

Министерство науки и высшего образования Российской Федерации
Федеральное государственное автономное образовательное учреждение
высшего образования
«СЕВЕРО-КАВКАЗСКИЙ ФЕДЕРАЛЬНЫЙ УНИВЕРСИТЕТ»

Институт цифрового развития
Кафедра инфокоммуникаций

ОТЧЕТ
ПО ПРАКТИЧЕСКОЙ РАБОТЕ №10
дисциплины «Алгоритмизация»
Вариант 7

Выполнил:
Горбунов Данила Евгеньевич
2 курс, группа ИВТ-б-о-22-1,
09.03.01 «Информатика и
вычислительная техника»,
направленность (профиль)
«Программное обеспечение средств
вычислительной техники и
автоматизированных систем», очная
форма обучения

(подпись)

Руководитель практики:
Воронкин Р. А., канд. технических
наук, доцент кафедры
инфокоммуникаций

(подпись)

Отчет защищен с оценкой _____ Дата защиты _____

Ставрополь, 2023 г.

Тема: Алгоритм сортировки кучей

Ход работы:

1. Написал программы, в каждой из которых реализовал алгоритм сортировки кучи и посчитал время работы в случае, когда на вход идут: отсортированный массив, массив, который обратный отсортированному и массив со случайными значениями:

Консоль отладки Microsoft Visual Studio

```
Размер массива: 1000, Время сортировки: 472 микросекунд
Размер массива: 2000, Время сортировки: 811 микросекунд
Размер массива: 3000, Время сортировки: 1258 микросекунд
Размер массива: 4000, Время сортировки: 1715 микросекунд
Размер массива: 5000, Время сортировки: 2262 микросекунд
Размер массива: 6000, Время сортировки: 2692 микросекунд
Размер массива: 7000, Время сортировки: 3231 микросекунд
Размер массива: 8000, Время сортировки: 3697 микросекунд
Размер массива: 9000, Время сортировки: 4168 микросекунд
Размер массива: 10000, Время сортировки: 4650 микросекунд
Размер массива: 11000, Время сортировки: 5448 микросекунд
Размер массива: 12000, Время сортировки: 6348 микросекунд
Размер массива: 13000, Время сортировки: 6835 микросекунд
Размер массива: 14000, Время сортировки: 7009 микросекунд
Размер массива: 15000, Время сортировки: 7793 микросекунд
Размер массива: 16000, Время сортировки: 8098 микросекунд
Размер массива: 17000, Время сортировки: 8439 микросекунд
Размер массива: 18000, Время сортировки: 9470 микросекунд
Размер массива: 19000, Время сортировки: 9694 микросекунд
Размер массива: 20000, Время сортировки: 10366 микросекунд
Размер массива: 21000, Время сортировки: 10612 микросекунд
Размер массива: 22000, Время сортировки: 11146 микросекунд
Размер массива: 23000, Время сортировки: 11684 микросекунд
Размер массива: 24000, Время сортировки: 12420 микросекунд
Размер массива: 25000, Время сортировки: 13049 микросекунд
Размер массива: 26000, Время сортировки: 13531 микросекунд
Размер массива: 27000, Время сортировки: 14250 микросекунд
Размер массива: 28000, Время сортировки: 14361 микросекунд
Размер массива: 29000, Время сортировки: 15179 микросекунд
Размер массива: 30000, Время сортировки: 15749 микросекунд
Размер массива: 31000, Время сортировки: 17273 микросекунд
Размер массива: 32000, Время сортировки: 16895 микросекунд
Размер массива: 33000, Время сортировки: 19715 микросекунд
Размер массива: 34000, Время сортировки: 20655 микросекунд
Размер массива: 35000, Время сортировки: 18822 микросекунд
Размер массива: 36000, Время сортировки: 19080 микросекунд
Размер массива: 37000, Время сортировки: 20035 микросекунд
Размер массива: 38000, Время сортировки: 19984 микросекунд
Размер массива: 39000, Время сортировки: 22331 микросекунд
Размер массива: 40000, Время сортировки: 22626 микросекунд
Размер массива: 41000, Время сортировки: 24561 микросекунд
Размер массива: 42000, Время сортировки: 22951 микросекунд
Размер массива: 43000, Время сортировки: 23365 микросекунд
Размер массива: 44000, Время сортировки: 24202 микросекунд
Размер массива: 45000, Время сортировки: 25368 микросекунд
Размер массива: 46000, Время сортировки: 28469 микросекунд
Размер массива: 47000, Время сортировки: 26247 микросекунд
Размер массива: 48000, Время сортировки: 27185 микросекунд
Размер массива: 49000, Время сортировки: 29874 микросекунд
Размер массива: 50000, Время сортировки: 30083 микросекунд
```

Рисунок 1. Результат работы программы

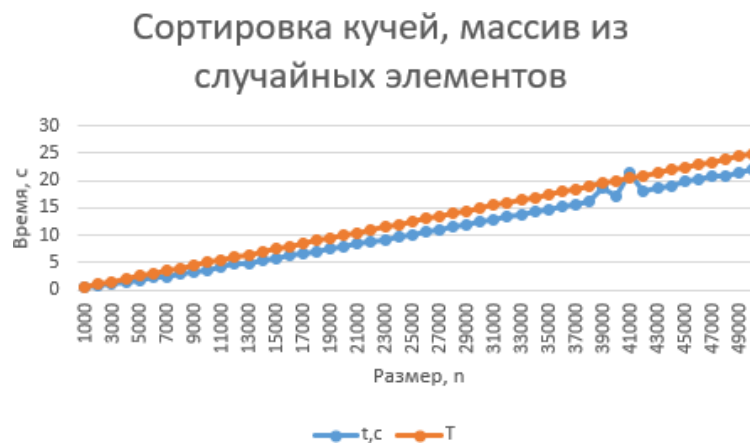
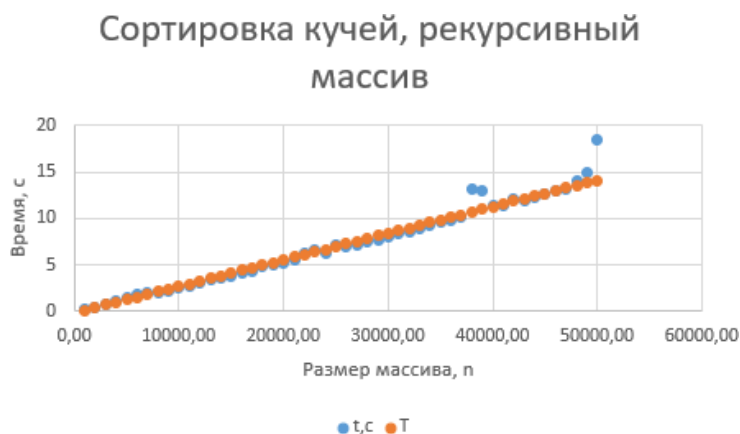
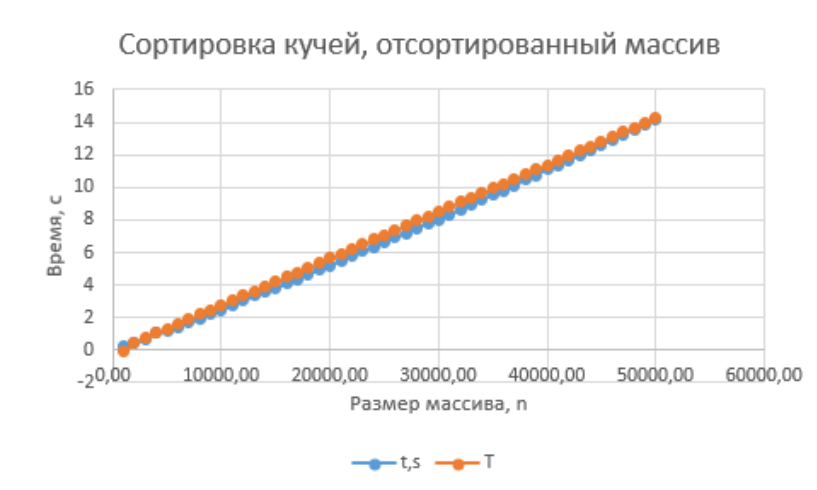


Рисунок 2 – Графики зависимости длины массива от времени

2. Сравнение с другими сортировками

Случай	Лучший	Средний	Худший
Heap Sort	$O(n \log n)$	$O(n \log n)$	$O(n \log n)$
Quick Sort	$O(n \log n)$	$O(n \log n)$	$O(n^2)$
Merge Sort	$O(n \log n)$	$O(n \log n)$	$O(n \log n)$

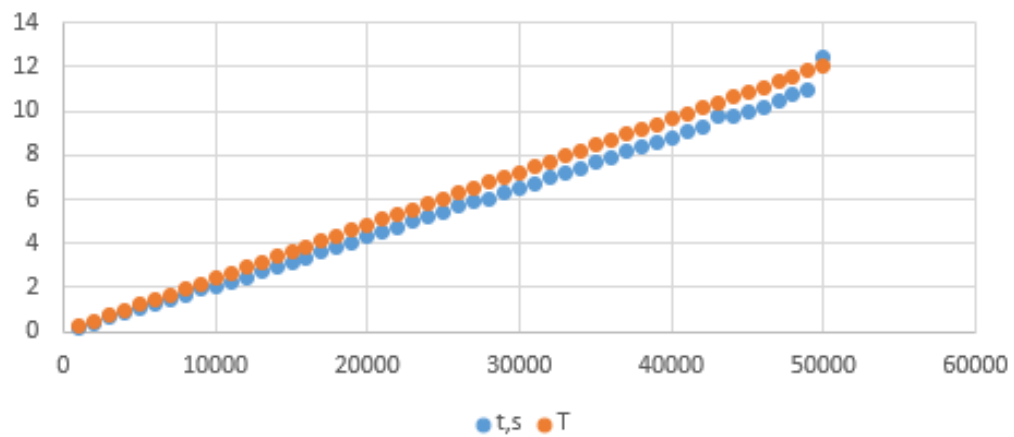
Алгоритм Heap Sort характеризуется временной сложностью $O(n \log n)$ в лучшем, среднем и худшем случае, что делает его подходящим для обработки больших объемов данных. Однако, он не обеспечивает стабильность сортировки, что может быть неудобно в некоторых ситуациях.

В свою очередь, алгоритм Merge Sort также имеет временную сложность $O(n \log n)$ во всех трех случаях: лучшем, среднем и худшем. Это делает его эффективным для сортировки больших массивов данных. Одним из его преимуществ является стабильность, то есть сохранение исходного порядка одинаковых элементов. Но этот алгоритм требует дополнительного пространства в памяти для слияния массивов, что может стать проблемой при работе с очень большими объемами данных.

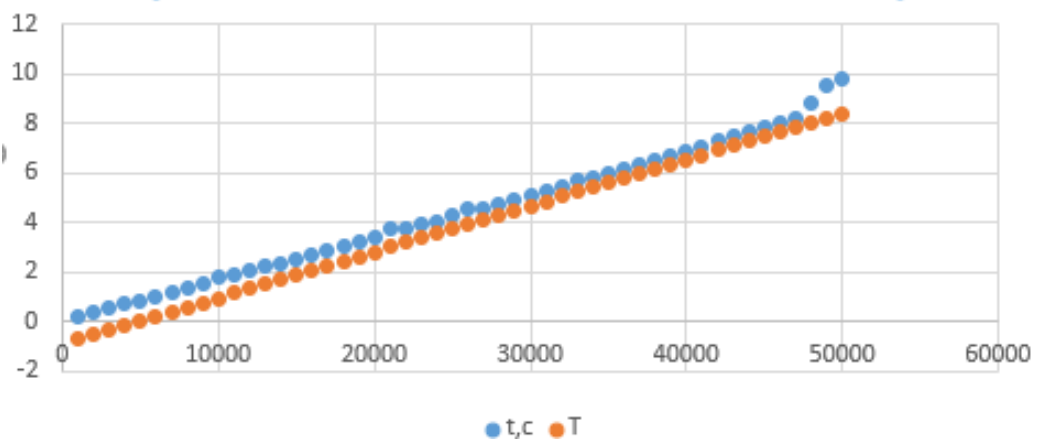
Алгоритм Quick Sort имеет временную сложность $O(n \log n)$ в среднем и лучшем случае, но в худшем случае его сложность может достигать $O(n^2)$. Несмотря на это, Quick Sort часто применяется на практике из-за своей высокой эффективности в среднем случае. Однако, в худшем случае, когда выбор опорного элемента не оптимален, производительность алгоритма может снизиться до квадратичной сложности.

3. Написал программы, в которых реализовал оптимизированный алгоритм сортировки кучи и посчитал время работы в случае, когда на вход идут: отсортированный массив, массив, который обратный отсортированному и массив с рандомными значениями, а оптимизация заключалась в том, что я использовал: использовал эффективную функцию `heapify` для построения кучи, использовал эффективную функцию `heapify` для восстановления кучи, использовал оптимизированные структуры данных и алгоритмов

Сортировка кучей, отсортированный массив



Сортировка кучей, рекурсивный массив



Сортировка кучей, массив из случайных элементов

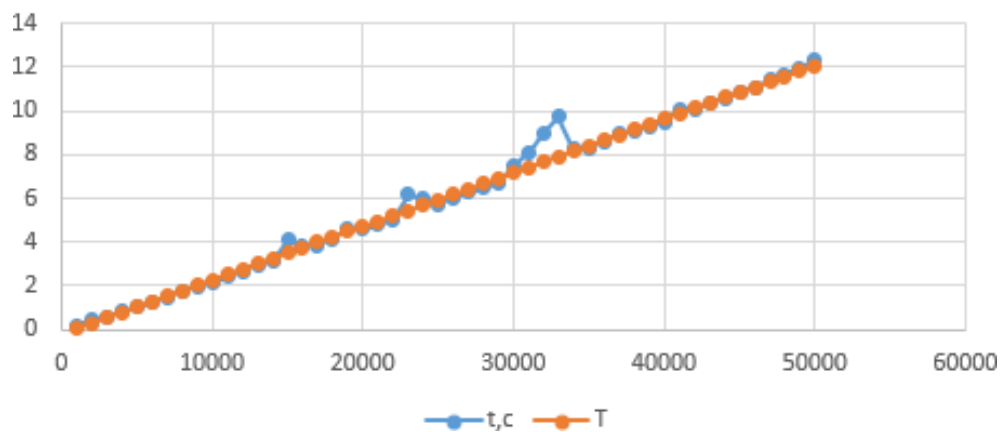


Рисунок 3 – Графики зависимости длины списка от времени

4. Применение в реальной жизни

Алгоритм Heap Sort может использоваться в реальных условиях, таких как улучшение функционирования баз данных, обработки событий и других вычислительных процедур. Его предпочтительность в определенных обстоятельствах обусловлена гарантией времени исполнения в худшем случае. Например, в ситуациях, связанных с базами данных, где важен быстрый доступ к упорядоченной информации, Heap Sort может оказаться полезным благодаря его эффективности и предсказуемости времени работы. Кроме того, в условиях, где требуется обработка больших объемов данных и где гарантированное время выполнения играет важную роль, алгоритм Heap Sort может стать предпочтительным решением.

5. Анализ сложности

Алгоритм Heap Sort имеет временную сложность $O(n \log n)$ в худшем, лучшем и среднем случае. Пространственная сложность составляет $O(1)$, что означает, что дополнительная память, используемая для сортировки, не зависит от размера входных данных.

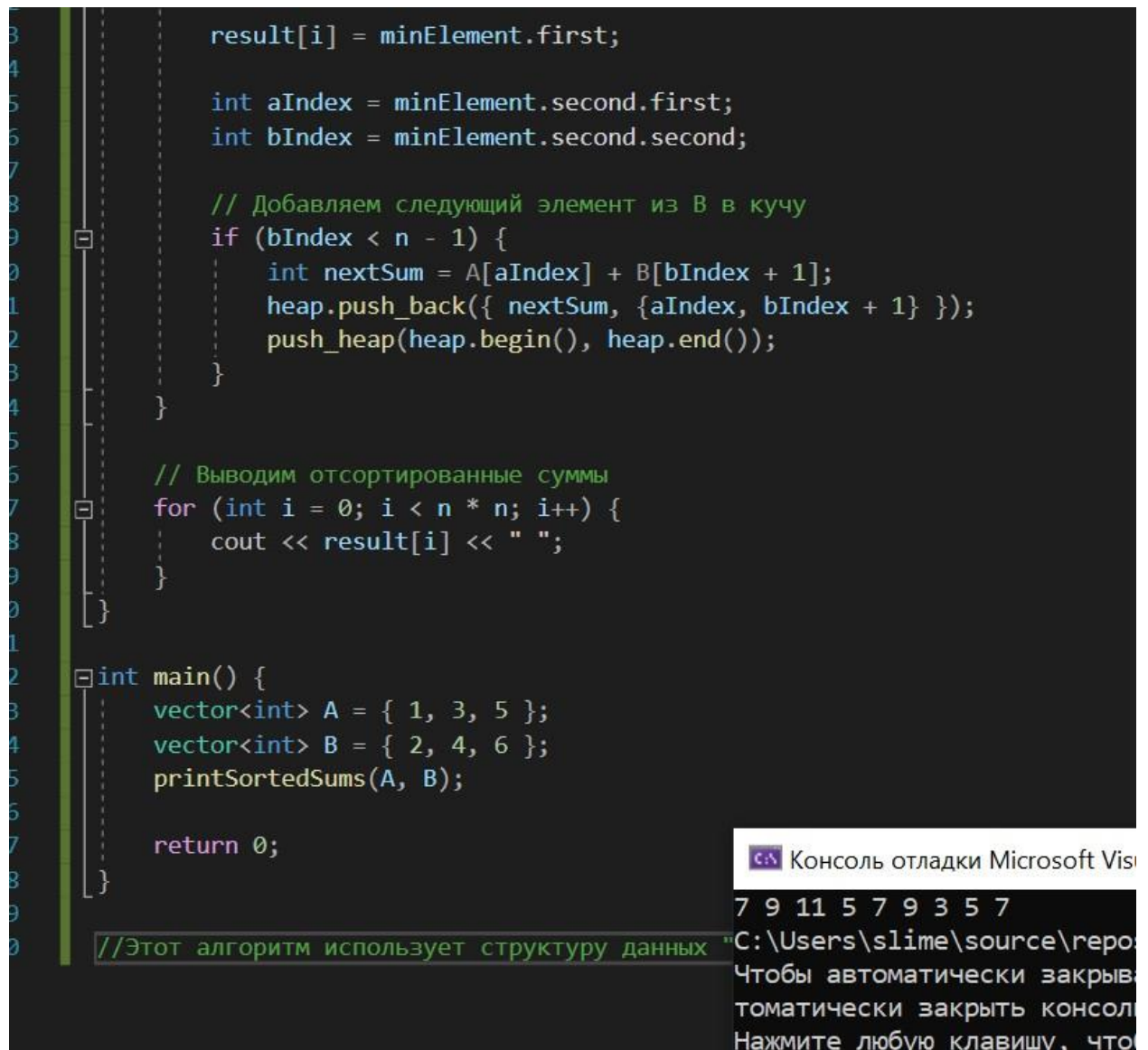
Характеристики алгоритма зависят от размера входных данных следующим образом: с увеличением размера входных данных время выполнения алгоритма Heap Sort увеличивается логарифмически, что делает его эффективным для больших объемов данных. Однако, в сравнении с другими алгоритмами сортировки, такими как Quick Sort или Merge Sort, в некоторых случаях Heap Sort может быть менее эффективным из-за постоянных операций с памятью и более сложной реализации.

Таким образом, Heap Sort может быть более эффективным в случаях, когда требуется гарантированное время выполнения в худшем случае и при работе с большими объемами данных, но может быть менее эффективным по сравнению с другими алгоритмами сортировки в некоторых сценариях, где важна простота реализации и минимальное использование памяти.

6. Даны массивы $A[1 \dots n]$ и $B[1 \dots n]$. Мы хотим вывести все n^2 сумм вида $A[i] + B[j]$ в возрастающем порядке. Наивный способ — создать массив,

содержащий все такие суммы, и отсортировать его. Соответствующий алгоритм имеет время работы $O(n^2 \log n)$ и использует $O(n^2)$ памяти. Приведите алгоритм с таким же временем работы, который использует линейную память.

Написал программу (task6.cpp), которая решает данную задачу:



```
3 result[i] = minElement.first;
4
5 int aIndex = minElement.second.first;
6 int bIndex = minElement.second.second;
7
8 // Добавляем следующий элемент из B в кучу
9 if (bIndex < n - 1) {
10     int nextSum = A[aIndex] + B[bIndex + 1];
11     heap.push_back({ nextSum, {aIndex, bIndex + 1} });
12     push_heap(heap.begin(), heap.end());
13 }
14 }
15
16 // Выводим отсортированные суммы
17 for (int i = 0; i < n * n; i++) {
18     cout << result[i] << " ";
19 }
20
21 int main() {
22     vector<int> A = { 1, 3, 5 };
23     vector<int> B = { 2, 4, 6 };
24     printSortedSums(A, B);
25
26     return 0;
27 }
```

//Этот алгоритм использует структуру данных

Консоль отладки Microsoft Visual Studio

7 9 11 5 7 9 3 5 7

C:\Users\slime\source\repo

Чтобы автоматически закрыть

томатически закрыть консол

Нажмите любую клавишу, что

Рисунок 4 – Результат выполнения программы task6.py

Вывод: в ходе выполнения лабораторной работы был исследован алгоритм сортировки кучей, было выяснено, что с использованием функций `make_heap` и `sort_heap`, алгоритм работает значительно быстрее изначальной реализации.