Rozdział 2.

Rekurencja

Tematem niniejszego rozdziału jest jeden z najważniejszych mechanizmów używanych w informatyce — *rekurencja*, zwana również *rekursją*[1]. Mimo iż użycie rekurencji nie jest obowiązkowe, jej zalety są oczywiste dla każdego, kto choć raz spróbował tego stylu programowania. Wbrew pozorom nie jest to wcale mechanizm prosty i szereg jego aspektów wymaga dogłębnej analizy. Rekurencja upraszcza jednak opis wielu zagadnień (np. pozwala na łatwe definiowanie struktur opartych na fraktalach), a w wielu zagadnieniach informatycznych, np. w strukturach „drzewiastych", jest wręcz niezbędna z uwagi na ich charakter.

Niniejszy rozdział ma kluczowe znaczenie dla pozostałej części książki — o ile jej lektura może być dość swobodna i nieograniczona naturalną kolejnością rozdziałów, o tyle bez dobrego zrozumienia samej istoty rekurencji nie będzie możliwe swobodne czytanie wielu zaprezentowanych dalej algorytmów i metod programowania.

Definicja rekurencji

Rekurencja jest często przeciwstawiana podejściu iteracyjnemu, czyli *n*-krotnemu wykonywaniu algorytmów w taki sposób, aby wyniki uzyskane podczas poprzednich iteracji (zwanych też „przebiegami") mogły służyć jako dane wejściowe do kolejnych. Sterowanie iteracjami zapewniają instrukcje pętli (np. `for` lub `while`). Rekurencja działa podobnie, tylko funkcję zapętlania pełni wywoływanie się tej samej procedury (funkcji) przez siebie samą, z innymi parametrami. Oczywiście w ciele procedury rekurencyjnej też można spotkać klasyczne pętle, ale pełnią one rolę usługową i nie stanowią kryterium klasyfikacji algorytmu.

Warto wiedzieć, że programy zapisane w formie rekurencyjnej mogą być przekształcone — z mniejszym lub większym wysiłkiem — na klasyczną postać iteracyjną. Zagadnieniu temu poświęcę cały rozdział 6., na razie jednak zajmiemy się zrozumieniem mechanizmu rekurencji, co, jak się okazuje, nie jest takie proste, jak się wydaje na pierwszy rzut oka.

Pojęcie rekurencji poznamy na przykładzie. Wyobraźmy sobie małe dziecko w wieku lat — przykładowo — trzech. Dostaje ono od rodziców zadanie zebrania do pudełka wszystkich drewnianych klocków, które „nieumyślnie" zostały rozsypane na podłodze. Klocki są bardzo prymitywne, to zwyczajne drewniane sześcianiki, które doskonale nadają się do wznoszenia nieskomplikowanych budowli. Polecenie jest bardzo proste: „Zbierz to wszystko razem i poukładaj

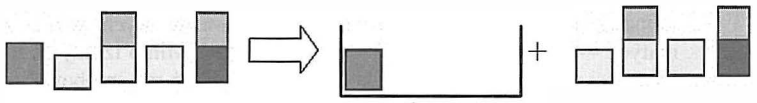
[1] Subtelna różnica między tymi pojęciami w zasadzie już się zatraciła w literaturze, dlatego też nie będziemy się niepotrzebnie zagłębiać w szczegóły terminologiczne.

tak, jak było w pudełku". Problem wyrażony w ten sposób jest dla dziecka potwornie skomplikowany: klocków jest cała masa i niespecjalnie wiadomo, jak się do tego całościowo zabrać. Mimo ograniczonych umiejętności na pewno nie przerasta go następująca czynność: *wziąć jeden klocek z podłogi i włożyć do pudełka*. Małe dziecko, zamiast przejmować się złożonością problemu, której być może sobie nawet nie uświadamia, bierze się do pracy i rodzice z przyjemnością obserwują, jak strefa porządku na podłodze powiększa się z minuty na minutę.

Zastanówmy się chwilkę nad metodą przyjętą przez dziecko: ono wie, że problem postawiony przez rodziców to wcale nie jest „zebrać wszystkie klocki" (bo to de facto jest niewykonalne za jednym zamachem), ale: „wziąć jeden klocek, przełożyć go do pudełka, a następnie zebrać do pudełka *pozostałe*". W jaki sposób można zrealizować to drugie? Proste, zupełnie tak, jak poprzednio: „bierzemy jeden klocek..." itd. — postępując tak aż do momentu wyczerpania się klocków.

Spójrzmy na rysunek 2.1, który przedstawia w sposób symboliczny tok rozumowania przyjęty przy rozwiązywaniu problemu „sprzątania rozsypanych klocków".

Rysunek 2.1. *„Sprzątanie klocków", czyli rekurencja w praktyce*



Jest mało prawdopodobne, aby dziecko uświadamiało sobie, że postępuje w sposób rekurencyjny, choć tak jest w istocie! Jeśli uważniej przyjrzymy się opisanemu powyżej problemowi, to zauważymy, że jego rozwiązanie charakteryzuje się następującymi, typowymi dla algorytmów rekurencyjnych, cechami:

- Zakończenie algorytmu jest jasno określone („w momencie gdy na podłodze nie będzie już klocków, możesz uznać, że zadanie zostało wykonane").
- „Duży", złożony problem został rozłożony na problemy elementarne (które umiemy rozwiązać) i na problemy o mniejszym stopniu skomplikowania niż ten, z którym mieliśmy do czynienia na początku.

Zauważmy, że w sposób dość śmiały użyte zostało określenie „algorytm". Czy jest sens mówić o opisanym powyżej problemie w kategorii algorytmu? Czy w ogóle możemy przypisywać pięcioletniemu dziecku wiedzę, z której nie zdaje sobie ono sprawy?

Przykład, na podstawie którego zostało wyjaśnione pojęcie algorytmu rekurencyjnego, jest niewątpliwie kontrowersyjny. Prawdopodobnie dowolny specjalista od psychologii zachowań dziecka chwyciłby się za głowę z rozpaczy, czytając powyższy wywód. Dlaczego jednak zdecydowałem się na użycie takiego właśnie, a nie innego — może bardziej informatycznego — przykładu? Otóż zasadniczym celem była chęć udowodnienia, iż myślenie w sposób rekurencyjny jest jak najbardziej zgodne z naturą człowieka i duża klasa problemów rozwiązywanych przez umysł ludzki jest traktowana podświadomie w sposób rekurencyjny. Pójdźmy dalej za tym śmiałym stwierdzeniem: jeśli tylko zdecydujemy się na intuicyjne podejście do algorytmów rekurencyjnych, to nie będą one stanowiły dla nas tajemnic, choć być może na początku nie będziemy w pełni świadomi wykorzystywanych w nich mechanizmów.

Powyższe wyjaśnienie pojęcia rekurencji powinno być znacznie czytelniejsze niż typowe podejście zatrzymujące się na niewiele mówiącym stwierdzeniu, że algorytm rekurencyjny jest to algorytm, który odwołuje się do samego siebie.

Ilustracja pojęcia rekurencji

Program, którego analizą będziemy się zajmowali w tym podrozdziale, jest bardzo zbliżony do problemu klocków, z jakim spotkaliśmy się przed chwilą. Schemat rekurencyjny zastosowany w nim jest identyczny, jedynie zagadnienie jest nieco bliższe rzeczywistości informatycznej.

Mamy do rozwiązania następujący problem:

- Dysponujemy tablicą *n* liczb całkowitych *tab[n] = tab[0], tab[1],..., tab[n–1]*.
- Czy w tablicy *tab* występuje liczba *x* (podana jako parametr)?

Jak postąpiłoby dziecko z przykładu, który posłużył nam za definicję pojęcia rekurencji, zakładając oczywiście, że dysponuje już ono pewną elementarną wiedzą informatyczną? Jest wysoce prawdopodobne, że rozumowałoby w sposób następujący:

- wziąć pierwszy niezbadany element tablicy *n*-elementowej;
- jeśli aktualnie analizowany element tablicy jest równy x, to:

  ***ogłoś** „Sukces" **i zakończ**;*

  w przeciwnym przypadku:

  ***zbadaj** pozostałą część tablicy.*

Wyżej podaliśmy warunki pozytywnego zakończenia programu. W przypadku gdy przebadaliśmy całą tablicę i element x nie został znaleziony, należy oczywiście zakończyć program w jakiś umówiony sposób — np. komunikatem o niepowodzeniu.

Proszę spojrzeć na przykładową realizację, jedną z kilku możliwych:


Listing

rek1.cpp

```
#include <iostream>
using namespace std;
const int n=10;
int tab[n]={1, 2, 3, 2, -7, 44, 5, 1, 0, -3};

void szukaj(int tab[n],int left,int right,int x){
// left, right = lewa i prawa granica obszaru poszukiwań
// tab = tablica
// x = wartość do odnalezienia
if (left>right)
   cout << "Element " << x << " nie został odnaleziony\n";
else
   if (tab[left]==x)
      cout << "Znalazłem szukany element "<< x << endl;
   else
      szukaj(tab,left+1,right,x);
}
int main()
{
szukaj(tab,0,n-1,7);
szukaj(tab,0,n-1,5);
}
```

Warunkiem zakończenia programu jest albo znalezienie szukanego elementu x, albo też wyjście poza obszar poszukiwań. Mimo swojej prostoty powyższy program dobrze ilustruje podstawowe, wspomniane już wcześniej cechy typowego programu rekurencyjnego. Przypatrzmy się zresztą uważniej:

- *Zakończenie* programu jest jasno określone:
  - Element zostaje znaleziony.
  - Następuje przekroczenie zakresu tablicy.
- Duży problem zostaje rozbity na problemy elementarne, które umiemy rozwiązać (patrz wyżej), i na analogiczny problem, tylko o mniejszym stopniu skomplikowania (z tablicy o rozmiarze n schodzimy do tablicy o rozmiarze n-1).

Na podstawie powyższych obserwacji można pokusić się o wyliczenie dwóch podstawowych błędów popełnianych przy konstruowaniu programów rekurencyjnych:

- złe określenie warunku zakończenia programu;
- niewłaściwa (nieefektywna) dekompozycja problemu.

W dalszej części rozdziału postaramy się wspólnie dojść do pewnych „zasad bezpieczeństwa", niezbędnych przy pisaniu programów rekurencyjnych. Zanim to jednak nastąpi, konieczne będzie dokładne wyjaśnienie schematu ich wykonywania.

Jak wykonują się programy rekurencyjne?

Dociekliwy Czytelnik będzie miał prawo zapytać w tym miejscu: „OK, zobaczyłem na przykładzie, że TO działa, ale mam też chyba prawo poznać nieco bardziej od podszewki, JAK to działa!". Pozostaje zatem podporządkować się temu słusznemu żądaniu.

Odpowiedzią na nie jest właśnie niniejszy podrozdział. Przykład w nim użyty będzie być może banalny, tym niemniej nadaje się doskonale do zilustrowania sposobu wykonywania programu rekurencyjnego.

Już w szkole średniej (lub może nawet podstawowej?!) na lekcjach matematyki dość często używa się tzw. *„silni n"*, czyli iloczynu wszystkich liczb naturalnych od *1* do *n* włącznie. Ten użyteczny symbol zdefiniowany jest w sposób następujący:

$$0! = 1,$$
$$n! = n*(n-1)!, \; gdzie \; n \in N, n \geq 1$$

Podstawiając za *n* wartości dozwolone (czyli większe od *1*), łatwo jest ręcznie wyliczyć początkowe wartości silni, dla dużych *n* nie jest to już takie proste, ale od czego są komputery — przecież nic nie stoi na przeszkodzie, aby napisać prosty program, który zajmuje się obliczaniem silni w sposób rekurencyjny:


Listing

rek2.cpp

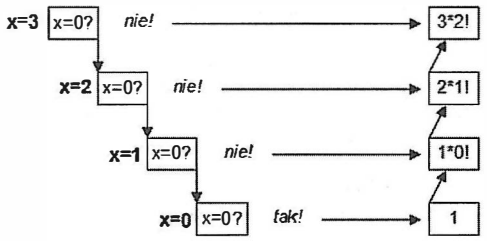
```
unsigned long int silnia(int x)
{
if (x==0)
   return 1;
else
   return x*silnia(x-1);
}

int main()
{
cout << "silnia(5)="<< silnia(5) << endl;
}
```

Prześledźmy na przykładzie, jak się wykonuje program, który obliczy `3!`. Rysunek 2.2 przedstawia kolejne etapy wywoływania procedury rekurencyjnej i badania warunku na przypadek elementarny.

Konwencje użyte na tym rysunku są następujące:

- Pionowe strzałki w dół oznaczają zagłębianie się programu z poziomu `n` na `n-1` itd. w celu dotarcia do przypadku elementarnego `0!`.
- Pozioma strzałka oznacza obliczanie wyników cząstkowych.
- Ukośna strzałka prezentuje proces przekazywania wyniku cząstkowego z poziomu niższego na wyższy.



Rysunek 2.2. *Drzewo wywołań funkcji silnia(3)*

Czymże są jednak owe tajemnicze *poziomy*, *przekazywanie parametrów*, etc.? Chwilowo te pojęcia mają prawo brzmieć lekko egzotycznie. Scharakteryzujmy zatem nieco dokładniej sposób obliczenia `silnia(1)`, opisując sposób działania programu:

- Funkcja `silnia` otrzymuje liczbę 1 jako parametr wywołania i analizuje: „czy 1 równa się 0?". Odpowiedź brzmi: „Nie", zatem funkcja przyjmuje, że jej wynikiem jest `1*silnia(0)`, czyli po prostu `silnia(0)`.
- Niestety wartość `silnia(0)` jest nieznana. Funkcja wywołuje zatem kolejny swój egzemplarz, który zajmie się obliczeniem wartości `silnia(0)`, wstrzymując jednocześnie obliczanie wyrażenia `1*silnia(0)`.
- Wywołanie funkcji `silnia(0)` zwraca już konkretny, liczbowy wynik cząstkowy (`1`), który może zostać użyty do obliczenia wyrażenia `1*silnia(0)`, czyli `silnia(1)`.

Technicznie przekazywanie parametrów odbywa się za pośrednictwem tzw. stosu, czyli specjalnego miejsca w pamięci operacyjnej, które jest używane do zapamiętywania informacji potrzebnych podczas wykonywania programów i dynamicznego przydzielania pamięci. Programista ma jednak prawo zupełnie się tym nie przejmować. Fakt, iż parametr zostanie zwrócony za pośrednictwem stosu, niewiele się bowiem różni od przedyktowania wyniku przez telefon. Końcowy efekt, wyrażony przez stwierdzenie „*Wynik jest gotowy!*", jest bowiem dokładnie taki sam w każdym przypadku, niezależnie od realizacji.

Gdzież się jednak znajdują wspomniane poziomy rekurencji? Spójrzmy raz jeszcze na rysunek 2.2. Aktualna wartość parametru `x` badanego przez funkcję `silnia` jest zaznaczona z lewej strony reprezentującego ją „pudełka". Ponieważ dany egzemplarz funkcji `silnia` czasami wywołuje kolejny swój egzemplarz (dla obliczenia wyniku cząstkowego), wypadałoby jakoś je różnicować. Najprostszą metodą jest dokonywanie tego poprzez wartość `x`, która jest dla nas punktem odniesienia używanym przy określaniu aktualnej głębokości rekurencji.

Niebezpieczeństwa rekurencji

Z użyciem rekurencji czasami związane są pewne niedogodności. Dwa klasyczne niebezpieczeństwa prezentują poniższe przykłady.

Ciąg Fibonacciego

Naszym pierwszym zadaniem jest napisanie programu, który liczyłby elementy tzw. ciągu Fibonacciego. Występuje on bardzo często w przyrodzie i jest definiowany następująco:

$$fib(0) = 0,$$
$$fib(1) = 1,$$
$$fib(n) = fib(n-1) + fib(n-2), \text{ gdzie } n \geq 2$$

Elementy tego ciągu stanowią liczby naturalne tworzące ciąg o takiej własności, że kolejny wyraz (z wyjątkiem dwóch pierwszych) jest sumą dwóch poprzednich (tj. *1*, *1*, *2*, *3*, *5*, *8*, *13*, ...). Nazwa

pochodzi od imienia Leonarda z Pizy zwanego Fibonaccim, który pierwszy opublikował ten ciąg[2].

Nawiązanie do zjawisk przyrodniczych nie jest przypadkowe. Wyobraźmy sobie, że hodujemy króliki i mamy gwarantowany wzrost populacji według reguł:

- Zaczynamy od jednej pary.
- Każda samica królika wydaje na świat potomstwo w miesiąc po kopulacji — jednego samca i jedną samicę.
- W miesiąc po urodzeniu królik może przystąpić do reprodukcji.

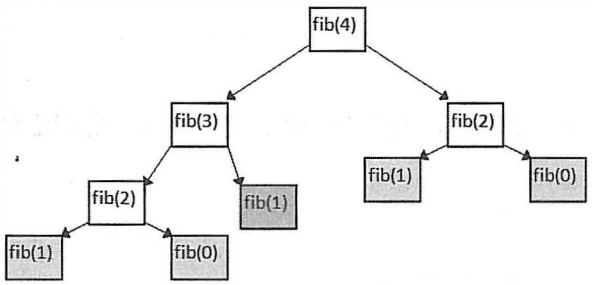
Jak w takiej sytuacji można obliczyć wzrost rozwoju naszej farmy? Przy końcu pierwszego miesiąca możemy się spodziewać pierwszego zapłodnienia w pierwszej parze królików. Pod koniec drugiego miesiąca samica urodzi parę młodych — na farmie będą już dwie pary. W trzecim miesiącu będziemy mieli już trzy pary, gdyż pierwsza samica wyda na świat kolejne potomstwo, a urodzone wcześniej przystąpi do kopulacji... W łatwy sposób można obliczyć, że liczebność w kolejnych miesiącach będzie wynosić *1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89,* ... A liczby te stanowią elementy ciągu Fibonacciego[3]!

Zaprezentowany niżej program jest niemal dokładnym przetłumaczeniem powyższego wzoru i nie powinien stanowić dla nikogo niespodzianki:

Listing *rek3.cpp*

```
unsigned long int fib(int x)
{
if (x<2)
   return x;
else
   return fib(x-1)+fib(x-2);
}
```

Spróbujmy prześledzić dokładnie wywołania rekurencyjne. Nieskomplikowana analiza prowadzi do następującego drzewa (rysunek 2.3):



Rysunek 2.3. *Obliczanie czwartego elementu ciągu Fibonacciego*

Każde zacieniowane wyrażenie stanowi problem elementarny; problem o rozmiarze n≥2 zostaje rozbity na dwa problemy o mniejszym stopniu skomplikowania: `n-1` i `n-2`, proces dekompozycji zatrzymuje się na przypadkach elementarnych.

[2] Mniej znane jest to, iż właśnie Fibonacci przywiózł z Azji do Europy wiedzę na temat nowoczesnej arytmetyki, m.in. liczb ujemnych i pozycyjnego systemu zapisu liczb.

[3] Gwoli ścisłości podam, że istnieje iteracyjna wersja tego wzoru, wykorzystująca fakt istnienia liczby φ (czytaj *fi*). Liczbę tę, zwaną złotym podziałem, otrzymujemy, dzieląc dowolną liczbę ciągu przez liczbę ją poprzedzającą (φ wynosi zatem około 1,61804).

Skąd się jednak wziął pesymistyczny tytuł tego podrozdziału? Przypatrzmy się dokładniej rysunkowi 2.3. Już w pierwszej chwili można dostrzec, że znaczna część obliczeń jest wykonywana więcej niż jeden raz (np. cała gałąź zaczynająca się od `fib(2)` jest wręcz zdublowana!). Funkcja `fib` nie ma żadnej możliwości, aby to zauważyć, w końcu jest to tylko program, który wykonuje to, co mu zlecamy. W rozdziale 9. zostanie omówiona ciekawa technika programowania (tzw. *programowanie dynamiczne*) pozwalająca poradzić sobie z powyższą wadą.

Stack overflow!

Tytuł niniejszego podrozdziału oznacza po polsku „przepełnienie stosu". Jak wykazuje praktyka programowania, pisanie programów podlega regułom raczej świata magii i nieokreśloności niż naszym zachciankom. Ile razy zdarzało się nam zawieszenie się komputera (przez co rozumiemy powszechnie stan, w którym program nie reaguje na nic i trzeba mu zasalutować trzema klawiszami[4]) na skutek działania naszego programu? Zdarza się to nawet najbardziej uważnym programistom i stanowi raczej nieodłączny element pracy programistycznej.

Istnieje kilka typowych przyczyn zawieszania się programów:

- zachwianie równowagi systemu operacyjnego przez „nielegalne" użycie jego zasobów;
- „nieskończone" pętle;
- brak pamięci;
- nieprawidłowe lub niejasne określenie warunków zakończenia programu;
- błąd programowania (np. zbyt wolno wykonujący się algorytm).

Programy rekurencyjne są zazwyczaj dość pamięciożerne: z każdym wywołaniem rekurencyjnym wiąże się konieczność zachowania pewnych informacji[5] niezbędnych do odtworzenia stanu sprzed wywołania, a to zawsze kosztuje trochę cennych bajtów pamięci. Spotyka się programy rekurencyjne, dla których określenie maksymalnego poziomu zagłębienia rekurencji podczas ich wykonywania jest dość łatwe. Analizując program obliczający `3!`, widzimy od razu, że wywoła sam siebie tylko *3* razy; w przypadku funkcji `fib` szybka diagnoza nie przynosi już tak kompletnej informacji.

Przybliżone szacunki nie zawsze należą do najprostszych. Dowodzi tego chyba najlepiej funkcja MacCarthy'ego, zaprezentowana poniżej:


Listing

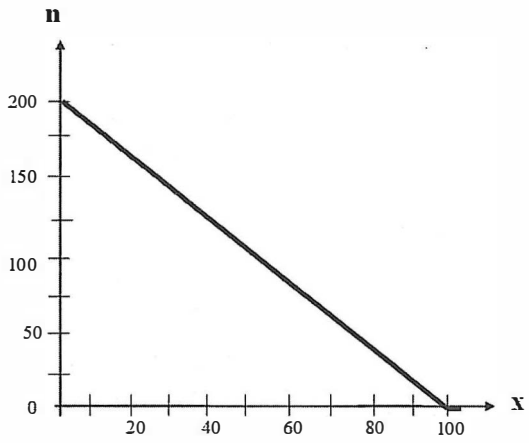
rek4.cpp

```
unsigned long int MacCarthy(int x)
{
if (x>100)
   return (x-10);
else
   return MacCarthy(MacCarthy(x+11));
}
```

Już na pierwszy rzut oka widać, że funkcja jest jakaś „dziwna". Kto potrafi powiedzieć w przybliżeniu, jak się przedstawia jej liczba wywołań w zależności od parametru `x` podanego w wywołaniu? Chyba niewielu byłoby w stanie od razu powiedzieć, że zależność ta ma postać przedstawioną na wykresie z rysunku 2.4.

[4] *Ctrl+Alt+Del* w systemie DOS lub Windows, instrukcja `kill` w systemie Unix, etc.

[5] W szczegóły wnikać nie będziemy, gdyż tematyka ta nie ma dla nas większego znaczenia w tym miejscu.



Rysunek 2.4. *Liczba wywołań funkcji MacCarthy'ego w funkcji parametru wywołania*

Bardziej opisowo można zauważyć, że dla wszystkich x większych od 100 funkcja wykona się tylko raz. Uruchom i skompiluj program przykładowy — pozwala on na podawanie parametru wejściowego, jako swój wynik wyświetla wartość funkcji MacCarthy'ego oraz liczbę jej wywołań.

Wyniki działania tej funkcji nie są wcale takie oczywiste, prawda?

Ćwiczenie 2.1.

Zbadaj funkcję MacCarthy'ego w większym przedziale liczbowym niż ten na rysunku. Jakich niebezpieczeństw można się doszukać?

Ćwiczenie 2.2.

Twoim zadaniem będzie narysowanie drzewa wywołań rekurencyjnych funkcji MacCarthy'ego dla granicznej liczby $x = 100$ i dla wartości $x = 99$. Następnie sprawdź wyniki, kompilując i uruchamiając program przykładowy.

Pułapek ciąg dalszy

Jakby nie dość było negatywnych stron programów rekurencyjnych, należy jeszcze dorzucić te, które nie wynikają z samej natury rekurencji, lecz raczej z błędów programisty. Być może warto w tym miejscu podkreślić, iż omawianie „ciemnych stron" rekurencji nie ma na celu zniechęcenia Czytelnika do jej stosowania! Chodzi raczej o wskazanie typowych pułapek i sposobów ich omijania — a te ostatecznie istnieją zawsze (pod warunkiem, że wiemy, CO omijać). Zapraszam zatem do lektury następnych paragrafów.

Stąd do wieczności

W wielu funkcjach rekurencyjnych, pozornie dobrze skonstruowanych, może z łatwością ukryć się błąd polegający na sprowokowaniu nieskończonej liczby wywołań rekurencyjnych. Taki właśnie zwodniczy przykład jest przedstawiony na następnej stronie (*std.cpp*).

Próba uruchomienia programu *std.cpp* dla wartości m=2 doprowadzi, w zależności od systemu operacyjnego, do komunikatu *Stack overflow* lub *Segmentation fault*. W skrócie: zabrakło pamięci i dzieje się tak już po kilku sekundach działania programu!


Listing

std.cpp

```
int StadDoWiecznosci(int n)
{
if (n==1)
   return 1;
else
   if ( (n%2)== 0  ) //n parzyste
      return StadDoWiecznosci(n-2)*n;
   else
      return StadDoWiecznosci(n-1)*n;
}
```

Gdzie jest umiejscowiony problem? Patrząc na ten program, trudno dopatrzyć się szczególnych niebezpieczeństw. W istocie definicja rekurencyjna wydaje się poprawna: mamy przypadek elementarny kończący łańcuch wywołań, problem o rozmiarze n jest upraszczany do problemu o rozmiarze n-1 lub n-2. Pułapka tkwi właśnie w tej naiwnej wierze, że proces upraszczania doprowadzi do przypadku elementarnego (czyli do n=1)! Po dokładniejszej analizie można wszakże zauważyć, że dla n≥2 wszystkie wywołania rekurencyjne kończą się *parzystą* wartością n. Implikuje to, iż w końcu dojdziemy do przypadku n=2, który zostanie zredukowany do n=0, który zostanie zredukowany do n=-2, który... Można tak kontynuować w nieskończoność, nigdzie po drodze nie ma żadnego przypadku elementarnego!

Wniosek nasuwa się sam: należy zwracać baczną uwagę na to, czy dla wartości parametrów wejściowych należących do dziedziny wartości, które mogą być użyte, rekurencja się kiedyś kończy.

Definicja poprawna, ale...

Rozpatrywany poprzednio przykład służył do zilustrowania problemów związanych ze zbieżnością procesu rekurencyjnego. Wydaje się, że dysponując poprawną definicją rekurencyjną dostarczoną przez matematyka, możemy już być spokojni o to, że analogiczny program rekurencyjny także będzie poprawny (tzn. nie zapętli się, będzie dostarczać oczekiwane wyniki, etc.). Niestety jest to wiara dość naiwna i niczym nieuzasadniona. Matematyk bowiem jest w stanie zrobić wszystko, co związane z jego dziedziną: określić dziedziny wartości funkcji, udowodnić, że ona się zakończy, wreszcie podać złożoność obliczeniową — jednej jednak rzeczy nie będzie mógł sprawdzić: jak rzeczywisty kompilator wykona tę funkcję! Mimo że większość kompilatorów działa podobnie, to zdarzają się pomiędzy nimi drobne różnice, które powodują, że identyczne programy będą dawać różne wyniki. Nasz kolejny przykład będzie dotyczył właśnie takiego przypadku.

Proszę spojrzeć na następującą funkcję:

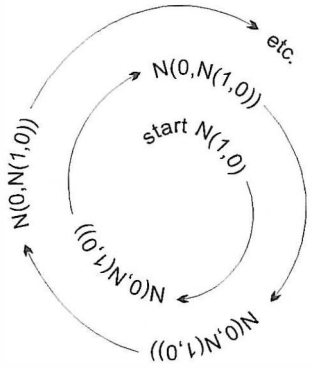
```
int N(int n, int p)
{
if (n==0)
   return 1;
else
   return N(n-1,N(n-p,p))
}
```

Można przeprowadzić dowód matematyczny[6], że powyższa definicja jest poprawna w tym sensie, iż dla dowolnych wartości n≥0 i p≥0 jej wynik jest określony i wynosi 1. Dowód ten opiera się na założeniu, że wartość argumentu wywołania funkcji jest obliczana tylko wtedy, gdy jest naprawdę niezbędna (co wydaje się dość logiczne). Jak się to zaś ma do typowego kompilatora C++?

[6] Patrz **[Kro89]**.

Otóż regułą jest, iż wszystkie parametry funkcji rekurencyjnej są obliczane jako pierwsze, a następnie dokonywane jest wywołanie samej funkcji. (Taki sposób pracy jest zwany *wywołaniem przez wartość*).

Problem może zaistnieć wówczas, gdy w wywołaniu funkcji spróbujemy umieścić ją samą. Zobaczmy, jak to się odbędzie w przypadku naszej funkcji, np. dla `N(1,0)` (patrz rysunek 2.5).



Rysunek 2.5. *Nieskończony ciąg wywołań rekurencyjnych*

Zapętlenie jest spowodowane próbą obliczenia parametru `p`, tymczasem to drugie wywołanie jest w ogóle niepotrzebne do zakończenia funkcji! Istnieje w niej bowiem warunek obejmujący przypadek elementarny: jeśli `n=0`, to zwróć 1. Niestety kompilator o tym nie wie i usiłuje obliczyć ten drugi parametr, powodując zapętlenie programu.

Przykład omówiony w niniejszym paragrafie należy traktować jako swoistą ciekawostkę, niemniej warto go zapamiętać ze względów czysto edukacyjnych.

Typy programów rekurencyjnych

Na podstawie lektury poprzednich paragrafów Czytelnik mógłby wyciągnąć kilka ogólnych wniosków na temat programów używających technik rekurencyjnych: typowo zachłanne w dysponowaniu pamięcią komputera, niekiedy zawieszają system operacyjny... Na szczęście jest to błędne wrażenie! Programy rekurencyjne mają jedną olbrzymią zaletę: są łatwe do zrozumienia i zazwyczaj zajmują mało miejsca, jeśli rozpatrujemy liczbę wierszy kodu użytego do ich realizacji. Z tym ostatnim jest ściśle związana względna łatwość odnajdywania ewentualnych błędów. Wróćmy jednak do tematu.

Zauważyliśmy wspólnie, że program rekurencyjny może być pamięciochłonny i wykonywać się dość wolno. Pytanie brzmi: czy istnieją jakieś techniki programowania pozwalające usunąć (lub co najmniej zredukować) powyższe wady z programu rekurencyjnego? Odpowiedź jest na szczęście pozytywna! Otóż pewna klasa problemów natury rekurencyjnej da się zrealizować na dwa sposoby, dające dokładnie taki sam efekt końcowy, ale różniące się nieco realizacją praktyczną. Podzielmy metody rekurencyjne, tytułem uproszczenia, na dwa podstawowe typy:

- rekurencja „naturalna",
- rekurencja „z parametrem dodatkowym"[7].

Typ pierwszy mieliśmy okazję zobaczyć podczas analizy dotychczasowych przykładów, teraz zapoznamy się z drugim.

[7] Pozostaniemy na moment przy tej nieprecyzyjnej nazwie; ten typ rekurencji będzie omawiany jeszcze w rozdziale 6. — w innym jednakże kontekście.

Rozważmy raz jeszcze przykład funkcji obliczającej silnię. Do tej pory znaliśmy ją w postaci:


Listing

rek5.cpp

```
unsigned long int silnia1(unsigned long int x)
{
if (x==0)
   return 1;
else
   return x*silnia1(x-1);
}
```

Nie jest to bynajmniej jedyna możliwa realizacja funkcji obliczającej silnię. Spójrzmy dla przykładu na następującą wersję:

```
unsigned long int silnia2(unsigned long int x,
                          unsigned long int tmp=1)
{
if (x==0)
   return tmp;
else
   return silnia2(x-1, x*tmp);
}
```

W pierwszym momencie działanie tej drugiej funkcji nie jest być może oczywiste, ale wystarczy wziąć kartkę i ołówek, aby przekonać się na kilku przykładach, że wykonuje ona swoje zadanie tak samo dobrze, jak wersja poprzednia. Osobom nieznającym dobrze C++ należy się niewątpliwie wyjaśnienie konstrukcji funkcji `silnia2`. Otóż dowolna funkcja w C++ może posiadać parametry domyślne.

Dzięki temu funkcja o przykładowym nagłówku:

```
int FunDom(int a, int k=1)
```

może zostać wywołana na dwa sposoby:

- Poprzez określenie wartości drugiego parametru, np. `FunDom(12,5)`: w tym przypadku `k` przyjmuje wartość `5`.
- Bez określania wartości drugiego parametru, np. `FunDom(12)`: `k` przyjmuje wtedy wartość domyślną równą tej podanej w nagłówku, czyli `1`.

Ta użyteczna cecha języka C++ wykorzystana została w drugiej wersji funkcji do obliczania silni. Jakie jednak istotne względy przemawiają za używaniem tej osobliwej z pozoru metody programowania? Argumentem nie jest tu wzrost czytelności programu, bowiem już na pierwszy rzut oka `silnia2` jest o wiele bardziej zagmatwana niż `silnia1`!

Istotna zaleta rekurencji „z parametrem dodatkowym" jest ukryta w sposobie wykonywania programu. Wyobraźmy sobie, że program rekurencyjny „bez parametru dodatkowego" wywołał sam siebie 10-krotnie, aby obliczyć dany wynik. Oznacza to, że wynik cząstkowy z dziesiątego, najgłębszego poziomu rekurencji będzie musiał być przekazany przez kolejne dziesięć poziomów do góry, do swojego pierwszego egzemplarza.

Jednocześnie z każdym „zamrożonym" poziomem, który czeka na nadejście wyniku cząstkowego, wiąże się pewna ilość pamięci, która służy do odtworzenia m.in. wartości zmiennych tego poziomu (tzw. kontekst). Co więcej, odtwarzanie kontekstu już samo w sobie zajmuje cenny czas procesora, który mógłby być wykorzystany np. na inne obliczenia.

Czytelnik już zapewne się domyśla, że program rekurencyjny „z parametrem dodatkowym" robi to wszystko nieco wydajniej. Ponieważ parametr dodatkowy służy do przekazywania elementów wyniku końcowego, dysponując nim, program nie ma potrzeby przekazywania wyniku obliczeń do góry, piętro po piętrze. Po prostu w momencie, w którym program „stwierdzi", że

obliczenia zostały zakończone, procedura wywołująca zostanie o tym poinformowana wprost z ostatniego aktywnego poziomu rekurencji. Co za tym wszystkim idzie, nie ma absolutnie żadnej potrzeby zachowywania kontekstu poszczególnych poziomów pośrednich, liczy się tylko ostatni aktywny poziom, który dostarczy wynik, i basta!

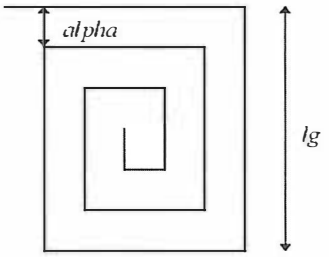
Myślenie rekurencyjne

Pomimo oczywistych przykładów na to, że rekurencja jest dla człowieka czymś jak najbardziej naturalnym, niektórzy mają pewne trudności z używaniem jej podczas programowania. Nieumiejętność wyczucia istoty tej techniki programowania może wynikać z braku dobrych i poglądowych przykładów na jej wykorzystanie. Idąc za tym stwierdzeniem, postanowiłem wybrać kilka prostych programów rekurencyjnych, generujących znane motywy graficzne — ich dobre zrozumienie będzie wystarczającym testem na oszacowanie swoich zdolności myślenia rekurencyjnego (ale nawet wówczas wykonanie zadań zamieszczonych pod koniec rozdziału będzie jak najbardziej wskazane).

Przykład 1.: Spirala

Zastanówmy się, jak można narysować spiralę rekurencyjnie jednym pociągnięciem kreski — rysunek 2.6.

Rysunek 2.6. *Spirala narysowana rekurencyjnie*



Parametrami programu są :

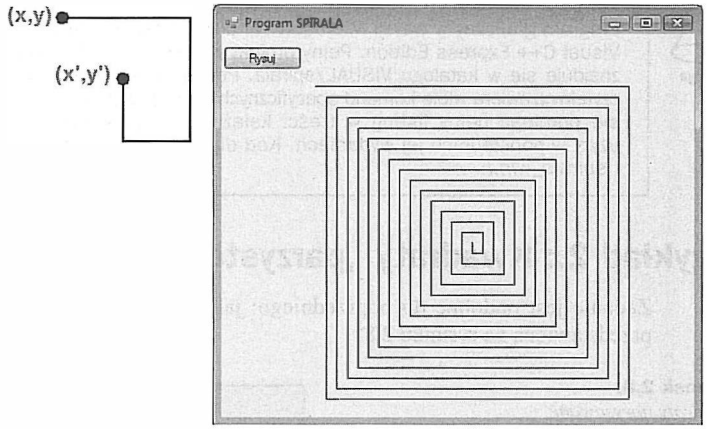
- odstęp pomiędzy liniami równoległymi: `alpha`;
- długość boku rysowanego w pierwszej kolejności: `lg`.

Algorytm iteracyjny byłby również nieskomplikowany (zwykła pętla), ale załóżmy, że zapomnimy chwilowo o jego istnieniu i wykonamy to samo rekurencyjnie. Istota rekurencji polega głównie na znalezieniu właściwej dekompozycji problemu. Tutaj jest ona przedstawiona na rysunku i w związku z tym ewentualne przetłumaczenie jej na program w C++ powinno być znacznie ułatwione.

Rekurencyjność naszego zadania jest oczywista, bowiem program wynikowy zajmuje się powtarzaniem głównie tych samych czynności (rysuje linie poziome i pionowe, jednakże o różnej długości). Naszym zadaniem będzie odszukanie schematu rekurencyjnego i warunków zakończenia procesu wywołań rekurencyjnych.

Jak rozwiązać to zadanie? Wpierw przybliżmy się nieco do „rzeczywistości ekranowej" i wybierzmy jako punkt startowy pewną parę `(x,y)`. Idea rozwiązania polega na narysowaniu 4 odcinków zewnętrznych spirali i dotarciu do punktu `(x',y')`. W tym nowym punkcie startowym możemy już wywołać rekurencyjnie procedurę rysowania, obarczoną oczywiście pewnymi warunkami gwarantującymi jej poprawne zakończenie.

Elementarny przypadek rozwiązania prezentuje rysunek 2.7.



Rysunek 2.7. *Spirala narysowana rekurencyjnie — szkic rozwiązania + wynik*

Jedna z kilku możliwych wersji programu, który realizuje to, co zostało wyżej opisane, jest przedstawiona poniżej. Program używa następujących, umownych instrukcji graficznych[8]:

- `lineto(x,y)` — kreśli odcinek prostej od pozycji bieżącej do punktu `(x, y)`
- `moveto(x,y)` — przesuwa kursor graficzny do punktu `(x, y)`
- `getmaxx()` — zwraca maksymalną współrzędną poziomą (zależy od rozdzielczości trybu graficznego)
- `getmaxy()` — zwraca maksymalną współrzędną pionową (j. w.)
- `getx()` — zwraca aktualną współrzędną poziomą
- `gety()` — zwraca aktualną współrzędną pionową

Listing

spirala.cpp

```
const double alpha=10;
void spirala(double lg,double x,double y)
{
if (lg>0)
   {
   lineto(x+lg,    y);
   lineto(x+lg,    y+lg);
   lineto(x+alpha, y+lg);
   lineto(x+alpha, y+alpha);
   spirala(lg-2*alpha, x+alpha, y+alpha);
   }
}
int main()
{
// tu zainicjuj tryb graficzny
moveto(90,50);
spirala(getmaxx()/2,getx(),gety());
getch();  // poczekaj na naciśnięcie klawisza
          // tu zamknij tryb graficzny
}
```

[8] Są one zgodne z bibliotekami kompilatora Borland dla systemu DOS (pliki znajdziesz w folderze INNE w archiwum ZIP na ftp).

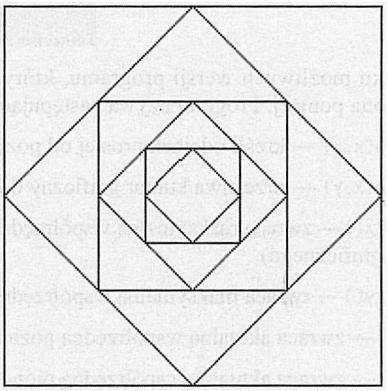

Uwaga

Program graficzny w wersji Microsoft Windows został przygotowany w kompilatorze Microsoft Visual C++ Express Edition. Pełny projekt przygotowany dla środowiska graficznego Windows znajduje się w katalogu VISUAL/spirala. Ponieważ kod rozwiązania dla Windows jest niezbyt czytelny (zawiera wiele komend specyficznych dla programowania w tym systemie), zdecydowałem się pominąć pełny listing w treści książki, pozostawiając w niej oryginalny, zwięzły kod użyty w poprzednich jej wydaniach. Kod dla Windows znajdziesz w plikach: *spirala_win.cpp* i *spirala_win.h*.

Przykład 2.: Kwadraty „parzyste"

Zadanie jest podobne do poprzedniego: jak jednym pociągnięciem kreski narysować figurę przedstawioną na rysunku 2.8?

Rysunek 2.8.
Kwadraty narysowane rekurencyjnie (n = 3)



Przypadkiem elementarnym będzie tutaj narysowanie jednej pary kwadratów (wewnętrzny obrócony w stosunku do zewnętrznego).

To zadanie jest nawet prostsze niż poprzednie, sztuka polega jedynie na wyborze właściwego miejsca wywołania rekurencyjnego[9]:


Listing

kwadraty.cpp

```
void kwadraty(int n, double lg, double x, double y)
{
// n — parzysta liczba kwadratów
// x, y — współrzędne punktu startowego
if (n>0)
   {
   lineto(x+lg,      y);
   lineto(x+lg,   y+lg);
   lineto(x,      y+lg);
   lineto(x,    y+lg/2);
   lineto(x+lg/2, y+lg);
   lineto(x+lg,  y+lg/2);
   lineto(x+lg/2,     y);
   lineto(x+lg/4,y+lg/4);
   kwadraty(n-1,lg/2, x+lg/4, y+lg/4);
   lineto(x,y+lg/2);
```

[9] Ten kod został zachowany z przyczyn historycznych, jest on zgodny z kompilatorem Borland i systemem DOS, znajdziesz go w folderze INNE w archiwum ZIP.

```
    lineto(x,y);
    }
}

int main()
{
// inicjuj tryb graficzny
moveto(90,50);
kwadraty(5, getmaxx()/2, getx(), gety());
getch();
// zamknij tryb graficzny
}
```


Uwaga

Program graficzny w wersji Microsoft Windows został przygotowany w kompilatorze Microsoft Visual C++ Express Edition, projekt znajduje się w katalogu VISUAL/kwadraty. Podobnie jak w poprzednim przykładzie, także i tu zdecydowałem się pominąć pełny listing w treści książki, pozostawiając w niej oryginalny, zwięzły kod użyty w poprzednich jej wydaniach. Kod dla Windows znajdziesz w plikach: *kwadraty_win.cpp* i *kwadraty_win.h*.

Uwagi praktyczne na temat technik rekurencyjnych

Szczegółowy wgląd w techniki rekurencyjne uświadomił nam, że niosą one ze sobą zarówno plusy, jak i minusy. Zasadniczą zaletą jest czytelność i naturalność zapisu algorytmów w formie rekursywnej — szczególnie gdy zarówno problem, jak i struktury danych z nim związane są wyrażone w postaci rekurencyjnej. Procedury rekurencyjne są zazwyczaj klarowne i krótkie, dzięki czemu dość łatwo jest wykryć w nich ewentualne błędy. Jak jednak ocenić algorytm rekurencyjny? Otóż nawet bez używania skomplikowanego aparatu matematycznego dość bezpieczną techniką oceny jakości programu rekurencyjnego jest ocena, czy warunek zakończenia jest poprawny (przypadek elementarny) i czy dekompozycja rekurencyjna prowadzi do *zmniejszenia* rozmiaru problemu. Wszelkie dziwaczne konstrukcje (patrz funkcja MacCarthy'ego) stanowią zazwyczaj bardziej ciekawostki akademickie niż przykłady poprawnych procedur.

Dużą wadą wielu algorytmów rekurencyjnych jest pamięciożerność: wielokrotne wywołania rekurencyjne mogą łatwo zablokować całą dostępną pamięć! Problemem jest tu jednak nie fakt zajętości pamięci, ale typowa niemożność łatwego jej oszacowania przez konkretny algorytm rekurencyjny. Można do tego wykorzystać metody służące do analizy efektywności algorytmów (patrz rozdział 3.), jednakże jest to dość nużące obliczeniowo, a czasami nawet po prostu niemożliwe.

W podrozdziale „Typy programów rekurencyjnych" poznaliśmy metodę na ominięcie kłopotów z pamięcią poprzez stosowanie rekurencji „z parametrem dodatkowym". Nie wszystkie jednak problemy dadzą się rozwiązać w ten sposób, ponadto programy używające tej metody tracą odrobinę na czytelności. No cóż, nie ma róży bez kolców...

Kiedy nie należy używać rekurencji? Ostateczna decyzja należy zawsze do programisty, tym niemniej istnieją sytuacje, gdy ów dylemat jest dość łatwy do rozstrzygnięcia. Nie powinniśmy używać rozwiązań rekurencyjnych, gdy:

- W miejsce algorytmu rekurencyjnego można podać czytelny lub szybki odpowiednik iteracyjny.
- Algorytm rekurencyjny jest *niestabilny* (np. dla pewnych wartości danych wejściowych może się zapętlić lub dawać dziwne wyniki — często wynika to ze specyfiki kompilatora lub cech platformy sprzętowej).

Ostatnią uwagę podaję już raczej, by dopełnić formalności. Otóż w literaturze można czasem napotkać rozważania na temat niekorzystnych cech tzw. *rekurencji skrośnej*: podprogram *A* wywołuje podprogram *B*, który wywołuje z kolei podprogram *A*. Nie podałem celowo przykładu takiego „dziwoląga", gdyż nadmiar złych przykładów może być szkodliwy. Praktyczny wniosek, który możemy wysnuć, analizując osobliwe programy rekurencyjne, pełne nieprawdopodobnych konstrukcji, jest jeden: UNIKAJMY ICH, jeśli tylko nie jesteśmy całkowicie pewni poprawności programu, a intuicja nam podpowiada, że w danej procedurze jest coś, co może spowodować kłopoty.

Korzystając z katalogów algorytmów, formalizując programowanie, etc., można bardzo łatwo zapomnieć, że wiele pięknych i eleganckich metod powstało samo z siebie — jako przebłysk geniuszu, intuicji, sztuki... A może i my moglibyśmy dołożyć nasze „co nieco" do tej kolekcji? Proponuję ocenić własne siły poprzez rozwiązywanie zadań, które odpowiedzą w sposób najbardziej obiektywny, czy rozumiemy rekurencję jako metodę programowania.

Zadania

Wybór reprezentatywnego dla rekurencji zestawu zadań wcale nie był łatwy dla autora tej książki — dziedzina ta jest bardzo rozległa i w zasadzie wszystko w niej jest w jakiś sposób interesujące. Ostatecznie, co zwykłem podkreślać, zadecydowały względy praktyczne i prostota.

Zadanie 1.

Załóżmy, że chcemy odwrócić w sposób rekurencyjny tablicę liczb całkowitych. Proszę zaproponować algorytm z użyciem rekurencji „naturalnej", który wykona to zadanie.

Zadanie 2.

Powróćmy do problemu poszukiwania pewnej zadanej liczby `x` w tablicy, tym razem jednak posortowanej od wartości minimalnych do maksymalnych. Metoda poszukiwania, bardzo znana i efektywna (tzw. *przeszukiwanie binarne*), polega na następującej obserwacji:

- Podzielmy tablicę o rozmiarze `n` na połowę:
  - `t[0]. t[1]... t[n/2-1]. t[n/2]. t[n/2+1]. .... t[n-1].`
  - Jeśli `x = t[n/2]`, to element `x` został znaleziony[10].
  - Jeśli `x < t[n/2]`, to element `x` być może znajduje się w lewej połowie tablicy; analizuj ją.
  - Jeśli `x > t[n/2]`, to element `x` być może znajduje się w prawej połowie tablicy; analizuj ją.

Wyrażenie *być może* daje nam furtkę bezpieczeństwa w przypadku niepowodzenia poszukiwania. Zadanie polega na napisaniu dwóch wersji funkcji realizującej powyższy algorytm, jednej używającej rekurencji naturalnej i drugiej — dla odmiany — nierekurencyjnej.

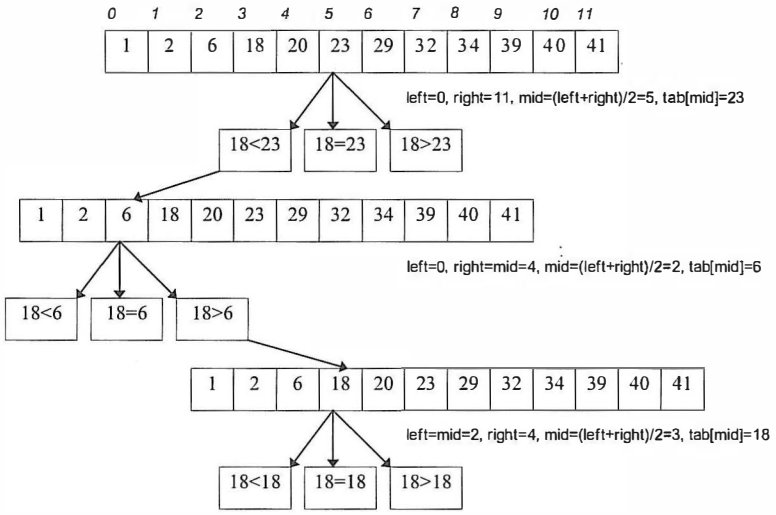
Rysunek 2.9 prezentuje działanie algorytmu dla następujących danych:

- 12-elementowa tablica zawiera liczby: 1, 2, 6, 18, 20, 23, 29, 32, 34, 39, 40, 41.
- Szukamy liczby 18.

W celu dokładniejszego przeanalizowania algorytmu posłużymy się kilkoma zmiennymi pomocniczymi:

- `left` — indeks tablicy ograniczający obserwowany obszar tablicy od lewej strony;
- `right` — indeks tablicy ograniczający obserwowany obszar tablicy od prawej strony;
- `mid` — indeks elementu środkowego obserwowanego aktualnie obszaru tablicy.

[10] W C++ dzielenie całkowite obcina wynik do liczby całkowitej (odpowiednik *div* w Pascalu).



Rysunek 2.9. *Przeszukiwanie binarne na przykładzie*

Na rysunku 2.9 przedstawione jest działanie algorytmu oraz wartości zmiennych `left`, `right` i `mid` podczas każdego ważniejszego etapu. Poszukiwanie zakończyło się pomyślnie już po trzech etapach[11]. Warto zauważyć, że to samo zadanie, rozwiązywane za pomocą przeglądania od lewej do prawej elementów tablicy, zostałoby ukończone dopiero po 4 etapach. Być może otrzymany zysk nie oszałamia, proszę sobie jednak wyobrazić, co by było, gdyby tablica miała rozmiar kilkanaście razy większy niż ten użyty w przykładzie?! Proszę napisać funkcję, która realizuje poszukiwanie binarne w sposób rekurencyjny.

Zadanie 3.

Napisz funkcję, która — otrzymując liczbę całkowitą dodatnią — wypisze jej *reprezentację dwójkową*. Należy wykorzystać znany algorytm dzielenia przez podstawę systemu.

Przykładowo zamieńmy liczbę 13 na jej postać binarną:

```
13 : 2 = 6 + 1
6  : 2 = 3 + 0
3  : 2 = 1 + 1
1  : 2 = 0 + 1
     ↑
Koniec algorytmu!
```

Problem polega na tym, że otrzymaliśmy prawidłowy wynik, ale... od tyłu! Algorytm dał nam `1011`, natomiast prawidłową postacią jest `1101`. Dopiero w tym miejscu zaczyna się właściwe zadanie:

Pytanie 1

Jak wykorzystać rekurencję do odwrócenia kolejności wypisywanych cyfr?

Pytanie 2

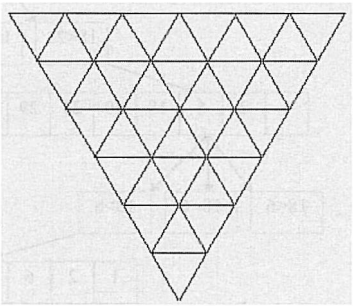
Czy istnieje nieskomplikowane i eleganckie rozwiązanie tego zadania, wykorzystujące rekurencję z „parametrem dodatkowym"?

[11] Za „etap" będziemy tu uważali moment testowania, czy dana jest tą liczba, której poszukujemy.

Zadanie 4.

Spróbuj napisać funkcję, która rekurencyjnie wyrysuje „dywanik" przedstawiony na rysunku 2.10:

Rysunek 2.10. *Trójkąty narysowane rekurencyjnie*



Zadanie 5.

W ramach relaksu zaprogramuj rekurencyjnie algorytm Euklidesa — algorytm poszukiwania największego wspólnego dzielnika dwóch liczb (NWD lub GCD po angielsku — *greatest common divisor*). Największy wspólny dzielnik, NWD, dla dwóch liczb naturalnych *a*, *b*: taka największa liczba naturalna *c*, że *a* oraz *b* dzieli się bez reszty przez *c*.

Euklides zauważył, że jeśli od większej liczby odejmiemy mniejszą, to ta mniejsza liczba i otrzymana różnica będą miały taki sam największy wspólny dzielnik, jak pierwotne liczby. Gdy przy kolejnym odejmowaniu otrzymamy parę takich samych liczb, to znaleźliśmy NWD.

Zadanie 6.

Napisz programu obliczający silnię bez użycia rekurencji.

Rozwiązania i wskazówki do zadań

Zadanie 1.

Idea rozwiązania jest następująca:

- Zamieńmy miejscami elementy skrajne tablicy (*przypadek elementarny*).
- Odwróćmy pozostałą część tablicy (*wywołanie rekurencyjne*).

Odpowiadający temu rozumowaniu program przedstawia się następująco:

Listing

rev_tab.cpp

```
// zamiana zmiennych:
void swap(int& a, int& b)
{
int temp=a;
a=b;
b=temp;
}


void odwroc(int *tab, int left, int right)
{
```

```
if(left<right)
   {
   swap(tab[left],tab[right]); // zamieniamy elementy skrajne
   odwroc(tab,left+1,right-1); // odwracamy resztę
   }
}
int main()
{
int tabl[8]={1,2,3,4,5,6,7,8};
for(int i=0;i<8;i++)
   cout << tabl[i] << "  "; cout << endl;
odwroc(tabl,0,7);    // przykładowe wywołanie i weryfikacja:
for(int i=0;i<8;i++)
   cout << tabl[i] << "  "; cout << endl;
}
```

Zadanie 2.

Poniżej przedstawiona jest wyłącznie wersja rekurencyjna programu. Jestem przekonany, że Czytelnik odkryje bez trudu analogiczne rozwiązanie iteracyjne[12]:

binary_s.cpp

```
int szukaj_rec(int* tab, int x, int left, int right)
{
if(left>right)
   return -1; // element nieznaleziony
else
   {
   int mid=(left+right)/2;
   if(tab[mid]==x)
      return mid; // element został znaleziony!
   else
      if(x<tab[mid])
         return szukaj_rec(tab,x,left,mid-1);
      Else
         return szukaj_rec(tab,x,mid+1,right);
      }
}
```

Zadanie 3.

Program nie należy do zbyt skomplikowanych, choć wcale nie jest trywialny. Zastanówmy się, jak zmusić algorytm do przedstawienia wyniku w postaci normalnej, tzn. od lewej do prawej. W tym celu przeanalizujmy raz jeszcze działanie algorytmu bazującego na dzieleniu przez podstawę systemu liczbowego (tutaj 2). Liczba `x` jest dzielona przez dwa, co daje nam liczbę `[x div 2]` plus reszta. Owa reszta to oczywiście `[x mod 2]` i jest to jednocześnie *ostatnia* cyfra reprezentacji binarnej, którą chcemy otrzymać.

Czy jest jakiś sposób, aby odwrócić kolejność wyprowadzania cyfr dwójkowych, korzystając ciągle z tego prostego algorytmu? Otóż tak, pod warunkiem że spojrzymy nań nieco inaczej. Popatrzmy, jak symbolicznie można rozpisać tworzenie reprezentacji dwójkowej pewnej liczby `x`, używając już właściwych dla C++ operatorów:

$$[x]_2 = [x\%2] * \left\lfloor \frac{x}{2} \right\rfloor_2$$

[12] Lub zajrzy do rozdziału 7.

Zapis ten sugeruje już, jak można rekurencyjnie przedstawić ten algorytm:

$$[x]_2 = przypadek\ elementarny * wywołanie\ rekurencyjne$$

Jeśli w powyższym algorytmie każemy komputerowi najpierw wypisać liczbę *[x/2]* dwójkowo, a dopiero potem *[x %2]* (które to wyrażenie przybiera dwie wartości: *0* lub *1*), to wynik pojawi się na ekranie w postaci normalnej, a nie odwrócony, jak poprzednio.

Warto zapamiętać tę sztuczkę, może być ona pomocna w wielu innych programach.

post_2.cpp

```
void post_dw(unsigned long int n)
{
if(n!=0)
   {
   post_dw(n/2);  // n modulo 2
   cout << n % 2; // reszta z dzielenia przez 2
   }
}
```

Co zaś się tyczy pytania drugiego, to z mojej strony mogę dać na nie odpowiedź: być może. Rozwiązałem ten problem z użyciem rekurencji „z parametrem dodatkowym", ale nie udało mi się znaleźć rozwiązania na tyle eleganckiego, aby było warte prezentacji jako odpowiedź. Być może któryś z Czytelników znajdzie więcej czasu i dokona tego wyczynu? Gorąco zachęcam do prób — być może do niczego nie doprowadzą, ale na pewno nauczą więcej niż lektura gotowych rozwiązań.

Zadanie 4.

Oto jedno z możliwych rozwiązań:

trojkaty.cpp

```
void trojkaty(double n, double lg, double x, double y)
{
// n = liczba podziałów
if (n>0)
   {
   double a=lg/n;
   double h=a*sqrt(3)/2.0;
   lineto(x-a/2.0,y-h);
   trojkaty(n-1,lg-a,x-a/2.0,y-h);
   lineto(x+a/2.0,y-h);
   for(double i=1;i<n;i++)
      {
      lineto(x+(i-1)*a/2.0,y-(i+1)*h);
      lineto(x+(i+1)*a/2.0,y-(i+1)*h);
      }
   lineto( x,y);
   }
}

int main()
{
// inicjuj tryb graficzny
moveto(getmaxx()/2,getmaxy());
trojkaty(6,getmaxx()/2,getx(),gety());
getch();
// zamknij tryb graficzny
}
```

Zadanie 5.

Oto jedno z możliwych rozwiązań z przykładem użycia (w pliku na *ftp* znajdziesz także wariant bardziej klasyczny):

nwd.cpp

```
int nwd(int a, int b)
   {
   if (b==0)
      return a;
   else
      nwd2(b, a % b); //modulo
}
int main()
{
cout << " nwd(12,3)  =" << nwd(12,3)  << endl;
cout << " nwd(24,30) =" << nwd(24,30) << endl;
cout << " nwd(5,7)   =" << nwd(5,7)   << endl;
cout << " nwd(54,69) =" << nwd(54,69) << endl;
}
```

Zadanie 6.

Funkcja silnia jest tak naprawdę bardzo prosta w realizacji, gdy zauważymy, że jest to seria sekwencyjnych operacji mnożenia (czytaj od prawej do lewej strony):

```
                                                    0!=1
                                          1!=1*0!     ↵
                                2!=2*1!     ↵
                      3!=3*2!     ↵
          ...     ↵
```

Jak łatwo zauważyć, aby obliczyć kolejną wartość silni, wystarczy dysponować skumulowanym wynikiem poprzednio wykonanych operacji mnożenia, co można zrealizować przy pomocy jednej pętli w C++. Rozwiązanie znajdziesz w programach dołączonych do książki, ale postaraj się do niego dojść samodzielnie, bez zaglądania do gotowego kodu.