Take-Home Exam 1

(Deadline: 23rd of November 2022, Wednesday, 23:59)

1 Introduction

Physical simulations have always been one of the most frequent applications of programming. At the heart of these simulations, mathematical constructs such as series and derivatives are often computed. Additionally, such simulations may deal with large amounts of data, which sometimes has to be compressed. In this take-home exam, you are given four tasks which relate these concepts.

2 Tasks

2.1 Simple Calculator

In this task, you will develop a program that acts as a simple calculator. Your program should calculate the result of the given arithmetic operation on two given numbers. The input to your program consists of two real numbers separated by the symbol of an arithmetic operation (`+`, `-`, `*` or `/`). Your program should print out the result of the operation on the two numbers, with 3 decimal digits (rounded if necessary).

Input format will be:

```
A o B
```

where `A` and `B` are the two numbers and `o` is the character describing the arithmetic operation. The fields will be separated by space and followed by a newline. Output format will be:

```
R
```

where `R` is the result. Your program should terminate after printing this.

We suggest using the `double` data type for representing the number. We naturally expect some numerical inaccuracies with your computation, therefore we will consider your program's answer correct if its error is within 1% of our answer.

Example Input	Example Output
`5.25 * 4.42`	`23.205`
`4.45 / 2.18`	`2.041`

2.2 Generalized Fibonacci Sequence

In mathematics, *the Fibonacci sequence* is an infinite sequence of integers such that its first two elements are 1 and its every other element is obtained by summing up the two previous elements in the sequence:

$$1, 1, 2, 3, 5, 8, 13, 21, 34, \ldots$$

The nth element in this sequence is known as the *nth Fibonacci number*, denoted by F_n. Thus, we can define F_n recursively as follows:

$$F_n = \begin{cases} 1 & \text{if } n = 1 \\ 1 & \text{if } n = 2 \\ F_{n-2} + F_{n-1} & \text{otherwise} \end{cases}$$

Suppose that we generalize this sequence by allowing the following to be specified:

- The first and the second elements of the sequence (let us call these a and b),
- The coefficients of F_{n-2} and F_{n-1} in the recursive formula (let us call these c and d),
- The operation applied to F_{n-2} and F_{n-1} in the recursive formula (let us denote this by $\oplus$).

In other words, we change the formula for F_n as:

$$F_n = \begin{cases} a & \text{if } n = 1 \\ b & \text{if } n = 2 \\ cF_{n-2} \oplus dF_{n-1} & \text{otherwise} \end{cases}$$

This definition generates a sequence for any given quintuple $(a, b, c, d, \oplus)$. For instance, for $(3, 4, 2, 1, -)$, we get the following sequence:

$$3, 4, 2, 6, -2, 14, -18, 46, -82, \ldots$$

Write a program to find the nth (generalized) Fibonacci number F_n, given $(a, b, c, d, \oplus)$.

Input format will be:

```
a b c d o n
```

where

- `a`, `b`, `c`, `d` are integers (within int range) that describe a, b, c, d,
- `o` is either the `+` character or the `-` character, describing $\oplus$,
- `n` is a positive integer (≤ 1000) describing n.

The fields will be separated by space and followed by a newline. Output format will be

```
F
```

where `F` is an integer describing F_n. You will be guaranteed that F_i, cF_i, and dF_i fit into an int for all $i \leq n$.

Example Input	Example Output
`3 4 2 1 - 8`	`46`
`-1 2 -1 2 + 1000`	`2996`

2.3 Run-Length Encoding

Run-length encoding (RLE) is a form of lossless data compression in which “runs” of symbols (i.e., consecutive repetitions of the same symbol) are written as pairs of an integer count and the symbol. For instance, run-length encoding of the text:

```
AAAAAAAAAAAABAAAAAAAAAAAABBBCCCCCCCCCCCCCCCCCCCCCCCCBAAAAAAAAAAAAAA
```

would be:

```
12A1B12A3B24C1B14A
```

Write a program that performs run-length encoding on a given word. Input format will be

```
w
```

where `w` is a word that consists at most 1000 uppercase English letters. It will be followed by a newline character, which your program can read and compare against `'\n'` to detect that the input ended. Output format will be

```
r
```

where `r` is the run-length encoding of `w`. This may be followed by a newline or not.

Example Input	Example Output
`TTTTUUUUUUUUUUUUUUTT`	`4T14U2T`
`ABCDEAABBCCDDEE`	`1A1B1C1D1E2A2B2C2D2E`
`AAAAAAAAAA`	`10A`

2.4 Polynomial Derivatives

A polynomial is a function that maps its single parameter to a summation of its non-negative integer powers with coefficients. Basic examples of polynomials include linear functions such as $f(x) = 2x + 3$ and quadratic functions such as $f(x) = 3x^2 - 5x + 4$.

A *polynomial of degree n* is a function of the general form:

$$f(x) = a_n x^n + a_{n-1} x^{n-1} + \ldots + a_2 x^2 + a_1 x + a_0$$

The *derivative $f'(x)$* of a polynomial $f(x)$ given in the general form above is obtained by replacing every term of the form ax^k with akx^{k-1}. In other words,

$$f'(x) = a_n n x^{n-1} + a_{n-1}(n-1)x^{n-2} + \ldots + a_2 2x + a_1$$

The derivative operation can be repeated multiple times. In particular, the *kth-order derivative* of $f(x)$, denoted by $f^{(k)}(x)$ is the function obtained from taking the derivative of $f(x)$ k times. For instance, if

$$f(x) = 5x^3 - 2x^2 + x - 4$$

then we can write:

$$\begin{array}{rcrcl} f(x) & = & f^{(0)}(x) & = & (5)x^3 + (-2)x^2 + (1)x^1 + (-4)x^0 \\ f'(x) & = & f^{(1)}(x) & = & (15)x^2 + (-4)x^1 + (1)x^0 \\ f''(x) & = & f^{(2)}(x) & = & (30)x^1 + (-4)x^0 \\ f'''(x) & = & f^{(3)}(x) & = & (30)x^0 \\ f''''(x) & = & f^{(4)}(x) & = & 0 \\ f'''''(x) & = & f^{(5)}(x) & = & 0 \\ & & \ldots & & \end{array}$$

Write a program that computes the kth-order derivative of a given polynomial $f(x)$ for a given x, i.e., $f^{(k)}(x)$.

Format of the input will be:

```
x k n a_n ... a_0
```

where

- `x` is a real number describing the value of x,
- `k` is a non-negative integer (≤ 20) describing the order of the derivative, k,
- `n` is a non-negative integer (≤ 20) describing the degree of the polynomial, n,
- and `a_n ... a_0` are $(n + 1)$ real numbers that lists the coefficients of the polynomial, $a_n, \ldots a_0$.

Note that if the polynomial is missing the term for the ith power of x, then the corresponding coefficient would be 0, i.e., $a_i = 0$. The fields will be separated by space and followed by a newline.

Output format will be

```
f
```

where `f` is a real representing $f^{(k)}(x)$, with exactly 3 decimal digits (rounded if necessary). We suggest using the `double` data type for representing numbers in this task. We naturally expect some numerical inaccuracies with your computation, therefore we will consider your program's answer correct if its error is within 1% of our answer.

Since you have not learned any mechanisms that can store arbitrarily large data, you need to do the derivative computation on the fly, term by term, while reading the polynomial from the input. Notice that the final (repeated) derivative can be obtained by summing the individual (repeated) derivatives of each term of the polynomial.

Example Input	Example Output
`2 2 4 3 0 2 1 -1`	`148.000`
`-0.01 4 6 -2 0.5 -1.5 1 1 0 0`	`-36.672`

In the first example, notice the polynomial and its (repeated) derivatives are as follows:

$$\begin{array}{rcl} f(x) & = & (3)x^4 + (0)x^3 + (2)x^2 + (1)x^1 + (-1)x^0 \quad = \quad 3x^4 + 2x^2 + x - 1 \\ f^{(1)}(x) & = & (12)x^3 + (0)x^2 + (4)x^1 + (0)x^0 \quad = \quad 12x^3 + 4x \\ f^{(2)}(x) & = & (36)x^2 + (0)x^1 + (4)x^0 \quad = \quad 36x^2 + 4 \end{array}$$

Therefore,

$$f^{(2)}(2) = 36 \times 2^2 + 4 = 148$$

In the second example, notice the polynomial and its (repeated) derivatives are as follows:

$$\begin{aligned}
f(x) &= (-2)x^6 + (0.5)x^5 + (-1.5)x^4 + (1)x^3 + (1)x^2 + (0)x^1 + (0)x^0 &= -2x^6 + 0.5x^5 - 1.5x^4 + x^3 + x^2 \\
f^{(1)}(x) &= (-12)x^5 + (2.5)x^4 + (-6)x^3 + (3)x^2 + (2)x^1 + (0)x^0 &= -12x^5 + 2.5x^4 - 6x^3 + 3x^2 + 2x \\
f^{(2)}(x) &= (-60)x^4 + (10)x^3 + (-18)x^2 + (6)x^1 + (2)x^0 &= -60x^4 + 10x^3 - 18x^2 + 6x + 2 \\
f^{(3)}(x) &= (-240)x^3 + (30)x^2 + (-36)x^1 + (6)x^0 &= -240x^3 + 30x^2 - 36x + 6 \\
f^{(4)}(x) &= (-720)x^2 + (60)x^1 + (-36)x^0 &= -720x^2 + 60x^1 - 36
\end{aligned}$$

Therefore,

$$f^{(4)}(-0.01) = -720 \times (-0.01)^2 + 60 \times (-0.01) - 36 = -36.672$$

3 Submission & Rules

- Submit a separate C source file (with given name in ODTÜClass/VPL system) for each of the tasks.
- We will use VPL system for submission (which will be case for lab exams as well.) You can use the system directly as an editor or work locally and upload your source files to the system later.
- **Run** and **debug** modes in VPL system are disabled since we have 4 main functions exist in submission files. You can directly use **evaluate** mode in VPL to run all tasks at once and see your grade. You can evaluate your submission infinitely many times.
- Late submission ***IS NOT*** allowed. **Please, do not ask for any deadline extensions**.
- In your solutions, you should not use any library functions other than those in *"stdio.h"*.
- You are expected to use the constructs that we have shown in the lectures so far (variables, data types, I/O functions, if, else, else-if, goto, while, for) or their derivatives (switch, do-while). It is ***FORBIDDEN*** to use arrays, multi-dimensional arrays, pointers, dynamic memory allocation or to define your own functions other than `main()`. (This is a warning for students who repeat the course.)
- We will compile your code with the following command:

```
gcc yourCode.c -o yourExecutable -g -std=c17 -Wall -Wextra -Wpedantic
```

- If you are working on a command prompt, you can likely feed input files to your program instead of typing inputs. This way, you can test your inputs faster. For instance, you can type:

```
> ./yourExecutable < yourInput.txt
```

- Your grade will be given by an auto-grader upon submission. The grade will be determined by running your code against several input files including, but not limited to the test cases given to you as an example. It is possible to get a partial grade from each task.
- You should stick to the output format described in the tasks. Your program's output should not contain any unnecessary characters or whitespace. (The only exception to this is that you may optionally print a newline character at the end of your output.) It is your responsibility to check the correctness of the output in terms of whether it contains any invalid invisible characters. This also means you are not supposed to print any prompts for input.
- Your programs should terminate with an exit code of 0.
- Your programs should be reasonably well-written such that they do not run slowly. The time limit we impose in the auto-grade is more than enough with respect to the tasks that you are assigned; however, if you have a bug or a code that is extremely slow, your program may exceed the time limit and you will not receive any points.
- Grade from the auto-grader is ***NOT*** final. We may re-evaluate your submissions in black-box or white-box manner and adjust your grade. Submissions that do not reasonably attempt to solve the tasks may lose points.
- As usual, you are expected to complete your solutions in ***academic integrity***. We have zero-tolerance policy for cheating. People involved in cheating will receive a grade of zero and may be prosecuted according to the university regulations. *Sharing code (in any amount) with each other or using third party code is strictly forbidden.* Be aware that there are "very advanced tools" that can detect this type of cheating.

Good Luck!