Rekurencja: przykłady zastosowań

Na dzisiejszych zajęciach przeanalizujemy kilka przykładów wykorzystania rekurencji. Na początek przeanalizujemy następujący listing (plik *bisekcja.cpp*):

```
#include<iostream>
using namespace std;

long bisekcja(float szukany, float tablica[], unsigned koniec, unsigned poczatek=0);

int main()
{
// Dla tego algorytmu dane musza byc posortowane (u nas rosnaco) i unikalne
float dane[] = {0, 0.11, 2.5, 3.33, 12.5, 32.5, 45.7}, wartosc;
long wynik, i;

   cout<<"Program wyszukuje podana wartosc z tablicy metoda bisekcji"<<endl;

   cout<<endl<<"Tablica zawiera nastepujace dane:";
   for(i=0; i<7; i++) cout << dane[i] << " ";

   cout<<endl<<"Podaj wartosc do odszukania:";
   cin>>wartosc;

   wynik=bisekcja(wartosc, dane, 6);
   cout<<endl;
   if(wynik<0) cout<<"Brak danej w tablicy lub blad wyszukiwania"<<endl;
   else
   cout<<"Podana dana znaleziono w komorce tablicy o indeksie "<<wynik<<endl;
}

long bisekcja(float szukany, float tablica[], unsigned koniec, unsigned poczatek)
{
   // Sprawdz prawidlowosc skrajnych indeksow tablicy
   if(koniec<poczatek) return -1;

   // Dla tablicy jednoelementowej sprawdz, czy zawiera szukany element
   if(poczatek==koniec)
   {
                if( tablica[poczatek] == szukany ) return poczatek;
                else
                return -2;
   }
   else
   // Dla wieloelementowej dziel rekurencyjnie problem
   {
   long wynik;

                if(szukany <= tablica[ (koniec+poczatek)/2 ] )
                   // Szukaj w pierwszej "polowie"
                   wynik=bisekcja(szukany, tablica, (koniec+poczatek)/2, poczatek);
                else
                   // Szukaj w drugiej "polowie"
                   wynik=bisekcja(szukany, tablica, koniec, (koniec+poczatek)/2+1);

                return wynik;
   }
}
```

Powyższy program realizuje wyszukiwanie zadanej wartości w tablicy metodą bisekcji. Termin *bisekcja* oznacza z łaciny „podział na dwie części”. W jakim celu stosuje się taki mechanizm podczas wyszukiwania? Wyobraźmy sobie sytuację, kiedy mamy tablicę z danymi, spełniającymi dwa poniższe założenia:

1. każda wartość występuje tylko raz (nie ma np. dwóch liczb 17.7 w tablicy)
2. wartości są uporządkowane rosnąco (po niewielkiej przeróbce nasz program działałby dla tablicy wartości uporządkowanych malejąco)

Przy powyższych założeniach można postąpić w następujący sposób: sprawdzić środkowy element w tablicy, jeśli jego wartość jest mniejsza od wartości, której poszukujemy, to możemy odrzucić od razu pierwszą połowę tablicy podczas wyszukiwania! W ten sposób znacząco uprościliśmy problem: mamy dwa razy mniej danych. Analogicznie jeśli wartość ze środkowej komórki tablicy była większa lub równa wartości szukanej, odrzuca się drugą połowę tablicy. Element graniczny dla tablic o nieparzystej ilości elementów rozdziela pozostałe elementy na dwie równe grupy, dla tablic o parzystej ilości elementów jest ostatnim z elementów z grupy elementów tablicy o niższych indeksach. Jest to wynikiem działania operatora dzielenia dla liczb całkowitych.

W tym momencie jesteśmy o krok od rekurencyjnego określenia rozważanego problemu. Powyższy sposób podziału tablicy doprowadzi nas zawsze do analizy w funkcji fragmentu tablicy, zawierającego jej jeden element. Dla tablicy zawierającej jeden element wyszukiwanie jest trywialne: jeśli element tablicy zawiera szukaną wartość, wtedy zwracamy indeks elementu jako wynik poszukiwania, a jeśli element nie zawiera szukanej wartości, to wtedy zwracamy informację, że poszukiwanej wartości nie ma w tablicy. Jeśli tablica zawiera więcej niż jeden element, to funkcja wywołuje się rekurencyjnie dla odpowiedniej części tablicy (jednej „połowy“) i zwraca jako wynik indeks elementu, dostarczony jej przez wywołanie rekurencyjne, lub przekazuje dalej informację o tym, że wartość nie istnieje w tablicy. Do rozwiązania pozostaje problem: jak funkcja może poinformować użytkownika, że nie znalazła poszukiwanej wartości w tablicy? W naszym przypadku najwygodniejsze są liczby ujemne: elementy tablic indeksuje się od zera, zatem jeśli funkcja zwróci wartość nieujemną, uznamy ją za indeks elementu tablicy, zawierającej poszukiwaną wartość, a jeśli funkcja zwróci wartość ujemna, uznamy to za kod błędu.

W powyższym programie pojawił się nowy element języka *C++*, a mianowicie tzw. argumenty domniemane. Spójrzmy na prototyp funkcji *bisekcja*:

```
long bisekcja(float szukany, float tablica[], unsigned koniec, unsigned poczatek=0);
```

Tak zadeklarowana funkcja może zostać wywołana następująco:

```
bisekcja(wartosc, dane, 6)
```

Proszę zauważyć, że lista argumentów zawiera trzy argumenty wywołania, a funkcja oczekuje czterech! W rozważanej sytuacji nie jest to błędem, gdyż kompilator został poinformowany, że w przypadku braku czwartego argumentu, ma on mieć wartość, równą zero. Innymi słowy, powyższe wywołanie jest równoznaczne następującemu:

```
bisekcja(wartosc, dane, 6, 0)
```

Ważne jest to, że argumenty domniemane muszą znajdować się na końcu listy argumentów funkcji, ponadto jeśli jest ich kilka, muszą stanowić „zwarty blok”: argument domniemany może być ostatnim elementem listy argumentów, albo musi mieć po swojej prawej stronie argument domniemany. Ponadto należy pamiętać, że argumenty domniemane deklarujemy tylko raz: w prototypie funkcji (w takim przypadku nie wolno deklarować wartości domniemanych w nagłówku definicji funkcji), a w przypadku, gdy funkcja nie ma prototypu, w nagłówku definicji funkcji.

Typowym przykładem rekurencyjnego określenia funkcji jest tzw. ciąg Fibonacciego:

$$fib(n)=\begin{cases} n & , gdy\ n\in\{0,\ 1\ \} \\ fib(n-1)+fib(n-2) & , gdy\ n>1,\ n\in\mathbb{N} \end{cases}$$

Przykładowy program, realizujący tę funkcję (plik *fibonacci.cpp*):

```
#include<iostream>
using namespace std;

unsigned fibonacci(unsigned);

int main()
{
unsigned i;

   cout<<"Program wypisuje kilka pierwszych liczb Fibonacci\'ego"<<endl<<endl;
   for(i=0; i<10; i++) cout<<"n="<<i<<", fibonacci(n) = "<<fibonacci(i)<<endl;
}

unsigned fibonacci(unsigned n)
{
   if(n<2) return 1;
   else
   return fibonacci(n-2)+fibonacci(n-1);
}
```

Wynik działania programu:

```
Program wypisuje kilka pierwszych liczb Fibonacci'ego

n=0, fibonacci(n) = 1
n=1, fibonacci(n) = 1
n=2, fibonacci(n) = 2
n=3, fibonacci(n) = 3
n=4, fibonacci(n) = 5
n=5, fibonacci(n) = 8
n=6, fibonacci(n) = 13
n=7, fibonacci(n) = 21
n=8, fibonacci(n) = 34
n=9, fibonacci(n) = 55
```

Zadanie:

Dany jest ciąg postaci:

$$1+\frac{1}{2}-\frac{1}{3}+\frac{1}{4}+...$$

Zadanie polega na znalezieniu sumy *n* pierwszych elementów powyższego ciągu za pomocą zależności rekurencyjnej.

Rozwiązanie:

Po pierwsze zauważmy, że suma „zero“ pierwszych elementów ciągu wynosi zero, a suma „jeden“ początkowych elementów wynosi jeden. Następnie zauważmy, że dla n większych od jedności, suma *n* pierwszych elementów tego ciągu to suma *n-tego* elementu i wartości sumy *n-1* wcześniejszych elementów. Otrzymaliśmy zatem określenie rekurencyjnego ciągu postaci:

$$szereg(n)=\begin{cases} 0 & , dla\ n=0 \\ 1 & , dla\ n=1 \\ (-1)^n\cdot\frac{1}{n}+szereg(n-1) & , dla\ n>1 \end{cases}$$

(nazwa *szereg* jest nieco myląca, gdyż ściśle rzecz biorąc, jest to ciąg sum częściowych)

Przykładowy listing, realizujący powyższe zadanie (plik *szereg.cpp*):

```
#include<iostream>
```

```
using namespace std;

double szereg(unsigned);
double znak(unsigned);

int main()
{
unsigned i;

   cout<<"Program wyliczy kilka pierwszych sum czesciowych szeregu..."<<endl<<endl;
   for(i=0; i<10; i++) cout<<"n="<<i<<", szereg(n) = "<<szereg(i)<<endl;
}

double szereg(unsigned n)
{
   if(n==0) return 0;
   if(n==1) return 1;
   else
   return znak(n)*1.0/n + szereg(n-1);
}

double znak(unsigned n)
{
   if(n % 2) return -1; else return 1;
}
```

Po wykonaniu programu otrzymamy następujące informacje:

```
Program wyliczy kilka pierwszych sum czesciowych szeregu...

n=0, szereg(n) = 0
n=1, szereg(n) = 1
n=2, szereg(n) = 1.5
n=3, szereg(n) = 1.16667
n=4, szereg(n) = 1.41667
n=5, szereg(n) = 1.21667
n=6, szereg(n) = 1.38333
n=7, szereg(n) = 1.24048
n=8, szereg(n) = 1.36548
n=9, szereg(n) = 1.25437
```