Tutoriat 8

Complexități

!Obs: funcția sort() are complexitate timp O(nlogn).

Analiza complexității unui algoritm are ca scop estimarea volumului de resurse de calcul necesare pentru execuția algoritmului.

Prin resurse se înțelege:

- Spat¸iul de memorie necesar pentru stocarea datelor pe care le prelucrează algoritmul.
- Timpul necesar pentru execuția tuturor prelucrărilor specificate în algoritm.

Vom considera că :

Prelucrările se efectuează în mod secvențial.

Operațiile elementare sunt efectuate în timp constant.

Notăm T(n) timpul de execuție al unei probleme de dimensiune n.

Exemple complexități:

```
l = [int(x) for x in input().split()] # O(n)

print(max(l)) #O(n)

l.sort()  # O(nlogn)

print(l) # O(n)

#Complexitate totala O(3n + nlogn ) =O(nlogn)
```

Tehnici de programare

Divide et Impera

Divide et impera se bazează pe principiul descompunerii problemei în două sau mai multe subprobleme (mai ușoare), care se rezolvă, iar soluția pentru problema inițială se obține combinând soluțiile subproblemelor.

Un algoritm prin divide et impera se elaborează astfel: la un anumit nivel avem două posibilități:

1.s-a ajuns la o problemă care admite o rezolvare imediată (condiția de terminare), caz în care se rezolvă și se revine din apel;

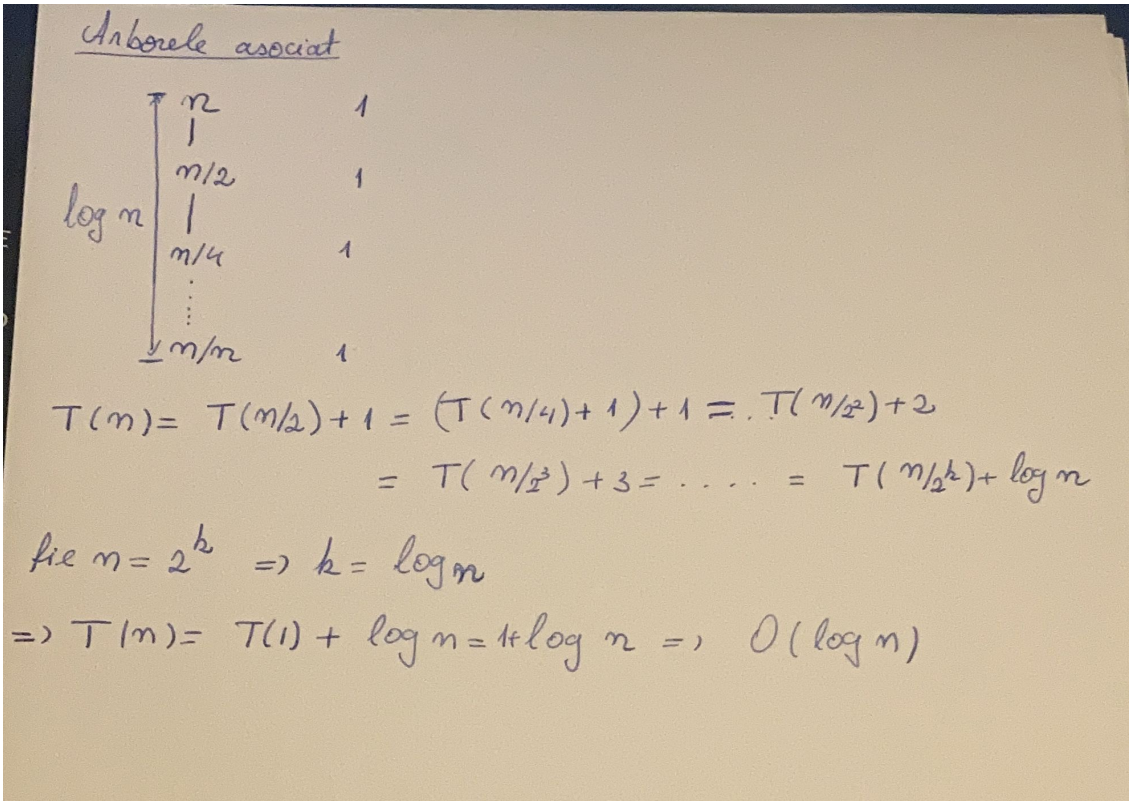
2.nu s-a ajuns în situația de la punctul 1, caz în care problema curentă este descompusă în (două sau mai multe) subprobleme, pentru fiecare din ele urmează un apel recursiv al funcției, după care combinarea rezultatelor are loc fie pentru fiecare subproblemă, fie la final, înaintea revenirii din apel.

Ex: Cautarea Binara.

```
def CB(s, d, x):
    m=s+d//2
    if v[m]==x:
        return m

    if s<d:
        if nr<v[m]:
            return CB(s,m-1)
        else:
            return CB(m+1,d)
    return None
```

Calculul complexitatii:



La fiecare pas se realizează o operație O(1) (ori return m ori return CB()), iar înălțimea arborelui este log(n) => complexitatea este O(log n)

Backtracking

Este o tehnica de programare, folosită pentru determinarea tuturor soluțiilor unei probleme.

Sablon BT:
!Orice model de sablon Backtracking invatat in liceu este bun!

#x este vectorul soluție

def back(k): #funcția primește ca parametru indicele

    if k==n+1: #dacă soluția are numărul de componente dorite, în cazul nostru n

      if conditii_finale(<parametri>): #condiții suplimentare pentru validarea soluției(poat fi
                                       #omise)
            print(*x)
    else:

       for i in range(1,n+1): #dăm valoare lui x[k]
          x[k]=i
          if validare(<parametrii>): validarea elementului x[k]
             back(k+1)   #trecem la elementul următor din soluție

Exemple:

1.Generarea mulțimii {(a,b,c,d) |a,b,c,d ∈ {0,1,...,n}}

```
n=int(input())
x=[0] *(n)

def back(k): #funcția primește ca parametru indicele

    if k==n: #dacă soluția are numărul de componente dorite, în cazul
nostru n
        print(*x)
    else:

        for i in range(1,n+1):  #dăm valoare lui x[k]
            x[k]=i
            if True:
```

```
                back(k+1)    #trecem la elementul următor din soluție
back(0)
```

2.Permutările mulțimii {1,2,..,n}

```
n=int(input())
x=[0] *(n)

def back(k): #funcția primește ca parametru indicele

    if k==n: #dacă soluția are numărul de componente dorite, în cazul
nostru n
        print(*x)
    else:

        for i in range(1,n+1):  #dăm valoare lui x[k]
            x[k]=i
            if x[k] not in x[:k] :
                back(k+1)    #trecem la elementul următor din soluție

back(0)
```