A3Q1 - Iteration Calculation

Calculate the value of **e** by following iteration:

$$\sum_{n=0}^{\infty} \frac{1}{n!} = \frac{1}{0!} + \frac{1}{1!} + \frac{1}{2!} + \frac{1}{3!} + \frac{1}{4!} + \frac{1}{5!} + \cdots$$

Input a **double-precision float** number which represents a precision threshold.

Your program should terminate the iteration when the difference between two successive iterations is smaller than the precision threshold.

Print the value of e (as **double-precision float**).

NOTES: all the numbers in the calculation are taken as double.

Sample Testcases

Input #1 Copy

```
10
```

Output #1 Copy

```
1.0
```

Input #2 Copy

```
1
```

Output #2 Copy

```
2.0
```

Input #3 Copy

```
0.01
```

Output #3 Copy

```
2.708333333333333
```

Input #3 Copy

```
0.01
```

Output #3 Copy

```
2.708333333333333
```

Input #4 Copy

```
0.0000248015873015873
```

Output #4 Copy

```
2.71827876984127
```

A3Q2 - IEEE745 Format

Get the input string (whose length is not exceed 10), take it as a **single-precision** floating number's IEEE745 code in hexadecimal, output its **sign bit**, **exponent** (in **hexadecimal**), **fraction part** (in **hexadecimal**) of IEEE745 code and its value.

If the input string is **invalid**, print the error info "`invalid input`" and exit the program.

NOTES:

- If the length is smaller than 10, take the string as the lower part of code of single float number, complete the higher part of the code with `0`s.
- If the first two character of the input string is neither `0x` nor `0X`, output "`invalid input`".
- If there is invalid hexadecimal character in the input string, output "`invalid input`".
- It is suggested to use "`print float`" syscall to print the value.

Sample Testcases

Input #1 Copy

```
0xbf800000
```

Output #1 Copy

```
s: 1, e: 0x7f, f: 0x000000, value: -1.0
```

Input #2 Copy

```
0x7f800000
```

Output #2 Copy

```
s: 0, e: 0xff, f: 0x000000, value: Infinity
```

0x7f800000	s: 0, e: 0xff, f: 0x000000, value: Infinity

Input #3 Copy

```
0x7fc00000
```

Output #3 Copy

```
s: 0, e: 0xff, f: 0x400000, value: NaN
```

Input #4 Copy

```
0x414a0000
```

Output #4 Copy

```
s: 0, e: 0x82, f: 0x4a0000, value: 12.625
```

Input #5 Copy

```
0x1
```

Output #5 Copy

```
s: 0, e: 0x00, f: 0x000001, value: 1.4E-45
```

Input #6 Copy

```
0x1yz
```

Output #6 Copy

```
invalid input
```

Input #7 Copy

```
123
```

Output #7 Copy

```
invalid input
```

A3Q3 - Division

Define a macro which is named as " `divSCheck` " with three parameter `(%f1, %f2, %f3)` , it is expected to implement following function:

1. Firstly check the value in register `%f3` , if it is **zero**, set the value of register `$k0` to be `11` , trig the exception.
2. Secondly divider the **single-precision** floating data in register `%f2` by the **single-precision** floating data in register `%f3` and store the result to the register `%f1` .

NOTES:

The code of Q3 and Q4 would be used in the following code:

```
.text
li $v0,6
syscall
mov.s $f20,$f0
li $v0,6
syscall
mov.s $f21,$f0
divSCheck($f12,$f20,$f21)
li $v0,2
 syscall
 li $v0,10
 syscall
```

Before or after the use of " `divSCheck` ", there would be some registers in Coprocessor1 be used. (Tips: using Stack to protect the 'old' value in this registers before change its)

Sample Testcases

Input #1 Copy

```
1.0 2.0
```

Output #1 Copy

```
0.5
```

Input #2 Copy

```
1.0 0
```

Output #2 Copy

```
exception:divisor is 0.0 Infinity
```

Input #3 Copy

```
0 0
```

Output #3 Copy

```
exception:divisor is 0.0 NaN
```

A3Q4 - Exception Handler

Define an **exception handler** which is sensitive with the exception event (the exception code is `11` in register `$k0`), in this situation it **report** the exception reason "`exception: divisor is 0.0`" then **return** to the **normal procession**.

NOTES:

The code of Q3 and Q4 would be used in the following code:

```
.text
li $v0,6
syscall
mov.s $f20,$f0
li $v0,6
syscall
mov.s $f21,$f0
divSCheck($f12,$f20,$f21)
li $v0,2
syscall
li $v0,10
syscall
```

Before or after the use of "`divSCheck`", there would be some registers in Coprocessor1 be used. (Tips: using Stack to protect the 'old' value in this registers before change its)

Sample Testcases

Input #1 Copy

```
1.0 2.0
```

Output #1 Copy

```
0.5
```

Input #2 Copy

```
1.0 0
```

Output #2 Copy

```
exception:divisor is 0.0 Infinity
```

Input #3 Copy

```
0 0
```

Output #3 Copy

```
exception:divisor is 0.0 NaN
```