Obsah

1 Úvod a motivace

algoritmus = způsob řešení problému

datová struktura = způsob uložení informací

techniky návrhu a analýzy algoritmů důkaz korektnosti algoritmu, analýza složitosti algoritmu, asymptotická notace, rozděl a panuj, rekurze

datové struktury halda, prioritní fronta, vyhledávací stromy, červeno-černé stromy, hašovací tabulky

algoritmy řazení – vkládáním, rozdělováním, slučováním, haldou, v lineárním čase

Úvod a definice

- **Intuice: Algoritmus** je jakýkoli dobře definovaný výpočetní postup, který bere jednu nebo více hodnot jako **vstup** a produkuje jednu nebo více hodnot jako **výstup** v konečném čase.
  - Algoritmus je posloupnost výpočetních kroků, které transformují vstup na výstup.
  - Myšlenka za programem.
  - Lze si jej představit jako nástroj pro řešení **dobře specifikovaných výpočetních problémů**.
- Formálně v jiném předmětu – Vyčíslitelnost.

- **Příklad:** Řadicí problém
  - **Vstup:** Posloupnost n čísel $\langle a_1, a_2, \ldots, a_n \rangle$.
  - **Výstup:** Permutace posloupnosti vstupu $\langle a'_1, a'_2, \ldots, a'_n \rangle$ taková, že $a'_1 \leq a'_2 \leq \cdots \leq a'_n$.

- **Problém:** Obecná specifikace vstupu a požadovaného výstupu (např. řadicí problém).
- **Instance problému:** Konkrétní vstup splňující specifikaci problému (např. konkrétní posloupnost čísel $\langle 5, 2, 8, 1 \rangle$).
- Algoritmy jsou **technologie**, která je všude kolem nás (internet, bankovnictví, navigace, bioinformatika, atd.).
- Důležité je rozumět, jak algoritmy fungují, a umět je **navrhovat a analyzovat**.
- **Efektivita:** Dva různé algoritmy pro stejný problém se mohou dramaticky lišit ve výkonu.
- Příklad: Problém řazení milionu čísel.
  - Rychlý algoritmus: Sekundy.
  - Pomalý algoritmus: Mnoho dní nebo let.
- Rychlejší počítače pomáhají, ale **efektivní algoritmy jsou důležitější**.
- **Příklad:**
  - Počítač A (super rychlý): 10^{10} instrukcí/sekundu.

  - Počítač B (super pomalý): 10^7 instrukcí/sekundu (1000x pomalejší než A).
  - **Algoritmus 1** (kvadratický, pomalý): $2n^2$ instrukcí.
  - **Algoritmus 2** (logaritmicko-lineární, rychlý): $50n \log_2 n$ instrukcí.
- **Pro $n = 10^7$ (10 milionů prvků):**
  - **Počítač A + Alg. 1:** trvá $2 \times (10^7)^2/10^{10} = 2 \times 10^{14}/10^{10} = 2 \times 10^4$ sekund ≈ 5.5 hodin.
  - **Počítač B + Alg. 2:** trvá $50 \times 10^7 \log_2(10^7)/10^7 \approx 50 \times 23.2 \approx 1160$ sekund ≈ 19 minut.
- **Závěr:** I 1000x pomalejší počítač s efektivnějším algoritmem je dramaticky rychlejší než rychlejší počítač s neefektivním algoritmem.

Algoritmy jako technologie

- Moderní počítače jsou neskutečně rychlé. Proč tedy studovat efektivní algoritmy?
- **Velikost vstupů:** Problémy se dnes často týkají obrovských datových sad (Big Data).
- Rychlejší hardware dokáže zrychlit i pomalý algoritmus, ale jen do určité míry.
- **Asymptotická složitost:** Jak se chování algoritmu mění s rostoucí velikostí vstupu.
- **Klíčová role:** Pokrok v technologiích často *závisí* na pokroku v algoritmech a jejich efektivitě.
- **Základ AI:** Algoritmy tvoří fundamentální stavební kameny umělé inteligence (AI).
- **Strojové učení (Machine Learning - ML):** Podoblast AI, kde se počítače učí z dat.
  - Klasické algoritmy: Řeší jasně definovaný problém krok za krokem (např. řazení).
  - ML: Řeší problém učením se z dat, jejich výstupem je často model, který pak dělá predikce nebo rozhodnutí.
  - Úspěch ML se v současnosti týká především problémů, u kterých lidé neví, co je ten pravý algoritmus.
- **Příklady použití ML algoritmů:**

  - Rozpoznávání obrazu a řeči.
  - Doporučovací systémy (filmy, produkty).
  - Zpracování přirozeného jazyka (překlady).
- **Data Science:** Multidisciplinární obor, který využívá vědecké metody ze statistiky, informatiky a optimalizace k extrahování poznatků z dat.
- **Velká data (Big Data):** S nástupem velkých dat se efektivní algoritmy staly ještě kritičtějšími pro jejich zpracování v rozumném čase.
- **Algoritmy jsou centrální** pro většinu moderních počítačových technologií.
- Schopnost **navrhovat a analyzovat efektivní algoritmy** je jednou z nejdůležitějších dovedností v informatice.
- S neustále rostoucími kapacitami počítačů je používáme k řešení větších problémů než kdykoli předtím. Právě u větších velikostí problémů se rozdíly v efektivitě mezi algoritmy stávají **obzvláště výraznými**.
- Solidní základ algoritmických znalostí a technik je jedna z charakteristik, která **definuje skutečně zručného programátora**.
- *S moderní výpočetní technologií můžete splnit některé úkoly, aniž byste toho o algoritmech mnoho věděli, ale s dobrým základem v algoritmech dokážete **mnohem víc**.*

Návrh a analýza algoritmů

- **Porozumění problému:** Co je vstup, co je požadovaný výstup, jaká jsou omezení.
- **Výběr vhodné datové struktury:** Jak efektivně ukládat a manipulovat s daty.
- **Vytvoření strategie:** Jak přistoupit k řešení (např. rozděl a panuj, dynamické programování, hladový algoritmus).
- **Korektnost:** Důkaz, že algoritmus vždy produkuje správný výstup pro jakýkoli platný vstup.
- **Časová složitost:** Kolik základních operací algoritmus provede v závislosti na velikosti vstupu.
- **Prostorová složitost:** Kolik paměti algoritmus potřebuje.
- **Růstová funkce:** Popisuje, jak se čas nebo prostor zvětšuje s rostoucí velikostí vstupu (n). Používá se asymptotická notace (např. $O(n)$, $O(n \log n)$, $O(n^2)$).

Proč asymptotická složitost?
Dovoluje nám porovnávat algoritmy nezávisle na konkrétní implementaci, programovacím jazyce nebo hardwaru. Zaměřujeme se na chování pro velké vstupy.

- Obvykle **jednoprocesorový model s náhodným přístupem k paměti (RAM)**.
- V modelu RAM se instrukce provádějí **jedna po druhé**, žádné souběžné operace.
- Algoritmy jsou implementovány jako počítačové programy.
- Každá instrukce trvá **stejnou dobu** jako jakákoli jiná instrukce.
- Každý přístup k datům (použití hodnoty proměnné nebo uložení do proměnné) trvá **stejnou dobu** jako jakýkoli jiný přístup k datům.
- Instrukce trvají konstantní čas (sčítání, porovnání, čtení z paměti).
- Tato abstrakce nám umožňuje analyzovat algoritmy bez ohledu na detaily konkrétního procesoru.
- Striktně vzato bychom měli přesně definovat instrukce modelu RAM a jejich cenu.
- To by však bylo **únavné a poskytlo by málo vhledu** do návrhu a analýzy algoritmů.
- Musíme být opatrní, abychom model RAM **nezneužívali** (např. předpokládat instrukci, která řadí v jednom kroku).
- Naším vodítkem je, **jak jsou navrženy skutečné počítače**.
- Model RAM obsahuje instrukce běžně se vyskytující ve skutečných počítačích:
  - **Aritmetické** (např. sčítání, odčítání, násobení, dělení, zbytek, dolní/horní celá část).
  - **Pohyb dat** (načtení, uložení, kopírování).
  - **Řídicí** (podmíněný a nepodmíněný skok, volání a návrat z podprogramu).
- Datové typy v modelu RAM jsou **celočíselné, s plovoucí desetinnou čárkou** (pro uložení aproximací reálných čísel) a **znaky**.
- Předpokládáme, že každé slovo dat má **omezení na počet bitů**.
- Například, při práci se vstupy o velikosti n typicky předpokládáme, že celá čísla jsou reprezentována $c(\lfloor \log_2 n \rfloor + 1)$ bity pro nějakou konstantu $c \geq 1$.

  - Požadujeme $c \geq 1$, aby každé slovo mohlo obsahovat hodnotu n (umožňuje indexování).
  - Omezujeme c na konstantu, aby velikost slova nerostla libovolně (nerealistický scénář, kdy bychom mohli ukládat obrovské množství dat v jednom slově a operovat s nimi v konstantním čase).

- Skutečné počítače obsahují instrukce, které nejsou výše uvedeny – ty představují tzv. **šedou zónu v modelu RAM**.
- **Příklad umocňování (x^n):**
  - V obecném případě ne konstantní čas.
  - Pokud je x přesná mocnina 2 (např. 2^n), může být obvykle považována za konstantní čas (např. posunem bitů o n pozic doleva pro 2^n).
- Pokusíme se těmto šedým zónám v modelu RAM vyhnout.
- **Paměťová hierarchie:** Model RAM nezohledňuje paměťovou hierarchii (cache, virtuální paměť), která je běžná v současných počítačích.
  - Jiné výpočetní modely se snaží zohlednit efekty paměťové hierarchie, které jsou někdy významné.
  - Naše analýzy neberou paměťovou hierarchii v úvahu, protože modely, které ji zahrnují, jsou složitější a analýzy modelu RAM jsou obvykle **vynikajícími prediktory výkonu na skutečných strojích**.
- Insertion Sort (Řazení vkládáním)
- Analýza Insertion Sortu
- Metoda Rozděl a panuj (Divide-and-Conquer)
- Merge Sort (Řazení slučováním)
- Analýza Merge Sortu: Rekurzivní vztahy a Master Theorem

Definice 1.1 (Problém řazení)**.**

- **Vstup:** Posloupnost n čísel $\langle a_1, a_2, \ldots, a_n \rangle$.
- **Výstup:** Permutace vstupní posloupnosti $\langle a'_1, a'_2, \ldots, a'_n \rangle$ taková, že $a'_1 \leq a'_2 \leq \ldots \leq a'_n$.
- Čísla mohou být například celá čísla, reálná čísla nebo jakékoli objekty, které lze porovnávat.
- Číslům, která se mají řadit, se říká **klíče**.

- Ačkoliv se problém koncepčně týká řazení posloupnosti, vstup je **pole s n prvky**.
- Klíče jsou **spojené s jinými daty**, která nazýváme **satelitní data**.
- Dohromady tvoří klíč a satelitní data **záznam (record)**.
- **Příklad:** Záznamy studentů s přidruženými daty: věk, studijní průměr a počet abs. kurzů.
  - Jakákoli z těchto veličin může být klíčem.
  - Když se tabulka řadí, přesune se celý **přidružený záznam** (satelitní data) společně s klíčem.
- Při popisu řadicího algoritmu se zaměřujeme na klíče, ale je důležité si pamatovat, že **obvykle existují přidružená satelitní data**.
- Jeden z nejjednodušších algoritmů řazení.
- **Princip:** Funguje jako řazení karet v ruce. Vezmeme jednu kartu (prvek) a vložíme ji na správné místo do již seřazené sady karet.
- V každém kroku 'j' udržuje podpole $A[1 \ldots j-1]$ seřazené.
- Postupně vkládá $A[j]$ do seřazené části $A[1 \ldots j-1]$.

Výhody a nevýhody

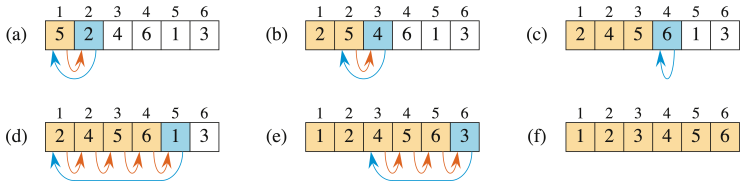
- **Výhody:** Jednoduchá implementace, efektivní pro malé vstupy, efektivní pro téměř seřazená pole.
- **Nevýhody:** Pomalý pro velké, náhodné vstupy.

INSERTION-SORT(A, n)

```
1  for i = 2 to n
2      key = A[i]
3      // Insert A[i] into the sorted subarray A[1 : i − 1].
4      j = i − 1
5      while j > 0 and A[j] > key
6          A[j + 1] = A[j]
7          j = j − 1
8      A[j + 1] = key
```

- Vstupem jsou dva parametry:
  - Pole `A` obsahující hodnoty k řazení.
  - Počet hodnot `n` k řazení.

- Hodnoty v poli zabírají pozice `A[1]` až `A[n]`, což značíme jako `A[1:n]`.



- Počáteční pole: $[5, \underline{2}, 4, 6, 1, 3]$ (**2** je *klíč*)
- $2 < 5$, posuň **5** doprava: $[2, 5, \underline{4}, 6, 1, 3]$ (**4** je *klíč*)
- $4 < 5$, posuň **5** doprava; $4 > 2$: $[2, 4, 5, \underline{6}, 1, 3]$ (**6** je *klíč*)
- …

2 Časová složitost

časová složitost **výpočtu** je součet cen všech vykonaných operací
časová složitost **algoritmu** je funkce délky vstupu

- složitost v *nejhorším* případě maximální délka výpočtu na vstupu délky n
- složitost v *nejlepším* případě minimální délka výpočtu na vstupu délky n
- *průměrná* složitost průměr složitostí výpočtů na všech vstupech délky n

složitost = časová složitost v nejhorším případě

- Analyzujeme počet elementárních operací (porovnání, přiřazení).
- Každý řádek kódu má určitou "cenu" provedení.
- t_i označuje počet opakování while cyklu pro danou hodnotu i

INSERTION-SORT(A, n)	*cost*	*times*
1 **for** $i = 2$ **to** n	c_1	n
2 $key = A[i]$	c_2	$n-1$
3 // Insert $A[i]$ into the sorted subarray $A[1:i-1]$.	0	$n-1$
4 $j = i-1$	c_4	$n-1$
5 **while** $j > 0$ and $A[j] > key$	c_5	$\sum_{i=2}^{n} t_i$
6 $A[j+1] = A[j]$	c_6	$\sum_{i=2}^{n}(t_i - 1)$
7 $j = j-1$	c_7	$\sum_{i=2}^{n}(t_i - 1)$
8 $A[j+1] = key$	c_8	$n-1$

$$T(n) = c_1 n + c_2(n-1) + c_4(n-1) + c_5 \sum_{i=2}^{n} t_i + c_6 \sum_{i=2}^{n} (t_i - 1) + c_7 \sum_{i=2}^{n} (t_i - 1) + c_8(n-1)$$

- Pole je již seřazeno.
- Vnitřní 'while' cyklus se pro každé j provede jen jednou (první porovnání selže).

$$\begin{aligned} T(n) &= c_1 n + c_2(n-1) + c_4(n-1) + c_5 \sum_{j=2}^{n} t_j \\ &\quad + c_6 \sum_{j=2}^{n} (t_j - 1) + c_7 \sum_{j=2}^{n} (t_j - 1) + c_8(n-1) \\ &= c_1 n + c_2(n-1) + c_4(n-1) + c_5(n-1) + c_8(n-1) \\ &= an + b \end{aligned}$$

lineární složitost

- Složitost: $\Theta(n)$ – později
- Pole je seřazeno v opačném pořadí (sestupně).

$$\begin{aligned} T(n) &= c_1 n + c_2(n-1) + c_4(n-1) + c_5 \left(\frac{n(n+1)}{2} - 1 \right) \\ &\quad + c_6 \left(\frac{n(n-1)}{2} \right) + c_7 \left(\frac{n(n-1)}{2} \right) + c_8(n-1) \\ &= an^2 + bn + c \end{aligned}$$

kvadratická složitost

- Složitost: $\Theta(n^2)$ – později

Závěr
Insertion Sort je efektivní pro malé vstupy nebo téměř seřazená pole, ale pro velké náhodné vstupy je neefektivní.

3 Korektnost algoritmů

vstupní podmínka – ze všech možných vstupů pro daný algoritmus vymezuje ty, pro které je algoritmus definován

výstupní podmínka – pro každý vstup daného algoritmu splňující vstupní podmínku určuje, jak má vypadat výsledek odpovídající danému vstupu

algoritmus je (totálně) korektní jestliže pro každý vstup splňující vstupní podmínku výpočet skončí a výsledek splňuje výstupní podmínku

úplnost (konvergence) – pro každý vstup splňující vstupní podmínku výpočet skončí

částečná (parciální) korektnost – pro každý vstup, který splňuje vstupní podmínku a výpočet na něm skončí, výstup splňuje výstupní podmínku

analyzujeme efekt jednotlivých operací

analýza efektu cyklu

- u vnořených cyklů začínáme od cyklu nejhlubší úrovně
- pro každý cyklus určíme jeho invariant
- *invariantem cyklu* je takové tvrzení, které platí před vykonáním a po vykonání každé iterace cyklu
- dokážeme, že invariant cyklu je pravdivý
- využitím invariantu
  - dokážeme konečnost výpočtu cyklu
  - dokážeme efekt cyklu

Inicializace: invariant je platný před začátkem vykonávání cyklu

Iterace: jestliže invariant platí před iterací cyklu, zůstává v platnosti i po vykonání iterace

Ukončení: cyklus skončí a po jeho ukončení garantuje platný invariant požadovaný efekt cyklu

INSERTION-SORT(A, n)

```
1  for i = 2 to n
2      key = A[i]
3      // Insert A[i] into the sorted subarray A[1 : i − 1].
4      j = i − 1
5      while j > 0 and A[j] > key
6          A[j + 1] = A[j]
7          j = j − 1
8      A[j + 1] = key
```

- Používáme **invariant cyklu** pro ‘for‘ cyklus:

- **Invariant:** Na začátku každé iterace ‘for‘ cyklu (pro index i) obsahuje pole $A[1:i-1]$ prvky původního pole $A[1:i-1]$ v seřazeném pořadí.

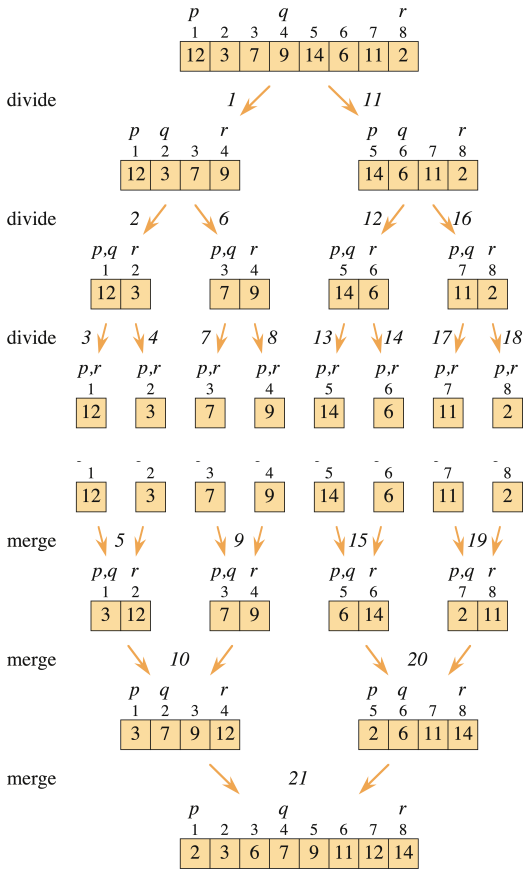
1. **Inicializace ($i = 2$):** Podpole $A[1]$ je vždy seřazené. Invariant platí.
2. **Induktivní krok:** Předpokládejme, že invariant platí pro i. Iterace algoritmu vezme $A[i]$ a vloží ho na správné místo do již seřazeného pole $A[1:i-1]$. Tím vznikne seřazené pole $A[1:i]$. Konec iterace, i se inkrementuje na $i+1$, a invariant platí pro další iteraci.
3. **Ukončení ($i = n+1$):** Proměnná cyklu i začíná na 2 a v každé iteraci se zvyšuje o 1. Jakmile hodnota i překročí n (na řádku 1), cyklus se ukončí (tj. i se rovná $n+1$). Dosazením $n+1$ za i do invariantu získáme, že pole $A[1:n]$ obsahuje prvky, které byly původně v $A[1:n]$, ale jsou nyní seřazeny. **Algoritmus je tedy korektní.**

4 Metoda Rozděl a panuj

- Mnoho efektivních algoritmů je založeno na této rekurzivní technice.
- **Tři kroky:**
  1. **Rozděl (Divide):** Rozděl problém na několik menších podproblémů, které jsou instancemi stejného problému, ale menší.
  2. **Panuj (Conquer):** Rekurzivně řeš podproblémy. Pokud jsou podproblémy dostatečně malé, řeš je přímo.
  3. **Sluč (Combine):** Zkombinuj řešení podproblémů do řešení původního problému.
- **Kroky Merge Sortu pro řazení pole $A[p:r]$:**
  1. **Rozděl:** Rozděl pole $A[p:r]$ na dvě podpole: $A[p:q]$ a $A[q+1:r]$, kde q je střed p a r.
  2. **Panuj:** Rekurzivně seřaď obě podpole voláním MERGE-SORT pro $A[p:q]$ a $A[q+1:r]$.
  3. **Sluč:** Sluč (merge) seřazená pole $A[p:q]$ a $A[q+1:r]$ do jednoho seřazeného pole $A[p:r]$.
- **Základní případ:** Pokud pole obsahuje jeden prvek, je již seřazeno.

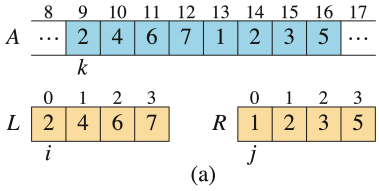
MERGE-SORT(A, p, r)

```
1  if p ≥ r                          // zero or one element?
2      return
3  q = ⌊(p + r)/2⌋                   // midpoint of A[p : r]
4  MERGE-SORT(A, p, q)               // recursively sort A[p : q]
5  MERGE-SORT(A, q + 1, r)           // recursively sort A[q + 1 : r]
6  // Merge A[p : q] and A[q + 1 : r] into A[p : r].
7  MERGE(A, p, q, r)
```

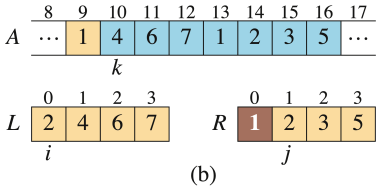



```
1  n_L = q − p + 1          // length of A[p : q]
2  n_R = r − q              // length of A[q + 1 : r]
3  let L[0 : n_L − 1] and R[0 : n_R − 1] be new arrays
4  for i = 0 to n_L − 1     // copy A[p : q] into L[0 : n_L − 1]
5      L[i] = A[p + i]
6  for j = 0 to n_R − 1     // copy A[q + 1 : r] into R[0 : n_R − 1]
7      R[j] = A[q + j + 1]
```

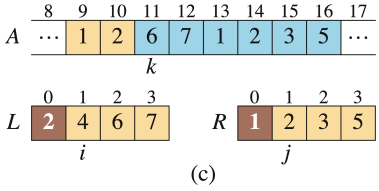
```
8   i = 0                    // i indexes the smallest remaining element in L
9   j = 0                    // j indexes the smallest remaining element in R
10  k = p                    // k indexes the location in A to fill
11  // As long as each of the arrays L and R contains an unmerged element,
    //       copy the smallest unmerged element back into A[p : r].
12  while i < n_L and j < n_R
13      if L[i] ≤ R[j]
14          A[k] = L[i]
15          i = i + 1
16      else A[k] = R[j]
17          j = j + 1
18      k = k + 1

19  // Having gone through one of L and R entirely, copy the
    //       remainder of the other to the end of A[p : r].
20  while i < n_L
21      A[k] = L[i]
22      i = i + 1
23      k = k + 1
24  while j < n_R
25      A[k] = R[j]
26      j = j + 1
27      k = k + 1
```



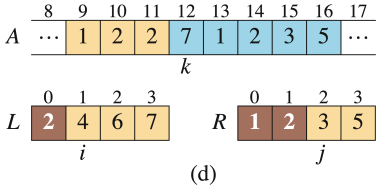
(a)



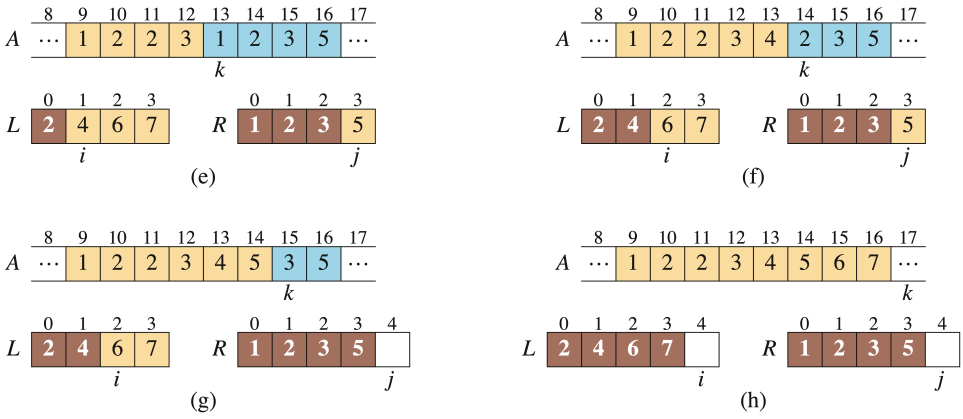
(b)



(c)



(d)



- Dobu běhu rekurzivního algoritmu lze často popsat **rekurentní rovnicí (rekurencí)**.
- Rekurence popisuje celkovou dobu běhu pro problém o velikosti n z hlediska doby běhu stejného algoritmu na menších vstupech.
- **Rekurence pro algoritmy "Rozděl a panuj":**
  - $T(n) =$ doba běhu pro problém velikosti n.
  - Pokud je velikost problému dostatečně malá (např. $n < n_0$ pro $n_0 > 0$), přímočaré řešení trvá **konstantní čas**, což píšeme jako $\Theta(1)$.
  - Rozdělení problému vede k a **podproblémům**, každý o velikosti n/b.
  - Řešení a podproblémů trvá $aT(n/b)$ času.
  - Pokud *dělení* problému na podproblémy trvá $D(n)$ času a *kombinování* řešení podproblémů trvá $C(n)$ času, dostaneme rekurenci:

$$T(n) = \begin{cases} \Theta(1) & \text{pokud } n < n_0 \\ D(n) + aT(n/b) + C(n) & \text{jinak} \end{cases}$$

- **Ignorování dolních a horních celých částí (floors a ceilings):**
  - Např. MERGE-SORT dělí problém velikosti n na podproblémy velikosti $\lfloor n/2 \rfloor$ a $\lceil n/2 \rceil$.
  - Protože rozdíl je nanejvýš 1, jednoduše je budeme nazývat oba jako velikost $n/2$.
  - Tato zjednodušení obecně **neovlivňují řád růstu** řešení rekurence.

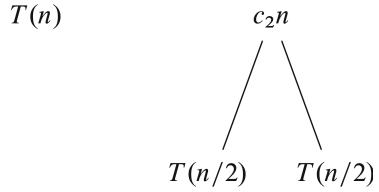
- **Vynechání explicitního uvádění bázových případů:**
  - Bázové případy jsou téměř vždy $T(n) = \Theta(1)$ pro $n < n_0$ (nějakou konstantu $n_0 > 0$).
  - Doba běhu algoritmu na vstupu konstantní velikosti je vždy konstantní.
  - Tím se ušetří mnoho zbytečného psaní.
- Pokud $n = 1$, pak $T(n) = \Theta(1)$ (základní případ).
- Pokud $n > 1$:
  1. **Rozděl:** Výpočet q trvá $D(n) = \Theta(1)$.
  2. **Panuj:** Dvě rekurzivní volání na pole o velikosti $n/2$. To je $2T(n/2)$.
  3. **Sluč:** Procedura MERGE pro pole o velikosti n trvá $\Theta(n)$.
- **Rekurzivní vztah:**

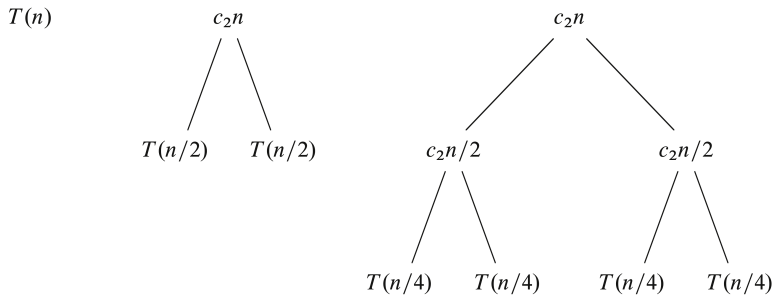
$$T(n) = \begin{cases} \Theta(1) & \text{pokud } n = 1 \\ 2T(n/2) + \Theta(n) & \text{pokud } n > 1 \end{cases}$$

- Složitost mergesortu je $T(n) = \Theta(n \log n)$.
- Co to znamená a jak jsme k tomu došli?
- K analýze složitosti mergesortu budeme potřebovat
  1. jak řešit rekurzivní vztahy $T(n)$ – nyní jen stručně, více později
  2. asymptotickou notaci – $\Theta()$ a pod. – za okamžik

$$T(n) = \begin{cases} c_1 & \text{pokud } n = 1 \\ 2T(n/2) + c_2 n & \text{pokud } n > 1 \end{cases}$$

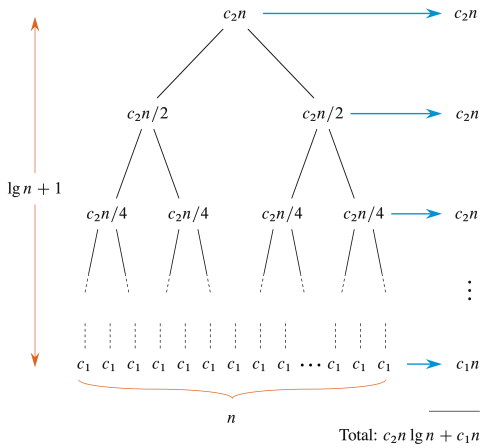
- **Úroveň 0 (kořen):** Problém velikosti n, cena $c_2 n$.
- **Úroveň 1:** Dva problémy velikosti $n/2$, každý s cenou $c_2 n/2$. Celkem $2 \times c_2 n/2 = c_2 n$.





$$T(n) = \begin{cases} c_1 & \text{pokud } n = 1 \\ 2T(n/2) + c_2 n & \text{pokud } n > 1 \end{cases}$$

- **Úroveň 0 (kořen):** Problém velikosti n, cena $c_2 n$.
- **Úroveň 1:** Dva problémy velikosti $n/2$, každý s cenou $c_2 n/2$. Celkem $2 \times c_2 n/2 = c_2 n$.
- **Úroveň i:** 2^i problémů velikosti $n/2^i$, cena celkem $2^i \times c_2 n/2^i = c_2 n$.
- **Poslední úroveň (listy):** Velikost problému je 1. $n/2^k = 1 \implies 2^k = n \implies k = \log_2 n$.
- Na každé úrovni je celková cena $c_2 n$ — u listů je to $c_1 n$!
- Počet úrovní je $\log_2 n + 1$ (začínáme od 0).



$$T(n) = \begin{cases} c_1 & \text{pokud } n = 1 \\ 2T(n/2) + c_2 n & \text{pokud } n > 1 \end{cases}$$

- **Úroveň 0 (kořen):** Problém velikosti n, cena $c_2 n$.
- **Úroveň 1:** Dva problémy velikosti $n/2$, každý s cenou $c_2 n/2$. Celkem $2 \times c_2 n/2 = c_2 n$.
- **Úroveň i:** 2^i problémů velikosti $n/2^i$, cena celkem $2^i \times c_2 n/2^i = c_2 n$.
- **Poslední úroveň (listy):** Velikost problému je 1. $n/2^k = 1 \implies 2^k = n \implies k = \log_2 n$.
- Na každé úrovni je celková cena $c_2 n$ — u listů je to $c_1 n$!
- Počet úrovní je $\log_2 n + 1$ (začínáme od 0).
- Celková cena: $(\log_2 n) \times c_2 n + c_1 n = c_2 n \log n + c_1 n$.

5 Úvod do asymptotické notace

Proč jsme místo $\Theta(1)$ psali c_1 a místo $\Theta(n)$ psali $c_2 n$ pro nějaké konstanty c_1 a c_2?

- Užitečný odhad časové (a prostorové) složitosti algoritmů.
- **Problém:** Přesný čas závisí na:
  - Konkrétním hardwaru (rychlost procesoru, paměť).
  - Kompilátoru a programovacím jazyce.
  - Kvalitě kódu programátora.
- Chceme **abstrahovat** od těchto konkrétních detailů.

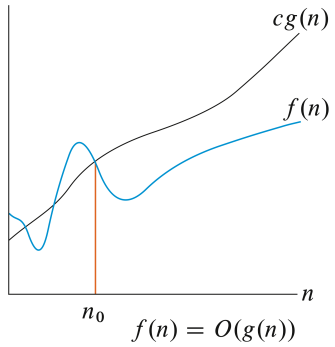
Řešení
Asymptotická notace nám umožňuje popsat **rychlost růstu** času běhu algoritmu s rostoucí velikostí vstupu.

- Čas běhu algoritmu $T(n)$ je často polynom, např. $T(n) = an^2 + bn + c$.
- Pro malé n mohou být konstanty a, b, c a nižší členy důležité.
- Pro **velké** n (což nás v analýze alg. zajímá nejvíce) dominuje **člen s nejvyšším stupněm**.
- Příklad: $T(n) = 2n^2 + 100n + 500$
  - Pro $n = 10$: $200 + 1000 + 500 = 1700$
  - Pro $n = 100$: $20000 + 10000 + 500 = 30500$
  - Pro $n = 1000$: $2\,000\,000 + 100\,000 + 500 = 2\,100\,500$

- Pro velké n, člen $2n^2$ tvoří drtivou většinu. Konstantní faktor 2 je méně důležitý než n^2.

Definice 5.1. Pro dané funkce $f(n)$ a $g(n)$ řekneme, že $f(n) \in \mathcal{O}(g(n))$, pokud existují **kladné** konstanty c a n_0 takové, že:

$$0 \leq f(n) \leq c \cdot g(n) \quad \text{pro všechna } n \geq n_0$$



- $\mathcal{O}(g(n))$ je množina funkcí. Místo $f(n) \in \mathcal{O}(g(n))$ se často píše $f(n) = \mathcal{O}(g(n))$.
- Znamená to, že $f(n)$ roste **nejvýše tak rychle** jako $g(n)$, až na konstantní faktor a pro dostatečně velké n.
- Poskytuje **asymptotický horní odhad**.
- Chceme najít c a n_0 tak, aby $2n^2 + 3n + 1 \leq c \cdot n^2$ pro všechna $n \geq n_0$.
- Pro $n \geq 1$:

$$\begin{aligned} 2n^2 + 3n + 1 &\leq 2n^2 + 3n^2 + n^2 \quad (\text{protože } n \leq n^2 \text{ a } 1 \leq n^2 \text{ pro } n \geq 1) \\ &= 6n^2 \end{aligned}$$

- Můžeme zvolit $c = 6$ a $n_0 = 1$.
- Tedy $2n^2 + 3n + 1 \in \mathcal{O}(n^2)$.

Obecně
Jakýkoli polynom $a_k n^k + a_{k-1} n^{k-1} + \cdots + a_0$ je v $\mathcal{O}(n^k)$. Stačí vzít c jako součet absolutních hodnot koeficientů a $n_0 = 1$.

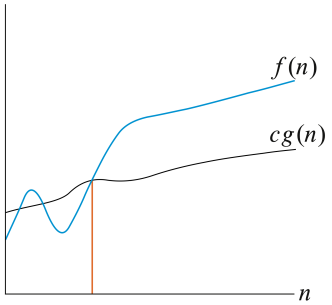
- Chceme $3n + 2 \leq c \cdot n$ pro $n \geq n_0$.
- Pokud $n \geq 2$, pak $3n + 2 \leq 3n + n = 4n$.
- Můžeme zvolit $c = 4$ a $n_0 = 2$.

- Tedy $3n + 2 \in \mathcal{O}(n)$.

Závěr
Big-O notace ignoruje nižší členy a konstantní faktory. Zaměřuje se na **řád růstu** funkce.

Definice 5.2. Pro dané funkce $f(n)$ a $g(n)$ řekneme, že $f(n) \in \Omega(g(n))$, pokud existují **kladné** konstanty c a n_0 takové, že:

$$0 \leq c \cdot g(n) \leq f(n) \quad \text{pro všechna } n \geq n_0$$



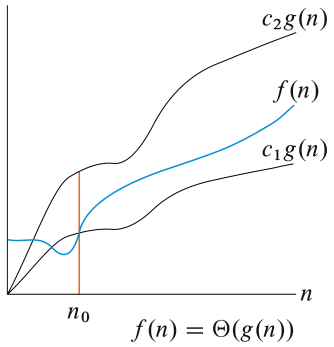
$f(n) = \Omega(g(n))$

- Znamená to, že $f(n)$ roste **alespoň tak rychle** jako $g(n)$, až na konstantní faktor a pro dostatečně velké n.
- Poskytuje **asymptotický spodní odhad**.
- Chceme najít c a n_0 tak, aby $c \cdot n^2 \leq 2n^2 + 3n + 1$ pro všechna $n \geq n_0$.
- Můžeme zvolit $c = 1$ a $n_0 = 1$.
- Pak $1 \cdot n^2 \leq 2n^2 + 3n + 1$ zjevně platí pro $n \geq 1$.
- Tedy $2n^2 + 3n + 1 \in \Omega(n^2)$.

Vztah s Big-O
$f(n) \in \Omega(g(n))$ právě tehdy, když $g(n) \in \mathcal{O}(f(n))$. Jsou to navzájem "inverzní" notace pro horní/dolní odhad.

Definice 5.3. Pro dané funkce $f(n)$ a $g(n)$ řekneme, že $f(n) \in \Theta(g(n))$, pokud existují **kladné** konstanty c_1, c_2 a n_0 takové, že:

$$0 \leq c_1 \cdot g(n) \leq f(n) \leq c_2 \cdot g(n) \quad \text{pro všechna } n \geq n_0$$



- Znamená to, že $f(n)$ roste **stejně rychle** jako $g(n)$, až na konstantní faktory a pro dostatečně velké n.
- Poskytuje **asymptoticky těsný odhad** (horní i dolní).
- **Ekvivalence:** $f(n) \in \Theta(g(n))$ právě tehdy, když $f(n) \in \mathcal{O}(g(n))$ **a** $f(n) \in \Omega(g(n))$.
- Chceme najít c_1, c_2 a n_0 tak, aby $c_1 n^2 \leq 2n^2 + 3n + 1 \leq c_2 n^2$.
- Z Big-O příkladu víme, že $2n^2 + 3n + 1 \leq 6n^2$ pro $n \geq 1$. Takže můžeme použít $c_2 = 6$.
- Z Omega příkladu víme, že $1n^2 \leq 2n^2 + 3n + 1$ pro $n \geq 1$. Takže můžeme použít $c_1 = 1$.
- Společné $n_0 = 1$.
- Tedy $2n^2 + 3n + 1 \in \Theta(n^2)$.

Shrnutí

- $f(n) = \mathcal{O}(g(n))$: $f(n)$ roste nejvýše tak rychle jako $g(n)$.
- $f(n) = \Omega(g(n))$: $f(n)$ roste alespoň tak rychle jako $g(n)$.
- $f(n) = \Theta(g(n))$: $f(n)$ roste stejně rychle jako $g(n)$.

Definice 5.4. Funkce $f(n)$ je v $o(g(n))$, pokud pro libovolnou kladnou konstantu $c > 0$ existuje konstanta $n_0 > 0$ taková, že

$$0 \leq f(n) < c \cdot g(n) \quad \text{pro všechna } n \geq n_0$$

- Znamená to, že $f(n)$ roste **podstatně pomaleji** než $g(n)$, tj. $\lim_{n\to\infty} \frac{f(n)}{g(n)} = 0$.
- Příklad: $n^2 \in o(n^3)$ protože $\lim_{n\to\infty} \frac{n^2}{n^3} = \lim_{n\to\infty} \frac{1}{n} = 0$.

- **Vztah s Big-O:** Pokud $f(n) \in o(g(n))$, pak $f(n) \in \mathcal{O}(g(n))$, ale ne naopak.

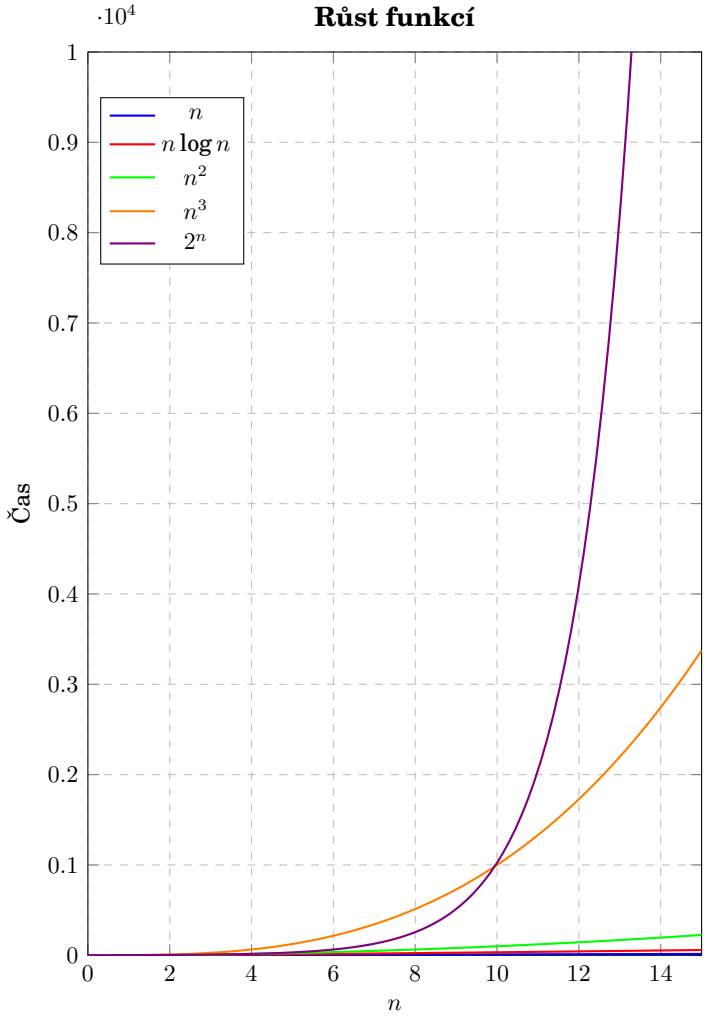
Definice 5.5. Funkce $f(n)$ je v $\omega(g(n))$, pokud $f(n)$ roste podstatně rychleji než $g(n)$. Formálně:

$$f(n) \in \omega(g(n)) \text{ právě tehdy, když } g(n) \in o(f(n))$$

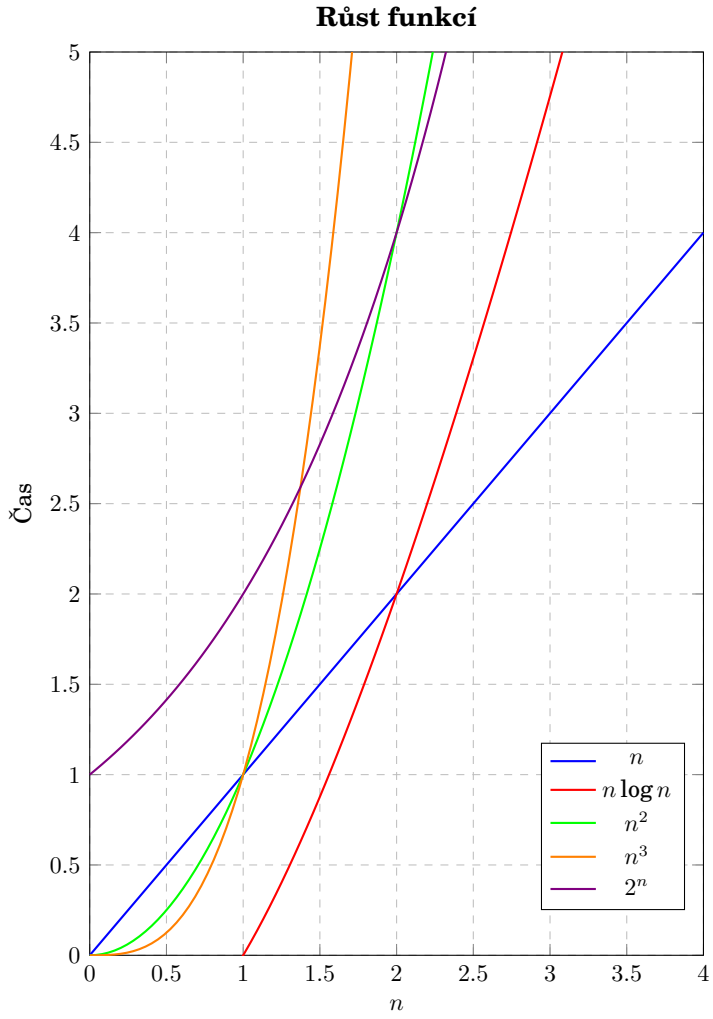
- Znamená to, že $f(n)$ roste **podstatně rychleji** než $g(n)$, tj. $\lim_{n\to\infty} \frac{f(n)}{g(n)} = \infty$.
- Příklad: $n^3 \in \omega(n^2)$ protože $\lim_{n\to\infty} \frac{n^3}{n^2} = \lim_{n\to\infty} n = \infty$.
- **Vztah s Big-Omega:** Pokud $f(n) \in \omega(g(n))$, pak $f(n) \in \Omega(g(n))$, ale ne naopak.
- Asymptotická notace nám umožňuje uspořádat funkce podle jejich rychlosti růstu.
- Intuitivně: $f \prec g$ znamená $f \in o(g)$.
- **Příklady běžných funkcí (od nejpomalejší po nejrychlejší):**
  - Konstantní: $\Theta(1)$
  - Logaritmické: $\Theta(\log n)$
  - Polylogaritmické: $\Theta(\log^k n)$
  - Lineární: $\Theta(n)$
  - Lineární logaritmické: $\Theta(n \log n)$
  - Kvadratické: $\Theta(n^2)$
  - Polynomiální: $\Theta(n^k)$ (kde $k > 1$ je konstanta)
  - Exponenciální: $\Theta(a^n)$ (kde $a > 1$ je konstanta)
  - Faktoriál: $\Theta(n!)$
- Vždy platí: $\log n \prec n \prec n \log n \prec n^2 \prec n^3 \prec 2^n \prec n!$.
- V analýze algoritmů se téměř vždy používá **základ 2 pro logaritmy** ($\log_2 n$). Pokud není specifikováno, $\log n$ znamená $\log_2 n$. Důvodem je, že změna báze logaritmu je jen konstantní faktor

  $$\log_b x = \frac{\log_a x}{\log_a b}$$

  a ty asymptotická notace ignoruje.
- **Polynomiální vs. Exponenciální:** Algoritmy s polynomiální složitostí (n^k) jsou považovány za **efektivní** (polynomiální čas), zatímco algoritmy s exponenciální složitostí (a^n) jsou obvykle **neefektivní** pro velké vstupy.



Obrázek 1: Grafické porovnání rychlosti růstu funkcí



Obrázek 2: Grafické porovnání rychlosti růstu funkcí

Cílem je použít asymptotickou notaci co **nejpřesněji**.

Příklady pro třídění vkládáním (Insertion Sort):

- **Nejhorší případ:**
  - Správně lze říci: $O(n^2)$, $\Omega(n^2)$, $\Theta(n^2)$.
  - $\Theta(n^2)$ **je nejpřesnější**, a proto nejpreferovanější.
- **Nejlepší případ:**
  - Správně lze říci: $O(n)$, $\Omega(n)$, $\Theta(n)$.
  - $\Theta(n)$ **je nejpřesnější**, a proto nejpreferovanější.
- **Co není správně:**
  - "Doba běhu řazení vkládáním je $\Theta(n^2)$."
  - Toto je **přehánění**, protože nezmiňuje "nejhorší případ" a vytváří plošné prohlášení. Insertion Sort neběží v čase $\Theta(n^2)$ ve všech případech (např. v nejlepším případě je $\Theta(n)$).
  - Podobně nelze říci: "Doba běhu řazení vkládáním je $\Theta(n)$."
- **Co je správně:**
  - "Doba běhu řazení vkládáním je $O(n^2)$", protože ve všech případech doba běhu neroste rychleji než n^2. (U $O(n^2)$ není problém, pokud existují případy, kde roste pomaleji).
  - "Doba běhu řazení vkládáním je $\Omega(n)$."
- **Pro Merge Sort:**
  - Jelikož Merge Sort běží v čase $\Theta(n \log n)$ **ve všech případech**, lze jednoduše říci: "Doba běhu Merge Sortu je $\Theta(n \log n)$." bez specifikace nejhoršího, nejlepšího nebo jiného případu.

Častá chyba: Záměna O-notace a Θ-notace.

- Lidé mylně používají O-notaci k označení asymptoticky těsného odhadu.
- Např. "algoritmus s časovou složitostí $O(n \log n)$ je rychlejší než algoritmus s $O(n^2)$." Toto tvrzení je **zavádějící**, protože algoritmus $O(n^2)$ může ve skutečnosti běžet v čase $\Theta(n)$.
- **Volte vhodnou notaci:** Pro asymptoticky těsný odhad použijte Θ-**-notaci**.

Cíl: Nejjednodušší a nejpřesnější odhady.

- Pokud má algoritmus dobu běhu $3n^2+20n$ ve všech případech, zapíšeme ji jako $\Theta(n^2)$.
- I když je formálně správné $O(n^3)$ nebo $\Theta(3n^2+20n)$, nejsou tyto výrazy tak užitečné:
  - $O(n^3)$ je **méně přesné**.
  - $\Theta(3n^2+20n)$ **zastiňuje řád růstu** zbytečnou složitostí.
- Formálně definujeme asymptotickou notaci pomocí množin, ale v rovnicích používáme **znaménko rovnosti (=)** namísto znaménka příslušnosti k množině ($\in$).
- **Interpretace:**
  - **Samostatná notace na pravé straně:** Např. $4n^2+100n+500 = O(n^2)$ znamená $4n^2+100n+500 \in O(n^2)$. Znaménko rovnosti zde znamená příslušnost k množině.
  - **Notace uvnitř složitějšího vzorce:** Asymptotická notace zde zastupuje **nějakou anonymní funkci**, kterou nechceme jmenovat.
    - Např. $2n^2+3n+1 = 2n^2+\Theta(n)$ znamená, že $2n^2+3n+1 = 2n^2+f(n)$, kde $f(n) \in \Theta(n)$. V tomto případě je $f(n) = 3n+1$.
- **Účel:** Použití asymptotické notace tímto způsobem může pomoci eliminovat nepodstatné detaily a nepřehlednost v rovnici.
  - Např. rekurence pro Merge Sort: $T(n) = 2T(n/2)+\Theta(n)$. Nemá smysl specifikovat přesně všechny členy nižšího řádu, když jsou zahrnuty v anonymní funkci $\Theta(n)$.
- **Počet anonymních funkcí:**
  - Odpovídá počtu, kolikrát se asymptotická notace objeví ve výrazu.
  - Např. ve výrazu $\sum_{i=1}^{n} O(i)$ existuje pouze **jedna anonymní funkce** (funkce proměnné i).
  - Tento výraz se liší od $O(1)+O(2)+\cdots+O(n)$, který nemá jasnou interpretaci.
- **Asymptotická notace na levé straně rovnice:**
  - Např. $2n^2+\Theta(n) = \Theta(n^2)$.
  - **Pravidlo interpretace:** Bez ohledu na to, jak jsou vybrány anonymní funkce na levé straně rovnosti, existuje způsob, jak vybrat anonymní funkce na pravé straně, aby rovnice platila.
  - **Příklad interpretace:** Pro jakoukoli funkci $f(n) \in \Theta(n)$ existuje funkce $g(n) \in \Theta(n^2)$ taková, že $2n^2+f(n) = g(n)$ pro všechna n.

  - To znamená, že pravá strana rovnice poskytuje **hrubší úroveň detailů** než levá strana.

- Je možné řetězit několik takových vztahů za sebou, např.:

$$2n^2 + 3n + 1 = 2n^2 + \Theta(n) = \Theta(n^2)$$

- Každá rovnice se interpretuje **samostatně** podle výše uvedených pravidel.

- **Interpretace řetězce:**
  - **První rovnice:** $2n^2 + 3n + 1 = 2n^2 + \Theta(n)$ znamená, že existuje funkce $f(n) \in \Theta(n)$ taková, že $2n^2+3n+1 = 2n^2+f(n)$ pro všechna n.
  - **Druhá rovnice:** $2n^2 + \Theta(n) = \Theta(n^2)$ znamená, že pro jakoukoli funkci $g(n) \in \Theta(n)$ (jako je $f(n)$ z předchozí rovnice) existuje funkce $h(n) \in \Theta(n^2)$ taková, že $2n^2 + g(n) = h(n)$ pro všechna n.

- Tato interpretace naznačuje, že $2n^2 + 3n + 1 = \Theta(n^2)$, což intuitivně vyplývá z řetězení rovnic.

- Dalším zjednodušením asymptotické notace je, že **proměnná jdoucí k nekonečnu musí být odvozena z kontextu**.

- **Příklad:**
  - $O(g(n))$: předpokládáme, že nás zajímá růst $g(n)$ s rostoucím n.
  - $O(g(m))$: mluvíme o růstu $g(m)$ s rostoucím m.
  - **Volná proměnná** ve výrazu ukazuje, která proměnná jde k nekonečnu.

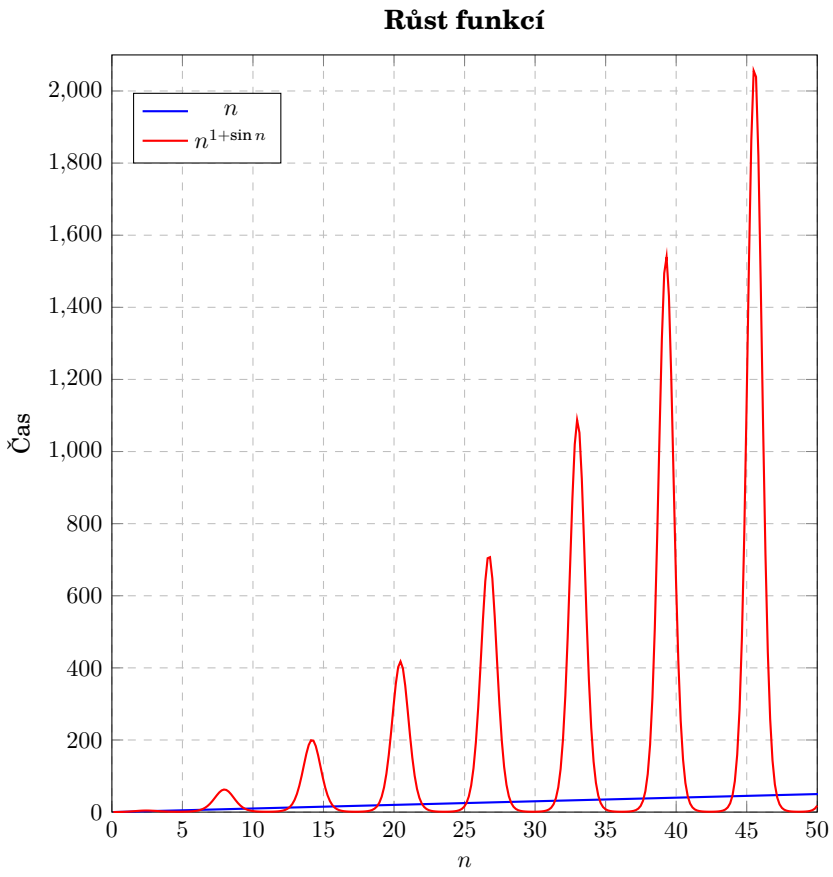
- **Nejčastější situace: Výraz $O(1)$.**
  - Z výrazu nelze odvodit, která proměnná jde k nekonečnu, protože tam žádná proměnná není.
  - **Kontext musí odstranit nejednoznačnost.** Např. v rovnici $f(n) = O(1)$ je zřejmé, že proměnnou je n.
  - Díky kontextu můžeme výrazu dokonale porozumět: $f(n) = O(1)$ znamená, že funkce $f(n)$ je **shora ohraničena konstantou**, když n jde k nekonečnu.
  - **Důvod zjednodušení:** Explicitní uvedení proměnné by notaci zahltilo. Kontext zajišťuje srozumitelnost.

- Další způsob zjednodušení nastává, když je funkce uvnitř asymptotické notace **ohraničena kladnou konstantou**, např. $T(n) = O(1)$, a to zejména při uvádění rekurencí.

- **Příklad:** Často se píše $T(n) = O(1)$ pro $n < 3$.
  - **Formálně je to nesmysl:** Definice O-notace říká, že $T(n)$ je shora ohraničeno kladnou konstantou c pro $n \geq n_0$ (pro nějaké $n_0 > 0$). Hodnota $T(n)$ pro $n < n_0$ nemusí být takto ohraničena. Tedy z $T(n) = O(1)$ pro $n < 3$ nelze odvodit žádné omezení, pokud by $n_0 > 3$.
- **Konvenční význam:** Když říkáme $T(n) = O(1)$ pro $n < 3$, konvenčně se tím myslí, že **existuje kladná konstanta** c **taková, že** $T(n) \leq c$ **pro** $n < 3$.
  - Toto konvenční použití **šetří námahu s pojmenováním** konstanty, umožňuje jí zůstat anonymní, zatímco se soustředíme na důležitější proměnné v analýze.
  - Podobná zjednodušení se vyskytují i u jiných asymptotických notací (např. $T(n) = \Theta(1)$ pro $n < 3$ znamená, že $T(n)$ je shora i zdola ohraničeno kladnými konstantami, když $n < 3$).
- **Omezené domény vstupních velikostí:**
  - Někdy funkce popisující dobu běhu algoritmu nemusí být definována pro určité velikosti vstupu (např. když algoritmus předpokládá, že velikost vstupu je přesná mocnina 2).
  - Asymptotickou notaci stále používáme k popisu růstu doby běhu, s pochopením, že jakékoli omezení platí pouze tam, kde je funkce definována.
  - Např. $f(n) = O(g(n))$ pro $f(n)$ definované jen na podmnožině přirozených čísel znamená, že odhad $0 \leq T(n) \leq cg(n)$ platí pro všechna $n \geq n_0$ **v doméně** $f(n)$.
  - Toto zjednodušení se zřídka zmiňuje, protože význam je obvykle jasný z kontextu.
- **Proč "zjednodušení" notace?**
  - V matematice je přijatelné a často žádoucí "zjednodušit" notaci, pokud ji **používáme správně**.
  - Pokud přesně rozumíme, co se zjednodušením myslí, a neděláme chybné závěry, může to:
    * Zjednodušit náš matematický jazyk.
    * Přispět k našemu pochopení na vyšší úrovni.
    * Pomoci nám soustředit se na to, co je skutečně důležité.
- **Tranzitivita:**
  - Pokud $f(n) \in \mathcal{O}(g(n))$ a $g(n) \in \mathcal{O}(h(n))$, pak $f(n) \in \mathcal{O}(h(n))$.

  - Platí obdobně pro $\Omega(), \Theta(), o(), \omega()$.

- **Reflexivita:**
  - $f(n) \in \mathcal{O}(f(n))$
  - $f(n) \in \Omega(f(n))$
  - $f(n) \in \Theta(f(n))$
- **Symetrie:**
  - $f(n) \in \Theta(g(n))$ právě tehdy, když $g(n) \in \Theta(f(n))$.
- **Transpoziční symetrie:**
  - $f(n) \in \mathcal{O}(g(n))$ právě tehdy, když $g(n) \in \Omega(f(n))$.
  - $f(n) \in o(g(n))$ právě tehdy, když $g(n) \in \omega(f(n))$.
- Díky vlastnostem asymptotických notací můžeme vytvořit **analogii** mezi asymptotickým srovnáním dvou funkcí f a g a srovnáním dvou reálných čísel a a b:
  - $f(n) = O(g(n))$ je jako $a \leq b$
  - $f(n) = \Omega(g(n))$ je jako $a \geq b$
  - $f(n) = \Theta(g(n))$ je jako $a = b$
  - $f(n) = o(g(n))$ je jako $a < b$
  - $f(n) = \omega(g(n))$ je jako $a > b$
- Říkáme, že $f(n)$ je **asymptoticky menší** než $g(n)$, pokud $f(n) = o(g(n))$.
- Říkáme, že $f(n)$ je **asymptoticky větší** než $g(n)$, pokud $f(n) = \omega(g(n))$.
- Jedna vlastnost reálných čísel se však na asymptotickou notaci **nepřenáší**:
- **Trichotomie pro reálná čísla:** Pro jakákoli dvě reálná čísla a a b musí platit přesně jedno z následujícího: $a < b$, $a = b$, nebo $a > b$.
- Ačkoliv lze porovnat jakákoli dvě reálná čísla, **ne všechny funkce jsou asymptoticky srovnatelné**.
  - To znamená, že pro dvě funkce $f(n)$ a $g(n)$ nemusí platit ani $f(n) = O(g(n))$, ani $f(n) = \Omega(g(n))$.
- **Příklad nesrovnatelných funkcí:**
  - Nelze porovnat funkce n a $n^{1+\sin n}$ pomocí asymptotické notace.
  - Hodnota exponentu v $n^{1+\sin n}$ osciluje mezi 0 a 2, přičemž nabývá všech hodnot mezi nimi.



Obrázek 3: Grafické porovnání rychlosti růstu funkcí

- **Asymptotická notace:** Nástroj pro popis rychlosti růstu funkcí, ignoruje konstantní faktory a nižší členy.
- **Big-O ($\mathcal{O}$):** Horní asymptotický odhad ($f(n)$ roste nejvýše tak rychle jako $g(n)$).
- **Omega (Ω):** Dolní asymptotický odhad ($f(n)$ roste alespoň tak rychle jako $g(n)$).
- **Theta (Θ):** Těsný asymptotický odhad ($f(n)$ roste stejně rychle jako $g(n)$).
- **Little-o (o):** Netěsný horní odhad ($f(n)$ roste podstatně pomaleji než $g(n)$).
- **Little-omega (ω):** Netěsný dolní odhad ($f(n)$ roste podstatně rychleji než $g(n)$).
- **Hierarchie funkcí:** Umožňuje srovnávat algoritmy a pochopit jejich škálovatelnost.

Monotonicita:

- Funkce $f(n)$ je **monotonně rostoucí**, pokud $m \leq n \implies f(m) \leq f(n)$.
- **Monotonně klesající**, pokud $m \leq n \implies f(m) \geq f(n)$.
- **Striktně rostoucí**, pokud $m < n \implies f(m) < f(n)$.
- **Striktně klesající**, pokud $m < n \implies f(m) > f(n)$.

Funkce dolní celá část ($\lfloor x \rfloor$) a horní celá část ($\lceil x \rceil$):

- $\lfloor x \rfloor$: největší celé číslo menší nebo rovné x.
- $\lceil x \rceil$: nejmenší celé číslo větší nebo rovné x.
- Obě funkce jsou monotonně rostoucí.
- **Klíčové vlastnosti:** Pro celé číslo n a reálné číslo x:
  - $\lfloor n \rfloor = n = \lceil n \rceil$
  - $x - 1 < \lfloor x \rfloor \leq x \leq \lceil x \rceil < x + 1$
  - $\lfloor -x \rfloor = -\lceil x \rceil$
  - $\lfloor n + x \rfloor = n + \lfloor x \rfloor$; $\lceil n + x \rceil = n + \lceil x \rceil$

Modulární aritmetika:

- $a \pmod{n}$: zbytek po dělení a/n.
- Platí: $0 \leq a \pmod{n} < n$ (i pro záporné a).

- **Ekvivalence modulo n ($a \equiv b \pmod{n}$):**
  - Znamená, že a a b mají stejný zbytek po dělení n.
  - Ekvivalentně: n je dělitelem $b - a$.
  - Zápis $a \not\equiv b \pmod{n}$ pro neekvivalentní hodnoty.

Polynomy:

- Polynom stupně d: $p(n) = \sum_{i=0}^{d} a_i n^i$, kde $a_d \neq 0$.
- **Asymptoticky pozitivní:** pokud $a_d > 0$.
- Pro asymptoticky pozitivní polynom stupně d: $p(n) = \Theta(n^d)$.
- **Vlastnosti n^a:**
  - Monotonně rostoucí pro $a \geq 0$.
  - Monotonně klesající pro $a \leq 0$.
- Funkce $f(n)$ je **polynomiálně ohraničená**, pokud $f(n) = O(n^k)$ pro nějakou konstantu k.
- **Základní identity exponenciál ($a > 0, m, n$ reálné):**
  - $a^0 = 1$, $a^1 = a$, $a^{-1} = 1/a$
  - $(a^m)^n = a^{mn}$, $(a^m)^n = (a^n)^m$, $a^m a^n = a^{m+n}$
- **Monotonicita:** Funkce a^n je monotonně rostoucí v n pro $a \geq 1$.
- **Vztah k polynomům:**
  - Pro $a > 1$ a libovolné b platí $\lim_{n\to\infty} \frac{n^b}{a^n} = 0$.
  - Z toho vyplývá: $n^b = o(a^n)$.
  - **Jakákoli exponenciální funkce se základem striktně větším než 1 roste rychleji než jakákoli polynomiální funkce.**
- **Základ $e \approx 2.71828$ (přirozený logaritmus):**
  - Taylorův rozvoj: $e^x = 1 + x + \frac{x^2}{2!} + \frac{x^3}{3!} + \cdots = \sum_{i=0}^{\infty} \frac{x^i}{i!}$.
  - Nerovnosti:
    - $1 + x \leq e^x$ (rovnost jen pro $x = 0$).
    - Pro $|x| \leq 1$: $1 + x \leq e^x \leq 1 + x + x^2$.
  - Aproximace pro $x \to 0$: $e^x = 1 + x + \Theta(x^2)$.
  - Důležitá limita: $\lim_{n\to\infty} \left(1 + \frac{x}{n}\right)^n = e^x$.
- **Notace:**

  - $\log n = \log_2 n$ (binární logaritmus)
  - $\ln n = \log_e n$ (přirozený logaritmus)
  - $\log^k n = (\log n)^k$ (exponent)
  - $\log \log n = \log(\log n)$ (kompozice)
- **Konvence:**
  - Bez závorek se logaritmická funkce vztahuje pouze na následující člen.
  - Např. $\log n + 1$ znamená $(\log n) + 1$, ne $\log(n + 1)$.
- **Obecné vlastnosti logaritmu (pro $b > 1$):**
  - Nedefinovaný, pokud $n \leq 0$.
  - Striktně rostoucí, pokud $n > 0$.
  - Negativní, pokud $0 < n < 1$.
  - Pozitivní, pokud $n > 1$.
  - 0, pokud $n = 1$.
- **Proč $\log n$ (základ 2) je preferovaný:**
  - Počítačoví vědci považují 2 za nejpřirozenější základ, protože mnoho algoritmů a datových struktur zahrnuje rozdělení problému na dvě části.
- **Logaritmické identity ($a, b, c > 0$, báze $\neq 1$):**
  - $a = b^{\log_b a}$
  - $\log_c(ab) = \log_c a + \log_c b$
  - $\log_b a^n = n \log_b a$
  - $\log_b(1/a) = -\log_b a$
  - $\log_b a = 1/\log_a b$
  - $a^{\log_b c} = c^{\log_b a}$
- **Řadový rozvoj pro $\ln(1 + x)$ (pro $|x| < 1$):**
  - $\ln(1 + x) = x - \frac{x^2}{2} + \frac{x^3}{3} - \frac{x^4}{4} + \frac{x^5}{5} - ...$
- **Nerovnosti pro $\ln(1 + x)$ (pro $x > -1$):**
  - $\frac{x}{1+x} \leq \ln(1 + x) \leq x$ (rovnost jen pro $x = 0$).
- **Polylogaritmicky ohraničená funkce:**
  - Funkce $f(n)$ je polylogaritmicky ohraničená, pokud $f(n) = O(\log^k n)$ pro nějakou konstantu k.

- **Vztah k polynomům:**
  - Pro $a > 0$ a libovolné b platí $\log^b n = o(n^a)$.
  - **Jakákoli pozitivní polynomiální funkce roste rychleji než jakákoli polylogaritmická funkce.**
- **Faktoriál ($n!$):**
  - Definice pro celá čísla $n \geq 0$:
    - $0! = 1$
    - $n! = n \cdot (n-1)!$ pro $n > 0$
  - Tedy $n! = 1 \cdot 2 \cdot 3 \cdot \cdots \cdot n$.
  - **Odhad faktoriálu:**
    - Slabý horní odhad: $n! \leq n^n$.
    - Stirlingova aproximace: $n! = \sqrt{2\pi n}\left(\frac{n}{e}\right)^n \left(1 + \Theta\left(\frac{1}{n}\right)\right)$.
  - **Asymptotické vlastnosti:**
    - $n! = o(n^n)$
    - $n! = \omega(2^n)$
    - $\log(n!) = \Theta(n \log n)$
- **Funkcionální iterace ($f^{(i)}(n)$):**
  - Iterativní aplikace funkce $f(n)$ i-krát na počáteční hodnotu n.
  - **Definice:**
    - $f^{(0)}(n) = n$
    - $f^{(i)}(n) = f(f^{(i-1)}(n))$ pro $i > 0$
  - **Příklad:** Pokud $f(n) = 2n$, pak $f^{(i)}(n) = 2^i n$.
- **Iterovaný logaritmus ($\log^* n$):**
  - Definice: $\log^* n = \min\{i \geq 0 \mid \log^{(i)} n \leq 1\}$, kde $\log^{(i)} n$ je logaritmus aplikovaný i-krát.
  - Je to **velmi pomalu rostoucí funkce**.
  - **Příklady hodnot:**
    - $\log^* 2 = 1$
    - $\log^* 4 = 2$
    - $\log^* 16 = 3$
    - $\log^* 65536 = 4$
    - $\log^*(2^{65536}) = 5$
  - Málokdy se setkáme se vstupní velikostí n, pro kterou $\log^* n > 5$.
- **Fibonacciho čísla (F_i):**

- Definice pro $i \geq 0$:
  * $F_0 = 0$
  * $F_1 = 1$
  * $F_i = F_{i-1} + F_{i-2}$ pro $i \geq 2$
- **Posloupnost:** $0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, ...$
- **Vztah ke zlatému řezu (ϕ):**
  * Zlatý řez $\phi = \frac{1+\sqrt{5}}{2} \approx 1.61803$ a jeho konjugace $\hat{\phi} = \frac{1-\sqrt{5}}{2} \approx -0.61803$ jsou kořeny rovnice $x^2 = x + 1$.
  * Uzavřený tvar (Binetův vzorec): $F_i = \frac{\phi^i - \hat{\phi}^i}{\sqrt{5}}$.
- **Vlastnost zaokrouhlení:** $F_i = \lfloor \frac{\phi^i}{\sqrt{5}} + \frac{1}{2} \rfloor$.
- Fibonacciho čísla rostou **exponenciálně**.

6 Úvod do rekurzivních vztahů

- Mnoho algoritmů je rekurzivních (např. Merge Sort, Quick Sort).
- Jejich čas běhu se přirozeně popisuje **rekurzivními vztahy (rekurencemi)**.
- **Rekurence** je rovnice, která popisuje funkci pomocí své hodnoty na jiných argumentech.
- Příklad (Merge Sort): $T(n) = 2T(n/2) + \Theta(n)$
  - $2T(n/2)$: Dvě rekurzivní volání na poloviční velikosti vstupu.
  - $\Theta(n)$: Cena dělení a slučování.
- Rekurence je **dobře definovaná**, pokud existuje alespoň jedna funkce, která ji splňuje.

Cíl
Naučit se metody pro řešení rekurzivních vztahů a získat asymptotický odhad jejich časové složitosti.

Rekurence $T(n)$ je **algoritmická**, pokud pro každou dostatečně velkou konstantu $n_0 > 0$ platí následující dvě vlastnosti:

- **Pro všechna $n < n_0$: $T(n) = \Theta(1)$.**
  - $T(n)$ je shora i zdola ohraničeno kladnými konstantami pro vstupy menší než n_0.
- **Pro všechna $n \geq n_0$: Každé volání rekurze končí v bázovém případě v konečném počtu rekurzivních volání.**

  - Každý platný vstup musí algoritmus vyřešit v konečném čase.
  - Pokud by to neplatilo, algoritmus by nebyl korektní (např. nekonečná rekurzivní smyčka).

Doba běhu $T(n)$ je chápána jako definovaná pouze pro hodnoty n, pro které má algoritmus platné vstupy. Rekurence pro nejhorší případ korektního "rozděl a panuj" algoritmu je algoritmická.

- **Konvence 1: Předpokládaný bázový případ.**
  - Kdykoli je rekurence uvedena bez bázového případu, **předpokládáme, že je algoritmická**.
  - Tj., můžeme si vybrat libovolnou dostatečně velkou konstantu n_0, kde $T(n) = \Theta(1)$.
- **Konvence 2: Vynechání dolní a horní celé části.**
  - Asymptotická řešení algoritmických "rozděl a panuj" rekurencí se obvykle nemění, když se vynechají funkce $\lfloor \cdot \rfloor$ a $\lceil \cdot \rceil$.
  - Např. budeme často psát $T(n/2)$ místo $T(\lfloor n/2 \rfloor)$ nebo $T(\lceil n/2 \rceil)$.
- **Rekurence jako nerovnosti:**
  - Někdy se setkáte s rekurencemi jako nerovnostmi, např. $T(n) \leq 2T(n/2) + \Theta(n)$.
  - Taková rekurence uvádí pouze horní odhad na $T(n)$ a řešení vyjadřujeme pomocí O**-notace**.
  - Podobně pokud je nerovnost opačná, např. $T(n) \geq 2T(n/2) + \Theta(n)$, vyjadřujeme řešení pomocí Ω**-notace**.

1. **Substituční metoda:** Hádáme řešení a pak ho dokážeme indukcí.
2. **Metoda rekurzního stromu:** Rozvineme rekurenci do stromu a sečteme náklady.
3. **Master Theorem:** Poskytuje "kuchařku" pro řešení určitého typu rekurencí.

6.1 Substituční metoda

- **Krok 1: Hádej řešení.** To je nejtěžší část. Často se používá intuice ze stromů rekurze nebo zkušenost.
- **Krok 2: Dokaž indukčně.** Použije se matematická indukce k prokázání, že hádané řešení je správné.
  - **Indukční předpoklad:** Předpokládáme, že řešení platí pro menší hodnoty $k < n$.

  - **Indukční krok:** Dokážeme, že řešení platí pro n.
  - **Základní případ:** Ověříme pro malé n (např. $n = 1$).

- **Hádej řešení.** Z Merge Sortu víme, že by to mohlo být $\Theta(n \log n)$. Zkusíme ukázat, že $T(n) \leq cn \log n$ pro nějaké $c > 0$ a $n_0 > 0$.

- **Důkaz indukcí.**

  - **Indukční předpoklad:** Předpokládejme, že $T(k) \leq ck \log k$ pro všechna $n_0 \leq k < n$. Konkrétně pro $k = \lfloor n/2 \rfloor \geq n_0$, tj. máme $n \geq 2n_0$.
  - **Indukční krok:**

$$\begin{aligned} T(n) &\leq 2T(\lfloor n/2 \rfloor) + c'n \\ &\leq 2(c\lfloor n/2 \rfloor \log(\lfloor n/2 \rfloor)) + c'n \quad \text{(dle indukčního předpokladu)} \\ &\leq 2(c(n/2)\log(n/2)) + c'n \quad \text{(protože } \lfloor n/2 \rfloor \leq n/2 \text{ a log je rostoucí)} \\ &= cn \log(n/2) + c'n \\ &= cn(\log n - \log 2) + c'n \\ &= cn \log n - cn + c'n \\ &= cn \log n + (c' - c)n. \end{aligned}$$

  - Aby $T(n) \leq cn \log n$, potřebujeme $cn \log n + (c' - c)n \leq cn \log n$, pro $n \geq 2n_0$.
  - To znamená, že $(c' - c)n \leq 0$. Protože $n > 0$, musí platit $c' - c \leq 0 \implies c \geq c'$.

- Jak je to pro $n_0 \leq n < 2n_0$?

- **Základní případ:** Pro $n_0 > 1$ máme $n \log n > 0$.

- Zvolme $n_0 = 2$, pak, protože rekurence je algoritmická, máme $T(2) = \Theta(1)$ a $T(3) = \Theta(1)$.

- Zvolme $c = \max\{c', T(2), T(3)\}$, pak $T(2) \leq c < c \cdot 2 \log 2 = 2c$ a $T(3) \leq c < c \cdot 3 \log 3$.

- Celkem $T(n) \leq cn \log n$ pro $n \geq 2$.

Závěr
Tím jsme ukázali, že $T(n) = \mathcal{O}(n \log n)$. Stejným způsobem bychom dokázali $T(n) = \Omega(n \log n)$, a tedy $T(n) = \Theta(n \log n)$.

- Rekurence: $T(n) = T(\lfloor n/2 \rfloor) + T(\lceil n/2 \rceil) + \Theta(1)$

- **Hádej řešení:** Možná $\mathcal{O}(n)$? Zkusíme $T(n) \leq cn$ pro nějaké $c > 0$ a $n_0 > 0$.

- **Důkaz indukcí:**

$$\begin{aligned} T(n) &\leq T(\lfloor n/2 \rfloor) + T(\lceil n/2 \rceil) + c' \\ &\leq c\lfloor n/2 \rfloor + c\lceil n/2 \rceil + c' \quad \text{(dle indukčního předpokladu)} \\ &= c(\lfloor n/2 \rfloor + \lceil n/2 \rceil) + c' \\ &= cn + c' \end{aligned}$$

- Aby $T(n) \leq cn$, potřebujeme $cn + c' \leq cn$, což znamená $c' \leq 0$. Toto je **kontradikce**.
- **Problém:** Ignorování nižších členů v indukčním předpokladu.
- **Řešení:** Změň indukční předpoklad. Zkus $T(n) \leq cn - b$ pro nějaké $b > 0$.

$$\begin{aligned} T(n) &\leq (c\lfloor n/2 \rfloor - b) + (c\lceil n/2 \rceil - b) + c' \\ &= c(\lfloor n/2 \rfloor + \lceil n/2 \rceil) - 2b + c' \\ &= cn - 2b + c' \end{aligned}$$

- Chceme $cn - 2b + c' \leq cn - b$.
- To znamená $-2b + c' \leq -b \implies c' \leq b$.
- Zvolíme $b = c'$. Pak $T(n) \leq cn - c'$.

Závěr
$T(n) = \Theta(n)$. Důležité je být flexibilní s hádanou formou řešení (někdy je potřeba odečíst nižší řád nebo konstantu).

- Někdy správně odhadneme **těsnou asymptotickou mez** pro řešení rekurence, ale důkaz indukcí se nedaří.
- **Problém:** Induktivní předpoklad **není dostatečně silný**.
- **Trik:** Upravte odhad odečtením **členu nižšího řádu**. Matematika se pak často podaří.
- **Příklad rekurence:** $T(n) = 2T(n/2) + \Theta(1)$, definovaná na reálných číslech.
- **Počáteční odhad:** Zkusme $T(n) = O(n)$, tj. chceme ukázat, že $T(n) \leq cn$ pro $n \geq n_0$.
- **Dosazení odhadu do rekurence:**

$$\begin{aligned} T(n) &\leq 2(c \cdot n/2) + \Theta(1) \\ T(n) &\leq cn + \Theta(1) \end{aligned}$$

- **Problém:** Výraz $cn + \Theta(1)$ bohužel **nezaručuje**, že $T(n) \leq cn$ pro jakoukoli volbu c.
- Můžeme být v pokušení zkusit větší odhad (např. $O(n^2)$), ale ten by poskytl pouze volnou horní mez. Náš původní odhad $T(n) = O(n)$ je správný a těsný.
- Abychom ukázali, že je správný, musíme **posílit náš induktivní předpoklad**.
- **Intuitivně:** Náš odhad je téměř správný, lišíme se jen o $\Theta(1)$, což je člen nižšího řádu.
- **Použití triku:** Zkusme odečíst člen nižšího řádu od našeho předchozího odhadu:
  - Nový odhad: $T(n) \leq cn - d$, kde $d \geq 0$ je konstanta.
- **Dosazení nového odhadu do rekurence:**

  $T(n) \leq 2(c \cdot n/2 - d) + \Theta(1)$
  $T(n) \leq cn - 2d + \Theta(1)$
  $T(n) \leq cn - d - (d - \Theta(1))$ (přeskupení)
  $T(n) \leq cn - d$ (platí, pokud zvolíme d větší než anonymní horní konstanta skrytá v $\Theta(1)$)

- **Výsledek:** Odečtení členu nižšího řádu funguje!
- **Bázový případ:** Nesmíme zapomenout na bázový případ – je nutné zvolit konstantu c dostatečně velkou, aby $cn - d$ dominovalo implicitním bázovým případům.
- Myšlenka odečítání členu nižšího řádu může být **neintuitivní**. Koneckonců, pokud se matematika nedaří, neměli bychom náš odhad zvýšit?
- **Vysvětlení:**
  - Pokud rekurence obsahuje **více než jedno rekurzivní volání** a **přidáte** člen nižšího řádu k odhadu, pak se tento člen přičte jednou pro každé rekurzivní volání.
    - To vás pak od induktivní hypotézy ještě více **oddálí**.
  - Na druhou stranu, pokud **odečtete** člen nižšího řádu od odhadu, pak ho odečtete jednou za každé rekurzivní volání.
    - V příkladu jsme odečetli konstantu d dvakrát (protože koeficient $T(n/2)$ je 2).
    - Výsledkem byla nerovnost $T(n) \leq cn - d - (d - \Theta(1))$, a snadno jsme našli vhodnou hodnotu pro d.

- **Vyvarujte se použití asymptotické notace v induktivní hypotéze** pro substituční metodu, je to náchylné k chybám.
- **Příklad chybného "důkazu":** Pro rekurenci $T(n) = 2T(\lfloor n/2 \rfloor) + \Theta(n)$ můžeme falešně "dokázat", že $T(n) = O(n)$, pokud neuváženě přijmeme $T(n) = O(n)$ jako naši induktivní hypotézu:

$$\begin{aligned} T(n) &\leq 2 \cdot O(\lfloor n/2 \rfloor) + \Theta(n) \\ T(n) &\leq 2 \cdot O(n) + \Theta(n) \\ T(n) &= O(n) \quad \text{— Špatně!} \end{aligned}$$

- **Problém tohoto uvažování:** Konstanta skrytá pod O-notací se **mění**.
- **Odhalení omylu (s explicitní konstantou):**
  - Pro induktivní hypotézu předpokládejme $T(n) \leq cn$ pro všechna $n \geq n_0$, kde $c, n_0 > 0$ jsou konstanty.
  - Opakování prvních dvou kroků v řetězci nerovností dává:

$$\begin{aligned} T(n) &\leq 2(c\lfloor n/2 \rfloor) + \Theta(n) \\ T(n) &\leq cn + \Theta(n) \end{aligned}$$

  - Nyní $cn + \Theta(n)$ je sice $O(n)$, ale konstanta skrytá pod O-notací musí být **větší než** c, protože anonymní funkce skrytá pod $\Theta(n)$ je asymptoticky pozitivní.
  - **Nemůžeme učinit třetí krok a dojít k závěru**, že $cn + \Theta(n) \leq cn$, čímž se odhaluje omyl.
- **Důležité pravidlo:** Při použití substituční metody (nebo obecně matematické indukce) musíte dbát na to, aby **konstanty skryté pod jakoukoli asymptotickou notací byly stejné po celou dobu důkazu**.
- **Doporučení:** Nejlepší je **vyhnout se asymptotické notaci** ve vaší induktivní hypotéze a **pojmenovat konstanty explicitně**.
- **Další chybný způsob** použití substituční metody:
  - Předpokládejme $T(n) \leq cn$ a pak argumentujeme:

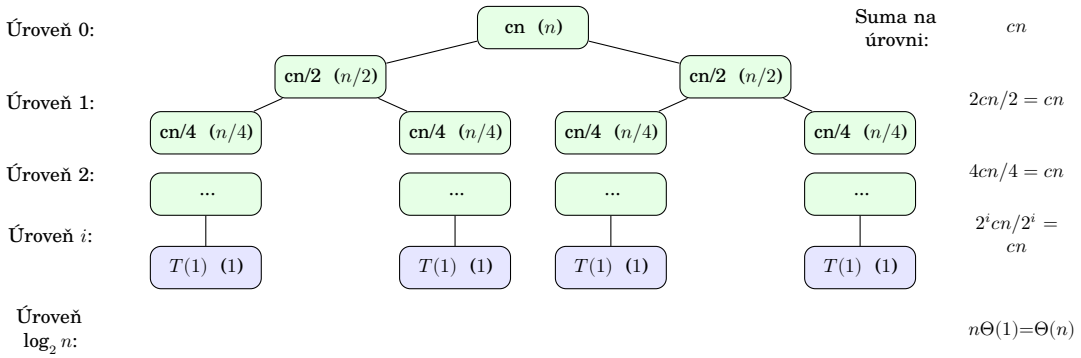
$$\begin{aligned} T(n) &\leq 2(c\lfloor n/2 \rfloor) + \Theta(n) \\ T(n) &\leq cn + \Theta(n) \\ T(n) &= O(n) \quad \text{— Špatně! (protože } c \text{ je kladná konstanta)} \end{aligned}$$

- **Příčina chyby:** Pramení z rozdílu mezi naším **cílem** – dokázat, že $T(n) = O(n)$ – a naší **induktivní hypotézou** – dokázat, že $T(n) \leq cn$.

- **Klíčové pravidlo:** Při použití substituční metody, nebo v jakémkoli induktivním důkazu, **musíte dokázat přesné znění induktivní hypotézy**.
- **Příklad:** Chcete-li ukázat, že $T(n) = O(n)$, musíte explicitně dokázat, že $T(n) \leq cn$.

6.2 Metoda rekurzního stromu

- Rekurzní strom je vizuální reprezentace rekurzivního vztahu.
- Každý uzel reprezentuje náklady na dané rekurzivní volání (dělení a slučování v dané úrovni).
- **Kroky:**
  1. **Nakreslete strom:** Rozkreslete první 2-3 úrovně stromu.
  2. **Určete náklady na každé úrovni:** Sečtěte náklady všech uzlů na dané úrovni.
  3. **Určete počet úrovní (hloubku stromu):** Dokud nedosáhneme základního případu.
  4. **Sečtěte náklady všech úrovní:** Včetně nákladů na listové uzly (základní případy).
- Tato metoda je dobrá pro **generování dobrého hádání** pro substituční metodu.



Celková suma: $\sum_{i=0}^{\log_2 n - 1} cn + \text{listy} = cn \log n + \Theta(n) = \Theta(n \log n)$

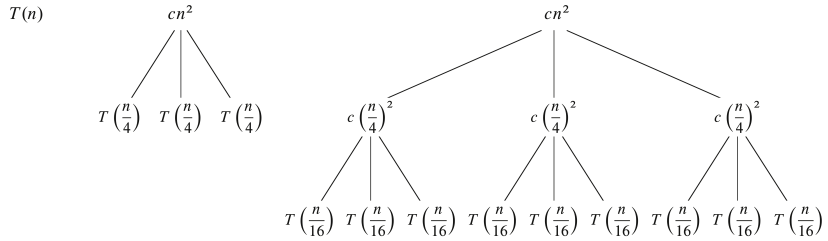
- **Co je strom rekurze:** Každý uzel reprezentuje náklady na jeden podproblém v rámci sady rekurzivních volání funkce.
- **Účel:**

  - Především slouží ke **generování intuice pro dobrý odhad** řešení rekurence.
  - Tento odhad je pak třeba **ověřit substituční metodou**.
  - Při pečlivém sestavení a součtu nákladů může strom rekurze sloužit i jako **přímý důkaz**.

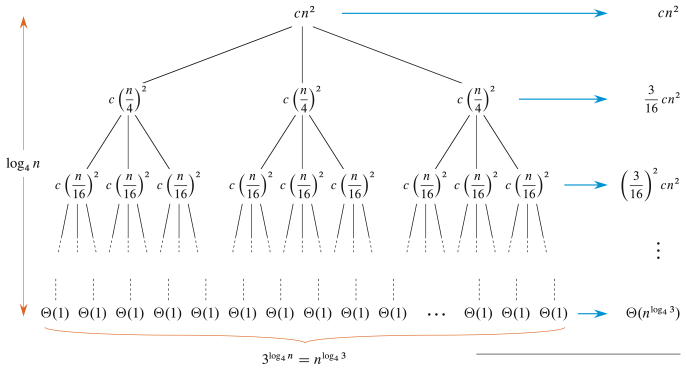
- **Jak metoda funguje:**
  - Typicky se sčítají náklady v rámci každé úrovně stromu, aby se získaly náklady na úroveň.
  - Poté se sečtou náklady všech úrovní, aby se zjistily celkové náklady všech úrovní rekurze.

- **Rekurence:** $T(n) = 3T(n/4) + \Theta(n^2)$
  - Pro analýzu použijeme $T(n) = 3T(n/4) + cn^2$, kde $c > 0$ je horní konstanta z $\Theta(n^2)$.

- **Konstrukce stromu:**
  - **Kořen:** Náklady cn^2. Má tři podstromy $T(n/4)$.
  - **Úroveň 1:** Tři uzly, každý s náklady $c(n/4)^2$. Celkové náklady na úroveň: $3c(n/4)^2 = \frac{3}{16}cn^2$.



- **Hloubka i:** Počet uzlů je 3^i. Velikost podproblému je $n/4^i$. Náklady na uzel jsou $c(n/4^i)^2$.
- **Celkové náklady na úroveň i:** $3^i \cdot c(n/4^i)^2 = 3^i \cdot c\frac{n^2}{16^i} = \left(\frac{3}{16}\right)^i cn^2$.
- **Bázový případ:** Pro zjednodušení předpokládáme $T(1) = \Theta(1)$ pro $n < n_0$.
- **Výška stromu:** Velikost podproblému $n/4^i$ dosáhne 1, když $i = \log_4 n =$ výška stromu.
- **Listy:** Na hloubce $\log_4 n$ je $3^{\log_4 n} = n^{\log_4 3}$ listů, každý stojí $\Theta(1)$.



- **Součet nákladů všech úrovní:**

$$\begin{aligned} T(n) &= \sum_{i=0}^{\log_4 n - 1} \left(\frac{3}{16}\right)^i cn^2 + \Theta(n^{\log_4 3}) \\ &< \sum_{i=0}^{\infty} \left(\frac{3}{16}\right)^i cn^2 + \Theta(n^{\log_4 3}) \\ &= \frac{1}{1-3/16} cn^2 + \Theta(n^{\log_4 3}) \\ &= \frac{16}{13} cn^2 + \Theta(n^{\log_4 3}) \end{aligned}$$

  - Jelikož $\log_4 3 \approx 0.792 < 2$, term cn^2 dominuje celkové náklady.
  - Odhad získaný metodou rekurzivního stromu je $T(n) = O(n^2)$.

- **Ověření substituční metodou:** Chceme ukázat, že $T(n) \leq dn^2$ pro nějakou konstantu $d > 0$.

$$\begin{aligned} T(n) &\leq 3T(n/4) + cn^2 \\ &\leq 3d(n/4)^2 + cn^2 \\ &= \frac{3}{16} dn^2 + cn^2 \\ &\leq dn^2 \end{aligned}$$

  - Poslední nerovnost platí, pokud zvolíme $d \geq \frac{16}{13} c$.
  - Pro bázový případ zvolte d dostatečně velké, aby dn^2 dominovalo $T(n) = \Theta(1)$ pro $n < n_0$.
  - Pak $dn^2 \geq d \geq T(n)$ pro $1 \leq n < n_0$.

- **Závěr:** Odhad $T(n) = O(n^2)$ je správný a je to také těsná horní mez ($\Omega(n^2)$).

- **Rekurence:** $T(n) = T(n/3) + T(2n/3) + \Theta(n)$.

- Pro analýzu použijeme $T(n) = T(n/3) + T(2n/3) + cn$.

- **Charakteristika stromu:**
  - Je **nevyvážený**: Cesty z kořene do listů mají různé délky.
  - Jít vlevo produkuje podproblém třetinové velikosti; jít vpravo produkuje podproblém dvoutřetinové velikosti.

- **Výška stromu:**
  - Nejdelší cesta vede vždy doprava.
  - Výška h je taková, že $(2/3)^h n \approx n_0$, což znamená $h = \Theta(\log_{3/2}(n/n_0)) = \Theta(\log n)$.
    * $T(n) = \Theta(1)$ dává konstantu n_0, $\Theta(n)$ dává další konstantu n_0, naše n_0 je větší z těchto dvou

- **Náklady na úroveň:**
  - Součet nákladů na každé úrovni (vnitřních uzlů) je cn.
  - Například na úrovni 1: $c(n/3) + c(2n/3) = cn$.
  - Jelikož každá úroveň má náklady cn a výška stromu je $\Theta(\log n)$, celkové náklady **vnitřních uzlů** jsou $O(n \log n)$.

- **Listy**
  - 2^h listů, tj. až $2^{\lfloor \log_{3/2} n \rfloor + 1} \leq 2n^{\log_{3/2} 2} = O(n^{1,71})$, což je asymptoticky větší než $O(n \log n)$.

- **Náklady listů:**
  - Každý list stojí $\Theta(1)$.
  - Pro určení počtu listů $L(n)$ v rekurzivním stromě můžeme definovat rekurenci:

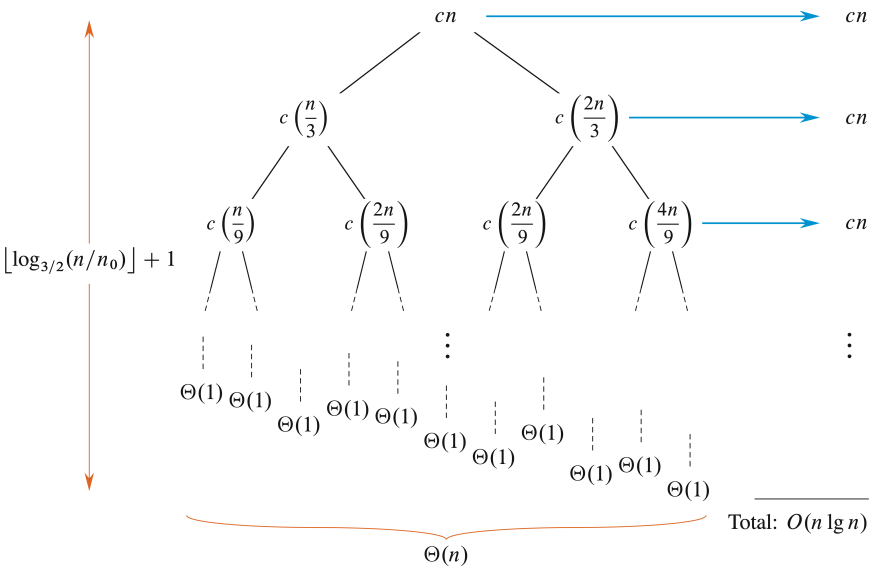
$$L(n) = \begin{cases} 1 & \text{pokud } n < n_0 \\ L(n/3) + L(2n/3) & \text{pokud } n \geq n_0 \end{cases}$$

  - Řešení této rekurence (pomocí substituční metody s hypotézou $L(n) \leq dn$):

$$L(n) = L(n/3) + L(2n/3) \leq d(n/3) + d(2n/3) = dn$$

  - Platí pro libovolné $d > 0$. Pro bázový případ $L(n) = 1$ pro $0 < n < n_0$ stačí $d = 1$.
  - Celkový počet listů je tedy $O(n)$.
  - Celkové náklady listů jsou $L(n) \cdot \Theta(1) = \Theta(n)$.

- **Celkové řešení:**
  - Celkové náklady = náklady vnitřních uzlů + náklady listů.
  - $T(n) = O(n \log n) + \Theta(n) = O(n \log n)$.
- **Doporučení:** Vždy **ověřte** jakýkoli odhad získaný metodou rekurzivního stromu pomocí substituční metody, zejména pokud byly provedeny zjednodušující předpoklady.



6.3 Master Theorem

- Nejjednodušší a nejrychlejší metoda pro řešení rekurzivních vztahů, které mají specifický tvar.
- Vztahuje se na rekurence tvaru

$$T(n) = aT(n/b) + f(n)$$

  kde:
  - $a \geq 1$: Počet rekurzivních podproblémů.
  - $b > 1$: Faktor, o kolik se zmenší velikost vstupu v každém podproblému.
  - $f(n)$: Cena dělení a slučování na každé úrovni rekurze.
- $aT(n/b)$ ve skutečnosti znamená $a'T(\lfloor n/b \rfloor) + a''T(\lceil n/b \rceil)$ kde $a' \geq 0$ a $a'' \geq 0$ jsou konstanty takové, že $a = a' + a''$.
- Master Theorem porovnává $f(n)$ s $n^{\log_b a}$.

Věta 6.1 (Master Theorem)**.** *Nechť $a \geq 1$ a $b > 1$ jsou konstanty a $f(n)$ je asymptoticky kladná funkce. Pak*

$$T(n) = aT(n/b) + f(n)$$

má následující asymptotické řešení:

1. *Pokud $f(n) = \mathcal{O}(n^{\log_b a - \epsilon})$ pro nějakou konstantu $\epsilon > 0$, pak $T(n) = \Theta(n^{\log_b a})$.*
2. *Pokud $f(n) = \Theta(n^{\log_b a} \log^k n)$ pro nějakou konst. $k \geq 0$, pak $T(n) = \Theta(n^{\log_b a} \log^{k+1} n)$.*
3. *Pokud $f(n) = \Omega(n^{\log_b a + \epsilon})$ pro nějakou konstantu $\epsilon > 0$, a pokud navíc $af(n/b) \leq cf(n)$ pro nějakou konstantu $c < 1$ a všechna dostatečně velká n, pak $T(n) = \Theta(f(n))$.*

- $n^{\log_b a}$: Reprezentuje náklady na *listy rekurzního stromu* (nebo přesněji náklady na rekurzivní volání samotná).
- $f(n)$: Reprezentuje náklady na *dělení a slučování* na každé úrovni.
- **Případ 1:** Cena listů dominuje.
  - $f(n)$ je polynomiálně menší než $n^{\log_b a}$.
- **Případ 2:** Náklady jsou "vyvážené".
  - $f(n)$ je asymptoticky stejné jako $n^{\log_b a}$.
  - Přidáváme $\log n$ faktor kvůli logaritmickému počtu úrovní.
- **Případ 3:** Cena dělení/slučování dominuje.
  - $f(n)$ je polynomiálně větší než $n^{\log_b a}$.
  - Vyžaduje se dodatečná podmínka pravidelnosti ($af(n/b) \leq cf(n)$).
- $a = 9$, $b = 3$, $f(n) = n$.
- Vypočítáme $n^{\log_b a} = n^{\log_3 9} = n^2$.
- Porovnáme $f(n)$ s $n^{\log_b a}$:
  - $f(n) = n$
  - $n^{\log_b a} = n^2$
- Vidíme, že $f(n) = n = \mathcal{O}(n^{2-\epsilon})$ pro $\epsilon = 1$ (protože $n = \mathcal{O}(n^{2-1})$).
- Spadáme do **Případu 1**.
- Řešení: $T(n) = \Theta(n^{\log_b a}) = \Theta(n^2)$.

- $a = 3$, $b = 4$, $f(n) = n \log n$.
- Vypočítáme $n^{\log_b a} = n^{\log_4 3}$. Protože $\log_4 3 \approx 0.793$, máme $n^{\log_4 3} = n^{0.793}$.
- Porovnáme $f(n)$ s $n^{\log_b a}$:
  - $f(n) = n \log n$
  - $n^{\log_b a} = n^{0.793}$
- Vidíme, že $f(n)$ je polynomiálně větší než $n^{\log_b a}$.
  - Např. $n \log n = \Omega(n^{0.793+\epsilon})$ (zvolíme $\epsilon = 0.207$, pak $n \log n$ je větší než n).
- Spadáme do **Případu 3**. Musíme ověřit podmínku pravidelnosti: $af(n/b) \leq cf(n)$.
  - $af(n/b) = 3(n/4) \log(n/4) = (3/4)n(\log n - \log 4)$
  - Chceme, aby $(3/4)n(\log n - \log 4) \leq cn \log n$.
  - $(3/4)n \log n - (3/4)n \log 4 \leq cn \log n$.
  - Pokud zvolíme $c = 3/4$, pak $(3/4)n \log n - (3/4)n \log 4 \leq (3/4)n \log n$.
  - Platí pro dostatečně velké n (protože $(3/4)n \log 4$ je kladné, takže levá strana je menší než $(3/4)n \log n$).
- Řešení: $T(n) = \Theta(f(n)) = \Theta(n \log n)$.
- **Příklad:** $T(n) = T(2n/3) + 1$
  - $a = 1, b = 3/2$.
  - $n^{\log_{3/2} 1} = n^0 = 1$.
  - $f(n) = 1$.
  - Protože $f(n) = 1 = \Theta(n^{\log_b a} \log^0 n) = \Theta(1)$, platí **Případ 2**.
  - **Řešení:** $T(n) = \Theta(\log n)$.
- **Příklad:** $T(n) = 2T(n/2) + n \log n$
  - $a = 2, b = 2$.
  - $n^{\log_2 2} = n^1 = n$.
  - $f(n) = n \log n$.
  - Protože $f(n) = n \log n = \Theta(n^{\log_b a} \log^1 n)$, platí **Případ 2**.
  - **Řešení:** $T(n) = \Theta(n \log^2 n)$.
- Master teorem **nepokrývá všechny možnosti** rekurencí tvaru $T(n) = aT(n/b) + f(n)$.
- **Situace, kdy nelze Master theorem použít:**

- **Asymptotické porovnání:** $n^{\log_b a}$ a $f(n)$ se nedají asymptoticky porovnat (tj. $f(n)$ není ani O, ani Ω vůči $n^{\log_b a}$).
- **Mezery mezi případy:**
  - **Mezi Případem 1 a 2:** Pokud $f(n) = o(n^{\log_b a})$, ale $n^{\log_b a}$ neroste *polynomiálně* rychleji než $f(n)$.
  - **Mezi Případem 2 a 3:** Pokud $f(n) = \omega(n^{\log_b a})$, ale $f(n)$ neroste *polynomiálně* rychleji než $n^{\log_b a}$ (pouze polylogaritmicaly rychleji).
- **Selhání podmínky regularity** v Případě 3 ($af(n/b) \leq cf(n)$ pro nějaké $c < 1$ a dostatečně velké n).

Věta 6.2 (Master Theorem)**.** *..., pak* $T(n) = aT(n/b) + f(n)$ *má následující asymptotické řešení:*

1. *Pokud* $f(n) = \mathcal{O}(n^{\log_b a - \epsilon})$ *pro nějakou konstantu* $\epsilon > 0$*, pak* $T(n) = \Theta(n^{\log_b a})$.
2. *Pokud* $f(n) = \Theta(n^{\log_b a} \log^k n)$ *pro nějakou konst.* $k \geq 0$*, pak* $T(n) = \Theta(n^{\log_b a} \log^{k+1} n)$.
3. *Pokud* $f(n) = \Omega(n^{\log_b a + \epsilon})$ *pro nějakou konstantu* $\epsilon > 0$*, a navíc* $af(n/b) \leq cf(n)$ *...*

Příklad 1: $T(n) = 2T(n/2) + n/\log n$

- $a = 2, b = 2, n^{\log_b a} = n^{\log_2 2} = n$.
- Porovnáme $f(n)$ s n: $\frac{n/\log n}{n} = \frac{1}{\log n} \to 0$ pro $n \to \infty$, tj. $f(n) = n/\log n = o(n)$.
- Víme, že $\log n = o(n^{\varepsilon})$ pro každou konstantu $\varepsilon > 0$, tj. $1/\log n = \omega(n^{-\varepsilon})$.
- To znamená, že $f(n) = \omega(n^{1-\epsilon}) = \omega(n^{\log_b a - \epsilon})$ pro *jakékoli* $\epsilon > 0$, což porušuje Případ 1.
- $f(n)$ roste logaritmicky pomaleji než n, ale **ne polynomiálně pomaleji**.
- Ač $f(n) = \Theta(n^{\log_b a} \log^k n)$ pro $k = -1$, Případ 2 vyžaduje $k \geq 0$.
- Proto Master Theorem nelze použít. (Řešení by bylo $T(n) = \Theta(n \log\log n)$.)

Příklad 2: $T(n) = T(n/2) + n^2$

- $a = 1, b = 2, n^{\log_b a} = n^{\log_2 1} = n^0 = 1$.
- $f(n) = n^2$.
- Porovnáme $f(n)$ s $n^{\log_b a}$: $n^2 = \Omega(n^{0+1})$ je polynomiálně větší než 1 (Případ 3).

- Ověřit podmínku pravidelnosti: $af(n/b) \leq cf(n)$
- $1 \cdot (n/2)^2 \leq cn^2$, tj. $n^2/4 \leq cn^2$.
- Zvolíme $c = 1/4$, což je menší než 1. Platí.
- Řešení: $T(n) = \Theta(f(n)) = \Theta(n^2)$. (Tento příklad funguje.)
- **Rekurzivní vztahy:** Matematický popis času běhu rekurzivních algoritmů.
- **Substituční metoda:**
  - Hádej řešení.
  - Dokaž indukcí (nezapomeň na základní případ a možnost odečíst nižší členy).
- **Metoda rekurzního stromu:**
  - Vizuální, pro generování hádání.
  - Sečtení nákladů na úrovních a určení hloubky stromu.
- **Master Theorem:**
  - "Kuchařka" pro $T(n) = aT(n/b) + f(n)$.
  - Porovnává $f(n)$ s $n^{\log_b a}$ ve třech případech.
  - Velmi rychlá a často použitelná metoda.
- Tyto metody jsou zásadní pro formální analýzu a porovnávání složitosti algoritmů.
- Rekurzivní vztah: $T(n) = 2T(n/2) + \Theta(n)$
- Porovnání s $T(n) = aT(n/b) + f(n)$:
  - $a = 2$
  - $b = 2$
  - $f(n) = \Theta(n)$
- Vypočítáme $n^{\log_b a}$: $n^{\log_2 2} = n^1 = n$.
- Nyní porovnáme $f(n)$ s $n^{\log_b a} = n$: $f(n) = \Theta(n)$ a $n^{\log_b a} = \Theta(n)$.
- Toto spadá do **Případu 2** Master Theoremu (pro $k = 0$).
- **Závěr:** Časová složitost Merge Sortu je $T(n) = \Theta(n \log n)$.

Nechť $a \geq 1$, $b > 1$ a $c \geq 0$ jsou konstanty a nechť $T(n)$ je definována na nezáporných číslech rekurentní rovnicí

$$T(n) = aT(n/b) + \Theta(n^c)$$

Potom platí

$$T(n) = \begin{cases} \Theta(n^c) & \text{pokud } a < b^c \quad \textit{případ 1} \\ \Theta(n^c \log n) & \text{pokud } a = b^c \quad \textit{případ 2} \\ \Theta(n^{\log_b a}) & \text{pokud } a > b^c \quad \textit{případ 3} \end{cases}$$

věta platí i pro $\mathcal{O}$ a Ω

- $T(n) = 4T(n/2) + 1 \implies T(n) = \Theta(n^2)$ případ 3, $a = 4, b = 2, c = 0,\ 4 > 2^0$
- $T(n) = 4T(n/2) + n \implies T(n) = \Theta(n^2)$ případ 3, $a = 4, b = 2, c = 1,\ 4 > 2^1$
- $T(n) = 4T(n/2) + n^2 \implies T(n) = \Theta(n^2 \log n)$ případ 2, $a = 4, b = 2, c = 2,\ 4 = 2^2$
- $T(n) = 4T(n/2) + n^3 \implies T(n) = \Theta(n^3)$ případ 1, $a = 4, b = 2, c = 3,\ 4 < 2^3$
- $T(n) = 2T(n/2)+1 \implies T(n) = \Theta(n)$ případ 3, $a = 2, b = 2, c = 0,\ 2 > 2^0$
- $T(n) = 2T(n/2) + n \implies T(n) = \Theta(n \log n)$ případ 2, $a = 2, b = 2, c = 1,\ 2 = 2^1$
- $T(n) = 2T(n/2) + n^2 \implies T(n) = \Theta(n^2)$ případ 1, $a = 2, b = 2, c = 2,\ 2 < 2^2$
- $T(n) = 2T(n/2) + n^3 \implies T(n) = \Theta(n^3)$ případ 1, $a = 2, b = 2, c = 3,\ 2 < 2^3$
- Mějme $n \times n$ matice $A = (a_{ik})$ a $B = (b_{jk})$. Jejich součin $C = A \cdot B$ je také $n \times n$ matice.
- Prvek c_{ij} matice C je dán vzorcem:

$$c_{ij} = \sum_{k=1}^{n} a_{ik} \cdot b_{kj}$$

MATRIX-MULTIPLY(A, B, C, n)

```
1  for i = 1 to n                    // compute entries in each of n rows
2     for j = 1 to n                 // compute n entries in row i
3        for k = 1 to n
4           c_ij = c_ij + a_ik · b_kj   // add in one more term of equation (4.1)
```

- **Procedura MATRIX-MULTIPLY(A, B, C, n):**
  - Přidává maticový součin $A \cdot B$ k matici C, výsledek ukládá do C (tj. $C \leftarrow C + A \cdot B$).
  - Pokud je potřeba jen $A \cdot B$, inicializujte C na nuly ($\Theta(n^2)$ čas), což je asymptoticky dominováno násobením.
- **Analýza složitosti:**
  - Trojitě vnořené 'for' cykly, každý běží n iterací.
  - Každá operace na řádku 4 trvá konstantní čas.
  - Celková doba je $\Theta(n^3)$.
- Pro $n > 1$ (předpokládáme, že n je mocnina 2), rozdělíme $n \times n$ matice na čtyři $n/2 \times n/2$ podmatice:

$$A = \begin{pmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{pmatrix}, \quad B = \begin{pmatrix} B_{11} & B_{12} \\ B_{21} & B_{22} \end{pmatrix}, \quad C = \begin{pmatrix} C_{11} & C_{12} \\ C_{21} & C_{22} \end{pmatrix}$$

- Maticový součin lze zapsat:

$$\begin{pmatrix} C_{11} & C_{12} \\ C_{21} & C_{22} \end{pmatrix} = \begin{pmatrix} A_{11}B_{11} + A_{12}B_{21} & A_{11}B_{12} + A_{12}B_{22} \\ A_{21}B_{11} + A_{22}B_{21} & A_{21}B_{12} + A_{22}B_{22} \end{pmatrix}$$

$$\begin{aligned} C_{11} &= A_{11}B_{11} + A_{12}B_{21} \\ C_{12} &= A_{11}B_{12} + A_{12}B_{22} \\ C_{21} &= A_{21}B_{11} + A_{22}B_{21} \\ C_{22} &= A_{21}B_{12} + A_{22}B_{22} \end{aligned}$$

- Tyto rovnice zahrnují: **Osm násobení** a **čtyři sčítání** $n/2 \times n/2$ matic.
- Procedura MATRIX-MULTIPLY-RECURSIVE(A,B,C,n) implementuje předchozí.
- Počítá $C \leftarrow C + A \cdot B$.

```
MATRIX-MULTIPLY-RECURSIVE(A, B, C, n)
1  if n == 1
2      // Base case.
3      c_11 = c_11 + a_11 · b_11
4      return
5  // Divide.
6  partition A, B, and C into n/2 × n/2 submatrices
       A_11, A_12, A_21, A_22; B_11, B_12, B_21, B_22;
       and C_11, C_12, C_21, C_22; respectively
7  // Conquer.
8  MATRIX-MULTIPLY-RECURSIVE(A_11, B_11, C_11, n/2)
9  MATRIX-MULTIPLY-RECURSIVE(A_11, B_12, C_12, n/2)
10 MATRIX-MULTIPLY-RECURSIVE(A_21, B_11, C_21, n/2)
11 MATRIX-MULTIPLY-RECURSIVE(A_21, B_12, C_22, n/2)
12 MATRIX-MULTIPLY-RECURSIVE(A_12, B_21, C_11, n/2)
13 MATRIX-MULTIPLY-RECURSIVE(A_12, B_22, C_12, n/2)
14 MATRIX-MULTIPLY-RECURSIVE(A_22, B_21, C_21, n/2)
15 MATRIX-MULTIPLY-RECURSIVE(A_22, B_22, C_22, n/2)
```

- **Rekurence pro dobu běhu $T(n)$:**
  - Bázový případ ($n = 1$): $T(1) = \Theta(1)$.
  - Rekurzivní případ ($n > 1$): Rozdělení trvá $\Theta(1)$, následuje 8 rekurzivních volání.
  - $T(n) = 8T(n/2) + \Theta(1)$
- Pomocí Master theoremu má tato rekurence řešení $T(n) = \Theta(n^3)$.
- Stejná asymptotická složitost jako MATRIX-MULTIPLY.
- **Proč $\Theta(n^3)$ a ne rychlejší?**
  - Rekurzivní strom pro tuto rekurenci má osm potomků na uzel (oproti dvěma u Merge Sortu).
  - To vede k rekurzivnímu stromu s mnohem více listy, což výrazně zvyšuje složitost.
- Mnozí matematici předpokládali, že není možné násobit matice v čase $o(n^3)$.
- V roce 1969 V. Strassen publikoval pozoruhodný rekurzivní algoritmus.
- **Strassenův algoritmus** běží v čase $\Theta(n^{\log 7})$.
  - Jelikož $\log 7 \approx 2.807$, Strassenův algoritmus běží v čase $O(n^{2.81})$, což je asymptoticky lepší než $\Theta(n^3)$.
- **Klíčová myšlenka:** Používá metodu ”rozděl a panuj”, ale **snižuje počet rekurzivních násobení**.
  - Místo osmi rekurzivních násobení $n/2 \times n/2$ matic provádí Strassenův algoritmus **pouze sedm**.

- **Motivace (trik z algebry):** Výpočet $x^2 - y^2$.
  - Přímý způsob: $x \cdot x - y \cdot y \rightsquigarrow$ 2 násobení, 1 odčítání.
  - Alternativní způsob: $(x+y)(x-y) \rightsquigarrow$ 1 násobení, 2 sčítání/odčítání.
  - Pro velké matice je cena násobení vyšší než cena sčítání, takže druhá metoda překonává první.
- **Vytvoření 10 matic S_i (čas $\Theta(n^2)$):**

$$S_1 = B_{12} - B_{22} \quad S_2 = A_{11} + A_{12} \quad S_3 = A_{21} + A_{22} \quad S_4 = B_{21} - B_{11}$$
$$S_5 = A_{11} + A_{22} \quad S_6 = B_{11} + B_{22} \quad S_7 = A_{12} - A_{22} \quad S_8 = B_{21} + B_{22}$$
$$S_9 = A_{11} - A_{21} \quad S_{10} = B_{11} + B_{12}$$

- **Rekurzivní výpočet 7 matic P_i (čas $7T(n/2)$):**

$$P_1 = A_{11} \cdot S_1 \quad P_2 = S_2 \cdot B_{22} \quad P_3 = S_3 \cdot B_{11} \quad P_4 = A_{22} \cdot S_4$$
$$P_5 = S_5 \cdot S_6 \quad P_6 = S_7 \cdot S_8 \quad P_7 = S_9 \cdot S_{10}$$

- **Krok 4: Aktualizace čtyř podmatic C_{ij} matice C ($\Theta(n^2)$ času):**

$$C_{11} \leftarrow C_{11} + P_5 + P_4 - P_2 + P_6 \quad C_{12} \leftarrow C_{12} + P_1 + P_2$$
$$C_{21} \leftarrow C_{21} + P_3 + P_4 \quad C_{22} \leftarrow C_{22} + P_5 + P_1 - P_3 - P_7$$

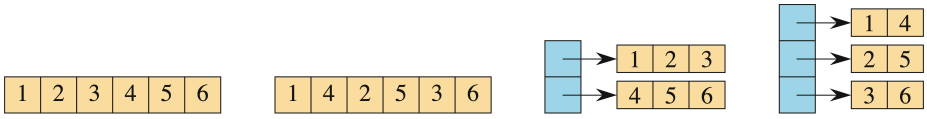
- Rekurence $T(n) = 7T(n/2) + \Theta(n^2)$ přesně charakterizuje jeho dobu běhu, která je asymptoticky lepší než $\Theta(n^3)$ algoritmy.
- **Rekurzivní násobení matic:** $T(n) = 8T(n/2) + \Theta(1)$
  - $a = 8, b = 2$, $n^{\log_2 8} = n^3$.
  - $f(n) = \Theta(1)$.
  - Protože $f(n) = O(n^{3-\epsilon})$ pro libovolné $\epsilon \in (0, 3)$, platí **Případ 1**.
  - **Řešení:** $T(n) = \Theta(n^3)$.
- **Strassenův algoritmus:** $T(n) = 7T(n/2) + \Theta(n^2)$
  - $a = 7, b = 2$, $n^{\log 7} \approx n^{2.807}$.
  - $f(n) = \Theta(n^2)$.
  - Protože $f(n) = O(n^{\log 7 - \epsilon})$ pro např. $\epsilon = 0.807$, platí **Případ 1**.
  - **Řešení:** $T(n) = \Theta(n^{\log 7})$.
- **Insertion Sort:** Jednoduchý, $O(n^2)$ v nejhorším případě, $O(n)$ v nejlepším. Dobrý pro malé/téměř seřazené vstupy. Korektnost pomocí invariantu cyklu.
- **Divide-and-Conquer:** Klíčová technika: Rozděl a panuj.
- **Merge Sort:** Klasický příklad Rozděl a panuj. Vždy $O(n \log n)$.
- **Analýza rekurzivních algoritmů:** Rekurzní vztahy a Master Theorem jsou mocné nástroje.

7 Datové struktury

- **Definice:** Způsob uložení a organizace dat v počítači, který umožňuje efektivní přístup a modifikaci.
- Kombinují data s operacemi, které na nich lze provádět.
- Volba vhodné datové struktury je klíčová pro efektivitu algoritmů.
- Představíme základní datové struktury:
  - Pole
  - Matice
  - Zásobníky (Stacks)
  - Fronty (Queues)
  - Spojové seznamy (Linked Lists)
  - Kořenové stromy (Rooted Trees)
- Předpokládáme, že pole je uloženo jako souvislá sekvence bytů v paměti.
- Přístup k libovolnému prvku pole trvá **konstantní čas** ($O(1)$) bez ohledu na index, za předpokladu RAM modelu.
- Většina programovacích jazyků vyžaduje, aby každý prvek pole měl stejnou velikost.
- Pokud prvky pole mohou mít různou velikost, pak místo samotných objektů jsou v poli uloženy **ukazatele na objekty**.
  - Velikost ukazatele je obvykle stejná bez ohledu na to, na co ukazatel odkazuje.
  - Přístup k objektu v takovém poli vyžaduje:
    1. Nalezení adresy ukazatele (konstantní čas).
    2. Následování ukazatele k přístupu k samotnému objektu (konstantní čas).
- Matice, neboli dvourozměrné pole, jsou typicky reprezentovány jedním nebo více jednorozměrnými poli.
- **Dva nejčastější způsoby uložení matice** $m \times n$**:**
  1. **řádkové uspořádání:** Matice je uložena řádek po řádku.
     - Příklad matice $M = \begin{pmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \end{pmatrix}$: Uloženo jako $1, 2, 3, 4, 5, 6$.
     - Index prvku $M[i, j]$ (při 0-indexování): $ni + j$.
  2. **sloupcové uspořádání:** Matice je uložena sloupec po sloupci.

  - Příklad matice $M = \begin{pmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \end{pmatrix}$: Uloženo jako $1, 4, 2, 5, 3, 6$.
  - Index prvku $M[i, j]$ (při 0-indexování): $i + mj$.

- Matici lze uložit i pomocí **více polí**.
  - **Jedno pole na řádek/sloupec:** Každý řádek (nebo sloupec) je uložen ve svém vlastním jednorozměrném poli.
  - **Pole ukazatelů:** Další pole obsahuje ukazatele na tato pole řádků/sloupců.

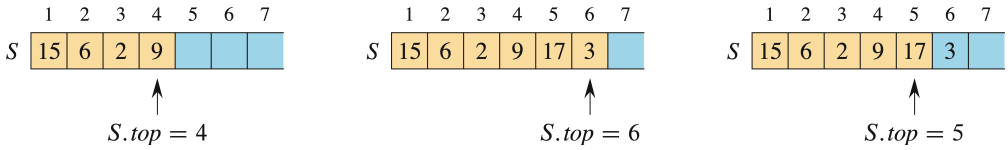


(a) (b) (c) (d)

- **Výhody/Nevýhody:**
  - **Jednorozměrné reprezentace:** Obvykle efektivnější na moderních strojích (lepší využití cache).
  - **Vícerozměrné reprezentace:** Flexibilnější, například pro "roztrhaná pole" (ragged arrays), kde řádky mohou mít různé délky.

- **Další schémata:**
  - **Bloková reprezentace:** Matice je rozdělena na bloky a každý blok je uložen souvisle.

- **Zásobník (Stack):**
  - LIFO - Last-In, First-Out.
  - Operace:
    - `PUSH`: Vloží prvek na vrchol zásobníku.
    - `POP`: Odebere prvek z vrcholu zásobníku.
  - Implementace: Používá pole $S[1 \ldots n]$ a atribut S.top (index nejnověji vloženého prvku).
    - S.top $= 0 \implies$ zásobník je prázdný.
    - Při pokusu o `POP` z prázdného zásobníku: podtečení (underflow).
    - Při pokusu o `PUSH` na plný zásobník (S.top $= S$.size): přetečení (overflow).
  - Všechny operace (`STACK-EMPTY`, `PUSH`, `POP`) mají časovou složitost $O(1)$.



STACK-EMPTY(S)

```
1  if S.top == 0
2      return TRUE
3  else return FALSE
```

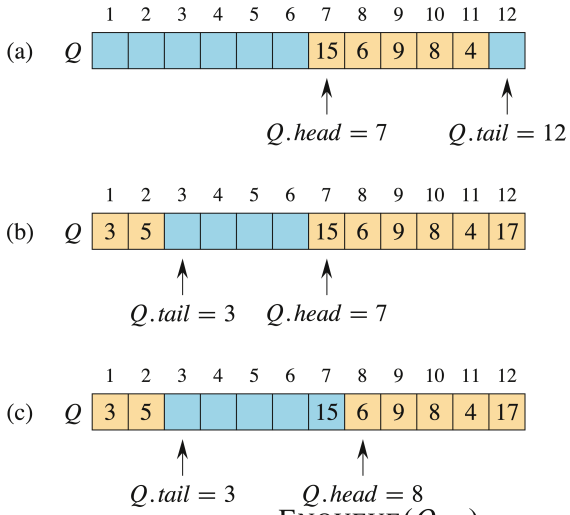
PUSH(S, x)

```
1  if S.top == S.size
2      error "overflow"
3  else S.top = S.top + 1
4      S[S.top] = x
```

POP(S)

```
1  if STACK-EMPTY(S)
2      error "underflow"
3  else S.top = S.top − 1
4      return S[S.top + 1]
```

- FIFO - First-In, First-Out.
- Operace:
  - `ENQUEUE`: Vloží prvek na konec fronty.
  - `DEQUEUE`: Odebere prvek z hlavy fronty.
- Implementace: Používá pole $Q[1 \dots n]$ a atributy Q.head (index hlavy fronty) a Q.tail (index dalšího místa pro nový prvek).
  - Prvky ve frontě se nacházejí v lokacích od Q.head do Q.tail $- 1$, s "omotáváním" (wrap-around) v kruhovém uspořádání (lokace 1 následuje lokaci n).
  - Q.head $= Q$.tail $\implies$ fronta je prázdná.
  - Podtečení/Přetečení: Detekce těchto stavů je důležitá.
- Všechny operace (`ENQUEUE`, `DEQUEUE`) mají časovou složitost $O(1)$.

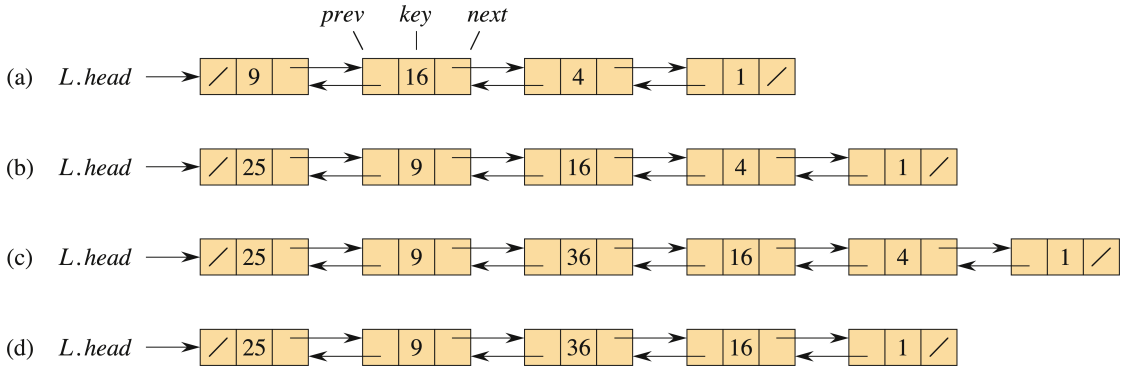



```
ENQUEUE(Q, x)
1  Q[Q.tail] = x
2  if Q.tail == Q.size
3       Q.tail = 1
4  else Q.tail = Q.tail + 1

DEQUEUE(Q)
1  x = Q[Q.head]
2  if Q.head == Q.size
3       Q.head = 1
4  else Q.head = Q.head + 1
5  return x
```

Cvičení: přidejte ověření přetečení a podtečení fronty.

- Datová struktura, kde jsou objekty uspořádány lineárně.
- Na rozdíl od pole, kde je pořadí dáno indexy, je u spojového seznamu určeno **ukazatelem v každém objektu**.
- Poskytují jednoduchou a flexibilní reprezentaci dynamických množin.
- **Dvojitě spojový seznam (Doubly Linked List):**
  - Každý prvek má atribut `key` a dva ukazatelové atributy: `next` (následník) a `prev` (předchůdce).
  - Pokud x.prev $=$ NIL, x je head seznamu.
  - Pokud x.next $=$ NIL, x je tail seznamu.
  - Atribut L.head ukazuje na první prvek seznamu. Pokud L.head $=$ NIL, seznam je prázdný.

- **Varianty seznamů:**
  - **Jednoduché (Singly linked):** Každý prvek má jen `next` ukazatel.
  - **Řazené (Sorted):** Prvky jsou uspořádány podle klíčů.
  - **Kruhové (Circular):** Ukazatel `prev` head ukazuje na tail, a ukazatel `next` tailu ukazuje na head.
- Předpokládáme **neseřazené a dvojitě spojové seznamy**.



- **Vyhledávání (SEARCH):**
  - Procedura LIST-SEARCH(L, k): Prohledává seznam lineárně pro první prvek s klíčem k.
  - Doba běhu: $\Theta(n)$ v nejhorším případě.

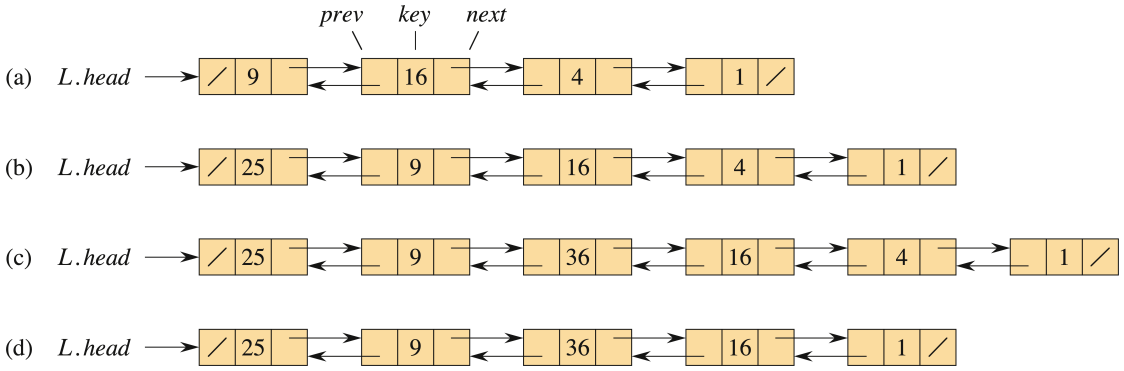
```
LIST-SEARCH(L, k)
1  x = L.head
2  while x ≠ NIL and x.key ≠ k
3      x = x.next
4  return x
```

- **Vkládání (INSERT):**
  - Procedura LIST-PREPEND(L, x): Přidá prvek x na začátek seznamu. Doba běhu $O(1)$.
  - Procedura LIST-INSERT(x, y): Vloží nový prvek x do seznamu *hned za* prvek y. Doba běhu $O(1)$.

```
LIST-PREPEND(L, x)
1  x.next = L.head
2  x.prev = NIL
3  if L.head ≠ NIL
4        L.head.prev = x
5  L.head = x
```

```
LIST-INSERT(x, y)
1  x.next = y.next
2  x.prev = y
3  if y.next ≠ NIL
4        y.next.prev = x
5  y.next = x
```

Operace LIST-PREPEND(L, z) a LIST-INSERT(x, y), kde $z.key = 25$, $x.key = 36$ a $y.key = 9$.

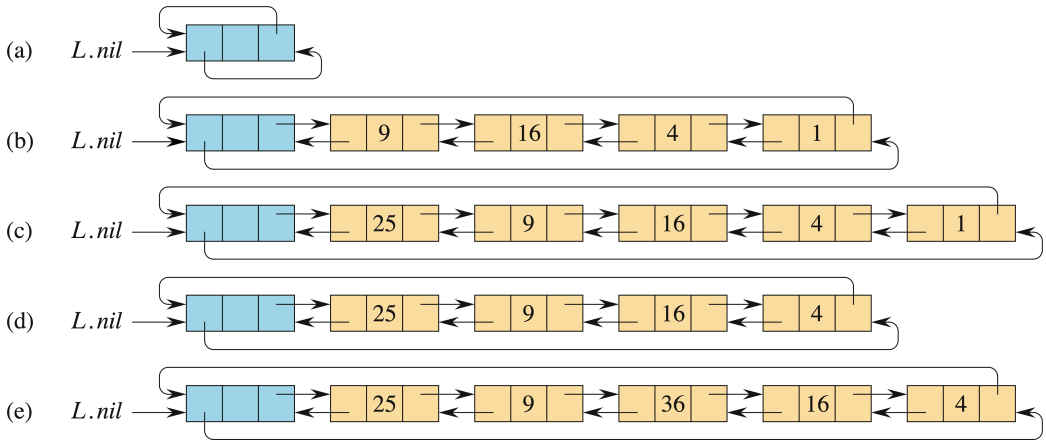


- **Mazání (DELETE):**
  - Procedura LIST-DELETE(L, x): Odebere prvek x ze seznamu L. Vyžaduje ukazatel na x.
  - Doba běhu: $O(1)$.
  - Chcete-li smazat prvek s daným klíčem, nejprve zavolejte LIST-SEARCH (což trvá $\Theta(n)$). Celková doba běhu: $\Theta(n)$.

```
LIST-DELETE(L, x)
1  if x.prev ≠ NIL
2        x.prev.next = x.next
3  else L.head = x.next
4  if x.next ≠ NIL
5        x.next.prev = x.prev
```

- **Porovnání s poli:**
  - Vkládání a mazání na začátku/konci pole trvá $\Theta(n)$ (přesouvání prvků). U dvojitě spojových seznamů $O(1)$.
  - Nalezení k-tého prvku: $O(1)$ v poli, $\Theta(k)$ ve spojovém seznamu.

- **Sentinel (strážce):** Fiktivní objekt, který zjednodušuje krajní případy (např. prázdný seznam, první/poslední prvek).
- **Kruhový, dvojitě spojový seznam se sentinelem:**
  - Sentinel L.nil nahrazuje `NIL` a má všechny atributy ostatních objektů.
  - L.nil leží mezi `head` a `tail`.
  - L.nil.next ukazuje na `head` seznamu, L.nil.prev ukazuje na `tail`.
  - Atribut L.head je eliminován.
  - Prázdný seznam se skládá pouze ze sentinelu, kde L.nil.next i L.nil.prev ukazují na L.nil.
- **Zjednodušené procedury:**
  - LIST-DELETE'(x): Nyní jen 2 řádky, protože se nemusí řešit okrajové případy.
  - LIST-INSERT'(x, y): Vkládá x za y. Lze použít pro vkládání na `head` ($y = L$.nil) nebo `tail` ($y = L$.nil.prev).
- Sentinely mohou zjednodušit kód a mírně zrychlit běh (konstantní faktor), ale typicky **nezlepšují asymptotickou časovou složitost**.
- **Pozor:** Sentinel by se nikdy neměl mazat, pokud nemažete celý seznam.



LIST-DELETE'(x)

```
1  x.prev.next = x.next
2  x.next.prev = x.prev
```

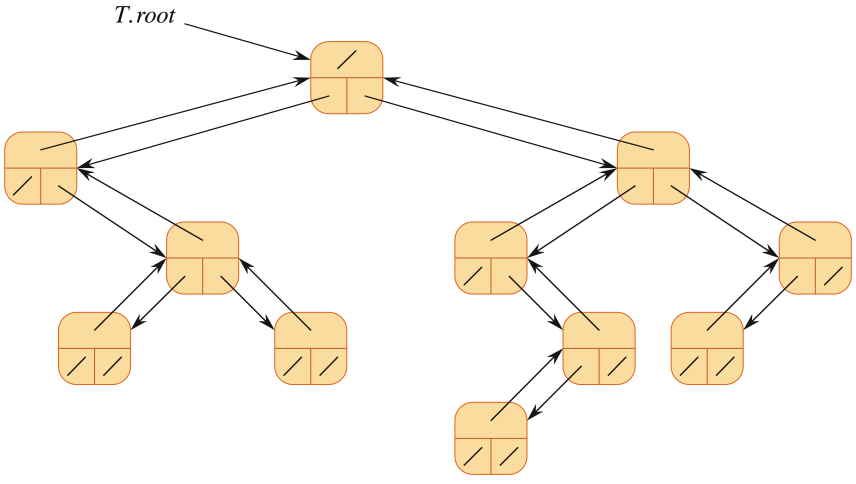
LIST-INSERT'(x, y)

```
1  x.next = y.next
2  x.prev = y
3  y.next.prev = x
4  y.next = x
```

LIST-SEARCH$'(L,k)$

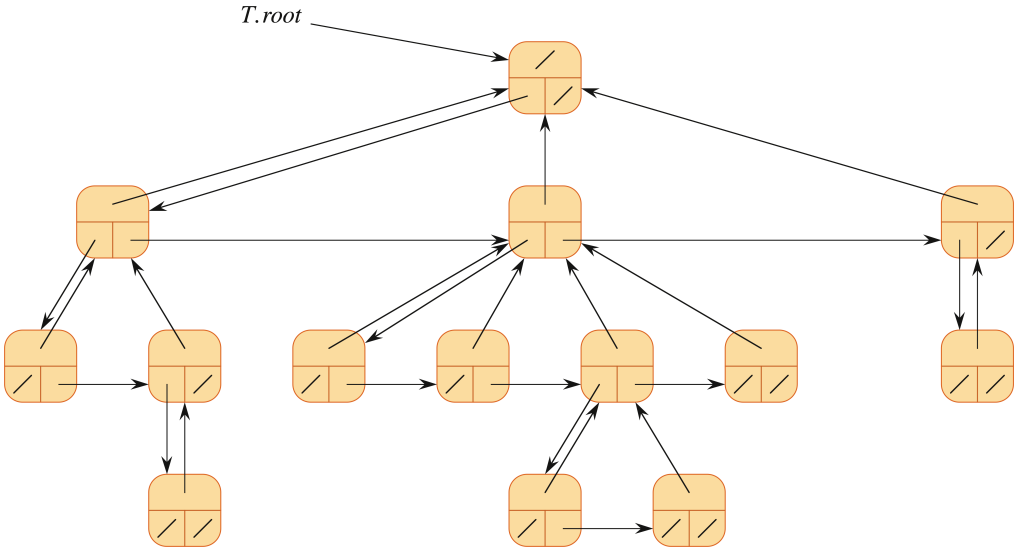
```
1  L.nil.key = k          // store the key in the sentinel to guarantee it is in list
2  x = L.nil.next         // start at the head of the list
3  while x.key ≠ k
4      x = x.next
5  if x == L.nil          // found k in the sentinel
6      return NIL         // k was not really in the list
7  else return x          // found k in element x
```

- Spojové seznamy jsou vhodné pro lineární vztahy, ale ne všechny vztahy jsou lineární.
- Reprezentace kořenových stromů pomocí spojových datových struktur.
- Každý uzel stromu je reprezentován objektem, který obsahuje atribut `key` a ukazatele na jiné uzly.
- Používají se atributy `p`, `left` a `right` pro ukazatele na rodiče, levého potomka a pravého potomka každého uzlu.
- Pokud x.p = NIL, pak x je kořen.
- Pokud uzel x nemá levého potomka, pak x.left = NIL (podobně pro pravého potomka).
- Kořen celého stromu T je ukazován atributem T.root. Pokud T.root = NIL, strom je prázdný.



- Rozšíření schématu pro binární strom na stromy, kde je počet potomků uzlu nejvýše nějaká konstanta k: nahraďte atributy `left` a `right` atributy $\texttt{child}_1$, $\texttt{child}_2$, ..., $\texttt{child}_k$.

- Toto schéma nefunguje, když je počet potomků uzlu **neomezený**, nebo pokud k je velká konstanta, ale většina uzlů má malý počet potomků (plýtvání pamětí).
- **Chytré schéma: Reprezentace levý potomek, pravý sourozenec (left-child, right-sibling representation).**
  - Používá pouze $O(n)$ prostoru pro libovolný kořenový strom s n uzly.
  - Každý uzel x má pouze dva ukazatele (kromě ukazatele na rodiče p):
    1. x.left-child: ukazuje na **nejlevějšího potomka** uzlu x.
    2. x.right-sibling: ukazuje na **sourozence** uzlu x ihned po jeho pravici.
  - Pokud uzel x nemá potomky, pak x.left-child $=$ NIL.
  - Pokud uzel x je nejpravějším potomkem svého rodiče, pak x.right-sibling $=$ NIL.



- Halda (Heap) reprezentovaná jedním polem.
- Stromy, které se procházejí jen směrem ke kořeni (pouze ukazatele na rodiče).
- Nejlepší schéma závisí na konkrétní aplikaci.
- **Pole:**
  - Kompaktní, rychlý přístup pomocí indexu.
  - Pevná velikost, neefektivní pro dynamické změny velikosti.

  - Dobré pro zásobníky, fronty (kruhové), haldy.

- **Spojové seznamy:**
  - Dynamická velikost, efektivní vkládání/mazání.
  - Pomalejší přístup k prvkům (nutno procházet).
  - Dobré pro implementaci zásobníků/front, řízení paměti, grafy.

- **Stromy (uzly s ukazateli):**
  - Reprezentace hierarchických dat.
  - Dynamická struktura.
  - Složitost operací závisí na výšce stromu (logaritmická pro vyvážené stromy).

- **Zásobníky (Stacks):** LIFO (Last-In, First-Out). Operace $O(1)$. Implementace polem nebo spojovým seznamem.

- **Fronty (Queues):** FIFO (First-In, First-Out). Operace $O(1)$. Implementace kruhovým polem nebo spojovým seznamem.

- **Spojové seznamy (Linked Lists):** Dynamické, ukazatele. Jednosměrné, obousměrné, kruhové. Vkládání/mazání $O(1)$ (s ukazatelem), hledání $O(n)$.

- **Kořenové stromy:** Reprezentace hierarchie. 'parent', 'child', 'sibling' pro obecné stromy. 'parent', 'left', 'right' pro binární stromy.

- Jsou to základní stavební kameny pro komplexnější algoritmy a datové struktury.

- Pochopení jejich principů a implementace je zásadní pro návrh efektivních algoritmů.

- Volba správné datové struktury může dramaticky ovlivnit výkon algoritmu.

- Tvoří základ pro pokročilé struktury jako hash tabulky, binární vyhledávací stromy, haldy a grafové algoritmy.

- **Problém řazení:**
  - **Vstup:** Posloupnost n čísel $\langle a_1, a_2, \ldots, a_n \rangle$.
  - **Výstup:** Permutace vstupní posloupnosti $\langle a'_1, a'_2, \ldots, a'_n \rangle$ taková, že $a'_1 \leq a'_2 \leq \cdots \leq a'_n$.

- **Struktura dat v praxi:**
  - Řazené hodnoty (tzv. **klíče**) jsou často součástí větších datových celků nazývaných **záznamy**.

  - Záznamy obsahují i **satelitní data**, která se s klíčem přenášejí.
  - Při řazení klíčů musí algoritmus přeskupit i satelitní data.
  - Pro minimalizaci přesunu dat se často operace provádí na **poli ukazatelů** na záznamy.
  - Pro účely algoritmické analýzy se obvykle předpokládá, že vstupem jsou pouze čísla.

- **Vlastní potřeba aplikací:**
  - Například banky potřebují řadit šeky podle čísla pro přípravu výpisů zákazníkům.
- **Klíčový podprogram:**
  - Algoritmy často používají řazení jako základní podprogram (např. pro renderování grafických objektů podle "nadřazeného" vztahu).
- **Bohatství technik:**
  - Existuje široká škála řadicích algoritmů, které využívají rozmanité návrhové techniky.
- **Teoretické limity:**
  - Lze dokázat **netriviální dolní mez** pro řazení porovnáváním ($\Omega(n \log n)$).
  - Některé algoritmy jsou **asymptoticky optimální**, protože jejich horní meze odpovídají dolní.
  - Dolní mez pro řazení lze použít k důkazu dolních mezí pro jiné problémy.
- **Inženýrské aspekty:**
  - Implementace řadicích algoritmů zahrnuje mnoho praktických faktorů (znalost klíčů a dat, paměťová hierarchie, softwarové prostředí).
- **Srovnávací řazení** určují seřazené pořadí vstupního pole porovnáváním prvků.
- **Insertion Sort (Řazení vkládáním):**
  - Časová složitost: $\Theta(n^2)$ (nejhorší případ).
  - Rychlé pro malé vstupy; řadí **na místě** (in-place, tj. s konstantním dodatečným prostorem).
- **Merge Sort (Řazení slučováním):**
  - Časová složitost: $\Theta(n \log n)$ (vždy).

  - Nevýhoda: **Nerařadí na místě** (vyžaduje dodatečný prostor).

- **Heapsort (Řazení haldou):**
  - Časová složitost: $O(n \log n)$ (nejhorší případ).
  - Řadí **na místě**; využívá důležitou datovou strukturu - **haldu** (heap).

- **Quicksort (Rychlé řazení):**
  - Časová složitost: $\Theta(n^2)$ (nejhorší případ), ale $\Theta(n \log n)$ **očekávaný čas**.
  - Řadí **na místě**; v praxi často překonává Heapsort.

- **Dolní mez pro srovnávací řazení:**
  - Pomocí modelu **rozhodovacího stromu** lze dokázat $\Omega(n \log n)$ pro nejhorší případ libovolného srovnávacího řazení.
  - Heapsort a Merge Sort jsou proto **asymptoticky optimální** srovnávací řadicí algoritmy.

- **Nesrovnávací řazení** mohou překonat dolní mez $\Omega(n \log n)$ tím, že získávají informace o seřazeném pořadí jinými prostředky než porovnáváním prvků.

- **Counting Sort (Řazení počítáním):**
  - Předpoklad: Vstupní čísla patří do množiny $\{0, 1, \ldots, k\}$.
  - Využívá indexování pole k určení relativního pořadí.
  - Časová složitost: $\Theta(k + n)$. Běží v lineárním čase, pokud $k = O(n)$.

- **Radix Sort (Radixové řazení):**
  - Rozšiřuje rozsah Counting Sortu.
  - Pokud existuje n celých čísel k řazení, každé s d číslicemi, a každá číslice může nabývat až k možných hodnot.
  - Časová složitost: $\Theta(d(n + k))$. Běží v lineárním čase, pokud $d =$ konst. a $k = O(n)$.

- **Bucket Sort (Řazení do kbelíků):**
  - Předpoklad: Vyžaduje znalost pravděpodobnostního rozdělení čísel ve vstupním poli (např. rovnoměrně rozdělená reálná čísla v intervalu $[0, 1)$).
  - Časová složitost: $O(n)$ **v průměrném případě**, $\Theta(n^2)$ v nejhorším případě.

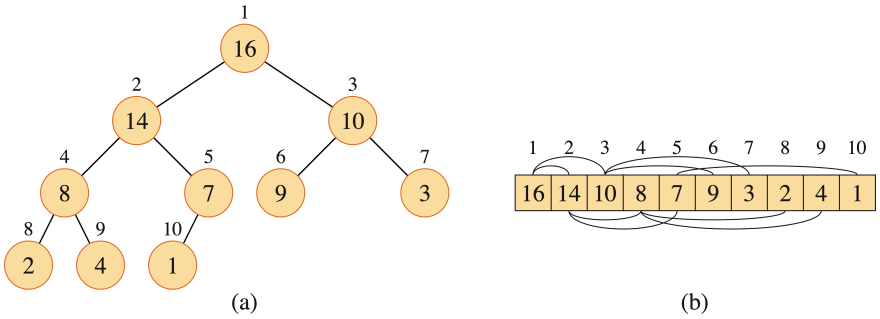
Algoritmus	Nejhorší případ	Průměrný/Očekávaný čas
Insertion Sort	$\Theta(n^2)$	$\Theta(n^2)$
Merge Sort	$\Theta(n \log n)$	$\Theta(n \log n)$
Heapsort	$O(n \log n)$	-
Quicksort	$\Theta(n^2)$	$\Theta(n \log n)$ (očekávaný)
Counting Sort	$\Theta(k+n)$	$\Theta(k+n)$
Radix Sort	$\Theta(d(n+k))$	$\Theta(d(n+k))$
Bucket Sort	$\Theta(n^2)$	$O(n)$ (průměrný)

- n: počet prvků k řazení.
- Pro Counting Sort: k je horní mezí pro hodnoty prvků ($[0, k]$).
- Pro Radix Sort: d je počet číslic, k je počet možných hodnot jedné číslice.
- Pro Bucket Sort: klíče jsou reálná čísla rovnoměrně rozdělená v $[0, 1)$.
- **Definice:** i-tá pořádková statistika množiny n čísel je i-té nejmenší číslo v této množině.
- **Naivní přístup:** Můžete najít i-tou pořádkovou statistiku seřazením vstupu a indexováním i-tého prvku ve výstupu.
  - Časová složitost: $\Omega(n \log n)$ (kvůli dolní mezi pro řazení).
- **Efektivnější algoritmy pro nalezení i-té pořádkové statistiky:**
  - Algoritmy, které umí najít i-tý nejmenší prvek v $O(n)$ čase i pro libovolná reálná čísla.
  - **Randomizovaný algoritmus:**
    - Časová složitost: $O(n)$ **očekávaný čas**, ale $\Theta(n^2)$ v nejhorším případě.
  - **Složitější deterministický algoritmus:**
    - Časová složitost: $O(n)$ **v nejhorším případě**.
- **Důležité pozadí:**
  - Analýzy algoritmů jako Quicksort, Bucket Sort a algoritmů pro pořádkové statistiky často využívají **pravděpodobnostní analýzu**.

8 Heapsort

- Predstavuje další algoritmus pro řazení: **Heapsort** (řazení haldou).
- **Kombinuje nejlepší vlastnosti** dříve probraných algoritmů:
  - Stejně jako Merge Sort: Časová složitost $O(n \log n)$.

  - Stejně jako Insertion Sort: Řadí **na místě** (in-place), tzn. vyžaduje pouze konstantní počet dodatečných prvků pole mimo vstupní pole.
- Zavádí novou techniku návrhu algoritmů: Použití **datové struktury** (v tomto případě **halda**) pro správu dat.
- Halda není jen pro Heapsort, ale je také základem pro efektivní implementaci **prioritní fronty**.
- **Halda:** Datová struktura pole, které lze vizualizovat jako **téměř úplný binární strom**



- Každý uzel stromu odpovídá prvku v poli.
- Strom je **zcela zaplněn na všech úrovních** až na nejnižší, která je zaplněna zleva doprava.
- **Reprezentace polem:** Pole $A[1 \ldots n]$ reprezentuje haldu. Má atribut $A.heap\text{-}size$, který udává počet prvků haldy aktuálně uložených v poli A.
  - $0 \leq A.heap\text{-}size \leq n$.
  - Pokud $A.heap\text{-}size = 0$, halda je prázdná.
- **Kořen stromu** je vždy prvek $A[1]$.
- Pro daný index uzlu i lze jednoduše vypočítat indexy jeho rodiče, levého potomka a pravého potomka:

```
PARENT(i)
1  return ⌊i/2⌋

LEFT(i)
1  return 2i

RIGHT(i)
1  return 2i + 1
```

- **Efektivita:** Na většině počítačů lze tyto operace implementovat pomocí bitových posunů:
  - $2i$: posun i doleva o jeden bit.
  - $\lfloor i/2 \rfloor$: posun i doprava o jeden bit.

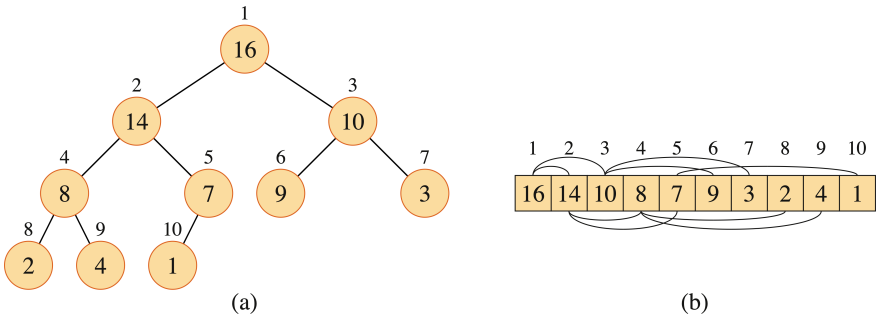
  Často se implementují jako makra nebo inline procedury pro maximální rychlost.
- Existují dva typy binárních hald: **max-haldy** a **min-haldy**.
- Oba typy splňují **vlastnost haldy**, jejíž specifika závisí na typu:
- **Max-halda:** Pro každý uzel i kromě kořene platí **vlastnost**:

$$A[\textsc{Parent}(i)] \geq A[i]$$

  To znamená, že hodnota uzlu je nejvýše rovna hodnotě jeho rodiče. Největší prvek v max-haldě je vždy uložen v kořeni. Podstrom s kořenem v daném uzlu obsahuje hodnoty ne větší než hodnota samotného uzlu.
- **Min-halda:** Pro každý uzel i kromě kořene platí **vlastnost**:

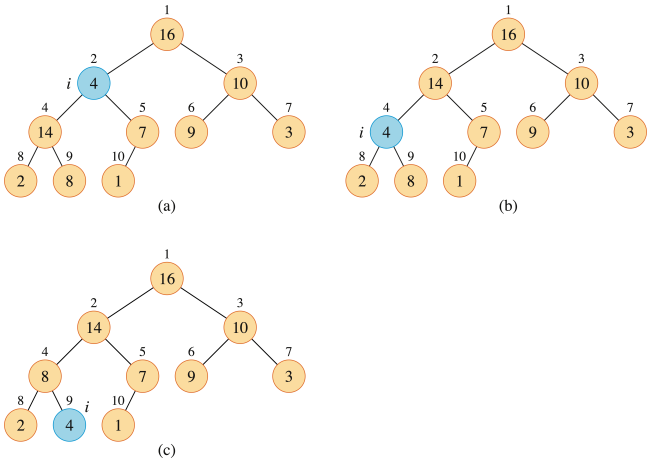
$$A[\textsc{Parent}(i)] \leq A[i]$$

  Nejmenší prvek v min-haldě je vždy v kořeni.
- **Heapsort používá max-haldy.** Min-haldy se běžně používají pro implementaci prioritních front (později).
- **Výška uzlu v haldě:** Počet hran na nejdelší cestě dolů od uzlu k listu.
- **Výška haldy:** Výška jejího kořene.
- Jelikož halda s n prvky je založena na úplném binárním stromu, její výška je $\Theta(\log n)$.
- **Časová složitost základních operací:** Základní operace s haldami běží v čase nejvýše úměrném výšce stromu, a tedy trvají $O(\log n)$.



(a) (b)

- Následující procedury jsou základem pro pochopení Heapsortu a prioritních front:
- **MAX-HEAPIFY(A,i):**
  - Klíčová procedura pro **udržování vlastnosti max-haldy**.
  - Běží v čase $O(\log n)$.
- **BUILD-MAX-HEAP(A,n):**
  - Vytvoří max-haldu z neseřazeného vstupního pole.
  - Běží v **lineárním čase**, $O(n)$.
- **HEAPSORT(A,n):**
  - Seřadí pole na místě.
  - Běží v čase $O(n \log n)$.
- **Procedury pro prioritní frontu:** MAX-HEAP-INSERT, MAX-HEAP-EXTRACT-MAX, MAX-HEAP-INCREASE-KEY, MAX-HEAP-MAXIMUM.
  - Běží v čase $O(\log n)$ plus čas na mapování objektů do haldy a zpět.
- Procedura **MAX-HEAPIFY**(A, i) udržuje vlastnost max-haldy.
- **Vstupy:** Pole A s atributem $A.heap\text{-}size$ a index i do pole.
- **Předpoklady při volání:**
  - Binární stromy s kořeny $LEFT(i)$ a $RIGHT(i)$ jsou **již max-haldy**.
  - Avšak $A[i]$ může být menší než jeho potomci, čímž porušuje vlastnost max-haldy.
- **Funkce:** MAX-HEAPIFY nechá hodnotu v $A[i]$ "propadnout" dolů v max-haldě tak, aby podstrom s kořenem v indexu i dodržoval vlastnost max-haldy.



MAX-HEAPIFY(A, i)

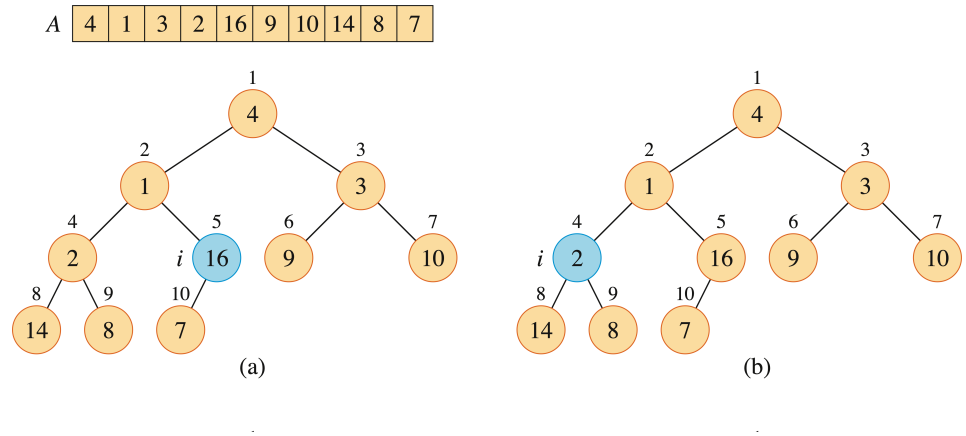
```
1  l = LEFT(i)
2  r = RIGHT(i)
3  if l ≤ A.heap-size and A[l] > A[i]
4       largest = l
5  else largest = i
6  if r ≤ A.heap-size and A[r] > A[largest]
7       largest = r
8  if largest ≠ i
9       exchange A[i] with A[largest]
10      MAX-HEAPIFY(A, largest)
```

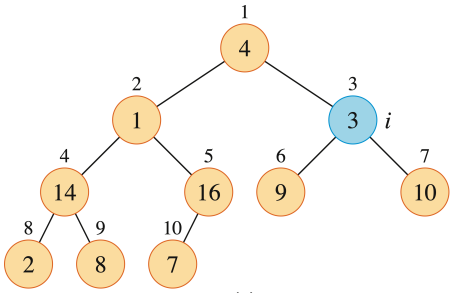
- **Logika operace:**
  1. Index největší z prvků $A[i]$, $A[LEFT(i)]$ a $A[RIGHT(i)]$ uloží do proměnné `largest`.
  2. Pokud $A[i]$ je největší, podstrom s kořenem i je již max-halda a nic dalšího není potřeba.
  3. Jinak, jeden z potomků obsahuje největší prvek. Dojde k **výměně obsahu** pozic i a `largest`. To zajistí, že uzel i a jeho potomci splňují vlastnost max-haldy.
  4. Uzel indexovaný proměnnou `largest` (který byl právě snížen) však mohl porušit vlastnost max-haldy. Proto se MAX-HEAPIFY **rekurzivně volá** na tento podstrom.
- Nechť $T(n)$ je časová složitost v nejhorším případě, kterou procedura zabere na stromu velikosti nejvýše n.
- Pro strom s kořenem v daném uzlu i je doba běhu $\Theta(1)$ (na úpravu vztahů mezi $A[i]$, $A[LEFT(i)]$ a $A[RIGHT(i)]$) plus čas na spuštění MAX-HEAPIFY na podstromu s kořenem v jednom z potomků uzlu i.

- Každý podstrom potomků má velikost nejvýše $2n/3$.
- Proto můžeme časovou složitost MAX-HEAPIFY popsat rekurencí:

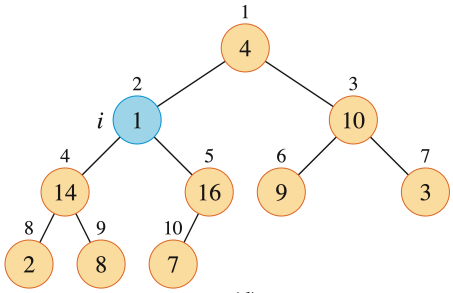
$$T(n) \leq T(2n/3) + \Theta(1)$$

- Řešení této rekurence (podle případu 2 Master theoremu) je $T(n) = O(\log n)$.
- Alternativně můžeme charakterizovat časovou složitost MAX-HEAPIFY na uzlu výšky h jako $O(h)$.
- Procedura **BUILD-MAX-HEAP**(A, n) převádí pole $A[1 \ldots n]$ na max--haldu.
- Volá MAX-HEAPIFY **zdola nahoru**.
- **Listy:** Prvky v podpoli $A[\lfloor n/2 \rfloor + 1 \ldots n]$ jsou všechny listy stromu, a proto je každý z nich na začátku 1-prvková halda.
- BUILD-MAX-HEAP prochází zbývající uzly stromu (ne-listy) a spouští na každém z nich MAX-HEAPIFY.



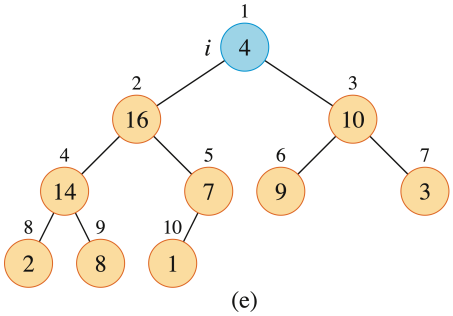


(c)

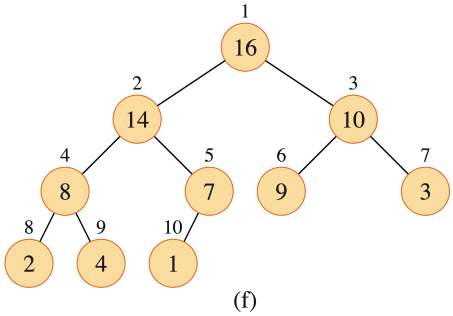


(d)

A	4	1	3	2	16	9	10	14	8	7



(e)



(f)

BUILD-MAX-HEAP(A, n)

1 $A.heap\text{-}size = n$
2 **for** $i = \lfloor n/2 \rfloor$ **downto** 1
3 MAX-HEAPIFY(A, i)

- **Důkaz správnosti (Loop Invariant):**
  - **Invariant:** Na začátku každé iterace cyklu je každý uzel $i+1, i+2, \ldots, n$ kořenem max-haldy.
  - **Inicializace:** Před první iterací ($i = \lfloor n/2 \rfloor$) jsou uzly $\lfloor n/2 \rfloor + 1, \ldots, n$ listy a jsou tedy triviální max-haldy.
  - **Indukce:** Potomci uzlu i mají vyšší indexy než i. Podle invariantu jsou tedy kořeny max-hald. To je přesně podmínka pro volání MAX-HEAPIFY(A, i), aby se uzel i stal kořenem max-haldy. MAX--HEAPIFY navíc zachovává vlastnost pro uzly $i+1, \ldots, n$. Snížení i obnoví invariant pro další iteraci.
  - **Ukončení:** Smyčka provede přesně $\lfloor n/2 \rfloor$ iterací. Po ukončení je $i = 0$. Podle invariantu je každý uzel $1, 2, \ldots, n$ kořenem max-haldy. Zvláště uzel 1 (kořen) je.

- **Jednoduchý horní odhad:** Každé volání MAX-HEAPIFY stojí $O(\log n)$ a BUILD-MAX-HEAP provede $O(n)$ takových volání. Celkový čas je $O(n \log n)$. Tento odhad je správný, ale není nejpřesnější.

- **Přesnější asymptotický odhad:**
  - Čas pro MAX-HEAPIFY se mění s výškou uzlu ve stromu. Většina uzlů má malou výšku.
  - Halda s n prvky má výšku $\lfloor \log n \rfloor$.
  - Na libovolné výšce h je nejvýše $\lceil n/2^{h+1} \rceil$ uzlů. Navíc, $\lceil n/2^{h+1} \rceil \geq 1/2$ pro $0 \leq h \leq \lfloor \log n \rfloor$.
  - Pro $x \geq 1/2$ platí $\lceil x \rceil \leq 2x$, proto $\lceil n/2^{h+1} \rceil \leq n/2^h$ pro $0 \leq h \leq \lfloor \log n \rfloor$.
  - Čas, který MAX-HEAPIFY potřebuje pro uzel výšky h je $O(h)$.

- Celkové náklady na BUILD-MAX-HEAP jsou shora omezeny:

$$\sum_{h=0}^{\lfloor \log n \rfloor} \lceil n/2^{h+1} \rceil \cdot ch$$

kde c je konstanta z $O(h)$. Odvození (použitím geom. řady $\sum_{k=0}^{\infty} kx^k = \frac{x}{(1-x)^2}, |x| < 1$):

$$\sum_{h=0}^{\lfloor \log n \rfloor} \left\lceil \frac{n}{2^{h+1}} \right\rceil ch \leq \sum_{h=0}^{\lfloor \log n \rfloor} \frac{n}{2^h} ch = cn \sum_{h=0}^{\lfloor \log n \rfloor} \frac{h}{2^h} \leq cn \sum_{h=0}^{\infty} \frac{h}{2^h} = cn \frac{1/2}{(1-1/2)^2} = 2cn = O(n)$$

- **Závěr:** Max-haldu z neseřazeného pole můžeme vytvořit v **lineárním čase**, $O(n)$.

- Obdobně lze vytvořit min-haldu (pomocí MIN-HEAPIFY) v lineárním čase.

- Algoritmus HEAPSORT začíná voláním procedury **BUILD-MAX-HEAP** pro vytvoření max-haldy ze vstupního pole $A[1 \ldots n]$.

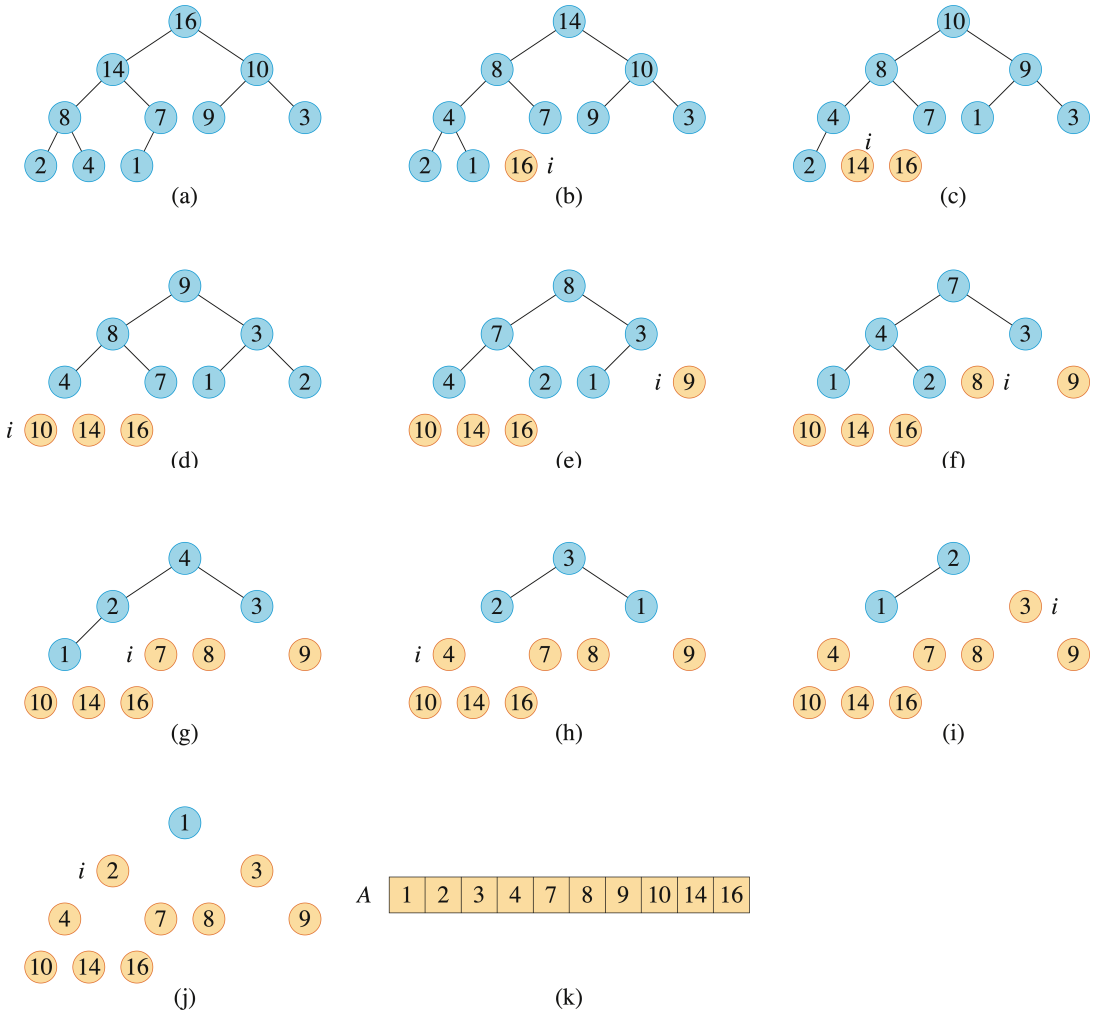
- **Princip řazení:** Protože maximální prvek pole je uložen v kořeni $A[1]$, Heapsort jej může umístit na správnou konečnou pozici výměnou s $A[n]$.

- **Iterativní proces:**
  - Jakmile je prvek $A[n]$ na svém místě, uzel n se "odstraní" z haldy (jednoduchým dekrementováním $A.heap\text{-}size$).
  - Potomci kořene zůstávají max-haldami, ale nový kořenový prvek může porušovat vlastnost max-haldy.
  - Pro obnovení vlastnosti max-haldy se jednoduše zavolá **MAX-HEAPIFY**$(A, 1)$, což vytvoří max-haldu v $A[1 \ldots n-1]$.

– HEAPSORT pak tento proces opakuje pro max-haldu velikosti $n-1$ až do haldy velikosti 2.

HEAPSORT(A, n)

```
1  BUILD-MAX-HEAP(A, n)
2  for i = n downto 2
3      exchange A[1] with A[i]
4      A.heap-size = A.heap-size − 1
5      MAX-HEAPIFY(A, 1)
```



- Volání **BUILD-MAX-HEAP** trvá $O(n)$.
- Každé z $n-1$ volání **MAX-HEAPIFY** trvá $O(\log n)$.

- **Celková časová složitost:** $O(n \log n)$.
- **Optimálnost:** Později uvidíme, že jakýkoli algoritmus řazení založený na porovnávání vyžaduje $\Omega(n \log n)$ porovnání, a tedy $\Omega(n \log n)$ času. Proto je Heapsort **asymptoticky optimální** mezi algoritmy založenými na porovnávání.
- **Praktické použití:** Přestože je Heapsort asymptoticky optimální, dobrá implementace Quicksortu (později) jej obvykle v praxi překonává.

Vlastnosti Heapsortu

- **V nejhorším případě:** $\Theta(n \log n)$.
- **Prostorová složitost:** $\mathcal{O}(1)$ (in-place řazení, protože používá pouze konstantní dodatečný prostor).
- Není stabilní (pořadí prvků se stejnou hodnotou se může změnit).
- **Halda (Heap):** Téměř úplný binární strom s vlastností haldy (rodič $\geq$ potomci pro max-haldu).
- **Reprezentace polem:** Efektivní díky vztahům rodič/potomek $i \leftrightarrow \lfloor i/2 \rfloor, 2i, 2i+1$.
- **MAX-HEAPIFY:**
  - Obnovuje vlastnost haldy dolů po stromě.
  - Časová složitost: $\Theta(\log n)$.
- **BUILD-MAX-HEAP:**
  - Přemění libovolné pole na haldu.
  - Klíčové: Volá ‘MAX-HEAPIFY‘ od spodních úrovní.
  - Časová složitost: $\Theta(n)$ (lineární!).
- **HEAPSORT:**
  - Fáze 1: `BUILD-MAX-HEAP` ($\Theta(n)$).
  - Fáze 2: Opakované vyjímání max prvku a ‘MAX-HEAPIFY‘ ($\Theta(n \log n)$).
  - Celková časová složitost: $\Theta(n \log n)$ v nejhorším případě.
  - Prostorová složitost: $\mathcal{O}(1)$ (in-place).
- **Prioritní fronta:** Datová struktura pro udržování množiny prvků, S, z nichž každý má přidruženou hodnotu zvanou **klíč**.
- Stejně jako haldy, prioritní fronty existují ve dvou formách: max-prioritní fronty a min-prioritní fronty. Zaměřujeme se na **max-prioritní fronty**, které jsou založeny na max-haldách.

- **Max-prioritní fronta podporuje následující operace:**
  - **INSERT**(S, x, k): Vloží prvek x s klíčem k do množiny S, což je ekvivalentní operaci $S = S \cup \{x\}$.
  - **MAXIMUM**(S): Vrátí prvek z S s největším klíčem.
  - **EXTRACT-MAX**(S): Odstraní a vrátí prvek z S s největším klíčem.
  - **INCREASE-KEY**(S, x, k): Zvýší hodnotu klíče prvku x na novou hodnotu k, která je alespoň tak velká jako aktuální hodnota klíče x.
- **Aplikace max-prioritních front:**
  - **Plánování úloh:** Můžete je použít k plánování úloh na počítači sdíleném více uživateli. Fronta priorit sleduje úlohy, které mají být provedeny, a jejich relativní priority. Když je úloha dokončena nebo přerušena, plánovač vybere úlohu s nejvyšší prioritou mezi čekajícími voláním EXTRACT-MAX. Nová úloha může být přidána do fronty kdykoli voláním INSERT.
- **Min-prioritní fronty:** Podporují operace INSERT, MINIMUM, EXTRACT-MIN a DECREASE-KEY.
  - **Simulace řízená událostmi:** Položky ve frontě jsou události, které mají být simulovány, každá s přidruženým časem výskytu, který slouží jako její klíč. Události musí být simulovány v pořadí jejich času výskytu, protože simulace jedné události může v budoucnu způsobit simulaci dalších událostí. Simulační program volá EXTRACT-MIN v každém kroku k výběru další události k simulaci.
- Předpokládáme, že $A[i]$ je ukazatel na objekt v prioritní frontě a $A[i].key$ je jeho klíč.

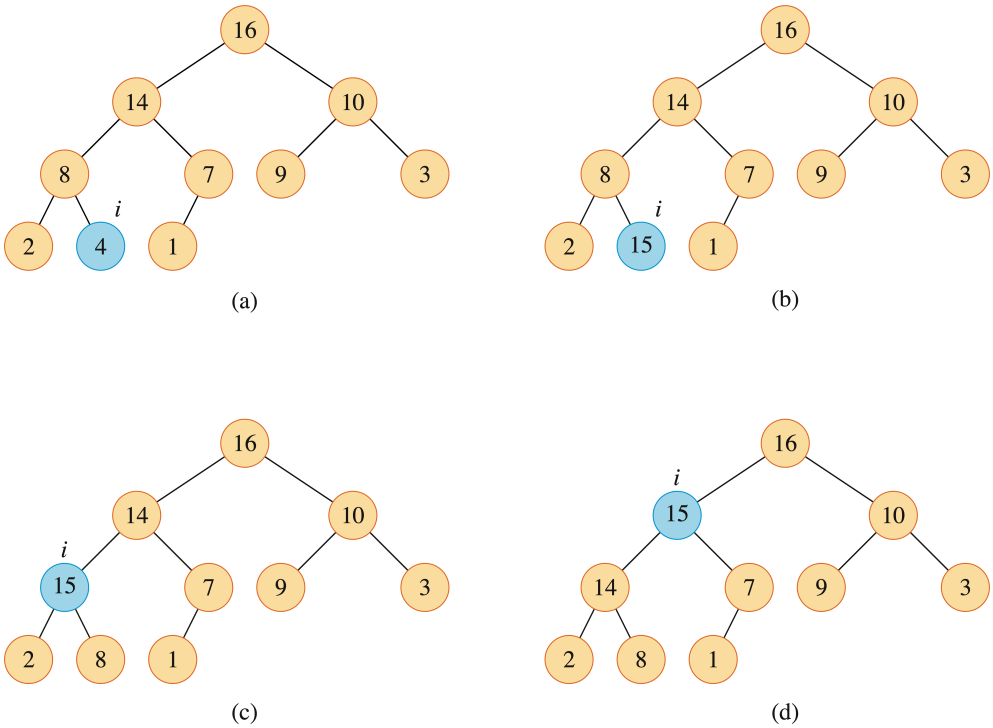
```
MAX-HEAP-MAXIMUM(A)
1  if A.heap-size < 1
2       error "heap underflow"
3  return A[1]

MAX-HEAP-EXTRACT-MAX(A)
1  max = MAX-HEAP-MAXIMUM(A)
2  A[1] = A[A.heap-size]
3  A.heap-size = A.heap-size − 1
4  MAX-HEAPIFY(A, 1)
5  return max
```

- **MAX-HEAP-MAXIMUM(A):** Časová složitost: $\Theta(1)$.
- **MAX-HEAP-EXTRACT-MAX(A):** Podobná části cyklu HEAPSORTu (řádky 3-5). Předpokládáme, že MAX-HEAPIFY porovnává objekty na základě jejich klíčových atributů a aktualizuje mapování. Časová složitost: $O(\log n)$ (konstantní práce + $O(\log n)$ pro MAX-HEAPIFY).

MAX-HEAP-INCREASE-KEY(A, x, k)

```
1  if k < x.key
2      error "new key is smaller than current key"
3  x.key = k
4  find the index i in array A where object x occurs
5  while i > 1 and A[PARENT(i)].key < A[i].key
6      exchange A[i] with A[PARENT(i)], updating the information
           priority queue objects to array indices
7      i = PARENT(i)
```

- **Princip:** Zvyšuje klíč prvku x na hodnotu k. Poté "posune" prvek nahoru v haldě, dokud není splněna vlastnost max-haldy (podobně jako vkládání v Insertion Sort).
- Pozn. potřebujeme udržovat informaci (ukazatel) z prvku x na jeho pozici v haldě.
- Časová složitost: $O(\log n)$ (délka cesty ke kořeni).



MAX-HEAP-INSERT(A, x, n)

```
1  if A.heap-size == n
2        error "heap overflow"
3  A.heap-size = A.heap-size + 1
4  k = x.key
5  x.key = −∞
6  A[A.heap-size] = x
7  map x to index heap-size in the array
8  MAX-HEAP-INCREASE-KEY(A, x, k)
```

- **Princip:** Rozšíří haldu přidáním nového listu s klíčem $-\infty$. Poté zavolá MAX-HEAP-INCREASE-KEY pro nastavení správné hodnoty klíče a udržení vlastnosti haldy.
- Časová složitost: $O(\log n)$ (plus režie pro mapování).

9 Quicksort

- Algoritmus **Quicksort** má časovou složitost v nejhorším případě $\Theta(n^2)$ pro pole n čísel.
- **Praktická volba pro řazení:** Navzdory pomalé časové složitosti v nejhorším případě je Quicksort často nejlepší volbou v praxi.
  - Je **mimořádně efektivní v průměru**: Očekávaná časová složitost je $\Theta(n \log n)$.
  - Konstantní faktory skryté v notaci $\Theta(n \log n)$ jsou malé.
- **Výhody oproti Merge Sortu:**
  - Řadí **na místě** (in-place) – vyžaduje pouze konstantní množství paměti mimo řazené pole.
  - Dobře funguje i v prostředích s **virtuální pamětí**.
- Naše studium Quicksortu je rozděleno do čtyř sekcí:
- **Popis Quicksortu**
  - Algoritmus a důležitá podprogramová procedura PARTITION pro rozdělení pole.
- **Výkonnost Quicksortu**
  - Intuitivní diskuse o výkonu v nejhorším, nejlepším a vyváženém případě.
- **Randomizovaná verze Quicksortu**

  - Tento randomizovaný algoritmus má dobrou očekávanou dobu běhu a žádný konkrétní vstup nevyvolává jeho chování v nejhorším případě.

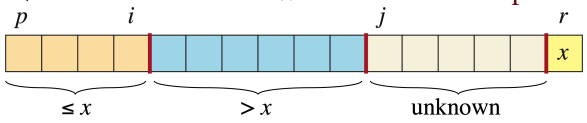
- **Analýza Randomizovaného Quicksortu**
  - Ukazuje časovou složitost $\Theta(n^2)$ v nejhorším případě a očekávaný čas $O(n \log n)$.

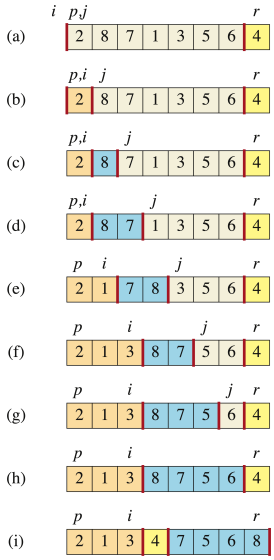
- Quicksort, podobně jako Merge Sort, používá metodu **rozděl a panuj** (divide-and-conquer).
- Třífázový proces pro řazení podpole $A[p \ldots r]$:
- **Rozděl (Divide):**
  - Rozdělení (přeskupení) pole $A[p \ldots r]$ na dvě (možná prázdná) podpole:
    * $A[p \ldots q-1]$ (levá strana)
    * $A[q+1 \ldots r]$ (pravá strana)
  - Každý prvek v levé straně je **menší nebo roven pivotu** $A[q]$.
  - Každý prvek v pravé straně je **větší než pivot** $A[q]$.
  - Index pivotu q je vypočítán jako součást procedury rozdělení.
- **Panuj (Conquer):**
  - Rekurzivní volání quicksortu pro seřazení každého z podpolí $A[p \ldots q-1]$ a $A[q+1 \ldots r]$.
- **Kombinuj (Combine):**
  - Nedělá se **nic**: obě podpole jsou již seřazena, není potřeba žádná práce pro jejich kombinaci.
  - Všechny prvky v $A[p \ldots q-1]$ jsou seřazeny a menší nebo rovny $A[q]$.
  - Všechny prvky v $A[q+1 \ldots r]$ jsou seřazeny a větší než $A[q]$.
  - Celé podpole $A[p \ldots r]$ je tak seřazeno!
- Pro seřazení celého pole $A[1 \ldots n]$ s n prvky je počáteční volání QUICKSORT$(A, 1, n)$.

```
QUICKSORT(A, p, r)
1  if p < r
2      // Partition the subarray around the pivot, which ends up in A[q].
3      q = PARTITION(A, p, r)
4      QUICKSORT(A, p, q − 1)  // recursively sort the low side
5      QUICKSORT(A, q + 1, r)  // recursively sort the high side
```

- Klíčem k algoritmu je procedura `PARTITION`, která přeskupí podpole $A[p \ldots r]$ na místě.
- Vrátí index dělícího prvku mezi dvěma stranami rozdělení.
- `PARTITION` vždy vybere prvek $x = A[r]$ jako **pivot**.
- Během provádění procedury spadá každý prvek do jedné ze čtyř oblastí (některé mohou být prázdné).
- **Invarient cyklu (Loop Invariant):** Na začátku každé iterace cyklu for v řádcích 3-6 platí pro libovolný index pole k následující podmínky:
  1. Pokud $p \leq k \leq i$, pak $A[k] \leq x$ (béžová oblast).
  2. Pokud $i + 1 \leq k \leq j - 1$, pak $A[k] > x$ (modrá oblast).
  3. Pokud $k = r$, pak $A[k] = x$ (žlutá oblast).

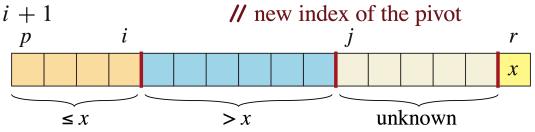
```
PARTITION(A, p, r)
1  x = A[r]                          // the pivot
2  i = p − 1                         // highest index into the low side
3  for j = p to r − 1                // process each element other than the pivot
4      if A[j] ≤ x                   // does this element belong on the low side?
5          i = i + 1                 // index of a new slot in the low side
6          exchange A[i] with A[j]   // put this element there
7  exchange A[i + 1] with A[r]       // pivot goes just to the right of the low side
8  return i + 1                      // new index of the pivot
```





PARTITION(A, p, r)

```
1  x = A[r]                          // the pivot
2  i = p − 1                         // highest index into the low side
3  for j = p to r − 1                // process each element other than the pivot
4      if A[j] ≤ x                   // does this element belong on the low side?
5          i = i + 1                     // index of a new slot in the low side
6          exchange A[i] with A[j]   // put this element there
7  exchange A[i + 1] with A[r]       // pivot goes just to the right of the low side
8  return i + 1                      // new index of the pivot
```



- **Finalizace:** Poslední dva řádky dokončují výměnou pivotu s levým prvkem větším než x, čímž se pivot přesune na správné místo v rozděleném poli, a poté vrátí nový index pivotu.
- Výstup nyní splňuje specifikace pro krok “rozděl”.
- **Invarient cyklu:** Na začátku každé iterace cyklu ’for’ platí pro libovolný index pole k:
  1. Pokud $p \leq k \leq i$, pak $A[k] \leq x$.
  2. Pokud $i + 1 \leq k \leq j - 1$, pak $A[k] > x$.
  3. Pokud $k = r$, pak $A[k] = x$.

PARTITION(A, p, r)

```
1  x = A[r]                          // the pivot
2  i = p − 1                         // highest index into the low side
3  for j = p to r − 1                // process each element other than the pivot
```

- Musíme ukázat, že invariant cyklu je pravdivý před první iterací, že každá iterace cyklu invariant udržuje, že se cyklus ukončí a že správnost vyplývá z invariantu po ukončení cyklu.
- **Inicializace:** Před první iterací cyklu máme $i = p - 1$ a $j = p$. Protože žádné hodnoty neleží mezi p a i a žádné hodnoty neleží mezi $i+1$ a $j-1$, první dvě podmínky invariantu cyklu jsou triviálně splněny. Přiřazení v řádku 1 splňuje třetí podmínku ($A[r] = x$).
- **Invarient cyklu:** Na začátku každé iterace cyklu 'for' platí pro libovolný index pole k:
  1. Pokud $p \leq k \leq i$, pak $A[k] \leq x$.
  2. Pokud $i + 1 \leq k \leq j - 1$, pak $A[k] > x$.
  3. Pokud $k = r$, pak $A[k] = x$.

```
3  for j = p to r − 1                  // process each element other than the pivot
4      if A[j] ≤ x                     // does this element belong on the low side?
5          i = i + 1                       // index of a new slot in the low side
6          exchange A[i] with A[j]     // put this element there
```

- **Iterace:** Uvažujeme dva případy, v závislosti na výsledku testu v řádku 4.
  - Pokud $A[j] > x$: Jediná akce v cyklu je inkrementace j. Po inkrementaci j platí druhá podmínka pro $A[j-1]$ a všechny ostatní záznamy zůstávají nezměněny.
  - Pokud $A[j] \leq x$: Cyklus inkrementuje i, prohodí $A[i]$ a $A[j]$, a poté inkrementuje j. Díky prohození máme nyní $A[i] \leq x$ a podmínka 1 je splněna. Podobně máme $A[j-1] > x$, protože prvek, který byl prohozen do $A[j-1]$, je podle invariantu cyklu větší než x.
- **Invarient cyklu:** Na začátku každé iterace cyklu 'for' platí pro libovolný index pole k:
  1. Pokud $p \leq k \leq i$, pak $A[k] \leq x$.
  2. Pokud $i + 1 \leq k \leq j - 1$, pak $A[k] > x$.
  3. Pokud $k = r$, pak $A[k] = x$.

```
3  for j = p to r − 1                  // process each element other than the pivot
4      if A[j] ≤ x                     // does this element belong on the low side?
5          i = i + 1                       // index of a new slot in the low side
6          exchange A[i] with A[j]     // put this element there
7  exchange A[i + 1] with A[r]         // pivot goes just to the right of the low side
8  return i + 1                        // new index of the pivot
```

- **Ukončení:** Cyklus provede přesně $r - p$ iterací, takže se ukončí, když $j = r$.

- V tomto okamžiku je neprozkoumané podpole $A[j \dots r-1]$ prázdné.
- Každý záznam v poli patří do jedné ze tří množin popsaných invariantem:
  1. Ty menší nebo rovné x (levá strana).
  2. Ty větší než x (pravá strana).
  3. Jednoprvková množina obsahující x (pivot).
- Časová složitost Quicksortu závisí na tom, jak vyvážené je každé rozdělení, což zase závisí na tom, které prvky jsou použity jako pivoty.
- **Vyvážené dělení:** Pokud jsou obě strany rozdělení přibližně stejné velikosti, algoritmus běží asymptoticky stejně rychle jako Merge Sort ($O(n \log n)$).
- **Nevyvážené dělení:** Pokud je dělení nevyvážené, může běžet asymptoticky stejně pomalu jako Insertion Sort ($O(n^2)$).
- Ačkoli Quicksort řadí na místě (in-place), množství paměti, které používá (kromě řazeného pole), není konstantní.
- Každé rekurzivní volání vyžaduje konstantní množství místa na zásobníku volání (runtime stack).
- Proto Quicksort vyžaduje paměť úměrnou maximální hloubce rekurze.
- V nejhorším případě to může být až $\Theta(n)$.
- K chování Quicksortu v nejhorším případě dochází, když dělení produkuje jeden podproblém s $n-1$ prvky a jeden s 0 prvky.
- Předpokládejme, že toto nevyvážené dělení nastává při každém rekurzivním volání.
- Náklady na dělení jsou $\Theta(n)$.
- Protože rekurzivní volání na pole velikosti 0 se jednoduše vrátí bez provedení čehokoli ($T(0) = \Theta(1)$), rekurence pro časovou složitost je:

$$\begin{aligned} T(n) &= T(n-1) + T(0) + \Theta(n) \\ &= T(n-1) + \Theta(n) \end{aligned}$$

- Součtem nákladů vynaložených na každé úrovni rekurze získáme aritmetickou řadu, která se vyhodnotí na $\Theta(n^2)$.
- Tedy, pokud je dělení maximálně nevyvážené na každé rekurzivní úrovni algoritmu, časová složitost je $\Theta(n^2)$.

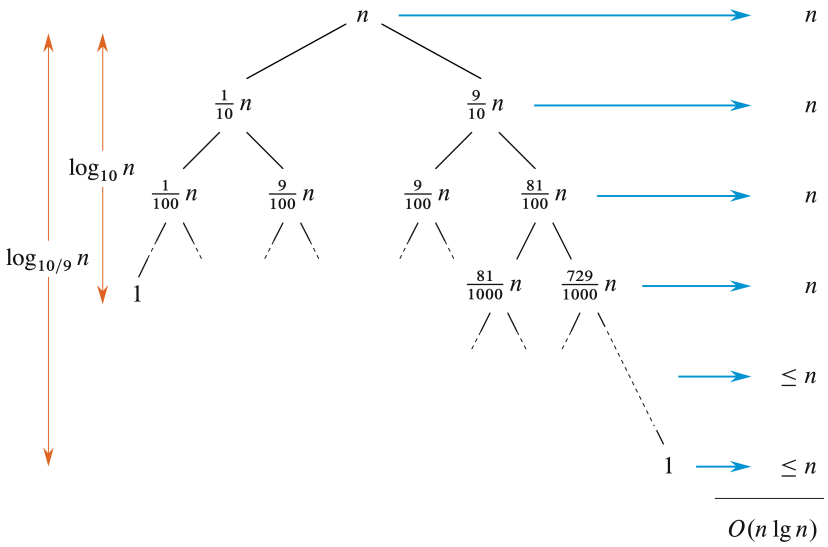
- **Důležité:** Časová složitost $\Theta(n^2)$ nastává, když je vstupní pole již zcela seřazeno (situace, ve které Insertion Sort běží v čase $O(n)$).
- V nejpravděpodobnějším rovnoměrném rozdělení produkuje PARTITION dva podproblémy, každý o velikosti nejvýše $n/2$.
  - Jeden je o velikosti $\lfloor (n-1)/2 \rfloor \leq n/2$.
  - Druhý je o velikosti $\lceil (n-1)/2 \rceil \leq n/2$.
- V tomto případě běží Quicksort mnohem rychleji. Horní hranice časové složitosti může být popsána rekurencí:

$$T(n) = 2T(n/2) + \Theta(n)$$

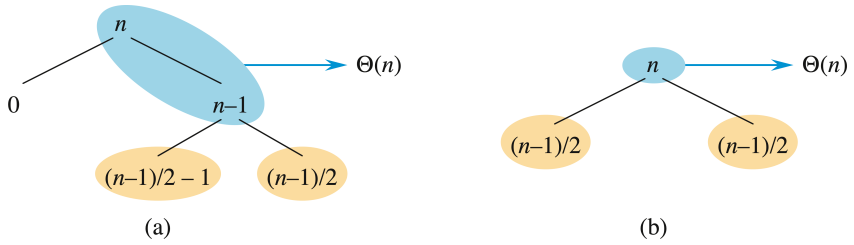
- Podle případu 2 Mater theoremu má tato rekurence řešení $T(n) = \Theta(n \log n)$.
- Tedy, pokud je dělení stejně vyvážené na každé úrovni rekurze, výsledkem je asymptoticky rychlejší algoritmus.
- Průměrná časová složitost Quicksortu je mnohem blíže nejlepšímu případu než nejhoršímu.
- Předpokládejme například, že algoritmus produkuje **proporcionální rozdělení 9 ku 1**.
- Pak získáme rekurenci:

$$T(n) = T(9n/10) + T(n/10) + \Theta(n)$$

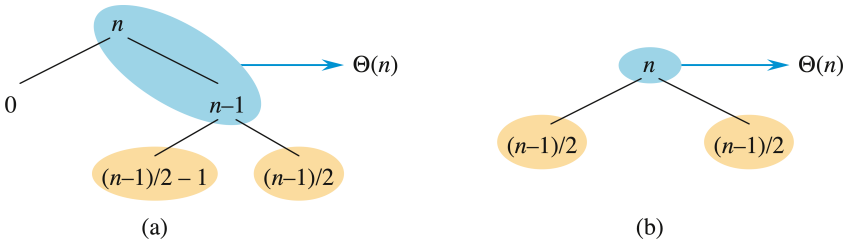
- Rekurze končí v hloubce $\log_{10} n = \Theta(\log n)$ (pro menší stranu).
- Rekurze končí v hloubce $\log_{10/9} n = \Theta(\log n)$ (pro větší stranu).
- Náklady na každé úrovni stromu rekurze jsou $O(n)$.
- **Závěr:** I s proporcionálním rozdělením 9 ku 1 na každé úrovni rekurze, což se intuitivně zdá být vysoce nevyvážené, běží Quicksort v čase $O(n \log n)$ – asymptoticky stejně jako by bylo rozdělení přesně uprostřed.
- Jakékoli rozdělení s konstantní proporcionalitou vede ke stromu rekurze hloubky $\Theta(\log n)$, kde náklady na každé úrovni jsou $O(n)$. Časová složitost je tedy $O(n \log n)$, kdykoli má rozdělení konstantní proporcionalitu. Poměr rozdělení ovlivňuje pouze konstantu skrytou v O-notaci.



- Abychom získali jasnou představu o očekávaném chování Quicksortu, musíme něco předpokládat o distribuci jeho vstupů.
- Jelikož Quicksort určuje seřazené pořadí pouze pomocí porovnání mezi vstupními prvky, jeho chování závisí na **relativním pořadí hodnot** v daných prvcích pole, nikoli na konkrétních hodnotách.
- Předpokládáme, že **všechny permutace vstupních čísel jsou stejně pravděpodobné** a že prvky jsou **různé**.
- V průměrném případě PARTITION produkuje směs "dobrých" a "špatných" rozdělení.
- V rekurzním stromu pro průměrné provedení PARTITION jsou dobrá a špatná rozdělení náhodně rozložena po celém stromě.
- Předpokládejme, že dobrá a špatná rozdělení střídají úrovně ve stromu a že dobrá rozdělení jsou nejlepším případem a špatná rozdělení jsou nejhorším případem.
- **Příklad:** Dvě po sobě jdoucí úrovně rekurzního stromu:
  - V kořeni jsou náklady na rozdělení n a vytvořená pole mají velikosti $n-1$ a 0 (nejhorší případ).
  - Na další úrovni se pole velikosti $n-1$ rozdělení na pole velikosti $(n-1)/2-1$ a $(n-1)/2$.



- Kombinace špatného rozdělení následovaného dobrým produkuje tři pole o velikostech 0, $(n-1)/2-1$ a $(n-1)/2$ s náklady na rozdělení $\Theta(n)+\Theta(n-1)=\Theta(n)$.
- Tato situace je nanejvýš o konstantní faktor horší než situace, kde se jedna úroveň rozdělí na dvě pole o velikosti $(n-1)/2$, s náklady $\Theta(n)$. A tato situace je vyvážená (obr. (b))!



- **Intuitivně:** Náklady $\Theta(n-1)$ špatného rozdělení se mohou "absorbovat" do nákladů $\Theta(n)$ dobrého rozdělení a výsledné rozdělení je dobré.
- **Závěr:** Časová složitost Quicksortu, když se úrovně střídají mezi dobrými a špatnými rozděleními, je podobná časové složitosti pouze pro dobrá rozdělení: stále $O(n \log n)$, ale s mírně větší konstantou skrytou O-notací.
- V předchozím jsme předpokládali, že všechny permutace vstupních čísel jsou stejně pravděpodobné. Tento předpoklad však ne vždy platí.
- **Randomizace** může být někdy přidána k algoritmu pro získání dobrého očekávaného výkonu pro všechny vstupy.
- Pro Quicksort randomizace přináší **rychlý a praktický algoritmus**. Mnoho softwarových knihoven poskytuje randomizovanou verzi Quicksortu jako preferovaný algoritmus pro řazení velkých datových sad.
- **O co jde?** Místo použití $A[r]$ jako pivotu, randomizovaná verze **náhodně vybere pivota** z pole $A[p \ldots r]$, kde každý prvek v $A[p \ldots r]$ má stejnou pravděpodobnost být vybrán.
- Poté tento prvek vymění s $A[r]$ před prováděním dělení.

- Protože pivot je vybrán náhodně, očekáváme, že rozdělení vstupního pole bude v průměru **rozumně dobře vyvážené**.
- Změny v PARTITION a QUICKSORT jsou malé.
- Nová procedura rozdělení, `RANDOMIZED-PARTITION`, jednoduše provede výměnu před provedením dělení.
- Nová procedura Quicksortu, `RANDOMIZED-QUICKSORT`, volá `RANDOMIZED--PARTITION` namísto `PARTITION`.

```
RANDOMIZED-PARTITION(A, p, r)
1  i = RANDOM(p, r)
2  exchange A[r] with A[i]
3  return PARTITION(A, p, r)
```

```
RANDOMIZED-QUICKSORT(A, p, r)
1  if p < r
2      q = RANDOMIZED-PARTITION(A, p, r)
3      RANDOMIZED-QUICKSORT(A, p, q − 1)
4      RANDOMIZED-QUICKSORT(A, q + 1, r)
```

- Dělení v nejhorším případě na každé úrovni rekurze má složitost $\Theta(n^2)$. Nyní to dokážeme.
- Použijeme substituční metodu.
- Nechť $T(n)$ je čas v nejhorším případě pro proceduru QUICKSORT na vstupu velikosti n.
- Protože PARTITION produkuje dva podproblémy s celkovou velikostí $n-1$, máme:

$$T(n) = \max_{0 \le q \le n-1} \{T(q) + T(n-1-q)\} + \Theta(n)$$

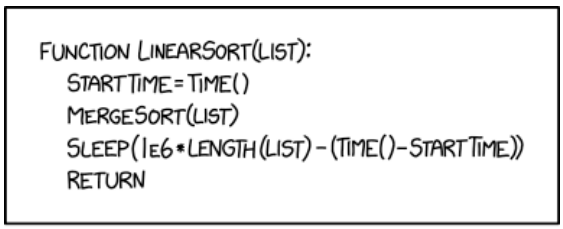
- **Hypotéza:** $T(n) \le cn^2$ pro nějakou konstantu $c > 0$.
- Dosazením hypotézy do rekurence:

$$T(n) \le \max_{0 \le q \le n-1} \{cq^2 + c(n-1-q)^2\} + \Theta(n) = c \cdot \max_{0 \le q \le n-1} \{q^2 + (n-1-q)^2\} + \Theta(n)$$

- Funkce $f(q) = q^2 + (n-1-q)^2 \le (n-1)^2$ pro $q = 0, \ldots, n-1$.
- Tedy: $T(n) \le c(n-1)^2 + \Theta(n) \le c(n^2 - 2n + 1) + \Theta(n) \le cn^2 - c(2n - 1) + \Theta(n)$ Volbou dostatečně velkého c tak, aby $c(2n-1)$ dominovalo $\Theta(n)$ dostaneme $T(n) \le cn^2$.

- **Závěr:** $T(n) = O(n^2)$. Protože jsme viděli, že v nejhorším případě je to $\Omega(n^2)$, celkově je časová složitost v nejhorším případě $\Theta(n^2)$.
- **Intuitivně:** Očekávaná doba běhu RANDOMIZED-QUICKSORTu je $O(n \log n)$. Pokud na každé úrovni rekurze dělení provedené RANDOMIZED-PARTITION umístí konstantní zlomek prvků na jednu stranu rozdělení, pak rekurzní strom má hloubku $\Theta(\log n)$ a na každé úrovni se provede $O(n)$ práce.
- I když přidáme několik nových úrovní s maximálně nevyváženým rozdělením mezi těmito úrovněmi, celkový čas zůstává $O(n \log n)$.
- Kombinací této horní hranice očekávané doby běhu s dolní hranicí nejlepšího případu $\Omega(n \log n)$ získáme očekávanou dobu běhu $\Theta(n \log n)$.
- **Předpoklad:** Hodnoty řazených prvků jsou **různé**.
- **Výběr pivota:**
  - Standardně: Poslední prvek (jako v naší 'PARTITION').
  - Randomizovaný: Náhodný prvek.
  - Medián ze tří: Vybere medián z prvního, středního a posledního prvku. To pomáhá snížit pravděpodobnost špatného pivota.
- **Zpracování malých podpolí:**
  - Pro velmi malá pole (např. $n \leq 10$) je Insert Sort často rychlejší než Quicksort kvůli menším konstantním faktorům.
  - Lze Quicksort upravit tak, aby pro malá podpole volal Insert Sort, nebo je ponechal neseřazené a seřadil celé pole jedním voláním Insert Sortu na konci.
- **Stabilita:** Quicksort není stabilní řadící algoritmus (pořadí prvků se stejnou hodnotou se může změnit).
- **Quicksort:** Algoritmus typu "rozděl a panuj", in-place řazení.
- **Procedura PARTITION:** Klíčová pro rozdělení pole kolem pivota. Běží v $\Theta(n)$.
- **Nejhorší případ:** $\Theta(n^2)$ (nevyvážené rozdělení).
- **Nejlepší a průměrný případ:** $\Theta(n \log n)$ (díky vyváženému rozdělení).
- **Randomizovaný Quicksort:**
  - Vybírá pivot náhodně, čímž eliminuje závislost na vstupních datech.
  - Zajišťuje, že očekávaný čas je $\Theta(n \log n)$ pro jakýkoli vstup.
- **Praktické výhody:** Rychlý v praxi, in-place.
- **Nevýhody:** Není stabilní, nejhorší případ $\Theta(n^2)$ (i když vzácný).

10 Řazení v lineárním čase

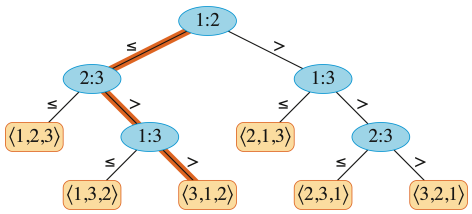


HOW TO SORT A LIST IN LINEAR TIME

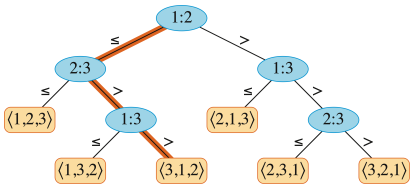
- Viděli jsme algoritmy, které umí řadit n čísel v čase $O(n \log n)$:
  - **Merge Sort** a **Heapsort** dosahují této horní hranice v nejhorším případě.
  - **Quicksort** ji dosahuje v průměru.
- Pro každý z těchto algoritmů můžeme vytvořit vstup, který způsobí běh v čase $\Omega(n \log n)$.
- Tyto algoritmy sdílejí zajímavou vlastnost: seřazené pořadí určují **pouze na základě porovnání** mezi vstupními prvky. Nazýváme je **srovnávací řadicí algoritmy**.
- **Dolní hranice pro řazení**
  - Dokážeme, že jakýkoli srovnávací řadicí algoritmus musí v nejhorším případě provést $\Omega(n \log n)$ porovnání.
  - Merge Sort a Heapsort jsou **asymptoticky optimální**.
- **Algoritmy s lineární časovou složitostí**
  - **Counting Sort** (Řazení počítáním)
  - **Radix Sort** (Radixové řazení)
  - **Bucket Sort** (Segmentové řazení)
- Tyto algoritmy používají **jiné operace než porovnávání** k určení seřazeného pořadí. Proto se na ně dolní hranice $\Omega(n \log n)$ nevztahuje.
- **Srovnávací řadicí algoritmus** používá pouze porovnání mezi prvky k získání informací o pořadí vstupní posloupnosti $\langle a_1, a_2, \ldots, a_n \rangle$.
- Pro dva prvky a_i a a_j provádí jeden z testů $a_i < a_j$, $a_i \leq a_j$, $a_i = a_j$, $a_i \geq a_j$, nebo $a_i > a_j$, aby určil jejich relativní pořadí.
- Nesmí kontrolovat hodnoty prvků ani získávat informace o pořadí jiným způsobem.

[0] Obrázek převzat z xkcd.

- **Předpoklad pro důkaz dolní hranice:** Všechny vstupní prvky jsou **různé**.
  - Dolní hranice platí i pro případ, kdy prvky nejsou různé.
  - Porovnání typu $a_i = a_j$ jsou zbytečná.
  - Porovnání $a_i \leq a_j$, $a_i \geq a_j$, $a_i > a_j$, a $a_i < a_j$ jsou ekvivalentní, protože poskytují stejné informace.
  - Proto předpokládáme, že všechna porovnání jsou typu $a_i \leq a_j$.
- Srovnávací řadicí algoritmy lze abstraktně znázornit pomocí **rozhodovacích stromů**.
- **Rozhodovací strom** je úplný binární strom (každý uzel je buď list nebo má oba potomky), který reprezentuje porovnání prováděná konkrétním řadicím algoritmem na vstupu dané velikosti.

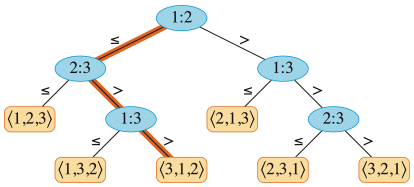


- Ignorují se řízení, přesun dat a všechny ostatní aspekty algoritmu.



- **Vnitřní uzel** je označen $i : j$ pro $i, j \in \{1, \ldots, n\}$, n je počet prvků vstupní posloupnosti.
  - Označuje porovnání $a_i \leq a_j$.
  - Levý podstrom určuje následná porovnání, pokud $a_i \leq a_j$.
  - Pravý podstrom určuje následná porovnání, pokud $a_i > a_j$.
- Příklad stromu pro $\langle a_1 = 6, a_2 = 8, a_3 = 5 \rangle$ (označeno červeně).
- Každý **list** je označen permutací $\langle \pi(1), \pi(2), \ldots, \pi(n) \rangle$.
  - Označuje seřazené pořadí $a_{\pi(1)} \leq a_{\pi(2)} \leq \cdots \leq a_{\pi(n)}$.
- Vykonání algoritmu odpovídá procházení cesty od kořene k listu.

- Každý správný řadicí algoritmus musí být schopen vygenerovat každou permutaci svého vstupu. Proto každá z $n!$ permutací na n prvcích musí být alespoň jedním z listů rozhodovacího stromu. Tyto listy musí být dosažitelné.



- Délka nejdelší cesty od kořene rozhodovacího stromu k jakémukoli jeho dosažitelnému listu představuje **nejhorší počet porovnání**, které odpovídající řadicí algoritmus provede.
- Nejhorší počet porovnání pro daný srovnávací řadicí algoritmus se rovná **výšce jeho rozhodovacího stromu**.
- Dolní hranice pro výšky všech rozhodovacích stromů, ve kterých se každá permutace objevuje jako dosažitelný list, je tedy dolní hranicí pro dobu běhu jakéhokoli srovnávacího řadicího algoritmu.

Věta 10.1. *Jakýkoli srovnávací řadicí algoritmus vyžaduje v nejhorším případě* $\Omega(n \log n)$ *porovnání.*

Důkaz.
- Nechť h je výška rozhodovacího stromu a ℓ je počet dosažitelných listů odpovídajících srovnávacímu řazení n prvků.
- Protože každá z $n!$ permutací vstupu se objeví jako jeden nebo více listů, platí $\ell \geq n!$
- Protože binární strom výšky h má nejvýše 2^h listů, platí $2^h \geq \ell \geq n!$
- Logaritmováním získáme: $h \geq \log(n!)$ (protože funkce log je monotónně rostoucí)
- Víme, že $\log(n!) = \Theta(n \log n)$
- Tedy: $h = \Omega(n \log n)$

□

10.1 Counting Sort

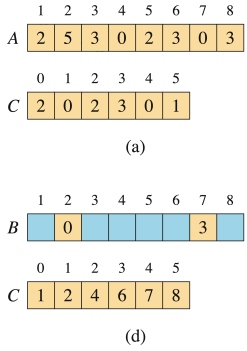
- Counting sort předpokládá, že **každý z** n **vstupních prvků je celé číslo v rozsahu** 0 **až** k, pro nějaké celé číslo k.
- Běží v čase $\Theta(n + k)$. Pokud $k = O(n)$, Counting sort běží v čase $\Theta(n)$.
- Counting sort nejprve pro každý vstupní prvek x určí počet prvků menších nebo rovných x.

- Poté tuto informaci použije k přímému umístění prvku x na jeho pozici ve výstupním poli. (Například, pokud je 17 prvků menších nebo rovných x, pak x patří na výstupní pozici 17.)
- Musíme tento plán mírně upravit, abychom zvládli situaci, kdy několik prvků má stejnou hodnotu, protože nechceme, aby všechny skončily na stejné pozici.
- Procedura COUNTING-SORT přijímá na vstupu pole $A[1 \ldots n]$, velikost n tohoto pole a limit k pro nezáporné celočíselné hodnoty v A. Vrátí svůj seřazený výstup v poli $B[1 \ldots n]$ a pro dočasné pracovní úložiště používá pole $C[0 \ldots k]$.

```
COUNTING-SORT(A, n, k)
1  let B[1:n] and C[0:k] be new arrays
2  for i = 0 to k
3      C[i] = 0
4  for j = 1 to n
5      C[A[j]] = C[A[j]] + 1
6  // C[i] now contains the number of elements equal to i.
7  for i = 1 to k
8      C[i] = C[i] + C[i − 1]
9  // C[i] now contains the number of elements less than or equal to i.
10 // Copy A to B, starting from the end of A.
11 for j = n downto 1
12     B[C[A[j]]] = A[j]
13     C[A[j]] = C[A[j]] − 1    // to handle duplicate values
14 return B
```

- **Fáze 1 (řádky 2-3):** Inicializace pole C na nuly. Trvá $\Theta(k)$ času.
- **Fáze 2 (řádky 4-5):** Průchod pole A. Nalezením prvku s hodnotou i se inkrementuje $C[i]$. Po tomto cyklu $C[i] =$ počet prvků rovných i. Trvá $\Theta(n)$ času.
- **Fáze 3 (řádky 7-8):** Průchod C. $C[i] =$ počet prvků $\leq i$. Trvá $\Theta(k)$ času.
- **Fáze 4 (řádky 11-13):** Průchod A v opačném pořadí, umístění prvků do B, zajišťuje **stabilitu**. Trvá $\Theta(n)$ času.


```
COUNTING-SORT(A, n, k)
1  let B[1:n] and C[0:k] be new arrays
2  for i = 0 to k
3      C[i] = 0
4  for j = 1 to n
5      C[A[j]] = C[A[j]] + 1
6  // C[i] now contains the number of elements equal to i.
7  for i = 1 to k
8      C[i] = C[i] + C[i − 1]
9  // C[i] now contains the number of elements less than or equal to i.
10 // Copy A to B, starting from the end of A.
11 for j = n downto 1
12     B[C[A[j]]] = A[j]
13     C[A[j]] = C[A[j]] − 1    // to handle duplicate values
14 return B
```

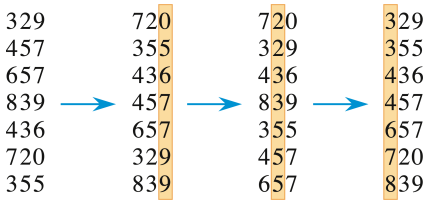



```
COUNTING-SORT(A, n, k)
 1  let B[1:n] and C[0:k] be new arrays
 2  for i = 0 to k
 3      C[i] = 0
 4  for j = 1 to n
 5      C[A[j]] = C[A[j]] + 1
 6  // C[i] now contains the number of elemen
 7  for i = 1 to k
 8      C[i] = C[i] + C[i − 1]
 9  // C[i] now contains the number of elemen
10  // Copy A to B, starting from the end of A
11  for j = n downto 1
12      B[C[A[j]]] = A[j]
13      C[A[j]] = C[A[j]] − 1    // to handle
14  return B
```

- Cyklus for v řádcích 2-3 trvá $\Theta(k)$ času.
- Cyklus for v řádcích 4-5 trvá $\Theta(n)$ času.
- Cyklus for v řádcích 7-8 trvá $\Theta(k)$ času.
- Cyklus for v řádcích 11-13 trvá $\Theta(n)$ času.
- Celkový čas je tedy $\Theta(k+n)$.
- V praxi se Counting sort obvykle používá, když máme $k = O(n)$, v takovém případě je doba běhu $\Theta(n)$.
- Counting sort může překonat dolní hranici $\Omega(n \log n)$, protože **není srovnávacím řadicím algoritmem**.
  - Ve skutečnosti se v kódu nikde nevyskytují žádná porovnání mezi vstupními prvky.
  - Místo toho Counting sort používá skutečné hodnoty prvků k indexování do pole.
- Důležitou vlastností Counting sortu je, že je **stabilní**: prvky se stejnou hodnotou se objeví ve výstupním poli ve stejném pořadí, v jakém se objevily ve vstupním poli.
  - Tedy, řeší shody mezi dvěma prvky pravidlem, že kterýkoli prvek se objeví dříve ve vstupním poli, objeví se dříve i ve výstupním poli.
- Vlastnost stability je obvykle důležitá pouze tehdy, když se s řazeným prvkem přenášejí i přidružená data (satellite data).
- Stabilita Counting sortu je důležitá i z jiného důvodu: Counting sort se často používá jako podprogram v Radix sortu. Aby Radix sort fungoval správně, Counting sort musí být stabilní.

10.2 Radix Sort

- Radix sort byl algoritmus používaný v děrovačkových strojích.
- Pro desetinná čísla používá každý sloupec pouze 10 pozic. d-ciferné číslo zabírá d sloupců.
- Radix sort řadí podle **nejméně významné číslice jako první**.
  - Algoritmus poté kombinuje sloupce do jediného.
  - Poté se vše znovu seřadí podle druhé nejméně významné číslice.
  - Proces pokračuje, dokud nejsou čísla seřazena podle všech d číslic.
- Pouze d průchodů je potřeba k seřazení.



- Aby Radix sort fungoval správně, řazení podle číslic musí být **stabilní**.
  - RADIX-SORT předpokládá, že každý prvek v poli $A[1 \ldots n]$ má d číslic, kde číslice 1 je číslice nejnižšího řádu a číslice d je číslice nejvyššího řádu.

```
RADIX-SORT(A, n, d)
1  for i = 1 to d
2      use a stable sort to sort array A[1 : n] on digit i
```

- RADIX-SORT neříká, který stabilní řadicí alg. použít, běžně se používá COUNTING-SORT.

Lemma 10.2. *Pro n d-ciferných čísel, kde každá číslice může nabývat až k hodnot, RADIX-SORT správně řadí tato čísla v čase $\Theta(d(n+k))$, pokud použitý stabilní řadicí algoritmus trvá $\Theta(n+k)$.*

Důkaz. Korektnost plyne indukcí pro řazený sloupec. Doba běhu závisí na použitém stabilním řadicím algoritmu. Každý průchod n d-cifernými čísly trvá $\Theta(n+k)$, je d průchodů, tj. $\Theta(d(n+k))$. □

- Pokud d je konstanta a $k = O(n)$, můžeme radix sort spustit v lineárním čase.

Lemma 10.3. *Pro n b-bitových čísel a libovolné kladné celé číslo $r \leq b$, RADIX-SORT správně řadí tato čísla v čase $\Theta((b/r)(n+2^r))$, pokud použitý stabilní řadicí algoritmus trvá $\Theta(n+k)$ pro vstupy v rozsahu 0 až k.*

Důkaz (shrnutí). Pro hodnotu $r \leq b$ vnímáme každý klíč jako $d = \lceil b/r \rceil$ číslic po r bitech. Každá číslice je celé číslo v rozsahu 0 až $2^r - 1$, takže můžeme použít Counting sort s $k = 2^r - 1$. Každý průchod Counting sortu trvá $\Theta(n + k) = \Theta(n + 2^r)$ a existuje d průchodů, celkem tedy $\Theta(d(n + 2^r)) = \Theta((b/r)(n + 2^r))$. □

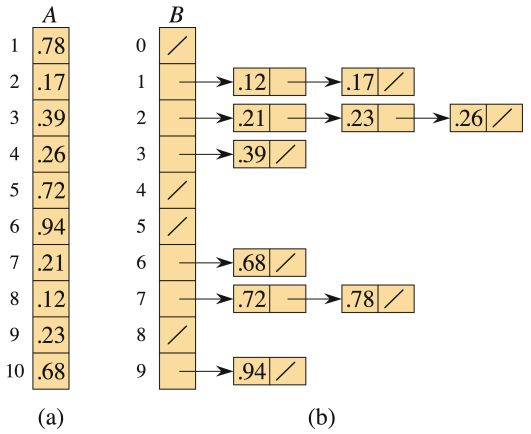
Např. 32-bit číslo jako čtyři 8-bitová čísla, tj. $b = 32$, $r = 8$, $k = 2^r - 1 = 255$ a $d = b/r = 4$.

- **Optimální r:** Pro minimalizaci výrazu $(b/r)(n + 2^r)$:
  - Pokud $b < \lfloor \log n \rfloor$: Volba $r = b$ dává časovou složitost $\Theta(n)$, což je asymptoticky optimální.
  - Pokud $b \geq \lfloor \log n \rfloor$: Volba $r = \lfloor \log n \rfloor$ dává nejlepší časovou složitost s přesností na konstantní faktor, a to $\Theta(bn/\log n)$.
- Je Radix sort lepší než algoritmy založené na porovnání, jako je Quicksort?
- Pokud $b = O(\log n)$ a $r \approx \log n$, pak časová složitost Radix sortu je $\Theta(n)$, což se zdá být lepší než očekávaná doba běhu Quicksortu $\Theta(n \log n)$.
- **Důležité rozdíly v konstantách:**
  - Radix sort může provést méně průchodů než Quicksort přes n klíčů, ale každý průchod Radix sortu může trvat podstatně déle.
  - Který řadicí algoritmus je lepší, závisí na vlastnostech implementace, použitém stroji (např. Quicksort často efektivněji využívá hardwarové cache) a vstupních datech.
- Verze Radix sortu, která používá Counting sort jako stabilní řadicí algoritmus, **neřadí na místě**, což mnoho srovnávacích řadicích algoritmů s časovou složitostí $\Theta(n \log n)$ dělá.
  - Pokud je tedy primární paměť cenná, může být lepší volbou in-place algoritmus, jako je Quicksort.

10.3 Bucket Sort

- Bucket sort předpokládá, že vstup je z **rovnoměrné distribuce** a má v průměrném případě časovou složitost $O(n)$.
- Je rychlý, protože předpokládá něco o vstupu: Vstup je generován náhodným procesem, který rovnoměrně a nezávisle distribuuje prvky v intervalu $[0, 1)$.
- Bucket sort rozděluje interval $[0, 1)$ na n stejně velkých podintervalů, neboli **kbelíků (buckets)**, a poté distribuuje n vstupních čísel do těchto kbelíků.

- Jelikož jsou vstupy rovnoměrně a nezávisle distribuovány, neočekáváme, že mnoho čísel spadne do stejného kbelíku.
- Pro vytvoření výstupu jednoduše seřadíme čísla v každém kbelíku a poté projdeme kbelíky v pořadí a vypíšeme prvky z každého.



- Procedura `BUCKET-SORT` předpokládá, že vstup je pole $A[1 \ldots n]$ a že každý prvek $A[i]$ v poli splňuje $0 \leq A[i] < 1$.
- Kód vyžaduje pomocné pole $B[0 \ldots n-1]$ seznamů (kbelíků).

BUCKET-SORT(A, n)

```
1  let B[0 : n − 1] be a new array
2  for i = 0 to n − 1
3      make B[i] an empty list
4  for i = 1 to n
5      insert A[i] into list B[⌊n · A[i]⌋]
6  for i = 0 to n − 1
7      sort list B[i] with insertion sort
8  concatenate the lists B[0], B[1], . . . , B[n − 1] together in order
9  return the concatenated lists
```

- **Korektnost:** Dva prvky $A[i]$ a $A[j]$ s $A[i] \leq A[j]$:
  - Buď jdou do stejného kbelíku, kde je seřadí řádky 6-7.
  - Nebo $A[i]$ jde do kbelíku s nižším indexem, kde je řádek 8 seřadí správně.
- Všechny řádky kromě řádku 7 trvají v nejhorším případě $O(n)$.
- Musíme analyzovat celkový čas strávený n voláními Insertion Sortu v řádku 7.

- Nechť n_i je náhodná proměnná označující počet prvků umístěných v kbelíku $B[i]$. Insertion sort běží v kvadratickém čase $O(n_i^2)$.
- Časová složitost Bucket sortu je: $T(n) = \Theta(n) + \sum_{i=0}^{n-1} O(n_i^2)$
- Průměrná časová složitost Bucket sortu je $\Theta(n)$ (výpočet mimo rozsah našeho kurzu).
- I když vstup není z rovnoměrné distribuce, Bucket sort může stále běžet v lineárním čase, pokud součet čtverců velikostí kbelíků je lineární v celkovém počtu prvků.
- **Dolní hranice pro porovnávací řazení:** $\Omega(n \log n)$ (důkaz pomocí rozhodovacích stromů). Žádný porovnávací algoritmus nemůže být rychlejší.
- **Řazení v lineárním čase:** Algoritmy, které překonávají dolní hranici $\Omega(n \log n)$ tím, že **nepoužívají pouze porovnávání**. Místo toho využívají informace o struktuře nebo rozsahu vstupních dat.
- **Counting Sort:**
  - Pro celá čísla v rozsahu $0 \dots k$.
  - Čas: $\Theta(n + k)$. Pokud $k = \mathcal{O}(n)$, pak $\Theta(n)$.
  - Je **stabilní**.
- **Radix Sort:**
  - Řadí čísla po cifrách od nejméně významné.
  - Používá **stabilní řadící algoritmus** (např. Counting Sort) pro každou cifru.
  - Čas: $\Theta(d(n + k))$, kde d je počet cifer a k je počet hodnot na cifře.
- **Bucket Sort:**
  - Pro vstupní data uniformně náhodně rozložená v rozsahu $[0, 1)$.
  - Rozdělí data do kyblíků a seřadí každý kyblík.
  - Očekávaný čas: $\Theta(n)$.

11 Úvod do Pořádkových Statistik

- **i-tá pořádková statistika** sady n prvků je i-tý nejmenší prvek.
  - Minimum je 1. pořádková statistika ($i = 1$).
  - Maximum je n-tá pořádková statistika ($i = n$).
- **Medián** je neformálně “střední bod” sady.

  - Pokud je n liché, medián je unikátní na pozici $i = (n+1)/2$.
  - Pokud je n sudé, existují dva mediány:
    * Spodní medián na pozici $i = n/2$.
    * Horní medián na pozici $i = n/2 + 1$.
  - V našem textu budeme "mediánem" označovat **spodní medián**.
- Cílem je nalézt i-tou pořádkovou statistiku z množiny n **různých** čísel.
  - Vstup: Množina A s n různými čísly a celé číslo i, kde $1 \leq i \leq n$.
  - Výstup: Prvek $x \in A$, který je větší než přesně $i-1$ ostatních prvků v A.
- Problém lze triviálně vyřešit v čase $O(n \log n)$ seřazením a následným výběrem i-tého prvku. Jde to **asymptoticky rychleji**?

Nalezení minima (nebo maxima):

- Kolik porovnání je nutných k určení minima sady n prvků?
- Horní hranice je $n-1$ porovnání.
- Algoritmus MINIMUM(A, n):

```
MINIMUM(A, n)
1  min = A[1]
2  for i = 2 to n
3      if min > A[i]
4          min = A[i]
5  return min
```

- **Optimálnost:** Tento algoritmus je optimální. Každý prvek mimo vítěze musí prohrát alespoň jeden zápas (porovnání). Proto je $n-1$ porovnání nezbytných.

Nalezení minima a maxima nezávisle: $2n - 2 = \Theta(n)$ porovnání.

Efektivnější přístup: Nalezení minima a maxima současně.

- Lze dosáhnout s nejvýše $3\lfloor n/2 \rfloor$ porovnání.
- **Myšlenka:** Udržujeme jak aktuální minimum, tak maximum. Místo zpracování každého prvku jednotlivě (což by stálo 2 porovnání na prvek), **zpracováváme prvky v párech**.
  - Porovnáme prvky v páru mezi sebou.
  - Menší prvek porovnáme s aktuálním minimem.
  - Větší prvek porovnáme s aktuálním maximem.

  - Celkem **3 porovnání na každé 2 prvky**.

Inicializace:

- Pokud n je **liché**: Nastavíme minimum i maximum na hodnotu prvního prvku. Zpracujeme zbývajících $n-1$ prvků v párech.
  - Porovnání: $3 \cdot (n-1)/2 = 3\lfloor n/2 \rfloor$.
- Pokud n je **sudé**: Provedeme 1 porovnání na prvních 2 prvcích k určení počátečního minima a maxima. Zpracujeme zbývajících $n-2$ prvků v párech.
  - Porovnání: $1 + 3 \cdot (n-2)/2 = 1 + 3n/2 - 3 = 3n/2 - 2$.

Závěr: V obou případech je celkový počet porovnání **nejvýše** $3\lfloor n/2 \rfloor$.

- Problém nalezení i-té pořádkové statistiky se zdá být obtížnější než nalezení minima. Překvapivě je asymptotická doba běhu pro oba problémy stejná: $\Theta(n)$.
- Nyní představíme algoritmus **rozděl a panuj** pro problém výběru.
- Algoritmus `RANDOMIZED-SELECT` je modelován podle Quicksortu.
  - Podobně jako Quicksort rekurzivně rozděluje vstupní pole.
  - Na rozdíl od Quicksortu, který rekurzivně zpracovává obě strany partition, `RANDOMIZED-SELECT` pracuje **pouze na jedné straně** partition.
  - Tento rozdíl se projevuje v analýze: zatímco Quicksort má očekávanou dobu běhu $\Theta(n \log n)$, `RANDOMIZED-SELECT` má očekávanou dobu běhu $\Theta(n)$, za předpokladu, že prvky jsou různé.
- `RANDOMIZED-SELECT`(A, p, r, i)
  - používá proceduru `RANDOMIZED-PARTITION` – je to **randomizovaný algoritmus**.
  - vrací i-tý nejmenší prvek z pole $A[p \ldots r]$, kde $1 \le i \le r - p + 1$.

```
RANDOMIZED-SELECT(A, p, r, i)
1  if p == r
2      return A[p]        // 1 ≤ i ≤ r − p + 1 when p == r means that i = 1
3  q = RANDOMIZED-PARTITION(A, p, r)
4  k = q − p + 1
5  if i == k
6      return A[q]        // the pivot value is the answer
7  elseif i < k
8      return RANDOMIZED-SELECT(A, p, q − 1, i)
9  else return RANDOMIZED-SELECT(A, q + 1, r, i − k)
```

- **Řádek 1-2 (základní případ):** Pokud pole $A[p \dots r]$ obsahuje pouze jeden prvek, vrátí se tento prvek.
- **Řádek 3 (partition):** Volání `RANDOMIZED-PARTITION` rozdělí pole $A[p \dots r]$ na dvě (možná prázdná) podpole $A[p \dots q-1]$ a $A[q+1 \dots r]$. Prvky v $A[p \dots q-1]$ jsou menší nebo rovny $A[q]$, které je zase menší než prvky v $A[q+1 \dots r]$. $A[q]$ je **pivotní prvek**.
- **Řádek 4 (výpočet k):** k je počet prvků v levém podpoli plus pivot.
- **Řádek 5-6 (kontrola pivota):** Pokud je pivot i-tý nejmenší prvek (tj. $i = k$), vrátí se $A[q]$.
- **Řádek 7-9 (rekurzivní volání):**
  - Pokud $i < k$, požadovaný prvek leží na levé straně partition. Rekurzivně se volá `RANDOMIZED-SELECT` na $A[p \dots q-1]$ pro i-tý prvek.
  - Pokud $i > k$, požadovaný prvek leží na pravé straně partition. Již známe k hodnot menších než i-tý nejmenší prvek (prvky v $A[p \dots q]$). Proto je požadovaný prvek $(i-k)$-tý nejmenší prvek z $A[q+1 \dots r]$.

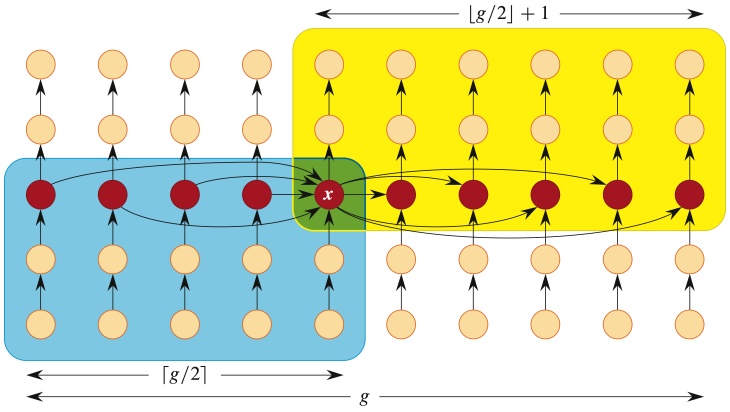
	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	p	r	i
$A^{(0)}$	6	19	4	12	14	9	15	7	8	11	3	13	2	5	10	1	15	5
$A^{(1)}$	6	4	12	10	9	7	8	11	3	13	2	5	14	19	15	1	12	5
$A^{(2)}$	3	2	4	10	9	7	8	11	6	13	5	12	14	19	15	4	12	2
$A^{(3)}$	3	2	4	10	9	7	8	11	6	12	5	13	14	19	15	4	11	2
$A^{(4)}$	3	2	4	5	6	7	8	11	9	12	10	13	14	19	15	4	5	2
$A^{(5)}$	3	2	4	5	6	7	8	11	9	12	10	13	14	19	15	5	5	1

- **Nejhorší případ:**
  - Pivot je vždy nejmenší nebo největší prvek.
  - Rekurzivní volání na pole velikosti $n-1$.
  - Rekurentní vztah: $T(n) = T(n-1) + O(n)$.
  - Řešení: $T(n) = O(n^2)$.
  - Stejné jako nejhorší případ Quicksortu.

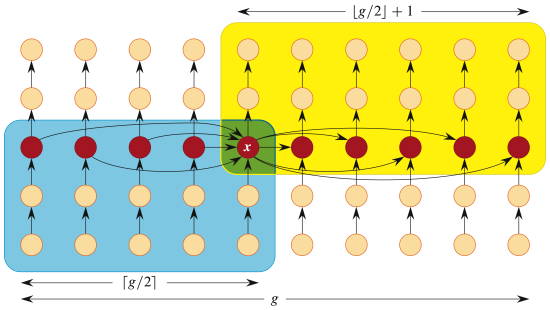
- **Nejlepší případ:**
  - Pivot je vždy medián.
  - Rekurzivní volání na pole velikosti $n/2$.
  - Rekurentní vztah: $T(n) = T(n/2) + O(n)$.
  - Řešení: $T(n) = O(n)$.
- Nyní se podíváme na pozoruhodný algoritmus pro výběr, jehož doba běhu je $\Theta(n)$ v **nejhorším případě**.
- Ačkoliv `RANDOMIZED-SELECT` dosahuje lineárního očekávaného času, jeho doba běhu v nejhorším případě je kvadratická.
- Algoritmus `SELECT` dosahuje lineárního času v nejhorším případě, ale **není zdaleka tak praktický** jako `RANDOMIZED-SELECT`. Je spíše teoretického zájmu.
- **Hlavní myšlenka:** Podobně jako `RANDOMIZED-SELECT`, `SELECT` nalezne požadovaný prvek rekurzivním rozdělením vstupního pole.
- Na rozdíl od `RANDOMIZED-SELECT`, `SELECT` zaručuje dobré rozdělení tím, že **volí prokazatelně dobrý pivot** při rozdělování pole.
  - Klíčová myšlenka spočívá v tom, že pivota **hledá rekurzivně**.
  - Existují tedy dvě rekurzivní volání `SELECT`: jedno k nalezení dobrého pivota a druhé k rekurzivnímu nalezení požadované pořádkové statistiky.
- Použitý algoritmus rozdělení je podobný deterministickému `PARTITION` z Quicksortu, ale modifikovaný tak, aby bral prvek, kolem kterého se má rozdělovat, jako dodatečný vstupní parametr (`PARTITION-AROUND`).

```
SELECT(A, p, r, i)
1  while (r − p + 1) mod 5 ≠ 0
2      for j = p + 1 to r                    // put the minimum into A[p]
3          if A[p] > A[j]
4              exchange A[p] with A[j]
5      // If we want the minimum of A[p : r], we're done.
6      if i == 1
7          return A[p]
8      // Otherwise, we want the (i − 1)st element of A[p + 1 : r].
9      p = p + 1
10     i = i − 1
11 g = (r − p + 1)/5                         // number of 5-element groups
12 for j = p to p + g − 1                    // sort each group
13     sort ⟨A[j], A[j + g], A[j + 2g], A[j + 3g], A[j + 4g]⟩ in place
14 // All group medians now lie in the middle fifth of A[p : r].
15 // Find the pivot x recursively as the median of the group medians.
16 x = SELECT(A, p + 2g, p + 3g − 1, ⌈g/2⌉)
17 q = PARTITION-AROUND(A, p, r, x)   // partition around the pivot
18 // The rest is just like lines 3–9 of RANDOMIZED-SELECT.
19 k = q − p + 1
20 if i == k
21     return A[q]                           // the pivot value is the answer
22 elseif i < k
23     return SELECT(A, p, q − 1, i)
24 else return SELECT(A, q + 1, r, i − k)
```

- Pseudokód nejprve sníží počet prvků v poli, aby byl dělitelný pěti. Provede se 0 až 4 krát, přičemž každé opakování přeuspořádá prvky $A[p:r]$ tak, aby $A[p]$ byl minimální prvek.
  - Pokud $i = 1$, tj. chceme minimální prvek, procedura jej vrátí v řádku 7.
  - Jinak SELECT odstraní minimum z podpole $A[p:r]$ a pokračuje v hledání $(i-1)$-ního prvku v $A[p+1:r]$.
- Rozdělíme prvky v $A[p:r]$ na $g = (r-p+1)/5$ skupin po pěti prvcích.
  - První pětielementová skupina je $\langle A[p], A[p+g], A[p+2g], A[p+3g], A[p+4g]\rangle$.
  - Druhá je $\langle A[p+1], A[p+g+1], A[p+2g+1], A[p+3g+1], A[p+4g+1]\rangle$.
  - Až po poslední, která je $\langle A[p+g-1], A[p+2g-1], A[p+3g-1], A[p+4g-1], A[r]\rangle$.



- Setřiď každou skupinu, tj. $A[j] \leq A[j+g] \leq A[j+2g] \leq A[j+3g] \leq A[j+4g]$ pro $j = p, p+1, \ldots, p+g-1$.
- Medián skupiny je $A[j+2g]$, tj. mediány skupin (červené) leží v $A[p+2g \ldots p+3g-1]$.
- Řádek 16 určí pivot x rekurzivním voláním SELECT k nalezení mediánu ($\lceil g/2 \rceil$-tého nejmenšího) z g mediánů.
- Řádek 17 použije modifikovaný algoritmus PARTITION-AROUND, aby rozdělil prvky $A[p:r]$ kolem x, čímž získá index q takový, že $A[q] = x$, prvky v $A[p:q]$ jsou nejvýše x a prvky v $A[q:r]$ jsou alespoň x.
- Zbytek kódu odpovídá algoritmu RANDOMIZED-SELECT.
- Pokud je pivot x i-tý největší, procedura jej vrátí.
- Jinak procedura rekurzivně volá samu sebe buď na $A[p:q-1]$, nebo na $A[q+1:r]$, v závislosti na hodnotě i.
- **Krok 1 (Rozdělení do skupin):** $O(n)$
- **Krok 2 (Mediány skupin):** Seřazení 5 prvků je konstantní. $\lceil n/5 \rceil$ skupin. Celkem $O(n)$.
- **Krok 3 (Medián mediánů):** Rekurzivní volání na pole velikosti $\lceil n/5 \rceil$. Čas $T(\lceil n/5 \rceil)$.
- **Krok 4 (Rozdělení):** $O(n)$
- **Krok 5 (Rekurzivní volání):**



- **Krok 5 (Rekurzivní volání):**
  - Aspoň polovina skupin má medián menší nebo roven x. Těchto mediánů je aspoň $\lceil(\lceil n/5\rceil)/2\rceil - 2$. Každá z těchto skupin má 3 prvky menší nebo rovny svému mediánu, takže alespoň $3\,(\lceil(\lceil n/5\rceil)/2\rceil - 2)$ prvků je menších nebo rovno x. To je přibližně $3n/10$ prvků.
  - Stejně tak alespoň $3n/10$ prvků je větších nebo rovno x.
  - Z toho vyplývá, že rekurzivní volání je maximálně na pole velikosti $n - 3n/10 = 7n/10$.
- **Rekurentní rovnice:** $T(n) \leq T(\lceil n/5\rceil) + T(\lceil 7n/10\rceil) + O(n)$
- **Závěr:** Algoritmus ‘SELECT‘ má časovou složitost $O(n)$ v nejhorším případě.
- **Pořádkové statistiky:** Nalezení k-tého nejmenšího prvku v množině.
- **Minimum/Maximum:** Lze nalézt v $O(n)$ čase. Současně v $\approx 3n/2$ porovnáních.
- **‘RANDOMIZED-SELECT‘:**
  - Adaptace Quicksortu.
  - Očekávaná časová složitost $O(n)$.
  - Nejhorší případ $O(n^2)$.
  - Prakticky velmi rychlý.
- **‘SELECT‘ (Median-of-Medians):**
  - Zaručuje lineární čas $O(n)$ v nejhorším případě.
  - Složitější algoritmus.
  - Rekurzivní volání na mediány mediánů zajišťuje dobrý pivot.
- Problém výběru je základní úlohou v informatice a má efektivní řešení.
- Existuje kompromis mezi jednoduchostí implementace (‘RANDOMIZED-SELECT‘) a garancí výkonu v nejhorším případě (‘SELECT‘).

- Aplikace pořádkových statistik sahají od statistiky po optimalizaci dalších algoritmů.
- Důkladné pochopení těchto algoritmů je klíčové pro efektivní návrh algoritmů.