Julia Plewa
MOwNiT
Laboratorium 1

1. **Porównać w języku Julia/C++ reprezentację bitową liczby 1/3 dla Float32, Float64 oraz liczby, która jest inicjalizowana jako Float32, a potem rzutowana na Float64. W przypadku C++ należy odpowiednio zmodyfikować nazwy typów zmiennych oraz wykonać stosowne rzutowania.**

Kod:

```
decode(x::Float32) = (b=bitstring(x); (b[1], b[2:9], b[10:32]))
decode(x::Float64) = (b=bitstring(x); (b[1], b[2:12], b[13:64]))

function ex1()
   a = Float32(1/3)
   b = Float64(1/3)
   c = Float64(a)

   println("\tliczba dziesiętnie\tznak\tcecha\t\tmantysa")
   println("a\t", a, "\t\t", decode(a)[1],"\t",decode(a)[2],"\t",decode(a)[3])
   println("b\t", b, "\t", decode(b)[1],"\t",decode(b)[2],"\t",decode(b)[3])
   println("c\t", c, "\t", decode(c)[1],"\t",decode(c)[2],"\t",decode(c)[3])
end
```

Wyniki:

```
        liczba dziesiętnie      znak    cecha           mantysa
a       0.33333334              0       01111101        01010101010101010101011
b       0.3333333333333333      0       01111111101     0101010101010101010101010101010101010101010101010101
c       0.3333333432674408      0       01111111101     0101010101010101010101100000000000000000000000000000
```

Wnioski:

Liczby znane według standardu *IEEE 754-2008* jako *binary32* mają 1 bit znaku, 8 bitów cechy oraz 23 bity matysy (24 wliczając ukrytą jedynkę):

$$binary32(x) = (-1)^s \cdot \left(1 + \sum_{i=1}^{23} b_{23-i}\, 2^{-i}\right) \cdot 2^{e-127}$$

Liczby znane według standardu *IEEE 754-2008* jako *binary64* mają 1 bit znaku, 11 bitów cechy oraz 52 bity matysy (53 wliczając ukrytą jedynkę).

$$binary64(x) = (-1)^s \cdot \left(1 + \sum_{i=1}^{52} b_{52-i}\, 2^{-i}\right) \cdot 2^{e-1023}$$

W powyższym przykładzie zmienne a i b pokazują odpowiednio 32- i 64-bitową reprezentację wartości ⅓. Przykład zmiennej c pokazuje, że zrzutowanie 32-bitowej reprezentacji na 64-bitową nie prowadzi do zwiększenia jej dokładności.

2. **Napisać program w języku C++, który oblicza kolejne wyrazy dowolnego nietrywialnego ciągu. Wykonać to zadanie dla reprezentacji float oraz double. Wykonać program na różnych maszynach. Objaśnić wyniki oraz fakt, że są różne.**

Ciąg:

$$x_0 = 15$$
$$x_1 = 30$$
$$x_n = \tfrac{6}{7} \cdot x_{n-1} - \tfrac{1}{7} \cdot x_{n-2}\,, \quad n > 1$$

Kod:

```
void ex2() {
   std::ofstream file_a;   // open result files and set precision
   std::ofstream file_b;
   file_a.precision(6);
   file_b.precision(15);
   file_a.setf(std::ios::scientific);
   file_b.setf(std::ios::scientific);
   file_a.open("ex2_a.txt");
   file_b.open("ex2_b.txt");
   float a;                // initial sequence values
```

```
    float prev_a = 30;
    float prev_prev_a = 15;
    double b;
    double prev_b = 30;
    double prev_prev_b = 15;

    // save first elements to their respective files
    print_to_files(prev_prev_a, prev_prev_b, &file_a, &file_b);
    print_to_files(prev_a, prev_b, &file_a, &file_b);

    // loop through sequence elements and save values into files
    for (int i = 0; i < EX2_LEN; i++) {
        a = prev_a * (float) 6 / (float) 7 - prev_prev_a * (float) 1 / (float) 7;
        b = prev_b * (double) 6 / (double) 7 - prev_prev_b * (double) 1 / (double) 7;
        prev_prev_a = prev_a;
        prev_prev_b = prev_b;
        prev_a = a;
        prev_b = b;
        print_to_files(a, b, &file_a, &file_b);
    }
    file_a.close();     // close result files
    file_b.close();
}
```

Wyniki:

n	a_n	b_n
0	1.500000e+01	1.500000000000000e+01
1	3.000000e+01	3.000000000000000e+0
2	2.357143e+01	2.357142857142857e+01
3	1.591837e+01	1.591836734693878e+01
. . .	. . .	. . .
16	4.116711e-02	4.116709902876997e-02
. . .	. . .	. . .
231	2.802597e-45	3.647035175886253e-45

232	1.401298e-45	2.299827447978406e-45
233	1.401298e-45	1.450275644569169e-45
234	1.401298e-45	9.145466313480870e-46
. . .	. . .	. . .
1620	1.401298e-45	1.976262583364986e-323
1621	1.401298e-45	9.881312916824931e-324
1622	1.401298e-45	4.940656458412465e-324
1623	1.401298e-45	4.940656458412465e-324
. . .	. . .	. . .

Wnioski:

Elementy wybranego przeze mnie ciągu maleją wraz z każdą iteracją. Od $n = 232$ ciąg a_n elementów typu *float* przyjmuje na stałe wartość 1.401298×10^{-45}. Jest to najmniejsza dodatnia zdenormalizowana 32-bitowa wartość zmiennoprzecinkowa (binarnie: $0.00000000000000000000001 \times 2^{-126}$). Czynnik $\frac{6}{7}$, przez który mnożona jest ta wartość w dalszych iteracjach, jest zbyt duży, aby wartość została zaokrąglona w dół do 0. Analogicznie ciąg 64-bitowych liczb b_n od $n = 1622$ na stałe przyjmuje wartość $4.940656458412465 \times 10^{-324}$ będącą najmniejszą dodatnią zdenormalizowaną 64-bitową liczbą zmiennoprzecinkową. Różnice pomiędzy poszczególnymi elementami a_n i b_n wynikają z zaokrąglania liczb na kolejnych etapach - utracone cyfry znaczące propagują na dalsze etapy obliczeń.

3. **Jedną z bibliotek numerycznych, jaką będziemy używać na zajęciach jest GSL (język C/C++). Korzystając ze wsparcia dla wyświetlania reprezentacji liczb zmiennoprzecinkowych zobaczyć jak zmienia się cecha i mantysa dla coraz mniejszych liczb. Zaobserwować, kiedy matysa przestaje być znormalizowana i dlaczego?**

Kod:

```c
void ex3() {
   // open result files
   FILE *file_a = fopen("ex3_a.txt", "w+");
   FILE *file_b = fopen("ex3_b.txt", "w+");

   float a = 1.1;       // set initial sequence values
   double b = 1.1;

   // loop through the sequence and record the elemts
   for (int i = 0; i < EX3_LEN; i++) {
       gsl_ieee_fprintf_float(file_a, &a);
       gsl_ieee_fprintf_double(file_b, &b);
       fprintf(file_a, "\n");
       fprintf(file_b, "\n");
       a /= ((float) 2);
       b /= ((double) 2);
   }
   fclose(file_a);     // close result files
   fclose(file_b);
}
```

Wyniki:

#	*float a*	*double b*
1	1.00011001100110011001101*2^0	1.0001100110011001100110011001100110011001100110011010*2^0
2	1.00011001100110011001101*2^-1	1.0001100110011001100110011001100110011001100110011010*2^-1
3	1.00011001100110011001101*2^-2	1.0001100110011001100110011001100110011001100110011010*2^-2
4	1.00011001100110011001101*2^-3	1.0001100110011001100110011001100110011001100110011010*2^-3
5	1.00011001100110011001101*2^-4	1.0001100110011001100110011001100110011001100110011010*2^-4
6	1.00011001100110011001101*2^-5	1.0001100110011001100110011001100110011001100110011010*2^-5

. . .	. . .	. . .
127	1.00011001100110011001101*2^-126	1.0001100110011001100110011001100110011001100110011010*2^-126
128	0.10001100110011001100110*2^-126	1.0001100110011001100110011001100110011001100110011010*2^-127
129	0.01000110011001100110011*2^-126	1.0001100110011001100110011001100110011001100110011010*2^-128
130	0.00100011001100110011010*2^-126	1.0001100110011001100110011001100110011001100110011010*2^-129
131	0.00010001100110011001101*2^-126	1.0001100110011001100110011001100110011001100110011010*2^-130
. . .	. . .	. . .
149	0.00000000000000000000010*2^-126	1.0001100110011001100110011001100110011001100110011010*2^-148
150	0.00000000000000000000001*2^-126	1.0001100110011001100110011001100110011001100110011010*2^-149
151	0	1.0001100110011001100110011001100110011001100110011010*2^-150
152	0	1.0001100110011001100110011001100110011001100110011010*2^-151
. . .	. . .	. . .
1023	0	1.0001100110011001100110011001100110011001100110011010*2^-1022
1024	0	0.1000110011001100110011001100110011001100110011001101*2^-1022
1025	0	0.0100011001100110011001100110011001100110011001100110*2^-1022
1026	0	0.0010001100110011001100110011001100110011001100110011*2^-1022
. . .	. . .	. . .
1074		0.0000000000000000000000000000000000000000000000000010*2^-1022
1075	0	0.0000000000000000000000000000000000000000000000000001*2^-1022
1076	0	0
1077	0	0
. . .	. . .	. . .

Wnioski:

W obydwu przypadkach liczby zachowują znormalizowaną formę, dopóki szerokość cechy na to pozwala. Dla 32-bitowej zmiennej typu *float* minimalna ujemna wartość wykładnika to -126 (na wykładnik poświęcone jest 8 bitów i jest przesunięty o -127), a dla 64-bitowej zmiennej typu *double* minimalna ujemna wartość wykładnika to -1022 (11 bitów i przesunięcie o -1023). Dalsze dzielenie liczby przez 2 prowadzi do zdenormalizowania jej reprezentacji - dla odróżnienia od postaci znormalizowanej wykładnik przyjmuje wtedy specjalną wartość -127 dla *floata* oraz -1023 dla *double'a*, a reprezentowana liczba ma następującą postać:

$$\begin{aligned} binary32(x) &= (-1)^s \cdot \left(\sum_{i=1}^{23} b_{23-i}\, 2^{-i} \right) \cdot 2^{-126} \\ binary64(x) &= (-1)^s \cdot \left(\sum_{i=1}^{52} b_{52-i}\, 2^{-i} \right) \cdot 2^{-1022} \end{aligned}$$

W powyższym przykładzie zdenormalizowane liczby zostały zaznaczone kolorem czerwonym. Gdy szerokość mantysy nie pozwala już na dalsze zmniejszanie wartości, a liczba nadal jest dzielona przez 2, liczba przyjmuje wartość 0.

4. Wymyślić własny przykład algorytmu niestablinego numerycznie.

Algorytm:

$$f(x) = \sqrt{x^2 + 1} - 1$$

$$g(x) = \frac{x^2}{\sqrt{x^2+1}+1}$$

Kod:

```
void ex4() {
   std::cout.precision(15);    // set output precision
   double values[] = {2.718281828459, 0.000000000000000001, 0.00000001826, 271828};
   // loop through the sample x values and calculate f(x) and g(x) separately
   // printing out the results
   for (int i = 0; i < 4; i++) {
       double x = values[i];
       double fx = sqrt(x * x + 1) - 1;
       double gx = (x * x) / (sqrt(x * x + 1) + 1);

       std::cout << "x = " << x << std::endl << "\tf(x) = " << fx << "\t";
       gsl_ieee_fprintf_double(stdout, &fx);
       std::cout << std::endl << "\tg(x) = " << gx << "\t";
       gsl_ieee_fprintf_double(stdout, &gx);
       std::cout << std::endl << std::endl;
   }
}
```

a. Zademostrować wersję niestabilną, pokazać, że działa źle.

Kod:

```
double fx = sqrt(x * x + 1) - 1
```

Wyniki:

x	**f(x)**	**f(x) wg WolframAlpha**	**błąd**
0.00000000000001	0	5×10^{-29}	100%
0.00000001826	$2.220446049250313 \times 10^{-16}$	$1.667138000000000 \times 10^{-16}$	33.2%
1.0	$4.142135623730951 \times 10^{-1}$	$4.142135623730950 \times 10^{-1}$	0.00%
10.0	9.04987562112089	9.04987562112089	0.00%

b. Wyjaśnić, dlaczego działa źle.

W przypadku wartości x zbliżonych do 0 następuje tzw. *catastrophic cancellation*, czyli nieodwracalna utrata cyfr znaczących będąca rezultatem odejmowania bliskich siebie liczb. Dla $x = 1e-14$ mamy do czynienia z całkowitą utratą cyfr znaczących.

Rozbieżność pomiędzy niestabilną funkcją $f(x)$ oraz jej stabilną wersją $g(x)$, zaprezentowaną w kolejnym podpunkcie, łatwo zaobserwować na przykładzie binarnej reprezentacji wyników dla $x = 1.826e-8$:

$f(x)$	$1.0000000000000000000000000000000000000000000000000000 \times 2^{-52}$
$g(x)$	$1.1000000001101010011101010010100111001011010100011111 \times 2^{-53}$

Błędy wynikające z nieodwracalnej utraty cyfr znaczących będą propagować na dalsze etapy obliczeń i mogą poważnie zniekształcać wyniki.

c. Zademonstrować wersję stabilną.

Kod:

```
double gx = (x * x) / (sqrt(x * x + 1) + 1);
```

Wyniki:

x	g(x)	g(x) wg WolframAlpha	błąd
0.00000000000001	5×10^{-29}	5×10^{-29}	0.00%
0.00000001826	$1.667138000000000 \times 10^{-16}$	$1.667138000000000 \times 10^{-16}$	0.00%
1.0	$4.142135623730951 \times 10^{-1}$	$4.142135623730950 \times 10^{-1}$	0.00%
10.0	9.04987562112089	9.04987562112089	0.00%