Федеральное государственное автономное образовательное учреждение высшего образования

НАЦИОНАЛЬНЫЙ ИССЛЕДОВАТЕЛЬСКИЙ УНИВЕРСИТЕТ

«ВЫСШАЯ ШКОЛА ЭКОНОМИКИ»

Факультет информатики, математики и компьютерных наук

КУРСОВАЯ РАБОТА

Сравнение эффективности алгоритмов сортировок целочисленных массивов

по направлению подготовки 09.03.04 "Программная инженерия" образовательная программа «Программная инженерия»

Выполнил:

Студент группы 19ПИ-1

Долгополов Алексей Геннадьевич

Научный руководитель:

Ефименков Вячеслав Валерьевич

Нижний Новгород 2020

Содержание

1. Введение

В современной действительности алгоритмы сортировок, предполагающие набор операций для упорядочивания элементов в списке, широко применяются и играют важную роль в программировании. Однако далеко не многие программисты задумываются, какой алгоритм работает эффективнее, а ведь каждый по-разному реагирует на различные входные данные. По моему мнению, лучшим способом выявить и сравнить эффективность будет сопоставление времени выполнения худшего случая для каждой сортировки.

В данной работе я собираюсь рассмотреть популярные алгоритмы сортировок, представить их краткое описание, а затем провести непосредственное тестирование, итоги которого будут занесены в таблицу, и, естественно, сделать окончательные выводы.

2. Постановка задачи

Передо мной стоит задача- разработать программу на языке С, которая выявит, какие данные являются для представленных сортировок наихудшими.

3. Сложность алгоритмов

Чтобы понять какие именно данные будут наиболее “трудными” для сортировок, необходимо узнать про асимптотическую сложность предоставленных алгоритмов сортировки, то есть сложность, когда размер входных данных стремится к бесконечности.

O – нотация:

Этот термин изначально появился в математике, и используется для сравнения асимптотического поведения функций. Формула O(f(n)) даёт приблизительный ответ на вопрос зависимости времени и памяти, используемых программой, от объема входных данных.

Каждый алгоритм сортировки имеет свою асимптотическую сложность для лучшего, среднего и худшего случая:

	Лучший случай	Средний случай	Худший случай
Insertion Sort	O(n)	O(n^2)	O(n^2)
Selection Sort	O(n^2)	O(n^2)	O(n^2)
Bubble Sort	O(n)	O(n^2)	O(n^2)
Bucket Sort	O(n+k)	O(n+k)	O(n^2)
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))

4. Предметная область

В работе используются 6 алгоритмов сортировки, за реализацию которых отвечала Вотинова Ксения. Ссылка на её репозиторий Github в списке литературы.

Для данной работы нам совершенно не важны потребление памяти и понятие устойчивости сортировок. Заострим внимание лишь на работе алгоритмов.

4.1 Insertion sort

1. Выбираем произвольный элемент входных данных.

2. Ставим его на нужную позицию в списке, который уже отсортирован.

3. Следует повторять операцию, пока не закончится набор входных данных.

4. Произвольный элемент выбирается любым удобным методом.

4.2 Selection sort

1. Находим минимальный элемент в списке.
2. Ставим этот элемент на первую позицию.
3. Сортируем оставшиеся элементы, исключив ранее упорядоченные.
4. Выполняем итерации до тех пор, пока не установится правильный порядок элементов.

4.3 Bubble sort

1. Проходим по сортируемому массиву.
2. Сравниваем два соседних элемента.
3. Если обнаруживается неверный порядок в паре, элементы меняются местами.
4. Проходим по массиву до тех пор, пока не установится правильный порядок элементов.

4.4 Bucket sort

1. Разделяем интервал [0;1) на n отрезков, равных друг другу.
2. Разбиваем по этим отрезкам n входных элементов.
3. Ожидается, что в каждом «блоке» окажется небольшое количество величин.
4. Сортируем числа в каждом «блоке».
5. Последовательно перечисляя элементы каждого «блока», получим искомый упорядоченный массив.

4.5 Quick sort

1. Выбираем опорный элемент.

2. Сравниваем все остальные входные данные с опорным.

3. Разбиваем множество величин на три части: «больше опорного», «равные» и «меньше» него.

4. Размещаем их последовательно: меньшие- равные- большие. 5. Сортируем получившиеся части рекурсивно.

4.6 Merge sort

1. Делим массив пополам.

2. Отдельно проводим сортировка для двух частей.

3. Соединяем упорядоченные массивы между собой

5. Разработка решения

5.1 Описание проекта.

Проект содержит в себе 4 файла:

1. Main.c – хранит в себе основную функцию main () из которой происходит вызов остальных функций программы.
2. Benchmark.c – состоит из функций, отвечающих за замеры времени, запуск сортировок на сгенерированных тестах и вывод результата на экран.
3. Sorting.c – предоставляет программе используемые ей сортировки.
4. Tests.c – отвечает за функции заполняющие ячейки массива элементами в описанном ранее порядке.

Для получения необходимых данных используем Benchmark и Sorting, которые перекочевали из работы Вотиновой Ксении (ссылка на репозиторий которой всё также находится в списке литературы). Только Benchmark должен в данной работе претерпеть некоторые изменения, позволяющие запускать его функции с

большим количеством входных тестов. (фрагменты кода, где произошли изменения ниже)

```
    enum SORT_NAME {
            select = 0, insert, bubble, merge, bucket, quick
            };

     void PrintTable() {
            printf("\t\t\t\t\t\t\x1B[30;1mTIME ON\n");
            printf("SORT NAME");
            printf("\t\tTest 1");
            printf("\tTest 2");
            printf("\tTest 3");
            printf("\tTest 4");
            printf("\tTest 5");
            printf("\tTest 6");
            printf("\tTest 7");
            printf("\tTest 8");
            printf("\tTest 9");
            printf("\tTest 10");
            printf("\n-----------------------
            ------------------------------");
            printf("-------------------------
            ----------------------
            \n\033[0m");
    }
void PrintTimeData(const TIMER* t, int sort, int data) {
          double time = t->finish - t->start;
time = (time / CLOCKS_PER_SEC) * 1000;       if
(data == 1) printf("\x1B[33m%.0f ", time);       if
(data == 2) printf("\x1B[35m%.0f ", time);       if
(data == 3) printf("\x1B[34m%.0f ", time);       if
(data == 4) printf("\x1B[31m%.0f ", time);       if
(data == 5) printf("\x1B[36m%.0f ", time);       if
(data == 6) printf("\x1B[33m%.0f ", time);       if
(data == 7) printf("\x1B[35m%.0f ", time);       if
(data == 8) printf("\x1B[34m%.0f ", time);       if
(data == 9) printf("\x1B[31m%.0f ", time);       if
(data == 10) printf("\x1B[36m%.0f ", time);
printf("ms\033[0m");       if (time < 10000)
printf("\t");
          else printf(""); }
```

Файл Main.c содержит в себе основную функцию main() и подключение стандартных библиотек stdio.h, stdlib.h, time.h, а также benchmark.h и tests.h.

В функции main() самая увлекательная часть- удобный запуск тестов всех необходимых размеров для вышеупомянутых видов сортировок и выявление времени, затраченного на саму сортировку массива, а следом и нахождение худших типов массивов, которые используются в качестве входных данных. Здесь содержится цикл for (int iteration = 0; iteration <15; iteration++), отвечающий за повторение запуска тестов, чтобы не запускать программу по нескольку раз с разной длиной массива, а за один запуск получить по 5 запусков сортировок для каждого размера массива. Во вложенном в него цикле хранится ещё один цикл, запускающий сам процесс тестирования сортировок. (цикл ниже)

```
for (int i = 0; i < 6; i++)
{
  PrintNameSort(i);      if (i
 < 3)
                size = SIZE / 100;
                else
        size = SIZE;
        SortTests(array, size);
        Benchmark(i, array, test1, size);
        SelectBadTests(array, size);
        Benchmark(i, array, test2, size);
        BubbleBadTests(array, size);
        Benchmark(i, array, test3, size);
        InsertBadTests(array, size);
        Benchmark(i, array, test4, size);
        QuickBadTests(array, size);
        Benchmark(i, array, test5, size);
        SwapTests(array, size);
        Benchmark(i, array, test6, size);
        RandTests(array, size);
        Benchmark(i, array, test7, size);
        MiniSortTests(array, size);
        Benchmark(i, array, test8, size);
        RepeatUnrandSortTests(array, size);
        Benchmark(i, array, test9, size);
```

```
        RepeatRandSortTests(array, size);
        Benchmark(i, array, test10, size);
        printf("\n");
}
```

Таким образом, мы сможем максимально быстро и чётко найти средние значения затраченного на сортировки массивов времени и увидеть, какие же варианты входных данных вызывают трудности у разных алгоритмов сортировок.

Файл tests.c хранит в себе функции, принимающие указатель на массив и размер самого массива, при вызове которых этот массив заполняется элементами. (пример такой функции также ниже)

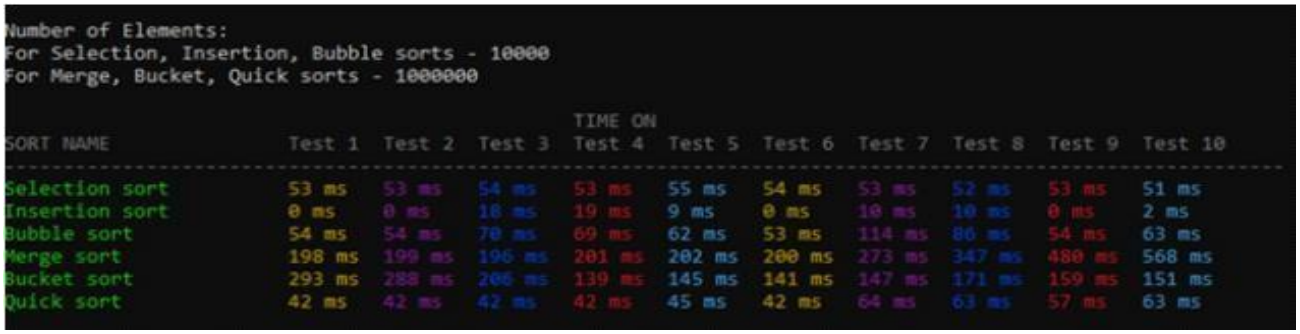
```
 void RandTests(int* array, int size)
{
        for (int i = 0; i < size; i++) array[i] = rand () % size;
        }
```

В ходе выполнения программы замерялось лишь время работы алгоритмов (в миллисекундах), не учитывая время, затраченное на генерирование входных массивов.

5.2 Входные и выходные данные

На вход программа не требует никаких манипуляций от пользователя, кроме двойного клика мышью для её запуска.

На выходе получаем 15 таблиц. На примере одной из них познакомимся с их внешним видом.



Number of Elements:
For Selection, Insertion, Bubble sorts - 10000
For Merge, Bucket, Quick sorts - 1000000

SORT NAME	TIME ON Test 1	Test 2	Test 3	Test 4	Test 5	Test 6	Test 7	Test 8	Test 9	Test 10
Selection sort	53 ms	53 ms	54 ms	53 ms	55 ms	54 ms	53 ms	52 ms	53 ms	51 ms
Insertion sort	0 ms	0 ms	18 ms	19 ms	9 ms	0 ms	10 ms	10 ms	0 ms	2 ms
Bubble sort	54 ms	54 ms	70 ms	69 ms	62 ms	53 ms	114 ms	86 ms	54 ms	63 ms
Merge sort	198 ms	199 ms	196 ms	201 ms	202 ms	200 ms	273 ms	347 ms	480 ms	568 ms
Bucket sort	293 ms	288 ms	206 ms	139 ms	145 ms	141 ms	147 ms	171 ms	159 ms	151 ms
Quick sort	42 ms	42 ms	42 ms	42 ms	45 ms	42 ms	64 ms	63 ms	57 ms	63 ms

Вверху указывается количество элементов массива на данном цикле. Затем идет таблица, в которой указаны сортировки(строки) и номера тестов(столбцы). В пересечении соответственно указано время, за которое алгоритм отсортировал сгенерированные числа.

5.3 Примеры тестов и их описание

Test 1 – отсортированный массив из size элементов. Ничего выдающегося в себе не имеет, код ниже.

```
void SortTests(int* array, int size)
{
  int now = 0;
  for (int i = 0; i < size; i++)
        {
            array[i] = now + rand() % (size / 10) + 1;
            now = array[i];
        }
}
```

Пример: 1 2 3 4 5

Test 2 – всё тот же отсортированный массив (даже заполняется с помощью SortTests), за одним исключением. Наименьший элемент поставлен в конец массива, а остальные смещены на шаг влево.

Пример: 2 3 4 5 1

Test 3 – точно такой же массив, как и в Test1, отсортированный в обратном порядке.

Пример: 5 4 3 2 1

Test 4 – точно такой же массив, как и в Test1, отсортированный в обратном порядке.

Пример: 5 4 3 2 1

Test 5 – взят отсортированный массив, а затем на четные места поставлены элементы с минимального по size/2 в возрастающем порядке, а на нечетные места с size/2 по максимальный в убывающем.

Пример: 5 1 4 2 3

Test 6 – отсортированный массив, в котором некоторые рядом стоящие элементы поменяны местами.

Пример: 1 2 4 3 5

Test 7 – случайно сгенерированный массив.

Пример: 4 2 5 1 3

Test 8 – массив, состоящий из отсортированных подмассивов.

Пример: 1 3 2 3 4

Test 9 – отсортированный массив с повторяющимися подряд элементами.

Пример: 1 2 2 2 3

Test 10 – случайный массив с повторяющимися подряд элементами.

Пример: 3 1 1 5 4

6. Тестирование программы

6.1 Техническая часть

Программа запускалась на Lenono ideaped 520, технические характеристики представлены ниже.

Процессор: Intel® Core™ i5-8250U CPU @ 1.60GHz 1.80GHz

Оперативная память: 6,00 Gb

Оперативная система: Windows 10 x64

6.2 Тесты

Производилось тестирование на массивах длиной 10^3, 10^4, 10^5, 10^6, 10^7. Исключительно с помощью таких размеров входных данных можно выявить рост времени, поскольку при меньших размерах растёт погрешность, а при больших тратится чрезмерно много времени на выполнение сортировки. Для всех алгоритмов генерировались тесты единым образом, то есть каждая сортировка упорядочила каждый из сгенерированных тестов. Для всех тестов было проведено по 5 запусков. Время, указанное в таблицах- среднее по всем итоговым значениям.

Для удобства разделим сортировки на две группы и найдём для каждой наихудший вариант входных данных. Будем делить в зависимости от асимптотической сложности в среднем случае: O(n^2) или O(n log(n)), при этом отнесём к второй группе Bucket sort со сложностью O(n+k).

Поскольку сортировки первой группы тратят время n^2, возьмём для них входные массивы с размерами: 10^3, 10^4, 10^5, иначе они будут тратить на упорядочивание элементов чрезмерно много времени. Для второй же группы входные данные будут иметь размеры: 10^5, 10^6, 10^7.

Рассмотрим результаты запусков тестов для первой группы сортировок: Selection, Insertion, Bubble.

Среднее значение для каждого из запусков (в мс):

1000 элементов в массиве:

	Test 1	Test 2	Test 3	Test 4	Test 5	Test 6	Test 7	Test 8	Test 9	Test10
Selection	0.5	0.5	0.5	0.5	0.5	0.5	0.5	0.5	0.5	0.5
Insertion	0	0	0.5	0.5	0.5	0	0	0	0	0
Bubble	0.5	0.5	0.5	0.5	0.5	0.5	0.5	0.5	0.5	0.5

Графическое представление:

Результаты показывают, что данные, полученные из тестов под номерами 3,4,5 в среднем, являются худшими для представленных сортировок. Посмотрим, что будет дальше.

10000 элементов в массиве:

	Test 1	Test 2	Test 3	Test 4	Test 5	Test 6	Test 7	Test 8	Test 9	Test10
Selection	54	54	54	54	55	54	53	52	54	51
Insertion	0	0	18	19	10	0	10	10	0	1
Bubble	54	54	70	69	62	54	114	86	54	95

Графическое представление:

В данном случае мы видим, что в среднем три выбранные сортировки показали худший результат во время выполнения тестов под номерами 3,4.

100000 элементов в массиве:

	Test 1	Test 2	Test 3	Test 4	Test 5	Test 6	Test 7	Test 8	Test 9	Test10
Selection	6815	7354	8092	7512	7185	6877	7117	7173	7051	7520
Insertion	0	0	2514	3131	1150	0	1316	1497	39	1047
Bubble	6866	6983	8790	9312	7581	7192	18959	8756	6904	11093

Графическое представление:

Несмотря на увеличение размера сгенерированного массива, три данные сортировки одинаково плохо относятся к входным данным тестов под номерами 3,4.

Рассмотрим результаты запусков тестов для второй группы сортировок: Merge, Bucket, Quick.

Среднее значение для каждого из запусков (в мс):

100000 элементов в массиве:

	Test 1	Test 2	Test 3	Test 4	Test 5	Test 6	Test 7	Test 8	Test 9	Test10
Merge	20	23	19	19	18	19	26	23	20	23

Bucket	6	6	6	10	23	6	17	16	13	14
Quick	1	1	1	1	18	1	7	6	4	5

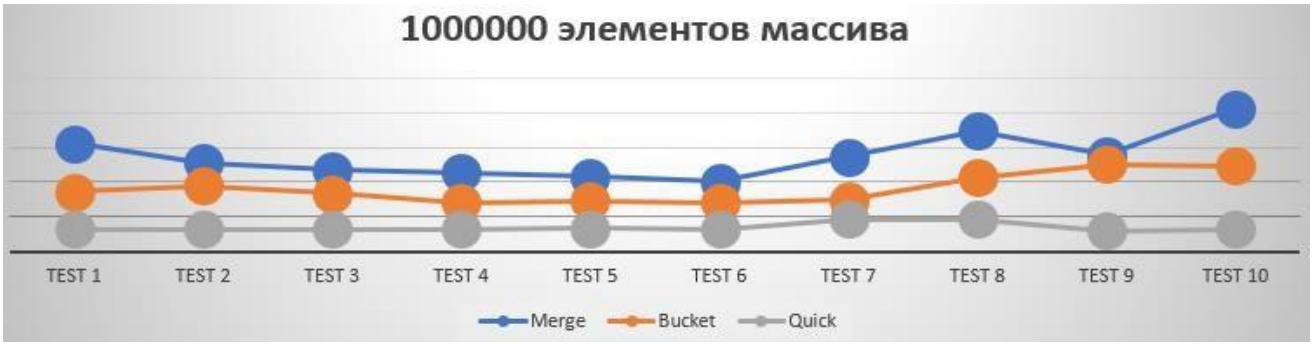
Графическое представление:

Исходя из полученных данных, можно сделать вывод, что в среднем три представленные выше сортировки выдают плохие результаты для массива, в котором на его четных местах находятся элементы с минимального по size/2 в возрастающем порядке, а на нечетных местах с size/2 по максимальный в убывающем.

1000000 элементов в массиве:

	Test 1	Test 2	Test 3	Test 4	Test 5	Test 6	Test 7	Test 8	Test 9	Test10
Merge	311	256	234	227	216	200	273	347	278	413
Bucket	172	189	168	140	145	140	147	213	251	246
Quick	65	65	65	65	67	65	93	93	59	64

Графическое представление:



По изменению графика видно, что с ростом размера входных данных, сортировки начинают испытывать трудности, упорядочивая массив с повторяющимися подряд элементами.

10000000 элементов в массиве:

	Test 1	Test 2	Test 3	Test 4	Test 5	Test 6	Test 7	Test 8	Test 9	Test10
Merge	3314	3318	3521	3282	3009	3213	4526	3787	3254	4200
Bucket	2914	2177	2903	2101	2815	2015	1712	2753	2741	2361
Quick	747	809	527	718	813	530	889	953	763	757

Графическое представление:

Представленный выше график и его данные подтверждают результат запуска для 1000000 элементов в массиве.

5. **Заключение**

Мною на языке C была разработана программа, выявившая, какие входные данные и какого размера оказались для представленных сортировок наихудшими. Ниже представлены итоги моей работы:

Отсортированный массив из любого количества элементов, тот же отсортированный массив, где наименьший элемент поставлен в конец массива, а остальные смещены на шаг влево, отсортированный массив, в котором некоторые рядом стоящие элементы поменяны местами, случайно сгенерированный массив, массив, состоящий из отсортированных подмассивов, отсортированный массив с повторяющимися подряд элементами не вызывают трудностей при выполнении сортировок у различных видов алгоритмов.

Больше всего проблем в среднем принесли следующие виды входных данных:

У Selection Sort, Insertion Sort и Bubble Sort со сложностью O(n^2) для любого количества элементов- массив, отсортированный в обратном порядке. Кроме того, массив, на четные места которого в возрастающем порядке поставлены элементы с минимального по size/2, а на нечетные- с size/2 по максимальный в убывающем, также вызвал трудности у перечисленных сортировок, но только в случае 1000 элементов входного массива.

Последний описанный входной массив является наихудшим вариантом данных для сортировок под названиями Merge, Bucket и Quick, имеющими асимптотическую сложность O(n*log(n)), исключительно в случае, когда на входе массив со 100000 элементов. Если перейти к значениям размера 10^6 и 10^7, то трудности будет вызывать сортировка массива, в котором повторяются подряд некоторые элементы.

При этом, в своих наихудших случаях эффективнее всего в первой группе алгоритмов работает Insertion Sort, а во второй группе- Quick Sort, полностью оправдывая своё название.

Стоит отметить, что не все виды алгоритмов сортировок приняли участие в данном тестировании, поскольку я выбирал только самые известные, удобные и

часто встречающиеся. Однако, мне кажется, что не сильно много от этого потеряно.

6. Список литературы

1. Алгоритмы сортировок. Википедия.
   [Электронный ресурс] URL:
   https://ru.wikipedia.org/wiki/Алгоритм_сортировки

2. Вотинова Ксения. Репозиторий
   [Электронный ресурс] URL:
   https://github.com/HSE-NN-SE/coursework-2020-Ksuvot

3. Константин Абакумов. И снова про сортировки: выбираем лучший алгоритм.
   [Электронный ресурс] URL:
   https://m.habr.com/ru/post/133996/

4. Никита Прияцелюк. Оценка сложности алгоритмов, или Что такое O(log n)
   [Электронный ресурс] URL:
   https://tproger.ru/articles/computational-complexity-explained/