Министерство науки и высшего образования

Пензенский государственный университет

Кафедра «Вычислительная техника»

ОТЧЕТ

по лабораторной работе №2

по дисциплине: "Логика и основы алгоритмизации в инженерных задачах"

на тему: " Оценка времени выполнения программ"

Выполнили:

студенты группы 23ВВВ4

Королёв Д.В.

Алешин К.А.

Приняли:

Юрова О.В.

Леев М.В.

Пенза, 2024

**Цель работы** - Освоение методов измерения времени выполнения программ и их частей на языке C с использованием стандартной библиотеки time.h, оценки времени выполнения программы и кода, выполняющего перемножение матриц.

**Общие сведения -** Для оценки времени выполнения программ языка Си или их частей могут использоваться средства, предоставляемые библиотекой **time.h**. Данная библиотека содержит описания типов и прототипы функций для работы с датой и временем.

Типы данных:

1. clock\_t - возвращается функцией clock(). Обычно типа int или long int.

2. time\_t - возвращается функцией time(). Обычно определён как int или long int.

3. struct tm - нелинейное, дискретное календарное представление времени.

Основные функции:

1. clock\_t clock(void) - возвращает время, измеряемое процессором в тактах от начала выполнения программы, или −1, если оно не известно. Пересчет этого времени в секунды выполняется по формуле:

clock() / CLOCKS\_PER\_SEC

где CLOCKS\_PER\_SEC – константа, определяющая количество тактов системных часов в секунду.

2. time\_t time(time\_t \*tp)

Возвращает текущее календарное время или −1, если это время не известно. Если указатель tp не равен NULL, то возвращаемое значение записывается также и в \*tp.

3. double difftime(time\_t time2,time\_t time1) Возвращает разность time2-time1, выраженную в секундах.

**Задание 1.**

**1)**



Рисунок № 1 - порядок сложности программы

**2)**

**Листинг**

#include <stdio.h>

#include <stdlib.h>

#include <time.h>

void matrix\_multiply(int size) {

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

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

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

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

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

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

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

}

clock\_t start, end;

double cpu\_time\_used;

int i, j, r, elem\_c;

start = clock(); // Начало измерения времени

srand(time(0));

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

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

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

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

}

}

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

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

elem\_c = 0;

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

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

}

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

}

}

end = clock(); // Конец измерения времени

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

printf("| Size: %d x %d\t| Runtime = %f seconds |\n", size, size, cpu\_time\_used);

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

free(a[i]);

free(b[i]);

free(c[i]);

}

free(a);

free(b);

free(c);

}

int main(void) {

int sizes[] = { 100, 200, 400, 1000, 2000, 4000, 10000 };

int num\_sizes = sizeof(sizes) / sizeof(sizes[0]);

setvbuf(stdin, NULL, \_IONBF, 0);

setvbuf(stdout, NULL, \_IONBF, 0);

for (int i = 0; i < num\_sizes; i++) {

matrix\_multiply(sizes[i]);

}

return 0;

}

Результат работы программы

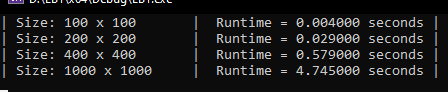


Рисунок № 2

**3)**

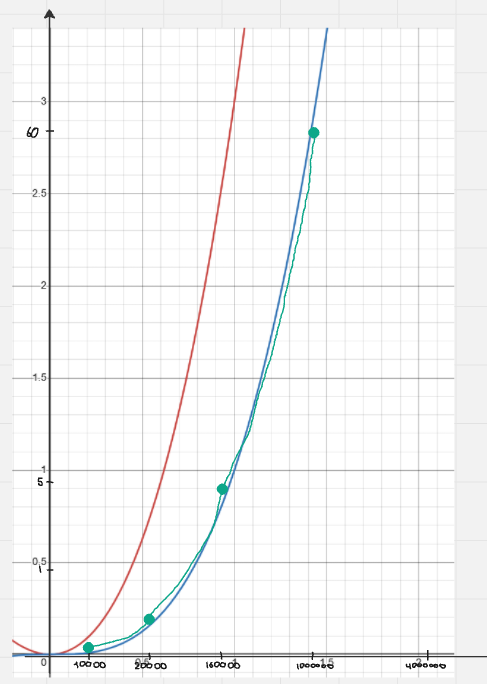


Рисунок №3 - график зависимости времени выполнения программы от размера матриц

Где красная линия – y = 2x^2 + x^3, синияя – y=x^3, зеленая – построенный график

**Вывод по 1 заданию :**

Различные архитектуры компьютеров имеют различную скорость выполнения. Например, процессоры различных производителей могут иметь различную скорость выполнения операций, а также различные поколения процессоров могут иметь различную производительность. Кроме того, различные архитектуры могут иметь различную скорость выполнения операций в зависимости от типа данных, например, целочисленные операции могут выполняться быстрее, чем операции с плавающей запятой. Наличие других процессов, работающих на компьютере, может также повлиять на скорость выполнения программы. Если на компьютере запущено множество других процессов, то процессор будет занят выполнением этих процессов, что может привести к замедлению выполнения программы. Кроме того, наличие других процессов может привести к конкуренции за память и дисковое пространство, что также может замедлить выполнение программы. Поэтому при выполнении тестов производительности программы необходимо убедиться, что на компьютере нет других процессов, которые могут повлиять на результаты тестирования

**Задание 2.**

**Листинг**

#include <iostream>

#include <ctime>

#include <cstdint>

void shell(int\* items, int count)

{

int i, j, gap, k;

int x, a[5];

a[0] = 9; a[1] = 5; a[2] = 3; a[3] = 2; a[4] = 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) //вызов функции: qs(items, 0, count-1);

{

int i, j;

int x, y;

i = left; j = right;

/\* выбор компаранда \*/

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);

}

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

{

int arg1 = \*(const int\*)a;

int arg2 = \*(const int\*)b;

if (arg1 < arg2) return -1;

if (arg1 > arg2) return 1;

return 0;

}

int main()

{

clock\_t start, end;

srand(time(0));

const size\_t SIZE\_FOR\_SHELL = 100000;

{

std::cout << "size array = " << SIZE\_FOR\_SHELL << std::endl;

int help\_var\_for\_sequence = 100000;

int\* arr\_for\_shell = (int\*)malloc(sizeof(int) \* SIZE\_FOR\_SHELL);

for (size\_t i = 0; i < SIZE\_FOR\_SHELL; ++i)

{

arr\_for\_shell[i] = rand() % 50 + 1;

}

start = clock();

shell(arr\_for\_shell, SIZE\_FOR\_SHELL);

end = clock();

std::cout << "shell (random numbers) execution time: " << static\_cast<double>(end - start) / CLOCKS\_PER\_SEC << std::endl;

end = 0;

for (size\_t i = 0; i < SIZE\_FOR\_SHELL; i++)

{

arr\_for\_shell[i] = help\_var\_for\_sequence + 12;

help\_var\_for\_sequence += 12;

}

shell(arr\_for\_shell, SIZE\_FOR\_SHELL);

end = clock();

std::cout << "shell (increasing sequence) execution time: " << static\_cast<double>(end - start) / CLOCKS\_PER\_SEC << std::endl;

help\_var\_for\_sequence = 100000;

end = 0;

for (size\_t i = 0; i < SIZE\_FOR\_SHELL; i++)

{

arr\_for\_shell[i] = help\_var\_for\_sequence;

help\_var\_for\_sequence -= 12;

}

shell(arr\_for\_shell, SIZE\_FOR\_SHELL);

end = clock();

std::cout << "shell (decreasing sequence) execution time: " << static\_cast<double>(end - start) / CLOCKS\_PER\_SEC << std::endl;

help\_var\_for\_sequence = 0;

end = 0;

for (size\_t i = 0; i < SIZE\_FOR\_SHELL / 2; i++)

{

arr\_for\_shell[i] = help\_var\_for\_sequence;

help\_var\_for\_sequence += 12;

}

for (size\_t i = SIZE\_FOR\_SHELL / 2; i < SIZE\_FOR\_SHELL; i++)

{

arr\_for\_shell[i] = help\_var\_for\_sequence;

help\_var\_for\_sequence -= 12;

}

shell(arr\_for\_shell, SIZE\_FOR\_SHELL);

end = clock();

std::cout << "shell (increasing sequence and decreasing sequence) execution time: " << static\_cast<double>(end - start) / CLOCKS\_PER\_SEC << std::endl;

help\_var\_for\_sequence = 0;

}

std::cout << std::endl;

std::cout << std::endl;

const size\_t SIZE\_FOR\_QS = 5000;

{

std::cout << "size array = " << SIZE\_FOR\_QS << " (more than 5000 - stack overflow due to deep recursion)" << std::endl;;

int help\_var\_for\_sequence = 100000;

int\* arr\_for\_qs = (int\*)malloc(sizeof(int) \* SIZE\_FOR\_QS);

for (size\_t i = 0; i < SIZE\_FOR\_QS; ++i)

{

arr\_for\_qs[i] = rand() % 10000 + 1;

}

start = clock();

qs(arr\_for\_qs, 0, SIZE\_FOR\_QS - 1);

end = clock();

std::cout << "qs (random numbers) execution time: " << static\_cast<double>(end - start) / CLOCKS\_PER\_SEC << std::endl;

end = 0;

for (size\_t i = 0; i < SIZE\_FOR\_QS; i++)

{

arr\_for\_qs[i] = help\_var\_for\_sequence + 12;

help\_var\_for\_sequence += 12;

}

qs(arr\_for\_qs, 0, SIZE\_FOR\_QS - 1);

end = clock();

std::cout << "qs (increasing sequence) execution time: " << static\_cast<double>(end - start) / CLOCKS\_PER\_SEC << std::endl;

help\_var\_for\_sequence = 0;

end = 0;

for (size\_t i = 0; i < SIZE\_FOR\_QS; i++)

{

arr\_for\_qs[i] = help\_var\_for\_sequence;

help\_var\_for\_sequence -= 12;

}

qs(arr\_for\_qs, 0, SIZE\_FOR\_QS - 1);

end = clock();

std::cout << "qs (decreasing sequence) execution time: " << static\_cast<double>(end - start) / CLOCKS\_PER\_SEC << std::endl;

help\_var\_for\_sequence = 0;

end = 0;

for (size\_t i = 0; i < SIZE\_FOR\_QS / 2; i++)

{

arr\_for\_qs[i] = help\_var\_for\_sequence;

help\_var\_for\_sequence += 12;

}

for (size\_t i = SIZE\_FOR\_QS / 2; i < SIZE\_FOR\_QS; i++)

{

arr\_for\_qs[i] = help\_var\_for\_sequence;

help\_var\_for\_sequence -= 12;

}

qs(arr\_for\_qs, 0, SIZE\_FOR\_QS - 1);

end = clock();

std::cout << "qs (increasing sequence and decreasing sequence) execution time: " << static\_cast<double>(end - start) / CLOCKS\_PER\_SEC << std::endl;

help\_var\_for\_sequence = 0;

}

std::cout << std::endl;

std::cout << std::endl;

const size\_t SIZE\_FOR\_STANDART\_QS = 100000;

{

std::cout << "size array = " << SIZE\_FOR\_STANDART\_QS << std::endl;

int help\_var\_for\_sequence = 100000;

int\* arr\_for\_standart\_qs = (int\*)malloc(sizeof(int) \* SIZE\_FOR\_STANDART\_QS);

for (size\_t i = 0; i < SIZE\_FOR\_STANDART\_QS; ++i)

{

arr\_for\_standart\_qs[i] = rand() % 50 + 1;

}

start = clock();

qsort(arr\_for\_standart\_qs, SIZE\_FOR\_STANDART\_QS, sizeof(int), compare\_ints);

end = clock();

std::cout << "standard qs (random numbers) execution time: " << static\_cast<double>(end - start) / CLOCKS\_PER\_SEC << std::endl;

end = 0;

for (size\_t i = 0; i < SIZE\_FOR\_STANDART\_QS; i++)

{

arr\_for\_standart\_qs[i] = help\_var\_for\_sequence + 12;

help\_var\_for\_sequence += 12;

}

qsort(arr\_for\_standart\_qs, SIZE\_FOR\_STANDART\_QS, sizeof(int), compare\_ints);

end = clock();

std::cout << "standard qs (increasing sequence) execution time: " << static\_cast<double>(end - start) / CLOCKS\_PER\_SEC << std::endl;

help\_var\_for\_sequence = 0;

end = 0;

for (size\_t i = 0; i < SIZE\_FOR\_STANDART\_QS; i++)

{

arr\_for\_standart\_qs[i] = help\_var\_for\_sequence;

help\_var\_for\_sequence -= 12;

}

qsort(arr\_for\_standart\_qs, SIZE\_FOR\_STANDART\_QS, sizeof(int), compare\_ints);

end = clock();

std::cout << "standard qs (decreasing sequence) execution time: " << static\_cast<double>(end - start) / CLOCKS\_PER\_SEC << std::endl;

help\_var\_for\_sequence = 0;

end = 0;

for (size\_t i = 0; i < SIZE\_FOR\_STANDART\_QS / 2; i++)

{

arr\_for\_standart\_qs[i] = help\_var\_for\_sequence;

help\_var\_for\_sequence += 12;

}

for (size\_t i = SIZE\_FOR\_STANDART\_QS / 2; i < SIZE\_FOR\_STANDART\_QS; i++)

{

arr\_for\_standart\_qs[i] = help\_var\_for\_sequence;

help\_var\_for\_sequence -= 12;

}

qsort(arr\_for\_standart\_qs, SIZE\_FOR\_STANDART\_QS, sizeof(int), compare\_ints);

end = clock();

std::cout << "standard qs (increasing sequence and decreasing sequence) execution time: " << static\_cast<double>(end - start) / CLOCKS\_PER\_SEC << std::endl;

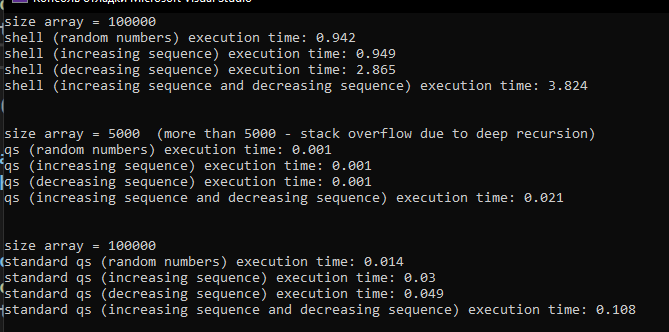
help\_var\_for\_sequence = 0;

}

return 0;

}

**Результат работы программы**



**Вывод по 2 заданию :**

**Алгоритм сортировки Шелла**

Алгоритм сортировки Шелла был разработан Дональдом Шеллом в 1959 году и является усовершенствованным вариантом сортировки вставками. Он работает по принципу сначала сортирует элементы на определенном расстоянии друг от друга, а затем уменьшает это расстояние и повторяет процесс до тех пор, пока расстояние не станет равным 1.

На упорядоченных последовательностях элементов алгоритм сортировки Шелла может работать очень быстро, так как он сначала сортирует элементы на большом расстоянии друг от друга, а затем уменьшает это расстояние и повторяет процесс до тех пор, пока расстояние не станет равным 1. На упорядоченных последовательностях, такой подход может привести к тому, что алгоритм будет выполнять меньше лишних операций, что ускорит его работу.

Однако, на случайном наборе значений массива и на массиве, одна половина которого представляет собой возрастающую последовательность чисел, а вторая – убывающую, время работы алгоритма Шелла оказалось медленнее, чем время работы алгоритма qsort. На случайном наборе значений массива, такой подход может привести к тому, что алгоритм будет выполнять много лишних операций, что замедлит его работу. А на массиве, одна половина которого представляет собой возрастающую последовательность чисел, а вторая – убывающую, алгоритм Шелла сталкивается с разными порядками элементов, что замедляет его работу.

**Алгоритм быстрой сортировки qsort**

Алгоритм быстрой сортировки qsort является одним из наиболее распространенных алгоритмов сортировки. Он был разработан Тони Хоаром в 1960 году и работает на основе выбора опорного элемента, который разбивает массив на две части: элементы меньше опорного и элементы больше опорного. При правильном выборе опорного элемента, алгоритм qsort может быстро отсортировать массив.

На случайном наборе значений массива и на массиве, одна половина которого представляет собой возрастающую последовательность чисел, а вторая – убывающую, время работы алгоритма qsort оказалось быстрее, чем время работы алгоритма Шелла. Это связано с тем, что алгоритм qsort работает на основе выбора опорного элемента, который может быть выбран оптимальным образом в случае случайного набора значений и на массиве, где порядок элементов меняется.

Однако, на упорядоченных последовательностях, время работы qsort увеличивается, так как он может выбрать неудачный опорный элемент, что замедлит его работу. В таких случаях алгоритм Шелла может работать быстрее.

**Стандартная функция qsort**

Стандартная функция qsort – это функция стандартной библиотеки языка Си, которая реализует алгоритм быстрой сортировки. Она работает на основе выбора опорного элемента, который может быть выбран оптимальным образом в случае случайного набора значений и на массиве, где порядок элементов меняется.

Стандартная функция qsort оказалась быстрее на случайном наборе значений массива и на массиве, одна половина которого представляет собой возрастающую последовательность чисел, а вторая – убывающую. Однако, на упорядоченных последовательностях, время работы qsort увеличивается, так как он может выбрать неудачный опорный элемент, что замедлит его работу. В таких случаях алгоритм Шелла может работать быстрее.

**Вывод**

Таким образом, для конкретной задачи сортировки массива необходимо учитывать порядок элементов в массиве и выбирать алгоритм, который будет работать наиболее эффективно в данном случае. На случайном наборе значений массива и на массиве, одна половина которого представляет собой возрастающую последовательность чисел, а вторая – убывающую, стандартная функция qsort оказалась быстрее. На упорядоченных последовательностях, алгоритм сортировки Шелла может работать быстрее. При выборе алгоритма сортировки необходимо учитывать все эти факторы и выбирать оптимальный алгоритм для конкретной задачи.