Fundamentele programării

Metoda bulelor

* Compară elemente consecutiv, dacă nu sunt în ordinea dorită le interschimbă
* Procesul de comparare continuă până când nu mai avem elemente consecutive ce trebuie interschimbate.

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: $O(n^2)$

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

* este un algoritm de sortare in-place

QuickSort

Bazat pe "divide and conquer"

* Divide: se împarte lista în 2 astfel încât elementele din dreapta pivotului sunt mai mari decât elementele din stânga pivotului
* Conquer: se sortează cele două subliste
* Combine: partiționarea se face-n aceeași listă

Partiționare: rearanjarea elementelor astfel încât elementul numit pivot ocupă locul final în secvență. Dacă poz pivotului este i:

$k_j \le k_i \le k_l$, for $Left \le j \le i \le l \le Right$

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

Exemplu vizual:

<5 (deasupra lui 1)

[5, 2, 9, 1, 5, 6]   left = 0, right = 5

i (sub 5)   j ← j ← j

>5 (deasupra lui 9)

[1, 2, 9, 1, 5, 6]

i → i → i   j

[1, 2, 9, 9, 5, 6]

i   j

[1, 2, 9, 9, 5, 6]
i
j

ieșim din while pt că i == j

[1, 2, 5, 9, 5, 6]

QuickSort - complexitate timp

Partiționarea necesită timp liniar

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

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

$T\left(\frac{n}{2}\right) = 2T\left(\frac{n}{4}\right) + n$

$T\left(\frac{n}{4}\right) = 2T\left(\frac{n}{8}\right) + n$

...

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

$= 2\left(2T\left(\frac{n}{4}\right) + \frac{n}{2}\right) + n$

$= 2^2 T\left(\frac{n}{4}\right) + n + n$

$= 2^2\left(2T\left(\frac{n}{8}\right) + \frac{n}{4}\right) + n + n$

$= 2^3 T\left(\frac{n}{8}\right) + n + n + n$

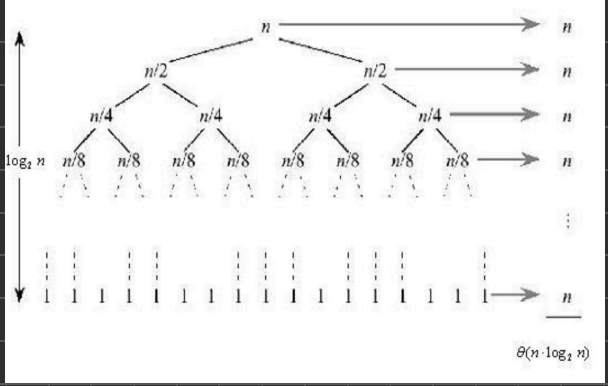
$\Downarrow$

$T(k) = 2^k T\left(\frac{n}{2^k}\right) + K \cdot n$

pentru $n = 2^K \Rightarrow K = \log n$

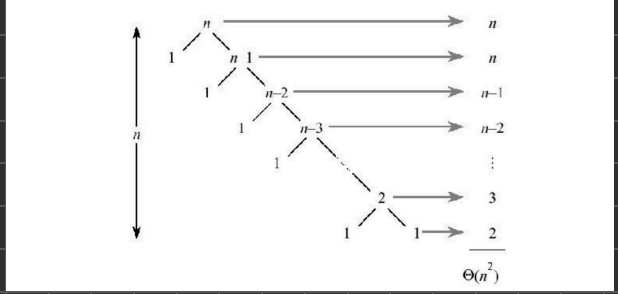
$T(n) = n\,T(1) + K \cdot n = n\,T(1) + n \log n \Rightarrow$

$\Rightarrow$ complexitatea este $\Theta(n \log n)$



Caz defavorabil: partiționarea tot timpul rezultă într-o partiție cu un singur element și o partiție cu $n-1$ elemente, adică dacă elementele sunt în ordine inversă

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



Caz mediu: Se alternează cazurile.

* BC : $\theta(n \cdot \log n) \overset{\text{n.o.t.}}{=\!=} B$

* WC : $\theta(n^2) \overset{\text{n.o.t.}}{=\!=} W$

Avem recurența:

$$\begin{cases} B(n) = 2 \cdot W\left(\frac{n}{2}\right) + n \\ W(n) = B(n-1) + n \end{cases} \Rightarrow$$

$$\Rightarrow B(n) = 2 \cdot \left(B\left(\frac{n}{2} - 1\right) + \frac{n}{2} \right) + n = 2 \cdot B\left(\frac{n}{2} - 1\right) + n = \Theta(n \cdot \log n)$$

Python - QuickSort

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
```

la fiecare x din lista originală includ în lista nouă doar elementele pentru care condiția x < pivot se respectă

List comprehensions - generatoare de liste

[x for x in list if x < pivot]

echivalent cu

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

- variantă concisă de a crea liste
- creează lista unde elementele listei rezultă din operații asupra unor elemente dintr-o altă secvență

• parantezare dreptă conținând o expresie 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)

Tipuri de parametrii:

positional-or-keyboard: parametrul poate fi transmis prin poz sau prin nume:

```
def func(a, b=None):
```

Apel prin poziție:

func(10, 20) ✓

Apel prin nume:

func(a=10, b=20) ✓

Keyword-only: parametrul poate fi transmis doar specificând numele

def func(a, *, b, c):

Tot ce apare după * poate fi transmis doar prin nume

Apel corect:

func(10, b=20, c=30) ✓

Apel greșit:

func(10, 20, 30) ✗

var-positional: se poate transmite arbitrar un număr arbitrar de parametrii poziționali

def func(*args):

    print(args)

Valorile se pot accesa folosind args care este un tuplu

Apel:

```
func(1, 2, 3, 4, 5)
# output: (1, 2, 3, 4, 5)
```

var - keyword: se poate transmite arbitrar un număr de argumente keyword (nume = valoare) care sunt colectate într-un dicționar

```
def func(**args)
    print(args)
```

Apel:

```
func(name = "Ion", age = 25, city = "București")
# output: {'name': 'Ion', 'age': 25, 'city': 'București'}
```

Sortare în python

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

Sortează folosind operatorul <. Nu creează altă listă, o sortează pe cea curentă.

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

reverse - True dacă vrem să sortăm descrescător

Python - funcții lambda

* Folosind lambda putem crea mici funcții

lambda x : x + 7

* Funcțiile lambda pot fi folosite oriunde:

```
def f(x):
    return x + 7
print(f(5))
```

⟶ `print((lambda x : x + 7)(5))`

* putem crea doar o expresie
* funcțiile lambda pot referi variabile din namespace, ex.

```
l = sorted(l, key = lambda x : x.nume, reverse = True)
```

Merge Sort

Secvența este împărțită în două subsecvențe egale și fiecare subsecvență este sortată. După sortare se intercalează cele două subsecvențe, astfel rezultă secvența sortată.
Pentru subsecvențe se aplică aceeași abordare până când ajungem la o subsecvență dintr-un singur element.

Python - MergeSort

```python
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))

, unde merge este

```python
def merge(l, start, end, m):
    # Creează două subliste: stânga și dreapta
    left = l[start:m]
    right = l[m:end]

    i = j = 0  # Indicii pentru subliste
    k = start  # Indicele pentru lista inițială

    # Compară elementele din cele două subliste și le adaugă în ordine în lista inițială
    while i < len(left) and j < len(right):
        if left[i] <= right[j]:
            l[k] = left[i]
            i += 1
        else:
            l[k] = right[j]
            j += 1
        k += 1

    # Adaugă elementele rămase din sublista stângă (dacă există)
    while i < len(left):
        l[k] = left[i]
        i += 1
        k += 1

    # Adaugă elementele rămase din sublista dreaptă (dacă există)
    while j < len(right):
        l[k] = right[j]
        j += 1
        k += 1
```

Complexitate sortare interclasare

$BC = AC = WC$

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

Complexitate: $\Theta(n \log n)$

Spațiu de memorare adițională: $\Theta(n)$