Proiectarea algoritmilor

Lucrare de laborator nr. 5

Sortare - Insertion_Sort, Shell_Sort și Radix_Sort

Cuprins

1 Sortare prin inserţie directă

1.1 Descriere

Algoritmul sortării prin inserţie directă consideră că în pasul k, elementele `a[0..k-1]` sunt sortate crescător, iar elementul `a[k]` va fi inserat, astfel încât, după această inserare, primele elemente `a[0..k]` să fie sortate crescător.

Inserarea elementului `a[k]` în secvenţa `a[0..k-1]` presupune:

1. memorarea elementului într-o variabilă temporară;
2. deplasarea tuturor elementelor din vectorul `a[0..k-1]` care sunt mai mari decât `a[k]`, cu o poziţie la dreapta (aceasta presupune o parcurgere de la dreapta la stânga);
3. plasarea lui `a[k]` în locul ultimului element deplasat.

1.2 Pseudocod

```
procedure insertionSort(a, n)
   for k ← 1 to n-1 do
       i ← k-1
       temp ← a[k]
       while ((i ≥ 0) and (temp < a[i])) do
          a[i+1] ← a[i]
          i ← i-1
       if (i ≠ k-1) then a[i+1] ← temp
end
```

1.3 Evaluarea algoritmului

- Căutarea poziţiei i în subsecvenţa `a[0..k-1]` necesită $O(k-1)$ timp.
- Timpul total în cazul cel mai nefavorabil este $O(1+\cdots+n-1) = O(n^2)$.
- Pentru cazul cel mai favorabil, când valoarea tabloului la intrare este deja în ordine crescătoare, timpul de execuţie este $O(n)$.

2 Sortare prin metoda lui Shell

2.1 Descriere

În algoritmul *insertionSort* elementele se deplasează numai cu câte o poziţie o dată şi prin urmare timpul mediu va fi proporţional cu n^2, deoarece fiecare element se deplasează în medie cu $n/3$ poziţii în timpul procesului de sortare. Din acest motiv s-au căutat metode care să îmbunătăţească inserţia directă, prin mecanisme cu ajutorul cărora elementele fac salturi mai lungi, în loc de paşi mici. O asemenea metodă a fost propusă în anul 1959 de Donald L. Shell, metodă pe care o vom mai numi *sortare cu micşorarea incrementului*. Exemplul următor ilustrează ideea generală care stă la baza metodei. Presupunem $n = 9$. Sunt executaţi următorii paşi (figura 1):

1. *Prima parcurgere a secvenţei*. Se împart elementele în grupe de câte trei sau două elemente (valoarea incrementului $h_1 = 4$) care se sortează separat:

$$(\mathtt{a}[0], \mathtt{a}[4], \mathtt{a}[8]), \ldots, (\mathtt{a}[3], \mathtt{a}[7])$$

2. *A doua parcurgere a secvenţei*. Se grupează elementele în două grupe de câte trei elemente (valoarea incrementului $h_2 = 3$): $(\mathtt{a}[0], \mathtt{a}[3], \mathtt{a}[6]) \ldots (\mathtt{a}[2], \mathtt{a}[5], \mathtt{a}[8])$ şi se sortează separat.

3. *A treia trecere*. Acest pas termină sortarea prin considerarea unei singure grupe care conţine toate elementele. În final, cele nouă elemente sunt sortate.

Fiecare dintre procesele intermediare de sortare implică fie o sublistă nesortată de dimensiune relativ scurtă, fie una aproape sortată, astfel că inserţia directă poate fi utilizată cu succes pentru fiecare operaţie de sortare. Prin inserţiie intermediare, elementele tind să conveargă rapid spre destinaţia lor finală. Secvenţa de incremente 4, 3, 1 nu este obligatorie; poate fi utilizată orice secvenţă $h_i > h_{i-1} > \cdots > h_0$, cu condiţia ca ultimul increment h_0 să fie 1.

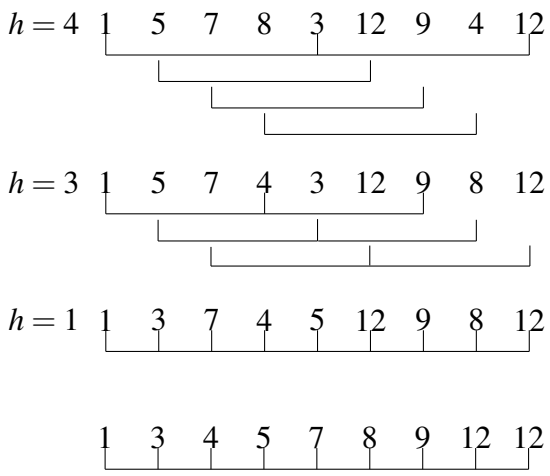

Figura 1: Sortarea prin metoda lui Shell

2.2 Pseudocod

- Presupunem că numărul de valori de incrementare este memorat de variabila `nIncr` şi că acestea sunt memorate în tabloul $(\texttt{valIncr[h]} \mid 0 \leq k \leq nIncr-1)$.

```
procedure ShellSort(a, n)
   for k ← nIncr-1 downto 0 do
       h ← valIncr[h]
       for i ← h to n-1 do
           temp ← a[i]
           j ← i-h
           while ((j ≥ 0) and (temp < a[j])) do
              a[j + h] ← a[j]
              j ← j-h
           if (j+h ≠ i) then a[j+h] ← temp
end
```

2.3 Evaluarea algoritmului - cazul cel mai favorabil şi cazul cel mai nefavorabil

Presupunem că elementele din secvenţă sunt diferite şi dispuse aleator. Denumim operaţia de sortare corespunzătoare primei faze h_t-*sortare*, apoi h_{t-1}-*sortare* etc. O subsecvenţă pentru care $a[i] \leq a[i+h]$, pentru $0 \leq i \leq n-1-h$, este denumită *h-ordonată*. Considerăm pentru început cea mai simplă generalizare a inserţiei directe, şi anume cazul când avem numai două valori de incrementare: $h_1 = 2$ şi $h_0 = 1$.

Cazul cel mai favorabil este obţinut când secvenţa de intrare este ordonată crescător şi sunt executate $2O(n) = O(n)$ comparaţii şi nici o deplasare.

Cazul cel mai nefavorabil, când secvenţa de intrare este ordonată descrescător, necesită $2O(1+2+3+(\frac{n}{2}-1))+O(1+2+3+(n-1)) = O(n^2)$ comparaţii şi $2O(1+2+3+(\frac{n}{2}-1))+O(1+2+3+(n-1)) = O(n^2)$ deplasări.

2.4 Evaluarea algoritmului - cazul mediu

În cea de-a doua parcurgere a tabloului `a`, avem o secvenţă 2-ordonată de elemente $a[0], a[1], \ldots, a[n-1]$. Este uşor de văzut că numărul de permutări $(i_0, i_1, \ldots i_{n-1})$ ale mulţimii $\{0, 1, \ldots, n-1\}$ cu pro-

prietatea $i_k \leq i_{k+2}$, pentru $0 \leq k \leq n-3$, este $C_n^{[\frac{n}{2}]}$, deoarece obţinem exact o permutare 2-ordonată pentru fiecare alegere de $[\frac{n}{2}]$ elemente care să fie puse pe poziţii impare.

Fiecare permutare 2-ordonată este egal posibilă după ce o subsecvenţă aleatoare a fost 2-ordonată.

Determinăm numărul mediu de inversări între astfel de permutări.

Fie A_n numărul total de inversări peste toate permutările 2-ordonate de $\{0,1,\ldots,n-1\}$. Relaţiile $A_1 = 0, A_2 = 1, A_3 = 2$ sunt evidente. Considerând cele şase cazuri 2-ordonate, pentru $n = 4$,

$$0\,1\,2\,3 \quad 0\,2\,1\,3 \quad 0\,1\,3\,2 \quad 1\,0\,2\,3 \quad 1\,0\,3\,2 \quad 2\,0\,3\,1$$

vom găsi $A_4 = 0+1+1+2+3 = 8$. În urma calculelor, care sunt un pic dificile, se obţine pentru A_n o formă destul de simplă:

$$A_n = \left[\frac{n}{2}\right] 2^{n-2}$$

Numărul mediu de inversări într-o permutare aleatoare 2-ordonată este:

$$\frac{\left[\frac{n}{2}\right] 2^{n-2}}{C_n^{[\frac{n}{2}]}}$$

După aproximarea lui Stirling, aceasta converge asimptotic către $\frac{\sqrt{\pi}}{128 n^{\frac{3}{2}}} \approx 0.15 n^{\frac{3}{2}}$.

Theorem 1 (1) *Numărul mediu de inversări executate de algoritmul lui Shell pentru secvenţa de incremente* $(2,1)$ *este* $O(n^{\frac{3}{2}})$.

Theorem 2 (2) *Dacă* $h \approx \left(\frac{16n}{\pi}\right)^{\frac{1}{3}}$, *atunci algoritmul lui Shell necesită timpul* $O(n^{\frac{5}{3}})$ *pentru cazul cel mai nefavorabil. (Knuth, 1976, pag. 89).*

Theorem 3 (3) *Dacă secvenţa de incremente* $h_{t-1},\ldots,h_0$ *satisface condiţia*

$$h_{s+1} \bmod h_s = 0 \text{ pentru } 0 \leq s < t-1,$$

atunci timpul de execuţie pentru cazul cel mai nefavorabil este $O(n^2)$.

O justificare intuitivă a teoremei de mai sus este următoarea:

Dacă $h_s = 2^s$, $0 \leq s \leq 3$, atunci o 8-sortare urmată de o 4-sortare, urmată de o 2-sortare nu permite nici o interacţiune între elementele de pe poziţiile pare şi impare. Fazei finale (1-sortaree) îi vor reveni $O(n^{\frac{3}{2}})$ inversări. O 7-sortare urmată de o 5-sortare, urmată de o 3-sortare amestecă astfel lucrurile încât în faza finală (1-sortarea) nu vor fi mai mult de $2n$ inversări.

Theorem 4 (4) *Timpul de execuţie în cazul cel mai nefavorabil a algoritmului* `ShellSort` *este* $O(n^{\frac{3}{2}})$, *atunci când* $h_s = 2^{s+1}-1,\ 0 \leq s \leq t-1, t = [\log_2 n]-1$.

3 Sortarea prin distribuire

Algoritmii de sortare prin distribuire presupun cunoaşterea de informaţii privind distribuţia acestor elemente.

Aceste informaţii sunt utilizate pentru a distribui elementele secvenţei de sortat în „pachete”care vor fi sortate în acelaşi mod sau prin altă metodă, după care pachetele se combină pentru a obţine lista finală sortată.

3.1 Sortarea cuvintelor

Presupunem că avem n fişe, iar fiecare fişă conţine un nume ce identifică în mod unic fişa (cheia). Se pune problema sortării manuale a fişelor. Pe baza experienţei câştigate se procedează astfel:

Se împart fişele în pachete, fiecare pachet conţinând fişele ale căror cheie începe cu aceeaşi literă. Apoi se sortează fiecare pachet în aceeaşi manieră după a doua literă, apoi etc. După sortarea tuturor pachetelor, acestea se concatenează rezultând o listă liniară sortată.

Vom încerca să formalizăm metoda de mai sus într-un algoritm de sortare a şirurilor de caractere (cuvinte). Presupunem că elementele secvenţei de sortat sunt şiruri de lungime fixată m definite peste un alfabet cu k litere. Echivalent, se poate presupune că elementele de sortat sunt numere reprezentate în baza k. Din acest motiv, sortarea cuvintelor este denumită în engleză *radix-sort* (cuvântul *radix* traducându-se prin *bază*).

Dacă urmăm ideea din exemplul cu fişele, atunci algoritmul ar putea fi descris recursiv astfel:

1. Se împart cele n cuvinte în k pachete, cuvintele din acelaşi pachet având aceeaşi literă pe poziţia i (numărând de la stânga la dreapta).

2. Apoi, fiecare pachet este sortat în aceeaşi manieră după literele de pe poziţiile $i+1,\ldots,m-1$.

3. Se concatenează cele k pachete în ordinea dată de literele de pe poziţia i. Lista obţinută este sortată după subcuvintele formate din literele de pe poziţiile $i,i+1,\ldots,m-1$.

Iniţial se consideră $i=0$. Apare următoarea problemă:

Un grup de k pachete nu va putea fi combinat într-o listă sortată decât dacă cele k pachete au fost sortate complet pentru subcuvintele corespunzătoare. Este deci necesară ţinerea unei evidenţe a pachetelor, fapt care conduce la utilizarea de memorie suplimentară şi creşterea gradului de complexitate a metodei.

O simplificare majoră apare dacă împărţirea cuvintelor în pachete se face parcurgând literele acestora de la dreapta la stânga. Procedând aşa, observăm următorul fapt surprinzător: după ce cuvintele au fost distribuite în k pachete după litera de pe poziţia i, cele k pachete pot fi combinate înainte de a le distribui după litera de pe poziţia $i-1$.

Exemplu: Presupunem că alfabetul este $\{0<1<2\}$ şi $m=3$. Cele trei faze care cuprind distribuirea elementelor listei în pachete şi apoi concatenarea acestora într-o singură listă sunt sugerate grafic în figura 2.

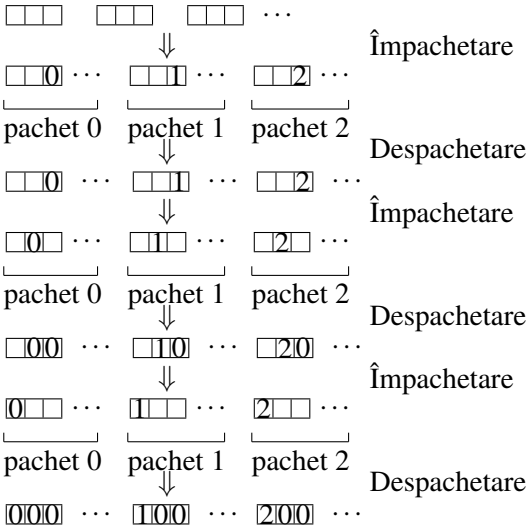

Figura 2: Sortare prin distribuire

3.2 Radix_Sort - descriere, pseudocod

Pentru gestionarea pachetelor vom utiliza un tablou de structuri de pointeri numit `pachet`, cu semnificaţia următoare:
pachet$[i]$ este structura de pointeri *pachet*$[i]$.*prim* şi *pachet*$[i]$.*ultim*,
pachet$[i]$.*prim* face referire la primul element din lista ce reprezintă pachetul i şi
pachet$[i]$.*ultim* face referire la ultimul element din lista corespunzătoare pachetului i.
Etapa de distribuire este realizată în modul următor:

1. Iniţial, se consideră listele *pachet*$[i]$ vide.
2. Apoi se parcurge secvenţial lista supusă distribuirii şi fiecare element al acesteia este distribuit în pachetul corespunzător.

Etapa de combinare a pachetelor constă în concatenarea celor k liste *pachet*$[i],\ i = 0,\ldots,k-1$.

```
procedure radixSort(L,m)
for i ← m-1 downto 0 do
    for j ← 0 to k-1 do
        pachet[j] ← listaVida()
    while (not esteVida(L)) do /* împachetare */
       w ← citeste(L, 0)
       elimina(L, 0)
       insereaza(pachet[w[i]], w)
    for j ← 0 to k-1 do /* despachetare */
        concateneaza(L, pachet[j])
end
```

3.3 Evaluarea algoritmului

Distribuirea în pachete presupune parcurgerea completă a listei de intrare, iar procesarea fiecărui element al listei necesită $O(1)$ operaţii.

- Faza de distribuire se face în timpul $O(n)$, unde n este numărul de elemente din listă.
- Combinarea pachetelor presupune o parcurgere a tabloului `pachet`, iar adăugarea unui pachet se face cu $O(1)$ operaţii, cu ajutorul tabloului `ultim`.
- Faza de combinare a pachetelor necesită $O(k)$ timp.

Algoritmul `radixSort` are un timp de execuţie de $O(m \cdot n)$.

4 Sarcini de lucru

1. Scrieţi un program C/C++ care implementează algoritmii insertionSort şi ShellSort.
2. Măsuraţi timpul de execuţie pentru n numere, unde $10.000 \leq n \leq 10.000.000$. Interpretaţi rezultatele.
3. Scrieţi un program C/C++ care implementează algoritmul radixSort. Se presupune că secvenţa de sortat este formată din n numere întregi, cu cifre din baza 10: $s = (a_0, a_1, \ldots, a_{n-1})$. Fiecare număr $a_i,\ i = 0, 1, \ldots, n-1$, din secvenţa de sortat are maxim k cifre ($a_i = c_1 c_2 \ldots c_k$).

Bibliografie

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