Techniky návrhu algoritmů

Karel Richta a kol.

Katedra počítačů
Fakulta elektrotechnická
České vysoké učení technické v Praze

Datové struktury a algoritmy, B6B36DSA
02/2023, Lekce 2

https://cw.fel.cvut.cz/b222/courses/b6b36dsa/start

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.

Rozklad na podproblémy

- Složité problémy řešíme zpravidla rozkladem na problémy jednodušší – tzv. podproblémy. Podproblémy mohou být jednodušší instance stejného problému.
- Podle způsobu rozdělení na podproblémy lze při řešení problémů rozkladem na podproblémy rozlišovat případy, kdy se vyčleněné podproblémy nepřekrývají – pak zpravidla využíváme klasické rekurzivní řešení pomocí techniky **rozděl a panuj.**
- Pokud se podproblémy mohou překrývat, lze využít tzv. **dynamického programování**, kde můžeme částečná řešení uchovávat a případně později opakovaně využít.
- Podproblémy k řešení lze také vyřazovat, pokud nevedou k řešení či nejsou perspektivní – technika nazývaná **prořezávej a hledej**, příp. pokud se pak vracíme k původnímu problému nazývaná **prohledávání s návratem** (backtracking).

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)

- Návrh algoritmu technikou rozděl a panuj má 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 zatřiďováním (**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: *p…q* a *q+1…r*
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.
- Př.: Faktoriál

$$\textsc{Fac}(n) = 1 \qquad \text{if } n = 1,$$
$$\textsc{Fac}(n) = n * \textsc{Fac}(n-1) \qquad \text{if } n > 1.$$

- Koncová rekurze (tail recursion): rekurzivní volání je posledním příkazem algoritmu, po kterém se už neprovádějí žádné další "odložené" operace.
- Př.: Největší společný dělitel (Euclidův algoritmus): Nechť n ≥ m ≥ 0.

$$\mathrm{GCD}(n, m) = n \qquad \text{if } m = 0,$$
$$\mathrm{GCD}(n, m) = \mathrm{GCD}(m, \textsc{Remainder}(n, m)) \qquad \text{if } m > 0.$$

Typy rekurze (pokr.)

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

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

- Vnořená rekurze: Rekurzivní funkce, jejíž argumenty jsou specifikovány rekurzivně.
- Př.: Ackermannova funkce (není tzv. primitivně rekurzivní)

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

- 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) = Θ (1).
- Pro matice rozměru 2 x 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)
n = A.rows
let C be a new n × n matrix
for i = 1 to n
    for j = 1 to n
        c_ij = 0
        for k = 1 to n
            c_ij = c_ij + a_ik · b_kj
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).

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

```
SQUARE-MATRIX-MULTIPLY-RECURSIVE(A, B)
n = A.rows
let C be a new n × n matrix
if n == 1
    c11 = a11 · b11
else partition A, B, and C
    C11 = SQUARE-MATRIX-MULTIPLY-RECURSIVE(A11, B11)
            + SQUARE-MATRIX-MULTIPLY-RECURSIVE(A12, B21)
    C12 = SQUARE-MATRIX-MULTIPLY-RECURSIVE(A11, B12)
            + SQUARE-MATRIX-MULTIPLY-RECURSIVE(A12, B22)
    C21 = SQUARE-MATRIX-MULTIPLY-RECURSIVE(A21, B11)
            + SQUARE-MATRIX-MULTIPLY-RECURSIVE(A22, B21)
    C22 = SQUARE-MATRIX-MULTIPLY-RECURSIVE(A21, B12)
            + SQUARE-MATRIX-MULTIPLY-RECURSIVE(A22, B22)
return C
```

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).

$$C_{11} = A_{11}B_{11} + A_{12}B_{21}$$
$$C_{12} = A_{11}B_{12} + A_{12}B_{22}$$
$$C_{21} = A_{21}B_{11} + A_{22}B_{21}$$
$$C_{22} = A_{21}B_{12} + A_{22}B_{22}$$

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

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

$$T(1) = \Theta(1)$$
$$T(n) = 7T(n/2) + \Theta(n^2)$$
$$T(n) = \Theta(n^{\log_2 7})$$

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 Θ(n), neboť v nejhorším případě musím projít úsek celý a porovnat všechny prvky. Zkusme najít řešení efektivnější.

Rekurzivní verze BINARY-SEARCH

$\textsc{Recursive-Binary-Search}(A, v, low, high)$

if $low > high$
  return NIL
$mid = \lfloor (low + high)/2 \rfloor$
if $v == A[mid]$
  return mid
elseif $v > A[mid]$
  return $\textsc{Recursive-Binary-Search}(A, v, mid + 1, high)$
else return $\textsc{Recursive-Binary-Search}(A, v, low, mid - 1)$

- 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

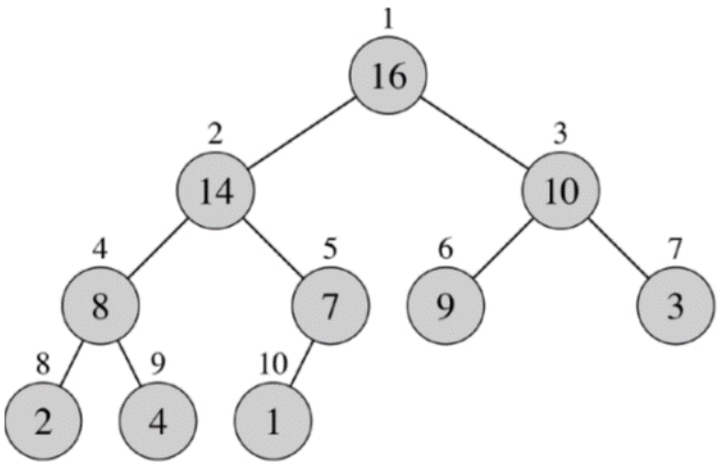
ITERATIVE-BINARY-SEARCH$(A, v, low, high)$

```
while low ≤ high
    mid = ⌊(low + high)/2⌋
    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

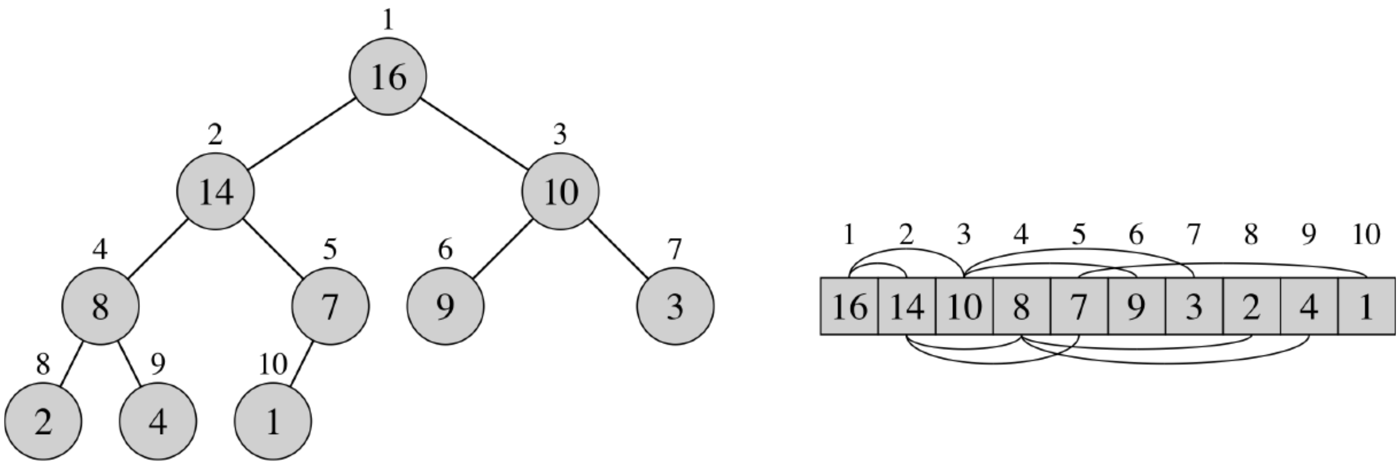
- (Binární) halda (heap) je téměř úplný (binární) strom. Neúplná může být pouze poslední vrstva listů a to ještě jen zprava:



- 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.:



(a) (b)

- 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.:

  PARENT(i) = $\lfloor i/2 \rfloor$

  LEFT(i) = $2*i$

  RIGHT(i) = $2*i + 1$ (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):

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

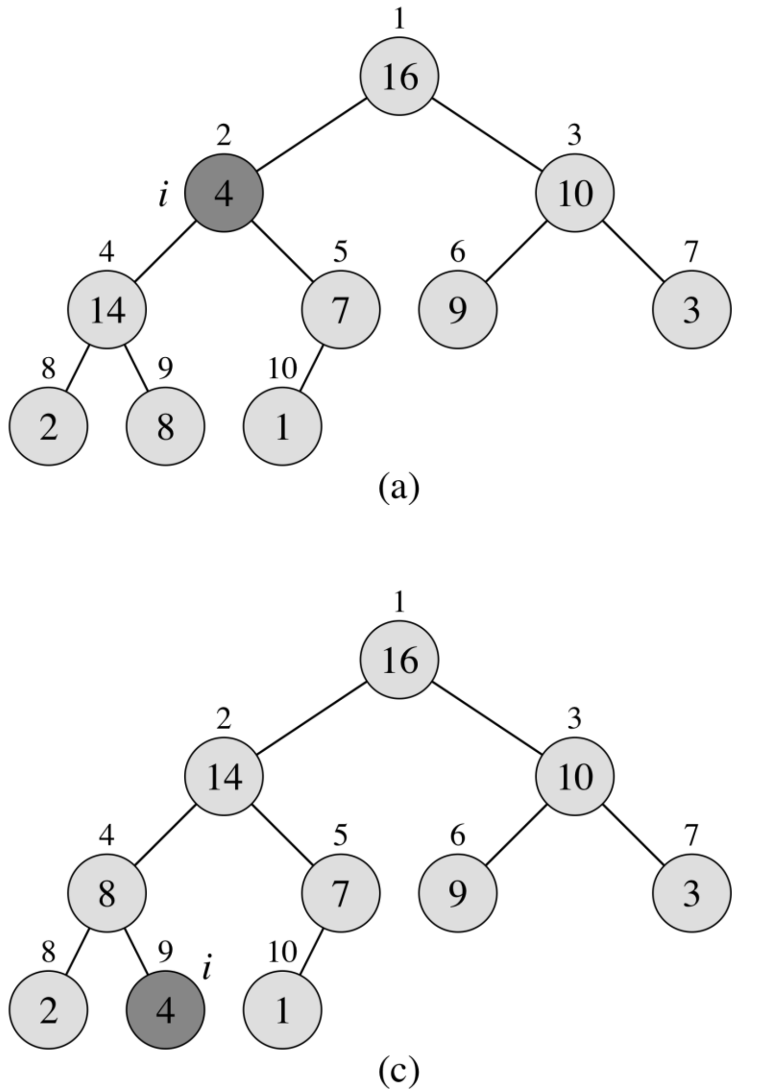
- Indukcí a na základě transitivity MAX-HEAP lze ukázat, že tato vlastnost zaručuje, že maximální element je kořen haldy. Tuto vlastnost haldy využijeme později při návrhu algoritmu HEAPSORT.
- Nejprve navrhneme proceduru MAX-HEAPIFY(*A,i*), která bude pro daný prvek zjišťovat, zda není menší než jeho potomci (splňuje vlastnost MAX-HEAP). Pokud ji nesplňuje, 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

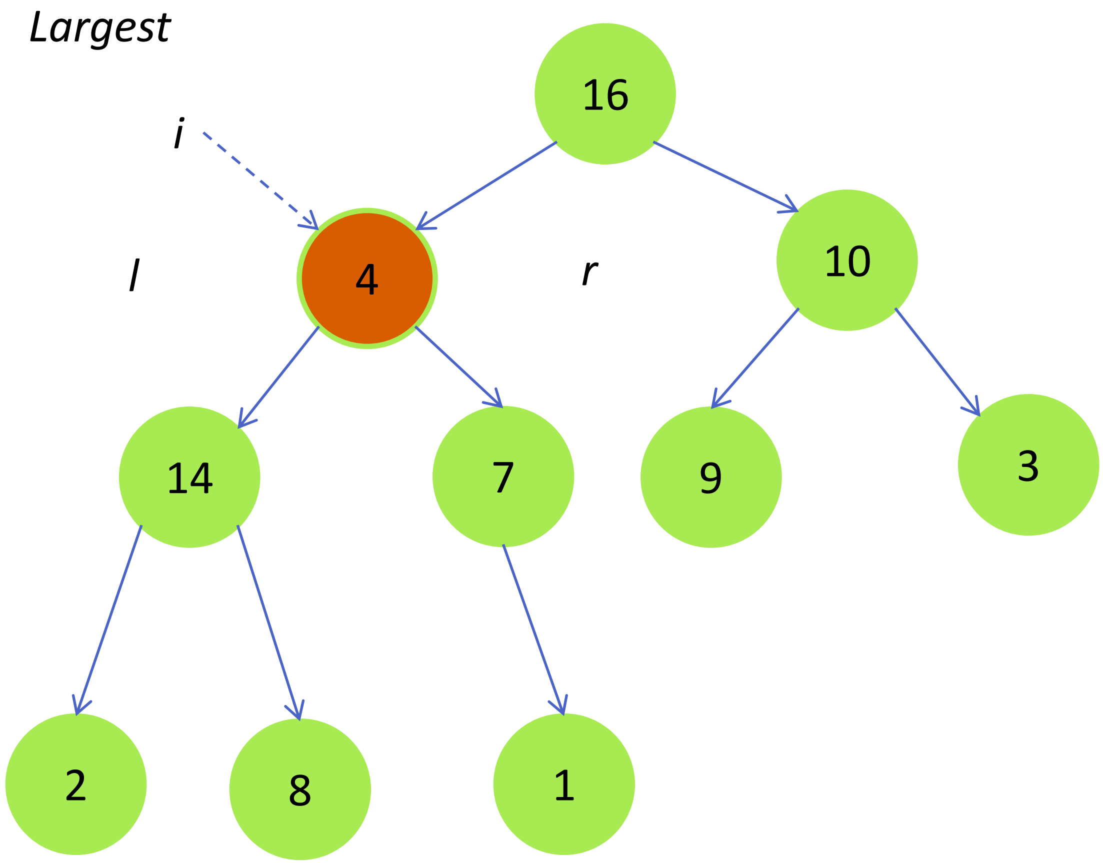
Rekurzivní algoritmus Max-Heapify (tvorba haldy rekurzí):

```
Max-Heapify(A, i) {
    l = Left(i);
    r = Right(i);
    if (l ≤ Heap-Size (A) & A[l] > A[i])
        then Largest = l;
        else Largest = i;
    if (r ≤ Heap-Size(A) & A[r] > A[Largest])
        then Largest = r;
    if (Largest ≠ i)
        then {A[i] ↔ A[Largest] ; Max-Heapify(A, Largest)}
}
```

Jak to funguje

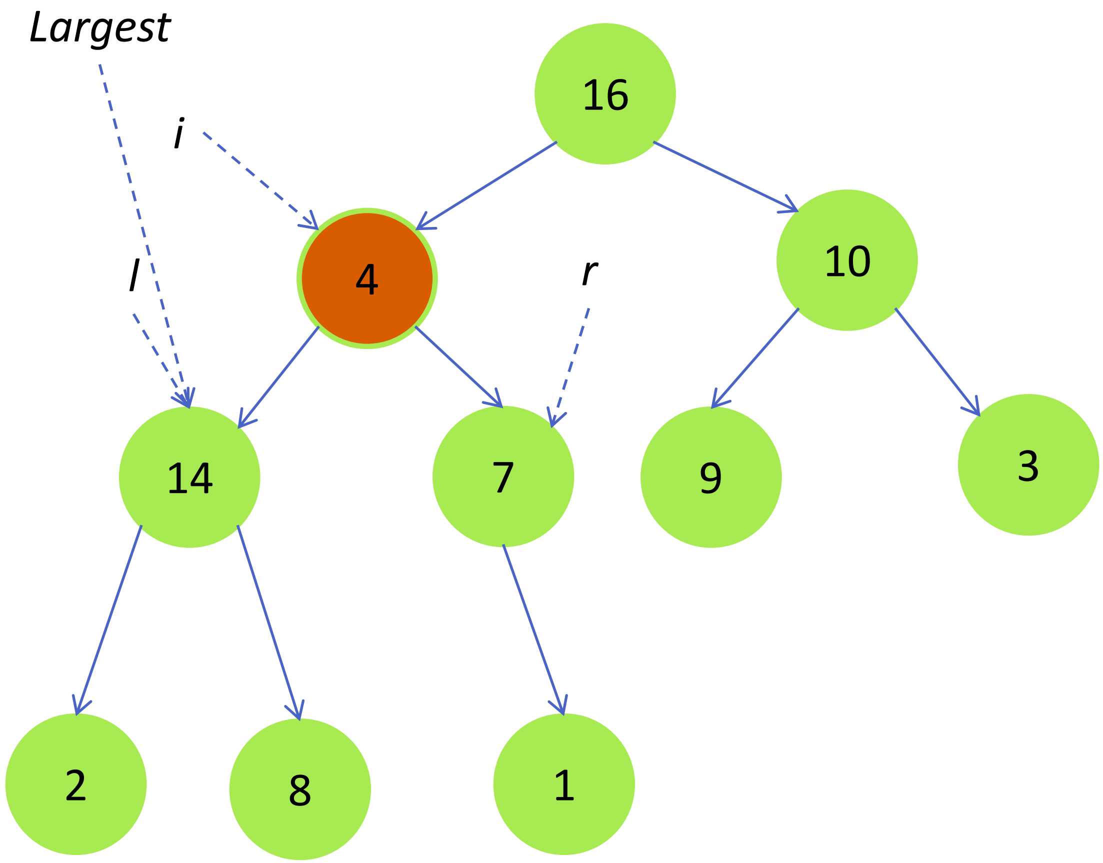


Jak funguje MAX-HEAPIFY



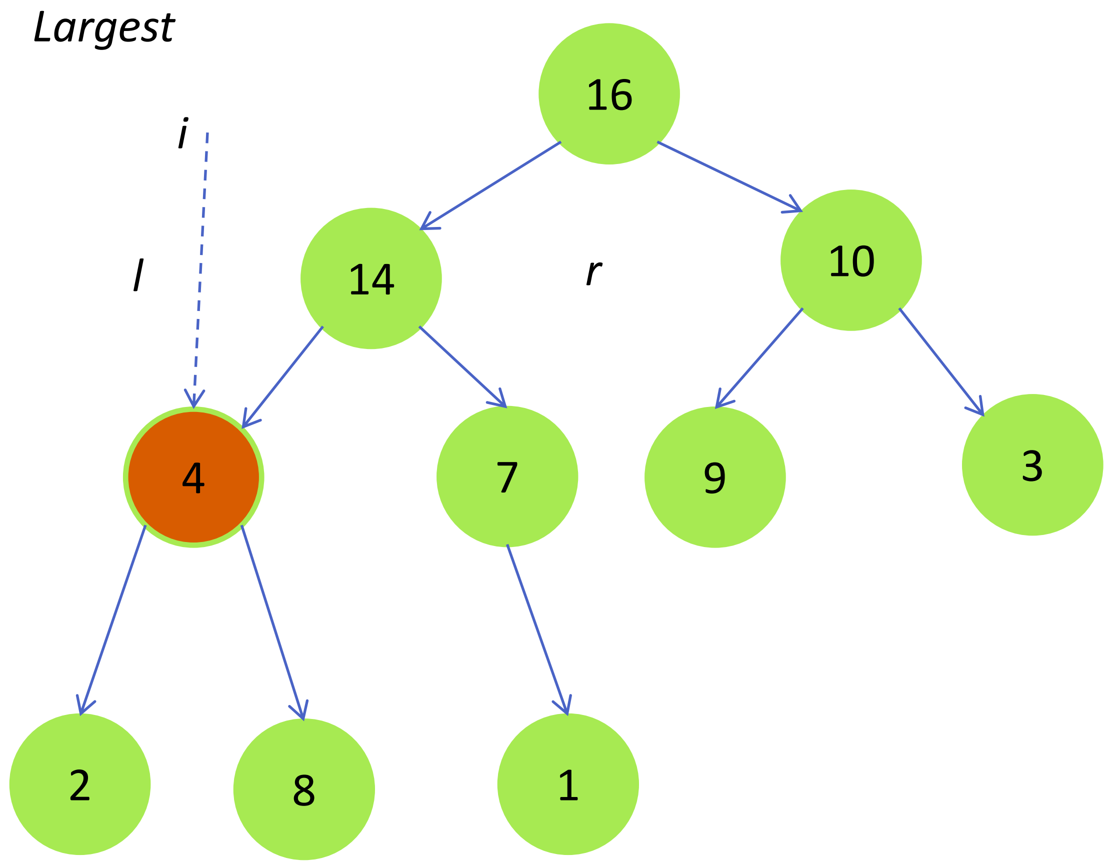

```
MAX-HEAPIFY(A, i) {
  l = LEFT(i);
  r = RIGHT(i);
  if (l ≤ HEAP-SIZE (A) &
       A[l] > A[i])
     then Largest = l;
     else Largest = i;
  if (r ≤ HEAP-SIZE(A) &
       A[r] > A[Largest])
     then Largest = r;
  if (Largest ≠ i)
     then {
       A[i] ↔ A[Largest] ;
       MAX-HEAPIFY(A, Largest)}
}
```

Jak funguje MAX-HEAPIFY



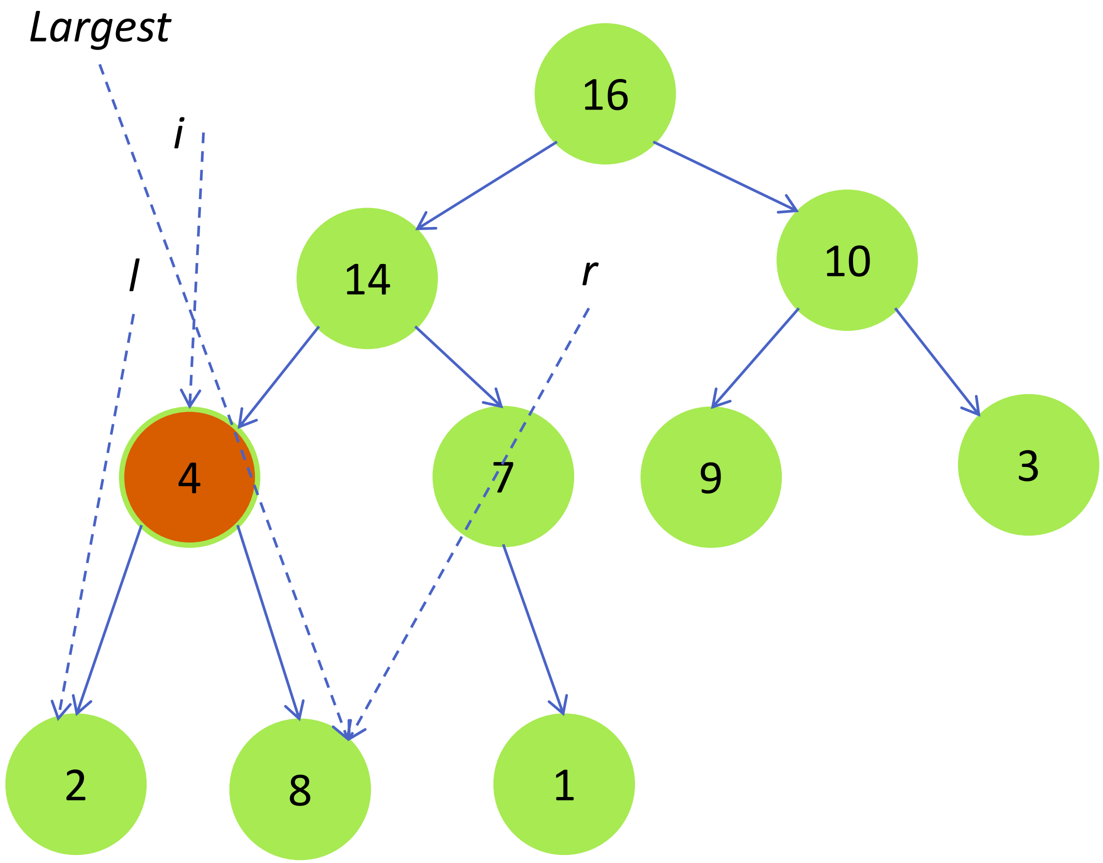

```
MAX-HEAPIFY(A, i) {
l = LEFT(i);
r = RIGHT(i);
if (l ≤ HEAP-SIZE (A) &
      A[l] > A[i])
   then Largest = l;
   else Largest = i;
if (r ≤ HEAP-SIZE(A) &
      A[r] > A[Largest])
   then Largest = r;
if (Largest ≠ i)
   then {
      A[i] ↔ A[Largest] ;
      MAX-HEAPIFY(A, Largest)}
}
```

Jak funguje MAX-HEAPIFY



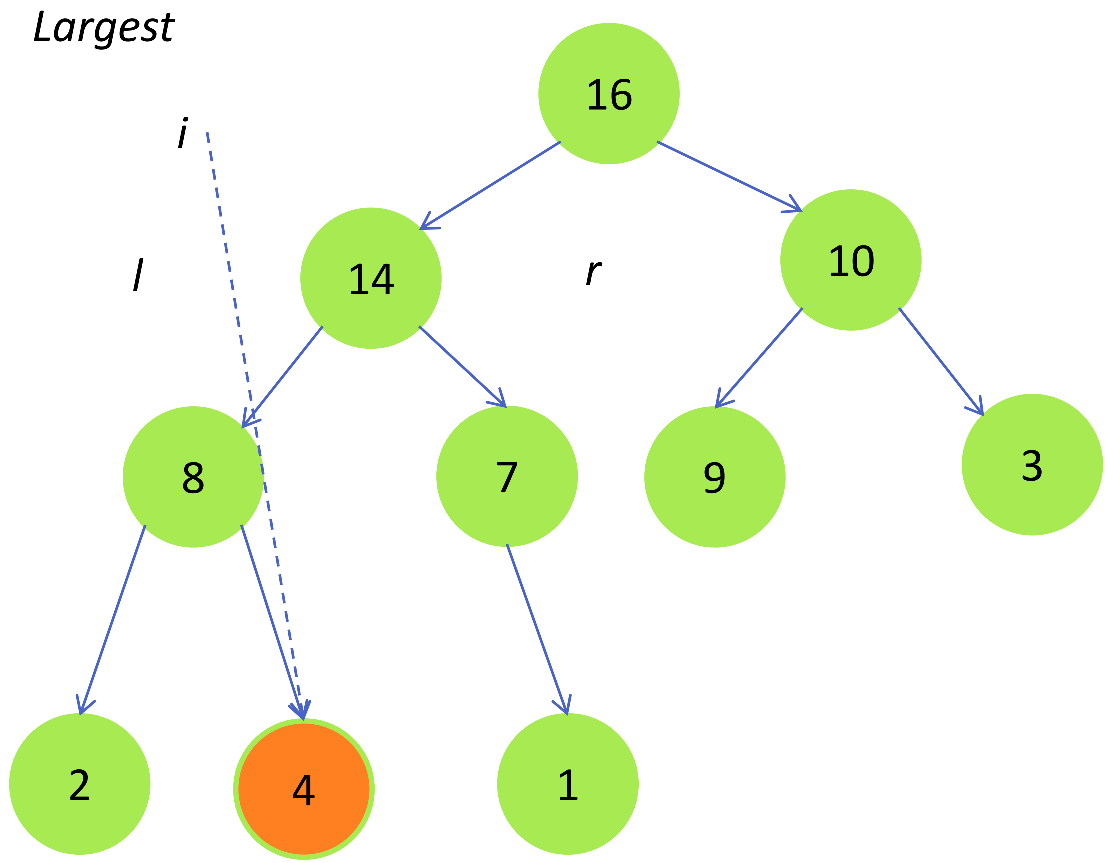

```
MAX-HEAPIFY(A, i) {
   l = LEFT(i);
   r = RIGHT(i);
   if (l ≤ HEAP-SIZE (A) &
        A[l] > A[i])
      then Largest = l;
      else Largest = i;
   if (r ≤ HEAP-SIZE(A) &
        A[r] > A[Largest])
      then Largest = r;
   if (Largest ≠ i)
      then {
        A[i] ↔ A[Largest] ;
        MAX-HEAPIFY(A, Largest)}
}
```

Jak funguje MAX-HEAPIFY




```
MAX-HEAPIFY(A, i) {
  l = LEFT(i);
  r = RIGHT(i);
  if (l ≤ HEAP-SIZE (A) &
       A[l] > A[i])
     then Largest = l;
     else Largest = i;
  if (r ≤ HEAP-SIZE(A) &
       A[r] > A[Largest])
     then Largest = r;
  if (Largest ≠ i)
     then {
        A[i] ↔ A[Largest] ;
        MAX-HEAPIFY(A, Largest)}
}
```

Jak funguje MAX-HEAPIFY




```
MAX-HEAPIFY(A, i) {
    l = LEFT(i);
    r = RIGHT(i);
    if (l ≤ HEAP-SIZE (A) &
          A[l] > A[i])
       then Largest = l;
       else Largest = i;
    if (r ≤ HEAP-SIZE(A) &
          A[r] > A[Largest])
       then Largest = r;
    if (Largest ≠ i)
       then {
         A[i] ↔ A[Largest] ;
         MAX-HEAPIFY(A, Largest)}
}
```

Iterativní MAX-HEAPIFY

Iterativní algoritmus MAX-HEAPIFY (tvorba haldy iterací):

```
MAX-HEAPIFY(A, i) {
    while (i ≤ ⌊ HEAP-SIZE(A)/2 ⌋) {
        l = LEFT(i);
        r = RIGHT(i);
        if (l ≤ HEAP-SIZE(A) & A[l] > A[i])
            then Largest = l else Largest = i;
        if (r ≤ HEAP-SIZE(A) & A[r] > A[Largest])
            then Largest = r;
        if (Largest ≠ i) return;
        A[i] ↔ A[Largest]; // výměna
        i = Largest;
    }
}
```

Řazení pomocí haldy (HeapSort)

Využijeme MAX-HEAPIFY k naplnění pole *A*

```
BUILD-MAX-HEAP(A)
A.heap-size = A.length
for i = ⌊A.length/2⌋ downto 1
    MAX-HEAPIFY(A, i)
```

Pak zkontrolujeme pomocí MAX-HEAPIFY všechny prvky *A*

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

Rekurzivní QUICK-SORT

Rekurzivní algoritmus řazení dělením

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

1. **if** (*low* < *high*)
2. **then** { *pivot* ← SELECTPIVOT(*A*, *low*, *high*);
3. *mid* ← ROZDEL(*A*, *low*, *high*, *pivot*);
4. QUICK-SORT(*A*, *low*, *mid*);
5. QUICK-SORT(*A*, *mid* + 1, *high*)
6. }

}

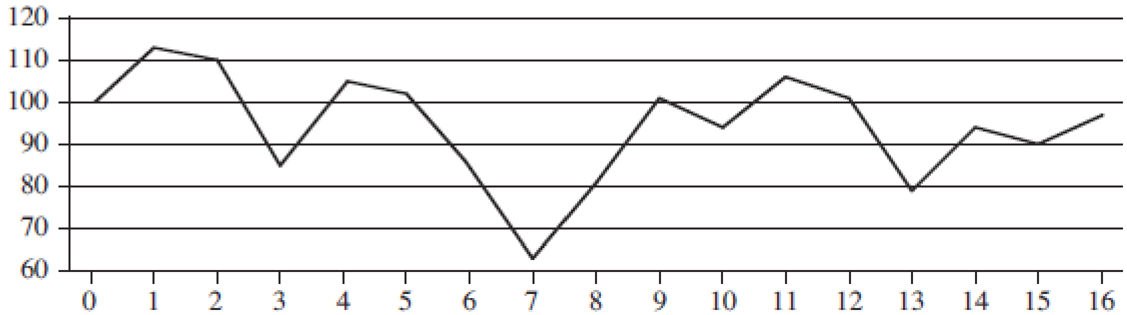
Méně rekurzivní QUICK-SORT

Méně rekurzivní algoritmus řazení dělením

```
QUICK-SORT(A, low, high) {
    while (low < high) {
        pivot = SELECTPIVOT(A, low, high);
        mid = ROZDEL(A, low, high, pivot);
        QUICK-SORT(A, low, mid);
        low = mid + 1
    }
}
```

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):

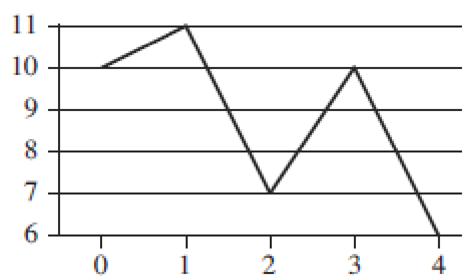


Day	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
Price	100	113	110	85	105	102	86	63	81	101	94	106	101	79	94	90	97
Change		13	−3	−25	20	−3	−16	−23	18	20	−7	12	−5	−22	15	−4	7

- 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 nejvyšší 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 prvých 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 $\rightarrow$ tím začneme.

Hledání podpole přetínajícího hranici

```
FIND-MAX-CROSSING-SUBARRAY(A, low, mid, high)
left-sum = - ∞; sum = 0;
for i = mid downto low { // maximální podpole na konci levé části
        sum = sum + A[i];
        if sum > left-sum then { left-sum = sum; max-left = i }
}
right-sum = - ∞; sum = 0;
for j = mid + 1 to high {// maximální podpole na začátku pravé části
         sum = sum + A[j];
        if sum > right-sum then { right-sum = sum; max-right = j }
}
return (max-left, max-right, left-sum + right-sum)
```

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

Celkové řešení technikou rozděl a panuj

Složitost $\Theta(n * log_2 n)$

```
FIND-MAXIMUM-SUBARRAY(A, low, high)
if high == low
    return (low, high, A[low]) // triviální případ: pouze jeden element
else mid = ⌊(low + high)/2⌋;
    (left-low, left-high, left-sum) = FIND-MAXIMUM-SUBARRAY(A, low, mid);
    (right-low, right-high, right-sum) =
        FIND-MAXIMUM-SUBARRAY(A, mid + 1, high);
    (cross-low, cross-high, cross-sum) =
        FIND-MAX-CROSSING- SUBARRAY(A, low, mid, high);
    if left-sum ≥ right-sum and left-sum ≥ cross-sum
        return (left-low, left-high, left-sum)
    elseif right-sum ≥ left-sum and right-sum ≥ cross-sum
        return (right-low, right-high, right-sum)
    else    return (cross-low, cross-high, cross-sum)
```

Techniky založené na stavech

- Problém modelujeme pomocí stavového prostoru
- Stav zde představuje souhrn všech důležitých aktuálních hodnot
- Vstupní data: počáteční stav
- Výstupní data: koncový stav(y)
- Stavy modelujeme jako uzly grafu, změny stavu jako přechody mezi uzly reprezentující nějakou akci – modelujeme je jako hrany grafu (orientované, příp. neorientované)
- Řešení problému vyjádříme jako algoritmus, který nalezne cestu ve stavovém prostoru vedoucí z počátečního stavu do stavu koncového
- Základní typy: prohledávání do šířky a prohledávání do hloubky
- Tyto techniky budeme podrobněji probírat, až budeme probírat grafy a další vhodné datové struktury
- Příklad: 3 misionáři, 3 kanibalové, řeka a loďka pro 2 osoby. Jak přepravit všechny na druhou stranu?

Další technika: Prořezávej a Hledej

- Tento postup optimalizace hledání je založen na postupném vyřazování části prohledávaných dat a tím redukováním složitosti hledání.
- Toto paradigma je velmi podobné algoritmům typu rozděl a panuj (divide and conquer), zásadní rozdíl je ovšem v tom, že při prořezávání neprocházíme všechny větve, ale pouze ty, které pro nás dávají smysl.
- V dynamickém programování se snažíme neopakovat výpočty, tady se snažíme je nedělat vůbec.

Příklad na Prořezávej a Hledej

- Hledáme i-té nejnižší číslo v neseřazeném seznamu čísel. Jednoduchý postup je seřadit seznam a vybrat i-tý prvek - tento postup však není nejefektivnější.
- Využijeme dělící funkce použité v algoritmu QUICKSORT, která v daném poli vybere pivota a prvky menší než pivot přehází do levé části pole a prvky větší do pravé části. Z pozice pivota v tomto poli můžeme určit, zda hledat i-té nejnižší číslo v levé nebo pravé části - v té rekurzivně opakujeme postup, druhou části se pak dále nemusíme zabývat.
- Příklad operuje nad globálním polem array a využívá následující funkci: **`int RSPLIT(l,r)`** - v části pole určené indexy **`l`** a **`r`** vybere pivota a prohází prvky pole tak, aby prvky menší než pivot byly nalevo a větší napravo. Návratová hodnota je index pivota.

Pseudokód pro Prořezávej a Hledej

```
int[] array; //
int RSELECT(int l, r, i) {
   // Vrátí index i-tého nejmenšího prvku
   int q = RSPLIT(l, r); // q je pivot
   int m = q - l + 1;    // m je počet prvků před pivotem
   if i < m then return RSELECT(l, q - 1, i);
        // Vlevo je vice prvků než i - tedy i-tý nejmenší musí ležet tam
   elsif i = m then return q;
        // Vlevo je právě i prvků - pivot je i-tý nejmenší
   else return RSELECT(q + 1, r, i - m);
        /* Vlevo je méně prvků než i - i-tý nejmenší musí ležet
           v pravé podčásti. i je zmenšeno o počet menších prvků,
           které jsme již našli (nalevo od pivota) */
   endif;
}
```

Jiná možnost: Zametací technika

- Je to postup převzatý z výpočetní geometrie, kdy postupujeme mezi objekty zleva doprava (zpravidla x souřadnice) a postupně zpracováváme místa (vyjádřená souřadnicí po které postupujeme), která nás zajímají.
  - Pro představu se používá idea svislé zametací přímky, která představuje onu hranici posouvající se zleva doprava.
- Základní charakterizující body jsou:
  - Suneme svislou přímku (scanline, SL, zametací přímku) zleva doprava nad množinou objektů.
  - Přímku posouváme na souřadnice, které nás zajímají (např. mezi body na grafu - nikoliv po x++).
  - Souřadnice, na které se chceme dostat, jsou v prioritní frontě (tu nazýváme postupový plán, x-struktura).
  - O již projitých místech/souřadnicích si ukládáme informace (nazývá se y-struktura).

Zametací technika

- Prostorem P, na němž probíhá řešení, proložíme zametací nadrovinu r (přímku v 2D, rovinu v 3D) a tuto nadrovinu posouváme mezi dvěma mezními polohami prostoru P.
- Spolu s nadrovinou udržujeme datovou strukturu S (y-struktura). Nadrovina rozděluje prostor řešení na „levý“ poloprostor, v němž již úloha byla vyřešena a „pravý“ poloprostor, v němž se řešení bude hledat.
- Struktura S obsahuje všechny informace o stavu řešení úlohy v levém podprostoru, které jsou relevantní pro řešení v pravém podprostoru. Stav struktury S se mění s polohou zametací nadroviny. Nadrovina se posouvá do pozic, kde se mění stavy řešení. Tyto pozice se nazývají postupový plán (x-struktura).

Zametací technika - pseudokód

Označme stav zametací přímky **y-struktura**. Zametací přímku budeme posouvat řešeným prostorem zleva doprava podle postupového plánu, který označíme **x-struktura**. Zametací techniku můžeme vyjádřit pseudokódem:

1: Inicializace x-struktury a y-struktury;

2: **while** x-struktura $\neq \emptyset$ **do**

begin p:=minimální prvek z x-struktury;

Posuň zametací přímku do p;

Vyjmi minimální prvek z x-struktury;

Uprav y-strukturu;

Generuj nové řešení odpovídající stavu y-struktury;

Uprav x-strukturu;

end;

Zametací technika - Příklad

- Hledám minimální body (takové, kde v levém dolním segmentu nejsou žádné body):
  1. Seřadím body dle x a uložím do x-struktury.
  2. Pamatuji si v y-struktuře jednu hodnotu - minimální (nejspodnější) y souřadnici. (Prvotní hodnota je nekonečno).
  3. Zametací přímku umisťuji do bodů x-struktury (tedy procházím zleva doprava body a v každém provedu aktualizaci y-struktury).
  4. Aktualizuji y-strukturu - pokud je aktuální souřadnice menší než hodnota v y-struktuře, nahradím.

Př.: Algoritmus pro Konvexní Obal

1. Setřídíme body podle x-ové souřadnice, označíme je $b_1,\ldots,b_n$.
2. Vložíme do horní a dolní obálky bod b_1: $H \leftarrow D \leftarrow (b_1)$.
3. Pro každý další bod $b = b_2,\ldots,b_n$:

   Přepočítáme horní obálku:

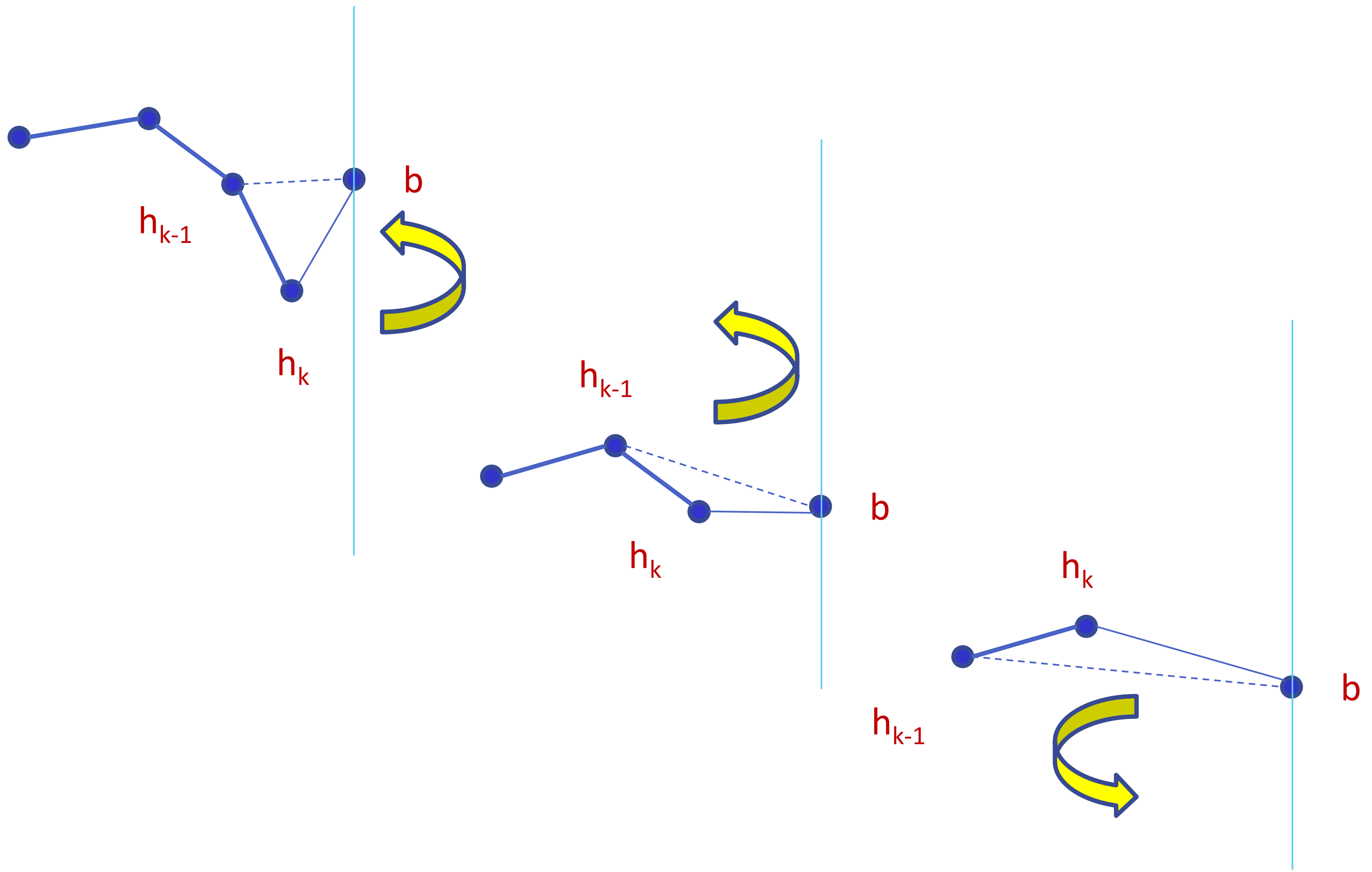
   a) Dokud $|H| \geq 2$, $H = (\ldots,h_{k-1},h_k)$ a úhel $h_{k-1}h_kb$ je orientovaný doleva:

      Odebereme poslední bod h_k z obálky H.

   b) Přidáme bod b na konec obálky H.

   Symetricky přepočteme dolní obálku (s orientací doprava).
4. Výsledný obal je tvořen body v obálkách H a D.

Složitost: $\Theta(n)$


b
h_{k-1}
h_k
h_{k-1}
b
h_k
h_k
b
h_{k-1}

Kdy brutální síla vítězí?

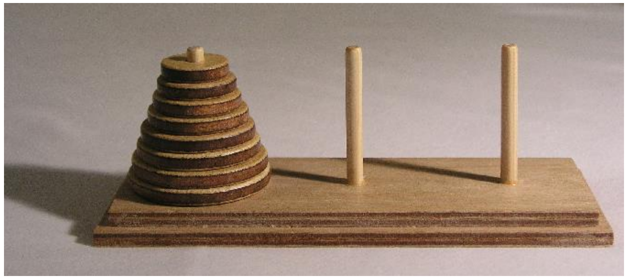
Existují případy, kdy brutální síla vítězí. Jinými slovy naivní algoritmus může být někdy úspěšným řešením. Jako příklad uvažme hru „Master Mind“. Brutální síla je zde efektivní, neboť řešení, které probíhá všechny možnosti a vyškrtává neplatné kombinace vede k cíli přímočaře a pravděpodobně nejrychleji. 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. Vyhoď 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 přemisťování nepřetržitě).
V okamžiku, kdy bude přemístěn poslední kotouč, nastane konec světa.
Vyřešení tohoto hlavolamu pro 64 kotoučů však vyžaduje 2↑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.

The End