

ФЕДЕРАЛЬНОЕ АГЕНТСТВО СВЯЗИ РОССИЙСКОЙ ФЕДЕРАЦИИ  
ФЕДЕРАЛЬНОЕ ГОСУДАРСТВЕННОЕ БЮДЖЕТНОЕ УЧРЕЖДЕНИЕ  
ВЫСШЕГО ОБРАЗОВАНИЯ  
«СИБИРСКИЙ ГОСУДАРСТВЕННЫЙ УНИВЕРСИТЕТ  
ТЕЛЕКОММУНИКАЦИЙ И ИНФОРМАТИКИ»

**ОТЧЁТ**

по лабораторной работе по дисциплине  
«Структуры и алгоритмы обработки данных»  
на тему  
АЛГОРИТМЫ СОРТИРОВКИ

Выполнил студент \_\_\_\_\_ Кулик Павел Евгеньевич  
Ф.И.О.

Группы \_\_\_\_\_ ИС-241

Работу принял \_\_\_\_\_ А. О. Насонова  
подпись

Защищена \_\_\_\_\_ Оценка \_\_\_\_\_

Новосибирск – 2023

## Оглавление

ВВЕДЕНИЕ .....	3
РАЗРАБОТКА.....	4
ИССЛЕДОВАНИЕ .....	5
ВЫВОД.....	7
ПРИЛОЖЕНИЕ .....	8

## **ВВЕДЕНИЕ**

В данной лабораторной работе требуется реализовать на языке программирования Си три предложенных алгоритма сортировки с целью дальнейшего исследования их эффективности.

Именно в этой лабораторной работе в соответствии с 10 вариантом исследуются такие алгоритмы, как Radix Sort, Selection Sort, Merge Sort.

## РАЗРАБОТКА

С целью исследования сложности алгоритмов было разработано тестовое приложение, содержащее в себе все функции, предназначенные для алгоритмов сортировки. Внутри функции `main` все названные выше функции используются последовательно 20 раз внутри цикла для сортировки массивов размером 50000 – 1000000 элементов с шагом в 50000 элементов. Время сортировки каждого из алгоритмов вычисляется с использованием предложенной функции `wtime()`.

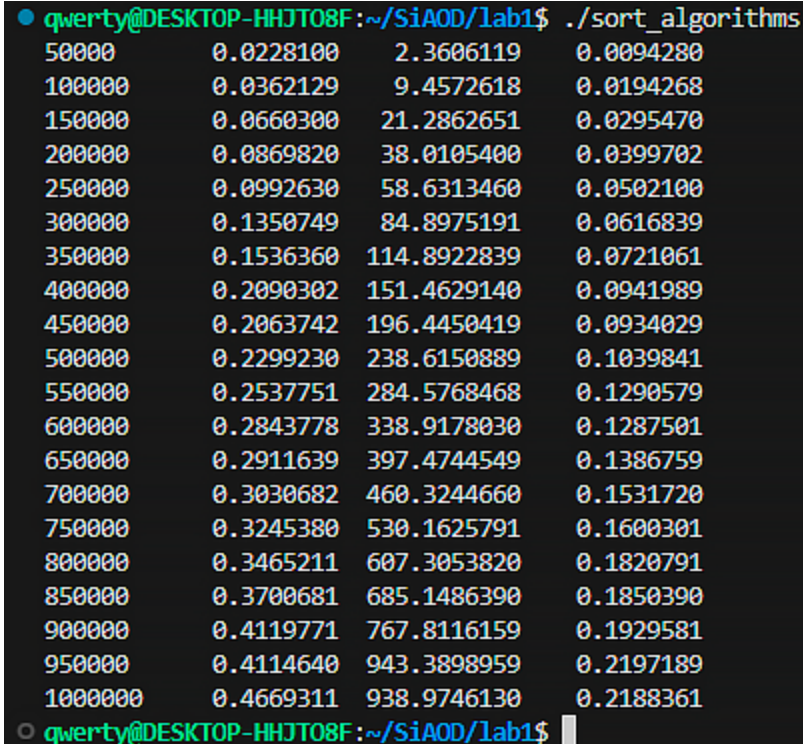
На каждом проходе цикла для сортируемых массивов выделяется память, которая освобождается после того, как все массивы отсортированы.

Время сортировки и размер сортируемых массивов выводится в стандартный поток вывода.

С исходным кодом программы можно ознакомиться в приложении.

Рисунок 1 – Пример вывода программ

(1)

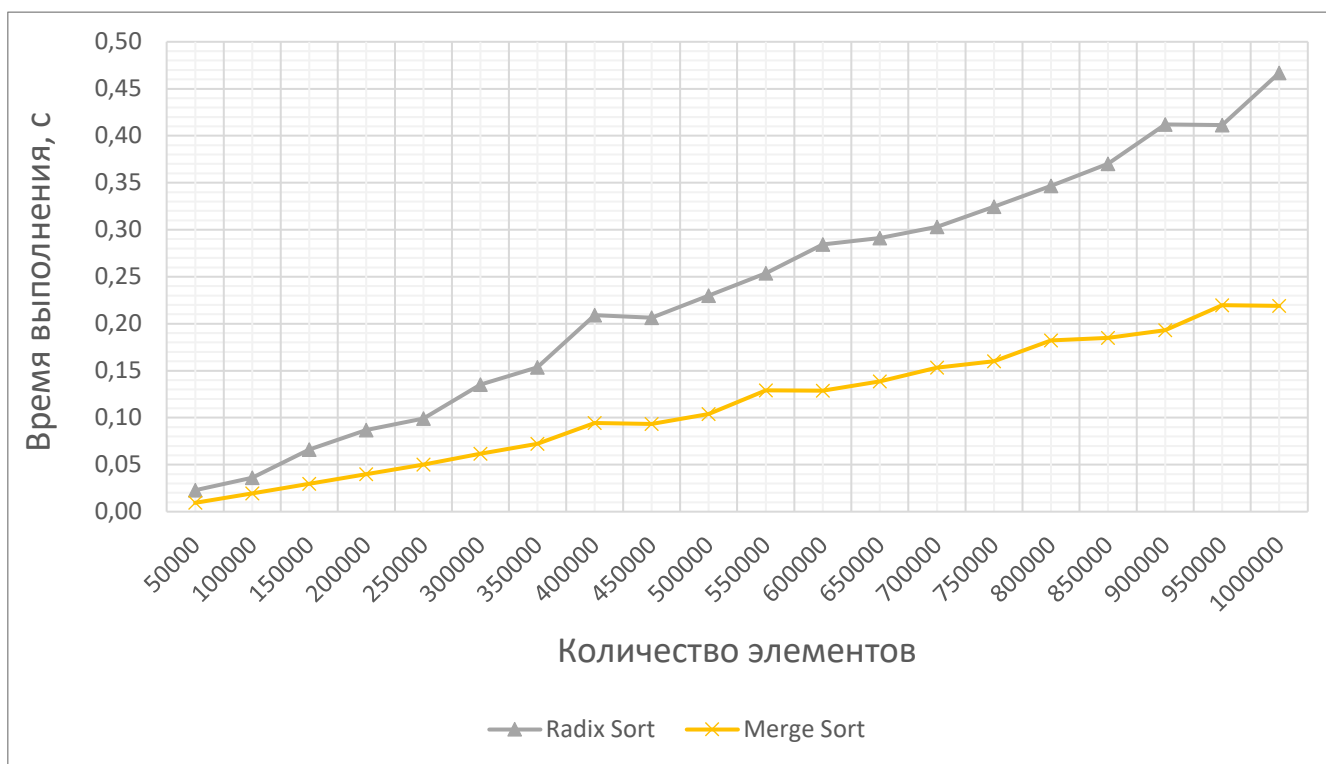


```
● qwerty@DESKTOP-HHJT08F:~/SiAOD/lab1$ ./sort_algorithms
50000      0.0228100    2.3606119    0.0094280
100000     0.0362129    9.4572618    0.0194268
150000     0.0660300    21.2862651    0.0295470
200000     0.0869820    38.0105400    0.0399702
250000     0.0992630    58.6313460    0.0502100
300000     0.1350749    84.8975191    0.0616839
350000     0.1536360    114.8922839   0.0721061
400000     0.2090302    151.4629140   0.0941989
450000     0.2063742    196.4450419   0.0934029
500000     0.2299230    238.6150889   0.1039841
550000     0.2537751    284.5768468   0.1290579
600000     0.2843778    338.9178030   0.1287501
650000     0.2911639    397.4744549   0.1386759
700000     0.3030682    460.3244660   0.1531720
750000     0.3245380    530.1625791   0.1600301
800000     0.3465211    607.3053820   0.1820791
850000     0.3700681    685.1486390   0.1850390
900000     0.4119771    767.8116159   0.1929581
950000     0.4114640    943.3898959   0.2197189
1000000    0.4669311    938.9746130   0.2188361
○ qwerty@DESKTOP-HHJT08F:~/SiAOD/lab1$
```

## ИССЛЕДОВАНИЕ

Полученные данные были перенесены в excel для построения графиков и дальнейшего анализа.

Рисунок 2 – Зависимость времени сортировки от количества элементов в массиве для Radix Sort и Merge Sort



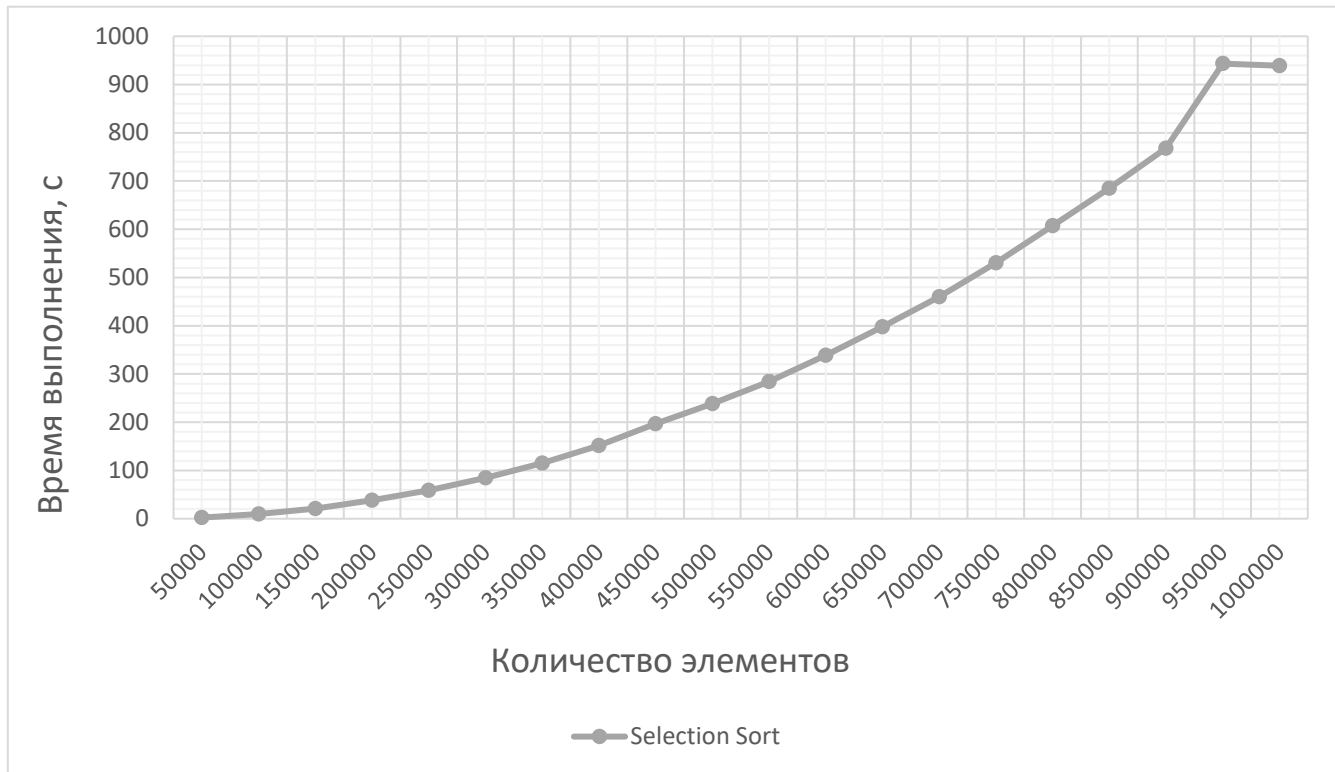
(2)

Оба этих алгоритма довольно хорошо справились с задачей сортировки большого количества элементов. Сложность поразрядной сортировки  $O(kn)$ , где  $n$  это количество элементов, а  $k$  – среднее количество цифр в числах (разрядов элементов). Сложность сортировки слиянием  $O(n \cdot \lg(n))$ , где  $n$  – количество элементов. Благодаря этому получились столь похожие графики.

Из-за того, что сложность поразрядной сортировки зависит от количества разрядов в значениях элементов, она больше подходит для массивов, хранящих небольшие значения.

А вот сортировку слиянием лучше применять к большим массивам, потому что в сложности количество элементов умножается на значение десятичного логарифма от значения элементов, который уменьшается в процентном соотношении относительно  $n$  при возрастании  $n$ .

Рисунок 3 – Зависимость времени сортировки от количества элементов в массиве для Selection Sort



(3)

Сложность алгоритма сортировки выбором  $O(n^2)$ . Из-за такой сложности сортировка является крайне медленной и подойдёт разве что для совсем небольших массивов.

## **ВЫВОД**

Существуют разные алгоритмы сортировки, каждый из которых обладает своими особенностями и подходит для разных задач.

## ПРИЛОЖЕНИЕ

### Приложение 1 “sort\_algorithms.c”

```
1  #include <inttypes.h>
2  #include <math.h>
3  #include <stdio.h>
4  #include <stdlib.h>
5  #include <sys/time.h>
6
7  #define LOW 50000
8  #define HIGH 1000000
9  #define STEP 50000
10
11 uint32_t findMax(uint32_t* arr, int size);
12 uint32_t number(uint32_t arrit, int k);
13 void radixSort(uint32_t* arr, int size);
14 void selectionSort(uint32_t* arr, int arrLength);
15 void merge(uint32_t* arr, int low, int mid, int high);
16 void mergeSort(uint32_t* arr, int low, int high);
17 int getrand(int min, int max);
18 double wtime();
19
20 int main()
21 {
22     uint32_t* testArr;
23     uint32_t* copy1;
24     uint32_t* copy2;
25     uint32_t* copy3;
26
27     int i, j;
28
29     double startTime, resultTime;
30
31     for (i = LOW; i <= HIGH; i += STEP) {
32         testArr = (uint32_t*)malloc(i * sizeof(uint32_t));
33         copy1 = (uint32_t*)malloc(i * sizeof(uint32_t));
34         copy2 = (uint32_t*)malloc(i * sizeof(uint32_t));
35         copy3 = (uint32_t*)malloc(i * sizeof(uint32_t));
36
37         for (j = 0; j < i; j++) {
38             testArr[j] = (uint32_t)getrand(0, 100001);
39             copy1[j] = testArr[j];
40             copy2[j] = testArr[j];
41             copy3[j] = testArr[j];
42         }
43
44         printf("%d\t", i);
45
46         startTime = wtime();
47         radixSort(copy1, i);
48         resultTime = wtime() - startTime;
49         printf("%13.7lf", resultTime);
```



```

50
51     startTime = wtime();
52     selectionSort(copy2, i);
53     resultTime = wtime() - startTime;
54     printf("%13.7lf", resultTime);
55
56     startTime = wtime();
57     mergeSort(copy3, 0, i - 1);
58     resultTime = wtime() - startTime;
59     printf("%13.7lf", resultTime);
60
61     printf("\n");
62
63     free(testArr);
64     free(copy1);
65     free(copy2);
66     free(copy3);
67 }
68 }
69
70 uint32_t findMax(uint32_t* arr, int size)
71 {
72     uint32_t max = 0;
73     for (int i = 0; i < size; i++) {
74         if (max < arr[i]) {
75             max = arr[i];
76         }
77     }
78     return max;
79 }
80
81 uint32_t number(uint32_t arrit, int k)
82 {
83     return (arrit / (uint32_t)pow(10, k - 1)) % 10;
84 }
85
86 void radixSort(uint32_t* arr, int size)
87 {
88     int i;
89     int k = 1;
90     uint32_t helpArr[size];
91     uint32_t max = findMax(arr, size);
92
93     while (max / (uint32_t)pow(10, k - 1) > 0) {
94         uint32_t digits[10] = {0};
95
96         for (i = 0; i < size; i++) {
97             digits[number(arr[i], k)]++;
98         }
99
100        for (i = 1; i < 10; i++) {
101            digits[i] += digits[i - 1];
102        }
103

```

```

104     for (i = size - 1; i >= 0; i--) {
105         helpArr[--digits[number(arr[i], k)]] = arr[i];
106     }
107
108     for (i = 0; i < size; i++) {
109         arr[i] = helpArr[i];
110     }
111
112     k++;
113 }
114 }
115
116 void selectionSort(uint32_t* arr, int arrLength)
117 {
118     int start, indexmin;
119     uint32_t min;
120     for (int i = 0; i < arrLength; i++) {
121         indexmin = i;
122         start = i;
123         for (int j = start; j < arrLength - 1; j++) {
124             if (arr[indexmin] > arr[j + 1]) {
125                 indexmin = j + 1;
126             }
127         }
128         min = arr[indexmin];
129         arr[indexmin] = arr[start];
130         arr[start] = min;
131     }
132 }
133
134 void merge(uint32_t* arr, int low, int mid, int high)
135 {
136     uint32_t* helpArr = (uint32_t*)malloc((high + 1) * sizeof(uint32_t));
137     for (int i = low; i <= high; i++) {
138         helpArr[i] = arr[i];
139     }
140     int l = low;
141     int r = mid + 1;
142     int i = low;
143
144     while ((l <= mid) && (r <= high)) {
145         if (helpArr[l] <= helpArr[r]) {
146             arr[i] = helpArr[l];
147             l++;
148         } else {
149             arr[i] = helpArr[r];
150             r++;
151         }
152         i++;
153     }
154
155     while (l <= mid) {
156         arr[i] = helpArr[l];
157         l++;

```

```

158         i++;
159     }
160
161     while (r <= high) {
162         arr[i] = helpArr[r];
163         r++;
164         i++;
165     }
166     free(helpArr);
167 }
168
169 void mergeSort(uint32_t* arr, int low, int high)
170 {
171     if (low < high) {
172         int mid = floor((low + high) / 2);
173         mergeSort(arr, low, mid);
174         mergeSort(arr, mid + 1, high);
175         merge(arr, low, mid, high);
176     }
177 }
178
179 int getrand(int min, int max)
180 {
181     return (double)rand() / (RAND_MAX + 1.0) * (max - min) + min;
182 }
183
184 double wtime()
185 {
186     struct timeval t;
187     gettimeofday(&t, NULL);
188     return (double)t.tv_sec + (double)t.tv_usec * 1E-6;
189 }

```