Национальный исследовательский университет “Высшая школа экономики”.

Факультет компьютерных наук. Программная инженерия.

Архитектура вычислительных систем.

Индивидуальное домашнее задание №1 студента группы БПИ213 Абрамова Александра Сергеевича.

Вариант 28.

Решение задачи на языке C представлено в файле `4-solution.c`. Для детерминированности реализации и улучшения надёжности программы было установлено ограничение на длину массива в `16777214` элементов.

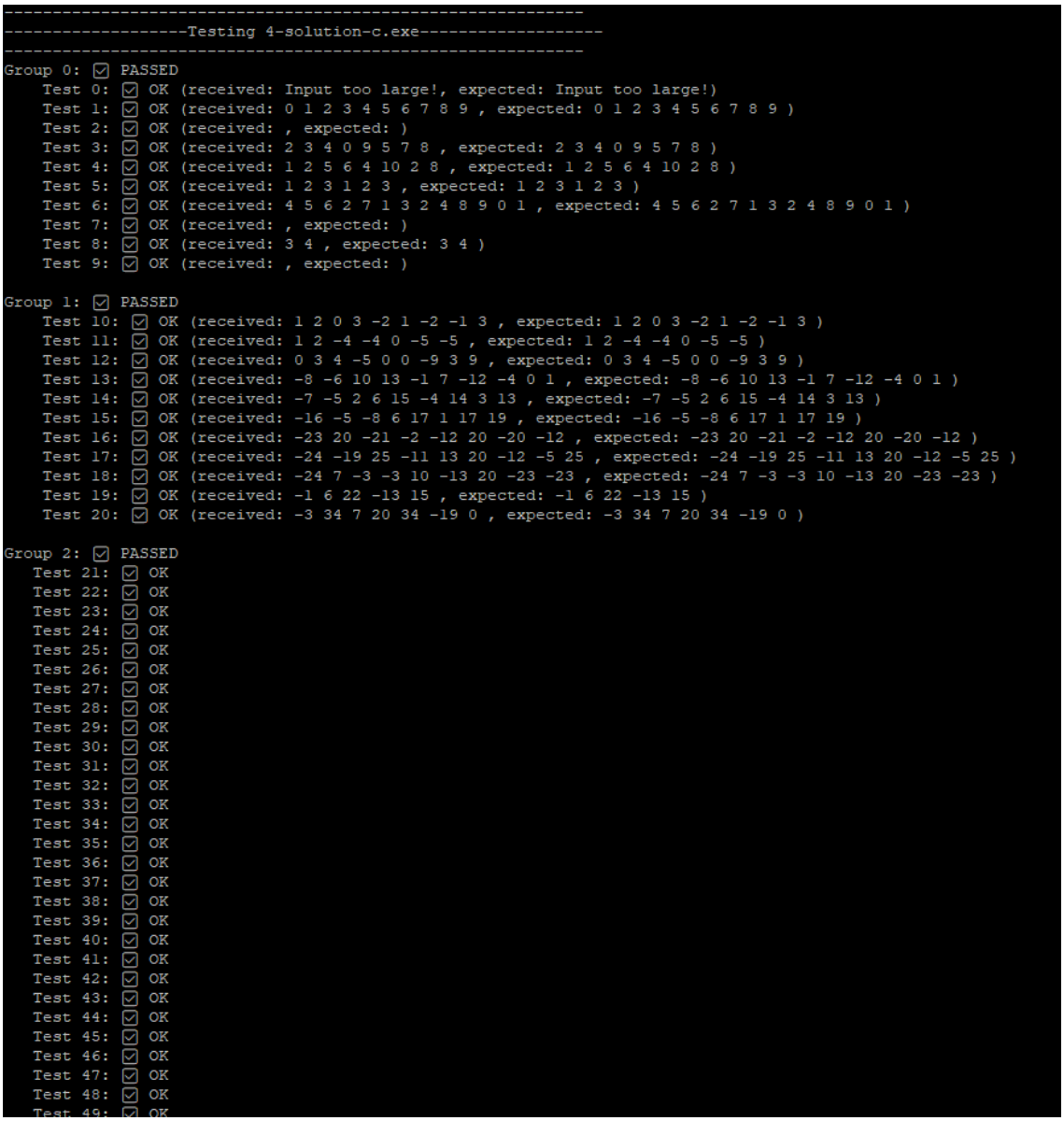
Для удобства компиляции программ был создан `shell`-скрипт `4-compile.sh`, собирающий программу `4-solution.c` в исполняемый файл `4-solution-c.exe` с помощью утилиты `gcc`.

Для проверки корректности работы программы был создан ряд тестов, размещённых в папке `tests`. Каждый тест состоит из набора входных (файл с расширением `in`) и выходных (файл с расширением `out`) данных. Тесты `0-9` созданы вручную для проверки правильности ответа, который выводит программа, тесты `10-120` сгенерированы программой `gen.js` и предназначены для тестирования эффективности и надёжности алгоритма. Выходных данные тестов `10-20` были дополнительно вручную проверены на правильность.

Номер группы тестов	Номера тестов	Длина вводимого массива (n)	Комментарий
0	0-9	-	Тестирование правильности ответа
1	10-20	10	Тестирование эффективности алгоритма. Элементы массива `val` удовлетворяет условию $\lvert val\rvert < n * \frac{\text{номер\_теста\_в\_группе}}{3}$
2	21-50	100	
3	51-75	1000	
4	76-100	10000	
5	101-110	100000	
6	111-115	1000000	
7	116-120	10000000	

Для удобства тестирования была создана программа `test(run.js)`, которая запускает указанные исполняемые файлы на всех тестовых входных данных, сравнивает вывод программы с корректными выходными данными и печатает вердикт в `stdout`.

Как показало тестирование, программа работает корректно на полученных случайных наборах входных данных.



Преобразуем код программы на язык ассемблера с помощью утилиты `gcc` без оптимизирующих и отладочных опций. Для этого был создан `shell`-скрипт `4-get_assembly.sh`, в результате запуска которого был создан файл `4-solution.s`. Для удобства работы копия этого файла была сохранена под именем `4-solution-refactored.s`, с которым и будет производиться дальнейшая работа.

Для удобства тестирования изменений в файл `4-compile.sh` были добавлены команды сборки модифицированного ассемблерного листинга из файла `4-solution-refactored.s` в исполняемый файл `4-solution-asm.exe` с помощью утилит `as` и `gcc`.

При модификации ассемблерного листинга были удалены лишние инструкции и макросы, а также добавлены комментарии ко всем командам:

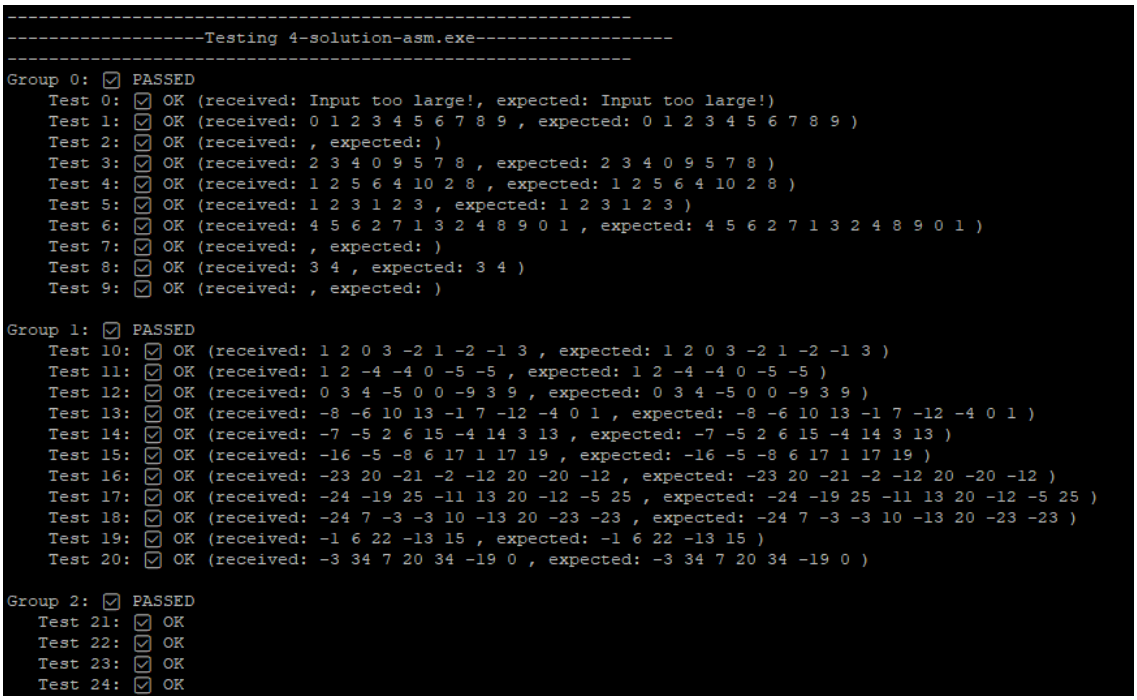
1. Директивы `.file`, `.size`, `.ident`, `.section .note.GNU-stack,"",@progbits`, `.type`, `.local` и `.align` имеют исключительно информационную функцию, в связи с чем были удалены.
2. Для всех функций, кроме `main`, была удалена директива `.globl`, открывающая видимость символа для других модулей трансляции.
3. Были удалены инструкции `nop`, не выполняющие никаких операций.
4. Переменные `ulli_input_template`, `lli_input_template`, `lli_output_template` и `too_long_array_error` объявлены в секции для неизменяемых данных `rodata`. Для этого компилятор разместил данные по некоторому адресу и установил перед ними метку с соответствующим переменной именем. Таким образом, адреса меток совпадают с адресами начал одноимённых строковых переменных.
5. Массивы `A`, `B`, а также переменные `A_length` и `B_length` расположены в секции `data`, и “созданы” с помощью директивы `.comm name, size, alignment`. Таким образом, программа дважды аллоцирует по `134217728` байт для массивов и по `8` байт для хранения их длин, после чего обращение к памяти производится по указанному имени.

Для проверки корректности изменений обе программы (исходная C-программа и модифицированная ассемблерная) были собраны и протестированы ранее описанным способом с помощью программы `test`.

```
------------------------------------------------------------
-------------------Testing 4-solution-c.exe-------------------
------------------------------------------------------------
Group 0: ☑ PASSED
    Test 0: ☑ OK (received: Input too large!, expected: Input too large!)
    Test 1: ☑ OK (received: 0 1 2 3 4 5 6 7 8 9 , expected: 0 1 2 3 4 5 6 7 8 9 )
    Test 2: ☑ OK (received: , expected: )
    Test 3: ☑ OK (received: 2 3 4 0 9 5 7 8 , expected: 2 3 4 0 9 5 7 8 )
    Test 4: ☑ OK (received: 1 2 5 6 4 10 2 8 , expected: 1 2 5 6 4 10 2 8 )
    Test 5: ☑ OK (received: 1 2 3 1 2 3 , expected: 1 2 3 1 2 3 )
    Test 6: ☑ OK (received: 4 5 6 2 7 1 3 2 4 8 9 0 1 , expected: 4 5 6 2 7 1 3 2 4 8 9 0 1 )
    Test 7: ☑ OK (received: , expected: )
    Test 8: ☑ OK (received: 3 4 , expected: 3 4 )
    Test 9: ☑ OK (received: , expected: )

Group 1: ☑ PASSED
    Test 10: ☑ OK (received: 1 2 0 3 -2 1 -2 -1 3 , expected: 1 2 0 3 -2 1 -2 -1 3 )
    Test 11: ☑ OK (received: 1 2 -4 -4 0 -5 -5 , expected: 1 2 -4 -4 0 -5 -5 )
    Test 12: ☑ OK (received: 0 3 4 -5 0 0 -9 3 9 , expected: 0 3 4 -5 0 0 -9 3 9 )
    Test 13: ☑ OK (received: -8 -6 10 13 -1 7 -12 -4 0 1 , expected: -8 -6 10 13 -1 7 -12 -4 0 1 )
    Test 14: ☑ OK (received: -7 -5 2 6 15 -4 14 3 13 , expected: -7 -5 2 6 15 -4 14 3 13 )
    Test 15: ☑ OK (received: -16 -5 -8 6 17 1 17 19 , expected: -16 -5 -8 6 17 1 17 19 )
    Test 16: ☑ OK (received: -23 20 -21 -2 -12 20 -20 -12 , expected: -23 20 -21 -2 -12 20 -20 -12 )
    Test 17: ☑ OK (received: -24 -19 25 -11 13 20 -12 -5 25 , expected: -24 -19 25 -11 13 20 -12 -5 25 )
    Test 18: ☑ OK (received: -24 7 -3 -3 10 -13 20 -23 -23 , expected: -24 7 -3 -3 10 -13 20 -23 -23 )
    Test 19: ☑ OK (received: -1 6 22 -13 15 , expected: -1 6 22 -13 15 )
    Test 20: ☑ OK (received: -3 34 7 20 34 -19 0 , expected: -3 34 7 20 34 -19 0 )

Group 2: ☑ PASSED
   Test 21: ☑ OK
   Test 22: ☑ OK
   Test 23: ☑ OK
```



Как нетрудно заметить, обе программы успешно проходят тестирование, возвращая одинаковые результаты на всех тестах.

Далее программа на языке C была модифицирована для удовлетворения критериям задания (результат сохранён в файле `5-solution.c`):

1. Убраны переменные, у которых область видимости - весь модуль трансляции:
   a. Область видимости переменных `A`, `B`, `A_length` и `B_length` изменена на локальную для функции `main` (таким образом, память для массивов `A` и `B` выделяется не в секции `data`, а на стеке в кадре функции `main`). Подпрограммы `input`, `solve` и `output` теперь получают необходимые для работы данные как параметры и возвращают результат работы.
   b. Строковые переменные `ulli_input_template`, `lli_input_template`, `lli_output_template` и `too_long_array_error` явно заменены в коде программы на соответствующие строковые литералы.
2. Индексация переменных в массиве `B` изменена на `1…n` вместо `0…n-1` для схожести с массивом `A`.
3. Внесены другие незначительные изменения для улучшения “красоты” кода с учётом новых возможностей функций.

Для проверки корректности работы изменённой программы был создан файл `5-compile.sh`, собирающий `5-solution.c` в `5-solution-c.exe`, а также был модифицирован `CONSTS.js` тестировщика `test` для проверки новой программы. Для удобства тестирования на устройстве с ОС `linux` (`Debian 10`) ограничение на длину массива было уменьшено до `131072` элементов и были убраны группы тестов 6 и 7.

Сборка и запуск программы показали, что она работает корректно:

```
------------------------------------------------------------
------------------Testing 5-solution-c.exe------------------
------------------------------------------------------------
Group 0: ☑ PASSED
    Test 0: ☑ OK (received: Input too large!, expected: Input too large!)
    Test 1: ☑ OK (received: 0 1 2 3 4 5 6 7 8 9 , expected: 0 1 2 3 4 5 6 7 8 9 )
    Test 2: ☑ OK (received: , expected: )
    Test 3: ☑ OK (received: 2 3 4 0 9 5 7 8 , expected: 2 3 4 0 9 5 7 8 )
    Test 4: ☑ OK (received: 1 2 5 6 4 10 2 8 , expected: 1 2 5 6 4 10 2 8 )
    Test 5: ☑ OK (received: 1 2 3 1 2 3 , expected: 1 2 3 1 2 3 )
    Test 6: ☑ OK (received: 4 5 6 2 7 1 3 2 4 8 9 0 1 , expected: 4 5 6 2 7 1 3 2 4 8 9 0 1 )
    Test 7: ☑ OK (received: , expected: )
    Test 8: ☑ OK (received: 3 4 , expected: 3 4 )
    Test 9: ☑ OK (received: , expected: )

Group 1: ☑ PASSED
    Test 10: ☑ OK (received: 1 2 0 3 -2 1 -2 -1 3 , expected: 1 2 0 3 -2 1 -2 -1 3 )
    Test 11: ☑ OK (received: 1 2 -4 -4 0 -5 -5 , expected: 1 2 -4 -4 0 -5 -5 )
    Test 12: ☑ OK (received: 0 3 4 -5 0 0 -9 3 9 , expected: 0 3 4 -5 0 0 -9 3 9 )
    Test 13: ☑ OK (received: -8 -6 10 13 -1 7 -12 -4 0 1 , expected: -8 -6 10 13 -1 7 -12 -4 0 1 )
    Test 14: ☑ OK (received: -7 -5 2 6 15 -4 14 3 13 , expected: -7 -5 2 6 15 -4 14 3 13 )
    Test 15: ☑ OK (received: -16 -5 -8 6 17 1 17 19 , expected: -16 -5 -8 6 17 1 17 19 )
    Test 16: ☑ OK (received: -23 20 -21 -2 -12 20 -20 -12 , expected: -23 20 -21 -2 -12 20 -20 -12 )
    Test 17: ☑ OK (received: -24 -19 25 -11 13 20 -12 -5 25 , expected: -24 -19 25 -11 13 20 -12 -5 25 )
    Test 18: ☑ OK (received: -24 7 -3 -3 10 -13 20 -23 -23 , expected: -24 7 -3 -3 10 -13 20 -23 -23 )
    Test 19: ☑ OK (received: -1 6 22 -13 15 , expected: -1 6 22 -13 15 )
    Test 20: ☑ OK (received: -3 34 7 20 34 -19 0 , expected: -3 34 7 20 34 -19 0 )

Group 2: ☑ PASSED
   Test 21: ☑ OK
```

Для анализа ассемблерного листинга новой программы был создан `shell`-скрипт `5-get_assembly.sh` на основе `4-get_assembly.sh`. В результате его запуска был создан файл `5-solution.s`, который был модифицирован аналогично описанному ранее для программы `4-solution.s`. Результат был сохранён в файл `5-solution-refactored.s`. Для проверки корректности изменений были внесены соответствующие модификации в `5-compile.sh` и программу `test`. Запуск показал, что внесённые в ассемблерный листинг изменения не повлияли на поведение программы.

```
------------------------------------------------------------
-------------------Testing 5-solution-asm.exe-------------------
------------------------------------------------------------
Group 0: ☑ PASSED
    Test 0: ☑ OK (received: Input too large!, expected: Input too large!)
    Test 1: ☑ OK (received: 0 1 2 3 4 5 6 7 8 9 , expected: 0 1 2 3 4 5 6 7 8 9 )
    Test 2: ☑ OK (received: , expected: )
    Test 3: ☑ OK (received: 2 3 4 0 9 5 7 8 , expected: 2 3 4 0 9 5 7 8 )
    Test 4: ☑ OK (received: 1 2 5 6 4 10 2 8 , expected: 1 2 5 6 4 10 2 8 )
    Test 5: ☑ OK (received: 1 2 3 1 2 3 , expected: 1 2 3 1 2 3 )
    Test 6: ☑ OK (received: 4 5 6 2 7 1 3 2 4 8 9 0 1 , expected: 4 5 6 2 7 1 3 2 4 8 9 0 1 )
    Test 7: ☑ OK (received: , expected: )
    Test 8: ☑ OK (received: 3 4 , expected: 3 4 )
    Test 9: ☑ OK (received: , expected: )

Group 1: ☑ PASSED
    Test 10: ☑ OK (received: 1 2 0 3 -2 1 -2 -1 3 , expected: 1 2 0 3 -2 1 -2 -1 3 )
    Test 11: ☑ OK (received: 1 2 -4 -4 0 -5 -5 , expected: 1 2 -4 -4 0 -5 -5 )
    Test 12: ☑ OK (received: 0 3 4 -5 0 0 -9 3 9 , expected: 0 3 4 -5 0 0 -9 3 9 )
    Test 13: ☑ OK (received: -8 -6 10 13 -1 7 -12 -4 0 1 , expected: -8 -6 10 13 -1 7 -12 -4 0 1 )
    Test 14: ☑ OK (received: -7 -5 2 6 15 -4 14 3 13 , expected: -7 -5 2 6 15 -4 14 3 13 )
    Test 15: ☑ OK (received: -16 -5 -8 6 17 1 17 19 , expected: -16 -5 -8 6 17 1 17 19 )
    Test 16: ☑ OK (received: -23 20 -21 -2 -12 20 -20 -12 , expected: -23 20 -21 -2 -12 20 -20 -12 )
    Test 17: ☑ OK (received: -24 -19 25 -11 13 20 -12 -5 25 , expected: -24 -19 25 -11 13 20 -12 -5 25 )
    Test 18: ☑ OK (received: -24 7 -3 -3 10 -13 20 -23 -23 , expected: -24 7 -3 -3 10 -13 20 -23 -23 )
    Test 19: ☑ OK (received: -1 6 22 -13 15 , expected: -1 6 22 -13 15 )
    Test 20: ☑ OK (received: -3 34 7 20 34 -19 0 , expected: -3 34 7 20 34 -19 0 )

Group 2: ☑ PASSED
   Test 21: ☑ OK
```

По соглашению о вызовах архитектуры `x64`, любая функция должна сохранять значения регистров `rbx` и `r12-r15` во время работы. Воспользуемся этим для избавления от лишних обращений к памяти. Для этого будем сохранять значения некоторых из указанных регистров в кадре функции на стеке и восстанавливать их перед выходом.

С учётом этого для оптимизации был проведён рефакторинг программы, полученной ранее, за счёт максимального использования регистров процессора. Для этого был создан файл `6-solution-refactored.s` как копия `5-solution-refactored.s`, и было проделано следующее:

1. Для реализации подпрограммы `input` достаточно использования трёх регистров, но функция `scanf`, вызываемая при работе `input`, по соглашению о вызовах требует выравнивания стека по 16-байтной границе, поэтому помимо сохранения регистров (что займёт 24 байта), на стеке требуется дополнительно выделить 8 байт.
2. Так как функция `solve` не делает никаких вызовов, выравнивание стека по 16-байтной границе не требуется, а также возможно беспрепятственное использование всех регистров, включая те, которые не должны сохраняться вызванной стороной. Это позволяет полностью уйти от использования памяти для локальных переменных в этой функции.

3. Подпрограмма `output` полагается на `printf`, поэтому для достижения минимального использования памяти допустимо использование только регистров, сохраняемых вызванной функцией. Таким образом, аналогично функции `input`, в `output` были использованы три “callee-saved” регистра.
4. В функции `main` был убран ряд лишних копирований значений между регистрами.

Для тестирования изменений был создан скрипт `6-compile.sh` на основе `5-compile.sh` и отредактириван `CONSTS.js` программы `test`. Как нетрудно заметить, программа работает корректно.

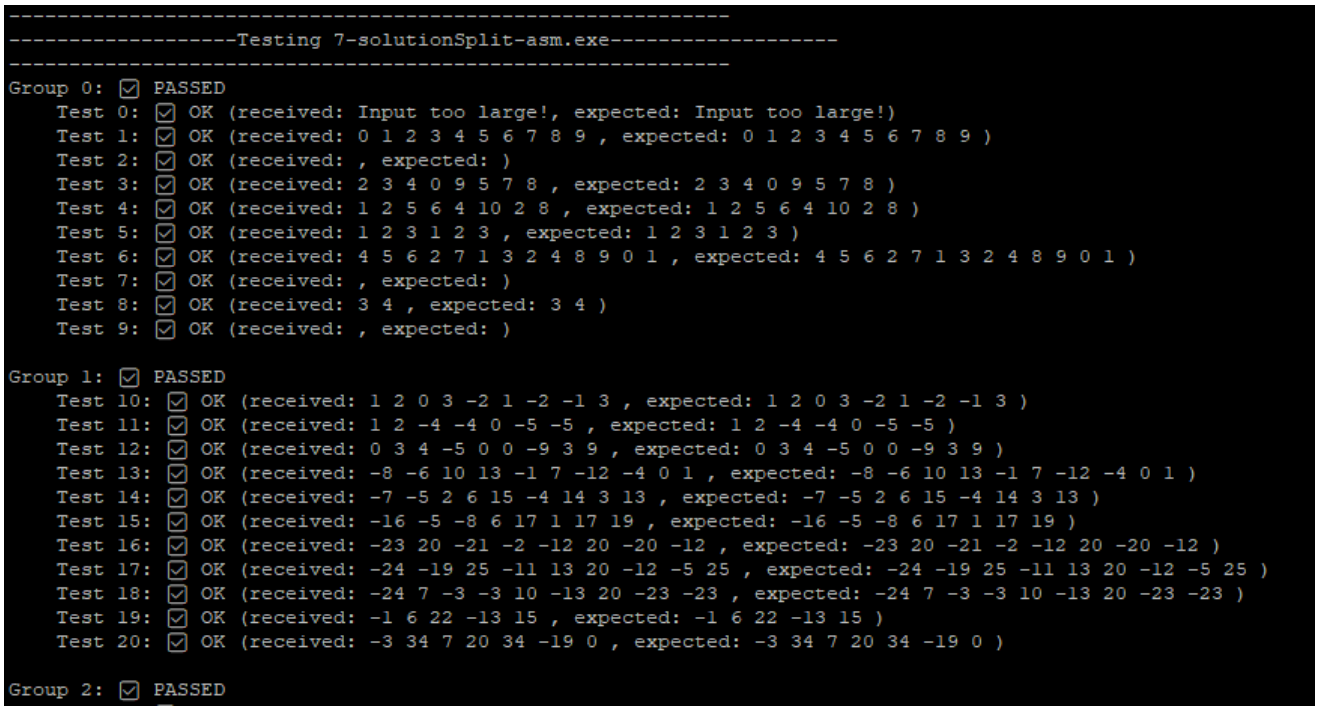
```
------------------------------------------------------------
------------------Testing 6-solution-asm.exe------------------
------------------------------------------------------------
Group 0: ☑ PASSED
    Test 0: ☑ OK (received: Input too large!, expected: Input too large!)
    Test 1: ☑ OK (received: 0 1 2 3 4 5 6 7 8 9 , expected: 0 1 2 3 4 5 6 7 8 9 )
    Test 2: ☑ OK (received: , expected: )
    Test 3: ☑ OK (received: 2 3 4 0 9 5 7 8 , expected: 2 3 4 0 9 5 7 8 )
    Test 4: ☑ OK (received: 1 2 5 6 4 10 2 8 , expected: 1 2 5 6 4 10 2 8 )
    Test 5: ☑ OK (received: 1 2 3 1 2 3 , expected: 1 2 3 1 2 3 )
    Test 6: ☑ OK (received: 4 5 6 2 7 1 3 2 4 8 9 0 1 , expected: 4 5 6 2 7 1 3 2 4 8 9 0 1 )
    Test 7: ☑ OK (received: , expected: )
    Test 8: ☑ OK (received: 3 4 , expected: 3 4 )
    Test 9: ☑ OK (received: , expected: )

Group 1: ☑ PASSED
    Test 10: ☑ OK (received: 1 2 0 3 -2 1 -2 -1 3 , expected: 1 2 0 3 -2 1 -2 -1 3 )
    Test 11: ☑ OK (received: 1 2 -4 -4 0 -5 -5 , expected: 1 2 -4 -4 0 -5 -5 )
    Test 12: ☑ OK (received: 0 3 4 -5 0 0 -9 3 9 , expected: 0 3 4 -5 0 0 -9 3 9 )
    Test 13: ☑ OK (received: -8 -6 10 13 -1 7 -12 -4 0 1 , expected: -8 -6 10 13 -1 7 -12 -4 0 1 )
    Test 14: ☑ OK (received: -7 -5 2 6 15 -4 14 3 13 , expected: -7 -5 2 6 15 -4 14 3 13 )
    Test 15: ☑ OK (received: -16 -5 -8 6 17 1 17 19 , expected: -16 -5 -8 6 17 1 17 19 )
    Test 16: ☑ OK (received: -23 20 -21 -2 -12 20 -20 -12 , expected: -23 20 -21 -2 -12 20 -20 -12 )
    Test 17: ☑ OK (received: -24 -19 25 -11 13 20 -12 -5 25 , expected: -24 -19 25 -11 13 20 -12 -5 25 )
    Test 18: ☑ OK (received: -24 7 -3 -3 10 -13 20 -23 -23 , expected: -24 7 -3 -3 10 -13 20 -23 -23 )
    Test 19: ☑ OK (received: -1 6 22 -13 15 , expected: -1 6 22 -13 15 )
    Test 20: ☑ OK (received: -3 34 7 20 34 -19 0 , expected: -3 34 7 20 34 -19 0 )

Group 2: ☑ PASSED
   Test 21: ☑ OK
   Test 22: ☑ OK
   Test 23: ☑ OK
   Test 24: ☑ OK
   Test 25: ☑ OK
   Test 26: ☑ OK
   Test 27: ☑ OK
   Test 28: ☑ OK
   Test 29: ☑ OK
```

Для реализации программы на ассемблере, полученной после рефакторинга, в виде двух или более единиц компиляции, были созданы файлы `7-solutionSplit-input.s`, `7-solutionSplit-solve.s`, `7-solutionSplit-output.s` и `7-solutionSplit-main.s`, в которых размещены соответствующие функции. Для успешной линковки программы все функции были указаны как глобальные добавлением директивы `.globl`. Для удобной сборки программы был создан `shell`-скрипт `7-compile.sh`, который компилирует все созданные элементы в объектные

файлы с помощью утилиты `as`, а затем линкует в один исполняемый файл с помощью утилиты `gcc`. Для проверки корректности работы новой программы была также аналогично описанному ранее модифицирована программа `test`. Как показал запуск, корректность работы алгоритма нарушена не была.



Для добавления возможности задания файлов с исходными данными и файла для вывода результатов с использованием аргументов командной строки были проведены дополнительные модификации обеих программ (C и ассемблерной). Для однозначности трактовки задания было принято следующее желаемое поведение программы. При запуске в качестве аргументов командной строки программе должны передаваться следующие данные:

`название_исполняемого_файла флаг_ввода файл_ввода флаг_вывода файл_вывода`

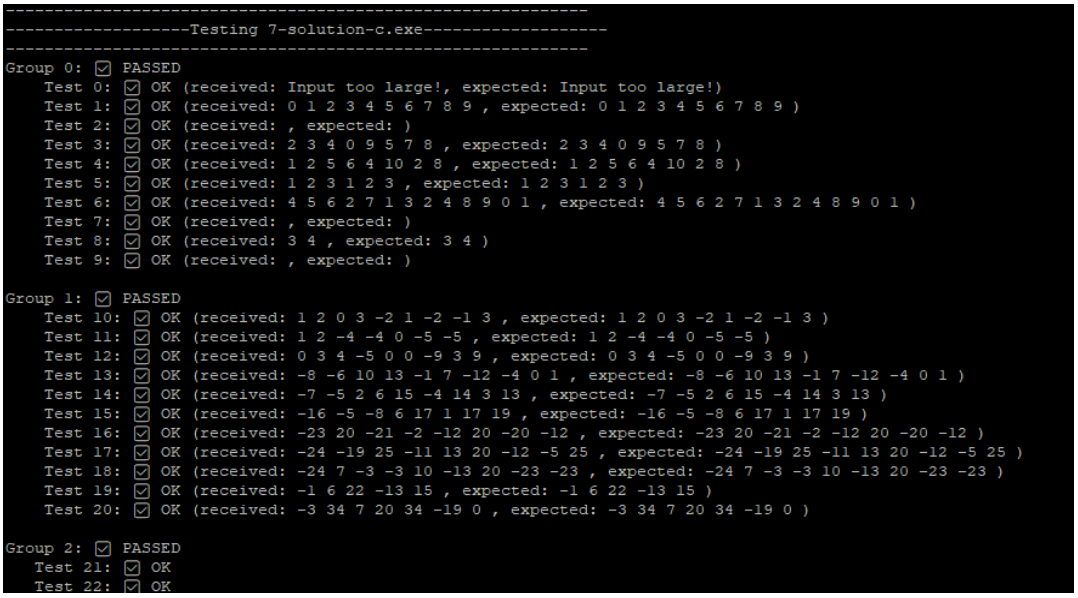
Здесь `флаг_ввода` и `флаг_вывода` принимают одно из трёх значений: 0, 1 или 2, где 0 означает использование стандартных потоков (`stdin`/`stdout`); 1 - использование файлового ввода/вывода (в таком случае названия файлов должны быть указаны на месте параметров `файл_ввода` или `файл_вывода` соответственно); 2 - значение, зарезервированное для дальнейших модификаций.

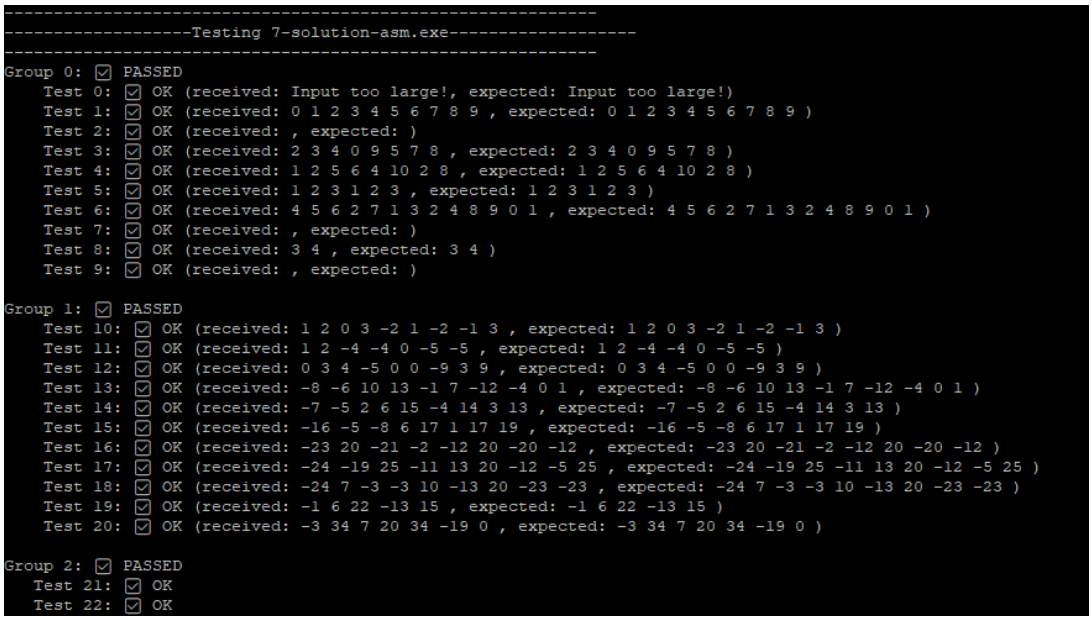
Программа на языке C, удовлетворяющая новым требованиям, представлена в файле `7-solution.c`. На языке ассемблера - в файлах `7-solution-*.s`. Во избежание внесения изменений в код подпрограмм, подключение файлов было реализовано функцией `freopen`, с помощью которой происходит “перенаправление” стандартных потоков ввода и вывода в указанные файлы.

Для проверки новых программ и упрощения измерения производительности в будущем, тестирующая программа была разделена на `test_stdin` и `test_file` в зависимости от используемого способа ввода/вывода.

Использование файлового ввода также позволило с помощью незначительной модификации программы увеличить ограничение на длину массива и вернуть группы тестов 6 и, в будущем, 7 для более точной оценки производительности программ. Для этого выделение памяти под массивы `A` и `B` было перемещено из фрейма функции `main` в секцию статических данных `data` (с использованием директивы `comm`).

Как показывает запуск тестирующей программы, работающей с файловым вводом и выводом, модифицированные программы работают корректно.

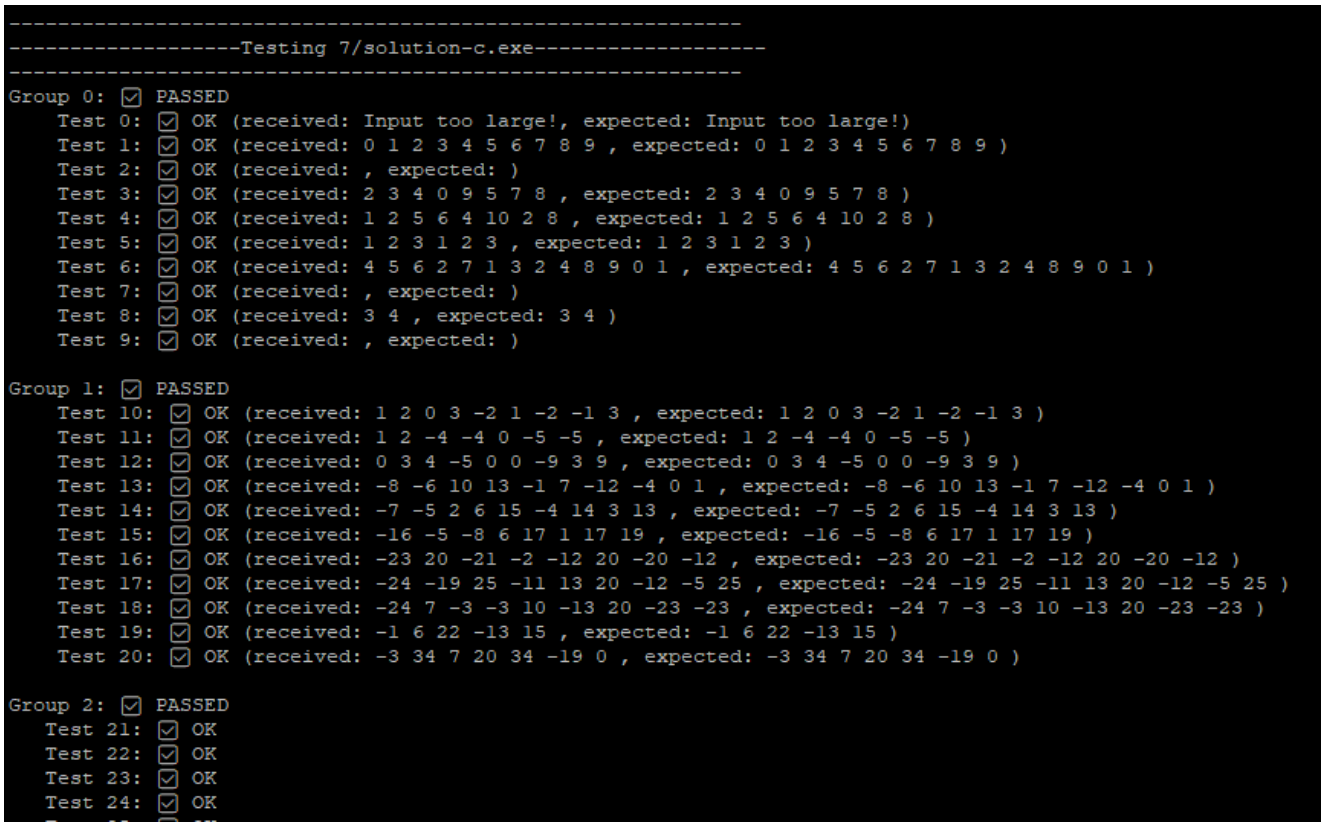


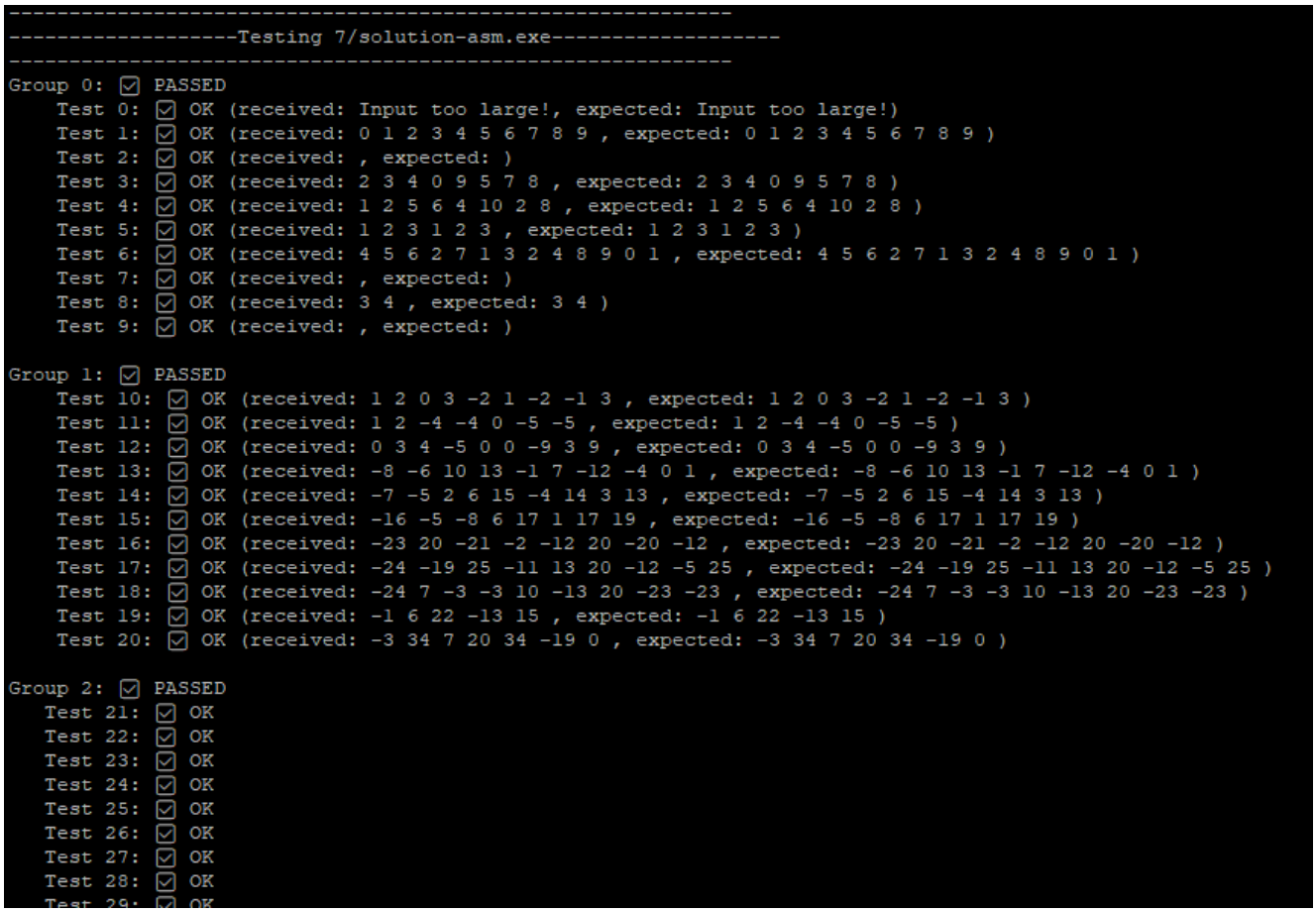


Для удобства дальнейшей работы и структуризации рабочего пространства дополнительно были внесены незначительные изменения в проект:

1. Все итерации программы были размещены в соответствующих папках.
2. Из названий файлов убран номер итерации.
3. В тестирующей программе `tets` названия исполняемых файлов заменены на относительные пути к ним от корневой папки.
4. Все `shell`-скрипты были адаптированы для работы с новой структурой проекта. Для корректной работы их запуск должен производиться из корневой папки командой `./iteration/filename.sh`, где `iteration` - название папки (являющееся номером итерации), `filename` - название необходимого `shell`-скрипта.

Проверка изменений, затрагивающих архивные итерации проекта, не проводилась. Как показало тестирование итерации №7, модификации не повлияли на корректность работы алгоритма.





Для дальнейшей модификации были взяты программы `solution.c` и `asm`, а также создан `shell`-скрипт `compile`, собирающий обе программы.

Генератор случайных наборов данных должен работать следующим образом: если аргумент командной строки `флаг_ввода` равен 2, то программа требует от пользователя ввод длины массива через стандартный поток, а заполнение массива производит самостоятельно на основе генератора случайных данных.

Для реализации этого в функцию `input` был добавлен дополнительный параметр `mode`, указывающий на способ получения элементов массива. Значение `0` указывает на ввод из возможно перенаправленного в файл `stdin`, `1` - на случайную генерацию.

Генерация случайных чисел происходит с помощью функции `rand` после установки семени генерации с помощью функции `srand`. Для простоты в качестве семени генерации используется текущее значение времени системы.

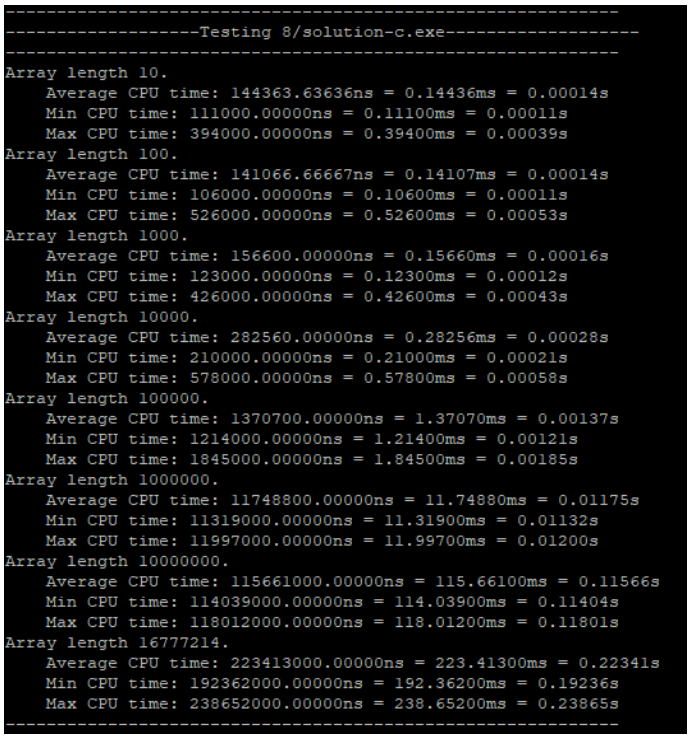
Соответствующие модификации были внесены как в `C`, так и в ассемблерную программу.

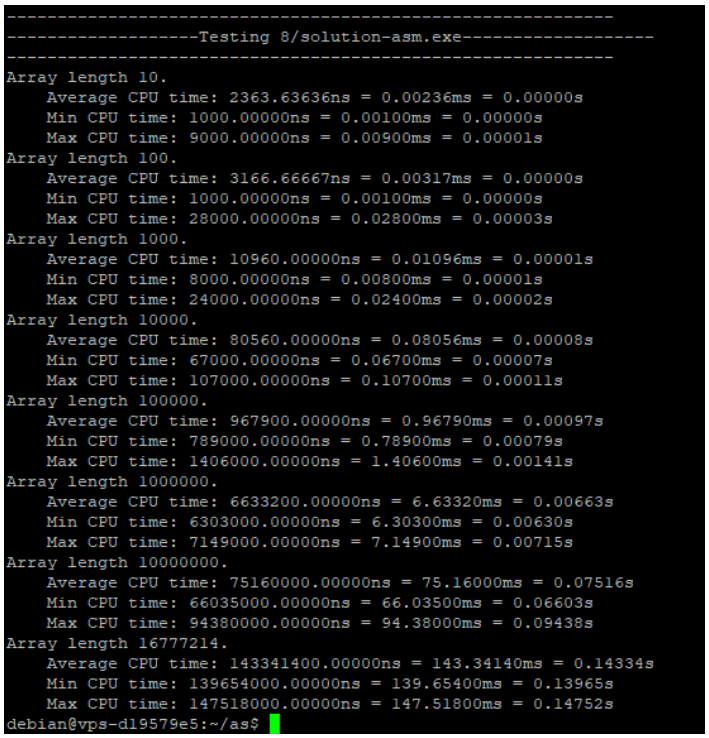
Для проведения сравнения на производительность обе программы были модифицированы для замера времени выполнения подпрограммы `solve` с помощью функции `clock`.

Для удобства измерений тестирующие программы были также модифицированы для вывода на экран среднего времени выполнения тестов в группе. Также была возвращена группа тестов 7 с помощью программы `gen`.

Для удобства измерения дополнительно была создана программа `benchmark`, которая не тестирует группу тестов 0 и не производит проверку правильности ответа, но дополнительно запускает программы на максимальном размере входных данных, используя генератор случайных чисел для получения исходного массива.

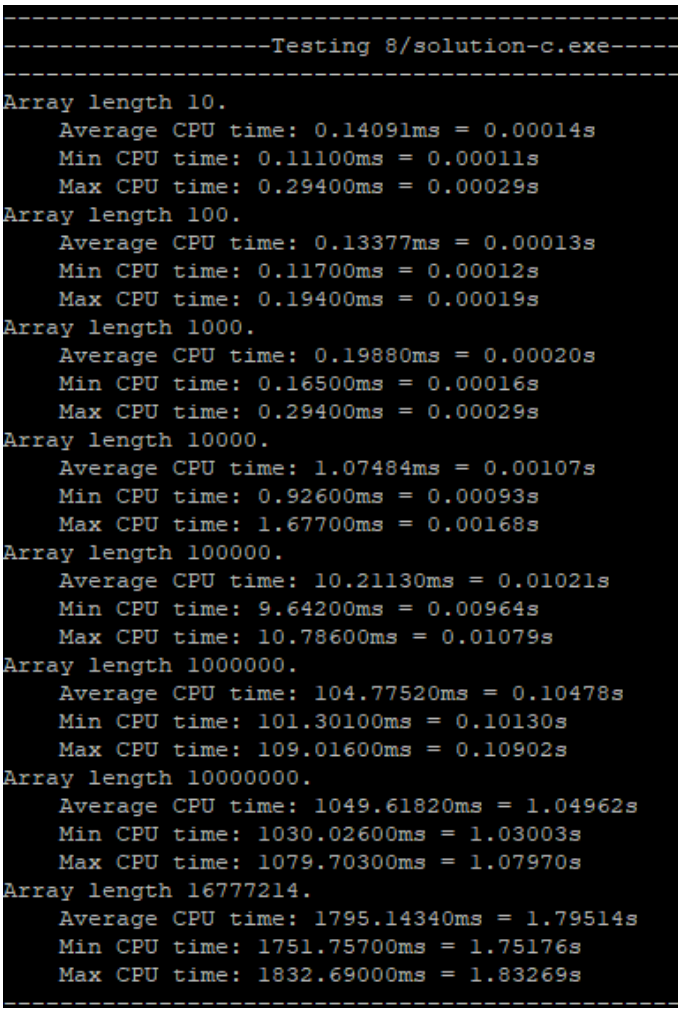
В результате запуска программы на устройстве с Intel Core Processor (Haswell, no TSX) @ 2.399GHz 2x vCPU, 4GB RAM, под управлением ОС Debian 10 получены следующие значения:





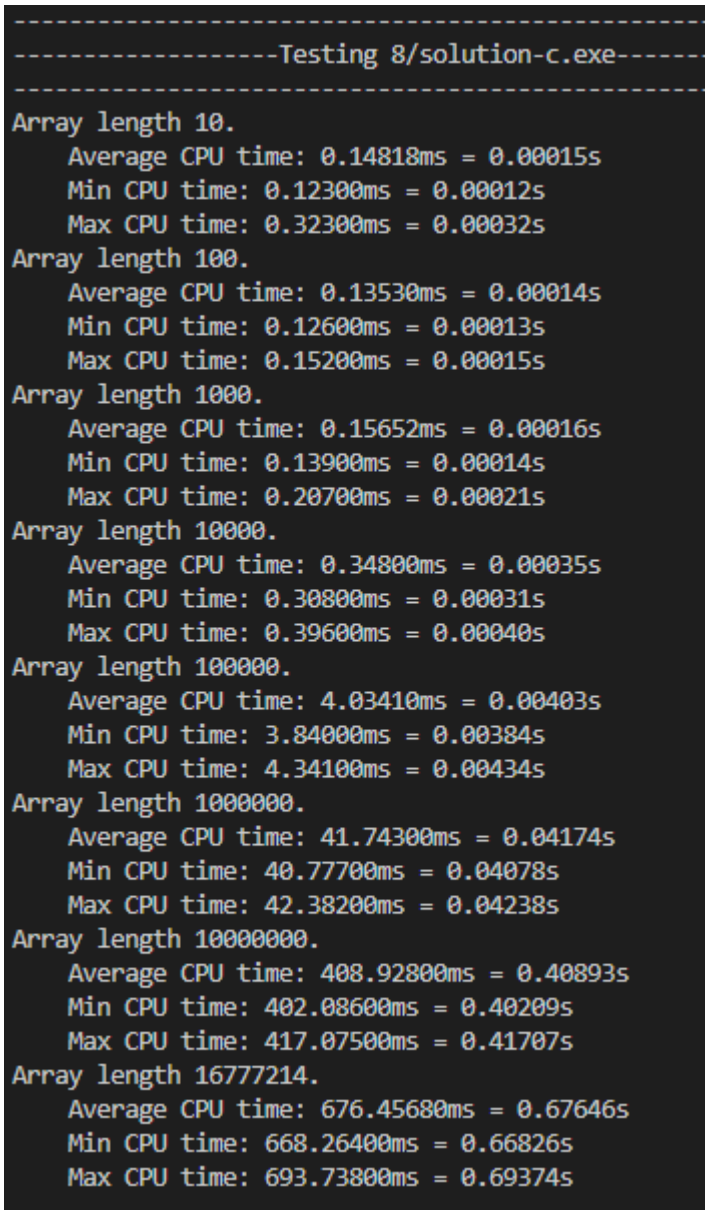
Нетрудно заметить, что С-программа, собранная компилятором `gcc`, работает до двух раз медленнее, чем вручную модифицированный для максимального использования регистров процессора ассемблерный листинг.

Для улучшения оценки незначительно модифицируем программы так, чтобы `solve` для каждого теста выполнялся 10 раз. Для удобства при повторном тестировании также был убран вывод времени выполнения в наносекундах.



Видно, что программа, полученная компиляцией кода на языке C, работает чуть менее, чем в 2 раза медленнее, чем оптимизированный вручную ассемблерный листинг.

Аналогичное тестирование на устройстве с 12th Gen Intel Core i5-12600K @ 3.6GHz 16 CPU, 32GB DDR5 RAM под управлением WSL Debian 9 на Windows 11 дало похожий результат:



```
------------------------------------------------
------------------Testing 8/solution-c.exe------
------------------------------------------------
Array length 10.
    Average CPU time: 0.14818ms = 0.00015s
    Min CPU time: 0.12300ms = 0.00012s
    Max CPU time: 0.32300ms = 0.00032s
Array length 100.
    Average CPU time: 0.13530ms = 0.00014s
    Min CPU time: 0.12600ms = 0.00013s
    Max CPU time: 0.15200ms = 0.00015s
Array length 1000.
    Average CPU time: 0.15652ms = 0.00016s
    Min CPU time: 0.13900ms = 0.00014s
    Max CPU time: 0.20700ms = 0.00021s
Array length 10000.
    Average CPU time: 0.34800ms = 0.00035s
    Min CPU time: 0.30800ms = 0.00031s
    Max CPU time: 0.39600ms = 0.00040s
Array length 100000.
    Average CPU time: 4.03410ms = 0.00403s
    Min CPU time: 3.84000ms = 0.00384s
    Max CPU time: 4.34100ms = 0.00434s
Array length 1000000.
    Average CPU time: 41.74300ms = 0.04174s
    Min CPU time: 40.77700ms = 0.04078s
    Max CPU time: 42.38200ms = 0.04238s
Array length 10000000.
    Average CPU time: 408.92800ms = 0.40893s
    Min CPU time: 402.08600ms = 0.40209s
    Max CPU time: 417.07500ms = 0.41707s
Array length 16777214.
    Average CPU time: 676.45680ms = 0.67646s
    Min CPU time: 668.26400ms = 0.66826s
    Max CPU time: 693.73800ms = 0.69374s
```

```
------------------------------------------------
------------------Testing 8/solution-asm.exe----
------------------------------------------------
Array length 10.
    Average CPU time: 0.00091ms = 0.00000s
    Min CPU time: 0.00000ms = 0.00000s
    Max CPU time: 0.00100ms = 0.00000s
Array length 100.
    Average CPU time: 0.00167ms = 0.00000s
    Min CPU time: 0.00100ms = 0.00000s
    Max CPU time: 0.00200ms = 0.00000s
Array length 1000.
    Average CPU time: 0.01168ms = 0.00001s
    Min CPU time: 0.01100ms = 0.00001s
    Max CPU time: 0.01400ms = 0.00001s
Array length 10000.
    Average CPU time: 0.14392ms = 0.00014s
    Min CPU time: 0.12800ms = 0.00013s
    Max CPU time: 0.17900ms = 0.00018s
Array length 100000.
    Average CPU time: 2.58090ms = 0.00258s
    Min CPU time: 2.51500ms = 0.00251s
    Max CPU time: 2.77500ms = 0.00278s
Array length 1000000.
    Average CPU time: 27.37140ms = 0.02737s
    Min CPU time: 26.59300ms = 0.02659s
    Max CPU time: 28.28300ms = 0.02828s
Array length 10000000.
    Average CPU time: 269.19960ms = 0.26920s
    Min CPU time: 263.14900ms = 0.26315s
    Max CPU time: 272.44200ms = 0.27244s
Array length 16777214.
    Average CPU time: 448.57820ms = 0.44858s
    Min CPU time: 440.55300ms = 0.44055s
    Max CPU time: 468.03200ms = 0.46803s
```

Для модификации и анализа в директорию 9 были скопированы последние версии программ на C и ассемблере.

Для повышения качества программы была произведена модификация разбора параметров командной строки во избежание ошибок во время выполнения, а также добавлена проверка на наличие файла со входными данными.

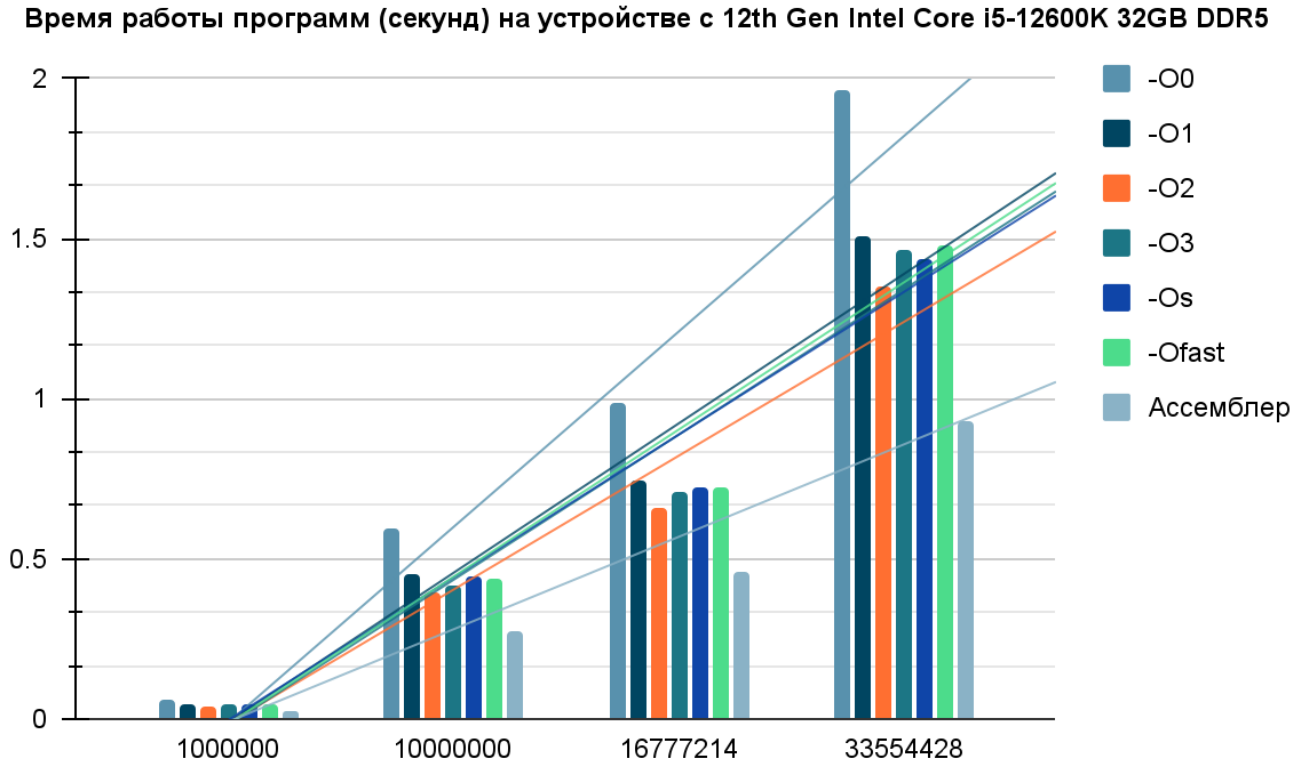
Для удобства дальнейшего тестирования были также произведены следующие изменения:

1. максимальная длина вводимого массива увеличена до `33554428`
2. количество прогонов подпрограммы `solve` увеличено до `15`

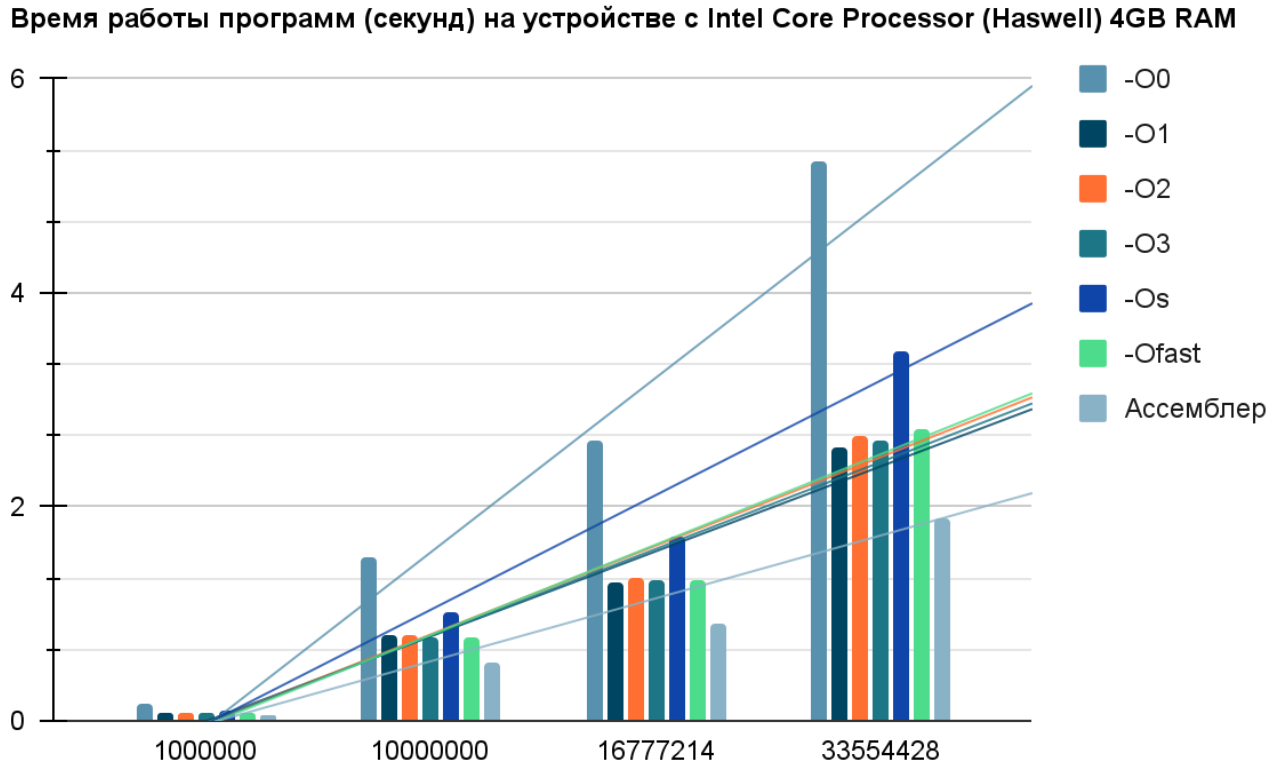
3. в программу `benchmark` добавлена группа тестов на новых максимальных входных данных
4. из программы `benchmark` убран вывод минимального и максимального зафиксированного времени выполнения.

Для проведения замеров был также модифицирован скрипт `9/compile.sh`, который теперь собирает несколько версий C-программы с разным опциями оптимизации по скорости и размеру. В результате тестирования были получены следующие результаты:

12th Gen Intel Core i5-12600K @ 3.6GHz 16 CPU, 32GB DDR5 RAM под управлением WSL Debian 9 на Windows 11						
Опции оптимизации	Время выполнения (s)				Размер ассемблерного листинга (строк)	Размер исполняемого файла (байт)
	1000000	10000000	16777214	33554428		
-O0	0.06012	0.59433	0.98813	1.96693	359	13,312
-O1	0.04702	0.45061	0.74765	1.50644	280	13,312
-O2	0.04033	0.39589	0.66170	1.34774	299	13,312
-O3	0.04705	0.41968	0.70604	1.46575	326	13,312
-Os	0.04538	0.44193	0.72667	1.43901	242	13,312
-Ofast	0.04378	0.43656	0.72626	1.48211	326	14,768
Ассемблер	0.02678	0.27611	0.46148	0.92906	312	14,136



Intel Core Processor (Haswell, no TSX) @ 2.399GHz 2x vCPU, 4GB RAM, под управлением ОС Debian 10						
Опции оптимизации	Время выполнения (s)				Размер ассемблерного листинга (строк)	Размер исполняемого файла (байт)
	1000000	10000000	16777214	33554428		
-O0	0.15311	1.52903	2.62062	5.23094	350	17,184
-O1	0.07653	0.80012	1.29262	2.56369	274	17,184
-O2	0.07680	0.80053	1.32868	2.66742	302	17,184
-O3	0.07973	0.77282	1.31987	2.61091	326	17,184
-Os	0.10012	1.01877	1.71199	3.44500	237	13,312
-Ofast	0.07823	0.77441	1.31090	2.71931	326	18,696
Ассемблер	0.05252	0.54209	0.90377	1.89365	312	18,008



Ожидаемо, ассемблерная программа, написанная вручную, работает наиболее быстро, а С-программа, собранная без опций оптимизации - наиболее медленно. Программы, собранные с опциями `O1`, `O2`, `O3` и `Ofast` работают примерно одинаково: размера программы оказалось недостаточно для того, чтобы различия в алгоритмах оптимизации были заметны. Размер ассемблерного листинга наиболее маленький при опции `Os`, которая в первую очередь применяет оптимизации по размеру файла, что заметно сказывается на времени выполнения. Тем не менее размер программы не достаточно велик, чтобы оценить разницу в размере исполняемого файла:

значения получаются примерно одинаковые в связи с выравниванием, которое операционные системы применяют к исполняемым файлам.

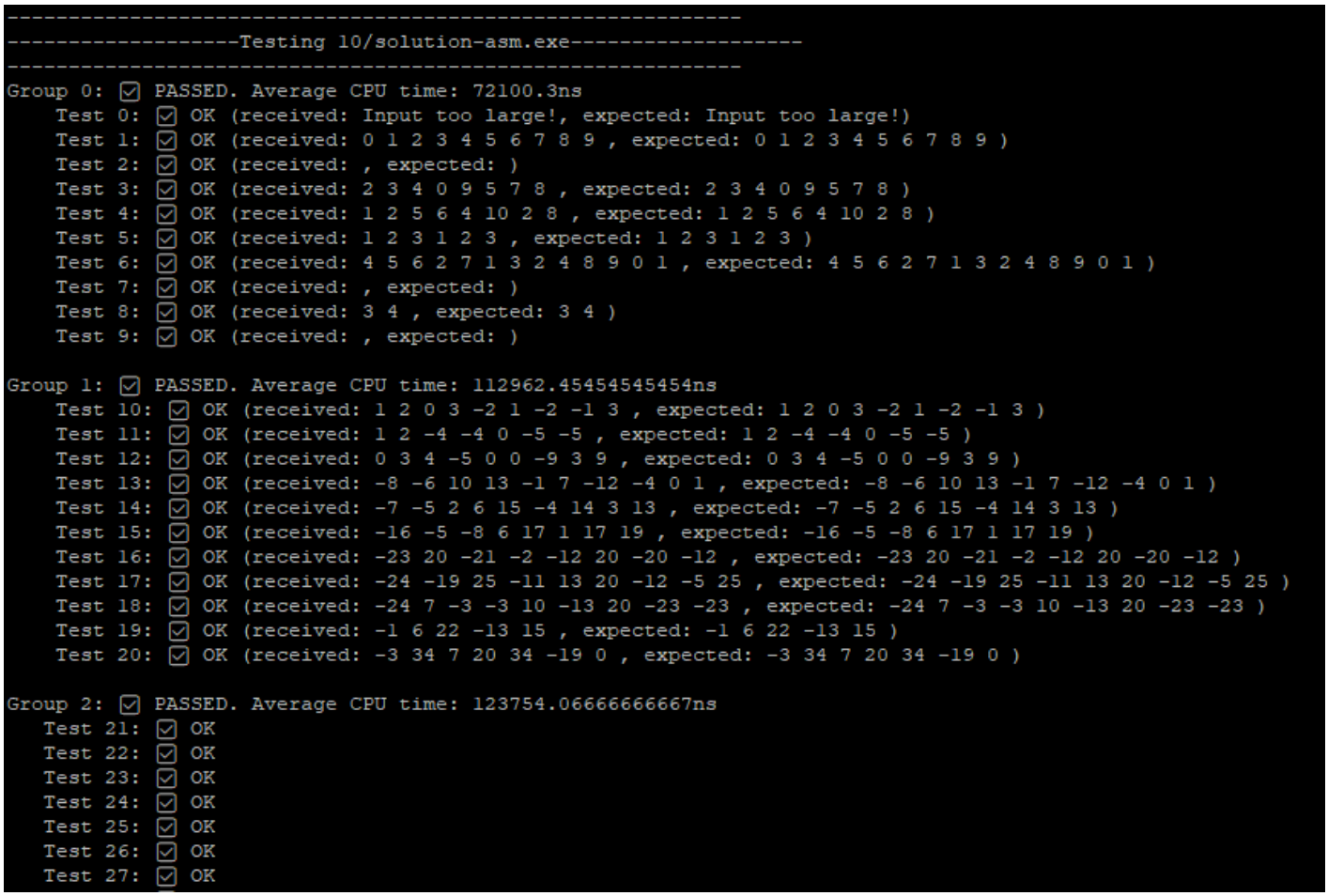
Также нетрудно заметить, что все программы работают примерно вдвое медленнее на процессоре от Intel 4-го поколения в сравнении с процессором 12-го поколения, что соответствует ожиданиям.

Для рефакторинга ассемблерной программы без использования `libc` последняя итерация была скопирована в папку `10`, где все функции стандартной библиотеки были заменены на системные вызовы:

1. Переменные `stdin` и `stdout`, указывающие на стандартные потоки ввода и вывода были реализованы в виде переменных `.instream` и `.outstream`.
2. Функции `scanf` и `printf` были реализованы с помощью системных выводов `sys_read` и `sys_write` соответственно, а также алгоритма преобразования строкового представления числа в десятичной системе счисления в численное значение регистра процессора.
3. `freopen` был заменён на пару системных вызовов `sys_open` и `sys_close`.
4. Функция `clock` была реализована с помощью системного вызова `sys_clock_gettime`
5. Вызов `rand` был заменён на системный вызов `sys_getrandom`, при этом предварительная установка семени генерации с помощью `time` и `srand` не требуется.
6. Функция `main` как точка входа программы была изменена на стандартный для ОС Linux `_start`. При этом незначительно изменён алгоритм разбора параметров командной строки, которые новая программа получает не в регистрах `rsi` и `rdi`, а на стеке.

Также используемый при сборке программы линкер был изменён с `gcc` на `ld` для уверенности в том, что зависимость программы от `libc` была полностью устранена.

Как показывает тестирование с помощью программы `test_file`, алгоритм работает верно:



Тем не менее надо отметить, что, хотя время работы подпрограммы `solve` не изменилось (те же `26488906`ns ~ `0.026s` на устройстве с 12th Gen Intel Core i5-12600K @ 3.6GHz 16 CPU, 32GB DDR5 RAM под управлением WSL Debian 9 на Windows 11 и `51600825ns ~ 0.052s` на устройстве с Intel Core Processor (Haswell, no TSX) @ 2.399GHz 2x vCPU, 4GB RAM под управлением ОС Debian 10 при входных данных размером `1000000`), суммарное время работы программы с учётом ввода/вывода значительно увеличилось, причём изменение значительно более заметно на устройстве под управлением WSL Debian 9 на Windows 11 (видимо, из-за особенностей организации системных вызовов виртуальной машиной), на котором даже не представляется возможным провести полный замер времени выполнения программ: ввод и вывод данных уже на размере `100000` достигает минуты для каждого теста.

Более точные замеры были получены с помощью программы `benchmark_io`, которая, помимо выведенного программой времени выполнения, анализирует и полное время работы с учётом ввода/вывода.

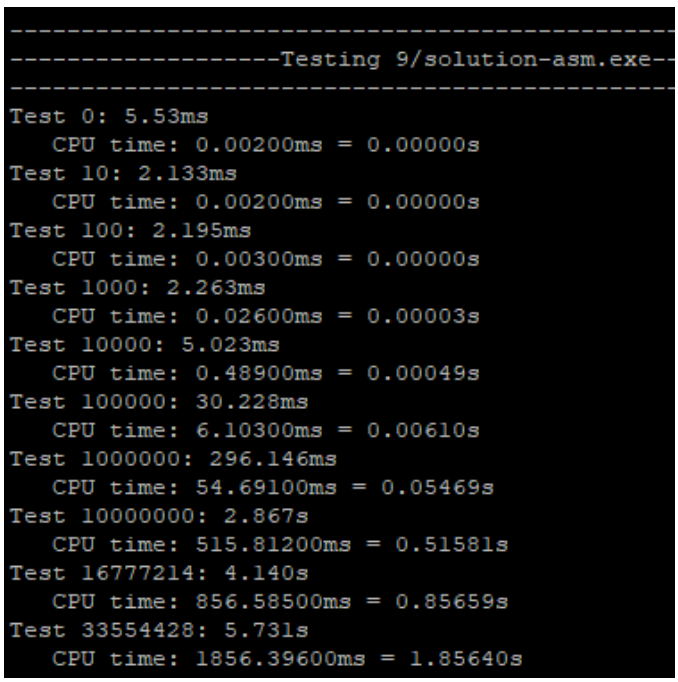
12th Gen Intel Core i5-12600K @ 3.6GHz 16 CPU, 32GB DDR5 RAM под управлением WSL Debian на Windows 11		
	Итерация 9	Итерация 10
Ввод из файла 101.in	30.771ms	58.234s
Генератор случайных чисел	224.155ms	74.216s

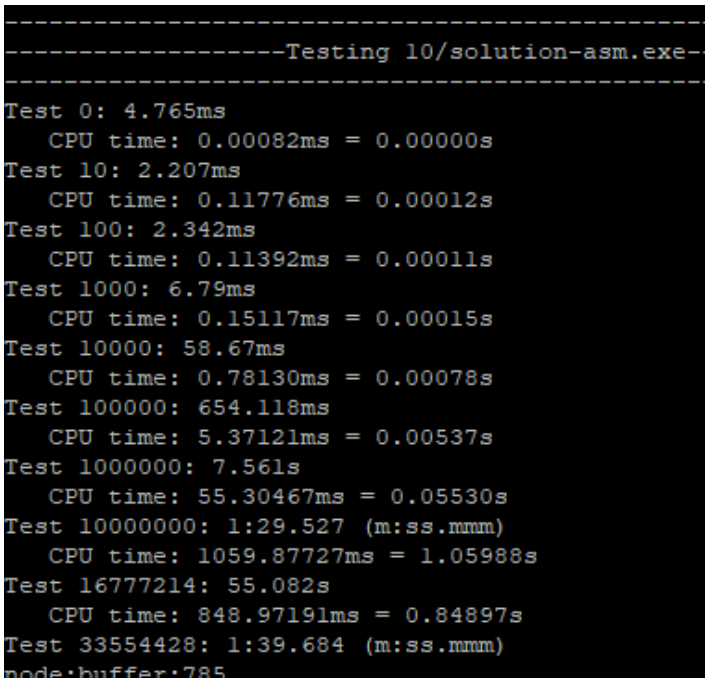
```
Iteration 9, input from 101.in: 30.771ms
CPU time: 2.73900ms = 0.00274s
Iteration 10, input from 101.in: 58.234s
CPU time: 2.88868ms = 0.00289s
Iteration 9, random input: 224.155ms
CPU time: 28.37400ms = 0.02837s
Iteration 10, random input: 1:14.216 (m:ss.mmm)
CPU time: 29.04104ms = 0.02904s
ttpo100ajiex@TTPO100AJIEX:/mnt/f/programming/IDZ1$
```

Intel Core Processor (Haswell, no TSX) @ 2.399GHz 2x vCPU, 4GB RAM, под управлением ОС Debian 10		
	Итерация 9	Итерация 10
Ввод из файла 101.in	40.314ms	654.557ms
Генератор случайных чисел	169.751ms	2.807s

```
Iteration 9, input from 101.in: 40.314ms
CPU time: 5.54300ms = 0.00554s
Iteration 10, input from 101.in: 654.557ms
CPU time: 5.18177ms = 0.00518s
Iteration 9, random input: 169.751ms
CPU time: 51.65200ms = 0.05165s
Iteration 10, random input: 2.807s
CPU time: 50.85694ms = 0.05086s
```

Для оценки разницы во времени работы программ на устройстве под управление Debian 10 была создана программа `benchmark_io_2`, которая выводит время выполнения программы с учётом ввода/вывода на одном тесте каждой группы. Многоразовое тестирование не проводилось, так как важны не точные значения, а разница в сравнении с программой итерации 9. Как нетрудно заметить, наивная реализация `libc` проигрывает в десятки и даже сотни раз (стоит обратить внимание, что случайная генерация массива длины `16777214` работает даже быстрее, чем чтение из файла массива длины `10000000`; к сожалению. выходной файл итерации 10 превысил 512МБ, из-за чего тестировщик не смог корректно его обработать и вывести время, затраченное на собственно алгоритм решение задачи).





Следовательно, модификация программы привела к её ухудшению в связи с тем, что библиотека `libc` использует значительно меньше системных вызовов, чем полученная мной программа. Вместо того, чтобы обращаться к системе много раз, функции `scanf`, `printf` и `rand` библиотеки сохраняют очередь запросов во внутреннем буфере, и делают системный вызов лишь тогда, когда в буфере накапливается достаточное количество данных в случае вывода или когда он опустошается в случае ввода. Это позволяет значительно улучшить производительность I/O операций.