Metode de sortare interna

Metode directe de sortare. Metode avansate de sortare

Problema sortarii

- Relatie de ordine liniara intre cheile obiectelor pe care dorim sa le sortam
- Avem o secventa de obiecte (inregistrari, elemente) $r_1, r_2, ..., r_n$ cu cheile $k_1, k_2, ..., k_n$, respectiv, trebuie sa re-aranjam obiectele in ordinea $r_{i1}, r_{i2}, ..., r_{in,}$ astfel incat $k_{i_1} \leq k_{i_2} \leq \ldots \leq k_{i_n}$
  - i.e. sa generam o permutare crescatoare
- Cum evaluam timpul de rulare
  - Numar de pasi pt a sorta n elemente
  - Numar de comparatii
  - Numar de “mutari” (atribuiri)

Algoritmi de sortare - dimensiuni de analiza

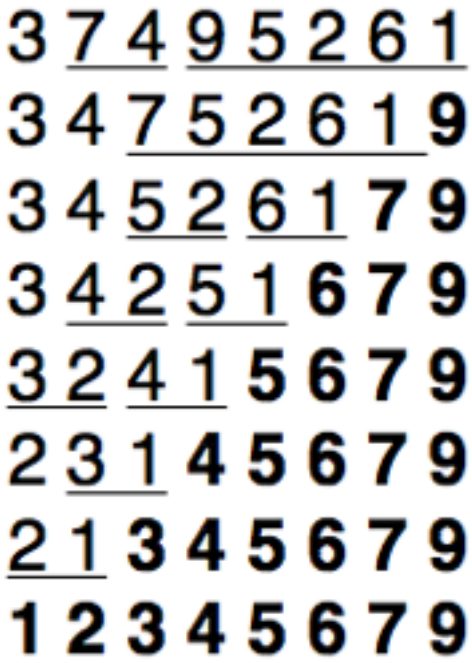
- Complexitatea computationala (defav, mediu, fav)
  - a atribuirilor
- Memoria aditionala utilizata
- Stabilitate
  - pastreaza ordinea relativa a elementelor egale
- Daca e bazata pe comparatii sau nu
- Strategia generala
  - insertie, interschimbare, selectie, interclasare
- Adaptabilitate

Sortarea prin interschimbare (Bubblesort)

```
n = length(A)
 repeat
   swapped = false
   for i = 1 to n-1 inclusive do
     /* if this pair is out of order */
     if A[i-1] > A[i] then
       /* swap them and remember something changed */
       swap( A[i-1], A[i] )
       swapped = true
     end if
   end for
   n=n-1
 until not swapped
```

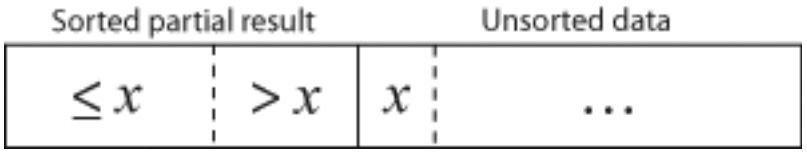
Sortarea prin interschimbare (Bubblesort)

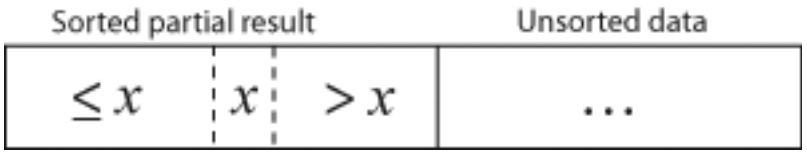
- Exemplu
- *Rabbits and turtles*
- Versiuni imbunatatite
  - cocktail sort
  - comb sort
- Complexitate?
  - favorabil, defavorabil
- Stabilitate?
- Memorie?



Sortarea prin inserare

```
for i = 1 to length(A)
    x = A[i]
    j = i - 1
    while j >= 0 and A[j] > x
        A[j+1] = A[j]
        j = j - 1
    end while
    A[j+1] = x
 end for
```





Sortarea prin inserare

- Exemplu
- Complexitate?
  - favorabil, defavorabil
- Stabilitate?
- Memorie?

3 7 4 9 5 2 6 1

3 7 4 9 5 2 6 1

3 7 4 9 5 2 6 1

3 4 7 9 5 2 6 1

3 4 7 9 5 2 6 1

3 4 5 7 9 2 6 1

2 3 4 5 7 9 6 1

2 3 4 5 6 7 9 1

1 2 3 4 5 6 7 9

Sortarea prin selectie

```
int i,j;
for (j = 0; j < n-1; j++) {
    /* find the min element in the unsorted a[j .. n-1] */
   int iMin = j;
    /* test against elements after j to find the smallest */
    for ( i = j+1; i < n; i++) {
        if (a[i] < a[iMin]) {
            /* found new minimum; remember its index */
            iMin = i;
        }
    }
    if(iMin != j) {
        swap(a[j], a[iMin]);
    }
}
```

Sortarea prin selectie

- Exemplu
- Complexitate?
  - favorabil, defavorabil
- Stabilitate?
- Memorie?
- Adaptabilitate?
  - dintre cei 3 algoritmi de pana acum, care este cel mai adaptabil?

3 7 4 9 5 2 6 1
1 7 4 9 5 2 6 3
1 2 4 9 5 7 6 3
1 2 3 9 5 7 6 4
1 2 3 4 5 7 6 9
1 2 3 4 5 7 6 9
1 2 3 4 5 6 7 9
1 2 3 4 5 6 7 9
1 2 3 4 5 6 7 9

Heapsort

- strategie bazata pe selectie
- se selecteaza minimul in timp logaritmic, nu liniar, prin utilizarea structurii de heap
  - *buildHeap* initial: *O(n)*
  - *extractMax* aplicat de n ori: *O(n*log(n))*

```
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)
```

- Exemplu
- Stabilitate?
- Memorie?

Sortarea prin numarare

- Cheile sunt in intervalul 1…k, deci pot fi folosite pt. indexarea unui vector (nu se bazeaza pe comparatii)
- Se numara aparitiile fiecarei chei in A, si se stocheaza aceasta informatie in vectorul C

	1	2	3	4	5	6
A **=**	3	4	1	4	1	1
C **=**	3	0	1	2		

- Apoi se determina numarul de elemente care sunt <= cu fiecare dintre chei, prin calcularea sumelor prefix

C **=**	3	0	4	6

Sortarea prin numarare

- Rezultatul se genereaza intr-un vector nou, ***B***, parcurgand ***A*** si determinand pozitia elementului prin informatia stocata in ***C***; se scade cu 1 valoarea intrarii elementului in ***C***

	1	*2*	*3*	*4*	*5*	*6*
A =	3	4	1	4	1	1
B =			1			
C =	2	3	4	6		
B =	1	1	1	3		
C =	1	3	4	6		
B =	1	1	1	3		4
C =	1	3	4	5		

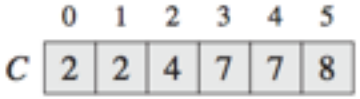
Sortarea prin numarare

```
COUNTING-SORT(A,B,k)
1 let C[0…k] be a new array
2 for i=0 to k
3     C[i] = 0
4 for j=1 to A.length
5     C[A[j]] = C[A[j]] + 1
6 // C[i] now contains the number of elements = i
7 for i=1 to k
8     C[i] = C[i] + C[i-1]
9 // C[i]i now contains the number of elements <= i
10 for j=A.length downto 1
11    B[C[A[j]]] = A[j]
12    C[A[j]] = C[A[j]] - 1
```

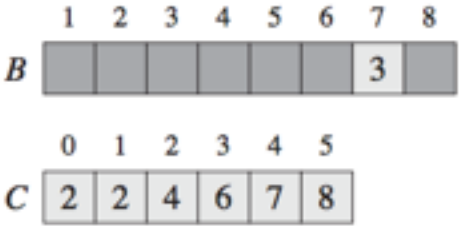
- Stabilitate?
- Memorie?

Sortarea prin numarare - exemplu

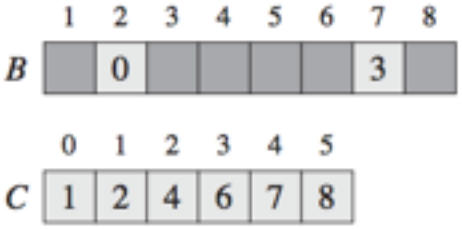
(a)



(b)

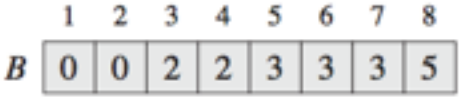


(c)



(d)

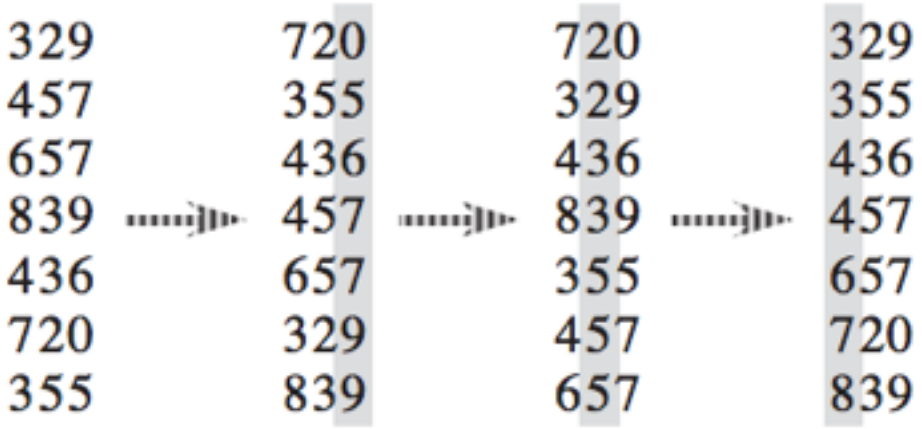
(e)



(f)

Radix Sort

- Pp. ca avem de sortat o secventa de intregi, avand 3 cifre => sortam intai dupa ultima cifra, apoi dupa a doua, apoi dupa prima




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

De ce sortare stabila?

Radix Sort

- Eficienta
  - daca avem cifrele de la 1 la k, putem folosi sortarea prin numarare pt. a sorta cifra *i*

$$\Theta(n+k)$$

  - facem d treceri prin sir

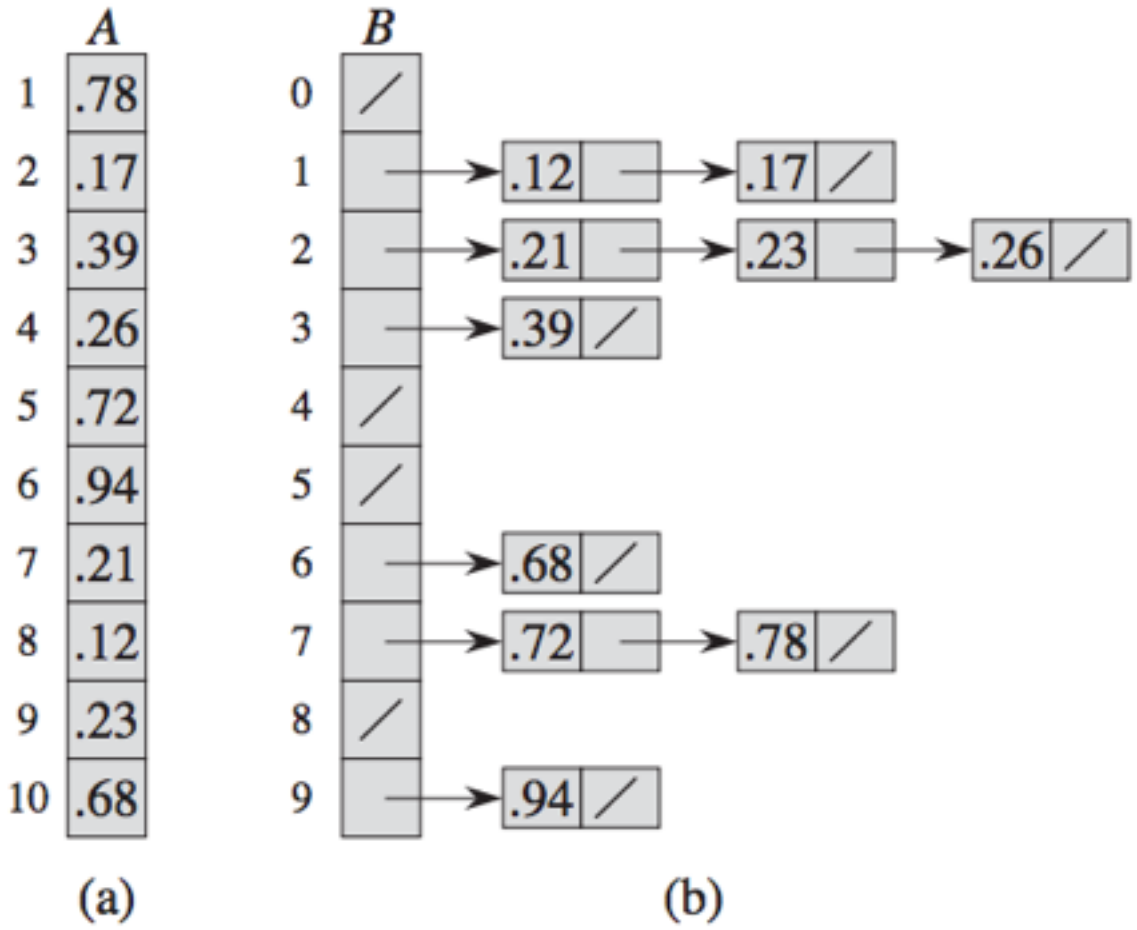
$$\Theta(d(n+k))$$

- Mai poate fi folosita la sortarea datelor (an-luna-zi)

Bucket Sort

- Cheile sunt numere reale in intervalul [0,1)
- Creem 10 *buckets*, cate una pentru fiecare interval $[i / 10, (i+1) / 10)$, si stocheaza fiecare element in *bucketul* corspunzator
- Sortam *bucketurile* cu un alt algoritm - e.g. insertion sort
- $\Theta(n)$ in cazul mediu, presupunand distributie uniforma a cheilor, si am ales intervalele suficient de mici
- Impartirea in *buckets* nu se bazeaza pe comparatii (e.g. se impart cheile la 10, si luam partea intreaga pt a selecta *bucketul*)

Bucket Sort


A
1 .78
2 .17
3 .39
4 .26
5 .72
6 .94
7 .21
8 .12
9 .23
10 .68
B
0
1 .12 .17
2 .21 .23 .26
3 .39
4
5
6 .68
7 .72 .78
8
9 .94
(a)
(b)

Bucket Sort

$n = \text{length}[A]$	$\Omega(1)$
for $i = 1$ to n	$O(n)$
do insert $A[i]$ into bucket $B\lfloor n[A[i]] \rfloor$	$\Omega(1)$ (**i.e. total** $O(n)$)
for $i = 0$ to $n-1$	$O(n)$
do sort bucket $B[i]$ with insertion sort	$O(n_i^2)$
Concatenate bucket $B[0], B[1], \ldots, B[n\text{-}1]$	$O(n)$

n_i este dimensiunea *bucketului* $B[i]$

Deci $T(n) = \Omega(n) + \sum_{i=0}^{n-1} O(n_i^2)$

$= \Omega(n) + nO(2\text{-}1/n) = \Omega(n)$

Cum alegem algoritmul potrivit

- Dimensiunea problemei
  - toate cursurile vs cursurile unui singur student
- Elementele de sortat ocupa memorie multa?
  - de evitat mutarile in cazul acesta
  - utilizare de structuri auxiliare - pointeri - mutam pointerii, nu elmentele
- Avem nevoie de garantii asupra timpului de sortare (e.g. sisteme de control, retele)
  - nu putem utiliza QuickSort datorita comportamentului sau in cazul defavorabil

Cum alegem algoritmul potrivit

- Elementele pot avea aceleasi chei? Avem nevoie de algoritm stabil?
  - $O(n^2)$ tind sa fie stabili, $O(nlgn)$ nu prea
  - Algoritmii instabili pot fi transformati in algoritmi stabili - cheie cu pozitia
    - costa spatiu si timp
  - Avem spatiu limitat?
    - MergeSort - O(n) - memorie
    - QuickSort - O(n) memorie in cazul defav.

Cum alegem algoritmul potrivit

- S-ar putea ca secventa sa nu incapa in memorie? (memorie virtuala)
  - Daza da se prefera algoritmi cu comportament local
    - HepSort, QuickSort, met. directe
- S-ar putea ca secventa sa nu incapa nici in memoria virtuala?
  - sortari externe
- Ce stim despre intrare?
  - Daca sunt intr-un domeniu mic -> CountingSort
  - Daca avem chei compuse din chei ce se compara individual -> RadixSort
  - Daca avem chei numere reale distribuite uniform intr-un anumit interval -> BucketSort