Бюджетное учреждение высшего образования   
Ханты-Мансийского автономного округа   
«Сургутский государственный университет»

Политехнический институт

Кафедра автоматики и компьютерных систем

**Отчет**

по лабораторной работе № 2

по дисциплине «Алгоритмы и структуры данных»

Выполнил: студент группы 609-21,

Шумилов И.Д.

Принял: старший преподаватель кафедры АиКС

Назаров Е.В.

Сургут

2024 г.

**Цель работы**: изучение алгоритмов сортировки, исследование сложности алгоритмов сортировки при различных исходных данных, закрепление навыков алгоритмизации.

**Общее задание на лабораторную работу**

1. Реализовать алгоритмы сортировки по варианту. Разработать функцию для проверки упорядоченности последовательности.

2. Разработать функции, позволяющие проводить оценку временной сложности T(n) и подсчет количества операций сравнения S(n), выполняемых в ходе работы функций сортировок.

3. На основе функций формирования последовательностей, реализованных в ходе лабораторной работы 1, и результатов выполнения пп. 1 и 2, разработать программу, с помощью которой по экспериментальным данным построить зависимости T(n) и S(n) для заданных алгоритмов сортировки по четырем последовательностям (возрастающей, случайной, убывающей и по варианту). Кроме того, необходимо оценить эти зависимости и у функции qsort из стандартной библиотеки C.

4. Составить отчет, в котором привести графики зависимостей T(n) и S(n), результаты анализа полученных экспериментальных данных и теоретических оценок сложности алгоритма, сравнительную оценку реализованных алгоритмов, выводы к работе.

**Индивидуальное задание на лабораторную работу (Вариант 17)**

1. Алгоритмы:
   1. Быстрая сортировка (сортировка Хоара)
   2. Пирамидальная S-арная сортировка (S = 2)
   3. Поразрядная LSD сортировка подсчетом
2. Тип: int
3. Последовательность: пилообразная

**Листинг кода программы**

Листинг 1 – Код программы

#include <iostream>

#include <stdlib.h>

#include <chrono>

#include <thread>

#include <stack>

#define ARR\_MIN 10

#define ARR\_MAX 10000

int operationCount = 0;

void swap(int \*arr, int a, int b) {

int tmp = arr[a];

arr[a] = arr[b];

arr[b] = tmp;

}

int randInt(int min, int max) {

double tmp = (double)rand() / RAND\_MAX;

return (int)(tmp \* (max - min + 1) + min);

}

void fillIncArray(int\* arr, int N, int min, int max) {

int step = (max - min) / N;

int tmp = min;

for (int i = 0; i < N; i++, tmp += step)

arr[i] = tmp;

}

void fillDecArray(int\* arr, int N, int min, int max) {

int step = (max - min) / N;

int tmp = max;

for (int i = 0; i < N; i++, tmp -= step)

arr[i] = tmp;

}

void fillRandArray(int\* arr, int N, int min, int max) {

for (int i = 0; i < N; i++)

arr[i] = randInt(min, max);

}

void fillSawArray(int\* arr, int N, int min, int max) {

int step = (max - min) / 10;

int tmp = min;

for (int i = 0; i < N; i++) {

arr[i] = tmp;

if (i != 0 && i % 10 == 0) {

tmp = min;

} else {

tmp += step;

}

}

}

enum Method { Inc = 1, Dec, Rand, Saw };

void fillArray(int\* arr, int N, int min, int max, Method m) {

switch (m) {

case Inc:

fillIncArray(arr, N, min, max);

break;

case Dec:

fillDecArray(arr, N, min, max);

break;

case Rand:

fillRandArray(arr, N, min, max);

break;

case Saw:

fillSawArray(arr, N, min, max);

break;

}

}

int partition(int \*arr, int low, int high) {

int pivot = arr[high];

int i = low - 1;

for (int j = low; j <= high; j++) {

operationCount++;

if (arr[j] < pivot) {

i++;

swap(arr, i, j);

}

}

swap(arr, i + 1, high);

return (i + 1);

}

void quickSort(int \*arr, int N) {

std::stack<std::pair<int, int>> s;

s.push({ 0, N - 1 });

while (!s.empty()) {

std::pair<int, int> p = s.top();

s.pop();

int low = p.first;

int high = p.second;

if (low < high) {

int pi = partition(arr, low, high);

s.push({ low, pi - 1 });

s.push({ pi + 1, high });

}

}

}

void heapify(int \*arr, int N, int i) {

int largest = i;

int l = 2 \* i + 1;

int r = 2 \* i + 2;

operationCount++;

if (l < N && arr[l] > arr[largest])

largest = l;

operationCount++;

if (r < N && arr[r] > arr[largest])

largest = r;

if (largest != i) {

swap(arr, largest, i);

heapify(arr, N, largest);

}

}

void heapSort(int \*arr, int N) {

for (int i = N / 2 - 1; i >= 0; i--)

heapify(arr, N, i);

for (int i = N - 1; i > 0; i--) {

swap(arr, 0, i);

heapify(arr, i, 0);

}

}

int getMax(int \*arr, int N) {

int tmp = arr[0];

for (int i = 1; i < N; i++) {

if (tmp < arr[i])

tmp = arr[i];

}

return tmp;

}

void countSort(int \*arr, int N, int exp) {

int \*output = new int[N];

int count[10] = { 0 };

for (int i = 0; i < N; i++)

count[(arr[i] / exp) % 10]++;

for (int i = 1; i < 10; i++)

count[i] += count[i - 1];

for (int i = N - 1; i >= 0; i--) {

output[count[(arr[i] / exp) % 10] - 1] = arr[i];

count[(arr[i] / exp) % 10]--;

}

for (int i = 0; i < N; i++)

arr[i] = output[i];

}

void radixSort(int \*arr, int N) {

int max = getMax(arr, N);

for (int exp = 1; max / exp > 0; exp \*= 10)

countSort(arr, N, exp);

}

bool isSorted(int\* arr, int N) {

for (int i = 0; i < N - 1; i++) {

if (arr[i] > arr[i + 1])

return false;

}

return true;

}

void printBigO(void func(int\*, int), int N) {

int min = ARR\_MIN, max = ARR\_MAX;

int \*arr = new int[N];

std::string method[4] = {"Increasing", "Decreasing", "Randomeowed", "Sawtoothed"};

for (int i = 0; i < 4; i++) {

fillArray(arr, N, min, max, (Method)i);

auto start = std::chrono::high\_resolution\_clock::now();

operationCount = 0;

func(arr, N);

auto end = std::chrono::high\_resolution\_clock::now();

std::chrono::duration<double, std::milli> time = end - start;

std::cout << method[i] << ". " << time.count() << "\tComparisons: " << operationCount << std::endl;

}

}

void printArray(int \*arr, int size) {

for (int i = 0; i < size; i++)

std::cout << arr[i] << ", ";

std::cout << "\b\b.\n" << std::endl;

}

int main(int argc, char\*\* argv) {

srand(time(NULL));

for (int i = 500; i < 5001; i += 500) {

std::cout << "\n\tSIZE: " << i << std::endl;

std::cout << "Quicksort: " << std::endl;

printBigO(quickSort, i);

std::cout << "\nHeapsort: " << std::endl;

printBigO(heapSort, i);

std::cout << "\nRadix sort: " << std::endl;

printBigO(radixSort, i);

}

return 0;

}

**Таблицы и графики с результатами измерений длительности формирования последовательностей**

Таблица 1 - Время сортировки последовательностей быстрой сортировкой

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| Размер массива (n) | Время сортировки типа последовательности T (мс) | | | | |
| Возрастающая | Убывающая | Случайная | Пилообразная | Среднее |
| 500 | 1,2351 | 1,0739 | 1,1802 | 0,4522 | 0.9854 |
| 1000 | 4,7931 | 14,4663 | 5,0759 | 1,1564 | 6.3729 |
| 1500 | 10,7877 | 17,0971 | 8,5148 | 4,1884 | 10.1470 |
| 2000 | 12,0927 | 22,8486 | 18,9275 | 1,5300 | 13.8497 |
| 2500 | 18,9087 | 29,2336 | 20,3206 | 5,3364 | 18.4498 |
| 3000 | 15,2759 | 37,7160 | 22,6904 | 2,8894 | 19.6429 |
| 3500 | 18,3664 | 44,8939 | 34,7280 | 3,6708 | 25.4148 |
| 4000 | 26,1377 | 71,9915 | 43,5706 | 2,5230 | 36.0557 |
| 4500 | 27,2374 | 96,9774 | 56,5804 | 3,4616 | 46.0642 |
| 5000 | 50,8101 | 119,3600 | 61,2682 | 4,0078 | 58.8615 |

Таблица 2 - Количество операций при сортировке последовательностей быстрой сортировкой

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| Размер массива (n) | Количество операций сравнения S | | | | |
| Возрастающая | Убывающая | Случайная | Пилообразная | Среднее |
| 500 | 125249 | 125249 | 125249 | 4653 | 95100 |
| 1000 | 500499 | 500499 | 500499 | 11008 | 378126.3 |
| 1500 | 1125749 | 1125749 | 1125749 | 17870 | 848779.3 |
| 2000 | 2000999 | 2000999 | 2000999 | 24917 | 1506979 |
| 2500 | 3126249 | 3126249 | 3126249 | 33287 | 2353009 |
| 3000 | 4501499 | 4501499 | 4501499 | 43002 | 3386875 |
| 3500 | 6126749 | 6126749 | 6126749 | 48572 | 4607205 |
| 4000 | 8001999 | 8001999 | 8001999 | 56280 | 6015569 |
| 4500 | 10127249 | 10127249 | 10127249 | 64261 | 7611502 |
| 5000 | 12502499 | 12502499 | 12502499 | 73011 | 9395127 |

Рис. 1 – График зависимости T(n) при быстрой сортировке

Рис. 2 – График зависимости S(n) при быстрой сортировке

На возрастающих и убывающих последовательностях: у быстрой сортировки время сортировки возрастает с увеличением размера массива, но намного медленнее, чем на случайных или пилообразных последовательностях. На случайных последовательностях: время сортировки также увеличивается с увеличением размера массива, но наибольший прирост происходит на больших размерах массива. На пилообразных последовательностях: время сортировки начинает расти значительно с меньшего размера массива.

Как и ожидалось, на пилообразных последовательностях количество операций сравнения существенно меньше по сравнению с другими типами последовательностей.

Таблица 3 - Время сортировки последовательностей пирамидальной двоичной сортировкой

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| Размер массива | Время сортировки типа последовательности T (мс) | | | | |
| Возрастающая | Убывающая | Случайная | Пилообразная | Среднее |
| 500 | 0,0064 | 0,0864 | 0,1123 | 0,0702 | 0.0688 |
| 1000 | 0,0122 | 0.1464 | 0,1627 | 0.1678 | 0.1223 |
| 1500 | 0,0334 | 0,2192 | 0,2137 | 0,2753 | 0.1854 |
| 2000 | 0,0233 | 0,4241 | 0,2793 | 0,4805 | 0.3018 |
| 2500 | 0,0321 | 0,4560 | 0,5657 | 0,4614 | 0.3788 |
| 3000 | 0,0801 | 0,6227 | 0,5758 | 0,5693 | 0.4620 |
| 3500 | 0,0461 | 0,6760 | 0,6671 | 0,6696 | 0.5147 |
| 4000 | 0,0545 | 0,6840 | 0,6865 | 0,8060 | 0.5578 |
| 4500 | 0,0536 | 0.8286 | 0.9001 | 1,0123 | 0.6987 |
| 5000 | 0,0654 | 0,9773 | 0,9286 | 1,0085 | 0.7450 |

Таблица 4 - Количество операций при сортировке последовательностей пирамидальной двоичной сортировкой

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| Размер массива (n) | Количество операций сравнения S | | | | |
| Возрастающая | Убывающая | Случайная | Пилообразная | Среднее |
| 500 | 1498 | 9208 | 7852 | 8542 | 6775 |
| 1000 | 2998 | 20416 | 17632 | 19136 | 15045.5 |
| 1500 | 4498 | 32596 | 28544 | 30620 | 24064.5 |
| 2000 | 5998 | 44600 | 39416 | 42346 | 33090 |
| 2500 | 7498 | 58128 | 50960 | 54526 | 42778 |
| 3000 | 8998 | 71504 | 62500 | 67174 | 52544 |
| 3500 | 10498 | 84896 | 74800 | 79980 | 62543.5 |
| 4000 | 11998 | 98264 | 86872 | 92670 | 72451 |
| 4500 | 13498 | 112464 | 99116 | 105918 | 82749 |
| 5000 | 14998 | 126864 | 111872 | 119256 | 93247.5 |

Рис. 3 – График зависимости T(n) при пирамидальной двоичной сортировке

Рис. 4 – График зависимости S(n) при пирамидальной двоичной сортировке

Время сортировки пирамидальной сортировки выглядит более стабильным по сравнению с быстрой сортировкой. Однако, на убывающих последовательностях время значительно увеличивается с увеличением размера массива.

Пирамидальная сортировка имеет сравнительно большое количество операций сравнения на любом типе последовательности, но это количество растет сравнительно медленно по сравнению с быстрой сортировкой.

Таблица 5 – Время сортировки последовательностей поразрядной LSD сортировкой подсчетом

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| Размер массива | Время сортировки типа последовательности T (мс) | | | | |
| Возрастающая | Убывающая | Случайная | Пилообразная | Среднее |
| 500 | 0,0012 | 0,0559 | 0,0636 | 0,0464 | 0.9854 |
| 1000 | 0,0021 | 0,1103 | 0,1617 | 0,2276 | 6.3729 |
| 1500 | 0,0029 | 0,8679 | 0,7483 | 0,1685 | 10.1470 |
| 2000 | 0,0040 | 1,4212 | 0,4492 | 0,4805 | 13.8497 |
| 2500 | 0,0048 | 0,2305 | 0,2755 | 0,2611 | 18.4498 |
| 3000 | 0,0056 | 0,6227 | 0,5758 | 0,5693 | 19.6429 |
| 3500 | 0,0065 | 0,3205 | 0,6496 | 0,3469 | 25.4148 |
| 4000 | 0,0074 | 0,4933 | 0,5769 | 0,4139 | 36.0557 |
| 4500 | 0,0085 | 0,5315 | 0,6028 | 0,7174 | 46.0642 |
| 5000 | 0,0174 | 0,7533 | 0,7439 | 0,0437 | 58.8615 |

Таблица 6 - Количество операций при сортировке последовательностей поразрядной LSD сортировкой подсчетом

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| Размер массива (n) | Количество операций сравнения S | | | | |
| Возрастающая | Убывающая | Случайная | Пилообразная | Среднее |
| 500 | 499 | 499 | 499 | 499 | 499 |
| 1000 | 999 | 999 | 999 | 999 | 999 |
| 1500 | 1499 | 1499 | 1499 | 1499 | 1499 |
| 2000 | 1999 | 1999 | 1999 | 1999 | 1999 |
| 2500 | 2499 | 2499 | 2499 | 2499 | 2499 |
| 3000 | 2999 | 2999 | 2999 | 2999 | 2999 |
| 3500 | 3499 | 3499 | 3499 | 3499 | 3499 |
| 4000 | 3999 | 3999 | 3999 | 3999 | 3999 |
| 4500 | 4499 | 4499 | 4499 | 4499 | 4499 |
| 5000 | 4999 | 4999 | 4999 | 4999 | 4999 |

Рис. 5 – График зависимости T(n) при поразрядной LSD сортировке

Рис. 6 – График зависимости S(n) при поразрядной LSD сортировке

Время сортировки этого метода в целом намного меньше по сравнению с двумя предыдущими алгоритмами на всех типах последовательностей.

Операции сравнения в поразрядной сортировке постоянны и не зависят от типа последовательности.

Код программы был модифицирован для того чтобы измерить время выполнения функции qsort(). Следующие функции были изменены:

Листинг 2 – измененные функции

int cmp(const void\* a, const void\* b) { opearionCount ++; return (\*(int\*)a) - (\*(int\*)b); }

void printBigO(int N) {

int min = ARR\_MIN, max = ARR\_MAX;

int \*arr = new int[N];

std::string method[4] = {"Increasing", "Decreasing", "Random", "Sawtoothed"};

for (int i = 0; i < 4; i++) {

fillArray(arr, N, min, max, (Method)i);

auto start = std::chrono::high\_resolution\_clock::now();

operationCount++;

std::qsort((void\*)arr, N, sizeof(int), cmp);

auto end = std::chrono::high\_resolution\_clock::now();

std::chrono::duration<double, std::milli> time = end - start;

std::cout << method[i] << ". " << time.count() << “\tOperations: ” << operationCount << std::endl;

if (!isSorted(arr, N)) {

std::cout << "Unsorted..." << std::endl;

return;

}

}

}

Таблица 7 – Время сортировки последовательностей функцией qsort()

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| Размер массива | Время сортировки типа последовательности T (мс) | | | | |
| Возрастающая | Убывающая | Случайная | Пилообразная | Среднее |
| 500 | 0,0076 | 0,0276 | 0,0311 | 0,0619 | 0.0321 |
| 1000 | 0,0094 | 0,0576 | 0,0638 | 0,1348 | 0.1960 |
| 1500 | 0,0149 | 0,0902 | 0,0984 | 0,2098 | 0.1033 |
| 2000 | 0,0191 | 0,1234 | 0,1355 | 0,2963 | 0.1436 |
| 2500 | 0,0227 | 0,2122 | 0,1762 | 0,4748 | 0.2215 |
| 3000 | 0,0271 | 0,1917 | 0,2088 | 0,4549 | 0.2206 |
| 3500 | 0,0318 | 0,2331 | 0,2524 | 0,5276 | 0.2612 |
| 4000 | 0,0360 | 0,2762 | 0,2849 | 0,6161 | 0.3033 |
| 4500 | 0,0405 | 0,3137 | 0,3288 | 0,7091 | 0.3480 |
| 5000 | 0,0450 | 0,3378 | 0,3807 | 0,8056 | 0.3923 |

Таблица 8 - Количество операций при сортировке функцией qsort()

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| Размер массива (n) | Количество операций сравнения S | | | | |
| Возрастающая | Убывающая | Случайная | Пилообразная | Среднее |
| 500 | 999 | 4347 | 4347 | 5069 | 3690.5 |
| 1000 | 1999 | 9684 | 9684 | 11522 | 8222.25 |
| 1500 | 2999 | 14683 | 14683 | 18472 | 12709.25 |
| 2000 | 3999 | 21357 | 21357 | 26163 | 18219 |
| 2500 | 4999 | 26446 | 26446 | 32066 | 22489.25 |
| 3000 | 5999 | 32356 | 32356 | 38824 | 27383.75 |
| 3500 | 6999 | 39285 | 39285 | 47181 | 33187.5 |
| 4000 | 7999 | 46702 | 46702 | 55475 | 39219.5 |
| 4500 | 8999 | 54607 | 54607 | 61712 | 44981.25 |
| 5000 | 9999 | 57891 | 57891 | 71038 | 49204.75 |

Рис. 7 – График зависимости T(n) при сортировке qsort

Рис. 8 – График зависимости S(n) при сортировке qsort

Функция qsort() демонстрирует хорошую производительность на всех типах последовательностей, причем время сортировки увеличивается пропорционально размеру массива. На убывающих и возрастающих последовательностях время сортировки значительно меньше, чем у быстрой сортировки и пирамидальной сортировки.

Количество операций сравнения увеличивается пропорционально размеру массива и типу последовательности. На пилообразных последовательностях операции сравнения занимают меньше времени, чем на убывающих или возрастающих последовательностях.

**Сравнительная оценка реализованных алгоритмов**

По времени сортировки:

1. Быстрая сортировка: Хорошо справляется с разнообразными последовательностями, но на пилообразных последовательностях может быть менее эффективной.
2. Пирамидальная двоичная сортировка: Обеспечивает стабильную производительность, но может иметь высокое время сортировки на убывающих последовательностях.
3. Поразрядная LSD сортировка подсчетом: Показывает хорошую производительность и универсальность на всех типах последовательностей, при этом имея меньшие требования к памяти.
4. Функция qsort(): Демонстрирует хорошую производительность на всех типах последовательностей, превосходя другие алгоритмы по времени сортировки.

По устойчивости:

1. Быстрая сортировка: Устойчивый алгоритм.
2. Пирамидальная двоичная сортировка: Устойчивый алгоритм.
3. Поразрядная LSD сортировка подсчетом: Устойчивый алгоритм.
4. Функция qsort(): Неустойчивый алгоритм.

По естественности:

1. Быстрая сортировка: Естественный и интуитивно понятный алгоритм.
2. Пирамидальная двоичная сортировка: Требует более подробного понимания структуры данных, но в целом естественный.
3. Поразрядная LSD сортировка подсчетом: Может быть менее естественной из-за своего специфического метода сортировки по разрядам.
4. Функция qsort(): Естественная и стандартная функция сортировки, легко использовать и понимать.

По требованиям к памяти:

1. Быстрая сортировка: Имеет небольшие требования к памяти.
2. Пирамидальная двоичная сортировка: Имеет средние требования к памяти.
3. Поразрядная LSD сортировка подсчетом: Обычно имеет минимальные требования к памяти.
4. Функция qsort(): Требует дополнительной памяти для хранения индексов массива.

Функция qsort() представляет собой стандартный и эффективный метод сортировки, который часто является предпочтительным в реальных приложениях благодаря своей высокой производительности. Быстрая сортировка, пирамидальная сортировка и поразрядная LSD сортировка также представляют собой важные инструменты с различными преимуществами и недостатками в зависимости от типа данных и условий использования. При выборе алгоритма сортировки важно учитывать требования к производительности, устойчивости, естественности и требованиям к памяти, чтобы выбрать наиболее подходящий алгоритм для конкретной задачи.

**Пример выполнения программы**

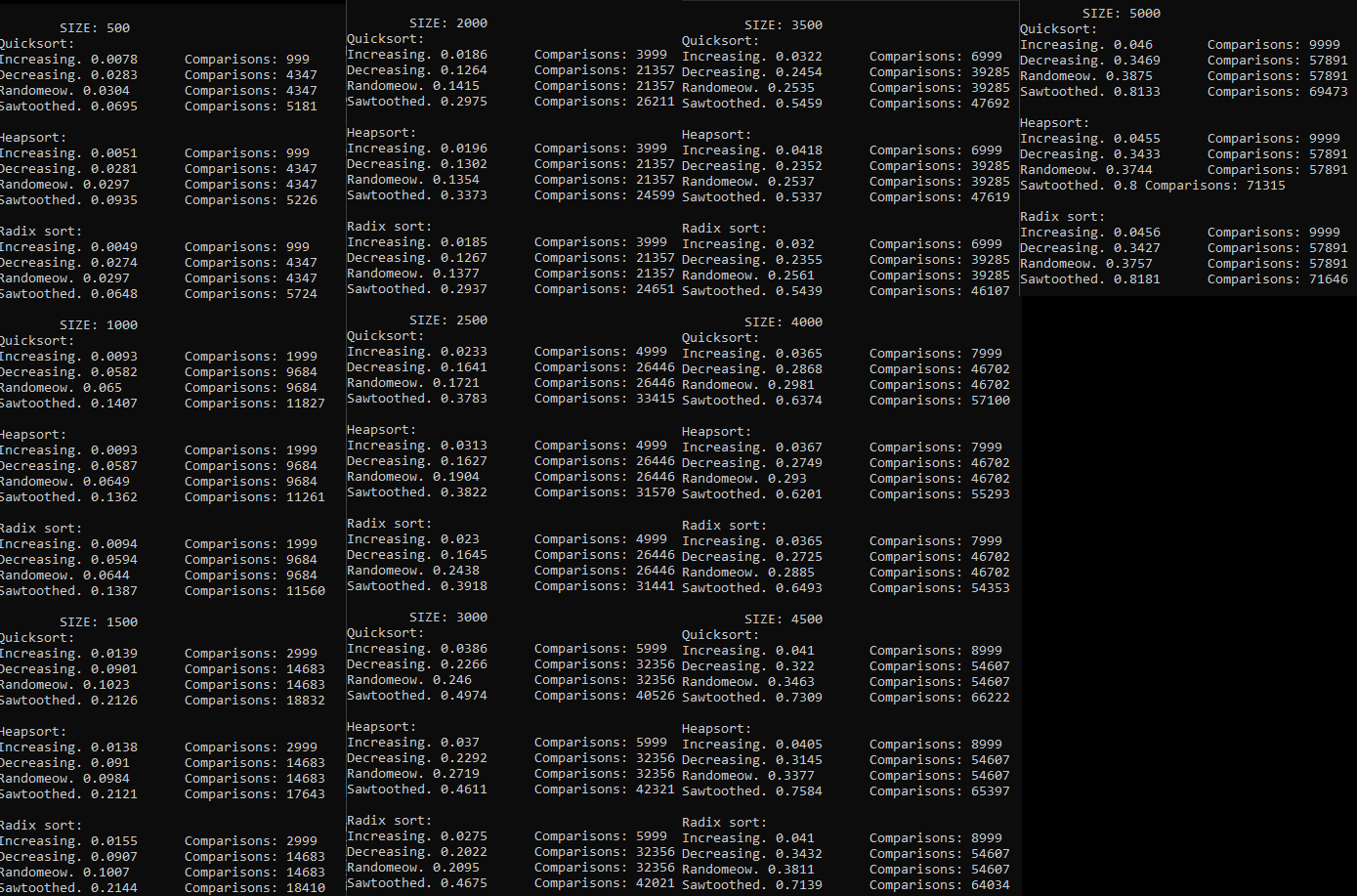


Рис. 9 – Пример выполнения программы

**Выводы**

Были изучены алгоритмы сортировки, исследованы сложности алгоритмов сортировки при различных исходных данных, закреплены навыки алгоритмизации.