An Introduction to Binary Search

Michael You

1 Why do we care about search?

When we think of "search," some examples that might pop into your head are

- Google search
- Phone directory
- Yelp restaurants

We associate search with **finding things**, and that's pretty much what it is.
In terms of some things we care about search:

- We usually want to have **accurate results**. Meaning if I want to know if something exists in a database, then I should be able to get a yes or no answer (and not something like "idk").
- We want to have the result **fast**. It's annoying to wait a long time for someone to find something, and it's the same for search engines or other services that are supposed to help you find things.

2 Search, formally

To help formalize this notion, we can actually break down search as the following.

Definition 1. A **search** can be defined as consisting of

- **Target**: What are we looking for? E.g. are we looking for a website, a video, a person?
- **Search space**: What is the set of things we are considering? Is it a database, is it the numbers 0 to 9, is it anything?

And the problem of searching is, given a **target** and a **search space**, we

- Return `true` if **target** $\in$ **search space**
- Return `false` otherwise

To get an idea of what search problems are like, let's consider the problem of looking for a word in a dictionary.

Example 2.1. Suppose we have a dictionary, and we want to find the word "elephant". How do we go about doing this?

- What if we had a newspaper instead, how would we find words there?

Example 2.2. Suppose we have a sorted array of integers. We want to find some target number t. How do we go about doing this?

- Suppose our array looks like

```
[-6, 1, 4, 5, 6, 8, 11, 19, 21]
```

- Find 4
- Find 11
- Find 7

When we are dealing with a sorted array, we notice that if we are currently at some element e, if our target t is

- $t < e$, then we know every number above e does not need to be considered anymore, because any $e' > e > t$
- $t > e$, similarly, if this is the case, then we can discard all elements $e' \leq e$

With this in mind, we can now motivate the idea of **binary search**.

3 Binary Search

The intuition goes like this

1. What is a good starting point, so we can discard the most elements?
2. After we eliminate some elements, what is the next best starting point?
3. When do we stop?

A natural question that might pop up is, why do we choose the halfway point? Does it really matter what we choose? Another question that might pop up is, can we trisect, quadsect, instead of just bisecting?

To quell these concerns, we can actually rigorously investigate these ideas with math.

Problem 3.1. What happens if we don't choose the halfway point?

Suppose we define $f(n)$ to be the expected number of searches with n elements. We are basically asking ourselves, what is the optimal definition of

$$\begin{aligned} f(n) &= 1 + P(t \text{ in first part}) \cdot f(\text{first part}) + P(t \text{ in second part}) \cdot f(\text{second part}) \\ &= 1 + a \cdot f(an) + (1-a) f((1-a)n), a \in [0,1] \end{aligned}$$

To get an idea of how this definition works, suppose we chose $a = \frac{1}{2}$, as we do in binary search.

$$\begin{aligned} f(n) &= 1 + \frac{1}{2} f\left(\frac{n}{2}\right) + \frac{1}{2} f\left(\frac{n}{2}\right) \\ &= 1 + f\left(\frac{n}{2}\right) \end{aligned}$$

If we define $f(k) = 1$ for $k \leq 1$, then we see it takes $\log_2(n)$ steps to get to 1, so this reduces to approximately $f(n) = 1 + \log_2(n)$.

So how about other values of a?

3.1 Binary search algorithm

Before we answer the second question about doing more than bisection, e.g. trisecting, let's look at the algorithm for binary search.

```
def binarySearch(array, start, end, target):
    if start > end:
        return -1

    mid = (start + end) / 2
```

```
    if array[mid] == target:
        return mid
    elif array[mid] > target:
        return binarySearch(array, start, mid - 1, target)
    else: # array[mid] < target:
        return binarySearch(array, mid + 1, end, target)
```

Listing 1: Python implementation for binary search

Example 3.1. Using the binary search algorithm in 1, perform binary search for 4, 11, 7 in

```
[-6, 1, 4, 5, 6, 8, 11, 19, 21]
```

For each one, you can organize your work by filling out the following table

start	end	mid	array[mid]

Problem 3.2. So now with our algorithm, why don't we do a trisect search?

If we investigate our binary search algorithm, we compare at most 2 times each iteration, and when we hit mid, we stop.

For ternary search, what would our algorithm look like? Even with $\log_3(n)$ depth of the function calls, how many times do we compare at each level? Is this better or worse than binary search?

4 Binary search problems

Example 4.1. Given an array of integers A that are sorted ascending, and given a target t, find the closest integer $k \in A$ such that $|t - k|$ is minimal.

We can solve this naively with a linear search, and keeping track of the minimum. But our array is sorted, can we do better?

Example 4.2. Given an array of integers A that are sorted ascending, and given a target t, find the number of times t occurs in A.

We can solve this naively with a linear search, and keeping track of the frequency of t. But our array is sorted, can we do better?

5 Problems

1. *Challenge.* It can be proven that with a comparison-based algorithm, the optimal search algorithm takes $O(\log n)$ steps. Can you prove this?

    - If we don't use a comparison-based algorithm, what can we do? What results do we get for the efficiency of search?