Problems on binary search

February 10, 2021

1 Introduction

In the last tutorial, we saw the predicate framework for binary search, and how it helps us apply binary search to a problem. We also briefly talked about the general condition that a problem must satisfy in order to be eligible for binary search. In this tutorial, we will be looking at a few representative problems and a general method that can be followed to solve those problems using binary search.

Binary search problems can be broadly divided into two categories :

1. Intuitive search spaces
2. Non - intuitive search spaces

The main difference in both the types is that in the first category, the problem explicitly asks for us to search for a key in a given search space. In most cases of category 1 problems, it's also apparent that the search space has some sort of ordering(Should I explain what ordering is?). Sometimes the ordering may not be readily apparent but will become so after a bit of work. In the second category of problems however, some working is needed before we can identify the problem as a searching one. In this tutorial we limit ourselves to category 1 problems. Category 2 problems will be taken up in the next tutorial.

The objective of this tutorial is to develop a method for identifying problems where binary search can be applied and for coding the solution of such problems. We want to use each problem as a tool to understand how similar problems can be coded. In each question pay close attention to :

- The cues used to identify that binary search will be needed to solve this question.(In category 1 problems, the cues will be fairly obvious)
- The method used to apply the predicate framework.
- The way an algorithmic solution is converted to code.

2 A brief recap

Before diving into the problems let's have a brief recap of the predicate framework. In the last tutorial, we defined our model problem. We also saw the four separate instances of our model problem. Then, we said that the "predicate" is a function that maps our search space to one instance of our model problem. While solving any problems on binary search, the main challenge is going to be identifying our search space and then defining a proper predicate function. Therefore, before moving to the problems, make sure that the concept of the predicate function is fully clear.

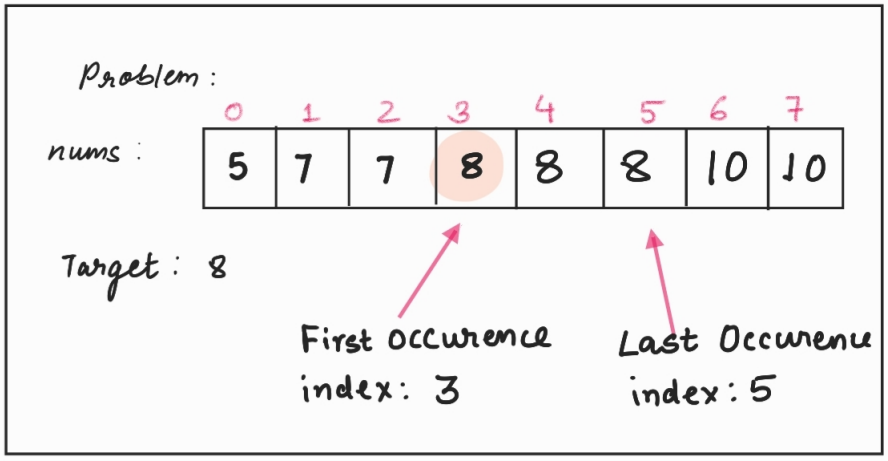
3 Find First and Last Position of Element in Sorted Array

3.1 Understanding the problem statement

The problem statement of our problem is as follows :

> **Given an array of integers nums sorted in ascending order, find the starting and ending position of a given target value.If target is not found in the array, return [-1, -1].**

The problem statement is pretty straight forward and an illustration is given below.



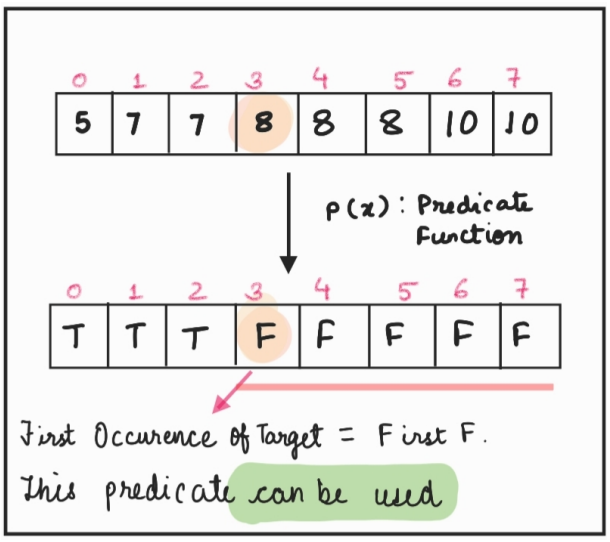
3.2 Algorithm for the solution

In this section, we will begin to formulate an algorithm for solving the question. The first point to be noted is that : **There are two things we need to find to solve the problem**. We need to find the first position as well as the last position of an element.

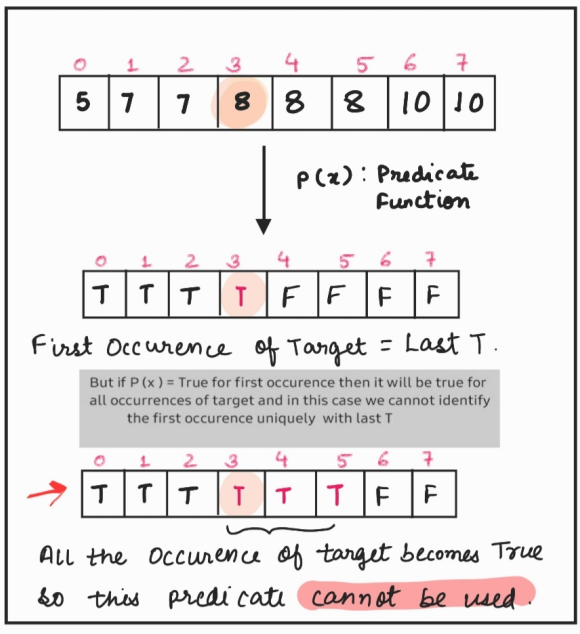
First, let us try to solve one part of the problem by finding the start position. It is fairly obvious here that binary search may work since, the array is sorted and we are "searching" for something. Now, assuming that binary search is to be used for this, we need to figure out how to apply binary search here. As we said in the last tutorial, we need to find an appropriate predicate function to reduce our real world problem to one instance of the model problem. Once that is done, we can apply the generalised algorithm we had developed in the last tutorial to code the solution.

3.2.1 Finding an appropriate predicate function

Let's say we decide to convert our array into a "TTTTFFFFF" form. (This choice is arbitrary and we could have gone with "FFFFTTTTT" also and it wouldn't have made a difference). Now, our predicate can evaluate T at our target value or F. Both the scenarios are illustrated below.



(a)



(b)

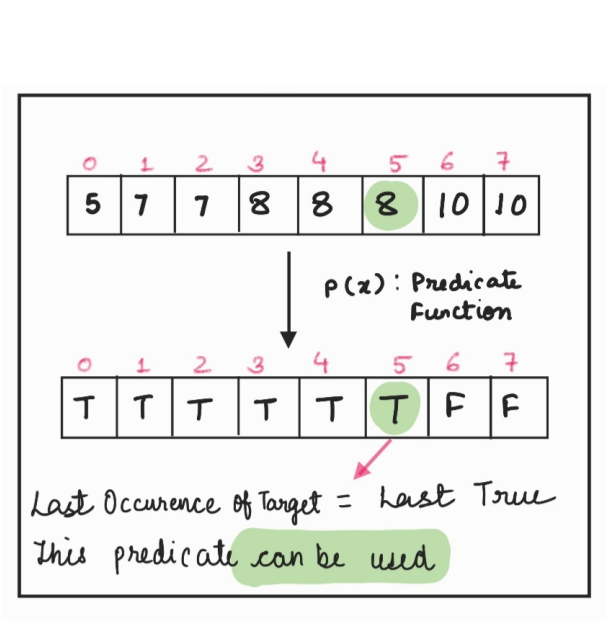
Two possibilities for the predicate

It is clear from the image that in the scenario (b) the first occurrence is neither the first F nor the first T. Therefore, we have to go with scenario (a) only. I request the reader to pause and reflect on what mapping could

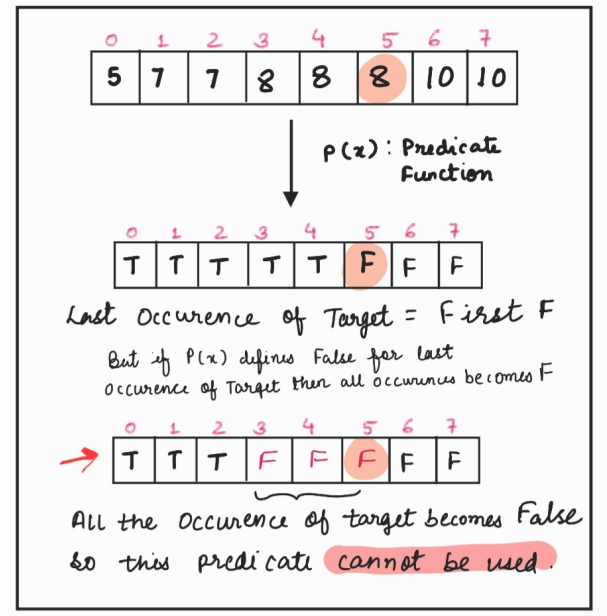
fit our requirements for the predicate function. The answer is given below :

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

Thus, we have successfully found our predicate function for the first problem. Let's now move to the second half of the problem. Let's again convert our array to the type TTTTTFFFF. As before, our predicate can evaluate T or F for the target value and both the scenarios are shown below.



(c)



(d)

Two possibilities for the predicate

Here also, our answer can only be the first scenario(c).The reason for that is same as above. In the second scenario, the last occurrence of the target value is neither the last T nor the first F and so we can't go with this. Thus we have to go with scenario (c). Again, pause and reflect on what mapping could fit our requirements for the predicate function.The answer is given below :

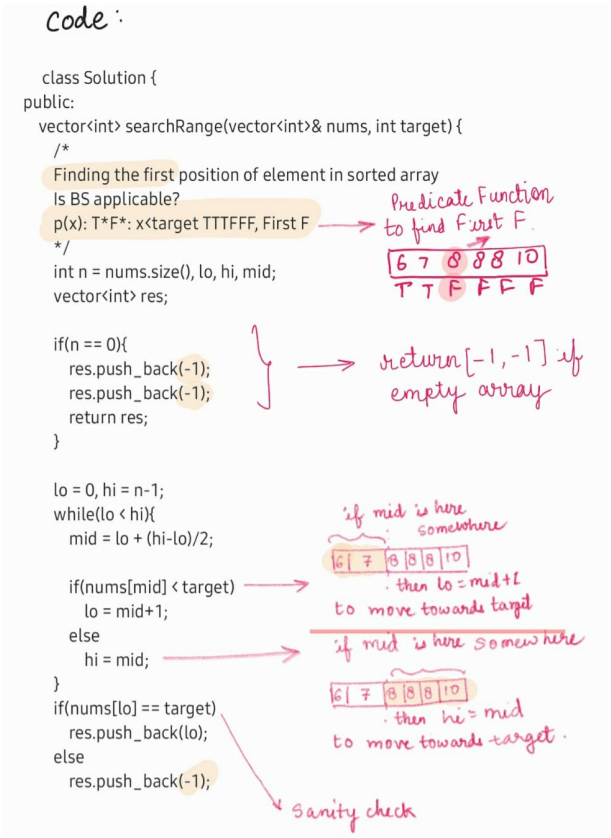
$$arr1[i] = \begin{cases} True & if\ arr[i] \leq target \\ False & if\ arr[i] > target \end{cases}$$

In the first part, the predicate reduced our problem to instance number 2 of our model problem and in the second part, our problem was reduced to instance number 1 of the problem(According to the table given below).

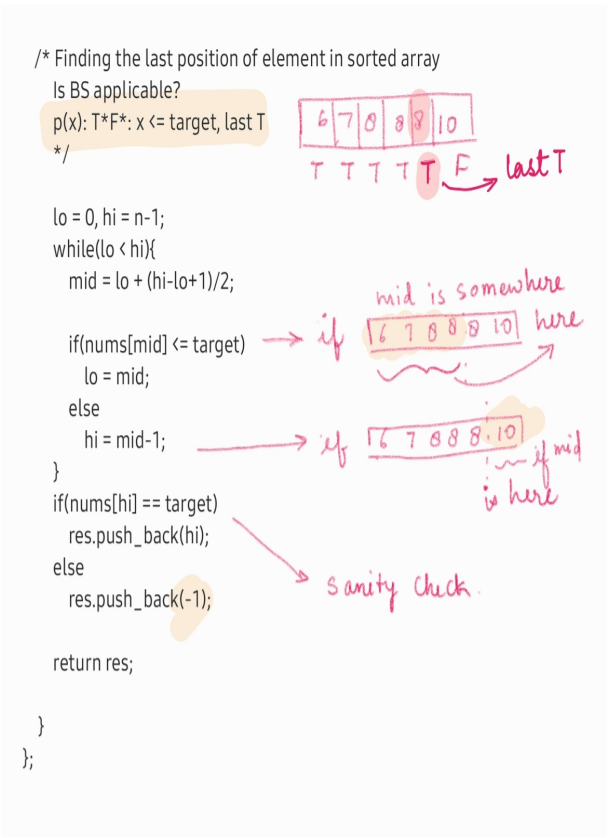
Type of array	**Value to look for : (var =)**
TTTTTFFFFF	the last "T"
TTTTTFFFFF	the first "F"
FFFFFTTTTT	the last "F"
FFFFFTTTTT	the first "T"

3.3 Coding the solution

It is now time to code our solution to the problem. This will be easy since, we already have a pseudo code ready from our previous tutorial. I request you to use this elementary example to understand the process of converting an algorithm to code. The code given below along with our annotations explains the relation between the actual code and our pseudo code.



(e)



(f)

Two possibilities for the predicate

TRY IT YOURSELF

Try repeating the entire problem by using FFFFTTT instead of TTTTFFFF. You should get a different predicate and different instance of the model problem.

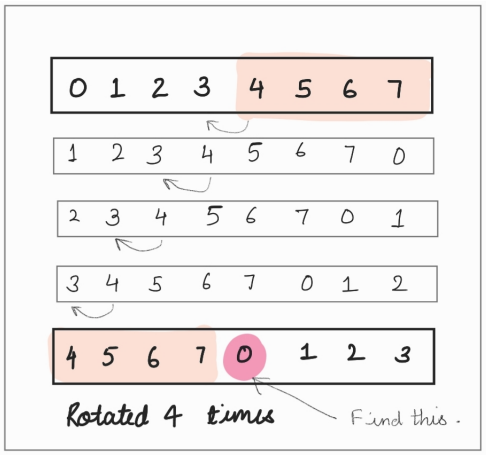
4 Search in Rotated Sorted Array

4.1 Understanding the problem statement

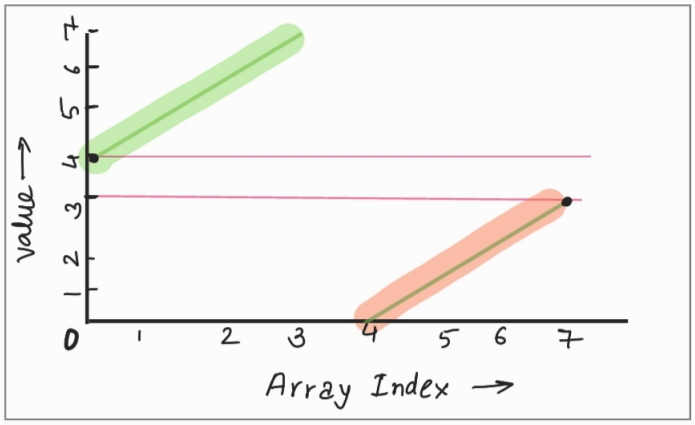
The problem statement of our problem is as follows :

> **Given the sorted rotated array nums, return the minimum element of this array.**

The problem says that we have a sorted array. Then, this array is rotated a specific number of times as shown below :



We have to find the minimum value in this rotated array. Let's try to visualise the distribution and range of values in the sorted rotated array by the means of a graph.



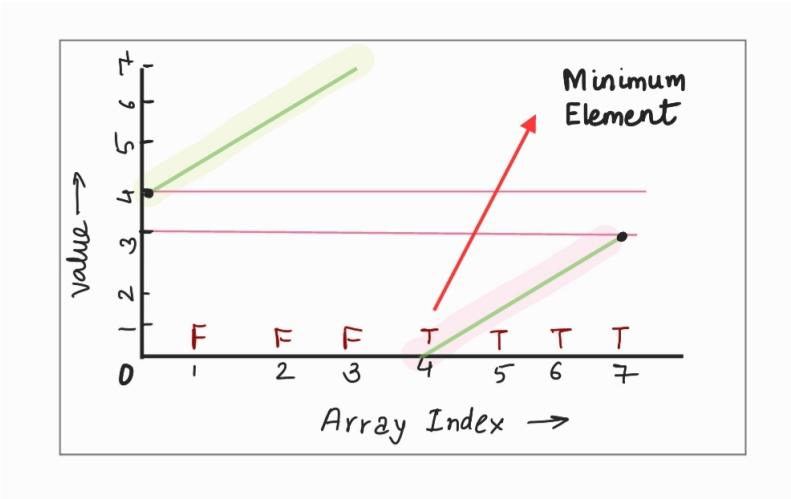
Carefully observe this graph and try to convince yourself that any sorted rotated array will have a graph like the one shown above. One of the important things you should observe in this graph is the position of the minimum.

4.1.1 Algorithm for the solution

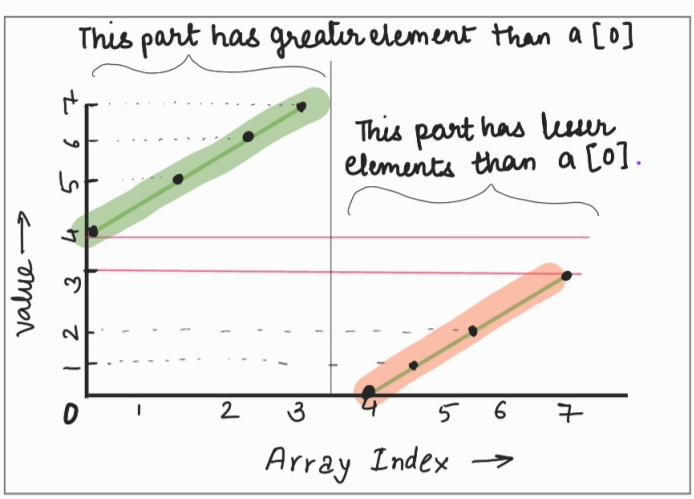
We know that this is a problem which involves searching. We also know that the array we are dealing with was originally sorted. Now, even though there have been some modifications on the array, it still holds some essence of sorting and hence, binary search would be a fair bet. Thus, let's try to apply binary search to the problem.

4.1.2 Finding an appropriate predicate function

Let's say we decide to convert our array into a "FFFFTTTT" form. Thus, we want our minimum to be the last F or first T. So, we want our predicate function to evaluate F for all elements before the minimum element and T for all elements after the minimum element. We will decide the value for the minimum element in a while. Therefore we want such a function :



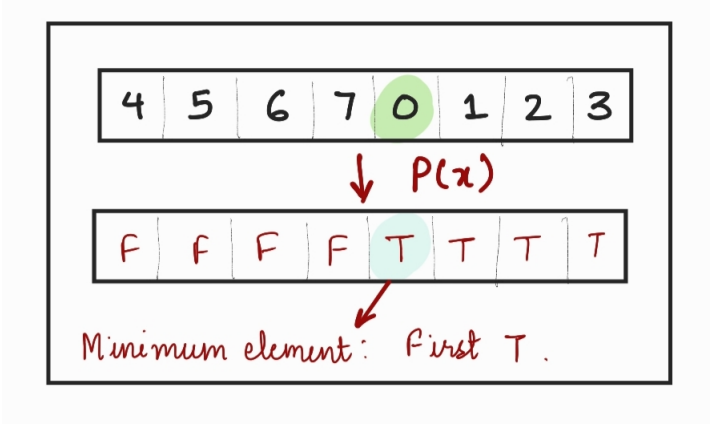
Let's try to use *comparison with the first element of the array* to build our predicate function. It is apparent from the graph that all elements before the minimum element are greater than the first element of the array and all elements after the minimum element are greater than it as shown below :



Thus, let's try to define our predicate function as follows:

$$arr1[i] = \begin{cases} False & if\ arr[i] \geq arr[0] \\ True & if\ arr[i] < arr[0] \end{cases}$$

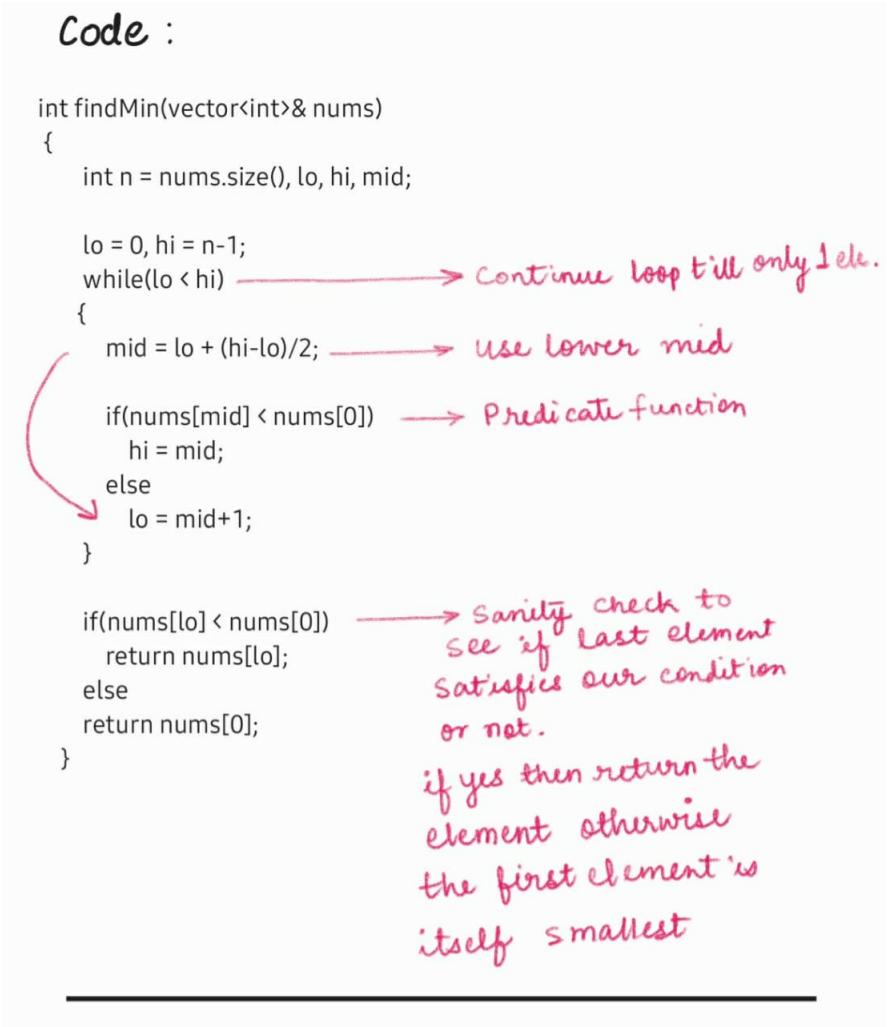
With this function, our array will be converted to FFFFTTTT as follows :



Note that, with this predicate function, our minimum element has become the first F. There is no possible way to modify this predicate to make our minimum element as the last T. So, our problem has reduced to instance 2 of the model problem.

4.2 Coding the solution

For coding the solution to our problem, we will follow the same method of converting pseudo code to actual code that we had followed in the previous problem. An illustration showing the correspondence between the solution code and the algorithm we had developed is given below for your reference.



TRY IT YOURSELF

Try repeating the entire problem by using the last element for comparison instead of the first element. Observe how it changes the process of finding the predicate and coding the solution.

5 Conclusion

In this tutorial, we saw two questions that could be solved using the binary search. We saw that since we used our ready made algorithm from the last class, coding these problems became very easy for us. The common method followed for these problems is as follows. The same method can be considered as a template to solve most binary search problems.

THE COMMON METHOD

- Identify the search space and the key.
- Decide whether the array should reduce to form TTTFFF or FFFFTTT.
- Decide whether they key should be the last T/F or the first F/T.
- Decide an appropriate predicate function which reduces your problem to the chosen instance of the model problem.
- Code the solution by following the generalized algorithm we had developed in the last tutorial.

The second problem that we solved serves as an example of how binary search can have more applications that just sorted arrays. As long as we are able to find a suitable predicate function, binary search can be used to solve the problem. Through the code provided, we also got a feel of how an algorithm can be converted to code.

In the next tutorial, we will build on the concepts of this tutorial to solve category two problems.