Big O and Memesorts:

How slow can we go?

Neil Thistlethwaite

Who needs Big O anyways?

Which of the following sorting algorithms is “better”?

Who needs Big O anyways?

Which of the following sorting algorithms is “better”?

> **Selection sort:** find the least element, swap with the first element of the array, and then repeat for the second-least element, swapping with the second element, etc.

Who needs Big O anyways?

Which of the following sorting algorithms is “better”?

Selection sort: find the least element, swap with the first element of the array, and then repeat for the second-least element, swapping with the second element, etc.

Bogo sort: while the list isn’t sorted, shuffle it.

Who needs Big O anyways?

Which of the following sorting algorithms is “better”?

Selection sort: find the least element, swap with the first element of the array, and then repeat for the second-least element, swapping with the second element, etc.

Bogo sort: while the list isn’t sorted, shuffle it.
(*clearly*, bogo sort)

Algorithm Analysis

Typically, when we consider multiple choices for an algorithm, we compare them by looking at tradeoffs in **time** and **memory**.

Algorithm Analysis

Typically, when we consider multiple choices for an algorithm, we compare them by looking at tradeoffs in **time** and **memory**.

Usually, we care about these tradeoffs as the size of our input to the program, denoted n, increases. This is because for small n, other costs tend to dominate.

Algorithm Analysis

Typically, when we consider multiple choices for an algorithm, we compare them by looking at tradeoffs in **time** and **memory**.

Usually, we care about these tradeoffs as the size of our input to the program, denoted n, increases. This is because for small n, other costs tend to dominate.

However, it's often hard (and unnecessary) to characterize runtime and memory usage explicitly in terms of n – is there something better we can use?

Big O

Big O

O

Big O

Big O notation is a mathematical notation we can use to describe the asymptotic behavior of a function (i.e. as its argument increases and approaches infinity).

Big O

Big O notation is a mathematical notation we can use to describe the asymptotic behavior of a function (i.e. as its argument increases and approaches infinity).

Definition: $f(n) = O(g(n))$, or $f(n)$ is "in $O(g(n))$" if and only if there exists some C and n_0 such that $|f(n)| \leq C\, g(n)$ for all $n \geq n_0$.

Big O

Big O notation is a mathematical notation we can use to describe the asymptotic behavior of a function (i.e. as its argument increases and approaches infinity).

Definition: $f(n) = O(g(n))$, or $f(n)$ is "in $O(g(n))$" if and only if there exists some C and n_0 such that $|f(n)| \leq C\,g(n)$ for all $n \geq n_0$.

In English: a function f is in Big O of another function g if there's some point where for any larger values of the argument, f is bounded by a constant multiple of g.

Big O

Big O notation is a mathematical notation we can use to describe the asymptotic behavior of a function (i.e. as its argument increases and approaches infinity).

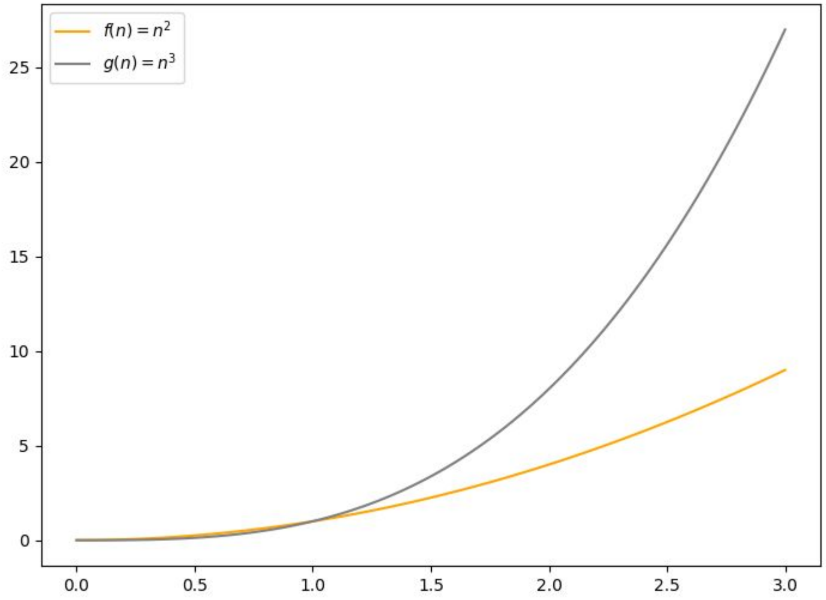
Definition: $f(n) = O(g(n))$, or $f(n)$ is “in $O(g(n))$” if and only if there exists some C and n_0 such that $|f(n)| \leq C\, g(n)$ for all $n \geq n_0$.

In English: a function f is in Big O of another function g if there’s some point where for any larger values of the argument, f is bounded by a constant multiple of g.

A pair of values C, n_0 that show this for some f and g are said to be **witnesses**.

Big O - Examples

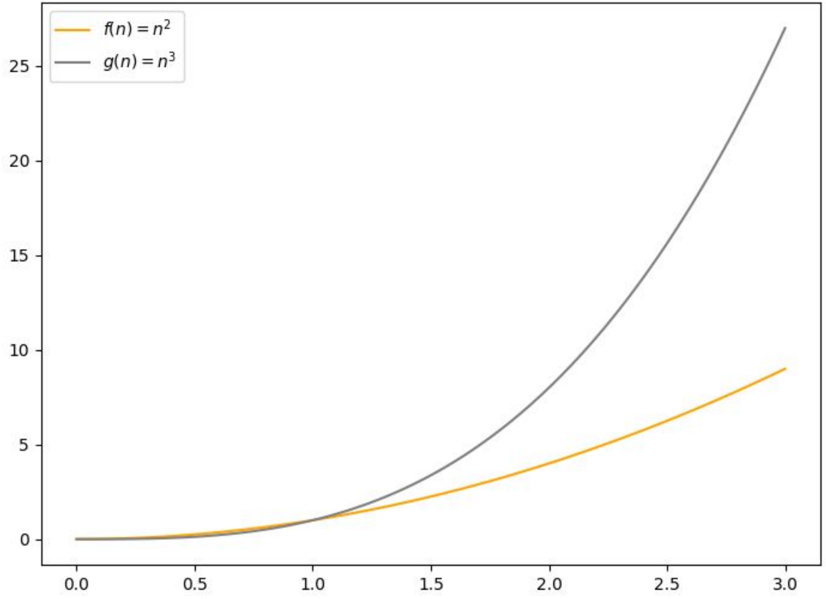
$n^2 = O(n^3)$



Big O - Examples

$n^2 = O(n^3)$

$4n(n-1) = O(n^2)$

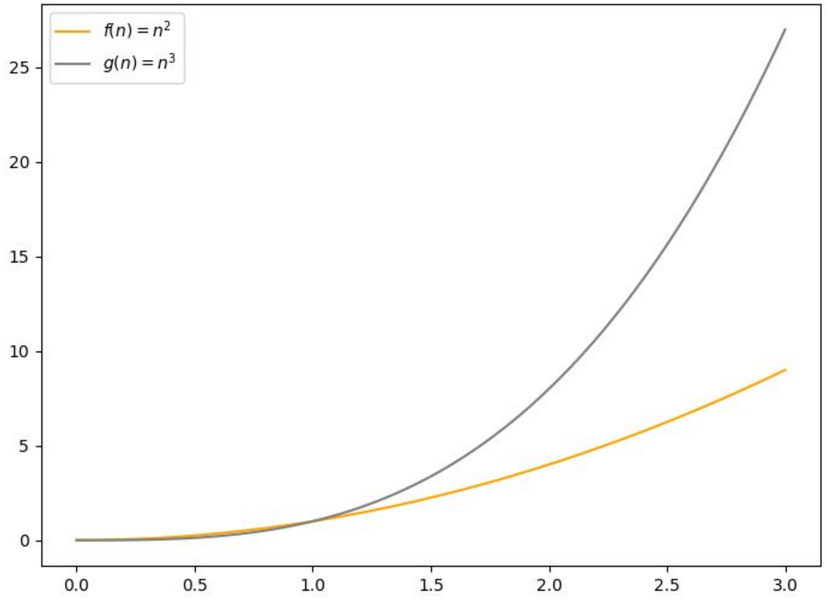


Big O - Examples

$n^2 = O(n^3)$

$4n(n-1) = O(n^2)$

$n \log n = O(n^2)$



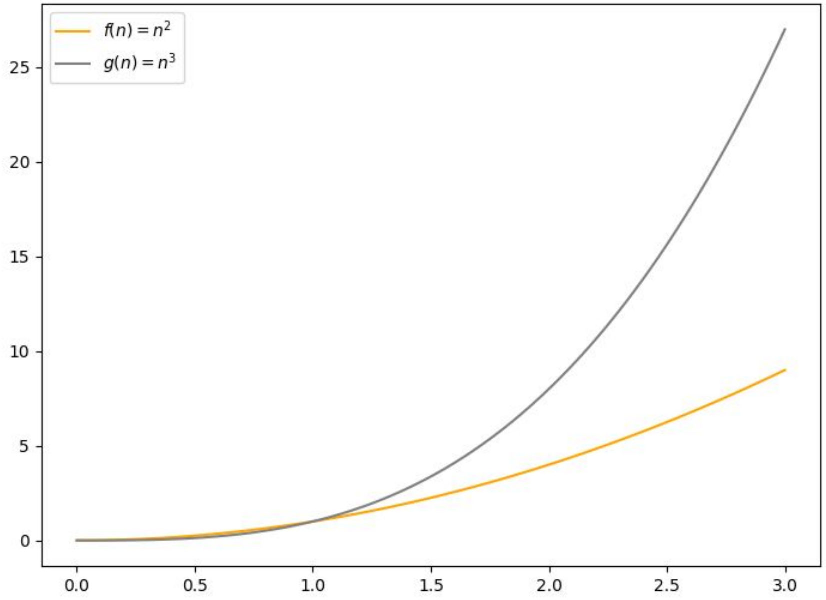
Big O - Examples

$n^2 = O(n^3)$

$4n(n-1) = O(n^2)$

$n \log n = O(n^2)$

$n! = O(n^n)$



Big O - How does this help?

We don’t have to have the function explicitly represented in order to be able to bound its asymptotic complexity.

Big O - How does this help?

We don't have to have the function explicitly represented in order to be able to bound its asymptotic complexity.

e.g. let $f(n)$ represent the time it takes me to make n slides for this presentation. We can think of Sherry saying "Neil make slides" as the algorithm, and n as the input (the number of slides needed for Sherry to be happy with the presentation).

Big O - How does this help?

We don’t have to have the function explicitly represented in order to be able to bound its asymptotic complexity.

e.g. let $f(n)$ represent the time it takes me to make n slides for this presentation. We can think of Sherry saying “Neil make slides” as the algorithm, and n as the input (the number of slides needed for Sherry to be happy with the presentation).

Since each slide will take Neil roughly the same amount of time to make (constant time is $O(1)$), it’s going to take him $n * O(1) = O(n)$ time to make n slides.

Big O - Sorting

A good comparison based sort should have asymptotic complexity O(n log n). Indeed, there are many algorithms that can achieve this: for example, quicksort and mergesort (although quicksort has worst case $O(n^2)$, there exist advanced variants without this degradation in performance)

Big O - Sorting

A good comparison based sort should have asymptotic complexity O(n log n). Indeed, there are many algorithms that can achieve this: for example, quicksort and mergesort (although quicksort has worst case $O(n^2)$, there exist advanced variants without this degradation in performance)

Can we do better?

Big O - Sorting

Can we do better?

NO!

Sorting Lower Bound Proof

Here’s the Sherry-mandated proof of this workshop:

Sorting Lower Bound Proof

Here’s the Sherry-mandated proof of this workshop:

We can prove that any comparison based sort must require at least O(n log n) comparisons:

Sorting Lower Bound Proof

Here's the Sherry-mandated proof of this workshop:

We can prove that any comparison based sort must require at least O(n log n) comparisons:

Suppose we're sorting a list $[a_1, a_2, \ldots , a_n]$. A single comparison between elements i and j returns the answer to whether $a_i > a_j$. Observe the following: without knowing anything about the elements, there are n! possible permutations, any of which could be the possible correct sorted list (why?).

Sorting Lower Bound Proof

Whenever we observe a comparison, we can at most rule out half of the remaining possible permutations. This is because if any given comparison were to rule out significantly more than half of the permutations, we could “adversarially” choose to give the opposite result for that comparison.

Sorting Lower Bound Proof

Whenever we observe a comparison, we can at most rule out half of the remaining possible permutations. This is because if any given comparison were to rule out significantly more than half of the permutations, we could "adversarially" choose to give the opposite result for that comparison (if a comparison is to be useful, then the results of the comparison divide the remaining "valid permutation set" into two non-overlapping subsets), forcing it to result in leaving significantly fewer than half of the permutations.

Sorting Lower Bound Proof

Whenever we observe a comparison, we can at most rule out half of the remaining possible permutations. This is because if any given comparison were to rule out significantly more than half of the permutations, we could “adversarially” choose to give the opposite result for that comparison (if a comparison is to be useful, then the results of the comparison divide the remaining “valid permutation set” into two non-overlapping subsets), forcing it to result in leaving significantly fewer than half of the permutations.

Therefore, our “worst case” is actually when a comparison rules out exactly half of the remaining permutations. From here, simple math will give us the number of comparisons we need.

Sorting Lower Bound Proof

The number of consistent permutations left after observing t comparisons is therefore $n! * (1/2)^t$. By definition, we are done sorting when there is only one permutation left.

Sorting Lower Bound Proof

The number of consistent permutations left after observing t comparisons is therefore $n! * (1/2)^t$. By definition, we are done sorting when there is only one permutation left. Solving for t,

$$n! * (1/2)^t \leq 1$$

Sorting Lower Bound Proof

The number of consistent permutations left after observing t comparisons is therefore $n! * (1/2)^t$. By definition, we are done sorting when there is only one permutation left. Solving for t,

$$n! * (1/2)^t \leq 1$$

$$n! \leq 2^t$$

Sorting Lower Bound Proof

The number of consistent permutations left after observing t comparisons is therefore $n! * (1/2)^t$. By definition, we are done sorting when there is only one permutation left. Solving for t,

$$n! * (1/2)^t \leq 1$$

$$n! \leq 2^t$$

$$\log_2 n! \leq \log_2 2^t = t$$

Sorting Lower Bound Proof

The number of consistent permutations left after observing t comparisons is therefore $n! * (1/2)^t$. By definition, we are done sorting when there is only one permutation left. Solving for t,

$$n! * (1/2)^t \leq 1$$

$$n! \leq 2^t$$

$$\log_2 n! \leq \log_2 2^t = t$$

By Stirling's Approximation,

$$\log_2 n! = n\log_2 n - n\log_2 e + O(\log_2 n) \leq t$$

Sorting Lower Bound Proof

The number of consistent permutations left after observing t comparisons is therefore $n! * (1/2)^t$. By definition, we are done sorting when there is only one permutation left. Solving for t,

$$n! * (1/2)^t \leq 1$$

$$n! \leq 2^t$$

$$\log_2 n! \leq \log_2 2^t = t$$

By Stirling's Approximation,

$$\log_2 n! = n \log_2 n - n \log_2 e + O(\log_2 n) \leq t$$

By definition of big O,

$$t = O(n \log_2 n) - O(n) + O(\log_2 n) = O(n \log n)$$

Let’s analyze some Meme Sorts!

Let’s get back to this question: which of the following sorting algorithms is “faster”?

Selection sort: find the least element, swap with the first element of the array, and then repeat for the second-least element, swapping with the second element, etc.

Bogo sort: while the list isn’t sorted, shuffle it.

Let’s analyze some Meme Sorts!

Let’s get back to this question: which of the following sorting algorithms is “faster”?

Selection sort: find the least element, swap with the first element of the array, and then repeat for the second-least element, swapping with the second element, etc.

Finding the least element is an $O(n)$ operation.

Let’s analyze some Meme Sorts!

Let’s get back to this question: which of the following sorting algorithms is “faster”?

> **Selection sort:** find the least element, swap with the first element of the array, and then repeat for the second-least element, swapping with the second element, etc.
>
> Finding the least element is an $O(n)$ operation. Swapping this with another element is an $O(1)$ operation.

Let’s analyze some Meme Sorts!

Let’s get back to this question: which of the following sorting algorithms is “faster”?

Selection sort: find the least element, swap with the first element of the array, and then repeat for the second-least element, swapping with the second element, etc.

Finding the least element is an $O(n)$ operation. Swapping this with another element is an $O(1)$ operation. However we repeat this $O(n)$ times, so we have $O(n) * (O(n) + O(1)) = O(n^2)$.

Let’s analyze some Meme Sorts!

Let’s get back to this question: which of the following sorting algorithms is “faster”?

Selection sort: find the least element, swap with the first element of the array, and then repeat for the second-least element, swapping with the second element, etc.

Finding the least element is an $O(n)$ operation. Swapping this with another element is an $O(1)$ operation. However we repeat this $O(n)$ times, so we have $O(n) * (O(n) + O(1)) = O(n^2)$. Note that on the k^{th} iteration, we can technically only look at the last n-k elements, which means we’re actually $O(n + (n-1) + (n-2) + \ldots + 1)$.

Let’s analyze some Meme Sorts!

Let’s get back to this question: which of the following sorting algorithms is “faster”?

Selection sort: find the least element, swap with the first element of the array, and then repeat for the second-least element, swapping with the second element, etc.

Finding the least element is an $O(n)$ operation. Swapping this with another element is an $O(1)$ operation. However we repeat this $O(n)$ times, so we have $O(n) * (O(n) + O(1)) = O(n^2)$. Note that on the k^{th} iteration, we can technically only look at the last n-k elements, which means we’re actually $O(n + (n-1) + (n-2) + \ldots + 1)$. But this is just $O((n)(n-1)/2) = O(n^2)$!

Let’s analyze some Meme Sorts!

Let’s get back to this question: which of the following sorting algorithms is “faster”?

Bogo sort: while the list isn’t sorted, shuffle it.

Let’s analyze some Meme Sorts!

Let’s get back to this question: which of the following sorting algorithms is “faster”?

> **Bogo sort:** while the list isn’t sorted, shuffle it.
>
> This is left as an exercise to the reader.

More Meme Sorts!

Stalin Sort: Delete any elements that aren’t in order (send them to the gulag).

More Meme Sorts!

Stalin Sort: Delete any elements that aren't in order (send them to the gulag).

Time Sort (Sleep Sort): To sort n elements, spawn n threads. Each thread sleeps for a_i seconds, and then places its value at the end of the work-in-progress sorted array.

More Meme Sorts!

Stalin Sort: Delete any elements that aren't in order (send them to the gulag).

Time Sort (Sleep Sort): To sort n elements, spawn n threads. Each thread sleeps for a_i seconds, and then places its value at the end of the work-in-progress sorted array.

Sorting as a Service (SaaS): It's 2019! Why sort yourself, when you can just send all your data to a sketchy API and have them sort it for you?

Problems! (Courtesy of Sherry)

1. Find the big O of $\sum_{k=1}^{n} k^c$ for some $c > 1$
2. Find the big O of $\sum_{k=1}^{n} \log k$
3. Prove that $\log^k n = O(n^m)$ for any $k, m \in \mathbb{R}^{++}$
4. An algorithm flips a fair coin until it gets heads. What is the expected run time of this algorithm?
5. Answer the following two questions:
   (a) Prove that the output of this algorithm is the greatest common divisor of a_0 and b_0.
   (b) Prove the algorithm runs in $O(\log b_0)$ time (assuming basic operations take constant time).