
Sortări

— 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 înlanțuite
- Fiecare bucket conține elemente care îndeplinesc aceleași condiții

IDEE:

- Fie **\mathbf{v}** vectorul de sortat și **\mathbf{b}** vectorul de buckets
- Se inițializează vectorul auxiliar cu liste (buckets) goale
- Iterăm prin **\mathbf{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 **\mathbf{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

Bucket Sort

Cum adăugăm elementele în bucket-ul corespunzător?

604

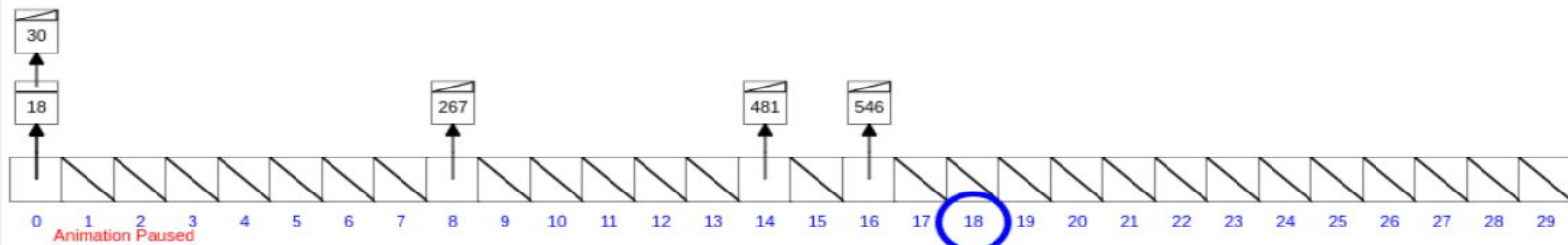
Linked List Array index =

$\text{Value} * \text{NUMBER_OF_ELEMENTS} / (\text{MAXIMUM_ARRAY_VALUE} + 1) =$

$(604 * 30) / 1000 =$

18

						350	130	175	279	838	935	587	257	529	626	980	130	167	432	117	641	847	913	813	606	870	581	946	685
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29



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 încheată

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 $\{0, \dots, B-1\}$

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

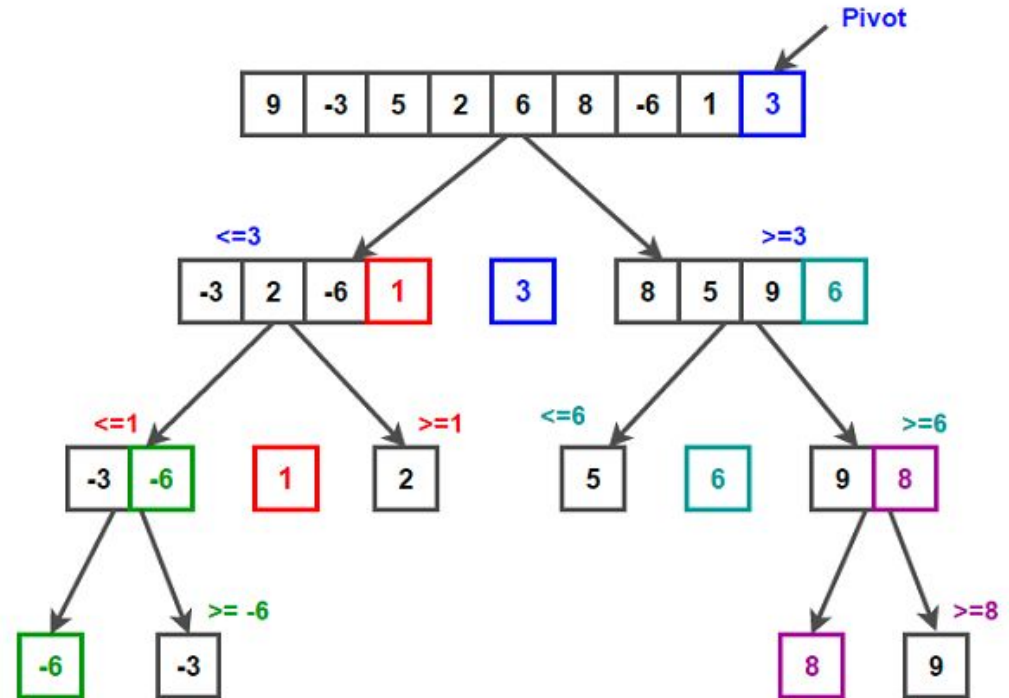
- MSD = **M**ost **S**ignificant **D**igit (recursiv, ca bucket sort)

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 $\leq x \leq$ elementele din subvectorul din dreapta
- **Impera:** se sortează recursiv cei doi subvectori

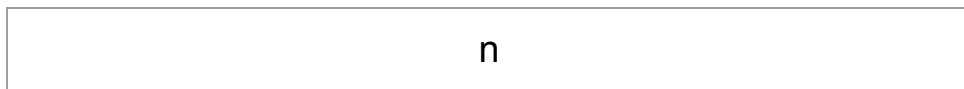
Quick Sort - exemplu

- Pivot ales la coadă
- Contraexemplu ?



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 * $n/4 = O(n)$

⋮

⋮



$\log n$ nivele, $O(n) / \text{nivel} = O(n \log n)$

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) $\rightarrow O(n - k)$ comparații
- Complexitate finală?
 - $O(n^2)$

Quick Sort

Cum alegem pivotul?

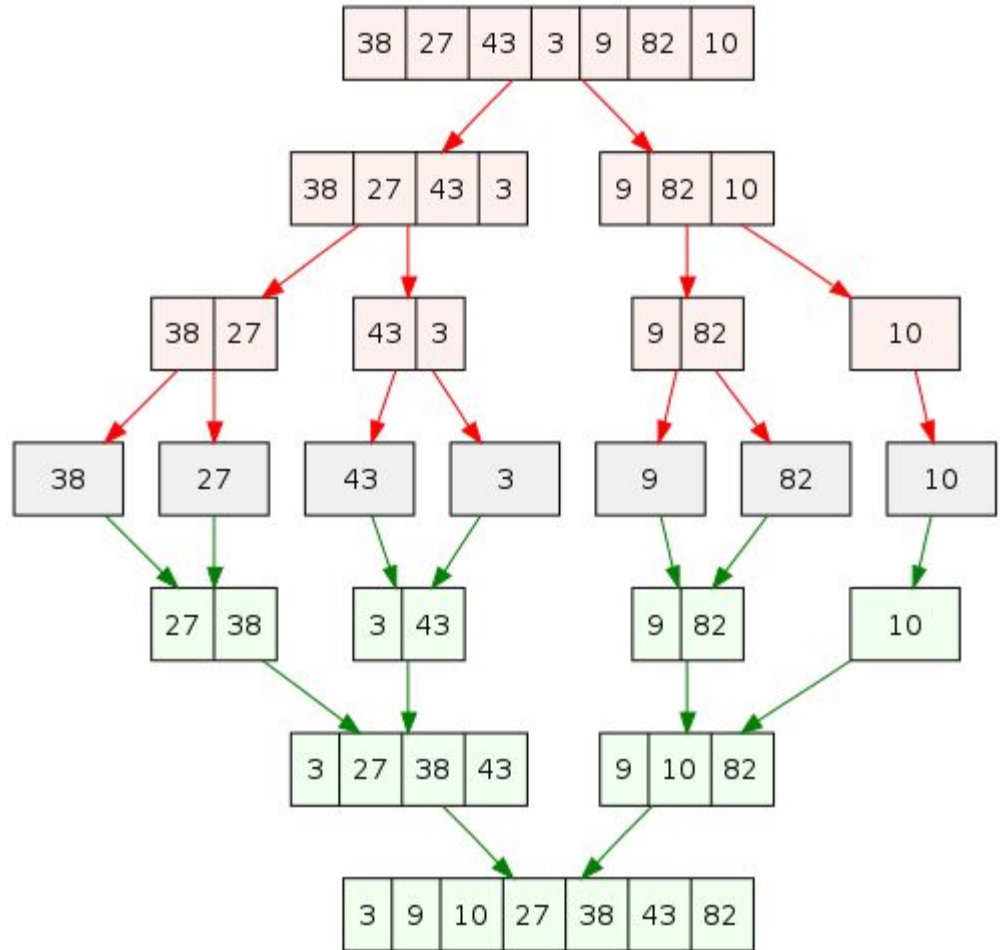
- Primul element
- Elementul din mijloc
- Ultimul element
- Un element random
- Mediana din 3
- Mediana din 5, 7 (**atenție** când vectorul devine mic, facem mult calcul pentru puțin)

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

- 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 $O(n \log n)$
 - 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 $\log n$
- 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}^+

Notatii:

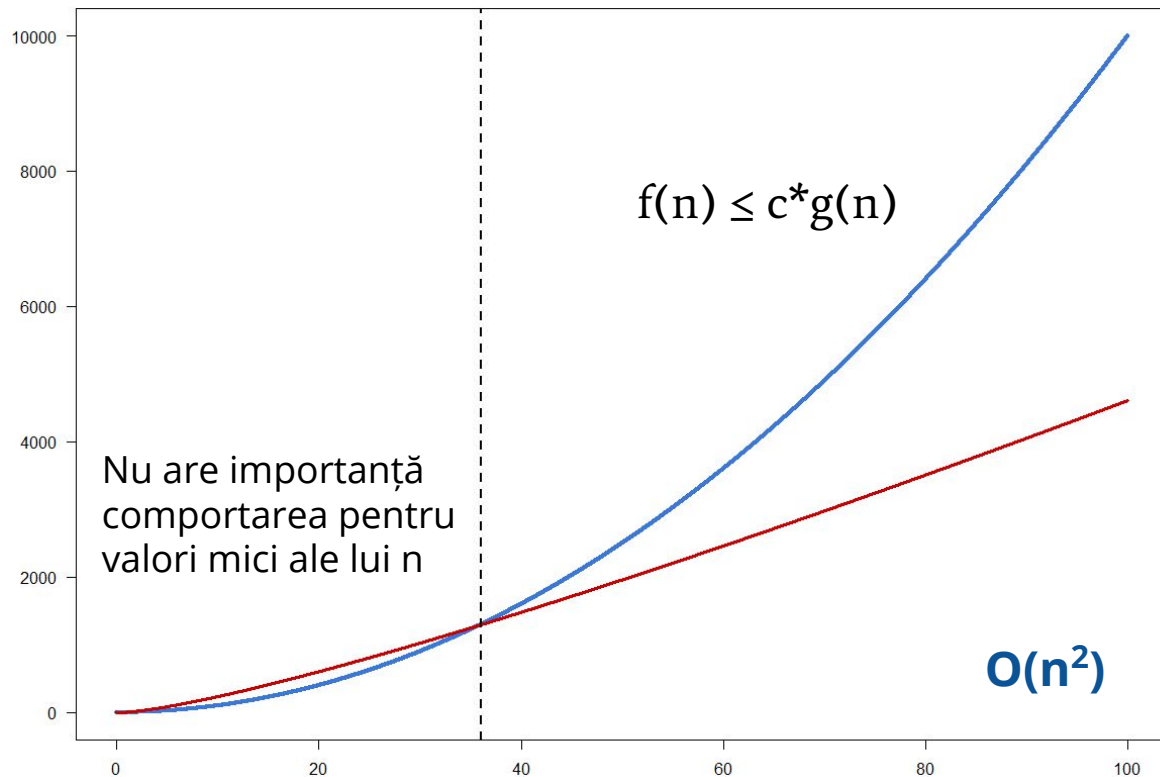
- O (margine superioară - *upper bound*)
- Ω (margine inferioară - *lower bound*)
- Θ (categorie constantă - *same order*)

Big-O

- $O \rightarrow$ mărginire superioară
 - Un algoritm care face $3 \cdot n$ 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 $f(n) \leq c \cdot g(n)$ pentru $n \geq n_0$

Big-O

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



$$c \cdot g(n) = n^2$$

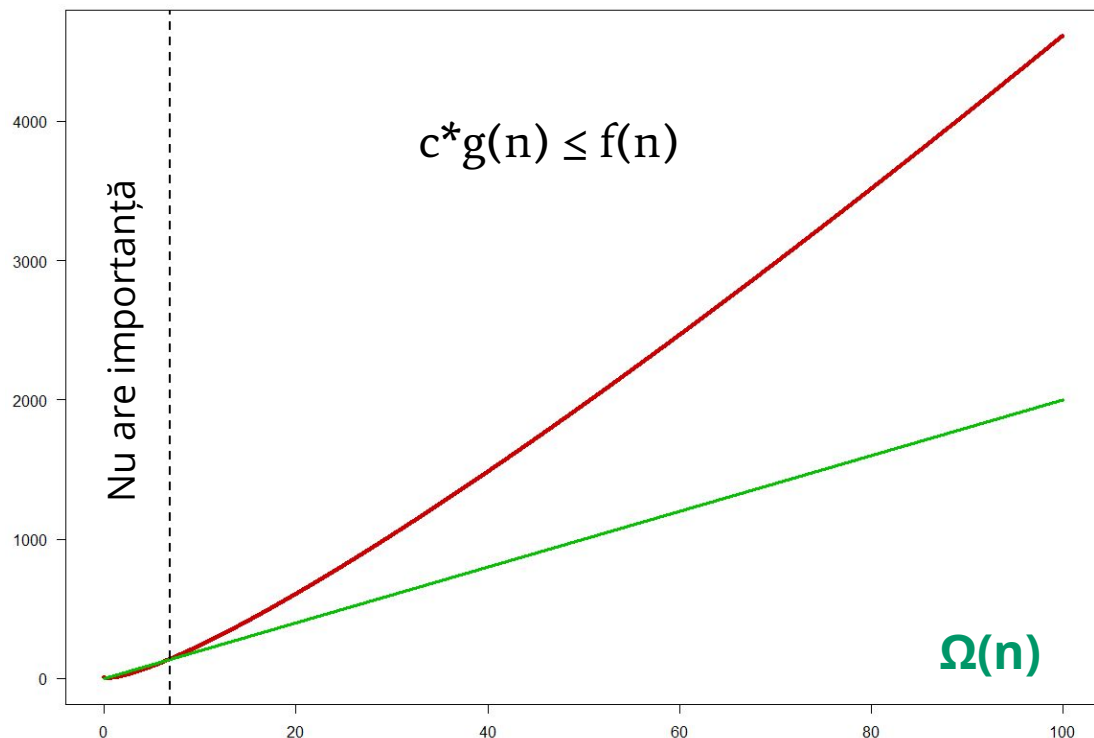
$$f(n) = 10n \cdot \ln(n) + 5$$

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 $c \cdot g(n)$, $c > 0$



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

$$c \cdot g(n) = 20n$$

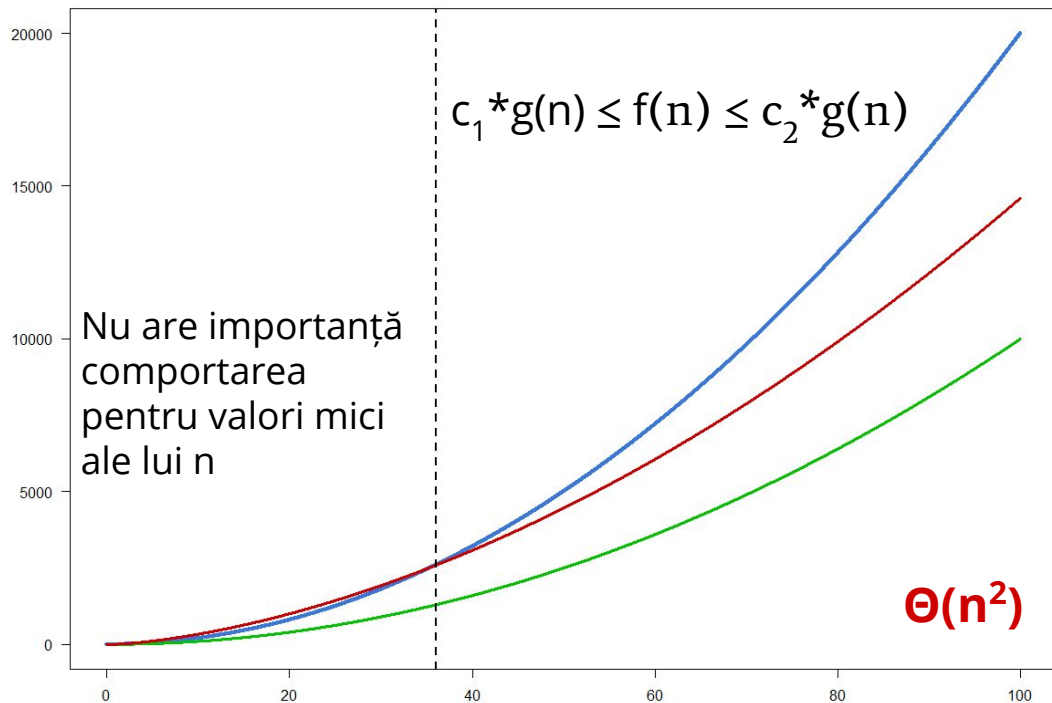
$\Omega(n)$

Big- Θ

- $\Theta \rightarrow$ marginire dubla (si inferioara si superioara)

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 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 \rightarrow obținem minim $\log_2(n!)$ comparații, dar

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

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 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 primă comparație, să zicem $a_1 ? a_2$.

Să zicem că $a_1 > a_2 \rightarrow$ rămân 3 posibilități: (1,2,3) (1,3,2) (2,3,1)

Dacă ulterior comparăm a_1 cu a_3 ... atunci:

- dacă $a_3 > a_1$ am terminat
- dacă $a_1 > a_3$ atunci rămânem cu (1,2,3) (1,3,2) și mai trebuie să facem a 3-a comparație...

Heap Sort

Vizualizare:

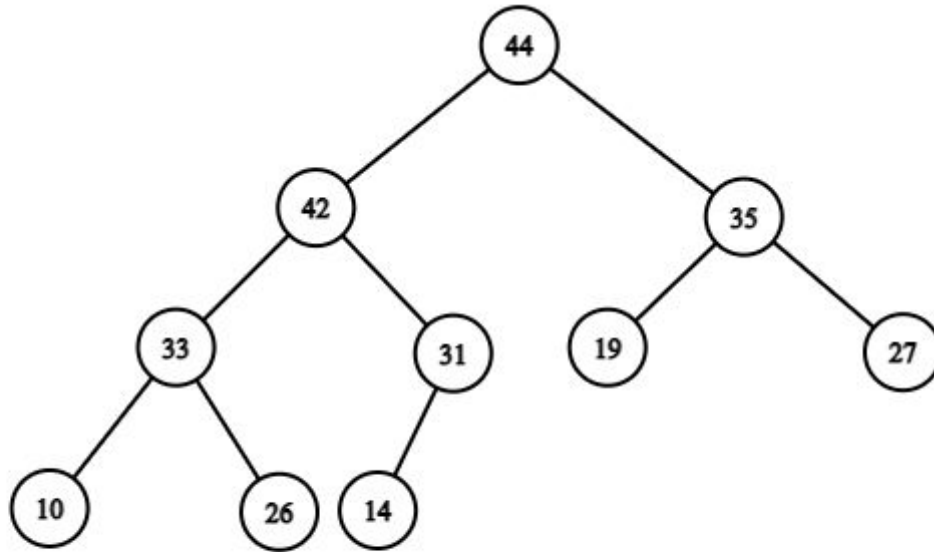
<https://www.cs.usfca.edu/~galles/visualization/HeapSort.html>

6 5 3 1 8 7 2 4

Scurtă introducere în heap-uri

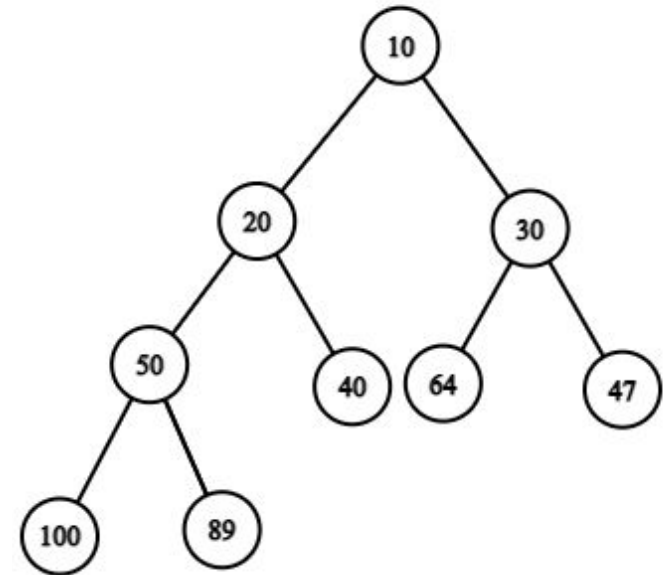
- Ce este un heap?
 - Arbore binar aproape complet
 - Are înălțimea $h = \log n$
- Max-heap
 - Pentru orice nod X , fie T tatăl lui X
 - T are valoarea \geq decât valoarea lui X
 - Elementul maxim este în rădăcină
- Min-heap
 - Pentru orice nod X , fie T tatăl lui X
 - T are valoarea \leq decât valoarea 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

Final