****

|  |  |
| --- | --- |
|  | **Министерство науки и высшего образования Российской Федерации**  **Федеральное государственное бюджетное образовательное учреждение**  **высшего образования**  **«Московский государственный технический университет**  **имени Н.Э. Баумана**  **(национальный исследовательский университет)»**  **(МГТУ им. Н.Э. Баумана)** |

ФАКУЛЬТЕТ « Информатика и системы управления»

КАФЕДРА «Программное обеспечение ЭВМ и информационные технологии»

**ОТЧЕТ ПО ПРОЕКТНО-ТЕХНОЛОГИЧЕСКОЙ**

**ПРАКТИКЕ**

Группа ИУ7-22Б

Тип практики учебная

Название предприятия **НУК ИУ МГТУ имени Н. Э. Баумана**

|  |  |  |
| --- | --- | --- |
| Студент: Фам Минь Хиеу |  |  |
| Руководитель практики от предприятия |  | Ломовской Игорь Владимирович |
| Руководитель практики от МГТУ им. Н. Э. Баумана |  | Кострицкий Александр Сергеевич |

*2022 г.*

# Задание 1

**Цель работы**: В рамках данного задания было необходимо реализовать набор скриптов для автоматизации запуска функциональных тестов лабораторных работ по курсу «Программирование на Си».

**1.Скрипт: comparator\_1.sh**

*Назначение скрипта* : сравнение последовательностей действительных чисел, располагающихся в двух текстовых файлах, с игнорированием остального содержимого.

Листинг кода:

|  |
| --- |
| #!/bin/bash  trash="../data/trash"  ans1=  ans2=  content1=$(cat "$1")  content2=$(cat "$2")  for i in $content1; do      echo "$i" | grep -E "[-+]?[0-9]+$" > "$trash"      status=$?      if [ $status -eq 0 ]; then          ans1=$ans1" "$i      fi  done  for i in $content2; do      echo "$i" | grep -E "[-+]?[0-9]+$" > "$trash"      status=$?      if [ $status -eq 0 ]; then          ans2=$ans2" "$i      fi  done  if [ "$ans1" != "$ans2" ]; then      exit 1  fi  exit 0 |

**2.Скрипт: comparator\_2.sh**

*Назначение скрипта* : сравнение содержимого двух текстовых файлов, располагающегося после первого вхождения подстроки «Result: ».

Листинг кода:

|  |
| --- |
| #!/bin/bash  ct1=$(cat "$1")  ct2=$(cat "$2")  ans1=  ans2=  flag=0  for i in $ct1; do      if [ "$i" == "Result:" ]; then          flag=1      fi      if [ $flag -eq 1 ]; then          ans1=$ans1" "$i      fi  done  flag=0  for i in $ct2; do      if [ "$i" == "Result:" ]; then          flag=1      fi      if [ $flag -eq 1 ]; then          ans2=$ans2" "$i      fi  done  if [ "$ans1" != "$ans2" ]; then      exit 1  fi  exit 0 |

**3.Скрипт: pos\_case.sh**

*Назначение скрипта* : проверка позитивного тестового случая по определённым далее правилам

Листинг кода:

|  |
| --- |
| #!/bin/bash  if [ $# -lt 2 ]; then      echo "number of argument invalid"      exit 1  fi  data\_in=$1  data\_out=$2  args=$3  cmp\_file="../data/cmp\_file"  if [ ! -v $USE\_VALGRIND ]; then      valgrind ../../app.exe "$args"< "$data\_in" > "$cmp\_file"  else      ../../app.exe < "$data\_in" > "$cmp\_file"  fi  status=$?  if [ $status -ne 0 ]; then      exit 1  fi  bash ./comparator.sh "$cmp\_file" "$data\_out"  status=$?  if [ $status -eq 0 ]; then      exit 0  fi  exit 1 |

**4.Скрипт: neg\_case.sh**

*Назначение скрипта* : проверка негативного тестового случая по определённым далее правилам.

Листинг кода:

|  |
| --- |
| #!/bin/bash  if [ $# -lt 1 ]; then      echo "number of parameter invalid"      exit 1  fi  data\_in=$1  args=$2  out="../data/out"  if [ ! -v $USE\_VALGRIND ]; then      valgrind ../../app.exe "$args"< "$data\_in" > "$out"  else      ../../app.exe < "$data\_in" > "$out"  fi  status=$?  if [ $status -eq 0 ]; then      exit 1  fi  exit 0 |

**5.Скрипт: func\_test.sh**

*Назначение скрипта* : автоматизация функционального тестирования

Листинг кода:

|  |
| --- |
| #!/bin/bash  path="../data/"  count\_f\_pos=0  for file in "$path"pos\_??\_in.txt; do      count\_f\_pos=$((count\_f\_pos+1))  done  i=1  count\_invalid\_test=0  echo "Positive test: "  while [ $i -le $count\_f\_pos ]; do      file\_in="../data/pos\_0${i}\_in.txt"      file\_out="../data/pos\_0${i}\_out.txt"      args="../data/pos\_0${i}\_args.txt"      bash ./pos\_case.sh "$file\_in" "$file\_out" "$args"      status=$?      echo -n "TEST $i: "      if [ $status -eq 0 ]; then          echo "PASS"      else          echo "FAILED"          count\_invalid\_test=$((count\_invalid\_test+1))      fi      i=$((i+1))  done  count\_f\_neg=0  for file in "$path"neg\_??\_in.txt; do      echo "$file" > /dev/null      count\_f\_neg=$((count\_f\_neg + 1))  done  echo "Negative test: "  j=1  while [ $j -le $count\_f\_neg ]; do      file\_in="../data/neg\_0${j}\_in.txt"      args="../data/pos\_0${j}\_args.txt"      bash ./neg\_case.sh "$file\_in" "$args"      status=$?      echo -n "TEST $j: "      if [ $status -eq 0 ]; then          echo "PASS"      else          echo "FAILED"          count\_invalid\_test=$((count\_invalid\_test+1))      fi      j=$((j+1))  done  exit $count\_invalid\_test |

**Вывод:** Реализованные скрипты помогли мне в написании тестовых данных лабораторных работ

# Задание 2

**Цель работы**: В рамках данного задания нужно было изучить «классические» этапы получения исполняемого файла и сравнить их с теми шагами, которые выполняют компиляторы gcc и clang.

**A1. Обработка препроцессором: cpp -o main.i main.c**

Препроцессор выполняет следующие действия:

* удаление комментариев;
* вставку файлов (директива include);
* текстовые замены (по-другому говорят - раскрытие макросов, директива define);
* условную компиляцию (директива if).

|  |
| --- |
| ...  extern int printf (const char \*\_\_restrict \_\_format, ...);  ...  # 7 "main.c"  int main(void)  {  int x = 52;  printf("year has %d weeks\n", x);  return 0; |

**A2. Трансляция на язык ассемблера: с99 -S -fverbose-asm -masm=intel main.i**

Файл, полученный препроцессором, передается на вход транслятору с99, который переводит его с языка Си на язык ассемблера.

|  |
| --- |
| .file "main.c"  ...  .text  .section .rodata  .LC0:  .string "year has %d weeks\n"  .text  .globl main  .type main, @function  main:  .LFB0:  .cfi\_startproc  endbr64  push rbp #  .cfi\_def\_cfa\_offset 16  .cfi\_offset 6, -16  mov rbp, rsp #,  .cfi\_def\_cfa\_register 6  sub rsp, 16... |

**A3. Ассемблирование в объектный файл: as main.s -o main.o**

**objdump -drw -Mintel main.o**

**nm main.o**

С языка ассемблера программа переводится в машинный код с помощью транслятора as

|  |
| --- |
| 0000000000000000 <main>:  0: f3 0f 1e fa endbr64  4: 55 push rbp  5: 48 89 e5 mov rbp,rsp  8: 48 83 ec 10 sub rsp,0x10  c: c7 45 fc 34 00 00 00 mov DWORD PTR [rbp-0x4],0x34  13: 8b 45 fc mov eax,DWORD PTR [rbp-0x4]  16: 89 c6 mov esi,eax  18: 48 8d 3d 00 00 00 00 lea rdi,[rip+0x0] # 1f <main+0x1f> 1b: R\_X86\_64\_PC32 .rodata-0x4  1f: b8 00 00 00 00 mov eax,0x0  24: e8 00 00 00 00 call 29 <main+0x29> 25: R\_X86\_64\_PLT32 printf-0x4  29: b8 00 00 00 00 mov eax,0x0  2e: c9 leave  2f: c3 ret |
| U \_GLOBAL\_OFFSET\_TABLE\_  0000000000000000 T main  U printf |

**A4. Компоновка: gcc -o main.exe main.o**

В процессе получения исполняемого файла компоновщик решает несколько задач

* объединяет несколько объектных файлов в единый исполняемый файл;
* выполняет связывание переменных и функций, которые требуются - очередному объектному файлу, но находятся где-то в другом месте;
* добавляет специальный код, который подготавливает окружение для вызова функции main, а после ее завершения выполняет обратные действия.

|  |
| --- |
| hieubmstu@hieubmstu-VirtualBox:~/task2$ gcc -o main.exe main.o  hieubmstu@hieubmstu-VirtualBox:~/task2$ ./main.exe  year has 52 weeks  hieubmstu@hieubmstu-VirtualBox:~/task2$ |

**Вывод:** После выполнения этого задания я узнал процесс получения исполняемого файла и организации объектных и исполняемых файлов.

# Задание 3-4-5-6

**Цель работы**: В рамках данных заданий нужно было изучить нахождение с помощью gdb ошибки в приложенных программах, представление в памяти многомерного статического массива, представление в памяти строк и массивов строк, выравнивание переменных, представление в памяти структур.

**1. Основные вопросы**

1. С ключом -g можно скомпилировать программу, чтобы можно было пользоваться отладчиком gdb.

Если собрать программу без этого ключа, и загрузить её в gdb, сообщение прошло, что нет отладочной информации из app.exe

2. запустить программу под отладчиком с помощью команды gdb. Чтобы досрочно завершить её работ, используем команду quit

3. Чтобы посмотреть в каком месте программы мы остановились, восспользуем команду layout next

4. С помощью команды print можно посмотреть значение переменной, с помощью команды set var можно изменить значение переменной

|  |
| --- |
| (gdb) print a $3 = 5 (gdb) set var a=10 (gdb) print a $4 = 10 |

5. С помощью команд next or step можно выполнить программу в пошаговом режиме.

* next выполняет вызов функции и не переходит в эту функцию
* step заходит в функцию

6. Чтобы понять, какая последовательность вызовов функций привела нас сюда, мы восспользуем команду disas

7. Чтобы установить точку останова, мы используем:

* break [имя\_файла:]номер\_строки
* break имя\_функции
* break
* tbreak (вр*е*менная точка останова)

8. Временная точка останова – это точка, которая удаляется сразу, как только достигнута

9. Для включения: enable [номер|диапазон]

Для выключения: disable [номер|диапазон]

пропустить некоторое количество срабатываний: ignore номер количество\_итераций

10. Задать условие остановки на точке основа:

break позиция if условие

Позиция – номер строки или имя функции.

11. Эти точки различаются в том, что точка наблюдения позволяет получать уведомления о доступах для чтения и записи к переменным.

12.

|  |
| --- |
| #include <stdio.h>  #define N 5  **int** main()  {  printf("Calculate factorial of %d", N);  **int** i = 1, temp = 1;  **while** (i <=N )  {  temp \*= i;  }  **return** 0;  } |

Мы установим точку наблюдения watch i, watch temp, чтобы проверить изменяются ли эти переменные

13. Чтобы посмотреть содержимое блоков памяти восспользуем:

x [/nfu] [адрес]

n – сколько единиц памяти должно быть выведено

f – спецификатор формата

u – размер выводимой единицы памяти

**2. Представление многомерного статического массива в памяти**

|  |
| --- |
| #include <stdio.h>  int main(void)  {      int a[2][3][4];      int count = 1;      for (int i = 0; i < 2; i++)          for (int j = 0; j < 3; j++)              for (int k = 0; k < 4; k++)              {                  a[i][j][k] = count;                  count ++;              }      printf("array was initialized\n");      return 0;  } |

Дамп памяти, который содержит этот массив полностью.

|  |
| --- |
| (gdb) x /96xb &a 0x7fffffffddf0: 0x01 0x00 0x00 0x00 0x05 0x00 0x00 0x00 0x7fffffffddf8: 0x04 0x00 0x00 0x00 0x02 0x00 0x00 0x00 0x7fffffffde00: 0x07 0x00 0x00 0x00 0x03 0x00 0x00 0x00 0x7fffffffde08: 0x0a 0x00 0x00 0x00 0x08 0x00 0x00 0x00 0x7fffffffde10: 0x09 0x00 0x00 0x00 0xfe 0xff 0xff 0xff 0x7fffffffde18: 0x05 0x00 0x00 0x00 0x04 0x00 0x00 0x00 0x7fffffffde20: 0x0b 0x00 0x00 0x00 0x03 0x00 0x00 0x00 0x7fffffffde28: 0x02 0x00 0x00 0x00 0x06 0x00 0x00 0x00 0x7fffffffde30: 0x01 0x00 0x00 0x00 0x04 0x00 0x00 0x00 0x7fffffffde38: 0x07 0x00 0x00 0x00 0x09 0x00 0x00 0x00 0x7fffffffde40: 0x06 0x00 0x00 0x00 0xfb 0xff 0xff 0xff 0x7fffffffde48: 0xfd 0xff 0xff 0xff 0x04 0x00 0x00 0x00 |

Мы видим, что компилятор Си располагает строки матрицы a в памяти одну за другой вплотную друг к другу.

**3. Представление массив строк в памяти**

**Способ 1: двумерный массив строк**

|  |
| --- |
| char arr\_1[][12] = {"Moscow", "Petersburg", "Vladivostok"}; |

|  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| M | o | s | c | o | w | \0 | \0 | \0 | \0 | \0 | \0 |
| P | e | t | e | r | s | b | u | r | g | \0 | \0 |
| V | l | a | d | i | v | o | s | t | o | k | \0 |

Представление двухмерного массива строк в памяти

|  |
| --- |
| (gdb) x /36db &arr\_1 0x7fffffffde20: 77 111 115 99 111 119 0 0 0x7fffffffde28: 0 0 0 0 80 101 116 101 0x7fffffffde30: 114 98 117 114 103 0 0 0 0x7fffffffde38: 86  108 97 100 105 118  111 115 0x7fffffffde40: 116 111 107 0 |

Полезные байты, быйты – которые представлены символы строки и заканчивающий символ \0, остальные являются вспомагательными

**Способ 2: массив указателей на строки (“ragged array”)**

|  |
| --- |
| char \*arr\_2[] = {"Moscow", "Petersburg", "Vladivostok"}; |

|  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
| [0] | ==> | M | o | s | c | o | w | \0 |
| [1] | ==> | P | e | t | e | r | s | b | u | r | g | \0 |
| [2] | ==> | V | l | a | d | i | v | o | s | t | o | k | \0 |

Теоретический расчет : p = (7 + 11 + 12) + 8 x 3 = 54 bytes.

Всего 30 байтов используются для записи всех символов и нулевых символов (полезные)

24 остальные байты выделяются для хранения указателей (вспомогательные)

**3.** [**Выравнивание переменных. Представление в памяти структур**](https://e-learning.bmstu.ru/iu7/mod/assign/view.php?id=853)

|  |
| --- |
| #include <stdio.h>  int main(void)  {      int i;      char c;      double d;      short h;      printf("variables were initialized\n");      return 0;  } |

Локальные переменные располагаются кратны размеру в памяти.

|  |  |  |
| --- | --- | --- |
| Имя переменной | Размер | Значение адреса |
| i | 4 | 000000000062FE1C |
| c | 1 | 000000000062FE1B |
| d | 8 | 000000000062FE10 |
| h | 2 | 000000000062FE0E |

**Вывод**: gdb – хороший полезный утилит, c помощью которого мы узнаем как программа работает по шагам, представление переменных данных в памяти.

# Задание 7

**Цель работы**: Определить времени выполнения программы и сравнить производительности работы программы по двум плоскостям с помощью утилита gnuplot.

**График зависимости времени выполнения программы от количества элементов массива**

*Замечание*: функция find\_1 : использование операции индексации a[i]

функция find\_2 : использование выражение \*(a+ i)

функция find\_3 : использование указателей

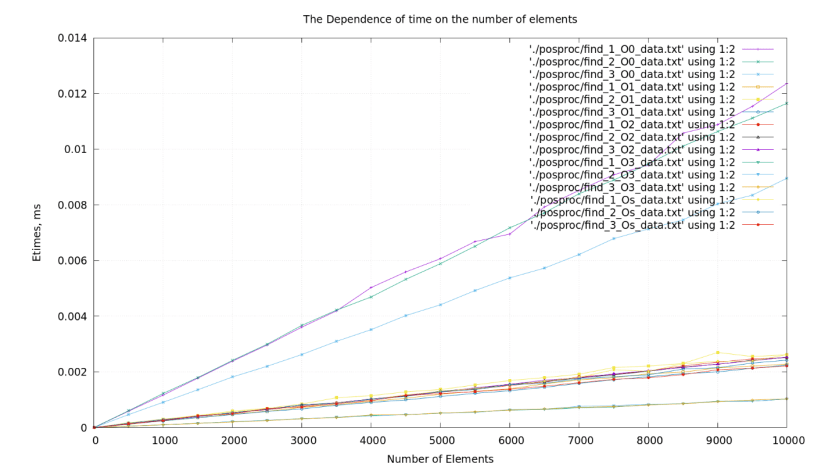


Рис 1. Кусочно-линейный график

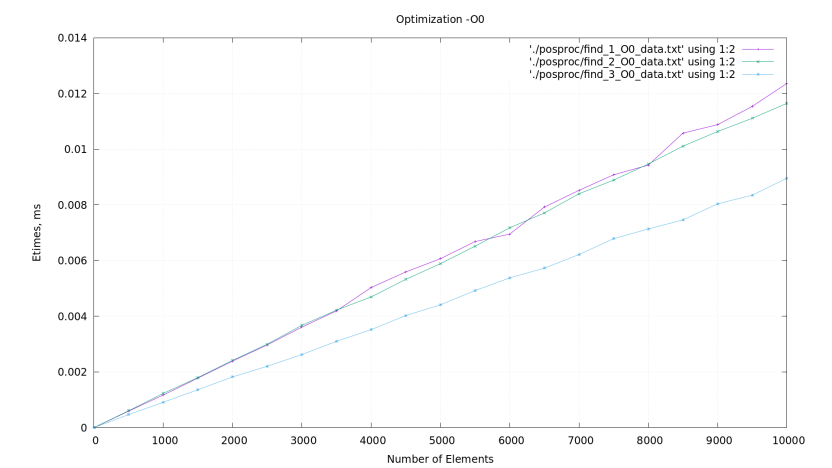


Рис 3: Уровень оптимизации -O0

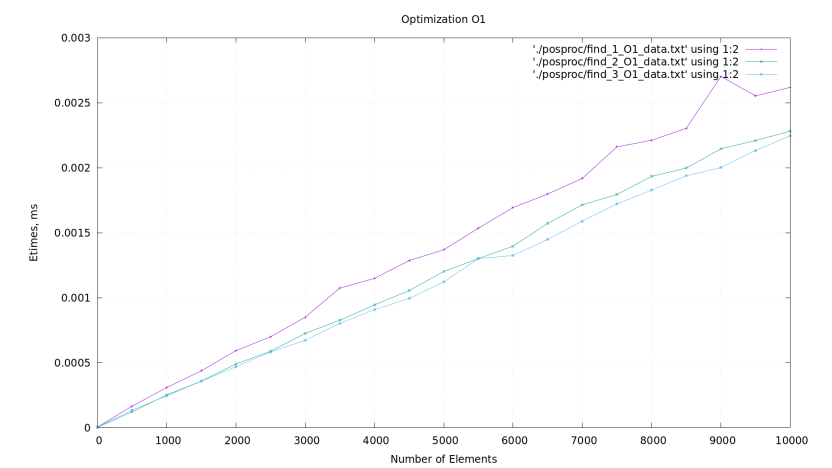


Рис 3: Уровень оптимизации -O1

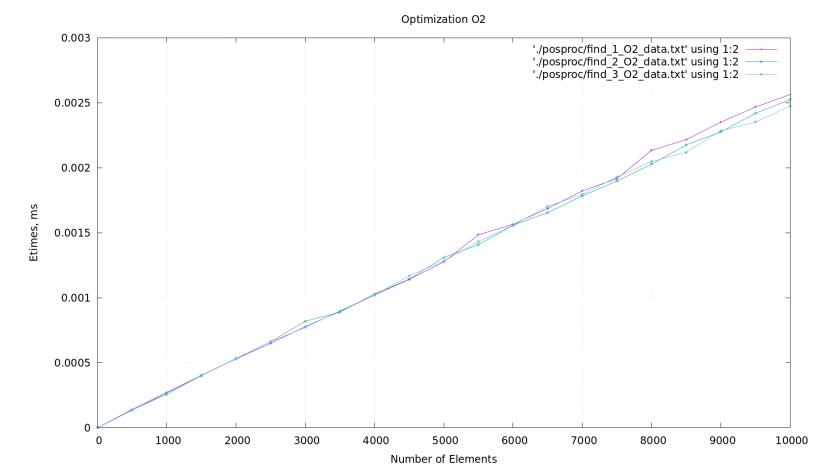


Рис 4: Уровень оптимизации -O2

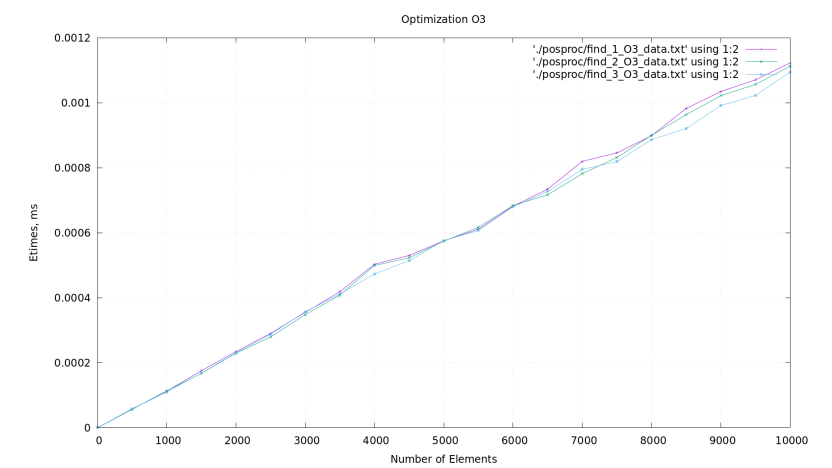
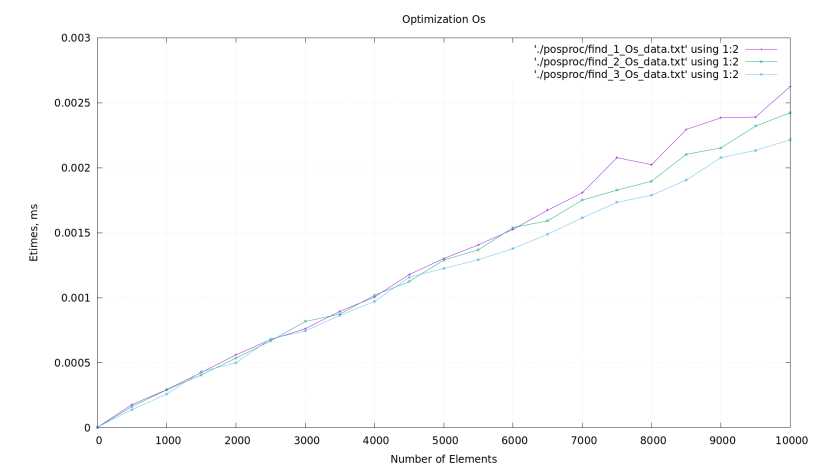


Рис 5: Уровень оптимизации -O3

 Рис 6: Уровень оптимизации -Os

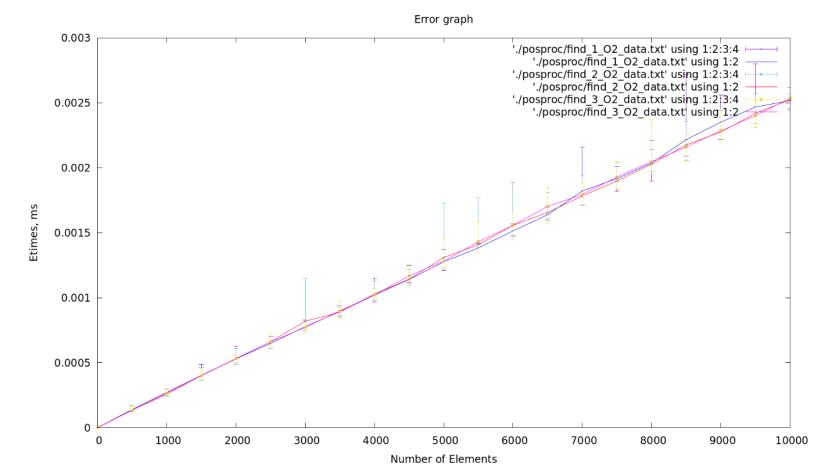


Рис 7: График с ошибкой для всех вариантов при -O2

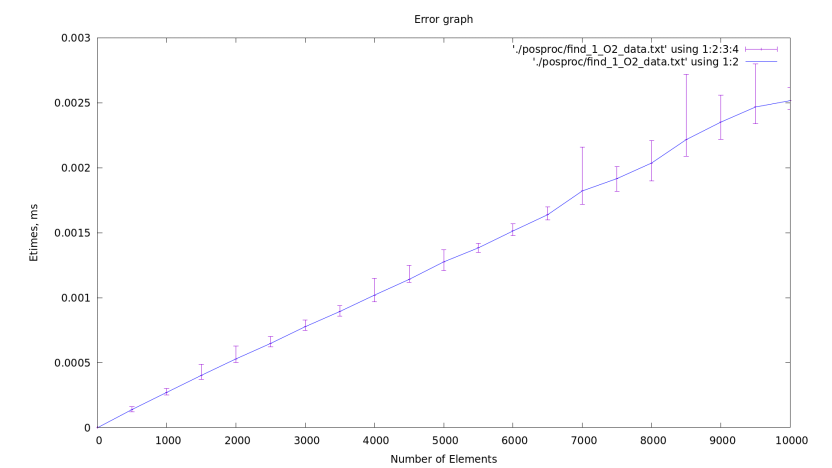


Рис 8: График с ошибкой для find\_1 при -O2

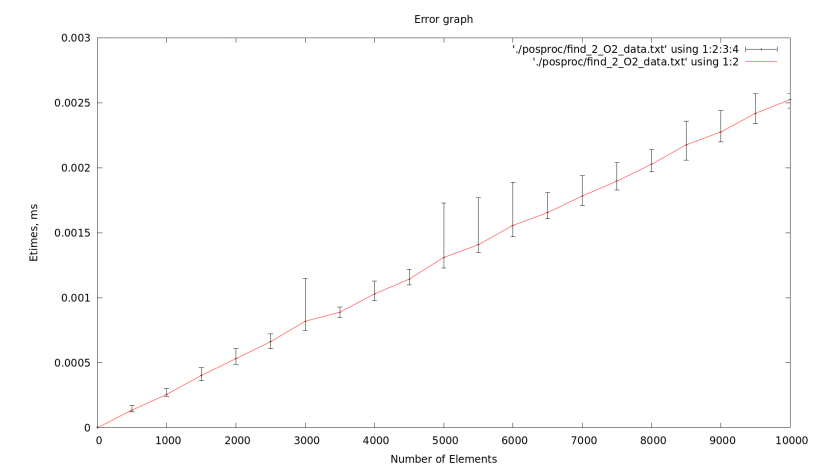


Рис 9: График с ошибкой для find\_2 при -O2

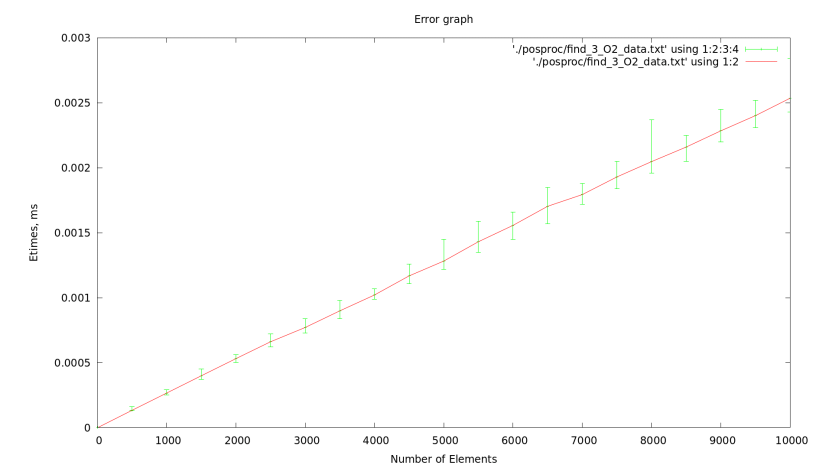


Рис 10: График с ошибкой для find\_3 при -O2

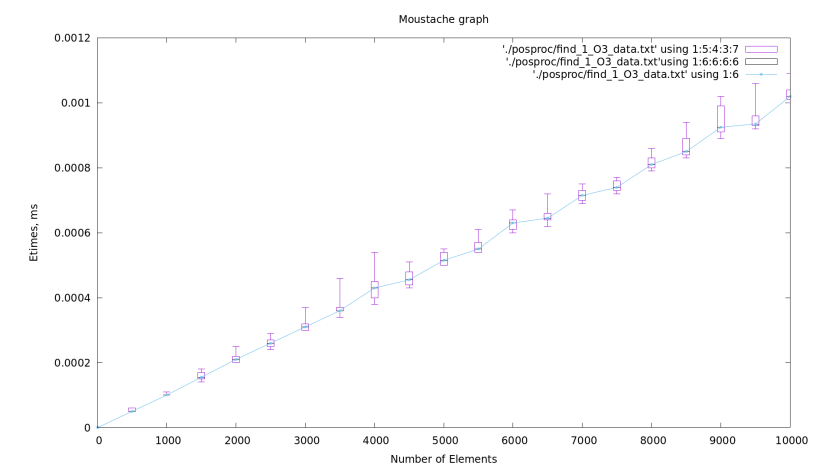


Рис 3: График с усами для find\_1 при -O3

**Выводы:** Время выполнения программы методов в порядке убывания. Метод обработки с использованием указателей самый быстрый. Обработке с использованием оператор индексации a[i] самый медленный. При реализации на компьютере первый способ приводится ко второму, т.е индексное выражение преобразуется к адресному. На пример a[i] преобразуется в \*(a + i). Мы видим что, первый и второй способы сначала требуются выполнять оператор сложение, а потом выполнять оператор разыменования. Обработка массива с помощью указателей требуется выполнять только оператор разыменования. Именно поэтому, время выпонения третьего способа намного быстрее.

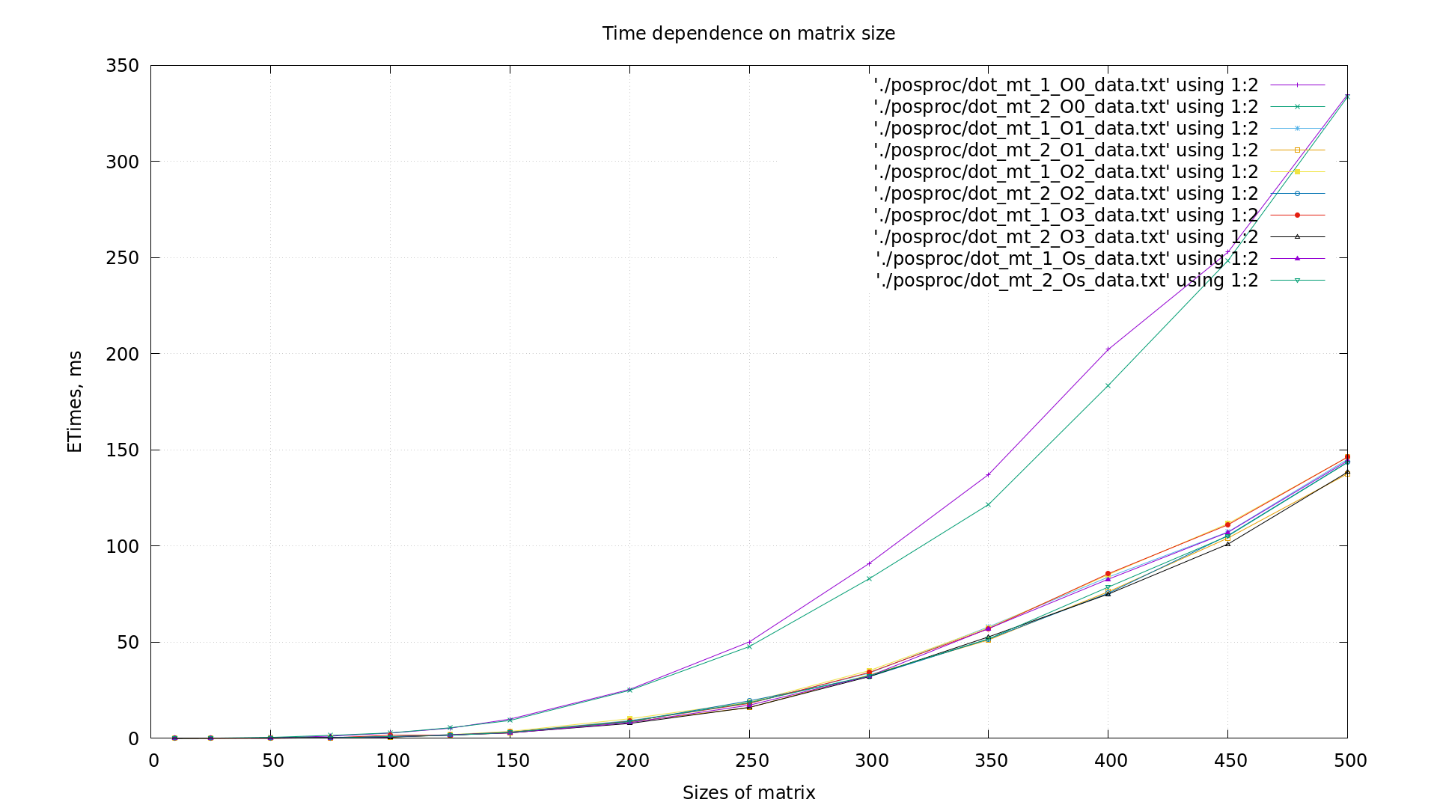
# Задание 8

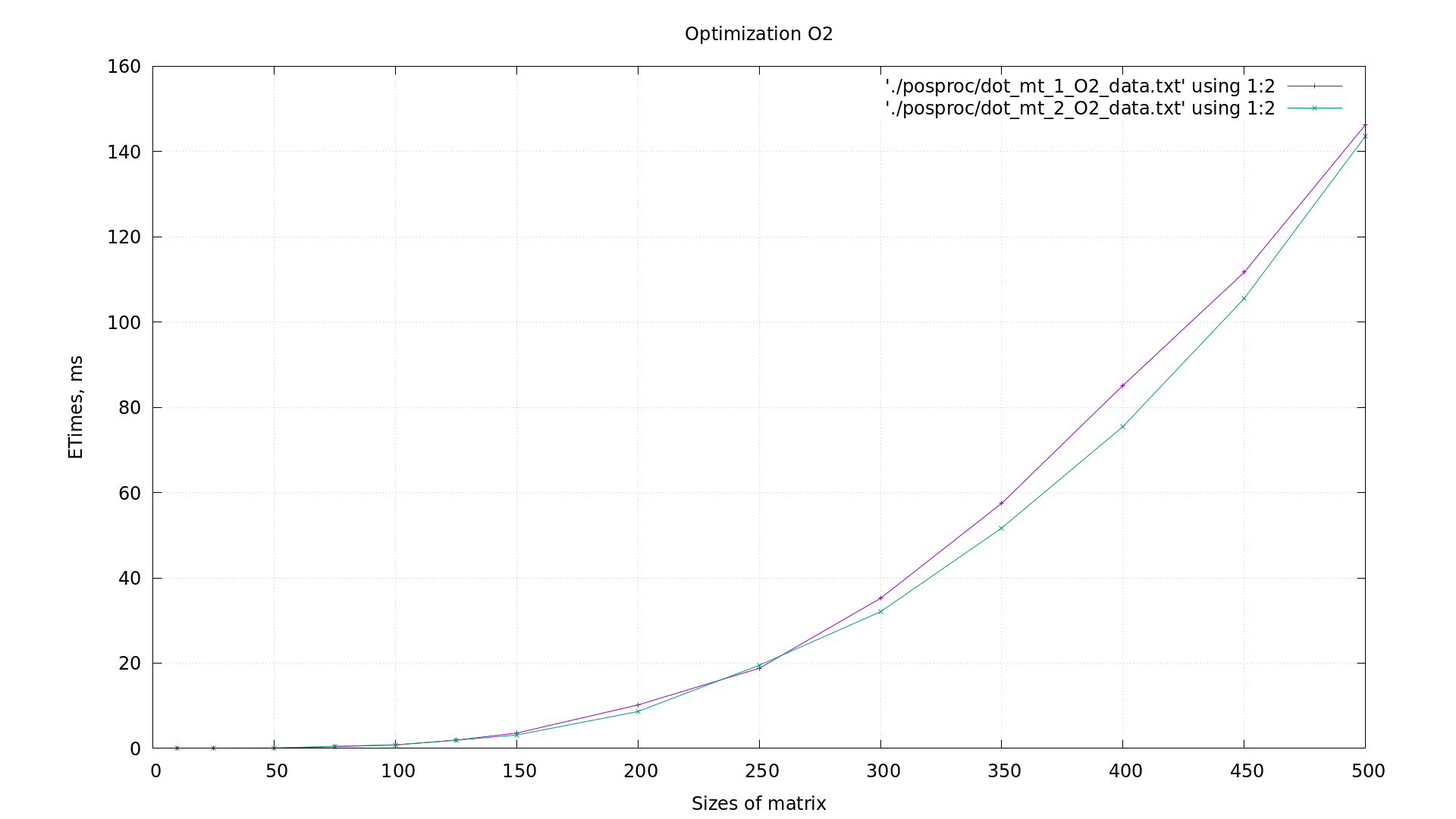
**Цель работы**: постановка времени выполнения программы, нарисовать графики с помощью утилита gnuplot.

**Вариант 1: Умножение квадратных матриц с транспонированием и без**

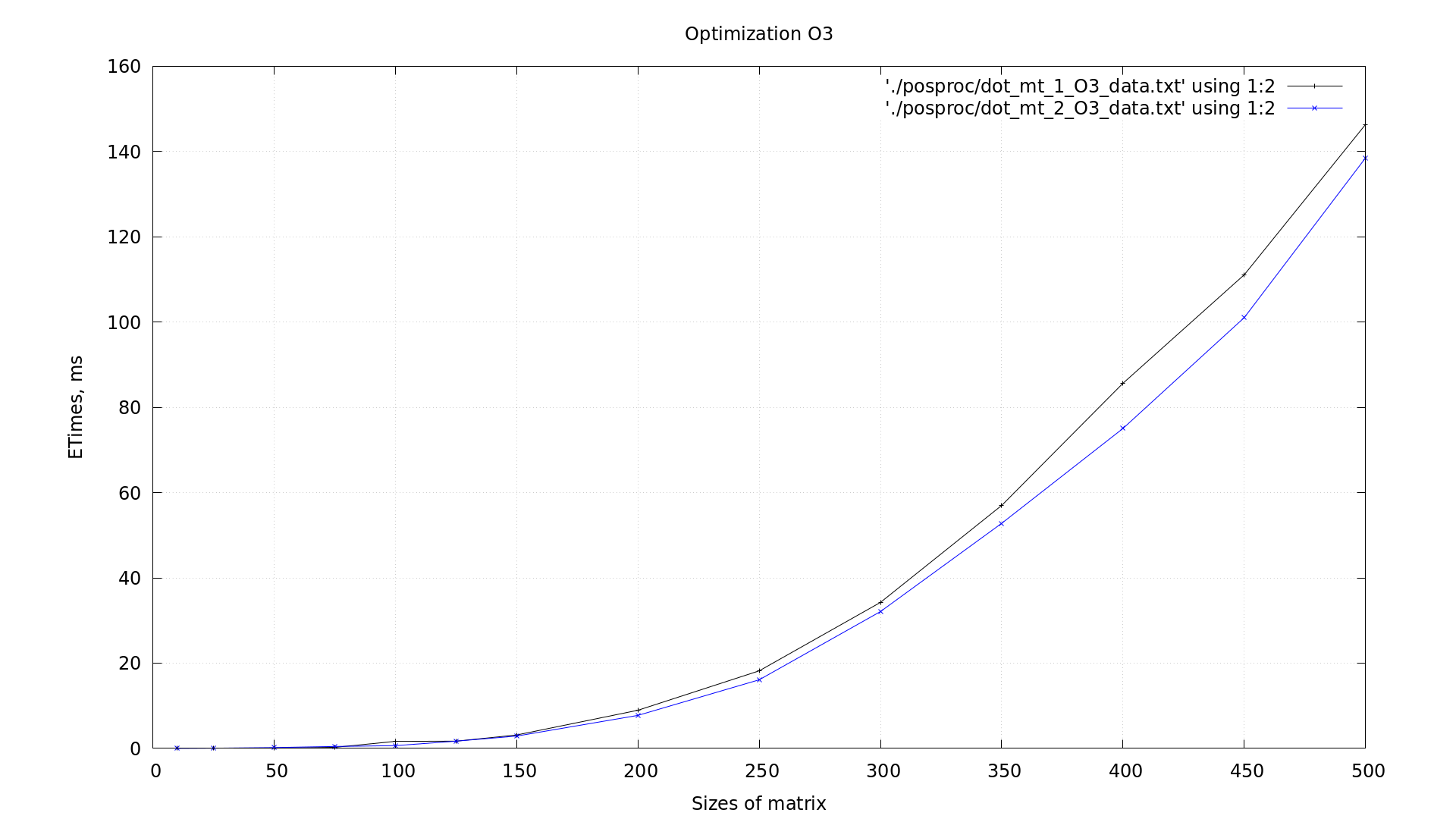
**Замечание:**

|  |
| --- |
| *Функция dot\_mt\_1: Умножение квадратных матриц без транспонирования*  *Функция dot\_mt\_2: Умножение квадратных матриц с транспонированием* |

 *Рис 1: Кусочно-линейный график*

**

*Рис 2: При уровне оптимизации -O2*



*Рис 3: При уровне оптимизации -O3*

**Выводы:** Из графика мы видим, что при всех уровнях оптимизации, умножение с траспонированием работает чуть побыстрее. Компилятор располагает строки матрицы в памяти одну за другой вплотную друг к другу, поэтому при умножении с транспонированием, доступ к элементу почти сразу, а в оборотном случае ( без транспонировании ) процесс чтения медленнее.