Proiectarea algoritmilor
Lucrare de laborator nr. 3

Paradigma *Divide_et_Impera* - Merge_Sort și Quick_Sort

Cuprins

1 Sortare prin interclasare - Merge_Sort

1.1 Descriere

Considerăm cazul când secvența ce urmează a fi sortată este memorată într-un tablou unidimensional. Prin interclasarea a două secvențe sortate se obține o secvență sortată ce conține toate elementele secvențelor de intrare. Ideea este de a utiliza interclasarea în etapa de asamblare a soluțiilor.

în urma rezolvării recursive a subproblemelor rezultă secvențe ordonate și prin interclasarea lor obținem secvența finală sortată. Primul pas constă în generalizarea problemei.

Presupunem că se cere sortarea unei secvențe memorate într-un tablou $\mathtt{a}[\mathtt{p}..\mathtt{q}]$ cu $\mathtt{p}$ și $\mathtt{q}$ variabile libere, în loc de sortarea unei secvențe memorate într-un tablou $\mathtt{a}[0..\mathtt{n}-1]$ cu $\mathtt{n}$ variabilă legată (deoarece este dată de intrare).

Divizarea problemei constă în împărțirea secvenței de sortat în două subsecvențe $\mathtt{a}[\mathtt{p}..\mathtt{m}]$ și $a[\mathtt{m}+1..\mathtt{q}]$, de preferat de lungimi aproximativ egale (de exemplu $\mathtt{m} = \left[\frac{\mathtt{p}+\mathtt{q}}{2}\right]$. Faza de combinare a soluțiilor constă în interclasarea celor două subsecvențe, după ce ele au fost sortate recursiv prin același procedeu.

1.2 Pseudocod

```
procedure mergeSort(a, p, q)
   if (p < q)
      then m ← [(p+q)/2]
           mergeSort(a, p, m)
           mergeSort(a, m+1, q)
           interclasare(a, p, q, m, temp)
           for i ← p to q do
               a[i] ← temp[i-p]
end

procedure interclasare(a, p, q, m, temp)
   i ← p
   j ← m+1
   k ← -1
   while ((i ≤ m) and (j ≤ q)) do
      k ← k+1
      if (a[i] ≤ a[j])
         then temp[k] ← a[i]
              i ← i+1
         else temp[k] ← a[j]
              j ← j+1
   while (i ≤ m) do
      k ← k+1
      temp[k] ← a[i]
      i ← i+1
   while (j ≤ q) do
      k ← k+1
      temp[k] ← a[j]
      j ← j+1
end
```

1.3 Evaluarea algoritmului

Divizare a problemei în subprobleme se face în timpul constant ($O(1)$). Asamblare a soluțiilor: Interclasarea a două secvențe ordonate crescător se face în timpul $O(m_1+m_2)$, unde m_1 și m_2 sunt lungimile celor două secvențe.

Complexitatea timp a algoritmului: Se aplică teorema complexității *Divide_et_Impera* pentru $a = 2, b = 2, k = 1$. Rezultă pentru algoritmul `Merge_Sort` timpul $O(n \log_2 n)$.

Complexitatea spațiu: Algoritmul utilizează $O(n+\log_2 n)$ memorie suplimentară (tabloul auxiliar și stiva cu apelurile recursive).

2 Sortare rapidă - Quick_Sort

2.1 Descriere

Ca și în cazul algoritmului *Merge Sort*, vom presupune că trebuie sortată o secvență memorată într-un tablou a[p..q].

Divizarea problemei constă în alegerea unei valori x din $a[p..q]$ și determinarea prin interschimbări a unui indice k cu proprietățile:

$p \le k \le q$ și $a[k] = x$; $\quad \forall i : p \le i \le k \Rightarrow a[i] \le a[k]$; $\quad \forall j : k < j \le q \Rightarrow a[k] \le a[j]$;

Elementul x este numit *pivot*. în general, se alege pivotul $x = a[p]$, dar nu este obligatoriu. Partiționarea tabloului se face prin interschimbări care mențin invariante proprietăți asemănătoare cu cele de mai sus.

Se consideră două variabile index: `i` cu care se parcurge tabloul de la stânga la dreapta și `j` cu care se parcurge tabloul de la dreapta la stânga. Inițial se ia $i = p+1$ și $j = q$.

Proprietățile menținute invariante în timpul procesului de partiționare sunt:

$$\forall i' : p \le i' < i \Rightarrow a[i'] \le x \tag{1}$$

și

$$\forall j' : j < j' \le q \Rightarrow a[j'] \ge x \tag{2}$$

Presupunem că la momentul curent sunt comparate elementele $\mathtt{a}[\mathtt{i}]$ și $\mathtt{a}[\mathtt{j}]$ cu $i < j$.

Distingem următoarele cazuri:

1. $a[i] \le x$. Transformarea $\mathtt{i} \leftarrow \mathtt{i}+1$ păstrează proprietatea (1).
2. $a[j] \ge x$. Transformarea $\mathtt{j} \leftarrow \mathtt{j}-1$ păstrează proprietatea (2).
3. $a[i] > x > a[j]$. Dacă se realizează interschimbarea $\mathtt{a}[\mathtt{i}] \leftrightarrow \mathtt{a}[\mathtt{j}]$ și se face $\mathtt{i} \leftarrow \mathtt{i}+1$ și $\mathtt{j} \leftarrow \mathtt{j}-1$, atunci sunt păstrate ambele predicate (1) și (2).

Operațiile de mai sus sunt repetate până când i devine mai mare decât j.

Considerând $k = i-1$ şi interschimbând $\mathtt{a}[\mathtt{p}]$ cu $\mathtt{a}[\mathtt{k}]$ obținem partiționarea dorită a tabloului.

După sortarea recursivă a sub-tablourilor $\mathtt{a}[\mathtt{p}..\mathtt{k}-1]$ și $\mathtt{a}[\mathtt{k}+1..\mathtt{q}]$ se observă că tabloul este sortat deja. Astfel partea de asamblare a soluțiilor este vidă.

2.2 Pseudocod

```
procedure quickSort1(a, p, q)
   if (p < q)
      then / * determină prin interschimbări indicele k pentru care:
                 p ≤ k ≤ q
                 (∀)i : p ≤ i ≤ k ⇒ a[i] ≤ a[k]
                 (∀)j : k < j ≤ q ⇒ a[k] ≥ a[j]  */
           partitioneaza(a, p, q, k)
           quickSort1(a, p, k-1)
           quickSort1(a, k+1, q)
end

procedure partitioneaza(a, p, q, k)
   x ← a[p]
   i ← p + 1
   j ← q
   while (i ≤ j) do
      if (a[i] ≤ x then i ← i + 1
      if (a[j] ≥ x then j ← j - 1
      if (i < j)
         then if ((a[i] > x) and (x > a[j]))
                 then interschimba(a[i],a[j])
                      i ← i + 1
                      j ← j - 1
   k ← i-1
   a[p] ← a[k]
   a[k] ← x
end
```

2.3 Evaluarea algoritmului

2.3.1 Complexitatea timp

Cazul cel mai nefavorabil se obține atunci când la fiecare partiționare se obține una din subprobleme cu dimensiunea 1. Deoarece operația de partiționare necesită $O(q-p)$ comparații, rezultă că pentru acest caz numărul de comparații este $O(n^2)$. Acest rezultat este oarecum surprinzător, având în vedere că numele algoritmului este " sortare rapidă". Numele algoritmului se justifică prin faptul că într-o distribuție normală, cazurile pentru care Quick_Sort execută n^2 comparații sunt rare, fapt care conduce la o complexitate medie foarte bună a algoritmului. Complexitatea medie a algoritmului Quick_Sort este $O(n \log_2 n)$.

2.3.2 Consumul de memorie

La execuția algoritmilor recursivi, importantă este şi spațiul de memorie ocupat de stivă. Considerăm spațiul de memorie ocupat de stivă în cazul cel mai nefavorabil, $k = q$. în acest caz, spațiul de memorie ocupat de stivă este $M(n) = c + M(n-1)$, ce implică $M(n) = O(n)$.

în general, pivotul împarte secvența de sortat în două subsecvențe. Dacă subsecvența mică este rezolvată recursiv, iar subsecvența mare este rezolvată iterativ, atunci consumul de memorie se reduce.

```
procedure quickSort2(a, p, q)
   while (p < q) do
      partitioneaza(a, p, q, k)
      if (k-p > q-k)
         then quickSort2(a, k+1, q)
              q ← k-1
         else quickSort2(a, p, k-1)
              p ← k+1
end
```

Spațiul de memorie ocupat de stivă pentru algoritmul îmbunătățit satisface relația $M(n) \leq c + M(n/2)$, de unde rezultă $M(n) = O(\log n)$.

3 Sarcini de lucru

1. Scrieți o funcție C/C++ care implementează algoritmul mergeSort;

2. Scrieți o funcție C/C++ care implementează algoritmul quickSort1;

3. Scrieți o funcție C/C++ care implementează algoritmul quickSort2;

4. Comparați funcțiile mergeSort, quickSort1 și quickSort2. Pentru aceasta, măsurați timpii de execuție pentru n chei de sortare ($10.000 \leq n \leq 10.000.000$). Interpretați rezultatele.

Bibliografie

[1] Lucanu, D. şi Craus, M., *Proiectarea algoritmilor*, Editura Polirom, 2008.