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* (oni to říkali spíš latinsky: *divide et impera*). 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.

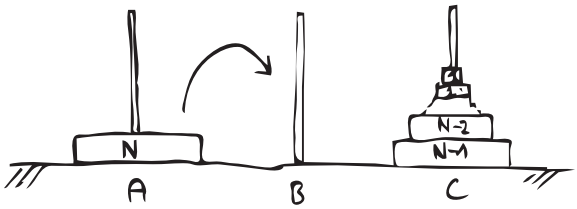
Myšlenka je to trochu bláznivá, ale často vede k překvapivě jednoduchému, rychlému a obvykle rekurzivnímu algoritmu. Postupně ji použijeme na třídění posloupností, násobení čísel i matic a hledání k-tého nejmenšího ze zadaných prvků.

Nejprve si tuto techniku ovšem vyzkoušejme 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.

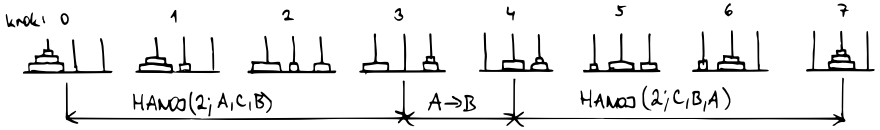


Stav hry při přenášení největšího disku

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, tedy rekurzivním zavoláním téhož algoritmu. Zastavíme se až u věže výšky 1, kterou zvládneme přemístit jedním tahem. Algoritmus bude vypadat takto:

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$(N-1; C, B, A)$.



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 1.1.1 ale snadno ukážeme, že exponenciální počet tahů je nejlepší možný. Pro $N = 64$ tedy 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, tedy 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 mezi 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, který jsme již zkoumali v jedné z předchozích kapitol.

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. Říká se mu *třídění sléváním* neboli MERGESORT.

Algoritmus MERGESORT (třídění sléváním)
Vstup: Posloupnost $a_1, \ldots, a_N$ k setřídění.
Výstup: Setříděná posloupnost $b_1, \ldots, b_N$.

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. $b_1, \ldots, b_N \leftarrow \text{MERGE}(x_1, \ldots, x_{\lfloor N/2 \rfloor}; y_1, \ldots, y_{\lceil N/2 \rceil})$

Procedura MERGE se stará o samotné slévání. To zařídíme snadno: Pokud chceme slít posloupnosti $x_1 \le x_2 \le \ldots \le x_m$ a $y_1 \le y_2 \le \ldots \le y_n$, bude výsledná posloupnost začínat menším z prvků x_1 a y_1. Tento prvek z příslušné vstupní posloupnosti přesuneme na výstup a pokračujeme stejným způsobem. Pokud to byl (řekněme) prvek x_1, zbývá nám slít $x_2, \ldots, x_m$ s $y_1, \ldots, y_n$. Dalším prvkem výstupu tedy bude minimum z x_2 a y_1. To opět přesuneme, a tak dále, než se buď x nebo y vyprázdní.

Algoritmus MERGE (slévání)
Vstup: Setříděné posloupnosti $x_1, \ldots, x_m$ a $y_1, \ldots, y_n$.
Výstup: Setříděná posloupnost $z_1, \ldots, z_{m+n}$.

1. $i \leftarrow 1$, $j \leftarrow 1$ *(zbývá slít $x_i, \ldots, x_m$ a $y_j, \ldots, y_n$)*
2. $k \leftarrow 1$ *(výsledek se objeví v $z_k, \ldots, z_{m+n}$)*
3. Dokud $i \leq m$ a $j \leq n$, opakujeme:
4. Je-li $x_i \leq y_j$, přesuneme prvek z x: $z_k \leftarrow x_i$, $i \leftarrow i + 1$.
5. Jinak přesouváme z y: $z_k \leftarrow y_j$, $j \leftarrow j + 1$.
6. $k \leftarrow k + 1$
7. Je-li $i \leq m$, zkopírujeme zbylá x: $z_k, \ldots, z_{m+n} \leftarrow x_i, \ldots, x_m$.
8. Je-li $j \leq n$, zkopírujeme zbylá y: $z_k, \ldots, z_{m+n} \leftarrow y_j, \ldots, y_n$.

Rozbor složitosti

Nyní si rozmysleme, kolik času tříděním strávíme. Slévání má lineární časovou složitost (MERGE pouze přesouvá prvky a každý přesune právě jednou). 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, a proto se nemusíme starat o zaokrouhlování. V kapitole 1.4 uvidíme, že to opravdu neuškodí.)

Jak tuto rekurentní rovnici vyřešíme? Zkusme si 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:

$$T(N) = 4 \cdot (2 \cdot T(N/8) + cN/4) + 2cN = 8 \cdot T(N/8) + 3cN.$$

Obecně po k rozepsáních:

$$T(N) = 2^k \cdot T(N/2^k) + kcN.$$

Nyní si 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 $\Theta(N \log N)$, což zní velice nadějně – vyrovná se nejrychlejšímu algoritmu z kapitoly o třídění, totiž Heapsortu, a je přitom daleko jednodušší. Jeho slabinou ale je spotřeba paměti. Zatímco Heapsort si vystačil se vstupním polem a pár pomocnými proměnnými, naše procedura MERGE sama o sobě pracuje se dvěma N-prvkovými poli.

Pojďme dokázat, že lineární množství *pomocné paměti* (to je paměť, do které nepočítáme velikost vstupu a výstupu) Mergesortu 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 stačí 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:

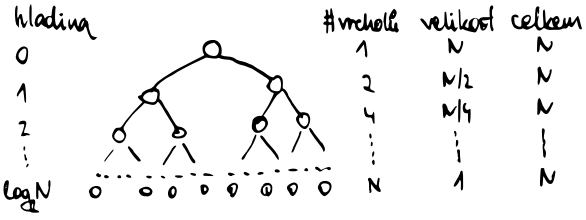
$$M(1) = 1,$$
$$M(N) = cN + M(N/2)$$

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

Výhody třídění sléváním se naplno projeví tehdy, když chceme třídit data v seznamech, ke kterým neumíme přistupovat napřeskáčku. Mergesort totiž prvky vždy prochází postupně jeden po druhém. Navíc ani nemusíme do pomocných posloupností data kopírovat, stačí vytvářet nové seznamy z původních prvků. Paměťové nároky jednoho volání funkce MERGE se proto smrsknou na konstantní a celého Mergesortu na $\Theta(\log N)$.

Stromy rekurze

Někdy je jednodušší místo počítání s rekurencemi odhadnout složitost úvahou o *stromu rekurzivních volání.* Nakresleme si 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 $\log_2 N$.



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 Mergesort bez rekurze. V k-tém průchodu si představujte, že pole je rozdělené na bloky o 2^k prvcích, a vždy slévejte dvojice sousedních bloků.
2. Naprogramujte třídění seznamu pomocí Mergesortu. Jde to snáze rekurzivně nebo cyklem?
3. Popište třídící algoritmus, který bude vstup rozkládat na více než dvě části a ty pak rekurzivně třídit. Může být rychlejší než Mergesort?

4* Naše procedura MERGE potřebuje lineární pomocnou paměť, což je nešikovné. Poměrně složitým trikem lze paměťové nároky srazit až na konstantu při zachování lineární časové složitosti. Zkuste objevit, jak je snížit alespoň na $\mathcal{O}(\sqrt{n})$.

1.3. Násobení čísel

Při třídění metodou Rozděl a panuj jsme získali algoritmus, který byl sice elegantnější než předchozí třídící 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. Rozdělme si 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 pak můžeme zapsat jako:

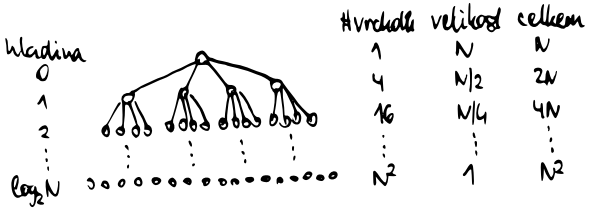
$$XY = AC \cdot 10^N + (AD + BC) \cdot 10^{N/2} + BD.$$

Spočítáme tedy rekurzivně součiny AC, AD, BC a BD a pak z nich složíme 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.



První pokus o násobení rekurzí

Po tomto neúspěchu se svého plánu ovšem 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:

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

Časová složitost se touto úpravou změní následovně:

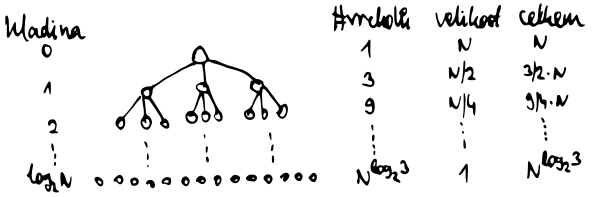
$$T(N) = 3 \cdot T(N/2) + \Theta(N).$$

Sledujme, jak se změnil strom: na i-té hladině nalezneme 3^i vrcholů s $(N/2^i)$-cifernými 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:

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

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

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



Strom rekurze algoritmu NÁSOB

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 přitom bude stačit lineárně mnoho (viz cvičení 1.3.3).

Algoritmus NÁSOB
Vstup: N-ciferná čísla X a Y
Výstup: Součin $Z = XY$

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. $Z \leftarrow P \cdot 10^N + (R - P - Q) \cdot 10^k + Q$

Zbývá dodat, že naším algoritmem vývoj neskončil a existují i asymptoticky rychlejší metody. Ty jednodušší z nich využívají podobný princip rozkladu na podproblémy, ovšem s více částmi (viz cvičení 1.3.5–1.3.6). Pokročilejší algoritmy jsou ale obvykle založeny na Fourierově transformaci. Arnold Schönhage v roce 1979 ukázal, že tímto způsobem lze dokonce dosáhnout lineární časové složitosti. Násobení je tedy, alespoň teoreticky, stejně těžké jako sčítání a odčítání. Ve cvičeních 1.3.7–1.3.9 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. Změnte 7. krok algoritmu tak, aby volal NÁSOB na $N/2$-ciferná čísla.

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 pro 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 následující podkapitoly.

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. obr

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 $2^{\Theta(k)}$-ciferými 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. Jak pomocí metody Rozděl a panuj vypsat N-ciferné číslo v soustavě o daném základu? Může se hodit výsledek předchozího cvičení.

1.4.* Složitost 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 $\Theta(N^c)$.[1] Dovolíme-li si opět zamést 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 algo-

[1] 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.

ritmu na násobení čísel, zkusme tedy výraz upravit obdobně:

$$q^{\log_b N} = \left(\frac{a}{b^c}\right)^{\log_b N} = \frac{a^{\log_b N}}{(b^c)^{\log_b N}} = \frac{b^{\log_b a \cdot \log_b N}}{b^{c \cdot \log_b N}} =$$

$$= \frac{(b^{\log_b N})^{\log_b a}}{(b^{\log_b N})^c} = \frac{N^{\log_b a}}{N^c}.$$

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

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 $T(N)$.[2] obr

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 $T(N) = T(\beta_1 N) + T(\beta_2 N) + \ldots + T(\beta_a N) + \Theta(N^c)$.

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 math

[2] 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 $g(n) = \Theta(z(n))$.

1969 ukázal pan 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 si je na čtvrtiny (budeme jim říkat *bloky*): matici X na A až D, matici Y na P až S, všechny formátu $N/2 \times N/2$. 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. Letmým nahlédnutím do kuchařkové věty z minulé kapitoly zjistíme, že tak získáme opět kubický algoritmus.⟨3⟩

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:

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

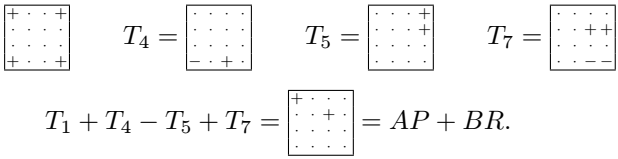
Stačí nám tedy provést 7 násobení menších matic a 18 maticových součtů a rozdílů. Součty a rozdíly umíme počítat v čase $\Theta(N^2)$, takže časovou složitost celého algoritmu bude popisovat rekurence $T(N) = 7T(N/2) + \Theta(N^2)$. Podle kuchařkové věty je jejím řešením $T(N) = \Theta(N^{\log_2 7}) \approx \Theta(N^{2,807})$.

Pro úplnost dodejme, že jsou známy i efektivnější algoritmy, které jsou ovšem mnohem složitější a které se vyplatí používat až pro opravdu obří matice. Nejrychlejší z nich (Coppersmithův-Winogradův) dosahuje složitosti $\Theta(N^{2,376})$ a mnozí se domnívají, že k $\Theta(N^2)$ se lze libovolně přiblížit.

⟨3⟩ Obvyklá terminologie je tu poněkud zavádějící – N zde neznačí velikost vstupu, nýbrž počet řádků matice; vstup je tedy velký N^2.

Cvičení:

1. Dokažte Strassenovy vzorce. Návod:



2. Rychlé násobení matic je základem rychlých algoritmů pro další operace z lineární algebry. Vymyslete algoritmus pro výpočet inverze trojúhelníkové matice.

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
Vstup: Posloupnost prvků $X = x_1, \ldots, x_N$ a číslo k $(1 \leq k \leq N)$.
Výstup: $y = k$-tý nejmenší prvek v X.

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 $y \leftarrow$ QuickSelect(L, k).
7. Jinak je-li $k \leq |L| + |S|$, nastavíme $y \leftarrow p$.
8. Jinak $y \leftarrow$ QuickSelect$(P, k - |L| - |S|)$.

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 *„lžimedián“* – tak budeme říkat prvku, který leží v prostředních dvou čtvrtinách setříděné posloupnosti. Tehdy bude nalevo i napravo nejvýše $3/4\cdot N$ 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 lžimedián 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 *lineární časovou složitost v průměrném případě.* Co to přesně znamená a jak to dokázat, odložíme do kapitoly 1.9. V praxi tento přístup každopádně funguje výtečně.

Prozatím se spokojíme s intuitivním vysvětlením: Alespoň polovina všech prvků jsou lžimediány, takže pokud se budeme trefovat náhodně (nebo pevně, ale vstup bude „dobře zamíchaný“), často se strefíme do lžimediánu 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?
2. Student Šťoura si místo lžimediánů za pivoty volí „ještě lživější mediány“, 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?

[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.

1.7. Ještě jednou třídění – Quicksort

Rozdělování vstupu podle pivota, které se osvědčilo v minulé kapitole, můžeme použít i ke třídění dat. Připomeňme si, 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 třídícímu algoritmu se říká *Quicksort.*⟨5⟩

Algoritmus QuickSort
Vstup: Posloupnost prvků $X = x_1, \ldots, x_N$ k setřídění.
Výstup: Setříděná posloupnost Y.

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: $Y \leftarrow L, S, P$.

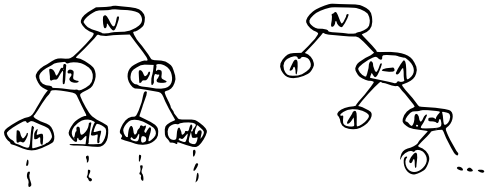
Dobrou představu o rychlosti algoritmu nám jako obvykle dá strom rekurzivních volání. V kořeni máme celý vstup, na první hladině jeho levou a pravou část, na druhé hladině levé a pravé části těchto částí, a tak dále, až v listech triviální posloupnosti délky 1. Rekurzivní volání na vstup nulové délky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

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 lžimediány, 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 $\mathcal{O}(N \log N)$.

⟨5⟩ Za jménem se často skrývá příběh. Quicksort (což znamená „Rychlotřidič“) přišel ke svému jménu tak, že v roce 1961, kdy vznikl, byl prvním třídícím algoritmem se složitostí $\mathcal{O}(N \log N)$ aspoň v průměrném případě.

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.



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 lžimediány). V praxi proto opět funguje spoléhat na náhodný generátor nebo dobře zamíchaný vstup. V kapitole 1.9 pak vypočteme, že Quicksort s náhodnou volbou pivota má časovou složitost $\mathcal{O}(N \log N)$ v průměrném případě.

Quicksort v praxi

Závěrem si dovolme krátkou poznámku o praktických implementacích Quicksortu. Ačkoliv tento algoritmus mezi ostatními třídícími 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 třídící 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 třídící algoritmy v poli a pak bude pouze prvky uvnitř tohoto pole prohazovat. 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 nám 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 tedy tyto dva prvky prohodit a pokračovat stejným způsobem dál, až se indexy setkají.

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. Fix!

Algoritmus QuickSort2
Vstup: Pole $P[1 \ldots N]$, indexy ℓ a r úseku, který třídíme.
Výstup: Úsek $P[\ell \ldots r]$ je setříděn.

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)

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:* Abychom netrávili tolik času rekurzivním voláním a předáváním parametrů, nahradíme rekurzi naším vlastním zásobníkem, na kterém si budeme pamatovat začátky a konce úseků, které nám ještě zbývá setřídit.
3. *Šetříme pamětí:* Vylepšíme postup z minulého cvičení tak, že dvojici rekurzivních volání v krocích 7 a 8 nahradíme 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, což je také celkové množství pomocné paměti, které algoritmus spotřebuje.

Fix: Obrázek situace.

4. *Včas se zastavíme:* Podobně jako při násobení čísel (cvičení 1.3.4) se i u Quicksortu hodí zastavit rekurzi předčasně (pro N menší než vhodná konstanta N_0) a přepnout na některý z kvadratických třídících algoritmů. Vyzkoušejte si najít hodnotu N_0, pro kterou algoritmus běží 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 lžimediánem?

1.8. k-tý nejmenší prvek v lineárním čase

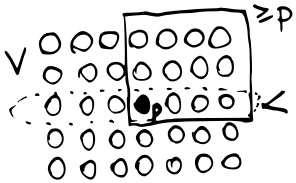
Algoritmus Quickselect pro hledání k-tého nejmenšího prvku, který jsme potkali v kapitole 1.6, pracuje v lineárním čase pouze v průměrném případě. Nyní si ukážeme, jak ho upravit, aby tuto časovou složitost měl vždy. Jediné, co změníme, bude volba pivota.

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
Vstup: Posloupnost prvků $X = x_1, \ldots, x_N$ a číslo k ($1 \leq k \leq N$).
Výstup: $y = k$-tý nejmenší prvek v X.

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 $y \leftarrow \textsc{LinearSelect}(P, k - |L| - |S|)$.

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



Pětice a jejich mediány

Překreslíme si do něj vstup a každou pětici uspořádáme zdola nahoru. Mediány pětic tedy budou ležet v prostředním řádku. Pětice si 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 voláme LinearSelect rekurzivně ve 4. kroku na $N/5$ prvků.
- Podruhé ho voláme 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 si 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 použil vždy 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.
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 zavislosti na hodnotě ε bude pohybovat mezi $\mathcal{O}(n)$ a $\mathcal{O}(n \log n)$. Najděte algoritmus s časovou složitostí $\mathcal{O}(n \log(1/\varepsilon))$.

1.9.* Pravděpodobnostní algoritmy

U algoritmů založených na výběru pivota (Quickselect a Quicksort) jsme spoléhali na to, že pokud budeme pivota volit náhodně, bude se algoritmus „chovat dobře.“ V této kapitole využijeme základní aparát teorie pravděpodobnosti k tomu, abychom těmto tvrzením dali smysl. math

TODO: Někde v knize bychom se měli zmínit o náhodných číslech a pravděpodobnostních algoritmech. Také by mělo padnout, že je zásadní rozdíl mezi průměrnou worst-case složitostí pp. algoritmu a složitostí deterministického algoritmu průměrovanou přes všechny vstupy. Že je mezi tím nějaký vztah (Yaův princip), to už je bezpečně mimo náš dosah. Jak se náhodná čísla generují?

Tak dlouho se chodí se džbánem ...

Začněme jednoduchým příkladem: budeme chodit se džbánem pro vodu tak

dlouho, než se utrhne ucho, což při každém pokusu nastane náhodně s pravděpodobností p ($0 < p < 1$), nezávisle na výsledcích předchozích pokusů. Kolik pokusů v průměru podnikneme?

Označme si T počet kroků, po kterém k utržení ucha dojde. To je nějaká náhodná veličina a nás bude zajímat její střední hodnota $\mathbb{E}[T]$. Spočítáme ji jednoduchým trikem (jak jinak než rekurzivním): V každém případě provedeme první pokus. Pokud se ucho utrhne (což nastane s pravděpodobností p), hra končí. Pokud se neutrhne (pravděpodobnost $1-p$), dostaneme se do přesně stejné situace, jako předtím – náš ideální džbán totiž nemá žádnou paměť. Z toho vyjde následující rovnice pro $\mathbb{E}[T]$:

$$\mathbb{E}[T] = 1 + p \cdot 0 + (1-p) \cdot \mathbb{E}[T].$$

Vyřešíme-li, dostaneme $\mathbb{E}[T] = 1/p$. Tento výsledek se nám bude často hodit, formulujme si ho proto jako lemma:

Lemma: *(O džbánu)* Čekáme-li na náhodný jev, který nastane s pravděpodobností p, dočkáme se ve střední hodnotě po $1/p$ pokusech.

Mediány, lžimediány a Quickselect

Úvaha o džbánu nám dává jednoduchý algoritmus, pomocí kterého umíme najít lžimedián posloupnosti N prvků: Vybereme si *rovnoměrně náhodně* jeden z prvků posloupnosti; tím *rovnoměrně* myslíme tak, aby všechny prvky měly stejnou pravděpodobnost. Pak ověříme, jestli jsme si vybrali lžimedián. Pokud ne, na vše zapomeneme a postup opakujeme.

Kolik pokusů budeme potřebovat, než algoritmus skončí? Lžimediány tvoří alespoň polovinu prvků, tedy pravděpodobnost, že se do nějakého strefíme, je minimálně 1/2. Podle lemmatu o džbánu tedy střední hodnota počtu pokusů bude nejvýše 2. (Počet pokusů v nejhorším případě ovšem neumíme omezit nijak – při dostatečné smůle můžeme stále vybírat nejmenší prvek. Že se to stane, má ale nulovou pravděpodobnost.)

Nyní už snadno spočítáme časovou složitost našeho algoritmu. Jeden pokus trvá $\Theta(N)$, střední hodnota počtu pokusů je $\mathcal{O}(1)$, takže střední hodnota časové složitosti je $\Theta(N)$. Obvykle budeme zkráceně mluvit o *průměrné* časové složitosti. term

Pokud tento výpočet lžimediánu použijeme v Quickselectu, získáme algoritmus pro hledání k-tého nejmenšího prvku s průměrnou složitostí $\Theta(N)$. Dobrá, tím jsme se ale šalamounsky vyhnuli otázce, jakou průměrnou složitost má původní Quickselect s rovnoměrně náhodnou volbou pivota.

Tu odhadneme snadno: Rozdělíme si běh algoritmu na *fáze*. Fáze bude končit v okamžiku, kdy si za pivota zvolíme lžimedián. Fáze se skládá z *kroků* spočívajících v náhodné volbě pivota, lineárně dlouhém výpočtu a zahození části vstupu. Už víme, že lžimedián se průměrně podaří najít za dva kroky, tudíž jedna fáze trvá v průměru lineárně dlouho. Navíc si všimneme, že během každé fáze se vstup zmenší alespoň o čtvrtinu. K tomu totiž stačil samotný poslední krok, ostatní kroky mohou situaci jedině zlepšit. Průměrnou složitost celého algoritmu pak vyjádříme jako součet průměrných složitostí jednotlivých fází: $\Theta(N)+\Theta((3/4)\cdot N)+\Theta((3/4)^2\cdot N)+\ldots = \Theta(N)$.

Indikátory a Quicksort

Podíváme-li se na výpočet v minulém odstavci s odstupem, všimneme si, že se opírá zejména o *linearitu střední hodnoty* – ta říká, že $\mathbb{E}[X+Y] = \mathbb{E}[X] + \mathbb{E}[Y]$. Časovou složitost celého algoritmu jsme vyjádřili jako součet složitostí fází. Přitom fázi jsme si nadefinovali tak, aby se již chovala dostatečně průhledně.

Podobně můžeme analyzovat i Quicksort, jen se nám bude hodit složitost rozložit na daleko více veličin. Mimo to si všimneme, že Quicksort na každé porovnání provede jen $\mathcal{O}(1)$ dalších operací, takže postačí odhadnout počet provedených porovnání.

Očíslujeme si prvky podle pořadí v setříděné posloupnosti $y_1, \ldots, y_N$. Zavedeme náhodné veličiny C_{ij} pro $1 \leq i < j \leq N$ tak, aby platilo $C_{ij} = 1$ právě tehdy, když během výpočtu došlo k porovnání y_i s y_j. V opačném případě je $C_{ij} = 0$. Proměnným, které nabývají hodnoty 0 nebo 1 podle toho, zda nějaká událost nastala, se obvykle říká *indikátory.* Počet všech porovnání je tudíž roven součtu všech indikátorů C_{ij}.

Zamysleme se nyní nad tím, kdy může být $C_{ij} = 1$. Algoritmus porovnává pouze s pivotem, takže jedna z hodnot y_i, y_j se těsně předtím musí stát pivotem. Navíc všechny hodnoty $y_{i+1}, \ldots, y_{j-1}$ se ještě pivoty stát nesměly, jelikož jinak by y_i a y_j už byly rozděleny v různých částech posloupnosti. Jinými slovy, C_{ij} je rovno jedné právě tehdy, když se z hodnot $y_i, y_{i+1}, \ldots, y_j$ stane jako první pivotem buď y_i nebo y_j. A poněvadž pivota vybíráme rovnoměrně náhodně, má každý z prvků $y_i, \ldots, y_j$ stejnou pravděpodobnost, že se stane pivotem jako první, totiž $1/(j-i+1)$. Proto $C_{ij} = 1$ nastane s pravděpodobností $2/(j-i+1)$.

Nyní si stačí uvědomit, že když indikátory nabývají pouze hodnot 0 a 1, je jejich střední hodnota rovna právě pravděpodobnosti jedničky, tedy také $2/(j-i+1)$. Sečtením přes všechny dvojice (i,j) pak dostaneme pro počet všech porovnání:

$$\mathbb{E}[C] = \sum_{1 \leq i < j \leq N} \frac{2}{j-i+1} \leq N \cdot \sum_{2 \leq d \leq N} \frac{1}{d}.$$

Nerovnost na pravé straně platí díky tomu, že rozdíly $j-i+1$ se nacházejí v intervalu $\langle 2, N \rangle$ a každým rozdílem přispěje nejvýše N různých dvojic (i,j). Poslední suma je tzv. harmonická suma, jejíž hodnota je $\ln N + \mathcal{O}(1)$. math

Spočítali jsme tedy, že střední hodnota časové složitosti Quicksortu s rovnoměrně náhodnou volbou pivota je $\mathcal{O}(N \log N)$.

Chování na náhodném vstupu

Když jsme poprvé přemýšleli o tom, jak volit pivota, všimli jsme si, že pokud volíme pivota pevně, náš algoritmus není odolný proti zlomyslnému uživateli. Takový uživatel může na vstupu zadat šikovně sestrojenou posloupnost, která algoritmus donutí vybrat si v každém kroku pivota nešikovně, takže poběží kvadraticky dlouho. Tomu jsme se přirozeně vyhnuli náhodnou volbou pivota – pro sebezlotřilejší vstup

doběhneme v průměru rychle. Hodí se ale také vědět, že i pro pevnou volbu pivota je špatných vstupů málo.

Zavedeme si proto ještě jeden druh složitosti algoritmů, tentokrát opět deterministických (bez náhodného generátoru). Bude to *složitost v průměru přes vstupy.* Jinými slovy algoritmus bude mít pevný průběh, ale budeme mu dávat náhodný vstup a počítat, jak dlouho v průměru poběží.

Co to ale takový náhodný vstup je? U našich dvou problémů to docela dobře vystihuje *náhodná permutace* – vybereme si rovnoměrně náhodně jednu z $N!$ permutací množiny $\{1, 2, \ldots, N\}$.

Jak Quicksort, tak Quickselect se pak budou chovat velmi podobně, jako když měly pevný vstup, ale volily náhodně pivota. Pokud je na vstupu rovnoměrně náhodná permutace, je její prostřední prvek rovnoměrně náhodně vybrané číslo z množiny $\{1, 2, \ldots, N\}$. Vybereme-li si ho za pivota a rozdělíme vstup na levou a pravou část, obě části budou opět náhodné permutace, takže se na nich algoritmus bude opět chovat tímto způsobem. Můžeme tedy analýzu z této kapitoly použít i na tento druh průměru se stejným výsledkem.

Cvičení:

1. *Ideální mince:* Mějme počítač, jehož náhodným generátorem je ideální mince. Jinými slovy, máme funkci `random`, ze které na každé zavolání vypadne jeden rovnoměrně náhodný bit vygenerovaný nezávisle na předchozích bitech. Jak pomocí takové funkce generovat rovnoměrně náhodná přirozená čísla od 1 do N? Minimalizujte průměrný počet hodů mincí.
2. Ukažte, že v předchozím cvičení nelze počet hodů mincí v nejhorším případě nijak omezit, leda že by N bylo mocninou dvojky.
3. *Míchání karet:* Popište algoritmus, který v lineárním čase vygeneruje náhodnou permutaci množiny $\{1, 2, \ldots, N\}$.
4. V mnoha programovacích jazycích je k dispozici funkci `random`, která nám vrátí rovnoměrně (pseudo)náhodné číslo z pevně daného intervalu. Lidé ji často používají pro generování čísel v rozsahu od 0 do $N-1$ tak, že spočtou `random` mod N. Jaký se v tom skrývá háček?
5. *Náhodná k-tice:* Na vstupu je číslo $k > 1$ a posloupnost navzájem různých přirozených čísel ukončená nulou. Vymyslete algoritmus, který z této posloupnosti vybere rovnoměrně náhodně k-tici čísel. Vstup může být tak dlouhý, že se celý nevejde do paměti; k-tice se tam spolehlivě vejde.
6* *Míchání podruhé:* Vasil Vasiljevič míchá karty takto: připraví si N prázdných přihrádek. Pak postupně umisťuje čísla $1, \ldots, N$ do přihrádek tak, že vždy vybere rovnoměrně náhodně přihrádku a pokud v ní již něco je, vybírá znovu. Kolik pokusů bude v průměru potřebovat?
7. Lemma o džbánu můžeme dokázat i přímo podle definice střední hodnoty: Pravděpodobnost, že ucho se poprvé utrhne při i-tém pokusu, je rovna $p(1-p)^{i-1}$. Proto $\mathbb{E}[T] = \sum_{i=1}^{\infty} ip(1-p)^{i-1} = p/(1-p) \cdot \sum_{i=1}^{\infty} i(1-p)^i$. Podobnou sumu jsme ovšem již potkali při analýze stavění haldy zespodu. Sečtěte ji. odkaz

8.* Průměrnou časovou složitost Quicksortu lze spočítat i podobnou úvahou, jakou jsme použili u Quickselectu. Jak?

9.* Ještě jeden způsob, jak analyzovat průměrnou složitost Quicksortu, je použítím podobné úvahy jako v důkazu Lemmatu o džbánu. Sestavíme rekurenci pro průměrný počet porovnání: $R(n) = n + \frac{1}{n}\sum_{i=1}^{n}(R(i-1) + R(n-i))$, $R(0) = R(1) = 0$. Dokažte indukcí, že $R(n) \leq 4n \ln n$.