Vyhľadávanie

Vyhľadávanie

Vstupy:

- N prvkov identifikovateľných kľúčom
- kľúč K, ktorý charakterizuje prvok, ktorý chceme nájsť

Výstupy:

- Úspech : prvok sa v zadanej množine našiel
- Neúspech: prvok sa nenašiel

Rozdelenie vyhľadávaní

- vnútorné, vonkajšie
- statické, dynamické
- usporiadaná, neusporiadaná postupnosť
- lineárne, binárne, pomocou rozptylovej funkcie

Jednoduché vyhľadávanie

Vstup: pole prvkov, kľúč
Výstup: pozícia nájdeného prvku v poli (ak sa prvok nenájde, tak sa vráti 0)

```
Procedure SEARCH(POST:pole, K: kľúč, var KDE: integer )
VAR NAJDENY : BOOLEAN;
    I  : 1..VELKOST;
BEGIN
   NAJDENY := FALSE;
   I := 1;
   WHILE((I <= VELKOST) AND (NOT NAJDENY)) DO
        IF K = POST[I].KLUC THEN
            BEGIN
                KDE := I;
                NAJDENY := TRUE;
            END
        ELSE
            I ++;
    IF NOT NAJDENY THEN
        KDE := 0;
END;
```


Jednoduché vyhľadávanie - lineárne

```
// vracia true akk existuje integer I také,
// že arr[i] == target a 0 <= i < arr.length

 boolean linearSearch(int[] arr, int target)
 {
        int i = 0;
        while (i < arr.length) {
          if (arr[i] == target) {
                return true;
          } // if
          ++i;
        }
        return false;
 }
```


Jednoduché vyhľadávanie - lineárne

```
type IntList is array(1..Max) of Integer;

function Search(List: IntList; Key: Integer)
return Integer
begin
   for I in List.Range loop
        if List(I) = Key then
                return I;
        end if;
   end loop;
return 0;
end Search;
```

popodmienka:
(List (*I*) = *Key*) $\vee$
((*I* = 0) $\wedge$ $\forall$*J*(1 $\leq$ *J* $\leq$ *Max* | (*List* (*J*) $\neq$ *Key*))

Jednoduché vyhľadávanie – lineárne so zarážkou

```
type IntList is array(0..Max) of Integer;

function Search(List: IntList; Key: Integer)
return Integer
I: Integer := Max;
begin
   List(0) := Key;
   while List(I) ≠ Key loop
            I := I - 1;
   end loop;
return I;
end Search;
```

invariant cyklu:
$\forall J(I < J \leq Max \mid List\ (J) \neq Key)$

Binárne vyhľadávanie

- Vstupom je usporiadané pole.
- Algoritmus porovná prvok nachádzajúci sa v strede poľa so zadaným kľúčom. Ak sa zhoduje, vráti jeho pozíciu. Ak sa nezhoduje, rozdelí pole na 2 polovice a pokračuje rovnakým hľadaním v tej polovici v ktorej sa prvok nachádza.

- Rekurzívna verzia algoritmu:

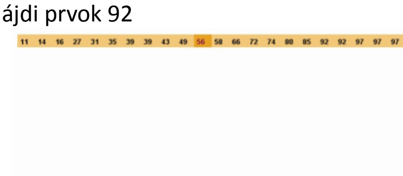
```
BinarySearch(A[0..N-1], value, low, high)
{
   if (high < low)
     return not_found
   mid = (low + high) / 2
   if (A[mid] > value)
     return BinarySearch(A, value, low, mid-1)
   else if (A[mid] < value)
     return BinarySearch(A, value, mid+1, high)
   else
     return mid
}
```


Binárne vyhľadávanie s cyklom

```
function Search(List: IntList; Key: Integer)
return Integer
          Low: Integer := List'First;
          High: Integer := List'Last;
          Mid: Integer;
          begin
          loop
                    Mid := (Low + High) / 2;
                    if Key = A[Mid] then return Mid;
                              elsif Key < A[Mid] then High := Mid - 1;
                              else Low := Mid + 1;
                    end if;
                    if Low > High then return 0;
          end loop;
end Search;
```


Binárne vyhľadávanie

- Nájdi prvok 92



Usporadúvanie

Usporadúvanie

- Postupnosť: a_1, a_2, a_3...a_n
- Položka:
  - a_i
  - Kľúč k_i položky a_i $k(a_i)$

Usporiadanie: binárna relácia

Úlohou je preusporiadať všetky položky postupnosti tak, aby platilo:

K1 <= K2 <= K3 <= ... <= Kn

Usporadúvanie

- Nech K je nekonečná lineárne usporiadaná množina (ďalej skratka LUM), t.j.
- množina, na ktorej je definovaná relácia < taká, že sú splnené tieto 2 podmienky:
  - zákon trichotómie
    - Pre ľubovoľné dva prvky K1, K2 ∈ K platí práve jedna z relácií : K1 < K2, K1 = K2, K1 > K2.
  - tranzitívny zákon
    - Pre ľubovoľné tri prvky K1, K2, K3 ∈ K platí : Ak K1 < K2 a K2 < K3, tak K1 < K3.
- Pre ľubovoľné K1, K2 ∈ K budeme písať, že K1 ≤ K2 akk K1 < K2 alebo K1 = K2.

Usporadúvanie

- Nech D je nekonečná množina a nech je na nej definovaná funkcia k : D → K.
- Definícia (problém usporadúvania).
  Nech je daná postupnosť $a_1, \ldots, a_n$, kde n ∈ N; a_i ∈ D. Potom všeobecný problém usporadúvania tkvie v určení takej permutácie π čísel 1,...,n, že platí $k(a_{\pi(1)}) \leq k(a_{\pi(2)}) \leq \ldots \leq k(a_{\pi(n)})$
- D – množina údajov
- k – funkcia usporiadania, daná zvyčajne explicitne pre každý prvok ako jeho zložka tzv. kľúč.
- Definícia. Permutácia n-prvkovej množiny je ľubovoľné usporiadanie prvkov tejto množiny do postupnosti.

Stabilný algoritmus

- Usporadúvací algoritmus je stabilný, ak vždy zachová originálne poradie elementov s rovnakými kľúčmi
- Ak elementy s rovnakými kľúčmi sú neodlíšiteľné, tak nie je potrebné sa zaoberať stabilitou algoritmu (napr. ak kľúčom je samotný element)
- Zachovať originálne poradie elementov je dôležité napr. pri viacnásobnom usporiadaní – najprv podľa priezviska a potom podľa mena.

Stabilný algoritmus

- Každý nestabilný algoritmus sa dá implementovať ako stabilný tým, že sa zapamätá originálne poradie elementov a pri zhodných kľúčoch sa berie do úvahy toto poradie
- Viacnásobné usporiadanie je možné obísť vytvorením jedného kľúča, ktorý je zložený z primárneho, sekundárneho, atď. kľúča usporiadania
  - Takéto úpravy nestabilných algoritmov majú negatívny vplyv na výpočtovú zložitosť.

Stabilný algoritmus

- Príklad – dvojice (kľúč, element):
    - (4, 5) (2, 7) (2, 3) (5, 6)
- Dve možné usporiadania:
    - **(2, 7) (2, 3)** (4, 5) (5, 6) – zachované poradie elementov s kľúčmi 2 – stabilné usporiadanie
    - **(2, 3) (2, 7)** (4, 5) (5, 6) – zmenené poradie elementov s kľúčmi 2 – nestabilné usporiadanie
- Príklad na viacnásobné usporiadanie – dvojice (kľúč 1, kľúč 2):
    - (4, 5) (2, 7) (2, 3) (4, 6)
- Usporiadanie najprv podľa kľúča 2, potom podľa kľúča 1:
    - (2, **3**) (4, **5**) (4, **6**) (2, **7**) – podľa kľúča 2
    - (**2**, 3) (**2**, 7) (**4**, 5) (**4**, 6) – podľa kľúča 1
- Usporiadanie najprv podľa kľúča 1, potom podľa kľúča 2:
    - (**2**, 7) (**2**, 3) (**4**, 5) (**4**, 6) – podľa kľúča 1
    - (2, **3**) (4, **5**) (4, **6**) (2, **7**) – podľa kľúča 2 – narušené poradie
- Pre zachovanie stability viacnásobného usporadúvania je potrebné usporadúvať postupne podľa kľúčov so zvyšujúcou sa prioritou.

Porovnávací algoritmus

- usporadúvací algoritmus, ktorý prechádza vstupné kľúče a na základe operácie porovnávania rozhoduje, ktorý z dvoch elementov sa má v usporiadanom poli objaviť ako prvý.
- Operácia porovnávania musí mať tieto vlastnosti:
    - Ak $a \leq b$ a $b \leq c$, tak $a \leq c$
    - Pre všetky a a b, buď $a \leq b$ alebo $b \leq a$
- Základným limitom je dolné ohraničenie počtu porovnávania Ω(n log n), ktoré je potrebné na usporiadanie postupností. Preto aj tie najlepšie algoritmy usporadúvania založené na porovnávaní majú priemernú časovú zložitosť O(n log n) – na rozdiel od neporovnávacích algoritmov, kde sa môže dosiahnuť časová zložitosť aj O(n).

Výhody porovnávacích algoritmov

- Použiteľné pre rôzne dátové typy
- Jednoduchá implementácia porovnávania n-tíc v lexikografickej postupnosti
- Reverzná funkcia porovnávania = reverzne usporiadaná postupnosť

19

Najznámejšie algoritmy usporadúvania

- založené na porovnávaní:
  - usporadúvanie výberom,
  - vkladaním,
  - výmenou,
  - zlučovaním,
  - quicksort,
  - heapsort, ...
- neporovnávacie algoritmy:
  - radix sort,
  - counting sort,
  - bucket sort, ...

20

Usporadúvanie vkladaním

- algoritmus, ktorý realizuje usporadúvanie priamym vkladaním.
- vychádza z predpokladu, že do už usporiadanej postupnosti sa na správne miesto vloží ďalší prvok.
- ak sa miesto, na ktoré sa nový prvok vkladá, zisťuje binárnym vyhľadávaním, hovoríme o usporadúvaní binárnym vkladaním

21

Usporadúvanie vkladaním

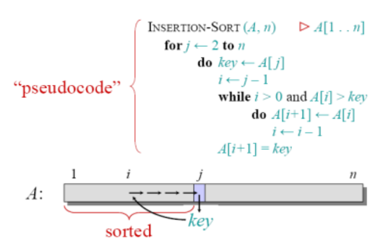
- Príklad:
  - usporiadajte vkladaním pole **6 4 5 2 3 1 7**
    - 1. krok 6 | **4** 5 2 3 1 7 –> **4** 6 | 5 2 3 1 7
    - 2. krok 4 6 | **5** 2 3 1 7 –> 4 **5** 6 | 2 3 1 7
    - 3. krok 4 5 6 | **2** 3 1 7 –> **2** 4 5 6 | 3 1 7
    - 4. krok 2 4 5 6 | **3** 1 7 –> 2 **3** 4 5 6 | 1 7
    - 5. krok 2 3 4 5 6 | **1** 7 –> **1** 2 3 4 5 6 | 7
    - 6. krok 1 2 3 4 5 6 | **7** –> 1 2 3 4 5 6 **7** |
    - 7. krok 1 2 3 4 5 6 7 | –> 1 2 3 4 5 6 7 |

22

Usporadúvanie vkladaním

- Časová zložitosť závisí od vstupného poľa
  - pre takmer usporiadané vstupné pole algoritmus prebehne rýchlo – vtedy sa akoby vymieňali len dva susedné prvky (najpravejší z usporiadanej časti s najľavejším z neusporiadanej časti) a nedochádza tak k posunu ostatných prvkov.

23



24

Usporadúvanie vkladaním

```
Procedure InsertionSort(var A: pole)
Var i, j, x : Integer;

BEGIN
  for i:= 2 to Dlzka(A) do begin
     x := A[i];
     a[0] := x;
     j := i - 1;
     while x < A[j] do
          begin
               a[j+1] := a[j];
               j := j - 1;
          end;
     a[j+1] := x;
  end
```

25

Usporadúvanie vkladaním

INSERTION-SORT(A)

1. **for** *j* = 2 to length[A]
2. **do** *key* ← A[*j*]
3. //insert A[*j*] to sorted sequence A[1..*j*-1]
4. *i* ← *j*-1
5. **while** *i* >0 and A[*i*]>*key*
6. **do** A[*i*+1] ← A[*i*] //move A[*i*] one position right
7. *i* ← *i*-1
8. A[*i*+1] ← *key*

26

správnosť algoritmu Insertion Sort

- invariant cyklu
  - na začiatku každej iterácie obsahuje podpole A[1..*j*-1] pôvodné hodnoty z A[1..*j*-1] ale v usporiadanom poradí.
- dôkaz:
  - inicializácia : *j*=2, A[1..*j*-1]=A[1..1]=A[1], je usporiadané.
  - udržiavanie: každý krok cyklu udržiava platnosť invariantu.
  - ukončenie: *j*=*n*+1, takže A[1..*j*-1]=A[1..*n*] je usporiadané.

27

analýza algoritmu Insertion Sort

INSERTION-SORT(A)	cena	počet opakovaní
1. **for** *j* = 2 to length[A]	c_1	n
2. **do** *key* ← A[*j*]	c_2	n-1
3. //insert A[*j*] to sorted sequence A[1..*j*-1]	0	n-1
4. *i* ← *j*-1	c_4	n-1
5. **while** *i* >0 and A[*i*]>*key*	c_5	$\sum_{j=2}^{n} t_j$
6. **do** A[*i*+1] ← A[*i*]	c_6	$\sum_{j=2}^{n}(t_j-1)$
7. *i* ← *i*-1	c_7	$\sum_{j=2}^{n}(t_j-1)$
8. A[*i*+1] ← *key*	c_8	$n-1$

(t_j udáva, koľkokrát sa vykoná test cyklu while v riadku 5 pre danú hodnotu j)

Celková cena $T(n)$ = suma *cena* × *počet opakovaní* pre každý riadok

$$=c_1 n + c_2(n\text{-}1) + c_4(n\text{-}1) + c_5\sum_{j=2}^{n} t_j + c_6\sum_{j=2}^{n}(t_j\text{-}1) + c_7\sum_{j=2}^{n}(t_j\text{-}1) + c_8(n\text{-}1)$$

28

analýza algoritmu Insertion Sort

- cena v najlepšom prípade: prvky sú už usporiadané
  - t_j=1, a riadky 6 a 7 sa vykonajú 0 krát
  - $T(n) = c_1 n + c_2(n\text{-}1) + c_4(n\text{-}1) + c_5(n\text{-}1) + c_8(n\text{-}1)$
    $=(c_1 + c_2 + c_4 + c_5 + c_8)n - (c_2 + c_4 + c_5 + c_8) = cn + c'$
- cena v najhoršom prípade: prvky sú už usporiadané, ale v opačnom poradí
  - t_j=j,
  - so $\sum_{j=2}^{n} t_j = \sum_{j=2}^{n} j = n(n+1)/2\text{-}1$, a $\sum_{j=2}^{n}(t_j-1) = \sum_{j=2}^{n}(j-1) = n(n\text{-}1)/2$, a
  - $T(n) = c_1 n + c_2(n\text{-}1) + c_4(n\text{-}1) + c_5(n(n+1)/2\ \text{-}1) + + c_6(n(n\text{-}1)/2\ \text{-}1) + c_7(n(n\text{-}1)/2) + c_8(n\text{-}1) =((c_5 + c_6 + c_7)/2)n_2 + (c_1 + c_2 + c_4 + c_5/2\text{-}c_6/2\text{-}c_7/2+c_8)n\text{-}(c_2 + c_4 + c_5 + c_8) = an^2+bn+c$
- cena v priemernom prípade: čísla sú náhodné
  - v priemere, $t_j = j/2$. $T(n)$ bude aj v priemernom prípade stále rádu n^2, rovnako ako v najhoršom prípade.

29

časová zložitosť algoritmu Insertion Sort

- časová zložitosť:
  - v najlepšom prípade: T(n) = Θ(n),
  - v najhoršom prípade: T(n) = Θ(n^2),
  - v priemernom prípade: T(n) = Θ(n^2)
- je to rýchly algoritmus?
  - pre malé n celkom prijateľný
  - pre veľké n vonkoncom nie.

30

Usporadúvanie výmenou - bublinkové (bubble)

- bubble sort usporadúva prvky priamou výmenou
- je implementačne jednoduchý, ale neefektívny
- pri usporadúvaní porovnáva dva susedné prvky a ak nie sú v správnom poradí, vymenia sa
- procedúra sa opakuje, až kým nie sú potrebné žiadne výmeny

31

Usporadúvanie výmenou - bublinkové (bubble)

- Príklad:
  - Usporiadajte bublinkovou výmenou pole **5 1 4 2 8**
    - 1. fáza
      - » **5 1** 4 2 8 –> **1 5** 4 2 8
      - » 1 **5 4** 2 8 –> 1 **4 5** 2 8
      - » 1 4 **5 2** 8 –> 1 4 **2 5** 8
      - » 1 4 2 **5 8** –> 1 4 2 **5 8**
    - 2. fáza
      - » **1 4** 2 5 8 –> **1 4** 2 5 8
      - » 1 **4 2** 5 8 –> 1 **2 4** 5 8
      - » 1 2 **4 5** 8 –> 1 2 **4 5** 8
      - » 1 2 4 **5 8** –> 1 2 4 **5 8**

32

Usporadúvanie výmenou - bublinkové (bubble)

- Príklad:
  - na začiatku sme mali pole **5 1 4 2 8**
    - 3. fáza
      - » **1 2** 4 5 8 –> **1 2** 4 5 8
      - » 1 **2 4** 5 8 –> 1 **2 4** 5 8
      - » 1 2 **4 5** 8 –> 1 2 **4 5** 8
      - » 1 2 4 **5 8** –> 1 2 4 **5 8**
  - na konci máme usporiadané pole **1 2 4 5 8**

33

Bublinkové usporadúvanie

```
Procedure BubbleSort(var A: pole)
Var i, j, t : integer;
Begin
  for i := Dlzka(A) downto 1 do
     for j := 1 to Dlzka(A)-1 do
          if A[j] > A[j+1] then
          begin
               T := A[j];
               A[j] := A[j+1];
               A[j+1] := T;
          end;
end;
```

34

Bublinkové usporadúvanie

```
void bubbleSort (Array A) {
    n = dĺžka(A);
    boolean isSorted = false;
    while(!isSorted) {
        isSorted = true;
        for(i = 0; i<n; i++) {
            if(A[i] > A[i+1]) {
                        int T = A[i];
                        A[i] = A[i+1];
                        A[i+1] = T;
                        isSorted = false;
            }
        }
}
```

35

časová zložitosť bublinkového usporadúvania

- 1 prechod = presun najväčšieho prvku na koniec
- i-tý prechod: n-i+1 operácií
- čas:
  $(n-1) + (n-2) + \ldots + 1 = (n-1)\, n / 2 = O(n^2)$
- ktorý prípad je najlepší?
- ktorý prípad je najhorší?

36

Usporadúvanie výberom

- algoritmus realizuje usporadúvanie priamym výberom
- vychádza z prepokladu, že najmenší prvok môžeme zaradiť priamo na začiatok vstupného poľa, najmenší prvok zo zvyšku poľa zase na jeho začiatok atď.
- podľa toho, či usporadúva prvky vzostupne/zostupne, sa môže označovať ako MinSort/MaxSort

Usporadúvanie výberom

- Príklad:
  - usporiadajte výberom pole **6 4 5 2 3 1 7**
    - 1. krok | 6 4 5 2 3 **1** 7 –> **1** | 6 4 5 2 3 7
    - 2. krok 1 | 6 4 5 **2** 3 7 –> 1 **2** | 6 4 5 3 7
    - 3. krok 1 2 | 6 4 5 **3** 7 –> 1 2 **3** | 6 4 5 7
    - 4. krok 1 2 3 | 6 **4** 5 7 –> 1 2 3 **4** | 6 5 7
    - 5. krok 1 2 3 4 | 6 **5** 7 –> 1 2 3 4 **5** | 6 7
    - 6. krok 1 2 3 4 5 | **6** 7 –> 1 2 3 4 5 **6** | 7
    - 7. krok 1 2 3 4 5 6 | **7** –> 1 2 3 4 5 6 **7** |
- Miera usporiadanosti vstupného poľa nemá vplyv na časovú zložitosť – vždy sa vykoná maximálny počet krokov.

Usporadúvanie výberom

```
procedure SelectionSort(var A: pole)
var i, j, t : integer;
begin
  for i:= 1 to Dlzka(A) – 1 do
          MinIndex := i;
  for j:= i to Dlzka(A)- 1 do
          if A[MinIndex] > A[j] then
               MinIndex := j;
          T := A[i];
          A[i] := A[MinIndex];
          A[MinIndex] := T;
          end;
```


Donald Shell

1.3.1924 –

1959 PhD Uni of Cincinnati

Shell, D.L. (1959). "A high-speed sorting procedure". Communications of the ACM 2 (7): 30–32.

Shellovo usporadúvanie

- algoritmus, ktorý realizuje usporadúvanie priamym vkladaním so zmenšovaním prírastku
- je to vlastne zlepšenie usporadúvania vkladaním a bublinkového
- táto metóda je jedna z najrýchlejších pre usporiadanie menších postupností (menej ako 1000 prvkov)

Shellovo usporadúvanie

- Príklad, prírastky n = n/2, atď (pôvodný návrh Shella):
  - Usporiadajte pole **6 4 5 2 8 3 1 7** pomocou algoritmu shell sort, n = 8/2 = 4
    - 1. krok, krokovanie 4, vyznačené čísla sa usporiadajú vkladaním
      - » **6** 4 5 2 **8** 3 1 7 –> **6** 4 5 2 **8** 3 1 7
      - » 6 **4** 5 2 8 **3** 1 7 –> 6 **3** 5 2 8 **4** 1 7
      - » 6 3 **5** 2 8 4 **1** 7 –> 6 3 **1** 2 8 4 **5** 7
      - » 6 3 1 **2** 8 4 5 **7** –> 6 3 1 **2** 8 4 5 **7**
    - 2. krok, krokovanie 2
      - » **6** 3 **1** 2 **8** 4 **5** 7 –> **1** 3 **5** 2 **6** 4 **8** 7
      - » 1 **3** 5 **2** 6 **4** 8 **7** –> 1 **2** 5 **3** 6 **4** 8 **7**
    - 3. krok, krokovanie 1
      - » **1 2 5 3 6 4 8 7** –> 1 2 3 4 5 6 7 8

Shellovo usporadúvanie

```
procedure ShellSort(var f: pole)
var i, j, h, v, N : integer;
begin
   N := Dlzka(f);
   h := 1;
   repeat // priprav prirastky podla Knutha
        h := (3 * h) + 1;
   until h > N;

   repeat
        h := (h div 3);
        for i:= (h + 1) to N do begin
            v := f[i];
            j := i;
            while ( (j > h) and ( f[j-h] > v ) ) do begin
                 f[j] := f[j - h];
                 dec(j, h);
            end;
            f[j] := v;
        end;
   until h = 1;
end
```


Shellovo usporadúvanie

- používa postupnosť prírastkov $h_1, h_2, \ldots, h_t$
- môže byť ľubovoľná postupnosť, len musí byť $h_1 = 1$ a $h_1 < h_2 < \ldots < h_t$
- rôzne voľby postupnosti prírastkov vedú k rôzne efektívnym verziám algoritmu.

Shellovo usporadúvanie

urči postupnosť prírastkov $h_1, h_2, \ldots, h_t$

urob t prechodov cez postupnosť prvkov

prechod 1: h_t – usporiadaná postupnosť, t.j.
pre $\forall i$ $a[i] \leq a[i + h_t]$

prechod 2: h_{t-1} – usporiadaná postupnosť, t.j.
pre $\forall i$ $a[i] \leq a[i + h_{t-1}]$

...

prechod t: h_1 – usporiadaná postupnosť, t.j.
pre $\forall i$ $a[i] \leq a[i + h_1]$

- v každom prechode dosiahni h_k – usporiadanie pomocou usporadúvania vkladaním

Shellovo usporadúvanie

- po každom prechode pri nejakom prírastku h_k, pre všetky i máme $a[i] \leq a[i + h_k]$ t.j. všetky prvky umiestnené od seba o h_k miest sú usporiadané.
- posledný prechod sa robí s prírastkom $h_1 = 1$
  $\Rightarrow$ pre $\forall i$ $a[i] \leq a[i + 1]$
  $\Rightarrow$ postupnosť a[ ] je usporiadaná.

Donald Knuth

10. január 1938 Milwaukee, Wisconsin –

BS and MS matematika - Case Institute of Technology

1963 - Ph.D. matematika California Institute of Technology

profesor na California Institute of Technology,

1968 – monografia The Art of Computer Programming (Umenie počítačového programovania), vydal postupne 4 zväzky, plán bol 7 zväzkov

LR syntaktická analýza. D. E. Knuth, On the translation of languages from left to right, Info. Control 8 (1965), 607-639.

TeX – štandard počítačového sádzania

Shellovo usporadúvanie: prírastky

- Shell: $\lfloor n/2 \rfloor, \lfloor n/2^2 \rfloor, \lfloor n/2^3 \rfloor, \ldots, 1$ alebo (ešte horšie) postupnosť prírastkov mocniny 2: $O(n^2)$
- Knuth: 1, 4, 13, 40, 121, 364,1093, 3280, 9841,... t.j. $h_1 = 1$, $h_{i+1} = 3*h_i + 1$
- Knuth: blízko $O(n \log^2 n)$ a $O(n^{1.25})$
- Hibbard: 1, 3, 7,...2^{k-1}: $O(n^{3/2})$
- Sedgewick: 1, 8, 23, 77, 281, 1073, 4193, 16577..., ($4^{i+1} + 3 \cdot 2^i + 1$) pre i > 0, má byť lepšia než Knuth
- Pratt: $\log^2 n$ prírastkov $2^i 3^j < \lfloor n/2 \rfloor$ (1, 2, 3, 4, 6, 8, 9, 12, 16, …)

Shellovo usporadúvanie

- najlepší prípad: postupnosť je už usporiadaná – bude treba menej porovnaní
- najhorší prípad (pre postupnosť prírastkov podľa Pratta): O(n $\log^2$ n)
- priemerný prípad (pre postupnosť prírastkov podľa Pratta): Θ(n $\log^2$ n)

jedna varianta inšpirovaná Shellovým usporadúvaním: Shaker-sort*

h-utrasenie postupnosti a[1], ..., a[n]:

porovnávanie a prípadná výmena týchto dvojíc prvkov v uvedenom poradí:
(1; 1 + h), (2; 2 + h),..., (N - h - 1; N - 1), (N - h; N),
(N - h - 1; N - 1), (N - h - 2; N - 2), ..., (2; 2 + h); (1; 1 + h)
všimnime si, že medzi prvkami každej dvojice je rovnaká vzdialenosť h.

usporadúvanie utrasením:

- urči postupnosť prírastkov $h_1, h_2, \dots, h_t$
- urob t prechodov cez postupnosť prvkov
  - prechod 1: h_1 – utrasenie
  - prechod 2: h_2 – utrasenie
  - ...
  - prechod t: h_t – utrasenie.
- usporiadaj vkladaním // alebo opakuj 1-utrasenie dovtedy, kým nie je postupnosť usporiadaná

*Incerpi, J. and R. Sedgewick, Practical Variations on Shellsort, Inform. Process. Lett. 26 (1987), 37-43.

Sir Charles Antony Richard Hoare
C.A.R. (Tony) Hoare

11.1..1934 Colombo, Ceylon – britský informatik
1956 - Bc klasické štúdiá, Oxford
1968 – profesor informatiky na Královninej univerzite v Belfaste
1977 – profesor informatiky na Oxforde
1960 – quicksort
Hoarova logika pre dokazovanie správnosti programov
Communicating Sequential Processes (CSP)

Sú dva spôsoby ako navrhovať softvér: jeden je urobiť ho tak jednoduchý, že v ňom zjavne nie sú nedostatky, druhý je urobiť ho tak zložitý, že v ňom nie sú zjavné nedostatky. Prvý spôsob je omnoho zložitejší.

Rýchle usporadúvanie (Quick Sort)

- quicksort alebo usporadúvanie rozdeľovaním je jeden z najrýchlejších známych algoritmov založených na porovnávaní prvkov
- jeho priemerná doba výpočtu je najlepšia zo všetkých podobných algoritmov
- nevýhodou je, že pri nevhodnom usporiadaní vstupných dát môže byť časová aj pamäťová náročnosť omnoho väčšia

Rýchle usporadúvanie (Quick Sort)

```
QUICKSORT(A, p, r)
  if p < r
     then q ← PARTITION(A, p, r)
          QUICKSORT(A, p, q − 1)
          QUICKSORT(A, q + 1, r)
```


rozčlenenie

```
PARTITION(A, p, r)
  x ← A[r]
  i ← p − 1
  for j ← p to r − 1
      do if A[j] ≤ x
            then i ← i + 1
                 exchange A[i] ↔ A[j]
  exchange A[i + 1] ↔ A[r]
  return i + 1
```

rozčlenenie

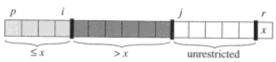


Figure 7.2 The four regions maintained by the procedure PARTITION on a subarray $A[p..r]$. The values in $A[p..i]$ are all less than or equal to x, the values in $A[i+1..j-1]$ are all greater than x, and $A[r] = x$. The values in $A[j..r-1]$ can take on any values.

rozčlenenie

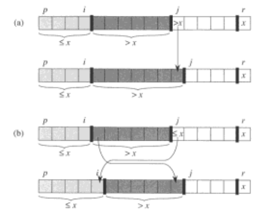


Figure 7.3 The two cases for one iteration of procedure PARTITION. **(a)** If $A[j] > x$, the only action is to increment j, which maintains the loop invariant. **(b)** If $A[j] \le x$, index i is incremented, $A[i]$ and $A[j]$ are swapped, and then j is incremented. Again, the loop invariant is maintained.

Hoarova formulácia rozčleňovania

```
HOARE-PARTITION(A, p, r)
x ← A[p]
i ← p − 1
j ← r + 1
while TRUE
      do repeat j ← j − 1
            until A[j] ≤ x
         repeat i ← i + 1
            until A[i] ≥ x
         if i < j
            then exchange A[i] ↔ A[j]
            else return j
```


trochu iná formulácia rozčleňovania

```
Partition(A,left,right):int
 p:=A[left]; l:=left+1; r:=right;
 while l<r do
    while A[l]<p do l:=l+1;
    while A[r]≥p do r:=r-1;
    if l<r then swap(A, l, r)
 A[left]:=A[r]; A[r]:=p;
 return r;
```


príklad rozčleňovania

- choose pivot: 4 3 6 9 2 4 3 1 2 1 8 9 3 5 6
- search: 4 3 6 9 2 4 3 1 2 1 8 9 3 5 6
- swap: 4 3 3 9 2 4 3 1 2 1 8 9 6 5 6
- search: 4 3 3 9 2 4 3 1 2 1 8 9 6 5 6
- swap: 4 3 3 1 2 4 3 1 2 9 8 9 6 5 6
- search: 4 3 3 1 2 4 3 1 2 9 8 9 6 5 6
- swap: 4 3 3 1 2 2 3 1 4 9 8 9 6 5 6
- search: 4 3 3 1 2 2 3 1 4 9 8 9 6 5 6 (l > r)
- swap with pivot: 1 3 3 1 2 2 3 4 4 9 8 9 6 5 6

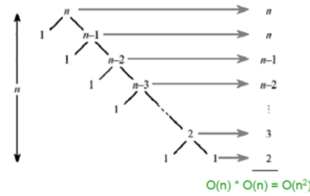
analýza rýchleho usporadúvania

- *najhorší prípad?*
  - rozčlenenie je vždy nevyvážené
- *najlepší prípad?*
  - rozčlenenie je dokonale vyvážené
- *ktorý prípad je častejší?*
  - ten druhý, s prevahou, okrem...
- *je nejaký vstup, ktorý spôsobí najhorší prípad?*
  - áno: usporiadaná postupnosť

analýza rýchleho usporadúvania

- v najhoršom prípade:
  $T(1) = \Theta(1)$
  $T(n) = T(n - 1) + \Theta(n)$
- z čoho vyjde
  $T(n) = \Theta(n^2)$

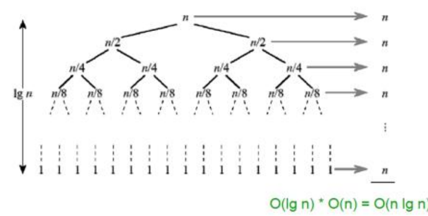
analýza rýchleho usporadúvania



analýza rýchleho usporadúvania

- v najlepšom prípade:
  $T(n) = 2T(n/2) + \Theta(n)$
- z čoho vyjde
  $T(n) = \Theta(n \lg n)$

analýza rýchleho usporadúvania



vylepšenie rýchleho usporadúvania

- naozajstnú záruku dáva r.u. na usporiadanú postupnosť s nevhodnou voľbou pivota - vtedy je kvadratický $O(n^2)$
- možnosti vylepšenia:
  - znáhodnenie (poradia) vstupnej postupnosti alebo
  - náhodná voľba pivota
- *v čom je vylepšenie?*
  - zabezpečením, že vstup nebude taký, že spôsobí r.u. vykonávané v kvadratickom čase $O(n^2)$

analýza rýchleho usporadúvania: priemerný prípad

- za predpokladu „náhodného“ vstupu je čas v priemernom prípade omnoho bližší k $O(n \lg n)$ než k $O(n^2)$
- názorné vysvetlenie/príklad:
  - predpokladajme, že rozčlenenie vždy dopadne 9-ku-1 (dosť nevyvážené)!
  - rekurentná rovnica:
    $T(n) = T(9n/10) + T(n/10) + n$

analýza rýchleho usporadúvania: priemerný prípad

- intuitívne sa dá predpokladať, že v skutočnosti r.u. prebehne ako zmes "zlých" a "dobrých" rozčlenení
  - dobré a zlé budú rozložené náhodne v strome rekurzie
  - predpokladajme, že sa bude striedať najlepší (n/2 : n/2) a najhorší prípad (n-1 : 1)
  - *čo sa stane, ak sa zle rozčlení hneď koreňový vrchol a potom sa dobre rozčlení z toho rezultujúci (n-1) vrchol?*

67

analýza rýchleho usporadúvania: priemerný prípad

- intuitívne sa dá predpokladať, že v skutočnosti r.u. prebehne ako zmes "zlých" a "dobrých" rozčlenení
  - dobré a zlé budú rozložené náhodne v strome rekurzie
  - predpokladajme, že sa bude striedať najlepší (n/2 : n/2) a najhorší prípad (n-1 : 1)
  - *čo sa stane, ak sa zle rozčlení hneď koreňový vrchol a potom sa dobre rozčlení z toho rezultujúci (n-1) vrchol?*
    - dostaneme 3 podpostupnosti s veľkosťami 1, (n-1)/2, (n-1)/2
    - celková cena rozčlenení = n + n -1 = 2n -1 = O(n)
    - o nič horšie ako keby sme mali dobré rozčlenenie hneď v koreňovom vrchole!

68

zlepšenie voľbou lepšieho pivota

- zatiaľ sme volili
  - krajný prvok (síce O(1), ale...)
  - jedno, či prvý alebo posledný
- zaručene najlepšia voľba?
  - koľko prvkov vľavo, toľko vpravo od neho
  - porovnaj definíciu mediánu
  - ako určiť medián? usporiadať a zvoliť prvok presne v strede!
- znáhodnenie

69

Iný spôsob voľby pivota a rozčlenenia

```
RANDOMIZED-PARTITION(A, p, r)
i ← RANDOM(p, r)
exchange A[r] ↔ A[i]
return PARTITION(A, p, r)
```

70

Znáhodnené rýchle usporadúvanie

```
RANDOMIZED-QUICKSORT(A, p, r)
if p < r
  then q ← RANDOMIZED-PARTITION(A, p, r)
       RANDOMIZED-QUICKSORT(A, p, q − 1)
       RANDOMIZED-QUICKSORT(A, q + 1, r)
```

71

Varianta

```
QUICKSORT′(A, p, r)
while p < r
   do ▷ Partition and sort left subarray.
      q ← PARTITION(A, p, r)
      QUICKSORT′(A, p, q − 1)
      p ← q + 1
```

72

Rýchle usporadúvanie - zhrnutie

- Základom je rozdelenie postupnosti na dve časti. V jednej časti sú čísla väčšie a v druhej menšie ako zvolená hodnota, ktorá sa nazýva **pivot**
- Je dôležité správne zvoliť pivot, najlepšie tak, aby rozdelené časti boli približne rovnako veľké, čím sa získa najrýchlejšie možné usporiadanie

73

Rýchle usporadúvanie (Quick Sort)

```
procedure quickSort(pole, lavy, pravy)
  if lavy < pravy
     pivot := lavy
     for i := lavy + 1 to pravy
          if pole[i] < pole[lavy]
               pivot := pivot + 1
               swap(pole[pivot], pole[i])
          end
     end
     swap(pole[pivot], pole[lavy])
     quickSort(pole, lavy, pivot)
     quickSort(pole, pivot + 1, pravy)
  end
end
```

74

Porovanie jednotlivých metód

	časová zložitosť
Vkladaním	0(N^2)
Výmenou	0(N^2)
Výberom	0(N^2)
Zlučovaním dvojcestné	0(N log N)
Radixové	0(N log N)
Výpočtom adries	0(N)
Shellovo	0(N $\log^2$ N)
rýchle	0(N log N)

75

János (John) von Neumann

(December 28, 1903, Budapest – February 8, 1957)

americký matematik, narodený v Rakúsku-Uhorsku.

- matematická štatistika a ekonometria,
- kvantová mechanika,
- ekonómia a teória hier,
- informatika:
  - architektúra počítačov
  - merge sort

76

Rozdeľuj a panuj – divide et impera

- metóda
  - riešenia problémov
  - navrhovania algoritmov
- ak problém je dostatočne jednoduchý, tak ho vyrieš priamo
- ak problém
  - je rozsiahly
  - dá sa rozdeliť na viacero menších neprekrývajúcich sa podproblémov
- tak
  - **rozdeľ** problém na 2 alebo viac menších podproblémov
  - (**panuj**) použi ten istý postup rekurzívne na riešenie podproblémov
  - **skombinuj** získané riešenia podproblémov do riešenia pôvodného problému

77

Usporadúvanie zlučovaním (merge sort)

- vychádza z metódy rozdeľuj a panuj, t.j. ľahšie sa usporiada menej položiek ako veľa
- usporadúvanie zlučovaním je opakom usporadúvania rozdeľovaním (quick sort)
- skladá sa z dvoch častí: rozdelenie na usporiadané podpostupnosti a opätovné spájanie
- usporiadané podpostupnosti sa rekurzívne zlučujú do jednej spoločnej usporiadanej postupnosti

78

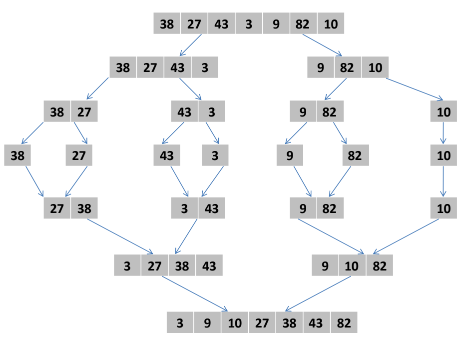
Usporadúvanie zlučovaním (merge sort)

- postupnosť najprv rozdelíme na dve približne rovnako veľké časti (pri nepárnom počte je jedna časť väčšia)
- ďalej sa budeme zaoberať každou z týchto dvoch častí zvlášť, a to takým istým spôsobom, t.j. rozdelíme ich na dve časti
- znovu vezmeme prvú z nich a rozdelíme ju na dve, atď. ... až kým nedostaneme jednoprvkové úseky
- je zrejmé, že jednoprvkové úseky sú vždy usporiadané
- teraz použijeme druhú časť algoritmu: zlúčenie dvoch usporiadaných častí tak, aby aj novovzniknutá časť bola usporiadaná, t.j. dostávame časť s dvoma položkami
- podobne sa rozdelí a zlúči aj druhá časť z rozdelenia a dostávame usporiadanú druhú dvojprvkovú časť
- ten istý algoritmus zlúčenia dvoch usporiadaných častí použijeme aj teraz a dostávame usporiadanú štvorprvkovú časť, atď.
- algoritmus sa bude opakovať, až kým nebude usporiadané celé pole

Usporadúvanie zlučovaním (merge sort)

- Výhody oproti quick sort:
  - stabilný algoritmus usporiadania
  - lepšie možnosť využitia paralelného spracovania
  - sekvenčný prístup k údajom umožňuje pracovať nad veľkým množstvom údajov, uložených na médiach so sekvenčným prístupom, bez nutnosti načítavať tieto údaje do pamäte
  - umožňuje „on-line“ pridávanie ďalších podpostupností (ktoré sa najprv usporiadajú) počas procesu zlučovania.

Usporadúvanie zlučovaním (merge sort)



Usporadúvanie zlučovaním (merge sort)

- Koncept rekurzívneho algoritmu:
  1. ak je postupnosť dĺžky 0 alebo 1, tak je postupnosť usporiadaná, ak nie, tak rozdeľ neusporiadanú postupnosť na dve podpostupnosti s polovičnou veľkosťou
  2. usporiadaj každú podpostupnosť rekurzívnym aplikovaním tohože algoritmu
  3. zlúč dve usporiadané podpostupnosti do jednej usporiadanej postupnosti.

MERGE-SORT(A,*p*,*r*)

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

na usporiadanie postupnosti zapísanej v A[1..n] sa zavolá MERGE-SORT(A,1,*n*) (n=length(A))

Merge – zlúč

- predpodmienka:
  - A[p..q] a A[q+1..r] sú usporiadané
- popodmienka
  - A[p..r] je usporadané

Merge (A,p,q,r)

$$s \leftarrow q - p + 1; t \leftarrow r - q;$$
$$L \leftarrow A[p..q]; R \leftarrow A[q+1, r]$$
$$L[s+1] \leftarrow \infty; R[t+1] \leftarrow \infty$$
$$A[p..r] \leftarrow MergeArray(L, R)$$

MergeArray – zlúč polia

- predpodmienka:
  - L[1..s] a R[1..t] sú usporiadané
- popodmienka
  - MergeArray(L, R) vytvorí jeden usporiadaný vektor A[1..s+t] z prvkov v L a R

A = MergeArray(*L*,*R*)
- $L[s+1] \leftarrow \infty; R[t+1] \leftarrow \infty$
- $i \leftarrow 1; j \leftarrow 1$
- **for** *k* ← *1* **to** *s* + *t*
  - **do if** $L[i] \le R[j]$
    - **then** $A[k] \leftarrow L[i]; i \leftarrow i+1$
    - **else** $A[k] \leftarrow R[j]; j \leftarrow j+1$

85

Správnosť procedúry MergeArray

- invariant cyklu for:
  - na začiatku vykonania každého kroku:
    - A[1..k-1] obsahuje k-1 najmenších prvkov z L[1..s+1] a R[1..t+1] v usporiadanom poradí. Okrem toho L[i] a R[j] sú najmenšie prvky vo svojich poliach, ktoré neboli skopírované späť do A

86

Správnosť procedúry MergeArray

- náčrt dôkazu indukciou
- inicializácia (invariant platí na začiatku):
  - k=1 teda A[1..k-1] je prázdne. Obsahuje k-1=0 najmenších prvkov z L a R. i=1 a j=1 preto naozaj L[i] a R[j] sú najmenšie prvky svojich polí, ktoré ešte neboli skopírované späť do A.

87

Správnosť procedúry MergeArray

- náčrt dôkazu indukciou
- indukčný krok (invariant platí po každom kroku cyklu):
  - bez straty všeobecnosti predpokladajme, že L[i] <= R[j] a preto L[i] je najmenší prvok, ktorý sa ešte neskopíroval do A. Preto po jeho skopírovaní do A[k] podpole A[1..k] obsahuje k najmenších prvkov. Zvýšením k o 1 a i o 1 sa obnoví platnosť invariantu pred ďalším krokom.

88

Správnosť procedúry MergeArray

- náčrt dôkazu indukciou
- ukončenie (z platnosti invariantu vyplýva čiastočná správnosť):
  - keď sa vykonanie cyklu skončí, platí k=s+t+1. Podľa invariantu obsahuje A k-1 najmenších prvkov z L a R v usporiadanom poradí. k-1=s+t+1-1=s+t čiže všetky prvky sú usporiadané.

89

Časová výpočtová zložitosť MergeArray

- v každom kroku cyklu sa vykoná:
  - 1 porovnanie
  - 1 priradenie (skopírovanie jedného prvku do A)
  - 2 inkrementácie (*k* a *i* alebo *j*)
- spolu v každom kroku cyklu 4 operácie
- celkovo čas 4.(s+t), čiže O(n)

90

časová zložitosť rozdeľuj a panuj

- opisuje rekurentná rovnosť
- predpokladajme, že $T(n)$ je čas riešenia problému rozsahu n.
- $T(n) = \begin{cases} \Theta(1) & \text{if } n \leq n_c \\ aT(n/b)+D(n)+C(n) & \text{if } n > n_c \end{cases}$

  kde a: počet podproblémov

  n/b: veľkosť každého podproblému

  $D(n)$: cena operácie rozdelenia

  $C(n)$: cena operácie kombinovania (zlúčenia)

časová zložitosť rozdeľuj a panuj

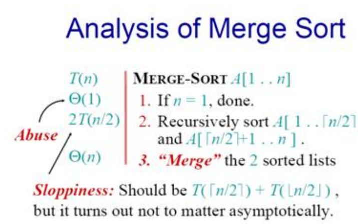
- $T(n) = \begin{cases} \text{čas potrebný na vyriešenie bázovej inštancie problému} & \text{if } n=1 \\ \text{počet podproblémov * } T(n/\text{podiel veľkosti podproblému}) \\ \quad + \text{čas na rozdelenie + čas na zlúčenie} & \text{if } n>1 \end{cases}$

- $T(n) = \begin{cases} c & \text{if } n=1 \\ 2T(n/2)+ cn & \text{if } n>1 \end{cases}$

analýza MERGE-SORT

- **Divide**: $D(n) = \Theta(1)$
- **Impera**: a=2,b=2, takže $2T(n/2)$
- **Skombinuj**: $C(n) = \Theta(n)$
- $T(n) = \begin{cases} \Theta(1) & \text{if } n=1 \\ 2T(n/2)+ \Theta(n) & \text{if } n>1 \end{cases}$
- $T(n) = \begin{cases} c & \text{if } n=1 \\ 2T(n/2)+ cn & \text{if } n>1 \end{cases}$

Časová výpočtová zložitosť usporadúvania zlučovaním



rekurentná rovnosť pre T(n)

Recurrence for Merge Sort

$$T(n) = \begin{cases} \Theta(1) \text{ if } n = 1; \\ 2T(n/2) + \Theta(n) \text{ if } n > 1. \end{cases}$$

- We shall usually omit stating the base case when $T(n) = \Theta(1)$ for sufficiently small n, but only when it has no effect on the asymptotic solution to the recurrence.

rekurentná rovnosť pre T(n) - riešenie

Recursion Tree

Solve $T(n) = 2T(n/2) + cn$, where $c > 0$ is constant.

rekurentná rovnosť pre T(n) - riešenie

Recursion Tree

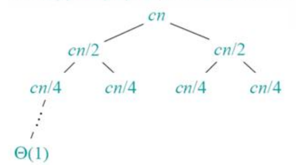
Solve $T(n) = 2T(n/2) + cn$, where $c > 0$ is constant.

$T(n)$

rekurentná rovnosť pre T(n) - riešenie

Recursion Tree

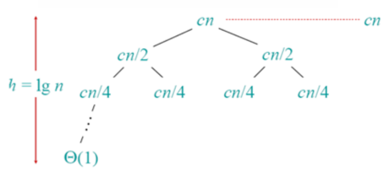
Solve $T(n) = 2T(n/2) + cn$, where $c > 0$ is constant.



rekurentná rovnosť pre T(n) - riešenie

Recursion Tree

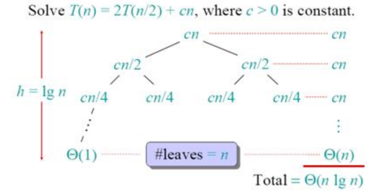
Solve $T(n) = 2T(n/2) + cn$, where $c > 0$ is constant.



rekurentná rovnosť pre T(n) - riešenie

Recursion Tree

Solve $T(n) = 2T(n/2) + cn$, where $c > 0$ is constant.



rekurentná rovnosť pre T(n) - riešenie

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

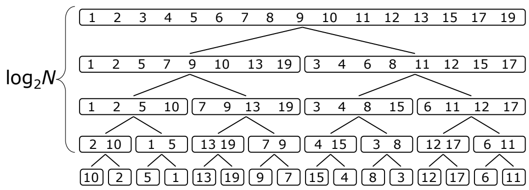
$$\begin{aligned}
T(n) &= 2T(n/2) + n && \text{substitute} \\
&= 2\big(2T(n/4) + n/2\big) + n && \text{expand} \\
&= 2^2 T(n/4) + 2n && \text{substitute} \\
&= 2^2(2T(n/8) + n/4) + 2n && \text{expand} \\
&= 2^3 T(n/8) + 3n && \text{observe the pattern} \\
T(n) &= 2^i T(n/2^i) + in && \textit{Let } 2^i = n,\ i = \lg n \\
&= 2^{\lg n} T(n/n) + n \lg n = n + n \lg n
\end{aligned}$$

porovnanie usporadúvania vkladaním a zlučovaním

Insertion and Merge Sort

- $\Theta(n \lg n)$ grows more slowly than $\Theta(n^2)$.
- Therefore, merge sort asymptotically beats insertion sort in the worst case.
- In practice, merge sort beats insertion sort for $n > 30$ or so.
- Go test it out for yourself!

strom rekurzie

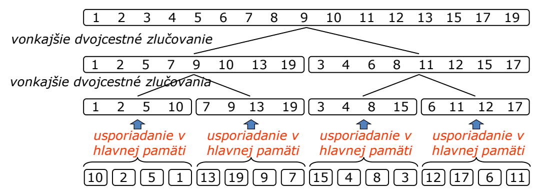


- na každej úrovni: zlúč usporiadané úseky (podpostupnosti) veľkosti *x* do úsekov veľkosti 2*x*, zníž počet úsekov na polovicu.
- ako by sa tento postup dal použiť na údaje zapísané v súbore vo vedľajšej pamäti?

103

usporadúvanie zlučovaním vo vonkajšej pamäti

- myšlienka: zväčšiť veľkosť pôvodných úsekov
  - veľkosť pôvodného úseku podľa veľkosti dostupnej časti hlavnej pamäti (*M* miest)



104

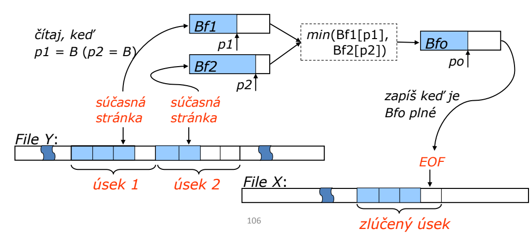
usporadúvanie zlučovaním vo vonkajšej pamäti

- vstupný súbor *X*, prázdny súbor Y
- *fáza* 1: repeat until eof(*X*):
  - prečítaj ďalších *M* prvkov z *X*
  - usporiadaj ich v hlavnej pamäti
  - zapíš ich na koniec súboru *Y*
- *fáza* 2: while not empty(*Y*):
  - vyprázdni *X*
  - *MergeAllRuns*(*Y*, *X*)
  - *súbor X* sa premenuje na *Y*, *Y* sa premenuje na *X*

105

zlučovanie pri vonkajšom usporadúvaní

- *MergeAllRuns*(*Y*, *X*): repeat until eof(*Y*):
  - vykonaj *TwowayMerge,* aby sa zlúčili nasledujúce dva úseky z Y do jedného, ktorý sa zapíše na koniec *X*
- *TwowayMerge*: používa 3 vektory v hlavnej pamäti veľkosti B



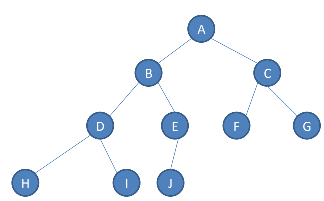
106

Usporadúvanie haldou (Heap Sort)

- Pri usporadúvaní využijeme špeciálny pojem **halda** - je to dátová štruktúra, ktorá:
  - má tvar "skoro" úplného binárneho stromu (len na poslednej úrovni binárneho stromu môžu chýbať synovia (vrcholov predposlednej úrovne) a to tak, že ak chýba nejaký syn, tak budú chýbať aj všetci synovia vpravo na najnižšej úrovni
  - pre všetky vrcholy stromu platí, že otec má väčšiu (alebo rovnú) hodnotu ako jeho synovia - ak existujú
  - haldu budeme reprezentovať v jednorozmernom poli indexovanom od 0 tak, že koreň stromu je na indexe 0 a i-ty vrchol má synov na indexoch **2*i+1** a **2*i+2**

107

Usporadúvanie haldou



	A	B	C	D	E	F	G	H	I	J	
0	1	2	3	4	5	6	7	8	9	10	11

108

Usporadúvanie haldou

- usporadúva prvky pomocou dátovej štruktúry binárna halda
- predstavuje efektívnejšiu verziu usporadúvania výberom
- najväčší (resp. najmenší) prvok sa vyberá z koreňa max-haldy (resp. min-haldy)
- max-halda je strom, pre ktorý platí, že každý potomok v strome má menšiu hodnotu ako jeho rodič (min-halda naopak)

109

Usporadúvanie haldou

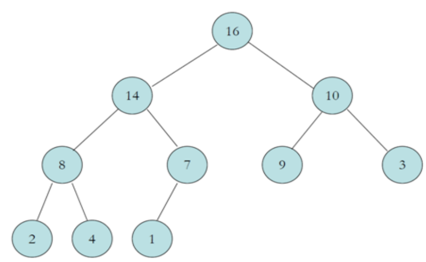
- samotný algoritmus má dve fázy:
  - vytvorenie haldy
  - v halde je koreň (t.j. prvý prvok poľa) najväčší prvok zo všetkých, jeho výmena s posledným prvkom (ešte neusporiadaného) poľa a nové „vyhaldovanie", t.j. oprava haldy
- zakaždým pracujeme s o 1 kratším poľom - haldou -> na jeho konci sa postupne sústreďujú najväčšie prvky

110

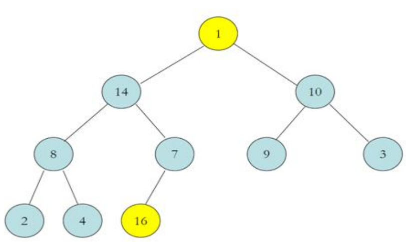
Heapsort

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

111

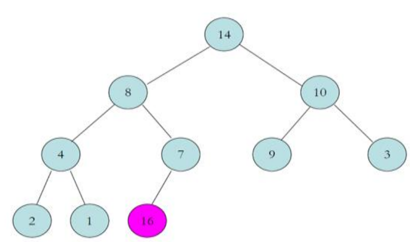


112



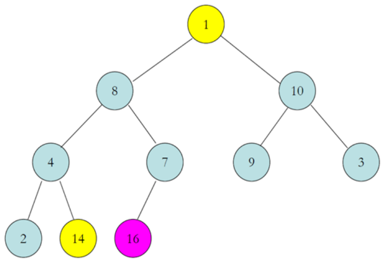
Swap $A[1] \leftrightarrow A[i]$

113

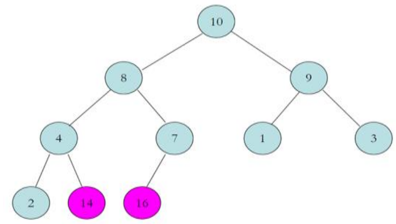


heap-size ← heap-size – 1
MAX-HEAPIFY(A, 1)

114

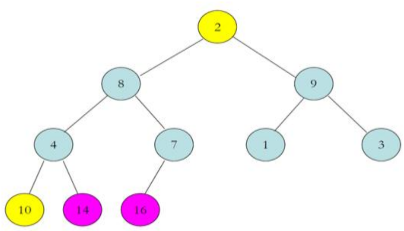


115



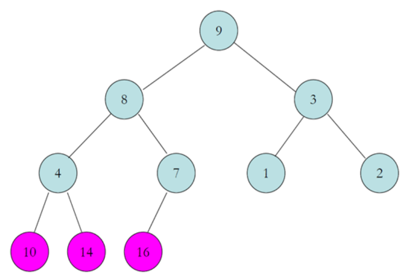
heap-size ← heap-size − 1
MAX-HEAPIFY(A, 1)

116

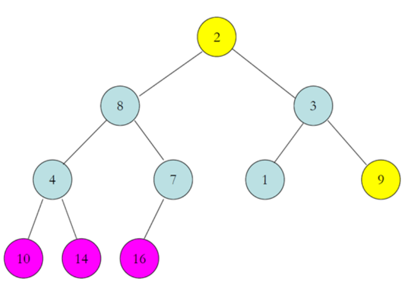


Swap $A[1] \leftrightarrow A[i]$

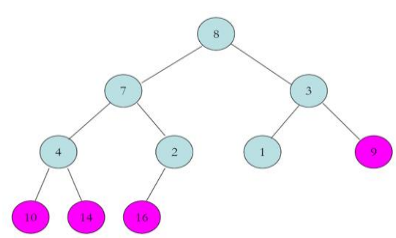
117



118



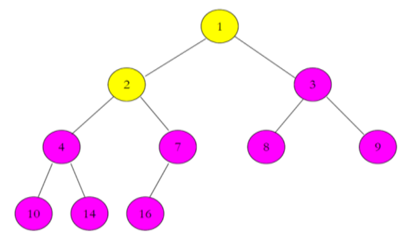
119



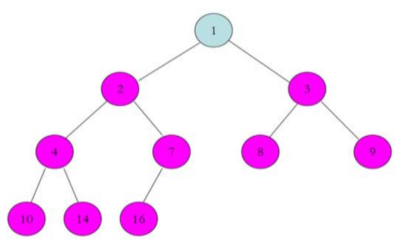
heap-size ← heap-size − 1
MAX-HEAPIFY(A, 1)

120

- ....



Swap $A[1] \leftrightarrow A[i]$



heap-size ← heap-size – 1
MAX-HEAPIFY(A, 1)

HEAPSORT(*A*)

1. BUILD-MAX-HEAP(*A*) — $O(n)$
2. **for** i ← length[*A*] **downto** 2
3. **do** exchange *A*[1] ↔ *A*[i] — $O(\lg n)$
4. MAX-HEAPIFY(*A*, i - 1)

n-1 x

- čas time: $O(n \log n)$

Usporadúvanie haldou (1)

```
procedure posun(var i: Integer; m:
  Integer)
begin
  // ak existuje syn, nastaví sa na väčšieho z nich
  if 2*i+1 <= m then // ak aspoň jeden syn existuje
  begin
     i := 2*i+1; // i je ľavý syn
     if (i < m) and (p[i+1] > p[i]) then
          i := i+1; // i je teraz už pravý syn - bol väčší
     Inc(pocet);
  end;
end;
```


Usporadúvanie haldou (2)

```
procedure vyhalduj(k, m: Integer)
var
  i: Integer;
begin
  // predpokladáme, že od k+1 do m to už halda je - pridáme k-ty
  i := k;
  posun(i, m);
  while p[k] < p[i] do
  begin
      vymen(k, i);
      k := i;
      posun(i, m); // i je nový väčší syn
      Inc(pocet);
  end;
  Inc(pocet);
end;
```

Usporadúvanie haldou (3)

```
procedure vytvor_haldu
   var i: Integer;
begin
   for i := (N-1) div 2 downto 0 do
        vyhalduj(i, N-1);
end;

Procedure HEAPSORT
var posledny: Integer;
begin
   vytvor_haldu;
   posledny := N-1; // ber postupne všetky prvky

   while posledny > 0 do
   begin
        vymen(0, posledny); // vymeň koreň s posledným prvkom
        Dec(posledny); // zmenši rozmer stromu
        vyhalduj(0, posledny); // oprav strom - koreň je teraz asi zlý
   end;
end;
```


Usporadúvanie haldou

- procedúra vyhalduj vytvára haldu v poli medzi zadanými dvoma indexmi poľa
- po jej skončení je časť poľa K až M haldou, pričom procedúra predpokladá, že časť K+1 až M je halda, ale tým, že sme pridali prvok na miesto K, mohla sa halda K až M pokaziť
- preto je potrebné pole znovu „vyhaldovať", t.j. zabezpečiť, aby aj pre K platilo, že má oboch synov menších ako on sám (ak to tak nie je, treba ho zrejme vymeniť s väčším zo synov a postup opakovať pre tohto syna a príslušný podstrom)
- pomocná procedúra posun posúva prvý parameter na väčšieho syna

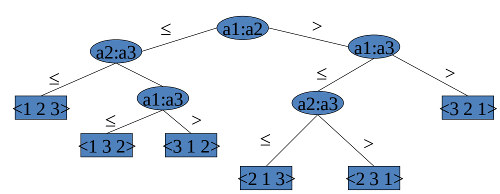
Usporadúvanie haldou - zložitosť

- heap sort má rovnaké časové zložitosti ako merge sort, t.j. garantuje zložitosť **O(n log n)**
- výhodou oproti merge sort je tzv. in-line pamäťová zložitosť – O(1), t.j. nepotrebuje dodatočnú pamäť, pracuje priamo nad pamäťou vstupnej postupnosti – merge sort O(n)
- nevýhody oproti merge sort:
  - Heap sort vyžaduje priamy prístup k údajom
  - Sekvenčný prístup merge sort-u môže lepšie využiť potenciál pamäte cache
  - Heap sort sa nedá paralelizovať
  - Heap sort nie je stabilný

dolné ohraničenie na usporadúvania porovnávaním

model rozhodovacieho stromu

predstavuje porovnania, ktoré vykoná usporadúvací algoritmus nad vstupom daného rozsahu



rozhodovacie stromy

- môžu modelovať usporadúvania porovávaním
- pre príslušný algoritmus:
  - jeden strom pre každé *n*
  - cesty v strome sú všetky možné stopy výpočtu
- *aká je asymptotická výška ľubovoľného rozhodovacieho stromu pre usporiadanie n prvkov?*
- $\Omega(n \lg n)$

Lineárne usporadúvanie

- porovnávacie algoritmy maximálne O(n log n)
- algoritmy s lineárnou časovou zložitosťou O(n) používajú iné operácie ako porovnávanie na usporiadanie postupností
- použitie distribuovaných algoritmov – algoritmy, kde údaje zo vstupu sú rozdelené do viacerých prechodných štruktúr, ktoré sa potom zhromaždia a umiestnia na výstupe

Lineárne usporadúvanie

- Výhody a nevýhody oproti porovnávacím algoritmom:
  - lepšia časová zložitosť
  - horšia pamäťová zložitosť – nedá sa pracovať na mieste (in-situ)
  - predpokladá, že vstupné údaje sú z nejakého intervalu

133

Usporadúvanie spočítavaním (counting sort)

- určuje počet prvkov menších ako prvok *x*, pomocou čoho zistí správnu pozíciu prvku *x* vo vstupnom poli
- predpokladá, že každý prvok z *n* vstupných prvkov je z nejakého intervalu 1..*k*.
- ak má byť efektívny, tak musí platiť, že *k* nie je oveľa väčšie ako *n*
- vhodný na použitie ak *k* je malé a kľúče sa často opakujú

- časová zložitosť O(n+k)
  - ak *k* patrí do O(n), tak beží v čase O(n). Ak *k* je oveľa väčšie ako *n*, napr. ak *k* patrí do O(n^2), tak sa berie O(*k*).

134

Usporadúvanie spočítavaním

Andy	Bubo	Cica	Dora	Edita	Fero	Gugo	Hugo	Ivo	Ján	Kika

n prvkov poľa, k rôznych kľúčov

135

spočítaj počet výskytov každého kľúča

Andy	Bubo	Cica	Dora	Edita	Fero	Gugo	Hugo	Ivo	Ján	Kika

jeden prechod cez pole, čas O(n)

	4
	1
	3
	3

136

transformuj na kumulatívne počty

Andy	Bubo	Cica	Dora	Edita	Fero	Gugo	Hugo	Ivo	Ján	Kika

jeden prechod cez tabuľku počtov, čas O(k)

	4
	5
	8
	11

137

transformuj na kumulatívne počty

Andy	Bubo	Cica	Dora	Edita	Fero	Gugo	Hugo	Ivo	Ján	Kika

jeden prechod cez tabuľku počtov, čas O(k)

	4
	5
	8
	11

prečo kumulatívne počty?
k.p. zelených kľúčov mi určuje, na ktoré miesto môžem bezpečne zapísať prvok so zeleným kľúčom
(a bude práve dosť miesta na všetky prvky, ktoré majú kľúč menší alebo rovný)

138

kopíruj do druhého poľa

Andy	Bubo	Cica	Dora	Edita	Fero	Gugo	Hugo	Ivo	Ján	Kika

				Kika						

	4
	5->4
	8
	11

jeden prechod cez pole, začínajúc vpravo. v každom kroku sa dekrementuje počítadlo v tabuľke počtov. čas O(n).

139

Andy	Bubo	Cica	Dora	Edita	Fero	Gugo	Hugo	Ivo	Ján	Kika

			Ján	Kika						

	4→3
	4
	8
	11

140

Andy	Bubo	Cica	Dora	Edita	Fero	Gugo	Hugo	Ivo	Ján	Kika

			Ján	Kika			Ivo			

	3
	4
	8→7
	11

141

Andy	Bubo	Cica	Dora	Edita	Fero	Gugo	Hugo	Ivo	Ján	Kika

		Hugo	Ján	Kika			Ivo			

	3→2
	4
	7
	11

142

Andy	Bubo	Cica	Dora	Edita	Fero	Gugo	Hugo	Ivo	Ján	Kika

		Hugo	Ján	Kika		Gugo	Ivo			

	2
	4
	7→6
	11

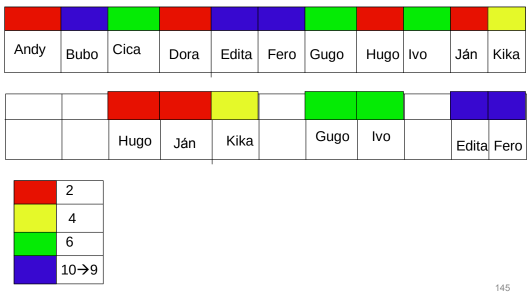
143

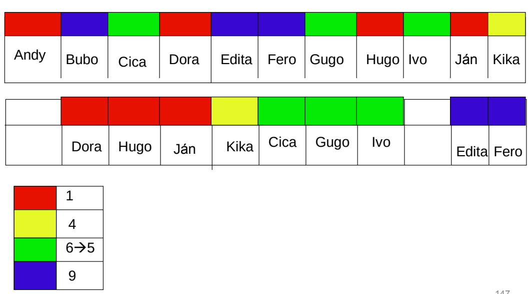
Andy	Bubo	Cica	Dora	Edita	Fero	Gugo	Hugo	Ivo	Ján	Kika

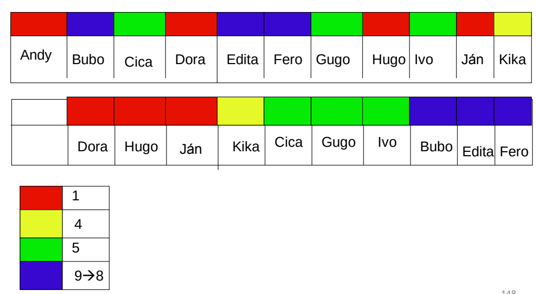
		Hugo	Ján	Kika		Gugo	Ivo			Fero

	2
	4
	6
	11→10

144







algoritmus je stabilný. celkový čas je O(n+k). ak k je konštantné a malé oproti n, O(n).

Usporadúvanie spočítavaním

- Pracuje s tromi poliami:
  - Pole *A[]* obsahuje údaje, ktoré sa majú usporiadať – veľkosť poľa je *n*
  - Pole *B[]* obsahuje konečný usporiadaný zoznam údajov – veľkosť poľa je *n*
  - Pole *C[]* je použité na počítanie počtu prvkov – veľkosť poľa je *k*

Usporadúvanie spočítavaním

- Fázy algoritmu:
  - prvý cyklus inicializuje pole *C[]* na nulové hodnoty
  - druhý cyklus inkrementuje hodnoty v *C[]* podľa početnosti ich výskytu v poli *A[]* – početnosť sa zapíše do indexu, ktorý zodpovedá hodnote prvku.
  - tretí cyklus pripočíta ku každej hodnote poľa *C[]* kumulatívny súčet predchádzajúcich hodnôt tohto poľa
  - štvrtý cyklus vypíše usporiadané hodnoty do poľa *B[]* – hodnota na indexe i poľa *C[]* určuje index v poli *B[]*, do ktorého sa zapíše prvok, ktorý sa rovná indexu *i*.

Usporadúvanie spočítavaním

vstup: A [1 .. n],
$A[J] \in \{1,2, \ldots, k\}$

výstup: B [1 .. n],
usporiadaný

používa C [1 .. k],
pomocná pamäť

Counting-Sort(*A*, *B*, *k*)
1. **for** $i \leftarrow 1$ **to** k
2. **do** $C[i] \leftarrow 0$
3. **for** $j \leftarrow 1$ **to** $length[A]$
4. **do** $C[A[j]] \leftarrow C[A[j]] + 1$
5. **for** $i \leftarrow 2$ **to** k
6. **do** $C[i] \leftarrow C[i] + C[i-1]$
7. **for** $j \leftarrow length[A]$ **down** 1
8. **do** $B[C[A[j]]] \leftarrow A[j]$
9. $C[A[j]] \leftarrow C[A[j]] - 1$

Analýza:

	1	2	3	4	5	6
A	4	1	3	4	3	4

k = 4, dĺžka (*length)* = 6

C	0	0	0	0

po riadkoch 1-2

C	1	0	2	3

po riadkoch 3-4

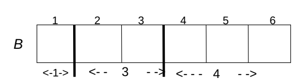
C	1	1	3	6

po riadkoch 5-6

Counting-Sort(*A*, *B*, *k*)
1. **for** $i \leftarrow 1$ **to** k
2. **do** $C[i] \leftarrow 0$
3. **for** $j \leftarrow 1$ **to** $length[A]$
4. **do** $C[A[j]] \leftarrow C[A[j]] + 1$
5. **for** $i \leftarrow 2$ **to** k
6. **do** $C[i] \leftarrow C[i] + C[i-1]$

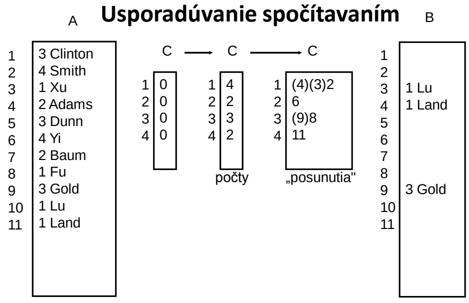
	1	2	3	4	5	6
A	4	1	3	4	3	4

7. **for** $j \leftarrow length[A]$ **down** 1
8. **do** $B[C[A[j]]] \leftarrow A[j]$
9. $C[A[j]] \leftarrow C[A[j]] - 1$



C	1	1	3	6

Usporadúvanie spočítavaním



Usporadúvanie spočítavaním

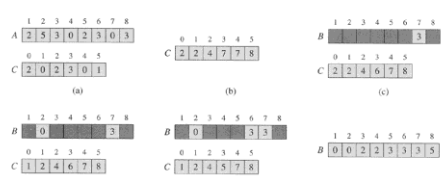
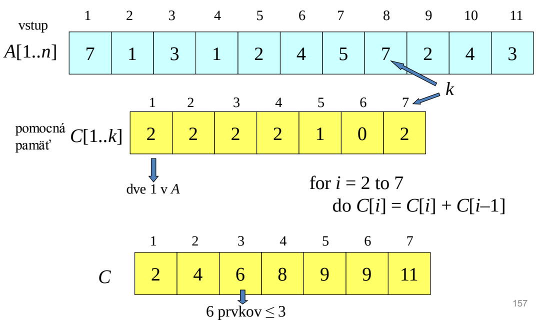
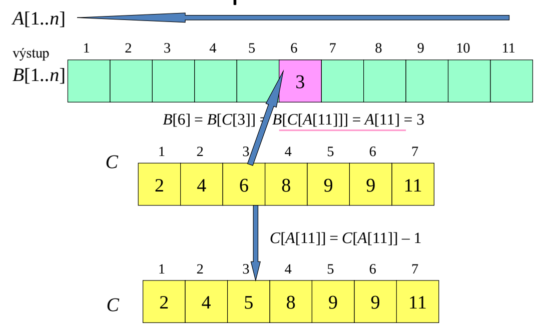


Figure 8.2 The operation of COUNTING-SORT on an input array $A[1..8]$, where each element of A is a nonnegative integer no larger than $k = 5$. **(a)** The array A and the auxiliary array C after line 4. **(b)** The array C after line 7. **(c)–(e)** The output array B and the auxiliary array C after one, two, and three iterations of the loop in lines 9–11, respectively. Only the lightly shaded elements of array B have been filled in. **(f)** The final sorted output array B.

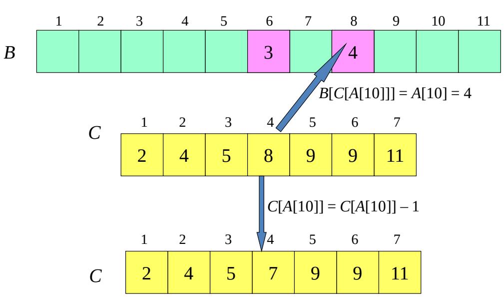
príklad



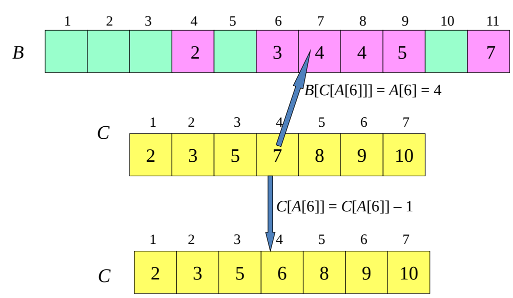
príklad



príklad



príklad



usporadúvanie spočítavaním

```
CountingSort(A, B, k)
    for i=1 to k
        C[i]= 0;
    for j=1 to n
        C[A[j]] += 1;
    for i=2 to k
        C[i] = C[i] + C[i-1];
    for j=n downto 1
        B[C[A[j]]] = A[j];
        C[A[j]] -= 1;
```


usporadúvanie spočítavaním

```
CountingSort(A, B, k)
    for i=1 to k                 čas O(k)
        C[i]= 0;
    for j=1 to n
        C[A[j]] += 1;
    for i=2 to k
        C[i] = C[i] + C[i-1];    čas O(n)
    for j=n downto 1
        B[C[A[j]]] = A[j];
        C[A[j]] -= 1;
```

usporadúvanie spočítavaním

- celkový čas: O(n + k)
  - zvyčajne, k = O(n)
  - teda celkovo O(n)
- čiže významne lepšie než $\Omega(n \lg n)$!
  - lebo toto nie je porovnávací algoritmus
  - je stabilný

163

usporadúvanie spočítavaním

- prečo vlastne nepoužívame vždy usporadúvanie spočítavaním?
- lebo závisí od rozsahu prvkov k
- *mohli by sme použiť usporadúvanie spočítavaním na usporiadanie celých čísiel zapísateľných do 32 bitov? Prečo áno/nie?*
- nie, k je príliš veľké (2^{32} = 4,294,967,296)

164

Herman Hollerith

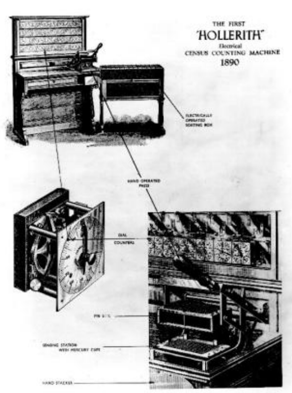
- (1860-1929)
- 1880 spracovanie sčítania ľudu USA trvalo skoro 10 rokov (robí sa každých 10 rokov).
- ako prednášateľ na MIT, Hollerith navrhol prototyp strojov na spracovanie diernych štítkov.
- pomocou jeho strojov sa doba spracovania ďalšieho sčítania ľudu v 1890 skrátila na 6 týždňov.
- založil firmu Tabulating Machine Company v 1911, ktorá sa spojila sa ďalšími firmami v 1924 - vznikla International Business Machines.

dierny štítok

- dierny štítok - obsahuje záznam údajov.
- dierka = hodnota.

kópia dierneho štítku použitého na spracovanie sčítania ľudu 1900 U.S.A. [Howells 2000]

Hollerithov tabulačný systém



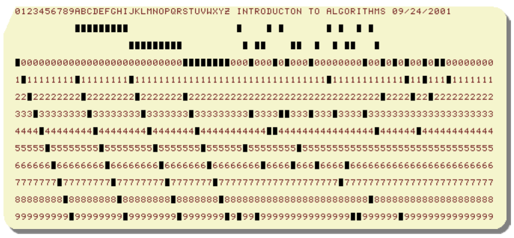
obrázok z [Howells 2000].

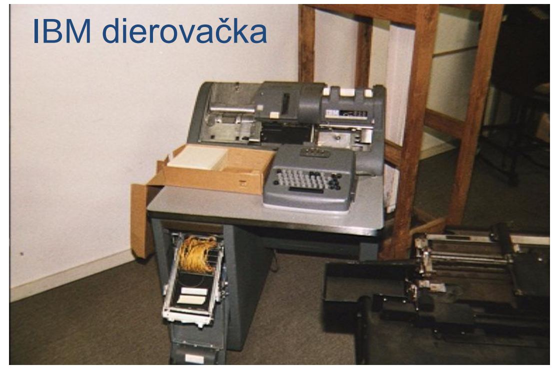
Ako pôvodne zbohatla IBM?

- začiatkom 20. storočia vyrobila čítačky diernych štítkov pre spracovanie údajov zo sčítania ľudu.
- dierny štítok má 80 stĺpcov, stĺpec má 12 miest pre dierky. pomocou dierovačiek diernych štítkov sa na štítky zapisovali údaje.
- triedičky mali 12 priečinkov.
- základná myšlienka: začni triediť podľa najnižšieho rádu.
- voči dovtedajšiemu spôsobu dokázala proces rádovo zrýchliť (1880=7 rokov)

Linear Sorts 168

dierny štítok





pôvod radixového usporadúvania

- Hollerithov pôvodný patent z r. 1889 naznačuje radixové usporadúvanie od najvyššieho rádu:
  - *„Najzložitejšie kombinácie [záznamy, pozostávajúce z položiek údajov] sa dajú ľahko spočítať [usporiadať] s relatívne malým počtom počítadiel tak, že sa najprv štítky zoradia podľa prvých položiek, vstupujúcich do kombinácií, potom sa preusporiadajú podľa druhej položky, vstupujúcej do kombinácie [záznamu] a tak ďalej a nakoniec spočítaním na tých málo počítadlách poslednú položku kombinácie pre každú skupinu [súbor] štítkov.„*
- Úprava algoritmu na usporadúvanie od najnižšieho rádu sa zdá byť výsledok ľudovej tvorivosti operátorov sčítačiek.
- poznámka: a potom, že algoritmus nie je patentovateľný! ak sa patentuje zariadenie, môže opis patentu obsahovať aj algoritmus.

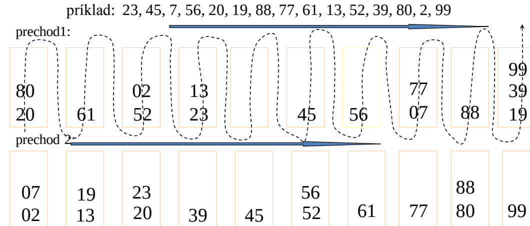
radixové usporadúvanie

RADIX-SORT(A, d)
  for i ← 1 **to** d
    do stabilné usporadúvanie(A) podľa číslice i

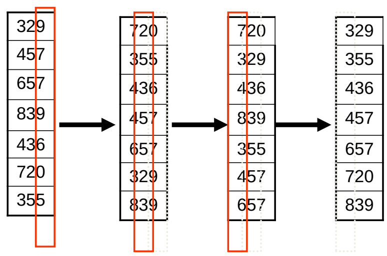
radixové usporadúvanie - príklad

usporiada množinu čísiel vo viacerých prechodoch, začínajúc od číslic najnižšieho (jednotkového) rádu, potom usporiada podľa číslic najbližšieho vyššieho (desiatkového) rádu atď.

príklad: 23, 45, 7, 56, 20, 19, 88, 77, 61, 13, 52, 39, 80, 2, 99



radixové usporadúvanie – ďalší príklad



radixové usporadúvanie

- číslo zapísané v pozičnej sústave so základom k
  - hodnota = $x_{d-1}k^{d-1} + x_{d-2}k^{d-2} + \ldots + x_2k^2 + x_1k^1 + x_0k^0$
- Radixové usporadúvanie neporovnáva dva kľúče, ale spracúva a porovnáva časti kľúčov
- Kľúče považuje za čísla zapísané v číselnej sústave so základom *k* (radix, koreň), pracuje s jednotlivými číslicami
- Dokáže usporadúvať čísla, znakové reťazce, dáta, ... (počítače reprezentujú všetky údaje ako postupnosti 1 a 0 – binárna sústava => 2 je základ)
  - problém: usporiadať 1 millión 64-bitových čísiel
  - 64 prechodov cez milión čísiel?
    - čo tak interpretovať ich ako čísla v sústave so základom (radixom) 2^{16}, budú to najviac 4-miestne čísla
    - vtedy radixové usporadúvanie usporiada len v 4 prechodoch!

radixové usporadúvanie

- LSD Radix sort (least significant digit) – usporadúvanie podľa číslic postupuje od poslednej číslice (s najmenšou váhou) k prvej číslici (s najväčšou váhou)– stabilný.
- MSD Radix sort – od prvej číslice k poslednej – lexikografické usporiadanie – nestabilný
- Je dôležité na samotné usporadúvanie podľa jednotlivých číslic použiť nejaký stabilný algoritmus, aby sa nemenilo poradie prvkov s rovnakými číslicami jednej váhy pri usporadúvaní podľa inej váhy.
- Keďže počet možných číslic (ak *k*=10) je len 10, tak na usporiadanie podľa nich je výhodné použiť usporadúvanie spočítavaním.

radixové usporadúvanie

- *ako preukázať, že naozaj usporadúva?*
  - predpokladajme, že postupnosť je podľa číslic nižších rádov {j: j<i} usporiadaná
  - treba ukázať, že usporiadanie podľa nasledujúcej číslice rádu i zanechá postupnosť usporiadanú (podľa nižších rádov ale už vrátane i)
    - ak sú dve číslice na i-tom ráde (mieste odspodu) rôzne, usporiadanie dvoch čísiel podľa tohto rádu je správne (je vyšší rád než všetky, podľa ktorých sa usporadúvalo doteraz, keďže sa ide od najnižšieho, nižšie rády sú irrelevantné)
    - ak sú dve číslice na i-tom ráde (mieste odspodu) rovnaké, čísla sú už usporiadané podľa nižších rádov. ak používame stabilný algoritmus, čísla zostanú v správnom poradí

radixové usporadúvanie

- *aký algoritmus použiť na usporiadanie podľa číslic?*
- ponúka sa usporadúvanie spočítavaním:
  - usporiada *n* čísiel podľa číslic v sústave so základom *k,* tj rozsah číslic je 0..*k-1*
  - čas: O(*n* + *k*)
- každý prechod cez *n* *d*-miestnych čísiel (s *d číslicami)* si vyžiada čas O(*n*+*k*), takže celkový čas je O(*dn*+*dk*)
  - ak *d* je konštantné a *k*=O(*n*), vyžiada si čas O(*n*)

radixové usporadúvanie

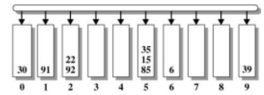
- r.u. založené na usporadúvaním spočítavaním je
  - rýchle
  - asymptoticky rýchle (t.j. O(*n*))
  - ľahko sa programuje

Iný príklad

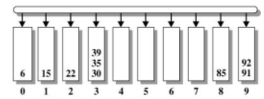
now	sob	tag	ace
for	nob	ace	bet
tip	ace	bet	dim
ilk	tag	dim	for
dim	ilk	tip	hut
tag	dim	sky	ilk
jot	tip	ilk	jot
sob	for	sob	nob
nob	jot	nob	now
sky	hut	for	sky
hut	bet	jot	sob
ace	now	now	tag
bet	sky	hut	tip

príklad

- vstup: [91, 6, 85, 15, 92, 35, 30, 22, 39]



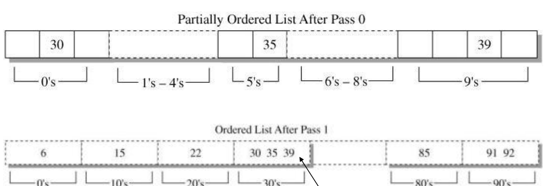
po prechode 0: [30, 91, 92, 22, 85, 15, 35, 6, 39]



po prechode 1: [6, 15, 22, 30, 35, 39, 85, 91, 92]

príklad pokr.

pôvodná postupnosť: {91, 6, 85, 15, 92, 35, 30, 22, 39}



čísla s rovnakým počtom desiatok. v poradí podľa jednotiek

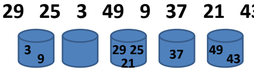
Vedierkové usporadúvanie (Bucket sort)

- Predpokladá, že vstup je akoby generovaný náhodným procesom, ktorý prvky distribuuje rovnomerne na celom intervale.
- Rozdelí interval na *n* rovnako veľkých disjunktných podintervalov (vedierok - bucketov) a potom do nich rozmiestni vstupné čísla
- osobitne v každom vedierku sa potom tieto čísla usporiadajú.

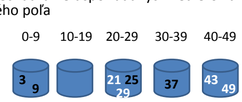
Vedierkové usporadúvanie

- Vytvoria sa prázdne vedierka veľkosti M/n (M – maximálna hodnota vstupného poľa, n – počet prvkov vstupného poľa)
- Rozptýlenie – prechádzanie vstupným poľom a rozmniestnenie každého prvku do prislúchajúceho vedierka

29 25 3 49 9 37 21 43



- Usporiadanie naplnených vedierok
- Zreťazenie vedierok – postupné prechádzanie usporiadaných vedierok a presúvanie prvkov späť do vstupného poľa

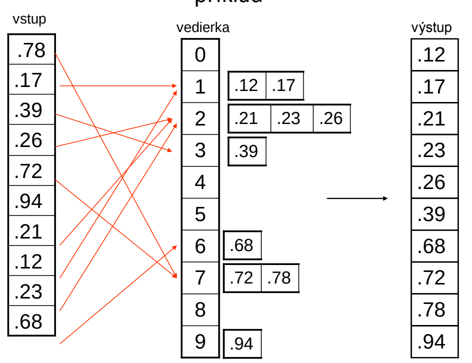


3 9 21 25 29 37 43 49

Vedierkové usporadúvanie

```
BUCKET-SORT(A)
    n ← length(A)
    for i ← 0 to n
        do vlož A[i] do zoznamu B[floor(n*A[i])]
    for i ← 0 to n –1
        do Insertion-Sort(B[i])
    zreťaz zoznamy B[0], B[1], … B[n –1] v tomto poradí
```

príklad



vedierko i obsahuje hodnoty z polouzavretého intervalu [i/10, (i + 1)/10).

187

Vedierkové usporadúvanie - zložitosť

- jednotlivé vedierka väčšinou predstavujú spájaný zoznam, do ktorého sa na správne miesto presúvajú prvky zo vstupného poľa (insert sort)
- činnosti ako vytvorenie vedierok, určenie prislúchajúceho vedierka, presunutie prvku do vedierka a zreťazenie vedierok do výslednej postupnosti trvajú **O(n)**
- časové usporiadanie prvkov vo vedierkach insert sortom trvá **O(n^2)**

188

Vedierkové usporadúvanie - zložitosť

- výsledná časová zložitosť závisí od rozloženia prvkov vo vedierkach. Ak sú prvky rozmiestnené nerovnomerne a v niektorých vedierkach ich je veľmi veľa, tak časová zložitosť insert sortu O(n^2) prevažuje nad lineárnou zložitosťou a predstavuje výslednú zložitosť celého usporadúvania
- takýto stav sa môže vyskytnúť ak rozsah prvkov *m* je oveľa väčší ako ich počet
- preto sa niekedy celková zložitosť značí podobne ako pri counting sorte O(n+m). Ak m=O(n), tak výsledná časová zložitosť je O(n)
- ak sa počet vedierok rovná počtu vstupných prvkov, tak v priemere to vychádza na jeden prvok v každom vedierku, a preto sa za priemernú zložitosť berie O(n)

189

Porovnanie jednotlivých metód

190

Podľa časovej zložitosti

	priem.	*najhoršia*
Vkladaním	0(N^2)	0(N^2)
Výmenou	0(N^2)	0(N^2)
Výberom	0(N^2)	0(N^2)
Shellovo	0(N log^{2}N)	0(N^2)
QuickSort	0(N log N)	0(N^2)
MergeSort	0(N log N)	0(N log N)
HeapSort	0(N log N)	0(N log N)
CountingSort	0(N + M)	0(N + M)
RadixSort	0(k.N)	0(k.N)
BucketSort	0(N)	0(N^2)

191

Podľa časovej zložitosti a stability

	pam. zložitosť	*stabilný*
Vkladaním	0(1)	áno
Výmenou	0(1)	áno
Výberom	0(1)	áno
Shellovo	0(1)	nie
QuickSort	0(log N)	nie
MergeSort	0(N)	áno
HeapSort	0(1)	nie
CountingSort	0(N + M)	áno
RadixSort	0(N)	áno(LSD)
BucketSort	0(N)	áno

192

Porovnanie jednotlivých metód

- aj napriek tomu, že distribuované algoritmy usporadúvania majú v priemere lineárnu zložitosť, tak kvôli vyššej réžii niektorých krokov (extrahovanie cifier, kopírovanie polí) sú v praxi väčšinou pomalšie ako porovnávacie algoritmy usporadúvania s priemernou časovou zložitosťou O(n log n)

193

Vyhľadávanie (druhé kolo, trochu náročnejšie)

194

nájdenie k-teho najmenšieho prvku

problém: Majme pole A[1..n] n čísel a celé číslo k (1<=k<=n). Treba nájsť k-te najmenšie číslo v A.

Napríklad

- pre k=1 ide o nájdenie najmenšieho prvku,
- pre k=n ide o nájdenie najväčšieho prvku,
- ak n je nepárne, k=(n+1)/2 dá medián
- ak n je párne, podľa dohody je medián niečo medzi prípadmi k=floor((n+1)/2) a k=ceiling((n+1)/2)

195

nájdenie k-teho najmenšieho prvku

- prvý nápad: usporiadať pole A a vybrať A[k]
  - usporiadať vieme na mieste O(n log n). Výber A[k] je O(1). Spolu O(n log n).
- Dá sa to rýchlejšie?
  - Ak máme už dané k, tak usporiadaním sa urobilo viac roboty, než je v skutočnosti treba na určenie k-teho najmenšieho prvku. Prečo?
  - Ak by k nebolo vopred dané, tak by práve usporiadané pole dávalo k-ty najmenší prvok pre ľubovoľné k.

196

nájdenie k-teho najmenšieho prvku

- druhý nápad: nájsť najmenší prvok v poli A a odstrániť ho. Pokračovať nájdením najmenšieho prvku vo zvyšku poľa A, odstránením atď. Opakovať k-krát.
  - násť minimum vieme na mieste O(n). Odstrániť ho je O(1). Spolu O(k*n).
- je to rýchlejšie?
  - závisí od porovnania k a log(n). pre malé k je prvý lepší, pre veľké k je druhý lepší.

197

nájdenie k-teho najmenšieho prvku

- Dá sa to rýchlejšie?
  - všimnime si, že v prvom aj druhom prípade dostaneme k najmenších prvkov usporiadaných.
  - Ale to (usporiadanie) nepotrebujeme! Robíme robotu navyše.
  - Výzva je urobiť to v čase O(n).

198

Pripomienka: Rýchle usporadúvanie

```
procedure rychle-usporaduvanie(A, lavy, pravy)
  if lavy < pravy
     ipivot := dajipivot(A,lavy, pravy)
      vymen(A[ipivot], A[pravy])
      medza := rozclenenie(A, lavy, pravy)
     rychle-usporaduvanie(A, lavy, medza)
     rychle-usporaduvanie(A, medza + 1, pravy)
  end
end
```


Rýchly výber (Quick Select) k-teho najmenšieho prvku

```
procedure rychly-vyber(A, lavy, pravy, k)
  if lavy >= pravy return A[lavy]
  ipivot := dajipivot(A,lavy, pravy)
  vymen(A[ipivot], A[pravy])
  medza := rozclenenie(A, lavy, pravy)
  if k < medza
     rychly-vyber(A, lavy, medza−1, k)
  else if k > medza
     rychly-vyber(A, medza+1, pravy, k-medza)
  else return A[medza]
end
```


myšlienka pôvodného Hoarovho algoritmu 65

```
Find(S, k)
// S je množina n reálnych čísel, n=|S|
if n=1 then return prvok S //je jediný
else // n>1
  zvoľ pivot x rovnomerne náhodne z S
  urči dve množiny S< = {s ∈ S : s < x} a
                   S> = {s ∈ S : s > x}
  if |S<| + 1 = k then return x
      else if |S<| + 1 > k then Find(S< ,k)
             else // |S<| + 1 < k
                    Find(S> ,k - |S<| + 1)
```


C. A. R. Hoare. 1961. Algorithm 65: find. *Commun. ACM* 4, 7 (July 1961), 321-322. DOI=10.1145/366622.366647 http://doi.acm.org/10.1145/366622.366647

ALGORITHM 65
FIND
C. A. R. HOARE
Elliott Brothers Ltd., Borehamwood, Hertfordshire, Eng.

```
procedure  find (A,M,N,K);  value M,N,K;
              array A;  integer M,N,K;
comment  Find will assign to A [K] the value which it would
have if the array A [M:N] had been sorted. The array A will be
partly sorted, and subsequent entries will be faster than the first;

begin     integer I,J;
          if M < N then begin partition (A, M, N, I, J);
                              if K≦I then find (A,M,I,K)
                              else if J≦K then find (A,J,N,K)
                              end
end       find
```


nájdenie k-teho najmenšieho prvku

- Blum, Floyd, Pratt, Rivest a Tarjan, 1973: algoritmus s mediánom mediánov ako pivotom
- predpoklad: všetky prvky v A sú rôzne.
- čo ak nie sú? nevadí, algoritmus je použiteľný, len treba upraviť prvky, napríklad takto:
  - každý prvok rozšírime na usporiadanú dvojicu:
    - A[j], 1<=j<=n -> (A[j], j)
  - relácia usporiadania bude:
    - $(x_1, j_1) < (x_2, j_2)$ ak buď $x_1 < x_2$ alebo $x_1=x_2$ a $j_1<j_2$
    - tento druh usporiadania sa nazýva lexikografické

nájdenie k-teho najmenšieho prvku s mediánom mediánov ako pivotom

```
Select (A, k)
1. x = median(A) // akurát, že zatiaľ nevieme ako v
   O(n)
2. rozčleň A podľa pivota x. Nech je m-1 prvkov
   takých, že A[i]<x. Potom bude A[m]=x a n-m prvkov
   bude takých, že A[i]>x.
3. if k=m then return x
            else if k<m then Select (A[1..m-1], k)
                  else Select (A[m+1..n], k-m)
```

zložitosť Select (A, k)

```
Select (A, k)
1. x = median(A) // akurát, že zatiaľ nevieme ako v O(n)
2. rozčleň A podľa pivota x. Nech je m-1 prvkov takých, že
   A[i]<x. Potom bude A[m]=x a n-m prvkov bude takých, že
   A[i]>x.
3. if k=m then return x
            else if k<m then Select (A[1..m-1], k)
                    else Select (A[m+1..n], k-m)
```

zložitosť:

$T_{select}(n)=T_{select}(n/2)+n+T_{median}(A)$

- predpokladajme, že $T_{median}(A)$ je $O(n)$ a preto

  $T_{select}(n)=T_{select}(n/2)+n$
- riešenie je 2n, t.j. je $O(n)$

približný medián

- predpokladali sme, že $T_{median}(A)$ je $O(n)$
- takže nepotrebujeme vlastne presný medián. Stačí taký, ktorý zaručene zabezpečí lineárnu zložitosť. Uvažujme približný medián, ktorý je zaručene väčší než 3/10 všetkých prvkov a menší než 3/10 všetkých prvkov. Presnejšie, uvažujme približný medián taký, že je x-tý najmenší zo všetkých prvkov v A a platí

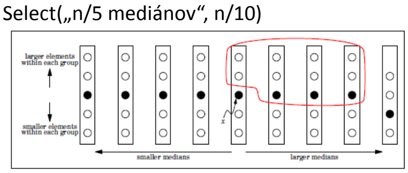
  $3n/10 \le x \le 7n/10$
- najhorší prípad je, keď sa rekurzívne volanie vykoná nad $7n/10$ prvkami. Celkový čas je

  $T_{select}(n) = T_{select}(7n/10)+n$
- riešenie tejto rekurentnej rovnice je $O(n)$.
- „ stačí“ vymyslieť, ako vypočítať približný medián.

medián mediánov

median(A)

1. rozdeľ n prvkov v poli A do skupín po piatich a možno jednej zvyškovej
2. nájdi medián každej skupiny (usporiadaním alebo natvrdo tretí najmenší). dostaneš n/5 mediánov.
3. Select(„n/5 mediánov“, n/10)



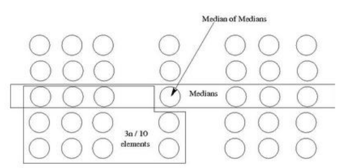
medián mediánov

zložitosť:

- usporiadať postupnosť 5 čísel vieme v čase 5*log(5). máme n/5 skupín čísel. (n/5)*5*log(5)= n*log(5). To je $O(n)$
- o mediáne tých n/5 mediánov platí:

  je väčší než n/10 mediánov. Každý z tých n/10 mediánov je väčší než 2 prvky vo svojej skupine. takže je celkovo väčší než 3n/10 prvkov.

  podobne je menší než 3n/10 prvkov.



nájdenie k-teho najmenšieho prvku s mediánom mediánov ako pivotom

Select (A, k)

1. rozdeľ n prvkov do skupín po piatich a možno jednej zvyškovej
2. nájdi medián každej skupiny (usporiadaním alebo natvrdo tretí najmenší)
3. x = Select („ceiling(n/5)) mediánov“, ceiling(n/5)/2)
4. rozčleň A podľa pivota x. Nech je m-1 prvkov takých, že A[i]<x. Potom bude A[m]=x a n-m prvkov bude takých, že A[i]>x.
5. if k=m then return x

   else if k<m then Select (A[1..m-1], k)

   else Select (A[m+1..n], k-m)

nájdenie k-teho najmenšieho prvku s mediánom mediánov ako pivotom

Select (A, k)

1. rozdeľ n prvkov do skupín po piatich a možno jednej zvyškovej
2. nájdi medián každej skupiny (usporiadaním alebo natvrdo tretí najmenší)
3. x = Select („ceiling(n/5)) mediánov“, ceiling(n/5)/2)
4. rozčleň A podľa pivota x. Nech je m-1 prvkov takých, že A[i]<x. Potom bude A[m]=x a n-m prvkov bude takých, že A[i]>x.
5. if k=m then return x

   else if k<m then Select (A[1..m-1], k)

   else Select (A[m+1..n], k-m)

zložitosť:

$T_{select}(n) = T_{select}(n/5) + T_{select}(7n/10)+n$

riešenie tejto rekurentnej rovnice je $O(n)$

počet prvkov v skupine môže byť ľubovoľne väčší než 5

211