Национальный исследовательский университет “Высшая школа экономики”.

Факультет компьютерных наук. Программная инженерия.

Архитектура вычислительных систем.

Индивидуальное домашнее задание №3 студента группы БПИ213 Абрамова Александра Сергеевича.

Вариант 5.

5. Разработать программу, вычисляющую с помощью степенного ряда с точностью не хуже 0,05% значение функции $\arcsin(x)$ для заданного параметра x.

СОДЕРЖАНИЕ

Требования к работе…….... 2

Организация решения и его теоретическое обоснование……... 4

Программа на языке С…….. 7

Тестирование и анализ программ…….. 10

Сборка и запуск программ…….. 12

Программа на языке ассемблера……14

Тестирование программ…….. 20

Сравнение программ по скорости выполнения…….. 22

Сравнение программ по размеру……..…. 27

Требования к работе

В результате анализа заданных требований было установлено, 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.
   1.3. С помощью аргументов командной строки должен быть доступен выбор способа ввода данных:
      1.3.1. Стандартный ввод из потока $stdin$;
      1.3.2. Ввод из указанного файла;
      1.3.3. Случайная генерация входных данных.
   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. С помощью аргументов командной строки должен быть доступен выбор способа вывода данных:
      1.5.1. Стандартный вывод в поток $stdout$;
      1.5.2. Вывод в указанный файл;
      1.5.3. Отсутствие вывода результата. Для вывода ошибок и других технических данных должен быть использован стандартный поток вывода $stdout$.
   1.6. Программа должна позволять проводить анализ времени выполнения алгоритма без учёта ввода и вывода данных. Для этого должно быть реализовано следующее:
      1.6.1. Вывод вычисленного времени выполнения указанным в командной строке способом (см. п.1.5) на отдельной строке;
      1.6.2. Включение с помощью аргументов командной строки многократного выполнения алгоритма для получения более точных результатов измерения времени выполнения.
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 C 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.

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.

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.

Организация решения и его теоретическое обоснование

Для определённости реализации установим следующее:

1. Все данные должны храниться в регистрах процессора, на стеке или в секции статических данных.
2. Разработанный алгоритм должен корректно работать только при вводе значений из интервала $[-1; 1]$ - область определения функции $arcsin(x)$ на множестве вещественных чисел. При вводе некорректных значений программа должна печатать ошибку и немедленно завершаться.
3. Формат аргументов командной строки должен быть следующий:

```
exe_name loop in_flag in_file out_flag out_file
```

Здесь под указанными именами понимаются следующие значения:

a. `exe_name` - имя запускаемого исполняемого файла;

b. `loop` - значение, указывающее на необходимость зацикливания алгоритма:

   a. 0 - зацикливание выполнять не требуется;

   b. 1 - требуется выполнить зацикливание;

c. `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;

d. `in_file` - имя файла со входными данными. Этот параметр должен быть указан только если `in_flag = 1`;

e. `out_flag` - значение, указывающее на способ вывода:

   a. 0 - использование стандартного потока вывода $stdout$;

   b. 1 - использование файлового вывода;

   c. 2 - отсутствие вывода результата;

f. `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.

4. Из курса математического анализа известно следующее:

$$arcsin(x) = \sum_{n=0}^{\infty} \frac{(2n)!}{4^n \cdot (n!)^2 \cdot (2n+1)} \cdot x^{2n+1} =$$

$$= \frac{(2\cdot 0)!}{4^0 \cdot (0!)^2 \cdot (2\cdot 0+1)} \cdot x^{2\cdot 0+1} + \frac{(2\cdot 1)!}{4^1 \cdot (1!)^2 \cdot (2\cdot 1+1)} \cdot x^{2\cdot 1+1} + \frac{(2\cdot 2)!}{4^2 \cdot (2!)^2 \cdot (2\cdot 2+1)} \cdot x^{2\cdot 2+1} + \ldots =$$

$$= \frac{1}{1 \cdot 1^2 \cdot 1} \cdot x^1 + \frac{1 \cdot 2}{4 \cdot 1^2 \cdot 3} \cdot x^3 + \frac{4!}{16 \cdot 2^2 \cdot 5} \cdot x^5 + \ldots = x + \frac{1}{6}x^3 + \frac{3}{40}x^5 + \ldots$$

$arcsin(1) = \frac{\pi}{2} \approx 1.5707963$; $arcsin(-1) = -\frac{\pi}{2} \approx -1.5707963$. Далее будем считать, что $x \in (-1; 1)$, то есть $|x| < 1$.

Пусть ответ был вычислен путём сложения k первых членов представленного степенного ряда, то есть по формуле $\sum_{n=0}^{k-1} \frac{(2n)!}{4^n \cdot (n!)^2 \cdot (2n+1)} \cdot x^{2n+1}$. Тогда погрешность вычисления равна остаточному члену ряда - сумме оставшихся элементов: $\sum_{n=k}^{\infty} \frac{(2n)!}{4^n \cdot (n!)^2 \cdot (2n+1)} \cdot x^{2n+1}$. Рассмотрим его абсолютную величину, то есть абсолютную величину погрешности вычисления: $\left| \sum_{n=k}^{\infty} \frac{(2n)!}{4^n \cdot (n!)^2 \cdot (2n+1)} \cdot x^{2n+1} \right| =$

$$= \sum_{n=k}^{\infty} \left| \frac{(2n)!}{4^n \cdot (n!)^2 \cdot (2n+1)} \right| \cdot \left| x^{2n+1} \right| = \sum_{n=k}^{\infty} \frac{(2n)!}{4^n \cdot (n!)^2 \cdot (2n+1)} \cdot |x|^{2n+1}$$

Заметим, что коэффициент при $|x|^{2n+1}$ уменьшается с увеличением n:

$$\frac{(2n)!}{4^n \cdot (n!)^2 \cdot (2n+1)} \vee \frac{(2(n+1))!}{4^{n+1} \cdot ((n+1)!)^2 \cdot (2(n+1)+1)}$$

$$\frac{\frac{(2n)!}{4^n \cdot (n!)^2 \cdot (2n+1)}}{\frac{(2(n+1))!}{4^{n+1} \cdot ((n+1)!)^2 \cdot (2(n+1)+1)}} \vee 1$$

$$\frac{(2n)! \cdot 4^{n+1} \cdot ((n+1)!)^2 \cdot (2(n+1)+1)}{(2(n+1))! \cdot 4^n \cdot (n!)^2 \cdot (2n+1)} \vee 1$$

$$\frac{(2n)! \cdot 4^{n+1} \cdot (n+1)! \cdot (n+1)! \cdot (2n+3)}{(2n+2)! \cdot 4^n \cdot n! \cdot n! \cdot (2n+1)} \vee 1$$

$$\frac{(2n)!}{(2n+2)!} \cdot \frac{4^{n+1}}{4^n} \cdot \frac{(n+1)!}{n!} \cdot \frac{(n+1)!}{n!} \cdot \frac{(2n+3)}{(2n+1)} \vee 1$$

$$\frac{1}{(2n+1)(2n+2)} \cdot 4 \cdot (n+1) \cdot (n+1) \cdot \frac{(2n+3)}{(2n+1)} \vee 1$$

$$\frac{4(n+1)^2(2n+3)}{(2n+1)^2(2n+2)} \vee 1$$

$$4(n+1)^2(2n+3) \vee (2n+1)^2(2n+2)$$

$$4(n+1)^2(2n+3) \vee (2n+1)^2(2n+2)$$

$$4(n+1)^2(2n+3) - (2n+1)^2(2n+2) \vee 0$$

$$(8n^3 + 28n^2 + 32n + 12) - (8n^3 + 16n^2 + 10n + 2) \vee 0$$

$$12n^2 + 22n + 10 \vee 0$$

$$\forall n > 0\colon\ 12n^2 + 22n + 10 > 0 \Rightarrow \frac{(2n)!}{4^n \cdot (n!)^2 \cdot (2n+1)} > \frac{(2(n+1))!}{4^{n+1} \cdot ((n+1)!)^2 \cdot (2(n+1)+1)}$$

Значит, $\sum_{n=k}^{\infty} \frac{(2n)!}{4^n \cdot (n!)^2 \cdot (2n+1)} \cdot |x|^{2n+1} \le \sum_{n=k}^{\infty} \frac{(2k)!}{4^k \cdot (k!)^2 \cdot (2k+1)} \cdot |x|^{2n+1} = \frac{(2k)!}{4^k \cdot (k!)^2 \cdot (2k+1)} \sum_{n=k}^{\infty} |x|^{2n+1}$

$\sum_{n=k}^{\infty} x^{2n+1}$ - сумма бесконечно убывающей геометрической прогрессии при $|x| < 1$. Значит, $\sum_{n=k}^{\infty} |x|^{2n+1} = \frac{|x|^{2k+1}}{1-x^2}$. Таким образом, $\sum_{n=k}^{\infty} \frac{(2n)!}{4^n \cdot (n!)^2 \cdot (2n+1)} \cdot |x|^{2n+1} \le \frac{(2k)!}{4^k \cdot (k!)^2 \cdot (2k+1)} \cdot \frac{|x|^{2k+1}}{1-x^2}$.

То есть сумма первых k членов степенного ряда отличается от точного результата не более, чем на $\frac{(2k)!}{4^k \cdot (k!)^2 \cdot (2k+1)} \cdot \frac{|x|^{2k+1}}{1-x^2}$. Значит, для обеспечения точности не хуже 0.0005 требуется, чтобы $\frac{(2k)!}{4^k \cdot (k!)^2 \cdot (2k+1)} \cdot \frac{|x|^{2k+1}}{1-x^2} \le 0.0005$

Итак, для алгоритмически-эффективного получения ответа программа должна вычислить сумму элементов вида $g(n) = \frac{(2n)!}{4^n \cdot (n!)^2 \cdot (2n+1)} \cdot x^{2n+1}$ для $n \in [0;\ k-1]$, где k минимально и удовлетворяет неравенству $h(k) = \frac{(2k)!}{4^k \cdot (k!)^2 \cdot (2k+1)} \cdot \frac{|x|^{2k+1}}{1-x^2} \le 0.0005$

Заметим, что $g(n) = \frac{(2n)!}{4^n \cdot (n!)^2 \cdot (2n+1)} \cdot x^{2n+1} = \frac{(2n-2)! \cdot (2n-1) \cdot 2n}{4^{n-1} \cdot 4 \cdot ((n-1)! \cdot n)^2 \cdot (2n+1) \cdot \frac{2n-1}{2n-1}} \cdot x^{2n-1} \cdot x^2 =$

$= \frac{(2(n-1))!}{4^{n-1} \cdot ((n-1)!)^2 \cdot (2n-1)} \cdot \frac{(2n-1) \cdot 2n}{4 \cdot n^2 \cdot \frac{2n+1}{2n-1}} \cdot x^{2n-1} \cdot x^2 = \frac{(2(n-1))!}{4^{n-1} \cdot ((n-1)!)^2 \cdot (2n-1)} \cdot x^{2n-1} \cdot \frac{(2n-1) \cdot 2n}{4 \cdot n^2 \cdot \frac{2n+1}{2n-1}} \cdot x^2 =$

$= g(n-1) \cdot \frac{(2n-1)^2}{2n \cdot (2n+1)} \cdot x^2$. При этом $g(0) = \frac{(2 \cdot 0)!}{4^0 \cdot (0!)^2 \cdot (2 \cdot 0+1)} \cdot x^{2 \cdot 0+1} = x$

Таким образом, для вычисления члена под номером i можно использовать результат вычисления члена под номером $i-1$, что позволяет оптимизировать вычисления.

Также обратим внимание, что $h(k) = \frac{(2k)!}{4^k \cdot (k!)^2 \cdot (2k+1)} \cdot \frac{|x|^{2k+1}}{1-x^2} = \left| \frac{(2k)!}{4^k \cdot (k!)^2 \cdot (2k+1)} \cdot \frac{x^{2k+1}}{1-x^2} \right| =$

$= \left| \frac{(2k)!}{4^k \cdot (k!)^2 \cdot (2k+1)} \cdot x^{2k+1} \cdot \frac{1}{1-x^2} \right| = \left| g(k) \cdot \frac{1}{1-x^2} \right|$.

Значит, для получения ответа требуется использовать следующий алгоритм:

1. Если $x = 1$, $arcsin(x) = \frac{\pi}{2} \approx 1.5707963$
2. Если $x = -1$, $arcsin(x) = -\frac{\pi}{2} \approx -1.5707963$
3. Иначе, $arcsin(x) \approx \sum_{n=0}^{k-1} g(n)$, где $\left| g(k) \cdot \frac{1}{1-x^2} \right| < 0.0005$

Программа на языке C

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 данных:
   1.1. В зависимости от параметра командной строка `loop` устанавливается значение переменной $loop$ логического типа, показывающее, требуется ли производить зацикливание программы.
   1.2. 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.
   1.3. Если `in_flag = 2`, переменной $random\_input$ устанавливается значение $true$, означающее, что требуется случайная генерация входных данных. В ином случае значение $random\_input$ равно $false$.
   1.4. 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. Если `out_flag = 2`, дополнительно устанавливается значение $false$ переменной do_write, указывающее, что вывод ответа не должен быть произведён.
   1.5. 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. Для ввода значения x реализована подпрограмма $input$, принимающая единственным параметром значение логического типа, указывающее на способ получения данных: чтение из файла/стандартного потока или случайная генерация.
   2.1. Если требуется случайная генерация данных, то подпрограмма устанавливает семя генерации случайных данных как текущее время системы с помощью вызовов $srand(time(NULL))$ (криптографически-сильная случайная генерация данных не требуется, поэтому этот алгоритм удовлетворяет заданным требованиям), после чего получает случайное целое число в интервале $[0; RAND\_MAX]$ и приводит его к интервалу $[-1; 1]$ арифметическими операциями деления, умножения и вычитания.

2.2. Если требуется чтение данных из возможно перенаправленного в файл потока $stdin$, то подпрограмма использует функцию $scanf$ стандартной библиотеки языка C. Функция $input$ проверку корректности входных данных не производит.

3. После ввода производится проверка корректности полученного значения: в соответствии с установленными требованиями, абсолютная величина x не должна превышать 1. Если введённое значение некорректно, программа печатает сообщение об ошибке и завершается.
4. 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 разность полученных значений на количество тактов процессора, выполняемых за секунду ($CLOCKS\_PER\_SEC$). Таким образом, в переменной $answer$ оказывается ответ на задачу, а в переменной cpu_time_used - время, затраченное на его вычисление. Более того, для удобства анализа вычисленного времени выполнения, в зависимости от значения переменной $loop$, вызов $solve$ может производиться многократно. Экспериментально получено, что 10^8 повторений позволяют увеличить суммарное время выполнения приблизительно до одной секунды, что и требуется для удобного анализа.
   4.1. Подпрограмма $solve$ получает на вход в качестве аргумента функции значение переменной x, полученное подпрограммой $input$.
   4.2. Для проверки случаев $x = 1$ и $x = -1$ вычисляется значение $(1 - x^2)$. Нетрудно заметить, что $(1 - x^2) = 0$ тогда и только тогда, когда $x = 1$ или $x = -1$. Во избежание ошибок, связанных с невозможностью поддержания точности при операциях над числами с плавающей точкой, значение $(1 - x^2)$ сравнивается с 10^{-7}. Это позволяет избежать ошибок вычислений при сохранении погрешности итогового результата не более, чем 0.0005.
   4.3. Если оказывается, что абсолютная величина x действительно близка к единице, происходит возврат вычисленного вручную значения выражения $\frac{\pi}{2}$ или $-\frac{\pi}{2}$ в зависимости от знака x с точностью в 7 знаков после десятичной точки, что обеспечивает требуемую точность.
   4.4. Если $|x| < 1$, то происходит вычисление ответа путём суммирования нескольких первых членов степенного ряда функции $arcsin(x)$. Для этого организуется цикл, выполняющий следующие действия:

4.4.1. Проверка необходимости продолжения вычислений по доказанной выше формуле: если значение уже получено с требуемой точностью, выполнение завершается и происходит возврат результата.

4.4.2. Происходит изменение значения переменной $answer$ - предполагаемого ответа на задачу - путём прибавления очередного члена степенного ряда.

4.4.3. С помощью переменной k - номера итерации цикла - организуется вычисление значения следующего члена степенного ряда g по полученной ранее формуле.

5. Если вывод результата требуется, происходит печать ответа с помощью подпрограммы $output$, которая в качестве единственного параметра функции принимает выводимое число с плавающей точкой. Для организации вывода $output$ использует функцию $printf$ стандартной библиотеки языка C. Причём вывод результата производится с точностью 7 знаков после десятичной точки, что обеспечивает погрешность, не превышающую 0.0005.
6. По завершении работы подпрограммы $output$ с помощью функции $printf$ стандартной библиотеки языка C выводится вычисленное значение времени выполнения в наносекундах.

Тестирование и анализ программ

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 такая папка содержит наборы входных (файлы с расширением `in`) и эталонных выходных (файлы с расширением `out`) данных. Названия файлов соответствуют номеру теста. Группы тестов организованы следующим образом:

Номер группы тестов	Номера тестов	Комментарий
1	$0,\ 10$	Проверка поведения программы при вводе некорректных данных.
	$1 - 9$	Вручную созданные наборы входных и выходных данных.
2	$11 - 120$	Наборы данных, сгенерированные программой `gen.js` (см п.3)

3. Программа `gen.js` позволяет произвести генерацию тестов для всех групп, заданных в файле конфигурации, кроме первой группы, которая создана вручную. Для этого программа для каждого теста генерирует случайное число x в интервале $[-1;\ 1]$, вычисляет величину $arcsin(x)$ и записывает данные 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 верными 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 для одного теста группы был получен неверный ответ, вся группа считается не пройденной.

4.1. Для проверки правильности ответа используется следующий алгоритм:

4.1.1. Если эталонные выходные данные - строка, то производится посимвольное сравнение вывода программы с верным ответом. Результат верен тогда и только тогда, когда строки совпадают. Этот случай соответствует тестам 0 и 10 на поведение программы при вводе некорректного значения.

4.1.2. Иначе, эталонные выходные данные - число с плавающей точкой. При этом выходные данные программы также интерпретируются как число с плавающей точкой. Если этого сделать не удалось, считается, что найденный программой ответ неверен. В ином случае вычисляется абсолютная величина разницы между эталонным ответом и результатом работы программы - погрешность. При этом ответ верный тогда и только тогда, когда погрешность не превышает 0.0005.

5. Также была реализована программа `benchmark.js`, которая тестирует только вторую группу тестов и не производит проверку правильности ответа, но дополнительно по 50 раз осуществляет запуск программ со случайными входными данными, используя встроенный генератор случайных чисел (`in_flag = 2`) для получения вводимого числа x. При этом программа анализирует напечатанное время работы алгоритма, а также время работы всей программы с учётом затрат на её запуск и организацию ввода и вывода данных, и печатает на стандартный поток вывода $stdout$ минимальное, среднее и максимальное время работы исполняемого файла на тестах каждой группы.

Сборка и запуск программ

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 языке 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 с синтаксисом $intel$.
   3.3. $Wall$ для отлавливания возможных ошибок, допущенных при написании программы 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 $compile\_asm\_lib\_*$, преобразующие соответствующие единицы компиляции собственной реализации функций библиотеки $libc$ на языке ассемблера (см. далее) в перемещаемые объектные файлы с помощью утилиты as.
5. Сценарий $compile\_asm\_lib$, преобразующий все единицы компиляции собственной реализации функций библиотеки $libc$ на языке ассемблера в перемещаемые объектные файлы.
6. Сценарий $combine\_asm\_lib$, преобразующий перемещаемые объектные файлы собственной реализации функций библиотеки $libc$ на языке ассемблера в статическую библиотеку `asm.lib` с помощью утилиты ar.
7. Сценарии вида $compile\_asm\_*$, преобразующие с помощью утилиты as соответствующие единицы компиляции программы на языке ассемблера (см. далее) в перемещаемые объектные файлы, сохраняемые в директории `int`.
8. Сценарий $compile\_asm$, выполняющий компиляцию всех модулей программы на языке ассемблера.

9. 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`, с подключением статической библиотеки `asm.lib`.
10. Сценарий *build_asm*, производящий компиляцию и линковку программы с использованием описанных в п.8-9 сценариев.
11. Сценарий *run_asm* запускает исполняемый файл `solution-asm.exe`, а сценарий *asm* собирает программу на языка ассемблера без использования опций отладки и запускает её.
12. Версии сценариев 8-11 для оптимизированной программы на языке ассемблера (см. далее).
13. Сценарий *build_all* производит сборку всех программ на языке *C* и ассемблера.
14. Сценарий *gen_tests*, запускающий программу генерации тестовых данных `gen.js`.
15. Сценарий *test_stdin*, запускающий тестирование указанных в файле конфигурации тестирующей программы `CONFIG.js` исполняемых файлов с использованием ввода через стандартный поток.
16. Сценарий *test_file*, запускающий тестирование указанных в файле конфигурации тестирующей программы `CONFIG.js` исполняемых файлов с использованием файлового ввода.
17. Сценарий *benchmark*, производящий замеры эффективности работы указанных в файле конфигурации тестирующей программы `CONFIG.js` исполняемых файлов с использованием программы `benchmark.js`.

Программа на языке ассемблера

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 была фактически написана вручную с опорой на сгенерированный компилятором листинг и представлена в директории `asm`. Программа состоит из следующих модулей:

1. `lib`, реализующий необходимые функции стандартной библиотеки языка C с использованием системных вызовов.
   1.1. `read_symbol.asm`, реализующий функцию считывания символа с потока, указываемого переменной $.instream$, с использованием системного вызова sys_read.
   1.2. `read_integer.asm`, реализующий функцию, которая с использованием $read\_symbol$ посимвольно считывает целое число как строку с потока, указываемого переменной $.instream$, преобразует полученную строку в число и возвращает его вместе с некоторой дополнительной информацией о его знаке, размере и первом считанном символе, который не был учтён в числе. При этом функция $read\_integer$ не удовлетворяет $cdecl$ - соглашению о вызовах в ОС $Linux$, что требуется учитывать при её использовании.
   1.3. `read_double.asm`, реализующий функцию считывания числа с плавающей точкой с потока, указываемого переменной $.instream$. При этом функция $read\_double$ возвращает считанное число в соответствии с требованиями $cdecl$, а при работе полагается на $read\_integer$.
   1.4. `integer_to_buffer.asm`, реализующий функцию $integer\_to\_buffer$, которая преобразует переданное целое число в строковое представление, хранящееся в переданном буфере, и возвращает количество символов в нём.
   1.5. `print_integer.asm`, реализующий функцию печати целого числа в поток, указываемый переменной $.outstream$, с помощью системного вызова sys_write. При этом $print\_integer$ использует реализацию $integer\_to\_buffer$.
   1.6. `print_double.asm`, реализующий функцию печати числа с плавающей точкой с точностью 7 знаков после запятой в поток, указываемый переменной $.outstream$, с помощью системного вызова sys_write. При этом $print\_double$ использует реализацию $integer\_to\_buffer$.
2. `main.asm`, который реализует принятие управление от операционной системы, разбор параметров командной строки и организацию работы алгоритма.

2.1. Выделение памяти для хранения дескрипторов используемых потоков ввода и вывода, доступ к которой доступен по именам $.instream$ и $.outstream$, а также установка их значения в соответствии с переданными аргументами командной строки аналогично программе на языке C: для организации переменных используются регистры $r12$, $r13$ и rbx (причём каждый из регистров может соответствовать разным переменным в разных частях программы), - 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 $.instream$ и $.outstream$. По окончании разбора параметров командной строки переменной $random\_input$ соответствует регистр rbx, а переменной do_write - $r12$. Создание переменной $loop$ отложено до того момента, когда она будет непосредственно необходима. Для этого значение переменной $argv$ сохранено в регистре $r13$.

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. Происходит последовательный вызов всех подпрограмм аналогично программе на языке C. Для передачи параметров в соответствии с $cdecl$ использованы регистры процессора rdi (первый целочисленный параметр) и $xmm0$ (первый вещественный параметр).

2.4.1. Вызов подпрограммы $input$ и обработка возвращаемого значения. Эквивалентное представление переменной x в программе на этом этапе - регистр $xmm0$.

2.4.2. Вычисление необходимого количества запусков подпрограммы $solve$ путём проверки второго аргумента командной строки.

2.4.3. Системный вызов $sys\_clock\_gettime$ для получения точного текущего времени системы, что необходимо для организации замера времени работы алгоритма. В связи с особенностями работы вызова, на стеке дополнительно выделяются 40 байт, используемые для следующих задач:

2.4.3.1. Сохранение значения регистра $r12$, чтобы использовать его для поддержания значения регистра $xmm0$, не сохраняемого вызванными функциями.

2.4.3.2. Сохранения текущего времени системы (16 байт) до работы $solve$ при вызове $sys\_clock\_gettime$.

2.4.3.3. Сохранения текущего времени системы (16 байт) после работы $solve$ при вызове $sys\_clock\_gettime$.

2.4.4. Вызов подпрограммы $solve$ один или более раз (в зависимости от результата выполнения п.2.4.4.2). При этом для организации цикла используется регистр $r13$, старое значение которого не нужно на этом этапе.

2.4.5. Для обеспечения корректной передачи ответа, значение регистра $xmm0$ по окончании цикла сохраняется в регистре $r12$, значение которого не изменяется при осуществлении системных вызовов.

2.4.6. Системный вызов $sys\_clock\_gettime$ для получения текущего времени системы по окончании работы алгоритма, его обработка, вычисление времени работы с использованием сохранённых до вызова $solve$ значений и его сохранение в регистре $r13$, использованном вместо переменной cpu_time_used.

2.4.7. Вызов подпрограммы $output$, которая организует печать результата работы в поток $.outstream$, если это необходимо.

2.4.8. Вывод времени работы алгоритма с помощью системного вызова sys_write для печати строк и функции $print\_integer$ для вывода числа.

2.4.9. 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.

2.4.10. Для обработки ошибок организована метка $throw\_error$, передача управления на которую приводит к печати ошибки, адрес строки-описания которой должен быть размещён в регистре rdi в момент передачи управления, в поток вывода с помощью системного вызова sys_write и немедленному завершению программы.

3. `input.asm`, производящий считывание входных данных из потока, дескриптор которого хранится в глобальной переменной $.instream$, или их случайную генерацию:

3.1. Производится определение и организация выбора требуемого способа получения данных с помощью инструкции $cmp\ rdi,\ 0$ и метки $.scanf$.

3.2. Если требуется считывание строки из потока $.instream$, программа использует функцию $read\_double$ для получения числа.

3.3. Если требуется генерация случайных данных, программа использует системный вызов $sys\_getrandom$ для получения случайного целого числа размером 4 байта, после чего с помощью арифметических операций деления и вычитания приводит его к интервалу $[-1;\ 1]$.

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

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

4.2. Аргумент функции передаётся подпрограмме в регистре $xmm0$.

4.3. В начале работы подпрограмма сохраняет значение переменной x в регистре $xmm7$, а также вычисляет значения x^2 и $(1 - x^2)$ в регистрах $xmm6$ и $xmm5$ соответственно, а также производит сравнение $(1 - x^2)$ с 10^{-7}, аналогично программе на языке C.

4.4. Если $|x|$ достаточно близок к единице, происходит проверка знака x и возврат соответствующей константы - $\frac{\pi}{2}$ или $-\frac{\pi}{2}$, загрузка которой в регистр $xmm0$ для возвращаемого значения происходит из статической памяти.

4.5. Происходит загрузка числа 0.0005^2 в регистр $xmm4$ для дальнейшего использования. Это позволяет избежать повторных обращений к памяти в цикле. Также происходит установка начальных значений регистров $xmm0$, $xmm1$ и rcx - аналогов переменных $answer$, g и k соответственно.

4.6. После этого для вычисления ответа организован цикл $loop$, каждая итерация которого выполняет следующие действия:

4.6.1. Вычисление значения $(\frac{g}{1-x^2})^2$ в регистре $xmm2$ и его сравнение со значением регистра $xmm4$. Это позволяет оценить погрешность результата, полученного на предыдущих итерациях, и остановить вычисления, если достаточная точность ответа уже достигнута.

4.6.2. Прибавление к ответу значения очередного члена степенного ряда, полученного на предыдущем шаге, с помощью инструкции $addsd$.

4.6.3. Увеличение номера итерации k на 1 и вычисление значений $2k$, $2k-1$ и $2k+1$, а также $(2k-1)^2$ и $2k(2k+1)$ с помощью инструкций целочисленной арифметики. Преобразование полученных значений в числа с плавающей точкой и размещение их в регистрах $xmm2$ и $xmm3$ соответственно с помощью инструкции $cvtsi2sd$.

4.6.4. Вычисление значения следующего члена степенного ряда с помощью инструкций $mulsd$ и $divsd$ умножения и деления чисел с плавающей точкой соответственно.

4.6.5. Важно отметить, что инструкция $pxor\ xmm2,\ xmm2$ позволяет указать процессору, что предыдущее значение регистра не влияет на результаты следующих операций, что значительно ускоряет программу. Компилятор этим также активно пользуется при генерации ассемблерного листинга даже при отсутствии опций оптимизации.

5. `solve_optimized.asm`, реализующий алгоритм решения задачи более эффективно. Подпрограмма работает аналогично `solve.asm` за исключением ряда оптимизаций, которые были применены с учётом алгебраических свойств используемых операций и решений компилятора:

5.1. Для оптимизации проверки условия цикла принято решение вычислять ответ только для положительных x, а вместо g поддерживать значение выражения $(\frac{g}{1-x^2})$. Это позволяет избавиться от излишних операций при проверке необходимости продолжения вычислений: в регистре $xmm0$ в начале каждой итерации уже содержится необходимое для проведения сравнения значение.

5.1.1. Для обработки знаков числа и ответа использован регистр $r10$, который сохраняет значение знака числа до вычислений, а затем используется для восстановления знака в ответе. Так как $arcsin$ - нечётная функция ($arcsin(-x) = -arcsin(x)$), это не влияет на результат работы подпрограммы.

5.1.2. Для получения абсолютной величины числа использована инструкция $andpd$ побитового И значений регистров типа xmm. Для этого также создана “переменная” $fabs\_mask$, 64-битное представление которой представляет все единицы и лишь один ноль в старшем бите. Таким образом, побитовое И с $fabs\_mask$ лишь устанавливает значение знакового бита числа с плавающей точкой в 0, то есть делает число положительным, не изменяя его абсолютное значение.

5.1.3. Для изменения знака числа с положительного на отрицательный создана переменная $fneg\_mask$, 64-битное представление которой представляет 63 нулевых бита, и лишь один - старший - установленный в единицу. Таким образом, побитовое ИЛИ числа с

плавающей точкой с $fneg\_mask$ не влияет на абсолютную величину числа, но устанавливает его старший - знаковый - бит в единицу, то есть делает число отрицательным.

5.1.4. Для избавления от лишних инструкций умножения и деления, в регистре $xmm0$ принято решение поддерживать значение выражения $(\frac{g}{1-x^2})$, а в регистре $xmm1$ - значение выражения $(\frac{answer}{1-x^2})$. Таким образом, после вычисления результата для получения верного ответа необходимо произвести умножение значения регистра $xmm1$ - $(\frac{answer}{1-x^2})$ - на значение регистра $xmm6$ - $(1-x^2)$. В связи с дистрибутивностью умножения относительно сложения, а также коммутативностью операции умножения чисел, проделанные изменения не влияют на логику остальных операций.

5.2. Было принято решение вместо переменной k в регистре rcx поддерживать три значения: $2k$ в регистре rax, $2k-1$ в rcx и $2k+1$ в rdx, что позволяет избавиться от нескольких инструкции целочисленной арифметики в цикле, заменив их лишь корректировкой значений трёх регистров инструкцией сложения add.

5.3. Принято решение помимо инструкции $movsd$ использовать инструкции $movapd$ и $movq$, которые выполняются другими логическими устройствами процессора, что позволяет производить лучшую конвейеризацию вычислений и даёт небольшое улучшение эффективности программы.

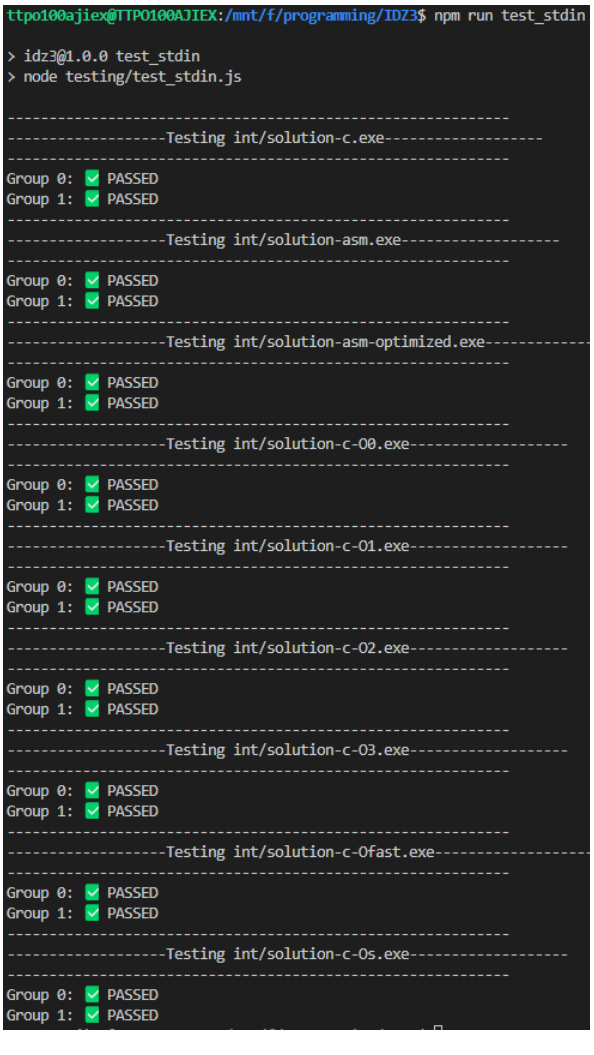
6. `output.asm`, производящий вывод результата в поток, дескриптор которого хранится в глобальной переменной $.outstream$. При этом используется функция $print\_double$ для вывода самого числа и системный вызов sys_write для перевода строки.

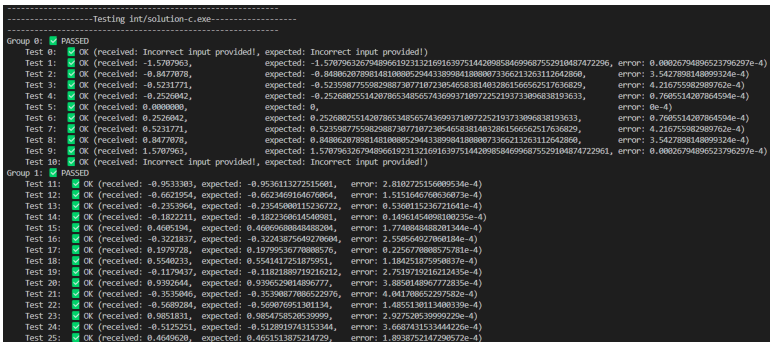
Тестирование программ

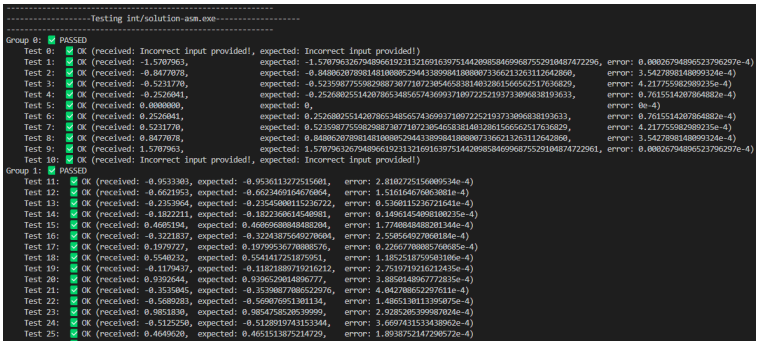
Для тестирования программ использовалось устройство с $12th\ Gen\ Intel\ Core\ i5\ -\ 12600K\ @\ 3.6GHz\ 16\ CPU,\ 32GB\ DDR5\ RAM$ под управлением $WSL\ Debian\ 11$ на $Windows\ 11$ (в директории `int` итогового проекта представлены исполняемые файлы и ассемблерные листинги, полученные именно на этом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

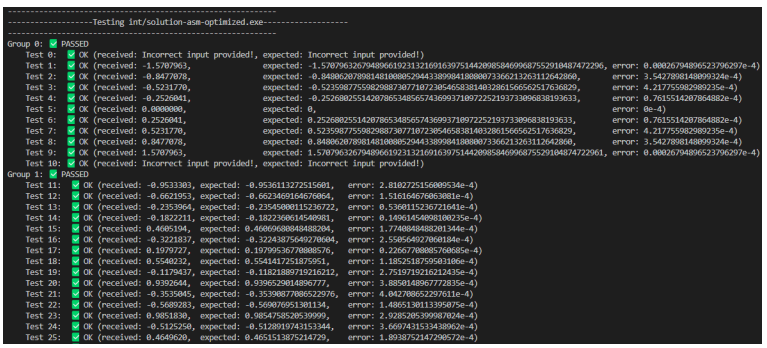
Для получения всех необходимых исполняемых файлов был использован сценарий *build_all*, после завершения работы которого в файл `CONFIG.js` программы-тестировщика были внесены имена всех полученных исполняемых файлов.

Запуск программы *test_stdin* с помощью одноимённого сценария показал, что все исполняемые файлы дают верные ответы на заданных тестах при вводе данных через стандартный поток ввода. Полная информация о запуске представлена в файле `test_stdin.txt` директории `report`.

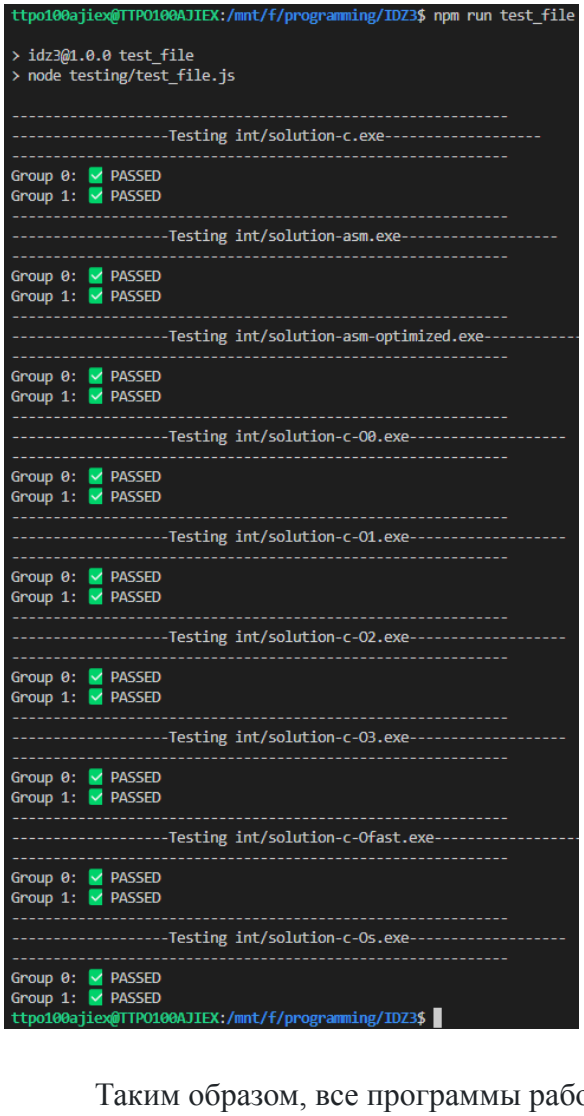


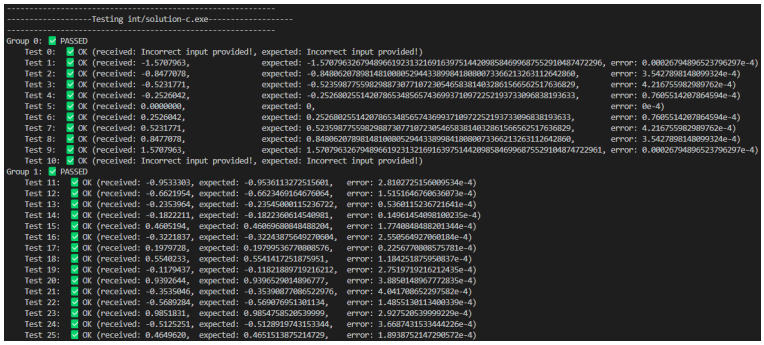


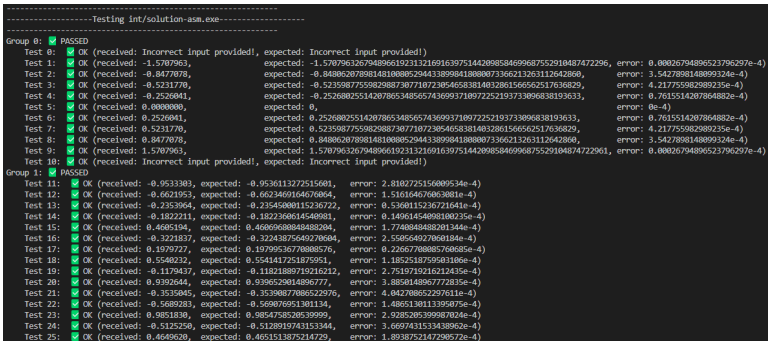


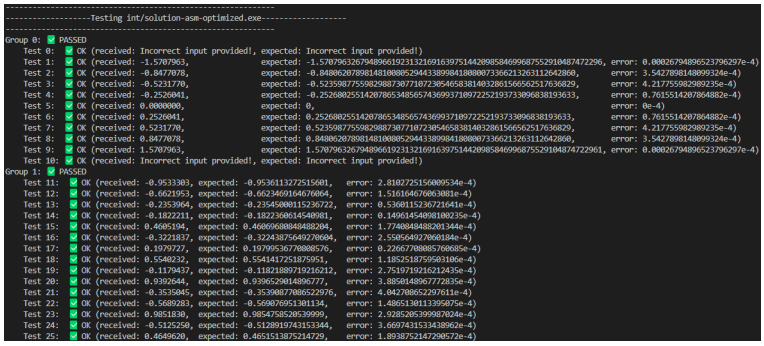


Запуск программы *test_file* с помощью одноимённого сценария показал, что все исполняемые файлы дают верные ответы на заданных тестах при использовании файлового ввода данных. Полная информация о запуске представлена в файле `test_file.txt` директории `report`.









Таким образом, все программы работают корректно независимо от способа ввода и вывода данных.

Сравнение программ по скорости выполнения

Для анализа эффективности программ с помощью одноимённого сценария была запущена программа *benchmark*, полный результат работы которой представлен в файле `benchmark.txt` директории `report`.

```
ttpo100ajiex@TTPO100AJIEX:/mnt/f/programming/IDZ3$ npm run benchmark

> idz3@1.0.0 benchmark
> node testing/benchmark.js

------------------------------------------------------------
------------------Testing int/solution-c.exe------------------
------------------------------------------------------------
Solution CPU time on fixed tests:
    Minimum: 382.74300ms = 0.38274s
    Average: 2414.77234ms = 2.41477s
    Maximum: 22109.72200ms = 22.10972s
Solution IO time on fixed tests:
    Minimum: 388.43633ms = 0.38844s
    Average: 2421.18160ms = 2.42118s
    Maximum: 22115.56502ms = 22.11557s
Solution CPU time on random tests:
    Minimum: 377.11200ms = 0.37711s
    Average: 2447.27576ms = 2.44728s
    Maximum: 31395.19200ms = 31.39519s
Solution IO time on random tests:
    Minimum: 380.29115ms = 0.38029s
    Average: 2450.77172ms = 2.45077s
    Maximum: 31398.55158ms = 31.39855s
------------------------------------------------------------
------------------Testing int/solution-asm.exe------------------
------------------------------------------------------------
Solution CPU time on fixed tests:
    Minimum: 255.66471ms = 0.25566s
    Average: 1804.34322ms = 1.80434s
    Maximum: 18028.86265ms = 18.02886s
Solution IO time on fixed tests:
    Minimum: 261.52295ms = 0.26152s
    Average: 1810.74808ms = 1.81075s
    Maximum: 18035.32136ms = 18.03532s
Solution CPU time on random tests:
    Minimum: 257.32045ms = 0.25732s
    Average: 4914.52991ms = 4.91453s
    Maximum: 123649.31628ms = 123.64932s
Solution IO time on random tests:
    Minimum: 260.28769ms = 0.26029s
    Average: 4917.61370ms = 4.91761s
    Maximum: 123651.92229ms = 123.65192s
------------------------------------------------------------
------------------Testing int/solution-asm-optimized.exe------------
------------------------------------------------------------
Solution CPU time on fixed tests:
    Minimum: 203.77101ms = 0.20377s
    Average: 1113.78219ms = 1.11378s
    Maximum: 11043.41172ms = 11.04341s
Solution IO time on fixed tests:
    Minimum: 210.61981ms = 0.21062s
    Average: 1120.00352ms = 1.12000s
    Maximum: 11049.49730ms = 11.04950s
Solution CPU time on random tests:
```

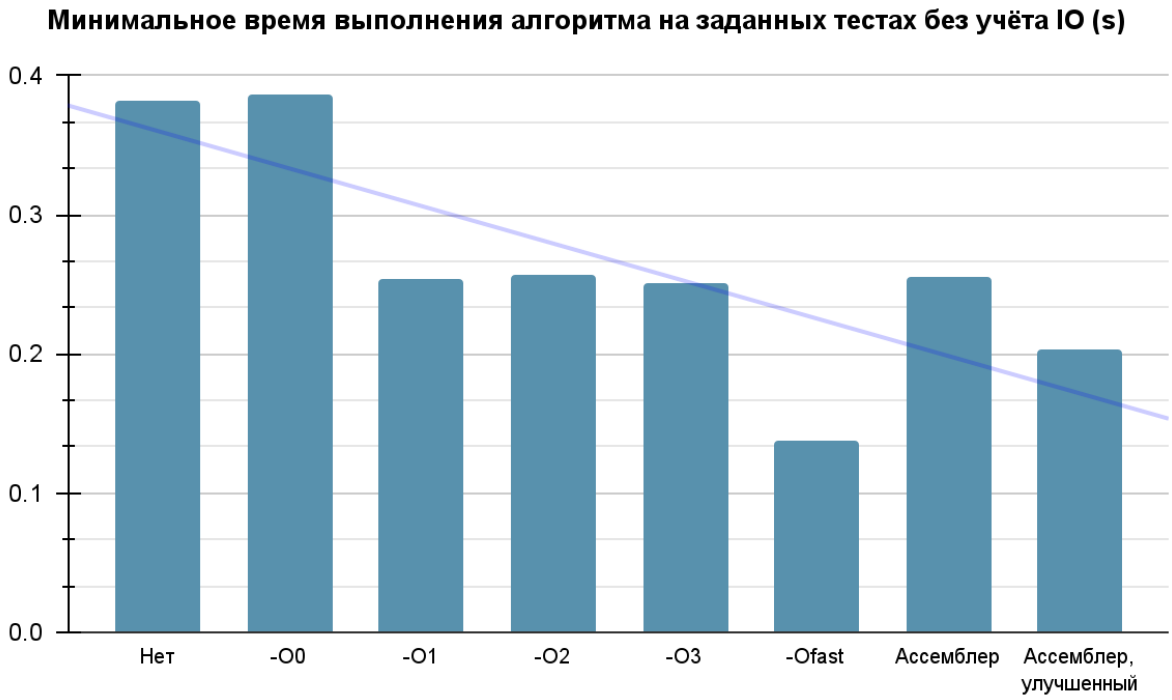
Опции оптимизации	Время выполнения алгоритма на заданных тестах (s)			Время выполнения алгоритма на заданных тестах с учётом IO (s)		
	Минимум	Среднее	Максимум	Минимум	Среднее	Максимум
Нет	0.38274	2.41477	22.10972	0.38844	2.42118	22.11557
-O0	0.38688	2.42721	22.70555	0.39185	2.43224	22.71051
-O1	0.25464	1.74303	17.23298	0.26059	1.74807	17.23763
-O2	0.25639	1.72543	16.84153	0.26075	1.73045	16.84630
-O3	0.25104	1.73060	16.75802	0.25573	1.73597	16.76281
-Ofast	0.13771	1.16512	11.74128	0.14224	1.17009	11.74622
-Os	0.25345	6.00412	87.79603	0.25846	6.00935	87.80120
Ассемблер	0.25566	1.80434	18.02886	0.26152	1.81075	18.03532
Ассемблер, улучшенный	0.20377	1.11378	11.04341	0.21062	1.12000	11.04950

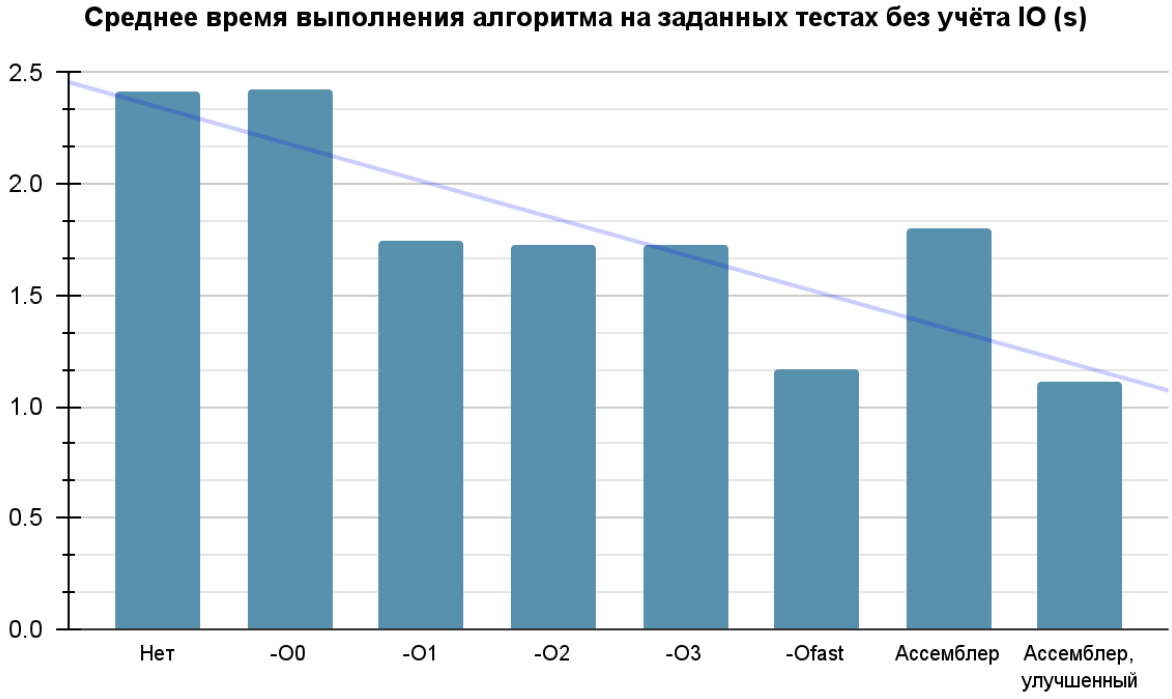
Опции оптимизации	Время выполнения алгоритма на случайных тестах (s)			Время выполнения алгоритма на случайных тестах с учётом IO (s)		
	Минимум	Среднее	Максимум	Минимум	Среднее	Максимум
Нет	0.37711	2.44728	31.39519	0.38029	2.45077	31.39855
-O0	0.38751	2.16528	24.17418	0.39071	2.16873	24.17761
-O1	0.2548	0.49092	2.12488	0.25804	0.49428	2.12786
-O2	0.25647	1.26631	11.12034	0.25998	1.26955	11.12386
-O3	0.25154	1.01445	8.81724	0.25465	1.01782	8.82022
-Ofast	0.13884	0.26354	0.96174	0.14155	0.26679	0.96477
-Os	0.251	2.48726	34.95632	0.25418	2.4905	34.95919
Ассемблер	0.25732	4.91453	123.64932	0.26029	4.91761	123.65192
Ассемблер, улучшенный	0.20537	1.71801	24.24662	0.20903	1.72115	24.25019

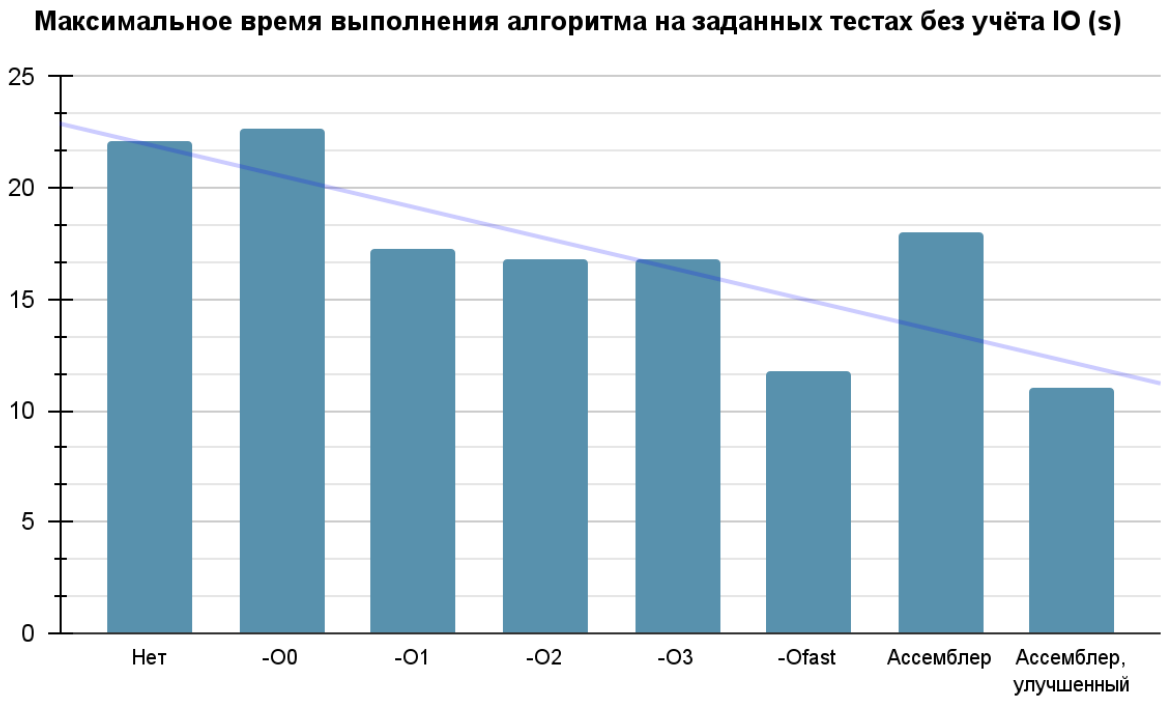
Ожидаемо, время с учётом IO-операций лишь незначительно превышает время выполнения алгоритма без учёта ввода/вывода: количество входных и выходных данных не достаточно мало для оценки эффективности подпрограмм ввода и вывода.

Также ожидаемо, что во времени работы программ на случайных тестах не прослеживается никаких закономерностей, так как время работы алгоритма сильно зависит от введённого числа. Нетрудно заметить, что чем ближе абсолютная величина введённого числа к единице, тем больше членов степенного ряда необходимо просуммировать, тем больше время выполнения программы.

Интерес представляет информация о времени работы программ на фиксированных тестах второй группы. Нетрудно заметить, что программа, оптимизированная компилятором gcc по размеру (опция командной строки Os) явно проигрывает всем остальным, что ожидаемо.







Также интересно, что оптимизации $O1$, $O2$ и $O3$ как для среднего, так и для крайних случаев, не дали заметного улучшения эффективности программы и лишь незначительно превосходят по эффективности написанную вручную ассемблерную программу без оптимизаций.

Оптимизация $Ofast$ же привела к значительному улучшению времени выполнения и явно превосходит ассемблерную программу без оптимизаций. Тем не менее такой результат ожидаем: при оптимизации $Ofast$ компилятор gcc применяет опцию $ffast - math$, которая позволяет значительно улучшить скорость работы за счёт избавления от инструкций, позволяющих отлавливать ошибки при вычислениях с плавающей точкой, а также осуществляет ряд других оптимизаций, которые могут нарушить корректность работы программы. Но как показало тестирование, применённые оптимизации не повлияли на правильность результата, поэтому действительно имеют право на существование и приводят к почти двукратному ускорению программы по сравнению с версией без оптимизаций.

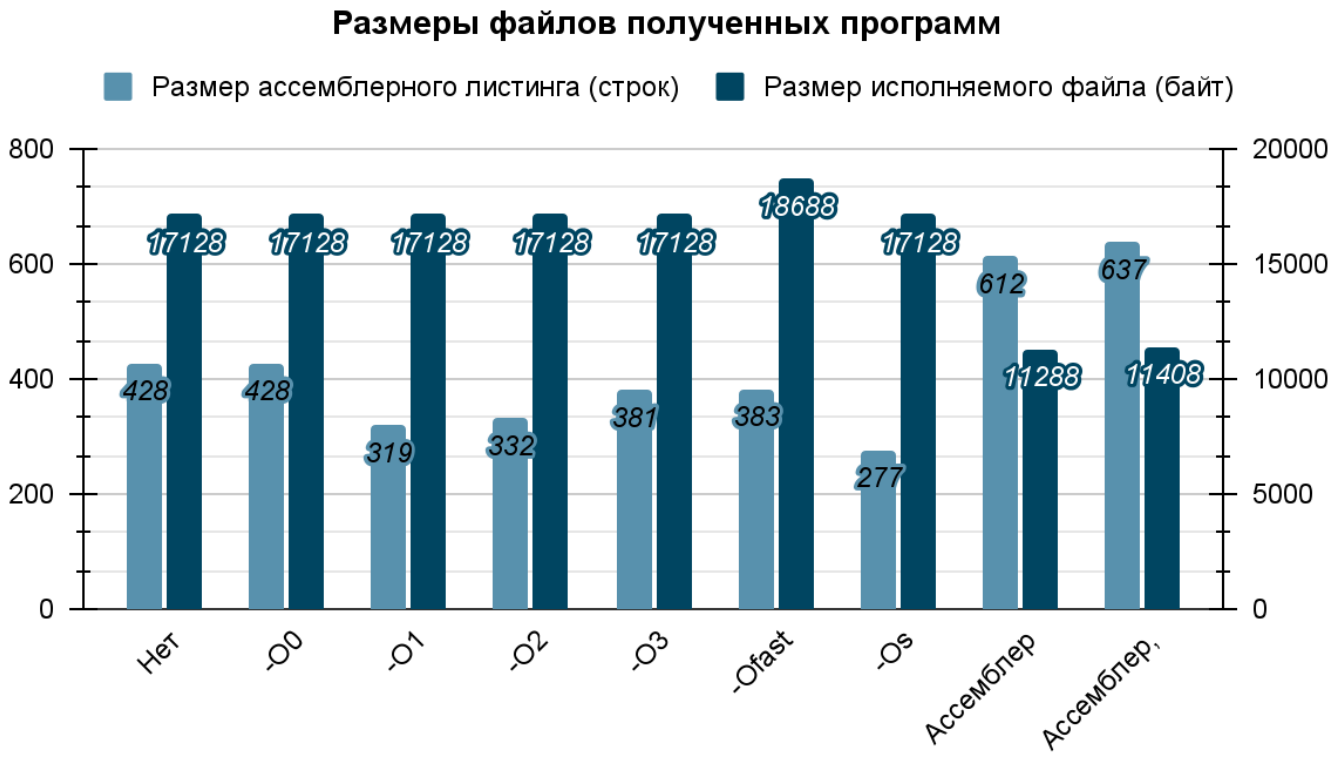
Интересно отметить, что вручную улучшенная ассемблерная программа в среднем и максимальном случаях всё равно превосходит версию, оптимизированную компилятором, несмотря на то, что не реализует методы оптимизации, которые требуют глубоких знаний архитектуры целевого процессора (оптимальный порядок вычислений, “подсказки” логическому устройству процессора, использование более эффективных, современных инструкций и т.д.). Таким образом, можно сделать вывод, что алгоритмические оптимизации программы, проделанные мной при создании улучшенной

версии подпрограммы $solve$, до которых компилятор “не догадался”, всё ещё способны привести к значительному выигрышу по производительности, в то время как архитектурные оптимизации зачастую дают заметный выигрыш лишь на “маленьких” тестах.

Сравнение программ по размеру

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

Опции оптимизации	Размер ассемблерного листинга (строк)	Размер исполняемого файла (байт)
Нет	428	17128
-O0	428	17128
-O1	319	17128
-O2	332	17128
-O3	381	17128
-Ofast	383	18688
-Os	277	17128
Ассемблер	$34 + 67 + 26 + 50 + 57 + 21 + + 39 + 222 + 22 + 74 = 612$	11288
Ассемблер, улучшенный	$34 + 67 + 26 + 50 + 57 + 21 + + 39 + 222 + 22 + 99 = 637$	11408



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, превосходит сгенерированную компилятором во всех аспектах: и по скорости выполнения, и по размеру исполняемого файла.