Wstęp do informatyki z przykładami w języku C

Jacek Cichoń, Przemysław Kobylański
WPPT, Politechnika Wrocławska

WRZESIEŃ 2010

Spis treści

Wstęp

Książka ta jest wstępem do informatyki rozumianej jako nauki o algorytmach. Takie potraktowanie informatyki jest pewnym zawężeniem dziedziny. Jednak według autora ten fragment informatyki jest jej jądrem, najbardziej istotną częścią. Nauka o algorytmach, zwana **ALGORYTMIKĄ**, odgrywa dla całej informatyki taką rolę jak logika dla dla matematyki (patrz [1]).

W książce nie ma informacji o tym jak jest zbudowany rzeczywisty komputer. Nie są omawiane systemy operacyjne. Nie jest omawiany system plików. Zakładamy, że czytelnik tej książki jest średnio zaawansowanym użytkownikiem komputerów. Mimo, że do zapisu algorytmów stosowany jest język C, nie jest omawiany żaden kompilator.

Rozdział 1

Wprowadzenie do języka C

```
#include <stdio.h>
int main(){
    printf("Hello world.\n");
    return(0);
}
```

W pierwszej części tej książki zajmiemy się podstawowymi technikami programowania - zmiennymi i stałymi, arytmetyką, instrukcjami sterującymi, głównymi typami danych oraz podstawowymi metodami wejścia i wyjścia. Programowania, tak samo jak pływania, nie można nauczyć się teoretycznie. Do jego opanowania potrzebna jest spora ilość praktyki i cierpliwości.Nie spodziewaj się, że po samym przeczytaniu tej książki będziesz umiał programować. Musisz robić zadania i pisać samodzielnie jak najwięcej programów. Wykonuj samodzielne eksperymenty i czytaj możliwie dużo na temat programowania.

W książce tej będziemy posługiwali się językiem programowania C. Został on stworzony w roku 1972 przez Dennisa Ritchie z Laboratorium Bella. W roku 1978 B. K. Kernighan i D. M. Ritchie opublikowali książkę *"The C Programming Language"* (patrz [2]), która spowodowała sporą rewolucję w świecie informatycznym.

Język C jest bazą bardziej rozbudowanych języków takich jak C++, PHP oraz Java. Jest bardzo zwięzłym językiem i pozwala na wykonywanie wielu niskopoziomowych (bliskich sprzętu) operacji. Jego niewątpliwą zaletą jest to, że pozwala programiście praktycznie na wszystko. Po pewnym czasie przekonasz się, że to, jak również jego zwięzłość jest równocześnie jego wadą.

Książka ta nie jest podręcznikiem programowania w języku C. Twoją podstawową lekturą uzupełniającą powinna być klasyczna książka B. K. Kernighana i D. M. Ritchie pt. „*Język ANSI C*" (patrz [3]). W naszej książce posługiwać się będziemy tylko podstawowymi konstrukcjami tego języka i równie dobrze do zapisu rozważanych algorytmów moglibyśmy posługiwać się innym, tak zwanym imperatywnym językiem programowania. Mógłby być nim na przykład Pascal, Basic, Ada, Fortran czy też Forth.

Program w języku C jest plikiem tekstowym, który przetłumaczony musi być na język zrozumiały przez komputer. Do tego przekształcania służą kompilatory.

Kompilatorami nazywamy programy które przekształcają kody programów napisanych w zbiorach tekstowych na pliki które mogą być wykonywane przez komputer.

Do pisanie programów w języku C wystarczy dowolny edytor, który potrafi zapisywać zbiory w postaci plików testowych. Jednakże nie jest to najlepszym rozwiązaniem. W trakcie pisania pisania programów łatwo można popełnić wiele błędów. Warto jest więc mieć narzędzie, które ułatwia nam pisanie programów, testowanie oraz wyszukiwanie i eliminowanie błędów. Istnieją kompilatory, które oferują użytkownikowi zintegrowane środowisko programowania. Zawierają one wyspecjalizowany edytor, narzędzia do analizy i śledzenia programów i wyposażone są w systemy pomocy. Niektóre z tych narzędzi są ogólnodostępne i łatwo je zdobyć za pomocą Internetu. Narzędzie takie musisz zdobyć, zainstalować na komputerze i bardzo dobrze poznać. Pamiętaj, że spędzisz z tym narzędziem bardzo wiele godzin. Warto więc je polubić.

W roku 1983 American National Standards Institute (ANSI) powołał komitet, którego zadaniem było opracowanie standardu języka C. Powstał on w roku 1988 i ta wersja języka nazywa się „ANSI C". W książce tej będziemy się posługiwali tylko pewnym fragmentem tego języka.

1.1 Pierwszy program

Wiele podręczników nauki programowania rozpoczyna się od analizy programu, których jedynym celem jest wyświetlenie na ekranie monitora napisu „Hello world.". Zadanie to wcale nie jest proste. Aby je zrealizować musisz umieć zapisać tekst programu przestrzegając ściśle określonych reguł języka programowania, przetłumaczyć go za pomocą kompilatora i uruchomić. A oto kod programu który wypisuje na ekranie napis „Hello world.": ℂ

```c
#include <stdio.h>
int main()
{
  printf(''Hello world.\n'');
  return(0);
}
```

Kod programu w języku C zapisany musi być w pliku tekstowym. Jeśli używasz zintegrowanego środowiska programowania, to musisz stworzyć nowy plik, przepisać powyższy tekst i zapisać go nadając mu nazwę np. „Proba01". Otrzyma on domyślne rozszerzenie, którym może być na przykład „.c". Następnie należy znaleźć polecenie kompiluj (compile). Jeśli niczego nie zepsujesz, to po chwili powinien pojawić się komunikat o pomyślnym zakończeniu kompilacji - jest jednak mało prawdopodobne, że uda Ci się to zrobić za pierwszym razem. Jeśli pojawią się komunikaty o błędach, to dowiesz się z nich w której linijce się one znajdują. Porównaj ją wtedy z oryginałem, popraw i ponownie uruchom kompilację. Po kilku próbach osiągniesz sukces. Podczas przepisania powyższego przykładu pamiętaj o tym, że **język C odróżnia duże litery od małych liter**. ℂ

Po poprawnej kompilacji odszukaj polecenie uruchomienia programu. Możesz również odszukać skompilowany będzie on miał nazwę „Proba01.out" (w systemie

operacyjnym Unix) lub „Proba01.exe” (w środowisku DOS lub Windows) i samodzielnie go uruchomić. Na ekranie powinien pojawić się napis „Hello world.”. Może Cię jednak spotkać niespodzianka - na ekranie pojawi się na bardzo krótką chwilę okno, które natychmiast potem zniknie. Będziesz musiał wtedy trochę przerobić program. Zastąp go, na przykład, następującym programem

```
#include <stdio.h>
#include <stdlib.h>
int main()
{
  printf(''Hello world.\n'');
  system("PAUSE");
  return(0);
}
```

i ponownie powtórz opisany wyżej proces. Wywoływana funkcja „system” pochodzi z biblioteki „stdlib”.

Każdy program w języku C składa się z pewnej liczby *funkcji*. Każdy program w języku C musi posiadać jedną funkcję o nazwie *main*. Funkcja ta jest pierwszą wykonywaną funkcją programu. Nasz program jest zbudowany tylko z jednej funkcji. Każda funkcja jest zbudowana z nagłówka oraz z ciała. Nagłówek funkcji main znajduje się w pierwszej linijce. Składa się on z określenia rodzaju wyniku zwracanego przez funkcję - jest nim int, czyli liczba całkowita, nazwy - jest nią main oraz z argumentów - lista ta jest w tym przypadku pusta. Ciało funkcji znajduje się pomiędzy nawiasami klamrowymi { }. Wewnątrz ciała funkcji znajdują się wykonywane przez nią instrukcje. Pierwsza linijka definicji funkcji nazywa się prototypem lub interfacem funkcji.

Pierwsza linijka programu analizowanego programu, #include <stdio.h>, zawiera polecenie dołączenia do programu biblioteki procedur o nazwie stdio. Biblioteka ta zawiera standardowe procedury wejścia i wyjścia. W naszym programie korzystamy z jednej z nich - z funkcji printf. Funkcja ta służy do wyprowadzania tekstu do pliku wyjściowe, którym w przypadku normalnego uruchomienia programu jest ekran. Ciąg znaków ”Hello world.\n” jest przykładem *stałej napisowej*. Znaki cudzysłowu nie są jej składowymi - są tylko jej ograniczeniami. Ciąg ”\n” jest znakiem nowego wiersza.

Ostatnia linijka programu określa jaki wynik zwraca funkcja main. Liczba 0 jest sygnalizuje, że program zakończył prawidłowo swoje działanie.

Druga linijka zmodyfikowanego programu służy do zatrzymania programu. Efekt ten można osiągnąć też innymi metodami. Zależą one od systemu operacyjnego i od kompilatora. Nie będziemy w kolejnych przykładach dołączali tego typu linijki do kodu. Musisz samemu wypracować sobie najwygodniejszy sposób realizacji tego zadania.

1.2 Zmienne i wyrażenia arytmetyczne

Nasz kolejny program będzie wykonywał następujące zadanie: dla podanej wysokości wyznaczać będzie czas swobodnego spadku ciała z podanej niezbyt dużej wysokości w pobliżu powierzchni Ziemi, oraz jego prędkość końcową wyrażoną w metrach na

sekundę i w kilometrach na godzinę. Nie będziemy uwzględniali oporu powietrza. Z lekcji z fizyki pamiętasz pewnie, że droga w ruchu jednostajnie przyśpieszonym wyraża się wzorem

$$s = s_0 + v_0 \cdot t + \frac{1}{2} \cdot a \cdot t^2,$$

gdzie s_0 jest położeniem początkowym, v_0 jest prędkością początkową, a jest czasem zaś t oznacza czas. W pobliżu powierzchni ziemi przyspieszenie wynosi w przybliżeniu $g = 9.81\, m/s^2$. Ze wzoru tego bez trudu wyprowadzić możesz wzór na czas swobodnego spadku z wysokości h:

$$T = \sqrt{\frac{2 \cdot h}{g}}.$$

Poradzić sobie więc będziemy musieli z wczytaniem wysokości, wyliczeniem czasu i wyprowadzeniem danych. Oprócz wyliczenia czasu wyznaczymy prędkość w momencie uderzenia w metrach na sekundę oraz w kilometrach na godzinę. Prędkość końcową wyznaczymy ze wzoru $v = a \cdot t$. Przeliczenie prędkości wyrażonej w metrach na sekundę w kilometry na godzinę wyrazimy za pomocą wzoru

$$1\frac{m}{s} = 1\frac{0.001km}{\frac{1}{3600}h} = 3.6\frac{km}{h}.$$

Zadanie to zrealizujemy rozbijając program na kilka funkcji. Znajoma już nam funkcja main będzie zajmować się tylko wprowadzaniem i wyprowadzaniem danych. Pozostałe funkcje będą wyspecjalizowane w wykonywaniu pojedynczych zadań. Oto kod programu realizującego opisane zadanie. ℂ

```c
#include <stdio.h>
#include <math.h>

#define STALA_G 9.81

float CzasSpadku(float h)
{
  return(sqrt(2.0*h/STALA_G));
}

float Predkosc(float h)
{
  return(STALA_G*CzasSpadku(h));
}

float Vkmh(float v)
{
  return(3.6*v);
}

int main()
{
  float w;

  printf(``Podaj wysokosc w metrach : ``);
  scanf (``%f``,&w);
```

```
  printf(``Wysokosc : %0.2f [m]\n'', w);
  printf(``Czas : %0.2f [s]\n'', CzasSpadku(w));
  printf(``Predkosc : %0.2f [m/s]\n'', Predkosc(w));
  printf(``Predkosc : %0.2f [km/h]\n'',Vkmh(Predkosc(w)));
  return(0);
}
```

Program rozpoczęliśmy od dołączenia biblioteki math. Znajduje się w niej szereg funkcji matematycznych. W programie korzystamy z jednej z nich - z funkcji sqrt, która oblicza pierwiastek kwadratowy. Wykorzystaliśmy ją w funkcji o nazwie CzasSpadku.

W kolejnej linijce programu zdefiniowaliśmy stałą o nazwie STALA_G. Zrobiliśmy to, gdyż ze stałej tej korzystamy w dwóch miejscach programu. Gdybyśmy chcieli stałą g zastąpić przyśpieszeniem na powierzchni Księżyca, to wystarczyłoby to zrobić w jednym miejscu programu.

Funkcja CzasSpadku jest funkcją typu float, czyli zwraca wynik typu float - służy on do reprezentowania liczb rzeczywistych. Jest ona funkcją jednej zmiennej typu float, która nazwaliśmy h. Ciało tej funkcji składa się tylko z jednej instrukcji:

```
return(sqrt(2.0*h/STALA_G));
```

Ciąg 2.0*h/STALA_G jest wyrażeniem arytmetycznym. Znaczenie tego wyrażenia jest chyba jasne. Występuje w nim *zmienna* h, która jest zmienną typu float. Zmienne wyobrażać sobie możesz jako podpisane pudełka, w których przechowywane są dane. „Pudełko" h ma nazwę „h" i przechowuje w środku liczbę rzeczywistą, którą w tym miejscu interpretujemy jako „wysokość". Wyrażenie 2.0*h/STALA_G zostało przekazane do funkcji sqrt. Wynikiem funkcji jest wartość wyrażenia sqrt(2.0*h/STALA_G). Zwróć uwagę na to, że instrukcja return(sqrt(2.0*h/STALA_G)); zakończona jest średnikiem. **W języku C średnik służy do kończenia instrukcji.** ℂ

Druga z funkcji, o nazwie Predkosc korzysta z funkcji CzasSpadku. Trzecia funkcja - ta o nazwie Vkmh - służy do przeliczenia prędkości wyrażonej w metrach na sekundę na kilometry na godzinę.

Ciało funkcji main rozpoczyna się od zadeklarowania zmiennej typu float o nazwie „w". Pierwsza instrukcja programu jest już nam znana. Druga instrukcja

```
scanf ("%f", &w);
```

służy do pobrania i podstawieniu pod zmienną w wysokości. Zwróć uwagę na to, że nazwę zmiennej poprzedziliśmy znakiem &. Znak ten ma bardzo ważne znaczenie w języku C, oznacza on tak zwany **wskaźnik** na obiekt. Nie będziemy się tym teraz zajmowali. Przyjmijmy na razie, po prostu, że zmienne przekazywane funkcji scanf muszą być poprzedzone znakiem &. Pierwszy parametr funkcji scanf jest stałą napisową "%f". Ciąg %f informuje kompilator o tym, że wczytywana liczba jest typu float.

W kolejnych instrukcjach wyświetlamy wyniki. Wykorzystujemy w nich ponownie funkcję printf, lecz tym razem przekazujemy do niej wartości zmiennych bądź wyrażeń. Fragment %0.2f występujący w jej pierwszym parametrze służy do określenia formatu wydruku. Oznacza on: „*wyświetl przekazywaną wartość, traktowaną jako float, z dokładności do dwóch miejsc po przecinku*".

Napisany program ma pewną wartość użytkową. Możesz z niego skorzystać aby wyznaczyć stosunkowo bezpieczną wysokość skoku. Musisz wiedzieć, że przy zderzeniu ze sztywnym, ciężkim ciałem z prędkości 20 km/h możesz się nieźle połamać.

W programie posługiwaliśmy się zmiennymi typu float. Podstawowymi typami języka C są

char, int, float, double, pointer

Typ char reprezentuje pojedynczy znak. Reprezentowane są one przez liczby całkowite. Na przykład, litera 'A' ma kod 65, litera 'B' ma kod 66, spacja mo kod 32 itd. Stosowana już przez nas stała '\n' oznacza znak nowego wiersza i ma kod 10. Pełen wykaz kodów znaków znajduje się w tabeli kodów ASCII (American Standard Code for Information Interchange). Pierwsze 127 kodów zawiera 26 małych liter alfabetu łacińskiego, 26 dużych liter alfabetu łacińskiego, 10 cyfr, spację, znaki specjalne, np. ! # $ % & oraz znaki sterujące (kody ASCII od 0 do 31), np. przejdź do nowego wiersza (oznaczenie LF od Line Feed), powrót karetki do początku wiersza (CR, od słów Carriage Return), tabulator, koniec tekstu (EOT, od słów End of Text). Kody ASCII powyżej 127 to tzw. zestaw rozszerzony; zapisuje się w nim znaki narodowe i znaki semigrafiki (symbole, pozwalające tworzyć na ekranie ramki itp.)

Typ int oraz jego warianty short int, long int, unsigned int, unsigned short int i unsigned long int służą do reprezentowania liczb całkowitych. Słowo unsigned oznacza „bez znaku". Zatem, jak łatwo się domyśleć, warianty „unsigned" służą do reprezentowania podzbioru liczb naturalnych.

Typy (float) i (double) służą do reprezentowania liczb rzeczywistych. Typ (double) zapewnia większą dokładność niż typ (float). Typ pointer, czyli wskaźniki, omówimy w dalszej części tego rozdziału.

Do konstruowania wyrażeń arytmetycznych korzystać możemy z operatorów +, - ,*, / oraz %. Znaczenie czterech pierwszy operatorów jest chyba jasne - jest to dodawanie, odejmowanie, mnożenie i dzielenie. Dzielenie zastosowane do zmiennych typu całkowitego obcina część ułamkową ilorazu. Operator % stosowany może być do typów całkowitych i oznacza dzielenie modulo. Tak więc a % b oznacza resztę z dzielenia liczby a przez liczbę b, czyli

$$(x = a\%b) \leftrightarrow (0 \leq x < b) \wedge (\exists k \in \mathbb{Z})(a = b \cdot k + x).$$

Na przykład $12\%10 = 2$, $12\%5 = 2$ i $12\%6 = 0$. W dalszych przykładach często będziemy korzystali z tego, że jeśli x i y są zmiennymi całkowitymi, oraz y>0, to wtedy prawdziwa jest równość

x = (x / y) * y + (x % y).

Wyrażenia arytmetyczne służyć mogą do podstawiania wartości pod zmienne. Do tego celu służy operacja przypisania, którym jest reprezentowany przez znak =. Zwróć uwagę na to, że znak = nie oznacza równości lecz operację przypisania.

Uwaga. W języku programowania Pascal podstawienie jest reprezentowane przez ciąg :=. Znak równości oznacza w nim to co powinien oznaczać, czyli równość.

Warto w tym miejscu zwrócić uwagę na pewną subtelność języka C, która wynika z powyższych rozważań. Otóż instrukcja

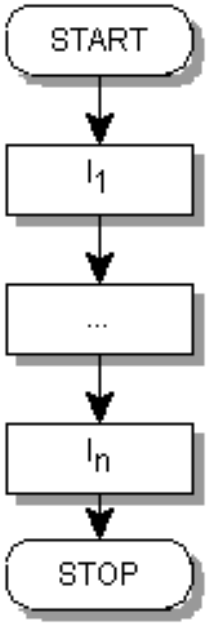
```
x = 9/5;
```

podstawi pod zmienną x, nawet jeśli jest ona typu double wartość 1, gdyż część całkowita liczby 9/5 wynosi 1, a stałe 5 i 9 są całkowite. Aby pod zmienną x podstawić 1.8 należy skorzystać z instrukcji

```
x = 9.0/5.0;
```

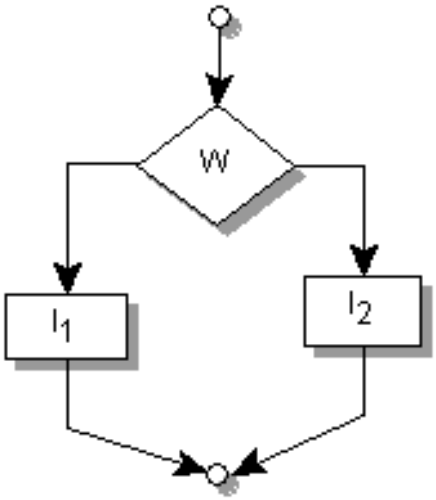
Możesz też, oczywiście, napisać to tak: `x = 1.8;`.

1.3 Konstrukcje warunkowe



Rozważane do tej pory przez nas programy miały ustaloną z góry kolejność wykonywania instrukcji. Można je przedstawić za pomocą schematu blokowego znajdującego się po prawej stronie tekstu. Owale z napisami „START" i „STOP" oznaczają początek i koniec programu. W prostokątach znajdują się instrukcje. Strzałki służą do określenia kolejności wykonywania instrukcji.

Programy tego typu są mało elastyczne, gdyż zarówno lista instrukcji jak też i kolejność ich wykonywania jest ustalona i nie zależy od danych wejściowych. Na szczęście język C, jak większość języków programowania, posiada mechanizmy do warunkowego wykonywania instrukcji, czyli do wykonywania pewnych instrukcji wtedy, gdy będą spełnione pewne, ściśle określone, warunki.



Język C posiada dwie konstrukcje warunkowe. Podstawowa konstrukcja ma postać

if (W) I_1 else I_2

gdzie W jest wyrażeniem zaś I_1 oraz I_2 są instrukcjami. Jeśli wyrażenie W jest prawdziwe, to wykonywana jest instrukcja I_1. W przeciwnym wypadku wykonywana jest instrukcja I_2.

Wyrażenie języka C jest prawdziwe jeśli jego wartość jest różna od zera !
Oznacza to zaś, że język C traktuje liczbę zero jako wartość logiczną „**fałsz**", zaś dowolną inną liczbę jako „**prawdę**".

Druga część instrukcji **if-else** jest opcjonalna. Może więc ona przyjmować postać if (W) then I. Do budowania wyrażeń testowanych w wyrażeniach warunkowych korzystać można z relacji pomiędzy wyrażeniami oraz z operatorów logicznych.

Zwróć uwagę na to, że symbol „=" jest symbolem przypisania. Nie stosuj go do testowania równości. Instrukcja if (x=2) I; będzie zawsze wykonana, bo wyrażenie

Relacja/Operator	Znaczenie
<	mniejszy
<=	mniejszy lub równy
>	większy
>=	większy lub równy
==	równość
!=	negacja równości
!	negacja
&&	koniunkcja
||	alternatywa

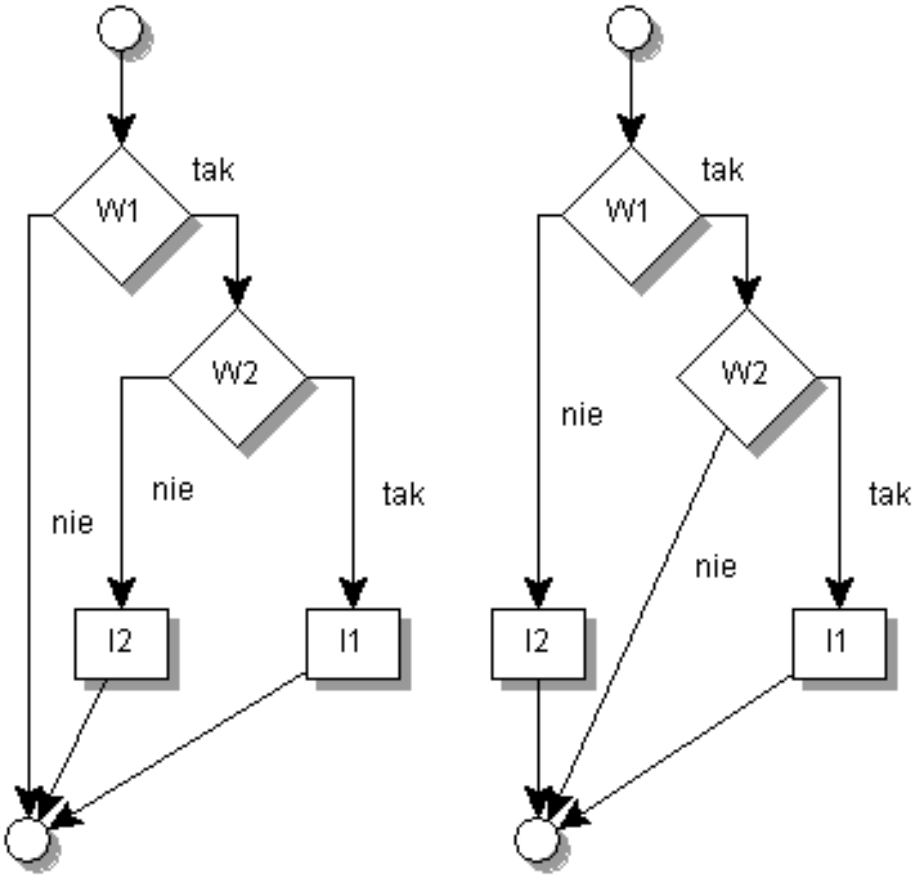
Tablica 1.1: Relacje i operatory logiczne

x=2 zwraca wartość 2, a więc wartość różną od zera. Wykonanie instrukcji if (x==2) I; spowoduje uruchomienie instrukcji I wtedy i tylko wtedy, gdy zmienna będzie miała wartość 2.

Interpretacja konstrukcji warunkowych postaci if (W) then I oraz if (W) then I_1 else $I2$ nie sprawia żadnego kłopotu. Jednakże z tego powodu, że część „else" instrukcji **if-else** nie jest obowiązkowa, w przypadku zagnieżdżonych instrukcji warunkowych mogą pojawić się dwuznaczności. Na przykład, konstrukcję

```
if (W1)
if (W2) I2
else I3
```

można by zinterpretować na dwa następujące sposoby:



Niejednoznaczność tę język ANSI C rozstrzyga następująco: **każda z części else jest przyporządkowana najbliższej z poprzednich instrukcji if nie zawierającej części else.** W analizowanym przykładzie wybrany zostanie więc pierwszy wariant do interpretacji konstrukcji. Mówiąc inaczej, konstrukcja

```
if (W1) if (W2) I1 else I2
```

jest równoważna konstrukcji

```
if (W1) {if (W2) I1 else I2}
```

Jeśli będziesz miał wątpliwości w jakiej kolejności wykonywane są obliczenie, to **pamiętaj, że lepiej dopisać kilka zbędnych nawiasów, niż popełnić błąd.**

Proste przykłady

Omówimy teraz kilka przykładów zastosowania omówionych instrukcji warunkowych.

Przykład 1.1 *Pierwszym przykładem jest funkcja służąca do wyznaczania wartości bezwzględnej danej liczby typu double.*

```
double WartBezwzgledna(double x)
{
  if (x>=0) return(x);
  else return(-x);
}
```

Przykład 1.2 *Drugim przykładem jest funkcją wyznaczającą znak podanej liczby typu double. Zwraca ona wartość* $+1$ *dla liczb dodatnich, zero dla liczby równej zero oraz* -1 *dla liczby ujemnej.*

```
int Znak(double x)
{
  if (x>0) return(1);
  else if (x==0) return(0);
  else return(-1);
}
```

Przykład 1.3 *Jako kolejny przykład rozważymy funkcję, która ustala, czy dany rok jest rokiem przestępnym, czy też normalnym. Przypomnijmy sobie, że zgodnie z używanym obecnie kalendarzem gregoriańskim rok jest przestępny, jeśli dzieli się przez 4, lecz nie dzieli się przez 100, chyba, że dzieli się przez 400. Inaczej mówiąc, rok r jest przestępny, jeśli*

$$((4|r) \wedge \neg(100|r)) \vee (400|r),$$

gdzie symbol $|$ *reprezentuje relację podzielności (bez reszty),* $\wedge$ *oznacza koniunkcję,* $\vee$ *alternatywę a* $\neg$ *oznacza negację.*

```
int Przestepny(int r)
{
  return( ((r%4 == 0)&&(r%100 != 0)) || (r%400 == 0));
}
```

Funkcja „Przestepny" zwraca wartość 1 jeśli rok jest przestępny i wartość 0 w przeciwnym przypadku. Reszta fragmentu kodu powinna być dla Ciebie oczywista.

Warto wiedzieć, że korzystają ze znajomości priorytetów operatorów języka C wyrażenie

```
( (r % 4 == 0) && (r % 100 != 0) ) || (r % 400 == 0)
```

można zapisać nieco prościej:

```
r % 4 == 0 && r % 100 != 0 || r % 400 == 0
```

Wynika to z tego, że priorytet operacji $\%$ jest większy od priorytetów relacji $==$ oraz $!=$, który jest większy od priorytetu operatora $\&\&$, który jest wyższy od priorytetu operacji $||$:

$$"\%" > "==","!=" > "\&\&" > "||"$$

Im większy jest zaś priorytet relacji, tym ma on większą siłę łączenia. Pamiętaj, że jeśli będziesz miał wątpliwości w jakiej kolejności będzie wykonywane obliczanie wyrażenia, to stosuj nawiasy. Ponownie przypomnijmy o zasadnie, że **lepiej dopisać kilka zbędnych nawiasów, niż popełnić błąd.**

Język C posiada jeszcze jedną postać instrukcji warunkowej. Jest to instrukcja switch. Ma ona postać

```
switch (W)
{
 case W1: I1
 ...
 case Wn: In
 default: J
}
```

W konstrukcji tej W musi być wyrażeniem przyjmującym wartości całkowite lub znakowe. Wyrażenia W1, ldots, Wn muszą być wyrażeniami stałymi. Fragment default: I jest opcjonalny - nie musi występować. Wykonanie tej konstrukcji jest równoważne wykonaniu następującego fragmentu kodu:

```
Rob = W;
if (Rob=W1) I1
...
if (Rob=Wn) In
if (!(W1 || ...||Wn)) J
```

Bardzo często stosowaną instrukcją wewnątrz instrukcji switch jest instrukcja break. Służy ona do natychmiastowego opuszczenia instrukcji switch. Instrukcja break jest również często wykorzystywana wewnątrz pętli, które omówimy w dalszej części tego rozdziału.

Przykład 1.4 *Następujący fragment kodu służy do wykonywania różnego rodzaju obliczeń - sumy, różnicy, iloczynu, ilorazu - w zależności od tego jaką wartość przyjmuje zmienna* operacja

```
switch (operacja)
{
  case '+' : z = x + y; break;
  case '-' : z = x - y; break;
  case '*' : z = x * y; break;
  case '/' : z = x / y; break;
}
```

1.4 Parametry funkcji i wskaźniki

Język C posiada tylko jeden mechanizm przekazywania zmiennych do wnętrza funkcji: **wewnątrz funkcji pracuje się tylko na kopiach zmiennych**. Metoda nazywa się przekazywaniem wartości przez wartość. A oznacza to, między innymi, że wewnątrz funkcji możesz bezkarnie zmieniać wartości zmiennych przekazanych do funkcji - po wyjściu z funkcji (czyli po zakończeniu jej działania) wartości przekazanych zmiennych nie ulegną zmianie.

Uwaga. Warto wiedzieć, że inne języki programowania, na przykład Pascal lub C++, posiadają więcej metod przekazywania parametrów.

Wewnątrz ciała funkcji możesz powoływać do życia nowe zmienne. Nazywają się one *zmiennymi lokalnymi*. Widoczne są tylko wewnątrz ciała funkcji. Pozostałe zmienne, a więc te które nie są zdefiniowane wewnątrz ciała funkcji są widoczne w całym pliku od momentu powołania ich do życia.

W wielu sytuacjach wygodne jest aby pisane przez Ciebie funkcje zmieniały wartości przekazywanych do niej parametrów. Załóżmy, na przykład, że chcesz napisać funkcję swapint, której celem będzie zamiana wartości dwóch przekazanych jej zmiennych typu integer. **Rozwiązanie**

```
void swapint(int x, int y) // ZŁE
{
  int rob;
  rob = x;
  x = y;
  y = rob;
}
```

jest błędne, gdyż wewnątrz funkcji swapint pracujemy z kopiami zmiennych. Aby osiągnąć ten cel musimy skorzystać z innego mechanizmu języka C. Są nimi wspomniane już wcześniej **wskaźniki**.

Definicja 1.1 ***Wskaźnikiem*** *nazywamy zmienną zawierającą informacje o położeniu innej zmiennej.*

Jak już wspomnieliśmy zmienną możemy interpretować jako pudełko w nazwą. Wskaźnik jest więc pudełkiem który zawiera adres innego pudełka. Rozmiar tego pudełka jest na tyle duży aby móc pomieścić informacje o położeniu dowolnego innego pudełka (czytaj - zmiennej) z którego korzystamy w programie języka C.

Do wyznaczenia adresu zmiennej służy jednoargumentowy operator &. Jeśli p jest wskaźnikiem na zmienną typu int a x jest zmienną typu int, to instrukcja

p = &x;

przypisuje zmiennej p adres zmiennej x. Zmienne wskaźnikowe deklarujemy za pomocą operatora *. Oto przykład:

```
int x = 5, y = 11;
int *px, *py;
px = &x;
py = &y;
printf(``%d %d'',*px,*py);
py = px;
printf(``%d %d'',*px,*py);
```

W pierwszej linijce zdefiniowaliśmy dwie zmienne x i y typu int. Zmiennej x nadaliśmy wartość 5 a zmiennej y wartość 11; W drugiej linijce zdefiniowaliśmy dwie zmienne px i py typu „wskaźnik” na int. W trzeciej linijce zmiennej px przypisaliśmy adres zmiennej x a W czwartej linijce zmiennej py przypisaliśmy adres zmiennej y. W linijce piątek drukujemy zawartości zmiennych typu int wskazywane przez px i py. Skorzystaliśmy tutaj z jednoargumetowego operatora *, zwanego *adresowaniem pośrednim*, który zastosowany do wskaźnika zwraca zawartość obiektu na który on wskazuje. W tym momencie wyprowadzony zostanie napis ”5 11”. W szóstej linijce pod zmienną py podstawiliśmy wartość zmiennej px. Ponieważ px wskazywał na x, więc od tej pory również py wskazywać będzie na x. Polecenie z ostatniej linijki wyprowadzi zatem napis "5 5".

Oto poprawna wersja funkcji swapint, która służy do zamiany wartości dwóch zmiennych typu int:

```
void swapint(int *x, int *y)
{
  int rob;
  rob = *x;
  *x = *y;
  *y = rob;
}
```

Deklaracja void służy do określenia, że funkcja ta nie zwraca żadnej wartości. Z funkcji tej korzystać możemy w następujący sposób

swapint(&a,&b);

1.5 Pętle

Klasa programów zbudowanych z wyrażeń arytmetycznych, instrukcji przypisania oraz instrukcji warunkowych jest stosunkowo mała. Za ich pomocą nie jesteśmy w stanie napisać programów których długość działania w sposób istotny zależy od wprowadzanych danych.

Język C posiada trzy konstrukcje pozwalające na wykonywanie zapętlonych obliczeń. Są to pętle **while**, **do-while** oraz **for**.

Pierwsza pętla nazywa się pętlą while. Ma ona postać

while (W) I

gdzie W jest wyrażeniem zaś I jest dowolną instrukcją. Oto jej schemat blokowy:

Wykonywanie instrukcji I pętli while (W) I wykonuje się tak długo, jak długo wyrażenie W przyjmuje wartość różną od zera. **Język programowania który posiada konstrukcje podstawiania, instrukcji warunkowych oraz pętli omówionego typu posiada uniwersalną moc obliczeniową.** Uwagą tą zajmiemy się bardziej szczegółowo w ostatnim rozdziale tej książki.

Przykład 1.5 *Oto jak dla danej liczby C wyznaczyć możemy najmniejszą liczbę naturalną n taką, że* $1^2 + 2^2 + \ldots + n^2 \geq C$*:*

```
int n=0,suma=0;
while (suma<C)
{
  n=n+1
  suma = suma + n*n;
}
```

Język C posiada metodę bardziej zwartego zapisu operacji zwiększania zmiennej typu całkowitego (oraz typu wskaźnikowego) o jeden. Do tego celu służą instrukcje x++ oraz ++x. Pierwsza z tych instrukcji służy do zwiększenia wartości zmiennej przed jej użyciem a druga po jej użyciu. Zwiększenie wartości zmiennej x o wartość y, czyli instrukcję x = x + y, można w bardziej zwartej postaci zapisać w postaci x += y. Korzystając z powyższych mechanizmów języka C pętlę z ostatniego przykładu można zapisać w sposób bardziej zwarty:

while (suma<C){n++; suma += n*n}

Przykład 1.6 *Stosunkowo łatwo można pokazać, że dla dowolnej liczby rzeczywistej* $a > 0$ *ciąg* $(x_n)_{n\in\mathbb{N}}$ *zadany wzorami* $x_0 = a$, $x_{n+1} = \frac{1}{2}(x_n + a/x_n)$ *jest zbieżny do liczby* $\sqrt{a}$ *(patrz zadanie 1.3). Obserwację tę łatwo można zamienić na następujący algorytm obliczania pierwiastka:*

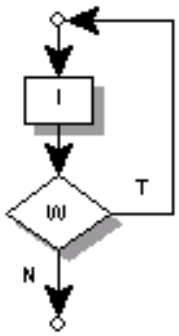
```
double x;
x = a;
while (fabs(a-x*x)>EPSILON)
  x = (x + a/x)/2;
```

gdzie stała EPSILON *służy do kontrolowania dokładności przybliżenia. Funkcja* fabs *pochodzi z biblioteki* math.h *i służy do wyznaczania wartości bezwzględnej argumentu typu double. Algorytm ten należy traktować tylko jako dydaktyczny przykład, gdyż w bibliotece math.h znajduje się funkcja* sqrt*. Jedna z zasad skutecznego programowania brzmi:* ***nie oprogramowuj funkcji bibliotecznych****.*

Drugi rodzaj pętli, zwany pętlą do-while, ma postać

do I while (W)

gdzie W, jak poprzednio, jest wyrażeniem a I instrukcją. Oto jej schemat blokowy:



Różnica pomiędzy pierwszą a drugą formą pętli polega na tym, że instrukcja I pętli „do I while (W)" wykonywana jet co najmniej raz. Testowanie warunku W, czyli sprawdzanie, czy przyjmuje on wartość różną od zera, odbywa się po każdym wykonaniu instrukcji I.

Trzecia wersja pętli, najczęściej wykorzystywana w praktyce, ma postać

for (W1;W2;W3) I

gdzie W1, W2 i W3 są wyrażeniami. Jej wykonanie jest równoważne wykonaniu sekwencji

W1; while (W2){I; W3}

Konstrukcja for wykorzystywana jest najczęściej do wykonywania pętli, których długość jest znana przed jej rozpoczęciem. Oto bardzo typowy przykład użycia tej konstrukcji.

Przykład 1.7 *Następujący fragment kodu wyznacza sumę* $\sum_{i=1}^{100} \frac{1}{i^2}$*:*

```
int i;
double suma=0;
for (i=1;i<=100;i++)
 suma += 1.0/(i*i);
```

Przykład 1.8 *Załóżmy, że masz zbadać zbieżność ciągu zadanego wzorem* $a_0 = \sqrt{6}$, $a_{n+1} = \sqrt{6 + a_n}$ *i, że z jakiegoś bardzo tajemniczego powodu nie wiesz jak się do tego zabrać. Następujący program pozwoli Ci postawić rozsądną hipotezę:*

```
#include <stdio.h>
#include <math.h>
int main()
{
   int i;
   float x;
   x=sqrt(6.0);
   for (i=0;i<=20;i++)
   {
      printf(" %d %f \n",i,x);
      x = sqrt(6 + x);
}
return(0);
}
```

Pętle nadają językowi programowania pełną moc programistyczną. Tę siłę trzeba jednak opłacić dużym kosztem. Programy z pętlami są czasami bardzo trudne do zrozumienia.

Przykład 1.9 *Rozważmy następujący zbiór przekształceń liczb naturalnych większych od zera: jeśli* $n = 1$*, to zatrzymaj się, jeśli* n *jest podzielne przez dwa, to podziel ją przez 2, jeśli zaś* $n > 1$ *i nie jest podzielne przez 2, to pomnóż ją przez 3 i dodaj 1. Reguły te możemy zapisać za pomocą następującej pętli języka C:*

```
while (n>1)
  if (n % 2 == 1)
    n = 3*n + 1;
  else
    n = n / 2;
```

lub w postaci bardziej zwartej:

```
while (n>1) if (n % 2) n = 3*n+1; else n /= 2;
```

Do tej pory nie wiadomo, czy pętla ta zatrzymuje się po skończonej liczbie kroków dla dowolnej liczby naturalnej n!

W ostatnim przykładzie zastosowaliśmy konstrukcję `n /= 2`, która jest (logicznie) równoważna konstrukcji `n = n / 2`.

1.6 Tablice i łańcuchy

Wiele algorytmów służy do przetwarzania dużej liczby danych tego samego typu. Do zapamiętywania dużej liczby zmiennych w programach w języku C tego samego typu służą tablice. Tworzymy je za pomocą deklaracji postaci

typ nazwa[zakres];

Na przykład, aby utworzyć tablicę 100 zmiennych typu int o nazwie liczby stosujemy deklarację

int liczby[100];

Numeracja elementów tablic w języku C zawsze zaczyna się od zera. W naszym przykładzie odwoływać się się możemy do elementów tablicy o numerach 0, 1, ..., 99. Do elementu tablicy liczby o numerze i odwołujemy się za pomocą konstrukcji liczby[i].

Przykład 1.10 *Ciąg Fibbonacciego definiujemy za pomocą następujących wzorów:* $F_0 = 1$, $F_1 = 1$, $F_{n+2} = F_n + F_{n+1}$. *Oto fragment programu, który wyznacza jego pierwsze 40 wyrazów:*

```
int i;
int Fibb[40];
Fibb[0] = 1;
Fibb[1] = 1;
for (i=2;i<40;i++)
  Fibb[i] = Fibb[i-2]+Fibb[i-1];
```

Tablice, podobnie jak zmienne, można inicjalizować w momencie deklaracji. Oto przykład

int x[10] = {9,8,7,6,5,4,3,2,1,0};

inicjalizacji tablicy dziesięciu liczb całowitych i postawieniu pod kolejne elementy x[0], x[1], ..., x[9] wartości 9, 8, ..., 0.

Przykład 1.11 *Następujący fragment kodu możesz użyć do podstawienia pod elementy tablicy liczb całkowitych* x *wartości losowych ze zbioru* $\{0, \ldots, 99\}$*:*

```
for (k=0;k<100;k++)
  x[k] = rand() % 100;
```

Tablice do funkcji przekazywać możemy za pomocą konstrukcji typ nazwa[]. Pamiętać musisz również o tym aby funkcja znała rozmiar tablicy. Standardowe rozwiązania polega na przekazaniu rozmiaru tablicy jako drugi parametr.

Przykład 1.12 *Następująca funkcja oblicza sumę elementów przekazanej jej tablicy liczb typu float:*

```
double SumujTabFloat(float t[], int rozmiar)
{
  int i;
  double su = 0;
  for (i = 0; i < rozmiar; i++)
    su += t[i];
  return(su);
}
```

Język C traktuje tablice jako wskaźnik do pierwszego elementu. Oznacza to, między innymi, że jeśli tablica zostanie przekazana do funkcji jako jej parametr, to wewnątrz funkcji pracować będziemy z oryginałem tablicy a nie z jej kopią !

Przykład 1.13 *Korzystając z traktowania przez język C tablicy tako wskaźnika do jej pierwszego elementu możemy napisać funkcję do zerowania zawartości tablicy:*

```
void ZerujTabInt(int t[], int rozmiar)
{
  int i;
  for (i = 0; i < rozmiar; i++)
    t[i] = 0;
}
```

Język C traktuje napisy nieco nonszalancko: są to tablice znaków zakończone znakiem o kodzie 0. Znak o kodzie zero jest kodowany jako '\0'. Napis „ala ma kota" jest więc reprezentowany jako tablica 12 znaków. W poniższej tabelce w pierwszym wierszu znajdują się znaki, a w drugim wierszu odpowiadające im kody:

	0	1	2	3	4	5	6	7	8	9	10	11
ZNAK	a	l	a		m	a		k	o	t	a	\0
ASCII	97	108	97	32	109	97	32	107	111	116	97	00

W tablicy kodów ASCII znak a ma kod 97, znak l ma kod 108, spacja ma kod 32, itd.

W języku C nie ma wyspecjalizowanych operacji do obsługi napisów. Na szczęście w bibliotece string.h istnieje kilka pożytecznych funkcji. Oto, na przykład, jak można by samodzielnie napisać funkcję wyznaczającą długość napisu:

```
int DlugoscNapisu(char s[])
{
  int i=0;
  while (s[i] != '\0')
    i++;
  return(i)
}
```

Funkcji tej nie musisz samodzielnie pisać. W wymienionej bibliotece string.h istnieje funkcja strlen, które to zadanie wykonuje za Ciebie. Skorzystamy z niej w następnym przykładzie.

Przykład 1.14 *bibliotece string.h nie ma funkcji do odwracania łańcucha. A funkcja taka przyda nam się w dalszych przykładach. Oto kod takiej funkcji*

```
void strev(char s[])
{
  int l=0,p;
  char c;
  p = strlen(s)-1;
  while (l<p)
  {
    c    = s[l];
    s[l] = s[p];
    s[p] = c;
```

```
        l++;
        p--;
    }
}
```

Kod tej funkcji można nieco uprościć korzystając zanurzając konstrukcje przyrostowe ++ oraz − w instrukcje podstawienia. Można ją nieco przyśpieszyć, posługując się wskaźnikami.

1.7 Biblioteki

Język C posiada kilka standardowych bibliotek, które niezwykle ułatwiają i przyśpieszają programowanie. Wymienimy tutaj tylko część z nich.

stdio podstawowa biblioteka operacji wejścia i wyjścia (standardowa dla C). Zawiera stosowane przez nas funkcje printf oraz scanf oraz szereg funkcji służących do obsługi plików, czyli do ich zakładania, otwierania, modyfikowania, zamykania, usuwania itd.

math zawiera podstawowe funkcje matematyczne takie jak sinus (sin), cosinus (cos), tangens (tan), logarytm (log, log10), potęgowanie (pow), pierwiastek kwadratowy (sqrt).

string zawiera podstawowe funkcje do operowania na ciągach znaków (czyli łańcuchach) – służące do kopiowania, wyszukiwania podciągów itp. W szczególności zawiera funkcję strlen wyznaczającą długość łańcucha

ctype zawiera funkcje do testowania czy dany znak jest znakiem, cyfrą, czy jest dużą, czy też małą literą, itd.

limits zawiera szereg stałych określających zakresy podstawowych typów danych takich jak unsigned char, char, int, specyficznych dla maszyny.

stdlib zawiera szereg funkcji służących do przekształcania liczb, np. atof - przekształcenie łańcucha na typ float, atoi - przekształcenie łańcucha na typ int, rand - kiepski generator liczb losowych, funkcje do przydzielania i zwalniania pamięci oraz funkcję system, z której w wersji system(”PAUSE”) korzystaliśmy do zatrzymania działania programu. numbers.

Funkcji równoważnych funkcją znajdujących się w standardowych bibliotekach nie należy samodzielnie pisać - chyba, że w celu nabrania wprawy, lub, że masz ku temu naprawdę poważny powód. W innym przypadku samodzielne oprogramowania funkcji bibliotecznej traktuj jako błąd. Zapoznaj się więc starannie ze standardowymi bibliotekami.

1.8 O stylu

Program wyznaczający czas spadku z danej wysokości (patrz punkt 1.2) można by zredagować następująco:

```
#include <stdio.h>
#include <math.h>
#define SG 9.81
float CS(float h){return(sqrt(2.0*h/SG));}
float Pr(float h){return(SG*CS(h));}
float Vk(float v){return(3.6*v);}
int main(){float w; printf(``Podaj wysokosc w metrach : ``);
scanf (``%f'', &w);
printf(``Wysokosc : %0.2f [m]\n''    ,w);
printf(``Czas     : %0.2f [s]\n''    ,CS(w));
printf(``Predkosc : %0.2f [m/s]\n''  ,Pr(w));
printf(``Predkosc : %0.2f [km/h]\n'',Vk(Pr(w)));}
```

Jest on bardzo trudny do przeczytania. Nie jest dobrze zredagowany - trudno się zorientować gdzie zaczynają się i kończą ciała funkcji. Nazwy funkcji zostały głupio wymyślone. Nazwa stałej SG nic nie mówi, a jest to przecież stała grawitacji.

Pisząc programy pisz je dla drugiego programisty - twój kolega ma je zrozumieć bez trudu. Stosuj rozsądne nazwy dla zmiennych i funkcji. Stosuj konsekwentnie wcięcia - ułatwiają one znacznie czytanie kodu. Zmienne globalne, czyli te które nie występują w ciele funkcji, powinny mieć zrozumiałe nazwy. Zmienne lokalne, takie jak liczniki pętli, mogą mieć nazwy krótkie. Staraj się umieszczać jedno polecenie w linii.

Problemowi stylu programowania poświęcimy cały oddzielny rozdział.

1.9 Ćwiczenia i zadania

Ćwiczenie 1.1 *Napisz program typu „ascii-art", który generuje mniej więcej takie rysunki różnych postaci:*

```
                                    ,.,
  /"""\       /"""\       /"""\     /"""\
  |o o|       |O O|       |o o|     |@ @|
  \ - /       \ U /       \ ~ /     \ = /
   |_|         |_|         |_|       |_|
normalny szczęśliwy  zakochany  po sesji
```

Ćwiczenie 1.2 *Napisz program, który wczytuje dwie liczby rzeczywiste i wyprowadza ich sumę, różnicę, iloczyn i iloraz. Powtarzaj to ćwiczenie tak długo, aż potrafisz samodzielnie napisać ten program, bez korzystania z żadnych materiałów pomocniczych.*

Ćwiczenie 1.3 *Napisz program który przekształca temperaturę podaną w stopniach Celsjusza na temperaturę w stopniach Farenheita. Związek pomiędzy temperaturą wyrażoną w stopniach Celsjusza i stopniach Farenheita wyraża się wzorem* $C = 1.8 \cdot F + 32$.

Ćwiczenie 1.4 *Napisz program wczytuje podany rok R i wyświetla, zgodnie z prawdą, napis „Rok R jest przestepny" lub „Rok R jest normalny".*

Ćwiczenie 1.5 *Zakładając, że od początku naszej ery obowiązywał kalendarz gregoriański (co, prawdę mówiąc, nie jest prawdą), napisz funkcję, która wylicza ile dni minęło od początku ery, do podanej daty. Napisz program, który obliczy ile dni minęło od daty Twoich urodzin do dnia dzisiejszego. Przelicz to na sekundy.*

Ćwiczenie 1.6 *Dla jakich liczb naturalnych n prawdziwa jest nierówność*

$$1 + \frac{1}{2} + \frac{1}{3} + \ldots + \frac{1}{n} > 10?$$

Ćwiczenie 1.7 *Dla danej liczby rzeczywistej $a > 0$ definiujemy ciąg*

$$u_n = \sqrt{n + a\sqrt{n} + 1} - \sqrt{n}.$$

Napisz program, który dla kilku różnych wartości a, na przykład dla a = 0.5, 1, 2 i 4, aproksymuje granicę tego ciągu. Postaw hipotezę o granicy i następnie ją udowodnij.

Ćwiczenie 1.8 *Przeprowadź badanie zbieżności pętli*

```
while (n>1) if (n % 2) n = 3*n+1; else n /= 2;
```

Sprawdź liczby kroków którą wykonuje ta pętla dla wszystkich liczb naturalnych $n \leq 1000$.

Ćwiczenie 1.9 *a. Średnią arytmetyczną ciągu liczb $(a_i)_{i=1\ldots n}$ nazywamy liczbę $E = (\sum_{i=1}^{n} a_i)/n$. Napisz funkcję wyznaczającą średnią z przekazanej jej tablicy liczb typu float.*
b. Standardowy odchyleniem ciągu liczb $(a_i)_{i=1\ldots n}$ nazywamy liczbę

$$S = \sqrt{\frac{\sum_{i=1}^{n}(a_i - E)^2}{n}},$$

gdzie E oznacza średnią arytmetyczną. Napisz funkcję wyznaczającą standardowe odchylenie tablicy liczb typu float.

Ćwiczenie 1.10 *Wyznacz stosunkowo dobre przybliżenie liczby $\sqrt[1000]{1000!}$ - możesz założyć, że arytmetyka liczb typu double zapewni Ci odpowiednią dokładność.*

Ćwiczenie 1.11 *Napis nazywa się palindromem, jeśli czytany od przodu i czytanym od tyłu jest taki sam, na przykład: „kajak", „zakaz", „oko", „radar", „potop", „ara", „oko w oko". Napisz funkcję, który sprawdza, czy dany łańcuch jest palidromem.*

Ćwiczenie 1.12 *Napis funkcję, która przekształca wszystkie litery w przekazanym jej łańcuchu do dużych liter. Skorzystaj z tego, że litery od 'a' do 'z' (oraz od 'A' do 'Z') mają kolejne kody ASCII.*

Ćwiczenie 1.13 *Napisz funkcję, która wyznacza minimalną wartość z przekazanej jej tablicy liczb typu float.*

Ćwiczenie 1.14 *Niech $\sigma(n)$ oznacza sumę wszystkich dzielników liczby naturalnej n mniejszych od liczby n (na przykład $\sigma(5) = 1$ oraz $\sigma(6) = 1 + 2 + 3 = 6$. Liczbę n nazywamy doskonałą jeśli $\sigma(n) = n$. Parę liczb (n, m) nazywamy zaprzyjaźnioną, jeśli $\sigma(n) = m$ oraz $\sigma(m) = n$. Znajdź wszystkie liczby doskonałe mniejsze od 1000. Wyznacz wszystkie zaprzyjaźnione pary liczb mniejszych niż 1000.*

Zadanie 1.1 *Narysuj schematy blokowe instrukcji*

1. *if (U) if (W) if (V) A else B*
2. *if (U) if (W) {if (V) A } else B*

Zadanie 1.2 *Wykaż, że każdy program zbudowany z wyrażeń arytmetycznych, instrukcji przypisania oraz instrukcji warunkowych zatrzymuje się po skończonym czasie. Spróbuj podać górne oszacowanie ilości kroków wykonywanych przez tego typu programy.*

Zadanie 1.3 *Niech $a > 0$. Definiujemy ciąg $x_0 = a$, $x_{n+1} = \frac{1}{2}(x_n + \frac{a}{x_n})$. Pokaż, że $\lim_{n\to\infty} x_n = \sqrt{a}$. Wskazówka: skorzystaj z tego, że każdy ograniczony i monotoniczny ciąg liczb rzeczywistych jest zbieżny.*

Rozdział 2

Algorytmy, algorytmy

```
while (x<>y)
if (x > y) x -= y; else y -= x;
```

Znaczna część kursu matematyki w szkole podstawowej oraz w szkole średniej polega opanowaniu metod rozwiązywania pewnej klasy zagadnień. Uczyliśmy się jak dodaje się i mnoży liczby. Jak rozwiązywać proste równania postaci $a \cdot x = b$, jak rozwiązywać równania kwadratowe, czyli równania postaci $a \cdot x^2 + b \cdot x + c = 0$, jak rozwiązywać proste układy równań liniowych. Uczyliśmy się również jak obliczać pochodną funkcji oraz jak badać jej przebieg zmienności. Krótko mówiąc - uczyliśmy się pewnych algorytmów. Rozważania tego rozdziału rozpoczniemy od omówienia kilka algorytmów ze szkoły średniej. Później omówimy kilka algorytmów o fundamentalnym znaczeniu dla całej informatyki. Będzie to prosty algorytm sortowania oraz wyszukiwania w ciągach uporządkowanych.

W przykładach kodów podawanych od tej pory w książce stosować będziemy kilka arbitralnych, acz wygodnych, konwencji. ℂ

1. korzystać będziemy ze stałych TRUE i FALSE zdefiniowanych następująco:
   #define TRUE 1
   #define FALSE 0
2. korzystać będziemy z nowego typu BOOLEAN zdefiniowanego następująco
   typedef int BOOLEAN;

2.1 Dodawanie i mnożenie dużych liczb

W większości programów które będziesz pisać do reprezentowania liczb całkowitych i rzeczywistych wystarczą Ci podstawowe typy języka C. Jednakże w niektórych sytuacjach konieczne jest wykonywanie precyzyjnych obliczeń na liczbach znacznie większych niż te które mogą być reprezentowane na przykład za pomocą typu long int.

Przypomnijmy sobie podstawowy algorytm dodawania dwóch liczb całkowitych, który opanowaliśmy w szkole podstawowej. Oto przykład jego zastosowania:

		2	3	4	5	4	7	5	
8	7	3	4	6	5	6	6	4	+
8	7	5	8	1^1	1^1	1^1	3^1	9	

Liczby X=2345455 i Y=873465664 zapisaliśmy w dwóch wierszach wyrównując je do lewej strony. Dodawanie zaczynamy od lewej strony. Z pierwszym dodawaniem

nie mamy kłopotu: 5+4 = 9, więc cyfra 9 jest pierwszą cyfrą sumy. Z następną cyfrą mamy trochę więcej kłopotu: 7+6=13. Co prawda jasne jest, że drugą cyfrą sumy jest 3, lecz zapamiętać musimy nadmiar, zwany przeniesieniem, który wynosi 1. Przyniesienie musimy uwzględnić przy wyznaczaniu następnej cyfry. I tak dalej. Pamiętać musimy jeszcze o drobnej subtelności: liczba X ma mniej cyfr niż liczba Y. Lecz z tym radzimy sobie bez trudu - uzupełniamy krótszą liczbę zerami z prawej strony. Otrzymujemy w wyniku liczbę 875811139.

Omówimy teraz implementację omówionego wyżej algorytmu. Do reprezentacji takich liczb będziemy stosowali tablice liczb całkowitych ustalonej długości, kontrolowanej przez stałą DLUGOSC. Numerować będziemy cyfry of strony lewej do strony prawej. Jeśli stała długość będzie miała wartość 10, to liczbę 1234 reprezentować będzie ciąg $(4, 3, 2, 1, 0, 0, 0, 0, 0, 0)$. Przy wykorzystaniu przedstawionej niżej procedury pamiętaj o wypełnieniu zerami całej tablicy reprezentującej liczbę. Oto kod procedury vliDodaj o trzech parametrach. Dwa pierwsze parametry (tablice A i B) traktujemy jako dane wejściowe. Wynik zwracany jest w parametrze C.

```
int vliDodaj(int A[], int B[], int C[])
{
  int i,p,su;
  p = 0; //przeniesienie
  for (i=0;i<DLUGOSC;i++)
  {
    su    = A[i]+B[i]+p;
    C[i] = su % 10;
    p     = su / 10;
  }
  return(p);
}
```

Funkcja vliDodaj zwraca końcową wartość przeniesienia. Wynik funkcji zawiera więc informację o tym, czy nie nastąpiło przekroczenie zakresu, czyli czy wynik nie powinien mieć więcej cyfr niż wartość stałej DLUGOSC.

Zwróć uwagę na to, że aby samodzielnie dodawać dowolne dwie liczby zapisane w układzie dziesiętnym musimy znać tabliczkę dodawania dla liczb ze zbioru $\{0, \ldots, 9\}$. Ponieważ łatwo można zapamiętać, że $x + 0 = 0$ oraz $x + y = y + x$, więc do sprawnego dodawania potrzebna jest nam znajomość $\binom{9}{2} = 36$ działań.

Przyjrzyjmy się ilości elementarnych operacji, do których zaliczymy podstawienie wartości pod zmienną, porównanie dwóch liczb, dodawań, mnożeń i dzieleń, które wykonywane są w trakcie wykonywania rozważanego algorytmu. Otóż działanie algorytmu rozpoczyna się od dwóch podstawień (p = 0; i = 0;). Każda pętla zaczyna się od sprawdzenia warunku `i<DLUGOSC`, następnie wykonywane są cztery operacje arytmetyczne oraz trzy podstawienia. Na końcu pętli wykonywane jest zwiększenie wartości zmiennej i. Program wykonuje więc

$$D(n) = 2 + 7 \cdot n$$

operacji, gdzie n oznacza wartość stałej DLUGOSC.

Definicja 2.1 *Niech* $f, g : \mathbb{N} \mapsto \mathbb{R}$. *Wtedy*

$$f = O(g) \leftrightarrow (\exists C > 0)(\exists k \in \mathbb{N})(\forall n > k)(f(n) < C \cdot g(n)),$$

$$f = \Theta(g) \leftrightarrow (f = O(g) \wedge g = O(f))$$

Interpretacja symbolu $f = O(g)$ dla rosnących funkcji f i g jest prosta: funkcja f rośnie asymptotycznie co najwyżej tak szybko - z dokładnością do stałej - jak funkcja g. Jeśli $f = \Theta(g)$ i obie funkcje są rosnące, to funkcje obie funkcje mają takie samo - z dokładnością do stałej - tempo wzrostu.

Twierdzenie 2.1 *Niech* $f, g : \mathbb{N} \mapsto \mathbb{R}$. *Załóżmy, że istnieje granica* $\lim_{n\to\infty} \frac{f(n)}{g(n)}$. *Wtedy*

1. *jeśli* $0 \leq \lim_{n\to\infty} \frac{f(n)}{g(n)} < \infty$ *to* $f = O(g)$,
2. *jeśli* $0 < \lim_{n\to\infty} \frac{f(n)}{g(n)} < \infty$ *to* $f = \Theta(g)$.

Dowód. Niech $c = \lim_{n\to\infty} \frac{f(n)}{g(n)}$. Załóżmy, że $0 \leq \lim_{n\to\infty} \frac{f(n)}{g(n)} < \infty$. Wtedy, dla dostatecznie dużych n, prawdziwa jest nierówność $0 \leq \frac{f(n)}{g(n)} < c + 1$. Zatem, dla dostatecznie dużych n, prawdziwa jest nierówność $f(n) < (c + 1)g(n)$, co kończy dowód punktu (1). Jeśli ponadto $c > 0$, to $0 \leq \lim_{n\to\infty} \frac{g(n)}{f(n)} = \frac{1}{c} < \infty$, więc z punktu (1) wynika, że g = O(f), czyli $f = \Theta(g)$. □

Zauważmy teraz, że

$$\lim_{n\to\infty} \frac{D(n)}{n} = \lim_{n\to\infty} \frac{2 + 7 \cdot n}{n} = \lim_{n\to\infty} (\frac{2}{n} + 7) = 7,$$

Zatem $D = O(n)$. Omówiony algorytm dodawania działa więc w czasie $O(N)$, gdzie N oznacza maksimum z ilości cyfr rozważanych liczb.

Algorytm mnożenia wymaga znajomości tabliczki mnożenia. Ponieważ $x \cdot 0 = 0$ oraz $x \cdot 1 = x$ więc znać musimy $\binom{8}{2} = 28$ działań. Przeanalizujmy algorytm mnożenia na przykładzie dwóch liczb $X = 1111$ oraz $Y = 989$.

			1	1	1	
			9	8	9	*
			9	9	9	
		8	8	8	0	
	9	9	9	0	0	
1	0	9	7	7	9	

Widzimy, że proces ręcznego mnożenia dwóch liczb całkowitych rozkłada się na kilka faz. Jedno z podzadań polega na pomnożeniu liczby przez cyfrę - algorytm tego podzadania jest zbliżony do ciała głównej pętli algorytmu dodawania i zajmuje O(N) działań. Po pomnożeniu pierwszej liczby o i-tą cyfę drugiej liczby należy wynik przesunąć o i pozycji w lewo. Na końcu należy zsumować otrzymane liczby. Ten ostatni etap można zrealizować nieco inaczej, mianowicie dodawanie możemy wykonywać w locie, zaraz po pomnożeniu przez cyfrę i przesunięciu w lewo. Oto szkic algorytmu mnożenia:

```
int vliMnoz(int A[], int B[], int C[])
{
 ...
 Zeruj(C);
 for (i=0;i<DLUGOSC;i++)
 {
```

```
    Kopiuj A do Rob;
    Pomnoż Rob przez cyfrę B[i];
    Przesuń Rob o i pozycji w lewo;
    Wypełnij pierwsze i−1 pozycji Rob zerami;
    Dodaj do liczbę C liczbę Rob i wynik zapamiętaj w C;
  }
}
```

Główna pętla omówionego algorytmu wykonywana jest N razy, gdzie N oznacza wartość stałej DLUGOSC. Wewnątrz tej pętli wykonywanych jest $O(N)$ działań. Liczba elementarnych operacji wykonywanych przez ten algorytm jest więc rzędu $O(N^2)$. W dalszej części tej książki pokażemy, że zadanie to można wykonać w istotnie krótszym czasie.

2.2 Układ równań liniowych

Z rozwiązywaniem równań liniowych jednej zmiennej $a \cdot x + b = c$ zetknęliśmy się jeszcze w szkole podstawowej. Jeśli $a \neq 0$ to równanie to ma rozwiązanie równe $(c - b)/a$. Jeśli $a = 0$ oraz $b = c$ to dowolna liczba rzeczywista x jest jego rozwiązaniem. Jeśli zaś $a = 0$ oraz $b \neq c$, to równanie nie ma żadnego rozwiązania. Omówioną metodę łatwo można zamienić na program w języku C który wczytuje parametry a,b i c i wyprowadza rozwiązanie bądź stwierdza, że układ jest nieoznaczony bądź też sprzeczny.

Układ równań liniowych

$$\begin{cases} ax + by = c \\ dx + ey = f \end{cases}$$

rozwiązać można za pomocą wzorów Cramera. W tym celu najpierw należy wyznaczyć jego wyznacznik główny $\Delta = ae - db$. Jeśli jest on równy zeru to układ jest sprzeczny lub nieoznaczony. Jeśli jest różny od zera, to układ ma rozwiązanie. Jest nim para $x = (ce - fb)/\Delta$ oraz $y = (af - dc)/\Delta$. Oto program, który jest implementacją omówionej metody:

```
#include <stdio.h>
BOOLEAN url2(double a,double  b,double c,double d,
          double e,double f, double *x, double *y)
{
  double det;
  det = a*e - d*b;
  if (det==0)
    return(FALSE);
  *x = (c*e - f*b) / det;
  *y = (a*f - d*c) / det;
  return(TRUE);
}

int main()
{
  double a,b,c,d,e,f,x,y;
  printf(''Podaj parametry a, b, c, d, e, f: '');
  scanf(''%lf %lf %lf %lf %lf %lf'', &a, &b, &c, &d, &e, &f);
  printf(''\n'');
```

```
if (url2(a,b,c,d,e,f,&x,&y))
    printf(``Rozwiązanie układu: x = %f y= %f\n'', x, y);
else
   printf(``Układ jest sprzeczny lub nieoznaczony.\n'');
return(0);
}
```

W programie tym oddzieliliśmy instrukcje wejścia i wyjścia od zasadniczej części programu, którą jest funkcja url2. Funkcja ta zwraca wartość FALSE (czyli zero) jeśli układ jest nieoznaczony lub sprzeczny oraz wartość TRUE (czyli 1) jeśli układ ma rozwiązanie.

Uwaga. Powyższy program może dawać wyniki mocno odbiegające od prawdziwych jeśli wartość bezwzględna wyznacznika det jest bardzo mała liczbą. Do problemu tego wrócimy w jednym z następnych rozdziałów.

2.3 Równanie kwadratowe

W szkole średniej byliśmy zmuszani, w większości przypadków skutecznie, do opanowania metody rozwiązywania równań kwadratowych, czyli równań postaci $a \cdot x^2 + b \cdot x + c = 0$, gdzie a jest liczbą rzeczywistą różną od zera. Pewnie wiesz o tym, że cała tajemnica tego równania tkwi w formule

$$a \cdot x^2 + b \cdot x + c = a\left((x + \frac{b}{2a})^2 - \frac{b^2 - 4ac}{2a}\right),$$

którą łatwo możesz sprawdzić bezpośrednim rachunkiem, lub też możesz ją wyprowadzić, korzystając z tego, że

$$x^2 + \frac{b}{a}x + \frac{c}{a} = \left(x + \frac{b}{2a}\right)^2 - \left(\frac{b}{2a}\right)^2 + \frac{c}{a}.$$

Jak pewnie dobrze pamiętasz, w celu wyznaczenia rozwiązań tego równania (czyli tak zwanych pierwiastków) należy obliczyć liczbę $\Delta = b^2 - 4ac$, zwaną wyróżnikiem równania, a następnie zbadać jej znak. Jeśli $\Delta < 0$, to równanie to nie ma rozwiązań w liczbach rzeczywistych, jeśli $\Delta = 0$, to równanie to ma jedno rozwiązanie równe $-\frac{b}{2a}$. Jeśli zaś $\Delta > 0$, to równanie to ma dwa rozwiązania

$$x_1 = \frac{-b - \sqrt{\Delta}}{2a}, x_1 = \frac{-b + \sqrt{\Delta}}{2a}.$$

Oto kod funkcji rkwad, której przekazywane są parametry a, b ,c równania kwadratowego i dwa wskaźniki na wynik. Wynikiem tej funkcji jest liczba rozwiązań równania. Jeśli parametr a jest zerowy, to funkcja ta zwraca wartość -1, sygnalizując w ten sposób błędne dane wejściowe.

```
int rkwad(double a,double b,double c,double *x1, double *x2)
{
  double Delta;

  if (a==0){
```

```
    return(-1);
  }
  Delta = b*b - 4.0*a*c;
  if (Delta < 0.0)
    return(0);
  else
  if (Delta==0)
  {
    *x1 = (-b)/(2.0*a);
    *x2 = *x1;
    return(1);
  }
  else
  {
    *x1 = (-b - sqrt(Delta))/(2.0*a);
    *x2 = (-b + sqrt(Delta))/(2.0*a);
    return(2);
  }
}
```

W podobny, lecz nieco bardziej złożony sposób możesz rozwiązać równanie stopnia trzeciego oraz czwartego. Nie będziemy omawiali tutaj potrzebnych wzorów - materiał szkoły średniej ich nie obejmuje. Być może z metodą rozwiązywania tych równań zapoznasz się na wykładach z algebry. Warto zaś wiedzieć, że dla równań stopnia pięć oraz większych nie istnieją analogiczne wzory. Pokazali to Abel i Galois w pierwszej połowie XIX wieku.

2.4 Liczby naturalne

Liczby naturalne stanowią szkielet matematyki. Można z nich skonstruować wszystkie pozostałe klasyczne struktury matematyczne: liczby wymierne, liczby rzeczywiste, płaszczyznę i przestrzeń. Do liczb naturalnych zaliczamy liczbę zero a zbiór liczb naturalnych oznaczamy symbolem $\mathbb{N}$, czyli

$$\mathbb{N} = \{0, 1, 2, 3, \ldots\}.$$

Relacja nierówności $\leq$ na zbiorze liczb naturalnych posiada następującą bardzo ważną własność:

> W każdym niepustym podzbiorze zbioru zbioru liczb naturalnych istnieje element najmniejszy.

Własność ta nazywa się „**zasadą dobrego uporządkowania**” liczb naturalnych. Drugą ważną własnością liczb naturalnych jest to, że

$$(\forall n \in \mathbb{N})(n \neq 0 \rightarrow (\exists m \in \mathbb{N})(n = m + 1)).$$

Własności te charakteryzują zbiór liczb naturalnych. W szczególności można z nich wyprowadzić wszystkie warianty zasady Indukcji Matematycznej z jakimi zetknęliśmy się w szkole średniej.

Twierdzenie 2.2 *Nie istnieje nieskończony i ostro malejący ciąg liczb naturalnych.*

Dowód. Załóżmy, że $(a_n)_{n\in\mathbb{N}}$ jest takim ciągiem, że $a_0 > a_1 > a_2 > \ldots$. Niech $A = \{a_n : n \in \mathbb{N}\}$. Wtedy A jest niepustym podzbiorem zbioru liczb naturalnych. Posiada więc element najmniejszy. Niech nim będzie a_k. Lecz wtedy $a_{k+1} < a_k$ i $a_{k+1} \in A$, co jest sprzeczne z wyborem element a_k. □

Twierdzenie 2.3 (O dzielenie z resztą) *Jeśli $a \in \mathbb{Z}$ oraz $b \in \mathbb{N} \setminus \{0\}$ to istnieją takie liczby $q, r \in \mathbb{Z}$, że*

$$(a = q \cdot b + r) \wedge (0 \leq r < b).$$

Liczbę r nazywamy resztą z dzielenia liczby a przez liczbę b lub też resztą z a modulo b. Liczbę q nazywamy ilorazem całkowito - liczbowym liczb a i b.
Dowód. Twierdzenie to dla liczb $a \geq 0$ łatwo można udowodnić Indukcją Matematyczną względem a. Jeśli $a < 0$, to należy zastosować udowodnione twierdzenie do liczby $-a$. □

Operatory % oraz / określone dla typów całkowitych mają wyraźny związek z udowodnionym właśnie twierdzeniem. Jeśli mianowicie a i b są zmiennymi typu całkowitego oraz b>0, to

$$(a = (a/b) \cdot b + (a\%b)) \wedge (0 \leq a\%b < b),$$

zatem a/b jest ilorazem całkowito - liczbowym liczb a i b, zaś a % b jest resztą z dzielenia zmiennej a przez zmienną b.

Na zbiorze liczb całkowitych określony jest pewien naturalny porządek, zwany relacją podzielności. Określony jest on formuła:

$$a|b \leftrightarrow (\exists x \in \mathbb{Z})(a \cdot x = b).$$

Kryterium podzielności dla zmiennych a, b typu całkowitego można więc zapisać następująco:

$$(a|b) \leftrightarrow (b\%a = 0)$$

Największy wspólny dzielnik

Wiele zagadnień teorii liczbowych ma związek z największym wspólnym dzielnikiem niezerowych liczb całkowitych. Przypomnijmy, że

$$NWD(a, b) = \max\{k \in \mathbb{N} : k|a \wedge k|b\}$$

Idea algorytmu wyznaczania największego wspólnego dzielnika pochodzi od Eulkidesa i oparta jest na dwóch łatwych własnościach największego wspólnego dzielnika:

Lemat 2.1 *Załóżmy, że a i b są dwoma niezerowymi liczbami całkowitymi. Wtedy*

1. *jeśli $b|a$ to $NWD(a, b) = b$,*
2. *jeśli $a = qb + r$, gdzie $r, q \in \mathbb{Z}$ i $r \neq 0$ to $NWD(a, b) = NWD(b, r)$.*

Dowód. Pierwsza część Lematu jest oczywista. Załóżmy najpierw, że $a = qb + r$, $r > 0$, $x \neq 0$ i $x|a$ oraz $x|b$. Niech $a' = a/x$ i $b' = b/x$. Wtedy $r = xa' - qxb' = x(a' - qb')$, więc x jest dzielnikiem liczb r oraz b. Załóżmy następnie, że $y|b$ i $y|r$. Niech $b' = b/y$ i $r' = r/y$. Wtedy $a = y(qb' + r')$, więc $y|a$. Zatem y jest wspólnym dzielnikiem liczb a i b. Pokazaliśmy więc, że pary (a, b) i $b, r)$ mają te same wspólne dzielniki.

□

Udowodniony lemat jest podstawą teoretyczną klasycznego algorytmu Euklidesa wyznaczania największej wspólnej wielokrotności dodatnich liczb naturalnych. Polega on na cyklicznym wyznaczaniu wartości x_n i y_n za pomocą zależności:

$$\begin{aligned} x_{n+1} &= y_n \\ y_{n+1} &= x_n \% y_n, \end{aligned} \tag{2.1}$$

gdzie x_0 i y_0 są danymi parametrami. Procedurę tę przerywamy, gdy okaże się, że y_n jest równe zero. Z udowodnionego lematu wynika, że jeśli $y_{n+1} \neq 0$, to $NWD(x_n, y_n) = NWD(x_{n+1}, y_{n+1})$. Jeśli zaś $y_{n+1} = 0$, to $y_n | x_n$, zatem $NWD(x_n, y_n) = x_{n+1}$. Zauważmy również, że $y_0 > y_1 > y_2 > \ldots$. **Ponieważ nie istnieją nieskończone malejące ciągi liczb naturalnych**, więc musi istnieć taka liczba naturalna n, że $y_n = 0$. Algorytm Euklidesa generuje więc taki ciąg $\{(x_k, y_k\}_{k=0\ldots n}$ par liczb naturalnych, że

$$NWD(x_0, y_0) = NWD(x_1, y_1) = \ldots = NWD(x_n, y_n)$$

i $y_{n+1} = x_n \% y_n = 0$ dla pewnego n. A wtedy $NWD(x_0, y_0) = y_n = x_{n+1}$.

Przed zamienieniem powyższego algorytmu na program w języku C zauważmy, że do jego realizacji wystarczą tylko dwie zmienne całkowite x i y. Program będzie zawierał pętlę, którą opuścimy, gdy zmienna y przyjmie wartość zero. W trakcie pętli wykonywać będziemy podstawienie $(x, y) \leftarrow (y, x\%y)$. W języku C nie ma jednak instrukcji **jednoczesnego podstawiania**. Zamiast tego wprowadzimy zmienną roboczą r i podstawienie to zrealizujemy za pomocą ciągu instrukcji $r \leftarrow x\%y; x \leftarrow y; y \leftarrow r$.

Oto ostateczna postać algorytmu wyznaczania największego wspólnego dzielnika dodatnich liczb naturalnych x oraz y:

```
long int NWD(long int x, long int y)
{
long int r;
do{
   r = x % y;  //r = reszta z dzielenia x przez y
   x = y;
   y = r;
}
while (y != 0);
return(x);
}
```

Twierdzenie 2.4 $(\forall a, b \in \mathbb{Z} \setminus \{0\})(\exists X, Y \in \mathbb{Z})(aX + bY = NWD(a, b))$

Dowód. Niech m będzie najmniejszą liczbą naturalną dodatnią która może zostać zapisana w postaci $aX_0 + bY_0$, gdzie $X_0, Y_0 \in \mathbb{Z}$. Zauważmy, że jeśli $k|a$ oraz $k|b$ to $k|(aX + bY)$. Zatem $NWD(a, b)|m$, a więc $NWD(a, b) \leq m$.

Załóżmy, że $a = qm + r$, gdzie $q \in \mathbb{Z}$ i $r \in \mathbb{N}$ oraz $0 < r < b$. Wtedy

$$r = a - qm = a - (aX_0 + bY) = a(1 - X_0) + b(-Y_0),$$

co jest sprzeczne z definicją liczby m. Zatem $m|a$. Podobnie pokazujemy, że $m|b$. Zatem $m \leq NWD(a, b)$.

□

Dowód ten jest bardzo krótki. Ma jednak pewną wadę - nie daje się w sposób oczywisty przełożyć na algorytm. Zawiera bowiem element nieskończony. Jest nim zbiór $Z = \{aX + bY : X, Y \in \mathbb{Z}\}$.

Wniosek 2.1 *Niech $a, b \in \mathbb{Z} \setminus \{0\}$ i $c \in \mathbb{Z}$. Równanie*

$$aX + bY = c$$

ma rozwiązanie w liczbach całkowitych wtedy i tylko wtedy, gdy

$$NWD(a, b)|c.$$

Liczby pierwsze

Liczbę naturalną nazywamy liczbą pierwszą jeśli jest większa od 1 oraz jeśli się dzieli tylko przez liczbę 1 oraz samą siebie. Inaczej mówiąc, liczba naturalna n jest pierwsza, jeśli

$$(n \geq 2) \wedge (\forall k, l \in \mathbb{N})(n = k \cdot l \rightarrow (k = 1 \vee k = n))$$

Liczby pierwsze odgrywają w matematyce podobną role do atomów w chemii. Oto podstawowa własność liczb pierwszych, zwana pierwszym twierdzeniem Euklidesa:

Twierdzenie 2.5 (Euklides) *Jeśli p jest liczbą pierwszą i $p|ab$ to $p|a$ lub $p|b$.*

Dowód. Załóżmy, że $p|ab$ ale $\neg(p|a)$. Wtedy liczby p i a są względnie pierwsze, więc $NDW(p, a) = 1$. Istnieją zatem takie liczby całkowite X i Y takie, że $pX + aY = 1$. Wtedy $bpX + baY = b$. Liczba p dzieli lewą stronę tej równości. Więc $p|b$. □

Z twierdzenia tego wynika „Podstawowe Twierdzenie Arytmetyki", które stwierdza, że dowolną liczbę naturalną $m > 1$ można w jednoznaczny sposób przedstawić w postaci iloczynu

$$m = p_1^{n_1} p_2^{n_2} \dots p_k^{n_k},$$

gdzie $p_1 < p_2 < \ldots < p_k$ są pewnymi liczbami pierwszymi i $n_1, \ldots, n_k$ dodatnimi liczbami naturalnymi. Euklides pokazał również, że liczb pierwszych jest nieskończenie wiele.

Zauważ, że jeśli $n = k \cdot l$ to jedna z tych liczb k bądź l musi być mniejsza lub równa niż pierwiastek kwadratowy z liczby n. Gdyby bowiem $k < \sqrt{n}$ oraz $l < \sqrt{n}$, to wtedy $k \cdot l < \sqrt{n} \cdot \sqrt{n} = n$. Sprawdzenie tego, że liczba $n > 2$ nie jest pierwsza możemy więc ograniczyć do sprawdzenia, czy dzieli się przez którąś z liczb ze zbioru $\{2, 3, 4, \ldots, \lfloor\sqrt{n}\rfloor\}$, gdzieś przez $\lfloor x \rfloor$ oznaczamy część całkowitą liczby x.

Jak już wiemy, sprawdzenie tego, czy liczba k dzieli liczbę l jest równoważne sprawdzeniu tego, czy `l % k = 0`. W bibliotece `math.h` istnieje funkcja `sqrt`, która służy do obliczenia pierwiastka kwadratowego z podanego parametru typu `double`. Zwraca na również wynik typu `double`. Aby wyznaczyć liczbę $\lfloor\sqrt{n}\rfloor$ skorzystamy z mechanizmu **rzutowania typów** języka C. Za pomocą instrukcji ℂ

```
(unsigned int) sqrt(n)
```

przekształcimy liczbę `sqrt(n)` na typ `unsigned int`. Oto kod funkcji która sprawdza, czy dana liczba $n \geq 2$ jest liczbą pierwszą:

```
BOOLEAN IsPrime(unsigned long int n)
{
  unsigned int gr;
  unsigned int i = 2;
  gr = (unsigned int) sqrt(n);
  while ((i<=gr)&&((n % i) != 0))
    i++;
  return(i>gr);
}
```

Zmienna i służy do przeglądania odcinka początkowego zbioru $\{2, 3, 4, \ldots, [\sqrt{n}]\}$. Pętla while zostaje przerwana w dwóch przypadkach. Pierwszym powodem przerwania pętli może być przekroczenie wartości zmiennej granica, czyli liczby $\lfloor\sqrt{n}\rfloor$. Wtedy wyrażenie i>granica przyjmuje wartość 1. Drugim powodem przerwania pętli może być spełnienie warunku n % i = 0. Wtedy liczba n jest podzielna przez i i wartość wyrażenia i>granica wynosi 0. Główną pętlę możesz zapisać nieco prościej

```
  while ((i<=gr)&&(n % i)) i++;
```

gdyż, jak pamiętasz, **język C traktuje wartości różne od zera jako prawdę**!

Sito Erastotenesa

Najstarszy znany algorytm wyznaczania liczb pierwszych pochodzi od Erastotenesa. Polega on kolejnym skreślaniu z tablicy liczb naturalnych liczb podzielnych przez 2, 3, 5, 7 itd. Mówiąc dokładniej, zaczynamy od wypełnienia wartościami TRUE tablicy T[2 ... N], gdzie N jest ustalona liczbą naturalną. Następnie skreślamy w tej tablicy wszystkie wielokrotności liczby 2, czyli liczby 2*2, 3*2, 3*4, itd. Przez skreślenie liczby i rozumiemy podstawieniu T[i] = FALSE. Pierwszą nieskreśloną liczbą większą od 2 będzie wtedy 3. Skreślamy teraz wielokrotności liczby 3. Pierwszą nieskreśloną liczbą będzie teraz 5. I tak dalej. Z powodów wyjaśnionych wyżej - przy omawianiu funkcji rozstrzygającej, czy dana liczba jest pierwsza - skreślanie wykonać wystarczy dla liczb od 2 do $\lfloor N\rfloor$.

Główną częścią algorytmu będzie pętla kontrolowana przez aktualnie skreślaną liczbę, którą reprezentować będzie zmienna i. Wewnątrz głównej pętli wykonujemy dwie czynności: skreślamy wielokrotności zmiennej i oraz poszukujemy najmniejszej większej od i nie skreślonej liczby. Bez przerwy dbamy o to aby nie przekroczyć zakresu tablicy, kontrolowanego przez zmienną MAX. Korzystamy również z tego, że $i \cdot (j + 1) = (i \cdot j) + i$.

```
void SitoE(BOOLEAN Tab[],long int MAX)
{
  long int i,j,gr;
  for (i=2;i<MAX;i++)
    Tab[i] = TRUE;
  gr = (long) sqrt(MAX);
  i  = 2;
  while (i<=gr)
  {
    j = i*2;
    while (j<MAX)
    {
      Tab[j] = FALSE;
      j += i;
```

```
      }
      i++;
      while ((i<=gr) && (Tab[i]==FALSE))
        i++;
   }
}
```

2.5 Sortowanie

Jednym z najważniejszych praktycznych problemów algorytmicznych jest sortowanie. Występuje jako element składowy wielu innych algorytmów. Jest ono również bardzo interesujące z teoretycznego punktu widzenia.

Polega ona uporządkowaniu elementów danej listy nieuporządkowanych obiektów. Przed precyzyjnym sformułowaniem tego zagadnienia musimy wprowadzić (bądź przypomnieć) kilka pojęć.

Definicja 2.2 ***Częściowym porządkiem*** *na zbiorze X nazywamy taką relację R pomiędzy elementami zbioru S, która spełnia następujące własności:*

1. $(\forall a \in X)(aRa)$ *(zwrotność),*
2. $(\forall a \in X)(\forall b \in X)(\forall c \in X)(aRb \wedge bRc \rightarrow aRc)$ *(przechodniość)*
3. $(\forall a \in X)(\forall b \in X)(aRb \wedge bRa \rightarrow a = b)$*(słaba antysymetria)*

Najważniejszym częściowym porządkiem jest relacja zawierania zbiorów, czyli inkluzja. Słaba antysymetria inkluzji jest treścią tak zwanego „Aksjomatu Ekstensjonalności". Bardzo ciekawym częściowym porządkiem jest, rozważana już wcześniej, relacja podzielności na zbiorze liczb $\mathbb{N} \setminus \{0\}$.

Definicja 2.3 *Częściowy porządkiem R na zbiorze X nazywamy* ***porządkiem liniowym****, jeśli*

$$(\forall a \in X)(\forall b \in X)(aRb \vee bRa)$$

Własność występującą w definicji liniowego porządku nazywamy *spójnością*. Standardowa relacje $\leq$ porządkująca liczby rzeczywiste jest oczywiście liniowym porządkiem. Inkluzja nie jest częściowym porządkiem. Relacja podzielności również nie jest liniowym porządkiem.

Załóżmy teraz, że mamy dany ciąg $a_0, \ldots a_{n-1}$ elementów ustalonego zbioru X uporządkowanego przez liniowy porządek $\preceq$. Problem sortowania polega na znalezieniu takiej permutacji π zbioru indeksów $\{0, \ldots, n\}$, że

$$a_{\pi(0)} \preceq a_{\pi(1)} \preceq \ldots \preceq a_{\pi(n-1)}.$$

Efektem sortowania ciągu liczb naturalnych $(12, 3, 23, 11, 7)$ w sensie naturalnego porządku jest ciąg $(3, 7, 11, 12, 23)$. Permutacja porządkującą wyjściowy ciąg jest określona równościami $\pi(0) = 1$, $\pi(1) = 4$, $\pi(2) = 3$, $\pi(3) = 0$ i $\pi(4) = 2$.

Istnieje wiele metod sortowania. Z grubsza mówiąc, można je podzielić na dwie kategorie: *wewnętrzne* i *zewnętrzne*. Metody wewnętrzne działają na danych przechowywanych w pamięci operacyjnej maszyny. Metody zewnętrzne służą do porządkowania danych przechowywanych na zewnętrznych nośnikach pamięci i dotyczą bardzo dużych zestawów danych. W książce tej omawiać będziemy tylko metody wewnętrzne.

W rozdziale tym omówimy jedną metodę, która nazywa się sortowaniem przez *wstawienie*. Metoda ta jest często stosowana przez osoby grające w karty. Elementy sortowanej listy podzielone są na dwie części: $(a_0, \ldots, a_{i-1})$ i $(a_i, \ldots, a_{n-1})$. Elementy pierwszej podlisty są uporządkowane, czyli $a_0 \leq \ldots \leq a_{i-1}$. Przyglądamy się teraz elementowi a_i i wstawiamy go do pierwszej podlisty we właściwe miejsce. Otrzymujemy w ten sposób dwie nowe listy $(a'_1, \ldots, a'_i)$ oraz $(a_{i+1}, \ldots, a_{n-1})$. Pierwsza z nich zwiększyła swoją długość o 1 i jest nadal uporządkowana. Druga skróciła się o jeden. Operację tę powtarzamy aż druga podlista stanie się pusta. Algorytm ten możemy zapisać następująco:

```
for (i=1;i<n;i++)
{
  m = x[i];\\
  ``wstaw m we właściwe miejsce x[0]\ldots x[i]''
}
```

Oczywiście, musimy teraz doprecyzować fragment kodu odpowiedzialny za wstawienie m we właściwe miejsce ciągu x[0] ... x[i]. Możemy to zrobić tak:

1. porównujemy m z x[i-1]; jeśli x[i-1]>m, to x[i-1] przeniesiemy do x[i]; jeśli x[i-1]<=m, to przerywamy postępowanie
2. porównujemy m z x[i-2];jeśli x[i-2]>m, to x[i-2] przeniesiemy do x[i-1]; jeśli x[i-2]<=m, to przerywamy postępowanie
3. itd.

Procedurę przerywamy też, gdy dojdziemy do zerowego elementu tablicy x. Po znalezieniu właściwego miejsca wstawiamy we właściwą komórkę wartość m. Oto pełna implementacja omówionej metody:

```
for (i=1;i<n;i++)
{
  m = x[i];
  j = i-1;
  while ((j>=0) && (x[j]>m))
  {
    x[j+1] = x[j];
    j--;
  }
  x[j+1] = m;
}
```

Oznaczmy przez C_n liczbę porównań oraz P_n liczbę podstawień wykonywanych w trakcie wykonania omówionej metody. Liczby te oczywiście zależą od danych początkowych. Jeśli początkowy ciąg jest już uporządkowany, czyli gdy $x[0] \leq x[1] \leq \ldots \leq x[n-1]$, to wykonanych jest tylko $n-1$ porównań i tyle samo podstawienień, czyli $(C_n)_{min} = (P_n)_{min} = n-1$. Najgorzej wygląda sprawa z ciągiem odwrotnie

uporządkowanym, czyli takim, że $x[0] \geq x[1] \geq \ldots \geq x[n-1]$. Wtedy w i-tym kroku iteracji wykonywane są porównania wartości x[i] ze wszystkimi wartościami $x[i-1], \ldots, x[0]$ i wykonywane są następujące podstawienia: x[i] = x[i-1], x[i-1] = x[i-2], ... x[1] = x[0] i w końcu x[0] = m. Zatem mamy

$$(C_n)_{max} = \sum_{i=1}^{n-1} i = \frac{n(n-1)}{2}$$

$$(P_n)_{max} = \sum_{i=1}^{n-1} (i+1) = \frac{(n+2)(n-1)}{2}$$

Zauważmy, że $\lim_{n\to\infty} \frac{n(n-1)/2}{n^2} = \lim_{n\to\infty} \frac{(n+2)(n-1)/2}{n^2} = \frac{1}{2}$. Zatem $(C_n)_{max} = O(n^2)$ oraz $(P_n)_{max} = O(n^2)$. Pokazać można również, że średnie ilości porównań $(C_n)_r$ $(P_n)_r$ są rzędu $O(n^2)$. Wynik ten nie jest zachwycający. Do posortowania tablicy jednego miliona liczb całkowitych za pomocą wstawiania potrzebujemy mniej więcej $(10^6)^2 = 10^{12}$ elementarnych operacji.

Załóżmy, że mamy do dyspozycji komputer który wykonuje 10^9 elementarnych operacji na sekundę, co z grubsza biorąc odpowiada częstotliwości procesora rzędu kilkudziesięciu mega herców. Komputer ten do wykonania 10^{12} operacji będzie potrzebował $10^{12}/10^9 = 10^3$ sekund, czyli około $\frac{1000}{60} \approx 15$ minut.

W tym miejscu warto jest zwrócić uwagę na kilka dosyć ważnych liczb. Jedna doba ma 24 godziny, godzina składa się z 60 minut a minuta z 60 sekund. Jedna doba ma zatem $86\,400 \approx 10^5$ sekund.

Zastanówmy się ile czasu może zająć posortowanie tablicy składającej się z 40 milionów liczb. Zwróć uwagę na to, że liczba ta to w przybliżeniu liczba ludności naszego kraju. Otóż $(40 \cdot 10^6)^2 = (4 \cdot 10^7)^2 = 16 \cdot 10^{14} \approx 10^{15}$. Sortowanie zajmie nam zatem około $10^{15}/10^9 = 10^6$ sekund, czyli około $10^6/10^5 = 10$ dni.

Rok ma mniej więcej 365 dni. Zatem jeden rok składa się $31\,536\,000 \approx 3 \cdot 10^7$. Liczbę tę łatwo można zapamiętać, jeśli się zna zasadę sformułowaną przez T. Duffa, która głosi, że z dokładnością do pół procenta

π sekund to nanowiek

(przypomnijmy, że „nano" oznacza jedną miliardową, czyli 10^{-9}). Wszechświat powstał około 15 miliardów lat temu. Liczy więc około $3 \cdot 10^7 \cdot 15 \cdot 10^9 = 45 \cdot 10^{16} \approx 5 \cdot 10^{17}$ sekund.

Oszacujmy ile czasu zajmie posortowanie tablicy 5 miliardów liczb. Liczba ta to przybliżenie rozmiaru obecne populacji ludzkiej. Otóż $(5 \cdot 10^9)^2 = 25 \cdot 10^{18}$. Zatem zadania to powinno nam zająć około $25 \cdot 10^{18}/10^9 = 25 \cdot 10^9$ sekund, czyli około $25 \cdot 10^9/3 \cdot 10^7 \approx 100$ lat. Sto lat stosunkowo krótki okres w porównaniu z wiekiem wszechświata. Jest to jednak dosyć spory szmat czasu. Widzimy, że metoda sortowania przez wstawienie nie jest rewelacyjnie szybka. Na szczęście, istnieją znacznie szybsze metody sortowania. Do tego tematu wrócimy w dalszych rozdziałach tej książki.

2.6 Wyszukiwanie binarne

Informacja o tym, że pewna tablica danych jest posortowana jest bardzo cenna. Przydawać się może do istotnego usprawnienia konstruowanych algorytmów. Rozważmy

zagadnienie wyszukiwania elementu w tablicy liczb całkowitych. Załóżmy mianowicie, że mamy dany ciąg $x_0x_1 \ldots x_{n-1}$ liczb całkowitych. Interesuje nas, czy dana liczba całkowita m występuje w tym ciągu. Zadanie to jest proste dla nas: przejrzymy w tym celu cały ciąg i w trakcie przeglądania będziemy porównywali kolejne elementy x_i z liczbą m. Oto fragment kodu:

```
boolean Istnieje(int x[], int n, int m)
{
 int i;
 for (i=0;i<n;i++)
   if (x[i] == m) return(TRUE);
 return(FALSE);
}
```

Liczba porównań wykonywanych w trakcie działania tego algorytmu zależy oczywiście od danych. Najdłużej będzie on działał wtedy, gdy element m nie występuje w tablicy x. Wykonywanych będzie wtedy $2n$ porównań (parametr 2 bierze się z tego, że dla każdej wartości i wykonujemy dwa porównania: $i < n$ oraz $x[i] == m$). Najgorszy czas działania tego algorytmu jest więc rzędu $O(n)$.

Załóżmy teraz, że wiemy, że ciąg $x_0x_1 \ldots x_{n-1}$ jest uporządkowany. Niech liczby L i P oznaczają lewą i prawą granicę obszaru poszukiwań. Na początku L będzie miało wartość 1 zaś P wartość $n-1$. Wyliczamy środek obszaru poszukiwań. Jest nim liczba $S = \lfloor \frac{L+P}{2} \rfloor$. Porównajmy element x_S z wartością m. Jeśli $x_S = m$, to poszukiwanie zakończyło się sukcesem. Jeśli $m < x_S$ to element m może pojawić się tylko w podciągu $x_0 \ldots x_{S-1}$. Wtedy nowym obszarem poszukiwań będzie przedział $x_0 \ldots x_{S-1}$. Jeśli zaś $x_S < m$ to element m może pojawić się tylko w podciągu $x_{S+1} \ldots x_{n-1}$. Wtedy nowym obszarem poszukiwań będzie przedział $x_{S+1} \ldots x_{n-1}$. Jeśli więc $x_s \neq m$, to nowy obszar poszukiwań staje się dwa ramy mniejszy od początkowego. Operację tę powtarzamy. Przerywamy jej działanie, jeśli obszar poszukiwań zrobi się pusty.

Zastanówmy się ile razy w najgorszym przypadku wykonywana będzie główna pętla opisanego wyżej algorytmu. Niech D_k oznacza wartość wyrażenia $(P-L)+1$ na początku k-tej iteracji pętli. $D-k$ jest długością obszaru poszukiwań. Oczywiście $D_1 = n$. Ponieważ w kolejnej iteracji długość obszaru ulega dwukrotnemu zmniejszeniu, więc $D_{k+1} \leq \frac{D_k}{2}$. Z tego wynika, że

$$D_k \leq \frac{D_{k-1}}{2} \leq \frac{D_{k-2}}{2^2} \leq \ldots \leq \frac{D_1}{2^k} = \frac{n}{2^k}.$$

Proces przeszukiwania ulega przerwaniu jeśli wartość D_k stanie się mniejsza od 1. A to nastąpi jeśli $\frac{n}{2^k} < 1$, czyli, gdy $n < 2^k$. W celu wyznaczenia granicznej liczby k nałożymy na obie strony tej nierówności funkcję $\log_2$. Otrzymamy nierówność $\log_2(n) < \log_2(2^k)$. Wiemy, że $\log_2(a^b) = b\log_2(a)$ oraz $\log_2(2) = 1$. Otrzymujemy więc nierówność $\log_n(n) < k$, z czego wynika, że iteracje zakończą się po $log_2(n)$ krokach. Otrzymaliśmy coś ciekawego: jeśli $n = 10^6$, to przeszukiwanie ciągu $x_0x_1 \ldots x_{n-1}$ zakończy się się po co najwyżej $\log_2(10^6) = 6\log_2(10) \approx 16 \cdot 3.22 \approx 20$ krokach. A proste wyszukiwanie wymaga około jednego miliona iteracji. Jeśli jeszcze nie czujesz tej różnicy, to pomyśl sobie przez chwilę o tym, co możesz kupić za 20 złotych a co za 1 000 000 złotych! Przyjrzyjmy się teraz implementacji tego algorytmu:

```
BOOLEAN IstniejeBS(int x[],int n, int m)
```

```
{
  int L=0,P=n-1,S;
  while (L<=P)
  {
    S = (L+P)/2;
    if (x[S] == m)
      return(TRUE);
    else if (x[S]<m)
      L = S+1;
    else
      P = S-1;
  }
  return(FALSE)
}
```

Wszelki komentarz jest tu chyba zbędny. Nasza procedura działa w czasie rzędu $O(\log_2(n))$. Omówiony algorytm nazywany jest binarnym przeszukiwaniem lub przeszukiwaniem połówkowym. Wprowadzimy teraz kolejne pojęcie, którą wykorzystywane jest do badania asymptotycznego zachowania funkcji.

Definicja 2.4 *Niech* $f, g : \mathbb{N} \mapsto \mathbb{R}^{>0}$. *Wtedy*

$$f = o(g) \leftrightarrow \lim_{n\to\infty} \frac{f(n)}{g(n)} = 0,$$

Zależność $f = o(g)$ interpretujemy następująco: funkcja f jest asymptotycznie mniejsza niż funkcja g. Z twierdzenia 2.1 wynika, że jeśli $f = o(g)$ to i również $f = O(g)$. Odwrotna implikacja nie jest prawdziwa, na przykład jeśli $f = g > 0$ to $f = O(g)$ ale nie jest prawdą, że $f = o(g)$.

Zamiast pisać $f = o(g)$ będziemy przez chwilę stosować zapis $f \ll g$. Łatwo sprawdzić, że prawdziwe są następujące zależności:

$$\ldots \ll \sqrt[4]{n} \ll \sqrt[3]{n} \ll \sqrt[2]{n} \ll n \ll n^2 \ll n^3 \ll \ldots$$

Do porównywania tempa wzrostu funkcji przydaje się następujące twierdzenie z Analizy Matematycznej:

Twierdzenie 2.6 (Reguła de l'Hospitala) *Załóżmy, że f i g są funkcjami różniczkowalnymi takimi, że $lim_{x\to\infty} f(x) = \infty$ oraz $lim_{x\to\infty} g(x) = \infty$. Wtedy*

$$\lim_{n\to\infty} \frac{f(n)}{g(n)} = lim_{n\to\infty} \frac{f'(n)}{g'(n)}.$$

Twierdzenia tego nie będziemy dowodzić - zostawmy sobie tę przyjemność na wykład z Analizy Matematycznej.

Wniosek 2.2 $(\forall \alpha > 0)(\ln_2(n) = o(n^\alpha))$

Dowód. Niech $\alpha > 0$. Funkcje $f(x) = log_2(x)$ oraz $g(x) = x^\alpha$ spełniają oba założenia twierdzenia de l'Hospitala. Zatem

$$\lim_{x\to\infty} \frac{\log_2(x)}{x^\alpha} = \lim_{x\to\infty} \frac{\frac{1}{x\ln(2)}}{\alpha x^{\alpha-1}} = \lim_{x\to\infty} \frac{1}{\alpha \ln(2) x^\alpha} = 0.$$

□

Funkcja $\log_2(n)$ rośnie więc znacznie wolniej od dowolnego pierwiastka z liczby n:

$$\log_2(n) \ll \ldots \ll \sqrt[4]{n} \ll \sqrt[3]{n} \ll \sqrt[2]{n} \ll n.$$

Widzimy więc, że liczba kroków wykonywanych przez algorytm przeszukiwania binarnego jest rzeczywiście bardzo mała w stosunku do rozmiaru przeszukiwanej tablicy.

2.7 Ćwiczenia i zadania

Ćwiczenie 2.1 *Napisz program który mnoży dwie długie liczby naturalne pamiętane jako tablice cyfr. Zastosuj zbudowane procedury do wyznaczania liczby* $100!$ *Sprawdź, czy otrzymasz odpowiedź*
100! = 93 326 215 443 944 152 681 699 238 856 266 700 490 715 968 264 381 621 468 592 9 63 895 217 599 993 229 915 608 941 463 976 156 518 286 253 697 920 827 223 758 2 51 185 210 916 864 000 000 000 000 000 000 000 000
Wyznacz liczbę 1000!.

Ćwiczenie 2.2 *Oprogramuj rozwiązywanie równania liniowego* $a \cdot x + b = c$*, gdzie* $a, b, c \in \mathbb{R}$*.*

Ćwiczenie 2.3 *Oprogramuj pełen algorytm rozwiązywania układu dwóch równań liniowych z dwoma zmiennymi, który badać będzie nieoznaczoność oraz sprzeczność układu.*

Ćwiczenie 2.4 *Napisz program który służy do rozwiązywania równań kwadratowych.*

Ćwiczenie 2.5 *Napisz funkcję która sprawdza, czy dana liczba naturalna jest pierwsza. Ma ona poprawnie obsługiwać również przypadek liczb 0 i 1.*

Ćwiczenie 2.6 *Niech* $\pi(n)$ *oznacza ilość liczb pierwszych mniejszych lub równych od liczby* n*. Napisz program który wyznacza wartości funkcji* $\pi(n)$ *dla n = 10 000, 20 000, 30 000,. . . , 1 000 000. Porównaj wartości tej funkcji z wartościami funkcji* $f(n) = \frac{n}{\ln n}$*.*

Ćwiczenie 2.7 *a. Parę liczb pierwszych* (p, q) *nazywamy bliźniakami, jeśli* $q = p+2$*. Ile jest par bliźniaków mniejszych od 1 000 000?*
b. Liczby pierwsze postaci p, p+2, p+6, p+8 nazywane są czworakami. Znajdź wszystkie czworaki mniejsze od 1000.
c. Znajdź wszystkie liczby pierwsze palindomiczne mniejsze od 1000.

Ćwiczenie 2.8 * *Napisz program rozwiązujący w liczbach całkowitych równanie* $aX + bY = c$*.*

Ćwiczenie 2.9 *Niech* $NWW(x, y)$ *oznacza najmniejszą wspólna wielokrotność dwóch niezerowych liczb całkowitych, czyli* $\min\{k \in \mathbb{N} \setminus \{0\} : x|k \wedge y|k\}$*. Korzystając ze wzoru* $NWW(x, y) = \frac{x \cdot y}{NWD(x,y)}$ *napisz funkcję wyznaczającą NWW.*

Ćwiczenie 2.10 *Przetestuj działanie procedury sortowania ciągu liczb naturalnych metodą prostego wstawiania. Zbadaj jej czas wykonania dla najgorszego przypadku dla n =* 10^4*,* $2 \cdot 10^4$*, . . . ,* 10^5*.*

Ćwiczenie 2.11 *Przetestuj metodę przeszukiwania binarnego posortowanej tablicy liczb naturalnych.*

Ćwiczenie 2.12 *Dwa wyrazy nazywamy anagramami jeśli drugi można otrzymać z pierwszego przez przestawienie kolejności liter, na przykład „kanonada" i „anakonda", „sekret" i „kretes". Napisz funkcję która sprawdza, czy dane dwa łańcuchy są anagramami. Oszacuj jej złożoność obliczeniową.*

Zadanie 2.1 *Oszacuj liczbę cyfr rozwinięcia dziesiętnego liczby* $1000!$. *Możesz skorzystać ze wzoru Stirlinga* $n! \approx \sqrt{2\pi n}(\frac{n}{e})^n$.

Zadanie 2.2 *Ile końcowych zer ma liczba* $10^6!$*?*

Zadanie 2.3 *Pokaż, że jeśli* $f = O(g)$ *i* $g = O(h)$, *to* $f = O(h)$.

Zadanie 2.4 *Pokaż, że* $a \cdot x^2 + b \cdot x + c = \Theta(x^2)$.

Zadanie 2.5 *Pokaż, że jeśli pętla wykonywana* $aN + b$ *razy (*a *i* b *są stałe) i wewnątrz pętli wykonywanych jest* $O(N)$ *operacji, to cała pętla jest wykonywana w czasie* $O(N^2)$.

Zadanie 2.6 *Czy* $2^{n+1} = O(2^n)$*? Czy* $2^{2n} = O(2^n)$*?*

Zadanie 2.7 *Niech* f *i* g *będą funkcjami o dodatnich wartościach. Pokaż, że* $\max\{f, g\} = \Theta(f + g)$.

Zadanie 2.8 *Uporządkuje według tempa wzrostu następujące funkcje:* $(3/2)^n$, $\sqrt{2}^{\log(n)}$, $\log(n^2)$, n^2, $(\log n)!$, n^3, $\log^2(n)$, $\log n!$, 2^{2^n}, $n^{1/logn}$, $\log\log n$, $n \cdot 2^n$, $n^{\log\log n}$, $\log n$, 2^n, $2^{\log n}$, $(\log n)^{\log n}$, $4^{\log n}$, $(n+1)!$, $\sqrt{\log n}$, $n!$, n, $n \log n$, 1.

Zadanie 2.9 *Przetestuj działanie algorytmu Euklidesa dla wyznaczenia następujących liczb:* $NWD(24, 140)$, $NWD(1234, 4321)$ *i* $NWD(2^{60}, 2^{60} + 2)$.

Zadanie 2.10 *Pokaż, że* $NWW(x, y) = \frac{x \cdot y}{NWD(x,y)}$ *dla niezerowych liczb całkowitych* x *i* y.

Zadanie 2.11 *Załóż, że w ciągu jednej sekundy potrafisz przeanalizować jeden milion permutacji zbioru liczb* $\{1, 2, \ldots, 25\}$. *Ile czasu zajmie Ci przeanalizowanie wszystkich permutacji?*

Zadanie 2.12 *Przypuśćmy, że wszystko uległo spowolnieniu milion razy. Ile czasu zajmie Twojemu komputerowi wykonanie jednej instrukcji? Ile czasu zajmie wystukanie na klawiaturze własnego nazwiska?*

Rozdział 3

Bity, bajty, liczby

```
int i = 0;
while (n){ n = n & (n-1); i++;}
```

Systemy pozycyjne służą do zapisywania liczb. W systemie pozycyjnym o podstawie r każda liczba naturalna jest reprezentowana za pomocą skończonego ciągu

$$(d_k, d_{k-1}, \ldots, d_1, d_0)$$

w którym każdy element d_i jest liczbą całkowitą spełniającą warunek $0 \leq d_i < r$. Ciąg ten służy do przedstawienia liczby

$$N = d_0 + d_1 r + d_2 r^2 + \ldots + d_k r^k.$$

Liczba ta oznaczana jest przez $(d_k d_{k-1} \ldots d_1 d_0)_r$. Indeks r oraz nawiasy są pomijane w przypadku $r = 10$.

Ludzkość stosowała różne wartości r do opisywania liczb. Babilończycy stosowali układ o podstawie 60 a Majowie układ o podstawie 20. Obecnie najczęściej stosowane są liczby $r = 10$ - metoda ta pochodzi od arabów, i $r = 2$ -system dwójkowy stosowany w informatyce. Układ dwójkowy stosowany jest do wewnętrznej reprezentacji liczb naturalnych. W większości języków programowania stosuje się, oprócz układu dziesiętnego, układ o podstawie $r = 16$ oraz czasami o podstawie $r = 8$.

Uwaga. Wybór podstawy dziesiętnej jako najbardziej powszechnej metody zapisywania liczb jest wynikiem przypadkowych zbiegów historycznych. Król szwedzki Karol XII w pierwszej połowie XVIII wieku zamierzał wprowadzić do powszechnego użytku system o podstawie 8, lecz zginął w bitwie kilka dni przed zadekretowaniem swego genialnego pomysłu. B. Pascal pisał: „*System dziesiętny został ustanowiony, skądinąd dość nierozumnie, zgodnie z ludzkim przyzwyczajeniem, s nie z naturalnej konieczności, jak większość byłaby skłonna sądzić.*" i twierdził, że należy posługiwać się systemem o podstawie 20.

Każdy system pozycyjny ma swoje wady i zalety. W systemie o podstawie r mnożenie liczb przez liczbę r jest bardzo proste: polega na dopisaniu jednego zera z prawej strony rozwinięcia liczby. Rzeczywiście:

$$(d_k, \ldots, d_1 0)_r = \sum_{i=1}^{k} d_i r^i = r \sum_{i=0}^{k-1} d_i r^{i-1} = r \cdot (d_k, \ldots, d_1)_r.$$

Przed przyjrzeniem się własnościom rozwinięć w różnych układach wprowadzimy kilka pomocniczych oznaczeń, które służyć będą do uporządkowania dyskusji.

Definicja 3.1 *Niech r i k będą dwoma dodatnimi liczbami naturalnymi.*

1. $\mathbb{C}_r^k = \{(d_{k-1}, d_{k-2}, \ldots, d_1, d_0) : 0 \leq d_i < r\}$,
2. $\mathbb{N}_r^k = \{0, 1, \ldots, r^k - 1\}$,
3. $\mathbb{U}_r(d_k, d_{k-1}, \ldots, d_1, d_0) = (d_k d_{k-1} \ldots d_1 d_0)_r$.

Przypomnijmy sobie teraz następujący wzór:

$$1 + r + r^2 + \ldots + r^{n-1} = \frac{r^n - 1}{r - 1}.$$

Jest on prawdziwy dla dowolnej liczby $r \neq 1$ i udowodnić go można, na przykład, indukcją matematyczną.

Wniosek 3.1 $(\forall r, k \in \mathbb{N} \setminus \{0\})(\mathbb{U}_r : \mathbb{C}_r^k \longrightarrow \mathbb{N}_r^k)$

Dowód. Niech $d = (d_{k-1}, d_{k-2}, \ldots, d_1, d_0) \in \mathbb{C}_r^k$. Wtedy

$$0 \leq \mathbb{U}_r(d) = \sum_{i=0}^{k-1} d_i r^i \leq (r-1) \cdot \sum_{i=0}^{k-1} r^i = (r-1)\frac{r^k - 1}{r - 1} = r^k - 1.$$

□

Metoda reprezentowania liczb w każdej podstawie $r > 0$ jest poprawna. Gwarantuje to następujący wynik:

Lemat 3.1 $(\forall r, k \in \mathbb{N} \setminus \{0\})(\forall n \in \mathbb{N}_r^k)(\exists b \in \mathbb{C}_r^k)(n = \mathbb{U}_r(b))$

Dowód. Dla $k = 1$ teza jest oczywista. Załóżmy więc, że teza jest prawdziwa dla liczby k i niech $a \in \mathbb{N}_r^{k+1}$, czyli niech $0 \leq a < r^{k+1}$. Niech $a = br + q$, gdzie $0 \leq q < r$. Wtedy $0 \leq b < r^k$. Zatem, z założenia indukcyjnego wynika, że istnieje $d = (d_{k-1} \ldots d_0) \in \mathbb{C}_r^k$ takie, że $b = \mathbb{U}_r^k(d)$. Wtedy

$$(d_{k-1} \ldots d_0 q)_r = r \cdot (d_{k-1} \ldots d_0)_r + q = rb + q = a.$$

□

Sformułujemy teraz pewną ogólną obserwację o zbiorach skończonych, z której wyniknie druga ważna własność rozwinięć w ustalonym systemie pozycyjnym.

Twierdzenie 3.1 *Załóżmy, że X i Y są zbiorami skończonymi o tej samej ilości elementów. Niech $f : X \to Y$. Wtedy następujące dwa zdania są równoważne:*

1. *f jest funkcją różnowartościową, czyli $(\forall a, b \in X)(f(a) = f(b) \to a = b)$;*
2. *f jest funkcją „na", czyli $(\forall y \in Y)(\exists x \in X)(f(x) = y)$.*

Dowód tego twierdzenia pozostawimy jako ćwiczenie czytelnikowi - udowodnić je można stosując Zasadę Indukcji Matematycznej.

Wniosek 3.2 $(\forall r, k \in \mathbb{N} \setminus \{0\})(\forall n \in \mathbb{N}_r^k)(\exists! b \in \mathbb{C}_r^k)(n = \mathbb{U}_r(b))$

Dowód. Zbiory C_r^k oraz N_r^k mają po r^k elementów. Lemat 3.1 głosi, że U_r^k jest odwzorowaniem „na". Teza wynika więc z Twierdzenia 3.1. □

Dowód Lematu 3.1 można bezpośrednio przekształcić na algorytm wyznaczania reprezentacji danej liczby n w układzie pozycyjnym o podstawie r. Wynik zapisany jest do tablicy o nazwie s.

```
void UInt2Str(unsigned int n, int r, char s[])
{
  const char ZNAKI[] = "0123456789ABCDEF";
  int i=0;
  if (n==0)
     s[i++]=0;
  else
  while (n)
  {
     s[i++] = ZNAKI[n % r];
     n = n / r;
  }
  s[i] = '\0';
  strev(s);
}
```

Ostatnia linijka omawianej funkcji służy do odwrócenia kolejności znaków w wygenerowanym łańcuchu. Korzystamy w niej funkcji omówionej w rozdziale pierwszym.

3.1 Układ dwójkowy

Podstawową jednostką informacji współczesnych komputerów (wrzesień 2003) jest *bit*. Przyjmować on może dwa stany, z których jeden umownie nazywany jest jedynką a drugi zerem. Bity mogą być być traktowane jako „prawda" i „fałsz", „tak" i „nie", „włączone" i „wyłączone" itd. Słowo „bit" jest skrótem słów „*binary digit*". Bity przechowywane są w pamięci komputera z reguły w małych kondensatorach. Jeśli jest on naładowany, to odpowiadający mu bit przyjmuje wartość 1. Mówimy również wtedy, że bit jest ustawiony. Jeśli kondensator nie ma ładunku to odpowiadający bit ma wartość 0. Mówimy wtedy, że bit jest zresetowany lub wyłączony.

Pamięć komputera podzielona jest na małe komórki zawierające osiem bitów. Układ ośmiu bitów nazywamy *bajtem*. Każda z tych komórek posiada w komputerze swój jednoznaczny numer. Nazywamy go adresem komórki.

W języku C podstawowe typy danych zorganizowane są jako ciągłe sekwencje bajtów. Język C nie determinuje ilości bajtów zajmowanych przez te typy. Zależą one od maszyny i kompilatora. Przestrzegana jest jest jedna zasada: ℂ

$$rozmiar(char) \leq rozmiar(shortint) \leq rozmiar(int) \leq rozmiar(longint)$$

Jest wielce prawdopodobne, że na Twojej maszynie rozmiary te wynoszą 1, 2, 4 i 4 bajty. Ich wersje bez znaku (unsigned) kodują więc liczby z przedziałów $0 \div 2^8 - 1$, $0 \div 2^{16} - 1$ i $0 \div 2^{32} - 1$, czyli z przedziałów $0 \div 255$, $0 \div 65\,535$ oraz $0 \div 4\,294\,967\,295$.

Zakresy zmiennych na Twoim komputerze możesz ustalić bez eksperymentów komputerowych. Określone są one w pliku bibliotecznym limits.h. Na przykład, maksymalną liczbę typu unsigned int definiuje w nim stała UINT_MAX.

Przez $P(X)$ oznaczamy zbiór potęgowy zbioru X, czyli rodzinę wszystkich podzbiór zbioru X. Do zbioru tego należy zbiór pusty $\emptyset$ oraz cały zbiór X. Jeśli zbiór X ma n elementów, to zbiór $P(X)$ ma 2^n elementów. Istnieje naturalna odpowiedniość pomiędzy elementami zbioru $\mathbb{C}_2^k$ a podzbiorami zbioru $\{0,\ldots,k-1\}$. Jest ona określona funkcją

$$set : \mathbb{C}_2^k \to P(\{0,\ldots,k-1\}) : (b_{k-1}\ldots b_0) \mapsto \{i < k : b_i = 1\}.$$

Na przykład, $set(00000000) = \emptyset$ oraz $set(11111111) = \{0,1,2,3,4,5,6,7\}$. Bez trudu pokazujemy, że odpowiedniość ta jest wzajemnie jednoznaczna, czyli, że

$$(\forall k > 0)((set : \mathbb{C}_2^k \xrightarrow[na]{1-1} P(\{0,\ldots,k-1\}).$$

Przy ustalonym k, jeśli $set(b) = A$, to ciąg b nazywamy mapą bitową zbioru A.

Język C posiada mechanizmy służące do operowania na poszczególnych bitach. Są to operacje `&`, `|`, `~`, `^`, `<<` i `>>`. Operacja `&` jest operacją koniunkcji bitowej. Operacja `|` jest alteratywą bitową zaś `~` jest negacją bitową. Znaczenie tych operacji ilustrować będziemy dla typu unsigned char, który utożsamiamy ze zbiorem B_8: $\mathbb{C}$

$$(b_7,\ldots,b_0)\&(c_7,\ldots,c_0) = (\min(b_7,c_7),\ldots,\min(b_0,c_0)),$$

$$(b_7,\ldots,b_0)|(c_7,\ldots,c_0) = (\max(b_7,c_7),\ldots,\max(b_0,c_0)),$$

$$\sim(b_7,\ldots,b_0) = (1-b_7,\ldots,1-b_0).$$

Operacja `^` jest bitową wersją operatora logicznego XOR i jej znaczenie określa formuła

$$b \text{ ^ } c = ((\sim b) \text{ \& } c) \text{ | } (b \text{ \& } (\sim c)).$$

Operator ten ma również nazwę rozłącznej alternatywy lub dyzjunkcji. W logice jest on oznaczany symbolem $\oplus$. Mamy zatem

$$b \oplus c \leftrightarrow (\neg p \wedge q) \vee (p \wedge \neg q).$$

Bez trudu - na przykład metodą tabel zero-jedynkowych - można pokazać, że

$$b \oplus c \leftrightarrow (p \vee q) \wedge \neg(p \wedge q).$$

Operator « służy do przesunięcia w lewą stronę ciągu bitów. Na przykład

00011111 « 2 = 01111100,
00000001 « 4 = 00010000.

Po przesunięciu w lewą stronę ciągu bitów o i pierwsze i bitów z prawej strony zostaje wypełnionych zerami. Operator » służy do przesunięcia w prawą stronę ciągu bitów. Na przykład

11110000 » 2 = 00111100,
10000000 » 4 = 00001000.

Po przesunięciu w lewą stronę ciągu bitów o i pierwsze i bitów z lewej strony zostaje wypełnionych zerami.

Przykład 3.1 *Operację przesunięcia bitów można wykorzystać do zapalania konkretnego bitu:*

```
void ZapalBit(unsigned int *n, int i)
{
  *n = *n | (1 « i);
}
```

Przykład 3.2 *Operację przesunięcia bitów można wykorzystać również do gaszenia konkretnego bitu:*

```
void ZgasBit(unsigned int *n, int i)
{
  *n = *n & (~ (1 « i));
}
```

Jedną z najbardziej naturalnych własności dowolnego zbioru jest liczba jego elementów, zwana również mocą zbioru. Oznacza się ją symbolem $|X|$ (w każdej sytuacji będzie jasne, czy mówimy o wartości bezwzględnej, czy też o mocy). Jeśli b jest mapą bitową zbioru X, to $|X|$ jest równa liczbie jedynek zapalonych w mapie b. Liczbę zapalonych bitów w zmiennej n typu long int trudu można wyznaczyć za pomocą następujące pętli:

```
int ile=0; for (i=0;i<32;i++) if (n & (1 « i)) ile++;
```

(uwaga: założyliśmy, że typ long int jest realizowany za pomocą czterech bajtów). Pętla ta będzie zawsze wykonywana 32 razy, bez względu na to ile jest zapalonych bitów w liczbie n. Jeśli spodziewamy się, że liczba zapalonych bitów jest mała, to możemy do tego zadania podejść trochę inaczej. Zauważmy najpierw, że

$$(100000)_2 - 1 = (011111)_2.$$

Zatem w układzie dwójkowym odjęcie liczby 1 od niezerowej liczby n polega na wyznaczeniu pierwszego, od lewej strony, zapalonego bitu, zgaszeniu go i zapaleniu wszystkich wcześniejszych bitów. Na przykład

$$40 - 1 = (101000)_2 - 1 = (100111)_2 = 39.$$

Zatem zgaszenie ostatniego bitu liczby n możemy osiągnąć przez zastąpienie liczby n liczbą $n\&(n-1)$. Oto kod funkcji wyznaczającej moc mapy bitowej n, która oparta jest na tej obserwacji:

```
int Moc(unsigned int n)
{
  int ile=0;
  while (n)
  {
```

```
    n = n & (n-1);
    ile++;
  }
  return(ile);
}
```

Przykład 3.3 *Przy pomocy operacji przesuwania bitów oraz dodawania można napisać szybki algorytm mnożenia liczb całkowitych. Oto jego kod:*

```
int c = 0;
for (i = 15; i >= 0; i–)
if ((b » i) & 1)
  c += (a « i);
```

3.2 Układ szesnastkowy

W układzie szesnastkowym, czyli w układzie o podstawie 16, do reprezentowania cyfr 10, 11, 12, 13, 14 i 15 używamy liter A, B, C, D, E i F. Warto zapamiętać, że $(F)_{16} = 2^4 - 1 = 15$, $(FF)_{16} = 2^8 - 1 = 255$, $(FFFF)_{16} = 2^{16} - 1 = 65\,535$ oraz $(FFFFFFFF)_{32} = 2^{32} - 1 = 4\,294\,967\,295$.

Stałe liczbowe w języku C można zapisywać w postaci szesnastkowej. Należy je $\mathbb{C}$
w tym celu poprzedzić ciągiem 0X. Na przykład $0XFA = 16 * F + A = 16 * 15 + 10 = 250$. Funkcję printf można nakłonić do drukowania zmiennych całkowitych w układzie szesnastkowym. W tym celu deklarację zmiennej należy poprzedzić ciągiem $0X$. Na przykład, jeśli zmienna całkowita n ma wartość 250, to polecenie printf("%0Xd",n) wyprowadzi napis FA.

3.3 Liczby ze znakiem

Do tej pory omawialiśmy tylko metody reprezentowania liczb naturalnych w różnych układach pozycyjnych. Istnieje kilka metod reprezentowania liczb całkowitych w komputerach. Najbardziej dziś powszechną jest metoda „drugiego uzupełnienia". Oparta jest ona o następującą funkcję:

Definicja 3.2 *Dla* $b = (b_{k-1} \dots b_0) \in \mathbb{C}_2^k$ *określamy*

$$\mathbb{S}_2(b) = (\sum_{i=1}^{k-2} b_i 2^i) - b_{k-1} 2^{k-1}.$$

Dla ciągów $b \in \mathbb{C}_2^k$ takich, że $b_{k-1} = 0$ mamy $\mathbb{S}_2(b) = \mathbb{U}_2(b)$. Jednak jeśli $b_{k-1} = 1$ mamy $\mathbb{S}_2(b) = \mathbb{U}_2(b) - 2^k$. Aby zobaczyć wyraźniej jak zachowuje się ta funkcja ustalmy liczbę $n = 3$ i przyjrzyjmy się wszystkim wartościom funkcji $\mathbb{S}_2$ oraz $\mathbb{U}_2$ na ciągach trzech bitów:

$\bar{b}$	$\mathbb{U}_2(\bar{b})$	$\mathbb{S}_2(\bar{b})$
000	0	0
001	1	1
010	2	2
011	3	3
100	4	-4
101	5	-3
110	6	-2
111	7	-1

Widzimy, że funkcja $\mathbb{S}_2$ przyjmuje wartości ze zbioru $\{-4, \ldots, 3\}$. Obserwację tę możemy uogólnić dla dowolnej liczby bitów:

Twierdzenie 3.2 $(\forall k > 1)(\mathbb{S}_2 : \mathbb{C}_2^k \xrightarrow[na]{1-1} \{-2^{k-1}, \ldots, 2^{k-1} - 1\})$.

Dowód. Ustalmy liczbę $k > 1$. Niech $B_0 = \{(0b_{k-2} \ldots b_0) : b_i \in \{0, 1\}\}$ oraz $B_1 = \{(1b_{k-2} \ldots b_0) : b_i \in \{0, 1\}\}$. Wtedy

$$\mathbb{S}_2(\mathbb{C}_2^k) = \mathbb{S}_2(B_0) \cup \mathbb{S}_2(B_1)$$

więc

$$\mathbb{S}_2(\mathbb{C}_2^k) = \{0, \ldots, 2^{k-1} - 1\} \cup \{0 - 2^{k-1}, \ldots, 2^{k-1} - 1 - 2^{k-1}\}.$$

Zatem $(\mathbb{S}_2 : \mathbb{C}_2^k \xrightarrow{na} \{-2^{k-1}, \ldots, 2^{k-1} - 1\}$. Zauważmy następnie, że

$$|\{-2^{k-1}, \ldots, 2^{k-1} - 1\}| = 2^k,$$

więc teza twierdzenia wynika z Twierdzenia 3.1. □

W poniższej tabeli znajduje się zestawienie zakresów typów całkowitych, przy założeniu, że char jest realizowany za pomocą 1 bajta, short int za pomocą 2 bajtów a typ int oraz long int za pomocą 4 bajtów.

Typ	min	max	min (hex)	max (hex)
(signed) char	-128	127	-80	7F
unsigned char	0	255	0	FF
(signed) short int	-32768	32767	-8000	7FFF
unsigned short int	0	65535	0	FFFF
(signed) int	-2147483648	2147483647	-80000000	7FFFFFFF
unsigned int	0	4294967295	0	FFFFFFFF
(signed) long	-2147483648	2147483647	-80000000	7FFFFFFF
unsigned long	0	4294967295	0	FFFFFFFF

Omówiony sposób reprezentowania liczb całkowitych może sprawiać wrażenie niezwykle dziwnego. Pierwsze metody reprezentowania liczb całkowitych polegały na poświęceniu najbardziej znaczącego bitu na znak. Wadą tej metody była niejednoznaczność zera: mogło być one reprezentowane przez $+0000000$ oraz -0000000. Omówiona metoda „drugiego uzupełnienia" nie ma tej wady. Druga sprawa związana jest z odejmowaniem. Okazuje się, że odejmowanie liczb realizuje się w tej metodzie

znacznie prościej.

Dodawanie liczb całkowitych (typu signed) odbywa się w następujący sposób: bierzemy dwie liczby, powiedzmy a i b, traktujemy je jako liczby bez znaku; dodajemy je bez zwracania uwagi na przepełnienie (odpowiada to dodawaniu modulo 2^k; wynik interpretujemy jako liczbę ze znakiem.

Prześledźmy ten proces na sztucznym przykładzie, mianowicie dla $k = 3$. Niech $a = -3$ oraz $b = 2$. Wtedy $a = \mathbb{S}_2(101)$, $b = \mathbb{S}_2(010)$; następnie $(101)_2 + (010)_2$ = $5 + 2$ = 7 = $(111)_2$; w końcu $\mathbb{S}_2(111) = -1$. Zatem $-3 + 2 = -1$. Czyli wszystko jest w porządku. Sprawdźmy jednak co się dzieje, gdy $a = 3$ i $b = 1$. Ponieważ $3 = \mathbb{S}_2(011)$ i $1 = \mathbb{S}_2(001)$ oraz $3 + 1 = 4 = (100)_2$ więc $3 + 1 = \mathbb{S}_2(100) = -4$. **CZYLI** $3 + 1 = -1$**!**

Eksperymenty te powinny nam uzmysłowić, że z liczbami całkowitymi należy obchodzić się bardzo ostrożnie. Jeśli chcesz dodać do siebie dwie liczby typu int a ich suma przekracza stałą INT_MAX z pliku limits.h, to wynik będzie liczbą ujemną! Pamiętaj o następujących łatwych do zapamiętania regułach:

1. jeśli suma dwóch dodatnich liczb tego samego typy całkowitego nie przekracza odpowiedniej dla typu stałej MAX, to wynik jest poprawny.

2. jeśli suma dwóch ujemnych liczb tego samego typy całkowitego nie przekracza odpowiedniej dla typu stałej MIN, to wynik jest poprawny.

3. suma liczby dodatniej i ujemnej tego samego typy całkowitego jest zawsze poprawna.

3.4 Liczby rzeczywiste

Liczby rzeczywiste, oznaczany przez $\mathbb{R}$, w sposób istotny różnią się od liczb całkowitych. Podstawowa różnica pomiędzy nimi to ich moc. Zbiór liczb całkowitych jest zbiorem przeliczalnym. Zbiór liczb rzeczywistych jest zbiorem nieprzeliczalnym. Ma istotnie więcej elementów niż liczby całkowite. Fakt ten ma swoje odbicie w reprezentacji liczb rzeczywistych w komputerach. Podstawową porządkową własnością liczb rzeczywistych jest *zupełność*:

> Każdy niepusty i ograniczony z góry podzbiór zbioru zbioru liczb ma kres górny.

Przez kres górny zbioru A rozumiemy najmniejszą taką liczbę rzeczywistą g, że $(\forall a \in A)(a \leq g)$. Kres górny zbioru A oznaczamy przez $\sup(A)$. Z własności tej wynika następujące twierdzenie które ma kluczowym znaczenie dla naszych dalszych rozważań:

Twierdzenie 3.3 *Dla każdej liczby $x \in \mathbb{R}$ i dla każdej liczby naturalnej $r > 1$ istnieje liczba całkowita k oraz ciąg $(b_k)_{n\geq 1} \in \{0, \ldots r-1\}^{\mathbb{N}\setminus\{0\}}$ taki, że*

$$x = k + \sum_{i=1}^{\infty} \frac{b_i}{r^i}.$$

Liczbę $\sum_{i=1}^{\infty} \frac{b_i}{r^i}$ oznaczamy, przez analogię do rozwinięć liczb całkowitych w układach pozycyjnych o podstawie r przez $(0.b_1 b_2 \ldots)_r$.

Dowód. Liczbę k można wyznaczyć bardzo prosto, jest nią bowiem $sup\{n \in \mathbb{N} : n \leq x\}$. Rozważać od tej pory będziemy liczbę $y = x - k$. Oczywiście $0 \leq y < 1$. Zdefiniujemy indukcyjnie dwa ciągi $(y_n)_{n\geq 0}$ oraz $(b_n)_{n\geq 1}$. Przyjmiemy $y_0 = y$ oraz

$$b_{n+1} = \max\{k \in \mathbb{N} : \frac{k}{r} \leq y_n\} \tag{3.1}$$

$$y_{n+1} = r \cdot y_n - b_{n+1} \tag{3.2}$$

Pokażemy najpierw, że $0 \leq y_n < 1$ dla każdej liczby n. Teza ta jest prawdziwa dla $n = 0$, gdyż $y_0 = y$. Załóżmy więc, że jest ona prawdziwa dla liczby n. Wtedy

$$\frac{b_{n+1}}{r} \leq y_n < \frac{b_{n+1} + 1}{r},$$

zatem $b_{n+1} \leq r \cdot y_n < b_{n+1} + 1$, czyli $0 \leq r \cdot y_n - b_{n+1} < 1$.

Pokażemy teraz, że

$$y_n = r^n (y - \sum_{i=1}^{n} \frac{b_i}{r^i}) \tag{3.3}$$

dla $n > 0$. Dla $n = 1$ mamy $y_1 = r \cdot y_0 - b_1 = r^1 \cdot (y - b_1/r)$. Załóżmy, że wzór 3.3 jest prawdziwy dla liczby n. Wtedy

$$y_{n+1} = r \cdot y_n - b_{n+1} = r \cdot r^n \cdot (y - \sum_{i=1}^{n} \frac{b_i}{r^i}) - b_{n+1} =$$

$$r^{n+1}(y - \sum_{i=1}^{n} \frac{b_i}{r^i} - \frac{b_{n+1}}{r^{n+1}}) = r^{n+1}(y - \sum_{i=1}^{n+1} \frac{b_i}{r^i}).$$

Ze wzoru 3.3 wynika, że

$$\sum_{i=1}^{n} \frac{b_i}{r^i} = y - \frac{y_n}{r^n}$$

Wiemy również, że $0 \leq y_n < 1$, z czego wynika, że $\lim_{n\to\infty} \frac{y_n}{r^n} = 0$, więc

$$\sum_{i=1}^{\infty} \frac{b_i}{r^i} = \lim_{n\to\infty} (y - \frac{y_n}{r^n}) = y,$$

co kończy dowód. □

Przedstawiony dowód łatwo można przerobić na algorytm. Oto kod funkcji, która wyznacza dokładnie kilkadziesiąt pierwszych cyfr rozwinięcia dwójkowego liczby $x \in [0, 1)$:

```
void D2Bin(double x, int c[], int max)
{
  int i=0;
  while (i<max)
  {
     if (x<0.5)
```

```
        c[i] = 0;
      else
        c[i] = 1;
      x = x*2-c[i++];
  }
}
```

Uwaga. Instrukcję if (x<0.5) c[i]=0; else c[i]=1; można zapisać w sposób nieco bardziej zwarty, korzystając z wyrażeń warunkowych języka C. Są to wyrażenia postaci ℂ

W1 ? W2 : W3

które wylicza się następująco: obliczmy wartość W1; jeśli jest ona różna od zera, to oblicza się wartość wyrażenia W2 i ona staje się wartością całego wyrażenia; jeśli W1 miało wartość 0, to oblicza się wartość wyrażenia W3 i ono staje się wartością wyrażania. Zatem bardziej zwartą formą zapisu instrukcji if (x<0.5) c[i]=0; else c[i]=1; jest

c[i] = (x<0.5) ? 0 : 1 ;

Przyznać musisz, że język C pozwala na bardzo zwięzłe zapisywanie kodu!

Oto rozwinięcie liczby $7/2$ w układzie dwójkowym:

$$\frac{7}{2} = (111)_2 \times 2^{-1} = (1.11)_2 \times 2^{-3} = (1 + (0.11)_2) \times 2^{-3}.$$

Liczba $1/10$ ma nieskończone rozwinięcia dwójkowe:

$$\frac{1}{10} = \frac{1}{2^4} + \frac{1}{2^5} + \frac{1}{2^8} + \frac{1}{2^9} + \ldots,$$

czyli

$$\frac{1}{10} = (0.00011001100110011\ldots)_2 = (1.100110011\ldots)_2 \times 2^{-4}.$$

Wiemy już, że liczby rzeczywiste z przedziału $[0, 1)$ mogę być przestawione w postaci nieskończonego ciągu $0.b_1b_2b_3\ldots$, gdzie $b_1 \in \{0, 1\}$ dla wszystkich i. Jeśli $b_i = 0$ dla wszystkich $i > n$, to liczbę $(0.b_1b_2b_3\ldots)_2$ zapisujemy prościej, a mianowicie jako $(0.b_1b_2b_3\ldots b_n)_2$ i wtedy

$$(0.b_1b_2b_3\ldots b_n)_2 = \sum_{i=1}^{n} \frac{b_i}{2^i} = \frac{b_1 2^{n-1} + \ldots b_{n-1}2 + b_n}{2^n} = \frac{(b_1\ldots b_n)_2}{2^n}. \quad (3.4)$$

Reprezentacja zmiennopozycyjna

W chwili obecnej większość komputerów wykorzystuje standard IEEE [1] z roku 1985 do reprezentowania liczb rzeczywistych. Bazuje on na rozwinięciach binarnych liczb rzeczywistych. Zaledwie kilka typów komputerów stosuje rozwinięcia o podstawie 16. Powstały również ongiś komputery stosujące trójkowy układ pozycyjny.

[1] Institute for Electrical and Electronics Engineers

Standard IEEE oparty jest na reprezentacji wykładniczej liczb rzeczywistych. W metodzie tej niezerowe liczby rzeczywiste zapisywane są w postaci

$$\pm M \times 10^E,$$

gdzie $1 \leq M < 10$ oraz E jest liczbą całkowitą. Liczbę M nazywa się częścią znaczącą a E wykładnikiem. Na przykład, wykładniczą reprezentacją liczby 365.25 jest 3.6525×10^2 a reprezentacją liczby 0.000123 jest 1.23×10^{-4}. Każdą niezerową liczbę rzeczywistą możemy zapisać w tej postaci. Przekształcenie liczby do reprezentacji wykładniczej polega na przeniesieniu kropki za pierwszą różną od zera cyfrę. Dlatego ta metoda nazywa się metodą „pływającej kropki" (float point) lub bardziej oficjalnie metodą zmiennopozycyjną. W komputerach podstawa 10 jest zastąpiona zwykle liczbą 2, zatem liczby niezerowe mają reprezentację

$$\pm M \times 2^E,$$

gdzie $1 \leq M < 2$. Zatem część znacząca M ma rozwinięcie dwójkowe

$$M = 1 + (0.b_1 b_2 b_3 \ldots)_2.$$

IEEE liczby pojedynczej i podwójnej precyzji

IEEE liczby pojedynczego formatu - odpowiadające najpewniej typowi float Twojego kompilatora języka C - reprezentowane są za pomocą 32 bitów. Pierwszy bit jest bitem znaku, następne 8 bitów reprezentują wykładnik a pozostałe 23 bity reprezentują część znaczącą:

znak	wykładnik	część znacząca
b_1	$e_1 \ldots e_8$	$m_1 \ldots m_{23}$

Znak plus jest reprezentowany przez zero a minus przez jeden. Dwie wartości wykładnika: 00000000 oraz 11111111 są zarezerwowane dla specjalnych celów. Dla wszystkich pozostałych wykładników liczbą rzeczywistą reprezentowaną przez ten ciąg jest liczba rzeczywista

$$val(s, \overrightarrow{b}, \overrightarrow{m}) = (-1)^s \times (1.mm \ldots mm)_2 \times 2^{(eeeeeeee)_2 - 127}.$$

Przyjrzyjmy się najpierw wykładnikowi. Zauważmy, że $(11111110)_2 = 2*(1111111)_2 = 2 * (2^7 - 1) = 254$ oraz, że 254-127= 127. Zatem największą możliwą wartości wykładnika jest liczba 127. Najmniejszą zaś jest liczba $(00000001)_2$ - 127 = -126. Największą możliwą do zapisania liczbą jest

$$(1.11111111)_2 \times 2^{127} = (2 - 0.00000001) \times 2^{127} \approx 2^{128} \approx 3.4 \times 10^{38}.$$

Najmniejszą możliwą do zapisania liczbą dodatnią jest

$$(1.00000000)_2 \times 2^{-126} = 2^{-126} \approx 1.2 \times 10^{-38}.$$

IEEE liczby podwójnego formatu - odpowiadające najprawdopodobniej typowi double Twojego kompilatora - reprezentowane są za pomocą 64 bitów:

znak	wykładnik	część znacząca
b_1	$e_1 \ldots e_{11}$	$m_1 \ldots m_{52}$

Podobnie jak dla liczb pojedynczej precyzji wartości e = 00000000000 oraz e = 111111111111 są zarezerwowane dla specjalnych celów. Dla wszystkich pozostałych ciągów określona jest wartość

$$val(s, \overrightarrow{b}, \overrightarrow{m}) = (-1)^s \times (1.mm\ldots mm)_2 \times 2^{(eeeeeeee)_2 - 1023}.$$

Liczby tej postaci, to znaczy te w których wykładnik jest różny od 00000000000 oraz od 11111111111 nazywamy **liczbami znormalizowanymi** i na razie będziemy się zajmowali tylko nimi. Łatwo sprawdzić, że największą możliwą wartością wykładnika jest liczba $(11111111110)_2 - 1023 = 1023$ a najmniejszą wartością jest $(00000000001)_2 - 1023 = -1022$. Liczba 1 ma reprezentację

$$1 = (-1)^0 \times (1.00\ldots00)_2 \times 2^{1023-1023} = val(0, \underbrace{00111111111}_{11}, \underbrace{00\ldots00}_{52})$$

Największą możliwą do zapisania liczbą jest

$$(1.1...1)_2 \times 2^{1023} = (2 - 0.0...01) \times 2^{1023} \approx 2^{1024} \approx 1.8 \times 10^{308}.$$

Najmniejszą możliwą do zapisania liczbą dodatnią jest

$$(1.0...0)_2 \times 2^{-1022} = 2^{-1022} \approx 2.2 \times 10^{-308}.$$

Zajmować się będziemy od tej pory liczbami podwójnej precyzji. Niech e będzie najmniejszą liczbę podwójnej precyzji większą od liczby 1. Epsilonem maszynowym ε nazywamy liczbę $e - 1$. Jest nią więc liczba

$$1.0000...001 \times 2^0 - 1 = \frac{1}{2^{52}} \approx 2.22 \times 10^{-16}$$

Zwróć uwagę na to, że liczby rzeczywiste, które mogą być wyrażone za pomocą liczb podwójnej precyzji są dziurawe. Jeśli przez DIEEE oznaczymy zbiór wszystkich liczb, które mogą być wyrażone za pomocą liczb podwójnej precyzji, to

$$DIEEE \cap [1, 2) = \{1, 1 + \frac{1}{2^{52}}, 1 + \frac{2}{2^{52}}, 1 + \frac{3}{2^{52}}, \ldots, 1 + \frac{2^{52} - 1}{2^{52}}\}.$$

W następnym przedziale dziury są dwukrotnie większe:

$$DIEEE \cap [2, 4) = \{2, 2 + \frac{2}{2^{52}}, 2 + \frac{4}{2^{52}}, 1 + \frac{6}{2^{52}}, \ldots, 1 + \frac{2 \cdot 2^{52} - 2}{2^{52}}\}.$$

Jeszcze ciekawiej wygląda sprawa w przedziale $[2^{52}, 2 \cdot 2^{52} - 1)$:

$$DIEEE \cap [2^{52}, 2^{53} - 1) = \{2^{52}, 2^{52} + 1, 2^{52} + 2, \ldots, 2^{53} - 1\}.$$

Dziury w tym przedziale są szerokości 1. W następnym przedziale sprawa wygląda jeszcze gorzej:

$$DIEEE \cap [2^{53}, 2^{54} - 1) = \{2^{53}, 2^{53} + 2, 2^{53} + 4, \ldots, 2^{54} - 1\}.$$

A z tego wynika, że w arytmetyce IEEE podwójnej precyzji zachodzi zaskakująca tożsamość:

$$\mathbf{2^{53} + 1 = 2^{53}}$$

Dokładność obliczeń

Jak już wiemy, te liczby rzeczywiste który mogą być reprezentowane jako IEEE liczby stanowią dyskretny podzbiór zbioru liczb rzeczywistych. Konsekwencją tego są dwa typy błędów które powstawać mogą w trakcie obliczeń.

Pierwszy z nich to **błąd reprezentacji**. Na przykład, liczba $\frac{1}{3}$ ma następujące rozwinięcie w układzie dwójkowym:

$$\frac{1}{3} = 0.01010101010101010101010101\ldots$$

Jeśli ta liczba zostanie przedstawiona jako double, to w pamięci komputera będziemy mieli zapamiętany ciąg

$$[0, \underbrace{00111111111}_{11}, \underbrace{0101\ldots01}_{52}]$$

Ponieważ

$$0.\underbrace{010101\ldots01}_{52} = \frac{1}{4} + \frac{1}{4^2} + \ldots + \frac{1}{4^{26}} = \ldots = \frac{1}{3}(1-\varepsilon)$$

więc liczba 1/3 zapamiętana jest tylko w przybliżeniu. Jego błąd wynosi $(1/3)\cdot\varepsilon$. Warto pamiętać o tym, że liczba 1/10 również nie ma dokładnej reprezentacji w arytmetyce IEEE, gdyż

$$\frac{1}{10} = (0.00011001100110011\ldots)_2.$$

Drugi rodzaj błędów, to **błędy operacji arytmetycznych**. Wiemy już, że wyliczona wartość sumy dwóch zmiennych typu double nie musi być równa ich dokładnej sumie, gdyż może ona nie mieć dokładnej IEEE reprezentacji. Oznaczmy przez $\oplus$, $\ominus$, $\otimes$ i $\oslash$ wyniki IEEE - działań arytmetycznych. Zgodnie z normą IEEE względne błędy tych działań nie mogą być większe od ε, czyli

$$\frac{|x \oplus y - x + y|}{|x+y|} \leq \varepsilon, \ldots, \frac{|x \oslash y - x/y|}{|x/y|} \leq \varepsilon.$$

Zatem

$$x \oplus y = x + y + \delta\cdot(x+y), \ldots, x \oslash y = \frac{x}{y} + \delta\cdot\frac{x}{y}$$

gdzie δ jest pewną liczbą zależną od działania oraz argumentów x i y taką, że $|\delta| \leq \varepsilon$.

Przykład 3.4 *Załóżmy, że chcemy rozwiązać układ równań*

$$\left\{\begin{array}{rcrcr} 10.0x & + & 1.0y & = & 11 \\ 3.0x & + & 0.3y & = & 3.3 \end{array}\right.$$

Jest to nieoznaczony układ równań. Jednak jeśli obliczmy wyznacznik główny tego układu, czyli obliczymy $det = (10.0 \otimes 0.3) \ominus (3.0 \otimes 1.0)$ *w arytmetyce IEEE to otrzymamy liczbę* -0.00000000000000011102*, czyli liczbę różną od zera!. Jest to oczywiście spowodowane błędami reprezentacji. Dalsze obliczenia za pomocą wzorów Cramera dadzą nam wynik* $x = -0.5$ *oraz* $y = 16$*. Widzimy więc, że do rozwiązywania układów równań liniowych powinniśmy podchodzić z dużą ostrożnością. Bezpieczną metodą będzie zastosowanie następującego fragmentu kodu:*

```
det = A*E-D*B;
if (abs(det)<1e-15)
{
  printf("układ jest przypuszczalnie nieoznaczony lub sprzeczny\n");
  return(0);
}
```

Dokładniejszą analizą problemu dokładności obliczeń zapoznasz się na zajęciach z analizy numerycznej. Na razie zapamiętajmy, że błędy działań arytmetycznych propagują się na wyniki obliczeń wartości złożonych z nich funkcji. Warto pamiętać o następującej przybliżonej regule służącej do szacowania błędów:

$$f(x+\delta) \simeq f(x) + \frac{df}{dx}(x) \cdot \delta.$$

Załóżmy, że mamy dane trzy zmienne X, Y i Z. Wtedy

$$(X \oplus Y) \oplus Z = ((X+Y) + (X+Y)\delta_{X+Y}) \oplus Z \simeq$$

$$(X+Y+Z) + (X+Y)\delta_{X+Y} + (X+Y+Z)\delta_{X+Y+Y}.$$

Zatem błąd obliczeń wyrażenia $(X \oplus Y) \oplus Z$ można oszacować przez $(2X+2Y+Z)\varepsilon$. Oto oszacowanie dla czterech zmiennych X, Y, Z i V: $(3X+3Y+2Z+V)\varepsilon$. Z oszacowań tych wynika następujący praktyczny i ważny wniosek:

> Jeśli masz wysumować liczby $a_1, \ldots, a_n$ z możliwie dużą precyzją, to przed sumowaniem zadbaj o to aby $a_1 \leq a_2 \leq \ldots \leq a_n$.

A oto inna praktyczna wskazówka:

> Jeśli masz wysumować liczby $a_1, \ldots, a_n$ i wiesz, że końcowa suma będzie znacznie mniejsza od najwiekszej liczby, to możesz się spodziewać poważnego błędu obliczeń.

3.5 Liczby zespolone

Liczbę zespoloną $a + bi$ interpretować możemy jako pary liczb rzeczywistych (a, b). Język nie posiada typu podstawowego, który służy do reprezentowania liczb zespolonych. Jednak język C posiada mechanizm definiowania obiektów złożonych z kilku zmiennych. Nazywają się one *strukturami*. Wykorzystamy go do samodzielnego zdefiniowania liczb zespolonych. Rozpoczniemy od zdefiniowania struktury: $\mathbb{C}$

W bibliotece liczb zespolonych warto wprowadzić nowy typ, który można by nazwać dcomplex za pomocą polecenia

```
typedef struct {
  double r;
  double i;
} dcomplex;
```

Zmienne występujące w definicji struktury nazywamy *składowymi* struktury. Zmienne typu dcomplex możemy od tej pory deklarować w następujący sposób:

struct dcomplex X,Y,Z;

Możemy również w trakcie deklaracji zmiennej inicjalizować ich wartości. Na przykład

struct dcomplex X = {1.0,1.0};

służy do zadeklarowania zmiennej X i podstawieniu pod nią wartości $1 + i$. Do odwoływania się się do określonej składowej struktury posługujemy się konstrukcją nazwa-struktury.składowa. Zatem podstawienie pod zmienną X typu complex liczby 1+i można by było zrealizować następująco:

X.r = 1.0;
X.i = 1.0;

Funkcje języka C mogą zwracać wartości typu struktur. Możemy więc bez trudu napisać funkcję, która będzie służyła do dodawania oraz mnożenia liczb zespolonych. Musimy w tym celu przypomnieć sobie podstawowe wzory: $(a + bi) + (c + di) = (a + c) + (b + d)i$ oraz $(a + bi) \cdot (c + di) = (ac - bd) + (ad + bc)i$:

```
dcomplex Cadd(dcomplex a, dcomplex b)
{
  dcomplex c;
  c.r = a.r+b.r;
  c.i = a.i+b.i;
  return c;
}

dcomplex Cmult(dcomplex a, dcomplex b)
{
  dcomplex c;
  c.r = a.r*b.r -a.i*b.i;
  c.i = a.i*b.r +a.r*b.i;
  return c;
}
```

W podobny sposób bez większego trudu można napisać pozostałe funkcje służące do operowania na liczbach zespolonych: odejmowanie, sprzężenie, moduł, dzielenie. Narzucającą się metodę wyznaczenia modułu liczby zespolonej $x + iy$ za pomocą wzoru sqrt(x*x+y*y) można zastąpić nieco bardziej dokładną. Otóż analiza błędów działań pokazuje, że jeśli |x|>|y|>0 to bardziej dokładny wynik otrzymamy korzystając ze wzoru

$$||x + iy|| = |x| \cdot \sqrt{1 + (\frac{y}{x})^2}.$$

W bibliotece liczb zespolonych warto wprowadzić nowy typ, który można by nazwać dcomplex za pomocą polecenia

typedef struct {double r,i;} dcomplex;

3.6 Ćwiczenia i zadania

Ćwiczenie 3.1 *Napisz program, który wyświetla minimalne i maksymalne zakresy wszystkich podstawowych typów zmiennych.*

Ćwiczenie 3.2 *Zbadaj zachowanie się funkcji* $S(x) = x+1$ *dla* x *ze zbioru {INT_MAX-10,. . . , INT_MAX+10} dla typu* int*. Zbadaj zachowanie się tej funkcji dla zmiennych typu* unsigned int *dla* x=UINT_MAX*.*

Ćwiczenie 3.3 *Napisz procedurę służącą do znajdowania rozwinięcia zadanej liczby naturalnej reprezentowanej jako* unsigned long *w układzie o zadanej postawie* d*. Możesz założyć, że* $d \in \{2, 3, \ldots, 16\}$

Ćwiczenie 3.4 *Napisz procedurę służącą do przekształcania ciągu znaków reprezentującego liczbę w układzie o podstawie* d *na typ* unsigned long*. Możesz założyć, że* $d \in \{2, 3, \ldots, 16\}$

Ćwiczenie 3.5 *Napisz funkcję wyznaczającą czterdzieści pierwszych cyfr rozwinięcia liczby* $x \in [0, 1)$ *w układzie o zadanej podstawie* $r > 1$*.*

Ćwiczenie 3.6 *Przerób algorytm wyznaczania sita Erastotenesa tak aby pracował on na pojedynczych bitach, czyli tak aby do reprezentacji jednej liczby naturalnej wykorzystywał tylko jeden bit.*

Ćwiczenie 3.7 *Napisz program który wyznacza wartości wyrażenia arytmetycznego* $x^7 - 7 * x^6 + 21 * x^5 - 35 * x^4 + 35 * x^3 - 21 * x^2 + 7 * x - 1$ *dla* $x = 0.096 \ldots 1.004$ *z krokiem* 0.0001*. Zapisz wyniki do pliku, wczytaj je do arkusza kalkulacyjnego i narysuj przybliżony wykres funkcji zdefiniowanej tym wyrażeniem. Porównaj go z wykresem funkcji* $y = (x - 1)^7$*.*

Ćwiczenie 3.8 *Napisz bibliotekę funkcji obsługujących liczby zespolone. Za pomocą procedur z tej biblioteki rozwiąż układ równań liniowych*

$$\begin{cases} (1+i)w & + & (2-i)z & = & 2-2i \\ (1-i)w & + & (3-i)z & = & 3+3i \end{cases}$$

Ćwiczenie 3.9 *Napisz funkcję do konwersji liczb zapisanych w notacji rzymskiej na liczby typu unsigned int oraz funkcję wykonującą przekształcenie odwrotne. Możesz ograniczyć się do liczb z zakresu 1 - 10000.*

Ćwiczenie 3.10 *Napisz funkcję do zamiany kąta wyrażonego w stopniach, minutach i sekundach na radiany. Napisz odwrotną funkcję.*

Zadanie 3.1 *Udowodnij wzór*

$$1 + p + p^2 + \ldots + p^{n-1} = \frac{p^n - 1}{p - 1}$$

Zadanie 3.2 *Niech* p*,* q *i* r *będą zmiennymi zdaniowymi. Przez* $\oplus$ *oznaczamy alternatywę wykluczającą. Pokaż, że* $p \oplus q \leftrightarrow q \oplus p$*,* $(p \oplus q) \oplus r \leftrightarrow p \oplus (q \oplus r)$ *i* $(p \oplus q) \oplus q \leftrightarrow p$*.*

Zadanie 3.3 *Zapisz swój wiek, liczbę 2003 w układzie dwójkowym i szesnastkowym. Oszacuj ilość cyfr potrzebnych do zapisania danej liczby n w układzie dwójkowym i szesnastkowym.*

Zadanie 3.4 *Przypomnij sobie cechy podzielności przez liczby 2, 5, 10, 3 i 9 w układzie dziesiętnym. Wyznacz cechy podzielności przez liczby 2, 4, 8, i 3 w układzie dwójkowym.*

Zadanie 3.5 *Pokaż, że jeśli X i Y są zbiorami skończonymi o tej samej ilości elementów oraz niech $f : X \mapsto Y$ będzie odwzorowaniem „na”, to wtedy f jest odwzorowaniem różnowartościowym.*

Zadanie 3.6 *Pokaż, że jeśli zbiór X ma n elementów, to zbiór $P(X)$ ma 2^n elementów.*

Zadanie 3.7 *Zbuduj tabliczki dodawania i mnożenia dla 3 bitowych liczb całkowitych ze znakiem.*

Zadanie 3.8 *Znajdź samodzielnie (na kartce papieru) rozwinięcie liczby $1/3$ w układzie dwójkowym.*

Rozdział 4

Podstawowe metody

W rozdziale tym omówimy kilka metod i narzędzi które stosowane są przy projektowaniu algorytmów: techniki rekurencyjne, przeszukiwanie z nawrotami, metodę „dziel i rządź", programowanie dynamiczne, wykorzystanie stosów, automaty skończone i systemy przepisujące.

4.1 Rekursja

Rekursją nazywamy metodę specyfikacji procesu w terminach definiowanego procesu. Dokładniej mówiąc, „skomplikowane" przypadki procesu redukowane są do „prostszych" przypadków. Stałymi składnikami metod rekurencyjnych są określenie warunku zatrzymania, oraz określenie metody redukcji problemu do prostszych przypadków.

Jednym z przykładów rekursji są rekurencyjnie definiowane funkcje.

Przykład 4.1 *Klasycznym przykładem funkcji zdefiniowanej rekurencyjnie jest silnia. Przypomnijmy, że dla liczb naturalnych n określamy $n! = 1 \cdot 2 \cdot \ldots \cdot n$. Najprostszy program do wyznaczania tej funkcji może być oparty o pętle. Jeśli jednak zauważymy, że $0! = 1$ oraz, że $n! = n \cdot (n-1)!$, to możemy napisać następującą funkcję*

```
int Silnia(int n) { return(n?n*Silnia(n-1):1); }
```

Język C pozwala na definicje rekurencyjne - w ciele funkcji może wystąpić odwołanie do jej samej. Nie wszystkie języki programowania na to pozwalają.

Rekurencyjne rozwiązywanie zadań polega na zredukowaniu ich do zadań prostszych i określeniu warunków zatrzymania. W przypadku funkcji Silnia zadanie dla liczby n zostało zredukowane do zadania dla $n-1$. Warunkiem zatrzymania jest $n = 0$. Następujące twierdzenie z Teorii Mnogości uzasadnia poprawność najprostszej formy definicji rekurencyjnych:

Twierdzenie 4.1 *Niech X będzie dowolnym zbiorem, Załóżmy, że $a \in X$ oraz $F : X \to X$. Istnieje wtedy dokładnie jedna funkcja $g : \mathbb{N} \to X$ taka, że $g(0) = a$ oraz $(\forall n \in \mathbb{N})(g(n+1) = F(g(n)))$.*

Rekursję można stosować nie tylko do definiowania funkcji. Można ją stosować do rozwiązywania problemów. W tym przypadku metoda polega na redukcji złożonego problemu na problemy prostsze.

Przykład 4.2 (Wieże z Hanoi) *Mamy trzy patyki ponumerowane liczbami 1, 2 i 3 oraz n krążków. Krążki są różnych rozmiarów. Początkowo wszystkie krążki nawleczone są na pierwszy patyk w malejącej kolejności: największy leży na dole a najmniejszy na samej górze. Należy je przestawić na trzeci patyk przestrzegając następujących dwóch reguł: wolno przekładać tylko jeden krążek w jednym ruchu, nigdy większy krążek nie może zostać położony na mniejszym krążku.*

Zadanie to jest oczywiście banalne, jeśli $n = 1$. Wtedy wystarczy przenieś krążek z pierwszego patyka na trzeci patyk. Rozwiązanie całego zadania stanie się jasne, jeśli zauważmy, że jeśli potrafimy przenieść n krążków z patyka i na patyk j, to potrafimy zadanie to rozwiązać dla $n + 1$ krążków: oto strategia:

> *wyznacz wolny patyk; przenieś górne n krążków z patyka i na wolny patyk, nie ruszając największego krążka; przenieś największy krążek na patyk j-ty; przenieś górne n krążków z wolnego patyka na patyk j-ty*

Oto jak tę strategię możemy zamienić na program:

```
void Hanoi(int n, int skad, int dokad)
{
  int wolny;
  if (n==1) printf("%d » %d",skad,dokad);
  else {
     wolny = 6 - (skad + dokad);
     H(n-1,skad,wolny);
     printf("%d » %d",skad,dokad);
     H(n-1,wolny,dokad);
  }
}
```

Uwaga. Problem Wież z Hanoi wymyślił Edouard Lucas w roku 1883. Według legendy którą wtedy opowiedział, w pewnej hinduskiej świątyni mnichowie przekładają bez przerwy układ 64 krążków zgodnie z regułami Wież z Hanoi. Po zrealizowaniu zadania świat ma się zakończyć. Nie wiadomo, czy Lucas sam wymyślił tę legendę, czy też ją usłyszał.

Niech $H(n)$ oznacza ilość operacji przeniesienia krążka, którą wykonuje nasza procedura dla konfiguracji n krążków. Oczywiście $H(1) = 1$ oraz $H(n+1) = H(n) + 1 + H(n) = 2H(n) + 1$. Zajmiemy się teraz wyznaczeniem wzoru na $H(n)$. Wypiszmy kilka pierwszych wartości:

$$\begin{cases} H(1) &= & 1 \\ H(2) &= & 2H(1)+1 \\ H(3) &= & 2H(2)+1 \\ H(4) &= & 2H(3)+1 \end{cases}$$

Pomnóżmy pierwszą równość przez 8, drugą przez 4 a trzecią przez 2. Otrzymamy

$$\begin{cases} 8H(1) &= & 8 \\ 4H(2) &= & 8H(1)+4 \\ 2H(3) &= & 4H(2)+2 \\ H(4) &= & 2H(3)+1 \end{cases}$$

Po zsumowaniu wszystkich równości otrzymamy

$$H(4) = 1 + 2 + 4 + 8 = 1 + 2 + 2^2 + 2^3 = 2^4 - 1,$$

co powinno nam nasunąć następującą hipotezę: $(\forall n > 0)(H(n) = 2^n - 1)$. Jest ona prawdziwa, co łatwo możemy sprawdzić indukcją matematyczną.

Definicje rekurencyjne stosować można również do funkcji wielu zmiennych. Oto przykład znanej funkcji dwóch zmiennych o bardzo prostej definicji która jest bardzo trudna do obliczenia nawet dla stosunkowo małych wartości parametrów.

Przykład 4.3 *Funkcją Ackermana nazywamy funkcję dwóch zmiennych naturalnych zdefiniowaną wzorem*

$$A(n,m) = \begin{cases} m+1 & : \quad n = 0 \\ A(n-1,1) & : \quad n > 0 \wedge m = 0 \\ A(n-1, A(n, m-1)) & : \quad n > 0 \wedge m > 0 \end{cases}$$

Niewłaściwe użycie rekursji

Pisząc procedurę rekurencyjną musimy zadbać o to aby przy każdym dopuszczalnym zestawie parametrów wejściowych zatrzymywała się ona po skończonej ilości kroków. Następująca funkcja

```
int R(int n){return(R(n-1));}
```

mimo iż jest poprawna pod względem składniowym, to jest rozbieżna dla każdego parametru wejściowego n. Nie ma ona określonego żadnego warunku zatrzymania.

Rozważymy teraz drugi rodzaj niewłaściwego użycia rekursji. Załóżmy, że chcemy wyznaczyć wartość współczynnika Newtona

$$\binom{n}{k} = \frac{n!}{k!(n-k)!}$$

korzystając z dobrze znanej równości Pascala

$$\binom{n}{k} = \binom{n-1}{k-1} + \binom{n-1}{k},$$

oraz z dwóch dodatkowych wzorów $\binom{n}{0} = \binom{n}{n} = 1$. Pomysł ten prowadzi do następującego kodu:

```
long Newton0(int n, int k)
{
  return((k==0)||(k==n)?1:Newton0(n-1,k-1)+Newton(n-1,k));
}
```

Przyjrzyjmy się jednak, jakie obliczenia będą wykonywane przez tak zaprogramowaną funkcję dla parametrów (7,4):

$$\binom{7}{4} \to \left\{\binom{6}{3}, \binom{6}{4}\right\} \to \left\{\left\{\binom{5}{2}, \binom{5}{3}\right\}, \left\{\binom{5}{3}, \binom{5}{4}\right\}\right\} \to$$

$$\left\{\left\{\left\{\binom{4}{1}, \binom{4}{2}\right\}, \left\{\binom{4}{2}, \binom{4}{3}\right\}\right\}, \left\{\left\{\binom{4}{2}, \binom{4}{3}\right\}, \left\{\binom{4}{3}, \binom{4}{4}\right\}\right\}\right\} \to \dots$$

Widzimy, że pewne obliczenia powtarzają się wielokrotnie. A efektem tego jest bardzo duża złożoność obliczeniowa zaprojektowanej procedury.

Metoda rekursji jest najbardziej efektywna jeśli uda się nam rozłożyć zadanie na kilka całkowicie niezależnych zadań. A w tym przypadku tak nie jest. Po chwili zastanowienia zauważymy pewnie, że posłużyć się możemy trochę innym wzorem:

$$\binom{n}{k} = \binom{n-1}{k-1} \cdot \frac{n}{k},$$

który prowadzi do następującego kodu

```
long Newton(int n, int k)
{
  return((k==0)||(k==n)?1:(Newton0(n-1,k-1)*n)/k));
}
```

o złożoności obliczeniowej $O(n)$.

4.2 Przeszukiwanie z nawrotami

Często spotykanym problemem jest znajdowanie wszystkich potencjalnych rozwiązań zadanego problemu. W niektórych sytuacjach, na przykład w zadaniach znalezienia drogi w labiryntach, nie znamy prostej metody wyświetlenia rozwiązania. Naturalnym sposobem może być wtedy tylko metoda prób i błędów.

Przykład 4.4 (Problem Hetmanów) *Przyjrzymy się teraz znanemu problemowi, który polega na znalezieniu takich rozstawień ośmiu hetmanów na szachownicy rozmiaru* 8×8 *tak aby żadne dwa z nich nie atakowały się nawzajem. Hetmany nawzajem nie atakujące się muszą stać w różnych kolumnach, tak więc do reprezentacji położenia hetmanów możemy używać tablicy H[1..8] liczb ze zbioru* $\{1, \ldots, 8\}$*. Problemem tym zajmował się już zajmował się Gauss.*

```
#define Abs(A) ((A)>0?(A):-(A))

// Polozenia hetmanow pamietane sa w komorkach X[1...8]

void Het(int X[],int k, int *licznik)
//Zakladamy, ze X[1] ... X[k] sa juz wyznaczone
{
  int i,j,OK;
  if (k==8)
  {
     (*licznik)++;
     drukuj rozwiazanie;
     return;
  }
  for (i=1;i<=8;i++)
  {
     OK= TRUE;
     for (j=1;j<=k;j++)
        OK = OK && (X[j] != i) && (Abs(X[j]-i) != k+1-j);
     if (OK)
```

```
      {
        X[k+1]= i;
        Het(X,k+1,licznik);
      }
  }
  return;
}
```

Procedurę Het wywołujemy następująco:

```
int main() {
  int X[9],Licznik=0;
  Het(X,0,&Licznik);
}
```

4.3 Dziel i rządź

Metodę rozwiązywania problemów "dziel i rządź"(*divide et impera*) zastosowana została podobno po raz pierwszy przez Juliusza Cezara do rozwiązywania problemów politycznych. W informatyce oznacza ona rozbicie zadania na mniejsze podzadania i po ich rozwiązaniu zsyntezowania rozwiązania całego zadania z rozwiązań mniejszych podzadań. Dokładniej mówiąc metoda ta składa się z następujących kroków:

1. dzielenie: podział zadania na kilka podzadań
2. rządzenie: rozwiązanie wszystkich podzadań
3. scalenie: złączenie rozwiązań podzadań w rozwiązanie całego zadania

Przykład 4.5 (Minimalne i maksymalne elementy) *Wyszukiwanie najmniejszego elementu z danej tablicy liczb nie sprawia żadnego kłopotu. Można go zrealizować za pomocą pętli*

```
wmin = tab[0];
for (i=1;i<n;i++)
if (tab[i]<wmin)
  wmin = tab[i];
```

W procedurze tej wykonaliśmy $n-1$ porównań. Podobnie do wyszukania elementu maksymalnego potrzebujemy $n-1$ porównań. Proces jednoczesnego znalezienia elementu najmniejszego i największego zrealizować potrafimy więc za pomocą 2n-2 porównań. Czy nie można jednak tego zrobić lepiej?

Załóżmy, że mamy znaleźć minimalne i maksymalne wartości z tablicy rozmiaru 2^n. Niech T(n) oznacza ilość potrzebnych do tego celu porównań. Jeśli $n = 1$ to do tego celu wystarczy nam jedno porównanie, zatem T(2)=1. Załóżmy teraz, że mamy daną tablicę A długości $2n$. Za pomocą T(n) porównań możemy wyznaczyć minimum a_1 i maksimum b_1 z $A[1..n]$ oraz za pomocą tej samej liczby porównań możemy wyznaczyć odpowiednie wartości a_2 i b_2 z tablicy $A[n+1..2n]$. Jeśli $a_1 < a_2$ to

minimalną wartości jest a_1 a w przeciwnym przypadku jest nią a_2. Podobnie, jeśli $b_1 > b_2$, to maksymalną wartością jest b_1 a w przeciwnym wypadku jest nią b_2. Zatem

$$T(2n) = 2 \cdot T(n) + 2.$$

Łatwo sprawdzić metodą indukcji matematycznej, że dla n będących potęgami dwójki mamy $T(n) = \frac{3}{2}n - 2$. ***Czary mary****: $\frac{3}{2}n - 2 < 2n - 2$.*

Do analizy złożoności obliczeniowej rozważanych dalej algorytmów wiele razy stosować będziemy następujące twierdzenie:

Twierdzenie 4.2 (Master Theorem) *Załóżmy, a,b są liczbami dodatnimi oraz c dodatnią liczbą naturalną, że T jest funkcją, która dla potęg liczby c spełnia równanie*

$$T(n) = \begin{cases} b & : \quad n = 1 \\ aT(\frac{n}{c}) + bn & : \quad n > 1 \end{cases}$$

Wtedy dla liczb naturalnych n będących potęgami liczby c mamy

1. *jeśli $a < c$ to $T = O(n)$,*
2. *jeśli $a = c$ to $T = O(n \log(n))$,*
3. *jeśli $a > c$ to $T = O(n^{\log_c a})$.*

Dowód. Pokażemy najpierw, że

$$T(c^n) = bc^n \sum_{i=0}^{n} (\frac{a}{c})^i$$

dla wszystkich $n \in \mathbb{N}$. Dla $n = 0$ równość ta jest spełniona. Załóżmy, że jest prawdziwa dla liczby n Wtedy

$$T(c^{n+1}) = aT(c^n) + bc^{n+1} = a(bc^n \sum_{i=0}^{n} (\frac{a}{c})^i) + bc^{n+1} =$$

$$bc^{n+1}(\frac{a}{c} \sum_{i=0}^{n} (\frac{a}{c})^i + 1) = bc^{n+1} \sum_{i=0}^{n+1} (\frac{a}{c})^i.$$

Zauważmy teraz, że jeśli $k = c^n$ to $n = \log_c k$. Zatem dla liczb k które są potęgami liczby c mamy

$$T(k) = bk \sum_{i=0}^{\log_c k} (\frac{a}{c})^i$$

Jeśli $a < c$ to szereg $\sum_{i=0}^{\infty} (\frac{a}{c})^i < \infty$ więc wtedy $T(k) = Ck$ dla pewnej stałej C. To pokazuje punkt (1). Jeśli $a = c$ to $\sum_{i=0}^{\log_c k} (\frac{a}{c})^i = \log_c k + 1$, więc $T(k) = bk(log_c k + 1)$, a więc $T = O(k \log k)$, co pokazuje punkt (2). Jeśli $a > c$, to

$$T(k) = bk \frac{(\frac{a}{c})^{\log_c k + 1} - 1}{\frac{a}{c} - 1} \simeq Ck(\frac{a}{c})^{\log_c k} = Ckc^{\log_c (a/c) \cdot \log_c k} = Dk^{\log_c (a)}.$$

□

Sortowanie przez scalanie

Spróbujmy zastosować podobną metodę do problemu sortowania. Załóżmy więc, że mamy daną tablicę liczb A[l..p]. **Podzielmy** ją na dwie tablice $A[l..s]$, $A[s+1..p]$, gdzie $s = [(l_p)/2]$, i **posortujmy je niezależnie od siebie niezależnie**. Po wykonaniu tych czynności **scalimy** dwie posortowane tablice w jedną tablicę. Będziemy to czynili tak: będziemy mieli pomocniczą tablicę Rob; ustalimy trzy indeksy i=l, j=s+1 oraz k=0; jeśli a[i]<a[j] to wykonamy podstawienie Rob[k]=a[i], zwiększymy i oraz k o jeden; jeśli jeśli a[i]>=a[j] to wykonamy podstawienie Rob[k]=a[j], zwiększymy j oraz k o jeden; wykonywać to będziemy tak długo, aż $i = s \vee j = p$; na koniec, jeśli coś nam zostanie w tablicy A[l,s], to dodamy do Rob i jeśli coś nam zostanie w tablicy A[s+1,p] to dodamy to do Rob; na koniec skopiujemy zawartość tablicy Rob do A[l..p]. Oto kod procedury:

```
#define MAX 1000000
int Rob[MAX];

void SortC(int A[], long l, long p)
{
   long i,j,k,s;
   if (l==p)
      return;
   s = (l+p)/2;
   SortC(A,l,s);
   SortC(A,s+1,p);
   //scalanie
   i= l;
   j= s+1;
   k= 0;
   while ((i<=s) && (j<=p))
   if (A[i]<A[j])Rob[k++]=A[i++];
      else Rob[k++]=A[j++];
   while (i<=s) Rob[k++]=A[i++];
   while (j<=p) Rob[k++]=A[j++];

   for (i=l,j=0;i<=p;i++,j++)
      A[i]=Rob[j];
}//SortC
```

Przeanalizujmy ilość porównań i podstawień $C(n)$ tej procedury dla tablicy rozmiaru n. Założymy przy tym, dla uproszczenia, że n jest potęgą liczby 2. Mamy oczywiście $C(1) = 0$ oraz $C(2n) = 2C(n) + Dn$ dla pewnej stałej D. Z Master Theorem otrzymujemy natychmiast następujący wniosek: C = $O(n \ln n)$ dla n będących potęgami dwójki. Wynik ten łatwo można rozszerzyć na dowolną liczbę n: jeśli mianowicie n nie jest potęgą dwójki, to zwiększymy jej rozmiar do najbliższej liczby m postaci 2^k i dodamy na koniec elementy większe od wszystkich elementów z oryginalnej tablicy. Taka liczba m istnieje zawsze w przedziale $[n, 2n-1)$, więc na pewno jest mniejsza od 2n. Pokazaliśmy zatem twierdzenie:

Twierdzenie 4.3 *Tablicę rozmiaru n można posortować za pomocą $O(n \log n)$ porównań i podstawień.*

Pokazać można, że każda metoda sortowania za pomoca porównywania elementów musi mieć złożoność obliczeniową rzędu $O(n \log c)$. Matoda sortowania przez scalanie jest więc optymalna, z dokładnością do stałej, pod względem ilości porównań.

Wadą omówionej metody sortowania przez scalanie jest korzystanie z dodatkowej tablicy Rob. Można ją wyeliminować. Ale nie będziemy tego omawiali w tej książce, gdyż zapoznasz się z nimi na wykładach z „Algorytmów i Struktur Danych". Zamiast tego omówimy inną metodę sortowania, której przeciętny czas działania jest rzędu $O(n \log n)$, lecz niestety, najgorszy czas jest rzędu $O(n^2)$. Nazywa się ona metodą szybkiego sortowania.

Algorytm mnożenia Karatsuby

Powróćmy do zadania mnożenia bardzo długich liczb całkowitych. W rozdziale 2.1 omówiliśmy klasyczną, szkolną metodę, która działa w czasie $O(N^2)$, gdzie N oznacza liczbę cyfr. Omówimy teraz szybszą metodę, wymyśloną przez Karatsubę w roku 1962. Załóżmy, że chcemy pomnożyć dwie długie liczby x i y całkowite zapisane w układzie o podstawie 10. Załóżmy, że obie liczby mają parzystą liczbę n = $2m$ cyfr (jeśli nie, to dodajmy zera z ich lewej strony). Możemy je zapisać w postaci

$$x = 10^m \cdot x_1 + x_2, y = 10^m \cdot y_1 + y_2.$$

gdzie wszystkie liczby x_1, x_2, y_1, y_2 są już m-cyfrowe. Wtedy

$$x \cdot y = 10^{2m} x_1 y_1 + 10^m (x_1 y_2 + x_2 y_1) + x_2 y_2,$$

więc powinniśmy umieć szybko wyznaczyć liczby $x_1 y_1$, $x_1 y_2 + x_2 y_1$ and $x_2 y_2$. Musimy więc umieć wyznaczyć cztery iloczyny liczb mniejszych. **Karatsuba zauważył, że można to wykonać za pomocą tylko trzech mnożeń**. Niech bowiem $A = x_1 y_1$, $B = x_2 y_2$ oraz $C = (x_1 + x_2)(y_1 + y_2)$. Wtedy liczbę $x_1 y_2 + x_2 y_1$ można wyliczyć z liczb A, B i C za pomocą dodawania i odejmowania:

$$C - (A + B) = x_1 y_2 + x_2 y_1.$$

Aby wyznaczyć produkty liczb m-cyfrowych można zastosować rekurencyjnie ten sam trik. Niech T(n) oznacza czas potrzebny do pomnożenia dwóch n - cyfrowych liczb metodą Karatsuby. Wtedy

$$T(n) \leq 3T(\frac{n}{2}) + b \cdot n.$$

dla pewnej stałej b.

Twierdzenie 4.4 *Algorytm Karatsuby działa w czasie* $O(n^{\log_2 3})$.

Dowód. Na mocy twierdzenia 4.2 jeśli funkcja T spełnia równanie $T(n) = 3T(\frac{n}{2}) + b \cdot n$, to $T(n) = O(n^{\log_2 3})$ dla liczb n będących potęgą liczby c. □

Zauważ, że $\log_2 3 = \frac{\log_{10} 3}{\log_{10} 2} \simeq 1.585$, zatem $n^{\log_2 3} = o(n^2)$. Algorytm Karatsuby jest więc znacznie szybszy od prostego „ręcznego" algorytmu omawianego w rozdziale 2.1. Znane są jeszcze szybsze metody mnożenia dużych liczb naturalnych. Algorytm Schönhage - Strassen z roku 1972 działa w czasie $O(n \log(n) \log(\log(n)))$. Wykorzystuje on stosunkowo zaawansowane narzędzia matematyczne, a mianowicie transformacje Fouriera.

4.4 Dynamiczne programowanie

4.5 Stosy

Techniki rekurencyjne, metoda dziel i rządź, przeszukiwanie z nawrotami oraz programowanie dynamiczne są metodami konstruowania algorytmów. Omówimy teraz jeden ze standardowych sposobów realizacji algorytmów. Jest nią narzędzie zwane **stosem**. Wyobrażamy go sobie jako listę obiektów jednego typu. Do listy tej możemy dodawać obiekty, ale tylko na jej koniec. Możemy pobierać obiekty, ale tylko z jej końca. Stos możemy zapytać również o ostatni element oraz ilość umieszczonym w nim obiektów.

Przykład 4.6 (Prosty interpretator)

```
#define OR(A,B) (A)>(B)?(A):(B)
#define AND(A,B) (A)>(B)?(B):(A)
#define NOT(A) 1-(A)

#define MAX_STOSU 100
int iSTOS[MAX_STOSU];
int iPOZYCJA = 0;

void iInit()
{
  iPOZYCJA = 0;
}
void iPush(int X)
{
  if (iPOZYCJA>=0)
     iSTOS[iPOZYCJA++]=X;
}
int iPop()
{  if (iPOZYCJA>=0)
     return(iSTOS[–iPOZYCJA]);
  else
     return(-1);
}

int Calc(char S[],int X[])
{
  int i,dl,a,b;
  dl= strlen(S);
  for (i=0;i<dl;i++)
  {
     switch (S[i]){
       case 'p': iPush(X[0]);break;
       case 'q': iPush(X[1]);break;
       case 'r': iPush(X[2]);break;
       case 's': iPush(X[3]);break;
       case 't': iPush(X[4]);break;
```

```
        case '+': a=iPop(); b=iPop();iPush(OR(a,b));break;
        case '*': a=iPop(); b=iPop();iPush(AND(a,b));break;
        case '-': a=iPop(); iPush(NOT(a));break;
        default :
      }
  }
  return(iPop());
}
```

Oto sposób użycia powyższych funkcji.

```
int main()
{
  char S[] = "pqr**"; // S= p*(q*r)
  int X[5];
  for (int i=0;i<2;i++)
  for (int j=0;j<2;j++)
  for (int k=0;k<2;k++)
  {
     X[0]=i; X[1]=j;X[2]=k;
     iInit;
     printf("%d %d %d ::: %d\n",i,j,k,Calc(S,X));
  }
}
```

4.6 Automaty skończone

Automaty skończone są zarazem bardzo prostym jak i bardzo ogólnym urządzeniem liczącym. Rozpoczniemy od formalnego opisu tych obiektów.

Definicja 4.1 *Automatem skończonym nazywamy* $\mathbf{A} = (A, \Sigma, \alpha, \beta, a)$*, gdzie* A *jest niepustym zbiorem zwany zbiorem stanów,*Σ *jest niepustym zbiorem zwanym alfabetem,* $\alpha : A \times \Sigma \to A$*,* $\beta : A \times \Sigma \to \Sigma$ *i* $a \in A$*.*

TUTAJ BĘDZIE PRZYKŁAD

4.7 Systemy przepisujące

Definicja 4.2 *Systemem przepisującym nazywamy strukturę* $\mathcal{S} = (X, R)$*, gdzie* X *jest niepustym zbiorem zaś* R *jest binarną relacją na zbiorze* X*.*

Niech $\mathcal{S} = (X, R)$ będzie systemem przepisującym. Element $x \in X$ nazywamy terminalnym, jeśli nie istnieje $y \in X$ taki, że $(x, y) \in R$. Elementy relacji R nazywamy regułami przepisywania.

Pojęcie systemu przepisującego jest bardzo ogólne. W praktyce stosuje się tylko takie systemy, dla których nie istnieje nieskończony ciąg $x_0, x_1, x_2, \ldots$ taki, że $(\forall n)((x_n, x_{n+1}) \in R)$.

```
T uprosc(T x)
{
  x = x0;
  while (!Terminal(x))
  {
     znajdź y t.że (x, y) ∈ R;
     x = y;
  }
  return(x); }
```

Przykład 4.7

4.8 Ćwiczenia i zadania

Ćwiczenie 4.1 *Napisz rekurencyjny program, który wyznacza sumę liczb zawartych w tablicy bez użycia pętli.*

Ćwiczenie 4.2 *Napisz program, który za pomocą* $\lceil \frac{3n}{2} \rceil + 1$ *porównań wyznacza jednocześnie minimalną i maksymalną wartość z zadanej tablicy* n *liczb rzeczywistych.*

Ćwiczenie 4.3 *Korzystając z tego, że* $x^1 = x$, $x^{2n} = (x^n)^2$ *oraz* $x^{2n+1} = (x^n)^2 \cdot x$ *napisz rekurencyjną wersję funkcji* double intpower(double x, int n) *która wyznacza liczbę* x^n. *Oszacuj liczbę mnożeń które wykonuje ta funkcja.*

Ćwiczenie 4.4 *Napisz program, który za pomocą scalania sortuje tablicę losowych liczb naturalnych. Porównaj tempo jej działania z procedurą sortowania przez wstawianie.*

Ćwiczenie 4.5 *Oszacuj dla jakich liczb* n *wszystkie liczby* $\binom{n}{0}$, $\binom{n}{1}$, $\ldots$ $\binom{n}{n}$ *są mniejsze od* 2^{64}. *Napisz rekurencyjną i nierekurencyjną wersję programu który wyznacza liczby* $\binom{30}{0}$, $\binom{30}{1}$, $\ldots$ $\binom{30}{30}$. *Wczytaj wygenerowane liczby do arkusza kalkulacyjnego i narysuj wykres zależności* $k \to \binom{30}{k}$.

Ćwiczenie 4.6 *Napisz funkcję strMatch, która ustala zgodność wzorca z łańcuchem. Znak '?' we wzorcu oznacza zgodność z dowolnym innym znakiem, znak '*' oznacza zgodność z dowolnym, również pustym, łańcuchem, znak różny od '?' i '*' oznacza zgodność tylko z samym sobą. Na przykład, strMatch("*.doc",s) ma zwracać TRUE wtedy i tylko wtedy, gdy napis s jest postaci "xxxxx.doc" oraz strMatch("a???",s) ma zwracać TRUE wtedy i tylko wtedy, gdy s ma długość 4 i zaczyna się od litery 'a'.*

Ćwiczenie 4.7 *Wyznacz taką sekwencję ruchów skoczka szachowego na szachownicy o rozmiarach* 8×8, *że każde pole szachownicy jest odwiedzone dokładnie jeden raz.*

Ćwiczenie 4.8 *Napisz procedurę, która generuje wszystkie permutacje skończonego zbioru* $\{1, 2, \ldots, n\}$.

Ćwiczenie 4.9 *Napisz procedurę, która znajduje wszystkie podzbiory* k *- elementowe zbioru* $\{1, 2, \ldots, n\}$.

Ćwiczenie 4.10 *Napisz program, który wczytuje formułę rachunku zdań zapisaną w notacji polskiej, zbudowaną ze zmiennych zdaniowych p, q, r, s, t, drukuje jej tablicę zero-jedynkową i i sprawdza, czy jest ona tautologią.*

Zadanie 4.1 *Jaka jest moc zbioru wszystkich permutacji zbioru* $\{1, \ldots n\}$*?*

Zadanie 4.2 *Jaka jest moc zbioru wszystkich k elementowych podzbiorów zbioru n elementowego?*

Zadanie 4.3 *Pokaż, że* $2^n - 1$ *ruchów jest konieczne i wystarczające do rozwiązania problemu wież z Hanoi*

Zadanie 4.4 *Pokaż, że funkcja Ackremana jest poprawnie zdefiniowana. Wyznacz wartości funkcji Ackremana dla małych wartości parametrów.*

Rozdział 5

Granice obliczalności

```
unsigned int main()
{
   unsigned int n;
   scanf ("%d", &n);
   if (U(n,n)==1) {while (1) do;};
   else return(1);
}
```

W rozdziale tym zajmiemy się dotarciem do granic informatyki - zajmiemy się znalezieniem naturalnych przykładów zagadnień, które nie są algorytmizowalne.

W rozdziale tym założymy, że mamy do dyspozycji super-komputer, w którym możemy posługiwać się liczbami naturalnymi dowolnych rozmiarów. Posługiwać się będziemy pewnym wariantem języka C, który oznaczymy symbolem $C^{\mathbb{N}}$. W języku tym posługiwać się można tylko jednym typem zmiennych podstawowych **unsigned int**, przy czym nie musimy się troszczyć o rozmiar przechowywanych w tych zmiennych danych - mogą to być liczby naturalne dowolnych rozmiarów.

Zakładać będziemy również, że nasz super-komputer dysponuje kompilatorem języka $C^{\mathbb{N}}$. Założymy, że jeśli w trakcie wykonywania jakiegoś $C^{\mathbb{N}}$ programu nastąpi dzielenie przez zero, to wykonywanie programu ulegnie przerwaniu i jako jego wynik działania pojawi się liczba zero.

5.1 Funkcje obliczalne

Rozważać będziemy pewną ograniczoną klasę programów w języku C^N, które nazywać będziemy C_n programami (gdzie $n \in \mathbb{N} \setminus \{0\}$). Główna funkcja w tym programach musi mieć postać

```
unsigned int main()
{
   unsigned int x1,...,xn;
   scanf ("%d ... %d", &x1,...,&xn);
   printf("%d",f(x1,...,xn));
   return(0);
}
```

Symbolem $f : \mathbb{N}^n \hookrightarrow \mathbb{N}$ oznaczać będziemy to, że f jest funkcją o dziedzinie zawartej w zbiorze $\mathbb{N}^n$ i obrazie zawartym w zbiorze $\mathbb{N}$. Funkcję $f : \mathbb{N}^n \hookrightarrow \mathbb{N}$ taką, że $dom(f) \neq \mathbb{N}^n$ nazywać będziemy **funkcją częściową**.

Definicja 5.1 *Funkcję $f : \mathbb{N}^n \hookrightarrow \mathbb{N}$ nazywamy **funkcją obliczalną** jeśli istnieje taki C_n program $\mathcal{P}$, że*

1. *jeśli $(x_1, \ldots, x_n) \in dom(f)$ to program $\mathcal{P}$ po wczytaniu danych $x_1, \ldots, x_n$, po wykonaniu skończonej liczby obliczeń zatrzymuje się i zwraca wartość $f(x_1, \ldots, x_n)$;*
2. *jeśli $(x_1, \ldots, x_n) \notin dom(f)$ to program $\mathcal{P}$ po wczytaniu danych $x_1, \ldots, x_n$ zapętla się, czyli nigdy się nie zatrzymuje.*

Aby oswoić się z definicją funkcji obliczalnej rozważymy najpierw kilka prostych przykładów.

Przykład 5.1 *Funkcja stale równa liczbie naturalnej c jest obliczalna. Rzeczywiście, można ją zrealizować za pomocą następującego programu*

```
unsigned int f(unsigned int n)
{
  return(c);
}

unsigned int main()
{
  unsigned int x1;
  scanf ("%d", &x1);
  printf("%d",f(x1));
  return(0);
}
```

Przykład 5.2 *Dodawanie dwóch liczb naturalnych jest funkcją dwóch zmiennych. Jest ono funkcją obliczalną gdyż można zrealizować je za pomocą następującego programu*

```
unsigned int f(unsigned int n, m)
{
  return(n+m);
}

unsigned int main()
{
  unsigned int x1,x2;
  scanf ("%d %d", &x1, &x2);
  printf("%d",f(x1,x2));
  return(0);
}
```

W podobny sposób możemy stwierdzić, że mnożenie i potęgowanie liczb naturalnych są funkcjami obliczanymi. Dzielenie liczb naturalnych jest funkcją częściową dwóch zmiennych, gdyż nie jest określony wynik dzielenia przez zero. Do napisania programu, który je reprezentuje musimy mieć jakiś sposób na zapętlenie programu. Chyba najprostszym sposobem zapętlenia programu jest instrukcja `while (1) do`. Wykorzystamy ja w następnym przykładzie.

Przykład 5.3 *Dzielenie dwóch liczb naturalnych jest funkcją obliczalną Zrealizować je można za pomocą następującego programu*

```
unsigned int f(unsigned int n, m)
{
  if (m==0){while (1) do;}
  else return(n/m);
}

unsigned int main()
{
  unsigned int x1,x2;
  scanf ("%d %d", &x1, &x2);
  printf("%d",f(x1,x2));
  return(0);
}
```

Klasę wszystkich funkcji obliczalnych n zmiennych oznaczać będziemy przez $\mathbb{O}_n$. Ponadto używać będziemy oznaczenia $\mathbb{O} = \bigcup_{n\geq 1} \mathbb{O}_n$ na klasę wszystkich funkcji obliczalnych dowolnej liczby zmiennych. Zauważmy, że do klasy tej należą również funkcje częściowe, czyli takie, których dziedzina jest właściwym podzbiorem $\mathbb{N}^k$. Przykładem takiej funkcji jest rozważane wcześniej dzielenie: mamy $dom(DIV) = \mathbb{N} \times (\mathbb{N} \setminus \{0\})$.

Najmniejszą funkcją częściową jest funkcja pusta. Oznaczać ją będziemy symbolem $\uparrow$. Traktować ją będziemy jako funkcję jednej zmiennej. Czyli $\uparrow\colon \mathbb{N} \hookrightarrow \mathbb{N}$ oraz $dom(\uparrow) = \emptyset$. Zauważmy, że funkcja $\uparrow$ jest obliczalna [1] oraz, że dla każdej liczby $n \in \mathbb{N}$ wartość $\uparrow(n)$ nie jest określona.

Twierdzenie 5.1 (O złożeniu) *Załóżmy, że $f \in \mathbb{O}_n$ oraz $g_1, \ldots, g_n \in \mathbb{O}_k$. Wtedy funkcja g określona wzorem*

$$g(x_1, \ldots, x_k) = f(g_1(x_1, \ldots, x_k), \ldots, g_n(x_1, \ldots, x_k))$$

jest funkcją obliczalną.

5.2 Zbiory rozstrzygalne

Ustalmy zbiór Ω. Przypomnijmy, że funkcją charakterystyczną zbioru $A \subseteq \Omega$ nazywamy funkcję zdefiniowaną wzorem

$$\mathbf{1}_A(x) = \begin{cases} 1 & : \quad x \in A \\ 0 & : \quad x \in \Omega \setminus A \end{cases}$$

Jeśli $\Omega = \mathbb{N}$ lub $\Omega = \{0, \ldots, n-1\}$ dla pewnej liczby naturalnej $n \in \mathbb{N}$, to funkcję $\mathbf{1}_(A)$ nazywamy mapą bitową zbioru A. Łatwo sprawdzić, że dla zbiorów $A, B \subset \Omega$ mamy $\mathbf{1}_{A\cap B} = \min\{\mathbf{1}_A, \mathbf{1}_B\}$, $\mathbf{1}_{A\cup B} = \max\{\mathbf{1}_A, \mathbf{1}_B\}$ oraz $\mathbf{1}_(A^c) = 1 - \mathbf{1}_A$.

[1] Przypuszczalnie każdemu z nas wiele razy napisać taki program, który wpada w nieskończoną pętle dla dowolnych danych wejściowych.

Definicja 5.2 *Zbiór $A \subseteq \mathbb{N}^n$ nazywamy rozstrzygalnym, jeśli funkcja $\mathbf{1}_A$ jest funkcją obliczalną.*

Twierdzenie 5.2 *Załóżmy, że $A, B \subseteq \mathbb{N}^n$ są zbiorami rozstrzygalymi. Wtedy również zbiory $A \cap B$, $A \cup B$ oraz $\mathbb{N}^n \setminus A$ są zbiorami rozstrzygalnymi.*

Dowód. Z tego, że A i B są zbiorami rozstrzygalnymi wynika, że funkcje $\mathbf{1}_A$ i $\mathbf{1}_B$ są funkcjami obliczalnymi. Bez trudu stwierdzamy również, że funkcje $max(x, y) = \max\{x, y\}$, $min(x, y) = \min\{x, y\}$ oraz $neg(x) = \max\{0, 1 - x\}$ są obliczalne. Z twierdzenia 5.1 wynika, że fukcje $\mathbf{1}_{A\cap B} = min \circ (\mathbf{1}_A, \mathbf{1}_B)$, $\mathbf{1}_{A\cup B} = max \circ (\mathbf{1}_A, \mathbf{1}_B)$ oraz $\mathbf{1}_{A^c} = neg \circ \mathbf{1}_A$ są obliczalne. □

Twierdzenie 5.3 (O definiowaniu przez przypadki) *Załóżmy, że $A \subseteq \mathbb{N}$ jest zbiorem rozstrzygalym oraz, że f i g sa funkcjami obliczalnymi jednej zmiennej. Wtedy funkcja*

$$h(x) = \begin{cases} f(x) & : \quad n \in A \\ g(x) & : \quad n \notin A \end{cases}$$

jest również funkcją obliczalną.

Dowód. Wystarczy zauważyć, że

$$h(x) = \mathbf{1}_A(x) \cdot f(x) + \mathbf{1}_{A^c}(x) \cdot g(x)$$

i skorzystać z twierdzenia o złożeniu. □

5.3 Funkcja uniwersalna

Każdy program w języku $C^{\mathbb{N}}$ traktować możemy jako ciąg $p = (c_1, \ldots, c_N)$ znaków ASCII. Niech ascii(c) oznacza kod ASCII znaku c. Pokażemy metodę przyporządkowania każdemu programowi p pewnej liczby naturalnej, zwanej kodem Gödla programu. Niech mianowicie $(p_k)_{k\in\mathbb{N}}$ będzie ciągiem kolejnych liczb pierwszych. Wtedy kodem Gödla programu p nazywamy liczbę

$$\mathfrak{g}(p) = \prod_{i=1}^{N} p_i^{\text{ascii}(c_i)} .$$

Z Podstawowego Twierdzenia Arytmetyki wynika jednoznaczność kodów Gödla: jeśli $\mathfrak{g}(p) = \mathfrak{g}(q)$ to $p = q$.

```
scanf("%d",p);
prog = ascii(p)
if P11(prog) and skompiluj(prog) then
   uruchom(prog);
else begin
   scanf(n);
   printf(0);
end;
```

5.4 Problem zatrzymania

Zastanówmy się nad następującym zagadnieniem: mamy dany program, który po wczytaniu jednej liczby naturalnej wykonywać ma pewne obliczenia i po pewnym czasie zatrzymać się oraz zwrócić wynik. Naszym zadaniem jest sprawdzenie tego faktu, czyli sprawdzenie, że dla dowolnej danej wejściowej, po skończonej liczbie kroków, program zakończyć swoje działanie. Jest chyba jasne, że metoda testowania poprawności programu nic tutaj nie da. Pytanie brzmi: czy jesteśmy w stanie napisać algorytm, który wykonywał by za nas tego typu pracę.

Niech

$$\text{STOP} = \{(n,m) \in \mathbb{N}^2 : (n,m) \in dom(\mathbb{U}_2)\}$$

oraz

$$\mathbb{T} = \{n \in \mathbb{N} : (n,n) \in \text{STOP}\}$$

Twierdzenie 5.4 *Zbiór $\mathbb{T}$ nie jest rozstrzygalny.*

Dowód. Załóżmy, że zbiór $\mathbb{T}$ jest zbiorem rozstrzygalnym. Rozważmy następującą funkcję

$$f(x) = \begin{cases} \uparrow(n) & : \quad n \in \mathbb{T} \\ 0 & : \quad n \notin \mathbb{T} \end{cases} \tag{5.1}$$

Z Twierdzenia refthm:obl:cases o definiowaniu przez przypadki wynika, że f jest funkcją obliczalną. Istnieje zatem n_0 takie, że $f(n) = U_2(n_0, n)$ dla wszystkich $n \in \mathbb{N}$ (przypomnijmy, że za n_0 należy wziąść kod Gödla programu obliczającego funkcję f). Lecz wtedy

$$n_0 \in dom(f) \leftrightarrow (n_0 \notin \mathbb{T}) \leftrightarrow (n_0, n_0) \notin dom(\mathbb{U}_2) \leftrightarrow$$
$$\neg((n_0, n_0) \in dom(\mathbb{U}_2)) \leftrightarrow \neg(n_0 \in dom(f)) \leftrightarrow n_0 \notin dom(f)\ .$$

Otrzymaliśmy więc sprzeczność - wyniknęła ona z założenia, że $\mathbb{T}$ jest zbiorem rozstrzygalnym. □

Wniosek 5.1 (Problem STOPU) *Zbiór STOP nie jest rozstrzygalny.*

5.5 Przegląd problemów nierozstrzyganlych

Lemat 5.1 (s twierdzenie) *Istnieje funkcja całkowita obliczalna $s : \mathbb{N}^2 \to \mathbb{N}$ taka, że*

Bibliografia

[1] D. Harrell. *Rzecz o istocie informatyki. Algorytmika*. Państwowe Wydawnictwo Naukowe, Warszawa, 2001.

[2] B. W. Kernighan and D. M. Ritchie. *Język C*. Wydawnictwa Naukowo-Techniczne, Warszawa, 1988.

[3] B. W. Kernighan and D. M. Ritchie. *Język ANSI C*. Wydawnictwa Naukowo-Techniczne, Warszawa, 2003.

Indeks