Algoritmi Avansați
Seminar 1

Gabriel Majeri

1 Introducere

Ce este o problemă (computațională)?

O **problemă computațională** [1] este o problemă la care putem determina răspunsul folosind un algoritm[1].

- **Exemplu de problemă computațională:** Care sunt factorii primi ai numărului natural n?
- **Exemplu de problemă care nu se poate rezolva folosind algoritmi:** Ce pereche de pantaloni să port astăzi?[2]

Ce tipuri de probleme computaționale există?

- **Probleme de decizie** [4]**:** Răspunsul este de tipul adevărat/fals (posibil/imposibil, există/nu există etc.)
- **Probleme de optimizare** [5]**:** Răspunsul este o *soluție* (un număr, un șir de numere, un text etc.) care este "cea mai bună" dintr-un anumit punct de vedere (consumă cele mai puține resurse, produce cel mai mare profit, acoperă cele mai multe noduri etc.) față de toate celelalte soluții posibile.

[1] Un *algoritm* [2] este o serie de pași bine-definiți care îți permit să obții un rezultat plecând de la niște date de intrare. În comparație, "să ghicești răspunsul corect" nu e tocmai un algoritm. Dar poate fi o euristică [3].

[2] Dacă ai un număr finit de opțiuni (pantaloni din care să alegi) și asociezi fiecărei perechi câte un *scor* (număr) care să indice dezirabilitatea acesteia, atunci această problemă se poate reduce la o problemă computațională, aceea de a găsi perechea de pantaloni cu scorul maxim.

2 Exerciții

Cunoștințe generale

1. Dați exemplu de **2 probleme** pe care le întâlniți în viața de zi cu zi care se pot interpreta ca probleme computaționale (e.g. cum alegi traseul pe care să vii la facultate).

Soluție: Există o mulțime de probleme care se pot rezolva algoritmic:

- Când te conectezi pe un website, la autentificare ți se poate cere adresa de e-mail și o parolă. Din motive de securitate, nu e bine ca o aplicație să păstreze parolele necriptate în baza de date [6]. Cel mai sigur este ca parola să fie trecută printr-un algoritm ca `bcrypt` [7], care îți permite să determini ulterior dacă utilizatorul a introdus parola corectă, dar fără să poți recupera parola originală.
- Vrei să faci niște cumpărături dar trebuie să treci pe la mai multe magazine, minimizând timpul pe care îl pierzi în deplasare. Alegerea optimă a ordinii în care să treci prin magazine se aseamănă cu problema comisului-voiajor [8].

□

2. Dați exemplu de **2 probleme de decizie** și **2 probleme de optimizare** pe care le-ați întâlnit în practică sau de care ați auzit.

Soluție: Probleme de decizie:

- Problema conectivității unui graf. De exemplu, am achiziționat și configurat câteva routere și vrem să știm dacă acum toate nodurile din rețeaua noastră pot comunica între ele.
- Determinarea dacă două grafuri sunt izomorfe. Chimiștii construiesc baze de date în care se află diferiți compuși chimici, iar ulterior vor să găsească și să compare substanțele, ignorând rotațiile sau simetriile structurii lor [9].

Probleme de optimizare:

- TEX, sistemul de tehnoredactare folosit la realizarea acestui material, își propune să aranjeze cuvintele pe linii și în paragrafe în așa fel încât să maximizeze aspectul estetic al paginii, utilizând cât mai puțin spațiu. Mai multe detalii se pot găsi în lucrarea lui Knuth și Plass din 1981 [10].

- Algoritmii de învățare automată, folosiți în inteligența artificială, sunt de fapt algoritmi care încearcă să minimizeze o funcție de cost.

Se pot găsi alte exemple pe Wikipedia [4, 5]. □

P versus NP

3. În cele ce urmează, să presupunem că avem **o listă** (un vector) **de** n **numere întregi**.

1. Propuneți un algoritm care să determine **cel mai mare element** (în **valoare absolută**) din listă. Ce complexitate de timp/memorie are algoritmul propus?

   Soluție: O soluție ar putea fi să citim toată lista în memorie și apoi să o parcurgem, reținând mereu valoarea maximă în modul:

   ```
   numbers = [int(x) for x in input().split()]

   max_n = numbers[0]
   for n in numbers[1:]:
       if abs(n) > abs(max_n):
           max_n = n

   print(max_n)
   ```

   Această metodă are complexitatea de timp $\mathcal{O}(n)$ (deoarece parcurgem toate numerele o singură dată), iar complexitatea de spațiu este $\mathcal{O}(n)$ (stocăm toate numerele într-o listă).

   O altă soluție, mai ineficientă ca timp, ar fi să ne folosim de funcția de sortare din Python, care are o complexitate de $\mathcal{O}(n \log n)$, și să comparăm capetele vectorului ordonat:

   ```
   numbers = [int(x) for x in input().split()]

   numbers.sort()

   if abs(numbers[0]) > abs(numbers[-1]):
       print(numbers[0])
   else:
       print(numbers[-1])
   ```

Dacă avem o cale de a citi șirul de numere element cu element (de exemplu, este stocat într-un fișier sau vine de pe rețea) putem reduce consumul de memorie:

```
max_n = 0

for line in open('numere.txt', 'r'):
    n = int(line)
    if abs(n) > abs(max_n):
        max_n = n

print(max_n)
```

În acest caz, memoria consumată este $\mathcal{O}(1)$. Totuși, fișierul consumă $\mathcal{O}(n)$ spațiu pe disc. □

2. Pentru un număr întreg dat S, propuneți un algoritm care să determine **toate perechile de numere** din lista inițială **care adunate dau** S. Puteți găsi un algoritm cu complexitate de timp, în cel mai rău caz, mai bună decât $\mathcal{O}(n^2)$?

Soluție: O soluție naivă este să parcurgem lista de numere cu două `for`-uri imbricate:

```
S = int(input())
numbers = [int(x) for x in input().split()]

for i in range(len(numbers) - 1):
    for j in range(i + 1, len(numbers)):
        if numbers[i] + numbers[j] == S:
            print(i, j)
```

Bucla interioară parcurge un subșir de $n-1$ numere, apoi de $n-2$ numere, ..., 2, 1. Deci numărul total de pași vine

$$1+2+3+\cdots+(n-2)+(n-1)=\frac{(n-1)(n-2)}{2}\in\mathcal{O}(n^2).$$

Memoria folosită se încadrează în $\mathcal{O}(n)$.

O soluție mai eficientă ar fi să parcurgem o dată vectorul și să salvăm într-un dicționar valoarea care, adunată cu elementul de pe poziția i, ne-ar da S:

```
S = int(input())
numbers = [int(x) for x in input().split()]

diff = { (S - n) : i for (i, n) in enumerate(numbers) }

for j, m in enumerate(numbers):
    if m in diff and j != diff[m]:
        print(i, j)
```

O problemă cu această soluție este că, dacă avem mai multe numere cu aceeași valoare în lista inițială, nu mai afișăm toate combinațiile posibile care generează suma S. Dar dacă ignorăm acest caz, soluția are complexitate de timp $\mathcal{O}(n)$. □

3. (3SUM [11]) Propuneți un algoritm care să determine dacă în lista inițială există **trei numere** care **adunate să aibă suma 0**. Credeți că puteți un algoritm determinist care să rezolve problema în mai puțin de n^2 pași? Dar dacă ați aborda problema în mod nedeterminist?

Soluție: Soluția deterministă care rezolvă problema în $\mathcal{O}(n^2)$ pași este:

```
numbers = [int(x) for x in input().split()]

diff = { (0 - n) : i for (i, n) in enumerate(numbers) }

for j in range(len(numbers) - 1):
    m = numbers[j]
    for k in range(j + 1, len(numbers)):
        p = numbers[k]
        rem = 0 - m - p
        if rem in diff and j != diff[rem] and k != diff[rem]:
            print(i, j, k)
            exit(0)

print('Nu există')
```

Dacă există sau nu vreun algoritm mai eficient de n^2 pentru a rezolva această problemă este o întrebare încă deschisă în informatică [11].

Pentru rezolvarea nedeterministă, să observăm că, dacă ni se spune că elementele de pe pozițiile i, j și k sunt o soluție, putem verifica asta

prin

$$numbers[i] + numbers[j] + numbers[k] == 0$$

care este timp $\mathcal{O}(1)$ în raport cu lungimea vectorului. Un algoritm nedeterminist ar ști cum să „aleagă" din prima indicii potriviți, iar verificarea s-ar face în timp polinomial (chiar constant), deci este o problemă NP (*nondeterministic polynomial time*). □

Metoda greedy

Mai multe informații despre principiile care stau la baza metodei greedy se pot găsi la [12].

4. Suntem administratorii unui spital și vrem să ne asigurăm că avem tot timpul cel puțin un medic prezent la camera de gardă într-un anumit interval de timp. Presupunem că avem la dispoziție n medici, care ar fi dispuși să stea de gardă fiecare între anumite ore. Vrem să asignăm câți mai puțini medici, ca ceilalți să se poată ocupa de alte urgențe.

Problema formalizată: Find dat un interval $[a, b]$ și o mulțime de intervale $[a_1, b_1], [a_2, b_2], ..., [a_n, b_n]$, vrem să alegem un număr *minim* dintre acestea astfel încât, reunite, să includă intervalul inițial $[a, b]$.

1. Aceasta este o problemă de **decizie** sau de **optimizare**? Dar dacă ne-ar interesa doar să vedem dacă putem acoperi intervalul dat cu intervalele disponibile?

   Soluție: Așa cum a fost dată, problema este una de optimizare; ne interesează să găsim o soluție care să acopere intervalul dat alegând *cât mai puține* intervale.

   Dacă ne interesează doar dacă se poate acoperi intervalul țintă cu intervale, atunci problema este una de decizie. □

2. Propuneți un algoritm care **să determine dacă** măcar putem acoperi intervalul dat cu intervalele disponibile.

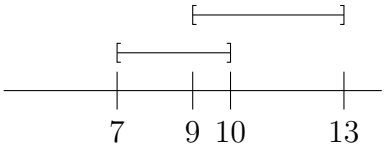
   Soluție: Începem prin a defini un mod de reprezentare al datelor de intrare ale problemei. Fiecare interval poate fi interpretat ca o pereche de numere reale $(start, end)$. Intervalul inițial îl vom nota cu (a, b), iar celelalte intervale vor fi stocate într-o listă de perechi:

```python
# Citim intervalul țintă
a, b = [float(x) for x in input().split()]

# „n" este numărul de intervale pe care le vom citi
n = int(input())

intervals = []
for i in range(n):
    start, end = [float(x) for x in input().split()]
    intervals.append((start, end))
```

Acum, să ne gândim ce fel de operații putem face cu niște intervale. Intervalele sunt de fapt **mulțimi** de numere reale, deci putem aplica operațiile uzuale de reuniune, intersecție, diferență, verificăm incluziuni etc. Nu vom avea nevoie de toate acestea ca să rezolvăm problema, dar să ne uităm peste cum le-am putea implementa:

∩ Pentru a calcula intersecția intervalelor $[a, b]$ și $[c, d]$, avem de luat în considerare două cazuri:

- Intervalele sunt disjuncte (adică $b < c$ sau $d < a$), deci intersecția este mulțimea vidă $\emptyset$.
- Intervalele se suprapun (măcar parțial) iar intersecția este nevidă. Dacă nu suntem în cazul de mai sus, intersecția intervalelor este un nou interval cu capetele $((\max(a, c), \min(b, d))$.



∪ Și pentru a calcula reuniunea intervalelor $[a, b]$ și $[b, c]$ avem de luat în calcul tot două cazuri:

- Dacă intervalele sunt disjuncte (putem verifica asta ca mai sus), atunci reuniunea lor nu mai este un interval; nu ne prea ajută acest caz, pentru că nu vrem să complicăm modul în care reprezentăm intervalele.
- Dacă intervalele nu sunt disjuncte, atunci reuniunea lor este un nou interval cu capetele $(\min(a, c), \max(b, d))$.

⊆ Pentru a verifica incluziunea intervalului (a, b) în intervalul c, d, putem pur și simplu să vedem dacă $c \leq a$ și $b \leq d$.

Aceste operații ne duc cu gândul la un mod de a rezolva problema. Dacă există o submulțime a intervalelor care, reunite, acoperă intervalul inițial, atunci sigur luând toate intervalele putem acoperi întreg intervalul țintă. Deci, o idee ar fi să construim $U = [a_1, b_1] \cup [a_2, b_2] \cup \cdots \cup [a_n, b_n]$ și să vedem dacă $[a, b] \subseteq U$.

Mai mult de atât, nici nu e nevoie să luăm *toate* intervalele la rând. Putem să ne uităm doar la cele care se intersectează cu $[a, b]$, sau cu alte cuvinte, să calculăm intersecțiile $[a, b] \cap [a_1, b_1], [a, b] \cap [a_2, b_2], \ldots$ și să le reunim pe acestea.

Ca să ne fie mai ușor să calculăm reuniunile, am vrea să nu tratăm și cazul în care unele intervale sunt disjuncte. De fapt, dacă rămânem cu o „groapă" între intervalele noastre, înseamnă că nu putem acoperi întreg intervalul țintă. În acest scop, vom **sorta** intervalele crescător după capătul din stânga.

```
# Sortarea va compara perechile de numere folosind
# ordinea lexicografică, deci va ordona mai întâi
# după capătul din stânga
intervals.sort()

i = 0

# Ignorăm intervalele care nu ne ajută pentru că
# oricum sunt la stânga intervalului țintă
while i < len(intervals) and intervals[i][1] < start:
    i += 1

# Toate intervalele sunt la stânga celui țintă
if i == len(intervals):
    print('NU')
    exit()

left, right = intervals[i]
for interval in intervals[i + 1:]:
    # Dacă nu putem reuni următorul interval,
    # înseamnă că există o „groapă" pe care
    # nu o putem acoperi
    if interval[0] > right:
        break
```

```
        # Încerc să reunesc noul interval
        right = max(right, interval[1])

        # Dacă deja am acoperit intervalul [a, b]...
        if right >= b:
            # ...nu mai continuăm parcurgerea
            break

# Verificăm dacă am reușit
# să cuprindem intervalul țintă
if left <= a and b <= right:
    print('DA')
else:
    print('NU')
```

□

3. Propuneți un algoritm care să rezolve problema inițială. **Demonstrați** că algoritmul propus obține soluția optimă (adică că nu poate exista o altă soluție, diferită de cea obținută prin metoda voastră, care să folosească mai puține intervale).

   Soluție: Pornind de la algoritmul implementat anterior, tot ce trebuie să modificăm pentru a obține în toate cazurile o soluție optimă este să reunim la fiecare pas intervalul „cel mai din dreapta" care poate fi folosit să extindă soluția curentă. O implementare în Python a acestei metode se poate găsi aici.

   Pentru a demonstra că în acest caz rezultatul este optim, putem să ne folosim de metoda reducerii la absurd. Dacă presupunem că soluția obținută de noi, notată SOL_{Greedy}, nu este optimă, înseamnă că există o altă mulțime de intervale SOL_{Opt} care acoperă intervalul inițial, cu $|SOL_{Opt}| < |U_{Greedy}|$.

   Putem presupune că cele două mulțimi coincid până la intervalul cu indicele k, iar apoi diferă:

$$SOL_{Greedy} = I_1, I_2, \dots, I_k, I_{k+1}, \dots, I_m$$
$$SOL_{Opt} = I_1, I_2, \dots, I_k, I_{k'+1}, \dots, I_{m'}$$

   unde $m' < m$.

Datorită modului în care am sortat și ales intervalele, știm că I_{k+1} este cu siguranță intervalul care are capătul din dreapta cel mai mare dintre intervalele compatibile cu soluția curentă. În particular, I_{k+1} se termină mai la dreapta față de (sau în același punct cu) $I_{k'+1}$.

Dacă ar fi să îl înlocuim pe $I_{k'+1}$ cu I_{k+1} în SOL_{Opt}, atunci soluția ar rămâne la fel (ca număr de intervale), sau cel mult s-ar putea să se reducă numărul de intervale necesare ca să fie o soluție validă (dar asta ar contrazice optimalitatea ei!). Tot aplicând aceste înlocuiri, am obține că soluția optimă are de fapt același număr de intervale ca soluția greedy, o contradicție cu $|SOL_{Opt}| < |U_{Greedy}|$. □

Referințe

[1] Wikipedia contributors, *Computational problem*, URL: https://en.wikipedia.org/wiki/Computational_problem.

[2] Wikipedia contributors, *Algorithm*, URL: https://en.wikipedia.org/wiki/Algorithm.

[3] Wikipedia contributors, *Heuristic*, URL: https://en.wikipedia.org/wiki/Heuristic_(computer_science).

[4] Wikipedia contributors, *Decision problem*, URL: https://en.wikipedia.org/wiki/Decision_problem.

[5] Wikipedia contributors, *Optimization problem*, URL: https://en.wikipedia.org/wiki/Optimization_problem.

[6] Information Security Stack Exchange Contributors, *Why shouldn't I store passwords in plaintext?*, 15 Apr. 2016, URL: https://security.stackexchange.com/a/120544.

[7] Wikipedia contributors, *bcrypt*, URL: https://en.wikipedia.org/wiki/Bcrypt.

[8] Wikipedia contributors, *Travelling salesman problem*, URL: https://en.wikipedia.org/wiki/Travelling_salesman_problem.

[9] Christophe-André Mario Irniger, *Graph matching: filtering databases of graphs using machine learning techniques*, Berlin: AKA, 2005, ISBN: 1-58603-557-6.

[10] Donald E. Knuth și Michael F. Plass, „Breaking paragraphs into lines", în *Software: Practice and Experience* 11.11 (1981), pp. 1119–1184, URL: http://www.eprg.org/G53DOC/pdfs/knuth-plass-breaking.pdf.

[11] Wikipedia contributors, *3SUM*, URL: https://en.wikipedia.org/wiki/3SUM.

[12] Karleigh Moore, Jimin Khim și Eli Ross, *Greedy Algorithm*, URL: https://brilliant.org/wiki/greedy-algorithm/.