Algoritmi Avansați
Laborator 1

Gabriel Majeri

Indicații generale

Mod de lucru și evaluare

- Puteți rezolva exercițiile **în orice ordine doriți**. Punctajul pe fiecare laborator se acordă pe baza problemelor *rezolvate corect* și *prezentate*.
- Rezolvările trebuie **prezentate** fizic **în cadrul laboratorului în care au fost alocate** sau **cel târziu în laboratorul următor**. Dacă nu puteți ajunge la un laborator, puteți încerca să veniți la altă grupă, sau dacă din motive excepționale nu puteți face asta, luați legătura cu laborantul.
- Studenții care termină mai devreme laboratorul (fie au rezolvat toate problemele și le-au prezentat, fie nu își mai doresc să le lucreze) pot ieși în pauză sau pot pleca acasă. Prezența la laborator este **opțională**, dar **puternic recomandată** (având în vedere ponderea notei de la laborator în nota finală și abilitățile de programare pe care le puteți dobândi).

Recomandări

- Aveți voie să căutați soluții pe internet sau să discutați problemele cu colegii voștri, dar trebuie să indicați acest lucru, să redactați **individual** rezolvările și să fiți capabili să răspundeți la întrebări puse de laborant/profesor legate de modul în care ați gândit soluția voastră. Copierea temei fără atribuire și fără a o înțelege este sancțiune minoră conform Regulamentului de etică și profesionalism al FMI.
- Luați-vă timpul necesar să gândiți și să rezolvați corect problemele, în cel mai elegant mod în care puteți.

- Documentați codul folosind subprograme și variabile cu nume clare și, la nevoie, comentarii.

Exerciții

Metoda greedy

1. (Fractional Knapsack Problem [1]). 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. Implementați un subprogram care să citească datele de intrare (numărul de saci, greutățile lor, valorile lor, cantitatea maximă pe care o putem căra) și să le rețină în memorie. Puteți citi datele de la tastatură sau dintr-un fișier (recomandat, ca să nu stați tot timpul să le introduceți).

2. Implementați o strategie de cumpărare (un algoritm) 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.

3. (0-1 Knapsack Problem) 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? (nu trebuie să implementați nimic la acest subpunct, doar să vă gândiți la aplicabilitatea metodei greedy în acest caz)

2. (Huffman Coding [2]) O problemă pe care vrem să o rezolvăm deseori în informatică este compresia fără pierderi a datelor [3]: cum putem reduce **dimensiunea** unui fișier fără să pierdem din informații?

Una dintre cele mai simple (și vechi) metode de a reduce spațiul necesar pentru stocarea unor informații este codarea Huffman [2]. Acest algoritm este

folosit, în diferite forme, în majoritatea standarelor de compresie moderne (ZIP, JPEG, MP3 etc.).

Algoritmii de compresie se bazează pe faptul că informația pe care vrem să o comprimăm nu este complet aleatoare; tinde să aibă destul de multă **redundanță** (e.g.: cuvinte care se repetă, porțiuni duplicate din imagini etc.). Algoritmi de tip *entropy coding*, din care face parte și codarea Huffman, înlocuiesc elementele duplicate cu un șir (cod) care ocupă mai puțini biți, reducând astfel consumul total de memorie.

Obiectivul acestui exercițiu este să realizați și voi o implementare a algoritmului de codare/decodare Huffman, urmând etapele de mai jos. De asemenea, puteți consulta orice altă referință de pe internet, cum ar fi [4].

1. Implementați un subprogram care primește ca date de intrare un **șir de caractere** și construiește un **vector de frecvență** pentru acesta (poate fi reprezentat ca un dicționar sau orice altă structură de date asociativă). Șirul de caractere poate fi citit ulterior de la tastatură sau dintr-un fișier. Ce eficiență de timp/memorie are subprogramul implementat?

2. Scrieți un subprogram care primește ca date de intrare un **vector de frecvență** (așa cum l-ați construit anterior) și îl transformă într-un **arbore de codificare Huffman**, folosind algoritmul de codificare Huffman [4, 5] (care este un algoritm de tip greedy). Ce eficiență de timp/memorie are algoritmul implementat de voi?

3. Scrieți un subprogram care primește ca date de intrare un **șir de caractere** și un **arbore Huffman** (așa cum l-ați construit anterior), și realizează codificarea într-un **șir de caractere format din 0 și 1**. Ce complexitate de timp/memorie are subprogramul?

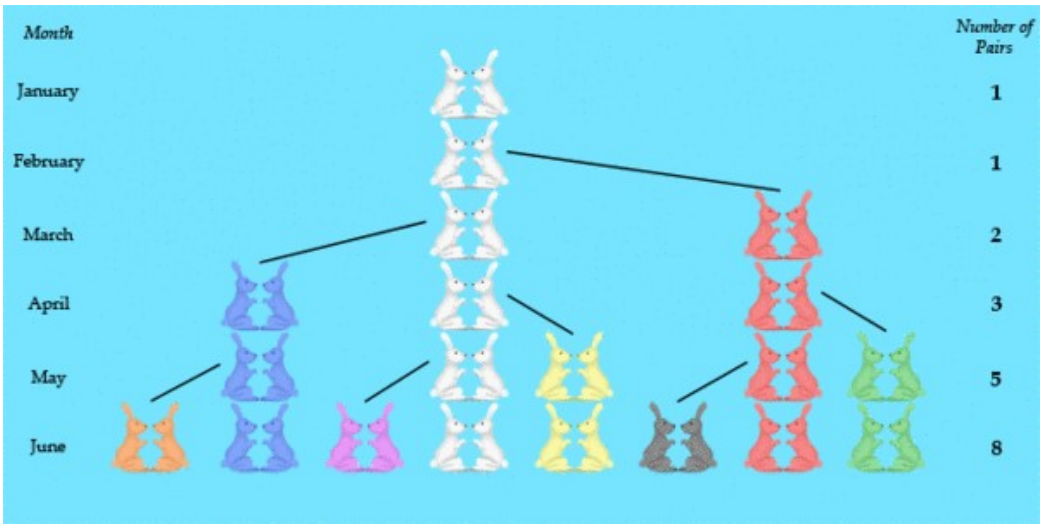
4. Scrieți un subprogram care primește ca date de intrare un **șir de caractere format din 0 și 1** și un **arbore Huffman** și realizează decodificarea lui înapoi în **șirul inițial**. Ce complexitate de timp/memorie are acest subprogram?

5. Testați algoritmul vostru de compresie pe acest document de ≈ 5 MB, care **conține operele complete ale lui Shakespeare** în format text. Având în vedere că un caracter ASCII obișnuit ar ocupa 8 biți, iar șirul de 0 și 1 pe care îl generează programul vostru ar putea fi stocat pe biți (deci ar consuma de 8 ori mai puțin spațiu decât ca șir de caractere 0 și 1), cât de eficientă este compresia implementată de voi în acest caz (ignorând dimensiunea arborelui Huffman)?

Programare dinamică

3. Leonardo din Pisa (cunoscut și ca Fibonacci [6]) 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: [7]

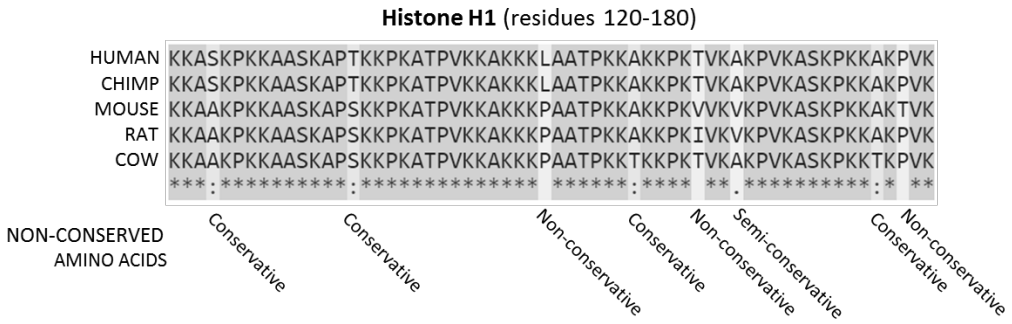
1. Implementați un algoritm care să calculeze în mod recursiv numărul de perechi de iepuri pe care i-am avea în scenariul de mai sus după n luni (i.e. care să calculeze **al n-ulea număr Fibonacci**), folosind formula de recurență
$$F_n = F_{n-1} + F_{n-2}.$$
Ce complexitate de timp, respectiv spațiu, are algoritmul propus de voi? (puteți nota asta sub formă de comentarii)

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. Implementați un algoritm care să calculeze al n-ulea număr Fibonacci într-un mod **iterativ**, fără recursivitate. Ce complexitate de timp, respectiv spațiu, avem în acest caz?

4. 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 comune între două șiruri de caractere** (reprezentând nucleotide sau aminoacizi).

Observație: Subsecvențele nu sunt *subșiruri*! O subsecvență poate avea mai multe caractere care nu sunt pe poziții imediat consecutive. Dar ordinea relativă a acestora se păstrează.



Sursa imaginii: [8]

1. Implementați un algoritm de tipul **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. Implementați 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 [9]). Ce complexitate de timp, respectiv spațiu, are acest algoritm?

Referințe

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

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

[3] Wikipedia contributors, *Lossless compression*, URL: https://en.wikipedia.org/wiki/Lossless_compression.

[4] Techie Delight, *Huffman Coding Compression Algorithm*, URL: https://www.techiedelight.com/huffman-coding/.

[5] GeeksforGeeks, *Huffman Coding | Greedy Algo-3*, URL: https://www.geeksforgeeks.org/huffman-coding-greedy-algo-3/.

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

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

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

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