|  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| |  |  |  | | --- | --- | --- | |  |  |  | | 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 простого выбора (selection sort) - это один из простейших алгоритмов сортировки, который используется для упорядочивания массива элементов. Алгоритм работает путем нахождения наименьшего элемента в массиве и перемещения его в начало массива. Затем алгоритм повторяется для оставшихся элементов, начиная со второго, третьего и т.д., пока все элементы не будут упорядочены. Алгоритм сортировки простого выбора можно разбить на две основные операции:

1. Нахождение наименьшего элемента в массиве. Для этого производится проход по массиву, сравнение каждого элемента с текущим наименьшим и обновление текущего наименьшего элемента, если текущий элемент меньше. Сложность этой операции составляет O(n), где n - размер массива.
2. Перемещение наименьшего элемента в начало массива. Для этого текущий наименьший элемент меняется местами с первым элементом в массиве. Сложность этой операции составляет O(1).

Общая сложность алгоритма сортировки простого выбора равна O(n^2), так как первая операция выполняется n раз, а вторая - n-1 раз, что в сумме даёт n\*(n-1)/2 операций. Несмотря на то, что сложность алгоритма является квадратичной, сортировка простого выбора может быть полезна для сортировки небольших массивов или в случаях, когда другие более эффективные алгоритмы не применимы из-за ограничений на использование памяти или требований к устойчивости сортировки.

Пирамидальная сортировка (Heap sort) - это алгоритм сортировки, который работает на основе двоичной кучи (heap), используя ее свойство: максимальный (минимальный) элемент всегда находится в корне кучи. Основная идея алгоритма заключается в том, чтобы сначала преобразовать входной массив в двоичную кучу, а затем извлекать максимальный (минимальный) элемент из кучи и помещать его в конец массива. Затем происходит перестройка кучи, и процесс повторяется для оставшейся части массива. Алгоритм пирамидальной сортировки можно разделить на две основные операции:

1. Построение двоичной кучи. Для этого сначала массив преобразуется в полную двоичную кучу, то есть каждый узел имеет двух потомков или только одного, если он является листом. Затем куча преобразуется в максимальную (минимальную), обеспечивая свойство, что значение каждого узла не меньше (больше) значений его потомков. Сложность этой операции составляет O(n), где n - размер массива.
2. Извлечение максимального (минимального) элемента из кучи. Для этого корень кучи (максимальный или минимальный элемент) меняется местами с последним элементом в массиве. Затем последний элемент удаляется из кучи, и происходит перестройка кучи. Сложность этой операции составляет O(log n).

Общая сложность алгоритма пирамидальной сортировки также равна O(n log n), так как в худшем случае операция извлечения максимального (минимального) элемента из кучи выполняется n раз, а сложность каждой операции составляет O(log n). Пирамидальная сортировка является стабильной и не зависит от расположения элементов в массиве. Она может быть полезна для сортировки больших объемов данных или в случаях, когда нужно выполнить многократные сортировки с разными критериями.

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 простого выбора.

|  |
| --- |
| void selection\_sort(int arr[], int n, long long &moves, long long &comps) {  int i, j, min\_idx;  for (i = 0; i < n - 1; i++) {  min\_idx = i;  for (j = i + 1; j < n; j++) {  comps++;  if (arr[j] < arr[min\_idx]) {  min\_idx = j;  }  }  if (min\_idx != i) {  swap(arr[i], arr[min\_idx]);  moves += 3;  }  }  } |

Данная функция реализует алгоритм сортировки выбором, который проходит по массиву и ищет минимальный элемент, после чего меняет его местами с первым элементом в массиве. Затем алгоритм продолжает поиск минимального элемента, начиная со второго элемента, и меняет его местами со вторым элементом в массиве, и так далее, пока весь массив не будет отсортирован. Переменная min\_idx отслеживает индекс текущего минимального элемента, переменная moves отслеживает количество перемещений элементов в процессе сортировки, а переменная comps - количество сравнений.

* 1. **Выполнение второго задания.**

Для выполнения второго задания необходимо разработать функцию, реализующую алгоритм пирамидальной сортировки.

|  |
| --- |
| void heapify(int arr[], int n, int i, long long& comps, long long& moves) {  int largest = i;  int l = 2 \* i + 1;  int r = 2 \* i + 2;  comps++;  if (l < n && arr[l] > arr[largest])  largest = l;  comps++;  if (r < n && arr[r] > arr[largest])  largest = r;  if (largest != i) {  moves += 3;  swap(arr[i], arr[largest]);  heapify(arr, n, largest, comps, moves);  }  }  void heapSort(int arr[], int n, long long& comps, long long& moves) {  for (int i = n / 2 - 1; i >= 0; i--) {  int largest = i;  int l = 2 \* i + 1;  int r = 2 \* i + 2;  comps++;  if (l < n && arr[l] > arr[largest])  largest = l;  comps++;  if (r < n && arr[r] > arr[largest])  largest = r;  if (largest != i) {  moves += 3;  swap(arr[i], arr[largest]);  heapify(arr, n, largest, comps, moves);  }  }  for (int i = n - 1; i >= 0; i--) {  moves += 3;  swap(arr[0], arr[i]);  heapify(arr, i, 0, comps, moves);  }  } |

Данная функция реализует алгоритм пирамидальной сортировки. Сначала функция heapSort строит кучу из массива arr. Для этого проходит цикл от i = n/2 - 1 до i = 0 и вызывает функцию heapify для каждого i, которая перестраивает поддерево с корнем i так, чтобы оно удовлетворяло свойствам кучи. Затем второй цикл от i = n-1 до i = 0 последовательно извлекает максимальный элемент из корня кучи (который находится в arr[0]) и перестраивает кучу так, чтобы ее корень содержал следующий по величине элемент. Для этого функция сначала меняет местами arr[0] и arr[i], затем вызывает heapify для поддерева с корнем в arr[0] и размером i. Переменная moves отслеживает количество перемещений элементов в процессе сортировки, а переменная comps - количество сравнений. Функция heapify принимает массив arr, размерность n, индекс i, а также ссылки на переменные comps и moves. Она сначала находит наибольший элемент среди i, 2i+1 и 2i+2, и записывает его индекс в largest. Затем проверяет, если largest не равен i, то меняет местами arr[i] и arr[largest], увеличивает moves на 3 и вызывает heapify для поддерева с корнем в largest и размером n. Функция heapSort не возвращает значение, так как сортирует исходный массив arr по ссылке.

* 1. **Выполнение третьего задания.**

Для выполнения последнего задания необходимо разработать функцию, реализующую сортировку методом простого слияния. 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;  while (i < leftHalfSize && j < rightHalfSize)  {  if (leftHalf[i] <= rightHalf[j])  {  arr[k++] = leftHalf[i++];  }  else  {  arr[k++] = rightHalf[j++];  }  moves++;  comps++;  }  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.

* 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.

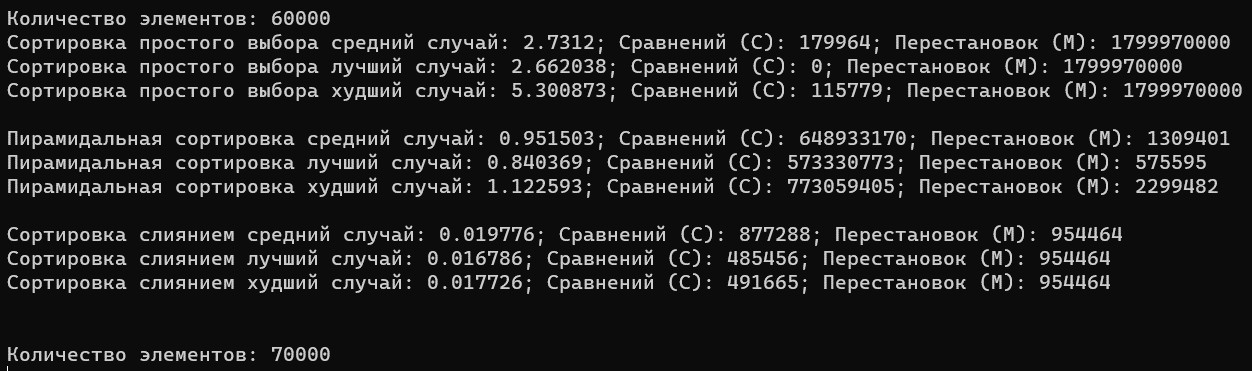


Рисунок 1. Интерфейс работы программы

1. **Тестирование**

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

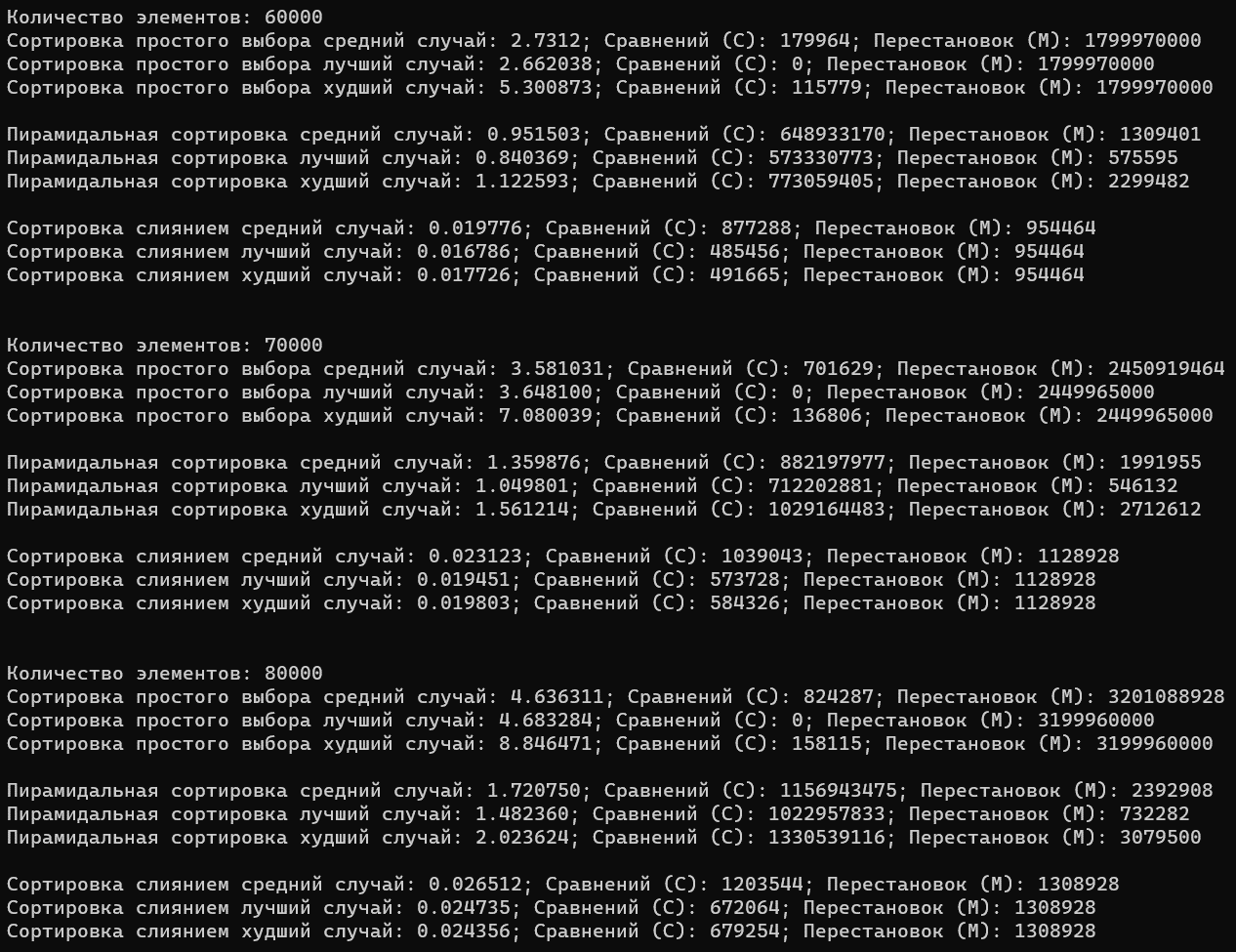


Рисунок 2. Тестирование работы трех алгоритмов  
на массивах размером с 60000 по 80000

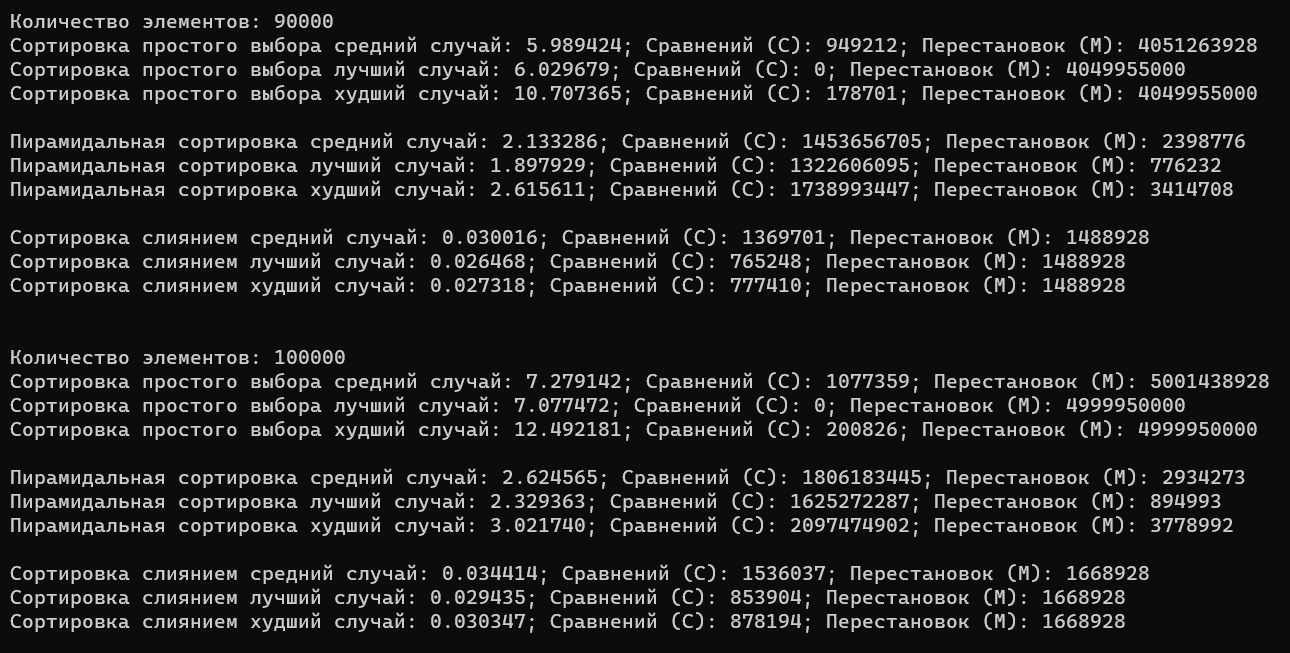


Рисунок 3. Тестирование работы трех алгоритмов  
на массивах размером 90000 и 100000

Программа работает корректно. Выпишем в таблицы 1-9 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 1. Таблица результатов тестирования сортировки простого выбора для среднего случая

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **N** | **t, сек** | **Tа(n)=N2** | **Tэ=С+M** | **Tэ/Tа** |
| 60000 | 2,7312 | 3600000000 | 1800149964 | 0,5000417 |
| 70000 | 3.581031 | 4900000000 | 2451621093 | 0,5003308 |
| 80000 | 4.636311 | 6400000000 | 3201913215 | 0,5002989 |
| 90000 | 5,989424 | 8100000000 | 4052213140 | 0,5002732 |
| 100000 | 7.279142 | 10000000000 | 5002516287 | 0,5002516 |

Таблица 2. Таблица результатов тестирования сортировки простого выбора для лучшего случая

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **N** | **t, сек** | **Tа(n)= N2** | **Tэ=С+M** | **Tэ/Tа** |
| 60000 | 2.662038 | 3600000000 | 1799970000 | 0,499992 |
| 70000 | 3.648100 | 4900000000 | 2449965000 | 0,499993 |
| 80000 | 4.683284 | 6400000000 | 3199960000 | 0,499994 |
| 90000 | 6.029679 | 8100000000 | 4049955000 | 0,499994 |
| 100000 | 7.077472 | 10000000000 | 4999950000 | 0,499995 |

Таблица 3. Таблица результатов тестирования сортировки простого выбора для худшего случая

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **N** | **t, сек** | **Tа(n)=N2** | **Tэ=С+M** | **Tэ/Tа** |
| 60000 | 5.300873 | 3600000000 | 1800085779 | 0,5000238 |
| 70000 | 7.080039 | 4900000000 | 2449965000 | 0,4999929 |
| 80000 | 8.846471 | 6400000000 | 3200118115 | 0,5000185 |
| 90000 | 10.707365 | 8100000000 | 4050133701 | 0,5000165 |
| 100000 | 12.492181 | 10000000000 | 5000150826 | 0,5000151 |

Таблица 4. Таблица результатов тестирования пирамидальной сортировки для среднего случая

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **N** | **t, сек** | **Tа(n)=Nlog(N)** | **Tэ=С+M** | **Tэ/Tа** |
| 60000 | 0.013454 | 828504.4 | 4557685 | 5.5014 |
| 70000 | 0.016455 | 996584.6 | 5394695 | 5.4177 |
| 80000 | 0.017751 | 1175722.2 | 6248136 | 5.3021 |
| 90000 | 0.022473 | 1355337.2 | 8007665 | 5.9072 |
| 100000 | 0.024997 | 1534264.5 | 9147108 | 5.9656 |

Таблица 5. Таблица результатов тестирования пирамидальной сортировки для лучшего случая

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **N** | **t, сек** | **Tа(n)= Nlog(N)** | **Tэ=С+M** | **Tэ/Tа** |
| 60000 | 0.011676 | 828504.4 | 5274695 | 6.3727 |
| 70000 | 0.014786 | 996584.6 | 6271669 | 6.2969 |
| 80000 | 0.017149 | 1175722.2 | 7271110 | 6.1756 |
| 90000 | 0.020718 | 1355337.2 | 8270552 | 6.1031 |
| 100000 | 0.023735 | 1534264.5 | 9269994 | 6.0424 |

Таблица 6. Таблица результатов тестирования пирамидальной сортировки для худшего случая

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **N** | **t, сек** | **Tа(n)= Nlog(N)** | **Tэ=С+M** | **Tэ/Tа** |
| 60000 | 0.010922 | 828504.4 | 4334309 | 5.2291 |
| 70000 | 0.012760 | 996584.6 | 5052885 | 5.0681 |
| 80000 | 0.016574 | 1175722.2 | 5771461 | 4.9057 |
| 90000 | 0.020147 | 1355337.2 | 6490037 | 4.7899 |
| 100000 | 0.022314 | 1534264.5 | 7208613 | 4.7005 |

Таблица 7. Таблица результатов тестирования сортировки простым слиянием для среднего случая

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **N** | **t, сек** | **Tа(n)=Nlog(N)** | **Tэ=С+M** | **Tэ/Tа** |
| 60000 | 0.020645 | 828504.4 | 1831752 | 2.207 |
| 70000 | 0.025401 | 996584.6 | 2167971 | 2.096 |
| 80000 | 0.031604 | 1175722.2 | 2512472 | 2.135 |
| 90000 | 0.032210 | 1355337.2 | 2858629 | 2.109 |
| 100000 | 0.035249 | 1534264.5 | 3204965 | 2.090 |

Таблица 8. Таблица результатов тестирования сортировки простым слиянием для лучшего случая

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **N** | **t, сек** | **Tа(n)= Nlog(N)** | **Tэ=С+M** | **Tэ/Tа** |
| 60000 | 0.017575 | 828504.4 | 954464 | 1.151 |
| 70000 | 0.019727 | 996584.6 | 954464 | 0.958 |
| 80000 | 0.023460 | 1175722.2 | 1308928 | 1.111 |
| 90000 | 0.026581 | 1355337.2 | 1488928 | 1.097 |
| 100000 | 0.028549 | 1534264.5 | 1668928 | 1.087 |

Таблица 9. Таблица результатов тестирования сортировки простым слиянием для худшего случая

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **N** | **t, сек** | **Tа(n)= Nlog(N)** | **Tэ=С+M** | **Tэ/Tа** |
| 60000 | 0.017245 | 828504.4 | 954464 | 1.151 |
| 70000 | 0.019908 | 996584.6 | 1128928 | 1.131 |
| 80000 | 0.024360 | 1175722.2 | 1308928 | 1.112 |
| 90000 | 0.031360 | 1355337.2 | 1488928 | 1.099 |
| 100000 | 0.029924 | 1534264.5 | 1668928 | 1.087 |

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

Сортировка простым выбором и пирамидальная сортировка имеют худшие показатели по количеству сравнений/перемещений и затраченному времени, чем сортировка простым слиянием, хоть и незначительные, если сравнивать сортировку простым слиянием и пирамидальную. Для больших массивов данных сортировка простым слиянием является более эффективной. Однако, в некоторых случаях, сортировка простым выбором и пирамидальная могут быть эффективнее, например, для небольших массивов данных или для частично отсортированных массивов. В целом, можно сказать, что сортировка простым слиянием является наиболее универсальной и эффективной для большинства случаев сортировки массивов данных.

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 selection\_sort(int arr[], int n, long long &moves, long long &comps) {  int i, j, min\_idx;  for (i = 0; i < n - 1; i++) {  min\_idx = i;  for (j = i + 1; j < n; j++) {  comps++;  if (arr[j] < arr[min\_idx]) {  min\_idx = j;  }  }  if (min\_idx != i) {  swap(arr[i], arr[min\_idx]);  moves += 3;  }  }  }  void heapify(int arr[], int n, int i, long long& comps, long long& moves) {  int largest = i;  int l = 2 \* i + 1;  int r = 2 \* i + 2;  comps++;  if (l < n && arr[l] > arr[largest])  largest = l;  comps++;  if (r < n && arr[r] > arr[largest])  largest = r;  if (largest != i) {  moves += 3;  swap(arr[i], arr[largest]);  heapify(arr, n, largest, comps, moves);  }  }  void heapSort(int arr[], int n, long long& comps, long long& moves) {  for (int i = n / 2 - 1; i >= 0; i--) {  int largest = i;  int l = 2 \* i + 1;  int r = 2 \* i + 2;  comps++;  if (l < n && arr[l] > arr[largest])  largest = l;  comps++;  if (r < n && arr[r] > arr[largest])  largest = r;  if (largest != i) {  moves += 3;  swap(arr[i], arr[largest]);  heapify(arr, n, largest, comps, moves);  }  }  for (int i = n - 1; i >= 0; i--) {  moves += 3;  swap(arr[0], arr[i]);  heapify(arr, i, 0, comps, moves);  }  }  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;  while (i < leftHalfSize && j < rightHalfSize)  {  if (leftHalf[i] <= rightHalf[j])  {  arr[k++] = leftHalf[i++];  }  else  {  arr[k++] = rightHalf[j++];  }  moves++;  comps++;  }  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();  selection\_sort(arr, size, comps, moves);  auto end = steady\_clock::now();  auto difference = duration\_cast<duration<float>>(end - start).count();  cout << "Сортировка простого выбора средний случай: " << difference << "; Сравнений (C): " << comps << "; Перестановок (M): " << moves << endl;  comps = 0, moves = 0;  start = steady\_clock::now();  selection\_sort(arr, size, comps, moves);  end = steady\_clock::now();  difference = duration\_cast<duration<float>>(end - start).count();  cout << "Сортировка простого выбора лучший случай: " << fixed << difference << "; Сравнений (C): " << comps << "; Перестановок (M): " << moves << endl;  comps = 0, moves = 0;  Reverse(arr, size);  start = steady\_clock::now();  selection\_sort(arr, size, comps, moves);  end = steady\_clock::now();  difference = duration\_cast<duration<float>>(end - start).count();  cout << "Сортировка простого выбора худший случай: " << difference << "; Сравнений (C): " << comps << "; Перестановок (M): " << moves << endl << endl;  CopyArray(arr, temp, size);  comps = 0, moves = 0;  start = steady\_clock::now();  heapSort(arr, size, comps, moves);  end = steady\_clock::now();  difference = duration\_cast<duration<float>>(end - start).count();  cout << "Пирамидальная сортировка средний случай: " << difference << "; Сравнений (C): " << comps << "; Перестановок (M): " << moves << endl;  comps = 0, moves = 0;  start = steady\_clock::now();  heapSort(arr, size, comps, moves);  end = steady\_clock::now();  difference = duration\_cast<duration<float>>(end - start).count();  cout << "Пирамидальная сортировка лучший случай: " << fixed << difference << "; Сравнений (C): " << comps << "; Перестановок (M): " << moves << endl;  comps = 0, moves = 0;  Reverse(arr, size);  start = steady\_clock::now();  heapSort(arr, size, comps, moves);  end = steady\_clock::now();  difference = duration\_cast<duration<float>>(end - start).count();  cout << "Пирамидальная 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;  }  } |