Binary Search: Find it Quickly

Competitive Programming UTEC

June 6, 2020

1 Introduction

1.1 Motivation

Lets say you are given an integer array A of size n, and you have to answer the following query: Given an integer x find, i such that $A_i = x$, or determine that such position doesn't exist.

There is one obvious trivial solution for this problem, we iterate through all elements in the array and find an index that satisfies the condition. It is simple to see that this approach would have a complexity of $O(n)$, so if we have multiple queries this might be too slow.

Another sensible approach would be to keep an index *map*, in which we store for every value in the array it's position. With this apporach we can have a time of $O(1)$ per query, but will also need an auxiliary space of $O(n)$.

For most arrays, this are our only options. However, if the array is *sorted*, we can use binary seach to solve this problem in logarithmic time and $O(1)$ of memory.

1.2 Binary Search

Binary search is a *divide and conquer* search algorithm that allows us to look for values in sorted lists in logarithmic time. Thanks to it versatility, binary search is used in thousands of algorithms in order to boost up performance. Most importantly, its uses go **way beyond looking for values in an array**, as we will learn later.

In order to be able to apply binary search the following conditions must be satisfied:

- **Random Access:** We must be able to jump to any element in the sequence with an $O(1)$ complexity.
- **Order:** Elements in the sequence must follow some kind of order.

1.3 The Algorithm

The binary search algorithm is very simple. Lets say I want to find element x in array A. I could begin by guessing at which position x will be, for example I could guess that $A[i] = x$. There are three possibilities:

1. If am very lucky then $A[i] = x$.
2. $A[i] < x$
3. $A[i] > x$

It is obvious that in the first case I just need to return i as my answer, but what about cases 2 and 3. Lets say that $A[i] < x$, then as the array is sorted, we know that it is impossible that x is to the left of $A[i]$ as they will be all smaller than $A[i]$, and therefore, smaller than x. This means that if I were to guess again, I could ignore all indexes that are smaller or equal to i. Symmetrically, if $A[i] > x$, than I now that it is impossible that x is anywhere to the right of $A[i]$.

Now what if instead of guessing, we picked an index systematically, so that after every comparison we can discard a large number of the elements in the array of our candidates. The best way to do this is to pick the middle element of the array, as regardless of how $A[i]$ compares with it, we will get rid of half of the array in one move. Then we can repeatedly apply binary search until we find the value we are looking for.

As after each iteration we are halving the array, it is simple to see than in the worst case we will have to split the array until a single element remains. This is what yields a complexity of $O(lgn)$ for the search.

Just as it is the case with most divide and conquer algorithms, the most intuitive approach for the implementation of binary search is recursive. In the implementation l and r represent that range in which we can still find value x. The algorithm stops when we find x of when $l > r$, which means that x wasn't in the array.

```
BINARYSEARCH(A, l, r, x)
1  if l > r
2       return -1
3  m = ⌊(l+r)/2⌋
4  if A[m] < x
5       return BINARYSEARCH(A, l, m − 1, x)
6  if A[m] > x
7       return BINARYSEARCH(A, m + 1, r, x)
8  return m
```

This solution occupies $O(lgn)$ memory because of the extra space that needs to be allocated in the stack, unless we use the *tail recursion optimization*, case in which it will just consume $O(1)$ memory. A simple iterative implementation of binary search also exists. The idea for this is the same, but instead of calling the binary search method recursively, we use an while loop to update the values of l and r.

```
BINARYSEARCH(A, l, r, x)
 1  l = 1
 2  r = A.size
 3  while l ≤ r
 4        m = ⌊(l+r)/2⌋
 5        if A[m] = x
 6              return m
 7        if A[m] < x
 8              r = m − 1
 9        else
10              l = m + 1
11  return −1
```

Both of this implementation have the same asymptotic complexity of $O(lgn)$, however the iterative implementation is usually faster.

2 Orderings

In *order* to better understand binary search we need to first understand what *sorted* means. We say thing are sorted when they follow an *order*, but this inevitably arises the question, what is order?

In mathematics order is a binary relationship that exists in a set of elements. We will usually refer to this relationship with the symbol $\leq$. In order for a relationship to be considered an order it must follow some conditions, the most important being:

- **Antisymmetry:** $a \leq b \wedge b \leq a \rightarrow a = b$
- **Transitivity:** $a \leq b \wedge b \leq c \rightarrow a \leq c$

Under this definition of *order* we can define as array as sorted if:

$$\forall i, j \leq n(i < j \rightarrow A_i \leq A_j)$$

The most important thing of this is to remember that $\leq$ can represent any relationship that satisfies the condition described above. For example, $a \leq b$ can mean a is smaller than b, a is greater than b, $|a|$ is grater than $|b|$, etc...

2.1 Types of order

There are two types of order: total order and partial order. The only difference between this is that total order maintains *connexity*, that is to say that the given a pair of elements a and b, there must exist a relationship between them (either $a \leq b$ or $b \leq a$). In practice, this means that in a partial order, some pairs of elements don't have a defined order between them.

For example, lets say that we define $a \leq b$ as a is an ancestor of b. How would I compare with my cousin? We can't either say that I am my cousins ancestor or that my cousin is my ancestor, so how we should compare is not defined.

When we define an order, it is really important that we make sure this order is total, as partial ordering might lead to undefined behaviour. From here forward when we talk about order we will be referring to total orders, as with partial ordering some complications arise in the algorithms and theory we are going to explore.

2.2 Order in a Computer

In the context of computer science, we can think of the relationship $a \leq b$, as a 2-parameter boolean function $\text{COMP}(a, b)$, that returns `true` if the relation $a \leq b$ exists and `false` otherwise. This function is called the *comparator* and it fully defines the order of the array. In order to ensure the correctness of our algorithms, we must be sure that our comparator obeys the same restrictions the order relationship $\leq$ did.