# Binární vyhledávání

Představte si, že jste k narozeninám dostali obrovské pole setříděných záznamů (to je, pravda, trochu netradiční dárek, ale proč ne – může to být třeba telefonní seznam). Záznamy mohou vypadat libovolně a to, že jsou setříděné, znamená jen a pouze, že $x_1 < x_2 < \ldots < x_N$, kde $<$ je nějaká relace, která nám řekne, který ze dvou záznamů je menší (pro jednoduchost předpokládáme, že žádné dva záznamy nejsou stejné).

Co si ale s takovým polem počneme? Zkusíme si v něm najít nějaký konkrétní záznam $z$. To můžeme udělat třeba tak, že si nalistujeme prostřední záznam (označíme si ho $x_m$) a porovnáme s ním naše $z$. Pokud $z < x_m$, víme, že se $z$ nemůže vyskytovat „napravo“ od $x_m$, protože tam jsou všechny záznamy větší než $x_m$ a tím spíše než $z$. Analogicky pokud $z > x_m$, nemůže se $z$ vyskytovat v první polovině pole. V obou případech nám zbude jedna polovina a v ní budeme pokračovat stejným způsobem. Tak budeme postupně zmenšovat interval, ve kterém se $z$ může nacházet, až buďto $z$ najdeme nebo vyloučíme všechny prvky, kde by mohlo být.

Tomuto principu se obvykle říká *binární vyhledávání* nebo také *hledání půlením intervalu* a snadno ho naprogramujeme buďto rekurzivně nebo pomocí cyklu, v němž si budeme udržovat interval $\langle l, r\rangle$, ve kterém se hledaný prvek může nacházet:

```
function BinSearch(z : integer): integer;
var l, r, m : integer;
begin
  l := 1;   { interval, ve kterém hledáme }
  r := N;
  while l <= r do begin { ještě není prázdný }
    m := (l+r) div 2;   { střed intervalu }
    if z < x[m] then
      r := m-1          { je vlevo }
    else if z > x[m] then
      l := m+1          { je vpravo }
    else begin          { Bingo! }
      hledej := m; exit;
    end;
  end;
  hledej := -1;         { nebyl nikde }
end;
```

Všimněte si, že průchodů cyklem `while` může být nejvýše $\lceil \log_2 N \rceil$, protože interval $\langle l, r\rangle$ na počátku obsahuje $N$ prvků a v každém průchodu jej zmenšíme na polovinu (ve skutečnosti ještě o jedničku, ale tím lépe pro nás). Proto po $k$ průchodech bude interval obsahovat nejvýše $N/2^k$ prvků a jelikož pro $N/2^k < 1$ se algoritmus zastaví, může být $k$ nejvýše $\log_2 N$. Proto je časová složitost binárního vyhledávání $\mathcal{O}(\log N)$.

**Poznámky**

- Algoritmus je nejjednodušším příkladem návrhového postupu zvaného *Rozděl a panuj,* kterému je v dalším textu věnována celá kapitola.
- Pokud záznamy můžeme jenom porovnávat, je binární vyhledávání nejlepší možné. Libovolné hledání založené na porovnávání lze totiž popsat binárním stromem a binární strom s $N$ vrcholy musí mít vždy hloubku alespoň $\lfloor \log_2 N \rfloor$. Pokud můžeme použít *hešování* (viz další kapitoly), dostaneme se na průměrně konstantní složitost (ale v nejhorším případě lineární). Jeho nevýhodou ovšem je, že udržuje jenom množinu prvků, nikoliv uspořádání na ní, takže například nelze najít k zadanému prvku nejbližší vyšší.
- Když hledáme v telefonním seznamu, nepůlíme intervaly, ale hádáme, kam přibližně by ze zkoumaného intervalu hledaná hodnota mohla přijít – pokud hledáme Zemana, otevřeme seznam někde u konce. Na tom je založené *interpolační hledání,* které v průměrném případě rovnoměrně rozložených dat najde výsledek v $\mathcal{O}(\log \log N)$.

  Od výše uvedeného kódu se liší právě jenom určováním „středu“:

  ```
  m := l + (z-x[l]) * (r-l) div (x[r]-x[l]);   { střed intervalu }
  ```

  Na nepravidelných datech takové hledání ale může trvat až lineárně dlouho. Můžeme to ošetřit tím, že budeme kroky binárního a interpolačního hledání střídat. Na špatných datech tak bude doba běhu omezená dvojnásobkem doby běhu hledání binárního a na dobrých dvojnásobkem časové složitosti interpolačního hledání.

  Stojí nám to však za tu námahu? Asi jen v případě, kdy nás stojí čtení prvků pole nemalý čas, tedy pokud je uloženo na pevném disku, popř. aspoň vypadlo-li pro svou velikost z procesorových keší.
- Co když potřebujeme seznam rychle upravovat? Do pole novou hodnotu rychle vložit nejde. Normálně bychom použili spojový seznam, ale tím bychom zhoršili složitost binárního vyhledávání k nepoužitelnosti, protože je v něm nalezení prostřední hodnoty v $\mathcal{O}(N)$. Řešením jsou *vyhledávací stromy,* o kterých mluvíme v jedné z následujících kuchařek.

*Martin Mareš, Tomáš Valla a Lukáš Lánský*

**Úloha 22-5-2: Stráže údolí**

Údolí draků hlídají havrani rozmístění na přímce. Jenže někteří jsou moc blízko u sebe a mají tendenci se místo hlídání vybavovat, takže jsme se rozhodli počet stráží zredukovat. Nevíme však, které propustit.

Známe polohu všech $N$ havranů na přímce, zadanou celočíselnými mezerami mezi nimi, a chceme jich vyřadit $K$, aby byli dva nejbližší havrani od sebe co nejdále. Potřebujeme tedy maximalizovat minimální vzdálenost mezi nimi. Dokážete pro nás rychle najít $K$ havranů, jež pošleme do výslužby?

Například pro $N = 6$, $K = 3$ a mezery mezi havrany `4, 6, 2, 5, 7` je správným řešení propustit 2., 3. a 5. havrana (bráno zleva), takže zůstanou mezery `12, 12`. Pro

$N = 14$, $K = 7$, mezery `5, 12, 6, 3, 8, 1, 4, 1, 1, 9, 15, 1, 16` vyhodíme 2., 4., 6., 7., 8., 9. a 12. (možností je tentokrát více).

**Úloha 23-1-3: Jedna maticová**

Na vstupu dostaneme matici, tj. dvojrozměrné pole celých čísel, která má navíc tu zvláštní vlastnost, že jsou čísla v každém jejím řádku a sloupci ostře rostoucí (liší se alespoň o 1). Potřebovali bychom rychle zjistit, zdali v ní neexistuje nějaké políčko v $i$-tém řádku a $j$-tém sloupci, které by mělo hodnotu přesně $i + j$.

Pokud hledaných políček existuje víc, můžete vypsat libovolné z nich. Pokuste se také vymyslet, jak rychle spočítat, kolik takových políček je.

Při zvažování časové složitosti nepočítejte dobu načítání: představujte si, že už máte matici v paměti. Zkuste zdůvodnit, proč nelze dosáhnout rychlejšího řešení.

*Příklad vstupu:*

```
-3  1  4
 4  5  6
 7  9 11
```

*Odpovídající výstup:* `1. řádek, 3. sloupec`

*Příklad vstupu:*

```
 3  4  5
 4  5  6
 5  6  7
```

*Odpovídající výstup:* `žádné takové políčko není`