Министерство науки и высшего образования Российской Федерации

Пензенский государственный университет

Кафедра «Вычислительная техника»

**ОТЧЁТ**

По лабораторной работе №2

По курсу «Логика и основы алгоритмизации в инженерных задачах»

на тему «Оценка времени выполнения программ»

**Выполнили**

**студенты группы 23ВВВ2:**

Пырков Д. А.

Родионов А. А.

**Приняли:**

Митрохин М. А.

Юрова О. В.

Пенза 2024

**Цель работы:** Провести анализ эффективности и сравнительную оценку времени выполнения различных алгоритмов обработки данных (включая перемножение матриц и алгоритмы сортировки) на разных наборах входных данных, а также сопоставить полученные практические результаты с теоретическими оценками сложности алгоритмов.

**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.

**Ход работы**

**Задание 1**

**1. Определение порядка сложности программы.** Исходная программа:

#include <stdio.h>

#include <stdlib.h>

#include <time.h>

int main(void) {

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

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

clock\_tstart, end;

inti=0, j=0, r;

int a[200][200], b[200][200], c[200][200], elem\_c;

srand(time(NULL));

while(i<200) {

while(j<200) {

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

j++;

}

i++;

}

srand(time(NULL));

i=0; j=0;

while(i<200) {

while(j<200) {

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

j++;

}

i++;

}

for(i=0;i<200;i++) {

for(j=0;j<200;j++) {

elem\_c=0;

for(r=0;r<200;r++) {

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

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

}

}

}

return(0);

}

Анализ сложности программы:

1. Первые два цикла while заполняют матрицы a и b случайными числами. Каждый из этих циклов имеет сложность O(n^2), так как они проходят через все элементы матрицы n x n.
2. Основная часть программы - это тройной вложенный цикл for, который выполняет умножение матриц. Этот цикл имеет сложность O(n^3), так как каждый из трех циклов выполняется n раз.
3. Остальные операции в программе (инициализация переменных, вызовы функций rand() и т.д.) имеют константную сложность O(1) и не влияют на общую асимптотическую сложность.

Итоговая сложность: O(n2) + O(n2) + O(n3) = O(n3)

**2. Оценка времени перемножения матриц.**

**Описание программы**

Для выполнения данного задания исходная программа была доработана.

Новая программа работает следующим образом:

1. Определяет 7 указанных размеров квадратных матриц: 100, 200, 400, 1000, 2000, 4000, 10000.
2. Для каждого размера вызывает функцию Matrix\_mult, которая:

* Создает две случайные матрицы A и B заданного размера;
* Выполняет умножение матриц A и B, сохраняя результат в матрице C;
* Измеряет время выполнения умножения матриц;
* Выводит таблицу с результатами работы программы;
* Освобождает выделенную память после каждого умножения.

1. Процесс повторяется для всех заданных размеров матриц.

Листинг программы смотрите в Приложении А.

**Результат работы программы**

На рисунке 1 приведён результат работы программы №1.

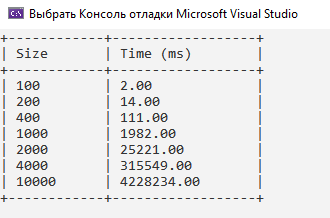


Рисунок 1 – результат работы программы №1

**3. Построение графика и сравнение с теоретической оценкой.**

По результатам работы программы был построен график (см. Рисунок 2).

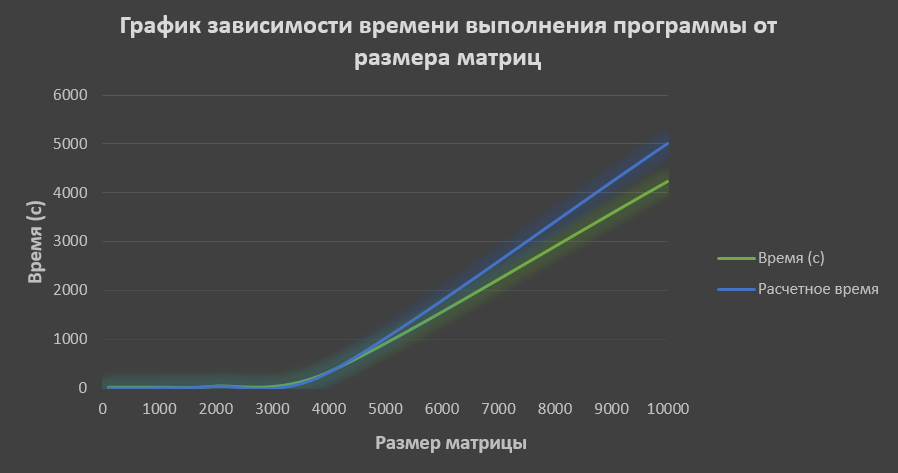


Рисунок 2 - график

**Теоретическая оценка**

Согласно O(n3), при увеличении размера матрицы в k раз, время выполнения должно увеличиваться примерно в k3 раз.

Анализ результатов:

* От 100 до 200: Увеличение в 2 раза, время увеличилось примерно в 7 раз (14.00 / 2.00 ≈ 7)
* От 200 до 400: Увеличение в 2 раза, время увеличилось примерно в 7.93 раза (111.00 / 14.00 ≈ 7.93)
* От 400 до 1000: Увеличение в 2.5 раза, время увеличилось примерно в 17.86 раза (1982.00 / 111.00 ≈ 17.86)
* От 1000 до 2000: Увеличение в 2 раза, время увеличилось примерно в 12.73 раза (25221.00 / 1982.00 ≈ 12.73)
* От 2000 до 4000: Увеличение в 2 раза, время увеличилось примерно в 12.51 раза (315549.00 / 25221.00 ≈ 12.51)
* От 4000 до 10000: Увеличение в 2.5 раза, время увеличилось примерно в 13.38 раза (4228234.00 / 315549.00 ≈ 13.38)

Сравнение с теорией:

* Для увеличения в 2 раза теоретически ожидаем увеличение времени в 23 = 8 раз
* Для увеличения в 2.5 раза теоретически ожидаем увеличение времени в 2.53 ≈ 15.63 раза

**Выводы**

Результаты в целом соответствуют теоретической оценке O(n3), но есть некоторые отклонения:

* Для малых размеров (100-400) наблюдается меньший рост, чем предсказывает теория.
* Для средних и больших размеров (1000-10000) рост близок к теоретическому, но немного выше ожидаемого.

Эти отклонения могут быть вызваны различными факторами, такими как оптимизации компилятора, особенности архитектуры процессора, влияние кэш-памяти и т.д. В целом, практические результаты подтверждают теоретическую оценку сложности O(n3) для алгоритма умножения матриц.

**Задание 2**

**Описание программы**

С использованием предоставленных алгоритмов сортировки была написана программа, которая сравнивает производительность трех алгоритмов сортировки (сортировка Шелла, быстрая сортировка и стандартная функция qsort) на различных наборах данных. Программа работает следующим образом:

1. Определяются функции сортировки:

* shell: реализация сортировки Шелла;
* qs: реализация быстрой сортировки;
* Стандартная функция qsort.

2. Определяются функции заполнения массива:

* Fill\_rand: случайными числами;
* Fill\_ascending: возрастающей последовательностью;
* Fill\_descending: убывающей последовательностью;
* Fill\_asc\_desc: первая половина возрастает, вторая убывает.

3. В main():

* Выделяется память под массив размером Size (150000 элементов);
* Для каждого алгоритма сортировки:  
  **a**. Заполняется массив каждым из четырех способов;  
  **b**. Выполняется сортировка;  
  **c**. Измеряется время выполнения сортировки;  
  **d**. Результаты выводятся в виде таблицы.
* Измерение времени выполняется с помощью функций clock() и CLOCKS\_PER\_SEC;
* Результаты выводятся в виде таблицы, где строки соответствуют алгоритмам сортировки, а столбцы - различным наборам данных. Время отображается в миллисекундах;
* После выполнения всех тестов память, выделенная под массив, освобождается.

Листинг программы смотрите в Приложении В.

**Результат работы программы**

На рисунке 3 приведён результат работы программы №2. Размер каждого массива - 150000 элементов.

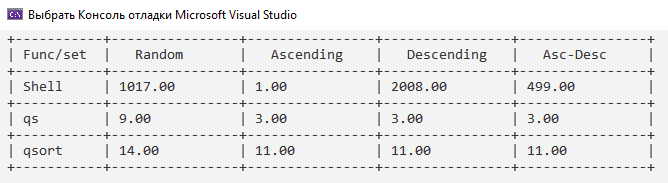


Рисунок 3 – результат работы программы №2

**Теоретическая оценка**

1. Сортировка Шелла (Shell):

* Хорошо справляется с частично отсортированными данными (Asc-Desc: 499 мс)
* Очень эффективна на уже отсортированных данных (Ascending: 1 мс)
* Медленнее на случайных (Random: 1017 мс) и обратно отсортированных данных (Descending: 2008 мс)

1. Быстрая сортировка (qs):
   * Показывает отличные результаты на всех типах данных (3-9 мс)
   * Наиболее эффективна среди всех используемых алгоритмов
2. Стандартная функция qsort:
   * Демонстрирует стабильные результаты на всех типах данных (11-14 мс)
   * Немного медленнее, чем ручная реализация быстрой сортировки

**Выводы**

Быстрая сортировка (qs) показывает наилучшие результаты во всех сценариях.

Сортировка Шелла эффективна на отсортированных и частично отсортированных данных, но существенно медленнее на случайных и обратно отсортированных.

Стандартная функция qsort показывает стабильные, но не самые быстрые результаты.

Выбор алгоритма может зависеть от ожидаемого характера входных данных и требований к производительности.

**Приложение А**

**Листинг программы для задания 1**

#include <stdio.h>

#include <stdlib.h>

#include <time.h>

#define Size\_1 100

#define Size\_2 200

#define Size\_3 400

#define Size\_4 1000

#define Size\_5 2000

#define Size\_6 4000

#define Size\_7 10000

void Matrix\_mult(int size);

int main(void) {

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

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

int sizes[] = { Size\_1, Size\_2, Size\_3, Size\_4, Size\_5, Size\_6, Size\_7 };

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

printf(" +-----------+------------------+\n");

printf(" | Size | Time (ms) |\n");

printf(" +-----------+------------------+\n");

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

Matrix\_mult(sizes[i]);

}

return 0;

}

void Matrix\_mult(int size) {

clock\_t start\_mult, end\_mult;

double time\_spent;

int i, j, r;

int elem\_c;

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

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

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

for (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));

}

srand(time(NULL));

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

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

A[i][j] = rand() % 10 + 1;

B[i][j] = rand() % 10 + 1;

}

}

start\_mult = clock(); // Начало умножения матриц

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\_mult = clock(); // Конец умножения матриц

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

printf(" | %-9d | %-16.2f |\n", size, time\_spent \* 1000);

printf(" +-----------+------------------+\n");

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

free(A[i]);

free(B[i]);

free(C[i]);

}

free(A);

free(B);

free(C);

}

**Приложение B**

**Листинг программы для задания 2**

#include <stdio.h>

#include <stdlib.h>

#include <time.h>

#define Size 150000

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

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

}

void shell(int\* items, int count, double\* time\_spent)

{

int i, j, gap, k;

int x, a[5];

clock\_t sort\_start, sort\_end;

a[0] = 9; a[1] = 5; a[2] = 3; a[3] = 2; a[4] = 1;

sort\_start = clock();

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;

}

}

sort\_end = clock();

\*time\_spent = ((double)(sort\_end - sort\_start)) / CLOCKS\_PER\_SEC \* 1000;

}

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);

}

void Fill\_rand(int\* mass, int size) {

srand(time(NULL));

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

mass[i] = rand() % 1000;

}

}

void Fill\_ascending(int\* mass, int size) {

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

mass[i] = i;

}

}

void Fill\_descending(int\* mass, int size) {

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

mass[i] = size - i - 1;

}

}

void Fill\_asc\_desc(int\* mass, int size) {

int half = size / 2;

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

mass[i] = i;

}

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

mass[i] = size - (i - half) - 1;

}

}

int main() {

printf(" +-----------+----------------+----------------+----------------+----------------+\n");

printf(" | Func/set | Random | Ascending | Descending | Asc-Desc |\n");

printf(" +-----------+----------------+----------------+----------------+----------------+\n");

int\* Mass = (int\*)malloc(Size \* sizeof(int));

clock\_t sort\_start, sort\_end;

double time\_spent\_rand, time\_spent\_asc, time\_spent\_desc, time\_spent\_asc\_desc;

// Shell sort

Fill\_rand(Mass, Size);

shell(Mass, Size, &time\_spent\_rand);

Fill\_ascending(Mass, Size);

shell(Mass, Size, &time\_spent\_asc);

Fill\_descending(Mass, Size);

shell(Mass, Size, &time\_spent\_desc);

Fill\_asc\_desc(Mass, Size);

shell(Mass, Size, &time\_spent\_asc\_desc);

printf(" | Shell | %-14.2f | %-14.2f | %-14.2f | %-14.2f |\n", time\_spent\_rand, time\_spent\_asc, time\_spent\_desc, time\_spent\_asc\_desc);

printf(" +-----------+----------------+----------------+----------------+----------------+\n");

// Quick sort

Fill\_rand(Mass, Size);

sort\_start = clock();

qs(Mass, 0, Size - 1);

sort\_end = clock();

time\_spent\_rand = ((double)(sort\_end - sort\_start)) / CLOCKS\_PER\_SEC \* 1000;

Fill\_ascending(Mass, Size);

sort\_start = clock();

qs(Mass, 0, Size - 1);

sort\_end = clock();

time\_spent\_asc = ((double)(sort\_end - sort\_start)) / CLOCKS\_PER\_SEC \* 1000;

Fill\_descending(Mass, Size);

sort\_start = clock();

qs(Mass, 0, Size - 1);

sort\_end = clock();

time\_spent\_desc = ((double)(sort\_end - sort\_start)) / CLOCKS\_PER\_SEC \* 1000;

Fill\_asc\_desc(Mass, Size);

sort\_start = clock();

qs(Mass, 0, Size - 1);

sort\_end = clock();

time\_spent\_asc\_desc = ((double)(sort\_end - sort\_start)) / CLOCKS\_PER\_SEC \* 1000;

printf(" | qs | %-14.2f | %-14.2f | %-14.2f | %-14.2f |\n", time\_spent\_rand, time\_spent\_asc, time\_spent\_desc, time\_spent\_asc\_desc);

printf(" +-----------+----------------+----------------+----------------+----------------+\n");

// default qsort

Fill\_rand(Mass, Size);

sort\_start = clock();

qsort(Mass, Size, sizeof(int), compare);

sort\_end = clock();

time\_spent\_rand = ((double)(sort\_end - sort\_start)) / CLOCKS\_PER\_SEC \* 1000;

Fill\_ascending(Mass, Size);

sort\_start = clock();

qsort(Mass, Size, sizeof(int), compare);

sort\_end = clock();

time\_spent\_asc = ((double)(sort\_end - sort\_start)) / CLOCKS\_PER\_SEC \* 1000;

Fill\_descending(Mass, Size);

sort\_start = clock();

qsort(Mass, Size, sizeof(int), compare);

sort\_end = clock();

time\_spent\_desc = ((double)(sort\_end - sort\_start)) / CLOCKS\_PER\_SEC \* 1000;

Fill\_asc\_desc(Mass, Size);

sort\_start = clock();

qsort(Mass, Size, sizeof(int), compare);

sort\_end = clock();

time\_spent\_asc\_desc = ((double)(sort\_end - sort\_start)) / CLOCKS\_PER\_SEC \* 1000;

printf(" | qsort | %-14.2f | %-14.2f | %-14.2f | %-14.2f |\n", time\_spent\_rand, time\_spent\_asc, time\_spent\_desc, time\_spent\_asc\_desc);

printf(" +-----------+----------------+----------------+----------------+----------------+\n");

free(Mass);

return 0;

}