Министерство науки и высшего образования Российской Федерации
Пензенский Государственный Университет
Кафедра “Вычислительная техника”

ОТЧЕТ

по лабораторной работе №2
по курсу «Логика и основы алгоритмизации в инженерных задачах»
на тему «Оценка времени выполнения программ»

Выполнили:
студенты группы 23ВВВ4
Брагин А.М.
Зарубин Я.Д.
Герасимов К.Б.

Приняли:
Деев М.В.
Юрова О.В.

Пенза 2024

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 определён как 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.

Практическая часть
Дана программа, вычисляющая произведение двух матриц:

#include <stdio.h>
#include <stdlib.h>

```
#include &lt;time.h&gt;
int main(void)
{
setvbuf(stdin, NULL, _IONBF, 0);
setvbuf(stdout, NULL, _IONBF, 0);
clock_t start, end; // объявляем переменные для определения времени выполнения
int i=0, j=0, r;
int a[200][200], b[200][200], c[200][200], elem_c;
srand(time(NULL)); // инициализируем параметры генератора случайных чисел
while(i&lt;200)
{
while(j&lt;200)
{
a[i][j]=rand()% 100 + 1; // заполняем массив случайными числами
j++;
}
i++;
}
srand(time(NULL)); // инициализируем параметры генератора случайных чисел
i=0; j=0;
while(i&lt;200)
{
while(j&lt;200)
{
b[i][j]=rand()% 100 + 1; // заполняем массив случайными числами
j++;
}
i++;
}
for(i=0;i&lt;200;i++)
{
for(j=0;j&lt;200;j++)
{
elem_c=0;
```

```
for(r=0;r&lt;200;r++)
{
elem_c=elem_c+a[i][r]*b[r][j];
c[i][j]=elem_c;
}
}
}
return(0);
}
```

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.

Листинг

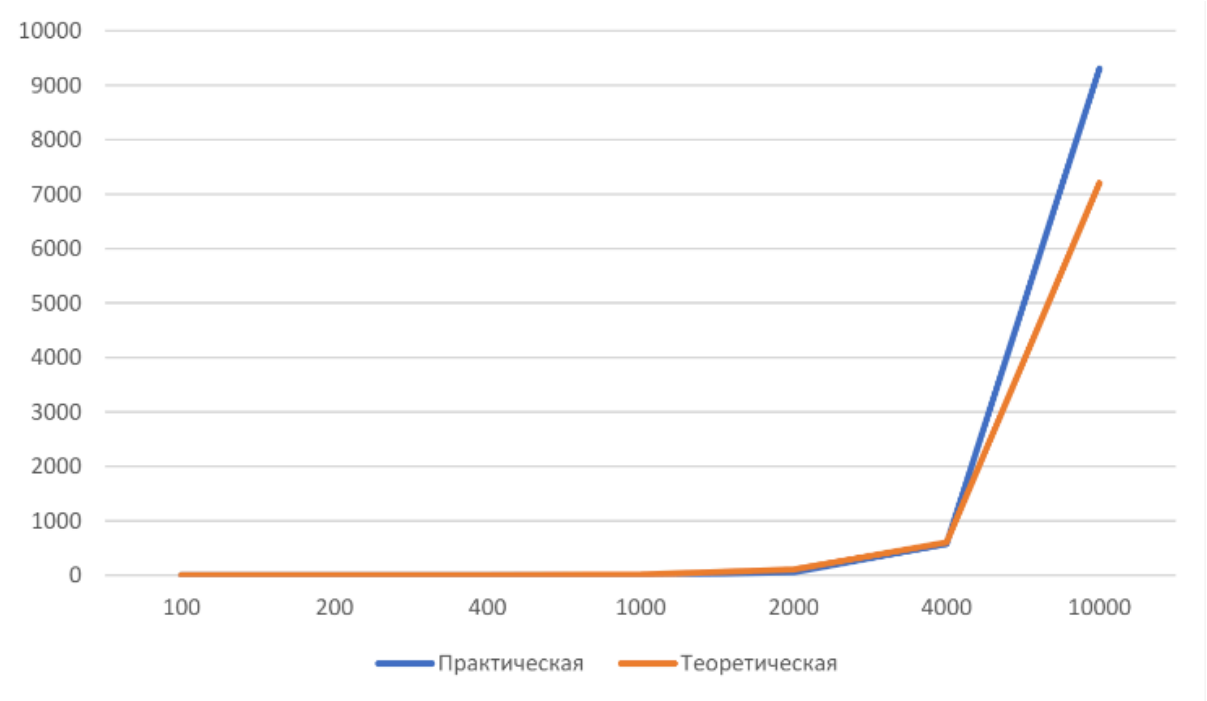
```
#include <stdio.h>
#include <stdlib.h>
#include <time.h>

int main(void)
{
   setvbuf(stdin, NULL, _IONBF, 0);
   setvbuf(stdout, NULL, _IONBF, 0);

   clock_t start, end;
   int i, j, r;
   int n = 10000; // размер матрицы
```

```
    // Динамическое выделение памяти для матриц
    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));
    }

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

    // Заполнение матрицы a случайными числами
    for (i = 0; i < n; i++) {
        for (j = 0; j < n; j++) {
            a[i][j] = rand() % 100 + 1;
        }
    }

    // Заполнение матрицы b случайными числами
    for (i = 0; i < n; i++) {
        for (j = 0; j < n; j++) {
            b[i][j] = rand() % 100 + 1;
        }
    }

    // Перемножение матриц
    start = clock();
    for (i = 0; i < n; i++) {
        for (j = 0; j < n; j++) {
            int elem_c = 0;
            for (r = 0; r < n; r++) {
                elem_c += a[i][r] * b[r][j];
            }
            c[i][j] = elem_c;
```

```
        }
    }
    end = clock();

    double time_spent = (double)(end - start) / CLOCKS_PER_SEC;
    printf("%f", time_spent);

    // Освобождение динамически выделенной памяти
    for (i = 0; i < n; i++) {
        free(a[i]);
        free(b[i]);
        free(c[i]);
    }
    free(a);
    free(b);
    free(c);

    return 0;
}
```

```
0.003000 0.018000 0.173000 4.875000 52.645000 596.596000 9312.5
```

100	**200**	**400**	**1000**	**2000**	**4000**	**10000**
0.003	0.018	0.173	4.875	52.645	596.596	9312.5



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.

Листинг

```
#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) {
```

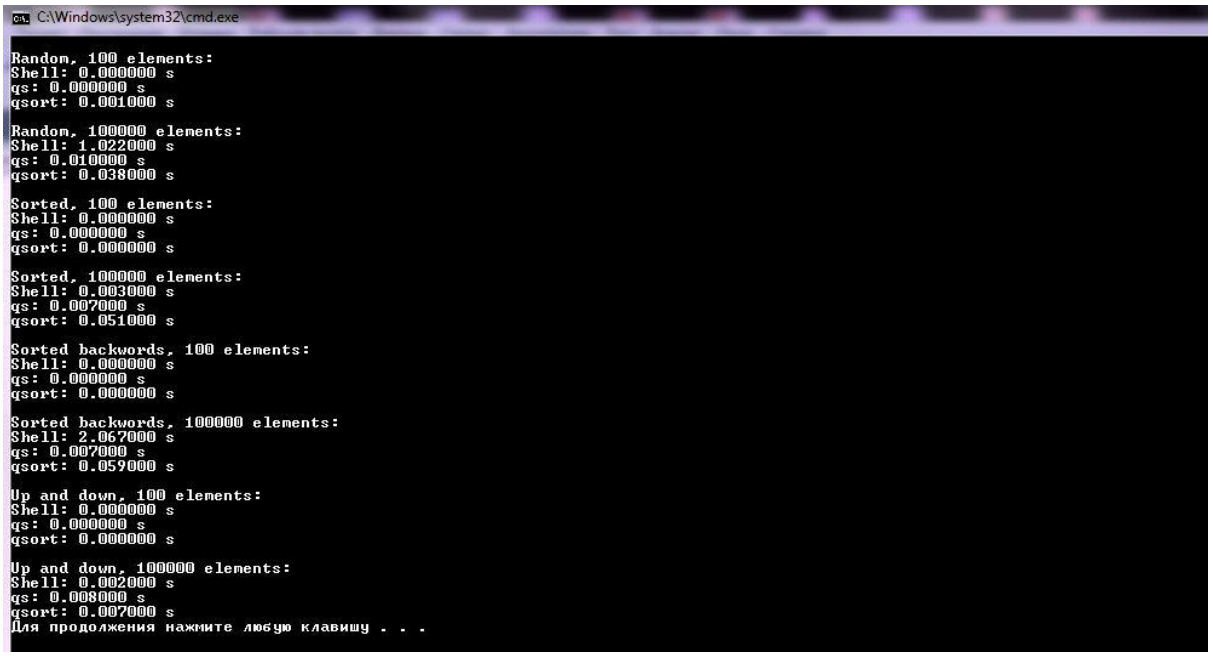
```
            y = items[i];
            items[i] = items[j];
            items[j] = y;
            i++; j--;
        }
    } while (i <= j);

    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] = { 1000,100000 };
    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);
```

```
        }


}
```

Результат работы программы



Выводы

Для маленьких наборов данных (1000 элементов), разница в скорости выполнения между алгоритмами незначительна. Все три алгоритма показывают практически мгновенное выполнение как на случайных, так и на отсортированных и обратно отсортированных данных. Это позволяет сделать вывод, что любой из алгоритмов (Shell, qs, qsort) одинаково хорошо подходит для сортировки небольших объемов данных.

Однако, для больших наборов данных (100,000 элементов) наблюдается значительное различие в производительности. Shell сорт показал значительно худшие результаты на случайных и обратно отсортированных данных, занимая 0.723000 с и 1.304000 с соответственно. В то же время, алгоритмы быстрой сортировки qs и qsort оказались существенно быстрее: qsort выполнил сортировку случайных данных за 0.019000 с, а qs — за 0.008000 с. Аналогичные результаты наблюдаются и на других типах данных, где быстрая сортировка значительно эффективнее.

Таким образом, для больших наборов данных или сложных типов данных, таких как случайные или обратно отсортированные, предпочтение следует отдавать быстрой сортировке (как qs, так и qsort). Shell сортировка оказывается менее эффективной для больших объемов данных, особенно при работе с неупорядоченными наборами.