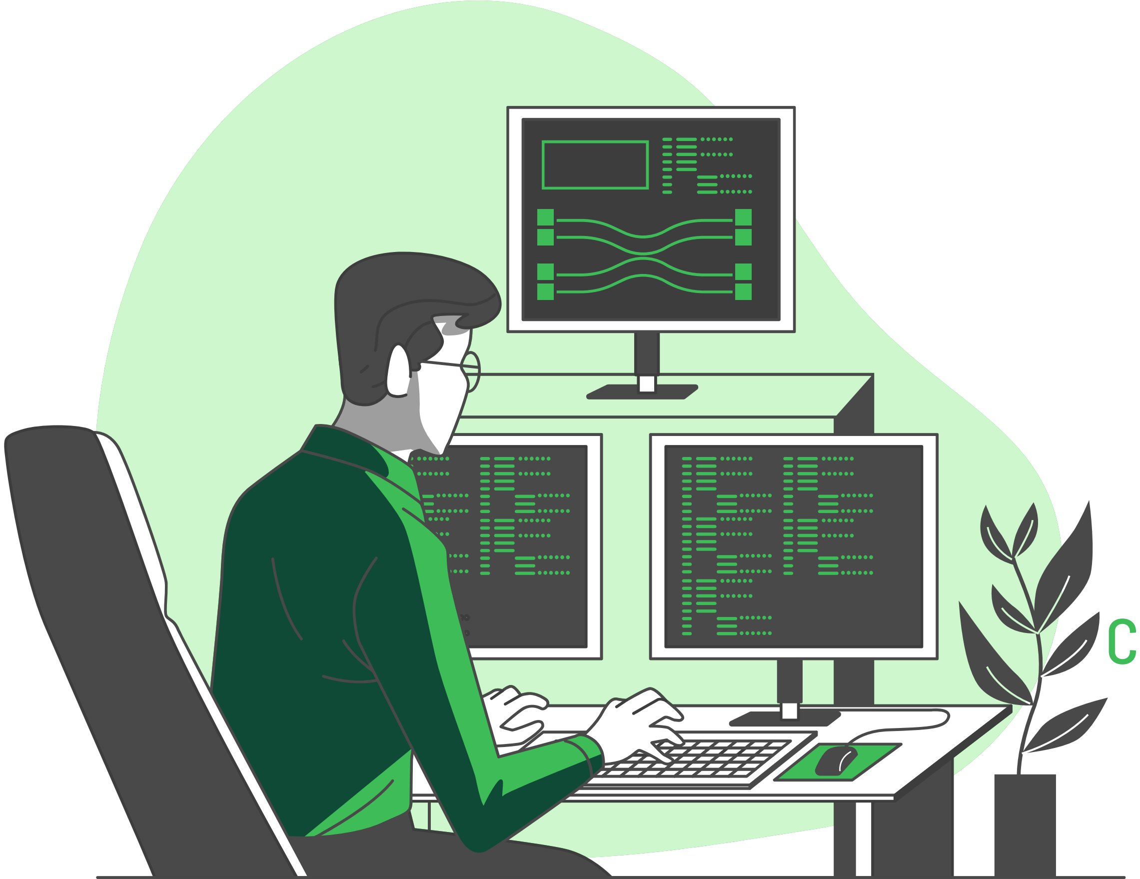
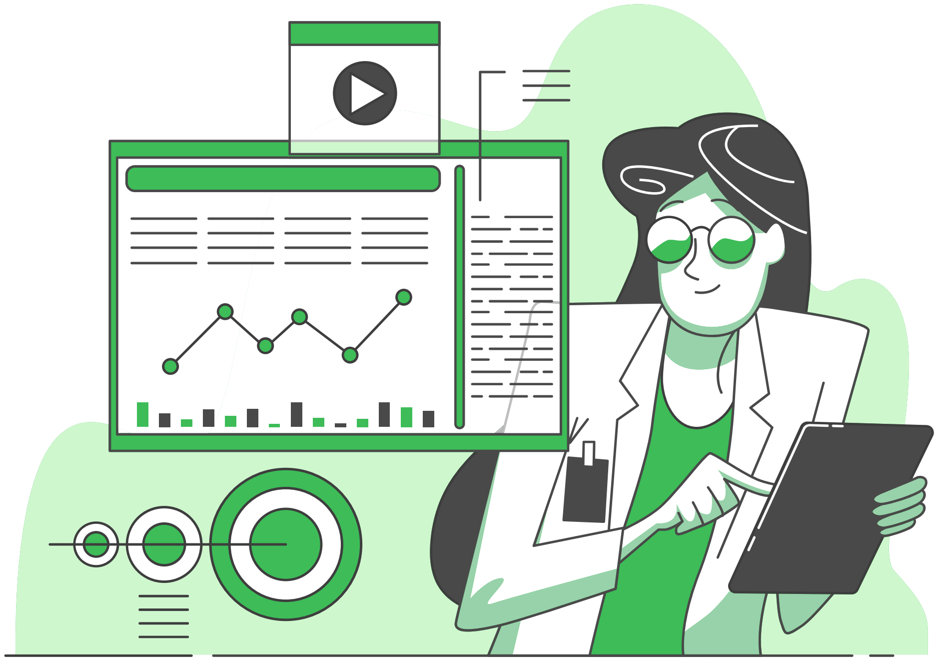


Asymptotic Runtime Analysis

CS 251 - Data Structures and Algorithms

Note:
Slides complement the discussion in class

Table of Contents

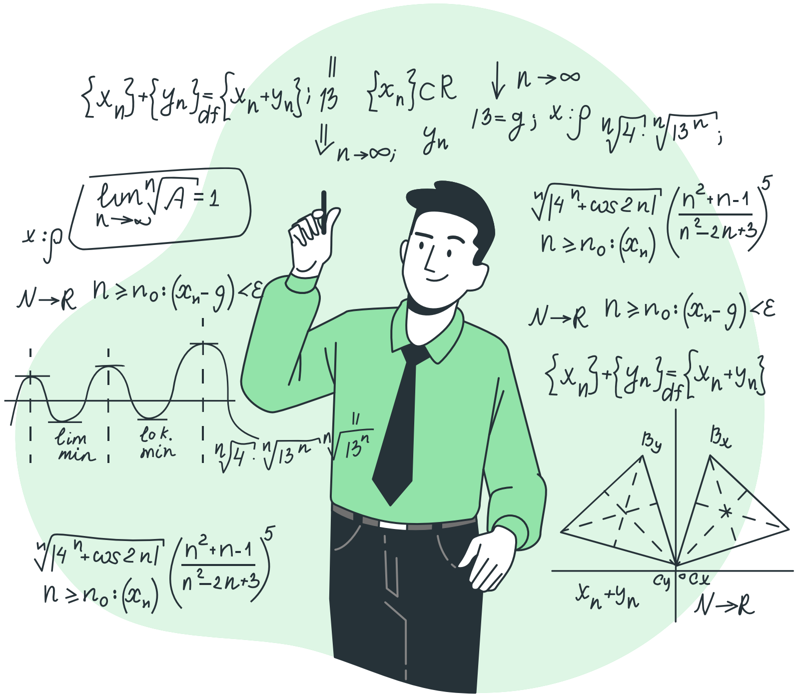


01

Computational Tractability

Let's dare to define algorithm efficiency

What Do We Know So Far?



We can run experimental analyses based on observed data.

We can define runtime expressions that count the number of steps run by an algorithm in terms of the input size or values.

Runtime expressions may have different terms and coefficients.

Idea

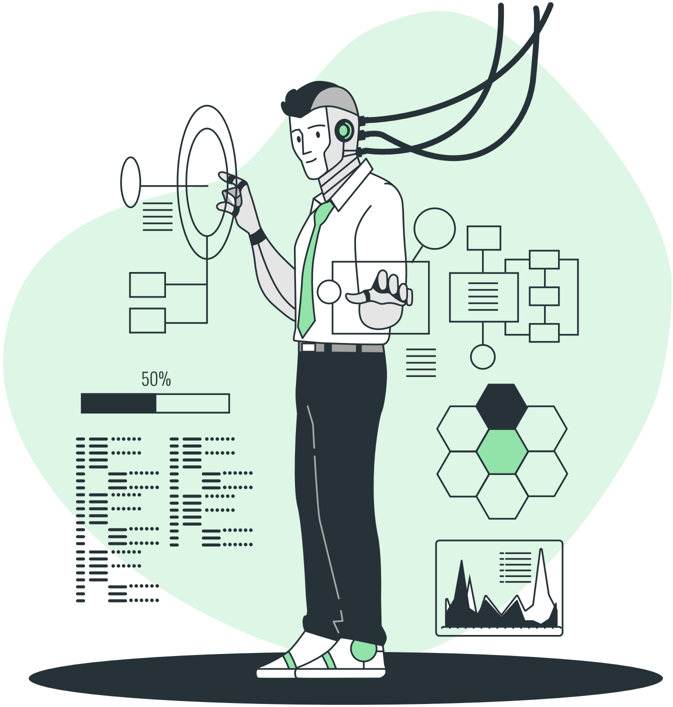
Tractability

We want to find efficient solutions for computational problems.

A **tractable** problem is *“a problem that can be solved using a reasonable amount of resources (i.e., time and space).”*

...

Algorithm Efficiency?



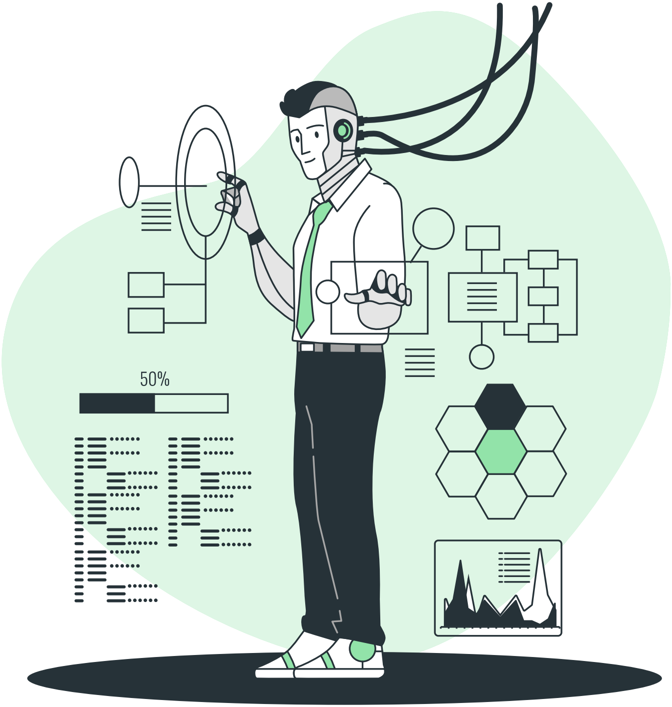
Proposed def. (1): *"An algorithm is efficient if, when implemented, it runs quickly on real input instances."*

Issues:

- *Where* do we run such an algorithm?
- *How well?* Does it run quickly for multiple input sizes?

Algorithm efficiency definition from: Algorithm Design, by Kleinberg and Tardos.

Algorithm Efficiency?



Proposed def. (2): *"An algorithm is efficient if it achieves qualitative better worst-case performance, at an analytical level, than brute-force search."*

Issues:

- Worst-case performance seems draconian. How about *average-case*?
- *Qualitative better* performance?

Algorithm efficiency definition from: Algorithm Design, by Kleinberg and Tardos.

Polynomial Time

Suppose an algorithm has the following property:

There are constants $a > 0$ and $b > 0$ so that on every input instance of size n, its running time is bounded by an^b primitive computational steps.

Let's consider an input size increase from n to $2n$. So, the runtime increases:

$$a(2n)^b = a2^b n^b$$

Since b is constant, so is 2^b (slow-down factor).

Conclusion: Lower degree polynomials exhibit better scaling behavior than higher degree polynomials.

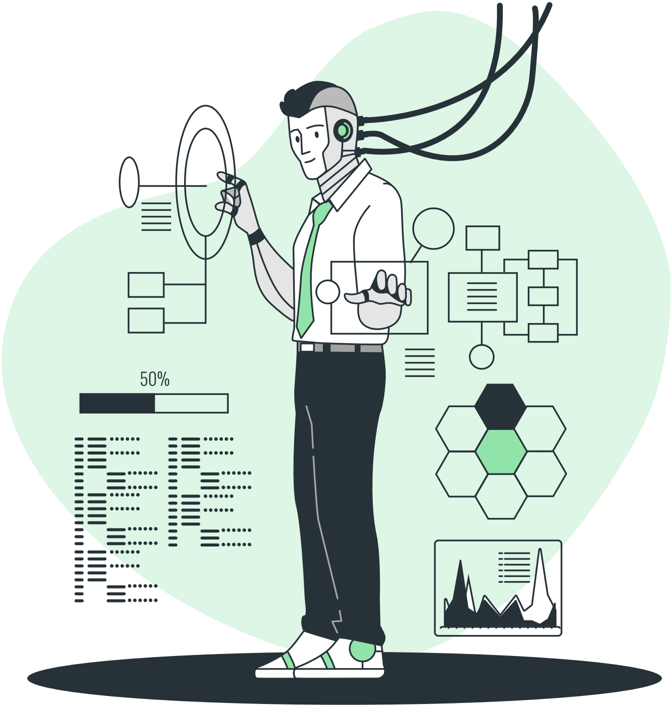
Explanation from: Algorithm Design, by Kleinberg and Tardos.

Operation Costs

Operation	Example	Nanoseconds
Integer add	a + b	2.1
Integer multiply	a * b	2.4
Integer divide	a / b	5.4
Floating-point add	a + b	4.6
Floating-point multiply	a * b	4.2
Floating-point divide	a / b	13.5
Sine	Math.sin(theta)	91.3
Arctangent	Math.atan2(y,x)	129.0

Running OS X on Macbook Pro 2.2Ghz with 2GB Ram

Algorithm Efficiency?



Proposed def. (3): *"An algorithm is efficient if it has a polynomial running time."*

Issues:

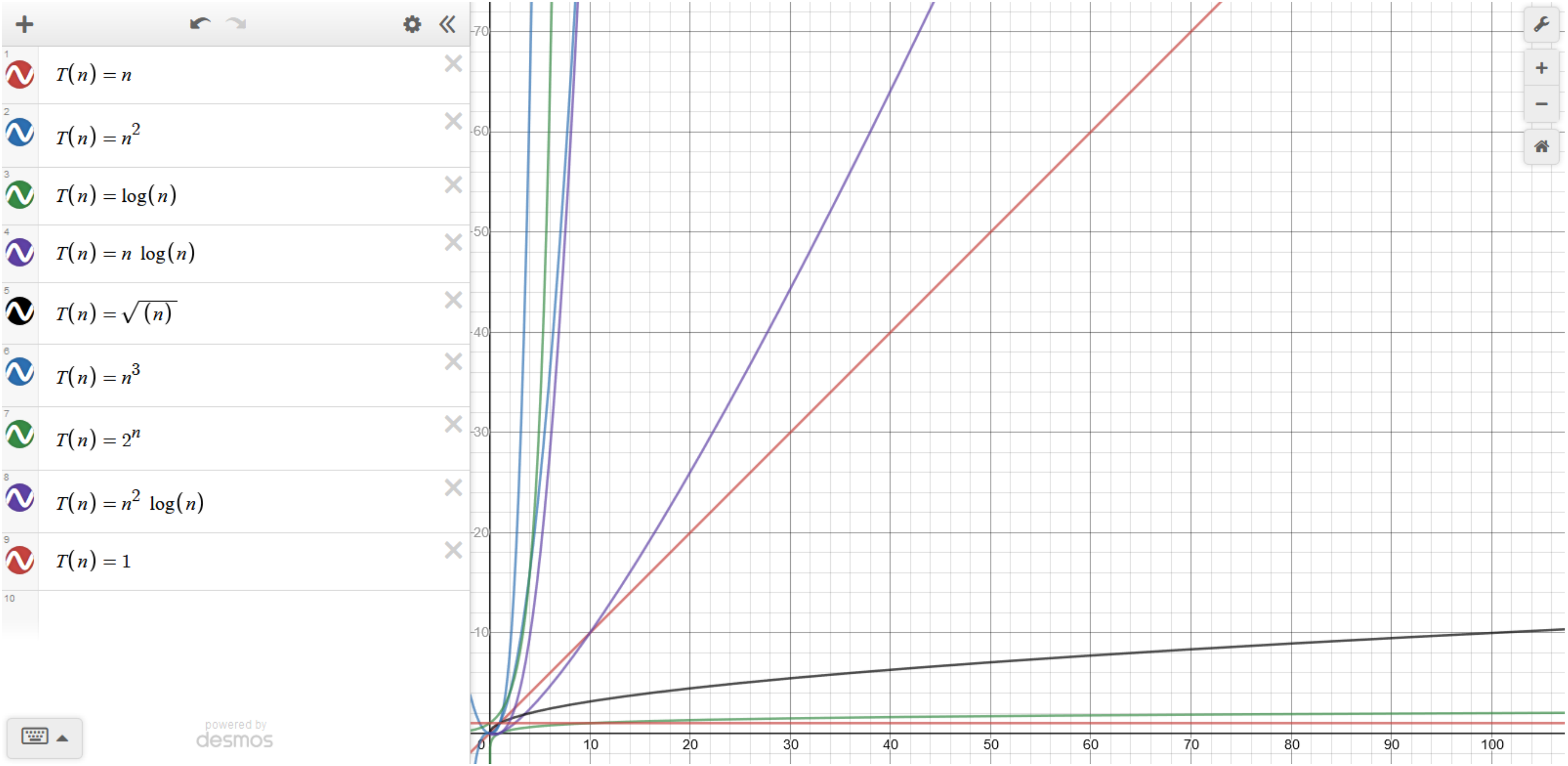
- Too prescriptive!
- We know "better" running times than polynomial time.

Algorithm efficiency definition from: Algorithm Design, by Kleinberg and Tardos.

	n	$n\log_2(n)$	n^2	n^3	1.5^n	2^n	$n!$
$n = 10$	< 1 sec	< 1 sec	< 1 sec	< 1 sec	< 1 sec	< 1 sec	4 sec
$n = 30$	< 1 sec	< 1 sec	< 1 sec	< 1 sec	< 1 sec	18 min	10^{25} years
$n = 50$	< 1 sec	< 1 sec	< 1 sec	< 1 sec	11 min	36 years	Very long
$n = 100$	< 1 sec	< 1 sec	< 1 sec	1 sec	12,892 years	10^{17} years	Very long
$n = 1{,}000$	< 1 sec	< 1 sec	1 sec	18 min	Very long	Very long	Very long
$n = 10{,}000$	< 1 sec	< 1 sec	2 min	12 days	Very long	Very long	Very long
$n = 100{,}000$	< 1 sec	2 sec	3 hours	32 years	Very long	Very long	Very long
$n = 1{,}000{,}000$	1 sec	20 sec	12 days	31,710 years	Very long	Very long	Very long

"The running times (rounded up) of different algorithms on inputs of increasing size, for a processor performing a million high-level instructions per second. In cases where the running time exceeds 10^{25} years, we simply record the algorithm as taking a very long time."

Table from: Algorithm Design, by Kleinberg and Tardos.



$T(n) = n$
$T(n) = n^2$
$T(n) = \log(n)$
$T(n) = n \log(n)$
$T(n) = \sqrt{(n)}$
$T(n) = n^3$
$T(n) = 2^n$
$T(n) = n^2 \log(n)$
$T(n) = 1$
powered by desmos

Grow Rate	Name	Example	Description
1	Constant	Assignment	Statement
$\log(n)$	Logarithmic	Binary Search	Non-linear increments
n	Linear	Linear Search	Loop
$n\log(n)$	Linearithmic	Merge Sort	Divide and Conquer
n^2	Quadratic	Check all pairs	Some double loops
n^3	Cubic	Check all triples	Some triple loops
2^n	Exponential	Check all subsets	Exhaustive search

What can we say about an algorithm as the input size increases?

Idea

Asymptotic

"Approaching a value or a curve (i.e., a runtime expression) arbitrarily closely"

Number of Operations?

```
algorithm ThreeSum(A:array)

   let n be the length of A
   count ← 0

   for i from 0 to n-1 do
      for j from i + 1 to n-1 do
         for k from j + 1 to n-1 do
            if A[i] + A[j] + A[k] = 0
               count ← count + 1
            end if
         end for
      end for
   end for

   return count
end algorithm
```

Let's focus on the **for loops** code for now:

3 variable declarations
8 integer additions
4 compares

Anything else? Are those numbers accurate?

How many times do those operations execute in terms of n?

Example: OneSum code snippet

```
int count = 0;

for (int i = 0; i < n; i++)
{
    if (A[i] == 0)
    {
        count++;
    }
}
```

Running this code for some positive integer value **n** will result in:

- Variable declarations: 2
- Assignment statements: 2
- $<$ compares: $n + 1$
- $==$ compares: n
- Array accesses: n
- Variable increments: n to $2n$

Example: TwoSum code snippet

```
int count = 0;

for (int i = 0; i < n; i++)
{
    for (int j = i + 1; j < n; j++)
    {
        if (A[i] + A[j] == 0)
        {
            count += 1;
        }
    }
}
```

Running this code for some positive integer value **n** will result in:

- Variable declarations: $n + 2$
- Assignment statements: $n + 2$
- $<$ compares: $n + 1 + \frac{1}{2}n(n+1)$
- $==$ compares: $\frac{1}{2}n(n-1)$
- Array accesses: $n(n-1)$
- Variable increments: $n + \frac{1}{2}n(n-1)$ to $n + n(n-1)$

So, We Have a T(n)...

- Runtime expressions may have **more than one** term.
- There are more "**dominant**" terms than others.
- Some terms appear "**frequently**".

- **Idea:** Let's consider how runtime expressions **grow** as the input size increases.

02

Big-O

Asymptotic upper bounds

Big-O Notation

Useful to classify algorithms according to an **upper bound**.

Definition: $f(n) \in O\big(g(n)\big)$ if $\exists c > 0, \exists n_0 > 0$ such that $0 \leq f(n) \leq cg(n), \forall n \geq n_0$.

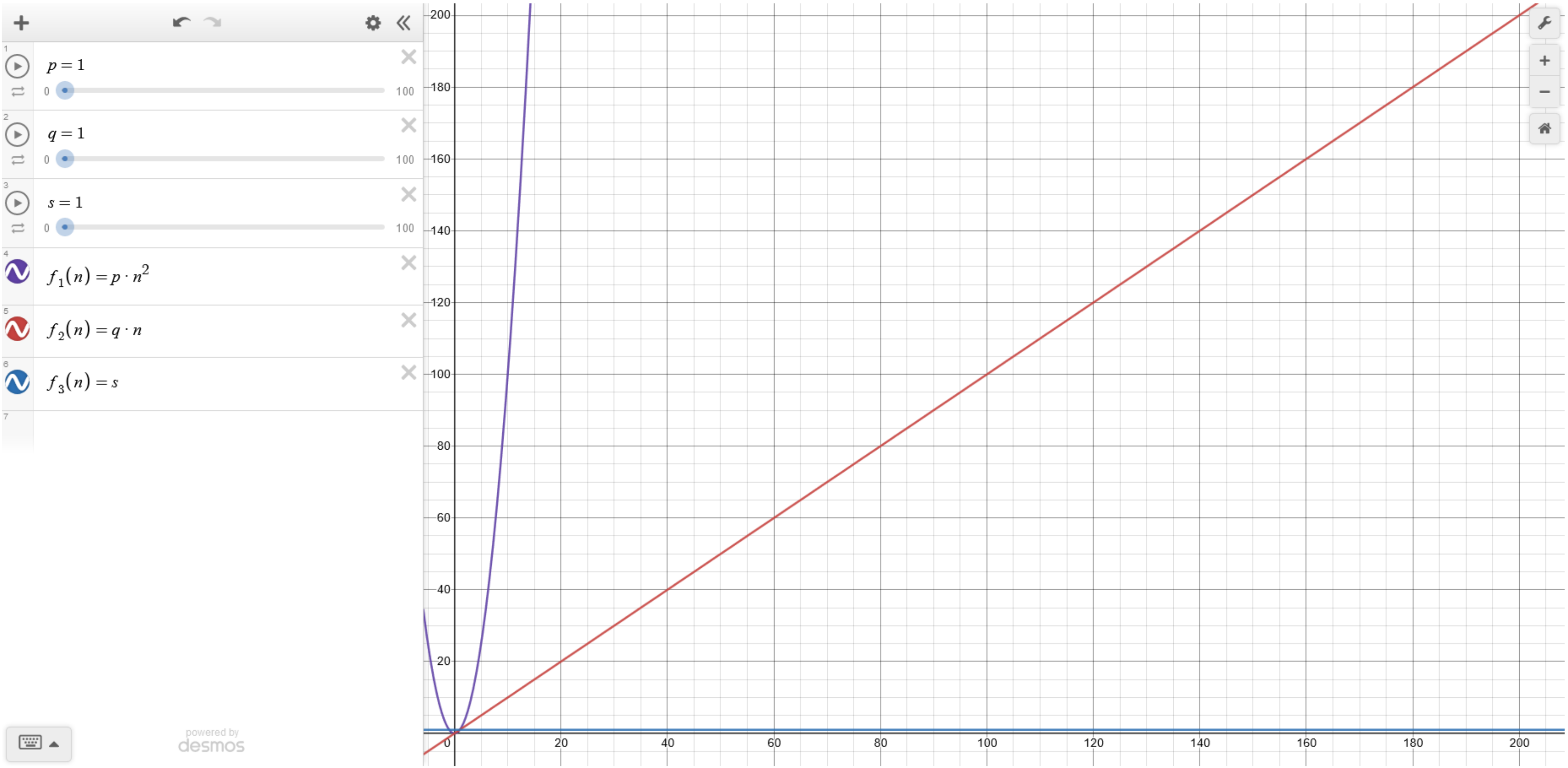
Plain English: If $f(n) \in O\big(g(n)\big)$, then $f(n)$ doesn't **grow** any **faster** than $g(n)$.

Example

Consider $T(n) = pn^2 + qn + s$, where $p, q, s \in \mathbb{R}^+$.

Claim: $T(n) \in O(n^2)$.

Where do we even start? Check which term from $T(n)$ grows the fastest.



$p = 1$
0
100
$q = 1$
0
100
$s = 1$
0
100
$f_1(n) = p \cdot n^2$
$f_2(n) = q \cdot n$
$f_3(n) = s$
powered by
desmos
200
180
160
140
120
100
80
60
40
20
0
20
40
60
80
100
120
140
160
180
200

Example

Consider $T(n) = pn^2 + qn + s$, where $p, q, s \in \mathbb{R}^+$.

Claim: $T(n) \in O(n^2)$.
Proof:

$$T(n) = pn^2 + qn + s \leq pn^2 + qn^2 + sn^2 = (p + q + s)n^2$$

for all $n \geq 1$. The obtained inequality is exactly the definition of $O(\cdot)$, which requires $T(n) \leq cn^2$.

So, $T(n) \in O(n^2)$, where $c = p + q + s$, $n_0 = 1$, and $g(n) = n^2$.

About Notation

Saying:

$$f(n) \in O\big(g(n)\big)$$

Is the same as the following expressions:

- $f(n) = O\big(g(n)\big)$
- $f(n)$ is $O\big(g(n)\big)$
- $f(n)$ is of order $g(n)$

But Wait...

Consider $T(n) = pn^2 + qn + s$, where $p, q, s \in \mathbb{R}^+$.

Claim: $T(n) \in O(n^3)$.
Proof:

$$T(n) = pn^2 + qn + s \leq pn^3 + qn^3 + sn^3 = (p + q + s)n^3$$

for all $n \geq 1$. The obtained inequality is exactly the definition of $O(\cdot)$, which requires $T(n) \leq cn^3$.

So, $T(n) \in O(n^3)$, where $c = p + q + s$, $n_0 = 1$, and $g(n) = n^3$.

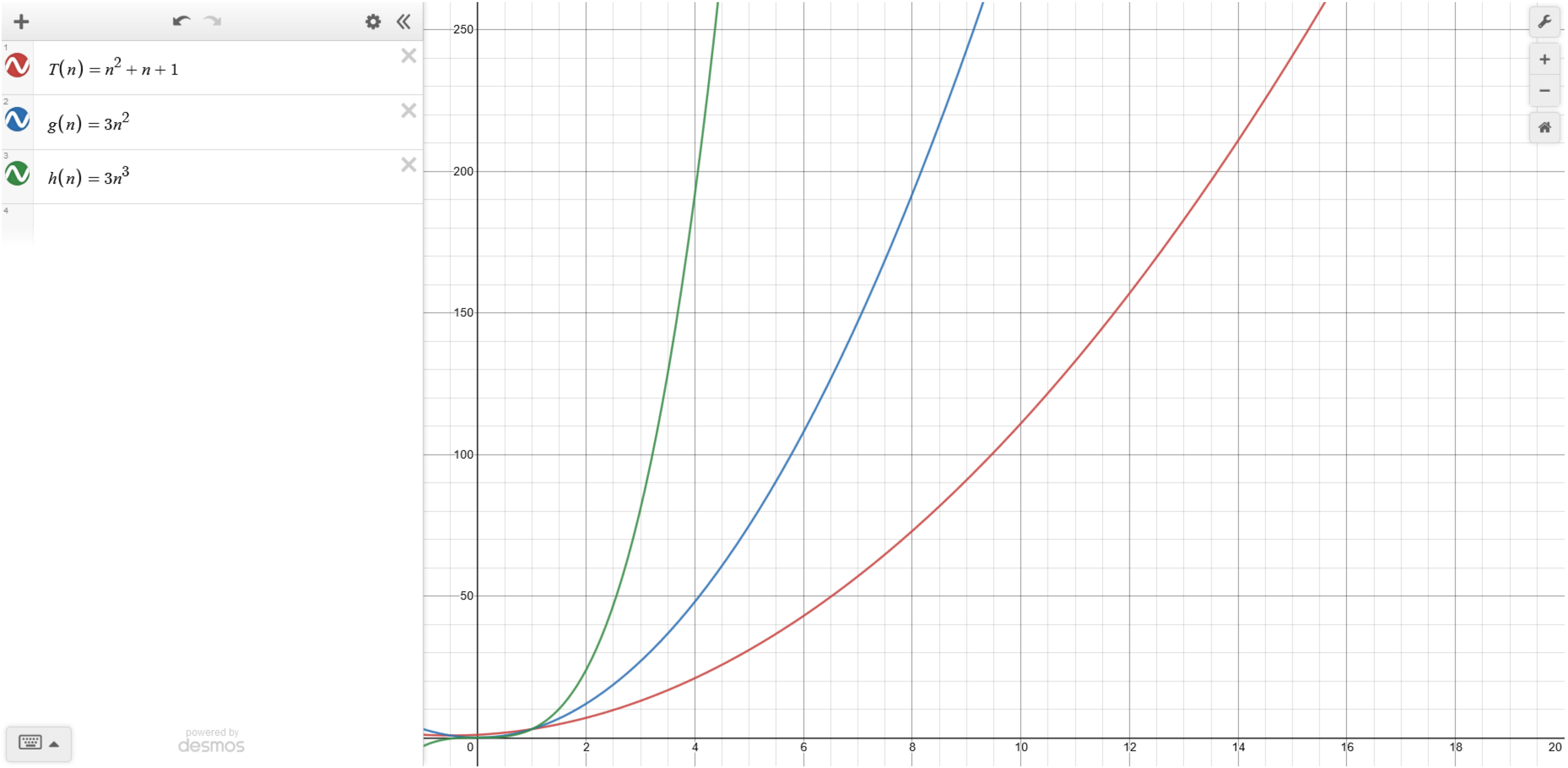
So, Which One Is It?

Both! It's just that n^2 is a tighter upper-bound for $T(n)$.

Let $p = q = s = 1$:

$$T(n) = n^2 + n + 1$$

So, $T(n) \leq 3n^2 \leq 3n^3$.


$T(n) = n^2 + n + 1$
$g(n) = 3n^2$
$h(n) = 3n^3$
250
200
150
100
50
0
2
4
6
8
10
12
14
16
18
20
powered by
desmos

From now on, when we ask for a $O(g(n))$ expression, we mean the tightest possible.

Example: Bubble Sort

```
algorithm BubbleSort(A:array)
   let n be the length of A
   repeat ← true

   while repeat do
      repeat ← false
      for i from 0 to n-2 do
         if A[i] > A[i+1] then
            swap(A, i, i+1)
            repeat ← true
         end if
      end for
   end while

   return A
end algorithm
```

Upper bound? Consider the content of an input array that forces the algorithm run the **most operations**.

Any idea?

Worst-case: The event of an input array sorted in descending order.

Example: Bubble Sort

```
algorithm BubbleSort(A:array)
   let n be the length of A
   repeat ← true

   while repeat do
      repeat ← false
      for i from 0 to n-2 do
         if A[i] > A[i+1] then
            swap(A, i, i+1)
            repeat ← true
         end if
      end for
   end while

   return A
end algorithm
```

Let c_1 be the costs associated to the statements outside of the while loop, and c_2 be the costs associated to the statements from and within the while loop.

$$T(n) = c_1 + \sum_{j=0}^{n-1} \sum_{i=0}^{n-2} c_2 = c_1 + c_2 n(n-1)$$
$$= c_1 + c_2 n^2 - c_2 n$$

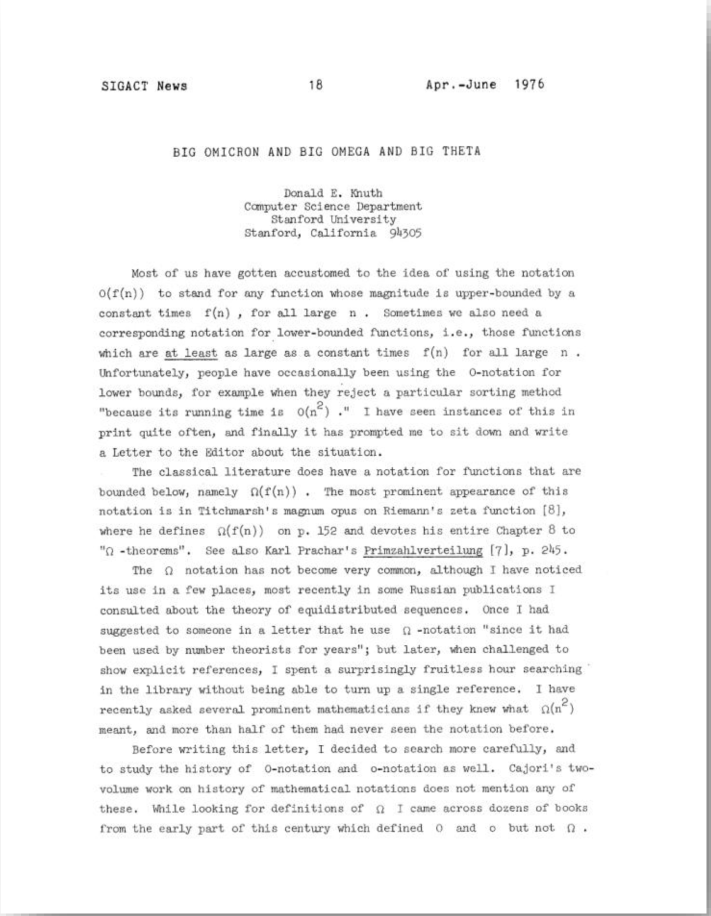
Claim: $T(n) \in O(n^2)$
Proof: Assume $c_1, c_2 \in \mathbb{Z}^+, c_1 \leq c_2$.

$$c_1 + c_2 n^2 - c_2 n \leq c_2 n^2$$

Which holds with $c = c_2$, $n_0 \geq 1$, and $g(n) = n^2$.

Unfortunately, people have occasionally been using the O-notation for lower bounds, for example when they reject a particular sorting method "because its running time is $O(n^2)$."

– Donald Knuth, 1976



SIGACT News 18 Apr.-June 1976

BIG OMICRON AND BIG OMEGA AND BIG THETA

Donald E. Knuth
Computer Science Department
Stanford University
Stanford, California 94305

Most of us have gotten accustomed to the idea of using the notation $O(f(n))$ to stand for any function whose magnitude is upper-bounded by a constant times $f(n)$, for all large n. Sometimes we also need a corresponding notation for lower-bounded functions, i.e., those functions which are at least as large as a constant times $f(n)$ for all large n. Unfortunately, people have occasionally been using the O-notation for lower bounds, for example when they reject a particular sorting method "because its running time is $O(n^2)$." I have seen instances of this in print quite often, and finally it has prompted me to sit down and write a Letter to the Editor about the situation.

The classical literature does have a notation for functions that are bounded below, namely $\Omega(f(n))$. The most prominent appearance of this notation is in Titchmarsh's magnum opus on Riemann's zeta function [8], where he defines $\Omega(f(n))$ on p. 152 and devotes his entire Chapter 8 to "Ω-theorems". See also Karl Prachar's Primzahlverteilung [7], p. 245.

The Ω notation has not become very common, although I have noticed its use in a few places, most recently in some Russian publications I consulted about the theory of equidistributed sequences. Once I had suggested to someone in a letter that he use Ω-notation "since it had been used by number theorists for years"; but later, when challenged to show explicit references, I spent a surprisingly fruitless hour searching in the library without being able to turn up a single reference. I have recently asked several prominent mathematicians if they knew what $\Omega(n^2)$ meant, and more than half of them had never seen the notation before.

Before writing this letter, I decided to search more carefully, and to study the history of O-notation and o-notation as well. Cajori's two-volume work on history of mathematical notations does not mention any of these. While looking for definitions of Ω I came across dozens of books from the early part of this century which defined O and o but not Ω.

https://dl.acm.org/doi/pdf/10.1145/1008328.1008329

03

Big-Ω

Asymptotic lower bounds

Big-Ω Notation

Useful to classify algorithms according to a **lower bound**.

Definition: $f(n) \in \Omega(g(n))$ if $\exists c > 0, \exists n_0 > 0$ such that $0 \leq cg(n) \leq f(n), \forall n \geq n_0$.

Plain English: If $f(n) \in \Omega(g(n))$, then $f(n)$ doesn't **grow** any **slower** than $g(n)$.

Example

Consider $T(n) = pn^2 + qn + s$, where $p, q, s \in \mathbb{R}^+$.

Claim: $T(n) \in \Omega(n^2)$.
Proof:

$$pn^2 \leq pn^2 + qn + s = T(n)$$

for all $n \geq 1$. The obtained inequality is exactly the definition of $\Omega(\cdot)$, which requires $pn^2 \leq T(n)$.

So, $T(n) \in \Omega(n^2)$, where $c = p$.

But Wait...

Consider $T(n) = pn^2 + qn + s$, where $p, q, s \in \mathbb{R}^+$.

Claim: $T(n) \in \Omega(n)$.
Proof:

$$pn \leq pn^2 + qn + s = T(n)$$

for all $n \geq 1$. The obtained inequality is exactly the definition of $\Omega(\cdot)$, which requires $pn \leq T(n)$.

So, $T(n) \in \Omega(n)$, where $c = p$.

Even Lower!

Consider $T(n) = pn^2 + qn + s$, where $p, q, s \in \mathbb{R}^+$.

Claim: $T(n) \in \Omega(1)$.
Proof:

$$1 \leq pn^2 + qn + s = T(n)$$

for all $n \geq 1$. The obtained inequality is exactly the definition of $\Omega(\cdot)$, which requires $1 \leq T(n)$.

So, $T(n) \in \Omega(1)$, where $c = 1$.

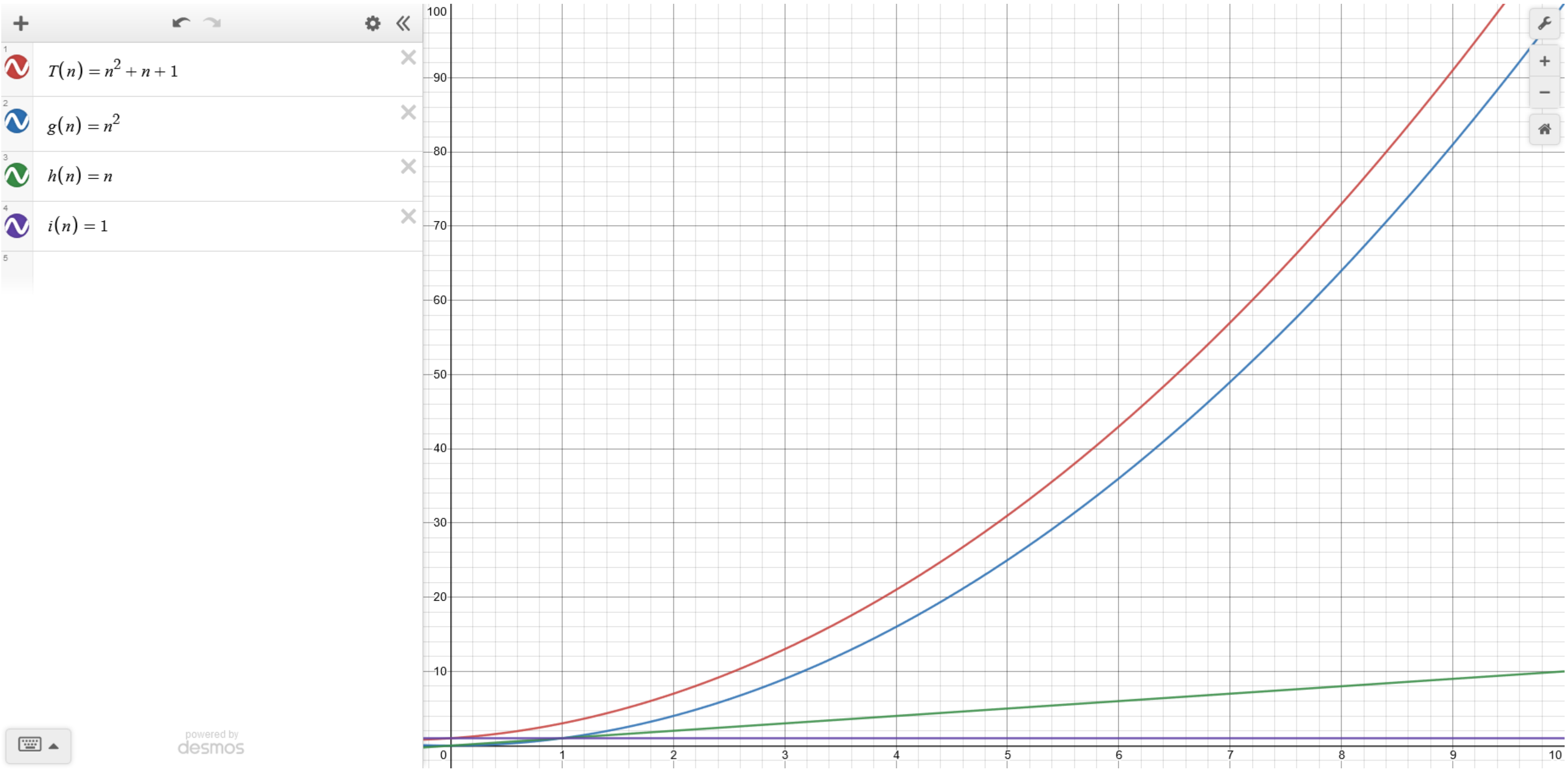
So, Which One Is it?

All three! It's just that n^2 is a tighter lower-bound for $T(n)$.

Let $p = q = s = 1$:

$$T(n) = n^2 + n + 1$$

So, $1 \leq n \leq n^2 \leq T(n)$.



$T(n) = n^2 + n + 1$
$g(n) = n^2$
$h(n) = n$
$i(n) = 1$
powered by
desmos
100
90
80
70
60
50
40
30
20
10
0
1
2
3
4
5
6
7
8
9
10

Example: Bubble Sort

```
algorithm BubbleSort(A:array)
   let n be the length of A
   repeat ← true

   while repeat do
      repeat ← false
      for i from 0 to n-2 do
         if A[i] > A[i+1] then
            swap(A, i, i+1)
            repeat ← true
         end if
      end for
   end while

   return A
end algorithm
```

Lower bound? Consider the content of an input array that makes the algorithm to run the **least operations**.

Any idea?

Best-case: The event of an input array already sorted.

Example: Bubble Sort

```
algorithm BubbleSort(A:array)
   let n be the length of A
   repeat ← true

   while repeat do
      repeat ← false
      for i from 0 to n-2 do
         if A[i] > A[i+1] then
            swap(A, i, i+1)
            repeat ← true
         end if
      end for
   end while

   return A
end algorithm
```

Let c_1 be the costs associated to the statements outside of the while loop, and c_2 be the costs associated to the statements from and within the while loop.

$$T(n) = c_1 + \sum_{i=0}^{n-2} c_2 = c_1 + c_2 n - c_2$$

Claim: $T(n) \in \Omega(n)$
Proof: Assume $c_1, c_2 \in \mathbb{Z}^+, c_1 \leq c_2$.

$$c_1 n \leq c_1 + c_2 n - c_2$$

The inequality holds with $c = c_1$, $n_0 \geq 1$, and $g(n) = n$.

04

Big-Θ

Asymptotic tight bounds

Big-Θ Notation

We use it to classify algorithms according to a **tight bound**.

Definition: $f(n) \in \Theta\big(g(n)\big)$ if $\exists c_1 > 0, \exists c_2 > 0, \exists n_0 > 0$ such that $0 \leq c_1 g(n) \leq f(n) \leq c_2 g(n), \forall n \geq n_0$.

Plain English: If $f(n) \in \Theta\big(g(n)\big)$, then $f(n)$ doesn't **grow** any **faster** nor **slower** than $g(n)$.

Example

Consider $T(n) = pn^2 + qn + s$, where $p, q, s \in \mathbb{R}^+$.

Claim: $T(n) \in \Theta(n^2)$.
Proof:

$$pn^2 \leq pn^2 + qn + s \leq (p + q + s)n^2$$

for all $n \geq 1$. The obtained inequality is exactly the definition of $\Theta(\cdot)$, which requires $pn^2 \leq T(n) \leq (p + q + s)n^2$.

So, $T(n) \in \Theta(n^2)$, where $c_1 = p$, $c_2 = p + q + s$, and $g(n) = n^2$.

Limits

$\lim_{n\to\infty} \frac{f(n)}{g(n)}$ reveals information about the asymptotic relationship between f and g:

$$\lim_{n\to\infty} \frac{f(n)}{g(n)} \neq 0, \infty \;\Rightarrow f \in \Theta(g)$$

$$\lim_{n\to\infty} \frac{f(n)}{g(n)} \neq \infty \;\Rightarrow f \in O(g)$$

$$\lim_{n\to\infty} \frac{f(n)}{g(n)} \neq 0 \;\Rightarrow f \in \Omega(g)$$

L'Hôpital's rule comes in handy: If $\lim_{n\to\infty} f(n) = \infty$ and $\log_{n\to\infty} g(n) = \infty$, then $\log_{n\to\infty} \frac{f(n)}{g(n)} = \log_{n\to\infty} \frac{f'(n)}{g'(n)}$.

http://web.mit.edu/broder/Public/asymptotics-cheatsheet.pdf

Asymptotic Growth Properties

If $f(n) \in O(g(n))$ and $f(n) \in \Omega(g(n))$, then $f(n) \in \Theta(g(n))$.

Transitivity:

- If $f(n) \in O(g(n))$ and $g(n) \in O(h(n))$, then $f(n) \in O(h(n))$.
- If $f(n) \in \Omega(g(n))$ and $g(n) \in \Omega(h(n))$, then $f(n) \in \Omega(h(n))$.
- If $f(n) \in \Theta(g(n))$ and $g(n) \in \Theta(h(n))$, then $f(n) \in \Theta(h(n))$.

Sum of functions:

- If $f(n) \in O(h(n))$ and $g(n) \in O(h(n))$, then $f(n) + g(n) \in O(h(n))$.
- Let k be a fixed constant, and let $f_1, f_2, \dots, f_k$ be functions such that $f_i(n) \in O(h(n))$ for all $1 \leq i \leq k$. Then $f_1(n) + f_2(n) + \cdots + f_k(n) \in O(h(n))$.
- If $f(n) \in \Theta(g(n))$, then $f(n) + g(n) \in \Theta(g(n))$.

Test yo' self

$3n^2 \log(n^2) + 4n \in \Omega(\log(n))$

$3n^2 \log(n) + 4n \in O(n^2)$

$6n^2 \log(2^n) + 4n^3 \in O(n^2 \log(n))$

$5n^2 + 5n + 7 \in \Theta(n^2)$

$4 \log^2(n) + n + 2^n \in \Theta(n)$

$15n^3 \in \Omega(1)$

Big-O is **not** a synonym of worst case.

Big-Ω is **not** a synonym of best case.

Big-Θ is **not** a synonym of average case.

We use asymptotic notation to represent the boundaries of the orders of growth for the runtime expressions in terms of the input size.

Best/Worst Case Example

```
i ← 2

while i < n * n do

   num ← Random(100)

   if num > 50 then
      i ← i * 3
   else
      i ← i + 5
   end if

end while
```

What does it mean to have a best/worst case?

Lower bound: $\Omega(\log(n))$ if the random number is always greater than 50.

Upper bound: $O(n^2)$ if the random number is always less than or equal than 50.

Best/Worst Case Example

```
Let A be an array storing n integers

for i from 0 to < n - 2 do

   j ← i + 1

   while j < n and A[i] < A[j] do
      j ← j + 1
   end while

   if j < n then
      temp ← A[i]
      A[i] ← A[j]
      A[j] ← temp
   end if

end for
```

What does it mean to have a best/worst case?

Lower bound: $\Omega(n)$ when the while loop doesn't execute because A[i] is never less than A[j].

Upper bound: $O(n^2)$ when the while loop always runs all the way from $i+1$ to $n-1$.

STAHP!

Do you have any questions?

CREDITS: This presentation template was created by Slidesgo, including icons by Flaticon, infographics & images by Freepik and illustrations by Stories