

TEHNICA DE PROGRAMARE "BACKTRACKING"

1. Prezentare generală

Tehnica de programare Backtracking este utilizată, de obicei, pentru determinarea tuturor soluțiilor unei probleme într-un mod progresiv, astfel încât să se evite generarea întregului spațiu al soluțiilor problemei. Practic, soluțiile se construiesc componentă cu componentă și se testează la fiecare pas validitatea lor (se verifică dacă sunt *soluții parțiale*), iar în cazul în care se constată faptul că nu se poate obține o soluție a problemei plecând de la soluția parțială curentă, aceasta este abandonată. Astfel, se evită o *rezolvare de tip forță-brută* a problemei, adică generarea și testarea tuturor soluțiilor posibile pentru a determina soluțiile problemei. Totuși, complexitatea algoritmilor de tip Backtracking rămâne una ridicată, deoarece se va genera și testa un procent semnificativ din întregul spațiu al soluțiilor.

De exemplu, să considerăm problema generării tuturor permutărilor mulțimii $A = \{1, 2, \dots, n\}$ pentru $n = 6$.

O rezolvare de tip forță-brută presupune generarea tuturor posibilelor soluții, adică a tuplurilor de forma $p = (p_1, p_2, p_3, p_4, p_5, p_6)$ cu $p_1, \dots, p_6 \in \{1, 2, \dots, 6\}$, și selectarea celor care sunt permutări, adică au toate valorile diferite între ele ($p_1 \neq p_2 \neq \dots \neq p_6$). Se observă foarte ușor faptul că se vor genera și testa $6^6 = 46656$ tupluri, din care doar $6! = 720$ vor fi permutări, deci eficiența acestei metode este foarte mică - în jurul unui procent de 1.5%! Eficiența scăzută a acestei metode este indusă de faptul că se generează multe tupluri inutile, care nu sunt sigur permutări. De exemplu, se vor genera toate tuplurile de forma $p = (1, 1, p_3, p_4, p_5, p_6)$, adică $6^4 = 1296$ de tupluri inutile deoarece $p_1 = p_2$, deci, evident, aceste tupluri nu pot fi permutări!

O rezolvare de tip Backtracking presupune generare progresivă a soluțiilor, evitând generarea unor tupluri inutile, astfel (vom ține cont de faptul că $p_1, \dots, p_6 \in \{1, 2, \dots, 6\}$):

- $p = (1)$ - este o soluție parțială (componentele sale sunt diferite între ele, deci poate fi o permutare), dar nu este o soluție a problemei (nu are 6 componente), astfel că trebuie să adăugăm cel puțin încă o componentă;
- $p = (1, 1)$ - nu este o soluție parțială (componentele sale sunt egale, deci nu vom obține o permutare indiferent de ce valori vom adăuga în continuare), astfel că nu are sens să adăugăm încă o componentă, ci vom genera următorul tuplu tot cu două componente;
- $p = (1, 2)$ - este o soluție parțială, dar nu este o soluție a problemei;
- $\left. \begin{array}{l} p = (1, 2, 1) \\ p = (1, 2, 2) \end{array} \right\}$ - nu sunt soluții parțiale;
- $p = (1, 2, 3)$ - este o soluție parțială, dar nu este o soluție a problemei;
- $\left. \begin{array}{l} p = (1, 2, 3, 1) \\ p = (1, 2, 3, 2) \\ p = (1, 2, 3, 3) \end{array} \right\}$ - nu sunt soluții parțiale;
- $p = (1, 2, 3, 4)$ - este o soluție parțială, dar nu este o soluție a problemei;

- $$\left. \begin{array}{l} p = (1,2,3,4,1) \\ p = (1,2,3,4,2) \\ p = (1,2,3,4,3) \\ p = (1,2,3,4,4) \end{array} \right\} - \text{nu sunt soluții parțiale;}$$
- $p = (1,2,3,4,5)$ - este o soluție parțială, dar nu este o soluție a problemei;
- $$\left. \begin{array}{l} p = (1,2,3,4,5,1) \\ p = (1,2,3,4,5,2) \\ p = (1,2,3,4,5,3) \\ p = (1,2,3,4,5,4) \\ p = (1,2,3,4,5,5) \end{array} \right\} - \text{nu sunt soluții parțiale;}$$
- $p = (1,2,3,4,5,6)$ - este o soluție parțială care este și soluție a problemei, deci o memorăm sau o prelucrăm (de exemplu, o afișăm), după care vom încerca generarea următorul tuplu. Deoarece ultima componentă are valoarea 6 (ultima posibilă), înseamnă că am epuizat toate valorile posibile pentru aceasta, deci o vom elimina și vom genera următorul tuplu format doar din 5 componente;
- $p = (1,2,3,4,6)$ - este o soluție parțială, dar nu este o soluție a problemei;
- $$\left. \begin{array}{l} p = (1,2,3,4,6,1) \\ p = (1,2,3,4,6,2) \\ p = (1,2,3,4,6,3) \\ p = (1,2,3,4,6,4) \end{array} \right\} - \text{nu sunt soluții parțiale;}$$
- $p = (1,2,3,4,6,5)$ - este o soluție parțială care este și soluție a problemei, deci o memorăm sau o prelucrăm, după care vom genera următorul tuplu;
- $p = (1,2,3,4,6,6)$ - nu este o soluție parțială, dar ultima componentă are valoarea maxim posibilă 6, deci o vom elimina și vom genera următorul tuplu format doar din 5 componente;
- $p = (1,2,3,4,6)$ - ultima componentă are valoarea maxim posibilă 6, deci o vom elimina și vom genera următorul tuplu format doar din 4 componente;
- $p = (1,2,3,5)$ - este o soluție parțială, dar nu este o soluție a problemei;
-
- $p = (6,5,4,3,2,1)$ - este o soluție parțială care este și ultima soluție a problemei, deci o memorăm sau o prelucrăm, după care vom încerca generarea următorului tuplu. Se observă cu ușurință faptul că, pe rând, nu vom mai găsi niciun tuplu convenabil, deci algoritmul se va termina.

Observație importantă: Determinarea exactă a complexității unui algoritm de tip Backtracking nu este simplă. Totuși, plecând de la observația că un algoritm nu poate avea o complexitate mai mică decât citirea datelor de intrare și/sau afișarea datelor de ieșire, de obicei, putem aproxima complexitatea unui astfel de algoritm prin numărul soluțiilor pe care el le afișează. De exemplu, algoritmul pentru generarea permutărilor de ordin n are complexitatea minimă egală cu $\mathcal{O}(n!)$, deoarece vor fi afișate $n!$ permutări. O astfel de complexitate este foarte mare, depășind-o pe cea exponențială!

2. Forma generală a unui algoritm de tip Backtracking

Vom începe prin a preciza faptul că majoritatea problemele de generare pot fi formalizate astfel: "Fie mulțimile nevide A_1, A_2, \dots, A_n și un predicat $P: A_1 \times \dots \times A_n \rightarrow \{0,1\}$. Să se genereze toate tuplurile de forma $S = (s_1, s_2, \dots, s_n)$ pentru care $s_1 \in A_1, \dots, s_n \in A_n$ și $P(s_1, s_2, \dots, s_n) = 1$ ". Practic, predicatul P cuantifică o proprietate pe care trebuie să o îndeplinească componentele tuplului S (o soluție a problemei) sub forma unei funcții de tip boolean ($0 = \text{fals}$ și $1 = \text{adevărat}$).

De exemplu, problema generării tuturor permutărilor poate fi formalizată în acest mod considerând $A_1 = \dots = A_n = \{1, 2, \dots, n\}$ și $P(s_1, s_2, \dots, s_n) = (s_1 \neq s_2) \text{ AND } (s_2 \neq s_3) \text{ AND } \dots \text{ AND } (s_{n-1} \neq s_n)$.

În continuare, vom prezenta câteva notații, definiții și observații:

- pentru orice componentă s_k vom nota cu \min_k cea mai mică valoare posibilă a sa, iar cu \max_k pe cea mai mare;
- într-un tuplu (s_1, s_2, \dots, s_k) , componenta s_k se numește *componentă curentă* (i.e., asupra sa se acționează în momentul respectiv);
- *condițiile de continuare* reprezintă condițiile pe care trebuie să le îndeplinească tuplul curent (s_1, s_2, \dots, s_k) astfel încât să aibă sens extinderea sa cu o nouă componentă s_{k+1} sau, altfel spus, există valori pe care le putem adăuga la el astfel încât să obținem o soluție $S = (s_1, \dots, s_n)$ a problemei;
- tuplul curent (s_1, \dots, s_k) este o *soluție parțială* dacă îndeplinește condițiile de continuare;
- *condițiile de continuare* se deduc din predicatul P și sunt neapărat necesare, fără a fi întotdeauna și suficiente;
- orice *soluție* a problemei este implicit și soluție parțială, dar trebuie să mai îndeplinească și alte condiții suplimentare.

Folosind observațiile anterioare, forma generală a unui algoritm de tip Backtracking, implementat folosind o funcție recursivă este următoarea:

```
# k reprezintă indicele componentei curente s[k]
# dintr-o listă s indexată de la 1
def bkt(k):
    global s
    # parcurgem toate valorile posibile v pentru s[k]
    for v in range(mink, maxk+1):
        # atribuim componentei curente s[k] valoarea v
        s[k] = v

        # dacă s[1],...,s[k] este soluție parțială
        if s[1],...,s[k] este soluție parțială:
            # dacă s[1],...,s[k] este o soluție
            if s[1],...,s[k] este soluție:
                # prelucrăm soluția curentă s[1],...,s[k]
            else:
                # s[1],...,s[k] este soluție parțială, dar nu este
                # soluție, deci adăugăm o nouă componentă s[k + 1]
                bkt(k+1)
```

Referitor la algoritmul general de Backtracking prezentat mai sus trebuie făcute câteva observații:

- bucățile de cod scrise cu roșu trebuie particularizate pentru fiecare problemă;
- am considerat tabloul s indexat de la 1, ci nu de la 0, pentru a permite o scriere naturală a unor condiții în care se utilizează indicii tabloului;
- \min_k și \max_k se deduc din semnificația componentei $s[k]$ a unei soluții;
- pentru a testa că $s[1], \dots, s[k]$ este soluție vom ține cont de faptul că $s[1], \dots, s[k]$ este soluție parțială, deci nu vom retesta condițiile de continuare, ci doar pe cele suplimentare lor;
- dacă $s[1], \dots, s[k]$ nu este soluție parțială, atunci nu vom adăuga o nouă componentă $s[k+1]$ prin apelul recursiv $\text{bkt}(k+1)$, deci instrucțiunea `for` va continua și componentei curente $s[k]$ i se va atribui următoarea valoare posibilă v , dacă aceasta există, iar în cazul în care aceasta nu există, instrucțiunea `for` corespunzătoare componentei curente $s[k]$ se va termina și, implicit, apelul funcției `bkt` corespunzător, deci se va reveni la componenta anterioară $s[k-1]$.

De exemplu, pentru a genera toate permutările de ordin n , observațiile de mai sus se particularizează, astfel:

- $s[k]$ reprezintă un element al permutării, deci $\min_k = 1$ și $\max_k = n$;
- $s[1], \dots, s[k-1], s[k]$ este soluție parțială dacă valoarea componentei curente $s[k]$ nu a mai fost utilizată anterior, adică $s[k] \neq s[i]$ pentru orice $i \in \{1, 2, \dots, k-1\}$. Se observă faptul că această condiție este dedusă din predicatul P (care impune ca toate cele n valori $s[1], \dots, s[n]$ dintr-o permutare de ordin n să fie distincte) și este neapărat necesară (dacă $s[1], \dots, s[k]$ nu sunt distincte, atunci, indiferent de ce valori am atribui celorlalte $n-k$ componente $s[k+1], \dots, s[n]$ nu vom obține o permutare de ordin n), fără a fi și suficientă (dacă valorile $s[1], \dots, s[k]$ sunt distincte nu înseamnă că ele formează o permutare de ordin n , ci trebuie impusă condiția suplimentară $k=n$);
- pentru a testa că $s[1], \dots, s[k]$ este soluție vom ține cont de faptul că $s[1], \dots, s[k]$ este soluție parțială, deci nu vom retesta condițiile de continuare ($s[k] \neq s[i]$ pentru orice $i \in \{1, 2, \dots, k-1\}$), ci doar condiția suplimentară $k=n$.

Aplicând observațiile anterioare în forma generală a algoritmului de Backtracking, obținem următoarea implementare în limbajul Python a algoritmului de generare a tuturor permutărilor mulțimii $\{1, 2, \dots, n\}$:

```
def bkt(k):
    global s, n

    for v in range(1, n+1):
        s[k] = v
        if s[k] not in s[:k]:
            if k == n:
                print(*s[1:], sep=",")
            else:
                bkt(k+1)
```

```

n = int(input("n = "))
# o soluție s va avea n elemente
s = [0]*(n+1)
print("Toate permutările de lungime " + str(n) + ":")
bkt(1)

```

Așa cum am menționat deja, complexitatea minimă a acestui algoritm este $\mathcal{O}(n!)$.

3. Probleme de generări combinatoriale

Pe lângă generarea permutărilor unei mulțimi, algoritmi de tip Backtracking mai pot fi utilizați și pentru rezolvarea altor probleme de generări combinatoriale, cum ar fi generarea aranjamentelor sau a combinărilor unei mulțimi. În continuare, vom exemplifica algoritmi pe care îi vom prezenta pentru mulțimea $A = \{1, 2, \dots, n\}$, deoarece elementele oricărei alte mulțimi cu n elemente pot fi accesate considerând elementele mulțimii A ca fiind indicii elementelor sale.

3.1. Generarea aranjamentelor

Aranjamentele cu m elemente ale unei mulțimi cu n elemente ($m \leq n$) reprezintă toate tuplurile care se pot forma utilizând m elemente distincte dintre cele n ale mulțimii. Numărul lor se notează cu A_n^m și este dat de formula $\frac{n!}{(n-m)!}$. De exemplu, numărul aranjamentelor cu $m = 3$ elemente ale unei mulțimi cu $n = 5$ elemente este $A_5^3 = 60$, o parte a lor fiind: $(1, 2, 3), (1, 3, 2), \dots, (3, 2, 1), \dots, (1, 3, 5), \dots, (5, 3, 1), \dots, (3, 4, 5), \dots, (5, 4, 3)$.

Se observă foarte ușor faptul că singura diferență față de generarea permutărilor o constituie lungimea unei soluții, care în acest caz este m în loc de n . De fapt, pentru $m = n$, aranjamentele unei mulțimi sunt chiar permutările sale!

La fel ca și în cazul generării permutărilor, putem aproxima complexitatea minimă a acestui algoritm prin numărul soluțiilor pe care le va afișa, deci prin $\mathcal{O}(A_n^m)$.

3.2. Generarea combinărilor

Combinările cu m elemente ale unei mulțimi cu n elemente ($m \leq n$) reprezintă toate submulțimile cu m elemente ale unei mulțimi cu n elemente. Numărul lor se notează cu C_n^m și este dat de formula $\frac{n!}{m!(n-m)!}$. De exemplu, numărul tuturor submulțimilor cu $m = 3$ elemente ale unei mulțimi cu $n = 5$ elemente este $C_5^3 = 10$, toate aceste submulțimi fiind: $\{1, 2, 3\}, \{1, 2, 4\}, \{1, 2, 5\}, \{1, 3, 4\}, \{1, 3, 5\}, \{1, 4, 5\}, \{2, 3, 4\}, \{2, 3, 5\}, \{2, 4, 5\}$ și $\{3, 4, 5\}$.

Spre deosebire de tupluri, în care contează ordinea elementelor (de exemplu, tuplurile $(1, 2, 3)$ și $(1, 3, 2)$ sunt considerate diferite), în cazul submulțimilor aceasta nu contează (de exemplu, submulțimile $\{1, 2, 3\}$ și $\{3, 1, 2\}$ sunt considerate egale). Din acest motiv, trebuie să găsim o posibilitate de a evita prelucrarea unei soluții care are aceleași elemente ca o altă soluție generată anterior, dar în altă ordine. În acest sens, o variantă simplă, dar ineficientă, o reprezintă prelucrarea doar a soluțiilor care au elementele în ordine strict crescătoare, dar astfel vom încălca chiar principiul de bază al metodei Backtracking, acela de a evita generarea și testarea unor tupluri inutile cât mai devreme posibil. O altă variantă o reprezintă generarea doar a soluțiilor cu elemente în ordine

strict crescătoare, această restricție putând fi impusă soluției curente în mai multe etape ale unui algoritm de tip Backtracking:

- *când testăm condițiile de continuare, verificând faptul că $s[k] > s[k-1]$ – această variantă este mai eficientă decât prima, dar, totuși se vor genera și testa multe tupluri inutile. De exemplu, pentru a extinde soluția parțială (1, 3), se vor genera și testa inutil tuplurile (1,3,1), (1,3,2) și (1,3,3), deși este evident faptul că la tuplul (1, 3) are sens să adăugăm doar o valoare cel puțin egală cu 4;*
- *inițializând componenta curentă cu prima valoare strict mai mare decât componenta anterioară ($\min_k = s[k-1]+1$) – este cea mai eficientă variantă posibilă, deoarece nu se generează și testează tupluri inutile și, mai mult, orice tuplu este soluție parțială (elementele sale sunt generate direct în ordine strict crescătoare, deci sunt distincte), ceea ce înseamnă că putem renunța la testarea condițiilor de continuare!*

Astfel, vom obține următorul program eficient de tip Backtracking pentru generarea combinațiilor:

```
def bkt(k):
    global n, m, sol, cnt

    for v in range(sol[k-1]+1, n+1):
        sol[k] = v
        if k == m:
            cnt += 1
            print(str(cnt).rjust(3) + ". ", end="")
            print(*sol[1:], sep=",")
        else:
            bkt(k+1)

n = int(input("n = "))
m = int(input("m = "))
# contor pentru soluțiile generate
cnt = 0
# o soluție va avea lungimea m
sol = [0] * (m+1)
print("Toate submulțimile cu ", m, "elemente ale unei mulțimi cu ",
      n, "elemente")
bkt(1)
```

Pentru $k=1$, variabila v din ciclul `for` va fi inițializată cu valoarea $s[0]+1$, respectiv chiar cu valoarea corectă 1, deoarece $s[0]$ are valoarea 0.

La fel ca și în cazul generării permutărilor, putem aproxima complexitatea minimă a acestui algoritm prin numărul soluțiilor pe care le va afișa, deci prin $\mathcal{O}(C_n^m)$.

Problemele de generare prezentate pot fi utilizate și pentru a genera permutările/aranjamentele/combinările unei colecții oarecare, considerând elementele mulțimii $\{1, 2, \dots, n\}$ ca fiind pozițiile elementelor colecției respective!

De exemplu, pentru a genera toate anagramele distincte ale unui cuvânt, vom genera toate permutările de lungime egală cu lungimea cuvântului, pentru fiecare permutare vom reconstitui cuvântul asociat și îl vom salva într-o mulțime (i.e., colecție de tip set):

```
def bkt(k):
    global s, n, cuv, cuv_dist

    for v in range(1, n+1):
        s[k] = v
        if s[k] not in s[1:k]:
            if k == n:
                aux = "".join([cuv[s[i]-1] for i in range(1, n+1)])
                cuv_dist.add(aux)
            else:
                bkt(k+1)

cuv = input("Cuvantul: ")
n = len(cuv)
cuv_dist = set()
s = [0] * (n+1)
bkt(1)
print("Anagramele distincte ale cuvântului " + cuv + ": ")
print(*cuv_dist, sep="\n")
```

4. Descompunerea unui număr natural ca sumă de numere naturale nenule

Această problemă apare destul de des în practică, în diverse forme: împărțirea unui produs dintr-un depozit între mai multe magazine de desfacere, distribuirea unui sume de bani (un buget) între mai multe firme sau persoane fizice, partiționarea unui teren între mai mulți cumpărători etc.

De exemplu, numărul natural $n=4$ poate fi descompus ca sumă de numere naturale nenule, astfel: $1+1+1+1$, $1+1+2$, $1+2+1$, $1+3$, $2+1+1$, $2+2$, $3+1$ și 4 . Restricția ca termenii sumei să fie numere naturale nenule este esențială, altfel problema ar avea o infinitate de soluții!

În cazul acestei probleme, observațiile de la forma generală a unui algoritm de tip Backtracking pot fi particularizate astfel :

- $s[k]$ reprezintă un termen al sumei, deci $\min_k=1$ și $\max_k=n-k+1$ (în momentul în care componenta curentă este $s[k]$, celelalte $k-1$ componente anterioare $s[1], \dots, s[k-1]$ au, fiecare, cel puțin valoarea 1, deci $s[k]$ nu poate să depășească valoarea $n-(k-1)=n-k+1$ deoarece atunci suma $s[1]+\dots+s[k]$ ar fi strict mai mare decât n);
- soluțiile problemei nu mai au toate lungimi egale, ci ele variază de la 1 la n ;
- $s[1], \dots, s[k]$ este soluție parțială dacă $s[1]+\dots+s[k] \leq n$. Se observă faptul că această condiție este dedusă din predicatul P (care impune $s[1]+\dots+s[k]=n$) și este neapărat necesară (dacă $s[1]+\dots+s[k] > n$ atunci, indiferent de ce numere naturale nenule $s[k+1], s[k+2], \dots$ am mai adăuga (inclusiv niciunul!), nu vom mai putea obține $s[1]+\dots+s[k]=n$), fără însă a fi și suficientă (dacă $s[1]+\dots+s[k] \leq n$ nu înseamnă obligatoriu că $s[1]+\dots+s[k]=n$);
- $s[1], \dots, s[k]$ este soluție dacă $s[1]+\dots+s[k]=n$.

Aplicând observațiile anterioare în forma generală a algoritmului de Backtracking, obținem următoarea implementare a acestui algoritm în limbajul Python:

```
def bkt(k):
    global sol, n

    for v in range(1, n-k+2):
        sol[k] = v
        scrt = sum(sol[:k+1])
        if scrt <= n:
            if scrt == n:
                print(*sol[1: k+1], sep="+")
            else:
                bkt(k+1)

n = int(input("n = "))
sol = [0]*(n+1)
bkt(1)
```

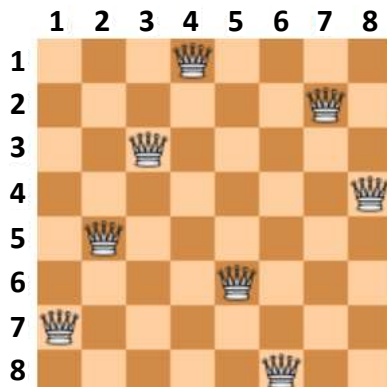
În practică, această problemă apare în multe variante, câteva dintre ele fiind următoarele (în toate exemplele am considerat $n=4$):

- *descompuneri distincte* (care nu conțin aceiași termeni, dar în altă ordine): 1+1+1+1, 1+1+2, 1+3, 2+2 și 4;
- *descompuneri cu termeni distincți* (care nu conțin termeni egali): 1+3, 3+1 și 4;
- *descompuneri distincte cu termeni distincți*: 1+3 și 4;
- *descompuneri ale căror lungimi verifică anumite condiții* (de exemplu, descompuneri de lungime egală cu 3: 1+1+2, 1+2+1 și 2+1+1);
- *descompuneri ale căror termeni verifică anumite condiții* (de exemplu, descompuneri cu toți termenii numere pare: 2+2 și 4);
- *descompuneri care verifică simultan mai multe dintre condițiile de mai sus.*

Complexitatea acestui algoritm poate fi estimată doar folosind cunoștințe avansate de teoria numerelor pentru a aproxima numărul soluțiilor pe care le va afișa ([https://en.wikipedia.org/wiki/Partition_function_\(number_theory\)](https://en.wikipedia.org/wiki/Partition_function_(number_theory))). Astfel, se poate demonstra faptul că acest algoritm are o complexitate de tip exponențial (de exemplu, numărul $n = 1000$ are aproximativ $24061467864032622473692149727991 \approx 2.4 \times 10^{31}$ descompuneri distincte!).

5. Problema celor n regine

Fiind dată o tablă de șah de dimensiune $n \times n$, problema cere să se determine toate modurile în care pot fi plasate n regine pe tablă astfel încât oricare două să nu se atace între ele. Două regine se atacă pe tabla de șah dacă se află pe aceeași linie, coloană sau diagonală. De exemplu, pentru $n = 8$, o posibilă soluție dintre cele 92 existente, este următoarea (sursa: https://en.wikipedia.org/wiki/Eight_queens_puzzle):



Problema a fost formulată de către creatorul de probleme șahistice Max Bezzel în 1848 pentru $n = 8$. În 1850 Franz Nauck a publicat primele soluții ale problemei și a generalizat-o pentru orice număr natural $n \geq 4$ (pentru $n \leq 3$ problema nu are soluții), ulterior ea fiind analizată de mai mulți matematicieni (e.g., C. F. Gauss) și informaticieni (e.g., E.W. Dijkstra) celebri. Problema poate fi rezolvată prin mai multe metode, astfel:

- considerând reginele numerotate de la 1 la n^2 , generăm, pe rând, toate cele $C_{n^2}^n$ submulțimi formate n regine și apoi le testăm (de exemplu, pentru $n = 8$ vom genera și testa $C_{64}^8 = 4426165368$ submulțimi). Evident, această metodă de tip forță-brută este foarte ineficientă, deoarece vom genera și testa inutil foarte multe submulțimi care sigur nu pot fi soluții (de exemplu, toate submulțimile care cuprind cel puțin două regine pe aceeași linie, coloană sau diagonală);
- observând faptul că pe o linie se poate poziționa exact o regină, vom genera, pe rând, toate cele n^n tupluri conținând coloanele pe care se află reginele de pe fiecare linie și le vom testa (de exemplu, pentru $n = 8$ vom genera și testa $8^8 = 16777216$ tupluri). Deși această metodă, tot de tip forță-brută, este de aproximativ 260 de ori mai rapidă decât precedenta, tot va genera și testa inutil multe tupluri care nu pot fi soluții (de exemplu, toate tuplurile care conțin cel puțin două valori egale, deoarece acest lucru înseamnă faptul că mai mult de două regine se află pe aceeași coloană, deci se atacă între ele);
- observând faptul că pe o linie și o coloană se poate poziționa exact o regină, vom genera, pe rând, utilizând metoda Backtracking, toate cele $n!$ permutări cu n elemente, testând la fiecare pas și condiția ca reginele să nu se atace pe diagonală (de exemplu, pentru $n = 8$ vom genera și testa $8! = 40320$ permutări). Evident, această metodă este mult mai eficientă decât primele două, fiind de aproximativ 420 de ori mai rapidă decât a doua metodă și de peste 110000 de ori decât prima!

În continuare, vom detalia puțin cea de-a treia variantă de rezolvare prezentată mai sus, bazată pe metoda Backtracking. Revenind la observațiile generale de la metoda Backtracking, acestea se vor particulariza, astfel:

- $s[k]$ reprezintă coloană pe care este poziționată regina de pe linia k . De exemplu, soluției prezentate în figura de mai sus îi corespunde tuplul $s = (4, 7, 3, 8, 2, 5, 1, 6)$;
- deoarece pe o linie k regina poate fi poziționată pe orice coloană $s[k]$ cuprinsă între 1 și n , obținem $\min_k = 1$ și $\max_k = n$;
- $s[1], \dots, s[k-1], s[k]$ este soluție parțială dacă regina curentă $R_k(k, s[k])$, adică regina aflată pe linia k și coloana $s[k]$, nu se atacă pe coloană sau diagonală cu nicio regină anterior poziționată pe o linie i și o coloană $s[i]$, pentru orice $i \in \{1, \dots, k-1\}$.

Condiția referitoare la coloană se deduce imediat, respectiv trebuie ca $s[k] \neq s[i]$ pentru orice $i \in \{1, 2, \dots, k-1\}$. Condiția referitoare la diagonală se poate deduce, de exemplu, astfel: regina $R_k(k, s[k])$ se atacă pe diagonală cu o altă regină $R_i(i, s[i])$ dacă și numai dacă dreapta $R_k R_i$ este paralelă cu una dintre cele două diagonale ale tablei de șah. Două drepte sunt paralele dacă și numai dacă au pantele egale, iar cele două diagonale au pantele egale cu $\tan 45^\circ = 1$ și $\tan 135^\circ = -1$, deci panta dreptei $R_k R_i$ trebuie să fie diferită de ± 1 , deci $m_{R_k R_i} = \frac{s[k]-s[i]}{k-i} \neq \pm 1$. Aplicând funcția modul, obținem $\left| \frac{s[k]-s[i]}{k-i} \right| \neq 1$ sau, echivalent, $|s[k] - s[i]| \neq |k - i|$ pentru orice $i \in \{1, 2, \dots, k-1\}$;

- pentru a testa că $s[1], \dots, s[k]$ este soluție vom ține cont de faptul că $s[1], \dots, s[k]$ este soluție parțială, deci nu vom retesta condițiile de continuare, ci doar condiția suplimentară $k=n$.

Practic, se observă faptul că problema celor n regine se reduce la generarea permutărilor de ordin n care verifică și condiția referitoare la diagonale, deci putem utiliza direct algoritmul de generare a permutărilor în care modificăm doar condiția de continuare, astfel:

```
if s[k] not in s[:k] and \
    True not in [abs(k - i) == abs(s[k] - s[i]) for i in range(1, k)]:
```

Complexitatea algoritmului de tip Backtracking de mai sus poate fi aproximată prin $\mathcal{O}(n!)$, fiind o variantă puțin modificată a algoritmului de generare a permutărilor de ordin n .

6. Problema plății unei sume folosind monede cu valori date

Considerând faptul că avem la dispoziție n monede cu valorile v_1, v_2, \dots, v_n pe care putem să le folosim pentru a plăti o sumă P , trebuie să determinăm toate modalitățile în care putem realiza acest lucru (vom presupune faptul că avem la dispoziție un număr suficient de monede de fiecare tip).

Exemplu: Dacă avem la dispoziție $n = 3$ tipuri de monede cu valorile $v = (2\$, 3\$, 5\$)$, atunci putem să plătim suma $P = 12\$$ în următoarele 5 moduri: $4 \times 3\$, 1 \times 2\$ + 2 \times 5\$, 2 \times 2\$ + 1 \times 3\$ + 1 \times 5\$, 3 \times 2\$ + 2 \times 3\$$ și $6 \times 2\$$.

Pentru a rezolva această problemă vom particulariza algoritmul generic de Backtracking, astfel:

- $s[k]$ reprezintă numărul de monede cu valoarea $v[k]$ utilizate pentru plata sumei P , deci obținem $\min_k = 0$ și $\max_k = P/v[k]$;
- $s[1], \dots, s[k-1], s[k]$ este soluție parțială dacă suma curentă este cel mult egală cu suma de plată P , adică $s[1]*v[1] + \dots + s[k]*v[k] \leq P$;
- $s[1], \dots, s[k]$ este soluție dacă suma curentă este egală cu suma de plată P , adică $s[1]*v[1] + \dots + s[k]*v[k] == P$ (se observă faptul că soluțiile au lungimi variabile, cuprinse între 1 și n);

- deoarece problema nu are întotdeauna soluție (de exemplu, dacă toate monedele date au valori pare și suma de plată este impară), vom adăuga o variabilă `nrs` care să contorizeze numărul soluțiilor găsite, iar după terminarea algoritmului vom verifica dacă problema a avut cel puțin o soluție sau nu.

Observație: Deoarece $\min_k=0$, înseamnă că tabloul `s` va conține, pe rând, valorile $(0), (0,0), \dots, \underbrace{(0,0,\dots,0)}_{\text{de } n \text{ ori}}$, pentru că, la fiecare pas, va fi verificată condiția de continuare de mai sus $(0*v[1]+\dots+0*v[k]=0 \leq P)$. Din acest motiv, trebuie să limităm extinderea tabloului la mai mult de `n` elemente pentru a evita erorile accesare a memoriei, deci vom efectua apelul recursiv `bkt(k+1)` doar în cazul în care $k < n$!

În continuare prezentăm implementarea algoritmului în limbajul Python:

```
def bkt(k):
    global s, P, v, n

    # s[k] = numarul de monede cu valoarea v[k] utilizate
    for m in range(0, P // v[k] + 1):
        s[k] = m
        scrt = sum([s[i] * v[i] for i in range(k+1)])
        if scrt <= P:
            if scrt == P and k == n:
                for i in range(1, n+1):
                    if s[i] != 0:
                        print(s[i], "x", v[i], "$ + ", end="")
                print()
            else:
                if k < len(v[1:]):
                    bkt(k+1)

P = int(input("Suma de plată: "))
aux = [int(x) for x in input("Valorile monedelor: ").split()]
v = [0]
v.extend(aux)
n = len(v[1:])
s = [0]*(len(v))
print("Toate modalitățile de plată:")
bkt(1)
```

Observație: Deoarece $\min_k=0$, înseamnă că tabloul `s` va conține, pe rând, valorile $(0), (0,0), \dots, \underbrace{(0,0,\dots,0)}_{\text{de } n \text{ ori}}$, pentru că, la fiecare pas, va fi verificată condiția de continuare de mai sus $(0*v[1]+\dots+0*v[k]=0 \leq P)$. Din acest motiv, trebuie să limităm extinderea tabloului la mai mult de `n` elemente pentru a evita erorile accesare a memoriei, deci vom efectua apelul recursiv `bkt(k+1)` doar în cazul în care $k < n$!

Complexitatea acestui algoritmului poate fi aproximată prin numărul maxim de tupluri care pot fi generate și testate, respectiv $\frac{P}{v_1} \cdot \frac{P}{v_2} \cdot \dots \cdot \frac{P}{v_n}$. În cazul unor date de intrare "reale", putem presupune faptul ca valorile monedelor v_1, v_2, \dots, v_n sunt cel mult egale cu suma P (o monedă cu o valoare strict mai mare decât P este inutilă, deci ar putea fi eliminată din datele de intrare), astfel încât fiecare raport $\frac{P}{v_k}$ va fi mai mare sau egal decât 1. De fapt, în realitate, valorile monedelor v_1, v_2, \dots, v_n sunt mult mai mici decât suma de plată P , deci fiecare raport $\frac{P}{v_k}$ va fi, în general, mai mare sau egal decât 2, deci complexitatea algoritmului poate fi aproximată prin $\mathcal{O}(2^n)$.

Observație: Metoda Backtracking poate fi modificată astfel încât să fie utilizată și pentru rezolvarea altor tipuri de probleme, în afara celor de generare a tuturor soluțiilor, astfel:

- *pentru probleme de numărare:* se generează toate soluțiile posibile și se înlocuiește secțiunea pentru afișarea unei soluții cu o simplă incrementare a unui contor, iar după terminarea algoritmului se afișează valoarea contorului. Atenție, de multe ori, problemele de numărare se pot rezolva mult mai eficient, fie utilizând o formulă matematică (de exemplu, numărul permutărilor p de ordin n fără puncte fixe, i.e. $p[k] \neq k, \forall k \in \{1, \dots, n\}$, poate fi calculat folosind o formulă: <https://en.wikipedia.org/wiki/Derangement>), fie utilizând alte tehnici de programare (de exemplu, numărul modalităților de plată a unei sume folosind monede cu valori date se poate calcula cu un algoritm care utilizează metoda programării dinamice și are complexitatea $\mathcal{O}(n^2)$: <https://www.geeksforgeeks.org/coin-change-dp-7/>).
- *pentru probleme de decizie:* într-o problemă de decizie ne interesează doar faptul că o problemă are soluție sau nu (de exemplu, problema plății unei sume folosind monede cu valori date poate fi transformată într-o problemă de decizie, astfel: "Să se verifice dacă o sumă de bani P poate fi plătită utilizând monede cu valorile v_1, v_2, \dots, v_n ."), deci fie vom opri forțat algoritmul Backtracking în momentul în care găsim prima soluție, fie acesta se va termina normal în cazul în care problema nu are soluție. Și în acest caz, de obicei, există algoritmi mai eficienți, care utilizează alte tehnici de programare (de exemplu, pentru a verifica dacă o sumă de bani poate fi plătită folosind anumite monede se poate utiliza algoritmul menționat anterior, având complexitatea $\mathcal{O}(n^2)$).
- *pentru probleme de optimizare:* într-o problemă de optimizare trebuie să găsim, de obicei, o singură soluție care, în plus, minimizează sau maximizează o anumită expresie matematică (de exemplu, se poate cere determinarea unei modalități de plată a unei sume folosind un număr minim de monede cu valori date). În acest caz, vom genera toate soluțiile problemei și vom reține, într-o structură de date auxiliară, o soluție optimă. În cazul acestor probleme există, de obicei, algoritmi mai eficienți, care utilizează alte tehnici de programare, cum ar fi metoda Greedy sau metoda programării dinamice. De exemplu, pentru a determina o modalitate de plată a unei sume folosind un număr minim de monede, există un algoritm cu complexitatea $\mathcal{O}(n \cdot P)$ bazat pe metoda programării dinamice: <https://www.geeksforgeeks.org/find-minimum-number-of-coins-that-make-a-change/>.

În încheiere, precizăm faptul că, în soluțiile problemelor pe care le-am prezentat, am dorit să accentuăm aspecte generale prin care algoritmul generic de Backtracking poate

fi particularizat pentru a rezolva diverse tipuri de probleme, neînsistând asupra unor modalități particulare de optimizare a lor, cum ar fi găsirea unui interval de valori cât mai mic pentru o componentă a soluției (i.e., diferența $\max_k - \min_k$ să fie minimă), utilizarea unor structuri de date auxiliare pentru a marca valorile deja utilizate (de exemplu, în algoritmul de generare a permutărilor se poate utiliza un vector de marcaje pentru a verifica direct dacă o anumită valoare a fost deja utilizată) sau actualizarea dinamică a unor valori necesare în verificarea condițiilor de continuare (de exemplu, în algoritmul pentru descompunerea unui număr natural ca sumă de numere naturale nenule se poate actualiza dinamic suma curentă, în momentele în care se adaugă la o soluție parțială o nouă componentă, se modifică valoarea componentei curente sau se renunță la componenta curentă). Din punctul nostru de vedere, aceste optimizări complică destul de mult codul sursă și nu sunt foarte utile, deoarece, oricum, algoritmii de tip Backtracking au un timp de executare acceptabil doar pentru dimensiuni mici ale datelor de intrare.