CoE 164

Computing Platforms

Assessments Week 01

Academic Period: 2nd Semester AY 2023-2024
Workload: 3 hours
Synopsis: Rust programming basics

SE Week 01A

This assessment will probably be your first program in Rust.

This is worth 40% of your grade for this week

Problem Statement

Welcome to EEE! Enjoy your stay.

Your first foray into EEEI includes circuit analysis. It is essential to simplify a resistor network to determine the total current supplied by a source. When resistors are connected in series, the total resistance of the network is simply their sum. On the other hand, when resistors are connected in parallel, the total resistance of the network is the reciprocal sum of the sum of reciprocals of each resistor.

Your task is to determine the total resistance of a bag of resistors you have found in one of the laboratories if 1) they are all connected in series, and 2) they are all connected in parallel. Since the resistor values can be quite "ugly", you have decided to write a simple program that will compute the equivalent resistance.

Input

The first line of the input consists of a number N indicating the number of resistor values that will follow. The next N lines contain R_n, the resistor value of the nth resistor in the network.

Output

The output should consist of two lines indicating the series and then the parallel equivalent value of the resistor network separated by a new line respectively. These values should be rounded down to 4 decimal places.

Constraints

Input Constraints

$N \leq 5$

$R_n \in R^+$

$0 < R_n \leq 10000$

You can assume that all of the inputs are well-formed and are always provided within these constraints. You are not required to handle any errors.

Functional Constraints

None

Sample Input/Output

Sample Input 1:

```
3
10
100
1000
```

Sample Output 1:

```
1110.0000
9.0090
```

Sample Input 2:

```
4
100
100
100
100
```

Sample Output 2:

```
400.0000
25.0000
```

Steps

1. Write your program in Rust. Compile and make sure that there are no syntax errors.
2. Make sure to accept input via standard input and print your output via standard output. For example, you can write your inputs into a text file named `in_pub.txt` and the expected and correct outputs into another text file named `out_pub_ans.txt`. If the compiled program is named `wa`, and you want the

printed output to be saved into a file named `out_pub.txt`, you can execute the following command from the following terminals to run it:

Windows (Powershell): `cat in_pub.txt | ./wa.exe | Out-File out_pub.txt`
Linux/macOS (bash, zsh): `./wa < in_pub.txt > out_pub.txt`

Then, compare the program output with the reference output by executing the following commands:

Windows (Powershell): `Compare-Object (gc out_pub.txt) (gc out_pub_ans.txt)`
Linux/macOS (bash, zsh): `diff out_pub.txt out_pub_ans.txt`

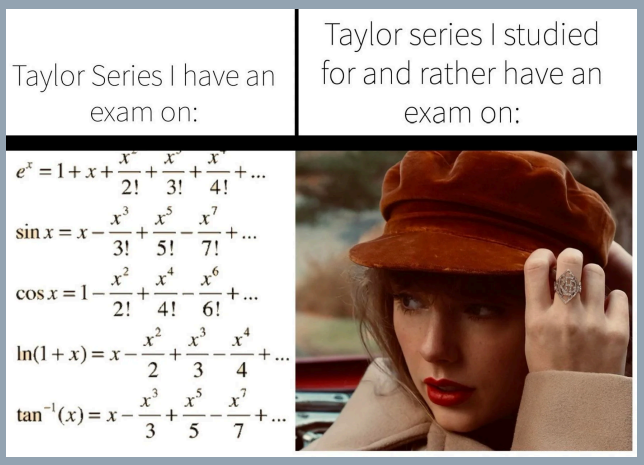
3. Submit a copy of the source code to the Week 01A submission bin. Make sure that you attach one (1) file in the bin containing the Rust source code with a `.rs` extension (preferably named `w01a.rs`). **Please do not send compressed files!**

SE Week 01B

This assessment will help you be familiar with math operations, loops, and conditionals in Rust.

This is worth 30% of your grade for this week.

Problem Statement



Hi, it's me, I'm the problem, it's me.

Taylor is curious on how calculators compute transcendental functions. She learned in EEE 143 that multiplication and division are done through a mix of repetitive addition and subtraction. She opened her digital calculator and found out that, *indeed*, the calculator computes using only addition and subtraction operations.

Looking back on her songs, she played her favorite song *22*… wait, that's also her favorite subject: Math 22! She remembered the concept of the use of Taylor polynomials to approximate the value of a function centered at some point a. However, the Taylor polynomial is only accurate to a certain extent. As reference, the Taylor series of a function $f(x)$ is given by:

$$f(x) = \sum_{n=0}^{\infty} \frac{f^{(n)}a}{n!}(x - a)^n$$

To keep things simple, we set $a = 0$ and limit ourselves in computing three transcendental functions: the natural exponent, sine, and cosine. The equivalent Taylor series expansion of each transcendental function given the input x is shown in the table below.

Function $f(x)$	**Taylor Series Expansion**
e^x	$\sum_{n=0}^{+\infty} \frac{x^n}{n!}$
$\sin(x)$	$\sum_{n=0}^{+\infty} \frac{(-1)^n}{(2n+1)!} x^{2n+1}$

$\cos(x)$	$\sum_{n=0}^{+\infty} \frac{(-1)^n}{(2n)!} x^{2n}$

Note that the series is infinite, but we only have a limited amount of time and patience to compute the approximation. Hence, for all of the above functions, we are going to sum only the first n terms such that n is the floored minimum value for which $\frac{x^n}{n!} \geq R_{desired}$. $R_{desired}$ is the approximation error and keep $0 \leq x \leq 1$ for it to be applicable.

Input

The input contains the number of functions to be evaluated, T. Followed by the transcendental function to be evaluated, that is: `e` for the natural exponential function, `s` for the sine function, and `c` for the cosine function. Then the approximation error $R_{desired}$ that we want, and finally the input to the function x.

Output

The output consists of T lines, with each line denoting the corresponding test case in the input. Each line should follow the format: `Function #T: <result>`, where `T` is the serial of the test case starting from 1, and `<result>` indicates the value of the function rounded down to 2 decimal places.

Constraints

$0 \leq T$ such that $T \in \mathbb{Z}^+$
$\text{function} \in \{\text{e}, \text{s}, \text{c}\}$
$0.0001 \leq R_{desired} \leq 0.001$ such that $0 < n < 21$
$0 \leq x \leq 1$

You can assume that all of the inputs are well-formed and are always provided within these constraints. You are not required to handle any errors.

Sample Input/Output

Sample Input 1:

```
6
e 0.0001 1
s 0.0001 1
c 0.0001 1
e 0.001 0.5
s 0.001 0.5
c 0.001 0.5
```

Sample Output 1:

```
Function #1: 2.72
```

```
Function #2: 0.84
Function #3: 0.54
Function #4: 1.65
Function #5: 0.48
Function #6: 0.88
```

Steps

1. Write your program in Rust. Compile and make sure that there are no syntax errors.
2. Make sure to accept input via standard input and print your output via standard output. For example, you can write your inputs into a text file named `in_pub.txt` and the expected and correct outputs into another text file named `out_pub_ans.txt`. If the compiled program is named `wa`, and you want the printed output to be saved into a file named `out_pub.txt`, you can execute the following command from the following terminals to run it:

   Windows (Powershell): `cat in_pub.txt | ./wa.exe | Out-File out_pub.txt`
   Linux/macOS (bash, zsh): `./wa < in_pub.txt > out_pub.txt`

   Then, compare the program output with the reference output by executing the following commands:

   Windows (Powershell): `Compare-Object (gc out_pub.txt) (gc out_pub_ans.txt)`
   Linux/macOS (bash, zsh): `diff out_pub.txt out_pub_ans.txt`
3. Submit a copy of the source code to the Week 01B submission bin. Make sure that you attach one (1) file in the bin containing the Rust source code with a `.rs` extension (preferably named `w01b.rs`). **Please do not send compressed files!**

SE Week 01C

This assessment will help you be familiar with arrays and conditionals in Rust.

This is worth 30% of your grade for this week

Problem Statement

The Philippines encodes mobile phone numbers as an eleven-digit (11) number. When dialed locally, a number starts with the digit 0, and then a three-digit prefix that indicates which mobile network provider the number originated from, and then finally a seven-digit unique identifier. The figure below shows a visualization of a mobile phone number.

Due to the tendency of the Filipino to be "terminally" online, there is an average of one mobile number per person in the country, and a single prefix for each of the dominant three network providers as of this time of writing cannot accommodate such number of subscribers. In addition, due to a changing business landscape, some old network providers have been absorbed by larger ones. Hence, a service provider usually "owns" multiple prefixes. Finally, to complicate matters, some network providers share a prefix. The first digit in the unique identifier is then used to disambiguate between those providers.

The following lists the dominant network providers and their corresponding prefixes. Note that these prefixes have their initial 0 digit at the leftmost omitted and that four-digit prefixes include the first digit of the unique id as its last digit.

- **Globe/TM**: 817, 905, 906, 915, 916, 917, 926, 927, 935, 936, 937, 945, 953, 954, 955, 956, 965, 966, 967, 975, 976, 977, 978, 979, 995, 996, 997, 9173, 9175, 9176, 9178, 9253, 9255, 9256, 9257, 9258
- **Smart/Sun/TNT**: 922, 923, 924, 925, 931, 932, 933, 934, 940, 941, 942, 943, 973, 974, 907, 909, 910, 912, 930, 938, 946, 948, 950, 908, 918, 919, 920, 921, 928, 929, 939, 946, 947, 949, 951, 961, 998, 999
- **DITO**: 895, 896, 897, 898, 991, 992, 993, 994

Your task is to create a program for a phonebook app that will output the network provider where the mobile number is first acquired. Since the app can include international numbers, the mobile number is to be displayed such that the starting 0 digit is replaced with the

country code `+63`. Then, the prefix, the first three (3) digits of the unique id, and the last four (4) digits of the unique id are separated by a single space.

Input

The input starts with a number T on a single line denoting the number of mobile numbers. T lines then follow, with each line M denoting a eleven-digit (11) mobile phone number written as a nonnegative integer.

Output

The output consists of T lines, with each line denoting the corresponding mobile number in the input. Each line should start with the text $Case\ \#t$: where t is the serial of the test case starting from 1. The text after the colon should either be one of the following:

- `Invalid`
  - The corresponding mobile number does not have a valid prefix
- `<network> | +63 <prefix> <uid_left> <uid_right>`
  - The corresponding mobile number has a valid prefix
  - `<network>` should only be from one of the following values
    - `Globe/TM`
    - `Smart/Sun/TNT`
    - `DITO`
  - `<prefix>` should have exactly three (3) digits
  - `<uid_left>` should have exactly three (3) digits
  - `<uid_right>` should have exactly four (4) digits

Constraints

Input Constraints

$T \leq 100$
$|M| = 11$
$M[0] = 0$

You can assume that all of the inputs are well-formed and are always provided within these constraints. You are not required to handle any errors.

Functional Constraints

You are **required** to have the following function signatures, their arguments in order, and their return values:

- `get_network_from_prefix()` - return a number representing which network provides numbers with the given prefix
  - Arguments
    - `prefix` - `u64` representing the first four (4) digits of a mobile number
    - `last_digit` - `u64` representing the first digit after the prefix

    - Return value - `u64` representing which network provides numbers with the given `prefix`; it can only be from one of the following values:
        - 0 - Invalid prefix
        - 1 - Globe/TM
        - 2 - Smart/Sun/TNT
        - 3 - DITO
- `main()` - entry point to the program
    - Arguments
        - None
    - Return value
        - None
    - Additional constraints
        - Input and output parsing should be done here

Failure to follow these functional constraints will mark your code with a score of zero.

Sample Input/Output

Sample Input 1:

```
2
09168756982
07852165966
```

Sample Output 1:

```
Case #1: Globe/TM | +63 916 875 6982
Case #2: Invalid
```

Sample Input 2:

```
4
09161234567
09087654321
08982536587
07809999999
```

Sample Output 2:

```
Case #1: Globe/TM | +63 916 123 4567
Case #2: Smart/Sun/TNT | +63 908 765 4321
Case #3: DITO | +63 898 253 6587
Case #4: Invalid
```

Sample Input 3:

```
2
09251523558
09253523558
```

Sample Output 3:

```
Case #1: Smart/Sun/TNT | +63 925 152 3558
Case #2: Globe/TM | +63 925 352 3558
```

Steps

1. Write your program in Rust. Compile and make sure that there are no syntax errors.
2. Make sure to accept input via standard input and print your output via standard output. For example, you can write your inputs into a text file named `in_pub.txt` and the expected and correct outputs into another text file named `out_pub_ans.txt`. If the compiled program is named `wa`, and you want the printed output to be saved into a file named `out_pub.txt`, you can execute the following command from the following terminals to run it:

   Windows (Powershell): `cat in_pub.txt | ./wa.exe | Out-File out_pub.txt`
   Linux/macOS (bash, zsh): `./wa < in_pub.txt > out_pub.txt`

   Then, compare the program output with the reference output by executing the following commands:

   Windows (Powershell): `Compare-Object (gc out_pub.txt) (gc out_pub_ans.txt)`
   Linux/macOS (bash, zsh): `diff out_pub.txt out_pub_ans.txt`
3. Submit a copy of the source code to the Week 01C submission bin. Make sure that you attach one (1) file in the bin containing the Rust source code with a `.rs` extension (preferably named `w01c.rs`). **Please do not send compressed files!**