Universitatea *Al. I. Cuza*, Iaşi
Facultatea de Informatică

`Numele:`
`Grupa:`

Proiectarea Algoritmilor - Test Scris (12 aprilie 2017), seriile B + E

Observaţii:

1. Nu este permisă consultarea bibliografiei.
2. Toate întrebările sunt obligatorii.
3. Fiecare întrebare/item este notată cu un număr de puncte indicat în paranteză. Descrieţi conceptele utilizate în răspunsuri.
4. Algorimii vor fi descrişi în limbajul Alk (cel utilizat la curs).
5. Nu este permisă utilizarea de foi suplimentare.
6. Răspunsurile deosebite pot primi bonusuri.
7. Timp de răspuns: 1 oră.

---

1. (9p) **Proiectare şi analiză, baza.**
Un *element-multiplu* este o pereche formată dintr-un element x şi numărul de apariţii ale acestuia. O *multi-mulţime* este o mulţime de elemente-multiple cu proprietatea că nu există două elemente-multiple care se referă la acelaşi element x. Un element-multiplu apare într-o multi-mulţime S dacă numărul de apariţii ale elementul curespunzător în S este mai mare sau egal cu cel din multi-elementul dat. De exemplu, a cu multiplicitatea 3 apare în S dacă şi numai dacă a apare în S cu multiplicitatea $n \geq 3$. Se consideră problema determinării apartenenţei unul element-multiplu la o multi-mulţime. Notăm această problemă cu INM.

   (a) (2p) Să se formuleze INM ca pereche (*input*,*output*). Se vor da formulări cât mai precise şi riguroase.
   Input. Un elelement-multiplu este reprezentat prin perechea (x, k), unde k reprezintă numărul de apariţii ale lui x.
   $S = \{(x_0, k_0), \ldots, (x_{n-1}, k_{n-1})\}$, $a = (y, k)$, a.î. $(\forall i, j \in \{0..n-1\}) i \neq j \implies x_i \neq x_j$.
   Output.
   true dacă $a \in S$, i.e. $(\exists i) 0 \leq i < n \wedge y == x_i \wedge n_i \geq k$,
   false altfel, i.e. $(\forall i \in \{0..n-1\}) \wedge y \neq x_i \vee n_i < k$.

   (b) (3p) Să se scrie un algoritm determinist care rezolvă INM. Presupunem perechile reprezentate prin structuri cu câmpurile `x` şi `k`, S reprezentată printr-o listă/tablou.

```
INM(S, a) {
  for (i=0; i < S.size(); ++i)
    if (S[i].n == a.n && S[i].k >= a.k) return true;
  return false;
}
```

   (c) (2p) Să justifice că algoritmul rezolvă corect problema.
   Invariantul buclei for: $(\forall j \in \{0..i-1\}) z \neq x_j \vee n_j < k$ (poate fi explicat şi în cuvinte).
   Funcţia întoarce *true* numai dacă `S[i].n == a.n && S[i].k >= a.k`, adică $a \in S$.
   Funcţia întoarce *false* numai dacă execuţia instrucţiunii `for` se termină normal, caz în care $i = S.size()$ şi care împreună cu invariantul asigură că $a \notin S$.

   (d) (2p) Să se calculeze complexitatea în cazul cel mai nefavorabil.
   Dimensiunea unei instanţe.
   $n = S.size()$
   Operaţii numărate.
   Comparaţii de elemente-multiple.
   Cazul cel mai nefavorabil.
   Când a nu apare în S (bucla `for` se execuă complet).
   Timpul pentru cazul cel mai nefavorabil.
   Bucla `for` se execută de n ori şi la fiecare iteraţie se execută exact o comparaţie.
   Timpul $= n = O(n)$.

2. (9p) **Algoritmi probabilişti, complexitate medie.**
Se consideră următorul algoritm probabilist, descris informal, care determină minimul dintr-un tablou:
minim(a, n)
1. Se iniţializează *min* cu o valoare foarte mare (notată cu ∞);
2. elementele tabloului a sunt alese pe rând aleatoriu, utilizând o distribuţie uniformă, şi pentru fiecare element $a[i]$ ales
2.1. dacă $a[i] < min$ atunci atribuie $a[i]$ lui min $(*)$.

(a) (3p) Să se descrie în Alk algoritmul. Se poate considera o funcţie `uniform(S)`, care întoarce un element ales aleatoriu uniform din S (de exemplu, S poate fi o mulţime de indici în tablou).

```
minim(a,n) {
   min = ∞;
   S = { 0 .. n };
   while (n > 0) {
     i = uniform(S);
     S = S \ singletonSet(i);
     if (a[i] < min) min = a[i];
     n = S.size();
   }
}
```

Se putea scrie şi un program care mai întâi să pemute aleatoriu elementele tabloului a.

(b) Să se calculeze probabilităţile ca algoritmul să execute atribuirea $(*)$

i. (1p) în n-a iteraţie:
Fie $i_1, i_2, \ldots, i_n$ secvenţa de indici aleşi aleatoriu.
În acest caz avem $a[i_n]$ elementul minim din a, iar celelalte elemente pot fi în orice ordine.
Echivalent, daca tabloul a este permutat aleatoriu şi apoi minimul căutat secvenţial, atunci minimuml apare pe poziţia $n-1$ în tabloul permutat.
Rezultă $p_n = \dfrac{(n-1)!}{n!} = \dfrac{1}{n}$.

ii. (1p) în a $(n-1)$-a iteraţie:
$a[i_{n-1}]$ trebuie să fie cel mai mic din $a[i_1..i_{n-1}]$.
Echivalent, daca tabloul a este permutat aleatoriu şi apoi minimul căutat secvenţial, atunci elementul de pe poziţia $n-2$ este mai mic decât cele din faţa lui în tabloul permutat.
Rezultă $p_{n-1} = \dfrac{(n-2)!}{(n-1)!} = \dfrac{1}{n-1}$.

iii. (2p) în a i-a iteraţie, $1 \le i \le n$:
Raţionând la fel ca în cazul precedent, obţinem
$p_i = \dfrac{(i-1)!}{i!} = \dfrac{1}{i}$

Justificaţi rezultatul în fiecare caz.
Hint. O descriere echivalentă a algoritmului constă în generarea unei permutări aleatorii uniforme a tabloului şi apoi explorarea secvenţială a permutării. Această idee poate fi utilizată la partea de analiză.

(c) (2p) Să se exprime ca o sumă numărul mediu de execuţii ale atribuirii $(*)$.
Fie A variabila aleatoare care întoarce numărul de atribuiri ale unei execuţii (= eveniment). Numărul mediu de execuţii ale atribuirii $(*)$ este
$M(A) = \sum_{i=1}^{n} 1 \cdot p_i = 1 + \dfrac{1}{2} + \cdots + \dfrac{1}{n-1} + \dfrac{1}{n}$.
Echivalent, dacă X_i este variabila aleatorie care ia valorea 1 dacă $(*)$ se execută la iteraţia i şi 0 altfel, aunci Deoarece $A = \sum_{i=1}^{m} X_i$, numărul mediu de execuţii ale atribuirii $(*)$ este
$M(A) = M(\sum_{i=1}^{m} X_i) = \sum_{i=1}^{m} M(X_i) = \sum_{i=1}^{m} p_i = \ldots$.

3. (9p) **Geometrie computaţională**.

(a) (3p) Să se scrie în Alk un algoritm care determină înfăşurătoarea convexă a unei linii poligonale simple. Complexitatea algoritmului trebuie să fie $O(n)$. Se presupune că primul element din listă este cel mai de jos-dreapta.

Presupunem că lista poligonală este reprezentată de lista P. Dacă linia este deschisă, atunci o transformăm într-o linie poligonală simplă închisă. Apoi înfăşurătoarea convexă se se obţine doar cu scanarea Graham:

```
CHLPS(P) {
  n = P.size();
  if (P[0] != P[n-1]) {
    P[n] = P[0];
    n = n+1;
  }
  L[0] = P[0]; L[1] = P[1]; k = 2;
  for (i = 2; i < n; ++i) {
    while (ccw(L[k - 2], L[k - 1], P[i]) < 0) {
      --k; // "sterge" ultimul element din L
    }
    L[k] = P[i];
    ++k;
  }
  return L;
}
```

(b) (2p) Exemplificaţi execuţia algoritmului pe un exemplu.

Omis.

(c) (2p) Să se justifice corectitudinea algoritmului.

Justificarea de la scanarea Graham:

Invariant `for`: Toate punctele din linia P se află în stânga liniei poligonale convexe $L[0..k-1]$

La sfârşit L devine poligon convex şi toate punctele din P se afă pe L sau în interior (care e în stânga mergând în sensul invers arcelor de ceasornic).

(d) (2p) Să se arate ca într-adevăr complexitatea este $O(n)$.

Numarul de puncte din $L[0..k-1] \cup P[i..n-1]$ scade cu o unitate la fiecare iteraţie `for` + `while`.