Find Bitonic Peak

In this lesson, you will learn how to find the bitonic peak using a binary search in Python.

We'll cover the following

In this lesson, we will be given an array that is bitonically sorted, an array that starts off with increasing terms and then concludes with decreasing terms. In any such sequence, there is a **"peak"** element which is the largest element in the sequence. We will be writing a solution to help us find the peak element of a bitonic sequence.

> A bitonic sequence is a sequence of integers such that:
> $x_0 < ... < x_k > ... > x_{n-1}$ for some k, 0 <= k < n

Notice that the sequence for this problem does not contain any duplicates.

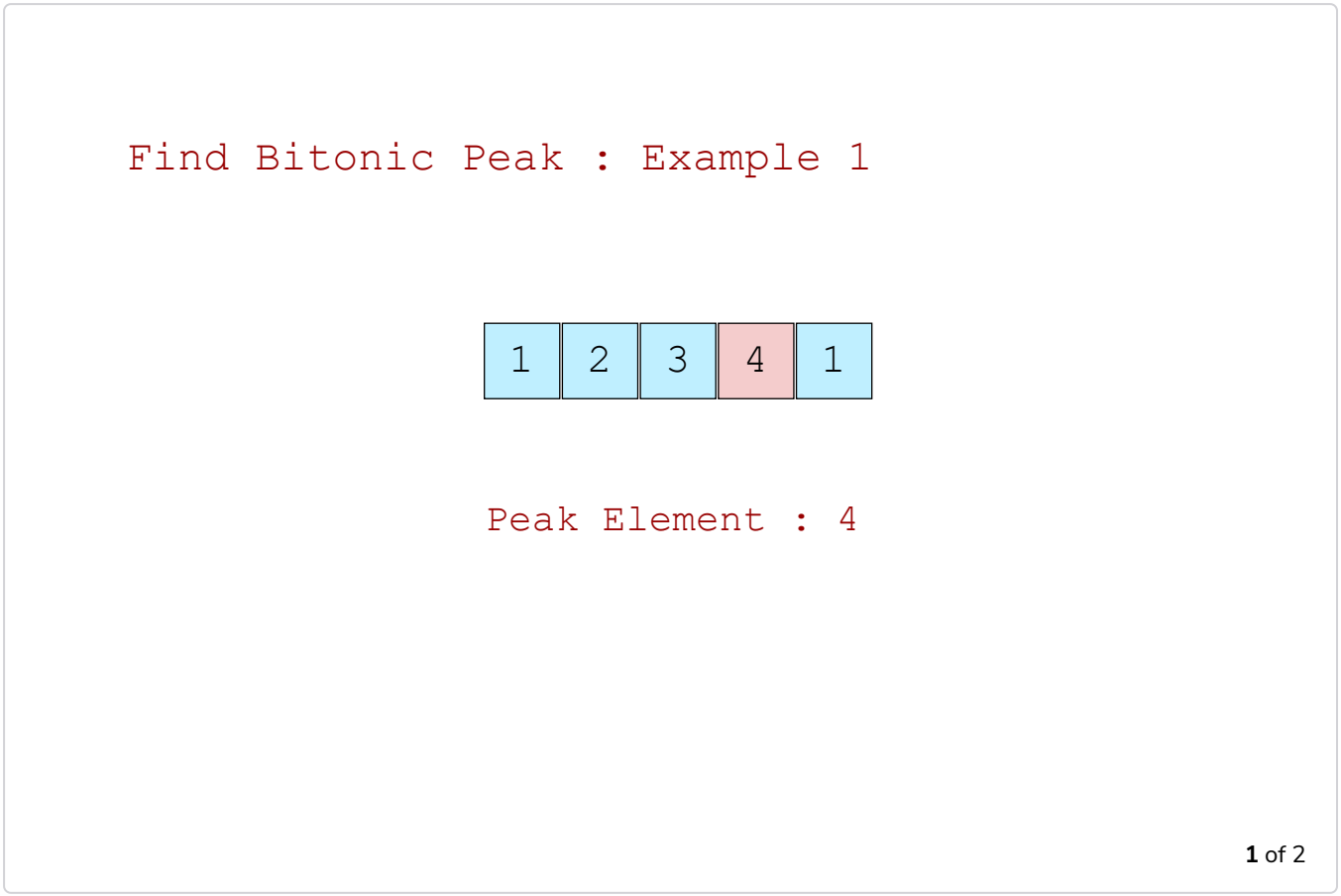
For example:

```
1, 2, 3, 4, 5, 4, 3, 2, 1
```

is a bitonic sequence. In the example above, the peak element is `5`.

We assume that a "peak" element will always exist.

Let's look at some other examples.



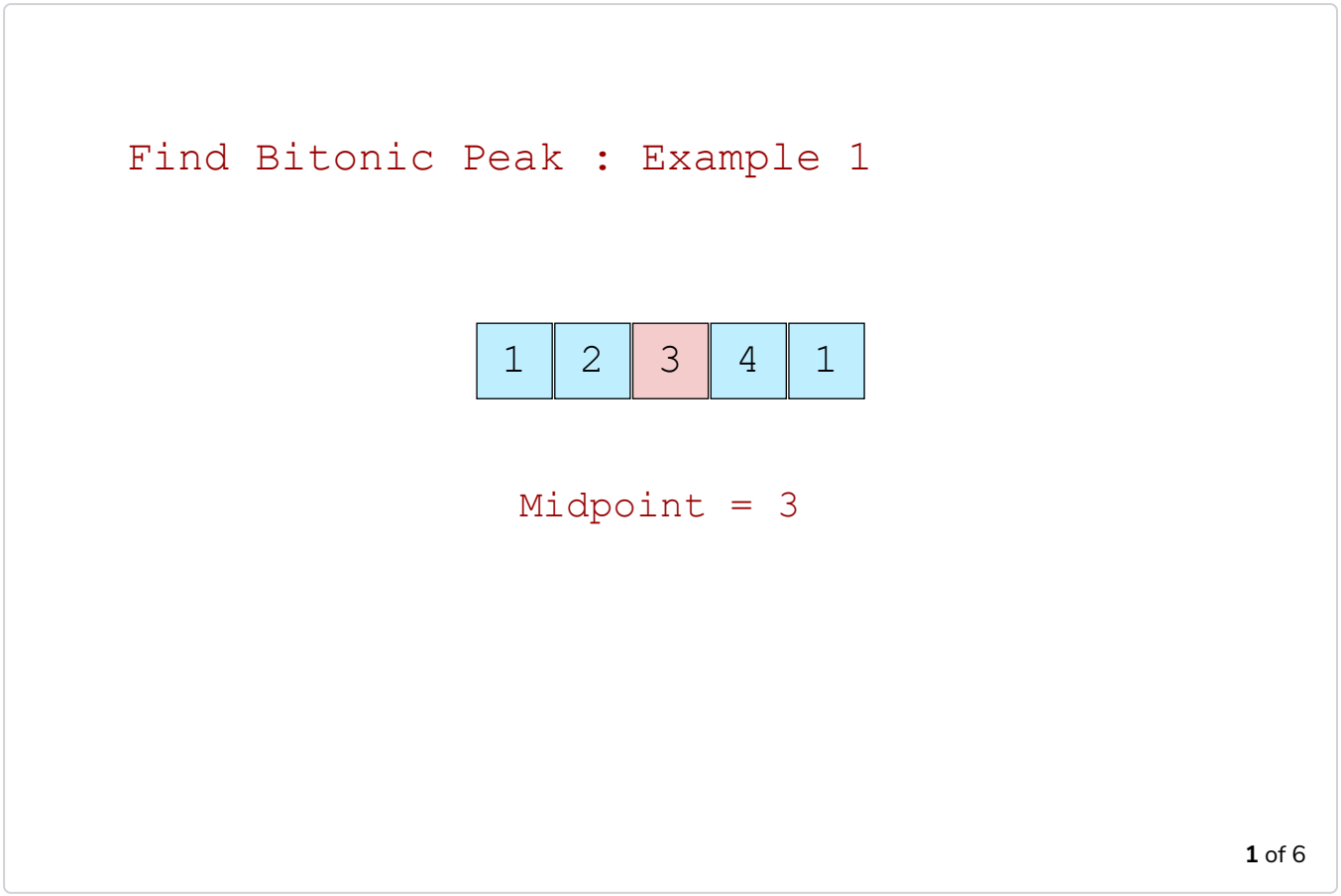
We can think about a naive way to solve this problem which checks the elements to the left and right of a given element to see if they satisfy the peak requirement. The peak requirement is as follows:

- The element to the left of the peak element is less than the peak element.
- The element to the right of the peak element is less than the peak element.

Let's say we start at the beginning of the array and go sequentially checking every element for the peak property until we arrive at an element that satisfies the requirement of being a peak element. This approach is going to give us a kind of linear runtime complexity.

Now let's think in terms of binary search. Think about the basis on which we can divide the array for search.

We have illustrated two examples for you so that you get an idea of how we will decrease the search space.

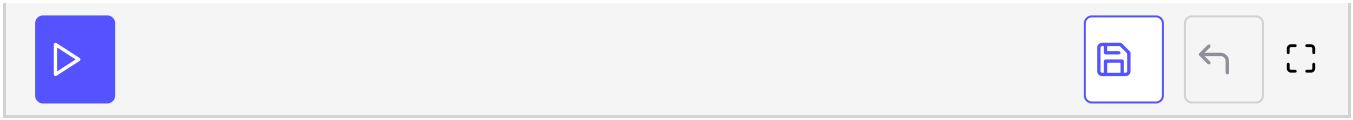


Implementation

If you understand the slides above, then the problem becomes simple to solve using a binary search. Let's find out more by looking at the implementation in Python below:

```
def find_highest_number(A):
    low = 0
    high = len(A) - 1

    # Require at least 3 elements for a bitonic sequence.
    if len(A) < 3:
        return None

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

        mid_left = A[mid - 1] if mid - 1 > 0 else float("-inf")
        mid_right = A[mid + 1] if mid + 1 < len(A) else float("inf")

        if mid_left < A[mid] and mid_right > A[mid]:
            low = mid + 1
        elif mid_left > A[mid] and mid_right < A[mid]:
            high = mid - 1
        elif mid_left < A[mid] and mid_right < A[mid]:
```

```
            return A[mid]
    return None


# Peak element is "5".
A = [1, 2, 3, 4, 5, 4, 3, 2, 1]
print(find_highest_number(A))
A = [1, 6, 5, 4, 3, 2, 1]
print(find_highest_number(A))
A = [1, 2, 3, 4, 5]
print(find_highest_number(A))
A = [5, 4, 3, 2, 1]
print(find_highest_number(A))
```



Explanation

As always in the case of Binary Search, `low` and `high` are set to `0` and `len(A) - 1` on **lines 2-3**. On **line 6**, we have an edge case that checks if the incoming array can be bitonic or not. For an array to be bitonic, it must have at least three elements. Therefore, on **line 7**, `None` is returned from the function in case `A` has less than three elements.

In the `while` loop, after calculating `mid` on **line 10**, the left and right elements to the midpoint are stored in `mid_left` and `mid_right` (**lines 12-13**). Before accessing the left and the right positions to the midpoint, we need to make sure that these positions are within the array. This is done by shorthand in Python. If you have only one statement to execute, one for if, and one for else, you can put it all on the same line which we have done on **lines 12-13**. If `mid - 1 > 0` is `True`, `mid_left` is set to `A[mid - 1]`. Otherwise, it will set to `float("-inf")`.

On the other hand, if `mid + 1 < len(A)` evaluates to `True`, `mid_right` is set equal to `A[mid + 1]`, otherwise it is set equal to `float("inf")`. From **line 15** to **line 20**, we check if `A[mid]` satisfies the peak property or not as specified in the slides. Based on the conditions discussed in the slides, we discard the left or the right side, but if the elements on both the sides of `A[mid]` are less than `A[mid]` i.e., the condition on **line 19** evaluates to `True`, we have found the peak element which is returned from the function on **line 20**.

I hope you are enjoying implementing the binary search for various problems. In the next lesson, we will learn how to find the first entry of an element in a list with duplicates. Happy learning!