COMP251: Running time analysis and the Big O notation

Jérôme Waldispühl & Roman Sarrazin-Gendron

School of Computer Science
McGill University

Based on slides from M. Langer and M. Blanchette

Outline

- Motivations
- The Big O notation
  - Definition
  - Examples
  - Rules
- Big Omega and Big Theta
- Applications
- About recursive algorithms

Background

WHAT IS THE RUNNING TIME?

Measuring the running "time"

- Goal: Analyze an algorithm written in pseudocode and describe its running time
  - Without having to write code
  - In a way that is independent of the computer used
- To achieve that, we need to
  - Make simplifying assumptions about the running time of each basic (primitive) operations
  - Study how the number of primitive operations depends on the size of the problem solved

Primitive Operations

Simple computer operation that can be performed in time that is always the same, independent of the size of the bigger problem solved (we say: constant time)

- **Assigning a value to a variable:** x ←1 — T_{assign}
- **Calling a method:** Expos.addWin() — T_{call}
  - Note: doesn't include the time to execute the method
- **Returning from a method**: return x; — T_{return}
- **Arithmetic operations on primitive types** — T_{arith}

  x + y, r*3.1416, x/y, etc.
- **Comparisons on primitive types:** x==y — T_{comp}
- **Conditionals:** if (...) then.. else... — T_{cond}
- **Indexing into an array:** A[i] — T_{index}
- **Following object reference:** Expos.losses — T_{ref}

Note: Multiplying two Large Integers is *not* a primitive operation, because the running time depends on the size of the numbers multiplied.

FindMin analysis

```
Algorithm findMin(A, start, stop)
Input: Array A, index start & stop
Output: Index of the smallest element of A[start:stop]
```

```
minvalue ← A[start]
minindex ← start
index ← start + 1
while ( index <= stop ) do {
   if (A[index]<minvalue)
   then {
        minvalue ← A[index]
        minindex ← index
   }
   index = index + 1
}
return minindex
```

Running time

$T_{index} + T_{assign}$	
T_{assign}	
$T_{arith} + T_{assign}$	
$T_{comp} + T_{cond}$	repeated
$T_{index} + T_{comp} + T_{cond}$	stop-start
$T_{index} + T_{assign}$	times
T_{assign}	
$T_{assign} + T_{arith}$	
$T_{comp} + T_{cond}$ (last check of loop)	
T_{return}	

Worst case running time

- Running time depends on *n* = stop – start + 1
  - But it also depends on the content of the array!
- What kind of array of *n* elements will give the *worst case running time* for findMin?

Example:

5	4	3	2	1	0

- The *best case running time*?

Example:

0	1	2	3	4	5

More assumptions

- Counting each type of primitive operations is tedious
- The running time of each operation is roughly comparable:

$T_{assign} \approx T_{comp} \approx T_{arith} \approx \ldots \approx T_{index} = 1$ primitive operation

- We are only interested in the **number** of primitive operations performed

Worst-case running time for findMin becomes:

$$T(n) = 8 + 10 * (n\text{-}1)$$

8 primitive operations outside the loop

10 primitive operations inside the loop

Selection Sort

	Primitive operations (worst case) :
Algorithm SelectionSort(A,n)	
Input: an array A of n elements	
Output: the array is sorted	
i← 0	1
while (i<n) **do** {	2
minindex ← findMin(A,i,n-1)	3 + $T_{FindMin}$(n-1-i+1)=3 +(10 (n-i) - 2)
t ← A[minindex]	2
A[minindex] ← A[i]	3
A[i] ← t	2
i ← i + 1	2
}	2 (last check of loop condition)

Selection sort builds a sorted list by casting findMin on the unsorted region and swapping

Selection Sort: adding it up

Total: $T(n) = 1 + \left(\sum_{i=0}^{n-1} 12 + 10\,(n - i) \right) + 2$

$= 3 + \left(12\,n + 10 \sum_{i=0}^{n-1} (n-i) \right)$

$= 3 + 12\,n + 10 \left(\sum_{i=0}^{n-1} n \right) - 10 \left(\sum_{i=0}^{n-1} i \right)$

$= 3 + 12\,n + 10\,n * n - 10\,((n-1)*n) / 2)$

$= 3 + 12\,n + 10\,n^2 - 5\,n^2 + 5\,n$

$= \mathbf{5\,n^2 + 17\,n + 3}$

You can take the time to go through the math later, here the main idea is that we can sum up all the individual operations to estimate the running time.

Conceptualization

CLASSIFYING RUNNING TIMES

More simplifications

We have: $T(n) = 5\ n^2 + 17\ n + 3$

Simplification #1:

When n is large, $T(n) \approx 5\ n^2$

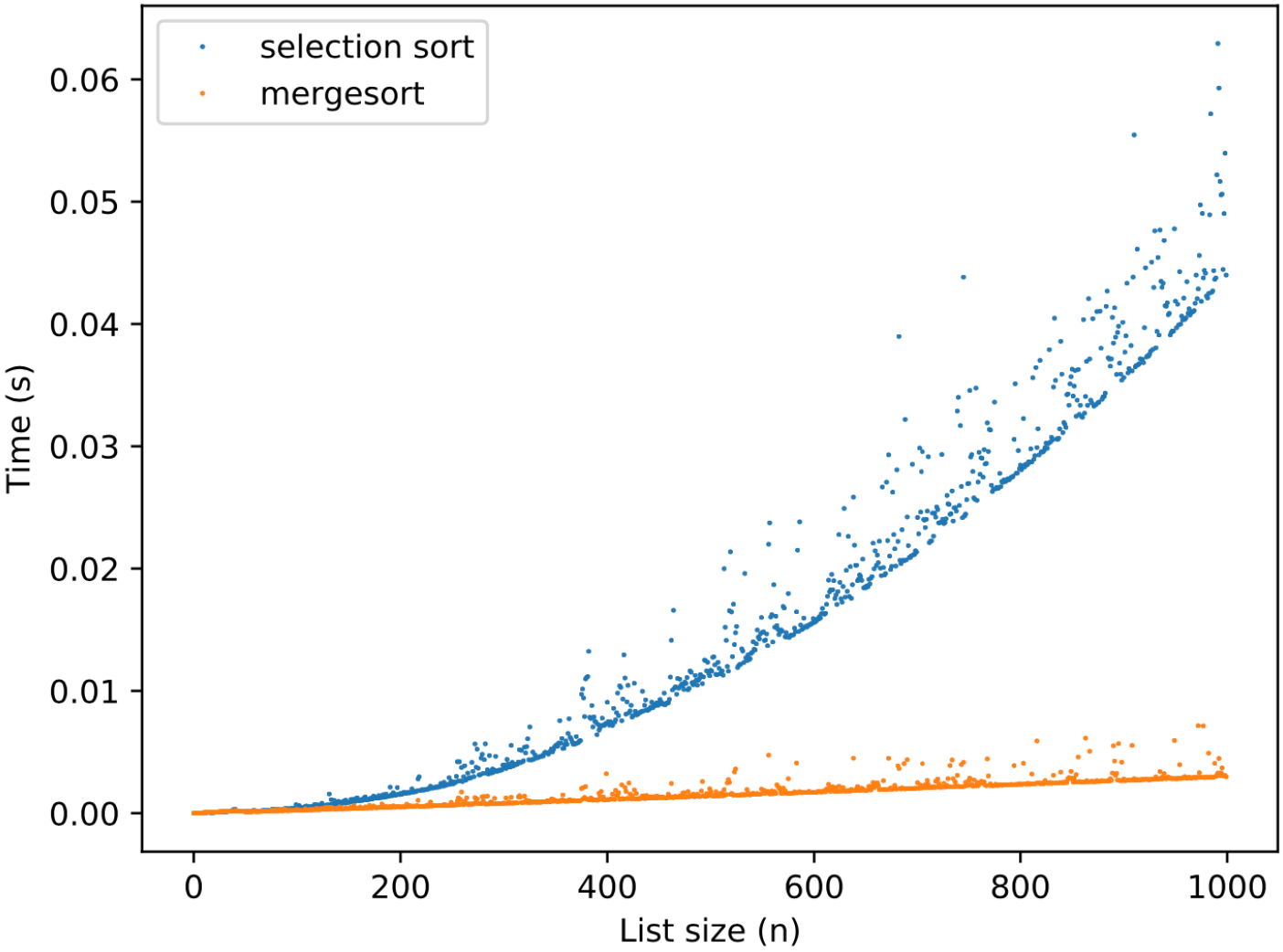
Simplification #2:

When n is large, T(n) grows approximately like n^2

We will write T(n) is $O(n^2)$

"T(n) is big O of n squared"

Asymptotic behavior



Asymptotic notation

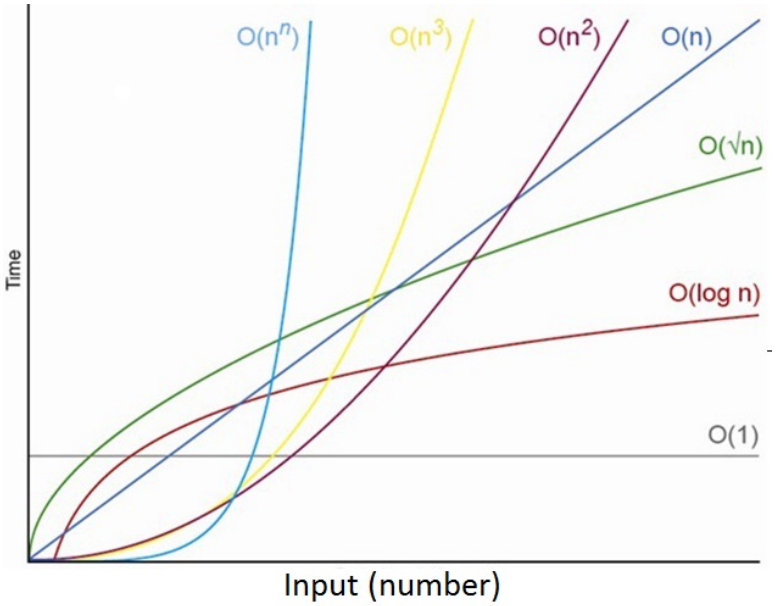
TOWARDS A FORMAL DEFINITION

Towards a formal definition of big O

Let $\mathrm{T}(n)$ be a function that describes the time it takes for some algorithm on input size n.

We would like to express how $\mathrm{T}(n)$ grows with n, as n becomes large i.e. *asymptotic* behavior.

Unlike with limits, we want to say that $\mathrm{T}(n)$ grows like certain *simpler* functions such as $\sqrt{n}, \log_2 n\,, n, n^2, 2^n$ …



Think about this as categorizing algorithms by running time "classes"

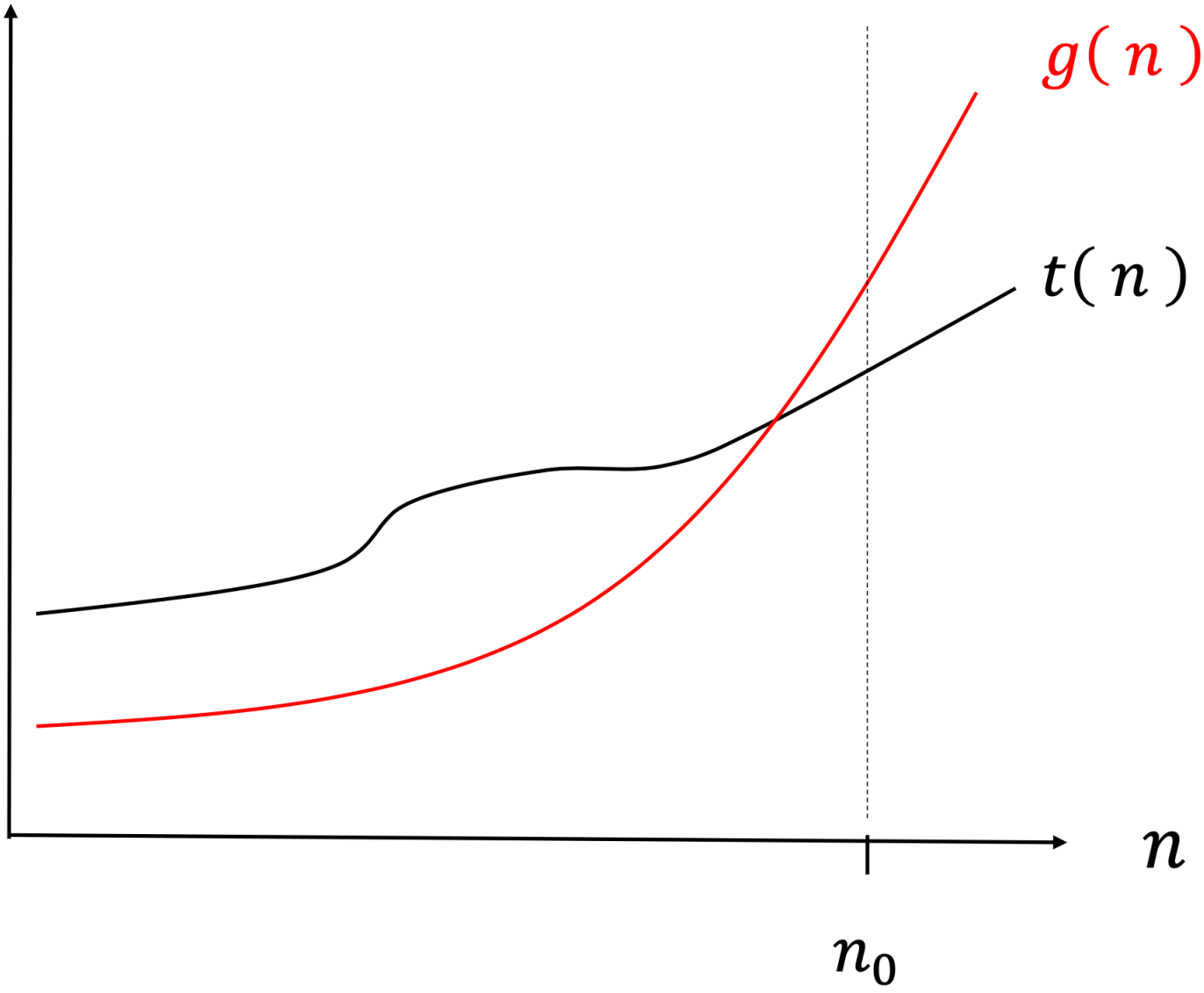
Preliminary Definition

Let $t(n)$ and $g(n)$ be two functions, where $n \geq 0$. We say $t(n)$ is *asymptotically bounded above* by $g(n)$ if there exists n_0 such that, for all $n \geq n_0$,

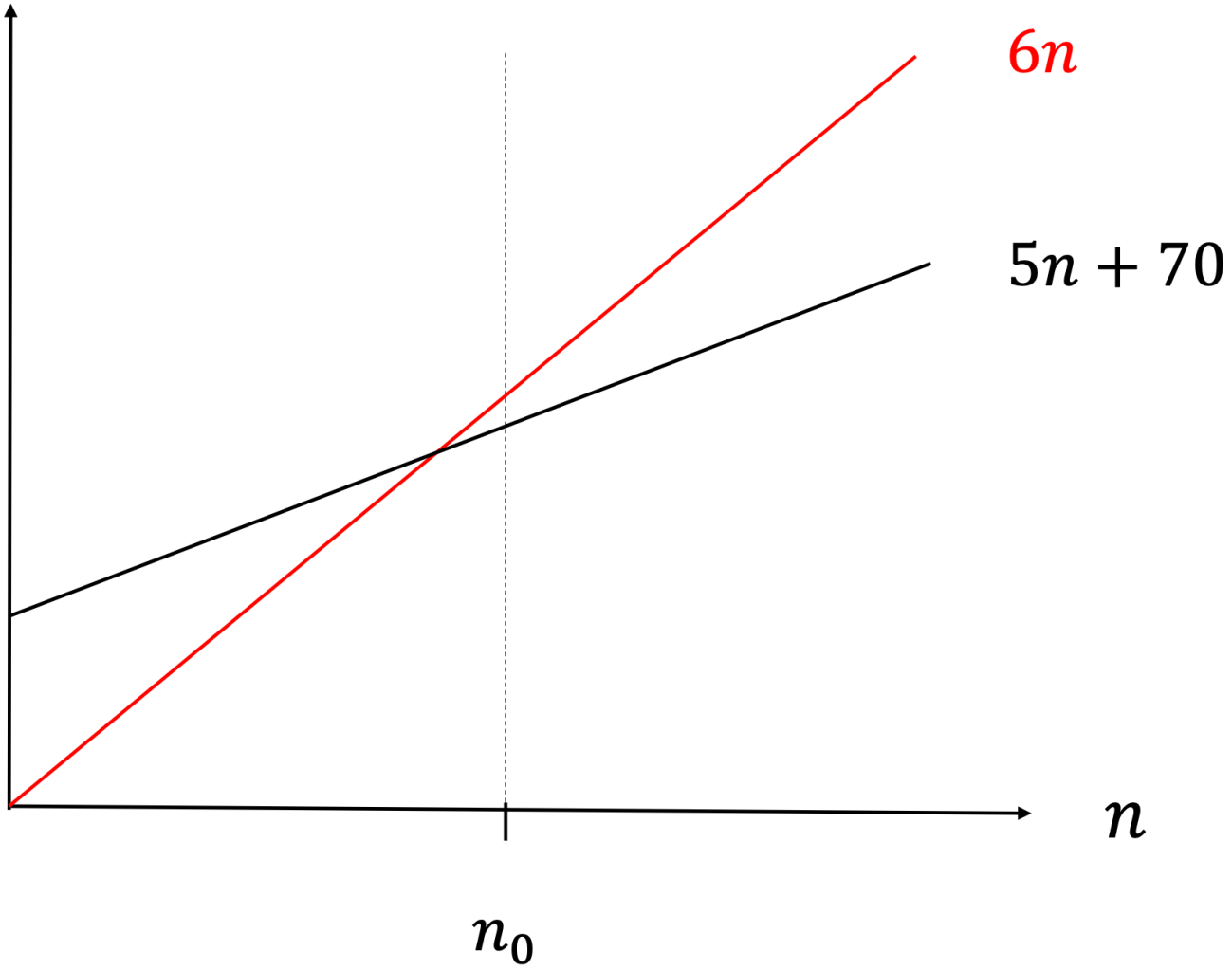
$$t(n) \leq g(n)$$

WARNING: This is not yet a formal definition!

$$for\ all\ n \geq n_0, t(n) \leq g(n)$$



Example



Example

Claim: $5n + 70$ is asymptotically bounded above by $6n$.

Proof:

(State definition) We want to show there exists an n_0 such that, for all $n \geq n_0, 5 \cdot n + 70 \leq 6 \cdot n$.

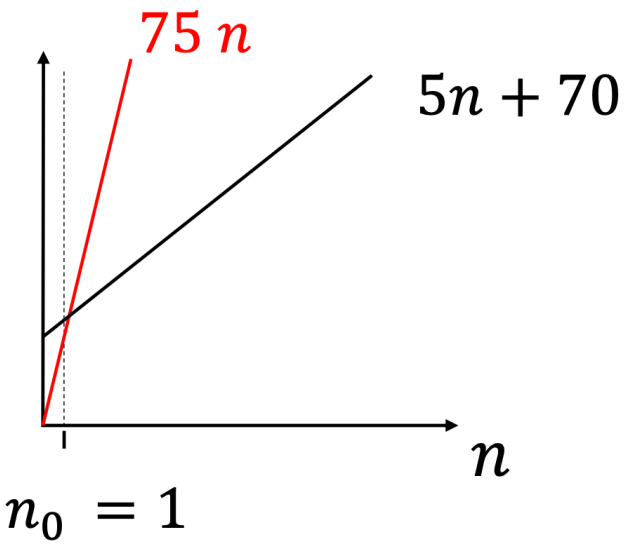
$$5n + 70 \leq 6n$$
$$\Leftrightarrow 70 \leq n$$

Thus, we can use $n_0 = 70$

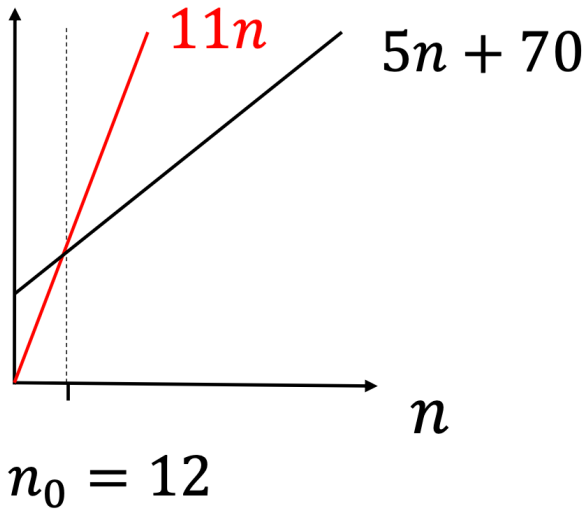
Symbol "$\Leftrightarrow$ " means "if and only if" i.e. logical equivalence

Choosing a function and constants

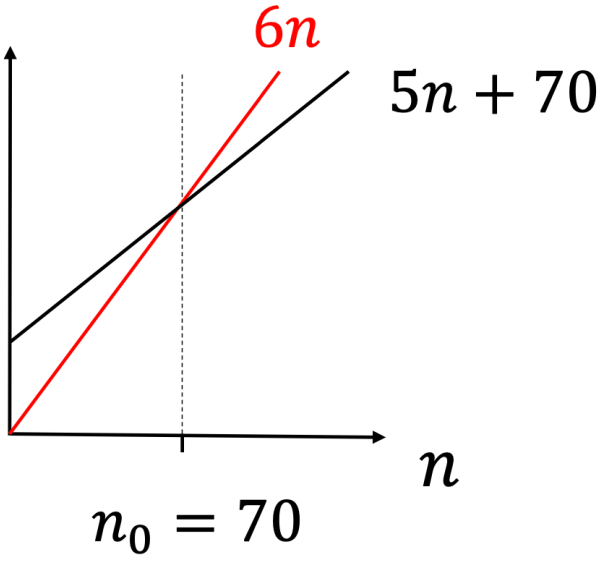
(A)



(B)



(C)



Which of these should we use as our upper bound?

Motivation

We would like to express formally how some function $t(n)$ grows with n, as n becomes large.

We would like to compare the function $t(n)$ with *simpler* functions , $g(n)$, such as $\sqrt{n}, \log_2 n\,, n, n^2, 2^n$ …

Formal Definition

Let $t\ n$ and $g\ n$ be two functions, where $n \geq 0$.

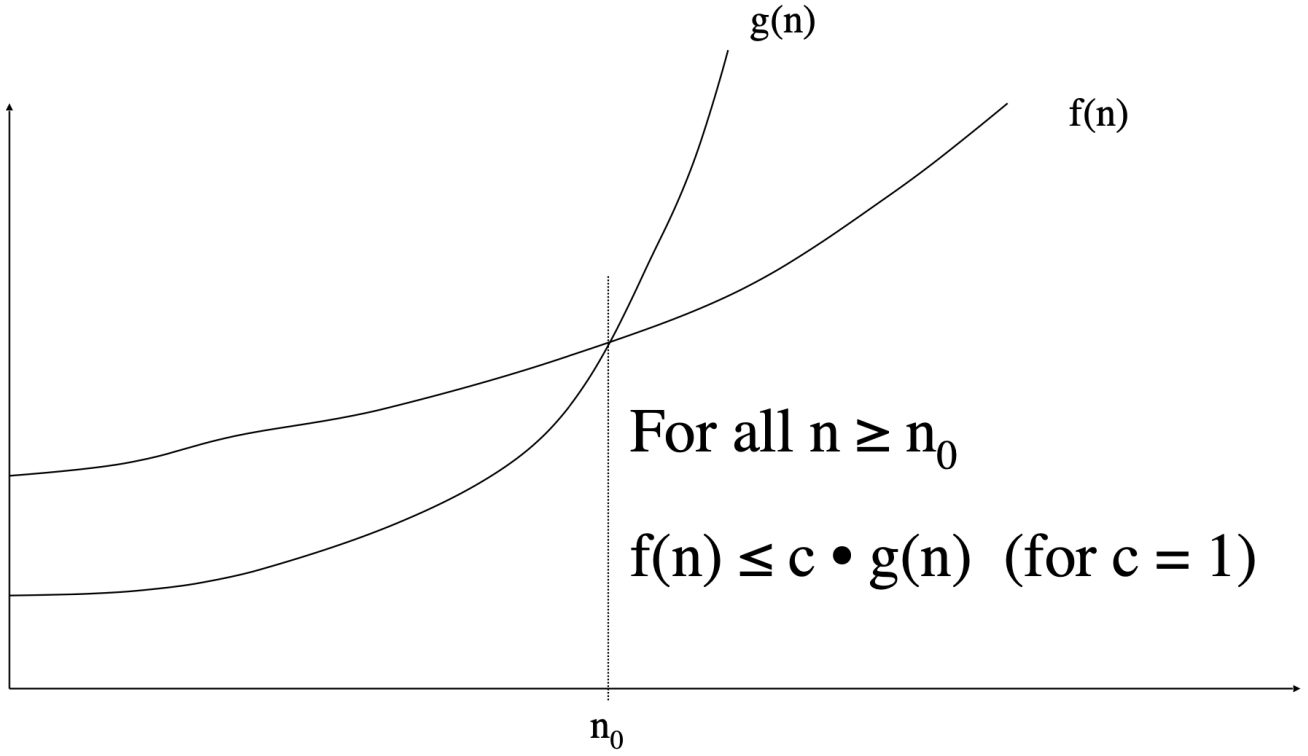
We say $t(n)$ is $O(g(n))$ if there exists two positive constants n_0 and c such that, for all $n \geq n_0$,

$$t(n) \leq c \cdot g(n)$$

Note: $g(n)$ will be a simple function, but this is not required in the definition.

Intuition

"$f(n)\ is\ O(g(n))$" if and only if there exists a point n_0 beyond which $f(n)$ is less than some fixed constant times $g(n)$.

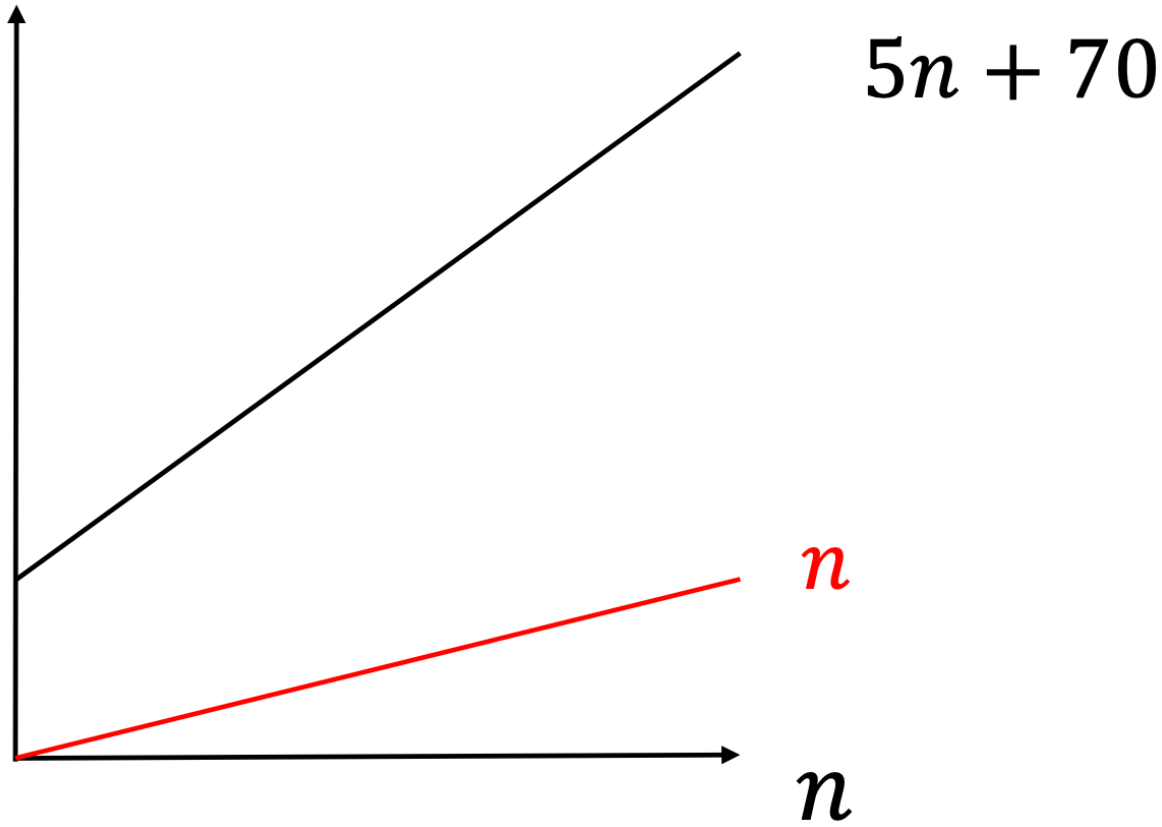


Examples

PRACTICING THE BIG-OH NOTATION

Example (1)

Claim: $5 \cdot n + 70 \; is \; O(n)$



Proof(s)

Claim: $5 \cdot n + 70 \; is \; O(n)$

Hint: substitute in something easy to deal with

Proof 1: $5 \cdot n + 70 \; \leq 5 \cdot n + 70 \cdot n = 75 \cdot n, \; if \; n \geq 1$

Thus, take $c = 75$ and $n_0 = 1$.

Proof 2: $5 \cdot n + 70 \; \leq 5 \cdot n + 6 \cdot n = 11 \cdot n, \; if \; n \geq 12$

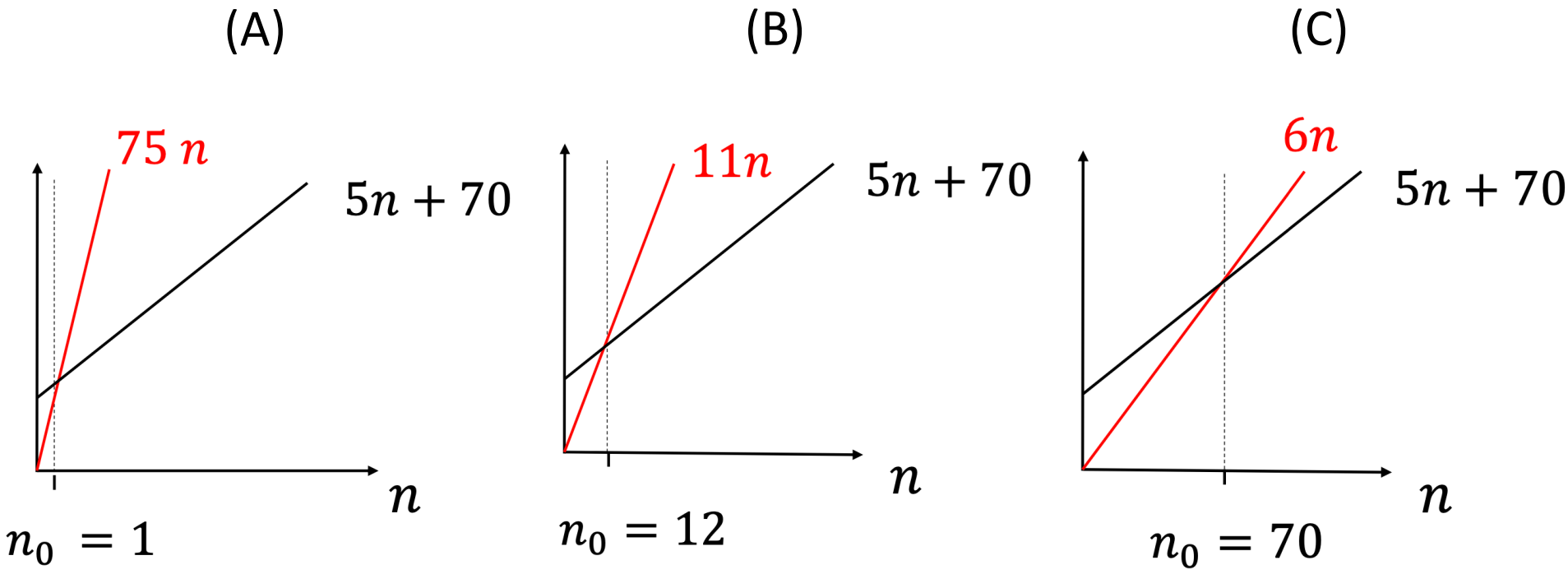
Thus, take $c = 11$ and $n_0 = 12$.

Proof 3: $5 \cdot n + 70 \; \leq 5 \cdot n + n = 6 \cdot n, \; if \; n \geq 70$

Thus, take $c = 6$ and $n_0 = 70$.

All these proofs are correct and show that $5 \cdot n + 70$ is $O(n)$

Visualization



All three upper bounds can be used to prove 5n + 70 is O(n)

Example (2)

Claim: $8 \cdot n^2 - 17 \cdot n + 46$ is $O(n^2)$.

Proof 1: $8n^2 - 17n + 46 \leq 8n^2 + 46n^2$, if $n \geq 1$

$\leq 54n^2$

Thus, we can take $c = 54$ and $n_0 = 1$.

Proof 2: $8n^2 - 17n + 46 \leq 8n^2$, if $n \geq 3$

Thus, we can take $c = 8$ and $n_0 = 3$.

Tips & rules

DEVELOPING YOUR INTUITION

What does O(1) mean?

We say $t(n)$ is $O(1)$, if there exist two positive constants n_0 and c such that, for all $n \geq n_0$.

$$t(n) \leq c$$

So, it means that $\boldsymbol{t(n)}$ **is bounded**.

Tips

Never write $O(3n), O(5\log_2 n), etc.$

Instead, write $O(n), O(\log_2 n), etc.$

Why? The point of the big O notation is to avoid dealing with constant factors. It's technically correct but we don't do it, since that would defeat the point of using big O notation

If you want to describe an exact upper bound, you can simply say a function is bounded above by another.

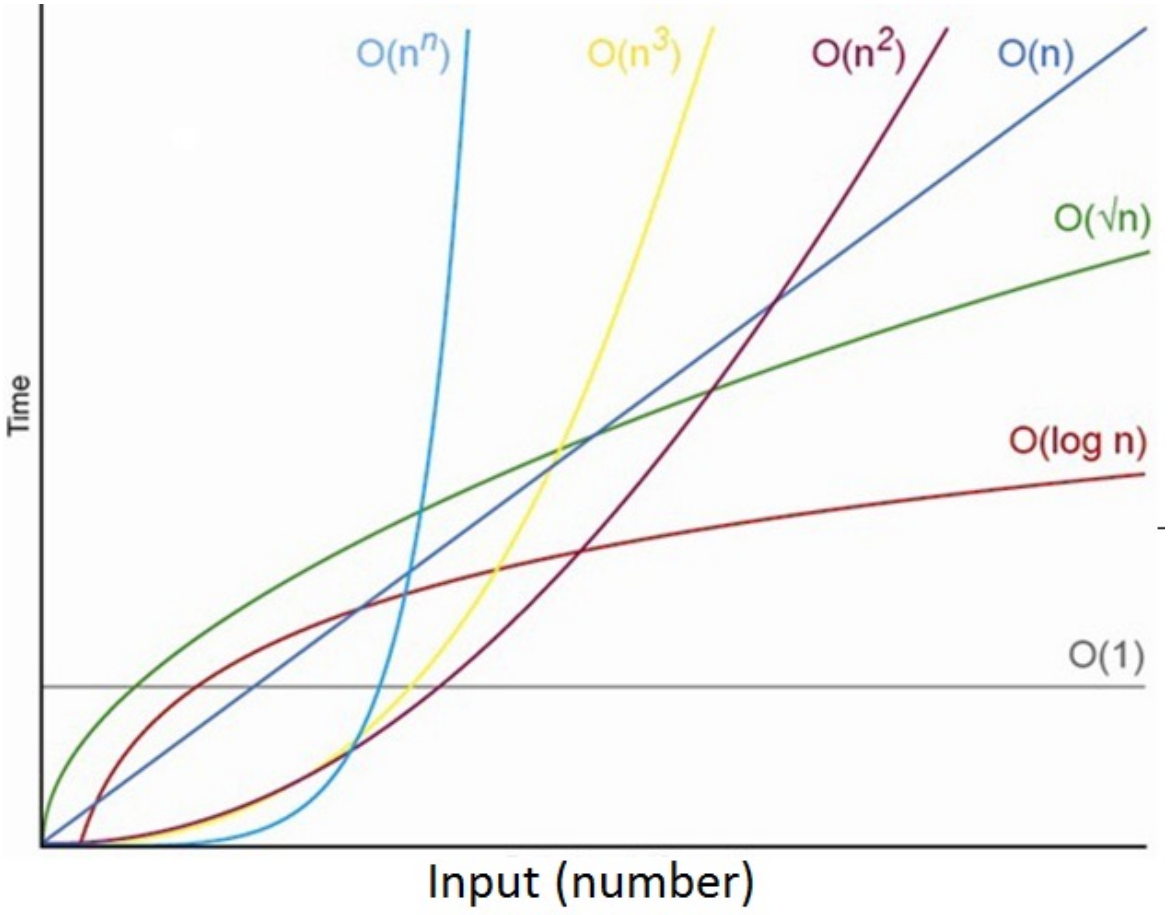
Other considerations

- n_0 and c are not *uniquely* defined. For a given n_0 and c that satisfies $O()$, we can increase one or both to again satisfy the definition. **There is not "better" choice of constants.**

- **However,** we generally want a "tight" upper bound (asymptotically), so functions in the big O gives us more information (Note: This is not the same as smaller n_0 or c). For instance, $f(n)$ that is $O(n)$ is also $O(n^2)$ and $O(2^n)$. But $O(n)$ is more informative.

If you think of big O complexities as classes, this makes sense!

Growth of functions



(from stackoverflow)

Tip: It is helpful to memorize the relationship between basic functions.

Practical meaning of big O…

	constant	*logarithmic*	*linear*	*N-log-N*	*quadratic*	*cubic*	*exponential*
n	**O(1)**	**O(log *n*)**	**O(*n*)**	**O(*n* log *n*)**	**O(n^2)**	**O(n^3)**	**O(2^n)**
1	1	1	1	1	1	1	2
2	1	1	2	2	4	8	4
4	1	2	4	8	16	64	16
8	1	3	8	24	64	512	256
16	1	4	16	64	256	4,096	65536
32	1	5	32	160	1,024	32,768	4,294,967,296
64	1	6	64	384	4,069	262,144	1.84×10^{19}

If the unit is in seconds, this would make ~10^{11} years…

Constant Factor rule

Suppose $\mathrm{f(n)}$ is $\mathrm{O(g(n))}$ and a is a positive constant.
Then, $a \cdot f(n)$ is also $O(g(n))$

Proof: By definition, if $\mathrm{f(n)}$ is $\mathrm{O(g(n))}$ then there exists two positive constants n_0 and c such that for all $n \geq n_0$,

$$f(n) \leq c \cdot g(n)$$

Thus, $$\mathrm{a} \cdot f(n) \leq \mathrm{a} \cdot c \cdot g(n)$$

We use the constant $a \cdot c$ to show that $a \cdot f(n)$ is $O(g(n))$.

Multiplying a function by a constant does not change its Big O upper bound since these upper bounds ignore constants

Sum rule

Suppose $f_1(n)$ *is* $O(g(n))$ and $f_2(n)$ *is* $O(g(n))$.

Then, $f_1(n) + f_2(n)$ *is* $O(g(n))$.

Proof: Let n_1, c_1 and n_2, c_2 be constants such that

$$f_1(n) \leq c_1 g(n), \text{ for all } n \geq n_1$$

$$f_2(n) \leq c_2 g(n), \text{ for all } n \geq n_2$$

So, $f_1(n) + f_2(n) \leq (c_1 + c_2) g(n), \text{ for all } n \geq \max(n_1, n_2)$.

We can use the constants $c_1 + c_2$ and $\max(n_1, n_2)$ to satisfy the definition.

A sum of two functions is inferior to a sum of two greater functions

Generalized Sum rule

Suppose $f_1(n)\ is\ O(g(n))$ and $f_2(n)\ is\ O(g(n))$.

Then, $f_1(n) + f_2(n) is\ O(g_1(n) + g_2(n))$.

Proof: Exercise…

Simply generalized the idea from the previous proof.
Share your answer on Ed!

Product Rule

Suppose $f_1(n)$ *is* $O(g(n))$ and $f_2(n)$ *is* $O(g(n))$.

Then, $f_1(n) \cdot f_2(n)$ *is* $O(g_1(n) \cdot g_2(n))$.

Proof: Let n_1, c_1 and n_2, c_2 be constants such that

$$f_1(n) \leq c_1 g_1(n), \text{ for all } n \geq n_1$$

$$f_2(n) \leq c_2 g_2(n), \text{ for all } n \geq n_2$$

So, $f_1(n) \cdot f_2(n) \leq (c_1 \cdot c_2) \cdot (g_1(n) \cdot g_2(n))$, for all $n \geq \max(n_1, n_2)$.

We can use the constants $c_1 \cdot c_2$ and $\max(n_1, n_2)$ to satisfy the definition.

A product of two functions is less than a product of two greater functions

Transitivity Rule

Suppose $f(n)\ is\ O(g(n))$ and $g(n)\ is\ O(h(n))$.

Then, $f(n) is\ O(h(n))$.

Proof: Let n_1, c_1 and n_2, c_2 be constants such that

$$f(n) \leq c_1 g(n), \text{ for all } n \geq n_1$$

$$g(n) \leq c_2 h(n), \text{ for all } n \geq n_2$$

So, $f(n) \leq (c_1 \cdot c_2) h(n), \text{ for all } n \geq \max(n_1, n_2)$.

We can use the constants $c_1 \cdot c_2$ and $\max(n_1, n_2)$ to satisfy the definition.

If a function A is greater than function B, and function B is greater than function C, then function A is greater than function C

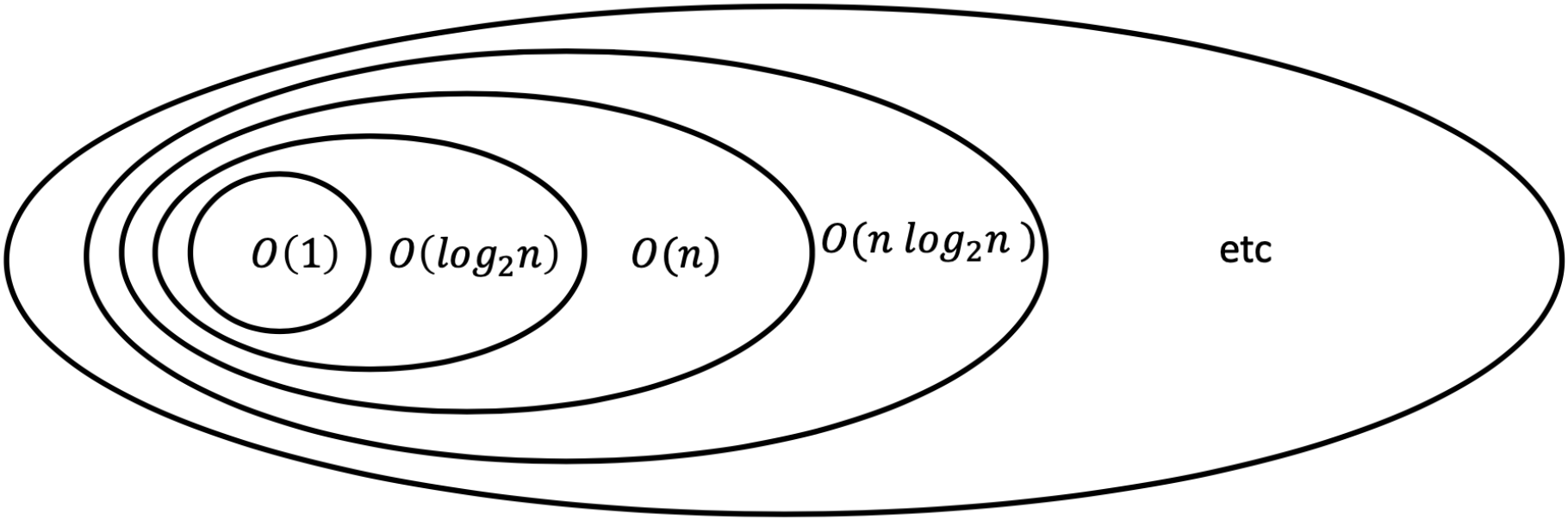
Notations

Element of/belongs to

If $f(n)$ is $O\big(g(n)\big)$, we often write $f(n) \in O(g(n))$. That is a member of the functions that are $O\big(g(n)\big)$.

For n sufficiently large we have, $1 < \log_2 n < n < n \log_2 n$ …
And we write $O(1) \subset O(\log_2 n) \subset O(n) \subset O(n \log_2 n)$ …

Strict subset of



(Big) omega and Theta

BEYOND THE BIG OH NOTATION

The Big Omega notation (Ω)

Let $t(n)$ and $g(n)$ be two functions with $n \geq 0$.

We say $t(n)$ is $\Omega(g(n))$, if there exists two positive constants n_0 and c such that, for all $n \geq n_0$,

$$t(n) \geq c \cdot g(n)$$

Note: This is the opposite of the big O notation. The function g is now used as a "*lower* bound".

Example

Claim: $\frac{n(n-1)}{2}$ is $\Omega(n^2)$.

Proof: We show first that $\frac{n(n-1)}{2} \geq \frac{n^2}{4}$.

$$\Leftrightarrow 2n(n-1) \geq n^2$$

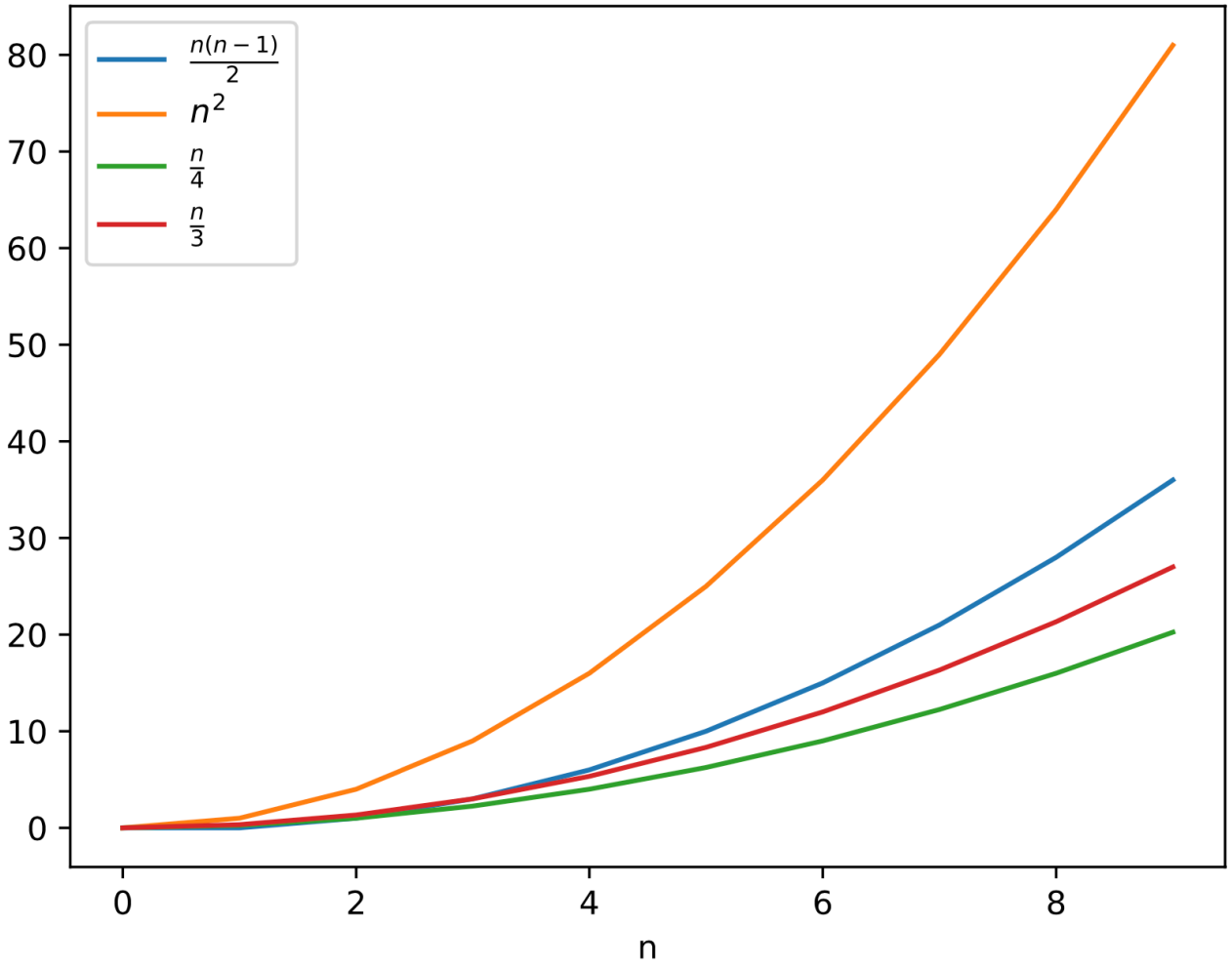
$$\Leftrightarrow n^2 \geq 2n$$

$$\Leftrightarrow n \geq 2$$

Thus, we take $c = \frac{1}{4}$ and $n_0 = 2$.

Exercise: Prove that it also works with $c = \frac{1}{3}$ and $n_0 = 3$

Intuition

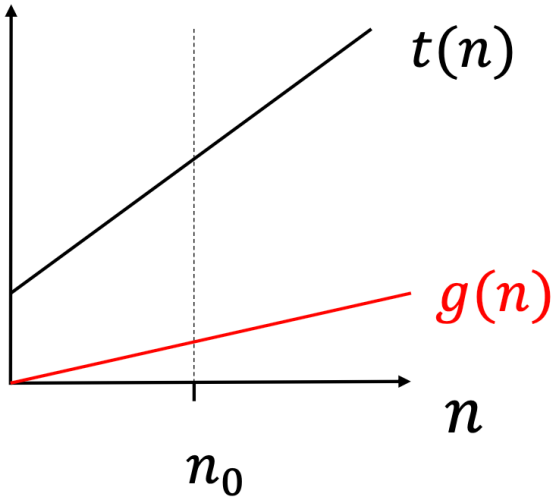


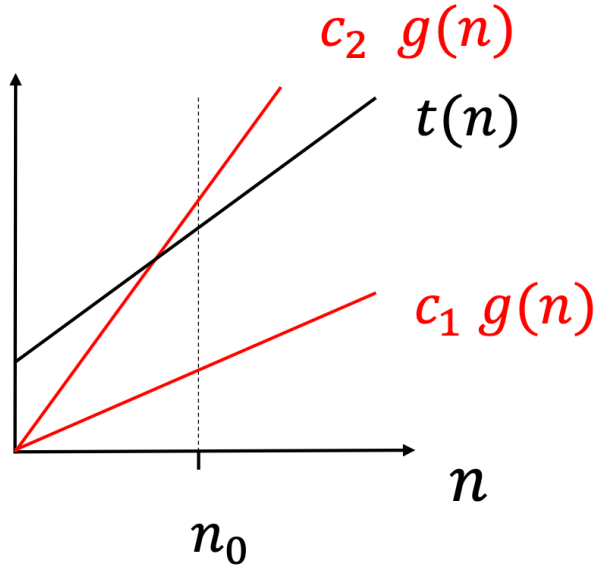
And now… big Theta!

Let $t(n)$ and $g(n)$ be two functions, where $n \geq 0$.

We say $t(n)$ is $\Theta(g(n))$ if there exists three positive constants n_0 and c_1, c_2 such that, for all $n \geq n_0$,

$$c_1 \cdot g(n) \leq t(n) \leq c_2 \cdot g(n)$$





Note: if $t(n)$ is $\Theta(g(n))$. Then, it is also $O(g(n))$ and $\Omega(g(n))$.

Example

Let $t(n) = 4 + 17\log_2 n + 3n + 9n\log_2 n + \frac{n(n-1)}{2}$

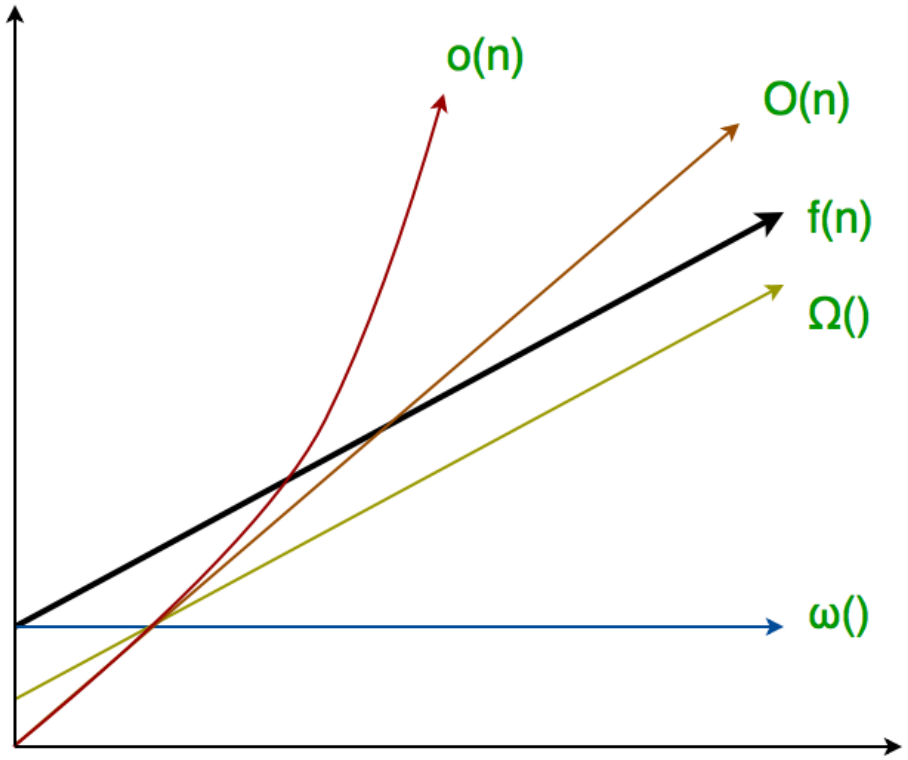
Claim: $t(n)$ is $\Theta(n^2)$

Proof:

$$\frac{n^2}{4} \leq t(n) \leq (4 + 17 + 3 + 9 + \frac{1}{2}) \cdot n^2$$

Big vs. little

The big O (resp. big Ω) denotes a tight upper (resp. lower) bounds, while the little o (resp. little ω) denotes a loose upper (resp. lower) bounds.



(from geekforgeek.org)

Algorithm analysis

RUNNING TIME OF RECURSIVE ALGORITHMS

Algorithm analysis

How to estimate the running time of a recursive algorithm?

1. Define a function T(*n*) representing the time spent by your algorithm to execute an entry of size *n*
2. Write a recursive formula computing T(*n*)
3. Solve the recurrence

Notes:

- *n* can be anything that characterizes accurately the size of the input (e.g., size of the array, number of bits)
- We count the number of elementary operations (e.g., addition, shift) to estimate the running time.
- We usually compute an upper bound rather than exact count.
- We will introduce later a general method to solve this.

Examples (binary search)

```
int bsearch(int[] A, int i, int j, int x) {
      if (i<=j) { // the region to search is non-empty
            int e = ⌊(i+j)/2⌋;
            if (A[e] > x) { return bsearch(A,i,e-1,x);
            } elif (A[e] < x) { return bsearch(A,e+1,j,x);
            } else { return e; }
      } else { return -1; } // value not found
}
```

$$T(n) = \begin{cases} c & if\ n = 1 \\ T\left(\frac{n}{2}\right) + c' & if\ n > 1 \end{cases}$$

Notes:

- n is the size of the array
- Formally, we should use $\leq$ rather than $=$

Example (naïve Fibonacci)

```
public static int Fib(int n) {
      if (n <= 1) { return n; }
      return Fib(n-1) + Fib(n-2);
}
```

$$T(n) = \begin{cases} c & if\ n \leq 1 \\ T(n-1) + T(n-2) + c' & if\ n > 1 \end{cases}$$

What are the value of c and c' ?

- If $n \leq 1$ there is only one comparison thus *c=1*
- If $n > 1$ there is one comparison and one addition thus *c'=2*

Notes:

- we neglect other constants
- We can approximate *c* and *c'* with an *asymptotic* notation O(1)

Example (Merge sort)

```
MergeSort (A, p, r)
if (p < r) then
      q ← ⌊(p+r)/2⌋
      MergeSort (A, p, q)
      MergeSort (A, q+1, r)
      Merge (A, p, q, r)
```

- Base case: constant time c
- Divide: computing the middle takes constant time c'
- Conquer: solving 2 subproblems takes $2 \cdot T(n/2)$
- Combine: merging n elements takes $k \cdot n$

$$T(n) = \begin{cases} c & if\ n = 1 \\ 2 \cdot T\left(\frac{n}{2}\right) + k \cdot n + c + c' & if\ n > 1 \end{cases}$$

Substitution method

How to solve a recursive equation?

1. Guess the solution.
2. Use induction to find the constants and show that the solution works.

Example:
$$T(n) = \begin{cases} 1 & if\ n = 0 \\ 2 \cdot T(n-1) & if\ n > 0 \end{cases}$$

Guess: $T(n) = 2^n$ (remember the Fibonacci recursive tree)
Base case: $T(0) = 2^0 = 1$ ✓
Inductive case:
Assume $T(n) = 2^n$ until rank *n-1, then show it is true at rank n.*
$T(n) = 2 \cdot T(n-1) = 2 \cdot 2^{n-1} = 2^n$ ✓

Running time of binary search

$$T(n) = \begin{cases} 0 & if\ n = 1 \\ T\left(\frac{n}{2}\right) + 1 & if\ n > 1 \end{cases}$$

Note: set the constant c=0 and c'=1

Guess: $T(n) = \log_2 n$

Base case: $T(1) = \log_2 1 = 0$ ✓

Inductive case:

Assume $T(n/2) = \log_2(n/2)$

$T(n) = T(n/2) + 1 = \log_2(n/2) + 1$

$= \log_2(n) - \log_2 2 + 1 = \log_2 n$ ✓

Induction hypothesis can be anything < n

We simplify the formula and say that $T(n)$ is $O(\log n)$

Running time of Merge Sort

This is not as easy to guess. Let's try plotting it!

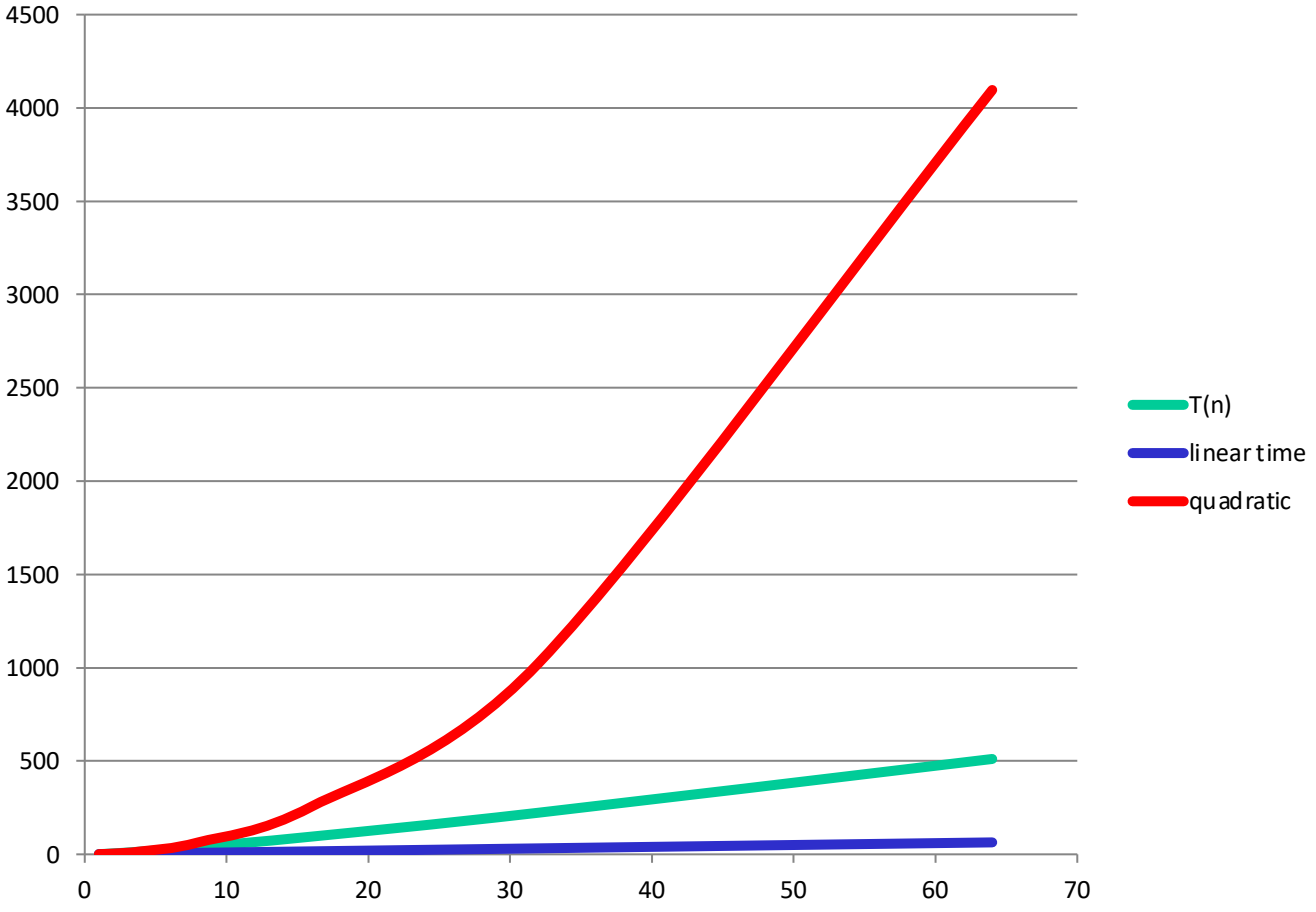
We use a simplified version:

$$T(n) = \begin{cases} 1 & if\ n = 1 \\ 2 \cdot T\left(\frac{n}{2}\right) + n & if\ n > 1 \end{cases}$$

Simulation:

n	1	2	4	8	16	32	64	…	n
T(n)	1	5	15	39	95	223	511	…	?

Running time of Merge Sort



Running time of Merge Sort

Remember the recursive case: $T(n) = \ 2 \cdot T\left(\frac{n}{2}\right) + n \quad if\ n > 1$

Guess: $T(n) = \mathrm{n} \cdot \log n + n$

Base case: $T(1) = 1 \cdot \log 1 + 1 = 1$ ✓

Inductive case:

Assume $T(n/2) = \frac{n}{2} \cdot \log\left(\frac{n}{2}\right) + \frac{n}{2}$

$T(n) = 2 \cdot T\left(\frac{n}{2}\right) + n = 2 \cdot \left(\frac{n}{2} \cdot \log\left(\frac{n}{2}\right) + \frac{n}{2}\right) + n$

$= n \cdot (\log n - \log 2) + n + n = n \cdot \log n - n + 2 \cdot n$

$= n \cdot \log n + n$ ✓

We simplify the formula and say that $T(n)$ is $O(n \cdot \log n)$