Analysis of Algorithm Efficiency

What is Efficiency?

- Running Time (time efficiency/ complexity)
  - Most of our focus
- Memory Space (space efficiency/ complexity)
  - No longer a huge concern

Analysis Framework

- Measure the input size
- Units for measuring Running Time
- Orders of Growth
- Best/Worst/Average Case Efficiencies

1. Input Size

- Larger the input*(n),* the longer the algorithm will run
  - Sorting large arrays
  - Multiplying large matrices

- The input size (*n*) is the number of elements in the list or total number of elements in all the matrices

- The input size(*n*) can sometimes be more than one parameter

2. Units for Measuring Running Time

- Focus is on an algorithms efficiency
- Determine time 3 different ways
  - Time in milliseconds
  - Count the number of times each operation is executed
  - ***Count the number of times the "Basic Operation" is executed***

Basic Operation Evaluation

- Basic Operation
  - Operation contributing the most to the total running time.
  - Calculate how many times this operation is executed
  - Usually the most time consuming operation in the algorithms innermost loop

- Types of Operations (in order)
  1. Division
  2. Multiplication
  3. Addition
  4. Subtraction
  5. Comparison

3. Order of Growth

- Order of Growth is used to evaluate algorithms overall efficiency
  - Which algorithm is better and why

- Easier to see and determine on large values of n (n = input)

- Algorithms that require an exponential number of operations are practical for only solving problems of small sizes

Orders of Growth

n	**lgn**	**n**	**n×lgn**	**n^2**	**n^3**	**2^n**	**n!**
10	3.3	10	3.3×10	10^2	10^3	10^3	$3.6×10^6$
10^2	6.6	10^2	$6.6×10^2$	10^4	10^6	$1.3×10^{30}$	$9.3×10^{157}$
10^3	10	10^3	$10×10^3$	10^6	10^9		
10^4	13	10^4	$13×10^4$	10^8	10^{12}		
10^5	17	10^5	$17×10^5$	10^{10}	10^{15}		
10^6	20	10^6	$20×10^6$	10^{12}	10^{18}		

Worst Case, Best Case, Average Case

- Worst Case – Efficiency for the worst case input of n
  - Runs the longest for all possible inputs of n
  - Backwards sorted array
  - Search value doesn’t exist or last index
- Best Case – Efficiency for the best case input of n
  - Runs the quickest for all possible inputs of n
  - Array is already sorted
  - Value is the first index in a search
- Average Case – Efficiency for a “typical” or “random” input of n
  - Make assumptions of possible inputs of n
  - Probabilities have to be the same
  - ***<u>NOT the best and worst case average together</u>***
- Amortized Efficiency – One single operation is expensive but the average is still always better

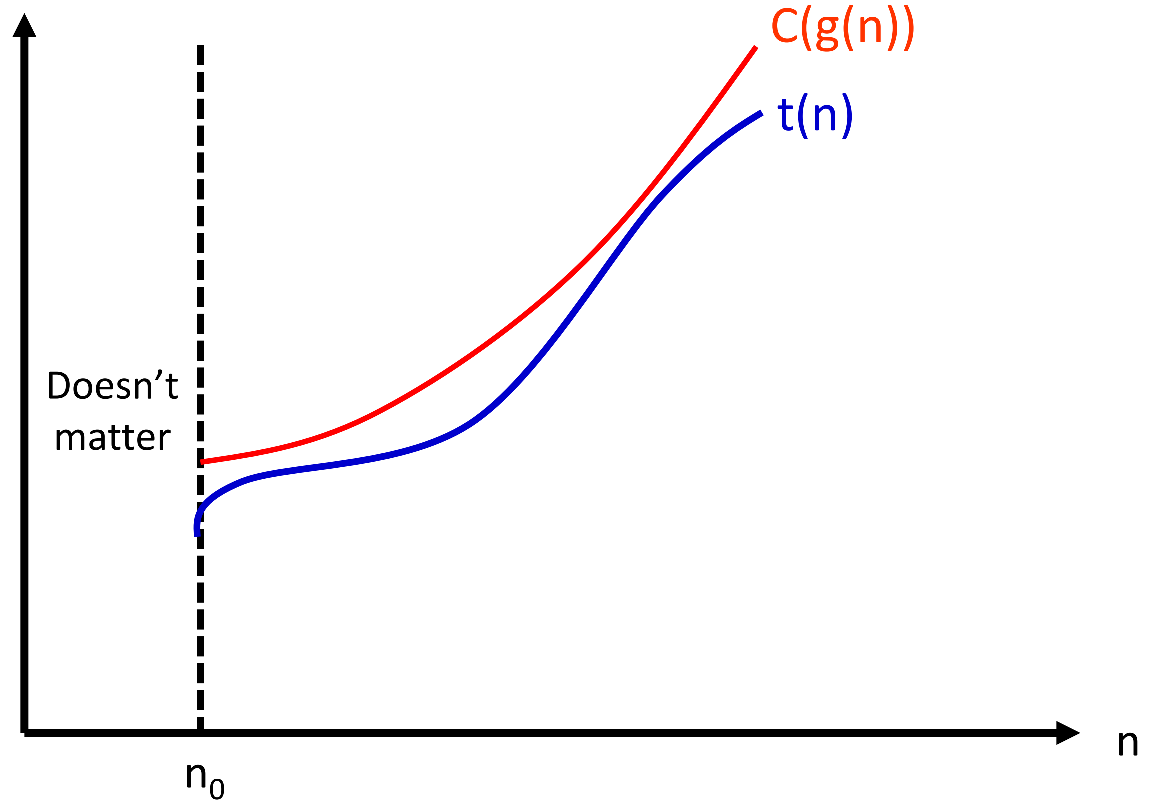
Summary of Analysis Framework

- Both time and space are measured from an algo's input size
- Time efficiency is counting basic operations
- Space is the extra memory consumed by the algorithm
- Efficiencies differ, so we use Best/Average/Worst case to distinguish
- Primary interest is in the order of growth of the running time (and extra memory consumed) as input goes from 1 to infinity.

Asymptotic Notations & Basic Efficiency Classes

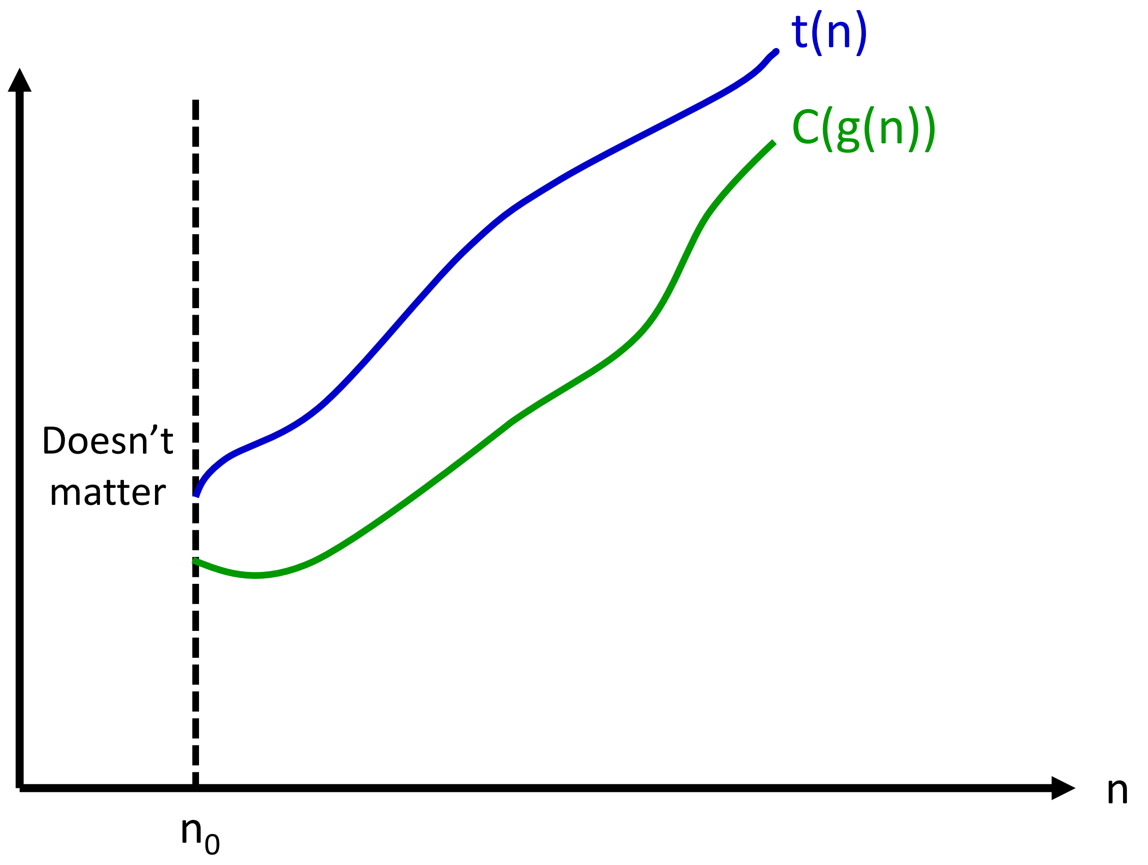
- Three types of Notations
  - Big O - O(g(n)): Set of functions that grow no faster than g(n)
  - Big omega - Ω(g(n)): Set of functions that grow at least as fast as g(n)
  - Big Theta - Θ(g(n)): Set of functions that grow at the same rate as g(n)
  - t(n) = algorithms running time (non-negative natural numbers)
  - C(n) = basic operation
  - g(n) = simple function to compare the count with (non-negative natural numbers)

O(big oh)-Notation



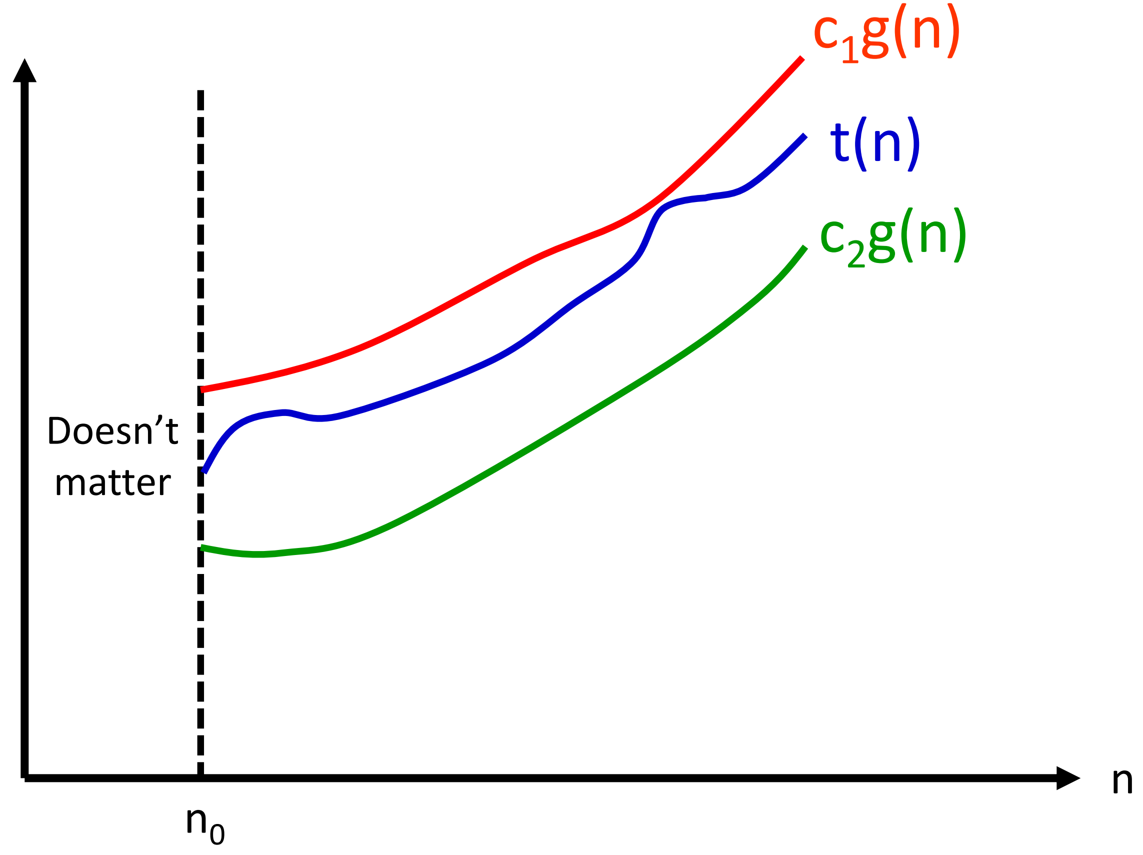
$t(n) \in O(g(n))$

Ω(big omega)-Notation



$t(n) \in \Omega(g(n))$

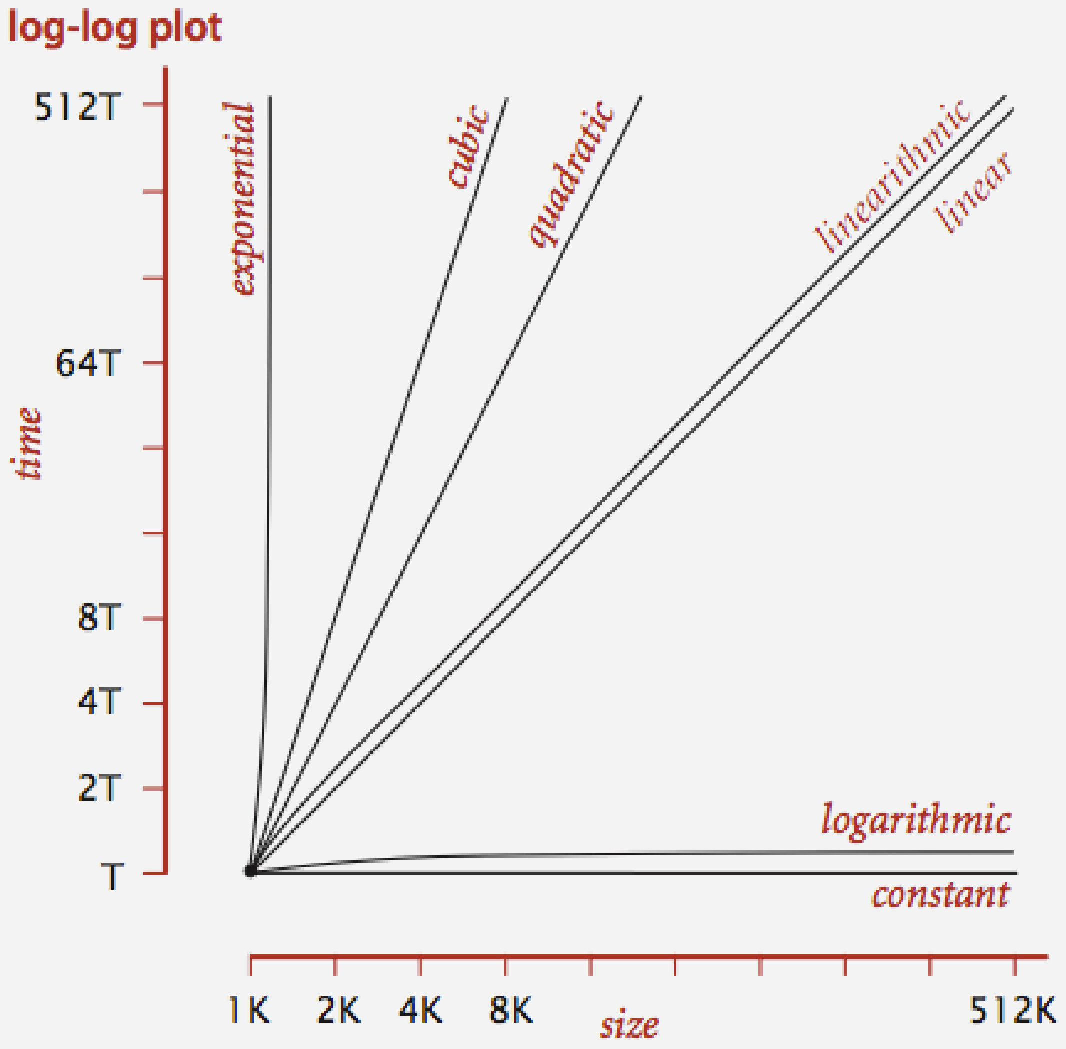
Θ(big theta)-Notation



$t(n) \in \Theta(g(n))$

Basic asymptotic efficiency of code

Class	Name	Sample algorithm type
O(1)	Constant	Algorithm ignores input (i.e., can't even scan input)
O(lgn)	Logarithmic	Cuts problem size by constant fraction on each iteration
O(n)	Linear	Algorithm scans its input (at least)
O(nlgn)	"n-log-n"	Some divide and conquer
O(n^2)	Quadratic	Loop inside loop = "nested loop"
O(n^3)	Cubic	Loop inside nested loop
O(2^n)	Exponential	Algorithm generates all subsets of n-element set of binary values
O(n!)	Factorial	Algorithm generates all permutations of n-element set



Typical orders of growth

Analysis of Nonrecursive Algorithms

```
ALGORITHM MaxElement(A[0..n-1])
//Determines largest element
maxval <- A[0]
for i <- 1 to n-1 do
    if A[i] > maxval
        maxval <- A[i]
return maxval
```

Input size: n

Basic operation: > or <-

C(n) = $\sum_{i=1}^{n-1} 1 = n - 1 - 1 + 1$

$= n - 1$ **є Θ(n)**

Analysis of Nonrecursive (contd.)

```
ALGORITHM UniqueElements(A[0..n-1])
//Determines whether all elements are //distinct
for i <- 0 to n-2 do
   for j <- i+1 to n-1 do
      if A[i] = A[j]
         return false
return true
```

- **Input size: n**
- **Basic operation: A[i] = A[j]**
- **Does C(n) depend on type of input?**

UniqueElements (contd.)

```
for i <- 0 to n-2 do
     for j <- i+1 to n-1 do
          if A[i] = A[j]
               return false
return true
```

C$_{\text{worst}}$(n) = $\sum_{i=0}^{n-2} \sum_{j=i+1}^{n-1} 1 = \frac{(n-1)n}{2} \approx \frac{1}{2} n^2 \in \Theta(n^2)$

Important Summation Formulas

- $\sum_{i=l}^{u} 1 = u - l + 1$
- $\sum_{i=1}^{n} i = \frac{n(n+1)}{2} \approx \frac{1}{2}n^2$
- $\sum_{i=1}^{n} i^2 = \frac{n(n+1)(2n+1)}{6} \approx \frac{1}{3}n^3$

- $\sum_{i=l}^{u} \boldsymbol{c a_i} = \boldsymbol{c} \sum_{i=l}^{u} \boldsymbol{a_i}$
- $\sum_{i=l}^{u} (\boldsymbol{a_i} \pm \boldsymbol{b_i}) = \sum_{i=l}^{u} \boldsymbol{a_i} \pm \sum_{i=l}^{u} \boldsymbol{b_i}$

Why?

- Most web-app building doesn’t require algorithm calculations
- Interviews require it for overall problem solving and understanding
- You should ***always*** be aware of time and space
    - Make informed choices between which to prioritize

It’s Fun!

- Algorithm analysis is fun and exciting!
- It allows us to really dive deeper into the code we write
- Allows us to “Defend our position” and not “Get Defensive”
- By understanding the overview – you will stand out amongst others