Proiectarea algoritmilor
Lucrare de laborator nr. 7

Paradigma *Divide_et_Impera*

Sortare rapidă (Quick Sort)

Cuprins

1 Descriere

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

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

$p \leq k \leq q$ şi $a[k] = x$; $\forall i : p \leq i \leq k \Rightarrow a[i] \leq a[k]$; $\forall j : k < j \leq q \Rightarrow a[k] \leq a[j]$;

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

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

Proprietăţile menţinute invariante în timpul procesului de partiţionare sunt:

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

şi

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

Presupunem că la momentul curent sunt comparate elementele `a[i]` şi `a[j]` cu $i < j$.

Distingem următoarele cazuri:

1. $a[i] \leq x$. Transformarea $\mathtt{i} \leftarrow \mathtt{i}+1$ păstrează proprietatea (1).

2. $a[j] \geq x$. Transformarea $\mathtt{j} \leftarrow \mathtt{j}-1$ păstrează proprietatea (2).

3. $a[i] > x > a[j]$. Dacă se realizează interschimbarea a[i] $\leftrightarrow$ a[j] şi se face i $\leftarrow$ i+1 şi j $\leftarrow$ j−1, atunci sunt păstrate ambele predicate (1) şi (2).

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

Considerând $k = i - 1$ şi interschimbând a[p] cu a[k] obţinem partiţionarea dorită a tabloului.

După sortarea recursivă a subtablourilor a[p..k−1] şi a[k+1..q] se observă că tabloul este sortat deja. Astfel partea de asamblare a soluţiilor este vidă.

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)
           quickSort(a, p, k-1)
           quickSort(a, k+1, q)
end

procedure partitioneaza1(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

procedure partitioneaza2(a, p, q, k)
   x ← a[p]
   i ← p
   j ← q
   while (i < j) do
      while (a[i] ≤ x and i ≤ q) do i ← i + 1
      while (a[j] > x and j ≥ p) do j ← j - 1
      if (i < j)
         then interschimba(a[i],a[j])
   k ← j
   a[p] ← a[k]
   a[k] ← x
end
```

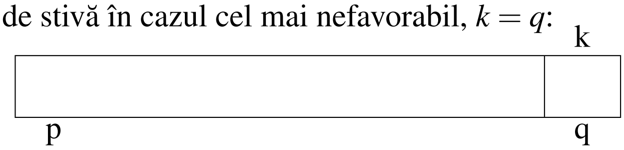
3 Evaluarea algoritmului

3.1 Complexitatea timp a algoritmului Quick Sort

Cazul cel mai nefavorabil se obţine atunci când la fiecare partiţionare se obţine una din subprobleme cu dimensiunea 1. Deoarece operaţia de partiţionare necesită $O(q-p)$ comparaţii, rezultă că pentru acest caz numărul de comparaţii 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 ītr-o distribuţie normală, cazurile pentru care quickSort execută n^2 comparaţii sunt rare, fapt care conduce la o complexitate medie foarte bună a algoritmului. Complexitatea medie a algoritmului QuickSort este $O(n\log_2 n)$.

3.2 Consumul de memorie

La execuţia algoritmilor recursivi, importantă este şi spaţiul de memorie ocupat de stivă.Considerăm spaţiul de memorie ocupat de stivă în cazul cel mai nefavorabil, $k=q$:



În acest caz, spaţiul de memorie ocupat de stivă este $M(n) = c + M(n-1)$, ce implică $M(n) = O(n)$.

În general, pivotul împarte secvenţa de sortat în două subsecvenţe. Dacă subsecvenţa mică este rezolvată recursiv, iar subsecvenţa 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 quickSort(a, k+1, q)
              q ← k-1
         else quickSort(a, p, k-1)
              p ← k+1
end
```

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

4 Sarcini de lucru şi barem de notare

Sarcini de lucru:

1. Scrieţi o funcţie C/C++ care implementează o algoritmul quickSort1;
2. Scrieţi o funcţie C/C++ care implementează o algoritmul quickSort2;
3. Comparaţi funcţiile quickSort1, quickSort1 şi qSort (din biblioteca STL). Pentru aceasta, măsuraţi timpii de execuţie pentru n chei de sortare ($10.000 \leq n \leq 10.000.000$).

Barem de notare:

1. Funcţia quickSort1: 4p
2. Funcţia quickSort2: 3p
3. Compararea funcţiilor quickSort1, quickSort1 şi qSort: 2p
4. Baza: 1p

Bibliografie

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