Řešení úlohy č. 1
Formace

1 Naivní řešení $\mathcal{O}(n^2)$

Tuto úlohu šlo vyřešit naivně tak, že jsme pro každý prvek x_i pole x našli maximum z množiny $\{x_s \mid i - k_i < s \leq i\}$, tj. z pole $x_{i-k_i+1}, x_{i-k_i+2}, \ldots, x_i$, přesně podle zadání. Protože k_i mohlo mít hodnotu až i, pro každý prvek x_i jsme udělali až i operací (nalezení maxima z k prvkového pole nám zabere právě k porovnání). Celkově jsme tedy udělali až

$$1 + 2 + \ldots + i + \ldots + n = \frac{n(n+1)}{2} = \mathcal{O}(n^2)$$

operací.

Nakonec zmíníme, že jsme schopni tento jednoduchý algoritmus zrychlit na $\mathcal{O}(n \log n)$, pokud bychom pro výběr maxima použili klouzavé okýnko implementované haldou.

2 Optimální řešení $\Theta(n)$

Z naivního řešení lze usoudit, kde je bottleneck našeho problému: hledání maxima pro každý prvek x_i. Budeme se tedy soustředit na zrychlení této části. Řešení v lineárním čase je pak úpravou *Queue modification (method 3)* z tohoto článku Minimum stack / Minimum queue. Nyní si tento algoritmus popíšeme.

Protože prvky $x_j, j > i$ nemohou nijak ovlivnit hledané maximum pro prvek x_i, můžeme vždy načíst prvek x_i a rovnou odpovědět výslednou hodnotou před načtením dalšího prvku x_{i+1}. Prvky si budeme ukládat ve frontě, která bude reprezentována pomocí 2 zásobníků. Ukažme si nejprve tuto implementaci.

Implementace fronty dvěma zásobníky

Označme jeden zásobník jako *Push zásobník* a druhý jako *Pop zásobník*. Abychom z těchto dvou zásobníků vytvořili frontu, budeme potřebovat naimplementovat operace:

- push(x), která přidá prvek x na konec fronty,
- pop(), která odebere prvek na začátku fronty,
- front(), která vrátí prvek na začátku fronty.

Operace push(x) je nejjednodušší: do Push zásobníku přidáme (pushneme) prvek x. Operaci pop() implementujeme následovně:

- Je-li Pop zásobník neprázdný, odeber (popni) prvek z vrcholu tohoto zásobníku a skonči.
- Je-li Push zásobník prázdný, pak skonči s chybou, neboť fronta je prázdná.
- "Přelej" Push zásobník do Pop zásobníku, tj. dokud není Push zásobník prázdný, tak postupně odebírej prvky z vrcholu Push zásobníku a přidávej je do Pop zásobníku.

- Nakonec odeber prvek z vrcholu Pop zásobníku.

Vyzýváme čtenáře, aby si tuto operaci zkusili nakreslit (*nápověda*: nakreslete si dva zásobníky jako chlívky, do kterých si budete uschovávat prvky).

Operaci front() naimplementujeme podobně jako operaci pop(), akorát místo odebírání prvku z fronty ho pouze vrátíme.

Že se tato datová struktura chová stejně jako fronta lze jednoduše nahlédnout z nakresleného obrázku. Skutečně, první prvek, který do této struktury přidáme pomocí operace push(x) se při první operaci pop() ocitne na vrcholu Pop zásobníku, ze kterého tento prvek nakonec vyjmeme. Stejný úsudek pak platí i pro druhý, třetí, . . . , vložený prvek.

Dále bychom měli ještě určit paměťové a časové nároky této struktury, které ale v tomto textu (prozatím) vynecháme. Laskavý čtenář si rozmyslí, že ani jedna z námi naimplementovaných operací neběží v čase $\mathcal{O}(1)$ (což nám ale nevadí!). Ve skutečnosti jsou všechny tyto operace tzv. *amortizovaně konstantní*, což je pojem, který čtenář může znát např. z Průvodce labyrintem algoritmů

Implementace maximové fronty

S nyní navrhnutou frontou pomocí 2 zásobníků můžeme zkonstruovat tzv. *maximovou frontu*. To je modifikace fronty, která kromě již podporovaných operací, navíc umožňuje z uložených hodnot vrátit maximum v čase $\mathcal{O}(1)$.

Abychom toho docílili, přidáme operaci max(), která ono maximum z fronty vybere. Na nalezení maxima z n prvků uložených ve frontě, bychom museli projít celou frontu a podívat se na každý takový prvek, což nám ale celkově zabere $\mathcal{O}(n)$ operací a to je pomalé. Bottleneck našeho problému je procházení všech prvků fronty. Abychom se tomuto zběsilému procházení vyvarovali při každém zavolání max(), mohli bychom si založit proměnnou m, která by uchovávala takové maximum, a při každé operaci push(x) by se prvek x akorát navíc porovnal s hodnotou uloženou v m a případně by se hodnota m přepsala novou hodnotou x. Tento algoritmus je jistě konstantní (vždy bychom pouze odpověděli hodnotou m), ale bohužel není funkční. Stačí, aby nám pop() odebrala maximální prvek fronty a jsme nahraní.

Navrhneme tedy trochu paměťově náročnější řešení. Předtím si ale zavedeme tři pojmy.

- *Lokální maximum zásobníku viděné prvkem x* je maximum z množiny všech prvků, které na daném zásobníku leží "pod"prvkem x včetně prvku x.
- *Lokální maximum zásobníku viděné vrcholem* je lokální maximum zásobníku viděné prvkem, který se nachází na vrcholu daného zásobníku. Čtenářům nyní doporučujeme si poslední dvě věty párkrát přečíst, nakreslit si zásobník a na něm si vyznačit nějaká tato lokální maxima.
- *Maximum zásobníku* je prvek, jehož hodnota je mezi všemi prvky na zásobníku největší.

Nyní upravíme operace push(x) a pop() a navrhneme funkční operaci max(), která bude počítat i s případným odebráním maxima z fronty. Začneme operací push(x). Vždy, když budeme chtít přidat do fronty (tj. do Push zásobníku) prvek x, tak místo toho, abychom ho přidávali takto osamocený, do fronty přidáme pár hodnot $\{x, m\}$, kde m je lokální maximum zásobníku viděné prvkem x. Je-li Push zásobník prázdný, pak triviálně přidáme pár $\{x, x\}$. Jsou-li na Push zásobníku již nějaké prvky, **které byly korektně vložené**, pak lze předpokládat, že maximum Push zásobníku je právě lokální maximum viděné vrcholem tohoto zásobníku. Při přidávání prvku x do neprázdného Push zásobníku, tak lokální maximum viděné prvkem x je nutně buď prvek x sám, nebo lokální maximum viděné vrcholem před samotným přidáním prvku x. Čtenáře bychom rádi upozornili na fakt, že byli právě svědky neformálního mini-důkazu matematickou indukcí :-). Upravený algoritmus push(x) tedy bude vypadat následovně:

- Je-li Push zásobník prázdný, pak přidej pár $\{x, x\}$.
- Jinak přidej na Push zásobník pár $\{x, m\}$, kde m je maximum z x a lokálního maxima Push zásobníku viděného vrcholem.

Dále upravíme funkci pop(). U ní jediná změna oproti předchozí implementaci bude v "přelévání". Kdykoliv začneme přelévat prvky z Push zásobníku do Pop zásobníku (tj. dokud není Push zásobník prázdný, postupně odebírej prvky z Push zásobníku a přidávej je do Pop zásobníku), stane se tak ve chvíli, kdy je Pop zásobník prázdný (pokud by prázdný nebyl, pop() by odebrala prvek z Pop zásobníku a nebylo by nutné prvky "přelévat"). Můžeme tak pro "přelévání" využít upravenou operaci push(), která ale nebude přidávat prvky do Push zásobníku, nýbrž do Pop zásobníku. Upravená operace pop() tak může vypadat následovně:

- Je-li Pop zásobník neprázdný, odeber (popni) prvek z vrcholu tohoto zásobníku a skonči.
- Je-li Push zásobník prázdný, pak skonči s chybou, neboť fronta je prázdná.
- "Přelej" Push zásobník do Pop zásobníku, tj. dokud není Push zásobník prázdný, tak:
  - Odeber prvek $\{x, m\}$ z vrcholu Push zásobníku.
  - Je-li Pop zásobník prázdný, pak přidej pár $\{x, x\}$.
  - Jinak přidej na Pop zásobník pár $\{x, m\}$, kde m je maximum x x a lokálního maxima **Pop** zásobníku viděného vrcholem.
- Nakonec odeber prvek z vrcholu Pop zásobníku.

Proč je operace "přelévání" definovaná právě takto, se dozvíte v následujícím odstavci.

Řešení

Tvrzení 1. *Maximum z neprázdné maximové fronty lze získat v čase $\mathcal{O}(1)$ jako větší z hodnot maxima viděného vrcholem Push zásobníku a maxima viděného vrcholem Pop zásobníku. Je-li Push, resp. Pop, zásobník prázdný, pak maximum fronty je maximem viděným vrcholem Pop, resp. Push, zásobníku.*

Důkaz. Je-li Push zásobník prázdný, pak se maximum fronty určitě nachází mezi prvky Pop zásobníku. Z odstavce o implementaci operace push() jsme si ale rozmysleli, že maximum z jediného zásobníku se nutně nachází na vrcholu takového zásobníku, tedy v našem případě na vrholu Pop zásobníku. Stejný argument platí i obráceně, tj. je-li Pop zásobník prázdný a Push zásobník nutně neprázdný. Uvažme nyní, že je Pop i Push zásobník neprázdný. Prvky uložené ve frontě jsou disjunktně uložené v Push a Pop zásobnících, jinak řečeno Push a Pop zásobníky rozdělují všechny prvky fronty do dvou nepřekrývajících se množin X_{Push} a X_{Pop}. Chceme-li znát maximum z těchto dvou množin, stačí nám vybrat maximum z každé z nich a pak tyto maxima porovnat a vybrat to větší. Protože ale výběr maxima množiny X_{Push}, resp. množiny X_{Pop}, znamená vybrat maximum Push, resp. Pop, zásobníku, které zvládneme vybrat v konstantním čase a porovnání dvou prvků nám zabere také konstantní čas, je celá operace max() konstantní.

Nyní si musíme rozmyslet, že nám případné operace push(x), pop() tuto implementaci nerozbijou, tj. že max() bude vždy pracovat správně. Nechť se (stále neprázdná) fronta nachází v nám neznámém stavu a nechť hodnota m_{Push} určuje maximum z nynějšího Push zásobníku, m_{Pop} z nynějšího Pop zásobníku a hodnota m maximum z celé fronty. Přidáme-li do fronty operací push(x) prvek x, pak v lokálním maximu Push zásobníku viděného vrcholem, bude uložené maximum Push zásobníku. Je-li tedy prvek x větší než m, pak ho při zavolání funkce max() opravdu najdeme. Je-li prvek x menší než m, pak buď je větší než m_{Push} a pak je maximem hodnota m_{Pop}, kterou určitě funkce max() najde, nebo je menší než m_{Push}, a proto je maximem buď hodnota m_{Push} nebo m_{Pop}, kterou taktéž funkce max() najde. Operace push(x) tedy určitě max() nerozbije.

Že operace pop() také nerozbíjí hledání maxima ověříme podobně. Důležité je si nyní uvědomit, co přesně nám poskytuje lokální maximum viděné nějakým prvkem. Předpokládejme, že Pop zásobník je neprázdný. Operace pop() odebere vrchol Pop zásobníku, přičemž mohli nastat tyto situace:

- Vrchol Pop zásobníku neudával maximum fronty, a proto platilo $m = m_{\text{Push}}$.
- Vrchol Pop zásobníku udával maximum fronty, tj. platilo $m = m_{\text{Pop}}$.

Pokud nastala první situace, pak funkce max(), bez ohledu na nynější stav datové struktury, maximum jistě nalezne správně. Pokud nastala druhá situace, pak se buď Pop zásobník stal prázdným, a funkce max() správně nalezne maximum z Push zásobníku, nebo na Pop zásobníku zůstaly ještě nějaké prvky, ale **lokální maximum Pop zásobníku viděné vrcholem po provedení operace** pop() **nám stále udává maximum Pop zásobníku** (rozmyslete si, jak se prvky na Pop zásobník dostaly a co se stalo s jejich lokálními maximy, případně si přečtěte další odstavec), a proto maximum fronty znovu nalezneme správně jako větší z prvků lokálních maxim Push a Pop zásobníků viděné jejich vrcholy.

Pokud je Pop zásobník prázdný a my zavoláme operaci pop(), pak dojde k "přelití"prvků z Push zásobníku do Pop zásobníku. Zde je zásadní si uvědomit, co se stalo s lokálními maximy viděnými jednotlivými prvky. Pokud jsme měli původně na Push zásobníku prvky $x_1, \ldots, x_n$ přidávané ve stejném pořadí, pak lokální maximum Push zásobníku viděného prvkem x_1 bylo maximum z množiny $\{x_1\}$, lokální maximum Push zásobníku viděného prvkem x_2 bylo maximum z množiny $\{x_1, x_2\}$, pro x_3 to bylo maximum z množiny $\{x_1, x_2, x_3\}$, pro x_n to bylo maximum z množiny $\{x_1, \ldots, x_n\}$. Po přelití se situace obrátila. Prvky jsou na Pop zásobníku uloženy v pořadí $x_n, \ldots, x_1$ od "nejvyššího"po "nejspodnější"a pro jejich lokální maxima nyní platí: pro x_n je to maximum z množiny $\{x_n\}$, pro x_{n-1} je to maximum z množiny $\{x_{n-1}, x_n\}$, pro x_1 je to maximum z množiny $\{x_1, x_2, \ldots, x_n\}$ a **hlavně pro x_2 je jeho lokální maximum rovno maximu z** $\{x_2, x_3, \ldots, x_n\}$. Proto jakmile operace pop() odebere vrchol Pop zásobníku, tj. pár s prvkem x_1, tak maximum fronty najdeme (díky prázdnému Push zásobníku) jako lokální maximum viděného vrcholem Pop zásobníku, nebo chcete-li jako lokální maximum viděného prvkem x_2, což je přesně to, co nalezne i funkce max(). □

Tvrzení 2. *Problém Formace jsme schopni vyřešit za lineární čas.*

Důkaz. S maximovou frontou už snadno navrhneme jednoduchý algoritmus. Je-li pole x prázdné, pak nemusíme nic dělat, proto předpokládejme, že se v poli x nachází alespoň jeden prvek. Ve frontě si v i-té iteraci budeme držet vždy k_i prvků, ze kterých máme právě vybírat onen maximální prvek.

- Zavolej push(x_1) a vypiš na výstup x_1.
- Pro všechny $i = 2, 3, \ldots, n$:
  - Je-li $k_i > k_{i-1}$, pak zavolej push(x_i).

- Jinak odeber z fronty $k_{i-1} - k_i + 1$ prvků, tj. zavolej tolikrát funkci pop().
- Zavolej funkci max() a vypiš vrácenou hodnotu na výstup.

Rychle rozebereme funkčnost algoritmu. Pro prvek x_1 je jistě správným výstupem sám x_1, protože $k_1 = 1$, neboť $k_1 \geq 1$ dle zadání a zároveň $k_1 \leq 1$, protože nelze šahat na prvky mimo pole x. Dále když $k_i > k_{i-1}$, tak budeme chtít vybírat maximum z fronty definované na konci $i - 1$. iterace s přidaným prvkem x_i. Naopak pokud $k_i \leq k_{i-1}$, tak budeme z fronty nějaké prvky odebírat, konkrétně jich budeme chtít odebrat $k_{i-1} - k_i + 1$. Např. pokud $k_i = k_{i-1}$, tak potřebujeme odebrat právě jeden prvek, což nám uvedená formulka krásně splňuje. Povšimněte si, že fronta po vložení úvodního prvku x_1 už nikdy v průběhu algoritmu nebude prázdná! Funkce max() tak vždy skončí s úspěšně vráceným maximálním prvkem. Funkce max() správně vybírá maximum z fronty, proto pro každý prvek x_i vrátí odpovídající maximální prvek.

Nakonec nám chybí odvození časové složitosti. Tato část může být pro čtenáře zajímavá, protože operace push(x) ani pop() není konstantní a alespoň jednu z těchto operací voláme pro každý prvek x_i. Nejméně efiktivní část našeho algoritmu je část "přelévání"z Push do Pop zásobníku. V každé iteraci můžeme takto přelét až n prvků, tedy pro každý prvek uděláme $\mathcal{O}(n)$ operací, a protože prvků je n, je složitost algoritmu $\mathcal{O}(n^2)$. Přestože toto je pravda, není to přesně to, co bychom chtěli ukázat. Abychom dokázali, že náš algoritmus skutečně běží v lineárním čase, budeme se muset podívat ještě jednou dovnitř do implementace námi navrhnuté datové struktury a jejich operací push(x) a pop(). Zásadním pozorováním bude, že každý prvek x bude vždy nanejvýš jednou v Push zásobníku a nanejvýš jednou v Pop zásobníku, tj. pokud prvek x opustí Push zásobník (tedy bude "přelit"do Pop zásobníku) už ho nikdy znovu do Push zásobníku nepřidáme. To stejné platí i pro Pop zásobník. Pokud prvek x bude jednou odebrán z Pop zásobníku, tak už se do něj nikdy znovu nevrátí. Každý prvek tak bude nejvýše jednou vložen do Push zásobníku, nejvýše jednou přelit do Pop zásobníku a nejvýše jednou z Pop zásobníku odebrán. Celkově tedy někde v průběhu algoritmu bude na každém prvku provedeny 3 operace přesunu, což je konstantně mnoho! Nevíme sice přesně kdy v průběhu algoritmu se daný prvek přesune, ale to nás vůbec nemusí zajímat! Pro nás je důležité, že pro každý prvek provedeme konstantně mnoho operací za **celý** algoritmus. Celkově pak dostáváme, že pro každý prvek, kterých je n, provedeme $\mathcal{O}(1)$ operací, a tedy celková složitost je $\mathcal{O}(n)$. Dokonce je časová složitost algoritmu rovná $\Theta(n)$, protože potřebujeme vždy přečíst celý vstup, všech n prvků a musíme tedy vykonat $\Omega(n)$ (čtěte "alespoň $c \cdot n$ operací"pro nějakou konstantu $c \in \mathbb{R}$). $\square$

Tvrzení 3. *Navržený algoritmus běží za lineární čas a je to algoritmus optimální, tj. neexistuje algoritmus, který by problém Formace vyřešil rychleji než v lineárním čase.*

Důkaz. Námi navržený algoritmus běží dle předchozího odstavce skutečně v lineárním čase a protože vždy musíme přečíst celý vstup, musíme vykonat alespoň lineární počet operací. Asymptoticky rychlejší algoritmus by nezvládl přečíst celý vstup, proto složitost našeho algoritmu je nejlepší možná a algoritmus je skutečně optimální. $\square$