**Analýza algoritmů**

* V Informatice představuje analýza algoritmů určení výpočetní složitosti algoritmů, tj. množství času, úložného prostoru (paměti) a/nebo dalších zdrojů nezbytných k jejich provedení.
* Obvykle to zahrnuje definování matematické funkce, která určuje závislost počtu kroků (časovou složitost algoritmu – time complexity) nebo velikost potřebné paměti, kterou algoritmus použije (paměťová/prostorová složitost – space complexity) na velikosti vstupu algoritmu.
* Algoritmus považujeme za efektivní, když jsou výstupy takovéto funkce nízké (pro známou konstantní velikost vstupu).
* ALE! Protože různé vstupy stejné délky mohou způsobit, že se algoritmus chová odlišně, funkce popisující jeho složitost je obvykle horní hranicí skutečného výkonu algoritmu (upper bound), určeného od nejhorších možných vstupů do algoritmu (worst case scenario)

**Časová a prostorová složitost / Time and space complexity**:

* Časová složitost udává, jak dlouho bude algoritmus běžet v závislosti na velikosti vstupních dat (nebo alternativně jaký bude počet kroků).
* Prostorová (paměťová) složitost definuje, kolik paměti je potřeba k provedení algoritmu v závislosti na velikosti vstupních dat (nebo alternativně velikost zásobníku, délka pásky pro stroje/automaty).

**Můžeme také uvážit další metriky:**

* Množství použité komunikace (communication complexity)
* Množství procesorů/vláken (parallel computing complexity)

**Pozor ale na terminologii!**

* **Teorie vypočítatelnosti** (Computability theory) - úzce souvisí s teorií složitosti, ale řeší obecnou otázku o algoritmech, které by bylo možné použít k vyřešení určitého problému.
* Oblast teoretické informatiky, která se zabývá tím, co může být vypočítáno a jakými prostředky. Je to formální studie toho, které problémy lze řešit pomocí algoritmů a které jsou mimo možnosti výpočetních strojů, ať už z důvodu časové nebo prostorové složitosti, nebo kvůli jiným omezením výpočetních modelů.
* Zavádí důležité pojmy, jako: Rozhodnutelnost, Rekurzivní spočetnost, Redukce a Třídy složitosti, a další…

Jestli toto někdo čte, tuto část zjednodušilo chat GPT z prezentace, snad to půjde pochopit.

**Co je teorie výpočetní složitosti?**

Teorie výpočetní složitosti se snaží pochopit, jak těžké (z hlediska času a paměti) je řešit různé problémy pomocí počítače. Zabývá se tím, co počítače zvládnou efektivně, co je naopak těžké a co možná ani nelze spočítat.

**Co znamenají třídy P a NP?**

**Třída P:**

* To jsou problémy, které lze rychle vyřešit na počítači (rychle znamená v „polynomiálním čase“ – například n,n2 , n3 kroků, kde n je velikost zadání).
* Příklady: Třídění čísel, hledání nejkratší cesty v jednoduchém grafu.

**Třída NP:**

* To jsou problémy, kde je složité najít řešení, ale když ho máte, lze rychle ověřit, jestli je správné.
* Příklad: Problém obchodního cestujícího (TSP) – najít nejlevnější trasu, která projde každým městem právě jednou. Když vám někdo řekne „tady je ta trasa“, můžete snadno spočítat její délku a ověřit, jestli je správná.

**Otázka P vs NP:**

* Nevíme, jestli každý problém v NP lze také rychle vyřešit (tedy jestli P = NP). Pokud by P = NP, znamenalo by to, že všechny problémy, které lze ověřit, lze i rychle vyřešit. To by bylo obrovské překvapení, protože dnes věříme, že některé problémy jsou tak složité, že rychle řešit nepůjdou.

**Co je problém a instance problému?**

* Problém: Obecná otázka, která má být vyřešena. Například: „Jak najít nejkratší cestu mezi městy?“
* Instance problému: Konkrétní zadání toho problému. Například: „Máme šest měst, mezi nimi vzdálenosti (1,2) = 10 km, (1,3) = 14 km, ... Jaká je nejkratší cesta?“

**Co je algoritmus?**

**Algoritmus:** Postup nebo sada kroků, kterými problém vyřešíme. Pro každou instanci problému algoritmus najde správné řešení v konečném počtu kroků.

**Například:** Algoritmus pro hledání nejkratší cesty může vyzkoušet všechny možné trasy a spočítat jejich délku.

**Jak měříme složitost algoritmů?**

* Počet kroků: Počítáme, kolik kroků algoritmus potřebuje na vyřešení dané instance.
* Asymptotická složitost: Nezajímá nás přesné číslo kroků, ale jak se tento počet chová, když vstup roste. Například:
  + Algoritmus má složitost O(n2 ), což znamená, že pokud vstup zdvojnásobíme, počet kroků se zhruba zčtyřnásobí.
  + Složitost O(n!) znamená, že algoritmus se stává extrémně pomalým, když vstup roste.

**Proč záleží na složitosti?**

Rychlejší algoritmy šetří čas i energii.

* Pokud máte špatný algoritmus, může trvat hodiny, dny nebo roky, než něco spočítá – i když počítač sám je rychlý.
* Proto je důležité najít efektivní řešení, která budou fungovat i pro velká data.

**Shrnutí na příkladu:**

Představte si, že organizujete cestu pro obchodního cestujícího, který chce navštívit 10 měst. Kdybychom zkusili všechny možné cesty, bylo by to 10! (tedy 3 628 800 možností). Takové výpočty jsou pro počítač obtížné, pokud je n (počet měst) větší. Místo toho potřebujeme chytré algoritmy, které problém vyřeší efektivněji.

**Teorie výpočetní složitosti (Asymptotická)**

* Teorie asymptotické složitosti zkoumá výpočetní výkon algoritmů, zejména jejich časovou a prostorovou náročnost, když velikost vstupu nnn roste. Pomocí asymptotické analýzy lze určit hranice běhové doby algoritmu bez ohledu na konkrétní hardware nebo implementaci.

**Klíčové body asymptotické analýzy**

1. **Nejlepší, průměrný a nejhorší případ:**
   * **Nejlepší případ**: Minimální doba běhu (např. už je vstup setříděný).
   * **Průměrný případ**: Očekávaná doba běhu pro různé vstupy.
   * **Nejhorší případ**: Maximální doba běhu (např. vstup je obráceně setříděný).
2. **Vliv vstupu:**
   * Velikost vstupu n ovlivňuje dobu běhu. U algoritmů bez vstupu lze předpokládat, že jejich doba běhu je konstantní (Θ(1)).
3. **Používané notace:**
   * **Big O (O)**: Horní hranice (nejhorší případ).
   * **Omega (Ω)**: Dolní hranice (nejlepší případ).
   * **Theta (Θ)**: Kombinace dolní i horní hranice (přesná asymptotická charakteristika).

**Důležitost asymptotické analýzy**

* Zaměřuje se na algoritmy, nikoli na konkrétní implementace.
* Ignoruje vlivy hardwaru, kompilátorů nebo optimalizací.
* Pomáhá porovnávat algoritmy na základě jejich výkonu při růstu nnn.

Příklad: Pokud algoritmus A má časovou složitost O(n2 ) a algoritmus B O(n log n), algoritmus B bude výhodnější pro velká nnn.

**Kategorie časové složitosti**

1. **Konstantní (Θ(1)):** Algoritmus nezávisí na velikosti vstupu (např. přístup k prvku pole).
2. **Logaritmická (Θ(log n)):** Např. binární vyhledávání.
3. **Lineární (Θ(n)):** Prochází každý prvek (např. lineární vyhledávání).
4. **Lineárně-logaritmická (Θ(nlog n)):** Např. efektivní třídicí algoritmy (merge sort, quicksort).
5. **Kvadratická (Θ(n2 ):** Např. naivní algoritmy pro třídění (bubblesort).
6. **Kubická (Θ(n3 )):** Např. jednoduché algoritmy pro násobení matic.
7. **Exponenciální (Θ(2n )):** Např. problémy řešené brute-force.
8. **Faktoriální (Θ(n!)):** Např. permutace všech možných řešení.

**Vztahy mezi složitostmi**

* **Logaritmický růst log n** je pomalejší než jakýkoli polynomiální růst nk .
* **Polynomiální růst** nk je pomalejší než exponenciální růst 2n
* **Faktoriální růst n!** převyšuje exponenciální růst

**Vlastnosti notací**

1. **Big O (O):**  
   Horní hranice. Příklad: Pro f(n)=O(g(n)), platí f(n) ≤ c⋅g(n), kde c je konstanta.
2. **Omega (Ω):**  
   Dolní hranice. Příklad: Pro f(n)=Ω(g(n)), platí f(n) ≥ c⋅g(n).
3. **Theta (Θ):**  
   Přesné vyjádření růstu. Příklad: Pro f(n)=Θ(g(n)), platí c1⋅g(n) ≤ f(n) ≤ c2⋅g(n)

**Praktické příklady složitostí**

1. **Hledání v seznamu:**
   * Lineární složitost (O(n)) pro neřazený seznam.
   * Logaritmická složitost (O(log n)) pro řazený seznam (binární vyhledávání).
2. **Třídění:**
   * Naivní algoritmy: O(n2 ) .
   * Efektivní algoritmy: O(n log n).
3. **Násobení matic:**
   * Naivní přístup: O(n3 )
   * Optimalizované algoritmy: O(n2.8 ) nebo méně.
4. **Grafové úlohy:**
   * Např. Dijkstrův algoritmus pro nejkratší cestu: O(n2 ) nebo O(n log n).

**Jak zrychlit algoritmy**

* Musí se zjistit, která část algoritmu je nejpomalejší a ta se musí zrychlit
* **Předpočítání**: Některé věci by měly být předem vypočítány na úkor použité paměti. Když je potřebujeme, v konstantní čase je vyhledáme v tabulce.
* **Preprocessing:** Připravit data (redukovat) předem (zvláště když spouštíme vícenásobné analýzy).
* **Odstranit rekurzi**: Například nahradíme faktoriální funkci naprogramovanou rekurzí pomocí for-cyklu. Podobného efektu dosáhneme výběrem správného programovacího jazyka, nebo nepoužíváme věci, které nejsou zrovna nutné a potřeba (například objekty).
* **Eliminace opakovaných výpočtů.** 
  + Nejčastěji se jedná o přesnou práci s pamětí (neplýtvejte – garbage collector, fit in memory), omezení přístupu k souborům (buď na pevný disk nebo jen psaní textu do terminálu, ukládání do vyrovnávací paměti) atd, vhodně napsaný data collector.