Metode de sortare

- pregătire admitere -

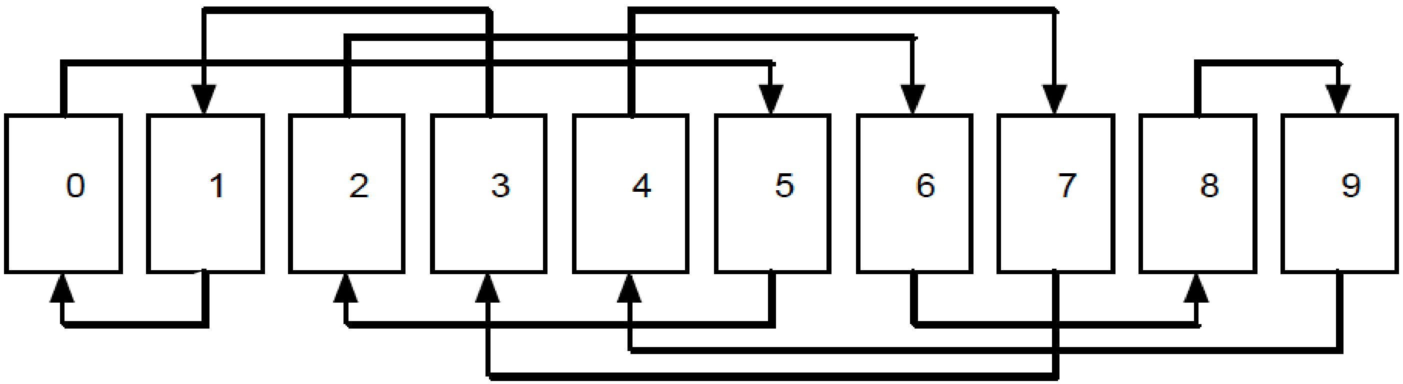
Universitatea din Bucureşti

Cuprins

- **Problema sortării**
- Algoritmul de sortare prin interschimbare (Bubble sort)
- Algoritmul de sortare prin inserţie (Insertion sort)
- Algoritmul de sortare prin selecţie (Selection sort)
- Algoritmul de sortare prin interclasare (Mergesort)
- Algoritmul de sortare rapidă (Quicksort)

Problema sortării

- Fie un şir de "entităţi": e.g. numere, caractere, înregistrări ale unei baze de date, etc. Fiecare entitate deţine o caracteristică numită cheie de sortare.



http://www.whydomath.org/Reading_Room_Material/ian_stewart/shuffle/shuffle.html

- Există o relaţie de ordine ce se defineşte peste mulţimea valorilor posibile ale cheii de sortare.
- Sortarea şirului = dispunerea elementelor şirului astfel încât valorile cheilor de sortare să se afle în ordine crescătoare (sau descrescătoare)

Problema sortării - Exemple

- Elemente numerice: $[1, 5, 2, 3, 6, 10, 22, 4, 44]$. În acest caz cheia de sortare este reprezentată de însuşi elementul şirului.

- Elemente definite de înregistrări cu mai multe câmpuri (nume şi vârstă):

  (Andrei, 29), (Paul, 29), (Tiberiu, 10), (Corina, 25), (Claudiu, 19).

  Două posibile chei de sortare pentru fiecare element:

  - Nume
  - Vârstă

  Sortare după nume:

  (Andrei, 29), (Claudiu, 19), (Corina, 25), (Paul, 29), (Tiberiu, 10).

  Sortare după vârstă:

  (Tiberiu, 10), (Claudiu, 19), (Corina, 25), (Andrei, 29), (Paul, 29).

Problema sortării

- Fie şirul:

$$x[1], \quad x[2], \quad \cdots x[n].$$

  Dacă notăm cheia elementului x_i cu K_i, atunci sortarea şirului presupune determinarea unei permutări $p(1), p(2), \cdots, p(n)$ astfel încât:

$$K_{p(1)} \leq K_{p(2)} \leq \cdots \leq K_{p(n)}.$$

- Pentru simplitate, considerăm că cheia de sortare K_i este reprezentată de întregul element x_i.

Proprietăţi ale metodelor de sortare

1. **Eficienţa**. Orice metodă de sortare este caracterizată de volumul de resurse (timp de execuţie) necesitat pentru rezolvarea problemei.
   - Număr de comparaţii (de chei)
   - Număr de deplasări (de date)

2. **Stabilitate**. O metodă de sortare este stabilă dacă ordinea relativă a elementelor cu aceeaşi cheie de sortare rămâne neschimbată.
   Aranjament iniţial:

   (Andrei, 29), (Paul, 29), (Tiberiu, 10), (Corina, 25), (Claudiu, 19).

   Sortare stabilă:

   (Tiberiu, 10), (Claudiu, 19), (Corina, 25), (Andrei, 29), (Paul, 29).

   Sortare instabilă:

   (Tiberiu, 10), (Claudiu, 19), (Corina, 25), (Paul, 29), (Andrei, 29).

3. **Simplitate**. O metodă de sortare este caracterizată de nivelul de complexitate al implementării sale în practică.

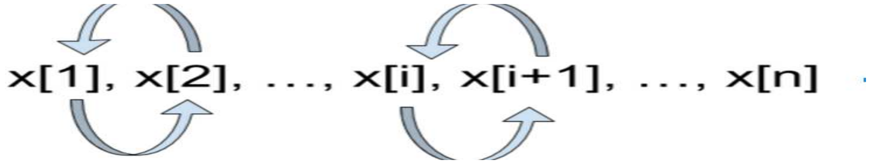
Cuprins

- Problema sortării
- **Algoritmul de sortare prin interschimbare (Bubble sort)**
- Algoritmul de sortare prin inserţie (Insertion sort)
- Algoritmul de sortare prin selecţie (Selection sort)
- Algoritmul de sortare prin interclasare (Mergesort)
- Algoritmul de sortare rapidă (Quicksort)

Sortare prin interschimbare (Bubble sort)

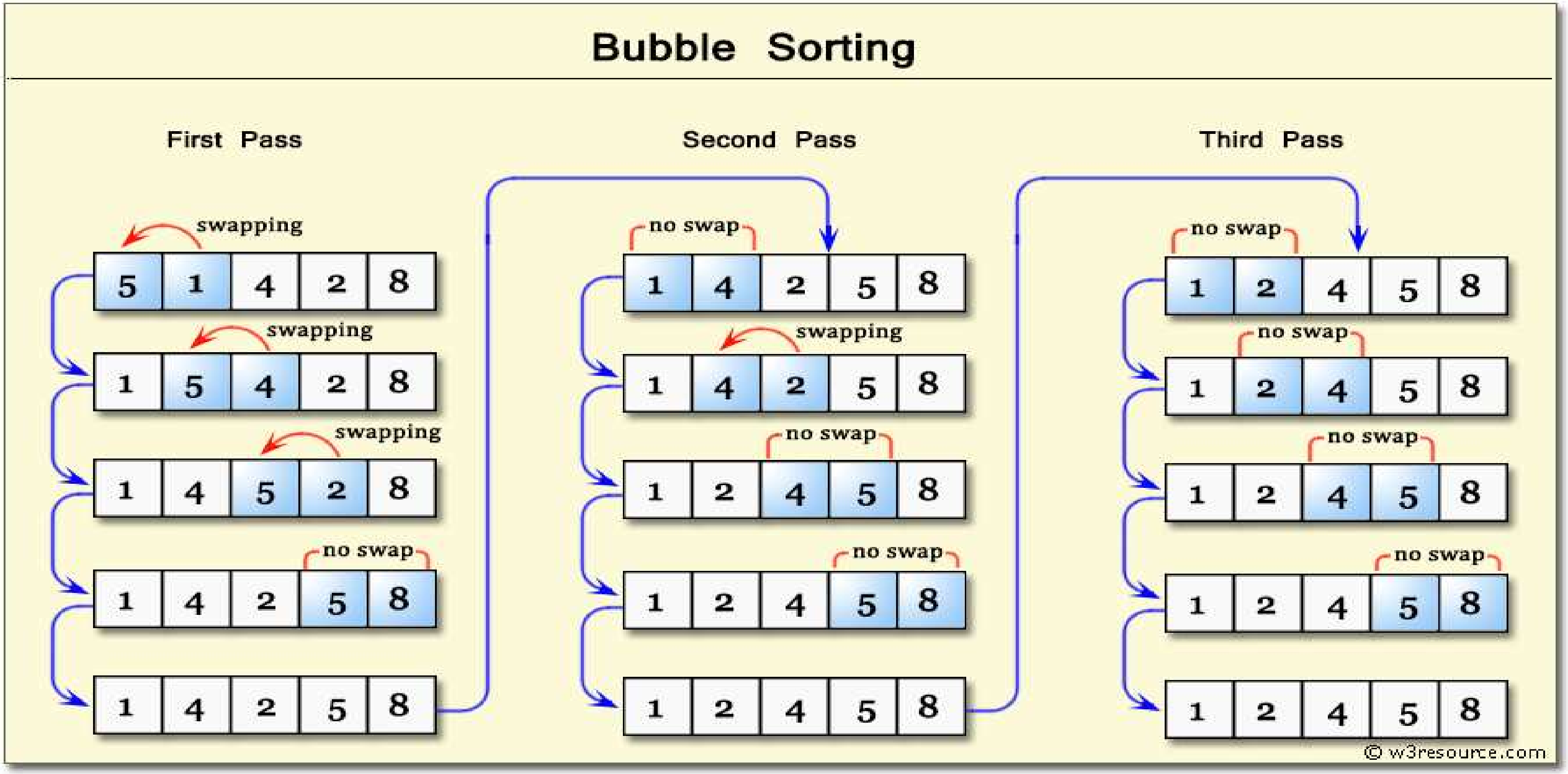
Idei generale:

- Considerăm şirul: $x[1], x[2], \cdots, x[n]$.
- Se parcurge secvenţial şirul (element cu element). Se compară elementul curent cu următorul şi, în cazul în care se află în ordinea nepotrivită, se interschimbă.



- Se efectuează parcurgeri până şirul nu mai necesită interschimbări

Sortare prin interschimbare - Exemplu



Sortare prin interschimbare

Algoritm **Sort-Interschimbare**(array x):

1. **repeat**
   1. **for** $i = 2 \cdots n$
      1. **if** $(x[i] < x[i-1])$ // dacă $x[i]$ şi $x[i-1]$ nu sunt ordonate
         1. $x[i-1] \leftrightarrow x[i]$ // se interschimbă $x[i]$ cu $x[i-1]$
         2. $swap =$true

         endif

      endfor

   until not swap // repetă până nu mai sunt necesare interschimbări
1. **return** x

Sortare prin interschimbare - Eficienţă

Algoritm **Sort-Interschimbare**(array x):

1. **repeat**
    1. **for** $i = 2 \cdots n$
        1. **if** $(x[i] < x[i-1])$
            1. $x[i-1] \leftrightarrow x[i]$
            2. $swap =$true

           endif

       endfor

   until not swap
2. **return** x

Operaţii:

- Comparaţii $T_C(n)$
- Deplasări elemente $T_D(n)$

Pentru fiecare i: 1 comparaţii

Pentru toate valorile i:

$$\sum_{i=2}^{n} 1 = n - 1$$

Pentru toate parcurgerile:

$$n - 1 \leq T_C(n) \leq \frac{n(n-1)}{2}$$

Sortare prin interschimbare - Eficienţă

Algoritm **Sort-Interschimbare**(array x):

1. **repeat**
    1. **for** $i = 2 \cdots n$
        1. **if** $(x[i] < x[i-1])$
            1. $x[i-1] \leftrightarrow x[i]$
            2. $swap =$true

            endif

        endfor

    until not swap
2. **return** x

Operaţii:

- Comparaţii $T_C(n)$
- Deplasări elemente $T_D(n)$

Pentru fiecare i: Maxim 2

Pentru toate valorile i:

$$0 \leq T_i \leq 2(n-i)$$

Pentru toate parcurgerile:

$$0 \leq T_D(n) \leq n(n-1)$$

Sortare prin interschimbare - Eficienţă

- Comparaţii: $n - 1 \leq T_C(n) \leq \frac{n(n+1)}{2}$

- Deplasări elemente: $0 \leq T_D(n) \leq n(n-1)$

- Complexitate totală:

$$n - 1 \leq T(n) = T_C(n) + T_D(n) \leq \frac{n(n+1)}{2} + n(n-1) \approx \mathcal{O}(n^2)$$

- Cel mai favorabil caz: $1, 2, 3, \cdots, n$; Cel mai nefavorabil: $n, n-1, \cdots, 1$

- Algoritmul de sortare prin interschimbare este stabil!

- Simplu de implementat, dar ineficient pentru şiruri de dimensiuni mari!

Cuprins

- Problema sortării
- Algoritmul de sortare prin interschimbare (Bubble sort)
- **Algoritmul de sortare prin inserţie (Insertion sort)**
- Algoritmul de sortare prin selecţie (Selection sort)
- Algoritmul de sortare prin interclasare (Mergesort)
- Algoritmul de sortare rapidă (Quicksort)

Sortare prin inserţie (Insertion sort)

Idei generale:

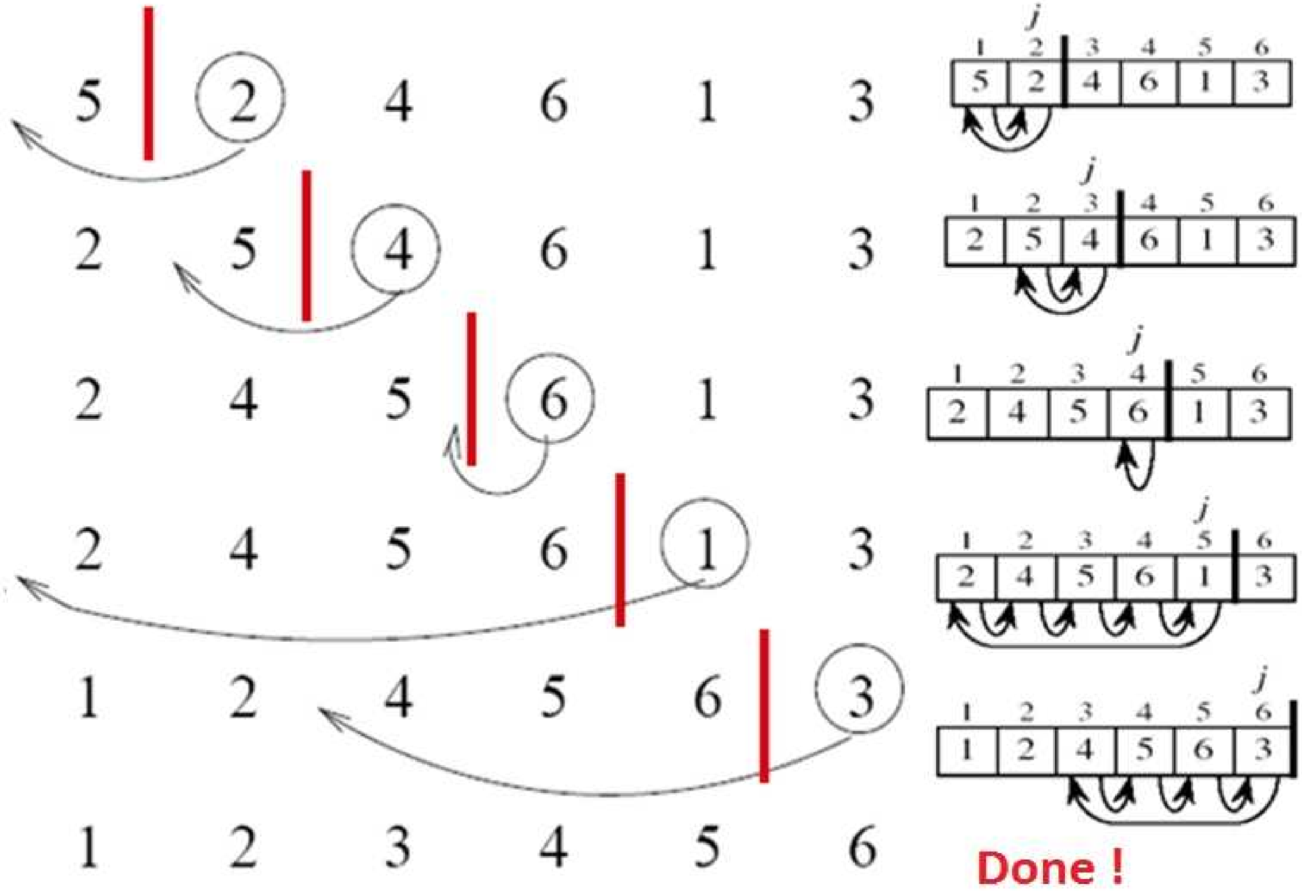
- Considerăm şirul: $x[1], x[2], \cdots, x[n]$.

- Se parcurge secvenţial şirul (element cu element). Se inserează elementul curent ($x[i]$) în subşirul care îl precede ($x[1], x[2], \cdots, x[i-1]$) astfel încât acesta să rămână ordonat:

$$x[1], x[2], \cdots, x[i-1], x[i], x[i+1], \cdots, x[n]$$

- Subşirul ce conţine elementele deja sortate creşte la fiecare iteraţie, astfel încât, după ce parcurgem toate elementele, secvenţa este sortată în întregime

Sortare prin inserţie - Exemplu



http://freefeast.info/general-it-articles/insertion-sort-pseudo-code-of-insertion-sort-insertion-sort-in-data-structure/

Sortare prin inserţie

Algoritm **Sort-Inserţie**(array x):

1. **for** $i = 2 \cdots n$
   1. aux $\leftarrow x[i]$ // fixarea elementului i
   2. $j \leftarrow i - 1$
   3. **while** $(j >= 1)$ and (aux $< x[j]$) // căutarea poziţiei
      1. $x[j+1] \leftarrow x[j]$ // deplasare element $x[j]$ pe poziţia $j+1$
      2. $j \leftarrow j - 1$

      endwhile
   4. $x[j+1] \leftarrow$ aux // deplasare element $x[i]$ pe poziţia căutată

   endfor
2. **return** x

Sortare prin inserţie - Eficienţă

Sort-Inserţie(array x)

1. **for** $i = 2 \cdots n$
   1. aux $\leftarrow x[i]$
   2. $j \leftarrow i - 1$
   3. **while** $(j >= 1)$ and (aux<$x[j]$)
      1. $x[j+1] \leftarrow x[j]$
      2. $j \leftarrow j - 1$

      endwhile
   4. $x[j+1] \leftarrow$ aux

   endfor
2. **return** x

Operaţii:

- Comparaţii $T_C(n)$
- Deplasări elemente $T_D(n)$

Pentru fiecare i:

$$1 \leq T_i(n) \leq i$$

Pentru toate valorile i:

$$n - 1 \leq T_C(n) \leq \frac{n(n+1)}{2} - 1$$

Sortare prin inserţie - Eficienţă

Sort-Inserţie(array x)

1. **for** $i = 2 \cdots n$
   1. aux $\leftarrow x[i]$
   2. $j \leftarrow i - 1$
   3. **while** $(j >= 1)$ and (aux<$x[j]$)
      1. $x[j+1] \leftarrow x[j]$
      2. $j \leftarrow j - 1$

      endwhile
   4. $x[j+1] \leftarrow$ aux

   endfor
2. **return** x

Operaţii:

- Comparaţii $T_C(n)$
- Deplasări elemente $T_D(n)$

Pentru fiecare i:

$$0 + 2 \leq T_i(n) \leq (i-1) + 2 = i + 1$$

Pentru toate valorile i:

$$2(n-1) \leq T_D(n) \leq \frac{(n+1)(n+2)}{2} - 3$$

Sortare prin inserţie - Eficienţă

- Comparaţii: $n-1 \le T_C(n) \le \frac{n(n+1)}{2} - 1$

- Deplasări elemente: $2(n-1) \le T_D(n) \le \frac{(n+1)(n+2)}{2} - 3$

- Complexitate totală:

$$3(n-1) \le T(n) = T_C(n) + T_D(n) \le n^2 + 2n - 3 \approx \mathcal{O}(n^2)$$

- Cel mai favorabil caz: $1, 2, 3, \cdots, n$; Cel mai nefavorabil: $n, n-1, \cdots, 1$

- Algoritmul de sortare prin inserţie este stabil!

- Este foarte simplu de implementat (există implementări in 5 linii de cod)!

- Recomandat pentru şiruri restrânse de elemente (performanţe practice bune!)

Sortare prin inserţie binară

Idei generale:

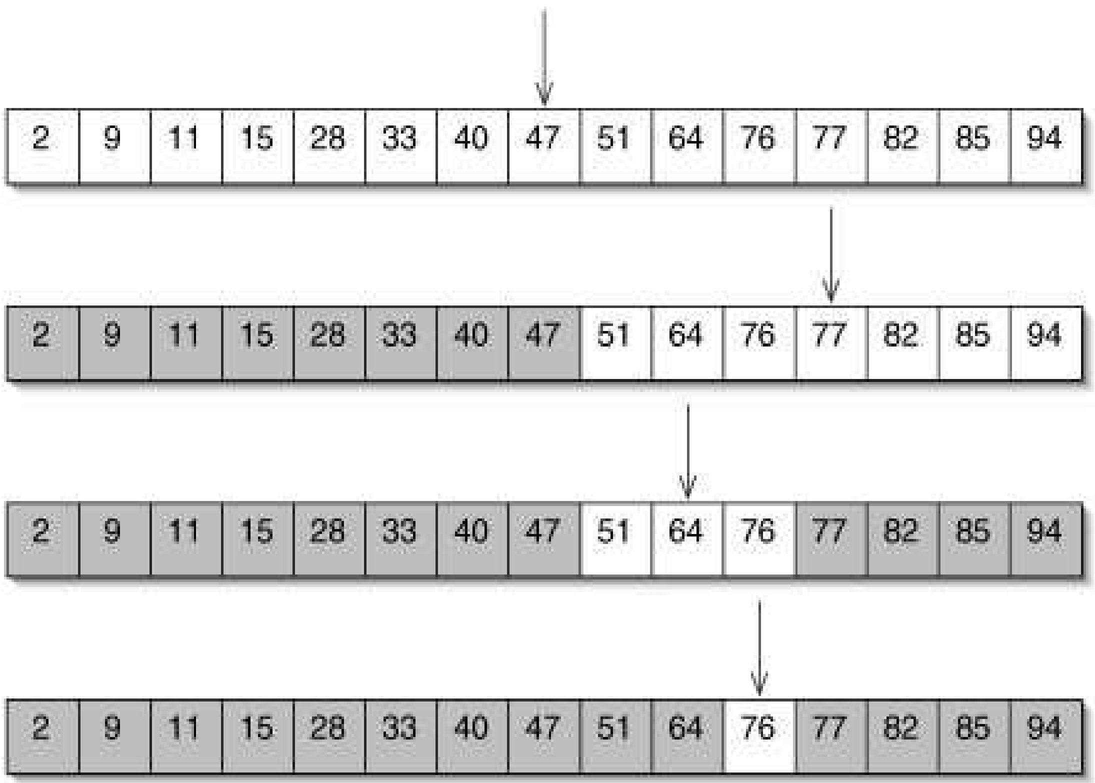
- Considerăm şirul: $x[1], x[2], \cdots, x[n]$.

- Se parcurge secvenţial şirul (element cu element). Se inserează elementul curent ($x[i]$) în subşirul care îl precede ($x[1], x[2], \cdots, x[i-1]$), folosind algoritmul de căutare binară, astfel încât acesta să rămână ordonat:

$$x[1], x[2], \cdots, x[i-1], x[i], x[i+1], \cdots, x[n]$$

- Îmbunătăţire a metodei de sortare prin inserţie directă: se înlocuieşte procedura de căutare liniară ($\mathcal{O}(n)$) cu cea de căutare binară ($\mathcal{O}(\log(n))$).

Sortare prin inserţie binară



http://flylib.com/books/en/2.300.1.75/1/

Sortare prin inserţie binară

Căutare-binară(array x, int p, int q, int key):

1. mid = p + ((q-p) / 2)
2. **if** (key$>$ $x[\text{mid}]$)
   1. return **Căutare-binară**(x, mid+1, q, key)

   else
   1. return **Căutare-binară**(x, p, mid, key)

   endif
3. **return** mid

- Complexitate: $T(n) = \mathcal{O}(\log(n))$
- Sortarea prin inserţie binară reduce numărul de comparaţii de la $\mathcal{O}(n^2)$ la $\mathcal{O}(n\log(n))$
- Cu toate astea, complexitatea totală se păstrează $\mathcal{O}(n^2)$, datorită numărului de deplasări!

Cuprins

- Problema sortării
- Algoritmul de sortare prin interschimbare (Bubble sort)
- Algoritmul de sortare prin inserţie (Insertion sort)
- **Algoritmul de sortare prin selecţie (Selection sort)**
- Algoritmul de sortare prin interclasare (Mergesort)
- Algoritmul de sortare rapidă (Quicksort)

Sortare prin selecţie (Selection sort)

Idei generale:

- Considerăm şirul: $x[1], x[2], \cdots, x[n]$.

- Se parcurge secvenţial şirul (element cu element). Se determină minimum-ul din subşirul ($x[i], x[i+1], \cdots, x[n]$), şi se înlocuieşte cu elementul curent ($x[i]$).

$$[x[1], x[2], \cdots, x[i-1], x[i], x[i+1], \cdots, x[n]]$$

- Subşirul ce conţine elementele deja sortate se mareşte la fiecare iteraţie, astfel încât, după ce parcurgem toate elementele, secvenţa este sortată în întregime

Sortare prin selecţie - Exemplu

http://freefeast.info/general-it-articles/selection-sort-pseudo-code-of-selection-sort-selection-sort-in-data-structure/

Sortare prin selecţie

Algoritm **Sort-Selecţie**(array x):

1. **for** $i = 1 \cdots n - 1$
   1. $k \leftarrow i$
   2. **for** $j = i + 1 \cdots n$ // căutare minim în subşirul $x[i : n]$
      1. **if** $x[k] > x[j]$
         1. $k \leftarrow j$

      endfor
   3. **if** $k \neq i$ // dacă elementul curent nu este minim-ul subşirului $x[i : n]$
      1. $x[k] \leftrightarrow x[i]$ // interschimbare minim cu elementul curent

   endfor
2. **return** x

Sortare prin selecţie - Eficienţă

Sort-Selecţie(array x):

1. **for** $i = 1 \cdots n-1$
    1. k $\leftarrow i$
    2. **for** $j = i+1 \cdots n$
        1. **if** $x[k] > x[j]$
            1. $k \leftarrow j$

        endfor
    3. **if** $k \neq i$
        1. $x[k] \leftrightarrow x[i]$

   endfor
2. **return** x

Operaţii:

- Comparaţii $T_C(n)$
- Deplasări elemente $T_D(n)$

Pentru fiecare i:

$$T_i(n) = n - i$$

Pentru toate valorile i:

$$T_C(n) = \frac{n(n-1)}{2}$$

Sortare prin selecţie - Eficienţă

Sort-Selecţie(array x):

1. **for** $i = 1 \cdots n-1$
   1. $k \leftarrow i$
   2. **for** $j = i+1 \cdots n$
      1. **if** $x[k] > x[j]$
         1. $k \leftarrow j$

      endfor
   3. **if** $k \neq i$
      1. $x[k] \leftrightarrow x[i]$

   endfor
2. **return** x

Operaţii:
- Comparaţii $T_C(n)$
- Deplasări elemente $T_D(n)$

Pentru fiecare i:

$$0 \leq T_i(n) \leq 3$$

- fiecare interschimbare necesită 3 deplasări

Pentru toate valorile i:

$$0 \leq T_D(n) \leq 3(n-1)$$

Algoritmul de sortare prin selecţie - Eficienţă

- Comparaţii: $T_C(n) = \frac{n(n-1)}{2}$

- Deplasări elemente: $0 \leq T_D(n) \leq 3(n-1)$

- Complexitate totală:

$$\frac{n(n-1)}{2} \leq T(n) = T_C(n) + T_D(n) \leq \frac{n(n-1)}{2} + 3(n-1) \approx \mathcal{O}(n^2)$$

- Număr redus de deplasări elemente! (faţă de Sortarea prin inserţie)

- Algoritmul de sortare prin selecţie nu este stabil!

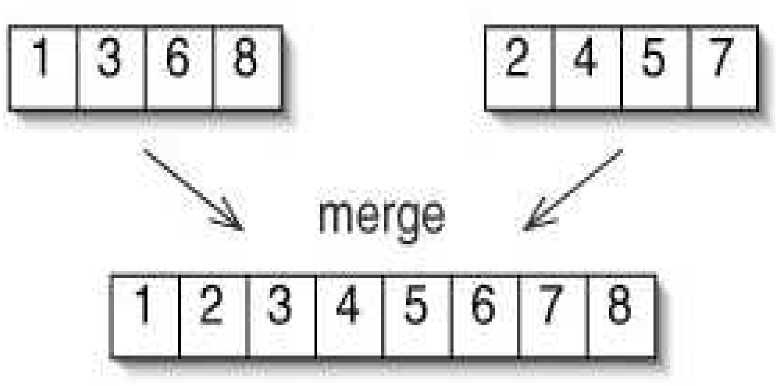
Cuprins

- Problema sortării
- Algoritmul de sortare prin interschimbare (Bubble sort)
- Algoritmul de sortare prin inserţie (Insertion sort)
- Algoritmul de sortare prin selecţie (Selection sort)
- **Algoritmul de sortare prin interclasare (Mergesort)**
- Algoritmul de sortare rapidă (Quicksort)

Interclasare

- Fie şirurile: $x[1], x[2], \cdots, x[n]$ şi $y[1], y[2], \cdots, y[m]$ $(m \approx n)$ ordonate crescător;

- Procedura de interclasare
  1. Se compară elementele minime din fiecare şir
  2. Se returnează cel mai mic
  3. Se repetă paşii precedenţi pană la epuizarea elementelor



http://flylib.com/books/en/2.300.1.86/1/

Interclasare

Algoritm **Interclasare**(array x, array y)

1. $i \leftarrow 1, j \leftarrow 1, k \leftarrow 1$
2. **while** $j \leq m$ and $i \leq n$
   2. **if** $x_i \leq y_j$
      1. $z_k \leftarrow x_i$
      2. $k \leftarrow k + 1$
      3. $i \leftarrow i + 1$

      else

      1. $z_k \leftarrow y_j$
      2. $k \leftarrow k + 1$
      3. $j \leftarrow j + 1$
3. **if** $i < n$
   1. $(z_k, \cdots, z_{m+n}) \leftarrow (x_i, \cdots, x_n)$

   else

   1. $(z_k, \cdots, z_{m+n}) \leftarrow (y_j, \cdots, y_m)$
4. return z

• Complexitate $\mathcal{O}(m + n)$

• Operaţie mult mai simplă decât sortarea

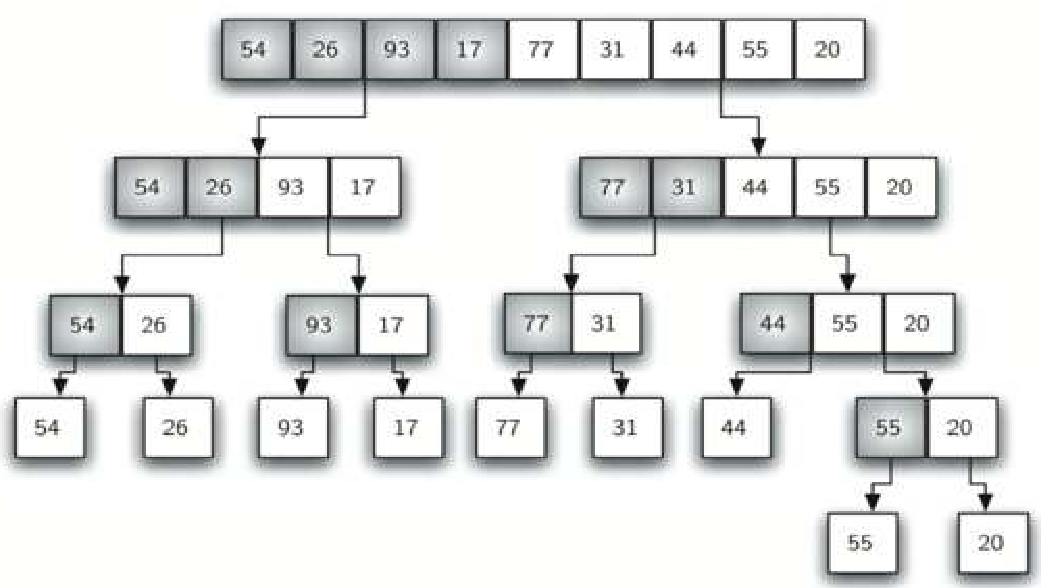
• Reducem procesul de sortare la operaţii de interclasare: se interclasează subşiruri din ce in ce mai lungi, până se obţine şirul sortat.

Sortare prin interclasare (Merge sort)

Structură generală:

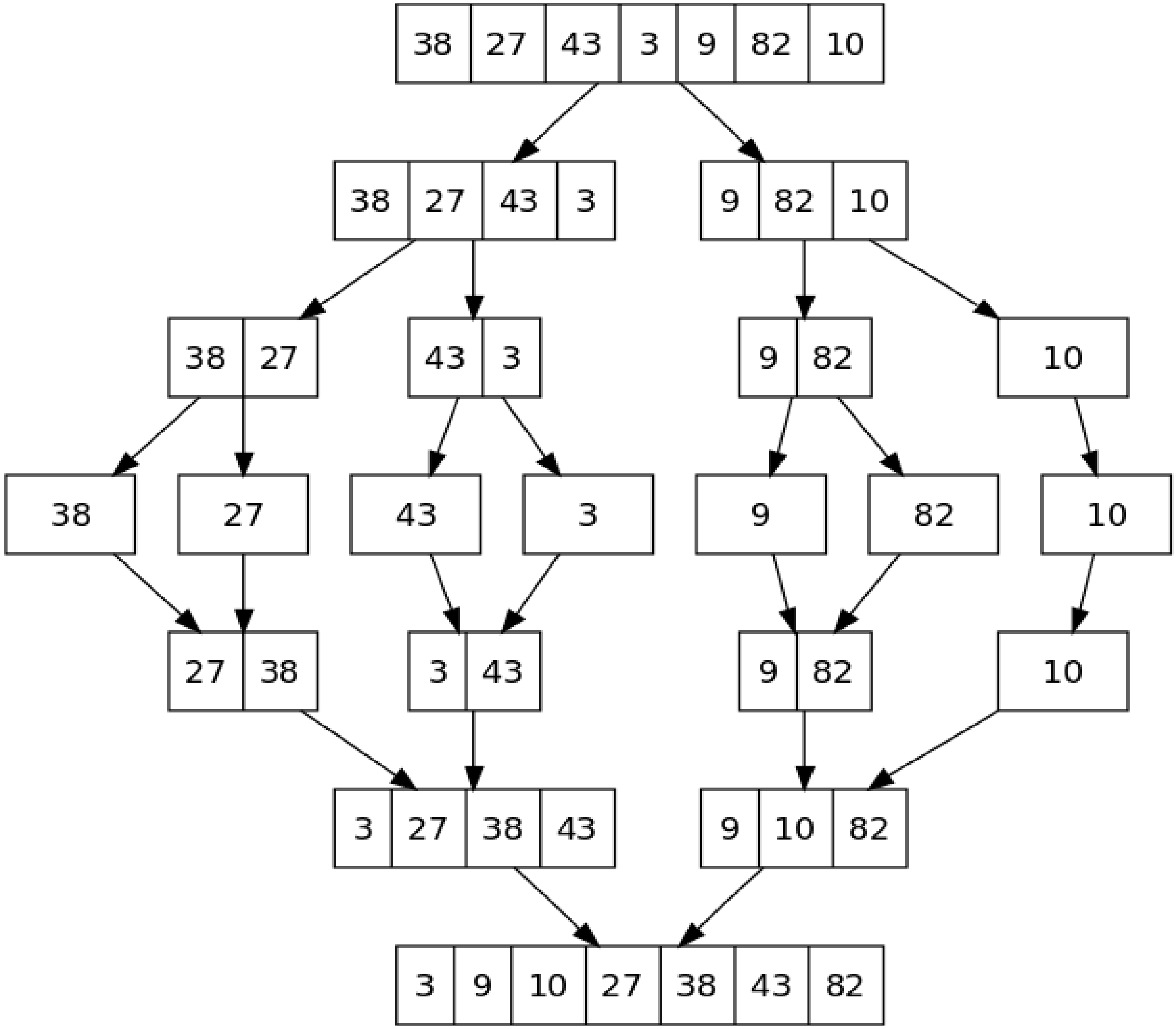
- Considerăm şirul: $x[1], x[2], \cdots, x[n]$.

- Se descompune şirul iniţial nesortat în n subşiruri de lungime 1
- Se interclasează subşirurile obţinute la pasul precedent până rezultă un singur şir



http://interactivepython.org/courselib/static/pythonds/SortSearch/TheMergeSort.html

Sortare prin interclasare - Exemplu



Algoritmul de sortare prin interclasare

Algoritm **Sort-Interclasare**(array x, int p, int q):

1. **if** $(p < r)$ // we have at least 2 items
   1. $q = (p + r)/2$
   2. **Sort-Interclasare**(x, p, q) // sortare x[p:q]
   3. **Sort-Interclasare**(x, q+1, r) // sortare x[q+1:r]
   4. **Interclasare**(x[p:q], x[q+1:r]) // interclasare subşirurilor

   endif

Sortare prin interclasare - Eficienţă

Algoritm **Interclasare**(array x, array y)

1. $i \leftarrow 1, j \leftarrow 1, k \leftarrow 1$
2. **while** $j \leq m$ and $i \leq n$
   2. **if** $x_i \leq y_j$
      1. $z_k \leftarrow x_i$
      2. $k \leftarrow k + 1$
      3. $i \leftarrow i + 1$

      else
      1. $z_k \leftarrow y_j$
      2. $k \leftarrow k + 1$
      3. $j \leftarrow j + 1$
4. **if** $i > n$
   1. $(z_k, \cdots, z_{m+n}) \leftarrow (x_i, \cdots, x_n)$

   else
   1. $(z_k, \cdots, z_{m+n}) \leftarrow (y_j, \cdots, y_m)$

- Operaţiile dominante ale algoritmului **Sort-Interclasare** au loc in procedura de interclasare

- Se poate estima iterativ complexitatea $T(n)$:
  - Problema este descompusă iterativ in două subprobleme cu dimensiune redusă la jumătate
  - O subproblemă de dimensiune 1 costă $\mathcal{O}(1)$ operaţii
  - Procedura de interclasare costă $\mathcal{O}(n)$

Sortare prin interclasare - Eficienţă

- Complexitate totală: $\mathcal{O}(n\log(n))$

$$T(n) = \begin{cases} 1, & \text{dacă } n = 1 \\ T\left(\lceil \frac{n}{2} \rceil\right) + T\left(\lceil \frac{n}{2} \rceil\right) + n, & \text{altfel.} \end{cases}$$

- Recomandat pentru şiruri de mari dimensiuni

- Cu toate că are o complexitate mai bună, în cazul şirurilor de dimensiune mici, metoda sortării prin inserţie prezintă performanţe mai bune decât sortarea prin interclasare

- Algoritmul de sortare prin interclasare este stabil!

Cuprins

- Problema sortării
- Algoritmul de sortare prin interschimbare (Bubble sort)
- Algoritmul de sortare prin inserţie (Insertion sort)
- Algoritmul de sortare prin selecţie (Selection sort)
- Algoritmul de sortare prin interclasare (Mergesort)
- **Algoritmul de sortare rapidă (Quicksort)**

Algoritmul de sortare rapidă (Quicksort)

- Algoritm de sortare eficient dezvoltat de Tony Hoare în 1959 şi publicat în 1961
- Implementările bune pot depăşi de 2-3 ori performanţele sortării prin interclasare

Idei generale:

- Considerăm şirul: $x[1], x[2], \cdots, x[n]$.

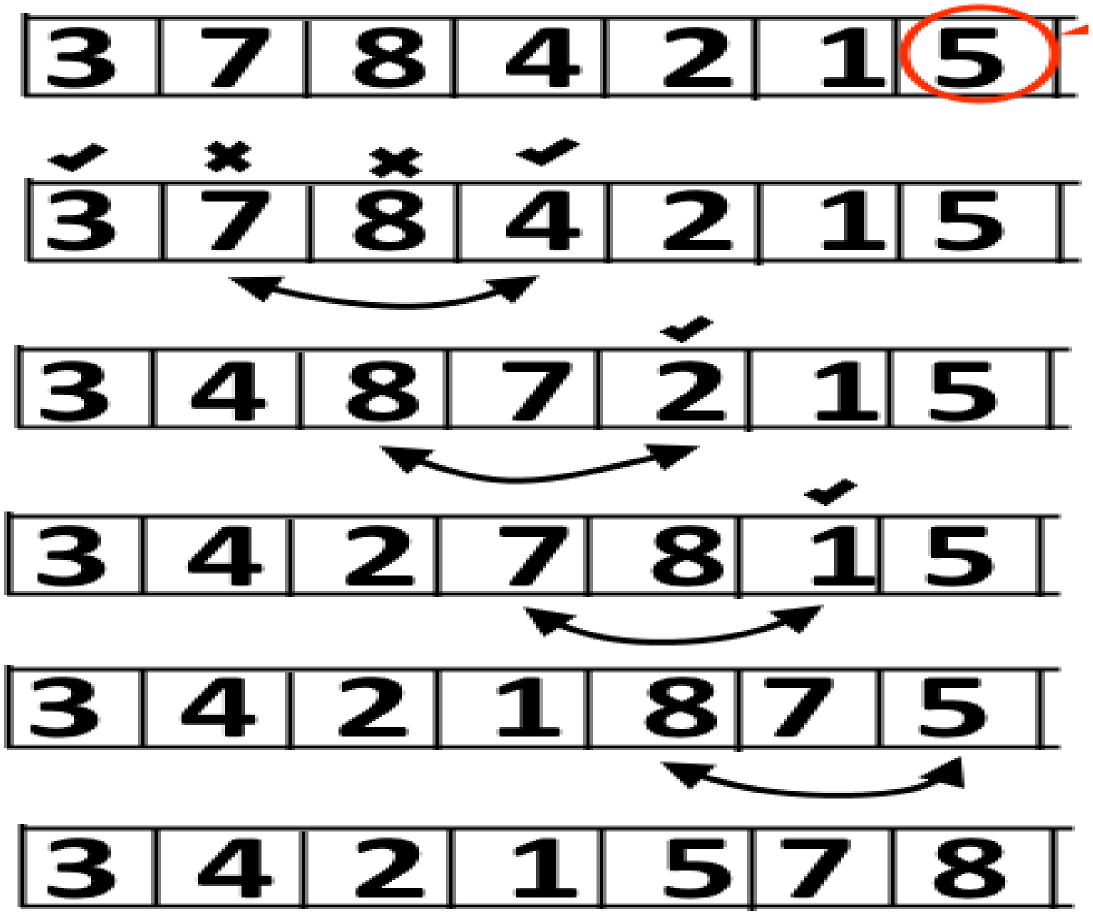
- Se alege din şir un element, numit *pivot*.

$$[x[1], x[2], \cdots, x[i-1], x[i], x[i+1], \cdots, x[n]]$$

  Se realizează *partiţionarea* şirului (în raport cu acest pivot): se reordonează şirul astfel încât elementele cu valori mai mici decât pivotul se plasează înainte de pivot, iar elementele cu valori mai mari decât pivotul se plasează după acesta (cele egale în oricare din parţi).

- După partiţionare, pivotul se află in poziţia finală. Se aplică recursiv aceeaşi procedură subşirului de elemente cu valori mai mici, iar separat subşirului cu valori mai mari.

Algoritmul de sortare rapidă (Quicksort)



Algoritmul de sortare rapidă (Quicksort)

Algoritm **Quicksort** (array x, int l, int h)

1. if ($l < h$)
   1. LocPivot = **Partitionare**(x, l, h)
   2. **Quicksort**(x, l, LocPivot) // recursie pe segmentul elementelor mai mici
   3. **Quicksort**(x, LocPivot $+1$, h) // recursie segment elemente mai mari

Algoritm **Partitionare** (array x, int l, int h)

1. pivot = $x[l]$ // considerăm pivot primul element
2. leftwall = l
3. **for** $i = l + 1 \cdots h$
   (a) **if** ($x[i] <$ pivot)
      1. $x[i] \leftrightarrow x$[leftwall] // elementul $x[i]$ se mută în segmentul corect
      2. leftwall = leftwall $+1$
4. pivot $\leftrightarrow x$[leftwall]
5. **return** leftwall

Algoritmul de sortare rapidă (Quicksort)

Algoritm **Quicksort** (array x, int l, int h)

1. if ($l < h$)
    1. LocPivot = **Partitionare**(x, l, h)
    2. **Quicksort**(x, l, LocPivot)
    3. **Quicksort**(x, LocPivot $+1$, h)

Algoritm **Partitionare** (array x, int l, int h)

1. pivot = $x[l]$
2. leftwall = l
3. **for** $i = l + 1 \cdots h$
    (a) **if** ($x[i] <$ pivot)
        1. $x[i] \leftrightarrow x$[leftwall]
        2. leftwall = leftwall $+1$
4. pivot $\leftrightarrow x$[leftwall]
5. **return** leftwall

- Operaţiile dominante ale algoritmului **Quicksort** au loc în procedura de partiţionare

- Estimare complexitate $T(n)$:
    - Problema este descompusă iterativ in două subprobleme cu dimensiune redusă la jumătate
    - O subproblemă de dimensiune 1 costă $\mathcal{O}(1)$ operaţii
    - Procedura de partiţionare costă, în cel mai rău caz, $\mathcal{O}(n)$ comparaţii

Sortare prin interclasare - Eficienţă

• Complexitate în cel mai rău scenariu: $\mathcal{O}(n^2)$

• Complexitate în medie: $\mathcal{O}(n \log{(n)})$

Avantaje:

• Necesită memorie $\mathcal{O}(\log(n))$; mult mai redusă decât sortarea prin interclasare

• Reprezintă cea mai bună alegere când viteza este foarte importantă, indiferent de dimensiunea setului de date.

Dezavantaje:

• Poate duce la umplerea stivei când se utilizează seturi largi de date.

• Performanţe modeste atunci când operează asupra unor liste aproape ordonate.

• Algoritmul de sortare rapidă este instabil!

Concluzii

- O mulţime de metode de sortare bazate pe comparaţii între chei, cu avantaje şi dezavantaje proprii

- Aplicaţia practică dictează alegerea metodei

- Metodele de sortare prin inserţie şi selecţie au performanţe bune când şirul de sortat are dimensiuni reduse

- Pentru şiruri de dimensiuni mari: sortare prin interclasare (Mergesort), sortare rapidă (Quicksort), etc.