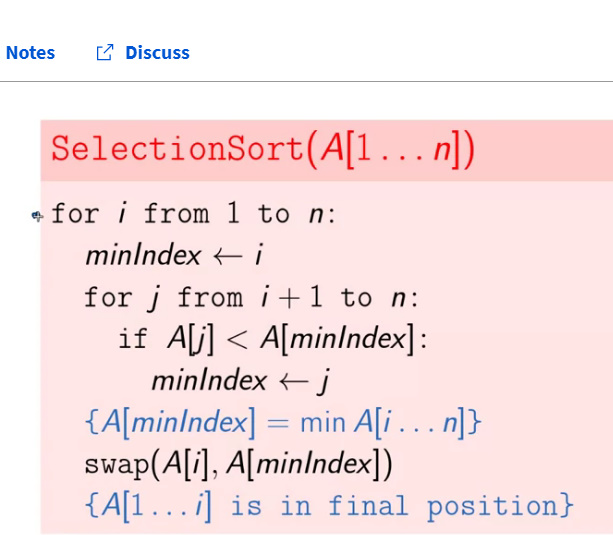
**Selection Sort**

Se bazeaza pe faptul ca gasim cel mai mic element din array si il punem pe primul loc, apoi gasim urmatorul cel mai mic element si il punem pe a 2 pozitie si tot asa.



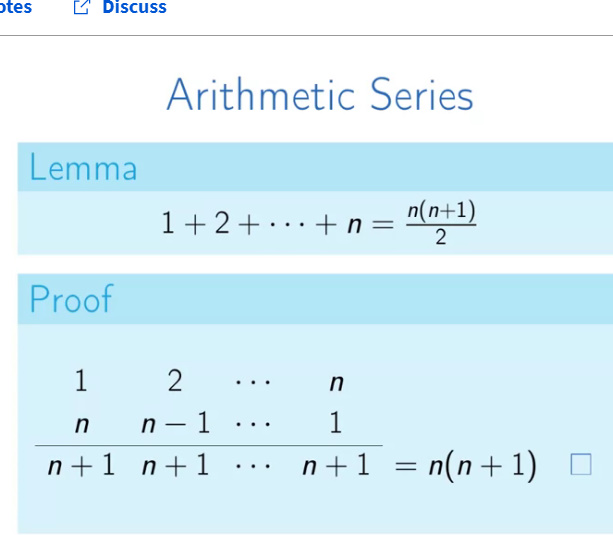
Running time al acestui algoritm nu depinde de datele introduse, caci ordinea lor din array nu conteaza, asa cum oricum array va fi de fiecara data parcurs de la pozitia i pana la final, unde i tot va creste si va creste. Depinde doar de marimea lui

O(n^2)

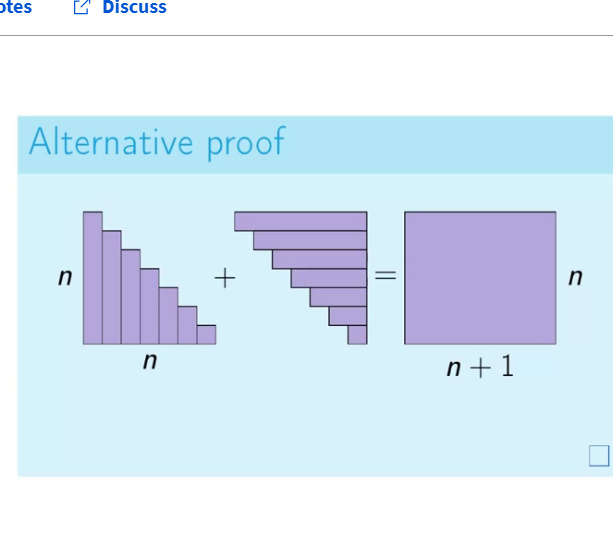
Desi, numarul de nested for se va executa de fiecare data cu 1 mai putin, totusi lasam O(n^2)

Demonastrare:

In primul rand nu exista formula concreta pentru a arata runtime la acest algoritm.

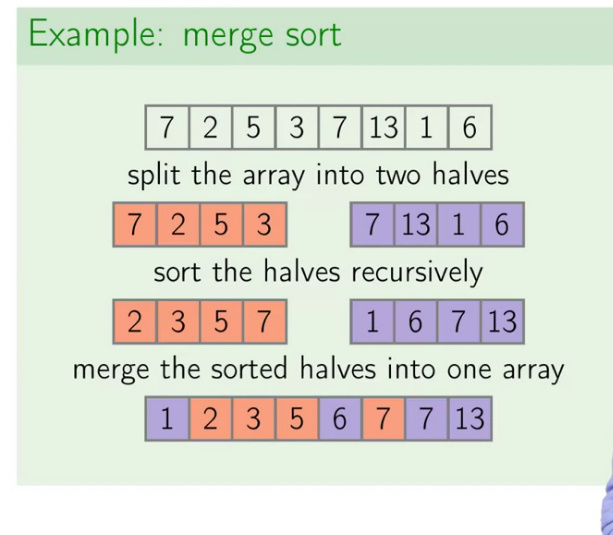


Dar, asa cum asymptotic notation ignora constantele, aici ignoram ½ si +1 si avem n\*n



**Merge Sort**

Luam un array, il impartim in 2 parti, cream 2 array separate din cele 2 parti si le sortam. Dupa, le vom uni, dar ne bazam pe faptul ca elementele din ambele array deja sunt sortate. Deci, luam primul element dintr-un array(x de ex) si il comparam cu primul din celalalt array(y de ex). Cream un array gol cu noul array sortat. Daca primul element din array1 x < y din al 2 array, in array nou,x se pune la urma, si deci dispare din array1. Urmatorul element dupa x iar e comparat cu y, si daca e mai mic, iar se duce la urma de array, si deci il va impinge pe elementul care deja era final cu o pozitie inainte, si deja el va fi ultimul. Asa elementele se vor impinge si vor ajunge la stanga, deci cele mai mici vor fi in fata.



2 3 5 7 1 6 7 13 array= [0 0 0 0 0 0 0 0]

1<2,da deci array = [0 0 0 0 0 0 1]

2 3 5 7 6 7 13

6<2, nu deci, array = [0 0 0 0 0 0 1 2]

3 5 7 6 7 13

6<3, nu, deci array=[0 0 0 0 0 1 2 3]

5 7 6 7 13

6<5,, nu deci array=[0 0 0 0 1 2 3 5]

7 6 7 13

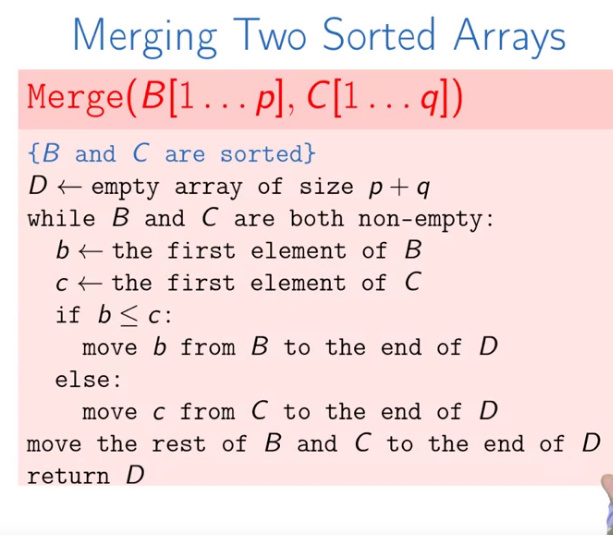
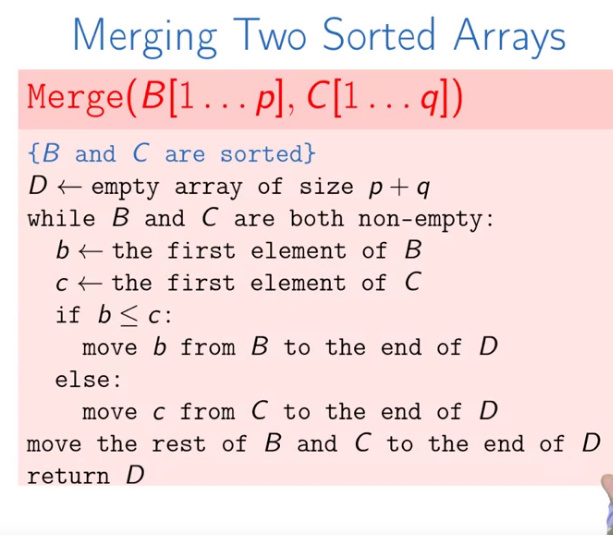
6<7, da, deci array=[0 0 0 1 2 3 5 6]

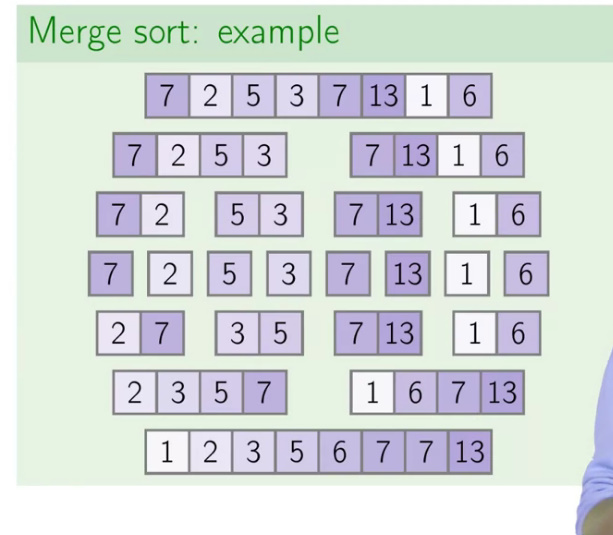
7 7 13

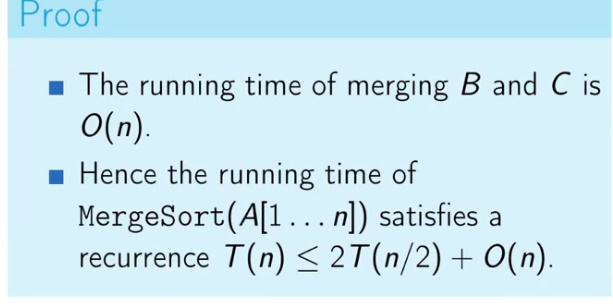
7<7,nu, deci array = [0 0 1 2 3 5 6 7]

.. 7 13

Asa cum array1 e gol, pur si simplu il punem pe array 2 la final si le impingem pe restu.







O(n) – runtime pentru a le uni

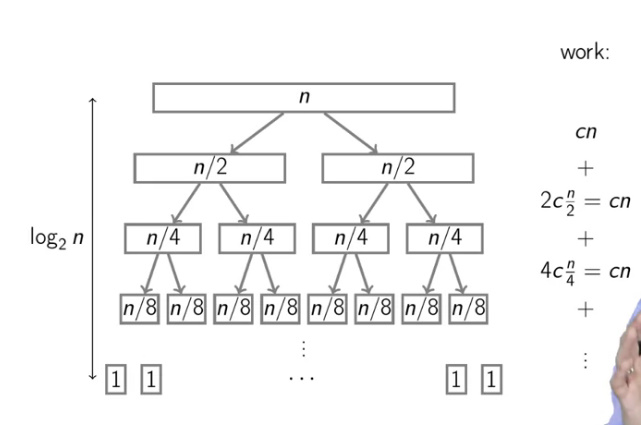
2T(n/2) – corespunde cu 2 chemari recursive ce se fac mereu

d = 1, a = 2, b = 2

d = log2(2)

O(n) = nlogn

In plus, fiecare iteratie ar avea O(n) = logn si ar fi logn + logn + logn ... si se ia cele care ajung la urma, unde returneaza 1, si se va face de n ori

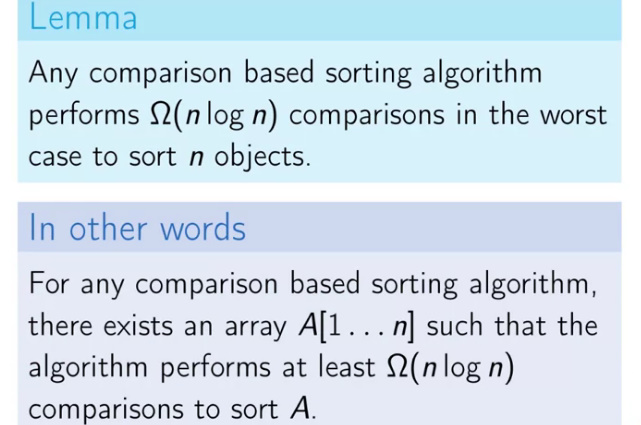


Vedem ca socotim doar elementele de la final, adica alea cu n/8 si ele se repeta de n ori, de acolo si nlogn

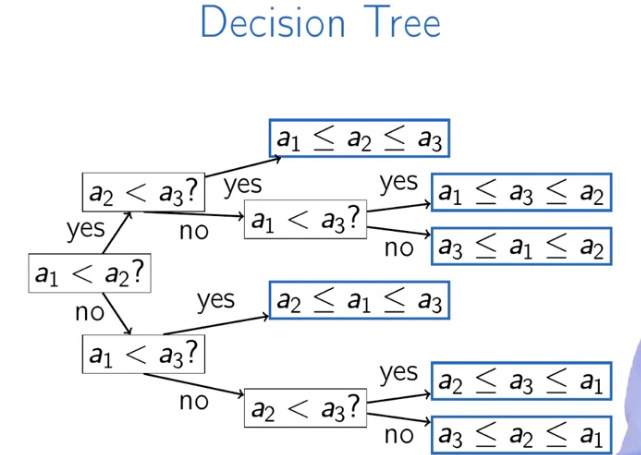
**O(nlogn) e un timp suficient. Deci acest runtime este optim.**

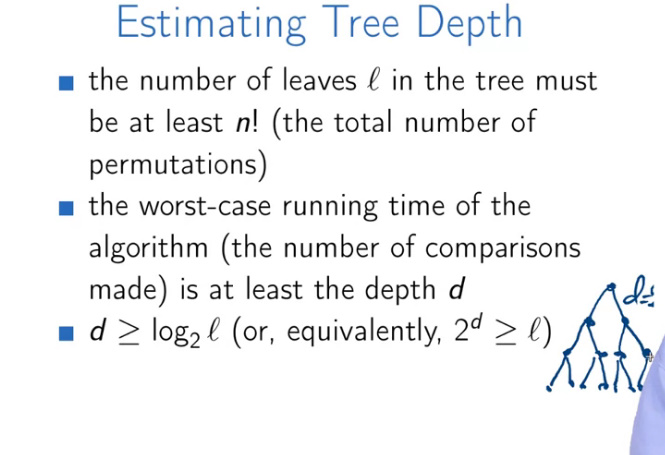
**Algoritmi bazati pe sortare**

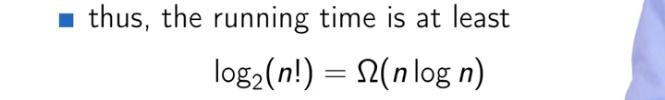
* sunt algoritmi care sorteaza bazanduse pe compararea a cate 2 elemente.
* Acest algoritm, pentru a fi optim, trebuie sa faca in cel putin **nlogn** comparari.

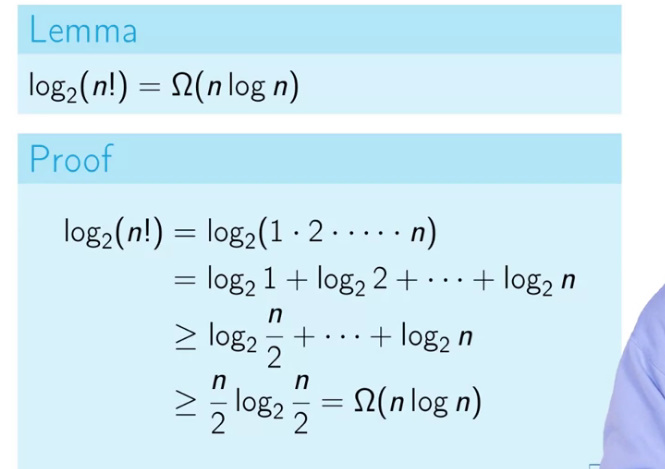


* Indiferent cate elemente are acest array, algoritmul nu poate nicidecum face mai putin de nlog n comparari, de asta e omega(nlog n) si nu O





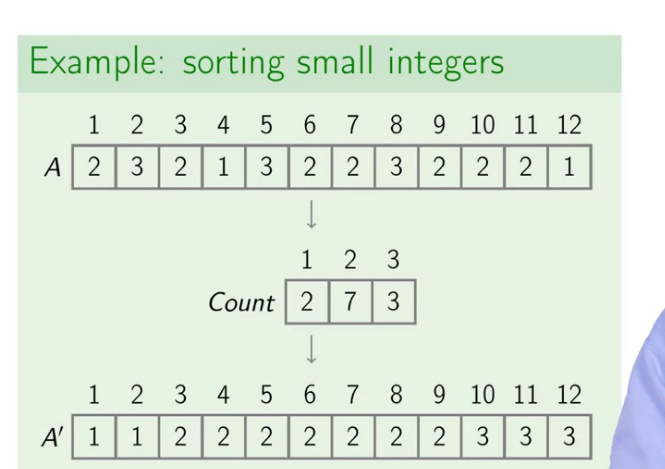


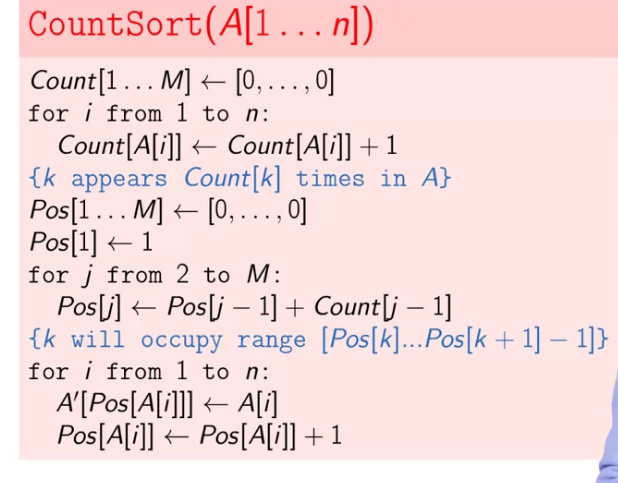


**Algoritmi de sortare care nu se bazeaza pe comparare**

* acest algoritm nu compara fiecare elementele intre ele.
* poate fi folosit pentru arrays cu date putine.
* Deobicei e bun cand datele se repeta, de ex;

**Counting sort alghoritm**





O(n)

Atentie! Aici vom incepe indecsii de la 1, nu 0!

1. Pentru fiecare element, socotim de cate ori apare int-un array. Fiecare element va fi un index din alt array. De ex, daca avem:

2 1 5 5 2 5 5

Facem un array din 5 element, anume Count unde peste tot are 0

Count[2]++

Count[1]++

Count[5]++

Count[2]++

Count[5]++

Count[5]++

Count[5]++

Deci, numarul de aparitii al fiecarui numar e stocat in tabloul Count si rezultatul e in indexul acelui numar.

1. Putem sa avem si elemente ce nu apar de loc, de ex aici 3 nu apare niciodata, si nici 4, deci Count[3] = 0 si Count[4] = 0

Vrem cumva sa stim fiecare element pana la ce pozitie va aparea. Facem un array Pos

Pentru a afla ce pozitie va ocupa acel nr in array final, pur si simplu adunam prima valaore cu a doua si apoi a doua cu a treia si tot asa. De ex, Count acum e asa:

Count = [1 2 0 0 4]

In Pos vom pune pana la ce pozitie apare fiecare numar. Ne asiguram ca Pos[1] = 1, ca sa putem aduna cu ceva, caci e pozitia initiala, desi am lasa 0 in cod, asa cum index incepe cu 0 in realitate.

Deci vom avea:

Pos[2] = Pos[1] + Count[1] = 1 + 1 = 2

Deci, elementul 1, care e index in Count, si deci nr in Array, va aparea de la pozitia 1 pana la 2

Pos[3] = Pos[2] + Count[2] = 2 + 2 = 4, deci 2 va aparea pana la pozitia 4(nu si la a 4 a )

Pos[4] = Pos[3] + Count[3] = 4 + 0 = 4

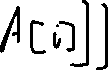
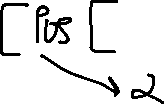
Pos[5] = 4 + 0 = 4

Pos[6] = 4 + 4 = 8, deci 4 apare pana la pozitia 8(exclusiv 8)

Pos = [1 2 4 4 4 8]

1. Acum, facem asa: fiecare element are pozitia de la care incepe si la care se termina. Insa, ne intereseaza aia la care incepe.

Fie Nou noul array deja cu elementele puse.



Incrementam Pos[A[i]] ca atunci cand acest element va accesa din nou acest element, sa stie ca pozitia lui nu va mai fi 2 si deja 3

