Curs 8: Aplicaţii ale tehnicii divizării. Sortarea prin interclasare şi sortarea rapidă.
- Problematica
- Sortare prin interclasare
- Sortare rapidă
- Exerciţii

1 Problematica

Considerăm din nou problema ordonării crescătoare a unui şir de valori; $(x_1, x_2, \ldots, x_n)$. Algoritmii elementari prezentaţi anterior au complexitate de ordinul $O(n^2)$. Ideea de start o reprezintă încercarea de a reduce această complexitate folosind principiul "divide et impera". Aceasta presupune: (i) descompunerea şirului iniţial în două subşiruri; (ii) ordonarea fiecăruia dintre acestea folosind aceeaşi tehnică; (iii) combinarea şirurilor ordonate parţiale pentru a obţine varianta ordonată a şirului total.

Dimensiunea critică, sub care problema poate fi rezolvată direct este $n_c = 1$. În acest caz subşirul se reduce la un singur element, implicit ordonat. Este posibil să se folosească şi $1 < n_c \leq 10$ aplicând pentru sortarea acestor subşiruri una dintre metodele elementare de sortare (de exemplu metoda inserţiei).

În continuare sunt prezentate două metode de sortare bazate pe principiul divizării: sortarea prin interclasare ("mergesort") şi sortarea rapidă ("quicksort"). Acestea diferă atât în etapa descompunerii şirului în subşiruri cât şi în recombinarea rezultatelor.

2 Sortare prin interclasare

Idee. Şirul $(x_1, x_2, \ldots, x_n)$ se descompune în două subşiruri de lungimi cât mai apropiate: $(x_1, \ldots, x_{\lfloor n/2 \rfloor})$ şi $(x_{\lfloor n/2 \rfloor + 1}, \ldots, x_n)$; se ordonează fiecare dintre subşiruri aplicând aceeaşi tehnică; se construieşte şirul final ordonat parcurgând cele două subşiruri şi preluând elemente din ele astfel încât să fie respectată relaţia de ordine (această prelucrare se numeşte interclasare). Modul de lucru al sortării prin interclasare este ilustrat în figura 1.

Structura generală a algoritmului.

```
mergesort(x[li..ls])
IF li < ls THEN
    ||m = ⌊(li + ls)/2⌋
    ||x[li..m] ← mergesort(x[li..m])
    ||x[m + 1..ls] ← mergesort(x[m + 1..ls])
    ||x[li..ls] ← merge(x[li..m], x[m + 1..ls])
RETURN x[1..n]
```

Interclasare. Etapa cea mai importantă a prelucrării este reprezentată de interclasare. Scopul acestei prelucrări este construirea unui şir ordonat pornind de la două şiruri ordonate. Ideea prelucrării constă în a parcurge în paralel cele două şiruri şi a compara elementele curente. În şirul final se tranferă elementul mai mic dintre cele două iar contorul utilizat pentru parcurgerea şirului din care s-a transferat un element este incrementat. Procesul continuă până când unul dintre şiruri a fost transferat în întregime. Elementele celuilalt şir sunt transferate direct în şirul final.

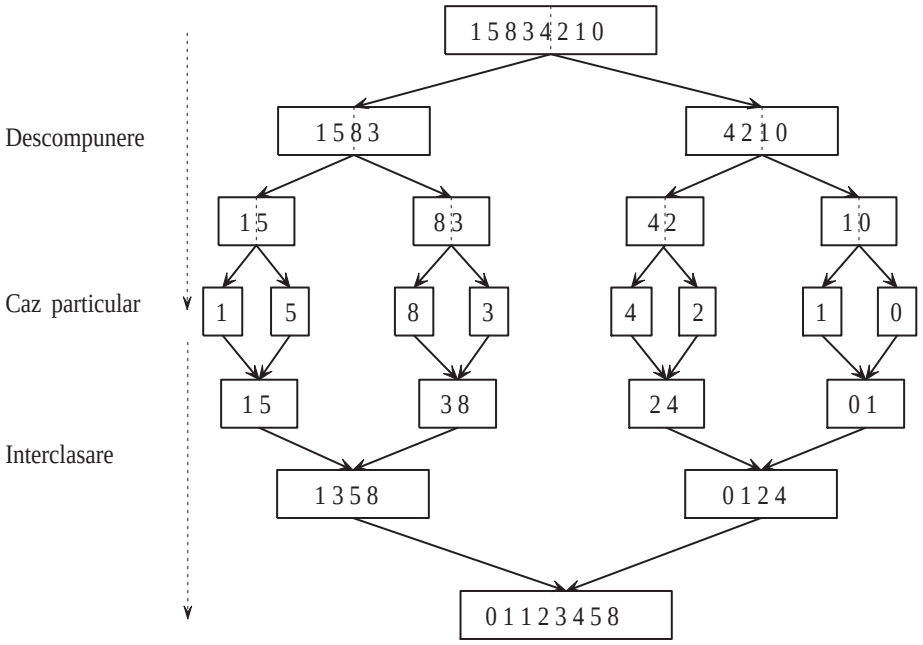

Figura 1: Sortare prin interclasare

Există diverse modalităţi de a descrie algoritmic această prelucrare. Una dintre acestea este:

---

merge($x[li..m]$,$x[m+1..ls]$)
$i \leftarrow li$; $j \leftarrow m+1$; $k \leftarrow 1$ // iniţializarea contoarelor
WHILE $i \leq m \wedge j \leq ls$ DO // parcurgerea până la sfârşitul unuia dintre tablouri
  ‖IF $x[i] \leq x[j]$ THEN // transfer din tabloul a
  ‖ ‖$c[k] \leftarrow x[i]$; $i \leftarrow i+1$; $k \leftarrow k+1$
  ‖ELSE // transfer din tabloul b
  ‖ ‖$c[k] \leftarrow x[j]$; $j \leftarrow j+1$; $k \leftarrow k+1$
WHILE $i \leq m$ DO // transferul eventualelor elemente rămase în a
  ‖$c[k] \leftarrow x[i]$; $i \leftarrow i+1$; $k \leftarrow k+1$
WHILE $j \leq ls$ DO // transferul eventualelor elemente rămase în b
  ‖$c[k] \leftarrow x[j]$; $j \leftarrow j+1$; $k \leftarrow k+1$
RETURN $c[1..k-1]$

Precondiţiile prelucrării de mai sus sunt: $\{x[li..m]$ crescător $\wedge$ $x[m+1..ls]$ crescător $\}$ iar postcondiţia este: $\{c[1..p+q]\}$ este crescător $\}$ (notăm cu p numărul de elemente din primul tablou şi cu q numărul de elemente din al doilea). Pentru a demonstra că algoritmul asigură satisfacerea postcondiţiei este suficient să se arate că $\{c[1..k-1]$ este crescător $\wedge k = i+j-1\}$ este proprietate invariantă pentru fiecare dintre cele trei prelucrări repetitive.

Contorizând numărul de comparaţii ($T_C(p,q)$) şi cel de transferuri ale elementelor ($T_M(p,q)$) se obţine: $T_C(p,q) \in \Omega(\min(p,q))$ (în cazul cel mai favorabil), $T_C(p,q) \in O(p+q-1)$ (în cazul cel mai defavorabil) iar $T_M(p,q) \in p+q$.

Să analizăm cazul interclasării a două tablouri $a[1..p]$ şi $b[1..q]$ (independent de algoritmul de sortare prin interclasare). O variantă de algoritm este cea care foloseşte câte un fanion pentru

fiecare dintre tablourile $a[1..p]$ şi $b[1..q]$. Fanioanele constau în valori mai mari decât toate valorile prezente în a şi b plasate pe poziţiile $p+1$ respectiv $q+1$. În acest caz algoritmul poate fi descris într-o formă mai condensată după cum urmează:

```
interclasare_fanion(a[1..p+1],b[1..q+1])
a[p+1] ← ∞; b[q+1] ← ∞; // fixarea fanioanelor
i ← 1; j ← 1; //iniţializarea indicilor de parcurgere
FOR k ← 1, p+q DO// parcurgerea poziţiilor în tabloul final
    ||IF a[i] ≤ b[j] THEN // preluare din a
    ||    ||c[k] ← a[i]
    ||    ||i ← i+1
    ||ELSE // preluare din b
    ||    ||c[k] ← b[j]
    ||    ||j ← j+1
RETURN c[1..p+q]
```

În acest caz atât numărul de comparaţii cât şi numărul de transferuri este $T_C(p,q) = T_M(p,q) = p+q$. Prin urmare în varianta cu fanion se efectuează mai multe comparaţii decât în cea fără fanion.
Analiza complexităţii. Notând cu $T(n)$ numărul de prelucrări (comparaţii şi transferuri) efectuate de către sortarea prin interclasare şi cu $T_{inter}(n)$ efectuate pe parcursul interclasării se obţine relaţia de recurenţă:

$$T(n) = \begin{cases} 0 & n = 1 \\ T(\lfloor n/2 \rfloor) + T(n - \lfloor n/2 \rfloor) + T_{inter}(n) & n > 1 \end{cases}$$

Cum $k = m = 2$ iar $T_{inter}(n) \in \Theta(n)$ rezultă că se poate aplica cazul doi al teoremei master ($d = 1$ astfel că $k = m^d$) obţinându-se că sortarea prin interclasare are complexitate $\Theta(n \lg n)$.
Observaţii. Costul redus al prelucrărilor din *mergesort* este însă contrabalansat de faptul că se utilizează (la interclasare) o zonă de manevră de dimensiune proporţională cu cea a tabloului iniţial.

3 Sortare rapidă

Idee. La sortarea prin interclasare descompunerea şirului iniţial în două subşiruri se realizează pe baza poziţiei elementelor. Asta face ca elemente cu valori mici (sau mari) să se poată afla în fiecare dintre subşiruri. Din această cauză este necesară combinarea rezultatelor prin interclasare. O altă modalitate de descompunere ar fi aceea în care se ţine cont şi de valoarea elementelor. Scopul urmărit este de a simplifica combinarea rezultatelor. Ideal ar fi ca prin concatenarea şirurilor ordonate să se obţină şirul ordonat în întregime.
Exemplul 1. Considerăm şirul $x = (3, 1, 2, 4, 7, 5, 8)$. Se observă că elementul 4 se află pe o poziţie privilegiată ($q = 4$) întrucât toate elementele care îl preced sunt mai mici decât el şi toate cele care îl succed sunt mai mari. Aceasta înseamnă că se află deja pe poziţia finală. Un element $x[q]$ având proprietăţile: $x[i] \leq x[q]$ pentru $i = \overline{1, q-1}$ şi $x[i] \geq x[q]$ pentru $i = \overline{q+1, n}$ se numeşte *pivot*. Existenţa unui pivot permite reducerea sortării şirului iniţial la sortarea subşirurilor $x[1..q-1]$ şi $x[q+1..n]$. Nu întotdeauna există un astfel de pivot după cum se observă din subşirul $(3, 1, 2)$. În aceste situaţii trebuie "creat" unul prin modificarea poziţiilor unor elemente. Alteori pivotul, dacă există, se află pe o poziţie care nu permite descompunerea problemei iniţiale în subprobleme de dimensiuni apropiate (de exemplu în subşirul $(7, 5, 8)$ valoarea de pe ultima poziţie poate fi considerată valoare pivot).

Exemplul 2. Să considerăm acum şirul $(3, 1, 2, 7, 5, 4, 8)$. Se observă că poziţia $q = 3$ are următoarea proprietate: $x[i] \leq x[j]$ pentru orice $i \in \{1, \ldots, q\}$ şi orice $j \in \{q+1, \ldots, n\}$. În acest caz sortarea şirului iniţial se reduce la sortarea subşirurilor $x[1..q]$ şi $x[q+1..n]$. Poziţia q este numită poziţie de partiţionare.

Structura generală. Pornind de la exemplele de mai sus se pot dezvolta două variante de sortare bazate pe descompunerea şirului iniţial în două subşiruri: una care foloseşte un element pivot iar alta care foloseşte o poziţie de partiţionare. Acest tip de sortare este cunoscută sub numele de sortare rapidă ("quicksort") şi a fost dezvoltată de Hoare. Structura generală a acestor algoritmi poate fi descrisă prin:

---

quicksort1 $(x[li..ls])$
IF $li < ls$ THEN
  $\|q \leftarrow partitie1(x[li..ls]$
  $\|x[1..q-1] \leftarrow quicksort1(x[1..q-1])$
  $\|x[q+1..n] \leftarrow quicksort1(x[q+1..n])$
RETURN $x[li..ls]$

---

respectiv

---

quicksort2 $(x[li..ls])$
IF $li < ls$ THEN
  $\|q \leftarrow partitie2(x[li..ls]$
  $\|x[1..q] \leftarrow quicksort1(x[1..q])$
  $\|x[q+1..n] \leftarrow quicksort1(x[q+1..n])$
RETURN $x[li..ls]$

Diferenţa dintre cele două variante este mică: în prima variantă în prelucrarea de partiţionare se asigură şi plasarea pe poziţia q a valorii finale pe când în a doua doar se determină poziţia de partiţionare.

Partiţionare. Este prelucrarea cea mai importantă a algoritmului. Discutăm separat cele două variante deşi după cum se va vedea diferenţele dintre ele sunt puţine.

Considerăm problema identificării în $x[1..n]$ a unui pivot, $x[q]$ cu proprietatea că $x[i] \leq x[q]$ pentru $i < q$ şi $x[i] \geq x[q]$ pentru $i > q$. După cum s-a văzut în exemplul anterior nu întotdeauna există un pivot. În aceste situaţii se crează unul. Ideea este următoarea: se alege o valoare arbitrară dintre cele prezente în şir şi se rearanjează elementele şirului (prin interschimbări) astfel încât elementele mai mici decât această valoare să fie în prima parte a şirului iar cele mai mari în a doua parte a şirului. În felul acesta se determină şi poziţia q pe care trebuie plasată valoarea.

Algoritmul poate fi descris astfel:

---

partitie1$(x[li..ls])$
$v \leftarrow x[ls]$ // se alege valoarea pivotului
$i \leftarrow li - 1$ // contor pentru parcurgere de la stânga la dreapta
$j \leftarrow ls$ // contor pentru parcurgere de la dreapta la stânga
WHILE $i < j$ DO
  $\|$REPEAT $i \leftarrow i + 1$ UNTIL $x[i] \geq v$
  $\|$REPEAT $j \leftarrow j - 1$ UNTIL $x[j] \leq v$
  $\|$IF $i < j$ THEN $x[i] \leftrightarrow x[j]$
$x[i] \leftrightarrow x[ls]$
RETURN i

Exemplul 1. Considerăm şirul $(1, 7, 5, 3, 8, 2, 4)$. Iniţial $i = 0$, $j = 7$ iar $v = 4$. Succesiunea prelucrărilor este:
Etapa 1. După execuţia primului ciclu REPEAT se obţine $i = 2$ (căci $7 > 4$) iar dupa execuţia celui de al doilea se obţine $j = 6$. Cum $i < j$ se interschimbă $x[2]$ cu $x[6]$ obţinându-se $(1, 2, 5, 3, 8, 7, 4)$.
Etapa 2. Parcurgerea în continuare de la stânga la dreapta conduce la $i = 3$ iar de la dreapta la stânga la $j = 4$. Cum $i < j$ se interschimbă $x[3]$ cu $x[4]$ şi se obţine $(1, 2, 3, 5, 8, 7, 4)$.
Etapa 3. Continuând parcurgerile în cele două direcţii se ajunge la $i = 4$ şi $j = 3$.
Cum $i > j$ se iese din prelucrarea repetitivă exterioară, iar elementele $x[4]$ şi $x[7]$ se interschimbă obţinându-se $(1, 2, 3, 4, 8, 7, 5)$, poziţia pivotului fiind 4. Această poziţie asigură o partiţionare echilibrată a şirului. Această situaţie nu este însă obţinută întotdeauna.
Exemplul 2. Să considerăm şirul $(4, 7, 5, 3, 8, 2, 1)$. Aplicând acelaşi algoritm după prima etapă se obţine: $i = 1$, $j = 1$ (a se observa că în acest caz fără a folosi o poziţie suplimentară cu rol de fanion, $x[0] = v$ condiţia de oprire a celui de-al doilea REPEAT nu este niciodată satisfăcută), şirul devine $(1, 7, 5, 3, 8, 2, 4)$ iar poziţia pivotului este $q = 1$. În acest caz s-a obţinut o partiţionare dezechilibrată (şirul se descompune într-un subşir vid, pivotul aflat pe prima poziţie şi un subşir constituit din celelalte elemente). Se poate remarca că pentru primul ciclu REPEAT $x[ls]$ joacă rolul unui fanion.
Observaţii. (i) Inegalităţile de tip $\geq$ şi $\leq$ din ciclurile REPEAT fac ca în cazul unor valori egale să se obţină o partiţionare echilibrată şi nu una dezechilibrată cum s-ar întâmpla dacă s-ar folosi inegalităţi stricte. În acelaşi timp dacă s-ar utiliza $>$ şi $<$ valoarea v nu ar putea fi folosită ca fanion.
(ii) Problema nesatisfacerii condiţiei de oprire pentru ciclul după j poate să apară doar în cazul în care $li = 1$ şi atunci când v este cea mai mică valoare din şir. Este de preferat să se folosească valoare fanion decât să se extindă condiţia de oprire de la ciclu (de exemplu cu $j < i$).
(iii) La ieşirea din prelucrarea repetitivă exterioară (WHILE) indicii i şi j satisfac una dintre relaţiile: $i = j$ sau $i = j + 1$. Ultima interschimbare asigură plasarea valorii fanion pe poziţia sa finală.

Pentru a justifica corectitudinea algoritmului *partiţie1* considerăm aserţiunea: {dacă $i < j$ atunci $x[k] \leq v$ pentru $k = \overline{li, i}$ iar $x[k] \geq v$ pentru $k = \overline{j, ls}$ iar dacă $i \geq j$ atunci $x[k] \leq v$ pentru $k = \overline{li, i}$ iar $x[k] \geq v$ pentru $k = \overline{j + 1, ls}\}$. Cum precondiţia este $li < ls$, deci $i < j$ şi postcondiţiile sunt $\{x[k] \leq x[i]$ pentru $k = \overline{1, i - 1}$, $x[k] \geq x[i]$ pentru $k = \overline{i + 1, ls}\}$, se poate arăta că aserţiunea de mai sus este invariantă în raport cu prelucrarea repetitivă WHILE iar după efectuarea ultimei interschimbări implică postcondiţiile.

Considerăm acum cealaltă variantă de partiţionare. Aceasta diferă de prima în special prin faptul că permite ca valoarea pivotului să participe la interschimbări. În plus el nu este plasat la sfârşit pe poziţia finală fiind necesară astfel sortarea subşirurilor $x[li..q]$ şi $x[q + 1..ls]$. Aceasta face să nu mai fie necesară plasarea pe poziţia 0 a unui fanion întrucât se creează în mod natural fanioane pentru fiecare dintre cele două cicluri REPEAT. Algoritmul poate fi descris după cum urmează.

partitie2($x[li..ls]$)
$v \leftarrow x[li]$ // se alege valoarea pivotului
$i \leftarrow li - 1$ // contor pentru parcurgere de la stânga la dreapta
$j \leftarrow ls + 1$ // contor pentru parcurgere de la dreapta la stânga
WHILE $i < j$ DO
  ||REPEAT $i \leftarrow i + 1$ UNTIL $x[i] \geq v$
  ||REPEAT $j \leftarrow j - 1$ UNTIL $x[j] \leq v$
  ||IF $i < j$ THEN $x[i] \leftrightarrow x[j]$
RETURN j

În acest caz se poate arăta că aserţiunea {dacă $i < j$ atunci $x[k] \leq v$ pentru $k = \overline{li, i}$ iar $x[k] \geq v$ pentru $k = \overline{j, ls}$ iar dacă $i \geq j$ atunci $x[k] \leq v$ pentru $k = \overline{li, i-1}$ iar $x[k] \geq v$ pentru $k = \overline{j+1, ls}$} este invariantă în raport cu ciclul WHILE iar la ieşirea din ciclu (când $i = j$ sau $i = j + 1$) implică $x[k] \leq v$ pentru $k = \overline{li, j}$ şi $x[k] \geq v$ pentru $k = \overline{j+1, ls}$ adică postcondiţia $x[k1] \leq x[k2]$ pentru orice $k1 \in \{li, \ldots, j\}$, $k2 \in \{j+1, \ldots, ls\}$.

De remarcat că dacă se alege ca pivot $x[ls]$ varianta *quicksort2* nu va funcţiona corect putând conduce la situaţia în care unul dintre subtablouri este vid (ceea ce provoacă o succesiune nesfârşită de apeluri recursive, întrucât nu se mai reduce dimensiunea problemei). În cazul în care se doreşte ca $x[ls]$ să fie valoarea pivot este necesar ca apelurile recursive să se facă pentru: *quicksort2*($x[li..q-1]$) şi *quicksort2*($x[q..ls]$).

Analiza complexităţii. Analizăm complexitatea variantei *quicksort1*. Pe parcursul partiţionării numărul de comparaţii depăşeşte în general numărul de interschimbări, motiv pentru care vom analiza doar numărul de comparaţii efectuate. Pentru un şir de lungime n numărul de comparaţii efectuate în cele două cicluri REPEAT din *partiţionare1* este $n + i - j$ (i şi j fiind valorile finale ale contoarelor). Astfel, dacă $i = j$ se vor efectua n comparaţii iar dacă $i = j + 1$ se vor efectua $n + 1$ comparaţii.

Influenţa majoră asupra numărului de comparaţii efectuate de către *quicksort1* o are raportul dintre dimensiunile celor două subşiruri obţinute după partiţionare. Cu cât dimensiunile celor două subşiruri sunt mai apropiate cu atât se reduce numărul comparaţiilor efectuate. Astfel cazurile cele mai favorabile corespund partiţionării echilibrate iar cele mai puţin favorabile partiţionării dezechilibrate.

Analiza în cazul cel mai favorabil. Presupunem că la fiecare partiţionare a unui şir de lungime n se obţin două subşiruri de lungimi $\lfloor n/2 \rfloor$ respectiv $n - \lfloor n/2 \rfloor - 1$ iar numărul de comparaţii din partiţionare este n. În aceste condiţii putem considera că marginea superioară a numărului de comparaţii satisface o relaţie de recurenţă de forma $T(n) = 2T(n/2) + n$ ceea ce conduce prin teorema master la o complexitate de ordin $n \lg n$. Prin urmare în cazul cel mai favorabil complexitatea algoritmului *quicksort1* este $\Theta(n \lg n)$.

Analiza în cazul cel mai defavorabil. Presupunem că la fiecare partiţionare se efectuează $n + 1$ comparaţii şi că se generează un subşir vid şi unul de lungime $n - 1$. Atunci relaţia de recurenţă corespunzătoare este $T(n) = T(n-1) + n + 1$. Aplicând metoda iteraţiei se obţine $T(n) = (n+1)(n+2)/2 - 3$. Se obţine astfel că în cazul cel mai defavorabil complexitatea este pătratică, adică sortarea rapidă aparţine clasei $O(n^2)$. Spre deosebire de metodele elementare de sortare pentru care un şir iniţial sortat conducea la un număr minim de prelucrări în acest caz tocmai aceste situaţii conduc la costul maxim (pentru un şir deja ordonat la fiecare partiţionare se obţine pivotul pe ultima poziţie).

Pentru a evita partiţionarea dezechilibrată au fost propuse diverse variante de alegere a valorii pivotului, una dintre acestea constând în selecţia a trei valori din şir şi alegerea ca valoare a pivotului a medianei acestor valori (valoarea aflată pe a doua poziţie în tripletul ordonat).

Analiza în cazul mediu. Se bazează pe ipoteza că toate cele n poziţii ale şirului au aceeaşi probabilitate de a fi selectate ca poziţie pivot (probabilitatea este în acest caz $1/n$). Presupunând că la fiecare partiţionare a unui şir de lungime n se efectuează $n+1$ comparaţii, numărul de comparaţii efectuate în cazul în care poziţia pivotului este q satisface: $T_q(n) = T(q-1) + T(n-q) + n + 1$. Prin urmare numărul mediu de comparaţii satisface:

$$T_m(n) = \frac{1}{n}\sum_{q=1}^{n} T_q(n) = (n+1) + \frac{1}{n}\sum_{q=1}^{n}(T_m(q-1) + T_m(n-q)) = (n+1) + \frac{2}{n}\sum_{q=1}^{n} T_m(q-1).$$

Scăzând între ele relaţiile:

$$nT_m(n) = 2\sum_{q=1}^{n} T_m(q-1) + n(n+1)$$

$$(n-1)T_m(n-1) = 2\sum_{q=1}^{n-1} T_m(q-1) + (n-1)n$$

se obţine relaţia de recurenţă pentru T_m:

$$nT_m(n) = (n+1)T_m(n-1) + 2n$$

Aplicând metoda iteraţiei rezultă:

$$\begin{array}{lcl|l}
T_m(n) & = & \frac{n+1}{n}T_m(n-1) + 2 & \\
T_m(n-1) & = & \frac{n}{n-1}T_m(n-2) + 2 & \cdot\frac{n+1}{n} \\
T_m(n-2) & = & \frac{n-1}{n-2}T_m(n-3) + 2 & \cdot\frac{n+1}{n-1} \\
\vdots & & \vdots & \\
T_m(2) & = & \frac{3}{2}T_m(1) + 2 & \cdot\frac{n+1}{3} \\
T_m(1) & = & 0 &
\end{array}$$

iar prin însumare se obţine:

$$T_m(n) = 2(n+1)(1/n + 1/(n-1) + \ldots + 1/3) + 2 = 2(n+1)\sum_{i=3}^{n}\frac{1}{i} + 2 \simeq 2(n+1)(\ln n - \ln 3) + 2.$$

Prin urmare numărul mediu de comparaţii este de ordinul $2n\ln n \simeq 1.38n\lg n$. Aceasta înseamnă că în cazul mediu sortarea rapidă este doar cu 38% mai costisitoare decât în cazul cel mai favorabil.
Observaţii. Sortarea rapidă nu este nici stabilă şi nici naturală.

4 Exerciţii

1. Să se extindă algoritmul de interclasare pentru un număr arbitrar de şiruri ordonate.
2. Să se studieze stabilitatea sortării prin interclasare. Care este operaţia determinantă în asigurarea stabilităţii ?

3. Descrieţi sortarea prin interclasare în manieră nerecursivă.

4. Propuneţi un algoritm de complexitate $\Theta(n\lg n)$ pentru verifica dacă într-un tablou $x[1..n]$ există cel puţin două elemente egale.

5. Rescrieţi algoritmul *partiţionare1* când valoarea pivotului este prima din şir, când este o valoare arbitrar aleasă din şir şi când se stabileşte cu regula medianei.

6. Construiţi un exemplu prin care să se ilustreze că sortarea rapidă nu este stabilă.

7. Propuneţi un algoritm eficient prin care se reorganizeaza elementele unui tablou astfel încât toate elementele negative să fie înaintea celor pozitive.

8. Modificaţi algoritmii de sortare prin interclasare şi sortare rapidă astfel încât să asigure ordonarea descrescătoare.