Отчет о выполнении лабораторной работы по информатике

Алгоритмы Сортировки

Студент: Копытова Виктория Сергеевна
Группа: Б03-304

1 Аннотация

Цель работы: научиться составлять алгоритмы и анализировать их время работы; научиться использовать различные степени оптимизации.

2 Ход работы

2.1 Сортировки с временной сложностью $O(N^2)$

Рассмотрит алгоритмы сортировки пузырьком (Bubble Sort), вставкой (Insertion Sort) и выбором (Selection Sort) и убедимся в том, что временная сложность – $O(N^2)$ (все расчёты проводятся на процессоре Intel Core i7 12700H 2,3 ГГц; ноутбук подключён к питанию).

Функция для измерения времени сортировки пузырьком, где rund_uns(min, max) возвращает случайные числа от min до max:

```
double BubbleSort(long long int n)
{
    int arr[100000]={0};
    for (long long int i=0;i<n;i++)
    {
        arr[i]=rand_uns(0,100000);
    }
    auto start = chrono::steady_clock::now();
    for (long long int i=0;i<n;i++)
    {
        for (long long int j=0;j<n-1;j++)
        {
            if (arr[i]<arr[j])
            {
                swap(arr[i],arr[j]);
            }
        }
    }
    auto end = chrono::steady_clock::now();
    auto elapsed = chrono::duration_cast<chrono::microseconds>(end - start);
    double time = (double)(elapsed.count());
    return time;
}
```

Функция для измерения времени сортировки выбором:

```
double SelectionSort(long long int n)
{
    int arr[100000]={0};
```

```
    for (long long int i=0;i<n;i++)
    {
        arr[i]=rand_uns(0,100000);
    }
    long long int m = 0;
    auto start = chrono::steady_clock::now();
    for (long long int i=0;i<n-1;i++)
    {
        for (long long int j=i+1;j<n;j++)
        {
            if (arr[i]<arr[m])
            {
                m=j;
            }
        }
        if (i != m)
        {
            swap(arr[i],arr[m]);
        }
    }
    auto end = chrono::steady_clock::now();
    auto elapsed = chrono::duration_cast<chrono::microseconds>(end - start);
    double time = (double)(elapsed.count());
    return time;
}
```

Функция для измерения времени сортировки выбором:

```
double InsertSort(long long int n)
{
    int arr[100000]={0};
    for (long long int i=0;i<n;i++)
    {
        arr[i]=rand_uns(0,100000);
    }
    auto start = chrono::steady_clock::now();
    for (long long int i=1;i<n;i++)
    {
        for (long long int j=i;j>0;j--)
        {
            if (arr[j-1]>arr[j])
            {
                swap(arr[j-1],arr[j]);
            }
        }
```

```
    }
    auto end = chrono::steady_clock::now();
    auto elapsed = chrono::duration_cast<chrono::microseconds>(end - start);
    double time = (double)(elapsed.count());
    return time;
}
```

Функции для расчёта среднего времени выполнения сортировки для 100 массивов случайных чисел (на примере сортировки пузырьком):

```
void BubbleSortMeanTime(long long int n)
{
    double meanTime = 0;
    cout<<"Bubble Sort"<<endl;
    for (long long int i = 1;i <= n;i *= 2)
    {
        for (int j=0;j<100;j++)
        {
            meanTime += BubbleSort(i);
        }
        cout<<(meanTime/100.0)<<','<<' ';
    }
    cout<<'\n';
}
```

Построим график зависимости времени выполнения сортировки от размера массива в координатах $(\log_2 t)(\log_2 N)$ (на всех графиках время измеряется в микросекундах).

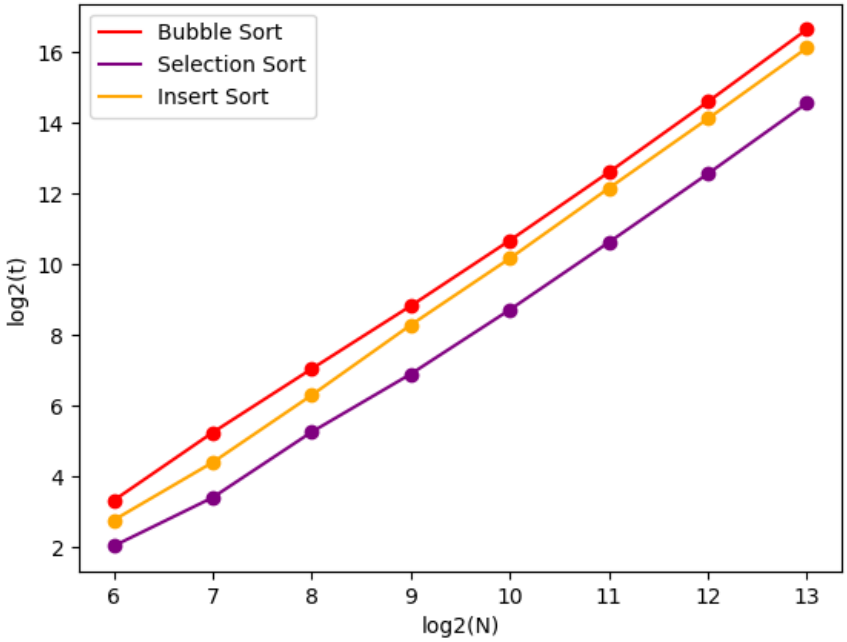

Рис. 1: Время выполнения сортировок $O(N^2)$

Мы видим, что зависимость линейная. Значит:

$$\log_2 t = \log_2 C + 2 \cdot \log_2 N$$

Наклон всех графиков одинаковый, но постоянная C отличается, так как она зависит от типов используемых переменных и выделяемой на них памяти.

3 Оптимизация сортировки пузырьком

В настройках компилятора будем выбирать различные уровни оптимизации для алгоритма сортировки из предыдущего пункта и строить график зависимотсти времени выполнения программы от размера массива.

Полученный результат:

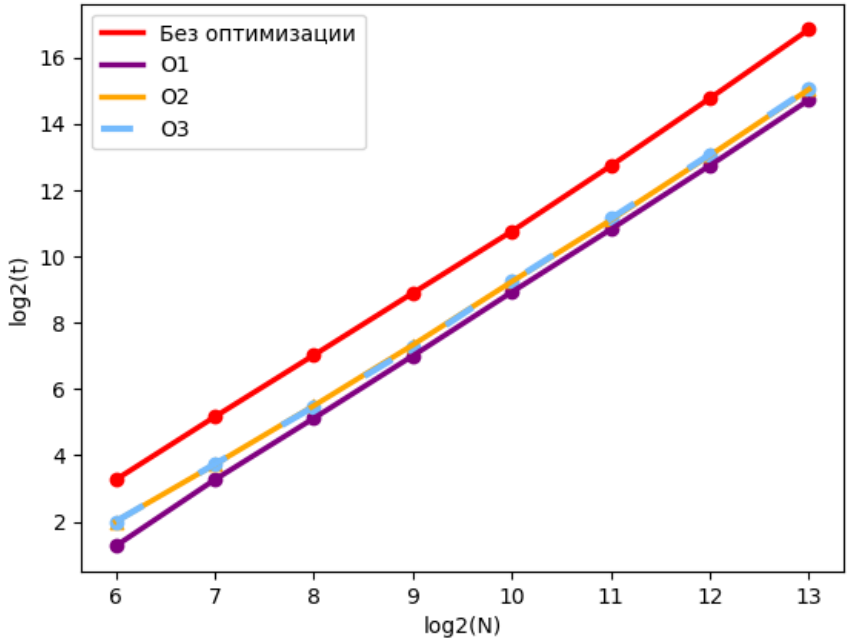

Рис. 2: Сортировка пузырьком с различными уровнями оптимизации

Оптимизация достаточно сильно уменьшает время выполнения сортировки, но не меняет сложность алгоритма. Так как программа не использует большого объёма памяти, оптимизации O2 и O3 почти не отличаются от O1.

4 Сортировки сложности $O(N \log N)$

Рассмотрим пирамидальную сортировку (Heap Sort), сортировку расчёской (Comb Sort), сортировку Хоара (Quick Sort). (при всех последующих измерениях оптимизация отключена)

Функция для измерения времени пирамидальной сортировки:

```
void heapify(int *arr, int n, int root)
{
    int largest = root;
    int left = 2*root + 1;
    int right = 2*root + 2;
    if (left < n && arr[left] > arr[largest])
        largest = left;
    if (right < n && arr[right] > arr[largest])
        largest = right;
    if (largest != root)
    {
```

```
        swap(arr[root], arr[largest]);

        heapify(arr, n, largest);
    }
}
double heapSort(int *arr, int n)
{
    auto start = chrono::steady_clock::now();
    for (int i = n / 2 - 1; i >= 0; i--)
        heapify(arr, n, i);
    for (int i=n-1; i>=0; i--)
    {
        swap(arr[0], arr[i]);
        heapify(arr, i, 0);
    }
    auto end = chrono::steady_clock::now();
    auto elapsed = chrono::duration_cast<chrono::microseconds>(end - start);
    double time = (double)(elapsed.count());
    return time;
}
```

Функция heapify() создаёт пирамиду вида:

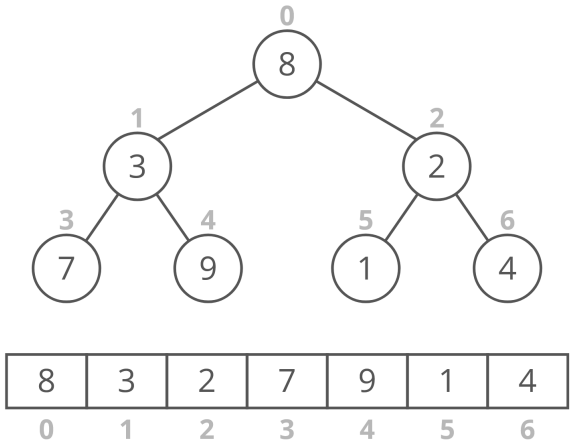

Рис. 3: Функция heapify()

Функция heapsort() перемещает корент массива в конец и вызывает heapify() от массива длиной N-1. Зависимость времени от количества элементов массива:

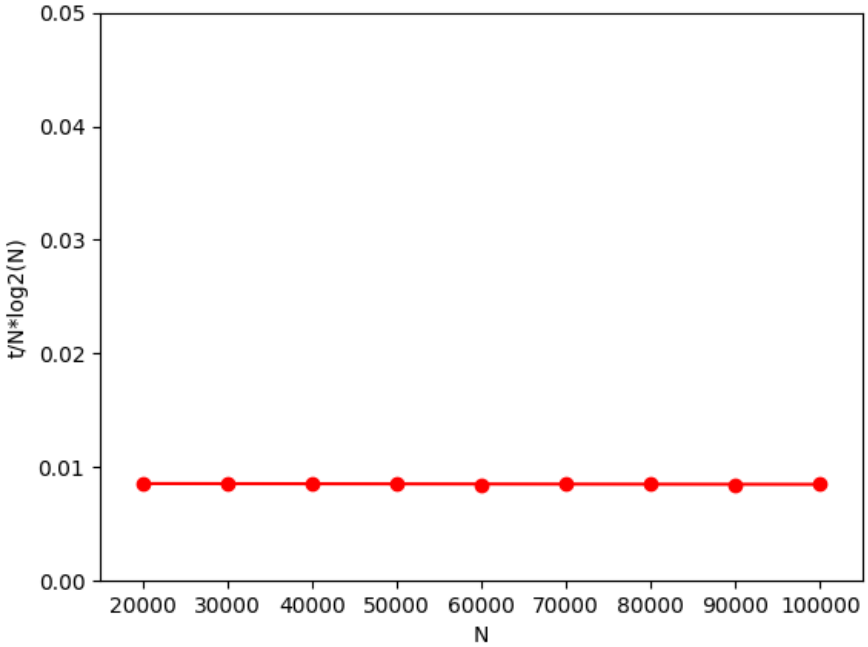

Рис. 4: Пирамидальная сортировка

Функция для измерения времени сортировки Хоара:

```
double quickSort(int *arr, int left, int right)
{
    auto start = chrono::steady_clock::now();
    if (left<right)
    {
        int i=left;
        int j=right;
        int x = arr[(left + right) / 2];
        while(i<=j)
        {
            while(arr[i] < x)
            {
                i+=1;
            }
            while(arr[j] > x)
            {
                j-=1;
            }
            if (i<=j)
            {
                swap(arr[i],arr[j]);
                i+=1;
```

```
                j-=1;
            }
        }
        quickSort(arr,left,j);
        quickSort(arr,i,right);
    }
    auto end = chrono::steady_clock::now();
    auto elapsed = chrono::duration_cast<chrono::microseconds>(end - start);
    double time = (double)(elapsed.count());
    return time;
}
```

Алгоритм основан на том, что опорный элемент сравнивается с остальными и выполняется перестановка вида:

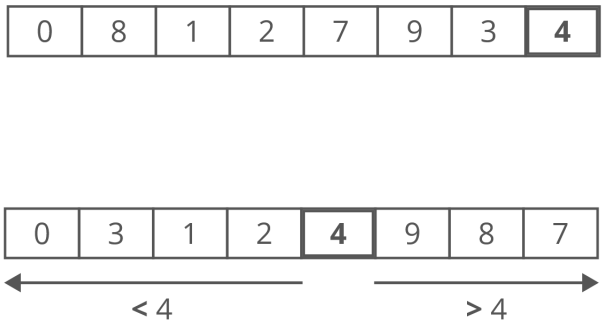

Рис. 5: Принцип сортировки Хоара

Далее процедура рекурсивно выполняется для левой и правой частей массива.

Зависимость времени от количества элементов массива:

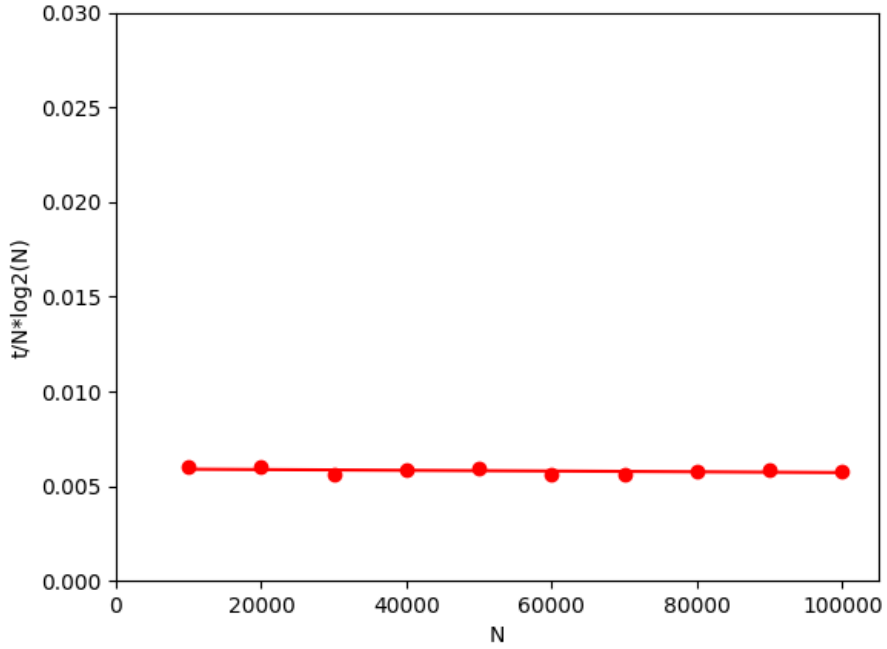

Рис. 6: Сортировка Хоара

Функция для измерения времени сортировки расчёской:

```
int getNextGap(int gap)
{
        gap = (gap*10)/13;

        if (gap < 1)
                return 1;
        return gap;
}
double combSort(int *a, int n)
{
        int gap = n;
        bool swapped = true;
        auto start = chrono::steady_clock::now();
        while (gap != 1 || swapped == true)
        {
                gap = getNextGap(gap);
                swapped = false;
                for (int i=0; i<n-gap; i++)
                {
                        if (a[i] > a[i+gap])
                        {
                                swap(a[i], a[i+gap]);
```

```
                                swapped = true;
                            }
                        }
                    }
                auto end = chrono::steady_clock::now();
            auto elapsed = chrono::duration_cast<chrono::microseconds>(end - start);
            double time = (double)(elapsed.count());
            return time;
}
```

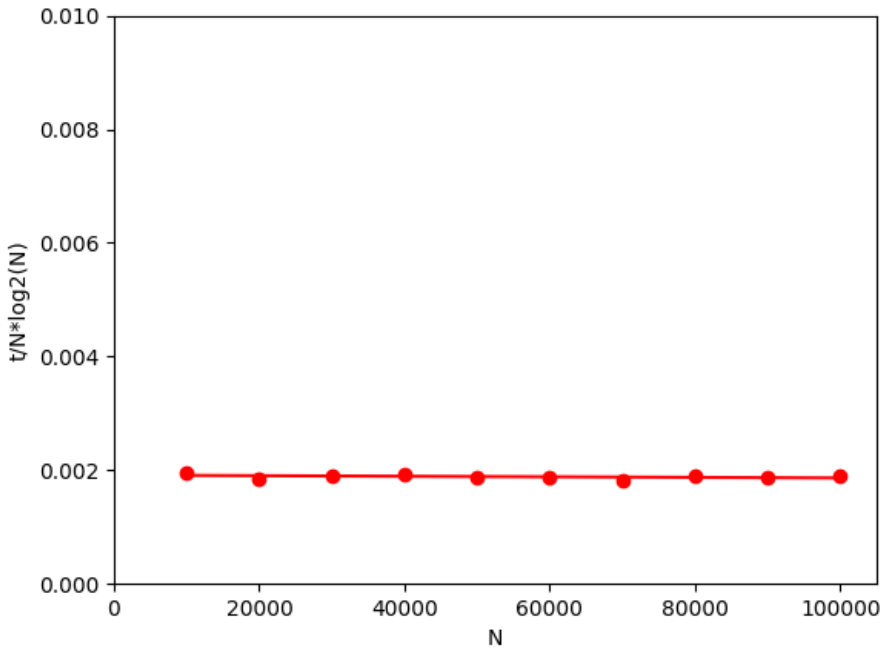

Рис. 7: Сортировка расчёской

Графики показывают, что сложность рассмотренных алгоритмов действительно $O(N \log N)$.

5 Сравнение алгоритмов $O(N^2)$ и $O(N \log N)$

Сравним сортировку пузырьком (сложность $O(N^2)$) и пирамидальную сортировку (сложность $O(N \log N)$).

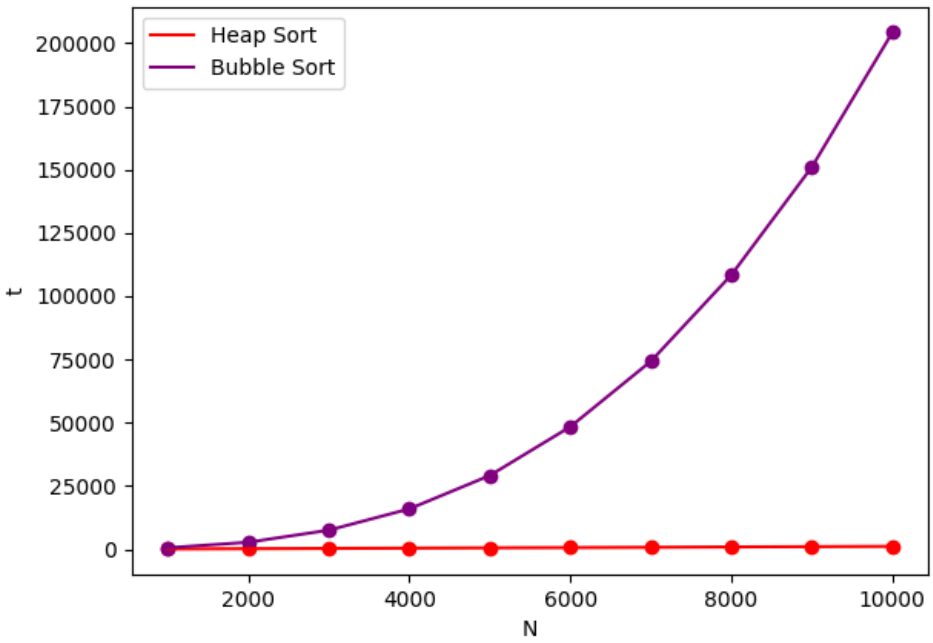

Рис. 8: Сравнение алгоритмов

Сортировка больших массивов пузырьком занимает значительно больше времени, чем сортировка Хоара. В данном масштабе график $N \log N$ напоминает прямую.

6 Зависимость времени выполнения сортировок от начальных данных

Рассмотрим время сортировки отсортированного массива, массива случайных чисел и массива, отсортированного в обратную сторону. Сортировки сложности $O(N^2)$ (на примере сортировки пузырьком):

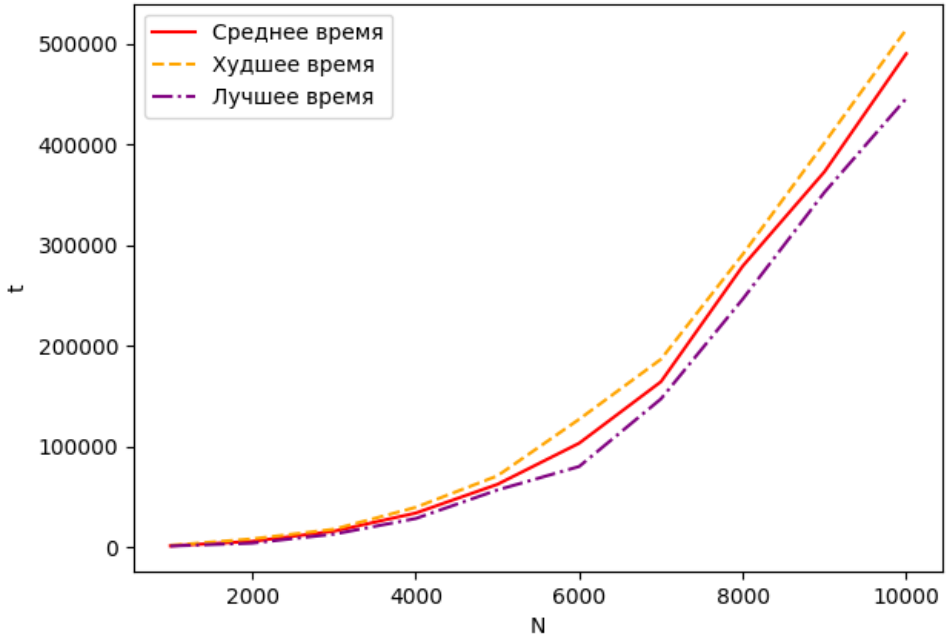

Рис. 9: Сортировки сложности $O(N^2)$

Исходные данные не меняют временной сложности алгоритма.

Сортировки сложности $O(N \log N)$ (на примере пирамидальной сортировки):

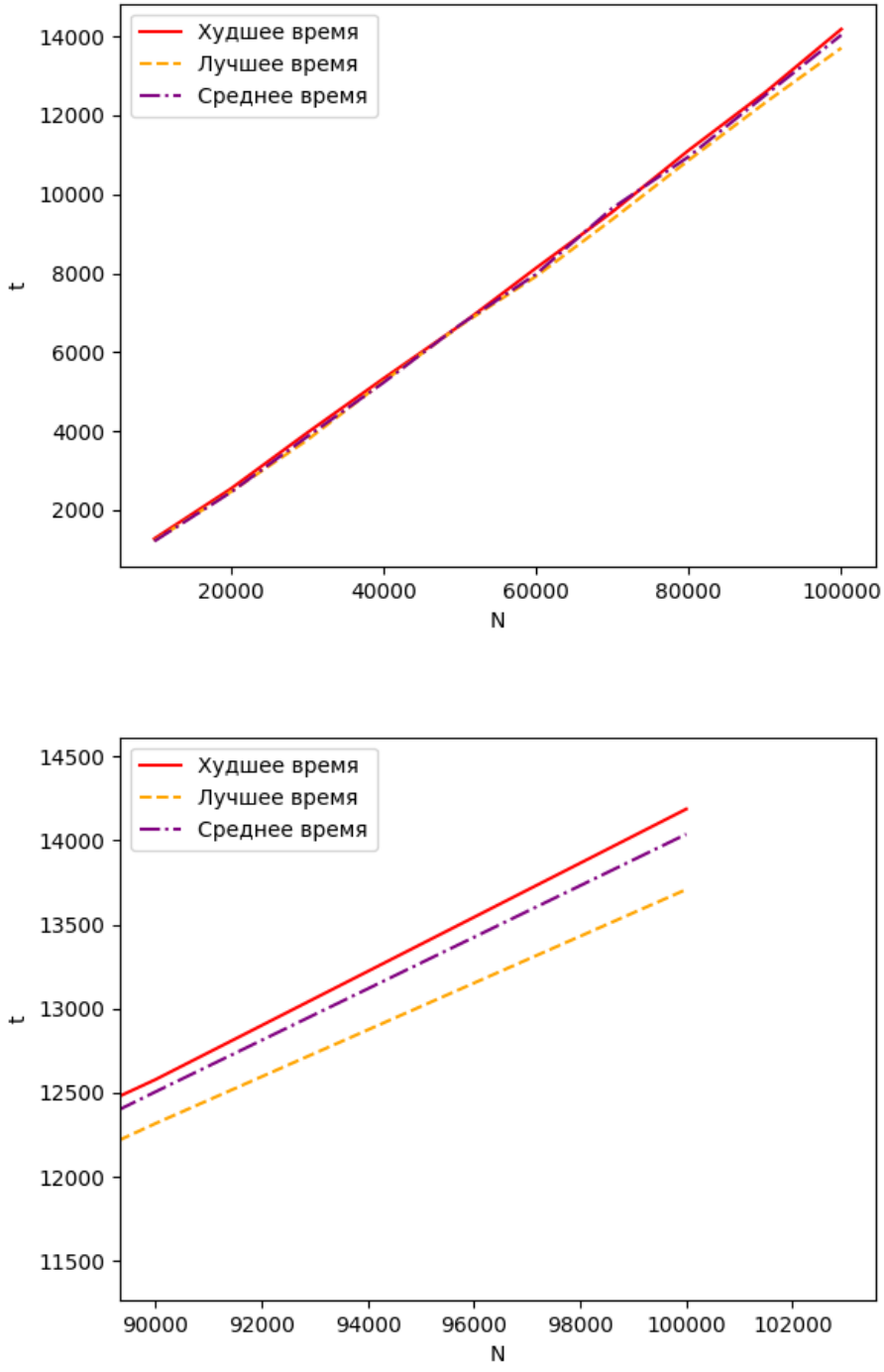

Рис. 10: Сортировки сложности $O(N \log N)$

Для отсортированного массива временная сложность – $O(N)$.

7 Вывод

В работе были рассмотрены основные алгоритмы сортировки с различной временной сложностью. Сравнение этих алгоритмов между собой показывает, что путём создания более продвинутых алгоритмов можно значительно уменьшить время выполнения программы на том же самом компьютере. Также для ускорения выполнения программы применяются различные методы оптимизации.