Curs 11: 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 10-11: Căutări - sortări

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

Sortare prin selecție 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]:
                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
```

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(1,len(l)):
            if l[i-1]>l[i]:
                swap(l, i, i-1)
                sorted = False  #the list is not sorted yet
```

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

Bazat pe “divide and conquer”

- **Divide**: se partiționează 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)

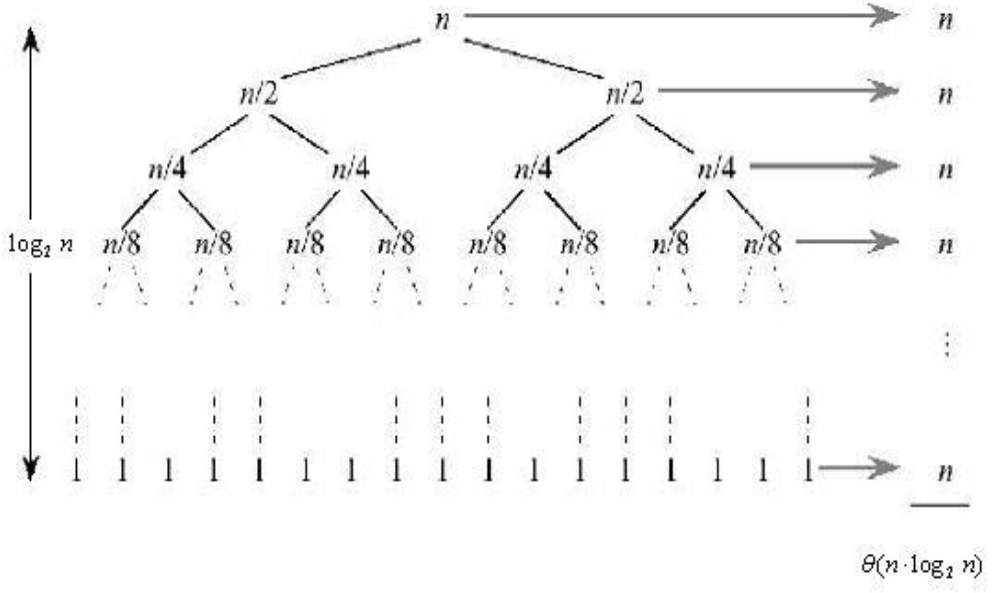
Partiționarea necesită timp linear.

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

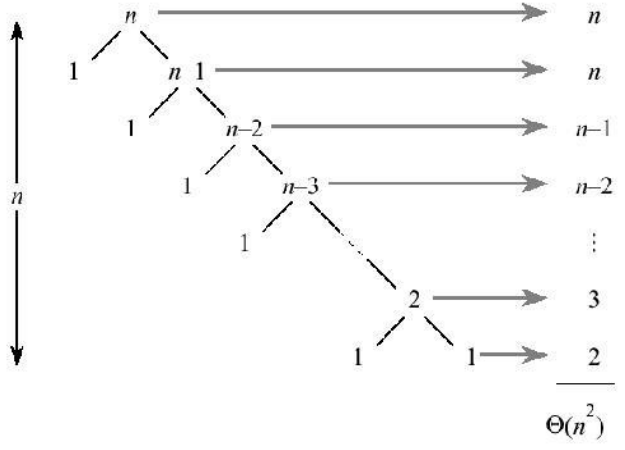
QuickSort – Caz favorabil



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 qsort(list):
    """
    Quicksort using list comprehensions

    """
    if list == []:
        return []
    else:
        pivot = list[0]
        lesser = qsort([x for x in list[1:] if x < pivot])
        greater = qsort([x for x in list[1:] if x >= pivot])
        return lesser + [pivot] + greater
```

List comprehensions – generatoare de liste

```
[x for x in list[1:] if x < pivot]
```

```
rez = []
for x in l[1:]:
    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)
```

TreeSort

Algoritmul creează un arbore binar cu proprietatea că la orice nod din arbore, arborele stâng conține doar elemente mai mici decât elementul din rădăcină iar arborele drept conține doar elemente mai mari decât rădăcina.

Dacă parcurgem arborele putem lua elementele în ordine crescătoare/descrescătoare.

Arborele este construit incremental prin inserarea succesivă de elemente. Elementele se inserează astfel încât se menține proprietatea ca în stânga avem doar elemente mai mici în dreapta doar elemente mai mari decât elementul din rădăcină.

Elementul nou inserat tot timpul ajunge într-un nod terminal (frunză) în arbore.

MergeSort

Bazat pe “divide and conquer”.

Secvența este împărțită în două subsecvențe și fiecare subsecvența este sortată. Dupa 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).

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: $\theta(m+n)$.

Spațiu de memorare adițională pentru merge sort $\theta(1)$

Curs 11: 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 12 – Technici de programare

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