ФЕДЕРАЛЬНОЕ ГОСУДАРСТВЕННОЕ БЮДЖЕТНОЕ ОБРАЗОВАТЕЛЬНОЕ УЧРЕЖДЕНИЕ ВЫСШЕГО ОБРАЗОВАНИЯ

«САНКТ-ПЕТЕРБУРГСКИЙ ГОСУДАРСТВЕННЫЙ УНИВЕРСИТЕТ ТЕЛЕКОММУНИКАЦИЙ ИМ. ПРОФ. М.А. БОНЧ-БРУЕВИЧА»

(СПбГУТ)

Факультет инфокоммуникационных Сетей и систем (иксс)

кафедра программной инженерии и вычислительной техники (пи и вт)

Дисциплина: «Алгоритмы и структуры данных»

Лабораторная работа №1.

Тема: «Методы сортировки»

Выполнили:

Студенты группы ИКПИ-05

Принял:

Доцент кафедры ПИиВТ

Молошников Ф.А., Мартынюк А.А.

Подпись\_\_\_\_\_\_\_\_\_\_\_\_

Дагаев А.В.

Подпись\_\_\_\_\_\_\_\_\_\_\_\_

«\_\_\_\_\_»\_\_\_\_\_\_\_\_2022

СОДЕРЖАНИЕ

[1. Цель работы 3](#_Toc96550274)

[2. Описание Программы 3](#_Toc96550275)

[2.1. Разработка классов и функций 3](#_Toc96550276)

[2.1.1. statistics 3](#_Toc96550277)

[2.1.2. Heapify, BuildHeap, HeapSort 3](#_Toc96550278)

[2.1.3. QuickSortHoare 4](#_Toc96550279)

[2.1.4. ShellSort 4](#_Toc96550280)

[3. Описание алгоритмов 4](#_Toc96550281)

[3.1. Сортировка двоичной кучей 4](#_Toc96550282)

[3.1.1. Восстановление свойств кучи. 4](#_Toc96550283)

[3.1.2. Построение кучи. 4](#_Toc96550284)

[3.1.3. Сортировка кучей 4](#_Toc96550285)

[3.2. Быстрая сортировка Хоара 4](#_Toc96550286)

[3.3. Сортировка Шелла 5](#_Toc96550287)

[4. Сравнительный анализ 5](#_Toc96550288)

[5. Результаты работы 11](#_Toc96550289)

[6. Выводы 12](#_Toc96550290)

1. Цель работы

Ознакомление с алгоритмами сортировки линейных и нелинейных структур и методикой оценки эффективности алгоритмов. Для сравнения были выбраны сортировка кучей, быстрая сортировка Хоара, а также сортировка Шелла.

1. Описание Программы
2. Язык: C++
3. Среда: VS Code
4. ОС: Debian 11
5. Оболочка: xfce 4
6. Библиотеки: -
7. Фреймворк: -
   1. Разработка классов и функций
      1. statistics

Шаблонный класс для исследования функций, выполняющих сортировку данных.

* + - 1. Свойства:

|  |  |
| --- | --- |
| int CurrentSize=0; | текущее число элементов в массиве |
| T\* Arr=nullptr; | Массив данных для сортировки |
| int NStart=0, NEnd=0, NStep=0; | Начальный, конечный размер массива, шаг изменения массива |
| int RandMax=-1; | Диапазон значений в массиве (по умолчанию весь допустимый для данного типа) |
| int NumberOfRuns=1; | Число прогонов для массива данного размера |

* + - 1. Методы:

|  |  |
| --- | --- |
| statistics(void (\*callback)(T\*, int, bool, bool, int), int NNStart, int NNEnd, int NNStep, bool SortOrder, bool IsStepbystep, int NRandMax, int NNumberOfRuns, char\* filename) | Конструктор |
| void printArr(char\* label) | Вывод массива в терминал |
| void printLabel(char\* filename) | Запись информации об условиях выполнения сортировки в файл |
| void printValue(int CurrentSize, double value, char\* filename) | Запись искомых значений времени в файл |

* + 1. Heapify, BuildHeap, HeapSort

Шаблонные функции для сортировки двоичной кучей

void Heapify(T \*Arr, int Length, int i, bool SortOrder, bool IsStepbystep, int reclvl) – восстанавливает свойства кучи

void BuildHeap(T \*Arr, int Length, bool SortOrder, bool IsStepbystep) – строит кучу

void HeapSort(T \*Arr, int Length, bool SortOrder, bool IsStepbystep, int reclvl) – сортирует с помощью кучи

* + 1. QuickSortHoare

Функция для осуществления быстрой сортировки Хоара

* + 1. ShellSort

Функция для осуществления сортировкой Шелла

1. Описание алгоритмов
   1. Сортировка двоичной кучей

Тип: Выбором

Сложность: **O(n\*log(n))**

Соритровка двоичной кучей состоит из двух этапов: построение двоичной кучи и удаление отсоритрованных элементов по одному с последующей перестройкой дерева(восстановление свойств кучи). Далее будем рассматривать сортировку по возрастанию.

Известно, что дерево, удовлетворяющее 2 и 3 условиям, можно представить в виде массива длиной n. Причем элемент **i** будет иметь потомков с индексами **2i+1** и **2i+2**

1. Значение в любой вершине не меньше, чем значения её потомков[К 1].
2. Глубина всех листьев (расстояние до корня) отличается не более чем на 1 слой.
3. Последний слой заполняется слева направо без «дырок».
   * 1. Восстановление свойств кучи.

Если в куче изменить корневой элемент, то она может перестать удовлетворять условиям кучи. Для восстановления свойств кучи была написана функция Heapify.

1. Сравнивается значение корневого элемента со значениями его непосредственных потомков, и если значение корня оказалось не наибольшим, то производится обмен значения корня со значением потомка, у которого значение наибольшее из этих 3 элементов.
2. Рекурсивно вызывается Heapify, для поддерева, которое было затронуто в результате такого обмена. Тривиальным случаем здесь является ситуация, когда у узла нет потомков.
   * 1. Построение кучи.
3. Для всех элементов, имеющих потомки, вызывается Heapify начиная от наиболее младших элементов.
   * 1. Сортировка кучей
4. После построения кучи корневой элемент (т.е. элемент с индексом **0**) является наибольшим. Происходит обмен корневого элемента(0го в массиве) и наиболее правого листа в дереве (последнего элемента в массиве), таким образом, элемент с наибольшим значением занимает свою позицию в массиве и исключается из дерева.
5. Для корня дерева, получившегося после исключения элемента с наибольшим значением, вызывается функция Heapify, после чего в корень дерева попадает наибольший элемент из оставшихся.
6. Этот элемент меняется с наиболее правым листом дерева(предпоследним элементом в массиве), в результате чего этот элемент занимает свою позицию в отсортированном массиве.
7. И так далее, пока массив не окажется отсортированным.

.

* 1. Быстрая сортировка Хоара

Тип: Обменная

Сложность: O(n^2)

Один из самых быстрых известных универсальных алгоритмов сортировки массивов.

1. Среди элементов массива выбирается базовый ключ. В нашем алгоритме это элемент, расположенный посередине массива.
2. Индексы **i j** предназначены, чтобы пробегать по массиву с начала в сторону конца и с конца в сторону начала. Итак, после выбора базового ключа **i** начинает пробегать от начала в конец, пока не встретит элемент, который окажется не меньше, чем базовый ключ.
3. Индекс **j** начинает пробегать с конца в начало, пока не встретит элемент, который окажется не больше, чем базовый ключ.
4. i и j элементы меняются местами. (таким образом, все элементы, стоящие в начале и в конце массива оказываются соответственно меньше и больше базового ключа)
5. 2 и 3 шаг повторяются, пока I и j не встретятся. В результате чего массив оказывается разделен на 2 части: в левой части элементы <= базовому ключу, а в правой части >= базовому ключу.
6. Далее для левой и правой частей массива функция вызывается рекурсивно. Тривиальный случай – когда массив состоит из одного эелмента.
   1. Сортировка Шелла

Тип: Вставками

Сложность: O(n^2)

Алгоритм представляет собой 3 вложенных в друг друга циклы

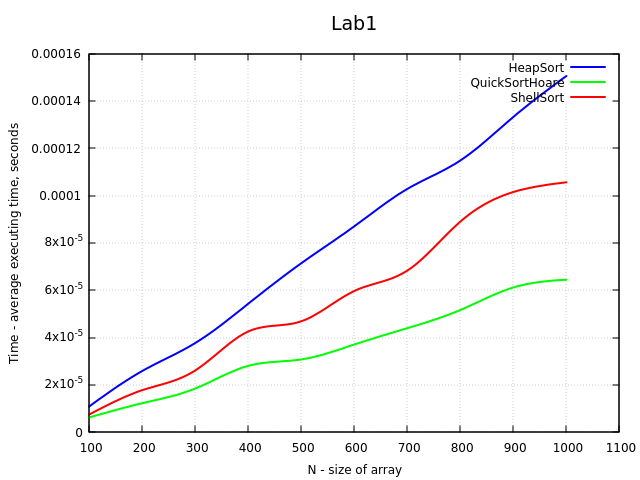
1. В цикле с уровнем вложенности **0** выбирается начальный шаг (размер массива пополам), через который отстоят элементы в каждом подсписке, на которые (умозрительно) разбивается массив, и которые не пересекаются друг с другом. В циклическом выражении цикла for шаг уменьшается в 2 раза.
2. Цикл с уровнем вложенности **1** выполняет операции над подсписками. Т.е. каждая его итерация связана с сортировкой некоторого подсписка. Но сортировка каждого подсписка может и не выполнятся за одну итерацию. Т.е. одной итерации соответствует один подсписок, но одному подсписку может соответствовать несколько итераций этого цикла.
3. Цикл с уровнем вложенности **2**. В этом цикле происходит сортировка части подсписка. После того, как этот цикл будет выполнен целиком, окажется упорядочена некоторая часть подсписка (его начало). Сколько конкретно будет упорядоченно элементов в начальной части массива зависит от текущего положения счетчика цикла с уровнем вложенности **1**.
4. Сравнительный анализ

|  |
| --- |
| HeapSort:  ===Arr (before sort):===  Arr[0]=87  Arr[1]=6  Arr[2]=28  Arr[3]=26  Arr[4]=78  BuildHeap Started!  BuildHeap: element will be Heapified:2  Heapify started!  ===Arr (Before Heapify):===  Arr[0]=87  Arr[1]=6  Arr[2]=28  Arr[3]=26  Arr[4]=78  Heapify finished!  ===Arr (After Heapify):===  Arr[0]=87  Arr[1]=6  Arr[2]=28  Arr[3]=26  Arr[4]=78  BuildHeap: element will be Heapified:1  Heapify started!  ===Arr (Before Heapify):===  Arr[0]=87  Arr[1]=6  Arr[2]=28  Arr[3]=26  Arr[4]=78  -Heapify started!  -===Arr (Before Heapify):===  -Arr[0]=87  -Arr[1]=78  -Arr[2]=28  -Arr[3]=26  -Arr[4]=6  -Heapify finished!  -===Arr (After Heapify):===  -Arr[0]=87  -Arr[1]=78  -Arr[2]=28  -Arr[3]=26  -Arr[4]=6  Heapify finished!  ===Arr (After Heapify):===  Arr[0]=87  Arr[1]=78  Arr[2]=28  Arr[3]=26  Arr[4]=6  BuildHeap: element will be Heapified:0  Heapify started!  ===Arr (Before Heapify):===  Arr[0]=87  Arr[1]=78  Arr[2]=28  Arr[3]=26  Arr[4]=6  Heapify finished!  ===Arr (After Heapify):===  Arr[0]=87  Arr[1]=78  Arr[2]=28  Arr[3]=26  Arr[4]=6  BuildHeap Finished!  ===Arr (After BuildHeap):===  Arr[0]=87  Arr[1]=78  Arr[2]=28  Arr[3]=26  Arr[4]=6  Heapify started!  ===Arr (Before Heapify):===  Arr[0]=6  Arr[1]=78  Arr[2]=28  Arr[3]=26  -Heapify started!  -===Arr (Before Heapify):===  -Arr[0]=78  -Arr[1]=6  -Arr[2]=28  -Arr[3]=26  - -Heapify started!  - -===Arr (Before Heapify):===  - -Arr[0]=78  - -Arr[1]=26  - -Arr[2]=28  - -Arr[3]=6  - -Heapify finished!  - -===Arr (After Heapify):===  - -Arr[0]=78  - -Arr[1]=26  - -Arr[2]=28  - -Arr[3]=6  -Heapify finished!  -===Arr (After Heapify):===  -Arr[0]=78  -Arr[1]=26  -Arr[2]=28  -Arr[3]=6  Heapify finished!  ===Arr (After Heapify):===  Arr[0]=78  Arr[1]=26  Arr[2]=28  Arr[3]=6  Heapify started!  ===Arr (Before Heapify):===  Arr[0]=6  Arr[1]=26  Arr[2]=28  -Heapify started!  -===Arr (Before Heapify):===  -Arr[0]=28  -Arr[1]=26  -Arr[2]=6  -Heapify finished!  -===Arr (After Heapify):===  -Arr[0]=28  -Arr[1]=26  -Arr[2]=6  Heapify finished!  ===Arr (After Heapify):===  Arr[0]=28  Arr[1]=26  Arr[2]=6  Heapify started!  ===Arr (Before Heapify):===  Arr[0]=6  Arr[1]=26  -Heapify started!  -===Arr (Before Heapify):===  -Arr[0]=26  -Arr[1]=6  -Heapify finished!  -===Arr (After Heapify):===  -Arr[0]=26  -Arr[1]=6  Heapify finished!  ===Arr (After Heapify):===  Arr[0]=26  Arr[1]=6  Heapify started!  ===Arr (Before Heapify):===  Arr[0]=6  Heapify finished!  ===Arr (After Heapify):===  Arr[0]=6  ===Arr (after sort):===  Arr[0]=6  Arr[1]=26  Arr[2]=28  Arr[3]=78  Arr[4]=87  CurrentSize=5; Iteration 0 is complete  QuickSortHoare:  ===Arr (before sort):===  Arr[0]=87  Arr[1]=6  Arr[2]=28  Arr[3]=26  Arr[4]=78  pivot is Arr[2]=p=28  Current i:(Arr[i] !< p) =0; Current j:(Arr[j] !> p) =3  ===Arr (After swap):===  Arr[0]=26  Arr[1]=6  Arr[2]=28  Arr[3]=87  Arr[4]=78  Current i:(Arr[i] !< p) =2; Current j:(Arr[j] !> p) =2  ===Arr (After swap):===  Arr[0]=26  Arr[1]=6  Arr[2]=28  Arr[3]=87  Arr[4]=78  pivot is Arr[1]=p=6  Current i:(Arr[i] !< p) =0; Current j:(Arr[j] !> p) =1  -===Arr (After swap):===  -Arr[0]=6  -Arr[1]=26  pivot is Arr[1]=p=78  Current i:(Arr[i] !< p) =0; Current j:(Arr[j] !> p) =1  -===Arr (After swap):===  -Arr[0]=78  -Arr[1]=87  ===Arr (after sort):===  Arr[0]=6  Arr[1]=26  Arr[2]=28  Arr[3]=78  Arr[4]=87  CurrentSize=5; Iteration 0 is complete  ShellSort:  ===Arr (before sort):===  Arr[0]=87  Arr[1]=6  Arr[2]=28  Arr[3]=26  Arr[4]=78  ===Arr (After Step s=2):===  Arr[0]=28  Arr[1]=6  Arr[2]=78  Arr[3]=26  Arr[4]=87  ===Arr (After Step s=1):===  Arr[0]=6  Arr[1]=26  Arr[2]=28  Arr[3]=78  Arr[4]=87  ===Arr (after sort):===  Arr[0]=6  Arr[1]=26  Arr[2]=28  Arr[3]=78  Arr[4]=87  CurrentSize=5; Iteration 0 is complete |

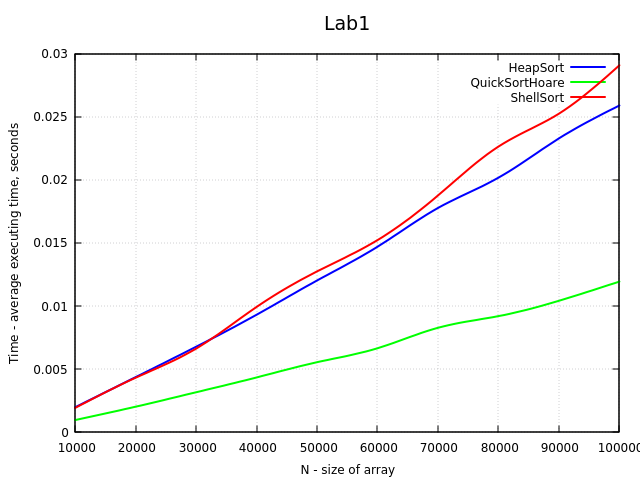
Пошаговое выполнение для массива размера 5

В таблице 1 показано пошаговое выполнение алгоритмов сортировки. Текущий уровень рекурсии обозначен «-».

1. Результаты работы



Время работы от размера массива



Время работы от размера массива.

Как видно из графиков, наиболее быстрой сортировкой оказалась быстрая сортировка Хоара. Сортировка двоичной кучей по времени работы сопоставима с сортировкой Шелла. Сортировка двоичной кучей оказалась медленнее соритровки Шелла на массивах меньшего размера(до 40000 элементов) и быстрее на массивах большего размера (40000 элементов и более).

При n=10^4

* t\_HeapSort/t\_QuickSortHoare= 2,03731037310373
* t\_ShellSort/t\_QuickSortHoare= 1,98236982369824
* t\_ShellSort/t\_HeapSort=0,973032803380962

При n=10^5

* t\_HeapSort/t\_QuickSortHoare=2,16707832712354
* t\_ShellSort/t\_QuickSortHoare=2,43336542889438
* t\_ShellSort/t\_HeapSort=1,1228783927364

1. Выводы

В ходе проведенной работы были изучены алгоритмы сортировки двоичной кучей, быстрой сортировки Хоара, сортировки Шелла. Была освоена программа gnuplot, были расширены знания по использованию утилиты CMake. Был разработан шаблонный класс для сбора статистики с функций сортировки, а также функции для сортировок двоичной кучей, быстрой сортировкой Хоара, сортировкой Шелла. Анализ показал, что алгоритм быстрой сортировки Хоара значительно быстрее алгоритмов сортировки двоичной кучей и Шелла.