31 Liceum Ogólnokształcące
im. Romana Ingardena w Krakowie

ROMAN CZAPLA

REKURENCJA

Kraków 2 października 2024

Rekurencja - definicje

- Popularne wśród programistów powiedzenie głosi, że aby zrozumieć rekurencję, należy najpierw zrozumieć rekurencję.
- Formalna definicja *rekurencji* (zwanej również *rekursją*[1]) opisuje ją jako algorytm składający się z przypadku *bazowego* (*początkowego*, *brzegowego*) oraz zestawu reguł pozwalających na zredukowanie do niego wszystkich pozostałych przypadków.
- Potocznie przyjęło się określać terminem rekurencji zagnieżdżonego odwołania się jakiejś funkcji lub metody do samej siebie.

Zasadniczo idea jest podobna. Definicja formalna określa jednak bardzo ważny element rekurencji: *warunek stopu* gwarantujący, że wykonywanie algorytmu kiedyś się skończy. Niestety większość błędów w algorytmach rekurencyjnych polega na tym, że warunek ten nigdy nie zachodzi.

[1] Tak naprawdę między tymi pojęciami istnieje subtelna różnica, ale obecnie używa się tych terminów wymiennie.

Rekurencja w matematyce

- W matematyce możemy spotakać wiele rekurencyjnych definicji czy algorytmów:
  - wielomiany Hermite'a,
  - wielomiany Legendre'a,
  - algorytm Euklidesa,
  - pojęcie silni,
  - symbol Newtona,
  - cecha podzielności przez 3 dla liczby w zapisie dziesiętnym,
  - schemat Hornera.
- Przykład rekurencyjnej definicji silni:

$$n! = \begin{cases} 1, & \text{dla } n = 0, \\ n \cdot (n-1)! & \text{dla } n > 0. \end{cases}$$

Rekurencja - technika programistyczna

- W języku Python, jak i w praktycznie każdym współczesnym języku programowania, dopuszcza się, aby w definicji funkcji (ciele funkcji) znajdowała się instrukcja wywołania tej samej funkcji - mówimy wówczas o funkcji *bezpośrednio rekurencyjnej*.
- Jeśli dana funkcja f zawiera odwołanie do innej funkcji g , która to zawiera bezpośrednie (lub pośrednie) odwołanie do funkcji f to mówimy, że funkcji f jest funkcją *pośrednio rekurencyjną*.
- Język Python obsługuje funkcje rekurencyjne dzięki zastosowaniu stosu wywołań - obszaru w pamięci wydzielonego dla danego wątku, służącego do przechowywania adresów powrotu i zmiennych lokalnych.

Podejście rekurencyjne, a podejście iteracyjne

- Rekurencja często jest przeciwstawiana podejściu iteracyjnemu.
- Algorytmy iteracyjne polegają na n-krotnym wykonywaniu instrukcji w taki sposób, żeby wyniki uzyskane w przednich iteracjach (przebiegach) mogły być wykorzystane jako dane wejściowe do wyznaczenia kolejnych (np. instrukcje pętli `for` lub `while`).
- Algorytmy rekurencyjne działają podobnie, ale proces zapętlania jest realizowany prze wywołanie tej samej funkcji (procedury) przez siebie samą z innymi parametrami.
- Należy mieć świadomość, że programy zapisane w formie rekurencyjnej mogą być zawsze przekształcone na klasyczną postać rekurencyjną - z większym lub mniejszym wysiłkiem[2].

  PRZYKŁAD 1

[2] Wynika to z tezy Churcha-Turinga.

Funkcja rekurencyjna - analiza przypadku

- Implementacja rekurencyjnego algorytmy wyznaczającego wartość silni (funkcja `factorial`):

  PRZYKŁAD 2

```python
def factorial(n):
    if n == 0:
        return 1;                           # przypadek bazowy
    else:
        return n * factorial(n - 1);        # redukcja


m = int(input("podaj m: "))
print(f"{m}! = {factorial(m)}")
```

Funkcja rekurencyjna - analiza przypadku

- Rozpisując algorytm obliczania silni z wykorzystaniem funkcji `factorial` na pojedyncze reguły dla `n = 4` otrzymamy następujący układ równań:

```
factorial(4) = 4 * factorial(3) // redukcja
factorial(3) = 3 * factorial(2) // redukcja
factorial(2) = 2 * factorial(1) // redukcja
factorial(1) = 1 * factorial(0) // redukcja
factorial(0) = 1
```

a jego rekurencyjne rozwinięcie w kolejnych krokach wyglądałoby następująco:

```
4 * factorial(3)
4 * (3 * factorial(2))
4 * (3 * (2 * factorial(1)))
4 * (3 * (2 * (1 * factorial(0))))
4 * (3 * (2 * (1 * 1)))
```

Implementacja rekurencji (funkcja `factorial`) – stos wywołań

- Przy każdym wywołaniu funkcji **`factorial`** na stos odkładane są kolejne wartości zmiennej **`n`**. Dzieje się tak do momentu, kiedy funkcja zwróci 1 – wówczas wartości te są w odwrotnej kolejności zdejmowane ze stosu i mnożone.
- *Stos* jest stałym, wydzielonym obszarem pamięci, który zostaje przydzielony przez system operacyjny każdej aplikacji do wykorzystania na własne potrzeby, w tym na przechowywanie zmiennych oraz informacji związanych z przepływem sterowania.
- W praktyce na stos za każdym wywołaniem funkcji **`factorial`** odkładane są również jej parametry, ewentualne zmienne lokalne oraz adres powrotu umożliwiający procesorowi kontynuowanie pracy po powrocie z podprogramu. Każda taka porcja danych nosi nazwę *ramki stosu* lub *rekordu stosu*. Analogicznie wygląda to w przypadku dowolnej funkcji rekurencyjnej.

Implementacja rekurencji – stos wywołań

- Stan dowolnej funkcji reprezentowany jest przez:
  - parametry funkcji;
  - adres powrotu - adres instrukcji wykonywanej po zakończeniu funkcji;
  - zmienne lokalne funkcji;
  - zwracana wartość – jeśli tylko funkcja takową zwraca.

Każde wywołanie funkcji powoduje utworzenie na stosie kolejnego obiektu ramki stosu.

Implementacja rekurencji – stos wywołań

- Stan stosu po kolejnych wywołaniach funkcji

```
⋮
def f3():
    pass
def f2():
    f3()
def f1():
    f2()

f1()
⋮
```

Analiza stanu stosu wywołań

- Wywołanie funkcji `factorial(3)`:

wywołanie nr 1

wierzchołek stosu ⇑

ramka stosu (`factorial(3)`)
wartość zwracana: `?`
zmienne lokalne:
adres powrotu: `wiersz nr` 9
parametry funkcji: `n = 3`

wywołanie nr 2

wierzchołek stosu ⇑

ramka stosu (`factorial(2)`)
wartość zwracana: `?`
zmienne lokalne:
adres powrotu: `wiersz nr` 5
parametry funkcji: `n = 2`
ramka stosu (`factorial(3)`)
wartość zwracana: `?`
zmienne lokalne:
adres powrotu: `wiersz nr` 6
parametry funkcji: `n = 3`

wywołanie nr 3

wierzchołek stosu ⇑

ramka stosu (`factorial(1)`)
wartość zwracana: `?`
zmienne lokalne:
adres powrotu: `wiersz nr` 5
parametry funkcji: `n = 1`
ramka stosu (`factorial(2)`)
wartość zwracana: `?`
zmienne lokalne:
adres powrotu: `wiersz nr` 5
parametry funkcji: `n = 2`
ramka stosu (`factorial(3)`)
wartość zwracana: `?`
zmienne lokalne:
adres powrotu: `wiersz nr` 9
parametry funkcji: `n = 3`

Analiza stanu stosu wywołań

- Wywołanie funkcji `factorial(3)`:

wierzchołek stosu ⇓

ramka stosu (`factorial(0)`)
wartość zwracana: `1`
zmienne lokalne:
adres powrotu: `wiersz nr` 5
parametry funkcji: `n = 0`
ramka stosu (`factorial(1)`)
wartość zwracana: `?`
zmienne lokalne:
adres powrotu: `wiersz nr` 5
parametry funkcji: `n = 1`
ramka stosu (`factorial(2)`)
wartość zwracana: `?`
zmienne lokalne:
adres powrotu: `wiersz nr` 5
parametry funkcji: `n = 2`
ramka stosu (`factorial(3)`)
wartość zwracana: `?`
zmienne lokalne:
adres powrotu: `wiersz nr` 9
parametry funkcji: `n = 3`

wywołanie nr 4

wierzchołek stosu ⇓

ramka stosu (`factorial(1)`)
wartość zwracana: `1 * 1`
zmienne lokalne:
adres powrotu: `wiersz nr` 5
parametry funkcji: `n = 1`
ramka stosu (`factorial(2)`)
wartość zwracana: `?`
zmienne lokalne:
adres powrotu: `wiersz nr` 5
parametry funkcji: `n = 2`
ramka stosu (`factorial(3)`)
wartość zwracana: `?`
zmienne lokalne:
adres powrotu: `wiersz nr` 9
parametry funkcji: `n = 3`

wywołanie nr 3

Analiza stanu stosu wywołań

- Wywołanie funkcji `factorial(3)`:

wierzchołek stosu ⇓

ramka stosu (`factorial(2)`)
wartość zwracana: `2 * (1 * 1)`
zmienne lokalne:
adres powrotu: `wiersz nr` 5
parametry funkcji: `n = 2`
ramka stosu (`factorial(3)`)
wartość zwracana: `?`
zmienne lokalne:
adres powrotu: `wiersz nr` 9
parametry funkcji: `n = 3`

wywołanie nr 2

wierzchołek stosu ⇓

ramka stosu (`factorial(3)`)
wartość zwracana: `3 * (2 * (1 * 1))`
zmienne lokalne:
adres powrotu: `wiersz nr` 9
parametry funkcji: `n = 3`

wywołanie nr 1

Problemy z funkcjami rekurencyjnymi

- Obszar pamięci rezerwowany na stos dla danego programu jest zazwyczaj stosunkowo mały i nie może być dynamicznie powiększany w trakcie działania aplikacji. Głównym problemem, który pojawia się podczas korzystania z rekurencji, jest ryzyko szybkiego wyczerpania pamięci stosu i w efekcie przerwanie wykonywania programu przez środowisko uruchomieniowe lub system operacyjny.
- Niektóre języki programowania (takie jak Python) posiadają wbudowane mechanizmy wykrywające odwołania rekurencyjne i przerywające wykonanie programu , jeśli ich liczba przekroczy określony próg. W innych językach, takich jak Java czy C, granicę dla aplikacji stanowi najczęściej ilość pamięci przeznaczonej na stos - mówimy wówczas o błędzie *przepełnienia stosu*.
- Problem przepełnienia stosu będzie jest jeszcze bardziej widoczny w przypadku funkcji, które rekurencyjnie odwołują się do siebie wielokrotnie.

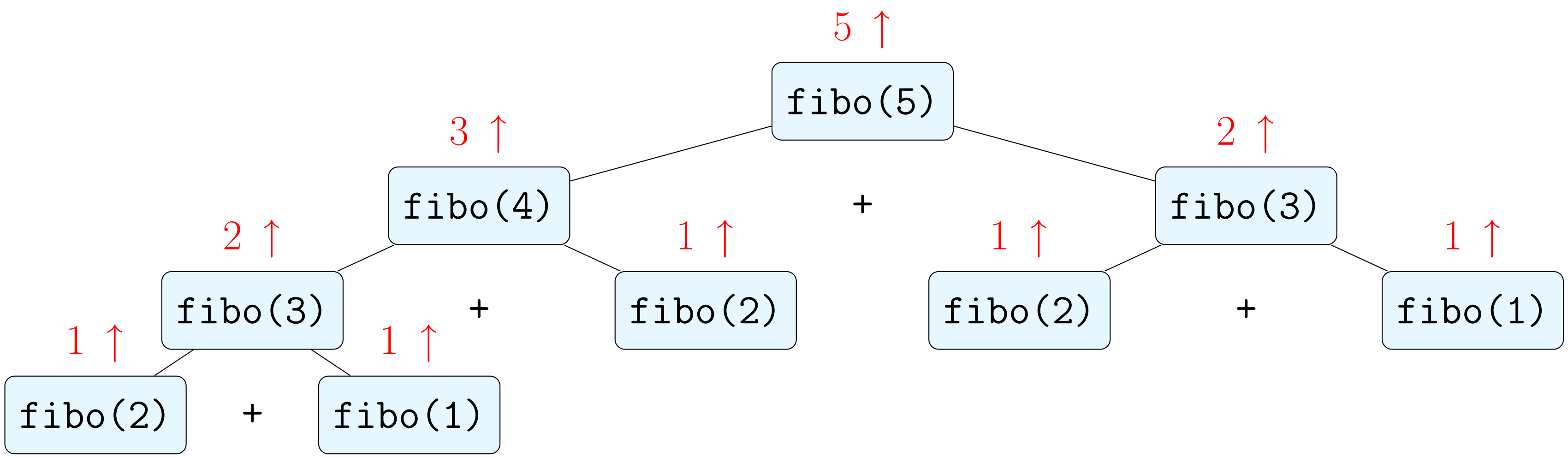
Ciąg Fibonacciego obliczany rekurencyjnie

- Rekurencja definicja ciągu Fibonacciego

$$F_n = \begin{cases} 1, & \text{dla } n = 1 \lor n = 2, \\ F_{n-1} + F_{n-2}, & \text{dla } n > 2. \end{cases}$$

PRZYKŁAD 3

- Drzewo wywołań rekurencyjnych funkcji `fibo` dla `n = 5`:



Zamiana algorytmy rekurencyjnego na iteracyjny

- Najczęściej spotykanym rozwiązaniem związanym z ograniczeniami algorytmów rekurencyjnych jest ich zamiana na wersje iteracyjne, czyli zamiana zagnieżdżonych wywołań funkcji na pętlę. Iteracyjne wywołania funkcji nie powodują konieczności odkładania coraz większej liczby informacji na stosie, a więc nie tylko unikamy ryzyka jego przepełnienia, ale również zbędnego alokowania kolejnych ramek stosu.
  - iteracyjna wersja funkcji `factorial`:

    PRZYKŁAD 4
  - iteracyjna wersja funkcji `fibo`:

    PRZYKŁAD 5