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 ИУ7-21Б

Тип практики учебная

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

Студент	Краснов Леонид Антонович
Руководитель практики от предприятия	Ломовской Игорь Владимирович
Руководитель практики от МГТУ им. Н. Э. Баумана	Кострицкий Александр Сергеевич

2022 г.

Оглавление

Автоматизация тестирования (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».

Проделанная работа

Идея автоматизации заключается в использовании перенаправления ввода/вывода. Данные для тестирования каждой лабораторной работы подготавливаются заранее. Позитивный (pos_case.sh) и негативный (neg_case.sh) сценарии тестирования реализуются раздельно. За «обход» тестовых данных и вызов позитивного и негативного сценариев отвечает скрипт func_tests.sh. Сравнение полученного результата работы программы и эталонного значения выполняется с помощью компаратора (comparator.sh). Было реализовано два универсальных компаратора:

- Компаратор, который сравнивает численный вывод программы с эталонным файлом, подготовленным пользователем, игнорируя символы.
- Компаратор, который сравнивает символьный вывод программы с эталонным файлом, подготовленным пользователем, игнорируя все до ключевого слова “Result: “.

Так же был реализован скрипт отладочной сборки (*build_debug.sh*), который собирает программу с отладочной информацией с помощью ключа “-g”, скрипт релизной сборки (*build_realise.sh*), который собирает программу с ключами и дает имя исполняемому файлу “app.exe” и скрипт для удаления мусора, который появляется в результате работы других скриптов (*clean.sh*).

Вывод

Реализованные скрипты помогли мне в написании лабораторных работ с 3-й до 5-ой, тем, что тестирование программы проходит намного быстрее, вместо того что бы каждый раз вводить данные в программу и запускать её, нужно только подготовить эталонные данные в почти неограниченном количестве и запустить скрипт.

Изучение этапов компиляции и строения исполняемого файла (задание 2)

Цель работы

Изучение этапов компиляции и строения исполняемого файла.

Проделанная работа

В рамках данного задания необходимо было изучить строение объектных и исполняемых файлов. Для этого использовались утилиты gcc, cpp, c99, as, ld.
Объектный (и исполняемый) файл состоит из секций:

- text
- data
- bss
- rodata
- comment
- note.GNU-stack
- eh_frame
- debug, если программа была собрана с отладочной информацией.

Был изучен процесс создания исполняемого файла. Исполняемый файл создается в 4 этапа:

- Обработка предпроцессором – из программы удаляются комментарии, заменяются директивы и мертвый код. Действие выполняется с помощью cpp.
- Трансляция на языке ассемблера – код программы переводится с языка Си на. Действие выполняется с помощью утилиты c99.
- Ассемблирование в объектный файл – из кода на Ассемблере создается объектный файл программы. Действие выполняется с помощью as.
- Компоновка – из объектного файла создается исполняемый файл. Действие выполняется с помощью ld.

Программа gcc выполняет эти 4 действия и поддерживает множество ключей, благодаря этому можно отключить удаление промежуточных вариантов сборки программы и выводить в консоль вызываемые программы с помощью ключей *v* и *save-temps*.
Для изучения содержимого объектного файла используется утилита objdump, а для сравнения дизассемблированных файлов используется утилита diff, которая выводит на консоль отличия, если отличий нет, значит файлы не отличаются.

Вывод

Видимых отличий между промежуточными файлами, появившихся при процессе создания исполняемого файла и файлов при ручной сборке, нет. Разница между двумя объектными файлами с отладочной информацией и без заключается в том, что добавляется секция debug в объектный и исполняемый файл с отладочной информацией, а остальное содержимое одинаково, следовательно, объектный файл без отладочной информации

меньше по размеру, чем объектный файл с отладочной информацией. Исполняемы файл с отладочной информацией больше по размеру, чем исполняемый файл без отладочной информации. Расположение функций, локальных и глобальных переменных осталось неизменным.

Отладка (задания 3 - 6)

Цель работы

Целью работы является научиться самостоятельно производить трассировку приложения. Изучить работу с отладчиком gdb. Изучить как в памяти представлен многомерный статический массив. Изучить, как в памяти представлена строка языка Си. Понять, какие байты за что отвечают. Изучить разные способы хранения массива слов в языке Си. Изучить как располагаются в памяти локальные переменные. Изучить как в памяти представлены структуры.

Проделанная работа

Задание 3

В рамках данного задания необходимо было помощью отладчика gdb отладить 3 программы task_01.c, task_02.c, task_03.c.

Все программы отлаживаются по одному алгоритму:

1. Собрать исполняемый файл и запустить его, чтобы убедится, что программа работает неверно.
2. Собрать исполняемый файл с отладочной информацией и запустить его под gdb.
3. С помощью gdb проверить как вводимые значения и какие значения попадают в функцию.
4. Проверить какие значения возвращает функция.
5. Проверить как происходит вывод результата работы программы.
6. Запустить программу и проверить правильность ее работы

Для сборки исполняемого файла используется утилита gcc. Для отладки используется утилита gdb.

была составлена таблицу размеров типов char, int, unsigned, long long, short, int32_t, int64_t на двух разных машинах.

Придуман пример программы с ошибкой, для нахождения которой удобно использовать точку наблюдения. Показано, как в памяти представлены переменные типов char, int, unsigned, long long. Рассмотрены как положительные значения этих переменных, так и отрицательные. Результаты пояснены.

Показано, как в памяти представлен массив целых чисел.

Продемонстрированы особенности выполнения операции сложения указателя с целым числом на примере массива.

Изучена работа с отладчиком в Qt Creator. Написана инструкцию, которая сопоставляет команды gdb и аналогичные действия в Qt Creator.

Задание 4

В рамках данного задания необходимо было описать трехмерный массив. С помощью gdb вывести на экран и изучить дамп памяти, который содержит этот массив полностью.

Было выяснено, что трехмерный массив состоит из двумерных массивов. Компонент 3-х мерного массива – двумерный массив. Двумерный массив

состоит из одномерных массивов. Компонент 2-х мерного массива – одномерный массив. Одномерный целочисленный массив состоит из целых чисел. Компонент одномерного массива – целое число. Найдены размеры всех компонентов трехмерного массива.

Задание 5

В рамках данного задания необходимо было показать дамп памяти, который содержит массив строк, сохраненный первым способом, показать дамп памяти, который содержит массив строк, сохраненный в памяти вторым способом. Прокомментировать, что есть что в этих дампах памяти. Рассчитать суммарный размер памяти, который занимает каждая структура данных.

Задание 6

В рамках данного задания необходимо было выяснить, как располагаются локальные переменные в памяти. Показать на дампе памяти, как переменные располагаются в памяти. Найти размеры объявленных переменных. Составить таблицу, в которой указывается имя переменной, ее размер и значение адреса, по которому располагается переменная. Проанализирована зависимость значения адреса переменной от её размера. Описать структуру, содержащую несколько полей разного типа. Продемонстрировать дамп памяти, который содержит эту структуру. На дампе показать расположение каждого поля структуры. Составить таблицу, в которой содержится имя поля, его размер, значение адреса, по которому располагается поле. Проанализировать зависимость значения адреса поля от его размера. Произвести упаковку структуры и выполнить предыдущие пункты для упакованной структуры. Переставить поля структуры, так, чтобы занимаемое структурой место стало минимальным. Выяснить размер структуры.

Выводы

задание 3

Для того, чтобы можно было использовать gdb, в исполняемый файл нужно добавить отладочную информацию с помощью ключа -g. Если ключ не добавить появится сообщение от gdb.
(gcc -Werror -g main.c -o app.exe) затем запустить gdb (gdb ./app.exe). Что бы досрочно завершить работу используется команда quit. Когда выполнение программы прерывается на точке остановы gdb автоматически выводит строку, ее номер и номер точки останова в случае остановки. Что бы посмотреть значения переменных используется команда info locals. Что бы изменить значение переменной используется команда set (set var [имя_переменной] = [значение_переменной]). С помощью команды next можно выполнять программу в пошаговом режиме, однако не осуществляются заходы внутрь функций. С помощью команды step можно выполнять программу в пошаговом режиме, с заходом в функции. Команда backtrace выводит цепочку вызовов стека это можно использовать, когда программа остановилась на точке остановы и необходимо узнать какая

цепочка вызовов функции привела к этой точке остановы. Точку остановы можно установить с помощью команды break. Если после break ввести номер строки, то точка остановы будет поставлена на соответствующую строку, так же если после break написать имя функции, то точка остановы будет поставлена в начало этой функции. Временная точка остановы – это такая точка остановы, которая удаляется после первой активации (tbreak). Что бы выключить/включить точку останова нужно использовать команду disable/enable [номер/диапазон]. Что бы пропустить некоторое количество срабатываний используется команда ignore [номер] [количество_итераций]. Что бы задать условие остановки используется (break позиция if условие). Точка наблюдения отличается от точки остановы, тем что точки наблюдения позволяют получать уведомления, когда происходят изменения в памяти. То есть точка наблюдения останавливает программу, когда меняются данные, а точка остановы, когда выполнение программы достигнет определенной строчки кода. Точку наблюдения удобно использовать во вложенных или больших циклах, например есть несколько вложенных циклов for, и некоторые переменные,
которые меняются в этих циклах, нужно остановить программу, когда изменяемые переменные, например равны или отличаются, или вообще просто изменилась некоторая переменная или элемент массива. Так же точки наблюдения удобно использовать при изменении переменных в программе через указатель и при рекурсии. Что бы посмотреть содержимое области памяти используется команда x[/nfu] [адрес]
где n - сколько единиц памяти должно быть выведено
f - спецификатор формата
u – размер выводимой единицы памяти
Интерфейс отладчика в Qt Creator позволяет пользователю выполнять отладку программы с помощью тех же инструментов, что и в gdb. Переменные, имеющие отрицательное значения записываются в памяти не так как переменные с положительным, так как к значению переменной дописывается допкод, чтобы показать отрицательность переменной. При сложении указателя на массив с целым числом, указатель меняет значение на указатель на элемент массива по индексу целого числа, которое было прибавлено к указателю.

Задание 4

Массив располагается в памяти последовательно, компонент за компонентом. Трехмерный массив состоит из двумерных массивов, двумерный массив состоит из одномерных массивов, одномерный массив состоит из элементов. Размер в памяти трехмерного массива равен количеству двумерных массивов в трехмерном * на количество одномерных массивов в двумерном * на количество элементов в одномерном массиве * на объем памяти, который занимает тип элемента одномерного массива.

Массивы разной размерности передаются в функцию через указатель, имеющий соответствующую размерность.

Задание 5

В первом способе хранения массива строк, на дампе памяти строки расположены последовательно, и каждый байт – целочисленное число, которое обозначает символ, однако для создания массива строк, каждая строка имеет фиксированную длину, из-за чего недостающие символы заменяются нулем. Во втором случае, строки имеют не фиксированную длину, из-за чего в каждой строке есть только 1 ноль, который используется как символ конца строки.

Размер структуры соответствует сумме размеров типов данных, которые она содержит + выравнивание. Отсюда можно сделать вывод, что полезные данные в структуре занимают все пространство кроме выравнивания.

Задание 6

Судя по адресам переменных в дампе памяти видно, что они располагаются друг за другом. Значение адреса переменной напрямую зависит от ее размера, размера соседней переменной и адреса соседней переменной. В структурах, если выравнивание выключено, то адрес переменной отличается от адреса соседней переменной на свой размер, если же выравнивание включено, то значение адреса переменной отличается от соседней на свой размер + количество байт, которые ушли на выравнивание. Выравнивание происходит по большему размеру типа поля структуры. Поля структуры располагаются в памяти друг за другом, где каждый следующий адрес – предыдущий адрес + размер типа + (если есть) байты для выравнивания.

У упакованной структуры на дампе памяти видно, что поля располагаются в памяти строго друг за другом, где каждый следующий адрес – предыдущий адрес + размер типа. Переменная структурного типа располагается по тому же адресу, что и первое поле структуры. Для того чтобы понять есть ли у структуры выравнивание, надо сравнить размер структуры с суммой размеров ее полей. Если результат и будет величиной выравнивания. Если результат равен нулю, то выравнивания нет.

Замерный эксперимент (задания 7 - 8)

Цель работы

Провести сравнение производительности работы программы по двум плоскостям – разные способы доступа к элементу массива и разные уровни оптимизации. Произвести сравнение производительности программы по двум плоскостям – работа с матрицей с кешированием промежуточных результатов и работа с матрицей без кеширования промежуточных результатов, разные уровни оптимизации.

Проделанная работа

Задание 7

В рамках данного задания необходимо было выполнить подготовку программы к тестированию. Обеспечить автоматическое заполнение массива элементами. Вставить в программу функцию для нахождения микросекунд. Добавить возможность передавать количество элементов в массиве при запуске программы. Написать три различные версии программы, которые отличаются друг от друга методами доступа к элементу массива:

- использование операции индексации a[i]. Программа program_1.c
- формальная замена операции индексации на выражение *(a + i). Программа: program_02.c
- использование указателей для работы с массивом. Программа program_3.c

Каждую из трех программ собрана с разным уровнем оптимизации:

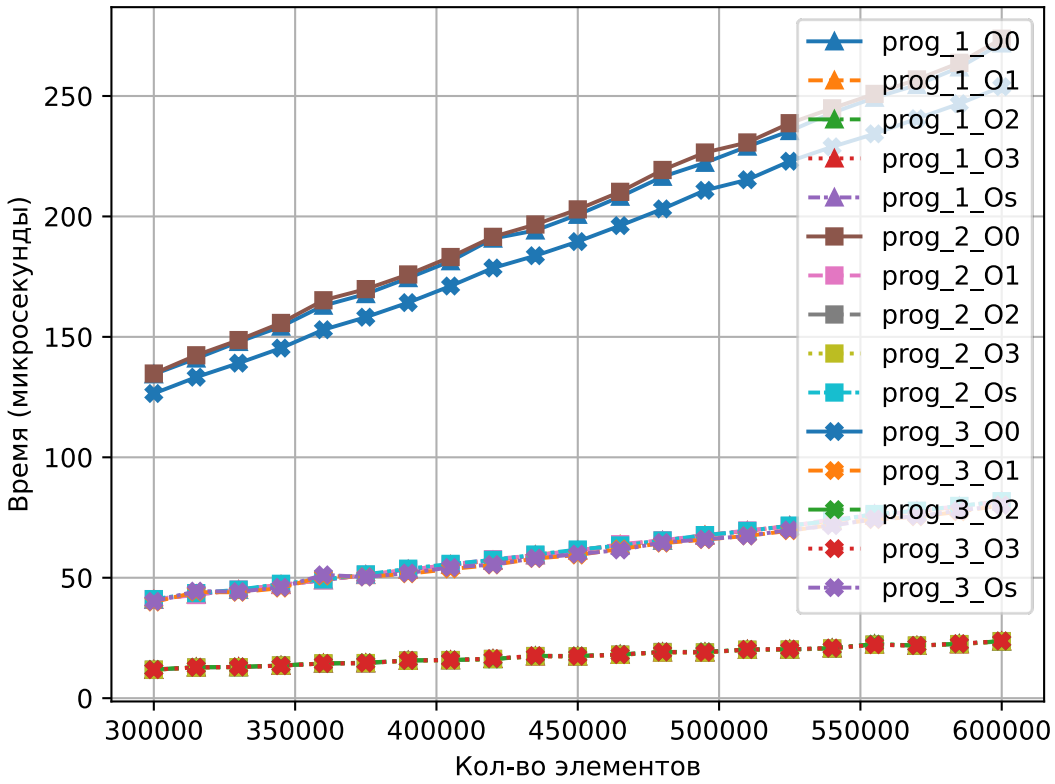
- Os
- O0
- O1
- O2
- O3

Написать скрипт **build_apps.sh**, вызвав который, можно получить весь набор необходимых исполняемых файлов. В результате получается 15 исполняемых файлов, которые помещаются в папку apps. Написать скрипт **update_data.sh**, вызвав который, можно добавить некоторые данные в датасет экспериментов. Скрипт по умолчанию запускает 15 версий программы с разным количеством элементов некоторое количество раз. Результаты записывает в папку data. Из-за простоты и линейности замеряемой функции, был выбран отрезок для замера времени от 300 000 элементов до 600 000 элементов с шагом 15 000 элементов.

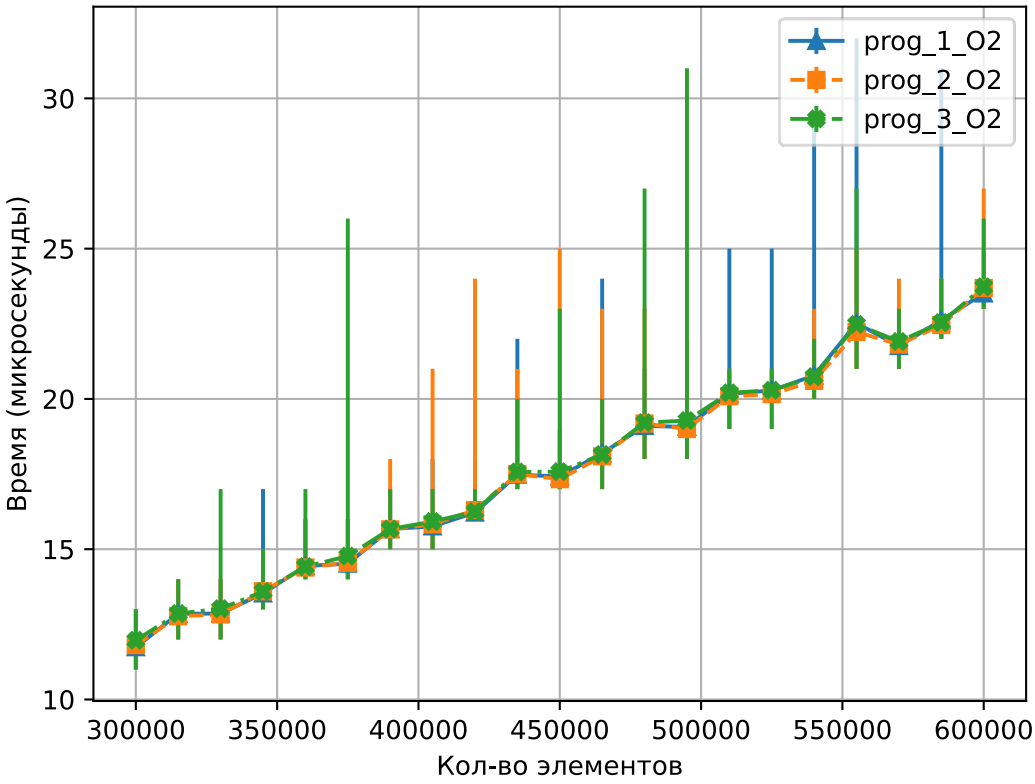
Написать скрипт **make_preproc.py**, который подготавливает данные из набора data, проводит первичный анализ: считает среднее арифметическое, медианное, находит максимум и минимум, вычисляет нижний и верхний квартили и записывает результаты в папку prep_data.

Написать скрипт **make_postproc.py**, который выполняет постобработку данных, строит и сохраняет по ним 3 графика:

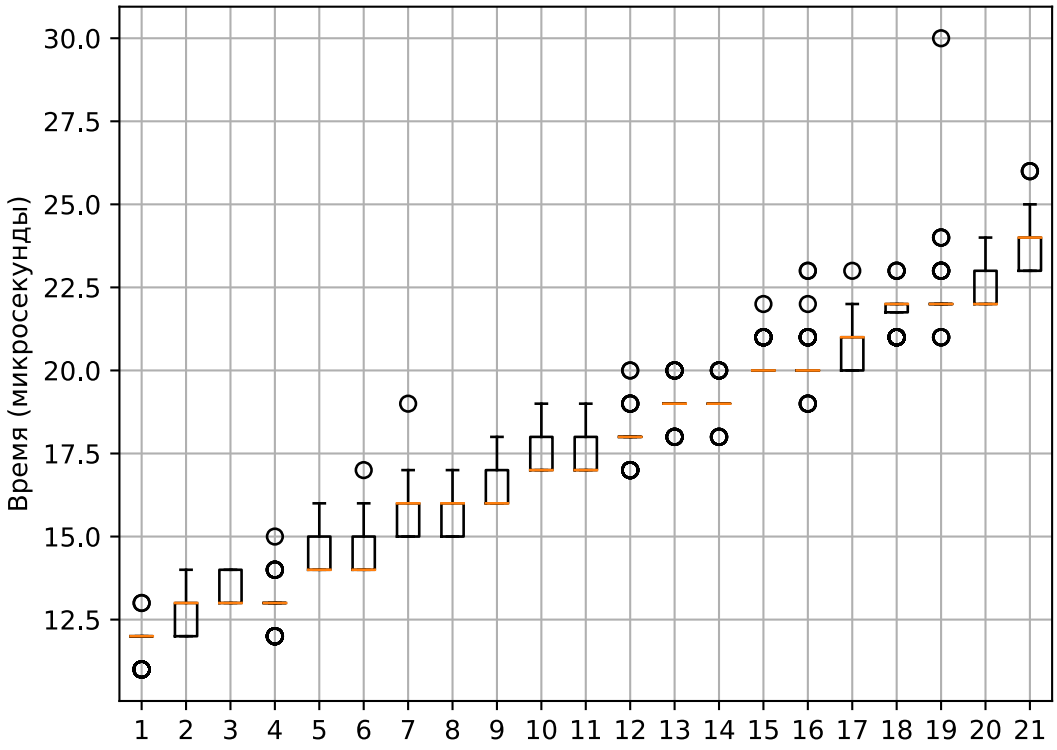
- **Обычный кусочно-линейный график** зависимости времени выполнения в любых единицах измерения времени от числа элементов массива для всех 15 вариантов программы.



- **Кусочно-линейный график с ошибкой** для всех вариантов обработки массива при уровне оптимизации 02.



- **График с усами** для варианта обработки «через квадратные скобки» при уровне оптимизации 03.



На этом графике кружки – выбросы из экспериментов, оранжевые линии – среднеарифметические значения, прямоугольниками обозначаются верхние и нижние квартили, усами максимальные и минимальные значения, не включая выбросы.

Написать скрипт **go.sh**, который выводит данные эксперимента включая графики. Этот скрипт вызывает по очереди предыдущие четыре.

Задание 8

В рамках данного задания необходимо было написать программу, которая сортирует строки матрицы по возрастанию суммы элементов в строке. Добавлена функция автоматического заполнения матрицы элементами. Добавлена возможность задавать количество строк в матрице при вызове программы. Так как матрица может быть не квадратной, то количество элементов в строке было принято за константное значение.

Написано две версии целевого алгоритма:

1. В первом случае программа проходит по строкам матрицы, считает их сумму и записывает в массив сумм, а потом сортирует строки матрицы по массиву сумм. Это вариант программы с кешированием промежуточных результатов
2. Во втором случае программа проходит по строкам матрицы, считает их сумму, если она больше, чем сумма в следующей строке, то она меняет

их местами. Так программа проходит столько раз, сколько кол-во строк в матрице.

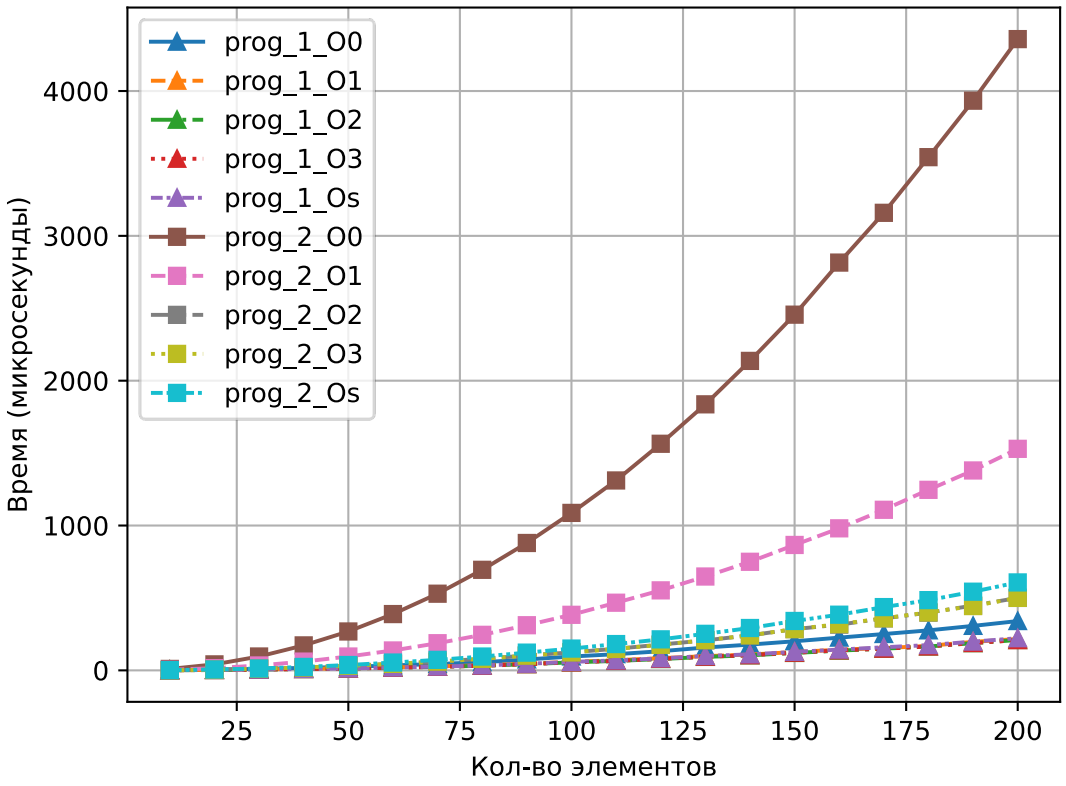
Для замера времени используется функция, возвращающая количество микросекунд.

Было собрано 10 исполняемых файлов программы, а именно пять уровней оптимизации и две версии программы с кешированием и без.

Для проведения эксперимента написать 5 скриптов:

- build_apps.sh – собирает 10 исполняемых файлов из 2 программ с разным уровнем оптимизации и записывает их в папку apps
- update_data.sh – добавляет экспериментальные данные в папку data, запуская все исполняемые файлы с различным количеством строк в матрице. И дописывает данные экспериментов.
- make_preproc.py – выполняет первоначальную обработку данных, а именно, находит среднеарифметическое, медианное, минимальное, максимальное, верхний квартиль и нижний квартиль данных из папки data и записывает в папку preproc_data.
- make_postproc.py – выполняет постобработку подготовленных данных и строит по ним график.
- go.sh – автоматически запускает предыдущие скрипты и выводит результаты эксперимента включая график.

Кусочно-линейные графики зависимости времени выполнения от числа строк матрицы для всех вариантов программы



Выводы

Задание 7

Исходя из кусочно-линейного графика, можно сделать вывод, что особенности доступа к элементам массива имеет значение только при уровне оптимизации O0, это доказывает график ошибок, где все 3 программы имеют уровень оптимизации O2, там графики слипаются. Программа с формальной заменой индексации работает медленнее остальных, программа с индексацией работает немного быстрее, а программа с работой через указатель работает значительно быстрее. Это происходит из-за того, что:

- При работе через указатель используется прямой доступ к памяти.
- При работе через индекс используется операция индексации, а после операция доступа к памяти
- При формальной индексации используются арифметические операции, а прямой доступ к памяти

Таким образом, для доступа к элементу массиву через указатель программа производит одну операцию, а если действовать оставшимися двумя способами, то программа производит две операции для достижения того-же результата.

Так как инициализация и заполнение массива элементами выполняется некоторое время, а задача измерить время выполнения целевого алгоритма, то для большей точности экспериментов, время замеряют только у целевого алгоритма.

Задание 8

На графике видно, что prog_1 (программа без кеширования промежуточных результатов) сильно проигрывает по времени prog_2 (программа с кешированием результатов, это происходит из-за того, что при кешировании результаты сохраняются и не надо каждый раз считать сумму чисел в строке. А так как происходит сортировка массива, то алгоритм без кеширования массива сумм на порядок сложнее алгоритма с кешированием сумм, из-за этого последний алгоритм при любом уровне оптимизации работает быстрее.