Министерство образования и науки РФ  
Федеральное государственное бюджетное образовательное учреждение   
Высшего образования  
<<Московский технологический университет>>  
МИРЭА

ИНСТИТУТ ИФОРМАЦИОННЫХ ТЕХНОЛОГИЙ

**Отчет по лабораторной работе №2**

**По дисциплине  
<<Анализ сложности алгоритмов>>**

Студент группы ИКБО-08-16 Кижнер.Л.Б.

Преподаватель Мирабо Е.И.

Отчет принят и проверен <<\_\_>>\_\_\_\_\_\_\_\_\_201\_\_г.

Москва 2017

**СОДЕРЖАНИЕ:**

ВВЕДЕНИЕ ……………………………….……………………………………………………………………………….3

1.Постановка задачи..……………………………………………………………………………………………...4

2.Теоретический раздел………………………………………………………………………………………….5

2.1. Сортировка простым обменом..…………………………………………………………..5

2.2. Быстрая Сортировка Хоара……………………………………………………………………6

2.3. Сортировка вставками…………..………………………………………………………………7

2.4. Сортировка с помощью двоичного дерева ………………………………………….8

2.5 Линейный поиск………………………………………………………………………………………9

2.6 Интерполяционный поиск …………………………………………………………………….10

2.7 Двоичное дерево поиска ……………………………………………………………………….11

3.Практический раздел ..………………………………………………………………………………………….12  
 3.1 Алгоритмы сортировки и поиска ….………………………………………………………..12

ЗАКЛЮЧЕНИЕ……………………………………………………………………………………………………………..16

**ВВЕДЕНИЕ**

Анализ сложности алгоритмов – это область теоретической информатики (theoretical computer science) и затрагивает вопросы не только со стороны теории , но так же и со стороны практики.  
Из наиболее важных приложений данной теории можно назвать способы построения и анализа эффективных алгоритмов.  
 Количественная характеристика потребляемых ресурсов , необходимых программе или алгоритму для работы(успешного решения задачи) – это и есть сложность алгоритма.  
 Основные ресурсы: время и объем памяти. Время более важный ресурс , нежели объём.  
Очевидно , что для разных входных значений алгоритму может потребоваться разное время (количество ресурсов) для решения задачи.  
 Цель данной работы заключается в использовании популярных алгоритмов сортировки на языке программирования и определения наиболее эффективных среди предложенных.

**1.ПОСТАНОВКА ЗАДАЧИ**

Анализ вычислительной сложности алгоритма сортировки массива методом простого обмена, быстрой сортировки, сортировки вставками , сортировки по двоичному дереву. Реализовать и исследовать следующие алгоритмы: линейный поиск.

В программе должны быть реализованы два варианта формирования массива(выбирается в начале программы):

1)ввод с клавиатуры(для тестового прогона программы), n = 10.

2)с помощью генератора псевдослучайных чисел (для рабочего прогона программы)  
 n = 10000, n = 30000 , n = 50000 , n = 70000 и n = 90000.

3)Провести теоретическую и практическую оценки вычислительной сложности алгоритмов.

Примечаний:

1)Формирование исходного массива из псевдослучайных чисел оформить в виде функции.

2)Практическая оценка вычислительной сложности алгоритма производится путем вычисления количества выполненных операций сравнения С и перемещения М.

**2.ТЕОРИТИЧЕСКИЙ РАЗДЕЛ**

**2.1 Сортировка простым обменом**

Сортировка простыми обменами - алгоритм сортировки прямым обменом основан на принципе сравнения и обмена пары соседних элементов до тех пор, пока не будут отсортированы все элементы. Как и в методе [прямого выбора](https://prog-cpp.ru/sort-select/), совершаются проходы по массиву, сдвигая каждый раз наименьший элемент оставшейся последовательности к началу массива.

Если рассматривать массивы как вертикальные, а не горизонтальные построения, то элементы можно интерпретировать как пузырьки в банке с водой, причем вес каждого соответствует его ключу. В этом случае при каждом проходе один пузырек как бы поднимается до уровня, соответствующего его весу. Такой метод известен под именем «пузырьковая сортировка».

Анализ алгоритма

Число сравнений в алгоритме прямого обмена

**С = (n2 - n)/2**,

а минимальное, среднее и максимальное число перемещений элементов равно соответственно

**Mmin = 0,**

**Mср = 3(n2 - n)/2,**

**Mmax = 3(n2 - n)/4**

**2.2 Быстрая сортировка, сортировка Хоара**

**Быстрая сортировка**, **сортировка Хоара** ([англ.](https://ru.wikipedia.org/wiki/%D0%90%D0%BD%D0%B3%D0%BB%D0%B8%D0%B9%D1%81%D0%BA%D0%B8%D0%B9_%D1%8F%D0%B7%D1%8B%D0%BA) *quicksort*), часто называемая ***qsort*** (по имени в стандартной [библиотеке](https://ru.wikipedia.org/wiki/%D0%91%D0%B8%D0%B1%D0%BB%D0%B8%D0%BE%D1%82%D0%B5%D0%BA%D0%B0_(%D0%BF%D1%80%D0%BE%D0%B3%D1%80%D0%B0%D0%BC%D0%BC%D0%B8%D1%80%D0%BE%D0%B2%D0%B0%D0%BD%D0%B8%D0%B5)) языка [Си](https://ru.wikipedia.org/wiki/%D0%A1%D0%B8_(%D1%8F%D0%B7%D1%8B%D0%BA_%D0%BF%D1%80%D0%BE%D0%B3%D1%80%D0%B0%D0%BC%D0%BC%D0%B8%D1%80%D0%BE%D0%B2%D0%B0%D0%BD%D0%B8%D1%8F))) — широко известный [алгоритм сортировки](https://ru.wikipedia.org/wiki/%D0%90%D0%BB%D0%B3%D0%BE%D1%80%D0%B8%D1%82%D0%BC_%D1%81%D0%BE%D1%80%D1%82%D0%B8%D1%80%D0%BE%D0%B2%D0%BA%D0%B8), разработанный английским информатиком [Чарльзом Хоаром](https://ru.wikipedia.org/wiki/%D0%A5%D0%BE%D0%B0%D1%80,_%D0%A7%D0%B0%D1%80%D0%BB%D1%8C%D0%B7_%D0%AD%D0%BD%D1%82%D0%BE%D0%BD%D0%B8_%D0%A0%D0%B8%D1%87%D0%B0%D1%80%D0%B4) во время его работы в [МГУ](https://ru.wikipedia.org/wiki/%D0%9C%D0%93%D0%A3) в [1960 году](https://ru.wikipedia.org/wiki/1960_%D0%B3%D0%BE%D0%B4).

[Один из самых быстрых](https://ru.wikipedia.org/wiki/%D0%A2%D0%B5%D0%BE%D1%80%D0%B8%D1%8F_%D1%81%D0%BB%D0%BE%D0%B6%D0%BD%D0%BE%D1%81%D1%82%D0%B8_%D0%B2%D1%8B%D1%87%D0%B8%D1%81%D0%BB%D0%B5%D0%BD%D0%B8%D0%B9) известных универсальных алгоритмов сортировки массивов: в среднем  обменов при упорядочении  элементов; из-за наличия ряда недостатков на практике обычно используется с некоторыми доработками.

QuickSort является существенно улучшенным вариантом алгоритма сортировки с помощью прямого обмена (его варианты известны как «[Пузырьковая сортировка](https://ru.wikipedia.org/wiki/%D0%9F%D1%83%D0%B7%D1%8B%D1%80%D1%8C%D0%BA%D0%BE%D0%B2%D0%B0%D1%8F_%D1%81%D0%BE%D1%80%D1%82%D0%B8%D1%80%D0%BE%D0%B2%D0%BA%D0%B0)» и «Шейкерная сортировка»), известного, в том числе, своей низкой эффективностью. Принципиальное отличие состоит в том, что в первую очередь производятся перестановки на наибольшем возможном расстоянии и после каждого прохода элементы делятся на две независимые группы. Любопытный факт: улучшение самого неэффективного прямого метода сортировки дало в результате один из наиболее эффективных улучшенных методов.

Общая идея алгоритма состоит в следующем:

* Выбрать из массива элемент, называемый опорным. Это может быть любой из элементов массива. От выбора опорного элемента не зависит корректность алгоритма, но в отдельных случаях может сильно зависеть его эффективность .
* Сравнить все остальные элементы с опорным и переставить их в массиве так, чтобы разбить массив на три непрерывных отрезка, следующие друг за другом: «меньшие опорного», «равные» и «большие».
* Для отрезков «меньших» и «больших» значений выполнить рекурсивно ту же последовательность операций, если длина отрезка больше единицы.

**Лучший случай.**

**O(n\*log2 n)**

**2.3 Сортировка вставками**

**Сортировка вставками** ([англ.](https://ru.wikipedia.org/wiki/%D0%90%D0%BD%D0%B3%D0%BB%D0%B8%D0%B9%D1%81%D0%BA%D0%B8%D0%B9_%D1%8F%D0%B7%D1%8B%D0%BA) *Insertion sort*) — [алгоритм сортировки](https://ru.wikipedia.org/wiki/%D0%90%D0%BB%D0%B3%D0%BE%D1%80%D0%B8%D1%82%D0%BC_%D1%81%D0%BE%D1%80%D1%82%D0%B8%D1%80%D0%BE%D0%B2%D0%BA%D0%B8), в котором элементы входной последовательности просматриваются по одному, и каждый новый поступивший элемент размещается в подходящее место среди ранее упорядоченных элементов. [Вычислительная сложность](https://ru.wikipedia.org/wiki/%D0%92%D1%8B%D1%87%D0%B8%D1%81%D0%BB%D0%B8%D1%82%D0%B5%D0%BB%D1%8C%D0%BD%D0%B0%D1%8F_%D1%81%D0%BB%D0%BE%D0%B6%D0%BD%D0%BE%D1%81%D1%82%D1%8C) —  **O(n^2)**

На вход алгоритма подаётся последовательность N чисел: a1 , a2 , …,an . Сортируемые числа также называют *ключами*. Входная последовательность на практике представляется в виде массива с N элементами. На выходе алгоритм должен вернуть перестановку исходной последовательности a’1 , a’2, ……,a’n, чтобы выполнялось следующее соотношение a’1 <= a’2 <= a’3 <= … <= a’n

В начальный момент отсортированная последовательность пуста. На каждом шаге алгоритма выбирается один из элементов входных данных и помещается на нужную позицию в уже отсортированной последовательности до тех пор, пока набор входных данных не будет исчерпан. В любой момент времени в отсортированной последовательности элементы удовлетворяют требованиям к выходным данным алгоритма.

**2.4 Сортировка с помощью двоичного дерева**

**Сортировка с помощью двоичного дерева** (сортировка двоичным деревом, сортировка деревом, древесная сортировка, сортировка с помощью бинарного дерева, [англ.](https://ru.wikipedia.org/wiki/%D0%90%D0%BD%D0%B3%D0%BB%D0%B8%D0%B9%D1%81%D0%BA%D0%B8%D0%B9_%D1%8F%D0%B7%D1%8B%D0%BA) *tree sort*) — универсальный [алгоритм сортировки](https://ru.wikipedia.org/wiki/%D0%90%D0%BB%D0%B3%D0%BE%D1%80%D0%B8%D1%82%D0%BC_%D1%81%D0%BE%D1%80%D1%82%D0%B8%D1%80%D0%BE%D0%B2%D0%BA%D0%B8), заключающийся в построении [двоичного дерева поиска](https://ru.wikipedia.org/wiki/%D0%94%D0%B2%D0%BE%D0%B8%D1%87%D0%BD%D0%BE%D0%B5_%D0%B4%D0%B5%D1%80%D0%B5%D0%B2%D0%BE_%D0%BF%D0%BE%D0%B8%D1%81%D0%BA%D0%B0) по ключам массива (списка), с последующей сборкой результирующего массива путём обхода узлов построенного дерева в необходимом порядке следования ключей. Данная сортировка является оптимальной при получении данных путём непосредственного чтения с [потока](https://ru.wikipedia.org/wiki/%D0%9F%D0%BE%D1%82%D0%BE%D0%BA_%D0%B4%D0%B0%D0%BD%D0%BD%D1%8B%D1%85) (например из файла, сокета или консоли).

**Алгоритм.**

1. Построение двоичного дерева.

2. Сборка результирующего массива путём обхода узлов в необходимом порядке следования ключей.

**Эффективность.**

Процедура добавления объекта в бинарное дерево имеет среднюю алгоритмическую сложность порядка O(log(n)). Соответственно, для n объектов сложность будет составлять O(n log(n)), что относит сортировку с помощью двоичного дерева к группе «быстрых сортировок». Однако, сложность добавления объекта в разбалансированное дерево может достигать O(n), что может привести к общей сложности порядка O(n²).

При физическом развёртывании древовидной структуры в памяти требуется не менее чем 4n ячеек дополнительной памяти (каждый узел должен содержать ссылки на элемент исходного массива, на родительский элемент, на левый и правый лист), однако, существуют способы уменьшить требуемую дополнительную память.

**2.5 Линейный поиск**

**Линейный, последовательный поиск** — [алгоритм](https://ru.wikipedia.org/wiki/%D0%90%D0%BB%D0%B3%D0%BE%D1%80%D0%B8%D1%82%D0%BC) нахождения заданного значения произвольной функции на некотором отрезке. Данный алгоритм является простейшим алгоритмом поиска и, в отличие, например, от [двоичного поиска](https://ru.wikipedia.org/wiki/%D0%94%D0%B2%D0%BE%D0%B8%D1%87%D0%BD%D1%8B%D0%B9_%D0%BF%D0%BE%D0%B8%D1%81%D0%BA), не накладывает никаких ограничений на функцию и имеет простейшую реализацию. Поиск значения функции осуществляется простым сравнением очередного рассматриваемого значения (как правило, поиск происходит слева направо, то есть от меньших значений аргумента к большим) и, если значения совпадают (с той или иной точностью), то поиск считается завершённым.

Если отрезок имеет длину N, то найти решение с точностью до O(n) можно за время . Т.о. [асимптотическая сложность алгоритма](https://ru.wikipedia.org/wiki/%D0%A1%D0%BB%D0%BE%D0%B6%D0%BD%D0%BE%D1%81%D1%82%D1%8C_%D0%B0%D0%BB%D0%B3%D0%BE%D1%80%D0%B8%D1%82%D0%BC%D0%B0) — N\e. В связи с малой эффективностью по сравнению с другими алгоритмами линейный поиск обычно используют, только если отрезок поиска содержит очень мало элементов, тем не менее, линейный поиск не требует дополнительной памяти или обработки/анализа функции, так что может работать в потоковом режиме при непосредственном получении данных из любого источника. Также линейный поиск часто используется в виде линейных алгоритмов поиска максимума/минимума.

**2.6** **Интерполяционный поиск**

**Интерполяционный поиск** (**интерполирующий поиск**) основан на принципе поиска в телефонной книге или, например, в словаре. Вместо сравнения каждого элемента с искомым, как при [линейном поиске](https://ru.wikipedia.org/wiki/%D0%9B%D0%B8%D0%BD%D0%B5%D0%B9%D0%BD%D1%8B%D0%B9_%D0%BF%D0%BE%D0%B8%D1%81%D0%BA), данный алгоритм производит предсказание местонахождения элемента: поиск происходит подобно [двоичному поиску](https://ru.wikipedia.org/wiki/%D0%94%D0%B2%D0%BE%D0%B8%D1%87%D0%BD%D1%8B%D0%B9_%D0%BF%D0%BE%D0%B8%D1%81%D0%BA), но вместо деления области поиска на две части, интерполирующий поиск производит оценку новой области поиска по расстоянию между ключом и текущим значением элемента. Другими словами, бинарный поиск учитывает лишь знак разности между ключом и текущим значением, а интерполирующий ещё учитывает и модуль этой разности и по данному значению производит предсказание позиции следующего элемента для проверки. В среднем интерполирующий поиск производит log(log(*N*)) операций, где *N* есть число элементов, среди которых производится поиск. Число необходимых операций зависит от равномерности распределения значений среди элементов. В плохом случае (например, когда значения элементов экспоненциально возрастают) интерполяционный поиск может потребовать до O(*N*) операций.

**2.7 Двоичное дерево поиска**

**Двоичное дерево поиска** ([англ.](https://ru.wikipedia.org/wiki/%D0%90%D0%BD%D0%B3%D0%BB%D0%B8%D0%B9%D1%81%D0%BA%D0%B8%D0%B9_%D1%8F%D0%B7%D1%8B%D0%BA) *binary search tree*, BST) — это [двоичное дерево](https://ru.wikipedia.org/wiki/%D0%94%D0%B2%D0%BE%D0%B8%D1%87%D0%BD%D0%BE%D0%B5_%D0%B4%D0%B5%D1%80%D0%B5%D0%B2%D0%BE), для которого выполняются следующие дополнительные условия (*свойства дерева поиска*):

* Оба поддерева — левое и правое — являются двоичными деревьями поиска.
* У всех узлов *левого* поддерева произвольного узла X значения ключей данных *меньше*, нежели значение ключа данных самого узла X.
* У всех узлов *правого* поддерева произвольного узла X значения ключей данных *больше либо равно*, нежели значение ключа данных самого узла X.

Очевидно, данные в каждом узле должны обладать ключами, на которых определена операция сравнения *меньше*.

Как правило, информация, представляющая каждый узел, является записью, а не единственным полем данных. Однако это касается реализации, а не природы двоичного дерева поиска.

Для целей реализации двоичное дерево поиска можно определить так:

* Двоичное дерево состоит из узлов (вершин) — записей вида (data, left, right), где data — некоторые данные, привязанные к узлу, left и right — ссылки на узлы, являющиеся детьми данного узла — левый и правый сыновья соответственно. Для оптимизации алгоритмов конкретные реализации предполагают также определения поля parent в каждом узле (кроме корневого) — ссылки на родительский элемент.
* Данные (data) обладают ключом (key), на котором определена операция сравнения «меньше». В конкретных реализациях это может быть пара (key, value) — (ключ и значение), или ссылка на такую пару, или простое определение операции сравнения на необходимой структуре данных или ссылке на неё.
* Для любого узла X выполняются свойства дерева поиска: key[left[X]] < key[X] ≤ key[right[X]], то есть ключи данных родительского узла больше ключей данных левого сына и нестрого меньше ключей данных правого.

**ПРАКТИЧЕСКИЙ РАЗДЕЛ**

**3.1 Алгоритмы сортировки и поиска**

В процессе завершения работы над реализацией алгоритмов была произведена оценка алгоритмов. Были зафиксированы след параметры: время(time)

Функция случайной генерации чисел создают массив в диапазоне от 0 до 100.

В результате сравнения при разном наборе данных была получена следующая таблица:  
 Таблица 1.Оценка практической работы алгоритмов сортировки

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
|  | Пузырьковая  сортировка | <<Быстрая сортировка>> | Сортировка  вставками | Сортировка по  бинарному дереву |

N = 10000 элементов

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Время выполнения(сек) | 34.928 | 0.131 | 29.448 | 0.236 |
| Кол-во сравнений | 23 854 321 | 59 821 | 24 672 912 | 203 841 |
| Кол-во перемещений | 49 995 000 | 91 924 | 24 940 636 | 209 976 |

N = 30000 элементов

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Время выполнения(сек) | 374.903 | 0.723 | 300.823 | 1.275 |
| Кол-во сравнений | 226 434 810 | 193 219 | 226 295 721 | 1 386 092 |
| Кол-во перемещений | 449 985 000 | 304 842 | 226 433 734 | 1 247 695 |

N = 50000 элементов

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Время выполнения(сек) | 930.635 | 1.621 | 758.253 | 2.482 |
| Кол-во сравнений | 622 909 427 | 334 624 | 622 909 427 | 3 114 032 |
| Кол-во перемещений | 1 249 975 000 | 528 453 | 622 959 426 | 3 114 034 |

N = 70000 элементов

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Время выполнения(сек) | 1740.486 | 2.374 | 28 788.991 | 2.939 |
| Кол-во сравнений | 1 217 092 767 | 494 941 | 1 217 092 767 | 5 717 240 |
| Кол-во перемещений | 2 449 965 000 | 761 344 | 1 217 162 766 | 5 717 242 |

N = 90000 элементов

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Время выполнения(сек) | 3251.934 | 2.812 | 69 235.932 | 3.395 |
| Кол-во сравнений | 2 025 054 213 | 633 826 | 2 025 054 213 | 9 267 573 |
| Кол-во перемещений | 4 049 955 000 | 991 390 | 2 025 144 212 | 9 267 575 |

Таблица 2.Оценка практической работы алгоритмов поиска.

|  |  |  |  |
| --- | --- | --- | --- |
|  | Линейный поиск | Интерполяционный поиск | Поиск по двоичному дереву |

Для n = 10000

|  |  |  |  |
| --- | --- | --- | --- |
| Время выполнения (сек) | 0.031 | 0.029 | 0.001 |
| Кол-во сравнений | 0 | 10 | 8 |
| Кол-во перемещений | 10000 | 10 009 | 7 |

Для n = 30000

|  |  |  |  |
| --- | --- | --- | --- |
| Время выполнения (сек) | 0.874 | 0.034 | 0.006 |
| Кол-во сравнений | 0 | 10 | 8 |
| Кол-во перемещений | 30 000 | 30 008 | 9 |

Для n = 50000

|  |  |  |  |
| --- | --- | --- | --- |
| Время выполнения (сек) | 3.043 | 0.039 | 0.011 |
| Кол-во сравнений | 0 | 10 | 5 |
| Кол-во перемещений | 50 000 | 50 009 | 6 |

Для n = 70000

|  |  |  |  |
| --- | --- | --- | --- |
| Время выполнения (сек) | 4.241 | 0.041 | 0.012 |
| Кол-во сравнений | 0 | 9 | 8 |
| Кол-во перемещений | 70 000 | 70 008 | 9 |

Для n = 90000

|  |  |  |  |
| --- | --- | --- | --- |
| Время выполнения (сек) | 8.437 | 0.044 | 0.015 |
| Кол-во сравнений | 0 | 10 | 5 |
| Кол-во перемещений | 90 000 | 90 008 | 6 |

**Заключение**

В результате изучения методов сортировок, применения их и сравнения были выявлены наиболее эффективные алгоритмы сортировок и поиска.

При выполнении лабораторной работы наиболее эффективным алгоритмом сортировки оказалась <<Быстрая>> Сортировка Хоара, а наиболее эффективным алгоритмом поиска оказался поиск по бинарному дереву.

Эти алгоритмы стали лучшими, т. к. они показали наименьшее время, а также использовали наименьшее количество итераций и сравнений (в зависимости от размера массива).