Find the Closest Number

In this lesson, you will learn how to find the closest number to a target number in Python.

We'll cover the following

In this lesson, we will be given a sorted array and a target number. Our goal is to find a number in the array that is closest to the target number. We will be making use of a binary search to solve this problem, so make sure that you have gone through the previous lesson.

The array may contain duplicate values and negative numbers.

Below are some examples to help you understand the problem:

Example 1

```
Input : arr[] = {1, 2, 4, 5, 6, 6, 8, 9}
Target number = 11
Output : 9
9 is closest to 11 in given array
```

Example 2

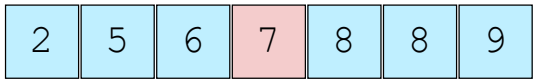
```
Input :arr[] = {2, 5, 6, 7, 8, 8, 9};
Target number = 4
Output : 5
```

Now the intuitive approach to solving this problem is to iterate through the array and calculate the difference between each element and the target element. The closest to target will be the element with the least difference. Unfortunately, the running time complexity for this algorithm will increase in proportion to the size of the array. We need to think of a better approach.

Algorithm

Have a look at the slides below to get an idea of what we are about to do:

Find Closest Number



target = 4
midpoint = 7
left of midpoint = 6
right of midpoint = 8

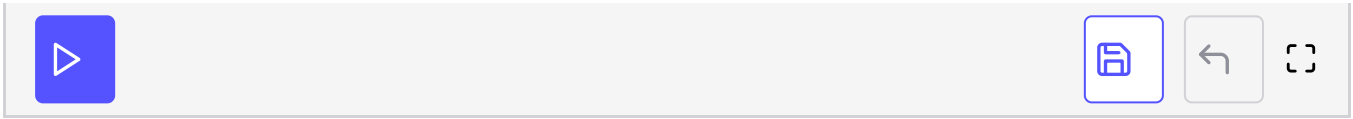
1 of 7

Implementation

Here we used the idea in the slides above to come up with a solution. Check it out below:

```
A1 = [1, 2, 4, 5, 6, 6, 8, 9]
A2 = [2, 5, 6, 7, 8, 8, 9]


def find_closest_num(A, target):
    min_diff = float("inf")
    low = 0
    high = len(A) - 1
    closest_num = None
```

```
    # Edge cases for empty list of list
    # with only one element:
    if len(A) == 0:
        return None
    if len(A) == 1:
        return A[0]

    while low <= high:
        mid = (low + high)//2

        # Ensure you do not read beyond the bounds
        # of the list.
        if mid+1 < len(A):
            min_diff_right = abs(A[mid + 1] - target)
        if mid > 0:
            min_diff_left = abs(A[mid - 1] - target)

        # Check if the absolute value between left
        # and right elements are smaller than any
        # seen prior.
        if min_diff_left < min_diff:
            min_diff = min_diff_left
            closest_num = A[mid - 1]

        if min_diff_right < min_diff:
            min_diff = min_diff_right
            closest_num = A[mid + 1]

        # Move the mid-point appropriately as is done
        # via binary search.
        if A[mid] < target:
            low = mid + 1
        elif A[mid] > target:
            high = mid - 1
        # If the element itself is the target, the closest
        # number to it is itself. Return the number.
        else:
            return A[mid]
    return closest_num


print(find_closest_num(A1, 11))
print(find_closest_num(A2, 4))
```



find_closest_num(A, target)

Explanation

`A` is the input array, and `target` is the element to be searched. `min_diff` is set to `float("inf")` on **line 6** so that it acts as an upper bound for the minimum difference between target and the other elements. Just as in binary search, `low` and `high` are set to `0` and `len(A) - 1` respectively (**lines 7-8**). On **line 9**, `closest_num` is initialized to `None` and we will update it as we move along the

solution.

Before we get to the crux of the solution, let's discuss the edge cases we have written from **lines 11-16**:

```
# Edge cases for empty list of list
    # with only one element:
    if len(A) == 0:
        return None
    if len(A) == 1:
        return A[0]
```

If there is an empty list, i.e., `len(A)` is equal to `0`, `None` is returned to indicate that there is no element closest to `target`. If there is only one element in the list, i.e., `len(A)` is equal to `1`, then only that element is returned on **line 16** to indicate that it is closest to `target` as there is no other element for comparison.

Next, we have a `while` loop just like we have in Binary Search. Below is a snippet from **lines 18-26**:

```
    while low <= high:
        mid = (low + high)//2

        # Ensure you do not read beyond the bounds
        # of the list.
        if mid+1 < len(A):
            min_diff_right = abs(A[mid + 1] - target)
        if mid > 0:
            min_diff_left = abs(A[mid - 1] - target)
```

We calculate `mid` in the same way we do in Binary Search. Then, we have to calculate the difference between `target` and the elements to the left and the right of `mid`. To ensure we don't go out of bounds of the list, we check if `mid+1` is less than the length of `A`, only then we access the element on the position `mid+1` on **line 24** to calculate the difference. On **line 24**, we take the absolute of the difference between `A[mid+1]` and `target` and store it in `min_diff_right`. Similarly, we ensure that we don't go out of bounds of the array on the left side and check if `mid` is greater than `0` on **line 25**. If it is, `min_diff_left` is calculated by taking the absolute of the difference of `target` and `A[mid-1]` on **line 26**.

After we are done calculating the difference between `target` and the elements to the left and right of the midpoint, we'll figure out which is the closest to target by comparing the differences. The code below is from **lines 28-37** which compares `min_diff_left` and `min_diff_right` to `min_diff` and updates the `min_diff` accordingly.

```
# Check if the absolute value between left
        # and right elements are smaller than any
        # seen prior.
        if min_diff_left < min_diff:
            min_diff = min_diff_left
            closest_num = A[mid - 1]

        if min_diff_right < min_diff:
            min_diff = min_diff_right
            closest_num = A[mid + 1]
```

If `min_diff_left < min_diff` is `True`, `closest_num` is set to `A[mid - 1]` on **line 33**. Otherwise, if `min_diff_right < min_diff` evaluates to `True`, on **line 37**, `closest_num` updates to `A[mid + 1]`.

Now let's discuss the code from **lines 39-49**:

```
        # Move the mid-point appropriately as is done
        # via binary search.
        if A[mid] < target:
            low = mid + 1
        elif A[mid] > target:
            high = mid - 1
        # If the element itself is the target, the closest
        # number to it is itself. Return the number.
        else:
            return A[mid]
    return closest_num
```

The next midpoint is updated in a similar way as in Binary Search. However, here we need to cater to an additional case in which the midpoint itself is the target element. For such a case, we return the midpoint, i.e., `A[mid]` on **line 48**. The `while` loop will keep iterating and dividing the array in case `low` is less than `high` to find the closest number to `target`. When the loop

terminates, we return `closest_num` on **line 49**.

That ends the discussion for this problem. In the next lesson, we will look at a different problem and analyze how it is solved using a binary search. See you there!