

(opcjonalnie) Wprowadzenie do rekursji

Jeśli dla uczniów jest to pierwszy kontakt z tym pojęciem - wymaga to osobnej lekcji, którą podajemy poniżej (w skrócie). Jeśli uczniowie już znają rekursję, można przypomnieć tylko punkt 4.

1. Definicja silni, obliczanie metodą iteracyjną
 - *Prosta pętla, uczniowie dochodzą do niej sami*
2. Co “w praktyce” dzieje się, kiedy wywołujemy w programie jakąkolwiek funkcję/ podprocedurę?
 - *Aktualny stan programu jest w całości zapamiętywany i odkładany na bok - na tzw. stos wywołań;*
 - *Program przeskakuje do funkcji, wstawia odpowiednie wartości w miejsce argumentów i wykonuje cały kod funkcji;*
 - *Kiedy już obliczy wartość funkcji, wraca dokładnie w miejsce, gdzie był - tam, gdzie nastąpiło wywołanie funkcji.*
3. Wywołanie rekurencyjne w silni - jak zadziała?
 - *Dokładnie tak samo, jak poprzednio - nie ma znaczenia dla programu, że wykonuje się wielokrotnie ta sama funkcja.*
 - *Piszemy na tablicy rekurencyjną funkcję $silnia()$ i dokładnie symulujemy działanie $silnia(4)$. Podkreślamy, że każde wywołanie $silnia()$ jest oddzielną “kopią” funkcji, ma własne zmienne i nie ingeruje na pozostałe wywołania - dopiero kiedy skończy, podaje swój wynik poprzedniemu wywołaniu.*
4. Kiedy (i jak) w ogólności działa rekursja
 - *Rekurencja wymaga trzech warunków do działania:*
 - *wywołanie istotnie następuje na mniejszych danych (przejdzie z $silnia(4)$ do $silnia(3)$, a nie np. do $silnia(6)$).*
 - *wiemy, jak z wyniku dla mniejszej liczby odtworzyć wynik dla większej (przejdzie $silnia(3)$ -> $silnia(4)$ jest łatwe).*
 - *znamy początek rekursji (umiemy obliczyć $silnia(0)$ bez wywołania rekurencyjnego).*

Szybkie potęgowanie

1. (Opcjonalnie) Przypominamy pojęcie potęgowania, zapisujemy iteracyjną procedurę $potęga(a,k)$, zwracającą a^k .
2. Zapisujemy rekurencyjną wersję procedurę $potęga(a,k)$.

funkcja $potęga(a, k)$

jeżeli $k = 0$

zwróć wynik 1

$w = potęga(a, k-1)$

zwróć wynik $a*w$

3. Dla przykładowych danych (np. $a = 2$ i $k = 20$) rysujemy schemat kolejnych wywołań i powrotów, jakie zostaną wykonane w programie, np. w takiej postaci:

$$2^{20} \rightarrow 2^{19} \rightarrow 2^{18} \rightarrow \dots \rightarrow 2^2 \rightarrow 2^1 \rightarrow 2^0.$$

$$2^0 = 1 \rightarrow 2^1 = 2 \rightarrow 2^2 = 4 \rightarrow \dots \rightarrow 2^{20} = 1\,048\,576$$

4. Ten program działa dlatego, że łatwo jest obliczyć 2^{20} mając 2^{19} , i ogólnie 2^k mając 2^{k-1} . Zauważmy jednak, że można obliczyć 2^{20} mając 2^{10} , podnosząc tę ostatnią liczbę do kwadratu - i analogicznie, jeśli k jest liczbą parzystą, obliczyć 2^k mając $2^{k/2}$:

jeśli k jest parzyste:
 $w = \text{potęga}(a, k/2)$
zwróć wynik $w * w$

Co jednak, jeśli k jest nieparzyste? Wtedy postępujemy jak poprzednio:

jeśli k jest nieparzyste:
 $w = \text{potęga}(a, k-1)$
zwróć wynik $a * w$

5. Pełny kod funkcji:

funkcja $\text{potęga}(a, k)$
jeżeli $k = 0$
zwróć wynik 1
jeśli k jest parzyste:
 $w = \text{potęga}(a, k/2)$
zwróć wynik $w * w$
jeśli k jest nieparzyste:
 $w = \text{potęga}(a, k-1)$
zwróć wynik $a * w$

Zapisujemy na tablicy nową wersję algorytmu i analizujemy łańcuch wywołań (i powrotów) rekurencyjnych na tym samym przykładzie:

$$2^{20} \rightarrow 2^{10} \rightarrow 2^5 \rightarrow 2^4 \rightarrow 2^2 \rightarrow 2^1 \rightarrow 2^0.$$

6. Złożoność procedury - ile nastąpi wywołań rekurencyjnych? Wywołania "parzyste" dwukrotnie zmniejszają k . Gdyby więc były tylko takie wywołania, byłoby ich nie więcej niż $\log(k)$. Są jednak jeszcze wywołania nieparzyste - dlaczego nie ma ich zbyt wiele?

7. Zauważamy, że nigdy nie będzie dwóch wywołań “nieparzystych” z rzędu. A więc nie może ich być więcej niż parzystych. Skoro parzystych jest $\log(k)$, to w sumie będzie nie więcej niż $2 \log(k)$. *(Opcjonalnie) To dobry przykład ilustrujący użycie notacji $O()$, jeśli chcemy ją wprowadzać - zapis $O(\log k)$ będzie w tym wypadku oznaczał “co najmniej $\log(k)$, ale co najwyżej $2 \log(k)$ ”.*
8. Zadanie do samodzielnego zaprogramowania przez uczniów: *“Mając dane liczby całkowite a , k i p , należy obliczyć resztę z dzielenia a^k przez p ”.*
- **Uwaga:** Nie można najpierw obliczyć a^k , a potem podzielić z resztą przez p - liczba a^k na ogół wychodzi jest za duża, żeby zmieściła się w typie całkowitym (w C++), i za duża, żeby się dało na niej szybko prowadzić obliczenia (w Pythonie/Javie).
 - Trzeba wprowadzić podstawy arytmetyki modularnej. Zalecamy najpierw tłumaczenie na wersji iteracyjnej, potem na rekurencyjnej - w obu ostatecznie jest tak samo: po każdym mnożeniu wyciągamy resztę modulo p .
 - Typowe pytanie uczniów piszących w C++: czemu nie można użyć funkcji `pow()`: bo wynik jest dużą liczbą rzeczywistą, w dodatku zapisaną niedokładnie - zilustrować przykładem). W Pythonie również problematyczne jest $a^{**}k$ - ta liczba ma wiele cyfr i operacje na niej będą zbyt wolne.