DESIGN AND ANALYSIS OF ALGORITHMS (DAA)

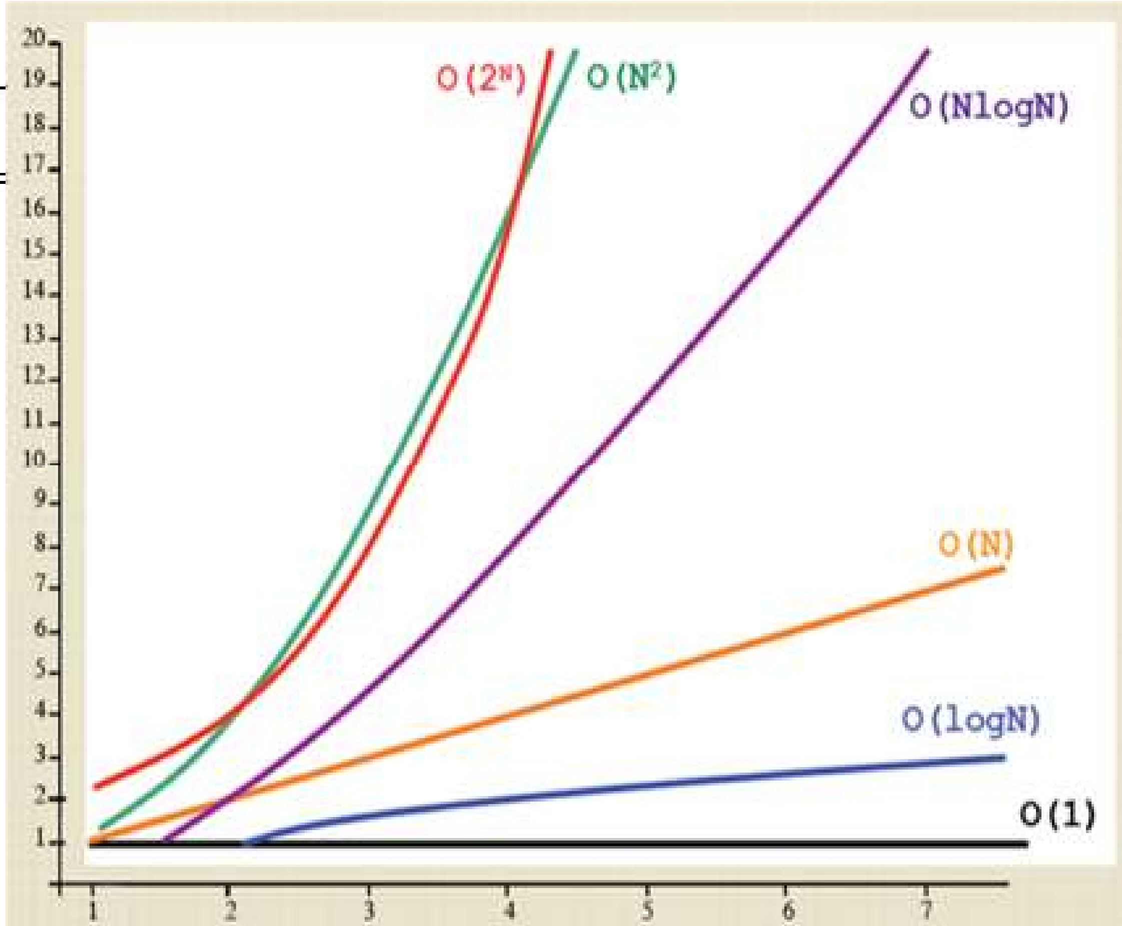
Module1 - Introduction

Radhika Chapaneri

Function of Growth rate

Function	Name
c	Constant
$\log N$	Logarithmic
$\log^2 N$	Log-squared
N	Linear
$N \log N$	$N \log N$
N^2	Quadratic
N^3	Cubic
2^N	Exponential

Functions in order of increasing growth rate



Aysmptotic Performance

- How does algorithm behave as the problem size gets very large?

- When we look at input sizes large enough to make only the order of growth of the running time relevant, we are studying the ***asymptotic*** efficiency of algorithms.

- For example: $an^2 + bn + c$: $O(n^2)$
  - Ignore actual and abstract statement costs
  - Highest-order term is what counts : Remember, we are doing asymptotic analysis. As the input size grows larger it is the high order term that dominates

Asymptotic Notations

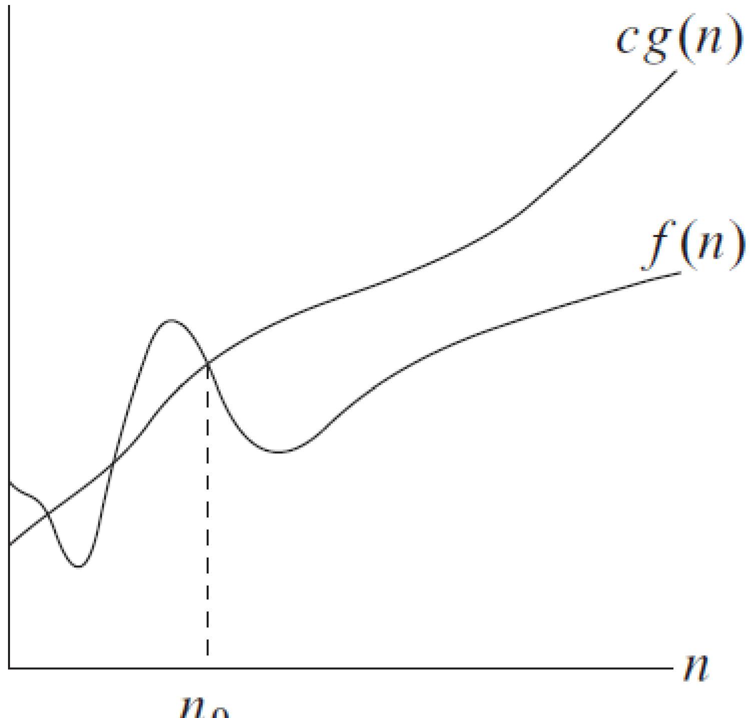
- **Asymptotic Efficiency**: When we look at input sizes large enough to make only the order of growth of the running time relevant, we are studying the ***asymptotic*** efficiency of algorithms.

- **Asymptotic Notations**: The notations we use to describe the asymptotic running time of an algorithm

- Three commonly used asymptotic notations are
  - Θ - tight bound
  - O – upper bound
  - Ω- lower bound

Big – Oh notation

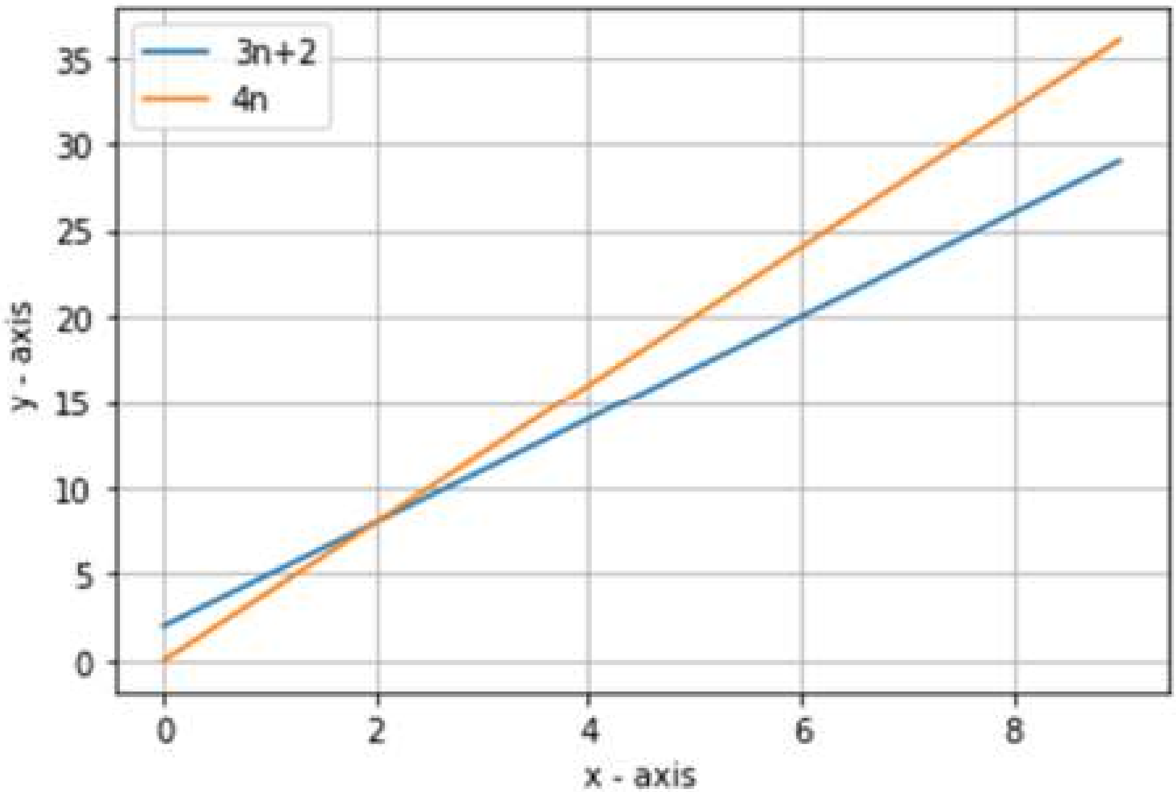
$f(n) = O(g(n))$: there exist positive constants c and n_0 such that $0 \le f(n) \le cg(n)$ for all $n \ge n_0$



$f(n) = O(g(n))$

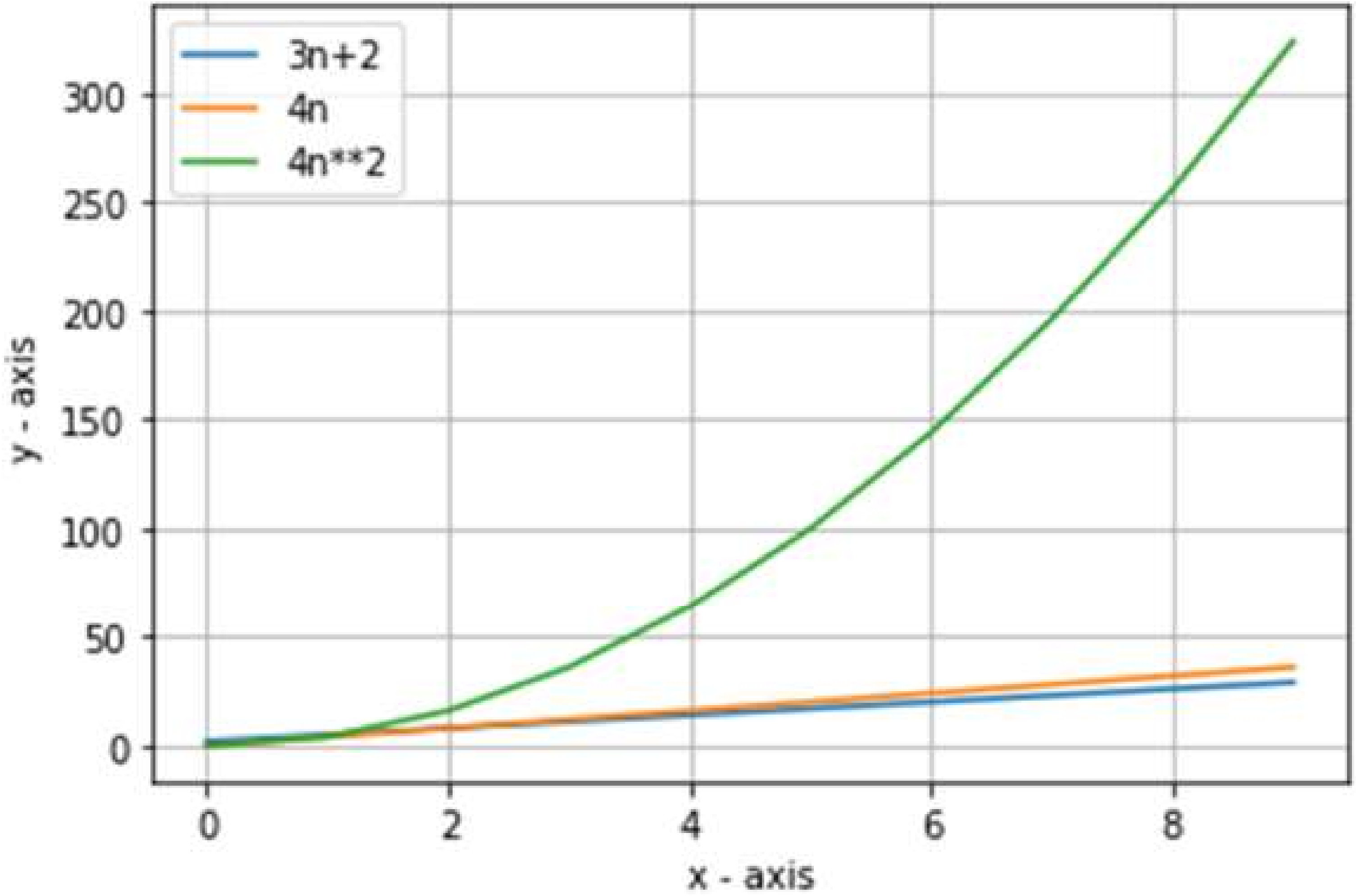
Big-Oh Notation

- $f(n) = 3n+2$ $g(n) = n$
- Is $f(n) = O(n)$?
- $f(n) \leq c\, g(n)$
- $3n+2 \leq c\, n$



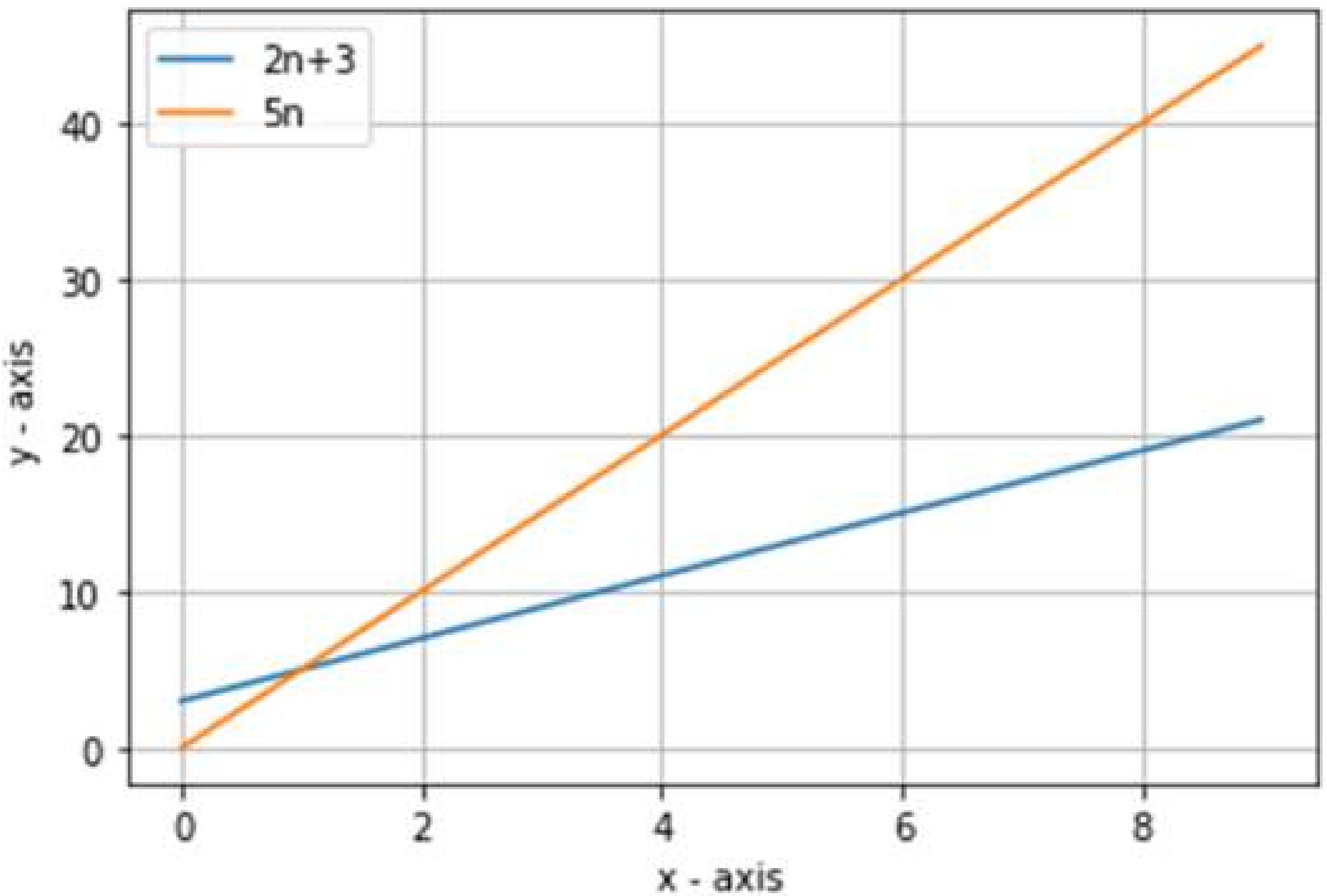
3n+2
4n
35
30
25
20
15
10
5
0
y - axis
0
2
4
6
8
x - axis

- $f(n) = 3n+2$ $g(n) = n$
- Is $f(n) = O(n)$?
- $f(n) \leq c\, g(n)$
- $3n+2 \leq c\, n$
- Suppose $c = 4$ and $n0=2$
- $3n+2 \leq 4n$



3n+2
4n
4n**2
300
250
200
150
100
50
0
0
2
4
6
8
y - axis
x - axis

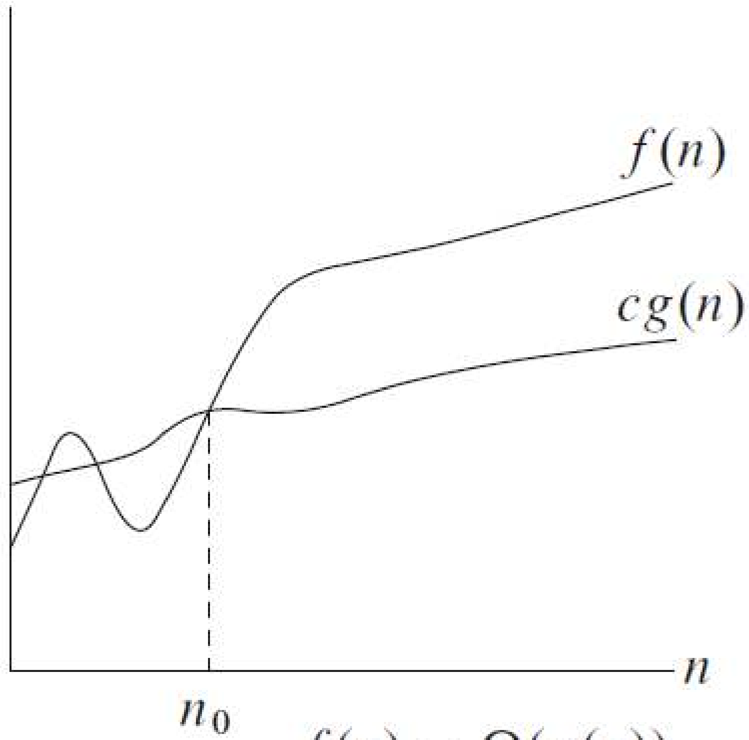
- $f(n) = 2n+3$ $g(n) = n$
- Is $f(n) = O(n)$?
- $f(n) \leq c\, g(n)$


2n+3
5n
40
30
20
10
0
y - axis
0
2
4
6
8
x - axis

Omega notation

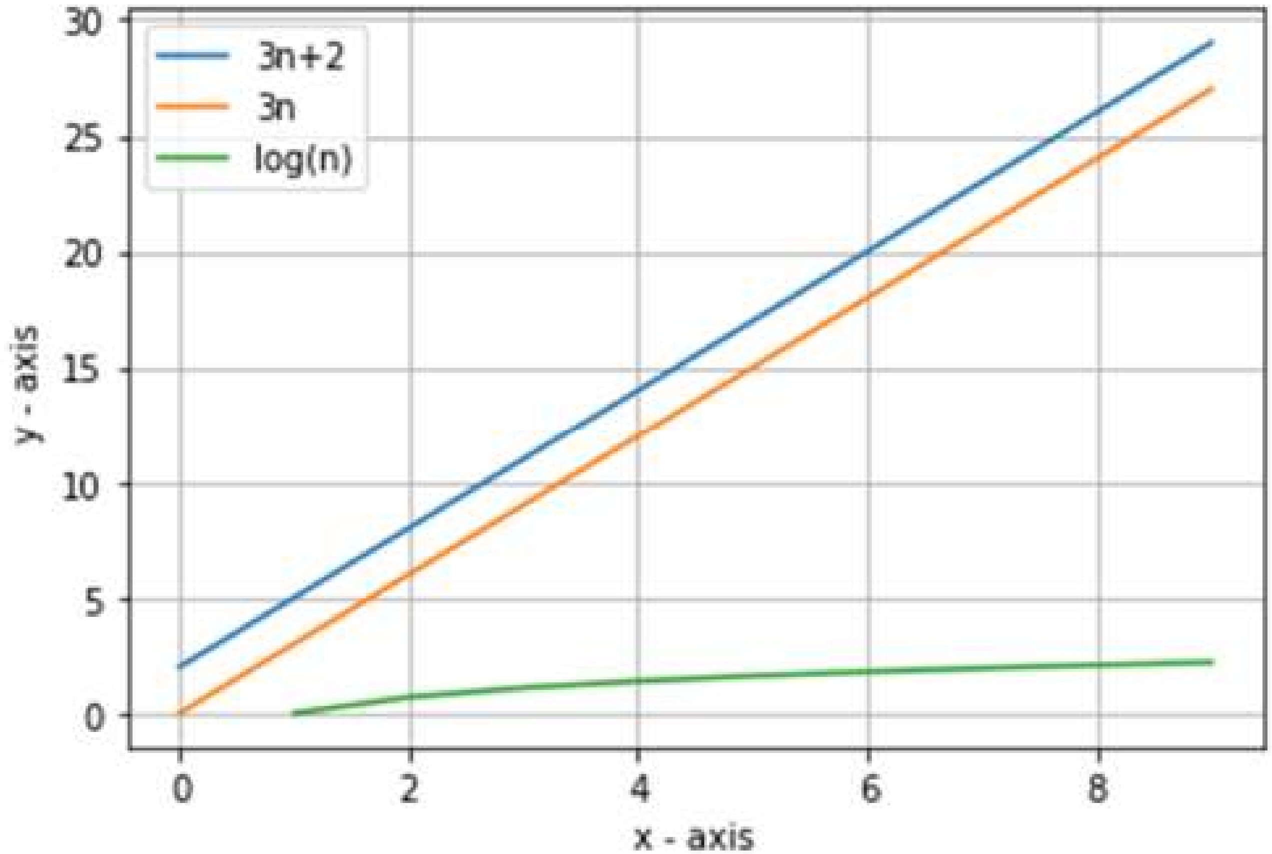
$f(n)=\Omega(g(n))$: there exist positive constants c and n_0 such that

$$0 \le cg(n) \le f(n) \text{ for all } n \ge n_0$$



$f(n) = \Omega(g(n))$

- $f(n) = 3n+2$ $g(n) = n$
- Is $f(n) = \Omega(n)$
- $f(n) \geq c\, g(n)$

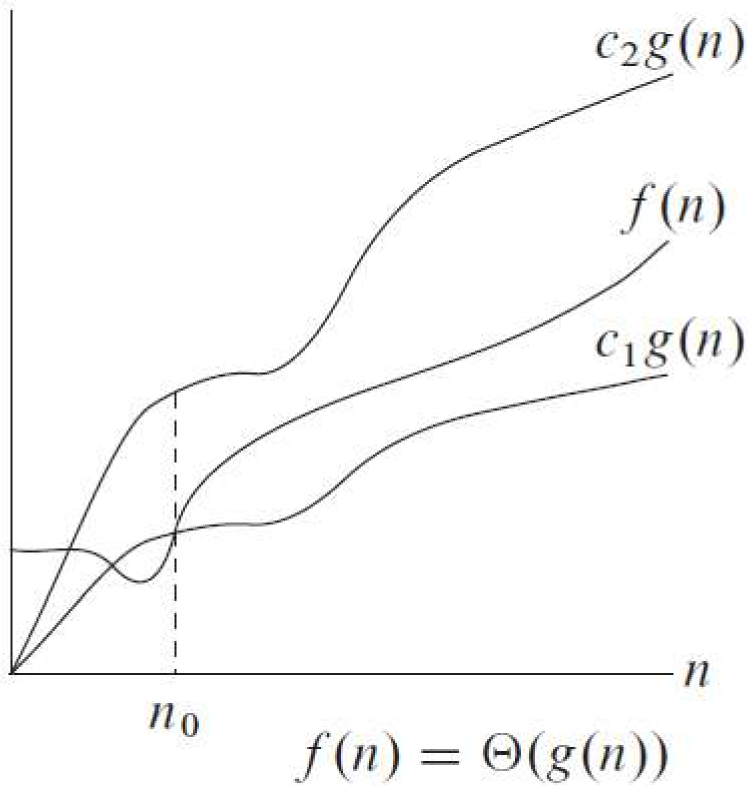


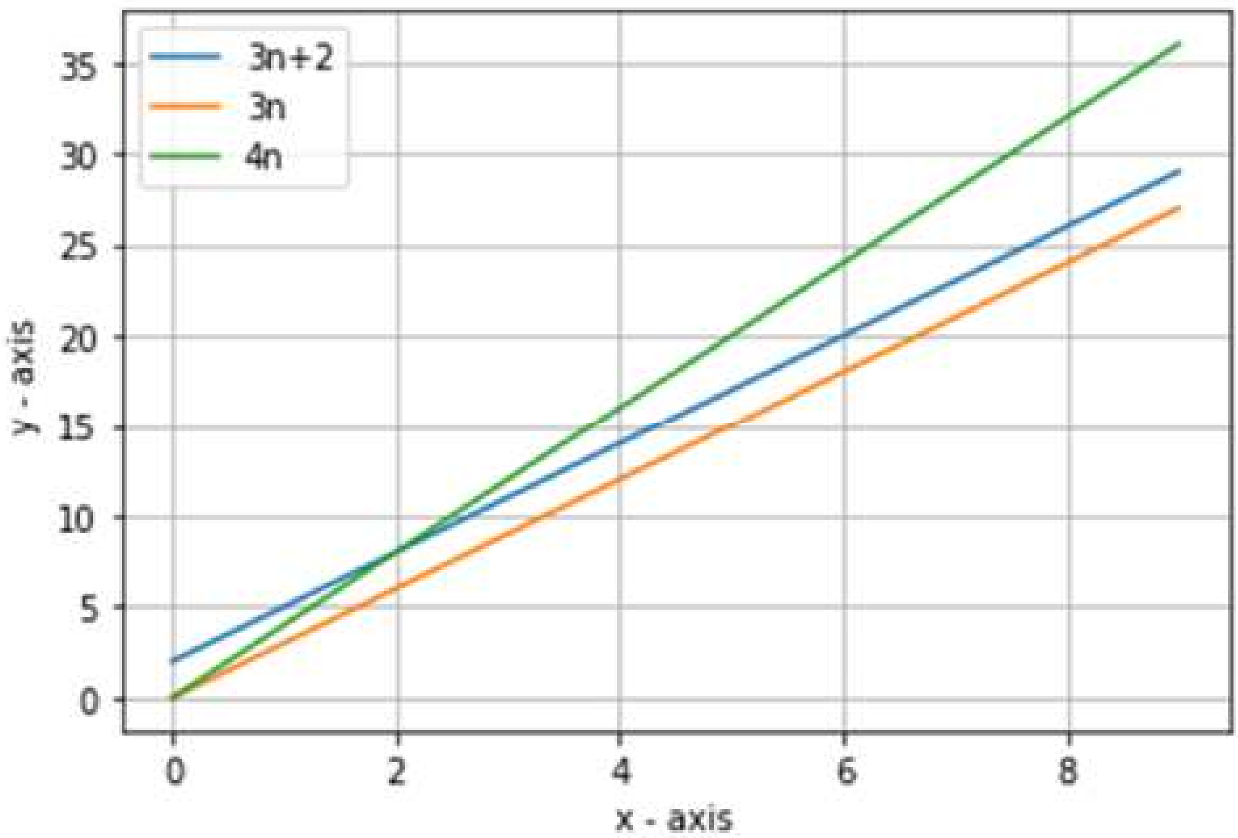
30
25
20
15
10
5
0
0
2
4
6
8
3n+2
3n
log(n)
y - axis
x - axis

Theta notation

$f(n) = \Theta(g(n))$: there exist positive constants c_1, c_2, and n_0 such that

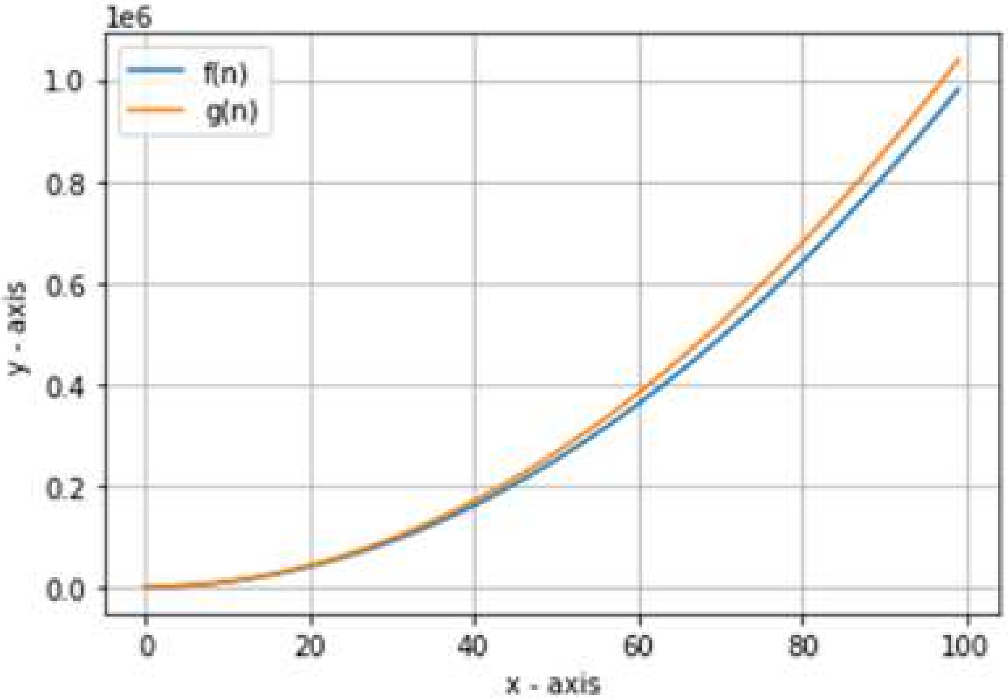
$$0 \le c_1 g(n) \le f(n) \le c_2 g(n) \text{ for all } n \ge n_0$$





3n+2
3n
4n
35
30
25
20
15
10
5
0
0
2
4
6
8
y - axis
x - axis

- Import matplotlib.pyplot as plt
- Import numpy as np
- x = np.arange(0, 10)
- y1 = 2*(x**2) + 3*x+1
- y2 = x
- plt.plot(x, y1, label = 'f(n)')
- plt.plot(x, y2, label = 'g(n)')
- plt.grid()
- plt.show()



1e6
f(n)
g(n)
1.0
0.8
0.6
0.4
0.2
0.0
y - axis
0
20
40
60
80
100
x - axis

Recurrence

- ***Recurrence relations are useful for expressing*** the running times of recursive algorithms

- When an algorithm contains a recursive call to itself, we can often describe its running time by a ***recurrence equation*** or ***recurrence***, which describes the overall running time on a problem of size n in terms of the running time on smaller inputs.

- We can then use mathematical tools to solve the recurrence and provide bounds on the performance of the algorithm.

Methods for solving Recurrence Relations

- **Substitution method:**
- Make a guess
- Verify the guess using induction
- **Recursion trees:**
- Visualize how the recurrence unfolds
- May lead to a guess to be verified using substitution
- If done carefully, may lead to an exact solution
- **Master theorem:**
- “Cook-book” solution to a common class of recurrence relations

The Master Method

- “Cookbook” approach for solving recurrences of the form

$$T(n) = aT(n/b) + f(n)$$

  - $a \geq 1$, $b > 1$ are constants.
  - $f(n)$ is asymptotically positive.
- The recurrence describes the running time of an algorithm that divides a problem of size n into a subproblems, each of size n/b, where a and b are positive constants. The a subproblems are solved recursively, each in time T (n/b). The function f (n) encompasses the cost of dividing the problem and combining the results of the subproblems.
- To solve such equation we require memorization of three cases.

The Master Theorem

Let $a \geq 1$ and $b > 1$ be constants, let $f(n)$ be a function, and Let $T(n)$ be defined on nonnegative integers by the recurrence

$$T(n) = aT(n/b) + f(n)$$

$T(n)$ can be bounded asymptotically in three cases:

1. If $\boldsymbol{f(n) = O(n^{\log_b a - \varepsilon})}$ for some constant $\varepsilon > 0$, then $\boldsymbol{T(n) = \Theta(n^{\log_b a})}$.

2. If $\boldsymbol{f(n) = \Theta(n^{\log_b a})}$, then $\boldsymbol{T(n) = \Theta(n^{\log_b a} \lg n)}$.

3. If $\boldsymbol{f(n) = \Omega(n^{\log_b a + \varepsilon})}$ for some constant $\varepsilon > 0$, and if, for some constant $c < 1$ and all sufficiently large n, we have $a \cdot f(n/b) \leq c\, f(n)$, then $\boldsymbol{T(n) = \Theta(f(n))}$.

The Master Method

- In each of the three cases, we compare the function f(n) with the function $\boldsymbol{n^{\log_b a}}$. Intuitively, the larger of the two functions determines the solution to the recurrence.

- If, as in case 1, the function $\boldsymbol{n^{\log_b a}}$ is the larger, then the solution is $\boldsymbol{T(n) = \Theta(n^{\log_b a})}$.

- If, as in case 3, the function f (n) is the larger, then the solution is $\boldsymbol{T(n) = \Theta(f(n))}$.

- If, as in case 2, the two functions are the same size, we multiply by a logarithmic factor, and the solution is $\boldsymbol{T(n) = \Theta(n^{\log_b a} \lg n)}$.

Case 1:

Compare $f(n)$ with $n^{\log_b a}$:

$f(n) = O(n^{\log_b a - \varepsilon})$ for some constant $\varepsilon > 0$.

$f(n)$ grows polynomially slower than $n^{\log_b a}$ (by an n^{ε} factor).

Solution: $T(n) = \Theta(n^{\log_b a})$.

Example:

- $T(n) = 9T(n/3) + n$
  - $a=9$, $b=3$, $f(n) = n$
  - $n^{\log_b a} = n^{\log_3 9} = \Theta(n^2)$
  - Since $f(n) = O(n^{\log_3 9 - \varepsilon})$, where $\varepsilon=1$, case 1 applies:
  - Thus the solution is $T(n) = \Theta(n^2)$

Examples

Ex. $T(n) = 4T(n/2) + n^2$

$a = 4,\ b = 2 \Rightarrow n^{\log_b a} = n^2;\ f(n) = n^2.$

CASE 2: $f(n) = \Theta(n^2 \lg^0 n)$, that is, $k = 0$.

$\therefore\ T(n) = \Theta(n^2 \lg n).$

Examples

Ex. $T(n) = 4T(n/2) + n^3$

$a = 4,\ b = 2 \Rightarrow n^{\log_b a} = n^2;\ f(n) = n^3.$

CASE 3: $f(n) = \Omega(n^{2+\varepsilon})$ for $\varepsilon = 1$

and $4(cn/2)^3 \le cn^3$ (reg. cond.) for $c = 1/2$.

$\therefore\ T(n) = \Theta(n^3)$.

Ex. $T(n) = 4T(n/2) + n^2/\lg n$

$a = 4,\ b = 2 \Rightarrow n^{\log_b a} = n^2;\ f(n) = n^2/\lg n.$

Master method does not apply. In particular, for every constant $\varepsilon > 0$, we have $n^{\varepsilon} = \omega(\lg n)$.

For master’s theorem problem – refer notebook.