1. Dynamické programování

V této kapitole prozkoumáme ještě jednu techniku návrhu algoritmů, která je založená na rekurzivním rozkladu problému na podproblémy. V tom je podobná metodě Rozděl a panuj, ovšem umí využít toho, že se podproblémy během rekurze opakují. Proto v mnoha případech vede na mnohem rychlejší algoritmy. Říká se jí poněkud tajemně *dynamické programování.*⟨1⟩

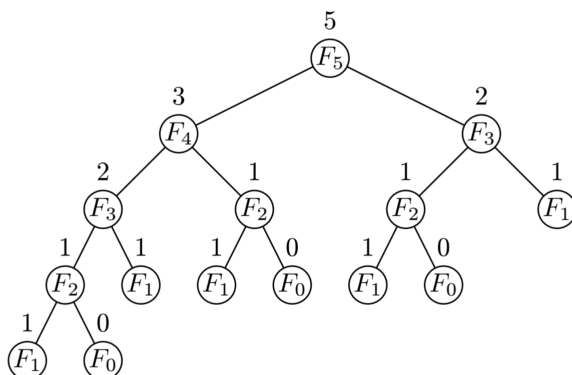
1.1. Fibonacciho čísla podruhé

Princip dynamického programování si nejprve vyzkoušíme na triviálním příkladu. Budeme počítat Fibonacciho čísla, se kterými jsme se už potkali v úvodní kapitole. Začneme přímočarým rekurzivním algoritmem na výpočet n-tého Fibonacciho čísla F_n, který postupuje přesně podle definice $F_n = F_{n-1} + F_{n-2}$.

Algoritmus Fib(n)

1. Pokud $n \leq 1$, vrátíme n.
2. Jinak vrátíme Fib($n-1$) + Fib($n-2$).

Zkusme zjistit, jakou časovou složitost tento algoritmus má. Sledujme strom rekurze na následujícím obrázku. V jeho kořeni počítáme F_n, v listech F_0 a F_1, vnitřní vrcholy odpovídají výpočtům čísel F_k pro $2 \leq k < n$.



Obr. 1.1: Rekurzivní výpočet Fibonacciho čísel

⟨1⟩ Legenda říká, že s tímto názvem přišel Richard Bellman, když v 50. letech pracoval v americkém armádním výzkumu a potřeboval nadřízeným vysvětlit, čím se vlastně zabývá. Programováním se tehdy mínilo zejména plánování (třeba postupu výroby) a Bellman zkoumal vícekrokové plánování, v němž optimální volba každého kroku záleží na předchozích krocích – proto dynamické.

Libovolný vnitřní vrchol přitom vrací součet hodnot ze svých synů. Pokud tento argument zopakujeme, dostaneme, že hodnota vnitřního vrcholu je rovna součtu hodnot všech listů ležících pod ním. Speciálně tedy F_n v kořeni musí být rovno součtu všech listů. Z každého listu přitom vracíme buďto 0 nebo 1, takže abychom nasčítali F_n, musí se ve stromu celkově nacházet alespoň F_n listů. Z cvičení **??** víme, že $F_n \approx 1.618^n$, takže strom rekurze má přinejmenším exponenciálně mnoho listů a celý algoritmus se plouží exponenciálně pomalu.

Nyní si všimněme, že funkci Fib voláme pouze pro argumenty z rozsahu 0 až n. Jediné možné vysvětlení exponenciální časové složitosti tedy je, že si necháváme mnohokrát spočítat totéž. To je vidět i na obrázku: F_2 vyhodnocujeme dvakrát a F_1 dokonce čtyřikrát.

Zkusme tomu zabránit. Pořídíme si tabulku T a budeme do ní vyplňovat, která Fibonacciho čísla jsme už spočítali a jak vyšly jejich hodnoty. Při každém volání rekurzivní funkce se pak podíváme do tabulky. Pokud již výsledek známe, rovnou ho vrátíme; v opačném případě ho poctivě spočítáme a hned uložíme do tabulky. Upravený algoritmus bude vypadat následovně.

Algoritmus Fib2(n)

1. Je-li $T[n]$ definováno, vrátíme $T[n]$.
2. Pokud $n \leq 1$, položíme $T[n] \leftarrow n$.
3. Jinak položíme $T[n] \leftarrow$ Fib2$(n-1)$ + Fib2$(n-2)$.
4. Vrátíme $T[n]$.

Jak se změnila časová složitost? K rekurzi nyní dojde jedině tehdy, vyplňujeme-li políčko tabulky, v němž dosud nic nebylo. To se může stát nejvýše $(n+1)$-krát, z toho dvakrát triviálně (pro F_0 a F_1), takže strom rekurze má nejvýše n vnitřních vrcholů. Pod každým z nich leží nejvýše 2 listy, takže celkově má strom nanejvýš $3n$ vrcholů. V každém z nich trávíme konstantní čas, celkově běží funkce Fib2 v čase $\mathcal{O}(n)$.

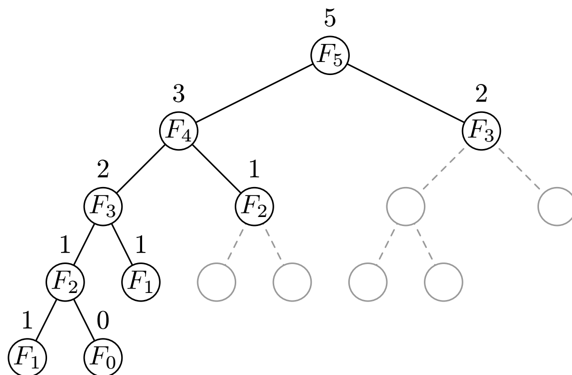
Ve stromu rekurze jsme tedy prořezali opakující se větve, až zbylo $\mathcal{O}(n)$ vrcholů. Jak to dopadne pro F_5, vidíme na obrázku 1.2.

Nakonec si uvědomíme, že tabulku mezivýsledků T nemusíme vyplňovat rekurzivně. Jelikož k výpočtu $T[k]$ potřebujeme pouze $T[k-1]$ a $T[k-2]$, stačí ji plnit v pořadí $T[0], T[1], T[2], \ldots$ a vždy budeme mít k dispozici všechny hodnoty, které v daném okamžiku potřebujeme. Dostaneme následující nerekurzivní algoritmus.

Algoritmus Fib3(n)

1. $T[0] \leftarrow 0$, $T[1] \leftarrow 1$
2. Pro $k = 2, \ldots, n$:
3. $\quad T[k] \leftarrow T[k-1] + T[k-2]$
4. Vrátíme $T[n]$.

Funkci Fib3 jsme pochopitelně mohli vymyslet přímo, bez úvah o rekurzi. Postup, který jsme si předvedli, ovšem funguje i v méně přímočarých případech. Zkusme proto shrnout, co jsme udělali.



Obr. 1.2: Prořezaný strom rekurze po zavedení tabulky

Princip dynamického programování:

- Začneme s rekurzivním algoritmem, který je exponenciálně pomalý.
- Odhalíme opakované výpočty stejných podproblémů.
- Pořídíme si tabulku a budeme si v ní pamatovat, které podproblémy jsme už vyřešili. Tím prořežeme strom rekurze a vznikne rychlejší algoritmus. Tomuto přístupu se často říká *kešování* a tabulce *keš* (anglicky cache).⟨2⟩
- Uvědomíme si, že keš lze vyplňovat bez rekurze, zvolíme-li vhodné pořadí podproblémů. Tím získáme stejně rychlý, ale jednodušší algoritmus.

Cvičení

1. Spočítejte, kolik přesně vrcholů má strom rekurze funkce Fib a dokažte, že časová složitost této funkce činí $\Theta(\tau^n)$, kde $\tau = (1+\sqrt{5})/2$ je zlatý řez.

1.2. Vybrané podposloupnosti

Metodu dynamického programování nyní předvedeme na méně triviálním příkladu. Dostaneme posloupnost $x_1, \ldots, x_n$ celých čísel a chceme z ní škrtnout co nejméně prvků tak, aby zbývající prvky tvořily rostoucí posloupnost. Jinak řečeno, chceme najít *nejdelší rostoucí podposloupnost (NRP).* Tu můžeme formálně popsat

⟨2⟩ Cache v angličtině znamená obecně skrýš, třeba tu, kam si veverka schovává oříšky. V informatice se tak říká různým druhům paměti na často používaná data. Je-li řeč o zrychlování rekurze, používá se též poněkud krkolomný termín *memoizace* – *memo* je zkrácenina z latinského *memorandum* a dnes značí libovolnou poznámku.

jako co nejdelší posloupnost indexů $i_1, \ldots, i_k$ takovou, že $1 \leq i_1 < \ldots < i_k \leq n$ a $x_{i_1} < \ldots < x_{i_k}$.

Například v následující posloupnosti je jedna z NRP vyznačena tučně.

i	1	2	3	4	5	6	7	8	9	10	11	12	13
x_i	**3**	**14**	**15**	92	65	**35**	89	79	32	**38**	46	26	**43**

Rekurzivní řešení

Nabízí se použít hladový algoritmus: začneme prvním prvkem posloupnosti a pokaždé budeme přidávat nejbližší další prvek, který je větší. Pro naši ukázkovou posloupnost bychom tedy začali $3, 14, 15, 92$ a dál bychom už nemohli přidat žádný další prvek. Optimální řešení je ale delší.

Problém byl v tom, že jsme z možných pokračování podposloupnosti (tedy čísel větších než poslední přidané a ležících napravo od něj) zvolili hladově to nejbližší. Pokud místo toho budeme zkoušet všechna možná pokračování, dostaneme rekurzivní algoritmus, který bude korektní, byť pomalý.

Jeho jádrem bude rekurzivní funkce $\textsc{Nrp}(i)$. Ta pro dané i spočítá maximální délku rostoucí podposloupnosti začínající prvkem x_i. Udělá to tak, že vyzkouší všechna x_j navazující na x_i (tedy $j > i$ a $x_j > x_i$) a pro každé z nich se zavolá rekurzivně. Z možných pokračování si pak vybere to, které dá celkově nejlepší výsledek.

Algoritmus $\textsc{Nrp}(i)$ (nejdelší rostoucí podposloupnost rekurzivně)

Vstup: Posloupnost $x_i, \ldots, x_n$

1. $d \leftarrow 1$
2. Pro $j = i + 1, \ldots, n$:
3. Je-li $x_j > x_i$:
4. $d \leftarrow \max(d, 1 + \textsc{Nrp}(j))$

Výstup: Délka d nejdelší rostoucí podposloupnosti v $x_i, \ldots, x_n$

Všimněme si, že rekurze se přirozeně zastaví pro $i = n$: tehdy totiž cyklus neproběhne ani jednou a funkce se ihned vrátí s výsledkem 1.

Řešení původní úlohy získáme tak, že zavoláme $\textsc{Nrp}(i)$ postupně pro $i = 1, \ldots, n$ a vypočteme maximum z výsledků. Probereme tedy všechny možnosti, kterým prvkem může optimální řešení začínat. Elegantnější ovšem je dodefinovat $x_0 = -\infty$. Tím získáme prvek, který se v optimálním řešení zaručeně vyskytuje, takže postačí zavolat $\textsc{Nrp}(0)$.

Tento algoritmus je korektní, nicméně má exponenciální časovou složitost: pokud je vstupní posloupnost sama o sobě rostoucí, projdeme během výpočtu úplně všechny podposloupnosti a těch je 2^n. Pro každý prvek si totiž nezávisle na ostatních můžeme vybrat, zda v podposloupnosti leží.

Podobně jako u příkladu s Fibonacciho čísly nás zachrání, když si budeme pamatovat, co jsme už spočítali, a nebudeme to počítat znovu. Funkci $\textsc{Nrp}$ totiž

můžeme zavolat pouze pro $n + 1$ různých argumentů. Pokaždé v ní strávíme čas $\mathcal{O}(n)$, takže celý algoritmus poběží v příjemném čase $\mathcal{O}(n^2)$.

Iterativní řešení

Sledujme dále osvědčený postup. Rekurze se můžeme zbavit a tabulku vyplňovat postupně od největšího i k nejmenšímu. Budeme tedy počítat $T[i]$, což bude délka té nejdelší ze všech rostoucích podposloupností začínajících prvkem x_i.

Algoritmus NRP2 (nejdelší rostoucí podposloupnost iterativně)
Vstup: Posloupnost $x_1, \ldots, x_n$

1. $x_0 \leftarrow -\infty$
2. Pro $i = n, n-1, \ldots, 0$: $\triangleleft$ *všechny možné začátky NRP*
3. $\quad T[i] \leftarrow 1$
4. $\quad P[i] \leftarrow 0$ $\triangleleft$ *bude se později hodit pro výpis řešení*
5. $\quad$ Pro $j = i+1, \ldots, n$: $\triangleleft$ *všechna možná pokračování*
6. $\qquad$ Pokud $x_i < x_j$ a $T[i] < 1 + T[j]$: $\triangleleft$ *máme lepší řešení*
7. $\qquad\quad T[i] \leftarrow 1 + T[j]$
8. $\qquad\quad P[i] \leftarrow j$

Výstup: Délka $T[0]$ nejdelší rostoucí podposloupnosti

Tento algoritmus běží také v kvadratickém čase. Jeho průběh na naší ukázkové posloupnosti ilustruje následující tabulka. (Všimněte si, že algoritmus našel jiné optimální řešení, než jakého jsme si prve všimli my.)

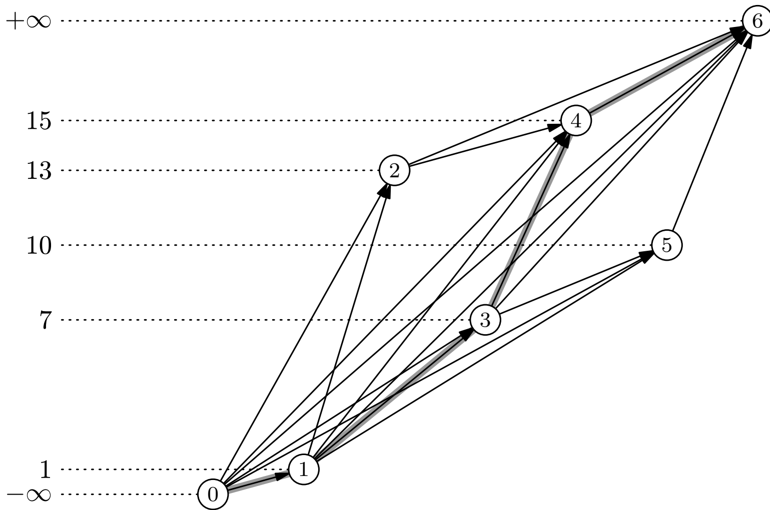
i	0	1	2	3	4	5	6	7	8	9	10	11	12	13
x_i	$-\infty$	**3**	**14**	**15**	92	65	**35**	89	79	32	**38**	**46**	26	43
$T[i]$	7	6	5	4	1	2	3	1	1	3	2	1	2	1
$P[i]$	1	2	3	6	0	7	10	0	0	10	11	0	13	0

Korektnost algoritmu můžeme dokázat zpětnou indukcí podle i. K tomu se nám hodí nahlédnout, že začíná-li optimální řešení pro vstup $x_i, \ldots, x_n$ dvojicí x_i, x_j, pak z něj odebráním x_i vznikne optimální řešení pro kratší vstup $x_j, \ldots, x_n$ začínající x_j. Kdyby totiž existovalo lepší řešení pro kratší vstup, mohli bychom ho rozšířit o x_i a získat lepší řešení pro původní vstup. Této vlastnosti se říká *optimální substruktura* a už jsme ji potkali například u nejkratších cest v grafech.

Zbývá domyslet, jak kromě délky NRP nalézt i posloupnost samu. K tomu nám pomůže, že kdykoliv jsme spočítali $T[i]$, uložili jsme do $P[i]$ index druhého prvku příslušné optimální podposloupnosti (prvním prvkem je vždy x_i). Proto $P[0]$ říká, jaký prvek je v optimálním řešení celé úlohy první, $P[P[0]]$ udává druhý a tak dále. Opět to funguje analogicky s hledáním nejkratší cesty třeba prohledáváním do šířky: tam jsme si pamatovali předchůdce každého vrcholu a pak zpětným průchodem rekonstruovali cestu.

Grafový pohled

Podobnost s hledáním cest v grafech není náhodná – celou naši úlohu totiž můžeme přeformulovat do řeči grafů. Sestrojíme orientovaný graf, jehož vrcholy budou prvky $x_0, \ldots, x_{n+1}$, přičemž dodefinujeme $x_0 = -\infty$ a $x_{n+1} = +\infty$. Hrana povede z x_i do x_j tehdy, mohou-li x_i a x_j sousedit v rostoucí podposloupnosti, čili pokud $i < j$ a současně $x_i < x_j$. (Na obrázku tyto hrany vedou „doprava nahoru“.)



Obr. 1.3: Graf reprezentující posloupnost $-\infty, 1, 13, 7, 15, 10, +\infty$ a jedna z nejdelších cest

Každá rostoucí podposloupnost pak odpovídá nějaké cestě v tomto grafu. Chceme proto nalézt nejdelší cestu. Ta bez újmy na obecnosti začíná v x_0 a končí v x_{n+1}.

Náš graf má $\Theta(n)$ vrcholů a $\mathcal{O}(n^2)$ hran. Navíc je acyklický a a pořadí vrcholů $x_0, \ldots, x_{n+1}$ je topologické. Nejdelší cestu tedy můžeme najít v čase $\mathcal{O}(n^2)$ indukcí podle topologického uspořádání (podrobněji viz oddíl **??**).

Výsledný algoritmus je přitom náramně podobný našemu Nrp2: vnější cyklus prochází pozpátku topologickým pořadím, vnitřní cyklus zkoumá hrany z vrcholu x_i a $T[i]$ můžeme interpretovat jako délku nejdelší cesty z x_i do x_{n+1}. To je poměrně typické: dynamické programování je často ekvivalentní s hledáním cesty ve vhodném grafu. Někdy je jednodušší nalézt tento graf, jindy zase k algoritmu dojít „převrácením“ rekurze.

Rychlejší algoritmus

Kvadratické řešení je jistě lepší než exponenciální, ale můžeme ho ještě zrychlit. Výběr maxima hrubou silou totiž můžeme nahradit použitím šikovné datové struktury. Ta si bude pro všechna zpracovaná x_i pamatovat dvojice $(x_i, T[i])$, přičemž x_i slouží jako klíč a $T[i]$ jako hodnota přiřazená tomuto klíči.

Algoritmus Nrp2 v každém průchodu vnějšího cyklu uvažuje jedno x_i. Výpočet vnitřního cyklu odpovídá tomu, že v datové struktuře hledáme největší z hodnot přiřazených klíčům z intervalu $(x_i, +\infty)$. Následně vložíme novou dvojici s klíčem x_i. Jednoduchá modifikace vyvážených vyhledávacích stromů (cvičení **??**) zvládne obě tyto operace v čase $\Theta(\log n)$. (Technický detail: Naše klíče se mohou opakovat. Tehdy stačí zapamatovat si největší z hodnot.)

Jeden průchod vnějšího cyklu pak zvládneme v čase $\Theta(\log n)$, takže celý algoritmus poběží v $\Theta(n \log n)$. Jiný stejně rychlý algoritmus odvodíme ve cvičení 4.

Cvičení

1. *Kopcem* nazveme podposloupnost, která nejprve roste a pak klesá. Vymyslete algoritmus, který v zadané posloupnosti nalezne nejdelší kopec.
2. NRP nemusí být jednoznačně určena. Jak spočítat, kolik různých NRP obsahuje zadaná posloupnost?
3. Pokud existuje více NRP, jakou význačnou vlastnost má ta, kterou najde algoritmus Nrp2?
4. Prozkoumejme jiný přístup ke hledání nejdelší rostoucí podposloupnosti. Zadanou posloupnost budeme procházet zleva doprava. Pro již zpracovanou část si budeme udržovat čísla $K[i]$ udávající, jakou nejmenší hodnotou může končit rostoucí podposloupnost délky i. Nahlédněte, že $K[i] < K[i+1]$. Ukažte, že rozšíříme-li vstup o další prvek x, změní se $\mathcal{O}(1)$ hodnot $K[i]$ a k jejich nalezení stačí nalézt binárním vyhledáváním, kam do posloupnosti K patří x. Z toho získejte algoritmus o složitosti $\Theta(n \log n)$.
5. Mějme posloupnost n knih. Každá kniha má nějakou šířku s_i a výšku v_i. Knihy chceme naskládat do knihovny s nějakým počtem polic tak, abychom dodrželi abecední pořadí. Prvních několik knih tedy půjde na první polici, další část na druhou polici, a tak dále. Máme zadanou šířku knihovny S a chceme rozmístit police tak, aby se do nich vešly všechny knihy a celkově byla knihovna co nejnižší. Tloušťku polic a horní a spodní desky přitom zanedbáváme.
6. Podobně jako v předchozím cvičení chceme navrhnout knihovnu, jež pojme dané knihy. Tentokrát ovšem máme zadanou maximální výšku knihovny a chceme najít minimální možnou šířku. Pokud vám to pomůže, předpokládejte, že všechny knihy mají jednotkovou šířku.
7. Dešifrovali jsme tajnou depeši, ale chybí v ní mezery. Známe však slovník všech slov, která se v depeši mohou vyskytnout. Chceme tedy rozdělit depeši na co nejméně slov ze slovníku.
8. Grafový pohled na dynamické programování funguje i pro Fibonacciho čísla. Ukažte, jak pro dané n sestrojit graf na $\mathcal{O}(n)$ vrcholech, v němž bude existovat právě F_n cest ze startu do cíle. Jak tento graf souvisí se stromem rekurze algoritmu Fib?

1.3. Editační vzdálenost

Pokud ve slově `koule` uděláme překlep, vznikne `boule`, nebo třeba `kdoule`. Kolik překlepů je potřeba, aby z `poutníka` vznikl `potemník`? Podobné otázky vedou ke zkoumání editační vzdálenosti řetězců nebo obecně posloupností.

Definice: *Editační operací* na řetězci nazveme vložení, smazání nebo změnu jednoho znaku. *Editační vzdálenost*⟨3⟩ řetězců $x = x_1 \ldots x_n$ a $y = y_1 \ldots y_m$ udává, kolik nejméně editačních operací je potřeba, abychom z prvního řetězce vytvořili druhý. Budeme ji značit $L(x, y)$.

V nejkratší posloupnosti operací se každého znaku týká nejvýše jedna editační operace, takže operace lze vždy uspořádat „zleva doprava". Můžeme si tedy představit, že procházíme řetězcem x od začátku do konce a postupně ho přetváříme na řetězec y.

Rekurzivní řešení

Zkusme rozlišit případy podle toho, jaká operace nastane v optimální posloupnosti na samém začátku řetězce:

- Pokud $x_1 = y_1$, můžeme první znak ponechat beze změny. Tehdy $L(x, y) = L(x_2 \ldots x_n, y_2 \ldots y_m)$.
- Znak x_1 změníme na y_1. Pak $L(x, y) = 1 + L(x_2 \ldots x_n, y_2 \ldots y_m)$.
- Znak x_1 smažeme. Tehdy $L(x, y) = 1 + L(x_2 \ldots x_n, y_1 \ldots y_m)$.
- Na začátek vložíme y_1. Tehdy $L(x, y) = 1 + L(x_1 \ldots x_n, y_2 \ldots y_m)$.

Pokaždé tedy $L(x, y)$ závisí na vzdálenosti nějakých *suffixů* řetězců x a y. Kdybychom tyto vzdálenosti znali, mohli bychom snadno rozpoznat, která z uvedených čtyř možností nastala – byla by to ta, z níž vyjde nejmenší $L(x, y)$.

Pokud vzdálenosti suffixů neznáme, vypočítáme je rekurzivně. Zastavíme se v případech, kdy už je jeden z řetězců prázdný – tehdy je evidentně vzdálenost rovna délce druhého řetězce.

Z toho vychází následující algoritmus. Pro výpočet $L(x, y)$ postačí zavolat $\textsc{Edit}(1, 1)$.

Algoritmus $\textsc{Edit}(i, j)$ (editační vzdálenost řetězců rekurzivně)
Vstup: Řetězce $x_i \ldots x_n$ a $y_j \ldots y_m$

1. Pokud $i > n$, vrátíme $m - j + 1$. ◁ *jeden z řetězců už skončil*
2. Pokud $j > m$, vrátíme $n - i + 1$.
3. $\ell_z \leftarrow \textsc{Edit}(i + 1, j + 1)$ ◁ *ponechání či změna znaku*
4. Pokud $x_i \neq y_j$: $\ell_z \leftarrow \ell_z + 1$.
5. $\ell_s \leftarrow \textsc{Edit}(i + 1, j)$ ◁ *smazání znaku*
6. $\ell_v \leftarrow \textsc{Edit}(i, j + 1)$ ◁ *vložení znaku*

⟨3⟩ Někdy též *Levenštejnova vzdálenost* podle Vladimira Josifoviče Levenštejna, který ji zkoumal okolo roku 1965. Odtud značení $L(x, y)$.

7. Vrátíme $\min(\ell_z, \ell_s, \ell_v)$

Výstup: Editační vzdálenost $L(x_i \ldots x_n, y_j \ldots y_m)$

Algoritmus je zjevně korektní, nicméně může běžet exponenciálně dlouho (třeba pro $x = y =$ `aaa`...`a`). Opět nás zachrání, že funkci Edit můžeme zavolat jen s $(n+1)(m+1)$ různými argumenty. Budeme si tedy kešovat, pro které argumenty už známe výsledek, a známé hodnoty nebudeme počítat znovu. Funkce pak poběží jen $\mathcal{O}(nm)$-krát a pokaždé spotřebuje konstantní čas.

Iterativní řešení

Pokračujme podobně jako v minulém oddílu. Otočíme směr výpočtu a tabulku T s výsledky podproblémů budeme vyplňovat bez použití rekurze. Představíme-li si ji jako matici, každý prvek závisí pouze na těch, které leží napravo a dolů od něj. Tabulku proto můžeme vyplňovat po řádcích zdola nahoru, zprava doleva.

Tím získáme následující jednodušší algoritmus, který zjevně běží v čase $\Theta(nm)$. Příklad výpočtu naleznete na obrázku 1.4.

Algoritmus Edit2 (editační vzdálenost řetězců iterativně)

Vstup: Řetězce $x_1 \ldots x_n$ a $y_1 \ldots y_m$

1. Pro $i = 1, \ldots, n+1$ položíme $T[i, m+1] \leftarrow n - i + 1$.
2. Pro $j = 1, \ldots, m+1$ položíme $T[n+1, j] \leftarrow m - j + 1$.
3. Pro $i = n, \ldots, 1$:
4. Pro $j = m, \ldots, 1$:
5. Je-li $x_i = y_j$: $\delta \leftarrow 0$, jinak $\delta \leftarrow 1$
6. $T[i,j] \leftarrow \min(\delta + T[i+1, j+1], 1 + T[i+1, j], 1 + T[i, j+1])$

Výstup: Editační vzdálenost $L(x_1 \ldots x_n, y_1 \ldots y_m) = T[1, 1]$

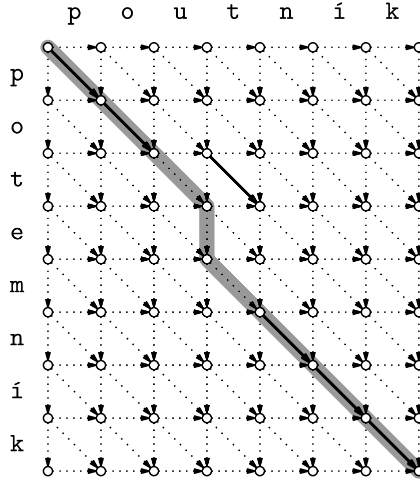
	p	o	t	e	m	n	í	k	
p	3	4	4	4	4	4	5	6	7
o	4	3	3	3	3	3	4	5	6
u	4	3	3	2	2	2	3	4	5
t	4	3	2	2	1	1	2	3	4
n	5	4	3	2	1	0	1	2	3
í	6	5	4	3	2	1	0	1	2
k	7	6	5	4	3	2	1	0	1
	8	7	6	5	4	3	2	1	0

Obr. 1.4: Tabulka T pro slova `poutník` a `potemník`

Grafové řešení

Editační vzdálenost můžeme také popsat pomocí vhodného orientovaného grafu (obrázek 1.5). Vrcholy budou odpovídat možným pozicím v obou řetězcích. Budou to tedy dvojice (i, j), kde $1 \leq i \leq n+1$ a $1 \leq j \leq m+1$. Hrany budou popisovat

možné operace: z vrcholu (i, j) povede hrana do $(i+1, j)$, $(i, j+1)$ a $(i+1, j+1)$. Tyto hrany odpovídají po řadě smazání znaku, vložení znaku a ponechání/záměně znaku. Všechny budou mít jednotkovou délku, pouze v případě ponechání nezměněného písmene $(x_i = y_j)$ bude délka nulová.



Obr. 1.5: Graf k výpočtu editační vzdálenosti.
Plné hrany mají délku 0, čárkované 1.

Každá cesta z vrcholu $(1, 1)$ do $(n+1, m+1)$ proto odpovídá jedné posloupnosti operací uspořádané zleva doprava, která z řetězce x vyrobí y. Jelikož graf je acyklický a má $\Theta(nm)$ vrcholů a $\Theta(nm)$ hran, můžeme v něm nalézt nejkratší cestu indukcí podle topologického uspořádání v čase $\Theta(nm)$.

Přesně to ostatně dělá náš algoritmus Edit2. Indukcí můžeme dokázat, že $T[i, j]$ je rovno délce nejkratší cesty z vrcholu (i, j) do $(n + 1, m + 1)$.

Cvičení

1. Dokažte, že editační vzdálenost $L(x, y)$ se chová jako *metrika:* je vždy nezáporná, nulová pouze pro $x = y$, symetrická $(L(x, y) = L(y, x))$ a splňuje trojúhelníkovou nerovnost $L(x, z) \le L(x, y) + L(y, z)$.
2. Algoritmus Edit2 zabere $\Theta(nm)$ buněk paměti na uložení tabulky T. Ukažte, jak spotřebu paměti snížit na $\Theta(n + m)$.
3. Kromě editační vzdálenosti můžeme chtít spočítat i příslušnou nejkratší posloupnost editačních operací. V grafové interpretaci našeho algoritmu je to triviální – prostě vypíšeme nalezenou nejkratší cestu. Ukažte, jak to udělat bez explicitního sestrojení grafu, třeba přímou úpravou algoritmu Edit2.

4.* I v předchozím cvičení si lze vystačit s pamětí $\Theta(n + m)$. Existuje pěkný trik založený na metodě Rozděl a panuj. Budeme hledat nejkratší cestu v grafu

z obrázku 1.5. Graf postačí procházet po řádcích (to je topologické pořadí), ale obvyklé grafové algoritmy by si pro vypisování cesty musely zapamatovat předchůdce každého vrcholu. My si místo toho uložíme jen to, ve kterém sloupci protľala nejkratší cesta z počátku do daného vrcholu $(n/2)$-tý řádek. To si postačí pamatovat jen v okolí aktuálního řádku. Na konci výpočtu zjistíme, jaký je prostřední vrchol na nejkratší cestě, takže umíme problém rozdělit na dva poloviční podproblémy. Doplňte detaily a dokažte, že tomuto algoritmu postačí lineární paměť, a přesto celkově poběží v čase $\Theta(nm)$.

5. Na první pohled se zdá, že čím podobnější řetězce dostaneme, tím by mělo být jednodušší zjistit jejich editační vzdálenost. Náš algoritmus ovšem pokaždé vyplňuje celou tabulku. Ukažte, jak ho zrychlit, aby počítal v čase $\mathcal{O}((n + m)(L(x,y) + 1))$.

6.* Jak by se výpočet editační vzdálenosti změnil, kdybychom mezi editační operace řadili i *prohození* dvou sousedních písmen?

7. Navrhněte algoritmus pro nalezení *nejdelší společné podposloupnosti* daných posloupností $x_1, \ldots, x_n$ a $y_1, \ldots, y_m$. Jak tento problém souvisí s editační vzdáleností a s grafem z obrázku 1.5?

8. Jak naopak najít *nejkratší společnou nadposloupnost* dvou posloupností A a B? Tím se myslí nejkratší posloupnost, která obsahuje jako podposloupnosti jak A, tak B.

1.4. Optimální vyhledávací stromy

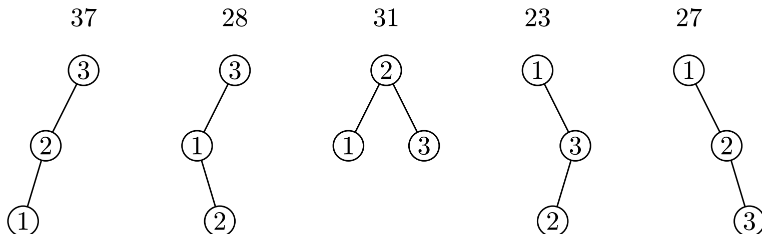
Když jsme vymýšleli binární vyhledávací stromy (BVS), uměli jsme zařídit, aby žádný prvek neležel příliš hluboko. Hned několik způsobů vyvažování nám zaručilo logaritmickou hloubku stromu. Co kdybychom ale věděli, že se na některé prvky budeme ptát mnohem častěji než na jiné? Nevyplatilo by se potom umístit tyto „oblíbené“ prvky blízko ke kořeni, byť by to znamenalo další prvky posunout níže?

Vyzkoušejme si to se třemi prvky. Na prvek 1 se budeme ptát celkem 10krát, na 2 jen jednou, na 3 celkem 5krát. Obrázek 1.6 ukazuje možné tvary vyhledávacího stromu a jejich *ceny* – počty vrcholů navštívených během všech 16 vyhledávání. Například pro prostřední, dokonale vyvážený strom nahlédneme při hledání prvku 1 do 2 vrcholů, při hledání 2 do 1 vrcholu a při hledání 3 opět do 2 vrcholů. Celková cena tedy činí $10 \cdot 2 + 1 \cdot 1 + 5 \cdot 2 = 31$. Následující strom ovšem dosahuje nižší ceny 23, protože často používaná 1 leží v kořeni.

Pojďme se na tento problém podívat obecněji. Máme n prvků s klíči $x_1 < \ldots < x_n$ a kladnými vahami $w_1, \ldots, w_n$. Každému binárnímu vyhledávacímu stromu pro tuto množinu klíčů přidělíme *cenu* $C = \sum_i w_i \cdot h_i$, kde h_i je hloubka klíče x_i (hloubky tentokrát počítáme od jedničky). Chceme najít *optimální vyhledávací strom*, tedy ten s nejnižší cenou.

Rekurzivní řešení

Představme si, že nám někdo napověděl, jaký prvek x_i se nachází v kořeni optimálního stromu. Hned víme, že levý podstrom obsahuje klíče $x_1, \ldots, x_{i-1}$ a pravý



Obr. 1.6: Cena hledání v různých vyhledávacích stromech

podstrom klíče $x_{i+1}, \ldots, x_n$. Navíc oba tyto podstromy musí být optimální – jinak bychom je mohli vyměnit za optimální a tím celý strom zlepšit.

Pokud nám prvek v kořeni nikdo nenapoví, vystačíme si sami: vyzkoušíme všechny možnosti a vybereme tu, která povede na minimální cenu. Levý a pravý podstrom pokaždé sestrojíme rekurzivním zavoláním téhož algoritmu.

Původní problém tedy postupně rozkládáme na podproblémy. V každém z nich hledáme optimální strom pro nějaký souvislý úsek klíčů $x_i, \ldots, x_j$. Zatím se spokojíme s tím, že spočítáme cenu tohoto stromu. Tím vznikne funkce $\text{OptStrom}(i, j)$ popsaná níže.

Funkce vyzkouší všechny možné kořeny, pro každý z nich rekurzivně spočítá optimální cenu c_ℓ levého podstromu a c_p pravého. Zbývá domyslet, jak z těchto cen spočítat cenu celého stromu. Všem prvkům v levém podstromu jsme zvýšili hloubku o 1, takže cena podstromu vzrostla o součet vah těchto prvků. Podobně to bude v pravém podstromu. Navíc přibyly dotazy na kořen, který má hloubku 1, takže přispívají k ceně přesně vahou kořene. Váhu každého prvku jsme tedy přičetli právě jednou, takže celková cena stromu činí $c_\ell + c_p + (w_i + \ldots + w_j)$.

Algoritmus $\text{OptStrom}(i, j)$ (cena optimálního BVS rekurzivně)

Vstup: Klíče $x_i, \ldots, x_j$ s vahami $w_i, \ldots, w_j$

1. Pokud $i > j$, vrátíme 0. ◁ *prázdný úsek dává prázdný strom*
2. $W \leftarrow w_i + \ldots + w_j$ ◁ *celková váha prvků*
3. $C \leftarrow +\infty$ ◁ *zatím nejlepší cena stromu*
4. Pro $k = i, \ldots, j$: ◁ *různé volby kořene*
5. $\quad c_\ell \leftarrow \text{OptStrom}(i, k-1)$ ◁ *levý podstrom*
6. $\quad c_p = \text{OptStrom}(k+1, j)$ ◁ *pravý podstrom*
7. $\quad C = \min(C, c_\ell + c_p + W)$ ◁ *cena celého stromu*

Výstup: Cena C optimálního vyhledávacího stromu

Jako obvykle jsme napoprvé získali exponenciální řešení, které půjde zrychlit kešováním spočítaných mezivýsledků. Budeme-li si pamatovat hodnoty $T[i, j] = \text{OptStrom}(i, j)$, spočítáme celkově $\mathcal{O}(n^2)$ políček tabulky a každým strávíme čas $\mathcal{O}(n)$. Celkem tedy algoritmus poběží v čase $\mathcal{O}(n^3)$.

Iterativní řešení

Nyní obrátíme směr výpočtu. Využijeme toho, že odpověď pro daný úsek závisí pouze na odpovědích pro kratší úseky. Proto můžeme tabulku mezivýsledků vyplňovat od nejkratších úseků k nejdelším. Tím vznikne následující iterativní algoritmus. Oproti předchozímu řešení si navíc budeme pro každý úsek pamatovat optimální kořen, což nám za chvíli usnadní rekonstrukci optimálního stromu.

Algoritmus OptStrom2 (cena optimálního BVS iterativně)

Vstup: Klíče $x_1, \ldots, x_n$ s vahami $w_1, \ldots, w_n$

1. Pro $i = 1, \ldots, n+1$: $T[i, i-1] \leftarrow 0$ ◁ *prázdné stromy nic nestojí*
2. Pro $\ell = 1, \ldots, n$: ◁ *délky úseků*
3. Pro $i = 1, \ldots, n - \ell + 1$: ◁ *začátky úseků*
4. $j \leftarrow i + \ell - 1$ ◁ *konec aktuálního úseku*
5. $W \leftarrow w_i + \ldots + w_j$ ◁ *celková váha úseku*
6. $T[i, j] \leftarrow +\infty$
7. Pro $k = i, \ldots, j$: ◁ *možné kořeny*
8. $C \leftarrow T[i, k-1] + T[k+1, j] + W$ ◁ *cena stromu*
9. Pokud $C < T[i, j]$: ◁ *průběžné minimum*
10. $T[i, j] \leftarrow C$
11. $K[i, j] \leftarrow k$

Výstup: Cena $T[1, n]$ optimálního stromu, pole K s optimálními kořeny

Spočítejme časovou složitost. Vnitřní cyklus (kroky 4 až 11) běží v čase $\mathcal{O}(n)$ a spouští se $\mathcal{O}(n^2)$-krát. To celkem dává $\mathcal{O}(n^3)$.

Odvodit ze zapamatovaných kořenů skutečnou podobu optimálního stromu už bude hračka. Kořenem je prvek s indexem $r = K[1, n]$. Jeho levým synem bude kořen optimálního stromu pro úsek $1, \ldots, r-1$, což je prvek s indexem $K[1, r-1]$, a tak dále. Z této úvahy ihned plyne následující rekurzivní algoritmus. Zavoláme-li OptStromReko$(1, n)$, vrátí nám celý optimální strom.

Algoritmus OptStromReko(i, j) (konstrukce optimálního BVS)

Vstup: Klíče $x_i, \ldots, x_j$, pole K spočítané algoritmem OptStrom2

1. Pokud $i > j$, vrátíme prázdný strom.
2. $r \leftarrow K[i, j]$ ◁ *kolikátý prvek je v kořeni?*
3. Vytvoříme nový vrchol v s klíčem x_r.
4. Jako levého syna nastavíme OptStromReko$(i, r-1)$.
5. Jako pravého syna nastavíme OptStromReko$(r+1, j)$.

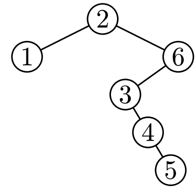
Výstup: Optimální vyhledávací strom s kořenem v

Samotná rekonstrukce stráví v každém rekurzivním volání konstantní čas a vyrobí přitom jeden vrchol stromu. Jelikož celkem vytvoříme n vrcholů, stihneme to v čase $\mathcal{O}(n)$. Celkem tedy hledáním optimálního stromu strávíme čas $\mathcal{O}(n^3)$. Dodejme, že existuje i kvadratický algoritmus (cvičení 7).

$w_1 = 1$
$w_2 = 10$
$w_3 = 3$
$w_4 = 2$
$w_5 = 1$
$w_6 = 9$

T	0	1	2	3	4	5	6
1	0	1	12	18	24	28	52
2	–	0	10	16	22	26	50
3	–	–	0	3	7	10	25
4	–	–	–	0	2	4	16
5	–	–	–	–	0	1	11
6	–	–	–	–	–	0	9
7	–	–	–	–	–	–	0

K	1	2	3	4	5	6
1	1	2	2	2	2	2
2	–	2	2	2	2	2
3	–	–	3	3	3	6
4	–	–	–	4	4	6
5	–	–	–	–	5	6
6	–	–	–	–	–	6



Obr. 1.7: Ukázka výpočtu algoritmu
OptStrom2 a nalezený optimální strom

Abstraktní pohled na dynamické programování

Na závěr se zkusme zamyslet nad tím, co mají jednotlivé aplikace dynamického programování v této kapitole společného – tedy kromě toho, že jsme je odvodili z rekurzivních algoritmů zavedením kešování.

Pokaždé umíme najít vhodný systém podproblémů – těm se často říká *stavy* dynamického programování. Závislosti mezi těmito podproblémy tvoří acyklický orientovaný graf. Díky tomu můžeme všechny stavy procházet v topologickém uspořádání a vždy mít připraveny všechny mezivýsledky potřebné k výpočtu aktuálního stavu.

Aby tento přístup fungoval, nesmí být stavů příliš mnoho: v našich případech jich bylo lineárně nebo kvadraticky. Každý stav jsme pak uměli spočítat v nejhůře lineárním čase, takže jsme dostali samé příjemně polynomiální algoritmy.

Někdy může být dynamické programování zajímavé i s exponenciálně mnoha stavy. Sice pak dostaneme algoritmus o exponenciální složitosti, ale i ten může být rychlejší než jiná možná řešení. Příklady tohoto typu najdete v oddílu **??**.

Cvičení

1. Optimální vyhledávací strom můžeme také definovat pomocí pravděpodobností. Nechť se na jednotlivé klíče ptáme náhodně, přičemž s pravděpodobností p_i se zeptáme na klíč x_i. Počet vrcholů navštívených při hledání se pak chová jako náhodná veličina se střední hodnotou $\sum_i p_i h_i$ (h_i je opět hloubka i-tého vrcholu). Zkuste formulovat podobný algoritmus v řeči těchto středních hodnot. Jak bude fungovat argument se skládáním stromů z podstromů?
2. Navrhněte, jak rovnou při výpočtu vah konstruovat strom. Využijte toho, že se více vrcholů může odkazovat na tytéž podstromy.
3. Rozmyslete si, že nastavíme-li všem prvkům stejnou váhu, vyjde dokonale vyvážený strom.
4. Jak se algoritmus změní, pokud budeme uvažovat i neúspěšné dotazy? Nejjednodušší je představit si, že váhy přidělujeme i externím vrcholům stromu, jež odpovídají intervalům (x_i, x_{i+1}) mezi klíči.

5. Co jsou v případě optimálních stromů stavy dynamického programování a jak vypadá graf jejich závislostí?

6.** *Knuthova nerovnost:* Nechť $K[i, j]$ je kořen spočítaný algoritmem OptStrom2 pro úsek $x_i, \ldots, x_j$ (je to tedy nejlevější z optimálních kořenů). Donald Knuth dokázal, že platí $K[i, j-1] \leq K[i, j] \leq K[i+1, j]$. Zkuste to dokázat i vy.

7.* *Rychlejší algoritmus:* Vymyslete jak pomocí nerovnosti z předchozího cvičení zrychlit algoritmus OptStrom2 na $\mathcal{O}(n^2)$.

8. *Součin matic:* Násobíme-li matice $X \in \mathbb{R}^{a \times b}$ a $Y \in \mathbb{R}^{b \times c}$ podle definice, počítáme $a \cdot b \cdot c$ součinů čísel. Pokud chceme spočítat maticový součin $X_1 \times \ldots \times X_n$, výsledek nezávisí na uzávorkování, ale časová složitost (měřená pro jednoduchost počtem součinů čísel) ano. Vymyslete algoritmus, který stanoví, jak výraz uzávorkovat, abychom složitost minimalizovali.

9. *Minimální triangulace:* Konvexní mnohoúhelník můžeme triangulovat, tedy rozřezat neprotínajícími se úhlopříčkami na trojúhelníky. Nalezněte takovou triangulaci, aby součet délek řezů byl nejmenší možný.

10.* *Optimalizace na stromech:* Ukažte, že předchozí dvě cvičení lze formulovat jako hledání optimálního binárního stromu vzhledem k nějaké cenové funkci. Rozšiřte algoritmy z tohoto oddílu, aby uměly pracovat s obecnými cenovými funkcemi a plynulo z nich automaticky i řešení minulých cvičení.