



МИНОБРНАУКИ РОССИИ

*Федеральное государственное бюджетное образовательное учреждение высшего
образования*

«МИРЭА - Российский технологический университет»

РТУ МИРЭА

Отчет по выполнению практического задания № 2

Тема:

«Эмпирический анализ сложности простых алгоритмов сортировки»

Дисциплина : «Структуры и алгоритмы обработки данных»

Выполнил студент: Борзов Дмитрий Олегович

Фамилия И.О

Группа: ИКБО-12-22

Номер группы

Москва - 2023

1 ЦЕЛЬ РАБОТЫ

1.1 Цель:

Актуализация знаний и приобретение практических умений по эмпирическому определению вычислительной сложности алгоритмов.

2 ХОД РАБОТЫ

2.1 Задание 1

2.1.1 Формулировка задачи:

Оценить эмпирически вычислительную сложность алгоритма простой сортировки на массиве, заполненном случайными числами (средний случай).

1. Составить функцию простой сортировки одномерного целочисленного массива $A[n]$, используя алгоритм согласно варианту индивидуального задания (столбец Алгоритм заданий 1 и 2 в таблице 1). Провести тестирование программы на исходном массиве $n=10$.

2. Используя теоретический подход, определить для алгоритма:

а. Что будет ситуациями лучшего, среднего и худшего случаев.

б. Функции роста времени работы алгоритма от объёма входа для лучшего и худшего случаев

3. Провести контрольные прогоны программы массивов случайных чисел при $n = 100, 1000, 10000, 100000$ и 1000000 элементов с вычислением времени выполнения $T(n)$ – (в миллисекундах/секундах). Полученные результаты свести в сводную таблицу 2.

4. Провести эмпирическую оценку вычислительной сложности алгоритма, для чего предусмотреть в программе подсчет фактического количества критических операций T_p как сумму сравнений S_p и перемещений M_p . Полученные результаты вставить в сводную таблицу 2.

5. Построить график функции роста T_p этого алгоритма от размера массива n .

6. Определить ёмкостную сложность алгоритма.

7. Сделать вывод об эмпирической вычислительной сложности алгоритма на основе скорости роста функции роста.

2.1.2 Алгоритм insertion sort

Алгоритм сортировки вставками (Insertion sort) относится к сортировкам сравнения, то есть он сравнивает элементы между собой и переставляет их в нужном порядке. Алгоритм очень прост и понятен, но не является самым эффективным на больших массивах.

Алгоритм сортировки вставками работает следующим образом:

1. Начинаем со второго элемента массива (первый элемент считаем отсортированным)
2. Сравниваем текущий элемент со всеми предыдущими элементами, начиная с предыдущего
3. Если предыдущий элемент больше текущего, то перемещаем его на следующую позицию вправо
4. Повторяем шаг 3, пока не найдем элемент, меньший или равный текущему (таким образом, мы нашли место, куда нужно вставить текущий элемент)
5. Вставляем текущий элемент на найденное место
6. Повторяем шаги 2-5 для всех оставшихся элементов массива

В результате выполнения алгоритма мы получим отсортированный массив. Он будет отсортирован по возрастанию, если мы сравниваем элементы в порядке возрастания, или по убыванию, если мы сравниваем элементы в порядке убывания. Блок схема алгоритма (рис. 1).

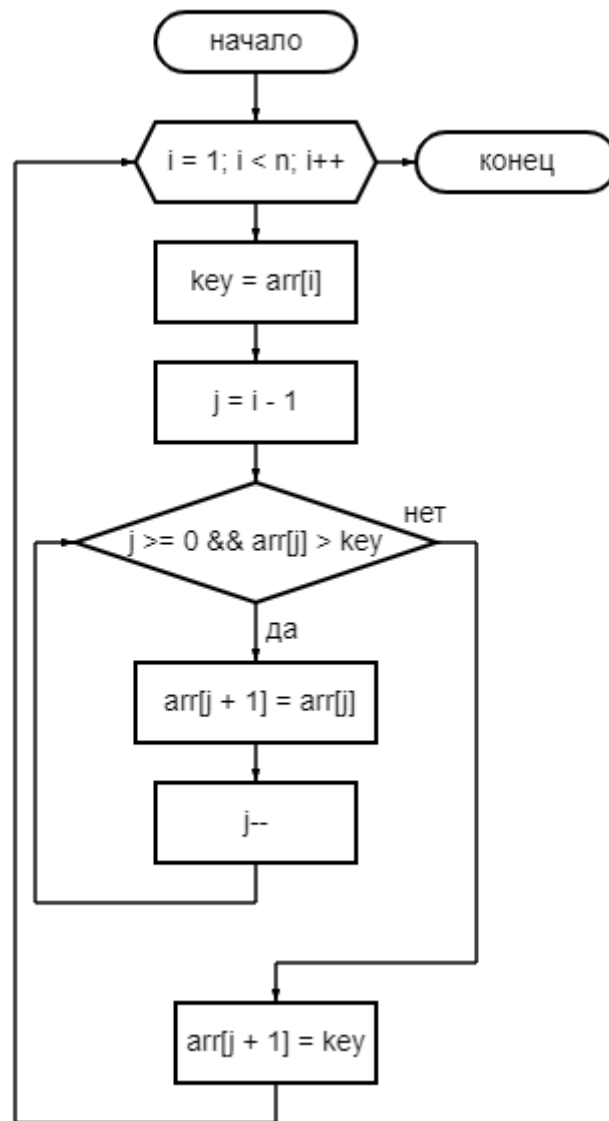


Рисунок 1 – Блок-схема алгоритма insertion sort

2.1.3 Функция роста

Таблица 2 – Вывод функции роста

Номер опе-ратора	Оператор	Кол-во выполнений оператора в строке (худший случай)	Кол-во выполнений оператора в строке (лучший случай)
1	<code>for (int i = 1; i < n; i++) {</code>	n	n
2	<code>int key = arr[i];</code>	n-1	n-1
3	<code>int j = i - 1;</code>	n-1	n-1
4	<code>while (j >= 0 && arr[j] > key){</code>	$1+2+\dots+(n-1) = n*(n-1)/2$	1
5	<code>arr[j + 1] = arr[j];</code>	$n*(n-1)/2-1$	0
6	<code>j--;</code>	$n*(n-1)/2-1$	0
	<code>}</code>		
7	<code>arr[j + 1] = key;</code>	n-1	n-1
	<code>}</code>		

Лучший случай для сортировки методом Простой вставки возникает, когда массив уже отсортирован. В этом случае внутренний цикл не выполняется, а функция роста $T(n) = 4n - 2$

Худший случай возникает, когда массив отсортирован в обратном порядке. В этом случае внутренний цикл будет выполнен максимальное количество раз для каждого элемента, а функция роста $T(n) = n^2 + 3n - 5 + \frac{n^2 - n}{2}$

Средний случай возникает, когда элементы массива расположены в случайном порядке. В этом случае среднее количество операций сравнения и перестановок будет пропорционально n^2

2.1.4 Код программы

```
#include <iostream>
#include <cstdlib>
#include <ctime>
#include <chrono>

using namespace std;

// Функция сортировки методом Простой вставки
void insertionSort(int arr[], int n) {
    chrono::steady_clock::time_point begin = chrono::steady_clock::now();
    int cnt = 0;
    for (int i = 1; i < n; i++) { // Цикл по всем элементам массива, начиная с
        //первого
        cnt++;
        int key = arr[i]; // Сохраняем текущий элемент в переменной key
        cnt++; cnt++;
        int j = i - 1; // Начинаем сравнивать с предыдущим элементом

        // Пока предыдущий элемент больше текущего и мы не дошли до начала массива
        while (j >= 0 && arr[j] > key) {
            cnt++; cnt++;
            arr[j + 1] = arr[j]; // Сдвигаем элементы вправо
            j--; cnt++; // Переходим к следующему элементу слева
        }
        cnt++;
        cnt++;
        arr[j + 1] = key; // Вставляем текущий элемент на правильное место
    }
    cnt++;
    chrono::steady_clock::time_point end = chrono::steady_clock::now();
    cout << "\nTime difference = " <<
    chrono::duration_cast<chrono::microseconds>(end - begin).count() << "[ms]" << endl;
    cout << "Time difference = " << chrono::duration_cast<chrono::nanoseconds>(end
    - begin).count() << "[ns]\n" << endl;
}

int main() {
    srand(time(NULL)); // Инициализируем генератор случайных чисел
    int n = 10; // Размер массива
    int* arr = new int[n]; // Создаем массив нужного размера
    // Заполняем массив случайными числами
    for (int i = 0; i < n; i++) {
        arr[i] = rand() % 100; // Генерируем случайное число от 0 до 99
    }
    // Выводим исходный массив
    cout << "Original array: ";
    for (int i = 0; i < n; i++) {
        cout << arr[i] << " ";
    }
    cout << endl;
    insertionSort(arr, n); // Сортируем массив методом Простой вставки

    // Выводим отсортированный массив
    cout << "Sorted array: ";
    for (int i = 0; i < n; i++) {
        cout << arr[i] << " ";
    }
    cout << endl;

    delete[] arr; // Освобождаем память, выделенную под массив

    return 0;
}
```

2.1.5 Тестирование

Результат работы программы для массива размерности 10 (рис. 2).

```
Original array: 67 66 3 46 22 41 40 37 97 21
cnt:130
Time difference = 0[ms]
Time difference = 300[ns]

Sorted array: 3 21 22 37 40 41 46 66 67 97
```

Рисунок 2 – Тест программы на массиве $n = 10$

2.1.6 Эмпирическая оценка вычислительной сложности алгоритма

Таблица 2 – Сводная таблица результатов

n	T(n), мс	T_T	T_{Π}
100	0.007		7966
1000	0.567		745855
10000	43.63		73641304
100000	4442.693		7417950757
1000000	449465.956		742713072145



2.1.7 Определение ёмкостной сложности

Для алгоритма сортировки методом простой вставки ёмкостная сложность является константной.

2.1.8 Вывод

На основе скорости роста функции роста, можно сделать вывод, что алгоритм сортировки вставками имеет квадратичную сложность, так как время выполнения растет пропорционально квадрату размера входных данных (n^2). Это подтверждается графически, так как на построенном графике зависимости времени выполнения от размера входных данных видно, что кривая имеет квадратичную форму.

Также можно заметить, что время выполнения алгоритма увеличивается значительно при увеличении размера входных данных, что свидетельствует о том, что данный алгоритм не является оптимальным для обработки больших объемов данных.

Для оптимизации времени работы данного алгоритма можно использовать более эффективные алгоритмы сортировки, такие как быстрая сортировка (QuickSort) или сортировка слиянием (MergeSort).

2.2 Задание 2

2.2.2 Худший случай

Таблица 3 – Сводная таблица результатов для худшего случая

n	T(n), мс	T_T	T_{Π}
100	0.01	15245	15346
1000	1.146	1502495	1503496
10000	90.134	150024995	150034996
100000	9132.065	15000249995	15000349996
1000000	920697.214	1500002499995	1500003499996

2.2.3 Лучший случай

Таблица 4 – Сводная таблица результатов для лучшего случая

n	T(n), мс	T_T	T_{Π}
100	0.001	398	496
1000	0.003	3998	4996
10000	0.026	39998	49996
100000	0.33	399998	499996
1000000	2.831	3999998	4999996

2.2.4 Вывод

Из результатов прогона программы можно сделать вывод, что алгоритм сортировки зависит от исходной упорядоченности массива. В наихудшем случае, когда массив отсортирован в обратном порядке, алгоритм сортировки производит наибольшее количество операций и работает медленнее. В наилучшем случае, когда массив уже отсортирован, алгоритм работает быстрее. Однако, даже в наилучшем случае время выполнения алгоритма все еще зависит от размера входных данных, и может быть существенным при больших значениях n . Поэтому для оптимизации времени выполнения алгоритма необходимо использовать более эффективные алгоритмы сортировки.

2.3 Задание 3

2.3.1 Алгоритм selectionSort

Сортировка выбором (Selection sort) - это алгоритм сортировки, который на каждой итерации выбирает наименьший (или наибольший) элемент в массиве и помещает его в начало (или конец) отсортированной части массива.

Алгоритм состоит из двух основных циклов. Внешний цикл проходит от начала массива до конца и выбирает элемент, который будет находиться на нужной позиции. Внутренний цикл проходит от текущей позиции до конца массива и ищет минимальный (или максимальный) элемент, который затем меняется местами с текущим элементом.

Этот процесс повторяется до тех пор, пока не будет отсортирован весь массив. На каждой итерации внешнего цикла гарантированно ставится на своё место хотя бы один элемент, что гарантирует правильность алгоритма.

Алгоритм сортировки выбором (Selection Sort) состоит из следующих шагов:

1. Находим минимальный элемент в неотсортированной части массива.
2. Меняем местами найденный минимальный элемент с первым элементом в неотсортированной части.
3. Повторяем шаги 1-2 для оставшейся неотсортированной части массива до тех пор, пока не отсортируем весь массив.

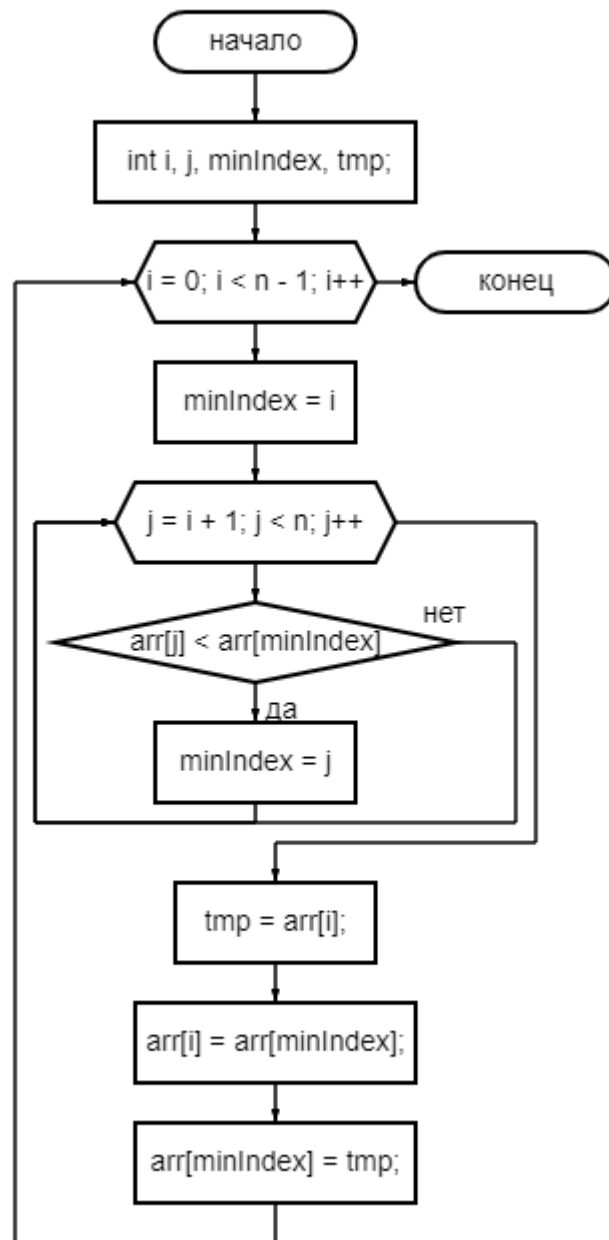


Рисунок 3 – Блок схема алгоритма selectionSort

2.3.2 Функция роста

Таблица 5 – Вывод функции роста

Номер оператора	Оператор	Кол-во выполнений оператора в строке (худший случай)	Кол-во выполнений оператора в строке (лучший случай)
1	for (i = 0; i < n - 1; i++) {	n	n
2	minIndex = i;	n-1	n-1
3	for (j = i + 1; j < n; j++) {	1+2+...+(n-1) = n*(n-1)/2	n*(n-1)/2
4	if (arr[j] < arr[minIndex]) {	n*(n-1)/2-1	n*(n-1)/2-1
5	minIndex = j;	n*(n-1)/2-2	0
	}		
	}		
6	tmp = arr[i];	n-1	n-1
7	arr[i] = arr[minIndex];	n-1	n-1
8	arr[minIndex] = tmp;	n-1	n-1
	}		

Худший случай наступает, когда массив уже отсортирован в обратном порядке. В этом случае для каждой итерации внешнего цикла будет найден минимальный элемент в оставшейся части массива и перемещен в начало.

Получаем функцию роста для худшего случая: $\frac{7n+3n^2}{2} - 7$

Лучший случай возникает, когда массив уже отсортирован в возрастающем порядке. В этом случае при каждой итерации внешнего цикла будет найден минимальный элемент в оставшейся части массива и перемещен в начало, но на самом деле перемещение не будет произведено, так как минимальный элемент уже находится в начале массива.

Получаем функцию роста для лучшего случая: $n^2 + 4n - 5$

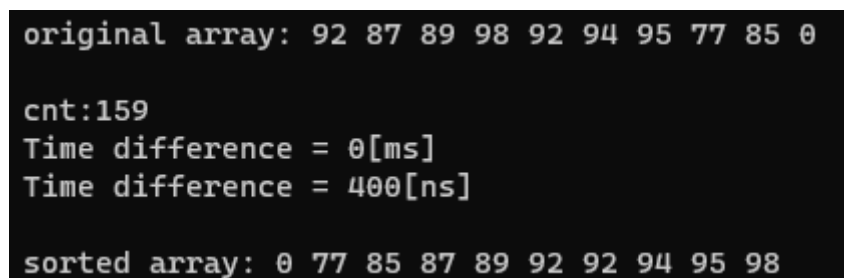
Средний случай трудно оценить, так как он зависит от распределения элементов в массиве. Однако, можно считать, что в среднем количество операций перемещения элементов будет примерно равно количеству операций сравнения элементов, то есть примерно $(n^2 - n)/2$ операций.

2.3.3 Код алгоритма

```
// Функция сортировки методом Простого выбора
void selectionSort(int arr[], int n) {
    chrono::steady_clock::time_point begin = chrono::steady_clock::now();
    unsigned long long cnt = 0;
    int i, j, minIndex, tmp;
    for (i = 0; i < n - 1; i++) {
        cnt++;
        minIndex = i; // минимальный элемент находится на i-ой позиции
        cnt++;
        for (j = i + 1; j < n; j++) {
            cnt++; cnt++;
            if (arr[j] < arr[minIndex])
            {
                minIndex = j; cnt++; // найден новый минимальный элемент
            }
        }
        cnt++;
        // меняем местами i-ый и минимальный элементы
        tmp = arr[i]; cnt++;
        arr[i] = arr[minIndex]; cnt++;
        arr[minIndex] = tmp; cnt++;
    }
    cnt++;
    chrono::steady_clock::time_point end = chrono::steady_clock::now();
    cout << "\ncnt:" << cnt << endl;
    cout << "Time difference = " << chrono::duration_cast<chrono::microseconds>(end
- begin).count() << "[ms]" << endl;
    cout << "Time difference = " << chrono::duration_cast<chrono::nanoseconds>(end
- begin).count() << "[ns]\n" << endl;
}
```

2.3.4 Тестирование

Результат работы алгоритма для массива размерности 10 (рис. 2).



```
original array: 92 87 89 98 92 94 95 77 85 0

cnt:159
Time difference = 0[ms]
Time difference = 400[ns]

sorted array: 0 77 85 87 89 92 92 94 95 98
```

Рисунок 4 – Тест алгоритма на массиве $n = 10$

2.3.5 Эмпирическая оценка вычислительной сложности алгоритма

Таблица 6 – Сводная таблица результатов для среднего случая

n	T(n), мс	T_T	T_{Π}
100	0.012		10782
1000	0.808		1008796
10000	66.487		100092324
100000	6751.805		10000917968
1000000	661897.008		1000009189988

Таблица 7– Сводная таблица результатов для худшего случая

n	T(n), мс	T_T	T_{Π}
100	0.011	15343	12995
1000	0.847	1503493	1254995
10000	74.747	150034993	125049995
100000	7552.356	15000349993	12500499995
1000000	749747.142	1500003499993	1250004999995

Таблица 8 – Сводная таблица результатов для лучшего случая

n	T(n), мс	T_T	T_{Π}
100	0.009	10395	10495
1000	0.652	1003995	1004995
10000	69.242	100039995	100049995
100000	6670.901	10000399995	10000499995
1000000	663638.306	1000003999995	1000004999995

2.3.6 Определение ёмкостной сложности

Ёмкостная сложность алгоритма определяет, сколько памяти требуется для его выполнения и как эта требуемая память зависит от размера входных данных.

Таким образом, ёмкостная сложность алгоритма сортировки выбором в наихудшем, наилучшем и среднем случаях является константной.

2.3.7 График

График худшего случая двух алгоритмов

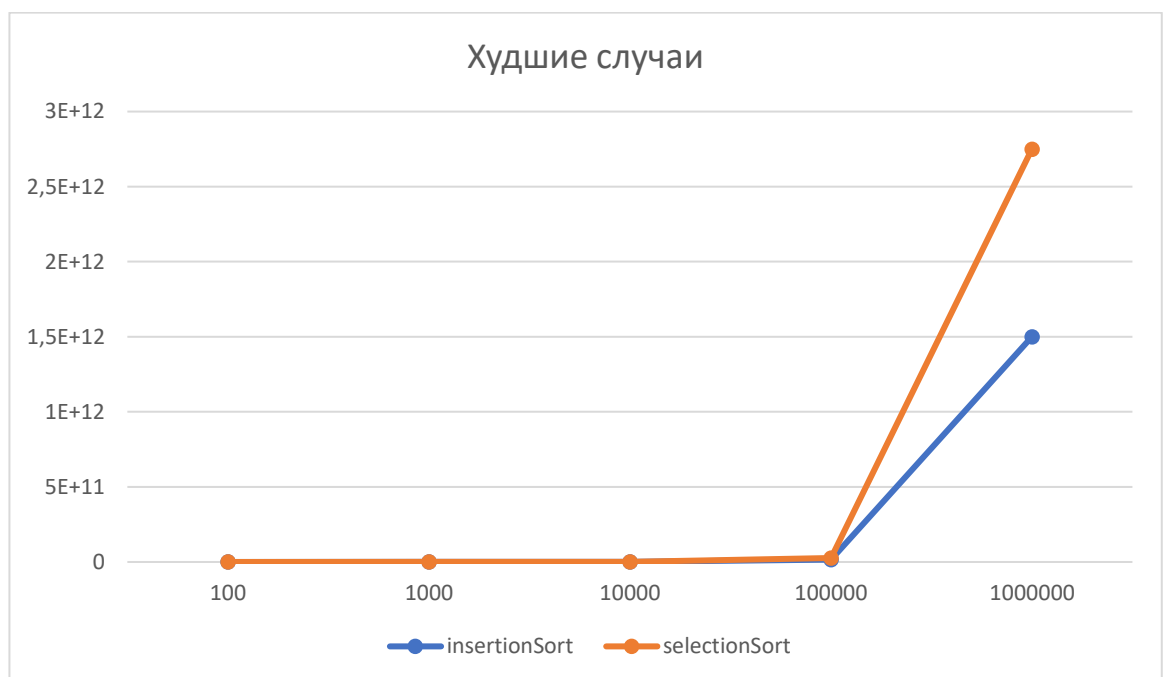
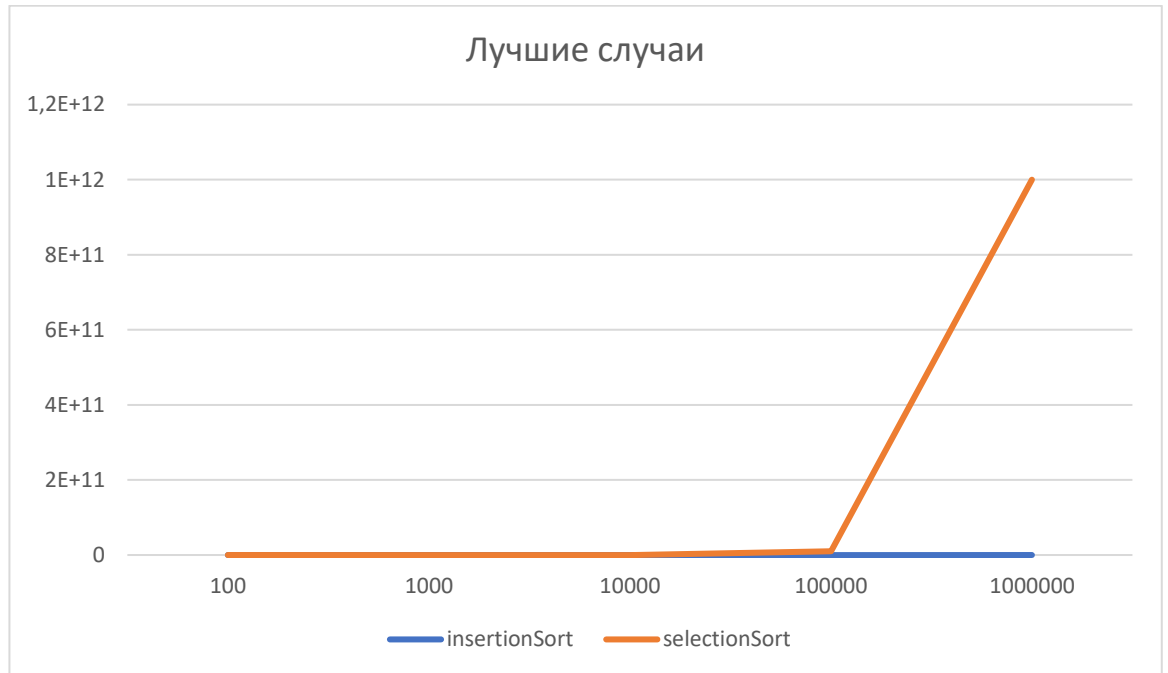


График лучшего случая двух алгоритмов



2.3.8 Вывод

Из эмпирического анализа результатов можно сделать вывод, что `insertionSort` является более эффективным алгоритмом сортировки

Алгоритм сортировки вставками (`insertion sort`) в общем случае эффективнее, чем алгоритм сортировки выбором (`selection sort`). Это происходит потому, что вставка элемента в отсортированную часть массива происходит за время $T(n)=(1)$, тогда как при поиске минимального элемента в неотсортированной части массива требуется проход по всей этой части, что в среднем занимает $T(n)=(n/2)$ операций.

Кроме того, алгоритм сортировки вставками лучше справляется с частично отсортированными массивами, так как в таком случае отсортированная часть массива будет нарастать быстрее, что уменьшает количество операций вставки новых элементов.

3 ВЫВОД

В рамках данной работы были проведены эксперименты для определения вычислительной сложности алгоритмов сортировки (insertionSort и selectionSort) на различных массивах, включая отсортированные и неотсортированные в порядке возрастания и убывания. Были получены эмпирические данные по времени выполнения алгоритмов на каждом из массивов при различных размерах данных.

Анализ полученных результатов показал, что алгоритм insertionSort имеет лучшую вычислительную сложность, чем алгоритм selectionSort, что подтверждается как теоретически, так и практически.

Также было выявлено, что сложность алгоритмов сильно зависит от упорядоченности входных данных, и наихудший случай возникает при сортировке отсортированных массивов в порядке убывания.

Таким образом, выполнение данной работы позволило закрепить знания по эмпирическому определению вычислительной сложности алгоритмов, а также проиллюстрировало важность выбора эффективного алгоритма с учетом упорядоченности входных данных.

4 СПИСОК ИНФОРМАЦИОННЫХ ИСТОЧНИКОВ

1. Рысин М.Л. и др. Введение в структуры и алгоритмы обработки данных. Ч. 1 - учебное пособие, 2022.pdf
2. ПР-1.1 (Теоретическая сложность алгоритма).
3. Лекционные материалы - СиАОД-1Файл
4. <https://metanit.com/>
5. <https://ru.cppreference.com/w/>
6. Оценка сложности алгоритмов <https://habr.co/>
7. Бхаргава А. Грожаем алгоритмы. Иллюстрированное пособие для программистов и любопытствующих. – СПб: Питер, 2017. – 288 с.
8. Вирт Н. Алгоритмы + структуры данных = программы. – М.: Мир, 1985. – 406 с.
9. Кнут Д.Э. Искусство программирования, том 3. Сортировка и поиск, 2-е изд. – М.: ООО «И.Д. Вильямс», 2018. – 832 с.
10. Седжвик Р. Фундаментальные алгоритмы на C++. Анализ/Структуры данных/Сортировка/Поиск. – К.: Издательство «Диасофт», 2001. – 688 с.
11. AlgoList – алгоритмы, методы, исходники [Электронный ресурс]. URL:<http://algotlist.manual.ru/> (дата обращения 15.03.2022).
12. Алгоритмы – всё об алгоритмах / Хабр [Электронный ресурс]. URL: <https://habr.com/ru/hub/algorithms/> (дата обращения 15.03.2022).
13. НОУ ИНТУИТ | Технопарк Mail.ru Group: Алгоритмы и структуры данных[Электронныйресурс].URL:<https://intuit.ru/studies/courses/3496/738/info> (дата обращения 15.03.2022).