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).

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.

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?

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 $\mathcal{O}(n^2)$?

3. (3SUM [6]) 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?

Metoda greedy

Mai multe informații despre principiile care stau la baza metodei greedy se pot găsi la [7].

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 $[a_1, b_1], [a_2, b_2], \ldots, [a_n, b_n]$, 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?

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

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).

5. (Fractional Knapsack Problem [8]). Ne jucăm un joc video în care suntem un negustor care trebuie să călătorească într-o țară îndepărtată, să cumpere diferite tipuri de grâne și apoi să se întoarcă și să le vândă în țara de origine pentru a obține un profit.

Ajunși în țara îndepărtată, descoperim că putem să alegem ce vrem să achiziționăm dintre n saci, fiecare cu un tip diferit de grână. Fiecare sac are o anumită **cantitate** (exprimată în *kilograme*) și o anumită **valoare totală** (exprimată în *lei*). Putem să achiziționăm sacii **întregi**, sau **orice cantitate** (procent) din fiecare sac. În același timp, nu putem căra mai mult de M kilograme (în total) cu noi.

1. Propuneți o strategie de cumpărare care să decidă ce saci achiziționăm din cei n și în ce cantități, astfel încât să **maximizăm profitul** pe care îl obținem când ne întoarcem în țară și vindem cantitățile de grâne achiziționate.

2. Demonstrați că strategia propusă aduce tot timpul profitul optim.

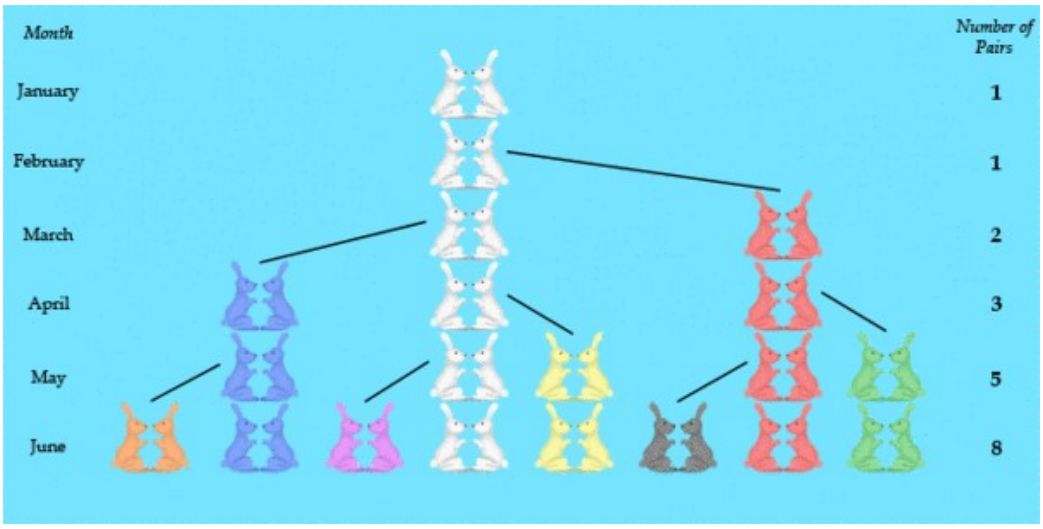
3. Ce se întâmplă dacă nu mai avem oportunitatea să achiziționăm orice fracție dintr-un sac de grâne, și suntem obligați să alegem dacă îl cumpărăm cu totul sau nu? Mai produce strategia anterior propusă rezultate optime?

Programare dinamică

6. Leonardo din Pisa (cunoscut și ca Fibonacci [9]) a fost un matematician italian din evul mediu care voia să descrie cum ar crește numeric o populație (imaginată) de iepuri.

La început, avem o pereche de iepuri tineri. După fiecare lună, perechile de iepuri tinere cresc și devin adulte, iar în fiecare lună, fiecare pereche de

iepuri adulți dă naștere la o nouă pereche de iepuri (inițial tineri). În acest experiment de gândire, iepurii nu mor niciodată.



Sursa imaginii: [10]

1. Descrieți un algoritm care să calculeze numărul de perechi de iepuri pe care i-am avea în scenariul de mai sus după n luni (i.e. care să calculeze **al n-ulea număr Fibonacci**), folosind formula de recurență

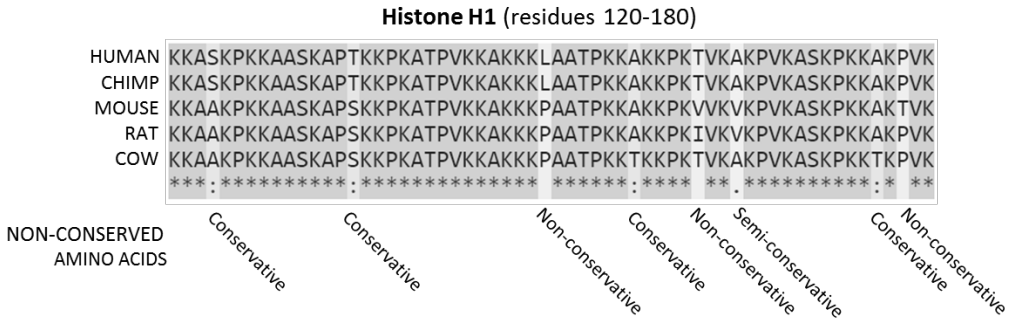
$$F_n = F_{n-1} + F_{n-2}.$$

   Ce complexitate de timp, respectiv spațiu, are algoritmul propus de voi?

2. Modificați algoritmul de la subpunctul anterior astfel încât **să salveze** într-o listă/într-un vector valorile intermediare F_i și **să le refolosească** pe măsură ce calculează al n-ulea număr Fibonacci. Ce complexitate de timp, respectiv spațiu, are noul algoritm?

3. Calculați al n-ulea număr Fibonacci într-un mod **iterativ**, fără recursivitate. Ce complexitate de timp, respectiv spațiu, avem în acest caz?

7. Pentru a identifica similarități funcționale/evoluționare/structurale între diferite secvențe de ADN sau ARN, în biologia moleculară există nevoia de a găsi **cele mai lungi subsecvențe (subșiruri) comune între două șiruri de caractere** (reprezentând nucleotide sau aminoacizi).



Sursa imaginii: [11]

1. Propuneți un algoritm **brute-force** care să rezolve problema celui mai lung subșir comun (nu contează eficiența, este în regulă dacă enumeră toate subșirurile posibile). Ce complexitate de timp, respectiv spațiu, are soluția propusă?

2. Dacă ați ști că ambele șiruri au **lungimea egală cu** 1 (un singur caracter), cum ați rezolva (cel mai eficient) problema?

3. Dar dacă ați ști că primul șir are **lungimea egală cu** 1 iar al doilea are **lungimea egală cu** n?

4. Propuneți un algoritm care să rezolve problema **în timp polinomial** (față de lungimile celor două șiruri), folosind **programare dinamică** (dacă nu vă vine nicio idee, îl puteți folosi pe cel din [12]). Ce complexitate de timp, respectiv spațiu, are acest algoritm?

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

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

[8] GeeksforGeeks, *Fractional Knapsack Problem*, URL: https://www.geeksforgeeks.org/fractional-knapsack-problem/.

[9] Wikipedia contributors, *Fibonacci*, URL: https://en.wikipedia.org/wiki/Fibonacci.

[10] The Department of Mathematics and Computer Science, Emory College, Oxford, *Fibonacci's Rabbits*, URL: http://mathcenter.oxford.emory.edu/site/math125/fibonacciRabbits/.

[11] Thomas Shafee, *Histone Alignment*, 8 Dec. 2014, URL: https://commons.wikimedia.org/w/index.php?curid=37188728.

[12] M. T. Goodrich și R. Tamassia, *Dynamic Programming: Longest Common Subsequences*, 2015, URL: https://www.ics.uci.edu/~goodrich/teach/cs260P/notes/LCS.pdf.