ZAAWANSOWANE PROGRAMOWANIE KOMPUTEROWE WNE (2022)

KRZYSZTOF ZIEMIAŃSKI

3. Rekurencja

Rekurencja to technika programowania polegająca na zastąpieniu pętli (iteracji) przez wywołania funkcji przez siebie samą. Oto klasyczny przykład.

3.1. **Silnia.** Funkcja silnia może być zdefiniowana na dwa sposoby:

- *Definicja iteracyjna:* $n! = 1 \cdot 2 \cdot 3 \cdot \ldots \cdot (n-1) \cdot n$.
- *Definicja rekurencyjna:* $n! = n \cdot (n-1)!$ jeśli $n > 0$ oraz $0! = 1$.

Poniżej funkcje korzystające z poszczególnych definicji.
Iteracyjna:

```
int silnia(int n) {
    int w=1;
    for(int i=1; i<=n; i++)
        w*=i;
    return w;
}
```

Rekurencyjna:

```
int silnia(int n) {
    if(n==0)
        return 1;
    return n*silnia(n-1);
}
```

Funkcja silnia nie jest wyjątkiem — każda funkcja, którą można zapisać za pomocą iteracji (tzn. pętli) można też zapisać za pomocą rekurencji i na odwrót: każdą funkcję rekurencyjną można zastąpić równoważną funkcją iteracyjną. Czasem łatwiej i lepiej użyć iteracji, a czasem odwrotnie.

3.2. **Drukowanie ciągu liczb.** Poniżej kolejny przykład: funkcja, która drukuje na ekranie liczby od 1 do n. Bardziej naturalna jest wersja iteracyjna:

```
void drukujLiczby1(int n) {
    for(int i=1; i<=n; i++)
        cout << i << " ";
}
```

Można też za pomocą rekurencji:

```
void drukujLiczby2(int n, int k=1) {
    if(k<=n) {
        cout << k << " ";
        g(n, k+1);
```

```
    }
}
```

Proszę zwrócić uwagę, że wymaga to użycia dodatkowego parametru `k`, który mówi od której liczby należy rozpocząć drukowanie. Tak naprawdę, funkcja `drukujliczby2` jest bardziej uniwersalna: drukuje liczby od dowolnego `k`. W przypadku zamiany iteracji na rekurencję czasem istnieje konieczność użycia takich "dodatkowych" parametrów. W tym przypadku można jednak tego uniknąć zamieniając kolejność drukowania liczby i wywołania rekurencyjnego funkcji:

```
void drukujLiczby3(int n) {
    if(n>0) {
        g(n-1);
        cout << n << " ";
    }
}
```

3.3. **Ciąg Fibonacciego.** Ciąg Fibonacciego to ciąg liczb

$$0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, 144, \ldots$$

zdefiniowany następująco

$$F_n = \begin{cases} 0 & \text{dla } n = 0 \\ 1 & \text{dla } n = 1 \\ F_{n-2} + F_{n-1} & \text{dla } n = 2, 3, 4, \ldots \end{cases}$$

Istnieje też jawny wzór na F_n

$$F_n = \frac{1}{\sqrt{5}} \left(\frac{1+\sqrt{5}}{2} \right)^n - \frac{1}{\sqrt{5}} \left(\frac{1-\sqrt{5}}{2} \right)^n,$$

który jednak nie wygląda zbyt prosto. Korzystając ze wzoru rekurencyjnego możemy łatwo napisać funkcję, która oblicza F_n:

```
int fibo(int n) {
    if(n==0) return 0;
    if(n==1) return 1;
    return fibo(n-2)+fibo(n-1);
}
```

Funkcja działa poprawnie, ale bardzo wolno (proszę spróbować obliczyć `fibo(45)`). Powodem jest to, że wartości `fibo(n)` dla poszczególnych liczb `n` są obliczane wielokrotnie. Można temu zaradzić na dwa sposoby:

1. Zapamiętywać wartości, które już zostały policzone:

```
vector<int> tablicaFibo; // wymaga biblioteki <vector>

int fibo(int n) {
    // przy pierwszym wywołaniu fibo ustawiamy dwa pierwsze elementy
    if(tablicaFibo.size()<2) {
        tablicaFibo.push_back(0); // fibo(0)==0
        tablicaFibo.push_back(1); // fibo(1)==1
    }
```

```
    // jeśli element nie został jeszcze policzony
    // to liczymy i wpisujemy do tablicy
    if(tablicaFibo.size()<=n) {
        int v=fibo(n-2)+fibo(n-1);
        tablicaFibo.push_back(v);
        // elementy na pewno licza się po kolei,
        // więc wiemy, że wpisze się na właściwym miejscu
    }
    return tablicaFibo[n];
}
```

2. Użyć iteracji:

```
int fibo(int n) {
    if(n==0) return 0;
    if(n==1) return 1;
    int a=0;
    int b=1;
    for(int i=2; i<=n; i++) {
        // Zastępujemy parę (a,b) przez (b,a+b)
        int c=a+b;
        a=b;
        b=c;
    }
    return b;
}
```

3.4. **Zadania.**

(1) Napisać rekurencyjną funcję

```
void ciag(int p, int r, int n);
```

która drukuje ciąg artymetyczny o `n` wyrazach, pierwszym wyrazie `p` oraz różnicy `r`.

(2) Napisać rekurencyjną funcję

```
int wystapienia(char* s, char c);
```

która oblicza liczbę wystąpień znaku `c` w napisie `s`.

(3) Napisać rekurencyjną funcję

```
bool prefiks(char* s, char* t);
```

która zwraca `true` jeśli napis `t` jest prefiksem napisu `s`.

(4) Napisać rekurencyjną funcję

```
int znajdz(char* s, char* t);
```

która zwraca indeks pierwszego wystąpienia napisu `t` w napisie `s`. Jeśli brak wystąpień, należy zwrócić -1.

(5) Napisać rekurencyjną funkcję

```
void drukuj(int n, int r)
```

która drukuje liczbę `n` w systemie pozycyjnym o podstawie `r`. Zadbać o to, aby cyfry drukowały sie w kolejności od najbardziej znaczącej do najmniej znaczącej.

(6) Napisać rekurencyjną funkcję

```
bool zawiera(char* s, char* t)
```

która zwraca `true` jeśli napis `t` można uzyskać z napisu `s` przez usunięcie pewnej liczby znaków (niekoniecznie kolejnych).

(7) Poniższa funkcja

```
void tabliczka(int n) {
    for(int i=1; i<=n; i++) {
        for(int j=1; j<=n; j++) {
            if(i*j<10) cout << " ";
            cout << " " << i*j;
        }
        cout << endl;
    }
}
```

drukuje tabliczkę mnożenia o wymiarach `n` $\times$ `n`. Napisać równoważną funkcję rekurencyjną (nie używającą pętli).

(8) Dwumian Newtona jest zdefiniowany wzorem ($n \geqslant k \geqslant 0$)

$$\binom{n}{k} = \frac{n!}{k!(n-k)!}$$

lub równoważnym wzorem rekurencyjnym

$$\binom{n}{k} = \begin{cases} 0 & \text{dla } n < k \text{ lub } k < 0 \\ 1 & \text{dla } 0 = k \leqslant n \text{ lub } 0 \leqslant k = n \\ \binom{n-1}{k-1} + \binom{n-1}{k} & \text{w pozostałych przypadkach.} \end{cases}$$

Należy napisać dwie funkcje

```
int dwumian(int n, int k)
```

jedna korzystająca z definicji iteracyjnej, a druga z rekurencyjnej oraz przetestować je i znaleźć ich wady (podpowiedź: dla dużych n i k jedna z nich będzie działać wolno, a druga będzie zwracać nieprawidłowe wyniki). Następnie napisać funkcję `dwumian`, która będzie tych wad pozbawiona, tzn. będzie działać szybko oraz poprawnie (o ile wynik będzie się mieścił w zakresie typu `int`, tj. około $2 \cdot 10^9$).