Binární vyhledávání

DELTA - Střední škola informatiky a ekonomie, s.r.o.

Ing. Luboš Zápotočný

11.12.2025

Binární vyhledávání

Proč binární vyhledávání?

Úloha: Najdi číslo v seřazeném poli

Proč binární vyhledávání?

Úloha: Najdi číslo v seřazeném poli

Možnosti:

Algoritmus	**Složitost**	**Pro 1,000,000 prvků**
Lineární vyhledávání	$O(n)$	

Proč binární vyhledávání?

Úloha: Najdi číslo v seřazeném poli

Možnosti:

Algoritmus	Složitost	Pro 1,000,000 prvků
Lineární vyhledávání	$O(n)$	1,000,000 kroků
Binární vyhledávání	$O(\log n)$	

Proč binární vyhledávání?

Úloha: Najdi číslo v seřazeném poli

Možnosti:

Algoritmus	**Složitost**	**Pro 1,000,000 prvků**
Lineární vyhledávání	$O(n)$	1,000,000 kroků
Binární vyhledávání	$O(\log n)$	20 kroků

Proč binární vyhledávání?

Úloha: Najdi číslo v seřazeném poli

Možnosti:

Algoritmus	Složitost	Pro 1,000,000 prvků
Lineární vyhledávání	$O(n)$	1,000,000 kroků
Binární vyhledávání	$O(\log n)$	20 kroků

Pro 1 miliardu prvků: lineární =

Proč binární vyhledávání?

Úloha: Najdi číslo v seřazeném poli

Možnosti:

Algoritmus	Složitost	Pro 1,000,000 prvků
Lineární vyhledávání	$O(n)$	1,000,000 kroků
Binární vyhledávání	$O(\log n)$	20 kroků

Pro 1 miliardu prvků: lineární = 1 miliarda kroků, binární =

Proč binární vyhledávání?

Úloha: Najdi číslo v seřazeném poli

Možnosti:

Algoritmus	Složitost	Pro 1,000,000 prvků
Lineární vyhledávání	$O(n)$	1,000,000 kroků
Binární vyhledávání	$O(\log n)$	20 kroků

Pro 1 miliardu prvků: lineární = 1 miliarda kroků, binární = 30 kroků

Princip binárního vyhledávání

Špatný způsob (lineární):

Princip binárního vyhledávání

Špatný způsob (lineární):

„Je to 1? Ne. Je to 2? Ne. Je to 3? Ne...“ $\rightarrow$

Princip binárního vyhledávání

Špatný způsob (lineární):

„Je to 1? Ne. Je to 2? Ne. Je to 3? Ne...“ $\rightarrow$ až 100 pokusů

Princip binárního vyhledávání

Špatný způsob (lineární):

„Je to 1? Ne. Je to 2? Ne. Je to 3? Ne…“ $\rightarrow$ až 100 pokusů

Dobrý způsob (binární):

Princip binárního vyhledávání

Špatný způsob (lineární):

„Je to 1? Ne. Je to 2? Ne. Je to 3? Ne..." → až 100 pokusů

Dobrý způsob (binární):

1. "Je to 50?" ⟶ "Vyšší"
2. "Je to 75?" ⟶ "Nižší"
3. "Je to 63?" ⟶ "Vyšší"
4. …

Princip binárního vyhledávání

Špatný způsob (lineární):

„Je to 1? Ne. Je to 2? Ne. Je to 3? Ne...“ → až 100 pokusů

Dobrý způsob (binární):

1. “Je to 50?” ⟶ “Vyšší”
2. “Je to 75?” ⟶ “Nižší”
3. “Je to 63?” ⟶ “Vyšší”
4. …

Princip: Každý pokus **eliminuje**

Princip binárního vyhledávání

Špatný způsob (lineární):

„Je to 1? Ne. Je to 2? Ne. Je to 3? Ne...“ → až 100 pokusů

Dobrý způsob (binární):

1. “Je to 50?” ⟶ “Vyšší”
2. “Je to 75?” ⟶ “Nižší”
3. “Je to 63?” ⟶ “Vyšší”
4. …

Princip: Každý pokus **eliminuje polovinu** možností!

Binární vyhledávání

Binární vyhledávání

1. Začni s celým polem (levá a pravá hranice)

Binární vyhledávání

1. Začni s celým polem (levá a pravá hranice)
2. Spočítej prostřední index: $m = \frac{l+r}{2}$

Binární vyhledávání

1. Začni s celým polem (levá a pravá hranice)
2. Spočítej prostřední index: $m = \frac{l+r}{2}$
3. Porovnej `arr[m]` s hledanou hodnotou `x`:

Binární vyhledávání

1. Začni s celým polem (levá a pravá hranice)
2. Spočítej prostřední index: $m = \frac{l+r}{2}$
3. Porovnej `arr[m]` s hledanou hodnotou `x`:
    - Pokud `arr[m] == x` $\longrightarrow$ našli jsme! 🎉
    - Pokud `arr[m] < x` $\longrightarrow$ hledej v pravé polovině
    - Pokud `arr[m] > x` $\longrightarrow$ hledej v levé polovině

Binární vyhledávání

1. Začni s celým polem (levá a pravá hranice)
2. Spočítej prostřední index: $m = \frac{l+r}{2}$
3. Porovnej `arr[m]` s hledanou hodnotou `x`:
   - Pokud `arr[m] == x` $\longrightarrow$ našli jsme! 🎉
   - Pokud `arr[m] < x` $\longrightarrow$ hledej v pravé polovině
   - Pokud `arr[m] > x` $\longrightarrow$ hledej v levé polovině
4. Opakuj dokud nenajdeš nebo dokud existuje interval na ověření

Binární vyhledávání

1. Začni s celým polem (levá a pravá hranice)
2. Spočítej prostřední index: $m = \frac{l+r}{2}$
3. Porovnej `arr[m]` s hledanou hodnotou `x`:
   - Pokud `arr[m] == x` $\longrightarrow$ našli jsme! 🎉
   - Pokud `arr[m] < x` $\longrightarrow$ hledej v pravé polovině
   - Pokud `arr[m] > x` $\longrightarrow$ hledej v levé polovině
4. Opakuj dokud nenajdeš nebo dokud existuje interval na ověření

Výsledek:

Binární vyhledávání

1. Začni s celým polem (levá a pravá hranice)
2. Spočítej prostřední index: $m = \frac{l+r}{2}$
3. Porovnej `arr[m]` s hledanou hodnotou `x`:
    - Pokud `arr[m] == x` → našli jsme! 🎉
    - Pokud `arr[m] < x` → hledej v pravé polovině
    - Pokud `arr[m] > x` → hledej v levé polovině
4. Opakuj dokud nenajdeš nebo dokud existuje interval na ověření

Výsledek: Vrátí index nebo –1 (nenalezeno)

Binární vyhledávání - příklad

Hledáme číslo 10

0	1	2	3	4	**5**	6	7	8	9	10	11
1	2	3	4	5	6	7	8	9	10	11	12

Binární vyhledávání - příklad

Hledáme číslo 10

0	1	2	3	4	**5**	6	7	8	9	10	11
1	2	3	4	5	6	7	8	9	10	11	12

6	7	**8**	9	10	11
7	8	9	10	11	12

Binární vyhledávání - příklad

9	**10**	11
10	11	12

Binární vyhledávání - výsledek

Hledáme číslo 10

9	**10**	11
10	11	12

Binární vyhledávání - výsledek

Hledáme číslo 10

9	**10**	11
10	11	12

9
10

Binární vyhledávání - výsledek

Hledáme číslo 10

9	**10**	11
10	11	12

9
10

Číslo 10 bylo nalezeno na indexu 9

Binární vyhledávání - výsledek

Bylo za potřebí pouze

Binární vyhledávání - výsledek

Bylo za potřebí pouze $\mathcal{O}(\log n)$ operací pro vyhledání kteréhokoli prvku v takto seřazeném poli

Binární vyhledávání - výsledek

Bylo za potřebí pouze $\mathcal{O}(\log n)$ operací pro vyhledání kteréhokoli prvku v takto seřazeném poli

Oproti lineárnímu vyhledávání, které by muselo zkontrolovat každý prvek, tedy

Binární vyhledávání - výsledek

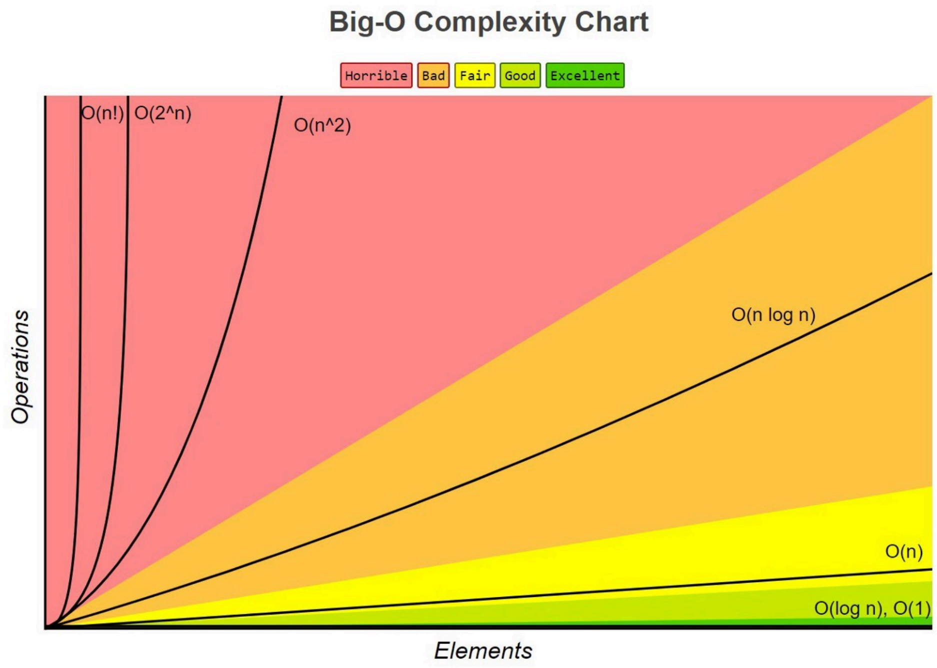
Bylo za potřebí pouze $\mathcal{O}(\log n)$ operací pro vyhledání kteréhokoli prvku v takto seřazeném poli

Oproti lineárnímu vyhledávání, které by muselo zkontrolovat každý prvek, tedy $\mathcal{O}(n)$ operací

Binární vyhledávání v C

```c
int binarySearch(int arr[], int l, int r, int x)
{
    while (l <= r) {
        int m = l + (r - l) / 2;
        // Check if x is present at mid
        if (arr[m] == x) return m;
        // If x greater, ignore left half
        if (arr[m] < x) l = m + 1;
        // If x is smaller, ignore right half
        else r = m - 1;
    }
    return -1; // if we reach here, then element was not present
}
```

Časová složitost algoritmů



towardsdatascience.com

Časová složitost algoritmů

Srovnání pro n = 1,000,000:

Časová složitost algoritmů

Srovnání pro n = 1,000,000:

- $O(1) =$ 1 operace

Časová složitost algoritmů

Srovnání pro n = 1,000,000:

- $O(1) =$ 1 operace
- $O(\log n) =$ 20 operací

Časová složitost algoritmů

Srovnání pro n = 1,000,000:

- $O(1) =$ 1 operace
- $O(\log n) =$ 20 operací
- $O(n) =$ 1,000,000 operací

Časová složitost algoritmů

Srovnání pro n = 1,000,000:

- $O(1) =$ 1 operace
- $O(\log n) =$ 20 operací
- $O(n) =$ 1,000,000 operací
- $O(n^2) =$ 1,000,000,000,000 operací

Časová složitost algoritmů

Srovnání pro n = 1,000,000:

- $O(1) = 1$ operace
- $O(\log n) = 20$ operací
- $O(n) = 1{,}000{,}000$ operací
- $O(n^2) = 1{,}000{,}000{,}000{,}000$ operací
- $O(2^n) = 9.9 \times 10^{301029}$ operací

Časová složitost algoritmů

Srovnání pro n = 1,000,000:

- $O(1)$ = 1 operace
- $O(\log n)$ = 20 operací
- $O(n)$ = 1,000,000 operací
- $O(n^2)$ = 1,000,000,000,000 operací
- $O(2^n)$ = 9.9×10^{301029} operací
- $O(n!)$ = 8.3×10^{5565706} operací