МИНОБРНАУКИ РОССИИ

Федеральное государственное бюджетное образовательное учреждение высшего образования

«МИРЭА – Российский технологический университет»

Отчет

Практическая работа №8

Дисциплина Структуры и алгоритмы обработки данных

Тема. Определение эффективного алгоритма сортировки

Выполнил студент Черномуров С. А.

Фамилия И.О.

Группа ИКБО-13-21

Номер группы

Москва 2022

Условие задания:

Разработать три алгоритма сортировки, определенные вариантом. Провести анализ вычислительной и емкостной сложности алгоритма на массивах, заполненных случайно. Определить наиболее эффективный алгоритм.

Задание 1

Вариант №23

1. Требования к выполнению задачи:

    1. Разработать алгоритм сортировки одномерного целочисленного массива A[n] и реализовать его функцией, используя алгоритм согласно варианту, индивидуального задания - *Алгоритм задания 1.* Провести тестирование программы на исходном массиве, сформированном вводом с клавиатуры, т.е. доказать ее работоспособность.
    2. Разработать функцию заполнения рабочего массива A с использованием генератора псевдослучайных чисел.
    3. Провести экспериментальную оценку вычислительной сложности алгоритма, для чего выполнить контрольные прогоны программы для размеров массива n = 100, 1000, 10000, 100000 и 1000000 элементов с вычислением времени T(n) выполнения T(n) – (в миллисекундах/секундах). Полученные результаты свести в сводную таблицу Таблица 1.
    4. Провести эмпирическую оценку вычислительной сложности алгоритма, определив функцию зависимости времени выполнения алгоритма от размера массива(задачи) и показать ее результат в таблице 1 в столбце $\mathbf{T_{эт}}$ =f(C+M).
    5. Провести эмпирическую оценку фактического количества операций сравнения Сф и количества операций перемещения Мф. Полученные результаты вставить в сводную таблицу в столбец $T_{эп}$=Сф+Мф.

Таблица 1. Сводная таблица результатов

n	**T(n) время в мс/сек**	**$T_{эт}$=f(C+M)-функция**	**$T_{эп}$=Сф+Мф-количество**
100			
1000			
10000			
100000			
1000000			

6. Провести дополнительные прогоны программы на массивах, отсортированных:

   А) строго в убывающем порядке значений элементов,

   результаты представить в сводной таблице 2 по формату *Таблица 1*;

   Б) строго в возрастающем порядке значений элементов,

   результаты представить в сводной таблице 3 по формату *Таблица 1*;

   В) Провести анализ зависимости (или независимости) алгоритма сортировки от исходной упорядоченности массива**.** Выводы описать.

7. Провести анализ полученных результатов и указать порядок роста сложности алгоритма при увеличении размера входных данных.
8. Сделать выводы о проделанной работе, основанные на полученных результатах.

2. Постановка задачи:

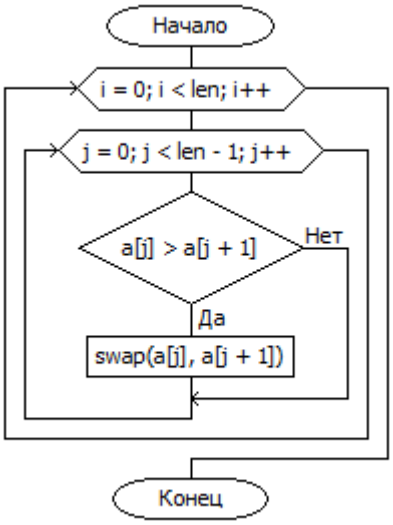
   Дано. Целочисленный массив и его длина.

   Результат. Отсортированный целочисленный массив.

3. Модель решения:

   Попарно сравниваются соседние элементы массива. Если элемент с меньшим индексом больше, чем элемент с большим индексом, то значения элементов массива меняются местами. Таким образом элементы массива сравниваются до тех пор, пока он не отсортируется.

4. Блок-схема функции сортировки простыми обменами:



5. **Сортировка массива простыми обменами:**

   Предусловие. a – целочисленный массив, len – длина массива.

Постусловие. a – отсортированный по неубыванию целочисленный массив.

```
void bubble_sort(int* a, int len);
```

Тест функции:

Номер теста	Исходные данные	Ожидаемый результат
1	a = {4, 3, 2}, len = 3	a = {2, 3, 4}
2	a = {1, 2, 3, 4}, len = 4	a = {1, 2, 3, 4}

```
void bubble_sort(int* a, int len) {
	for (int i = 0; i < len; i++)
		for (int j = 0; j < len-1; j++)
			if (a[j] > a[j+1]) swap(a[j], a[j+1]);
}
```

Асимптотическая сложность алгоритма сортировки простыми обменами $O(n^2)$

Тестирование программы

```
Лабораторная работа №8 ИКБО-13-21 Черномуров С.А. Вариант 23

Выберите сортировку:
1) Простыми обменами
2) Подсчётом
3) Быстрая сортировка с опорным последним элементом
0) Закончить программу
1
```

```
Введите способ заполнения массива:
1) С клавиатуры
2) Случайными числами
2
Введите длину массива: 10
Сгенерированный массив:
40 54 26 43 57 80 91 24 26 22
Отсортированный массив:
22 24 26 26 40 43 54 57 80 91
```

```
Введите способ заполнения массива:
1) С клавиатуры
2) Случайными числами
2
Введите длину массива: 100
Сгенерированный массив:
96 69 5  35 96 99 35 28 11 9  46 58 48 6  5  69 61 55 2  91 59 81 8  32 22 72 46 58 30 27 23 80 54 17 66 72 11 75 75 75
64 46 18 88 67 28 8  51 21 82 29 3  17 65 21 41 42 42 29 72 62 18 99 10 33 93 84 24 3  61 78 69 33 55 26 79 10 82 68 39
41 98 33 86 25 63 7  15 62 42 91 22 86 12 8  30 14 51 64 8
Отсортированный массив:
2  3  3  5  5  6  7  8  8  8  8  9  10 10 11 11 12 14 15 17 17 18 18 21 21 22 22 23 24 25 26 27 28 28 29 29 30 30 32 33
33 33 35 35 39 41 41 42 42 42 46 46 46 48 51 51 54 55 55 58 58 59 61 61 62 62 63 64 64 65 66 67 68 69 69 69 72 72 72 75
75 75 78 79 80 81 82 82 84 86 86 88 91 91 93 96 96 98 99 99
```

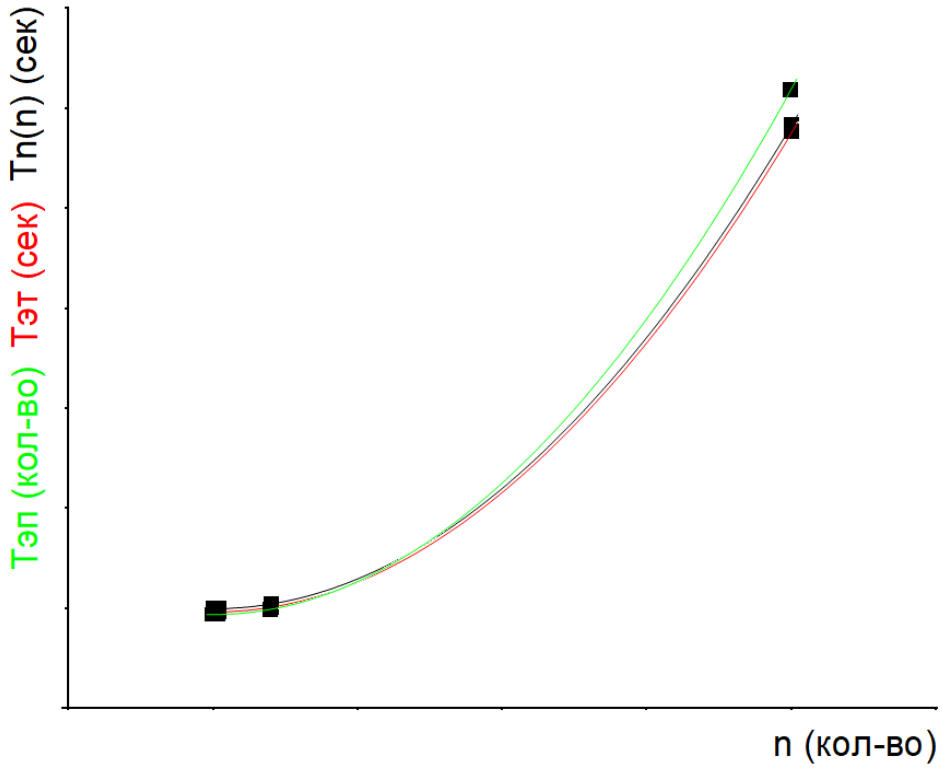
Время работы алгоритма для случайного массива

n	T(n) время в секундах	$T_{эт}$=f(C+M)-функция, секунд	$T_{эп}$=Сф+Мф-количество
100	0.001	0.00012	15000
1000	0.013	0.012	1500000
10000	1.190	1.2	150000000
100000	118.704	120	15000000000
1000000	12107.222	12000	1500000000000

$T_{эт}(n)=n^2/100\ 000\ 000*1.2$, где (100 000 000 – примерное количество операций, выполняемых в C++ за секунду, 1.2 – погрешность отклонения от «идеальных» теоретических результатов (во всех прогонах она появляется))

$T_{эп}(n)=n^2+0.5*n*n=1.5n^2$, где ($n^2$-количество сравнений, 0.5*n*n – количество перестановок элементов (за один проход 0.5*n перестановок (учитывается нормальное распределение, примерно половина пар элементов в массиве уже изначально отсортированы, проверено экспериментально) за n проходов по массиву))

График зависимости



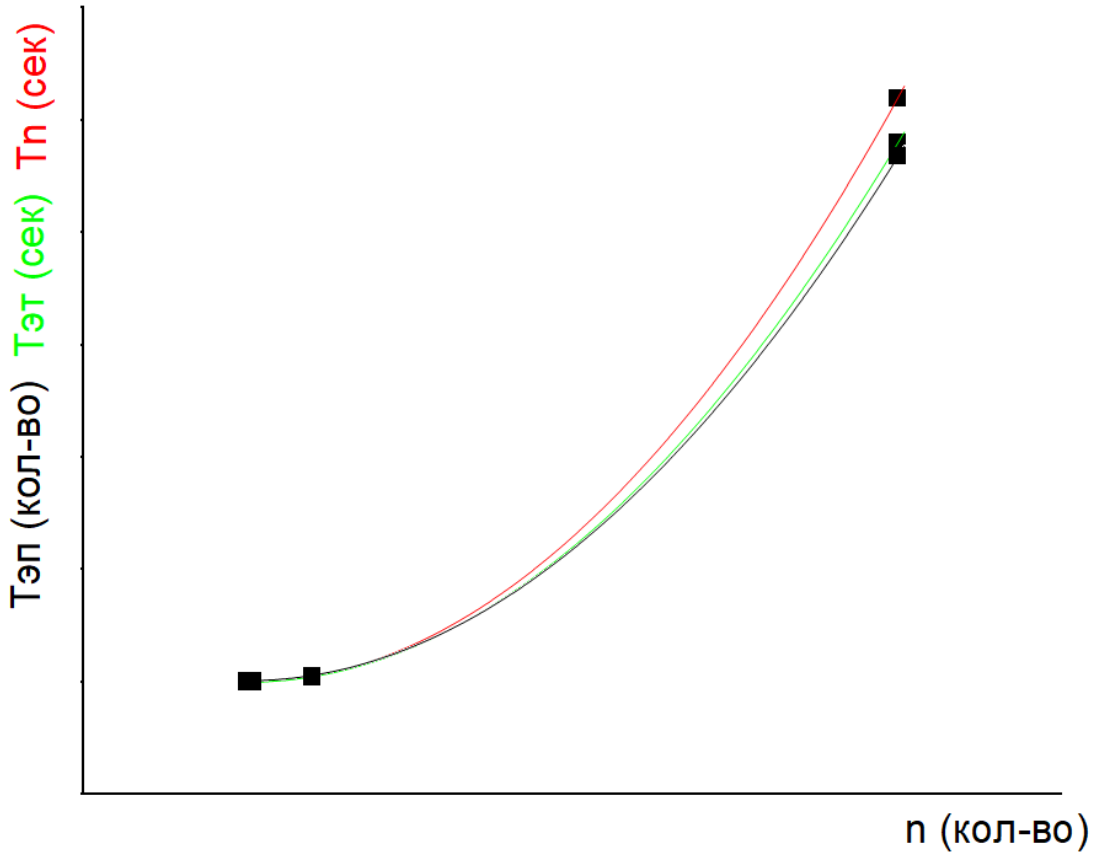
Время работы алгоритма для полностью отсортированного массива

n	T(n) время в секундах	$T_{эт}$=f(C+M)-функция, секунд	$T_{эп}$=Сф+Мф-количество
100	0	0.000012	10000
1000	0.002	0.0012	1000000
10000	0.121	0.12	100000000
100000	12.328	12	10000000000
1000000	1300.199	1200	1000000000000

$T_{эт}(n)=n^2/100\ 000\ 000/10*1.2$, где (100 000 000 – примерное количество операций, выполняемых в C++ за секунду, 1.2 – погрешность отклонения от «идеальных» теоретических результатов (во всех прогонах она появляется))

$T_{эп}(n)=n^2$, где (n^2-количество сравнений)

График зависимости



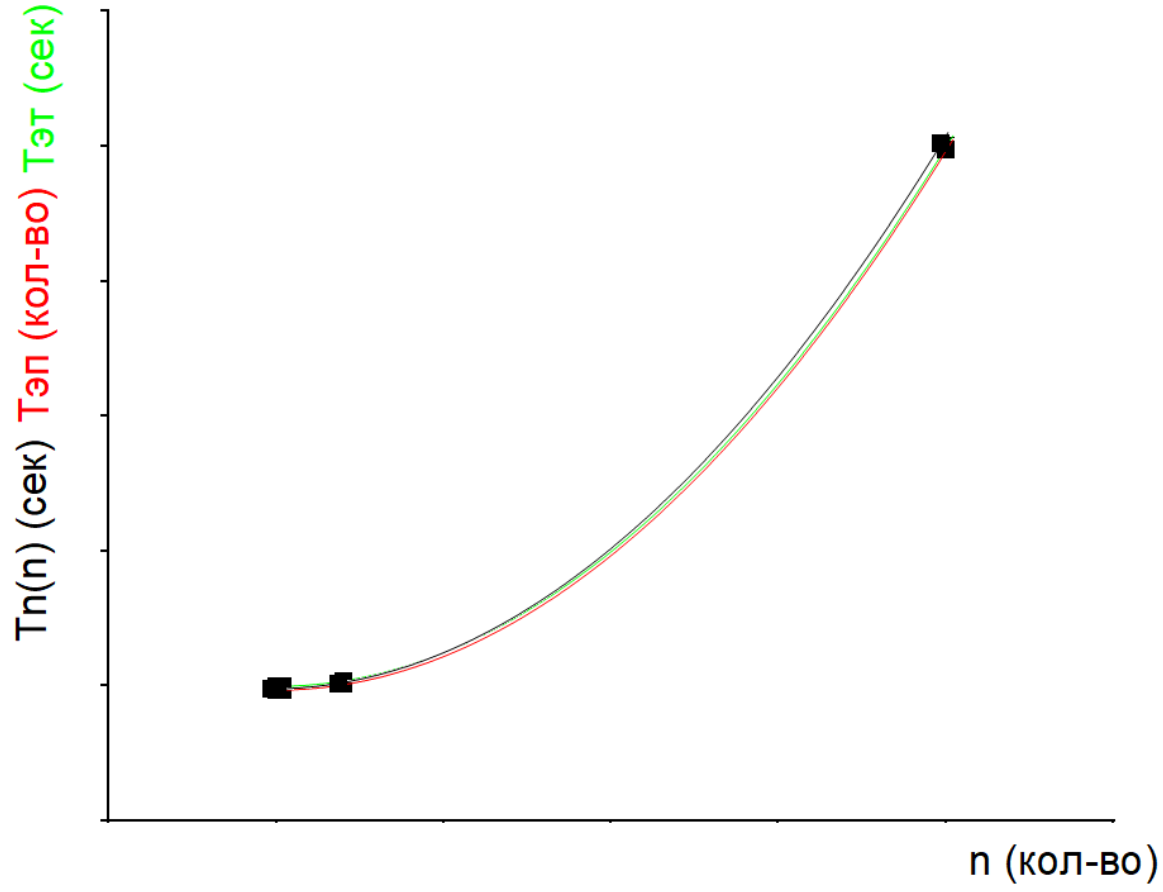
Время работы алгоритма для массива, отсортированного по убыванию

n	T(n) время в секундах	$T_{эт}$=f(C+M)-функция, секунд	$T_{эп}$=Сф+Мф-количество
100	0.001	0.0002	20000
1000	0.020	0.020	2000000
10000	2.072	2	200000000
100000	205.731	200	20000000000
1000000	20208.640	20000	2000000000000

$T_{эт}(n)=n^2/100\ 000\ 000*2$, где (100 000 000 – примерное количество операций, выполняемых в C++ за секунду)

$T_{эп}(n)=n^2+n^2=2n^2$, где (n^2-количество сравнений, n^2 – количество перестановок элементов)

График зависимости

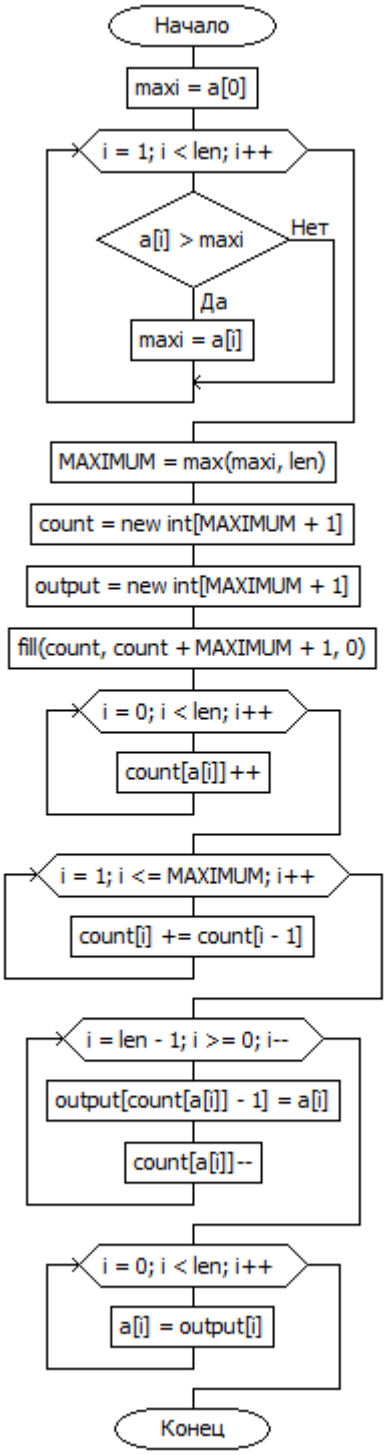


6. Выводы по эффективности работы алгоритма:
   Алгоритм сортировки простыми обменами является неэффективным с точки зрения скорости работы, но его плюс заключается в том, что это один из самых простых методов сортировки, его легче всего написать. Этот алгоритм подходит в качестве учебного примера, либо для сортировки небольших массивов (до 10000 элементов, дальше эффективность снижается слишком сильно). Также время сортировки массива сильно отличается при разных входных данных:
   При передаче случайного массива $O(1.2n^2)$
   При передаче отсортированного по возрастанию массива $O(0.12n^2)$
   При передаче отсортированного по убыванию массива $O(2n^2)$
   (коэффициенты оставлены для того, чтобы показать различие во времени)

Задание 2

Вариант №23

1. Требования к выполнению задачи:
    1. Разработать алгоритм усовершенствованной сортировки (задача 2), определенной в варианте, реализовать алгоритм. Сформировать таблицу результатов сортировки по формату Таблица 1 для массива, заполненного случайными числами. Определить емкостную сложность алгоритма. *Определить асимптотическую сложность алгоритма.*
    2. Провести дополнительные прогоны программы для оценки эффективности алгоритмов в наилучшем и наихудшем случаях. Сформировать таблицы 5 и 6.
    3. Выполнить анализ полученных результатов по таблицам 4, 5, 6.
    4. Определить эффективный из алгоритмов (задача 1) и (задача 2) по временной и емкостной сложности.
    5. Представить график зависимости Сф+Мф для анализируемых алгоритмов.
2. Постановка задачи:
   Дано. Целочисленный массив и его длина.
   Результат. Отсортированный целочисленный массив.
3. Модель решения:
   Сортировка подсчетом – это алгоритм сортировки, который сортирует элементы массива путем подсчета количества вхождений каждого уникального элемента в массиве. Количество хранится во вспомогательном массиве, а сортировка выполняется путем сопоставления количества как индекса вспомогательного массива.
4. Блок-схема функции сортировки подсчетом:



5. **Сортировка массива подсчетом:**

   Предусловие. a – целочисленный массив, len – длина массива.

   Постусловие. a – отсортированный по неубыванию целочисленный массив.

```
void count_sort(int* a, int len);
```

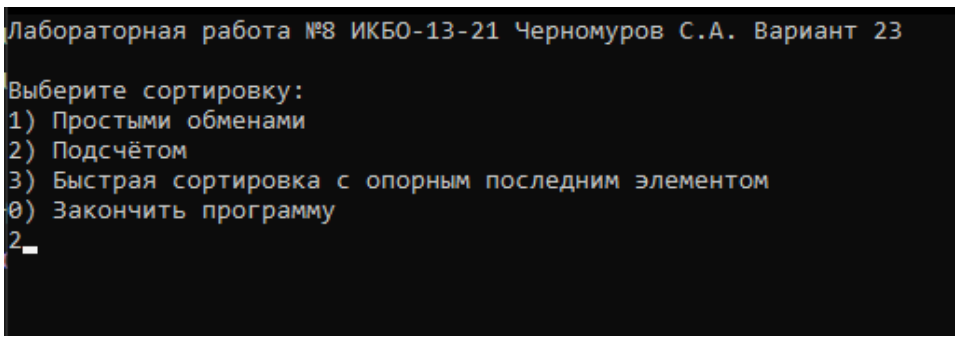
Тест функции:

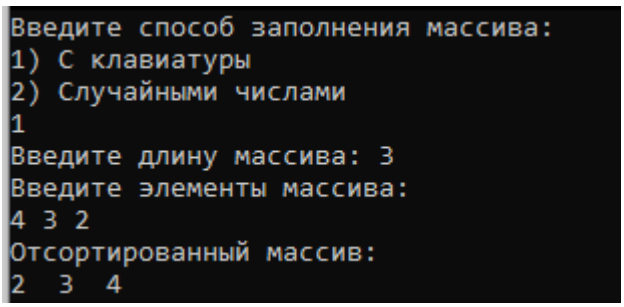
Номер теста	Исходные данные	Ожидаемый результат
1	a = {4, 3, 2}, len = 3	a = {2, 3, 4}
2	a = {1, 2, 3, 4}, len = 4	a = {1, 2, 3, 4}

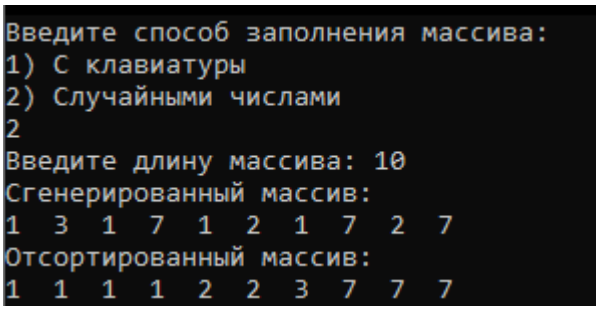
```
void count_sort(int* a, int len) {
	int maxi = a[0];

	for (int i = 1; i < len; i++)
		if (a[i] > maxi) maxi = a[i];

	int MAXIMUM = max(maxi, len);

	int* count = new int[MAXIMUM+1];
	int* output = new int[MAXIMUM+1];

	fill(count, count+MAXIMUM + 1, 0);

	for (int i = 0; i < len; i++)
		count[a[i]]++;

	for (int i = 1; i <= MAXIMUM; i++)
		count[i] += count[i - 1];

	for (int i = len - 1; i >= 0; i--) {
		output[count[a[i]] - 1] = a[i];
		count[a[i]]--;
	}

	for (int i = 0; i < len; i++) {
		a[i] = output[i];
	}
}
```

Асимптотическая сложность алгоритма сортировки подсчетом сильно зависит от входных данных, поэтому сложность будет зависеть от максимального элемента входного массива. Сложность сортировки подсчетом составляет O(n+k), где k-значение максимального элемента массива, а затраты по памяти составляют O(n+k).

Тестирование программы







Время работы алгоритма для случайного массива

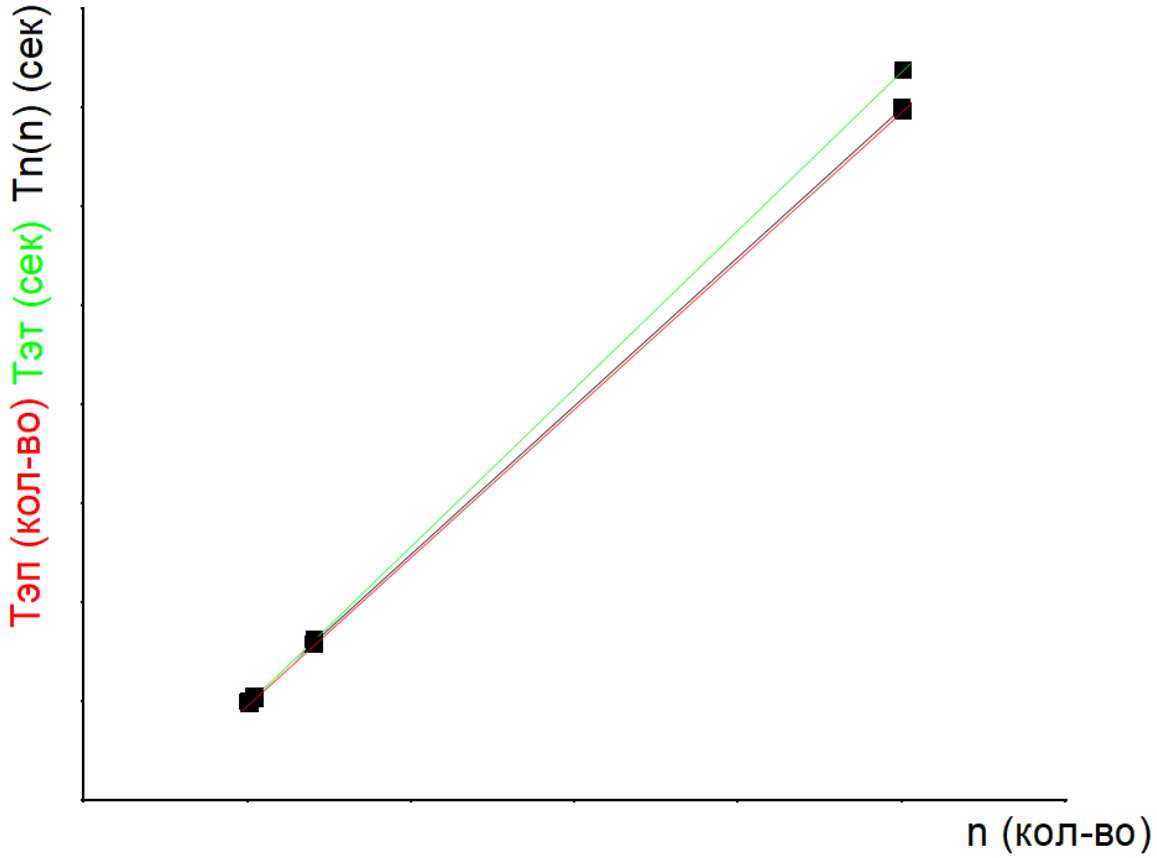
n	**T(n) время в секундах**	**$T_{эт}$=f(C+M)-функция, секунд**	**$T_{эп}$=Сф+Мф-количество**
10000	0.000	0.00015	60000
100000	0.002	0.0015	600000
1000000	0.015	0.015	6000000
10000000	0.145	0.150	60000000
100000000	1.502	1.500	600000000

Значения n увеличены, так как clock() не может улавливать настолько маленькие промежутки времени

$T_{эт}(n)=1.5n/100\ 000\ 000$, где (100 000 000 – примерное количество операций, выполняемых в C++ за секунду)

$T_{эп}(n)=n+k+n+k+n+n=4n+2k$, где (k – максимальное между (максимальный элемент массива, длина массива))

График зависимости



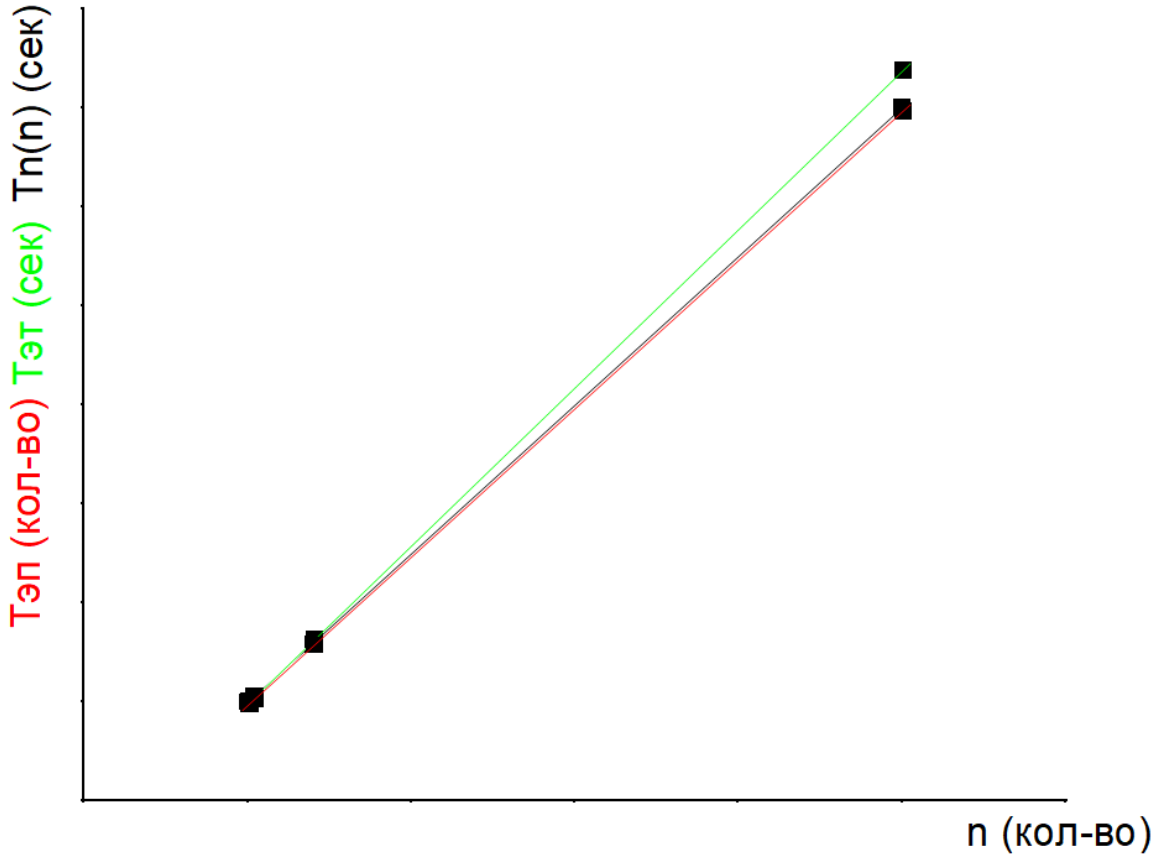
Время работы алгоритма для полностью отсортированного массива

n	T(n) время в секундах	$T_{эт}$=f(C+M)-функция, секунд	$T_{эп}$=Сф+Мф-количество
10000	0.000	0.00015	60000
100000	0.002	0.0015	600000
1000000	0.013	0.015	6000000
10000000	0.138	0.150	60000000
100000000	1.457	1.500	600000000

$T_{эт}(n)=1.5n/100\ 000\ 000$, где (100 000 000 – примерное количество операций, выполняемых в C++ за секунду)

$T_{эп}(n)=n+k+n+k+n+n=4n+2k$, где (k – максимальное между (максимальный элемент массива, длина массива))

График зависимости



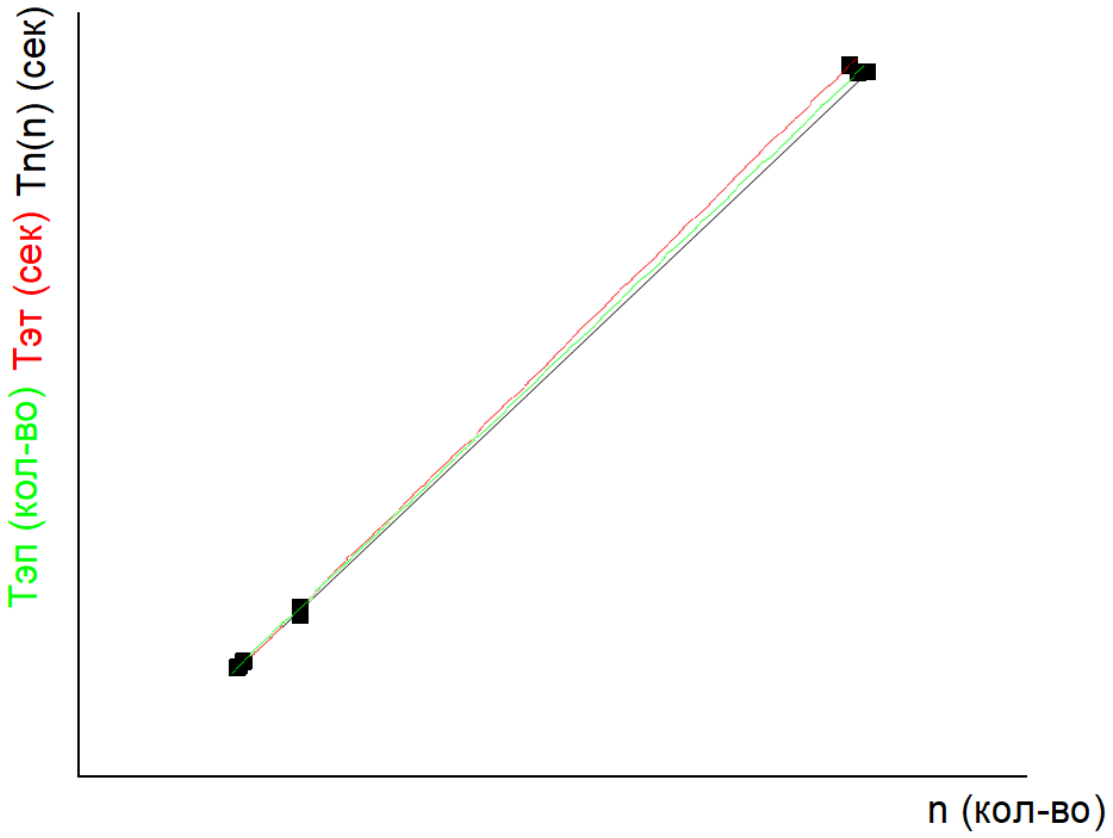
Время работы алгоритма для массива, отсортированного по убыванию

n	T(n) время в секундах	$T_{эт}$=f(C+M)-функция, секунд	$T_{эп}$=Сф+Мф-количество
10000	0.001	0.00015	60000
100000	0.002	0.0015	600000
1000000	0.013	0.015	6000000
10000000	0.121	0.150	60000000
100000000	1.369	1.500	600000000

$T_{эт}$(n)=1.5n/100 000 000, где (100 000 000 – примерное количество операций, выполняемых в C++ за секунду)

$T_{эп}$(n)=n+k+n+k+n+n=4n+2k, где (k – максимальное между (максимальный элемент массива, длина массива))

График зависимости



6. Выводы по эффективности работы алгоритма:

   Алгоритм сортировки подсчетом является эффективным с точки зрения скорости работы. Также время сортировки массива не отличается при разных входных данных:
   При передаче случайного массива O(n+k)
   При передаче отсортированного по возрастанию массива O(n+k)
   При передаче отсортированного по убыванию массива O(n+k)
   Сортировка подсчетом является гораздо более эффективным алгоритмом, чем сортировка простыми обменами, но требует дополнительной памяти, и подходит для сортировки больших массивов данных.

Задание 3

Вариант №23

1. Требования к выполнению задачи:

   1. Разработать алгоритм ускоренной сортировки (задача 3), реализовать алгоритм. Сформировать таблицу результатов сортировки по формату Таблица 1 для массива, заполненного случайными числами. Определить емкостную сложность

алгоритма. *Определить асимптотическую сложность алгоритма.*

2. Провести дополнительные прогоны программы для оценки эффективности алгоритмов в наилучшем и наихудшем случаях.
3. Представить график зависимости Сф+Мф для анализируемых алгоритмов (все три алгоритма) для наихудшего случая.

2. Постановка задачи:

   Дано. Целочисленный массив и его длина.

   Результат. Отсортированный целочисленный массив.

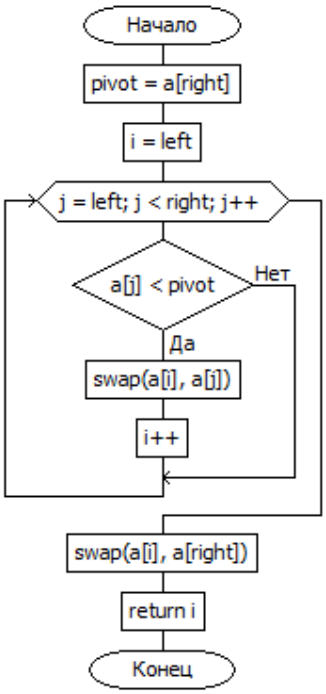
3. Модель решения:

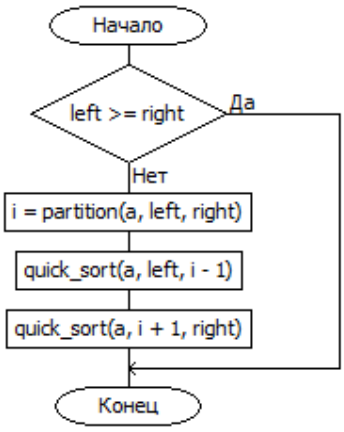
   Быстрая сортировка является существенно улучшенным вариантом алгоритма сортировки с помощью прямого обмена. Принципиальное отличие состоит в том, что в первую очередь производятся перестановки на наибольшем возможном расстоянии и после каждого прохода элементы делятся на две независимые группы.

   Общая идея алгоритма:

   - Выбрать из массива элемент, называемый опорным (в случае варианта 23 опорным элементом является последний).
   - Сравнить все остальные элементы с опорным и переставить их в массиве так, чтобы разбить массив на два непрерывных отрезка, следующих друг за другом: «элементы меньшие опорного» и «большие или равные».
   - Для отрезков «меньших» и «больших или равных» значений выполнить рекурсивно ту же последовательность операций, если длина отрезка больше единицы.

4. Блок-схемы функций быстрой сортировки:





5. **Сортировка массива методом быстрой сортировки:**
   Предусловие. a – целочисленный массив, left – левая граница сортировки, right – правая граница сортировки.
   Постусловие. a – отсортированный по неубыванию целочисленный массив.
   ```
   void quick_sort(int* a, int left, int right);
   ```
   Тест функции:

Номер теста	Исходные данные	Ожидаемый результат
1	a = {4, 3, 2}, len = 3	a = {2, 3, 4}
2	a = {1, 2, 3, 4}, len = 4	a = {1, 2, 3, 4}

```
void quick_sort(int* a, int left, int right)
{
	if (left >= right)
		return;

	int i = partition(a, left, right);

	quick_sort(a, left, i - 1);
	quick_sort(a, i + 1, right);
}
```

6. **Разделение массива на «меньшие» и «большие или равные» части и нахождение индекса опорного элемента:**

   Предусловие. a – целочисленный массив, left – левая граница, right – правая граница.
   Постусловие. i – индекс границы частей.

   ```
   int partition(int* a, int left, int right);
   ```

   Тест функции:

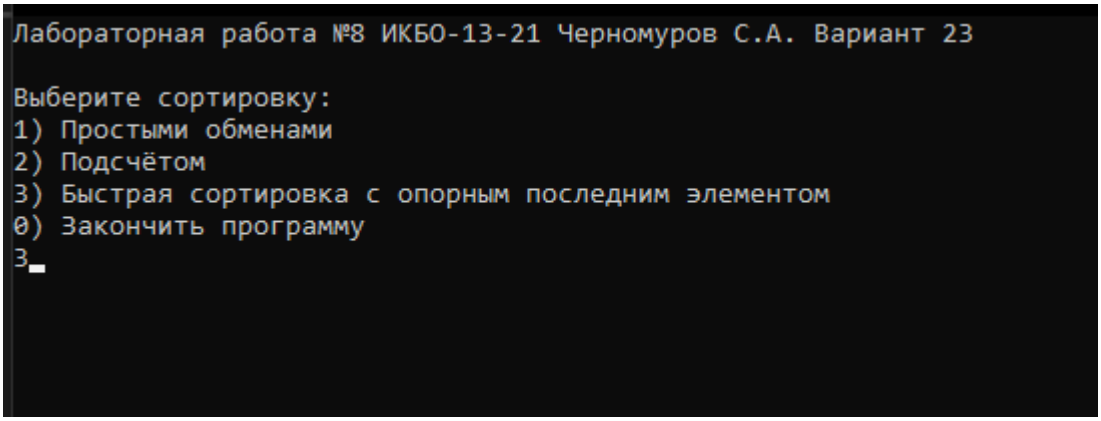
Номер теста	Исходные данные	Ожидаемый результат
1	a = {4, 3, 2}, left=0, right=2	i = 0
2	a = {1, 2, 3, 4}, left=0, right=3	I = 3

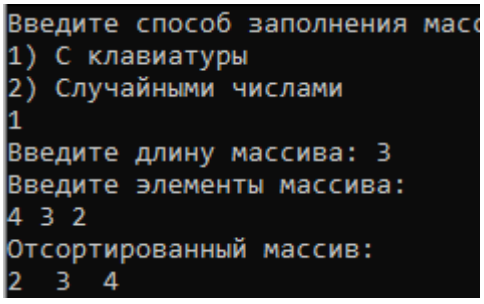
```
int partition(int* a, int left, int right) // перемещение всех элементов меньше опорного влево
{
	int pivot = a[right]; // опорный элемент - последний
	int i = left;

	for (int j = left; j < right; j++) {
		if (a[j] < pivot) {
			swap (a[i], a[j]);
			i++;
		}
```

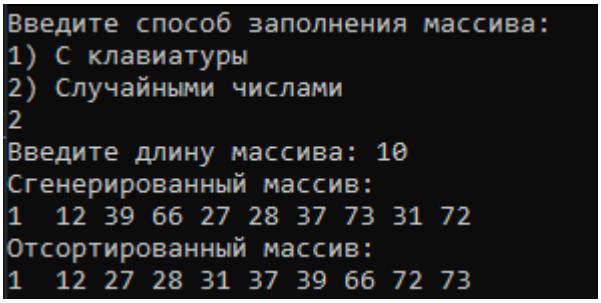
```
        }
        swap(a[i], a[right]);

        return i;
}
```

Асимптотическая сложность алгоритма быстрой сортировки O(n*log(n)). Затраты по памяти составляют O(n)

Тестирование программы







Время работы алгоритма для случайного массива

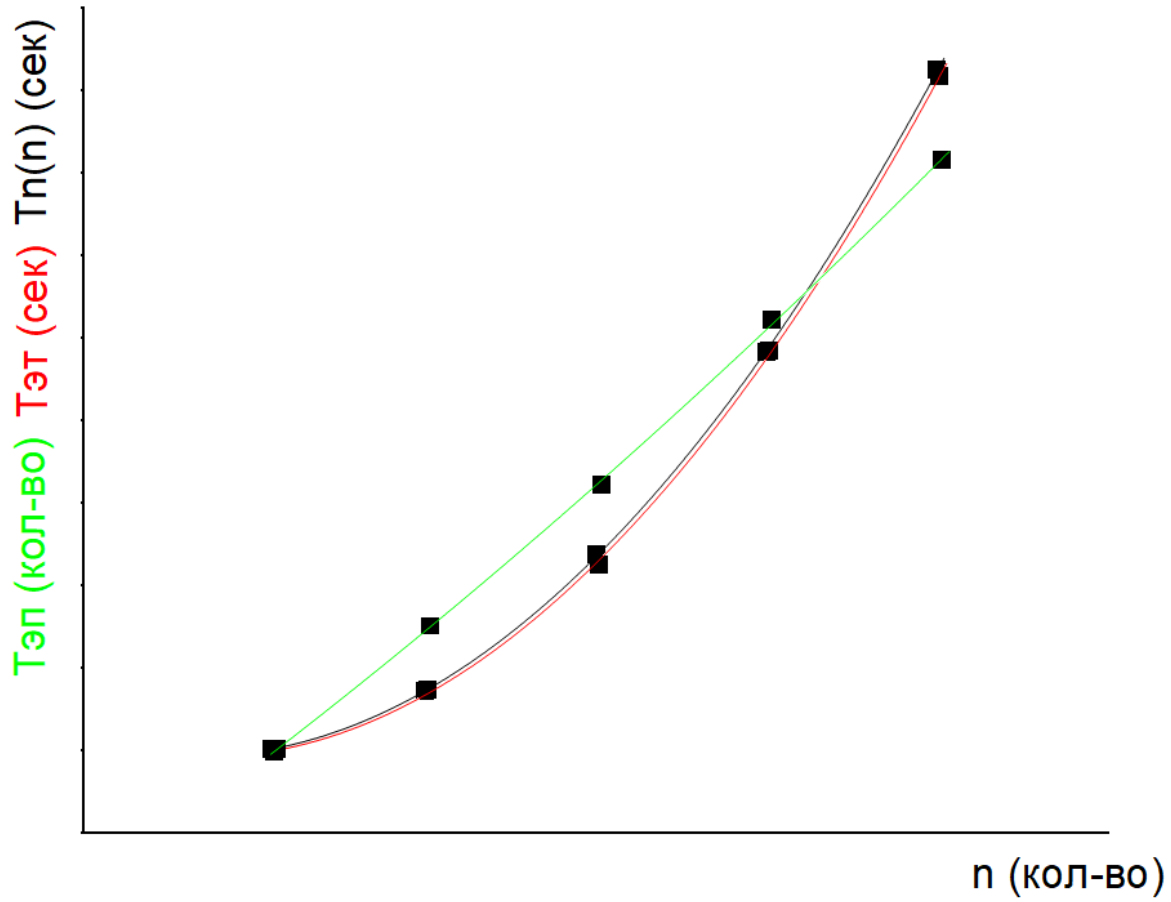
n	T(n) время в секундах	$T_{эт}$=f(C+M)-функция, секунд	$T_{эп}$=Сф+Мф-количество
10000	0.003	0.006	780000
100000	0.072	0.080	9700000
200000	0.239	0.232	20000000
300000	0.484	0.491	32000000
400000	0.826	0.824	43600000

Значения n увеличены, так как clock() не может улавливать настолько маленькие промежутки времени

$T_{эт}(n)=2*n*\log_{k/3}(n)/100\ 000\ 000$, где (100 000 000 – примерное количество операций, выполняемых в C++ за секунду, k – число, 3<k<=4)

$T_{эп}(n)= 2*n*\log_{k/3}(n)$, где (k – число, 3<k<=4)

График зависимости



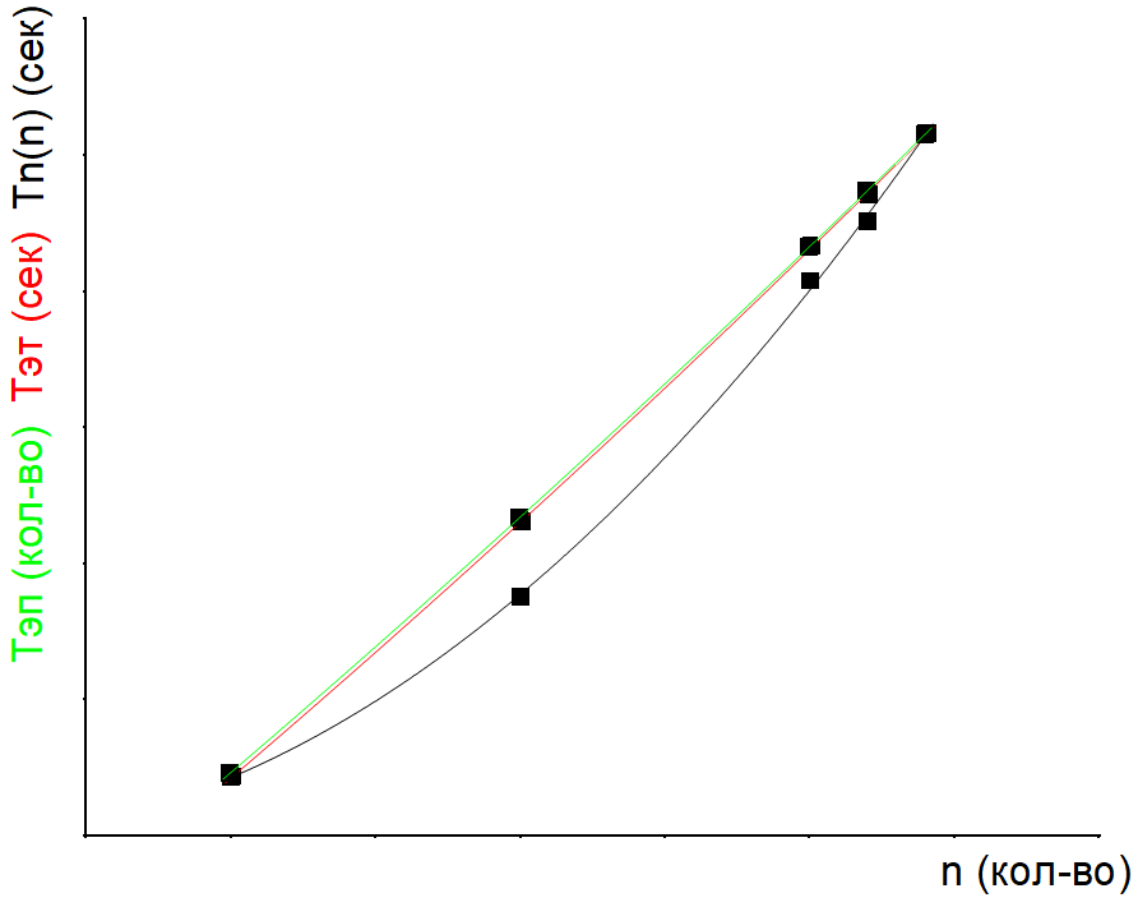
Время работы алгоритма для полностью отсортированного массива

n	T(n) время в секундах	$T_{эт}$=f(C+M)-функция, секунд	$T_{эп}$=Сф+Мф-количество
1000	0.011	0.020	2070000
2000	0.044	0.045	4570000
3000	0.102	0.072	7220000
3200	0.113	0.077	7770000
3400	0.129	0.083	8320000

$T_{эт}(n)=2*n*\log_{k/3}(n)/100\ 000\ 000$, где (100 000 000 – примерное количество операций, выполняемых в C++ за секунду, k – число, 3<k<=4)

$T_{эп}(n)= 2*n*\log_{k/3}(n)$, где (k – число, 3<k<=4)

График зависимости



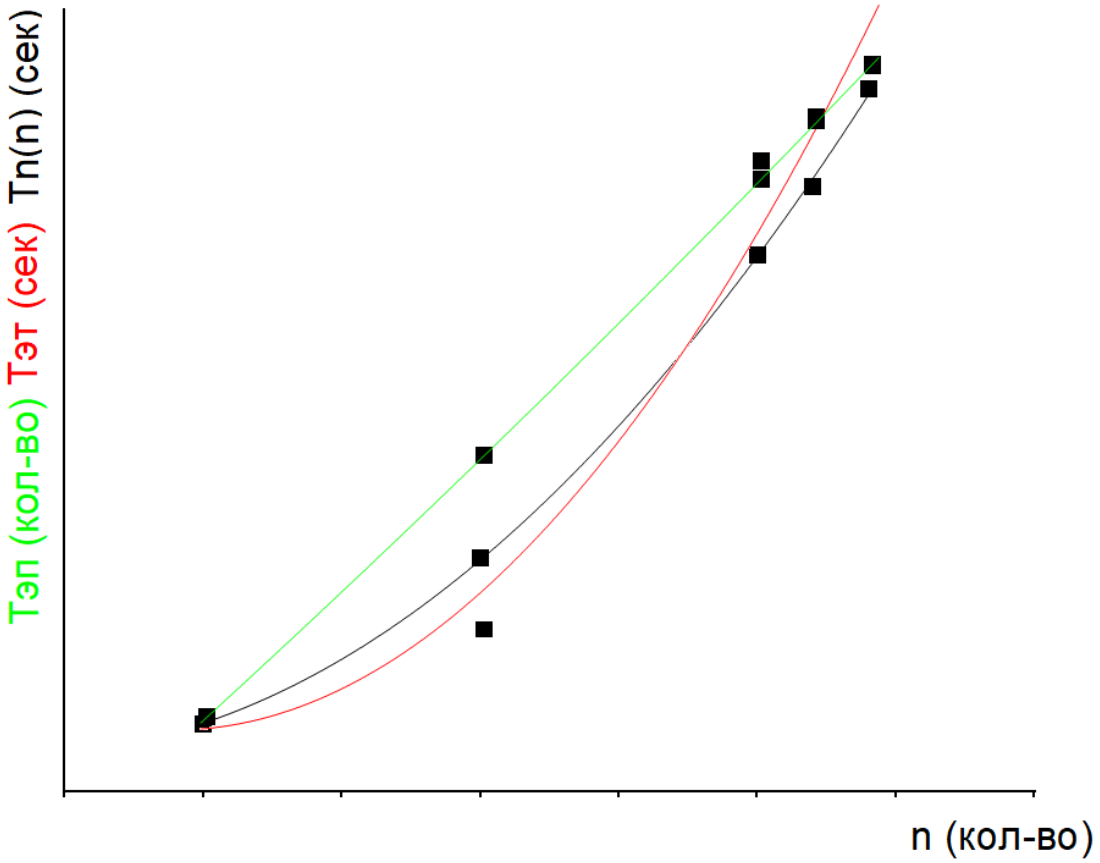
Время работы алгоритма для массива, отсортированного по убыванию

n	T(n) время в секундах	$T_{эт}$=f(C+M)-функция, секунд	$T_{эп}$=Сф+Мф-количество
1000	0.007	0.008	1380000
2000	0.024	0.018	3050000
3000	0.055	0.072	4820000
3200	0.062	0.077	5190000
3400	0.072	0.083	5550000

$T_{эт}(n)=2*n*\log_{k/3}(n)/100\ 000\ 000$, где (100 000 000 – примерное количество операций, выполняемых в C++ за секунду, k – число, 3<k<=4)

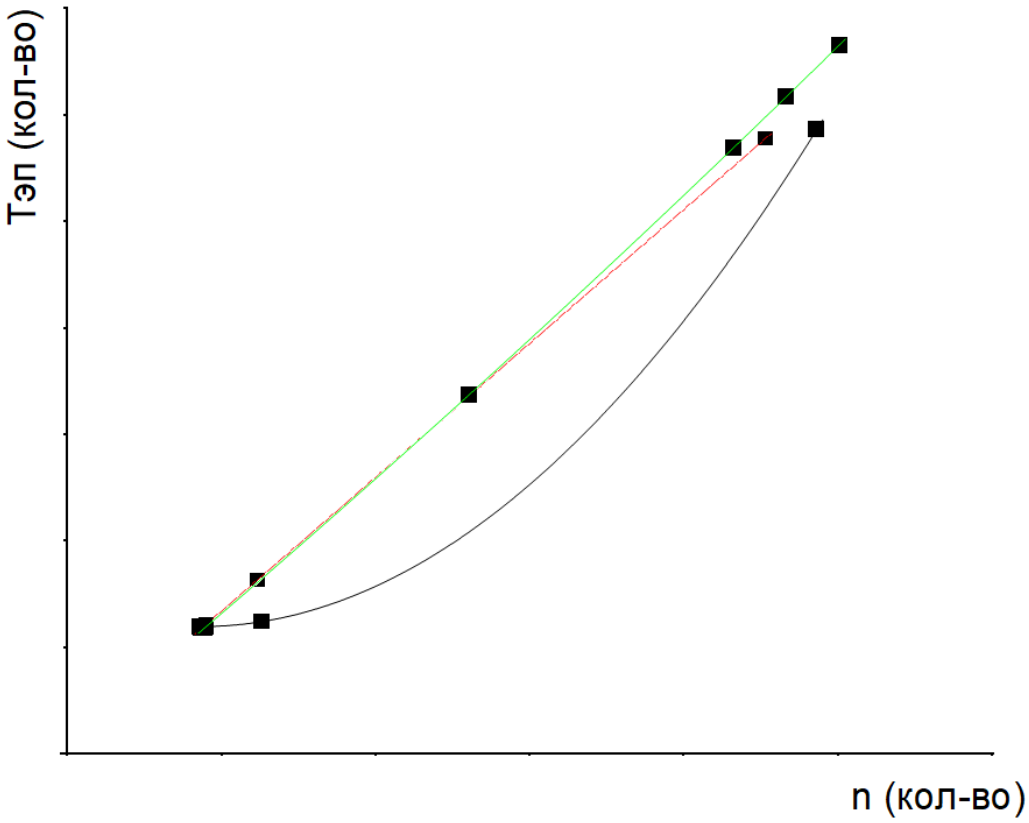
$T_{эп}(n)= 2*n*\log_{k/3}(n)$, где (k – число, 3<k<=4)

График зависимости



7. Выводы по эффективности работы алгоритма:
   Алгоритм быстрой сортировки является эффективным с точки зрения скорости работы, но сильно зависит от выбора опорного элемента. Также в неудачных случаях его асимптотическая сложность может составить $O(n^2)$, а стек может переполнится, что приведет к ошибкам выполнения программы

Графики зависимости Тэп(n) для всех трёх алгоритмов (черный – простые обмены, красный – сортировка подсчетом, зеленый – быстрая сортировка)



Полный код программы на языке C++

Файл main.cpp (основной алгоритм программы)

```
#include <iostream>
#include <string>
#include "functions.h"
#include "random"
#include <iomanip>
#include <ctime>
using namespace std;

int main() {
	setlocale(LC_ALL, "");
	srand(time(NULL));
	cout << "Лабораторная работа №8 ИКБО-13-21 Черномуров С.А. Вариант 23" << endl
<< endl;
```

```
	cout << "Выберите сортировку:\n1) Простыми обменами\n2) Подсчётом\n3) Быстрая сортировка с опорным последним элементом\n0) Закончить программу\n";

	int choice1;

	do {
		cin >> choice1;

		if (choice1 != 1 && choice1 != 2 && choice1 != 3 && choice1 != 0) cout << "Введено неверное значение, попробуйте снова.\n";
	} while (choice1 != 1 && choice1 != 2 && choice1 != 3 && choice1 != 0);

	switch (choice1) {
	case 0:
		return 0;
	}

	system("cls");

	cout << "Введите способ заполнения массива:\n1) С клавиатуры\n2) Случайными числами\n";

	int choice2;
	do {
		cin >> choice2;

		if (choice2 != 1 && choice2 != 2) cout << "Введено неверное значение, попробуйте снова.\n";
	} while (choice2 != 1 && choice2 != 2);

	int array_length;
	cout << "Введите длину массива: ";
	cin >> array_length;

	int* arr = (int*)malloc(sizeof(int) * array_length);
	//int* arr = new int[array_length];
	switch (choice2) {
	case 1:
		cout << "Введите элементы массива:\n";
		fill_arr(arr, array_length);
		break;
	case 2:
		fill_random(arr, array_length);
		cout << "Сгенерированный массив:\n";
	//print_array(arr, array_length);
		cout << endl;
		break;
	}

	switch (choice1) {
	case 1: {
		long double start_time = clock();
		bubble_sort(arr, array_length);
		long double end_time = clock();
		long double search_time = end_time - start_time;
		cout << "\n\n" << fixed << setprecision(14) << search_time << "\n\n\n\n\n";
		break;
	}
	case 2: {
		long double start_time = clock();
		count_sort(arr, array_length);
		long double end_time = clock();
		long double search_time = end_time - start_time;
```

```
                cout << "\n\n" << fixed << setprecision(14) << search_time <<
"\n\n\n\n\n\n";

                break;
        }
        case 3: {
                long double start_time = clock();
                quick_sort(arr, 0, array_length - 1);
                long double end_time = clock();
                long double search_time = end_time - start_time;
                cout << "\n\n" << fixed << setprecision(14) << search_time <<
"\n\n\n\n\n\n";
                break;
        }
        }

        cout << "Отсортированный массив:\n";
        //print_array(arr, array_length);
        cout << endl;


        main();
}
```

Файл functions.h (содержит прототипы функций)

```
#pragma once
#include <iostream>
#include <algorithm>
using namespace std;

void bubble_sort(int* a, int len);

void count_sort(int* a, int len);

int partition(int* a, int left, int right);
void quick_sort(int* a, int left, int right);

void fill_arr(int* a, int len);
void fill_random(int* a, int len);
void print_array(int* a, int len);
```

Файл functions.cpp (содержит тела функций)

```
#include "functions.h"
#include <iomanip>
#include <iostream>
#include <ctime>
#include <algorithm>
using namespace std;

void bubble_sort(int* a, int len) {
        for (int i = 0; i < len; i++)
        {
                if (i % 1000 == 0) cout << i << " ";
                for (int j = 0; j < len - 1; j++)
                        if (a[j] > a[j + 1]) swap(a[j], a[j + 1]);
        }
}

void count_sort(int* a, int len) {
        int maxi = a[0];
```

```
	for (int i = 1; i < len; i++)
		if (a[i] > maxi) maxi = a[i];

	int MAXIMUM = max(maxi, len);

	int* count = new int[MAXIMUM+1];
	int* output = new int[MAXIMUM+1];

	fill(count, count+MAXIMUM + 1, 0);

	for (int i = 0; i < len; i++)
		count[a[i]]++;

	for (int i = 1; i <= MAXIMUM; i++)
		count[i] += count[i - 1];

	for (int i = len - 1; i >= 0; i--) {
		output[count[a[i]] - 1] = a[i];
		count[a[i]]--;
	}

	for (int i = 0; i < len; i++) {
		a[i] = output[i];
	}
}


int partition(int* a, int left, int right) // перемещение всех элементов меньше
опорного влево
{
	int pivot = a[right]; // опорный элемент - последний
	int i = left;

	for (int j = left; j < right; j++) {
		if (a[j] < pivot) {
			swap (a[i], a[j]);
			i++;
		}
	}
	swap(a[i], a[right]);

	return i;
}

void quick_sort(int* a, int left, int right)
{
	if (left >= right)
		return;

	int i = partition(a, left, right);

	quick_sort(a, left, i - 1);
	quick_sort(a, i + 1, right);
}

void fill_arr(int* a, int len) {
	for (int i = 0; i < len; i++) {
		cin >> a[i];
	}
}
void fill_random(int* a, int len) {
	for (int i = 0; i < len; i++) {
		a[i] = rand()%100;
	}
```

```
}
void print_array(int* a, int len) {
	for (int i = 0; i < len; i++) {
		cout << left << setw(3) << a[i];
	}
}
```

Вывод

В ходе выполнения работы были получены навыки по анализу вычислительной сложности нескольких алгоритмов сортировки и определению наиболее эффективного алгоритма:

Название алгоритма	Асимптотическая сложность алгоритма			
	Наихудший случай	Наилучший случай	Средний случай	Емкостная сложность
Сортировка простыми обменами	$O(n^2)$	$O(n^2)$	$O(n^2)$	O(n)
Сортировка подсчетом	O(n+k)	O(n)	O(n+k)	O(n+k)
Быстрая сортировка с последним опорным элементом	$O(n^2)$	O(n*log(n))	O(n*log(n))	O(n)