Structuri de date

Lect. Dr. Marius Dumitran

Organizatorice

- Notare
- Laboratoare / laboranți
- Curs live (cu anumite schimbari de orar)

Notare

- 40% laborator + 0.5p bonus
  - **Nota minim 5!!**
- 20% seminar + 0.5p bonus
  - Prezență, activitate și teme

- 40% examen + 0.5 p bonus
  - Examen scris (11-12 iunie ??)
  - **Nota minim 5!!**

- Posibil bonus la curs (rar) Kahoot/sau raspunsuri.
- Bonusurile doar pentru cei cu nota de trecere
- Media minim **5.50**

Notare

- 40% laborator (40 de puncte)
  - **Nota minim 5!!**
  - Teme
    - Proiect Sortari 10 p
    - Proiect Structura de Date Avansata 10p
    - Probleme diverse 20p
  - Nota laborator + bonus maxim 0.5p de la laborant (bonusul îl pot primi doar cei care au punctaj din teme)
  - **Laboratorul nu se poate reface decat anul viitor!**
  - Proiectele trimise in primele 10 zile dupa deadline au punctajul /2
  - Proiectele trimise dupa primele 10 zile de la deadline nu se iau in calcul.
  - Un proiect poate fi prezentat in urmatorele 2 laboaratoarele de dupa deadline (incluzand acel laborator)... apoi au valoarea 0...

- Radem glumim.... Dar invatam.
  - Examenul este important și o sa fie greu...
  - Cine vine la seminar/laborator deobicei trece.. Cine nu... mai rar..
  - Invatati din timpul anului, puneti intrebari la seminar/lab/curs...
  - Activitatea se puncteaza si la curs si la seminar si la laborator!

Curs live/online

- Programul probabil al cursurilor. Seminariile conform orarului.
  - Saptamana 2 - **1 martie (16-20) Cursurile 1 si 2**
  - Saptamana 3 - **8 martie (16-20) cursurile 3 si 4**
  - Saptamana 4 - 14 sau 15 martie cursul 5 (urmeaza sa stabilim data pe 8 martie)
  - Saptamana 5 - **22 martie nu facem curs**
  - Saptamana 6 - ???
  - Saptamana 7 - ???
  - Saptamana 8 - **12 aprile (16-20) 2 cursuri**
  - Saptamana 9 - ?
  - Saptamana 10 **3 mai curs normal**
  - Saptamana 11 **10 mai (16-20)**
  - Saptamana 12-14 **curs normal**

Overview al materiei

- Curs 1-2 Sortări/Căutare binară
  - count sort, radix sort, quick sort, merge sort
- Curs 3 Vectori/Liste înlănțuite
  - Cozi
  - Stive
  - Deque
- Curs 4 Heapuri
- Curs 5 Heapuri binomiale - fibonacci
- Curs 6 Huffman
- Curs 7 Arbori binari de căutare
- Curs 8 AVL / Red black
- Curs 9 Skip Lists / Treaps
- Curs 10 Arbori de intervale
- Curs 11 RMQ & LCA & LA
- Curs 12-13 Hashuri
- Curs 14 Tries / Suffix trees ?

Algoritmi de sortare

Ce algoritmi de sortare cunoașteți?

Algoritmi de sortare

Ce algoritmi de sortare cunoașteți?

- Bubble O(n^2)
- Merge O(nlogn)
- Interschimbare O(n^2)
- Radix
- Quick O(nlogn)?
- Heap O(nlogn)
- Bucket Sort
- Count Sort
- Bogo Sort O(n!*n)
- Gravity Sort O(n^2)
- Selection Sort O(n^2)
- Insert sort O(n^2)
- Shell Sort O(n sqrt n) ~ discutabil
- Intro Sort O(nlogn) alg hibrid
- Tim Sort O(nlogn) alg hibrid

Putem grupa după:

- Complexitate
- Complexitate spațiu
- Stabilitate
- Dacă se bazează pe comparații sau nu

Algoritmi de sortare stabili

- Un algoritm de sortare este stabil dacă păstrează ordinea elementelor egale.
- 5 5 5 -> 5 5 5 (sortare stabilă)
- 5 5 5 -> 5 5 5 (sortare instabilă) sau oricare alta permutare

Atenție: Și unii algoritmi instabili pot sorta stabil uneori, algoritmii stabili garantează asta pentru orice input.

Pentru numere naturale nu este important, dar când sortăm altfel de obiecte acest lucru poate deveni important.

Algoritmi de sortare

Clasificare

Elementari	Prin comparație	Prin numărare
Insertion sort → $O(n^2)$	Quick sort → O(n logn)	Bucket sort
Selection sort → $O(n^2)$	Merge sort → O(n logn)	Counting sort
Bubble sort → $O(n^2)$	Heap sort → O(n logn)	Radix sort
	Intro sort → O(n logn)	

Tabel cu sortări:
https://en.wikipedia.org/wiki/Sorting_algorithm#Comparison_of_algorithms

Sortare prin numărare / Counting Sort

- Algoritm de sortare a numerelor întregi mici
- Presupunem că vectorul de sortat **v** conține **n** elemente din mulțimea {0, ..., max}

IDEE:

- Creem un vector de frecvență **fr**
- Numărăm aparițiile fiecărui element din **v**
- Modificăm vectorul **fr** a.î.
  - `fr[i] = numărul de elemente cu valoare = i`
- La final, iterăm prin vectorul `fr[i]` și afișăm `i` de `fr[i]` ori pentru toate numerele de la 1 la max.

Sortare prin numărare / Counting Sort

Exemplu: sortăm note

Note	5	3	2	9	4	5	1	7	10	3	9	2	5
nota	1	2	3	4	5	6	7	8	9	10			
fr	1	2	2	1	3		1		2	1			

Sortare prin numărare / Counting Sort

Exemplu: sortăm note

Note	5	3	2	9	4	5	1	7	10	3	9	2	5
nota	1	2	3	4	5	6	7	8	9	10			
fr	1	2	2	1	3	0	1	0	2	1			

Sortare prin numărare / Counting Sort

Exemplu: sortăm note

nota	1	2	3	4	5	6	7	8	9	10			
fr	1	2	2	1	3	0	1	0	2	1			
soluție	1	2	2	3	3	4	5	5	5	7	9	9	10

Sortare prin numărare / Counting Sort

Exemplu: sortăm note

Note	5	3	2	9	4	5	1	7	10	3	9	2	5
soluție	1	2	2	3	3	4	5	5	5	7	9	9	10

Cod

```
//Pasul 1: Crestem frecventa fiecarui element din vector:

 for (int i = 0; i < n; ++i)   // O(n)
        fr[note[i]]++, maxn = max(maxn, note[i]);

// Pasul 2: afisam fiecare element de atatea ori cat apare in vectorul de frecventa
// O(maxn * n)
// O(maxn + n)

 for (int i = 0; i <= maxn; ++i) {    // pana la maxn
        for (int j = 1; j <= fr[i]; ++j) {   // worst case se duce pana la n
                                             // for-ul va face n + maxn
            cout << i << " ";  // Afisam de fix n ori
        }
 }
```

Complexitate? Spațiu? Timp?

Counting Sort

Complexitate

- Timp:
  - O(n + max)

- Spațiu:
  - O(max)

Counting Sort

Vizualizare:

https://visualgo.net/bn/sorting

Counting Sort

Ce ne facem dacă avem de sortat numere mari...

- Până la 10^6 ?
- Până la 10^{18} ?
- Numere care nu sunt întregi ?

Counting Sort

Ce ne facem dacă avem de sortat numere mari...

- Până la 10^6 ?
  - Depinde de N, dar **Count Sort** poate fi cea mai bună opțiune...
- Până la 10^{18} ?
  - Nu mai putem folosi Count Sort. Putem folosi în schimb **Radix Sort**
- Numere care nu sunt întregi ?
  - Mai greu și cu Radix Sort (nu e imposibil, dacă sunt doar 1-2 zecimale putem înmulți cu 10, 100) ... altfel putem folosi **Bucket Sort**

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
- Sortăm fiecare bucket (discutăm 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

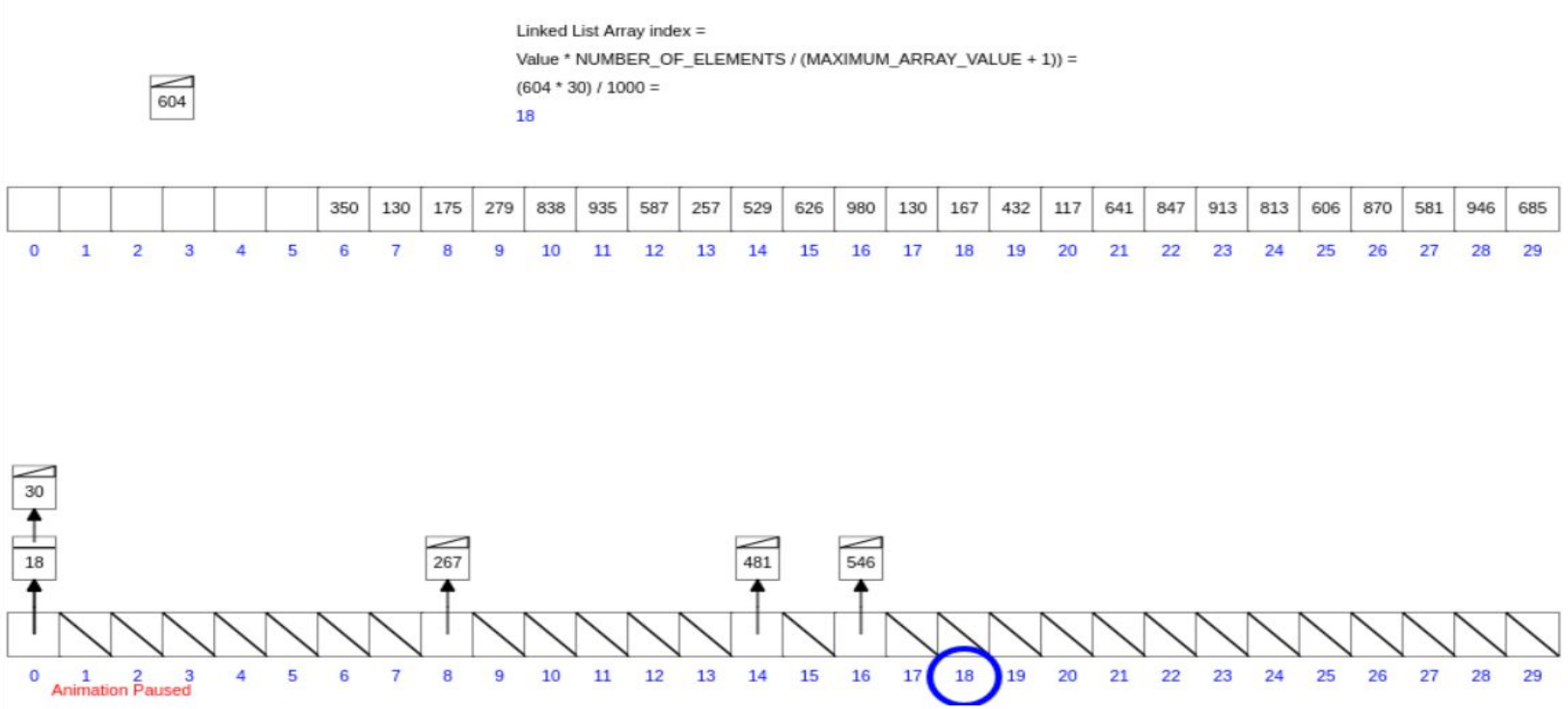
Cum adăugăm elementele în bucket-ul corespunzător?

- Metoda clasică este să împărțim la o valoare și, în funcție de câtul împărțirii, punem valoarea în bucketul corespunzător.

- În animație foloseam 30 de bucketuri și, cum numerele erau până la 1000, înmulțeam cu 30 și împărțeam la 1000
- E mai frumos sa împărțim la puteri de 2... (folosind operatia de shiftare pe biti)

Bucket Sort

Cum adăugăm elementele în bucket-ul corespunzător?



Bucket Sort

Cum adăugăm elementele în bucket-ul corespunzător?

- Metoda clasică este să împărțim la o valoare și, în funcție de cât, să punem în bucketul corespunzător

Cum sortăm bucketurile ?

- Putem aplica recursiv tot bucketsort sau, dacă avem puține elemente, să folosim o sortare simplă (insertion / selection / bubble sort) ....
  - Cum adică să folosim bubble sort? De ce nu quicksort ???
    - Pentru n mic, constanta de la quicksort, mergesort face ca sortarea să fie mai înceată

Bucket Sort

- Câte bucketuri ?
  - Dacă sunt foarte multe, inițializăm spațiu prea mare
  - Dacă sunt prea puține, nu dispersăm suficient...
    - Ce se întâmplă dacă toate pică în același bucket ?
  - Contează foarte mult și distribuția inputului.

Bucket Sort

Complexitate?

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

Algoritm bun dacă avem o distribuție uniformă 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 să considerăm numerele (putem folosi 10, 100, 10^4 sau 2, 2^4, 2^{16} ...)

- 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) (discuție mai lungă)

- 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

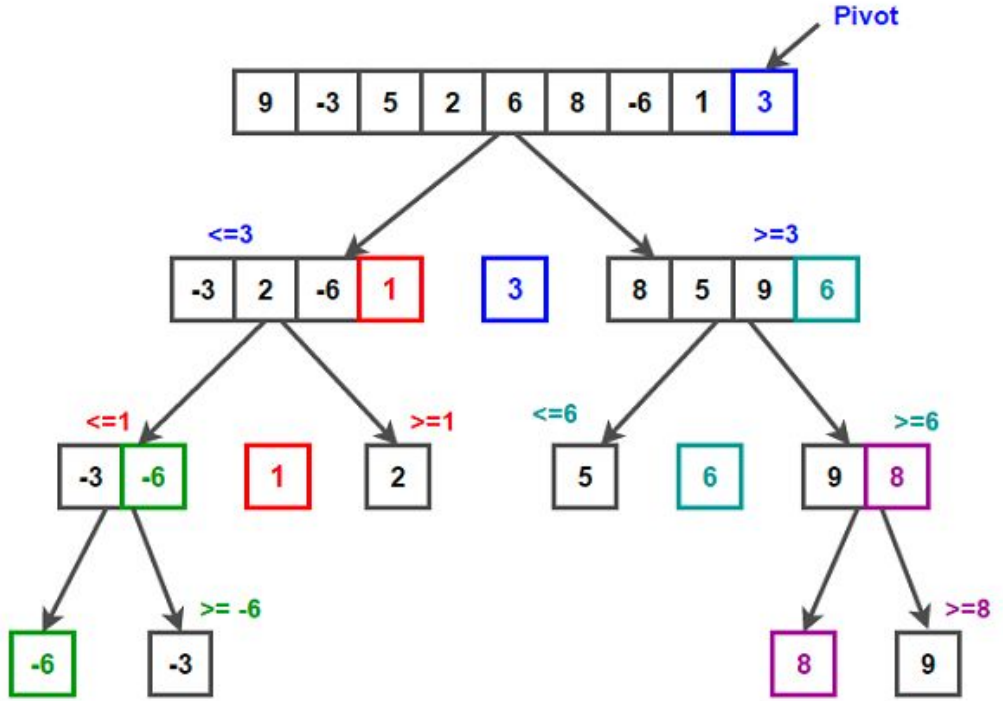
- Algoritm Divide et Impera
- Este un algoritm eficient în practică (implementarea este foarte importantă)

- **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

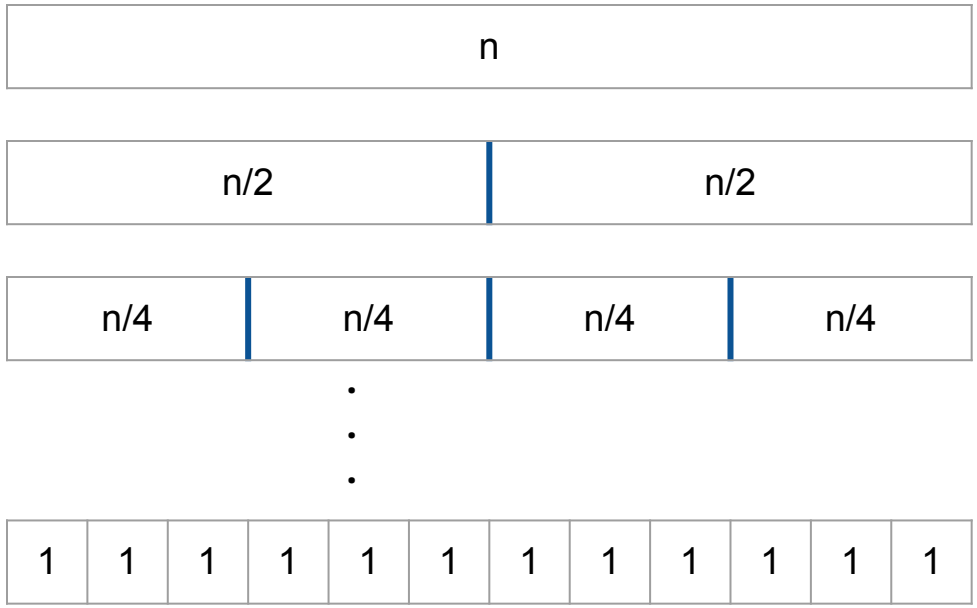
Quick Sort - exemplu

- Pivot ales la coadă
- Contraexemplu ?
  - 1 2 3 4 5 6 7 8 9
- Pivotul în centru
  - 4 3 2 1 **9** 8 7 6 5
  - *4 3 2* ***1*** *8 7 6 5* **9**
  - **1** *4 3 2 8 7 6 5* **9**



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?

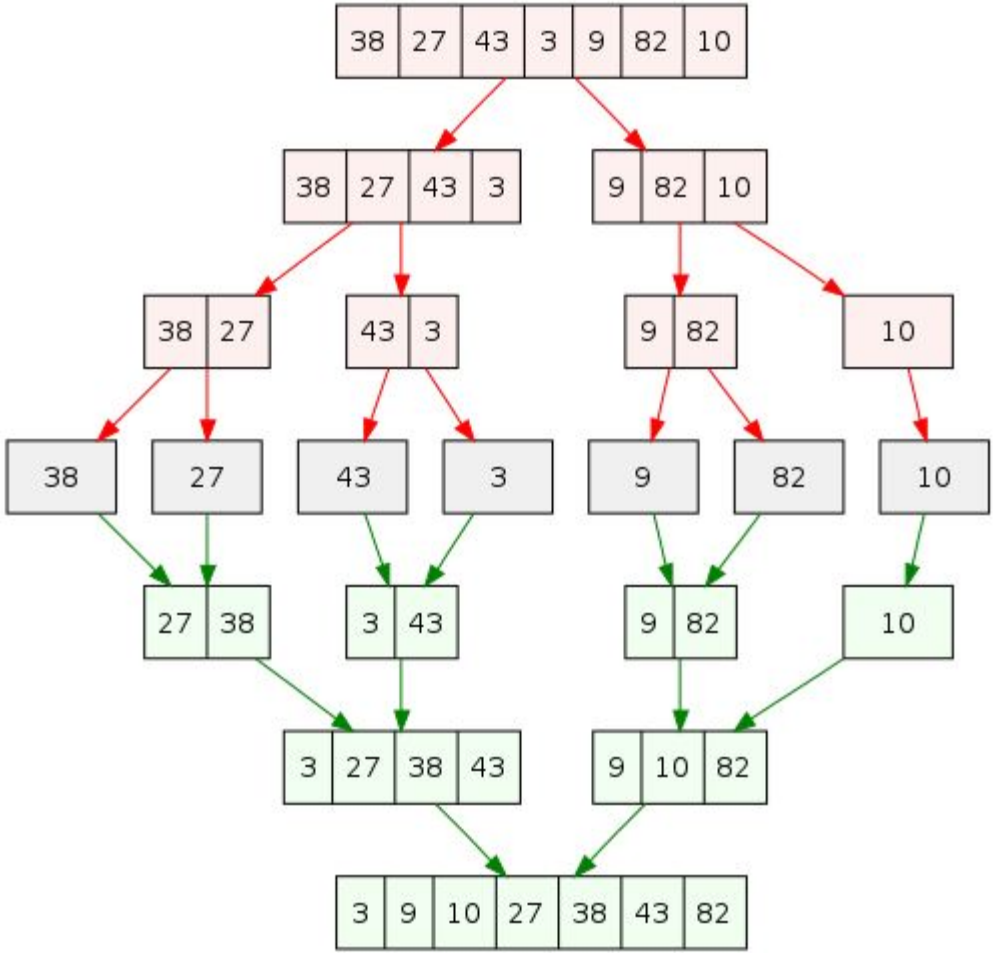
- Primul element
- Elementul din mijloc
- Ultimul element
- Un element random
- Mediana din 3
- Mediana din 5, 7 (**atenție** când vectorul devine mic, facem mult calcul pentru puțin)
- Mediana medianelor

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

Merge Sort

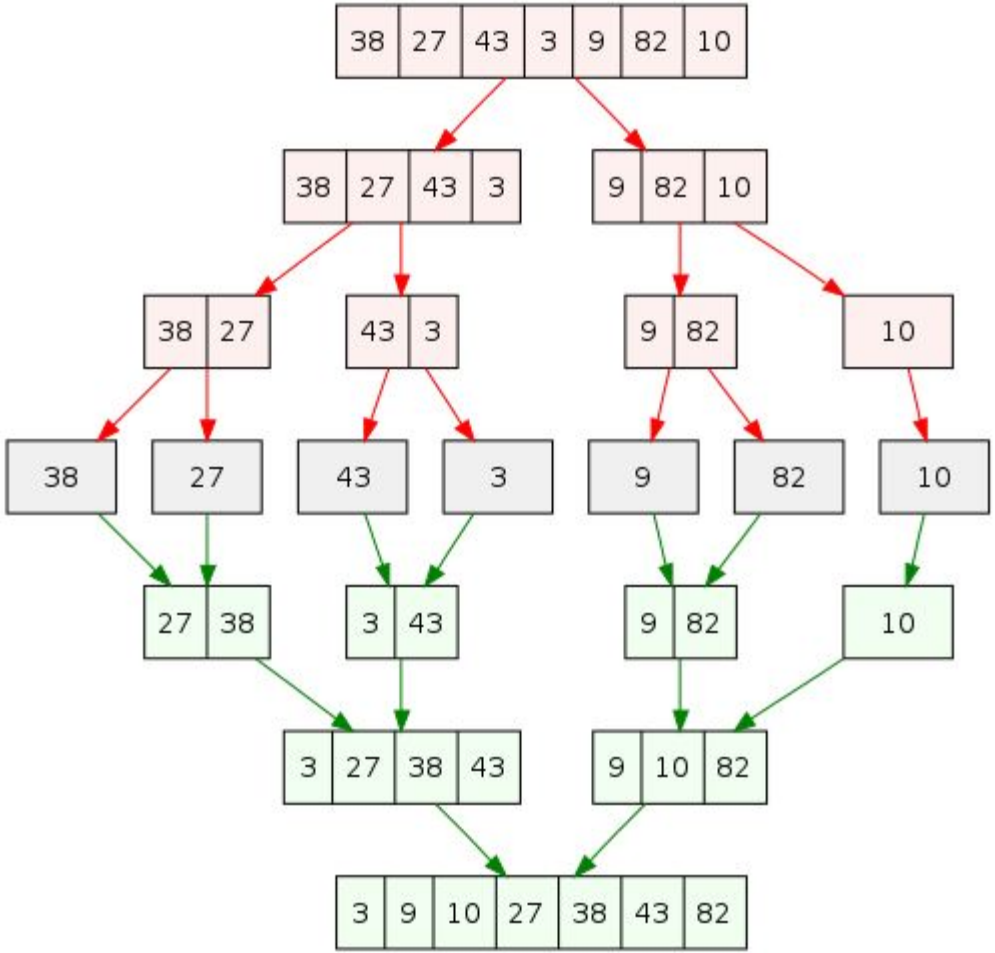
- Algoritm Divide et Impera
- **Divide:** se împarte vectorul în jumătate și se sortează independent fiecare parte
- **Impera:** se sortează recursiv cei doi subvectori

Merge Sort - exemplu



Merge Sort - exemplu

Complexitate: Cati pasi avem ?
Log2 (n)
Cat ma costa interclasarea a
M +n elemente ? O(m+n)
Cat ma costa ultimul nivel ? O(n)
n/2+n/2 -> n
Cat ma costa nivelul anterior tot
(n/4+n/4) + (n/4+n/4) = n
Nivelul de mai sus o sa fie
4 * (n/8 + n/8) = n
-> Complexitatea total nr_de_pa
O(nlogn)



Merge Sort

- Când se oprește recursivitatea?
  - Când vectorul ajunge de lungime 1 sau 2 (depinde de implementare)
  - La fel ca la quicksort, ne-am putea opri mai repede ca să evităm multe operații pentru puține 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 practică atunci când cazul ideal de la Quick Sort e când împărțim în 2 exact ce face Merge Sort?

- Merge Sort are nevoie de un vector suplimentar și face multe mutări suplimentare.

- Quick Sort e “in place”... memoria suplimentară e pentru stivă...

In-Place Merge Sort

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

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 îmbină Merge Sort 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

- Fie f, g două funcții definite pe $\mathbb{Z}^+$

Notații:

- O->O mare(margine superioară - *upper bound*)
- Ω->Omega(margine inferioară - *lower bound*)
- Θ->Teta(categorie constantă - *same order*)

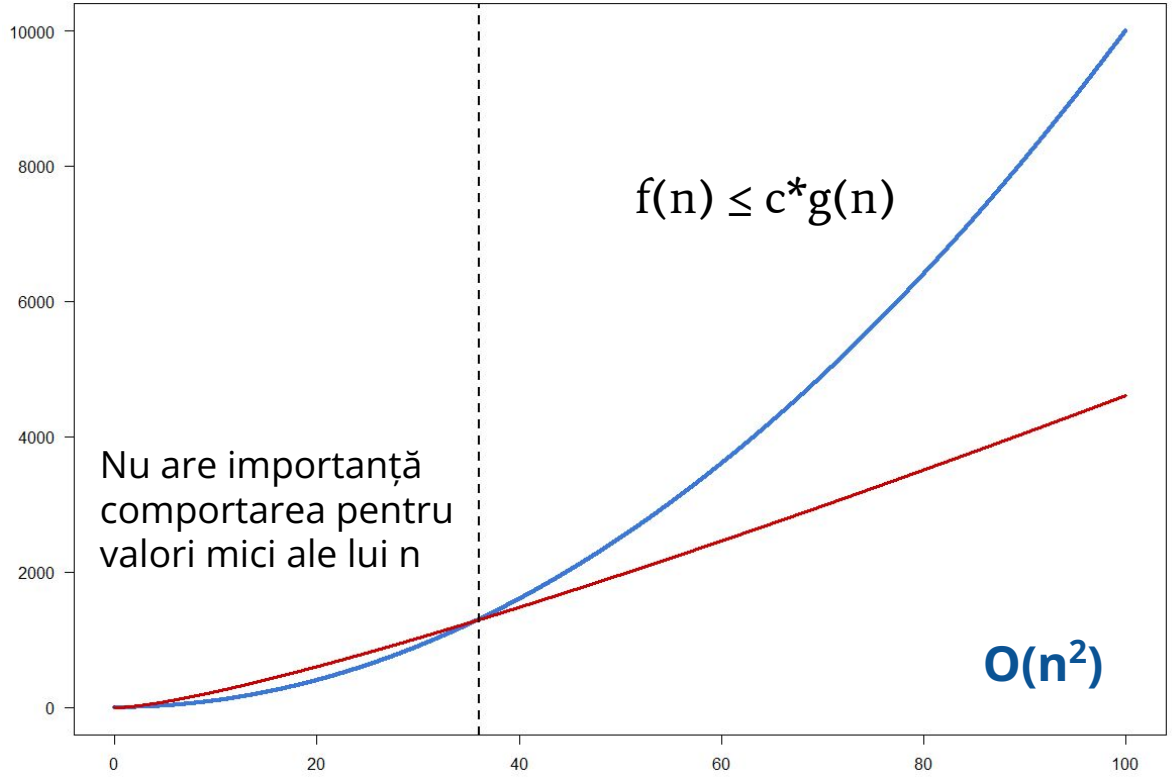
Big-O

- O → mărginire superioară
  - Un algoritm care face 3*n operații este și O(n), dar și O(n^2) și O(n!)
  - În general, vom vrea totuși marginea strânsă, care este de fapt Θ

- $f(n) = O(g(n))$, dacă există constantele *c* și n_0 astfel încât $f(n) \leq c * g(n)$ pentru $n \geq n_0$

Big-O

Ilustrare grafică. Pentru valori mari ale lui n, f(n) este mărginită superior de c*g(n), c > 0



$c*g(n) = n^2$

$f(n) = 10n*\ln(n) + 5$

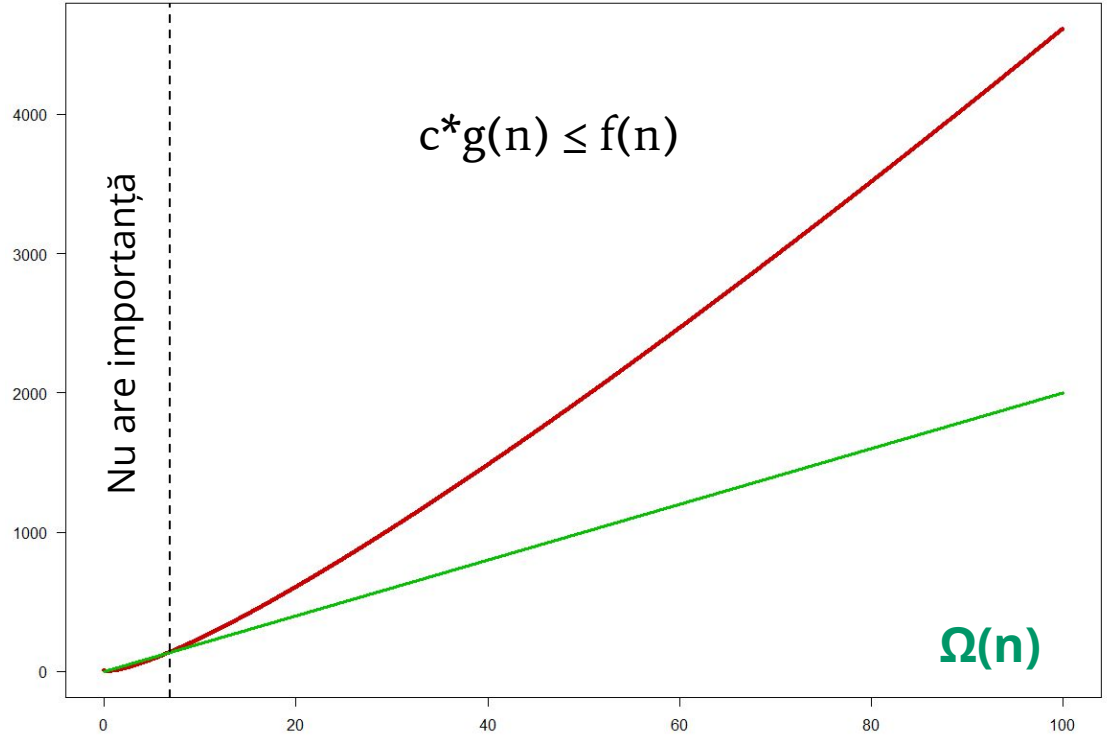
Big-Ω

- $\Omega \rightarrow$ mărginire inferioară

- $f(n) = \Omega(g(n))$, dacă există constantele *c* și n_0 astfel încât $f(n) \geq c * g(n)$ pentru $n \geq n_0$

Big-Ω

Ilustrare grafică. Pentru valori mari ale lui n, f(n) este mărginită inferior de c*g(n), c > 0



$f(n) = 10n*\ln(n) + 5$

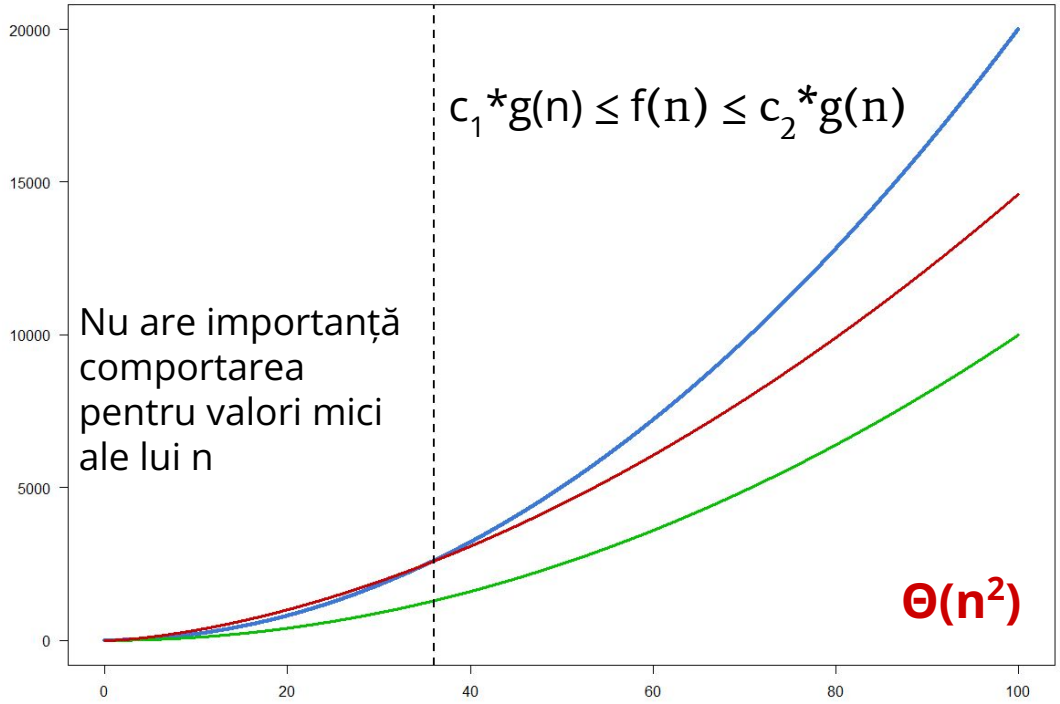
$c*g(n) = 20n$

Big-Θ

- Θ → mărginire dublă (si inferioară și superioară)

Big-Θ

Ilustrare grafică. Pentru valori mari ale lui n, f(n) este mărginită atât inferior cât și superior de $c_1*g(n)$, respectiv $c_2*g(n)$, $c_1, c_2 > 0$



$c_2*g(n) = 2n^2$

$f(n) = n^2 + 10n*\ln(n) + 5$

$c_1*g(n) = n^2$

Complexitatea minimă pentru o sortare prin comparație

Teoremă: Orice algoritm de sortare care se bazează pe comparații face cel putin $\Omega(n \log n)$ comparații.

Schiță de demonstrație:
Sunt în total n! permutări. Algoritmul nostru de sortare trebuie să sorteze toate aceste n! permutări. La fiecare pas, pe baza unei comparații între 2 elemente, putem, în funcție de răspuns, să eliminăm o parte din comparații. La fiecare pas, putem înjumătăți numărul de permutări → obținem minim $\log_2(n!)$ comparatii, dar

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

Complexitatea minimă pentru o sortare prin comparație

Teoremă: Orice algoritm de sortare care se bazează pe comparații face cel putin Ω(n logn) comparații.

Exemplu: N = 3, vrem să sortăm **descrescător** orice permutare a vectorului {1,2,3}:

(A1,A2,A3) (A1,A3,A2) (A2,A1,A3) (A2,A3,A1) (A3,A1,A2) (A3,A2,A1)

Facem o primă comparație, să zicem a_1 ***?*** a_2.
Să zicem că $a_1 > a_2 \rightarrow$ rămân 3 posibilități: (a1,a2,a3) (a1,a3,a2) (a3,a1,a2)
Dacă ulterior comparăm a_1 cu a_3 ... atunci:

- dacă $a_3 > a_1$ am terminat
- dacă $a_1 > a_3$ atunci rămânem cu (a1,a2,a3) (a1,a3,a2) și mai trebuie să facem a 3-a comparație...

Heap Sort

Vizualizare:

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

6 5 3 1 8 7 2 4

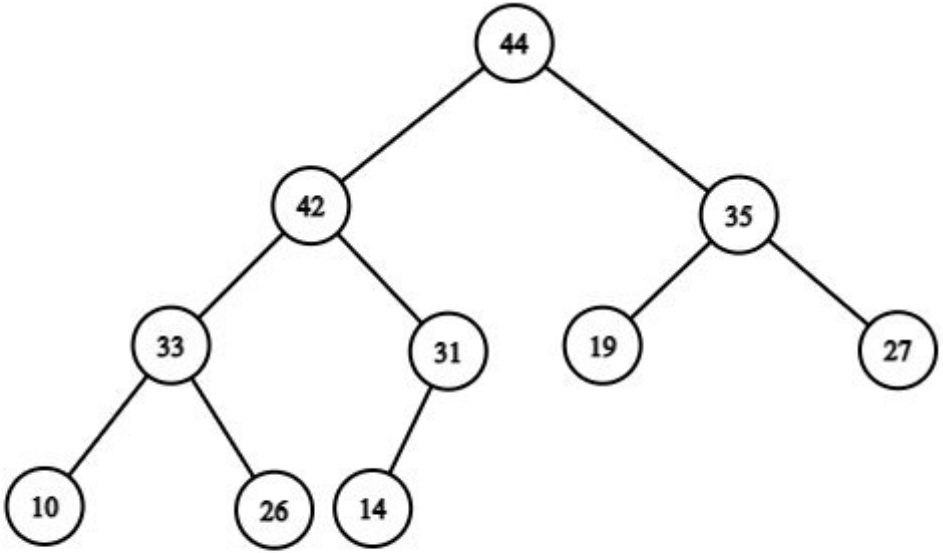
Scurtă introducere în heap-uri

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

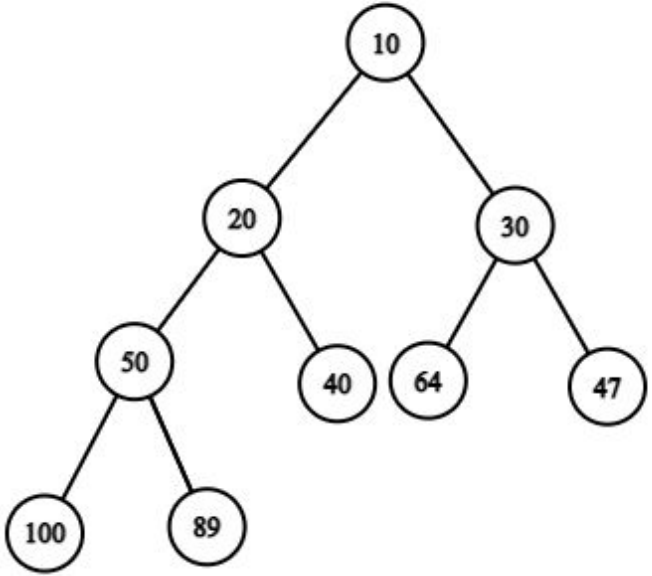
- Max-heap
  - Pentru orice nod X, fie T tatăl lui X
  - T are valoarea ≥ decât valoarea lui X
  - Elementul maxim este în rădăcină

- 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

Kahoot

https://create.kahoot.it/creator/2281f10b-f400-43fe-981c-85377fd66c12

Final