Data Structures & Algorithms

Adil M. Khan
Professor of Computer Science
Innopolis University

Fundamental Techniques In Handling People – Dale Carnegie

1. Don’t Criticize, Condemn or Complain

Recap

- What is an “Algorithm”?
- What are ”data structures”?
- Why is it important to study them?

Today’s Objectives

- What is “Algorithm Analysis”?
- Why should we analyze algorithms?
- Understand mathematical machinery needed to analyze algorithms
- Learn what it means for one function to grow faster than another function

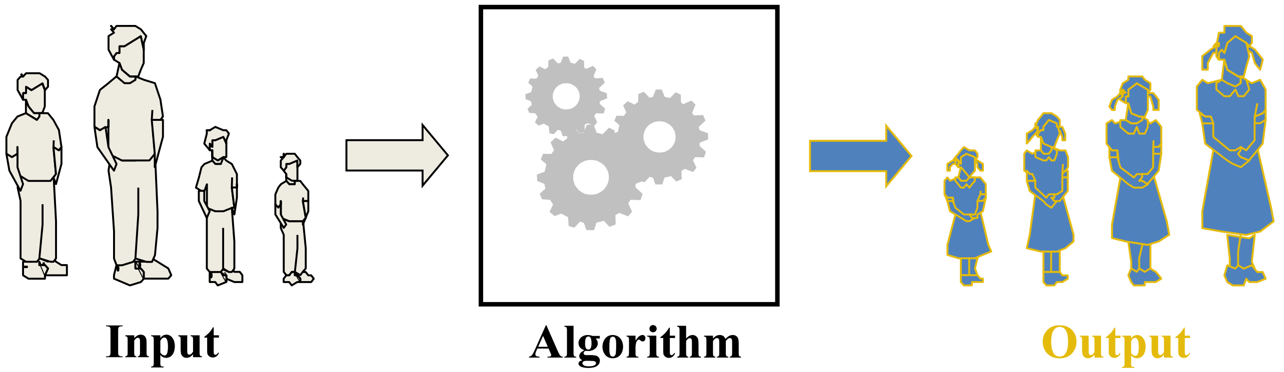
What is “Algorithm Analysis”?

Algorithm Analysis

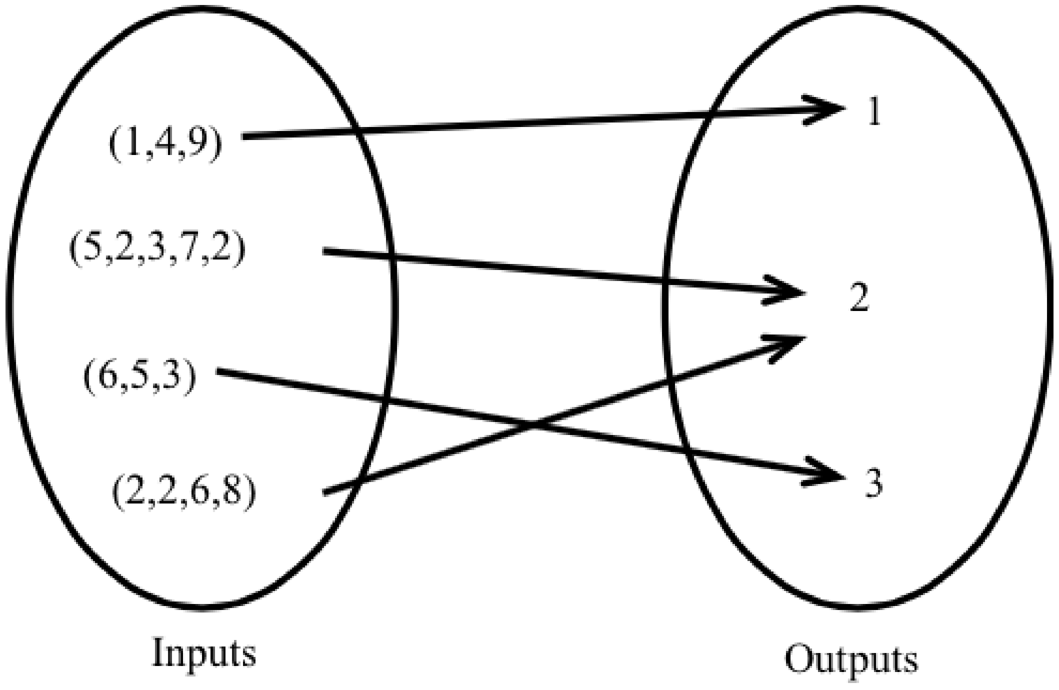
- Analyzing how resource requirements of an algorithm will scale when increasing the input size

Why analyze algorithms?

Algorithm



A more specific example: Find Minimum!



Think of a few more examples as an exercise!

Algorithm

- Another way
  - A tool to solve a well-defined computational problem
  - The statement of the problem defines the desired input/output relationship

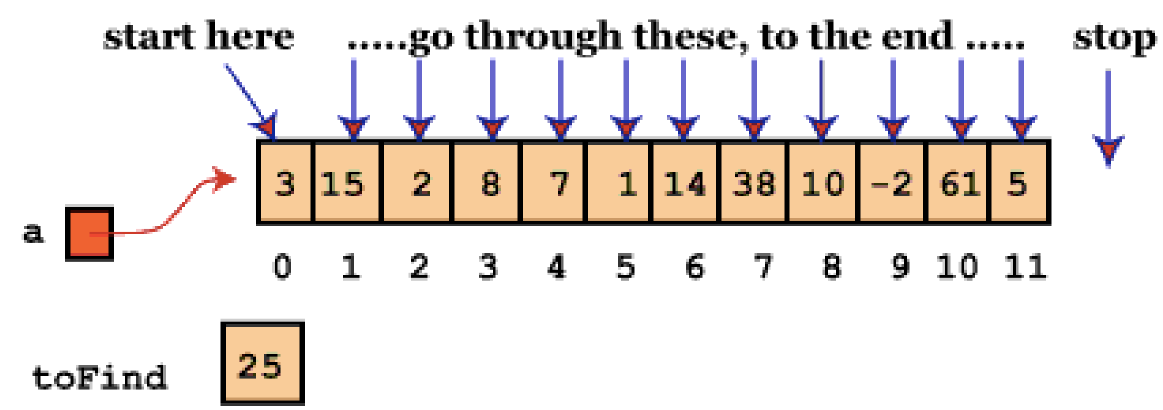
"Sorting a sequence of n elements in a non-decreasing order"

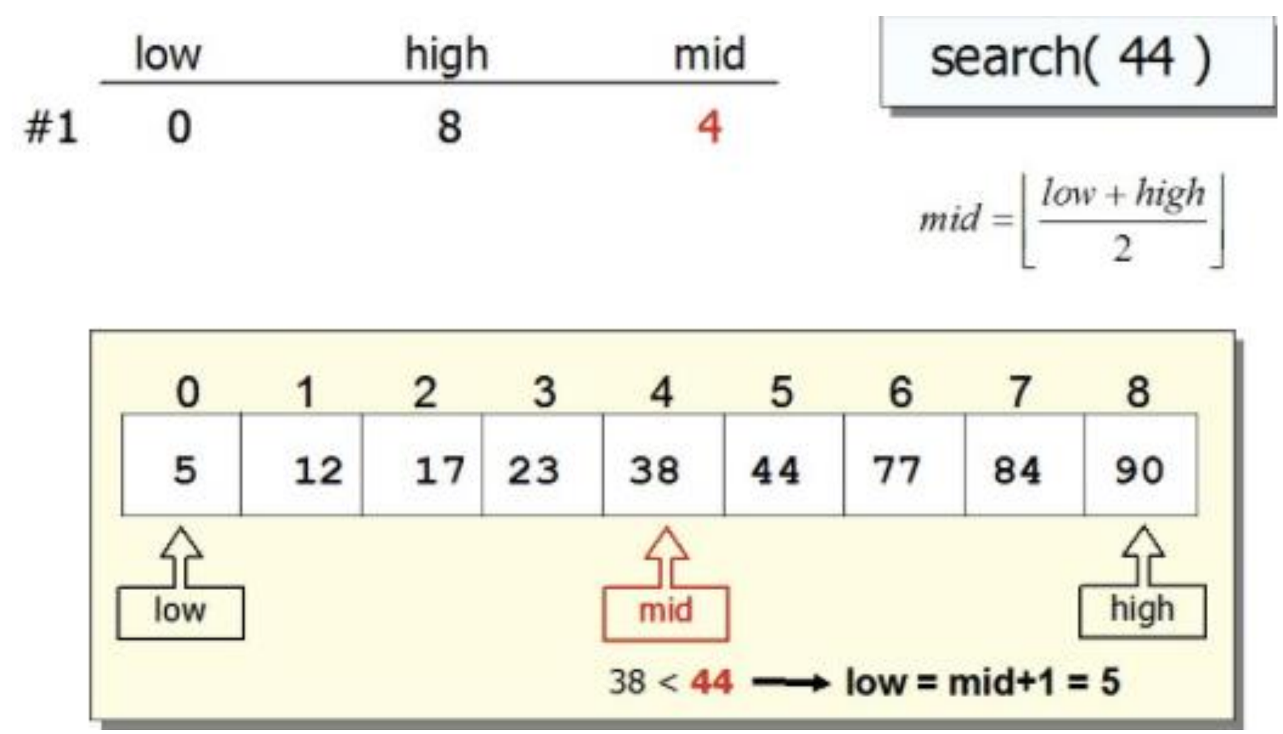
Two Characteristics of Algorithmic Problems

- They have practical applications
- They have many candidate solutions

Algorithm Analysis

- Allows us to:
  - Compare the merits of two alternative approaches to a problem we need to solve
  - Determine whether a proposed solution will meet required resource constraints before we invest money and time coding





Performed before coding!

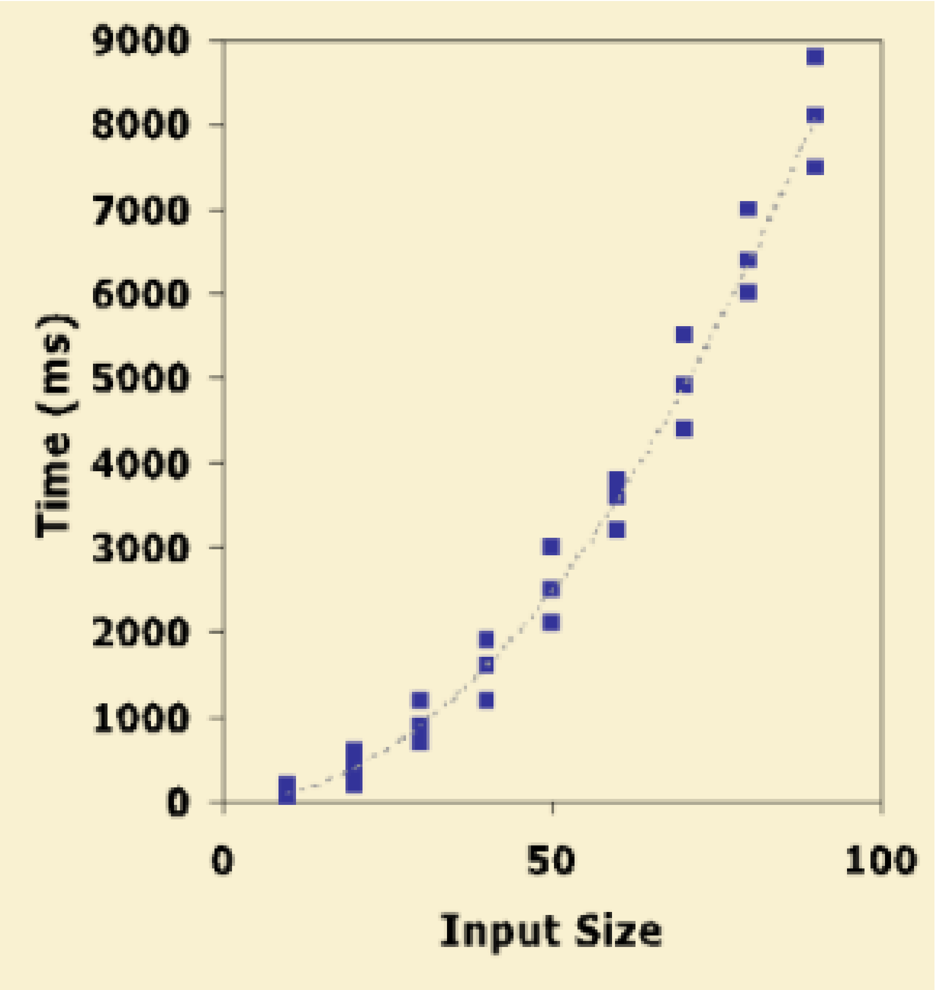
How to analyze algorithms?

Time Complexity

- As said earlier, we will focus on time complexity
- That is, to analyze how much time does an algorithm take to run to its completion
- There are ***Two ways*** with which we can do this!

Experimental Analysis

- Write a program implementing the algorithm
- Run it with inputs of varying size and composition
- Measure the actual running time
- Plot the results



What is wrong with this approach?

Theoretical Analysis

- **Pseudocode** description of the algorithm instead of an implementation
  - Characterize running time as a function of the input size, n --- $T(n)$
  - Allows us to evaluate the running time of an algorithm independent of the hardware/software environment

Pseudocode

- A high-level description of an algorithm
- More structured than Eng prose
- Less detailed than a program

Example: find max element of an array

Algorithm ***arrayMax***(***A***, ***n***)

Input: array ***A*** of ***n*** integers

Output: maximum element of ***A***

```
currentMax ← A[0]
for i ← 1 to n − 1 do
  if A[i] > currentMax then
    currentMax ← A[i]
return currentMax
```

Input Size (n)

- The n could be
  - The number of items in a container
  - The length of a string or file
  - The number of digits (or bits) in an integer
  - The degree of a polynomial

Measuring Time Complexity

- Even for inputs of the same size, the time consumed can be very different

Example: an algorithm that finds the first prime number in an array by scanning it left to right

How different situations can affect the running time of this algorithm?

Measuring Time Complexity

- Analyze running time for the
  - ***best case:*** usually useless
  - ***average case:*** very difficult to determine
  - ***worst case:*** a safer choice

Why is the worst case a safer choice?

How to Measure $T(n)$?

- Consider this statement in your algorithm

$$x = x + 1;$$

- What we want to measure is

❖ ***Execution time:*** The time a single execution of this statement would take

❖ ***Frequency count:*** The number of times it is executed

Execution Time

- Tied to the underline machine and compiler

- To simplify this, we use the ***RAM*** model

1. Each simple operation (+, *, -, =, if, call) takes exactly one step
2. Loops and subroutines are not considered simple operations
3. Each memory access takes exactly one time stamp

Measuring Time Complexity

- Total time taken by each statement is ***approximately*** the product of **execution time** (represented as constants) and the **frequency count**

Example

INSERTION-SORT(A) *cost* *times*

```
for j = 2 to A.length
    key = A[j]
    // Insert A[j] into the sorted
          sequence A[1 .. j - 1].
    i = j - 1
    while i > 0 and A[i] > key
        A[i + 1] = A[i]
        i = i - 1
    A[i + 1] = key
```

Example

We know that

$$\sum_{j=2}^{n} j = \frac{n(n+1)}{2} - 1$$

and

$$\sum_{j=2}^{n} (j-1) = \frac{n(n-1)}{2}$$

Example

We know that

$$\sum_{j=2}^{n} j = \frac{n(n+1)}{2} - 1$$

and

$$\sum_{j=2}^{n} (j-1) = \frac{n(n-1)}{2}$$

Thus

$$\begin{aligned} T(n) &= c_1 n + c_2(n-1) + c_4(n-1) + c_5\left(\frac{n(n+1)}{2} - 1\right) \\ &\quad + c_6\left(\frac{n(n-1)}{2}\right) + c_7\left(\frac{n(n-1)}{2}\right) + c_8(n-1) \\ &= \left(\frac{c_5}{2} + \frac{c_6}{2} + \frac{c_7}{2}\right) n^2 + \left(c_1 + c_2 + c_4 + \frac{c_5}{2} - \frac{c_6}{2} - \frac{c_7}{2} + c_8\right) n \\ &\quad - (c_2 + c_4 + c_5 + c_8) . \end{aligned}$$

Example

This can be expressed as

$$T(n) = an^2 + bn + c$$

Thus the machinery that we have developed, enabled us to express the time Complexity of Insertion Sort as a function of its input, independent of underlying platform, hardware, programming language and so on.

But what can we do with it?

Time Complexity

- Time complexities of algorithms when expressed in the form of numerical functions over the size of the input are difficult to work with:

  - ❖ Have too many bumps

  - ❖ Require too much detail to specify

- Thus, to make analysis easier, we talk about upper and lower bounds of these functions – **Big Oh Analysis**

- This helps ignore details that do not impact our comparison of algorithms

Big Oh Analysis

- In simple words, we can ignore
  - constant factors
  - lower-order terms
- Examples:
  - $10^2 n + 10^5$ is a ***linear function***
  - $10^2 n^2 + 10^5 n$ is a ***quadratic function***

Big Oh Analysis

- Why is it not affected by the constant factors and the lower order terms?

- $6n$ ***vs.*** $3n$ — getting a computer twice as fast makes the former same as the latter

- $2n$ ***vs.*** $2n + 8$ — difference becomes insignificant when n becomes larger and larger

- x^3***vs.*** kx^2 — the former will always eventually overtake the latter no matter how big you make k

Big Oh Analysis

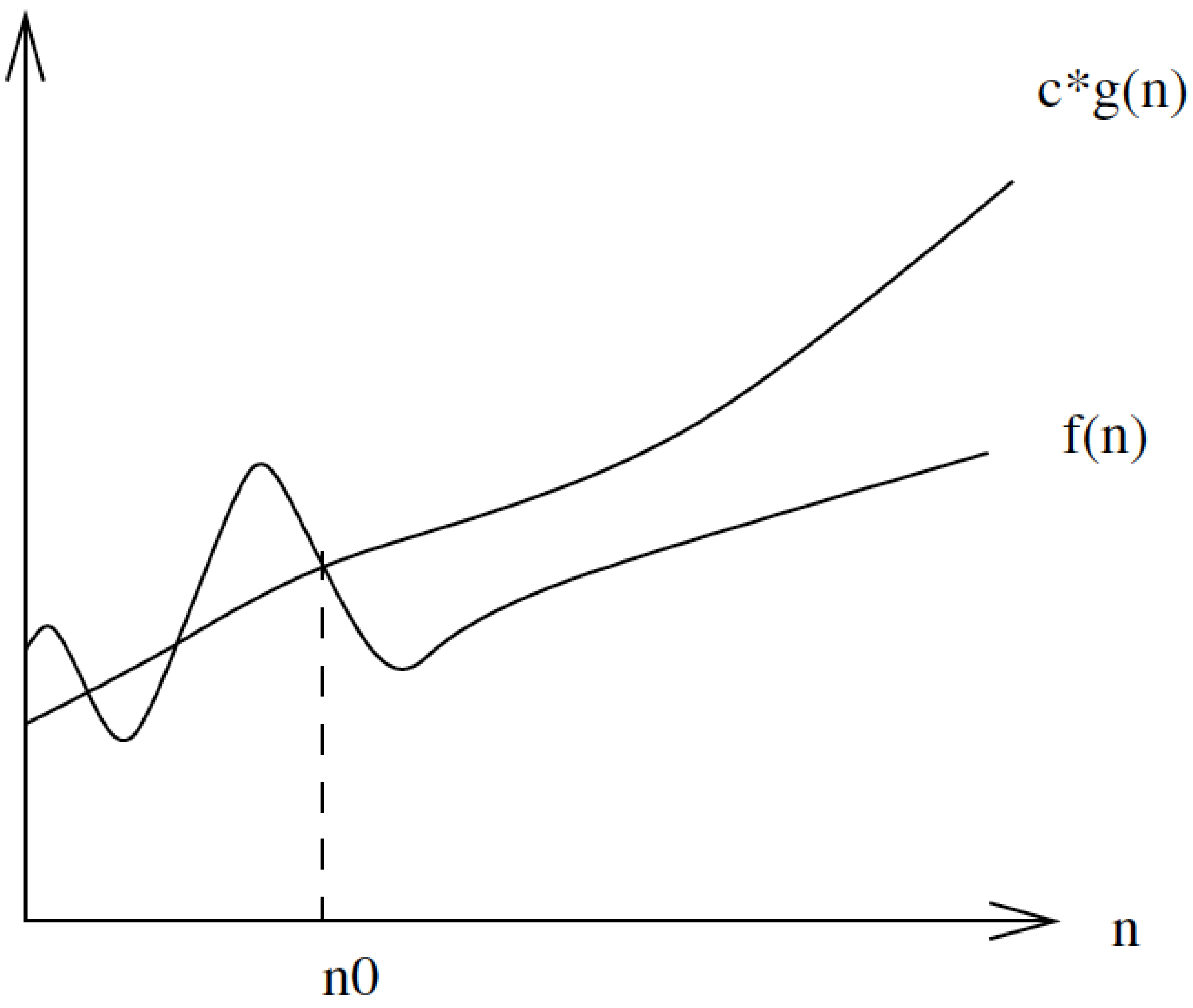
- So for our example: $T(n) = an^2 + bn + c$
- But we just learned that constant terms and lower order terms don’t matter
- Thus, under Big Oh analysis, we can express it as

$$T(n) = O(n^2)$$

Big Oh Notations

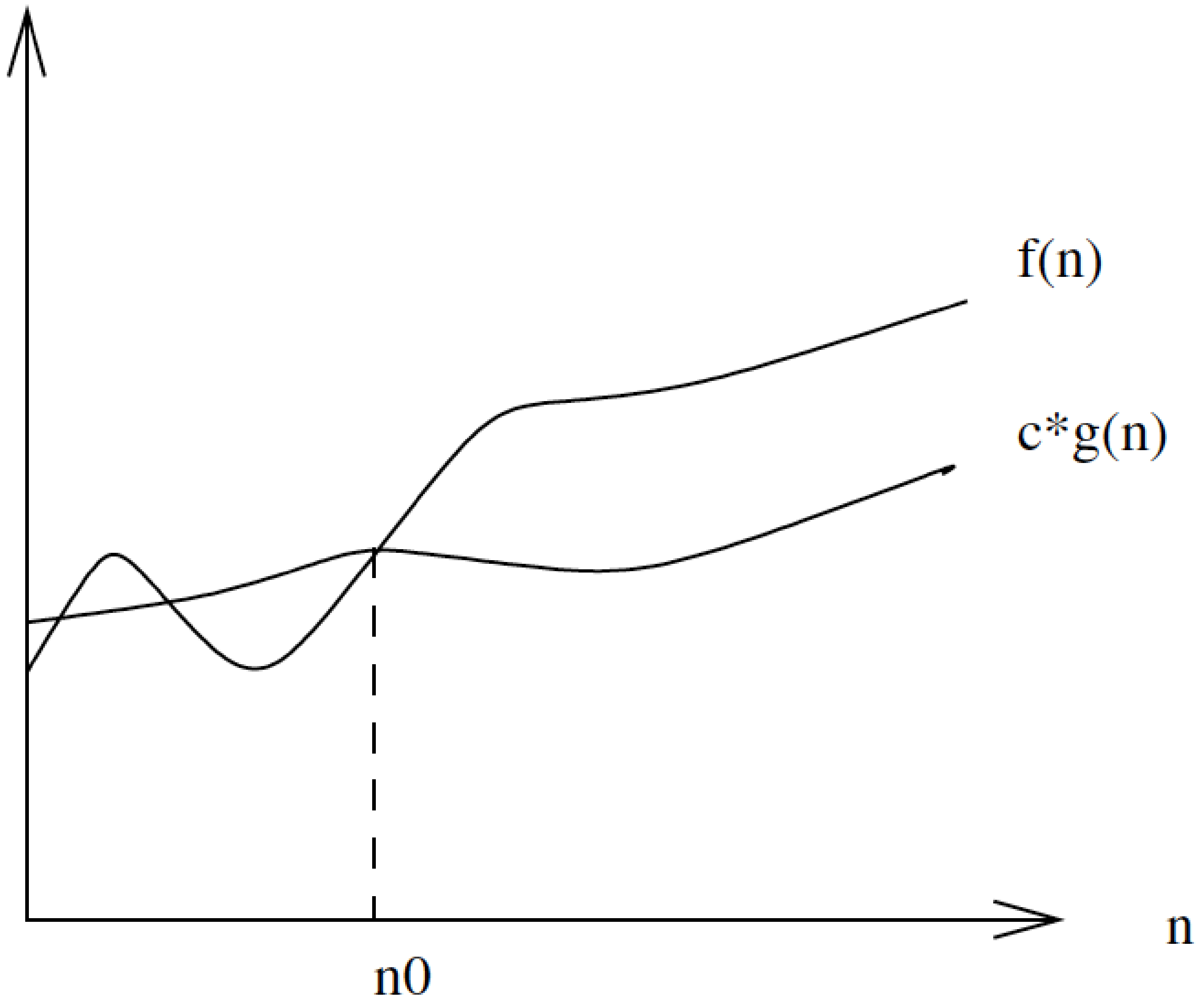
- $f(n) = O(g(n))$ means $c \cdot g(n)$ is an *upper bound* on $f(n)$. Thus there exists some constant c such that $f(n)$ is always $\leq c \cdot g(n)$, for large enough n (i.e. , $n \geq n_0$ for some constant n_0).
- $f(n) = \Omega(g(n))$ means $c \cdot g(n)$ is a *lower bound* on $f(n)$. Thus there exists some constant c such that $f(n)$ is always $\geq c \cdot g(n)$, for all $n \geq n_0$.
- $f(n) = \Theta(g(n))$ means $c_1 \cdot g(n)$ is an upper bound on $f(n)$ and $c_2 \cdot g(n)$ is a lower bound on $f(n)$, for all $n \geq n_0$. Thus there exist constants c_1 and c_2 such that $f(n) \leq c_1 \cdot g(n)$ and $f(n) \geq c_2 \cdot g(n)$. This means that $g(n)$ provides a nice, tight bound on $f(n)$.

Big Oh - O



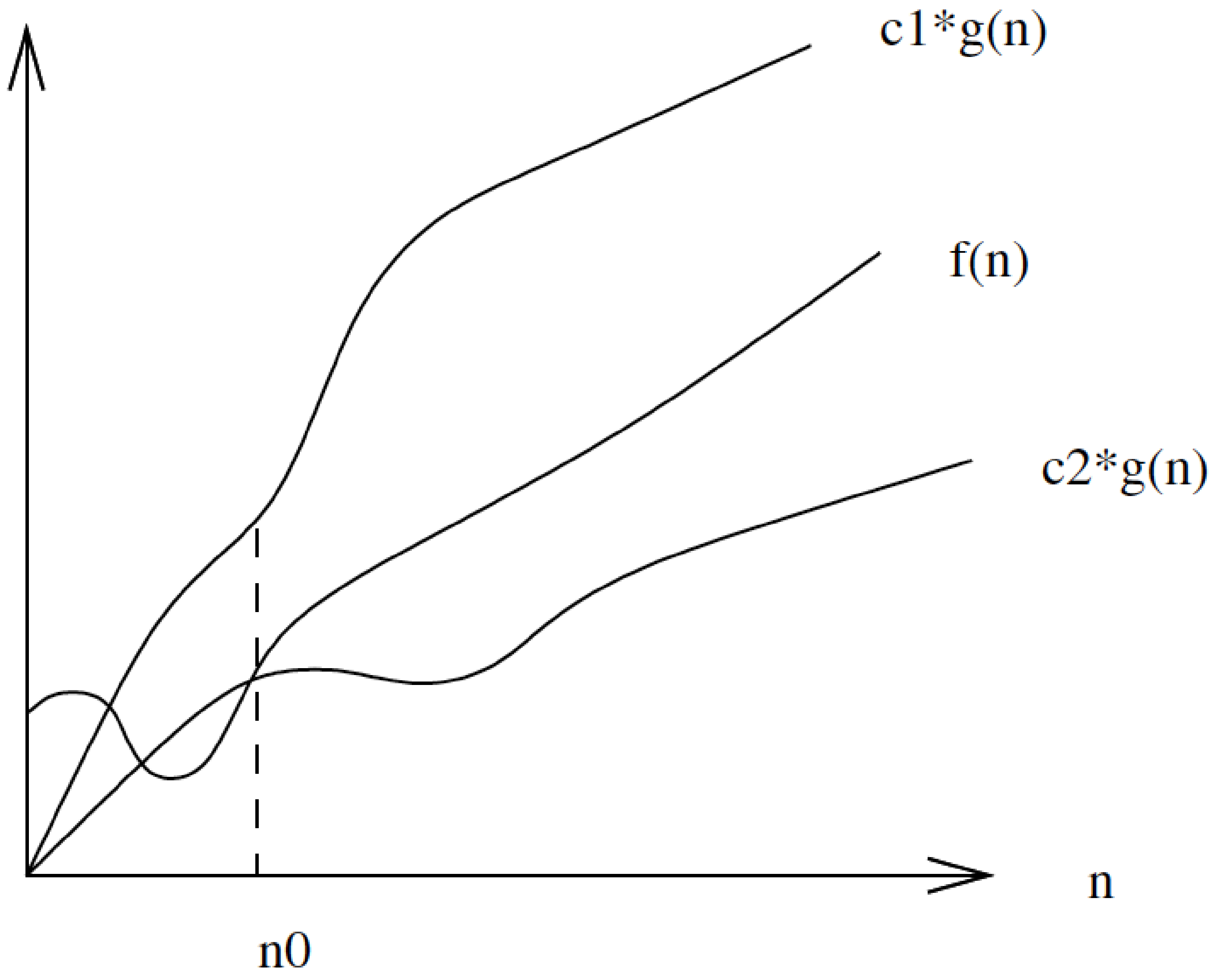
$f(n) = O(g(n))$ means $c \cdot g(n)$ is an *upper bound* on $f(n)$. Thus there exists some constant c such that $f(n)$ is always $\leq c \cdot g(n)$, for large enough n (i.e. , $n \geq n_0$ for some constant n_0).

Big Omega - Ω



$f(n) = \Omega(g(n))$ means $c \cdot g(n)$ is a *lower bound* on $f(n)$. Thus there exists some constant c such that $f(n)$ is always $\geq c \cdot g(n)$, for all $n \geq n_0$.

Big Theta - Θ



$f(n) = \Theta(g(n))$ means $c_1 \cdot g(n)$ is an upper bound on $f(n)$ and $c_2 \cdot g(n)$ is a lower bound on $f(n)$, for all $n \geq n_0$. Thus there exist constants c_1 and c_2 such that $f(n) \leq c_1 \cdot g(n)$ and $f(n) \geq c_2 \cdot g(n)$. This means that $g(n)$ provides a nice, tight bound on $f(n)$.

What do you think?

Is $2^{n+1} = \Theta(2^n)$?

What do you think?

Is $(x + y)^2 = O(x^2 + y^2)$.

Properties

- Transitivity
- Reflexivity
- Symmetry
- Transpose Symmetry

Growth Rates of Common Functions

n $f(n)$	$\lg n$	n	$n \lg n$	n^2	2^n	$n!$
10	0.003 μs	0.01 μs	0.033 μs	0.1 μs	1 μs	3.63 ms
20	0.004 μs	0.02 μs	0.086 μs	0.4 μs	1 ms	77.1 years
30	0.005 μs	0.03 μs	0.147 μs	0.9 μs	1 sec	8.4×10^{15} yrs
40	0.005 μs	0.04 μs	0.213 μs	1.6 μs	18.3 min	
50	0.006 μs	0.05 μs	0.282 μs	2.5 μs	13 days	
100	0.007 μs	0.1 μs	0.644 μs	10 μs	4×10^{13} yrs	
1,000	0.010 μs	1.00 μs	9.966 μs	1 ms		
10,000	0.013 μs	10 μs	130 μs	100 ms		
100,000	0.017 μs	0.10 ms	1.67 ms	10 sec		
1,000,000	0.020 μs	1 ms	19.93 ms	16.7 min		
10,000,000	0.023 μs	0.01 sec	0.23 sec	1.16 days		
100,000,000	0.027 μs	0.10 sec	2.66 sec	115.7 days		
1,000,000,000	0.030 μs	1 sec	29.90 sec	31.7 years		

Space Complexity

- Determine how much space an algorithm requires by analyzing its storage requirements as a function of the input size

- ***Example:***

  - Let’s say, our algorithm reads a stream of ***n*** characters

  - But always <u>stores a constant number</u> of them

  - then, its space complexity is ***O(1)***

Space Complexity

- ***Another Example:***
  - Let's say, our algorithm reads a stream of ***n*** characters
  - and <u>stores all</u> of them
  - then, its space complexity is ***O(n)***

Space Complexity

- ***Exercise:***

  - Let’s say, our algorithm reads a stream of ***n*** characters

  - and <u>stores all</u> of them, and each record results in the creation of a <u>constant number</u> of other records

  - then, its space complexity is ?

Space Complexity

- ***Another Exercise:***
  - Let's say, our algorithm reads a stream of ***n*** characters
  - and <u>stores all</u> of them, and each record results in the creation of a number of new records — the <u>number is proportional to the size of the data</u>
  - then, its space complexity is ?

Time-Space Tradeoff

- Generally, decreasing the time complexity of an algorithm results in increasing its space complexity — and vice versa

- This is called the time-space tradeoff

- ***Example:*** Storing a sparse matrix as a two-dimensional linked list vs. a two-dimensional array

Did we achieve today’s objectives?

- What is “Algorithm Analysis”?
- Why should we analyze algorithms?
- Understand mathematical machinery needed to analyze algorithms
- Learn what it means for one function to grow faster than another