informatyka+

Algorytmika i programowanie

Bazy danych

Multimedia, grafika i technologie internetowe

Sieci komputerowe

Tendencje w rozwoju informatyki i jej zastosowań

Człowiek – najlepsza inwestycja

KAPITAŁ LUDZKI
NARODOWA STRATEGIA SPÓJNOŚCI

Warszawska Wyższa Szkoła Informatyki

UNIA EUROPEJSKA
EUROPEJSKI
FUNDUSZ SPOŁECZNY

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

informatyka+

Wszechnica Poranna: Algorytmika i programowanie

Techniki algorytmiczne – przybliżone i dokładne

Maciej M Sysło

Człowiek – najlepsza inwestycja

KAPITAŁ LUDZKI
NARODOWA STRATEGIA SPÓJNOŚCI

Warszawska Wyższa Szkoła Informatyki

UNIA EUROPEJSKA
EUROPEJSKI
FUNDUSZ SPOŁECZNY

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

Techniki algorytmiczne – przybliżone i dokładne

Rodzaj zajęć: Wszechnica Poranna
Tytuł: Techniki algorytmiczne – przybliżone i dokładne
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

KAPITAŁ LUDZKI
NARODOWA STRATEGIA SPÓJNOŚCI

Warszawska Wyższa Szkoła Informatyki

UNIA EUROPEJSKA
EUROPEJSKI FUNDUSZ SPOŁECZNY

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

Techniki algorytmiczne – przybliżone i dokładne

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

Streszczenie

Te zajęcia są trzecią częścią wprowadzenia do algorytmiki i programowania. Omawiane są jednak również podstawowe pojęcia z zakresu algorytmiki, takie jak: specyfikacja problemu, podstawowe struktury danych (tablice jedno- i dwuwymiarowe) oraz efektywność i pracochłonność (złożoność) algorytmów. Na warsztatach zostają wprowadzone podstawowe instrukcje języka programowania (iteracyjna i warunkowa oraz procedura i funkcja niestandardowa), wystarczające do zaprogramowania i uruchomienia komputerowych realizacji algorytmów omówionych na wykładzie. Przytoczono ciekawe przykłady zastosowań omawianych zagadnień.

Zakres tematyczny obejmuje różnorodne **techniki algorytmiczne** i ich wykorzystanie w rozwiązaniach wybranych problemów. **Metoda zachłanna** jest użyta m.in. do wydawania reszty, pakowania plecaka i chodzenia po piramidzie. **Przeszukiwanie z nawrotami** służy do znajdowania wyjścia z labiryntu i rozstawiania nieatakujących się hetmanów na szachownicy. **Strategia dziel i zwyciężaj** jest jedynie przypomniana – pojawiła się na wcześniejszych zajęciach przy poszukiwaniu elementów i przeszukiwaniu zbiorów uporządkowanych. Zaś **rekurencja** jest zilustrowana m.in. w algorytmie wypisywania liczb w różnych systemach.

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.

i+

Spis treści

1 WPROWADZENIE

Celem tych zajęć jest zwrócenie uwagi uczniów na ogólne metody, techniki i strategie stosowane przy rozwiązywaniu przeróżnych problemów. Wśród tych ogólnych metod można wyróżnić m.in. konstruowanie rozwiązań w sposób zachłanny, przeszukiwanie z nawrotami przestrzeni rozwiązań, strategię dziel i zwyciężaj oraz rekurencję, która bardzo mocno wiąże się z komputerowymi metodami rozwiązywania problemów. Ogólność tych metod polega na tym, że rządzą nimi pewne ogólne zasady postępowania, niezależne od problemów, mogą więc być one stosowane do rozwiązywania najprzeróżniejszych problemów.

Rozważania ogólne na temat algorytmiki, algorytmicznego myślenia i rozwiązywania problemów z pomocą komputerów są zamieszczone w Dodatku (rozdz. 6). Materiał tam zawarty może być dobrym podsumowaniem zajęć.

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

2 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.

2.1 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.

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

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.

(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[1], 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.

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 1. 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. 1. 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 2. 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 3. 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] 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.

	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 1.
Arkusz służący do wydawania reszty metodą zachłanną

```
 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.
```

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 4. 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. 3. Porównaj wyniki otrzymane w arkuszu i otrzymane za pomocą swojego programu.

Ćwiczenie 5. 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 6. 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?

Nominały większości walut na świecie nie są przypadkowymi liczbami. Zostały one tak wybrane, by sprzedawcy dysponujący dostatecznym zasobem monet mogli stosować metodę zachłanną do wydawania reszt w postaci najmniejszej liczby banknotów i 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 7. 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 8. 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.

2.2 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 1 są zamieszczone przykładowe godziny wyświetlania filmów. Pytanie: ile z tych filmów może obejrzeć kinoman w całości jednego dnia?

Tabela 1.
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 9. 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 1.

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 10. 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.

2.3 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 założymy, 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 2.
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 11. Dla danych zamieszczonych w tabeli 2 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 12. Nie powinieneś mieć większego kłopotu z zapakowaniem do plecaka rzeczy o łącznej wartości 80 – jak?

Wykazaliśmy na tym przykładzie – co uświadamia nam ćwicz. 12 – 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 13. Uzupełnij powyższy fragment programu do pełnego programu, służącego do wyznaczania zachłannego rozwiązania ogólnego problemu plecakowego. Sprawdź działanie swojego programu na przykładzie z tabeli 2 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 14. Uzupełnij program otrzymany w ćwicz. 13 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 15. Znajdź rozwiązania zachłanne dla decyzyjnego problemu plecakowego, dla którego dane znajdują się w tabeli 2. Czy jest wśród nich rozwiązanie optymalne, czyli możliwie najlepsze? Uzasadnij swoją odpowiedź.

Ćwiczenie 16. Zmodyfikuj program otrzymany w ćwicz. 13 tak, aby rozwiązywał decyzyjny problem plecakowy. Podobnie, w tym celu zmodyfikuj program otrzymany w ćwicz. 14, jeśli je rozwiązałeś.

2.4 NAJDŁUŻSZA DROGA NA PIRAMIDZIE

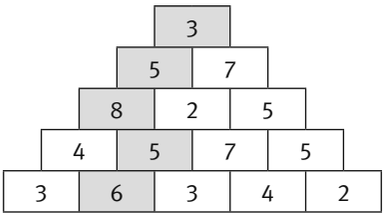
Kolejny problem ma charakter łamigłówki liczbowej. Polega on na znalezieniu takiej drogi przejścia z wierzchołka piramidy – patrz rys. 2 – do punktu w podstawie piramidy, aby suma elementów, przez które prowadzi droga, była jak największa. Droga z korzenia do podstawy powinna zawierać dokładnie jeden element z każdego poziomu i każde dwa elementy z sąsiednich poziomów powinny ze sobą sąsiadować – taką drogę zaznaczono na rys. 2 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 17. Znajdź rozwiązanie zachłanne dla piramidy pokazanej na rys. 2. 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 18. 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)-szym będą sąsiednie dla *k*-tego elementu w wierszu *i*-tym.



Rysunek 2.
Piramida z wagami

Teraz mamy nieco trudniejsze zadanie.

Ćwiczenie 19. 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. 18.

2.5 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. 3.1. 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].

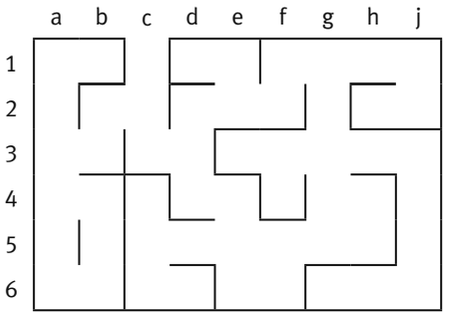
3 PRZESZUKIWANIE Z NAWROTAMI

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.

3.1 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. 3.



Rysunek 3.
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. 3 – 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 labiryncie.

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 wykonaj krok 1.

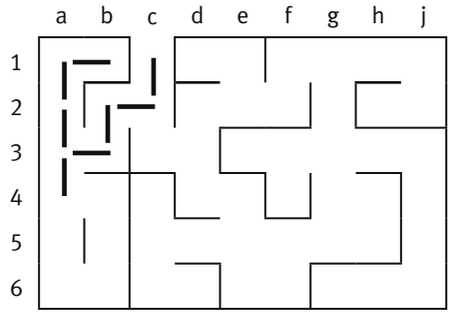
Ruch, będący powrotem, będziemy oznaczali literą B. Każde przejście możemy opisać nazwą ruchu (kierunku) i nazwą pola, na które nas wiedzie. Zróbmy jeszcze jedno, dość naturalne założenie: kierunek poruszania się po labiryncie określamy w zależności od naszego ustawienia i przyjmujemy przy tym, że cały 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. 3. 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. 4 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 rucha-

mi 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 4.
Poszukiwanie wyjścia z labiryntu przedstawionego na rys. 3, 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 20. Posługując się algorytmem **Zgłębianie labiryntu** wypisz kolejno odwiedzane pola na drodze do wyjścia z pola 4g w labiryncie pokazanym na rys. 3. 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.

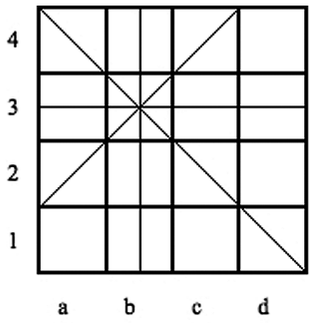
Ćwiczenie 21. Uzasadnij, że z każdego pola labiryntu algorytm z nawrotami **Zgłębianie labiryntu** znajdzie drogę z tego pola do wyjścia z labiryntu.

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.

3.2 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. 5 – hetman stoi 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 oczywi-



Rysunek 5.
Pola atakowane przez hetmana, stojącego na polu b3

ście nie wszystkich ustawień, bo jest ich bardzo dużo[2]. 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 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 22. 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.

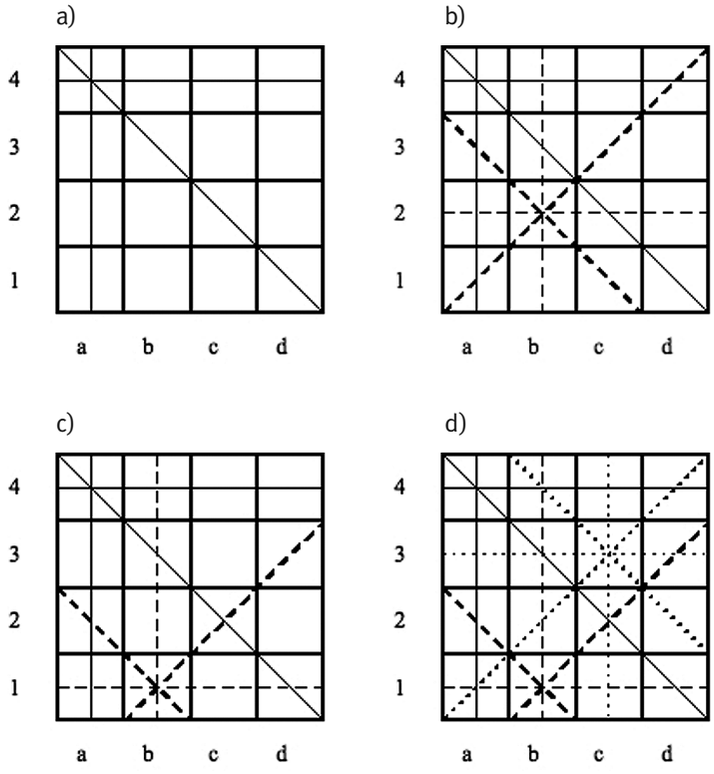
Zanim sformułujemy podstawową regułę, na której jest oparte przeszukiwanie z nawrotami, prześledźmy początkowe próby ustawienia 4 hetmanów na szachownicy 4x4. Zgodnie z przyjętym założenie o kierunku poszukiwań, w kolejnych ruchach będziemy się posuwali kolumnami, począwszy od kolumny a, i od góry, czyli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

Na rysunkach 6 i 7 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. 6a). W drugiej kolumnie pozostają wolne pola b2 i b1. Ustawiamy więc hetmana najpierw na polu b2 (rys. 6b) 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. 6c). Tym razem w trzeciej kolumnie jest wolne pole c3 – ustawiamy więc na nim trzeciego hetmana i otrzymujemy konfigurację pokazaną na rys. 6d. 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

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

postawić teraz pierwszego hetmana na polu a3 (rys. 7a). Na rysunkach 7b, 7c i 7d zilustrowano dostawianie kolejnego hetmana w następnych kolumnach – na ostatnim z nich jest pokazane końcowe ustawienie czterech nieatakujących się hetmanów.



Rysunek 6.
Próby uzupełnienia hetmana stojącego na polu a 4 (rys. a) do pełnego ustawienia 4 nieatakujących się hetmanów – niepowodzenie

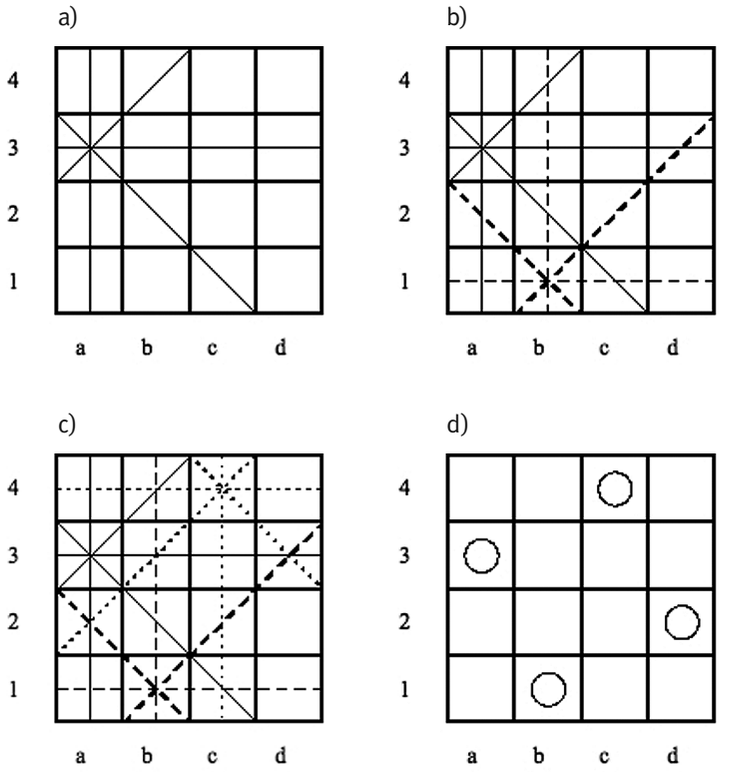
Możemy podsumować to postępowanie formułując algorytm, jaki wykonywaliśmy.

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**;
   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 6b i 6c 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. 7, ta pozycja hetmana w pierwszej kolumnie może być uzupełniona do pełnego rozmieszczenia 4 hetmanów. .



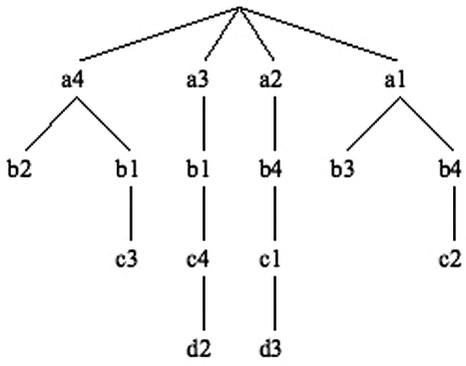
Rysunek 7.
Próby uzupełnienia hetmana, stojącego na polu a3, do pełnego rozmieszczenia hetmanów, zakończone sukcesem

Zilustrowany algorytm, po znalezieniu pełnego rozstawienia hetmanów (rys. 7d), można kontynuować (patrz krok 1.1), aby się przekonać, czy nie ma jeszcze innego ustawienia czterech nieatakujących się hetmanów.

Ćwiczenie 23. 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?

Ćwiczenie 24. Uzasadnij, w jaki sposób można otrzymać to drugie rozwiązanie (otrzymane w ćwicz. 23) 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. 8. 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 23 i 24.



Rysunek 8.
Drzewo ilustrujące przebieg algorytmu poszukiwań z nawrotami, służącego do znalezienia wszystkich ustawień czterech nieatakujących się hetmanów na szachownicy o wymiarach 4×4

Ćwiczenie 25. Narysuj szachownicę 5×5 i posługując się opisanym algorytmem rozstawiania hetmanów znajdź na niej wszystkie ustawienia pięciu nieatakujących się hetmanów.

Ćwiczenie 26. 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. 7d 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] ≠ 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]| ≠ |l − i|` dla każdego `l` spełniającego `l < i`.

Ćwiczenie 27. 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 × 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 28. Umieść opis funkcji `Hetmany` w pełnym programie i uruchom ten program. Sprawdź działanie tego programu na szachownicy o rozmiarze 4×4.

Ćwiczenie 29. Zastosuj program opracowany w poprzednim zadaniu do szachownicy o rozmiarach: `n`×`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.

4 STRATEGIA DZIEL I ZWYCIĘŻAJ

Jedną z najbardziej popularnych metod rozwiązywania problemów, zwłaszcza za pomocą komputera, jest metoda oparta na strategii dziel i zwyciężaj. Ogólnie, **metoda dziel i zwyciężaj**, użyta do rozwiązywania wybranego problemu, składa się z następujących kroków:

1. Problem jest dzielony na przynajmniej dwa podproblemy, definiowane na podzbiorach zbioru danych, w których liczba elementów jest niemal jednakowa.
2. Podproblemy są rozwiązywane ... tą samą lub bardzo podobną metodą.
3. Rozwiązania podproblemów są składane w rozwiązanie problem, który mieliśmy rozwiązać.

Krok 2 może się wydawać nieco dziwny – rozwiązujemy w nim podproblemy taką samą metodą, jak cały problem. Rzeczywiście, można mieć wątpliwości, jak to należy robić. Tutaj kłania się **myślenie rekurencyjne**, które polega na przykład na tym, że wartość pewnej wielkości obliczamy odwołując się do wielkości określonych dla mniejszych wartości parametrów.

Na tych zajęciach nie będziemy szczegółowo omawiać metody dziel i zwyciężaj, jedynie przypomnimy, gdzie ta metoda już się pojawiła na zajęciach w Projekcie Informatyka +. Dwa pierwsze przykłady poniżej są związane z wyszukiwaniem i porządkowaniem informacji.

Jednoczesne znajdowanie najmniejszego i największego elementu w zbiorze

W danym zbiorze liczb należy jednocześnie znaleźć elementy najmniejszy i największy (te elementy są np. potrzebne do obliczenia rozpiętości zbioru). Najszybszy, faktycznie – optymalny algorytm rozwiązywania tego problemu polega na wykonaniu dwóch kroków:

1. podziel zbiór danych na dwa podzbiory – w jednym powinny się znaleźć elementy kandydujące na minimum, a w drugim – kandydujące na maksimum; takie zbiory można otrzymać porównując elementy danego zbioru parami;
2. znajdź minimum w zbiorze kandydatów na minimum i maksimum w zbiorze kandydatów na maksimum.

Jeśli zbiór danych zawiera n elementów, to w tym algorytmie jest wykonywanych około $3n/2$ porównania i jest to optymalny pod względem złożoności algorytm dla tego problemu (szybszy już nie istnieje).

Zauważmy, że w podanym algorytmie dochodzi do podziału zbioru na podzbiory tylko raz. Inną metodą rozwiązywania tego problemu, o tej samej złożoności obliczeniowej, jest algorytm, w którym stosowana jest metoda dziel i zwyciężaj w sposób rekurencyjny, patrz p. 9.1 w książce [5].

Poszukiwania w zbiorze uporządkowanym

Chyba najczęściej stosowanym podejściem dziel i zwyciężaj, często nieświadomie, jest metoda poszukiwania elementu (lub miejsca dla nowego elementu) w zbiorze uporządkowanym, np. słowa w słowniku, numeru telefonu w książce telefonicznej, hasła w encyklopedii. Wszystko są to zbiory (książki), w których informacje są uporządkowane (we wszystkich tych książkach jest to porządek słownikowy, zwany **leksykograficznym**). Metoda ta nosi nazwę **poszukiwanie przez połowienie**, gdyż w kolejnych krokach, pozostały do przeszukania zbiór jest dzielony na dwie części i poszukiwanie przenosi się do jednej z nich, która ma nie więcej niż połowę elementów ze zbioru, występującego w poprzednim kroku.

Ten przykład problemu jest o tyle ciekawy, że w zastosowanej metodzie dziel i zwyciężaj, zbiór jest dzielony na dwie części, ale rozwiązywany jest tylko jeden podproblem, a mianowicie na podzbiorze, w którym jest szansa, że znajduje się poszukiwany przez nas element.

Znaczenie poszukiwania przez połowienie wynika z dużej efektywności tej metody – faktycznie jest to również algorytm optymalny. Otóż, gdyby słownik zawierał 1000 stron, ale słowa nie były w nim uporządkowane, to należałoby przejrzeć wszystkie 1000 stron, by odnaleźć poszukiwane słowo lub przekonać się, że go nie ma. Jeśli natomiast słowa w tym słowniku są uporządkowane, to wystarczy przejrzeć 10 stron, by odnaleźć poszukiwane słowo lub jego miejsce w tym słowniku. To porównanie efektywności algorytmów poszukiwania na zbiorach nieuporządkowanych i uporządkowanych świadczy również o olbrzymim znaczeniu porządku wśród informacji.

Porządkowanie (sortowanie)

Na jednych z zajęć omawialiśmy wybrane algorytmy porządkowania liczb. Nie doszliśmy jednak do algorytmów, które wykorzystują podejście dziel i zwyciężaj. Takimi algorytmami są sortowanie przez scalanie i sortowanie szybkie – dokładny opis tych algorytmów można znaleźć w książce [5].

5 REKURENCJA

Wiele wielkości w informatyce jest definiowanych rekurencyjnie, czyli przez odwołanie do tych samych wielkości ale z mniejszymi wartościami parametrów.

5.1 POTĘGOWANIE

Na jednych z zajęć omawialiśmy szybkie potęgowanie, czyli szybkie obliczanie wartości potęgi x^m, gdzie m jest liczbą naturalną, a x może być dowolną liczbą rzeczywistą. Do obliczenia potęgi korzystaliśmy tam z rozkładu wykładnika na postać binarną. Do tego samego algorytmu prowadzi rozumowanie rekurencyjne. 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: $x^{22} = (x^{11})^2 = (((x^5)^2 x)^2 = (((x^2)^2 x)^2 x)^2$. Identyczną zależność otrzymamy rozkładając wykładnik 22 na postać binarną.

5.2 ALGORYTM EUKLIDESA

Na innych zajęciach omawiany był również algorytm Euklidesa, służący do znajdowania największego wspólnego dzielnika dwóch danych liczb. Istnieje wiele implementacji algorytmu Euklidesa, np. wykonujących dzielenie lub odejmowanie zamiast dzielenia. Jedna z najciekawszych implementacji korzysta z **rekurencji**, czyli funkcji, która odwołuje się do siebie.

Danymi są dwie nieujemne liczby całkowite m i n. Oznaczamy przez NWD(m,n) największy wspólny dzielnik m i n, czyli największą liczbę k, która dzieli m oraz n. Załóżmy, że $n \geq m$. Wtedy dzieląc n przez m otrzymujemy następującą równość:

$$n = qm + r, \quad \text{gdzie } 0 \leq r < m.$$

Wielkości q i r są odpowiednio **ilorazem** i **resztą** z dzielenia n przez m. Z tej równości wynika, że każda liczba dzieląca n i m dzieli również r. Tę własność można zapisać w postaci równości:

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

w której przyjęliśmy, że NWD(0,m) = m, gdyż 0 jest podzielne przez każdą liczbę różną od zera. Prowadzi to do następującej, rekurencyjnej implementacji powyższej zależności:

```
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}
```

5.3 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 30. 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.

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**.

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. $m \text{ div } 10 \geq 0$. 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[3]:

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

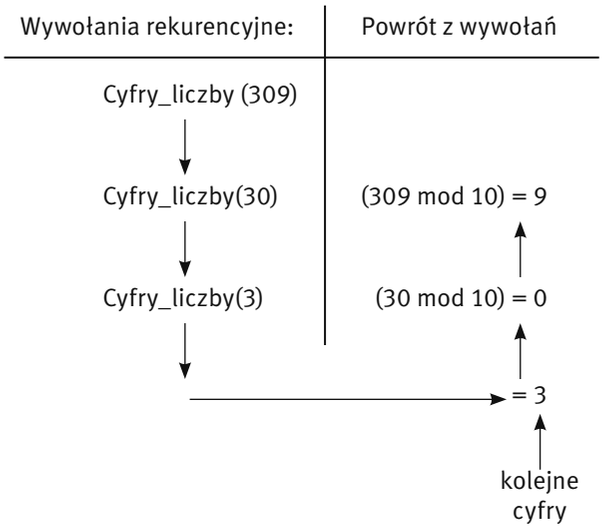
Jest to **zależność rekurencyjna** – ciąg kolejnych cyfr liczby *m* składa się z kolejnych cyfr liczby równej (m div 10) oraz cyfry (m mod 10). Ten proces rekurencyjnych odwołań jest wykonywany tak długo, aż bieżąca wartość *m* stanie się mniejsza od 10. Wtedy jest ona najbardziej znaczącą cyfrą liczby *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. 9. 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.}
```

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

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



Rysunek 9.
Przykład działania rekurencyjnego algorytmu generowania kolejnych cyfr w reprezentacji liczb

Ćwiczenie 31. 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 32. 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 33. 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 34. 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 35. 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 36. 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 37. 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.

5.4 REKURENCJA – PODSUMOWANIE

Na zakończenie tego rozdziału poświęconego rekurencji zwróćmy uwagę na dwie 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 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[3] .

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 zachwycała 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.

[3] 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.*

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 pokona-

nia, 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