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

`Numele:`
`Grupa:`

Proiectarea Algoritmilor - Test Scris (08 iunie 2018), seriile A + X

---

1. (12p) **Greedy**. Dându-se o mulțime I de intervale închise, cu capetele numere naturale, să se găsească o cea mai mică mulțime de numere P astfel încât fiecare interval din I să includă cel puțin unul dintre punctele din P.

   (a) (3p) Să se formuleze problema de mai sus ca pereche (*input*,*output*). Se vor da formulări cât mai precise şi riguroase.

   Input: $n \in \mathbb{N}$ – numărul de intervale; $s[0..n-1]$ – vector de numere naturale reprezentând pozițiile de început ale fiecărui interval; $f[0..n-1]$ – vector de numere naturale reprezentând pozițiile de sfârșit ale fiecărui interval cu proprietatea $s[i] \leq f[i]$ pentru orice $0 \leq i \leq n-1$.

   Output: $m \in \mathbb{N}$ - numărul de puncte și $p[0..m-1]$ - vector de numere naturale astfel încât $\forall i \in \{0, \ldots, n-1\}.\exists j \in \{0, \ldots, m-1\}.p_j \in [s_i, f_i]$ și m să fie minim.

   (b) (3p) Să se arate că strategia greedy care adaugă câte un punct în locul în care se intersectează un număr maxim de intervale neacoperite încă nu conduce tot timpul la soluția optimă.

   Exemplu: pentru intervalele $[0, 10]$, $[9, 15]$, $[8, 16]$, $[11, 19]$, $[12, 18]$, $[17, 30]$ ar trebui să adaugăm câte un punct în poziția 14, apoi în poziția 5, apoi în poziția 25 (de exemplu). Dar soluția optimă ar avea 2 puncte: 10 și 17.

   (c) (4p) Să se descrie o strategie greedy care conduce la soluția optimă. Argumentați că strategia propusă produce soluția optimă.

   La fiecare pas, se va alege cel mai mic punct dintre extremitățile-dreapta ale intervalelor care nu sunt deja acoperite cu puncte.

   Argumentare: orice soluție optimă poate fi transformată în soluția produsă de strategia de mai sus astfel: se mută repetat primul punct din soluția optimă care nu coincide cu soluția de mai sus înspre dreapta până atinge extremitatea dreapta a primul interval – rezultatul rămânând o soluție.

   (d) (2p) Să se scrie în Alk un algoritm pentru problema de mai sus care implementează strategia greedy propusă la punctul anterior.

```
greedy(n, s, f) // presupun f in ordine crescatoare
{
  acoperit = [];
  for (i = 0; i < n; ++i) {
    acoperit[i] = 0; // niciun interval nu e acoperit inca
  }
  p = []; // vector de puncte rezultat
  m = 0; // nr de puncte, initial 0
  for (i = 0; i < n; ++i) {
    if (!acoperit[i]) { // aleg primul interval neacoperit
      p[m++] = f[i]; // adaug extremitatea dreapta
      for (j = i; j < n; ++j) {
        if (s[j] <= f[i] && f[i] <= f[j]) {
          acoperit[j] = 1; // marchez intervalele nou acoperite
        }
      }
    }
  }
  return p;
}
```

Ciornă.

2. (12p) **Programare Dinamică**.
Context: Proiectarea unui algoritm, bazat pe paradigma programării dinamice, care găsește lungimea celei mai lungi 1-subsecvențe (subsecvență ale cărei elemente de pe poziții consecutive diferă prin cel mult 1). Exemplu: pentru șirul $0, 2, 1, 3, 0, 2$, astfel de subsecvențe de lungime maximă sunt $0, 1, 0$ sau $2, 3, 2$ (și altele).
Cerințe:
   (a) (3p) Să se formuleze problema de mai sus ca pereche (*input*,*output*). Se vor da formulări cât mai precise şi riguroase.
   Input: n – numărul de elemente ale șirului și $S[0..n-1]$ – șirul
   Output: cel mai mare număr l a.î. $\exists i_0, \ldots, i_{l-1}$ cu proprietatea că $i_0 < i_1 < \ldots < i_{l-1}$ și $\forall j \in \{0, \ldots, l-2\}, |S[i_j] - S[i_{j+1}]| \leq 1$.

   (b) (3p) Fie $0 \leq i < n$ o poziție în șir, unde n e lungimea șirului; notăm cu $d(i)$ lungimea celei mai lungi subsecvențe care satisface cerințele și care începe pe poziția i. Să se precizeze $d(n-1)$.

$$d(n-1) = 1.$$

   (c) (4p) Fie $0 \leq i < n-1$ o poziție în șir; să se determine o relație de recurență pentru $d(i)$, în funcție de $d(i+1), d(i+2), \ldots, d(n-1)$.

$$d(i) = 1 + max(\{0\} \cup \{d(j) \mid i < j < n \wedge |S[i] - S[j]| \leq 1\})$$

   (d) (2p) Să se enunțe proprietatea de substructură optimă specifică problemei.
   Fie $i_0, \ldots, i_l$ o soluție optimă. Atunci $i_1, \ldots, i_l$ este o soluție optimă a subproblemei care începe pe poziția i_1.

Ciornă.

3. (12p) **Probleme NP-complete**.
Problema `CLIQUE`.
Input: Un graf neorientat $G = (V, E)$, un număr natural k.
Output: "Da", dacă există $V' \subseteq V$ astfel încât $|V'| \geq k$ și există muchie între oricare două noduri din V'. "Nu", altfel.

(a) (3p) Să se definească clasa `NP`.
Clasa `NP` este clasa tuturor problemelor de decizie care pot fi rezolvate de un algoritm nedeterminist în timp polinomial în cazul cel mai nefavorabil.

(b) (3p) Să se arate că `CLIQUE` $\in$ `NP`.
Este suficient să găsim un algoritm nedeterminist polinomial pentru `CLIQUE`:

```
clique(V, E, k)
{
  count = 0;
  for each v in V {
    choose x[v] in { 0 , 1 };
    count = count + x[v];
  }
  if (count < k) {
    failure;
  }
  for each u in V such that x[u] == 1 {
    for each v in V such that x[v] == 1 {
      if muchia {u,v} nu apartine E {
        failure;
      }
    }
  }
  success;
}
```

(c) (3p) Care dintre următoarele reduceri este suficientă pentru a arăta că problema `CLIQUE` este `NP`-dificilă? De ce?
  i. `CLIQUE` la o problemă despre care știm deja că este NP-completă, în timp polinomial;
  ii. o problemă despre care știm deja că e NP-completă la `CLIQUE`, în timp polinomial.

Trebuie să găsim o reducere polinomială de la o problemă despre care știm deja că e `NP`-dificilă (e.g., `INDEPENDENT-SET`) la `CLIQUE`. `INDEPENDENT-SET` fiind `NP`-dificilă, știm că orice problemă din `NP` se reduce polinomial la `INDEPENDENT-SET`. Reducerea polinomială fiind tranzitivă, rezultă că orice problemă din `NP` se reduce polinomial la `CLIQUE`.

(d) (3p) Să se arate că problema `CLIQUE` este `NP`-completă (în rezolvare, puteți folosi fără demonstrație `NP`-completitudinea oricărei probleme `NP`-complete canonice, alta decât CLIQUE, cum ar fi SAT, 3-SAT, INDEPENDENT SET, SUBSET SUM, VERTEX COVER, sau altele).
Știm de la punctul (2) că `CLIQUE` $\in$ `NP`, deci este suficient să arătăm că este `NP`-dificilă. Arătăm că `INDEPENDENT-SET` se reduce polinomial la `CLIQUE`. Adică găsim un algoritm `AlgIS` polinomial pentru `INDEPENDENT-SET`, presupunând că avem un algoritm `AlgClique` pentru `CLIQUE`:

```
AlgIS(V,E,k)
{
  E' = emptyset; // calculam in E' complementul lui E
  for each u in V {
    for each v in V {
      if ((u != v) && ((not ({u, v} in E))) {
        E'.insert({u, v});
      }
    }
  }
  return AlgClique(V,E',k);
}
```

Ciornă.

Ciornă.