Základní myšlenka

Jako základní myšlenku začnu nejhrubším (prvním) algoritmem co mě napadl. Jednoduše projdeme celý vstup a najdeme nejvyšší číslo, poté vstup projdeme podruhý a zapíšeme všechny indexy nejvyššího čísla do nové arraye. Potom bychom neustále iterovali přes tento nový array a porovnali *+1, +2, . . . , +i* index vstupu pro každý uložený index, dokud by nezbyl jen jeden.

Až na problém vstupů s opakující čísly, tento algoritmes je také dost pomalý. Na vstup, kde by byly všude dvojky až na jednu jedničku, by musel *n-1* krát projít array postupně zmenšující od *n-1* do *1*, tedy $\frac{(n-1)n}{2}$ iterace, což z hlediska časové složitosti vychází kvadraticky. Ale to jsme ještě nezapočítali jednu klíčovou část, odstraňování z arraye, která zvýšší časovou složitost z n^2 na n^3 a to opravdu není hezké.

První optimalizace

Používaní arraye nám výrazně zhoršilo složitost, tak tedy použijeme něco jiného, ideálně něco co má konstantní vymazávaní. První co mě napadlo byl linked list. V této úloze se linked list dá krásně využít, a abych mohl jednoduššeji popsat výhody, napíšu ještě příklad algoritmu.

První část algoritmu zůstává stejný, projdeme celý vstup a najdeme nejvyšší číslo, které určí jaké indexy nás zajímá. Potom co budeme mít nejvyšší číslo, opět projedem od začátku vstup a uložíme každý index jako uzel do našeho linked listu. Náš linked list bude cyklický (poslední uzel se bude vázat na první) a bez újmy na obecnosti určíme první uzel jako první index, který chceme přidat. Každý uzel bude mít uložený další uzel, předchozí uzel, původní index vstupu (na výstup) a aktuální index, který porovnáváme s ostatní uzly.

A teď si ukážeme proč takový linked list funguje: V první iteraci budeme porovnánat každý index *+1*. Začeneme od námi určený první uzlem a porovnáme ho jenom s následujícím. Pokud jsou obě hodnoty na *+1* indexy stejné, pokračujeme dál, jinak vymažeme uzel s nižší hodnotou na *+1* index a případně upravíme jaký uzel definujeme jako první. Druhé porovnání proběhne stejně, akorát že si můžeme všimnout, že pokud další (třetí) uzel má větší hodnotu na *+1* index a první a druhý uzel mají stejné hodnoty, podle dřívějšího popisu vymažeme jenom druhý uzel a první uzel tam zůstane (chybně). Tady ale snadno nahlédneme, že pokud j-tý uzel má menší hodnotu na *+i* index, než *(j+1)*-tý uzel, všechny uzly od prvního k j-tému budou mít menší hondotu na *+i* index, tudíž je můžeme všechny vymazat. Kvůli tomu je důležítý mít definovaný první uzel, abychom věděli kde zastavit.

Takhle nám funguje algoritmus v podstatě stejně jako v základní myšlence, akorát že pomocí jiné datové struktury zachováme kvadratickou časovou složitost. Také stále není schopný vyřešit opakující se vstup.

Druhá optimalizace

Sice jsme výrazně vylepšili časovou složitost, ale pro úlohu tohoto typu to určitě jde ještě zlepšit, ale jak? První co jsem si uvědomil bylo, že abychom jednoznačně určili nejlepší index na zapínání, je potřeba porovnat celý úsek od jednoho uzlu k dalšímu uzlu. Musíme tedy vylepšit porovnávání. Pro lepší pochopení určím uzel n jako ten, pro který teď zjišťujeme $+i$-tý index. Dále budeme mít uzel m, které zrovná začíná na $+i$-tém indexu uzlu n. Z našeho algoritmu víme, že v předchozím kroku jsme určili, že všechny uzly mají stejné hondoty, které jsou také nejlepší možné hodnoty. Dále si uvědomíme, že to znamená, že jestliže uzel m je pokračování uzlu n, je to nejlepší možné pokračování. Totiž, kdyby existovalo lepší pokračování, tak bychom ho už našli a určili jako lepší úsek než ty ostatní uložené uzly.

Tedy, do našeho algoritmu přidáme podmínku, že pokud $+i$-tý index od začátku uzlu se zrovna rovná začátku následujícího uzlu (uložili jsme je v pořadí podle indexu), můžeme ten celý druhý uzel smazat. Takové přirovnání bychom provedli pro všechny další uzly, ale ještě včetně uzlu, který teď "snědlo" následující uzel. Toto nám trochu urychlí algoritmus v případě, že by byl takhle cyklický celý vstupní pole. Po tomto procesu by zůstali jenom ty uzly, které také "snědli" jejich následující uzel. Poté bychom mohli pokračovat rovnou po "sněženém" uzlu, tedy od $+2i$. Takhle bychom prošli každý úsek původní pole jenom jednou.

Tento algoritmus už bych bral jako nejlepší, protože je lineární a nelze najít nejlepší úsek podle nějakých požadavků lépe než lineárně, protože musíme aspoň nahlédnout na každý prvek. Teď si ale projdeme každým krokem algoritmu a určili jeho komplexitu.

Samozřejmě, najít nejvyšší číslo a poté najít všechny instance nejvyššího čísla můžeme dělat při dvou procházení pole, ale také je můžeme sloučit do jednoho, jestli nám nevadí mazání nedokončeného linked listu. I kdybychom museli smazat dost velký linked list, stále by to bylo míň operací než dvakrát procházení pole, dokud nezačneme počítat komplexitu jednolivých operací... ale to zanedbáme. Jediné co zbývá je procházení linked listu a tím pádem celé pole. Procházení linked listu samotné nemusíme řešit, protože každý uzel přísluší jednomu indexu v poli. Dále porovnáváme jenom indexy, které nejsou uzly. V případě, že ten index je uzel, tak už jsem popsal proces "snězení" a pokračovali bychom dále s indexem, který není uzel. Tedy celkém s linked listem bychom prošli celé pole právě jednou, což je krásně lineární.

Celková časová složitost je tedy jasně $\mathcal{O}(n)$. Prostorová složitost také vychází jako $\mathcal{O}(n)$, protože si ukládáme jenom ten linked list (a konstantní množství vlastností uzlů) a ten linked list má maximální délku n.