

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, \dots, 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: $n \in \mathbb{N}, n \geq 0;$

Rezultate: $p;$

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

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=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 $O(n)$

Specificații pentru căutare – chei ordonate:

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

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

Rezultate p ;

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

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 $O(n)$

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))
```


Recurența căutare binară

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

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 cautări

[illegible]

```
# analyzed list, % middle
```

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)
```

Performanță - 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 colecție astfel încât o cheie verifică o relație de ordine dată

- › *internal sorting* - datele sunt în memorie
- › *external sorting* - datele sunt în fișier

Elementele colecției sunt *înregistrări*, o înregistrare are una sau mai multe câmpuri

Cheia K este asociată cu fiecare înregistrare, în general este un câmp.

Colecția 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 ; $\{K=(k_1, k_2, \dots, k_n)\}$

Precondiții: $k_i \in R, i=1, n$

Rezultate K' ;

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

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

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* . Algoritm care nu folosește memorie adițională (doar un mic factor constant).
- › *Out-of-place* sau *not-in-space*. Algoritm care folosește memorie adițională pentru sortare.

Selection sort is an *in-place* sorting algorithm.

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]:  
                swap(l, i, j)
```

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 , $1 \leq k \leq i$, probabilitatea ca x_i să fie al k -lea cel mai mare element în subsecvența x_1, x_2, \dots, x_i este $\frac{1}{i}$

Astfel, pentru i fixat, putem deduce:

Numarul 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:

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

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 meorie

complexitate memorie aditională 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