1. Rozděl a panuj

Potkáme-li spletitý problém, často pomáhá rozdělit ho na jednodušší části a s těmi se pak vypořádat postupně. Jak říkali staří Římané: *rozděl a panuj.*[1] Tato zásada se pak osvědčila nejen ve starořímské politice, ale také o dvě tisíciletí později při návrhu algoritmů.

Nás v této kapitole bude přirozeně zajímat zejména algoritmická stránka věci. Naším cílem bude rozkládat zadaný problém na menší podproblémy a z jejich výsledků pak skládat řešení celého problému. S jednotlivými podproblémy potom naložíme stejně – opět je rozložíme na ještě menší a tak budeme pokračovat, než se dostaneme k tak jednoduchým vstupům, že je už umíme vyřešit přímo.

Myšlenka je to trochu bláznivá, ale často vede k překvapivě jednoduchému, rychlému a obvykle rekurzivnímu algoritmu. Postupně ji použijeme na třídění posloupností, násobení čísel i matic a hledání k-tého nejmenšího ze zadaných prvků.

Nejprve si tuto techniku ovšem vyzkoušíme na jednoduchém hlavolamu známém pod názvem Hanojské věže.

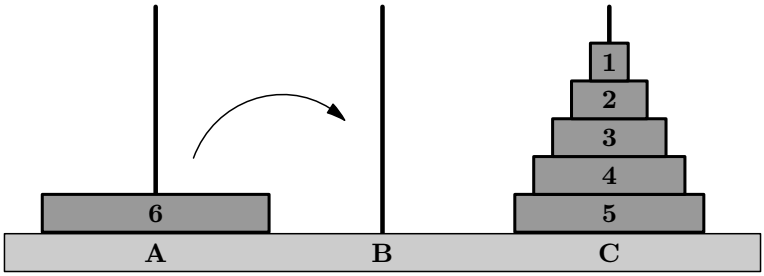
1.1. Hanojské věže

Legenda vypráví, že v daleké Hanoji stojí starobylý klášter. V jeho sklepení se skrývá rozlehlá jeskyně, v níž stojí tři sloupy. Na nich je navlečeno celkem 64 zlatých disků různých velikostí. Za úsvitu věků byly všechny disky srovnané podle velikosti na prvním sloupu: největší disk dole, nejmenší nahoře. Od té doby mniši každý den za hlaholu zvonů obřadně přenesou nejvyšší disk z některého sloupu na jiný sloup. Tradice jim přitom zakazuje položit větší disk na menší a také zopakovat již jednou použité rozmístění disků. Říká se, že až se všechny disky opět sejdou na jednom sloupu, nastane konec světa.

Nabízí se samozřejmě otázka, za jak dlouho se mnichům může podařit splnit svůj úkol a celou „věž“ z disků přenést. Zamysleme se nad tím rovnou pro obecný počet disků a očíslujme si je podle velikosti od 1 (nejmenší disk) do n (největší). Také si označme sloupy: na sloupu A budou disky na počátku, na sloup B je chceme přemístit a sloup C můžeme používat jako pomocný.

Ať už zvolíme jakýkoliv postup, někdy během něj musíme přemístit největší disk na sloup B. V tomto okamžiku musí být na jiném sloupu samotný největší disk a všechny ostatní disky na zbývajícím sloupu (viz obrázek). Nabízí se tedy nejprve přemístit disky $1, \ldots, n-1$ z A na C, pak přesunout disk n z A na B a konečně přestěhovat disky $1, \ldots, n-1$ z C na B. Tím jsme tedy problém přesunu věže výšky n převedli na dva problémy s věží výšky $n-1$. Ty ovšem můžeme vyřešit stejně, rekurzivním zavoláním téhož algoritmu. Zastavíme se až u věže výšky 1, kterou zvládneme přemístit jedním tahem. Algoritmus bude vypadat takto:

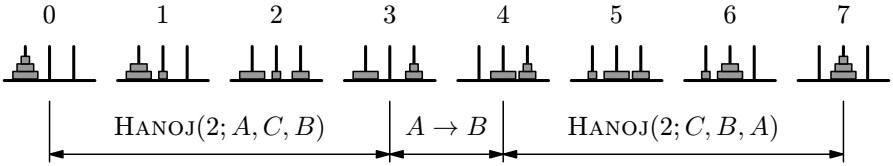
[1] Oni to říkali spíš latinsky: *divide et impera*.



Obr. 1.1: Stav hry při přenášení největšího disku ($n = 5$)

Algoritmus HANOJ
Vstup: Výška věže n; sloupy A (zdrojový), B (cílový), C (pomocný)

1. Pokud je $n = 1$, přesuneme disk 1 z A na B.
2. Jinak:
3. Zavoláme HANOJ$(n-1; A, C, B)$.
4. Přesuneme disk n z A na B.
5. Zavoláme HANOJ$(n-1; C, B, A)$.



Obr. 1.2: Průběh algoritmu HANOJ pro $n = 3$

Ujistěme se, že náš algoritmus při přenášení věží neporuší pravidla. Když v kroku 3 přesouváme disk z A na B, o všech menších discích víme, že jsou na věži C, takže na ně určitě nic nepoložíme. Taktéž nikdy nepoužijeme žádnou konfiguraci dvakrát. K tomu si stačí uvědomit, že se konfigurace navštívené během obou rekurzivních volání liší polohou n-tého disku.

Spočítejme, kolik tahů náš algoritmus spotřebuje. Pokud si označíme $T(n)$ počet tahů použitý pro věž výšky n, bude platit:

$$T(1) = 1,$$
$$T(n) = 2 \cdot T(n-1) + 1.$$

Z tohoto vztahu okamžitě zjistíme, že $T(2) = 3$, $T(3) = 7$ a $T(4) = 15$. Nabízí se, že by mohlo platit $T(n) = 2^n - 1$. To snadno ověříme indukcí: Pro $n = 1$ je tvrzení pravdivé. Pokud platí pro $N - 1$, dostaneme:

$$T(n) = 2 \cdot (2^{n-1} - 1) + 1 = 2 \cdot 2^{n-1} - 2 + 1 = 2^n - 1.$$

Časová složitost algoritmu je tedy exponenciální. Ve cvičení 1 ale snadno ukážeme, že exponenciální počet tahů je nejlepší možný. Pro $n = 64$ proto mniši budou pracovat minimálně $2^{64} \approx 1.84 \cdot 10^{19}$ dní, takže konce světa se alespoň po nejbližších pár biliard let nemusíme obávat.

Cvičení

1. Dokažte, že algoritmus HANOJ je nejlepší možný, čili že $2^n - 1$ tahů je opravdu potřeba.
2. Přidejme k regulím hanojských mnichů ještě jedno pravidlo: je zakázáno přenášet disky přímo ze sloupu A na B nebo opačně (každý přesun se tedy musí uskutečnit přes sloup C). I nyní je problém řešitelný. Jak a s jakou časovou složitostí?
3. Dokažte, že algoritmus z předchozího cvičení navštíví každé korektní rozmístění disků na sloupy (tj. takové, v němž nikde neleží větší disk na menším) právě jednou.
4. Vymyslete algoritmus, který pro zadané rozmístění disků na sloupy co nejrychleji přemístí všechny disky na libovolný jeden sloup.

5.* Navrhněte takové řešení Hanojských věží, které místo rekurze bude umět z pořadového čísla tahu rovnou určit, který disk přesunout a kam.

1.2. Třídění sléváním – Mergesort

Zopakujme si, jakým způsobem jsme vyřešili úlohu z minulé kapitoly. Nejprve jsme ji rozložili na dvě úlohy menší (věže výšky $n - 1$), ty jsme vyřešili rekurzivně, a pak jsme z jejich výsledků přidáním jednoho tahu utvořili výsledek úlohy původní. Podívejme se nyní, jak se podobný přístup osvědčí na problému třídění posloupnosti. Ukážeme rekurzivní verzi *třídění sléváním* – algoritmu Mergesort z oddílu **??**.

Dostaneme-li posloupnost n prvků, jistě ji můžeme rozdělit na dvě části poloviční délky (řekněme prvních $\lfloor n/2 \rfloor$ a zbývajících $\lceil n/2 \rceil$ prvků). Ty setřídíme rekurzivním zavoláním téhož algoritmu. Setříděné poloviny posléze *slijeme* dohromady do jedné setříděné posloupnosti a máme výsledek. Když ještě ošetříme triviální případ $n = 1$, aby se nám rekurze zastavila (na to není radno zapomínat), dostaneme následující algoritmus.

Algoritmus MERGESORT (rekurzivní třídění sléváním)
Vstup: Posloupnost $a_1, \ldots, a_n$ k setřídění

1. Pokud $n = 1$, vrátíme jako výsledek $b_1 = a_1$ a skončíme.
2. $x_1, \ldots, x_{\lfloor n/2 \rfloor} \leftarrow \text{MERGESORT}(a_1, \ldots, a_{\lfloor n/2 \rfloor})$
3. $y_1, \ldots, y_{\lceil n/2 \rceil} \leftarrow \text{MERGESORT}(a_{\lfloor n/2 \rfloor + 1}, \ldots, a_n)$
4. $b_1, \ldots, b_n \leftarrow \text{MERGE}(x_1, \ldots, x_{\lfloor n/2 \rfloor}; y_1, \ldots, y_{\lceil n/2 \rceil})$

Výstup: Setříděná posloupnost $b_1, \ldots, b_n$

Procedura MERGE má na vstupu dva vzestupně setříděné sousedící úseky prvků v poli a provádí samotné jejich slévání do jediného setříděného úseku. Byla popsána

v oddílu **??** a připomeneme jen, že má lineární časovou složitost vzhledem k délce slévaných úseků a vyžaduje lineárně velkou pomocnou paměť ve formě pomocného pole.

Rozbor složitosti

Spočítejme, kolik času tříděním strávíme. Slévání ve funkci MERGE má lineární časovou složitost. Složitost samotného třídění můžeme popsat takto:

$$\begin{aligned} T(1) &= 1, \\ T(n) &= 2 \cdot T(n/2) + cn. \end{aligned}$$

První rovnost nám popisuje, co se stane, když už v posloupnosti zbývá jediný prvek. Dobu trvání této operace jsme si přitom zvolili za jednotku času. Druhá rovnost pak odpovídá „zajímavé“ části algoritmu. Čas cn potřebujeme na rozdělení posloupnosti a slití setříděných kusů. Mimo to voláme dvakrát sebe sama na vstup velikosti $n/2$, což pokaždé trvá $T(n/2)$. (Zde se dopouštíme malého podvůdku a předpokládáme, že n je mocnina dvojky, takže se nemusíme starat o zaokrouhlování. V oddílu 1.4 uvidíme, že to opravdu neuškodí.)

Jak tuto rekurentní rovnici vyřešíme? Zkusme v druhém vztahu za $T(n/2)$ dosadit podle téže rovnice:

$$\begin{aligned} T(n) &= 2 \cdot (2 \cdot T(n/4) + cn/2) + cn = \\ &= 4 \cdot T(n/4) + 2cn. \end{aligned}$$

To můžeme dále rozepsat na:

$$T(n) = 4 \cdot (2 \cdot T(n/8) + cn/4) + 2cn = 8 \cdot T(n/8) + 3cn.$$

Vida, pravá strana se chová poměrně pravidelně. Můžeme obecně popsat, že po k rozepsáních dostaneme:

$$T(n) = 2^k \cdot T(n/2^k) + kcn.$$

Nyní zvolme k tak, aby $n/2^k$ bylo rovno jedné, čili $k = \log_2 n$. Dostaneme:

$$\begin{aligned} T(n) &= 2^{\log_2 n} \cdot T(1) + \log_2 n \cdot cn = \\ &= n + cn \log_2 n. \end{aligned}$$

Časová složitost Mergesortu je tedy $\Theta(n \log n)$, stejně jako u nerekurzivní verze. Jaké má paměťové nároky? Nerekurzivní Mergesort vyžadoval lineárně velké pomocné pole na slévání. Pojďme dokázat, že nám lineární množství *pomocné paměti* (to je paměť, do které nepočítáme velikost vstupu a výstupu) také úplně stačí.

Zavoláme-li funkci MERGESORT na vstup velikosti n, potřebujeme si pamatovat lokální proměnné této funkce (poloviny vstupu a jejich setříděné verze – dohromady $\Theta(n)$ paměti) a pak také, kam se z funkce máme vrátit (na to potřebujeme konstantní

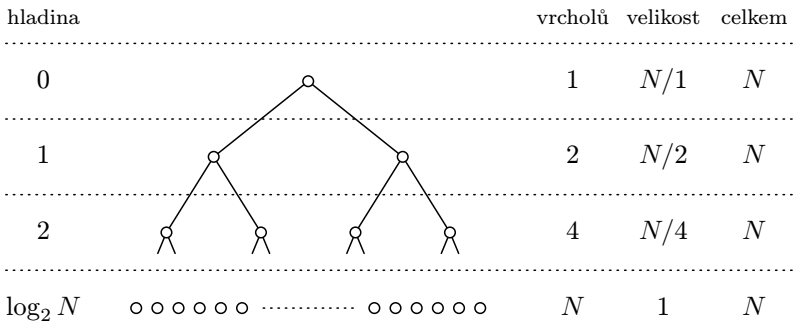
množství paměti). Mimo to nějakou paměť spotřebují obě rekurzivní volání, ale jelikož vždy běží nejvýše jedno z nich, stačí ji započítat jen jednou. Opět dostaneme jednoduchou rekurentní rovnici:

$$M(1) = 1,$$
$$M(n) = dn + M(n/2)$$

pro nějakou kladnou konstantu d. To nám pro $M(n)$ dává geometrickou řadu $dn + dn/2 + dn/4 + \ldots$, která má součet $\Theta(n)$. Prostorová složitost je tedy opravdu lineární.

Stromy rekurze

Někdy je jednodušší místo počítání s rekurencemi odhadnout složitost úvahou o *stromu rekurzivních volání*. Nakreslíme strom, jehož vrcholy budou odpovídat jednotlivým podúlohám, které řešíme. Kořen je původní úloha velikosti n, jeho dva synové podúlohy velikosti $n/2$. Pak následují 4 podúlohy velikosti $n/4$, a tak dále až k listům, což jsou podúlohy o jednom prvku. Obecně i-tá hladina bude mít 2^i vrcholů pro podúlohy velikosti $n/2^i$, takže hladin bude celkem $\log_2 n$.



Obr. 1.3: Strom rekurze algoritmu MergeSort

Rozmysleme si nyní, kolik času kde trávíme. Rozdělování i slévání jsou lineární, takže jeden vrchol na i-té hladině spotřebuje čas $\Theta(n/2^i)$. Celá i-tá hladina přispěje časem $2^i \cdot \Theta(n/2^i) = \Theta(n)$. Když tento čas sečteme přes všechny hladiny, dostaneme celkovou časovou složitost $\Theta(n \log n)$.

Všimněte si, že tento „stromový důkaz“ docela věrně odpovídá tomu, jak jsme předtím rozepisovali rekurenci. Situace po k-tém rozepsání totiž popisuje řez stromem rekurze na k-té hladině. Vyšší hladiny jsme již sečetli, nižší nás teprve čekají.

I prostorové nároky algoritmu můžeme vyčíst ze stromu. V každém vrcholu potřebujeme paměť lineární s velikostí podúlohy, ve vrcholu na i-té hladině tedy $\Theta(n/2^i)$. V paměti je vždy vrchol, který právě zpracováváme, a všichni jeho předci. Maximálně tedy nějaká cesta z kořene do listu. Sečteme-li prostor zabraný vrcholy na takové cestě, dostaneme $\Theta(n) + \Theta(n/2) + \Theta(n/4) + \ldots + \Theta(1) = \Theta(n)$.

Cvičení

1. Naprogramujte třídění seznamu pomocí Mergesortu. Jde to snáze rekurzivně, nebo cyklem?
2. Popište třídicí algoritmus, který bude vstup rozkládat na více než dvě části a ty pak rekurzivně třídit. Může být rychlejší než náš Mergesort?

1.3. Násobení čísel – Karacubův algoritmus

Při třídění metodou Rozděl a panuj jsme získali algoritmus, který byl sice elegantnější než předchozí třídicí algoritmy, ale měl stejnou časovou složitost. Pojďme se nyní podívat na příklad, kdy nám tato metoda pomůže k efektivnějšímu algoritmu. Půjde o násobení dlouhých čísel.

Mějme n-ciferná čísla X a Y, která chceme vynásobit. Rozdělíme je na horních $n/2$ a dolních $n/2$ cifer (pro jednoduchost opět předpokládejme, že n je mocnina dvojky). Platí tedy:

$$X = A \cdot 10^{n/2} + B,$$
$$Y = C \cdot 10^{n/2} + D$$

pro nějaká $(n/2)$-ciferná čísla A, B, C, D. Hledaný součin XY můžeme zapsat takto:

$$XY = AC \cdot 10^n + (AD + BC) \cdot 10^{n/2} + BD.$$

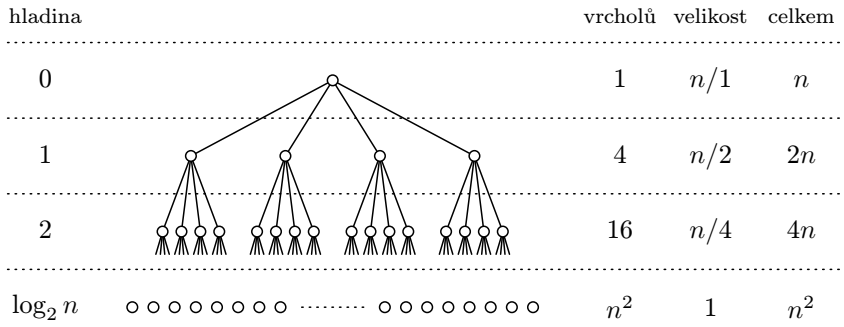
Nabízí se spočítat rekurzivně součiny AC, AD, BC a BD a pak z nich složit výsledek. Skládání obnáší několik $2n$-ciferných sčítání a několik násobení mocninou desítky, to druhé ovšem není nic jiného, než doplňování nul na konec čísla. Řešíme tedy čtyři podproblémy poloviční velikosti a k tomu spotřebujeme lineární čas. Pro časovou složitost proto platí:

$$T(1) = 1,$$
$$T(n) = 4 \cdot T(n/2) + \Theta(n).$$

Podobně jako minule, i zde k vyřešení rovnice stačí rozmyslet si, jak vypadá strom rekurzivních volání. Na jeho i-té hladině se nachází 4^i vrcholů s podproblémy o $n/2^i$ cifrách. V každém vrcholu tedy trávíme čas $\Theta(n/2^i)$ a na celé hladině $4^i \cdot \Theta(n/2^i) = \Theta(2^i \cdot n)$. Jelikož hladin je opět $\log_2 n$, strávíme jenom na poslední hladině čas $\Theta(2^{\log_2 n} \cdot n) = \Theta(n^2)$. Oproti běžnému „školnímu“ násobení jsme si tedy vůbec nepomohli.

Po počátečním neúspěchu se svého plánu nevzdáme, nýbrž nahlédneme, že ze zmíněných čtyř násobení poloviční velikosti můžeme jedno ušetřit. Když vynásobíme $(A+B)\cdot(C+D)$, dostaneme $AC+AD+BC+BD$. To se od závorky $(AD+BC)$, kterou potřebujeme, liší jen o $AC + BD$. Tyto dva členy nicméně známe, takže je můžeme odečíst. Získáme následující formulku pro XY:

$$XY = AC \cdot 10^n + ((A + B)(C + D) - AC - BD) \cdot 10^{n/2} + BD.$$



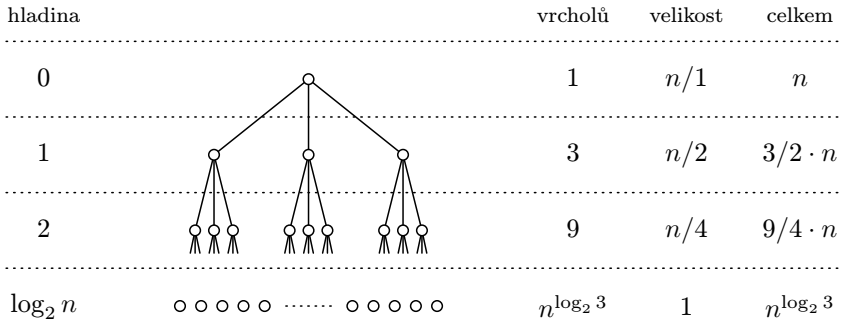
Obr. 1.4: První pokus o násobení rekurzí

Časová složitost se touto úpravou změní následovně:

$$T(n) = 3 \cdot T(n/2) + \Theta(n).$$

Sledujme, jak se změnil strom: na i-té hladině nalezneme 3^i vrcholů s $(n/2^i)$-ciferný-mi problémy a jeho hloubka bude nadále činit $\log_2 n$. Na i-té hladině nyní dohromady trávíme čas $\Theta(n \cdot (3/2)^i)$, v součtu přes všechny hladiny dostaneme:

$$T(n) = \Theta(n \cdot [(3/2)^0 + (3/2)^1 + \ldots (3/2)^{\log_2 n}]).$$



Obr. 1.5: Strom rekurze algoritmu Násob

Výraz v hranatých závorkách je geometrická řada s kvocientem 3/2. Tu můžeme sečíst obvyklým způsobem na

$$\frac{(3/2)^{1+\log_2 n} - 1}{3/2 - 1}.$$

Když zanedbáme konstanty, obdržíme $(3/2)^{\log_2 n}$. To dále upravíme na

$$(2^{\log_2(3/2)})^{\log_2 n} = 2^{\log_2(3/2) \cdot \log_2 n} = (2^{\log_2 n})^{\log_2(3/2)} = n^{\log_2(3/2)} = n^{\log_2 3 - 1}.$$

Časová složitost našeho algoritmu tedy činí $\Theta(n \cdot n^{\log_2 3-1}) = \Theta(n^{\log_2 3}) \approx \Theta(n^{1.59})$. To je již podstatně lepší než obvyklý kvadratický algoritmus. Paměti nám bude stačit lineárně mnoho (viz cvičení 3).

Algoritmus NÁSOB (násobení čísel – Karacubův algoritmus)
Vstup: n-ciferná čísla X a Y

1. Pokud $n \leq 1$, vrátíme $Z = XY$ a skončíme.
2. $k = \lfloor n/2 \rfloor$
3. $A \leftarrow \lfloor X/10^k \rfloor$, $B \leftarrow X \bmod 10^k$
4. $C \leftarrow \lfloor Y/10^k \rfloor$, $D \leftarrow Y \bmod 10^k$
5. $P \leftarrow$ NÁSOB(A, C)
6. $Q \leftarrow$ NÁSOB(B, D)
7. $R \leftarrow$ NÁSOB$(A + B, C + D)$
8. $Z \leftarrow P \cdot 10^n + (R - P - Q) \cdot 10^k + Q$

Výstup: Součin $Z = XY$

Dodejme ještě, že tento princip není žádná novinka: objevil ho už v roce 1960 Anatolij Alexejevič Karacuba. Dnes jsou však známé i rychlejší metody. Ty jednodušší z nich využívají podobný princip rozkladu na podproblémy, ovšem s více částmi (cvičení 5–6). Pokročilejší algoritmy jsou pak často založeny na Fourierově transformaci, s níž se potkáme v kapitole **??**. Arnold Schönhage v roce 1979 ukázal, že obdobným způsobem lze dokonce dosáhnout lineární časové složitosti. Násobení je tedy, alespoň teoreticky, stejně těžké jako sčítání a odčítání. Ve cvičeních 7–9 navíc odvodíme, že dělit lze stejně rychle jako násobit.

Cvičení

1. Pozornému čtenáři jistě neuniklo, že se v našem rozboru časové složitosti skrývá drobná chybička: čísla $A + B$ a $C + D$ mohou mít více než $n/2$ cifer, konkrétně $\lceil n/2 \rceil + 1$. Ukažte, že to časovou složitost algoritmu neovlivní.
2. Problému z předchozího cvičení se lze také vyhnout jednoduchou úpravou algoritmu. Místo $(A + B)(C + D)$ počítejte $(A - B)(C - D)$.
3. Dokažte, že funkce NÁSOB má lineární prostorovou složitost. (Podobnou úvahou jako u Mergesortu.)
4. Algoritmus NÁSOB je sice pro velká n rychlejší než školní násobení, ale pro malé vstupy se ho nevyplatí použít, protože režie na rekurzi a spojování mezivýsledků bude daleko větší než čas spotřebovaný kvadratickým algoritmem. Často proto pomůže „zkřížit“ chytrý rekurzivní algoritmus s nějakým primitivním. Pokud velikosti vstupu klesne pod vhodně zvolenou konstantu n_0, rekurzi zastavíme a použijeme hrubou sílu. Zkuste si takový hybridní algoritmus pro násobení naprogramovat a experimentálně zjistit nejvýhodnější hodnotu hranice n_0.

5.* Zrychlete algoritmus NÁSOB tím, že budete číslo dělit na tři části a rekurzivně počítat pět součinů. Nazveme-li části čísla X po řadě X_2, X_1, X_0 a analogicky

pro Y, budeme počítat tyto součiny:

$$\begin{aligned}
W_0 &= X_0 Y_0, \\
W_1 &= (X_2 + X_1 + X_0)(Y_2 + Y_1 + Y_0), \\
W_2 &= (X_2 - X_1 + X_0)(Y_2 - Y_1 + Y_0), \\
W_3 &= (4X_2 + 2X_1 + X_0)(4Y_2 + 2Y_1 + Y_0), \\
W_4 &= (4X_2 - 2X_1 + X_0)(4Y_2 - 2Y_1 + Y_0).
\end{aligned}$$

Ukažte, že součin XY lze zapsat jako lineární kombinaci těchto mezivýsledků. Jakou bude mít tento algoritmus časovou složitost?

6.** Pomocí nápovědy k předchozímu cvičení ukažte, jak pro libovolné $r \geq 1$ čísla dělit na $r+1$ částí a rekurzivně počítat $2r+1$ součinů. Co z toho plyne pro časovou složitost násobení? Může se hodit Kuchařková věta z následujícího oddílu.

7.* Z rychlého násobení můžeme odvodit i efektivní algoritmus pro dělení. Hodí se k tomu Newtonova iterační metoda řešení rovnic, zvaná též metoda tečen. Vyzkoušíme si ji na výpočtu n cifer podílu $1/a$ pro $2^{n-1} \leq a < 2^n$. Uvážíme funkci $f(x) = 1/x - a$. Tato funkce nabývá nulové hodnoty pro $x = 1/a$ a její derivace je $f'(x) = -1/x^2$. Budeme vytvářet posloupnost aproximací kořene této funkce. Za počáteční aproximaci x_0 zvolíme 2^{-n}, hodnotu x_{i+1} získáme z x_i tak, že sestrojíme tečnu ke grafu funkce f v bodě $(x_i, f(x_i))$ a vezmeme si x-ovou souřadnici průsečíku této tečny s osou x. Pro tuto souřadnici platí $x_{i+1} = x_i - f(x_i)/f'(x_i) = 2x_i - ax_i^2$. Posloupnost $x_0, x_1, x_2, \ldots$ velmi rychle konverguje ke kořeni $x = 1/a$ a k jejímu výpočtu stačí pouze sčítání, odčítání a násobení čísel. Rozmyslete si, jak tímto způsobem dělit libovolné číslo libovolným.

8.** Dokažte, že newtonovské dělení z minulého cvičení nalezne podíl po $\mathcal{O}(\log n)$ iteracích. Pracuje tedy v čase $\mathcal{O}(M(n) \log n)$, kde $M(n)$ je čas potřebný na vynásobení dvou n-ciferných čísel.

9.** Logaritmu v předchozím odhadu se lze zbavit, pokud funkce $M(n)$ roste alespoň lineárně, čili platí $M(cn) = \mathcal{O}(cM(n))$ pro každé $c \geq 1$. Stačí pak v k-té iteraci algoritmu počítat pouze s $2^{\Theta(k)}$-ciferný mi čísly, čímž složitost klesne na $\mathcal{O}(M(n) + M(n/2) + M(n/4) + \ldots) = \mathcal{O}(M(n) \cdot (1 + 1/2 + 1/4 + \ldots)) = \mathcal{O}(M(n))$. Dělení je tedy stejně těžké jako násobení.

10. *Převod mezi soustavami:* Máme n-ciferné číslo v soustavě o základu z a chceme ho převést do soustavy o jiném základu. Ukažte, jak to metodou Rozděl a panuj zvládnout v čase $\mathcal{O}(M(n))$, kde $M(n)$ je čas potřebný na násobení n-ciferných čísel v soustavě o novém základu.

1.4. Kuchařková věta o složitosti rekurzivních algoritmů

U předchozích algoritmů založených na principu Rozděl a panuj jsme pozorovali několik různých časových složitostí: $\Theta(2^n)$ u Hanojských věží, $\Theta(n \log n)$ u Mergesortu a $\Theta(n^{1.59})$ u násobení čísel. Hned se nabízí otázka, jestli v těchto složitostech lze

nalézt nějaký řád. Pojďme to zkusit: Uvažme rekurzivní algoritmus, který vstup rozloží na a podproblémů velikosti n/b a z jejich výsledků složí celkovou odpověď v čase $\Theta(n^c)$.[2] Dovolíme-li si zamést zase pod rohožku případné zaokrouhlování předpokladem, že n je mocninou čísla b, bude příslušná rekurence vypadat takto:

$$\begin{aligned} T(1) &= 1, \\ T(n) &= a \cdot T(n/b) + \Theta(n^c). \end{aligned}$$

Použijeme osvědčenou metodu založenou na stromu rekurze. Jak tento strom vypadá? Každý vnitřní vrchol stromu má přesně a synů, takže na i-té hladině se nachází a^i vrcholů. Velikost problému se zmenšuje b-krát, proto na i-té hladině leží podproblémy velikosti n/b^i. Po $\log_b n$ hladinách se tudíž rekurze zastaví.

Nyní počítejme, kolik času kde strávíme. V jednom vrcholu i-té hladiny je to $\Theta((n/b^i)^c)$, na celé hladině pak $\Theta(a^i \cdot (n/b^i)^c)$. Tento výraz snadno upravíme na $\Theta(n^c \cdot (a/b^c)^i)$, což v součtu přes všechny hladiny dá:

$$\Theta(n^c \cdot [(a/b^c)^0 + (a/b^c)^1 + \ldots + (a/b^c)^{\log_b n}]).$$

Výraz v hranatých závorkách je opět nějaká geometrická řada, tentokrát s kvocientem $q = a/b^c$. Její chování bude proto záviset na tom, jak velký je kvocient:

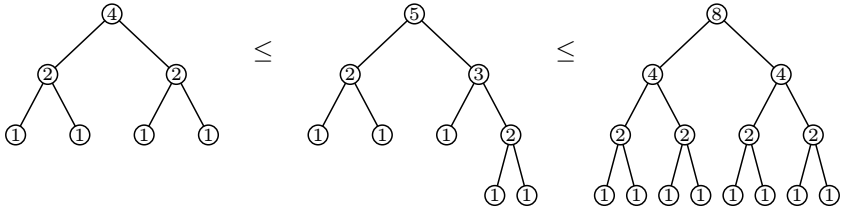
- $q = 1$: Všechny členy řady jsou rovny jedné, takže se řada sečte na $\log_b n + 1$. Tomu odpovídá časová složitost $T(n) = \Theta(n^c \log n)$. Tak se chová například Mergesort – na všech hladinách stromu se vykonává stejné množství práce.
- $q < 1$: I kdyby řada byla nekonečná, bude mít součet nejvýše $1/(1-q)$, a to je konstanta. Dostaneme tedy $T(n) = \Theta(n^c)$. To znamená, že podstatnou část času trávíme v kořeni stromu a zbytek je zanedbatelný. Algoritmus tohoto typu jsme ještě nepotkali.
- $q > 1$: Řadu sečteme na $(q^{1+\log_b n} - 1)/(q-1) = \Theta(q^{\log_b n})$, dominantní je tentokrát čas trávený v listech. To jsme už viděli u algoritmu na násobení čísel, zkusme tedy výraz upravit obdobně:

$$\begin{aligned} q^{\log_b n} = \left(\frac{a}{b^c}\right)^{\log_b n} = \frac{a^{\log_b n}}{(b^c)^{\log_b n}} = \frac{b^{\log_b a \cdot \log_b n}}{b^{c \cdot \log_b n}} = \\ = \frac{(b^{\log_b n})^{\log_b a}}{(b^{\log_b n})^c} = \frac{n^{\log_b a}}{n^c}. \end{aligned}$$

  Vyjde nám $T(n) = \Theta(n^c \cdot q^{\log_b n}) = \Theta(n^{\log_b a})$.

[2] Do tohoto schématu nám nezapadají Hanojské věže, ale ty jsou neobvyklé i tím, že jejich podproblémy jsou jen o jedničku menší než původní problém. Mimo to algoritmy s exponenciální složitostí jsou poněkud nepraktické.

Zbývá maličkost: vymést zpod rohožky případ, kdy n není mocninou čísla b. Tehdy dělení na podproblémy nebude úplně rovnoměrné – některé budou mít velikost $\lfloor n/b \rfloor$, jiné $\lceil n/b \rceil$. My se ale komplikovanému počítání vyhneme následující úvahou: označme si n^- nejbližší nižší mocninu b a n^+ nejbližší vyšší. Jelikož časová složitost s rostoucím n jistě neklesá, leží $T(n)$ mezi $T(n^-)$ a $T(n^+)$. Jenže n^- a n^+ se liší jen b-krát, což se do $T(\ldots)$ ve všech třech typech chování promítne pouze konstantou. Proto jsou $T(n^-)$ i $T(n^+)$ asymptoticky stejné a taková musí být i $T(n)$.[3]



Obr. 1.6: Obecné n jsme sevřeli mezi n^- a n^+

Zjistili jsme tedy, že hledaná funkce $T(n)$ se vždy chová jedním ze tří popsaných způsobů. To můžeme shrnout do následující „kuchařkové“ věty, známé také pod anglickým názvem *Master theorem:*

Věta (kuchařka na řešení rekurencí): Rekurentní rovnice $T(n) = a \cdot T(n/b) + \Theta(n^c)$, $T(1) = 1$ má pro konstanty $a \geq 1, b > 1, c \geq 0$ řešení:

- $T(n) = \Theta(n^c \log n)$, pokud $a/b^c = 1$;
- $T(n) = \Theta(n^c)$, pokud $a/b^c < 1$;
- $T(n) = \Theta(n^{\log_b a})$, pokud $a/b^c > 1$.

Cvičení

1. Nalezněte nějaký algoritmus, který odpovídá druhému typu chování ($q < 1$).

2.* Vylepšete kuchařkovou větu, aby pokrývala i případy, v nichž se velikosti podproblémů liší až o nějakou konstantu. To by se hodilo například u násobení čísel.

3.** *Kuchařka pro různě hladové jedlíky:* Jak by věta vypadala, kdybychom problém dělili na nestejně velké části? Tedy kdyby rekurence měla tvar $T(n) = T(\beta_1 n) + T(\beta_2 n) + \ldots + T(\beta_a n) + \Theta(n^c)$.

4. Řešte „nekuchařkovou“ rekurenci $T(n) = 2T(n/2) + \Theta(n \log n)$, $T(1) = 1$.

5. Jiná „nekuchařková“ rekurence: $T(n) = n^{1/2} \cdot T(n^{1/2}) + \Theta(n)$, $T(1) = 1$.

[3] To trochu připomíná „Větu o policajtech“ z matematické analýzy. Vlastně říkáme, že pokud $f(n) \leq g(n) \leq h(n)$ a existuje nějaká funkce $z(n)$ taková, že $f(n) = \Theta(z(n))$ a $h(n) = \Theta(z(n))$, pak také platí $g(n) = \Theta(z(n))$.

1.5. Násobení matic – Strassenův algoritmus

Nejen násobením čísel živ jest matematik. Často je potřeba násobit i složitější matematické objekty, zejména pak čtvercové matice. Pokud počítáme součin dvou matic tvaru $n \times n$ přesně podle definice, potřebujeme $\Theta(n^3)$ kroků. Jak v roce 1969 ukázal Volker Strassen, i zde dělení na menší podproblémy přináší ovoce v podobě rychlejšího algoritmu.

Nejprve si rozmyslíme, že stačí umět násobit matice, jejichž velikost je mocnina dvojky. Jinak stačí matice doplnit vpravo a dole nulami a nahlédnout, že vynásobením takto orámovaných matic získáme stejným způsobem orámovaný součin původních matic. Navíc orámované matice obsahují nejvýše čtyřikrát tolik prvků, takže se nemusíme obávat, že bychom tím algoritmus podstatně zpomalili.

Mějme tedy matice X a Y, obě tvaru $n \times n$ pro $n = 2^k$. Rozdělíme je na čtvrtiny – *bloky* tvaru $n/2 \times n/2$. Bloky matice X označíme A až D, bloky matice Y nazveme P až S. Pomocí těchto bloků můžeme snadno zapsat jednotlivé bloky součinu $X \cdot Y$:

$$X \cdot Y = \begin{pmatrix} A & B \\ C & D \end{pmatrix} \cdot \begin{pmatrix} P & Q \\ R & S \end{pmatrix} = \begin{pmatrix} AP + BR & AQ + BS \\ CP + DR & CQ + DS \end{pmatrix}.$$

Tento vztah vlastně vypadá úplně stejně jako klasická definice násobení matic, jen zde jednotlivá písmena nezastupují čísla, nýbrž bloky.

Jedno násobení matic $n \times n$ jsme tedy převedli na 8 násobení matic poloviční velikosti a režii $\Theta(n^2)$. Letmým nahlédnutím do kuchařkové věty z minulé kapitoly zjistíme, že vznikne opět kubický algoritmus. (Pozor, obvyklá terminologie je tu poněkud zavádějící – n zde neznačí velikost vstupu, nýbrž počet řádků matice; vstup je tedy velký n^2.)

Stejně jako u násobení čísel nás zachrání, že dovedeme jedno násobení ušetřit. Jen příslušné formule jsou daleko komplikovanější a připomínají králíka vytaženého z klobouku. Neprozradíme vám, jak kouzelník pan Strassen svůj trik vymyslel (sami neznáme žádný systematický postup, jak na to přijít), ale když už vzorce známe, není těžké ověřit, že opravdu fungují (viz cvičení). Formulky vypadají takto:

$$X \cdot Y = \begin{pmatrix} T_1 + T_4 - T_5 + T_7 & T_3 + T_5 \\ T_2 + T_4 & T_1 - T_2 + T_3 + T_6 \end{pmatrix},$$

kde:

$$\begin{aligned} T_1 &= (A + D) \cdot (P + S) & T_5 &= (A + B) \cdot S \\ T_2 &= (C + D) \cdot P & T_6 &= (C - A) \cdot (P + Q) \\ T_3 &= A \cdot (Q - S) & T_7 &= (B - D) \cdot (R + S) \\ T_4 &= D \cdot (R - P) \end{aligned}$$

Stačí nám tedy provést 7 násobení menších matic a 18 maticových součtů a rozdílů. Součty a rozdíly umíme počítat v čase $\Theta(n^2)$, takže časovou složitost celého algoritmu bude popisovat rekurence $T(n) = 7T(n/2) + \Theta(n^2)$. Podle kuchařkové věty je jejím řešením $T(n) = \Theta(n^{\log_2 7}) \approx \Theta(n^{2.807})$.

Pro úplnost dodejme, že jsou známy i efektivnější algoritmy, které jsou ovšem mnohem složitější a vyplatí se je používat až pro opravdu obří matice. Nejrychlejší z nich (Le Gallův z roku 2014) dosahuje složitosti cca $\Theta(n^{2.373})$ a obecně se soudí, že k $\Theta(n^2)$ se lze libovolně přiblížit.

Cvičení

1. Dokažte Strassenovy vzorce. Návod:

$$T_1 = \begin{bmatrix} + & \cdot & \cdot & + \\ \cdot & \cdot & \cdot & \cdot \\ \cdot & \cdot & \cdot & \cdot \\ + & \cdot & \cdot & + \end{bmatrix} \qquad T_4 = \begin{bmatrix} \cdot & \cdot & \cdot & \cdot \\ \cdot & \cdot & \cdot & \cdot \\ \cdot & \cdot & \cdot & \cdot \\ - & \cdot & + & \cdot \end{bmatrix} \qquad T_5 = \begin{bmatrix} \cdot & \cdot & \cdot & + \\ \cdot & \cdot & \cdot & + \\ \cdot & \cdot & \cdot & \cdot \\ \cdot & \cdot & \cdot & \cdot \end{bmatrix} \qquad T_7 = \begin{bmatrix} \cdot & \cdot & \cdot & \cdot \\ \cdot & \cdot & + & + \\ \cdot & \cdot & \cdot & \cdot \\ \cdot & \cdot & - & - \end{bmatrix}$$

$$T_1 + T_4 - T_5 + T_7 = \begin{bmatrix} + & \cdot & \cdot & \cdot \\ \cdot & \cdot & + & \cdot \\ \cdot & \cdot & \cdot & \cdot \\ \cdot & \cdot & \cdot & \cdot \end{bmatrix} = AP + BR.$$

2. *Tranzitivní uzávěr* orientovaného grafu s vrcholy $\{1, \ldots, n\}$ je nula-jedničková matice T tvaru $n \times n$, kde $T_{uv} = 1$ pravě tehdy, když v grafu existuje cesta z vrcholu u do vrcholu v. Ukažte, že umíme-li násobit matice $n \times n$ v čase $\mathcal{O}(n^\omega)$, můžeme vypočítat tranzitivní uzávěr v čase $\mathcal{O}(n^\omega \log n)$. Inspirujte se cvičením **??** z kapitoly o grafech.

1.6. Hledání k-tého nejmenšího prvku – Quickselect

Při použití metody Rozděl a panuj se někdy ukáže, že některé z částí, na které jsme vstup rozdělili, nemusíme vůbec zpracovávat. Typickým příkladem je následující algoritmus na hledání k-tého nejmenšího prvku posloupnosti.

Dostaneme-li na vstupu nějakou posloupnost prvků, jeden z nich si vybereme a budeme mu říkat *pivot.* Zadané prvky poté „rozhrneme“ na tři části: *doleva* půjdou prvky menší než pivot, *doprava* prvky větší než pivot a *uprostřed* zůstanou ty, které se pivotovi rovnají. Tyto části budeme značit po řadě L, P a S.

Kdybychom posloupnost setřídili, musí v ní vystupovat nejdříve všechny prvky z levé části, pak prvky z části střední a konečně ty z pravé. Pokud je tedy $k \leq |L|$, musí se hledaný prvek nalézat nalevo a musí tam být k-tý nejmenší (žádný prvek z jiné části ho nemohl předběhnout). Podobně je-li $|L| < k \leq |L|+|S|$, padne hledaný prvek v setříděné posloupnosti tam, kde leží S, a tedy je roven pivotovi. A konečně pro $k > |L| + |S|$ se musí nacházet v pravé části a musí tam být $(k - |L| - |S|)$-tý nejmenší.

Ze tří částí vstupu jsme si tedy vybrali jednu a v ní opět hledáme několikátý nejmenší prvek, na což samozřejmě použijeme rekurzi. Vznikne následující algoritmus, obvykle známý pod názvem Quickselect:

Algoritmus QuickSelect (hledání k-tého nejmenšího prvku)
Vstup: Posloupnost prvků $X = x_1, \ldots, x_n$ a číslo k $(1 \leq k \leq n)$

1. Pokud $n = 1$, vrátíme $y = x_1$ a skončíme.

2. $p \leftarrow$ některý z prvků $x_1, \ldots, x_n$ (pivot)
3. $L \leftarrow$ prvky v X, které jsou menší než p
4. $P \leftarrow$ prvky v X, které jsou větší než p
5. $S \leftarrow$ prvky v X, které jsou rovny p
6. Pokud $k \leq |L|$, pak $y \leftarrow \textsc{QuickSelect}(L, k)$.
7. Jinak je-li $k \leq |L| + |S|$, nastavíme $y \leftarrow p$.
8. Jinak $y \leftarrow \textsc{QuickSelect}(P, k - |L| - |S|)$.

Výstup: $y = k$-tý nejmenší prvek v X

Správnost algoritmů je evidentní, ale jak to bude s časovou složitostí? Pokaždé strávíme lineární čas rozdělováním posloupnosti a pak se zavoláme rekurzivně na menší vstup. O kolik menší bude, to závisí zejména na volbě pivota. Jestliže si ho budeme vybírat nešikovně, například jako největší prvek vstupu, skončí $n - 1$ prvků nalevo. Pokud navíc bude $k = 1$, budeme se rekurzivně volat vždy na tuto obří levou část. Ta se pak opět zmenší pouhou o jedničku a tak dále, takže celková časová složitost vyjde $\Theta(n) + \Theta(n-1) + \ldots + \Theta(1) = \Theta(n^2)$.

Obecněji pokud rozdělujeme vstup nerovnoměrně, hrozí nám, že nepřítel zvolí k tak, aby nás vždy vehnal do té větší části. Ideální obranou by tedy pochopitelně bylo volit za pivota *medián* posloupnosti.⟨4⟩ Tehdy bude nalevo i napravo nejvýše $n/2$ prvků (alespoň jeden je uprostřed) a ať už si během rekurze vybereme levou nebo pravou část, n bude exponenciálně klesat. Algoritmus pak doběhne v čase $\Theta(n) + \Theta(n/2) + \Theta(n/4) + \ldots + \Theta(1) = \Theta(n)$.

Medián ovšem není jediným pivotem, pro kterého algoritmus poběží lineárně. Zkusme za pivota zvolit *„skoromedián“* – tak budeme říkat prvku, který leží v prostředních dvou čtvrtinách setříděné posloupnosti. Tehdy bude nalevo i napravo nejvýše $3/4 \cdot n$ prvků a velikost vstupu bude opět exponenciálně klesat, byť o chlup pomaleji: $\Theta(n) + \Theta(3/4 \cdot n) + \Theta((3/4)^2 \cdot n) + \ldots + \Theta(1)$. To je opět geometrická řada se součtem $\Theta(n)$.

Ani medián, ani skoromedián bohužel neumíme rychle najít. Jakého pivota tedy v algoritmu používat? Ukazuje se, že na tom příliš nezáleží – můžeme zvolit třeba prvek $x_{\lfloor n/2 \rfloor}$ nebo si hodit kostkou (totiž pseudonáhodným generátorem) a vybrat ze všech x_i náhodně. Algoritmus pak bude mít průměrně lineární časovou složitost. Co to přesně znamená a jak to dokázat, odložíme do kapitoly **??**. V praxi tento přístup každopádně funguje výtečně.

Prozatím se spokojíme s intuitivním vysvětlením: Alespoň polovina všech prvků jsou skoromediány, takže pokud se budeme trefovat náhodně (nebo pevně, ale vstup bude „dobře zamíchaný“), často se strefíme do skoromediánu a algoritmus bude „postupovat kupředu“ dostatečně rychle.

Cvičení

1. Proč navrhujeme volit za pivota prvek $x_{\lfloor n/2 \rfloor}$, a ne třeba x_1 nebo x_n?

⟨4⟩ *Medián* je prvek, pro který platí, že nejvýše polovina prvků je menší než on a nejvýše polovina větší; tuto vlastnost má $\lfloor n/2 \rfloor$-tý a $\lceil n/2 \rceil$-tý nejmenší prvek.

2. Student Šťoura si místo skoromediánů za pivoty volí „skoroskoromediány“, které leží v prostředních šesti osminách vstupu. Jaké dosahuje časové složitosti?
3. Jak by dopadlo, kdybychom na vstupu dostali posloupnost reálných čísel a jako pivota používali aritmetický průměr?
4. Uvědomte si, že binární vyhledávání je také algoritmus typu Rozděl a panuj, v němž velikost vstupu exponenciálně klesá. Spočítejte jeho časovou složitost metodami z této kapitoly. Čím se liší od Quickselectu?

1.7. Ještě jednou třídění – Quicksort

Rozdělování vstupu podle pivota, které se osvědčilo v minulé kapitole, můžeme použít i ke třídění dat. Připomeňme, že rozdělíme-li vstup na levou, pravou a střední část, budou v setříděné posloupnosti vystupovat nejdříve prvky z levé části, pak ty z prostřední a nakonec prvky z části pravé. Můžeme tedy rekurzivně setřídit levou a pravou část (prostřední je sama od sebe setříděná), pak části poskládat ve správném pořadí a získat setříděnou posloupnost. Tomuto třídicímu algoritmu se říká *Quicksort.*⟨5⟩

Algoritmus QuickSort
Vstup: Posloupnost prvků $X = x_1, \ldots, x_n$ k setřídění

1. Pokud $n \leq 1$, vrátíme $Y = X$ a skončíme.
2. $p \leftarrow$ některý z prvků $x_1, \ldots, x_n$ (pivot)
3. $L \leftarrow$ prvky v X, které jsou menší než p
4. $P \leftarrow$ prvky v X, které jsou větší než p
5. $S \leftarrow$ prvky v X, které jsou rovny p
6. Rekurzivně setřídíme části:
7. $\quad L \leftarrow \text{QuickSort}(L)$
8. $\quad P \leftarrow \text{QuickSort}(P)$
9. Slepíme části za sebe: $Y \leftarrow L, S, P$.

Výstup: Setříděná posloupnost Y

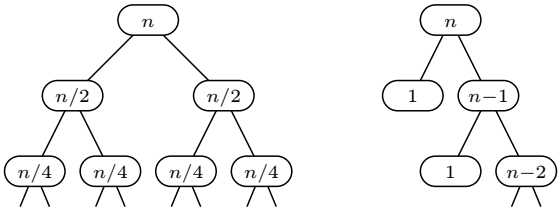
Dobrou představu o rychlosti algoritmu nám jako obvykle dá strom rekurzivních volání. V kořeni máme celý vstup, na první hladině jeho levou a pravou část, na druhé hladině levé a pravé části těchto částí, a tak dále, až v listech triviální posloupnosti délky 1. Rekurzivní volání na vstup nulové délky do stromu kreslit nebudeme a rovnou je zabudujeme do jejich otců.

⟨5⟩ Za jménem se často skrývá příběh. Quicksort (což znamená „Rychlotřidič“) přišel ke svému jménu tak, že v roce 1961, kdy vznikl, byl prvním třídicím algoritmem se složitostí $\mathcal{O}(n \log n)$ aspoň v průměru. Dodejme ještě, jeho objevitel Anthony Hoare, byl později anglickou královnou povýšen na rytíře mimo jiné za zásluhy o informatiku.

Jelikož rozkládání vstupu i skládání výsledku jistě pokaždé stihneme v lineárním čase, trávíme v každém vrcholu čas přímo úměrný velikosti příslušného podproblému. Pro libovolnou hladinu navíc platí, že podproblémy, které na ní leží, mají dohromady nejvýše n prvků – vznikly totiž rozdělením vstupu na disjunktní části a ještě se nám při tom některé prvky (pivoti) poztrácely. Na jedné hladině proto trávíme čas $\mathcal{O}(n)$.

Tvar stromu a s ním i časová složitost samozřejmě opět stojí a padají s volbou pivota. Pokud za pivoty volíme mediány nebo alespoň skoromediány, klesají velikosti podproblémů exponenciálně (na i-té hladině $\mathcal{O}((3/4)^i \cdot n)$), takže strom je vyvážený a má hloubku $\mathcal{O}(\log n)$. V součtu přes všechny hladiny proto časová složitost činí $\mathcal{O}(n \log n)$.

Jestliže naopak volíme pivoty nešťastně jako (řekněme) největší prvky vstupu, oddělí se na každé hladině od vstupu jen úsek o jednom prvku a hladin bude $\Theta(n)$. To povede na kvadratickou časovou složitost. Horší případ již nenastane, neboť na každé hladině přijdeme alespoň o prvek, který se stal pivotem.



Obr. 1.7: Quicksort při dobré a špatné volbě pivota

Podobně jako u Quickselectu, i zde je mnoho „dobrých“ pivotů, se kterými se algoritmus chová efektivně (alespoň polovina prvků jsou skoromediány). V praxi proto opět funguje spoléhat na náhodný generátor nebo dobře zamíchaný vstup. V kapitole **??** pak vypočteme, že Quicksort s náhodnou volbou pivota má časovou složitost $\mathcal{O}(n \log n)$ v průměru.

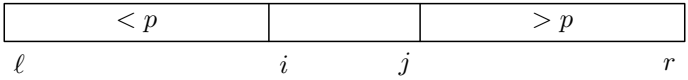
Quicksort v praxi

Závěrem si dovolme krátkou poznámku o praktických implementacích Quicksortu. Ačkoliv tento algoritmus mezi ostatními třídicími algoritmy na první pohled ničím nevyniká, u většiny překladačů se v roli standardní funkce pro třídění setkáte právě s ním. Důvodem této nezvyklé popularity není móda, nýbrž praktické zkušenosti. Dobře vyladěná implementace Quicksortu totiž na reálném počítači běží výrazně rychleji než jiné třídicí algoritmy.

Cesta od našeho poměrně obecně formulovaného algoritmu k takto propracovanému programu je samozřejmě složitá a vyžaduje mimo mistrného zvládnutí programátorského řemesla i detailní znalost konkrétního počítače. My si ukážeme alespoň první kroky této cesty.

Především Quicksort upravíme tak, aby prvky zbytečně nekopíroval. Vstup dostane jako ostatní třídicí algoritmy v poli a pak bude pouze prohazovat prvky uvnitř tohoto pole. Rekurzivně tedy budeme třídit různé úseky společného pole. Kterým úsekem se máme právě zabývat, vymezíme snadno indexy krajních prvků. Levý okraj úseku (ten blíže k začátku pole) budeme značit ℓ, pravý pak r.

Rozdělování se zjednoduší, budeme-li vstup dělit jen na dvě části namísto tří – prvky rovné pivotovi mohou bez újmy na korektnosti přijít jak nalevo, tak napravo. Budeme postupovat následovně: Použijeme dva indexy i a j. První z nich bude procházet tříděným úsekem zleva doprava a přeskakovat prvky, které mají zůstat nalevo; druhý index půjde zprava doleva a bude přeskakovat prvky patřící do pravé části. Levý index se tudíž zastaví na prvním prvku, který je vlevo, ale patří doprava; podobně pravý index se zastaví na nejbližším prvku vpravo, který patří doleva. Stačí pak tyto dva prvky prohodit a pokračovat stejným způsobem dál, až se indexy setkají.



Obr. 1.8: Quicksort s rozdělováním na místě

Nyní máme obě části uložené v souvislých úsecích pole, takže je můžeme rekurzivně setřídit. Navíc se tyto úseky vyskytují přesně tam, kde mají ležet v setříděné posloupnosti, takže „slepovací“ krok 9 původního Quicksortu můžeme zcela vynechat.

Algoritmus QuickSort2
Vstup: Pole $P[1 \ldots n]$, indexy ℓ a r úseku, který třídíme

1. Pokud $\ell \geq r$, ihned skončíme.
2. $p \leftarrow$ některý z prvků $P[\ell], \ldots, P[r]$ (pivot)
3. $i = \ell$, $j = r$
4. Dokud $i \leq j$, opakujeme:
5. Dokud $P[i] < p$, zvyšujeme i o 1.
6. Dokud $P[j] > p$, snižujeme j o 1.
7. Je-li $i < j$, prohodíme $P[i]$ a $P[j]$.
8. Je-li $i \leq j$, pak $i \leftarrow i + 1$, $j \leftarrow j - 1$.
9. Rekurzivně setřídíme části:
10. QuickSort2(P, ℓ, j)
11. QuickSort2(P, i, r)

Výstup: Úsek $P[\ell \ldots r]$ je setříděn

Popsanými úpravami jsme jistě nezhoršili časovou složitost: V krocích 2–8 zpracujeme každý prvek úseku nejvýše jednou, celkově tedy rozdělováním trávíme čas lineární s délkou úseku, s čímž naše analýza časové složitosti počítala. Naopak jsme

se zbavili zbytečného kopírování prvků do pomocné paměti. Další možná vylepšení ponecháváme čtenáři jako cvičení s nápovědou.

Cvičení

1. Rozmyslete si, že procedura QuickSort2 je korektní. Zejména si uvědomte, co se stane, když si jako pivota vybereme nejmenší nebo největší prvek úseku nebo když dokonce budou všechny prvky v úseku stejné. Ani tehdy během kroků 4–8 nemohou indexy i, j opustit tříděný úsek a každá z částí, na které se rekurzivně zavoláme, bude ostře menší než původní vstup.
2. *Vlastní zásobník:* Abyste netrávili tolik času rekurzivním voláním a předáváním parametrů, nahraďte rekurzi svým vlastním zásobníkem, na kterém si budete pamatovat začátky a konce úseků, které ještě zbývá setřídit.
3. *Šetříme pamětí:* Vylepšete postup z minulého cvičení tak, že dvojici rekurzivních volání v krocích 7 a 8 nahradíte uložením *většího* úseku na zásobník a pokračováním tříděním *menšího* úseku. Dokažte, že po této úpravě může být v libovolný okamžik na zásobníku jen $\mathcal{O}(\log n)$ úseků. Tím množství potřebné pomocné paměti kleslo z lineárního na logaritmické.
4. *Včas se zastavíme:* Podobně jako při násobení čísel (cvičení 1.3.4) se u Quicksortu hodí zastavit rekurzi předčasně (pro n menší než vhodná konstanta n_0) a přepnout na některý z kvadratických třídicích algoritmů. Vyzkoušejte si pro svůj konkrétní program najít hodnotu n_0, se kterou poběží nejrychleji.
5. *Medián ze tří:* Oblíbený trik na výběr pivota je spočítat medián z prvního, prostředního a posledního prvku úseku. Předpokládáme-li, že na vstupu dostaneme náhodnou permutaci čísel $1, \ldots, n$, jaká je pravděpodobnost, že takový pivot bude skoromediánem? S jakou pravděpodobností bude minimem?

1.8. k-tý nejmenší prvek v lineárním čase

Algoritmus Quickselect pro hledání k-tého nejmenšího prvku, který jsme potkali v kapitole 1.6, pracuje v lineárním čase pouze průměrně. Nyní ukážeme, jak ho upravit, aby tuto časovou složitost měl i v nejhorším případě. Jediné, co změníme, bude volba pivota.

Prvky si nejprve seskupíme do pětic (není-li poslední pětice úplná, doplníme ji „nekonečně velkými“ hodnotami). Poté nalezneme medián každé pětice a z těchto mediánů rekurzivním zavoláním našeho algoritmu spočítáme opět medián. Ten pak použijeme jako pivota k rozdělení vstupu na levou, střední a pravou část a pokračujeme jako v původním Quickselectu. Celý algoritmus bude vypadat takto:

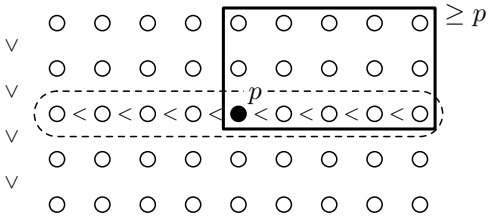
Algoritmus LinearSelect (k-tý nejmenší prvek v lineárním čase)
Vstup: Posloupnost prvků $X = x_1, \ldots, x_n$ a číslo k $(1 \leq k \leq n)$

1. Pokud $n \leq 5$, úlohu vyřešíme triviálním algoritmem.
2. Prvky rozdělíme na pětice $P_1, \ldots, P_{\lceil n/5 \rceil}$.
3. Spočítáme mediány pětic: $m_i \leftarrow$ medián P_i.

4. Najdeme pivota: $p \leftarrow \textsc{LinearSelect}(m_1, \ldots, m_{\lceil n/5 \rceil}; \lceil n/10 \rceil)$.
5. $L, P, S \leftarrow$ prvky z X, které jsou menší než p, větší než p, rovny p.
6. Pokud $k \leq |L|$, pak $y \leftarrow \textsc{LinearSelect}(L, k)$.
7. Jinak je-li $k \leq |L| + |S|$, nastavíme $y \leftarrow p$.
8. Jinak $y \leftarrow \textsc{LinearSelect}(P, k - |L| - |S|)$.

Výstup: $y = k$-tý nejmenší prvek v X

Abychom chování algoritmu pochopili, uvědomíme si nejdříve, že vybraný pivot není příliš daleko od mediánu celé posloupnosti X. K tomu nám pomůže obrázek.



Obr. 1.9: Pětice a jejich mediány

Překreslíme do něj vstup a každou pětici uspořádáme zdola nahoru. Mediány pětic tedy budou ležet v prostředním řádku. Pětice ještě přeházíme tak, aby jejich mediány rostly zleva doprava. (Pozor, algoritmus nic takového nedělá, pouze my při jeho analýze!) Navíc budeme pro jednoduchost předpokládat, že pětic je lichý počet a že všechny prvky jsou navzájem různé.

Náš pivot (medián mediánů pětic) se tedy na obrázku nachází přesně uprostřed. Mediány všech pětic, které leží napravo od něj, jsou proto větší než pivot. Všechny prvky umístěné nad nimi jsou ještě větší, takže celý obdélník, jehož levým dolním rohem je pivot, padne v našem algoritmu do části P nebo S. Počítejme, kolik obsahuje prvků: Všech pětic je $n/5$, polovina z nich ($\lceil n/10 \rceil$ pětic) zasahuje do našeho obdélníku, a to třemi prvky. To celkem dává alespoň $3/10 \cdot n$ prvků, o kterých s jistotou víme, že se neobjeví v L. Levá část proto měří nejvýše $7/10 \cdot n$.

Podobně nahlédneme, že napravo je také nejvýše $7/10 \cdot n$ prvků – stačí uvážit obdélník, který se rozprostírá od pivota doleva dolů. Všechny jeho prvky leží v L nebo S a opět jich je minimálně $3/10 \cdot n$.

Tato úvaha nám pomůže v odhadu časové složitosti:

- Rozdělovaní na pětice a počítání jejich mediánů je lineární – pětice jsou konstantně velké, takže medián jedné spočítáme sebehloupějším algoritmem za konstantní čas.
- Dělení posloupnosti na části L, P a S a rozhodování, do které z částí se vydat, trvá také $\Theta(n)$.
- Poprvé zavoláme LinearSelect rekurzivně ve 4. kroku na $n/5$ prvků.

- Podruhé ho zavoláme v kroku 6 nebo 8, a to na levou nebo pravou část vstupu. Jak už víme, každá z nich měří nejvýše $7/10 \cdot n$.

Pro časovou složitost v nejhorším případě proto dostaneme následující rekurentní rovnici (konstant jsme se zbavili vhodnou volbou jednotky času):

$$\begin{aligned} T(1) &= \mathcal{O}(1), \\ T(n) &= T(n/5) + T(7/10 \cdot n) + n. \end{aligned}$$

Metody z předchozích kapitol jsou na vyřešení této rekurence krátké (s výjimkou obecného postupu z cvičení 1.4.3). Pomůže nám válečná lest: uhodneme, že $T(n) = cn$, a ověříme dosazením, že existuje taková konstanta c, pro kterou tato funkce naši rekurenci splňuje:

$$\begin{aligned} cn &= 1/5 \cdot cn + 7/10 \cdot cn + n = \\ &= 9/10 \cdot cn + n. \end{aligned}$$

Tato rovnost platí pro $c = 10$. Náš algoritmus tedy opravdu hledá k-tý nejmenší prvek v lineárním čase.

Nyní bychom mohli upravit Quicksort, aby jako pivota používal medián spočítaný tímto algoritmem. Pak by třídil v čase $\Theta(n \log n)$ i v nejhorším případě. Příliš praktický takový algoritmus ale není. Jak asi tušíte, naše dvojitě rekurzivní hledání mediánu je sice asymptoticky lineární, ale konstanty, které v jeho složitosti vystupují, nejsou zrovna malé. Bývá proto užitečnější volit pivota náhodně a smířit se s tím, že občas promarníme jeden průchod kvůli nešikovnému pivotovi, než si třídění stále brzdit důmyslným vybíráním kvalitních pivotů.

Cvičení

1. Rozmyslete si, že našemu algoritmu nevadí, když prvky na vstupu nebudou navzájem různé.
2. Upravte funkci LinearSelect tak, aby si vystačila s konstantně velkou pomocnou pamětí. Prvky ve vstupním poli můžete libovolně přeskupovat.
3. Jak bude vypadat strom rekurzivních volání funkce LinearSelect? Kolik bude mít listů? Jak dlouhá bude nejkratší a nejdelší větev?
4. Proč při vybírání k-tého nejmenšího prvku používáme zrovna pětice? Fungoval by algoritmus s trojicemi? Nebo se sedmicemi? Byl by pak stále lineární?

5.* Na medián se můžeme dívat také tak, že je to „patník“ na půli cesty od minima k maximu. Jinými slovy, mezi minimem a mediánem leží přibližně stejně prvků jako mezi mediánem a maximem. Co kdybychom chtěli mezi minimum a maximum co nejrovnoměrněji rozmístit více patníků?

Přesněji: pro n-prvkovou množinu prvků X a číslo ε $(0 < \varepsilon < 1)$ definujeme *ε-síť* jako posloupnost $\min X = x_0 < x_1 < \ldots < x_{\lceil 1/\varepsilon \rceil} = \max X$ prvků vybraných z X tak, aby se mezi x_i a x_{i+1} vždy nacházelo nejvýše εn prvků z X. Pro

$\varepsilon = 1/2$ tedy počítáme minimum, medián a maximum, pro $\varepsilon = 1/4$ přidáme prvky ve čtvrtinách, ..., a při $\varepsilon = 1/n$ už třídíme.

Složitost hledaní ε-sítě se tedy v závislosti na hodnotě ε bude pohybovat mezi $\mathcal{O}(n)$ a $\mathcal{O}(n \log n)$. Najděte algoritmus s časovou složitostí $\mathcal{O}(n \log(1/\varepsilon))$.

6. Je dáno n-prvkové pole, ve kterém jsou za sebou dvě vzestupně setříděné posloupnosti (ne nutně stejně dlouhé). Navrhněte algoritmus, který najde medián sjednocení obou posloupností v sublineárním čase. (Lze řešit v čase $\mathcal{O}(\log n)$.)

1.9. Další cvičení

1. *Spletitý kabel:* Mějme dlouhý kabel, z jehož obou konců vystupuje po n drátech. Každý drát na levém konci je propojen s právě jedním na konci druhém a my chceme zjistit, který s kterým. K tomu můžeme používat následující operace: (1) přivést napětí na daný drát na levém konci, (2) odpojit napětí z daného drátu na levém konci, (3) změřit napětí na daném drátu na pravém konci. Navrhněte algoritmus, který pomocí těchto operací zjistí, co je s čím propojeno. Snažte se počet operací minimalizovat.

2.* Nalezněte *neadaptivní* řešení předchozího cvičení, tedy takové, v němž položené dotazy nezávisí na výsledcích předchozích dotazů.

3. Dokažte, že v předchozích dvou cvičeních je potřeba $\Omega(n \log n)$ dotazů. Pokud nevíte, jak na to, dokažte to alespoň pro neadaptivní algoritmy.

4. *Inverze matice:* Navrhněte algoritmus typu Rozděl a panuj na výpočet inverze trojúhelníkové matice $n \times n$ v čase lepším než $\Omega(n^3)$. Jako podprogram se může hodit Strassenovo násobení matic z oddílu 1.5. Můžete předpokládat, že n je mocnina dvojky.

5. *Inverze v posloupnosti* $x_1, \ldots, x_n$ říkáme každé dvojici (i, j) takové, že $i < j$ a současně $x_i > x_j$. Vymyslete algoritmus, který spočítá, kolik daná posloupnost obsahuje inverzí. To může sloužit jako míra neuspořádanosti.

6.* *Nejbližší body:* Máme n bodů v rovině a chceme najít dvojici s nejmenší vzdáleností. Nabízí se rozdělit body vodorovnou přímkou podle mediánu y-ových souřadnic, rekurzivně spočítat nejmenší vzdálenosti ε_1 a ε_2 v obou polorovinách a pak dopočítat, co se děje v pásu o šíři $2\min(\varepsilon_1, \varepsilon_2)$ podél dělící přímky. Dokažte, že probíráme-li body pásu zleva doprava, stačí každý bod porovnat s $\mathcal{O}(1)$ sousedy. To vede na algoritmus o složitosti $\Theta(n \log n)$.

7.* *Šroubky a matičky:* Na stole leží n různých šroubků a n matiček. Každá matička pasuje na právě jeden šroub a my chceme zjistit, která na který. Umíme ale pouze porovnávat šroub s matičkou – tím získáme jeden ze tří možných výsledků: matička je příliš velká, příliš malá, nebo správně velká. Nalezněte co nejefektivnější algoritmus.