Vyčíslitelnost a složitost výpočtů

Časová složitost algoritmu

- Časová složitost algoritmu popisuje, jak se doba běhu algoritmu mění v závislosti na velikosti vstupních dat.
- Především u velkých dat.
- Časová složitost se obvykle vyjadřuje pomocí tzv. **asymptotické notace** (Horní odhad růstu).

Časová složitost algoritmu

O(1) – Konstanta:

- Algoritmus běží vždy ve stejném čase bez ohledu na velikost vstupu.
- Příklad: Přístup k prvku v poli pomocí indexu.

O(n) – Lineární:

- Doba běhu algoritmu roste přímo úměrně s velikostí vstupu.
- Příklad: Prohledání nesetříděného seznamu s n prvky.

O(log n) – Logaritmická:

- Časová složitost roste pomaleji než velikost vstupu, což je typické pro algoritmy, které opakovaně dělí problém na menší části.
- Příklad: Binární vyhledávání.

O(n log n) – Lineárně logaritmická:

- Typická pro efektivní třídicí algoritmy jako je *quicksort* nebo *mergesort*.
- Časová složitost roste rychleji než lineární, ale pomaleji než kvadratická.

Časová složitost algoritmu

$O(n^2)$ – Kvadratická:

- Doba běhu roste úměrně druhé mocnině velikosti vstupu.
- Příklad: Třídění bublinkou (bubble sort).

$O(2^n)$ – Exponenciální:

- Časová složitost exponenciálně roste s velikostí vstupu.
- Příklad: Rekurzivní řešení problému cestujícího obchodníka.

$O(n!)$ – Faktoriální:

- Extrémně náročné algoritmy, kde doba běhu dramaticky roste s velikostí vstupu.
- Příklad: Brute-force.

Časová složitost algoritmu

1. Identifikace základních operací algoritmu

Každý algoritmus je sestaven z jednotlivých operací, jako jsou:

- Přiřazení hodnot proměnným
- Podmínky (if-else)
- Smyčky (for, while)
- Rekurze

Časová složitost algoritmu

1. Identifikace základních operací algoritmu

Každý algoritmus je sestaven z jednotlivých operací, jako jsou:

- Přiřazení hodnot proměnným
- Podmínky (if-else)
- Smyčky (for, while)
- Rekurze

Časová složitost algoritmu

2. Analýza smyčky

- Smyčky (cykly) mají velký vliv na časovou složitost algoritmu, protože určují, jak často se provádí základní operace.
- **Jednoduchá smyčka, která běží n krát** má časovou složitost **O(n)**.
- **Smyčka uvnitř smyčky** (vnořená smyčka) má složitost násobenou, tj. pro dvě vnořené smyčky, kde každá běží n krát, bude celková složitost **$O(n^2)$**.

Časová složitost algoritmu

3. Podmínky

- Podmínky (if-else) mohou mít různé větve, ale z hlediska asymptotické notace se uvažuje vždy ta nejhorší možná větev (*worst-case* analýza).
- Pokud rozhodnutí v podmínce nijak neovlivňuje složitost (provádí se jen jednou), přidává jen **O(1)** (konstantní složitost).

Časová složitost algoritmu

4. Rekurze

- U rekurzivních algoritmů musíte vytvořit tzv. rekurentní vztah a analyzovat ho.

Příklad:

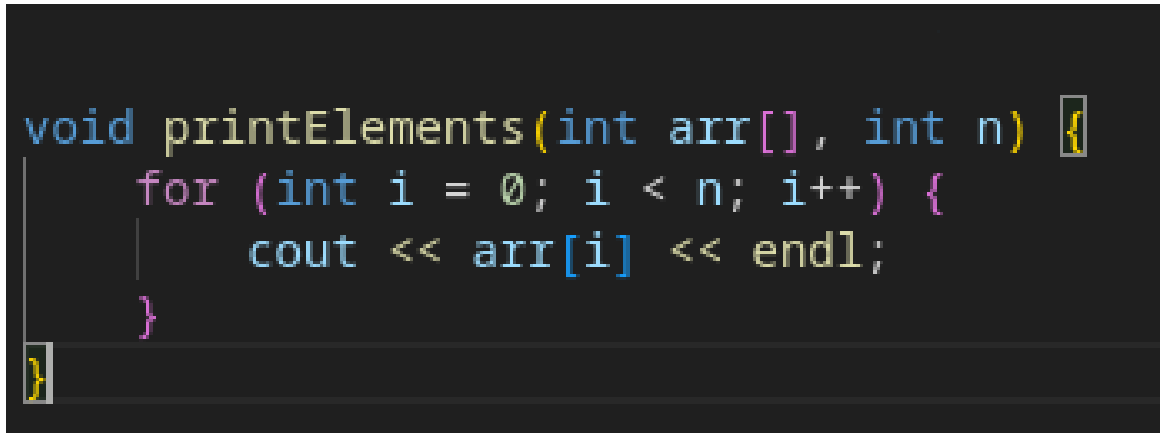
- **Binární vyhledávání**: Rozdělí vstup na polovinu v každém kroku, což vede k **O(log n)**.
- **Třídění pomocí merge sortu**: Rozděluje vstupní seznam a kombinuje je zpět, což vede k **O(n log n)**.

Časová složitost algoritmu

```
void example(int arr[], int n) {
    for (int i = 0; i < n; i++) {
        cout << arr[i] << endl;
    }

    for (int i = 0; i < n; i++) {
        for (int j = 0; j < n; j++) {
            cout << arr[i] + arr[j] << endl;
        }
    }
}
```

Časová složitost algoritmu



Časová složitost algoritmu

```
int binarySearch(int arr[], int n, int target) {
    int left = 0, right = n - 1;
    while (left <= right) {
        int mid = (left + right) / 2;
        if (arr[mid] == target) return mid;
        else if (arr[mid] < target) left = mid + 1;
        else right = mid - 1;
    }
    return -1;
}
```

Prostorová složitost algoritmů

Prostorová složitost algoritmů

- Množství paměti potřebné pro běh algoritmu vzhledem k velikosti vstupu.
- U paměťově náročných výpočtů nebo aplikací, kde je omezená paměť.
- Časová složitost hodnotí, jak dlouho algoritmus běží, zatímco prostorová složitost zohledňuje, kolik paměti využívá.
- Ideální algoritmy jsou efektivní jak časově, tak prostorově, ale často je potřeba najít kompromis mezi těmito dvěma faktory.

Prostorová složitost algoritmů

- **O(1)** – Konstantní:
  - Algoritmus potřebuje konstantní paměť, bez ohledu na velikost vstupu.
  - Příklad: Vyhledání minimálního nebo maximálního čísla v seznamu (pouze několik proměnných).
- **O(log n)** – Logaritmická:
  - Objevuje se u rekurzivních algoritmů, které dělí problém na poloviny.
  - Příklad: rekurzivní binární hledání.
- **O(n)** – Lineární:
  - Paměťové nároky algoritmu rostou přímo úměrně velikosti vstupu.
  - Příklad: Kopírování pole, kde je potřeba nová paměť pro každý prvek vstupního pole.

Prostorová složitost algoritmů

- **O(n log n)** – Lineárně-logaritmická prostorová složitost
  - Vzácné – nastává, pouze když **na každé z log n úrovní dělení vzniká nová kopie dat**, která se **uchovává zároveň**
- **O(n^2)** – Kvadratická:
  - Algoritmus potřebuje paměť, která roste kvadraticky s velikostí vstupu.
  - Příklad: Dynamické programování.
- **O(2^n)** – Exponenciální prostorová složitost
  - Objevuje se v algoritmech, které generují nebo ukládají **všechny podmnožiny** nebo **kombinace** prvků, protože počet těchto množin je 2^n.
  - Ukládání všech podmnožin množiny.

Prostorová složitost algoritmů

- **O(n!)** – Faktoriální prostorová složitost
  - Nastává v případech, kdy algoritmus ukládá všechna možná uspořádání nebo kombinace prvků. Běžně se objevuje u algoritmů, které generují a ukládají všechna řešení (například všechny permutace).
  - Příklad: generování a ukládání všech permutací n prvků.

Prostorová složitost algoritmů

Identifikace konstantní paměti:

- **Statické datové struktury:** Pole, seznamy nebo objekty s pevnou velikostí (například pole s pevnou délkou) představují konstantní paměť, O(1)
- int a = 5; int b = 10;

Prostorová složitost algoritmů

Analýza dynamicky alokovaných datových struktur:

- **Pole, seznamy, objekty:** Pokud kód obsahuje struktury, jejichž velikost roste s velikostí vstupu (například seznam, který ukládá všechny prvky z pole), sledujte, jak se mění jejich velikost. Taková struktura obvykle představuje lineární prostorovou složitost O(n).

```
List<int> numbers = new List<int>();
for (int i = 0; i < n; i++) {
    numbers.Add(i);
}
```

Prostorová složitost algoritmů

Dvourozměrné datové struktury:

- Kvadratická prostorová složitost $O(n^2)$ nastává, když algoritmus vyžaduje paměť, která roste s druhou mocninou velikosti vstupu. To se často stává, když algoritmus pracuje s dvourozměrnými datovými strukturami, jako jsou matice nebo tabulky, kde je třeba uložit informace pro každou kombinaci nebo dvojici prvků.

Prostorová složitost algoritmů

```
int[,] result = new int[n, n];
for (int i = 0; i < n; i++) {
    for (int j = 0; j < n; j++) {
        for (int k = 0; k < n; k++) {
            result[i, j] += matrixA[i, k] * matrixB[k, j];
        }
    }
}
```

Prostorová složitost algoritmů

Logaritmická prostorová složitost:

- Prostorová složitost **O(log n)** znamená, že algoritmus vyžaduje paměť, která roste logaritmicky vzhledem k velikosti vstupu. To je typické pro algoritmy, které uchovávají pouze malé množství informací o každé úrovni při zmenšování problému na polovinu nebo na jiný pevný zlomek původní velikosti.

Prostorová složitost algoritmů

```
public static int BinarySearch(int[] arr, int x)
{
    int left = 0;
    int right = arr.Length - 1;

    while (left <= right)
    {
        int mid = left + (right - left) / 2;

        if (arr[mid] == x)
            return mid;

        if (arr[mid] > x)
            right = mid - 1;
        else
            left = mid + 1;
    }

    return -1;
}
```