



Министерство науки и высшего образования Российской Федерации  
Федеральное государственное бюджетное образовательное учреждение  
высшего образования  
«Московский государственный технический университет имени  
Н.Э. Баумана  
(национальный исследовательский университет)»  
(МГТУ им. Н.Э. Баумана)

---

ФАКУЛЬТЕТ «Информатика и системы управления»

---

КАФЕДРА «Программное обеспечение ЭВМ и информационные технологии»

---

## Отчет по лабораторной работе №3 по курсу "Анализ алгоритмов"

Тема Трудоёмкость сортировок

Студент Романов С.К.

Группа ИУ7-55Б

Оценка (баллы) \_\_\_\_\_

Преподаватели Волкова Л.Л., Строганов Ю.В.

# Оглавление

<b>ВВЕДЕНИЕ</b>	<b>3</b>
<b>1 Аналитическая часть</b>	<b>4</b>
1.1 Пузырьковая сортировка . . . . .	4
1.1.1 Описание алгоритма . . . . .	4
1.1.2 Псевдокод . . . . .	4
1.1.3 Анализ алгоритма . . . . .	4
1.2 Сортировка подсчетом . . . . .	5
1.2.1 Описание алгоритма . . . . .	5
1.2.2 Псевдокод . . . . .	5
1.2.3 Анализ алгоритма . . . . .	6
1.3 Быстрая сортировка . . . . .	6
1.3.1 Описание алгоритма . . . . .	6
1.3.2 Псевдокод . . . . .	7
1.3.3 Анализ алгоритма . . . . .	7
<b>2 Конструкторская часть</b>	<b>8</b>
2.1 Разработка алгоритмов . . . . .	8
2.2 Трудоёмкость алгоритмов . . . . .	12
2.3 Модель вычислений . . . . .	12
2.4 Трудоёмкость алгоритмов . . . . .	12
2.4.1 Алгоритм сортировки пузырьком . . . . .	12
2.4.2 Алгоритм сортировки подсчетом . . . . .	13
2.4.3 Алгоритм итеративной быстрой сортировки . . . . .	14
<b>3 Технологическая часть</b>	<b>18</b>
3.1 Требования к программному обеспечению . . . . .	18
3.2 Средства реализации . . . . .	18
3.3 Листинги кода . . . . .	18

<b>4</b>	<b>Исследовательская часть</b>	<b>21</b>
4.1	Время выполнения алгоритмов . . . . .	21
	<b>ЗАКЛЮЧЕНИЕ</b>	<b>24</b>
	<b>СПИСОК ИСПОЛЬЗОВАННЫХ ИСТОЧНИКОВ</b>	<b>25</b>

# ВВЕДЕНИЕ

Цель лабораторной работы: изучить, реализовать и протестировать алгоритмы сортировки, оценить их трудоемкость.

## **Задачи лабораторной работы:**

1. Изучить алгоритмы сортировки.
  - Пузырьковая сортировка.
  - Сортировка подсчетом.
  - Быстрая сортировка.
2. Оценить трудоемкость алгоритмов и сравнивать их временные характеристики экспериментально.
3. Сделать выводы на основе полученных результатов.

## **Для достижения поставленных целей и задач необходимо:**

1. Изучить теоретические основы алгоритмов сортировки.
2. Реализовать алгоритмы сортировки.
3. Протестировать корректность алгоритмов сортировки.
4. Провести экспериментальное исследование.

## **В ходе работы будут затронуты следующие темы:**

1. Алгоритмы сортировки.
2. Сложность алгоритмов.
3. Тестирование алгоритмов

# 1 Аналитическая часть

## 1.1 Пузырьковая сортировка

### 1.1.1 Описание алгоритма

Алгоритм состоит из повторяющихся проходов по сортируемому массиву. За каждый проход элементы последовательно сравниваются попарно и, если порядок в паре неверный, выполняется обмен элементов. Проходы по массиву повторяются  $N - 1$  раз, но есть модифицированная версия, где если окажется, что обмены больше не нужны, значит проходы прекращаются. При каждом проходе алгоритма по внутреннему циклу очередной наибольший элемент массива ставится на свое место в конце массива рядом с предыдущим “наибольшим элементом”, а наименьший элемент массива перемещается на одну позицию к началу массива (“всплывает” до нужной позиции, как пузырёк в воде – отсюда и название алгоритма).

### 1.1.2 Псевдокод

```
procedure BUBBLESORT( $A$ )  
   $n \leftarrow \text{len}(A)$   
  for  $i \leftarrow 1$  to  $n$  do  
    for  $j \leftarrow 1$  to  $n - i$  do  
      if  $A[j] > A[j + 1]$  then  
         $SWAP(A[j], A[j + 1])$   
      end if  
    end for  
  end for  
end procedure
```

### 1.1.3 Анализ алгоритма

- Время выполнения алгоритма:  $O(n^2)$
- Память:  $O(1)$

## 1.2 Сортировка подсчетом

### 1.2.1 Описание алгоритма

Сортировка подсчетом — простейший способ упорядочить массив за линейное время. Применять его можно только для целых чисел, небольшого диапазона, т.к. он требует  $O(M)$  дополнительной памяти, где  $M$  — ширина диапазона сортируемых чисел. Алгоритм особо эффективен когда мы сортируем большое количество чисел, значения которых имеют небольшой разброс — например: массив из 1000000 целых чисел, которые принимают значения от 0 до 1000.

### 1.2.2 Псевдокод

```
procedure COUNTING SORT( $A$ )  
   $n \leftarrow \text{len}(A)$   
   $C \leftarrow \text{new array}(n)$   
  for  $i \leftarrow 1$  to  $n$  do  
     $C[i] \leftarrow 0$   
  end for  
  for  $i \leftarrow 1$  to  $n$  do  
     $C[A[i]] \leftarrow C[A[i]] + 1$   
  end for  
   $i \leftarrow 0$   
   $j \leftarrow 0$   
  while  $i \neq \text{len}(A) \ \&\& \ j \neq n$  do  
    if  $C[j] \neq 0$  then  
       $A[i] \leftarrow j$   
       $C[j] \leftarrow C[j] - 1$   
       $i \leftarrow i + 1$   
    else  
       $j \leftarrow j + 1$   
    end if  
  end while
```

end procedure

### 1.2.3 Анализ алгоритма

- Время выполнения алгоритма:  $O(n + k)$
- Память:  $O(n + k)$ 
  - $n$  - размер массива
  - $k$  - максимальное значение элемента массива
- Сортировка подсчетом работает только с целыми числами
- Сортировка подсчетом не является устойчивой
- Сортировка подсчетом не подходит для сортировки больших массивов

## 1.3 Быстрая сортировка

### 1.3.1 Описание алгоритма

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, следующих друг за другом: «элементы меньше опорного», «равные» и «большие».
- для отрезков «меньших» и «больших» значений выполнить рекурсивно ту же последовательность операций, если длина отрезка больше единицы.

На практике массив обычно делят не на три, а на две части: например, «меньшие опорного» и «равные и большие»; такой подход в общем случае эффективнее, так как упрощает алгоритм разделения.

### 1.3.2 Псевдокод

**procedure** QUICKSORT( $A$ )

$n \leftarrow \text{len}(A)$

$S \leftarrow \{\}$

▷ Стек

$S \leftarrow S \cup \{0, n - 1\}$

**while**  $S \neq \{\}$  **do**

$r \leftarrow S[n_S - 1]$

$l \leftarrow S[n_S - 2]$

$S \leftarrow S \setminus \{r, l\}$

$p \leftarrow \text{partition}(A, l, r)$

$S \leftarrow S \cup \{l, p - 1\}$

$S \leftarrow S \cup \{p + 1, r\}$

**end while**

**end procedure**

### 1.3.3 Анализ алгоритма

- Время выполнения алгоритма:  $O(n \log n)$
- Память:  $O(\log n)$

### Вывод

В данном разделе были описаны основные алгоритмы сортировки. Были рассмотрены следующие алгоритмы:

- Сортировка пузырьком
- Сортировка подсчетом
- Быстрая сортировка

Были рассмотрены их особенности, сложность, а также приведен псевдокод.



## 2 Конструкторская часть

### 2.1 Разработка алгоритмов

На рисунках 2.1, 2.2 и рисунках 2.3, 2.4 приведены схемы алгоритмов сортировки пузырьком, сортировки подсчетом и быстрой сортировки соответственно.

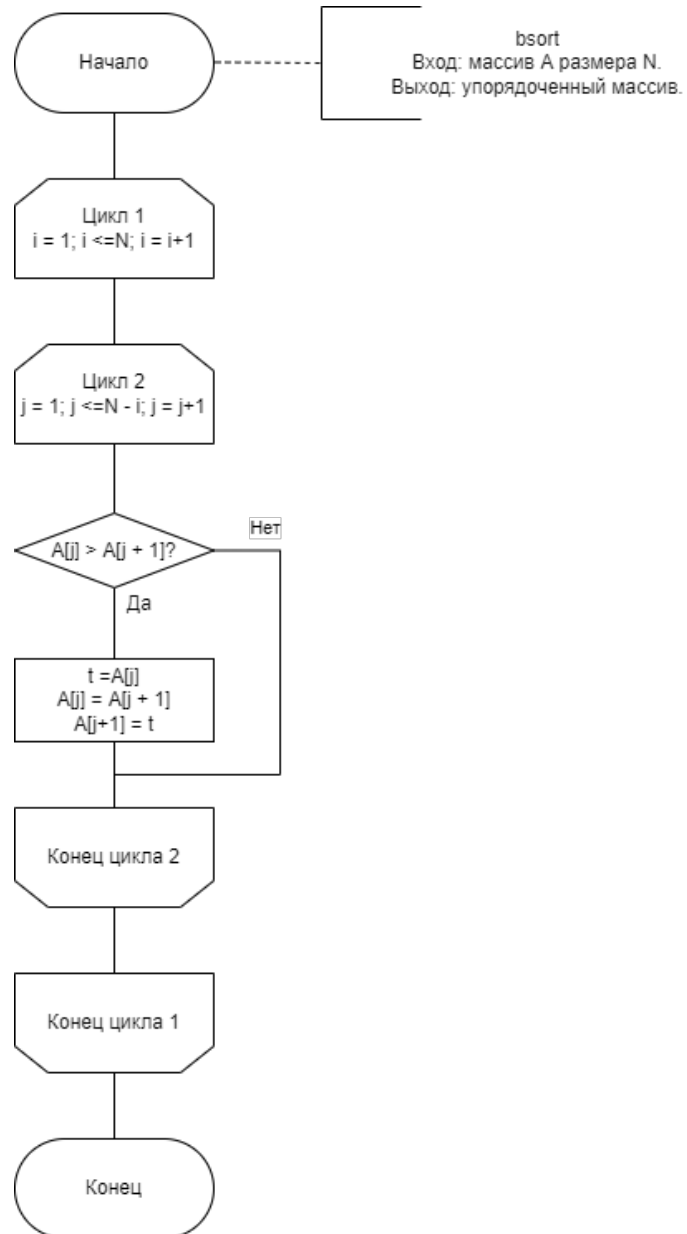


Рисунок 2.1 – Сортировка пузырьком

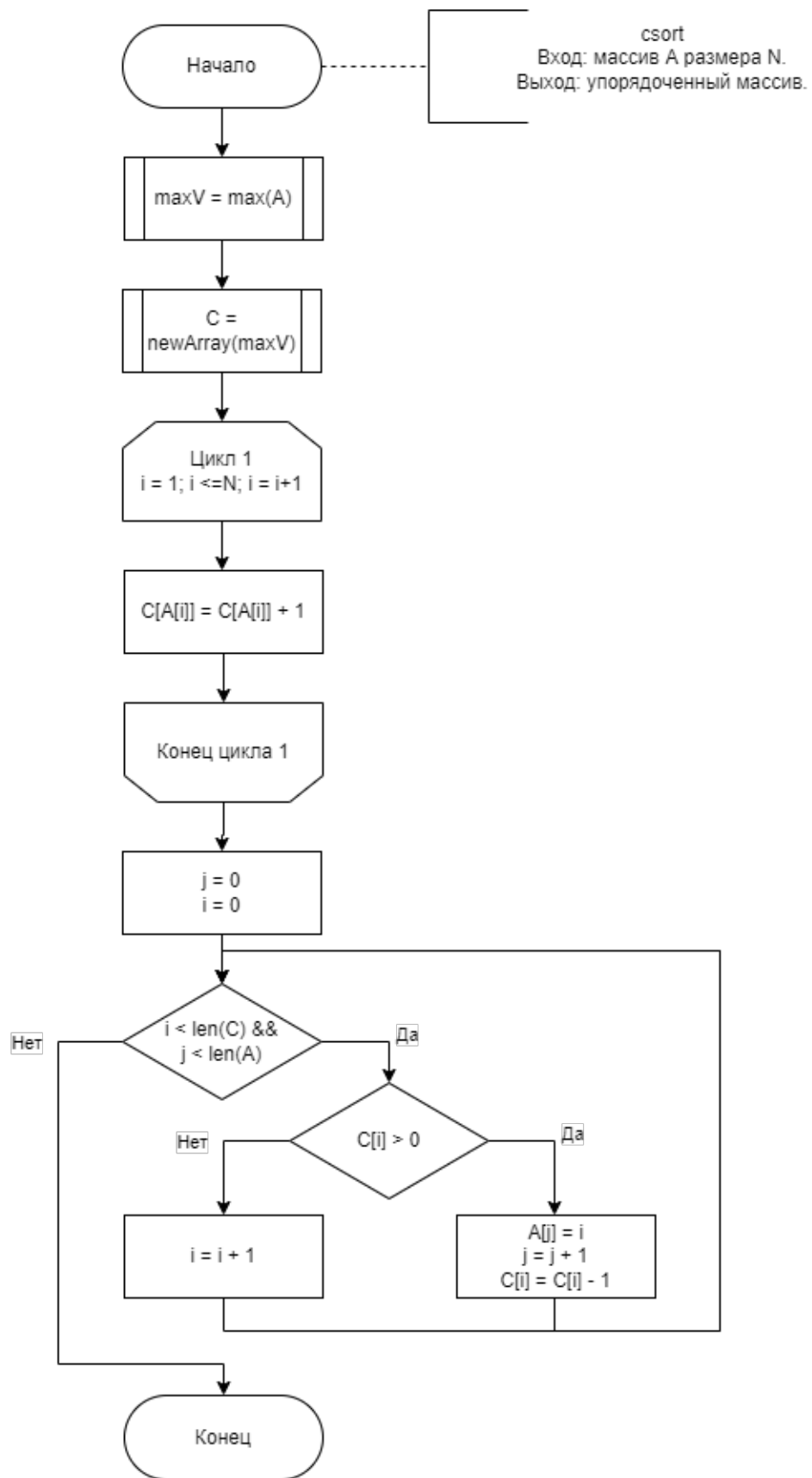


Рисунок 2.2 – Сортировка вставками

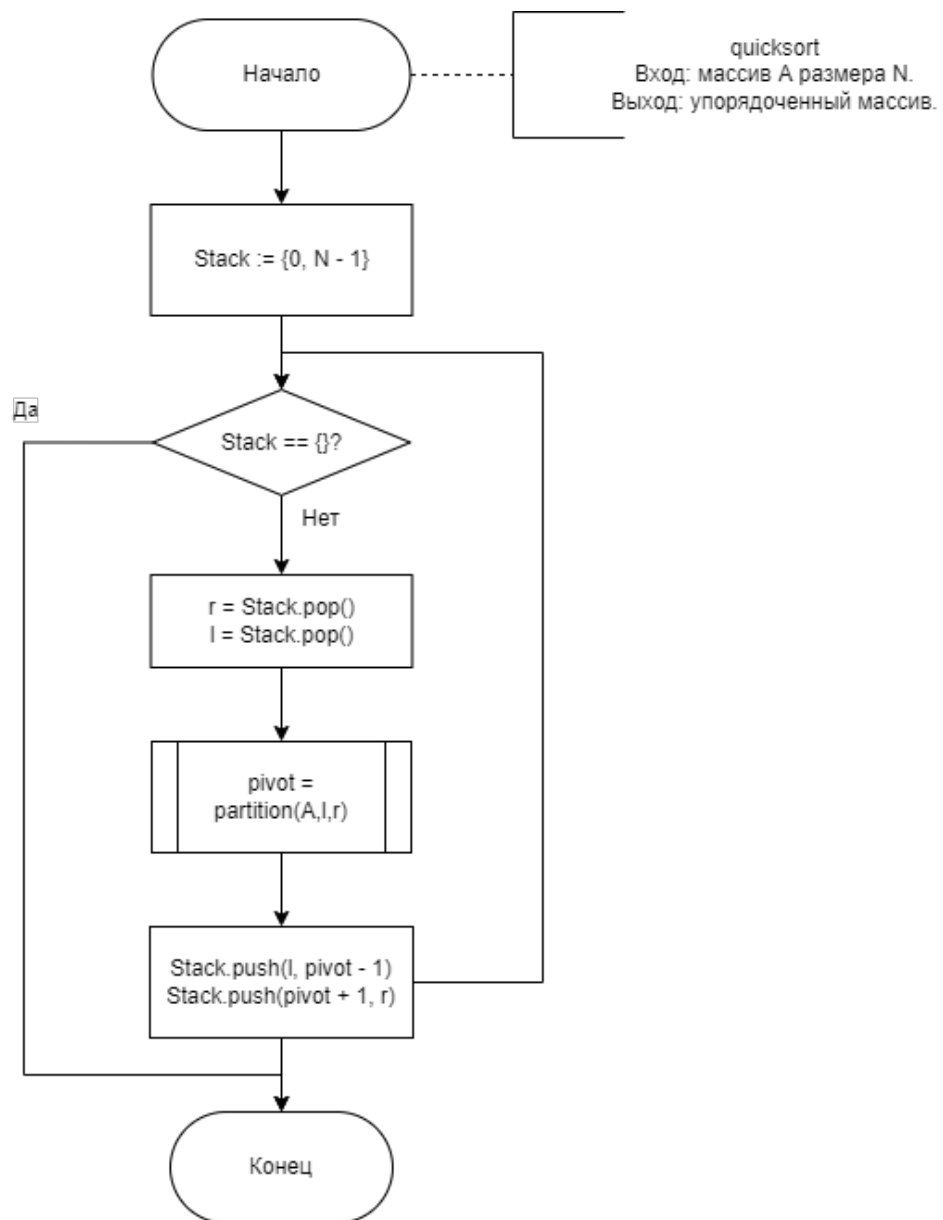


Рисунок 2.3 – Быстрая сортировка

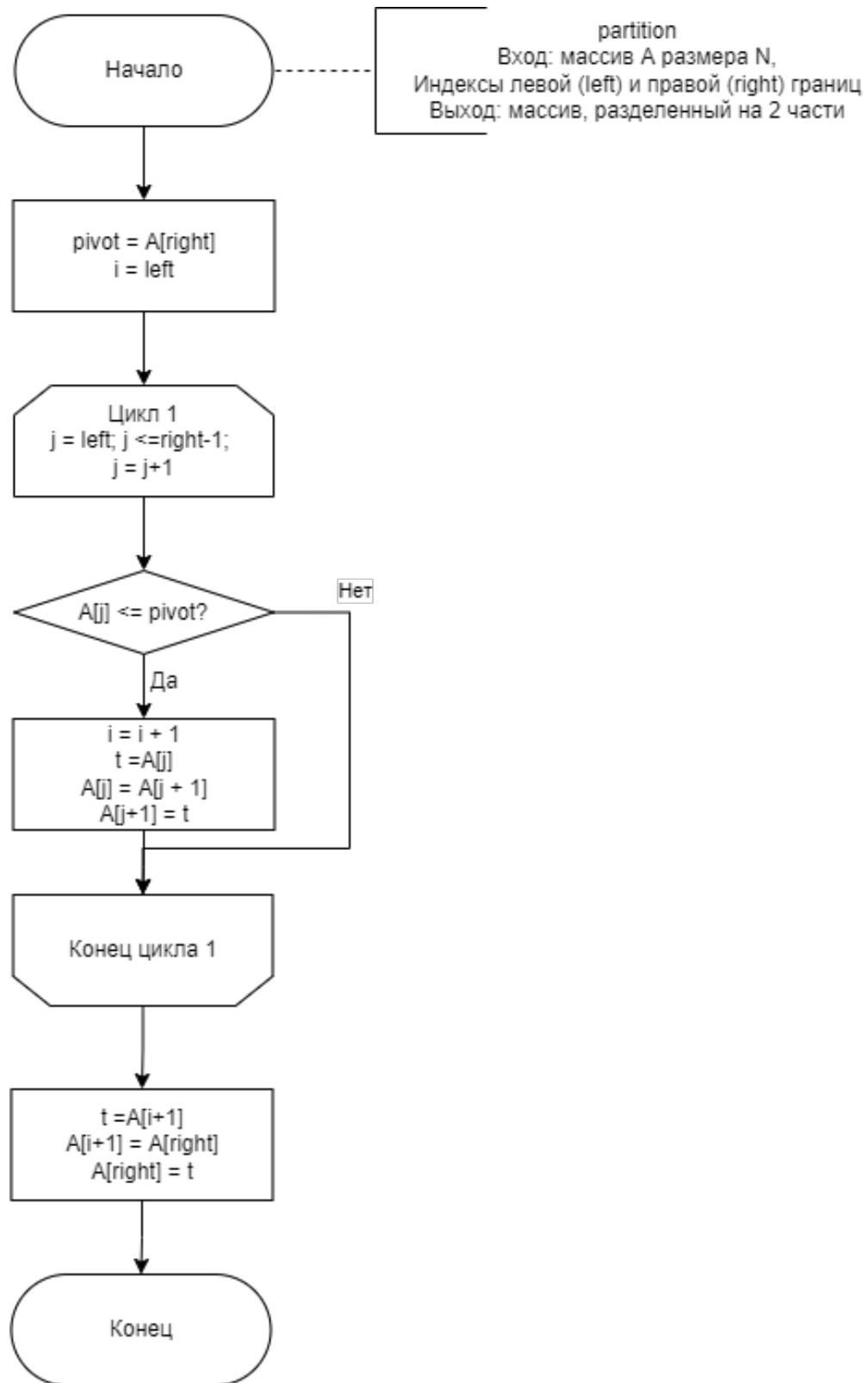


Рисунок 2.4 – Быстрая сортировка, разделение массива

## 2.2 Трудоёмкость алгоритмов

## 2.3 Модель вычислений

Для последующего вычисления трудоемкости введём модель вычислений:

1. Операции из списка (2.1) имеют трудоемкость 1.

$$+, -, /, \%, ==, !=, <, >, <=, >=, [], ++, -- \quad (2.1)$$

2. Трудоемкость оператора выбора if условие then A else B рассчитывается, как (2.2).

$$f_{if} = f_{\text{условия}} + \begin{cases} f_A, & \text{если условие выполняется,} \\ f_B, & \text{иначе.} \end{cases} \quad (2.2)$$

3. Трудоемкость цикла рассчитывается, как (2.3).

$$f_{for} = f_{\text{инициализации}} + f_{\text{сравнения}} + N(f_{\text{тела}} + f_{\text{инкремента}} + f_{\text{сравнения}}) \quad (2.3)$$

4. Трудоемкость вызова функции равна 0.

## 2.4 Трудоёмкость алгоритмов

Пусть размер массивов во всех вычислениях обозначается как  $N$ .

### 2.4.1 Алгоритм сортировки пузырьком

Трудоёмкость алгоритма сортировки пузырьком состоит из:

- трудоёмкость сравнения и инкремента внешнего цикла  $i \in [1..N)$  (2.4):

$$f_i = 2 + 2(N - 1) \quad (2.4)$$

- суммарная трудоёмкость внутренних циклов, количество итераций которых

меняется в промежутке  $[1..N - 1]$  (2.5):

$$f_j = 3(N - 1) + \frac{N \cdot (N - 1)}{2} \cdot (3 + f_{if}) \quad (2.5)$$

- трудоёмкость условия во внутреннем цикле (2.6):

$$f_{if} = 4 + \begin{cases} 0, & \text{в лучшем случае} \\ 9, & \text{в худшем случае} \end{cases} \quad (2.6)$$

Трудоёмкость в **лучшем** случае (2.7):

$$f_{best} = \frac{7}{2}N^2 + \frac{3}{2}N - 3 \approx \frac{7}{2}N^2 = O(N^2) \quad (2.7)$$

Трудоёмкость в **худшем** случае (2.8):

$$f_{worst} = 8N^2 - 8N - 3 \approx 8N^2 = O(N^2) \quad (2.8)$$

## 2.4.2 Алгоритм сортировки подсчетом

Трудоёмкость алгоритма сортировки подсчетом состоит из:

1. нахождение максимального числа в массиве  $A$ .
2. трудоёмкость инициализации массива  $C$  размером  $k$  нулями, где  $k = \max(A)$ .
3. проход по массиву  $A$  и подсчет количества вхождений каждого числа в массиве  $C$ .
4. проход по массиву  $C$  и запись чисел в массив  $A$  в соответствии с их количеством в массиве  $C$ .

Вычислим каждую трудоёмкость отдельно.

1. Трудоёмкость нахождения максимального числа в массиве  $A$  (2.9).

$$f_{max} = 2 + 2(N - 1) = 4N - 3 \quad (2.9)$$

2. Трудоёмкость инициализации массива  $C$ (2.10).

$$f_C = 2 + 2(k - 1) = 4k - 3 \quad (2.10)$$

3. Трудоёмкость подсчета количества вхождений каждого числа в массиве  $C$ (2.11).

$$f_{count} = 2 + 2(N - 1) = 4N - 3 \quad (2.11)$$

4. проход по массиву  $C$  и запись чисел в массив  $A$  в соответствии с их количеством в массиве  $C$ (2.12).

$$f_{write} = 6 + 7 * N + \begin{cases} 0, & \text{в лучшем случае} \\ 3k, & \text{в худшем случае} \end{cases} \quad (2.12)$$

Трудоёмкость в **лучшем** случае (2.13):

$$f_{best} = 15N + 4k - 3 \approx 15N + 4k = O(N + k) \quad (2.13)$$

Трудоёмкость в **худшем** случае (2.14):

$$f_{worst} = 15N + 7k - 3 \approx 15N + 7k = O(N + k) \quad (2.14)$$

### 2.4.3 Алгоритм итеративной быстрой сортировки

Трудоёмкость алгоритма итеративной быстрой сортировки (2.15):

$$T(N) = T(J) + T(N - J) + M(N) \quad (2.15)$$

где

1.  $T(N)$  - трудоемкость быстрой сортировки массива размера  $N$ .
2.  $T(J)$  - трудоемкость быстрой сортировки массива размера  $J$ .
3.  $T(N - J)$  - трудоемкость быстрой сортировки массива размера  $N - J$ .

4.  $M(N)$  - трудоёмкость разделения массива на две части.

**Вычислим для лучшего случая:**

- $T(N) = 2T(\frac{N}{2}) + C * N$  - трудоемкость быстрой сортировки массива размера  $N$ .
  - $2T(\frac{N}{2})$  поскольку мы разделяем массив на 2 равные части
  - $C * N$  поскольку мы будем проходить все элементы массива на каждом уровне "дерева"
- Следующий шаг - разделить дальше на 4 части ((2.16) и (2.17)):

$$T(N) = 2(2 * T(\frac{N}{4}) + C * N/2) + C * N \quad (2.16)$$

$$T(N) = 4T(\frac{N}{4}) + 2C * N \quad (2.17)$$

- В общем случае (2.18):

$$T(N) = 2^k * T(N/(2^k)) + k * C * N \quad (2.18)$$

- Для лучшего случая -  $N = 2^k$  - идеально распределенное дерево (отсюда следует, что  $k = \log_2(N)$ ) (2.19)

$$T(N) = 2^k * T(1) + k * C * (2^k) \quad (2.19)$$

- Вычислим  $T(1)$  - трудоемкость при массиве длиной 1 - и  $C$  - трудоемкость разделения (2.20):

$$T(1) = 6 + 1 * 9 = 15; C = 12 + N/(2^k) * 8 \quad (2.20)$$

- Полная сложность (при  $k = \log_2(N)$ ) (2.21):

$$T(N) = 15N + \log_2(N) * (12 + 8) = 15N + 20N\log_2(N) \quad (2.21)$$



Теперь вычислим для худшего случая:

- $T(N) = T(N - 1) + C * N$  - трудоемкость быстрой сортировки массива размера  $N$ .

–  $T(N - 1)$  поскольку мы разделяем массив на 2 неравные части: пустое множество и полное множество за исключением "середины"

- Следующие шаги очевидны (2.22), (2.23):

$$T(N) = T(N - 2) + C(N - 1) + C * N = T(N - 2) + 2C * N - C \quad (2.22)$$

$$T(N) = T(N - 3) + 3C * N - 2C * N - C \quad (2.23)$$

- В общем случае (2.24):

$$T(N) = T(N - k) + k * C * N - C\left(\frac{k(k - 1)}{2}\right) \quad (2.24)$$

- Для худшего случая -  $N = k$  - нераспределенное дерево (2.25):

$$T(N) = T(0) + N * C * N - C\left(\frac{N(N - 1)}{2}\right) \quad (2.25)$$

– Вычислим параметры (2.26):

$$T(0) = 1; C = 12 + (N - k) * 8 \quad (2.26)$$

- Полная сложность (при  $N = k$ ) (2.27):

$$T(N) = 1 + N * N * 12 - 12 * \left(\frac{N(N - 1)}{2}\right) = 1 + 6N^2 + 6N \quad (2.27)$$

Трудоёмкость в **лучшем** случае (2.28):

$$f_{best} = 15N + 20N \log_2(N) \approx 20N \log_2(N) = O(N \log_2(N)) \quad (2.28)$$

Трудоёмкость в **худшем** случае (2.29):

$$f_{worst} = 1 + 6N^2 + 6N \approx 6N^2 = O(N^2) \quad (2.29)$$

## Вывод

На основе теоретических данных, полученных из аналитического раздела, были построены схемы трёх алгоритмов сортировки. Оценены их трудоёмкости в лучшем и худшем случаях.

## 3 Технологическая часть

В данном разделе приведены требования к программному обеспечению, средства реализации и листинги кода.

### 3.1 Требования к программному обеспечению

К программе предъявляется ряд условий:

- на вход программе подается размер массива (целое число) и массив (целые числа);
- на выходе программа должна выдать отсортированный массив одним из 3 способов;

### 3.2 Средства реализации

Для реализации данной лабораторной работы необходимо установить следующее программное обеспечение:

- Golang 1.19.1 - язык программирования
- Benchdraw 0.1.0 - Средство визуализации данных
- LaTeX - система документооборота

### 3.3 Листинги кода

В листингах 3.1, 3.2 и 3.3 приведены реализации алгоритмов сортировки пузырьком, подсчетом и итеративная быстрая соответственно.

Листинг 3.1 – Сортировка массива пузырьком

```
1 func BubbleSort[T constraints.Ordered](array []T) {
2     n := len(array) - 1
3     for i := 0; i < n; i++ {
4         for j := 0; j < n-i; j++ {
5             if array[j] > array[j+1] {
6                 array[j], array[j+1] = array[j+1], array[j]
7             }
8         }
9     }
10 }
```

### Листинг 3.2 – Сортировка массива подсчетом

```
1 func CountingSort(array []int) {
2     maxVal := array[0]
3     for _, val := range array {
4         if val > maxVal {
5             maxVal = val
6         }
7     }
8     temp := make([]int, maxVal+1)
9     for _, val := range array {
10         temp[val]++
11     }
12
13     j, i := 0, 0
14
15     for i < len(temp) && j < len(array) {
16         if temp[i] > 0 {
17             array[j] = i
18             j++
19             temp[i]--
20         } else {
21             i++
22         }
23     }
24 }
```

### Листинг 3.3 – Итеративная быстрая сортировка массива

```
1
2 func partition[T constraints.Ordered](arr []T, left int, right int) int {
3     pivot := arr[right]
4     i := left - 1
5     for j := left; j <= right-1; j++ {
6         if arr[j] <= pivot {
7             i++
8             arr[i], arr[j] = arr[j], arr[i]
9         }
10    }
11    arr[i+1], arr[right] = arr[right], arr[i+1]
12
13    return i + 1
14 }
15
16 func Quicksort[T constraints.Ordered](array []T) {
17     if len(array) < 2 {
18         return
19     }
20 }
```

```

19 }
20
21 left := 0
22 right := len(array) - 1
23 stack := Stack[[2]int]{}
24 stack.Push([2]int{left, right})
25 for !stack.IsEmpty() {
26     lr := stack.Pop()
27     left, right = lr[0], lr[1]
28     if right <= left {
29         continue
30     }
31     pivot := partition(array, left, right)
32     stack.Push([2]int{left, pivot - 1})
33     stack.Push([2]int{pivot + 1, right})
34 }
35 }

```

В таблице 3.1 приведены тесты для функций, реализующих алгоритмы сортировки. Все тесты пройдены успешно.

Таблица 3.1 – Тестирование функций

Входной массив	Результат	Ожидаемый результат
[1, 2, 3, 4, 5]	[1, 2, 3, 4, 5]	[1, 2, 3, 4, 5]
[5, 4, 3, 2, 1]	[1, 2, 3, 4, 5]	[1, 2, 3, 4, 5]
[-1, -2, -3, -2, -5]	[-5, -3, -2, -2, -1]	[-5, -3, -2, -2, -1]
[-2, 7, 0, -1, 3]	[-2, -1, 0, 3, 7]	[-2, -1, 0, 3, 7]
[50]	[50]	[50]
[-40]	[-40]	[-40]
Пустой массив	Пустой массив	Пустой массив

## Вывод

В данном разделе был разработан исходный код трёх алгоритмов сортировки: пузырьком, вставками и быстрая сортировка.

## 4 Исследовательская часть

Ниже приведены технические характеристики устройства, на котором было проведено тестирование ПО:

- Операционная система: Windows 11, используя Windows Subsystem for Linux 2 (WSL2)[1] Имитирующей Arch Linux[2] 64-bit.
- Оперативная память: 16 ГБ.
- Процессор: 11th Gen Intel® Core™ i5-11320H @ 3.20ГГц[3].

### 4.1 Время выполнения алгоритмов

Алгоритмы замерылись при помощи инструментов замера времени, предоставляемых встроенными средствами языка Go[4]. Пример функции по замеру времени приведен в листинге 4.1. Количество повторов регулируется тестирующей системой самостоятельно, хотя при желании оные можно задать. Точное количество повторов определяется в зависимости от того, был ли получен стабильный результат согласно системе.

Листинг 4.1 – Пример бенчмарка

```
1 func BenchmarkQuicksortSorted(b *testing.B) {
2     for steps, amount := 0, 0; steps < STEPS; steps++ {
3         amount += INC
4         b.Run(fmt.Sprintf("size=%d", amount), func(b *testing.B) {
5             sample := GenerateSortedArray(amount)
6             sampleCopy := make([]int, amount)
7             b.ResetTimer()
8             for i := 0; i < b.N; i++ {
9                 //b.StopTimer() // Disabled Due performance reasons
10                copy(sampleCopy, sample)
11                //b.StartTimer() // Disabled Due performance reasons
12                Quicksort(sampleCopy)
13            }
14        })
15    }
16 }
```

График времени выполнения сортировок на отсортированных данных (в наносекундах)

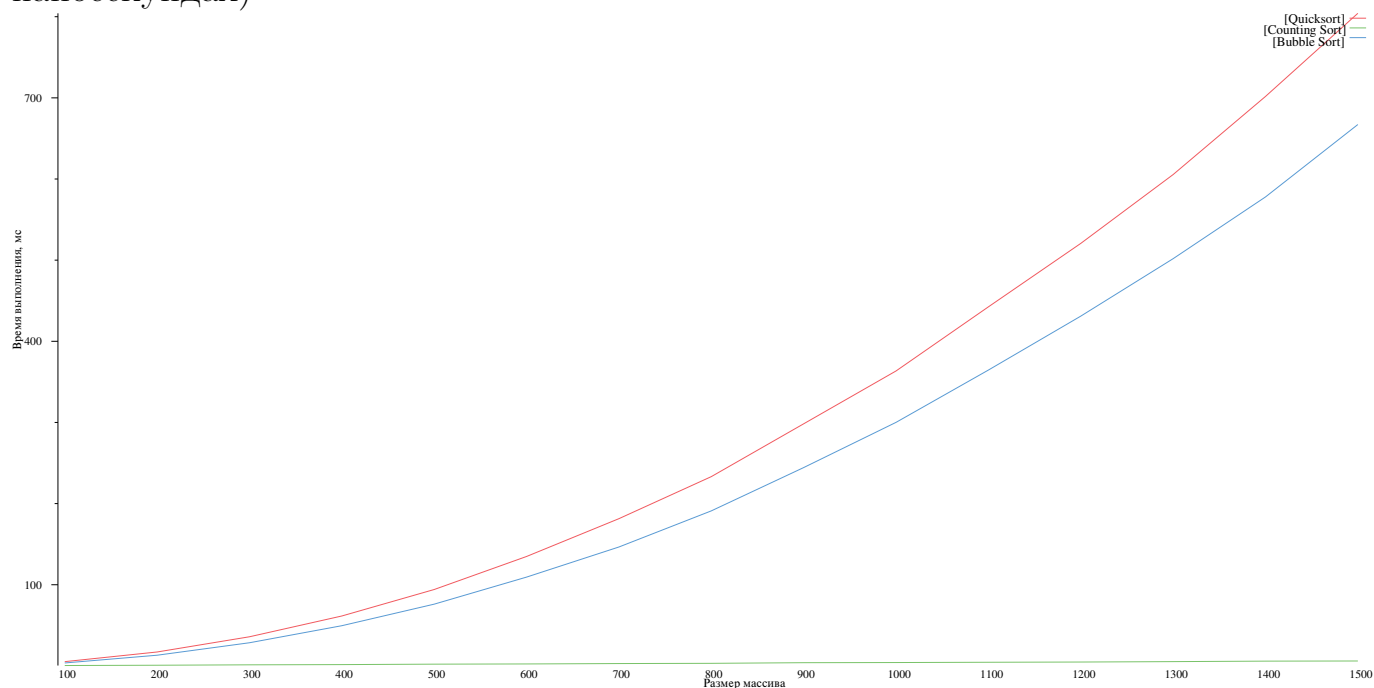


График времени выполнения сортировок на отсортированных данных в обратном порядке (в наносекундах)

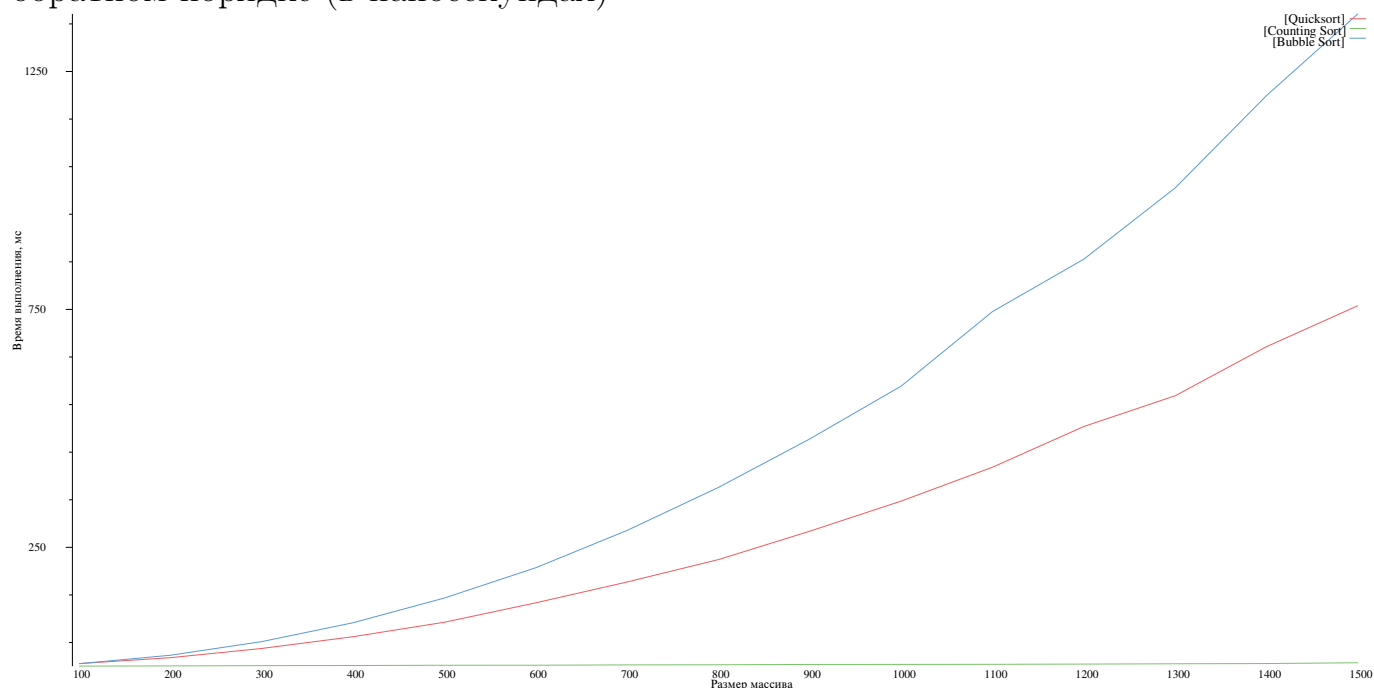
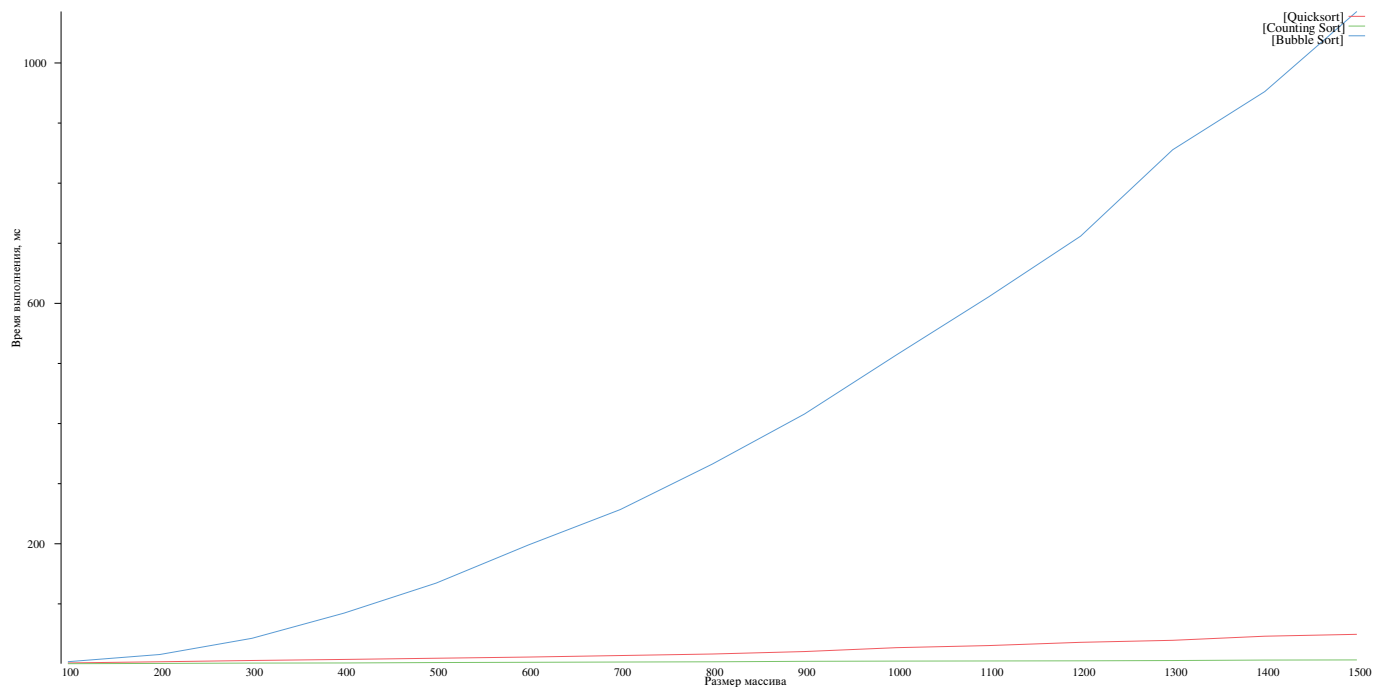


График времени выполнения сортировок на случайных данных (в наносекундах)



## Вывод

Как и можно было ожидать, сортировка подсчетом оказалась самой быстрой, а сортировка пузырьком самой медленной при случайных данных. Быстрая сортировка оказалась быстрой, но при отсортированных данных она оказалась медленнее сортировки пузырьком из-за сложности алгоритма.



## ЗАКЛЮЧЕНИЕ

В ходе выполнения лабораторной работы были достигнуты следующие задачи:

- были изучены и реализованы 3 алгоритма сортировки: пузырьёк, подсчетом, быстрая сортировка;
- были приведены аналитические данные о выше перечисленных алгоритмах;
- были приведены подробные блок-схемы, описывающие алгоритмы;
- был проведён сравнительный анализ алгоритмов на основе экспериментальных данных.

Экспериментальные данные показали различные сильные и слабые стороны каждого алгоритма. Так например:

- сортировка пузырьком работает крайне медленно независимо от входных данных;
- быстрая сортировка работает быстрее на случайных данных, поэтому лучше всего подходит как общее решение абстрактной задачи на сортировку данных;
- быстрая сортировка работает медленно на отсортированном массиве.
- сортировка подсчетом работает быстрее всех выше перечисленных алгоритмов, однако требует сильно больше памяти, чем остальные алгоритмы.

## СПИСОК ИСПОЛЬЗОВАННЫХ ИСТОЧНИКОВ

1. Windows Subsystem for Linux 2 [Электронный ресурс]. — Дата обращения: 19.09.2022. Режим доступа: <https://learn.microsoft.com/en-us/windows/wsl/about>.
2. Arch Linux [Электронный ресурс]. — Дата обращения: 19.09.2022. Режим доступа: <https://archlinux.org/>.
3. Процессор Intel® Core™ i5-11320H [Электронный ресурс]. — Дата обращения: 19.09.2022. Режим доступа: <https://ark.intel.com/content/www/ru/ru/ark/products/217183/intel-core-i511320h-processor-8m-cache-up-to-4-50-ghz-with-ipu.html>.
4. Go [Электронный ресурс]. — Дата обращения: 19.09.2022. Режим доступа: <https://go.dev/>.