Proiect - Analiza algoritimilor
Subset sum

January 16, 2026

1 Introducere

1.1 Descrierea problemei

Problema *Subset Sum* este una dintre cele mai cunoscute probleme fundamentale din teoria algoritmilor și a complexității computaționale. Aceasta poate fi formulată astfel: fiind dată o mulțime finită de numere întregi pozitive $A = \{a_1, a_2, \ldots, a_n\}$ și o valoare țintă S, se cere să se determine dacă există un subansamblu al lui A a cărui sumă este exact egală cu S.

Formal, problema urmărește existența unui subset $A' \subseteq A$ astfel încât:

$$\sum_{a_i \in A'} a_i = S$$

Subset Sum aparține clasei problemelor **NP-Complete** în varianta sa de decizie. Deși enunțul este simplu, complexitatea sa crește exponențial în raport cu dimensiunea mulțimii de intrare, ceea ce face imposibilă, în general, găsirea unei soluții eficiente pentru instanțe mari folosind algoritmi exacți.

Importanța acestei probleme derivă atât din rolul său teoretic – fiind utilizată frecvent în demonstrații de NP-dificultate prin reducere – cât și din numeroasele aplicații practice în care apare sub diverse forme. În practică, problema este abordată folosind o varietate de strategii algoritmice, de la metode brute-force și backtracking, la programare dinamică, optimizări pseudo-polinomiale și algoritmi aproximați sau euristici.

În cadrul acestui studiu, ne concentrăm pe principalele abordări algoritmice utilizate pentru rezolvarea problemei *Subset Sum*, punand in evidenta legaturile intre corectitudine, timpul de executie si consumul de memorie

1.2 Aplicații practice

Deși este o problemă teoretică dificilă, *Subset Sum* apare în mod natural într-o varietate de domenii practice, printer care se Numara criptografia, managementul resurselor si al bugetelor, optimizare si planificare, si nu in ultimul rand, in inteligenta artificiala.

În **criptografie**, problema stă la baza unor scheme criptografice clasice, precum sistemul Merkle–Hellman, care utilizează dificultatea *Subset Sum* pentru a asigura securitatea mesajelor. De asemenea, variante ale problemei sunt întâlnite în atacuri criptanalitice și în analiza securității algoritmilor de criptare.

În **managementul resurselor și al bugetelor**, problema poate fi utilizată pentru a decide dacă un anumit buget poate fi atins prin alegerea unui subset de costuri sau investiții, fiind importantă în planificare financiară sau alocarea eficientă a resurselor.

În **optimizare și planificare**, *Subset Sum* apare ca subproblemă în probleme mai complexe, precum *Knapsack*, *scheduling* sau partitionarea mulțimilor, unde se urmărește împărțirea unui set de valori în grupuri cu proprietăți bine definite.

Problema este utilizată în **inteligență artificială** și **analiza combinatorială**, în contexte care implică explorarea spațiilor de căutare sau satisfacerea unor constrângeri numerice.

Datorită acestor aplicații, studiul algoritmilor care rezolvă problema *Subset Sum* este necesar din perspectivă teoretică și practică, justificând analiza detaliată a performanței algoritmilor pe seturi diverse de date.

1.3 Construirea seturilor de teste

Pentru evaluarea algoritmului analizat am creat un set de teste sintetice, care să acopere atât cazuri uzuale, cât şi numeroase situaţii de tip *corner-case*. Acelaşi set de intrări va fi folosit şi în eventuale comparaţii cu alte implementări ale problemei *Subset Sum*, astfel încât evaluarea să fie corectă.

Fiecare fişier de test conţine urmatoarele:

- linia 1: Numărul N ce reprezintă numărul de elemente din tablou şi T reprezintă suma ţintă;
- următoarele N linii: elementele tabloului, aşezate individual pe câte o linie .

Pentru fiecare fişier de intrare a fost generat şi un fişier de ieşire asociat, care conţine soluţia corectă a instanţei respective. Formatul fişierelor de ieşire respectă cerinţele problemei:

- pe prima linie se află numărul k de elemente selectate din mulţime;
- pe a doua linie se află indicii (1-based) ai elementelor care formează un subset valid a cărui sumă este exact T;
- dacă există mai multe soluţii posibile, a fost selectată arbitrar una dintre ele;
- dacă nu există nicio soluţie, prima linie conţine valoarea 0, iar a doua linie rămâne goală.

Acest format permite verificarea automată a corectitudinii atât pentru implementări care returnează doar verdictul (`true`/`false`), cât şi pentru cele care reconstruiesc efectiv subsetul soluţie. Checker-ul poate confirma uşor validitatea soluţiei verificând:

$$\sum_{i \in indices} v_i = T.$$

1.3.1 Teste de dimensiune mică şi medie

Primul set include teste cu dimensiuni reduse (câţiva până la câteva zeci de elemente), folosite în principal pentru validarea corectitudinii. Acestea acoperă explicit:

- cazuri de bază: un singur element, egal sau diferit de suma ţintă;
- situaţii în care suma ţintă este zero, fie în prezenţa unui tablou nevid, fie cu multe valori zero (unde subsetul vid trebuie recunoscut corect ca soluţie);
- cazuri în care suma ţintă depăşeşte suma tuturor elementelor, pentru care răspunsul trebuie să fie `false`;
- tablouri formate din numere pare cu ţintă impară, respectiv din multipli ai unei valori fixe (de exemplu doar multipli de 5) cu o ţintă care nu este multiplu al acesteia, astfel încât problema devine imposibilă din motive aritmetice (paritate/divizibilitate);
- tablouri cu valori repetate (de exemplu toate elementele egale cu 5 sau cu 0), pentru a verifica tratamentul corect al multiset-urilor şi al combinaţiilor cu mulţi termeni identici;
- tablouri mixte, nesortate, cu valori de mărimi diferite, pentru a evita orice presupunere accidentală asupra ordonării datelor.

Pentru acest grup de teste am inclus atât instanţe cu răspuns `true`, cât şi instanţe `false`, frecvent construite în perechi pe acelaşi tablou (de exemplu aceeaşi secvenţă de numere, dar cu două valori diferite pentru suma ţintă). Această structură ne permite să verificăm nu doar corectitudinea logicii, ci şi sensibilitatea algoritmului la schimbări mici ale intrării.

1.3.2 Teste mari (peste 100 de elemente)

Al doilea grup conţine teste cu peste 100 de elemente, concepute pentru a stresa mai puternic implementarea şi pentru a evidenţia comportamentul său pe instanţe mai mari. Am urmărit să combinăm structuri cu semnificaţie combinatorială clară cu situaţii de tip corner-case:

- teste cu foarte multe zerouri urmate de o secvenţă de valori consecutive (de forma $0, \ldots, 0, 1, 2, \ldots, 50$) şi sumă ţintă $S = 0$, care verifică tratamentul

corect al cazului *"sumă zero"* în prezenţa unui număr mare de elemente inutile din punct de vedere al sumei;

- teste cu peste 100 de numere pare şi ţintă impară, respectiv cu mulţi multipli de 5 şi ţintă congruentă cu 1 mod 5, pentru a crea instanţe mari, dar în mod structural imposibile;
- teste cu valori consecutive (de exemplu $1, 2, \ldots, 150$) în care ţinta este suma totală a tabloului; în aceste cazuri singurul subset valid este întregul tablou, ceea ce obligă implementarea să nu taie" prematur ramuri importante din spaţiul de căutare;
- teste cu combinaţii de multe valori mici (de exemplu zeci de valori 0 şi 1) plus câteva valori mari (de tip $50, 75, 100, 125, 150$), ceea ce creează un număr foarte mare de combinaţii posibile pentru aceeaşi sumă şi reprezintă un scenariu dificil pentru o abordare recursivă/backtracking.

Aceste teste sunt utilizate în principal pentru analiza performanţei pe instanţe de dimensiune medie, în care timpul de execuţie începe să devină sensibil la structura datelor de intrare.

1.3.3 Teste foarte mari (peste 1000 de elemente)

Pentru a evalua comportamentul algoritmului pe instanţe de dimensiune mare, am construit şi două teste cu peste 1000 de elemente:

- un test *"normal"*, cu tablou format din valori consecutive $1, 2, \ldots, 1100$ şi sumă ţintă egală cu suma primelor 700 elemente. Acest test reprezintă un caz în care există multe subseturi posibile care ating ţinta, dar în acelaşi timp dimensiunea mare a tabloului pune în evidenţă limitele unei abordări exponenţiale;
- un test de tip *corner-case* cu 1200 de numere pare, de forma $2, 4, 6, \ldots, 2400$, şi sumă ţintă impară. Din punct de vedere teoretic, orice sumă a elementelor va fi pară, deci instanţa este imposibilă. Testul este util pentru a observa cum se comportă algoritmul pe un input foarte mare în care răspunsul este `false` şi nu există soluţii care să oprească rapid explorarea.

Aceste instanţe foarte mari sunt folosite în special pentru evaluarea timpului de execuţie şi a consumului de memorie, fiind suficiente pentru a evidenţia clar limitele de scalabilitate ale implementării analizate.