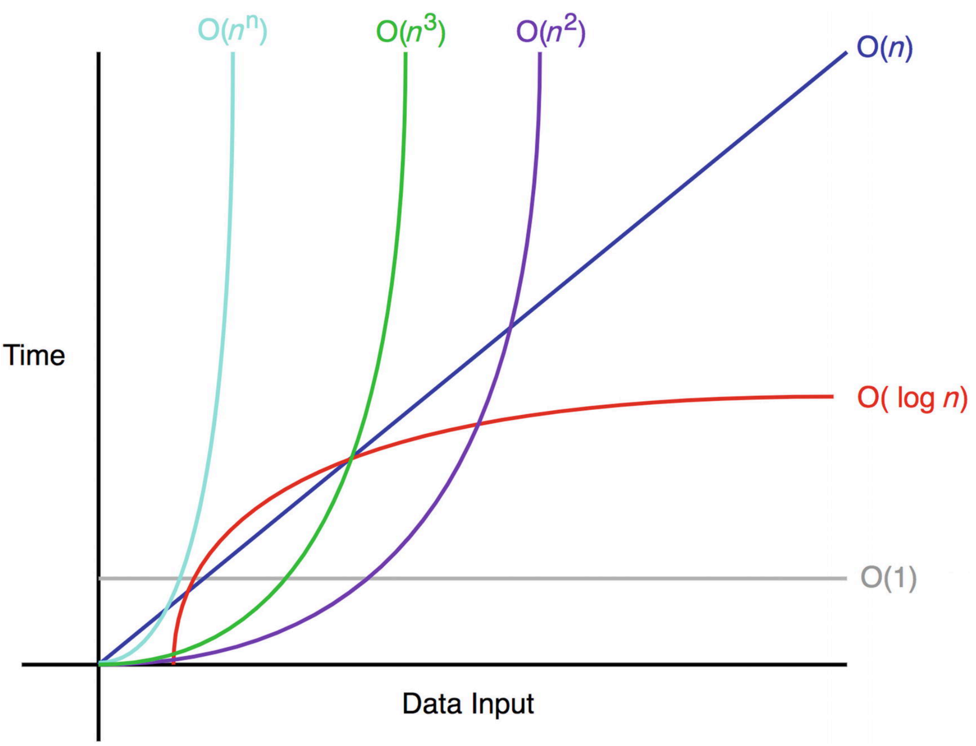
Asymptotické paměťové a časové složitosti
Asymptotická složitost je způsob, kterým měříme efektivitu/výkon algoritmu z hlediska času nebo paměti v závislosti na vstupních datech. Pomáhá nám odhadnout jaký algoritmus zvolit pro daný problém a jak se bude chovat s velkými datovými vstupy.

Druhy asymptotických notací

1. Big O ()
    - představuje horní hranici (nejhorší možný případ) složitosti
    - v praxi nejčastěji používán
    - Bubble sort = O(n^2)

2. Big Ω
    - představuje dolní hranici (nejlepší možný případ) složitosti
    - nepříliš používaný
    - Bubble sort, nejlepší možný scénář, když je všechno už seřazený = Ω(n) - lineární

3. Big Θ
    - přesný asymptotický odhad složitosti
    - udává horní a dolní odhad
    - Merge sort = O(n logn)



Časová složitost
Vyjadřuje počet operací, které algoritmus provede v závislosti na velikosti vstupních dat (n). Například když máme pole o velikosti 10 prvků a pole o velikosti 1000 prvků, časová složitost nám říká, jak se změní doba běhu algoritmu mezi těmito dvěma případy.

Druhy časových složitostí

1. Konstantní O(1)
    - algoritmus provádí stejný počet operací bez ohledu na velikost vstupu
    - nejefektivnější možná složitost
    - Příklad = vyhledání prvku v poli pomocí indexu, basic operace matematický

```
array = [1,2,3,4,5,6,7,8,9,10]
x=array[4]
print(x)
```

2. Logaritmická O(log n)
    - algoritmus v každém kroku zmenšuje problém na menší část
    - efektivní pro velmi velké vstupy
    - Příklad = počítaní číslic v čísle

```
def pocet_cislic(cislo):
   # Pro záporná čísla pracujeme s absolutní hodnotou
   cislo = abs(cislo)

   # Speciální případ pro číslo 0
   if cislo == 0:
       return 1

   pocet = 0
   while cislo > 0:
       pocet += 1  # Přičteme jednu číslici
       cislo = cislo // 10  # Odstraníme poslední číslici

   return pocet
```

3. Lineární O(n)
    - počet operací roste úměrně s velikostí vstupu
    - algoritmus prochází každý prvek vstupu právě jednou
    - Příklad = hledání čísla v nesetřídaném poli

```
array = [5923,12895,195812,581,89512,41289]

def linearn_search(array,x):
   for i in range(len(array)):
       if array[i] ==x:
           return i
```

```
print(linearn_search(array,12895))
```

4. Lineárně logaritmická O(n log n)
    - vyskytuje se převážně u řadících algoritmů typu rozděl a panuj
    - Merge sort….

5. Kvadratická O(n^2)
    - počet operací roste kvadraticky vůči velikosti vstupu
    - Bubble Sort

6. Exponenciální O(k^n)
    - počet operací roste exponenciálně
    - vyskytují se u rekurzivních algoritmů
    - Příklad = rekurzivní výpočet fibonacciho posloupnosti, pro každý problém generují dva nebo více podproblémů

```
def fibonacci(n):
   if n <= 1:
       return n
   return fibonacci(n-1) + fibonacci(n-2)  # Dvě rekurzivní volání
```

7. Faktoriál O(n!)
    - extrémní rychlost počtu růstu operací
    - nejhorší možnost
    - generování permutací, řešení kombinatorických problémů brute forcem
    - lodičky

Paměťová složitost
Narozdíl od časové složitosti, u které je účelem měření rychlosti algoritmu, paměťová složitost určuje, kolik dodatečné paměti algoritmus potřebuje k dokončení svých operací v závislosti na vstupních datech.

Typy paměťové složitosti
1. Konstatní O(1)
    - používá pouze pevně daný počet promenných
    - přistup k prvku v poli

2. Lineární O(n)
    - alokuje paměť úměrnou s velikostí vstupu
    - DFS

3. Lineárně logaritmická O(n log n)
    - rozdělují vstup a potřebují dočasné uložiště

- Merge sort

4. Kvadratická O(n^2)
   - potřebuje mnoho paměti pro uložení mezivýsledků, roste s druhou mocninou velikosti vstupu
   - Matice sousednosti

5. Exponenciální O(k^n)
   - paměťové nároky rostou exponenciálně s velikostí vstupu
   - rekurzivní algoritmy, kde se každé volání rozvětví do dvou nových volání
   - Fibonacciho posloupnost rekurzivně

6. Faktoriál O(n!)
   - rostou faktoriálně s velikostí vstupu
   - algoritmy, které generují všechny možné kombinace
   - Generování permutací