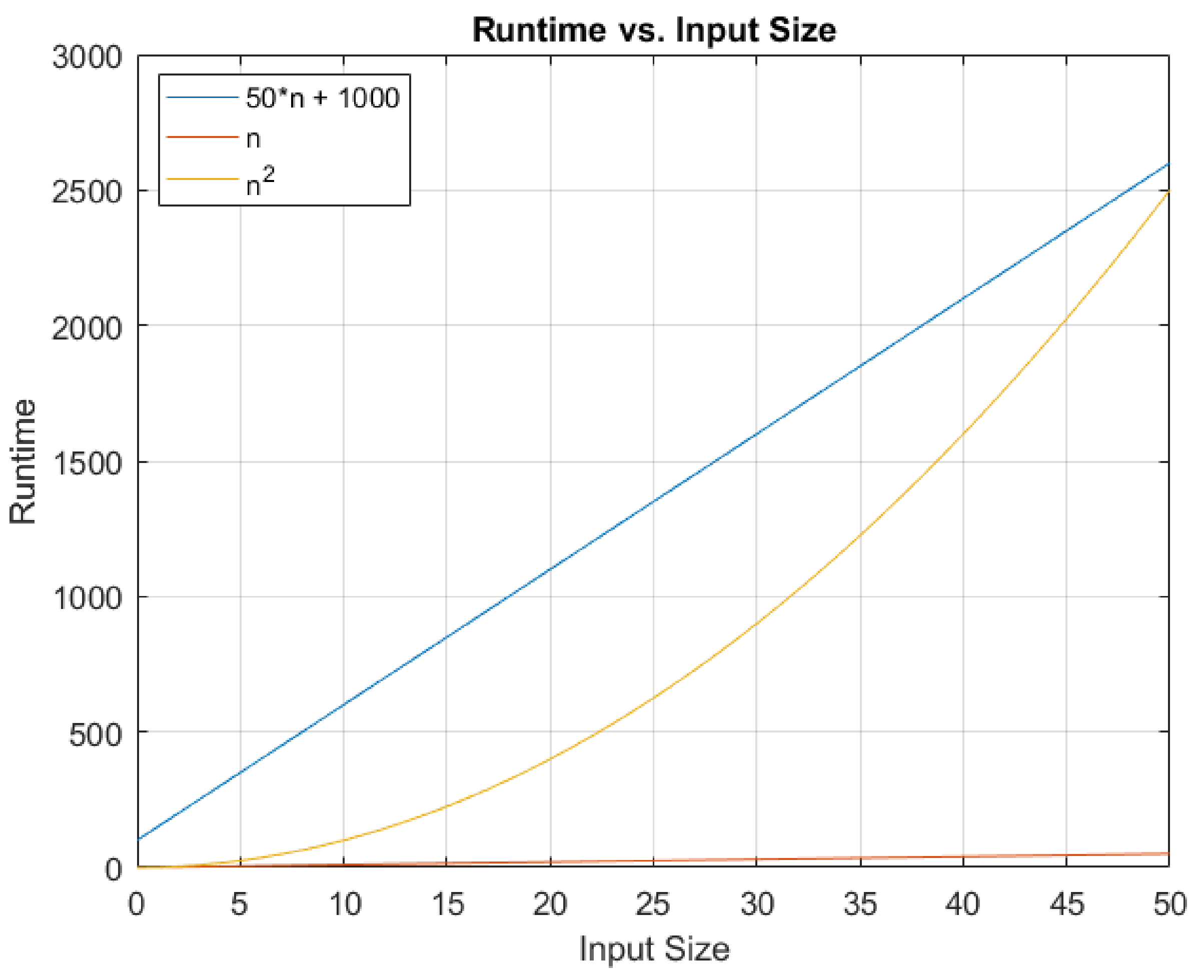
Basics of Big-O, Sorting

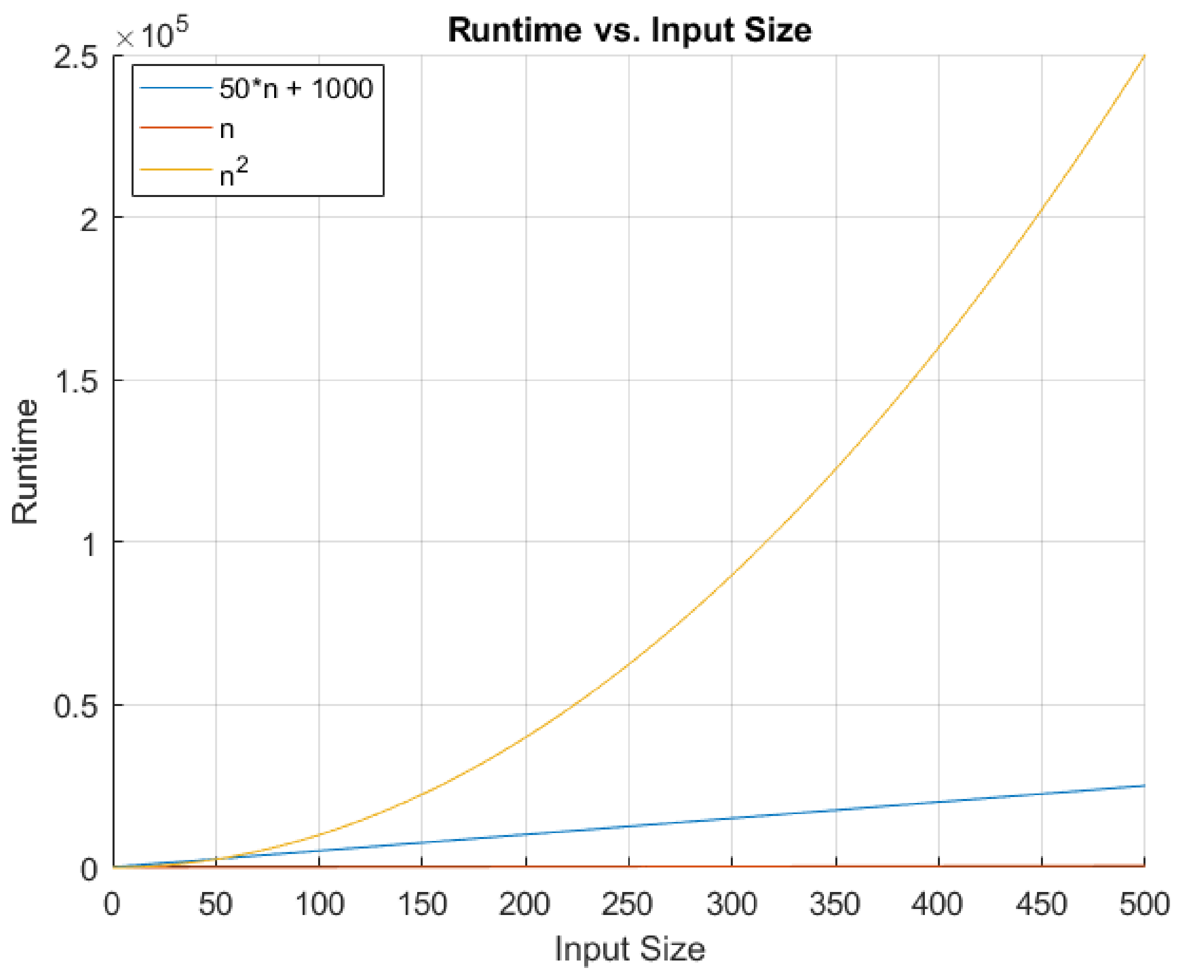
medium.com/@micah.shute/basics-of-big-o-sorting-94d0c04d0f53

What is “Big O”

It describes the asymptotic nature of a system as the input grows. Usually, we are describing runtime of an algorithm, but it can also be used to describe space complexity or even systems outside the realm of computer science. Here I will assume it describes runtime of an algorithm unless stated otherwise. Asymptotic analysis means that you focus on how the runtime of the algorithm grows as the input grows and approaches infinity. The input is usually denoted as `n`. As our datasets get larger, it is the growth function which will be the dominating factor of runtime. For example, O(n) suggests that the runtime changes linearly with the input `n`. O(n^2) suggests that the runtime changes proportionally to the size of the input squared. With large datasets, this is usually enough information to tell you which one will be faster.

When you analyze asymptotic characteristics of a function, you want to discount added constants, as well as drop coefficients and lower-order terms. For example, f(n) = 100 * n * lg(n) + n + 10000 is described as O(n * lg(n)), because as n goes to infinity, the constant and the coefficient become insignificant. Check these graphs out:


Runtime vs. Input Size
50*n + 1000
n
n^2
Runtime
Input Size
0
500
1000
1500
2000
2500
3000
0
5
10
15
20
25
30
35
40
45
50



As you can see, in small datasets, big O notation may not be telling the whole story, but even going from 50 to 500 data points in the above graph made the asymptotic nature of the functions take over.

Certainly, there are merits to optimizing code within a certain asymptotic runtime to maximize performance, but that should be done only after the proper asymptotic algorithm is being used.

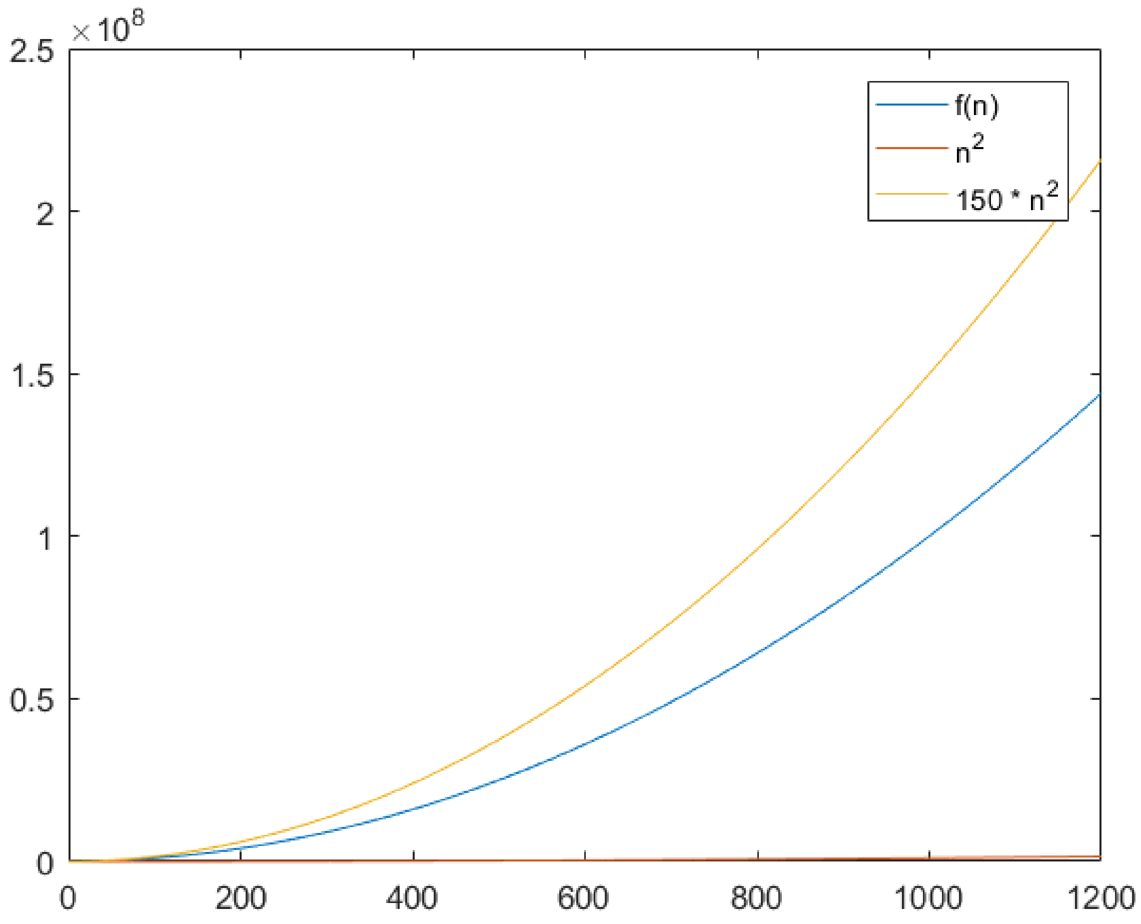
Although it seems that colloquially "big-O" is used to mean the asymptotic runtime which describes a function's 'tight asymptotic' nature, (ie describes both upper and lower bound), this isn't actually accurate. The formal definitions of asymptotic runtime are briefly described below [1]:

Big Ɵ (theta) :

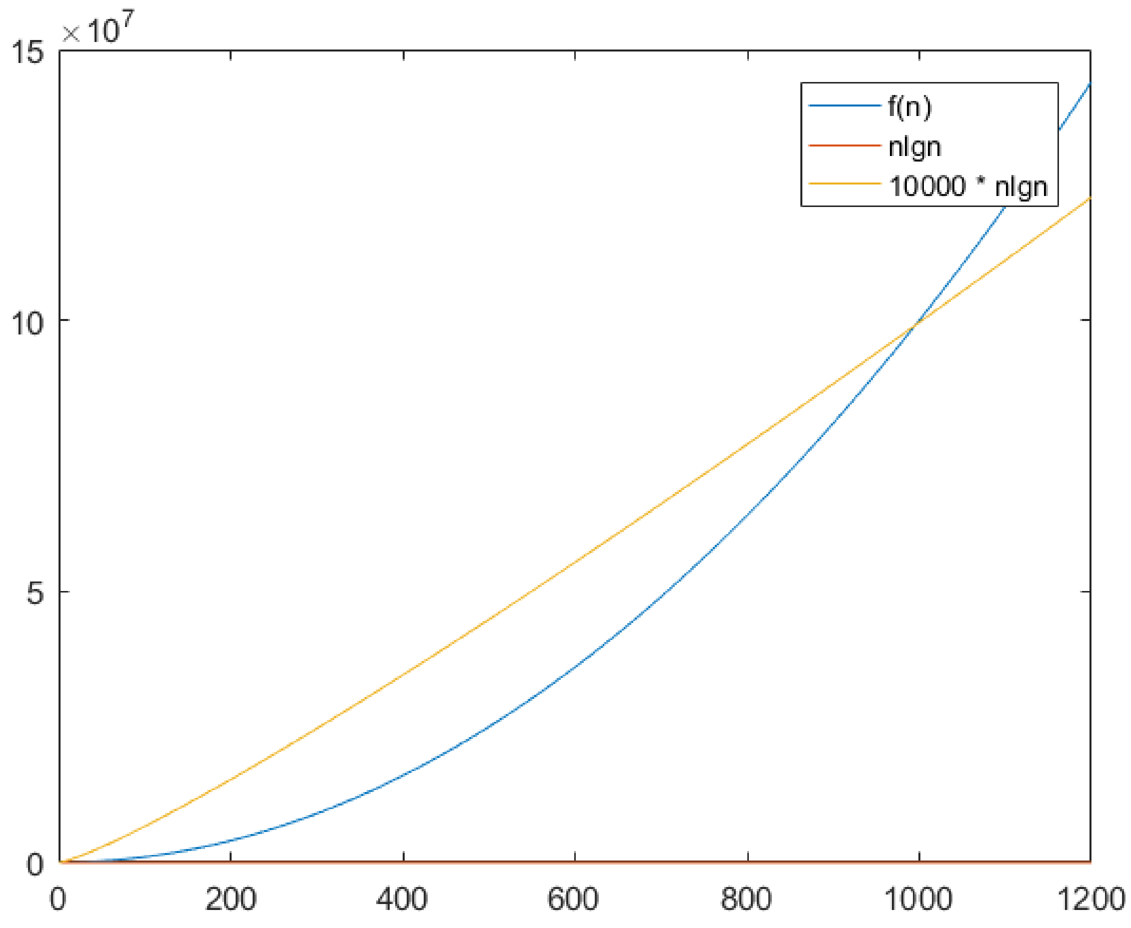
Formally, for Ɵ(g(n)) to describe a function f(n), there exist positive constants c1, c2, and n_o such that `0 <= c1 * g(n) <= f(n) <= c2 * g(n) for all n >= n_o`

To clarify: imagine you have an algorithm that is defined by a complex equation f(n). You can characterize f(n) in big-theta notation by a very simple equation g(n) which has all constants, coefficients, and lower-order terms dropped IF you can re-add 2 arbitrary constants to g(n) which can "sandwich" your original equation f(n) past a certain input data size n_o.

Example: let's say your f(n) = `100*n^2 + 4*n*lg(n)`. We can show that n^2 can meet this definition with the following equations and graph



Now, if we tried to sandwich our f(n) with nlgn, we see that the upper bound is inevitably passed by f(n) as n grows, regardless of the constant that we set:



So, technically every time I called a function big-O above, what I should have said is big-theta, ie ϴ(n) instead of O(n).

Big-O:

Big-O only provides an asymptotic upper bound as opposed to an upper and lower bound provided by ϴ notation. Formally, for O(g(n)) to describe a function f(n), there exist positive constants c and n_o such that `0 <= f(n) <= c*g(n) for all n >= n_0`

So, it is actually correct to say an algorithm defined by `f(n) = 3*n + 100` is O(n^3), even though it is ϴ(n).

There also exist other less-used notations defined similarly as the above:

- Ω (big-omega) notation defines an asymptotic lower bound only
- o (little-o) notation defines an upper bound which is NOT asymptotically tight (such as the example above where we describe a ϴ(n) function as O(n^3)—that is an asymptotically not-tight definition

and you could therefore say it is also o(n^3). o(n) would be incorrect)
- ω (little-omega) notation defines a lower bound which must NOT be asymptotically tight.

When describing algorithms, the most used concept is Ɵ notation, and it is usually communicated as O notation.

Although algorithms can be analyzed by their best-case, average-case, and worst-case runtimes, we usually (with a few exceptions below) focus on worst-case for 3 main reasons:

1. It is easier to calculate the worst-case runtime than the average runtime (fewer factors to account for)
2. A worst-case analysis is a guarantee that we will never take any longer
3. It is not uncommon that the worst-case scenario happens frequently when implemented

It turns out that lgn (shorthand for log base 2 of n) is important to a lot of runtime discussions. A quick review of logarithms:

A logarithm describes what power the base must be brought to in order to create a certain number. For example, since our base is 2, if we want to find lg(8), we ask ourselves: “To what power can I take 2 to in order to obtain the answer 8?”. In this case, since we know 2^3 is 8, we know that lg(8) = 3. You are kind of “deconstructing” an exponential. This is an important concept because if we can make an algorithm that is defined by a log, it grows super slowly! It actually grows as slowly as exponentials grow quickly, which is a lot.

Imagine you have a dataset of size 262144, and you are performing an algorithm with runtime f(n) = n on it. If the dataset grows to 524288, you are doing 262144 more operations! But if your f(n) = n^2, you are doing 68719476736 MORE operations! If your f(n) = lg(n), you are doing 1 more operation. If your f(n) = lg(n), every time your dataset doubles, you do 1 extra operation.

Why is it important?

When you work with a large dataset or a dataset which may become large as you gain users or gather more information, your runtime will be determined by its asymptotic growth. As described above, constants, coefficients, and lower-order terms have relatively minor significance compared to the highest-order component. The difference between an O(n*lg(n)) algorithm and an O(n^2) algorithm can easily be the difference between a well-performing app and an absolutely unusable app.

How do you use O-notation practically?

Understand when it is important to be mindful of your algorithms. If you are working on a small dataset and it has absolutely no way of getting big, it may not be worth your time developing super-optimized code or trying to get a better runtime for your algorithm.

If your dataset is or may get very large, it is important to understand how your algorithms are affecting your runtime (and your memory). Thus, you have to be able to tell what the asymptotic runtime is of your code. Some ways to do this:

- Look through your loops. If you are iterating through a collection once, you most likely have a Ɵ(n) algorithm. If your loop has an inner loop which iterates through all of your data (or n-1 each time), you most likely have a Ɵ(n^2) algorithm, which is very dangerous.
- Understand the “behind-the-scenes” code that occurs when you call a function which you did not write yourself or that is native to the language. There could be a better-suited way to do something, and not understanding the guts of the methods you are calling can make it very hard to optimize.
- Look through your code for functions which are linked to the size of the input data, and disregard constants or repetition of that code which is not dependent on data size. This can help you more easily determine your Ɵ(g(n))

Adopt a *divide and conquer* mindset

Divide and conquer has the following steps:

1. Divide the problem into smaller problems
2. Conquer the subproblems recursively until they are small enough to have a simple solution.
3. Combine the subproblem solutions to create the overall solution.

Basically, if you can keep dividing your problem into equal halves (which is usually the division point) until it is small enough that it is trivial (easy) to solve. Then you can solve that small, easy problem and combine those easy answers into a full answer. It is very possible you have just changed your algorithm from Ө(n^2) to Ө(nlgn) runtime.

The reason this transformation happens is that if you break down your code into equal halves, you can make that division lgn times (ie how many times can I multiply by 2 to get my data size n?). Each time you divide your problem by 2, we iterate through our data to do something to it, hence n * lgn. The sorting algorithms below will show instances of this.

Sorting

This is a simple application of big O and is something that every programmer should have a basic understanding of.

Why is it important?

Most languages have built-in sorting algorithms, but it is still important to understand how sorting works. Why?

1. **It has a lot of important use cases.** Arguably the biggest use is increasing the speed of search through an array from O(n) to O(lgn) — a huge improvement. (i.e. ability to use binary search). Also, delivering data to a user in a certain format (price from high to low) is quite common. It is also ordinary to have to manipulate data based on its location in a sorted list (i.e. display the most recent posts)
2. It is asked in technical interviews
3. Sorting algorithms can have vastly different runtimes based on the data on which they are being run. Although many given sorting algorithms work pretty well on most data, it is important to know

what different algorithms may work better for your use case. It might not be the best choice to simply use the language's standard sorting algorithm.
4. Lastly, and perhaps most importantly, it helps you get an understanding of why big O matters, and how to go about problem-solving in a way more elegant than a brute-force solution.

How can you program a sorting algorithm, with what time complexity?

Disclaimers:

These examples are focused on time complexity and space complexity will not be taken into account.

Also, better metrics for these algorithms are the number of switches and comparisons they make, not the runtime on a specific machine. For ease, intuition, and clarity I will be providing runtimes. My RAM was never fully used, and IO to disk did not factor into the runtimes.

Algorithms

Bubble Sort

Characteristics

- $\Theta(n^2)$ worst-case and average-case
- // → Notice the nested loops giving an indication of $\Theta(n^2)$ runtime
- $\Theta(n)$ best-case

Description

> Iterate through the list checking each element with its adjacent element. Swap them if they are out of order. Repeat this until all are swapped. As an optimization, you can iterate through `n` minus `i` elements each time, where `i` is the number of iterations you have already done. This is because after the first sweep you guaranteed the smallest number has "bubbled" to the top (or largest number has dropped to the bottom, depending on your iteration direction), so you only need to check the indices below (above) it.

The Wikipedia page for bubble sort shows a simpler implementation of bubble sort at first and works its way to the “optimized” version.

Even while optimized, bubble sort is extremely slow on large datasets. You can program it to `return` if no swaps are made, which means that the data is in order, but this only helps if you have a nearly perfectly sorted array. I did not include this optimization - for clarity, and because this characteristic is better utilized in insertion sort than bubble sort.

Insertion Sort

Characteristics:

- $\Theta(n^2)$ worst-case and average-case.
- $\Theta(n)$ best-case
- This best-case scenario happens when the list is already sorted (or very close to being sorted)

Description:

- Imagine your collection is broken up into two parts. The farthest index on the left (ie index 0) is a “sorted array” of one element. The rest of the array is an unsorted mess. We can go one by one in our unsorted righthand side and insert that element correctly in the lefthand side of the array in order to maintain the left side sorted and to make the sorted side grow by one. By the time that we get to the end of the array, the whole lefthand side (ie the entire array) is sorted.
- This is $\Theta(n^2)$ because, for each index in our unsorted side, we have to search through the sorted side for its correct location. Not only that, but we have to shift every element over on the way to make room for our insertion, which is also costly.
- // → It is important to note both the number of comparisons and the number of swaps.
- // → For every index in the righthand side of the array, we are looking through 1 up to `n` other elements in the lefthand side for the correct insertion position. The summation of 1 to n is of order Θ(n^2)

We can evaluate this algorithm via a “loop invariant”:

A loop invariant is a way to prove the correctness of an algorithm which performs repetitive tasks. To do this correctly, we have to show a *loop invariant (LI) statement* ***correct*** for:

- Initialization — your LI statement is true before the first time you loop
- Maintenance — your LI statement is true throughout the duration of each subsequent loop
- Termination — your LI statement is at the end of the program and gives a way to ‘prove’ your algorithm is correct.

In this case, a loop invariant can be stated (paraphrased from [1]): “At the start of each iteration of our for loop, the subarray from 0 to j-1 (inclusive) consists of the elements originally in that range of indices but now in sorted order”

- **Initialization**: Because the lefthand side is a single element (ie 0 to j=1–1=0 inclusive), it is sorted by definition.
- **Maintenance**: Each time you loop, you insert an item from the unsorted side to the correct position on the sorted side. Then you update your indices so you know where that divide is. Your sorted side is bigger by one and still sorted. Before the loop, and after the loop, the subarray 0 to j-1 is sorted, but at the end of the loop, j is one unit larger than before.
- **Termination**: Your loop is over when j is equal to your array length minus 1. So, we know that the array from 0 to (array length — 1) inclusive is sorted. That is every index in our array. Our whole array is sorted.

As you will see below, although most of the time Insertion Sort is very slow, it is a rockstar at sorting almost-sorted data.

Merge Sort

Characteristics

Ɵ(n*lgn) best case, worst case, and average case

Description

Divide and conquer algorithm:

- **Divide** the problem by half until the array size is 1
- **Conquer** the sorting problem when the size equals one—its already done for you because there is only one element. This will give you your first return from the `merge_sort` function instead of a recursion. This return will always be a sorted array (either of length 1 or `merge`d into a sorted array) and allows you to proceed with the algorithm by now just preserving the 'sorted' status of these arrays as you `combine` them.
- **Combine** your two smaller halves which are both of size 1 (and sorted)together by comparison to make a sorted array of length 2.
- // → One level up the recursion will now have 2 arrays of size 2 which **are sorted**. ***Combine*** them together to create a new sorted array of length 4, and so on until your entire array is sorted.

Merge Loop Invariant [1], (references the code below):

- **Statement:** At the start of the while loop, `merged_arr` will contain the smallest (i + j) elements from the arrays `left` and `right`, in sorted order. `left[i]` and `right[j]` will contain the smallest numbers in their respective arrays for indices greater than i and j respectively.
- **Initialization**: `merged_arr` is empty, and i + j = 0. `left` and `right` are sorted arrays, so both of their 0th elements will be the smallest in their entire arrays.
- **Maintenance**: The smaller of the two numbers from `left[i]` and `right[j]` is appended to the end of merged_arr. Since both arrays are in sorted order and i and j are both incremented from 0, we are guaranteed that each iteration of the while loop only considers numbers to append to `merged_arr` which are larger or equal to the numbers already appended to `merged_arr`, and smaller than or equal to the numbers of higher index remaining in `left` and `right`. This ensures the sorted condition of `merged_arr`. When an element is appended to `merged_arr` from `left`, only `i` is incremented by 1, and when an element is appended to `merged_arr` from `right` only `j`

is incremented by 1. This ensures that `merged_arr.length` is equal to i + j, that `merged_arr` 's values are in sorted order, and the next two numbers `left[i]` and `right[j]` are greater than or equal to all of the numbers in `merged_arr`, and less than or equal to all other values left in their respective arrays. IE, they are the two smallest values their arrays have to offer which have not already been put in `merge_arr`.
- **Termination**: The loop breaks when one of the arrays is 'empty'. We know that at this point `merged_arr` is in sorted order, and has all the values of the expended array and all the values of the other array up to the index of `i` or `j`. This termination is great but requires just a little cleanup. You have to now push the remainder of the array with values left over onto the end of `merge_arr`. You know you can just push them at the end, because the array with leftover values is already sorted, and its lowest valued number was determined greater than or equal to the largest value in the other array.

Note that sometimes two "sentinels" are placed at the end of `left` and `right` and given the value of infinity. Then, the while loop is replaced by a for loop which runs `left.length + right.length - 2` times, so that only the two infinity sentinel values are left. In Ruby, you can represent infinity with `Float::INFINITY`, in Python `float("inf")`, in Javascript `Infinity`, and in Java `Double.POSITIVE_INFINITY`.

Code (uncomment to see it in action—this is helpful to see how the sorting works)

Quick Sort

Characteristics

- Θ(n*lgn) best case, average case
- Θ(n^2) worst case (when nearly sorted)
- // → If you look at the code below, you can see that this is because the pivot, in the sorted case, splits the problem into two sizes n -1 and 1, instead of n/2 and n/2. For a randomized array, it is more likely that the pivot will be a value closer to the median value. This

will be described more below.

Description

Divide and Conquer

- **Divide** Choose a ‘pivot point’ as the last index in the array. Iterate through the array and switch values which are on the ‘wrong side’ of the pivot value until all values larger or equal than the pivot are on the right side of some index, and all the smaller values are on the left side of that same index. Divide the array by that index to create two smaller subproblems.
- **Conquer** When the sizes are small enough (ie 1 or 2), this **divide** procedure will cuase the subsets to be sorted. Grouping the numbers by greater than or less than a pivot on a large scale ensures that the algorithm done on a smaller scale array of 2 (which actually causes sorting of the array) is done at a location in the array that will cause the overall array to be sorted, and not just have clumps of localized sorted data.
- **Combine** Just like the **Conquer** of the Mergesort algorithm was trivial because all of the work was done in the Mergesort **Combine**, the **Combine** of quicksort is trivial because all of the work is done in the **Divide** step. After the array is divided down to sets of 1 and 2, it is sorted in place. And the algorithm will stop recursing when the size of the subarray become equal to or less than zero.

Partition Loop Invariant (see code below)

- **Statement:** Values of `arr` from the original argument value `lo` to the current value of `lo` are all less than or equal to the pivot value. Values of `arr` from the original argument value `hi` to the current value of `hi` are greater than or equal to the pivot value.
- **Initialization** `lo` and `hi` are initialized to be one lower and greater than their initial values, respectively. Being outside of the array and not associated with values, you trivially meet the requirement that the related values of `arr` are as stated in the LI.
- **Maintenance** After a single loop, the `lo` index value is incremented

up past any indices whose value are smaller than the pivot. When the loop finally breaks, `lo` is therefore guaranteed to be fixed at an index whose value is greater than or equal to the pivot value, with all values to the left of it guaranteed to be less than the pivot value (or equal to the pivot value if it was swapped to the left of `lo`). 'hi' is decremented in the same manner, stopping on an index whose value is less than or equal to the pivot value, guaranteeing that all values to the right of 'hi' are greater than the pivot value (or equal to the pivot value if it was swapped to the right of `hi`). If `hi` > `lo`, we swap them because we know that the inner loops stopped on values which should be associated with 'the other side' or the 'middle' of an array divided by the pivot value, and we have not yet reached the 'middle'. This means that the new value for `arr[lo]` must meet the requirement for the `arr[hi]` loop to break (ie `<= pivot`), and the new value for `arr[hi]` must meet the requirement for the `arr[lo]` loop to break (ie `>= pivot`). So, it is maintained that all values at index `lo` and below are either less than the pivot (if they passed over by the loop) or less than or equal to the pivot value (if they were swapped from a point previously at an index `hi`). In other words, they are all less than or equal to the pivot value. Conversely, it is maintained that all values at index 'hi' and greater are either greater than the pivot value or equal to the pivot value.
- **Termination** If `lo` >= `hi`, we know we have hit the place in the array which is divided numerically by the pivot value. We return the higher of the two indices, allowing us to create a subdivision of the array in which all values below the `partition_index - 1` are less than or equal to the pivot and all values above the `partition` are greater than or equal to the pivot. Also, by subdividing the array the way that we do in `quicksort!` we prevent an infinite loop by preventing a partition index to subdivide into an empty array and the original array. When the subarray is of size 1 or 2, `partition` results in a sorted subarray section, which is also in its correct position relative to the entire array.

Code

Algorithm runtime on randomized and nearly-sorted data

Random Numbers Output:

It is a very common use case to have to sort a collection which is already mostly sorted. Let’s see how the algorithms do with that:

If you know you need to frequently sort a collection which is very nearly already sorted and you want to have the fastest possible sort time, a simple Insertion Sort algorithm actually beats out the fully-optimized built-in Ruby #sort, in this case by a factor of about 3.

QuickSort: Avoiding Worst Case

When I attempted to run QuickSort on the mostly-sorted list, it crashed the program until I brought the dataset size all the way down from 1 million to about 15,000.

At this point, the runtime for these 15,000 sorted elements was the same as QuickSort running on 1 MILLION randomized elements, and while the number of quicksort recursions was smaller, the number of loops which occurred in the partition method was drastically larger.

As described above, this happens because the pivot point is at the end of the array, so every single other element in `arr` is less than the pivot point! Instead of making two even subproblems to achieve the nlgn runtime, we are creating 2 subproblems of size 1 and n-1. The summation of 1 to n is ϴ(n^2). Think about the summation of 1 to 10. It is `(10 + 1) * (10 / 2) = 55`. Replacing 10 with n, `(n + 1) * (n/2) = (n^2 )/2 + n/2`. Drop coefficients and lower order terms to get `n^2`.

To solve this problem of ϴ(n^2) runtime for a nearly sorted list, we can be smart about how we choose a pivot element. If I run optimize_pivot to choose my pivot point, I find the median value between the values of the low, hi, and med indices in the subarray. I then switch that value with `arr[hi]` and continue to use `hi` as my pivot index. If the array is already sorted, I am now using the median value of the list rather than an extreme and can achieve Ө(n * lgn) runtime on a sorted list because I am back to breaking my problem in halves rather than subtracting one from the size.

Better QuickSort Code

Here is a second performance metric of all the algorithms including the optimized quicksort

Resources

[1] *Introduction to Algorithms*, Cormen, Leiserson, Rivest, Stein