Fundamentele programării

Divide and conquer - Metoda divizării - pași

- Pas 1 Divide - se împarte problema în probleme mai mici
- Pas 2 Conquer - se rezolvă subproblemele recursiv
- Pas 3 Combine - combinarea rezultatelor

Divide and conquer - 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)
```

Complexitatea ca timp de execuție:

$$T(n) = \begin{cases} \text{solving trivial problem} \\ K \cdot T(n/K) + \text{time for dividing} + \text{time for combining} \end{cases}$$

Divide and conquer 1/n-1

Putem divide datele în: date de dimensiune 1 și date de dimensiune n-1

Exemplu. Caută maximul

```
def findMax(l):
    """
      find the greatest element in the list
      l list of elements
      return max
    """
    if len(l)==1:
        #base case
        return l[0]
    #divide into list of 1 elements and a list of n-1 elements
    max = findMax(l[1:])
    #combine the results
    if max>l[0]:
        return max
    return l[0]
```

Complexitate timp:

Recurență: $T(n) = \begin{cases} 1 & \text{for } n = 1 \\ T(n-1)+1 & \text{otherwise} \end{cases}$

$T(n) = \cancel{T(n-1)} + 1$

$\cancel{T(n-1)} = \cancel{T(n-2)} + 1$ $\quad \Rightarrow T(n) = 1 + 1 + \ldots + 1 = n \in \Theta(n)$

$\cancel{\ldots}$

$\cancel{T(2)} = \cancel{T(1)} + 1$

$\cancel{T(1)} = 1$

Divizare în date de dimensiune n/k

```python
def findMax(l):
    """
      find the greatest element in the list
      l list of elements
      return max
    """
    if len(l)==1:
        #base case
        return l[0]
    #divide into 2 of size n/2
    mid = len(l) /2
    max1 = findMax(l[:mid])
    max2 = findMax(l[mid:])
    #combine the results
    if max1<max2:
        return max2
    return max1
```

Complexitate ca timp:

Recurență: $T(n) = \begin{cases} 1 & \text{for } n = 1 \\ 2T(n/2)+1 & \text{otherwise} \end{cases}$

$T(2^k) = 2T(2^{k-1}) + 1$

$2T(2^{k-1}) = 2^2 T(2^{k-2}) + 2$

$2^2 T(2^{k-2}) = 2^3 T(2^{k-3}) + 2^2$ $\quad$ Notăm: $n = 2^k \Rightarrow k = \log_2 n$

$$\ldots = \ldots$$

$$2^{(k-1)} T(2) = 2^{k} T(1) + 2^{(k-1)}$$

$$T(n) = 1 + 2^1 + 2^2 + \ldots + 2^k = \frac{2^{k+1} - 1}{2 - 1} = 2^k \cdot 2 - 1 = 2n - 1 \in \Theta(n)$$

Divide and conquer - Exemplu

Calculați x^k, $k \geq 1$

Abordare simplă: $x^k = x \cdot x \cdot \ldots \cdot x$, $k-1$ înmulțiri $T(n) \in \Theta(n)$

Rezolvare cu metoda divizării:

$$x^k = \begin{cases} x^{(k/2)} \cdot x^{(k/2)} & \text{for } k \text{ even} \\ x^{(k/2)} \cdot x^{(k/2)} \cdot x & \text{for } k \text{ odd} \end{cases}$$

```python
def power(x, k):
    """
      compute x^k
      x real number
      k integer number
      return x^k
    """
    if k==1:
        #base case
        return x
    #divide
    half = k/2
    aux =  power(x, half)
    #conquer
    if k%2==0:
        return aux*aux
    else:
        return aux*aux*x
```

Divide: calculează k/2

Conquer: un apel recursiv pentru a calcul $x^{(k/2)}$

Combine: una sau doua înmulțiri

Complexitate: $T(n) \in \Theta(\log_2 n)$

Divide and conquer

- Merge Sort
  - Divide - împărțim lista în două liste egale
  - Conquer - sortare recursivă pentru
  - Combine - interclasare liste sortate

Backtracking

- se aplică la probleme de căutare unde se caută mai multe soluții
- generează toate soluțiile
- caută sistematic prin toate variantele de soluții posibile
- este o tehnică generală - trebuie adaptată pentru fiecare problemă în parte
- dezavantaj - are timp de execuție exponențial

Metoda generării și testării

Ex: Fie n un număr natural. Tipăriți toate permutările numerelor 1, 2, ..., n. Pentru n = 3:

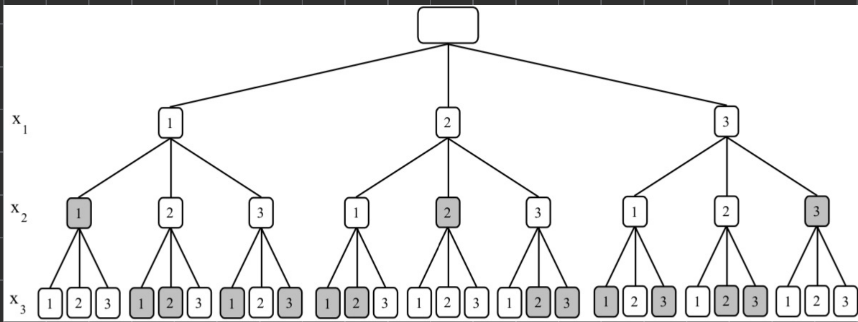
```python
def perm3():
    for i in range(0,3):
        for j in range(0,3):
            for k in range(0,3):
                #a possible solution
                possibleSol = [i,j,k]
                if i!=j and j!=k and i!=k:
                    #is a solution
                    print possibleSol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

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

Generate and Test:

- Generare: se generează toate variantele posibile
- Testare: se testează fiecare variantă pentru a verifica dacă este soluție

Generare și testare



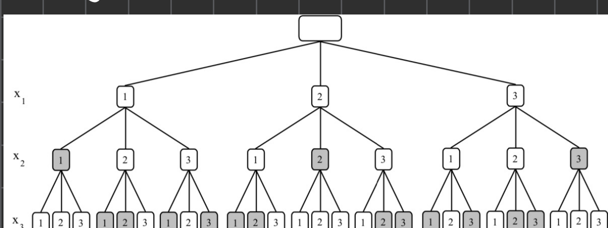
Probleme:

Numărul total de liste generate este 3^3, în cazul general n^n

Inițial se generează toate componentele listei, apoi se verifică dacă lista este o permutare

Îmbunătățiri posibile

- Să evităm crearea completă a soluției posibile în cazul în care știm că nu se ajunge la o soluție
  - Dacă prima componentă este 1, atunci nu are sens să asignăm 1 pentru a doua componentă



- lucrăm cu liste parțiale
- extindem lista cu componente noi doar dacă sunt îndeplinite anumite condiții precum: dacă lista parțială nu conține duplicate

Generate and test - recursiv

folosim recursivitatea pentru a genera toate soluțiile posibile:

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

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)
    x.pop()
generateAndTest([],3)
```

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

Pentru a îmbunătăți algoritmul ar trebui să nu mai generăm dacă conține duplicate. Un candidat e valid (merită să continuăm cu el) doar dacă nu conține duplicate

```
def backtracking(x,DIM):
    if len(x)==DIM:
        print (x)
        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)
    x.pop()

backtracking([], 3)
```

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

Este mai bine decât varianta generează și testează, dar complexitatea timp este tot exponențială.

Permutation problem

- rezultat: $x = (x_0, x_1, \ldots, x_n)$, $x_i \in (0, 1, \ldots, n-1)$
- o soluție: $x_i \neq x_j$ for any $i \neq j$

8 Queens problem:

Plasați pe o tablă 8 regine care nu se atacă

Numărul total de posibile poziții (valide + nevalide):

$C_{64}^{8} \approx 4.5 \cdot 10^9$

Generează și testează nu rezolvă problema în timp rezonabil

Ar trebui să generăm doar poziții care pot conduce la un rezultat: dacă avem 2 regine care se atacă nu ar trebui să mai continuăm cu această soluție

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
    x.pop()
```

Algoritm mai general

```
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)
```

Cum rezolvăm problema folosind algoritmul generic:

- definim ce este o soluție candidat valid (reducem spațiul de căutare)
- definim condiția care ne zice dacă un 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 backtracking

Rezolvare permutări de N

soluție candidat:

$x=(x_{0,}x_{1,..},x_k),\ x_i\in(0,1,..,N-1)$

condiție consistent:

$x=(x_{0,}x_{1,..},x_k)$ *e consistent dacă* $x_i\neq x_j$ *pentru* $\forall\ i\neq j$

condiție soluție:

$x=(x_{0,}x_{1,..},x_k)$ *e soluţie dacă e consistent şi* $k=N-1$

Rezolvare problema reginelor

soluție candidat:

$x=(x_{0,}x_{1,..},x_k),\ x_i\in(0,1,..,7)$

$(i,x_i)\forall\ i\in(0,1,..,k)$ *reprezintă poziţia unei regine pe tablă*

condiție consistent:

$x=(x_{0,}x_{1,..},x_k)$ *e consistent dacă reginele nu se atacă*

$x_i\neq x_j$ *pentru* $\forall\ i\neq j$ *nu avem două regine pe acelaşi coloană*

$|i-j|\neq|x_i-x_j|\forall\ i\neq j$ *nu se află pe acelaşi diagonală*

condiție soluție:

$x=(x_{0,}x_{1,..},x_k)$ *e soluţie dacă e consistent şi* $k=7$