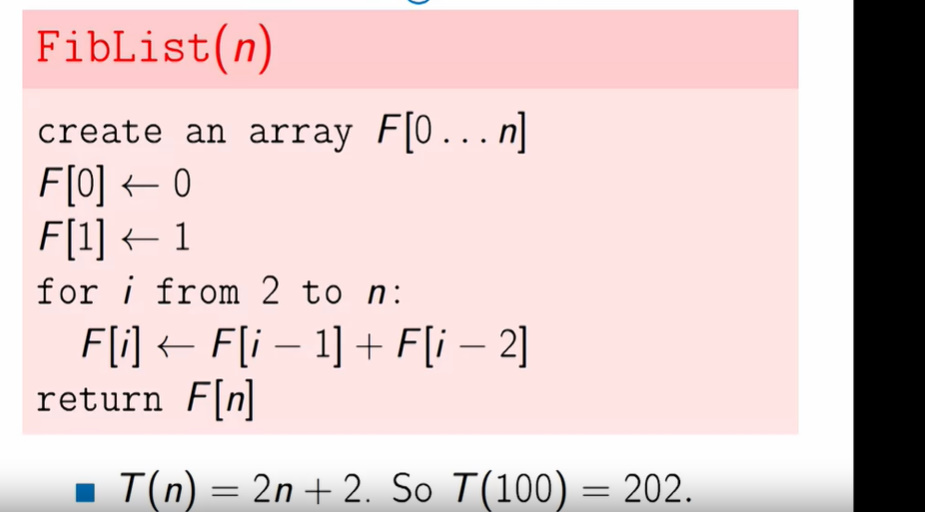
[Big-O Notation: Plots | Coursera](https://www.coursera.org/learn/algorithmic-toolbox/ungradedLab/Qmv41/big-o-notation-plots/lab?path=%2Fnotebooks%2Fbigo.ipynb)

[Big-θ (Big-Theta) notation (article) | Khan Academy](https://www.khanacademy.org/computing/computer-science/algorithms/asymptotic-notation/a/big-big-theta-notation)

**Fibonacci Alghoritm**

* metoda recusrsiva de a rezolva aceasta problema este foarte lenta si rea.
* Insa putem crea o lista de dimensiunea data de noi, punem 0 si 1 ca primele 2, si apoi fiecare nou element va fi suma elementelor de pe cele 2 pozitii din urma.
* Pentru a afla cat de eficient e algoritmul, trebuie sa calculam numarul de linii care le va executa.



Deci, oricare ar fi n ce vrem sa il aflam, numarul de linii executate va fi:

Aici avem T(n), relatie de recurenta, nu O(n)

T(n) = 2n + 2;

In primul rand, putem deodata sa vedem ca vor fi 4 instructiuni:

1. create an array F[0...n]
2. F[0] <-- 0
3. F[1] <-- 1
4. return F[n]

In bucla for, se vor executa pentru fiecare iteratie cate 2 linii, adica for(incrementare,comparare etc.) si din for chiar. Nu uitam ca e si al 0 nr, deci plus inca un nr va mai fi, de ex la n = 5 vor fi 6 numere si in bucla for n+1, cu tot cu 0.

Insa, T(n) != 2n + 2 +4 != 2n + 6;

Asta din cauza ca, pentru n=0 si n=1 , bucla for nu se va executa, adica ea va incepe sa execute cu n = 2, asa cum pentru n = 0 si n = 1, nu avem ce calcula, ci incepem sa calculam elementul al 2 din lista:

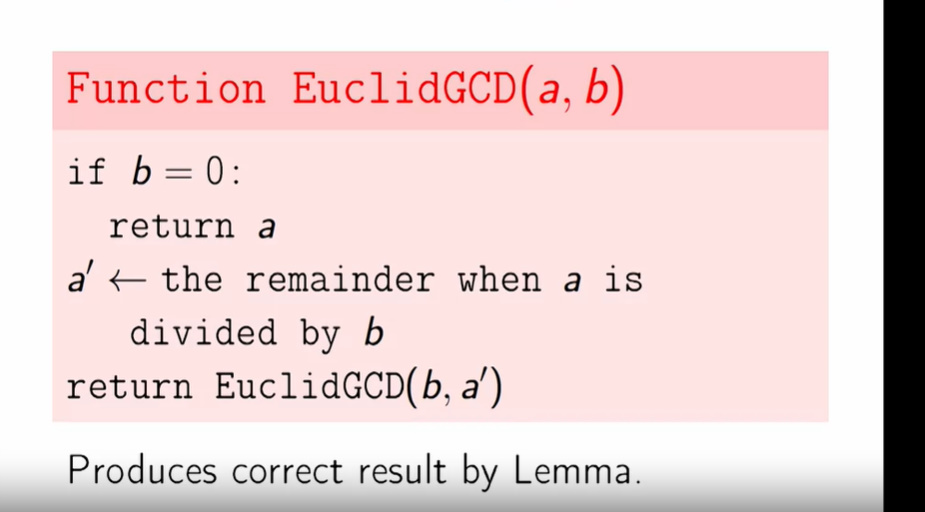
0 1 1 2 3 5.....

vedem ca nu putem incepe bucla cu 0, adica primul element, caci 0 nu are cu ce se aduna din urma si am avea index negativ, de aceea putem elimina cele 2 operatii pentru elementul ce va fi omis si pentru 1 e la fel, caci 1 nu se obtine din nimic.

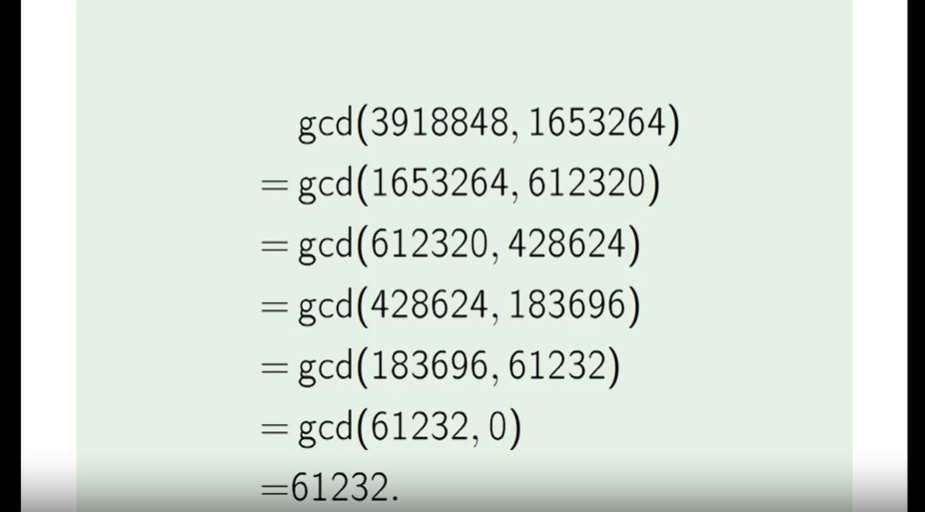
Avem T(n) = 2n + 6 – 4 = 2n + 2;

**Cel mai mare divizor comun**

* Sa zicem ca avem 2 numere a si b si vrem sa gasim cel mai mare divizor comun al lor.
* O solutie naiva ar fi sa cautam toate numerele posibile pana ajungem la cel mai mic element din cele 2, insa asta ar lua foarte mult timp.
* Un algoritm mai bun poate fi facut asa:
* fie a\* restul impartarii lui a/b
* se stie ca cel mai mare divizor a lui a si b este si cel mai mare divizor a lui a\* si b, respectiv b si a\*



Deci, defiecare data vom trimite la functie valoarea b si restul impartarii. De ex:



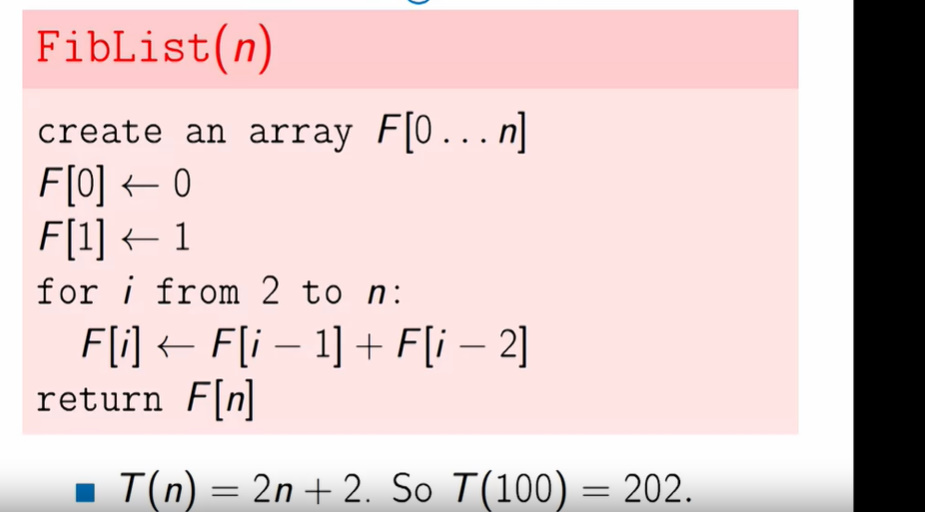
612320 este restul 3918848/1653264

**Complexitatea algoritmului**

* Complexitatea algoritmului depinde de 2 factori:
* Time Factor – durata de timp necesara pentru executie
* Space Factor – memoria maxima necesara pentru a stoca datele

**Calcularea la RunTimes**

* Calcularea la RunTimes prevede sa calculam numarul de linii sau instructiuni executate de algoritm si sa aflam deci timpul lor de executie.
* Aflarea numarului exact de pasi poate fi dificila si problematica.
* De ex:

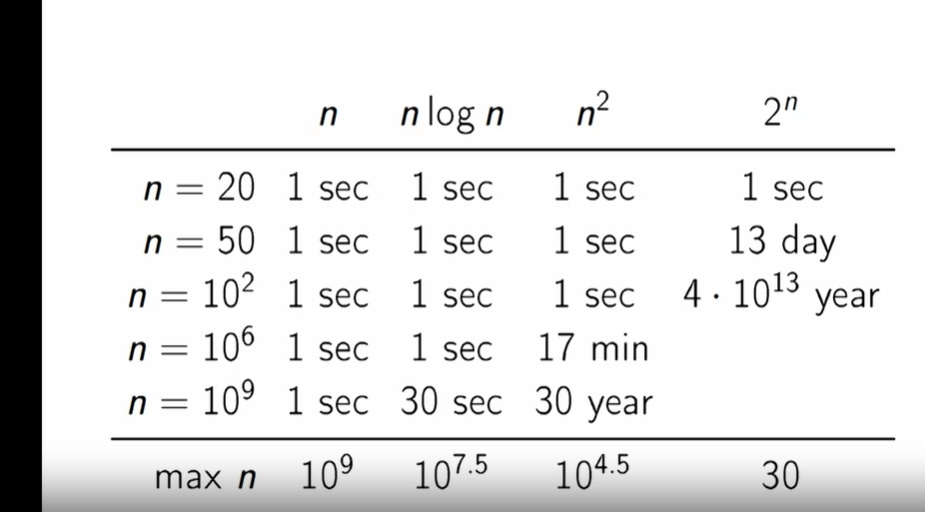


Noi socotim ca crearea la array F[0...n] e o operatie, un rand, dar asta poate defapt lua mai mult. Pot fi diferite cosntrangeri, operatii suplimentare etc. In plus, calculatorul, chiar si pentru linia F[0] = 0 nu va face o singura operatie, ci mai multe. Avem obtinerea pointerului, asignarea valorii etc. Bucla for compara, incrementeaza, si iar nu e o operatie doar. Apoi, fiecare operatie din buclar for va fi obtinerea elementelor de pe pozitiile date, adunarea si asignarea, si iar nu e o operatie doar.

* Sa nu uitam ca **depistarea timpului** de compilare e si mai greu de prezis, caci aici deja depinde de CPU, RAM, operatiile de optimizare si multe altele
* Deci, nu putem calcula exact timpul de executie al algoritmului. Totusi, trebuie cumva sa putem masura eficienta lui.
* Deci, metodele de analiza a algoritmului nu se bazeaza pe aflarea la runtime acestuia. Nu putem nicidecum afla acest timp de executie. Exista unele metode de masurare a runtime, dar ele nu se bazeaza pe masurarea timpului.

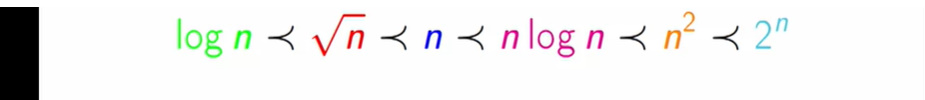
**Asymptotic Notation**

* Arata **cum** **ar varia timpul de executie la cresterea cantitatii de date**,
* Schema de mai jos ne arata doar cum ar varia timpul de executie pentru diferite valori ale lui in n in diferite situatii:



deci 2^n e cel mai rau ce putem avea.

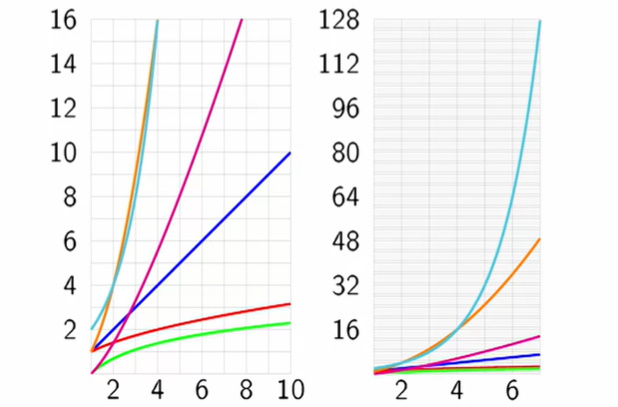
Ordinea creascatoare:



Deci, constanta din fata, ca 100n sau 1000n e mai putin importanta ca puterea sau logaritmul de ex. Constantele nu conteaza prea mult.

De ex, diferenta intre n si n^2 e mult mai mare decat orice constanta.

* Totusi, contantele nu trebuie ignorate. Insa, ne ocupam de ele abea dupa ce ne asiguram ca algoritmul are un bun Asymptotic Runtime



* Asympotic annotation afla performanta algoritmului pe baza cantitatii de date introduse in el. Nu calculam de cat timp are nevoie, dar **cum** **ar varia timpul la cresterea cantitatii de date**, desi oricum vom numi rezultatul runtimes, dar **Asymptotic RuntTimes.**
* Graficele astea nu ne ajuta cu nimic la determinarea timpului de rulare, dar ne permit sa cmparam algoritmii, pentru a vedea care e mai performant, la care rata de crestere a timpului in raport cu datele e mai mica. **Deci, e mai mult pentru comparare.**
* Ea ignora toate constantele ce ar fi necesare in dependenta de specificatiile la pc pentru a afla real runtimes.
* **Asimptotic** – se refera la valori mari!

**Fixed and Variable part**

* Un algoritm are o parte fixa si una variabila.
* De ex, daca avem un algoritm pentru crearea unui array manual, ar fi cam asa:

1. Oferim un numar de elemente de la tastatura
2. Cream array
3. Setam elementele in bucla for

Numarul de pasi ar fi n^2 + 2

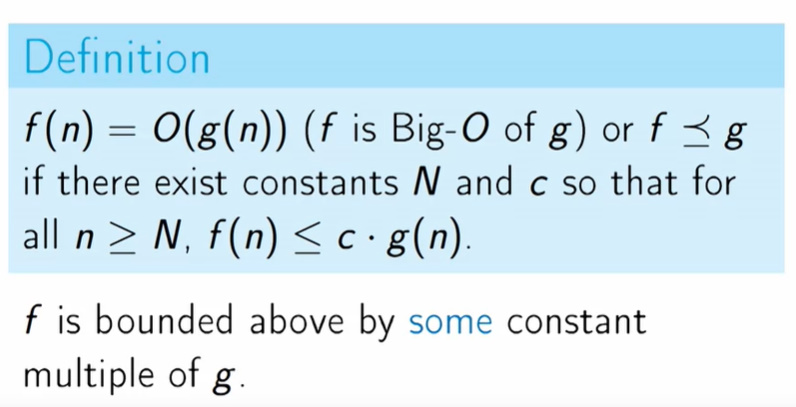
2 – e partea fixa

n – partea variabila

Aici vedem ca acest + 2 nu joaca un rol prea mare.

**Big O notation**

* Big O notation este un tip de asymtotic notation cel mai des utilizat.
* Se bazeaza pe worst-case, adica ia mereu numarul maxim de operatii ce pot avea loc.



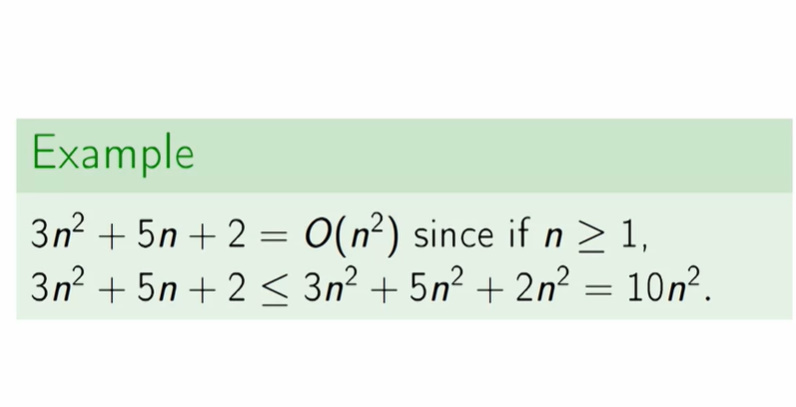
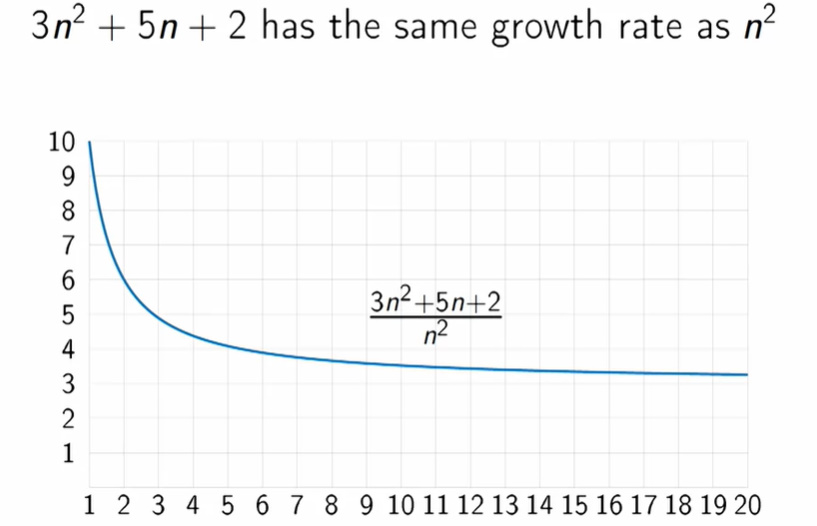
* f(n) = O(g(n)) (f este Big O a lui g) daca exista o constanta N destul de mare si o constanta c asa incat n>=N( pentru valori foarte mari ale lui n) avem f(n) <= c\*g(n) , adica functia f(n) **va creste nu mai repede ca** g(n) chiar daca langa constante s-ar pune g(n), asa cum constantele nu influenteaza rata de crestere.

De ex:

f(n) = 5n^2 + 7n + 10

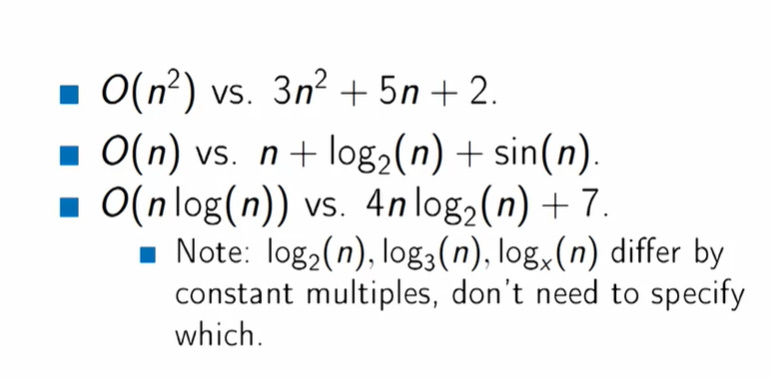
O(g(n)) = n^2

f(n) < =5n^2 + 7n^2 + 10n^2

* sau f(n) este Big O a lui g(n) daca pentru valorile destul de mari, f(n) este marginit superior de anumite constante de timp fixate g(n)
* Asta inseamna ca folosind constantele din f(x) pentru fiecare g(x), g(x) ar putea sa il depaseasca pe f(x) la rata de crestere, insa f(x) nu. **Putem spune si ca, oricate instructiuni nu ar fi executate cu f(x), rata de crestere a timpul lor de executie oricum nu va intrece rata de crestere a timpului din g(x), asa cum g(x) ia rata maxima posibila de crestere a timpului in raport cu datele.** Vezi mai jos exemplu:
* 
* 

Vedem ca functiile 3n^2 + 5n + 2 si n^2 , desi la inceput au rate de crestere mari care difera, peste putin timp rata lor de crestere diferentiaza putin. Deci, in orice functii ce au, de ex, n^2, chiar daca mai e si n, ca 10n^ 2 + 8n + 6, rata de crestere e in mare parte determinata de n^2, si mai mult pe el si ne focusam.

* din grafic, f(x)/g(x) tinde spre 0, adica cu cat e mai mare valoarea, cu atat functiile date au rata de crestere mai putin diferita.



Deci in loc de 3n^2 + 5n + 2 putem scrie doar n^2. E mai clar si usor de lucrat.

In loc de n + log(n) + sin(n) putem scrie doar n.

In loc de 4nlog(n) + 7 scriem doar nlog(n), caci depinde mai mult de n, nu si de constanta 4.

Logaritm in baza 2,3,4,..., de n se scrie doar logn

O de orice constanta e 1, de ex 10 = O(1) 10000 = O(1)

* **Big O notation e mult mai buna, deoarece, desi nu permite sa calculam viteza de rulare a algoritmului, asa cum ea depinde de PC, deci de o constanta multipla, ea ne permite sa vedem cum se comporta algoritmii pentru diferite valori ale lui n, chiar si imense.**
* **Big O este doar asimptotic! El ne arata mai mult ce se intampla cand punem date foarte foarte mari in algoritmi diferiti.**
* **Big O ascunde constantele, de aceea pentru o valoare concreta nu spune cat timp va lua rularea algoritmului.**

**Reguli**

* Constantele de multiplicare se omit:

10n^2 = O(n^2)

(n^3)/10 = O(n^3)

* Cand avem mai multi n, il alegem pe cel care creste cel mai repede:

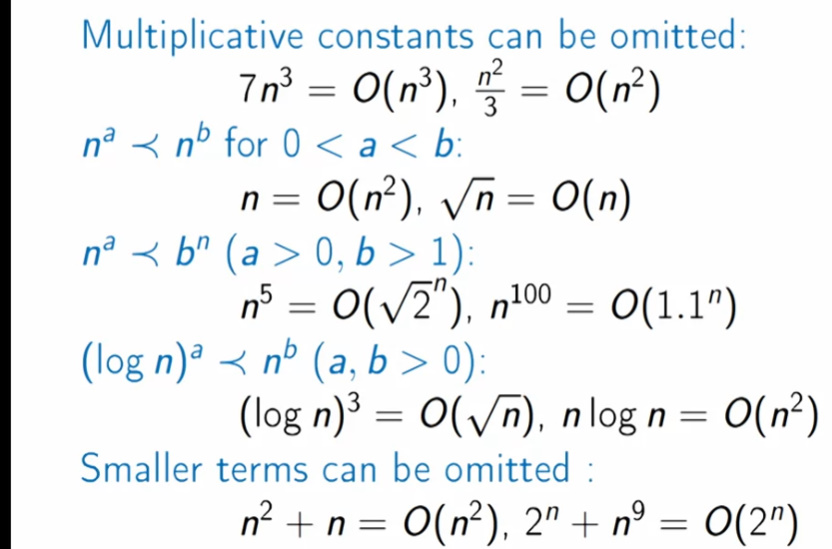
5n^3 + 10n^2 + 8n = O(n^3)

* Daca avem puteri in ambele parti, putem sa le simplificam:

x^2 = O(n^4) ---> x = O(n^2) – am simplificat cu n^2

* cand avem 2 expresii putere, creste mai repede cel cu baza mai mare:

n^2 creste mai lent ca n^5

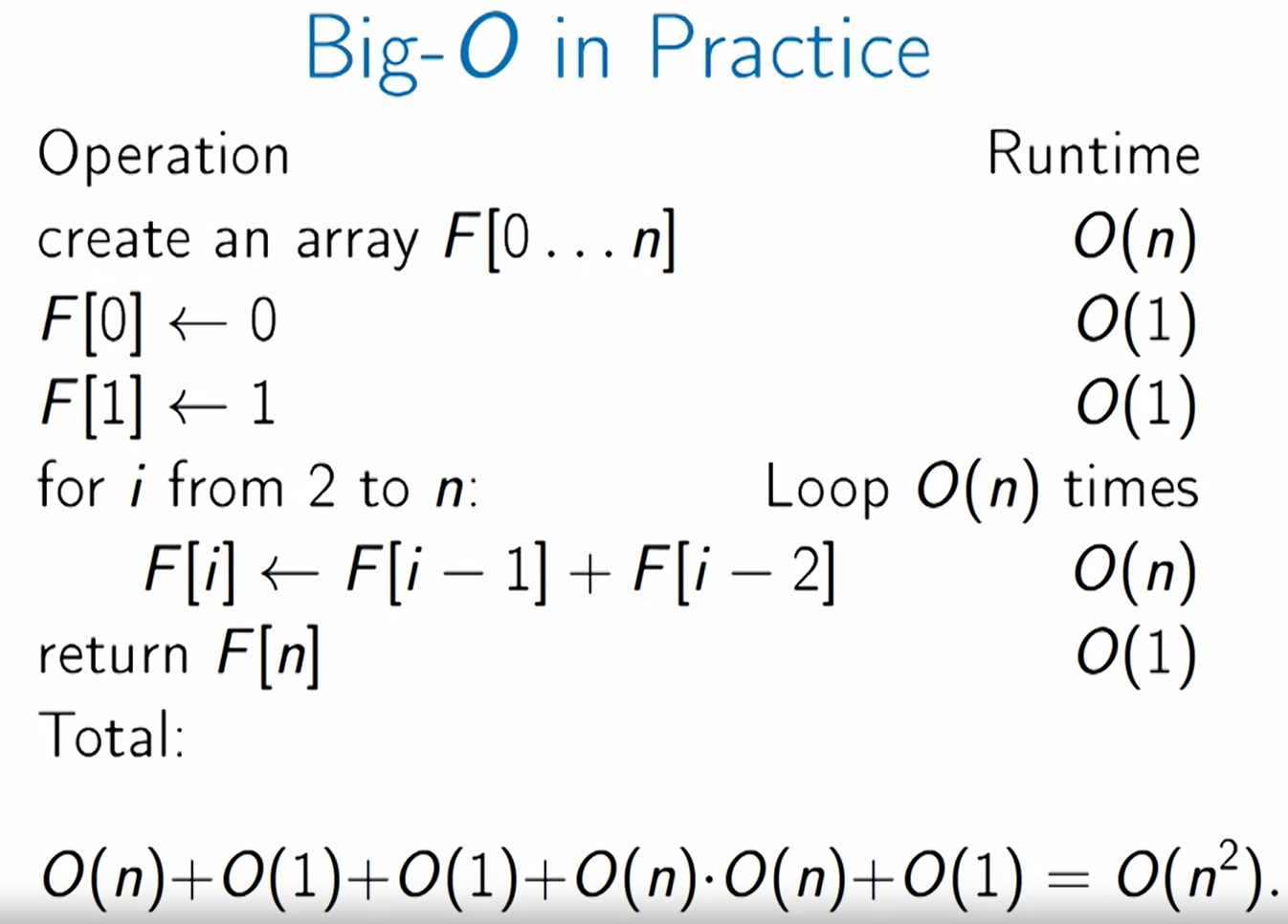


**= de aici inseamna ca functia din stanga nu creste mai repede ca cea din dreapta.**

* Orice functie polinomiala creste mai incet ca orice functie exponentiala:

n3≺2nn3≺2n, n10≺1.1n

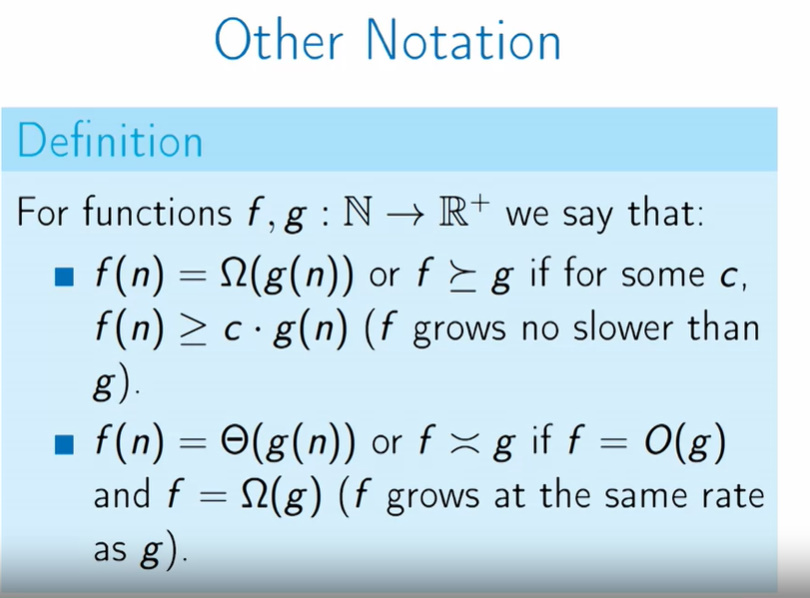
**Exemplu**

****

1. Crearea unui array poate implica mai multe operatii, chiar si de compilator, sau de ex setarea valorilor, dar fiecare celula va avea un nr constant de operatii, si va fi xn, deci vom avea O(n)
2. Forul face verificare, incrementare, ceva gen 2n la fiecare iteratie
3. Asignarea tot e xn
4. Instructiunile din for se executa tot de atatea ori cat si forul, deci n\*n

**Alte Notatii**

**Omega Notation si Teta Notation**



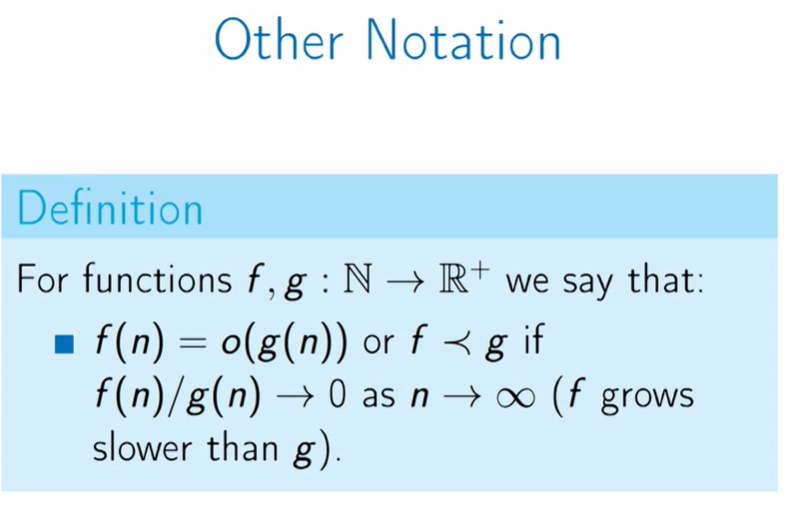
1. Prima spune ca f(n) **creste nu mai incet ca** g

se bazeaza pe best case. Arata anume numarul minim de operatii al unui algoritm.

se bazeaza pe faptul ca g(x) va varia mereu mai slab ca f(x)

1. A doua e Teta Notation si spune ca f(n) creste la fel ca g, adica f = O(f) si f = omega(g), adica f e si Teta si Omega si creste si nu mai repede ca g si nici mai lent, ci exact ca g.

**o Annotation**



spunem ca f creste strict mai putin ca g. Deci, se spune ca f(n) **creste strict mai incet** ca g. Deci rata de crestere a lui f(n) tinde la 0. Atentie! E exact ca big O, doar ca nu e <= ci doar <.

* este ceva intre O si Teta annotation

**Masurarea complexitatii algoritmului**

Pe baza notatiilor de mai sus, exista 3 cazuri de a analiza un algoritm:

* worst case – calculeaza numarul maxim de operatii ce pot fi executate in algoritm. De ex, daca dorim sa gasim un element intr-un array cu o bucla for, avem O(n) = n, dar pot fi si cazuri mai bune, asta e el mai rau posibil CEL MAI DES UTILIZATA
* best case – se calculeaza minimul de operatii. De ex, cel mai bun caz pentru problema de sus, cand folosim for, e cand caracterul e gasit din prima, deci O(n) = 1. FOARTE RAR UTILIZATA
* avergae case – luam toate iputurile posibile, calculam numarul lor de operatii si facem media. RAR UTILIZAT

**Confuzii**

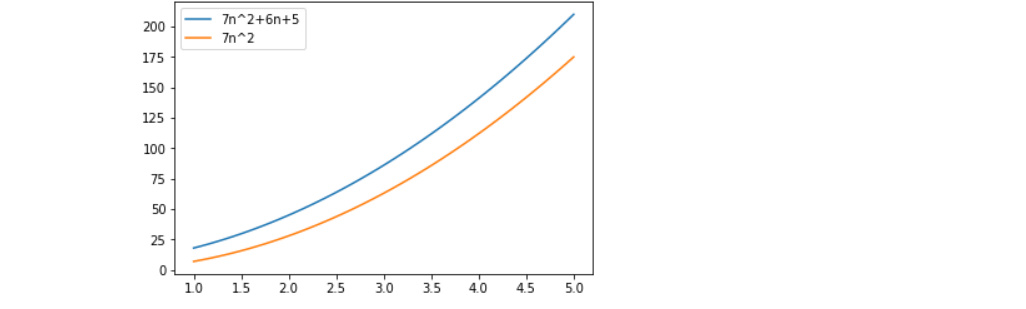
* 5n^3 = O(n^3) daaar 5n^3=O(n^4) sau 5n^3=O(n^10)

Big O spune doar ca f creste nu mai rapid ca g, deci mai sus am spus ca 5n^3 creste nu mai repede ca n^3 sau n^4 sau n^10. Toate sunt adevarate.

* Daca f<g atunci f<=g – daca f creste mai incet ca g, atunci cu siguranta creste nu mai repede ca g
* ≤ si ⪯. nu sunt la fel Caci ⪯ arata ca nu creste mai repede ca, dar primul arata valorile concrete. De ex:

5n2≰n25n2≰n2, but 5n2⪯n2

**Ex practice**



Vedem ca 7n^2 + 6n + 5 nu creste mai rapid ca 7n^2. Big O nu inseamna ca 7n^2 + 6n + 5 ar lua valori mai mici, ci ne intereseaza rata de crestere, si vedem ca e aceeasi. Insa la o mic, adica o(), rata de crestere trebuie sa fie strict mai mica, ceea ce aici nu este, caci e si rata egala de crestere.

Daca y e foarte mare, vedem ca valoarea lui 6n + 5 e nesemnificativa, si rata de crestere e aproape aceeasi:

