informatyka+

Algorytmika i programowanie

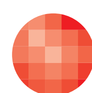
Bazy danych

Multimedia, grafika i technologie internetowe

Sieci komputerowe

Tendencje w rozwoju informatyki i jej zastosowań

Człowiek – najlepsza inwestycja



Projekt współfinansowany ze środków Unii Europejskiej w ramach Europejskiego Funduszu Społecznego.

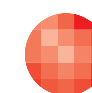
informatyka+

Wszechnica Informatyczna: Algorytmika i programowanie

Różnorodne algorytmy obliczeń i ich komputerowe realizacje

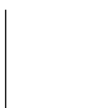
Maciej M Sysło

Człowiek – najlepsza inwestycja



Projekt współfinansowany ze środków Unii Europejskiej w ramach Europejskiego Funduszu Społecznego.

Różnorodne algorytmy obliczeń i ich komputerowe realizacje



Rodzaj zajęć: Wszechnica Informatyczna
Tytuł: Różnorodne algorytmy obliczeń i ich komputerowe realizacje
Autor: prof. dr hab. Maciej M Sysło

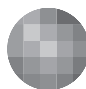
Redaktor merytoryczny: prof. dr hab. Maciej M Sysło

Zeszyt dydaktyczny opracowany w ramach projektu edukacyjnego
Informatyka+ — ponadregionalny program rozwijania kompetencji uczniów szkół ponadgimnazjalnych w zakresie technologii informacyjno-komunikacyjnych (ICT).
www.informatykaplus.edu.pl
kontakt@informatykaplus.edu.pl

Wydawca: Warszawska Wyższa Szkoła Informatyki
ul. Lewartowskiego 17, 00-169 Warszawa
www.wwsi.edu.pl
rektorat@wwsi.edu.pl

Projekt graficzny: FRYCZ I WICHA

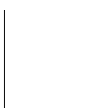
Warszawa 2010
Copyright © Warszawska Wyższa Szkoła Informatyki 2009
Publikacja nie jest przeznaczona do sprzedaży.



Projekt współfinansowany ze środków Unii Europejskiej w ramach Europejskiego Funduszu Społecznego.

Różnorodne algorytmy obliczeń i ich komputerowe realizacje

Maciej M. Sysło
Uniwersytet Wrocławski, UMK w Toruniu
syslo@ii.uni.wroc.pl, syslo@mat.uni.torun.pl



Streszczenie
Komputery nie przestały być maszynami matematycznymi – jak kiedyś je nazywano – i obecnie służą również do wykonywania różnych obliczeń. Kurs jest poświęcony m.in. algorytmom wyznaczania: dziesiętnej i binarnej reprezentacji liczb, obliczania wartości wielomianu, największego wspólnego dzielnika dwóch liczb (algorytm Euklidesa) oraz wartości potęgi. Motywacją dla wprowadzenia tych algorytmów jest chęć objaśnienia metody szyfrowania informacji z kluczem publicznym RSA, powszechnie stosowanej w kryptografii komputerowej.

Z podstaw algorytmiki omawiane są m.in. specyfikacja problemu, schematy blokowe algorytmów, podstawowe struktury danych (ciąg i tablica) oraz pracochłonność algorytmów. Część praktyczna zajęć jest poświęcona wprowadzeniu podstawowych instrukcji języka programowania (iteracyjnych i warunkowych oraz procedury i funkcji niestandardowej), wystarczających do zaprogramowania i uruchomienia komputerowych realizacji omówionych algorytmów. Język programowania służy głównie do zapisywania i testowania rozwiązań rozważanych problemów, a nie jest celem zajęć samym w sobie. Wykorzystywane jest również oprogramowanie edukacyjne, ułatwiające zrozumienie działania algorytmów i umożliwiające wykonywanie eksperymentów z algorytmami bez konieczności ich programowania. Przytoczono ciekawe przykłady zastosowań omawianych zagadnień.

W drugiej części kursu są przedstawiane wybrane techniki rozwiązywania problemów za pomocą komputera, m.in. podejście zachłanne (do wydawania reszty), przeszukiwanie z nawrotami (do ustawiania hetmanów na szachownicy) i rekurencja w realizacji wybranych algorytmów. Zajęcia praktyczne są poświęcone komputerowej realizacji wybranych technik algorytmicznych na odpowiednio dobranych przykładach problemów.

Rozważania są prowadzone na elementarnym poziomie i do ich wysłuchania oraz wzięcia udziału w warsztatach wystarczy znajomość informatyki wyniesiona z gimnazjum oraz matematyki na poziomie szkoły średniej. Te zajęcia są adresowane do wszystkich uczniów w szkołach ponadgimnazjalnych, zgodnie bowiem z nową podstawą programową, kształceniem umiejętności algorytmicznego rozwiązywania problemów mają być objęci wszyscy uczniowie.

W tych materiałach dla słuchaczy, algorytmy są zapisywane w języku Pascal. Odpowiednie wersje w języku C++ będą przekazane słuchaczom na zajęciach.

Spis treści

1. **Wprowadzenie** ... 6

2. ***Warm-up*** **– rozgrzewka** ... 6

3. **Obliczanie wartości wielomianu – schemat Hornera** ... 8
   - 3.1. Wyprowadzenie schematu Hornera ... 9
   - 3.2. Schemat blokowy schematu Hornera ... 11
   - 3.3. Komputerowa realizacja schematu Hornera ... 13

4. **System dziesiętny i system binarny** ... 15
   - 4.1. Zamiany liczby binarnej na liczbę dziesiętną ... 16
   - 4.2. Otrzymywanie binarnego rozwinięcia liczby dziesiętnej ... 17
   - 4.3. Długość rozwinięcia binarnego liczby ... 18

5. **Rekurencja** ... 19
   - 5.1. Wieże Hanoi ... 20
   - 5.2. Króliki i chaotyczny profesor – liczby Fibonacciego ... 24
   - 5.3. Inne spojrzenie na schemat Hornera ... 28
   - 5.4. Wyprowadzanie liczb od początku ... 29

6. **Szybkie obliczanie wartości potęgi** ... 32

7. **Algorytm Euklidesa** ... 34

8. **Algorytmy zachłanne** ... 37
   - 8.1. Problem wydawania reszty ... 37
   - 8.2. Zmartwienie kinomana ... 40
   - 8.3. Pakowanie najcenniejszego plecaka ... 41
   - 8.4. Najdłuższa droga na piramidzie ... 43
   - 8.5. Inne przykłady użycia metody zachłannej ... 44

9. **Przeszukiwanie z nawrotami** ... 45
   - 9.1. Wyjście z labiryntu metodą zgłębiania ... 45
   - 9.2. Rozmieszczanie hetmanów na szachownicy ... 47

10. **Dodatek. Algorytm, algorytmika i algorytmiczne rozwiązywanie problemów** ... 53

Literatura ... 55

1 WPROWADZENIE

Komputery nadal – jak kiedyś – służą również do wykonywania różnych obliczeń. Kurs jest poświęcony m.in. algorytmom: wyznaczania dziesiętnej i binarnej reprezentacji liczb, obliczania wartości wielomianu, największego wspólnego dzielnika dwóch liczb (algorytm Euklidesa) oraz wartości potęgi. Motywacją dla wprowadzenia tych algorytmów jest chęć objaśnienia metody szyfrowania informacji z kluczem publicznym RSA, powszechnie stosowanej w kryptografii komputerowej.

Część praktyczna zajęć jest poświęcona wprowadzeniu podstawowych instrukcji języka programowania (iteracyjnych i warunkowych oraz procedury i funkcji niestandardowej), wystarczających do zaprogramowania i uruchomienia komputerowych realizacji omówionych algorytmów. Język programowania służy głównie do zapisywania i testowania rozwiązań rozważanych problemów, a nie jest celem zajęć samym w sobie. Przytoczono ciekawe przykłady zastosowań omawianych zagadnień.

W drugiej części kursu są przedstawione wybrane techniki rozwiązywania problemów za pomocą komputera, m.in. podejście zachłanne (do wydawania reszty), przeszukiwanie z nawrotami (do ustawiania hetmanów na szachownicy) i rekurencja w realizacji wybranych algorytmów. Zajęcia praktyczne będą poświęcone komputerowej realizacji wybranych technik algorytmicznych na odpowiednio dobranych przykładach problemów.

Rozważania ogólne na temat algorytmiki, algorytmicznego myślenia i rozwiązywania problemów z pomocą komputerów są zamieszczone w Dodatku (rozdz. 10). Materiał tam zawarty może być dobrym podsumowaniem zajęć.

Jako literaturę rozszerzającą prowadzone tutaj rozważania polecamy podręczniki [2], a zwłaszcza książki [5] i [6].

i+

2 *WARM-UP* – ROZGRZEWKA

W tym rozdziale, naszym celem jest uruchomienie pierwszego programu. A zatem, Ci uczniowie, którzy mają już pewne doświadczenie w programowaniu, mogą go pominąć, zachęcamy ich jednak do szybkiego zapoznania się z zawartym tutaj materiałem.

Zacznijmy od bardzo prostego problemu. Każdy problem, który będziemy rozwiązywać na tych zajęciach, będzie opisany swoją nazwą oraz wyszczególnieniem danych i wyników. Taki opis problemu nazywa się **specyfikacją** – patrz rozdz. 10. A więc nasz problem ma postać:

Problem. Pole trójkąta
Dane: a, b, c – trzy dodatnie liczby.
Wynik: S – pole trójkąta, którego boki mają długości a, b i c.

Jeśli ten problem ma rozwiązać za nas komputer, to musimy go poinformować o danych, czyli o trzech liczbach, oraz, co ma być wynikiem, czyli jak pole trójkąta zależy od długości jego boków – posłużymy się tutaj **wzorem Herona**. Te czynności zapiszemy jako nasz pierwszy algorytm w postaci listy kroków[1], a powyższą specyfikację problemu przyjmiemy za specyfikację tego, co ma wykonać algorytm a później program.

Algorytm. Pole trójkąta
Dane: a, b, c – trzy dodatnie liczby.
Wynik: S – pole trójkąta, którego boki mają długości a, b i c.
Krok 1. Wprowadź trzy liczby: a, b, c.
Krok 2. Oblicz połowę długości obwodu trójkąta: $p = (a + b + c)/2$.
Krok 3. Oblicz pole trójkąta według wzoru:

[1] Algorytmy, poza programami, mogą być przedstawione również w postaci **schematów blokowych**. Przykładowy schemat blokowy jest zamieszczony w p. 3.2.

$$S=\sqrt{p(p-a)(p-b)(p-c)}$$

Program dla komputera w języku Pascal, będący realizacją tego algorytmu, jest przedstawiony w tabeli 1.

Tabela 1.
Pierwszy program

Wiersze programu	**Objaśnienia**
`Program Trojkat;`	`Program` i nazwa program, na końcu średnik
`var a,b,c,p:real;`	Deklaracja zmiennych – `real` oznacza, że wartości tych zmiennych nie muszą być całkowite
`begin`	Początek zasadniczej części programu
`read(a,b,c);`	Czytanie wartości zmiennych do programu
`p:=(a+b+c)/2;`	Obliczanie pomocniczej wielkości – połowy obwodu
`write(sqrt(p*(p-a)*(p-b)*(p-c)))`	Wypisywanie wielkości pola trójkąta o bokach *a*, *b*, *c*
`end.`	Koniec programu – na końcu kropka.

Ćwiczenie 1. Przepisz program z tabeli 1 w edytorze języka Pascal i uruchom go dla różnych trójek danych, np. 1, 1, 1; 2, 2, 2; 1, 2, 2; 1, 2, 4. Uruchom także dla danych: 1, 2, 3 oraz 1, 3, 1.

Jakie wyniki działania programu otrzymałeś w ostatnich dwóch przypadkach danych? Czy wiesz, dlaczego? Przypomnij więc sobie, jakie warunki muszą spełniać trzy liczby, aby mogły być długościami boków trójkąta? Jak więc wygląda trójkąt o bokach 1, 2, 3? A dlaczego nie otrzymałeś żadnego wyniku, tylko jakiś komunikat, dla liczb 1, 3, 1?

Aby trzy liczby mogły być długościami boków trójkąta, muszą spełniać **warunek trójkąta**, tzn. każda suma dwóch długości boków musi być większa od trzeciej liczby, czyli;

$$a + b > c, \quad a + c > b, \quad b + c > a.$$

Łatwo można sprawdzić, że te trzy nierówności są równoważne nierównościom:

$$p - a > 0, \quad p - b > 0, \quad p - c > 0.$$

Uwzględnienie tych warunków w algorytmie przekłada się na modyfikację trzeciego kroku, który otrzymuje postać:

Krok 3. Jeśli $p*(p-a)*(p-b)*(p-c) < 0$, to zakończ obliczenia z komunikatem, że podane trzy liczby nie są długościami boków w trójkącie. W przeciwnym razie oblicz pole trójkąta według wzoru:

$$S=\sqrt{p(p-a)(p-b)(p-c)}$$

W tabeli 2 jest zawarty program `Trojkaty _ mod`, w którym uwzględniono zmienioną postać kroku 3 – pojawia się instrukcja warunkowa `if … then … else`.

Tabela 2.
Zmodyfikowany program `Trojkat _ mod`

Wiersze programu	Objaśnienia
`Program Trojkat _ mod;`	Zmieniono nazwę.
`var a,b,c,p,q:real;`	Dodatkowa zmienna `q`.
`begin`	
`read(a,b,c);`	
`p:=(a+b+c)/2;`	
`q:=p*(p-a)*(p-b)*(p-c);`	Wyrażenie pod pierwiastkiem

`if q <= 0 then writeln('To nie sa boki trojkata')`
`else write('Pole trojkata = ',sqrt(q))`
	Instrukcja warunkowa:
	`if ... hen ...`
	`else ...`

`end.`	

Ćwiczenie 2. Uruchom program `Trojkat _ mod` z tabeli 2. Sprawdź jego działanie na trójkach liczb, które spełniają warunek trójkąta i na takich, które nie spełniają. W pierwszym przypadku wybierz trójki liczb, dla których znasz pole trójkąta, np. 1, 1 i 1.4142.

Faktyczna moc komputerów objawia się dopiero przy wielokrotnym wykonywaniu tych samych pleceń, być może dla różnych danych. Polecenie w algorytmie lub instrukcja w języku programowania, która umożliwia powtarzanie poleceń, nazywa się **iteracją** lub instrukcja **pętli**. W tabeli 3 przedstawiamy jedną z takich instrukcji – instrukcję `for` – która w tym programie służy do powtarzenia obliczeń ustaloną liczbę razy `n`.

Tabela 3.
Program `Trojkaty` – wielokrotne obliczanie pola trójkąta dla różnych długości boków

Wiersze programu	Objaśnienia
`Program Trojkaty;`	
`var i,n: integer;`	Zmienne użyte w instrukcji `for`
`a,b,c,p,q:real;`	
`begin`	
`read(n);`	Wczytanie liczby powtórzeń `n`
`for i:=1 to n do begin`	Instrukcja iteracyjna `for`
`write('Boki trojkata: ');`	
`read(a,b,c);`	
`p:=(a+b+c)/2;`	
`q:=p*(p-a)*(p-b)*(p-c);`	
`write('Pola trojkata: ');`	
`if q <= 0 then writeln('To nie sa boki trojkata')`	
`else writeln(sqrt(q):3:3);`	
`writeln;`	
`end`	Koniec instrukcji `for`
`end.`	

Inne instrukcje iteracyjne wprowadzimy na dalszych zajęciach. Program w tabeli 3 służy do obliczania pola trójkąta dla *n* trójek, będących długościami boków trójkąta.

Ada Augusta, córka Byrona, uznawana powszechnie za pierwszą programistkę komputerów, przełomowe znaczenie maszyny analitycznej Ch. Babbage'a, pierwowzoru późniejszych komputerów, upatrywała właśnie „w możliwości wielokrotnego wykonywania przez nią danego ciągu instrukcji, z liczbą powtórzeń z góry zadaną lub zależną od wyników obliczeń".

Ćwiczenie 3. Uruchom program `Trojkaty` na tych samych danych, na których testowałeś program w ćwicz. 2. Porównaj wyniki.

Podsumowując rozgrzewkę z programowania, wykonaliśmy trzy programy, o rozwijającej się strukturze:

- pierwszy – zawiera, poza deklaracjami zmiennych, tylko listę poleceń;
- w drugim – wykonywanie poleceń zależy od spełnienia warunku;
- trzeci zaś wykonuje polecenia w pętli.

Z takim, niewielkim arsenałem, możemy rozpocząć programować poważniejsze algorytmy.

3 OBLICZANIE WARTOŚCI WIELOMIANU – SCHEMAT HORNERA

Jak wspomnieliśmy już, jednym z najważniejszych kroków w algorytmach jest powtarzanie tej samej czynności (operacji), czyli **iteracja**. W tym punkcie zilustrujemy iterację na przykładzie szybkiego sposobu obliczania wartości wielomianu.

Obliczanie wartości wielomianu o zadanych współczynnikach jest jedną z najczęściej wykonywanych operacji w komputerze. Wynika to z ważnego faktu matematycznego, zgodnie z którym każdą funkcję (np. funkcje trygonometryczne) można zastąpić wielomianem, którego postać zależy od funkcji i od tego, jaką chcemy uzyskać dokładność obliczeń.

Na przykład, obliczanie wartości funkcji cos x w przedziale $0 \le x \le \pi/2$ z błędem nie większym niż 0.0009, można zastąpić obliczaniem wartości następującego wielomianu:

$$\cos x = 1 - 0.49670x^2 + 0.03705x^4.$$

3.1 WYPROWADZENIE SCHEMATU HORNERA

Zacznijmy od prostych ćwiczeń. Jeśli dane są wartości współczynników a, b i c wielomianu stopnia 2:

$$w(x) = ax^2 + bx + c,$$

to jego wartość można obliczyć wykonując zaznaczone mnożenia i dodawania: $a{\cdot}x{\cdot}x + b{\cdot}x + c$, czyli 3 mnożenia i dwa dodawania. Można także nieco inaczej – wyłączmy x z dwóch pierwszych składników – wtedy otrzymamy:

$$w(x) = (ax + b)x + c$$

i dla tej postaci obliczanie wartości tego wielomianu przyjmuje postać: $(a{\cdot}x + b){\cdot}x + c$, czyli są wykonywane tylko 2 mnożenia i dwa dodawania.

W podobny sposób można przedstawić wielomian stopnia 3:

$$v(x) = ax^3 + bx^2 + cx + d$$

najpierw wyłączam x z pierwszych trzech wyrazów, a następnie wyłączamy x z dwóch pierwszych wyrazów w nawiasie:

$$v(x) = ax^3 + bx^2 + cx + d = v(x) = v(x) = (ax^2 + bx + c)x + d = ((a \cdot x + b) \cdot x + c) \cdot x + d.$$

Widać z tego ostatniego wzoru, że wartość wielomiany stopnia 3 można obliczyć za pomocą 3 mnożeń i 3 dodawań.

Rozważmy teraz wielomian stopnia n:

$$w_n(x) = a_0x^n + a_1x^{n-1} + \ldots + a_{n-1}x + a_n \qquad (1)$$

i zastosujmy grupowanie wyrazów podobnie, jak w wielomianie stopnia 3 – najpierw wyłączamy x ze wszystkich wyrazów z wyjątkiem ostatniego, a następnie stosujmy ten sam krok wielokrotnie do wyrazów, które będą pojawiały się w najgłębszych nawiasach, aż pozostanie jednomian:

$$\begin{aligned} w_n(x) &= a_0x^n + a_1x^{n-1} + \ldots + a_{n-1}x + a_n = (a_0x^{n-1} + a_1x^{n-2} + \ldots + a_{n-1})x + a_n \\ &= ((a_0x^{n-2} + a_1x^{n-3} + \ldots + a_{n-2})x + a_{n-1})x + a_n = \\ &= (\ldots((a_0x + a_1)x + a_2)x + \ldots + a_{n-2})x + a_{n-1})x + a_n \end{aligned} \qquad (2)$$

Ćwiczenie 4. Przedstaw w postaci (2) następujące wielomiany:
$w(x) = 3x^4 - x^3 + 5x^2 + 7x - 2$
$w(x) = x^5 - x^3 + 4x^2 + 3x - 1$
$w(x) = x^6 - x^3 + x$

Zapiszmy teraz **specyfikację**, czyli dokładny opis rozważanego tutaj problemu:

Problem. Obliczanie wartości wielomianu
Dane: n – nieujemna liczba całkowita – stopień wielomianu;
$a_0, a_1, \ldots a_n$ – n+1 współczynników wielomianu;
z – wartość argumentu.
Wynik: Wartość wielomianu (2) stopnia n dla wartości argumentu $x = z$.

Aby obliczyć ze wzoru (2) wartość wielomianu dla wartości argumentu z, należy postępować następująco (y oznacza pomocniczą zmienną):

$y := a_0$
$y := yz + a_1$
$y := yz + a_2$
...
$y := yz + a_{n-1}$
$y := yz + a_n$

Schemat Hornera został podany przez jego autora w 1819 roku, chociaż znacznie wcześniej Isaac Newton stosował podobną metodę obliczania wartości wielomianów w swoich rachunkach fizycznych. W 1971 roku, A. Borodin udowodnił, że schemat Hornera jest optymalnym, pod względem liczby wykonywanych działań, algorytmem obliczania wartości wielomianu.

Wszystkie wiersze, z wyjątkiem pierwszego można zapisać w jednolity sposób – otrzymujemy wtedy:

$$\begin{aligned} &y := a_0 \\ &y := yz + a_i \quad \text{dla } i = 1, 2, \ldots, n. \end{aligned} \qquad (3)$$

Ten sposób obliczania wartości wielomianu nazywa się **schematem Hornera**.

Uwaga. W opisie algorytmu występuje instrukcja **przypisania**[2], wcześniej już użyta, np. $y := a_0$, w której symbol := jest złożony z dwóch znaków: dwukropka i równości. Przypisanie oznacza nadanie wielkości (zmien-

[2] Polecenie przypisania jest czasem nazywane niepoprawnie podstawieniem.

nej) stojącej po lewej strony tego symbolu wartości równej wartości wyrażenia (w szczególnym przypadku to wyrażenie może być zmienną) znajdującego się po prawej stronie tego symbolu. Przypisanie jest stosowane na przykład wtedy, gdy należy zmienić wartość zmiennej, np. $i := i + 1$ – w tym przypadku ta sama zmienna występuje po lewej i po prawej stronie symbolu przypisania. Polecenie przypisania występuje w większości języków programowania, stosowane są tylko różne symbole i ich uproszczenia dla jego oznaczenia.

Ćwiczenie 5. Zastosuj schemat Hornera do obliczenia wartości wielomianów z ćwicz. 4 w punkcie $z = 1$.

Możemy więc teraz zapisać:
Algorytm: Schemat Hornera
Dane: n – nieujemna liczba całkowita (stopień wielomianu);
$a_0, a_1, \ldots, a_n$ – n+1 współczynników wielomianu;
z – wartość argumentu.
Wynik: $y = w_n(z)$ – wartość wielomianu (1) stopnia n dla wartości argumentu $x = z$.
Krok 1. $y := a_0$ – początkową wartością jest współczynnik przy najwyższej potędze.
Krok 2. Dla $i = 1, 2, \ldots, n$ oblicz wartość dwumianu $y := yz + a_i$

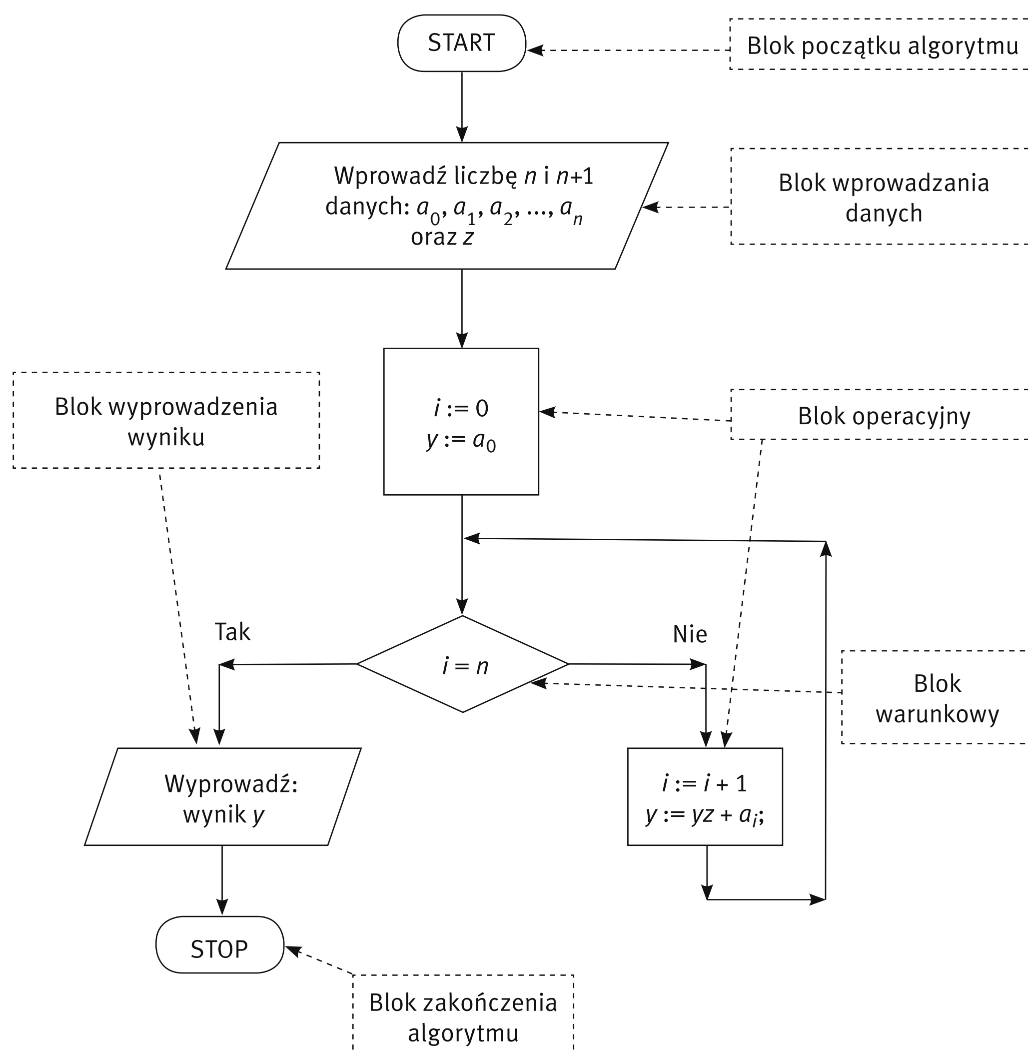
Wynika stąd, że aby obliczyć wartość wielomianu stopnia n wystarczy wykonać n mnożeń i n dodawań. Udowodniono, że jest to najszybszy sposób obliczania wartości wielomianu.

3.2 SCHEMAT BLOKOWY SCHEMATU HORNERA

Schemat blokowy algorytmu (zwany również **siecią działań** lub **siecią obliczeń**) jest graficznym opisem: działań składających się na algorytm, ich wzajemnych powiązań i kolejności ich wykonywania. W informatyce miejsce schematów blokowych jest pomiędzy opisem algorytmu w postaci listy kroków, a programem, napisanym w wybranym języku programowania. Należą one do kanonu wiedzy informatycznej, nie są jednak niezbędnym jej elementem, chociaż mogą okazać się bardzo przydatne na początkowym etapie projektowania algorytmów i programów komputerowych. Z drugiej strony, w wielu publikacjach algorytmy są przedstawiane w postaci schematów blokowych, pożądana jest więc umiejętność ich odczytywania i rozumienia. Warto nadmienić, że ten sposób reprezentowania algorytmów pojawia się w zadaniach maturalnych z informatyki.

Na rys. 1 jest przedstawiony schemat blokowy algorytmu **Schemat Hornera**. Jest on zbudowany z **bloków**, których kształty zależą od rodzaju wykonywanych w nich poleceń. I tak mamy:
- blok początku i blok końca algorytmu;
- blok wprowadzania (wczytywania) danych i wyprowadzania (drukowania) wyników – bloki te mają taki sam kształt;
- blok operacyjny, w którym są wykonywane operacje przypisania;
- blok warunkowy, w którym jest formułowany warunek;
- blok informacyjny, który może służyć do komentowania fragmentów schematu lub łączenia ze sobą części większych schematów blokowych.

Nie istnieje pełny układ zasad poprawnego konstruowania schematów blokowych. Można natomiast wymienić dość naturalne zasady, wynikające z charakteru bloków:
- schemat zawiera dokładnie jeden blok początkowy, ale może zwierać wiele bloków końcowych – początek algorytmu jest jednoznacznie określony, ale algorytm może się kończyć na wiele różnych sposobów;
- z bloków: początkowego, wprowadzania danych, wyprowadzania wyników, operacyjnego wychodzi dokładnie jedno połączenie, może jednak wchodzić do nich wiele połączeń;
- z bloku warunkowego wychodzą dwa połączenia, oznaczone wartościami warunku: Tak lub Nie;
- połączenia wychodzące mogą dochodzić do bloków lub do innych połączeń.



Rysunek 1.
Realizacja schematu Hornera w postaci schematu blokowego

Ćwiczenie 6. Zakreśl na schemacie blokowym na rys. 1 fragmenty odpowiadające poszczególnym krokom w opisie algorytmu **Schemat Hornera**. Zauważ, ten że schemat blokowy zawiera również fragmenty odpowiadające wczytywaniu danym i wypisywaniu wyników.

Ćwiczenie 7. Blok wprowadzania danych w schemacie na rys. 1 polega na wczytaniu liczby n a później na wczytaniu $n + 1$ liczb a_i. Narysuj szczegółowy schemat blokowy tego bloku.

Schematy blokowe mają wady, trudne do wyeliminowania. Łatwo konstruuje się z ich pomocą algorytmy dla obliczeń nie zawierających iteracji i warunków, którym w schematach odpowiadają rozgałęzienia, nieco trudniej dla obliczeń z rozgałęzieniami, a trudniej dla obliczeń iteracyjnych. Za pomocą schematów blokowych nie można w naturalny sposób zapisać rekurencji oraz objaśnić znaczenia wielu pojęć związanych z algorytmiką, takich np. jak programowanie z użyciem procedur, czyli podprogramów z parametrami. .

W dalszej części zajęć rzadko będziemy korzystać ze schematów blokowych, wystarczy nam bowiem umiejętność programowania.

Schemat Hornera ma wiele zastosowań. Może być wykorzystany m.in. do:

- obliczania wartości dziesiętnej liczb danych w innym systemie pozycyjnym (rozdz. 4);
- szybkiego obliczania wartości potęgi (rozdz. 6).

3.3 KOMPUTEROWA REALIZACJA SCHEMATU HORNERA

Zanim podamy komputerową realizację schematu Hornera, musimy ustalić, w jaki sposób będą reprezentowane w algorytmie dane i jak będziemy je podawać do algorytmu.

Danymi dla schematu Hornera są stopień wielomianu *n*, zapisane w postaci ciągu $n + 1$ liczb współczynniki wielomianu $a_0, a_1, a_2, \ldots, a_n$ oraz liczba *z*, dla której chcemy obliczyć wartość wielomianu. Stopień wielomianu jest liczbą naturalną (czyli dodatnią liczbą całkowitą), a pozostałe liczby mogą być całkowite lub rzeczywiste, czyli np. dziesiętne (tj. z kropką). Rodzaj danych liczb nazywa się **typem danych**. Przyjmujemy, że poza stopniem wielomianu, pozostałe dane są rzeczywiste, a więc mogą zawierać kropkę.

Dla wygody będziemy zakładać, że wiemy, ile będzie danych i ta liczba danych występuje na początku danych – jest nią liczba *n*, stopień wielomianu.

Zbiór danych, który jest przetwarzany za pomocą algorytmu, może być podawany (czytany) z klawiatury, czytany z pliku lub może być zapisany w algorytmie w odpowiedniej strukturze danych.

Dane podawane z klawiatury

Zapiszemy teraz schemat Hornera posługując się poleceniami języka Pascal. Przyjmujemy na początku, że dane są podawane z klawiatury – na początku należy wpisać liczbę wszystkich danych, a po niej kolejne elementy danych w podanej ilości. Po każdej danej liczbie naciskamy klawisz Enter.

Program, który jest zapisem schematu Hornera w języku Pascal, jest umieszczony w drugiej kolumnie w tab. 4. Język Pascal jest zrozumiały dla komputerów, które są wyposażone w specjalne programy, tzw. **kompilatory**, przekładające programy użytkowników na język wewnętrzny komputerów. Program w tab. 4 bez większego trudu zrozumie także człowiek dysponujący opisem schematu Hornera w postaci listy kroków. Słuchacze tych zajęć poznali już elementy języka Pascal podczas rozgrzewki (rozdz. 2), zatem tutaj tylko kilka słów komentarza. W wierszach nr 2 i 3 znajdują się **deklaracje** zmiennych – komputer musi wiedzieć, jakimi wielkościami posługuje się algorytm i jakie to są liczby, czyli jakiego są one **typu** – `integer` oznacza liczby całkowite (np. stopień wielomianu jest liczbą całkowitą), `real` oznacza liczby rzeczywiste, czyli np. liczby, które mogą zawierać kropkę dziesiętną. Polecenia w językach programowania nazywają się **instrukcjami**. Jeśli chcemy z kilku instrukcji zrobić jedną, to tworzymy z nich **blok**, umieszczając na jego początku słowo `begin`, a na końcu – `end`. Pojedyncze instrukcje kończymy **średnikiem**. Na końcu programu stawiamy kropkę. Pokrótce to wszystkie podstawowe zasady pisania programów w języku Pascal dla komputerów.

Jedna instrukcja wymaga wytłumaczenie, chociaż również jest dość oczywista i pojawiła się w rozdz. 2. W wierszach 9 – 12 znajdują się instrukcje, które realizują Krok 2 algorytmu polegający na wykonaniu jednej iteracji schematu Hornera. Dodatkowo, wcześniej jest czytany kolejny współczynnik wielomianu. Instrukcja, służąca do wielokrotnego wykonania innych instrukcji nazywa się **instrukcją iteracyjną** lub **instrukcją pętli**. W programie w tab. 4 ta instrukcja zaczyna się w wierszu nr 9 a kończy w wierszu nr 12:

```
for i:=1 to n do begin
...
end
```

Ta instrukcja iteracyjna, jak napisaliśmy, służy do powtórzenia dwóch instrukcji:

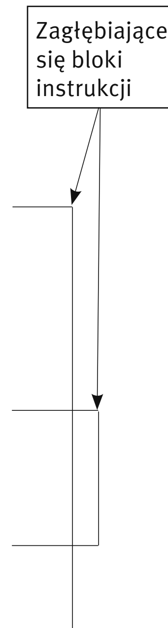
```
read(a);
y:=y*z+a
```

Inne rodzaje instrukcji iteracyjnej będą wprowadzane sukcesywnie.

Uwaga. Ponieważ założyliśmy, że dane są podawane z klawiatury, zmieniliśmy ich kolejność w stosunku do specyfikacji – liczba *z*, dla której jest obliczana wartość wielomianu, jest podawana na początku, przed współczynnikami wielomianu.

Tabela 4.
Program w języku Pascal (druga kolumna)

Lp.	Program w języku Pascal	Objaśnienia
1.	`Program Horner;`	nazwa programu
2.	`var i,n :integer;`	deklaracja zmiennych i, n
3.	`var a,y,z:real;`	deklaracja zmiennych a, y, z
4.	`begin`	początek głównego bloku programu
5.	`read(n);`	czytaj(n)
6.	`read(z);`	czytaj(z)
7.	`read(a);`	czytaj(a) – pierwszy współczynnik
8.	`y:=a;`	początkowa wartość wielomianu
9.	`for i:=1 to n do begin`	dla i:=1 do n wykonaj
10.	`read(a);`	czytaj(a)
11.	`y:=y*z+a`	modyfikacja wartości wielomianu
12.	`end;`	koniec iteracji
13.	`write(y)`	drukuj(y) – wartość wielomianu
14.	`end.`	koniec. – na końcu stawiamy kropkę



W wyniku wykonania tego programu, wypisywana jest na ekranie wartość wielomianu *y* w punkcie *z*. Ta wartość jest wypisywana w **postaci normalnej**, np. liczba 30 jest wypisywana jako:

```
3.00000000000000E+001
```

czyli 3.00000000000000·10^1. Nie jest to najwygodniejsza postać liczb, zwłaszcza liczb o niewielkich wartościach. Aby wybrać inny format wyprowadzanych liczb, możemy napisać:

```
write(y:2:2)
```

co będzie oznaczać, że wartość *y* zostanie wyświetlona (wydrukowana, wyprowadzona) z dwoma cyframi przed kropką i z dwoma cyframi po kropce.

Ćwiczenie 8. Uruchom program `Horner` i wykonaj obliczenia dla wybranego wielomianu i kilku jego argumentów.

Dane przechowywane z tablicy

Zmodyfikujemy teraz nasz program tak, aby:

- stopień i współczynniki wielomianu były czytane na początku i przechowywane w programie – do przechowania w programie współczynników użyjemy **tablicy**, która w języku programowania jest synonimem ciągu;
- można było obliczyć wartość wielomianu o wczytanych współczynnikach dla wielu argumentów – zakładamy w tym celu, że ciąg argumentów jest zakończony liczbą 0 (jest to tak zwany **wartownik** ciągu, gdyż jego rolą jest pilnowanie końca ciągu.

Przy tych założeniach, program będący **implementacją**[3] schematu Hornera, może mieć następująca postać:

[3] Terminem **implementacja** określa się w informatyce realizację algorytmu w postaci programu komputerowego.

```
Program Horner _ tablica;
 var i,n:integer;
     y,z:real;
     a  :array[0..100] of real;
     {Przyjmujemy, ze wielomian ma stopien co najwyzej 100}
begin
 read(n);
 for i:=0 to n do read(a[i]);
 writeln(' z     y');
 read(z);
 while z <> 0 do begin
  y:=a[0];
  for i:=1 to n do
   y:=y*z+a[i];
  write('    ',y:2:5);
  writeln;
  read(z)
 end
end.
```

Skomentowania wymaga użyta **instrukcja warunkowa**:

```
 while z <> 0 do begin
  y:=a[0];
  for i:=1 to n do
   y:=y*z+a[i];
  write('    ',y:2:5);
  writeln;
  read(z)
 end
```

W bloku tej instrukcji jest wykonywany schemat Hornera dla kolejnych wartości argumentu *z* tak długo, jak długo czytana liczba *z* jest różna od 0 (`z <> 0`).

Dodatkowo, w programie wprowadziliśmy drukowanie wyników w postaci tabelki.

Ćwiczenie 9. Uruchom program `Horner _ tablica` i wykonaj obliczenia dla wybranego wielomianu i kilku wartości jego argumentu. Popraw wygląd wyników.

4 SYSTEM DZIESIĘTNY I SYSTEM BINARNY

Obecnie w powszechnym użyciu jest **system dziesiętny**, zwany też **systemem dziesiątkowym**. Komputery natomiast wszystkie operacje wykonują w **systemie binarnym**, zwanym również **systemem dwójkowym**. Oba systemy, to dwa przykłady tzw. **systemu pozycyjnego** zapisywania wartości liczbowych. Powinny one być Wam znane z lekcji matematyki. Przypomnijmy jednak jeden i drugi system.

Liczba zapisana w systemie dziesiętnym, np. 357, oznacza wartość, na którą składają się trzy setki, pięć dziesiątek i siedem jedności, a zatem wartość tej liczby można zapisać jako $357 = 3 \cdot 100 + 5 \cdot 10 + 7 \cdot 1$, a także jako $357 = 3 \cdot 10^2 + 5 \cdot 10^1 + 7 \cdot 10^0$.

A zatem, dziesiętna liczba naturalna złożona z *n* cyfr:

$$d_{n-1}\, d_{n-2} \dots d_1\, d_0$$

oznacza wartość:

$$d_{n-1}10^{n-1} + d_{n-2}10^{n-2} + ... + d_1 10^1 + d_0 10^0. \quad (4)$$

Liczba 10 w tej reprezentacji nazywa się **podstawą systemu liczenia**. Do zapisywania liczb w systemie dziesiętnym są używane cyfry: 0, 1, 2, 3, 4, 5, 6, 7, 8 i 9. Ten sposób reprezentowania liczb nazywa się **system pozycyjnym**, gdyż w tym systemie znaczenie cyfry zależy od jej pozycji w ciągu cyfr, określającym liczbę. Na przykład, liczby 123 i 321 mają różną wartość, chociaż są złożone z tych samych cyfr, a liczba 111 jest złożona z trzech jedynek, z których każda ma inne znaczenie dla wartości tej liczby.

Dwa tysiące lat przed naszą erą Babilończycy stosowali system **kopowy** (tj. przy podstawie 60) – przypuszcza się, że stąd wziął się podział kąta pełnego na 360 stopni, godziny – na 60 minut, a minuty – na 60 sekund.

Ćwiczenie. 10. Wszystkim są znane tzw. **cyfry rzymskie**: I, V, X, L, C, D. M. Oznaczają one odpowiednio liczby: 1, 5, 10, 50, 100, 500, 1000. W tym systemie, liczba III ma dziesiętną wartość 3, a zatem każda z cyfr w tej liczbie ma takie samo znaczenie, a wartość jest obliczana przez dodawanie wartości cyfr. Zapisz w tym systemie liczby 2009 i 2012. Na liczbach rzymskich trudno wykonuje się podstawowe działania: dodawanie, odejmowanie i mnożenie. Liczby te były stosowane przez Rzymian głównie do zapisywania wartości liczbowych, a nie do wykonywania na nich działań.

Za podstawę systemu liczenia można przyjąć dowolną liczbę naturalną p większą od 1, np. 2, 5, 8, 16 czy 60. Jeśli p jest podstawą systemu liczenia, to liczby w tym systemie są zapisywane za pomocą cyfr ze zbioru $\{0, 1, ..., p - 1\}$.

W rozważaniach, związanych z komputerami, pojawiają się liczby w systemach o podstawach 2, 8 i 16. Naszą uwagę skupimy głównie na systemie o podstawie 2, a o innych systemach jest mowa w ćwiczeniach.

Jeśli $p = 2$, to system nazywa się **binarnym** lub **dwójkowym** – cyfry liczby, zapisanej w tym systemie, mogą mieć wartość 0 lub 1 i nazywają się **bitami**. Liczba naturalna a dana w systemie dziesiętnym ma następującą postać w systemie binarnym dla pewnego n:

$$a = b_{n-1}2^{n-1} + b_{n-2}2^{n-2} + ... + b_1 2^1 + b_0 2^0, \quad (5)$$

Za prekursora systemu binarnego uważa się G.W. Leibniza, który w pracy opublikowanej w 1703 roku zilustrował wykonywanie czterech podstawowych działań arytmetycznych na liczbach binarnych.

gdzie współczynniki $b_{n-1}, b_{n-2}, ..., b_1, b_0$ są liczbami 0 lub 1. W skrócie piszemy zwykle $a = (b_{n-1}b_{n-2} ... b_1b_0)_2$ i ciąg bitów w nawiasie nazywamy **binarnym rozwinięciem liczby** a. Na przykład mamy, $8 = (1000)_2$, $12 = (1100)_2$, $7 = (111)_2$. Cyfra b_{n-1} jest **najbardziej znaczącą**, a b_0 – **najmniej znaczącą** w rozwinięciu liczby a.

Jeśli się dobrze przyjrzymy wzorom oznaczonym przez (4) i (5), to zauważymy, że przypominają one specjalne wielomiany – współczynnikami tych wielomianów są cyfry rozwinięcia, a argumentem wielomianu – jest podstawa systemu liczenia. Aby więc obliczyć wartość dziesiętną liczby, danej w innym systemie, możemy skorzystać ze schematu Hornera, bardzo efektywnego algorytmu obliczania wartości wielomianu.

4.1 ZAMIANA LICZBY BINARNEJ NA LICZBĘ DZIESIĘTNĄ

Binarne rozwinięcie dziesiętnej liczby naturalnej (prawa strona we wzorze (5)) przypomina swoją postacią wielomian (patrz wzór (1)), gdzie a jest wartością wielomianu stopnia $n - 1$ o współczynnikach $b_{n-1}, b_{n-2}, ... b_1, b_0$ należących do zbioru $\{0, 1\}$, dla wartości argumentu $x = 2$. Z tej interpretacji rozwinięcia (5) wynika sposób obliczania dziesiętnej wartości liczby naturalnej a, gdy jest dane jej binarne rozwinięcie $(b_{n-1}b_{n-2} ... b_1b_0)_2$. Wystarczy zastosować schemat Hornera, który dla wielomianu, danego wzorem (5), przyjmuje następującą postać:

$$a = b_{n-1}2^{n-1} + b_{n-2}2^{n-2} + \ldots + b_1 2^1 + b_0 2^0$$
$$= (\ldots(b_{n-1}2 + b_{n-2})2 + \ldots + b_1)2 + b_0 \qquad (6)$$

Ćwiczenie 11. Napisz program służący do obliczania dziesiętnej wartości liczby *a* danej w postaci binarnego rozwinięcia $(b_{n-1}b_{n-2} \ldots b_1b_0)_2$.
Wskazówka. Zmodyfikuj odpowiednio program `Hornera _ Klawiatura`. Zwróć uwagę, że kolejne cyfry rozwinięcia binarnego liczby w postaci (5) są ponumerowane odwrotnie niż współczynniki wielomianu we wzorze (1) i zauważ przy tym, że indeks współczynnika odpowiada potędze liczby 2, przed którą stoi ten współczynnik. Nie powinno to jednak utrudnić Ci wykonania tego ćwiczenia.

Postać rozwinięcia binarnego (6) sugeruje bardzo prosty sposób obliczania wartości liczby *a* za pomocą kalkulatora.

Ćwiczenie 12. Oblicz za pomocą kalkulatora dziesiętną wartość liczby *a*, przedstawionej w postaci rozwinięcia binarnego. Wykorzystaj algorytm wynikający z postaci (6). Zauważ, że nie musisz korzystać z dodatkowej pamięci kalkulatora.
Wskazówka. Możesz skorzystać z kalkulatora dostępnego wśród akcesoriów środowiska Windows.

Wszystkie fakty podane powyżej przenoszą się na rozwinięcia liczby w systemie przy dowolnej podstawie *p*.

Ćwiczenie 13. Zmodyfikuj program napisany w ćwicz. 11 tak, aby mógł być stosowany do obliczania dziesiętnej wartości liczby *a* danej w postaci rozwinięcia przy podstawie *p*.

4.2 OTRZYMYWANIE BINARNEGO ROZWINIĘCIA LICZBY DZIESIĘTNEJ

Interesuje nas teraz czynność odwrotna – otrzymanie binarnego rozwinięcia dla danej dziesiętnej liczby naturalnej *a*.

Zauważmy, że we wzorze (5), czyli w binarnym przedstawieniu liczby *a*, wszystkie składniki z wyjątkiem ostatniego są podzielne przez 2, a zatem ten ostatni składnik, czyli b_0 jest resztą z dzielenia liczby *a* przez 2. Bity b_1, b_2,..., to reszty z dzielenia przez 2 kolejno otrzymywanych ilorazów. Dzielenie kończymy, gdy iloraz wynosi 0, gdyż wtedy kolejne reszty będą już cały czas równe 0, a zera na początku rozwinięcia binarnego nie mają żadnego znaczenia. Prześledź ten proces na przykładzie z rys. 2.

dzielenie	iloraz	reszta
749 | 2	374	**1**
374 | 2	187	**0**
187 | 2	93	**1**
93 | 2	46	**1**
46 | 2	23	**0**
23 | 2	11	**1**
11 | 2	5	**1**
5 | 2	2	**1**
2 | 2	1	**0**
1 | 2	0	**1**

Rysunek 2.
Przykład tworzenia binarnej reprezentacji liczby dziesiętnej 749. Otrzymaliśmy $749 = (1011101101)_2$

Zauważmy, że w powyższym algorytmie binarna reprezentacja liczby jest tworzona od końca, czyli od najmniej znaczącego bitu. W punkcie 5.4 omawiamy generowanie cyfr liczb od początku.

Wprowadzimy teraz dwie operacje, wykonywane na liczbach całkowitych, których wyniki są również liczbami całkowitymi, a które są przydatne przy obliczaniu ilorazu i reszty z dzielenia liczb całkowitych przez siebie.

Dla dwóch liczb całkowitych k i l definiujemy:

$r = k$ **mod** l – r jest resztą z dzielenia k przez l, czyli r spełnia nierówności $0 \leq r < l$, gdyż reszta jest nieujemną liczbą mniejszą od dzielnika;

$q = k$ **div** l – q jest ilorazem całkowitym z dzielenia k przez l, czyli q jest wynikiem dzielenia k przez l z pominięciem części ułamkowej.

Z definicji tych dwóch operacji wynika następująca równość:

$$k = l \cdot q + r = l \cdot (k \text{ div } l) + (k \text{ mod } l) \qquad (7)$$

Upewnij się, że dobrze rozumiesz te dwie operacje, które często występują w obliczeniach komputerowych na liczbach całkowitych. .

Ćwiczenie 14. Dla liczby naturalnej $l = 6$ i dla liczby naturalnej k, zmieniającej się od 0 co 1 do 20 oblicz wartości k div l oraz k mod l i sprawdź prawdziwość równości (7).

Ćwiczenie 15. Przyjmij, że $l = 2$, a więc interesuje nas iloraz i reszta z dzielenia liczby naturalnej k przez 2. Podaj, w zależności od parzystości liczby k, ile wynosi k mod 2 oraz k div 2.

Dotychczasowa dyskusja prowadzi nas do następującego algorytmu:

Algorytm: Zamiana dziesiętnej liczby naturalnej na postać binarną

Dane: Dziesiętna liczba naturalna a.

Wynik: Ciąg bitów, tworzących binarne rozwinięcie liczby a, w kolejności od najmniej znaczącego bitu.

Krok 1. Powtarzaj krok 2 dopóki a jest liczbą większą od zera, w przeciwnym razie zakończ algorytm.

Krok 2. Za kolejny bit (od końca) rozwinięcia przyjmij: a mod 2 i przypisz: $a := a$ div 2.

Poniżej przedstawiamy implementację tego algorytmu w języku Pascal.

```
Program Rozwiniecie _ binarne;
 var a:integer;
begin
 read(a);
 while a <> 0 do begin
  write(a mod 2,’  ‘);
  a:=a div 2
 end
end.
```

Ćwiczenie 16. W powyższym programie, kolejne bity rozwinięcia binarnego liczby a są wypisywane w kolejności od najmniej znaczącego, a więc odwrotnie, niż to się przyjmuje. Zmodyfikuj ten program tak, aby binarne rozwinięcie danej liczby było wyprowadzane od najbardziej znaczącego bitu.
Wskazówka. Posłuż się tablicą, w której będą przechowywane kolejno generowane bity.

4.3 DŁUGOŚĆ ROZWINIĘCIA BINARNEGO LICZBY

Osoby, które nie znają logarytmu, mogą opuścić ten podpunkt.

Liczby w komputerze są zapisywane w postaci binarnej. Interesujące jest więc pytanie, ile miejsca w komputerze, czyli ile bitów, zajmuje liczba naturalna a w postaci binarnej. Odpowiedź na to pytanie jest bardzo waż-

na w informatyce, nawet dzisiaj, kiedy można korzystać z niemal nieograniczonej pamięci komputerów i nie trzeba się obawiać, że jej zabraknie.

Najpierw rozważymy odwrotne pytanie: Jaką największą liczbę naturalną można zapisać na n bitach? Liczba taka ma wszystkie bity równe 1 w swoim rozwinięciu binarnym $(11...11)_2$, a więc jej wartość, jako sumą n początkowych wyrazów ciągu geometrycznego z ilorazem równym 2, wynosi:

$$2^{n-1} + 2^{n-2} + ... + 2^1 + 2^0 = 2^0 + 2^1 + ... + 2^{n-2} + 2^{n-2} = (1 - 2^n)/(1 - 2) = 2^n - 1$$

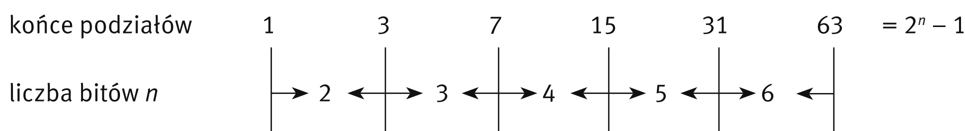
Ta równość ma ciekawą interpretację. Wartość sumy po lewej stronie jest liczbą o jeden mniejszą od liczby $(100...00)_2$, która w rozwinięciu binarnym składa się z n+1 bitów i ma tylko jeden bit równy 1 – najbardziej znaczący. Tą liczbą jest 2^n. Na rys. 3 pokazano, ile bitów potrzeba do przedstawienia w postaci binarnej liczb z poszczególnych przedziałów. Z ostatniej równości wynika, że liczba a może być zapisana na n bitach, jeśli spełnia nierówność:

$$2^n - 1 \geq a, \qquad \text{czyli} \qquad 2^n \geq a + 1.$$

Dla danej liczby a wybieramy najmniejsze n spełniające tę nierówność, gdyż początkowe zera w rozwinięciu liczby nie mają znaczenia. Logarytmując obie strony nierówności można dokładnie określić wartość n:

$$n = \lceil \log_2 (a + 1) \rceil.$$

Użyliśmy funkcji powała, której wartością jest najmniejsza liczba całkowita większa od liczby stojącej pod powałą, gdyż liczba bitów w rozwinięciu liczby a jest zawsze liczbą naturalną, a wartość logarytmu może nie być liczbą całkowitą,



Rysunek 3.
Liczba bitów potrzebnych do zapamiętania liczb a z poszczególnych przedziałów

Z powyższych rozważań należy zapamiętać, że liczba bitów potrzebnych do zapamiętania w komputerze liczby naturalnej jest w przybliżeniu równa logarytmowi przy podstawie 2 z wartości tej liczby. na przykład, liczba 1000 jest pamiętana na $\lceil \log_2 (1000 + 1) \rceil = 10$ bitach.

Przy tej okazji porównaj szybkość wzrastania funkcji liniowej i funkcji logarytmicznej.

Ćwiczenie 17. Narysuj w jednym układzie współrzędnych wykresy dwóch funkcji, logarytmicznej i liniowej. Utwórz także tabelę wartości obu funkcji dla liczb wybranych z przedziału $[1, 10^9]$. Zauważ, jak wolno rośnie funkcja logarytmiczna w porównaniu z funkcją liniową.

Funkcja logarytmiczna odgrywa bardzo ważną rolę w informatyce.

5 REKURENCJA

W rozwiązywaniu problemów często jest stosowana metoda, w której korzystamy ze znanych już rozwiązań innych problemów. Dotyczy to nie tylko problemów tutaj rozpatrywanych czy problemów matematycznych, ale tak postępujemy niemal w każdej dziedzinie.

Szczególnym przypadkiem metody, korzystającej z rozwiązań innych problemów, jest metoda, w której tymi „innymi problemami” jest rozwiązywany właśnie problem, ale dla mniejszych rozmiarów danych. W ta-

kiej sytuacji jest stosowany **algorytm rekurencyjny**, czyli algorytm, który odwołuje się do siebie. Rekurencję można stosować nie tylko do przedstawiania rozwiązań problemów, ale także do opisu wykonywania pewnych czynności, które mają charakter iteracyjny i powtarzana jest ta sama czynność. Ogólnie, rekurencja to sposób myślenia ułatwiający radzenie sobie z problemami i ich rozwiązaniami. Ze względu na znaczenie tego podejścia do rozwiązywania problemów z użyciem komputerów, algorytmiczne języki programowania, takie jak Pascal i C++ umożliwiają zapisywanie rozwiązań rekurencyjnych i ich wykonywanie. Rekurencyjne opisy rozwiązań komputerowych są na ogół bardzo zwarte – zobaczymy to na wielu prezentowanych tutaj przykładach – jest to jednak często okupione efektywnością obliczeń, znaczna część organizacji obliczeń rekurencyjnych jest bowiem przejmowana przez komputer. Można więc spojrzeć na rekurencję jak na sposób „przerzucania roboty na komputer" przy rozwiązywaniu problemów.

Przytoczymy tutaj dwa przykłady rekurencyjnych „procedur" postępowania, które są dalekie od charakteru naszych rozważań – sformułowaliśmy je jednak używając instrukcji języka Pascal – ale ilustrują rekurencyjny sposób myślenia. Pierwszy przykład pochodzi od znanego Radzieckiego informatyka, Andrieja Jerszowa.

```
procedure Jedz kaszkę;
 if talerz jest pusty then koniec jedzenia
 else begin
  weź łyżkę kaszki;
  Jedz kaszkę {wywołanie rekurencyjne}
 end

procedure Tańcz;
 if nie gra muzyka then koniec tańczenia
 else begin
  zrób krok;
  Tańcz       {wywołanie rekurencyjne}
 end
```

W obu przykładach, po `if` występuje warunek, który gwarantuje, że obie procedury mogą kończyć działanie, gdy talerz będzie pusty lub gdy przestanie grać muzyka. Można sobie wyobrazić jednak sytuację, że muzyka gra bez przerwy, wtedy druga procedura opisuje nieskończony algorytm rekurencyjnego wykonywania kroków tańca i dalszego tańczenia. Stąd też wniosek dla opisów algorytmów komputerowych, o których na ogół zakładamy, że kończą działanie – w opisach rekurencyjnych powinien wystąpić warunek zakończenia rekurencji, nosi on nazwę **warunku początkowego**.

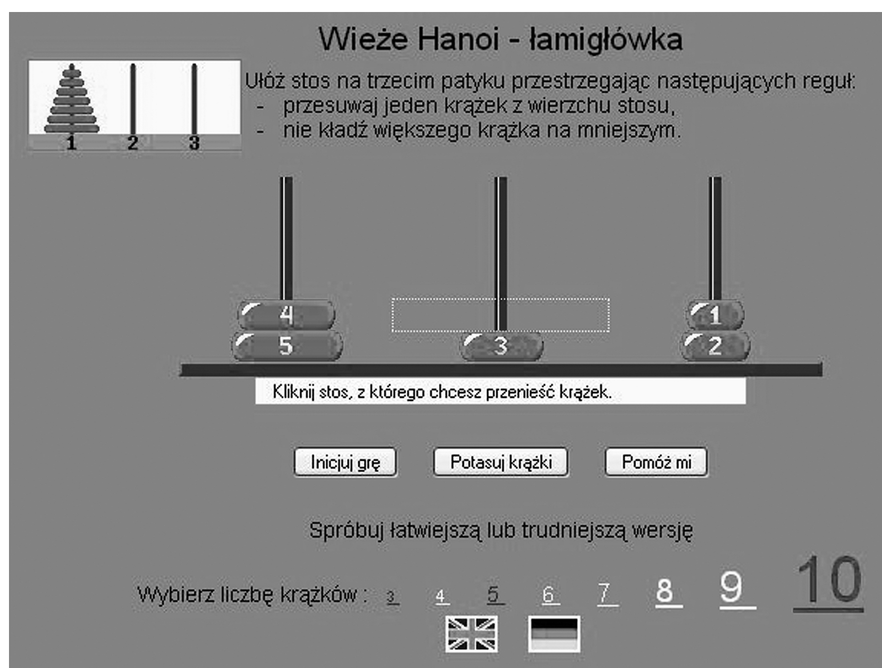
Dalej w tym rozdziale ilustrujemy rekurencyjny sposób rozwiązywania problemów przykładami, o których można powiedzieć, że są wzięte z życia, jak wieże Hanoi czy rozmnażanie się królików. W rozważaniu tych problemów można posłużyć się schematami graficznymi, co ułatwia zaobserwowanie odpowiednich zależności.

Ilustrujemy także, że podane wcześniej rozwiązania niektórych problemów można zapisać w postaci rekurencyjnej. Dotyczy to rozwiązań zawierających iterację. Z drugiej strony, pokazujemy również, ze rozwiązania rekurencyjne można zastąpić rozwiązaniami iteracyjnymi. Prowadzić to do konkluzji, że rekurencja jest faktycznie pewnym sposobem zapisu powtarzających się czynności (np. obliczeń), czyli innym zapisem iteracji.

W dalszych rozdziałach wielokrotnie przedstawiamy rozwiązania rozważanych problemów również w postaci rekurencyjnej, patrz rozdz. 6, 7 i 9.

5.1 WIEŻE HANOI

Tą nazwą opatruje się łamigłówkę logiczną, która jest klasycznym przykładem problemu algorytmicznego i służy w informatyce jako ilustracja wielu pojęć i metod rozwiązywania problemów, w tym zwłaszcza rekurencji. Zajrzyj na stronę http://wipos.p.lodz.pl/zylla/games/hanoi3p.html, gdzie możesz zapoznać się z tą łamigłówką, patrz rys.4.



Rysunek 4.
Rozwiązywanie łamigłówki Wież Hanoi w sieci

W łamigłówce mamy trzy paliki – oznaczmy je przez A, B i C – oraz pewną liczbę krążków różnej wielkości z otworami, nanizanych na palik A w kolejności od największego do najmniejszego, największy znajduje się na dole. Łamigłówka polega na przeniesieniu wszystkich krążków z palika A na palik B, z możliwością posłużenia się przy tym palikiem C, w taki sposób, że:

- pojedynczy ruch polega na przeniesieniu jednego krążka między dwoma palikami;
- w żadnej chwili rozwiązywania łamigłówki, większy krążek nie może leżeć na mniejszym.

Jeśli na paliku A znajduje się jeden krążek, to wystarczy przenieść go na palik B, a więc w tym przypadku wystarczy jeden ruch. Jeśli na A są dwa krążki, to łamigłówka również nie jest trudna: górny przenosimy na palik C, dolny na B i krążek z C przenosimy na B. Zatem wykonujemy w tym przypadku 3 ruchy.

Ćwiczenie 18. Na stronie http://wipos.p.lodz.pl/zylla/games/hanoi3p.html, najpierw rozwiąż tę łamigłówkę dla 3 krążków a później dla 4 i 5. W ilu ruchach ja rozwiązałeś?

Zapewne podczas tych prób poczyniłeś następujące spostrzeżenia:

- najmniejszy krążek znajduje się zawsze na górze któregoś z palików;
- na górze dwóch pozostałych palików, jeden z krążków jest mniejszy od drugiego i tylko ten jeden krążek można przenieść — żaden inny ruch między tymi dwoma palikami nie jest możliwy;
- z poprzedniego spostrzeżenia wynika, że najmniejszy krążek musi być przenoszony co drugi ruch, a w co drugim ruchu przenoszenie krążka jest jednoznacznie określone – musimy więc jedynie określić, na który palik należy przenieść najmniejszy krążek – szczegóły podajemy już w algorytmie.

Algorytm iteracyjny rozwiązania łamigłówki Wież Hanoi

Dane: Trzy paliki A, B i C, oraz *n* krążków o różnych średnicach, nanizanych od największego do najmniejszego na palik A. Krążki można przenosić między palikami tylko pojedynczo i nigdy nie można położyć większego na mniejszym.

Wynik: Krążki nanizane na palik B lub C – do uzyskania tego wyniku można wykonywać jedynie dopuszczalne przenoszenia.

Uwaga: Paliki A, B i C traktujemy tak, jakby były ustawione cyklicznie, zgodnie z ruchem wskazówek zegara, zatem dla każdego palika jest określony następny palik w tym porządku.
Krok 1. Przenieś najmniejszy krążek z palika, na którym się znajduje, na następny palik. Jeśli wszystkie krążki są ułożone na jednym paliku, to zakończ algorytm.
Krok 2. Wykonaj jedyne możliwe przeniesienie krążka, który nie jest najmniejszym krążkiem i wróć do kroku 1.

Ćwiczenie 19. Ponownie przejdź na stronę http://wipos.p.lodz.pl/zylla/games/hanoi3p.html i rozwiąż tę łamigłówkę dla 3, 4 i 5 krążków posługując się powyższym algorytmem. Czy wykonałeś tyle samo ruchów, co wykonując poprzednie ćwiczenie?

Rozwiązanie rekurencyjne
Zacznijmy od pytania, które ma Ciebie naprowadzić na rozwiązanie rekurencyjne: gdy wiesz, jak rozwiązać łamigłówkę dla trzech krążków, czy potrafisz wykorzystać to rozwiązanie, gdy masz o jeden krążek więcej?

Zastanów się, jaki powinien być układ krążków, gdy wreszcie można największy z nich przenieść z A na B – pamiętaj przy tym, że tego największego krążka nie można położyć na żadnym innym, gdyż wszystkie pozostałe krążki są od niego mniejsze. Zatem, gdy możemy przenieść największy krążek, wszystkie pozostałe krążki muszą być ułożone na paliku C zgodnie z ich wielkością. Stąd wynika, że możliwe jest rozwiązanie, które składa się z trzech etapów — zapiszemy je dla dowolnej liczby *n* krążków na paliku A:

1. Przenieś n – 1 górnych krążków z palika A na palik C, używając B.
2. Przenieś największy krążek z palika A na palik B.
3. Przenieś wszystkie krążki z palika C na palik B, używając A.

Zatem, jeśli umiemy rozwiązać tę łamigłówkę z trzema krążkami, to powyższą metodą możemy znaleźć rozwiązanie dla czterech krążków, na tej podstawie – dla pięciu krążków itd. Możemy skorzystać z tej zasady również w przypadku trzech krążków, gdyż wiemy, jak przenosi się dwa krążki. Powstaje jednak pytanie, czy opisane wyżej kroki mogą być zawsze wykonane? Krok 2 już objaśniliśmy. Kroki 1 i 3 są podobne, a ich wykonalność wynika stąd, że mamy do pełnej dyspozycji trzy paliki, gdyż palik zawierający największy krążek może być również swobodnie wykorzystany przez wszystkie pozostałe krążki. Możemy więc być pewni, że kroki 1 i 3 mogą być również wykonane. Powyższy opis posłuży nam teraz do zapisania rekurencyjnego rozwiązania łamigłówki Wież Hanoi w postaci algorytmu.

Algorytm rekurencyjny rozwiązania łamigłówki Wież Hanoi
Dane: Trzy paliki A, B i C, oraz *n* krążków o różnych średnicach, nanizanych od największego do najmniejszego na palik A. Krążki można przenosić między palikami tylko pojedynczo i nigdy nie można położyć większego na mniejszym.
Wynik: Krążki nanizane na palik B – do uzyskania tego wyniku można wykonywać jedynie dopuszczalne przenoszenia.
Krok 1. Jeśli $n = 1$, to przenieś krążek z palika A na B i zakończ algorytm dla $n = 1$.
Krok 2. {W tym przypadku liczba krążków na paliku A jest większa od 1.}
2a. Stosując ten algorytm, przenieś n – 1 krążków z A na C, używając B.
2b. Przenieś pozostały krążek z A na B.
2c. Stosując ten algorytm, przenieś n – 1 krążków z C na B, używając A.

Aby bardziej precyzyjnie opisać realizację powyższego algorytmu, oznaczmy przez (X→Y) przeniesienie krążka z palika X na palik Y, a przez (*k*,X,Y,Z) — przeniesienie *k* krążków z palika X na palik Y z wykorzystaniem palika Z, gdzie X, Y, Z oznaczają różne paliki spośród A, B i C. Przy tych oznaczeniach, powyższy algorytm można zapisać następująco:

Algorytm rekurencyjny rozwiązania łamigłówki Wież Hanoi (*n*,A,B,C)
Krok 1. Jeśli $n = 1$, to (A→B) i zakończ algorytm dla tego przypadku.
Krok 2. {W tym przypadku liczba krążków na paliku A jest większa od 1.}
2a. Zastosuj ten algorytm dla (n – 1,A,C,B).

2b. Przenieś pozostały krążek (A→B).
2c. Zastosuj ten algorytm dla (n – 1,C,B,A).

Można poczynić jeszcze dalsze uproszczenia w tym algorytmie. Zauważmy, że w krokach 1 i 2b jest wykonywane takie samo przeniesienie krążka, krok 1 jest więc szczególnym przypadkiem kroku 2, w którym dla n = 1 przyjmujemy, że kroki 2a i 2c są puste (czyli nic nie jest wykonywane). Ostatecznie otrzymujemy następujący algorytm rekurencyjny:

Algorytm rekurencyjny rozwiązania łamigłówki Wież Hanoi (n,A,B,C)
Krok 1. Jeśli n = 0, to nic nie rób i zakończ algorytm dla tego przypadku.
Krok 2. {W tym przypadku liczba krążków na paliku A jest większa od 0.}
2a. Zastosuj ten algorytm dla (n – 1,A,C,B).
2b. Przenieś pozostały krążek (A→B).
2c. Zastosuj ten algorytm dla (n – 1,C,B,A).

Opisy dwóch ostatnich wersji algorytmu rozwiązania łamigłówki Wież Hanoi są w pełni rekurencyjne. Układ parametrów w nazwie algorytmu umożliwia rekurencyjne wywołania wewnątrz algorytmu.

Ciekawe może być porównanie przebiegu obu wersji algorytmu iteracyjnego i rekurencyjnego.

Ćwiczenie 20. Przekonaj się, że kolejność przemieszczania krążków w rekurencyjnym algorytmie rozwiązania łamigłówki Wież Hanoi dla n = 3 jest dokładnie taka sama, jak w algorytmie iteracyjnym.

Implementacja rekurencyjnego rozwiązania łamigłówki Wież Hanoi ma bardzo prostą postać w języku Pascal:

```
procedure HanoiRek(n:integer; A,B,C:char);
 {Rekurencyjne rozwiazanie zagadki Wiez Hanoi}
begin
 if n>0 then begin
  HanoiRek(n-1,A,C,B);
  writeln('Przenies ',n,' z ',A,' na ',B);
  HanoiRek(n-1,C,B,A)
 end
end; {HanoiRek}
```

Ćwiczenie 21. Umieść procedurę `HanoiRek` w programie i użyj swojego programu do otrzymania rozwiązania dla n = 3 i n = 4. Porównaj te rozwiązania z otrzymanymi wcześniej.

Wcześniej nie proponowaliśmy utworzenia implementacji algorytmu służącego do iteracyjnego rozwiązywania łamigłówki Wież Hanoi, gdyż jest ona dość złożona. Jeśli jednak masz ochotę, to utwórz taką.

Liczba przeniesień krążków
Historia głosi, że łamigłówkę Wież Hanoi rozwiązują mnisi w jednym z klasztorów w Tybecie. Początkowo na paliku A znajdowały się n = 64 krążki i gdy wszystkie zostaną odpowiednio ułożone na paliku B, to nastąpi koniec świata. Dobrze jest więc wiedzieć, ile zajmie im to czasu. Proponujemy następujące ćwiczenie.

Ćwiczenie 22. Przyjmij, że grupa mnichów w Tybecie rozpoczęła rozwiązywać łamigłówkę Wież Hanoi z n = 64 krążkami na początku naszej ery i pracuje z prędkością komputera średniej mocy przenosząc 100 milionów pojedynczych krążków na sekundę! Oblicz, kiedy nastąpi koniec świata.

Oznaczmy przez h_n liczbę ruchów pojedynczymi krążkami, by rozwiązać łamigłówkę Wież Hanoi z n krążkami. Przypomnij sobie, ile wykonałeś ruchów dla trzech i czterech krążków – było to odpowiednio 7 i 15, a zatem $h_3 = 7$ i $h_4 = 15$. Co przypominają Ci te liczby? Nawet z niewielkim obyciem w dziedzinie algorytmów, można zauważyć, że te liczby są o jeden mniejsze od kolejnych potęg liczby 2, mamy więc $h_1 = 1 = 2^1 - 1$, $h_2 = 3 = 2^2 - 1$, $h_3 = 7 = 2^3 - 1$, i $h_4 = 15 = 2^4 - 1$ – potęga liczby 2 jest równa indeksowi liczby h. Na tej podstawie możemy przypuszczać, że $h_n = 2^n - 1$ dla dowolnej liczby krążków n. Wykażemy, że tak jest rzeczywiście, ale są to trochę trudniejsze rozważania. Ilustrujemy nimi również otrzymywanie zależności rekurencyjnych na podstawie algorytm rekurencyjnego oraz prosty sposób rozwiązywania takich zależności.

Z rekurencyjnego algorytmu rozwiązywania łamigłówki Wież Hanoi wynika następująca **zależność rekurencyjna** między liczbami h_n – zauważ, że h_n zależy od tej samej wielkości h_{n-1} tylko z mniejszym indeksem:

$$h_n = \begin{cases} 1 & n = 1 \\ 2h_{n-1} + 1 & n \geq 2 \end{cases}$$

Jeśli bowiem $n = 1$, to wykonujemy jedno przeniesienie krążka, a jeśli $n > 1$, to stosujemy rekurencyjny krok algorytmu, w którym dwa razy przenosimy tym samym algorytmem $n - 1$ krążków (wykonując przy tym h_{n-1} przeniesień krążków w obu przypadkach) i raz przenosimy największy krążek.

W jaki sposób na podstawie powyższej zależności rekurencyjnej można znaleźć wartości liczb h_n? W przypadku tej zależności jest to dość proste, gdyż możemy zastosować **metodę wstawiania**. Polega ona na tym, że wielkość stojącą po prawej stronie zależności rekurencyjnej można również wyrazić przez tę samą zależność. Zatem, dla $n - 1$ otrzymujemy z zależności następującą równość $h_{n-1} = 2h_{n-2} + 1$ i po wstawieniu do wzoru na h_n w zależności powyżej otrzymujemy:

$$h_n = 2h_{n-1} + 1 = 2(2h_{n-2} + 1) + 1 = 2^2h_{n-2} + 2 + 1$$

Wykonajmy jeszcze jeden krok wstawienia, by zauważyć pewną regularność. Z zależność rekurencyjnej dla $n - 2$ otrzymujemy $h_{n-2} = 2h_{n-3} + 1$ i po wstawieniu do powyższego wzoru otrzymujemy:

$$h_n = 2^2h_{n-2} + 2 + 1 = 2^2(2h_{n-3} + 1) + 2 + 1 = 2^3h_{n-3} + 2^2 + 2 + 1$$

Takie wstawianie kontynuujemy aż do otrzymania h_1 po prawej stronie znaku równości, wtedy bowiem możemy przerwać rekurencyjne zastępowanie według drugiej części zależności rekurencyjnej i zastąpić h_1 przez liczbę 1. Następuje to po $n - 1$ wstawieniach. Wtedy wyrażenie na h_n przyjmuje postać:

$$h_n = 2^{n-1}h_1 + 2^{n-2} + 2^{n-3} + \ldots + 2^2 + 2 + 1 = 2^{n-1} + 2^{n-2} + 2^{n-3} + \ldots + 2^2 + 2 + 1$$

Wartość sumy po prawej stronie ostatniej równości jest liczbą, która w rozwinięcie binarnym ma n jedynek. Następna liczba, czyli większa o 1, ma w rozwinięciu binarnym jedynkę na $n + 1$ pozycji, jest więc równa 2^n. Stąd otrzymujemy ostatecznie:

$$h_n = 2^n - 1$$

tak, jak przewidywaliśmy. Z pomocą tej równości rozwiąż teraz ćwicz. 22.

5.2 KRÓLIKI I CHAOTYCZNY PROFESOR – LICZBY FIBONACCIEGO

Jedne z najpopularniejszych liczb występujących w informatyce są związane z pytaniem, jakie zawarł Leonardo Fibonacci w swojej książce *Liber Abbaci* opublikowanej w 1202 roku.

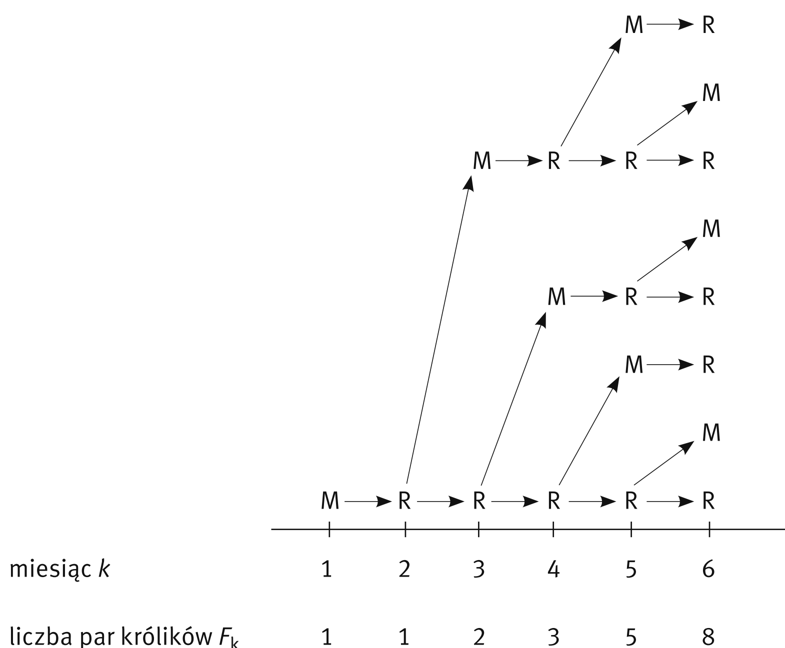
Szybkość rozmnażania się królików

Pytanie Fibonacciego dotyczyło szybkości rozmnażania się królików. Na początku mamy parę nowonarodzonych królików i o każdej parze królików zakładamy, że:

- nowa para staje się płodna po miesiącu życia;

- każda płodna para rodzi jedną parę nowych królików w miesiącu;
- króliki nigdy nie umierają.

Oryginalne pytanie Fibonacciego brzmiało: ile będzie par królików po roku czasu? Najczęściej pyta się, ile będzie par królików po upływie k miesięcy – oznaczmy tę liczbę przez F_k. Na rys. 5 przedstawiono rozrastanie się stada królików w ciągu kilku początkowych miesięcy (litera M oznacza parę młodych, a litera R – parę rozmnażających się już królików). W pierwszym i drugim miesiącu mamy tylko jedną parę, z tym że w drugim miesiącu może ona dać już parę młodych. Zatem w trzecim miesiącu są już dwie pary, przy czym tylko ta starsza może dalej rodzić młode. Stąd, w czwartym miesiącu są już trzy pary, z których dwie, a więc tyle, ile było już w poprzednim miesiącu, mogą rodzić. Czyli w następnym miesiącu mamy te trzy pary i dwie pary młodych, razem pięć par. I tak dalej. Otrzymujemy więc ciąg liczb: 1, 1, 2, 3, 5, 8, 13, 21, 34, 55 itd.



Rysunek 5.
Schemat rozrastania się stada królików w oryginalnym pytaniu Fibonacciego. M oznacza parę młodych, a R oznacza parę dorosłych, czyli rozmnażających się królików

Z tego przykładu i z warunków rozmnażania się królików wnioskujemy, że w kolejnym miesiącu liczba par królików będzie równa liczbie par z poprzedniego miesiąca, gdyż króliki nie wymierają, plus liczba par nowonarodzonych królików, a tych jest tyle, ile było par dwa miesiące wcześniej. Zatem kolejna liczba Fibonacciego jest sumą dwóch poprzednich liczb Fibonacciego. Stosując oznaczenie na liczbę par królików w danym miesiącu, ten wniosek można zapisać w następującej postaci $F_k = F_{k-1} + F_{k-2}$, gdzie k jest równe przynajmniej 3, aby można się było odwoływać do poprzednich miesięcy. Musimy zatem wartości dwóch pierwszych liczb Fibonacciego zdefiniować osobno i wprost, nie odwołując się do żadnych innych wartości tego ciągu. Z tych rozważań wynika więc następująca postać liczb Fibonacciego:

$$F_k = \begin{cases} 1 & k = 1, 2 \\ F_{k-1} + F_{k-2} & k \geq 3 \end{cases}$$

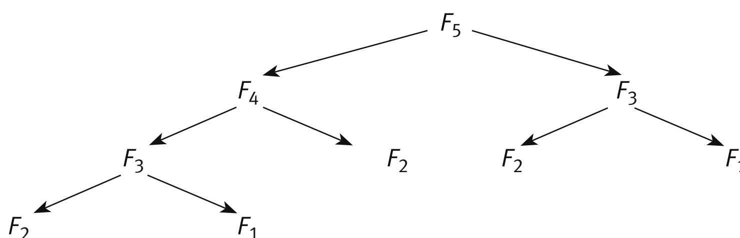
Ten wzór ma postać **zależności rekurencyjnej** i można go zastosować do obliczania wartości dowolnej liczby Fibonacciego. Implementacja wzoru rekurencyjnego w języku Pascal jest niemal automatyczna

```
function FibRek(k:integer):integer;
  {Wartoscia funkcji jest k-ta liczba Fibonacciego, obliczona
```

```
   za pomoca algorytmu rekurencyjnego}
begin
 if k<=2 then Fibrek:=1
 else FibRek:=FibRek(k-1)+FibRek(k-2)
end; {FibRek}
```

Ćwiczenie 23. Oblicz wartość F_5 wprost ze wzoru rekurencyjnego, posługując się przy tym funkcją `FibRek`.

Wykonując to ćwiczenie można zauważyć, że obliczenia dyktowane wzorem rekurencyjnym nie przebiegają w takiej samej kolejności, jak na rys. 5, zaczynamy je bowiem jakby „od końca”: chcemy obliczyć F_5, ale nie mamy dwóch poprzednich wartości F_4 i F_3. Musimy więc je wyznaczyć posługując się ... tym samym wzorem. Na rys. 6 jest przedstawiony schemat odwołań do wzoru rekurencyjnego w trakcie obliczania wartości F_5. Można zauważyć pewną rozrzutność, polegającą na tym, że kilka razy odwołujemy się do tych samych wartości: dwa razy do F_3, trzy razy do F_2 i dwa razy do F_1. Te odwołania są rzeczywiście wykonywane niezależnie jedno od drugiego w tym sensie, że na przykład z wartości F_3 obliczonej w lewym poddrzewie nie korzystamy, gdy potrzebna jest nam ta sama wartość w prawym poddrzewie.



Rysunek 6.
Schemat odwołań do wzoru rekurencyjnego w trakcie obliczania z tego wzoru wartości F_5

Niezadowoleni ze sposobu obliczania wartości liczb Fibonacciego, dyktowanego przez wzór rekurencyjny, wróćmy do sposobu obliczania tych wartości, który wynika wprost z definicji tych liczb, a który wykorzystaliśmy, by utworzyć ilustrację na rys. 5. Ostatecznie, z samej natury pytania Fibonacciego wynika, że aby w piątym miesiącu były jakieś króliki, to muszą być już w czwartym, a więc i w trzecim, drugim i pierwszym. A więc, liczby Fibonacciego powinno się dać obliczać zaczynając od najmniejszej. I tak rzeczywiście jest. Zapiszmy ten algorytm w sposób ścisły.

Algorytm iteracyjnego wyznaczania liczb Fibonacciego
Dana: Liczba naturalna k równa przynajmniej 1.
Wynik: Wartość liczby Fibonacciego F_k.
Krok 1. Jeśli $k = 1$ lub $k = 2$, to przyjmij $F_k = 1$ i zakończ algorytm.
Krok 2. Przyjmij *Fib*1:=1 oraz *Fib*2:=1. {*Fib*1 oznacza liczbę par królików w poprzednim miesiącu, a *Fib*2 – w dwa miesiące wcześniej.}
Krok 3. Wykonaj k – 2 razy następujące instrukcje przypisania:
Fib:=*Fib*1 + *Fib*2; {*Fib* jest wartością kolejnej liczby Fibonacciego.}
{Dwie następne instrukcje są przygotowaniem do następnej iteracji tego kroku.}
*Fib*2:=*Fib*1;
*Fib*1:=*Fib*;
Krok 4. Wartością F_k jest *Fib*.

Zapisanie implementacji iteracyjnego algorytmu obliczania wartości liczb Fibonacciego jest również proste.

```
function FibIter(k:integer):integer;
  {Wartoscia funkcji jest k-ta liczba Fibonacciego
   obliczona za pomoca algorytmu iteracyjnego.}
 var Fib,Fib1,Fib2:integer;
begin
 if (k=1) or (k=2) then FibIter:=1
 else begin
  Fib1:=1;  Fib2:=1;
  while k-2>0 do begin
   Fib:=Fib1+Fib2;
   Fib2:=Fib1;  Fib1:=Fib;
   k:=k-1
  end;  {while}
  FibIter:=Fib
 end
end; {FibIter}
```

Ćwiczenie 24. Umieść obie procedury `FibRek` i `FibIter` w jednym programie i sprawdź poprawność ich działania. By porównać ich czas działania, najpierw dowiedz się, jak mierzyć w programie czas działania fragmentu programu. Ponadto, ponieważ będziesz sprawdzał czas obliczeń dla większych liczb Fibonacciego, wprowadź typ `longint` w miejsce `integer`, by móc wykonywać działania na większych liczbach.

W książce [6], rozdz. 6 szczegółowo omówiono różne efektywne algorytmy, służące do obliczania wartości liczb Fibonacciego.

Chaotyczny profesor S i inne sytuacje

Jak wspomnieliśmy, liczby Fibonacciego są bardzo popularne w informatyce. Występują również w innych dziedzinach nauki, a także w sztuce. Swoją popularność w niewielkim stopniu zawdzięczają powiązaniom z rozmnażaniem się królików.

Rozwiązania następujących dwóch ćwiczeń prowadzą również do liczb Fibonacciego – najpierw wypisz rozwiązania dla $n = 1, 2, 3, 4$, a następnie posłuż się rozumowaniem rekurencyjnym. Polega ono na tym, że szukaną wartość dla ustalonego n staramy się uzależnić od wartości dla wcześniejszych n, czyli dla $n - 1$ i $n - 2$.

Ćwiczenie 25. Profesor S. bardzo chaotycznie chodzi po schodach i czasem pokonuje dwa schody, a czasem tylko jeden. Na ile sposobów profesor S. może wejść do swojego gabinetu, mieszczącego się na piętrze, które dzieli od parteru 10 schodów. Wyprowadź ogólną zależność na b_n – liczbę różnych sposobów pokonania n schodów przez profesora S.

Ćwiczenie 26. Dany jest zbiór $N_n = \{1, 2, 3, \ldots, n\}$ kolejnych liczb naturalnych. Na ile sposobów można wybrać podzbiór zbioru N_n, który nie zawiera dwóch kolejnych liczb?

Nieoczekiwanie, liczby Fibonacciego pojawiają się również w rozwiązaniu następującego ćwiczenia:

Ćwiczenie 27. Dany jest skończony, co najmniej trzyelementowy zbiór B dodatnich liczb całkowitych, nie większych niż miliard (tj. nie większych niż $10^9 = 1000000000$). Napisz program, który sprawdza, czy w zbiorze B istnieją trzy liczby, które mogą być długościami boków jakiegoś trójkąta.
Wskazówka. Weź jakiekolwiek trzy kolejne liczby Fibonacciego – czy można zbudować trójkąt o takich długościach boków? Weź jakikolwiek podzbiór liczb Fibonacciego – czy istnieją w nim trzy liczby, będące długościami boków jakiegoś trójkąta?

Liczby Fibonacciego całkiem niespodziewania i niewytłumaczalnie pojawiają się i występują w … obiektach przyrody. Na przykład, jako liczby spiral utworzonych przez łuski na szyszce lub pestki na tarczy słonecznika. Liczby Fibonacciego mają również związek z złotym (doskonałym) podziałem. Zainteresowanych tą tematyką odsyłamy do książki [6], rozdz. 6.

5.3 INNE SPOJRZENIE NA SCHEMAT HORNERA

Przy wyprowadzaniu schematu Hornera (patrz p. 3.1), podaliśmy następującą postać wielomianu stopnia n dla $n \geq 0$:

$$w_n(x) = a_0x^n + a_1x^{n-1} + \ldots + a_{n-1}x + a_n = (a_0x^{n-1} + a_1x^{n-2} + \ldots + a_{n-1})x + a_n$$

którą można również zapisać jako:

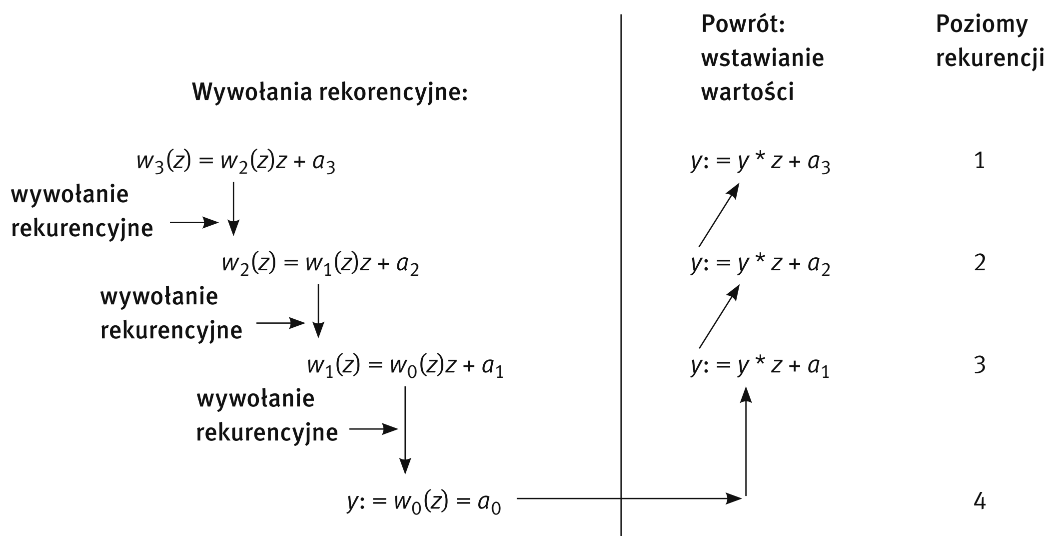
$$w_n(x) = w_{n-1}(x)x + a_n$$

gdzie $w_{n-1}(x) = a_0x^{n-1} + a_1x^{n-2} + \ldots + a_{n-1}$. Z tego wzoru wynika, że wartość wielomianu stopnia n może być obliczona, jeśli tylko znamy wartość odpowiedniego wielomianu stopnia $n - 1$ w tym samym punkcie. Sprawdźmy, czy to spostrzeżenie jest prawdziwe dla każdego n. Dla $n = 1$, na podstawie tego wzoru mamy $w_1(x) = w_0(x)x + a_1$, gdzie $w_0(x) = a_0$, a więc poprawnie. Dla $n = 0$ natomiast po prawej stronie pojawia się wielomian stopnia –1, a takiego nie znamy. Zatem, dla $n = 0$ nie możemy korzystać ze tego wzoru, przyjmujemy więc dodatkowo, że $w_0(x) = a_0$. Z tej dyskusji wynika, że wielomian stopnia n można zapisać w następującej postaci:

$$w_n(x) = \begin{cases} a_0 & \text{dla } n = 0 \\ w_{n-1}(x)x + a_n & \text{dla } n \geq 1 \end{cases}$$

Ta zależność jest rekurencyjną postacią schematu Hornera. Ciekawe jest przyjrzeć się, jak przebiega rekurencyjne obliczanie wartości wielomianu na podstawie tej zależności. Ilustrujemy to na rys. 7 dla wielomianu stopnia $n = 3$ i dla $x = z$. Jest to także dobra ilustracja całego procesu rekurencyjnych obliczeń – wywołania rekurencyjne są przedstawione w lewej części rysunku, a powrót z tych wywołań – w prawej części.

Z ilustracji na rys. 7 wynika także, że obliczenia wykonywane w trakcie powrotu z wywołań rekurencyjnych są w gruncie rzeczy realizacją iteracyjnego algorytmu Hornera. Można stąd wysnuć dwa wnioski, przynajmniej w tym przypadku: rekurencja jest innym sposobem realizacji iteracji (co potwierdza nasze wcześniejsze stwierdzenie) i jest przy tym sposobem bardziej rozrzutnym, gdyż pierwszy etap jest jakby zbędny w porównaniu z rozwiązaniem iteracyjnym.



Rysunek 7.
Schematyczne przedstawienie kolejności wykonywania działań w rekurencyjnym algorytmie obliczania wartości wielomianu trzeciego stopnia

Opis rekurencyjnej wersji schematu Hornera w języku Pascal ilustruje zwartość i elegancję tego sposobu realizacji algorytmów.

```
function HornerRek(n:integer; x:real):real;
 {Rekurencyjna realizacja schematu Hornera.
  Wspoczynniki wielomianu a[0..n] sa zmiennymi globalnymi
  dla tej funkcji.}
begin
 if n=0 then HornerRek:=a[0]
 else HornerRek:=HornerRek(n-1,x)*x+a[n]
end; {HornerRek}
```

Ćwiczenie 28. Umieść w jednym programie obie implementacje schematu Hornera, iteracyjną i rekurencyjną, i przetestuj ich działanie na przykładach kilku wielomianów.

5.4 WYPROWADZANIE LICZB OD POCZĄTKU

Zajmiemy się teraz problemem, którego bardzo proste i intuicyjne rozwiązanie korzysta z rekurencji.

Problem. Wypisywanie kolejnych cyfr liczby dziesiętnej
Dane: Liczba naturalna *m*, czyli nieujemna liczba całkowita.
Wynik: Wypisz kolejne cyfry dziesiętne liczby *m*.

Mając wypisaną liczbę, łatwo dostrzegamy jej kolejne cyfry – kolejnymi cyframi liczby 309 są: 3 – cyfra setek, 0 – cyfra dziesiątek, 9 – cyfra jedności. Związek, jaki istnieje między liczbą, a dokładniej – między wartością liczby a jej cyframi zapisujemy korzystając z faktu, że podstawą systemu liczenia jest liczba 10. W przypadku liczby 309 jest spełniona następująca równość:

$$309 = 3 \cdot 100 + 0 \cdot 10 + 9 \cdot 1,$$

czyli

$$309 = 3 \cdot 10^2 + 0 \cdot 10^1 + 9 \cdot 10^0.$$

W obu równościach występują te same cyfry danej liczby, które widzimy w jej zapisie, oraz wartości kolejnych potęg liczby 10, podstawy dziesiętnego systemu liczenia, którym posługujemy się na co dzień.

Podobny związek występuje wtedy, gdy liczba jest zapisana w **systemie binarnym**. Na przykład, liczba binarna $(100110101)_2$, czyli zapisana za pomocą cyfr 0 i 1, ma następujące przedstawienie, z którego można obliczyć jej wartość dziesiętną:

$$\begin{aligned}(100110101)_2 &= 1 \cdot 2^8 + 0 \cdot 2^7 + 0 \cdot 2^6 + 1 \cdot 2^5 + 1 \cdot 2^4 + 0 \cdot 2^3 + 1 \cdot 2^2 + 0 \cdot 2^1 + 1 \cdot 2^0 = \\ &= 1 \cdot 256 + 0 \cdot 128 + 0 \cdot 64 + 1 \cdot 32 + 1 \cdot 16 + 0 \cdot 8 + 1 \cdot 4 + 0 \cdot 2 + 1 \cdot 1 = \\ &= 309\end{aligned}$$

Z tego przykładu można wyciągnąć wniosek, że czym innym jest liczba (a dokładniej jej wartość) dziesiętna, a czym innym cyfry występujące w przedstawieniu liczby w wybranym systemie pozycyjnym. Zwracamy na to uwagę przede wszystkim dlatego, że w komputerze liczby są przedstawiane w systemie binarnym. Co więcej – i to jest tutaj ważniejszym uzasadnieniem dla naszych rozważań – w wielu algorytmach posługujemy się nie tylko liczbami, ale również cyframi wziętymi z ich postaci dziesiętnej, binarnej lub przy innej podstawie. .

Chcielibyśmy więc dysponować prostym algorytmem, który dla liczby dziesiętnej *m*, danej w komputerze, wypisuje na ekranie lub drukuje na papierze kolejne cyfry w binarnej lub dziesiętnej reprezentacji tej liczby, począwszy od najbardziej znaczącej cyfry.

Ćwiczenie 29. Nasz cel, określony powyżej, można osiągnąć nieco okrężna drogą. Najpierw znajdujemy i zapisujemy w komputerze kolejne cyfry (np. bity) reprezentacji, począwszy od najmniej znaczącej (w takiej kolejności są wyznaczane z dziesiętnej wartości liczby). Następnie, albo bezpośrednio wyprowadzamy te cyfry (lub tylko odczytujemy) w kolejności od najbardziej znaczącej, albo odwracamy przechowywaną reprezentację. Zaproponuj algorytm służący do odwracania danej reprezentacji liczby, w którym ta operacja jest wykonywana w tym samym miejscu (w tej samej tablicy), w którym znajduje się dana reprezentacja, algorytm nie korzysta więc z żadnych dodatkowych struktur danych.

Chodzi nam jednak tutaj o taką metodę, w której cyfry rozwinięcia byłyby generowane i wypisywane w kolejności od najbardziej znaczącej, bez jakiegokolwiek etapu pośredniego. Ale jak stwierdzić, że w danej liczbie *m*, przechowywanej w komputerze, najbardziej znacząca jest np. cyfra setek?

Zauważmy, że liczba *m* ma cyfrę jedności, jeśli ma wartość co najmniej 0, $m \geq 0$. Liczba *m* ma cyfrę dziesiątek, jeśli ma wartość co najmniej 10, czyli gdy $m \geq 10$ lub innymi słowy, gdy wynik dzielenia całkowitego *m* przez 10 jest większy od 0, tj. *m* div 10 ≥ 0. To spostrzeżenie można zastosować również do następnych cyfr, tj. cyfr setek, tysięcy itd. Aby jednak nie wykonywać sprawdzania tego warunku osobno i niezależnie dla każdej z cyfr, można sprawdzać go dla cyfr jedności, dziesiątek, setek itd., aż do osiągnięcia najbardziej znaczącej cyfry i dopiero od tego momentu wyprowadzać kolejne cyfry, począwszy od najbardziej znaczącej. Ten ogólny schemat można zapisać następująco[4]:

Przypomnijmy definicję dwóch funkcji, których argumenty i wartości są liczbami całkowitymi. Niech *m* będzie liczba naturalną, wtedy ***m* div 10** jest jej liczbą dziesiątek (nie mylić liczby dziesiątek z cyfrą dziesiątek), czyli liczbą otrzymaną z *m* przez odrzucenie ostatniej cyfry, a ***m* mod 10** jest cyfrą jedności w liczbie *m*, czyli jej ostatnią cyfrą. Działanie div jest **dzieleniem całkowitym**, a mod – **resztą**.

$$\text{Cyfry_liczby}(m) = \begin{cases} m & m < 10 \\ \text{Cyfry_liczby } (m \text{ div } 10) \text{ a po nich cyfra } (m \text{ mod } 10) & m \geq 10 \end{cases}$$

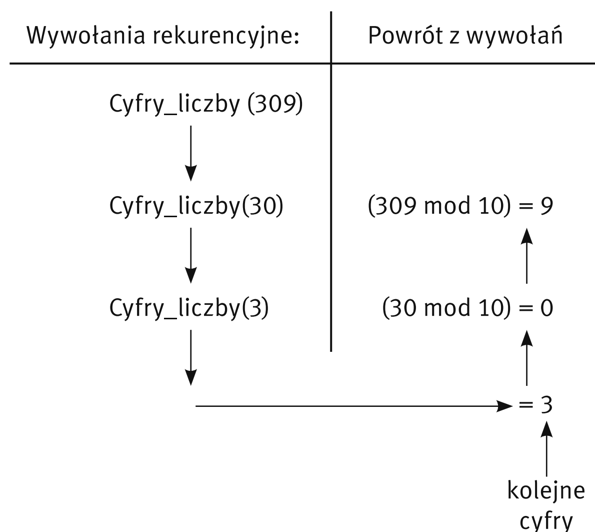
Jest to **zależność rekurencyjna** – ciąg kolejnych cyfr liczby *m* składa się z kolejnych cyfr liczby równej (*m* div 10) oraz cyfry (*m* mod 10). Ten proces rekurencyjnych odwołań jest wykonywany tak długo, aż bieżąca wartość *m* stanie się mniejsza od 10. Wtedy jest ona najbardziej znaczącą cyfrą liczby *m* danej na początku i wracając z kolejnych poziomów rekurencji, możemy dopisywać następne cyfry rozwinięcia. Zastosowanie powyższej równości do wyznaczenia kolejnych cyfr liczby 309, począwszy od najbardziej znaczącej, można ująć w schemat przedstawiony na rys. 8. Zatem rzeczywiście, cyfry liczby 309 są generowane za pomocą algorytmu działającego zgodnie z podanym wzorem rekurencyjnym w kolejności 3, 0, 9.

Podamy teraz opis procedury `Cyfry`, która służy do wyprowadzania kolejnych, dziesiętnych cyfr liczby `m`, począwszy od najbardziej znaczącej.

```
procedure Cyfry(m:integer);
 {Wypisywanie cyfr liczby m w kolejnosci od najbardziej znaczącej.}
 procedure KolejnaCyfra(m:integer);
  {Procedura rekurencyjna, ktora znajduje i wypisuje cyfry
   liczby m, w kolejnosci od najbardziej znaczacej.}
 begin
  if m < 10 then write(m)
  else begin
```

[4] Dla uniknięcia nieporozumień i zapewnienia jednoznaczności zapisu, operacje mod i div wraz ze swoimi argumentami są ujęte w nawiasy okrągłe.

```
    KolejnaCyfra(m div 10);
    write(m mod 10)
   end
  end; {KolejnaCyfra}
begin
  KolejnaCyfra(m)
end; {Cyfry}
```



Rysunek 8.
Przykład działania rekurencyjnego algorytmu generowania kolejnych cyfr w reprezentacji liczb

Ćwiczenie 30. Zapoznaj się z działaniem procedury `Cyfry` na przykładzie liczby $m = 309$. Sprawdź, że rzeczywiście cyfry liczby m są wyprowadzane w kolejności od najbardziej znaczącej.

Ćwiczenie 31. Zmień powyższą zależność rekurencyjną tak, aby mogła być użyta do wyznaczania kolejnych cyfry, od najbardziej znaczącej, w binarnym rozwinięciu danej liczby dziesiętnej m. Zastosuj otrzymaną zależność do uzyskania binarnej postaci liczby 309 w kolejności od najbardziej znaczącego bitu. A jak należy zmienić tę zależność, aby była poprawna dla dowolnej podstawy systemu liczenia?

Ćwiczenie 32. Zmodyfikuj opis procedury `Cyfry` do opisu procedury `Cyfry(m,b:integer)`, która będzie wypisywać, w kolejności od najbardziej znaczącej, cyfry liczby `m` w reprezentacji przy podstawie `b`.

Ćwiczenie 33. Jeśli już dobrze rozumiesz, jak działa procedura `Cyfry`, to postaraj się tak ją zmodyfikować, aby zamiast wyprowadzania kolejnych cyfr liczby `m` w reprezentacji przy podstawie `b`, jej wartością była liczba cyfr w liczbie m w reprezentacji przy podstawie `b`.

Ćwiczenie 34. Napisz program, w którym umieścisz procedurę `Cyfry(m,b:integer)`, i zastosuj go do znalezienia rozwinięcia wybranych liczb w kilku systemach pozycyjnych. W szczególności:
1. Sprawdź, że dla wybranej liczby m i podstawy $b = 10$, otrzymuje się dokładnie cyfry dziesiętne liczby m.
2. Dla wybranej liczby m, sprawdź, jakie są cyfry jej rozwinięcia w systemie o podstawach $b = 15$ i 60?

3. Napisz wersję procedury `Cyfry(m,b:integer)` dla systemu o podstawie $b = 16$, w której „cyfry” tej reprezentacji większe od 9 są oznaczone następująco: 10 = A, 11 = B, 12 = C, 13 = D, 14 = E i 15 = F.
4. Dla wybranej liczby *m*, znajdź jej cyfry w systemach o podstawach $b = 2, 4, 8$ i 16. Porównaj liczby cyfr w tych rozwinięciach – czy zauważasz, jaki jest związek między tymi liczbami? A jaki jest związek między cyframi lub grupami cyfr w tych rozwinięciach?
5. Na podstawie zależności zauważonych w punkcie 4 zaproponuj algorytm, służący do zmiany reprezentacji liczby między dwoma systemami o podstawach 2 i 8.

Ćwiczenie 35. Jak należy zmienić procedurę `Cyfry(m,b)`, by wyprowadzane były kolejne cyfry liczby `m` ale od tyłu?
Podpowiedź. Wystarczy zmienić kolejność dwóch instrukcji. Których?

Ćwiczenie 36. Określ, ile razy, w zależności od wartości liczb *m* i *b*, jest wykonywana operacja mod w procedurze `Cyfry(m,b:integer)`.
Wskazówka. To zadanie jest w gruncie rzeczy pytaniem o to, ile cyfr ma liczba dziesiętna zapisana w systemie o podstawie *b*. W książce *Algorytmy* (p. 7.1 w [5]) odpowiadamy na to pytanie dla $b = 2$.

Rekurencja – podsumowanie

Na zakończenie tego rozdziału poświęconego rekurencji zwróćmy jeszcze raz uwagę na dwie podstawowe cechy algorytmów rekurencyjnych i ich komputerowych realizacji:

1. Rekurencyjny zapis rozwiązania, czyli w postaci procedury rekurencyjnej (np. w językach Pascal lub C++), jest bardzo zwięzły, znacznie krótszy niż zapis odpowiedniej procedury iteracyjnej. Zawdzięcza się to m.in. odwołaniom do tej samej procedury.
2. Zwartość zapisu algorytmów w postaci rekurencyjnej zawdzięcza się również temu, że organizacją rekurencyjnego wykonania algorytmu w komputerze zajmuje się kompilator i nie musimy bezpośrednio rozpisywać szczegółowo wszystkich kroków. To „zrzucenie roboty na komputer”, czyli przekazanie mu części organizacyjnej działania algorytmu, objawia się na ogół zwiększeniem czasu jego działania podczas wykonywania obliczeń.

Wynika stąd sugestia, by stosować rekurencję w opisach algorytmów, ale takie algorytmy realizować na komputerze zamieniając rekurencję na iterację.

6 SZYBKIE OBLICZANIE WARTOŚCI POTĘGI

Binarna reprezentacja liczb naturalnych jest źródłem szybkich metod obliczania wartości potęgi x^m, gdzie *n* jest liczbą naturalną, a *x* może być dowolną liczbą rzeczywistą (wartość podstawy *x* nie ma znaczenia dla naszych rozważań). Metody te znajdują zastosowanie w algorytmach kryptograficznych, w których są obliczane wartości potęg o bardzo dużych wykładnikach.

Posłużmy się najpierw przykładem. Przypuśćmy, że chcemy obliczyć wartość potęgi. x^{22}. Proste wymnożenie podstawy przez siebie $x^{22} = x \cdot x \cdot \ldots \cdot x$ to 21 mnożeń. A skorzystajmy z binarnej reprezentacji wykładnika potęgi. Można go przedstawić jako sumę potęg liczby 2:

$$22 = 2 + 4 + 16 = 2^1 + 2^2 + 2^4$$

wtedy nasza potęga przyjmuje postać $x^{22} = x^{2+4+16} = x^2 \cdot x^4 \cdot x^{16}$ i można ją obliczyć wielokrotnie podnosząc do kwadratu podstawę *x* i mnożąc przez siebie odpowiednie czynniki. Dla potęgi 22 obliczanie potęgi przebiega następująco: x^2, $x^4 = (x^2)^2$, $x^8 = (x^4)^2$, $x^{16} = (x^8)^2$ i mnożymy przez siebie $x^2 \cdot x^4 \cdot x^{16}$. W sumie wykonujemy 6 mnożeń.

Korzystają z przedstawienia wykładnika w reprezentacji binarnej $22 = (10110)_2$ w postaci schematu Hornera, wykładnik można zapisać $22 = 1 \cdot 2^4 + 0 \cdot 2^3 + 1 \cdot 2^2 + 1 \cdot 2^1 + 0 \cdot 2^0 = (((2 + 0)2 + 1)2 + 1)2 + 0$, a stąd otrzymujemy:

$$x^{(((2+0)2+1)2+1)2+0} = x^{(((2+0)2+1)2+1)2+0} = (x^{((2+0)2+1)2+1})^2 = (x^{((2+0)2+1)2}x)^2 = (x^{(2+0)2+1})^2x)^2$$
$$= (x^{(2+0)2}x)^2x)^2 = (x^{(2+0)})^2x)^2x)^2 = (((x^2)^2x)^2x)^2.$$

Korzystając z tego zapisu, potęgę x^{22} obliczamy wykonując kolejno następujące mnożenia: x^2, $x^4 = (x^2)^2$, $x^5 = x^4 \cdot x$, $x^{10} = (x^5)^2$, $x^{11} = x^{10} \cdot x$, $x^{22} = (x^{11})^2$. W sumie wykonujemy również 6 mnożeń.

Ćwiczenie 37. W oparciu o jedną i drugą metodę naszkicowaną powyżej, podaj kolejność wykonywania mnożeń podczas obliczania wartości potęgi x^{313}. Ile mnożeń jest wykonywanych w obu przypadkach?

Obie metody potęgowania korzystają z binarnego rozwinięcia wykładnika, ale różnią się tym, że w pierwszym przypadku to rozwinięcie jest przeglądane od najmniej znaczącego bitu (mówimy o metodzie **od prawej do lewej**), a w drugim – od najbardziej znaczącego (czyli metodą **od lewej do prawej**). Ponieważ reprezentacja binarna liczby naturalnej jest tworzona od najmniej znaczącego bitu (zob. p. 4.2), podamy teraz algorytm obliczania wartości potęgi, który wykonuje potęgowanie rozkładając wykładnik na postać binarną (postać ta jednak nie jest zachowywana w algorytmie). Szybkie potęgowanie, wykorzystujące schemat Hornera pozostawiamy jako ćwiczenie.

Binarny algorytm potęgowania „od prawej do lewej”
Dane: Liczba naturalna m i dowolna liczba x.
Wynik: Wartość potęgi $y = x^m$.
Krok 0. {Ustalenie początkowych wartości potęgi y i zmiennych pomocniczych z i l.}
$y := 1$; $l := m$; $z := x$;
Krok 1. Jeśli l mod 2 = 1 (czyli l jest liczbą nieparzystą), to $y := y \cdot z$;
Krok 2. $l := l$ div 2; jeśli $l = 0$, to zakończ algorytm – wynikiem jest bieżąca wartość y.
Krok 3. $z := z \cdot z$; wróć do kroku 1.

Ćwiczenie 38. Wykonaj powyższy algorytm dla $m = 22$ i $m = 313$. Wypisz kolejne wartości zmiennych y, l, z i porównaj kolejne wartości zmiennej y z wcześniej otrzymanymi wartościami potęgi dla tych wartości wykładnika.

Ćwiczenie 39. Napisz program w języku Pascal, będący realizacją algorytmu podnoszenia do potęgi „od prawej do lewej”. Jako wynik, dla danego wykładnika m, wyprowadzaj liczbę mnożeń wykonywanych w tym algorytmie dla obliczenia wartości x^m.

Szybkie algorytmy potęgowania są stosowane w algorytmach szyfrujących, w których wykładniki potęg są bardzo dużymi liczbami.

Algorytm rekurencyjny

Zauważmy, że jeśli m jest liczbą parzystą, to zamiast obliczać wartość potęgi x^m, wystarczy obliczyć $y = x^{m/2}$ a następnie ponieść y do kwadratu. Jeśli m jest liczbą nieparzystą, to $m - 1$ jest liczbą parzystą. A zatem mamy następującą zależność:

$$x^m = \begin{cases} 1 & \text{jeśli } m = 0 \\ (x^{m/2})^2 & \text{jeśli } m \text{ jest liczbą parzystą} \\ (x^{(m-1)/2})^2 x & \text{jeśli } m \text{ jest liczbą nieparzystą} \end{cases}$$

która ma charakter **rekurencyjny** – po prawej stronie równości są odwołania do potęgowania, czyli do tej samej operacji, której wartości liczymy, ale dla mniejszych wykładników. Pierwszy wiersz w powyższej równości to tzw. **warunek początkowy** – służy do zakończenia (zastopowania) odwołań rekurencyjnych do coraz mniejszych argumentów, by cały proces obliczeń zakończył się.

Jeśli chcielibyśmy obliczyć x^{22}, to powyższa zależność rekurencyjna prowadzi nas przez następujące odwołania rekurencyjne: $x^{22} = (x^{11})^2 = ((x^5)^2x)^2 = (((x^2)^2x)^2x)^2$. Identyczną zależność otrzymamy rozkładając wykładnik 22 na postać binarną.

Ćwiczenie 40. Podaj w jakiej kolejności będą obliczane potęgi w trakcie obliczania wartości x^{51} za pomocą algorytmu rekurencyjnego.

Poniżej zamieszczamy funkcję, będącą implementacją rekurencyjnego obliczania wartości potęgi.

```
function PotegaRek(m:integer; x:real):real;
  {Wartoscia funkcji jest x podniesione do potegi m
   obliczona wedlug rekurencyjnego wzoru.}
 function Kwadrat(x:real):real;
  {Wartoscia tej funkcji jest kwadrat argumentu x.}
 begin
  Kwadrat:=x*x
 end; {Kwadrat}
begin
 if m=1 then PotegaRek:=x
 else
  if m mod 2=0
  then PotegaRek:=Kwadrat(PotegaRek(m div 2,x))
  else PotegaRek:=Kwadrat(PotegaRek((m-1) div 2,x))*x
end; {PotegaRek}
```

Ćwiczenie 41. Zwróć uwagę, w jaki sposób unikamy w funkcji `PotegaRek` obliczania dwa razy tej samej wartości potęgi, gdy wykładnik jest liczbą parzystą. Umieść tę funkcję w programie i sprawdź jej działanie na kilku wybranych potęgach.

7. ALGORYTM EUKLIDESA

Przedstawiamy w tym punkcie algorytm, który wśród przepisów opatrywanych tych mianem, został odkryty jako jeden z najwcześniejszych, bo jeszcze w starożytności, przez Euklidesa.

Danymi są dwie nieujemne liczby całkowite *m* i *n*. Liczba *k* jest **największym wspólnym dzielnikiem** *m* i *n*, jeśli dzieli *m* oraz *n* i jest największą liczbą o tej własności – oznaczamy ją przez NWD(*m*,*n*). **Algorytm Euklidesa** jest najstarszą metodą znajdowania największego wspólnego dzielnika dwóch liczb (patrz ramka obok).

Euklides stosował swój algorytm do znajdowania najdłuższego odcinka mieszczącego się całkowitą liczbę razy w dwóch danych odcinkach. Było to około 300 roku p.n.e., na długo przed pojawieniem się określenia algorytm.
Algorytm Euklidesa jest jednym z najczęściej przytaczanych algorytmów w książkach informatycznych i matematycznych. Powodem jest nie tylko szacunek dla jego „wieku”, ale to, że ma wiele zastosowań w rozwiązywaniu problemów rachunkowych i posługując się nim, można ilustrować wiele podstawowych pojęć i własności informatycznych.

Ćwiczenie 42. Pierwszą metodą, jaka może przyjść na myśl, gdy chcemy znaleźć największy wspólny dzielnik dwóch danych liczb *m* i *n*, jest sprawdzanie podzielności tych liczb przez kolejne liczby naturalne, począwszy od 2, a skończywszy na mniejszej z *m* i *n*. Ile dzieleń należy wykonać, aby obliczyć tą metodą NWD(24,48) oraz NWD(46,48)? A dla dowolnych *m* i *n*? Opisz tę metodę w postaci algorytmu w wybranej przez siebie reprezentacji. Możesz opisać tę metodę w postaci programu komputerowego.

Nie możesz być chyba zadowolony z efektywności tego algorytmu, zwłaszcza dla przykładowych danych liczbowych – w pierwszym przypadku, gdy jedna z liczb dzieli drugą, od razu widać, że ta mniejsza jest poszukiwanym największym wspólnym dzielnikiem obu liczb. A jeśli żadna z liczb nie dzieli drugiej, tak jak w drugim przypadku?

Załóżmy, że $n \geq m$. Wtedy dzieląc n przez m otrzymujemy następującą równość:

$$n = qm + r, \qquad \text{gdzie} \qquad 0 \leq r < m \tag{8}$$

Wielkości q i r są odpowiednio **ilorazem** i **resztą** z dzielenia n przez m. Dla danych z ćwicz. 42, równość (8) przyjmuje postać: $48 = 2 \cdot 24$ i $48 = 1 \cdot 46 + 2$. Wynikają stąd następujące wnioski:

- jeśli $r = 0$, to NWD$(m,n) = m$, czyli jeśli jedna z liczb dzieli drugą bez reszty, to mniejsza z tych liczb jest ich największym wspólnym dzielnikiem;
- jeśli $r \neq 0$, to równość (8) można zapisać w postaci $r = n - qm$; a stąd wynika, że każda liczba dzieląca n i m dzieli całe wyrażenie po prawej stronie tej równości, a więc dzieli również r; zatem, największy wspólny dzielnik m i n dzieli również resztę r.

Te dwa wnioski można zapisać w postaci następującej równości:

$$\text{NWD}(m,n) = \text{NWD}(r,m), \tag{9}$$

w której przyjęliśmy, że NWD$(0,m) = m$, gdyż 0 jest podzielne przez każdą liczbę różną od zera. Zastosujmy teraz te obserwacje do obliczenia NWD(25,70). Otrzymamy:

$$\text{NWD}(25,70) = \text{NWD}(20,25) = \text{NWD}(5,20) = \text{NWD}(0,5) = 5,$$

gdyż stosując równość (8) otrzymujemy kolejno:

$$70 = 2 \cdot 25 + 20$$

$$25 = 1 \cdot 20 + 5$$

$$20 = 4 \cdot 5$$

To postępowanie zawsze się kończy, bo pierwszy element w parze argumentów funkcji NWD maleje, gdyż jest to reszta.

Możemy teraz przedstawić opis algorytmu Euklidesa. Zauważmy jeszcze, że iloraz q w równości (8) nie występuje w następnych iteracjach tej równości, zatem nie trzeba go wyznaczać.

Algorytm Euklidesa (z dzieleniem) wyznaczania NWD

Dane: Dwie liczby naturalne m i n, .
Wynik: NWD(m,n) – największy wspólny dzielnik m i n.
Krok 1. Jeśli $m = 0$, to n jest szukanym dzielnikiem. Zakończ algorytm.
Krok 2. $r := n$ mod m, $n := m$, $m := r$. Wróć do kroku 1.

Następne ćwiczenie zwraca uwagę, że dla pewnych liczb algorytm Euklidesa wykonuje dużo iteracji.

Ćwiczenie 43. Zastosuj algorytm Euklidesa do liczb 34 i 55. Co ciekawego możesz powiedzieć o działaniu algorytmu w tym przypadku – ile wynoszą kolejne ilorazy? Wypisz od końca ciąg reszt tworzonych w tym przypadku. Jak inaczej można zdefiniować ten ciąg? Podaj inną parę liczb, dla której algorytm Euklidesa działa podobnie.

Zapiszemy teraz algorytm Euklidesa w postaci programu:

```
program Euklides;
 var m,n,r:integer;
begin
 read(m,n);
 while m>0 do begin
  r:=n mod m;
  n:=m;
  m:=r
 end;
 write(n)
end.
```

Zapiszemy teraz algorytm Euklidesa w postaci wydzielonej funkcji `NWD`, której wartością jest właśnie największy wspólny dzielnik dwóch liczb, będących argumentami tej funkcji.

```
program Euklides _ funkcja;
 var m,n:integer;

 function NWD(m,n:integer):integer;
  var r:integer;
 begin
  while m>0 do begin
   r:=n mod m; n:=m; m:=r
  end;
  NWD:=n
 end; {NWD}

begin
 read(m,n);
 writeln(NWD(m,n))
end.{Euklides _ funkcja}
```

Istnieje jeszcze wiele innych implementacji algorytmu Euklidesa, np. wykonujących odejmowanie zamiast dzielenia. Jedna z najciekawszych implementacji wykorzystuje **rekurencję**, czyli funkcję, która odwołuje się do siebie. Ta implementacja wynika wprost z zależności (9).

```
program Euklides _ rekurencja;
 var m,n:integer;
 function NWD _ rek(m,n:integer):integer;
 begin
  if m>n then NWD _ rek:=NWD _ rek(n,m)
  else if m = 0 then NWD _ rek:=n
       else NWD _ rek:=NWD _ rek(n mod m,m)
 end; {NWD _ rek}
begin
 read(m,n);
 writeln(NWD _ rek(m,n))
end.{Euklides _ rekurencja}
```

Ćwiczenie 44. Zapoznaj się z przedstawionymi implementacjami algorytmu Euklidesa, uruchom te programy i przetestuj je na wybranych parach liczb naturalnych, w tym m.in. na parze liczb 34 i 55.

8 ALGORYTMY ZACHŁANNE

Człowiek zawsze starał się upraszczać wykonywane przez siebie czynności, od budowania piramid, wychodzenia z labiryntu czy poruszania się po najkrótszych drogach do celu, aż po sterowanie maszynami, porządkowanie obiektów i informacji oraz pakowanie plecaka. Zwykle, pierwszym zamierzeniem w nowym działaniu jest osiągnięcie wyznaczonego celu w jakikolwiek sposób, a gdy już potrafimy coś robić, to zastanawiamy się, jak to można zrobić mniejszym wysiłkiem, szybciej, z największym zyskiem lub z najmniejszymi stratami. Jeśli nasze zadanie polega na osiągnięciu celu w kilku etapach, to dość często pomocna może być strategia, zgodnie z którą na każdym kroku staramy się wykonać możliwie najlepszy ruch, podjąć najlepszą decyzję. Postępujemy więc w sposób, który ma cechy **zachłanności,**

> **metoda zachłanna** jest stosowana do otrzymywania rozwiązań, które składają się z ciągu decyzji i na każdym kroku podejmowana jest możliwie najlepsza decyzja.

W tym rozdziale zajmiemy się problemami, w których celem jest otrzymanie możliwie najlepszego rozwiązania. Wspólną cechą przedstawionych rozwiązań będzie sposób ich otrzymywania, polegający na zastosowaniu podejścia zachłannego.

Wśród omówionych problemów będą jednocześnie przykłady, które posłużą do zilustrowania, że podejście zachłanne nie zawsze gwarantuje otrzymania najlepszego rozwiązania.

W następnych punktach omawiamy szczegółowo zastosowanie metody zachłannej do rozwiązywania kilku prostych problemów, a w ostatnim punkcie wymieniamy inne problemy, które są również rozwiązywane metodami zachłannymi.

8.1 PROBLEM WYDAWANIA RESZTY

Problem reszty polega na takim wydawaniu reszty, pozostałej po uiszczeniu zapłaty, aby klient otrzymywał jak najmniejszą liczbę banknotów i monet. Podobne życzenie możemy mieć w kasie oszczędności lub w banku, wybierając jakąś kwotę, a więc resztą może być jakakolwiek kwota pieniędzy, Zatem nasz problem polega na przedstawieniu danej kwoty pieniędzy w postaci jak najmniejszej liczby banknotów i monet.

Zanim zajmiemy się matematycznym i informatycznym rozwiązaniem tego problemu, dobrze jest podpatrzyć sprzedawców w sklepach, w jaki sposób wydają reszty klientom – często życie samo dostarcza nam rozwiązań.

> Problem reszty, podobnie jak każda w niej moneta, ma dwie strony: odbierający resztę chciałby dostać jak najmniej monet, a wydający – pozbyć się ich jak najwięcej. Obie tendencje mają swoje zachłanne realizacje. Możemy jednak podpowiedzieć sprzedającemu, jak mógłby postępować zgodnie z oczekiwaniami klientów – czyli wydawać resztę jak najmniejszą ilością monet – przy tym samemu mieć mniej do roboty. Sprzedawca miałby także mniej okazji, by się pomylić.

Dla uproszczenia rozważań banknoty będziemy również nazywali monetami i przyjmiemy, że wszystkie nominały monet (a więc również banknotów) są podane w groszach. Mamy więc następujące nominały na naszym rynku: 1 gr, 2 gr, 5 gr, 10 gr, 20 gr, 50 gr, 100 gr (1 zł), 200 gr (2 zł), 500 gr (5 zł), 1000 gr (10 zł), 2000 gr (20 zł), 5000 gr (50 zł), 10 000 gr (100 zł), 20 000 gr (200 zł). W dalszej części opuszczamy miano gr.

Każdą resztę można zawsze wydać, np. w postaci monet o nominałach 1[5], ale bardzo krzywimy się na sprzedawcę, gdy wydaje nam resztę samymi drobnymi monetami. Jak więc miałby on postępować, aby reszta była złożona z możliwie jak najmniejszej liczby monet? W tym miejscu przypomnij sobie, jak postępujesz, gdy masz zbyt wiele drobnych. Jeśli masz dwie monety o nominale 1, to starasz się zamienić je na jedną monetę o nominale 2. Jeśli jest ich pięć, to zamieniasz na monetę o nominale 5, a jeśli miałbyś dwadzieścia tysięcy monet jednogroszowych, to najlepiej byłoby je zamienić w banku na banknot dwustuzłotowy. Podobnie możesz postąpić z większą liczbą monet o innych nominałach.

[5] Zauważ, że gdyby nie było monety o nominale 1, to pewnych kwot nie bylibyśmy w stanie wydać. Podaj przykłady takich kwot. Chyba nie ma waluty na świecie, która nie zawierałaby monety o nominale 1. A może jest? Jeśli natknąłeś się na taką, to poinformuj o tym autora.

Zamiana większej liczby monet o małych nominałach na monetę o większym nominale podpowiada, jak mogłoby wyglądać postępowanie zachłanne, w którym od razu staramy się używać jak największych nominałów. Zresztą zapewne zaobserwowałeś taki sposób wydawania reszty u wielu sprzedawców.

Algorytm Reszta – Zachłanny sposób wydawania reszty
Dane: Nominały monet oraz reszta do wydania.
Wynik: Przedstawienie reszty w postaci najmniejszej liczby monet.
Krok iteracyjny. Dopóki reszta nie jest równa zero, odejmij od niej największy, mieszczący się w niej nominał, i wydaj odpowiednią monetę.

Ćwiczenie 45. Zastosuj zachłanny algorytm wydawania reszty do utworzenia kwot groszowych 63, 87 i 117 z możliwie najmniejszej liczby monet. Sprawdź na tych przykładach, czy czasem nie można utworzyć tych reszt z jeszcze mniejszej liczby monet.

Realizacja algorytmu Reszta w arkuszu kalkulacyjnym

Zanim zapiszemy algorytm wydawania reszty w języku programowania, utworzymy dla niego arkusz kalkulacyjny. Chcemy, abyś utworzył arkusz, który ma postać pokazaną na rys. 9. W kolumnie **A** są umieszczone nominały naszej waluty, a w komórce **D2** jest umieszczona kwota, którą mamy utworzyć z najmniejszej liczby banknotów i monet. Kwota ta jest redukowana w kolejnych wierszach o kwotę umieszczoną w kolumnie **C**, która została wydana w sposób zachłanny kolejnym co do wielkości nominałem banknotu lub monety.

Ćwiczenie 46. Utwórz arkusz, który umożliwi Ci obliczanie dla danej kwoty (zapisanej w komórkach **D2** i **D4**), najmniejszej liczby banknotów i monet, z jakich można ją złożyć. Najważniejszą decyzją, jaką musisz podjąć, jest wpisanie odpowiedniej formuły do komórek w kolumnie **B**. Oczywiście wystarczy, że wpiszesz formułę do komórki **B5**, a następnie ją skopiujesz przez przeciągnięcie do dołu. A zatem, jak obliczyć, ile banknotów 200 złotowych mieści się w kwocie, która jest wpisana do komórki **D4**?

Ćwiczenie 47. Uruchom utworzony arkusz dla kilku wybranych kwot, np. 17 gr, 29 gr, 63 gr, 29,29 zł, 1234,56 zł i innych. Sprawdzaj w polach **D2** i **C19**, czy otrzymujesz te same kwoty.

Realizacja algorytmu Reszta w języku programowania

Zamieszczamy poniżej kod programu w języku Pascal, który jest realizacją algorytmu zachłannego.

```
 1. Program Zachlanna_reszta;
 2.  var i,ile,kwota_int: integer;
 3.      kwota:           real;
 4.      nominal: array[1..14] of integer
 5.               =(20000,10000,5000,2000,1000,500,200,100,50,20,10,5,2,1);
 6.      reszta:  array[1..14] of integer;
 7. begin
 8.  read(kwota);
 9.  kwota_int:=round(kwota*100);
10.  for i:=1 to 14 do begin
11.   ile:=kwota_int div nominal[i];
12.   reszta[i]:=ile;
13.   kwota_int:=kwota_int-ile*nominal[i]
14.  end;
15.  for i:=1 to 14 do
16.   writeln(nominal[i],' gr:  ',reszta[i])
17. end.
```

	A	B	C	D
1				
2			Kwota do wydania	1 234,19 zł
3	Nominały	Liczba nominałów	Kwota	Pozostało
4				1 234,19 zł
5	200,00 zł	6	1 200,00 zł	34,19 zł
6	100,00 zł	0	- zł	34,19 zł
7	50,00 zł	0	- zł	34,19 zł
8	20,00 zł	1	20,00 zł	14,19 zł
9	10,00 zł	1	10,00 zł	4,19 zł
10	5,00 zł	0	- zł	4,19 zł
11	2,00 zł	2	4,00 zł	0,19 zł
12	1,00 zł	0	- zł	0,19 zł
13	0,50 zł	0	- zł	0,19 zł
14	0,20 zł	0	- zł	0,19 zł
15	0,10 zł	1	0,10 zł	0,09 zł
16	0,05 zł	1	0,05 zł	0,04 zł
17	0,02 zł	2	0,04 zł	- zł
18	0,01 zł	0	- zł	- zł
19		Razem	1 234,19 zł	

Rysunek 9.
Arkusz służący do wydawania reszty metodą zachłanną

Znaczenie poszczególnych wierszy kodu powinno być oczywiste nie tylko dla tych, którzy napisali już jakiś program w języku Pascal. Wyjaśnijmy jednak znaczenie wybranych wierszy w programie.

1. wiersze 4 i 5: `nominal` – jest tablicą nominałów zamienionych na grosze;
2. wiersz 6: w tablicy `reszta` są przechowywane wyliczone ilości poszczególnych nominałów;
3. wiersz 8: czytana jest `kwota` do wydania; zakładamy, że kwota jest w złotych, czyli ta dana może zawierać kropkę i dwie cyfry po kropce, oznaczające liczbę groszy w kwocie; zatem 13 oznacza 13 zł, a chcąc utworzyć resztę dla kwoty 13 gr musimy podać 0,13 jako daną;
4. wiersz 9: `kwota` w złotych jest zamieniana na `kwota _ int` w groszach;
5. wiersze 10-14: **instrukcja iteracyjna** – obliczenie dla kolejnych nominałów, ile razy mieszczą się w kwocie, która nie została jeszcze wydana – stosowane jest dzielenie całkowite `div` (jego wynikiem jest część całkowita ilorazu);
6. wiersze 15-16: ponownie jest użyta instrukcja iteracyjna, która tym razem służy do wypisywanie na ekranie w dwóch kolumnach nominałów i ich ilości, składających się na wczytaną kwotę.

Ćwiczenie 48. Napisz i uruchom samodzielnie program do wydawania reszty metodą zachłanną. Możesz się wzorować na naszym rozwiązaniu podanym powyżej. Wykonaj swój program dla kwot wymienionych w ćwicz. 47. Porównaj wyniki otrzymane w arkuszu i otrzymane za pomocą swojego programu.

Ćwiczenie 49. Posługując się swoim programem spróbuj obliczyć, jak zostanie wydana kwota 1234.56. Jak zakończyło się wykonywanie Twojego programu? Czy potrafisz wytłumaczyć, co się stało? Otóż podana kwota zamieniona na grosze jest większa od największej liczby typu `integer` (czyli największej dodatniej liczby całkowitej), jaka może być zapisana w komputerze – jest nią 32767. Można pozbyć się tego ograniczenia na wielkość kwoty deklarując zmienną `kwota _ int` jako `longint`, czyli jako długą liczbę całkowitą. Wtedy będzie ona mogła przyjmować wartości do ponad 21 milionów (niewiele jest okazji, by wydawać tak duże reszty!). Zmodyfikuj odpowiednio swój program.

Ciekawi nas teraz, czy wydawanie reszt algorytmem zachłannym zawsze gwarantuje, że otrzymamy najmniejszą liczbę banknotów i monet. Odpowiedź na to pytanie nie jest jednoznaczna i zależy od rodzajów nominałów, którymi dysponujemy.

Ćwiczenie 50. Przypuśćmy, że Mennica Polska wypuściła na rynek dodatkową monetę dla hazardzistów o nominale 21 groszy. Podaj przykłady reszt (kwot), dla których algorytm zachłanny wydawania reszty za pomocą polskich monet, powiększonych o ten nominał nie zapewnia, że każdą resztę otrzymamy w postaci najmniejszej liczby monet?

Dotychczas zakładaliśmy milcząco, że w kasie jest dostateczna liczba monet każdego nominału. Okazuje się jednak, że jeśli brakuje niektórych monet, to w algorytmie zachłannym mogą nie być tworzone reszty złożone z najmniejszej liczby monet. Rozwiąż następne ćwiczenie.

Ćwiczenie 51. Przypuśćmy, że w kasie zabrakło nagle monet o nominałach 10 i 5 groszy. Znajdź przykłady kwot, które w tym przypadku nie będą utworzone przez algorytm zachłanny z najmniejszej możliwej liczby banknotów i monet.

Powyższe przykłady pokazują, że to, czy zestawy monet tworzone przez algorytm zachłanny zawierają najmniejszą ich liczbę, zależy od dostępności nominałów, czyli zależy od danych, a zatem stosowana metoda nie daje najlepszych rozwiązań we wszystkich przypadkach. To jest niestety charakterystyczna cecha wielu algorytmów zachłannych.

Ćwiczenie 52. Monety i banknoty amerykańskie mają nominały w centach: 1 (*penny*), 5 (*nickel*), 10 (*dime*), 25 (*quarter*), 50 i w dolarach: 1, 2, 5, 10, 20, 50, 100, 200, 1000.

- Zmodyfikuj swój program do wydawania reszty tak, aby z jego pomocą można było tworzyć reszty dla kwot w dolarach i centach.
- Przypuśćmy, że w kasie brakło nagle pięciocentówek. Podaj przykład reszty, której algorytm zachłanny nie utworzy w tym przypadku z najmniejszej liczby banknotów i monet.

8.2 ZMARTWIENIE KINOMANA

Kinoman dysponuje repertuarem filmów w Multikinie na dany dzień – z każdym filmem jest związana godzina jego rozpoczęcia i zakończenia. Zakładamy, że filmy są różne i kinoman chciałby obejrzeć ich możliwie jak najwięcej. W tabeli 5 są zamieszczone przykładowe godziny wyświetlania filmów. Pytanie: ile z tych filmów może obejrzeć kinoman w całości jednego dnia?

Tabela 5.
Przykładowe godziny rozpoczęcia i zakończenia filmów

	1	2	3	4	5	6	7	8	9	10	11
początek	9:00	8:00	11:00	10:00	11:00	12:00	14:00	16:00	15:00	18:00	19.00
koniec	11:00	12:00	13:00	12:00	15:00	16:00	17:00	18:00	19:00	21:00	21.00

Ćwiczenie 53. Jaką podpowiedź dasz kinomanowi? Podpowiadamy Tobie, byś zastosował metodę zachłanną, ale sam określ, na czym ma polegać zachłanność w tym przypadku. Sprawdź swój algorytm na danych z tabeli 5.

Dość oczywistym podejściem jest wybieranie kolejnych filmów, które kończą się najwcześniej. W ten sposób kinomanowi pozostaje więcej czasu na obejrzenie następnych filmów.

Uzasadnienie, że jest to **strategia optymalna**, czyli możliwie najlepsza, jest dość proste. Zastosujemy rozumowanie nie wprost. Załóżmy, że istnieje inne rozwiązanie, które jest optymalne, czyli zawiera ono więcej filmów niż rozwiązanie otrzymane zasugerowaną metodą zachłanną. Przeglądamy oba rozwiązania w kolejności filmów od najwcześniejszego i zatrzymujemy się, gdy w rozwiązaniu optymalnym jest inny film niż w rozwiązaniu zachłannym. Zgodnie ze strategia zachłanną, film w rozwiązaniu optymalnym kończy się nie wcześniej niż film w rozwiązaniu zachłannym. Możemy zatem zamienić film w rozwiązaniu optymalnym z filmem w rozwiązaniu zachłannym. Przechodząc w ten sposób do końca obu rozwiązań dochodzimy do wniosku, że rozwiązanie zachłanne zawiera przynajmniej tyle filmów co optymalne, a więc jest także rozwiązaniem optymalnym.

Ćwiczenie 54. Napisz program, który dla danego repertuaru Multikina znajduje opisaną wyżej metodą zachłanną największą liczbę filmów, jakie można obejrzeć jednego dnia. Czasy rozpoczęcia i zakończenia filmów przechowuj w tablicach.

8.3 PAKOWANIE NAJCENNIEJSZEGO PLECAKA

W tym punkcie zajmiemy się rozwiązywaniem jednej z wersji **problemu plecakowego**. Problem ten polega na umieszczeniu w plecaku o ograniczonej pojemności możliwie najcenniejszej zawartości. W innych zastosowaniach ten problem może dotyczyć pakowania: walizek, paczek, samochodów, samolotów itp. Zakładamy, że pakowane rzeczy są niepodzielne, tzn. nie można wziąć tylko połowy jakiejś rzeczy. Na początku załóżmy, że każda rzecz jest dostępna w nieograniczonej ilości. Opiszmy dokładniej ten problem, podając jego specyfikację.

Ogólny problem plecakowy

Dane: *n* rzeczy (towarów, produktów itp.), każda w nieograniczonej ilości:
i-ta rzecz waży w_i jednostek i ma wartość p_i;
W – maksymalna pojemność plecaka.

Wynik: ilości poszczególnych rzeczy (mogą być zerowe), których całkowita waga nie przekracza *W* i których sumaryczna wartość jest największa wśród wypełnień plecaka rzeczami o wadze nie przekraczającej *W*.

Problem plecakowy jest tylko z pozoru bardzo prosty. Opracowano dla jego rozwiązywania wiele algorytmów o różnej złożoności. Nie jest jednak znana metoda szybka, która jednocześnie gwarantuje, że zawsze jest generowane możliwie najlepsze upakowanie plecaka. W takiej sytuacji na ogół staramy się rozwiązywać problem metodą zachłanną.

Algorytm zachłanny dla ogólnego problemu plecakowego

Przypomnijmy sobie, w jaki sposób na ogół pakujemy plecak. Dobrze, jeśli wszystkie zaplanowane do wzięcia rzeczy mieszczą się w nim. Ale jeśli nie, to najczęściej działamy dość chaotycznie i w naszych decyzjach ścierają się ze sobą trzy **kryteria wyboru** kolejnych rzeczy do zapakowania:

1. wybierać najcenniejsze rzeczy, czyli w kolejności nierosnących wartości p_i;
2. wybierać rzeczy zajmujące najmniej miejsca, czyli w kolejności niemalejących wag w_i;
3. wybierać rzeczy najcenniejsze w stosunku do swojej wagi, czyli w kolejności nierosnących wartości ilorazu p_i / w_i – iloraz ten można uznać za jednostkową wartość rzeczy *i*.

Wszystkie te kryteria wyboru są przejawem strategii zachłannej. Zawartość plecaka jest kompletowana krok po kroku i każda decyzja polega na dokonaniu wyboru, który wydaje się być najlepszy na danym etapie z nadzieją, że ostatecznie doprowadzi to do znalezienia najlepszego upakowania.

Tabela 6.
Dane do przykładu ogólnego problemu plecakowego

i	1	2	3	4	5	6	*W*
p_i	6	4	5	7	10	2	
w_i	6	2	3	2	3	1	23

Ćwiczenie 55. Dla danych zamieszczonych w tabeli 6 wyznacz najcenniejsze zawartości plecaka, stosując każde z powyższych kryteriów wyboru kolejnej rzeczy.

Pakując rzeczy w kolejności ich wartości, czyli w kolejności wartości współczynników p_i, najpierw wybieramy najcenniejszą rzecz, nr 5, i możemy wziąć ich aż 7, gdyż każda z nich waży 3 jednostki a pojemność plecaka wynosi 23. Pozostaje w plecaku miejsce na rzecz, która waży 2 jednostki. Następną w kolejności wartości jest rzecz nr 4 i waży ona akurat 2 jednostki, pakujemy ją. Zatem pakując plecak w kolejności największych wartości rzeczy, wybieramy 7 sztuk rzeczy nr 5 i jedną rzecz nr 4 – otrzymujemy zawartość plecaka o wartości 77.

Pakując rzeczy w kolejności wag, możemy umieścić 23 sztuki rzeczy najlżejszej, nr 6 – ta zawartość plecaka ma wartość tylko 46.

Pakujemy teraz plecak w kolejności nierosnących proporcji wartości do wagi. W naszym przykładzie, nierosnącej kolejności ilorazów (7/2, 10/3, 4/2, 2/1, 5/3, 6/6) odpowiada następująca kolejność rzeczy (4, 5, 2, 6, 3, 1). Zatem, najpierw umieszczamy w plecaku 11 sztuk rzeczy nr 4, z których każda waży 2 jednostki. Pozostałą jednostkę pojemności plecaka zapełniamy rzeczą nr 6. Otrzymujemy zawartość plecaka o wartości 79, najcenniejszą wśród zachłannie pakowanych.

Ćwiczenie 56. Nie powinieneś mieć większego kłopotu z zapakowaniem do plecaka rzeczy o łącznej wartości 80 – jak?

Wykazaliśmy na tym przykładzie – co uświadamia nam ćwicz. 56 – że żadna z trzech powyższych strategii zachłannych może nie gwarantować znalezienia najlepszego wypełnienia plecaka. Otrzymane rozwiązania są **przybliżone** w stosunku do rozwiązania optymalnego (czyli najlepszego). Najbardziej uzasadnione jednak wydaje się być trzecie kryterium, gdyż są w nim uwzględnione oba parametry opisujące każdą rzecz, wartość i waga.

Implementacja, czyli komputerowa realizacja zachłannej metody pakowania plecaka jest bardzo prosta. Zakładamy, że rzeczy są uporządkowane zgodnie z jednym z przyjętych kryteriów kolejności wybierania rzeczy do plecaka. Dane są umieszczone w tablicach następującego typu:

```
type Tablica1n  =array[1..Maxn] of integer;
```

gdzie `Maxn` jest stałą oznaczającą maksymalną liczbę różnych rodzajów rzeczy. Oznaczmy przez `Waga` pojemność plecaka *W*, a przez `Plecak` wartość całkowitej zawartości plecaka. Wtedy ilości poszczególnych rzeczy, których jest `n`, oznaczone przez `q[i]`, są obliczane w następujący sposób:

```
Plecak:=0;
for i:=1 to n do begin
 q[i]:=Waga div w[i];
 Waga:=Waga-q[i]*w[i];
 Plecak:=Plecak+p[i]*q[i]
end
```

Ćwiczenie 57. Uzupełnij powyższy fragment programu do pełnego programu, służącego do wyznaczania zachłannego rozwiązania ogólnego problemu plecakowego. Sprawdź działanie swojego programu na przykładzie z tabeli 6 i porównaj wyniki z otrzymanymi w rachunkach bez pomocy komputera. Za kolejność rzeczy wybieraj kolejności odpowiadające wszystkim trzem kryteriom.

W powyższej implementacji zakładamy, że rzeczy są rozpatrywane w kolejności określonej wcześniej, a więc są już uporządkowane zgodnie z pewnym kryterium. Jeśli znasz jakiś algorytm porządkowania (sortowania) i potrafisz go zaprogramować, to wykonaj następne ćwiczenie.

Ćwiczenie 58. Uzupełnij program otrzymany w ćwicz. 57 o porządkowanie rzeczy zgodnie z wybranym kryterium.

Decyzyjna wersja problemu plecakowego

Omawiana wyżej wersja problemu plecakowego jest określana jako **ogólna**, gdyż założyliśmy, że dysponujemy nieograniczoną ilością każdej z rzeczy. Często jednak pakując plecak mamy tylko po jednej sztuce każdej rzeczy, wtedy nasz wybór polega jedynie na podjęciu decyzji: wziąć tę rzecz, czy jej nie brać. Taka wersja tego problemu nazywa się **decyzyjnym problemem plecakowym.** Mimo wprowadzonego do wersji ogólnej ograniczenia na liczbę dostępnych egzemplarzy poszczególnych rzeczy, problem decyzyjny jest nadal dość ogólny, gdyż, jeśli jakaś rzecz występuje w większej ilości, to możemy każdy jej egzemplarz nazwać inaczej i wtedy każda rzecz (egzemplarz) będzie występować pojedynczo.

Ćwiczenie 59. Znajdź rozwiązania zachłanne dla decyzyjnego problemu plecakowego, dla którego dane znajdują się w tabeli 6. Czy jest wśród nich rozwiązanie optymalne, czyli możliwie najlepsze? Uzasadnij swoją odpowiedź.

Ćwiczenie 60. Zmodyfikuj program otrzymany w ćwicz. 57 tak, aby rozwiązywał decyzyjny problem plecakowy. Podobnie, w tym celu zmodyfikuj program otrzymany w ćwicz. 58, jeśli je rozwiązałeś.

8.4 NAJDŁUŻSZA DROGA NA PIRAMIDZIE

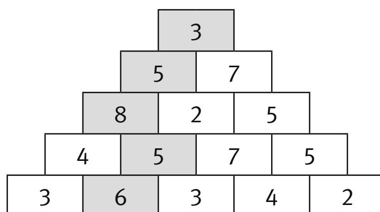
Kolejny problem ma charakter łamigłówki liczbowej. Polega on na znalezieniu takiej drogi przejścia z wierzchołka piramidy – patrz rys. 10 – do punktu w podstawie piramidy, aby suma elementów, przez które prowadzi droga, była jak największa. Droga z korzenia do podstawy powinna zawierać dokładnie jeden element z każdego poziomu i każde dwa elementy z sąsiednich poziomów powinny ze sobą sąsiadować – taką drogę zaznaczono na rys. 10 ciemnym kolorem.

Rozwiązanie zachłanne samo się nasuwa: każdy element sąsiaduje z dwoma na niższym poziomie, a zatem, by uzyskać drogę o możliwie największej sumie, spośród tych dwóch elementów wybieramy ten, w którym jest większa liczba. Tylko w pierwszym kroku nie mamy wyboru – zaczynamy bowiem od czubka piramidy.

Ćwiczenie 61. Znajdź rozwiązanie zachłanne dla piramidy pokazanej na rys. 10. Czy jest to droga o największej sumie elementów? Sprawdź inne drogi.

A zatem także w przypadku tego problemu, algorytm zachłanny nie znajduje najlepszego z możliwych rozwiązania.

Ćwiczenie 62. Napisz program, który będzie służył do znajdowania zachłannego rozwiązania problemu najdłuższej drogi na piramidzie. Przyjmij, że piramida ma *n* wierszy (*i*-ty wiersz składa się z *i* elementów). Piramidę przechowuj w tablicy dwuwymiarowej tak, jakby jej poziomy były przesunięte do lewej, czyli *i*-ty wiersz tablicy będzie zawierał *i* elementów. Odpowiedz najpierw, które elementy w wierszu (*i*+1)-wszym będą sąsiednie dla *k*-tego elementu w wierszu *i*-tym.



Rysunek 10.
Piramida z wagami

Teraz mamy nieco trudniejsze zadanie.

Ćwiczenie 63. Drogę na piramidzie o największej sumie można znaleźć zaczynając jej szukać nie od wierzchołka piramidy, ale od podstawy piramidy. Postaraj się zaproponować taką metodę. Podpowiemy Ci tylko, że w tej metodzie są rozważane jednocześnie wszystkie drogi kończące się na ostatnim poziomie, następnie na poziomie przedostatnim itd. aż do wierzchołka piramidy. Jeśli już znajdziesz taki algorytm, to napisz dla niego program, korzystając ze wskazówek umieszczonych w ćwicz. 62.

8.5 INNE PRZYKŁADY UŻYCIA METODY ZACHŁANNEJ

Dobór w trwałe pary
Osoby z dwóch równolicznych grup mają połączyć się w pary, które nie powinny rozpaść się zbyt szybko (np. taneczne lub małżeńskie). Każda z osób ma skrystalizowane preferencje wobec wszystkich osób w drugiej grupie. Strategia zachłanna w tym przypadku oznacza, że kolejno każda osoba z jednej grupy dobiera sobie z drugiej grupy najbardziej preferowanego partnera. Takie postępowanie może zakończyć się skompletowaniem wszystkich par tylko wtedy, gdy różne są największe preferencje wszystkich osób, czyli gdy nie ma dwóch osób, które umieściły na pierwszym miejscu tę samą osobę. Na ogół w każdej grupie są osoby bardziej lubiane niż inne. Jak więc postępować, nie rezygnując jednak zbytnio ze swoich największych preferencji? Wystarczy nieco zmodyfikować tę prostą strategię zachłanną – jeśli nasz najlepszy wybór zostaje odrzucony przez wybranego przez nas partnera (gdyż otrzymał propozycję od osoby, którą bardziej preferuje), to wybieramy następną osobę w kolejności naszych preferencji. Okazuje się, że takie postępowanie prowadzi do układu trwałych par (w pewnym sensie). Problem ten jest szczegółowo opisany w rozdz. 11 w książce [6].

Poszukiwanie wyjścia z labiryntu
Jest to sytuacja chyba najbardziej podatna na działanie zachłanne — znajdujemy się w jakimś zakamarku mrocznego labiryntu i o niczym innym nie marzymy, jak tylko o wydostaniu się z niego jak najprędzej (patrz p. 11.1 w książce [5]). Pierwsza metoda, jaka przychodzi nam do głowy w takiej sytuacji, to, trzymając rękę na ścianie, próbować „wymacać" drogę do wyjścia. Inna metoda, która jest chyba najbardziej „zachłanna" w tym przypadku, polega „na zgłębianiu" labiryntu, czyli na przechodzeniu jak najdalej i jak najgłębiej, jak to tylko możliwe. Dzięki uwzględnieniu w metodzie zgłębiania możliwości cofania się po zabrnięciu w ślepą uliczkę, ta strategia zachłanna zawsze prowadzi do znalezienia wyjścia z labiryntu, gdy tylko ono istnieje. Tę metodę omawiamy w rozdziale dotyczącym przeszukiwania z nawrotami – patrz p. 9.1. Nie zawsze jednak zgłębianie możliwych korytarzy w labiryncie wiedzie nas najkrótszą drogą do wyjścia z niego. Potrzebny jest tutaj inny

rodzaj zachłanności – taka metoda jest związana ze znajdowaniem najkrótszych dróg w grafach (takimi problemami zajmujemy się na zajęciach poświęconych grafom).

Najszybsze pomieszanie kolorowych kartek

Mamy kilka stosów z kolorowymi kartkami i chcemy utworzyć jeden stos, w którym kolory z poszczególnych stosów będą wymieszane. Można to osiągnąć za pomocą następującego algorytmu:

1. wybieramy dwa stosy kartek;
2. **scalamy** wybrane stosy kartek w jeden stos wybierając na przemian po jednej kartce z jednego i z drugiego stosu;
3. tak powstały stos kartek kładziemy obok innych stosów i powtarzamy kroki 1 i 2 tak długo, aż pozostanie tylko jeden stos.

W jakiej kolejności należy scalać stosy kartek, by w całym algorytmie możliwie najmniej się napracować przy przekładaniu kartek ze stosów na nowy stos? Zachłanność podpowiada nam, by na każdym kroku najmniej przekładać kartek, a więc wybierać do scalania zawsze dwa najniższe stosy kartek. I okazuje się, że tą metodą otrzymuje się możliwie najlepsze rozwiązanie, patrz rozdz. 13 w książce [6].

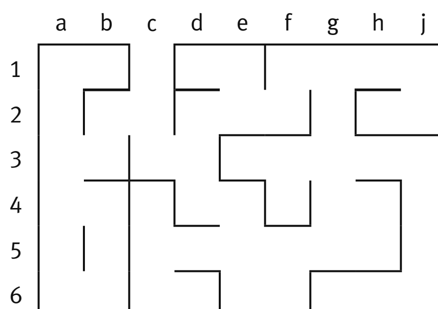
9 PRZESZUKIWANIE Z NAWROTAMI

Istnieje wiele sytuacji, w których, by odpowiedzieć na postawione pytanie lub znaleźć potrzebne rozwiązanie problemu, musimy przeszukać niemal cały zbiór wszystkich możliwych rozwiązań lub dużą jego część. W tym rozdziale zajmiemy się dwoma problemami, dla których nie potrafimy znaleźć rozwiązania w inny sposób, niż przeszukując dużą część możliwych rozwiązań. O jednym z tych problemów wspomnieliśmy już w poprzednim rozdziale – chodzi o znalezienie wyjścia z labiryntu, a drugi problem dotyczy rozmieszczenia figur na szachownicy.

Charakterystyczną cechą prezentowanych w tym rozdziale metod – znanych jako **przeszukiwanie z nawrotami** – jest sposób, w jaki z ich pomocą poszukuje się rozwiązania. Rozwiązanie jest rozbudowywane w kolejnych krokach, a jeśli w danym kroku nie może być powiększone, to następuje wycofanie (nawrót) do poprzedniego kroku i podejmowana jest próba znalezienia innej możliwości w tym kroku. Ten sposób uporządkowanego przeglądania wszystkich możliwych rozwiązań jest gwarancją, że żadne rozwiązanie nie zostanie pominięte w rozważaniach i jeśli istnieje interesujące nas rozwiązanie (np. wyjście z labiryntu), to będzie ono znalezione.

9.1 WYJŚCIE Z LABIRYNTU METODĄ ZGŁĘBIANIA

Zakładamy, że **labirynt** jest zamknięty w prostokącie, ma tylko jedno wyjście (wejście) i wszystkie jego ściany wewnętrzne są równoległe do zewnętrznych. Dla uproszczenia rozważań załóżmy również, że ściany są fragmentami siatki kwadratowej. Zatem wnętrze labiryntu jest złożone z kwadratowych pól tej siatki i każdy wewnętrzny punkt labiryntu możemy utożsamiać z polem, w którym leży. Ponadto przyjmujemy, że w labiryncie nie ma zamkniętych komnat, a więc z każdego punktu wewnętrznego istnieje droga prowadząca do wyjścia. Przykładowy labirynt jest pokazany na rys.11.



Rysunek11.
Przykładowy labirynt

Naszym celem jest podanie algorytmu, który z każdego pola labiryntu zaprowadzi nas do jego wyjścia. Metoda wychodzenia z labiryntu jest na ogół opisem sposobu chodzenia po istniejących odcinkach korytarzy (utworzonych w naszym przypadku z pól) i można w niej wyróżnić dwa elementy:

- regułę gwarantującą, że żaden odcinek drogi w labiryncie nie przechodzimy więcej niż jeden raz;
- strategię jak najszybszego znalezienia wyjścia z labiryntu.

W punkcie 2.5 wspomnieliśmy o wychodzeniu z labiryntu metodą „z ręką na ścianie", która, jest może intuicyjna i w pewnym sensie zachłanna, nie ma jednak zbyt ciekawych własności. W tym punkcie opiszemy metodę „zgłębiania" labiryntu, która ma cechę postępowania zachłannego i dodatkowo zawiera mechanizm **powrotów**, który gwarantuje, że nawet w przypadku chwilowych niepowodzeń, zawsze dojdziemy do wyjścia z labiryntu. Niestety, nie zawsze najkrótszą drogą – sposobu opuszczania labiryntu najkrótszą drogą nie będziemy jednak tutaj omawiać.

W każdym polu labiryntu – przedstawionego tak, jak na rys.11 – są co najwyżej cztery możliwości wykonania następnego ruchu: w górę, w lewo, w prawo, w dół – oznaczmy te kierunki przez (G, L, P, D). Możemy więc zaproponować następujący algorytm poruszania się po naszymlabiryncie:

Algorytm. Zgłębianie labiryntu

Stosuj następujące zasady poruszania się po labiryncie, poczynając od pola, z którego chcesz trafić do wyjścia:

Krok 1. W polu, w którym się znalazłeś, wybierz z listy (G, L, P, D) pierwszy kierunek, który jeszcze nie był badany i w tym kierunku jest przejście z pola, w którym się znajdujesz, na pole, które nie jest oddzielone ścianą i jeszcze dotychczas na nim nie byłeś – przejdź na to pole i kontynuuj ten krok;

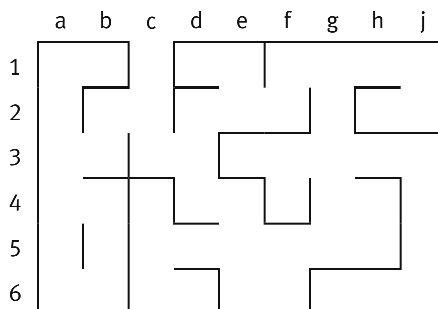
Krok 2. Jeśli z danego pola nie można już przejść w żadnym kierunku, to wróć do pola, z którego przyszedłeś, i przejdź do kroku 1.

Ruch, będący powrotem, będziemy oznaczali literą B. Każde przejście możemy opisać nazwą ruchu (kierunku) i nazwą pola, na które nas wiedzie. Zróbmy jeszcze jedno, dość naturalne założenie: kierunek poruszania się po labiryncie określamy w zależności od naszego ustawienia i przyjmujemy przy tym, że cały czas poruszamy się twarzą do przodu, z wyjątkiem ruchów typu B. Stąd wynika, że kierunek G jest zawsze przed nami.

Zastosujmy powyższy algorytm do znalezienia wyjścia z pola 3b w labiryncie na rys. 11. Pierwszy ruch ma postać G-2b – poruszamy się w górę na pole 2b – ale w następnym kroku nie możemy już iść ani do góry ani w lewo, więc wykonujemy ruch P-2c, a z pola 2c — ruch L-1c i już jesteśmy przy wyjściu z labiryntu.

Jeśli szukamy wyjścia z pola 4a, to początkowy fragment drogi ma postać: G-3a, G-2a, G-1a. Z pola 1a nie możemy iść ani w kierunku G, ani L, wykonujemy więc P-1b. Z pola 1b nie możemy już przejść do żadnego nowego pola, wracamy więc, i to aż do pola 3a: B-1a, B-2a, B-3a. Z pola 3a możemy teraz przejść w prawo, a więc wykonujemy kolejne ruchy: P-3b, L-2b, P-2c, L-1c. Na rys. 12 zaznaczono wykonane ruchy w tym przypadku.

Jeśli poszukiwanie wyjścia rozpoczynamy w punkcie 2a, to początkowe ruchy są podobne: G-1a, P-1b, B-1a, B-2a i wracamy do punktu wyjścia. Możliwy do wykonania pozostał jeszcze ruch do dołu: D-3a. Znaleźliśmy się w punkcie, z którego poprzednie poszukiwanie wyjścia zakończyło się dość szybko. Teraz jednak nasz kierunek poruszania się jest do dołu (zgodnie z zasadą „twarzą do przodu"), więc następnymi ruchami są: G-4a, G-5a, G-6a, L-6b, L-5b, G-4b. Z pola 4b nie możemy jednak przejść na pole 4a, aby się nie zapętlić, gdyż przechodziliśmy już przez nie w tym poszukiwaniu wyjścia, zatem wracamy: B-5b, B-6b, B-6a, B-5a, B-4a, B-3a, a dalej już do wyjścia: L-3b, L-2b, P-2c, L-1c.



Rysunek 12.

Poszukiwanie wyjścia z labiryntu przedstawionego na rys. 11. zaczynając w polu 4a. Kolejno wykonywane ruchy i odwiedzane pola: G-3a, G-2a, G-1a, P-1b, B-1a, B-2a, B-3a, P-3b, L-2b, P-2c, L-1c

Ćwiczenie 64. Posługując się algorytmem **Zgłębianie labiryntu** wypisz kolejno odwiedzane pola na drodze do wyjścia z pola 4g w labiryncie pokazanym na rys. 11. Jest to jednak dość długa droga – aby ją znaleźć, musisz przejść przez 26 pól, przez niektóre po kilka razy.

Przedstawiony algorytm jest realizacją metody przeszukiwania przestrzeni możliwych rozwiązań, która nazywa się **przeszukiwaniem w głąb**, gdyż w kolejnych krokach przeszukiwanie zagłębia się coraz bardziej, tak daleko, jak to możliwe. Ten algorytm należy również do rodziny metod nazywanych **przeszukiwaniem z nawrotami**, gdyż są w nim wykonywane nawroty do poprzednich punktów, gdy poszukiwanie zabrnie w ślepy zaułek i nie można kontynuować go do przodu.

Zauważmy, jak proste jest sformułowanie algorytmu **Zgłębianie labiryntu**. Tę prostotę zawdzięczamy przede wszystkim użyciu w jego opisie **rekurencji** – w obu krokach 1 i 2 następuje rekurencyjne odwołanie do kroku 1 z nowego pola w labiryncie, w którym znalazł się proces poszukiwania.

Zgodnie z naszym założeniem, w labiryncie nie ma obszarów zamkniętych, a więc z każdego pola istnieje droga do wyjścia. Na tej podstawie nie powinieneś mieć kłopotu z wykonaniem następnego ćwiczenia.

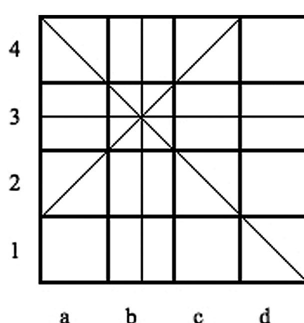
Ćwiczenie 65. Uzasadnij, że z każdego pola labiryntu algorytm z nawrotami **Zgłębianie labiryntu** znajdzie drogę z tego pola do wyjścia z labiryntu.

Algorytm przeszukiwania z nawrotami ma bardzo intuicyjny opis w języku grafów. W przypadku problemu poszukiwania wyjścia z labiryntu, konstruujemy dla labiryntu odpowiedni graf – jego wierzchołkami są pola a krawędziami możliwe przejścia między nimi – i przeszukujemy go metodą w głąb. Taka implementacja algorytmu **Zgłębiania labiryntu** zostanie przedstawiona na innych zajęciach, poświęconych grafom i ich algorytmom.

i+

9.2 ROZMIESZCZANIE HETMANÓW NA SZACHOWNICY

Zakładamy, że wiesz jak wygląda szachownica. Jeśli natomiast nie umiesz grać w szachy, to przyjmij, że hetman jest figurą, która atakuje figury stojące na wszystkich liniach przechodzących przez pole, na którym stoi – poziomej, pionowej i obu przekątnych, tak jak na tym rys. 13 – hetman stoi na polu b3:



Rysunek 13.
Pola atakowane przez hetmana, stojącego na polu b3

Problem hetmanów polega na rozmieszczeniu na szachownicy jak największej liczby hetmanów, z których żadne dwa nie atakują się. Zastosujemy metodę przeglądania możliwych ustawień hetmanów, ale oczywiście nie wszystkich ustawień, bo jest ich bardzo dużo[6]. Ponieważ, w każdym wierszu i w każdej kolumnie szachownicy może się znajdować co najwyżej jeden hetman, na szachownicy o wymiarach $n \times n$ można umieścić

[6] Uzasadnij, że na szachownicy 8x8 można ustawić osiem hetmanów na 8^8 sposobów.

co najwyżej *n* hetmanów. z których żadne dwa nie atakują się. Postaramy się umieścić ich dokładnie *n*. Dla uproszczenia, nasze rozważania będziemy ilustrować szachownicą 4×4, ale program, jaki napiszemy, będzie mógł być użyty dla dowolnej szachownicy.

Ćwiczenie 66. Przygotuj sobie planszę szachownicy o wymiarach 4×4 oraz cztery jednakowe figury – mogą to być nawet pionki, które będziesz traktował jak hetmany. Stosując metodę prób i błędów, spróbuj umieścić te figury na szachownicy, jako czterech nieatakujące się hetmany.

Przypuszczamy, że udało Ci się wykonać to zadanie. Czy posłużyłeś się jakąś szczególną metodą? Postaraj się ją opisać.

Podamy teraz metodę **poszukiwania z nawrotami**, która w systematyczny sposób sprawdza wszystkie możliwe ustawienia nieatakujących się hetmanów, by ustawić na szachownicy maksymalną ich liczbę. Te poszukiwania będą prowadzone na tyle oszczędnie, że w sytuacji, gdy danej konfiguracji nie można powiększyć o kolejnego hetmana, następuje wycofanie się z niej i ponawiana jest próba utworzenia nowej konfiguracji w poprzednim kroku, a potem – jej powiększenia. Odpowiednich pozycji na szachownicy dla hetmanów będziemy poszukiwali kolumnami, począwszy od kolumny a, a w kolumnach poszukiwania będą przebiegały od góry, czyli od ostatniego wiersza.

Zanim sformułujemy podstawową regułę, na której jest oparte przeszukiwanie z nawrotami, prześledźmy początkowe próby ustawienia 4 hetmanów na szachownicy 4x4. Zgodnie z przyjętym założenie o kierunku poszukiwań, w kolejnych ruchach będziemy się posuwali kolumnami, począwszy od kolumny a, i od góry, czyli od wiersza 4.

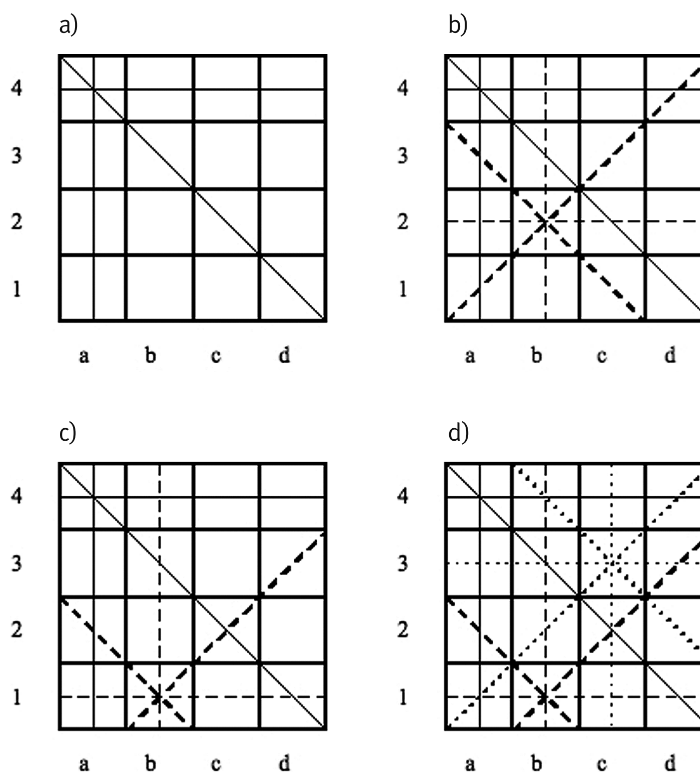
Na rysunkach 14 i 15 są przedstawione kolejne ustawienia nieatakujących się hetmanów. Wraz z ustawieniem hetmanów, jednocześnie są zaznaczone linie ich ataku na inne pola – pola nieprzekreślone są tymi, na których może jeszcze stanąć nowy hetman. Rozpoczynamy w polu a4 (rys. 14a). W drugiej kolumnie pozostają wolne pola b2 i b1. Ustawiamy więc hetmana najpierw na polu b2 (rys. 14b) i przechodzimy do trzeciej kolumny. Niestety, wszystkie pola w trzeciej kolumnie są już atakowane – wracamy więc z trzeciej kolumny do drugiej i stawiamy hetmana na następnym polu w tej kolumnie, czyli na b1 (rys. 14c). Tym razem w trzeciej kolumnie jest wolne pole c3 – ustawiamy więc na nim trzeciego hetmana i otrzymujemy konfigurację pokazana na rys. 14d. Tym razem, nie ma już wolnego pola dla hetmana w czwartej kolumnie. Czy to oznacza, że nie można ustawić cztery nieatakujące się hetmany na szachownicy o wymiarach 4×4?

Otóż dotychczas wykorzystaliśmy jedynie pierwszą możliwość ustawienia hetmana w pierwszej kolumnie – na polu a4, i sprawdziliśmy, że to ustawienie nie daje się rozszerzyć na cztery hetmany. Spróbujmy więc postawić teraz pierwszego hetmana na polu a3 (rys. 15a). Na rysunkach 6b, c i d zilustrowano dostawianie kolejnego hetmana w następnych kolumnach – na ostatnim z nich jest pokazane końcowe ustawienie czterech nieatakujących się hetmanów.

Możemy podsumować to postępowanie formułując algorytm, jaki wykonywaliśmy.

Algorytm. Rozstawianie hetmanów
Ustal kolejność przeglądania kolumn szachownicy i wykonuj następujące ruchy (kroki), zaczynając od kolumny pierwszej.
Ruch do przodu. Gdy znajdujesz się w ustalonej kolumnie, wybierz w niej dla hetmana pierwsze nie rozpatrzone w tym wykonaniu **Ruchu do przodu** pole, które nie jest atakowane przez żadnego hetmana ustawionego w którejś z poprzednich kolumn.

1. Jeśli istnieje takie pole, to ustaw na nim hetmana, a następnie:
   1.1. jeśli jest to ostatnia kolumna, to wypisz uzyskane ustawienie hetmanów, złożone z *n* hetmanów, gdzie *n* jest rozmiarem szachownicy. Jeśli ma być znalezione tylko jedno rozstawienie hetmanów, to zakończ ten algorytm. Jeśli chcesz przekonać się, czy istnieją jeszcze inne rozstawienia, to wykonaj **nawrót** do poprzedniej kolumny i wykonaj dla niej **Ruch do przodu**;



Rysunek 14.
Próby uzupełnienia hetmana stojącego na polu a4 (rys. a) do pełnego ustawienia
4 nieatakujących się hetmanów – niepowodzenie

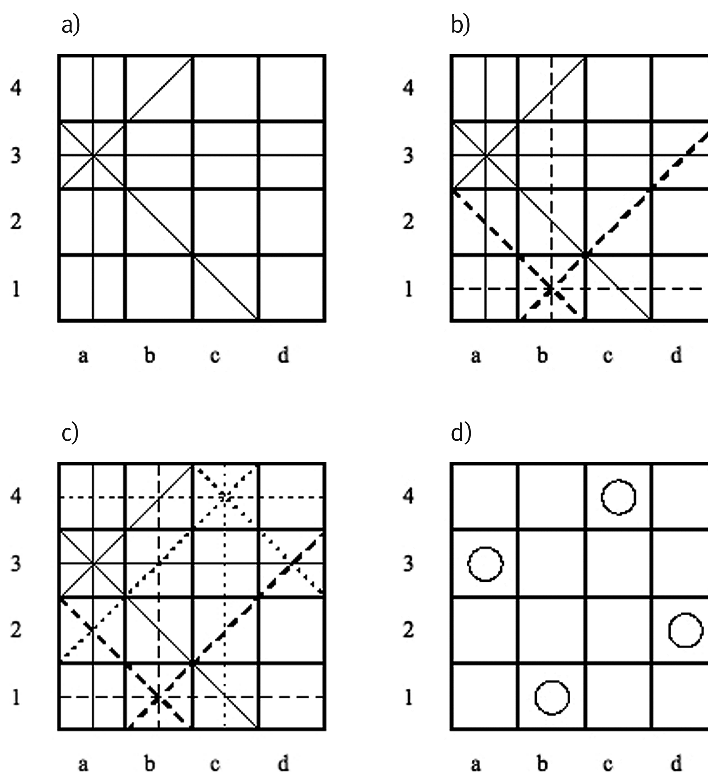
1.2. jeśli nie jest to ostatnia kolumna, to przejdź do następnej kolumny i wykonaj dla tej kolumny **Ruch do przodu**.

2. Jeśli nie istnieje takie pole, to wykonaj **nawrót** do poprzedniej kolumny i przejdź do wykonania dla niej **Ruchu do przodu**.

Rysunki 14b i 14c są ilustracją **nawrotu**, czyli ruchu do tyłu podczas poszukiwania miejsca dla kolejnego hetmana. Będąc w drugiej kolumnie, najpierw zostało wykorzystane pole b2, ale okazało się, że nie ma wolnego pola w trzeciej kolumnie dla trzeciego hetmana, wróciliśmy więc do drugiej kolumny i ustawiliśmy hetmana na drugim wolnym polu w tej kolumnie, czyli na b1. Okazało się, że i tym razem, ale w czwartej kolumnie, zabrakło nieatakowanego miejsca dla hetmana. W tej sytuacji, można powiedzieć, że dla ustawienia hetmana w pierwszej kolumnie na polu a4, w drugiej kolumnie sprawdziliśmy wszystkie możliwe ustawienia drugiego hetmana. Wykonujemy więc nawrót do pierwszej kolumny i ustawiamy hetmana na następnym możliwym polu, czyli na a3. Jak pokazują ilustracje na rys. 15, ta pozycja hetmana w pierwszej kolumnie może być uzupełniona do pełnego rozmieszczenia 4 hetmanów. .

Zilustrowany algorytm, po znalezieniu pełnego rozstawienia hetmanów (rys. 14.2d), można kontynuować (patrz krok 1.1), aby się przekonać, czy nie ma jeszcze innego ustawienia czterech nieatakujących się hetmanów.

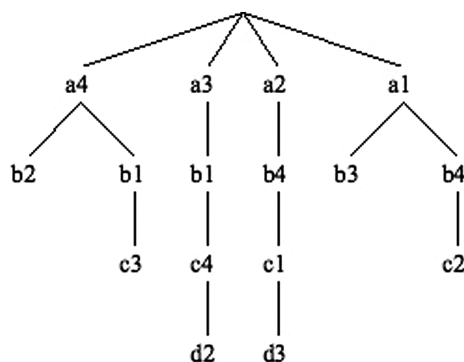
Ćwiczenie 67. Kontynuuj przerwany algorytm, by znaleźć jeszcze inne rozstawienia czterech nieatakujących się hetmanów na szachownicy 4×4. Znajdziesz jeszcze jedno pełne rozstawienie, a więc jest ich dwa. Jaka jest między nimi zależność?



Rysunek 15.
Próby uzupełnienia hetmana stojącego na polu a3, do pełnego rozmieszczenia hetmanów, zakończone sukcesem

Ćwiczenie 68. Uzasadnij, w jaki sposób można otrzymać to drugie rozwiązanie (otrzymane w ćwicz. 67) bez uruchamiania dalszego przebiegu algorytmu poszukiwania z nawrotami. Jakie jeszcze inne przekształcenia planszy szachownicy, a wraz z nimi – końcowego ustawienia hetmanów, mogą dać nowe ustawienia?

Przebieg poszukiwania z nawrotami przedstawia się zwykle w postaci **drzewa poszukiwań**. Dla naszego przykładu szachownicy 4×4 takie drzewo jest przedstawione na rys. 16. Poziomy w tym drzewie odpowiadają kolumnom (na pierwszym poziomie mamy pola a*, na drugim – pola b*, na trzecim – pola c* i na czwartym – d*), a wierzchołki – stawianym hetmanom (na rysunku podaliśmy w wierzchołkach pozycje hetmanów na planszy). Zauważ symetrię w tym drzewie względem pionowej osi – odpowiada ona symetrii planszy, którą zapewne wykorzystałeś w rozwiązaniach ćwiczeń 67 i 68.



Rysunek 16.
Drzewo ilustrujące przebieg algorytmu poszukiwań z nawrotami, służącego do znalezienia wszystkich ustawień czterech nieatakujących się hetmanów na szachownicy o wymiarach 4×4

Ćwiczenie 69. Narysuj szachownicę 5×5 i posługując się opisanym algorytmem rozstawiania hetmanów znajdź na niej wszystkie ustawienia pięciu nieatakujących się hetmanów.

Ćwiczenie 70. Weź prawdziwą szachownicę i spróbuj ustawić na niej 8 nieatakujących się hetmanów. Po otrzymaniu pierwszego ustawienia, wykonaj dalsze kroki algorytmu, by otrzymać kilka następnych ustawień.

Opis poszukiwania z nawrotami w języku Pascal

Zaprogramowanie poszukiwania z nawrotami nie jest specjalnie trudne. Najpierw musimy rozstrzygnąć dwie kwestie: w jaki sposób reprezentować ustawienie hetmanów na planszy i jak sprawdzać, czy dwa hetmany nie atakują się.

1. Ponieważ w każdej kolumnie może się znaleźć co najwyżej jeden hetman, ustawienie nieatakujących się hetmanów można opisać, podając dla każdej kolumny jedynie numer wiersza, w którym stoi hetman znajdujący się w tej kolumnie. Niech `h[1..n]` będzie tablicą, w której `h[i]` jest numerem wiersza zawierającego hetmana, stojącego w kolumnie `i`. Ustawienie z rys. 15d można opisać jako (3, 1, 4, 2).
2. Ustawiając kolejnego hetmana musimy sprawdzić, czy jego pozycja nie jest atakowana przez żadnego innego hetmana, ustawionego we wcześniejszych kolumnach. Przypuśćmy, że chcemy ustawić hetmana w kolumnie `i` w wierszu `h[i]`. Najpierw musimy sprawdzić, że żaden hetman w poprzednich kolumnach nie stoi w tym samym wierszu, czyli ma być $h[l] \neq h[i]$ dla każdego `l` spełniającego $l < i$. Ponadto, aby dwa hetmany nie atakowały się po przekątnych, musi być spełniony warunek $|h[l] - h[i]| \neq |l - i|$ dla każdego `l` spełniającego $l < i$.

Ćwiczenie 71. Uzasadnij, że rzeczywiście spełnienie warunku wymienionego w punkcie 2 powyżej gwarantuje, że żadne dwa hetmany nie atakują się po przekątnych. Sprawdź najpierw na przykładzie szachownicy 4×4.

Przedstawiamy poniżej funkcję w języku Pascal o nagłówku:

```
function Hetmany(n:integer):integer;
```

której wartością jest liczba różnych rozstawień nieatakujących się hetmanów na szachownicy `n` x `n`. Tekst tej funkcji zawiera wiele komentarzy, ułatwiających zrozumienie przeznaczenia poszczególnych instrukcji i efektów ich działania. Wyjaśnijmy jedynie, że `x[i]` oznacza numer następnego wiersza w kolumnie `i`, w którym można ustawić hetmana nie atakowanego przez żadnego innego hetmana z poprzednich kolumn.

```
function Hetmany(n:integer):integer;
  {Wartoscia tej funkcji jest liczba ustawien nieatakujacych sie
   hetmanów na szachownicy n x n. W trakcie obliczania wartości
   tej funkcji sa wypisywane kolejno znajdowane ustawienia.
   W programie glownym nalezy zdefiniowac typ danych:
       WektorI=array[1..n] of integer;}

 var h,x       :WektorI;
     i,Licznik:integer;

 procedure WolnePole(i:integer);
   {Okreslane jest nastepne wolne pole w kolumnie i.}
  var b:Boolean;
      l:integer;
```

```
 begin
  b:=false;
  while not b and (x[i]<=n) do begin
   b:=true;
   l:=1;
   while b and (l<=i-1) do begin
    b:=b and (abs(h[l]-x[i])<>abs(l-i)) and (h[l]<>x[i]);
    l:=l+1
   end;
   if not b then x[i]:=x[i]+1
  end
 end; {WolnePole}


 procedure WypiszUstawienie;
   {Wypisywane jest kolejne ustawienie hetmanow.}
  var j:integer;
 begin
  for j:=1 to n do write(h[j],'  ');
  writeln
 end; {WypiszUstawienie}


begin
 Licznik:=0;   {Licznik = liczba znalezionych ustwien}
 x[1]:=1;
 i:=1;
 while i>0 do begin
   {W tej petli sa przegladane wszystkie kolumny.}
  while x[i]<=n do begin
    {W tej petli sa przegladane nieatakowane pola w kolumnie i.}
   h[i]:=x[i];     {Ustawienie hetmana w kolumnie i.}
   if i=n then begin
    WypiszUstawienie;   Licznik:=Licznik+1;
    x[i]:=n+1   {Wymuszenie powrotu do poprzedniej kolumny}
   end {i=n}
   else begin
    x[i]:=x[i]+1;
    WolnePole(i);   {Wolne pole w kolumnie i.}
    i:=i+1;
    x[i]:=1;
    WolnePole(i)    {Wolne pole w nastepnej kolumnie.}
   end {i<n}
  end; {while}
  i:=i-1    {Powrot do poprzedniej kolumny.}
 end; {while i>0}
 Hetmany:=Licznik
end; {Hetmany}
```

Ćwiczenie 72. Umieść opis funkcji `Hetmany` w pełnym programie i uruchom ten program. Sprawdź działanie tego programu na szachownicy o rozmiarze 4×4.

Ćwiczenie 73. Zastosuj program opracowany w poprzednim zadaniu do szachownicy o rozmiarach: `n` x `n` dla `n` = 5, 6, 7 i do tradycyjnej szachownicy 8×8. Dla `n` = 5 i 6 wypisz wszystkie ustawienia nieatakujących się hetmanów. Dla ustalonego `n`, pogrupuj te ustawienia, które mogą być otrzymane przez zastosowanie operacji symetrii.

10 DODATEK: ALGORYTM, ALGORYTMIKA I ALGORYTMICZNE ROZWIĄZYWANIE PROBLEMÓW

Ten rozdział jest krótkim wprowadzeniem do zajęć w module „Algorytmika i programowanie". Krótko wyjaśniamy w nim podstawowe pojęcia oraz stosowane na zajęciach podejście do rozwiązywania problemów z pomocą komputera.

Algorytm

Powszechnie przyjmuje się, że **algorytm** jest opisem krok po kroku rozwiązania postawionego problemu lub sposobu osiągnięcia jakiegoś celu. To pojęcie wywodzi się z matematyki i informatyki – za pierwszy algorytm uznaje się bowiem algorytm Euklidesa (patrz rozdz. 6), podany ponad 2300 lat temu. W ostatnich latach algorytm stał się bardzo popularnym synonimem przepisu lub instrukcji postępowania.

W szkole, algorytm pojawia się po raz pierwszy na lekcjach matematyki już w szkole podstawowej, na przykład jako algorytm pisemnego dodawania dwóch liczb, wiele klas wcześniej, zanim staje się przedmiotem zajęć informatycznych.

O znaczeniu algorytmów w informatyce może świadczyć następujące określenie, przyjmowane za definicję informatyki:

informatyka jest dziedziną wiedzy i działalności zajmującą się algorytmami

W tej definicji informatyki nie ma dużej przesady, gdyż zawarte są w niej pośrednio inne pojęcia stosowane do definiowania informatyki: **komputery** – jako urządzenia wykonujące odpowiednio dla nich zapisane algorytmy (czyli niejako wprawiane w ruch algorytmami); **informacja** – jako materiał przetwarzany i produkowany przez komputery; **programowanie** – jako zespół metod i środków (np. języków i systemów użytkowych) do zapisywania algorytmów w postaci programów.

Położenie nacisku w poznawaniu informatyki na algorytmy jest jeszcze uzasadnione tym, że zarówno konstrukcje komputerów, jak i ich oprogramowanie bardzo szybko się starzeją, natomiast podstawy stosowania komputerów, które są przedmiotem zainteresowań algorytmiki, zmieniają się bardzo powoli, a niektóre z nich w ogóle nie ulegają zmianie.

Algorytmy, zwłaszcza w swoim popularnym znaczeniu, występują wszędzie wokół nas – niemal każdy ruch człowieka, zarówno angażujący jego mięśnie, jak i będący jedynie działaniem umysłu, jest wykonywany według jakiegoś przepisu postępowania, którego nie zawsze jesteśmy nawet świadomi. Wiele naszych czynności potrafimy wyabstrahować i podać w postaci precyzyjnego opisu, ale w bardzo wielu przypadkach nie potrafimy nawet powtórzyć, jak to się dzieje lub jak to się stało[7].

Nie wszystkie postępowania z naszego otoczenia, nazywane algorytmami, są ściśle związane z komputerami i nie wszystkie przepisy działań można uznać za algorytmy w znaczeniu informatycznym. Na przykład nie są nimi na ogół przepisy kulinarne, chociaż odwołuje się do nich David Harel w swoim fundamentalnym dziele o algorytmach i algorytmice [3]. Otóż przepis np. na sporządzenie „ciągutki z wiśniami", którą za-

[7] Interesująco ujął to J. Nievergelt – *Jest tak, jakby na przykład stonoga chciała wyjaśnić, w jakiej kolejności wprawia w ruch swoje nogi, ale z przerażeniem stwierdza, że nie może iść dalej.*

chwycała się Alicja w Krainie Czarów, nie jest algorytmem, gdyż nie ma dwóch osób, które na jego podstawie, dysponując tymi samymi produktami, zrobiłyby taką samą, czyli jednakowo smakującą ciągutkę. Nie może być bowiem algorytmem przepis, który dla identycznych danych daje różne wyniki w dwóch różnych wykonaniach, jak to najczęściej bywa w przypadku robienia potraw według „algorytmów kulinarnych".

Algorytmika

Algorytmika to dział informatyki, zajmujący się różnymi aspektami tworzenia i analizowania algorytmów, przede wszystkim w odniesieniu do ich roli jako precyzyjnego opisu postępowania, mającego na celu znalezienie rozwiązania postawionego problemu. Algorytm może być wykonywany przez człowieka, przez komputer lub w inny sposób, np. przez specjalnie dla niego zbudowane urządzenie. W ostatnich latach postęp w rozwoju komputerów i informatyki był nierozerwalnie związany z rozwojem coraz doskonalszych algorytmów.

Informatyka jest dziedziną zajmującą się rozwiązywaniem problemów z wykorzystaniem komputerów. O znaczeniu algorytmu w informatyce może świadczyć fakt, że każdy program komputerowy działa zgodnie z jakimś algorytmem, a więc zanim zadamy komputerowi nowe zadanie do wykonania powinniśmy umieć „wytłumaczyć" mu dokładnie, co ma robić. Bardzo trafnie to sformułował Donald E. Knuth, jeden z najznakomitszych, żyjących informatyków:

Mówi się często, że człowiek dotąd nie zrozumie czegoś,
zanim nie nauczy tego – kogoś innego.
W rzeczywistości,
człowiek nie zrozumie czegoś naprawdę,
zanim nie zdoła nauczyć tego – komputera.

Staramy się, by prezentowane algorytmy były jak najprostsze i by działały jak najszybciej. To ostatnie żądanie może wydawać się dziwne, przecież dysponujemy już teraz bardzo szybkimi komputerami i szybkość działania procesorów stale rośnie (według prawa Moore'a podwaja się co 18 miesięcy). Mimo to istnieją problemy, których obecnie nie jest w stanie rozwiązać żaden komputer i zwiększenie szybkości komputerów niewiele pomoże, kluczowe więc staje się opracowywanie coraz szybszych algorytmów. Jak to ujął Ralf Gomory, szef ośrodka badawczego IBM:

Najlepszym sposobem przyspieszania komputerów
jest obarczanie ich mniejszą liczbą działań.

Algorytmiczne rozwiązywanie problemów

Komputer jest stosowany do rozwiązywania problemów zarówno przez profesjonalnych informatyków, którzy projektują i tworzą oprogramowanie, jak i przez tych, którzy stosują tylko technologię informacyjno-komunikacyjną, czyli nie wykraczają poza posługiwanie się gotowymi narzędziami informatycznymi. W obu przypadkach ma zastosowanie podejście do **rozwiązywania problemów algorytmicznych**, która polega na systematycznej pracy nad komputerowym rozwiązaniem problemu i obejmuje cały proces projektowania i otrzymania rozwiązania. Celem nadrzędnym tej metodologii jest otrzymanie **dobrego rozwiązania**, czyli takiego, które jest:

- **zrozumiałe dla każdego**, kto zna dziedzinę rozwiązywanego problemu i użyte narzędzia komputerowe,
- **poprawne**, czyli spełnia specyfikację problemu, a więc dokładny opis problemu,
- **efektywne**, czyli niepotrzebnie nie marnuje zasobów komputerowych, czasu i pamięci.

Ta metoda składa się z następujących sześciu etapów:

1. *Opis i analiza sytuacji problemowej*. Na podstawie opisu i analizy sytuacji problemowej należy w pełni zrozumieć, na czym polega problem, jakie są dane dla problemu i jakich oczekujemy wyników, oraz jakie są możliwe ograniczenia.
2. *Sporządzenie specyfikacji problemu*, czyli dokładnego opisu problemu na podstawie rezultatów etapu 1. **Specyfikacja problemu** zawiera:

- opis danych,
- opis wyników,
- opis relacji (powiązań, zależności) między danymi i wynikami.

Specyfikacja jest wykorzystana w następnym etapie jako specyfikacja tworzonego rozwiązania (np. programu).

3. *Zaprojektowanie rozwiązania.* Dla sporządzonej na poprzednim etapie specyfikacji problemu, jest projektowane rozwiązanie komputerowe (np. program), czyli wybierany odpowiedni algorytm i dobierane do niego struktury danych. Wybierane jest także środowisko komputerowe (np. język programowania), w którym będzie realizowane rozwiązanie na komputerze.
4. *Komputerowa realizacja rozwiązania.* Dla projektu rozwiązania, opracowanego na poprzednim etapie, jest budowane kompletne rozwiązanie komputerowe, np. w postaci programu w wybranym języku programowania. Następnie, testowana jest poprawność rozwiązania komputerowego i badana jego efektywność działania na różnych danych.
5. *Testowanie rozwiązania.* Ten etap jest poświęcony na systematyczną weryfikację poprawności rozwiązania i testowanie jego własności, w tym zgodności ze specyfikacją.
6. *Prezentacja rozwiązania.* Dla otrzymanego rozwiązania należy jeszcze opracować dokumentację i pomoc dla (innego) użytkownika. Cały proces rozwiązywania problemu kończy prezentacja innym zainteresowanym osobom (uczniom, nauczycielowi) sposobu otrzymania rozwiązania oraz samego rozwiązania wraz z dokumentacją.

Chociaż powyższa metodologia jest stosowana głównie do otrzymywania komputerowych rozwiązań, które mają postać programów napisanych w wybranym języku programowania, może być zastosowana również do otrzymywania rozwiązań komputerowych większości problemów z obszaru zastosowań informatyki i posługiwania się technologią informacyjno-komunikacyjną, czyli gotowym oprogramowaniem.

Dwie uwagi do powyższych rozważań.

Uwaga 1. Wszyscy, w mniejszym lub większym stopniu, zmagamy się z problemami, pochodzącymi z różnych dziedzin (przedmiotów). W naszych rozważaniach, problem nie jest jednak wyzwaniem nie do pokonania, przyjmujemy bowiem, że **problem** jest sytuacją, w której uczeń ma przedstawić jej rozwiązanie bazując na tym, co wie, ale nie ma powiedziane, jak to ma zrobić. Problem na ogół zawiera pewną trudność, nie jest rutynowym zadaniem. Na takie sytuacje problemowe rozszerzamy pojęcie problemu, wymagającego przedstawienia rozwiązania komputerowego.

Uwaga 2. W tych rozważaniach rozszerzamy także pojęcie **programowania**. Jak powszechnie wiadomo, komputery wykonują tylko programy. Użytkownik komputera może korzystać z istniejących programów (np. za pakietu Office), a może także posługiwać się własnymi programami, napisanymi w języku programowania, który „rozumieją” komputery. W szkole nie ma zbyt wiele czasu, by uczyć programowania, uczniowie też nie są odpowiednio przygotowani do programowania komputerów. Istnieje jednak wiele sposobności, by kształcić zdolność komunikowania się z komputerem za pomocą programów, które powstają w inny sposób niż za pomocą programowania w wybranym języku programowania. Szczególnym przypadkiem takich programów jest oprogramowanie edukacyjne, które służy do wykonywania i śledzenia działania algorytmów. „Programowanie” w przypadku takiego oprogramowania polega na dobieraniu odpowiednich parametrów, które mają wpływ na działanie algorytmów i tym samym umożliwiają lepsze zapoznanie się z nimi.

LITERATURA

1. Cormen T.H., Leiserson C.E., Rivest R.L., *Wprowadzenie do algorytmów*, WNT, Warszawa 1997
2. Gurbiel E., Hard-Olejniczak G., Kołczyk E., Krupicka H., Sysło M.M., *Informatyka, Część 1 i 2, Podręcznik dla LO*, WSiP, Warszawa 2002-2003
3. Harel D., *Algorytmika. Rzecz o istocie informatyki*, WNT, Warszawa 1992
4. Knuth D.E., *Sztuka programowania*, Tomy 1 – 3, WNT, Warszawa 2003
5. Sysło M.M., *Algorytmy*, WSiP, Warszawa 1997
6. Sysło M.M., *Piramidy, szyszki i inne konstrukcje algorytmiczne*, WSiP, Warszawa 1998. Kolejne rozdziały tej książki są zamieszczone na stronie: http://www.wsipnet.pl/kluby/informatyka_ekstra.php?k=69
7. Wirth N., *Algorytmy + struktury danych = programy*, WNT, Warszawa 1980

W projekcie **Informatyka +**, poza wykładami i warsztatami,
przewidziano następujące działania:

- 24-godzinne kursy dla uczniów w ramach modułów tematycznych
- 24-godzinne kursy metodyczne dla nauczycieli, przygotowujące do pracy z uczniem zdolnym
- nagrania 60 wykładów informatycznych, prowadzonych przez wybitnych specjalistów i nauczycieli akademickich
- konkursy dla uczniów, trzy w ciągu roku
- udział uczniów w pracach kół naukowych
- udział uczniów w konferencjach naukowych
- obozy wypoczynkowo-naukowe.

Szczegółowe informacje znajdują się na stronie projektu
www.informatykaplus.edu.pl