1. Ukládání mezivýsledků

V předchozí části knihy jsme potkali řadu algoritmů. Některé z nich přistupovaly k úloze naprosto přímočaře – například algoritmy pro práci s dlouhými čísly pouze formálně popisovaly, jak jsme s čísly odedávna zvyklí zacházet. Jen jsme si přitom přenosy mezi řády nemumlali pod vousy, ale spořádaně ukládali do pomocných proměnných.

Často se ovšem ukázalo, že umíme dosáhnout daleko lepší časové složitosti. Někdy pomohl šikovný trik, jindy zase stačilo podívat se na problém odlišným způsobem. Například jsme zavedením haldy zrychlili třídění z $\Theta(N^2)$ na $\Theta(N \log N)$ kroků. Takový algoritmus může na první pohled připomínat kouzelnické číslo – mistr magie vytáhl králíka z klobouku a na nás je, abychom mlčky obdivovali.

Obdiv je samozřejmě na místě, neboť pěkný algoritmus má mnoho společného s fortelným řemeslem i s uměním. Pojďme se ale raději naučit něco o tom, jak elegantní algoritmy vytvářet sami. Žádný „univerzální recept na dokonalá řešení úloh“ sice neznáme, ale i tak lze nalézt celou řadu obecných návrhových technik. V následujících kapitolách si ty nejdůležitější z nich předvedeme.

Začněme tou nejsnazší, založenou na prostém odstranění opakovaných výpočtů ukládáním vhodných mezivýsledků.

1.1. Přidávání na konec

Úsek s maximálním součtem

Akcionáři firmy O. Škubal a synové měli jisté podezření, že firma v posledních letech neprosperuje tak, jak by si představovali. Hlavní účetní se je nicméně rozhodl přesvědčit o opaku. Vypsal si posloupnost zisků v jednotlivých dnech minulého roku (některé mohou samozřejmě být záporné) a chce v ní nalézt takové období, aby celkový zisk v něm dosažený byl co možná největší.

Řešíme tedy následující úlohu: máme zadanou nějakou posloupnost $x_1, \ldots, x_N$ celých čísel a chceme v ní nalézt *úsek* (tím myslíme souvislou podposloupnost), jehož součet je největší možný. Takovému úseku budeme říkat *nejbohatší.* Jako výstup nám postačí hodnota jeho součtu, nebude nutné ohlásit přesnou polohu úseku.

Nejprve si rozmyslíme triviální případy: Kdyby se na vstupu nevyskytovalo žádné záporné číslo, má evidentně maximální součet celá vstupní posloupnost. Pokud by naopak byla všechna x_i záporná, nejlepší je odpovědět prázdným úsekem, který má nulový součet; všechny ostatní úseky mají součet záporný.

Obecný případ bude komplikovanější: například v posloupnosti

$$1, -2, 4, 5, -1, -5, 2, 7$$

najdeme dva úseky kladných čísel se součtem 9 (totiž $4, 5$ a $2, 7$), ale dokonce se hodí spojit je přes záporná čísla $-1, -5$ do jediného úseku se součtem 12. Naopak hodnotu

-2 se použít nevyplácí, jelikož přes ní je dosažitelná pouze počáteční jednička, takže bychom si o 1 pohoršili.

Nejpřímočařejší možný algoritmus by téměř doslovně kopíroval zadání: Vyzkoušel by všechny možnosti, kde může úsek začínat a končit, pro každou z nich by spočítal součet prvků v úseku a pak našel z těchto součtů maximum. Časovou složitost není těžké odhadnout: máme řádově N^2 dvojic (*začátek*, *konec*) a pro každou z nich v čase $\mathcal{O}(N)$ spočítáme součet a upravíme průběžné maximum. To je celkem $\mathcal{O}(N^3)$.

Algoritmus MaxSoučet1

Vstup: Posloupnost $X = x_1, \ldots, x_N$ uložená v poli.

Výstup: Součet M nejbohatšího úseku v X.

1. $M \leftarrow 0$ *(zatím jsme potkali jen prázdný úsek)*
2. Pro $i = 1, \ldots, N$ opakujeme: *(i je začátek úseku)*
3. Pro $j = i, \ldots, N$ opakujeme: *(j je konec úseku)*
4. $s \leftarrow 0$ *(součet úseku)*
5. Pro k od i do j opakujeme:
6. $s \leftarrow s + x_k$
7. $M \leftarrow \max(M, s)$

Podívejme se nyní, čím tento algoritmus tráví nejvíce času. Jistě počítáním součtů. Například sčítá jak úsek $x_i, \ldots, x_j$, tak $x_i, \ldots, x_{j+1}$, aniž by využil toho, že druhý součet je o x_{j+1} vyšší než ten první. Nabízí se tedy zvolit pevný začátek úseku i a vyzkoušet všechny možné konce j od nejlevějšího k nejpravějšímu. Každý další součet pak dovedeme spočítat z předchozího v konstantním čase. Pro jedno i tedy strávíme čas $\mathcal{O}(N)$, celkově pak $\mathcal{O}(N^2)$.

Algoritmus MaxSoučet2

Vstup: Posloupnost $X = x_1, \ldots, x_N$ uložená v poli.

Výstup: Součet M nejbohatšího úseku v X.

1. $M \leftarrow 0$ *(zatím jsme potkali jen prázdný úsek)*
2. Pro $i = 1, \ldots, N$ opakujeme: *(i je začátek úseku)*
3. $s \leftarrow 0$ *(součet úseku)*
4. Pro $j = i, \ldots, N$ opakujeme: *(j je konec úseku)*
5. $s \leftarrow s + x_j$
6. $M \leftarrow \max(M, s)$

Myšlenka průběžného přepočítávání se ale dá využít i lépe, totiž na celou úlohu. Uvažme, jak se změní výsledek, když ke vstupu $x_1, \ldots, x_N$ přidáme ještě x_{N+1}. Všechny úseky z původního vstupu zůstanou zachovány a navíc k nim přibudou nové úseky $x_i, \ldots, x_{N+1}$. Stačí tedy ověřit, zda součet některého z nových úseků nepřekročil dosavadní maximum, čili porovnat toto maximum se součtem nejbohatšího *koncového* úseku v nové posloupnosti.

Nejbohatší koncový úsek neumíme najít v konstantním čase, ale umíme ho velmi snadno při rozšíření vstupu přepočítat. Pokud přidáme x_{N+1}, prodlouží se o tento nový prvek všechny dosavadní koncové úseky a navíc se objeví nový jednoprvkový úsek. Maximální součet proto získáme buďto přičtením x_{N+1} k předchozímu maximálnímu součtu, nebo jako hodnotu x_{N+1} samotnou. (To druhé může být výhodnější například tehdy, měly-li zatím všechny koncové úseky záporné součty.)

Označíme-li si tedy K maximální součet koncového úseku, přidáním nového prvku se tato hodnota změní na $\max(K + x_N, x_N) = x_N + \max(K, 0)$. Jinými slovy počítáme průběžné součty, jen pokud součet klesne pod nulu, tak ho vynulujeme. Hledaný maximální součet M je pak maximem ze všech průběžných součtů. Tímto principem se řídí náš třetí algoritmus:

Algoritmus MaxSoučet3
Vstup: Posloupnost $X = x_1, \ldots, x_N$ uložená v poli.
Výstup: Součet M nejbohatšího úseku v X.

1. $M \leftarrow 0$ *(prázdný úsek je tu vždy)*
2. $K \leftarrow 0$ *(maximální součet koncového úseku)*
3. Pro i od 1 do N opakujeme:
4. $\quad K \leftarrow \max(K, 0) + x_i$
5. $\quad M \leftarrow \max(M, K)$

V každém průchodu cyklem nyní trávíme přepočítáním proměnných K a M pouze konstantní čas. Celkem má tedy náš algoritmus časovou složitost $\mathcal{O}(N)$. Hodnoty ze vstupu navíc potřebuje jen jednou, takže je může číst postupně a vystačí si tudíž s konstantní pamětí.

Dvojice s daným součtem

V minulém příkladu se nám osvědčilo řešit úlohu tak, že jsme začali s prázdným vstupem a příslušným triviálním výstupem, načež jsme postupně přidávali další prvky vstupu a přepočítávali výstup. Nakonec jsme dospěli k celému zadání, a tedy i k celé odpovědi. To je poměrně častý přístup, vysloužil si dokonce svůj vlastní název *inkrementální algoritmus.* Vyzkoušíme si ho proto ještě na jednom problému.

Známá hračkářská firma Děd Vševěd a vnukové vyrábí mechanické krokodýly na míru. Mají připraveno M hlav o rozměrech $x_1 < \ldots < x_M$ a N ocasů délek $y_1 < \ldots < y_N$. Když přijde zákazník a přeje si krokodýla délky D, hračkáři najdou ve svých zásobách vhodnou hlavu a ocas tak, aby jejich délky daly dohromady právě určené D.

Poněkud matematičtěji bychom situaci mohli popsat tak, že máme zadané dvě rostoucí posloupnosti čísel a ptáme se, zda lze z každé z nich vybrat po jednom prvku tak, aby součet těchto prvků byl roven danému číslu D.

Začneme opět naivními algoritmy: Máme M možností, jak zvolit první číslo, a N možností, jak zvolit druhé, takže v čase $\mathcal{O}(MN)$ hrubou silou vyzkoušíme všechny dvojice. To je poměrně pomalé, ale ještě jsme nevyužili toho, že čísla na vstupu

jsou setříděná. Stačí probírat všechna x_i a pro každé z nich zjistit, zda se mezi ocasy (totiž ypsilony) nachází hodnota $D - x_i$. Jelikož délky ocasů tvoří rostoucí posloupnost, můžeme přizvat na pomoc binární vyhledávání a to nám odpoví v čase $\mathcal{O}(\log N)$. Celkově tedy provedeme $\mathcal{O}(M \log N)$ kroků.

Předchozí algoritmus v sobě skrývá zárodek inkrementální úvahy, která nás dovede až k lineárnímu řešení. Všimněte si, že začínáme vstupem, v němž je jedna hlava a všechny ocasy, a postupně přidáváme další hlavy. Pojďme tedy při přidávání hlavy x_{N+1} využít něčeho, co jsme si zapamatovali během přidávání x_N.

To něco bude pozice, na které jsme se zastavili při hledání $D - x_N$ mezi ypsilony. Nepoužijeme tentokrát binární vyhledávání, nýbrž budeme ocasy procházet od nejdelšího jeden po druhém a zastavíme se, jakmile jejich délka klesne pod $D - x_N$. Když nyní hledáme $D - x_{N+1}$, všimneme si, že se nemůže nacházet mezi ocasy, které jsme už prošli. To proto, že $x_{N+1} > x_N$, takže $D - x_{N+1} < D - x_N$. Postačí tedy pokračovat v hledání od bodu, kde jsme se zastavili minule. Zde je hotový algoritmus:

Algoritmus KROKODÝLI

Vstup: Posloupnosti $x_1 < \ldots < x_M$, $y_1 < \ldots < y_N$ v polích, číslo D.

Výstup: Indexy i, j takové, že $x_i + y_j = D$, nebo zpráva, že neexistují.

1. $j \leftarrow N$
2. Pro i od 1 do M opakujeme:
3. Dokud $j > 1$ a $x_i + y_j > D$, snižujeme j o 1.
4. Je-li $x_i + y_j = D$, ohlásíme dvojici (i, j) a zastavíme se.
5. Oznámíme, že hledaná dvojice neexistuje.

Spočítejme časovou složitost. Na první pohled se sice zdá, že dosahuje $\Theta(MN)$, protože vnější cyklus proběhne M-krát a vnitřní cyklus až N-krát. Snadno ovšem nahlédneme, že složitost je ve skutečnosti lineární. Index j se totiž při každém průchodu vnitřním cyklem sníží o 1 a nikdy se nezvyšuje, tudíž vnitřním cyklem projdeme dohromady nejvýše N-krát. Trávíme tam proto čas $\mathcal{O}(N)$, ve zbytku algoritmu $\mathcal{O}(M)$, a celkem tedy $\mathcal{O}(M + N)$.

Cvičení:

1. Upravte algoritmus MAXSOUČET3, aby oznámil nejen maximální součet, ale také polohu příslušného úseku.
2. Krokodýlí algoritmus bychom mohli zapsat i pěkně symetricky: Začneme s $i = 1$ a $j = N$ a pak podle toho, zda je zrovna $x_i + y_j$ menší nebo větší než D, vždy buď zvýšíme i nebo snížíme j. Ukažte, že i tento algoritmus dává správný výsledek.
3. Jak by vypadala sada co nejmenšího počtu hlav a ocasů, z nichž by šli složit všichni krokodýli délek $1, \ldots, N$?
4. Úloha s nejbohatším úsekem byla pro kladné vstupy triviální. Všimněte si, že v úloze o krokodýlech na znaménkách nezáleží. Ukažte, jak libovolný vstup snadno upravit na takový, ve kterém jsou pouze kladná čísla, aniž by se změnila poloha hledané dvojice.

5. Úsek posloupnosti je *vyvážený*, pokud se v něm vyskytuje stejný počet kladných čísel jako záporných. Vymyslete algoritmus na nalezení nejdelšího vyváženého úseku.

6.* Úsek je *k-hladký* (pro $k \geq 0$), pokud se každé dva jeho prvky liší nejvýše o k. Popište co nejefektivnější algoritmus pro hledání nejdelšího k-hladkého úseku.

7. Jak spočítat kombinační číslo $\binom{N}{k}$? Výpočtu přímo podle definice brání potenciálně obrovské mezivýsledky (až $N!$), které se nevejdou do celočíselné proměnné. Navrhněte algoritmus, který si vystačí s čísly nejvýše rovnými N-násobku výsledku.

8.* Navrhněte algoritmus, který spočítá $\binom{N}{k} \bmod M$ a vystačí si přitom s čísly, která nepřekročí $\max(M^2, N)$.

9.** Jak vyřešit předchozí cvičení efektivněji, víme-li, že M je mnohem menší než N?

1.2. Předvýpočet

Úsek s daným součtem

U inkrementálních algoritmů jsme se opakovaných výpočtů zbavili tím, že jsme úlohu rozdělili na podúlohy, které si byly podobné. Mohli jsme tak z odpovědi na jednu podúlohu rychle spočítat odpověď na tu následující. Tímto způsobem přelstíme nejeden zákeřný problém, ale často je potřeba hledat i jiné podúlohy než jenom různě dlouhé části původního vstupu. Pojďme si to vyzkoušet na příkladu.

Jak už se v této kapitole stává tradicí, je opět zadána nějaká posloupnost $a_1, \ldots, a_N$, tentokrát kladných čísel, a číslo S. Chceme zjistit, zda v této posloupnosti existuje úsek, jehož součet je roven S.

Zamysleme se nejdříve nad úseky začínajícími prvkem a_1 – těm budeme říkat *prefixy* posloupnosti. Součty těchto úseků $p_i = a_1 + \ldots + a_i$ spočítáme lehce v čase $\mathcal{O}(N)$, jelikož $p_{i+1} = p_i + a_{i+1}$ a $p_0 = 0$ (prázdný prefix).

Jakmile známe součty všech prefixů, můžeme z nich spočítat součty všech ostatních úseků. Libovolný úsek totiž můžeme vyjádřit jako rozdíl dvou prefixů:

$$a_i + a_{i+1} + \ldots + a_j = (a_1 + a_2 + \ldots + a_j) - (a_1 + a_2 + \ldots + a_{i-1}) = p_j - p_{i-1}.$$

Prvky $a_1, \ldots, a_{i-1}$ jsme jednou přičetli a jednou odečetli, a proto výsledek neovlivní. Nesmíme zapomínat na prázdný úsek, ten získáme, kdykoliv $i = j + 1$.

Pro nalezení úseku se součtem S tedy stačí hledat dvojici indexů i, j takových, že $p_j - p_{i-1} = S$. Tedy téměř – skrývá se tu jistý háček: zatímco součet každého úseku odpovídá nějakému rozdílu prefixů, opačně to neplatí. Pokud $i > j + 1$, dostaneme $p_j - p_{i-1} = -(p_{i-1} - p_j) = -(a_{j+1} + \ldots + a_{i-1})$. To je „úsek naruby“, jehož součet má opačné znaménko než součet nějakého skutečného úseku. Zachrání nás však, že všechna zadaná čísla jsou kladná, takže úseky naruby mají záporné součty a ty se jistě nebudou rovnat S.

Úlohu jsme tedy převedli na případ podobný problému s krokodýly. Dostaneme posloupnost prefixových součtů a hledáme v ní dvě hodnoty s daným rozdílem. Navíc je tato posloupnost uspořádaná vzestupně (opět pomáhá, že a_i jsou nezáporná). A vskutku – algoritmus pro krokodýlí problém můžeme využít. Stačí položit $x_i = p_i$, $y_i = -p_{N-i+1}$, $D = S$ a dostaneme dvě neklesající posloupnosti, pro něž $x_i + y_j = D$ odpovídá našemu hledanému $p_i - p_{N-j+1} = S$, čili úseku $x_{N-j+2}, \ldots, x_i$ se součtem S.

Zopakujme si, co jsme udělali: Nejprve jsme pro zadanou posloupnost spočítali její prefixové součty. Pak jsme z nich vytvořili vstup pro krokodýlí problém. Ten jsme následně vyřešili a jeho výsledek nám ukázal, jak vypadá úsek s hledaným součtem. Všechny tyto kroky umíme provést v lineárním čase, takže celý algoritmus je také lineární.

Algoritmus ÚsekSeSoučtem

Vstup: Posloupnost $a_1, \ldots, a_N$ kladných čísel, číslo $S \geq 0$.

Výstup: Indexy i, j takové, že $a_i + a_{i+1} + \ldots + a_j = S$, nebo zpráva, že neexistují.

1. Spočítáme prefixové součty:
2. $\quad p_0 \leftarrow 0$
3. $\quad$ Pro $t = 1, \ldots, N$: $p_t \leftarrow p_{t-1} + a_t$
4. Převedeme na známý problém:
5. $\quad$ Pro $t = 0, \ldots, N$: $y_{t+1} \leftarrow p_{N-t}$
6. $\quad (k, \ell) \leftarrow$ Krokodýli$(p_1, \ldots, p_N; y_1, \ldots, y_{N+1}; S)$
7. $\quad$ Pokud (k, ℓ) není definováno, odpovíme, že řešení neexistuje.
8. $\quad$ Jinak vrátíme jako výsledek dvojici $(N - \ell + 2, k)$.

Co když povolíme záporná čísla?

Jak se předchozí příklad změní, připustíme-li, aby vstup obsahoval i záporná čísla? Jednak se již nemůžeme spolehnout na to, že je posloupnost prefixových součtů setříděná, jednak se budeme muset postarat o „úseky naruby“, protože nyní mohou mít součet rovný S. Myšlenku odečítání prefixů ale přesto můžeme využít.

Kdybychom na chvíli na úseky naruby zapomněli, stačilo by posloupnost prefixových součtů setřídit a pak v ní pro každé $i = 1, \ldots, N + 1$ binárně hledat p_j rovné $S + p_{i-1}$. Jak ale zařídit, aby bylo j větší nebo rovné i, když jsme tříděním o všechny informace o poloze prefixu přišli? Postačí místo hodnot p_j třídit dvojice (p_j, j) lexikograficky – tedy nejdříve podle hodnoty, pak podle pozice – a ze všech dvojic se stejnou hodnotou si zapamatovat tu nejpravější. Když pak najdeme dvojici (p_j, j), pro níž $p_j = S + p_{i-1}$, ověříme, zda je vpravo od p_{i-1}. Pokud není vpravo, žádná jiná dvojice s toutéž hodnotou také nebude.

Celý algoritmus bude vypadat následovně:

Algoritmus ÚsekSeSoučtemZ

Vstup: Posloupnost $a_1, \ldots, a_N$ celých čísel, číslo $S \geq 0$.

Výstup: Indexy i, j takové, že $a_i + a_{i+1} + \ldots + a_j = S$, nebo zpráva, že neexistují.

1. Spočítáme prefixové součty $p_0, \ldots, p_N$ jako v předchozím algoritmu.
2. Vytvoříme posloupnost $P \leftarrow (p_0, 0), \ldots, (p_N, N)$.
3. Setřídíme posloupnost P lexikograficky.
4. Ponecháme v posloupnosti P jen nejpravější výskyt každé hodnoty.
5. Pro $i = 0, \ldots, N$ opakujeme:
6. Binárně najdeme v P dvojici (p_j, j) s hodnotou $S + p_i$.
7. Pokud existuje a $j \geq i$, ohlásíme řešení $(i + 1, j)$ a skončíme.
8. Pokud jsme se dostali až sem, odpovíme, že hledaný úsek neexistuje.

Jak je toto řešení rychlé? První dva kroky stihneme lineárně, tříděním strávíme čas $\mathcal{O}(N \log N)$, probírání hodnot v následujícím kroku stihneme lineárně. Hlavním cyklem projdeme N-krát a pokaždé strávíme čas $\mathcal{O}(\log N)$ binárním vyhledáváním. Celkově má tedy náš algoritmus časovou složitost $\mathcal{O}(N \log N)$.

Cvičení:

1. Vyřešte úlohu o úseku s maximálním součtem pomocí prefixových součtů. Porovnejte své řešení s algoritmem MaxSoučet3.
2. Vymyslete algoritmus, který v posloupnosti celých čísel najde úsek se součtem co nejbližším danému číslu.
3. Navrhněte dvojrozměrnou analogii prefixových součtů: Pro matici $M \times N$ předpočítejte v čase $\mathcal{O}(MN)$ údaje, pomocí nichž půjde v konstantním čase vypočíst součet hodnot v libovolné souvislé obdélníkové podmatici.

4.* Jak by vypadaly k-rozměrné prefixové součty?

1.3. Největší prázdná podmatice

Ukažme si ještě jeden, trochu složitější příklad, ve kterém pomůže kombinace obou technik z této kapitoly.

V Duchovanech se rozhodli postavit novou jadernou elektrárnu. Pozemky za městem, kde by se dalo stavět, se rozkládají až k dalekému obzoru, leč je tu jistý zádrhel – na některých pozemcích straší duchové dávných majitelů. Jelikož bezpečnostní normy pro jaderné elektrárny přítomnost strašidla v objektu přísně zakazují, nezbývá než najít dostatečně velkou oblast prostou duchů.

Situaci popíšeme maticí A velikosti $M \times N$. Každé její políčko bude odpovídat pozemku 100 m × 100 m a bude obsahovat jedničku, pokud na pozemku straší, a jinak nulu. Naším úkolem bude najít v této matici *největší prázdnou podmatici,* tedy takovou, která obsahuje samé nuly. Velikost přitom měříme počtem políček podmatice.

Nabízí se vyzkoušet všechny možné podmatice a pro každou ověřit, zda je prázdná. Každá podmatice je jednoznačně určena polohou levého horního a pravého

dolního rohu, takže existuje celkem $\Theta(M^2N^2)$ podmatic. Jak v konstantním čase zjistit, zda je podmatice prázdná?

Zvolíme vždy pevně levý horní roh (x, y) a postupně budeme procházet všechny možné polohy (i, j) pravého dolního rohu, a to po řádcích shora dolů, v každém zleva doprava. Průběžně budeme upravovat číslo $S_{i,j}$ – součet prvků v podmatici (x, y)–(i, j). Pro každý řádek začneme maticí nulové šířky ($S_{i,j} = 0$ pro $j = y - 1$) a postupně ji rozšiřujeme doprava. Když zvýšíme j o 1, musíme přičíst všechna políčka v nově přibyvším sloupečku: $S_{i,j+1} = S_{i,j} + A_{x,j+1} + A_{x+1,j+1} + \ldots + A_{i,j+1}$. Ta dokážeme sečíst v konstantním čase, pokud si pro každý sloupeček předpočítáme prefixové součty. (Jinou možnost, jak sčítat podmatice, dává cvičení 1.2.3.)

Tak získáme následující algoritmus. Pro přehlednost počítáme pouze velikost nalezené matice, opět by bylo snadné algoritmus upravit, aby hlásil i její polohu.

Algoritmus PrázdnáPodmatice1

Vstup: Matice A tvaru $M \times N$.

Výstup: Velikost V největší prázdné podmatice.

1. Spočítáme všechny prefixové součty $X_{i,j} = A_{1,j} + A_{2,j} + \ldots + A_{i,j}$.
2. $V \leftarrow 0$ *(zatím maximální nalezená velikost)*
3. Pro $x = 1, \ldots, M$ a $y = 1, \ldots, N$ opakujeme: *(volíme levý horní roh)*
4. Pro $i = x, \ldots, M$ opakujeme: *(spodní řádek)*
5. $S \leftarrow 0$ *(součet hodnot v podmatici)*
6. Pro $j = y, \ldots, N$ opakujeme: *(pravý sloupec)*
7. $S \leftarrow S + X_{i,j} - X_{x-1,j}$
8. Pokud $S = 0$: $V \leftarrow \max(V, (i - x + 1) \cdot (j - y + 1))$

Jaké časové složitosti jsme dosáhli? Pro každou z MN poloh levého horního rohu vyzkoušíme všechny polohy pravého dolního rohu v čase $\Theta(MN)$. Celkem tedy spotřebujeme čas $\Theta(M^2N^2)$.

Přemýšlejme, která část algoritmu zdržuje nejvíce. Vhodným kandidátem je cyklus v kroku 6. Pokud v něm pro nějaké j zjistíme, že matice má nenulový součet, nemá již smysl j zvyšovat – všechny širší podmatice také nebudou prázdné – a rovnou můžeme přejít na další i. Navíc pokud podmatice (x, y)–(i, j) nebyla prázdná, podmatice (x, y)–$(i + 1, j)$ taktéž nebude. Proto stačí při zvýšení i uvažovat pouze snižování j.

Tento trik už známe z úlohy s krokodýly. Vede na algoritmus se složitostí $\mathcal{O}(M^2N)$, jíž samozřejmě můžeme zlepšit na $\mathcal{O}(MN \cdot \min(M, N))$, pokud v případě, že M je větší než N, prohodíme řádky a sloupce.

Ke stejné složitosti ale můžeme dospět i přímočařeji. Předpočítáme si hodnoty $P_{i,j}$, které budou říkat, kolik políček od políčka (i, j) doprava je prázdných. To zvládneme pro každý řádek lineárním průchodem zprava doleva. Pak budeme volit všechny polohy levého horního rohu (x, y) a všechny možné spodní řádky i. Pro

každý z nich posléze spočítáme maximální šířku s_i, pro kterou je podmatice (x, y)–$(i, y + s_i - 1)$ ještě prázdná.

Tyto šířky můžeme počítat inkrementálně: pokud přejdeme od $i - 1$ k i, pak nejširší prázdná podmatice buďto zůstane stejná (pokud od políčka (i, x) doprava leží alespoň s_{i-1} nul), nebo se zúží (pokud je nul méně). S pomocí předpočítaných hodnot tak dojdeme ke vztahu $s_i = \min(s_{i-1}, P_{i,y})$. Hotový algoritmus můžeme zapsat například takto:

Algoritmus PrázdnáPodmatice2

Vstup: Matice A tvaru $M \times N$.

Výstup: Velikost V největší prázdné podmatice.

1. Pokud by bylo $M > N$, prohodíme řádky a sloupce a také M a N.
2. Předpočítáme počty $P_{i,j}$ prázdných políček vpravo od (i, j):
3. Pro $i = 1, \ldots, M$ opakujeme:
4. $P_{i,N+1} = 0$
5. Pro $j = N, \ldots, 1$ opakujeme:
6. Pokud $A_{i,j} = 1$, položíme $P_{i,j} \leftarrow 0$.
7. Jinak $P_{i,j} \leftarrow P_{i,j+1} + 1$.
8. $V \leftarrow 0$ *(zatím maximální nalezená velikost)*
9. Pro $x = 1, \ldots, M$ a $y = 1, \ldots, N$ opakujeme: *(levý horní roh)*
10. $s \leftarrow N$ *(zatím maximální šířka)*
11. Pro $i = x, \ldots, M$ opakujeme: *(spodní řádek)*
12. $s \leftarrow \min(j, P_{i,y})$
13. $V \leftarrow \max(V, (i - x + 1) \cdot s)$

Zkontrolujeme časovou složitost: Zkoušíme $\mathcal{O}(MN)$ levých horních rohů, pro každý z nich $\mathcal{O}(N)$ dolních řádků a vždy v čase $\mathcal{O}(1)$ aktualizujeme j_i. Celkem tedy $\mathcal{O}(M^2N)$, respektive $\mathcal{O}(MN \cdot \min(M, N))$.

Ani zde se ještě na naší pouti za optimálním řešením nezastavíme. Uvažujme takto: hledaná maximální díra určitě svým horním okrajem přiléhá k nějaké jedničce (nebo k hornímu okraji, tak si pomůžeme orámováním matice jedničkami). Budeme tedy namísto levých horních rohů vybírat všechny jedničky a pro každou z nich hledat největší prázdnou podmatici, která k této jedničce přiléhá.

Postupně vyzkoušíme všechny možné výšky podmatice a budeme udržovat maximální šířku podmatice podobně jako v minulém algoritmu, ovšem zvlášť doleva a doprava od *osy* podmatice, tedy od sloupce, v němž leží zvolená jednička. Obojí dokážeme po přidání řádku aktualizovat v konstantním čase – stačí, když si předpočítáme jak počet nul vpravo od políčka, tak vlevo od něj.

Algoritmus následuje. Přidali jsme do něj ještě jeden drobný trik: pokud při prohlubování matice narazíme na jedničku ležící na ose, zastavíme se; další šířky už tak jako tak budou vycházet nulové.

Algoritmus PrázdnáPodmatice3

Vstup: Matice A tvaru $M \times N$.

Výstup: Velikost V největší prázdné podmatice.

1. Doplníme do matice A řádky číslo 0 a $M+1$ plné jedniček.
2. Předpočítáme počty $P_{i,j}$ prázdných políček vpravo od (i,j).
3. Předpočítáme počty $L_{i,j}$ prázdných políček vlevo od (i,j).
4. $V \leftarrow 0$ *(zatím maximální nalezená velikost)*
5. Pro $x = 0, \ldots, M-1$ a $y = 1, \ldots, N$ opakujeme: *(pozice jedničky)*
6. Pokud $A_{x,y} = 1$:
7. $\ell \leftarrow N, p \leftarrow N$ *(šířka podmatice doleva a doprava od osy)*
8. $i \leftarrow x+1$ *(spodní řádek)*
9. Dokud $A_{i,y} = 0$, opakujeme:
10. $\ell \leftarrow \min(\ell, L_{i,y})$
11. $p \leftarrow \min(p, P_{i,y})$
12. $V \leftarrow \max(V, (i-x+1) \cdot (\ell+p-1))$
13. $i \leftarrow i+1$

Zbývá rozebrat časovou složitost. Zkoušíme $\mathcal{O}(MN)$ políček s jedničkami, pro každé z nich $\mathcal{O}(M)$ výšek a jednu výšku otestujeme v konstantním čase. To opět vede na $\mathcal{O}(M^2N)$ – tak kde je slibované zlepšení? Pomůže zaražení se o jedničku na ose v kroku 9. Díky němu totiž pro každou jedničku vyzkoušíme přesně tolik výšek, kolik leží nul bezprostředně pod touto jedničkou. To je dohromady za celou dobu běhu algoritmu rovné celkovému počtu nul v matici, čili $\mathcal{O}(MN)$. A to je také časová složitost celého algoritmu.

Kombinací technik předvýpočtu a inkrementálního počítáni jsme tedy dokázali časovou složitost hledání největší prázdné podmatice postupně snížit až na lineární s plochou matice, což je nejlepší možné (viz cvičení 1.3.2).

Cvičení:

1. Navrhněte, jak algoritmus MaxSoučet3 upravit, aby si vystačil pouze s lineárním množstvím paměti, nepočítáme-li vstupní matici.
2. Dokažte, že algoritmus MaxSoučet3 je optimální. Nalezněte pro každé M, N matici, v níž je $\Omega(MN)$ políček takových, že změnou kteréhokoliv z nich se změní odpověď algoritmu. Algoritmus proto musí všechna tato políčka přečíst.
3. Co kdyby duchovanští radní věděli, jak velký pozemek potřebují, a hledali proto v matici všechny prázdné podmatice předem určené velikosti?
4. … co kdyby znali pouze jeden z rozměrů podmatice?
5. Jak nalézt největší *čtvercovou* podmatici ze samých nul?
6. Co kdybychom místo největší prázdné podmatice hledali podmatici s maximálním součtem? Zadané velikosti? Čtvercovou?

7. Je dána ostře rostoucí celočíselná posloupnost $x_1, \ldots, x_N$. Nalezněte index i takový, že $x_i = i$.
8. Je dána matice celočíselná matice A tvaru $N \times N$, ostře rostoucí v každém řádku i sloupci. Nalezněte alespoň jednu dvojici indexů (i, j), pro které platí $A_{i,j} = i + j$. Podobně jako v cvičení 1.3.2 dokažte, že vaše řešení je nejrychlejší možné.
9. Upravte algoritmus z předchozího cvičení, aby nahlásil všechny dvojice (i, j) splňující podmínku. S vhodně zvoleným formátem výstupu se nezhorší časová složitost.
10. * Dokažte, že podmínka celočíselnosti je v cvičeních 1.3.7 a 1.3.8 zásadní – v opačném případě úlohu nelze řešit rychleji než lineárně s velikostí vstupu.