K.R. MANGALAM UNIVERSITY
THE COMPLETE WORLD OF EDUCATION

Course: Data Structure

(Course Code : ENCS205)

UNIT-1: Foundations of Data Structures

School of Engineering & Technology
K.R. Mangalam University

SESSION 5:

Asymptotic Notations (Big-O, Theta, Omega)

Recapitulation (Previous Session)

Quiz

Q1: An array where each element itself is an array is called as '__________'.

Q2: The process of visiting each element of an array is called _____________.

Q3: To find the length of an array in Java, you use the property called ___________.

Q4: In most programming languages, array indices start at _____________.

Recapitulation (Previous Session)

Quiz

Q5: What is the term used to describe an array where elements are stored consecutively in memory?

Q6: What is the main advantage of using arrays?

a) Efficient storage

b) Dynamic resizing

c) Random access

d) Ability to store elements of different data types

Quiz (Answers)

A1: multidimensional array

A2: traversal

A3: length

A4: 0

A5: Contiguous array

A6: c) Random access

Asymptotic Analysis

- The efficiency of an algorithm depends on the amount of time, storage and other resources required to execute the algorithm.

- An algorithm may not have the same performance for different types of inputs.

- The study of change in performance of the algorithm with the change in the order of the input size is defined as asymptotic analysis.

Asymptotic Analysis (Cont..)

- Mathematical notations used to describe the running time of an algorithm when the input tends towards a particular value or a limiting value.

- For example: In bubble sort, when the input array is already sorted, the time taken by the algorithm is linear i.e. the best case.

- But, when the input array is in reverse condition, the algorithm takes the maximum time (quadratic) to sort the elements i.e. the worst case.

Asymptotic Analysis (Cont..)

- When the input array is neither sorted nor in reverse order, then it takes average time. These durations are denoted using asymptotic notations.

- There are mainly three asymptotic notations:

a. Big-O notation
b. Omega notation
c. Theta notation

How do we decide which algorithm is better?

- **Given two algorithms for the same problem, how do we decide which one is better**
- **One approach- implement both the algorithms on same machine.**
- **There are many problems with this approach.**
- **Dependency on programming & machine dependent factors.**
- **Second approach-Asymptotic Analysis**

Asymptotic Analysis of Algorithms

- **We estimate the performance of an algorithm in terms of input size (we don't measure the actual running time);**
- **Determines the growth rate of algorithms;**
- **Asymptotic analysis determines the best case, average case, and worst case scenario;**
- **To compare the efficiency of different algorithms;**
- **Asymptotic Analysis is not perfect,**

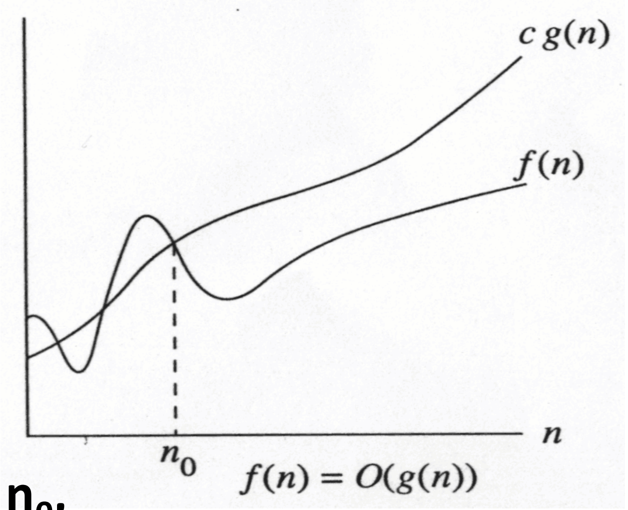
but that's the best way available for analyzing algorithms

Big-O Notation

Big-O is an Asymptotic Notation for the worst case

- It provides us with an ***asymptotic upper bound*** for the growth rate of runtime of an algorithm

- Say f(n) is your algorithm runtime, and g(n) is an arbitrary time complexity you are trying to relate to your algorithm.



- f(n) is O(g(n)), if for some real constants c (c > 0) and n_0, Such that f(n) <= c g(n) for every input size n (n > n_0).

- f(n) grows no faster than g(n) for "large" n
- g(n) defines an upper bound on f(n)

Big-O Notation (Cont..)

Examples:

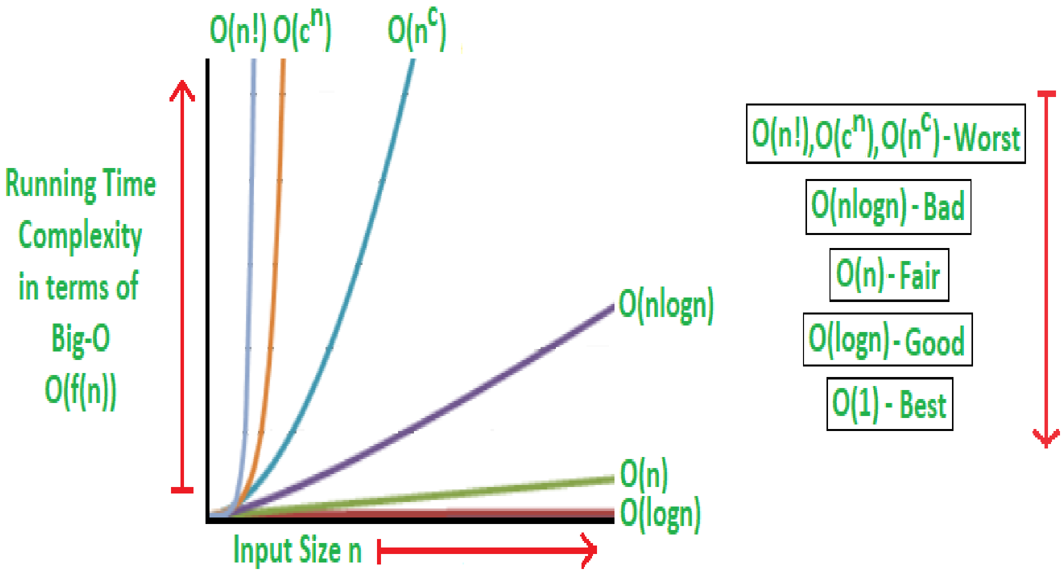
If $f(n)=2n+3$ is your algorithm runtime ,
Can we write $f(n)=O(n)$?

$f(n) = 3*n^2$ and $g(n) = n$;
Is $f(n)=O(n)$?

$2n + 3$ is $O(n)$ and $5n$ is $O(n)$;
place $2n + 3$ and $5n$ in the same category

- Big-Oh allows us to ignore constant factors and lower order (or less dominant) terms

Common Asymptotic notations & comparison



constant	O(1)
logarithmic	O(log n)
linear	O(n)
n log n	O(n log n)
quadratic	$O(n^2)$
cubic	$O(n^3)$
polynomial	$n^{O(1)}$
exponential	$2^{O(n)}$

One should remember the general order of following functions.

$O(1) < O(\log n) < O(n) < O(n \log n) < O(n*n) < O(n*n*n) < O(n^k) < O(2^n)$

Simplifying with Big-O

By definition, Big-O allows us to:

Eliminate low order terms

- $4n + 5 \Rightarrow 4n$
- $0.5\ n \log n\ -\ 2n\ +\ 7 \Rightarrow 0.5\ n \log n$

Eliminate constant coefficients

- $4n \Rightarrow n$
- $0.5\ n \log n \Rightarrow n \log n$
- $\log n^2 = 2 \log n \Rightarrow \log n$
- $\log_3 n = (\log_3 2) \log n \Rightarrow \log n$

Big-O Examples

$n^2 + 100\,n = O(n^2)$

follows from ... $(n^2 + 100\,n) \leq 2\,n^2$ for $n \geq 10$

$n^2 + 100\,n = \Omega(n^2)$

follows from ...$(n^2 + 100\,n) \geq 1\,n^2$ for $n \geq 0$

$n^2 + 100\,n = \theta(n^2)$

by definition

$n \log n = O(n^2)$

$n \log n = \theta(n \log n)$

$n \log n = \Omega(n)$

Little o asymptotic notation

- Big-O is used as a tight upper-bound on the growth of an algorithm's effort.
- "Little-o" (o()) notation is used to describe an upper-bound that cannot be tight.

- **Definition** : Let f(n) and g(n) be functions that map positive integers to positive real numbers.
- We say that f(n) is o(g(n)) if for any real constant $c > 0$, there exists an integer constant $n0 \geq 1$ such that $0 \leq f(n) < c*g(n)$.

- **For big Oh**: true for *at least one* constant c
- **For little o**: true for all constant c

f(n) = o(g(n)) means

$$\lim_{n \to \infty} f(n)/g(n) = 0$$

- Big-O is an inclusive upper bound, while little-o is a strict upper bound.

Little o asymptotic notation

$f(n) = n+2$
Can we write $f(n)=O(n^2)$?

The following are true for Big-O

$x^2 \in O(x^2)$

$x^2 \in O(x^2 + x)$

$x^2 \in O(200 * x^2)$

The following are true for little-o:

$x^2 \in o(x^3)$

$x^2 \in o(x!)$

Is $7n + 8 \in o(n^2)$?

$\lim_{n \to \infty} f(n)/g(n)$

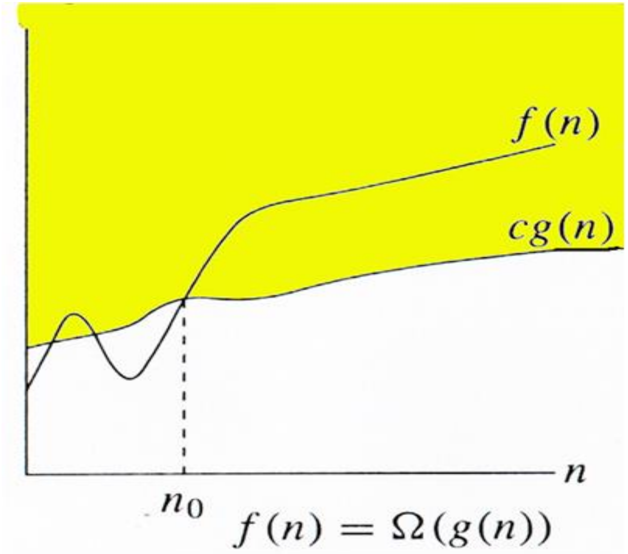
$= \lim_{n \to \infty} (7n + 8)/(n^2)$

$= \lim_{n \to \infty} 7/2n = 0$

Omega Notation (Ω)

- Express the lower bound of an algorithm's running time.
- measure of best case time complexity

for a function *f*(n)
If f(n)= Ω(*g*(n) : there exists c > 0 and n_0 such that c.*g*(n) ≤ *f*(n) for all n > n_0. }



Ex: if f(n)=3n^2+2n+1 then we can take g(n)=n^2 for c=1, s.t
f(n)>=3.g(n) for all n>=1(n_0), c=3
Therefore *f*(n)= Ω(*n*)

Little ω (omega) asymptotic notation

Small-omega, commonly written as ω,

denotes the lower bound (that is not asymptotically tight) on the growth rate of runtime of an algorithm.

f(n)=ω(g(n)), if for all real constants c (c > 0) and n_0 (n_0 > 0), f(n) is > c g(n) for every input size n (n > n_0).

in f(n) = Ω(g(n)), the bound f(n) >= g(n) holds for *some* constant c > 0,

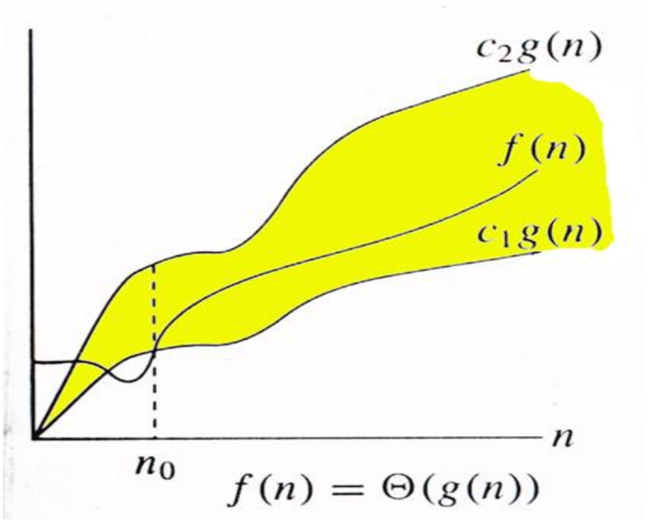
but in f(n) = ω(g(n)), the bound f(n) > c g(n) holds for *all* constants c > 0.

Theta Notation(θ)

- Theta, commonly written as Θ, is an Asymptotic Notation to denote the *asymptotically tight bound* on the growth rate of runtime of an algorithm.
- Express both lower bound and the upper bound of an algorithm's running time.

Θ(g(n)) = {f(n): there exist positive constants c1, c2 and n0 such that 0 <= c1*g(n) <= f(n) <= c2*g(n) for all n >= n0}

f(n) is always between c1*g(n) and c2*g(n) for large values of n (n >= n0)



Theta Notation(θ)

Example: f(n)=3n+2

1) We show that f(n)<=C1.g(n)

   let g(n)=n and C1=4

   by def. of Big-O

   3n+2<=4. n which is true for all n>2(n0)

2) Now we have to show C2.g(n)<=f(n) for satisfying omega notation

   here g(n)=n, let C2=1, then

   1. n<=3n+2 is also true for all n>2(n0)

Therefore by definition of theta notation

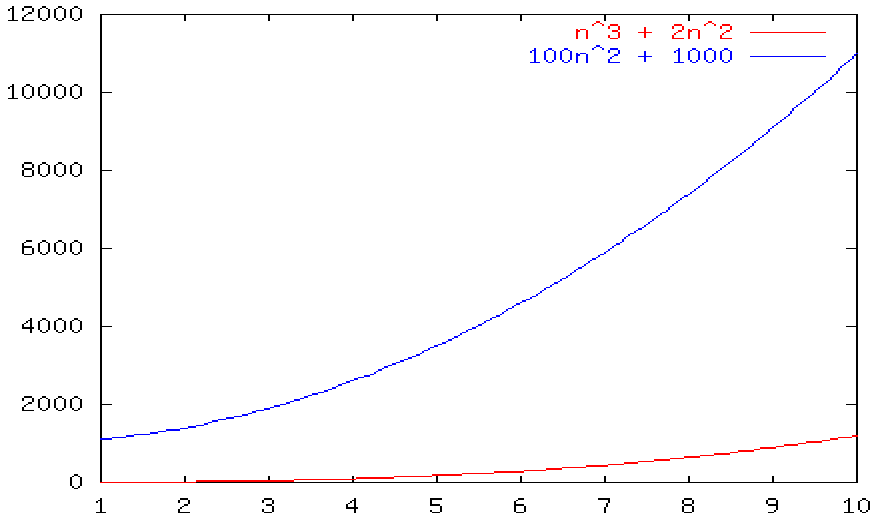
F(n)= Θ(g(n))= Θ(n) for constants C1=4 and C2=1 for all n>2

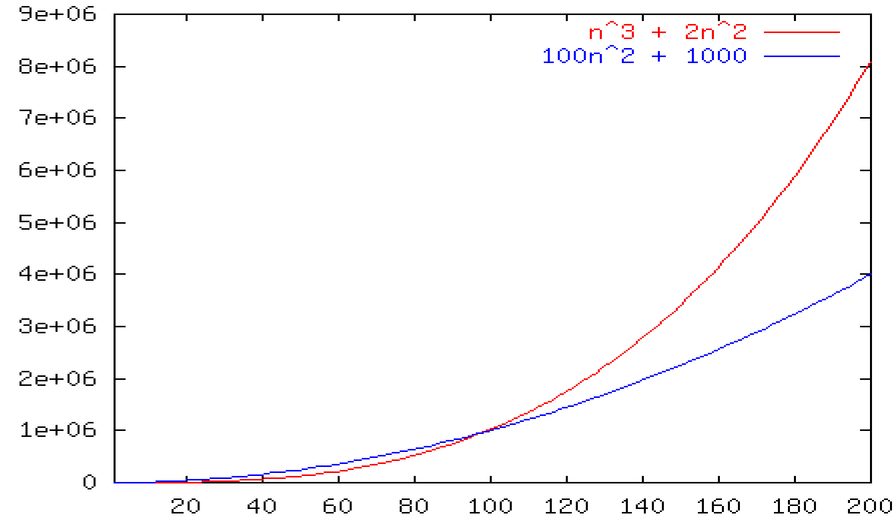
Order Notations

Big-O	$T(n) = \mathbf{O}(\ f(n)\ )$ Exist positive constants c, n_0 such that $T(n) \le cf(n)$ for all $n \ge n_0$	Upper bound
Omega	$T(n) = \Omega(\ f(n)\ )$ Exist positive constants c, n_0 such that $T(n) \ge cf(n)$ for all $n \ge n_0$	Lower bound
Theta	$T(n) = \theta(\ f(n)\ )$ $T(n) = O(f(n))$ AND $T(n) = \Omega(f(n))$	Tight bound
little-o	$T(n) = o(\ f(n)\ )$ $T(n) = O(f(n))$ AND $T(n) \mathrel{!=} \theta(f(n))$	Strict upper bound

Race 1

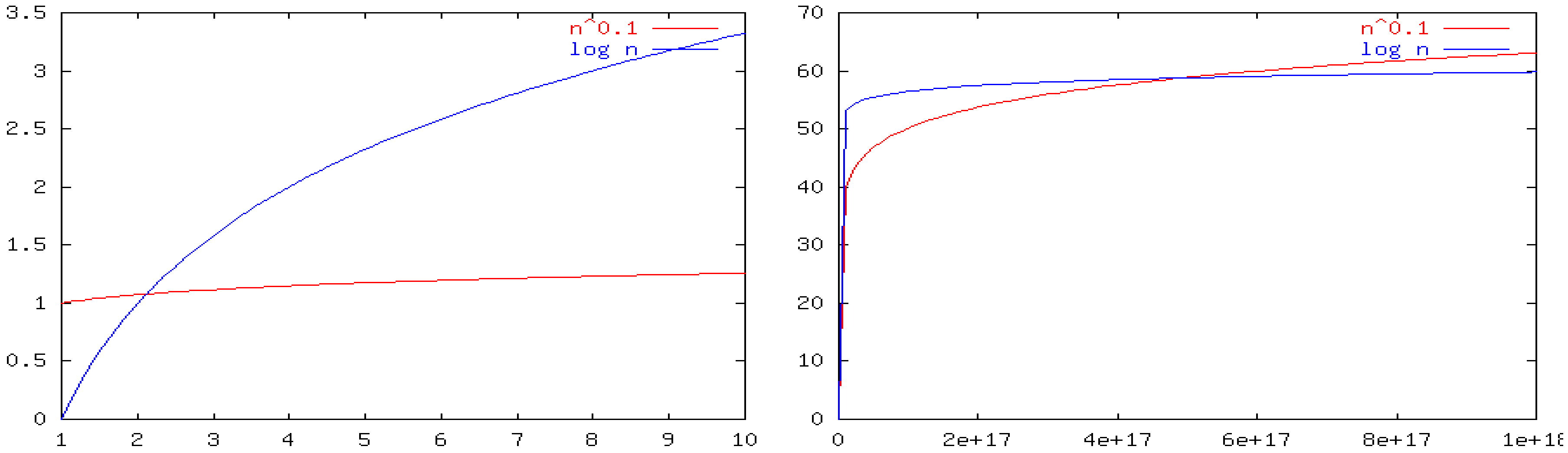
$n^3 + 2n^2$ vs. $100n^2 + 1000$





Race 2

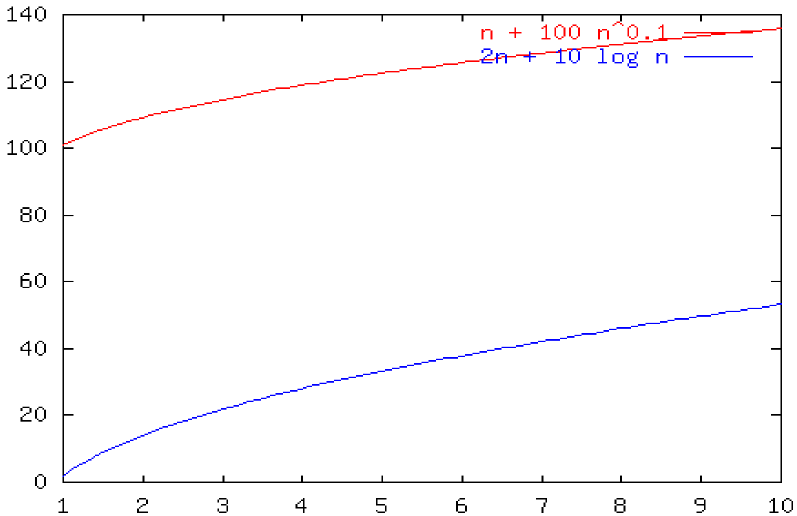
$n^{0.1}$ vs. $\log n$

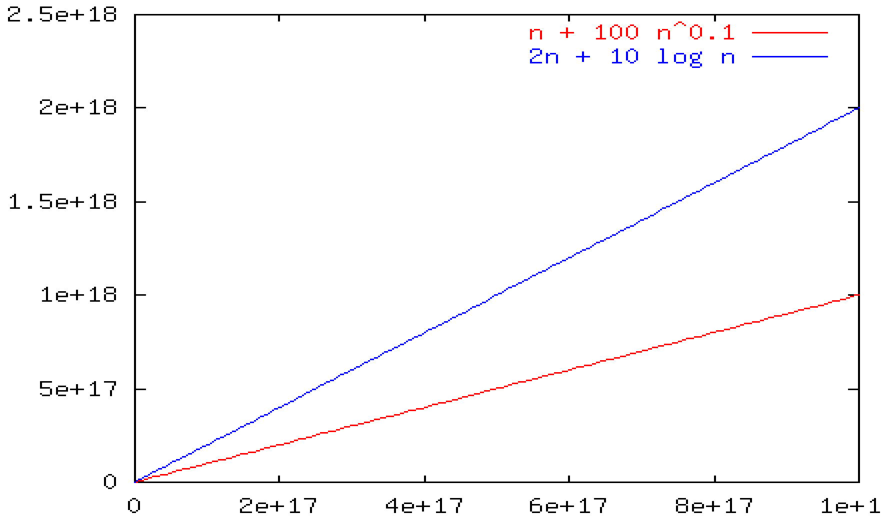


In this one, crossover point is **very late**! So, which algorithm is really better???

Race 3

$n + 100n^{0.1}$ vs. $2n + 10 \log n$

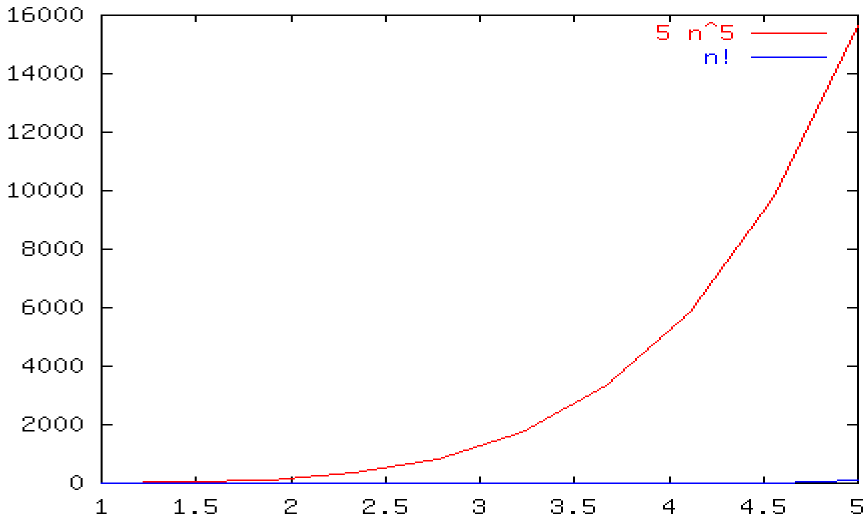


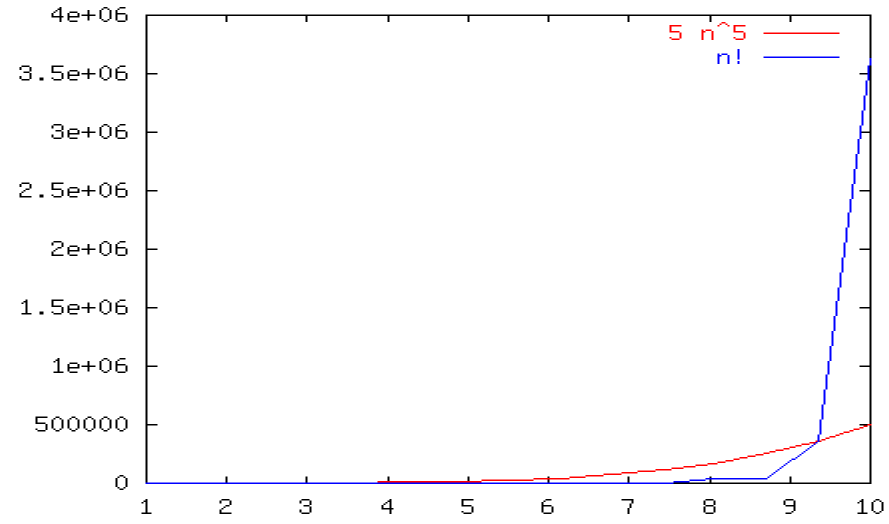


Is the "better" algorithm **asymptotically** better???

Race 4

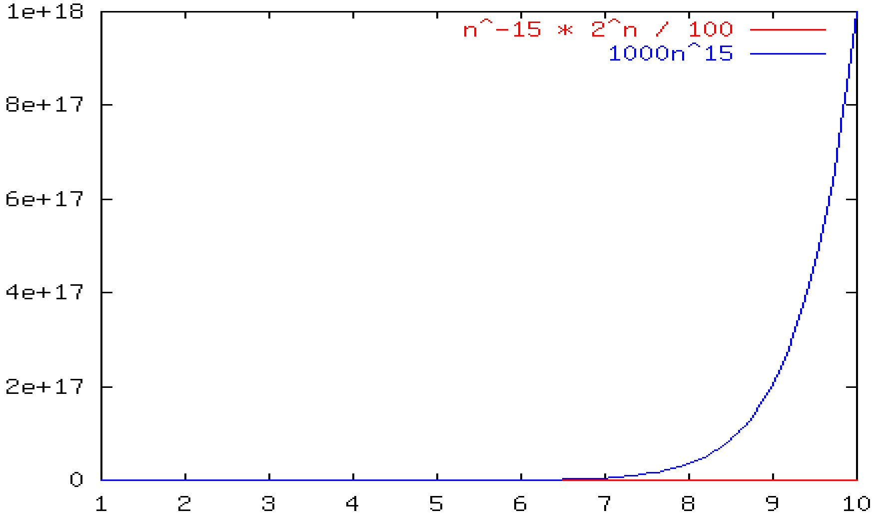
$5n^5$ vs. $n!$

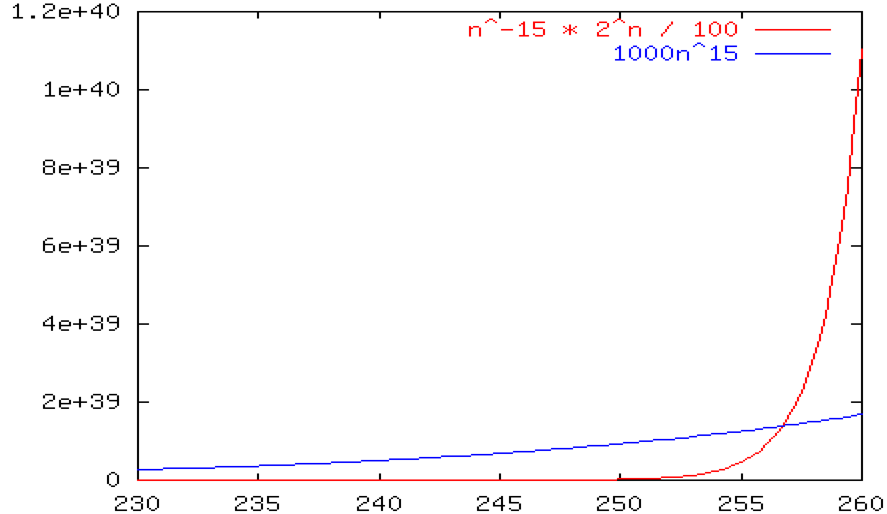




Race 5

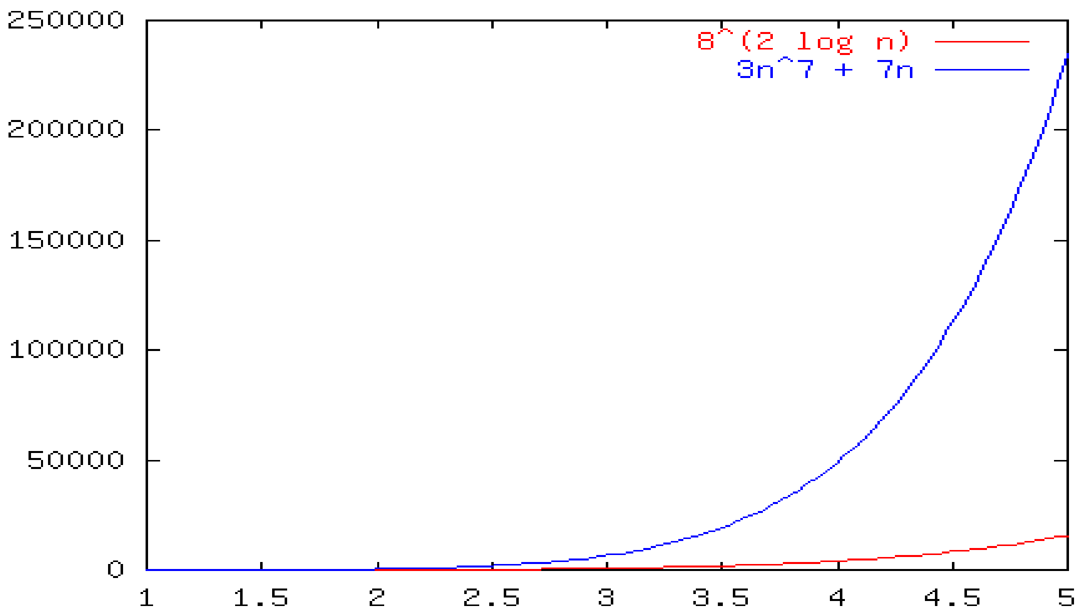
$$n^{-15}2^n/100 \quad \text{vs.} \quad 1000n^{15}$$





Race 6

$8^{2\log(n)}$ vs. $3n^7 + 7n$



Big-O Winners (i.e. losers)

Function A		Function B	Winner
$n^3 + 2n^2$		$\mathbf{100n^2 + 1000}$	$O(n^2)$
$n^{0.1}$		$\mathbf{\log n}$	$O(\log n)$
$\mathbf{n + 100n^{0.1}}$	vs.	$\mathbf{2n + 10 \log n}$	$O(n)$ TIE
$\mathbf{5n^5}$		$n!$	$O(n^5)$
$n^{-15}2^n/100$		$\mathbf{1000n^{15}}$	$O(n^{15})$

Big-O Common Names

constant:	$O(1)$	
logarithmic:	$O(\log n)$	
linear:	$O(n)$	
log-linear:	$O(n \log n)$	
superlinear:	$O(n^{1+c})$	(c is a constant > 0)
quadratic:	$O(n^2)$	
polynomial:	$O(n^k)$	(k is a constant)
exponential:	$O(c^n)$	(c is a constant > 1)

Practice Questions

Q1: From lowest to highest, what is the correct order of the complexities O (n2), O (3n), O (2n), O (n2 lg n), O (1), O (n lg n), O (n3), O (n!), O (lg n), O (n)?

Q2: Suppose we have written a procedure to add m square matrices of size n x n. If adding two square matrices requires O (n^2) running time, what is the complexity of this procedure in terms of m and n?

Practice Questions

Q3: Suppose we have two algorithms to solve the same problem. One runs in time $T1(n) = 400n$, whereas the other runs in time $T2(n) = n^2$. What are the complexities of these two algorithms? For what values of n might we consider using the algorithm with the higher complexity?

Q4: Consider the following three claims

1. $(n + k)^m = \Theta(n^m)$, where k and m are constants
2. $2^{n+1} = O(2^n)$
3. $2^{2n+1} = O(2^n)$

Which of these claims are correct ?

Answers

A1: From lowest to highest, the correct order of these complexities is O (1), O (lg n), O (n), O (n lg n), O (n^2), O (n^2 lg n), O (n^3), O (2n), O (3n), O (n!).

A2: To add m matrices of size n x n, we must perform m - 1 additions, each requiring time O (n2). Therefore, the overall running time of this procedure is:

O(m-1)O(n2) = O(m)O(n2) = O(mn2)

Answers

A3: The complexity of T1 is O (n), and the complexity of T2 is O (n2). However, the algorithm described by T1 involves such a large constant coefficient for n that when n < 400, the algorithm described by T2 would be preferable. This is a good example of why we sometimes consider other factors besides the complexity of an algorithm alone.

Answers

A4:
Explanation: $(n + k)^m$ and $\Theta(n^m)$ are asymptotically same as theta notation can always be written by taking the leading order term in a polynomial expression.
$2^{n + 1}$ and $O(2^n)$ are also asymptotically same as $2^{n + 1}$ can be written as $2 * 2^n$ and constant multiplication/addition doesn't matter in theta notation.
$2^{2n + 1}$ and $O(2^n)$ are not same as constant is in power.

THANK YOU