Căutare secvențială – cheile nu sunt ordonate

```python
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
```

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

```python
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 $O(n)$

Căutare secvențială – chei ordonate

```python
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
```

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

```python
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
```

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 $O(n)$

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))
```

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

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
```

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	$\theta(1)$	$\theta(\log_2 n)$	$\theta(\log_2 n)$	$O(\log_2 n)$

Selection Sort – Sortare prin selecție

```
def SelectionSort(l):
    for i in range(0, len(l) - 1):
        ind = i
        for j in range(i + 1, len(l)):
            if (l[j] < l[ind]):
                ind = j
        if (i < ind):
            l[i], l[ind] = l[ind], l[i]
    return l
```

Numărul total de conparații este:

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

Complexitatea este: $\theta(n^2)$.

Este un algoritm in-place.

Direct Selection Sort – Sortare prin selecție directă

```
def DirectSelectionSort(l):
    for i in range(0, len(l)-1):
        for j in range(i+1, len(l)):
            if l[j] < l[i]:
                l[i], l[j] = l[j], l[i]
    return l
```

Complexitatea timp este:

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

Insertion Sort – Sortare prin inserție

```
def InsertionSort(l):
    for i in range(1, len(l)):
        ind = i - 1
        el = l[i]
        while ind >= 0 and el < l[ind]:
            l[ind + 1] = l[ind]
            ind = ind - 1
        l[ind + 1] = el
    return l
```

Complexitate:

Caz defavorabil:

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

Caz favorabil:

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

Caz mediu:

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

Complexitate generală: $O(n^2)$.

Complexitate memorie adițională: $\theta(1)$.

Insertion sort este un algoritm in-place.

Bubble Sort – Sortarea bulelor

```
def BubbleSort(l):
    sortat = False
    while not sortat:
        sortat = True
        for i in range(len(l) - 1):
            if l[i + 1] < l[i]:
                l[i], l[i + 1] = l[i + 1], l[i]
                sortat = False
    return l
```

```
def BubbleSort(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]
    return l
```

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*** .

Quick Sort – Sortare Rapidă

```
def quickSortRec(l,left,right):
    pos = partition(l, left, right)
    if left<pos-1:
        quickSortRec(l, left, pos-1)
    if pos+1<right:
        quickSortRec(l, pos+1, right)
    return l
```

Caz favorabil: $\theta(n\log_2 n)$.

Caz defavorabil: $\theta(n^2)$.

Complexitate caz mediu: $\theta(n\log_2 n)$.

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

Quick Sort cu list comprehensions:

```
def QuickSort(lista):
    if len(lista) <= 1:
        return lista
    pivot = lista.pop()
    mai_mic = QuickSort([x for x in lista if x < pivot])
    mai_mare = QuickSort([x for x in lista if x >= pivot])
    return mai_mic + [pivot] + mai_mare
```

Merge Sort – Sortare prin interclasare

```
def MergeSort(l):
    if(len(l)) > 1:
        mij = len(l) // 2
        st = l[:mij]
        dr = l[mij:]

        #Apel recursiv pentru fiecare parte a listei
        MergeSort(st)
        MergeSort(dr)

        i, j, k = 0, 0, 0

        while i < len(st) and j < len(dr):
            if st[i] <= dr[j]:
                l[k] = st[i]
                i += 1
            else:
                l[k] = dr[j]
                j += 1
            k += 1

        while i < len(st):
            l[k] = st[i]
            i += 1
            k += 1

        while j < len(dr):
            l[k] = dr[j]
            j += 1
            k += 1

    return l
```

Complexitate de timp medie: $\theta(n \log_2 n)$.

Complexitate de memorie adițională: $\theta(n)$.

Sortare	Caz defavorabil	Caz mediu	Caz favorabil	Complexitate memorie
Selection Sort	$\theta(n^2)$	$\theta(n^2)$	$\theta(n^2)$	$\theta(1)$
Direct Selection Sort	$\theta(n^2)$	$\theta(n^2)$	$\theta(n^2)$	$\theta(1)$
Insertion Sort	$\theta(n^2)$	$\theta(n^2)$	$\theta(n^2)$	$\theta(1)$
Bubble Sort	$\theta(n^2)$	$\theta(n^2)$	$\theta(n)$	$\theta(1)$
Quick Sort	$\theta(n^2)$	$\theta(n \log_2 n)$	$\theta(n \log_2 n)$	$\theta(\log_2 n)$
Merge Sort	$\theta(n \log_2 n)$	$\theta(n \log_2 n)$	$\theta(n \log_2 n)$	$\theta(n)$

Divide and Conquer

```
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
```

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

Găsire maxim din listă.

```
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
```

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

Divide: calculează k/2

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

Combine: una sau doua înmulțiri

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

Ridicare la putere.

Backtracking

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()
```

```
def subsecventa_recursiv(lista, n, x, poz, solutii):
    """
    Conditie de baza: Verifica daca lungimea listei este egala
    cu pozitia ultimului element, atunci goleste lista
    initiala pentru a putea continua
    """


    for i in range(poz, len(lista)):
        x.append(0)
        x[-1] = lista[i]
        if este_consistenta(x, lista):
            if este_solutie(x, n) and x not in solutii:
                solutie = copy.deepcopy(x)
                solutii.append(solutie)
            subsecventa_recursiv(lista, n, x, i + 1, solutii)
    x.clear()
```

```
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:

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

condiție consistent:

$$x=(x_0, x_1, \ldots, x_k) \textit{ e consistent dacă } x_i \neq x_j \textit{ pentru} \forall\, i \neq j$$

condiție soluție:

$$x=(x_0, x_1, \ldots, x_k) \textit{ e soluție dacă e consistent şi } k=N-1$$

Metoda Greedy

```
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
```

Programare Dinamică:

```
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)
```

Lungimea celei mai lungi subliste crescatoare.

```
def fiboDP(n):
    """
    DP(1) = DP(2) = 1
    DP(i) = DP(i-1) + DP(i-2)
    """
    if n <= 2:
        return 1
    dict = [0]*(n)
    dict[0] = 1
    dict[1] = 1
    for i in range(2,n):
        dict[i] = dict[i-1] + dict[i-2]
    return dict[n-1]
```

```python
def longestSublist(a):
    """
    Determina cea mai lunga sublista crescatoare
    a - lista de elemente
    """
    #initializam l si p
    l = [0] * len(a)
    p = [0] * len(a)
    l[len(a)-1] = 1
    p[len(a)-1] = -1
    for k in range(len(a)-2, -1, -1):
        p[k] = -1
        l[k] = 1
        for i in range(k+1, len(a)):
            if a[i] >= a[k] and l[k] < l[i]+1:
                l[k] = l[i]+1
                p[k] = i
    #identificam cea mai lunga sublista
    #cautam lungimea maxima
    j = 0
    for i in range(0, len(a)):
        if l[i] > l[j]:
            j = i
    #colectam rezultatele folosind lista de pozitii p
    rez = []
    while j != -1:
        rez = rez + [a[j]]
        j = p[j]
    return rez
```