SEMINAR 14

1. METODA GREEDY

RECAP

- Găsește soluția incremental: construiește soluții prin extinderea lor treptată
- La fiecare pas, soluția este extinsă **cu cel mai bun candidat dintre candidații rămași la un moment dat** („greedy" = lacom) - se adaugă succesiv la rezultat elementul care realizează optimul local
- O decizie luată pe parcurs nu se mai modifică ulterior

Pentru un algoritm Greedy, avem următoarele elemente

- Mulțimea candidat (**c**): de unde se aleg elementele soluției
- Funcția de selecție (**select_most_promising**): alege cel mai bun candidat pentru a fi adăugat la soluție
- Funcția acceptabil (**is_acceptable)**: determină dacă un candidat poate fi folosit pentru a obține o soluție
- Funcția obiectiv: valoarea pentru soluții parțiale/soluție
- Funcția soluție (**is_solution**): determină dacă soluția parțială dată ca parametru este soluție

---

PROBLEME

În curs: Se dă o sumă M și tipuri de bancnote. Să se găsească o modalitate de a plăti suma M de bani folosind cât mai puține bancnote.

GREEDY1.

Se dă o mulțime de *n* activități care doresc să folosească o sală de lectură. Sala de lectură poate fi folosită de o singură activitate la un anumit moment de timp. Fiecare activitate are un *timp de pornire*, notat cu s_i, și un timp de oprire f_i, unde $s_i \leq f_i$. Dacă este selectată activitatea *i,* ea se desfășoară în intervalul $[s_i, f_i]$. Activitățile i și j se numesc **compatibile** dacă intervalele acestora nu se intersectează ($s_i \geq f_j$ sau $s_j \leq f_i$).

Din mulțimea dată de activități, selectați o mulțime maximală de activități compatibile pentru a fi programate în sala de lectură.

Exemple:
Dacă mulțimea activităților (11 activități) este:

$$A = \{a_1 = (1, 3), a_2 = (2, 5), a_3 = (4, 8)\}$$

Mulțimea maximală: $\{a_1, a_2\}$.

Dacă mulțimea activităților (11 activități) este:

$$A = \left\{ \begin{matrix} a_1 = (1,4), a_2 = (3,5), a_3 = (0,6), \\ a_4 = (5,7), a_5 = (3,8), a_6 = (5,9), \\ a_7 = (6,10), a_8 = (8,11), a_9 = (8,12), \\ a_{10} = (2,13), a_{11} = (12,14) \end{matrix} \right\}$$

Mulțimea maximală: $\{a_1, a_4, a_8, a_{11}\}$

GREEDY2. Fractional Knapsack Problem

Un hoț care jefuiește un magazin găsește n obiecte. Obiectul i are valoarea v_i şi greutatea w_i, unde v_i şi w_i sunt numere întregi. El doreşte să ia o încărcătură cât mai valoroasă posibil, dar nu poate căra în sac o greutate mai mare decât W, unde W este număr întreg. Ce obiecte trebuie să ia? Hoţului îi este permis să ia şi părţi din obiecte.

Sidenote:

0-1 Knapsack Problem - hoţul poate să ia un obiect în întregime sau deloc, nu poate lua părţi din obiecte (rezolvăm cu programare dinamică)

Fractional: praf de aur

0-1: lingou de aur

GREEDY3. Dijkstra's Shortest Path Algorithm

Given a graph and a source vertex in the graph, find the shortest paths from the source to all vertices in the given graph.

2. PROGRAMARE DINAMICĂ

RECAP

Principiul optimalității

[Bellman, 1950] "An optimal policy has the property that whatever the initial state and initial decision are, the remaining decisions must constitute an optimal policy with regard to the state resulting from the first decision."

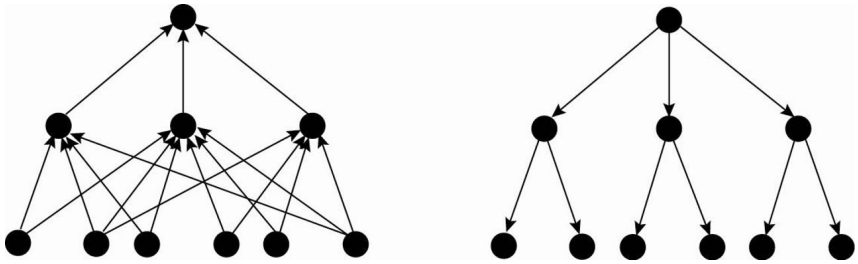
It can be summarized simply as follows: "*every optimal policy consists only of optimal sub policies.*"

Substructura optimă

O problemă prezintă o substructură optimă dacă o soluție optimă a problemei poate fi obținută folosind soluțiile optime pentru subproblemele sale.

Suprapunerea problemelor:

O problemă are probleme suprapuse dacă aceasta poate fi împărțită în subprobleme pentru care rezultatul este folosit de mai multe ori (refolosit).



Programare dinamică (stânga) vs. Divide&Conquer (dreapta): subprobleme suprapuse vs. subprobleme independente

Memoization vs. Tabulation

- Memoization:
  - Salvarea rezultatelor pentru a evita recalcularea
  - generally used to optimize recursion, top-down approach
- Tabulation:
  - Începem cu subproblema cea mai simplă, și construim soluțiile optime pentru fiecare pas
  - generally iterative, bottom-up approach

PROBLEME

În curs: cea mai lungă sublistă[1] crescătoare

1. Se dă o listă cu elemente numere întregi. Să se determine subsecvența[2] cu suma cea mai mare, și suma acesteia.

Exemplu

Pentru lista

-2	1	-3	4	-1	2	1	-5	4

suma maximă este 6, iar subsecvența pentru care obținem această sumă este colorată cu verde.

ALTE PROBLEME (to practice for exam)

1. Se dă o listă de numere. Verificați dacă această listă poate fi împărțită în două subseturi cu sumă egală. Afișați o posibilă soluție (două subseturi cu sumă egală).

2. Se dă o listă de numere naturale. Împărțiți lista dată în două subseturi între care diferența să fie cât mai mică. Afișați o soluție (două subseturi cu această proprietate) și diferența minimă.

---

[1] [în cadrul acestor exerciții] dacă avem lista [1,3,4,9,2,16], **subliste** sunt și liste precum: [1,3,9], [1,3,9,16], [1,4,2] etc (se poate sări peste elemente, însă ordinea acestora se păstrează

[2] [în cadrul acestor exerciții] **Subsecvență**: doar liste în care avem elemente de pe poziții alăturate (e.g. pentru lista anterioară, [1,3,4], [3,4,9], [2,16] etc)

3. Se dă o listă de numere naturale și o valoare k. Afișați un subset al listei pentru care suma este k, sau un mesaj de eroare dacă nu există un astfel de subset.

4. Se dă o listă de elemente . Maximizați valoarea expresiei A[m] - A[n] + A[p] - A[q], unde m, n, p, q sunt poziții (indici) din listă, iar m > n > p > q.

5. 0-1 Knapsack Problem

6. Longest Common Subsequence (atenție, aici subsequence!=subsecvență folosită în exercițiile de mai sus, = sublistă)

7. Cutting a rod