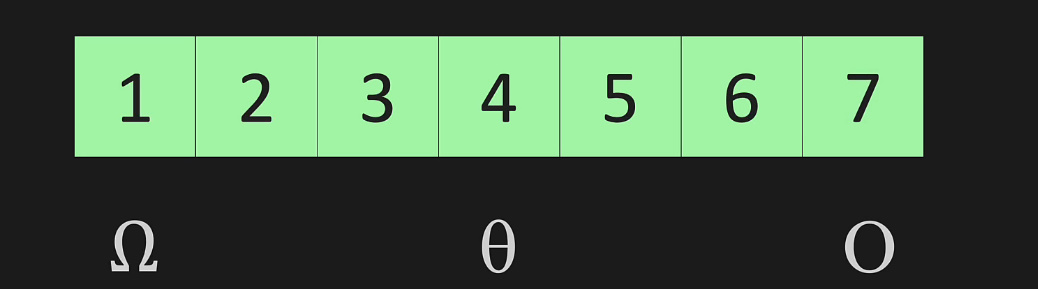
**Big O**

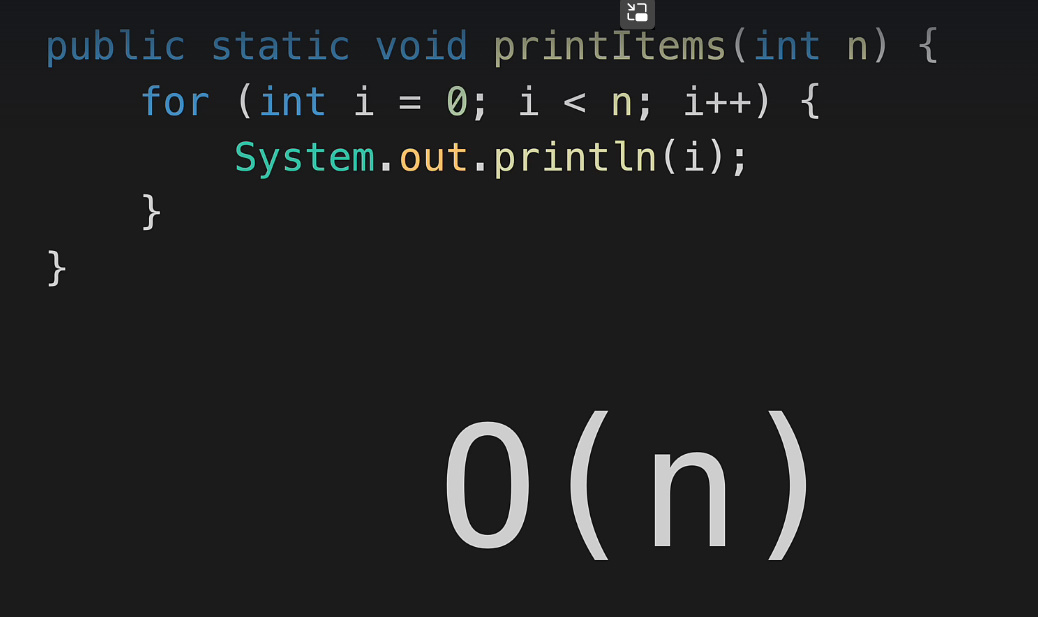
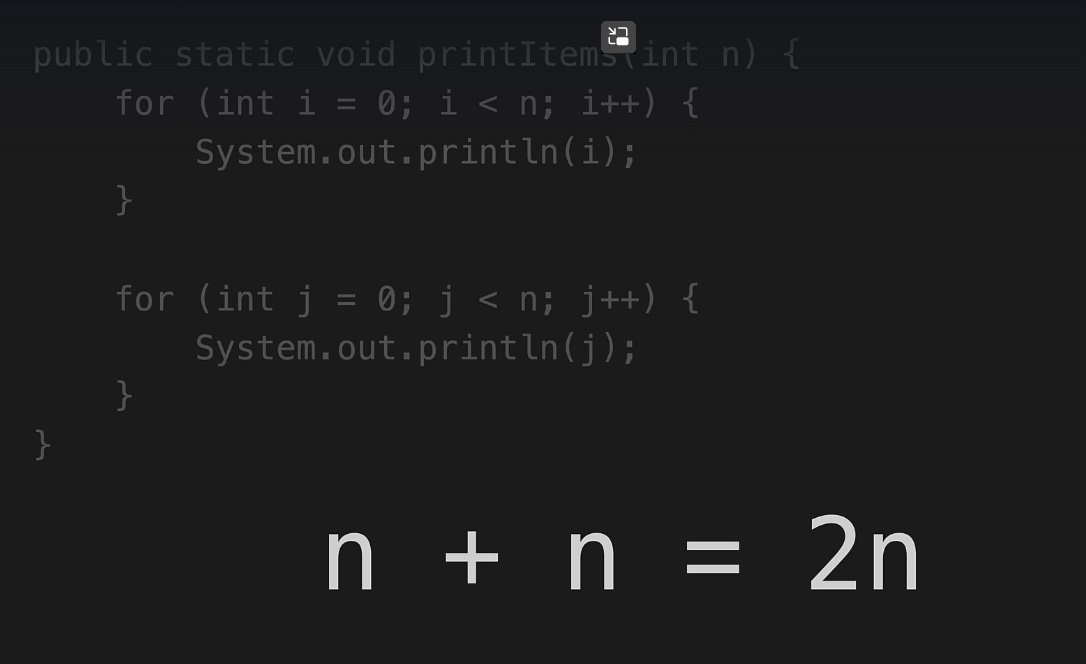
* Big O este un mod de a compara doua parti de cod
* Sa zicem ca avem 2 coduri, Cod1 si Cod2 care fac acelasi lucru. Trebuie cumva sa le comparam sa stim care e mai bun
* **Time Complexity** – este o marime de a masura rata de crestere a operatiilor a unui algoritm, dar e interesant ca nu se masoara in unitati de timp, deoarece pe diferite calculatoare timpul ar fi diferit.
* Time Complexity masoara numarul de operatii
* **Space Complexity** – masoara cantitatea de memorie pe care o ocupa un cod cand e executat
* 

Sa zicem ca vrem sa cautam un element in acest array cu un for loop

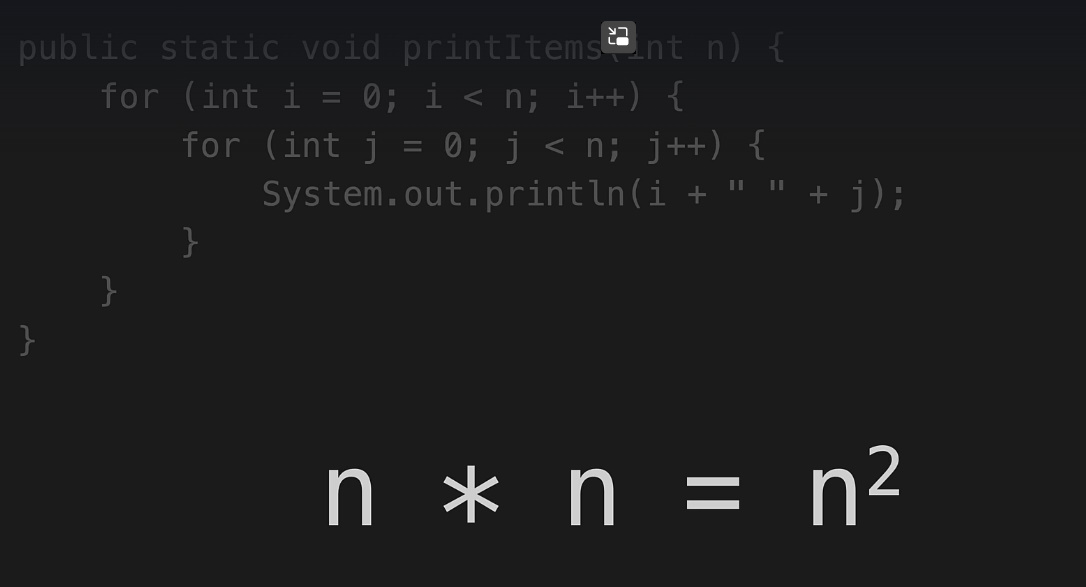
**Best Case(Omega)** – cel mai bun caz e cand avem nevoie de elementul 1

**Average Case(Teta)** – cazul mijlociu e cand cautam elementul 4, din mijloc

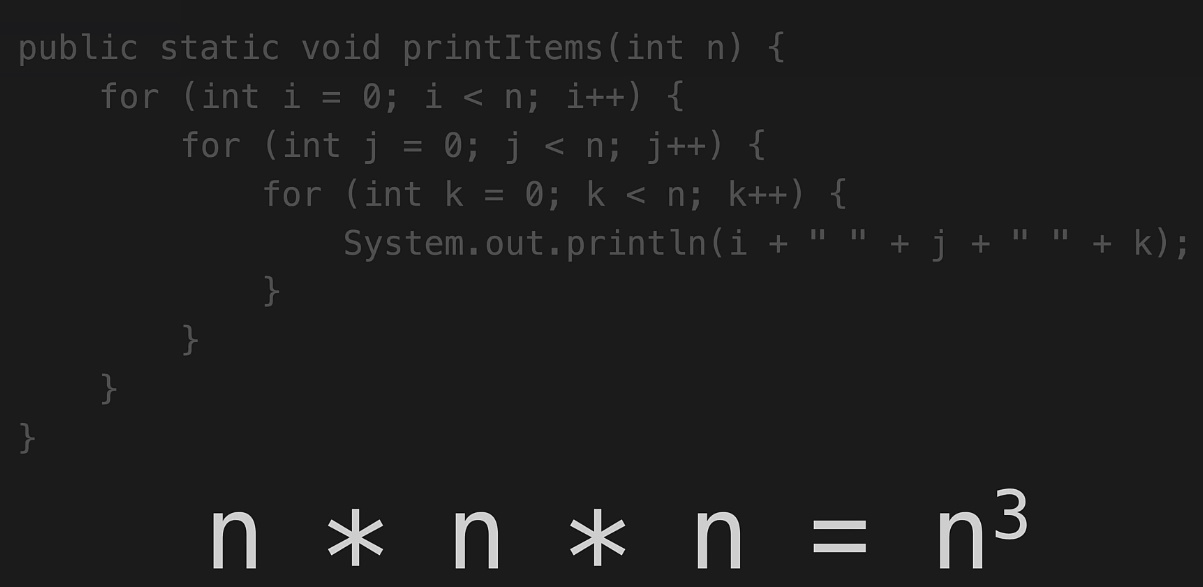
**Worst Case(Omicron sau O)** – cazul cel mai rau e cand cautam elementul 7

* 
* 

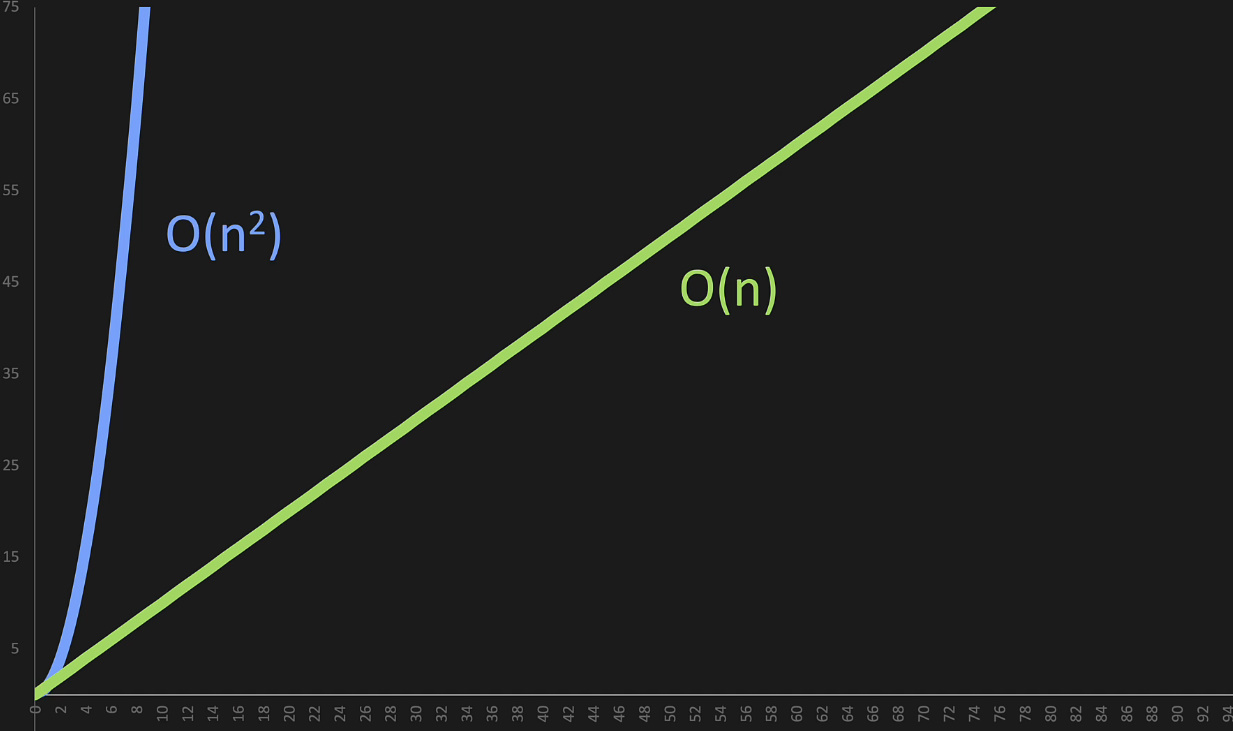
Dar eliminam constantele, si O(n) = n

* 

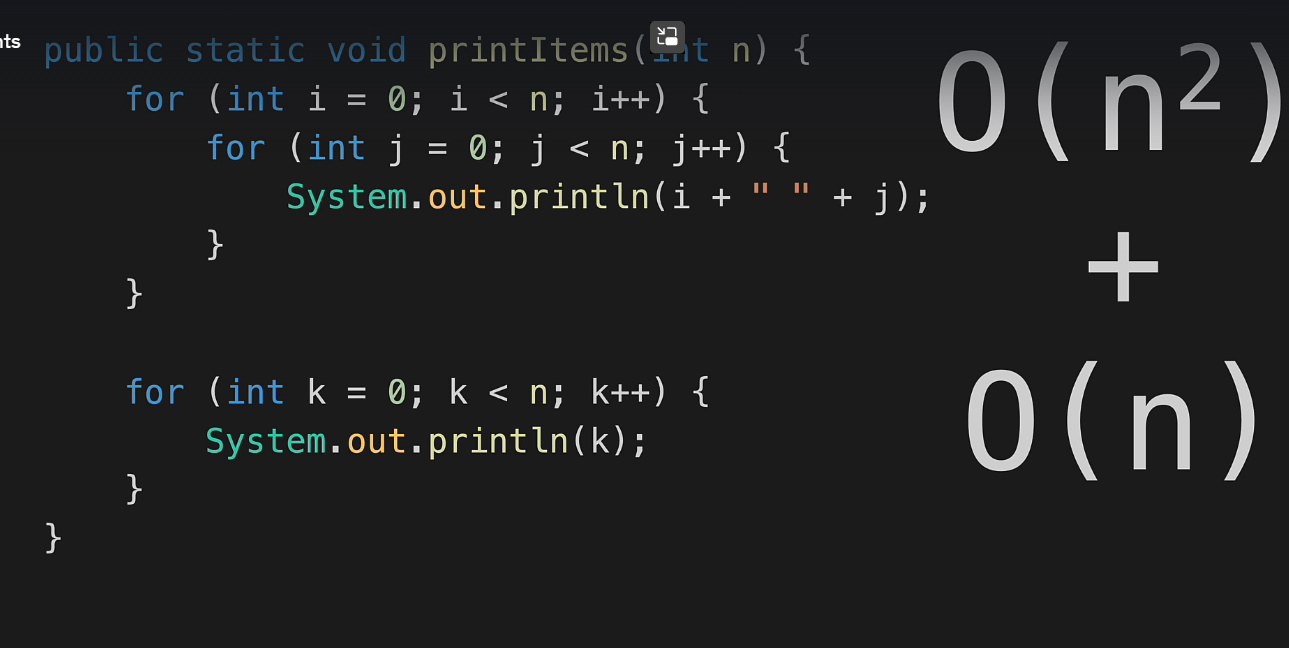
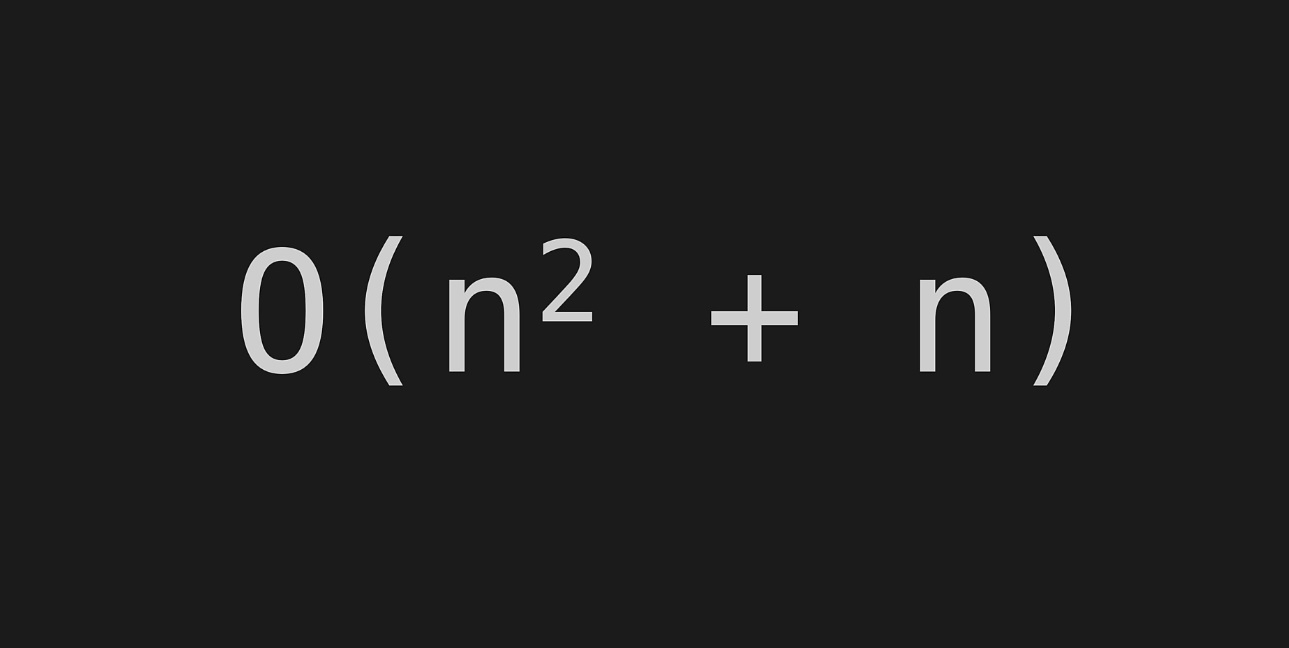
O(n) = n^2

* 

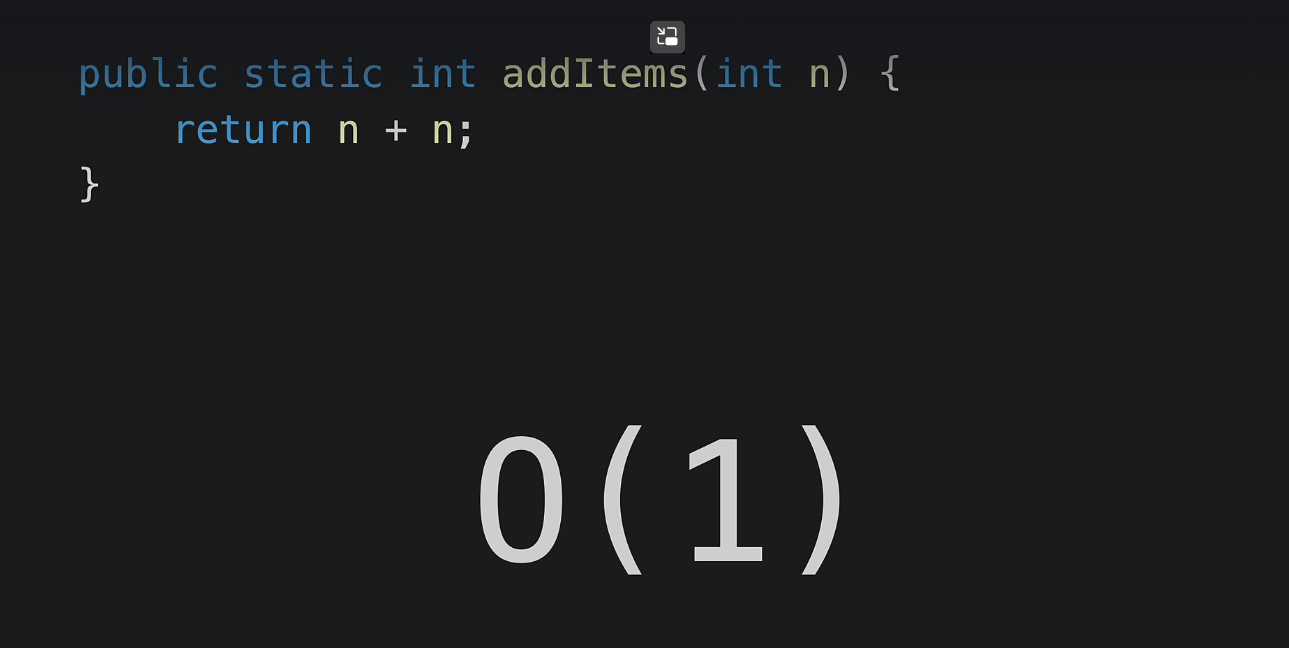
O(n) = n^2 deoarece simplificam puterea la n^2 mereu

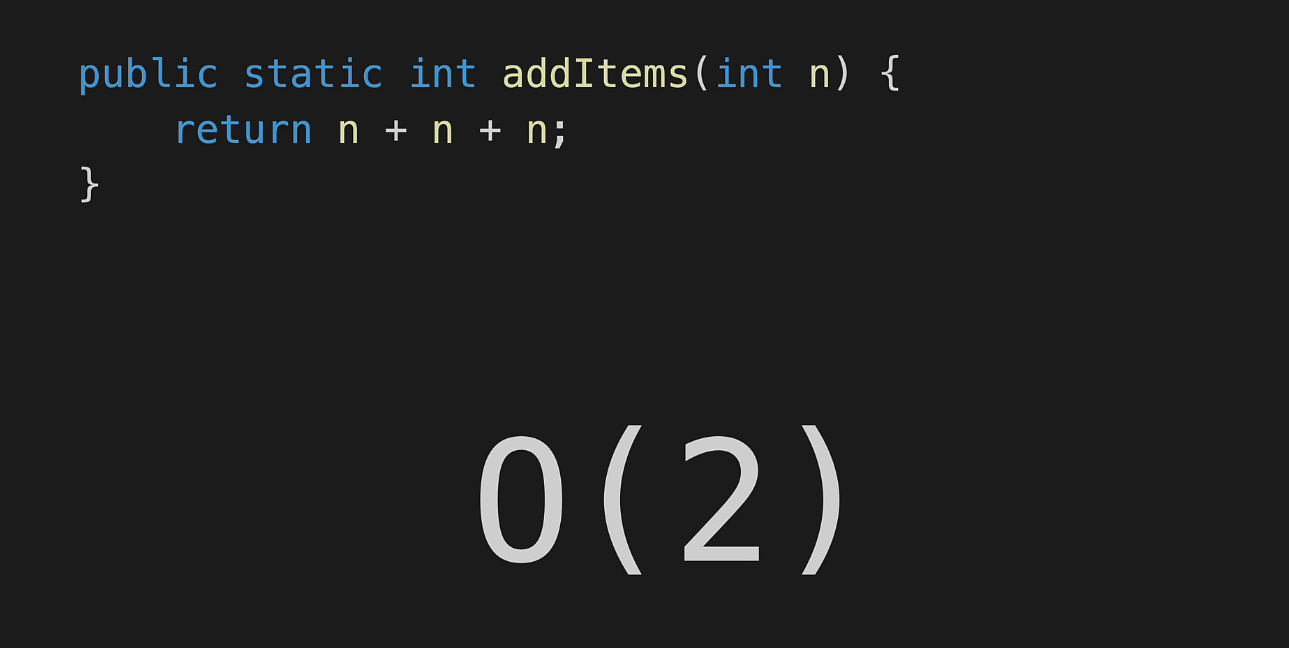
* 

Graficul ne arata cat de mult cresc operatiile din cod la cresterea datelor

* 
* 

Dar n^2 creste mult mai repede, de aceea il eliminam pe n. Cu cat n e mai mare, cu atat n^1 va fi mai mic ca n^2, si va fi nesemnificativ

* 



O(constanta) = 1, deoarece numarul de operatii nu creste la marimea lui n, de aceea si se considera 1

* Acum sa zicem ca avem un array:

1 2 3 4 5 6 7 8

Si sa zicem ca vrem sa cautam un element,gen 8. Sa cream un for ce parcurge tot array nu e cea mai rapida metoda, caci va fi O(n)

* Putem in schimb sa tot divizam arrayul in 2 si sa vedem in care parte se poate situa elementul:

1 2 3 4 5 6 7 8

5 6 7 8

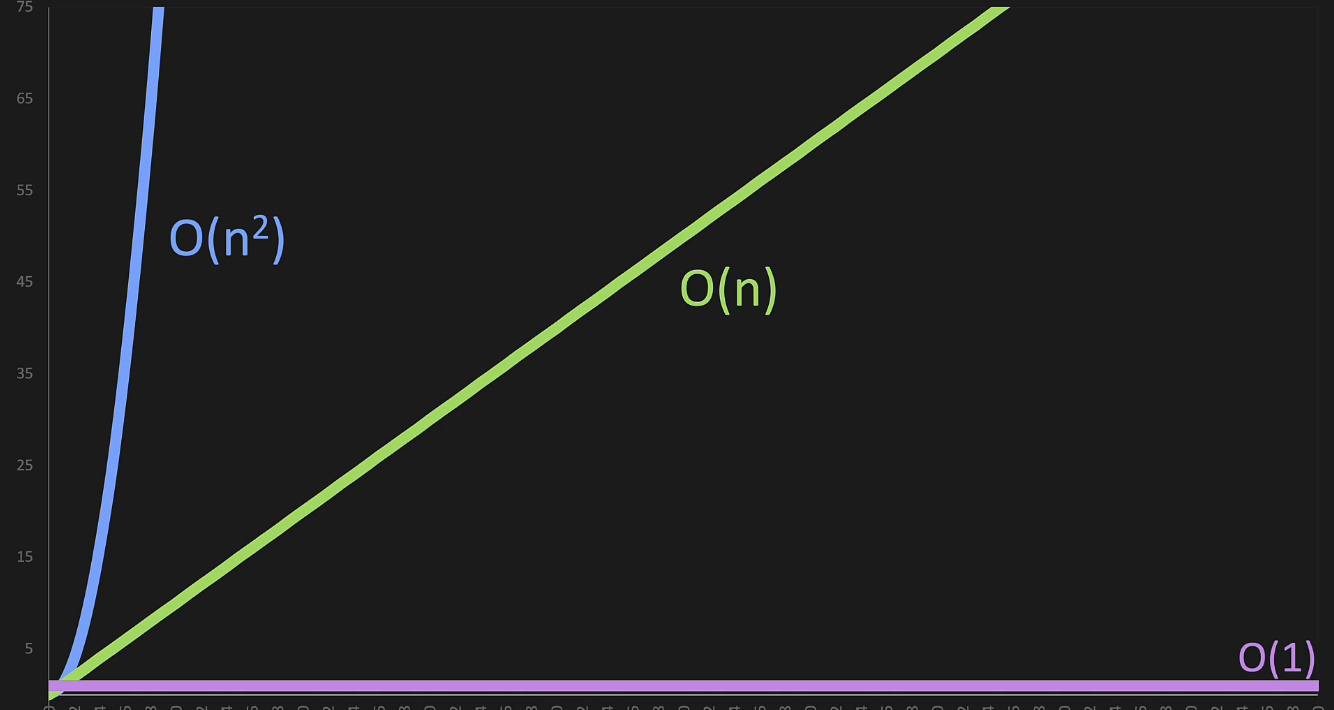
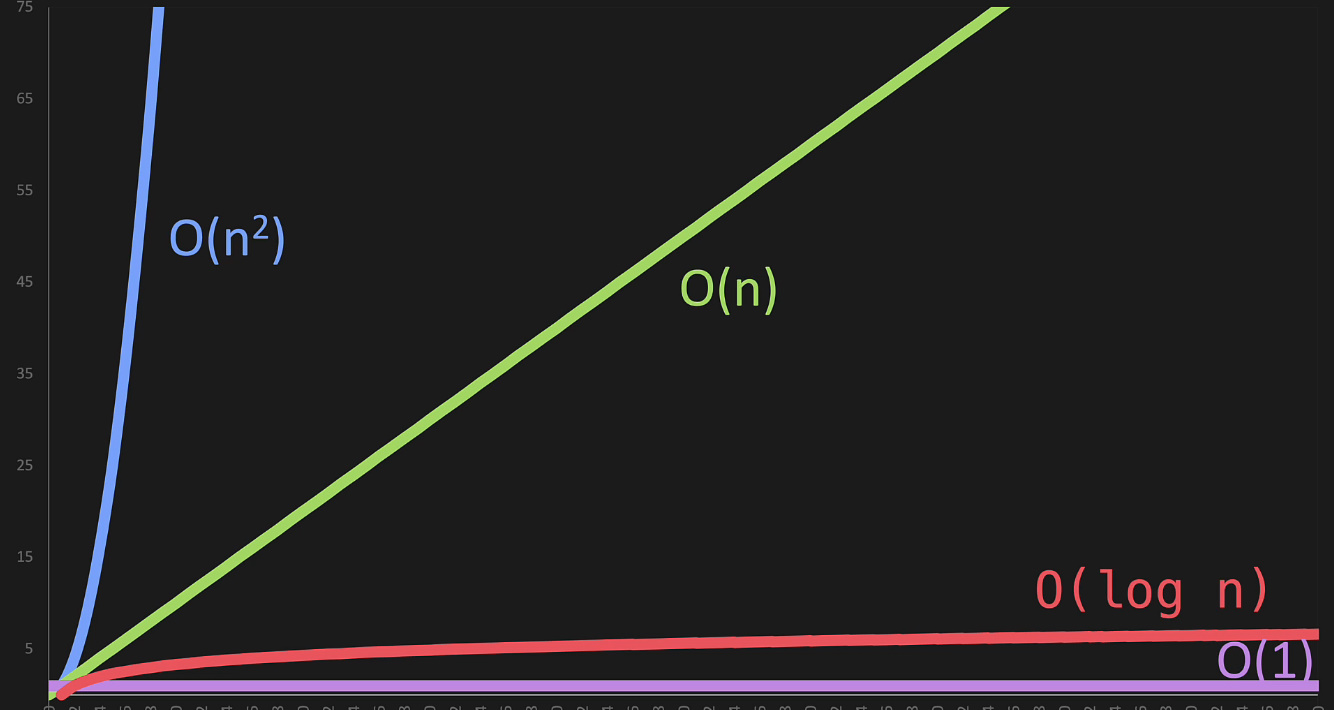
5 6 7 8

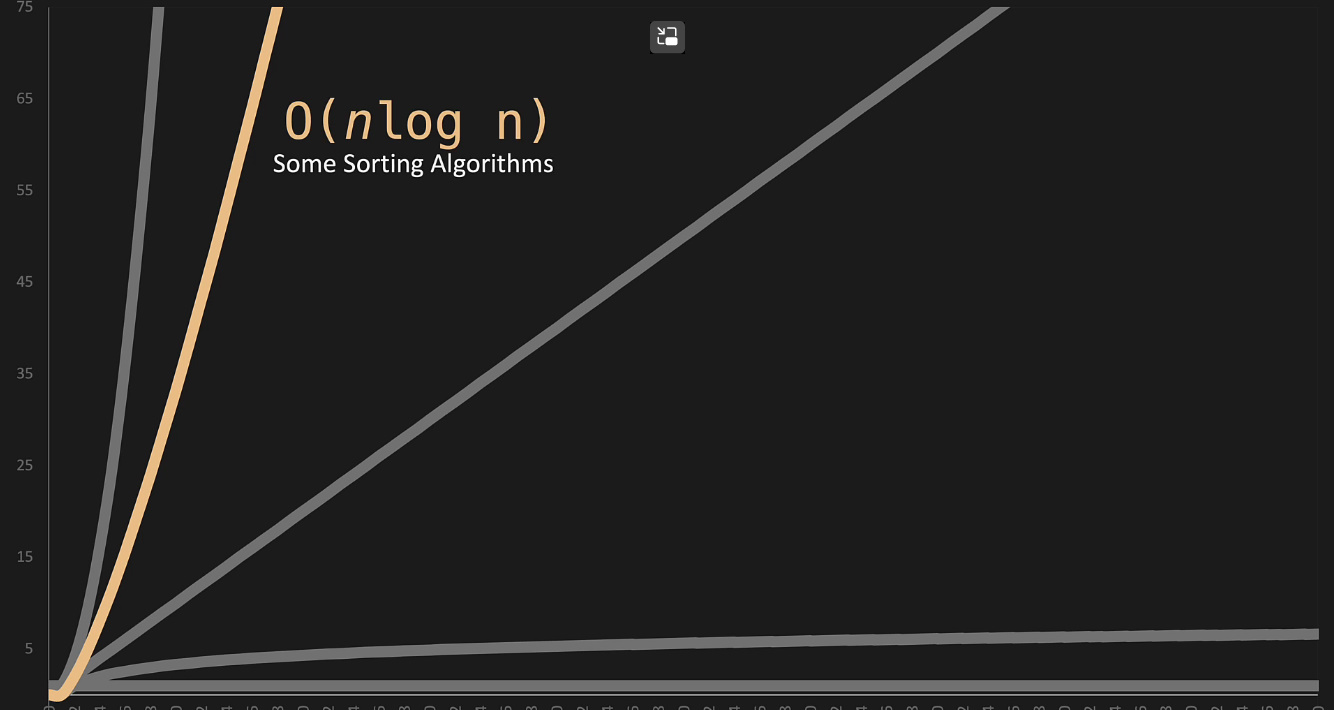
7 8

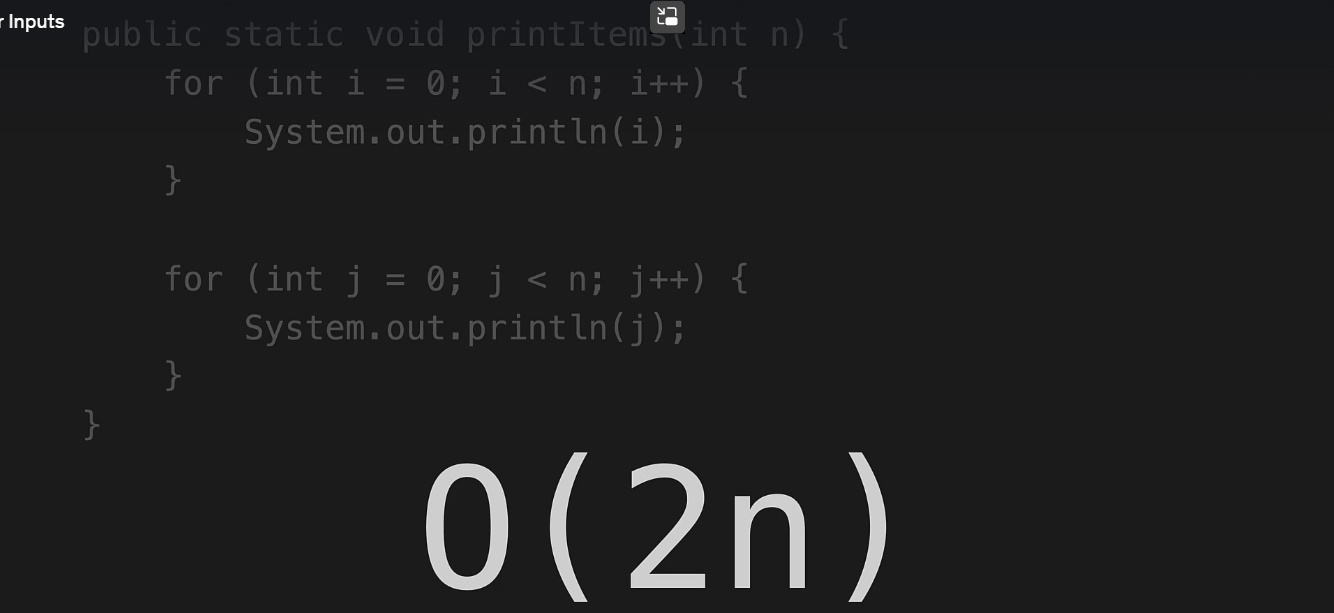
7 8

Aici, avem O(logn)

Nu punem baza, deoarece si asa e mereu baza 2, asa cum si puterea mereu se ia 2, indiferent cat de mare nu aar fi

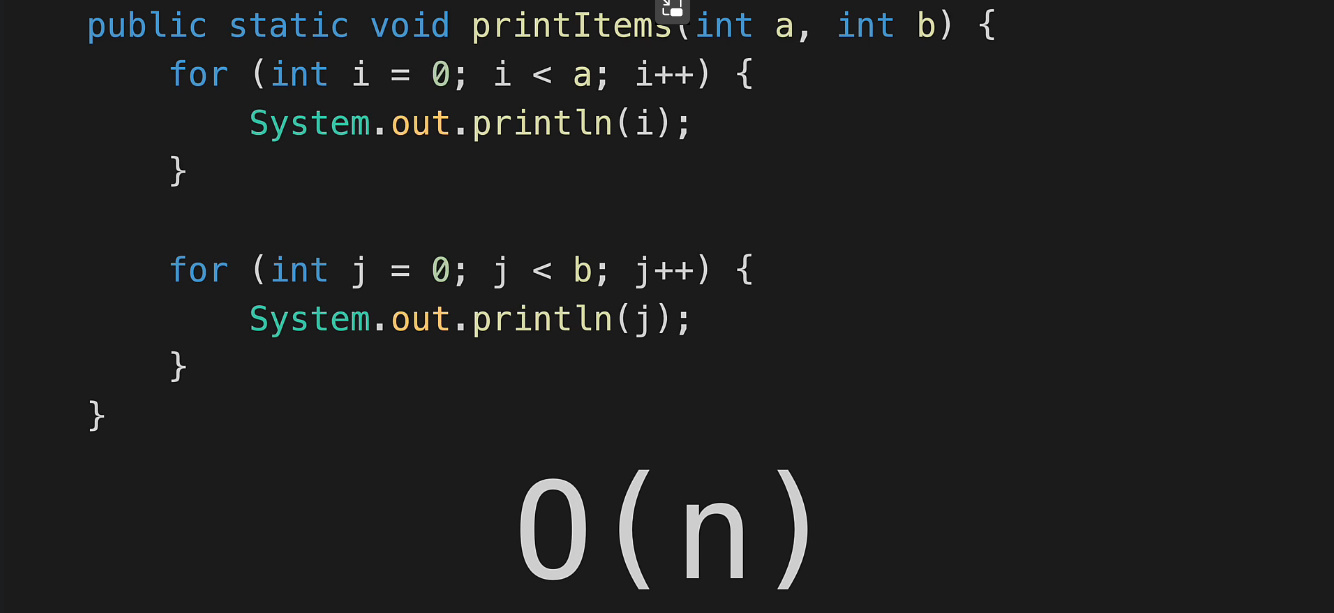
* 
* 



* 

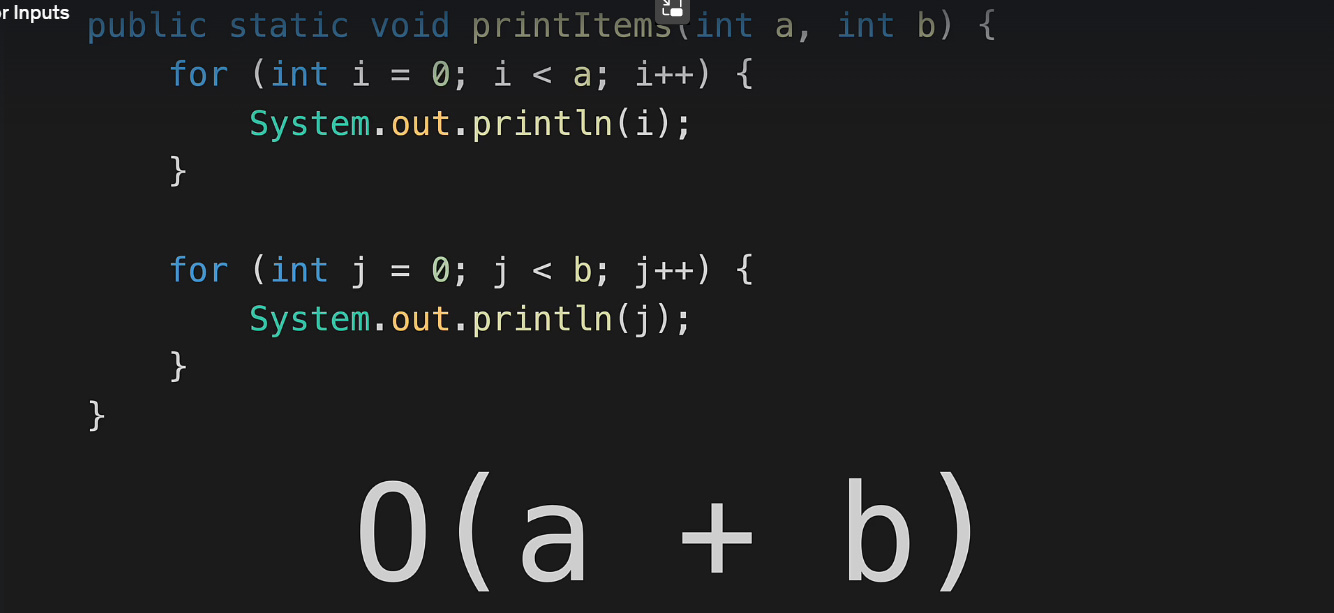


* Atentie insa la asa situatie:(**Interview**)



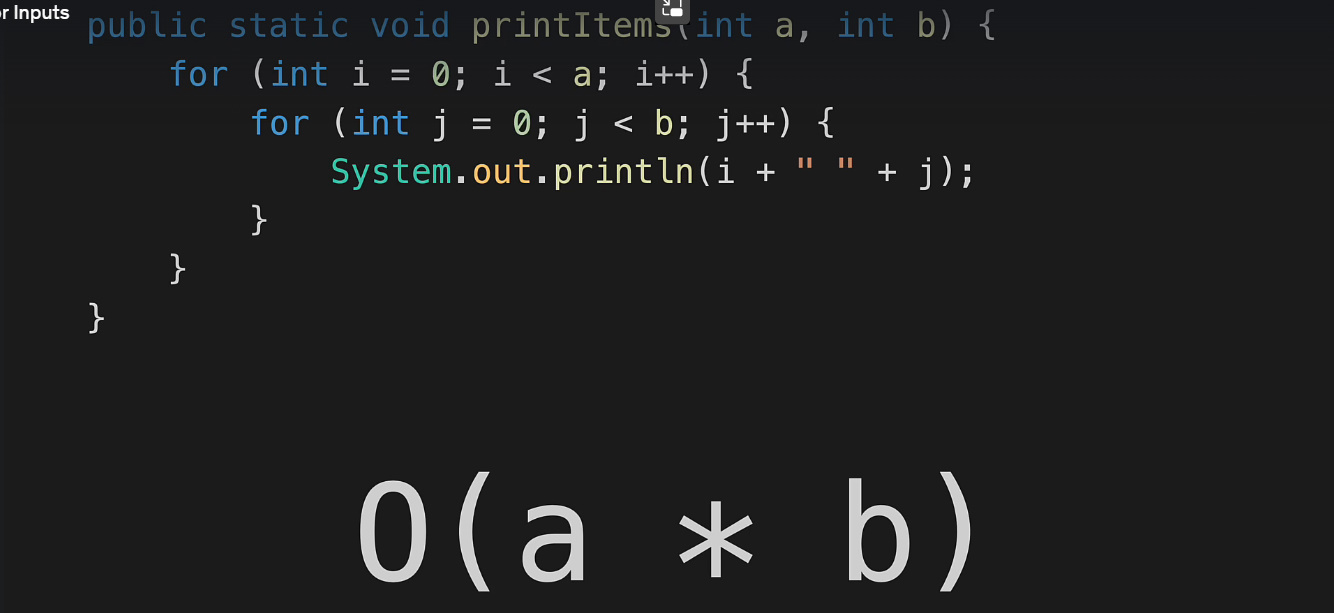


**Dar O(n) nu e corect!!!!**

****

**Mai mult de atat nu putem simplifica.**

**a si b ar putea avea valori ce diferentiaza cu milioanele si fiecare for poate avea complexitate diferita inca**



**Liste**

* Sa zicem ca avem lista:

list = 1 5 7 10

* Procesul de adaugare a unui element la final:

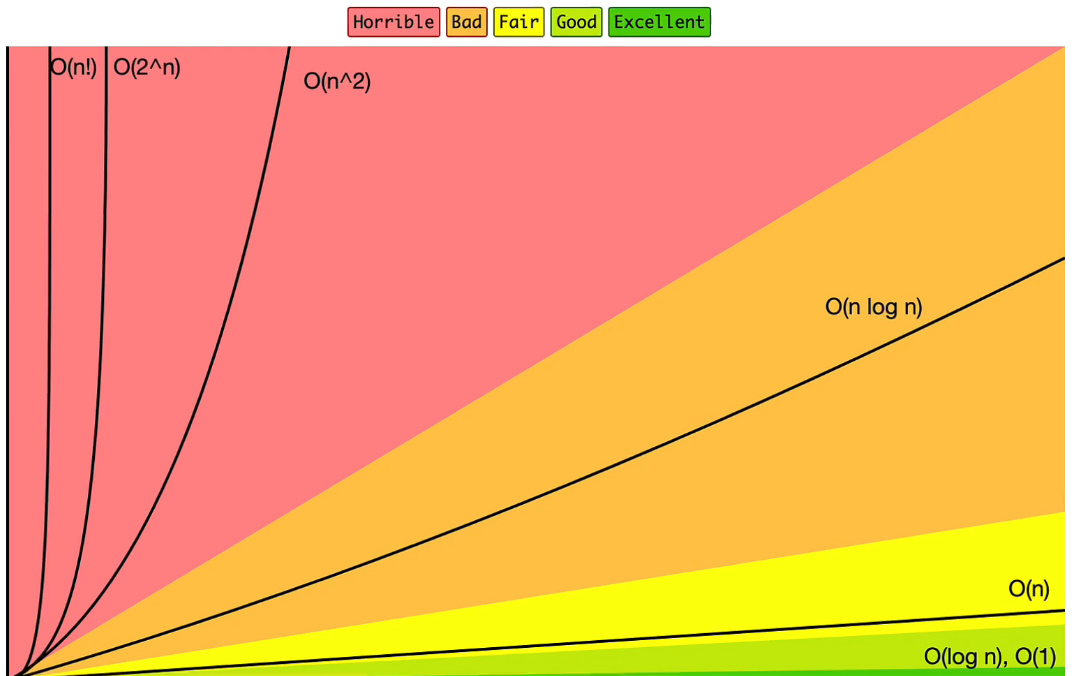
list.add(element) – O(1) deoarece elementul pur si simplu e pus la final si atat, si asa e mereu

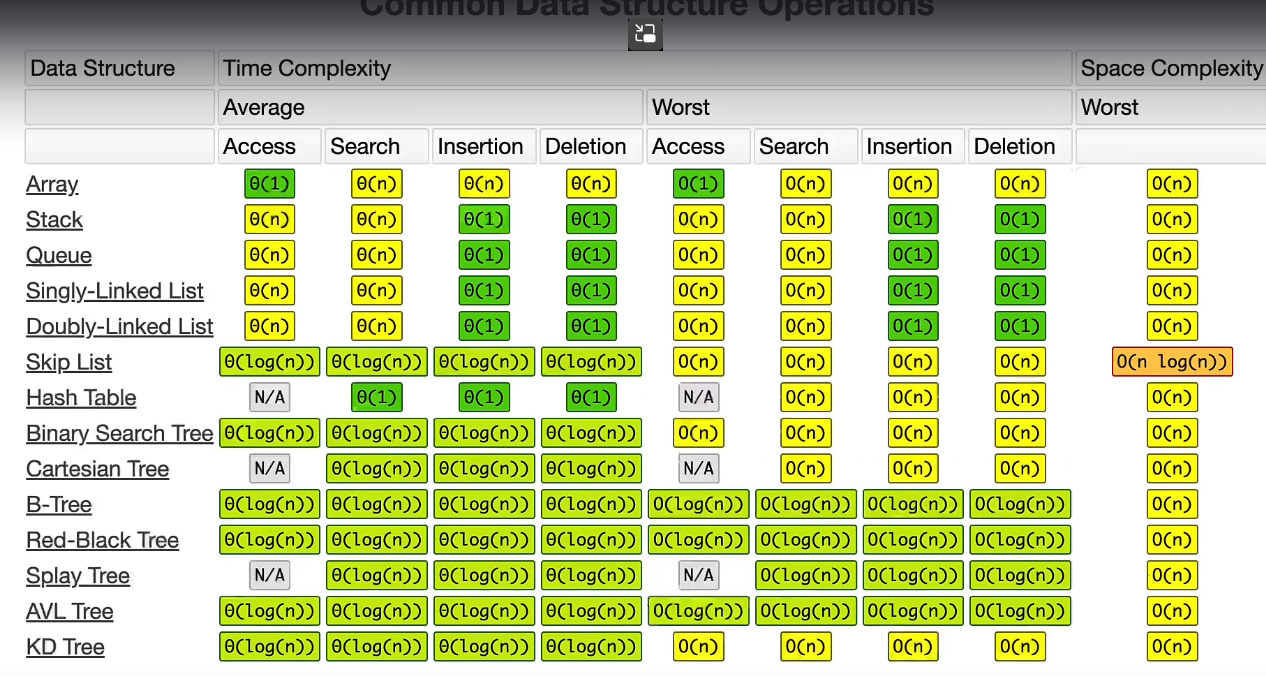
* Dar procesul de stergere a unui element, nu mai e O(1), deoarece, stergerea de ex a lui 7 va duce la accea ca elementele trebuie sa isi modifice si ele pozitiile, ca sa nu ramana spatiul lui 7 gol,si in asa caz avem O(n)
* Si adaugarea pe o pozitie e O(n)

list.add(1,11) deoarece iarasi sunt elemente ce trebuie sa isi modifice indexii, si daca e list.add(0,11) – toate trebuie

**Terminologii**

* O(n^2) – loop inside a loop
* O(n) – proportional
* O(logn) – divide and conquer
* O(1) – constant





Aproape toate structurile de date au aceeasi space complexity

