 | 27 | 780937,54 | 1684128 | 2.16 |
| 80000 | 49 | 903182,55 | 1945792 | 2.15 |
| 90000 | 29 | 1026680,85 | 2212608 | 2.16 |
| 100000 | 43 | 1151292,55 | 2483952 | 2.16 |

Таблица 9. Рабочий прогон сортировки слиянием на неотсортированных массивах

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **N** | **t, мс** | **Ta(n)=N\*log(N)** | **Tэ=C+M** | **Tэ / Ta** |
| 10000 | 6 | 92103,40 | 254183 | 2.76 |
| 20000 | 9 | 198069,75 | 548044 | 2.77 |
| 30000 | 13 | 309268,58 | 855739 | 2.77 |
| 40000 | 17 | 423865,39 | 1176372 | 2.78 |
| 50000 | 23 | 540988,91 | 1502722 | 2.78 |
| 60000 | 29 | 660125,99 | 1831744 | 2.77 |
| 70000 | 35 | 780937,54 | 2167929 | 2.78 |
| 80000 | 38 | 903182,55 | 2512517 | 2.78 |
| 90000 | 46 | 1026680,85 | 2858336 | 2.78 |
| 100000 | 52 | 1151292,55 | 3205265 | 2.78 |

Рисунок 2. График (C + M) для таблиц 1, 4, 7 (прогоны на массивах, отсортированных по возрастанию)

Рисунок 3. График T для таблиц 1, 4, 7 (прогоны на массивах, отсортированных по возрастанию)

Рисунок 4. График (C + M) для таблиц 2, 5, 8 (прогоны на массивах, отсортированных по убыванию)

Рисунок 5. График T для таблиц 2, 5, 8 (прогоны на массивах, отсортированных по убыванию)

Рисунок 6. График (C + M) для таблиц 3, 6, 9 (прогоны на массивах, не отсортированных)

Рисунок 7. График T для таблиц 3, 6, 9 (прогоны на массивах, неотсортированных)

Из результатов выполнения программы видно, что сортировка слиянием сильно превосходит по сложности пирамидальную сортировку и сортировку слиянием, поэтому использовать её неэффективно.

# **Вывод**

В результате выполнения работы я:

1. Освоил алгоритмы работы с сортировками и их реализацию на языке программирования C++
2. Научился программировать автоматическое тестирование простых программ

# **Исходный код программы**

|  |
| --- |
| #include <iostream>  #include <vector>  #include <string>  #include <algorithm>  #include <random>  #include <ctime>  #include <chrono>  #include <fstream>  typedef unsigned long long ull;  struct sorting\_data  {  ull comparisons;  ull moves;  sorting\_data();  sorting\_data(ull comparisons, ull moves);  };  sorting\_data::sorting\_data()  {  comparisons = 0;  moves = 0;  }  sorting\_data::sorting\_data(ull comparisons, ull moves)  {  this->comparisons = comparisons;  this->moves = moves;  }  sorting\_data selection\_sort(std::vector<int>& vec)  {  sorting\_data data;  for (int i = 0; i < vec.size(); i++)  {  int min\_index = i;  for (int j = i + 1; j < vec.size(); j++)  {  data.comparisons++;  if (vec[j] < vec[min\_index])  {  min\_index = j;  }  }  if (min\_index != i)  {  data.moves++;  std::swap(vec[i], vec[min\_index]);  }  }  return data;  }  sorting\_data heapify(std::vector<int>& vec, int head, int N = -1)  {  sorting\_data data;    if (N == -1)  {  N = vec.size();  }  int largest = head;  int left = 2 \* head + 1;  int right = 2 \* head + 2;  data.comparisons++;  if (left >= 0 && left < N && vec[left] > vec[largest])  {  largest = left;  }  data.comparisons++;  if (right >= 0 && right < N && vec[right] > vec[largest])  {  largest = right;  }  if (largest != head)  {  data.moves++;  std::swap(vec[head], vec[largest]);  sorting\_data returned = heapify(vec, largest, N);  return sorting\_data(data.comparisons + returned.comparisons,  data.moves + returned.moves);  }  return data;  }  sorting\_data heap\_sort(std::vector<int>& vec)  {  sorting\_data data;  for (int i = vec.size() / 2 - 1; i >= 0; i--)  {  sorting\_data returned = heapify(vec, i);  data.comparisons += returned.comparisons;  data.moves += returned.moves;  }  for (int i = vec.size() - 1; i > 0; i--)  {  data.moves++;  std::swap(vec[i], vec[0]);  sorting\_data returned = heapify(vec, 0, i);  data.comparisons += returned.comparisons;  data.moves += returned.moves;  }  return data;  }  sorting\_data merge\_sort\_with\_borders(std::vector<int>& vec, int left, int right)  {  sorting\_data data;  if (left <= -1 || left >= vec.size() ||  right <= -1 || right >= vec.size() ||  left == right)  {  return data;  }  int middle = left + (right - left) / 2;  sorting\_data returned\_left = merge\_sort\_with\_borders(vec, left, middle);  sorting\_data returned\_right = merge\_sort\_with\_borders(vec, middle + 1, right);  data.comparisons += returned\_left.comparisons + returned\_right.comparisons;  data.moves += returned\_left.moves + returned\_right.moves;  std::vector<int> tmp(right - left + 1);  int curr\_left = left;  int curr\_right = middle + 1;  for (int i = 0; i < tmp.size(); i++)  {  if (curr\_left <= middle && curr\_right <= right)  {  data.comparisons++;  if (vec[curr\_left] < vec[curr\_right])  {  tmp[i] = vec[curr\_left];  curr\_left++;  }  else  {  tmp[i] = vec[curr\_right];  curr\_right++;  }  }  else if (curr\_left <= middle)  {  tmp[i] = vec[curr\_left];  curr\_left++;  }  else if (curr\_right <= right)  {  tmp[i] = vec[curr\_right];  curr\_right++;  }  }  for (int i = left; i <= right; i++)  {  vec[i] = tmp[i - left];  }  data.moves += right - left + 1;  return data;  }  sorting\_data merge\_sort(std::vector<int>& vec)  {  return merge\_sort\_with\_borders(vec, 0, vec.size() - 1);  }  bool is\_sorted(const std::vector<int>& vec)  {  for (int i = 0; i < vec.size() - 1; i++)  {  if (vec[i] > vec[i + 1])  {  return false;  }  }  return true;  }  std::vector<int> random\_unsorted\_vector(int size)  {  std::vector<int> vec(size);  for (int i = 0; i < size; i++)  {  vec[i] = std::rand();  }  return vec;  }  std::vector<int> random\_sorted\_vector(int size)  {  std::vector<int> vec = random\_unsorted\_vector(size);  std::sort(vec.begin(), vec.end());  return vec;  }  std::vector<int> random\_reversed\_sorted\_vector(int size)  {  std::vector<int> vec = random\_unsorted\_vector(size);  std::sort(vec.begin(), vec.end(), std::greater<int>());  return vec;  }  void test\_sort(sorting\_data (\*sort)(std::vector<int>&),  std::vector<int> (\*generate\_vector)(int),  std::string output\_file\_name, int min\_size, int max\_size, int step)  {  std::ofstream file(output\_file\_name);  for (int size = min\_size; size <= max\_size; size += step)  {  std::vector<int> vec = generate\_vector(size);  auto start = std::chrono::high\_resolution\_clock::now();  sorting\_data data = sort(vec);  auto stop = std::chrono::high\_resolution\_clock::now();  auto duration = std::chrono::duration\_cast  <std::chrono::milliseconds>(stop - start);  if (!is\_sorted(vec))  {  file << "ERROR: ouput vector is not sorted\n";  }  else  {  file << size << ' ' << duration.count() << ' '  << data.comparisons << ' ' << data.moves << '\n';  }  }  file.close();  }  int main()  {  std::srand(std::time(nullptr));  const int max\_size = 10000;  const int min\_size = 1000;  const int step = 1000;  std::cout << "Testing selection sort on sorted arrays...\n";  test\_sort(selection\_sort, random\_sorted\_vector,  "selection\_sort\_sorted\_results.txt",  min\_size, max\_size, step);  std::cout << "Testing selection sort on reversed sorted arrays...\n";  test\_sort(selection\_sort, random\_reversed\_sorted\_vector,  "selection\_sort\_reversed\_sorted\_results.txt",  min\_size, max\_size, step);  std::cout << "Testing selection sort on unsorted arrays...\n";  test\_sort(selection\_sort, random\_unsorted\_vector,  "selection\_sort\_unsorted\_results.txt",  min\_size, max\_size, step);    std::cout << "Testing heap sort on sorted arrays...\n";  test\_sort(heap\_sort, random\_sorted\_vector,  "heap\_sort\_sorted\_results.txt",  min\_size, max\_size, step);  std::cout << "Testing heap sort on reversed sorted arrays...\n";  test\_sort(heap\_sort, random\_reversed\_sorted\_vector,  "heap\_sort\_reversed\_sorted\_results.txt",  min\_size, max\_size, step);  std::cout << "Testing heap sort on unsorted arrays...\n";  test\_sort(heap\_sort, random\_unsorted\_vector,  "heap\_sort\_unsorted\_results.txt",  min\_size, max\_size, step);  std::cout << "Testing merge sort on sorted arrays...\n";  test\_sort(merge\_sort, random\_sorted\_vector,  "merge\_sort\_sorted\_results.txt",  min\_size, max\_size, step);  std::cout << "Testing merge sort on reversed sorted arrays...\n";  test\_sort(merge\_sort, random\_reversed\_sorted\_vector,  "merge\_sort\_reversed\_sorted\_results.txt",  min\_size, max\_size, step);  std::cout << "Testing merge sort on unsorted arrays...\n";  test\_sort(merge\_sort, random\_unsorted\_vector,  "merge\_sort\_unsorted\_results.txt",  min\_size, max\_size, step);  std::cout << "Done.\n";  return 0;  } |