Curs 9 – Complexitatea algoritmilor

- Recursivitate
- Complexitate

Curs 8 – Testarea programelor

- Moștenire, UML
- Unit teste in Python
- Depanarea aplicațiilor Python

Recursivitate

O noțiune e recursivă dacă e folosit în propria sa definiție.

O funcție recursivă: funcție care se auto-apelează.
Rezultatul este obținut apelând același funcție dar cu alți parametrii

```
def factorial(n):
    """
      compute the factorial
      n  is a positive integer
      return n!
    """
    if n== 0:
        return 1
    return  factorial(n-1)*n
```

- Recursivitate directă: P apelează P
- Recursivitate indirectă: P apelează Q, Q apelează P

Cum rezolvăm probleme folosind recursivitatea:

- Definim cazul de bază: soluția cea mai simplă.
  - Punctul în care problema devine trivială (unde se oprește apelul recursiv)
- Pas inductiv: împărțim problema într-o variantă mai simplă al aceleași probleme plus ceva pași simplii
  - ex. apel cu n-1, sau doua apeluri recursive cu n/2

```
def recursiveSum(l):
    """
    Compute the sum of numbers
    l - list of number
    return int, the sum of numbers
    """
    #base case
    if l==[]:
        return 0
    #inductive step
    return l[0]+recursiveSum(l[1:])
```

```
def fibonacci(n):
    """
    compute the fibonacci number
    n - a positive integer
    return the fibonacci number for a given n
    """
    #base case
    if n==0 or n==1:
        return 1
    #inductive step
    return fibonacci(n-1)+fibonacci(n-2)
```

Obs recursiveSum(l[1:]):

l[1:] - crează o copie a listei l

exercițiu: modificați funcția recursiveSum pentru a evita l[1:]

Recursivitate în Python:

- la fiecare apel de metodă se creează o noua tabelă de simboluri (un nou namespace). Această tabelă conține valorile pentru parametrii și pentru variabilele locale
- tabela de simboluri este salvat pe stack, când apelul se termină tabela se elimină din stivă

```
def isPalindrome(str):
    """
      verify if a string is a palindrome
      str - string
      return True if the string is a palindrome False otherwise
    """
    dict = locals()
    print (id(dict),dict)

    if len(str)==0 or len(str)==1:
        return True

    return str[0]==str[-1] and isPalindrome(str[1:-1])
```

Recursivitate

Avantaje:

- claritate
- cod mai simplu

Dezavantaje:

- consum de memorie mai mare
    - pentru fiecare recursie se creează o nouă tabelă de simboluri

Analiza complexității

Analiza complexități – studiul eficienței algoritmilor.

Eficiența algoritmilor în raport cu:

- timpul de execuție – necesar pentru rularea programului
- spațiu necesar de memorie

Timp de execuție, depinde de:

- algoritmul folosit
- datele de intrare
- hardware-ul folosit
- sistemul de operare (apar diferențe de la o rulare la alta).

Exemplu timp de execuție

```
def fibonacci(n):
    """
     compute the fibonacci number
     n - a positive integer
     return the fibonacci number for a given n
    """
    #base case
    if n==0 or n==1:
        return 1
    #inductive step
    return fibonacci(n-1)+fibonacci(n-2)
```

```
def fibonacci2(n):
    """
     compute the fibonacci number
     n - a positive integer
     return the fibonacci number for a given n
    """
    sum1 = 1
    sum2 = 1
    rez = 0
    for i in range(2, n+1):
        rez = sum1+sum2
        sum1 = sum2
        sum2 = rez
    return rez
```

```
def measureFibo(nr):
    sw = StopWatch()
    print "fibonacci2(", nr, ") =", fibonacci2(nr)
    print "fibonacci2 take " +str(sw.stop())+" seconds"

    sw = StopWatch()
    print "fibonacci(", nr, ") =", fibonacci(nr)
    print "fibonacci take " +str(sw.stop())+" seconds"


measureFibo(32)
```

```
fibonacci2( 32 ) = 3524578
fibonacci2 take 0.0 seconds
fibonacci( 32 ) = 3524578
fibonacci take 1.7610001564 seconds
```

Eficiența algoritmilor

- Eficiența algoritmilor poate fi definită ca fiind cantitatea de resurse utilizate de algoritm (timp, memorie).

Măsurarea eficienței:

- analiză matematică a algoritmului - **analiză asimptotică**

Descrie eficiența sub forma unei funcții matematice.
Estimează timpul de execuție pentru toate intrările posibile.

- o analiză **empirică** a algoritmului

determinarea timpului exact de execuție pentru date specifice

nu putem prezice timpul pentru toate datele de intrare.

Timpul de execuție pentru un algoritm este studiat în relație cu dimensiunea datelor de intrare.

- Estimăm timpul de execuție în funcție de dimensiunea datelor.
- Realizăm o **analiză *asimptotică***. Determinăm ordinul de mărime pentru resursa utilizată (timp, memorie), ne interesează în special pentru cazurile în care datele de intrare sunt mari

Complexitate

- **caz favorabil** - datele de intrare care conduc la timp de execuție minim
  - *best-case complexity* (BC): $BC(A) = \min_{I \in D} E(I)$
- **caz defavorabil –** date de intrare unde avem cel mai mare timp de execuție.
  - *worst-case complexity* (WC): $WC(A) = \max_{I \in D} E(I)$
- **caz mediu** - timp de execuție.
  - *average complexity* (AC): $AC(A) = \sum_{I \in D} P(I)E(I)$

A - algoritm; *E(I)* număr de operații; P*(I)* probabilitatea de a avea *I* ca și date de intrare

D – mulțimea tuturor datelor de intrare posibile pentru un n fixat

Obs. Dimensiunea datelor (n) este fixat (**un număr mare**) caz favorabil/caz defavorabil se referă la un **anumit aranjament al datelor** de intrare care produc timp minim/maxim

Complexitate timp de execuție

- **numărăm pași** (operații elementare) efectuați (de exemplu numărul de instrucțiuni, număr de comparații, număr de adunări).
- numărul de pași nu este un număr fixat, **este o funcție**, notat $T(n)$, este in funcție de dimensiunea datelor (n), nu rezultă timpul exact de execuție
- Se surprinde doar esențialul: cum crește timpul de execuție în funcție de dimensiunea datelor. Ne oferă **ordinea de mărime** pentru timpul de execuție (dacă $n \to \infty$, atunci $3 \cdot n^2 \approx n^2$).
- putem **ignora constante mici** – dacă $n \to \infty$ aceste constante nu afectează ordinea de mărime.

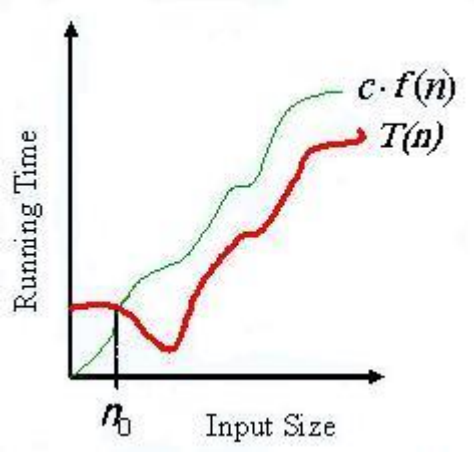
Ex : $T(n) = 13 \cdot n^3 + 42 \cdot n^2 + 2 \cdot n \cdot \log_2 n + 3 \cdot \sqrt{n}$

Fiindcă $0 < \log_2 n < n, \ \forall n > 1$ și $\sqrt{n} < n, \ \forall n > 1$, putem conclude că termenul n^3 domină această expresie când n este mare

Ca urmare, timpul de execuție a algoritmului crește cu ordinul lui n^3, ceea se scrie sub forma $T(n) \in O(n^3)$ și se citește "T(n) este de ordinul n^3

În continuare, vom nota prin f o funcție $f : N \to \Re$ și prin T funcția care dă complexitatea timp de execuție a unui algoritm, $T : N \to N$.

<u>Definiția 1</u> (Notația O, "Big-oh"). Spunem că $T(n) \in O(f(n))$ dacă există **c** şi $\mathbf{n_0}$ constante pozitive (care nu depind de n) astfel încât $0 \le T(n) \le c \cdot f(n), \quad \forall n \ge n_0$.



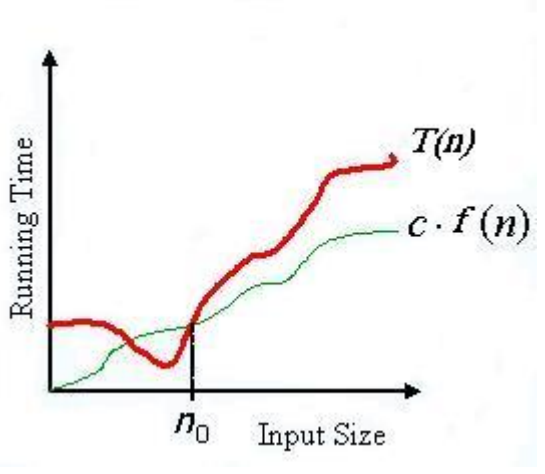
Cu alte cuvinte, notaţia O dă marginea superioară

Definiția alternativă: Spunem că $T(n) \in O(f(n))$ dacă $\lim_{n\to\infty} \frac{T(n)}{f(n)}$ este 0 sau o constantă, dar **<u>nu</u>** ∞.

Observaţii.

1. Dacă $T(n) = 13 \cdot n^3 + 42 \cdot n^2 + 2 \cdot n \cdot \log_2 n + 3 \cdot \sqrt{n}$, atunci $\lim_{n\to\infty} \frac{T(n)}{n^3} = 13$. Deci, putem spune că $T(n) \in O(n^3)$.
2. Notația *O* este bună pentru a da o limită superioară unei funcții. Observăm, totuși, că dacă $T(n) \in O(n^3)$, atunci este și $O(n^4)$, $O(n^5)$, etc atâta timp cât limita este 0. Din această cauză avem nevoie de o notație pentru limita inferioară a complexității. Această notație este Ω.

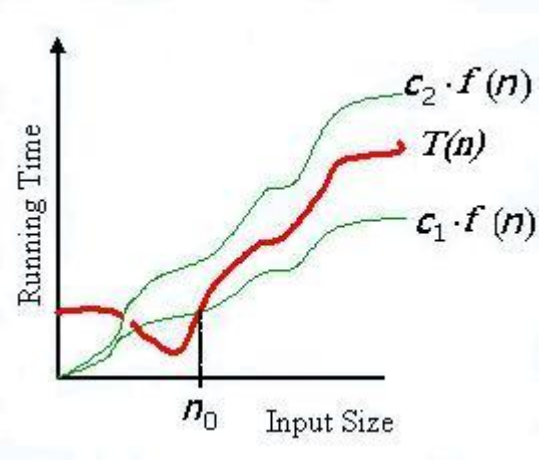
Definiţia 2 (Notaţia Ω, "Big-omega"). Spunem că $T(n) \in \Omega(f(n))$ dacă există **c** şi $\mathbf{n_0}$ constante pozitive (care nu depind de *n*) astfel încât $0 \le c \cdot f(n) \le T(n), \qquad \forall n \ge n_0$.



notația Ω dă marginea inferioară

Definiţia alternativă: Spunem că $T(n) \in \Omega(f(n))$ dacă $\lim_{n\to\infty} \frac{T(n)}{f(n)}$ este o constantă sau ∞, dar **nu** 0.

<u>Definiția 3</u> (Notația θ , "Big-theta"). Spunem că $T(n) \in \theta(f(n))$ dacă $T(n) \in O(f(n))$ şi dacă $T(n) \in \Omega(f(n))$, altfel spus dacă există **c1, c2** şi $\mathbf{n_0}$ constante pozitive (care nu depind de n) astfel încât $c1 \cdot f(n) \le T(n) \le c2 \cdot f(n), \quad \forall n \ge n_0$.



notația θ mărginește o funcție până la factori constanți

Definiţia alternativă Spunem că $T(n) \in \theta(f(n))$ dacă $\lim_{n\to\infty} \frac{T(n)}{f(n)}$ este o constantă nenulă (dar **<u>nu</u>** 0 sau ∞).

Observaţii.

1. Timpul de execuție al unui algoritm este $\theta(f(n))$ dacă și numai dacă timpul său de execuție în cazul cel mai defavorabil este $O(f(n))$ și timpul său de execuție în cazul cel mai favorabil este $\Omega(f(n))$.
2. Notația $O(f(n))$ este de cele mai multe ori folosită în locul notației $\theta(f(n))$.
3. Dacă $T(n) = 13 \cdot n^3 + 42 \cdot n^2 + 2 \cdot n \cdot \log_2 n + 3 \cdot \sqrt{n}$, atunci $\lim_{n\to\infty} \frac{T(n)}{n^3} = 13$. Deci, $T(n) \in \theta(n^3)$. Acest lucru poate fi dedus şi din faptul că $T(n) \in O(n^3)$ şi $T(n) \in \Omega(n^3)$.

Sume

for i *in range(0, n):*
    #some instructions

presupunând că ceea ce este în corpul structurii repetitive (*) se execută în *f(i)* pași timpul de execuție al întregii structuri repetitive poate fi estimat astfel

$$T(n) = \sum_{i=1}^{n} f(i)$$

Se poate observa că, în cazul în care se folosesc bucle imbricate, vor rezulta sume imbricate. În continuare, vom prezenta câteva dintre sumele uzuale:

Calculul se efectuează astfel:

- se simplifică sumele – eliminăm constantele, separăm termenii in sume individuale
- facem calculul pentru sumele simplificate.

Exemple cu sume

Analizați complexitatea ca timp de execuție pentru următoarele funcții

```
def f1(n):
    s = 0
    for i in range(1,n+1):
        s=s+i
    return s
```

$T(n) = \sum_{(i=1)}^{n} 1 = n \rightarrow T(n) \in \Theta(n)$

Complexitate (Overall complexity) $\Theta(n)$

Cazurile Favorabil/Mediu/Defavorabil sunt identice

```
def f2(n):
    i = 0
    while i<=n:
        #atomic operation
        i = i + 1
```

$T(n) = \sum_{(i=1)}^{n} 1 = n \rightarrow T(n) \in \Theta(n)$

Overall complexity $\Theta(n)$

Cazurile Favorabil/Mediu/Defavorabil sunt identice

```
def f3(l):
    """
    l - list of numbers
    return True if the list contains
an even nr
    """
    poz = 0
    while poz<len(l) and l[poz]%2 !=0:
        poz = poz+1
    return poz<len(l)
```

Caz favorabil:

primul element e număr par: $T(n) = 1 \in \Theta(1)$

Caz defavorabil:

Nu avem numere pare în listă: $T(n) = n \in \Theta(n)$

Caz mediu:

While poate fi executat 1,2,..n ori (același probabilitate).

Numărul de pași = numărul mediu de iterații

$T(n) = (1 + 2 + \ldots + n)/n = (n + 1)/2 \rightarrow T(n) \in \Theta(n)$

Complexitate $O(n)$

Exemple cu sume

```
def f4(n):
    for i in range(1,2*n-2):
        for j in range(i+2,2*n):
            #some computation
            pass
```

$$T(n) = \sum_{(i=1)}^{(2n-2)} \sum_{(j=i+2)}^{2n} 1 = \sum_{(i=1)}^{(2n-2)} (2n - i - 1)$$

$$T(n) = \sum_{(i=1)}^{(2n-2)} 2n - \sum_{(i=1)}^{(2n-2)} i - \sum_{(i=1)}^{(2n-2)} 1$$

$$T(n) = 2n \sum_{(i=1)}^{(2n-2)} 1 - (2n-2)(2n-1)/2 - (2n-2)$$

...

$T(n) = 2n^2 - 3n + 1 \in \Theta(n^2)$ Overall complexity $\Theta(n^2)$

```
def f5():
    for i in range(1,2*n-2):
        j = i+1
        cond = True
        while j<2*n and cond:
         #elementary operation
         if someCond:
                cond = False
```

Caz favorabil: While se execută odată $T(n) = \sum_{(i=1)}^{(2n-2)} 1 = 2n - 2 \in \Theta(n)$

Caz defavorabil: While executat $2n - (i+1)$ori

$$T(n) = \sum_{(i=1)}^{(2n-2)} (2n - i - 1) = \ldots = 2n^2 - 3n + 1 \in \Theta(n^2)$$

Caz mediu: Pentru un i fixat While poate fi executat 1,2..2n-i-1 ori număr mediu de pași: $C_i = (1 + 2 + \ldots + 2n - i - 1)/2n - i - 1 = \ldots = (2n - i)/2$

$$T(n) = \sum_{(i=1)}^{(2n-2)} C_i = \sum_{(i=1)}^{(2n-2)} (2n - i)/2 = \ldots \in \Theta(n^2)$$

Overall complexity $O(n^2)$

Formule cu sume:

$$\sum_{i=1}^{n} 1 = n$$ suma constantă.

$$\sum_{i=1}^{n} i = \frac{n(n+1)}{2}$$ suma liniară (progresia aritmetică)

$$\sum_{i=1}^{n} i^2 = \frac{n(n+1)(2n+1)}{6}$$ suma pătratică

$$\sum_{i=1}^{n} \frac{1}{i} = \ln(n) + O(1)$$ suma armonică

$$\sum_{i=1}^{n} c^i = \frac{c^{n+1}-1}{c-1}, \quad c \neq 1$$ progresia geometrică (crește exponențial)

Complexităţi uzuale

$T(n) \in O(1)$ - **timp *constant*.** Algoritmul se executa in timp constant indiferent de dimensiunea datelor.

$T(n) \in O(\log_2 \log_2 n)$ - timp foarte rapid (aproape la fel de rapid ca un timp constant)

$T(n) \in O(\log_2 n)$ - complexitate *logaritmică*:
timp foarte bun (este ceea ce căutăm, în general, pentru orice algoritm);

$\log_2 1000 \approx 10, \quad \log_2 1.000.000 \approx 20$;
complexitate căutare binară, înălțimea unei arbore binar echilibrat

$T(n) \in O((\log_2 n)^k)$ - unde *k* este factor constant; se numește complexitate *polilogaritmică* (este destul de bună);

Complexităţi uzuale

$T(n) \in O(n)$ - complexitate *liniară;*

$T(n) \in O(n \cdot \log_2 n)$ - este o complexitate faimoasă, întâlnită mai ales la sortări (MergeSort, QuickSort);

$T(n) \in O(n^2)$ - *este complexitate pătratică (cuadratică); dacă n este de ordinul milioanelor, nu este prea bună;*

$T(n) \in O(n^k)$ - *unde k este constant; este complexitatea polinomială (este practică doar daca k nu este prea mare);*

$T(n) \in O(2^n), O(n^3), O(n!)$ - complexitate *exponențială* (algoritmii cu astfel de complexități sunt practici doar pentru valori mici ale lui *n*: $n \le 10, n \le 20$).

Recurențe

O **recurență** este o formulă matematică definită recursiv.

Ex. numărul de noduri (notat *N(h)*) dintr-un arbore ternar complet de înălțime *h* ar putea fi descris sub forma următoarei formule de recurență:

$$\begin{cases} N(0) = 1 \\ N(h) = 3 \cdot N(h-1) + 1, \qquad h \geq 1 \end{cases}$$

Explicația ar fi următoarea:

- Numărul de noduri dintr-un arbore ternar complet de înălțime 0 este 1.
- Numărul de noduri dintr-un arbore ternar complet de înălțime *h* se obține ca fiind de 3 ori numărul de noduri din subarborele de înălțime *h-1*, la care se mai adaugă un nod (rădăcina arborelui).

Dacă ar fi să rezolvăm recurența, am obține că numărul de noduri din arborele ternar complet de înălţime *h* este $N(h) = 3^h \cdot N(0) + (1 + 3^1 + 3^2 + ... + 3^{h-1}) = \sum_{i=0}^{h} 3^i$

Exemple

```
def recursiveSum(l):
    """Compute the sum of numbers
    l - list of number
    return the sum of numbers """
    #base case
    if l==[]:
        return 0
    #inductive step
    return l[0]+recursiveSum(l[1:])
```

Recurrence: $T(n) = \begin{cases} 1 for n = 0 \\ T(n-1) + 1 otherwise \end{cases}$

$$T(n) = T(n-1) + 1$$
$$T(n-1) = T(n-2) + 1$$
$$T(n-2) = T(n-3) + 1 \Rightarrow T(n) = n + 1 \in \Theta(n)$$
$$\dots = \dots$$
$$T(1) = T(0) + 1$$

```
def hanoi(n, x, y, z):
    """
       n -number of disk on the x
stick
       x - source stick
       y - destination stick
       z - intermediate stick
    """
    if n==1:
      print ("disk 1 from",x,"to",y)
      return
    hanoi(n-1, x, z, y)
    print ("disk ",n,"from",x,"to",y)
    hanoi(n-1, z, y, x)
```

Recurrence: $T(n) = \begin{cases} 1 for n = 1 \\ 2T(n-1) + 1 otherwise \end{cases}$

$$T(n) = 2T(n-1) + 1$$
$$T(n-1) = 2T(n-2) + 1 =$$
$$T(n-2) = 2T(n-3) + 1 \Rightarrow$$
$$\dots = \dots$$
$$T(1) = T(0) + 1$$
$$T(n) = 2T(n-1) + 1$$
$$2T(n-1) = 2^2T(n-2) + 2$$
$$2^2T(n-2) = 2^3T(n-3) + 2^2$$
$$\dots = \dots$$
$$2^{(n-2)}T(2) = 2^{(n-1)}T(1) + 2^{(n-2)}$$

$$T(n) = 2^{(n-1)} + 1 + 2 + 2^2 + 2^3 + \dots + 2^{(n-2)}$$

$$T(n) = 2^n - 1 \in \Theta(2^n)$$

Complexitatea spațiu de memorare

Complexitatea unui algoritm din punct de vedere al *spațiului de memorare* estimează cantitatea de memorie necesară algoritmului pentru stocarea datelor de intrare, a rezultatelor finale şi a rezultatelor intermediare. Se estimează, ca și *timpul de execuție* al unui algoritm, în notațiile O, Θ, Ω.

Toate observațiile referitoare la notația asimptotică a complexității ca timp de execuție sunt valabile și pentru complexitatea ca spațiu de memorare.

Exemplu

Analizați complexitatea ca spațiu de memorare pentru următoarele funcții

`def iterativeSum(l):`
`    """`
`    Compute the sum of numbers`
`    l - list of number`
`    return int, the sum of numbers`
`    """`
`    rez = 0`
`    for nr in l:`
`        rez = rez+nr`
`    return rez`
	Avem nevoie de spațiu pentru numerele din listă
	
	$T(n) = n \in \Theta(n)$

`def recursiveSum(l):`
`    """`
`    Compute the sum of numbers`
`    l - list of number`
`    return int, the sum of numbers`
`    """`
`    #base case`
`    if l==[]:`
`        return 0`
`    #inductive step`
`    return l[0]+recursiveSum(l[1:])`
	Recurență:$T(n) = \{_{T(n-1)+1 otherwise}^{0 for n = 1}$

Analiza complexității (timp/spațiu) pentru o funcție

1 Dacă există caz favorabil/defavorabil:

- descrie **Caz favorabil**
- calculează complexitatea pentru **Best Case**
- descrie **Worst Case**
- calculează complexitatea pentru **Worst case**
- calculează complexitatea **medie**
- calculează complexitatea **generală**

2 Dacă Favorabil = Defavorabil = Mediu – (nu avem cazuri favorabile/defavorabile)

- calculează complexitatea

Calculează complexitatea:

- dacă avem recurență
  - calculează folosind egalități
- altfel
  - calculează folosind sume

Curs 9 – Complexitatea algoritmilor

- Recursivitate
- Complexitate

Curs 10 – Căutări sortări

- Căutări
- Sortări

Curs 10-11: Căutări - sortări

- **Căutări**
- **Algoritmi de sortare: selecție, selecție directă, inserție**

Curs 9 – Complexitatea algoritmilor

- Recursivitate
- Complexitate

Algoritmi de căutare

- datele sunt în memorie, o *secvență de înregistrări (k_1, k_2, ..., k_n)*
- se caută o înregistrare având un câmp egal cu o valoare dată – *cheia de căutare*.
- Dacă am găsit înregistrarea, se returnează poziția înregistrării în secvență
- dacă cheile sunt ordonate atunci ne interesează poziția în care trebuie inserată o înregistrare nouă astfel încât ordinea se menține

Specificații pentru căutare:

Date: a,n,(k_i, i=0,n-1);

Precondiții: $n \in N$, $n \geq 0$;

Rezultate: p;

Post-condiții: ($0 \leq p \leq n-1$ and $a = k_p$) or (p=-1 dacă cheia nu există).

Căutare secvențială – cheile nu sunt ordonate

```
def searchSeq(el,l):
    """
      Search for an element in a list
      el - element
      l - list of elements
      return the position of the element
         or -1 if the element is not in l
    """
    poz = -1
    for i in range(0,len(l)):
        if el==l[i]:
            poz = i
    return poz
```

$$T(n) = \sum_{(i=0)}^{(n-1)} 1 = n \in \Theta(n)$$

```
def searchSucc(el,l):
    """
      Search for an element in a list
      el - element
      l - list of elements
      return the position of first occurrence
            or -1 if the element is not in l
    """
    i = 0
    while i<len(l) and el!=l[i]:
        i=i+1
    if i<len(l):
        return i
    return -1
```

Best case: the element is at the first position
$T(n) \in \theta(1)$
Worst-case: the element is in the n-1 position
$T(n) \in \theta(n)$
Average case: while can be executed 0,1,2,n-1 times
$T(n) = (1 + 2 + \ldots + n - 1)/n \in \Theta(n)$
Overall complexity $O(n)$

Specificații pentru căutare – chei ordonate:

Date $a, n, (k_i, i=0,n-1)$;

Precondiții: $n \in N$, $n \geq 0$, *and* $k_0 < k_1 < .... < k_{n-1}$;

Rezultate p;

Post-condiții: $(p=0$ *and* $a \leq k_0)$ *or* $(p=n$ *and* $a > k_{n-1})$ *or*

$((0<p \leq n-1)$ *and* $(k_{p-1} < a \leq k_p))$.

Căutare secvențială – chei ordonate

```
def searchSeq(el,l):
    """
      Search for an element in a list
      el - element
      l - list of ordered elements
      return the position of first occurrence
             or the position where the element
             can be inserted
    """
    if len(l)==0:
        return 0
    poz = -1
    for i in range(0,len(l)):
        if el<=l[i]:
            poz = i
    if poz==-1:
        return len(l)
    return poz
```

```
def searchSucc(el,l):
    """
      Search for an element in a list
      el - element
      l - list of ordered elements
      return the position of first occurrence
            or the position where the element
            can be inserted
    """
    if len(l)==0:
        return 0
    if el<=l[0]:
        return 0
    if el>=l[len(l)-1]:
        return len(l)
    i = 0
    while i<len(l) and el>l[i]:
        i=i+1
    return i
```

$$T(n) = \sum_{(i=0)}^{(n-1)} 1 = n \in \Theta(n)$$

Best case: the element is at the first position
$T(n) \in \theta(1)$
Worst-case: the element is in the n-1 position
$T(n) \in \theta(n)$
Average case: while can be executed 0,1,2,n-1 times
$T(n) = (1 + 2 + \ldots + n - 1)/n \in \Theta(n)$
Overall complexity $O(n)$

Algoritmi de căutare

- *căutare secvențială*
  - se examinează succesiv toate cheile
  - cheile nu sunt ordonate
- *căutare binară*
  - folosește “divide and conquer”
  - cheile sunt ordonate

Căutare binară (recursiv)

```
def binaryS(el, l, left, right):
    """
      Search an element in a list
      el - element to be searched; l - a list of ordered elements
      left,right the sublist in which we search
      return the position of first occurrence or the insert position
    """
    if left>=right-1:
        return right
    m = (left+right)/2
    if el<=l[m]:
        return binaryS(el, l, left, m)
    else:
        return binaryS(el, l, m, right)

def searchBinaryRec(el, l):
    """
      Search an element in a list
      el - element to be searched
      l - a list of ordered elements
      return the position of first occurrence or the insert position
    """
    if len(l)==0:
        return 0
    if el<l[0]:
        return 0
    if el>l[len(l)-1]:
        return len(l)
    return binaryS(el, l, 0, len(l))
```

Recurență căutare binară

$$T(n) = \begin{cases} \theta(1), & if\ n = 1 \\ T\left(\frac{n}{2}\right) + \theta(1), & otherwise \end{cases}$$

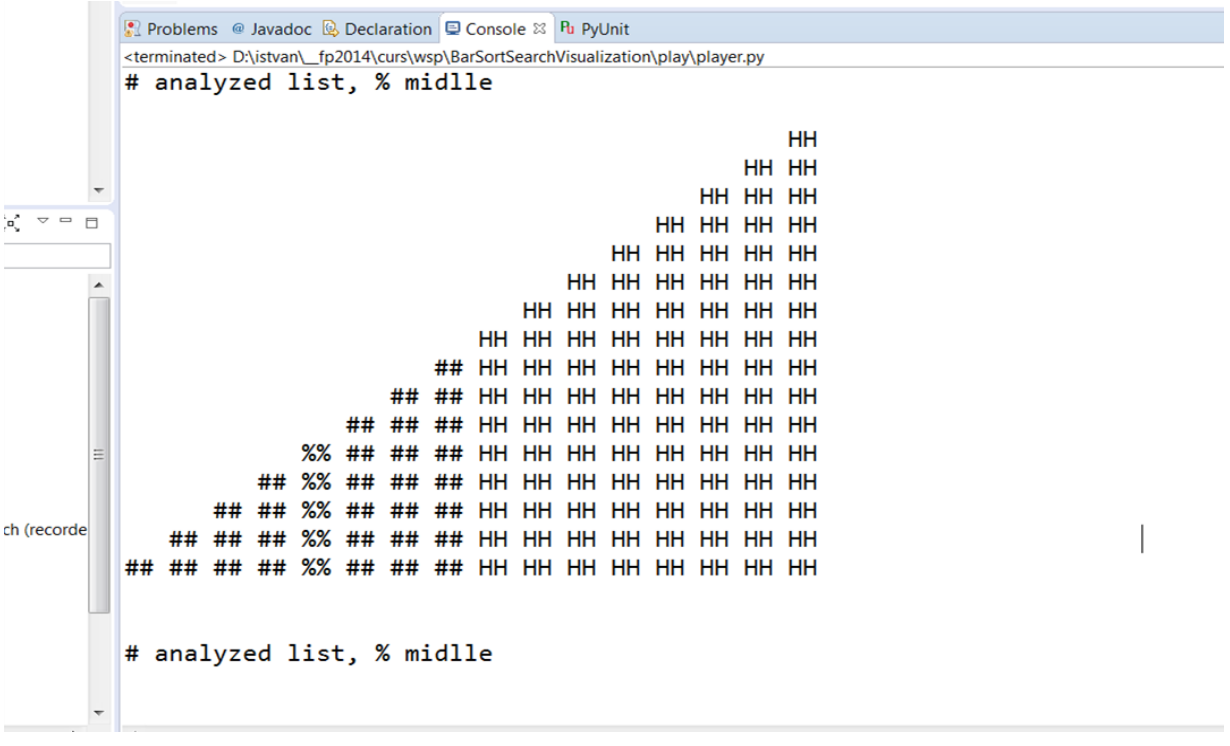
Căutare binară (iterativ)

```
def searchBinaryNonRec(el, l):
    """
      Search an element in a list
      el - element to be searched
      l - a list of ordered elements
      return the position of first occurrence or the position where the element can be
inserted
    """
    if len(l)==0:
        return 0
    if el<=l[0]:
        return 0
    if el>=l[len(l)-1]:
        return len(l)
    right=len(l)
    left = 0
    while right-left>1:
        m = (left+right)/2
        if el<=l[m]:
            right=m
        else:
            left=m
    return right
```

Complexitate

Algoritm	Timp de execuție			
	best case	worst case	average	overall
SearchSeq	$\theta(n)$	$\theta(n)$	$\theta(n)$	$\theta(n)$
SearchSucc	$\theta(1)$	$\theta(n)$	$\theta(n)$	$O(n)$
SearchBin	$\theta(1)$	$\theta(\log_2 n)$	$\theta(\log_2 n)$	$O(\log_2 n)$

Vizualizare căutări



Căutare in python - index()

```
l = range(1,10)
try:
    poz = l.index(11)
except ValueError:
    # element is not in the list
```

- __eq__

```
class MyClass:
    def __init__(self,id,name):
        self.id = id
        self.name = name

    def __eq__(self,ot):
        return self.id == ot.id


def testIndex():
    l = []
    for i in range(0,200):
        ob = MyClass(i,"ad")
        l.append(ob)

    findObj = MyClass(32,"ad")
    print ("positions:" +str(l.index(findObj)))
```

Searching in python- “in”

```
l = range(1,10)
found = 4 in l
```

– iterable (definiți __iter__ and __next__)

```
class MyClass2:
    def __init__(self):
        self.l = []

    def add(self,obj):
        self.l.append(obj)

    def __iter__(self):
        """
          Return an iterator object
        """
        self.iterPoz = 0
        return self
```

```
def __next__(self):
        """
          Return the next element in the iteration
     raise StopIteration exception if we are at the end
        """
        if (self.iterPoz>=len(self.l)):
            raise StopIteration()

        rez = self.l[self.iterPoz]
        self.iterPoz = self.iterPoz +1
        return rez

def testIn():
    container = MyClass2()
    for i in range(0,200):
        container.add(MyClass(i,"ad"))
    findObj = MyClass(20,"asdasd")
    print (findObj in container)
```

```
#we can use any iterable in a for
    container = MyClass2()
    for el in container:
        print (el)
```

Performanţă - căutare

```
def measureBinary(e, l):
    sw = StopWatch()
    poz = searchBinaryRec(e, l)
    print ("    BinaryRec in %f sec; poz=%i" %(sw.stop(),poz))

def measurePythonIndex(e, l):
    sw = StopWatch()
    poz = -2
    try:
        poz = l.index(e)
    except ValueError:
        pass #we ignore the error..
    print ("    PythIndex in %f sec; poz=%i" %(sw.stop(),poz))

def measureSearchSeq(e, l):
    sw = StopWatch()
    poz = searchSeq(e, l)
    print ("    searchSeq in %f sec; poz=%i" %(sw.stop(),poz))
```

```
search 200
    BinaryRec in 0.000000 sec; poz=200
    PythIndex in 0.000000 sec; poz=200
     PythonIn in 0.000000 sec
    BinaryNon in 0.000000 sec; poz=200
    searchSuc in 0.000000 sec; poz=200
```

```
search 10000000
    BinaryRec in 0.000000 sec; poz=10000000
    PythIndex in 0.234000 sec; poz=10000000
     PythonIn in 0.238000 sec
    BinaryNon in 0.000000 sec; poz=10000000
    searchSuc in 2.050000 sec; poz=10000000
```

Sortare

Rearanjarea datelor dintr-o colecţie astfel încât o cheie verifică o relaţie de ordine dată

- *internal sorting* - datele sunt în memorie
- *external sorting* - datele sunt în fişier

Elementele colecţiei sunt *înregistrări,* o înregistrare are una sau mai multe câmpuri
Cheia K este asociată cu fiecare înregistrare, în general este un câmp.

Colecţia este sortat:

- *crescător* după cheia *K :* if $K(i) \leq K(j)$ for $0 \leq i < j < n$
- *descrescător: if* $K(i) \geq K(j)$ *for* $0 \leq i < j < n$

Sortare internă – în memorie

Date n,K; {K=(k_1*,*k_2*,...,*k_n*)}*

Precondiții: $k_i \in R$, $i=1,n$

Rezultate K';

Post-condiții: K' e permutare al lui K, având elementele sortate,

$$k'_1 \leq k'_2 \leq \ldots \leq k'_n.$$

Sortare prin selecție (Selection Sort)

- se determină elementul având cea mai mica cheie, interschimbare elementul cu elementul de pe prima poziție
- reluat procedura penru restul de elemente până când toate elementele au fost considerate.

Sortare prin selecție

```
def selectionSort(l):
    """
       sort the element of the list
       l - list of element
       return the ordered list (l[0]<l[1]<...)
    """
    for i in range(0,len(l)-1):
        ind = i
        #find the smallest element in the rest of the list
        for j in range(i+1,len(l)):
            if (l[j]<l[ind]):
                ind =j
        if (i<ind):
            #interchange
            aux = l[i]
            l[i] = l[ind]
            l[ind] = aux
```

Complexitate – timp de execuție

Numărul total de comparații este:

$$\sum_{i=1}^{n-1} \sum_{j=i+1}^{n} 1 = \frac{n \cdot (n-1)}{2} \in \theta(n^2)$$

Este independent de datele de intrare:

- caz favorabil/defavorabil/mediu sunt la fel, complexitatea este $\theta(n^2)$.

Complexitate – spațiu de memorie

Sortare prin selecție – este un algoritm **In-place**:

memoria adițională (alta decât memoria necesară pentru datele de intrare) este $\theta(1)$

- *In-place* . Algoritmul care nu folosește memorie adițională (doar un mic factor constant).
- O*ut-of-place* sau *not-in-space*. Algoritmul folosește memorie adițională pentru sortare.

Selection sort este un algoritm an *in-place* .

Sortare prin selecție directă (Direct selection sort)

```
def directSelectionSort(l):
    """
       sort the element of the list
       l - list of element
       return the ordered list (l[0]<l[1]<...)
    """
    for i in range(0,len(l)-1):
        #select the smallest element
        for j in range(i+1,len(l)):
            if l[j]<l[i]:
                l[i],l[j] = l[j],l[i]
```

Overall time complexity: $\sum_{i=1}^{n-1}\sum_{j=i+1}^{n}1=\frac{n\cdot(n-1)}{2}\in\theta(n^2)$

Sortare prin inserție - Insertion Sort

- se parcurg elementele
- se inserează elementul curent pe poziția corectă în sub-secvența deja sortată.
- În sub-secvența ce conține elementele deja sortate se țin elementele sortate pe tot parcursul algoritmului, astfel după ce parcurgem toate elementele secvența este sortată în întregime

Sortare prin inserție

```
def insertSort(l):
    """
       sort the element of the list
       l - list of element
       return the ordered list (l[0]<l[1]<...)
    """
    for i in range(1,len(l)):
        ind = i-1
        a = l[i]
        #insert a in the right position
        while ind>=0 and a<l[ind]:
            l[ind+1] = l[ind]
            ind = ind-1
        l[ind+1] = a
```

Insertion Sort - complexitate – timp de execuție

Caz defavorabil: $T(n) = \sum_{i=2}^{n} (i-1) = \frac{n \cdot (n-1)}{2} \in \theta(n^2)$

Avem numărul maxim de iterații când lista este ordonat descrescător

Caz mediu: $\frac{n^2 + 3 \cdot n}{4} - \sum_{i=1}^{n} \frac{1}{i} \in \theta(n^2)$

Pentru un *i* fixat și un *k*, $1 \le k \le i$, probabilitatea ca x_i să fie al *k*-lea cel mai mare element în subsecvența $x_1, x_2, ..., x_i$ este $\frac{1}{i}$

Astfel, pentru *i* fixat, putem deduce:

Numărul de iterații **while**	Probabilitatea sa avem numărul de iterații **while** din prima coloană	caz
1	$\frac{1}{i}$	un caz în care **while** se execută odată: $x_i < x_{i-1}$
2	$\frac{1}{i}$	un caz în care **while** se execută de două ori: $x_i < x_{i-2}$
...	$\frac{1}{i}$	**...**
i-1	$\frac{2}{i}$	un caz în care **while** se execută de **i-1** ori: $x_i < x_1$ and $x_1 \leq x_i < x_2$

Rezultă că numărul de iterații **while** medii pentru un i fixat este:

$$c_i = 1 \cdot \frac{1}{i} + 2 \cdot \frac{1}{i} + ... + (i-2) \cdot \frac{1}{i} + (i-1) \cdot \frac{2}{i} = \frac{i+1}{2} - \frac{1}{i}$$

Caz favorabil: $T(n) = \sum_{i=2}^{n} 1 = n - 1 \in \theta(n)$ lista este sortată

Sortare prin inserție

- complexitate generală este $O(n^2)$.

Complexitate – spațiu de memorie

complexitate memorie adițională este: $\theta(1)$.

- *Sortare prin inserție* este un algoritm *in-place*.

Curs 10-11: Căutări - sortări

- **Algoritmi de sortare**
  - **metoda bulelor, quick-sort, tree sort, merge sort**
- **Sortare in Python: sort, sorted**
  - **parametrii: poziționali, prin nume, implicita, variabil**
  - **list comprehension, funcții lambda**

Curs 11: Sortări

- **Algoritmi de sortare**
  - **metoda bulelor, quick-sort, merge sort**
- **Sortare in Python: sort, sorted**
  - **parametrii: poziționali, prin nume, implicita, variabil**
  - **list comprehension, funcții lambda**

Curs 10-11: Căutări - sortări

- **Căutări**
- **Algoritmi de sortare: selecție, selecție directă, inserție**

Metoda bulelor - Bubble sort

- Compară elemente consecutive, dacă nu sunt în ordinea dorită le interschimbă.
- Procesul de comparare continuă până când nu mai avem elemente consecutive ce trebuie interschimbate (toate perechile respectă relația de ordine dată).

Sortare prin metoda bulelor

```
def bubbleSort(l):
    sorted = False
    while not sorted:
        sorted = True  # assume the list is already sorted
        for i in range(len(l)-1):
            if l[i+1]<l[i]:
                l[i], l[i + 1] = l[i + 1], l[i]
                sorted = False  # the list is not sorted yet
```

```
def bubbleSort2(l):
    for j in range(1,len(l)):
        for i in range(len(l)-j):
            if l[i+1]<l[i]:
               l[i], l[i + 1] = l[i + 1], l[i]
```

Complexitate metoda bulelor

Caz favorabil: $\theta(n)$. Lista este sortată

Caz defavorabil: $\theta(n^2)$. Lista este sortată descrescător

Caz mediu $\theta(n^2)$.

Complexitate generală este $O(n^2)$

Complexitate ca spațiu adițional de memorie este $\theta(1)$.

- este un algoritm de sortare ***in-place*** .

QuickSort

Introdus de Sir Charles Antony Richard (Tony) **Hoare** in 1959

Bazat pe “divide and conquer”

- **Divide**: se împarte lista în 2 astfel încât elementele din dreapta pivotului sunt mai mici decât elementele din stânga pivotului.
- **Conquer**: se sortează cele două subliste
- **Combine**: trivial – dacă partiționarea se face în același listă

Partiționare: re-aranjarea elementelor astfel încât elementul numit *pivot* ocupă locul final în secvență. Dacă poziția pivotului este i :

$$k_j \leq k_i \leq k_l \text{ , for } Left \leq j < i < l \leq Right$$

Quick-Sort

```
def partition(l,left,right):
    """
    Split the values:
            smaller pivot greater
    return pivot position
    post: left we have < pivot
          right we have > pivot
    """
    pivot = l[left]
    i = left
    j = right
    while i!=j:
        while l[j]>=pivot and i<j:
            j = j-1
        l[i] = l[j]
        while l[i]<=pivot and i<j:
            i = i+1
        l[j] = l[i]
    l[i] = pivot
    return i
```

```
def quickSortRec(l,left,right):

    #partition the list
    pos = partition(l, left, right)

    #order the left part
    if left<pos-1:
           quickSortRec(l, left, pos-1)
    #order the right part
    if pos+1<right:
           quickSortRec(l, pos+1, right)
```

QuickSort – complexitate timp de execuție

Timpul de execuție depinde de distribuția partiționării (câte elemente sunt mai mici decât pivotul câte sunt mai mari)

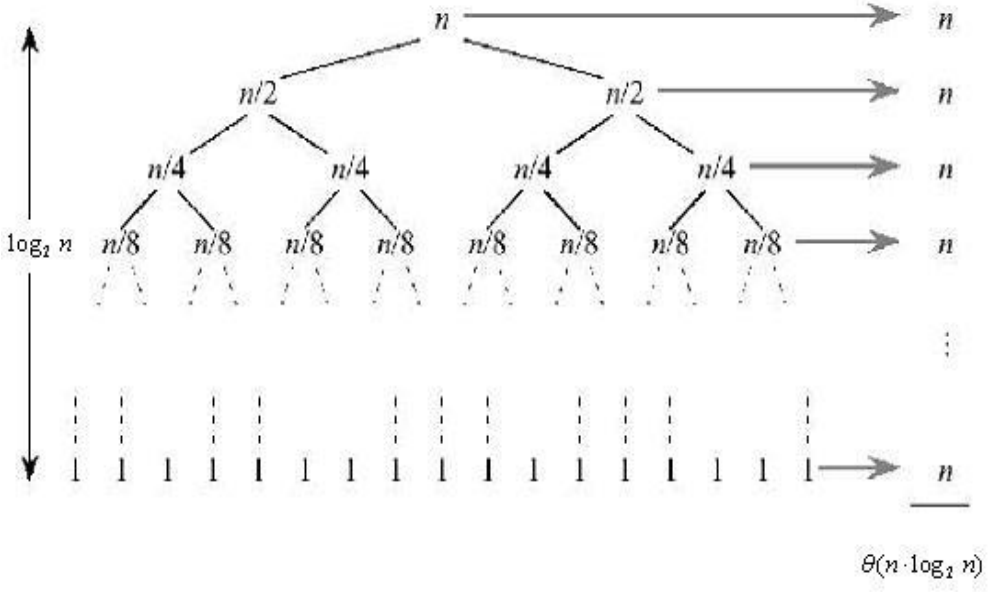
Partiționarea necesită timp linear.

Caz favorabil:, partiționarea exact la mijloc (numere mai mici ca pivotul = cu numere mai mari ca pivotul):

$$T(n) = 2 \cdot T\left(\frac{n}{2}\right) + \theta(n)$$

Complexitatea este $\theta(n \cdot \log n)$.

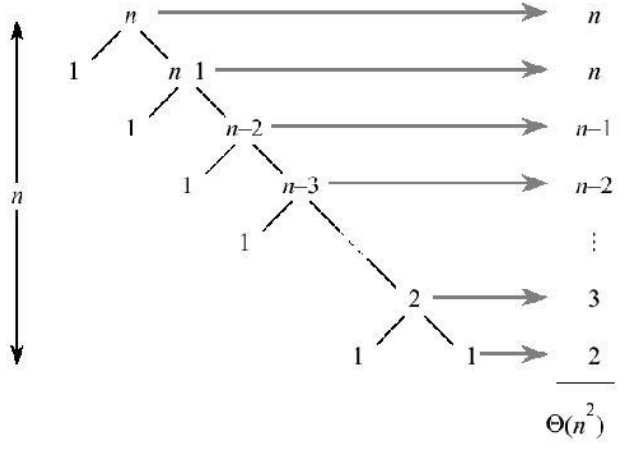
QuickSort – Caz favorabil



QuickSort – Caz defavorabil

Partiționarea tot timpul rezultă într-o partiție cu un singur element şi o partiţie cu n-1 elemente

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



caz defavorabil: dacă elementele sunt în ordine inversă

QuickSort – Caz mediu

Se alternează cazurile:

- caz favorabil (lucky) cu complexitatea $\theta(n \cdot \log n)$ (notăm cu L)
- caz defavorabil (unlucky) cu complexitatea $\theta(n^2)$ (notăm cu U).

Avem recurența:

$$\begin{cases} L(n) = 2 \cdot U\left(\frac{n}{2}\right) + \theta(n) & \text{lucky case} \\ U(n) = L(n-1) + \theta(n) & \text{unlucky case} \end{cases}$$

Rezultă

$$L(n) = 2 \cdot \left(L\left(\frac{n}{2} - 1\right) + \theta\left(\frac{n}{2}\right) \right) + \theta(n) = 2 \cdot L\left(\frac{n}{2} - 1\right) + \theta(n) = \theta(n \cdot \log_2 n),$$

Complexitatea caz mediu: $T(n) = L(n) \in \theta(n \cdot \log_2 n)$.

Coplexitatea ca timp de execuție pentru sortări:

	Complexity	
Algorithm	**worst-case**	**average**
SelectionSort	$\theta(n^2)$	$\theta(n^2)$
InsertionSort	$\theta(n^2)$	$\theta(n^2)$
BubbleSort	$\theta(n^2)$	$\theta(n^2)$
QuickSort	$\theta(n^2)$	$\theta(n \cdot \log n)$

Python - Quick-Sort

```
def quickSort(list):
    """
    Quicksort using list comprehensions
    return a new list
    """
    if len(list) <= 1:
        return list
    pivot = list.pop()
    lesser = quickSort([x for x in list if x < pivot])
    greater = quickSort([x for x in list if x >= pivot])
    return lesser + [pivot] + greater
```

List comprehensions – generatoare de liste

```
[x for x in list if x < pivot]
```

```
rez = []
for x in list:
    if x<pivot:
        rez.append(x)
```

- Variantă concisă de a crea liste
- creează liste unde elementele listei rezultă din operații asupra unor elemente dintr-o altă secvență
- paranteze drepte conținând o expresie urmată de o clauză for , apoi zero sau mai multe clauze for sau if

Python – Parametrii opționali parametrii cu nume

- Putem avea parametrii cu valori default;

```
def f(a=7,b = [],c="adsdsa"):
```

- Dacă se apelează metoda fără parametru actual se vor folosi valorile default

```
def f(a=7,b = [],c="adsdsa"):
    print (a)
    print (b)
    print (c)

f()
```

```
Console:
7
[]
adsdsa
```

- Argumentele se pot specifica în orice ordine

```
f(b=[1,2],c="abc",a=20)
```

```
Console:
20
[1, 2]
abc
```

- Parametrii formali se adaugă într-un dicționar (namespace)
- Trebuie oferit o valoare actuală pentru fiecare parametru formal - prin orice metodă: standard, prin nume, default value

Tipuri de parametrii – cum specificăm parametru actual

positional-or-keyword : parametru poate fi transmis prin poziție sau prin nume (keyword)

```
def func(foo, bar=None):
```

keyword-only : parametru poate fi transmis doar specificând numele

```
def func(arg, *, kw_only1, kw_only2):
```

Tot ce apare dupa * se poate transmite doar prin nume

var-positional : se pot transmite un număr arbitrar de parametri poziționali

```
def func(*args):
```

Valorile transmise se pot accesa folosind args, args este un tuplu

***var-keyword*:** un număr arbitrar de parametrii prin nume

```
def func(**args):
```

Valorile transmise se pot accesa folosind args, args este un dicționar

Sortare în python - list.sort() / funcție build in : sorted

sort(*, *key=None*,*reverse=None*)

Sortează folosind operatorul <. Lista curentă este sortată (nu se creează o altă listă)

key – o funcție cu un argument care calculează o valoare pentru fiecare element, ordonarea se face după valoarea cheii. În loc de o1 < o2 se face key(o1) < key(o2)

reverse – true daca vrem sa sortăm descrescător

```
l.sort()
print (l)
```

```
l.sort(reverse=True)
print (l)
```

sorted(*iterable[, key][, reverse]*)

Returnează lista sortată

```
l = sorted([1,7,3,2,5,4])
print (l)
```

```
l = sorted([1,7,3,2,5,4],reverse=True)
print (l)
```

```
def keyF(o1):
    return o1.name

ls = sorted(l,keyF)
```

Sort stability

Stabil (stable) – dacă avem mai multe elemente cu același cheie, se menține ordinea inițială

Python – funcţii lambda (anonymous functions, lambda form)

- Folosind `lambda` putem crea mici funcţii

```
lambda x:x+7
```

- Funcţiile lambda pot fi folosite oriunde e nevoie de un obiect funcţie

```
def f(x):
    return x+7

print ( f(5) )
```

```
print ( (lambda x:x+7)(5) )
```

- Putem avea doar o expresie.
- Sunt o metodă convenientă de a crea funcții mici.
- Similar cu funcțiile definite în interiorul altor funcții, funcțiile lambda pot referi variabile din namespace

```
l = []
l.append(MyClass(2,"a"))
l.append(MyClass(7,"d"))
l.append(MyClass(1,"c"))
l.append(MyClass(6,"b"))
#sort on name
ls = sorted(l,key=lambda o:o.name)
for x in ls:
    print (x)
```

```
l = []
l.append(MyClass(2,"a"))
l.append(MyClass(7,"d"))
l.append(MyClass(1,"c"))
l.append(MyClass(6,"b"))
#sort on id
ls = sorted(l,key=lambda o:o.id)
for x in ls:
    print (x)
```

MergeSort

Introdus de John von Neumann în 1945
Bazat pe “divide and conquer”.

Secvența este împărțită în două subsecvențe egale și fiecare subsecvența este sortată. După sortare se interclasează cele două subsecvențe, astfel rezultă secvența sortată în întregime.

Pentru subsecvențe se aplică același abordare până când ajungem la o subsecvență elementară care se poate sorta fără împărțire (secvență cu un singur element).

Python - Merge Sort

```
def mergeSort(l, start, end):
    """
       sort the element of the list
       l - list of element
       return the ordered list (l[0]<l[1]<...)
    """
    if end-start <= 1:
        return
    m = (end + start) // 2
    mergeSort(l, start, m)
    mergeSort(l, m, end)
    merge(l, start, end, m)
```

Apel: mergeSort(l,0,len(l))

Interclasare (Merging)

Date m, (x_i, i=1,m), n, (y_i, i=1,n);

Precondiții: {x_1 <= x_2...<=x_m} și {y_1 <= y_2 <=... <=y_n}

Rezultate k, (z_i, i=1,k);

Post-condiții: {k=m+n} și {z_1<=z_2<=...<=z_k} și (z_1,z_2,..., z_k) este o permutare a valorilor (x_1, ..., x_m,y_1,..., y_n)

complexitate interclasare: $\theta(m+n)$.

Complexitate sortare prin interclasare

Caz favorabil = Caz defavorabil = Caz mediu

$$T(n) = 2 \cdot T\left(\frac{n}{2}\right) + \theta(n)$$

Complexitate: $\theta(n \cdot \log n)$

Spațiu de memorare adițională pentru merge sort $\theta(n)$

Curs 11: Sortări

- **Algoritmi de sortare**
  - **metoda bulelor, quick-sort, merge sort**
- **Sortare in Python: sort, sorted**
  - **parametrii: poziționali, prin nume, implicita, variabil**
  - **list comprehension, funcții lambda**

Curs 12 – Technici de programare

- **Divide-et-impera (divide and conquer)**
- **Backtracking**

Curs 12 – Tehnici de programare

- **Divide-et-impera (divide and conquer)**
- **Backtracking**

Curs 11: Sortări

- **Algoritmi de sortare: metoda bulelor, quick-sort, tree sort, merge sort**
- **Sortare in Python: sort, sorted, parametrii list comprehension, funcții lambda**

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

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

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

Notăm: $n=2^k \Rightarrow k=\log_2 n$

$$\begin{aligned} T(2^k) &= 2T(2^{(k-1)})+1 \\ 2T(2^{(k-1)}) &= 2^2T(2^{(k-2)})+2 \\ 2^2T(2^{(k-2)}) &= 2^3T(2^{(k-3)})+2^2 \\ \ldots &= \ldots \\ 2^{(k-1)}T(2) &= 2^kT(1)+2^{(k-1)} \end{aligned} \Rightarrow$$

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

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:

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

```
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

- **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*.

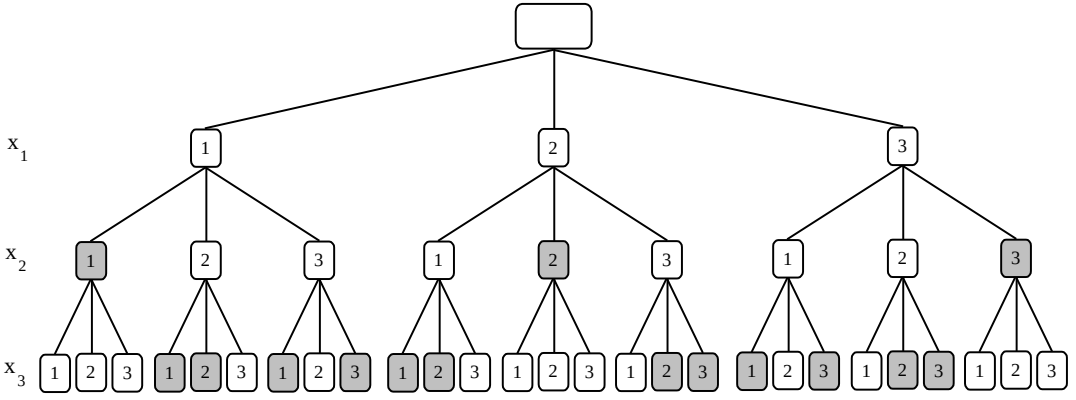
Pentru n=3

```
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



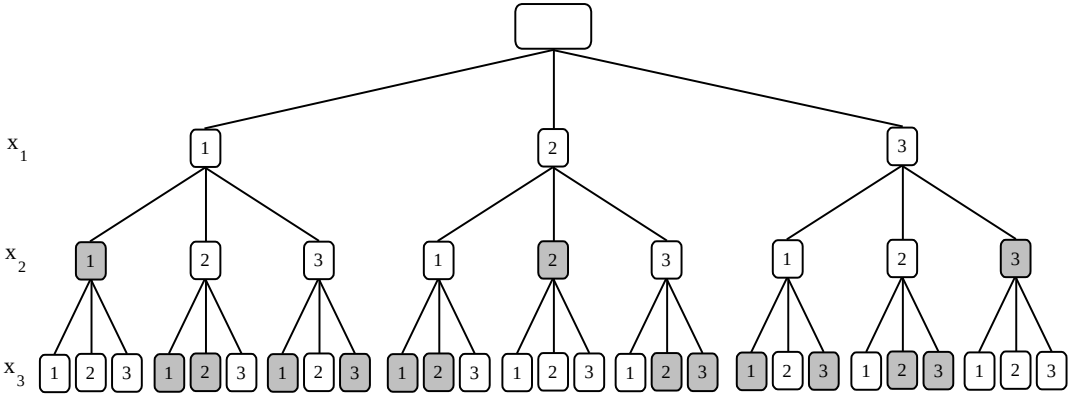
Probleme:

- Numărul total de **liste generate este 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, $O(k^n)$.

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)

```
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]
```

- **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

```
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 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**: $S = S_1 \times S_2 \times ... \times S_n$;
- ***x*** este un vector ce reprezintă **soluția**;
- $x[1..k]$ în $S_1 \times S_2 \times ... \times S_k$ 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** $X = (x_0, x_1, \dots x_n) \in S_0 \, x \, S_1 \, x \dots x \, S_n$
- **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 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* $k = 7$

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)
```

Curs 13 - Tehnici de programare

- **Greedy**
- **Programare dinamică**

Curs 12 – Tehnici de programare

- **Divide-et-impera (divide and conquer)**
- **Backtracking**

Metoda Greedy

- o strategie de rezolvare pentru probleme de optimizare
- aplicabil unde optimul global se poate afla selectând succesiv optime locale
- permite rezolvarea problemei fără a reveni asupra alegerilor făcute pe parcurs
- folosit în multe probleme practice care necesite selecția unei mulțimi de elemente care satisfac anumite condiții și realizează un optim

Probleme

Problema rucsacului – varianta fracționară

Se dă un set de obiecte, caracterizate prin greutate și utilitate, și un rucsac care are capacitatea totala W. Se caută o submulțime de obiecte astfel încât greutatea totală este mai mică decât W și suma utilității obiectelor este maximal.

Monede

Se da o sumă M și tipuri (ex: 1, 5, 25) de monede (avem un număr nelimitat de monede din fiecare tip de monedă). Să se găsească o modalitate de a plăti suma M de bani folosind cât mai puține monezi.

Forma generală a unei probleme Greedy-like

Având un set de obiecte C candidați pentru a face parte din soluție, se cere să se găsească un subset B (B $\subseteq$ C) care îndeplinește anumite condiții (condiții interne) și maximizează (minimizează) o funcție de obiectiv.

- Dacă un subset *X* îndeplinește condițiile interne atunci subsetul X este *acceptabil (*posibil)
- Unele probleme pot avea mai multe soluții acceptabile, caz în care se caută o soluție cât mai bună, daca se poate soluția ceea mai bună (cel care realizează maximul pentru o funcție obiectiv).

Pentru a putea rezolva o problema folosind metoda Greedy, problema trebuie să satisfacă proprietatea:

- dacă *B* este o soluție acceptabilă și *X* e submulțime al lui *B* atunci și *X* este o soluție acceptabilă

Algoritmul Greedy

Algoritmul Greedy găsește soluția incremental, construind soluții acceptabile care se tot extind pe parcurs. La fiecare pas soluția este extinsă cu cel mai bun candidat dintre candidații rămași la un moment dat. (Strategie greedy - lacom)

Principiu (strategia) Greedy :

- adaugă succesiv la rezultat elementul care realizează optimul local
- o decizie luată pe parcurs nu se mai modifică ulterior

Algoritmul Greedy

- Poate furniza soluția optimă (doar pentru anumite probleme)
  - alegerea optimului local nu garantează tot timpul optimul global
  - soluție optimă - dacă se găsește o modalitate de a alege (optimul local) astfel încât se ajunge la soluție optimă
  - în unele situații se preferă o soluție, chiar și suboptimă, dar obținut în timp rezonabil
- Construiește treptat soluția (fără reveniri ca în cazul backtracking)
- Furnizează o singură soluție
- Timp de lucru polinomial

Greedy – 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
```

Algoritm Greedy - elemente

1. **Mulțimea candidat** (***candidate set)*** **–** de unde se aleg elementele soluției
2. **Funcție de selecție** (***selection function)*** **–** alege cel mai bun candidat pentru a fi adăugat la soluție;
3. **Acceptabil** (***feasibility function)*** **–** folosit pentru a determina dacă un candidat poate contribui la soluție
4. **Funcție obiectiv** (***objective function)*** – o valoare pentru soluție și pentru orice soluție parțială
5. **Soluție (*solution function)***, - indică dacă am ajuns la soluție

Exemplu

Se da o sumă M și tipuri (ex: 1, 5, 25) de monede (avem un numar nelimitat de monede din fiecare tip de monedă). Să se găsească o modalitate de a plăti suma M de bani folosind cât mai puține monezi.

Set Candidat:

Lista de monede - COINS = {1, 5, 25, 50}

Soluție Candidat:

o listă de monede - $X = (X_0, X_{1,}., X_k)$ **unde** $X_i \in COINS$**– monedă**

Funcția de selecție:

Soluție candidat: $X = (X_0, X_{1,}., X_k)$

alege moneda cea mai mare care e mai mic decât ce mai e de plătit din sumă

Acceptabil (*feasibility function)*:

Soluție candidat: $X = (X_0, X_{1,}., X_k)$ $S = \sum_{(i=0)}^{k}$ $X_i \leq M$

suma monedelor din soluția candidat nu depășește suma cerută

Soluție:

Soluție candidat: $X = (X_0, X_{1,}., X_k)$ $S = \sum_{(i=0)}^{k}$ $X_i = M$

suma monedelor din soluția candidat este egal cu suma cerută

Monede – cod python

```
#Let us consider that we have a sum M of money and coins of 1, 5, 25 units (an unlimited number of coins).
#The problem is to establish a modality to pay the sum M using a minimum number of coins.
```

```
def selectMostPromissing(c):
    """
      select the largest coin from the remaining
      c - candidate coins
      return a coin
    """
    return max(c)
```

```
def acceptable(b):
    """
   verify if a candidate solution is valid
   basically verify if we are not over the sum M
    """
    sum = _computeSum(b)
    return sum<=SUM
```

```
def solution(b):
    """
   verify if a candidate solution is an actual solution
   basically verify if the coins conduct to the sum M
   b - candidate solution
    """
    sum = _computeSum(b)
    return sum==SUM

def _computeSum(b):
    """
   compute the payed amount with the current candidate
    return int, the payment
   b - candidate solution
    """
    sum = 0
    for coin in b:
        nrCoins = (SUM-sum) / coin
        #if this is in a candidate solution we need to
use at least 1 coin
        if nrCoins==0: nrCoins =1
        sum += nrCoins*coin
    return sum
```

```
def printSol(b):
 """
   Print the solution: NrCoinns1 * Coin1 +  NrCoinns2 *
Coin2 +...
 """
    solStr = ""
    sum = 0
    for coin in b:
        nrCoins = (SUM-sum) / coin
        solStr+=str(nrCoins)+"*"+str(coin)
        sum += nrCoins*coin
        if SUM-sum>0:solStr+=" + "
    print solStr
```

Greedy

1. Algoritmul Greedy are complexitate polinomială - $O(n^2)$ unde n este numărul de elemente din lista candidat C
2. Înainte de a aplica Greedy este nevoie să demonstrăm că metoda găsește soluția optimă. De multe ori demonstrația este netrivială
3. Există o mulțime de probleme ce se pot rezolva cu greedy. Ex: Algoritmul Kruskal – determinarea arborelui de acoperire, Dijkstra, Bellman-Kalaba – drum minim într-un graf neorientat
4. 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).*

Programare Dinamică

Se poate folosi pentru a rezolva probleme de optimizare, unde:

- soluția este rezultatul unui șir de decizii, $d_1, d_2, ..., d_n$,
- *principiul optimalității* este satisfăcut.
- în general timp polinomial de execuție
- tot timpul găsește soluția optimă.
- Se definește structura soluției folosind o recurentă
- Rezolvă problema combinând sub soluții de la subprobleme, calculează sub soluția doar o singură data (salvând rezultatul pentru a fi refolosit mai târziu).

Fie stările $s_0, s_1, ..., s_{n-1}, s_n$, unde s_0 este starea inițială, s_n este starea finală, prin aplicarea succesivă a deciziilor $d_1, d_2, ..., d_n$ se ajunge la starea finală (decizia d_i duce din starea s_{i-1} în starea s_i, pentru i=1,n):

$$s_0 \xrightarrow{d_1} s_1 \xrightarrow{d_2} s_2 \rightarrow \dots \rightarrow s_{n-1} \xrightarrow{d_n} s_n$$

Programare Dinamică – exemplu Fibonacci

Recursiv - ineficient	Recursiv – folosește memoizare	Programare Dinamică

`def fibo(n):`
`  if n<=2: return 1`
`  return fibo(n-1)+fibo(n-2)`
	`def fiboMem(n,mem):`
	`  if n in mem: return mem[n]`
	`  if n<=2: rez = 1`
	`  else: rez=fiboMem(n-1,mem)+fiboMem(n-2,mem)`
	`  mem[n] = rez`
	`  return rez`
	
	`def fiboMemoization(n):`
	`  return fiboMem(n, {})`
		`def fiboDP(n):`
		`  """`
		`  Dynamic programming`
		`  bottom up (backward) variant`
		`  DP(1) = DP(2) = 1`
		`  DP(i) = DP(i-1) + DP(i-2)`
		`  """`
		`  if n<=2: return 1`
		`  mem = [None]*(n)`
		`  mem[0] = 1`
		`  mem[1] = 1`
		`  for i in range(2,n):`
		`  mem[i] = mem[i-1]+mem[i-2]`
		`  return mem[n-1]`

Complexitate: $0(2^n)$	*Complexitate: 0(n)*	*Complexitate: 0(n)*
	Salvăm numărul Fibonacci odată calculat intr-un dicționar si folosim ulterior (in loc sa recalculam)	*Salvăm (memoizare) rezultatele de la subprobleme si calculam soluția finală refolosind soluțiile de la subprobleme*

Programare Dinamică – exemplu sublistă crescătoare

Având o lista l = [1,2,1,3,4,2,1] avem următoarele subliste crescătoare: [1],[1,2],[1,2,3],[1,2,3,4],[1,3,4],[1,4],...etc.

Pentru o lista data găsiți lungimea sublistei crescătoare de lungime maxima din lista data.

***Ex: l* = [1, 2, 3, 1, 5, 3, 4, 1, 4] rezultat: 6 (lista:[1,2,3,3,4,4])**

Recursiv	*Recursiv cu memoizare*	*Programare dinamică*

`def lgSublistaCresc(l,poz):`
`  if poz==len(l)-1:`
`  return 1`
`  maxLg = 1`
`  for i in range(poz+1,len(l)):`
`  if l[poz]<=l[i]:`
`  lg = 1+lgSublistaCresc(l,i)`
`  if lg>maxLg: maxLg = lg`
`  return maxLg`
	`def lgSublistaCrescMem(l,poz,mem):`
	`  if poz==len(l)-1:`
	`  return 1`
	`  if poz in mem: return mem[poz]`
	`  maxLg = 1`
	`  for i in range(poz+1,len(l)):`
	`  if l[poz]<=l[i]:`
	`  lg = 1+lgSublistaCrescMem(l,i,mem)`
	`  if lg>maxLg: maxLg = lg`
	`  mem[poz] = maxLg`
	`  return maxLg`
		`def lgSublistaCrescDP(l):`
		`  lgs = [0]*len(l)`
		`  lgs[-1] = 1`
		`  for i in range(len(l)-2,-1,-1):`
		`  lgs[i] = 1`
		`  for j in range(i+1,len(l)):`
		`  if l[i]<=l[j] and lgs[i]<lgs[j]+1:`
		`  lgs[i] = lgs[j]+1`
		`  return max(lgs)`

Se calculează de mai multe ori lungimea maximă a sublistei crescătoare care începe de la o anumita poziție	Folosim memoizare pentru a evita recalcularea	*DP(n) = 1* *DP(i)=1+max(DP(j) unde l[i]<=l[j],j=i..n)* *DP – varianta forward*

Programare Dinamică

Caracteristici:

- principiul optimalității;
- probleme suprapuse (overlapping sub problems);
- *memoization.*

Principiul optimalității

- *optimul general* implică *optimul parțial*
  - la greedy aveam *optimul local* implică *optimul global*
- într-o secvență de decizii optime, fiecare decizie este optimă.
- Principiul nu este satisfăcut pentru orice fel de problemă. In general nu e adevărat în cazul in care sub-secvențele de decizii nu sunt independente și optimizarea uneia este în conflict cu optimizarea de la alta secvența de decizii.

Principiul optimalității

Dacă $d_1, d_2, ..., d_n$ este o secvență de decizii care conduce optim sistemul din starea inițială s_0 în starea finală s_n, atunci una din următoarele sunt satisfăcute:

1). $d_k, d_{k+1}, ..., d_n$ este o secvență optimă de decizii care conduce sistemul din starea s_{k-1} în starea s_n, $\forall k, 1 \le k \le n$. **(*forward* variant –** decizia d_k depinde de deciziile $d_{k+1}..d_n$**)**

2). $d_1, d_2, ..., d_k$ este o secvență optimă de decizii care conduce sistemul din starea s_0 în starea s_k, $\forall k, 1 \le k \le n$. **(*backward* variant)**

3). $d_{k+1}, d_{k+2}, ..., d_n$ și $d_1, d_2, ..., d_k$ sunt secvențe optime de decizii care conduc sistemul din starea s_k în starea s_n, respectiv, din starea s_0 în starea s_k, $\forall k, 1 \le k \le n$. **(*mixed* variant)**

Sub-Probleme suprapuse (Overlapping Sub-problems)

O problema are sub-probleme suprapuse daca poate fi împărțit în subprobleme care se refolosesc de mai multe ori

Memorizare (Memorization)

salvarea rezultatelor de la o subproblemă pentru a fi refolosit

Cum aplicăm programare dinamică

- Principiul optimalității (oricare variantă: forward, backward or mixed) este demonstrat.
- Se definește structura soluției optime.
- Bazat pe principiul optimalității, valoarea soluției optime se definește recursiv. Se definește o recurență care indică modalitatea prin care se obține optimul general din optime parțiale.
- Soluția optimă se calculează in manieră bottom-up, începem cu subproblema cea mai simplă pentru care soluția este cunoscută.

Cea mai lungă sub listă crescătoare

Se dă o listă $a_1, a_2, ..., a_n$. *Determinaţi cea mai lungă sub listă crescătoare* $a_{i_1}, a_{i_2}, ..., a_{i_s}$ *a listei date.*

Soluție:

- *Principiul optimalităţii*
  - varianta înainte *forward*
- *Structura soluţiei optime:*
  - Construim două șiruri: $l =< l_1.l_2, ...l_n >$ și $p =< p_1.p_2, ...p_n >$.
    - l_k lungime sub listei care începe cu elementul a_k.
    - p_k indexul elementului a care urmează după elementul a_k în sublista cea mai lungă care începe cu a_k.
- *Definiţia recursivă*

_ $l_n = 1,\ p_n = 0$

_ $l_k = \max\{1 + l_i \mid a_i \geq a_k,\ k+1 \leq i \leq n\}, \quad \forall k = n-1, n-2, ...1$

_ $p_k = \arg\max\{1 + l_i \mid a_i \geq a_k,\ k+1 \leq i \leq n\}, \quad \forall k = n-1, n-2, ..., 1$

Cea mai lungă sublistă crescătoare– python

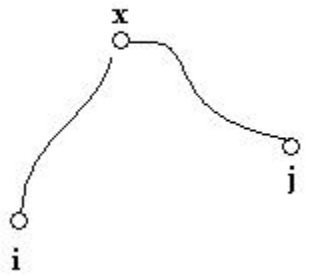
```
def longestSublist(a):
    """
      Determines the longest increasing sub-list
      a -  a list of element
      return sublist of x, the longest increasing sublist

    """
    #initialise l and p
    l = [0]*len(a)
    p = [0]*len(a)
    l[lg-1] = 1
    p[lg-1]=-1
    for k in range(lg-2, -1, -1):
        p[k] = -1
        l[k]=1
        for i in range(k+1, lg):
            if a[i]>=a[k] and l[k]<l[i]+1:
                l[k] = l[i]+1
                p[k] = i

    #identify the longest sublist
    #find the maximum length
    j = 0
    for i in range(0, lg):
        if l[i]>l[j]:
            j=i
    #collect the results using the position list
    rez = []
    while j!=-1:
        rez = rez+[a[j]]
        j = p[j]
    return rez
```

Dynamic programming vs. Greedy

- ambele se aplică în probleme de optimizare
- Greedy : *optimul general* se obține din optime *parțiale* (*locale*)
- DP *optimul general* implică optimul *parțial*.

Determinați drumul cel mai scurt între nodul **i** și **j** într-un graf:

- *Principiul optimalității* este verificat: daca drumul de la **i** la **j** este optimal și trece prin nodul **x**, atunci drumul de la **i** la **x**, și de la **x** la **j** este optimal.



- DP poate fi aplicat pentru a rezolva problema drumului minim
- Greedy nu se poate aplica fiindcă: dacă drumul de la **i** la **x**, și de la **x** la **j** este optim, nu există garanții că drumul cel mai scurt de la **i** la **j** trece prin **x**.