Национальный исследовательский университет “Высшая школа экономики”.

Факультет компьютерных наук. Программная инженерия.

Архитектура вычислительных систем.

Индивидуальное домашнее задание №2 студента группы БПИ213 Абрамова Александра Сергеевича.

Вариант 16.

16. Разработать программу, которая вычисляет **количество цифр и букв** в заданной ASCII-строке.

В результате анализа установленных требований было выявлено, что необходимо выполнить следующее:

1. Разработать программу на языке C, которая решает поставленную задачу и удовлетворяет следующим требованиям:
   - 1.1. Должны присутствовать функции с передачей данных через параметры.
   - 1.2. Должны использоваться локальные переменные, в том числе для хранения введённой строки в виде массива символов (обрабатывать вводимые данные посимвольно при вводе запрещается).
   - 1.3. Выбор способа ввода данных с помощью аргументов командной строки:
     - 1.3.1. Стандартный ввод из потока `stdin`
     - 1.3.2. Ввод из указанного файла
     - 1.3.3. Случайная генерация входных данных заданного в `stdin` размера.
   - 1.4. Программа не должна аварийно завершаться при любых переданных параметрах командной строки и полученных входных данных (в том числе при невозможности открытия указанных файлов).
   - 1.5. Выбор способа вывода данных с помощью аргументов командной строки:
     - 1.5.1. Стандартный вывод в поток `stdout`
     - 1.5.2. Вывод в указанный файл
   - 1.6. Программа должна позволять проводить анализ времени выполнения алгоритма без учёта ввода и вывода данных. Для этого следует реализовать вывод вычисленного времени выполнения указанным в командной строке способом (см. п.1.5) на отдельной строке.
2. Получить ассемблерный листинг программы на языке C без оптимизирующих и отладочных опций и улучшить его:

2.1. Удалить лишние макросы, инструкции и директивы.

2.2. Провести рефакторинг программы с максимальным использованием регистров процессора.

2.3. Добавить комментарии, поясняющие эквивалентное представление переменных в программе на C.

2.4. Добавить комментарии, описывающие передачу фактических параметров и перенос возвращаемого результата.

2.5. В функциях для формальных параметров добавить комментарии, описывающие связь между параметрами языка Си и регистрами (стеком).

2.6. Реализовать программу в виде двух или более единиц компиляции.

2.7. Использовать вместо методов библиотеки `libc` собственные реализации, опирающиеся на системные вызовы операционной системы.

3. Реализовать удобный способ тестирования программ и измерения их времени выполнения

3.1. Вручную создать не менее десяти наборов тестовых данных для проверки программ.

3.2. Реализовать программу на языке JavaScript, которая позволит сгенерировать достаточное для эффективного тестирования количество наборов входных и выходных данных. Сгенерировать с помощью этой программы не менее 100 наборов тестовых данных разного размера.

3.3. Реализовать программу на языке JavaScript, которая автоматически запускает переданные ей с помощью файла конфигурации исполняемые файлы, реализует последовательный ввод данных через стандартный поток, обработку результата работы программы и его сравнение с правильным ответом, а также вывод вердикта о корректности работы алгоритма.

3.4. Реализовать аналогичную описанной в п.3.3 программу, которая использует файловый ввод данных вместо стандартного потока.

3.5. Реализовать программу на языке JavaScript, которая производит запуск переданных ей с помощью файла конфигурации исполняемых файлов на различных наборах входных данных и выводит время работы алгоритмов с учётом и без учёта времени, затраченного на запуск программы и организацию ввода и вывода данных, при разных размерах входных данных.

4. Провести сравнение размера и эффективности ассемблерной программы, полученной после рефакторинга, и программы на языке C, собранной с различными опциями компиляции утилиты `gcc` (`-O0`, `-O1`, `-O2`, `-O3`, `-Ofast`, `-Os`) и без них.

5. При разработке допускается сохранять промежуточные версии программ, а также создавать дополнительные решающие задачу программы для проведения более полного анализа по размеру ассемблерного листинга, исполняемого файла и производительности.

Для определённости реализации установим следующие дополнительные требования:

1. Все данные, включая массив, хранящий введённую строку, должны храниться в регистрах процессора, на стеке или в секции статических данных.
2. Разработанный алгоритм должен корректно работать только на строках, состоящих из символов, имеющих ASCII-код в интервале $[0; 127]$.
3. Длина вводимой строки не должна превышать $2^{30} = 1073741824$ символов. Это позволит обеспечить достаточно большой размер входных данных при сравнительно небольшом использовании памяти. Для хранения каждого символа должно быть отведено не более 1 байта памяти. Следовательно, для хранения 2^{30} символов потребуется не более 1 ГБайт памяти.

   $$2^{30}\ \text{Символов} \leq 2^{30}\ \text{Байт} = 2^{20}\ \text{Кбайт} = 2^{10}\ \text{МБайт} = 1\ \text{ГБайт}$$

   Такой объём доступен на большинстве современных устройств. Тем не менее размера стека (который ограничен 8 Мбайт на устройствах под управлением ОС Linux) для поддержания такого объёма данных недостаточно, поэтому для хранения представления строки в памяти программы должна быть использована секция статических данных.
4. В случае обнаружения ввода, не удовлетворяющего требования п.2 или п.3 программа должна вывести ошибку и немедленно завершиться.
5. Формат аргументов командной строки должен быть следующий:

   ```
   exe_name in_flag in_file out_flag out_file
   ```

   Здесь под указанными именами понимаются следующие значения:

   a. `exe_name` - имя запускаемого исполняемого файла
   b. `in_flag` - значение, указывающее на способ ввода
      a. 0 - использование стандартного потока ввода `stdin`
      b. 1 - использование файлового ввода
      c. 2 - использование генератора случайных данных.
   c. `in_file` - имя файла со входными данными. Этот параметр должен быть указан только если `in_flag = 1`
   d. `out_flag` - значение, указывающее на способ вывода

a. 0 - использование стандартного потока вывода `stdout`

b. 1 - использование файлового вывода

e. `out_file` - имя файла для выходных данных. Этот параметр должен быть указан только если `out_flag = 1`

При передаче некорректных параметров командной строки программа может как аварийно завершиться, так и продолжить работу, интерпретировав данные произвольным образом.

С учётом всех вышеизложенных требований была реализована программа на языке `C` в файле `c/solution.c`:

1. При запуске программа обрабатывает переданные аргументы командной строки и настраивает способы ввода и вывода:
   1.1. Если `in_flag = 0` или `in_flag = 1`, по окончании работы подпрограммы имя `stdin` указывает на соответствующий поток ввода. Для обеспечения этого для открытии файла используется функция `freopen` стандартной библиотеки языка `C`.
   1.2. Если `in_flag = 2`, переменной `read_mode` устанавливается значение `1`, означающее, что требуется случайная генерация входных данных. В ином случае значение `read_mode` равно нулю.
   1.3. При любом значении `out_flag` по окончании работы подпрограммы имя `stdout` указывает на соответствующий поток вывода. Для этого для открытии файла используется функция `freopen` стандартной библиотеки языка `C`.
   1.4. Если во время обработки аргументов командной строки произошла ошибка: обнаружено недостаточное количество параметров или не удалось открыть один из указанных файлов - программа выводит сообщение об ошибке на стандартный поток вывода и завершается.
2. Далее происходит выделение статической памяти под строку: массив размером `MAX_INPUT_LENGTH + 1` значений однобайтового беззнакового типа `unsigned char`. Здесь `MAX_INPUT_LENGTH` - макро-определённое значение, равное `1073741824` - зафиксированному в требованиях значению наибольшей допустимой длины строки.
3. Выполнение п.1 позволяет произвести ввод данных лишь по двум значениям: адрес выделенной в п.2 памяти и `read_mode`, указывающий на способ получения данных: из `stdin` или путём случайной генерации. Таким образом, подпрограмма `input` осуществляет заполнение строки, получая указанные данные как параметры функции.

3.1. Если требуется ввод из `stdin`, программа использует функцию стандартной библиотеки языка `C` `fread`, которая позволяет одним вызовом считать все входные данные или их часть, если фактический размер вводимой строки превышает установленное ограничение. Таким образом, в результате вызова в блок памяти, адрес которого хранится в переменной `buffer`, будет записано не более `MAX_INPUT_LENGTH + 1` значений размером `1` байт из `stdin`. Возвращаемое значение - длина строки - записывается в переменную `length`.

3.1.1. Если количество считанных данных превышает максимальный допустимый размер ввода, подпрограмма экстренно завершает работу, указывая на это возвращением числа `MAX_INPUT_LENGTH + 1`.

3.1.2. В ином случае подпрограмма проверяет, являются ли все символы введённой строки ASCII-символами с кодами в интервале $[0; 127]$. Для этого организуется цикл по всей строке и посимвольное сравнение со значением 127. Так как каждый символ хранится в беззнаковом типе данных, отрицательных значений после ввода быть не может и любой некорректный символ будет иметь код, превышающий 127.

3.1.2.1. Если такой символ обнаружен, подпрограмма экстренно завершает работу, указывая на это возвращением числа `MAX_INPUT_LENGTH + 1`.

3.1.2.2. Если введённая строка корректна, подпрограмма `input` возвращает её длину - количество считанных символов.

3.2. Если требуется случайная генерация входных данных, то подпрограмма запрашивает ввод длины строки через `stdin` и проверяет, не превышает ли введённое значение установленное ограничение: при этом достаточно проверить только верхнюю границу, так как введённое пользователем число интерпретируется программой как беззнаковое.

3.2.1. Если введённая длина корректна, то программа устанавливает семя генерации случайных данных как текущее время системы с помощью вызовов `srand(time(NULL))`: криптографически-сильная случайная генерация данных не требуется, поэтому данный алгоритм удовлетворяет заданным требованиям.

3.2.2. После этого с помощью цикла по всей строке программа посимвольно генерирует входные данные и приводит их к интервалу $[0; 127]$ с помощью взятия остатка от деления на `128`, что реализовано с помощью битовых операций.

3.2.3. По окончании генерации подпрограмма `input` возвращает длину строки.

4. После заполнения строки программа проверяет, было ли оно успешно: для этого производится проверка, не превышает ли возвращённое подпрограммой `input` значение максимальный

допустимый размер ввода. Таким образом, “флаг” экстренного завершения подпрограммы `input` - возврат значения `MAX_INPUT_LENGTH + 1` - приведёт к завершению всей программы.

5. Далее происходит собственно решение задачи с помощью подпрограммы `solve`. При этом производится замер времени выполнения алгоритма с помощью функции `clock` стандартной библиотеки языка `C`: программа сохраняет значение точного числа тактов процессора до и после вызова функции `solve`, что позволяет вычислить затраченное время, нормировав их разность на количество тактов процессора, выполняемых за секунду. Таким образом, в переменной `answer` оказывается ответ на задачу, а в переменной `cpu_time_used` - время, затраченное на его вычисление.
    5.1. Подпрограмма `solve` получает на вход в качестве аргументов функции адрес памяти, в которой хранится введённая строка, а также длину этой строки.
    5.2. Для удобства вычисления и передачи ответа на задачу была создана структура `Result`, которая содержит количество цифр и букв в строке под именами `numbers` и `letters` соответственно.
    5.3. Для вычисления ответа программа осуществляет последовательную проверку каждого символа строки:
        5.3.1. Так как все цифры в кодировке `ASCII` располагаются последовательно, для проверки того, что символ - цифра, необходимо и достаточно проверить, находится ли код символа между кодами символов `0` и `9` включительно.
        5.3.2. Аналогично, все строчные и все заглавные буквы кодируются последовательными числами. Следовательно, для проверки того, что символ - буква, необходимо и достаточно проверить, лежит ли код символа между `a` и `z` или `A` и `Z`.
6. Для вывода ответа создана подпрограмма `output`, которая принимает на вход в качестве параметров функции одно значение типа `Result`. Для печати значения в поток, на который указывает `stdout`, была использована функция `printf` стандартной библиотеки языка `C`.
7. После вывода ответа аналогично выводится вычисленное значение время выполнения в наносекундах.

Для удобства тестирования реализованной программы и верификации дальнейших изменений была создана программа для тестирования исполняемых файлов, размещённая в папке `testing` и состоящая из следующих частей:

1. Файл конфигурации `CONFIG.js` позволяет установить список тестируемых исполняемых файлов (`to_test`) и список групп тестов (`test_groups`).
2. В папке `tests` размещены наборы входных и выходных данных, которые использовались для тестирования. Непосредственно в директории `tests` расположены папки, соответствующие каждой группе тестов, каждая из которых содержит по папке для каждого теста. Каждая такая папка содержит файл со входными данными `in.in` и с эталонными выходными данными `out.out`. Группы тестов организованы следующим образом:

Номер группы тестов	Номера тестов	Длина вводимой строки (n)	Комментарий
0	1-10	-	Вручную созданные небольшие наборы данных.
1	11-20	10	Наборы данных, сгенерированные программой `gen.js` (см п.3)
2	21-35	10^2	
3	36-50	10^3	
4	51-65	10^4	
5	66-80	10^5	
6	81-95	10^6	
7	96-110	10^7	
8	111-120	10^8	

3. Программа `gen.js` позволяет произвести генерацию тестов для групп, заданных в файле конфигурации. Для этого программа случайным образом получает указанное количество символов с кодами в интервале [0; 127], объединяет их в одну строку, подсчитывает верный ответ на задачу, после чего записывает в соответствующие файлы в папке `tests`.
4. Программы `test_stdin.js` и `test_file.js` последовательно запускают указанные исполняемые файлы на всех тестовых входных данных, сравнивают вывод программы с корректными выходными данными и печатают вердикт в `stdout`. При этом первая программа использует для ввода стандартный поток ввода, а вторая - файловый ввод. Способ вывода в обеих программах совпадает со способом ввода (`test_file.js` организует обработку данных с

помощью файла `tmp.out` в директории `int` для временных промежуточных файлов). Если хотя бы на одном тесте группы был получен неверный ответ, вся группа считается не пройденной.

5. Также была реализована программа `benchmark.js`, которая не тестирует первую группу тестов и не производит проверку правильности ответа, но дополнительно осуществляет запуск алгоритмов на входных данных максимального размера, используя встроенный генератор случайных чисел (`in_flag = 2`) для получения исходной строки. При этом программа анализирует напечатанное время работы алгоритма, а также время работы всей программы с учётом затрат на ввод и вывод данных, и печатает на стандартный поток вывода `stdout` среднее время работы исполняемого файла на тестах каждой группы.

Для удобства сборки и запуска программ было принято решение использовать утилиту `npm`. Для этого в файле `package.json` были реализованы следующие сценарии (`scripts`):

1. Сценарии вида `build_c_*` преобразуют программу на языке `C` в исполняемый файл с использованием соответствующих опций утилиты `gcc`. Для удобства сценарий `build_c_all` собирает все программы. Полученные исполняемые файлы сохраняются в папке `int`.
2. Сценарий `run_c` запускает собранную без оптимизирующих опций программу, а сценарий `c` собирает программу на языка `C` без использования опций оптимизации и запускает её.
3. Сценарии вида `get_assembly_*` преобразуют программу на языке `C` в ассемблерный листинг с использованием соответствующих опций утилиты `gcc`. Полученные файлы сохраняются в папке `int`. При ассемблировании используются следующие аргументы командной строки:
   3.1. Одна из оптимизирующих опций, кроме сценария `get_assembly`
   3.2. `-masm=intel` для генерации ассемблерного листинга в синтаксисе `intel`
   3.3. `-Wall` для отлавливания возможных ошибок, допущенных в программе на языке `C`
   3.4. `-fno-asynchronous-unwind-tables` и `-fcf-protection=none` для избавления от излишних в данном случае инструкций, помогающих избежать ошибок при работе с памятью, которые могут создавать уязвимости в программе.
   3.5. `-S` указывает утилите `gcc`, что необходим именно ассемблерный листинг
4. Сценарии вида `compile_asm_*`, преобразующие соответствующие единицы компиляции программы на языке ассемблера (см. далее) в объектные файлы, сохраняемый в директории `int`, с помощью утилиты `as`.

5. Сценарий `compile_asm`, выполняющий компиляцию всех модулей программы на языке ассемблра.
6. Сценарий `link_asm`, использующий утилиту `ld` для сборки объектных модулей программы на языке ассемблера в исполняемый файл, сохраняемый в папке `int` под именем `solution-asm.exe`
7. Сценарий `build_asm`, производящий компиляцию и линковку программы с использованием описанных в п.5-6 сценариев.
8. Сценарий `run_asm` запускает исполняемый файл `solution-asm.exe`, а сценарий `asm` собирает программу на языка ассемблера без использования опций отладки и запускает её.
9. Сценарий `build_all` производит сборку обеих программ: на языке `C` и на языке ассемблера.
10. Сценарий `gen_tests`, запускающий программу генерации тестовых данных `gen.js`
11. Сценарий `test_stdin`, запускающий тестирование указанных в файле конфигурации тестирующей программы `CONFIG.js` исполняемых файлов с использованием ввода через стандартный поток.
12. Сценарий `test_file`, запускающий тестирование указанных в файле конфигурации тестирующей программы `CONFIG.js` исполняемых файлов с использованием файлового ввода.
13. Сценарий `benchmark`, производящий замеры эффективности работы указанных в файле конфигурации тестирующей программы `CONFIG.js` исполняемых файлов с использованием программы `benchmark.js`

Ассемблерный листинг программы на языке `C` был получен с помощью сценария `get_assembly` и представлен в файле `solution.s` директории `int`.

Программа на языке ассемблера, полученная после рефакторинга, была фактически написана вручную с опорой на сгенерированный компилятором листинг и представлена в директории `asm`.

1. Программа состоит из пяти единиц компиляции:
   1.1. `main.asm`, который реализует принятие управление от операционной системы, разбор параметров командной строки и организацию работы алгоритма
   1.2. `lib.asm`, реализующий функции `scanf` и `printf` ввода и вывода чисел без использования стандартной библиотеки языка `C`
   1.3. `input.asm`, производящий считывание строки из потока, дескриптор которого хранится в глобальной переменной `.instream`, или её случайную генерацию

1.4. `solve.asm`, реализующий алгоритм решения задачи

1.5. `output.asm`, производящий вывод результата в поток, дескриптор которого хранится в глобальной переменной `.outstream`

2. `main.asm` выполняет следующие операции:

2.1. Выделяет память для хранения дескрипторов используемых потоков ввода и вывода, доступ к которой доступен по именам `.instream` и `.outstream`, а также устанавливает их значения, проводя разбор аргументов командной строки аналогично программе на языке `C`: для организации переменной `array_read_mode` используется регистр `rsi`, а для открытия файла для ввода или вывода, если это необходимо, используется системный вызов `sys_open` ОС Linux с последующей проверкой возвращаемого результата и сохранением в памяти, адресуемой указателями `.instream` или `.outstream`

2.2. Для обеспечения максимального использования регистров процессора функция `_start` - точка входа в программу - сохраняет на стеке регистры `r12`, `r13`, `rbx` и `rbp`, что позволяет использовать их в реализации, не нарушая соглашение о вызовах функций в ОС linux, а перед завершением работы - восстанавливает их значения со стека.

2.3. Для оптимизации обращений к памяти значения переменных `int argc` и `char** argv`, получаемые функцией `_start` через стек, перемещаются в регистры `r12` и `r13` соответветственно.

2.4. Память для хранения строки выделяется локально для данной единицы компиляции в секции статических данных с помощью инструкции `.comm`

2.5. Происходит последовательный вызов всех функций аналогично программе на языке `C`. Для передачи параметров в соответствии с `cdecl` использованы регистры процессора `rdi` (первый параметр) и `rsi` (второй параметр).

2.5.1. Вызов подпрограммы `input` и обработка возвращаемого значения - длины массива. Эквивалентное представление переменной `length` в программе на этом этапе - регистр `r12`.

2.5.2. Системный вызов `sys_clock_gettime` для получения точного текущего времени системы, что необходимо при организации замера времени работы алгоритма. В связи с особенностями работы вызова, на стеке дополнительно выделяется 16 байт, в которых размещается текущее время системы при вызове `sys_clock_gettime`.

После вызова происходит обработка значения и его сохранения в регистре r13 для дальнейшего использования.

2.5.3. Вызов подпрограммы solve и сохранение пары возвращаемых через регистры rdx и rax значений в регистрах rbx и r12 соответственно.

2.5.4. Системный вызов sys_clock_gettime для получения текущего времени системы после окончания алгоритма, его обработка, вычисление времени работы алгоритма с использованием сохранённого до вызова solve времени системы и его сохранение в регистре r13, использованном вместо переменной cpu_time_used

2.5.5. Вызов подпрограммы output, которая организует печать результата работы в поток .outstream

2.5.6. Вывод времени работы алгоритма с помощью системного вызова sys_write для печати строк и функции printf для вывода числа.

2.5.7. Завершение программы: закрытие потоков ввода и вывода с помощью вызова sys_close, восстановление значений сохранённых регистров и выход из функции с помощью системного вызова sys_exit.

2.5.8. Для обработки ошибок организованы метки error_cli_print, error_input_print и error_output_print, передача управления на которые приводит к печати соответствующей ошибки в поток вывода и немедленному завершению программы.

3. lib.asm реализует функции scanf и printf ввода и вывода числа:

3.1. Подпрограмма scanf использует системный вызов sys_read для посимвольного считывания числа как строки, а также реализует преобразование считанной строки в число, возвращаемое в регистре rax.

3.2. Подпрограмма printf реализует преобразования числа, переданного в качестве аргумента функции, в строку на стеке и производит её печать с помощью системного вызова sys_write.

4. input.asm производит заполнение строки, расположенной по адресу, переданному в функцию в качестве аргумента.

4.1. Для оптимизации обращений к памяти значение переменной unsigned char* buffer, получаемое функцией input, сохраняется в регистре r12.

4.2. Производится определение требуемого способа ввода и заполнение памяти:

4.2.1. Если требуется считывание строки из потока `.instream`, программа использует системный вызов `sys_read` для заполнения всей памяти, выделенной под строку.

4.2.1.1. Из-за того, что операционная система производит буферизацию потоков ввода (в частности, `stdin`), передавая данные программе при обнаружении перевода строки, для корректного считывания всех данных происходит многократное повторение вызова, пока не было считано количество символов, превышающее установленный предел `MAX_INPUT_LENGTH`, или пока передача данных не произошла из-за получения `EOF`, а не символа перевода строки (ASCII-код 10). Вычисление суммарной длины считанной строки происходит в регистре `r13` - аналоге переменной `uint64_t length`

4.2.1.2. По завершении считывания для обеспечения корректности работы программы производится проверка кода каждого считанного символа на принадлежность установленному интервалу [0; 127]. Для этого организован цикл `check_ascii`, использующий регистр `rcx` как аналог переменной `uint64_t i`

4.2.2. Если требуется генерация случайных данных, управление передаётся на метку `input_random`.

4.2.2.1. С помощью функции `scanf` производится считывание желаемой длины строки с последующей проверкой корректности ввода и размещением значения в регистре `r13`.

4.2.2.2. Для организации генерации случайных данных недостаточно регистров `r12` и `r13`, значения которых были ранее сохранены на стеке, поэтому дополнительно производится сохранение значения регистра `r14` и последующее его восстановление.

4.2.2.3. Для генерации случайных данных используется системный вызов `sys_getrandom`, имеющий достаточно маленькие ограничения на количество генерируемых за один вызов байт (не более `33554431` байт при чтении из источника `urandom`). Вследствие этого, для заполнения всей строки вызов производится многократно, пока строка не будет заполнена

(организован цикл `fill_string` с использованием значения регистра `r14` для итерирования).

4.2.2.4. Производится нормализация сгенерированных данных: каждый символ полученной строки заменяется на символ с кодом в интервале [0; 127] путём взятия остатка от деления случайного значения на 128, что реализовано в виде побитового `И` с числом `127`. Для этого организован цикл `fix_ascii`, итерируемый значением регистра `rcx`.

4.3. Если заполнение строки было произведено успешно, производится размещение длины полученной строки в регистре `rax` для передачи возвращаемого значения, восстановление значений сохранённых регистров и выход из функции.

4.4. Для обработки ошибок организованы метки `too_long_array` и `not_ascii`, передача управления на которые приводит к печати соответствующей ошибки в поток вывода и немедленному завершению подпрограммы с возвратом значения, превышающего максимальную длину ввода.

5. `solve.asm` производит решение задачи - вычисление количества букв и цифр в переданной строке:

5.1. Организация фрейма не требуется: подпрограмма не совершает вызовов сторонних функций и не использует память на стеке.

5.2. Для работы алгоритма организован цикл `loop`, итерируемый значением регистра `rcx` - аналогом переменной `uint64_t i` в программе на `C`. Также, здесь `rsi` - аналог переменной `const uint64_t length`, а `rdi` - переменной `const unsigned char* buffer`.

5.3. На каждой итерации цикла производится проверка отдельного символа строки: если он - число (то есть ASCII-код символа лежит между кодами символов `0` и `9`), увеличивается счётчик `rax`; если он - строчная или заглавная буква (то есть ASCII-код символа лежит между кодами символов `a` и `z` или `A` и `Z`), увеличивается счётчик `rdx`.

5.4. По окончании цикла возвращаемое значение корректно размещено в регистрах `rax` и `rdx` в соответствии с соглашением о вызовах ОС Linux, поэтому достаточно лишь произвести возврат из функции с помощью инструкции `ret`.

6. `output.asm` производит печать ответа на задачу, передаваемого в паре регистров `rdi` и `rsi`, на поток вывода, указываемый переменной `.outstream`.

6.1. Во избежание потери значений при организации вызовов, оба значения сохраняются на стеке.

6.2. Производится вывод первой части строки-ответа с помощью системного вызова `sys_write`.

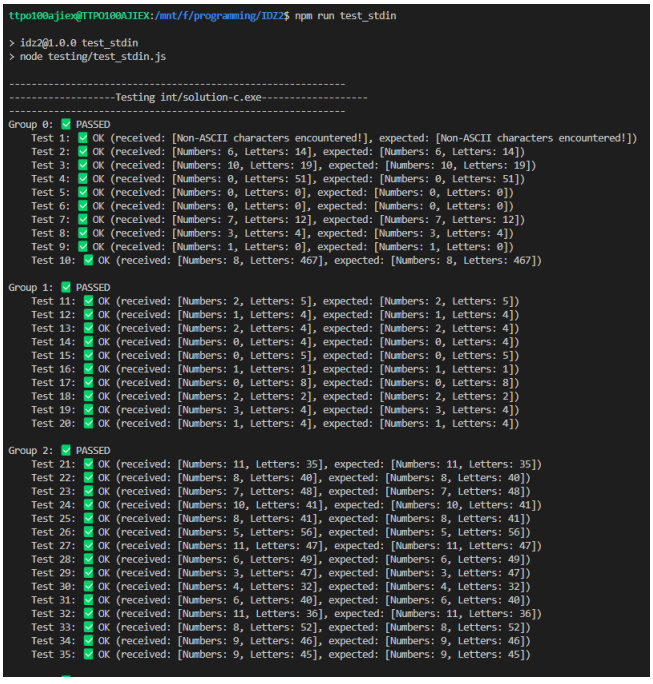
6.3. Производится вывод количества цифр в строке с помощью функции `printf`, причём необходимое значение помещается в регистр `rdi` из стека.

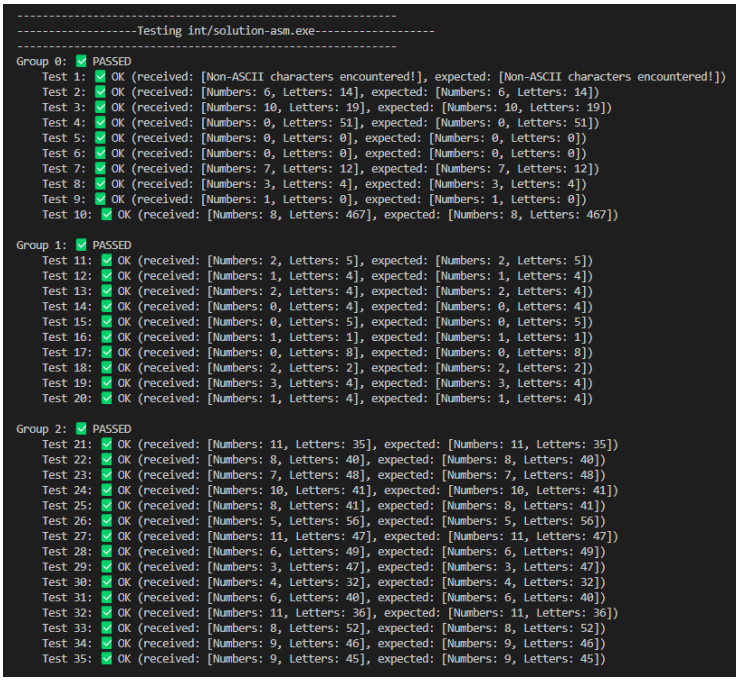
6.4. Аналогично описанному в п.6.2-6.3 производится вывод второй части ответа - количества букв.

6.5. В результате описанных действий, оба размещённых на стеке значения были оттуда “забраны”, вследствие чего для завершения подпрограммы достаточно лишь осуществить передачу управления по адресу возврата с помощью инструкции `ret`. Возвращаемым значением в таком случае является статус успешности последнего системного вызова.

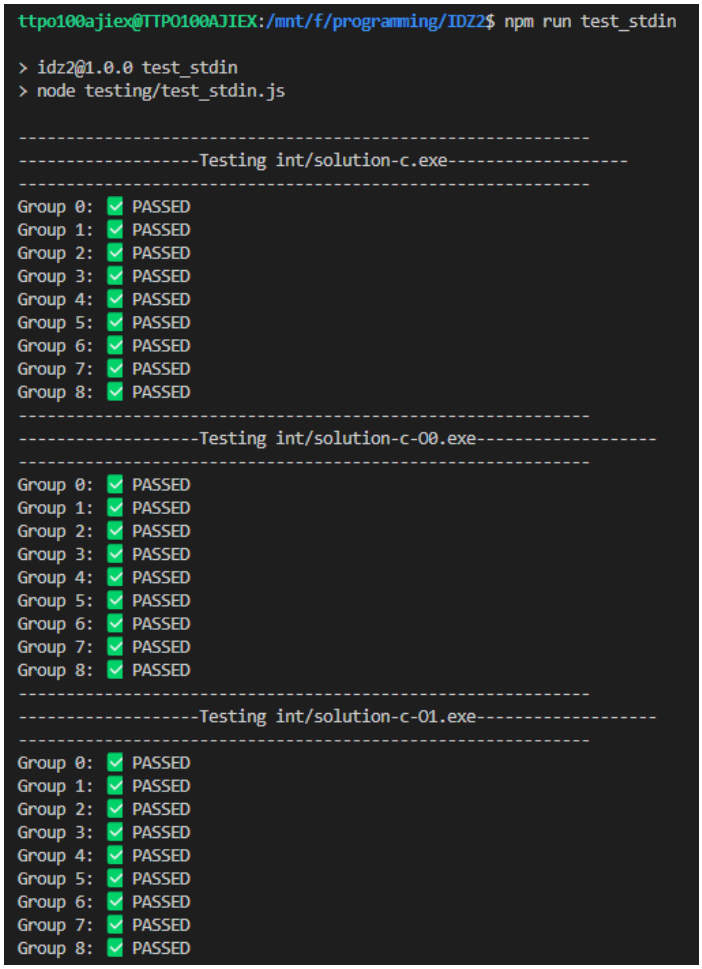
Для тестирования и измерения производительности использовалось устройство с 12th Gen Intel Core i5-12600K @ 3.6GHz 16 CPU, 32GB DDR5 RAM под управлением WSL Debian 11 на Windows 11 (в папке `int` итогового проекта представлены исполняемые файлы и ассемблерные листинги, полученные именно на этом устройстве).

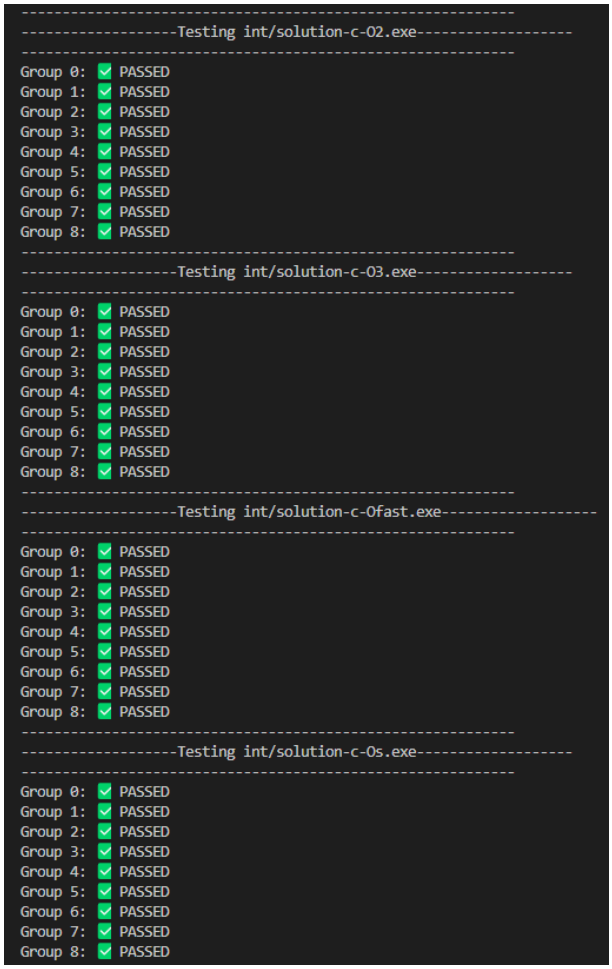
Для тестирования ввода через `stdin` были взяты полученные путём запуска сценария `build_all` исполняемые файлы `solution-c.exe` и `solution-asm.exe`. Как показывает результат запуска (копия результата представлена в файле `tests.stdin.txt` директории `report`), программы работают корректно.

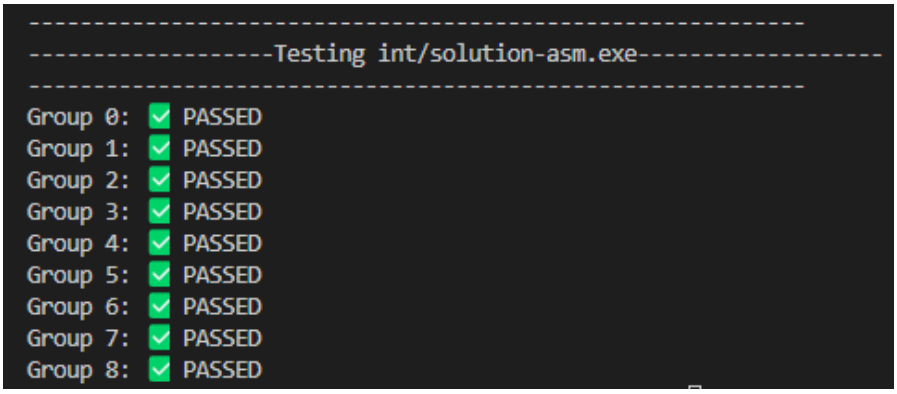




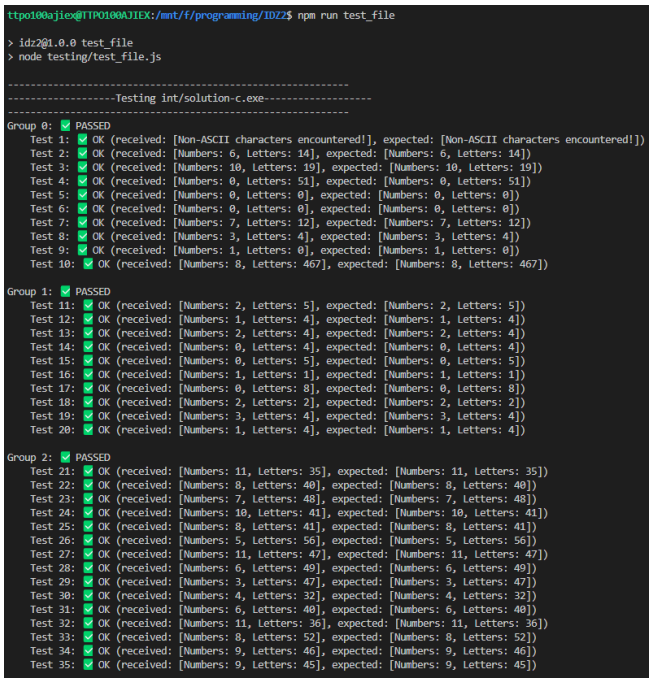
Дополнительно были также проверены и остальные исполняемые файлы. Копия полной версии вердикта тестировщика представлена в файле `tests.stdin.full.txt` директории `report`.

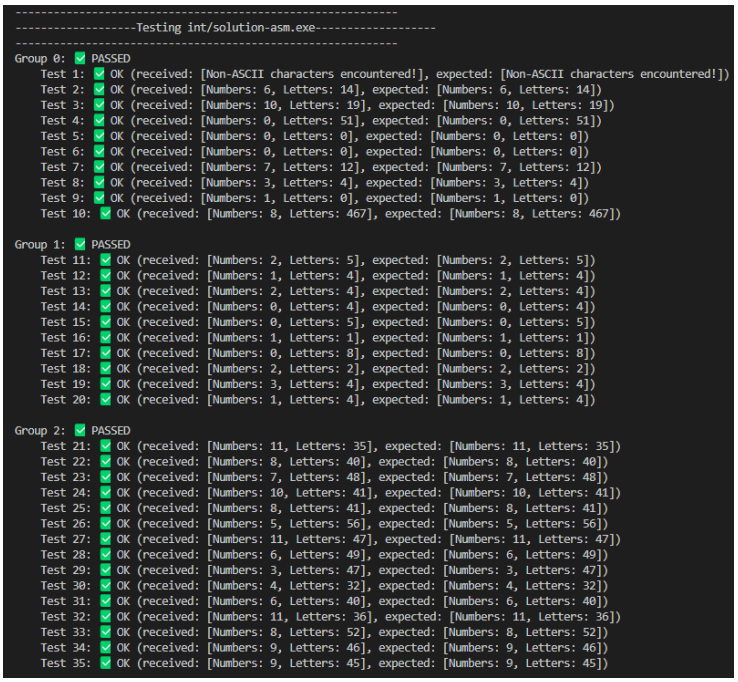


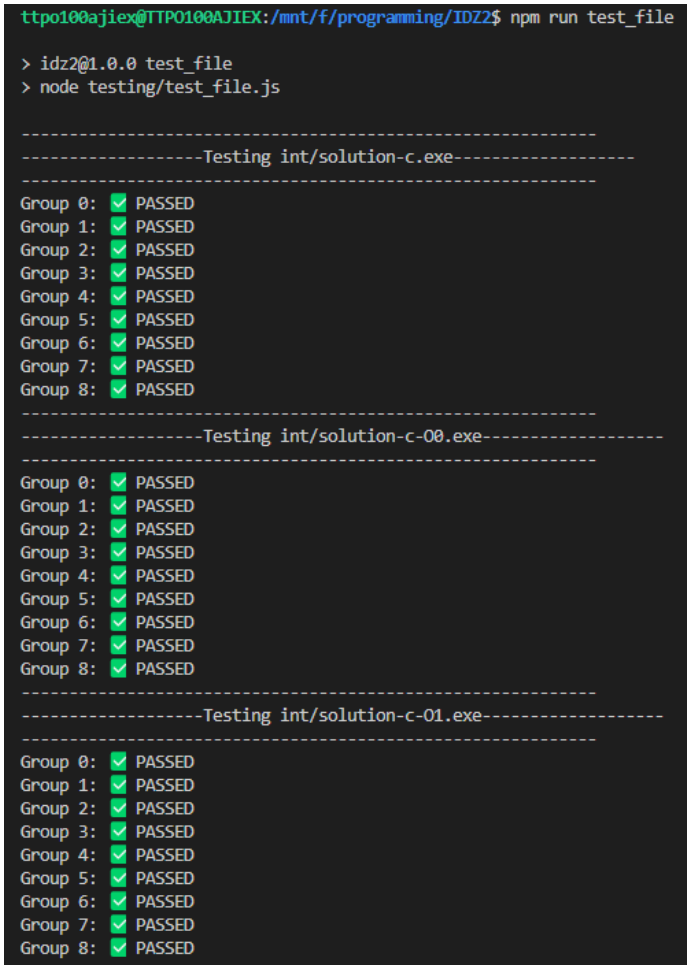


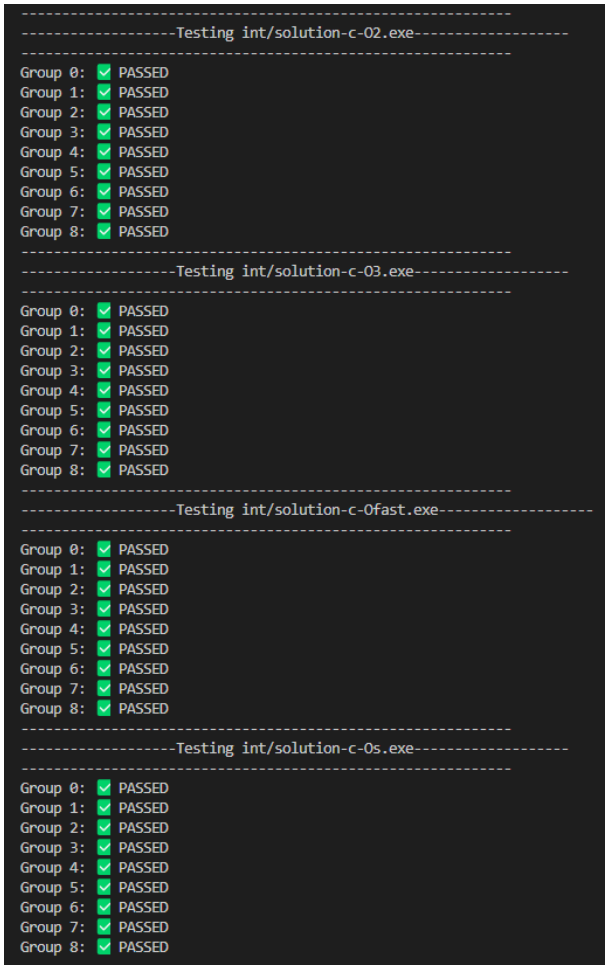


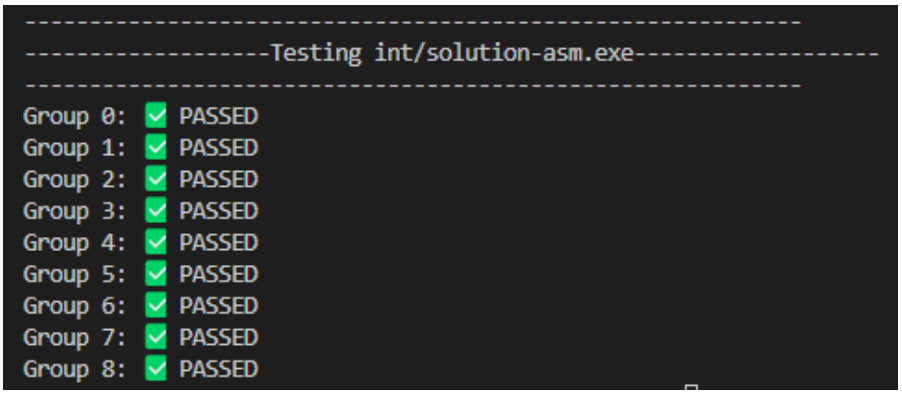
Аналогичное тестирование было произведено и с использованием файлового ввода и вывода. Результаты представлены в файлах `tests.file.txt` и `tests.file.full.txt` директории `report`.


ttpo100ajiex@TTPO100AJIEX:/mnt/f/programming/IDZ2$ npm run test_file
> idz2@1.0.0 test_file
> node testing/test_file.js
----------Testing int/solution-c.exe----------
Group 0: PASSED
Test 1: OK (received: [Non-ASCII characters encountered!], expected: [Non-ASCII characters encountered!])
Test 2: OK (received: [Numbers: 6, Letters: 14], expected: [Numbers: 6, Letters: 14])
Test 3: OK (received: [Numbers: 10, Letters: 19], expected: [Numbers: 10, Letters: 19])
Test 4: OK (received: [Numbers: 0, Letters: 51], expected: [Numbers: 0, Letters: 51])
Test 5: OK (received: [Numbers: 0, Letters: 0], expected: [Numbers: 0, Letters: 0])
Test 6: OK (received: [Numbers: 0, Letters: 0], expected: [Numbers: 0, Letters: 0])
Test 7: OK (received: [Numbers: 7, Letters: 12], expected: [Numbers: 7, Letters: 12])
Test 8: OK (received: [Numbers: 3, Letters: 4], expected: [Numbers: 3, Letters: 4])
Test 9: OK (received: [Numbers: 1, Letters: 0], expected: [Numbers: 1, Letters: 0])
Test 10: OK (received: [Numbers: 8, Letters: 467], expected: [Numbers: 8, Letters: 467])
Group 1: PASSED
Test 11: OK (received: [Numbers: 2, Letters: 5], expected: [Numbers: 2, Letters: 5])
Test 12: OK (received: [Numbers: 1, Letters: 4], expected: [Numbers: 1, Letters: 4])
Test 13: OK (received: [Numbers: 2, Letters: 4], expected: [Numbers: 2, Letters: 4])
Test 14: OK (received: [Numbers: 0, Letters: 4], expected: [Numbers: 0, Letters: 4])
Test 15: OK (received: [Numbers: 0, Letters: 5], expected: [Numbers: 0, Letters: 5])
Test 16: OK (received: [Numbers: 1, Letters: 1], expected: [Numbers: 1, Letters: 1])
Test 17: OK (received: [Numbers: 0, Letters: 8], expected: [Numbers: 0, Letters: 8])
Test 18: OK (received: [Numbers: 2, Letters: 2], expected: [Numbers: 2, Letters: 2])
Test 19: OK (received: [Numbers: 3, Letters: 4], expected: [Numbers: 3, Letters: 4])
Test 20: OK (received: [Numbers: 1, Letters: 4], expected: [Numbers: 1, Letters: 4])
Group 2: PASSED
Test 21: OK (received: [Numbers: 11, Letters: 35], expected: [Numbers: 11, Letters: 35])
Test 22: OK (received: [Numbers: 8, Letters: 40], expected: [Numbers: 8, Letters: 40])
Test 23: OK (received: [Numbers: 7, Letters: 48], expected: [Numbers: 7, Letters: 48])
Test 24: OK (received: [Numbers: 10, Letters: 41], expected: [Numbers: 10, Letters: 41])
Test 25: OK (received: [Numbers: 8, Letters: 41], expected: [Numbers: 8, Letters: 41])
Test 26: OK (received: [Numbers: 5, Letters: 56], expected: [Numbers: 5, Letters: 56])
Test 27: OK (received: [Numbers: 11, Letters: 47], expected: [Numbers: 11, Letters: 47])
Test 28: OK (received: [Numbers: 6, Letters: 49], expected: [Numbers: 6, Letters: 49])
Test 29: OK (received: [Numbers: 3, Letters: 47], expected: [Numbers: 3, Letters: 47])
Test 30: OK (received: [Numbers: 4, Letters: 32], expected: [Numbers: 4, Letters: 32])
Test 31: OK (received: [Numbers: 6, Letters: 40], expected: [Numbers: 6, Letters: 40])
Test 32: OK (received: [Numbers: 11, Letters: 36], expected: [Numbers: 11, Letters: 36])
Test 33: OK (received: [Numbers: 8, Letters: 52], expected: [Numbers: 8, Letters: 52])
Test 34: OK (received: [Numbers: 9, Letters: 46], expected: [Numbers: 9, Letters: 46])
Test 35: OK (received: [Numbers: 9, Letters: 45], expected: [Numbers: 9, Letters: 45])


----------Testing int/solution-asm.exe----------
Group 0: PASSED
Test 1: OK (received: [Non-ASCII characters encountered!], expected: [Non-ASCII characters encountered!])
Test 2: OK (received: [Numbers: 6, Letters: 14], expected: [Numbers: 6, Letters: 14])
Test 3: OK (received: [Numbers: 10, Letters: 19], expected: [Numbers: 10, Letters: 19])
Test 4: OK (received: [Numbers: 0, Letters: 51], expected: [Numbers: 0, Letters: 51])
Test 5: OK (received: [Numbers: 0, Letters: 0], expected: [Numbers: 0, Letters: 0])
Test 6: OK (received: [Numbers: 0, Letters: 0], expected: [Numbers: 0, Letters: 0])
Test 7: OK (received: [Numbers: 7, Letters: 12], expected: [Numbers: 7, Letters: 12])
Test 8: OK (received: [Numbers: 3, Letters: 4], expected: [Numbers: 3, Letters: 4])
Test 9: OK (received: [Numbers: 1, Letters: 0], expected: [Numbers: 1, Letters: 0])
Test 10: OK (received: [Numbers: 8, Letters: 467], expected: [Numbers: 8, Letters: 467])
Group 1: PASSED
Test 11: OK (received: [Numbers: 2, Letters: 5], expected: [Numbers: 2, Letters: 5])
Test 12: OK (received: [Numbers: 1, Letters: 4], expected: [Numbers: 1, Letters: 4])
Test 13: OK (received: [Numbers: 2, Letters: 4], expected: [Numbers: 2, Letters: 4])
Test 14: OK (received: [Numbers: 0, Letters: 4], expected: [Numbers: 0, Letters: 4])
Test 15: OK (received: [Numbers: 0, Letters: 5], expected: [Numbers: 0, Letters: 5])
Test 16: OK (received: [Numbers: 1, Letters: 1], expected: [Numbers: 1, Letters: 1])
Test 17: OK (received: [Numbers: 0, Letters: 8], expected: [Numbers: 0, Letters: 8])
Test 18: OK (received: [Numbers: 2, Letters: 2], expected: [Numbers: 2, Letters: 2])
Test 19: OK (received: [Numbers: 3, Letters: 4], expected: [Numbers: 3, Letters: 4])
Test 20: OK (received: [Numbers: 1, Letters: 4], expected: [Numbers: 1, Letters: 4])
Group 2: PASSED
Test 21: OK (received: [Numbers: 11, Letters: 35], expected: [Numbers: 11, Letters: 35])
Test 22: OK (received: [Numbers: 8, Letters: 40], expected: [Numbers: 8, Letters: 40])
Test 23: OK (received: [Numbers: 7, Letters: 48], expected: [Numbers: 7, Letters: 48])
Test 24: OK (received: [Numbers: 10, Letters: 41], expected: [Numbers: 10, Letters: 41])
Test 25: OK (received: [Numbers: 8, Letters: 41], expected: [Numbers: 8, Letters: 41])
Test 26: OK (received: [Numbers: 5, Letters: 56], expected: [Numbers: 5, Letters: 56])
Test 27: OK (received: [Numbers: 11, Letters: 47], expected: [Numbers: 11, Letters: 47])
Test 28: OK (received: [Numbers: 6, Letters: 49], expected: [Numbers: 6, Letters: 49])
Test 29: OK (received: [Numbers: 3, Letters: 47], expected: [Numbers: 3, Letters: 47])
Test 30: OK (received: [Numbers: 4, Letters: 32], expected: [Numbers: 4, Letters: 32])
Test 31: OK (received: [Numbers: 6, Letters: 40], expected: [Numbers: 6, Letters: 40])
Test 32: OK (received: [Numbers: 11, Letters: 36], expected: [Numbers: 11, Letters: 36])
Test 33: OK (received: [Numbers: 8, Letters: 52], expected: [Numbers: 8, Letters: 52])
Test 34: OK (received: [Numbers: 9, Letters: 46], expected: [Numbers: 9, Letters: 46])
Test 35: OK (received: [Numbers: 9, Letters: 45], expected: [Numbers: 9, Letters: 45])


ttpo100ajiex@TTPO100AJIEX:/mnt/f/programming/IDZ2$ npm run test_file
> idz2@1.0.0 test_file
> node testing/test_file.js
----------Testing int/solution-c.exe----------
Group 0: PASSED
Group 1: PASSED
Group 2: PASSED
Group 3: PASSED
Group 4: PASSED
Group 5: PASSED
Group 6: PASSED
Group 7: PASSED
Group 8: PASSED
----------Testing int/solution-c-O0.exe----------
Group 0: PASSED
Group 1: PASSED
Group 2: PASSED
Group 3: PASSED
Group 4: PASSED
Group 5: PASSED
Group 6: PASSED
Group 7: PASSED
Group 8: PASSED
----------Testing int/solution-c-O1.exe----------
Group 0: PASSED
Group 1: PASSED
Group 2: PASSED
Group 3: PASSED
Group 4: PASSED
Group 5: PASSED
Group 6: PASSED
Group 7: PASSED
Group 8: PASSED


----------Testing int/solution-c-O2.exe----------
Group 0: PASSED
Group 1: PASSED
Group 2: PASSED
Group 3: PASSED
Group 4: PASSED
Group 5: PASSED
Group 6: PASSED
Group 7: PASSED
Group 8: PASSED
----------Testing int/solution-c-O3.exe----------
Group 0: PASSED
Group 1: PASSED
Group 2: PASSED
Group 3: PASSED
Group 4: PASSED
Group 5: PASSED
Group 6: PASSED
Group 7: PASSED
Group 8: PASSED
----------Testing int/solution-c-Ofast.exe----------
Group 0: PASSED
Group 1: PASSED
Group 2: PASSED
Group 3: PASSED
Group 4: PASSED
Group 5: PASSED
Group 6: PASSED
Group 7: PASSED
Group 8: PASSED
----------Testing int/solution-c-Os.exe----------
Group 0: PASSED
Group 1: PASSED
Group 2: PASSED
Group 3: PASSED
Group 4: PASSED
Group 5: PASSED
Group 6: PASSED
Group 7: PASSED
Group 8: PASSED



Таким образом, все исполняемые файлы работают корректно. Проведём измерение эффективности программ. Полный результат запуска программы `benchmark` представлен в файле `benchmark.txt` директории `report`.

Опции оптимизации	Время выполнения алгоритма (s)			Время выполнения алгоритма с учётом IO (s)		
	10000000	100000000	1073741824	10000000	100000000	1073741824
Нет	0.07270	0.70952	7.54286	0.09226	0.83895	11.69146
-O0	0.07243	0.71023	7.58907	0.09154	0.83965	11.75225
-O1	0.00921	0.09418	0.97257	0.02458	0.20207	5.17970
-O2	0.00938	0.09490	0.99419	0.02617	0.19500	4.97142
-O3	0.00931	0.09411	0.96545	0.02507	0.19643	4.91316
-Ofast	0.01006	0.10136	1.04110	0.02650	0.20237	4.96954
-Os	0.01397	0.13920	1.45908	0.02949	0.23982	5.84259
Ассемблер	0.06081	0.58506	6.05087	0.07941	0.69036	13.31701

Обратим внимание, что полученная ассемблерная программа работает значительно дольше программы на языке `C`, оптимизированной компилятором как с учётом ввода, так и без этого.

Проанализируем оптимизации, произведённые компилятором, которые привели к столь значительному (шестикратному) улучшению времени выполнения. Для этого получим ассемблерный листинг всех программ путём запуска сценария `get_assembly_all` и проанализируем решения компилятора. Учитывая, что все оптимизированные программы затрачивают приблизительно одинаковое время, обратимся, например, к листингу `solution-O1.s`, который должен быть устроен наиболее просто.

Подпрограммы `input` и `output` ожидаемо работают значительно быстрее при сборке утилитой `gcc`, чем путём ручного написания ассемблерного кода в связи с буфферизацией и кэшированием, которые библиотека `libc` активно применяет в своей реализации. Функция `main` также не позволит получить значительный выигрыш во времени выполнения, так как выполняется не

достаточно долго, чтобы это было различимо при замерах (без учёта вызываемых подпрограмм). Следовательно, обратимся к функции `solve`, которая, как показывают замеры, работает примерно в 6 раз быстрее при оптимизации компилятором.

Нетрудно заметить, что компилятор так же организовал цикл по строке с помощью меток `LC18` - `LC21`, в котором последовательно проверяется каждый символ строки и увеличиваются счётчики - значения регистров `r8` и `rsi`. Обратим внимание, что программа использует лишь одно обращение к памяти - `movzx eax, BYTE PTR [rdx]`. Но полученная мной программа также обращается к память лишь единожды на каждой итерации цикла. Значит, проблема не в количестве обращений к памяти.

Далее происходит вычитание 48 из значения кода текущего символа, то есть в регистр `ecx` записывается код текущего символа минус 48. Легко проверить, что 48 - код символа `0`. После этого полученное значение беззнаковым образом сравнивается с числом `9`. Так как из значения кода символа был вычтен код символа `0`, текущий символ является числом тогда и только тогда, когда его “нормированный” код не превышает `9`. Таким образом, инструкция `cmp cl, 9` позволяет определить, является ли символ числом, а инструкция `ja .L19` переходит к проверке, является ли символ буквой только если символ - не число. В ином случае происходит увеличение счётчика `r8` на один.

Проверка, является ли символ числом реализована в 4 инструкции:

```
and eax, -33
sub eax, 65
cmp al, 26
adc rsi, 0
```

Ожидаемо, этот код работает значительно быстрее представленной мной реализации. Разберёмся, что он делает.

1. После передачи управления по метке `.L19` в регистре `rax` находится код текущего символа строки. Значит, эти инструкции позволяют определить, находится ли значиение регистра `rax` (`eax`) в интервале $[65;\ 90]$ или $[97;\ 122]$.
2. $65_{10} = 1000001_2;\ 90_{10} = 1011010_2;\ 97_{10} = 1100001_2;\ 122_{10} = 1111010_2$. Заметим, что коды символов, отвечающих одной латинской букве, отличаются только одним битом: “шестым справа”, а число `-33` представляется в дополнительном коде длины 32 (так как работа происходит с 32-битным регистром `eax`) как `11111111111111111111111111011111`. Следовательно,

операция `and eax, -33` эквивалентна изменению значения шестого справа бита `eax` в `0`, то есть вычитанию из кода символа значения $2^5 = 32$, если символ - заглавная “версия” буквы.

3. Таким образом, после первой инструкции значение регистра `eax` является кодом буквы тогда и только тогда, когда оно лежит в интервале [65; 90]. Это и проверяют инструкции 2 и 3, а инструкция 4 увеличивает значение счётчика `rsi`, если символ действительно является буквой.

В директории `asm` с программой на языке ассемблера создадим копию подпрограммы `solve` под именем `solve_optimized` и модифицруем её с учётом новых идей. Для тестирования новой программы и измерения её эффективности в файл `package.json` были добавлены соответствующие сценарии аналогично описанным ранее.

Соберём программу с помощью запуска сценария `build_asm_optimized` и протестируем её ранее описанными способами. Полные результаты тестирования представлены в файле `test.stdin.asm-optimized.txt` и `test.file.asm-optimized.txt` директории `report`.

```
ttpo100ajiex@TTPO100AJIEX:/mnt/f/programming/IDZ2$ npm run test_stdin

> idz2@1.0.0 test_stdin
> node testing/test_stdin.js

------------------------------------------------------------
-------------------Testing int/solution-asm_optimized.exe-------------------
------------------------------------------------------------
Group 0: ✅ PASSED
    Test 1: ✅ OK (received: [Non-ASCII characters encountered!], expected: [Non-ASCII characters encountered!])
    Test 2: ✅ OK (received: [Numbers: 6, Letters: 14], expected: [Numbers: 6, Letters: 14])
    Test 3: ✅ OK (received: [Numbers: 10, Letters: 19], expected: [Numbers: 10, Letters: 19])
    Test 4: ✅ OK (received: [Numbers: 0, Letters: 51], expected: [Numbers: 0, Letters: 51])
    Test 5: ✅ OK (received: [Numbers: 0, Letters: 0], expected: [Numbers: 0, Letters: 0])
    Test 6: ✅ OK (received: [Numbers: 0, Letters: 0], expected: [Numbers: 0, Letters: 0])
    Test 7: ✅ OK (received: [Numbers: 7, Letters: 12], expected: [Numbers: 7, Letters: 12])
    Test 8: ✅ OK (received: [Numbers: 3, Letters: 4], expected: [Numbers: 3, Letters: 4])
    Test 9: ✅ OK (received: [Numbers: 1, Letters: 0], expected: [Numbers: 1, Letters: 0])
    Test 10: ✅ OK (received: [Numbers: 8, Letters: 467], expected: [Numbers: 8, Letters: 467])

Group 1: ✅ PASSED
    Test 11: ✅ OK (received: [Numbers: 2, Letters: 5], expected: [Numbers: 2, Letters: 5])
    Test 12: ✅ OK (received: [Numbers: 1, Letters: 4], expected: [Numbers: 1, Letters: 4])
    Test 13: ✅ OK (received: [Numbers: 2, Letters: 4], expected: [Numbers: 2, Letters: 4])
    Test 14: ✅ OK (received: [Numbers: 0, Letters: 4], expected: [Numbers: 0, Letters: 4])
    Test 15: ✅ OK (received: [Numbers: 0, Letters: 5], expected: [Numbers: 0, Letters: 5])
    Test 16: ✅ OK (received: [Numbers: 1, Letters: 1], expected: [Numbers: 1, Letters: 1])
    Test 17: ✅ OK (received: [Numbers: 0, Letters: 8], expected: [Numbers: 0, Letters: 8])
    Test 18: ✅ OK (received: [Numbers: 2, Letters: 2], expected: [Numbers: 2, Letters: 2])
    Test 19: ✅ OK (received: [Numbers: 3, Letters: 4], expected: [Numbers: 3, Letters: 4])
    Test 20: ✅ OK (received: [Numbers: 1, Letters: 4], expected: [Numbers: 1, Letters: 4])

Group 2: ✅ PASSED
    Test 21: ✅ OK (received: [Numbers: 11, Letters: 35], expected: [Numbers: 11, Letters: 35])
    Test 22: ✅ OK (received: [Numbers: 8, Letters: 40], expected: [Numbers: 8, Letters: 40])
    Test 23: ✅ OK (received: [Numbers: 7, Letters: 48], expected: [Numbers: 7, Letters: 48])
    Test 24: ✅ OK (received: [Numbers: 10, Letters: 41], expected: [Numbers: 10, Letters: 41])
    Test 25: ✅ OK (received: [Numbers: 8, Letters: 41], expected: [Numbers: 8, Letters: 41])
    Test 26: ✅ OK (received: [Numbers: 5, Letters: 56], expected: [Numbers: 5, Letters: 56])
    Test 27: ✅ OK (received: [Numbers: 11, Letters: 47], expected: [Numbers: 11, Letters: 47])
    Test 28: ✅ OK (received: [Numbers: 6, Letters: 49], expected: [Numbers: 6, Letters: 49])
    Test 29: ✅ OK (received: [Numbers: 3, Letters: 47], expected: [Numbers: 3, Letters: 47])
    Test 30: ✅ OK (received: [Numbers: 4, Letters: 32], expected: [Numbers: 4, Letters: 32])
    Test 31: ✅ OK (received: [Numbers: 6, Letters: 40], expected: [Numbers: 6, Letters: 40])
    Test 32: ✅ OK (received: [Numbers: 11, Letters: 36], expected: [Numbers: 11, Letters: 36])
    Test 33: ✅ OK (received: [Numbers: 8, Letters: 52], expected: [Numbers: 8, Letters: 52])
    Test 34: ✅ OK (received: [Numbers: 9, Letters: 46], expected: [Numbers: 9, Letters: 46])
    Test 35: ✅ OK (received: [Numbers: 9, Letters: 45], expected: [Numbers: 9, Letters: 45])
```

```
ttpo100ajiex@TTPO100AJIEX:/mnt/f/programming/IDZ2$ npm run test_file

> idz2@1.0.0 test_file
> node testing/test_file.js

------------------------------------------------------------
-------------------Testing int/solution-asm_optimized.exe-------------------
------------------------------------------------------------
Group 0: ✅ PASSED
    Test 1: ✅ OK (received: [Non-ASCII characters encountered!], expected: [Non-ASCII characters encountered!])
    Test 2: ✅ OK (received: [Numbers: 6, Letters: 14], expected: [Numbers: 6, Letters: 14])
    Test 3: ✅ OK (received: [Numbers: 10, Letters: 19], expected: [Numbers: 10, Letters: 19])
    Test 4: ✅ OK (received: [Numbers: 0, Letters: 51], expected: [Numbers: 0, Letters: 51])
    Test 5: ✅ OK (received: [Numbers: 0, Letters: 0], expected: [Numbers: 0, Letters: 0])
    Test 6: ✅ OK (received: [Numbers: 0, Letters: 0], expected: [Numbers: 0, Letters: 0])
    Test 7: ✅ OK (received: [Numbers: 7, Letters: 12], expected: [Numbers: 7, Letters: 12])
    Test 8: ✅ OK (received: [Numbers: 3, Letters: 4], expected: [Numbers: 3, Letters: 4])
    Test 9: ✅ OK (received: [Numbers: 1, Letters: 0], expected: [Numbers: 1, Letters: 0])
    Test 10: ✅ OK (received: [Numbers: 8, Letters: 467], expected: [Numbers: 8, Letters: 467])

Group 1: ✅ PASSED
    Test 11: ✅ OK (received: [Numbers: 2, Letters: 5], expected: [Numbers: 2, Letters: 5])
    Test 12: ✅ OK (received: [Numbers: 1, Letters: 4], expected: [Numbers: 1, Letters: 4])
    Test 13: ✅ OK (received: [Numbers: 2, Letters: 4], expected: [Numbers: 2, Letters: 4])
    Test 14: ✅ OK (received: [Numbers: 0, Letters: 4], expected: [Numbers: 0, Letters: 4])
    Test 15: ✅ OK (received: [Numbers: 0, Letters: 5], expected: [Numbers: 0, Letters: 5])
    Test 16: ✅ OK (received: [Numbers: 1, Letters: 1], expected: [Numbers: 1, Letters: 1])
    Test 17: ✅ OK (received: [Numbers: 0, Letters: 8], expected: [Numbers: 0, Letters: 8])
    Test 18: ✅ OK (received: [Numbers: 2, Letters: 2], expected: [Numbers: 2, Letters: 2])
    Test 19: ✅ OK (received: [Numbers: 3, Letters: 4], expected: [Numbers: 3, Letters: 4])
    Test 20: ✅ OK (received: [Numbers: 1, Letters: 4], expected: [Numbers: 1, Letters: 4])

Group 2: ✅ PASSED
    Test 21: ✅ OK (received: [Numbers: 11, Letters: 35], expected: [Numbers: 11, Letters: 35])
    Test 22: ✅ OK (received: [Numbers: 8, Letters: 40], expected: [Numbers: 8, Letters: 40])
    Test 23: ✅ OK (received: [Numbers: 7, Letters: 48], expected: [Numbers: 7, Letters: 48])
    Test 24: ✅ OK (received: [Numbers: 10, Letters: 41], expected: [Numbers: 10, Letters: 41])
    Test 25: ✅ OK (received: [Numbers: 8, Letters: 41], expected: [Numbers: 8, Letters: 41])
    Test 26: ✅ OK (received: [Numbers: 5, Letters: 56], expected: [Numbers: 5, Letters: 56])
    Test 27: ✅ OK (received: [Numbers: 11, Letters: 47], expected: [Numbers: 11, Letters: 47])
    Test 28: ✅ OK (received: [Numbers: 6, Letters: 49], expected: [Numbers: 6, Letters: 49])
    Test 29: ✅ OK (received: [Numbers: 3, Letters: 47], expected: [Numbers: 3, Letters: 47])
    Test 30: ✅ OK (received: [Numbers: 4, Letters: 32], expected: [Numbers: 4, Letters: 32])
    Test 31: ✅ OK (received: [Numbers: 6, Letters: 40], expected: [Numbers: 6, Letters: 40])
    Test 32: ✅ OK (received: [Numbers: 11, Letters: 36], expected: [Numbers: 11, Letters: 36])
    Test 33: ✅ OK (received: [Numbers: 8, Letters: 52], expected: [Numbers: 8, Letters: 52])
    Test 34: ✅ OK (received: [Numbers: 9, Letters: 46], expected: [Numbers: 9, Letters: 46])
    Test 35: ✅ OK (received: [Numbers: 9, Letters: 45], expected: [Numbers: 9, Letters: 45])
```

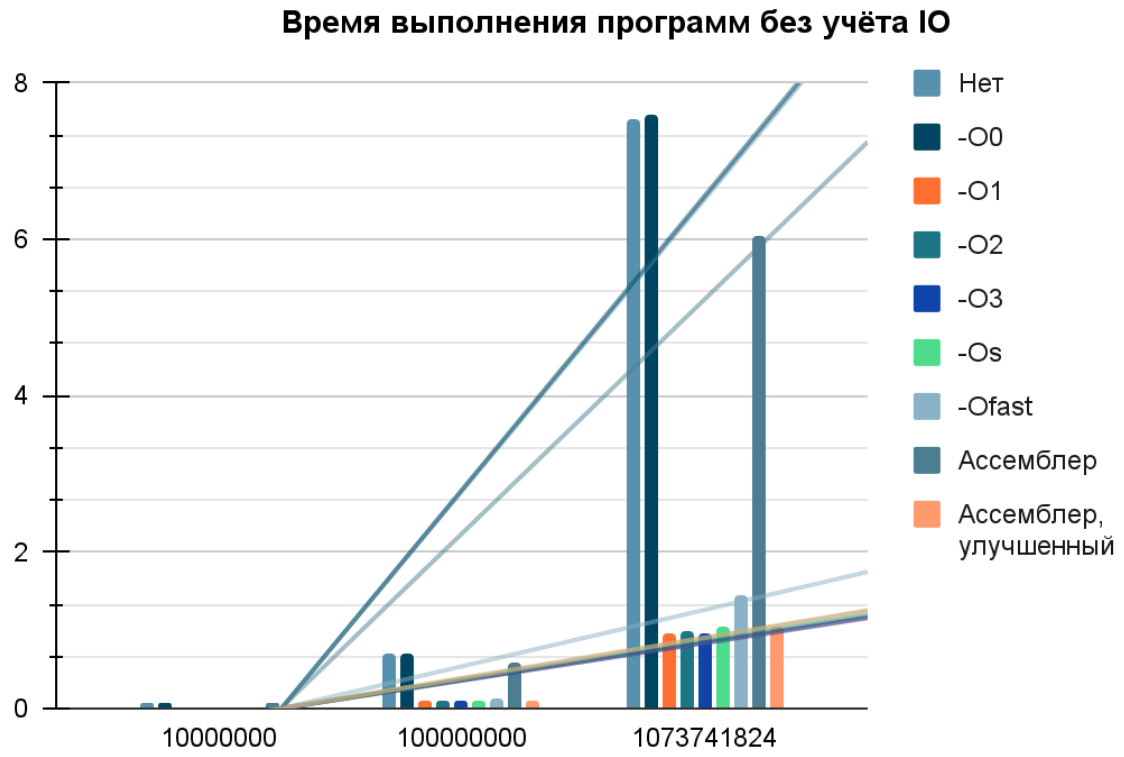
Результат измерения производительности получен с помощью программы `benchmark` и представлен в файле `benchmark.asm-optimized.txt` директории `report`.

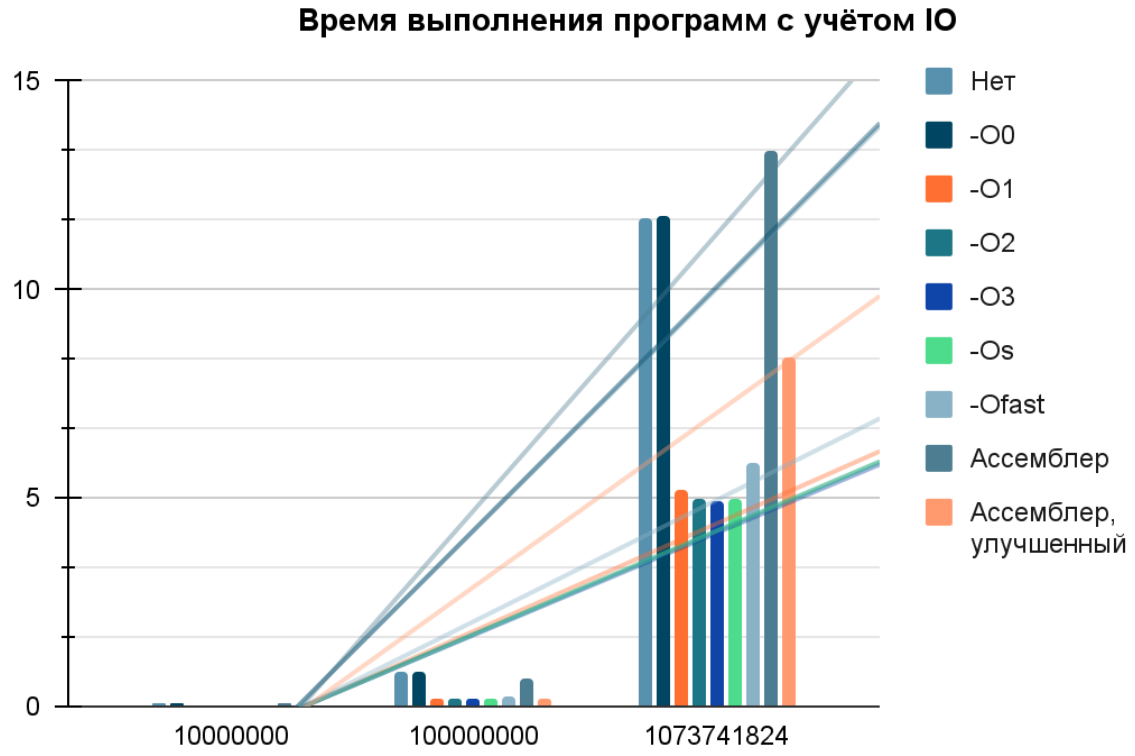
```
ttpo100ajiex@TTPO100AJIEX:/mnt/f/programming/IDZ2$ npm run benchmark

> idz2@1.0.0 benchmark
> node testing/benchmark.js

------------------------------------------------------------
-------------------Benchmarking int/solution-asm_optimized.exe-------------------
------------------------------------------------------------
String length: 10.
    Average solution CPU time: 0.00024ms = 0.00000s
    Average IO CPU time: 5.81891ms = 0.00582s
String length: 100.
    Average solution CPU time: 0.00036ms = 0.00000s
    Average IO CPU time: 5.60301ms = 0.00560s
String length: 1000.
    Average solution CPU time: 0.00148ms = 0.00000s
    Average IO CPU time: 6.26794ms = 0.00627s
String length: 10000.
    Average solution CPU time: 0.01126ms = 0.00001s
    Average IO CPU time: 6.11078ms = 0.00611s
String length: 100000.
    Average solution CPU time: 0.11263ms = 0.00011s
    Average IO CPU time: 6.13865ms = 0.00614s
String length: 1000000.
    Average solution CPU time: 1.02493ms = 0.00102s
    Average IO CPU time: 8.16133ms = 0.00816s
String length: 10000000.
    Average solution CPU time: 10.28244ms = 0.01028s
    Average IO CPU time: 28.17614ms = 0.02818s
String length: 100000000.
    Average solution CPU time: 100.33475ms = 0.10033s
    Average IO CPU time: 205.65502ms = 0.20566s
String length: 1073741824.
    Average solution CPU time: 1048.16872ms = 1.04817s
    Average IO CPU time: 8360.04547ms = 8.36005s
```

Опции оптимизации	Время выполнения алгоритма (s)			Время выполнения алгоритма с учётом IO (s)		
	10000000	100000000	1073741824	10000000	100000000	1073741824
Ассемблер, улучшенный	0.01028	0.10033	1.04817	0.02818	0.20566	8.36005

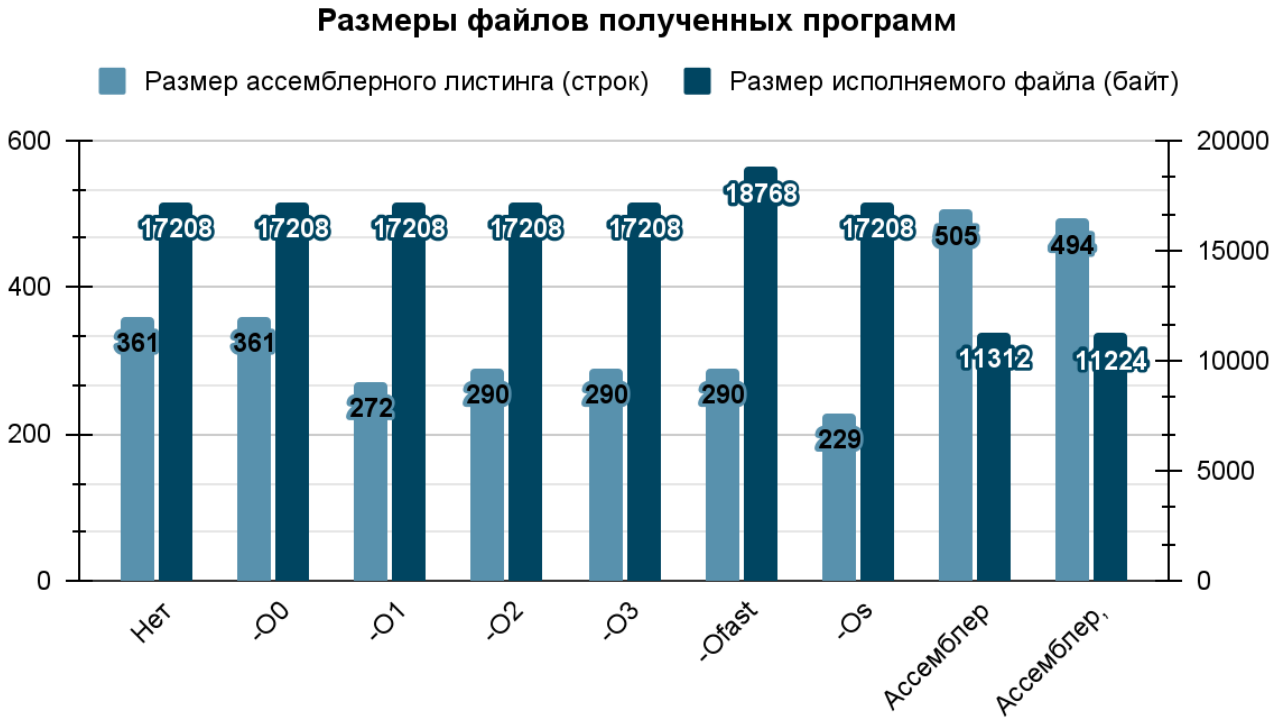




Заметим, что время работы алгоритма действительно улучшилось и лишь незначительно отличается от лучших результатов, полученных при сборке с помощью утилиты `gcc`.

Проведём сравнение размера ассемблерного листинга и исполняемого файла для полученных программ.

Опции оптимизации	Размер ассемблерного листинга (строк)	Размер исполняемого файла (байт)
Нет	361	17208
-O0	361	17208
-O1	272	17208
-O2	290	17208
-O3	290	17208
-Ofast	290	18768
-Os	229	17208
Ассемблер	$201 + 115 + 110 + 41 + 38 = 505$	11312
Ассемблер, улучшенный	$201 + 115 + 110 + 30 + 38 = 494$	11224



Нетрудно заметить, что ассемблерный листинг наименьшего размера, ожидаемо, получается при использовании опции `Os`, которая в первую очередь применяет оптимизации по размеру итогового файла, что заметно сказывается на времени выполнения.

Наименьший исполняемый файл же получается при сборке самостоятельно написанной ассемблерной программы, что явно выделяется среди полученных результатов. Тем не менее размер программы не достаточно велик, чтобы оценить разницу в размере исполняемого файла, получаемого компилятором `gcc` при сборке программы с различными опциями оптимизации: в связи с выравниванием, которое операционные системы применяют к исполняемым файлам, почти все значения получаются одинаковые и лишь одно, соответствующее опции `Ofast`, имеет незначительно больший размер, что ожидаемо: данная опция не применяет оптимизации по размеру вовсе и делает уклон на оптимизации по скорости выполнения.

Таким образом, программа на ассемблере, написанная вручную, как минимум не хуже полученной компилятором: при незначительном проигрыше по времени выполнения, который находится в пределах погрешности измерений, размер полученного исполняемого файла оказался значительно меньше.