Analiza Algoritmilor. Algoritmi Elementari

CURS 5

Complexitatea Algoritmilor

1. Algoritmi de căutare

2. Algoritmi de sortare

3. Problemă rezolvată de un algoritm

Complexitatea Algoritmilor

Algoritmi de căutare

Căutare secvenţială

Ideea:

Verificăm pe rând elementele din șir până când găsim elementul căutat – poziția

Exemplu:

sir = 1, 7, 3, 5, 2

x = 3

Rezultat: 2

sir = 1, 7, 3, 5, 2

x = 4

Rezultat: 0

Complexitatea Algoritmilor

Căutare secvenţială

poz=0;

i=1;

Cât timp (i<=v_n) and (poz=0) do

dacă (v_i = x) atunci poz=i

altfel i=i+1;

returnez poz;

Complexitatea?

Complexitatea Algoritmilor

Căutare binară (doar pentru șir ordonat)

Ideea:

Se determină în ce relaţie se află elementul aflat în mijlocul șirului cu elementul ce se caută. În urma acestei verificări căutarea se continuă doar într-o jumătate a șirului. În acest mod, prin înjumătăţiri succesive se micşorează volumul șirului rămas pentru căutare.

Complexitatea Algoritmilor

Căutare binară (doar pentru șir ordonat)

{v este ordonat crescator}

{rezultatul e pozitia pe care apare x sau pe care ar trebui inserat x}

mij, inf, sup, ind

inf=1, sup=n

ind=0

execută

mij=(inf+sup)/2

dacă v_{mij} = k atunci ind=1

altfel dacă v_{mij} < k atunci inf=mij+1

altfel sup=mij-1

până când inf <= sup si ind=0

dacă ind=1 return mij

altfel return 0

Complexitatea?

Complexitatea Algoritmilor

Algoritmi de sortare

Metoda bulelor

Ideea:

Compară două câte două elemente consecutive iar în cazul în care acestea nu se află în relaţia dorită, ele vor fi interschimbate. Procesul de comparare se va încheia în momentul în care toate perechile de elemente consecutive sunt în relaţia de ordine dorită.

Complexitatea Algoritmilor

Algoritmi de sortare

Metoda bulelor

execută

ordonat=true;

pentru i=2 , v_n

dacă $v_{i-1} > v_i$ atunci

$t=v_{i-1}$

$v_{i-1} =v_i$

$v_i =t$

ordonat=false;

până când ordonat;

Complexitatea?

Complexitatea Algoritmilor

Algoritmi de sortare

Metoda inserției

Ideea:

Traversăm elementele, inserăm elementul curent pe poziția corectă în subșirul care este deja ordonat. În acest fel elementele care au fost deja procesate sunt în ordinea corectă. După ce am traversat tot șirul toate elementele vor fi sortate.

Complexitatea Algoritmilor

Algoritmi de sortare

Metoda inserției

Pentru i=2, v_n

ind=i-1;

$x=v_i$

cât timp ind>0 și $x<v_{ind}$

$v_{ind+1} = v_{ind}$

ind=ind -1

Complexitate ?

Complexitatea Algoritmilor

Algoritmi de sortare

Metoda selecției

Ideea:

Se determină poziţia elementului cu valoare minimă (respectiv maximă), după care acesta se va interschimba cu primul element.

Acest procedeu se repetă pentru subșirul rămas, până când mai rămâne doar elementul maxim.

Complexitatea Algoritmilor

Algoritmi de sortare

Metoda selecției

pentru i=1, v_{n-1}

ind=i;

for j=i+1, v_n

if $v_j < v_{ind}$ atunci ind=j;

if i<ind atunci

$t=v_i$

$v_i = v_{ind}$

$v_{ind} = t$

Complexitatea?

Complexitatea Algoritmilor

Algoritmi de sortare

Metoda numărării

Ideea:

Se da un sir de elemente distincte. Pentru fiecare element numarăm elementele care sunt mai mici, astfel aflăm poziția elementului în șirul ordonat.

Complexitatea Algoritmilor

Algoritmi de sortare

Metoda numărării

v1=v

pentr i=1 , v_n

//numar cate elemente sunt mai mici decat v_i

k=0;

x=v_i

for j=1 , v_n

dacă v_j < x atunci k=k+1;

v_{k+1} = x

Complexitatea?

Problemă rezolvată de un algoritm

Atunci când vorbim de rezolvarea unei probleme cu ajutorul unui algoritm vorbim de date de intrare – input și de date de ieșire – output.

Spunem că un algoritm A rezolvă o problemă P dacă pentru orice date de intrare execuția lui A dintr-o configurație inițială se termină într-o configurație finală ce are ca rezultat datele de ieșire.

O problemă P este **rezolvabilă** dacă există un algoritm A care rezolvă P.

O problemă P este **nerezolvabilă** dacă NU există un algoritm A care rezolvă P.

O problemă de **decizie** este o problemă P la care soluția este de tipul ”da” sau ”nu” – true sau false.

O problemă **decidabilă** este o problemă de decizie rezolvabilă.

O problemă **nedecidabilă** este o problemă de decizie nerezolvabilă.

Problemă rezolvată de un algoritm

De exemplu problema opririi (Halting Problem) este o problemă **nedecidabilă**:

"***Există un algoritm Turing-echivalent A care primind ca parametru de intrare specificația (e.g. codul sursă) al unui program arbitrar să decidă într-un număr finit de pași dacă programul a cărui specificație a primit-o ca parametru se încheie într-un număr finit de pași sau rulează la infinit?***„

Răspunsul este nu, pentru orice specificație de cod sursă (orice limbaj), adică nu putem ști despre **orice program** dacă el se va opri odată sau nu.

Teorema a fost demonstrată de Alan Turing in 1936 prin faptul că putem da unei mașini Turing H specificațiile unei mașini H1 Turing despre care să spunem dacă se oprește sau nu, însă mașina H1 fiind tot o mașină Turing putem să o descriem pe ea însăși, adică H primește de fapt la intrare propriile specificații.

Demonstrație bazată pe paradoxul mincinosului de la greci: “Eu mint”.

Problemă rezolvată de un algoritm

Există algoritmi determiniști pentru care plecând de la un set de date anume rezultatul este unic ori decâte ori este executat algoritmul (dată o mașină algoritmul trece prin aceeași succesiune de stări) (mașina Turing).

Există și algoritmi nedeterminiști care, spre deosebire de cei determiniști acceptă și următoarele instrucțiuni:

- succes și failure, când algoritmul se termină cu succes sau cu eșec
- choice(A), unde A este o mulțime finită de valori, iar funcția întoarce un element din A ales arbitrar

Există operații al căror rezultat nu este unic definit ci are valori într-o mulțime finită de posibilități.

Etape în execuția unui algoritm nedeterminist:

- choise – generează o copie pentru fiecare element din mulțimea A – partea nedeterministă a algoritmului. Are complexitatea O(1)
- testare – fiecare copie generată este testată dacă e o soluție corectă

Un algoritm nedeterminist se termină cu eșec dacă nu există o modalitate de a efectua alegeri care să conducă la o instrucțiune de succes.

Problemă rezolvată de un algoritm

Exemple de algoritmi nedeterminiști:

Exemplul 1: - se verifică dacă elementul x apare sau nu în mulțimea A

Algoritmul determinist este de ordin O(n) iar cel nedeterminist este O(1)

i -> choise() // generarea poziției elementului căutat

dacă a_i = x atunci   //testarea elementului de căutat

scrie i

scrie success

altfel failure

Problemă rezolvată de un algoritm

Exemple de algoritmi nedeterminiști:

Exemplul 2: ordonarea crescătoare a unui vector A

Complexitatea în cazul cel mai nefavorabil este $O(n^2)$ iar cel pentru cel nedeterminist este O(n)

Se copie elementele vectorului A într-un vector auxiliar B după care se verifică dacă elementele lui B sunt ordonate crescător

Pentru i -> 1 la n

j -> choice() //generarea poziția elementului vectorului B

dacă b_j = NULL atunci b_j -> a_i // testare dacă elementul b_j nu e ocupat

altfel failure

pentru i -> 1 la n-1

dacă $b_i > b_{i+1}$ atunci failure //testare dacă vectorul nu e sortat

scrie b ; succes //dacă testul nu a dat fail atunci avem succes

Problemă rezolvată de un algoritm

Exemple de algoritmi nedeterminiști:

Exemplul 3: Problema celor N regine. Avem o tablă de șah n x n și o așezare a n piese de tip regină a.î. Nicio regină nu atacă o altă regină.

Complexitatea este $O(n^2)$

Ataca (VectorSolutii, i, j)

pentru k = 0 , (i-1)

dacă $VectorSolutii_k = j$ sau $VectorSolutii_k - j = k - i$ sau $VectorSolutii_k - j = i - k$

return true

return false

Regine (n, VectorSolutii)

pentru i = 0, n

j = choise() //generare linia reginei de pe coloana i

dacă *Ataca(VectorSolutii*, i, j) failure //testare dacă se atacă

$VectorSolutii_i = j$

succes //dacă testul nu generează failure atunci avem succes