

# Fundamentele programării

## Algoritmi de căutare

- \* datele sunt în memorie, o secvență de înregistrări  $(k_1, k_2, \dots, k_n)$
- \* se cauta o înregistrare având un camp egal cu o valoare dată
- \* dacă găsim înregistrarea, returnăm poziția înregistrării în secvență

## Specificații de căutare:

Datu:  $a, m, (k_i, i=0, m-1)$ ;

Precondiții:  $m \in \mathbb{N}, m \geq 0$

Rezultat:  $p$ :

P+ condiții:  $(0 \leq p \leq m-1 \text{ and } a = k_p) \text{ sau } (p=-1 \text{ dacă cheia nu există})$

## 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 += 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  
 $T(n) = (1 + 2 + \dots + n - 1)/n \in \Theta(n)$   
 Overall complexity  $\Theta(n)$

## Specificații pentru căutare - chei ordonate

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

Precondiții :  $m \in \mathbb{N}, m \geq 0$  and  $k_0 < k_1 < \dots < k_{m-1}$ ;

Rezultat p:

Post-condiții:  $(p=0 \text{ and } a \leq k_0) \text{ or } (p=m \text{ and } a > k_{m-1}) \text{ or }$   
 $((0 < p \leq m-1) \text{ and } (k_{p-1} < a \leq k_p))$

## 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
```

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

```
def searchSucc(el, l):
    """
    Search for an element in a list
    el - element
    l - list of 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

$$T(n) = (1 + 2 + \dots + n - 1)/n \in \Theta(n)$$

Overall complexity  $\Theta(n)$

## 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))
```

Condiția de oprire:  
left >= right-1 : intervalul  
nu mai poate fi împărțit  
în două părți.

## Recurvență căutare binară

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

## Căutare binară (iterativă)

```
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
```

## Căutare în python - index()

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

$l.index(11)$  încercă să găsească poziția numărului 11 în lista  $l$ . Dacă 11 nu se află în intervalul de la 1 la 9, acestă căutare produce o valoare de tip `ValueError`.

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 ( \_\_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)
```

## Performance 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 ("    PythonIndex 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
PythonIndex 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	search 10000000
BinaryRec in 0.000000 sec; poz=10000000	BinaryRec in 0.000000 sec; poz=10000000
PythIndex in 0.234000 sec; poz=10000000	PythIndex in 0.234000 sec; poz=10000000
PythonIn in 0.238000 sec;	PythonIn in 0.238000 sec;
BinaryNon in 0.000000 sec; poz=10000000	BinaryNon in 0.000000 sec; poz=10000000
searchSuc in 2.050000 sec; poz=10000000	searchSuc in 2.050000 sec; poz=10000000

## Sortare

Rearanjarea datelor dintr-o colecție încât o cheie verifică o relație de ordine dată.

\* internal sorting - datele sunt în memorie

\* external sorting - datele sunt în fisier

Elementele unei colecții sunt înregistrări, o înregistrare are una sau mai multe câmpuri

Există o cheie  $K$  asociată fiecărui înregistrare după care se face sortarea, în general este un câmp

Colecția este sortată:

\* crescător după cheia  $K$ : if  $K(i) \leq K(j)$  for  $0 \leq i < j < m$

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

Sortare prin selecție

\* se determină elementul având cea mai mică cheie, interschimbă elementul cu elementul de pe prima poziție

\* repetă procedura pentru 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

$$\sum_{i=1}^{m-1} \sum_{j=i+1}^m 1 = \sum_{i=1}^{m-1} m-i = \sum_{i=1}^{m-1} m - \sum_{i=1}^{m-1} i = m(m-1) - \frac{m(m-1)}{2} = \frac{m(m-1)}{2} \in \Theta(m^2)$$

$\overbrace{\hspace{10em}}^{1+2+\dots+m-1}$

$\underbrace{\hspace{5em}}_{m-1 \text{ ori}}$

Ește independent de datele de intrare:

- \* caz favorabil / mediu / defavorabil sunt la fel, complexitatea este  $\Theta(m^2)$

## Complexitate - spatiu de memorie

Sorțare prin selecție este un algoritm In-place:

memoria aditională (altă memorie necesară pentru datele de intrare) este  $\Theta(1)$

- \* In-place Algoritmul care nu folosește memorie aditională (doar un mic factor constant)

\* Out-of-place sun not-in-space. Algoritmul folosește memorie aditională pentru sortare.

## Sorțare prim selectie directă

```
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]
```

## Sorțare prim inserție - Insertion Sort

- \* se parcurge elementele
- \* se insurăza elementul curent pe poziția corectă în subsecvența deja sortată
- \* în subsecvența ce conține elementele deja sortate se lipesc elementele sortate pe tot parcursul algoritmului, astfel după ce parcugem toate elementele secvența este sortată.

## Sorțare prim 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
```

## Complexitate - timp de execuție

$$\text{Cazul dezfavorabil: } T(n) = \sum_{i=2}^n (i-1) = \frac{n(n-1)}{2} \in \Theta(n^2)$$

Caz mediu: Pentru un  $i$  fixat și un  $k$ ,  $1 \leq k \leq i$ , probabilitatea ca  $x_i$  să fie al  $k$ -lea cel mai mare element în secvență  $x_1, x_2, \dots, x_i$  este  $\frac{1}{i}$ .

Astfel pentru un  $i$  fixat putem deduce:

Numărul de iterări while	Probabilitatea sa avem numărul de iterări 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 $x_1 \leq x_i < x_2$

||

numărul de iterații while medii pentru un  $i$  fixat este:

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

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

Sortare prin inserție

\* complexitate generală este  $O(n^2)$

Complexitate spațială de memorie

complexitate memorie aditională este:  $\Theta(1)$

\* sortare prin inserție este un algoritm in-place