CURS 10
TEHNICA DE PROGRAMARE "BACKTRACKING"

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

Această problemă apare destul de des în practică, în diverse forme: împărţirea 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, partiţionarea unui teren între mai mulţi 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` şi `4`. Restricţia ca termenii sumei să fie numere naturale nenule este esenţială, altfel problema ar avea o infinitate de soluţii!

În cazul acestei probleme, observaţiile 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` şi max_k=`n-k+1` (în momentul în care componenta curentă este `s[k]`, celelalte `k-1` componente anterioare `s[1],…,s[k-1]` au, fiecare, cel puţin valoarea `1`, deci `s[k]` nu poate să depăşească valoarea `n-(k-1)=n-k+1` deoarece atunci suma `s[1]+…+s[k]` ar fi strict mai mare decât `n`);
- `succ(v)=v+1`, deoarece valorile posibile pentru `s[k]` sunt numere naturale consecutive cuprinse între `1` şi `n-k+1`;
- soluţiile problemei nu mai au toate lungimi egale, ci ele variază de la `1` la `n`;
- `s[1],…,s[k]` este soluţie parţială dacă `s[1]+…+s[k]<=n`. Se observă faptul că această condiţie este dedusă din predicatul *P* (care impune `s[1]+…+s[k]=n`) şi este neapărat necesară (dacă `s[1]+…+s[k]>n` atunci, indiferent de ce numere naturale nenule `s[k+1],s[k+2],…` am mai adăuga (inclusiv niciunul!), nu vom mai putea obţine `s[1]+…+s[k]=n`), fără însă a fi şi suficientă (dacă `s[1]+…+s[k]<=n` nu înseamnă obligatoriu `s[1]+…+s[k]=n`);
- `s[1],…,s[k]` este soluţie dacă `s[1]+…+s[k]=n`;
- deoarece trebuie să calculăm suma `s[1]+…+s[k]` şi când testăm condiţia de continuare şi când testăm condiţia de soluţie, vom utiliza o funcţie `int suma(int k)` care să o calculeze şi vom renunţa la funcţia `int solp(int k)`.

Aplicând observaţiile anterioare în forma generală a algoritmului de Backtracking, obţinem următoarea implementare a acestui algoritm în limbajul C:

```
#include<stdio.h>

int s[101], n;

int suma(int k)
{
    int i, scrt;

    scrt = 0;
    for(i = 1; i <= k; i++)
```

```
        scrt = scrt + s[i];
    return scrt;
}


void bkt(int k)
{
    int i, v, scrt;

    for(v = 1; v <= n-k+1; v++)
    {
        s[k] = v;
        scrt = suma(k);
        if(scrt <= n)
            if(scrt == n)
            {
                for(i = 1; i <= k; i++)
                    printf("%d ", s[i]);
                printf("\n");
            }
            else
                bkt(k+1);
    }
}

int main()
{
    printf("n = ");
    scanf("%d", &n);

    bkt(1);

    return 0;
}
```

Instrucţiunea **if**(scrt <= n) este neapărat necesară sau o putem elimina? Argumentaţi!

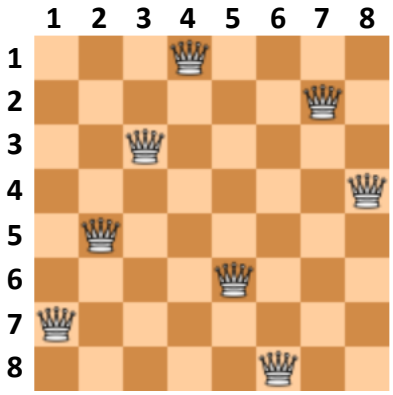
În practică, această problema apare în multe variante, câteva dintre ele fiind următoarele (în toate exemplele am considerat n=4):

- *descompuneri distincte* (care nu conţin aceiaşi termeni, dar în altă ordine): 1+1+1+1, 1+1+2, 1+3, 2+2 şi 4;
- *descompuneri cu termeni distincţi* (care nu conţin termeni egali): 1+3, 3+1 şi 4;
- *descompuneri distincte cu termeni distincţi*: 1+3 şi 4;
- *descompuneri ale căror lungimi verifică anumite condiţii* (de exemplu, descompuneri de lungime egală cu 3: 1+1+2, 1+2+1 şi 2+1+1);
- *descompuneri ale căror termeni verifică anumite condiţii* (de exemplu, descompuneri cu termeni cel mult egali cu 2: 1+1+1+1, 1+1+2, 1+2+1, 2+1+1, şi 2+2);
- *descompuneri care verifică simultan mai multe dintre condiţiile de mai sus*.

Complexitatea acestui algoritm poate fi estimată doar folosind cunoştinţe avansate de teoria numerelor pentru a aproxima numărul soluţiilor pe care le va afişa (https://en.wikipedia.org/wiki/Partition_function_(number_theory)). Astfel, se poate demonstra faptul că acest algoritm are o complexitate de tip exponenţial (de exemplu, numărul $n = 1000$ are aproximativ $2406146786403262247369214972 7991 \approx 2.4 \times 10^{31}$ descompuneri distincte!).

2. Problema celor n regine

Fiind dată o tablă de şah 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 şah dacă se află pe aceeaşi linie, coloană sau diagonală. De exemplu, pentru $n = 8$, o posibilă soluţie dintre cele 92 existente, este următoarea (sursa: https://en.wikipedia.org/wiki/Eight_queens_puzzle):



Problema a fost formulată de către creatorul de probleme şahistice Max Bezzel în 1848 pentru $n = 8$. În 1850 Franz Nauck a publicat primele soluţii ale problemei şi a generalizat-o pentru orice număr natural $n \geq 4$ (pentru $n \leq 3$ problema nu are soluţii), ulterior ea fiind analizată de mai mulţi matematicieni (e.g., C. F. Gauss) şi 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 $C_{n^2}^n$ submulţimi formate n regine şi apoi le testăm (de exemplu, pentru $n = 8$ vom genera şi testa $C_{64}^8 = 4426165368$ submulţimi). Evident, această metodă de tip forţă-brută este foarte ineficientă, deoarece vom genera şi testa inutil foarte multe submulţimi care sigur nu pot fi soluţii (de exemplu, toate submulţimile care cuprind cel puţin două regine pe aceeaşi linie, coloană sau diagonală);
- observând faptul că pe o linie se poate poziţiona exact o regină, vom genera, pe rând, toate cele n^n tupluri conţinând coloanele pe care se află reginele de pe fiecare linie şi le vom testa (de exemplu, pentru $n = 8$ vom genera şi testa $8^8 = 16777216$ tupluri). Deşi această metodă, tot de tip forţă-brută, este de aproximativ 260 de ori mai rapidă decât precedenta, tot va genera şi testa inutil multe tupluri care nu pot fi soluţii (de exemplu, toate tuplurile care conţin cel puţin două valori egale, deoarece acest lucru înseamnă faptul că mai mult de două regine se află pe aceeaşi coloană, deci se atacă între ele);
- observând faptul că pe o linie şi o coloană se poate poziţiona 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 $\texttt{min}_k=1$ și $\texttt{max}_k=n$;
- `succ(v)=v+1`, deoarece valorile posibile pentru `s[k]` sunt numere naturale consecutive cuprinse între `1` și `n`;
- `s[1],…,s[k-1],s[k]` este soluție parțială dacă regina curentă $\texttt{R}_k\texttt{(k,s[k])}$, 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_kR_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 $\operatorname{tg} 45° = 1$ și $\operatorname{tg} 135° = -1$, deci panta dreptei R_kR_i trebuie să fie diferită de ± 1, deci $m_{R_kR_i} = \frac{s[k]-s[i]}{k-i} \neq \pm 1$. 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],…,s[k]` este soluție vom ține cont de faptul că `s[1],…,s[k]` este soluție parțială, deci nu vom retesta condițiile de continuare, ci doar condiția suplimentară `k=n`.

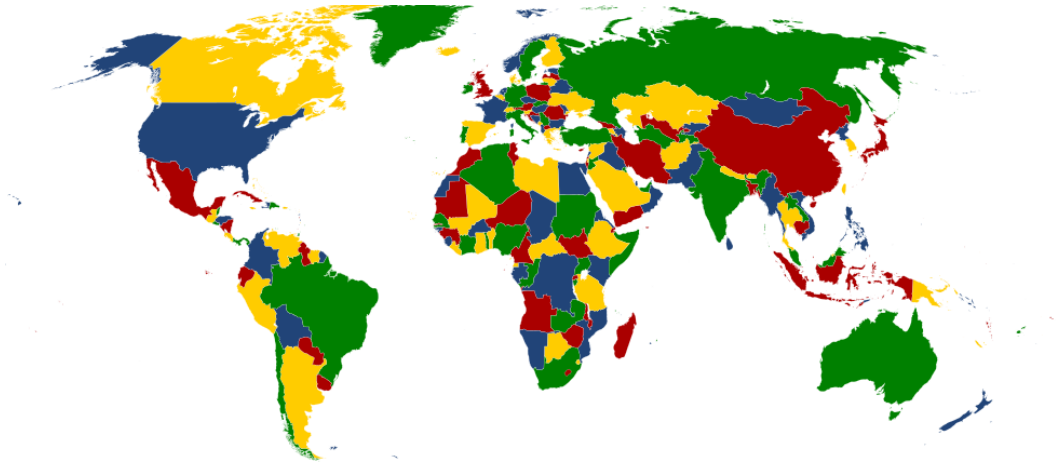
Practic, se observă faptul că problema celor n regine se reduce la generarea permutărilor de ordin n care verifică și condiția referitoare la diagonale, deci putem utiliza direct algoritmul de generare a permutărilor în care modificăm doar funcția `solp` de testare a condițiilor de continuare, astfel:

```
int solp(int k)
{
    int i;
    for(i = 1; i < k; i++)
        if(s[k] == s[i] || abs(s[k] - s[i]) == abs(k - i))
            return 0;
    return 1;
}
```

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.

3. Problema colorării hărţilor

Fiind dată o hartă cu n ţări, problema cere să se determine toate modurile în care aceasta poate fi colorată folosind cel mult 4 culori, astfel încât oricare două ţări vecine să fie colorate diferit. De exemplu, o colorare de acest tip a mapamondului este dată în figura de mai jos: (sursa: https://commons.wikimedia.org/wiki/File:Four_color_world_map.svg):



Pentru a putea fi colorată folosind cel mult 4 culori, o hartă trebuie să îndeplinească următoarele două condiţii (https://en.wikipedia.org/wiki/Four_color_theorem):

- graniţa dintre două ţări nu se reduce la un singur punct;
- dacă teritoriul unei ţări nu este continuu (de exemplu, statul Alaska aparţine Statelor Unite ale Americii, dar este complet separat de restul teritoriului său), atunci părţile sale nu trebuie să fie colorate folosind aceeaşi culoare.

Problema are o istorie foarte interesantă, fiind formulată în 1852 de către matematicianul şi botanistul sud-african Francis Guthrie sub forma unei conjecturi (i.e., "*Orice hartă poate fi colorată folosind cel mult 4 culori astfel încât oricare două ţări vecine să fie colorate diferit.*") pe care fratele său, Frederick Guthrie, i-a prezentat-o marelui matematician britanic Augustus De Morgan. Ulterior, a fost reformulată folosind noţiuni din teoria grafurilor, fiind studiată de mai mulţi matematicieni de prestigiu (e.g., A. De Morgan, P.G. Tait, A. Kempe etc.) , dar fără a reuşi să o rezolve (i.e., fie arătând că este falsă printr-un contraexemplu, fie arătând că este adevărată printr-o demonstraţie). Prima demonstraţie a corectitudinii sale a fost realizată abia în anul 1976 de către matematicienii americani K. Appel şi W. Haken de la Universitatea din Illinois, combinând noţiuni de algebră cu rezultate obţinute cu ajutorul unui computer. Astfel, pentru prima dată în istoria matematicii, o demonstraţie era efectuată, chiar şi numai parţial, cu ajutorul unui computer, astfel demonstraţia fiind foarte greu acceptată de comunitatea matematică! Ulterior, au fost găsite unele erori în demonstraţia sa, dar acestea au fost corectate de către K. Appel şi W. Haken, iar în 1989 aceştia au publicat varianta finală a demonstraţiei lor. În 1996, matematicienii americani N. Robertson, D.P. Sanders, P. Seymour şi R. Thomas au simplificat demonstraţia lui K. Appel şi W. Haken, realizând şi o optimizare a algoritmului utilizat. În 2005, B. Werner şi G. Gonthier au reuşit să demonstreze teorema utilizând Coq - un instrument software pentru demonstrarea interactivă a teoremelor (https://en.wikipedia.org/wiki/Coq). Astfel, deşi au trecut mai mult de 150 de ani de la formularea problemei colorării hărţilor, încă nu a fost realizată nicio demonstraţie a sa care să nu necesite utilizarea unui computer!

În cadrul demonstrației lor, K. Appel și W. Haken au utilizat un algoritm doar pentru a testa dacă o anumită hartă poate fi colorată conform restricțiilor, deci nu pentru a genera toate modurile în care poate fi colorată harta respectivă. Cu toate acestea, computerul utilizat a avut nevoie de peste 1000 de ore pentru a testa în jur de 1800 de hărți!

Pentru a rezolva problema generării tuturor modurilor în care poate fi colorată o hartă dată respectând restricțiile indicate vom utiliza metoda Bactracking particularizată, astfel:

- considerând cele n țări ca fiind numerotate cu numerele naturale de la 1 la n, relațiile de vecinătate dintre țări vor fi memorate utilizând o matrice de adiacență A de dimensiune $n \times n$, definită astfel:

$$a[i][j] = \begin{cases} 1, \text{dacă țara } i \text{ este vecină cu țara } j \\ 0, \text{dacă țara } i \text{ nu este vecină cu țara } j \end{cases}$$

  Se observă ușor faptul că matricea A are toate elementele de pe diagonala principală egale cu 0 (deoarece o țară nu poate fi vecină cu ea însăși) și este simetrică față de diagonala principală (i.e., $a[i][j] = a[j][i]$, deoarece dacă țara i este vecină cu țara j, atunci și țara j este vecină cu țara i).
- `s[k]` reprezintă culoarea atribuită țării `k`, deci obținem $\texttt{min}_k$=1 și $\texttt{max}_k$=4;
- `succ(v)=v+1`, deoarece valorile posibile pentru `s[k]` sunt numere naturale consecutive cuprinse între `1` și `n`;
- `s[1],…,s[k-1],s[k]` este soluție parțială dacă nu există nicio țară `i` colorată anterior, deci `1 ≤ i < k`, care să fie vecină cu țara `k` și colorată la fel, adică folosind tot culoarea `s[k]`;
- pentru a testa că `s[1],…,s[k]` este soluție vom ține cont de faptul că `s[1],…,s[k]` este soluție parțială, deci nu vom retesta condițiile de continuare, ci doar condiția suplimentară `k=n`.

Complexitatea acestui algoritm de tip Backtracking poate fi aproximată prin $\mathcal{O}(4^n)$, deoarece fiecare dintre cele n țări poate fi colorată folosind cel mult 4 culori, deci se vor testa cel mult $\underbrace{4 \cdot 4 \cdot \ldots \cdot 4}_{n \text{ ori}} = 4^n$ tupluri reprezentând posibile colorări.

4. Determinarea tuturor numerelor având cifre distincte și suma cifrelor dată

Problema a fost dată la primul examen de Bacalaureat în care elevii de la profilul Informatică au susținut obligatoriu o probă la disciplina Informatică. Problema cerea ca pentru un număr natural c citit de la tastatură să se afișeze toate numerele formate din cifre distincte și suma cifrelor egală cu c. De exemplu, pentru $c = 3$, trebuie să fie afișate numerele: 102, 12, 120, 201, 21, 210, 3 și 30 (nu neapărat în această ordine).

Analizând enunțul problemei, observăm faptul că orice număr care este soluție a problemei are cel mult 10 cifre, deoarece acestea trebuie să fie distincte, și problema are soluție doar în cazul în care $c \in \{0,1, \ldots ,45\}$, deoarece cea mai mare sumă care se poate obține din cifre distincte este egală cu $1 + 2 + \cdots + 9 = 45$.

Pentru a rezolva problema vom utiliza metoda Bactracking, astfel:

- `s[k]` reprezintă cifra aflată pe poziția `k` într-un număr (considerăm cifrele unui număr ca fiind numerotate de la stânga spre dreapta), deci obținem $min_k=1$ pentru `k=1` (prima cifră a unui număr nu poate fi 0) sau $min_k=0$ pentru $k\geq 2$, respectiv $max_k=9$;
- `succ(v)=v+1`;
- `s[1],…,s[k-1],s[k]` este soluție parțială dacă cifra curentă `s[k]` nu a mai fost utilizată anterior, adică $s[k]\neq s[i]$ pentru orice $i\in\{1,2,\ldots,k-1\}$, și $s[1]+\ldots+s[k]\leq c$;
- pentru a testa dacă `s[1],…,s[k]` este soluție vom ține cont de faptul că cifrele `s[1],…,s[k]` sunt distincte (din condițiile de continuare), deci vom verifica doar condiția suplimentară $s[1]+\ldots+s[k]==c$.

Implementând algoritmul Backtracking corespunzător observațiilor de mai sus, nu vom obține toate soluțiile, ci doar o parte dintre ele! De exemplu, pentru $c = 3$, **nu** vom obține numerele scrise îngroșat: 102, 12, **120**, 201, 21, **210**, 3 și **30**. Explicația acestui fapt necesită o înțelegere aprofundată a metodei Backtracking: numerele scrise îngroșat sunt soluții care se obțin din soluțiile care nu conțin cifra 0 (de exemplu, numărul **120** se obține din numărul 12, care nu conține cifra 0, prin adăugarea unui 0 la sfârșitul său)! În forma sa generală, algoritmul Backtracking **nu** va furniza niciodată numerele scrise îngroșat, deoarece după găsirea unei soluții a problemei, algoritmul **nu** va încerca niciodată să adauge încă un element (o cifră, în acest caz) la ea! Din acest motiv, o soluție completă care nu modifică foarte mult algoritmul general Backtracking se poate obține astfel: în momentul afișării unei soluții, verificăm dacă ea conține deja o cifră egală cu 0, iar în caz negativ o afișăm încă o dată și-i adăugăm un 0 la sfârșit.

Programul scris în limbajul C care implementează rezolvarea completă a acestei probleme este următorul:

```
#include<stdio.h>
int s[11], c;

//funcție care verifică dacă cifra s[k] a fost utilizată anterior
int distincte(int k)
{
    int i;

    for(i = 1; i < k; i++)
        if(s[k] == s[i]) return 0;
    return 1;
}

//funcție care calculează suma s[1]+...+s[k]
int suma(int k)
{
    int i, scrt;

    scrt = 0;
    for(i = 1; i <= k; i++)
        scrt = scrt + s[i];
    return scrt;
}
```

```
void bkt(int k)
{
    int i, v, scrt, z;

    //valoarea minimă a unei cifre depinde de poziția sa k
    for(v = (k == 1 ? 1 : 0); v <= 9; v++)
    {
        s[k] = v;
        scrt = suma(k);

        //verificăm condițiile de continuare
        if(scrt <= c && distincte(k) == 1)
            //verificăm condiția de soluție
            if(scrt == c)
            {
                //verificăm dacă soluția curentă conține cifra 0
                z = 0;
                for(i = 1; i <= k; i++)
                {
                    printf("%d", s[i]);
                    if(s[i] == 0) z = 1;
                }
                printf("\n");

                //dacă soluția curentă nu conține cifra 0,
                //o reafișăm și adăugăm cifra 0 la sfârșitul său
                if(z == 0)
                {
                    for(i = 1; i <= k; i++) printf("%d", s[i]);
                    printf("0\n");
                }
            }
            else
                bkt(k+1);
    }
}

int main()
{
    printf("c = ");
    scanf("%d", &c);

    //verificăm dacă problema are soluție sau nu
    if(c < 0 || c > 45)
        printf("Problema nu are solutie!");
    else
        bkt(1);

    return 0;
}
```

5. Problema plăţii unei sume folosind monede cu valori date

Considerând faptul că avem la dispoziţie 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 modalităţile în care putem realiza acest lucru (vom presupune faptul că avem la dispoziţie un număr suficient de monede de fiecare tip).

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

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

- `s[k]` reprezintă numărul de monede cu valoarea `v[k]` utilizate pentru plata sumei `P`, deci obţinem $\texttt{min}_k$`=0` şi $\texttt{max}_k$`=P/v[k]`;
- `succ(v)=v+1`, deoarece valorile posibile pentru `s[k]` sunt numere naturale consecutive;
- `s[1],…,s[k-1],s[k]` este soluţie parţială dacă suma curentă este cel mult egală cu suma de plată `P`, adică `s[1]*v[1]+…+s[k]*v[k]<=P`;
- `s[1],…,s[k]` este soluţie dacă suma curentă este egală cu suma de plată `P`, adică `s[1]*v[1]+…+s[k]*v[k]=P` (se observă faptul că soluţiile au lungimi variabile, cuprinse între `1` şi `n`);
- deoarece problema nu are întotdeauna soluţie (de exemplu, dacă toate monedele date au valori pare şi suma de plată este impară), vom adăuga o variabila `nrs` care să contorizeze numărul soluţiilor găsite, iar după terminarea algoritmului vom verifica dacă problema a avut cel puţin o soluţie sau nu.

Observaţie: Deoarece $\texttt{min}_k$`=0`, înseamnă că tabloul s va conţine, 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 condiţia de continuare de mai sus (`0*v[1]+…+0*v[k]=0<=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 C:

```
#include<stdio.h>

int s[101], n, v[101], P, nrs;

int suma(int k)
{
    int i, scrt = 0;

    for(i = 1; i <= k; i++) scrt = scrt + s[i]*v[i];

    return scrt;
}
```

```
void bkt(int k)
{
    int i, m, scrt;

    for(m = 0; m <= P/v[k]; m++)
    {
        s[k] = m;

        scrt = suma(k);
        if(scrt <= P)
            if(scrt == P)
            {
                nrs++;
                printf("%3d. %d$ = ", nrs, P);
                for(i = 1; i <= k; i++)
                    if(s[i] > 0)
                        printf("%dx%d$+", s[i], v[i]);
                printf("\b \n");
            }
            else
                if(k < n)
                    bkt(k+1);
    }
}

int main()
{
    int i;

    printf("Numarul de monede: ");
    scanf("%d", &n);

    printf("Valorile monedelor:\n");
    for(i = 1; i <= n; i++)
    {
        printf("v[%d] = ", i);
        scanf("%d", &v[i]);
    }

    printf("Suma de plata: ");
    scanf("%d", &P);

    printf("\nModalitatile de plata:\n");

    bkt(1);

    if(nrs == 0)
        printf("niciuna");

    return 0;
}
```

Observaţie: Deoarece $\min_k=0$, înseamnă că tabloul s va conţine, pe rând, valorile $(0), (0,0), \ldots, (\underbrace{0,0,\ldots,0}_{\text{de } n \text{ ori}})$, pentru că, la fiecare pas, va fi verificată condiţia de continuare

de mai sus ($0 * v[1] + \cdots + 0 * v[k] = 0 <= P$). 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 şi testate, respectiv $\frac{P}{v_1} \cdot \frac{P}{v_2} \cdot \ldots \cdot \frac{P}{v_n}$. În cazul unor date de intrare "reale", putem presupune faptul ca valorile monedelor $v_1, v_2, \ldots, v_n$ 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 $v_1, v_2, \ldots, v_n$ 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 $\mathcal{O}(2^n)$.

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

- *pentru probleme de numărare*: se generează toate soluţiile posibile şi se înlocuieşte secţiunea pentru afişarea unei soluţii cu o simplă incrementare a unui contor, iar după terminarea algoritmului se afişează valoarea contorului. Atenţie, 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. $p[k] \neq k, \forall k \in \{1, \ldots, n\}$, poate fi calculat folosind o formulă: https://en.wikipedia.org/wiki/Derangement), fie utilizând alte tehnici de programare (de exemplu, numărul modalităţilor de plată a unei sume folosind monede cu valori date se poate calcula cu un algoritm care utilizează metoda programării dinamice şi 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 soluţie sau nu (de exemplu, problema plăţii 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* $v_1, v_2, \ldots, v_n$."), deci fie vom opri forţat algoritmul Backtracking în momentul în care găsim prima soluţie, fie acesta se va termina normal în cazul în care problema nu are soluţie. Şi în acest caz, de obicei, există algoritmi mai eficienţi, 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 menţionat anterior, având complexitatea $\mathcal{O}(n^2)$).

- *pentru probleme de optimizare*: într-o problemă de optimizare trebuie să găsim, de obicei, o singură soluţie care, în plus, minimizează sau maximizează o anumită expresie matematică (de exemplu, se poate cere determinarea unei modalităţi de plată a unei sume folosind un număr minim de monede cu valori date). În acest caz, vom genera toate soluţiile problemei şi vom reţine, într-o structură de date auxiliară, o soluţie optimă. În cazul acestor probleme există, de obicei, algoritmi mai eficienţi, care utilizează alte tehnici de programare, cum ar fi metoda Greedy sau metoda programării dinamice. De exemplu, pentru a determinarea 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 soluţiile 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, neinsistând asupra unor modalităţi particulare de optimizare a lor, cum ar fi: găsirea unui interval de valori cât mai mic pentru o componentă a soluţiei (i.e., diferenţa $\texttt{max}_k\texttt{-min}_k$ 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 condiţiilor 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 soluţie parţială o nouă componentă, se modifică valoarea componentei curente sau se renunţă la componenta curentă). Din punctul nostru de vedere, aceste optimizări complică destul de mult codul sursă şi nu sunt foarte utile, deoarece, oricum, algoritmii de tip Backtracking au un timp de executare acceptabil doar pentru dimensiuni mici ale datelor de intrare.