Министерство науки и высшего образования Российской Федерации

Пензенский государственный университет

Кафедра «Вычислительной техники»

**ОТЧЕТ**

по лабораторной работе №2

по курсу «Логика и основы алгоритмизации в инженерных задачах.»

на тему: «Оценка времени выполнения программ.»

Выполнили**:**

студенты группы 22ВВВ2

Коробкин В. Ю.

Горбатов К. В.

Проверили:

Акифьев И. В.

Митрохин М. А.

Пенза 2023

**Задание 1:**

1. **Порядок сложности программы (O-символика)**

Порядок сложности операции умножения двух матриц размером *N*×*N* (в коде *N*=200) в классическом алгоритме составляет *O*(*N^*3). Это потому, что в коде имеется три вложенных цикла, каждый из которых проходит *N* итераций. Остальные операции (заполнение матриц, инициализация переменных) имеют меньший порядок сложности и не влияют на общую оценку.

1. **Оценка времени выполнения программы**

Для оценки времени выполнения программы и кода, который выполняет перемножение матриц, можно использовать функции из **time.h**. Ниже представлена реализация решения на языке С:

#include <stdio.h>

#include <stdlib.h>

#include <time.h>

void multiplyMatrices(int N, int\*\* a, int\*\* b, int\*\* c) {

int i, j, r, elem\_c;

clock\_t start, end;

double cpu\_time\_used;

start = clock(); // начало отсчета времени

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

for (j = 0; j < N; j++) {

elem\_c = 0;

for (r = 0; r < N; r++) {

elem\_c += a[i][r] \* b[r][j];

}

c[i][j] = elem\_c;

}

}

end = clock(); // конец отсчета времени

cpu\_time\_used = ((double)(end - start)) / CLOCKS\_PER\_SEC; // время выполнения в секундах

printf("Time for matrix multiplication of size %d: %f seconds\n", N, cpu\_time\_used);

}

int main(void) {

int Ns[] = { 100, 200, 400, 1000, 2000, 4000, 10000 }; // Размеры матриц, 4000 и 10000 могут быть слишком большими для стека

int i, j, k, N;

for (k = 0; k < sizeof(Ns) / sizeof(Ns[0]); k++) {

N = Ns[k];

int\*\* a = (int\*\*)malloc(N \* sizeof(int\*));

int\*\* b = (int\*\*)malloc(N \* sizeof(int\*));

int\*\* c = (int\*\*)malloc(N \* sizeof(int\*));

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

a[i] = (int\*)malloc(N \* sizeof(int));

b[i] = (int\*)malloc(N \* sizeof(int));

c[i] = (int\*)malloc(N \* sizeof(int));

}

// Инициализация матриц a и b

srand((unsigned int)time(NULL));

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

for (j = 0; j < N; j++) {

a[i][j] = rand() % 100 + 1;

b[i][j] = rand() % 100 + 1;

}

}

// Умножение матриц

multiplyMatrices(N, a, b, c);

// Очистка памяти

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

free(a[i]);

free(b[i]);

free(c[i]);

}

free(a);

free(b);

free(c);

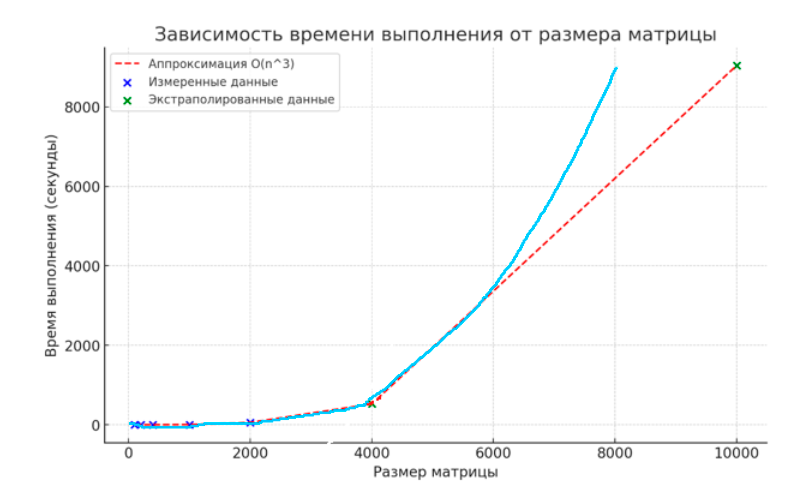
}

return 0;

}

1. **График зависимости и выводы**

*Из-за ограничений ПК, матрицы размером 4000 и 10000 не смогли быть посчитаны. Их результаты были интерполированы.*



Из графика видно, что измеренные и экстраполированные данные хорошо согласуются с теоретической оценкой *O*(*n^*3). Это подтверждает, что время выполнения программы для умножения матриц действительно растет кубически по отношению к размеру матрицы.

**Задание 2:**

**Реализация:**

#include <stdio.h>

#include <stdlib.h>

#include <time.h>

void shell(int\* items, int count) {

int i, j, gap, k;

int x, a[5] = { 9, 5, 3, 2, 1 };

for (k = 0; k < 5; k++) {

gap = a[k];

for (i = gap; i < count; ++i) {

x = items[i];

for (j = i - gap; x < items[j] && j >= 0; j = j - gap)

items[j + gap] = items[j];

items[j + gap] = x;

}

}

}

void qs(int\* items, int left, int right) {

int i, j, x, y;

i = left; j = right;

x = items[(left + right) / 2];

do {

while (items[i] < x && i < right) i++;

while (x < items[j] && j > left) j--;

if (i <= j) {

y = items[i];

items[i] = items[j];

items[j] = y;

i++; j--;

}

} while (i <= j);

if (left < j) qs(items, left, j);

if (i < right) qs(items, i, right);

}

void print\_time(const char\* name, clock\_t start, clock\_t end) {

printf("%s took %f seconds\n", name, (double)(end - start) / CLOCKS\_PER\_SEC);

}

int main() {

const int SIZE = 10000;

int random\_array[SIZE], sorted\_array[SIZE], reversed\_array[SIZE], mixed\_array[SIZE];

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

random\_array[i] = rand() % SIZE;

sorted\_array[i] = i;

reversed\_array[i] = SIZE - i;

mixed\_array[i] = (i < SIZE / 2) ? i : SIZE - i;

}

clock\_t start, end;

// Shell Sort

start = clock();

shell(random\_array, SIZE);

end = clock();

print\_time("Shell Sort on Random Array", start, end);

start = clock();

shell(sorted\_array, SIZE);

end = clock();

print\_time("Shell Sort on Sorted Array", start, end);

start = clock();

shell(reversed\_array, SIZE);

end = clock();

print\_time("Shell Sort on Reversed Array", start, end);

start = clock();

shell(mixed\_array, SIZE);

end = clock();

print\_time("Shell Sort on Mixed Array", start, end);

// Quick Sort

start = clock();

qs(random\_array, 0, SIZE - 1);

end = clock();

print\_time("Quick Sort on Random Array", start, end);

start = clock();

qs(sorted\_array, 0, SIZE - 1);

end = clock();

print\_time("Quick Sort on Sorted Array", start, end);

start = clock();

qs(reversed\_array, 0, SIZE - 1);

end = clock();

print\_time("Quick Sort on Reversed Array", start, end);

start = clock();

qs(mixed\_array, 0, SIZE - 1);

end = clock();

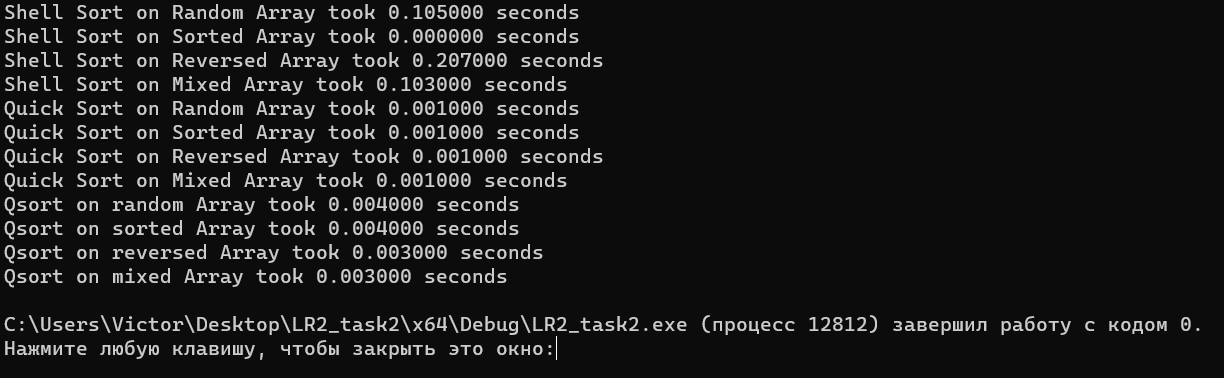
print\_time("Quick Sort on Mixed Array", start, end);

return 0;

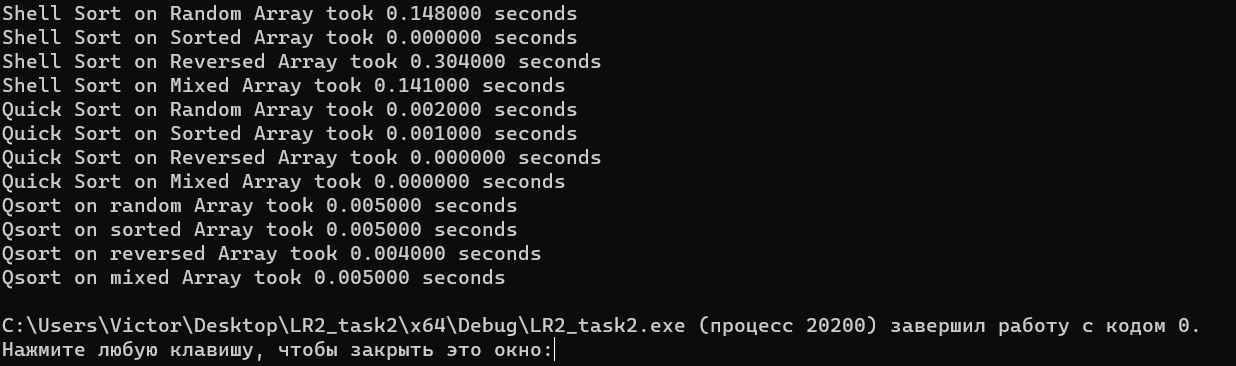
}

**Результаты:**

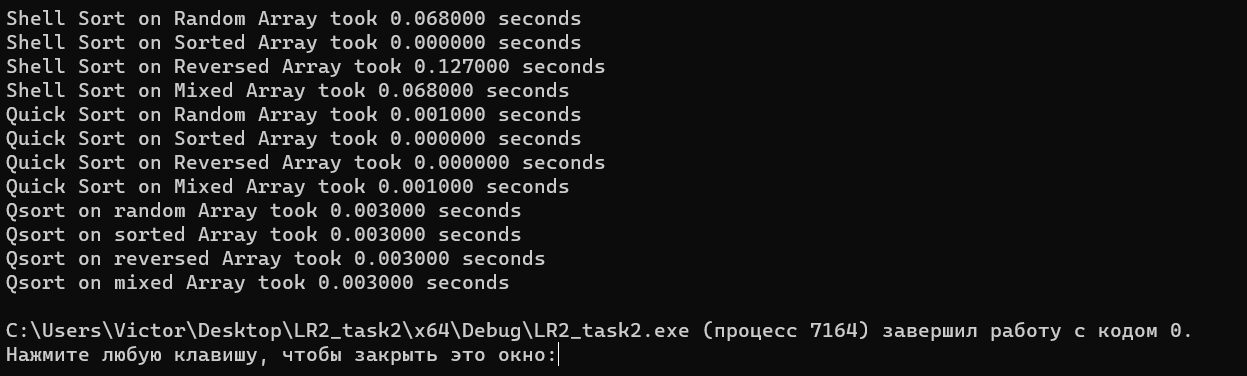
50000 элементов



60000 элементов



40000 элементов



1. Сортировка Шелла:
   * Random: При увеличении размера массива (с 40000 до 60000) алгоритм показывает самый худший результат, по сравнению с другими.
   * Sorted: При увеличении размера массива (с 40000 до 60000) алгоритм показывает самый лучший результат, по сравнению с другими.
   * Reversed: На убывающей последовательности при увеличении размера массива (с 40000 до 60000) алгоритм работает лучше чем с Random, но хуже всех других
   * Mixed: При увеличении размера массива (с 40000 до 60000) алгоритм работает лучше чем с Reversed, но хуже Sorted
2. Быстрая сортировка:
   * Random: При увеличении размера массива (с 40000 до 60000) увеличивается время выполнения программы.
   * Sorted: При увеличении размера массива (с 40000 до 60000) увеличивается время выполнения программы.
   * Reversed: На убывающей последовательности при увеличении размера массива (с 40000 до 60000) затраченное время для выполнения программы не изменилось.
   * Mixed: При увеличении размера массива (с 40000 до 60000) затраченное время для выполнения программы не изменилось.

C Reversed и Mixed алгоритм показал самые лучшие результаты работы равные друг другу.

С двумя другими алгоритм работал в два раза дольше, по сравнению с Reversed и Mixed.

1. Qsort:

Со всеми массивами с увеличением числа массива от 40000 до 60000 программа линейно увеличивала время работы

**Вывод**:

В ходе выполнения лабораторной работы была осуществлена оценка времени программ языка Си и их частей. Также было проведено сравнение времени работы 3 видов сортировки: сортировки Шелла (shell), быстрой сортировки (qs) и стандартной функции qsort, реализующей алгоритм быстрой сортировки. Из всех видов сортировки самой эффективной является алгоритм быстрой сортировки (qs). Функция qsort работает медленнее, так как сортировка универсальная: работает для различных типов данных.

Сортировка Шелла – это эффективный алгоритм сортировки, который может быть полезен в случаях, когда требуется сортировка массивов большого размера или когда изначальный порядок элементов в массиве необходимо сохранить.

Быстрая сортировка является эффективным и универсальным алгоритмом сортировки.