Metode de SORTARE

Mihai Scorțaru

Operația de ordonare a unor articole în funcție de diverse criterii este foarte des întâlnită în practică. Se cunosc o mulțime de algoritmi de sortare, majoritatea fiind foarte simpli. În cadrul acestui articol vom încerca să realizăm o prezentare comparativă a performanțelor care pot fi obținute folosind două dintre cele mai cunoscute metode de sortare.

Introducere

Metodele de sortare cele mai des folosite pot fi clasificate în două categorii: *metode directe* și *metode avansate*.

Metodele directe se bazează pe algoritmi de dificultate redusă, ușor de găsit și de înțeles. Metodele directe pe care le vom lua în considerare sunt sortarea prin selecție (*SelectSort*), sortarea prin inserție (*InsertSort*) și sortarea cu bule (*BubbleSort*).

Metodele avansate se bazează pe algoritmi puțin mai complicați, dar care nu necesită cunoștințe avansate de algoritmică. Câteva din cele mai cunoscute sunt sortarea rapidă (*QuickSort*), sortarea prin interclasare (*MergeSort*) și sortarea cu ansamble (*HeapSort*).

Vom studia doar performanțele a două metode directe de sortare și anume sortarea prin selecție și sortarea prin inserție. Este cunoscut faptul că nu există algoritmi generali de sortare care să aibă, în cel mai defavorabil caz, un ordin de complexitate mai mic decât $\Omega(n \cdot \log n)$. Vom studia ordinele de complexitate ale algoritmilor prezentați pentru cazul cel mai favorabil, cazul mediu și cazul cel mai defavorabil. După cum veți vedea, în cazul unor anumite metode este foarte dificil sau imposibil de determinat cazul cel mai defavorabil.

Măsurarea performanțelor

Vom studia performanțele algoritmilor din punct de vedere teoretic. Pentru aceasta va trebui să alegem o măsură a performanțelor acestora.

Nu vom putea compara doi algoritmi doar pe baza ordinului de complexitate deoarece, în multe cazuri, cei doi algoritmi vor avea același ordin de complexitate.

Nu putem măsura nici timpii de execuție ai unor programe, deoarece vom prezenta doar teoretic algoritmii, fără a studia efectiv și implementările.

Pentru a determina măsura performanțelor vom identifica operațiile care sunt efectuate în timpul execuției.

Trebuie remarcat faptul că, de obicei, nu se sortează șiruri de numere, ci șiruri de articole, deci o comparare între două articole implică, în general, mult mai multe operații decât o comparare între două numere întregi.

Este evident că vom efectua parcurgeri (liniare sau nu) ale șirurilor care trebuie sortate. Așadar, vom avea nevoie de anumiți indecși care vor fi folosiți pentru aceste parcurgeri. Acești indecși sunt, de obicei, numere întregi și operațiile efectuate asupra lor sunt rapide.

Pentru a sorta elementele șirului vom avea nevoie de comparații. O comparare între două elemente ale vectorului este o operație mult mai lentă.

De asemenea, vom efectua diferite atribuiri. Durata unei operații de atribuire este comparabilă cu cea a unei operații de comparare, dar nu se poate spune că cele două durate sunt egale.

Așadar, vom lua în considerare doar operațiile de comparare și atribuire care sunt efectuate asupra elementelor vectorului sau asupra unor elemente care au aceeași structură cu elementele vectorului. Cel mai bun exemplu de astfel de elemente îl reprezintă variabilele auxiliare.

Nu vom lua în considerare operațiile de comparare sau atribuire efectuate asupra indecșilor sau a altor elemente având structură similară.

De fapt, practic, atribuirile pot fi evitate în totalitate dacă operațiile nu se efectuează asupra elementelor vectorului propriu-zis, ci asupra unui vector auxiliar de indecși. Deși, în practică, acestă metodă este folosită aproape întotdeauna, vom lucra direct cu elementele șirului pentru a obține o mai bună caracterizare a performanțelor algoritmilor studiați.

Din aceste motive, trebuie reținut faptul că o măsură a performanțelor general acceptată este reprezentată de *numărul comparărilor efectuate*.

Totuși, în cele ce urmează, vom folosi două măsuri diferite ale performanței pentru a caracteriza algoritmii și

anume: ***numărul comparărilor efectuate*** și ***numărul atribuirilor efectuate***.

Apare în mod evident întrebarea: *cum vom determina numărul comparărilor și numărul atribuirilor*? Răspunsul este unul foarte simplu: *le vom număra*.

Pentru a număra comparările și atribuirile vom folosi două variabile care vor fi incrementate de fiecare dată când se execută o comparare de articole sau o atribuire de articole. În acest articol vom denumi aceste variabile `comp` și `atr`.

În cazul în care o comparare apare în cadrul condiției de continuare a unei structuri repetitive anterior condiționate, atunci se va executa o comparare pentru fiecare execuție a corpului buclei și una suplimentară în momentul în care condiția de executare a corpului buclei nu mai este satisfăcută.

În cazul în care compararea apare în cadrul condiției de continuare a unei structuri repetitive posterior condiționate, nu se execută și compararea suplimentară, fiind executate doar comparările corespunzătoare execuțiilor corpului buclei.

În cazul în care compararea apare în cadrul condiției decizionale a unei structuri alternative, numărul comparațiilor va crește indiferent de rezultatul evaluării condiției.

Vom determina numărul de atribuiri și comparații pentru șiruri formate din 10, 20, 30, ..., 100 de numere.

În continuare vom prezenta metodele directe de sortare și vom realiza o prezentare comparativă a performanțelor acestora.

Pentru simplificare, vom considera că vom sorta un șir de n numere întregi și că acestea trebuie ordonate crescător. Această alegere nu reduce generalitatea, deoarece vom lua în considerare doar comparațiile și atribuirile efectuate asupra elementelor vectorului și asupra variabilelor auxiliare. Vom ignora toate celelalte atribuiri și comparări, chiar dacă ele sunt efectuate tot asupra unor numere întregi.

Sortarea prin selecție

Ideea care stă la baza sortării prin selecție este aceea de a alege primul element al șirului sortat și a-l așeza pe prima poziție. Presupunem că acest element s-a aflat pe poziția x în șirul inițial; primul element în șirul inițial va fi mutat în poziția x. În continuare vom aplica aceeași regulă pentru restul șirului (începând cu al doilea element) și vom obține primele două elemente ale șirului sortat. Vom aplica apoi regula pentru al treilea element, al patrulea element etc. până când vom obține șirul sortat.

Prezentarea algoritmului

Se observă că vom parcurge șirul inițial pentru a amplasa pe poziția curentă elementul corespunzător din șirul ordonat. Pentru a realiza amplasarea va trebui să căutăm cel mai mic element al subșirului rămas, deci vom realiza o parcurgere a acestui subșir. După identificarea elementului care trebuie amplasat în poziția curentă, vom realiza o interschimbare.

Varianta #1

Versiunea în pseudocod a algoritmului care realizează exact aceste operații este următoarea:

```
atr ← 0
comp ← 0
pentru i ← 1, n execută
  min ← a_i                          % atribuire %
  atr ← atr + 1
  ind ← i
  pentru j ← i + 1, n execută
    dacă min > a_j atunci            % comparare %
      min ← a_j                      % atribuire %
      atr ← atr + 1
      ind ← j
    sfârșit dacă
    comp ← comp + 1
    a_ind ← a_i                      % atribuire %
    atr ← atr + 1
    a_i ← min                        % atribuire %
    atr ← atr + 1
  sfârșit pentru
sfârșit pentru
```

Pentru realizarea interschimbării nu a fost nevoie de o variabilă auxiliară, variabila `min` având acest rol.

Se observă că, pentru un șir cu n elemente, numărul de comparări este același, indiferent de valoarea celor n numere.

Numărul atribuirilor poate varia doar datorită atribuirii din interiorul structurii alternative **`dacă`**.

Cazul cel mai favorabil apare atunci când această atribuire se execută de cele mai puține ori. Se observă că, în cazul în care șirul inițial este deja sortat, această atribuire nu este executată deloc, deci cel mai favorabil caz este cel în care șirul dat este deja sortat.

Cazul cel mai defavorabil apare atunci când atribuirea se execută de cele mai multe ori. Am putea crede că numărul maxim este $(n - 1) + (n - 2) + ... + 2 + 1 = n \cdot (n - 1) / 2$. Totuși, datorită permutărilor efectuate în cadrul algoritmului, nu putem găsi nici un șir pentru care această atribuire să fie executată de $n \cdot (n - 1) / 2$ ori.

La prima execuție a buclei **`pentru`** exterioare, bucla **`pentru`** interioară se execută de $n - 1$ ori, deci numărul maxim de execuții ale atribuirii este $n - 1$. Însă, după efectuarea permutării, pe ultima poziție ajunge cel mai mare element al șirului, deci la următoarea execuție a buclei **`pentru`** exterioare, numărul maxim de execuții ale atribuirii este $n - 3$. Este evident că, la fiecare pas, numărul de execuții se reduce cu 2.

Este ușor de găsit un șir care să ducă la o astfel de execuție a algoritmului: șirul trebuie să fie ordonat descrescător.

Se observă că nu există nici o altă configurație a șirului care să determine un număr mai mare de execuții ale atribuirii, deci putem considera că șirul ordonat descrescător reprezintă cazul cel mai defavorabil pentru acest algoritm.

În tabelul 1 este prezentată o statistică a numărului de atribuiri și comparări efectuate în diferite situații.

N	Favorabil		Mediu		Defavorabil	
	comp	atr	comp	atr	comp	atr
10	45	30	45	41	45	55
20	190	60	190	94	190	160
30	435	90	435	151	435	315
40	780	120	780	211	780	520
50	1225	150	1225	273	1225	775
60	1770	180	1770	336	1770	1080
70	2415	210	2415	401	2415	1435
80	3160	240	3160	467	3160	1840
90	4005	270	4005	533	4005	2295
100	4950	300	4950	600	4950	2800

Tabelul 1: SelectSort varianta #1

Varianta #2

Putem optimiza acest algoritm dacă, la fiecare pas, nu păstrăm valoarea celui mai mic număr, ci doar poziția acestuia. Se observă că numărul minim nu este suprascris decât după ce a fost determinat cu certitudine, minimele intermediare rămânând nemodificate.

Versiunea în pseudocod a algoritmului care realizează această mică optimizare este următoarea:

```
atr ← 0
comp ← 0
pentru i ← 1, n execută
  ind ← i
  pentru j ← i + 1, n execută
    dacă a_ind > a_j atunci        % comparare %
      ind ← j
    sfârșit dacă
    comp ← comp + 1
    aux ← a_ind                    % atribuire %
    atr ← atr + 1
    a_ind ← a_i                    % atribuire %
    atr ← atr + 1
    a_i ← aux                      % atribuire %
    atr ← atr + 1
  sfârșit pentru
sfârșit pentru
```

Pentru realizarea interschimbării în această versiune avem nevoie de o variabilă auxiliară, deoarece variabila `min` nu mai este folosită.

Și în acest caz, numărul comparărilor este constant. Se observă însă, că și numărul atribuirilor este constant deoarece în interiorul structurii alternative nu se mai execută atribuiri asupra elementelor vectorului. Așadar, pentru această variantă a algoritmului numărul comparărilor și atribuirilor nu variază în funcție de configurația datelor de intrare, deci nu există cazuri favorabile sau defavorabile.

În tabelul 2 este prezentată o statistică a numărului de atribuiri și comparări efectuate în diferite situații. Pentru a nu crea confuzii vom păstra toate cele șapte coloane ale tabelului anterior, chiar dacă informațiile sunt redundante.

N	Favorabil		Mediu		Defavorabil	
	comp	atr	comp	atr	comp	atr
10	45	30	45	30	45	30
20	190	60	190	60	190	60
30	435	90	435	90	435	90
40	780	120	780	120	780	120
50	1225	150	1225	150	1225	150
60	1770	180	1770	180	1770	180
70	2415	210	2415	210	2415	210
80	3160	240	3160	240	3160	240
90	4005	270	4005	270	4005	270
100	4950	300	4950	300	4950	300

Tabelul 2: SelectSort varianta #2

Analiza variantelor

Se observă că, pentru cele două variante numărul comparărilor este același, deci, din acest punct de vedere ele sunt la fel de performante.

Numărul atribuirilor efectuate de cea de-a doua variantă a algoritmului este egal cu cel efectuat de prima variantă în cazul cel mai favorabil. Așadar, din acest punct de vedere, al doilea algoritm este cel puțin la fel de performant ca și primul. În figura 1 este prezentat graficul numărului de atribuiri pentru cazul mediu.

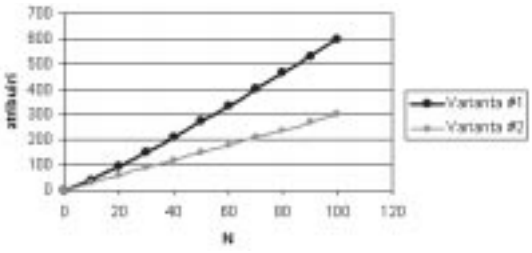

Figura 1: SelectSort: cazul mediu - atribuiri

Graficul numărului de atribuiri pentru cazul cel mai defavorabil este prezentat în figura 2.

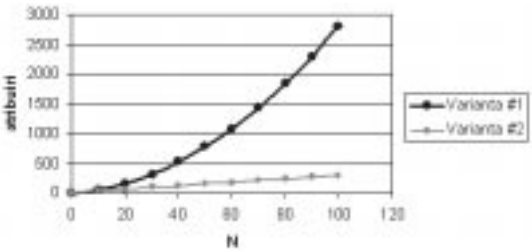

Figura 2: SelectSort: cazul cel mai defavorabil - atribuiri

Se observă clar că pentru varianta #1 graficul ia forma unei funcții pătratice, în timp ce pentru varianta #2 graficul are forma unei funcții liniare.

În figura 3 vom prezenta graficul numărului de comparări efectuate de algoritmul *SelectSort*. Numărul comparărilor nu variază în funcție de varianta folosită.

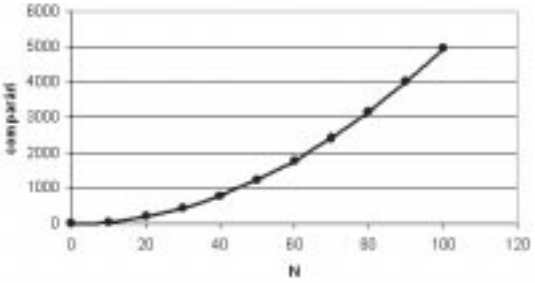

Figura 3: SelectSort: comparări

Din grafice putem trage concluzia că algoritmul *SelectSort* are un ordin de complexitate pătratic indiferent de structura datelor de intrare.

Sortarea prin inserare

Algoritmul de sortare prin inserare constă în parcurgerea șirului neordonat și inserarea elementului curent *i* în poziția corectă în subșirul de *i* - 1 elemente deja ordonat.

Mai exact, se începe cu elementul de pe poziția a doua și se verifică dacă este sau nu mai mic decât primul element. În caz afirmativ cele două elemente sunt interschimbate. După eventuala interschimbare suntem siguri că subșirul format din primele două elemente este ordonat. Se trece la al treilea element și se determină poziția în care trebuie inserat acesta în subșirul format din primele două elemente. Elementul este amplasat în poziția corectă, iar elementele care urmează după el sunt deplasate la dreapta cu o poziție. După inserare, subșirul format din primele trei elemente este sortat. Se continuă până la inserarea tuturor elementelor. În final vom obține un șir ordonat.

Există două variante importante ale algoritmului, fiecare având diferite subvariante. Diferența principală constă în modul în care este realizată căutarea: căutare *liniară* sau *binară*.

Vom prezenta câte două subvariante pentru fiecare dintre modalitățile de căutare. Datorită faptului că algoritmul de căutare liniară este mai simplu, vom trata, pentru început, subvariantele în care poziția elementului curent este determinată folosind o căutare liniară.

Varianta #1 subvarianta #1

Vom prezenta acum o variantă în care se folosește un algoritm de căutare liniară pentru a determina poziția în care este inserat elementul curent aflat inițial pe o poziție *i*. Pentru aceasta vom parcurge șirul de la stânga la dreapta până vom găsi un element mai mare decât cel curent, iar apoi vom deplasa celelalte elemente (până la poziția *i* - 1) cu o poziție la dreapta și vom insera elementul curent în subșirul ordonat pe poziția corectă. Dacă nu există nici un element mai mare decât cel de pe poziția *i*, atunci poziția acestuia nu va mai fi modificată.

Pseudocodul algoritmului descris este următorul:

```
atr ← 0
comp ← 0
pentru i ← 2, n execută
   j ← 0
   cât timp a_j < a_i și j < i execută
         % comparare în buclă anterior condiționată%
      j ← j + 1
      comp ← comp + 1
   sfârșit cât timp
   comp ← comp + 1
   aux2 ← a_j                          % atribuire %
   atr ← atr + 1
   pentru k ← j, i - 1 execută
      aux1 ← a_{k+1}                   % atribuire %
      atr ← atr + 1
      a_{k+1} ← aux2                   % atribuire %
      atr ← atr + 1
      aux2 ← aux1                      % atribuire %
      atr ← atr + 1
   sfârșit pentru
   a_j ← aux2                          % atribuire %
   atr ← atr + 1
sfârșit pentru
```

Trebuie remarcat faptul că pentru a deplasa elementele avem nevoie, la fiecare pas, de trei atribuiri.

Vom încerca să găsim acum cazul cel mai favorabil din punct de vedere al comparărilor efectuate. În cel mai favorabil caz corpul buclei **cât timp** nu ar trebui să se execute nici o dată. Pentru aceasta va trebui ca elementul curent să fie inserat întotdeauna pe prima poziție, deci șirul ar trebui să fie ordonat descrescător.

Cazul cel mai defavorabil apare atunci când corpul buclei **cât timp** se execută de cele mai multe ori, așadar elementul curent ar trebui să nu își modifice niciodată poziția, deci șirul trebuie să fie ordonat crescător.

Vom studia acum cazul cel mai favorabil și cel mai defavorabil pentru atriburi. În cel mai favorabil caz, corpul buclei interioare **pentru** nu trebuie să se execute nici o dată. Aceasta înseamnă că nu sunt necesare deplasări, deci elementele nu trebuie să își modifice poziția. Ca urmare, în cel mai favorabil caz șirul trebuie să fie ordonat crescător.

Cazul cel mai defavorabil apare în situația în care corpul buclei interioare **pentru** se execută de cele mai multe ori. Aceasta înseamnă că elementul curent ar trebui inserat întotdeauna pe prima poziție, deci șirul trebuie să fie ordonat descrescător.

Se observă că situația este oarecum bizară. Cazul cel mai favorabil pentru comparări este cazul cel mai defavorabil pentru atribuiri, iar cazul cel mai defavorabil pentru comparări este cazul cel mai favorabil pentru atribuiri.

Datorită faptului că nu poate fi determinată o echivalență absolută între un anumit număr de atribuiri și un anumit număr de comparări, nu există posibilitatea determinării cu exactitate a celui mai favorabil sau celui mai defavorabil caz. În tabelul 3 este prezentată o statistică a numărului de atribuiri și comparări efectuate în diferite

situații. Datorită situației prezentate, nu mai există coloanele corespunzătoare cazului celui mai favorabil și cazului celui mai puțin favorabil. Ele sunt înlocuite de coloane corespunzătoare unui șir ale cărui elemente sunt ordonate crescător și unui șir ale cărui elemente sunt ordonate descrescător.

N	Crescător		Mediu		Descrescător	
	comp	atr	comp	atr	comp	atr
10	54	18	31	86	9	153
20	209	38	113	326	19	608
30	464	58	244	716	29	1363
40	819	78	425	1259	39	2418
50	1274	98	655	1953	49	3773
60	1829	118	935	2799	59	5428
70	2484	138	1263	3798	69	7383
80	3239	158	1642	4947	79	9638
90	4094	178	2071	6244	89	12193
100	5049	198	2547	7702	90	15048

Tabelul 3: InsertSort varianta #1 subvarianta #1

Varianta #1 subvarianta #2

Se observă imediat o posibilitate de optimizare a algoritmului anterior. Putem păstra elementul a cărui poziție trebuie modificată într-o variabilă auxiliară și apoi vom putea deplasa elementele la dreapta cu o poziție, parcurgând subșirul deja ordonat de la dreapta la stânga. Astfel, pentru deplasări nu se vor mai efectua trei atribuiri la fiecare pas, ci doar una singură, deoarece interschimbarea este înlocuită de o singură atribuire.

De asemenea, putem determina poziția în care trebuie inserat elementul parcurgând subșirul ordonat de la dreapta la stânga. Așadar, vom parcurge elementele până vom găsi unul care este mai mic decât elementul curent sau ajungem pe prima poziție.

Pe măsură ce efectuăm căutarea, putem să efectuăm și deplasările.

În final, vom insera elementul curent în poziția determinată.

Pseudocodul acestui algoritm este prezentat în continuare:

```
atr ← 0
comp ← 0
pentru i ← 2, n execută
   j ← i
   aux ← a_i                    % atribuire %
   atr ← atr + 1
   cât timp aux < a_{j-1} și j > 1 execută
         % comparare în buclă anterior condiționată%
      comp ← comp + 1
      a_j ← a_{j-1}             % atribuire %
      atr ← atr + 1
      j ← j - 1
   sfârșit cât timp
   comp ← comp + 1
   a_j ← aux                    % atribuire %
   atr ← atr + 1
sfârșit pentru
```

Vom încerca să găsim acum cazul cel mai favorabil pentru acest algoritm. Se observă că la fiecare execuție a buclei **cât timp** se efectuează câte o atribuire și câte o comparare, iar în exteriorul acesteia nu există alte bucle care să influențeze numărul total de comparări și atribuiri. Putem trage concluzia că vom avea aceeași configurație pentru șirul care trebuie ordonat atât în cazul cel mai favorabil din punct de vedere al comparărilor, cât și în cazul cel mai favorabil din punct de vedere al atribuirilor. Situația este aceeași și pentru cazul cel mai defavorabil.

Cazul cel mai favorabil apare atunci când corpul buclei **cât timp** nu se execută nici o dată, deci poziția elementului curent nu se modifică. Așadar, șirul trebuie să fie ordonat crescător. Cazul cel mai defavorabil apare atunci când corpul buclei **cât timp** se execută de cele mai multe ori, deci elementul curent va fi inserat întotdeauna pe prima poziție. Așadar, șirul trebuie să fie ordonat descrescător.

În tabelul 4 este prezentată o statistică a numărului de atribuiri și comparări efectuate în diferite situații.

N	Favorabil		Mediu		Defavorabil	
	comp	atr	comp	atr	comp	atr
10	9	18	30	39	54	63
20	19	38	113	132	209	228
30	29	58	243	272	464	493
40	39	78	425	464	819	858
50	49	98	654	703	1274	1323
60	59	118	934	993	1829	1888
70	69	138	1264	1333	2484	2553
80	79	158	1642	1721	3239	3318
90	89	178	2070	2159	4094	4183
100	99	198	2550	2649	5049	5148

Tabelul 4: InsertSort varianta #1 subvarianta #2

Compararea subvariantelor

Se observă că, pentru cele două subvariante, numărul comparărilor este aproximativ același deci, din acest punct de vedere ele sunt la fel de performante. Singurele diferențe apar în cazul mediu, dar sunt nesemnificative și sunt datorate faptului că șirurile folosite pentru determinarea numărului de atribuiri și comparări în cazul mediu au fost generate aleator.

În cazul mediu și în cazul cel mai defavorabil numărul atribuirilor efectuate de cea de-a doua subvariantă a algoritmului este de aproximativ trei ori mai mic decât numărul atribuirilor efectuate de prima variantă. În cazul cel mai favorabil, numărul atribuirilor efectuate de cele două subvariante este identic. Așadar, din acest punct de vedere, al doilea algoritm este cel puțin la fel de performant ca și primul. În figura 4 este prezentat graficul numărului de atribuiri pentru cazul mediu.

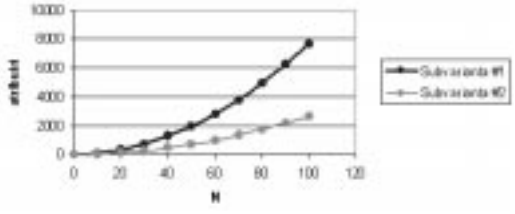

Figura 4: InsertSort #1: cazul mediu - atribuiri

Graficul numărului de atribuiri pentru cazul cel mai defavorabil este prezentat în figura 5.

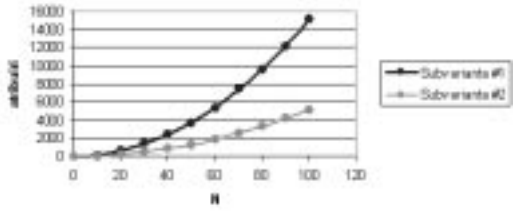

Figura 5: InsertSort #1: cazul cel mai defavorabil - atribuiri

Se observă că graficele numărului de atribuiri au forma unor funcții pătratice.

În figura 6 vom prezenta graficul numărului de comparări pentru cazul mediu și cazul cel mai defavorabil.

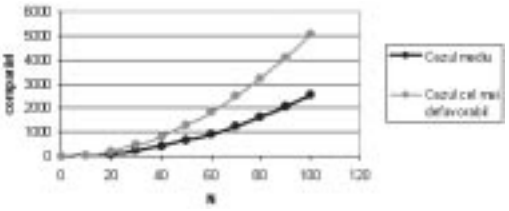

Figura 6: InsertSort #1: comparări

Din grafice putem trage concluzia că algoritmul *InsertSort* are un ordin de complexitate pătratic pentru cazul mediu și cazul cel mai defavorabil. În cel mai favorabil caz, cea de-a doua subvariantă a algoritmului rulează în timp liniar.

Varianta #2 subvarianta #1

Următoarea variantă pe care o prezentăm folosește un algoritm de căutare binară pentru a determina poziția în care este inserat elementul curent aflat inițial pe o poziție *i*. După determinarea poziției în care va fi inserat elementul curent, elementele cu valori mai mari sunt deplasate la stânga cu o poziție.

O versiune a pseudocodului acestui algoritm este:

```
atr ← 0
comp ← 0
pentru i ← 2, n execută
   stg ← 0
   dr ← i
   mijl ← [(stg + dr) / 2]
   cât timp stg < dr execută
      comp ← comp + 1
      dacă a_mijl < a_i                    % comparare %
         atunci stg ← mijl + 1
         altfel dr ← mijl - 1
      sfârșit dacă
   sfârșit cât timp
      dacă a_mijl < a_i                    % comparare %
         atunci mijl ← mijl + 1
      sfârșit dacă
      comp ← comp + 1
      aux2 ← a_mijl                        % atribuire %
      atr ← atr + 1
      pentru k ← mijl, i - 1 execută
         aux1 ← a_k+1                      % atribuire %
         atr ← atr + 1
         a_k+1 ← aux2                      % atribuire %
         atr ← atr + 1
         aux2 ← aux1                       % atribuire %
         atr ← atr + 1
      sfârșit pentru
      a_j ← aux2                           % atribuire %
      atr ← atr + 1
sfârșit pentru
```

Se observă că singura diferență față de prima subvariantă a primei variante este dată de modul în care se realizează căutarea. Aceasta se realizează în timp constant $O(\log i)$ unde i este indicele elementului a cărui poziție se caută și nu depinde de configurația șirului.

Există o mică variație nesemnificativă datorită faptului că în urma înjumătățirilor, diferența dintre numărul elementelor din cele două jumătăți ar putea fi 1, deci, în unele cazuri, s-ar putea ca numărul de pași necesari pentru găsirea pozițiilor să varieze. În urma studierii rezultatelor se va observa că, din nou, cazul cel mai favorabil din punct de vedere al comparărilor este identic cu cazul cel mai defavorabil din punct de vedere al atribuirilor și cazul cel mai defavorabil din punct de vedere al comparărilor este identic cu cazul cel mai favorabil din punct de vedere al atribuirilor. Totuși, diferențele sunt foarte mici și pot fi ignorate.

Deplasările se realizează în același mod, deci vom avea cazul cel mai defavorabil atunci când elementele sunt inserate pe prima poziție (șirul este ordonat descrescător) și cazul cel mai favorabil atunci când poziția elementului nu se modifică (șirul este ordonat crescător).

În tabelul 5 este prezentată o statistică a numărului de atribuiri și comparări efectuate pentru cazul cel mai favorabil, cazul mediu și cazul cel mai defavorabil.

N	Favorabil		Mediu		Defavorabil	
	comp	atr	comp	atr	comp	atr
10	28	18	26	86	24	153
20	73	38	68	326	64	608
30	123	58	117	716	112	1363
40	182	78	171	1259	162	2418
50	242	98	230	1953	216	3773
60	302	118	290	2799	276	5428
70	369	138	351	3798	336	7383
80	439	158	416	4947	396	9638
90	509	178	483	6244	456	12193
100	579	198	553	7702	522	15048

Tabelul 5: InsertSort varianta #2 subvarianta #1

Varianta #2 subvarianta #2

Vom efectua aceeași optimizare ca în cazul primei variante. Pentru efectuarea deplasărilor, vom parcurge subșirul ordonat de la dreapta spre stânga. Totuși, nu vom efectua simultan și comparările, deoarece căutarea nu mai este li-

niară, ci binară. Pseudocodul acestui algoritm este prezentat în continuare:

```
atr ← 0
comp ← 0
pentru i ← 2, n execută
  stg ← 0
  dr ← i
  mijl ← [(stg + dr) / 2]
  cât timp stg < dr execută
    comp ← comp + 1
    dacă a_mijl < a_i                    % comparare %
      atunci stg ← mijl + 1
      altfel dr ← mijl - 1
    sfârșit dacă
  sfârșit cât timp
    dacă a_mijl < a_i                    % comparare %
      atunci mijl ← mijl + 1
    sfârșit dacă
  comp ← comp + 1
  aux ← a_i                              % atribuire %
  atr ← atr + 1
  pentru k ← i - 1, mijl pas -1 execută
    a_k+1 ← a_k                          % atribuire %
    atr ← atr + 1
  sfârșit pentru
  a_mijl ← aux                           % atribuire %
  atr ← atr + 1
sfârșit pentru
```

Comparările sunt efectuate în același mod ca și în cazul subvariantei anterioare, deci numărul lor poate fi considerat constant.

Atribuirile sunt efectuate la fel ca în cazul celei de-a doua subvariante a primei variante, deci vom avea cazul cel mai favorabil atunci când șirul este ordonat crescător și cazul cel mai defavorabil atunci când șirul este ordonat descrescător.

Numărul atribuirilor și comparărilor pentru diferite situații apare în tabelul 6.

N	Favorabil		Mediu		Defavorabil	
	comp	atr	comp	atr	comp	atr
10	28	18	26	39	24	63
20	73	38	68	132	64	228
30	123	58	117	272	112	493
40	182	78	171	464	162	858
50	242	98	230	703	216	1323
60	302	118	290	993	276	1888
70	369	138	351	1333	336	2553
80	439	158	416	1721	396	3318
90	509	178	483	2159	456	4183
100	579	198	553	2649	522	5148

Tabelul 6: InsertSort varianta #2 subvarianta #2

Compararea subvariantelor

Din nou, pentru cele două subvariante numărul comparărilor este aproximativ același, deci, din acest punct de vedere ele sunt la fel de performante.

Numărul comparărilor nu variază, practic, în funcție de configurația șirului care trebuie ordonat, deci poate fi considerat constant.

În figura 7 este prezentat graficul comparărilor pentru varianta algoritmului de sortare prin inserție care folosește căutarea binară pentru determinarea poziției în care trebuie inserat elementul curent.

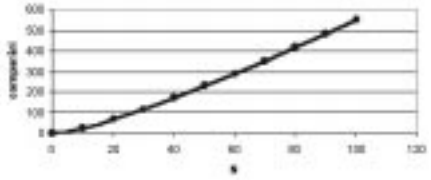

Figura 7: InsertSort #2: comparări

Fiecare căutare se realizează într-un timp logaritmic. Deoarece efectuăm n - 1 astfel de căutări, timpul total va fi unul liniar-logaritmic, deci are ordinul de complexitate $O(n \cdot \log n)$. Se poate vedea și în grafic că funcția corespunzătoare nu este chiar liniară. Caracterul liniar-logaritmic poate fi observat, în special, pentru valori mici ale numărului de elemente.

Se observă că, indiferent de modul în care se face căutarea, elementul curent va fi inserat pe aceeași poziție. Așadar, numărul atribuirilor pentru prima subvariantă este același indiferent de modul de căutare. Situația este aceeași și pentru cea de doua subvariantă. Așadar graficele care prezintă numărul atribuirilor sunt aceleași ca și în cazul primei variante.

Așadar, din punct de vedere al atribuirilor, putem trage aceleași concluzii ca și în cazul primei variante: algoritmul *InsertSort* are un ordin de complexitate pătratic pentru cazul mediu și cazul cel mai defavorabil.

Compararea variantelor

Vom compara acum cele două variante de realizare a algoritmului de sortare prin inserție.

Vom alege cea mai performantă dintre cele două subvariante a celor două variante și anume cea de-a doua.

Așa cum am arătat anterior, numărul atribuirilor nu diferă de la o variantă la alta, deoarece elementul curent va fi inserat în aceeași poziție și modul de inserare al elementelor este același.

Diferențe apar doar în cazul comparărilor datorită faptului că diferă modul de căutare al poziției în care este inserat elementul curent.

Din acest punct de vedere, prima variantă rulează în timp liniar pentru cazul cel mai favorabil și în timp pătratic pentru cazul mediu și cazul cel mai defavorabil. Cea de-a doua variantă rulează în timp liniar-logaritmic indiferent de configurația șirului.

Vom prezenta în continuare graficele numărului de comparări pentru cazul cel mai favorabil, cazul mediu și cazul cel mai defavorabil.

Pentru cazul cel mai favorabil, graficul comparărilor este cel din figura 8.

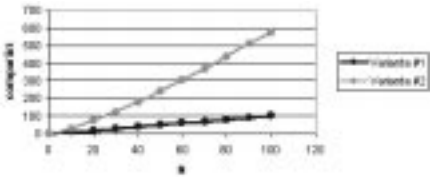

Figura 8: InsertSort: cazul cel mai favorabil - comparări

Este evident faptul că, în cazul cel mai favorabil, prima variantă este mai performantă. Aceasta se datorează faptului că o căutare liniară se realizează în timp $O(1)$ pentru cazul cel mai favorabil în timp ce căutarea binară necesită un timp $O(\log n)$.

Pentru cazul mediu, graficul comparărilor este prezentat în figura 9.

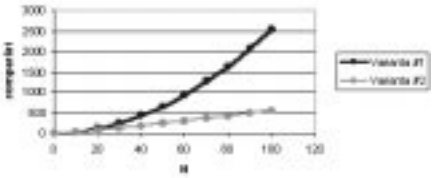

Figura 9: InsertSort: cazul mediu - comparări

În acest caz își face simțită prezența perfomanța superioară a algoritmului de căutare binară, față de algoritmul de căutare liniară. Chiar dacă pentru valori mici ale numărului de elemente ale șirului, numărul comparărilor nu diferă foarte mult de la o variantă la alta, pentru valori mari diferența este evidentă.

Pentru cazul mediu, căutarea liniară are un timp de execuție liniar (în medie, pentru determinarea poziției în care trebuie inserat al i-lea element al vectorului, se efectuează i / 2 comparări), în timp ce timpul de execuție al căutării binare este unul logaritmic. Aceasta este explicația motivului pentru care cea de-a doua variantă este mai performantă.

În figura 10 este prezentat graficul comparărilor pentru cel mai defavorabil caz.

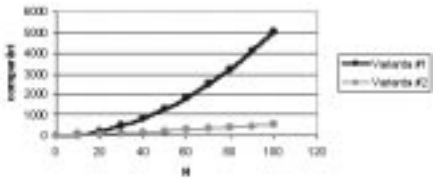

Figura 10: InsertSort: cazul cel mai defavorabil - comparări

Diferența dintre cele două metode de căutare este acum și mai evidentă. În acest caz, pentru determinarea poziției în care trebuie inserat al i-lea element al vectorului, se efectuează i - 1 comparări.

Așadar, cea de-a doua variantă a algoritmului este mai performantă în cazul cel mai defavorabil.

Putem trage concluzia că a doua variantă a algoritmului este, în majoritatea cazurilor, mai performantă decât prima variantă.

Concluzii

Probabil că v-ați dat seama că scopul acestui articol nu a fost prezentarea unor metode de sortare care, de altfel, sunt cunoscute de marea majoritate a cititorilor revistei noastre.

Am încercat să arătăm modul în care pot fi comparați diferiți algoritmi care rezolvă aceeași problemă pentru a-l alege pe cel mai performant.

De asemenea, am dorit să arătăm importanța detaliilor de implementare. În funcție de diferite alegeri pe care le facem putem îmbunătăți semnificativ performanțele.

Nu am prezentat demonstrații riguroase ale diferitelor afirmații pentru a nu încărca articolul cu elemente matematice mai greu de înțeles și, practic, neinteresante pentru majoritatea cititorilor.

Am prezentat câteva metode intuitive prin care se pot găsi ordinele de complexitate ale diferitelor variante ale unor algoritmi, fără a fi nevoie de un aparat matematic bine pus la punct.

În practică, sunt puțini cei care realizează demonstrații riguroase; majoritatea sunt mulțumiți de demonstrațiile intuitive și în foarte puține situații se dovedește că astfel de demonstrații sunt eronate.

Sfatul nostru este să încercați întotdeauna să folosiți cea mai performantă variantă a unui algoritm, iar dacă există mai mulți algoritmi cu ajutorul cărora poate fi rezolvată problema cu care vă confruntați, să îl alegeți întotdeauna pe cel care duce la rezultate corecte în timpul cel mai scurt.

Pentru aceasta nu este necesar să vă complicați folosind calcule matematice complexe, ci puteți desena doar câteva grafice și diferențele între algoritmi vor fi evidente în majoritatea cazurilor.

Chiar și simplele tabele pe care le-am folosit sunt suficiente pentru a vă da seama care dintre algoritmii sau variantele pe care le aveți la dispoziție va duce la obținerea celor mai bune performanțe.

Așa cum ați văzut, detalii minore de implementare pot îmbunătăți semnificativ performanțele. Așa cum am arătat în cazul sortării prin inserția, simpla modificare a direcției de parcurgere a șirului poate duce la reducerea de aproximativ trei ori a timpului de execuție.

Probabil că vă veți întâlni în multe situații cu necesitatea de a alege între doi sau mai mulți algoritmi. Vor exista algoritmi mai performanți în cazul mediu, dar mai puțin performanți în cazul cel mai defavorabil. Un exemplu în acest sens ar putea fi algoritmii *HeapSort* și *QuickSort*. Va trebui ca, întotdeauna, în funcție de situațiile particulare ale problemei, să puteți alege care dintre algoritmi este mai potrivit.

Mihai Scorțaru este redactor-șef adjunct al GInfo. Poate fi contactat prin e-mail la adresa skortzy@yahoo.com.