Binary Search

February 1, 2021

1 Introduction

It is now time to turn our attention to searching and in particular, binary search - one of the classical problems in algorithms. A typical searching problem consists of three main components :

1. **The key.** The key uniquely identifies the element we are searching for. E.g. While searching for a particular student's roll number among all the students of a school, the key can be that student's roll number.
2. **Search space.** This is the set in which we have to search for our key. E.g. In our example, the search space will be the set consisting of all possible roll numbers.
3. **The searching algorithm.** This is the algorithm executed to find our key in the search space. E.g. in the above example, we can simply scan the roll numbers one by one until we find that student's roll number.

In most cases, the only search algorithm we can follow is linear search - going through the elements of the search space one by one until we find the key.

However, when our search space follows some particular properties, we have scope to optimise our search algorithm in terms of time and space complexity. Binary search is such an optimised search algorithm which works on a particular class of search spaces. We know that, binary search can be used on sorted arrays and brings down the time complexity of our search algorithm from O(n) to O(log n) where n is the size of the array. However, sorted arrays are just one specific instance amongst the possible areas where binary search can be applied. As this tutorial proceeds, we will try to find a more general property that needs to be specified in order to use binary search.

Here is how this tutorial is organised :

- We'll first try to apply Binary Search to a specific type of problem and try to essence out the essential things to keep in mind when devising a binary search solution. We'll see the power that this brings - getting to avoid the one-off boundary cases that we encounter frequently.
- We'll then see how any binary search problem can be reduced to this specific instance and in the process devise a general condition using which we can determine when to use binary search.

We fondly call this framework - "Predicate Framework"!

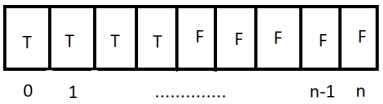
2 Predicate framework for Binary Search

In this section, we are going to be learning the predicate framework for binary search which is essential in solving all binary search problems. First, we are going to be looking at a model problem and methods for solving it and then we are gradually going to work our way up to see how all binary search problems can be reduced to our model problem.

2.1 Defining our model problem

Let's start with the introduction of one instance of our model problem.

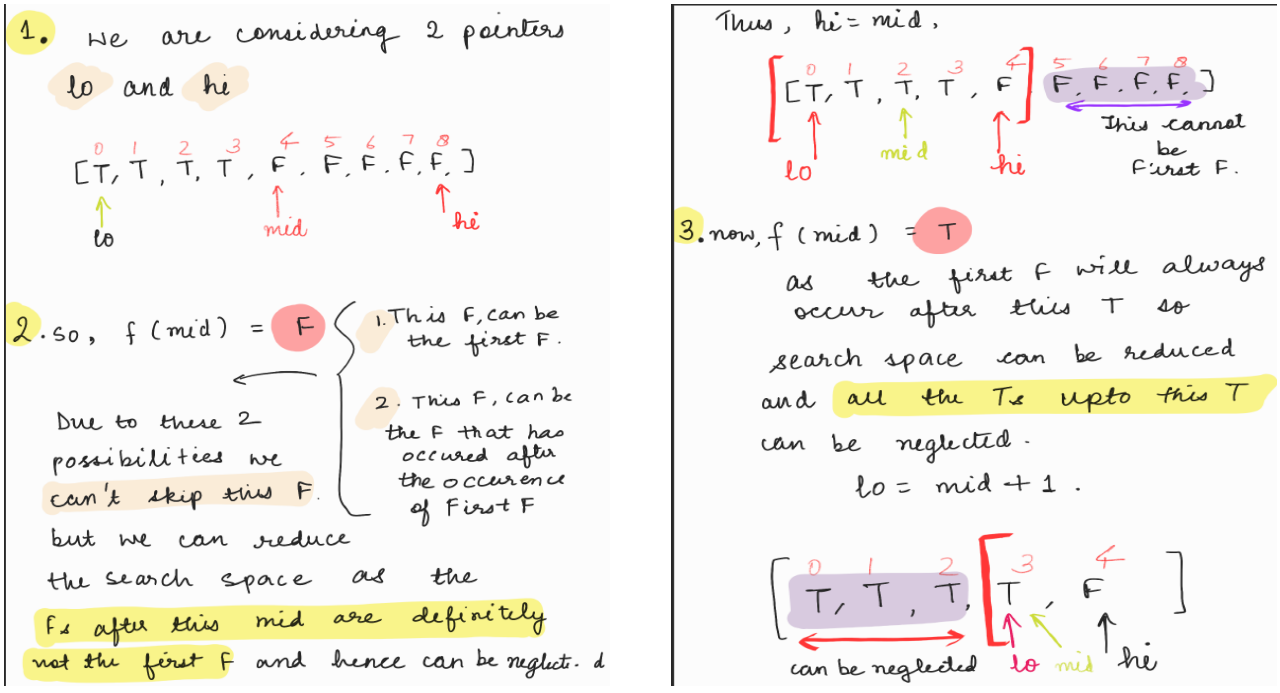
Consider an array of **"n" Boolean variables**. Our task is to find the index of the **first "F"** i.e. the smallest index in the array that is equal to an "F". Let's use the name **var** to denote this element. Here **"var" will be equal to "F"**.



Note that in the figure we can have zero "T"s and only "F"s and vice-versa.
As we saw, our task is to find the index of the first "F". We do this by following the method stated below. At this point, we just want to get familiarised with what we are trying to do. As the tutorial proceeds we will be discussing the finer details of this method.

- Step 1 : Initialise the search space. We typically define the search space by initialising the lower limit which we call "lo" and the upper limit which we call as "hi". Here, lo will be set to 0 and hi will be set to n-1.
- Step 2 : Check if the array contains more than one element by comparing the value of lo and hi. If, $lo \geq hi$ (i.e. one element) then go to step 6.
- Step 3 : Initialise the middle element known as "mid" by setting $mid = lo + (hi - lo)/2$
- Step 4: Check the value of arr[mid].
  - If mid is true, then we are still in the part of the array containing "T"s and we can limit our search to the right half of our array by putting $lo = mid + 1$. We can discard the mid as well because we are interested in an "F" after all.
  - If mid is false we have two possibilities:
    1. We are at the first F.
    2. We are still towards the right of the first F.

    To handle both the cases, we put $hi = mid$.
- Step 5 : Go back to step 2.
- Step 6 : Conduct a sanity check on the last remaining element. (This element can be identified as arr[lo] or arr[hi].) Check the value of arr[lo].
  - If arr[lo] = False then lo is our answer.
  - Else, the array only contains True variables. In this case no answer exists.



(a) (b)

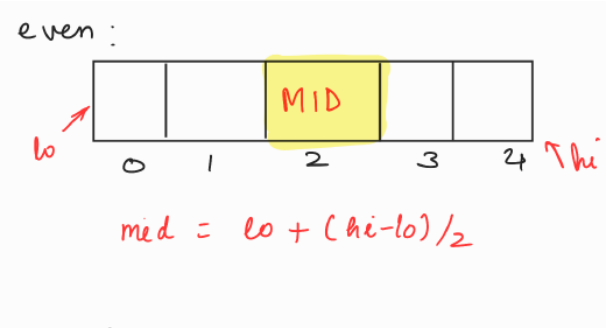
A visualisation of the method

This concludes our answer for our model problem. I want the reader to pause at this point and go through the algorithm in detail. You have to convince yourself that this algorithm works for our test case before moving on to the next part of the tutorial. I will also suggest writing a small simple code to make sure the algorithm works and that you have understood it fully.

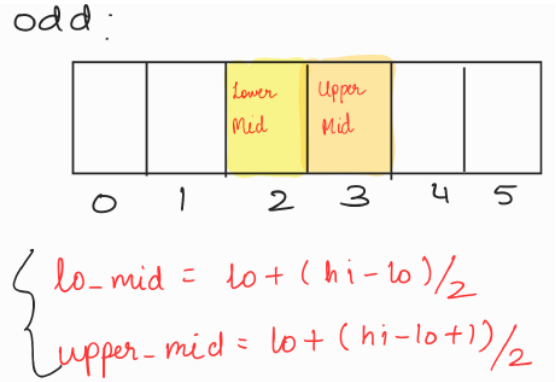
POINTS TO NOTE

There are some fine points in this algorithm that should be pointed out now that we have gone over this algorithm.

- Our loop always only runs : $while(lo < hi)$ i.e. we don't stop even if mid is the element that we are looking for. Once we are left with a particular element, we check if it satisfies our need (sanity check).
- In step 4, when arr[mid] was False, we put hi as mid and not mid+1. This was done to include mid in our next iteration of searching.
- In step 3, we had two choices for mid. We could have put $mid = lo + (hi - lo)/2$ [lower- mid] or we could have put $mid = lo + (hi - lo + 1)/2$ [upper-mid]. We deliberately chose to go with one of these choices to avoid an infinite loop. Below we have given an illustration of what would have happened if we had gone with the other choice for mid.
- The sanity check. This is an important concept and I want to emphasise more on why it is needed. While solving the problem, we had assumed that at least one "T" exists in the array. But, it may happen that the array contains only "F" variables. In that case, our loop will stop but arr[lo] will not contain "T". Our sanity check basically checks for this possibility.

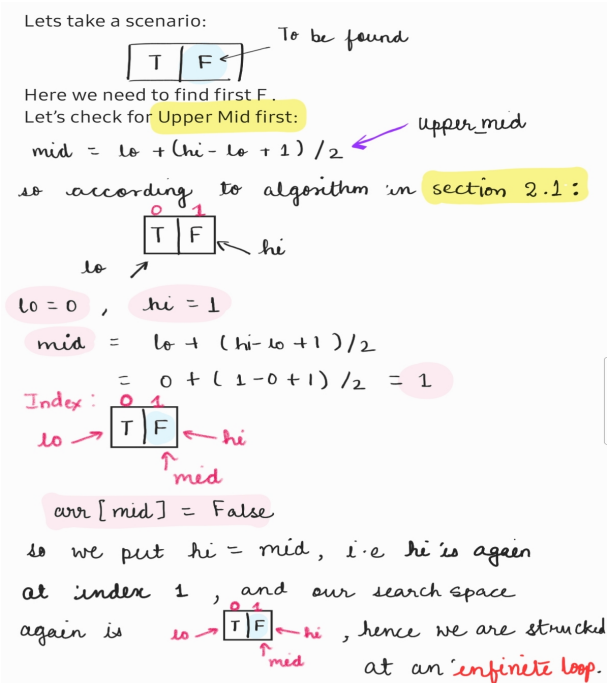


(c)

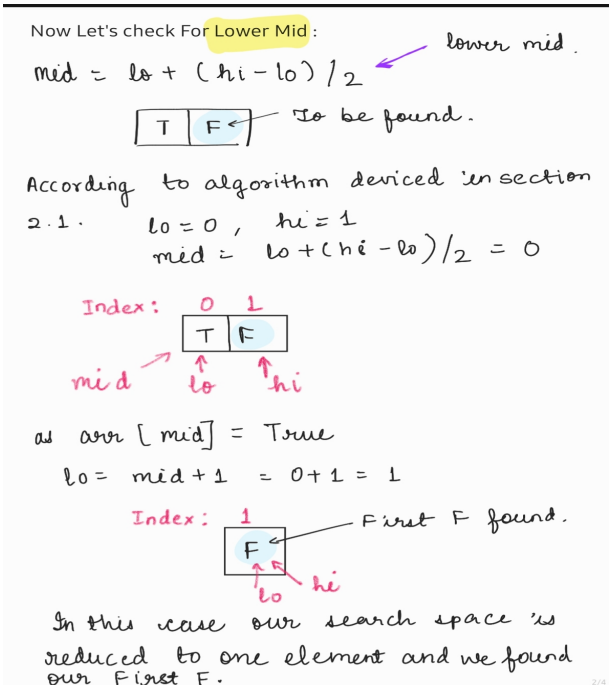


(d)

The two choices of mid



(e)



(f)

Illustration for two choices for mid

2.2 Instances of model problem

The example we solved above is just one instance of our model problem. In this section, we look for a generalised algorithm to solve all instances of our model problem. Let us now look at all the four possible instances of our model problem :

Type of array	**Value to look for : (var =)**
TTTTTFFFFF	the last "T"
TTTTTFFFFF	the first "F"
FFFFFTTTTT	the last "F"
FFFFFTTTTT	the first "T"

2.3 Common algorithm for all instances of the model problem

We now try to modify our approach so that it uniformly applies to all the instances.Most of the steps that we followed remain the same and we make minor changes to a few steps.

- Step 1 : Initialise lo and hi values. Here, $lo = 0$ and $hi = n - 1$.
- Step 2 : Check the values of lo and hi, if $lo \geq hi$ then stop the search.
- Step 3 : Define the mid appropriately [lower or upper]. Skip this step for now. We come back to it later. For now, assume that mid is appropriately defined.
- Step 4: Check the value of arr[mid]. Now using the same method as followed previously, we set appropriate value for lo and hi. Let's say "var" is the value we are looking for. "var" can be True or False according to the instance of the problem we have been given. "var" will be True in instance 1 and 4 and will be False in instance 2 and 3. Then, our code will always have structure :

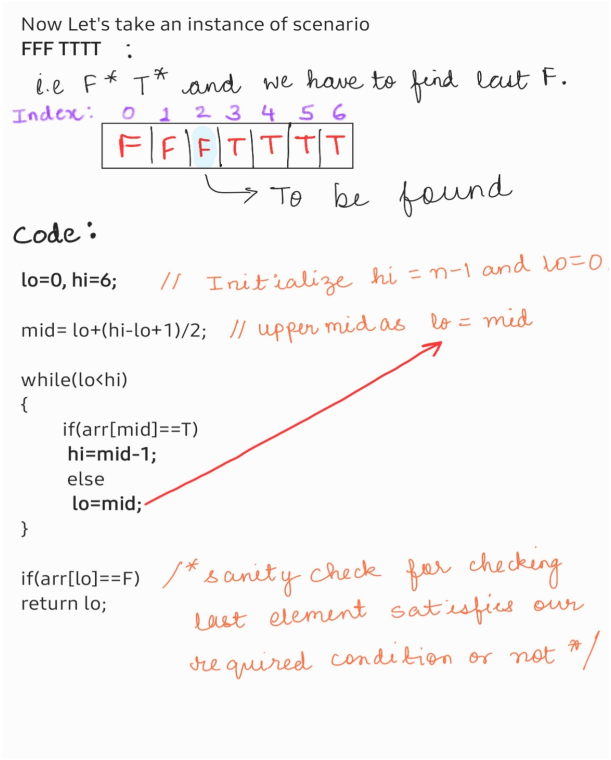
```
if(arr[mid]==var)
     \\execute part A of the code
else
     \\execute part B of the code
```

  Consider the case where arr[mid] == var. Then, since mid is a possible candidate for our solution, we obviously know that mid is included in the remaining set of values that we are going to search. So, the A part of the code will either have $lo = mid$ or $hi = mid$.. Consequently, part B of the code will have $lo = mid + 1$ or $mid = hi - 1$ As an example, refer to the steps we had done above in one particular instance of the problem.
- Step 5 : Go back to step 2.
- Step 6 : Sanity check. Check to see if arr[lo] is equal to var. If it's not, we don't have an answer. This step is same as for the above method.

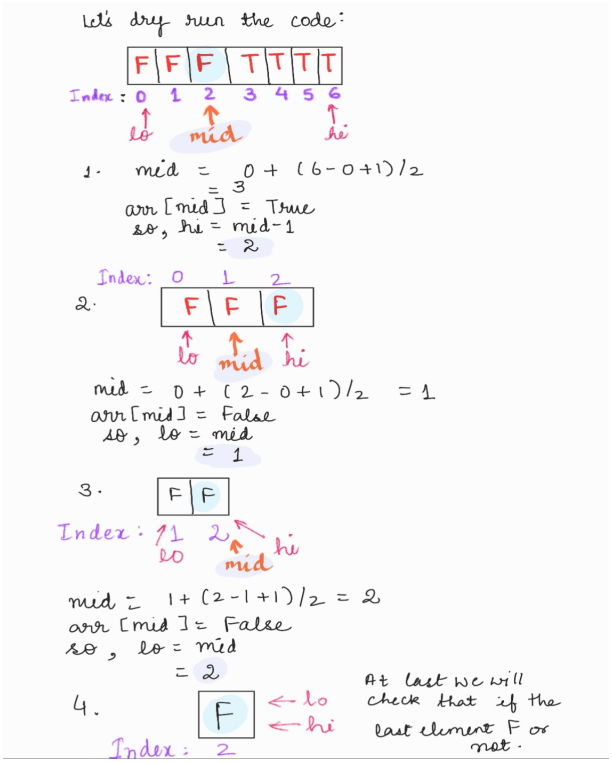
Let us know turn our attention to the step 2 that we had omitted above. I am now going to give a simple thumb rule for deciding the mid. The following rule helps decide which mid to use. As we saw above, we will always either have hi = mid or lo = mid in the "A" part of our code.

Select the lower-mid if you have hi = mid in A.

Select the upper-mid if you have lo = mid in A.



(g)



(h)

The generalised method applied to one particular instance of the model problem.

We have reached the end of our solution to the model problem. Let us now understand why this model problem is important or relevant. You can see that with this solution, we have devised a method which is never going to give us an infinite loop. It also removes all confusion about whether lo or hi contains our answer at the end. We know that we can use either of arr[lo] or arr[hi] in our sanity check.

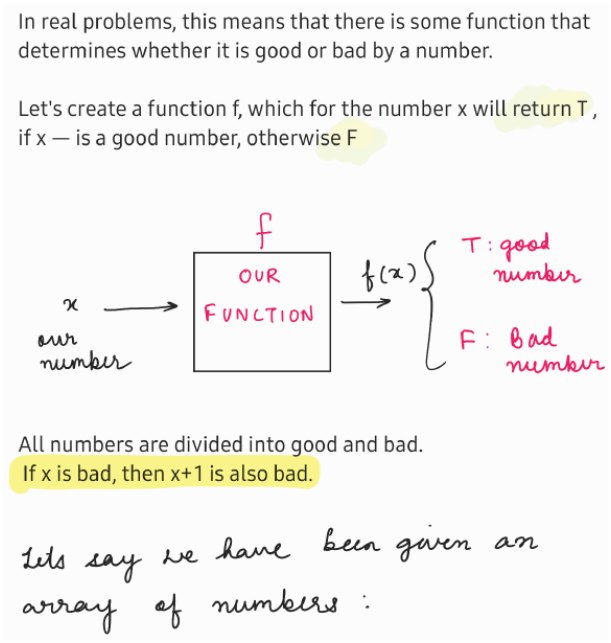
We should also note what we have skipped in this tutorial. Even though we saw an effective and efficient method, we never talked about why it always works in all possible instances of our problem. I request you to pause here and convince yourself that this solution works and try to reason about why it always works.

3 Reducing every problem to an instance of the model problem

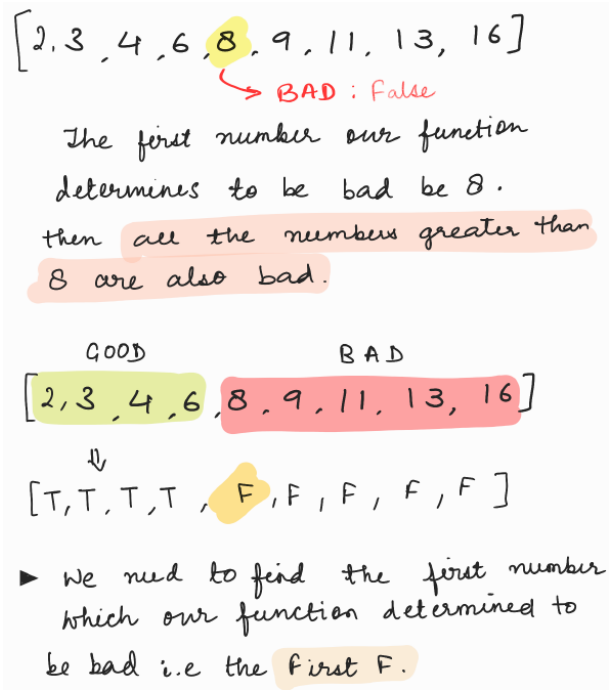
Now that we have spent all this time developing a solution for our model problem(s), it is time to see how it is applicable to any binary search problem. Consider a very easy and typical binary search problem - You are given a sorted array of numbers called arr and you have to find the index of a random number say 'k'. To solve this problem, consider an array of Boolean variables called arr1 defined as follows :

$$arr1[i] = \begin{cases} True & if\ arr[i] < k \\ False & if\ arr[i] \geq k \end{cases}$$

Now our problem reduces to type 2 of the table above and we can easily code it using the solution we had developed. The only part that needs to be changed is the sanity check - after we are left with one element we can simply check if it equals k and return appropriately. The crux of this method is that any problem which can be modeled such that we have a sequence of "T"s followed by a sequence of "F"s or vice versa and finding the index of the last "T" or the last "F" will give us our solution can now be very easily solved by us using binary search. At this point, this sounds like an incredibly specific condition but over the course of the next tutorial, we will see that several problems do satisfy these conditions. We will learn how to solve identify these problems as well as how to solve them using our framework.



(i)



(j)

A visualisation of the problem