Universidade Federal de Santa Catarina - UFSC
Centro Tecnológico - CTC
Departamento de Informática e Estatística - INE

Lista de Exercícios 1 - INE5202/INE5232/INE5409
Prof. Sérgio Peters (`sergio.peters@ufsc.br`)

Resoluções [1]

1) (a) $(10.1011)_2 = 1 \cdot 2^1 + 1 \cdot 2^{-1} + 1 \cdot 2^{-3} + 1 \cdot 2^{-4} = \mathbf{(2.6875)_{10}}$

(b) $(10.57)_{10} = (?)_2$

$$(10)_{10} = (?)_2 \longrightarrow \begin{array}{ccccc} 10 & \underline{|\,2} & & & \\ 0 & 5 & \underline{|\,2} & & \\ & 1 & 2 & \underline{|\,2} & \\ & & 0 & 1 & \underline{|\,2} \\ & & & 1 & 0 \end{array}$$

$$\longleftarrow$$

$$(10)_{10} = (1010)_2 = 1 \cdot 2^3 + 1 \cdot 2^1$$

$$(0.57)_{10} = (?)_2 \longrightarrow$$

$$\begin{array}{cccccccccccccccc} 0.57 & 0.14 & 0.28 & 0.56 & 0.12 & 0.24 & 0.48 & 0.96 & 0.92 & 0.84 & 0.68 & 0.36 & 0.72 & 0.44 & \\ \times 2 & \times 2 & \times 2 & \times 2 & \times 2 & \times 2 & \times 2 & \times 2 & \times 2 & \times 2 & \times 2 & \times 2 & \times 2 & \times 2 & \ldots \\ 1.14 & 0.28 & 0.56 & 1.12 & 0.24 & 0.48 & 0.96 & 1.92 & 1.84 & 1.68 & 1.36 & 0.72 & 1.44 & 0.88 & \end{array}$$

$$(0.57)_{10} = (0.10010001111010\ldots)_2$$

Logo, $(10.57)_{10} = \mathbf{(1010.10010001111010\ldots)_2}$.

2)
```python
from functools import reduce

for n in [2, 3, 10, 16]:
    values = reduce(lambda x, _: x + [(n + 1) * x[-1] - 1], range(10), [1/n])
    print("\n".join("{:.30f}".format(i) for i in values))
```

- A função *reduce* é utilizada para criar o padrão requerido pelo enunciado: uma lista de números onde todos os valores estão diretamente relacionados a seus antecessores.
- Aplica-se a fórmula do enunciado no último termo da lista a cada iteração, por 10 vezes, iniciando pela inversa multiplicativa do número fornecido.
- Adiciona-se então mais casas após à vírgula para melhor visualização de como o arredondamento funciona.
- O objetivo do algoritmo é verificar como os valores são influenciados gradativamente por computações anteriores: estes perdem precisão exponencialmente pois um valor decimal não pode ser representado perfeitamente com um número finito de bits, exceto por potências de 2. A saída do algoritmo mostra duas possibilidades para cada caso, e como a quantidade de bits divergentes à divisão inicial, que já não se mostra exata, aumenta ao longo das iterações.
- Exemplo de saída (parcial) do programa:

```
0.333333333333333314829616256247
0.333333333333333259318465024990
0.333333333333333037273860099958
0.333333333333332149095440399833
0.333333333333328596381761599332
0.333333333333314385527046397328
0.333333333333257542108185589314
0.333333333333030168432742357254
0.333333333332120673730969429016
0.333333333328482694923877716064
0.333333333313930779695510864258
```

3) (a) $(3021)_{F!} = 3 \cdot 4! + 0 \cdot 3! + 2 \cdot 2! + 1 \cdot 1! = \mathbf{(77)_{10}}$

(b) $(4321)_{F!} = 4 \cdot 4! + 3 \cdot 3! + 2 \cdot 2! + 1 \cdot 1! = \mathbf{(119)_{10}}$

(c) $(10000)_{F!} = 1 \cdot 5! = \mathbf{(120)_{10}}$

[1] por Gustavo Zambonin (`gustavo.zambonin@grad.ufsc.br`)

(d) $(0.02)_{F!} = \frac{0}{2!} + \frac{2}{3!} = (\frac{2}{6})_{10} = (\frac{1}{3})_{10} = \mathbf{(0.\overline{3})_{10}}$

(e) $(0.113)_{F!} = \frac{1}{2!} + \frac{1}{3!} + \frac{3}{4!} = (\frac{19}{24})_{10} = \mathbf{(0.791\overline{6})_{10}}$

(f) $(321.123)_{F!} = 3 \cdot 3! + 2 \cdot 2! + 1 \cdot 1! + \frac{1}{2!} + \frac{2}{3!} + \frac{3}{4!} = (\frac{575}{24})_{10} = \mathbf{(23.958\overline{3})_{10}}$

4) (a) $(10111.1101)_2 = (0001\ 0111.1101)_2 = (17.D)_{16} = 1 \cdot 16^1 + 7 \cdot 16^0 + 13 \cdot 16^{-1} = (\frac{381}{16})_{10} = \mathbf{(23.8125)_{10}}$

(b) $(BD.0E)_{16} = (1011\ 1101.0000\ 1110)_2 = 1 \cdot 2^7 + 1 \cdot 2^5 + 1 \cdot 2^4 + 1 \cdot 2^3 + 1 \cdot 2^2 + 1 \cdot 2^0 + 1 \cdot 2^{-5} + 1 \cdot 2^{-6} + 1 \cdot 2^{-7} = (\frac{24199}{128})_{10} = \mathbf{(189.0546875)_{10}}$

(c) $(41.1)_{10} = (?)_2 = (?)_{16}$

$$(41)_{10} = (?)_2 \longrightarrow \begin{array}{lllllll}
41 & \underline{|\ 2} & & & & & \\
1 & 20 & \underline{|\ 2} & & & & \\
 & 0 & 10 & \underline{|\ 2} & & & \\
 & & 0 & 5 & \underline{|\ 2} & & \\
 & & & 1 & 2 & \underline{|\ 2} & \\
 & & & & 0 & 1 & \underline{|\ 2} \\
 & & & & & 1 & 0
\end{array}$$

$$\longleftarrow$$

$$(41)_{10} = (101001)_2 = 1 \cdot 2^5 + 1 \cdot 2^3 + 1 \cdot 2^0$$

$$(0.1)_{10} = (?)_2 \longrightarrow$$

$$\begin{array}{cccccccccc}
0.1 & 0.2 & 0.4 & 0.8 & 0.6 & 0.2 & 0.4 & 0.8 & 0.6 & 0.2 \\
\times 2 & \times 2 & \times 2 & \times 2 & \times 2 & \times 2 & \times 2 & \times 2 & \times 2 & \times 2 & \ldots \\
0.2 & 0.4 & 0.8 & 1.6 & 1.2 & 0.4 & 0.8 & 1.6 & 1.2 & 0.4
\end{array}$$

$$(0.1)_{10} = (0.00011001100110011\ldots)_2$$

Logo, $(41.1)_{10} = (101001.0\overline{0011})_2 = (0010\ 1001.0001\ \overline{1001})_2 = \mathbf{(29.1\overline{9})_{16}}$. Haverá perda de dígitos significativos.

5) $F(\beta, t, I, S) \longrightarrow F(2, 3, -3, +3) \longrightarrow$ $\begin{array}{|c|c|c|c|c|c|c|}\hline s_1 & d_1 & d_2 & d_3 & s_2 & e_1 & e_2 \\ \hline\end{array}$

(a) $NM = (\beta - 1) \cdot \beta^{t-1} = (2 - 1) \cdot 2^{3-1} = \mathbf{4}$

(b) $NE = S - I + 1 = 3 - (-3) + 1 = \mathbf{7}$

(c) $NR = 2 \cdot NM \cdot NE + 1 = \mathbf{57}$

(d) $\begin{array}{|c|c|c|c|c|c|c|}\hline 0 & 1 & 0 & 0 & 1 & 1 & 1 \\ \hline\end{array} \longrightarrow m.p. = (0.1)_2 \cdot (2^{-3})_{10} = (2^{-1})_{10} \cdot (2^{-3})_{10} = (0.0625)_{10}$

Logo, a região de *underflow* é $\mathbf{\{x \in \mathbb{R} \mid -(0.0625)_{10} < x < (0.0625)_{10}\}}$.

$\begin{array}{|c|c|c|c|c|c|c|}\hline 0 & 1 & 1 & 1 & 0 & 1 & 1 \\ \hline\end{array} \longrightarrow M.P. = (0.111)_2 \cdot (2^3)_{10} = (0.875)_{10} \cdot (2^3)_{10} = (7)_{10}$

Logo, a região de *overflow* é $\mathbf{\{x \in \mathbb{R} \mid x < -(7)_{10} \cup x > (7)_{10}\}}$.

(e) A precisão decimal equivalente é igual a $2^{1-t} = 2^{1-3} = 2^{-2} = \mathbf{2.5 \cdot 10^{-1}}$.

(f) $F(2, 3, 0, 7)$ padrão IEEE 754 (1985) com $d_1 \neq 0 = 1$ antes da vírgula e representado implicitamente antes da vírgula e mais 3 bits significativos após a vírgula e polarização $p = +3$, conforme segue:

$$\begin{cases}
0 < e < 7 & \longrightarrow v = (-1)^s \cdot 2^{e-3} \cdot (1.f)_2 \\
e = 0, f \neq 0 & \longrightarrow (-1)^s \cdot 2^{-2} \cdot (0.f)_2 \\
e = 0, f = 0 & \longrightarrow 0 \\
e = 7 & \longrightarrow v \text{ pertence à região de } \textit{overflow}
\end{cases}$$

6) $F(\beta, t, I, S) \longrightarrow F(2, 3, 0, +7) \longrightarrow$ $\begin{array}{|c|c|c|c|c|c|c|c|}\hline s_1 & d_2 & d_3 & d_4 & 0 & e_1 & e_2 & e_3 \\ \hline\end{array}$

(a) $NM = (\beta - 1) \cdot \beta^{t-1} = (2 - 1) \cdot 2^{4-1} = \mathbf{8}$

(b) $NE = S - I + 1 = 7 - 0 + 1 = \mathbf{8}$

(c) $NR = 2 \cdot NM \cdot NE + 1 = \mathbf{129}$

(d) $\begin{array}{|c|c|c|c|c|c|c|c|}\hline 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 \\ \hline\end{array} \longrightarrow m.p. = (0.001)_2 \cdot (2^{-2})_{10} = (2^{-5})_{10} \cdot (1)_{10} = \mathbf{(0.03125)_{10}}$

Logo, a região de *underflow* é $\mathbf{\{x \in \mathbb{R} \mid -(0.5)_{10} < x < (0.5)_{10}\}}$.

$\begin{array}{|c|c|c|c|c|c|c|c|}\hline 0 & 1 & 1 & 1 & 0 & 1 & 1 & 1 \\ \hline\end{array} \longrightarrow M.P. = (1.111)_2 \cdot (2^{6-3})_{10} = (1.875)_{10} \cdot (2^3)_{10} = \mathbf{(15)_{10}}$

Logo, a região de *overflow* é $\mathbf{\{x \in \mathbb{R} \mid x < -(15)_{10} \cup x > (15)_{10}\}}$.

(e) A precisão decimal equivalente é igual a $2^{1-t} = 2^{1-4} = 2^{-3} = \mathbf{1.25 \cdot 10^{-1}}$.

(f) $(-1)^s \cdot 2^{-2} \cdot 0 = \mathbf{0}$

7) O formato de ponto flutuante de precisão dupla tem 1 bit de sinal, 11 bits para expoente e 52 bits explícitos para decimais (o 1º bit é implícito por conta do armazenamento do número de acordo com o padrão IEEE 754).

Também foi decidido neste padrão facilitar o armazenamento do expoente. Assim, um conceito chamado *exponent bias*, calculado por $2^{k-1} - 1$ onde k é o número de bits do expoente, foi criado com esta função. Este *bias* é essencial para a codificação do expoente como um valor `unsigned`. Para obter o expoente verdadeiro, subtrai-se o *bias* do expoente representado.

Assim sendo, $bias = 2^{11-1} - 1 = 1023$. $e_{min} = 1 - 1023 = -1022$ e $e_{max} = 2046 - 1023 = 1023$. (Os expoentes 0 e 2047 são reservados.)

Portanto, o menor número passível de ser representado com precisão dupla é $2^{-1022} \approx \mathbf{2.225 \cdot 10^{-308}}$, e o maior número é 2^{1023} com todos os bits da mantissa ativados, ou seja, $2^{1023} \cdot (2 - 2^{-52}) \approx \mathbf{1.797 \cdot 10^{308}}$. Às custas da perda de precisão, existem números que vão abaixo da fronteira de *underflow*, chamados de subnormais.

```
>>> 2**(-1022 - 52)
5e-324
>>> 2**(-1022)
2.2250738585072014e-308
>>> (2 - 2**(-52)) * (2**1023)
1.7976931348623157e+308
>>> float(2**1024)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
OverflowError: int too large to convert to float
```

8) (a) $2^0 + 2^{-52} \longrightarrow$ precisão de até **17** dígitos decimais.

(b) $2^{-52} = 10^{1-d} \longrightarrow 1 - d = -52 \cdot log(2) \longrightarrow d = 1 + 52 \cdot log(2) \approx \mathbf{16.65356}$.

(c) Imprimindo `double x = 0.1` com 30 casas após à vírgula, observa-se que existem 17 decimais exatos, portanto a precisão decimal equivalente está entre **16 e 17 dígitos**.

9)
```python
from numpy import float32 as single

h = 1/2
x = 2/3 - h
y = 3/5 - h
e = 3*x - h
f = 5*y - h
g = e/f

for i in [h, x, y, e, f, g]:
    print("{:.55f}\n{:.55f}\n".format(single(i), i))
```

- O processo descrito em código pode ser reescrito aritmeticamente da seguinte maneira:

$$g = \frac{e}{f} = \frac{3 \cdot x - h}{5 \cdot y - h} = \frac{3 \cdot (\frac{2}{3} - h) - h}{5 \cdot (\frac{3}{5} - h) - h} = \frac{3 \cdot (\frac{2}{3} - \frac{1}{2}) - \frac{1}{2}}{5 \cdot (\frac{3}{5} - \frac{1}{2}) - \frac{1}{2}} = \frac{3 \cdot \frac{1}{6} - \frac{1}{2}}{5 \cdot \frac{1}{10} - \frac{1}{2}} = \frac{\frac{1}{2} - \frac{1}{2}}{\frac{1}{2} - \frac{1}{2}} = \frac{0}{0}$$

  Porém, nota-se que na saída do programa, as variáveis `x` e `y` são responsáveis pelo resultado válido (`g` $= 1$), em virtude da imprecisão de seu armazenamento quando $\beta = 2$ (as multiplicações realizadas nunca serão iguais a um valor exato binariamente como $\frac{1}{2}$). Assim, como `e` $=$ `f` e $e, f \neq 0$, então $\frac{e}{f} = g = 1$.

- Saída do programa:

```
0.5000000000000000000000000000000000000000000000000000000
0.5000000000000000000000000000000000000000000000000000000

0.1666666716337203979492187500000000000000000000000000000
0.1666666666666666296592325124947819858789443969726562500

0.1000000014901161193847656250000000000000000000000000000
0.0999999999999999777955395074968691915273666381835937500

-0.0000000000000001110223024625156540423631668090820312500
-0.0000000000000001110223024625156540423631668090820312500

-0.0000000000000001110223024625156540423631668090820312500
-0.0000000000000001110223024625156540423631668090820312500
```

```
1.0000000000000000000000000000000000000000000000000000
1.0000000000000000000000000000000000000000000000000000
```

10)

```python
# Numerical Analysis (Burden & Faires, 9th ed., p. 25)
# Demonstration of how subtracting floating point numbers affects greatly the
# final result. The methods purposely round numbers to show errors that may
# appear, and thus, shall not be used for real solving purposes.


def quad_eq(a, b, c):
    """Quadratic formula simple implementation."""
    d = round((b**2 - 4*a*c)**(1.0/2), 2)
    return (- b / (2*a)) if d == 0 else ((- b + d) / (2*a), (- b - d) / (2*a))


def rat_quad_eq(a, b, c):
    """Rationalized quadratic formula. It should provide
    further accuracy with certain coefficients."""
    d = round((b**2 - 4*a*c)**(1.0/2), 2)
    return (- b / (2*a)) if d == 0 else ((- 2*c) / (b + d), (- 2*c) / (b - d))


def relative_error(desired, obtained):
    """Presents the relative error according to results
    obtained from floating point manipulation, and a real value."""
    return abs((desired - obtained) / obtained)


origx1, origx2 = -0.0161072, -62.0839

quadx1, quadx2 = quad_eq(1, 62.1, 1)
print(round(quadx1, 4), round(quadx2, 2))
print(relative_error(origx1, round(quadx1, 4)),
      relative_error(origx2, round(quadx2, 2)))

ratx1, ratx2 = rat_quad_eq(1, 62.1, 1)
print(round(ratx1, 4), round(ratx2, 2))
print(relative_error(origx1, round(ratx1, 4)),
      relative_error(origx2, round(ratx2, 2)))
```

- O código acima simula os cálculos com precisão dupla nos dois tipos de equação quadrática, além de imprimir o resultado arredondado para o operador aritmético $F(10, 4, -99, +99)$ e os resultados reais.
- Observa-se que, onde não existem subtrações internas para calcular o resultado final, o erro relativo mostra-se muito inferior, assim gerando resultado mais confiáveis. Utilizando precisão dupla, os erros tornam-se irrelevantes entre si – a diferença entre métodos é da ordem de $1 \cdot 10^{-14}$.
- Saída do programa:

```
-0.015 -62.09
0.07381333333333329 9.824448381387577e-05
-0.0161 -66.67
0.00044720496894402943 0.06878806059697018
```

11) (a)
- Erro absoluto: $|VA - VE| = |1102.345 - 1100.9| = \mathbf{1.445}$
- Erro relativo: $\left|\frac{VA-VE}{VE}\right| = \left|\frac{1.445}{1100.9}\right| \approx \mathbf{0.00131}$
- Percentual: $0.00131 \cdot 100 = \mathbf{0.131\%}$

(b)
- Erro absoluto: $|VA - VE| = |0.01245 - 0.0119| = \mathbf{0.00055}$
- Erro relativo: $\left|\frac{VA-VE}{VE}\right| = \left|\frac{0.00055}{0.0119}\right| \approx \mathbf{0.04622}$
- Percentual: $0.04622 \cdot 100 = \mathbf{4.622\%}$

12) (a) Sabe-se que $(10)_{10} = (1010)_2$. O cálculo da parte decimal segue abaixo: $-(0.05)_{10} = (?)_2$

$$\begin{array}{cccccccccccc} 0.05 & 0.1 & 0.2 & 0.4 & 0.8 & 0.6 & 0.2 & 0.4 & 0.8 & 0.6 & \\ \times 2 & \times 2 & \times 2 & \times 2 & \times 2 & \times 2 & \times 2 & \times 2 & \times 2 & \times 2 & \ldots \\ \color{red}{0}.1 & \color{red}{0}.2 & \color{red}{0}.4 & \color{red}{0}.8 & \color{red}{1}.6 & \color{red}{1}.2 & \color{red}{0}.4 & \color{red}{0}.8 & \color{red}{1}.6 & \color{red}{1}.2 & \end{array}$$

$$-(0.05)_{10} = -(0.0\overline{0011})_2 = 1 \cdot 2^{-5} \cdot -(0.\overline{1001})_2$$

$$(-10.05)_{10} = (1010.0000\overline{1100})_2 = (1.0100000\overline{1100})_2 \cdot 2^3$$

$s = 1$ (sinal negativo)
$e = 3 + 127 = 130$ (polarização ou *exponent bias*)

O arredondamento no bit final da mantissa acontece pois a fração binária termina por 1. A parcela binária arredondada é igual a $2^{-23} \cdot (0.\overline{1100})_2$

Resultado final: | **1** | **10000010** | **01000001100110011001101** |

(b) Convertendo novamente o número para decimal, tem-se $VA = (-10.050000190734863)_{10}$.
O percentual de erro é igual a $\left|\frac{VA-VE}{VE}\right| \cdot 100 \approx \mathbf{1.897856 \cdot 10^{-6}\%}$.

13) | 1 | 00000000 | 10000000000000000000001 | (Número subnormal)

$$e = 0, f \neq 0 \longrightarrow v = (-1)^s \cdot \left(\sum_{i=1}^{23} b_{23-i}2^{-i}\right) \cdot 2^{e-126} = (-1)^1 \cdot (2^{-1} + 2^{-23}) \cdot 2^{-126} \approx \mathbf{-5.877473 \cdot 10^{-39}}$$

14) (a) Para descobrir a ordem desse número, é necessário estimar a partir do cálculo de expoente. Ignorando o sinal e a mantissa, sabe-se que $(1.51 \cdot 10^{37})_{10} \approx 2^{e-127}$ (aparentemente, $0 < e < 255$).
Segue que $log_2(1.51 \cdot 10^{37}) = log_2(2^{e-127})$. Logo, $123.5058 \approx e - 127$ e, como o número de bits precisa ser arredondado para baixo, então $123 = e - 127$ e por fim $e = 250$. O número é negativo, então $s = 1$.
Tem-se então, $(-1.51 \cdot 10^{+37})_{10} = (-1)^1 \cdot 2^{250-127} \cdot (1.f)_2 \longrightarrow \frac{1.51 \cdot 10^{37}}{2^{123}} = (1.f)_2$. Multiplicando sucessivamente o decimal obtido, conclui-se que $(-1.51 \cdot 10^{+37})_{10} = (-1.0110101110000100111011111010\ldots)_2 \cdot 2^{123}$.
$f = (01101011100001001110111)_2 + (1)_2 = (01101011100001001111000)_2$ (primeiros 23 bits + arredondamento)

Resultado final: | **1** | **11111010** | **01101011100001001111000** |

(b) Convertendo novamente o número para decimal, tem-se $VA = (-1.5100000419798622984748429241797836 8 \cdot 10^{37})_{10}$.
O percentual de erro é igual a $\left|\frac{VA-VE}{VE}\right| \cdot 100 \approx \mathbf{2.7801 \cdot 10^{-6}\%}$.

15) (a) $(-1.1 \cdot 10^{-41})_{10} = (-1)^1 \cdot 2^{-126} \cdot (0.f)_2$ com $e \leq 0$, pois $log_2(1.1 \cdot 10^{-41}) \approx -136$, o que implica em $e = -136 + 127 = -9 < 0 \longrightarrow$ número subnormal)
Então, para descobrir a mantissa f, calcula-se $\frac{1.1 \cdot 10^{-41}}{2^{-126}}$ e multiplica-se o decimal para encontrar seu valor binário. Por fim, $(-1.1 \cdot 10^{-41})_{10} = (-1.11101010100111011100110000 0\ldots)_2 \cdot 2^{-137}$.
É necessário mover o decimal para que o número possa ser representado com $e = 0$. Então, $(-1.111010101001110111001100000\ldots)_2 \cdot 2^{-137} = (-0.00000000001111010101001\ldots)_2 \cdot 2^{-126}$.

Resultado final: | **1** | **00000000** | **00000000001111010101010** |

(b) Convertendo novamente o número para decimal, tem-se $VA = -1.1000192944949814 \cdot 10^{-41}$
O percentual de erro é igual a $\left|\frac{VA-VE}{VE}\right| \cdot 100 = \mathbf{1.754044998\overline{309} \cdot 10^{-3}\%}$.

16) Caso seja viável calcular o resultado desejado à mão, ou utilizando ferramentas de maior precisão, é possível verificar o erro relativo sem maiores dificuldades. Porém, na maioria dos casos isso não acontece, e é necessário observar como o computador se comporta nestes casos. Esta aritmética de ponto flutuante é uma grande área de estudo do cálculo numérico.

17, 18)

```python
from functools import reduce
from math import factorial as fact
from numpy import float32 as single

n = 4
x = -.111
desired = 0.894938748929031

for y in [single(x), x]:
    obtained = reduce(lambda w, i: w + (y**i)/fact(i), range(n), 0)
    print("{:.65f}%".format(abs((desired - obtained) / desired) * 100))
```

- O decimal desejado foi retirado do Wolfram Alpha, para comparação com os resultados calculados pelo programa em variáveis de precisão simples e dupla, respectivamente. O erro da variável de precisão simples é maior pela quantidade reduzida de bits disponíveis para armazenar a parte decimal do resultado.
- Novamente, a função *reduce* é utilizada para simular o cálculo da série de Maclaurin com os argumentos fornecidos.
- Saída do programa:

```
0.00069152325516942810771509053680006218201015144586563110351562500%
0.00069138016857893662848316695956896182906348258256912231445312500%
```