Asymptotická složitost programu

[**https://cw.fel.cvut.cz/b182/\_media/courses/b6b36dsa/dsa-3-slozitostalgoritmu.pdf**](https://cw.fel.cvut.cz/b182/_media/courses/b6b36dsa/dsa-3-slozitostalgoritmu.pdf)

**Asymptotická složitost programu** je způsob, kterým se dá určit **kolik prostředků bude potřeba k dosažení cíle**.

Říká se jí asymptotická, protože se zabývá **výkonností u programu, který může pracovat s neomezeným počtem dat**.

Základní rozlišení složitosti se rozděluje na **časovou** (time complexity)a **paměťovou** (space/memory complexity).

**Časová složitost** **určuje dobu, kterou procesor vyžaduje k vykonání úlohy** (počítá se s tím, že máme neomezenou paměť).

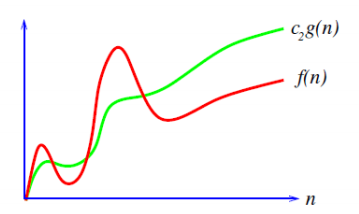
Je důležité zmínit, že jiné zařízení mají jiné parametry a tím pádem se výsledek bude lišit na různých zařízeních a proto se **tyto složitosti rozlišují tím, jak doba vykonání roste se změnou množství dat**.

Poté tu je již zmiňovaná **paměťová složitost**, která určuje **kolik operační paměti bude využito**.

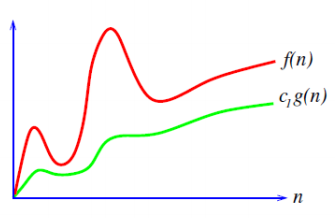
Častá chyba je, že si někdo může říct, že paměťová složitost nelze rozlišit, protože na různých zařízeních bude využívat stejné množství paměti, což sice je pravda, ale stále ji můžeme rozlišit. A rozlišit jí můžeme naprosto stejně jako u časové, akorát se změní otázka na **jak** **paměť vykonání roste se změnou množství dat.**

Tyto složitosti se **dají označit několika způsoby**:

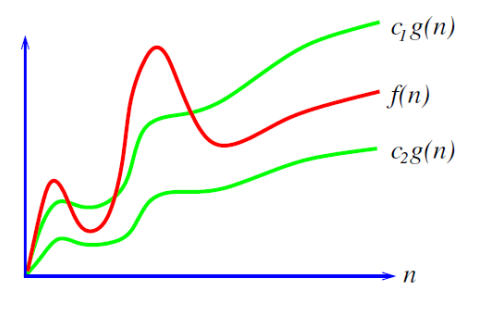
**1.** a nejpoužívanější je **Big O** (výslovnost big ou) a určuje **nejhorší možný případ složitosti**. (Horní mez)



**2.** je **Big Ω** (výslovnost big omega) a určuje **nejlepší možný případ složitosti**. (Dolní mez)



**3.** je **Big Θ** (výslovnost big théta) a určuje **složitost ve vymezených hranicích**, což znamená, že se bude lišit **od perfektní maximálně ve vymezených limitách**. (Máme dvě funkce, jenž jsou stejné až na multiplikativní konstantu) = průměr



\*příklady kódu pro dané složitosti jsou níž v dokumentu\*

Dále se **rozdělují na samotné složitosti**. Nejznámějšími jsou:

1. je **konstantní**, která se označuje **O(1)** a znamená, že **nezáleží na množství dat**. Algoritmus bude trvat stále stejně. Jako příklad si můžeme uvést **vyvolání hodnoty z listu**.

(za předpokladu že máme dynamické pole, bude akce vyvolání prvního prvku a náhodného prvku stejně složitá)

2. je **logaritmická**, která se značí **O(log (n))**.

(Binární vyhledávání půlením intervalu, vzpomeňte si na cvičení z druháku, kde jsme měli za pomoci stromu řešit slova v Morseově abecedě. Tečka byla vždy na větvi do prava, čárka doleva. Vyhledávání v takovém stromu by bylo příkladem logaritmické složitosti.

2.5. je **odmocninová** O(√n)

Příklady algoritmů s odmocninovou složitostí zahrnují algoritmy pro hledání prvočísel (např. Eratosthenovo síto), určování nejkratších cest v grafu (např. algoritmus pro hledání nejkratších cest pomocí Dijkstrova algoritmu s použitím haldy).

Je důležité si uvědomit, že odmocninová složitost může být rychlejší nebo pomalejší než lineární složitost (O(n)), v závislosti na konkrétní implementaci algoritmu a velikosti vstupu.

3. je **lineární**, která se značí **O(n)** a znamená, že **čas/paměť běží úměrně k tomu, jaké množství dat je využito**. Jako příklad si můžeme uvést **for cyklus, jehož hodnota se zvětší o 1 při každém proběhnutí**.

(příkladem je vyvolávání prvku ve spojovém seznamu, kde se musíme k prvku dostat skrze všechny předcházející prvky)

4. je **kombinace lineární a logaritmické (Linearitmická)**, která se značí **O(n log (n))**. Příkladem může být Merge sort nebo Quick sort. („Rozděl a panuj“)

5. je **kvadratická**, která se značí **O(n²)**.

(Uvažujte cyklus, který se nachází uvnitř dalšího cyklu, přičemž oba cykly jsou identické. Příkladem je Bubble sort)

6. je **exponenciální**, která se značí **O(2ⁿ)**. Tuto složitost mají **algoritmy, jejichž složitost se zdvojnásobí při každém dalším prvku**. Jako příklad si můžeme uvést **funkce pro výpočet Fibonacciho sekvence**.

7. je **faktoriální**, která se značí **O(n!)**. Tuto složitost mají většinou **brute force řešení**. Jako příklad si můžeme uvést **for cyklus, který beží n!-krát**.

Důležité je zmínit, že např. v konstantní složitosti znaménko 1 neznamená, že program bude běžet 1ms, nebo že využije 1 MB paměti.

//Dále při např. lineární složitosti nezapisujeme např. O(2\*n)/O(5+10\*n), protože při jakémkoliv množství dat bude lineární.//

## Asymptotická složitost programu

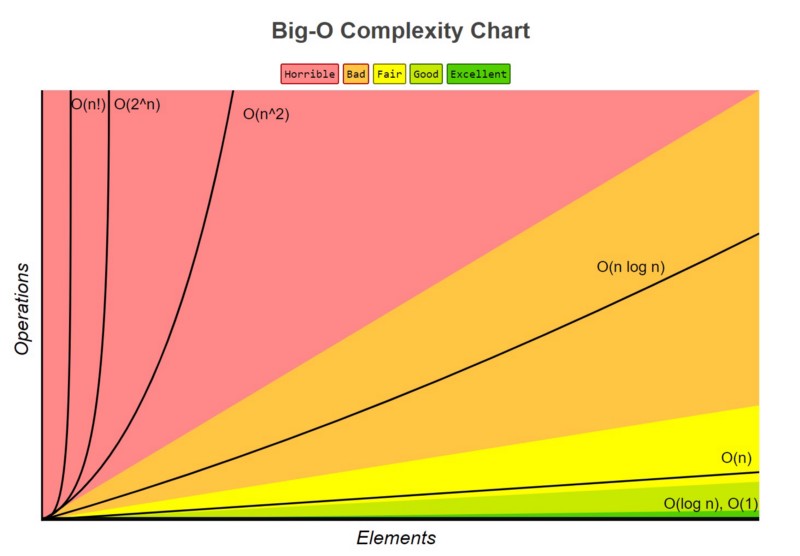
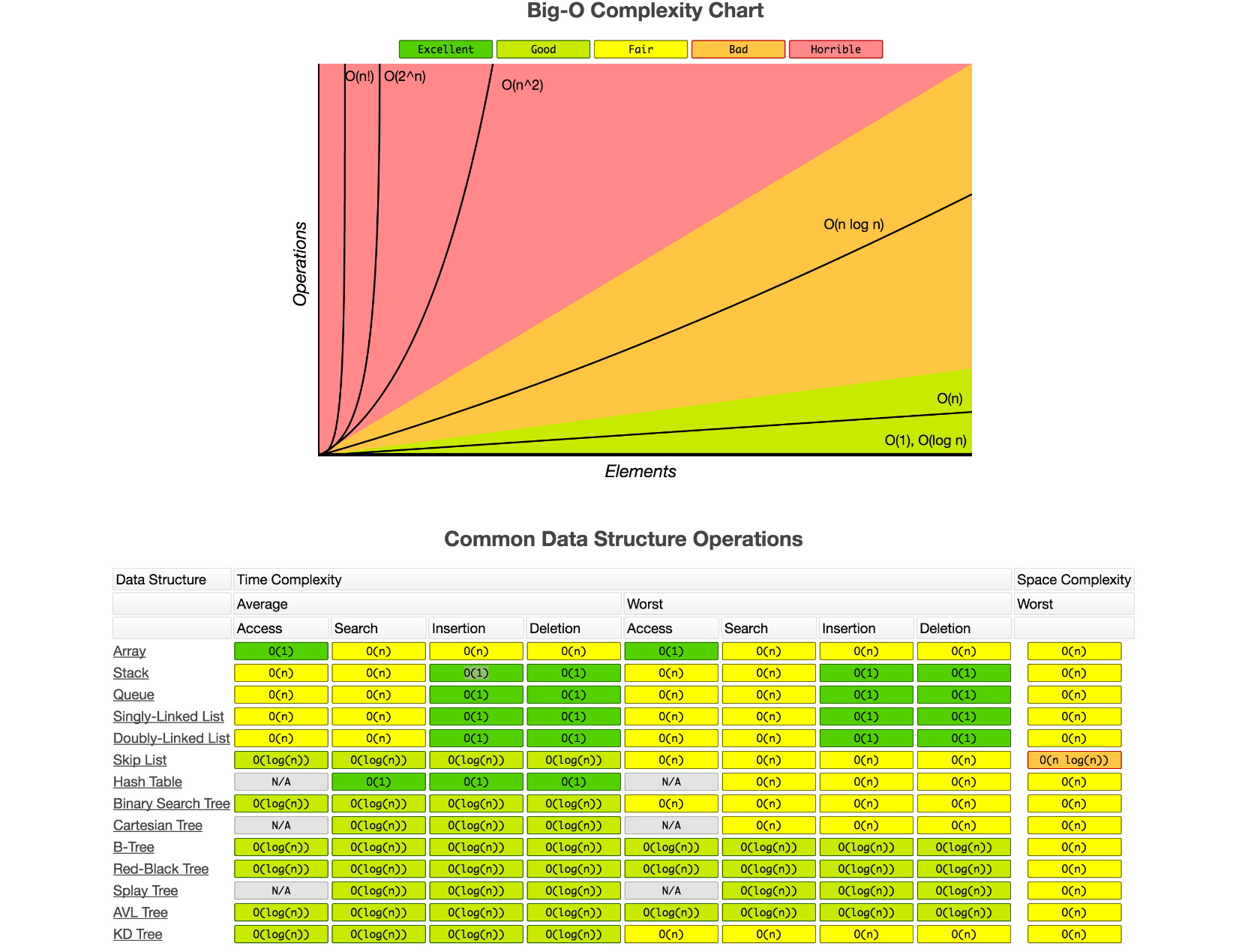
* Způsob, kterým se dá u programu určit, kolik prostředků k dosažení svého cíle bude potřebovat při různých objemech dat.
* Asymptotická je proto, že se zabývá výkonností, když je do programu zadáno **neomezené množství dat**.
* Dá se určit **paměťová** a **časová**.
* **Proč:** aby se dalo předpovědět, jak si program poradí s většími objemy dat.
* **Značí se jako:**
  1. **Big O (big ou)**
  + určuje se nejhorší možný případ. Ve většině případů se používá právě toto značení
  1. **Big Θ (velká théta)**
  + určuje se případ ve vymezených hranicích. To znamená, že reálná složitost se bude lišit od perfektní složitosti maximálně v rámci vymezených limitů.
  + **Big Ω (big omega)**
  + určuje se nejlepší možný případ

1. **Časová složitost**

* Určuje, za jak dlouho program dojde do cíle s množstvím dat **x** za předpokladu, že má neomezené množství paměti (kolik **času procesoru** využije).

1. **Paměťová složitost**

* Určuje, kolik **operační paměti** program využije na množství dat **x**.



**y** - složitost (čas)

**x** - množství dat

**Příklady složitosti**

1. **O(1) - konstantní**

* kód, který se ukončí stejně rychle a nezáleží mu na velikosti dat. Třeba vkládání do proměnné:

int c = List[x];

1. **O(log n) - logaritmická**

* Třeba for cyklus, který násobí po každém proběhnutí.

for(int i = 1; i < **x**; i**\***=2){

//něco udělej

}

1. **O(n) - lineární**

* kód, který běží úměrně rychle tomu, kolik dat je do něj vloženo. Třeba for cyklus, který přičítá po každém proběhnutí. Může být třeba plnění pole.

for(int i = 0; i > **x**; i**+**=1){

//něco udělej

}

1. **O(n log n) - kombinace lineární a logaritmické**

* Třeba for cyklus, který násobí (log n), vložen do cyklu, který přičítá (n)

for(int i = 0; i > **x**; i**+**=1){

for(int j = 1; j < **x**; j**\***=2){

//něco udělej

}

}

1. **O(n^2) - kvadratická**

* Třeba dva for cykly, které přičítají (n), v sobě. Může být třeba plnění dvojrozměrného pole. Čím větším číslem se n umocňuje, tím víc cyklů je v sobě.

for(int i = 0; i > **x**; i**+**=1){

for(int j = 0; j < **x**; j**+**=2){

//něco udělej

}

}

* Pozor, když má pole jiné velikosti rozměrů (třeba obdélník, místo čtverce strany **a** a **b**), tak se algoritmus zapisuje třeba jako **O(a\*b)**, podle počtu rozměrů.

1. **O(2^n) - exponenciální**

* Algoritmus, u kterého se složitost zdvojnásobí s každým dalším prvkem. Klasický příklad je třeba funkce pro výpočet Fibonacciho sekvence, při kterém se používá rekurze.

int Fibonacci(int n){  
 if (**n** <= 1){  
 return n;

}else{  
 return Fibonacci(n - 1) + Fibonacci(n - 2);

}

}

1. **O(n!) - faktoriálová**

* Algoritmus, u kterého složitost = n!, postupuje hodně rychle nahoru s více daty. Většinou tuto složitost mají bruteforce řešení. Klasický příklad je **problém cestujícího prodavače**, který musí najít nejkratší cestu mezi **n** městy a vrátit se zpět do počátku. Bruteforce způsob je zjistit všechny kombinace měst a z nich vybrat tu nejkratší.
* Příklad n! složitosti je třeba for loop, který běží n!-krát.

for(int j = 0; j <= **factorial(n)**; j+=1){

//něco udělej}