Curs 13

- Tehnici de programare
- Organizare examen

Tehnici de programare

- **strategii de rezolvare a problemelor mai dificile**
- **algoritmi generali pentru rezolvarea unor tipuri de probleme**
- **de multe ori o problemă se poate rezolva cu mai multe tehnici – se alege metoda mai eficientă**
- **problema trebuie să satisfacă anumite criterii pentru a putea aplica tehnică**
- **descriem algoritmul general pentru fiecare tehnică**

Divide and conquer – Metoda divizării - pași

- Pas 1 **Divide -** se împarte problema în probleme mai mici (de același structură)
  - împărțirea problemei în două sau mai multe probleme disjuncte care se poate rezolva folosind același algoritm
- Pas 2 **Conquer –** se rezolvă subproblemele recursiv
- Step3 **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)
```

Putem aplica divide and conquer dacă:

O problemă *P* pe un set de date *D* poate fi rezolvat prin rezolvarea aceleiași probleme P pe un alt set de date D”= d_1, d_2, ..., d_k, de dimensiune mai mică decât dimensiunea lui *D*

Complexitatea ca timp de execuție pentru o problemă rezolvată folosind divide and conquer poate fi descrisă de recurența:

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

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

Recurența: $T(n)=\begin{cases} 1 \; for \; n=1 \\ T(n-1)+1 \; otherwise \end{cases}$

$$\begin{aligned} T(n) &= T(n-1)+1 \\ T(n-1) &= T(n-2)+1 \\ T(n-2) &= T(n-3)+1 \\ \ldots &= \ldots \\ T(2) &= T(1)+1 \end{aligned} \quad \Rightarrow \quad T(n)=1+1+\ldots+1=n\in\theta(n)$$

Divizare în date de dimensiune n/k

```
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:

Recurența: $T(n)=\begin{cases} 1 \; for \; n=1 \\ 2T(n/2)+1 \; otherwise \end{cases}$

Notăm: $n=2^k$ **=>** $k=\log_2 n$ $\begin{aligned} T(2^k) &= 2T(2^{(k-1)})+1 \\ 2T(2^{(k-1)}) &= 2^2T(2^{(k-2)})+2 \\ 2^2T(2^{(k-2)}) &= 2^3T(2^{(k-3)})+2^2 \\ \ldots &= \ldots \\ 2^{(k-1)}T(2) &= 2^kT(1)+2^{(k-1)} \end{aligned}$ **=>**

$$T(n)=1+2^1+2^2\ldots+2^k=(2^{(k+1)}-1)/(2-1)=2^k2-1=2n-1\in\theta(n)$$

Divide and conquer - Exemplu

Calculați x^k **unde** $k \geq 1$ **număr întreg**

Aborare simplă: $x^k = k * k * ... * k$ **- k-1 înmulțiri (se poate folosi un for)** $T(n) \in \Theta(n)$

Rezolvare cu metoda divizării:

$$x^k = \begin{cases} x^{(k/2)} x^{(k/2)} \; for\, k\, even \\ x^{(k/2)} x^{(k/2)} x \; for\, k\, odd \end{cases}$$

<table>
<tr>
<td>

```
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
```

</td>
<td>

Divide: calculează k/2

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

Combine: una sau doua înmulțiri

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

</td>
</tr>
</table>

Divide and conquer

- Căutare binară ($T(n) \in \theta(\log_2 n)$)
  - Divide – impărțim lista în două liste egale
  - Conquer – căutăm în stânga sau în dreapta
  - Combine – nu e nevoie
- Quick-Sort ($T(n) \in \theta(n \log_2 n)$ mediu)
- Merge-Sort
  - Divide – impărțim lista în două liste egale
  - Conquer – sortare recursivă pentru cele două liste
  - Combine – interclasare liste sortate

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 tehnică generală – trebuie adaptat pentru fiecare problemă în parte.**
- **Dezavantaj – are timp de execuție exponențial**

Algoritm general de descoperire a tuturor soluțiilor unei probleme
Se bazează pe construirea incrementală de soluții-candidat, abandonând fiecare candidat parțial imediat ce devine clar că acesta nu are șanse să devină o soluție validă

Metoda generării și testării (Generate and test)

Problemă – Fie *n* un număr natural. Tipăriți toate permutările numerelor 1, 2, ..., *n*.

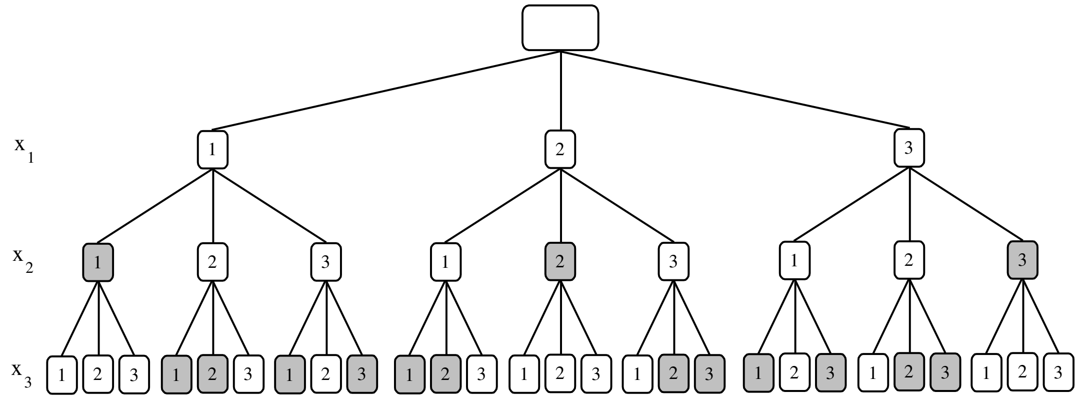
Pentru n=3

```
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]
```

- **Metoda generării și testării -** ***Generate and Test***
  - **Generare: se generează toate variantele posibile de liste de lungime 3 care conțin doar numerele 0,1,2**
  - **Testare: se testează fiecare variantă pentru a verifica dacă este soluție.**

Generare și testare – toate combinațiile posibile



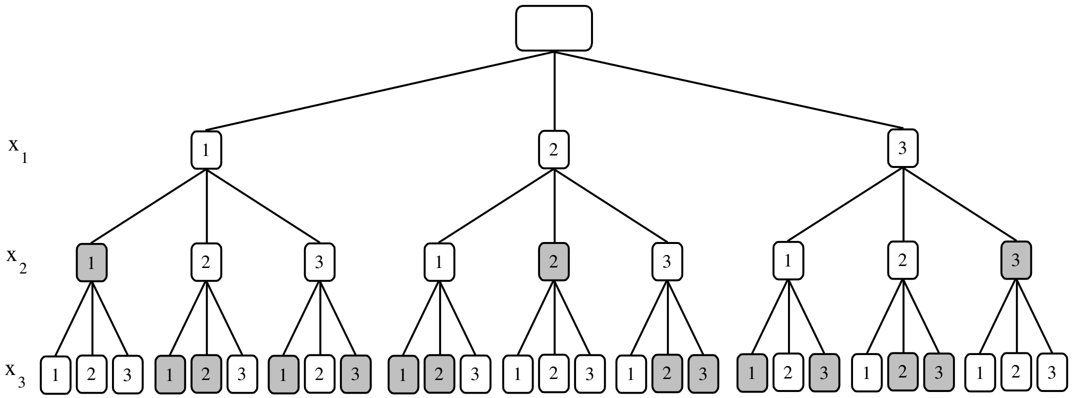
Probleme:

- Numărul total de **liste generate este $\mathbf{3^3}$**, în cazul general $\boldsymbol{n^n}$
- 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, $O(k^n)$.

Îmbunătățiri posibile

- **să evităm crearea completă 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 and test - recursiv

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

<pre>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)</pre>	<pre>[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] ...</pre>

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]```

- **În continuare se generează 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) 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)</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.

Permutări

- **rezultat:** $x=(x_{0,}x_{1,..},x_n), x_i\in(0,1,..,n-1)$
- **e o soluție:** $x_i\neq x_j \; for\, any\, i\neq j$

8 Queens problem:

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

- **Rezultat: 8 poziții de regine pe tablă**
- **Un rezultat parțial e valid: dacă nu există regine care se atacă**
  - **nu e pe același coloana, linie 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 problema 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
    x.pop()
```

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** $X=(x_{0,}x_{1,}\ldots x_n)\in S_0\,x\,S_1\,x\ldots x\,S_n$
- **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 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:

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

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

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

condiție soluție:

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

Implementare -

```
def consistentQ(x, dim):
    # we only check the last queen (the other queens checked before)
    for i in range(len(x) - 1):    # no queen on the same column
        if x[i] == x[-1]:
            return False
    # no queen on the same diagonal
    lastX = len(x)-1
    lastY = x[-1]
    for i in range(len(x)-1):
        if abs(i - lastX) == abs(x[i] - lastY):
            return False
    return True
```

```
def solutionQ(x, dim):
    return len(x) == dim
```

```
def solutionFoundQ(x, dim):
    # print a chess board
    for column in range(dim):
        # prepare a line
        cLine = ["0"] * dim
        cLine[x[column]] = "X"
        print (" ".join(cLine))
    print ("__"*dim)
```

```
backRecQ([], 8)
```