CSC 212: Data Structures and Abstractions

Big O Notation

Marco Alvarez

Department of Computer Science and Statistics
University of Rhode Island

Fall 2020

Example Review

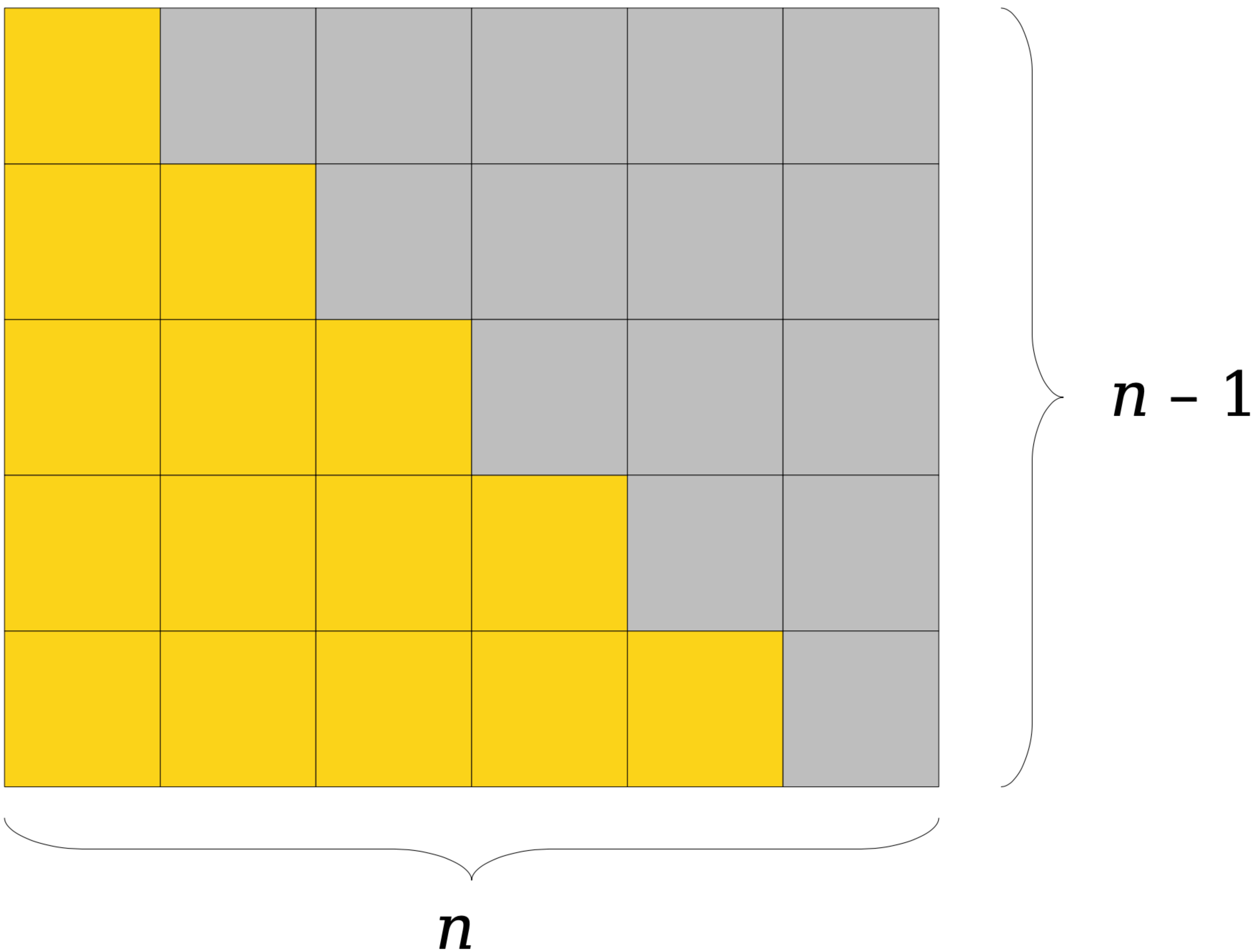
```
for (int i = 0 ; i < n ; i ++) {
    for (int j = 0 ; j < i*i ; j ++) {
        for (int k = 0 ; k < j ; k ++) {
            // count 1 instruction
        }
    }
}
```

The story so far …

- Can measure actual runtime to compare algorithms
  - however, runtime is noisy (highly sensitive to HW/SW and implementation details)
- Can count instructions to compare algorithms
  - can define T(n), which depends on the input size
  - for large inputs, our focus should be on the dominant terms of T(n)
  - we will now see formal ways for this approach

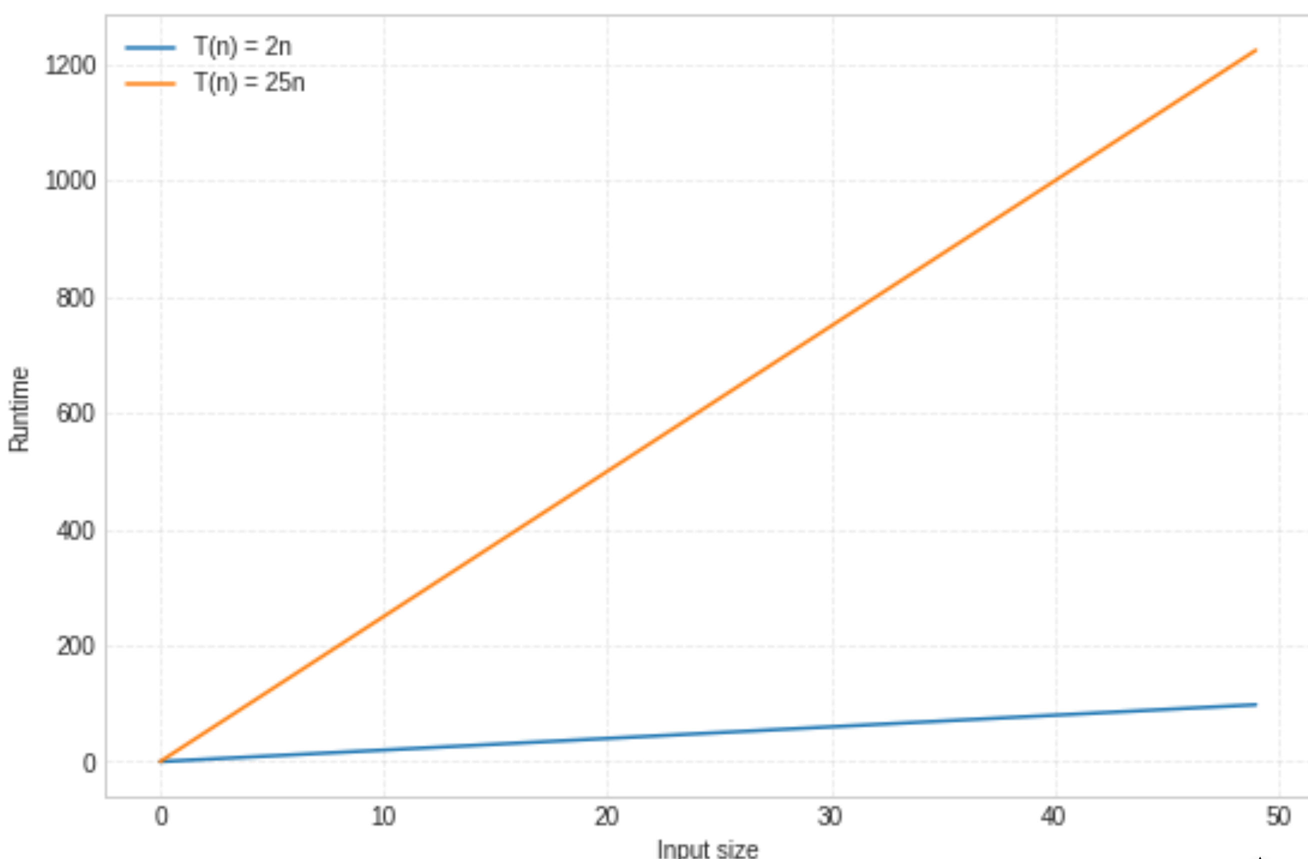
$$\sum_{i=1}^{n} i = 1 + 2 + 3 + \ldots + (n-1) + n$$

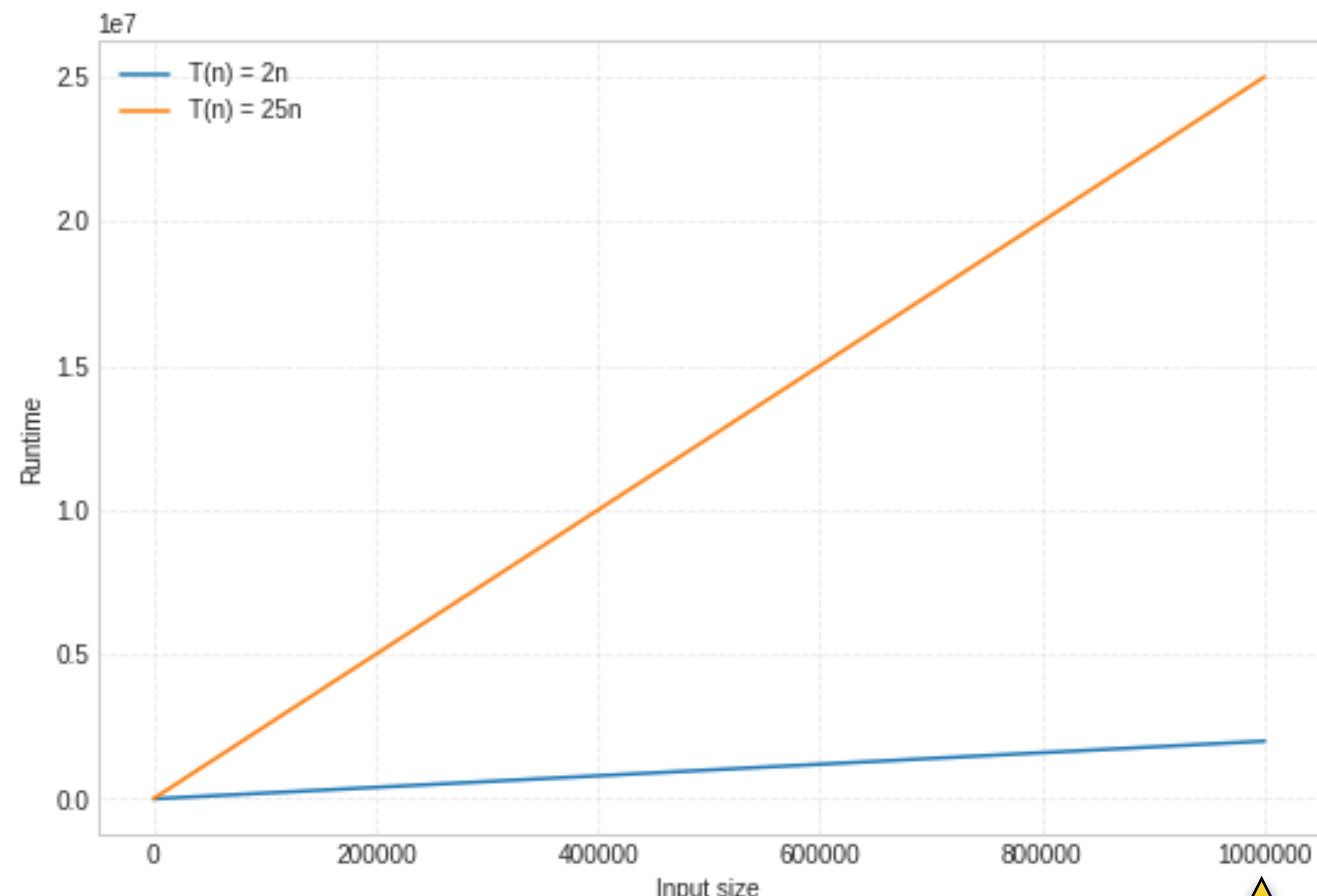
$$\sum_{i=1}^{n-1} i = 1 + 2 + 3 + \ldots + (n-2) + (n-1)$$



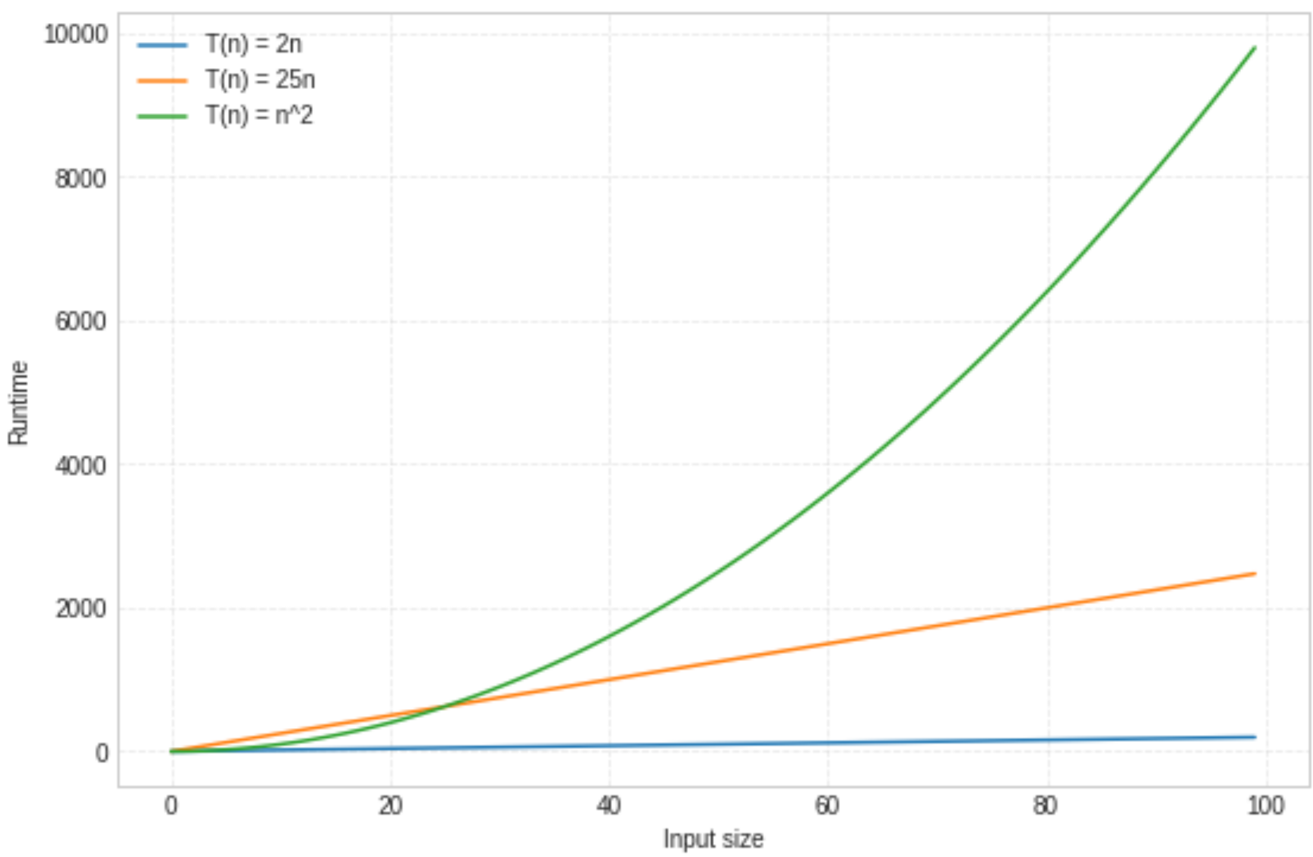
Should we consider these the same?

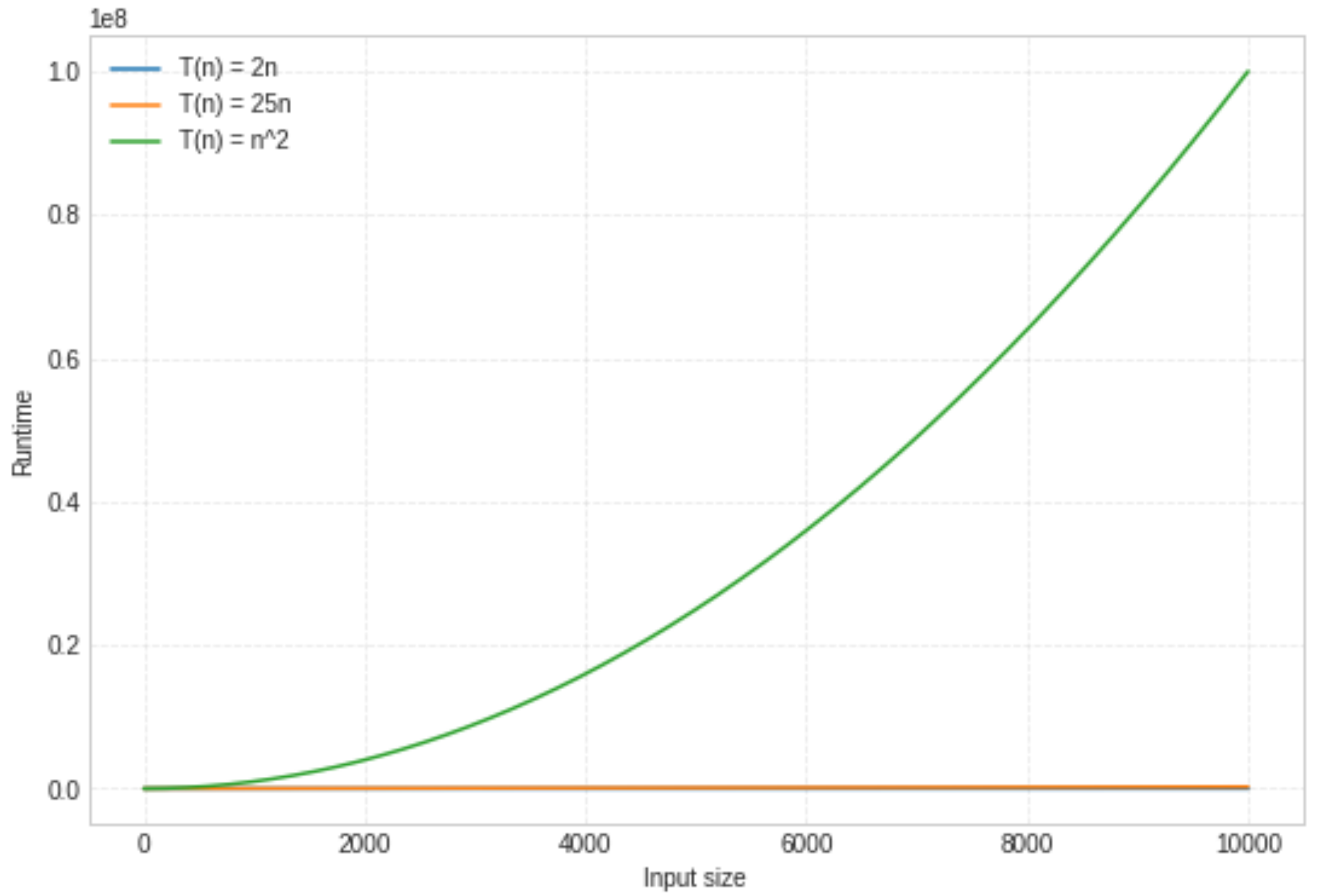
Comparing two algorithms with $T(n) = 2n$ and $T(n) = 25n$ respectively



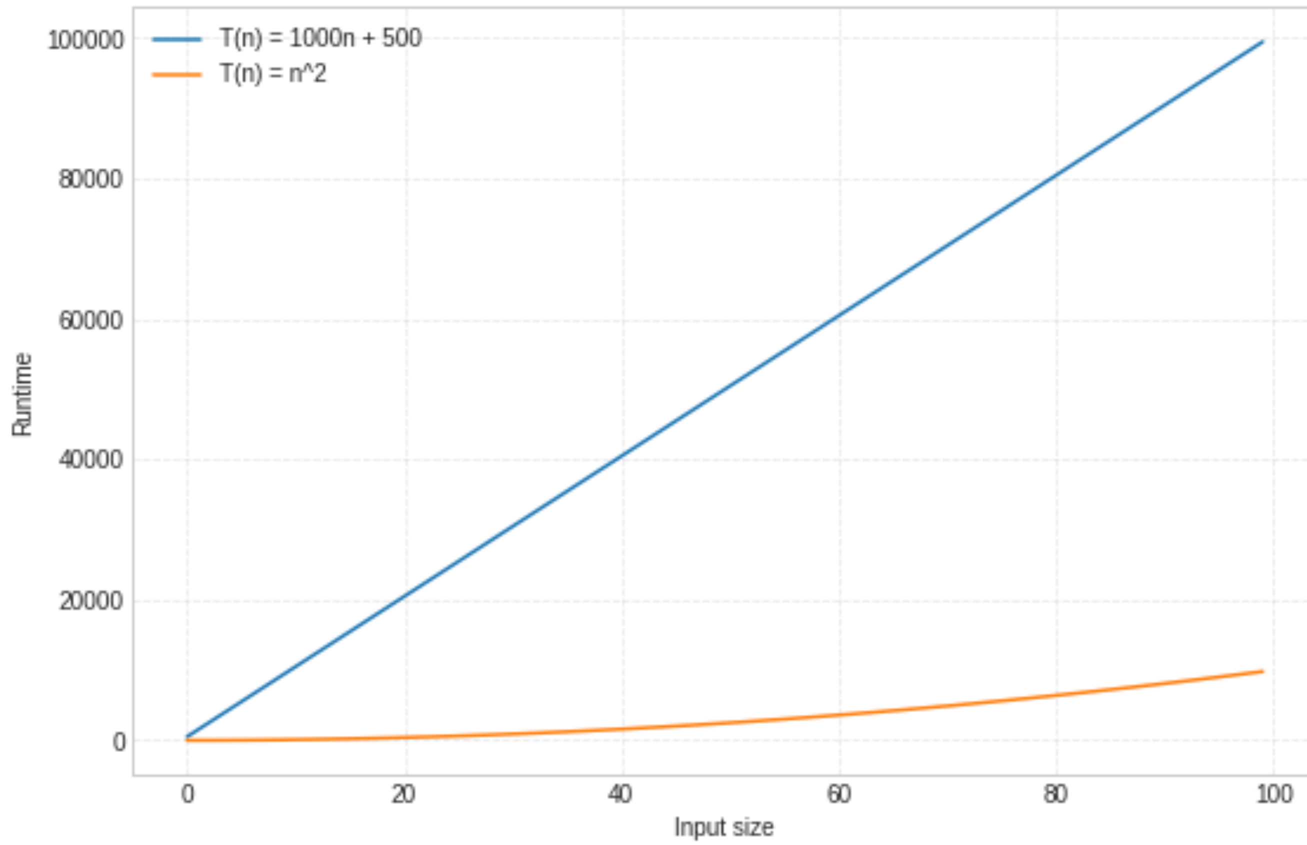


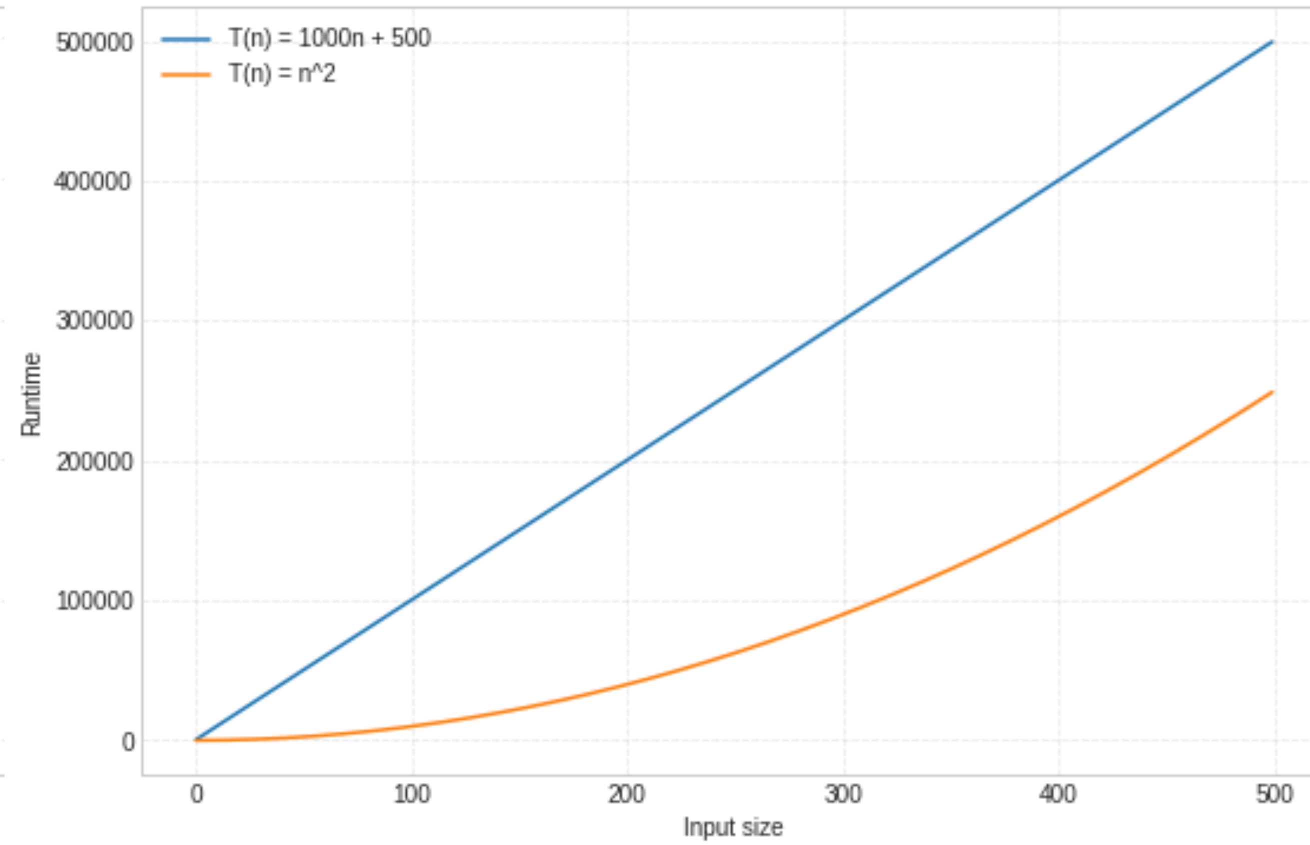
What about now?

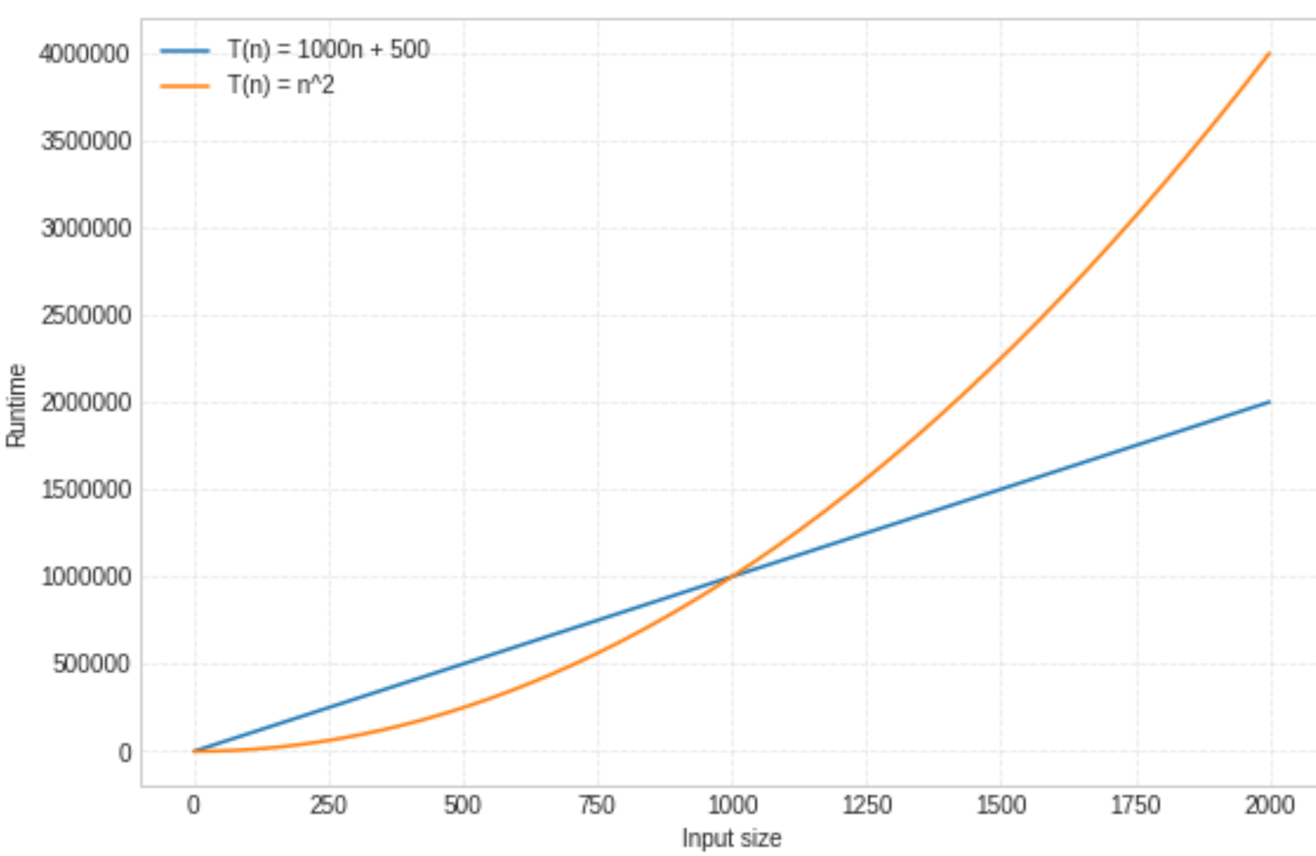


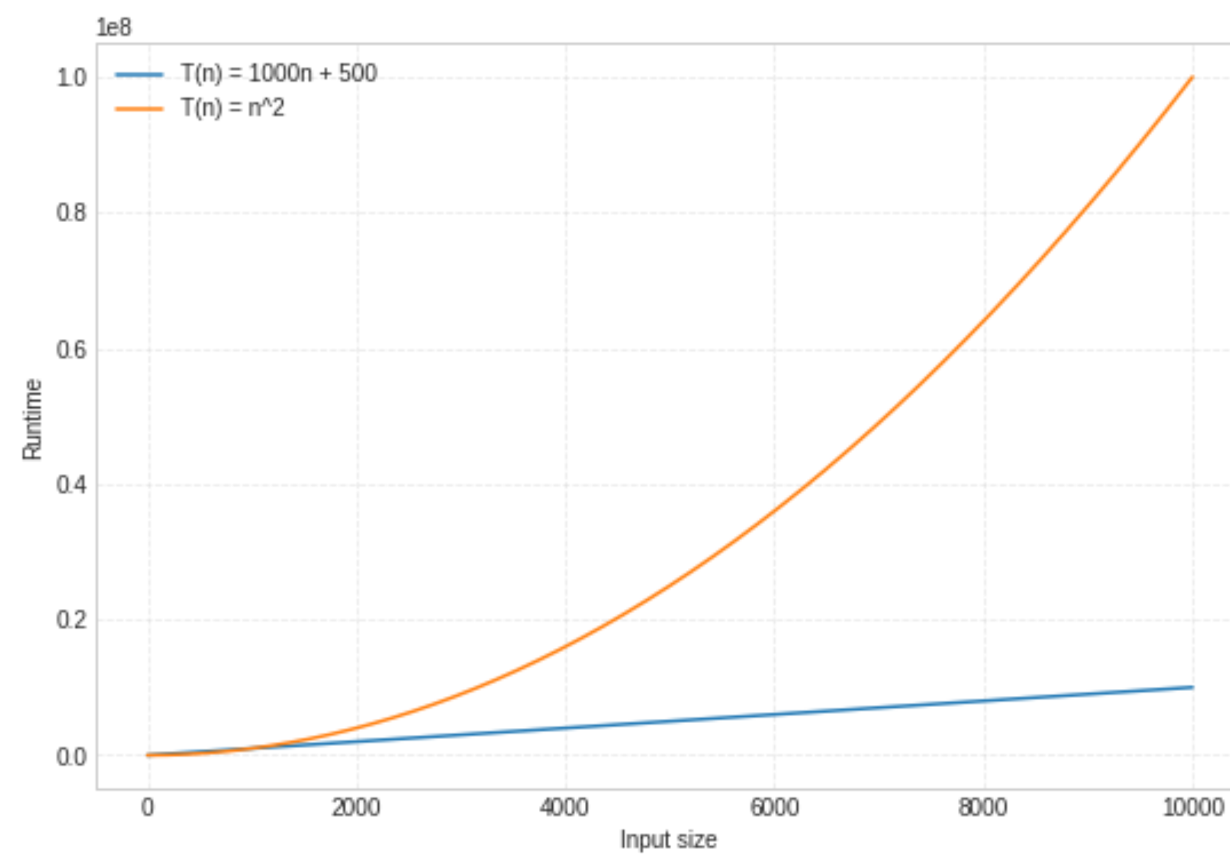


Now consider this example …





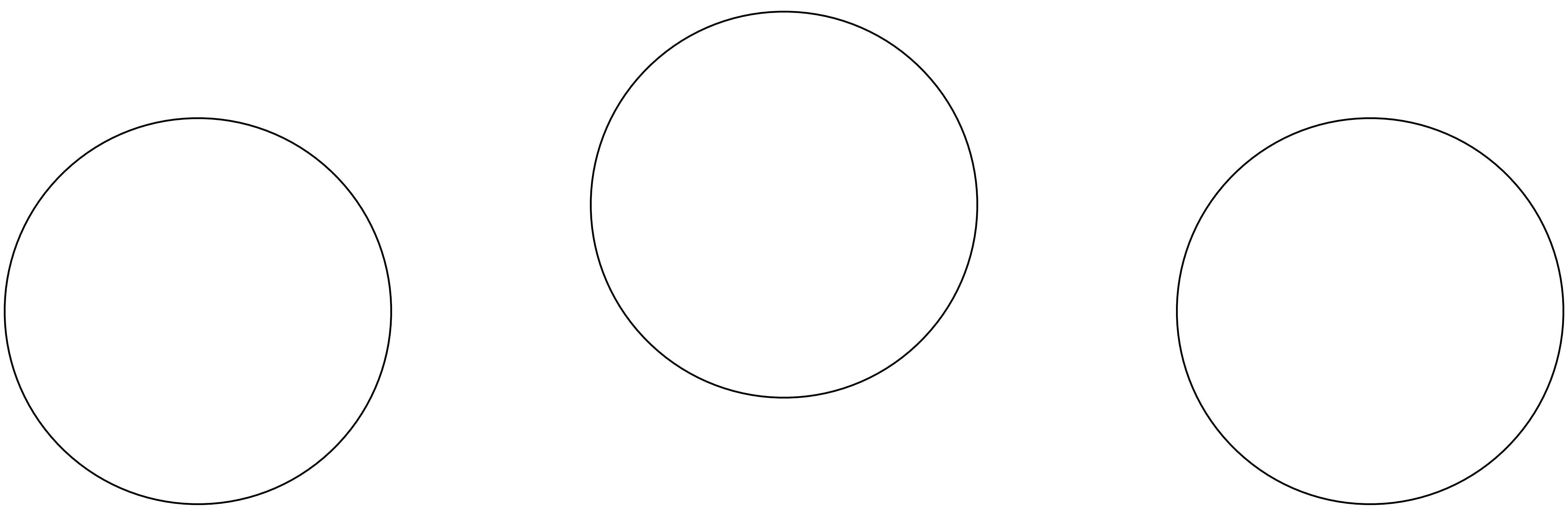




Bottom line …

- We are trying to compare **T(n)** functions, but we also care about large values of **n**

- Can we properly define ‘<=‘ for functions?
  - we can group functions into **‘sets’** and make our lives easier



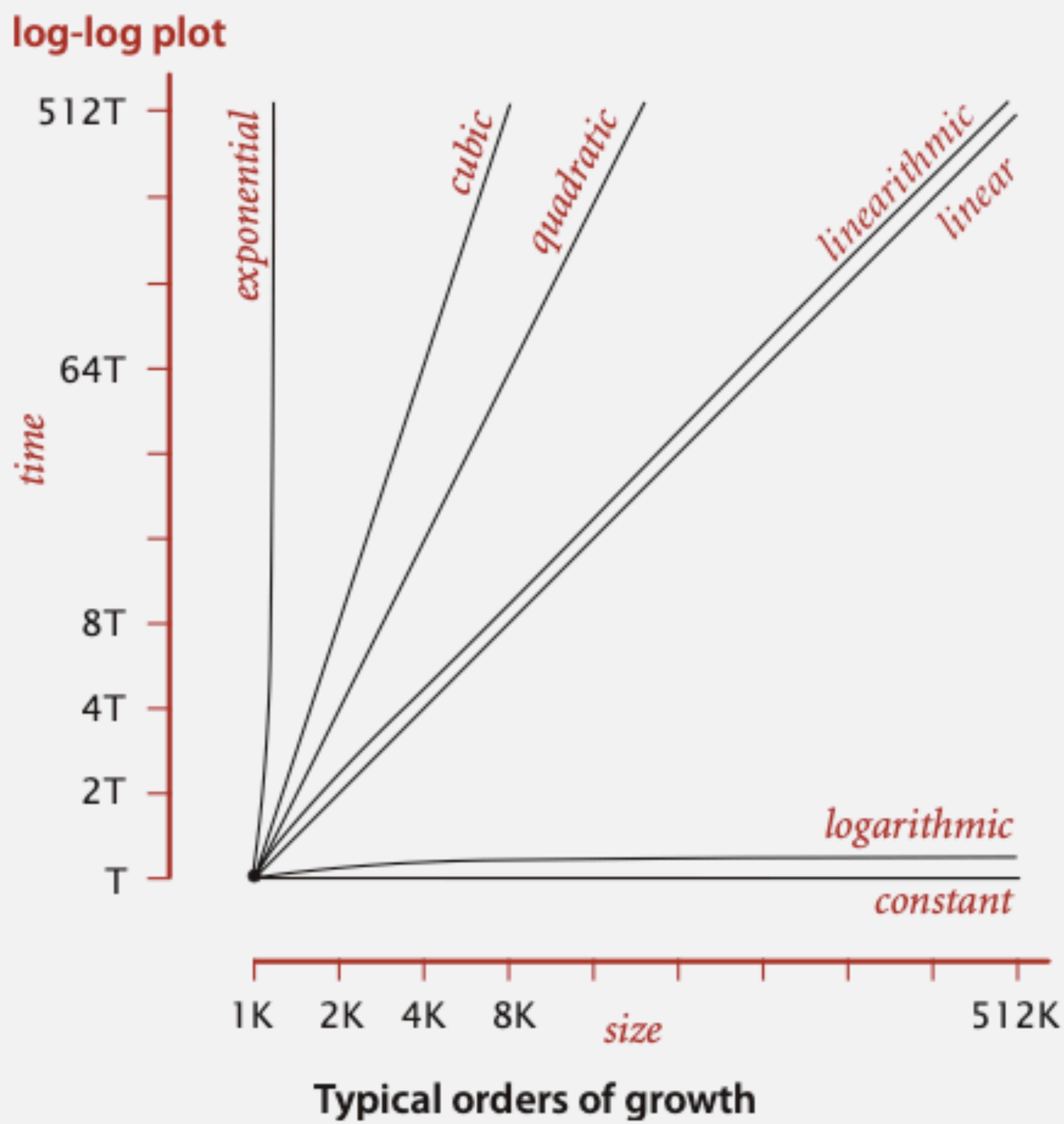
Asymptotic Analysis

- Refers to the study of an algorithm as the input size "gets big" or reaches a limit (in the calculus sense)

- **Growth rate**
  - rate at which the cost of an algorithm grows as the size of its input grows

$$c_1 n \qquad\qquad c_2 n^2$$

Common sets of functions



Faster growth rate … slower algorithm

Algorithm A is better than algorithm B if for large values of n, $T_A(n)$ grows slower than $T_B(n)$

credit: COS 226, Princeton

A few examples ...

order of growth	name	typical code framework	description	example
1	**constant**	`a = b + c;`	statement	add two numbers
$\log n$	**logarithmic**	`while (n > 1)` `{ n = n/2; ... }`	divide in half	binary search
n	**linear**	`for (int i = 0; i < n; i++)` `{ ... }`	single loop	find the maximum
$n \log n$	**linearithmic**	*see mergesort lecture*	divide and conquer	mergesort

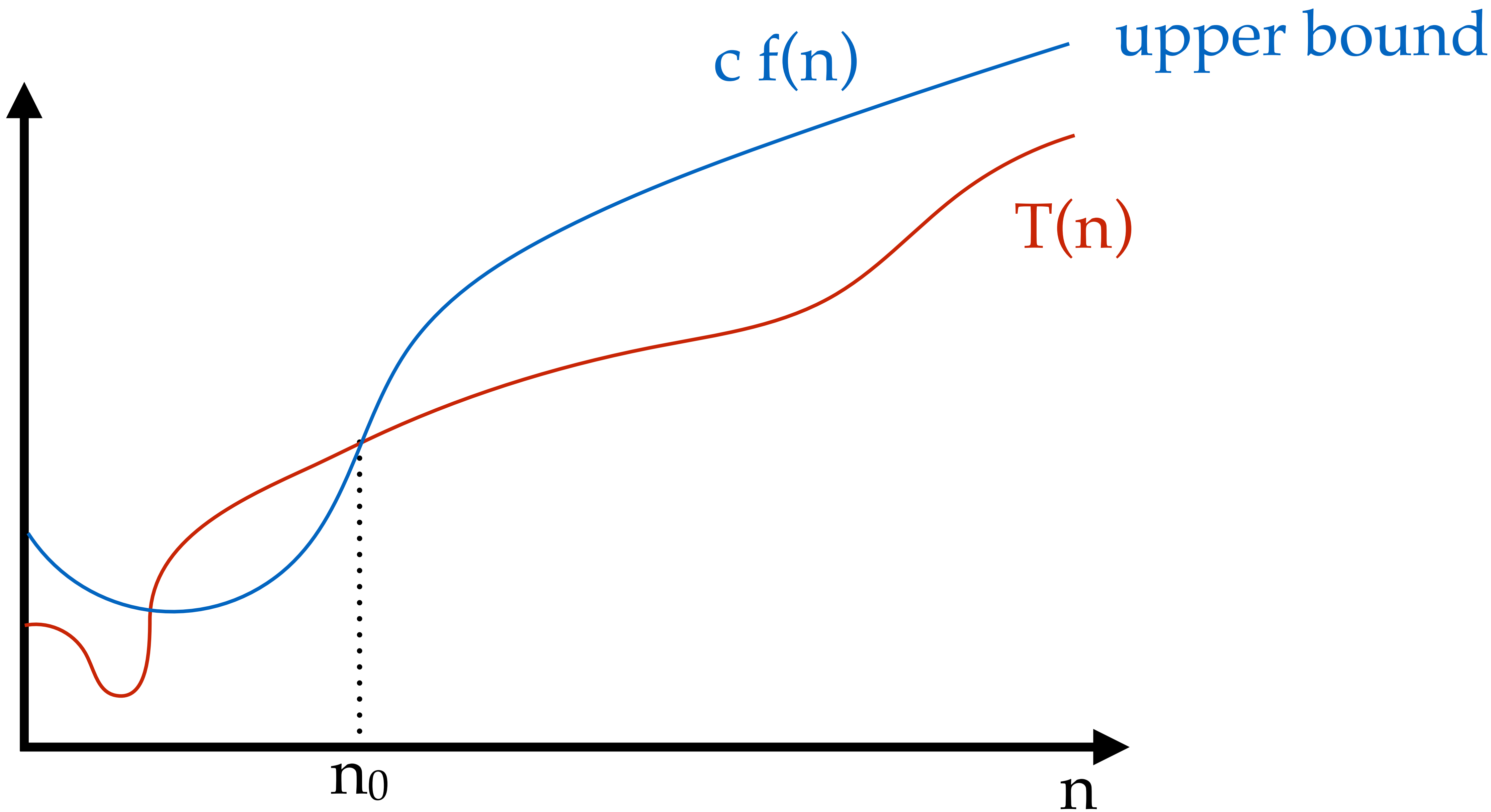
n^2	**quadratic**	`for (int i = 0; i < n; i++)`
		`for (int j = 0; j < n; j++)`
		`{ ... }`
			double loop	check all pairs

n^3	**cubic**	`for (int i = 0; i < n; i++)`
		`for (int j = 0; j < n; j++)`
		`for (int k = 0; k < n; k++)`
		`{ ... }`
			triple loop	check all triples

2^n	**exponential**	*see combinatorial search lecture*	exhaustive search	check all subsets

credit: COS 226, Princeton

Big O



$$T(n) \text{ is } O(f(n)) \iff \exists \text{ positive } c, n_0 \mid 0 \le T(n) \le cf(n), \forall n \ge n_0$$

set of functions

Examples

$$7n - 2 = O(n)$$

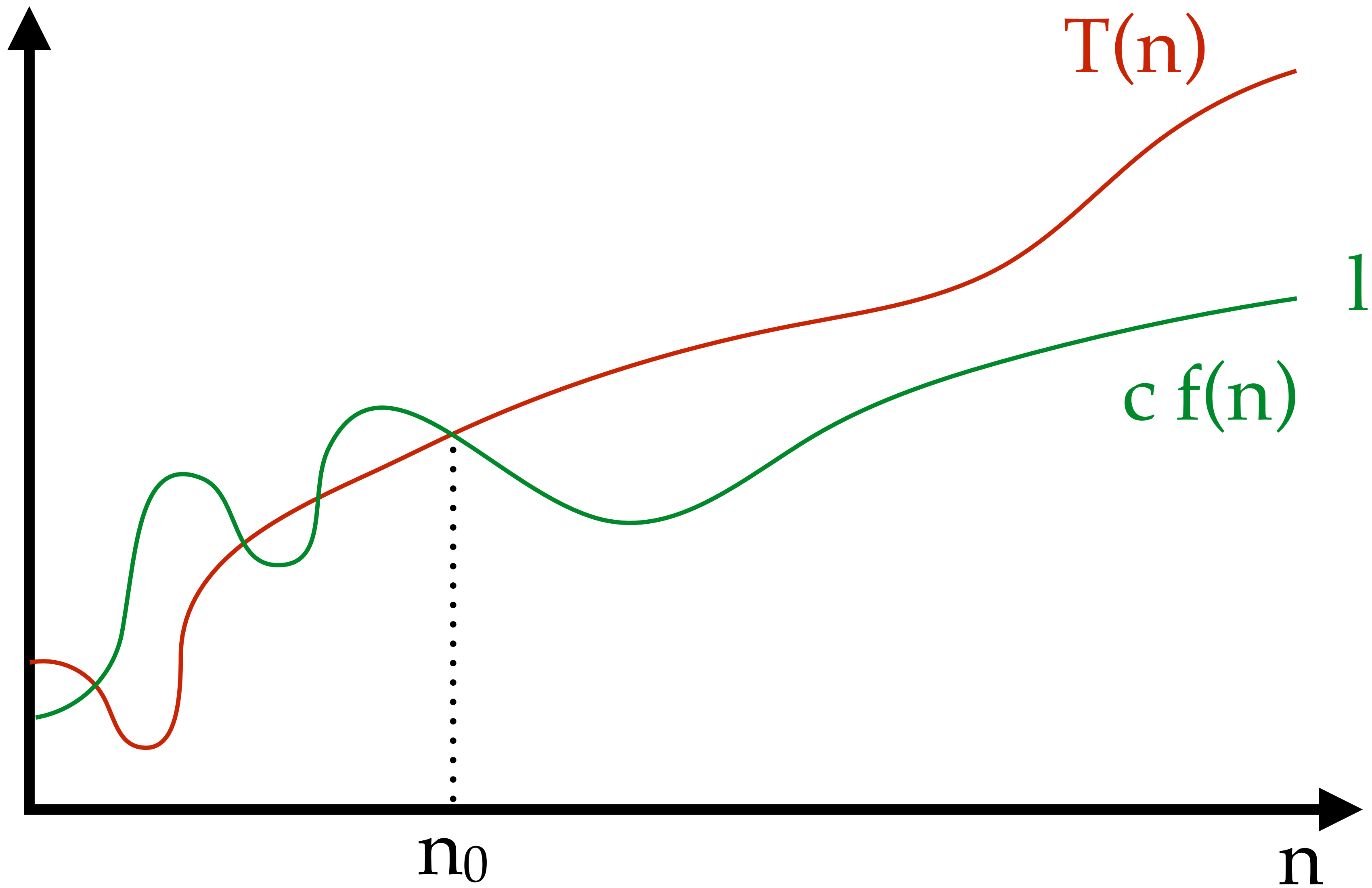
$$20n^3 + 10n \log n + 5 = O(n^3)$$

$$3 \log n + \log \log n = O(\log n)$$

$$2^{100} = O(1)$$

$$T(n) \text{ is } O(f(n)) \iff \exists \text{ positive } c, n_0 \mid 0 \leq T(n) \leq cf(n), \forall n \geq n_0$$

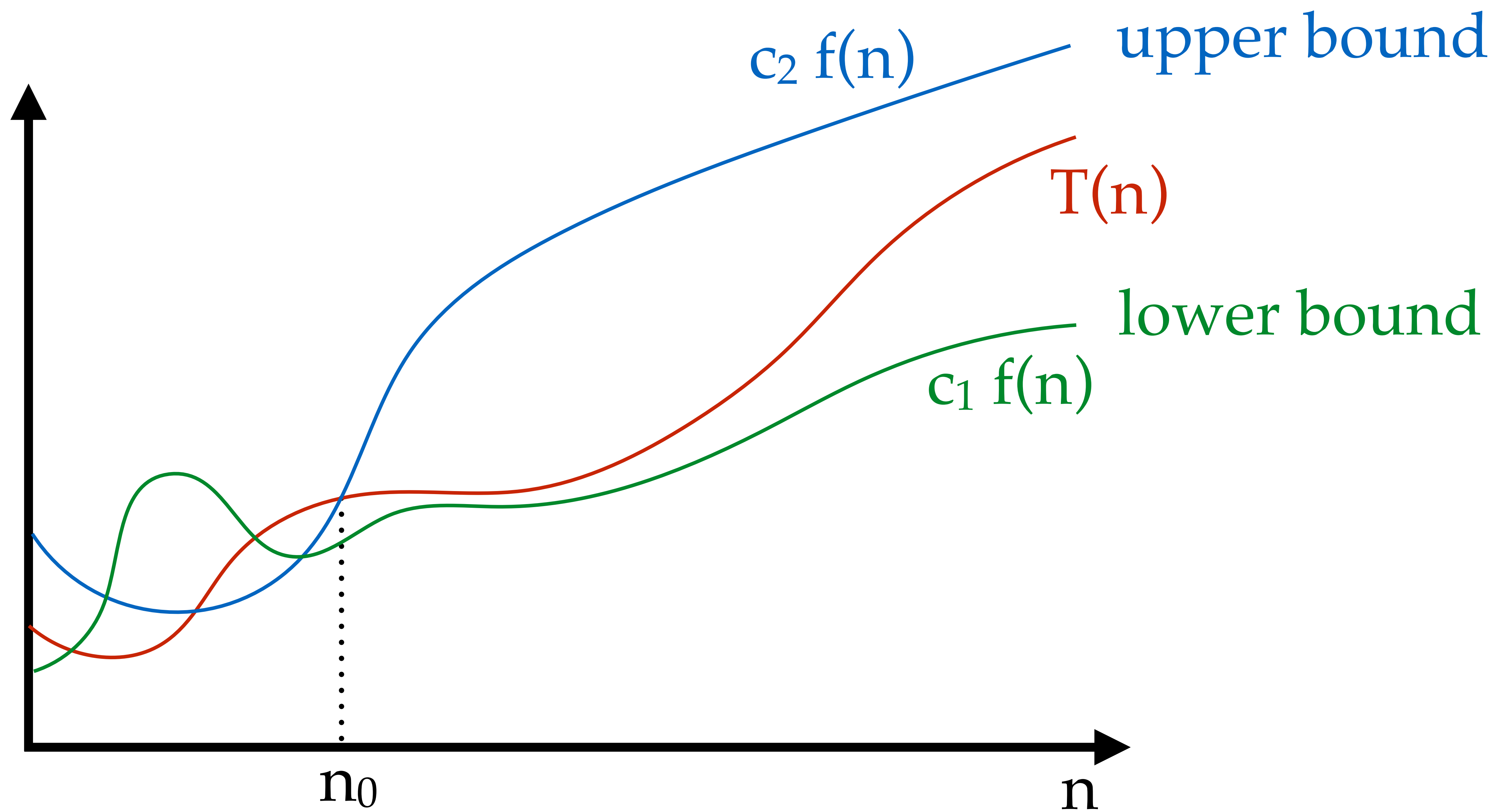
Big Omega



$$T(n) \text{ is } \Omega(f(n)) \iff \exists \text{ positive } c, n_0 \mid 0 \le cf(n) \le T(n), \forall n \ge n_0$$

set of functions

Big Theta



$$T(n) \text{ is } \Theta(f(n)) \iff T(n) \text{ is } O(f(n)) \text{ and } T(n) \text{ is } \Omega(f(n))$$

Prove that …

$$3\log n + \log\log n = \Omega(\log n)$$

$$3\log n + \log\log n = \Theta(\log n)$$

$T(n)$ is $O(f(n)) \iff \exists$ positive $c, n_0 \mid 0 \le T(n) \le cf(n), \forall n \ge n_0$
$T(n)$ is $\Omega(f(n)) \iff \exists$ positive $c, n_0 \mid 0 \le cf(n) \le T(n), \forall n \ge n_0$

In practice you can …

“ignore constants and drop lower order terms”

True or False?

$\{n^2, n^4, 2^n, \log n, \ldots\}$

	Big O	Big Omega	Big Theta
$10^2 + 3000n + 10$			
$21 \log n$			
$500 \log n + n^4$			
$\sqrt{n} + \log n^{50}$			
$4^n + n^{5000}$			
$3000n^3 + n^{3.5}$			
$2^5 + n!$			

Asymptotic Performance

- For large values of **n**, a $\mathbf{\Theta(n^2)}$ algorithm always beats a $\mathbf{\Theta(n^3)}$ algorithm

However, we shouldn't completely ignore asymptotically slower algorithms