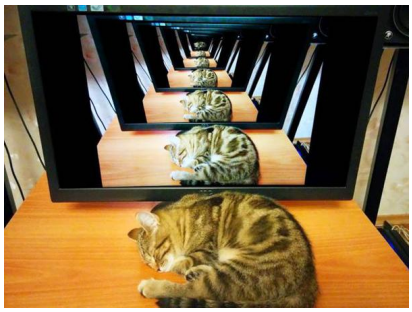
O rekurencji



#rekurencja #iteracja
#instancja_funkcji
#stos #złożoność_pamięciowa
#spamiętywanie #zmienna_globalna

Z rekurencją mamy do czynienia, gdy funkcja lub definicja odwołuje się sama do siebie. Weźmy dla przykładu definicję *silni* – funkcji bardzo często pojawiającej się w różnych działach matematyki. Otóż silnia (ang. **factorial**, wymawiaj: *faktorial*), oznaczana symbolem wykrzyknika (!) to iloczyn kolejnych liczb naturalnych:

$$n! = 1 \cdot 2 \cdot 3 \cdot \ldots \cdot (n-1) \cdot n$$

Ten wzór ma sens, jeśli n jest dodatnią liczbą naturalną. Przyjmuje się również, że $0! = 1$. Definicja przytoczona powyżej jest typową definicją odwołującą się do *iteracji*, czyli powtarzania pewnego rodzaju operacji (tutaj: mnożenia i powiększania mnożnika). Stąd wniosek, że możemy zaimplementować tę funkcję przy użyciu prostej pętli:*

```
int factorial(int n)
{
  int f{ 1 };
  for(int k{ 2 }; k <= n; k++)
    f *= k;
  return f;
}
```

Na pewno już wiemy, że operator `*=` oznacza pomnożenie lewej strony instrukcji przez wyrażenie po jej prawej stronie. Zauważmy, że tak zdefiniowana funkcja daje poprawny wynik także dla $n = 0$.

Możemy nawet powyższą funkcję uprościć, rezygnując z pomocniczej zmiennej k, to działa (dlaczego?):

```
int factorial(int n)
{
  int f{ 1 };
  while(n > 1)
    f *= n--;
  return f;
}
```

*Proszę być ostrożnym z zakresem wartości argumentu n: silnia jest funkcją bardzo szybko rosnącą i dla n rzędu kilkanaście na bank wyjdziemy poza zakres typu `int`. Możemy ratować się użyciem typu `long long`, ale to tylko odroczy wyrok.

Do przetestowania powyższych funkcji może posłużyć prościutka funkcja `main()`:

```
int main()
{
  int n;
  cin >> n;
  cout << factorial(n) << '\n';
  return 0;
}
```

Nagłówek programu chyba nie stworzy problemu Czytelnikowi.

Tyle o iteracji, czas na rekurencję.

Definicję silni możemy zbudować w inny sposób. Czym różni się $n!$ od $(n-1)!$? No, jeśli n jest większe od 0, to mamy $n! = n \cdot (n-1)!$, prawda? Czyli możemy zaimplementować tę funkcję odwołując się do niej samej – ale od mniejszego argumentu. Tylko uwaga: takie odwoływanie się musi mieć warunek zakończenia, aby algorytm wykonywał się w skończonym czasie. W przypadku silni jest to warunek $n! = 1$ dla n mniejszych od 2. (Mówimy tylko o wartościach nieujemnych n.) Oto proponowana funkcja:

```
int factorial(int n)
{
  if(n < 2) return 1;
  return n * factorial(n - 1);
}
```

O żeż kurczak, tak krótko? Nawet `else`'a poskąpili? No tak, `return` oznacza wyjście z funkcji, więc `else` jest niepotrzebny.

To jest tak dobre, że jest warte dokładnego prześledzenia. Użyjemy do tego *stosu* (ang. **stack**, wymawiaj: *stak*), czyli struktury danych, gdzie odkładamy coś na jej wierzch, a potem zdejmujemy.[†] Zobaczymy, co dzieje się, gdy chcemy obliczyć na przykład `factorial(4)`. Czas na generalną uwagę: funkcja zdefiniowana przez nas w programie to jakby wzorzec czy przepis na pewną sekwencję operacji. Jeśli ją uruchomimy, wtedy system operacyjny na moment przenosi sterowanie do jej wnętrza, zapamiętując na *stosie* adres powrotny, to znaczy miejsce, gdzie trzeba będzie skoczyć, gdy funkcja wypełni swoje działanie. Tym samym w systemie pojawia się *instancja funkcji*, czyli jej działający egzemplarz. Na przykład po wywołaniu funkcji `factorial(4)` wspomniany stos wygląda tak (zawiera tylko jedną instancję funkcji `factorial`):

```
factorial(4)
  { return 4 * factorial(3); }
```

Opuściliśmy `if`-a, przecież tutaj nie zadziała.

[†] Wrócimy do tego w jednym z najbliższych podrozdziałów.

No dobrze, mamy teraz wywołanie `factorial(3)` i co z nim? Otóż ląduje ono na szczycie wspomnianego stosu jako kolejna instancja funkcji `factorial()`:

```
factorial(3)
  { return 3 * factorial(2); }
factorial(4)
  { return 4 * factorial(3); }
```

Mamy teraz wywołanie funkcji `factorial(2)`, więc dokładamy je na stos:

```
factorial(2)
  { return 2 * factorial(1); }
factorial(3)
  { return 3 * factorial(2); }
factorial(4)
  { return 4 * factorial(3); }
```

Tym razem jednak mamy nową jakość: `factorial(1)` nie musi odwoływać się do kolejnej instancji, tylko może po prostu zwrócić wartość 1, zatem nasz stos wygląda tak:

```
factorial(1)
  { return 1; }
factorial(2)
  { return 2 * factorial(1); }
factorial(3)
  { return 3 * factorial(2); }
factorial(4)
  { return 4 * factorial(3); }
```

Nasz stos zaczyna się zwijać, począwszy od szczytu:

```
factorial(2)
  { return 2 * 1; }
factorial(3)
  { return 3 * factorial(2); }
factorial(4)
  { return 4 * factorial(3); }
```

Dalej mamy:

```
factorial(3)
  { return 3 * 2; }
factorial(4)
  { return 4 * factorial(3); }
```

W kolejnym kroku:

```
factorial(4)
  { return 4 * 6; }
```

co w zasadzie kończy sprawę, bo funkcja zwraca oczekiwaną wartość 24.

Tak to mniej więcej działa.

Teraz możemy spróbować przerobić jakiś znany nam już przykład na postać rekurencyjną. Może obliczanie ilości cyfr liczby naturalnej? Było, w podrozdziale *Ilość cyfr*. Warunkiem zakończenia rekurencji będzie oczywiście sytuacja, gdy dana liczba jest mniejsza od 10 – wtedy wynikiem funkcji będzie 1. W przeciwnym razie postępujemy tak: wywołujemy kolejną instancję funkcji (zwaną też *potomkiem*) z naszą liczbą pozbawioną ostatniej cyfry – i do jej wyniku dodajemy 1. Obcinanie ostatniej cyfry to po prostu dzielenie całkowite przez 10, zatem funkcja może mieć taką oto zwięzłą postać:

```
int digits(int n)
{
  if(n < 10) return 1;
  return 1 + digits(n / 10);
}
```

Fajne, prawda? Istotnie, zapis rekurencyjny jest często bardzo elegancki i efektowny. Ale czy daje on efektywny kod? Niekoniecznie.

Podczas wywołań rekurencyjnych tworzone są kolejne instancje funkcji, co zajmuje pamięć operacyjną. Oczywiście przytoczone tu przykłady są bardzo proste i w żadnym przypadku nie stworzą problemu, ale generalnie rzecz biorąc algorytmy rekurencyjne cechuje duża *złożoność pamięciowa*.

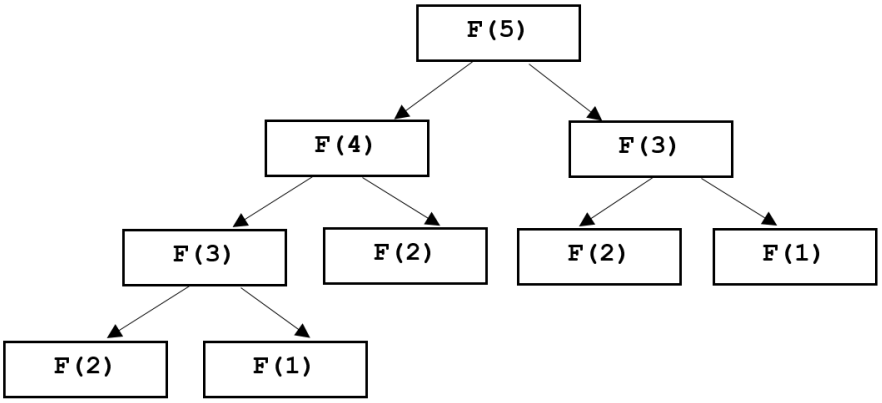
To jeszcze można by przeboleć, gdyby nie kolejny przykład, a dokładniej przeróbka omawianego przez nas algorytmu obliczania liczb Fibonacci'ego (przypominamy podrozdział *Kot Leonarda*). Zastanówmy się, co by się stało, gdybyśmy zastosowali wprost przytoczony wzór – jako żywo rekurencyjny:

$$\begin{aligned} F_0 &= 0, \\ F_1 &= 1, \\ F_n &= F_{n-1} + F_{n-2}, \quad \text{dla } n \geqslant 2. \end{aligned}$$

Wtedy otrzymalibyśmy następującą funkcję:

```
int F(int n)
{
  if(n < 2) return n;
  return F(n - 1) + F(n - 2);
}
```

Z pozoru wygląda niegroźnie, ale wyobraźmy sobie, co się dzieje, gdy wywołamy przykładowo `F(5)`. Na diagramie przedstawiamy układ wywołań kolejnych instancji funkcji – tym razem od góry na dół:



Jasno widać, że na przykład funkcja `F(2)` jest wywoływana aż trzykrotnie i są to zupełnie niezależne instancje funkcji. Dla większych argumentów powyższy diagram rozrasta się niemożebnie i obliczanie wartości funkcji trwa bardzo długo. Nie jest to więc efektywna metoda obliczania liczb Fibonacci'ego.

Pojawia się naturalne pytanie: czy można połączyć elegancję algorytmu rekurencyjnego z efektywnością obliczeniową? Odpowiedź jest pozytywna, zaś rozwiązaniem problemu jest technika zwana *spamiętywaniem*,[‡] będąca podstawą *programowania dynamicznego*, które opiszemy w dalszej części tego podręcznika.

Trick polega na tym, aby raz obliczoną wartość funkcji zapamiętać w podręcznej tablicy, tak aby mogła być wykorzystana przez inne instancje funkcji. Potrzebna jest zatem pomocnicza tablica (nazwiemy ją $Ftab[\,]$) wypełniona na początku jakąś neutralną wartością: taką, jaka nigdy nie występuje wśród liczb Fibonacci'ego – na przykład -1.[§] Taka inicjalizacja musi zostać zrealizowana przed jakimkolwiek odwołaniem do funkcji `F()`.

Poprawiona wersja funkcji `F()` mogłaby na przykład wyglądać tak:

```
const int FMAX{ 46 };
int Ftab[FMAX];  // konieczna inicjalizacja!

int F(int n)
{
  if(Ftab[n] < 0)
  {
    if(n < 2)
      Ftab[n] = n;
    else
      Ftab[n] = F(n - 1) + F(n - 2);
  }
  return Ftab[n];
}
```

[‡]Zwanym też *memoizacją*.

[§]Tak naprawdę istnieją ujemne liczby Fibonacci'ego, ale nie będziemy się nimi tutaj zajmować. Można o nich przeczytać w znakomitej skądinąd książce *Matematyka konkretna*, którą napisał Donald E. Knuth ze swymi współpracownikami.

Nawias klamrowy otaczający wewnętrzną instrukcję warunkową nie jest konieczny, ale został dodany dla większej przejrzystości kodu.

Zmienne/stałe zadeklarowane poza jakąkolwiek funkcją określa się mianem *globalnych*, gdyż są dostępne wszędzie w programie, począwszy od miejsca deklaracji. Rodzi to pewne ryzyko, że na przykład we wnętrzu jakiejś funkcji zmienimy przypadkowo wartość zmiennej globalnej (choćby na skutek błędu literowego w nazwie zmiennej), co może mieć fatalne skutki. Dlatego też starajmy się ograniczać ich używanie.

Do pełnego działania konieczne jest jeszcze użycie poniższej instrukcji przed pierwszym wywołaniem funkcji:

```
fill(Ftab, Ftab + FMAX, -1);
```

Funkcja `fill()` wypełnia wskazaną strukturę danych wybraną przez nas wartością.[¶]

Oczywiście sposób obliczania liczb Fibonacci'ego przedstawiony w podrozdziale *Kot Leonarda* można uznać za optymalny w każdym zastosowaniu. Metoda spamiętywania przydaje się jednak wtedy, gdy obliczenie wartości funkcji zajmuje wiele czasu, a z jakiegoś powodu nie chcemy rezygnować z rekurencji.

[¶]Jeśli do przechowywania danych korzystamy z kontenera w rodzaju `vector<>`, wtedy jego inicjalizację możemy zawrzeć bezpośrednio w deklaracji struktury lub podczas ustalania jej rozmiaru (funkcja `resize()`) – o tym będzie później.