Analiza Algoritmilor – Seria CA

Test 1
09.12.2013

1. (2p) Fie 3 multimi **A**, **B** si **C** $\subseteq$ **N** despre care stim ca:
   - **A** $\leq_T$ **B** (notatie care trebuie interpretata in felul urmator: predicatul care decide multimea A $\leq_T$ predicatul care decide multimea B)
   - multimea **B** este recursiv-numarabila
   - multimea **C** este decidabila

   Ce putem spune sigur despre multimea **A \ C** (este decidabila / semidecidabila / nedecidabila) ? Justificati!

Rezolvare:

A $\leq_T$ B si B semidecidabila => A semidecidabila

Explicatie:

B semidecidabila => exista procedura PB care, pentru un input oarecare x, intoarce 1 cand x $\in$ B si nu ofera niciun raspuns atunci cand x $\notin$ B.

A $\leq_T$ B => orice input x pentru predicatul PA care decide apartenenta la multimea A poate fi transformat intr-un input f(x) pentru PB a.i. PA(x) = 1 $\Leftrightarrow$ PB(f(x)) = 1.

Cu alte cuvinte, putem folosi PB pentru a stabili ca un element oarecare x apartine lui A => exista o procedura PA care, pentru un input oarecare x, intoarce 1 cand x $\in$ A si nu ofera niciun raspuns atunci cand x $\notin$ A => A semidecidabila.

In particular, A ar putea fi chiar decidabila, insa acest lucru nu rezulta din simpla relatie A $\leq_T$ B.

A semidecidabila si C decidabila => A \ C semidecidabila

Explicatie:

C decidabila => exista procedura PC a.i. PC(x) = 1, pentru x $\in$ C si PC(x) = 0, pentru x $\notin$ C.

Pentru a decide ca x $\in$ A \ C vom rula PA(x) si, in cazul in care raspunsul este 1, vom cere si ca PC(x) = 0. Daca PA(x) nu ofera niciun raspuns inseamna ca x $\notin$ A, deci x $\notin$ A \ C. Prin urmare putem construi o procedura care, pentru un input oarecare x, intoarce 1 atunci cand x $\in$ A \ C si nu ofera niciun raspuns in caz contrar => A \ C semidecidabila.

2. (3p)

a) Fie doua functii **f**, **g: N** $\rightarrow$ $\mathbf{R_+}$. Daca stim ca **f(n) ∈ O(g(n))**, atunci gasiti limite asimptotice cat mai stricte pentru: **o(f(n)) + o(g(n))**.
b) Daca functia **f(n) ∈ O(log n)**, este adevarat ca $\mathbf{2^{f(n)} \in O(n)}$? Justificati!

Rezolvare:
a) Pornim de la definitii:

f(n) ∈ O(g(n)) => ∃ c1 > 0, n01 ∈ N ai f(n) ≤ c1 * g(n) ∀ n ≥ n01

Fie f1(n) ∈ **o(f(n))** => ∀ c2 > 0, n02 ∈ N ai f1(n) < c2 * f(n) ∀ n ≥ n02
=> f1(n) < c1 * c2 * g(n) ∀ n ≥ max(n01, n02)
Fie f2(n) ∈ **o(g(n))** => ∀ c3 > 0, n03 ∈ N ai f2(n) < c3 * g(n) ∀ n ≥ n03

=> f1(n) + f2(n) < (c1*c2 + c3) * g(n) ∀ n ≥ max(n01, n02, n03)

=> ∀ c4 = c1*c2 + c3 > 0 si ∀ n ≥ max(n01, n02, n03) , f1(n) + f2(n) < c4 * g(n)
=> **o(f(n)) + o(g(n)) = o(g(n))**

b) Fie **f(n) ∈ O(log n)** => ∃ c > 0, n0 ∈ N ai:

f(n) ≤ c * logn ∀ n ≥ n0

Cum ambele functii din relatia de mai sus sunt monotone, aplicam 2^x (2 la puterea x)

=> $2^{f(n)} \leq 2^{c * \log n}$ ∀ n ≥ n0
=> $2^{f(n)} \leq (2^{\log n})^c$ ∀ n ≥ n0
=> $2^{f(n)} \leq n^c$ ∀ n ≥ n0

In concluzie, am obtinut ca afirmatia este falsa (in cazul general). Ea este adevarata doar daca c≤1.

3. (3p) Aflati solutiile urmatoarelor recurente folosind o metoda la alegere:

a) $T(n) = T(|B|) + T(|G|) + cn$, stiind ca **c ≥ 0**, iar **|B|+|G| ≤ (1-ε)n** , unde **ε ≥ 0**
b) $T(n) = 2^k T\left(\frac{n}{2}\right) + 1$, stiind ca **k ≥ 1**

a) Daca **ε = 0** => **|B|+|G| = n** si se obtine exact recurenta de la quick-sort. Ea este tratata in Cormen si are multe solutii particulare (variind intre Ω(nlogn) si $O(n^2)$, in functie de cum sunt distribuite efectiv elementele in |B| si |G|.

In schimb, daca **ε > 0** => **|B|+|G| < n**. In situatia aceasta, se poate demonstra prin metoda substitutiei (nu facem asta aici) ca recurenta are o complexitate lineara: T(n) ∈ O(n)

b) Solutia recurentei se poate gasi prin mai multe metode, destul de simplu. Insa cea mai rapida este teorema Master:

$a = 2^k$
$b = 2 \Rightarrow e = \log_b(a) = \log_2(2^k) = k$
$f(n) = 1$

Deci $f(n) \in O(n^{e-\varepsilon}) = O(n^{k-\varepsilon})$ cu $\varepsilon \in (0,k] \Rightarrow$ TM cazul 1 $\Rightarrow T(n) \in O(n^k)$

4. (2p) Fie o tabela (vector) dinamica, **T**, care permite inserari (operatia **add**) si stergeri (operatia **remove**) de elemente. Notam cu **N(T)** = numarul de elemente din tabela, **S(T)** = dimensiunea tabelei.
   Presupunem ca in timpul unei operatii de stergere, daca dupa eliminarea elementului curent tabela ajunge sa fie plina la o capacitate de mai putin de **1/3** din dimensiunea sa (**N(T) < S(T)/3**), aceasta este redimensionata la parte intreaga inferioara din **2/3** din dimensiunea anterioara **S(T)** a tabelei. Acest lucru implica alocarea unei noi tabele de dimensiune **floor(2/3*S(T))** si copierea elementelor ramase in noua tabela.
   a) Calculati costul real al operatiei **remove**.
   b) Dandu-se functia de potential **Φ(T) = |2*N(T) - S(T)|**, sa se determine costul amortizat al operatiei **remove**.

Rezolvare:

a)

Stergere fara redimensionare: $c_{remove} = 1$ (strict operatia de stergere)

Stergere cu redimensionare: $c_{remove} = S(T)/3$ (alocarea unei noi tabele (O(1)) + copierea a S(T)/3-1 elemente in noua tabela)

b)

$\hat{c}_{remove} = c_{remove} + \Phi(T_i) - \Phi(T_{i-1})$ (costul amortizat = costul real + diferenta de potential)

Pentru $2*(N(T)-1) - S(T) \geq 0$:
$\hat{c}_{remove} = 1 + 2*(N(T)-1) - S(T) - (2*N(T) - S(T)) = 1-2 = -1$

Pentru $2*N(T) - S(T) \leq 0$, cand stergerea nu necesita redimensionare:
$\hat{c}_{remove} = 1 + S(T) - 2*(N(T)-1) - (S(T) - 2*N(T)) = 1+2 = 3$

Pentru 2*N(T) - S(T) > 0 si 2*(N(T)-1) - S(T) < 0:

$\hat{C}_{remove} = 1 + 1 - 1 = 1$

Cand stergerea necesita redimensionare:

$\hat{C}_{remove}$ = S(T)/3 (costul real)

+ 2*S(T)/3 – 2*(N(T)-1) – (S(T) – 2*N(T)) = 2

Prin urmare, costul amortizat al operatiei remove este constant in toate cele 4 situatii posibile.

5. (1.5p) Scrieti pseudocodul unui algoritm care afiseaza toate numerele prime mai mici sau egale cu **N**. Determinati complexitatea acestui algoritm.

Rezolvare:

O solutie de complexitate buna, dar care poate fi imbunatatita in continuare (nu se cerea acest lucru pentru test), poate fi gasita folosind "ciurul" lui Eratostene:

```
Print_primes(n){
    is_prime[2..N];
    for (j = 2; j <= n; j++){
        is_prime[j] = 1;
    }

    // o imbunatatire este sa aveti conditia de oprire div*div <= n
    for (div = 2; div <= n;){
         // o imbunatatire simpla ar fi ca initializarea sa fie j = div. de ce merge?
        for (j = 2; j <= n/div; j++){
            is_prime[j*div] = 0;
        }
        // aceasta este a treia imbunatatire, algoritmul poate functiona si fara ea
        // (si sa faceti un simplu div++
        do{
            div++;
        } while (is_prime[div] == 0);
    }

    for (j = 2; j <= n; j++){
        if (is_prime[j] == 1){
            cout << j << " ";
        }
    }
```

Pentru a determina complexitatea, este suficient sa gasiti de cate ori se intra in cele 2 for-uri imbricate (care contin instructiunea `is_prime[j*div] = 0`). In rest, toate celelalte operatii dureaza Θ(n).

In cele 2 for-uri imbricate se intra pentru fiecare *div* diferit, iar in cazul cel mai defavorabil ati avea:

$\sum_{div=2}^{n} \frac{n}{div} = n * \sum_{div=2}^{n} \frac{1}{div} \leq n * logn$ (ultima inegalitate ar trebui sa o stiti de la matematica: http://stackoverflow.com/questions/3748196/1-1-2-1-3-1-n)