Выполнили:

студенты группы 23ВВВ4

Зарубин Я.Д.

Герасимов К.Б.

Приняли:

Деев М.В.

Юрова О.В.

**Название**

Оценка времени выполнения программ

**Цель работы**

Осуществить замер времени работы программы библиотекой time.h, содержащую описания типов и прототипы функций для работы с датой и временем.

**Лабораторное задание**

На основе приведенных в лабораторной работе примеров выполнить следующие задания:

**Задание 1:**

1. Вычислить порядок сложности программы (*О*-символику).
2. 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.
3. Построить график зависимости времени выполнения программы от размера матриц и сравнить полученный результат с теоретической оценкой.

**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.

**Описание метода решения задачи**

На основе описанного программного кода, опираясь на теорию, вычислим порядок сложности программы и оценим время выполнения перемножения матриц с соответствующим графиком зависимости

### Листинг

а)

#include <stdio.h>

#include <stdlib.h>

#include <time.h>

#define arraysize 10000

voidmain(void){

clock\_tstart, end; // объявляем переменные для определения времени выполнения

int i = 0, j = 0, r;

char\* a = (char\*)malloc(arraysize \* arraysize);

char\* b = (char\*)malloc(arraysize \* arraysize);

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

int elem\_c;

//char a[arraysize][arraysize], b[arraysize][arraysize], c[arraysize][arraysize], elem\_c;

srand(time(NULL)); // инициализируем параметры генератора случайных чисел

while (i < arraysize)

{

while (j < arraysize)

{

\*(a + i \* arraysize + j) = rand() % 100 + 1; // заполняем массив случайными числами

j++;

}

i++;

}

srand(time(NULL)); // инициализируем параметры генератора случайных чисел

i = 0; j = 0;

while (i < arraysize)

{

while (j < arraysize)

{

\*(b + i \* arraysize + j) = rand() % 100 + 1; // заполняем массив случайными числами

j++;

}

i++;

}

start = clock();

for (i = 0; i < arraysize; i++)

{

for (j = 0; j < arraysize; j++)

{

elem\_c = 0;

for (r = 0; r < arraysize; r++)

{

elem\_c = (int) elem\_c + \*(a + i \* arraysize + r) \* \*(b + i \* arraysize + j);

\*(c + i \* arraysize + j) = elem\_c;

}

}

}

end = clock();

double time\_spent = (double)(end - start) / CLOCKS\_PER\_SEC;

printf("Time spent:%lf s", time\_spent);

free(a);

free(b);

free(c);

}

b)

#include <stdio.h>

#include <stdlib.h>

#include <time.h>

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(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() {

int arraysize[2] = { 100,100000 };

clock\_t start, end;

for (char k = 0; k < 8; k++) {

srand(time(NULL));

int\* a = (int\*)malloc(arraysize[k % 2] \* sizeof(int));

int\* b = (int\*)malloc(arraysize[k % 2] \* sizeof(int));

int\* c = (int\*)malloc(arraysize[k % 2] \* sizeof(int));

for (int i = 0; i < arraysize[k % 2]; i++){

switch (k / 2) {

case 0: {

if (i == 0) printf("\nRandom, %i elements:\n", arraysize[k % 2]);

b[i] = c[i] = a[i] = rand() % 101;

break;

}

case 1: {

if (i == 0) printf("\nSorted, %i elements:\n", arraysize[k % 2]);

a[i] = b[i] = c[i] = i;

break;

}

case 2: {

if (i == 0) printf("\nSorted backwords, %i elements:\n", arraysize[k % 2]);

a[i] = b[i] = c[i] = arraysize[k % 2] - i;

break;

}

case 3: {

if (i == 0) printf("\nUp and down, %i elements:\n", arraysize[k % 2]);

if (i = arraysize[k % 2] / 2) {

i = arraysize[k % 2];

break;

}

a[i] = b[i] = c[i] = i;

a[arraysize[k % 2] - i - 1] = b[arraysize[k % 2] - i - 1] = c[arraysize[k % 2] - i - 1] = i;

break;

}

}

}

start = clock();

shell(a,arraysize[k%2]);

end = clock();

printf("Shell: %lf s\n", (double)(end - start) / CLOCKS\_PER\_SEC);

start = clock();

qs(b,0,arraysize[k%2]-1);

end = clock();

printf("qs: %lf s\n", (double)(end - start) / CLOCKS\_PER\_SEC);

start = clock();

qsort(c,arraysize[k%2],sizeof(int),compare);

end = clock();

printf("qsort: %lf s\n", (double)(end - start) / CLOCKS\_PER\_SEC);

free(a);

free(b);

free(c);

}

}

**Пояснительный текст к программе**

А) Наибольшая сложность в программе — это сложность умножения матриц, которая является доминирующей. Следовательно, общая сложность программы составляет:

O(arraysize³)

Это означает, что время выполнения программы увеличивается кубически с увеличением размера массивов.

Время выполнения программы и кода, выполняющего перемножение матриц, используя функции библиотеки time.h для матриц размерами от

100: 0,003 c.

200: 0,032 c.

400: 0,234 c.

1000: 3,546 c.

2000: 28,419 c.

4000: 227,7 c.

10000: 3574,127 c.

Из графика зависимости t от размера массива и графика O - символики видно, что предположительная зависимость оказалась верна.

Б) По заданию 2, мы провели 4 замера времени каждого из реализованных алгоритмов в массивах с различными вариациями сортировки на соответствующей размерности: 100 и 100000 элементов.

1)Случайный набор значений массива

100 элементов: Shell: 0.000000 s, qs: 0.000000, qsort: 0.001000 s

100000 элементов: Shell: 1.022000 s, qs: 0.010000 s, qsort: 0.038000 s

Исходя из выше приведенных значений времени выполнения, можно выявить, что на 100 элементах алгоритмы сортировки Шелла и быстрой сортировки почти не занимают так такого времени, их выполнение неизмеримо быстро, когда как обычная функция занимает 1 тысячную секунды.

Хотя на 100000 алгоритм Шелла стал самый медленный (чуть более одной в отличии от быстрой сортировки (наиболее быстрой 1 сотая секунды), среднее значение заняла стандартная функция.

Таким образом на случайном наборе значений лучше всего использовать алгоритм быстрой сортировки

2)Возрастающая последовательность чисел, 100 элементов:

Shell: 0.000000 s

qs: 0.000000 s

qsort: 0.000000 s

100000 элементов:

Shell: 0.003000 s

qs: 0.007000 s

qsort: 0.051000 s

Исходя из выше приведенных значений времени выполнения, можно выявить, что на 100 элементах алгоритмы сортировки и функция сортировки почти не занимают так такого времени, их выполнение неизмеримо быстро.

На возрастающей последовательности по времени выигрывает алгоритм Шелла, который вероятно и следует использовать.  
Как и следовало ожидать, функция и в данном случае работала медленнее, чем предложенный алгоритм.

3) Убывающая последовательность, 100 элементов:

Shell: 0.000000 s

qs: 0.000000 s

qsort: 0.000000 s

100000 элементов:

Shell: 2.067000 s

qs: 0.007000 s

qsort: 0.059000 s

К сожалению на 100 элементах и в данном случае не рассмотреть скорость выполнения.

На убывающей последовательности как и в случае со случайным набором чисел победителем по времени является алгоритм быстрой сортировки, время кстати аналогично совпадает с пунктом 2 (0.007 s.).   
В остальном ситуация аналогична пункту 1, аутсайдер – алгоритм Шелла, после него идет стандартная функция.

4) Половина – возрастающая, половина - убывающая, 100 элементов:

Shell: 0.000000 s

qs: 0.000000 s

qsort: 0.000000 s

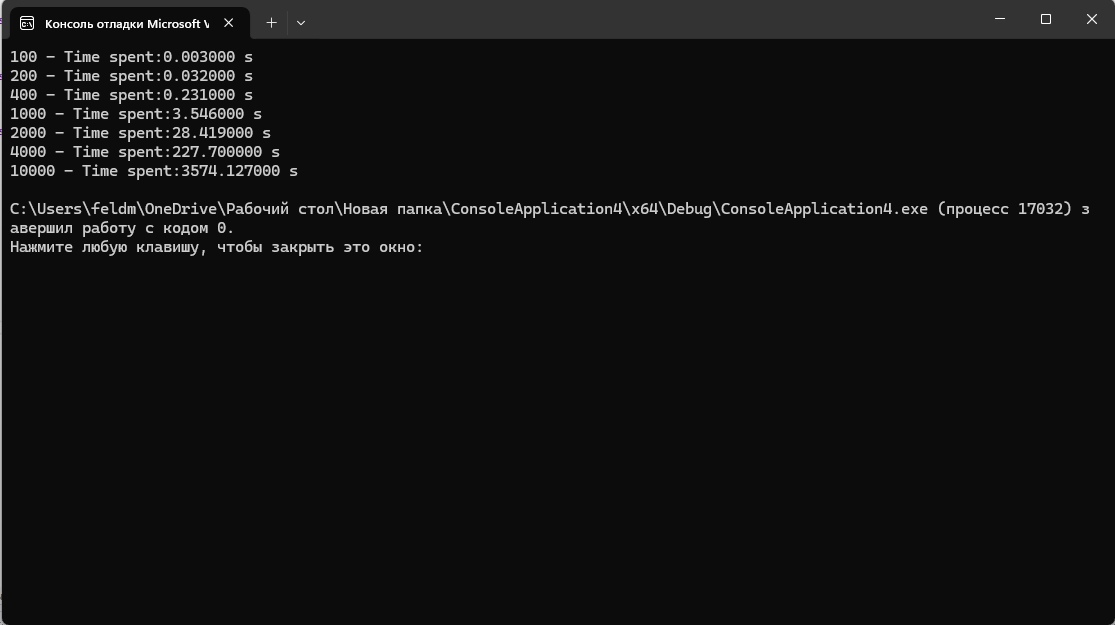
100000 элементов:

Shell: 0.002000 s

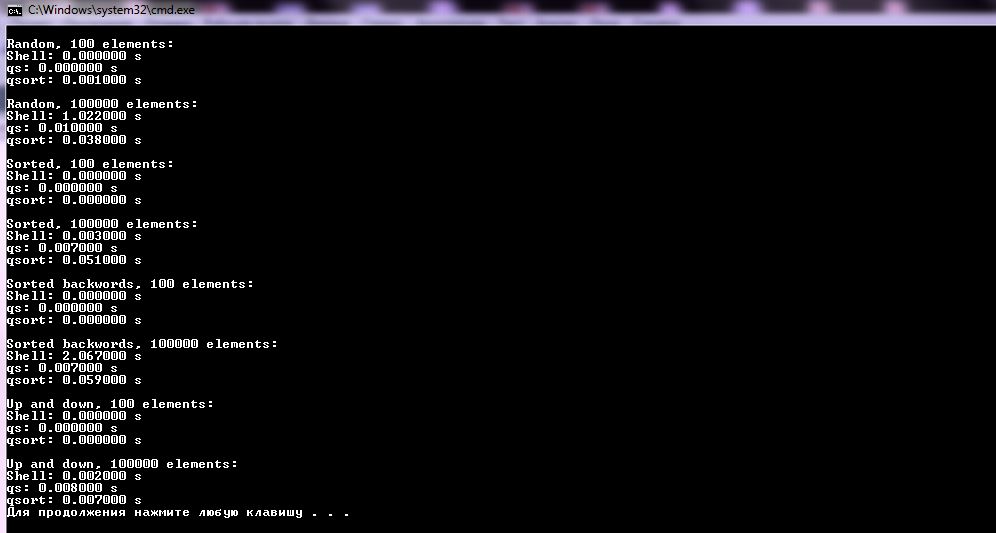
qs: 0.008000 s

gsort: 0.007000 s

В такой сортировке победа за алгоритмом Шелла. Странно то, что алгоритм быстрой сортировки оказался медленнее, чем стандартная функция.

**Результат работы программы**

**Рисунок 3 — Результаты работы программы а.**

****

**Рисунок 4 — Результаты работы программы b.**

Результат работы программы, показанный на рисунке 1, совпал с результатами трассировки.

### Выводы

Повторили принципы использования и применения простых структур данных языка Си (массивов, строк) для решения поставленных задач.