Jakub Rynio | nr albumu: 117323
Informatyka niestacjonarna
semestr: 3
Grupa laboratoryjna: 2

Algorytmy i złożoność:
Iteracja i Rekurencja

Wstęp

Iteracja i rekurencja to dwa podstawowe podejścia stosowane w programowaniu do rozwiązywania problemów i implementacji algorytmów. Iteracja opiera się na wykorzystaniu struktur kontrolnych, takich jak pętle (np. for czy while), które pozwalają na wielokrotne wykonanie określonego fragmentu kodu, aż do spełnienia określonego warunku. Z kolei rekurencja polega na wywoływaniu funkcji przez samą siebie, co pozwala na rozwiązywanie problemów poprzez ich podział na mniejsze, podobne do siebie podproblemy. Oba podejścia różnią się pod względem struktury, wydajności, złożoności implementacji i zastosowań, dlatego ich porównanie pozwala lepiej zrozumieć, kiedy i dlaczego warto zastosować jedno z nich.

Algorytm generujący wyrazy ciągu fibonacciego napisany iteracyjnie

```python
import time
import os

def error():
    print("Nieprawidlowe dane wejsciowe")
    input()
    os.system("cls")

def get_number(prompt, return_type="float", only_poz = False, zero = True):
    while True:
        try:
            if return_type == "float":
                output = float(input(prompt))

            elif return_type == "int":
                output = int(input(prompt))
        except ValueError:
            error()
            continue

        if output < 0 and only_poz or output == 0 and zero == False:
            error()
            continue
        else:
            return output

def fibonacci_iterative(n):
    start_time = time.time()
    a, b = 0, 1
    for _ in range(n):
        print(a)
        a, b = b, a + b
    end_time = time.time()
    print(f"\nCzas wykonania (iteracyjnie): {end_time - start_time} sekundy")

while True:
    liczba_wyrazow = get_number("Podaj dlugosc ciagu: ", "int", True, False)
    fibonacci_iterative(liczba_wyrazow)
    input()
```

```
40      os.system("cls")
```

Algorytm generujący wyrazy ciągu fibonacciego napisany rekurencyjnie

```
1   import time
2   import os
3
4   def error():
5       print("Nieprawidlowe dane wejsciowe")
6       input()
7       os.system("cls")
8
9   def get_number(prompt, return_type="float", only_poz = False, zero = True):
10      while True:
11          try:
12              if return_type == "float":
13                  output = float(input(prompt))
14
15              elif return_type == "int":
16                  output = int(input(prompt))
17          except ValueError:
18              error()
19              continue
20
21          if output < 0 and only_poz or output == 0 and zero == False:
22              error()
23              continue
24          else:
25              return output
26  def fibonacci_recursive(n, a=0, b=1):
27      start_time = time.time()
28      if n <= 0:
29          return
30      print(a)
31      fibonacci_recursive(n - 1, b, a + b)
32      end_time = time.time()
33      if n == 10:
34          print(f"\nCzas wykonania (rekurencyjnie): {end_time - start_time} sekundy")
35
36  while True:
37      liczba_wyrazow = get_number("Podaj dlugosc ciagu: ", "int", True, False)
38      fibonacci_recursive(liczba_wyrazow)
39      input()
40      os.system("cls")
```

Zalety i wady implementacji iteracyjnej i rekurencyjnej algorytmu obliczającego ciąg Fibonacciego

Podejście iteracyjne

Zalety:

- **Efektywność pamięciowa:** Iteracyjne obliczenia nie wykorzystują stosu wywołań, dzięki czemu nie zajmują dodatkowej pamięci dla każdego wywołania funkcji. W pamięci przechowywane są jedynie bieżące wartości (np. a i b).
- **Szybkość wykonania:** Operacje są wykonywane sekwencyjnie w pętli, co jest szybsze od rekurencji z powodu braku narzutu związanego z wywoływaniem funkcji.
- **Bezpieczeństwo:** Brak ryzyka przepełnienia stosu (ang. *stack overflow*), co może wystąpić w przypadku rekurencji przy dużych wartościach n.

Wady:

- **Kod mniej intuicyjny:** Dla niektórych problemów algorytmy iteracyjne mogą być trudniejsze do zrozumienia niż ich rekurencyjne odpowiedniki, szczególnie w przypadku bardziej złożonych logik.
- **Brak elegancji matematycznej:** W wielu przypadkach rekurencja lepiej odwzorowuje naturalną definicję matematyczną problemu, co sprawia, że iteracja wydaje się mniej naturalna.

Podejście rekurencyjne

Zalety:

- **Prostota i elegancja kodu:** Kod rekurencyjny jest bardziej zwięzły i łatwiejszy do napisania, szczególnie w przypadku problemów, które mają naturalną definicję rekurencyjną, takich jak ciąg Fibonacciego.
- **Zgodność z definicją matematyczną:** Rekurencyjne rozwiązanie przypomina matematyczną definicję ciągu Fibonacciego, co czyni kod bardziej przejrzystym i intuicyjnym dla czytelnika znającego podstawy matematyki.

Wady:

- **Zwiększone użycie pamięci:** Każde wywołanie rekurencyjne powoduje zapisanie bieżącego stanu funkcji na stosie wywołań. Dla dużych wartości n, stos wywołań może się przepełnić, co skutkuje błędem pamięci (*stack overflow*).
- **Niska efektywność czasu wykonania:** Przy prostym podejściu rekurencyjnym bez pamiętania wyników (ang. *memoization*), funkcja może wykonywać wiele zbędnych obliczeń, ponieważ każde wywołanie oblicza te same wartości wielokrotnie.
- **Ryzyko błędów przy dużych wartościach n:** Przekroczenie dostępnej głębokości stosu prowadzi do błędu programu, szczególnie w językach, które mają ograniczenia co do głębokości rekurencji (np. Python).

Porównanie podejść

Kryterium	Iteracyjne	Rekurencyjne
Wydajność czasowa	Szybsze	Wolniejsze, szczególnie przy dużym n
Zużycie pamięci	Minimalne	Wysokie przy dużym n
Czytelność kodu	Mniej intuicyjne	Bardziej eleganckie
Przepełnienie stosu	Nie występuje	Może wystąpić
Intuicyjność	Bardziej naturalne	Mniej naturalne

Porównanie prędkości

Liczba wyrazow ciagu	100	998	10000
Iteracje	0	0.028952360153198242	0.9479048252105713
Rekuręcja	0	0.0010390281677246094	RecursionError: maximum recursion depth

Możliwy błąd pamięci w rekurencji

Rekurencja w Pythonie jest ograniczona przez maksymalną głębokość stosu, która domyślnie wynosi około 1000 (można ją zmienić, ale wiąże się to z ryzykiem). Jeśli n przekroczy tę wartość:

- Każde wywołanie rekurencyjne zajmuje miejsce na stosie wywołań.
- Przy zbyt dużej głębokości, program napotka błąd przepełnienia stosu (ang. `RecursionError: maximum recursion depth exceeded`).

Przykład problemu:

Dla $n = 10^6$, każda funkcja rekurencyjna wywołuje kolejną, co może prowadzić do wyczerpania dostępnej pamięci stosu. Dlatego w takich przypadkach preferuje się iteracyjne podejście lub optymalizację rekurencji z memoizacją.

Wniosek

W praktycznych zastosowaniach, szczególnie dla dużych wartości n, podejście iteracyjne jest znacznie bardziej wydajne i bezpieczne. Rekurencja może być używana dla mniejszych danych wejściowych lub gdy priorytetem jest prostota i czytelność kodu. Aby połączyć zalety obu podejść, można zastosować techniki takie jak *rekurencja z memoizacją* lub algorytmy dynamiczne, które eliminują redundantne obliczenia.