STRUCTURI DE DATE

Lect. Dr. Marius Dumitran

Organizatorice

- Notare
- Laboratoare / laboranți

Notare

- 30% laborator + 0.5p bonus
  - **Nota minim 5!!**
- 30% seminar + 1p bonus
  - 7 teme * 5p (30p pentru 10 la seminar)
  - **Minim 3 teme rezolvate complet (nu neaparat corect) (nu se pot recupera la restante…)**
  - **Minim 4 prezențe din 7**
  - **Prezenta la si 10 minute!!**

- 40% examen + **0.5 p bonus curs**
  - Examen scris (24 iunie ora 10:00(10:15 start))
  - **Nota minim 5!!**
- Posibil bonus la curs (rar) Kahoot/sau răspunsuri.
- Bonusurile doar pentru cei cu notă de trecere
- **Întrebările despre curs se pun live sau prin șeful grupei!**
- Media minim **5.50**

Notare

- 30% laborator (30 de puncte) ???
  - **Nota minim 5!!**
  - Teme de grup
    - Proiect Sortări 10 p
    - Proiect Structură de Date Avansată 10p
    - Exercitii diverse 10p

  - 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 decât anul viitor!**
  - **Fiecare zi un punct minus.**
  - Un proiect poate fi prezentat în următoarele 2 laboratoare de după deadline (incluzând acel laborator)... apoi au valoarea 0…

- Râdem, glumim…. Dar învățăm.
  - Examenul este important și o să fie greu…
  - Cine vine la seminar/laborator de obicei trece.. Cine nu… mai rar..
  - Învățați din timpul anului, puneți întrebări la seminar/lab/curs…
  - Activitatea se punctează și la curs și la seminar și la laborator!

Overview al materiei

- Curs 1-2 Sortări/Căutare binară
  - count sort, radix sort, quick sort, merge sort
- Curs 3-4 Vectori/Liste înlănțuite
  - Cozi
  - Stive
  - Deque
- Curs 5 Heapuri
- Curs 6 Heapuri binomiale - fibonacci, 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 altă 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 (vacute dupa numarul de buline albe…).

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:

- Creăm un vector de frecvență **fr**
- Numărăm aparițiile fiecărui element din **v**
- Modificăm vectorul **fr** a.î.
  - fr[i] = numărul de elemente cu valoare egală cu 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: Creștem frecvența fiecărui element din vector:

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

// Pasul 2: afișăm fiecare element de atâtea ori cât apare în vectorul de frecvență
// O(maxn + n)

 for (int i = 0; i <= maxn; ++i) {   // până la maxn
       for (int j = 1; j <= fr[i]; ++j) {  // worst case se duce până la n

// for-ul va face n + maxn
            cout << i << " ";  // Afișăm 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

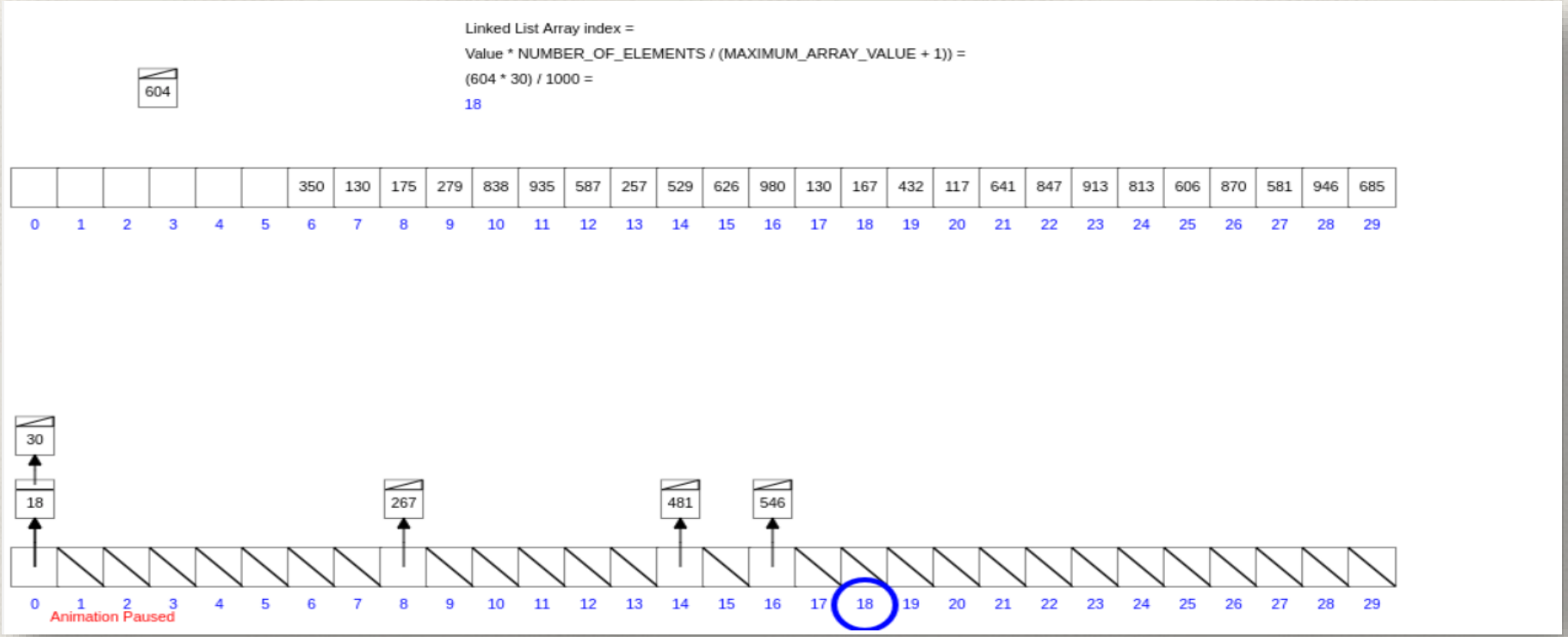
Cum adăugăm elementele în bucket-ul corespunzător?

- 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 operația de shiftare pe biți)

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 bucket-urile ?

- 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

Cum sortăm bucket-urile ?
-Wikipedia:

Pseudocode [edit]

```
function bucketSort(array, k) is
    buckets ← new array of k empty lists
    M ← 1 + the maximum key value in the array
    for i = 0 to length(array) do
        insert array[i] into buckets[floor(k × array[i] / M)]
    for i = 0 to k do
        nextSort(buckets[i])
    return the concatenation of buckets[0], ...., buckets[k]
```

Let *array* denote the array to be sorted and *k* denote the number of buckets to use. One can compute the maximum key value in linear time by iterating over all the keys once. The floor function must be used to convert a floating number to an integer (and possibly casting of datatypes too). The function *nextSort* is a sorting function used to sort each bucket. Conventionally, insertion sort is used, but other algorithms could be used as well, such as *selection sort* or *merge sort*. Using *bucketSort* itself as *nextSort* produces a relative of radix sort; in particular, the case *n = 2* corresponds to quicksort (although potentially with poor pivot choices).

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), unde k reprezintă numărul de bucket-uri

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

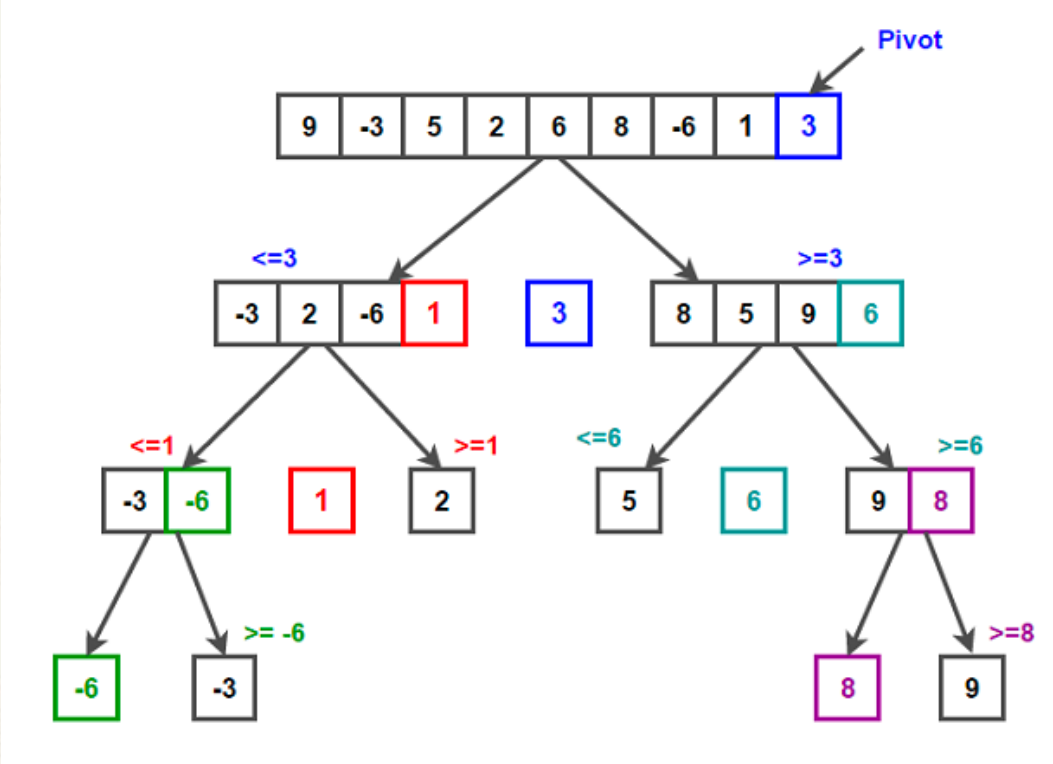
- Algoritm Divide et Impera
- Este un algoritm eficient în practică (implementarea este foarte importantă)

- **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

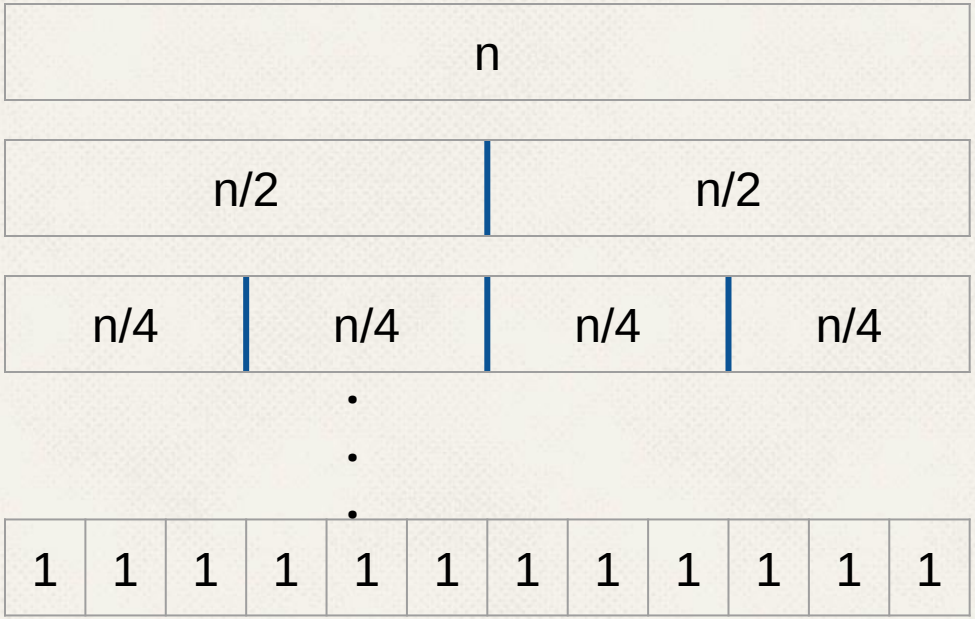
Quick Sort - exemplu

- 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



1 partiție * n = O(n)

2 partiții * n/2 = O(n)

4 partiții * n/4 = O(n)

.
.
.

logn nivele, O(n) / nivel = O(n logn)

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?
  - **$O(n^2)$**

Average Case O(n log n) (proof)

Quick Sort

Cum alegem pivotul?

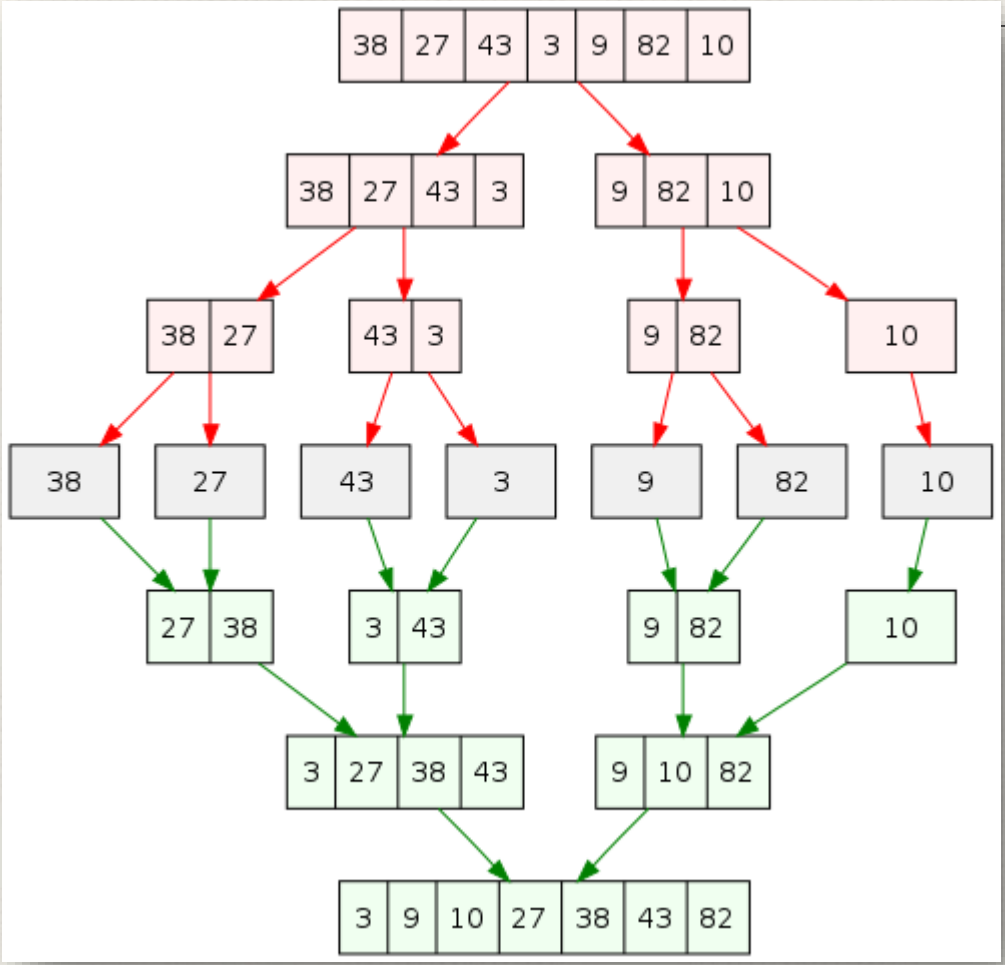
- Primul element
- Elementul din mijloc
- Ultimul element
- Un element random
- Mediana din 3
- Mediana din 5, 7 (**atenție** când vectorul devine mic, facem mult calcul pentru puțin)
- Mediana medianelor

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: Câți pași avem ? **Log2 (n)**

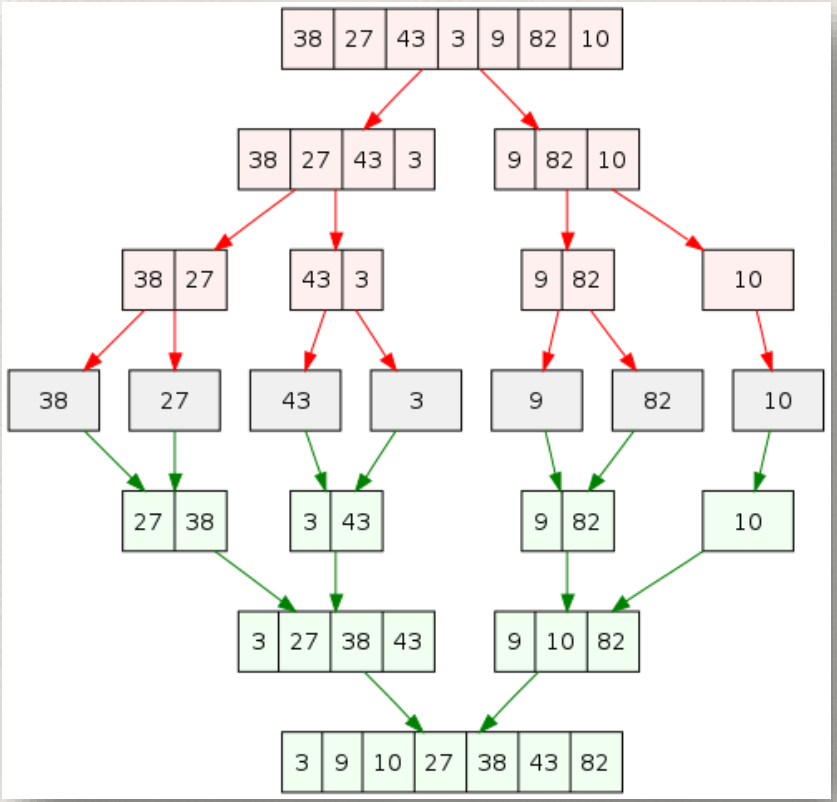
Cât mă costă interclasarea a m+n elemente ? **O(m+n)**

Cât mă costă ultimul nivel ? **O(n)**
Explicație: n/2+n/2 -> n

Cât mă costă nivelul anterior? **Tot n**
Explicație: (n/4+n/4) + (n/4+n/4) = n

Nivelul de mai sus o să fie 4 * (n/8 + n/8) = n

- Complexitatea total nr_de_pași * Cost_pe_pas
  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
  - Creăm 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 log n	● respectiv trece în Insertion Sort dacă numărul de elemente de sortat scade sub o valoare (32/64)

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

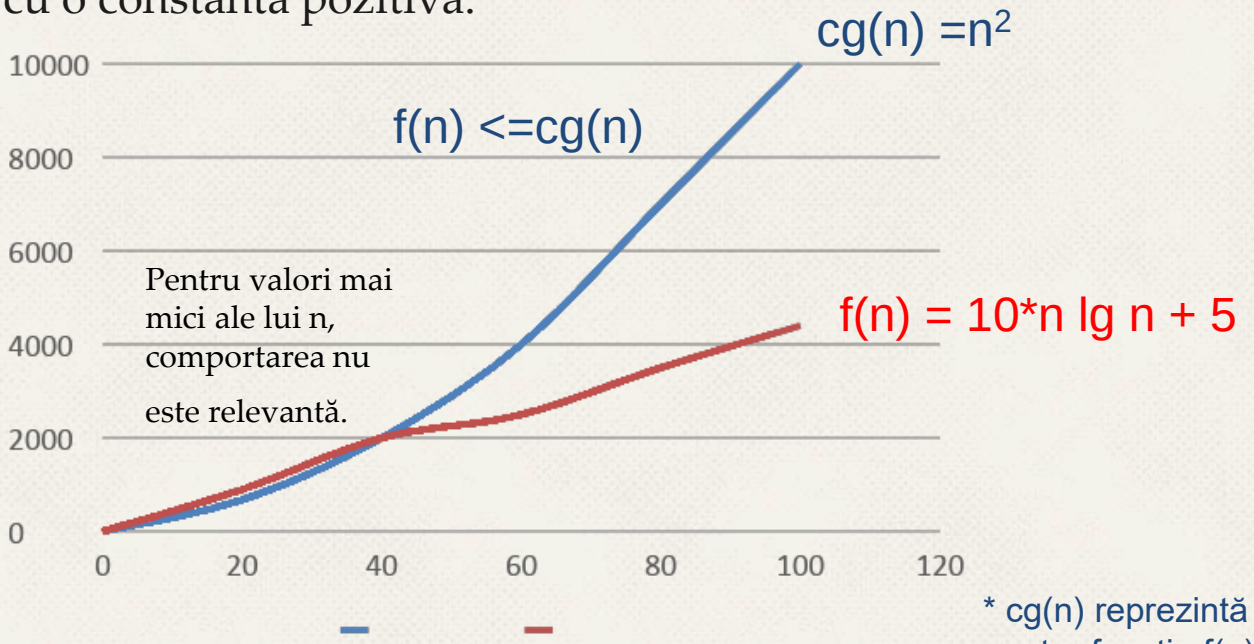
- O → mărginire superioară
  - Un algoritm care face 3*n operații este și O(n), dar și $O(n^2)$ și O(n!)
  - În general, vom vrea totuși marginea strânsă, care este de fapt Θ

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

Clase de complexitate

Notația O

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



* cg(n) reprezintă o limită superioară pentru funcția f(n)

Big-Ω

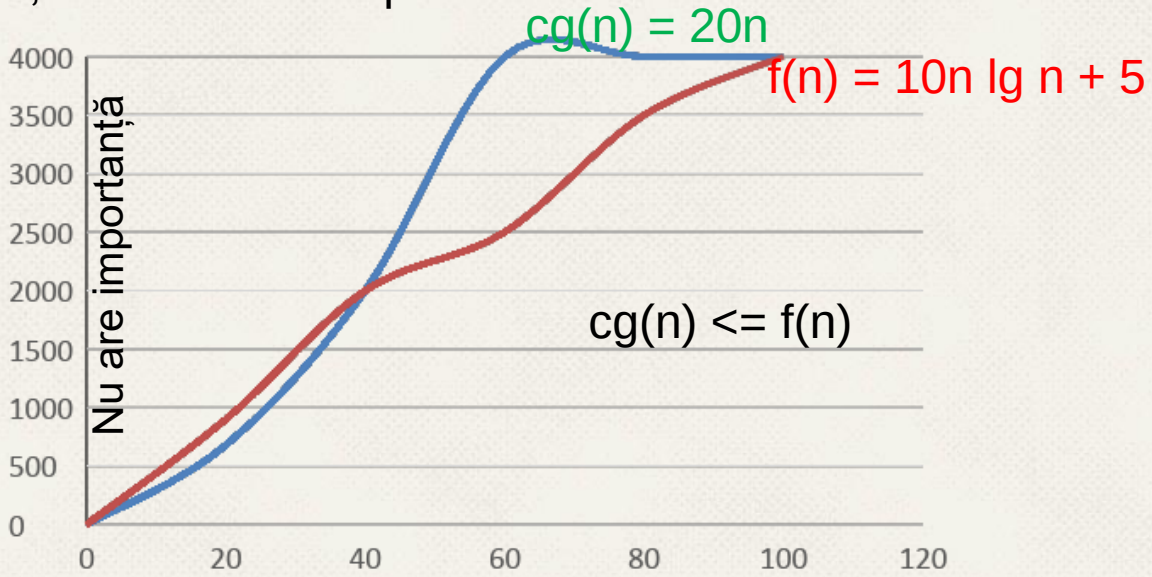
- $\Omega \rightarrow$ mărginire inferioară

- f(n) = Ω(g(n)), dacă există constantele c și n_0 astfel încât f(n) ≥ c * g(n) pentru n ≥ n_0

Clase de complexitate

Notația Ω

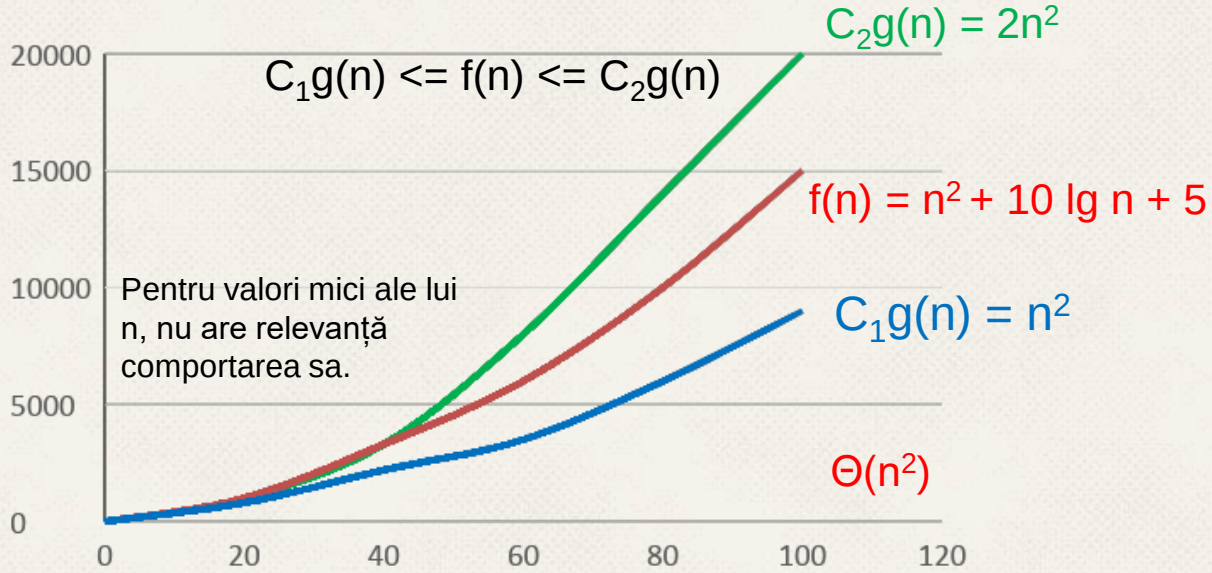
Ilustrare grafică. Pentru valori mai mari ale lui n, f(n) este marginită inferior de g(n) multiplicată, eventual, de o constantă pozitivă.



Clase de complexitate

Notația Θ

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



Complexitatea minimă pentru o sortare prin comparație

Teoremă: Orice algoritm de sortare care se bazează pe comparații face cel puțin $\Omega(n \log n)$ 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 $\log_2(n!)$ comparații, 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 puțin $\Omega(n \log n)$ 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

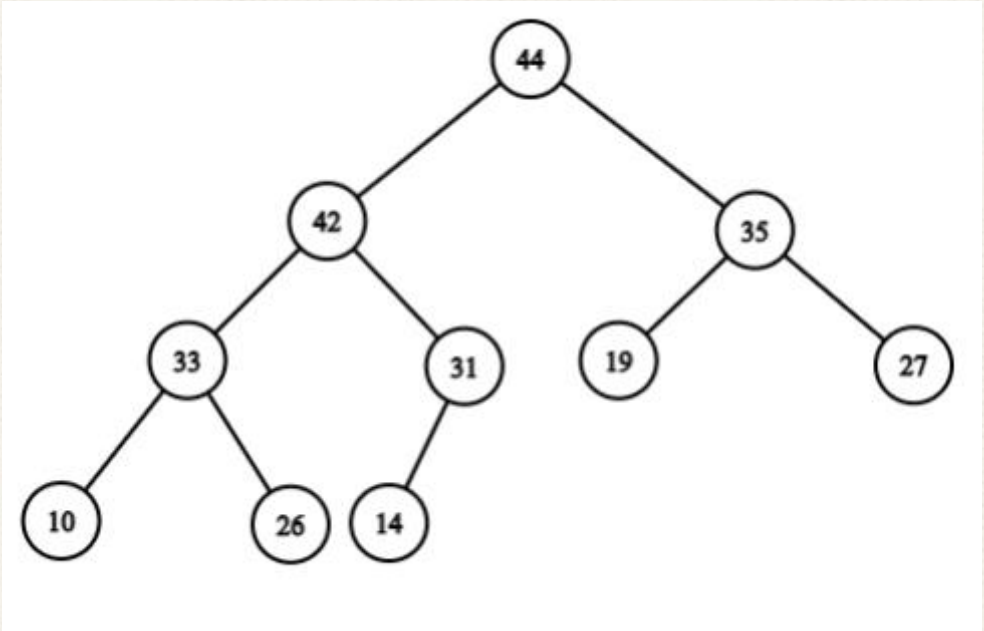
Scurtă introducere în heap-uri

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

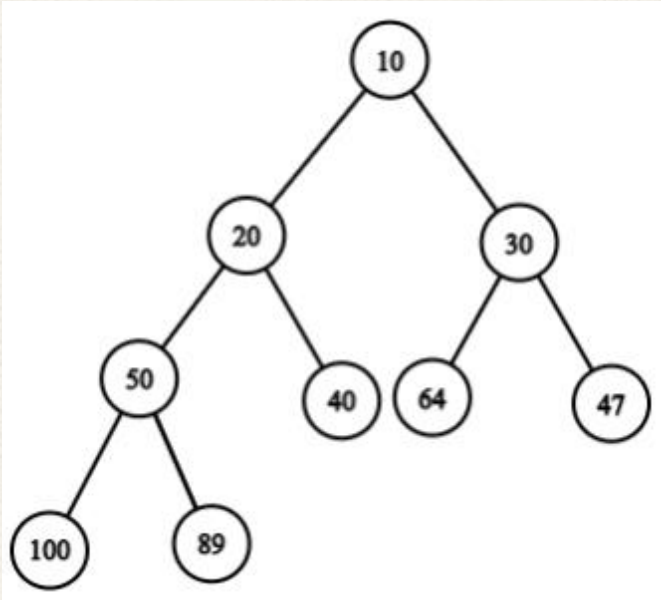
- Max-heap
  - Pentru orice nod X, fie T tatăl lui X
  - T are valoarea ≥ decât valoarea lui X
  - Elementul maxim este în rădăcină

- 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

https://www.mentimeter.com/app/presentation/alza967bkkyfus3auujmotibyo5x7a1b/nxqwg2qwbkbp