

Techniky návrhu algoritmů

Karel Richta a kol.

Přednášky byly připraveny s pomocí materiálů, které vyrobili Marko Berezovský, Petr Felkel, Josef Kolář, Michal Píše a Pavel Tvrdík

Katedra počítačů
Fakulta elektrotechnická
České vysoké učení technické v Praze

© Karel Richta a kol., 2017

Datové struktury a algoritmy, B6B36DSA
02/2017, Lekce 2

<https://moodle.fel.cvut.cz/course/view.php?id=1238>



Evropský sociální fond
Praha & EU: Investujeme do vaší budoucnosti

Techniky návrhu algoritmů

- Při návrhu řešení nějakého problému se snažíme najít algoritmus, který daný problém řeší.
- Zpravidla lze rychle navrhnout naivní řešení, které přímo vychází z definice problému. Toto řešení ale zpravidla nebývá příliš efektivní – problém je zadán tak, aby jeho popis byl co nejjednodušší, naivní řešení toto zadání kopíruje.
- **Př.:** problém řazení posloupnosti čísel lze vyjádřit tak, že hledáme permutaci vstupní posloupnosti, která je seřazená (např. vzestupně). Naivní řešení tedy bude probírat všechny permutace vstupní posloupnosti a každou testovat, zda je seřazená. V nejhorším případě tedy najdeme správné řešení jako poslední a složitost řazení bude $O(n!)$.
- Pokud by byla vstupní data konečná a dostatečně malá, lze předpočítat všechny výstupy a odpovídající algoritmus by měl konstantní složitost $O(1)$.
- Ve skutečnosti jsou ale vstupní data zpravidla v principu nekonečná a pro návrh algoritmu musíme použít nějakou metodu založenou na rekurzi.

Rekurze

Rekurze je metoda zápisu algoritmu, kde se stejný postup opakuje na částech vstupních dat.

Výhody:

- úspora: zápis kódu je kratší a konečným zápisem lze definovat ∞ množinu dat,
- přirozené vyjádření: opakování a sebepodobnost jsou v přírodě běžné,
- intuitivní vyjádření: explicitně pojmenovává to, co se opakuje v menším,
- expresivní vyjádření: rekurzivní specifikace umožňuje poměrně snadnou analýzu složitosti a ověření správnosti (viz např. **MERGESORT** dále).

Nevýhody:

- interpretace nebo provádění rekurzivního kódu používá systémový zásobník pro předání parametrů volání a proto vyžaduje systémovou paměť navíc,
- dynamická alokace systémového zásobníku a ukládání parametrů na něj navíc představuje také časovou režii.

Kvůli výpočetní úplnosti ale prakticky všechny dnešní programovací jazyky rekurzi povolují.

Technika „rozděl a panuj“ (divide-and-conquer, divide et impera)

- Algoritmy navrhované technikou rozděl a panuj mají tři fáze:
 - rozdělení problému na menší podproblémy (divide),
 - rekurzivní volání sama sebe na menší podproblémy (conquer) a
 - sestavení výsledného řešení z řešení podproblémů (combine).
- Jako každá rekurze, i algoritmy založené na technice rozděl a panuj musí někdy svůj rekurzivní sestup zastavit. Typicky se tak děje u triviálních velikostí problému.
- Dělení na podproblémy použijeme pro netriviální velikosti dat.
- Zpravidla lze rozlišit dělení na dělení vyvážené, kdy se vstupní data snažíme rozdělit na několik velikostí vyvážených podskupin.
- Další rozlišení technik rozděl a panuj lze charakterizovat podle důrazu kladeného buď na analýzu vstupních dat (divide), nebo na syntézu výstupu (combine).

Řešení řazení technikou rozděl a panuj

- Vyvážené dělení vstupní posloupnosti na přibližně stejně velké části:
 - Důraz kladen na analýzu vstupu – snažíme se rozdělit vstupní posloupnost na dvě přibližně stejně velké části, kde první obsahuje data, která jsou všechna menší než data druhé skupiny – řazení dělením (**QUICK-SORT**).
 - Důraz kladen na syntézu výstupu – vstupní posloupnost rozdělíme bez rozmýšlení na dvě přibližně stejně velké části. Při syntéze slučujeme seřazené posloupnosti do výsledku – řazení slučováním (**MERGE-SORT**).
- Nevyvážené dělení vstupní posloupnosti na část obsahující jeden prvek a zbytek:
 - Důraz kladen na analýzu vstupu – snažíme se ve vstupní posloupnosti najít minimální prvek, který vložíme do čela výstupní posloupnosti – řazení přímým výběrem (**SELECTION-SORT**).
 - Důraz kladen na syntézu výstupu – ze vstupní posloupnosti oddělíme bez rozmýšlení první prvek, který při syntéze zařadíme do výsledku na správné místo – řazení zatřídováním (**INSERTION-SORT**).

Př.: Řazení slučováním (MERGE-SORT)

Rekurzivní řešení - potřebujeme seřadit obsah pole A mezi indexy p a r

1. Najdeme přibližný střed q a rozdělíme A na dvě části
2. Rekurzivně seřadíme obě části
3. Seřazené části sloučíme

MERGE-SORT(A, p, r)

1. **if** $p < r$ **then** {
2. $q = \lfloor (p + r)/2 \rfloor$;
3. MERGE-SORT(A, p, q);
4. MERGE-SORT($A, q + 1, r$);
5. MERGE(A, p, q, r)
6. }

Časová složitost $T(n)$:

Pro $n=1$ – konstantní

Pro $n>1$ – 2-krát seřadit poloviny + sloučení

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

Tj. $T(n) = \Theta(n \ln n)$... naučíme se později

Typy rekurze

- Lineární rekurze: V těle algoritmu je pouze jedno rekurzivní volání anebo jsou dvě, ale vyskytují se v disjunktních větvích podmíněných příkazů a nikdy se neprovedou současně.

- Příklad: Faktoriál

$$\text{FAC}(n) = 1 \quad \text{if } n = 1,$$

$$\text{FAC}(n) = n * \text{FAC}(n - 1) \quad \text{if } n > 1.$$

- Koncová rekurze (tail recursion): rekurzivní volání je posledním příkazem algoritmu, po kterém se už neprovádějí žádné další "odložené" operace.

- Příklad: Největší společný dělitel (Euclidův algoritmus): Necht' $n \geq m \geq 0$.

$$\text{GCD}(n, m) = n \quad \text{if } m = 0,$$

$$\text{GCD}(n, m) = \text{GCD}(m, \text{REMAINDER}(n, m)) \quad \text{if } m > 0.$$

Typy rekurze (pokr.)

- Kaskádní rekurze: V těle algoritmu jsou vedle sebe aspoň dvě rekurzivní volání. Viz MERGESORT, QUICKSORT.

- Příklad: Fibonacciho čísla

$$F(n) = 1 \quad \text{if } n = 1, 2$$

$$F(n) = F(n - 1) + F(n - 2) \quad \text{if } n > 2.$$

- Vnořená rekurze: Rekurzivní funkce, jejíž argumenty jsou specifikovány rekurzivně.

- Příklad: Ackermannova funkce (není tzv. primitivně rekurzivní)

$$A(m, n) = n + 1 \quad \text{if } m = 0,$$

$$A(m, n) = A(m - 1, 1) \quad \text{if } m > 0 \text{ and } n = 0,$$

$$A(m, n) = A(m - 1, A(m, n - 1)) \quad \text{if } m > 0 \text{ and } n > 0.$$

- Neuvěřitelně rychle rostoucí funkce, viz

<http://mathworld.wolfram.com/AckermannFunction.html>.

Neefektivita rekurze

- Řešení pomocí rekurze nemusí být nejefektivnější, zejména pokud je rekurze nelineární.
- Uvažme problém násobení matic A a B :

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

- Pro $n=1$ je: $T(1) = \Theta(1)$.
- Pro matice rozměru 2×2 (podobně pro libovolnou mocninu 2) :

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

- Tj.:

$$C_{11} = A_{11}B_{11} + A_{12}B_{21}$$

$$C_{12} = A_{11}B_{12} + A_{12}B_{22}$$

$$C_{21} = A_{21}B_{11} + A_{22}B_{21}$$

$$C_{22} = A_{21}B_{12} + A_{22}B_{22}$$

Řešení bez rekurze

```
SQUARE-MATRIX-MULTIPLY(A, B)
```

```
1  n = A.rows  
2  let C be a new n × n matrix  
3  for i = 1 to n  
4      for j = 1 to n  
5          cij = 0  
6          for k = 1 to n  
7              cij = cij + aik · bkj  
8  return C
```

- Evidentně je složitost pro čtvercové matice rozměru n dána 3-mi vnořenými cykly v rozsahu od 1 do n :

$$T(n) = \Theta(n^3)$$

Rekurzivní řešení

- Uvažme pro představu, že n je mocnina 2 (abychom mohli matici rozdělit napůl v obou směrech).

SQUARE-MATRIX-MULTIPLY-RECURSIVE(A, B)

```

1   $n = A.rows$ 
2  let  $C$  be a new  $n \times n$  matrix
3  if  $n == 1$ 
4       $c_{11} = a_{11} \cdot b_{11}$ 
5  else partition  $A, B$ , and  $C$ 
6       $C_{11} = \text{SQUARE-MATRIX-MULTIPLY-RECURSIVE}(A_{11}, B_{11})$ 
           +  $\text{SQUARE-MATRIX-MULTIPLY-RECURSIVE}(A_{12}, B_{21})$ 
7       $C_{12} = \text{SQUARE-MATRIX-MULTIPLY-RECURSIVE}(A_{11}, B_{12})$ 
           +  $\text{SQUARE-MATRIX-MULTIPLY-RECURSIVE}(A_{12}, B_{22})$ 
8       $C_{21} = \text{SQUARE-MATRIX-MULTIPLY-RECURSIVE}(A_{21}, B_{11})$ 
           +  $\text{SQUARE-MATRIX-MULTIPLY-RECURSIVE}(A_{22}, B_{21})$ 
9       $C_{22} = \text{SQUARE-MATRIX-MULTIPLY-RECURSIVE}(A_{21}, B_{12})$ 
           +  $\text{SQUARE-MATRIX-MULTIPLY-RECURSIVE}(A_{22}, B_{22})$ 
10 return  $C$ 

```

$$C_{11} = A_{11}B_{11} + A_{12}B_{21}$$

$$C_{12} = A_{11}B_{12} + A_{12}B_{22}$$

$$C_{21} = A_{21}B_{11} + A_{22}B_{21}$$

$$C_{22} = A_{21}B_{12} + A_{22}B_{22}$$

Složitost rekurzivního řešení

- Evidentně pro $n=1$ je: $T(1) = \Theta(1)$.
- Pro $n > 0$ je nutno vzít v úvahu náročnost rozdělení na submatice, což je $\Theta(1)$.
- Dále je nutno rekurzivně vynásobit 8 dvojic polovičních matic, což je $8 \times T(n/2)$.
- Výsledné matice je pak nutno sečíst, tj. 4-krát sečíst 2 matice velikosti $n/2 \times n/2$, což je $\Theta(n^2)$.
- Celkově tedy

$$T(n) = \Theta(1) + 8 \times T(n/2) + \Theta(n^2)$$

- Lze ukázat, že z toho plyne celková složitost:

$$T(n) = \Theta(n^3)$$

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

Strassenovo násobení matic

- Hlavní myšlenka: snížíme počet rekurzivních volání na sedm za cenu navýšení počtu sčítání (i přesto se to vyplatí).

$$C_{11} = A_{11}B_{11} + A_{12}B_{21} \quad S_1 = B_{12} - B_{22} \quad P_1 = A_{11} \cdot S_1$$

$$C_{12} = A_{11}B_{12} + A_{12}B_{22} \quad S_2 = A_{11} + A_{12} \quad P_2 = S_2 \cdot B_{22}$$

$$C_{21} = A_{21}B_{11} + A_{22}B_{21} \quad S_3 = A_{21} + A_{22} \quad P_3 = S_3 \cdot B_{11}$$

$$C_{22} = A_{21}B_{12} + A_{22}B_{22} \quad S_4 = B_{21} - B_{11} \quad P_4 = A_{22} \cdot S_4$$

$$S_5 = A_{11} + A_{22} \quad P_5 = S_5 \cdot S_6$$

$$S_6 = B_{11} + B_{22} \quad P_6 = S_7 \cdot S_8$$

$$S_7 = A_{12} - A_{22} \quad P_7 = S_9 \cdot S_{10}$$

$$S_8 = B_{21} + B_{22}$$

$$S_9 = A_{21} - A_{11}$$

$$S_{10} = B_{11} + B_{12}$$

$$T(1) = \Theta(1)$$

$$T(n) = 7T(n/2) + \Theta(n^2)$$

$$T(n) = \Theta(n^{\log_2 7})$$

Rekurze vs. iterace

- Rekurzivní programování má základní oporu v teoretické informatice, kde bylo dokázáno, že:
 1. Každou funkci, která může být implementovaná na vonNeumannovském počítači, lze vyjádřit rekurzivně bez použití iterace.
 2. Každá rekurzivní funkce se dá vyjádřit iterativně s použitím zásobníku (implicitní zásobník se stane viditelný uživateli).
- Koncovou rekurzi lze vždy nahradit iterací bez nutnosti zásobníku. Ta bývá zpravidla efektivnější.
- V rekurzivní proceduře se při rekurzivním volání téhož podprogramu (procedury nebo funkce) musí na systémový zásobník uložit hodnoty parametrů a lokálních proměnných volajícího podprogramu.
- Při návratu po ukončení tohoto vnořeného volání se tyto hodnoty obnoví.
- Výška zásobníku se rovná hloubce stromu rekurzivních volání procedury.
- Rekurzivní výpočet má tedy skrytou paměťovou náročnost úměrnou hloubce tohoto stromu.

Př.: Binární hledání (BINARY-SEARCH)

Zadání:

- Procedura BINARY-SEARCH prochází seřazené pole A v zadaném rozsahu $A[low..high]$ a hledá zadanou hodnotu v . Vrací index nalezené hodnoty, nebo NIL jako příznak, že hodnota v v daném úseku nebyla nalezena.
- Označení jako binární hledání je založeno na myšlence, že obsah pole A lze rozdělit na dvě části a podle hodnoty v místě dělení hledat dále jen v jedné z těchto částí (také se mluví o hledání binárním půlením).
- Zkusíme navrhnout rekurzivní i iterativní verzi tohoto algoritmu jako odpovídající procedury RECURSIVE-BINARY-SEARCH a ITERATIVE-BINARY-SEARCH s parametry BINARY-SEARCH($A, v, low, high$), kde A je prohledávané pole, v je hledaná hodnota a $low..high$ je vymezení prohledávaného úseku.
- Naivní řešení hrubou silou je postupné procházení pole A v rozsahu $low..high$ a porovnávání $A[i]$ s hodnotou v . Složitost této verze bude zřejmě $\Theta(n)$, neboť v nejhorším případě musím projít úsek celý a porovnat všechny prvky. Zkusme najít řešení efektivnější.

Rekurzivní verze BINARY-SEARCH

RECURSIVE-BINARY-SEARCH($A, v, low, high$)

if $low > high$

return NIL

$mid = \lfloor (low + high) / 2 \rfloor$

if $v == A[mid]$

return mid

elseif $v > A[mid]$

return RECURSIVE-BINARY-SEARCH($A, v, mid + 1, high$)

else return RECURSIVE-BINARY-SEARCH($A, v, low, mid - 1$)

- Procedura vždy skončí (rekurze pracuje vždy s menším a menším rozsahem).
- Složitost je dána: $T(n) = T(n/2) + \Theta(1) = \Theta(\lg n)$.

Iterativní verze BINARY-SEARCH

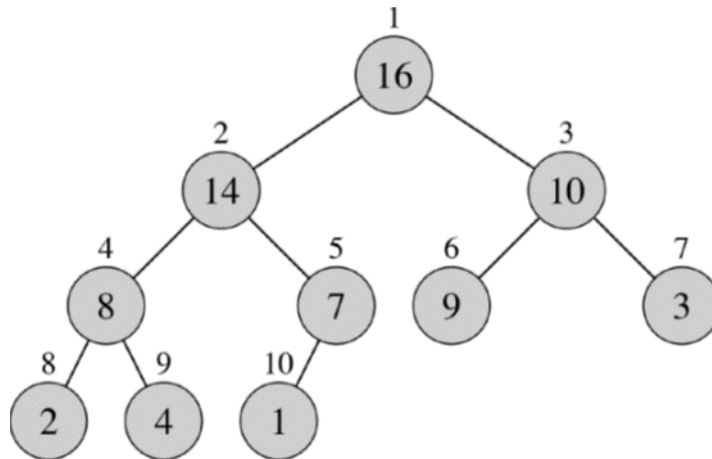
ITERATIVE-BINARY-SEARCH($A, v, low, high$)

```
while  $low \leq high$   
     $mid = \lfloor (low + high) / 2 \rfloor$   
    if  $v == A[mid]$   
        return  $mid$   
    elseif  $v > A[mid]$   
         $low = mid + 1$   
    else  $high = mid - 1$   
return NIL
```

- Procedura vždy skončí (indexy v iteraci se přibližují).
- Složitost je dána: $T(n) = T(n/2) + \Theta(1) = \Theta(\lg n)$.

Př.: Halda

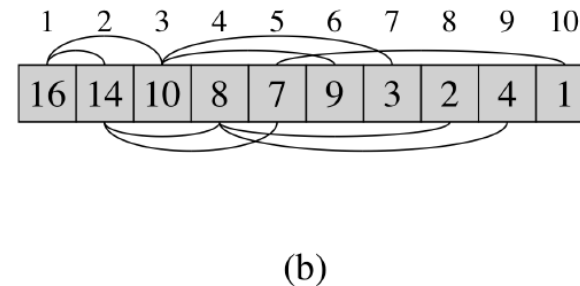
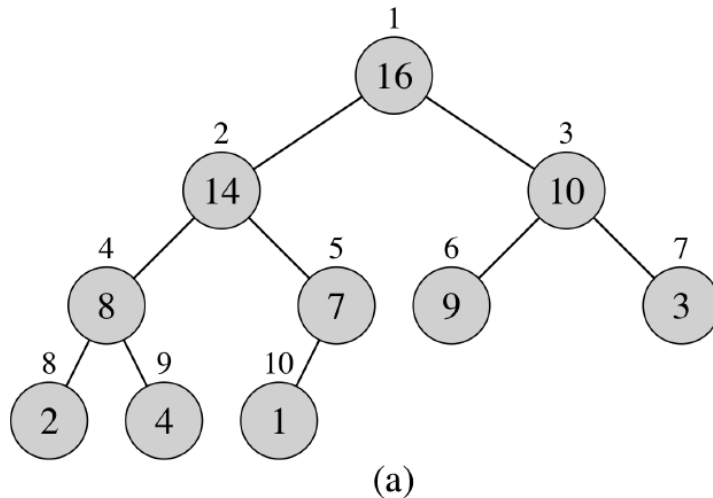
- (Binární) halda (heap) je téměř úplný (binární) strom. Neúplná může být pouze poslední vrstva listů a to ještě jen zprava:



- Halda může být uspořádána vzestupně (MIN-HEAP), kdy je minimum na vrcholu (v kořenu), nebo sestupně (MAX-HEAP), kdy je na vrcholu maximum – tu budeme používat (viz obr.).
- Hloubka uzlu v haldě (HEIGHT) je rovna počtu hran z uzlu do nejbližšího listu stromu. Hloubka haldy = hloubka kořene stromu = $\Theta(\lg n)$.

Budování haldy

- Halda může být uložena v poli A – viz obr.:



- Kořen haldy je $A[1]$, rodič $A[i]$ je $A[\lfloor i/2 \rfloor]$, levý potomek $A[i]$ je $A[2*i]$, pravý potomek $A[i]$ je $A[2*i + 1]$, tj.:

$$\text{PARENT}(i) = \lfloor i/2 \rfloor$$

$$\text{LEFT}(i) = 2*i$$

$$\text{RIGHT}(i) = 2*i + 1 \text{ (lze realizovat posuny).}$$

Vlastnosti haldy

- Pro haldy takové, že největší element je v kořenu haldy a směrem k listům se hodnoty snižují, platí vlastnost MAX-HEAP: pro všechny uzly s indexem i platí (s výjimkou kořene):

$$\forall i > 1 . A[\text{PARENT}(i)] \geq A[i] \quad (\text{MAX-HEAP})$$

- Indukcí a na základě transitivity MAX-HEAP lze ukázat, že tato vlastnost zaručuje, že maximální element je kořen haldy. Tuto vlastnost haldy využijeme později při návrhu algoritmu HEAPSORT.
- Nejprve navrhne proceduru MAX-HEAPIFY(A, i), která bude pro daný prvek zjišťovat, zda není menší než jeho potomci (splňuje vlastnost MAX-HEAP). Pokud ji nespĺňuje, přehodí prvky tak, aby MAX-HEAP platilo.
- Předpokládáme, že levý i pravý podstrom stromu s kořenem v i splňují vlastnost MAX-HEAP.
- Po skončení procedury MAX-HEAPIFY(A, i), bude podstrom s kořenem v i splňovat vlastnost MAX-HEAP.
- Pozn.: Obecně lze konstruovat k -ární haldy, nejen binární.

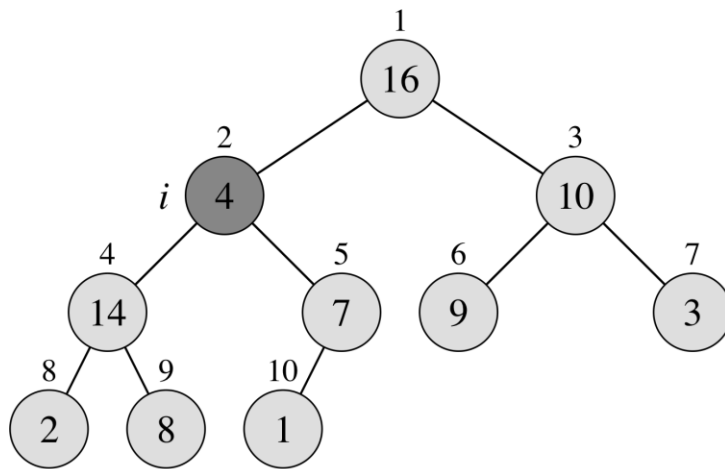
Rekurzivní MAX-HEAPIFY

Rekurzivní algoritmus MAX-HEAPIFY (tvorba haldy rekurzí):

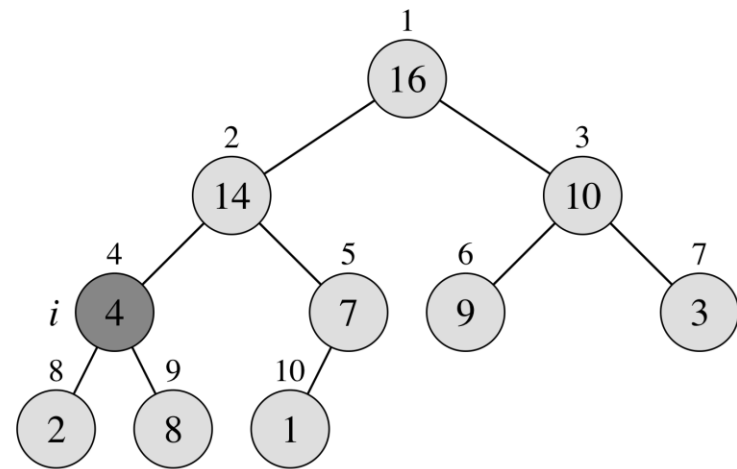
MAX-HEAPIFY(A, i) {

1. $l = \text{LEFT}(i)$;
 2. $r = \text{RIGHT}(i)$;
 3. **if** ($l \leq \text{HEAP-SIZE}(A) \ \& \ A[l] > A[i]$)
 4. **then** $Largest = l$;
 5. **else** $Largest = i$;
 6. **if** ($r \leq \text{HEAP-SIZE}(A) \ \& \ A[r] > A[Largest]$)
 7. **then** $Largest = r$;
 8. **if** ($Largest \neq i$)
 9. **then** $\{A[i] \leftrightarrow A[Largest] ; \text{MAX-HEAPIFY}(A, Largest)\}$
- }

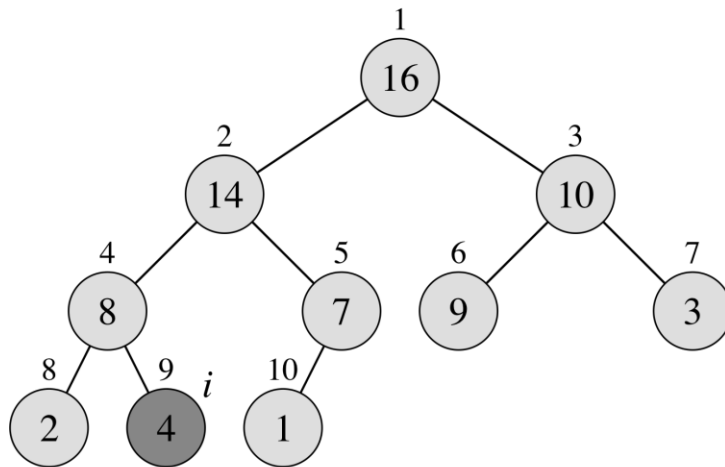
Jak to funguje



(a)

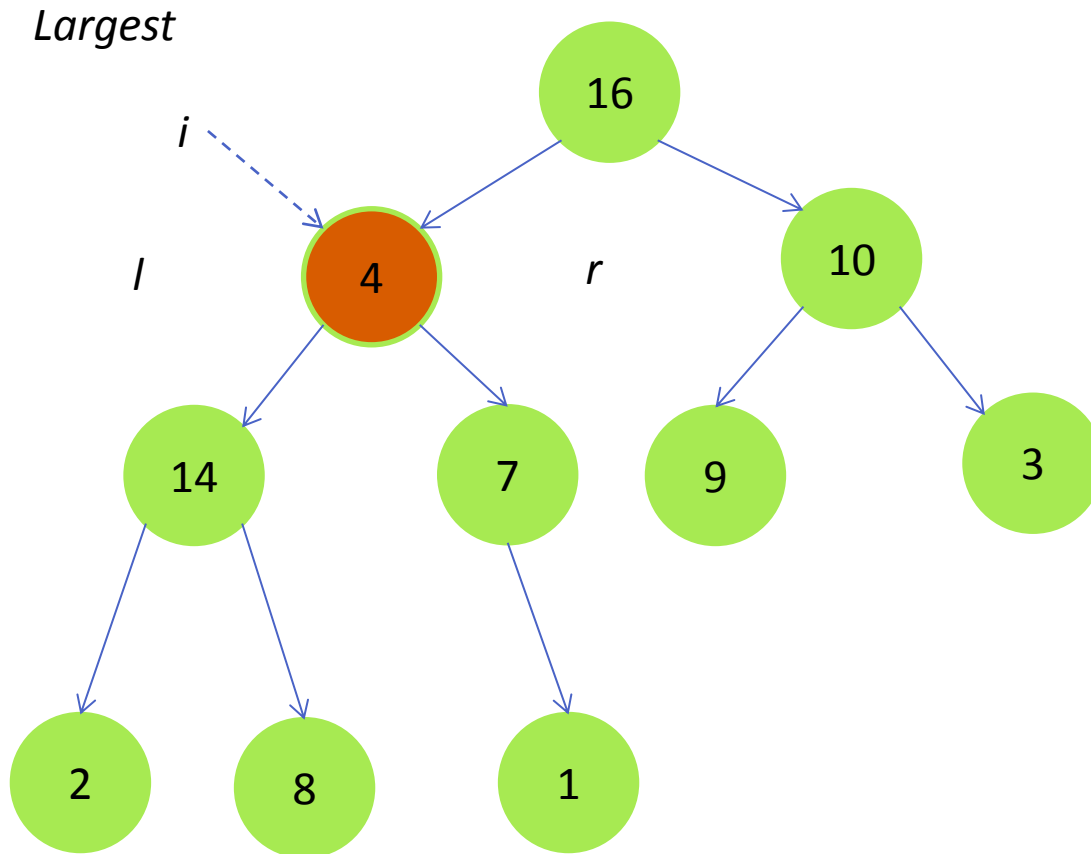


(b)



(c)

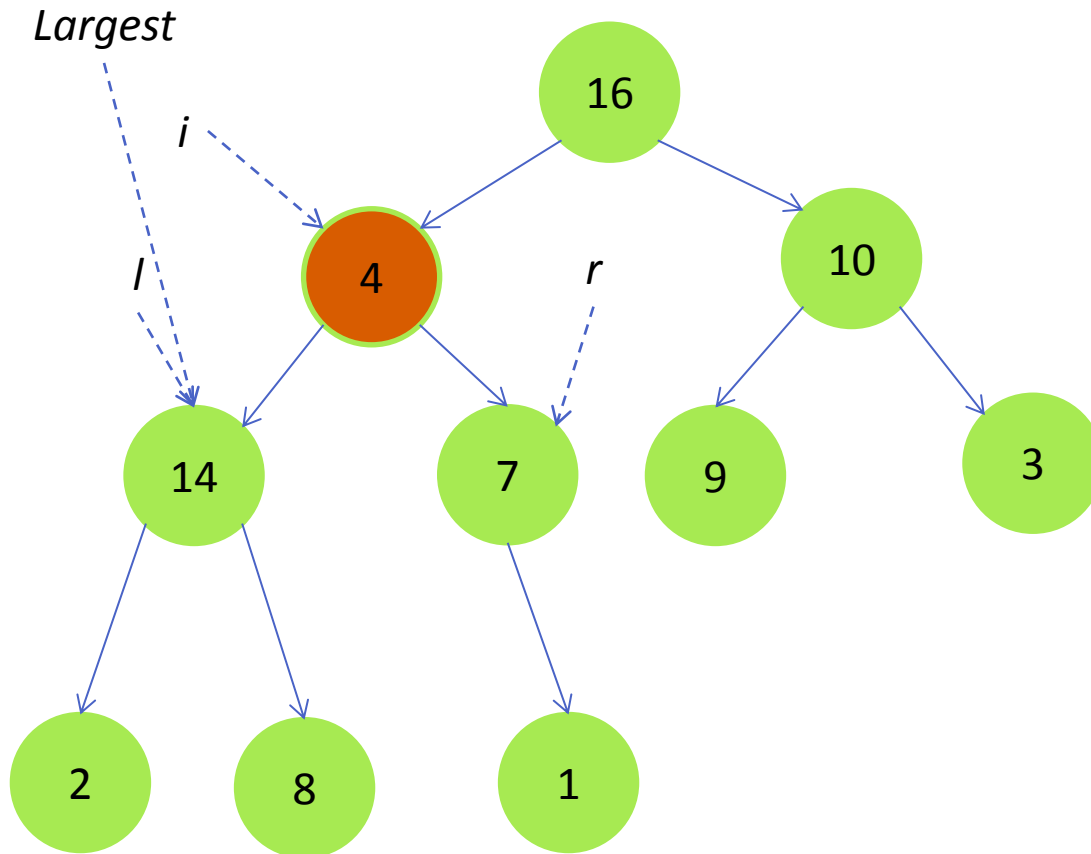
Jak funguje MAX-HEAPIFY



```

MAX-HEAPIFY(A, i) {
1.  l = LEFT(i);
2.  r = RIGHT(i);
3.  if (l ≤ HEAP-SIZE(A) &
4.     A[l] > A[i])
5.    then Largest = l;
6.  else Largest = i;
7.  if (r ≤ HEAP-SIZE(A) &
8.     A[r] > A[Largest])
9.    then Largest = r;
10. if (Largest ≠ i)
11.  then {
12.    A[i] ↔ A[Largest];
13.    MAX-HEAPIFY(A, Largest)}
}
  
```

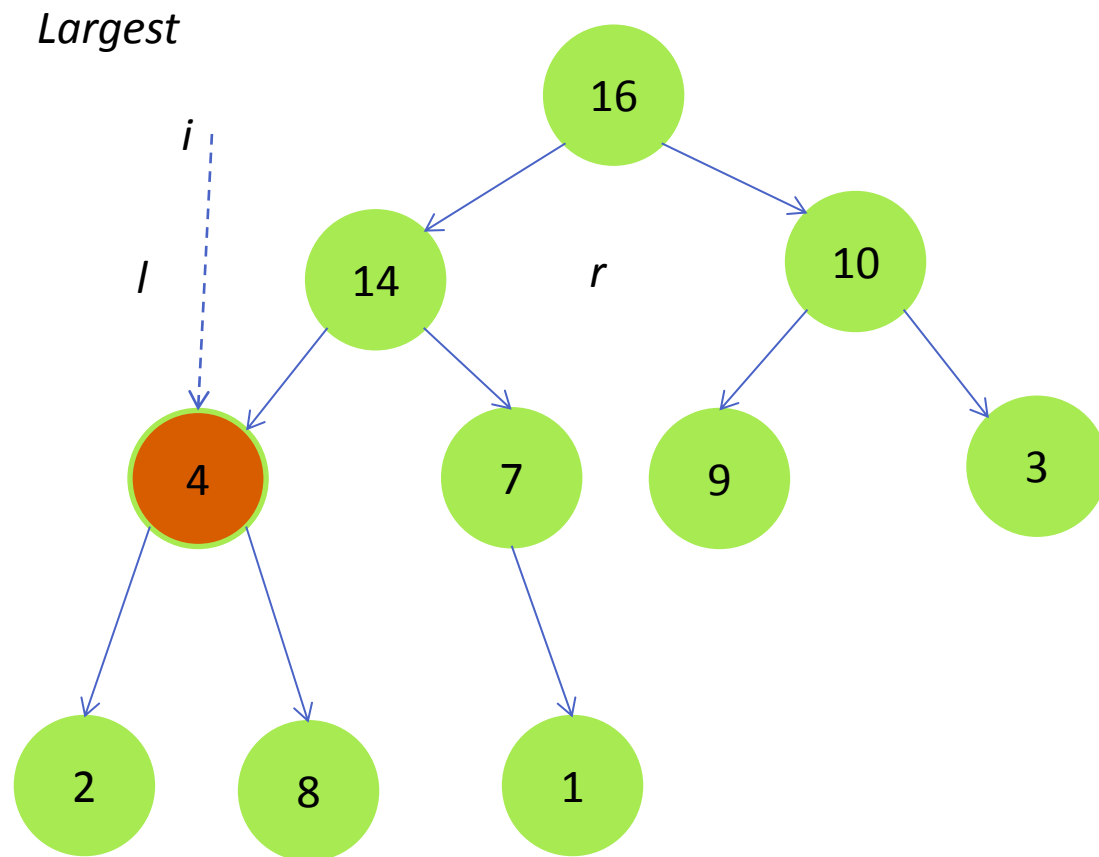
Jak funguje MAX-HEAPIFY



```

MAX-HEAPIFY(A, i) {
1.  l = LEFT(i);
2.  r = RIGHT(i);
3.  if (l ≤ HEAP-SIZE(A) &
4.     A[l] > A[i])
5.    then Largest = l;
6.  else Largest = i;
7.  if (r ≤ HEAP-SIZE(A) &
8.     A[r] > A[Largest])
9.    then Largest = r;
10. if (Largest ≠ i)
11. then {
12.   A[i] ↔ A[Largest] ;
13.   MAX-HEAPIFY(A, Largest)}
}
  
```

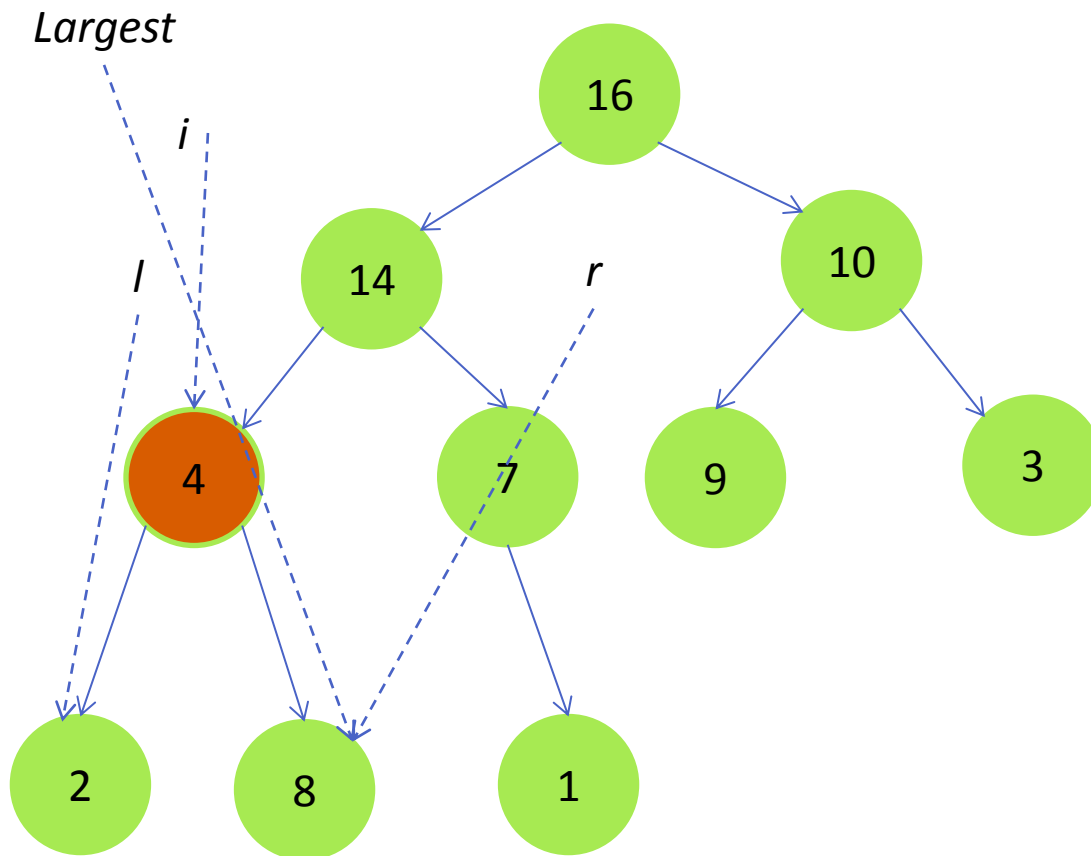

Jak funguje MAX-HEAPIFY



```

MAX-HEAPIFY(A, i) {
1.  l = LEFT(i);
2.  r = RIGHT(i);
3.  if (l ≤ HEAP-SIZE(A) &
4.     A[l] > A[i])
5.    then Largest = l;
6.  else Largest = i;
7.  if (r ≤ HEAP-SIZE(A) &
8.     A[r] > A[Largest])
9.    then Largest = r;
10. if (Largest ≠ i)
11. then {
12.   A[i] ↔ A[Largest];
13.   MAX-HEAPIFY(A, Largest)}
}
  
```

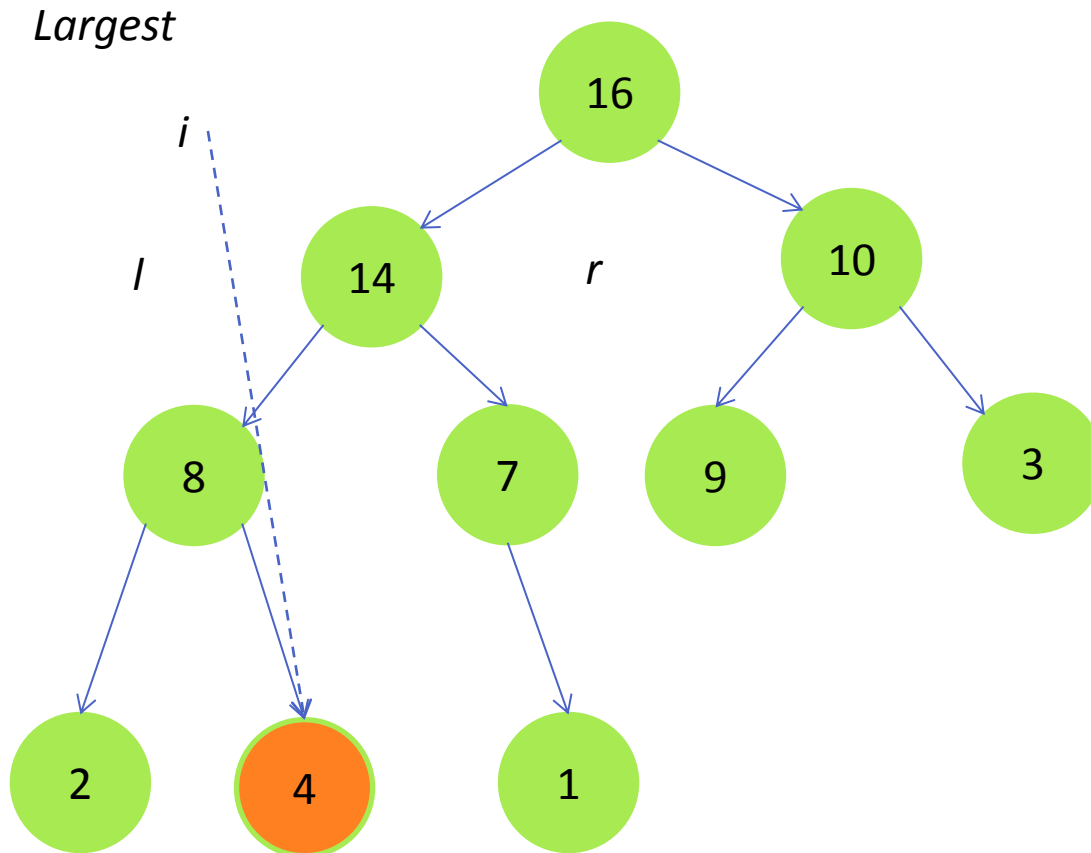
Jak funguje MAX-HEAPIFY



```

MAX-HEAPIFY(A, i) {
1.  l = LEFT(i);
2.  r = RIGHT(i);
3.  if (l ≤ HEAP-SIZE(A) &
4.     A[l] > A[i])
5.    then Largest = l;
6.  else Largest = i;
7.  if (r ≤ HEAP-SIZE(A) &
8.     A[r] > A[Largest])
9.    then Largest = r;
10. if (Largest ≠ i)
11.  then {
12.    A[i] ↔ A[Largest];
13.    MAX-HEAPIFY(A, Largest)}
}
  
```

Jak funguje MAX-HEAPIFY



```

MAX-HEAPIFY(A, i) {
1.  l = LEFT(i);
2.  r = RIGHT(i);
3.  if (l ≤ HEAP-SIZE(A) &
4.     A[l] > A[i])
5.    then Largest = l;
6.  else Largest = i;
7.  if (r ≤ HEAP-SIZE(A) &
8.     A[r] > A[Largest])
9.    then Largest = r;
10. if (Largest ≠ i)
11. then {
12.   A[i] ↔ A[Largest];
13.   MAX-HEAPIFY(A, Largest)}
}
  
```

Iterativní MAX-HEAPIFY

Iterativní algoritmus MAX-HEAPIFY (tvorba haldy iterací):

```
MAX-HEAPIFY(A, i) {  
1.  while ( $i \leq \lfloor \text{HEAP-SIZE}(A)/2 \rfloor$ ) {  
2.      l = LEFT(i);  
3.      r = RIGHT(i);  
4.      if ( $l \leq \text{HEAP-SIZE}(A) \ \& \ A[l] > A[i]$ )  
5.          then Largest = l else Largest = i;  
6.      if ( $r \leq \text{HEAP-SIZE}(A) \ \& \ A[r] > A[\textit{Largest}]$ )  
7.          then Largest = r;  
8.      if (Largest  $\neq$  i) return;  
9.       $A[i] \leftrightarrow A[\textit{Largest}]$ ; // výměna  
10.     i = Largest;  
11. }  
}
```

Řazení pomocí haldy (HeapSort)

Využijeme MAX-HEAPIFY k naplnění pole A

```
BUILD-MAX-HEAP( $A$ )
1   $A.heap-size = A.length$ 
2  for  $i = \lfloor A.length/2 \rfloor$  downto 1
3      MAX-HEAPIFY( $A, i$ )
```

Pak zkontrolujeme pomocí MAX-HEAPIFY všechny prvky A

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

Rekurzivní QUICK-SORT

Rekurzivní algoritmus řazení dělením

QUICK-SORT(A , low , $high$) {

1. **if** ($low < high$)

2. **then** { $pivot \leftarrow \text{SELECTPIVOT}(A, low, high);$

3. $mid \leftarrow \text{ROZDEL}(A, low, high, pivot);$

4. QUICK-SORT(A, low, mid);

5. QUICK-SORT($A, mid + 1, high$)

6. }

}

Méně rekurzivní QUICK-SORT

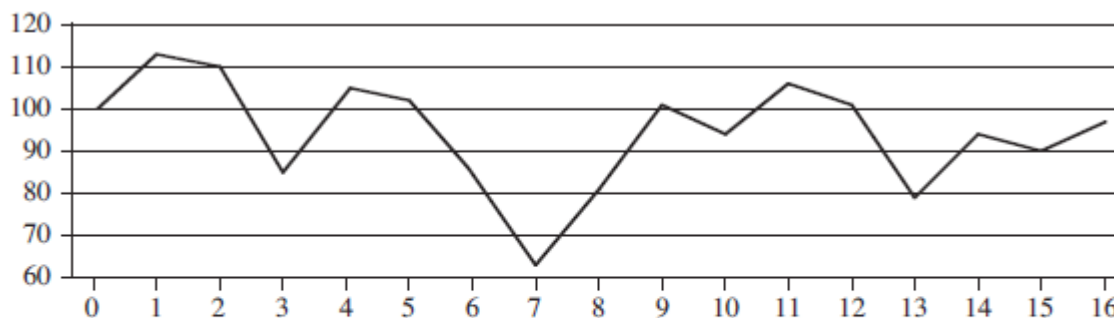
Méně rekurzivní algoritmus řazení dělením

```
QUICK-SORT(A, low, high) {
```

```
1.  while (low < high) {  
2.      pivot = SELECTPIVOT(A, low, high);  
3.      mid = ROZDEL(A, low, high, pivot);  
4.      QUICK-SORT(A, low, mid);  
5.      low = mid + 1  
6.  }  
}
```

Př.: Optimalizace nákupu a prodeje akcií

- Předpokládejme, že máme možnost investovat do akcií Volatile Chemical Corporation. Cena akcií této společnosti se hodně mění.
- Máme povoleno zakoupit jednu akcii za den, prodat ji můžeme nejdříve příští den. Pro kompenzaci tohoto omezení, můžeme zjistit cenu akcie v budoucnosti (viz graf a tabulka pro příštích 17 dnů):

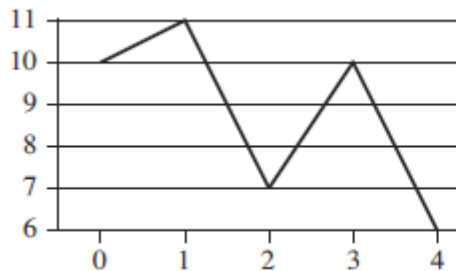


Day	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
Price	100	113	110	85	105	102	86	63	81	101	94	106	101	79	94	90	97
Change		13	-3	-25	20	-3	-16	-23	18	20	-7	12	-5	-22	15	-4	7

- Náš cíl je maximalizace zisku.

Př.: Pokračování

- Akcie můžeme zakoupit kdykoliv, počínaje dnem 0 (cena je \$100 za akcii).
- Samozřejmě se snažíme “buy low, sell high”, abychom maximalizovali zisk.
- Naneštěstí nelze zakoupit akcie za nejnižší cenu a poté prodat za cenu nejvyšší v rámci uvedeného intervalu. Podle tabulky je nejnižší cena v 7.den, což je po 1.dni, kdy byla cena nejvyšší.
- Zpočátku si můžeme myslet, že maximalizace zisku dosáhneme tak, že najdeme nevyšší a nejnižší cenu, poté najdeme nejnižší cenu před cenou nejvyšší a nejvyšší cenu po ceně nejnižší. Z těchto možností vybereme dvojici s největším rozdílem cen. To ale není pravda – viz obrázek:



Day	0	1	2	3	4
Price	10	11	7	10	6
Change		1	-4	3	-4

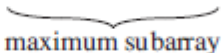
Naivní řešení hrubou silou

- Naivní řešení hrubou silou je založeno na následujícím postupu:
- Vyber všechny dvojice nákup-prodej takové, že datum nákupu předchází datum prodeje. Uvažujeme-li interval n dnů, pak takových dvojic může existovat $\binom{n}{2}$ kombinací, což odpovídá složitosti $\Theta(n^2)$.
- V nejlepším případě můžeme kontrolu jedné dvojice provést v konstantním čase, čemuž odpovídá celková složitost $\Omega(n^2)$.
- Lze to řešit lépe?

Řešení transformací na jiný problém

- Abychom se pokusili navrhnout lepší algoritmus, můžeme zkusit zpracovat vstup jiným způsobem. Můžeme se pokusit najít ve vstupu takovou posloupnost dvojic dnů, kde je maximální čistý rozdíl mezi počáteční a koncovou cenou. Místo, abychom uvažovali ceny v určitém dni, zabýváme se změnami v ceně za den – změna za den se určí jako rozdíl mezi cenou v době i a cenou v době $i+1$. Výsledkem je níže uvedené pole A :

	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
A	13	-3	-25	20	-3	-16	-23	18	20	-7	12	-5	-22	15	-4	7



maximum subarray

- V tomto poli se můžeme pokusit nalézt spojitý úsek, jehož součet hodnot je maximální. V našem příkladu je to úsek mezi indexy 8 a 11, kde součet hodnot je 43 (tzv. problém maximální podposloupnosti).
- Výsledek říká, že bychom měli nakupovat akcie těsně před dnem 8 a prodávat po dni 11, výnos bude \$43 za akcii. Složitost je ale stále $\Theta(n^2)$.

Řešení technikou rozděl a panuj

- Předpokládejme, že chceme najít maximální podposloupnost v poli $A[low...high]$. Technika rozděl a panuj říká, že bychom měli pole rozdělit na menší části – pro vyvážené dělení na dvě přibližně stejné části.
- Nalezneme tedy střed – index mid , a uvažujeme dvě podposloupnosti $A[low...mid]$ a $A[mid...high]$. Libovolný spojitý úsek $A[i...j]$ musí ležet v jedné z následujících částí:
 - celý v podposloupnosti $A[low...mid]$, takže $low \leq i \leq j \leq mid$,
 - celý v podposloupnosti $A[mid+1...high]$, takže $mid < i \leq j \leq high$,
 - nebo přetíná hranici zlomu, takže $low \leq i \leq mid < j \leq high$.
- V prvních dvou případech můžeme hledání maximální podposloupnosti v poli $A[low...high]$ převést na hledání maximální podposloupnosti v poli $A[low...mid]$ nebo $A[mid...high]$.
- V posledním případě se nejedná o zmenšenou instanci původního problému, neboť musíme najít obě části a kombinovat je → tím začneme.

Hledání podpole přetínajícího hranici

FIND-MAX-CROSSING-SUBARRAY(A , low , mid , $high$)

1. $left\text{-}sum = -\infty$; $sum = 0$;
2. **for** $i = mid$ **downto** low { // maximální podpole na konci levé části
3. $sum = sum + A[i]$;
4. **if** $sum > left\text{-}sum$ **then** { $left\text{-}sum = sum$; $max\text{-}left = i$ }
5. }
6. $right\text{-}sum = -\infty$; $sum = 0$;
7. **for** $j = mid + 1$ **to** $high$ { // maximální podpole na začátku pravé části
8. $sum = sum + A[j]$;
9. **if** $sum > right\text{-}sum$ **then** { $right\text{-}sum = sum$; $max\text{-}right = j$ }
10. }
11. **return** ($max\text{-}left$, $max\text{-}right$, $left\text{-}sum + right\text{-}sum$)

Složitost $\Theta(n)$, pro $n = high - low + 1$

Celkové řešení technikou rozděl a panuj

FIND-MAXIMUM-SUBARRAY(A , low , $high$)

Složitost $\Theta(n \cdot \log_2 n)$

1. **if** $high == low$
2. **return** (low , $high$, $A[low]$) // triviální případ: pouze jeden element
3. **else** $mid = \lfloor (low + high) / 2 \rfloor$;
4. $(left-low, left-high, left-sum) =$ FIND-MAXIMUM-SUBARRAY(A , low , mid);
5. $(right-low, right-high, right-sum) =$
6. FIND-MAXIMUM-SUBARRAY(A , $mid + 1$, $high$);
7. $(cross-low, cross-high, cross-sum) =$
8. FIND-MAX-CROSSING-SUBARRAY(A , low , mid , $high$);
9. **if** $left-sum \geq right-sum$ **and** $left-sum \geq cross-sum$
10. **return** ($left-low$, $left-high$, $left-sum$)
11. **elseif** $right-sum \geq left-sum$ **and** $right-sum \geq cross-sum$
12. **return** ($right-low$, $right-high$, $right-sum$)
13. **else** **return** ($cross-low$, $cross-high$, $cross-sum$)

Kdy brutální síla vítězí?

Existují případy, kdy brutální síla vítězí. Jinými slovy naivní algoritmus může být někdy úspěšným řešením. Jako příklad uvažme hru „[Master Mind](#)“. Brutální síla je zde efektivní, neboť řešení, které probíhá všechny možnosti a vyškrtává neplatné kombinace vede k cíli přímočaře a pravděpodobně nejrychleji. Sofistikované algoritmy pro procházení stavového prostoru možností tak úspěšné nebývají. Je to ale tím, že rozsah stavového prostoru je poměrně malý – variace K prvků třídy $N = N^K$. Pro $N=4$ a $K=6$ tedy 1296 možných variací. Maximální počet pokusů je ≤ 5 .

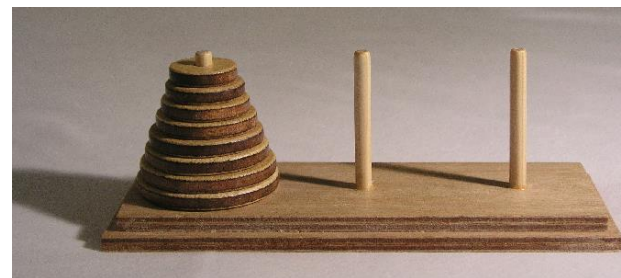


[Donald Knuth](#) (1977) navrhl algoritmus:

1. Vytvoř množinu S zbývajících možností (na začátku 1296). Prvý odhad bude aabb
2. Vyhod' z S všechny možnosti, které neodpovídají ohodnocení.
3. Pro všechny možné odhady (nemusí být v S) spočti, kolik budou eliminovat prvků z S – to je jeho skóre a vyber vždy odhad s nejmenším skóre.
4. Jako další odhad použij ten, který má největší skóre (minimax).
5. Pokud jsi neuhodl , jdi na krok 2.

Kdy naopak vítězí rozděl a panuj

- Problém Hanojských věží - přesun N disků z tyče A na tyč B pomocí tyče C.
- http://www.mathsisfun.com/games/hanoi_solver.html
- Jednoduché řešení pomocí techniky rozděl a panuj – pro přesun N disků z tyče A na tyč B pomocí tyče C postupuj následovně:
 - Přesuň N-1 disků z tyče A na tyč C.
 - Přesuň jeden disk z tyče A na tyč B.
 - Přesuň N-1 disků z tyče C na tyč B.
- [Řešení na YouTube](#)
- Jednoduché řešení založené na principu problému. Jiná sofistikovanější řešení nejsou tak úspěšná.



Vypráví se legenda, že někde ve Vietnamu nebo Indii stojí klášter nebo chrám, v němž jsou hanojské věže se 64 zlatými kotouči. Mniši (kněží) každý den v poledne za zvuku zvonů slavnostně přemístí jeden kotouč (v jiných verzích probíhá přemísťování nepřetržitě).

V okamžiku, kdy bude přemístěn poslední kotouč, nastane konec světa.

Vyřešení tohoto hlavolamu pro 64 kotoučů však vyžaduje $2 \uparrow 64 - 1 = 18\,446\,744\,073\,709\,551\,615$ tahů,

takže i kdyby mniši stihli provést jeden tah každou sekundu

(a postupovali nejkratším možným způsobem), trvalo by jim vyřešení celého hlavolamu přibližně 600 miliard let.

1.domácí úkol

Vaším úkolem bude naprogramovat třídu **Homework1** implementující rozhraní **MERGE-SORT** s následujícími metodami:

```
interface MergeSort {
```

1. `int[] getFirstHalfOf(int[] array);`
 2. `int[] getSecondHalfOf(int[] array);`
 3. `int[] merge(int[] firstHalf, int[] secondHalf);`
 4. `int[] mergesort(int[] array);`
- ```
}
```

# The End