Universitatea *Al. I. Cuza*, Iaşi `Numele:`
Facultatea de Informatică `Grupa:`

Proiectarea Algoritmilor - Test Scris (12 aprilie 2017), seriile A + X

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**.
   Se consideră problema testării dacă o listă de elemente reprezintă o mulţime. Reamintim că într-o mulţime un element poate apărea cel mult o dată. Notăm această problemă cu ST.
   (a) (2p) Să se formuleze ST ca pereche (*input*,*output*). Se vor da formulări cât mai precise şi riguroase.
   Input. $L = (a_0, a_1, \ldots, a_{n-1})$, $n \geq 0$ (dacă $n = 0$, lista este vidă: $L = (\,)$, $a_i \in A$)

   Output. *true* dacă $(\forall i, j) i \neq j \implies a_i \neq a_j$
   false, altfel, i.e. $(\exists i, j \in \{0..n-1\}) i \neq j \wedge a_i = a_j$.
   (b) (3p) Să se scrie un algoritm determinist care rezolvă ST.

```
ST(L) {
   if (L.size() == 0) return true;
   for (i=0; i < L.size()-1; ++i)
     for (j=i+1; j < L.size(); ++j)
       if (L[i] == L[j]) return false;
   return true;
}
```

   (c) (2p) Să justifice că algoritmul rezolvă corect problema.
   Invariantul primului for: $(\forall k \in \{0..i-1\}, \ell \in \{0..n-1\}) k \neq \ell \implies L[k] \neq L[\ell]$ (poate fi explicat şi în cuvinte)
   Invariantul celui d-al doilea for: invariantul primului for în conjuncţie cu proprietăţile $(\forall \ell \in \{i+1..j-1\}) L[i] \neq L[\ell]$ şi $i \neq j$ (poate fi explicat şi în cuvinte)
   Programul se termină dacă $i \neq j$ şi $L[i] == L[j]$ (condiţia `if` adevărată), sau dacă se termină ambele instrucţiuni `for` şi, caz în care $i == n == L.size()$; aceasta împreună cu invariantul primul for asigură faptul că valoarea *true* este corect calculată.
   (d) (2p) Să se calculeze complexitatea în cazul cel mai nefavorabil.
   Dimensiunea unei instanţe. $n =$ dimensiunea listei L
   Operaţii numărate. comparaţii în care apar elemente ale listei L
   Cazul cel mai nefavorabil. L reprezintă o mulţime (instrucţiunile `for` se execută complet).
   Timpul pentru cazul cel mai nefavorabil.
   Există $\frac{n(n-1)}{2}$ perechi $i < j$, de unde rezultă că în cazul cel mai nefavorabil complexitatea timp este $\frac{n(n-1)}{2} = O(n^2)$.

2. (9p) **Algoritmi probabilişti, complexitate medie**.
Se consideră următorul algoritm probabilist, descris informal:
unulDinTrei()
1. Se alege aleatoriu uniform un element din $\{0, 1\}$;
2. Dacă elementul ales este 0, întoarce 0;
3. Altfel întoarce 1 − unulDinTrei().
Este posibil să existe şi o execuţie infinită a algoritmului.

(a) (3p) Să se descrie în Alk algoritmul (ca o funcţie recursivă).

```
unulDinTrei() {
  n = random(2);
  if (n == 0) retrun 0;
  else return 1 - unulDinTrei()
}
```

(b) Să se calculeze probabiltăţile ca algoritmul să execute

i. (1p) exact un apel recursiv:
Deoarece `random` alege aleatoriu cu distribuţia uniformă, rezultă că ambele valori $n = 0$ şi $n = 1$ sunt alese cu probabilitatea $\frac{1}{2}$. Deci primul apel recursiv se face cu probabilitatea $p_1 = \frac{1}{2}$. Acest apel se termină dacă `random` întoarce acum 0, care este ales cu probabilitatea $\frac{1}{2}$. Deoarece trebuie să aibă ambele evenimente, probabilitatea este $\frac{1}{2} \cdot \frac{1}{2} = \frac{1}{2^2}$.

ii. (1p) două apeluri recursive:
Al doilea apel recursiv apare numai dacă apare şi primul (probabilitate condiţionată): $p_2 = \frac{1}{2^2} \cdot \frac{1}{2} = \frac{1}{2^3}$

iii. (2p) n apeluri recursive, $n \geq 1$:
$p_n = \frac{1}{2^{n+1}}$ (se verifică prin inducţie, unde pasul inductiv se arată asemănător cum cazul $n = 2$ se obţine din cazul $n = 1$).

Justificaţi rezultatul în fiecare caz.

(c) (2p) Să se exprime ca o sumă infinită numărul mediu de apeluri recursive.
Numărul mediu de apeluri recursive $= \sum_{n \geq 1} n \cdot p_n = 1 \cdot \frac{1}{2^2} + 2 \cdot \frac{1}{2^3} + \cdots + n \cdot \frac{1}{2^{n+1}} + \cdots$

3. (9p) **Geometrie computaţională**.

(a) (2p) Să se scrie în Alk o funcţie `cmp`(P, Q, R), care rezolvă următoarea problemă:
Input. Trei puncte din plan (P, Q, R) astfel încât Q şi R se află deasupra lui P.
Output. -1 dacă Q este mai mic în coordonate polare decât R; 0 dacă Q şi R sunt egale în coordonate polare; $+1$ dacă Q este mai mare în coordonate polare decât R.
Coordonatele polare sunt calculate relativ la sistemul cu originea în P. Complexitatea algoritmului trebuie să fie $O(1)$ (se poate utiliza primitiva `ccw`).

```
cmp(P, Q, R) {
  if (ccw(P, Q, R) > 0) return -1;
  if (ccw(P, Q, R) < 0) return 1;
  d1 = dist2(P, Q);
  d2 = dist2(P, R)
  if (d1 == d2) return 0;
  if (d1 < d2) return -1;
  else return 1;
}
```

unde `dist2(P, Q)` calculează pătratul distantei de la P la Q:

```
dist2(P, Q) {
  return (Q.x - P.x)*(Q.x - P.x) + (Q.y - P.y)*(Q.y - P.y);
}
```

(b) (2p) Exemplificaţi execuţia funcţiei `lt`(P, Q, R) pe două exemple.
1. $P = \{x \to 2, y \to 2\}$, $Q = \{x \to 4, y \to 3\}$, $R = \{x \to 3, y \to 4\}$
Avem `ccw`$(P, Q, R) = 1$, deci `cmp`(P, Q, R) întoarce -1.
2. $P = \{x \to 2, y \to 2\}$, $Q = \{x \to 4, y \to 6\}$, $R = \{x \to 3, y \to 4\}$
Avem `ccw`$(P, Q, R) = 0$, `dist2`$(P, Q) = 20 > 5 =$ `dist2`(P, R), deci `cmp`(P, Q, R) întoarce 1.

(c) (2p) Să se scrie în Alk un algoritm `lowermost`(S), care determină cel mai de jos punct din mulţimea S.

```
lowermost(S) {
  if (S.size() = 0) return "error";
  min = S[0];
  for (i=1; i < S.size(); ++i)
    if (min.y < S[i].y) min = S[i];
  return min;
}
```

(d) (2p) Să se scrie în Alk un algoritm `sort`(S), care sortează după coordonatele polare punctele din S, utilizând `cmp()` şi `lowermost()`.

```
sort (out S) {
  P = lowermost(S);
  for (i=0; i < S.size(); ++i)
    for (j=i+1; j < S.size(); ++j)
      if (cmp(P, S[i], S[j]) == 1)
        swap(S, i, j);
}
```

Observaţie. Prezentarea corectă a unui algoritm $O(n \log n)$ aduce un bonus de 1p.

(e) (1p) Să se precizeze complexitatea algoritmului de sortare.
Dimensiunea unei instanţe.
$n = S.size()$.
Operaţii numărate.
Comparaţii în care apar elemente din S.
Cazul cel mai nefavorabil.
Nu există.
Timpul pentru cazul cel mai nefavorabil. Complexitatea timp al apelului `lowermost(S)` este $O(n)$.
Cele două instrucţiuni for realizează $\frac{n(n-1)}{2} = O(n^2)$ comparaţii.