

The background is a solid teal color. It is decorated with several stylized autumn-themed elements: a large red leaf with wavy veins in the top left, a light blue leaf with thin blue outlines in the top left, a single orange leaf in the top right, a small white flower with pink leaves in the bottom left, a small white spiral in the top right, and a large yellow lemon with black seeds in the bottom right.

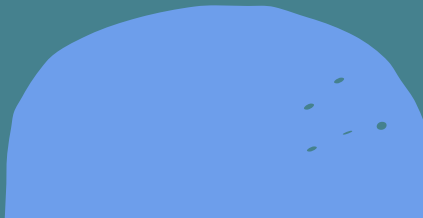
Algoritmi de sortare

Negruț Maria-Daniela

Grupa 133



Algoritmi:

- Bubble Sort - $O(n^2)$
 - Counting Sort - $O(n + k)$
 - Merge Sort - $O(n \log n)$
 - Natural merge Sort - $O(n \log n)$
 - Radix Sort base 10 - $O(n \lg k)$
 - Radix Sort base 2 - $O(n \log_2 k)$
 - Quick Sort – pivot mediana medianelor din 5 - $O(n \log n)$
 - Quick Sort – pivot random - $O(n \log n)$
 - Quick Sort – pivot capăt dreapta - $O(n \log n)$
- 

Bubble Sort

Principiu:

Prin comparații successive între vecini duce la fiecare iterație cel mai mare număr current la locul lui în vectorul sortat.

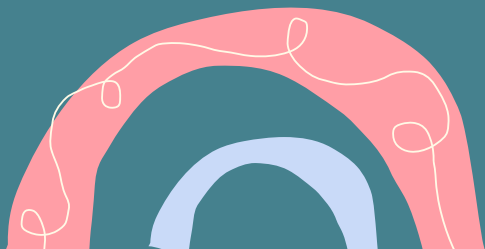


6 5 3 1 8 7 2 4

Îmbunătățire:

Știm la fiecare iterație că ultimele k elemente sunt sortate -> nu mai este nevoie să le comparăm

Dacă la o parcurgere întreagă a vectorului nu se realizează nicio interschimbare între elemente -> deja sortat ne putem opri



Complexități:

Best: $O(n)$

Recunoaște dacă vectorul e sortat de la bun început

Worst: $O(n^2)$

Extrem de lent!!!

Singurul beneficiu este simplitatea codului și ușurința de înțelegere pentru începători

```

def Bubble(N, Max, L): #  $O(n^2)$ 
    # N =  $10^3$  - aprox 0.1 max
    # N =  $10^4$  - aprox 10 max
    # N =  $10^5$  - // too much
    """ Pentru fiecare iteratie a for-ului parinte stim ca ultimele i elemente sunt deja sortate
        din modul de functionare a algoritmului (vezi powerpoint) asa ca pt fiecare iteratie
        a for-ului copil putem sa scapam de i comparatii
        Daca pentru o iteratie a for-ului copil niciun element nu isi schimba locul sortarea
        poate fi considerata terminata deoarece inseamna ca sortarea a fost terminata anterior"""

    for i in range(N):
        did_i_swap = False
        for j in range(N - i - 1): # pt ca accesam ultimul element oricum prin j+1
            if L[j] > L[j + 1]:
                L[j + 1], L[j] = L[j], L[j + 1]
                did_i_swap = True
        if not did_i_swap:
            break
    return L

```

Îmbunătățiri:

Pentru a nu ocupa memorie inutilă determinăm minimul și maximul din listă și index aparițiile numerelor din acest interval

Pentru a le putea indexa într-o listă de lungimea intervalului fiecare număr va fi reprezentat de $nr - \text{minim}$

Această variantă nu este stabilă (pt. stabilitate trebuie ca lista de apariții să devină un bucket - listă de liste în Python)

Complexități:

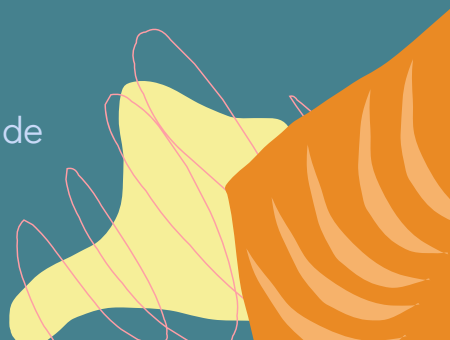
Worst = Best = $O(n + k)$

Space : $O(n + k)$ -> ineficient pentru probleme cu valori de dimensiuni mari

Counting Sort

Principiu:

Pentru fiecare număr se reține numărul de apariții și se pun la final în listă în ordine crescătoare de câte ori au apărut inițial




```
def Count(N, Max, L): # O(n+k)
```

```
    """
```

```
    Optimizam memoria prin a declara vectorul counter doar pt nr din intervalul minim, maxim  
    in loc de toate numere de la 0 la Max
```

```
    Algoritmul este problematic pt int cu aprox. Max >= 4*10^6 -> prea multa memorie ocupata
```

```
    """
```

```
    minim = min(L) # O(n) - python documentation
```

```
    maxim = max(L) # O(n) - python documentation
```

```
    counter = [0] * (maxim - minim + 1) # pt a salva memorie, nu avem nevoie de vector pt elemente
```

```
    for i in range(N):
```

```
        counter[L[i] - minim] += 1
```

```
    index = 0
```

```
    for nr in range(minim, (maxim + 1)):
```

```
        for j in range(counter[nr - minim]):
```

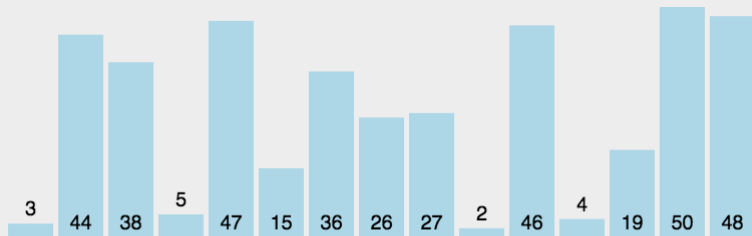
```
            L[index] = nr
```

```
            index = index + 1
```

```
    return L
```

Principiu:

Algoritm pe bază de comparații și principiul “divide et impera”. Se divide lista inițială în subprobleme de dimensiune 1 și se interclasează fiecare subproblemă până ajungem la lista finală sortată.




Merge Sort

Complexități:

Worst = Best : $O(n \log n)$

Lista se divide indiferent și se interclasează liniar
Nu recunoaște dacă lista este deja sortată

Space: $O(n)$ – listă auxiliară pentru interclasare



```
def Merge(N, Max, L): # O(nlogn)
    def merge(t, st, mij, dr): # O(n) n = nr de elemente intre st si dr
        # interclasare din acelasi vector doar ca pointeri capete diferiti
        i = st
        j = mij + 1
        aux = []
        while i <= mij and j <= dr:
            if t[i] <= t[j]:
                aux.append(t[i])
                i += 1
            else:
                aux.append(t[j])
                j += 1
        aux.extend(t[i:mij + 1])
        aux.extend(t[j:dr + 1])
        t[st:dr + 1] = aux[:]
```

```
def Merge(N, Max, L): # O(nlogn)
    def merge(t, st, mij, dr):...

    def sort(t, st, dr):
        if dr - st >= 1:
            mij = (dr + st) // 2
            sort(t, st, mij)
            sort(t, mij + 1, dr)
            if t[mij] > t[mij + 1]: # pt a evita interclasari inutile
                # daca ultimul el din jum stanga e mai mic decat primul element din jum dreapta
                # inseamna ca ambele jumatati sunt sortate corect -> nu mai e nevoie de interclasare
                merge(t, st, mij, dr)

    sort(L, 0, N - 1)
    return L
```

Natural merge Sort

Singura diferență față de un Merge sort este faptul că nu mai spargem lista în subprobleme de dimensiune 1, ci în subsecvențe crescătoare de lungime aleatorie.

Inițial trecem prin listă și determinăm pozițiile pe care se termină fiecare subsecvență crescătoare. Interclasăm secvențele crescătoare 2 câte două (dacă avem nr. impar lăsăm o subsecvență așa) și reactualizăm pozițiile de pe care încep.

Ne oprim când mai avem doar o subsecvență crescătoare = chiar lista sortată.

Complexitatea rămâne $O(n \log n)$ doar că e puțin mai bună

```
Start      : 3  4  2  1  7  5  8  9  0  6
Select runs : (3  4)(2)(1  7)(5  8  9)(0  6)
Merge      : (2  3  4)(1  5  7  8  9)(0  6)
Merge      : (1  2  3  4  5  7  8  9)(0  6)
Merge      : (0  1  2  3  4  5  6  7  8  9)
```

```

def NaturalMerge(N, Max, L): # O(nlogn) but slightly better
    def interclasare(t, st, mij, dr):...

    output = [0] * N
    idx_grupari = [0]
    nr_grupari_sortate = 0
    for i in range(1, N + 1):
        if i == N or L[i - 1] > L[i]:
            nr_grupari_sortate += 1
            idx_grupari.append(i) # aici se termina o prima subsecv cresc

    while nr_grupari_sortate > 1:
        new = 0
        for i in range(0, nr_grupari_sortate - 1, 2):
            output[idx_grupari[i]: idx_grupari[i + 2]] = interclasare(L, idx_grupari[i], idx_grupari[i + 1], idx_grupari[i + 2])
            idx_grupari[new] = idx_grupari[i] # am combinat 2 grupari sortate -> scap de una din index
            new += 1
        if nr_grupari_sortate % 2 == 1:
            last = idx_grupari[nr_grupari_sortate - 1]
            output[last:N] = L[last:N]
            idx_grupari[new] = idx_grupari[nr_grupari_sortate - 1]
            new += 1
        # gruparea fara pereche am considerat-o "merged" cu nimic deci se poate scoate
        idx_grupari[new] = N # tot timpul ultima secv cresc se termina pe N - 1
        nr_grupari_sortate = new
        L = output

    return L

```

Radix Sort - LSD

Principiu:

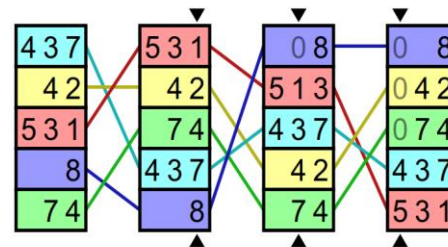
Similar cu Counting Sort sau Bucket Sort. Începând cu cea mai din dreapta cifră din baza data sortăm numerele, apoi luăm următoarea cifră și sortăm numerele fără să stricăm ordinea anterioară, până rămânem fără cifre.

Implementarea nu este stabilă – pentru stabilitate trebuie ca pt. fiecare cifre să facem bucket sort

Complexitate:

Worst = Best = $O(n \log_b \max) = O(nw)$ cu w lungimea maximului în baza b

Radix Sort (LSD)



```

def Radix(N, Max, L): # O(nlgMax)
    def countingSort(N, L, base, exp, minim):
        # punem in galeati doar cate elemente cu % base = index
        galeata = [0] * base
        output = [0] * N
        index = 0
        for i in range(N):
            index = ((L[i] - minim) // exp) % base # cifra la care ne uitam
            galeata[index] += 1
        for i in range(1, base):
            galeata[i] += galeata[i - 1]
        for i in range(N - 1, -1, -1):
            index = ((L[i] - minim) // exp) % base
            output[galeata[index] - 1] = L[i]
            galeata[index] -= 1
        return output

    base = 10
    minim = min(L) # n
    maxim = max(L) # n
    exp = 1
    while (maxim - minim) // exp >= 1:
        L = countingSort(N, L, base, exp, minim)
        exp *= base
    return L

```

```

def Radix2(N, Max, L): # O(nlog2(Max))
    def bucketSort(N, L, base, exp, minim):
        galeata = {i: [] for i in range(base)}
        output = []
        for i in range(N):
            index = (L[i] >> exp) & 1
            galeata[index].append(L[i])
        for b in range(base):
            k = len(galeata[b])
            for i in range(k):
                output.append(galeata[b][i])
        return output

    base = 2
    minim = min(L)
    exp = 0
    while (Max - minim) >> exp >= 1: ## lungimea avarage
        L = bucketSort(N, L, base, exp, minim)
        exp += 1
    return L

```

Quick Sort

Principiu:

Similar cu Merge Sort algoritmul se bazează pe “divide et impera”. În funcție de un pivot ales se partiționează vectorul pentru a pune pivotul în poziția bună. Iar funcția propriu zisă de “impera” nu mai există, toate modificările realizându-se pe vectorul dat.

Complexitate:

Best = $O(n \log n)$

Worst = $O(n^2)$

Funcționează greu pe liste cu nr mare de elemente
Însă nu alocă memorie în plus

Unsorted Array



Quick Sort

Alegerea pivotului:

- Algoritmul de mediana medianelor din 5 – complexitate $O(n)$, ocupă memorie în plus, în teorie este o idee bună deoarece partiția se realizează aproximativ egal, doar că în practică durează mult prea mult
- Random – o soluție practică bună, în general tinde să realizeze o partiție bună
- Dreapta – o soluție bună cât timp lista nu este sortată

Mediane

2	7	4	5	19
3	8	8	10	21
10	12	13	15	40
14	22	18	20	50
15	35	40	51	61



```
def BFPRT(N, A): # mediana de 5 ocupa mult spatiu
    if N <= 5:
        A.sort()
        return A[N//2]

    grupuri = [sorted(A[i: i + 5]) for i in range(0, len(A), 5)]
    mediane = [grup[len(grup) // 2] for grup in grupuri]

    return BFPRT(len(mediane), mediane)
```

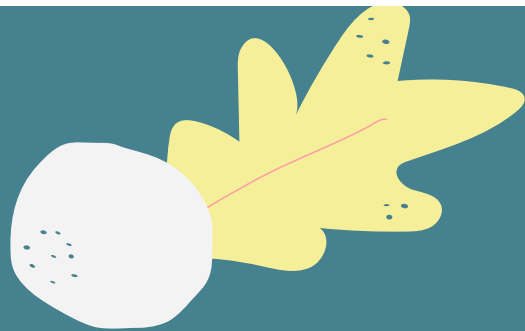
```
def partition(L, st, dr, med):
    for i in range(st, dr + 1):
        if L[i] == med:
            L[i], L[dr] = L[dr], L[i]
            break
    i = st
    for j in range(st, dr):
        if L[j] <= med:
            L[i], L[j] = L[j], L[i]
            i += 1
    L[i], L[dr] = L[dr], L[i]
    return i

def quickSort(L, st, dr):
    if st < dr:
        med = L[random.randint(st, dr)]
        q = partition(L, st, dr, med)
        quickSort(L, st, q - 1)
        quickSort(L, q + 1, dr)

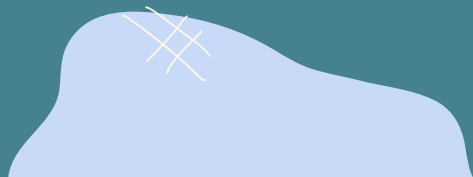
quickSort(L, 0, N - 1)
return L
```

Alegerea pivotului (poate să
fie de ex L[dr])





Rulări:



1. Lista deja sortată:

```
1 -----  
N=1000000 Max=999999 list_sorted  
Default sort in python --- Timsort O(nlogn) takes: 0.009011507034301758 s  
sorted ok Bubble 0.10393905639648438 s  
sorted ok Count 0.6001594066619873 s  
sorted ok Merge 0.49871134757995605 s  
sorted ok NaturalMerge 0.13292312622070312 s  
sorted ok Radix 3.553382396697998 s  
sorted ok Radix2 10.34955096244812 s  
sorted ok QuickMediana5 281.51155638694763 s  
sorted ok QuickRandom 5.25095796585083 s  
QuickDreapta pe list_sorted genereaza recursion depth error
```

Calculator 1

Calculator 2

```
N=1000000 Max=999999 list_sorted  
Default sort in python --- Timsort O(nlogn) takes: 0.015626192092895508 s  
sorted ok Bubble 0.03775787353515625 s  
sorted ok Count 0.34053826332092285 s  
sorted ok Merge 0.28275465965270996 s  
sorted ok NaturalMerge 0.07021975517272949 s  
sorted ok Radix 1.8350319862365723 s  
sorted ok Radix2 4.120499134063721 s  
sorted ok QuickMediana5 130.06076765060425 s  
sorted ok QuickRandom 2.7325119972229004 s  
QuickDreapta pe list_sorted genereaza recursion depth error
```

Bubble se parcurge în $O(n)$ -> recunoaște lista sortată deja
Quick se partiționează prost în funcție de pivot

2. Lista de tip munte (1 2 3 4 5 4 3 2 1):

```
2 -----  
N=2000 Max=1000 list_munte  
Default sort in python --- Timsort O(nlogn) takes: 0.0 s  
sorted ok Bubble 0.3777952194213867 s  
sorted ok Count 0.0010001659393310547 s  
sorted ok Merge 0.003996849060058594 s  
sorted ok NaturalMerge 0.003984689712524414 s  
sorted ok Radix 0.003997087478637695 s  
sorted ok Radix2 0.006996631622314453 s  
sorted ok QuickMediana5 0.009994029998779297 s  
sorted ok QuickRandom 0.0059967041015625 s  
QuickDreapta pe list_munte genereaza recursion depth error
```

Calculator 1

Calculator 2

```
2 -----  
N=2000 Max=1000 list_munte  
Default sort in python --- Timsort O(nlogn) takes: 0.0 s  
sorted ok Bubble 0.18489551544189453 s  
sorted ok Count 0.0 s  
sorted ok Merge 0.0 s  
sorted ok NaturalMerge 0.015626192092895508 s  
sorted ok Radix 0.0 s  
sorted ok Radix2 0.0 s  
sorted ok QuickMediana5 0.0 s  
sorted ok QuickRandom 0.0 s  
QuickDreapta pe list_munte genereaza recursion depth error
```

Chiar dacă dimensiunea listei este mica Bubble durează mult mai mult
Quick cu pivotul în dreapta nu partiționează bine

3. Lista cu un singur element de mai multe ori:

```
N=100000 Max=2862352 list_OneElementMultipleTimes
Default sort in python --- Timsort O(nlogn) takes: 0.0010104179382324219 s
sorted ok Bubble 0.009994029998779297 s
sorted ok Count 0.024971961975097656 s
sorted ok Merge 0.049970149993896484 s
sorted ok NaturalMerge 0.012979269027709961 s
sorted ok Radix 0.0039975643157958984 s
sorted ok Radix2 0.0009999275207519531 s
QuickMediana5 pe list_OneElementMultipleTimes genereaza recursion depth error
QuickRandom pe list_OneElementMultipleTimes gener
QuickDreapta pe list_OneElementMultipleTimes gene
```

Calculator 1

Calculator 2

```
N=100000 Max=7760217 list_OneElementMultipleTimes
Default sort in python --- Timsort O(nlogn) takes: 0.0 s
sorted ok Bubble 0.009573936462402344 s
sorted ok Count 0.015022039413452148 s
sorted ok Merge 0.030023813247680664 s
sorted ok NaturalMerge 0.007019996643066406 s
sorted ok Radix 0.0 s
sorted ok Radix2 0.0 s
QuickMediana5 pe list_OneElementMultipleTimes genereaza recursion depth error
QuickRandom pe list_OneElementMultipleTimes genereaza recursion depth error
QuickDreapta pe list_OneElementMultipleTimes genereaza recursion depth error
```

Quick nu partiționează bine

4. Lista cu elementul maxim foarte mare:

```
N=1000000 Max=39999985 list_BigNumbers
```

```
Default sort in python --- Timsort O(nlogn) takes: 0.3837759494781494 s
```

```
Bubble sort merge foarte greu pentru mai mult de 10^5 numere
```

```
sorted ok Count 11.942792177200317 s
```

```
sorted ok Merge 6.632610559463501 s
```

```
sorted ok NaturalMerge 5.674736738204956 s
```

```
sorted ok Radix 6.0355384349823 s
```

```
sorted ok Radix2 16.64591145515442 s
```

```
sorted ok QuickMediana5 11.664175510406494 s
```

```
sorted ok QuickRandom 7.108902215957642 s
```

```
sorted ok QuickDreapta 6.712606906890869 s
```

Calculator 1

Calculator 2

```
N=1000000 Max=39999934 list_BigNumbers
```

```
Default sort in python --- Timsort O(nlogn) takes: 0.19641637802124023 s
```

```
Bubble sort merge foarte greu pentru mai mult de 10^5 numere
```

```
sorted ok Count 5.2258570194244385 s
```

```
sorted ok Merge 2.8915252685546875 s
```

```
sorted ok NaturalMerge 2.96197772026062 s
```

```
sorted ok Radix 2.992525577545166 s
```

```
sorted ok Radix2 8.335499048233032 s
```

```
sorted ok QuickMediana5 5.415700674057007 s
```

```
sorted ok QuickRandom 3.265488862991333 s
```

```
sorted ok QuickDreapta 3.3038783073425293 s
```

Count și Radix au timpul de rulare mare pentru numere de dimensiune mare

5. Lista cu elementul maxim foarte mic:

```
N=1000000 Max=100000 list_SmallNumbers
```

```
Default sort in python --- Timsort O(nlogn) takes: 0.3148155212402344 s
```

```
Bubble sort merge foarte greu pentru mai mult de 10^5 numere
```

```
sorted ok Count 0.28783369064331055 s
```

```
sorted ok Merge 6.080492973327637 s
```

```
sorted ok NaturalMerge 5.866495132446289 s
```

```
sorted ok Radix 4.141594886779785 s
```

```
sorted ok Radix2 10.587399244308472 s
```

```
sorted ok QuickMediana5 12.445842742919922 s
```

```
sorted ok QuickRandom 7.569639444351196 s
```

```
sorted ok QuickDreapta 6.627178192138672 s
```

Calculator 2

```
N=1000000 Max=100000 list_SmallNumbers
```

```
Default sort in python --- Timsort O(nlogn) takes: 0.1888599395751953 s
```

```
Bubble sort merge foarte greu pentru mai mult de 10^5 numere
```

```
sorted ok Count 0.15063834190368652 s
```

```
sorted ok Merge 2.851231813430786 s
```

```
sorted ok NaturalMerge 2.9718081951141357 s
```

```
sorted ok Radix 2.168450117111206 s
```

```
sorted ok Radix2 5.222826957702637 s
```

```
sorted ok QuickMediana5 6.12988018989563 s
```

```
sorted ok QuickRandom 3.373331308364868 s
```

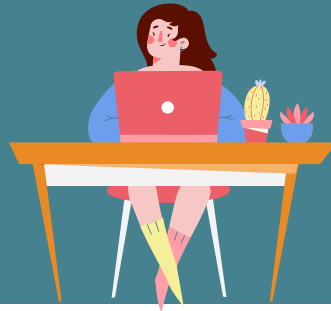
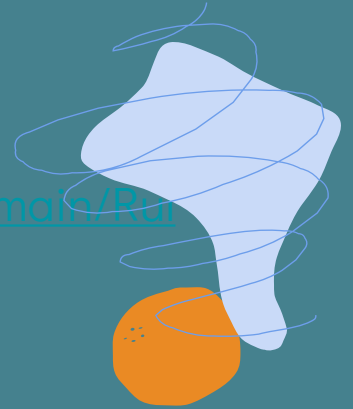
```
sorted ok QuickDreapta 2.702763795852661 s
```

Calculator 1

Pentru numere de dimensiune mica Count are un timp semnificativ mai mic decât toți ceilalți algoritmi.

Rulări pe teste generate:

https://github.com/NMDMaria/Tema_Laborator_Sortari/tree/main/Rul%C4%83ri



Concluzie:

BUBBLE SORT – foarte greu când testele nu au puține numere

- recunoaște listele deja sortate

COUNTING SORT – ocupă memorie în plus

- probleme la numere de dimensiune mare
- timp foarte bun în majoritatea cazurilor

MERGE SORT – ocupă memorie suplimentară

- timp bun în general

RADIX SORT – ocupă memorie suplimentară, probleme la numere de dimensiune mare

- timp bun în funcție de bază

QUICK SORT – instabil la liste de dimensiuni mari (in python genereaza recursion depth error)

- instabil la liste deja sortate în funcție de alegerea pivotului – dacă partiția e făcută prost (prea multe elemente mai mici ca pivotul, respective mai mari)
- Cu pivotul ales bine timpul este bun pe liste de dimensiune nu foarte mare

The background is a solid teal color. In the top left corner, there is an orange leaf with a small blue dot above it. In the bottom left corner, there is a yellow lemon with small black seeds. In the top right corner, there is a light blue abstract shape with thin blue lines. In the bottom right corner, there is a white flower with orange lines and a red stem with several red leaves.

CREDITS: This presentation template was created by Slidesgo,
including icons by Flaticon, and infographics & images by Freepik

Please keep this slide for attribution