CURS 9
TEHNICA DE PROGRAMARE "BACKTRACKING"

1. Prezentare generală

Tehnica de programare Backtracking este utilizată, de obicei, pentru determinarea tuturor soluţiilor unei probleme într-un mod progresiv, astfel încât să se evite generarea întregului spaţiu al soluţiilor problemei. Practic, soluţiile se construiesc componentă cu componentă şi se testează la fiecare pas validitatea lor (se verifică dacă sunt *soluţii parţiale*), iar în cazul în care se constată faptul că nu se poate obţine o soluţie a problemei plecând de la soluţia parţială curentă, aceasta este abandonată. Astfel, se evită o *rezolvare de tip forţă-brută* a problemei, adică generarea şi testarea tuturor soluţiilor posibile pentru a determina soluţiile problemei. Totuşi, complexitatea algoritmilor de tip Backtracking rămâne una ridicată, deoarece se va genera şi testa un procent semnificativ din întregul spaţiu al soluţiilor.

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

O rezolvare de tip forţă-brută presupune generarea tuturor posibilelor soluţii, adică a tuplurilor de forma $p = (p_1, p_2, p_3, p_4, p_5, p_6)$ cu $p_1, \dots, p_6 \in \{1,2,\dots,6\}$, şi selectarea celor care sunt permutări, adică au toate valorile diferite între ele ($p_1 \neq p_2 \neq \cdots \neq p_6$). Se observă foarte uşor faptul că se vor genera şi testa $6^6 = 46656$ tupluri, din care doar $6! = 720$ vor fi permutări, deci eficienţa acestei metode este foarte mică - în jurul unui procent de 1.5%! Eficienţa 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 soluţiilor, evitând generarea unor tupluri inutile, astfel (vom ţine cont de faptul că $p_1, \dots, p_6 \in \{1,2,\dots,6\}$):

- $p = (1)$ – este o soluţie parţială (componentele sale sunt diferite între ele, deci poate fi o permutare), dar nu este o soluţie a problemei (nu are 6 componente), astfel că trebuie să adăugăm cel puţin încă o componentă;
- $p = (1,1)$ – nu este o soluţie parţială (componentele sale sunt egale, deci nu vom obţine 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 soluţie parţială, dar nu este o soluţie a problemei;
- $\left.\begin{matrix} p = (1,2,1) \\ p = (1,2,2) \end{matrix}\right\}$ – nu sunt soluţii parţiale;
- $p = (1,2,3)$ – este o soluţie parţială, dar nu este o soluţie a problemei;
- $\left.\begin{matrix} p = (1,2,3,1) \\ p = (1,2,3,2) \\ p = (1,2,3,3) \end{matrix}\right\}$ – nu sunt soluţii parţiale;
- $p = (1,2,3,4)$ – este o soluţie parţială, dar nu este o soluţie a problemei;

- $\left.\begin{matrix} p=(1,2,3,4,1) \\ p=(1,2,3,4,2) \\ p=(1,2,3,4,3) \\ p=(1,2,3,4,4) \end{matrix}\right\}$ – nu sunt soluţii parţiale;
- $p=(1,2,3,4,5)$ – este o soluţie parţială, dar nu este o soluţie a problemei;
- $\left.\begin{matrix} p=(1,2,3,4,5,1) \\ p=(1,2,3,4,5,2) \\ p=(1,2,3,4,5,3) \\ p=(1,2,3,4,5,4) \\ p=(1,2,3,4,5,5) \end{matrix}\right\}$ – nu sunt soluţii parţiale;
- $p=(1,2,3,4,5,6)$ – este o soluţie parţială care este şi soluţie a problemei, deci o memorăm sau o prelucrăm (de exemplu, o afişăm), 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 şi vom genera următorul tuplu format doar din 5 componente;
- $p=(1,2,3,4,6)$ – este o soluţie parţială, dar nu este o soluţie a problemei;
- $\left.\begin{matrix} p=(1,2,3,4,6,1) \\ p=(1,2,3,4,6,2) \\ p=(1,2,3,4,6,3) \\ p=(1,2,3,4,6,4) \end{matrix}\right\}$ – nu sunt soluţii parţiale;
- $p=(1,2,3,4,6,5)$ – este o soluţie parţială care este şi soluţie a problemei, deci o memorăm sau o prelucrăm, după care vom genera următorul tuplu;
- $p=(1,2,3,4,6,6)$ – nu este o soluţie parţială, dar ultima componentă are valoarea maxim posibilă 6, deci o vom elimina şi 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 şi vom genera următorul tuplu format doar din 4 componente;
- $p=(1,2,3,5)$ – este o soluţie parţială, dar nu este o soluţie a problemei;
- .....
- $p=(6,5,4,3,2,1)$ – este o soluţie parţială care este şi ultima soluţie a problemei, deci o memorăm sau o prelucrăm, după care vom încerca generarea următorului tuplu. Se observă cu uşurinţă faptul că, pe rând, nu vom mai găsi niciun tuplu convenabil, deci algoritmul se va termina.

Observaţie importantă: Determinarea exactă a complexităţii unui algoritm de tip Backtracking nu este simplă. Totuşi, plecând de la observaţia că un algoritm nu poate avea o complexitate mai mică decât citirea datelor de intrare şi/sau afişarea datelor de ieşire, de obicei, putem aproxima complexitatea unui astfel de algoritm prin numărul soluţiilor pe care el le afişează. De exemplu, algoritmul pentru generarea permutărilor de ordin n are complexitatea minimă egală cu $\mathcal{O}(n!)$, deoarece vor fi afişate $n!$ permutări. O astfel de complexitate este foarte mare, depăşind-o pe cea exponenţială!

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 mulţimile nevide* $A_1, A_2, \dots, A_n$ *şi un predicat* $P: A_1 \times \dots \times A_n \to \{0,1\}$*. Să se genereze toate tuplurile de forma* $S = (s_1, s_2, \dots, s_n)$ *pentru care* $s_1 \in A_1, \dots, s_n \in A_n$ *şi* $P(s_1, s_2, \dots, s_n) = 1$.". Practic, predicatul P cuantifică o proprietate pe care trebuie să o îndeplinească componentele tuplului S (o soluţie a problemei) sub forma unei funcţii de tip boolean (0 = false şi 1 = true).

De exemplu, problema generării permutărilor poate fi formalizată în acest mod considerând $A_1 = \dots = A_n = \{1,2, \dots, n\}$ şi $P(s_1, s_2, \dots, s_n) = (s_1 \neq s_2) \wedge (s_2 \neq s_3) \wedge \dots \wedge (s_{n-1} \neq s_n)$, unde prin $\wedge$ am notat operatorul logic AND.

În continuare, vom prezenta câteva notaţii, definiţii şi observaţii:

- 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;
- vom nota cu $succ(x)$ următoarea valoare posibilă după x (în foarte multe cazuri, $succ(x) = x + 1$);
- într-un tuplu $(s_1, s_2, \dots, s_k)$, componenta s_k se numeşte *componentă curentă* (asupra sa se acţionează în momentul respectiv);
- *condiţiile de continuare* reprezintă condiţiile 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 obţinem o soluţie $S = (s_1, \dots, s_n)$ a problemei;
- *condiţiile de continuare* se deduc din predicatul P şi sunt neapărat necesare, fără a fi întotdeauna şi suficiente;
- tuplul curent $(s_1, \dots, s_k)$ este o *soluţie parţială* dacă el îndeplineşte condiţiile de continuare;
- orice *soluţie* a problemei este implicit şi soluţie parţială, dar trebuie să mai îndeplinească şi alte condiţii suplimentare.

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

```
//k reprezintă indicele componentei curent s[k] dintr-un
//tablou unidimensional s indexat de la 1
void bkt(k)
{
   int v;

   //parcurgem toate valorile posibile v pentru s[k]
   for(v = min_k; v <= max_k; v = succ(v))
   {
        //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]
                  //s[1],…,s[k] este soluție parțială, dar nu este
                  //soluție, deci adăugăm o nouă componentă s[k+1]
                  else
                        bkt(k+1);
      }
}
```

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

- bucăţile de cod scrise cu roşu 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 condiţii în care se utilizează indicii tabloului;
- $\texttt{min}_k$ şi $\texttt{max}_k$ se deduc din semnificaţia componentei `s[k]` a unei soluţii;
- aşa cum am menţionat anterior, în majoritatea problemelor, `succ(v)=v+1`;
- testarea faptului că `s[1],…,s[k]` este soluţie parţială se realizează, de obicei, în cadrul unei funcţii (în exemplele pe care le vom prezenta ulterior în limbajul C vom utiliza o funcţie `int solp(int k)`);
- pentru a testa că `s[1],…,s[k]` este soluţie vom ţine cont de faptul că `s[1],…,s[k]` este soluţie parţială, deci nu vom retesta condiţiile de continuare, ci doar pe cele suplimentare lor;
- dacă `s[1],…,s[k]` nu este soluţie parţială, atunci nu vom adăuga o nouă componentă `s[k+1]` prin apelul recursiv `bkt(k+1)`, deci instrucţiunea `for` va continua şi componentei curente `s[k]` i se va atribui următoarea valoare posibilă `v`, dacă aceasta există, iar în cazul în care aceasta nu există, instrucţiunea `for` corespunzătoare componentei curente `s[k]` se va termina şi, implicit, apelul funcţiei `bkt` corespunzător, deci se va reveni la componenta anterioară `s[k-1]`.

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

- `s[k]` reprezintă un element al permutării, deci $\texttt{min}_k=1$ şi $\texttt{max}_k=n$;
- `succ(v)=v+1`, deoarece valorile posibile pentru `s[k]` sunt numere naturale consecutive cuprinse între `1` şi `n`;
- `s[1],…,s[k-1],s[k]` este soluţie parţială dacă valoarea componentei curente `s[k]` nu a mai fost utilizată anterior, adică `s[k]`$\neq$`s[i]` pentru orice `i`$\in${`1,2,…,k-1`}. Se observă faptul că această condiţie este dedusă din predicatul P (care impune ca toate cele n valori `s[1],…,s[n]` dintr-o permutare de ordin n să fie distincte) şi este neapărat necesară (dacă `s[1],…,s[k]` nu sunt distincte, atunci, indiferent de ce valori am atribui celorlalte `n-k` componente `s[k+1],…,s[n]` nu vom obţine o permutare de ordin `n`), fără a fi şi suficientă (dacă valorile `s[1],…,s[k]` sunt distincte nu înseamnă că ele formează o permutare de ordin `n`, ci trebuie impusă condiţia suplimentară `k=n`);

- pentru a testa că `s[1],…,s[k]` este soluţie vom ţine cont de faptul că `s[1],…,s[k]` este soluţie parţială, deci nu vom retesta condiţiile de continuare (`s[k]≠s[i]` pentru orice `i∈{1,2,…,k-1}`), ci doar condiţia suplimentară `k=n`.

Aplicând observaţiile anterioare în forma generală a algoritmului de Backtracking, obţinem următoarea implementare în limbajul C a algoritmului de generare a tuturor permutărilor mulţimii $\{1,2,\dots,n\}$:

```
#include<stdio.h>

int s[101], n;

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

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

    for(v = 1; v <= n; v++)
    {
        s[k] = v;
        if(solp(k) == 1)
            if(k == 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;
}
```

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

3. Probleme de generări combinatoriale

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

3.1. Generarea aranjamentelor

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

Se observă foarte uşor faptul că singura diferenţă faţă de generarea permutărilor o constituie lungimea unei soluţii, care în acest caz este m în loc de n. De fapt, pentru $m = n$, aranjamentele unei mulţimi sunt chiar permutările sale!

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

3.2. Generarea combinărilor

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

Spre deosebire de tupluri, în care contează ordinea elementelor (de exemplu, tuplurile (1,2,3) şi (1,3,2) sunt considerate diferite), în cazul submulţimilor aceasta nu contează (de exemplu, submulţimile {1,2,3} şi {3,1,2} sunt considerate egale). Din acest motiv, trebuie să găsim o posibilitate de a evita prelucrarea unei soluţii care are aceleaşi elemente ca o altă soluţie generată anterior, dar în altă ordine. În acest sens, o variantă simplă, dar ineficientă, o reprezintă prelucrarea doar a soluţiilor care au elementele în ordine strict crescătoare, dar astfel vom încălca chiar principiul de bază al metodei Backtracking, acela de a evita generarea şi testarea unor tupluri inutile cât mai devreme posibil. O altă variantă o reprezintă generarea doar a soluţiilor cu elemente în ordine strict crescătoare, această restricţie putând fi impusă soluţiei curente în mai multe etape ale unui algoritm de tip Backtracking:

- *când testăm condiţiile de continuare, verificând faptul că* `s[k]>s[k-1]` – această variantă este mai eficientă decât prima, dar, totuşi se vor genera şi testa multe tupluri inutile. De exemplu, pentru a extinde soluţia parţială $(1, 3)$, se vor genera şi testa inutil tuplurile (1,3,1), (1,3,2) şi (1,3,3), deşi este evident faptul că la tuplul $(1, 3)$ are sens să adăugăm doar o valoare cel puţin egală cu 4;
- *iniţializând 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 şi testează tupluri inutile şi, 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 algoritm eficient de tip Backtracking pentru generarea combinărilor:

```
#include<stdio.h>

int s[101], n, m;

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

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

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

    printf("m = ");
    scanf("%d", &m);

    bkt(1);

    return 0;
}
```

Pentru `k=1`, variabila `v` din ciclul `for` va fi inițializată cu valoarea `s[0]+1`, respectiv chiar cu valoarea corectă `1`, deoarece tabloul s este declarat global, deci elementele sale sunt inițializate automat cu `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 $\mathcal{O}(C_n^m)$.