Zajęcia 3, 4

30 listopada 2025

Treści programowe

Na zajęciach omawiamy następujące zagadnienia z sylabusa:

> **4. Rekurencja i programowanie dynamiczne**
> Pojęcie rekurencji, przykłady procedur rekurencyjnych, własność optymalnej podstruktury, programowanie dynamiczne, przykłady algorytmów wykorzystujących programowanie dynamiczne.

> **5. Analiza algorytmów**
> Notacja asymptotyczna, złożoność czasowa i pamięciowa algorytmu, złożoność optymistyczna, pesymistyczna i średnia, twierdzenie o rekurencji uniwersalnej, klasy złożoności, dowodzenie poprawności konstrukcji wybranych algorytmów.

Do samodzielnego przerobienia w domu:

> **6. Algorytmy sortowania**
> Sortowanie przez wstawianie, bąbelkowe, przez scalanie, szybkie, przez zliczanie.

Plan zajęć

(i) **Rekurencja:** Funkcje rekursyjne to funkcje, które wywołują same siebie. Klasycznym przykładem jest silnia. Dla liczby naturalnej n definiujemy silnię $n!$ jako:

$$n! = 1 \cdot 2 \cdot 3 \cdot \dots \cdot (n-1) \cdot n = \prod_{k=1}^{n} k.$$

Łatwo zauważyć, że

$$n! = (n-1)! \cdot n,$$

a to oznacza, że funkcję obliczającą silnię możemy zaimplementować w następujący sposób:

```
1  def factorial(n: int) -> int:
2      if n == 0: return 1
3      return n * factorial(n-1)
```

Przypadek `n==0` nazwiemy **przypadkiem bazowym** – definiuje on moment, w którym rekurencja powinna się zakończyć. Przypadek `n != 0` nazwiemy **przypadkiem rekurencyjnym** – zmniejsza on "rozmiar" problemu (zamiast $n!$ liczymy $(n-1)!$).

Zadanie 1, 2AB

(ii) **Własność optymalnej podstruktury:**

W wielu problemach algorytmicznych zadania rozbijamy na mniejsze *podproblemy*.

> **Definicja**
> Problem ma **własność optymalnej podstruktury**, jeżeli jakieś rozwiązanie jest **optymalne** dla całego problemu, to każdy jego fragment, który sam jest rozwiązaniem mniejszego podproblemu, jest jego **optymalnym** rozwiązaniem.

Innymi słowy: nie da się wziąć fragmentu optymalnego rozwiązania, poprawić go lokalnie i w ten sposób poprawić globalnego wyniku — inaczej rozwiązanie nie było optymalne.

> Rozważmy najkrótszą drogę z miasta A do miasta D. Jeśli globalnie najkrótsza trasa to
> $$A \to B \to C \to D,$$
> to podtrasa $A \to B \to C$ musi być najkrótszą możliwą drogą z A do C. Gdyby istniała krótsza, moglibyśmy ją podmienić i dostać jeszcze krótszą trasę z A do D.

Własność optymalnej struktury jest istotna w programowaniu dynamicznym – zazwyczaj możemy ją wykorzystać, żeby znaleźć algorytmy rozwiązujące problemy o takiej własności.

Zadanie 2C

(iii) **Programowanie dynamiczne:** Kiedy rozbijamy problem na podproblemy, często dokładnie te same podproblemy pojawiają się wielokrotnie – obliczanie ich od nowa za każdym razem może okazać się kosztowne. Programowanie dynamiczne polega na zapisaniu zależność między podproblemami, obliczeniu ich dokładnie raz i zapisaniu wyników w pamięci.

Na przykład, obliczając n-ty wyraz ciągu Fibonacciego rekurencyjnie w naiwny sposób, często będziemy powtarzać ten sam krok wielokrotnie. Skoro

$$F_n = F_{n-1} + F_{n-2} = F_{n-2} + F_{n-3} + F_{n-2},$$

to wyraz F_{n-2} zostanie obliczony dwukrotnie. Liczba podproblemów rośnie wykładniczo z wielkością n.

Natomiast, jeśli zaczniemy od $F_0 = 1$, $F_1 = 1$, a ostatnie dwa obliczone wyrazy będziemy przechowywać w pamięci, to każdy wyraz obliczymy dokładnie raz – wynik dla n otrzymamy dużo szybciej.

Dla porównania, funkcja `f` oblicza F_{40} w ok. 13 sekund, funkcja `g` w 0.000002 sekundy:

```python
def f(n: int):
    if n <= 1:
      return 1
    else:
      return f(n - 1) + f(n - 2)
```

```python
def g(n):
    if n <= 1: return 1
    p2, p1 = 1, 1
    for _ in range(2, n + 1):
        curr = p1 + p2
        p2, p1 = 1, curr
    return p1
```

Zadanie 2ED

(iv) **Złożoność algorytmiczna:** Obliczenia kosztują: czas, który możemy poświęcić na obliczenia jest ograniczony, pamięć komputera też jest ograniczona. Złożoność algorytmiczna to zbiór technik pozwalających precyzyjnie wyrazić czasowy i pamięciowy koszt działania procedur.

Koszt działania będzie zależał od wielkości danych wejściowych – ich rozmiar oznaczamy przez n. Złożoność czasową opisujemy funkcją $T(n)$, która mówi, ile operacji algorytm wykona dla danych o rozmiarze n – koszt przejścia przez tablicę o długości n będzie proporcjonalny do n, podczas gdy koszt przejścia przez dwuwymiarową tablicę o rozmiarze $n \times n$ będzie proporcjonalny do n^2, itd.

Zadanie 3

(v) **Złożoność asymptotyczna:** Komputery są szybkie. Zwykle nie interesuje nas, czy złożoność algorytmu to $T(n) = f(n)$ czy $T(n) = f(n) + 10000$ (gdzie $f : \mathbf{N} \to \mathbf{N}$ to *jakaś* funkcja), ponieważ ostateczna różnica w czasie jest znikoma. Tak samo zwykle nie ma większego znaczenia, czy jest to $f(n)$ czy $10 \cdot f(n)$. Istotny jest natomiast typ złożoności – czy $T(n)$ zachowuje się jak logarytm, funkcja wykładnicza, wielomian (jeśli tak, to wielomian którego stopnia), itd.

Podobnie nie ineteresują nas specjalnie wartości $T(n)$ dla małych rozmiarów danych, ponieważ jeśli n jest małe, to $T(n)$, niezależnie czy jest ono postaci $n^{10} + n^5 + 1$, $n^5 \log(n)$ czy $100 \exp(n)$ z reguły również jest małe – a na pewno mniejsze niż dla dużych n. Dlatego badamy złożoność **asymptotyczną** – wartości $T(n)$ dla dużych n.

- Górne oszacowanie: $T(n) = O(f(n))$ – dla dużych n wartość $T(n)$ rośnie co najwyżej jak $f(n)$
- Dolne oszacowanie: $T(n) = \Omega(f(n))$ – dla dużych n wartość $T(n)$ rośnie co najmniej jak $f(n)$
- Dokładny rząd wzrostu: $T(n) = \Theta(f(n))$, jeśli $T(n) = O(f(b))$ i $T(n) = \Omega(f(n))$

Zadanie 4

(vi) **Twierdzenie o rekurencji uniwersalnej:** Twierdzenie (tzw. *master theorem*) brzmi następująco:

> Niech $a \geq 1$ i $b > 1$ będą stałymi, niech $f(n)$ będzie funkcją nieujemną i niech $T(n)$ będzie zdefiniowane dla nieujemnych liczb całkowitych przez zależność rekurencyjną
>
> $$T(n) = aT\left(\frac{n}{b}\right) + f(n).$$
>
> Wówczas:
>
> (i) Jeśli $f(n) = O\big(n^{\log_b a - \varepsilon}\big)$ dla pewnego $\varepsilon > 0$, to $T(n) = \Theta\big(n^{\log_b a}\big)$.
>
> (ii) Jeśli $f(n) = \Theta\big(n^{\log_b a}\big)$, to $T(n) = \Theta\big(n^{\log_b a} \log n\big)$.
>
> (iii) Jeśli $f(n) = \Omega\big(n^{\log_b a + \varepsilon}\big)$ dla pewnego $\varepsilon > 0$ oraz $af\big(\frac{n}{b}\big) \leq cf(n)$ dla pewnej stałej $c < 1$ i *dostatecznie dużych* n (tzw. *warunek regularności*), to $T(n) = \Theta(f(n))$.

Stała a mówi na ilu podproblemach wołamy algorytm rekurencyjnie, stała b mówi jak zmniejszamy rozmiar problemu, natomiast $f(n)$ to koszt dzielenia problemu i połączenia rozwiązań podproblemów. W skrócie: problem o rozmiarze n dzielimy na a podproblemów o wielkości n/b. Wartość $aT(n/b)$ to koszt działania oddelegowany podproblemom, $f(n)$ to koszt podzielenia danych i połączenia wyników.

Zadanie 5

Zadania

Zadanie 1: Funkcje rekurencyjne. Zdefiniuj funkcje rekurencyjne i zaimplementuj je w języku Python w poniższych przypadkach:

- A. Funkcja `sum(A: list[int]) -> int`, która oblicza sumę listy liczb naturalnych.
- B. Funkcja `is_palindrome(S: str) -> bool`, która sprawdza, czy ciąg znaków jest palindromem.
- C. Funkcja `fib(n: int) -> int`, która oblicza n-ty wyraz ciągu Fibonnaciego.

Zadanie 2: Schody – programowanie dynamiczne. Rozważmy schody zbudowane z $n+1$ stopni ponumerowanych $0, 1, 2, \ldots, n$. Niech `c[i]` będzie listą liczb naturalnych o długości $n+1$. Po schodach możemy wejść pokonując w każdym korku 1 lub 2 stopnie.

- A. Znajdź zależność rekurencyjną określającą liczbę różnych sposobów wejścia po schodach.
- B. Korzystając ze znalezionej zależności, zaimplementuj w Pythonie funkcję `stairs(n: int) -> int`, która oblicza tę liczbę.

Załóżmy, że za każdym razem, gdy staniemy na i-tym stopniu, płacimy koszt równy `c[i]`. Chcemy znaleźć sposób wejścia po schodach, który będzie kosztował nas najmniej, oraz koszt takiego wejścia.

- C. Udowodnij, że taki problem spełnia warunek optymalnej podstruktury.
- D. Za pomocą technik programowania dynamicznego znajdź algorytm rozwiązujący problem.
- E. Zaimplementuj ten algorytm w Pythonie: napisz funkcję `optimal_cost(n: int, c: list[int])`, która zwróci listę kolejnych kroków (tzn. czy wykonujemy krok o `1` czy `2` stopnie) oraz łączny koszt.

Zadanie 3: Złożoność czasowa procedur. Opisz funkcję złożoności czasowej $T(n)$ w każdym z poniższych przypadków. Zakładamy, że jako pojedynczą operację traktujemy przypisanie wartości do zmiennej, wykonanie działania arytmetycznego na dwóch zmiennych oraz zwiększenie indeksu w pętli typu `for i in range(n)`.

Przypadek A

```
x=0
for i in range(n):
  x = x + 3
  x = x * x
  x = x - 1
  x = x / 10
```

Przypadek B

```
x = [] ; sum = 0
for i in range(n):
 for j in range(n):
  x.append(1)
for k in range(len(x)):
  sum = sum + x[k]
```

Zadanie 4: Szacowanie złożoności asymptotycznej. Znajdź *możliwie dokładne* oszacowania typu O, Ω i Θ dla funkcji $T_1(n) = 3n + 7$, $T_2(n) = 3n + \log(n^2 + n + 30) + 5$ oraz $T_3(n) = 2^{\log_2 n + 5}$.

Zadanie 5: Twierdzenie o rekurencji uniwersalnej. Wykorzystując twierdzenie o rekurencji uniwersalnej, oszacuj złożoność asymptotyczną dla następujących zależności rekurencyjnych:

A. $T(n) = 2 \cdot T(n/2) + n$ B. $T(n) = 4 \cdot T(n/2) + n$ C. $T(n) = 4 \cdot T(n/2) + n^3$

D. $T(n) = 3 \cdot T(n/3) + n$ E. $T(n) = 3 \cdot T(n/3) + n^2$ F. $T(n) = 3 \cdot T(n/3) + \log n$.