Kapitola 1

Rozděl a panuj

Rozděl a panuj je programovací metoda. Často se označuje latinsky *"Divide et Empera"* nebo anglicky *"Divide and Conquer"*. Vychází z toho, že umíme zadaný problém rozložit na menší podproblémy stejného typu.

Na začátku rozdělíme práci a určíme, které podproblémy je potřeba vyřešit. Tyto podproblémy si necháme vyřešit rekurzivně. Složením jejich řešení dostaneme řešení původního problému. Algoritmus se volá rekurzivně tak dlouho, dokud se nedostane k problémům konstantní velikosti, které už umí hravě vyřešit.

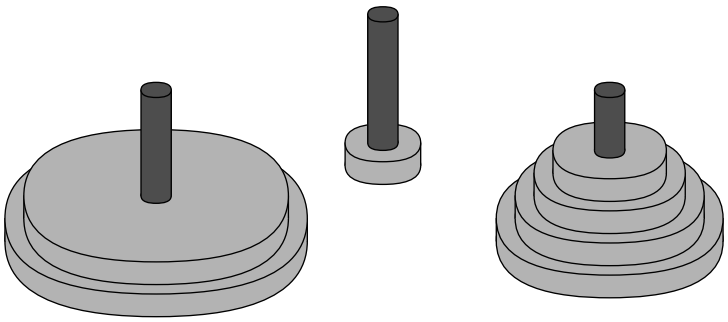
Vysvětlíme si to ještě jednou na pohádce o tom, jak si římský císař hraje na popelku. Inu i římskému císaři se někdy stane, že mu v sýpkách smíchají čočku a hrách. Jak to roztřídit? Císař přeci nemůže dělat takovou práci sám, má na to lidi. A tak si zavolá vojáky a nechá směs čočky a hrachu rozdělit na tři zhruba stejně velké hromádky. Potom zavolá své generály, každému přiřadí jednu hromádku a nechá je oddělit čočku od hrachu. Ale i takový generál se nebude dřít sám. Po vzoru císaře, rozdělí svojí hromádku na tři a rozdělí práci mezi své podřízené. Takto projde předávání práce celou hierarchií římské armády, až to skončí u otroků. Každý otrok dostane jednu misku směsi a tu roztřídí. Jeho pán si od všech svých otroků vybere zvlášť čočku a zvlášť hrách a předá je výše. Takto projde slučování hromádek zpátky až k císaři, který, ke své spokojenosti a rozmarnosti, nechá přesunout všechnu čočku zpátky do sýpky a hrách hodí sviním, aby se příště s čočkou nesmíchal.

Použití metody rozděl a panuj si ukážeme v následujících úlohách.

1.1 Hanojské věže

Příběh: V jednom indickém chrámu mají tři věže – Věž zrození, Věž života a Věž zkázy. Uvnitř každé z nich je kůl, na kterém je navlečeno několik zlatých disků. Ve všech třech věžích je dohromady 64 disků. Každý disk je jinak široký. Disky smí být na kůlu navlečeny pouze tak, že menší a užší disk leží na širším. Jinak to prý přinese smůlu. Proto každý kůl spolu s disky vypadá jako kužel.

Při stvoření chrámu byly všechny disky navlečeny na jeden kůl ve Věži stvoření. Kněží, kteří v chrámu přebývají, každý den přesunou jeden disk (víc jich přesunout nesmí) tak, aby se jim co nejdříve podařilo přesunout všechny disky na kůl ve Věži zkázy. Jinak by se zastavilo plynutí života. Stará legenda říká, že až se jim to podaří, tak nastane konec světa.



Úkol pro Vás: Dokázali byste napsat program, který kněžím vypíše instrukce, jak mají disky přesunovat? Už jsou z toho celí zmatení a moc by jim to pomohlo. Na začátku jsou všechny disky navlečeny na první kůl a máte je přestěhovat na poslední kůl.

Pokud se budeme držet zásady rozděl a panuj, tak zkusíme problém rozložit na podproblémy, jejichž vyřešení přehodíme na někoho jiného (třeba na rekurzi). Abychom mohli největší disk přesunout na správné místo, tak nejprve necháme všechny menší disky přestěhovat na třetí odkládací tyčku. Pak přesuneme největší disk. Na závěr necháme všechny odložené disky přesunout nad největší disk.

```
1: Hanoj(n, odkud, pres, kam)
2:     if n ≥ 1 then
3:         Hanoj(n − 1, odkud, kam, pres)
4:         Vypiš("Přeneste disk z  ",odkud," do  ", kam)
5:         Hanoj(n − 1, pres, odkud, kam)
```

Jaká je časová složitost tohoto algoritmu? Ta je úměrná tomu, kolikrát budou muset kněží přenést nějaký disk. Označme celkový počet přenesení disků pomocí $T(n)$. Z uvedeného algoritmu je vidět rekurence $T(n) = 2T(n-1) + 1$. Rekurenci můžeme vyřešit například postupným rozepisováním. Dostaneme

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

Takže časová složitost algoritmu Hanoj je exponenciální. Legenda říká, že disků je 64. Jestliže byl chrám založen zhruba před 3000 lety, dokážete určit kdy nastane konec světa? Má smysl se toho obávat?

1.2 Mergesort

Mergesort je třídící algoritmus využívající metody rozděl a panuj. Dostane vstup, například posloupnost čísel, který rozdělí na dvě skoro stejné části. Ty nechá setřídit rekurzivně. Výsledné posloupnosti slije do jedné a tu vrátí jako setříděnou posloupnost.

Co znamená slít dvě setříděné posloupnosti P_1, P_2 do jedné? Cílem je vytvořit jednu posloupnost P, která obsahuje prvky obou předchozích posloupností a je setříděná. Menší prvek z počátečních (a nejmenších) prvků setříděných posloupností P_1, P_2 je určitě nejmenším prvkem výsledné posloupnosti P. Proto ho odtrhneme a přidáme na začátek P. Tím z P_1, P_2 vzniknou posloupnosti P_1', P_2'. Pro ty můžeme celý postup zopakovat a tím dostaneme druhý nejmenší prvek P. Dalším opakováním postupně odtrháme všechny prvky obou setříděných posloupností P_1, P_2 a sestavujeme požadovanou posloupnost P.

Slévání si můžeme představovat trochu jako zip na oblečení. Dvě ozubené části zipu (setříděné posloupnosti) přiložíme k sobě a zapneme je jezdcem. Jezdec kontroluje, že do sebe ozubené části zapadají podle velikosti.

Klasická implementace Mergesortu pracuje se spojovým seznamem prvků. Pokud chceme mergesort implementovat v poli, tak se neobejdeme bez pomocného pole $\mathtt{B}[\cdot]$, kam budeme ukládat mezivýsledky. Procedura Mergesort(l, p) setřídí úsek pole $\mathtt{A}[l..p]$ mezi indexy l a p. Pole $\mathtt{A}[n]$ i pomocné pole $\mathtt{B}[n]$ jsou globální proměnné.

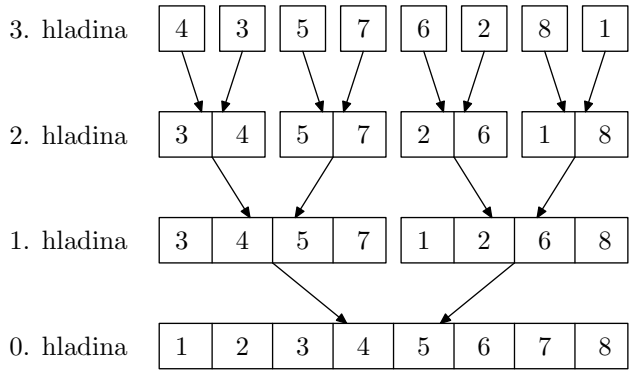
```
1: Mergesort(l, p)
2:     if l < p then
3:         stred := ⌊(l + p)/2⌋
4:         Mergesort(l, stred)
5:         Mergesort(stred + 1, p)
6:         Merge(l,p)
```

Procedura Merge(l, p) slije setříděné podposloupnosti $\mathtt{A}[l..stred]$ a $\mathtt{A}[stred + 1..p]$ do jedné setříděné posloupnosti $\mathtt{A}[l..p]$. Pro ukládání mezivýsledků potřebuje pomocné pole $\mathtt{B}[l..p]$. Procedura Merge má časovou složitost $\mathcal{O}(m)$, kde m je délka výsledné posloupnosti.

```
1: Merge(l, p)
2:     stred := ⌊(l + p)/2⌋
3:     i := l              {Index na začátek první setříděné posloupnosti. }
4:     j := stred + 1      {Index na začátek druhé setříděné posloupnosti.}
5:     k := l              {Index do pole B[ · ], kam zapisujeme výsledek.}
6:     while k ≤ p do
7:         if i > stred then                {První posloupnost došla.}
8:             B[k] := A[j]
9:             j := j + 1
10:        else if j > p then               {Druhá posloupnost došla.}
11:            B[k] := A[i]
12:            i := i + 1
13:        else if A[i] < A[j] then         {Žádná posloupnost nedošla.}
14:            B[k] := A[i]
15:            i := i + 1
16:        else
17:            B[k] := A[j]
18:            j := j + 1
19:        k := k + 1
20:    for k := l to p do {Kopírování výsledků z B[l..p] do A[l..p]}
21:        A[k] := B[i]
```

Časová složitost Mergesort je určena rekurencí $T(n) = 2T(n/2) + \mathcal{O}(n)$, protože dvakrát řešíme podúlohu poloviční velikosti a jednou sléváme výsledky dohromady.

Následující obrázek ilustruje průběh algoritmu a slévání jednotlivých částí. Strom rekurze máme nakreslený vzhůru nohama, protože se algoritmus nejprve zanoří, co nejhlouběji to jde, a teprve při zpáteční cestě provádí slévání podposloupností (úseků pole). Pro lidi je přirozenější ta zpáteční cesta, na které se něco děje.



Strom rekurze má hloubku nejvýše $\lceil \log n \rceil$, protože při každém zavolání Mergesort rozdělíme posloupnost v půlce. Na k-té hladině[1] se slévají dvě posloupnosti délky $n/2^{k+1}$ do posloupnosti délky $n/2^k$. Těchto slévání se na k-té hladině vykoná 2^k. Proto je celková práce na k-té hladině rovna $\mathcal{O}(n)$. Z toho dostáváme časovou složitost algoritmu mergesort $\mathcal{O}(n \log n)$.

Poznámka k implementaci: Na konci každého volání Merge kopírujeme výsledky z pomocného pole `B`$[\cdot]$ do pole `A`$[\cdot]$. Tomu se můžeme vyhnout tak, že budeme při rekurzivním volání Mergesortu střídat význam pole `A`$[\,\cdot\,]$ a `B`$[\,\cdot\,]$. Na sudých hladinách dostane Merge posloupnost v poli `B`$[\,\cdot\,]$ a vytvoří setříděnou posloupnost do pole `A`$[\,\cdot\,]$ a v lichých hladinách naopak. Prohazování můžeme zrealizovat tak, že funkci Merge předáme i ukazatele na tyto pole a při rekurzivním zavolání ukazatele prohodíme.

Mohlo by se stát, že střídání polí po hladinách vyjde tak, že budeme v poslední hladině chtít slévat položky z `B`$[\,\cdot\,]$ do pole `A`$[\,\cdot\,]$. Proto na začátku celého algoritmu nakopírujeme celý vstup z `A`$[\,\cdot\,]$ i do pole `B`$[\,\cdot\,]$.

1.3 Medián posloupnosti

Máme posloupnost n prvků, například čísel. *Medián* je takové číslo z posloupnosti čísel, že 50% čísel je větších nebo rovno mediánu, ale také 50% čísel je menších nebo rovno mediánu. Jinými slovy, když si posloupnost setřídíme, tak nazveme prvek, který leží uprostřed, *mediánem.* Pokud je prvků lichý počet, tak je medián ten prostřední prvek. Pokud je počet prvků sudý, tak máme mediány dva – spodní a horní medián. Abychom si zjednodušili výklad, budeme za medián považovat spodní medián.[2]

Medián se používá ve statistice, protože to je jedno číslo, které vypovídá něco typického o množině čísel. Podobným číslem je průměr.[3]

Poznámka: Kolik informace nám přinese údaj o průměrné mzdě v Heské republice? Pro běžné lidi je to jen číslo, kvůli kterému jsou nespokojeni, že nevydělávají dost. Pokud bude 10% obyvatel vydělávat 110tis. Kč/měsíc a zbylých 90% obyvatel jen 10tis. Kč/měsíc, tak je průměrná mzda v Heské republice 20tis. Kč. Ovšem kdo ji má? Naproti tomu medián platů v Heské republice je 10tis. Kč/měsíc. To je číslo, které mnohem lépe popisuje celou situaci.

[1] k-tou hladinou myslíme k-tou hladinu od kořene. Na obrázku je strom nakreslen vzhůru nohama a proto je to na obrázku k-tá hladina od zdola.

[2] Běžně se za medián bere průměr spodního a horního mediánu. To je ale hodnota, ne prvek.

[3] Víte o tom, že máte nadprůměrný počet očí? Tedy pokud nejste ten výjimečný případ.

Úkol: Dosteneme pole obsahující n různých čísel.[4] Jak co nejrychleji najít medián? Nebo když tento problém zobecníme, jak co nejrychleji najít k-tý nejmenší prvek?[5]

Řešení 1: Nejjednodušším řešením je, že si pole setřídíme a potom vypíšeme k-tý prvek. Třídění nám zabere čas $\mathcal{O}(n \log n)$.

Řešení 2: Můžeme upravit Quicksort ze strany **??** tak, aby po rozdělení úseku pole $\mathtt{A}[l..p]$ pivotem nezpracovával levý i pravý úsek pole, ale aby už pracoval jen s úsekem, ve kterém leží k-tý nejmenší prvek.

```
1:  QuickSelect(l, p, k)
2:      if l < p then
3:          pivot := A[⌊(l + p)/2⌋]
4:          q :=Partition(l, r, pivot)
5:          if q ≤ k then
6:              QuickSelect(l, q, k)
7:          else
8:              QuickSelect(q + 1, p, k)
9:      else
10:         Output(A[l])
```

Připomeňme, že funkce q :=Partition(l, p, $pivot$) rozdělí úsek pole $\mathtt{A}[l..p]$ na dva úseky $\mathtt{A}[l..q]$ a $\mathtt{A}[q+1..p]$, kde první úsek obsahuje pouze prvky menší nebo rovny pivotu a druhý úsek pouze prvky větší než pivot. Implementace funkce je popsána u quicksortu na straně **??**.

V ideálním případě, kdy se nám podaří vybrat všechny pivoty tak, aby se každý úsek pole rozdělil přesně napůl, bude časová složitost tohoto řešení $\mathcal{O}(n + n/2 + n/4 + \dots) = \mathcal{O}(2n) = \mathcal{O}(n)$. Ovšem pokud budeme mít smůlu a pivot bude vždy nejmenším nebo největším prvkem z aktuálního úseku, tak bude časová složitost tohoto řešení $\mathcal{O}(n + (n-1) + (n-2) + \cdots + 1) = \mathcal{O}(n^2)$. Naštěstí, stejně jako u quicksortu, v průměrném případě nastane první varianta a časová složitost bude lineární.

Řešení 3: Předchozí řešení vylepšíme tak, že si v každé iteraci vybereme dobrého pivota, který leží dostatečně daleko od okrajů aktuálního úseku. Chceme, aby v úseku pole $\mathtt{A}[l..p]$ délky n vždy existovalo aspoň $3n/10$ prvků menších než pivot a $3n/10$ prvků větších než pivot. Pivota vybereme následovně:

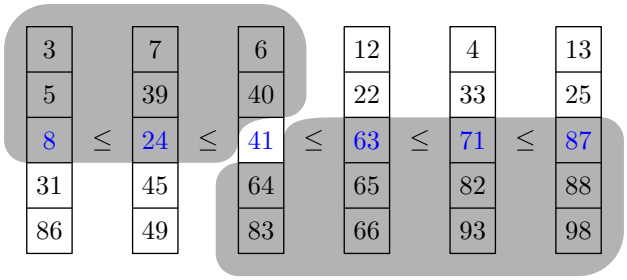
1. Prvky pole rozdělíme do pětic a v každé pětici nalezneme medián.

2. Vezmeme pouze mediány ze všech pětic a rekurzivně pomocí QuickSelect() na nich najdeme medián.

Podívejme se na následující obrázek. V každé pětici jsou hodnoty setříděné podle velikosti a sloupce s pěticemi jsou seřazeny podle hodnot jejich mediánů. Algoritmus nepotřebuje nic třídit, pouze nalezne v každé pětici medián (znázorněn modře) a potom nalezne medián mediánů (na tomto obrázku má hodnotu 41).

[4] Proč různých čísel? Pro různá čísla na vstupu nebudeme muset diskutovat okrajové případy u výpočtu mediánu přes pětice. Zjednoduší se tím výklad, ale algoritmus funguje i pro čísla, která nemusí být různá.

[5] U řady problémů, které se řeší pomocí metody Rozděl a panuj, je jednodušší vyřešit zobecněný problém, než hledat přímé řešení problému.



Všechny prvky, které leží v první šedé oblasti jsou menší než medián mediánů a všechny prvky ve druhé šedé oblasti jsou větší než medián mediánů. Prvků větších než medián mediánů je aspoň $3 \cdot n/10 - 1 \geq 3n/10 - 1$. Podobně prvků menších než medián mediánů je aspoň $3n/10 - 1$. (Protože n nemusí být dělitelné 5, může být v poslední pětici jen 1 prvek. Proto musíme počítat pečlivěji než podle obrázku).

Analýza časové složitosti: Nechť $T(n)$ označuje časovou složitost algoritmu QuickSelect. Nalezení mediána mediánů bude trvat $\mathcal{O}(n) + T(\lceil n/5 \rceil)$. Rekurzivní volání QuickSelect() na podúseku pole proběhne v čase $\mathcal{O}(n) + T(7n/10 + 1)$. Celkem dostáváme $T(n) = T(7n/10 + 1) + T(\lceil n/5 \rceil) + \mathcal{O}(n)$. Teď už stačí jen vyřešit tuto rekurenci.

Zkusíme hledat řešení rekurence ve tvaru $T(n) = cn$, pro nějakou zatím neznámou konstantu c. Dosazením do rekurence dostaneme $T(n) \leq c(7n/10 + 1) + c(\lceil n/5 \rceil) + an \leq c(9n/10 + 2) + an = cn - (cn/10 - 2c - an)$. Pokud bude $(cn/10 - 2c - an) \leq 0$, tak máme vyhráno. Tato podmínka je ekvivalentní s $c \leq 10a(n/(n - 20))$. Stačí zvolit $c = 10a$ a pro $n > 20$ bude řešení $T(n) = cn$ platit. Pro $n \leq 20$ je $T(n) = \mathcal{O}(1)$.

1.4 Master theorem, řešení rekurencí

Při určování časové složitosti algoritmu založeného na metodě Rozděl a panuj se typicky dostaneme k rekurenci, kterou potřebujeme vyřešit. Abychom ji nemuseli zdlouhavě rozepisovat a přemýšlet, jak ji vyřešit, naučíme se následující univerzální metodu.

Věta 1 (Master theorem) *Nechť $T : \mathbb{N} \to \mathbb{N}$ je funkce splňující*

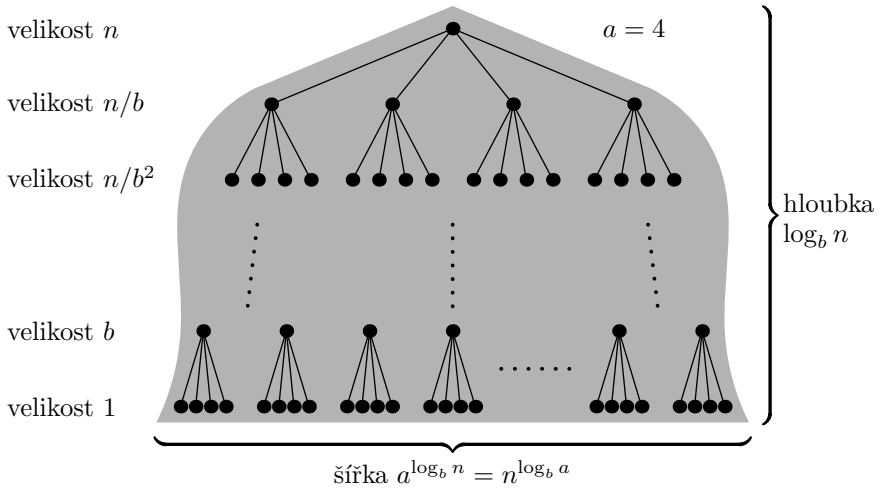
$$T(n) = aT(\lceil n/b \rceil) + \mathcal{O}(n^d)$$

pro nějaké konstanty $a > 0$, $b > 0$ a $d \geq 0$. Potom

$$T(n) = \begin{cases} \mathcal{O}(n^d) & \text{pokud } d > \log_b a \\ \mathcal{O}(n^d \log n) & \text{pokud } d = \log_b a \\ \mathcal{O}(n^{\log_b a}) & \text{pokud } d < \log_b a \end{cases}$$

Důkaz: Pro jednoduchost předpokládejme, že n je mocnina b. Díky je $\lceil n/b \rceil = n/b$.

Velikost podproblémů se při každém rekurzivním zanoření zmenší b krát. Proto se po nejvýše $\log_b n$ zanořeních zmenší až na konstantu (základní případ). Z toho dostáváme, že výška stromu rekurze je nejvýše $\log_b n$.



Při každém rekurzivním zavolání se rozvětvíme na a podproblémů. Proto bude na k-té hladině stromu rekurze a^k podproblémů, každý o velikosti n/b^k. Všechna práce prováděná na k-té hladině trvá

$$a^k \cdot \mathcal{O}(\left(\frac{n}{b^k}\right)^d) = \mathcal{O}(n^d) \cdot \left(\frac{a}{b^d}\right)^k.$$

Časy strávené v hladinách 0 až $\log_b n$ tvoří geometrickou posloupnost s kvocientem a/b^d. Celková časová složitost $T(n) = \mathcal{O}(n^d) \cdot \sum_{k=0}^{\log_b n} \left(\frac{a}{b^d}\right)^k$. Podle hodnoty kvocientu nastanou následující tři případy:

- $a/b^d < 1$. Posloupnost je klesající. Největší je první člen $\mathcal{O}(n^d)$. Součet prvních $\log_b n$ členů geometrické posloupnosti je nejvýše kvocient-krát větší než první člen. Proto $T(n) = \mathcal{O}(n^d)$.
- $a/b^d = 1$. Všechny členy posloupnosti mají stejnou velikost a to $\mathcal{O}(n^d)$. Proto $T(n) = \mathcal{O}(n^d) \cdot \log_b n$.
- $a/b^d > 1$. Posloupnost je rostoucí. Součet prvních $\log_b n$ členů geometrické posloupnosti je nejvýše kvocient-krát větší než poslední a největší člen. Jeho hodnota je $\mathcal{O}(n^d \left(a/b^d\right)^{\log_b n})$.

  $$n^d \left(\frac{a}{b^d}\right)^{\log_b n} = n^d \left(\frac{a^{\log_b n}}{(b^{\log_b n})^d}\right) = a^{\log_b n} = a^{(\log_a n)(\log_b a)} = n^{\log_b a}.$$

  Proto $T(n) = \mathcal{O}(n^{log_b a})$.

Tyto případy přesně odpovídají rozdělení případů z věty.

Zbývá ukázat, že věta platí i pro n, které není mocninou b. Už víme, že věta platí pro n, která jsou mocninou b. Toho využijeme. Nechť n_+ je nejbližší mocnina b větší než n. Podívejme se na první případ, kdy má řešení rekurence tvar $T(n) = \mathcal{O}(n^d)$. Ostaní případy si odůvodníte analogicky.

Funkce $T(n)$ je rostoucí a proto $T(n) < T(n_+)$. Z nerovnosti $n_+ < bn$ plyne, že $\mathcal{O}(n_+^d) = \mathcal{O}((bn)^d) = \mathcal{O}(n^d)$. Poskládáním nerovností dostaneme $T(n) < T(n_+) = \mathcal{O}(n_+^d) = \mathcal{O}(n^d)$. Proto $T(n) = \mathcal{O}(n^d)$. ∎

Jak se dá Master Theorem použít v algoritmech z předchozích sekcí knihy, u kterých už jsme časovou složitost spočítali jinak? Časová složitost binárního vyhledávání (vyhledávání půlením intervalu) je určena rekurencí $T(n) = T(n/2) + \mathcal{O}(1)$.

Pro její vyřešení použijeme Master Theorem s konstantami $a = 1$, $b = 2$, $d = 0$ a dostaneme $T(n) = \mathcal{O}(\log n)$. Časová složitost mergesortu je určena rekurencí $T(n) = 2T(n/2) + \mathcal{O}(n)$. Pro její vyřešení použijeme Master Theorem s konstantami $a = 2$, $b = 2$, $d = 1$ a dostaneme $T(n) = \mathcal{O}(n \log n)$.

Příklad: Vyřešte rekurenci $T(n) = 2T(\sqrt{n}) + \log n$.

Tato rekurence vypadá jako ošklivá čarodějnice, ale můžeme z ní udělat Popelku, když vhodně převlékneme proměnné. Nejprve použijeme substituci proměnné $m = \log n$ a dostaneme $T(2^m) = 2T(2^{m/2}) + m$. Pak použijeme substituci funkce $S(m) = T(2^m)$ a dostaneme $S(m) = 2S(m/2) + m$. Tuhle krásku už umíme vyřešit například pomocí Master Theoremu. $S(m) = \mathcal{O}(m \log m)$. Změnou $S(m)$ zpět na $T(n)$ dostaneme $T(n) = T(2^m) = S(m) = \mathcal{O}(\log n \log \log n)$. (Všechny substituce jsou korektní, protože jsme použili rostoucí funkce.)

1.5 Příklady

1. (Ruční třídění karet)

   Dostanete balíček zamíchaných karet. Jakým způsobem ho setřídíte? Dokážete něco říci o tomto algoritmu? Jakou bude mít časovou složitost? Jaké budou jeho nároky na prostor (paměť)?

2. (Hledání pevného bodu zobrazení)

   Dostaneme setříděné pole různých čísel $\mathtt{A}[1, \ldots, n]$. Chceme zjistit, jestli existuje index i, pro který $\mathtt{A}[i] = i$. Pomocí metody rozděl a panuj navrhněte algoritmus běžící v čase $\mathcal{O}(\log n)$.

3. (Většinový prvek)

   Dostaneme pole n prvků $\mathtt{A}[1, \ldots, n]$. Prvky nemusíme nutně umět porovnávat,[6] ale pro každé dva indexy i, j umíme zjistit, jestli $\mathtt{A}[i] = \mathtt{A}[j]$. Prvek má v poli většinu, pokud se vyskytuje na více než polovině políček. Navrhněte algoritmus, který zjistí, jestli pole obsahuje prvek mající většinu a případně ho vypíše.

   (a) Ukažte, jak vyřešit tento problém v čase $\mathcal{O}(n \log n)$.
   Nápověda: Rozdělte pole na dvě pole poloviční velikosti a zkuste v nich najít prvek mající většinu.

   (b) Najděte algoritmus běžící v lineálním čase.
   Nápověda: Prvky libovolně spárujte. Dostanete $n/2$ párů. Pro každý pár zjistěte, jestli jsou prvky stejné. Pokud ano, tak nechte jeden z nich a druhý prvek smažte. Pokud jsou prvky různé, tak smažte oba. Takto získáme nejvýše $n/2$ prvků. Ukažte, že nově získané prvky mají většinový prvek právě tehdy, když ho má i původních n prvků.

4. (Házení vajíčka z mrakodrapu)

   Na návštěvě Singapuru jste dostali jako dárek na uvítanou "nerozbitelné vajíčko". Tvrdí Vám, že je vyrobeno nejmodernější technologií a i když má jen tuhou skořápku, tak prý nejde rozbít. Rozhodnete se to vyzkoušet a proto budete vajíčko házet z mrakodrapu na chodník.[7]

[6] Prvky mohou odpovídat obrázkům či jiným souborům. Místo dlouhého souboru si stačí pamatovat hodnotu "dlouhé" hašovací funkce. Existují hašovací funkce (například MD5), které nám s velmi vysokou pravděpodobností zaručí, že každý obrázek dostene jinou hodnotu hašovací funkce.

[7] Singapur má jedny z nejvyšších mrakodrapů světa. Než to ale budete realizovat, tak si pořádně prostudujte, co zakazují přísné singapurské zákony. Pokud byste z mrakodrapu plivali, tak vás každé plivnutí vyjde na $1000.

Mrakodrap má n pater. Jaké je nejnižší patro, ze kterého už se vajíčko rozbije? (Klidně se může stát, že se vajíčko nerozbije ani z n-ého patra.) Kolik nejméně pokusů budete muset provést, aby jste to zjistili? Pokusem myslíme jedno hození vajíčka na chodník.

(a) Máte jen jedno vajíčko.

(b) A co když budete mít dvě, naprosto stejná vajíčka?

(c) Jak to zobecnit pro k vajíček?

Nejprve zkuste najít co nejlepší horní odhad na počet pokusů. Teprve pak se zamyslete nad tím, jak dokázat, že to lépe nejde.

Nápověda: Formulka pro počet pokusů není jednoduchá. Stačí, když ukážete, jak spočítat počet pokusů pro n-patrový mrakodrap například na počítači.

5. (Procvičení řešení rekurencí)

Vyřešte následující rekurence. Řešení určete s přesností Θ-notace.

(a) $T(n) = 2T(n/3) + 1$

(b) $T(n) = 5T(n/4) + n$

(c) $T(n) = 7T(n/7) + n$

(d) $T(n) = 9T(n/3) + n^2$

(e) $T(n) = 8T(n/2) + n^3$

(f) $T(n) = 49T(n/25) + n^{3/2} \log n$

(g) $T(n) = 2T(n/2) + n/\log n$

(h) $T(n) = 16T(n/4) + n!$

(i) $T(n) = T(n-1) + 2$

(j) $T(n) = T(n-1) + n^c$, kde $c \geq 1$ je konstanta

(k) $T(n) = T(n-1) + c^n$, kde $c > 1$ je konstanta

(l) $T(n) = 2T(n-1) + 1$

(m) $T(n) = T(\sqrt{n}) + 1$

(n) $T(n) = \sqrt{n}T(\sqrt{n}) + 1$

Nápověda: (k předposlední rekurenci) Zkuste použít substituci $S(m) = T(2^m)$. Substituce je korektní, protože 2^m je rostoucí funkce.

Literatura