# Intervalové stromy

Představme si, že máme posloupnost celých čísel $p_0, p_1, \ldots p_{N-1}$, se kterou budeme průběžně provádět tyto dvě operace:

1. Změna jednoho čísla v posloupnosti.
2. Zjištění součtu čísel na nějakém intervalu $[a, b]$, tedy $p_a + p_{a+1} + \ldots + p_b$.

Nejdříve se zkusíme zamyslet, jak bychom úlohu řešili, kdybychom měli jen druhou operaci, tj. dotazy na součty na konkrétních intervalech. K řešení využijeme pole *prefixových součtů*.

Pole prefixových součtů je pole délky $N+1$, ve kterém na indexu $i$ leží součet prvků posloupnosti od indexu 0 až do indexu $i-1$. Tedy

$$pref[i] = p[0] + \ldots + p[i-1], pref[0] = 0$$

Není těžké si rozmyslet, že toto pole dokážeme jednoduše spočítat v čase $\mathcal{O}(N)$.

Nyní, když už známe všechny prefixové součty posloupnosti, umíme snadno spočítat součet na libovolném intervalu $[a, b]$:

$$s[a, b] = pref[b+1] - pref[a]$$

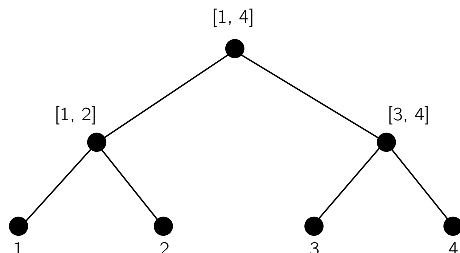
Každý dotaz dokážeme zodpovědět v konstantním čase. Celý algoritmus má tedy složitost $\mathcal{O}(N + D)$, kde $N$ je délka posloupnosti a $D$ je počet dotazů.

Když si do úlohy přidáme i operaci č. 1 (změna čísla v posloupnosti), tak se nám pokazí časová složitost. S prefixovými součty stále dokážeme dotaz č. 2 provádět v konstantním čase, ale při operaci č. 1 se nám může stát, že musíme změnit až všechny prefixové součty, takže složitost této operace je $\mathcal{O}(N)$ a celková složitost pro $Z$ změn a $D$ dotazů je v nejhorším případě $\mathcal{O}(NZ + D)$.

S touto složitostí se samozřejmě nespokojíme a budeme se snažit, abychom výsledné intervaly uměli co nejrychleji skládat z předpočítaných hodnot a abychom při změně posloupnosti museli změnit co nejméně hodnot. K tomu se nám bude hodit datová struktura jménem intervalový strom.

**Zavedení intervalového stromu**

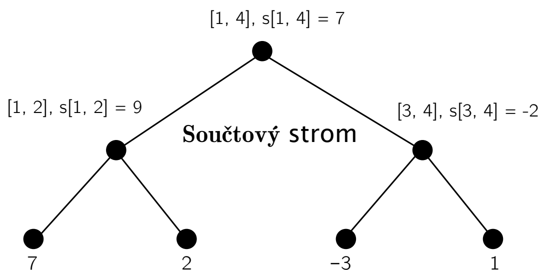
*Intervalový strom* je dokonale vyvážený binární strom, jehož každý list představuje nějaký interval a všechny ostatní vrcholy reprezentují interval, který vznikne složením intervalů jejich synů. Zároveň intervaly vrcholů jedné hladiny na sebe navazují (vždy směrem zleva doprava). Z toho vyplývá, že složením intervalů z vrcholů jedné hladiny dostaneme interval, který si pamatujeme v kořeni.




Intervalových stromů existuje více druhů. Obvykle je rozlišujeme podle toho, jaké informace si v nich pamatujeme. Například ve stromě pro součty si každý vrchol pamatuje součet na svém intervalu, ve stromě pro maxima si pamatuje maximum

na intervalu, apod. Můžeme ale klidně mít strom, který si pamatuje, jestli celý jeho interval obsahuje jen jednu hodnotu a pokud ano, tak jakou.

My se teď zaměříme na intervalový strom pro součty a pomocí něj vyřešíme úvodní úlohu.

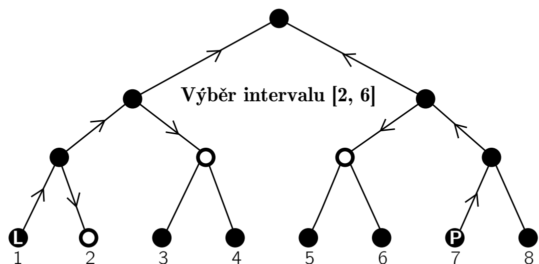



Na začátku budeme chtít, aby v listech intervalového stromu byly hodnoty původní posloupnosti, přičemž první a poslední list stromu necháme volné, později uvidíme, proč. Zároveň ale chceme, aby tento strom byl dokonale vyvážený.

Posloupnost tedy prodloužíme tak, aby její velikost byla mocnina dvojky minus dva (na její konec přidáme nějaké prvky). Všimněte si, že tím jsme strom nezvětšili více než dvakrát a že nám nezáleží na tom, jaké prvky jsme do stromu přidali, protože s nimi nikdy nebudeme pracovat. Nyní k jednotlivým operacím.

Změnu čísla v posloupnosti uděláme jednoduše. Zjistíme, o kolik se hodnota prvku posloupnosti změní, najdeme odpovídající list a k tomuto listu a ke všem jeho předkům přičteme daný rozdíl. Tím jsme upravili všechny intervaly, do kterých tento prvek patří.

Nyní se podívejme, jak ze stromu zjistíme součet na nějakém intervalu $[a, b]$. Jinými slovy: potřebujeme ze stromu vybrat takové vrcholy, aby sjednocení jejich intervalů byl náš dotazovaný interval, a zároveň chceme, aby těchto vrcholů bylo co nejméně.




Součet intervalu $[a, b]$ zjistíme tak, že si ve stromě najdeme listy reprezentující pozice $a-1$ a $b+1$ posloupnosti a jejich nejbližšího společného předka $p$. Nyní budeme postupovat z listu od $a-1$ až do $p$ a vždy když do nějakého vrcholu přijdeme z levého syna, tak do výsledku přidáme interval pravého syna. Stejně tak postupujeme od $b+1$ k $p$ a pokud do vrcholu přijdeme z pravého syna, tak přidáme jeho levého syna.

Všimněte si, že při takovémto průchodu složíme celý interval. Vše je vidět na obrázku vpravo.

Způsobů, jak pracovat z intervalovým stromem a zjišťování informací z něj, je více. Toto byl jeden z nich.

Změna prvku posloupnosti má časovou složitost $\mathcal{O}(\log N)$, protože jsme na každé hladině změnili pouze jeden interval a strom má $\mathcal{O}(\log N)$ hladin. Zjištění součtu na intervalu má také složitost $\mathcal{O}(\log N)$, jelikož jsme do výsledku přidali maximálně $2 \log N$ intervalů: nejvýše $\log N$ při cestě z listu $a-1$ a $\log N$ při cestě z $b+1$.

**Implementace intervalového stromu**

Při implementaci intervalového stromu využijeme jeho dokonalé vyváženosti a budeme jej implementovat v poli (stejně jako jsme do pole ukládali haldu). Kořen stromu bude v poli na indexu 1, vrcholy z druhé hladiny budou mít postupně indexy 2, 3, ..., až listy budou mít indexy $N$, ..., $2N - 1$. V této reprezentaci platí pro vrchol s indexem $i$ následující pravidla:

1. $2i$ a $2i + 1$ jsou jeho synové.
2. $\lfloor i/2 \rfloor$ je jeho předek (pro $i > 1$).
3. Pokud je $i$ sudé, tak je vrchol levým synem, jinak pravým.
4. Pro sudé $i$ je $i + 1$ pravý bratr, pro liché $i$ je $i - 1$ levý bratr.

Nyní víme vše potřebné, tak se podívejme na samotnou implementaci v jazyce C:

```
int N = 100;     // velikost posloupnosti
int posl[100];   // posloupnost
int *strom;      // intervalový strom

// Deklarace funkcí
void inic(int N);
void pricti(int index, int hodnota);
int soucet(int A, int B);

/* Inicializace intervalového stromu
 * Pozor: prvky posloupnosti indexujeme 1, ..., N
 */
void inic(int N) {
  // Najdeme nejbližší vyšší mocninu dvojky
  int listy = 1;
  while (listy<N+2) listy = listy*2;
  // Pro strom potřebujeme 2*(počet listů) vrcholů
  // (nepoužíváme strom[0])
  strom = (int*)malloc(sizeof(int)*2*listy);
  N = listy;
  for (int i=0; i<2*listy; i++) strom[i] = 0;
  // Na příslušná místa přičteme hodnoty posloupnosti
  for (int i=0; i<N; i++)
    pricti(i, posl[i]);
}

// Přičtení hodnoty na dané místo posloupnosti
void pricti(int index, int hodnota) {
  int k = N + index;
  while(k>0) {
    strom[k] = strom[k] + hodnota;
    k = k/2;
  }
}
```

```
// Zjištění součtu na intervalu
int soucet(int A, int B) {
  int souc = 0;
  int a = N + A - 1;
  int b = N + B + 1;
  while (a!=b) {
    // Pokud je a levý syn, tak přičti pravého bratra
    if (a%2==0) souc = souc + strom[a+1];
    // Pokud je b pravý syn, tak přičti levého bratra
    if (b%2==1) souc = souc + strom[b-1];
    // Přesun na otce
    a = a/2; b = b/2;
  }
  // Navíc jsme přičetli syny společného předka.
  souc = souc - strom[2*a] - strom[2*a+1];
  return souc;
}
```

V této implementaci jsme strom upravovali zdola směrem nahoru. Existuje ještě rekurzivní implementace, kde se strom upravuje od kořene směrem dolů, ale tu si zde ukazovat nebudeme.

**Cvičení**

- Naprogramujte rekurzivní implementaci operací (strom se prochází shora dolů).
- Jak by vypadala implementace intervalového stromu pro maxima?

**Použití intervalového stromu**

Intervalový strom je silný nástroj, kterým se dá vyřešit spousta úloh. Ale než ho začnete používat, tak si vždy rozmyslete, zda úloha nelze řešit elegantněji bez intervalového stromu. Ne všechny druhy intervalových stromů se dobře implementují.

Intervalový strom obvykle použijeme, pokud potřebujme průběžně zjišťovat informace o intervalech a zároveň je i měnit. Pokud používáme jen jednu z těchto operací (a tu druhou jen zřídka), existuje často lepší řešení než intervalový strom – viz úvodní příklad.

**Fenwickův strom**

*Fenwickův strom,* někdy také nazývaný jako *finský strom,* je v podstatě jen strom reprezentovaný v poli. Jeho používání je podobné jako používání intervalového stromu pro součty. Rozdíl je jen v implementaci daných funkcí. My si Fenwickův strom opět ukážeme na úvodním příkladu. Zase tedy budeme potřebovat funkci pro změnu hodnoty v posloupnosti a funkci pro zjištění součtu na intervalu. (Ve skutečnosti zjistíme dva prefixové součty a z nich pak spočítáme výsledný interval.)

Fenwickův strom je trochu magická datová struktura. Abychom si tuto magii mohli užít, zvolíme trochu netradiční způsob vysvětlování a nejdříve si ukážeme, jak se Fenwickův strom implementuje a teprve pak si vysvětlíme, jak to všechno funguje.

Fenwickův strom bude pole velikosti $N+1$, kde index 0 nebudeme používat. Používat budeme pouze prvky $1, \ldots, N$, které všechny na začátku nastavíme na 0. Pokud v posloupnosti změníme hodnotu, stejně jako u intervalového stromu, ve Fenwickově stromě na některá místa přičteme rozdíl oproti předchozí hodnotě.

```
void pricti(unsigned int index, int rozdil) {
  while (index<=N) {
    strom[index] += rozdil;
    index = index + (index & -index);   // bitový and
  }
}
```

A zde je funkce pro zjištění prefixového součtu:

```
int prefSoucet(unsigned int index) {
  int soucet = 0;
  while (index>0) {
    soucet = soucet + strom[index];
    index = index & (index-1);
  }
  return soucet;
}
```

Toť celá implementace. No, nevypadá na první pohled magicky? Pokud chcete vědět, jak tohle celé funguje, tak čtěte dál.

Ve Fenwickově stromě je na indexu 1 uložen první prvek, na indexu 2 součet prvního a druhého, na indexu 3 třetí prvek na indexu 4 součet prvních čtyř, ... na indexu $N$ je uložen součet posledních $2^K$ hodnot, kde $K$ je pozice prvního jedničkového bitu v binárním zápise čísla $N$. Ve stromě máme tedy uloženou takovou pravidelnou strukturu intervalů.

Nyní se podíváme, co dělají naše magické funkce na posouvání ve stromě a pak najednou bude všechno jasné. Ve výrazu `index & (index-1)` z funkce `prefSoucet()` se neděje nic jiného než, že se vynuluje nejpravější jedničkový bit v indexu. Tím se dostaneme na první interval, který jsme ještě nepřičetli. V momentě, kdy se dostaneme na index 0, tak už máme dotazovaný interval kompletní a výpočet můžeme ukončit.

Výraz `index + (index & -index)` dělá to, že se v pomyslném stromě intervalů posune o úroveň výš. Pokud jsme tedy v intervalu o velikosti 2, tak se dostaneme do intervalu velikosti 4, který daný interval obsahuje (tento interval je jednoznačný). Samotný výpočet dělá to, že v čísle `index` vezme nejpravější jedničku a znova ji přičte.

Fenwickův strom se používá hlavně kvůli jednoduchosti jeho naprogramování a také kvůli efektivitě samotného výpočtu a nevelké náročnosti na paměť. Při jeho implementaci doporučujeme dávat si pozor na správnost bitových funkcí.

**Cvičení**

- Rozmyslete si, že oba magické výpočty opravdu dělají to, co mají, a také, proč vše vlastně funguje.

*Karel Tesař*

**Úloha 16-3-1: Fyzikova blecha**

Newtoon trénuje svoji blechu na bleší turnaj. Ten probíhá na svislé stěně, na které jsou vodorovné plošinky. Cílem blechy je slézt ze startovací polohy co nejdříve na podlahu. Pohyb blechy závisí na tom, zda je blecha na nějaké plošince, či padá. Pokud padá, klesne za jednu blechovteřinu o jeden blechometr. Pokud je na plošince, posune se za jednu blechovteřinu o jeden blechometr vlevo či vpravo.

Závod tedy probíhá tak, že blecha padá, padá, až dopadne na plošinku. Pak se rozhodne (nebo jí její majitel přikáže), zda půjde doleva nebo doprava, a jde, dokud nedojde na konec plošinky. Z ní pak seskočí a zase padá, dokud se nedostane na podlahu. Vítězí blecha, která přistane na podlaze jako první. Ovšem je nutné, aby žádná blecha nespadla z větší výšky než $v$ blechometrů, jinak se totiž po dopadu urazí a odmítne pokračovat v závodě.

Newtoon již vytrénoval svou blechu tak, že ho poslouchá na slovo. Problém je ten, že sám neví, jak blechu navigovat, aby prošla bludištěm nejkratší možnou cestou. Protože Vás ale zajímá, jak vypadá blecha, která poslouchá, rozhodli jste se Newtoonovi pomoci.

Na vstupu dostanete jednak počáteční souřadnice blechy (v celých blechometrech), $v$, což je největší výška, ze které může blecha spadnout, aby se neurazila, a $N$, což je počet plošinek na stěně. Dále dostanete popis $N$ plošinek, u každé plošinky souřadnice jejího horního dolního rohu a její šířku (vše opět v celých blechometrech). Všechny plošinky jsou vysoké jeden blechometr a žádné dvě se nedotýkají. Blecha je na podlaze, pokud se nachází na souřadnicích $[x; 0]$, kde $x$ je libovolné celé číslo.

Výstupem vašeho programu je nejmenší počet blechovteřin, které bude blecha potřebovat, aby se dostala na podlahu. Kromě tohoto počtu vypište i počet plošinek, na které blecha dopadne, a u každé plošinky (v pořadí, jak na ně blecha dopadá) rozhodněte, zda má blecha jít vlevo či vpravo. Pokud úloha nemá řešení (moc malé $v$), vypište odpovídající zprávu.

*Příklad:* Blecha se nachází na souřadnicích $[5; 12]$, $v = 4$, $N = 3$. Plošinky jsou $[3; 8]; 5$, $[3; 4]; 5$ a $[7; 6]; 3$ ([souřadnice levého horního rohu]; délka). Nejkratší cesta trvá 17 blechovteřin, blecha navštíví všechny tři plošinky a půjde vpravo na první, vlevo na druhé a vpravo na třetí navštívené plošince.