Kiedy i dlaczego komputery kłamią — czyli paradoksy arytmetyki komputerowej

Stefan Sokołowski*

Gdańsk, 2 marca 2022

Spis treści

*Instytut Informatyki Stosowanej im. K. Brzeskiego, Akademia Nauk Stosowanych w Elblągu.

1 Dziwne wyniki

Matematyka jest abstrakcyjna i idealna. Technika jest praktyczna i zawsze niedoskonała. Kiedy powierzamy obliczenie matematyczne urządzeniu technicznemu, musimy liczyć się z tym, że wykona je po swojemu, nie zawsze zgodnie z ideałem. To jest prawda nawet dla tak prostych operacji jak dodawanie na dziecięcym liczydle: specjaliści potrafią je użyć do rachunków na liczbach 10-cyfrowych[1], ale wyżej już ani w ząb. No tak, liczydło jest urządzeniem materialnym, technicznym, więc gdzie mu tam do pełnej nieskończonej matematyki...

Komputer jest również urządzeniem technicznym, więc programy często dają dziwaczne wyniki.

1.1 Przykładowe dziwactwo 1

Na przykład proszę samemu sprawdzić działanie następującego programiku w C (łatwo go przepisać na dowolny inny język):

```
#include<stdio.h>
int main() {
  int n=50000;
  printf("\n  %i * %i == %i\n\n", n, n, n*n);
  return 0;
}
```

U mnie drukuje on surrealistyczne

```
 50000 * 50000 == -1794967296
```

Iloczyn liczb dodatnich jest ujemny?![2] Komputer skłamał!

1.2 Przykładowe dziwactwo 2

Albo zobaczmy, co robi taki programik:

[1] Jeśli Czytelnik nie jest specjalistą od liczydła, to zapewne nie zajdzie dalej niż 2 cyfry — liczba 100 będzie nieprzekraczalnym sufitem.

[2] Tak dzieje się na większości komputerów, jakich używamy. Jeżeli na komputerze Czytelnika ten błąd się nie pojawia, to zapewne stosowane są liczby całkowite więcej niż 32-bitowe. Błąd pojawi się, jeśli w programie użyć jeszcze większej liczby zamiast 50 000. Proponuję zaeksperymentować z liczbą 4 000 000 000 (4 miliardy), bo ona jest za duża nawet dla liczb 64-bitowych — wyjaśnienie dalej, por. podrozdz. 2.7 (str. 12).

```
#include<stdio.h>
int main() {
  if ( 5 / 3.0 == 5 * (1 / 3.0) )  printf("\n  rowne \n\n");
  if ( 5 / 3.0 <  5 * (1 / 3.0) )  printf("\n  mniejsze \n\n");
  if ( 5 / 3.0 >  5 * (1 / 3.0) )  printf("\n  wieksze \n\n");
  return 0;
}
```

Liczba 5 podzielona przez 3 powinna być równa liczbie 5 pomnożonej przez $\frac{1}{3}$. Ale ten program upiera się, że jest większa![3]

1.3 Zagrożenia na każdym kroku...

Czy skutki takich błędów mogą być groźne? Czy może zdarzają się tak rzadko i są tak niewielkie, że nie warto się nimi przejmować? To w dużym stopniu zależy od tego, co chcemy liczyć.

Może wielkości fizyczne?

Na przykład rok świetlny ma około

$$365.25 \cdot 24 \cdot 60 \cdot 60 \cdot 300\,000 \text{ km} \approx 9.5 \cdot 10^{12} \text{ km} = 9.5 \text{ biliona km}$$

Dla tak dużych liczb programy w C na powszechnie stosowanych komputerach haniebnie zawodzą.

Więc może finanse?

Z obliczaniem własnej pensji raczej nie będzie problemu, chyba żeby inflacja spowodowała, że zaczniemy zarabiać i wydawać miliardy. Ale gdybyśmy chcieli policzyć łączne przychody mieszkańców sporego miasta, powiedzmy, ze 100 000 mieszkańcami zarabiającymi każdy po 3010 zł/miesiąc (to jest obecna płaca minimalna), to wychodzi

$$100\,000 \cdot 12 \cdot 3010 \text{ zł/rok} \approx 3.6 \cdot 10^{9} \text{ zł/rok} = 3.6 \text{ miliarda zł/rok}$$

To również jest już za dużo. Jeśli Czytelnik będzie kiedyś oprogramowywał na przykład system podatkowy dla takiego miasta i komputer wykaże przychody ujemne, to szef może być bardzo niezadowolony.

W dalszej części tej opowieści spróbuję wyjaśnić, skąd biorą się komputerowe przekłamania, kiedy należy się ich spodziewać i jak można im zapobiegać. Ale oczywiście to ostatnie nie jest proste — gdyby istniał łatwy sposób uniknięcia błędów w obliczeniach, to komputery od dawna liczyłyby poprawnie.

[3] W tym przykładzie **koniecznie trzeba** pisać 3.0 zamiast prostego 3. Czytelnik na pewno rozumie dlaczego...

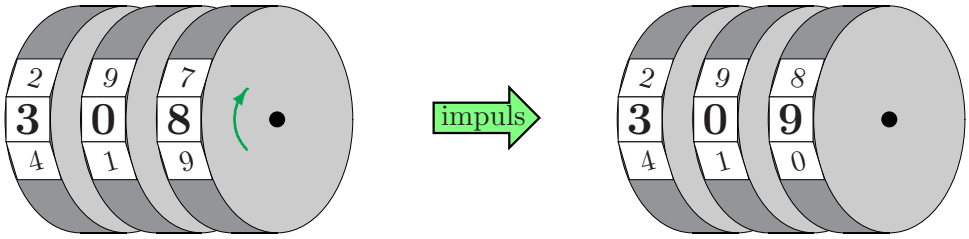
2 Dookoła Wojtek, czyli liczby całkowite

Proszę uważnie prześledzić następny podrozdział o licznikach, on *naprawdę* ma bliski związek z komputerową reprezentacją liczb całkowitych i wyjaśnia pewne paradoksy arytmetyczne.

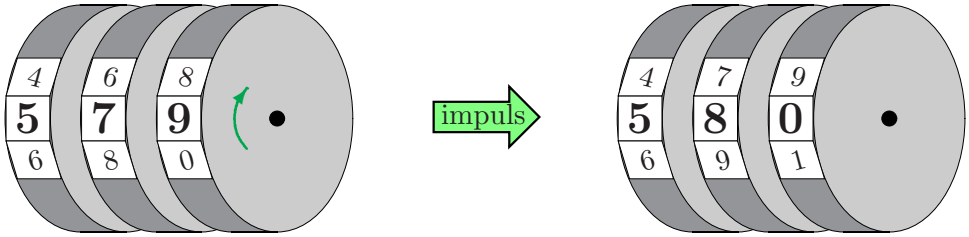
2.1 Liczniki z kółeczkami

Staroświeckie mechaniczne liczniki zliczające impulsy (w arytmometrach, magnetofonach, rowerach itp.) są zwykle zbudowane tak jak na rysunku obok. Mają rząd kółeczek z cyframi; za każdym impulsem, który należy zliczyć, najbardziej prawe kółeczko obraca się o jeden ząbek; a jak któreś kółeczko wykona pełny obrót (o 10 ząbków), to kółeczko po jego lewej stronie obraca się o jeden ząbek.

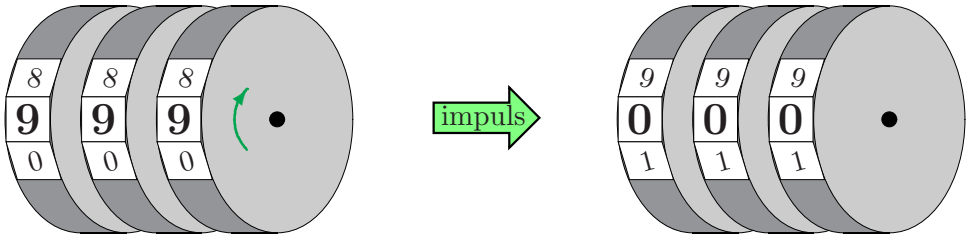
Na przykład jeśli stan licznika w pewnym momencie jest, powiedzmy, **308**, to po przyjęciu impulsu prawe kółko przekręci się o jeden ząbek:



Ale jeśli wyjdziemy od liczby kończącej się na **9**, to impuls spowoduje obrót prawego kółeczka (z **9** na **0**) i tym samym zakończenie pełnego obrotu; więc i środkowe kółko przekręci się o ząbek:



A co spowoduje kolejny impuls dla stanu **999**? Prawe kółko dokona pełnego obrotu; więc środkowe przekręci się o ząbek, ale to spowoduje, że środkowe dokona pełnego obrotu; więc i lewe kółko przekręci się o ząbek:



Licznik „uważa” więc, że liczbą o jeden większą od **999** jest **000**! Taki licznik nie prezentuje nieskończonego ciągu liczb naturalnych, tylko biega w kółko.

I tak samo zachowują się komputerowe liczby całkowite.

2.2 Komputerowe liczby całkowite

Główna różnica między licznikiem mechanicznym z podrozdziału 2.1 (str. 4) a zmiennymi całkowitymi w programie w C polega na tym, że kółeczka w liczniku mogły przybierać po 10 możliwych wartości, a odpowiadające im *bity* zawarte w zmiennej — po dwie wartości: **0** i **1**. Innymi słowami: liczba całkowita nieujemna zapisana *dziesiętnie* (czyli w zwykły sposób) cyframi $c_{n-1}c_{n-2}\ldots c_2c_1c_0$ ma wartość

$$\sum_{i=0}^{n-1} c_i \cdot 10^i \;=\; c_0 \cdot 10^0 + c_1 \cdot 10^1 + \ldots + c_{n-1} \cdot 10^{n-1}$$

przy czym wszystkie cyfry c_i muszą należeć do zbioru $\{0,1,2,3,4,5,6,7,8,9\}$; natomiast liczba całkowita nieujemna zapisana *binarnie* (czyli tak jak w komputerze) bitami $c_{n-1}c_{n-2}\ldots c_2c_1c_0$ ma wartość

$$\sum_{i=0}^{n-1} c_i \cdot 2^i \;=\; c_0 \cdot 2^0 + c_1 \cdot 2^1 + \ldots + c_{n-1} \cdot 2^{n-1} \tag{1}$$

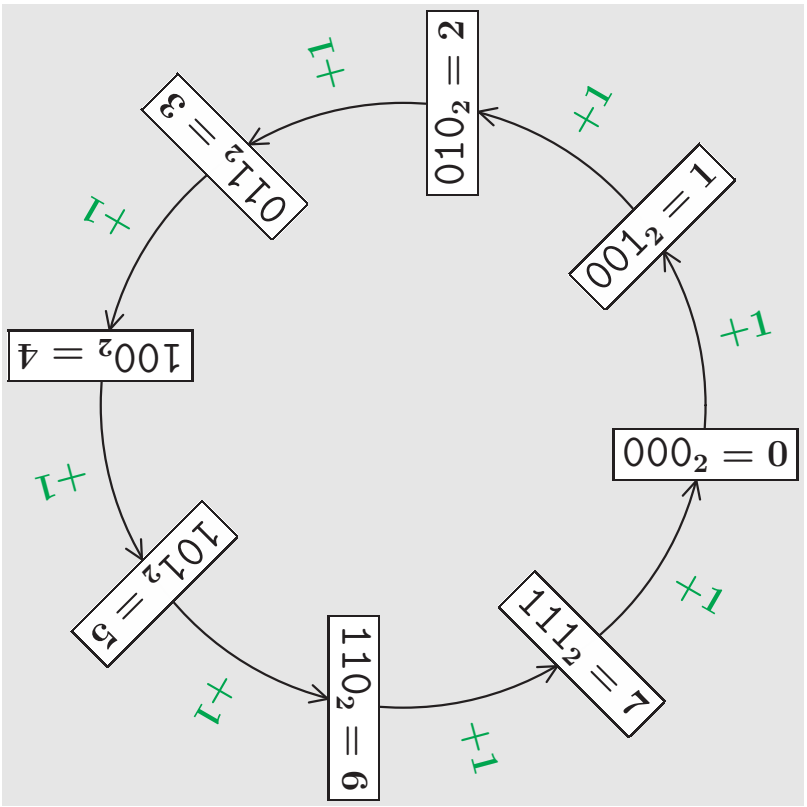
przy czym wszystkie cyfry c_i muszą należeć do zbioru $\{0,1\}$.

Krok w górę polega na zamianie końcowego zera na jedynkę:

Ale jeśli na końcu już jest jedynka, to zostaje zamieniona na zero, a powiększenie przenosi się na bit stojący bezpośrednio po lewej:

To może spowodować zmianę *kilku* bitów po lewej; np. w powyższym przykładzie zmieniły się wszystkie 3 bity. A jeśli cała zmienna jest wypełniona jedynkami, to kolejny krok w górę doprowadza do trzech zer:

Takie rozwiązanie prowadzi do tego, że mamy do czynienia nie z nieskończoną „prostą” liczb całkowitych, tylko ze skończonym „okręgiem” liczb:



Komputerowe zmienne całkowite działają tak, jak na powyższym rysunku; różnica polega tylko na tym, że mogą mieć więcej bitów, czyli miejsc do zajmowania przez zera i jedynki. Dokładniej: jeśli na zmienną przeznacza się n bitów, to może ona przyjmować 2^n różnych wartości. W powyższym prymitywnym przykładzie mamy $n = 3$, więc różnych wartości może być $2^3 = 8$ — od 0 do 7 włącznie. W języku C zmienne całkowite mają standardowo po 32 bity[4], więc różnych liczb tego typu może być $2^{32} = 4\,294\,967\,296$, czyli trochę ponad 4 miliardy.

I jeszcze jedna uwaga: na razie mówimy tylko o liczbach całkowitych nieujemnych, czyli w żargonie języka C — „bezznakowych”, czyli należących do typu `unsigned int`. O ujemnych będzie powiedziane dalej, w podrozdziale 2.5 (str. 8).

[4] 32 bity najczęściej. Ale poszczególne implementacje C mogą od tego odbiegać.

2.3 Przeliczyć bity na decymale i z powrotem

A teraz wystraszę wszystkich, którzy nie uważali na lekcjach o logarytmach! Mianowicie ile bitów wchodzi na jedną cyfrę dziesiętną?

Przy pomocy n cyfr dziesiętnych można wyrazić 10^n różnych liczb. Przy pomocy k bitów można wyrazić 2^k różnych liczb. Żeby te wartości były równe, czyli żeby $2^k = 10^n$, musi zachodzić

$$k \;=\; \log_2(10^n) \;=\; n \cdot \log_2(10) \;\approx\; n \cdot 3.322$$

Tak więc na każdą cyfrę dziesiętną przypada $\frac{k}{n} \approx 3.322$ bitów. Czyli zapis bitowy będzie mniej więcej 3.3-krotnie dłuższy niż dziesiętny.

2.4 Co te komputery właściwie rachują?

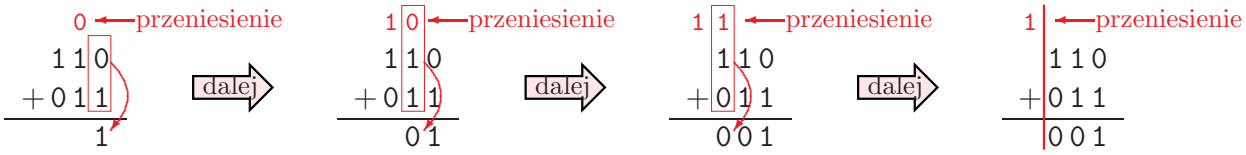
Jasne jest, że w taki sposób nie da się zrealizować prawdziwej arytmetyki liczb całkowitych — chociażby dlatego, że liczb całkowitych jest nieskończenie wiele, a ten licznikowy model pozwala tylko na skończoną ich ilość (jak już wspominałem, na n bitach od 0 do $2^n - 1$). Zamiast na liczbach całkowitych, komputer rachuje na **resztach** z dzielenia tych liczb przez 2^n.

Na przykład dla $n = 3$ będą to reszty z dzielenia przez $2^3 = 8$. Tak więc liczby $1, 9, 17, \ldots$ są reprezentowane przez 1; a liczby $5, 13, 21, \ldots$ są reprezentowane przez 5.

Dopóki zależy nam na liczbach małych w porównaniu z 2^n, to nie ma znaczenia, bo reszta z dzielenia małej liczby k przez 2^n jest równa k. Ale jeśli obliczenie programu na jakimś etapie zbliża się do 2^n, to należy wzmóc czujność, bo już coś może pójść niedobrze.

Arytmetyka reszt jest realizowana przez ucinanie na każdym etapie obliczenia tych bitów, które „wystają” w lewo poza przewidzianą długość liczby.

Na przykład dla naszej uproszczonej arytmetyki 3-bitowej dodawanie liczb 6 (binarnie: `110`) i 3 (binarnie: `011`) przebiega tak (od prawej do lewej):



Ponieważ w ostatnim dodawaniu bitów przeniesienie wypada już poza 3-bitowym polem, zostaje ucięte i wynikiem jest `001` czyli 1, czyli reszta z dzielenia $(6 + 3)$ przez 8;

Załóżmy teraz, że prowadzimy skomplikowany rachunek, przechodzimy przez liczby bardzo duże, ale spodziewamy się niezbyt wielkiego wyniku, swobodnie mieszczącego się na n bitach dla niego przeznaczonych. Czy fakt przejścia po drodze przez za duże liczby może spowodować, że ten niewielki wynik jest niezgodny z normalną matematyką? Na przykład, czy może nam wyjść $2 + 2 \neq 4$ dlatego, że liczyliśmy nie wprost, tylko stosując jakieś prawa matematyki, których arytmetyka komputerowa nie spełnia?

Okazuje się, że dopóki tylko dodajemy, odejmujemy i mnożymy, to jesteśmy pod tym względem bezpieczni[5]. Wynika to z faktu, że przejście do reszt jest *homomorfizmem*[6], czyli zachowuje te działania. Jest wszystko jedno, czy obliczenie wykonamy na prawdziwych liczbach całkowitych, a następnie przejdziemy do reszt z dzielenia przez 2^n; czy też będziemy każdy wynik pośredni *od razu* zastępować resztą z dzielenia przez 2^n (jak to czyni komputer). Dlatego wszystko będzie dobrze, o ile skądś wiemy, że ostateczny wynik będzie mniejszy niż 2^n.

Ale gwarancja bezpieczeństwa dotyczy tylko tych trzech działań: dodawania, odejmowania i mnożenia. Z dzieleniem jest niedobrze, nawet wtedy, gdy jego wynik jest całkowity (co wcale nie musi zachodzić). Na przykład ten program

```
#include<stdio.h>
int main() {
  unsigned int p = 2147483648;
  printf("\n  (%u * 3) / 4 == %10u", p, (p*3)/4);
  printf("\n  (%u / 4) * 3 == %10u\n\n", p, (p/4)*3);
  return 0;
}
```

= 2^{31}

znacznik %u dotyczy typu unsigned int

twierdzi (proszę samemu sprawdzić!), że wynik pomnożenia przez 3 i podzielenia przez 4 zależy od kolejności tych działań:

```
(2147483648 * 3) / 4 ==  536870912
(2147483648 / 4) * 3 == 1610612736
```

Żeby zrozumieć, co w tym programie się stało, wróćmy do modelu z niewielkimi liczbami bitów; niech znowu $n = 3$; a za wartość zmiennej `p` weźmy $2^2 = 4$. W arytmetyce takiego uproszczonego komputerka pierwsze obliczenie przebiegnie tak:

$$\begin{aligned}(4 \cdot 3)/4 &= 12/4 \quad \text{(ale w arytmetyce 3-bitowej: 12=4, więc:)}\\ &= 4/4 = 1\end{aligned}$$

a drugie obliczenie tak:

$$(4/4) \cdot 3 = 1 \cdot 3 = 3$$

Pierwsze obliczenie prowadziło przez liczby przekraczające 2^3 i dlatego dało niepoprawny wynik.

2.5 Gdzie są liczby ujemne?!

Na razie liczb ujemnych nie ma. Kogoś to zeźliło i postanowił nadać nazwę „-1” liczbie, która staje się zerem po dodaniu 1, a więc ciągowi bitów złożonemu z samych jedynek. I

[5]Podkreślam: jesteśmy bezpieczni **tylko wtedy**, gdy wiemy skądinąd, że ostateczny wynik zmieści się na n bitach. Jeśli ostateczny wynik jest za duży, to dostaniemy z niego tylko resztę z dzielenia przez 2^n.

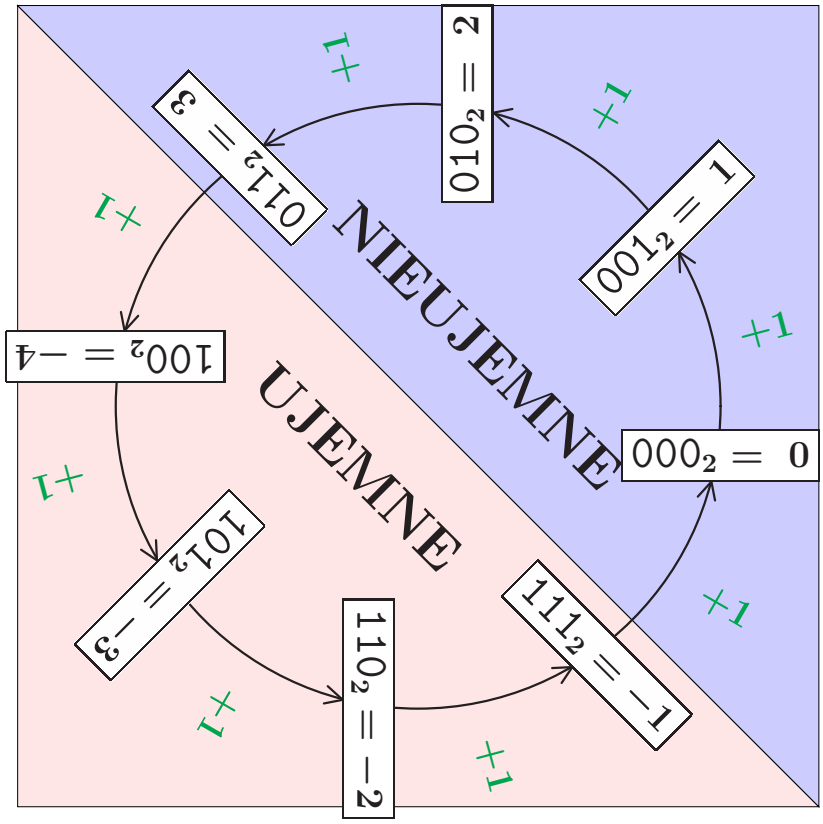
[6]Dokładniej: homomorfizmem pierścieni z $\mathbb{Z}$ do $\mathbb{Z}_{2^n}$. Szczegóły w podręcznikach algebry.

dalej: za ujemne uznać te liczby, które na najbardziej lewym bicie mają **1**; i każdą z nich nazwać

„minus [to, co trzeba do niej dodać, żeby otrzymać zero]”

W ten sposób powstał t.zw. *system U2 kodowania liczb całkowitych* — obecnie podstawa komputerowej arytmetyki liczb całkowitych.

Teraz nasz okrąg liczbowy z podrozdziału 2.2 (str. 5) będzie wyglądał tak:



Jak widać,

- najmniejszą liczbą ujemną jest -4,
- największą liczbą dodatnią jest 3,

a dodanie 1 do największej liczby dodatniej daje najmniejszą liczbę ujemną.

W rzeczywistej arytmetyce komputerowej te granice są oczywiście szersze. Poniższy program ilustruje skutek dodania 1 do najwyższej liczby dodatniej:

```
#include<stdio.h>
int main( ) {
  int  n1=0, n2=1;
  while (n2>=0) {
    n1++; n2++;
  }
  printf("\n  %i + 1 == %i\n\n", n1, n1+1);
  return 0;
}
```

Dwie zmienne `n1` i `n2` rosną tak długo, dopóki `n2` jest nieujemna. Gdyby to były *prawdziwe* liczby całkowite, to ta pętla powinna działać w nieskończoność, a wartości `n1` i `n2` powinny rosnąć nieograniczenie. Ale w arytmetyce komputerowej pętla `while` kończy działanie, a wtedy $\texttt{n2} < 0$ ale ciągle jeszcze $\texttt{n1} \geqslant 0$. Tak więc ten programik znajduje największą i najmniejszą liczbę typu `int` w danej implementacji języka C. Po uruchomieniu trzeba chwilę poczekać na wynik — w końcu ponad 2 miliardy obrotów pętli nie wykonują się błyskawicznie — po czym program wyświetla[7]:

```
2147483647 + 1 == -2147483648
```

Tak więc dla liczb całkowitych 32-bitowych:

- najmniejszą liczbą ujemną jest $-2^{31} = -2147483648$,
- największą liczbą dodatnią jest $2^{31} - 1 = 2147483647$.

Na początku tego podrozdziału napisałem beztrosko, że liczby ujemne to tylko *nazwy* pewnych ciągów bitów. Okazuje się, że tak jest w istocie. Zmienne typu `int` oraz typu bezznakowego `unsigned int` są w komputerze reprezentowane przez te same ciągi bitów i tak samo się je dodaje, odejmuje i mnoży. Czyli: wykonując te działania komputer „nie musi wiedzieć", czy dany ciąg bitów ma odpowiadać zmiennej typu `int` czy typu `unsigned int`. Rozróżnienie tych typów jest istotne tylko

- **w operacjach wejścia/wyjścia**; np. czy program ma wydrukować $2^{31}+2$ w postaci `-2147483646` czy w postaci `2147483650`;
- **przy porównywaniu wielkości**; np. czy program ma uznać, że liczba 2^{31} jest większa czy mniejsza od zera;
- **w innych operacjach arytmetycznych**; np. przy liczeniu ilorazu całkowitego albo reszty.

W (1 (str. 5)) było powiedziane, że ciąg bitów $c_{n-1}c_{n-2}\ldots c_2c_1c_0$ reprezentuje wartość

$$\sum_{i=0}^{n-1} c_i \cdot 2^i \;=\; c_0 \cdot 2^0 + c_1 \cdot 2^1 + \ldots + c_{n-1} \cdot 2^{n-1}$$

Ale przy naszej umowie dotyczącej liczb ujemnych należy przyjąć trochę inny wzór: bit stojący w liczbie na jej lewym skraju liczyć ze znakiem przeciwnym. Tak więc w n-bitowej

[7] Na moim komputerze; i na każdym stosującym dla liczb całkowitych zwykłą 32-bitową arytmetykę.

arytmetyce U2 ciąg $c_{n-1}c_{n-2}\ldots c_2c_1c_0$ bitów (t.zn. każde c_i jest albo zerem albo jedynką) reprezentuje liczbę

$$\sum_{i=0}^{n-2} c_i \cdot 2^i - c_{n-1} \cdot 2^{n-1} \;=\; \left(c_0 \cdot 2^0 + c_1 \cdot 2^1 + \ldots + c_{n-2} \cdot 2^{n-2}\right) - c_{n-1} \cdot 2^{n-1} \qquad (2)$$

Czyli: kolejne bity (od prawej do lewej) są mnożone przez kolejne potęgi dwójki i **sumowane**, z wyjątkiem najbardziej lewego, który jest mnożone przez kolejną potęgę dwójki i **odejmowany**.

2.6 Królik doświadczalny podstaw informatyki

— to oczywiście program liczący silnię. Dla każdego nowo poznawanego języka programowania student najpierw poznaje program drukujący *Hello, world!* , a zaraz potem program obliczający silnię. Taki obyczaj.

Moje programy na silnię dały mi takie wyniki:

dla 32-bitowych liczb całkowitych `int`	dla 64-bitowych liczb całkowitych `long`
1! == 1	1! == 1
2! == 2	2! == 2
3! == 6	3! == 6
4! == 24	4! == 24
5! == 120	5! == 120
6! == 720	6! == 720
7! == 5040	7! == 5040
8! == 40320	8! == 40320
9! == 362880	9! == 362880
10! == 3628800	10! == 3628800
11! == 39916800	11! == 39916800
12! == 479001600	12! == 479001600
13! == 1932053504	13! == 6227020800
14! == 1278945280	14! == 87178291200
15! == 2004310016	15! == 1307674368000
16! == 2004189184	16! == 20922789888000
17! == -288522240	17! == 355687428096000
	18! == 6402373705728000
	19! == 121645100408832000
	20! == 2432902008176640000
	21! == -4249290049419214848

? ?

Proszę przyjrzeć się tym liczbom. W lewej kolumnie `17!` okazało się ujemne — to znaczy, że typ `int` nie dał rady. Jednak problem z wynikami zaczął się wcześniej: już `13!` jest

różne w obu kolumnach. Ale nawet bez porównania kolumn jasne jest, że wynik w lewej jest niepoprawny, bo $13! = 1 \cdot \ldots \cdot 10 \cdot \ldots \cdot 13$, więc **musi** kończyć się zerem. Tak więc silnia liczby 13 to już za dużo dla typu `int`.

W typie `long` wynik ujemny otrzymałem dopiero dla `21!`. Wydaje się, że mniejsze silnie są policzone poprawnie, ale skąd pewność? Będziemy mogli ją mieć dopiero po przeczytaniu i przemyśleniu następnego podrozdziału 2.7 (str. 12).

2.7 Przy granicy jest niebezpiecznie...

Najmniejsze i największe liczby całkowite mogą być różne dla różnych języków programowania i różnych platform, na których te języki są zaimplementowane. Program w C może znaleźć aktualne wartości tych ograniczeń w pliku bibliotecznym `limits.h`. Dla mojego kompilatora `gcc` pod Linuksem wynoszą one:

typ	liczba bajtów	liczba bitów	zakres
`short`	2	16	od `SHRT_MIN` $= -2^{15} \approx -33$ tys. do `SHRT_MAX` $= 2^{15}-1 \approx 33$ tys.
`unsigned short`	2	16	od 0 do `USHRT_MAX` $= 2^{16}-1 \approx 66$ tys.
`int`	4	32	od `INT_MIN` $= -2^{31} \approx -2$ mld do `INT_MAX` $= 2^{31}-1 \approx 2$ mld
`unsigned int`	4	32	od 0 do `UINT_MAX` $= 2^{32}-1 \approx 4$ mld
`long`	8	64	od `LONG_MIN` $= -2^{63} \approx -9$ trln do `LONG_MAX` $= 2^{63}-1 \approx 9$ trln
`unsigned long`	8	64	od 0 do `ULONG_MAX` $= 2^{64}-1 \approx 18$ trln

1 mld (miliard) $= 10^9$ = tysiąc milionów
1 bln (bilion) $= 10^{12}$ = milion milionów
1 trln (trylion) $= 10^{18}$ = milion bilionów

(konkretnych liczb **w żadnym razie** nie trzeba znać na pamięć; należy tylko rozumieć, skąd się wzięły i jak je wyznaczyć „w razie czego".)

Dopóki wielkości przetwarzane przez nasz program leżą daleko od granic zakresów wynikających z powyższej tabelki, komputerowa arytmetyka liczb całkowitych jest zgodna z prawie wszystkimi prawami matematyki[8]. Jeśli jednak na jakimś etapie obliczeń wynik

[8] Wyjątkiem jest operacja ilorazu całkowitego `/` oraz operacja `%` reszty z dzielenia. Wyniki tych operacji odbiegają od matematycznie poprawnych wartości wtedy, gdy dzielna jest ujemna. Na przykład program w C stwierdzi, że

- iloraz całkowity: $(-7)/3 = -2$, oraz
- reszta: $(-7)\%3 = -1$.

(następna strona)⟶

częściowy przekroczy granicę zakresu, to wynik ostateczny może być niepoprawny. Programista ma często słabą kontrolę nad tym, co dzieje się we wnętrzu pętli obliczających skomplikowane wyrażenia, więc może taki błąd przeoczyć; szczególnie wtedy, gdy jest on mniej oczywisty niż w przykładach podanych powyżej (czyli **zawsze** w prawdziwym programowaniu). Wtedy na podstawie błędnego wyniku obliczenia ktoś podejmie niewłaściwą decyzję techniczną lub ekonomiczną.

Proszę więc przy programowaniu **bardzo** uważać na wszelkie okazje do nieświadomego przekroczenia zakresów.

Jeszcze uwaga na temat typów `short` i `unsigned short`. Należą one do historii informatyki; do czasów, kiedy pamięci było mało i oszczędność każdego bajtu była na wagę złota. Obecnie używanie ich do *jakichkolwiek* „normalnych" celów nie wydaje się sensowne. Zaoszczędzenie dwóch bajtów na liczbie prawie nigdy (poza bardzo specjalnymi zastosowaniami) nie jest warte niebezpieczeństwa przekroczenia 33 tysięcy i wygenerowania błędu obliczenia.

W „prawdziwej" arytmetyce reszta **zawsze** musi być nieujemna, więc wyniki będą takie:

- iloraz całkowity: $(-7)/3 = -3$, oraz
- reszta: $(-7) \mod 3 = 2$.

Zarówno w arytmetyce komputerowej jak prawdziwej zachodzi równość

$$\text{dzielna} = \text{dzielnik} \cdot \text{iloraz} + \text{reszta}$$

3 Jak rzeczywiste są komputerowe liczby rzeczywiste?

Nie są rzeczywiste, w żaden sposób nie mogą. Jeśli na komputerową zmienną rzeczywistą przeznaczymy n-bitów, to (jak już wiemy) może ona przyjąć najwyżej 2^n różnych wartości. A liczb rzeczywistych jest nieskończenie wiele. I to nie jest problem nieudanych prób sięgania po osi liczbowej zbyt daleko w górę i w dół, jak to było z liczbami całkowitymi. Nawet w niewielkim odcinku $[0\,..\,1]$ mieszka nieskończenie wiele liczb rzeczywistych, więc i ich wszystkich nie da się wiernie przedstawić w komputerze.

Może pójdzie nam lepiej, jeśli ograniczymy się do liczb wymiernych? — oczywiście nie. Wymiernych jest również nieskończenie wiele w dowolnym odcineczku o niezerowej długości. Komputerowe liczby rzeczywiste są wymierne[9], ale jest ich tylko znikoma garstka w stosunku do wszystkich liczb wymiernych.

Ponieważ wszystkie komputerowe liczby „rzeczywiste" są wymierne, więc nie możemy marzyć nie tylko o dokładnym obliczeniu obwodu koła o danej średnicy (bo π jest niewymierne), ale nawet długości przekątnej kwadratu o danym boku (bo $\sqrt{2}$ jest niewymierne). Musimy *przybliżać*, co powoduje błędy; a jeszcze należałoby się zastanowić, czy komputerowe liczby są na osi rzeczywistej rozmieszczone w sposób umożliwiający przybliżanie.

3.1 Jak się zapisuje duże liczby

Na pytanie „ile to jest bilion?" potoczną odpowiedzią może być „jedynka z 12 zerami"; to znaczy 1 pomnożone przez 10^{12}. Taki sposób prezentowania dużych liczb jest przyjęty w fizyce i innych naukach przyrodniczych. Podaje się niewielką liczbę dziesiętną z przedziału $[1\,..\,10)$, a następnie pisze się, przez jaką potęgę 10 należy ją pomnożyć, żeby uzyskać przybliżoną wartość, dającą pojęcie o rzędzie wielkości. Np.

długość orbity Ziemi	wynosi	$9.39887974 \cdot 10^{11}\,\mathrm{m}$	czyli	$9\underbrace{39887974000}_{11}\,\mathrm{m}$
masa Ziemi	wynosi	$5.97219 \cdot 10^{24}\,\mathrm{kg}$	czyli	$5\underbrace{972190000000000000000000}_{24}\,\mathrm{kg}$
stała Plancka	wynosi	$6.62607015 \cdot 10^{-34}\,\mathrm{kg{\cdot}m^2/s}$	czyli	$0.\underbrace{000\ldots06}_{34}62607015\,\mathrm{kg{\cdot}m^2/s}$

Dla komputerowych liczb rzeczywistych przyjęto taki właśnie t.zw. „zmiennopozycyjny" (ang. *floating point*) sposób zapisu: zmiana wykładnika potęgi powoduje przesunięcie kropki dziesiętnej w liczbie. Jednak jest to — jak zwykle w informatyce — kropka nie dziesiętna tylko binarna.

Zapis liczby rzeczywistej składa się więc ze *znaku* Z, *mantysy* M i *cechy* C; reprezentowana liczba jest równa

$$\ell \;=\; Z \cdot M \cdot 2^C$$

Przy tym

[9] To zostanie wyjaśnione bliżej w podrozdziale 3.2 (str. 16).

- **znak** ma tylko dwie możliwości: albo $Z = 1$, albo $Z = -1$;
- **mantysa** jest liczbą z przedziału $[1\,..\,2)$: $1 \leqslant M < 2$;
- **cecha** jest liczbą całkowitą.

Każda niezerowa liczba rzeczywista ma *dokładnie jedną* reprezentację $\langle Z, M, C\rangle$ spełniającą te warunki. Ta jedyność jest istotna, bo umożliwia łatwe porównywanie liczb. Dwie niezerowe liczby są równe wtedy i tylko wtedy, gdy wszystkie ich trzy części są odpowiednio równe: $Z_1 = Z_2$ i $M_1 = M_2$ i $C_1 = C_2$ więc dla porównania nie potrzebujemy stosować żadnego „sprowadzania do wspólnego mianownika" ani innych zabiegów.

Tylko dla zera należy zastosować jakieś niestandardowe wyjście.

Przykład 3.1
Żeby w ten sposób zapisać liczbę -12.4, działamy następująco:

- $Z = -1$, bo liczba jest ujemna; pozostaje wyznaczyć mantysę i cechę dla liczby 12.4;
- mantysa musi mieścić się w przedziale $[1\,..\,2)$, więc **dzielimy** ją kilkukrotnie przez 2 jednocześnie **zwiększając** cechę:

$$12.4 \;=\; 12.4 \cdot 2^0 \;=\; 6.2 \cdot 2^1 \;=\; 3.1 \cdot 2^2 \;=\; 1.55 \cdot 2^3$$

tak więc $M = 1.55$, $C = 3$.

Przykład 3.2
Żeby zapisać liczbę 0.0942, działamy następująco:

- $Z = 1$, bo liczba jest dodatnia; pozostaje wyznaczyć mantysę i cechę;
- mantysa musi mieścić się w przedziale $[1\,..\,2)$, więc **mnożymy** ją kilkukrotnie przez 2 jednocześnie **zmniejszając** cechę:

$$\begin{aligned} 0.0942 \;&=\; 0.0942 \cdot 2^0 \;=\; 0.1884 \cdot 2^{-1} \;=\; 0.3768 \cdot 2^{-2} \\ &=\; 0.7536 \cdot 2^{-3} \;=\; 1.5072 \cdot 2^{-4} \end{aligned}$$

tak więc $M = 1.5072$, $C = -4$.

A jak zapisuje się w pamięci komputera znak, mantysę i cechę?
Znak to tylko 1 bit; przyjmuje się, że `0` oznacza 1, a `1` oznacza -1.
Cecha to zwykła liczba całkowita zapisana w kodowaniu uzupełnieniowym U2; czyli wg wzoru (2 (str. 11)).
Kolejne bity *mantysy* $b_0 b_1 b_2 \ldots b_{n-1}$ to mnożniki do niedodatnich malejących potęg liczby 2:

$$M \;=\; \sum_{i=0}^{n-1} \frac{b_i}{2^i} \;=\; \frac{b_0}{1} + \frac{b_1}{2} + \frac{b_2}{4} + \ldots + \frac{b_{n-1}}{2^{n-1}}$$

Te trzy części liczby zapisuje się jedna za drugą w polu przeznaczonym na zmienną:

Na przykład gdybyśmy przeznaczyli na mantysę 3 bity, a na cechę 2 bity (więc na liczbę razem ze znakiem — 6 bitów), to zgodnie z tym, co było powiedziane wcześniej, liczbę -0.3125 zapiszemy tak:

- $Z = -1$, więc odpowiada mu bit `1`;
- $0.3125 < 1$, więc trzeba mnożyć:

$$0.3125 \cdot 2^0 \;=\; 0.625 \cdot 2^{-1} \;=\; 1.25 \cdot 2^{-2}$$

i dlatego $C = -2$ — bity `10`,
oraz $M = 1.25 \;=\; \frac{1}{1} + \frac{0}{2} + \frac{1}{4}$ — bity `101`.

Wobec tego:

$-0.3125 \;=\;$

1	**1**	**0**	**1**	**1**	**0**
znak	mantysa			cecha	

Proszę zwrócić uwagę na to, że

- żeby spełnić wymaganie $1 \leqslant M < 2$, najbardziej lewy bit mantysy *musi* być jedynką;
- nie da się w taki sposób zapisać liczby rzeczywistej 0.0.

Dlatego dla liczby 0.0 robimy wyjątek: będzie to ciąg samych zer. A jeśli to nie jest ciąg samych zer, ale najbardziej lewym bitem mantysy jest zero, to trzeba zasygnalizować błąd, bo to *nie jest* liczba rzeczywista.

3.2 Równy ciąg kuleczek, czy rozsypany bałagan?

Jakie liczby dają się *w ogóle* zapisać w ten sposób?

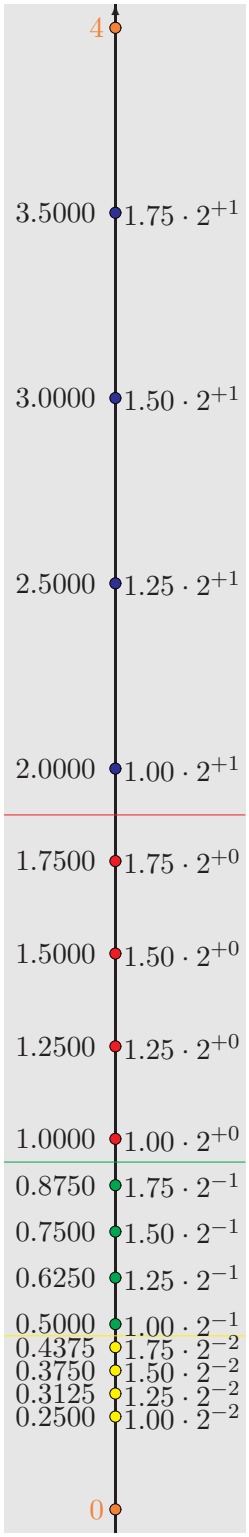
Przede wszystkim są to **wyłącznie** liczby „dwójkowo wymierne", czyli ułamki mające liczbę całkowitą w liczniku i potęgę liczby 2 w mianowniku. Więc jak zapisać choćby liczbę $\frac{1}{10}$? — ona nie jest dwójkowo wymierna (chociaż oczywiście jest wymierna), więc w komputerowej arytmetyce liczb rzeczywistych można ją tylko przybliżać. Łatwo pokazać, że wyrazy ciągu

$$\begin{aligned}
\tfrac{25}{256} &= \left(1 + \tfrac{9}{16}\right) \cdot 2^{-4}, \\
\tfrac{409}{4096} &= \left(1 + \tfrac{9}{16} + \tfrac{9}{16^2}\right) \cdot 2^{-4}, \\
\tfrac{6553}{65536} &= \left(1 + \tfrac{9}{16} + \tfrac{9}{16^2} + \tfrac{9}{16^3}\right) \cdot 2^{-4}, \\
\ldots &= \ldots \\
\left(1 + \tfrac{3}{5} \cdot \left(1 - \tfrac{1}{16^n}\right)\right) \cdot 2^{-4} &= \left(1 + \tfrac{9}{16} + \tfrac{9}{16^2} + \tfrac{9}{16^3} + \ldots + \tfrac{9}{16^n}\right) \cdot 2^{-4}, \\
\ldots &= \ldots
\end{aligned}$$

nieograniczenie zbliżają się do $\frac{1}{10}$.

Każdą liczbę rzeczywistą można przybliżać liczbami dwójkowo wymiernymi, ale większości z nich nie da się w ten sposób wyrazić dokładnie.

Z przyjętych zasad wynika, że tak zapisywane liczby, przynajmniej dopóki przeznaczymy na nie niewiele bitów, sięgają „niezbyt daleko” i „niezbyt głęboko”. Obok na rysunku zaznaczone są **wszystkie** liczby dodatnie możliwe do zapisania przy pomocy 3-bitowej mantysy i 2-bitowej cechy. Jeśli uwzględnimy znak, to dojdzie jeszcze tyle samo liczb ujemnych, rozłożonych symetrycznie; a jeśli uwzględnimy umowę dotyczącą zera, to jeszcze liczba 0.0. Razem — 33 różne liczby (na 6 bitach).



Proszę zwrócić uwagę na to, że „ziarno”, czyli odległość między dwiema kolejnymi wyrażalnymi liczbami, nie jest stałe. Im większa cecha, tym ziarno jest większe. Dokładniej: jeśli cecha wynosi C (a więc wystarcza do zapisu liczb w zakresie $[2^C \,..\, 2^{C+1})$), a mantysa ma m bitów, to ziarno wynosi $2^{-m+1} \cdot 2^C = 2^{C-m+1}$.

W poniższym zestawieniu będę podawał ziarno dla liczb w okolicach miliona. Ponieważ $2^{19} = 524\,288 \leqslant 1\,000\,000 < 1\,048\,576 = 2^{20}$, więc będę brał pod uwagę $C = 19$.

Dla *zwiększenia zakresu* wielkości liczb, trzeba przeznaczyć więcej bitów na cechę; żeby *zmniejszyć ziarno* przy zachowaniu tej samej wielkości, trzeba przeznaczyć więcej bitów na mantysę. Na przykład w kompilatorze `gcc` języka C przeznacza się

typ	**znak** bity	**mantysa** bity	**cecha** bity	**razem** **bity**	**razem** **bajty**
`float`	1	23	8	32	4
`double`	1	52	11	64	8

Tak więc wartość `cechy` liczby typu

- `float` mieści się w przedziale $[-2^7 \,..\, 2^7 - 1]$, czyli w przedziale $[-128 \,..\, 127]$;
- `double` mieści się w przedziale $[-2^{10} \,..\, 2^{10} - 1]$, czyli w przedziale $[-1024 \,..\, 1023]$.

Stąd wynika, że

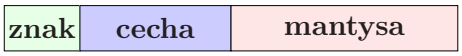
- liczby typu `float` sięgają prawie do $2 \cdot 2^{127} \approx 3.4 \cdot 10^{38}$ z ziarnem dla liczb w okolicy miliona wynoszącym $2^{19-23+1} = 2^{-3} = 0.125$
- liczby typu `double` sięgają prawie do $2 \cdot 2^{1023} \approx 1.8 \cdot 10^{308}$ z ziarnem dla liczb w okolicy miliona wynoszącym $2^{19-52+1} = 2^{-32} \approx 0.00000000023$

(również tych liczb **w żadnym razie** nie trzeba znać na pamięć; należy tylko rozumieć,

skąd się wzięły i jak do nich dojść „w razie czego").

Dla większości zastosowań ziarno wielkości 0.125 jest zbyt grube. Typ `float` jest takim samym archaicznym pterodaktylem jak typ `short`. Programy komputerowe często działają w ciasnych pętlach; jeżeli w każdym obrocie takiej pętli wyniki będą różnić się od poprawnych o $\frac{1}{8}$, to te różnice mogą się skumulować do ogromnego błędu. Proponuję dla liczb rzeczywistych stosować *wyłącznie* typ `double`.

Jeszcze ostrzeżenie dla tych, którzy chcieliby osobiście wniknąć w bity składające się na liczbę rzeczywistą. Otóż z różnych powodów technicznych bitowy kształt takiej liczby bywa inny niż opisany w tym podrozdziale. Po pierwsze cecha jest zwykle zamieniona z mantysą:



Poza tym reprezentacja bitowa fragmentów liczby odbiega od właśnie omówionego prostego obrazka. Jednak te odstępstwa nie mają wpływu na rozważania dotyczące paradoksów arytmetyki komputerowej. Dla ich wyjaśnienia takie nieco uproszczone omówienie jest wystarczające — chyba, żeby ktoś istotnie chciał grzebać się w pojedynczych bitach liczby rzeczywistej.

3.3 Niedokładne obliczenia na rzeczywistych

Ile wynosi 1.0 podzielone przez 3? Ze szkoły wiemy, że $0.(3)$, czyli „3 w okresie". Ale zapytajmy komputera. Poniższy program drukuje wynik z wzrastającą liczbą cyfr po kropce:

```
int main() {
  printf("\n 4 cyfry po kropce: 1/3 == %.4lf",  1.0/3);
  printf("\n 8 cyfr po kropce:  1/3 == %.8lf",  1.0/3);
  printf("\n 12 cyfr po kropce: 1/3 == %.12lf", 1.0/3);
  printf("\n 16 cyfr po kropce: 1/3 == %.16lf", 1.0/3);
  printf("\n 20 cyfr po kropce: 1/3 == %.20lf", 1.0/3);
  printf("\n\n");
  return 0;
}
```

Wydruk (proszę sprawdzić samemu):

```
4 cyfry po kropce: 1/3 == 0.3333
8 cyfr po kropce:  1/3 == 0.33333333
12 cyfr po kropce: 1/3 == 0.333333333333
16 cyfr po kropce: 1/3 == 0.3333333333333333
20 cyfr po kropce: 1/3 == 0.33333333333333331483
```

Jak widać, wyliczone wartości są takie, jakich należy się spodziewać, aż do 16-tego miejsca dziesiętnego po kropce; a dalej oczekiwane „3 w okresie" zawodzi. Istotnie, liczba $\frac{1}{3}$ powinna

być zapisana jako $\frac{4}{3} \cdot 2^{-2}$, a dla takich liczb wielkość ziarna wynosi

$$2^{-2-52+1} = 2^{-53} \approx 10^{-16}$$

Tak więc w typie `double` dla takich ułamków nie da się uzyskać lepszej dokładności niż do 16-tego miejsca po kropce.

W podrozdziale (1.2 (str. 2)) podany był przykład programu, który twierdził, że $\frac{5}{3}$ jest ostro większe od $5 \cdot \frac{1}{3}$. Żeby zrozumieć, dlaczego tak się działo, wydrukujmy te dwie liczby z 20 cyframi po kropce; tak samo, jak przed chwilą drukowaliśmy $\frac{1}{3}$. Otrzymujemy:

```
20 cyfr po kropce:     5/3 == 1.66666666666666674068
20 cyfr po kropce: 5*(1/3) == 1.66666666666666651864
```

15 cyfr

Cecha dla $\frac{5}{3} = \frac{5}{3} \cdot 2^0$ wynosi 0, więc ziarno jest wielkości $2^{0-52+1} = 2^{-51} \approx 4.5 \cdot 10^{-16}$. Dlatego wyniki zgadzają się do 15-tego miejsca po kropce; a dalej — hulaj dusza — zdaniem komputera pierwsza z tych liczb jest większa.

3.4 Zawodna równość

Wyniki różnych obliczeń na liczbach rzeczywistych, które *powinny* być takie same, mogą się różnić „o ziarno". Wtedy będą traktowane przez komputer jak gdyby były nierówne. Dlatego podstawową zasadą jest, żeby **nigdy** nie porównywać liczb rzeczywistych operatorem `==` ani `!=`. Proszę pamiętać: dla liczb rzeczywistych operatory `==` i `!=` są **bezwartościowe**. Czyli w rachunkach na liczbach rzeczywistych należy stosować się do zasady:

Chcesz uniknąć stu przykrości,
to zapomnij o równości.

Ale czym zastąpić takie porównania? — oczywiście sprawdzeniem bliskości. Zdecydować się, jaką różnicę uznać za zaniedbywalną i pytać o jej przekroczenie. Np. jeśli to jest jedna milionowa, 10^{-6}, to:

```
if (x1-x2 < 0.000001 && x2-x1 < 0.000001)  ...można uznać za równe...
else  ...nie są równe...
```

3.5 Inne zawodności

Proszę rozważyć program w C sprawdzający różne arytmetyczne oczywistości. Wiemy już, że w pewnych sytuacjach komputer będzie wyrażał inne zdanie niż matematyk. W szczególności wtedy, gdy spotykać się będą liczby znacznie różniące się wielkością. Tak jak w tym programie:

```
#include<stdio.h>

int main() {
  double x=1e16;

  if (1 == 0) printf("\n 1 == 0");
  if (1 > 0) printf("\n 1 > 0");
  if (1 < 0) printf("\n 1 < 0");

  if (x+1 == x) printf("\n x+1 == x");
  if (x+1 > x) printf("\n x+1 > x");
  if (x+1 < x) printf("\n x+1 > x");

  if ((x+1)+1 == x+(1+1)) printf("\n (x+1)+1 == x+(1+1)");
  if ((x+1)+1 > x+(1+1)) printf("\n (x+1)+1 > x+(1+1)");
  if ((x+1)+1 < x+(1+1)) printf("\n (x+1)+1 < x+(1+1)");

  if ((x+1)-(x-1) == 0) printf("\n (x+1)-(x-1) == 0");
  if ((x+1)-(x-1) > 0) printf("\n (x+1)-(x-1) > 0");
  if ((x+1)-(x-1) < 0) printf("\n (x+1)-(x-1) < 0");

  if ((x+1)*(x+1) == x*x+2*x+1) printf("\n (x+1)*(x+1) == x*x+2*x+1");
  if ((x+1)*(x+1) > x*x+2*x+1) printf("\n (x+1)*(x+1) > x*x+2*x+1");
  if ((x+1)*(x+1) < x*x+2*x+1) printf("\n (x+1)*(x+1) < x*x+2*x+1");

  printf("\n\n");
  return 0;
}
```

1 z 16 zerami

dla upewnienia się,
że program odróżnia liczbę 1 od zera

W każdej grupie komend program ma zadane pytanie, na które odpowiedź wydaje się oczywista, i ma do wyboru jedną z trzech możliwych odpowiedzi. Wydruk programu wygląda tak:

```
  1 > 0
  x+1 == x
  (x+1)+1 < x+(1+1)
  (x+1)-(x-1) == 0
  (x+1)*(x+1) < x*x+2*x+1
```

jedynka znika
dodawanie nie jest łączne
powinno wyjść 2 a nie 0
wzór skróc. mnoż. nie działa

Dokładna analiza, dlaczego tak się dzieje, może być nudna i żmudna. Wyjaśnię tylko, w jaki sposób wykonuje się dodawanie liczb znacznie różniących się wielkością.

Otóż kiedy komputer ma wykonać dodawanie dwóch liczb zmiennopozycyjnych o różnych cechach

$$\text{liczba}_1 + \text{liczba}_2 \;=\; M_1 \cdot 2^{C_1} + M_2 \cdot 2^{C_2} \;=\; ?$$

przy czym

$$1 \leqslant M_1 < 2 \qquad 1 \leqslant M_2 < 2 \qquad C_1 > C_2$$

to najpierw musi je sprowadzić do wspólnej cechy:

$$M_1 \cdot 2^{C_1} + M_2 \cdot 2^{C_2} \;=\; M_1 \cdot 2^{C_1} + \underbrace{(M_2 \cdot 2^{C_2 - C_1})}_{\text{nowa mantysa}} \cdot 2^{C_1} \;=\; (M_1 + (M_2 \cdot 2^{C_2 - C_1})) \cdot 2^{C_1}$$

Ale wykładnik $C_2 - C_1$ jest liczbą ujemną, co oznacza, że „nowa mantysa" $M_2 \cdot 2^{C_2 - C_1}$ już nie leży w przepisanym dla mantys przedziale $[1 \ldots 2)$, ale jest przesunięta w prawo o $C_2 - C_1$ bitów. Jeśli to przesunięcie jest duże, to może spowodować wysunięcie najbardziej prawych bitów mantysy poza przewidziany dla niej zakres; w skrajnych przypadkach nawet całej mantysy. W ten sposób liczba, którą komputer bez trudu odróżnia od zera, w trakcie dodawania może chwilowo być od zera nieodróżnialna.

4 Więc jak żyć?!

Jak się bronić przed czyhającymi zewsząd błędami obliczeń?

4.1 Brutalna siła

Wszystkie odstępstwa arytmetyki komputerowej od prawdziwej biorą się z konieczności ucinania różnych nieskończoności do skończonych zakresów, skończonego miejsca, ew. skończonego czasu. Im więcej tego miejsca czy czasu możemy przeznaczyć, tym mniejsze są błędy.

Nasuwa się więc propozycja, żeby inwestować w coraz większe i szybsze komputery, a na liczby poświęcać coraz więcej bitów. Ale to jest kosztowne; tym bardziej, że „praktyczne zastosowania" rzadko wymagają wielkich mocy, więc będą się one marnować przez większość „życia" komputera. W dodatku takie podejście nie likwiduje problemów, tylko je odsuwa.

W podrozdziale 2.6 (str. 11) zostało zademonstrowane, że już $13!$ nie mieści się w 32 bitach; dodanie następnych 32 bitów poprawiło sytuację tylko dla 8 liczb — już $21!$ nie mieści się w 64 bitach. No to dodajmy jeszcze więcej miejsca, jak w tej tabelce:

bajty	bity	największe n, dla którego $n!$ daje się zapisać
4	32	12
8	64	20
12	96	27
16	128	33

Jak widać, nadal, mimo czterokrotnego zwiększenia miejsca na liczbę, nie możemy nawet marzyć o silni dla liczb 3-cyfrowych. Brutalna siła zawiodła.

Ale jeśli nie zależy nam na kolejnych cyfrach wyniku a tylko na rzędzie wielkości, to możemy zadowolić się tym, że $34! \approx 3 \cdot 10^{38}$ — w przybliżeniu trójka z 38 zerami; a $100! \approx 10^{158}$. W wielu zadaniach nie warto zwiększać siły ciosu, lepiej poprzestać na skromniejszych sukcesach. Jak się nie ma, co się lubi, to się lubi, co się ma.

4.2 Analiza błędów

W metodach numerycznych poświęca się wiele uwagi błędom zaokrągleń. Oczywiście nie temu, jak je całkiem wyeliminować, bo to jest niemożliwe. Raczej takiemu opracowaniu algorytmów, żeby błędy zaokrągleń w kolejnych krokach obliczenia miały tendencję do wzajemnego kasowania się, a nie kumulowania. Oraz oszacowaniu spodziewanego lub maksymalnego błędu po wykonaniu wielu trochę błędnych kroków.

Takie podejście jest mniej prymitywne niż proste zwiększanie mocy komputera — ale analiza numeryczna jest trudna.

4.3 Mowa-trawa na zakończenie

Poświęciłem ponad 20 stron, żeby Czytelnika przerazić i przekonać, że to, co on w programowaniu obliczeń mógł uważać za twardy grunt, to w istocie grząskie bagno. Proszę jednak nie opuszczać rąk. Obliczeniom komputerowym *można* ufać tak samo, jak ufamy technicznemu projektowi budynku, czy prawostronnemu ruchowi na drodze. Czyli z ograniczeniami.

Zasada ograniczonego zaufania nie oznacza bezgranicznej nieufności. Oznacza tylko otwartość na to, że coś może pójść nie tak: architekt może się pomylić, kierowca może być pijany, a obliczenie komputera może dać absurdalny wynik. Powinniśmy rozpoznawać takie sytuacje i odpowiednio reagować. W szczególności informatyk musi wiedzieć, kiedy grozi błędność obliczenia, umieć ją rozpoznać, kiedy się zdarzy, i mieć w zanadrzu środki zaradcze.

Zakresy wielkości dla najpotrzebniejszych typów, takich jak `int` czy `double`, są tak dobrane, że wystarczają w „normalnych" rachunkach dla „normalnych" potrzeb. To stwierdzenie jest mętne, bo nie potrafię zdefiniować normalności tak, żeby ją ostro oddzielić od przesadnych oczekiwań. Ale jednak mało komu potrzebna jest silnia dla liczb 3-cyfrowych, czy dokładne dodawanie liczb różniących się o 16 rzędów wielkości. Dopóki więc unikamy takich pętli, w których niedokładności mogłyby się kumulować, możemy komputerom udzielać ograniczonego zaufania.

A ograniczenie zaufania powinno w praktyce programisty polegać na tym, żeby każdy program przed uznaniem za ukończony starannie testować; i żeby wynikom zawsze przyglądać się krytycznie i pytać zdrowego rozsądku, czy jemu te wyniki nie wydają się absurdalne.