CS240

Nathan Sprague

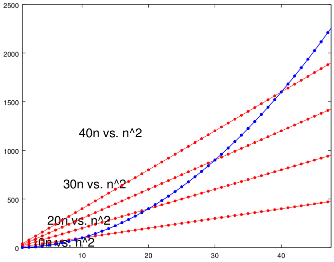
9/1/2021

Big-Θ / Order Notation

- Informal description: Growth functions are categorized according to their dominant (fastest growing) term
- Constants and lower-order terms are discarded
- Examples:
  - $10n \in \Theta(n)$
  - $5n^2 + 2n + 3 \in \Theta(n^2)$
  - $n \log n + n \in \Theta(n \log n)$
- We could read this as "$10n$ is order n"

Why Drop the Constants?

- Example...

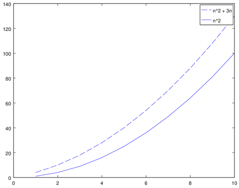


Why Drop the Constants?

- Despite constants, functions from slower growing classes will always be faster eventually

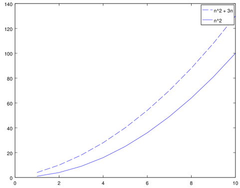
Why Drop Lower Order Terms

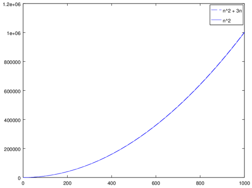
- Contribution of lower-order terms becomes insignificant as input size increases
- This difference looks important:



Why Drop Lower Order Terms

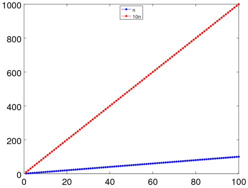
- Contribution of lower-order terms becomes insignificant as input size increases
- This difference looks important:
- It looks less important now.





Are we SURE we want to drop the constants?

For two growth functions in the same complexity class, constant factors continue to have an impact, regardless of input size...



Why Drop the Constants? (Again?)

- Real goal is to understand the relative impact of increasing input size
- Equivalently: allow us to predict the impact of using a faster computer
- Constant factors are influenced by all the distractions we mentioned before:
  - Choice of basic operation
  - Programming language
  - ...

Why Drop the Constants? (Again?)

- Real goal is to understand the relative impact of increasing input size
- Equivalently: allow us to predict the impact of using a faster computer
- Constant factors are influenced by all the distractions we mentioned before:
  - Choice of basic operation
  - Programming language
  - ...
- That said... We DO care about constant factors.

Formal Definition of Big-O

Big O

For $T(n)$ a non-negative function, $T(n) \in O(f(n))$ if and only if there exist positive constants c and n_0 such that

$$T(n) \leq cf(n) \text{ for all } n > n_0.$$

Formal Definition of Big-O

Big O

For $T(n)$ a non-negative function, $T(n) \in O(f(n))$ if and only if there exist positive constants c and n_0 such that

$$T(n) \leq cf(n) \text{ for all } n > n_0.$$

- Informal rule of "dropping constants" follows immediately:
  - $50n \stackrel{?}{\in} O(n)$

Formal Definition of Big-O

Big O

For $T(n)$ a non-negative function, $T(n) \in O(f(n))$ if and only if there exist positive constants c and n_0 such that

$$T(n) \leq cf(n) \text{ for all } n > n_0.$$

- Informal rule of "dropping constants" follows immediately:
  - $50n \stackrel{?}{\in} O(n)$
  - Yes! choose c $= 50$, $n_0 = 1$, clearly
  - $50n \leq 50n$ for all $n > 1$

Formal Definition of Big-O

Big O

For $T(n)$ a non-negative function, $T(n) \in O(f(n))$ if and only if there exist positive constants c and n_0 such that

$$T(n) \leq cf(n) \text{ for all } n > n_0.$$

- Informal rule of "dropping lower-order terms" also follows:
  - $n^2 + 40n \stackrel{?}{\in} O(n^2)$

Formal Definition of Big-O

Big O

For $T(n)$ a non-negative function, $T(n) \in O(f(n))$ if and only if there exist positive constants c and n_0 such that

$$T(n) \leq cf(n) \text{ for all } n > n_0.$$

- Informal rule of "dropping lower-order terms" also follows:
  - $n^2 + 40n \stackrel{?}{\in} O(n^2)$
  - Notice that: $n^2 + 40n \leq n^2 + 40n^2 = 41n^2$

Formal Definition of Big-O

Big O

For $T(n)$ a non-negative function, $T(n) \in O(f(n))$ if and only if there exist positive constants c and n_0 such that

$$T(n) \leq cf(n) \text{ for all } n > n_0.$$

- Informal rule of "dropping lower-order terms" also follows:
  - $n^2 + 40n \overset{?}{\in} O(n^2)$
  - Notice that: $n^2 + 40n \leq n^2 + 40n^2 = 41n^2$
  - Choose c $= 41$, $n_0 = 1$, clearly $n^2 + 40n \leq 41n^2$ for all $n > 1$

Big O Describes an Upper Bound

- Big O is loosely analogous to $\leq$
- All of these statements are true:
  $n^2 \in O(n^2)$
  $n^2 \in O(n^4)$
  $n^2 \in O(n!)$
  ...
  $2n^2 \in O(n^2)$

Upper Bounds

- Big-O descriptions are imprecise in two different ways:
  - No constants or lower-order terms
    - GOOD: fewer distractions

Upper Bounds

- Big-O descriptions are imprecise in two different ways:
  - No constants or lower-order terms
    - GOOD: fewer distractions
  - Only provides an upper bound. Correct to say an algorithm requires $O(n^3)$ steps, even if it only requires n steps.
    - UNFORTUNATE: conveys an incomplete analysis

Socrative Quiz!

Alyce is working on the analysis of a complex algorithm for finding sequence matches in a DNA database. She can easily show that the algorithm requires no more than $n^2 + n$ base-pair comparisons in the worst case. She hopes to show that the algorithm requires at most $n \log n + n$ comparisons. How should Alyce describe the running time of the algorithm given the current state of her analysis?

A) $O(n^3)$
B) $O(n^2 + n)$
C) $O(n^2)$
D) $O(n \log n + n)$
E) $O(n)$

Big Omega

Big Ω

For $T(n)$ a non-negative function, $T(n) \in \Omega(f(n))$ if and only if there exist positive constants c and n_0 such that

$$T(n) \geq cf(n) \text{ for all } n > n_0.$$

- Big Ω is loosely analogous to $\geq$
- All of these statements are true:
  $n^2 \in \Omega(n^2)$
  $n^4 \in \Omega(n^2)$
  $n! \in \Omega(n^2)$
  ...
  $n^2 \in \Omega(2n^2)$

Big Theta

Big Θ

$f(n) \in \theta(g(n))$ iff,

$$f(n) \in O(g(n)) \text{ and } f(n) \in \Omega(g(n))$$

- Big Θ is loosely analogous to $=$
- Which of these statements are true?
  $n^2 \stackrel{?}{\in} \Theta(n^2)$
  $2n^2 \stackrel{?}{\in} \Theta(n^2)$
  $n^2 \stackrel{?}{\in} \Theta(n^4)$
  $5n^2 + 2n \stackrel{?}{\in} \Theta(4n^3)$

Big Theta

Big Θ

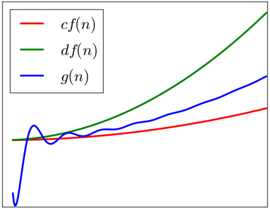
$f(n) \in \theta(g(n))$ iff,

$$f(n) \in O(g(n)) \text{ and } f(n) \in \Omega(g(n))$$

- Big Θ is loosely analogous to $=$
- Which of these statements are true?
  $n^2 \in \Theta(n^2)$
  $2n^2 \in \Theta(n^2)$
  $n^2 \notin \Theta(n^4)$
  $5n^2 + 2n \notin \Theta(4n^3)$

Socrative Quiz

What relationship(s) is(are) illustrated by the following figure?



A) $f(n) \in O(g(n))$
B) $f(n) \in \Omega(g(n))$
C) $f(n) \in \Theta(g(n))$
D) $g(n) \in O(f(n))$
E) $g(n) \in \Omega(f(n))$
F) $g(n) \in \Theta(f(n))$
G) A, B and C are all correct
H) D, E and F are all correct

Alternate Definition of Big-O

Big O

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

$$\lim_{n \to \infty} \frac{f(n)}{g(n)} = c < \infty$$

where c is some constant (possibly 0)

Alternate Definitions of Big-O

Big O

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

$$\lim_{n\to\infty} \frac{f(n)}{g(n)} = c < \infty$$

where c is some constant (possibly 0)

- $n^3 + 2n \in n^3$

Alternate Definitions of Big-O

Big O

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

$$\lim_{n \to \infty} \frac{f(n)}{g(n)} = c < \infty$$

where c is some constant (possibly 0)

- $n^3 + 2n \in n^3$
- $$\lim_{n \to \infty} \frac{n^3 + 2n}{n^3} = \lim_{n \to \infty} 1 + \frac{2}{n^2} = 1$$

Alternate Definitions of O, Ω, Θ

Big Ω

$f(n) \in \Omega(g(n))$ if

$$\lim_{n\to\infty} \frac{f(n)}{g(n)} = c > 0$$

where c is some constant (possibly ∞)

Alternate Definitions of O, Ω, Θ

Big Θ

$f(n) \in \Theta(g(n))$ if

$$\lim_{n \to \infty} \frac{f(n)}{g(n)} = c, 0 < c < \infty$$

where c is some constant.

Algorithm Analysis Algorithm

- STEP 1: Select a measure of input size and a basic operation
- STEP 2: Develop a function $T(n)$ that describes the number of times the basic operation occurs as a function of input size
- STEP 3: Describe $T(n)$ using order notation (Big-O)
  - Big-O for an upper bound

    "The algorithm is at least this fast!"
  - Big-Ω for a lower bound

    "The algorithm is at least this slow!"
  - Big-Θ for both upper and lower bound

A Complication

- Let's analyze this algorithm:

```java
public static boolean contains(int target,
                               int[] numbers) {
  for (int number : numbers) {
    if (number == target) {
      return true;
    }
  }
  return false;
}
```

Best, Worst, Average Case

```java
public static boolean contains(int target,
                               int[] numbers) {
  for (int number : numbers) {
    if (number == target) {
      return true;
    }
  }
  return false;
}
```

- Best Case: 1 comparison, $O(1)$
- Worst Case: n comparisons, $O(n)$
- Average Case: $\frac{n+1}{2}$ comparisons, $O(n)$

Refined Algorithm Analysis Algorithm

- STEP 1: Decide on best, worst, or average case analysis
- STEP 2: Select a measure of input size and a basic operation
- STEP 3: Find a function $T(n)$ that describes the number of times the basic operation occurs
- STEP 4: Describe $T(n)$ using order notation:
  - Big-O for an upper bound

    "The algorithm is at least this fast!"
  - Big-Ω for a lower bound

    "The algorithm is at least this slow!"
  - Big-Θ for both upper and lower bound

Socrative Quiz (1)

What is the exact growth function for the following code snippet, using "+=" as the basic operation and the length of numbers as the input size?

```
public static int someFunc1(int\[\] numbers) {
  int sum = 0;

  for (int num : numbers) {
    sum += num;
    for (int i = 0; i < 20; i++) {
      sum += i;
    }
  }
  return sum;
}
```

A) $T(n) = n$
B) $T(n) = 20$
C) $T(n) = 21n$
D) $T(n) = n + 20$
E) None of the above

Socrative Quiz

How should we describe the running time of the following code snippet?

```
public static int someFunc1(int[] numbers) {
   int sum = 0;

   for (int num : numbers) {
     sum += num;
     for (int i = 0; i < 20; i++) {
       sum += i;
     }
   }
   return sum;
 }
```

A) $O(n)$
B) $\Omega(n)$
C) $\Theta(n)$
D) $O(21n)$
E) $\Omega(21n)$
F) $\Theta(21n)$

Quiz

- Input size? Basic operation? Exact growth function?
- Big-O, Ω, Θ?

```java
public static int someFunc2(int[] numbers) {
  int sum = 0;
  int index = 1;

  while (index < numbers.length) {
    sum += numbers[index];
    index *= 2;
  }

  return sum;
}
```

Quiz

- Input size? Basic operation? Exact growth function?
- Big-O, Ω, Θ?

```java
public static int someFunc3(int[] numbers) {
  int sum = 0;
  int index = 1;

  while (index < numbers.length) {
    sum += numbers[index];
    index *= 2;

    for (int i = 0; i < numbers.length; i++) {
      sum += i;
    }
  }
  return sum;
}
```

L'Hôpital's Rule

L'Hôpital's Rule

If $\lim_{n\to\infty} f(n) = \lim_{n\to\infty} g(n) = \infty$ and $f'(n)$ and $g'(n)$ exist, then

$$\lim_{n\to\infty} \frac{f(n)}{g(n)} = \lim_{n\to\infty} \frac{f'(n)}{g'(n)}$$

L'Hôpital Example

- $n \log_2 n \stackrel{?}{\in} O(n^2)$

L'Hôpital Example

- $n \log_2 n \stackrel{?}{\in} O(n^2)$
- $\displaystyle\lim_{n\to\infty} \frac{n \log_2 n}{n^2} = \lim_{n\to\infty} \frac{\log_2 n}{n}$

L'Hôpital Example

- $n \log_2 n \overset{?}{\in} O(n^2)$
- $\lim_{n \to \infty} \frac{n \log_2 n}{n^2} = \lim_{n \to \infty} \frac{\log_2 n}{n}$
- $= \lim_{n \to \infty} \frac{\ln n}{n \ln 2}$ (Recall that $\log_b(n) = \frac{\log_k n}{\log_k b}$)

L'Hôpital Example

- $n \log_2 n \overset{?}{\in} O(n^2)$
- $\lim_{n \to \infty} \frac{n \log_2 n}{n^2} = \lim_{n \to \infty} \frac{\log_2 n}{n}$
- $= \lim_{n \to \infty} \frac{\ln n}{n \ln 2}$ (Recall that $\log_b(n) = \frac{\log_k n}{\log_k b}$)
- Apply L'Hôpital's rule:
- $= \lim_{n \to \infty} \frac{\frac{1}{n}}{\ln 2}$ (Recall that $\frac{d}{dx} \ln x = 1/x$)

L'Hôpital Example

- $n \log_2 n \overset{?}{\in} O(n^2)$
- $\lim_{n \to \infty} \frac{n \log_2 n}{n^2} = \lim_{n \to \infty} \frac{\log_2 n}{n}$
- $= \lim_{n \to \infty} \frac{\ln n}{n \ln 2}$ (Recall that $\log_b(n) = \frac{\log_k n}{\log_k b}$)
- Apply L'Hôpital's rule:
- $= \lim_{n \to \infty} \frac{\frac{1}{n}}{\ln 2}$ (Recall that $\frac{d}{dx} \ln x = 1/x$)
- $= \lim_{n \to \infty} \frac{1}{n \ln 2} = 0$

What If We Want to Show That f(n) is NOT O(g(n))

- Easiest approach is usually to show:

$$\lim_{n \to \infty} \frac{f(n)}{g(n)} = \infty$$

OpenDSA Question

Suppose that a particular algorithm has time complexity $T(n) = 3 \times 2^n$ and that executing an implementation of it on a particular machine takes t seconds for n inputs. Now suppose that we are presented with a machine that is 64 times as fast. How many inputs could we process on the new machine in t seconds?

OpenDSA Question

- Let's call the input size we could handle before n_{old}. The number of steps we completed in t seconds was: $3 \times 2^{n_{old}}$.
- Since our new computer is 64 times faster, the number of steps we can perform in t seconds is now $64 \times 3 \times 2^{n_{old}}$
- Our complexity function tells us that $steps = 3 \times 2^n$, we can solve for size (n):
- $s = 3 \times 2^n$
- $s/3 = 2^n$
- $\log_2(s/3) = n$
- $n = \log_2(s/3)$

OpenDSA Question

Now we plug in our step budget for s:

- $n = \log_2(\frac{64 \times 3 \times 2^{n_{old}}}{3})$
- $n = \log_2(64 \times 2^{n_{old}})$
- $n = \log_2(2^6 \times 2^{n_{old}})$
- $n = \log_2(2^{n_{old}+6})$
- $n = n_{old} + 6$