Tehnici de programare

Divide et impera

Pas 1: **Divide**: - se împarte problema în probleme mai mici (de aceeași structură)
Pas 2: **Conquer** - se rezolvă subproblemele recursiv
Pas 3: **Combine** - combinarea rezultatelor

Algoritm general:

```
def divideAndConquer(data):
    if size(data)<a:
        #solve the problem directly
        #base case
        return rez
    #decompose data into d1,d2,..,dk
    rez_1 = divideAndConquer(d1)
    rez_2 = divideAndConquer(d2)
    ...
    rez_k = divideAndConquer(dk)
    #combine the results
    return combine(rez_1,rez_2,...,rez_k)
```

Complexitate:

$$T(n) = \begin{cases} \text{solving trivial problem, if } n \text{ small enough} \\ k \cdot T(n/k) + \text{time for combining, otherwise} \end{cases}$$

Putem aplica Divide et impera dacă: O problemă P pe un set de date D poate fi rezolvată prin rezolvarea aceleiași probleme P pe un alt set de date D" = $d_1, d_2, \dots, d_k$ de dimensiune mai mică decât dimensiunea lui D

exemple:

→ căutarea binară
→ quicksort
→ merge sort
→ căutare maxim / minim în listă prin subliste

Backtracking

- se aplică la probleme de căutare unde se caută mai multe soluții
- generează toate soluțiile (dacă sunt mai multe) pentru problemă
- caută sistematic prin toate variantele de soluții posibile
- este o metodă sistematică de a itera toate posibilele configurații în spațiu de căutare
- este o technică generală – trebuie adaptat pentru fiecare problemă în parte.
- Dezavantaj – are timp de execuție exponențial

Metoda Generate & test

Problemă: Fie n un nr nat. Tipăriți toate permutările 1, 2, ..., n

ex: n = 3

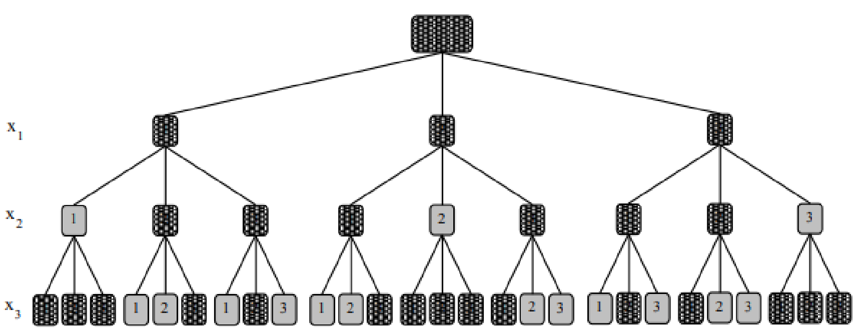
```
def perm3():
    for i in range(0,3):
        for j in range(0,3):
            for k in range(0,3):
                #a possible solution
                possibleSol = [i,j,k]
                if i!=j and j!=k and i!=k:
                    #is a solution
                    print possibleSol
```

⟹ [0, 1, 2]
[0, 2, 1]
[1, 0, 2]
[1, 2, 0]
[2, 0, 1]
[2, 1, 0]

Se generează toate variantele posibile de liste de lungime 3 care conțin doar numerele 0,1,2

Se testează fiecare variantă pt a se verifica dacă este soluție

Generare și testare – toate combinațiile posibile



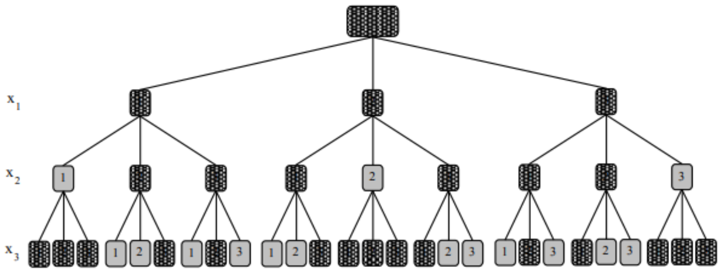
Probleme:

- Numărul total de **liste generate este** 3^3, în cazul general n^n
- inițial se generează toate componentele listei, apoi se verifica dacă lista este o permutare – in unele cazul nu era nevoie sa continuăm generarea (ex. Lista ce incepe cu 1,1 sigur nu conduce la o permutare
- Nu este general. Funcționează doar pentru n=3

În general: dacă n este afâncimea arborelui (numărul de variabile/componente în soluție) și presupunând că fiecare componentă poate avea k posibile valori, numărul de noduri în arbore este k^n. Înseamnă că pentru căutarea în întreg arborele avem o complexitate exponențială, $O(k^n)$.

Înbunătățiri posibile

- să evităm crearea comăletă a soluției posibile în cazul în care știm cu siguranță că nu se ajunge la o soluție.
  - Dacă prima componentă este 1, atunci nu are sens să asignam 1 să pentru a doua componentă



- lucrăm cu liste parțiale (soluție parțială)
- extindem lista cu componente noi doar dacă sunt îndeplinite anumite condiții (*condiții de continuare)*
  - *dacă lista parțială nu conține duplicate*

Generate & Test (recursiv)

folosim recursivitate pentru a genera toate soluțiile posibile (soluții candidat)

```
def generate(x,DIM):
    if len(x)==DIM:
        print x
    if len(x)>DIM:
        return
    x.append(0)
    for i in range(0,DIM):
        x[-1] = i
        generate(x[:],DIM)

generate([],3)
```

```
[0, 0, 0]
[0, 0, 1]
[0, 0, 2]
[0, 1, 0]
[0, 1, 1]
[0, 1, 2]
[0, 2, 0]
[0, 2, 1]
[0, 2, 2]
[1, 0, 0]
...
```

Testare – se tipărește doar soluția

```
def generateAndTest(x,DIM):
    if len(x)==DIM and isSet(x):
        print x
    if len(x)>DIM:
        return
    x.append(0)
    for i in range(0,DIM):
        x[-1] = i
        generateAndTest(x[:],DIM)
generateAndTest([],3)
```

```
[0, 1, 2]
[0, 2, 1]
[1, 0, 2]
[1, 2, 0]
[2, 0, 1]
[2, 1, 0]
```

- În continuare se genereaza toate listele ex: liste care încep cu 0,0
- ar trebui sa nu mai generăm dacă conține duplicate Ex (0,0) – aceste liste cu siguranță nu conduc la rezultat – la o permutare

Reducem spatiu de căutare – nu generăm chiar toate listele posibile

Un candidat e valid (merită să continuăm cu el) doar dacă nu conține duplicate

<pre>def backtracking(x,DIM):
    if len(x)==DIM:
        print x
    if len(x)>DIM:
        return #stop recursion
    x.append(0)
    for i in range(0,DIM):
        x[-1] = i
        if isSet(x):
            #continue only if x can conduct to a solution
            backtracking(x[:],DIM)
 	
backtracking([], 3)</pre>
	<pre>[0, 1, 2]
	[0, 2, 1]
	[1, 0, 2]
	[1, 2, 0]
	[2, 0, 1]
	[2, 1, 0]</pre>

Este mai bine decât varianta *generează și testează*, dar complexitatea ca timp de execuție este tot exponențial.

Permutation problem

- **rezultat:** $x=(x_0, x_1, .., x_n), x_i \in (0,1,..,n-1)$
- **e o solutie:** $x_i \neq x_j \; for\, any\, i \neq j$

8 Queens problem:

Plasați pe o tablă de sah 8 regine care nu se atacă.

- **Rezultat: 8 poziții de regine pe tablă**
- **Un rezultat partial e valid: dacă nu există regine care se atacă**
  - **nu e pe acelși coloana, linieor sau diagonală**
- **Numărul total de posibile poziții (atât valide cât și invalide):**
  - **combinări de 64 luate câte 8, C(64, 8) $\approx 4.5 \times 10^9$)**
- **Generează și testează – nu rezolvă problma în timp rezonabil**

Ar trebui sa generăm doar poziții care pot conduce la un rezultat (sa reducem spațiu de căutare)

- **Dacă avem deja 2 regine care se atacă nu ar trebui să mai continuăm cu această configurație**
- **avem nevoie de toate soluțiile**

Backtracking

- **spațiu de căutare**: $S = S_1 \times S_2 \times \ldots \times S_n$;
- ***x*** este un vector ce reprezintă **soluția**;
- $x[1..k]$ în $S_1 \times S_2 \times \ldots \times S_k$ este o **soluție candidat**; este o configurație parțială care ar putea conduce la rezultat; k este numărul de componente deja construită;
- **consistent** – o funcție care verifică dacă o soluție parțială este soluție candidat (poate conduce la rezultat)
- **soluție** este o funcție care verifică dacă o soluție candidat $x[1..k]$ este o soluție pentru problemă.

Algoritmul Backtracking – recursiv

```
def backRec(x):
    x.append(0) #add a new component to the candidate solution
    for i in range(0,DIM):
        x[-1] = i  #set current component
        if consistent(x):
            if solution(x):
                solutionFound(x)
            backRec(x[:]) #recursive invocation to deal with next components
```

Algoritm mai general (componentele soluției pot avea domenii diferite (iau valori din domenii diferite)

```
def backRec(x):
    el = first(x)
    x.append(el)
    while el!=None:
        x[-1] = el
        if consistent(x):
            if solution(x):
                outputSolution(x)
            backRec(x[:])
        el = next(x)
```

Backtracking

Cum rezolvăm problema folosind algoritmul generic:

- **trebuie sa reprezentăm soluția sub forma unui vector** $X=(x_0, x_1, \ldots x_n) \in S_0 \times S_1 \times \ldots \times S_n$
- **definim ce este o soluție candidat valid (condiție prin care reducem spațiu de căutare)**
- **definim condiția care ne zice daca o soluție candidat este soluție**

```
def consistent(x):
    """
      The candidate can lead to an actual
      permutation only if there are no duplicate elements
    """
    return isSet(x)


def solution(x):
    """
      The candidate x is a solution if
      we have all the elements in the permutation
    """
    return len(x)==DIM
```

Backtracking – iterativ

```
def backIter(dim):
    x=[-1]   #candidate solution
    while len(x)>0:
        choosed = False
        while not choosed and x[-1]<dim-1:
            x[-1] = x[-1]+1  #increase the last component
            choosed = consistent(x, dim)
        if choosed:
            if solution(x, dim):
                solutionFound(x, dim)
            x.append(-1) # expand candidate solution
        else:
            x = x[:-1]  #go back one component
```

Descrierea soluției:

ex: rezolvare permutări de N

soluție candidat

$x = (x_0, x_1, \ldots, x_k)$, $x_i \in (0, 1, \ldots, N-1)$

condiție consistent:

$x = (x_0, x_1, \ldots, x_k)$ consistent dacă $x_i \neq x_j$, pt $\forall i \neq j$

condiție soluție:

$x = (x_0, x_1, \ldots, x_k)$ e soluție dacă e consistent și $k = N-1$

- **spațiu de căutare**: $S = S_1 \times S_2 \times \ldots \times S_n$;
- ***x*** este un vector ce reprezintă **soluția**;
- $x[1..k]$ în $S_1 \times S_2 \times \ldots \times S_k$ este o **soluție candidat**; este o configurație parțială care ar putea conduce la rezultat; k este numărul de componente deja construită;
- **consistent** – o funcție care verifică dacă o soluție parțială este soluție candidat (poate conduce la rezultat)
- **soluție** este o funcție care verifică dacă o soluție candidat $x[1..k]$ este o soluție pentru problemă.

Metoda Greedy

Metoda Greedy este o metodă care poate fi uneori folosită în rezolvarea problemelor de următorul tip: *Se dă o mulțime `A`. Să se determine o submulțime `B` a lui `A` astfel încât să fie îndeplinite anumite condiții – acestea depinzând de problema propriu-zisă.*

Dacă un subset X îndeplinește condițiile interne atunci subsetul X este acceptabil (posibil)

Observații

- stabilirea elementului care va fi adăugat în soluția `B` se face alegându-l pe cel mai bun din acel moment – este un *optim local*. Din acest motiv se numește **Greedy** (lacom);
- după adăugarea în soluția `B` a unui anumit element, acesta va rămâne în soluție până la final. Nu există un mecanism de revenire la la un pas anterior, precum la metoda Backtracking;
- alegerea optimului local nu duce întotdeauna la cea mai bună soluție `B`; metoda Greedy nu este întotdeauna corectă;
- schema prezentată mai sus este vagă și nu poate fi standardizată – să avem un algoritm detaliat care să poată fi aplicat de fiecare dată;
- sunt relativ puține probleme care pot fi rezolvate cu metoda Greedy;
- complexitatea metodei este de regulă polinomială – $O(n^k)$, unde k este constant;
- folosim metoda Greedy în două situații:
  - știm sigur că rezolvarea este corectă (avem o demonstrație de natură matematică a corectitudinii);
  - nu avem decât soluții exponențiale (de tip Backtracking) și un algoritm Greedy dă o soluție nu neapărat optimă, dar acceptabilă.
- de regulă, înainte de începe alegerea elementelor convenabile din mulțimea `A`, elementele sale sunt ordonate după un criteriu specific, astfel încât alegerea optimului local să fie cât mai rapidă;

Greedy – Python

```python
def greedy(c):
    """
       Greedy algorithm
       c - a list of candidates
       return a list (B) the solution found (if exists) using the greedy
strategy, None if the algorithm
       selectMostPromissing - a function that return the most promising
candidate
       acceptable - a function that returns True if a candidate solution can be
extended to a solution
       solution - verify if a given candidate is a solution
    """
    b = [] #start with an empty set as a candidate solution
    while not solution(b) and c!=[]:
        #select the local optimum (the best candidate)
        candidate = selectMostPromissing(c)
        #remove the current candidate
        c.remove(candidate)
        #if the new extended  candidate solution is acceptable
        if acceptable(b+[candidate]):
            b.append(candidate)

    if solution(b):
        return b
    #there is no solution
    return None
```

Elemente

→ mulțimea candidat (candidate set)
de unde se aleg elementele

→ funcție de selecție (selection function)
alege cel mai bun candidat să fie adăugat la soluție

→ acceptabil (feasibility function)
folosit pt a determina dacă un candidat poate contribui la soluție

→ funcție obiectiv
o valoare pt soluție și pt orice soluție parțială

→ soluție (solution function)
indică dacă am ajuns la soluție

! Înainte de a aplica Greedy este nevoie să demonstrăm că metoda găsește soluția optimă. De multe ori demonstrația e netrivială

Există probleme pentru care Greedy nu găsește soluția optimă. În unele cazuri se preferă o soluție obținut în timp rezonabil (polinomial) care e aproape de soluția optimă, în loc de soluția optimă obținută în timp exponențial (*heuristics algorithms*).