Министерство образования Российской Федерации

Пензенский государственный университет

Кафедра «Вычислительная техника»

**ОТЧЁТ**

по лабораторной работе №2

по курсу «Логика и основы алгоритмизации в ИЗ»

на тему: «Оценка времени выполнения программ»

Выполнил:

студент группы 23ВВВ4

Соснин Глеб

Проверил:

доцент, Юрова О. В.

Пенза, 2024

**Цель** – изучить приёмы оценки времени выполнения программ, отработать навыки измерения времени выполнения программ.

**Задание 1:**

1. **Вычислить порядок сложности программы (О-символику)**.

**Листинг программы:**

#include <iostream>

#include <cstdlib>

#include <ctime> // для замера времени выполнения программы

using namespace std;

int main() {

setlocale(LC\_ALL, "Russian");

// Переменные для замера времени

clock\_t start, end;

// Размер матрицы

int n;

cout << "Введите размер матрицы: ";

cin >> n;

// Динамическое выделение памяти для матриц

int\*\* a = new int\* [n];

int\*\* b = new int\* [n];

int\*\* c = new int\* [n];

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

a[i] = new int[n];

b[i] = new int[n];

c[i] = new int[n];

}

// Заполнение матриц случайными числами

srand(time(0));

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

for (int j = 0; j < n; j++) {

a[i][j] = rand() % 100 + 1; // числа от 1 до 100

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

c[i][j] = 0;

}

}

// Замер времени выполнения перемножения матриц

start = clock();

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

for (int j = 0; j < n; j++) {

for (int k = 0; k < n; k++) {

c[i][j] += a[i][k] \* b[k][j];

}

}

}

end = clock();

double time\_taken = double(end - start) / CLOCKS\_PER\_SEC; // вычисление времени

// Вывод времени выполнения

cout << "Время выполнения перемножения матриц: " << time\_taken << " секунд\n";

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

delete[] a[i];

delete[] b[i];

delete[] c[i];

}

delete[] a;

delete[] b;

delete[] c;

return 0;

}

Алгоритм перемножения двух матриц размером n x n имеет временную сложность O(n^3), так как используются три вложенных цикла, каждый из которых выполняется n раз.

1. **Оценить время выполнения программы и кода, выполняющего перемножение матриц, используя функции библиотеки time.h для матриц размерами от 100, 200, 400, 1000, 2000, 4000, 10000.**

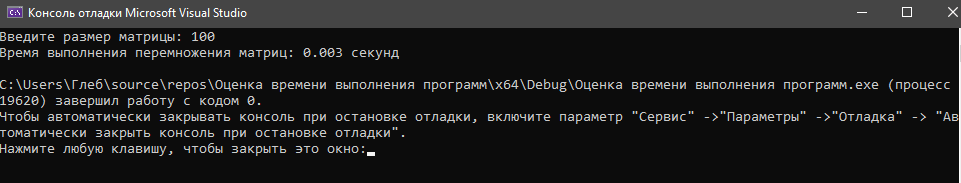


Рисунок 1 – Время выполнения программы для матрицы размером 100

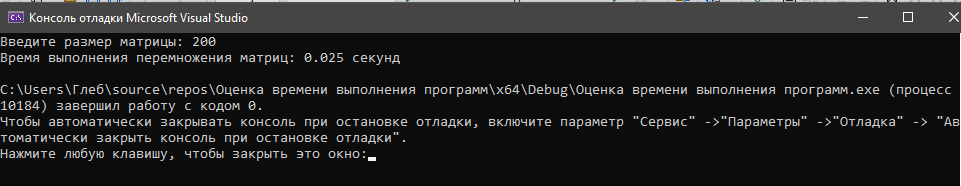


Рисунок 2 – Время выполнения программы для матрицы размером 200

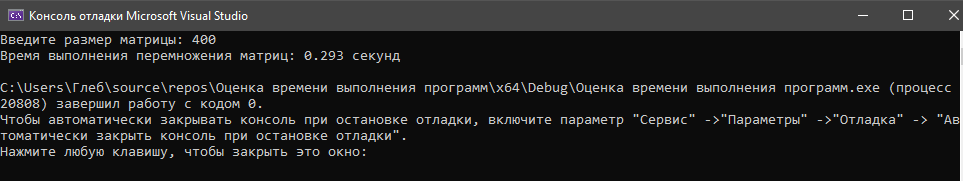


Рисунок 3 – Время выполнения программы для матрицы размером 400

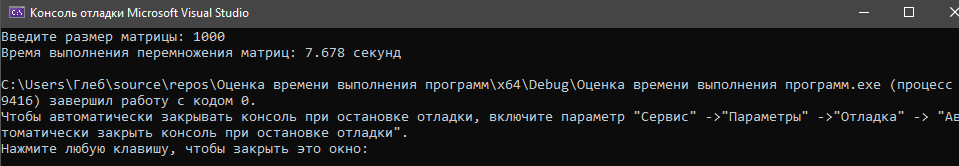


Рисунок 4 – Время выполнения программы для матрицы размером 1000

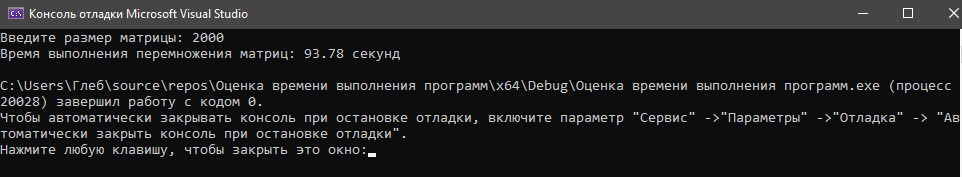


Рисунок 5 – Время выполнения программы для матрицы размером 2000

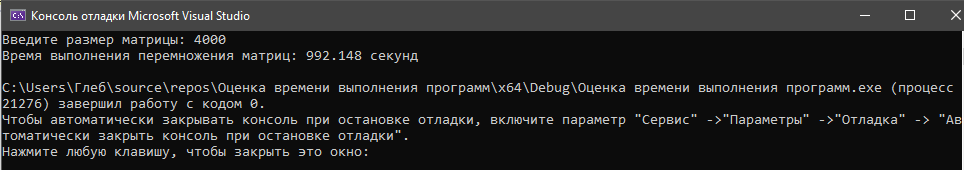


Рисунок 6 – Время выполнения программы для матрицы размером 4000

1. **Построить график зависимости времени выполнения программы от размера матриц и сравнить полученный результат с теоретической оценкой.**

Для построения графика использовали программу на web-сайте . На оси X отложили размер матрицы n, а на оси Y — время выполнения программы.

Таблица 1 – Данные для графика

|  |  |
| --- | --- |
| **Размер матрицы** | **Время выполнения (сек)** |
| 100 | 0,003 |
| 200 | 0,025 |

Продолжение таблицы 1.

|  |  |
| --- | --- |
| 400 | 0,293 |
| 1000 | 7,678 |
| 2000 | 93,78 |
| 4000 | 992,148 |

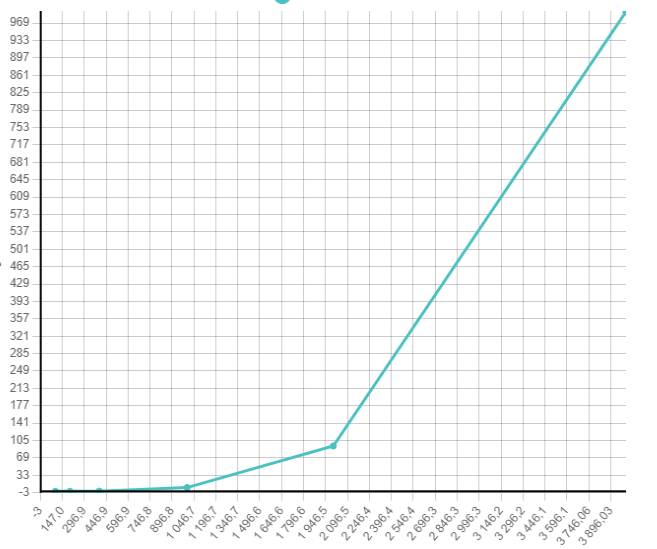


Рисунок 7 – График зависимости времени выполнения программы от размера матриц

Этот график позволяет визуализировать, как сложность O(n^3) влияет на время выполнения при увеличении размера матрицы.

**Задание 2:**

1. **Оценить время работы каждого из реализованных алгоритмов на случайном наборе значений массива.**
2. **Оценить время работы каждого из реализованных алгоритмов на массиве, представляющем собой возрастающую последовательность чисел.**
3. **Оценить время работы каждого из реализованных алгоритмов на массиве, представляющем собой убывающую последовательность чисел.**
4. **Оценить время работы каждого из реализованных алгоритмов на массиве, одна половина которого представляет собой возрастающую последовательность чисел, а вторая, – убывающую.**
5. **Оценить время работы стандартной функции *qsort*, реализующей алгоритм быстрой сортировки на выше указанных наборах данных.**

**Листинг программы:**

#include <iostream>

#include <stack>

#include <cstdlib>

#include <algorithm>

#include <chrono> // Для замера времени

#include <locale>

using namespace std;

using namespace std::chrono;

// Вспомогательная функция для перестановки элементов

void swap(int& a, int& b) {

int temp = a;

a = b;

b = temp;

}

void quicksort\_iterative(int\* items, int left, int right) {

stack<pair<int, int>> stack;

stack.push({ left, right });

while (!stack.empty()) {

auto range = stack.top();

stack.pop();

left = range.first;

right = range.second;

while (left < right) {

int i = left, j = right;

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

while (i <= j) {

while (items[i] < pivot) i++;

while (items[j] > pivot) j--;

if (i <= j) {

swap(items[i], items[j]);

i++;

j--;

}

}

if (j - left < right - i) {

if (left < j) stack.push({ left, j });

left = i;

}

else {

if (i < right) stack.push({ i, right });

right = j;

}

}

}

}

// Реализация сортировки Шелла

void shell\_sort(int\* arr, int size) {

for (int gap = size / 2; gap > 0; gap /= 2) {

for (int i = gap; i < size; i++) {

int temp = arr[i];

int j;

for (j = i; j >= gap && arr[j - gap] > temp; j -= gap) {

arr[j] = arr[j - gap];

}

arr[j] = temp;

}

}

}

// Вспомогательная функция для вызова qsort

int compare(const void\* a, const void\* b) {

return (\*(int\*)a - \*(int\*)b);

}

// Функция для замера времени работы сортировок

void measure\_time(void (\*sort\_func)(int\*, int, int), int\* arr, int size, const string& description) {

int\* copy\_arr = new int[size];

copy(arr, arr + size, copy\_arr);

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

sort\_func(copy\_arr, 0, size - 1);

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

auto duration = duration\_cast<milliseconds>(end - start).count();

cout << description << " заняла: " << duration << " мс." << endl;

delete[] copy\_arr;

}

void std\_sort\_wrapper(int\* arr, int left, int right) {

sort(arr, arr + (right + 1));

}

void qsort\_wrapper(int\* arr, int left, int right) {

qsort(arr, right - left + 1, sizeof(int), compare);

}

// сортировка Шелла

void shell\_sort\_wrapper(int\* arr, int left, int right) {

shell\_sort(arr, right + 1);

}

int main() {

setlocale(LC\_ALL, "Russian");

const int SIZE = 10000;

int arr[SIZE];

// Генерация различных массивов

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

arr[i] = rand() % 100;

}

cout << "Оценка времени работы на случайных данных:" << endl;

measure\_time(quicksort\_iterative, arr, SIZE, "Итеративная быстрая сортировка");

measure\_time(shell\_sort\_wrapper, arr, SIZE, "Сортировка Шелла");

measure\_time(std\_sort\_wrapper, arr, SIZE, "Стандартная std::sort");

measure\_time(qsort\_wrapper, arr, SIZE, "Стандартная qsort");

// Возрастающая последовательность

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

arr[i] = i;

}

cout << "\nОценка времени работы на возрастающей последовательности:" << endl;

measure\_time(quicksort\_iterative, arr, SIZE, "Итеративная быстрая сортировка");

measure\_time(shell\_sort\_wrapper, arr, SIZE, "Сортировка Шелла");

measure\_time(std\_sort\_wrapper, arr, SIZE, "Стандартная std::sort");

measure\_time(qsort\_wrapper, arr, SIZE, "Стандартная qsort");

// Убывающая последовательность

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

arr[i] = SIZE - i;

}

cout << "\nОценка времени работы на убывающей последовательности:" << endl;

measure\_time(quicksort\_iterative, arr, SIZE, "Итеративная быстрая сортировка");

measure\_time(shell\_sort\_wrapper, arr, SIZE, "Сортировка Шелла");

measure\_time(std\_sort\_wrapper, arr, SIZE, "Стандартная std::sort");

measure\_time(qsort\_wrapper, arr, SIZE, "Стандартная qsort");

for (int i = 0; i < SIZE / 2; i++) {

arr[i] = i;

}

for (int i = SIZE / 2; i < SIZE; i++) {

arr[i] = SIZE - i;

}

cout << "\nОценка времени работы на полуупорядоченном массиве (возрастание и убывание):" << endl;

measure\_time(quicksort\_iterative, arr, SIZE, "Итеративная быстрая сортировка");

measure\_time(shell\_sort\_wrapper, arr, SIZE, "Сортировка Шелла");

measure\_time(std\_sort\_wrapper, arr, SIZE, "Стандартная std::sort");

measure\_time(qsort\_wrapper, arr, SIZE, "Стандартная qsort");

return 0;

}

**Пояснение программы:**

Измерение времени:

Для замера времени работы сортировки используется *std::chrono*, который измеряет время выполнения сортировки в миллисекундах.

Массивы:

Случайные значения: создается массив случайных чисел.

Возрастающая последовательность: массив заполняется возрастающими числами.

Убывающая последовательность: массив заполняется убывающими числами.

Полуупорядоченный массив: первая половина массива — возрастающая последовательность, вторая — убывающая.

Сравнение алгоритмов:

Оценка времени работы итеративной быстрой сортировки.

Оценка времени работы стандартной сортировки *std::sort.*

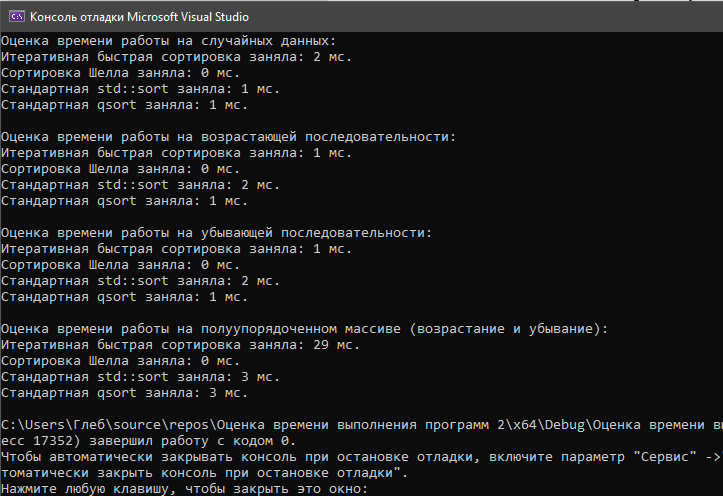
Оценка времени работы стандартной функции *qsort.*

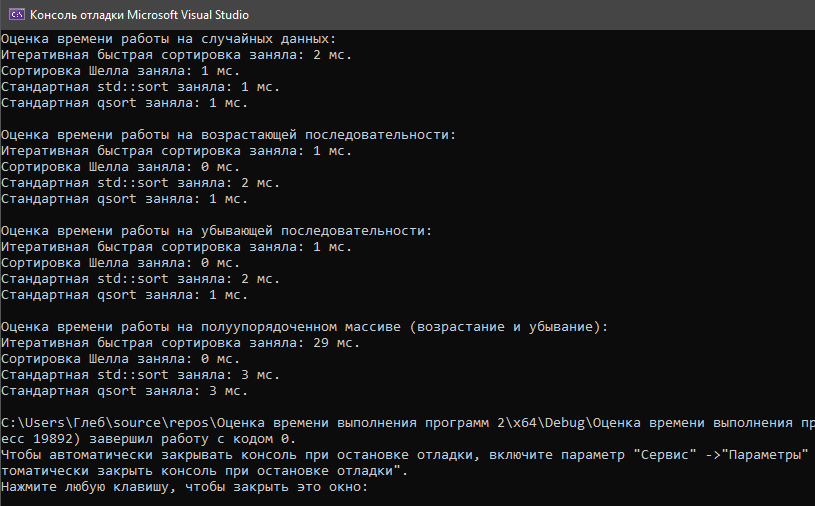
Сортировка Шелла:

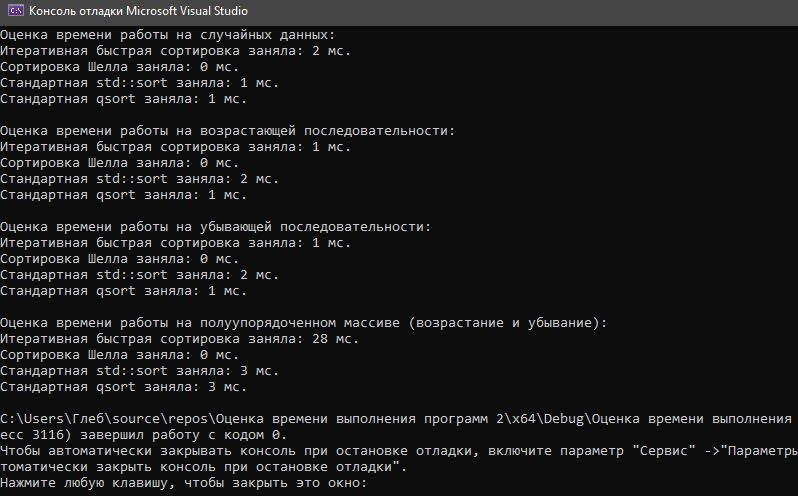
Добавлена функция *shell\_sort*, которая использует последовательность уменьшения шага и сортирует элементы в подмассиве с шагом *gap*.

Добавлена обёртка *shell\_sort\_wrapper* для удобства использования в функции замера времени.

Функция *measure\_time* теперь замеряет время выполнения всех сортировок (итеративной быстрой сортировки, сортировки Шелла, стандартной *std::sort и qsort*).

****

****



Рисунки 8, 9, 10 – Оценка времени работы реализованных алгоритмов

Из скриншотов видно, что программа при повторном запуске генерирует разные значения, следовательно, программа работает.

**Вывод:** изучили приёмы оценки времени выполнения программ, отработали навыки измерения времени выполнения программ.