Funkcje *div* i *mod*

Antoni Kościelski

1 Iloraz i reszta

1.1 Trochę informacji wstępnych

Mówiąc iloraz myślimy oczywiście o ilorazie całkowitoliczbowym (ilorazie w sensie dzielenia z resztą). W najprostszym przypadku liczb naturalnych z tym pojęciem i z resztą spotykamy się już w szkole podstawowej, gdy zapoznajemy się z algorytmem dzielenia. Z podobnymi pojęciami mamy do czynienia w przypadku wielomianów, a matematycy rozważają jeszcze inne, analogiczne struktury, w których możemy mówić o dzieleniu, na przykład pierścień Gaussa złożony z liczb całkowitych i dodatkowo rozszerzony o i (pierwiastek z -1).

Przypuśćmy, że dzielimy x przez d. Na ogół przyjmuje się, że iloraz q z dzielenia tych liczb i reszta r powinny spełniać następujące warunki:

$$x = q \cdot d + r \quad \text{oraz} \quad |\, r \,| < |\, d \,|, \tag{1}$$

gdzie $|\, d \,|$ oznacza tzw. funkcję Euklidesa dla d: w przypadku liczb całkowitych jest to moduł, a dla wielomianów – stopień. Rzecz jasna nas w pierwszym rzędzie będą interesować liczby całkowite.

Oczywiście, warunki (1) nie definiują ani ilorazu, ani reszty. Mamy na przykład $q \cdot d + r = (q+1) \cdot d + (r - d)$. Jednak w wielu przypadkach te warunki są wystarczające. Matematycy – w przeciwieństwie do informatyków – nie mają ważnych powodów, aby je ujednoznaczniać. Przydają się także w bardziej ogólnych sytuacjach. Wszyscy się jednak zgadzają, że powinny być spełnione.

1.2 Definicja matematyczna, zwana też euklidesową

Następujące twierdzenie (czasem przypisywane Euklidesowi) słuszne jest nawet w przypadku liczb rzeczywistych:

Twierdzenie 1.1 *Niech d będzie niezerową liczbą rzeczywistą. Dla każdej liczby rzeczywistej x istnieją jednoznacznie określone liczby q i r takie, że*

$$x \;=\; q \cdot d + r \quad \textit{oraz} \quad 0 \leqslant r < |\, d \,|. \tag{2}$$

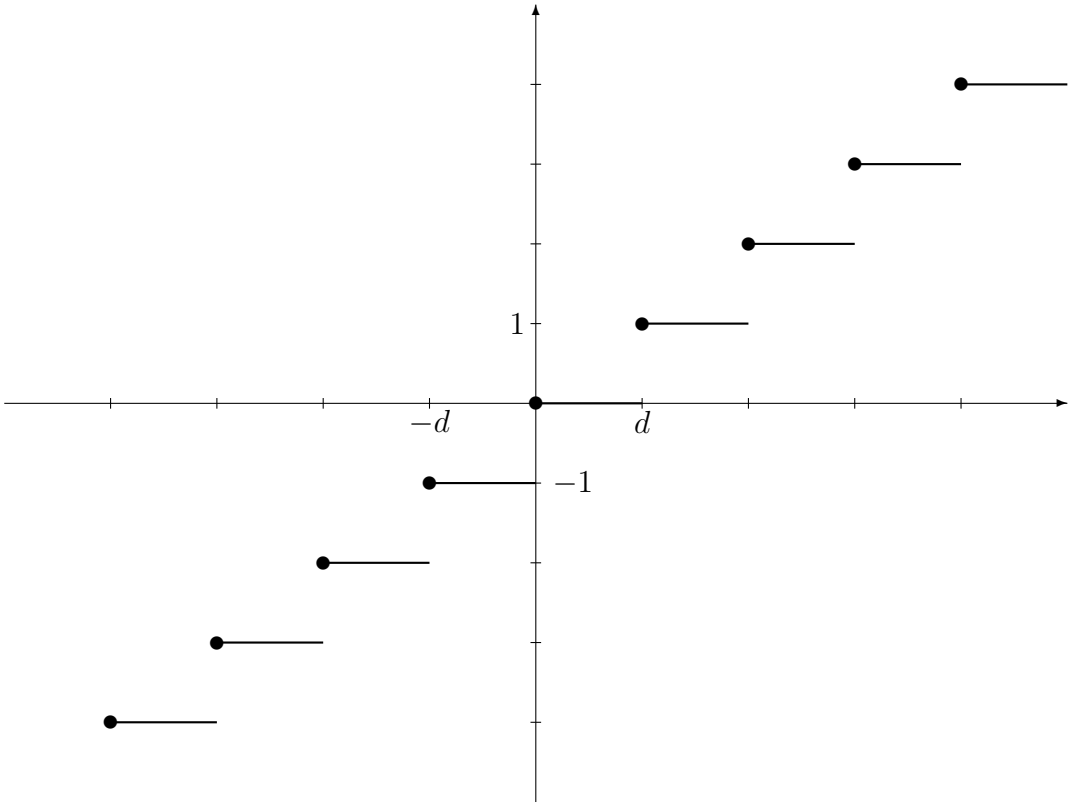
Nie jest trudne do udowodnienia i dla $d \neq 0$ pozwala zdefiniować dwie wartości $x\ div_E\ d$ oraz $x\ mod_E\ d$. Aby to zrobić wystarczy zażądać, by zachodziły warunki (2), a więc by

$$x \ = \ (x\ div_E\ d) \cdot d + r \quad \text{dla pewnego } \ r \ \text{ takiego, że } \ 0 \leqslant r < |\, d\,|$$

oraz

$$x\ mod_E\ d \ \ = \ \ x - (x\ div_E\ d) \cdot d.$$

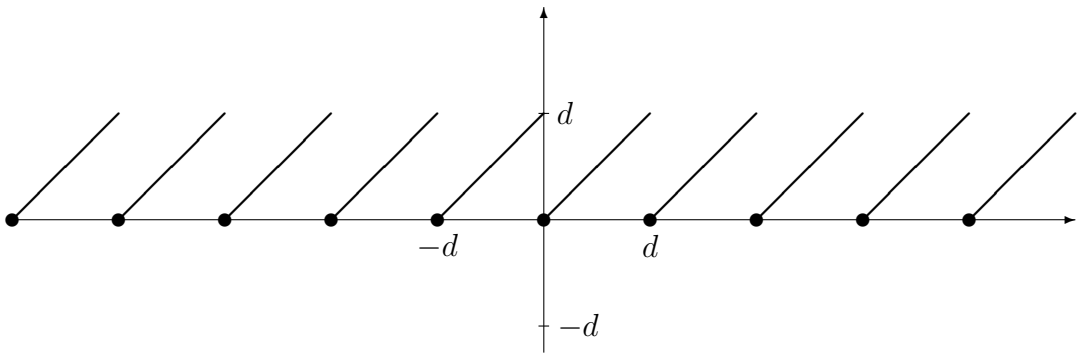
Jak widać nie został określony sposób obliczania wartości $x\ div_E\ d$. Zostały tylko podane warunki, jakie ta wartość powinna spełniać. Dla dodatnich argumentów do obliczenia $x\ div_E\ d$ możemy stosować znany ze szkoły algorytm dzielenia. Niżej są przedstawione wykresy funkcji div_E i mod_E dla ustalonej, dodatniej wartości d. Zauważmy jeszcze, że obie funkcje są zdefiniowane dla argumentów rzeczywistych, ale dalej zwykle będziemy je ograniczać do argumentów całkowitych.



Rys. 1. Wykres funkcji $x\ div_E\ d$ dla $d > 0$.

Nietrudno zauważyć, że $x\ div_E\ (-d) = -(x\ div_E\ d)$ dla wszystkich $d \neq 0$. Wzór ten pozwala narysować wykres funkcji $x\ div_E\ d$ także dla $d < 0$.

Przedstawiona definicja ilorazu i reszty jest bardzo naturalna i nie budzi zastrzeżeń informatyków. Mimo to w wielu językach programowania są implementowane inne definicje.



Rys. 2. Wykres funkcji $x\ mod_E\ d$ dla $d > 0$, jest też poprawny dla $d < 0$, w tym drugim przypadku powinniśmy na rysunku d zastąpić przez $\mid d \mid$.

1.3 Inne definicje ilorazu i reszty

Oprócz euklidesowej rozważa się wiele innych definicji ilorazu całkowitoliczbowego. Zajmiemy się teraz tylko dwoma podstawowymi i najczęściej stosowanymi.

Będą nam potrzebne dwie funkcje kownertujące liczby rzeczywiste na całkowite. Pierwszą będzie zwykła część całkowita liczby, która powinna nam się kojarzyć z literką F. Przyjmijmy, że

$$F(x) = floor(x) = \lfloor x \rfloor = \max\{a \in Z \mid a \leqslant x\}.$$

Druga funkcja – będziemy ją oznaczać literką T – jest czasem określana jako przycinanie i jest zdefiniowana wzorem

$$T(x) = trunc(x) = \begin{cases} \max\{a \in Z \mid a \leqslant x\} & \text{dla } \ x \geqslant 0, \\ \min\{a \in Z \mid x \leqslant a\} & \text{w przeciwnym razie.} \end{cases}$$

Możemy wyobrażać sobie, że $F(x)$ to pierwsza napotkana liczba całkowita podczas wędrówki wzdłuż osi x-ów od punktu o współrzędnej x, w lewo, a $T(x)$ to pierwsza napotkana liczba całkowita podczas wędrówki od punktu x w kierunku liczby 0. Mamy więc $F(-1,33) = -2$ oraz $T(-1,33) = -1$.

Mając te dwie funkcje możemy zdefiniować iloraz i resztę na dwa sposoby przyjmując, że

$$x\ div_F\ d = F(x/d) \ \text{ oraz } \ x\ mod_F\ d = x - (x\ div_F\ d) \cdot d,$$

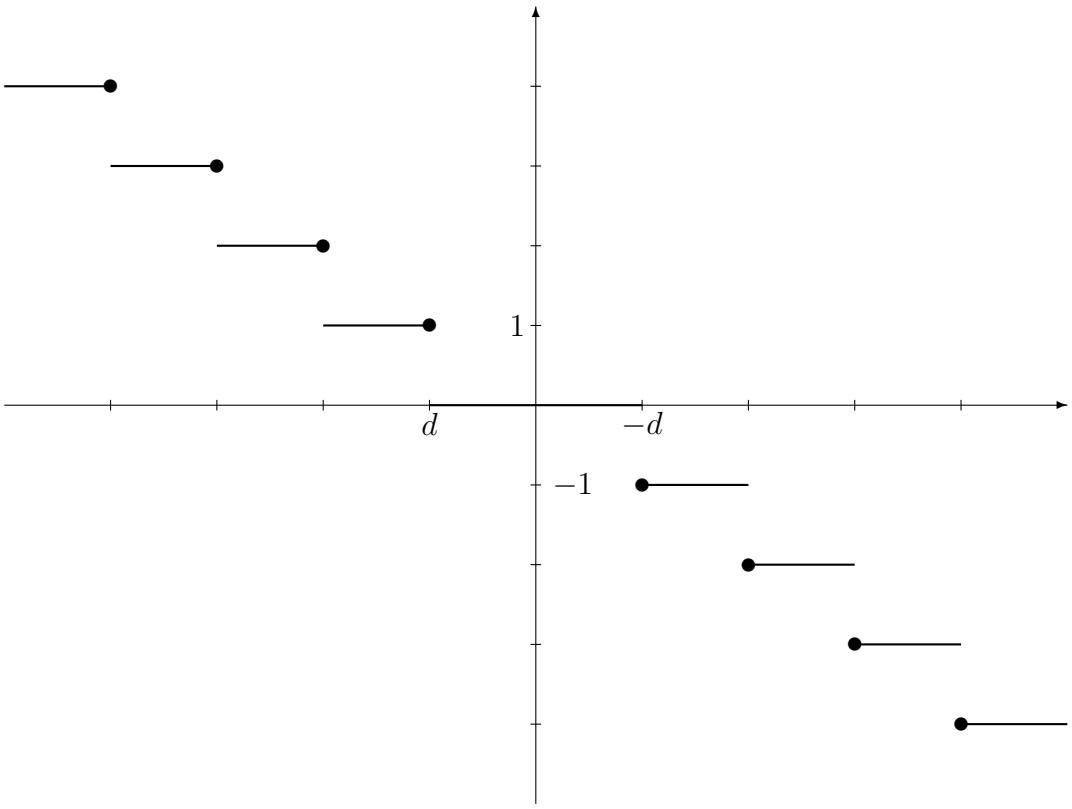
gdzie / oznacza dzielenie w liczbach rzeczywistych, oraz

$$x\ div_T\ d = T(x/d) \ \text{ oraz } \ x\ mod_T\ d = x - (x\ div_T\ d) \cdot d.$$

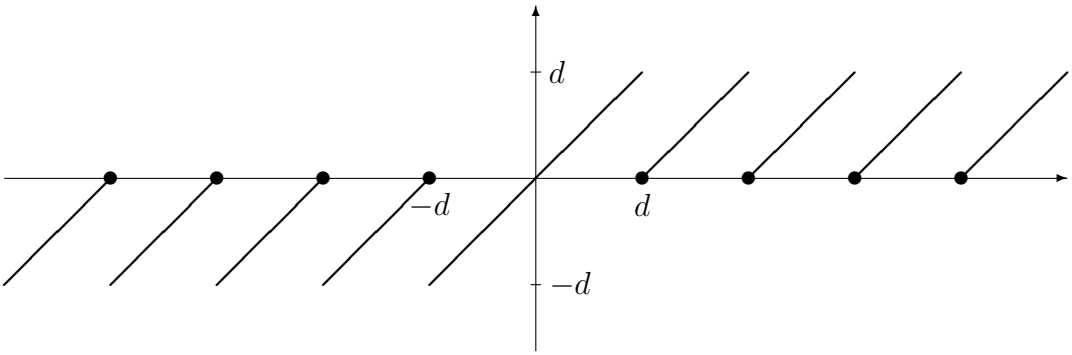
Można zauważyć, że funkcje F i T dla każdego argumentu dodatniego przyjmują te same wartości, to samo dotyczy ilorazów div_E, div_F oraz div_T, jak i reszt mod_E, mod_F i mod_T.

Obie zdefiniowane wyżej pary funkcji *div* i *mod* spełniają warunki (1). Parę div_F i mod_F analizował i może polecał Donald Knuth.

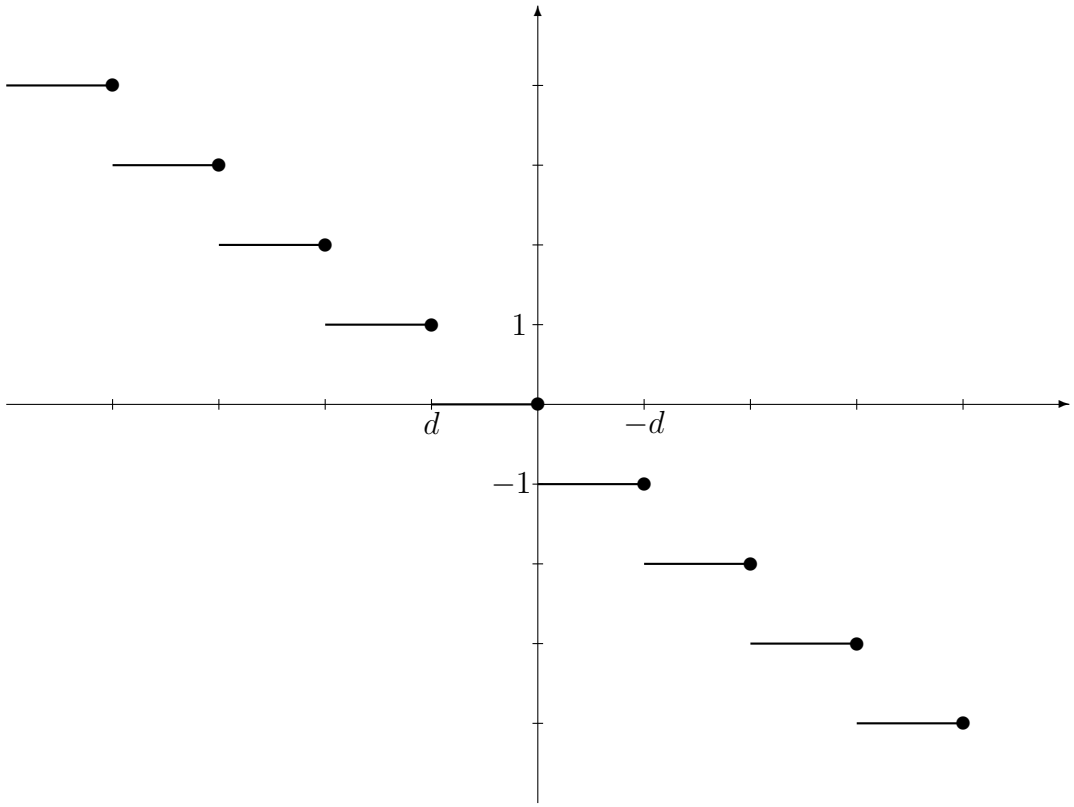
Niżej zostały przedstawione wykresy ilorazów div_T oraz div_F, jak i reszt mod_T i mod_F, narysowane czasem przy pewnych założeniach. Proponuję sprawdzenia poprawności tych wykresów. Oglądając je można zauważyć, że mamy do czynienia z trzema różnymi definiacjami ilorazów i reszt.



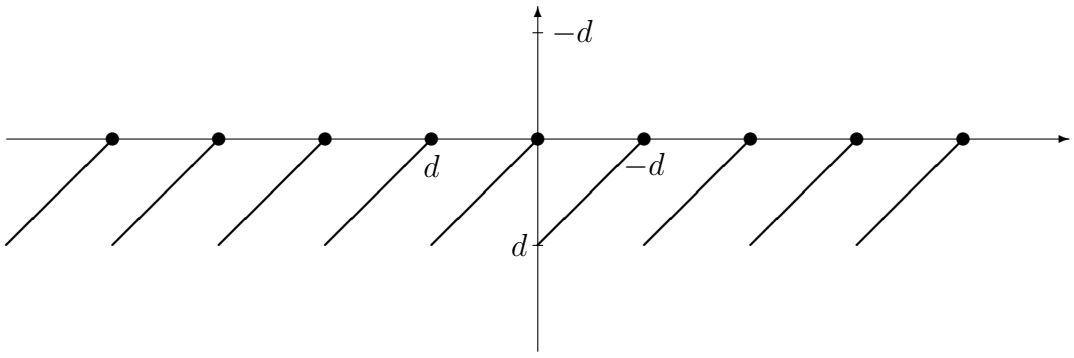
Rys. 3. Wykres funkcji $x\ div_T\ d$ dla $d < 0$.



Rys. 4. Wykres funkcji $x\ mod_T\ d$ dla dowolnych d,
dla $d < 0$ powinniśmy na rysunku zastąpić d przez $\mid d \mid$.



Rys. 5. Wykres funkcji $x\ div_F\ d$ dla $d < 0$.



Rys. 6. Wykres funkcji $x\ mod_F\ d$ dla $d < 0$.

1.4 Iloraz całkowitoliczbowy w różnych językach programowania

Dzielenie całkowitoliczbowe jest różnie definiowane w różnych językach programowania. Nie jest mi znany język, który oferowałby euklidesową definicję ilorazu.

W Pascalu i kilku innych, starszych językach były wykorzystywane definicje określane jako „złe". Nie zawsze spełniały warunek (1).

W języku *C* początkowo sposób implementowania dzielenia zależał od twórców kompilatora. Teraz przyjmuje się, że dla liczb całkowitych $x/d = x\ div_T\ d$ oraz $x\%d = x\ mod_T\ d$.

W Jawie prawdopodobnie jest tak samo, a autorzy jednego z opisów uważają, że dostatecznie dużo informacji na ten temat zawiera zdanie *Musimy także wspomnieć o tym, że operator procentu (%) to reszta z dzielenia i może zwrócić wartość ujemną.*

W Pythonie dzielenie jest definiowane za pomocą części całkowitej (używane są funkcje div_F i mod_F). Podobie jest w Prologu, a podstawowy podręcznik do programowania w tym języku zawiera bardzo dobrą definicję ilorazu.

W końcu Haskell oferuje programistom dwie pary funkcji: *qout* i *rem* oraz *div* i *mod*. Są to odpowiednio pary div_T i mod_T oraz div_F i mod_F.

2 Zadanie 2 z listy 2 (2018)

2.1 Sformułowanie

Podaj precyzyjną specyfikację i zgodny z nią algorytm w postaci schematu blokowego dla następującego, sformułowanego opisowo problemu algorytmicznego: dla danych dwóch liczb całkowitych podaj wynik ich dzielenia całkowitego i resztę z tego dzielenia. Napisz w kodzie RAM program odpowiadający podanemu schematowi i wyznacz jego złożoność asymptotyczną. Uwzględnij, że mogą zostać podane liczby ujemne.

2.2 Specyfikacje

Rozważane zadanie może być chyba rozumiane na kilka sposobów. Bierze się to stąd, że – jak sądzę – nie ma ustalonej definicji dzielenia całkowitego w przypadku, gdy dzielna, dzielnik, lub obie dzielone liczby są ujemne.

Na wykładzie jest rozważana maszyna RAM, która wykonuje dzielenie całkowite (rozkaz DIV), także dla ujemnych danych. Co więcej, chyba wiadomo, jaki jest efekt wykonania tego rozkazu: oblicza on iloraz rozumiany jako wartość funkcji div_F. Może więc trzeba jeszcze tylko wyliczyć resztę, czyli wartość funkcji mod_F?

Takie rozumienie zadanie prowadzi do następującej specyfikacji:

Specyfikacja 1.

Dane: dwie liczby całkowite: m (dzielna) i $n \neq 0$ (dzielnik),

Wyniki: dwie wartości $m\ div_F\ n$ i $m\ mod_F\ n$ (zdefiniowane wyżej).

Ta specyfikacja ma pewną wadę. Zakłada, że maszyna RAM dobrze dzieli (dzieli, jak chcemy) wszelkiego rodzaju liczby całkowite i w konsekwencji nie musimy uwzględniać, że coś zależy od znaków danych. Może więc powinniśmy myśleć inaczej. Na przykład: maszyna RAM, co prawda, potrafi wykonywać rozkaz DIV i liczyć wartości funkcji div_F, ale prawdziwym wynikiem dzielenia jest wartość funkcji div_E, a prawdziwą resztę z dzielenia zwraca funkcja mod_E. Takie rozumienie treści zadania oddaje

Specyfikacja 2.

Dane: dwie liczby całkowite: m (dzielna) i $n \neq 0$ (dzielnik),

Wyniki: dwie wartości $m\ div_E\ n$ i $m\ mod_E\ n$ (zdefiniowane wyżej).

Można zaproponować jeszcze kilka dalszych specyfikacji opartych o takie rozumienie treści zadania: może mamy wyliczyć jakoś rozumiany iloraz przy założeniu, że maszyna potrafi obliczać jakiś inny.

Specyfikacja 3. Najładniejsze rozumienie treści zadania może być jeszcze inne. Może być formalnie zgodne ze specyfikacją 2, ale będziemy dodatkowo zakładać, że RAM nie wykonuje ani rozkazu DIV, ani MULT. Nie jest to dziwne założenie. Pierwotnie maszyny RAM nie wykonywały tych dwóch rozkazów, były modelami rzeczywistych komputerów, a dawne procesory potrafiły głównie dodawać i odejmować. Aby wtedy pomnożyć lub podzielić dane, procesor musiał już wykonać jakiś programik wykorzystujący dodawanie i odejmowanie a zwykle także pewne rozkazy pomocnicze.

2.3 Algorytmy

2.4 Algorytm realizujący specyfikację 1

Podamy tylko program dla maszyny RAM realizujący obliczenia zgodnie ze specyfikacją 1. Nie będzie on spełniał wszystkich warunków wymienionych w definicji takich programów. W szczególności, zamiast numerów (adresów) komórek pamięci wystąpią w nim symboliczne nazwy komórek takie, jak $\langle dzielna \rangle$. W bardziej formalnym programie te nazwy powinny zostać zastąpione przez odpowiednie numery, w sposób różnowartościowy.

Programując maszynę RAM możemy naśladować programowanie w zwykłych językach programowania. Możemy też naśladować programowanie procesora w kodzie wewnętrznym. W tym drugim przypadku odczytywanie danych i wypisywanie wyników powinno być wykonywane przez procesor i powinniśmy korzystać z rozkazów `READ 0` i `WRITE 0`. Będziemy jednak raczej wyświetlać wartości komórek naśladując wyświetlanie wartości zmiennych. Wydłuża to program o przesyłanie wyników obliczeń z procesora do komórki pamięci. Dodatkowo zmianę znaku liczby wykonujemy za pomocą „skomplikowanego” rozkazu `MULT = -1`, za to robimy to w bardzo krótki sposób.

```
READ ⟨dzielna⟩
READ ⟨dzielnik⟩
LOAD ⟨dzielna⟩
DIV ⟨dzielnik⟩     //w akumulatorze F(x/d)
STORE ⟨iloraz⟩
MULT ⟨dzielnik⟩     //w akumulatorze F(x/d)·d
SUB ⟨dzielna⟩     //w akumulatorze F(x/d)·d − x
MULT = -1    //zmiana znaku, reszta w akumulatorze
STORE ⟨reszta⟩
WRITE ⟨iloraz⟩
WRITE ⟨reszta⟩
```

2.5 Dla specyfikacji 2

Najpierw musimy opracować algorytm obliczający div_E i mod_E przy założeniu, że potrafimy obliczać wartość $F(x/d)$. Zauważmy więc, że

$$F(x/d) \leqslant x/d < F(x/d) + 1$$

(z definicji F). Stąd dla nieujemnych d otrzymujemy, że $0 \leqslant x - F(x/d) \cdot d < d$. W przeciwnym razie zachodzi nierówność $d < x - F(x/d) \cdot d \leqslant 0$. Tak więc zawsze mamy

$$| x - F(x/d) \cdot d | < | d |.$$

Mamy też oczywistą równość $x = F(x/d) \cdot d + (x - F(x/d) \cdot d)$. Jeżeli dodatkowo założymy, że $0 \leqslant x - F(x/d) \cdot d$, to na podstawie twierdzenia 1.1 możemy stwierdzić, że

$$x \; div_E \; d = F(x/d) \;\text{ oraz }\; x \; mod_E \; d = x - F(x/d) \cdot d. \tag{3}$$

Natomiast gdy $x - F(x/d) \cdot d < 0$, to liczba d musi być ujemna i wtedy mamy

$$0 < x - F(x/d) \cdot d - d < -d = | d |.$$

Ponieważ $x = (F(x/d) + 1) \cdot d + (x - F(x/d) \cdot d - d)$, więc z twierdzenia 1.1 otrzymujemy, że

$$x \; div_E \; d = F(x/d) + 1 \;\text{ oraz }\; x \; mod_E \; d = x - F(x/d) \cdot d - d. \tag{4}$$

Przedstawione rozważania pozwalają opracować algorytm realizujący specyfikację 2: obliczamy wartość $x - F(x/d) \cdot d$ i w zależności od tego, czy jest to liczba ujemna, czy nie, zwracamy wyżej określone wyniki. Zgodnie z takim algorytmem działa maszyna RAM wykonująca następujący program. Program ten został napisany zgodnie z takimi samymi zasadami, jak poprzedni.

```
     READ ⟨dzielna⟩
     READ ⟨dzielnik⟩
     LOAD ⟨dzielna⟩
     DIV ⟨dzielnik⟩    //w akumulatorze F(x/d)
     STORE ⟨iloraz⟩
     MULT ⟨dzielnik⟩    //w akumulatorze F(x/d)·d
     SUB ⟨dzielna⟩    //w akumulatorze F-reszta ze zmienionym znakiem
           //czyli F(x/d)·d − x.
     JGTZ mod    //skok, gdy F-reszta jest ujemna, są wtedy konieczne poprawki
     MULT =-1    //teraz resztę (oraz iloraz) liczymy zgodnie z (3)
     STORE ⟨reszta⟩
     JUMP end
mod: MULT =-1    //wyliczenie E-reszty, liczymy zgodnie z (4)
     ADD ⟨dzielnik⟩    //w akumulatorze x − F(x/d)·d − d
     STORE ⟨reszta⟩
     LOAD ⟨iloraz⟩    //liczymy iloraz zgodnie z (4)
     SUB =1
     STORE ⟨iloraz⟩
end: WRITE ⟨iloraz⟩    //przekazanie wyników
     WRITE ⟨reszta⟩
```

2.6 Specyfikacja 3

Też najpierw musimy opracować potrzebny algorytm. Szukanie ilorazu i reszty jest proste, gdy dzielna i dzielnik są dodatnie. Wtedy odejmujemy dzielnik od dzielnej tak długo, aż to, co zostaje z dzielnej stanie się mniejsze od dzielnika. Iloraz w tym przypadku jest liczbą wykonanych odejmowań. Podobnie postępujemy, gdy dzielnik jest dodatni, a dzielna – ujemna: wtedy dzielną powiększamy o dzielnik tak długo, aż przestanie być ujemna. A co robimy, gdy dzielnik jest ujemny? Otóż możemy taką sytuację sprowadzić do poprzedniej korzystając z wzoru $a \cdot (-b) = (-a) \cdot b$ (możemy zmienić znak dzielnika, wyliczyć iloraz, a następnie poprawić znak ilorazu).

Mając jakieś wyobrażenie o algorytmie możemy napisać program. Tym razem napiszemy go w języku przypominającym C.

read(dzielna); **read**(dzielnik);

iloraz = 0; reszta = dzielna; znak = 1; //znak ma być znakiem dzielnika

if (dzielnik < 0) {znak = − 1; dzielnik = − dzielnik;}

  //dzielnik jest teraz dodatni i od tego momentu podczas wykonywania programu

  //stale zachodzi równość: dzielna = iloraz · dzielnik + reszta

while (dzielnik ⩽ reszta) {reszta = reszta − dzielnik; iloraz = iloraz + 1;}

  //teraz: reszta < dzielnik i następna instrukcja tego nie zmienia

while (reszta < 0) {reszta = reszta + dzielnik; iloraz = iloraz − 1;}

  //zachodzi: 0 ⩽ reszta < dzielnik

dzielnik = znak · dzielnik; iloraz = znak · iloraz; //dzielnik odzyskał początkową wartość

  //nadal dzielna = iloraz · dzielnik + reszta, a także 0 ⩽ reszta < | dzielnik |

write(iloraz); **write**(reszta);