Wojciech Myszka

Laboratorium 7: Funkcje — rekurencja

2016-05-07 09:03:56 +0200

1. Wprowadzenie

Przypadek gdy funkcja wywołuje samą siebie nazywa się rekurencją. W wielu przypadkach pozwala ona dosyć łatwo tworzyć nowe algorytmy. Nie jest, natomiast, zbyt dobrą metodą ich programowania.

1.1. Silnia

Rekurencja używana jest również dosyć często podczas definiowania nowych pojęć (w matematyce czy logice). Najlepszym (najbardziej znanym?) przykładem jest definicja funkcji (symbolu) silni:

$$n! = \begin{cases} 0 & \text{jeżeli } n = 0 \\ n \times (n-1)! & \text{w przeciwnym razie.} \end{cases} \tag{1}$$

Patrząc na powyższy wzór można zacząć tworzyć funkcję wyliczającą silnię w języku C. *Przy czym wzór „czytamy" tak: Jeżeli* $n = 0$ ***zwróć*** *0, w przeciwnym razie* ***zwróć*** n *przemnożone przez* $n - 1$ *silnia.*

```
1 int silnia(int n)
2 {
3     if ( n == 0 )
4         return 1;
5     else
6         return n * silnia(n - 1);
7 }
```

(Linia 6 kodu jest bardzo istotna: to tam realizowana jest cząstkowe mnożenie liczb i zwracanie wyniku.)

Zwracam uwagę, że wywołanie funkcji silnia w linii 6 kodu oznacza — jak gdyby — „skok” do linii 3, gdzie sprawdzany jest warunek. Zatem kod jest równoważny poniższemu:

```
1  int silnia1(int n)
2  {
3      int s = 1;
4      while ( n > 0 )
5      {
6          s = s * n;
7          n = n - 1;
8      }
9      return s;
10 }
```

1.2. Ciąg Fibonacciego

Kolejny przykład, który łatwo dosyć opisać rekurencyjnie to ciąg Fibonacciego:

$$\mathrm{fib}(n) = \begin{cases} 0 & \text{gdy } n = 0 \\ 1 & \text{gdy } n = 1 \\ \mathrm{fib}(n-1) + \mathrm{fib}(n-2) & \text{dla } n \geqslant 2. \end{cases} \tag{2}$$

Poniższy kod (prezentowany na wykładzie) realizuje rekurencyjną wersję algorytmu.

```
#include <stdio.h>
#include <time.h>
unsigned long int k;
long int fib(int n)
{
    k++;
    if ( n == 0 )
        return 0;
    else if ( n == 1 )
        return 1;
    else
        return fib(n - 1) + fib(n - 2);
}

int main(int argc, char **argv)
{
```

```
    long int n, m;
    time_t a, b;
    for ( n = 0; n < 100; n++ )
    {
        k = 0;
        a = time(NULL);
        m = fib(n);
        b = time(NULL) - a;
        printf("%3ld , %15ld , %15lu %lu\n", n, m, k, b);
    }
    return 0;
}
```

Program używa zmiennej (globalnej) k w celu zliczania ilości wywołań funkcji fib podczas wyliczania wartości każdej kolejnej wartości ciągu. Dodatkowo w sposób przybliżony wyznaczany jest czas poświęcony na wyliczenie każdego wyrazu ciągu.

Uruchamiając ten program łatwo przekonać się, że rekurencyjne obliczanie kolejnych wartości ciągu nie jest najlepszym pomysłem.

1.3. Podsumowanie

Pisanie funkcji rekurencyjnych nie jest łatwym zadaniem. Trudno ogarnąć to co się dzieje gdy po raz kolejny wchodzimy do funkcji. Zasada jest taka: gdy funkcja wywołuje samą siebie wykonywany jest ten sam kod, ale kontekst danych, w którym się to odbywa — jest zupełnie inny. Kod operuje na „świeżym" zestawie (dynamicznych) danych.

Trzeba też pamiętać, żeby zapewnić „wyjście" z ciągu wywołań funkcji. W przypadku rekurencyjnej funkcji silnia polecenie **return** 1; przerywa pętlę wywołań, a polecenie **return** n $*$ silnia(n $-$ 1) inicjuje kolejny „przebieg pętli".

2. „Wieże Hanoi"

Algorytm opisany został na wykładzie z Technologii informacyjnych poświęconym różnym rodzajom algorytmów w dziale rekurencja (slajd 18 i następne). Samo problem wież Hanoii zaczyna się na slajdzie 42. Wariant rekurencyjny opisany jest na slajdzie 69 i następnych. Nierekurencyjna wersja algorytmu podana została na wykładzie z Technologii Informacyjnych dotyczącym złożoności obliczeniowej (slajd 41 i następne).

Idea algorytmu jest następująca:

— Gdy krążek jest tylko jeden — problem nie istnieje (przenosimy go z patyka, na którym się znajduje na patyk docelowy).

— Dla dwu krążków problem jest banalny (najmniejszy krążek przenosimy na roboczy, większy na docelowy i ponownie najmniejszy na docelowy).
— Obejrzyjmy problem dla trzech krążków: można go podzielić na trzy zadania:
  1. Przenosimy dwa „górne" krążki na „trzeci patyczek" (patyk roboczy).
  2. Przeniesienie największego krążka na „patyczek drugi" (docelowy).
  3. Ponowne przeniesienie dwu krążków z „roboczego" na krążek największy (znajdujący się na patyku docelowym)...

A ogólnie, będzia jakoś tak: A — patyk, na którym są wszystkie krążki, B — patyk docelowy, a C — patyk roboczy).

Procedura jest następująca:

1. Przenieśmy z A na C $N-1$ krążków (używamy do tego proceury); B jest patykiem roboczym.
2. Pozostały krążek (największy! — ale z czego to wynika?) przenieśmy z A na B (miejsce docelowe).
3. Do pozostałych $(N-1)$ krążków, które znajdują się na patyku C, zastosujmy powyższy algorytm (patyk B wykorzystujemy jako roboczy, bo na samym spodzie znajduje się krążek największy). Zatem ruch wygląda tak: przenieś $N-1$ krążków z C na B używając A jako patyka roboczego.
4. powyższą procedurę należy powtarzać aż do zakończenia zadania.

Algorytm generuje **jedynie** podpowiedzi w formie $\alpha \rightarrow \beta$ oznaczające „weź krążek z patyka α i przenieś go na patyk β.

2.1. Udoskonalenia

Zaawansowani studenci mogą pokusić się o prostą wizualizację realizacji algorytmu.

Idea jaką można spróbować tu wykorzystać jest następująca:

Niech N to liczba krążków.

1. Tworzymy strukturę danych służącą do „wizualizacji" krążków na patykach. Może to być tablica o trzech kolumnach (tyle jest patyków) oraz tylu wierszach ile jest krążków. Niech tablica nazywa się K: **int** K[N,3];
2. Tablica K w pierwszej kolumnie zawiera liczby od 1 do N (1 — najmniejszy krążek, N — krążek największy).
3. Dodatkowa tablica pomocnicza zawierać będzie informację o liczbie krążków na każdym patyku. Niech nazywa się L: **int** L[3] = { N };.
4. Slajd 71 Zawiera zaerys procedury przenies realizującej rekurencyjną wersję algorytmu. W punktach 1 oraz 2.1 następuje wyprowadzenie podpowiedzi, z którego patyka krążek zdjąć i na który go nałożyć. Normalnie polecenia te komunikują tylko co zrobić należy. Można tu, na przykład, dodać dwie funkcje „zdejmij" i „naloz" które będą dokonywały odpowiednich operacji na naszych danych i procedura drukuj wyyświetlająca aktualną zawartośc tablicy K.

Polecenie a −> b (oznaczające przenieś z a na b) może być zapisane tak:

```
krazek = zdejmij( a, N, K, L );
naloz( krazek, b, N, K, L );
drukuj( N, K );
```

Operacja „zdejmowania" zwraca znajdujący się najwyżej w kolumnie a tablicy K (i dokonuje korekty, zmniejszając o 1 element a tablicy L).
Operacja „nakładania" na wolne miejsce w tablicy K (kolumna b) wstawia krążek (czy dokładniej jego numer) i uaktualnia L[b].
Procedura drukuj w najprostszym wariancie drukuje numerki krążków (zamiast zer wstawiając odstępy). Wersja bardziej zaawansowana dla krążka o numerze 1 drukuje coś takiego:

```
 _
|_|
```

dla krążka o numerze 2

```
 ___
|___|
```

Zatem „pozycja wyjściowa" wyglądać będzie jakoś tak:

```
   |_|
  |___|
 |_____|
```

(Widzę tu miejsce na udoskonalenia :–)

5. Osobna procedura, na każdym kroku, powinna drukować aktualny stan struktury danych w najprostszej formie, na przykład (wizualizacja pierwszych trzech ruchów):

```
2                       1
3  1     3  1  2  3     2
-------  -------  -------
0  1  2  0  1  2  0  1  2
```

(na samym dole są numery patyków, a powyżej kreski ich zawartość; tu po pierwszych, hipotetycznych ruchach.)

3. Dodatkowe zadania do wykonania

Praktycznie każde z programowanych dotychczas zadań (metoda połowienia, wyszukiwanie binarne czy metoda Newtona Raphsona może być zaprogramowana w wariancie rekurencyjnym.

Ambitny i pracowity student może pokusić się o realizację jednego z tych (lub wszystkich) zadań w wariancie rekurencyjnym.

4. Wersja PDF tego dokumentu...

...pod adresem.

Wersja: 50 z drobnymi modyfikacjami! data ostatniej modyfikacji 2016-05-07 09:03:56 +0200