Sortari

Bucket, Radix, Quick, Merge, Heap ?

Bucket Sort

- Elementele vectorului sunt distribuite în bucket-uri după anumite criterii
- Bucket-urile sunt reprezentate de elemente ale unui vector de liste înlănțuite
- Fiecare bucket conține elemente care îndeplinesc aceleași condiții

IDEE:

- Fie **v** vectorul de sortat și **b** vectorul de buckets
- Se inițializează vectorul auxiliar cu liste (buckets) goale
- Iterăm prin **v** și adăugăm fiecare element în bucket-ul corespunzător
- Sortam fiecare bucket (discutam cum)
- Iterăm prin fiecare bucket, de la primul la ultimul, adăugând elementele înapoi în **v**

Bucket Sort

Vizualizare:

https://www.cs.usfca.edu/~galles/visualization/BucketSort.html

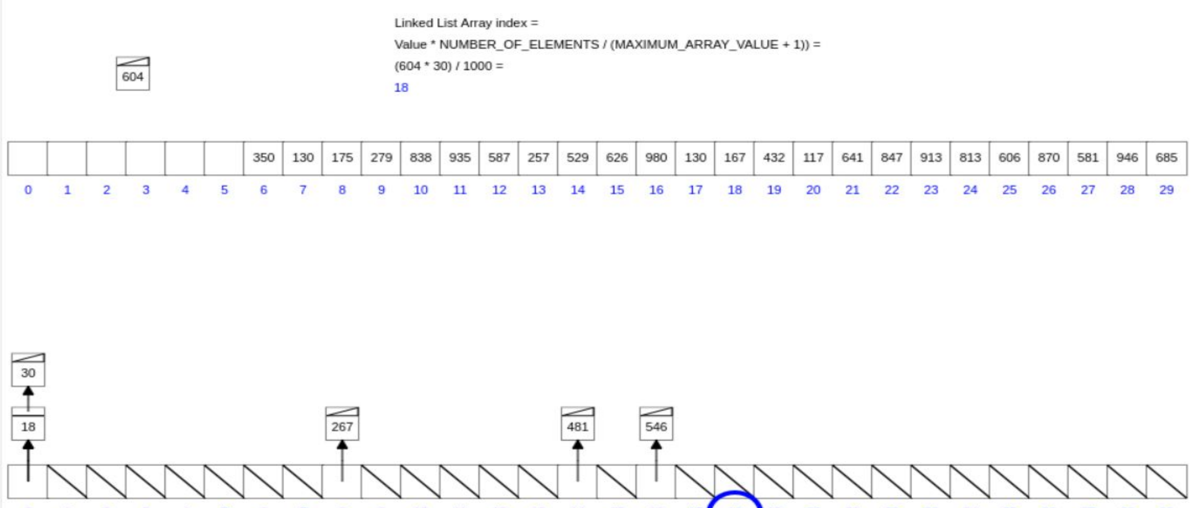
Bucket Sort

Cum adăugăm elementele în bucket-ul corespunzător?

-

Bucket Sort

Cum adăugăm elementele în bucket-ul corespunzător?

- Metoda clasică este sa împărțim la o valoare și în funcție de catul împărțirii punem valoarea în bucketul corespunzător.
- În animație foloseam 30 de bucketuri și cum numerele erau pana la 1000, inmulteam cu 30 și imparteam la 1000
- 

Bucket Sort

Cum adăugăm elementele în bucket-ul corespunzător?

- Metoda clasică este sa împărțim la o valoare și în funcție de cat sa punem în bucketul corespunzător

Cum sortam bucketurile ?

-

Bucket Sort

Cum adăugăm elementele în bucket-ul corespunzător?

- Metoda clasică este sa împărțim la o valoare și în funcție de cat sa punem în bucketul corespunzător

Cum sortam bucketurile ?

- Putem aplica recursiv tot bucketsort sau dacă avem puține elemente sa folosim o sortare simpla (insertion/selection/bubble sort) ….
    - Cum adica sa folosim bubble sort de ce nu quick sort ???

Bucket Sort

Cum adăugăm elementele în bucket-ul corespunzător?

- Metoda clasică este sa împărțim la o valoare și în funcție de cat sa punem în bucketul corespunzător

Cum sortam bucketurile ?

- Putem aplica recursiv tot bucketsort sau dacă avem putine elemente sa folosim o sortare simpla (insertion/selection/bubble sort) ….
    - Cum adica sa folosim bubble sort de ce nu quick sort ???
        - Pentru n mic constanta de la quicksort, mergesort face ca sortarea sa fie mai înceată

Bucket Sort

- Cate bucketuri ?
    -

Bucket Sort

- Cate bucketuri ?
  - Dacă sunt foarte multe initializam spațiu prea mare
  - Dacă sunt prea puține nu dispersam suficient…
    - Ce se intampla dacă toate pica în același bucket ?
  - Conteaza foarte mult și distribuția inputului.

Bucket Sort

Complexitate?

- Timp:
  -

- Spațiu:
  -

Bucket Sort

Complexitate?

- Timp:
  - Average O(n+k)
  - Worst case O(n^2)

Algoritm bun dacă avem o distribuție uniforma a numerelor...

- Spațiu:
  - O(n+k)

Radix Sort

- Este un algoritm folosit în special pentru ordonarea șirurilor de caractere
  - Pentru numere - funcționează pe aceeași idee

- Asemănător cu bucket sort - este o generalizare pentru numere mari
- Împărțim în **B** bucketuri unde **B** este baza în care vrem sa considerăm numerele(putem folosi 10 sau 100 sau 10^4 sau 2 sau 2^4, 2^6 ...)

- Presupunem că vectorul de sortat **v** conține elemente întregi, cu cifre din mulțimea {0, …, B-1}

Radix Sort

- Cum sunt utilizate bucket-urile?
  - Elementele sunt sortate după fiecare cifră, pe rând
  - Bucket Urile sunt cifrele numerelor
  - Fiecare bucket `b[i]` conține, la un pas, elementele care au cifra curentă = i

- Numărul de bucket-uri necesare?
  - Baza în care sunt scrise numerele

Radix Sort

Complexitate?

- Timp:
  - O(n log max) (discutie mai lunga)

- Spațiu:
  - O(n+b)

Radix Sort

Vizualizare:

https://visualgo.net/bn/sorting

Radix Sort - LSD

- LSD = **L**east **S**ignificant **D**igit (iterativ rapid)

Radix Sort - MSD

- MSD = **M**ost **S**ignificant **D**igit (recursiv, ca bucket sort)

Quick Sort

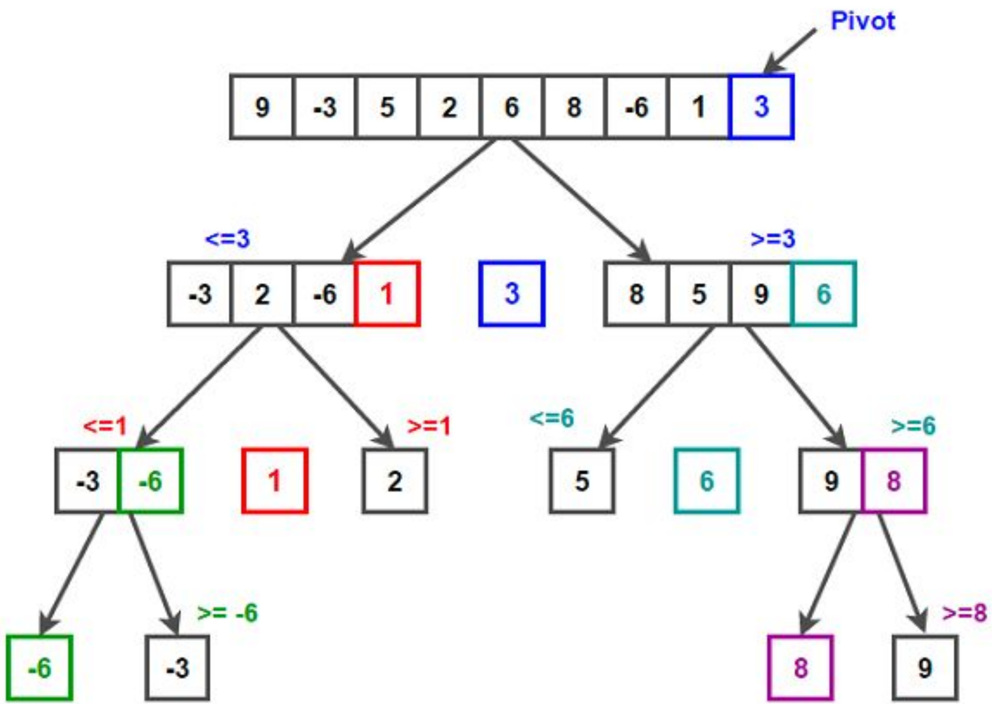
- Algoritm Divide et Impera
- Este un algoritm eficient în practica (implementarea este foarte importantă)

- **Divide:** se împarte vectorul în doi subvectori în funcție de un **pivot x** astfel încât elementele din subvectorul din stânga sunt **≤ x ≤** elementele din subvectorul din dreapta

- **Impera:** se sortează recursiv cei doi subvectori

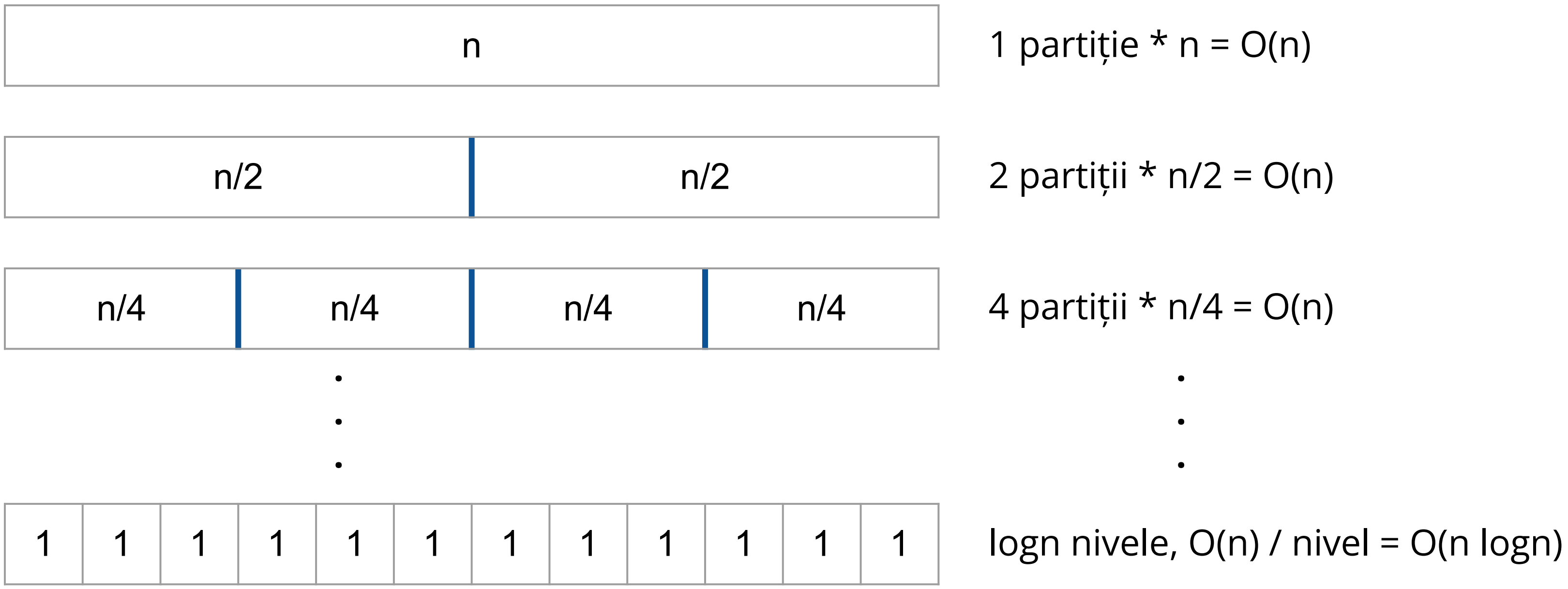
Quick sort - exemplu

- Pivot ales la coada
- Contraexemplu ?



Quick Sort

- În cel mai bun caz, pivotul x este chiar mediana, adică împarte vectorul în 2 subvectori de n/2 elemente fiecare



Quick Sort

Worst case?

- Când alegem cel mai mic sau cel mai mare element din vector la fiecare pas
- Una din cele două partiții va fi goală
- Cealaltă partiție are restul elementelor, mai puțin pivotul
- Număr de apeluri recursive?
  - n - 1
- Lungime partiție?
  - n - k (unde k = numărul apelului recursiv) → O(n - k) comparații
- Complexitate finală?
  - $O(n^2)$

Quick Sort

Cum alegem pivotul?

- Primul element
- Elementul din mijloc
- Ultimul element
- Un element random
- Mediana din 3
- Mediana din 5,7 (atenție cand vectorul devine mic, facem mult calcul pentru putin)

https://en.wikipedia.org/wiki/Quicksort#Choice_of_pivot

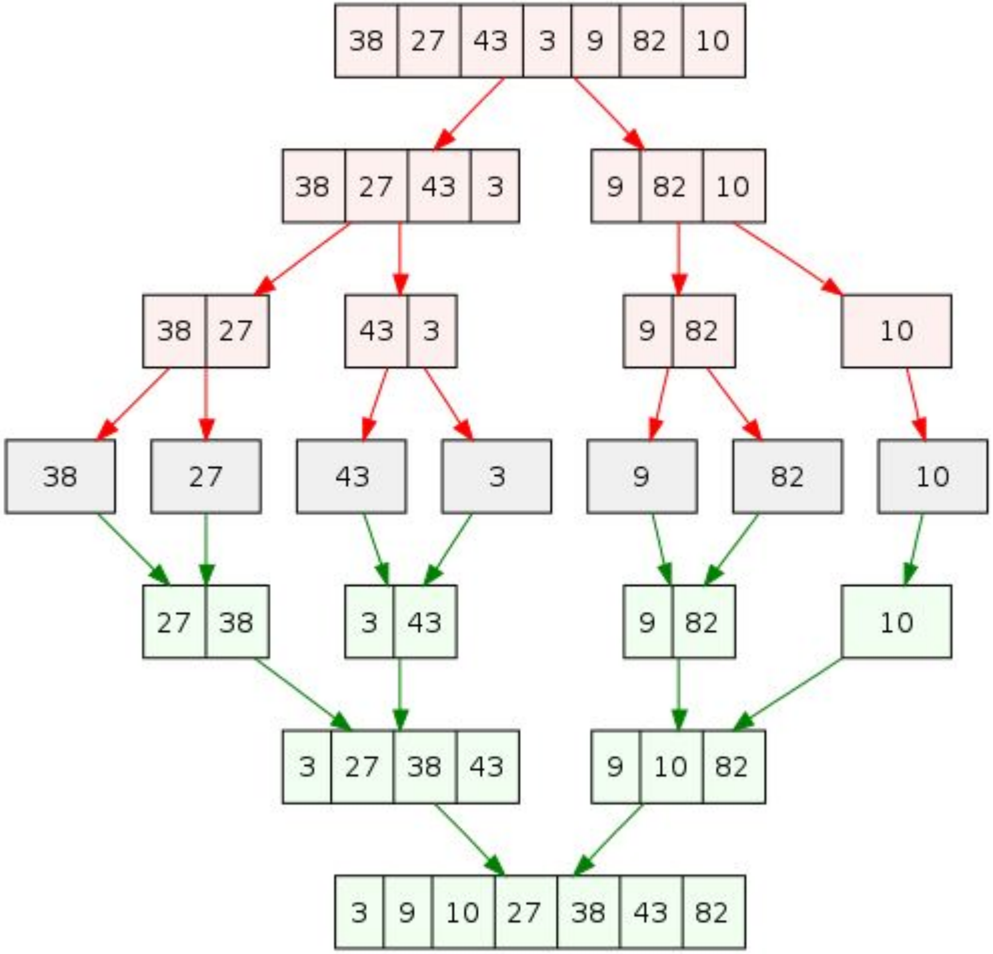
Merge Sort

- Algoritm Divide et Impera

- **Divide:** se împarte vectorul în jumătate și se sortează independent fiecare parte

- **Impera:** se sortează recursiv cei doi subvectori

Merge Sort - example



Merge Sort

- Când se oprește recursivitatea?
  - Când vectorul ajunge de lungime 1 sau 2 (depinde de implementare)
  - La fel ca și la quicksort ne-am putea opri mai repede ca sa evitam multe operatii pentru putine numere
- Algoritm de merging
  - Creem un vector temporar
  - Iterăm cele două jumătăți sortate de la stânga la dreapta
  - Copiem în vectorul temporar elementul mai mic dintre cele două

Merge Sort Vs Quick Sort

De ce e Quick Sort mai rapid în practica cand cazul ideal de la Quick Sort e ca împărțim în 2 exact ce face Merge Sortul?

- Merge Sortul are nevoie de un vector suplimentar și face multe mutari suplimentare.
- Quick Sortul e în place… memoria suplimentară e pentru stiva...

In-Place Merge Sort

- Nu folosim vector suplimentar ca în cazul Merge Sort
  - Nu este O(n logn)
  - Mai complicat
  - O alta optiune este Block Sort

Heap Sort

Vizualizare:

https://www.cs.usfca.edu/~galles/visualization/HeapSort.html

6 5 3 1 8 7 2 4

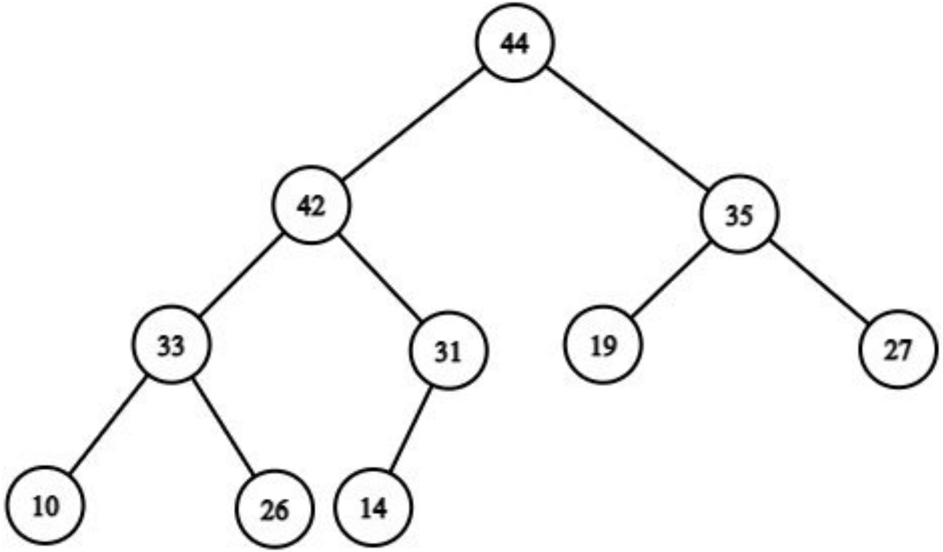
Scurtă introducere în heap-uri

- Ce este un heap?
  - Arbore binar aproape complet
  - Are înălțimea h = logn

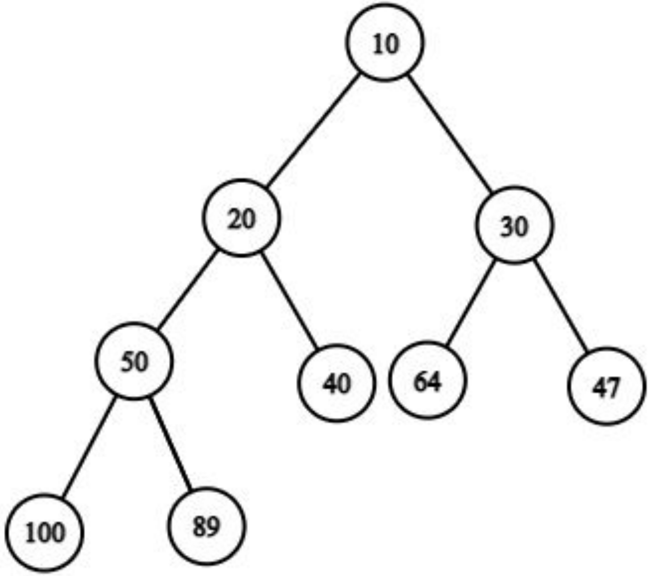
- Max-heap
  - Pentru orice nod X, fie T tatăl lui X
  - T are valoarea ≥ decât valoarea lui X
  - Elementul maxim este în rădăcină

- Min-heap
  - Pentru orice nod X, fie T tatăl lui X
  - T are valoarea ≤ decât valoarea lui X
  - Elementul minim este în rădăcină

Scurtă introducere în heap-uri



Max-heap
Ultima poziție: 14



Min-heap
Ultima poziție: 89

Heap Sort

- În funcție de sortarea dorită (ascendentă sau descendentă) - se folosește max-heap sau min-heap

IDEE:

- Elementele vectorului inițial sunt adăugate într-un heap
- La fiecare pas, este reparat heap-ul după condiția de min/max-heap
- Cât timp mai sunt elemente în heap:
  - Fie X elementul maxim
  - X este interschimbat cu cel de pe ultima poziție în heap
  - X este adăugat la vectorul sortat (final)
  - X este eliminat din heap
  - Heap-ul este reparat după condiția de min/max-heap

Intro Sort

- Se mai numește Introspective Sort
- Este sortarea din anumite implementări ale STL-ului
- Este un algoritm hibrid (combină mai mulți algoritmi care rezolvă aceeași problemă)
- Este format din Quick Sort, Heap Sort și Insertion Sort

IDEE:

- Algoritmul începe cu Quick Sort
- Trece în Heap Sort dacă nivelul recursivității crește peste logn
- Trece în Insertion Sort dacă numărul de elemente de sortat scade sub o anumită limită

TimSort

- Sortarea din python
- Este un algoritm hibrid care îmbina MergeSort cu sortare prin inserare.

IDEE

- Algoritmul începe cu Merge Sort
- Trece în Insertion Sort dacă numărul de elemente de sortat scade sub o anumită limită (32, 64)

Sortări prin comparație

Vizualizare:

https://www.cs.usfca.edu/~galles/visualization/ComparisonSort.html

- Quick Sort
- Merge Sort
- Algoritmi elementari de sortare

Clase de complexitate

Urmatoarele slideuri sunt copiate de la:http://cadredidactice.ub.ro/simonavarlan/files/2018/02/Curs-2-2018.pdf

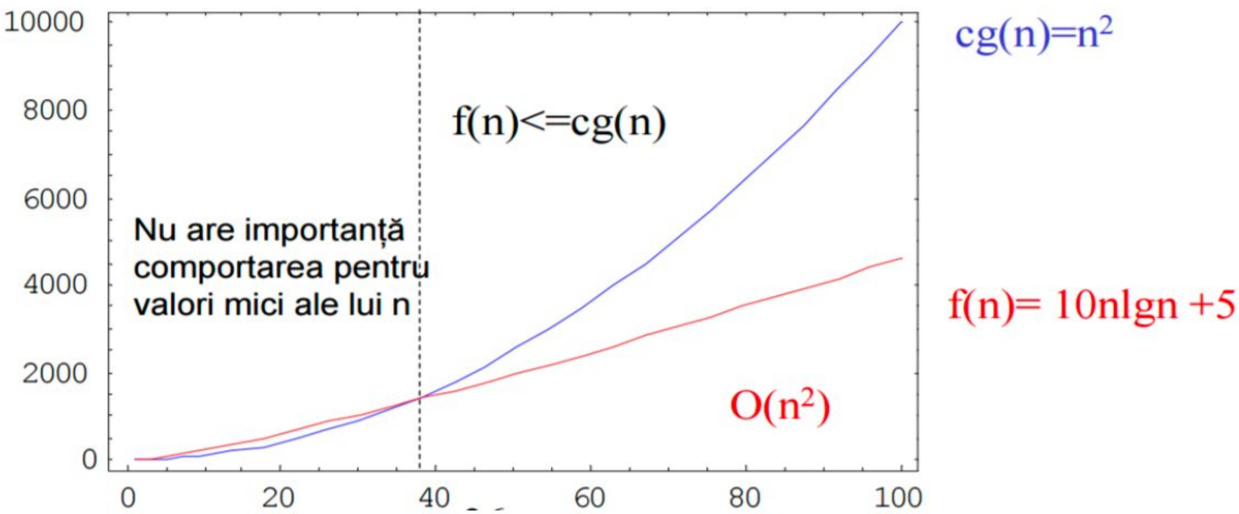
Ideal ar fi sa le refaci dacă ti-e rezonabil de usor .

Clase de complexitate

Complexitatea Algoritmilor

Notatia O

Ilustrare grafica. Pentru valori mari ale lui n, f(n) este marginită superior de g(n) multiplicată cu o constantă pozitivă



Big O

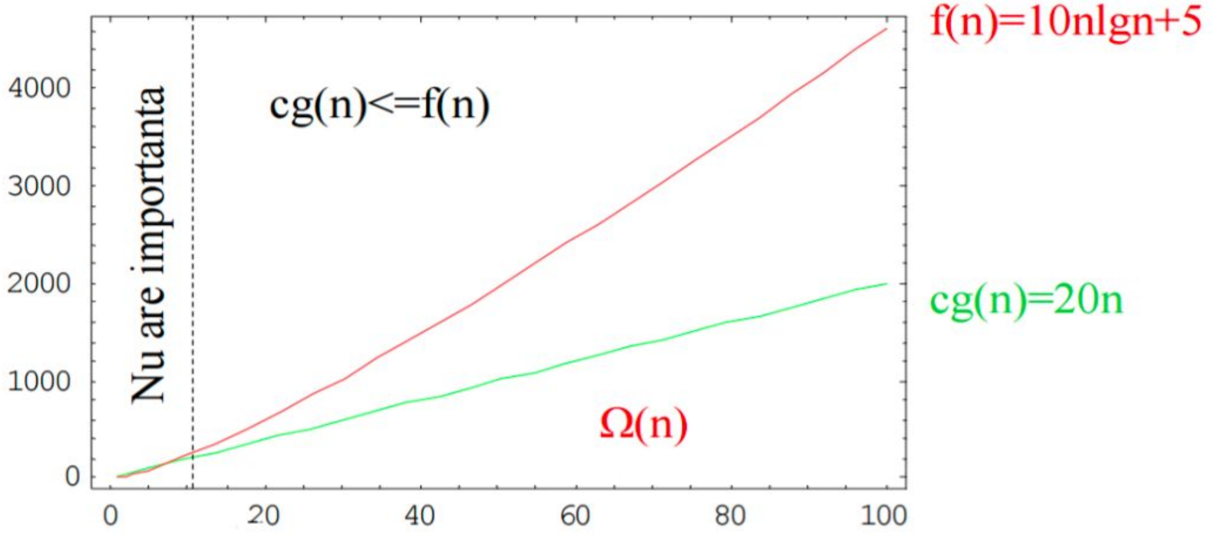
- O -> marginire superioară
  - Un algoritm care face 3 n operații este și O(n) dar și O(n^2) și O(n!)
  - În general vom vrea totusi marginea stransa care este de fapt **ϴ**

Clase de complexitate

Complexitatea Algoritmilor

Notația Ω

Ilustrare grafică. Pentru valori mari ale lui n, funcția f(n) este marginită inferior de g(n) multiplicată eventual de o constantă pozitivă

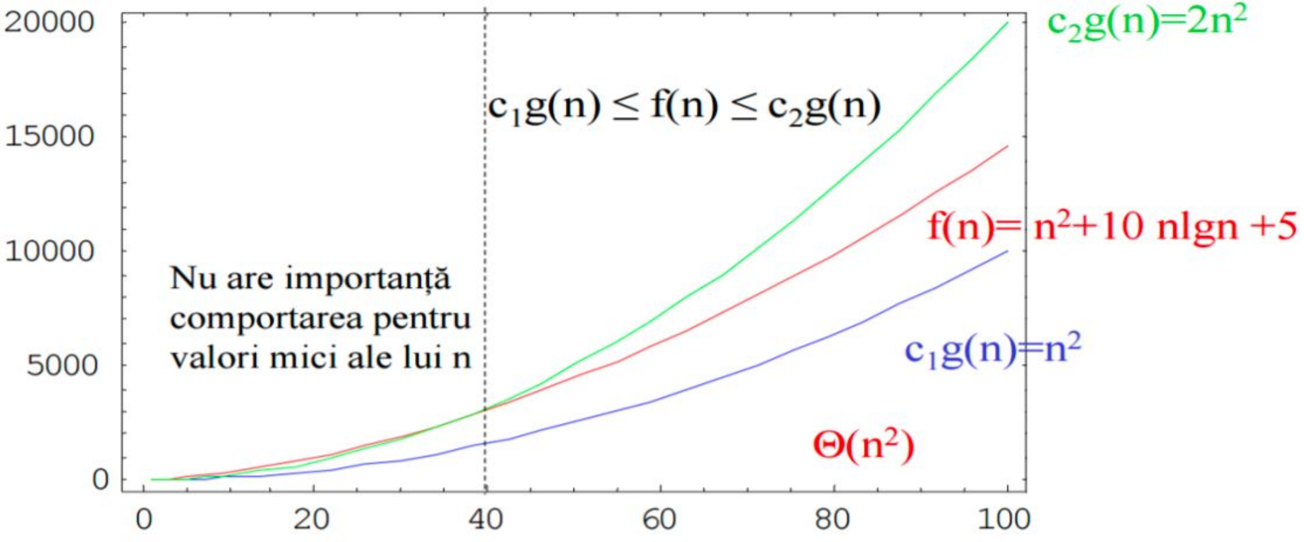


Clase de complexitate

Complexitatea Algoritmilor

Notația Θ

Ilustrare grafică. Pentru valori mari ale lui n, f(n) este mărginită, atât superior cât și inferior de g(n) înmulțit cu niște constante pozitive



Complexitatea minima pentru o sortare prin comparație

Teorema: Orice algoritm de sortare care se bazeaza pe comparatii face cel putin $\Omega(n \log n)$ comparatii.

Schita Demonstrație: Sunt în total n! permutari. Algoritmul nostru de sortare trebuie sa sorteze toate aceste n! permutări. La fiecare pas pe baza unei comparatii intre 2 elemente putem în funcție de răspuns sa eliminăm o parte din comparatii. La fiecare pas putem injumatatii numărul de permutări -> obținem minim: $\log_2 (n!)$ comparatii,

dar $\log_2 (n!) = \log_2 (n) + \log_2 (n - 1) + ... + \log_2 (2) = \Omega(n \log n)$.

Complexitatea minima pentru o sortare prin comparație

Teorema: Orice algoritm de sortare care se bazeaza pe comparatii face cel putin $\Omega(n \log n)$ comparatii.

Exemplu: N = 3, vrem sa sortam orice permutare a vectorului {1,2,3}:

(1,2,3) (1,3,2) (2,1,3) (2,3,1) (3,1,2) (3,2,1)

Facem o prima comparație sa zicem a1 ? a2. Sa zicem ca a1 > a2 -> raman 3 posibilitati: (1,2,3) (1,3,2), (2,3,1)

Dacă ulterior comparam 1 cu 3 … atunci dacă 3 > 1 am terminat dar dacă 1 > 3 ramanem cu (1,2,3) (1,3,2) și mai trebuie sa facem a 3-a comparatie...

Final