Prereq Review #4: Binary Search

Ian Tullis, CS161 Summer 2022

Binary search is a critical tool in algorithm design, and it often helps us cut a factor of n down to a factor of $\log n$. I want to stress that this is a **huge** difference that is easy to miss when just looking at the notation. Suppose we mean log base 2 – then let's compare n and $\log n$. Say $n = 4294967296$, i.e., 2^{32}. Then $\log n$ is a mere 32. So the difference between n^2 and $n \log n$ can be enormous!

I think our minds naturally try to think of $O(n \log n)$ algorithms as being somewhere "midway between" $O(n)$ and $O(n^2)$ algorithms. But this is far from the case – remember that any polynomial grows faster even than any *power* of a logarithm. $n \log_2 n$ is actually $O(n^{1.000000001})$, for example! Even $n \log_2^{1000} n$ is actually $O(n^{1.000000001})$.

I want you to know when you can apply binary search; there are multiple CS161 homework problems that require that. I also want you to know how to implement it – in my experience, it's incredibly easy to mess that up, and there will be one coding homework problem that requires implementing it correctly.

What Is Binary Search? When Can We Use It?

In its narrowest sense, binary search refers to searching a list to see if a value is present. But here we will be talking about a broader class of problems that can be solved using the same.

In this broader sense, binary search works when you have a finite range $[a, b]$ or list of values, such that when you look at them from smallest to largest:

- For all values in the range $[a, n]$, a certain property P is true.
- For all values in the range $(n, b]$ and no greater than b, P is false.[1]

Or it can be vice versa – P is false up to a certain point, then becomes true. But the important part is that there is just one switching point, and the purpose of binary search is to find it. (Technically, I should say *at most* one – it is possible that there is no switching point, and we want to discover that.)

Let's consider a concrete example. As a kid, did you ever play that guessing game where your friend thinks of an integer n between, say, 1 and 1000, and then you get to ask up to 10 questions of the form "Is your number (smaller/bigger) than (some value)"? Binary search absolutely *destroys* this game, and we can use it because:

[1]If you have not seen this notation before, (and) denote exclusive endpoints of a range, and [and] denote inclusive endpoints. So if we are only talking about integers, for example, then $[2, 4)$ would contain only the values 2 and 3. If we are talking about real numbers, $[2, 4)$ contains every value that is both no less than 2 and strictly less than 4.

- For values $1 \leq v \leq n$, our friend will answer **No** to "Is your number smaller than v?"
- For values $v > n$, our friend will answer **Yes** to "Is your number smaller than v?"

Because of this, we can do the following:

- Start by asking if the number is smaller than 501.
- If our friend says No, we know that $n \geq 501$, so the number is in the range $[501, 1000]$. So we can restrict all future guesses to that range.
- If our friend says Yes, we know the number is in the range $[1, 500]$. So we can restrict all future guesses to that range.

Notice that in either situation, we are able to throw away half of the list, halving the size of the range we need to search! And then we can recursively do the same thing – e.g., if our friend says that yes, the number is smaller than 501, we can next ask: is the number smaller than 251? Once again, no matter what our friend answers, we will have cut the range in half again – we will now be searching either $[1, 250]$ or $[251, 500]$. And so on, until we constrict the range to include only a single number, which is surely our friend's number. (We're glossing over some details here, which will come back to bite us! But let's be handwavy for now.)

Whoops! This should say log_2(2^r) = r, not log_2(r)

In the ideal case in which the range size is a power of 2, i.e., 2^r for some nonnegative integer r, we have to halve 2^r a total of $\log_2 r$ times to reduce it to 1. (This is in some sense the definition of "log base 2".) We will not always necessarily be able to perfectly halve the range, but in the worst case, we can certainly split it such that one part of the range is only one smaller than the other part, so it's still nearly half and half. But a range of size no greater than 2^r takes no more than $\log_2 r$ rounds to search – for example, we could pad one end of the range with values larger than the largest value in the range, bringing the total length to exactly 2^r, before beginning our search.

Binary search works on lists as well as ranges! Suppose we have a **sorted** list of integers, and we want to know how many of the values in the list are greater than 100. There is no reason to walk through the entire list checking every value, or even checking every value in order until we find the smallest value greater than 100 and then reasoning that every other value is also greater than 100. We can instead use binary search: look at a value around the middle index of the list (e.g., the 500th element of a 1000-element list) and see whether it is greater than 100. If it is, we know we should search the lower half of the list. Otherwise, we search the upper half. And so on.

When Can We Not Use Binary Search?

Suppose we want to answer the question "What is the largest prime number in the range $[1, 1000]$?" Binary search will not help us here – we can't look at 500 and see that it is not prime, then check 750 and see that it is not prime, etc. This is because there is more than one point in the range $[1, 1000]$ at which values switch from being prime to not prime or vice-versa. So the whole idea of "well then the value we want must certainly be greater than / less than this value" falls apart. Knowing that 500 is not prime tells us nothing about whether 250 or 750 is prime. (Neither is!)

Binary Search From Scratch: A Tragedy In Many Acts

Pixelated Boat
@pixelatedboat

The whole internet loves binary search, a lovely algorithm that seems easy to code! *5 seconds later* We regret to inform you the way you coded it loops forever and is also wrong

Retweets 10,146 Likes 24,084

1:07 AM - 12 Jun 2016

81 10K 24K

Suppose we have a sorted list of n integers. The list contains every integer between 0 and n (inclusive) exactly once, *except* for one integer that is missing. **How do we find the missing integer?** E.g., for $[1, 2]$, it's 0. For $[0, 1]$, it's 2. For $[0, 2]$, it's 1.

This is a great opportunity for binary search, although it may not be immediately obvious why: for all numbers in the list less than the missing number, they are at the list index matching their value. For all numbers in the list greater than the missing number, they are at the list index one more than their value. That is, suppose that 2 is the missing number; then 0 is at index 0 (we use 0-based indexing because we'll be coding soon) and 1 is at index 1, but 3 is at index 2, 4 is at index 3, etc.

Consider this solution that tries to find the point at which there is a discontinuity – a jump of more than 1 between two successive elements. Before turning the page, think about it. Looks pretty much like the strategy we were describing earlier, right?

```
def solve(ls):
    lo_index = 0
    hi_index = len(ls) - 1
    while True:
        mid_index = (lo_index + hi_index) // 2
        if ls[mid_index] + 1 != ls[mid_index + 1]:
            return ls[mid_index] + 1
        elif ls[mid_index] > mid_index:
            hi_index = mid_index
        else:
            lo_index = mid_index
```

Unfortunately, this is quite buggy! Here's its performance on some lists:

```
[1]: crashes
[0]: crashes
[1, 2]: runs forever
[0, 2]: correct
[0, 1]: runs forever
[1, 2, 3]: runs forever
[0, 2, 3]: correct
[0, 1, 3]: correct
[0, 1, 2]: runs forever
```

First of all, we have a crashing problem when the list is too small. We could just special-case the length-1 situation: if our list contains just 0, we are missing 1, and vice versa.

Second, the algorithm sometimes runs forever. Inspecting the above results, we see that this happens when the missing value is either 0 or n. When the missing value is 0, we are trying to find a discontinuity – two consecutive elements that are 1 apart – in a list like $[1, 2, 3]$ in which no such discontinuity exists. Similiarly, when the missing value is n, we are trying to find a discontinuity in a list like $[0, 1, 2]$.

One way to try fix this (and also fix the length-1 issue even without any special-casing) is to pad the list with two values that will surely create discontinuities. That is, we put -1 on the left (since the smallest possible missing value in the list is 0), and $n + 1$ on the right (since the largest possible missing value in the list is n).

```
def solve(ls):
    ls = [-1] + ls + [len(ls)+1]
    ...
```

But padding a list is potentially expensive – if it's an array, the whole existing array must be moved and/or resized. And this solution still doesn't fix everything:

```
[1]: runs forever
[0]: correct
[1, 2]: runs forever
[0, 2]: correct
[0, 1]: correct
[1, 2, 3]: runs forever
[0, 2, 3]: runs forever
[0, 1, 3]: correct
[0, 1, 2]: correct
```

What a mess! Let's scrap this approach and go back to basics: we are trying to identify the point at which we switch from having a property (each element of the list equals its index) to not having the property. Let's write the binary search in another way, where we try to contract the range until contains just a single value. Here's an attempt:

```
def solve(ls):
    lo_index = 0
    hi_index = len(ls) - 1
    while lo_index != hi_index:
        mid_index = (lo_index + hi_index) // 2
        if ls[mid_index] == mid_index:
            lo_index = mid_index
        else:
            hi_index = mid_index
    return lo_index + 1
```

We've exchanged one set of problems for another set of problems! Worse yet, now we're sometimes **wrong**.

```
[1]: wrong answer
[0]: correct
[1, 2]: wrong answer
[0, 2]: runs forever
[0, 1]: runs forever
[1, 2, 3]: wrong answer
[0, 2, 3]: runs forever
[0, 1, 3]: runs forever
[0, 1, 2]: runs forever
```

Now what's the issue? Observe that when we have, e.g., `lo_index` = 1, `hi_index` = 2, then `mid_index` just ends up being 1 again, because integer division floors the result. So if we are in the situation in which `lo_index` really is the largest index for which `ls[i] == i`, we keep setting `lo_index` to the value it already is, and so we end up looping forever!

So OK, here's a possible fix. We want to check whether `lo_index` is one smaller than `hi_index`, not whether they are equal.

```
def solve(ls):
    lo_index = 0
    hi_index = len(ls) - 1
    while lo_index + 1 != hi_index:
        mid_index = (lo_index + hi_index) // 2
        if ls[mid_index] == mid_index:
            lo_index = mid_index
        else:
            hi_index = mid_index
    return lo_index + 1
```

Welcome to the next page. I have some bad news.

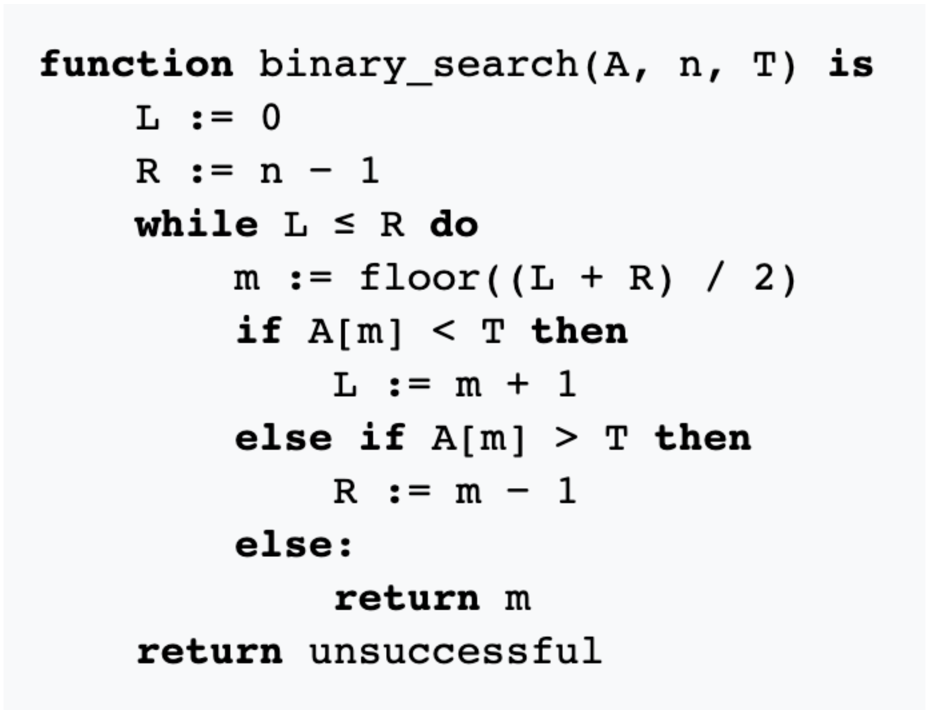
```
[1]: runs forever
[0]: runs forever
[1, 2]: wrong answer
[0, 2]: correct
[0, 1]: runs forever
[1, 2, 3]: runs forever
[0, 2, 3]: correct
[0, 1, 3]: correct
[0, 1, 2]: wrong answer
```

At least we seem to be correctly handling cases in which the missing value is not at the end of the list. What if we special-case the ends?

```
def solve(ls):
    n = len(ls)
    if ls[0] == 1:
        return 0
    elif ls[-1] == n-1:
        return n
    lo_index = 0
    hi_index = n - 1
    while lo_index + 1 != hi_index:
        mid_index = (lo_index + hi_index) // 2
        if ls[mid_index] == mid_index:
            lo_index = mid_index
        else:
            hi_index = mid_index
    return lo_index + 1
```

Now this passes all the cases.

If this does not leave you raring to implement binary search, I understand completely. The good news is that when we are searching for a target value instead of a discontinuity, as in the narrower sense of “binary search”, the algorithm is more elegant, as seen here from Wikipedia:



```
function binary_search(A, n, T) is
    L := 0
    R := n - 1
    while L ≤ R do
        m := floor((L + R) / 2)
        if A[m] < T then
            L := m + 1
        else if A[m] > T then
            R := m - 1
        else:
            return m
    return unsuccessful
```

This approach gets around the floor-division issue by ensuring that the value of either the low index or the high index always changes. It is not obvious how to adapt it to our missing-number problem, though. (If you come up with a good way, I'd love to include it here!)

The main points I want to get across are:

- Although the idea behind binary search is not too hard to grasp, implementing a correct binary search from scratch is surprisingly hard.
- When setting a midpoint as an average of a low value and a high value, beware of how integer division works. You might get an infinite loop if you always expect this to change one of the two values.
- Make sure your algorithm correctly handles cases in which the target value is at the beginning or end of the list – or, when searching for a "switching point" / discontinuity – cases in which that occurs at the beginning or end of the list, or does not occur at all. Special-casing these situations might be easier than trying to write something more elegant that naturally gets them right.
- Make sure your algorithm correctly handles cases in which the list is one or two elements long. Testing on lists of size 3 might also be helpful.