**Temat 6**

*Algorytmy rekurencyjne* to algorytmy, których funkcja wywołuje samą siebie do obliczenia wyniku lub grupa funkcji wywołuje się naprzemiennie, cyklicznie, w celu uzyskania wyniku. Trzeba uważać, żeby nie zrobić nieskończonej rekurencji (bo wtedy wszechświat wybuchnie, nie sprawdzajcie tego) i w dobrym momencie zwrócić wartość (return).

Kolejną pułapką jest złożoność czasowa- zastosowanie rekurencji zamiast iteracji w niektórych przypadkach drastycznie zwiększa ilość wykonywanych operacji. Ratunkiem bywają techniki programowania dynamicznego (np. zapisywanie raz już wyliczonych wyników w tablicy)

Rekurencje wykorzystują więcej pamięci. Źle zaimplementowana rekurencja może niepotrzebnie wykorzystać skrajnie dużo pamięci.

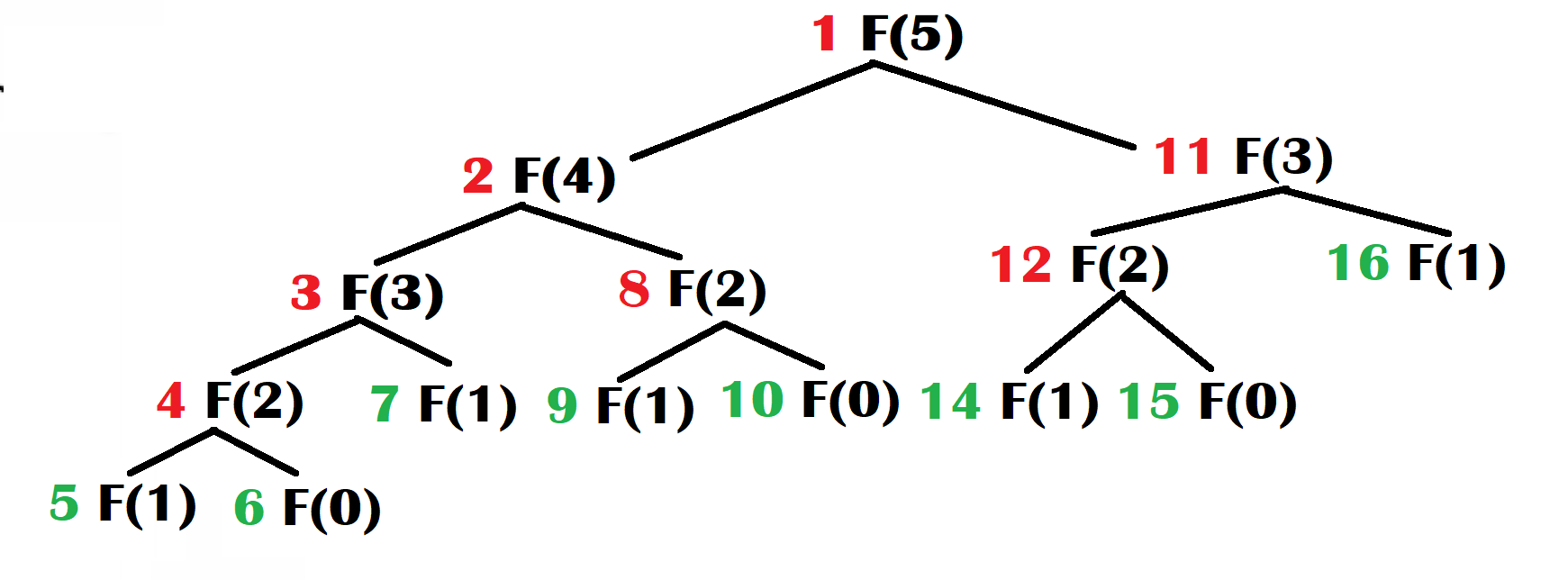
Rekurencję nazywamy ogonową kiedy wywołanie rekurencyjne jest ostatnią rzeczą, którą zajmuje się funkcja. Rekurencja ogonowa korzysta najczęściej z dodatkowego argumetnu

**Zadanie 1.** Fibbonaci

Wzór rekurencyjny na n-ty wyraz ciągu Fibbonaciego:

Funkcja **F()**, ma zwracać określone wartości dla **n = 0** lub **n = 1**, więc one będą punktem granicznym rekurencji. W pozostałych przypadkach funkcja fibbonaci wywołuje dwukrotnie samą siebie   
**F(n-1) + F(n-2).**

Drzewo wywołań funkcji fibbonaci dla **n=5**:



*Jak widać, funkcja wywołuje się aż 16 razy dla n=5.*

W ogólnym przypadku ilość wywołań **f(n)** wynosi dla

Przejdźmy do kodu:

**def** fibbonaci(n):  
 **if** n < 0:  
 **return None  
 if** n == 0 **or** n == 1:  
 **return** n  
 **return** fibbonaci(n-1) + fibbonaci(n-2)

Definiujemy funkcję **fibbonaci(n**) przyjmującą jako argument numer wyrazu ciągu, którego ma zwrócić wartość.

Jeżeli **n** jest ujemne zwracamy **None,** bo .

Jeżeli **n** wynosi **0** lub **1** funkcja zwraca **n**, bo:

* Dla **n = 0** , **F(0) = 0** czyli **F(n) = n**
* Dla **n = 1**, **F(1) = 1** czyli **F(n) = n**

W każdym innym przypadku (**n > 1**) funcja dwukrotnie wywołuje samą siebie, do wyliczenia sumy dwóch poprzednich wyrazów,

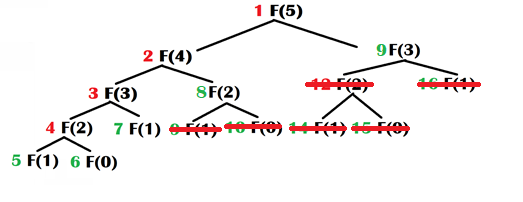
Ostatecznie funkcja zwraca n-ty wyraz ciągu Fibbonaciego.

**Zadanie 1.** Fibbonaci optymalizowanie

Powyższy algorytm można zoptymalizować. Zauważmy, że wyliczając **F(n**) niektóre poprzedzające wyrazy ciągu są wyliczane kilkukrotnie, im większe **n**, tym więcej identycznych wywołań.

Patrząc na drzewo wywołań widać, że **F(3)** jest wyliczane dwukrotnie.

Dobrym pomysłem jest zapisywanie wyliczonych wartości funkcji dla danych **n** w tablicy.



Liczba wywołań funkcji wynosi wtedy 2\*n – 1.

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| **n** | 0 | 1 | 2 | 3 | 4 | 5 |
| **F(n)=** | 0 | 1 | 1 | 2 | 3 | 5 |

Funkcja fibbonaci optymalizowana w ten sposób:

wyniki = [1 **for** x **in** range(n+1)]  
  
**def** fibbonaci(n):  
 **if** n < 0:  
 **return None  
 if** n == 0 **or** n == 1:  
 **return** n  
 **if** value[n] != 1:  
 **return** value[n]  
 value[n] = fibbonaci(n - 1) + fibbonaci(n - 2)  
 **return** value[n]

Przed wywołaniem funkcji fibbonaci, tworzymy tablice zawierającą same liczby **1**. Możemy tak zrobić, ponieważ przypadek gdy **n = 0** lub **n = 1** jest rozpatrywany osobno, a dla **n > 1**:

Pierwsze dwa warunki sprawdzane identycznie jak w pierwszej wersji funkcji.

Kiedy **n > 1** sprawdamy czy **value[n]** jest różne od **1.** Jeżeli tak jest oznacza to, że dany wyraz ciągu został już wyliczony wcześniej. Zwracamy wynik pobrany z tablicy.

W przypadku kiedy **value[n] = 1**, wyliczamy **n**-ty wyarz ciągu rekurencyjnie i zapisuje go w tabilcy **value[n]**  
Wywołana funkcja ostatecznie zwraca **n**-ty wyraz ciągu.

**Zadanie 1.** Fibbonaci iteracyjnie

Można zastosować algorytm iteracyjny do znalezienia **n**-tego wyrazy ciągu (ale na UPEL ma być rekurencyjny). Algorytm ma złożoność **O(n)**.

value = [0,1]  
  
**def** fibbonaci(n):  
 **if** n < 0:  
 **return None  
 for** k **in** range(2, n+1):  
 value.append(value[k-1] + value[k-2])  
 **return** value[n]

Na początku tworzę listę value. **value[n]** oznacza **n**-ty wyraz ciągu Fibbonaciego

**value[0] = 0**

**value[1] = 1**

Tworzę pętlę **for** działającą w zakresie **[2, n+1).** **k** przyjmuje wartości całkowite **2,3,4,…,n**

W każdym obiegu pętli do listy dodaje kolejny wyraz wyliczony z dwóch poprzednich.

Po zakończeniu działania pętli wypisuje **value[n]** (**n**-ty wyraz ciągu)

**Zadanie 2**. Newton

Dla wartość symbolu Newtona możemy wyliczyć następująco:

wynosi **1** dla **k = 0** lub **k = n**, to są przypadki graniczne

Ilość wywołań zwiększa się w podobnym tempie jak w zadaniu 1.

Przejdźmy do kodu:

**def** newton(n,k):  
 **if** k < 0 **or** n < k:  
 **return None  
 if** n == k **or** k == 0:  
 **return** 1  
 **return** newton(n-1,k-1) + newton(n-1,k)

Sprawdzam warunki. Jeżeli nie zachodzi funkcja zwraca **None**.

Sprawdzam czy przypadek jest graniczny. Jeżeli **n = k** lub **k = 0** funkcja zwraca **1**.

W innym przypadku (czyli ) funkcja zwraca sumę wyliczoną rekurencyjnie sumę **newton(n-1,k-1)** i **newton(n-1,k)**

Ostatecznie funkcja zwraca wartość symbolu newtona dla danego **n** i **k**.

*Zadanie można optymalizować podobnie jak poprzednie, wykorzystując listę dwuwymiarową lub słownik. Lista zajmuje* ***n \* k*** *pamięci, a słownik tyle, ile razy funkcja się wywoła. W przypadku użycia słownika, kluczem może być krotka* ***(n,k)****.*

**Zadanie 3.** Liczby rzymskie, rekurencyjnie

Aby przeliczyć liczbę rzymską na liczbę arabską musimy dodać do wyniku wartość zapisaną jedną lub dwoma pierwszymi liczbami oraz wartość pozostałych znaków. Proces powtarzamy aż do konwersji wszystkich znaków

Funkcja **convert(roman\_number)** konwertuje liczbę **roman\_number** podaną jako string wykorzystując rekurencje.

Początkowo:

**wynik = 0**

**rzym** jest słownikiem, którego kluczem jest liczba rzymska, a wartością jej odpowiednik arabski w postaci liczby całkowitej

|  |  |
| --- | --- |
| **MCMLXIX** | rzym[M] = 1000  return rzym[M] + convert(„CMLXIX”) |
| **CMLXIX** | rzym[CM] = 900  return rzym[CM] + convert(„LXIX”) |
| **LXIX** | rzym[L] = 50  return rzym[L] + convert(„XIX”) |
| **XIX** | rzym[X] = 10  return rzym[X] + convert(„IX”) |
| **IX** | rzym[IX] = 9  return rzym[IX] + convert(„”) |
|  | return 0 |

Najpierw rekurencja „wejdzie” do najgłębszego poziomu, następnie będzie sumować wyliczone wartości 0+9+10+50+900+1000.

Przejdźmy do kodu :

rzym = {**…**}  
  
**def** convert(roman\_number):  
 **if** len(roman\_number) == 0:  
 **return** 0  
 **if** len(roman\_number)>1:  
 **if** roman\_number[0] + roman\_number[1] **in** rzym:  
 **return** rzym[roman\_number[0] + roman\_number[1]] + convert(roman\_number[2:])  
 **return** rzym[roman\_number[0]] + convert(roman\_number[1:])

Tworzę słownik **rzym** którego kluczami są liczby rzymskie, a wartościami ich odpowiedniki arabskie (słownik skopiowany z UPEL’a)

Definiuje funkcję **convert(roman\_number)**, argumentem jest liczba rzymska typu string.

Jeżeli string **roman\_number** jest pusty (długość = **0**) funkcja zwraca **0**. brak znaków = brak wartości.

W przypadku gdyby **roman\_number** miał długość **1**, wtedy **roman\_number[1]** zwróciłby błąd- nie można odwołać się do indexu większego niż długość tablicy. Sprawdzenie długości wcześniejszym **if**’em chroni przed tym błędem (**len(roman\_number) > 1**).

Jeżeli długość **roman\_number** jest większa od **1** sprawdzam czy pierwsze dwa znaki są liczbą rzymską (istnieje taki klucz w słowniku).

Jeżeli dwa pierwsze znaki są liczbą rzymską zwracam sumę ich wartości odczytaną ze słownika oraz wywołuje rekurencyjnie convert(**roman\_number[2:]**) .

Wynikiem **roman\_number[2:]** jestnapis **roman\_number** bez dwóch pierwszych znaków.

W każdym innym przypadku (długość = **1** lub dwa pierwsze znaki nie są liczbą) funkcja zwraca sumę wartości jednego znaku oraz **convert(roman\_number[1:])**

Ostatecznie wynikiem wywołania **convert(roman\_number)** jest wartość **roman\_number** zapisana liczbami arabskimi.

**Zadanie 3** Liczby Rzymskie, iteracyjnie

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| **M** | **C** | **M** | **L** | **X** | **I** | **X** | **#** |
| **0** | **1** | **2** | **3** | **4** | **5** | **6** | **7** |
| **p** | **p+1** |  |  |  |  |  |  |

**p** „wskazuje” na kolejny znak **roman\_number**

Sprawdzamy czy znak na pozycji **p** ze znakiem na pozycji **p+1** jest liczbą (czyli jest taki klucz w słowniku)

W przykładzie „MC” nie jest kluczem. Przeliczamy tylko wartość znaku na pozycji **p**. Zwiększamy  
 **p += 1** przechodząc do kolejnego nie przeliczonego znaku.

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| **M** | **C** | **M** | **L** | **X** | **I** | **X** | **#** |
| **0** | **1** | **2** | **3** | **4** | **5** | **6** | **7** |
|  | **p** | **p+1** |  |  |  |  |  |

Teraz znak na pozycji **p** ze znakiem na pozycji **p+1** tworzą liczbę (CM = 900). Dodajemy przeliczoną wartość do wyniku i zwiększamy **p+=2** żeby przejść do kolejnego nieprzeliczonego znaku (na pozycji 3)

Przed rozpoczęciem dodaliśmy na końcu liczby znak **„#”** (wartownik). Razem z poprzedzającym znakiem na pewno nie stanowi liczby, więc nie musimy się martwić, że komputer spróbuje go przeliczyć. Umożliwia on odwołanie się do ostatniej cyfry rzymskiej, jednocześnie sprawdzając pozycję **p+1**, bez ryzyka wyjścia poza zakres.

|  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
| **M** | **C** | **M** | **L** | **X** | **I** | **X** | | **#** |
| **0** | **1** | **2** | **3** | **4** | **5** | **6** | **7** | |
|  |  |  |  |  |  | **p** | **p+1** | |

**Zadanie 3.** Liczby Rzymskie, rekurencja ogonowa

Do zadania można podejść w inny sposób. Początkowo ustawiamy wartość zmiennej wynik na 0. Badamy po jednej liczbie:

* Jeżeli badana liczba jest mniejsza niż kolejna, odejmujemy jej wartość od wyniku

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| **M** | **C** | **M** | **L** | **X** | **I** | **X** |

*M(1000) > C (100), wynik + 1000*

* Jeżeli badana liczba jest większa niż kolejna, odejmujemy jej wartość od wyniku

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| **M** | **C** | **M** | **L** | **X** | **I** | **X** |

*C(1000) < M(100), wynik - 100*

* Jeżeli liczba jest ostatnia dodajemy jej wartość do wyniku

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| **M** | **C** | **M** | **L** | **X** | **I** | **X** |

*wynik + 10*

Przejdźmy do kodu:

**def** convert(roman\_number, current\_result):  
 **if** len(roman\_number) == 1:  
 **return** current\_result + rzym[roman\_number[0]]  
 **return** convert(roman\_number[1:], current\_result + math.copysign(rzym[roman\_number[0]], rzym[roman\_number[0]] - rzym[roman\_number[1]]))

*Uwaga! Drugi return (4,5,6 linijka), jest jedną linią tekstu*

Definiuje **funkcję convert(roman\_number, current\_result)**

* **roman\_number** to ciąg znaków zawierający cyfry rzysmkie
* **current\_result** to liczba będąca sumą wartości wcześniej przeliczonych liczb

Jeżeli długość **roman\_number = 1** (pozostała ostatnia cyfra) funkcja zwraca sumę wszystkich poprzednich liczb i wartość ostatniej liczby

W innym przypadku wywołuję rekurencyjnie funkcję **convert** z liczbą rzymską bez pierwszej cyfry oraz wynikiem zwiększonym lub zmniejszonym o aktualnie przeliczaną cyfrę

Funkcja zwraca przeliczoną liczbę.

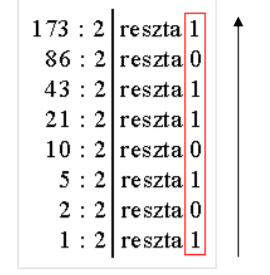
*math.copysing(a,b) zwraca liczbę a \* signum(b), czyli liczbę a ze znakiem liczby b*

*Zatem:*

*Jeżeli aktualnie przeliczana cyfra jest większa niż następna to ich różnica jest dodatnia. Do wyniku zostanie dodana wartość liczby.*

*Jeżeli aktualnie przeliczana cyfra jest mniejszaniż następna to ich różnica jest dodatnia. Do wyniku zostanie dodana liczba przeciwna do aktualnie przeliczanej, czyli od wyniku zostanie odjęta aktualnie przeliczana liczba.*

**Zadanie 4.** Binarna

Algorytm przeliczania liczby dziesiętnej na binarną wygląda następująco:

Dzielimy liczbę całkowicie przez 2, spisujemy resztę z prawej strony, iloraz pod liczbą.

Algorytm powtarzamy dopóki dzielona liczba jest większa od 0.

Czytamy przeliczoną liczbę „od dołu”

Implementacja:

**def** convert(n):  
 **if** n == 1:  
 **return "1"  
 else**:  
 **return** convert(n//2) + str(n%2)

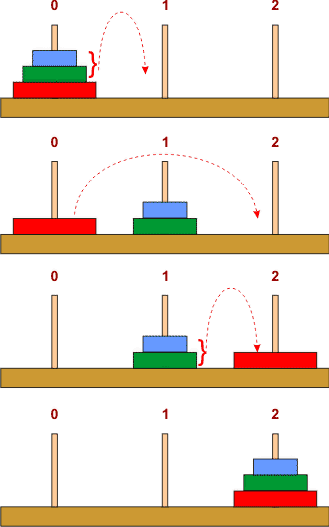
Definiuje funkcję **convert(n),** która jako argument przyjmuje liczbę **n**.

Dla **n = 1** funkcja zwraca „**1**” typu string

Dla **n > 1** funkcja zwraca napis złożony z wyniku **convert(n//2)** z dopisaną resztą z dzielenia **n** przez dwa na końcu.

Ostatecznie funkcja rekurencyjna zwraca liczbę **n** zapisaną w systemie dwójkowym. Wynik jest typu string.

**Zadanie 5.** Hanoi



Rozważmy wieżę Hanoi składającą się z trzech krążków.

Paliki mają swoje funkcje- początkowy, pomocniczy i docelowy

Aby przełożyć trzy elementową wieżę musimy najpierw przełożyć dwu elemenową wieżę na palik pomocniczy, następnie przełożyć jeden element na palik docelowy, a na końcu przełożyć dwuelementową wieżę na palik docelowy.

Aby przekładać dwuelementową wieżę musimy najpierw przełełożyć pierwszy krążek na palik pomocniczy, drugi na palik docelowy i pierwszy położyć na palik docelowy.

Przekładanie wieży o wysokości 1 jest operacją podstawową, którą możemy zapisywać jako część wyniku.

Zatem operacja przenoszenia wieży o wysokości N wygląda następująco:

* *przenieś N-1 górnych krążków z palika początkowego na palik pomocniczy*
* *przenieś N-ty krążek na palik docelowy*
* *przenieś N-1 górnych krążków z palika pomocniczego na docelowy*

Jej implementacja w Pythonie wygląda tak:

**def** hanoi(n, start, target, support):  
 **if** n == 1:  
 **return** str(start) + **" -> "** + str(target) + **"\n"  
 if** n > 1:  
 **return** hanoi(n-1, start, support, target) \  
 + hanoi(1, start, target, support) \  
 + hanoi(n-1, support, target, start)

Definiuje funkcję hanoi przyjmującą argumenty:

* **n**- wysokość przenoszonej wieży
* **start**- palik, na którym znajdują się krążki
* **target**- palik, na którym mają znaleźć się krążki
* **support**- palik pomocniczy

Operacje przeniesienia pierwszego z wierzchu krążka zapisuje:

**start** -> **target**

np. 1 -> 3 mówi, że górny krążek z pierwszego palika przenoszę na trzeci palik

Na początku sprawdzam czy wysokość akturalnie rozpatrywanej wieży jest równa **1** (**n = 1)**. Jeżeli tak, przenoszę krążek z aktualnego palika na palik docelowy.

Jeżeli **n >1** zwracam:

1. Znalezione rekurencyjnie operacje potrzebne do przeniesienia wieży o wysokości **n-1** na palik pomocniczy
2. Operacje potrzebną do przeniesienia krążka na palik docelowy
3. Znalezione rekurencyjnie operacje potrzebne do przeniesienia wieży o wysokości **n-1** z palika pomocniczego na palik docelowy

Program zwraca listę kroków potrzebnych do przeniesienia Wieży Hanoi jako tekst (typ string).