Structuri de date și algoritmi

Universitatea Politehnica Timişoara
P-ţa Victoriei nr. 2
RO 300006 - Timişoara
Tel: +4 0256 403000
Fax: +4 0256 403021
rector@rectorat.upt.ro
www.upt.ro

Domeniul de studii: Informatică/ Specializarea: Informatică

SDA – Cursul 4

Ș.l. dr.ing. Adriana ALBU
adriana.albu@upt.ro
http://www.aut.upt.ro/~adrianaa

Universitatea
Politehnica
Timişoara

P-ţa Victoriei nr. 2
RO 300006 - Timişoara
Tel: +4 0256 403000
Fax: +4 0256 403021
rector@rectorat.upt.ro
www.upt.ro

2. Noțiuni despre algoritmi

3. Tehnici de sortare (partea întâia)

2 Noțiuni despre algoritmi / 2.1 Definiții, caracteristici

- **Algoritm** = metoda de rezolvare a unei probleme (clase de probleme)
- Elemente constitutive: Orice algoritm ia în considerare un
  - set de **date inițiale**, prelucrate pe baza unui
  - set de **reguli de transformare**, cu scopul obținerii unui
  - set de **date finale**
- Algoritmii diferă în mod esențial prin natura și ordinea în care se executa relațiile de transformare
- Caracteristici:
  - generalitatea
  - finitudinea
  - unicitatea – determinism

Algoritmii sunt **abstractizări** (elemente abstracte) care prin implementare devin program (proceduri, funcții)

2 Noțiuni despre algoritmi / 2.2 Analiza algoritmilor

- Analiza algoritmilor urmărește două obiective:
  - **precizarea predictivă a comportamentului** unui algoritm în timpul execuției sale
  - **compararea** unor algoritmi și **ierarhizarea** acestora în raport cu **performanțele** lor
- Se bazează pe ipoteza că toate sistemele de calcul pe care se execută algoritmii sunt convenționale
  - execută la un moment dat o singură instrucțiune, care durează un timp finit
  - timpul total al execuției = suma timpilor necesari execuției tuturor instrucțiunilor
- Obiectivele analizei se realizează prin:
  - **determinarea operațiilor** realizate în cadrul algoritmului și a **costurilor** lor relative, exprimate în **timp de execuție** (costul unei instrucțiuni poate fi cunoscut aprioric, cu excepția operațiilor cu șiruri de caractere și a anumitor bucle dinamice)
  - **aprecierea performanței** algoritmului prin execuția sa efectivă, utilizând seturi speciale de date de intrare, astfel încât să fie evidențiat comportamentul algoritmului, dar și extremele acestui comportament (cel mai defavorabil, cel mai favorabil)

2 Noțiuni despre algoritmi / 2.3 Notații asimptotice

Analiza algoritmilor se desfășoară în două etape

Analiza apriorică

- determinarea unei **funcții care mărginește asimptotic timpul de execuție** al algoritmului și care depinde de anumiți parametri relevanți (dimensiune date de intrare, dimensiune tablou etc).
- utilizarea notațiilor asimptotice permite **determinarea ordinului de mărime al timpului de execuție**

Analiza efectuată posterior implementării

- realizarea profilului algoritmului, respectiv **execuția algoritmului** pentru diferite seturi de date de intrare și **măsurarea timpului** de execuție
- această analiză va confirma sau va infirma rezultatele analizei apriorice

2 Noțiuni despre algoritmi / 2.3 Notații asimptotice

- **Ordinul de mărime** al timpului de execuție
  - caracteristică a eficienței algoritmului
  - permite compararea relativă a algoritmilor alternativi

- Studiul eficienței asimptotice a unui algoritm
  - se realizează utilizând mărimi de intrare suficient de mari pentru a face relevant
    - ordinul de mărime al timpului de execuție
    - limita la care tinde timpul de execuție odată cu creșterea nelimitată a mărimilor de intrare
    - (ex. 1000n vs. n^2)

2.3 Notații asimptotice / Notația Θ (theta)

➢ Prin definiție, fiind dată o funcție g(n) prin notația Θ (g(n)) se desemnează o mulțime de funcții, definită prin relația:

- $\Theta(g(n)) = \{f(n)$: există constante pozitive $c_1>0$, $c_2>0$, $n_0>0$
  astfel încât $0 <= c_1 g(n) <= f(n) <= c_2 g(n)$
  pentru orice $n >= n_0\}$

➢ Reformulare: O funcție f(n) aparține mulțimii Θ(g(n)) dacă există constantele pozitive c_1 și c_2 astfel încât funcția f(n) să poată fi cuprinsă între $c_1 g(n)$ și $c_2 g(n)$ pentru un n suficient de mare

➢ Deși Θ(g(n)) reprezintă o mulțime de funcții se utilizează frecvent notația:

- $f(n)= \Theta(g(n))$

➢ g(n) reprezintă o **margine asimptotică strânsă**

➢ Definiția notației (noțiunii) Θ este validă doar dacă f(n) este o funcție asimptotic crescătoare în raport cu n și dacă n ia valori suficient de mari

2.3 Notații asimptotice / Notația Θ (theta)

- f(n) – funcție polinomială (asimptotic crescătoare pentru n suficient de mare)
- g(n) poate fi estimată din f(n) dacă se neglijează termenii de ordin inferior și se alege pentru c_1 o valoare < coeficientul termenului de ordinul cel mai mare, iar pentru c_2 o valoarea mai mare
- Ex:
  - $f(n) = n^2 + 100n + \log_{10}n + 1000$
  - rata de creștere a termenilor lui f(n) în raport cu n

n	f(n)	n^2	100n	$\log_{10}n$	1000
1	1101	1	100	0	1000
10	2101	100	1000	1	1000
100	21002	10000	10000	2	1000
1000	1101003	1000000	100000	3	1000
10000	101001004	100000000	1000000	4	1000

  - $f(n)=an^3+bn^2+cn+d \Rightarrow f(n)= \Theta(n^3)$
  - funcție polinomială de grad 0 => $\Theta(1)$

2.3 Notații asimptotice / Notația O (O mare)

- Precizează **marginea asimptotic superioară**, într-o manieră similară notației Θ:
  - O(g(n))={f(n): există constante pozitive c>0 și n_0>0
    astfel încât 0 <= f(n) <= cg(n)
    pentru orice n >= n_0}
- Notația O are drept scop stabilirea unei ordonări relative a funcțiilor
  - ceea ce se compară este rata relativă de creștere a funcțiilor și nu valorile lor în anumite puncte
- f(n)=O(g(n)); rata de creștere a lui f(n) este <= cu cea a lui g(n); uzual se alege marginea cea mai apropiată
- Notația Θ este mai restrictivă decât notația O; Θ(g(n)) este inclusă în O(g(n)); Θ nu se refera la orice intrare
- Notația **O descrie cazul de execuție cel mai defavorabil** și, prin implicație, mărginește comportamentul algoritmului pentru orice intrare

2.3 Notații asimptotice / Notația O (O mare)

- Ex.
  - 1000n > n^2 pentru valori mici ale lui n, dar n^2 crește cu o rată mult mai mare decât 1000n; n^2 are un ordin de mărime mai mare decât 1000n
- Ex.
  - $f(n)=2n^2+3n+1=O(n^2)$; $2n^2+3n+1 <= cn^2$; => o mulțime de perechi c și n_0; f(n) și g(n) cresc cu aceeași rată
- Pentru un n dat, timpul de execuție depinde de configurația particulară a intrării de dimensiune n
- Timpul de execuție (TE) nu este o funcție de n; exprimarea „TE este O(g(n))” referă cazul cel mai defavorabil al TE
- Uzual definiția lui O nu se aplică în mod formal, ci se utilizează rezultate cunoscute care permit ordonarea funcțiilor după rata de creștere:
  - $O(1)<O(\log n)<O(n)<O(n\log n)<O(n^2)<O(2^n)<O(\log^2 n)<O(10^n)<O(n^m)$

2.3 Notații asimptotice / Notația O (O mare)

➢ Proprietăți ale notației O:

- R1. O(k)<O(n) pentru orice constantă k
- R2. ignorarea constantelor: kf(n)=O(f(n))
- R3. tranzitivitate: f(n)=O(g(n)) și g(n)=O(h(n)) => f(n)=O(h(n))
- R4. suma
  - f1(n) = O(f(n))
  - f2(n) = O(g(n))
  - f1(n) + f2(n) = max(O(f(n)),O(g(n)))
- R5. produsul
  - f1(n) = O(g1(n))
  - f2(n) = O(g2(n))
  - f1(n) · f2(n) = O(g1(n) · g2(n))
- R6. Dacă f(n) este un polinom de grad k, atunci $f(n)=O(n^k)$
- R7. $\log^k n=O(n)$ pentru orice constantă k
- R8. $\log_a n = O(\log_b n)$ pentru orice a>1 și b >1
- R9. $\log_a n=O(\lg n)$ unde $\lg n= \log_2 n$

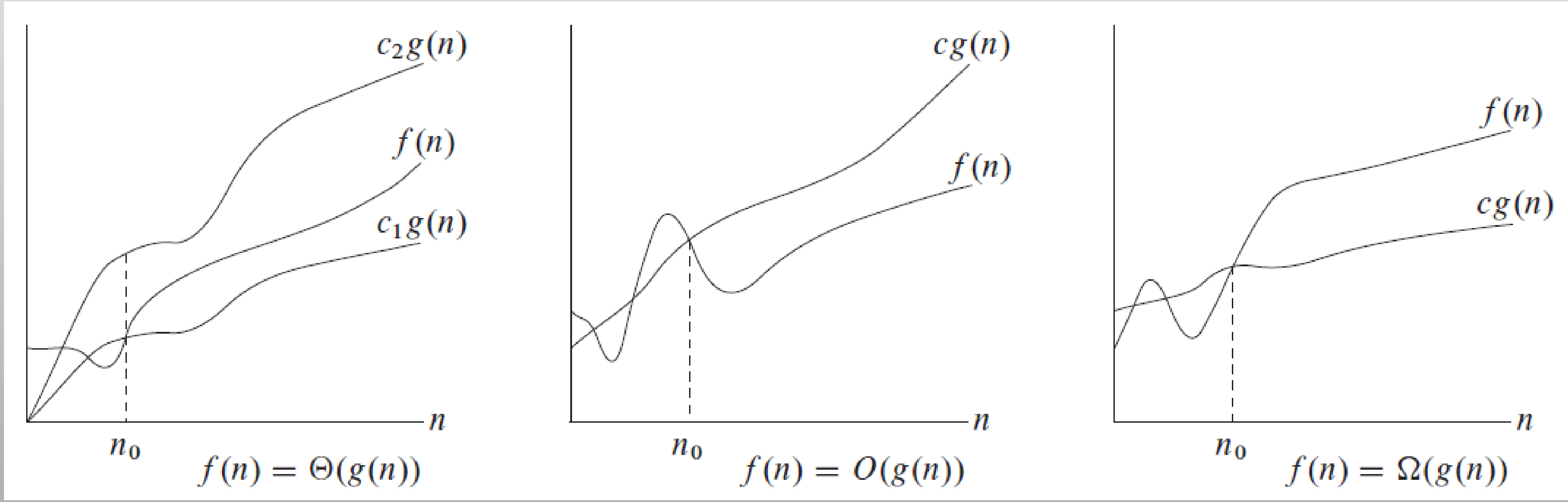
2.3 Notații asimptotice / Notația Ω (omega)

- Precizează marginea asimptotică inferioară
  - $\Omega(g(n))$={f(n): există constante pozitive c>0 și $n_0>0$
    astfel încât 0 <= cg(n)<=f(n)
    pentru orice n>= n_0}

- Utilizare: $f(n)=\Omega(g(n))$, pentru a preciza **cazul cel mai favorabil** al execuției unui algoritm
- Prin implicație, $\Omega(g(n))$ va mărgini inferior TE pentru orice intrare arbitrară a algoritmului indiferent de dimensiunea lui n și structura intrării

- $f(n)=\Theta(g(n))$ doar dacă f(n)=O(g(n) și $f(n)=\Omega(g(n))$

2.3 Notații asimptotice



2.3 Notații asimptotice / Notația o (o mic)

- Precizează marginea asimptotica superioară **lejeră**
  - $o(g(n))=\{f(n)$: pentru orice constantă $c>0$ există o constantă $n_0>0$
    astfel încât $0<=f(n)<cg(n)$
    pentru orice $n>=n_0\}$

- $f(n)=o(g(n))$ dacă $f(n)=O(g(n))$ și $f(n)!=\Theta(g(n))$

- În cazul notației o (o mic) funcția f devine nesemnificativă în raport cu g când $n \rightarrow$infinit
- lim $f(n)/g(n) = 0$ pentru $n \rightarrow$ infinit

2.3 Notații asimptotice / Notația ω (omega mic)

- Precizează marginea asimptotică inferioară **lejeră**
  - $\omega(g(n))$={f(n): pentru orice constantă c>0 există o constantă $n_0>0$
    astfel încât $0 <= cg(n) < f(n)$
    pentru orice $n >= n_0$}

- f(n) devine din ce în ce mai semnificativă față de g(n) pe măsura ce n crește
- lim f(n)/g(n) = infinit pentru n $\rightarrow$ infinit

Proprietati generale ale notațiilor asimptotice

- Tranzitivitate
  - $f(n)=\Theta(g(n))$ și $g(n)=\Theta(h(n)) \rightarrow f(n)=\Theta(h(n))$
  - $f(n)=O(g(n))$ și $g(n)=O(h(n)) \rightarrow f(n)=O(h(n))$
  - …
- Reflexivitate
  - $f(n)=O(f(n))$
  - $f(n)=\Theta(f(n))$
- Simetria
  - $f(n)=\Theta(g(n)) \rightarrow g(n)=\Theta(f(n))$
- Simetria transpusa
  - $f(n)=O(g(n))$ $g(n)=\Omega(f(n))$
  - $f(n)=o(g(n))$ $g(n)=\omega(f(n))$

Aprecierea timpului de execuție al algoritmilor

- **Notația asimptotică O este utilizată pentru aprecierea timpului de execuție al unui algoritm** (timpul de execuție absolut, complexitate temporală)
- Un algoritm a cărui complexitate temporală este spre exemplu $O(n^2)$ va rula ca program întotdeauna în $O(n^2)$ unități de timp, indiferent de natura implementării sale
- **Timpul de execuție al unui program depinde de**:
  - dimensiunea și structura datelor de intrare
  - caracteristicile sistemului de calcul
  - eficiența codului generat

Aprecierea timpului de execuție al algoritmilor

➢ **Considerații**:
- fiecare instrucțiune se execută în același interval de timp (unit. de timp)
- numărul de ciclări pentru instrucțiunile de ciclare este maxim
- instrucțiunile condiționale se execută pe ramura cea mai lungă
- instrucțiunile consecutive produc adunarea timpilor de execuție
- instrucțiunile de ciclare imbricate $\rightarrow$ produs

➢ Cu regulile precizate anterior, timpul de execuție este **supraestimat** în unele situații, dar nu este niciodată **subestimat**

➢ O strategie generală pentru calculul timpului de execuție al algoritmilor impune analiza de la „interior spre exterior”
- de exemplu dacă algoritmul conține apeluri de funcții, timpul de execuție al acelor funcții trebuie analizat mai întâi

➢ În cazul apelurilor recursive se încearcă fie transformarea recursivității într-o iterație (structură de buclă), fie găsirea unei relații de recurență

Profilul performanței algoritmului

- Profilarea este o activitate prin care se determină **exact** timpii de execuție a algoritmului pentru o anumită implementare și pentru seturi diferite de date de intrare
- Profilul performanței **confirmă** aprecierea timpului de execuție realizată în **analiza apriorică**
  - pentru aceasta se vor furniza seturi de date de intrare de **dimensiuni** din ce în ce mai mari, respectiv seturi de date de intrare corespunzătoare cazurilor „**cel mai favorabil**” și „**cel mai defavorabil**”
- Prin compararea rulării pe același sistem de calcul a mai multor algoritmi care realizează aceeași funcție se face o analiză a **performanței algoritmilor**
- Prin compararea rulării aceluiași algoritm pe sisteme de calcul diferite se poate face o analiză a **performanței sistemului**

Universitatea
Politehnica
Timişoara

P-ţa Victoriei nr. 2
RO 300006 - Timişoara
Tel: +4 0256 403000
Fax: +4 0256 403021
rector@rectorat.upt.ro
www.upt.ro

2. Noțiuni despre algoritmi

3. Tehnici de sortare (partea întâia)

3.1 Conceptul de sortare / Sortarea tablourilor

- Sortarea este o activitate fundamentală cu caracter universal
- Prin sortare se înțelege **ordonarea după un criteriu precizat** a unei mulțimi de elemente, cu scopul facilitării operației de căutare a unui element dat
- Algoritmii de sortare
  - subliniază **interdependența** între structura de date și modul de proiectare a algoritmului
  - permit **analiza comparativă** a algoritmilor, cu evidențierea avantajelor și dezavantajelor asociate diferitelor implementări
- Structura de date supusă sortării:
  - tablou de elemente (structurate sau nu, cu sau fără câmp cheie)
- Operația de sortare constă în permutarea elementelor tabloului astfel încât să fie satisfăcută relația de ordonare (ordine crescătoare, descrescătoare, cu sau fără egalitate)
  - ex: $a_1 <= a_2 <= a_3 <= \ldots <= a_n$

3.1 Conceptul de sortare / Sortarea tablourilor

- O metodă de sortare este **stabilă** dacă în urma procesului de sortare elementele identice (cheile identice, dacă se utilizează un câmp cheie) nu își schimbă ordinea relativă
- Funcție de locul în care se afla elementele de sortat rezultă următoarele categorii:
  - sortare **internă** – elementele de sortat sunt stocate în memoria interna
  - sortare **externă** – elementele de sortat sunt stocate pe suport extern
- O **cerința impusă** algoritmilor de sortare legată de eficiență este utilizarea în procesul de sortare a unei zone de memorie cât mai redusă
- Algoritmii care nu utilizează zone suplimentare de memorie realizează o sortare „in situ” (pe loc)

3.1 Conceptul de sortare / Sortarea tablourilor

- Elementele care influențează **performanța** algoritmilor de sortare, respectiv **timpul de execuție** a acestora sunt:
  - numărul de comparații C
  - numărul de mișcări M

dependente la rândul lor de **dimensiunea n** a tabloului (nr. elementelor de sortat)

- Funcție de complexitatea temporală, metodele de sortare sunt:
  - **directe**
    - simple, potrivite pentru a explicita mecanismele de sortare $\rightarrow O(n^2)$
  - **avansate**
    - conduc prin implementare la mai puține operații
    - sunt mai complexe în detalii
    - își dovedesc utilitatea pentru valori mari ale lui n $\rightarrow O(n\lg n)$

3.1 Conceptul de sortare / Sortarea tablourilor

- **Metodele de sortare directe** au la bază **principiile** (pt. ordonare crescătoare):
  - **inserție** – se ia un element de sortat la întâmplare (până la epuizarea elementelor supuse sortării) și se inserează în structura sortată la locul potrivit
  - **selecție** – se parcurg toate elementele, se alege cel mai mic și se plasează pe prima poziție; se reia algoritmul pentru cele rămase, până la epuizarea elementelor supuse sortării
  - **schimburi** (interschimbare) – în toate perechile de elemente care se pot imagina, se plasează elementul cel mai mic pe primul loc

3.2 Tehnica sortării prin inserție

- Elementele de sortat $a_1, a_2, \ldots, a_{i-1}$, $a_i, a_{i+1}, \ldots, a_n$
  - sunt în mod conceptual divizate în 2 secvențe:
  - secvența destinație $a_1, a_2, \ldots, a_{i-1}$
  - secvența sursă $a_i, a_{i+1}, \ldots, a_n$
- Selecția locului de inserție pentru elementul a_i se va face parcurgând secvența destinație de la dreapta la stânga, oprirea realizându-se pe primul element $a_j \le a_i$ sau pe prima poziție, dacă un astfel de element a_j nu este găsit
- Simultan cu parcurgerea secvenței destinație pentru căutarea locului de inserție se face și deplasarea la dreapta a fiecărui element testat, până la îndeplinirea condiției

3.2 Tehnica sortării prin inserție

➢ Exemplu:

- sortarea în ordine crescătoare a unui șir de caractere format doar din litere mici

```
void insertSort(char *s, int n){
int i, j;
char t;
     for (i=1; i<n; ++i){
          t=s[i];
          j=i-1;
          while(j>=0 && t<s[j]){
               s[j+1]=s[j];
               j--;
          }
          s[j+1]=t;
     }
}
```

d	a	m	x	b	d	n
a	d	m	x	b	d	n
a	d	m	x	b	d	n
a	d	m	x	b	d	n
a	b	d	m	x	d	n
a	b	d	d	m	x	n
a	b	d	d	m	n	x

3.2 Tehnica sortării prin inserție – analiză

- Pentru pasul i al ciclului for numărul de comparații asociat C_i depinde de ordinea inițială a elementelor, fiind:
  - cel puțin 1
  - cel mult i-1
  - presupunând că toate permutările sunt în mod egal posibile, C_i poate fi considerat pentru cazul mediu ca fiind i/2
  - Ciclul for se va repeta de n-1 ori pentru i=1, …, n-1, rezultând astfel C_{min}, C_{max} și C_{med}
- Numărul de mișcări M_i realizat în pasul i al ciclului for are două componente:
  - C_i componenta determinată de bucla while
  - 2 sau 3 atribuiri – cele exterioare ciclului while
- M_{min}, M_{max} și M_{med} se calculează considerând cele n-1 repetări ale ciclului for
- Sortarea prin inserție este stabilă
- **Valorile maxime** pt. C și M se obțin când șirul inițial este ordonat invers
- **Performantele sunt scăzute** deoarece deplasarea elementelor se realizează de fiecare dată cu o singură poziție

3.2 Tehnica sortării prin inserție

<u>Algoritm de inserție binară</u>

- Secvența destinație este deja o structură ordonată
  - => căutarea locului inserției se poate face aplicând tehnica de **căutare binară**
- Căutarea binară **reduce numărul de comparații**:
  - într-un interval de i chei, locul pentru inserarea elementului va fi găsit după $\log_2 i$ pași (rotunjit superior)
  - pentru cele n-1 repetări ale ciclului for: $C=O(n\log_2 n)$
- Numărul mișcărilor însă rămâne nemodificat: $O(n^2)$
- Performanța algoritmului de inserție binară **rămâne** în domeniul $O(n^2)$
  - costul operației de interschimbare este mai mare decât cel necesar comparației a două elemente
  - **performantă** atunci când șirul inițial este ordonat invers
  - **performanțe scăzute** față de inserția prin căutare liniară dacă șirul inițial este deja sortat

3.3 Tehnica sortării prin selecție

- Principiul constă în căutarea într-o secvență a elementului cu cheie minimă (a_k) și plasarea lui pe prima poziție (a_{i+1}) prin interschimbare
- Procedeul se reia pentru cele n-1, n-2, …1 elemente rămase

- $a_1, a_2, \dots a_i, a_{i+1}, \dots, a_k, \dots a_n$

- Este o metodă opusă ca principiu sortării prin inserție, căutarea executându-se în secvența sursă

3.3 Tehnica sortării prin selecție

```
void selectSort(char *s, int n){
     char t;
     int i, j, k;
     for (i=0; i<n-1; ++i){
          k=i;
          t=s[i];
          for(j=i+1; j<n; ++j){
               if(s[j]<t){
                    k=j;
                    t=s[j];
               }
          }
          s[k]=s[i];
          s[i]=t;
     }
}
```

d a m x b d n

a d m x b d n

a b m x d d n

a b d x m d n

a b d d m x n

a b d d m x n

a b d d m n x

3.3 Tehnica sortării prin selecție – analiză

- În pasul i al ciclului for se vor executa i - 1 comparații
  - numărul comparațiilor C este **independent de ordinea inițială**, metoda comportându-se mai puțin natural decât sortarea prin inserție
- Ciclul for se va repeta pentru i = 1, 2, …, n-1, rezultând astfel:
  - $C_{min} = C_{max} = C = (n-1)(n-2)/2 \Rightarrow O(n^2)$
- Numărul mișcărilor M este de cel puțin 3 pentru fiecare valoare a lui i
  - acest minim poate să apară dacă elementele sunt deja sortate
    - $M_{min} = 3(n-1)$
  - dacă elementele sunt inițial în ordine inversă, numărul de mișcări M este maxim:
    - $M_{max} = n^2/4$ (rotunjit inferior)$+3(n-1)$
  - M_{med} se obține printr-un raționament probabilistic care ia in considerare toate permutările posibile asociate unui număr de elemente
    - pentru fiecare permutare numărul mișcărilor este determinat de numărul elementelor având proprietatea de a fi mai mici decât termenii precedenți

3.3 Tehnica sortării prin selecție – analiză

- În procesul de sortare se parcurg secvențe de lungimi n, n-1, n-2, ..., 1
  - în urma însumării mișcărilor de pe fiecare secvență va rezulta:
  - M_{med}= n ln n => O(nlnn)

- Performanța în raport cu numărul de mișcări situează sortarea prin selecție pe **primul loc** în ierarhia algoritmilor de sortare directă

- O varianta îmbunătățită a sortării prin selecție se obține dacă se determină doar poziția minimului, eliminându-se astfel atribuirea aferentă instrucțiunii condiționale => <u>sortarea prin selecție performantă</u>:
  - M_{min}= 3(n-1)

3.4 Tehnica sortării prin interschimbare

- Principiul metodei:
  - se compară pe rând și se interschimbă între ele toate elementele alăturate, până când tabloul este în întregime sortat
- Sortarea se va realiza într-o maniera ordonata, executând **treceri repetate** prin tablou
  - în cazul sortării crescătoare, la fiecare trecere, cel mai mic element al secvenței procesate se va deplasa spre capătul din stânga
- Pentru o mulțime de n elemente supuse sortării sunt necesare n-1 treceri prin tablou, asigurate de un for exterior
- Bucla interioară va executa comparațiile și interschimbările, dacă acestea sunt necesare
- Se pot evidenția și aici cele două secvențe conceptuale, sursă și destinație
- In pasul k se vor compara de fiecare dată n-k elemente
  - după k pași, k elemente vor fi deja ordonate
  - tabloul va fi în întregime ordonat după n-1 pași

3.4 Tehnica sortării prin interschimbare

```
void bubbleSort(char *s, int n){
int i, j;
char t;
     for(i=1;i<n;++i)
          for(j=n-1;j>=i;--j){
               if(s[j-1]>s[j]){
                    t=s[j-1];
                    s[j-1]=s[j];
                    s[j]=t;
               }
          }
}
```

d	a	m	x	b	d	n
d	a	m	b	x	d	n
d	a	b	m	x	d	n
a	d	b	m	x	d	n
a	d	b	m	d	x	n
a	d	b	d	m	x	n
a	b	d	d	m	x	n
a	b	d	d	m	n	x
a	b	d	d	m	n	x
a	b	d	d	m	n	x
a	b	d	d	m	n	x

Deși în acest punct tabloul este sortat, algoritmul continuă verificările

3.4 Tehnica sortării prin interschimbare

- În multe cazuri sortarea se termină înainte de a epuiza toate reluările precizate prin bucla for exterioară
  - o îmbunătățire se obține dacă reluarea ciclului este condiționată de efectuarea unei schimbări în pasul precedent
- Altă îmbunătățire se obține dacă se memorează indicele k al ultimei schimbări
  - toate elementele aflate sub acest indice fiind deja sortate
- Algoritmul prezintă o asimetrie în comportament:
  - un element „ușor” (valoare mică în cazul sortării crescătoare), aflat pe poziția inferioară, ajunge la locul lui într-o singura trecere
  - un element „greu” (valoare mare în cazul sortării crescătoare), aflat pe prima poziție a tabloului are nevoie de n-1 treceri pentru a ajunge la locul potrivit
- Ex: 83 12 18 22 24 04

3.4 Tehnica sortării prin interschimbare

➢ Sortare amestecată

```
void shakerSort(char *s, int n){
int j, k, l, r;
char t;
  l=1; r=n-1; k=n-1;
  do{
    for(j=r;j>=l;--j){
      if(s[j-1]>s[j]){
        t=s[j-1];s[j-1]=s[j];s[j]=t;
        k=j;
        /*memoreaza ultima inversare*/
      }
    }
    l=k+1;
    for(j=l;j<=r;++j){
      if(s[j-1]>s[j]){
        t=s[j-1];
        s[j-1]=s[j];
        s[j]=t;
        k=j;
      }
    }
    r=k-1;
  }while(l<=r);
}
```

3.4 Tehnica sortării prin interschimbare

➢ <u>Sortare amestecată</u>

r->l	d	a	m	x	b	d	n
r->l	d	a	m	b	x	d	n
r->l	d	a	b	m	x	d	n
l->r	a	d	b	m	x	d	n
l->r	a	b	d	m	x	d	n
l->r	a	b	d	m	d	x	n
r->l	a	b	d	m	d	n	x
r->l	a	b	d	d	m	n	x
l->r	a	b	d	d	m	n	x

3.4 Tehnica sortării prin interschimbare – analiză

- **Bubblesort:**
  - Numărul de comparații C_i efectuate în pasul i este i-1:
    - $C=1+2+3+\ldots+(n-2)=(n^2-3n+2)/2 \Rightarrow O(n^2)$
  - Numărul de mișcări M depinde de starea de ordonare inițială:
    - $M_{min} = 0$
    - $M_{max} = 3C = 3(n^2-3n+2)/2$
    - $M_{med} = 3(n^2-3n+2)/4$
- **Shakersort:**
  - Se reduce numărul de comparații:
    - $C_{min} = n-1$; $C_{med} = 1/2(n^2 - n(k + \ln n)) \Rightarrow O(n^2)$
  - Numărul de interschimbări rămâne același => $O(n^2)$
  - Distanța parcursă de fiecare element este în medie n/3 locuri

Universitatea
Politehnica
Timişoara

P-ţa Victoriei nr. 2
RO 300006 - Timişoara
Tel: +4 0256 403000
Fax: +4 0256 403021
rector@rectorat.upt.ro
www.upt.ro

Vă mulțumesc!