Exploration: Searching, sorting, algorithm analysis

Searching

Searching a list for a particular item is a very common computational task. The simplest approach to this task is to just start at the first element of the list and compare it to the target value. If they’re not equal, move to the next element of the list and compare again. Keep doing this until either the target value is found, or we reach the end of the list without finding it. This algorithm is called **linear search**. In Python it looks like this:

Graphical user interface, text, application

Description automatically generated

Linear search is a very simple, straightforward algorithm, but how efficient is it? The worst case is if the target value isn’t in the list. In that case we must compare the target with every element in the list, so for a list of *n* elements, we must perform *n* comparisons. Therefore, the **time complexity** of linear search is O(n). This means that the number of comparisons we must perform (and hence the time needed to run the program) increases at a rate proportional to *n*. If *n* is small, that’s not so bad, but if it’s large, then it would be awfully convenient if we had a more efficient algorithm.

Fortunately, there is a more efficient algorithm for searching a list, called **binary search**. It’s still pretty easy to understand, but the algorithm is slightly more complicated than for linear search. It also makes an additional assumption, which is that the elements of the list we’re searching are in sorted order. If you’re looking for the name “Daenerys Targaryen” in the Westeros phone book, you wouldn’t start at the first name and slowly work your way through in a linear search. Instead, you might flip it open to the middle and see where you’re at. If you’re at “Targaryen”, then great, you’re done, but what if you weren’t that lucky? If you opened to something that’s alphabetically after “Targaryen”, then you know to look in the first half. If you opened to something that’s alphabetically before “Targaryen”, then you know to look in the second half. Now that you know what half of the phone book it’s in, you don’t even have to consider the other half. Now you can repeat the same process with the half that you know the name is in. Flip it open to the middle of that half, see if you found it yet, or if not, see which half of that half you need to look at next. Proceed like this until you either find the name or find the place where the name should be if it were in the list.

Graphical user interface, text, application

Description automatically generated

This algorithm works by finding the midpoint of the list and comparing it to the target. If it matches, then we’re done. Otherwise, if we should look in the first half, it updates *last* to be the end of the first half (*first* is already the beginning of the first half). But if we should look in the second half, it updates *first* to be the beginning of the second half (*last* is already the end of the second half). Then we go back to the top of the loop to find the midpoint of the section we’re looking at now. This continues until we either find the target value or there are no values between first and last, meaning that the target is not in the list.

How many comparisons do we have to perform in the worst case? If the target isn’t in the list, we keep going until we reach an empty sub-list. Since we’re ruling out half of the remaining list at each iteration of the loop, that will happen after log2n comparisons, so the time complexity of binary search is O(log n). We’ll see later why I omit the base of the logarithm in that notation. When n is large, log n will be much smaller than n, so for a long list, binary search will be much faster than linear search, **if** the list is sorted.

Sorting

Sorting a list is another very common computational task. Let’s look at a couple of simple sorting algorithms. **Bubble sort** starts by comparing the first two values in the list. If they’re in the wrong order, it swaps them, otherwise it leaves them where they are. Next it compares the second and third values and does the same thing. It proceeds in this fashion till it reaches the end of the list. At this point, the list is probably not sorted, but we can guarantee that the largest value is now at the end of the list. Next, bubble sort makes another pass through the list (except the last value, which we know is in the correct place). At the end of this pass, we know that the second-largest value is now in its correct place. At the end of *i* passes, *i* values will be in their correct places. Bubble sort therefore makes n-1 passes (if all but one of the values are in the correct place, then the last value must also be in the right place). In Python, that looks like this:

Graphical user interface, text, application

Description automatically generated

What is the time complexity for bubble sort? We would say that bubble sort is O(n2). What does that mean? It means we need approximately n2 comparisons to sort a list of length n, so the amount of time needed to sort a list using bubble sort increases pretty quickly as the size of the list increases. Soon you'll see how we calculate the time complexity.

Let’s look at another simple sort called **insertion sort**. This sort works by first pulling the second value up out of the list, then looking at the numbers to the left of the “vacant” spot and “sliding over” all of the values larger than the one we pulled out. Then it drops the value that was pulled out into whatever spot is now “vacant”. At this point, the first two values are in sorted order. Next it pulls the third value up out of the list and repeats what it did for the second value, inserting it into its correct position in the sorted list that is being built out from the left side of the array. It proceeds this way until every value has been inserted in its correct position.

Graphical user interface, text, application

Description automatically generated

After the *i*thvalue has been inserted, we know that the leftmost *i*+1 values are in sorted order. In the worst case, the list starts out in reverse sorted order, so that all of the elements to the left of the “vacant” spot must be compared and shifted over. The first time through the list requires 1 comparison, the second time requires 2 comparisons, and so on. The last time through the list requires n-1 comparisons. Therefore in the worst case, insertion sort will require ∑�=1�−1� = 12(�2−�) comparisons\*. That's close enough to n2 that we can say insertion sort has O(n2) time complexity (we'll define what "close enough" means below). However, although bubble sort and insertion sort have the same worst-case time complexity, insertion sort is usually much faster. Since lists don’t usually fall into insertion sort’s worst case, its inner loop doesn’t usually have to perform so many comparisons.

\*You may not have seen summations for a while. The summation above (the left side of the equation) just means that we are summing together all values from 1 to n-1 to get the total number of comparisons. The right side of the equation is what that summation reduces to. You can review summation notation at [Khan Academy - Summation NotationLinks to an external site.](https://www.khanacademy.org/math/ap-calculus-ab/ab-integration-new/ab-6-3/a/review-summation-notation).

Check out the animations of bubble sort and insertion sort at [Comparison Sorting AlgorithmsLinks to an external site.](https://www.cs.usfca.edu/~galles/visualization/ComparisonSort.html) (University of San Francisco). You can adjust the speed. Also see a direct matchup of the two [hereLinks to an external site.](https://www.youtube.com/watch?v=TZRWRjq2CAg" \t "_blank). Watching how the elements get moved around helps make the behavior of these algorithms more intuitive (for a more whimsical illustration, you can see them interpreted as dances [hereLinks to an external site.](https://www.youtube.com/user/AlgoRythmics/videos" \t "_blank)). It can also be very instructive to step through the code for the search and sort algorithms using the [PyCharm debuggerLinks to an external site.](https://www.jetbrains.com/help/pycharm/part-1-debugging-python-code.html).

Algorithm analysis

In order to compare the efficiency of algorithms, we first need to decide what our criteria are. Are we concerned with using up the least amount of time, or using up the least amount of space (memory)? Most often we’re concerned with time complexity, but space complexity can also be an important consideration, not least because there's often a tradeoff between the amount of space needed and the amount of time needed. We also need to decide what operations we’re counting. For the example of searching and sorting algorithms we looked at, we counted the number of comparisons. Usually it's whatever step the algorithm is executing most frequently.

Time complexity doesn’t tell you how long it will take to run the algorithm – that depends on the machine being used. **Time complexity is a machine-independent way to express how quickly the amount of time required increases as the problem size increases.** I think it’s easiest to see this by looking at graphs of the curves for different functions, where the x-axis shows the size of the input and the y-axis shows how many operations (comparisons, for example) are required:

Chart, line chart

Description automatically generated

You can see that as the size of the list gets larger, you would rather be using binary search, which is O(log n) than linear search, which is O(n).

Big-O notation

Text, application

Description automatically generatedTime complexity is a rough guide. Even if two algorithms have the same time complexity one can still be significantly more efficient than the other, as with insertion sort versus bubble sort. And even if one algorithm has a higher time complexity than another, it’s possible that it could still be faster for low input sizes. Despite these caveats, it's still very useful for comparing the efficiency of different algorithms.

Occasionally there’s an algorithm for which the worst case is rare, and its usual performance is much better than its worst-case time complexity would suggest. For such an algorithm it can make sense to consider its average-case complexity. The classic example of this is a sorting algorithm called quicksort, which in the worst case is O(n2), but on average is O(n log n). However, calculating the average-case complexity is usually less straightforward.

If the time required remains constant as the problem size increases, we call that "constant time" or O(1).

There are several sorting algorithms that have a time complexity of O(n log n), which is faster than O(n2). You’ll see one or two of these in CS 261.

Exercises

1. We know that binary search requires that the list be sorted. If you have an unsorted list, would it be more efficient to use linear search, or to sort the list and then use binary search? What factors might affect your answer?

2. Modify the insertion\_sort function to sort a list in **descending** order. The unit test file can be downloaded from [here:](https://canvas.oregonstate.edu/courses/1915078/files/98541892?wrap=1)[Download here:](https://canvas.oregonstate.edu/courses/1915078/files/98541892/download?download_frd=1)[Searching 2.py](https://canvas.oregonstate.edu/courses/1915078/files/98541892?wrap=1)[Download Searching 2.py](https://canvas.oregonstate.edu/courses/1915078/files/98541892/download?download_frd=1)

3. Is 2n - 4n in O(n2)?