educative

Binary Search

In this lesson, you will learn about the Binary Search algorithm and its implementation in Python.

We'll cover the following

In this lesson, we take a look at the well-known Binary Search algorithm in Python. **Binary Search** is a technique that allows you to search an ordered list of elements using a divide-and-conquer strategy. It's also an algorithm you'll want to know very well before you step into your technical interview. Now before we dive into discussing binary search, let's talk about linear search.

Linear Search

Linear search is when you iterate through an array looking for your target element. Essentially, it means sequentially scanning all the elements in the array one by one until you find your target element.

Let's see how we do this in Python:

```
def linear_search(data, target):
    for i in range(len(data)):
        if data[i] == target:
            return True
    return False
```

linear_search(data, target)

The `for` loop on **line 2** starts from `i` equal `0` and runs until `i` equal `len(data) - 1`. If in any iteration `data[i]` equals target, we return `True` to indicate that we have found `target` in `data`. On the other hand, if the `for` loop terminates and the condition on **line 3** never comes out to be `True`, `False` is returned from the function (**line 5**). In the worst case, we might have to scan an entire array and not find what we are looking for. Thus, the worst-case runtime of a linear search would be $O(n)$.

This is where binary search comes into play. Binary search is more efficient than the linear search. Let's find out how.

Binary Search (Iterative)

Binary search assumes that the array on which the search will take place is sorted in ascending order. In binary search, the target element is compared with the middle element of the array following which the next chunk of the array to be searched is decided. If the target matches the middle element, we are successful. Otherwise, since the array is sorted, if the target is smaller than the middle element, it could only be in the left half of the array. Alternatively, if the target is greater than the middle element, it could be in the right half of the array. So, we exclude one half of the array from the further search and repeat the same strategy to the remaining half.

Let's jump to the code below so you get a clearer idea of binary search.

```
def binary_search_iterative(data, target):
    low = 0
    high = len(data) - 1

    while low <= high:
        mid = (low + high) // 2
        if target == data[mid]:
            return True
```

```
 8             return True
 9         elif target < data[mid]:
10             high = mid - 1
11         else:
12             low = mid + 1
13     return False
```

binary_search_iterative(data, target)

`data` and `target` are the input parameters to `binary_search_iterative` function. `data` is the array in which we are searching, and `target` is the value that we are searching for. On **lines 2-3**, `low` and `high` are initialized to `0` and `len(data) - 1` respectively. Based on the assumption that `data` is a sorted list, `low` and `high` have been assigned as the indices for the minimum and the maximum values in `data`.

Next, the `while` loop on **line 5** will run until `low` is less than or equal to `high`. On **line 6**, `mid` is calculated by dividing the sum of `low` and `high` by `2` and getting the floored value because of the `//` operator. As specified before, `target` will be compared to the middle element, which is what happens on **line** 7. If `target` is equal to `data[mid]` (the middle element), it implies `target` exists in `data` and `True` is returned from the function as an indication. On the other hand, if `target` is less than the middle element, it means that `target` is somewhere in the first half of the array as the array is sorted. Therefore, we set `high` to `mid - 1`, i.e., the upper bound of the chunk of the array to be searched will be at a position to the left of `mid`. In contrast, if `target` is greater than `data[mid]`, `target` must be in the second half of the array, so the lower bound (`low`) is set to `mid + 1`.

In general, we keep dividing the array into halves in the binary search instead of iterating through all the elements to search for the target element. This implies that it takes $O(logn)$ steps to divide into halves until we reach a single element. As a result, the worst-case time complexity of a binary search is $O(logn)$.

Binary Search (Recursive)

Now that we have implemented binary search iteratively, let's go ahead and learn how to implement the algorithm recursively:

```
1  def binary_search_recursive(data, target, low, hi
2      if low > high:
3          return False
4      else:
5          mid = (low + high) // 2
6          if target == data[mid]:
7              return True
8          elif target < data[mid]:
9              return binary_search_recursive(data, t
10         else:
11             return binary_search_recursive(data, t
```

binary_search_recursive(data, target, low, high)

In the recursive approach, in addition to `data` and `target`, `low` and `high` are also passed as input parameters to `binary_search_recursive`. This is to help us code our base case. The base case for this recursive function will be when `low` becomes greater than `high`. If the base case turns out to be `True`, `False` is returned from the function to end the recursive calls (**lines 2-3**). On the other hand, if `low` is less than or equal to `high`, execution jumps to **line 5** where `mid` is calculated in the same way as in the iterative function. If `target` is equal to `data[mid]`, `True` is returned (**line** 7). If not, then the condition on **line 8** is evaluated. If `target` is less than `data[mid]`, we make a recursive call to `binary_search_recursive` and pass `mid - 1` which is the `high` in the scope of the next recursive call. This will reduce the search span as it will be halved with each recursive call. Similarly, if `target` is greater than `data[mid]`, `low` needs to be adjusted and so we pass `mid + 1` to the recursive call on **line 11** which is `low` in the next recursive call.

We keep dividing the array into halves with recursive calls until the base case is reached. As every recursive call takes constant time, the worst-case time complexity of the recursive approach is also $O(logn)$.

Below is an executable code with all the functions that we have implemented in this lesson in a sample test case:

```
        mid = (low + high) // 2
        if target == data[mid]:
            return True
        elif target < data[mid]:
            high = mid - 1
        else:
            low = mid + 1
    return False

# Recursive Binary Search
def binary_search_recursive(data, target, low, hi
    if low > high:
        return False
    else:
        mid = (low + high) // 2
        if target == data[mid]:
            return True
        elif target < data[mid]:
            return binary_search_recursive(data, t
        else:
            return binary_search_recursive(data, t


data·=·[2,4,5,7,8,9,12,14,17,19,22,25,27,28,33,37
target·=·37

print(binary_search_recursive(data,·target,·0,·le
print(binary_search_iterative(data,·target))
```

Output 0.88s

```
True
True
```

In the next lesson, we look at a problem and solve it using a binary search.

Back

Quiz

Next

Find the Closest Number

Mark as Completed

Report an Issue

Ask a Question (https://discuss.educative.io/tag/binary-search__binary-search__data-structures-and-algorithms-in-python)

educative

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
```

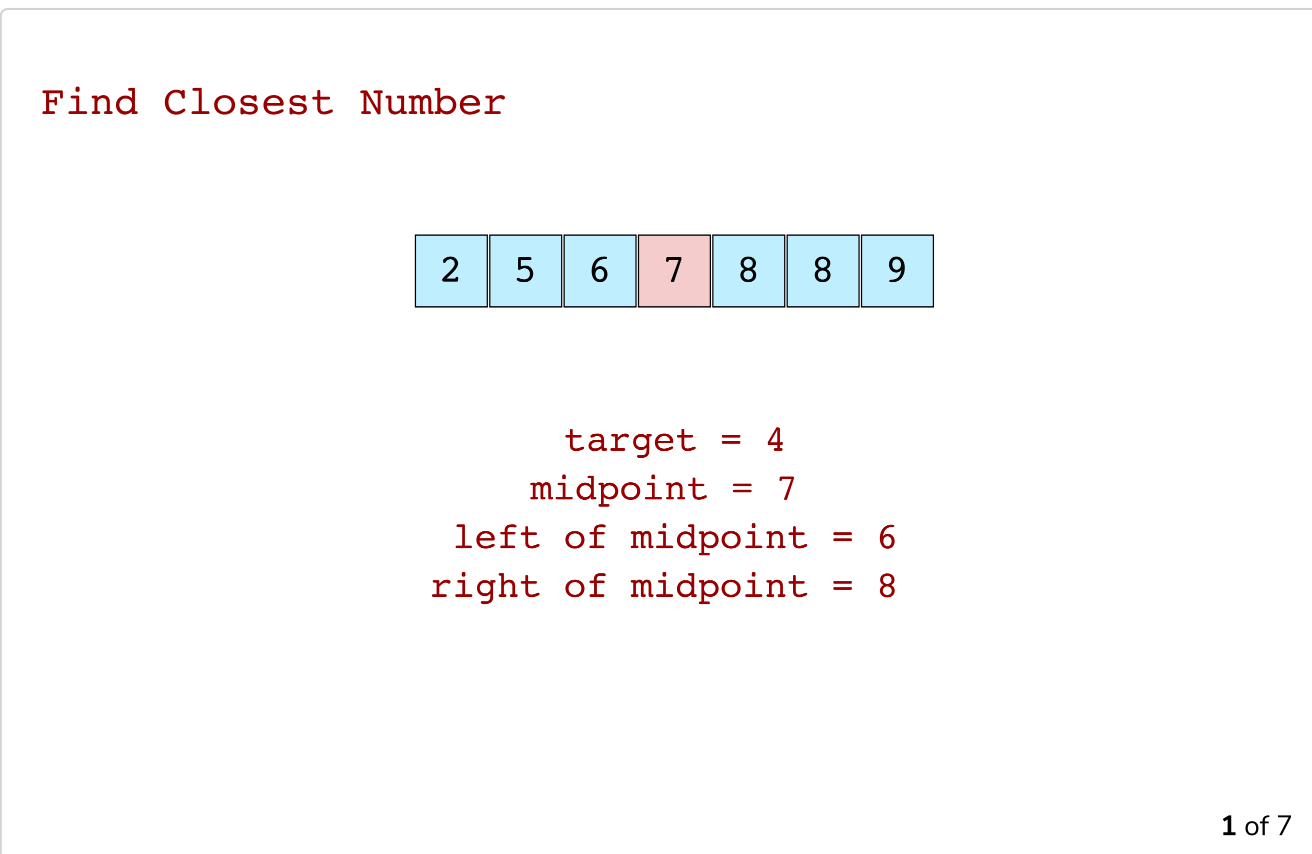
Example 2

```
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 : Calculating difference

2	5	6	7	8	8	9

target = 4
midpoint = 7
left of midpoint = 6 => 6 - 4 = 2
right of midpoint = 8 => 8 - 4 = 4

2 of 7

Find Closest Number: Eliminating Right Side

2	5	6	7	8	8	9

target = 4
midpoint = 5
left of midpoint = 2
right of midpoint = 6

3 of 7

Find Closest Number: Calculating Difference

2	5	6	7	8	8	9

target = 4
midpoint = 5
difference with left of midpoint = 2 - 4 = |-2|= 2
difference with right of midpoint = 6 - 4 = 2

4 of 7

Find Closest Number

2	5	6	7	8	8	9

target = 4
midpoint = 2
left of midpoint = null
right of midpoint = 5

5 of 7

Find Closest Number

2	5	6	7	8	8	9

target = 4
midpoint = 2
left of midpoint = null
difference with right of midpoint = 5 - 4 = 1

6 of 7

Find Closest Number

2	5	6	7	8	8	9

target = 4
midpoint = 2
left of midpoint = null
difference with right of midpoint = 5 - 4 = 1

5 is closest to the midpoint as
1 is the least difference
we have encountered so far.

7 of 7

Implementation

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

```
            min_diff_left = abs(A[mid - 1] - targe

        # 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 c
        # 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
········#·number·to·it·is·itself.·Return·the·numbe
        else:
            return A[mid]
    return closest_num


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

Output 1.55s

```
9
5
```

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!

Back

Binary Search

Next

Find Fixed Number

Mark as Completed

Report an Issue

Ask a Question (https://discuss.educative.io/tag/find-the-closest-number__binary-search__data-structures-and-algorithms-in-python)

educative

Find Fixed Number

In this lesson, you will learn how to find a fixed number in a list using a binary search in Python.

We'll cover the following

In this lesson, we will be solving the following problem:

Given an array of n distinct integers sorted in ascending order, write a function that returns a **fixed point** in the array. If there is not a fixed point, return `None`.

> A fixed point in an array `A` is an index `i` such that `A[i]` is equal to `i`.

The naive approach to solving this problem is pretty simple. You iterate through the list and check if each element matches its index. If you find a match, you return that element. Otherwise, you return `None` if you don't find a match by the end of the `for` loop. Have a look at the code below:

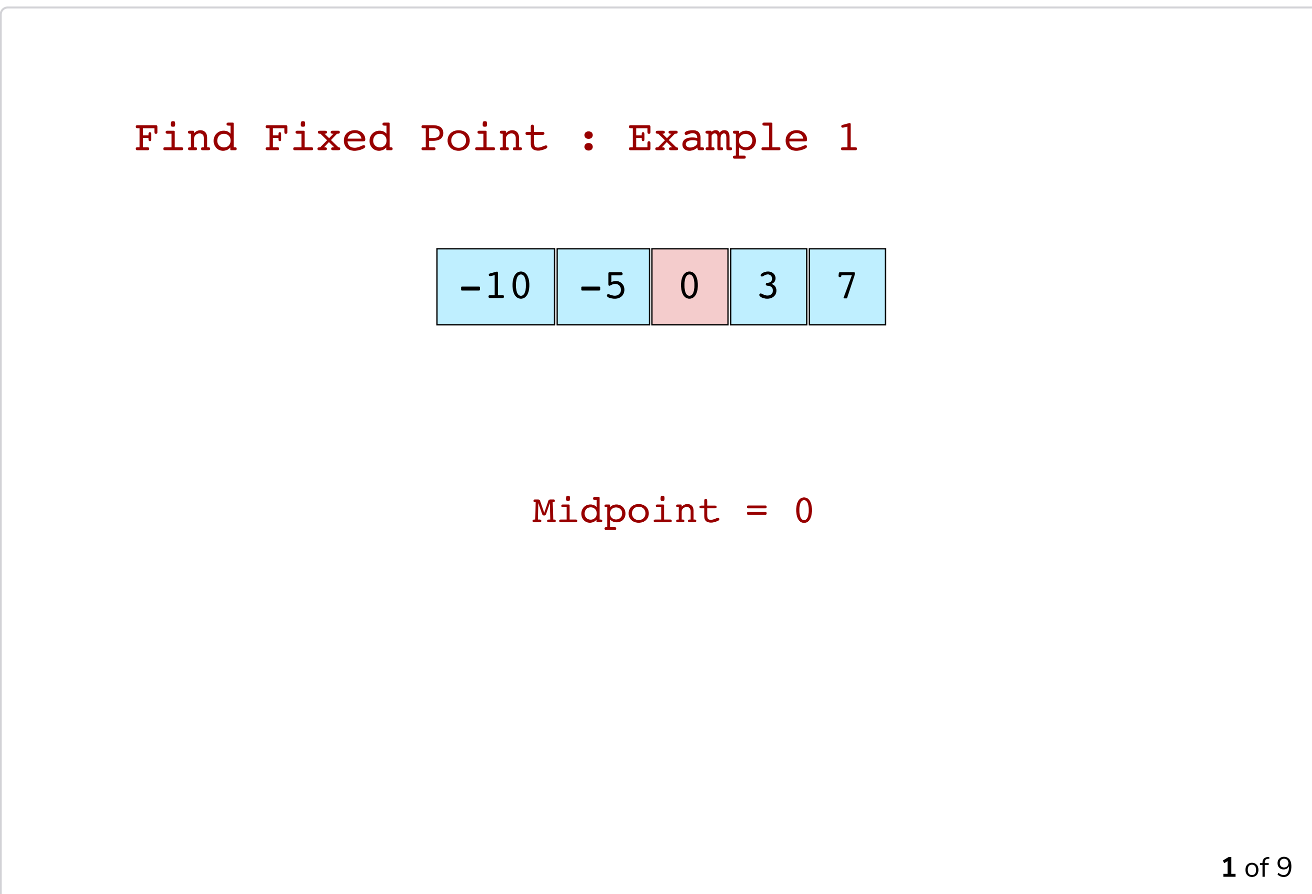
```
# Time Complexity: O(n)
# Space Complexity: O(1)
def find_fixed_point_linear(A):
    for i in range(len(A)):
        if A[i] == i:
            return A[i]
    return None
```

```
find_fixed_point_linear(A)
```

As the entire list is traversed once to find the fixed point, spending constant time on each element, the time complexity for the linear implementation above is $O(n)$. As we haven't used any additional space in the implementation above, the space complexity is $O(1)$. Now we need to think about how we can improve the solution above. We can use the following two facts to our advantage:

- The list is sorted.
- The list contains *distinct* elements.

Let's look at the slides below to get a rough idea of how we have taken advantage of the above facts.



Find Fixed Point : Example 1

-10	-5	0	3	7

Midpoint = 0
If 0 is less than its index (2),
it implies every element to the left
of the midpoint will be less than
its index. This is because the array is
sorted and element are decreasing to the
left of the midpoint.

2 of 9

Find Fixed Point : Example 1

-10	-5	0	3	7

Midpoint = 0
We discard the portion to the left
of the midpoint.

3 of 9

Find Fixed Point : Example 2

0	2	5	8	17

Midpoint = 5

4 of 9

Find Fixed Point : Example 2

0	2	5	8	17

Midpoint = 5
If 5 is greater than its index (2), it implies every element to the right of the midpoint will be greater than its index. This is because the array is sorted and element are increasing to the right of the midpoint.

5 of 9

Find Fixed Point : Example 2

0	2	5	8	17

Midpoint = 5
We discard the portion to
the right of the mipoint.

6 of 9

Find Fixed Point : Example 3

0	2	5	5	5	5

Midpoint = 5
Now consider this possibility where
elements are not distinct.

7 of 9

Find Fixed Point : Example 3

0	2	5	5	5	5

Midpoint = 5
Even if 5 is greater than its index (2), we can not guarantee that every element to the right of the midpoint will be greater than its index. This is because the elements are not distinct and we may have a fixed point in the right portion of the array.

8 of 9

Find Fixed Point : Example 3

0	2	5	5	5	5

Midpoint = 5
However, as the problem clearly states that the elements are distinct, we won't encounter such a possibility.

9 of 9

Implementation

If you have gone through the slides, the implementation must be pretty clear to you. Let's jump to the implementation in Python:

```
1  # Time Complexity: O(log n)
2  # Space Complexity: O(1)
3  def find_fixed_point(A):
4      low = 0
5      high = len(A) - 1
6
7      while low <= high:
8          mid = (low + high)//2
9
10         if A[mid] < mid:
11             low = mid + 1
12         elif A[mid] > mid:
13             high = mid - 1
14         else:
15             return A[mid]
16     return None
```

find_fixed_point(A)

Explanation

On **lines 4-5**, we define `low` and `high` in the same way we have always defined them for Binary Search. The next few lines **(lines 7-8)** are also the same as the code in a binary search. On **line 10**, we check if `A[mid]` is less than `mid` to decide which portion of the array to discard in further search. If the condition on **line 10** evaluates to `True`, execution jumps to **line 11** where `low` is set to `mid+1` to discard the portion to the left of `mid`. However, if the condition on **line 10** evaluates to `False`, the condition on **line 12** is evaluated. If `A[mid]` is greater than `mid`, i.e., `high` is set to `mid-1` to disregard the portion to the right of the midpoint. If both the conditions on

line 10 and **line 11** are `False`, it implies that `A[mid]` is equal to `mid`. We have found a fixed point! In this case, `A[mid]` is returned from the function on **line 15**. To cater to the case if there is no fixed point in the array, we return `None` on **line 16** after the `while` loop terminates.

As we have employed a binary search to write the above code, the time complexity for the code above is $O(logn)$ while the space complexity is $O(1)$.

The solution above was pretty straightforward. You can run the linear and binary search solution in the code widget below.

```python
# Time Complexity: O(n)
# Space Complexity: O(1)
def find_fixed_point_linear(A):
    for i in range(len(A)):
        if A[i] == i:
            return A[i]
    return None


# Time Complexity: O(log n)
# Space Complexity: O(1)
def find_fixed_point(A):
    low = 0
    high = len(A) - 1

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

        if A[mid] < mid:
            low = mid + 1
        elif A[mid] > mid:
            high = mid - 1
        else:
            return A[mid]
    return None

# Fixed point is 3:
A1 = [-10, -5, 0, 3, 7]
```

```
Output                                                        1.21s

 Linear Approach
 [-10, -5, 0, 3, 7]
 3
 [0, 2, 5, 8, 17]
 0
 [-10, -5, 3, 4, 7, 9]
 None
 Binary Search Approach
 [-10, -5, 0, 3, 7]
```

You'll hopefully be getting the hang of binary search by now. Let's solve another problem using binary search in the next lesson.

← Back
Find the Closest Number

Next →
Find Bitonic Peak

Mark as Completed

Report an Issue

Ask a Question
(https://discuss.educative.io/tag/find-fixed-number__binary-search__data-structures-and-algorithms-in-python)

educative

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

- Implementation
- Explanation

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.

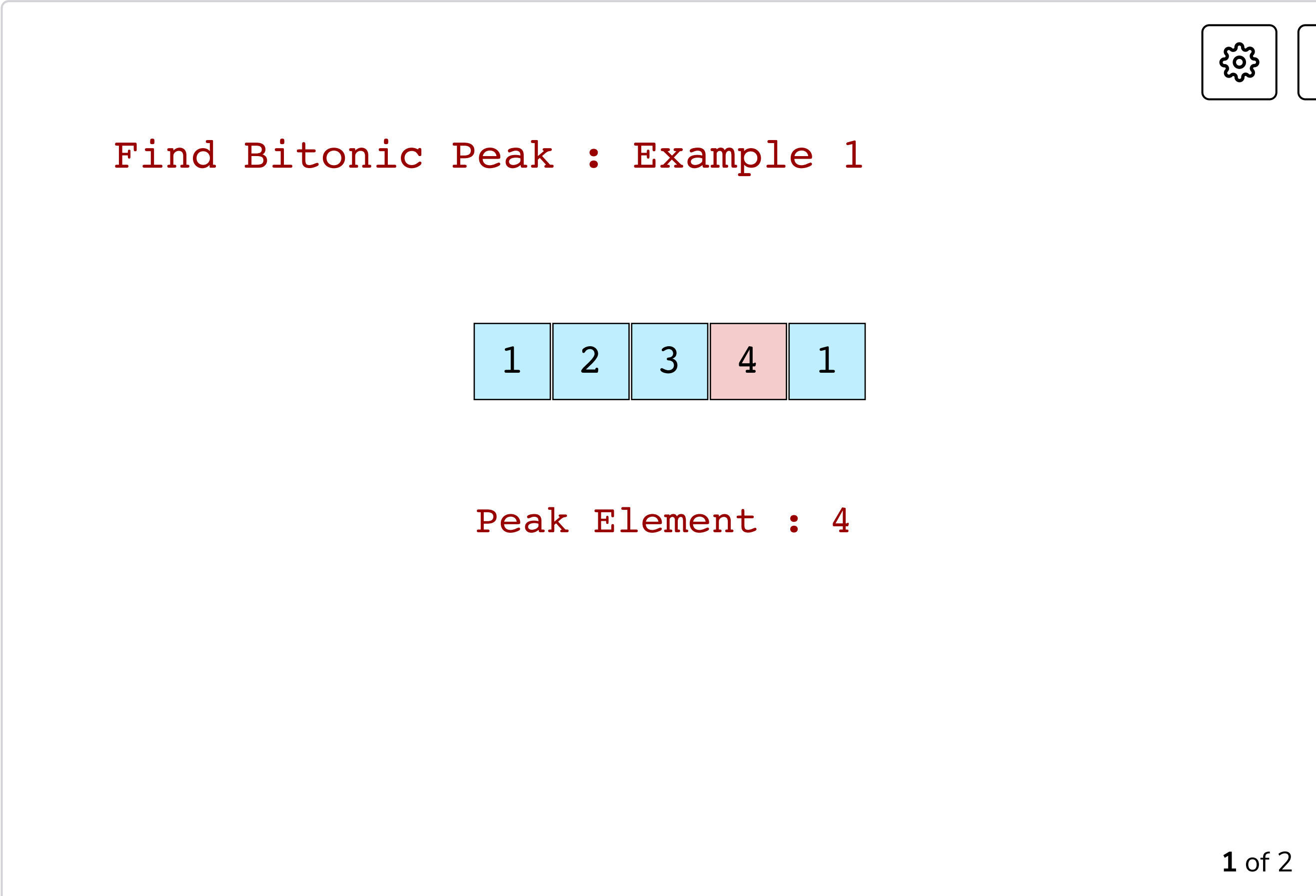
For example:

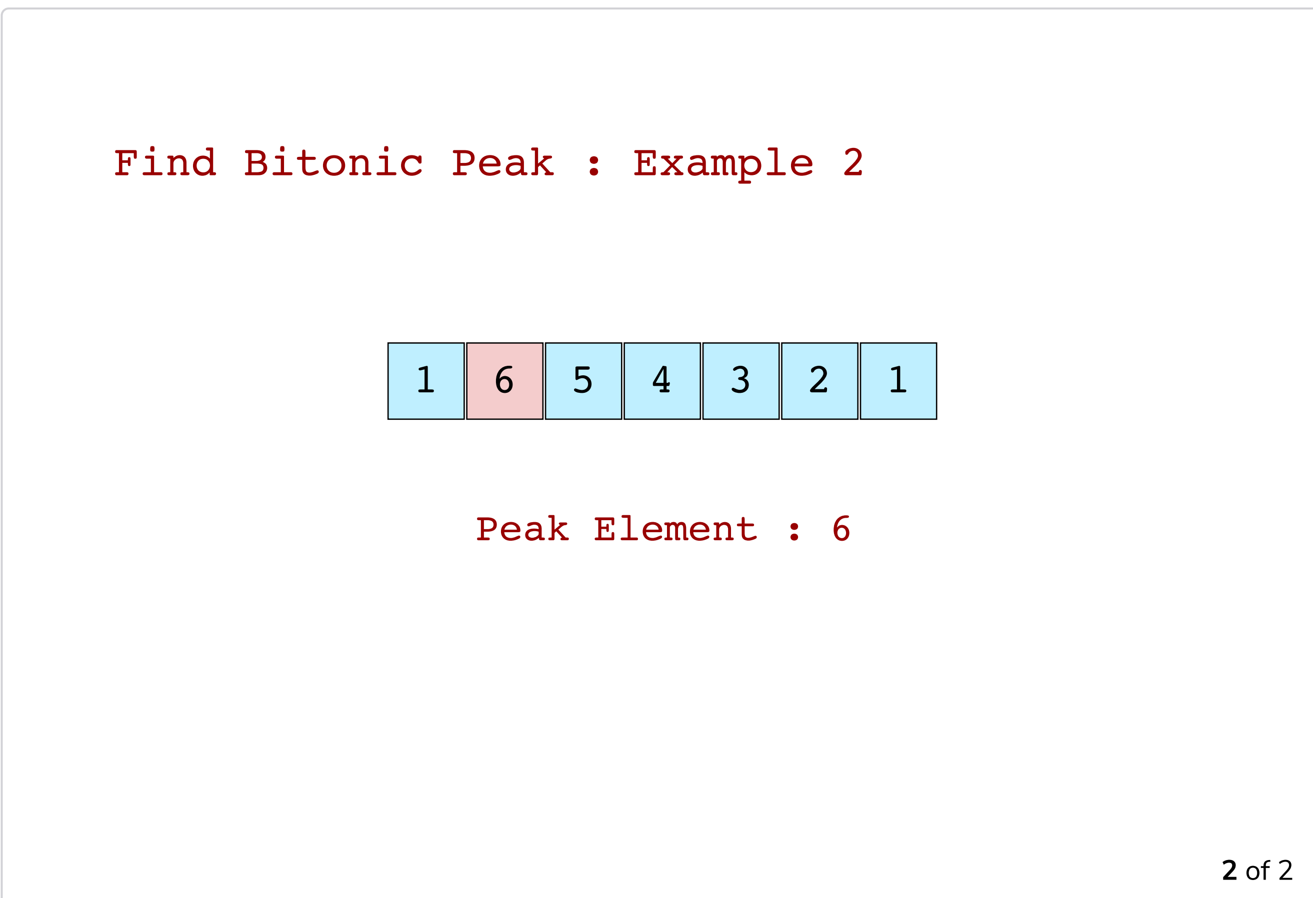
```
1, 2, 3, 4, 5, 4, 3, 2, 1
```

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.


Find Bitonic Peak : Example 1
1 2 3 4 1
Peak Element : 4
1 of 2


Find Bitonic Peak : Example 2
1 6 5 4 3 2 1
Peak Element : 6
2 of 2

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.

Find Bitonic Peak : Example 1

1	2	3	4	1

Midpoint = 3

1 of 6

Find Bitonic Peak : Example 1

1	2	3	4	1

Midpoint = 3

If Mid_Left < Midpoint and Midpoint < Mid_Right
(2 < 3 and 3 < 4),
then the peak element should be to the
right of the midpoint as
elements will decrease in value to the
left of midpoint.

2 of 6

Find Bitonic Peak : Discarding Left Side

1	2	3	4	1

Midpoint = 3
If Mid_Left < Midpoint and Midpoint < Mid_Right
(2 < 3 and 3 < 4),
then the peak element should be to the
right of the midpoint as
elements will increase in value to the
right of midpoint.

3 of 6

Find Bitonic Peak : Example 2

1	6	5	4	3	2	1

Midpoint = 4

4 of 6

Find Bitonic Peak : Example 2

1	6	5	4	3	2	1

Midpoint = 4
If Mid_Left > Midpoint and Midpoint > Mid_Right
(5 > 4 and 4 > 3),
then the peak element should be to the
left of the midpoint as
elements will decrease in value to the
right of midpoint.

5 of 6

Find Bitonic Peak : Discarding Right Side

1	6	5	4	3	2	1

Midpoint = 4
If Mid_Left > Midpoint and Midpoint > Mid_Right
(5 > 4 and 4 > 3),
then the peak element should be to the
left of the midpoint as
elements will decrease in value to the
right of midpoint.

6 of 6

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:

```

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

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

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

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

Output 1.55s

```
5
6
None
5
```

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!

← **Back**

Find Fixed Number

Next →

Find First Entry in List with Duplicates

Mark as Completed

Report an Issue

Ask a Question (https://discuss.educative.io/tag/find-bitonic-peak__binary-search__data-structures-and-algorithms-in-python)

educative

Find First Entry in List with Duplicates

In this lesson, you will learn how to find the first entry in a list with duplicates using a binary search in Python.

We'll cover the following

In this lesson, we will be writing a function that takes an array of sorted integers and a key and returns the index of the first occurrence of that key from the array.

For example, for the array:

```
[-14, -10, 2, 108, 108, 243, 285, 285, 285, 401]
```

with

```
target = 108
```

the algorithm would return `3`, as the first occurrence of `108` in the above array is located at index `3`.

The most naive approach to solving this problem is to loop through each element in the array. If you stumble upon the target element, it will be the first occurrence because the array is sorted. Otherwise, if the number does not exist in the array, we return `None` to indicate that the number is not present in a list.

```
def find(A,target):
  for i in range(len(A)):
    if A[i] == target:
      return i
    return None
```

The above code will work, but it will take linear time to complete as the time it takes for the code to execute is proportional to the size of the array for the worst-case.

Let's go ahead and apply a binary search to make our lives easier to solve this problem. We can reduce the problem from linear Big $O(n)$ to $O(logn)$ where n is the size of the array. In our current problem, we have non-distinct entries in sorted order, so we need to tweak the binary search algorithm to solve this variance of the problem.

Now we will tweak the binary search so that we can redefine the high point of the array based on whether the entry to the left of the target element is the same or not.

Implementation

Check out the code below:

```
def find(A, target):
    low = 0
    high = len(A) - 1

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

        if A[mid] < target:
            low = mid + 1
        elif A[mid] > target:
            high = mid - 1
        else:
            if mid - 1 < 0:
                return mid
            if A[mid - 1] != target:
```

```
15              if A[mid - 1] != target:
16                  return mid
17              high = mid - 1
18
19  A = [-14, -10, 2, 108, 108, 243, 285, 285, 285, 4
20  target = 108
21  x = find(A, target)
22  print(x)
```

Output 1.4s

```
3
```

Explanation

The code that we have written above is almost the same as the original binary search algorithm with some slight variation in statements on **lines 12-17**. The binary search will help us find the target element, but our problem is to find the first occurrence of that target element. The code above updates the lower and upper bounds for our search space according to the result of the comparison between the value of `target` and the value of the midpoint (`A[mid]`) **(lines 8 -11)** just as in the original Binary Search. The execution jumps to **line 13** when `target` is equal to `A[mid]` . Since the array is sorted in ascending order, our target is either the middle element or an element on its left. Now we have to make sure that we return the *first occurrence* from the function.

On **line 13**, we check if `mid - 1` is less than `0` , this is an edge case we would deal with if the *first occurrence* is on the first index. Next, we check if the element to the left of `A[mid]` , i.e., `A[mid - 1]` is the target element or not

(line 15). If it isn't, the condition on **line 15** is `True`, and this implies that `A[mid]` is the *first occurrence*, so we return `mid` on **line 16**. However, if the element to the left is also the same as the target element, we update `high` to `mid - 1`. In this way, we condense our search space to find the *first occurrence* of the target which will be to the left of the midpoint.

One other way of thinking is to combine the linear approach and the binary search approach to solving this problem. You find your target element using binary search, and once you find it, you can keep going to the left of the array until you hit an element where the left of that element is not the element that we're looking for. This approach will work, but in the worst case you can have an element or an array consisting of all elements of the same number, and thus this approach will boil down to giving you the same runtime as the initial naive approach that we came up with previously.

I hope you are clear with everything we have learned so far. In the next lesson, we will have a look at Python's Bisect method. Stay tuned!

Back
Find Bitonic Peak

Next
Python's Bisect Method

Mark as Completed

Report an Issue

Ask a Question
(https://discuss.educative.io/tag/find-first-entry-in-list-with-duplicates__binary-search__data-structures-and-algorithms-in-python)

Python's Bisect Method

In this lesson, you will learn about the bisect module in Python.

We'll cover the following

In this lesson, we will learn about a function that takes an array of sorted integers and a key and returns the index of the first occurrence of that key from the array.

For example, for the array:

```
[-14, -10, 2, 108, 108, 243, 285, 285, 285, 401]
```

with

```
target = 108
```

the function would return `3`, as the first occurrence of `108` in the above array is located at index `3`.

We introduce to you: the `bisect` module in Python! Bisect is a built-in binary search method in Python. It can be used to search for an element in a sorted list. Let's see how we make use of different methods provided by the `bisect` module.

bisect_left()

The `bisect_left` function finds the index of the target element. In the event where duplicate entries are satisfying the target element, the `bisect_left` function returns the left-most occurrence. The input parameters to the method are the sorted list and the target element to be searched.

Have a look at the examples below:

```
# Import allows us to make use of the bisect modu
import bisect

# This sorted list will be used throughout this le
# to showcase the functionality of the "bisect" me
A = [-14, -10, 2, 108, 108, 243, 285, 285, 285, 40

# -10 is at index 1
print(bisect.bisect_left(A, -10))

# First occurrence of 285 is at index 6
print(bisect.bisect_left(A, 285))
```

```
Output                                                0.74s

1
6
```

bisect_right()

The `bisect_right` function returns the insertion point which comes after, or to the right of, any existing entries of the target element in the list. It takes in a sorted list as the first parameter and the target element to be searched as the second parameter.

```
# Import allows us to make use of the bisect modu
import bisect

# This sorted list will be used throughout this le
# to showcase the functionality of the "bisect" me
A = [-14, -10, 2, 108, 108, 243, 285, 285, 285, 4(

# Index position to right of -10 is 2.
print(bisect.bisect_right(A, -10)) 

# Index position after last occurrence of 285 is 9
print(bisect.bisect_right(A, 285))
```

Output 0.8s

```
2
9
```

`bisect()`

There is also just a regular default `bisect` function. This function is equivalent to `bisect_right` and takes a sorted list and the target element to be searched as input parameters:

```
# Import allows us to make use of the bisect modul
import bisect

```

```
# This sorted list will be used throughout this le
# to showcase the functionality of the "bisect" me
A = [-14, -10, 2, 108, 108, 243, 285, 285, 285, 40

# Index position to right of -10 is 2. (Same as bi
print(bisect.bisect(A, -10))

# Index position after last occurrence of 285 is 9
print(bisect.bisect(A, 285))
```

Output 0.71s

```
2
9
```

`insort_left()` and `insort_right`

Given that the list `A` is sorted, it is possible to insert elements into `A` so that the list remains sorted. Functions `insort_left` and `insort_right` behave in a similar way to `bisect_left` and `bisect_right`, only the *insort* functions insert at the index positions. The input parameters to the method are the sorted list and the element to be inserted at a position so that the list remains sorted.

```
# Import allows us to make use of the bisect modu
import bisect

# This sorted list will be used throughout this le
# to showcase the functionality of the "bisect" me
A = [-14, -10, 2, 108, 108, 243, 285, 285, 285, 40


print(A)
bisect.insort_left(A, 108)
```

```
print(A)

bisect.insort_right(A, 108)
print(A)
```

Output 0.84s

```
[-14, -10, 2, 108, 108, 243, 285, 285, 285, 401]
[-14, -10, 2, 108, 108, 108, 243, 285, 285, 285, 401]
[-14, -10, 2, 108, 108, 108, 108, 243, 285, 285, 285, 401]
```

I encourage you to know how to write your own binary search because that's going to be very useful, especially in the context of an interview. However, if you want to apply the binary search in Python, then you can utilize this module.

In the context of an interview, if you mention that you know how to use the `bisect` module, that could be perhaps a feather in your cap because it shows a bit of maturity with the language and it shows that you know some of lesser-known features of Python.

By now, you have had good practice with solving problems using binary search, so let's get ready for some challenges in the next few lessons!

← Back

Find First Entry in List with Duplicates

Next →

Exercise: Integer Square Root

Mark as Completed

Ask a Question

Report an Issue

(https://discuss.educative.io/tag/pythons-bisect-method__binary-search__data-structures-and-algorithms-in-python)

Exercise: Integer Square Root

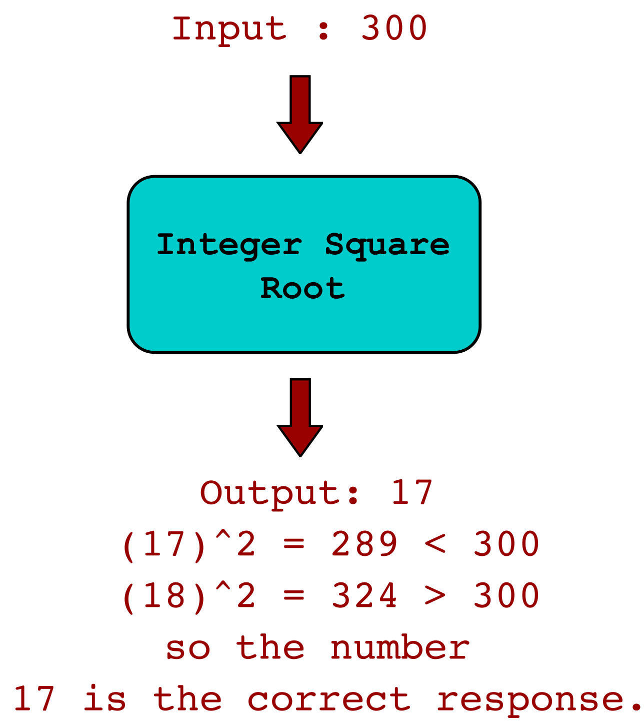
Challenge yourself with an exercise in which you'll have to return the largest integer whose square is less than or equal to the given integer.

We'll cover the following

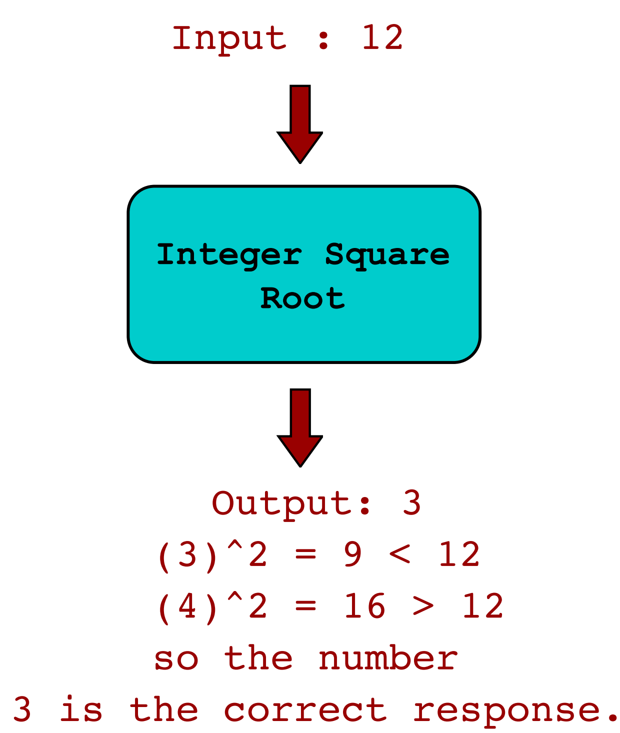
Problem

You are required to write a function that takes a non-negative integer, `k`, and returns the largest integer whose square is less than or equal to the specified integer `k`.

Let's have a look at some examples:


Input : 300
Integer Square Root
Output: 17
(17)^2 = 289 < 300
(18)^2 = 324 > 300
so the number
17 is the correct response.

1 of 2


Input : 12
Integer Square Root
Output: 3
(3)^2 = 9 < 12
(4)^2 = 16 > 12
so the number
3 is the correct response.

2 of 2

Coding Time!

Your task is to return the largest integer whose square is less than or equal to the `k` from the function `integer_square_root(k)` given in the code widget below. The input parameter `k` is a non-negative integer. Make use of a binary search strategy in your solution.

Good luck!

```
def integer_square_root(k):
    low = 0
    high = k
    root = 0
    while low <= high:
        mid = (low + high)//2
        if mid*mid == k or mid == root:
            return mid
        elif mid*mid < k:
            low = mid
            if mid > root:
                root = mid
        else:
            high = mid
    return root

```

Show Results Show Console

0.71s

4 of 4 Tests Passed

Result	Input	Expected Output	Actual Output	Reason
✓	integer_square_root(300)	17	17	Succeeded

Result	Input	Expected Output	Actual Output	Reason
✓	integer_square_root(12)	3	3	Succeeded
✓	integer_square_root(1000)	31	31	Succeeded
✓	integer_square_root(625)	25	25	Succeeded

← Back

Python's Bisect Method

Next →

Solution Review: Integer Square Root

Mark as Completed

Report an Issue

Ask a Question (https://discuss.educative.io/tag/exercise-integer-square-root__binary-search__data-structures-and-algorithms-in-python)

Solution Review: Integer Square Root

This lesson contains the solution review for the challenge to find the largest integer whose square is less than or equal to the given integer.

We'll cover the following

First of all, let's repeat the problem statement.

You are required to write a function that takes a non-negative integer, `k`, and returns the largest integer whose square is less than or equal to the specified integer `k`.

Algorithm

The naive approach to solve this problem is to start from `1` and check the square of every number up until `k`. Have a look at the slides below:

input : 12

Let's check the squares of all numbers starting from 1 till the given input.

1 of 7

input : 12

Let's check the squares of all numbers starting from 1 till the given input.

1^2 = 1

2 of 7

input : 12

Let's check the squares of all numbers starting from 1 till the given input.

1^2 = 1
2^2 = 4

3 of 7

input : 12

Let's check the squares of all numbers starting from 1 till the given input.

1^2 = 1
2^2 = 4
3^2 = 9

4 of 7

input : 12

Let's check the squares of all numbers starting from 1 till the given input.

1^2 = 1
2^2 = 4
3^2 = 9
4^2 = 16

5 of 7

input : 12

Let's check the squares of all numbers starting from 1 till the given input.

1^2 = 1
2^2 = 4
3^2 = 9
4^2 = 16

(3)^2 = 9 < 12
(4)^2 = 16 > 12
so the number
3 is the correct response.

6 of 7

input : 12

Let's check the squares of all numbers starting from 1 till the given input.

1^2 = 1
2^2 = 4
3^2 = 9
4^2 = 16

(3)^2 = 9 < 12
(4)^2 = 16 > 12
so the number
3 is the correct response.
This approach has a run time complexity of O(k).

7 of 7

Now we want to improve things. Let's think in terms of binary search. One thing that we can note from the above example is that if we are on integer 3, which squares up to 9, decreasing the integer 3 will not take us anywhere near 12. On the other hand, if we are on integer 4, which squares up to 16, increasing the number will not take us closer to 12. We can make use of this observation and tweak the binary search algorithm to solve our problem efficiently. This observation enables us to reduce our search span.

Implementation

Let's have a look at the code below:

```
def integer_square_root(k):

    low = 0
    high = k
```

```
5
6       while low <= high:
7           mid = (low + high) // 2
8           mid_squared = mid * mid
9
10          if mid_squared <= k:
11              low = mid + 1
12          else:
13              high = mid - 1
14      return low - 1
15
16  k = 300
17  print(integer_square_root(k))
```

Output 0.82s

```
17
```

Explanation

On **lines 3-4**, `low` and `high` are initialized to `0` and `k`. The `while` loop on **line 6** will terminate when `low` becomes greater than `high`. In the next line, we calculate `mid` as we have always done while implementing the binary search. Additionally, we take the square of `mid` and store it in `mid_squared` on **line 8**. Now we need to compare `mid_squared` with `k`. If `mid_squared` is less than or equal to `k`, we have to discard all the numbers less than `mid`. Therefore, we set `low` equal to `mid + 1`. On the other hand, if `mid_squared` is greater than `k`, then all the numbers greater than `mid` will not be useful in our search, so we set `high` equal to `mid-1` (**line 13**). Finally, after the `while` loop terminates, `low - 1` will be the answer we are looking for, i.e., the largest integer whose square is less than or equal to `k`.

I hope you enjoyed this challenge. We have another waiting for you in the next lesson. All the best!

Back

Next

Exercise: Integer Square Root

Exercise: Cyclically Shifted Array

Mark as Completed

Report an Issue

Ask a Question (https://discuss.educative.io/tag/solution-review-integer-square-root__binary-search__data-structures-and-algorithms-in-python)

Exercise: Cyclically Shifted Array

Challenge yourself with an exercise in which you'll have to return the index of the smallest number in a cyclically shifted array.

We'll cover the following

Problem

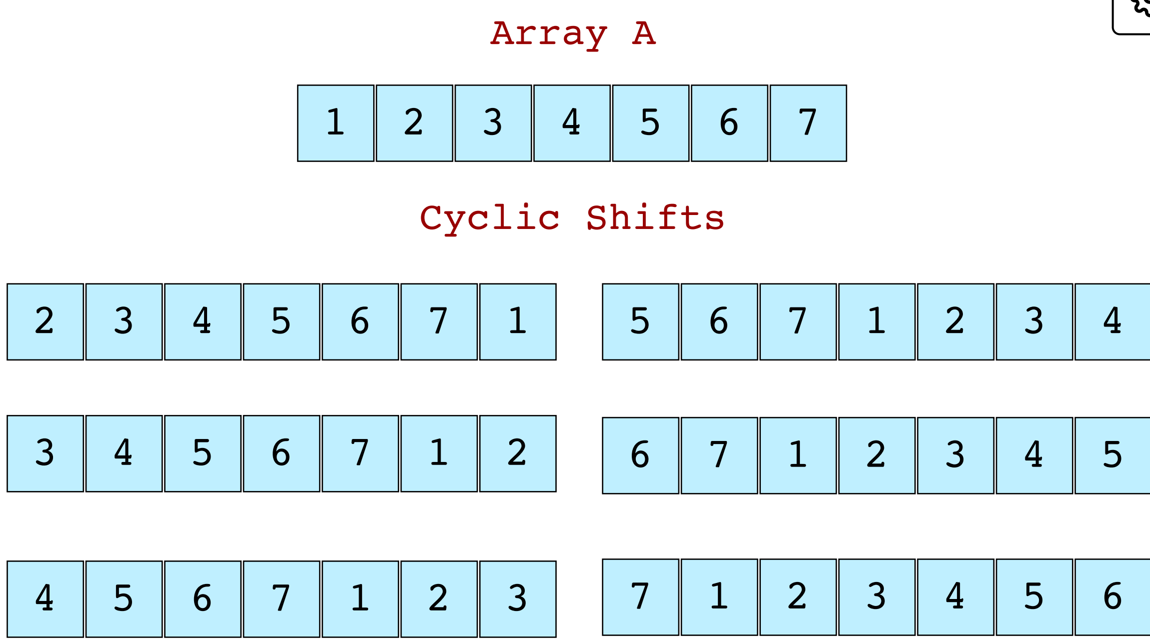
You are required to write a function that determines the index of the smallest element of the cyclically shifted array.

An array is **"cyclically shifted"** if it is possible to shift its entries cyclically so that it becomes sorted.

The following list is an example of a cyclically shifted array:

```
A = [4, 5, 6, 7, 1, 2, 3]
```

Below are all the possible cyclic shifts of an array:



Coding Time!

Your task is to return the index of the smallest number in the list `A` (cyclically shifted array) from the function `find(A)` given in the code widget below.

Good luck!

```
                min_index = mid
            low = mid + 1
    return min_index
```

Show Results Show Console

0.78s

7 of 7 Tests Passed

Result	Input	Expected Output	Actual Output	Reason
✓	find([4, 5, 6, 7, 1, 2, 3])	4	4	Succeeded
✓	find([6, 7, 1, 2, 3, 4, 5])	2	2	Succeeded
✓	find([7, 1, 2, 3, 4, 5, 6])	1	1	Succeeded
✓	find([1, 2, 3, 4, 5, 6, 7])	0	0	Succeeded
✓	find([3, 4, 5, 6, 7, 1, 2])	5	5	Succeeded
✓	find([2, 3, 4, 5, 6, 7, 1])	6	6	Succeeded
✓	find([5, 6, 7, 1, 2, 3, 5])	3	3	Succeeded

← Back

Solution Review: Integer Square Root

Next →

Solution Review: Cyclically Shifted Arr...

Mark as Completed

Report an Issue

Ask a Question (https://discuss.educative.io/tag/exercise-cyclically-shifted-array__binary-search__data-structures-and-algorithms-in-python)

Solution Review: Cyclically Shifted Array

This lesson contains the solution review for the challenge to find the index of the smallest number in a cyclically shifted array.

We'll cover the following

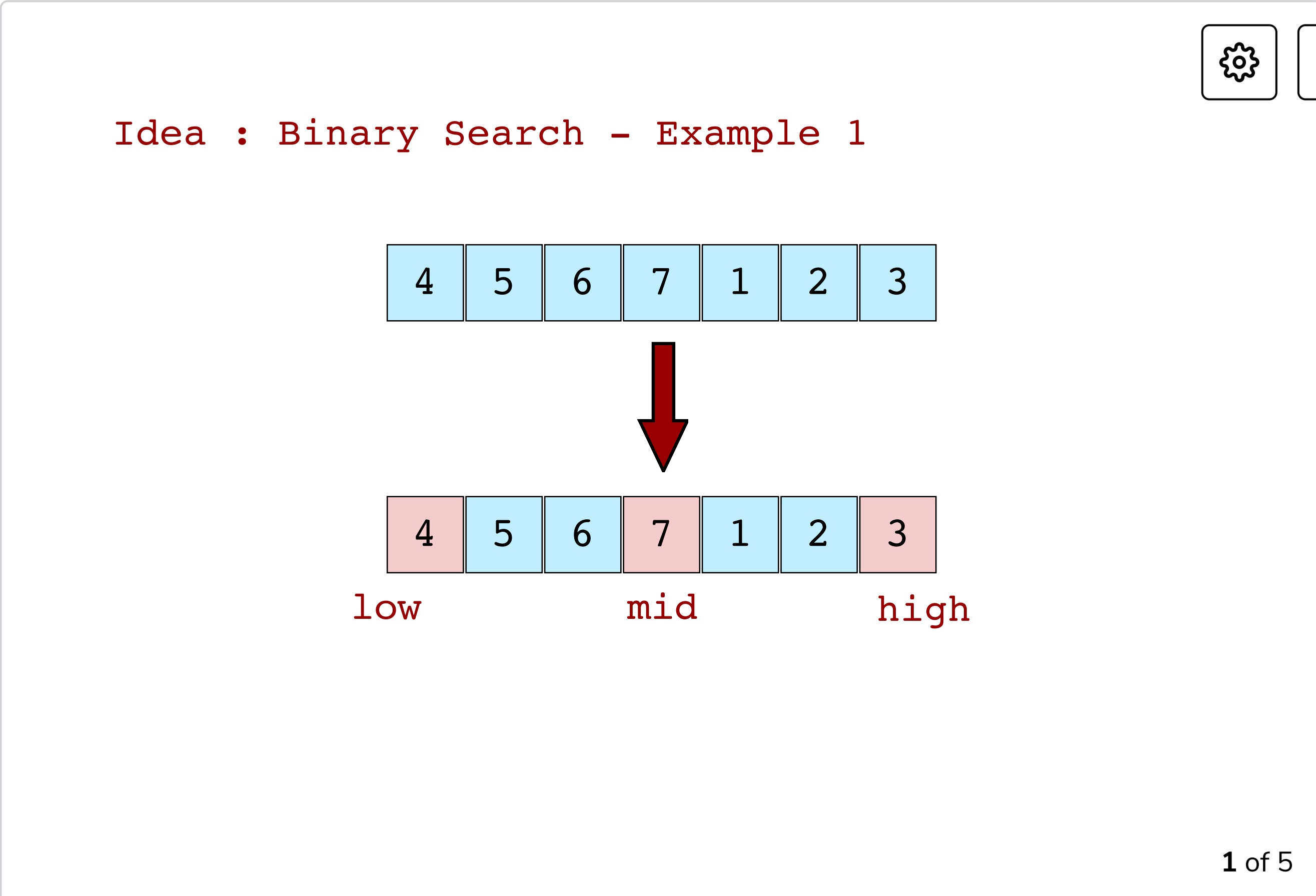
Let's reiterate the problem statement from the previous challenge.

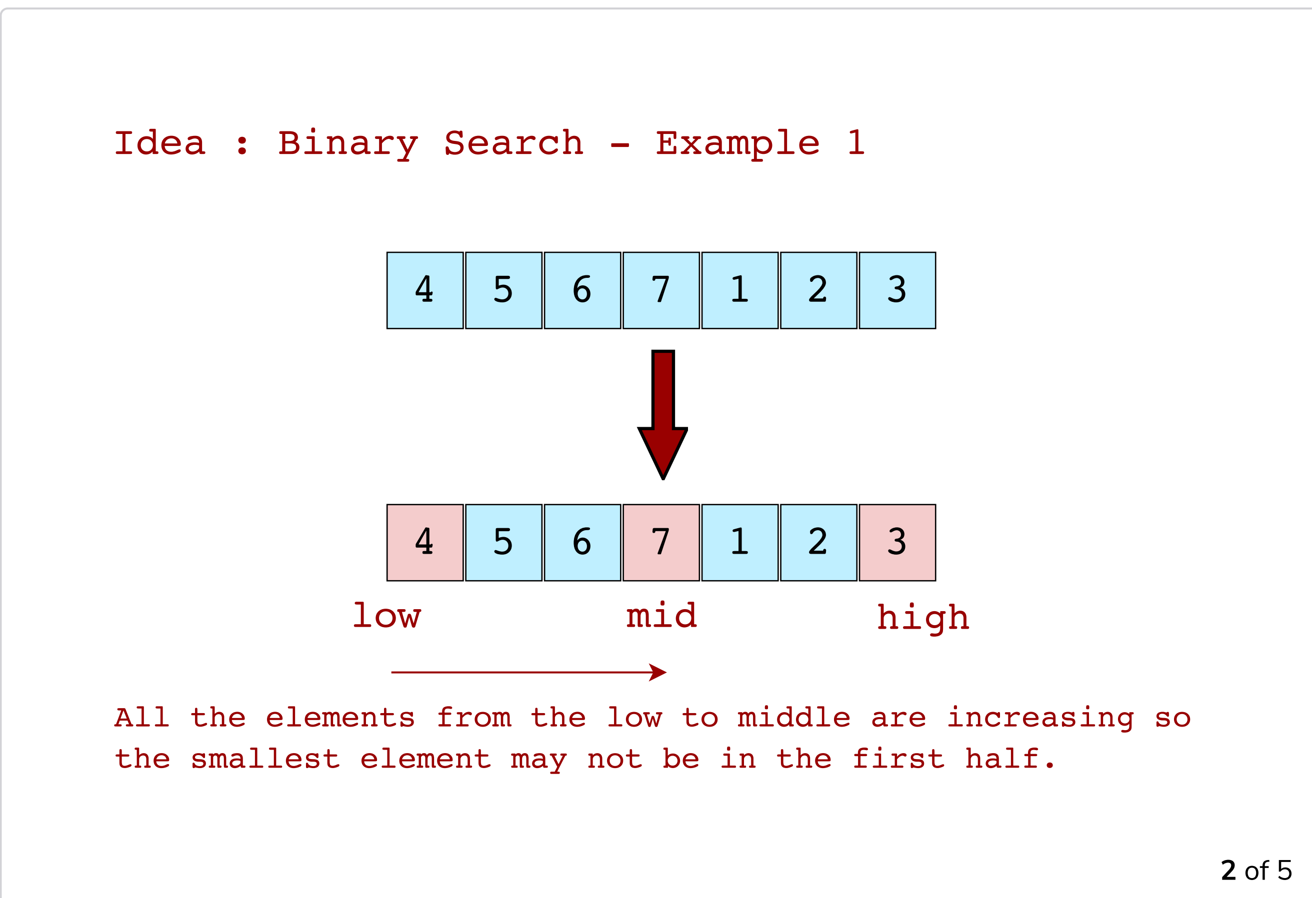
You are required to write a function that determines the index of the smallest element of the cyclically shifted array.

An array is **"cyclically shifted"** if it is possible to shift its entries cyclically so that it becomes sorted.

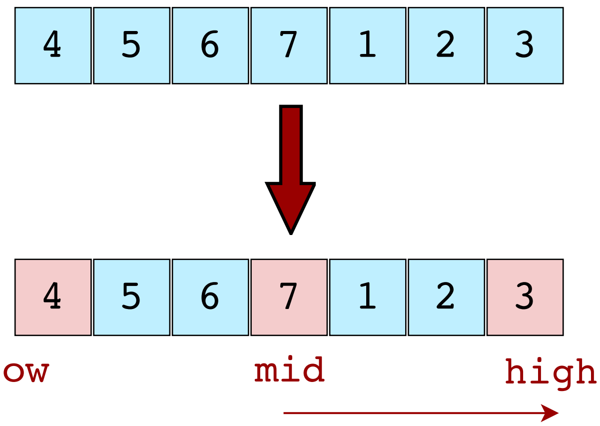
Algorithm

Now we need to come up with a strategy to eliminate parts of the search space. Have a look at the slides below to take note of some observations


Idea : Binary Search - Example 1
4 5 6 7 1 2 3
4 5 6 7 1 2 3
low
mid
high
1 of 5

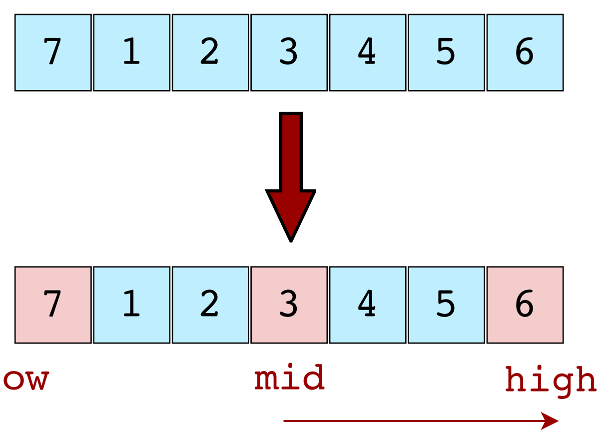

Idea : Binary Search - Example 1
4 5 6 7 1 2 3
4 5 6 7 1 2 3
low
mid
high
All the elements from the low to middle are increasing so
the smallest element may not be in the first half.
2 of 5

Idea : Binary Search - Example 1

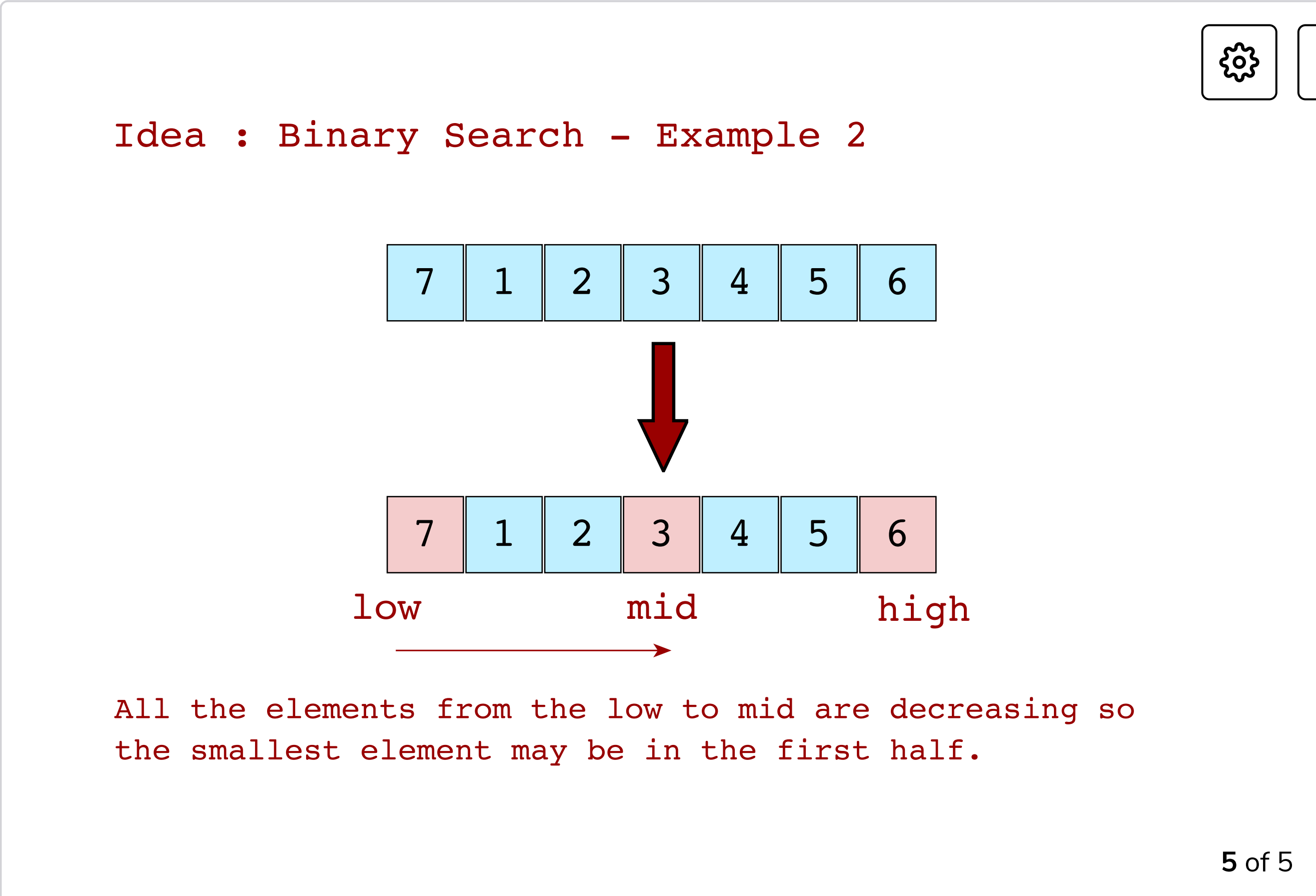


All the elements from the middle to high are decreasing so the smallest element may be in the second half.

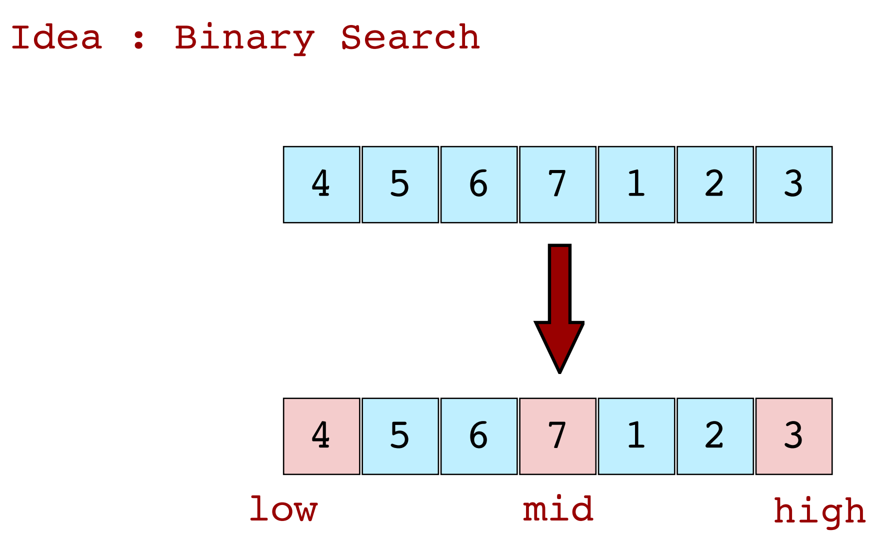
Idea : Binary Search - Example 2



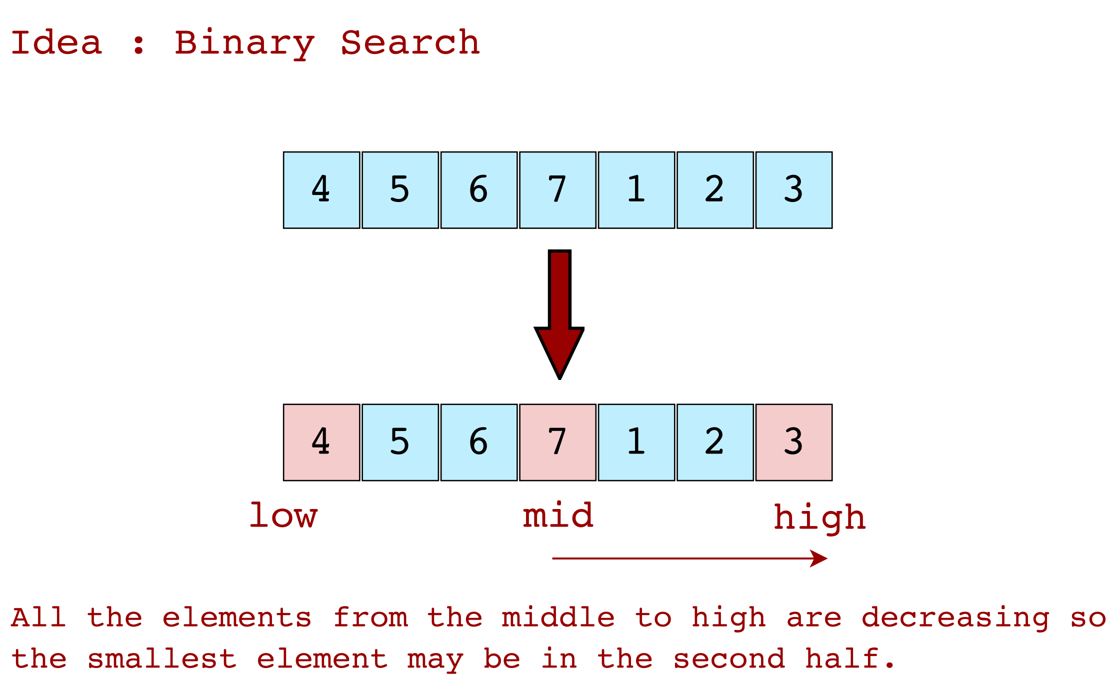
All the elements from the middle to high are increasing so the smallest element may not be in the second half.



At this point, you will have a basic idea of how to solve this problem. Let’s step more into the algorithm which is applied to *Example 1* from the slides above:

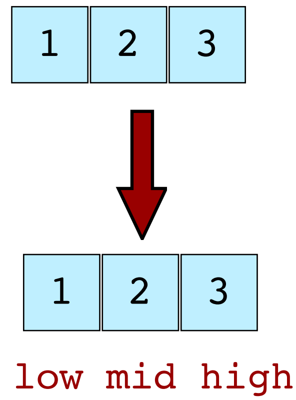

Idea : Binary Search
4
5
6
7
1
2
3
4
5
6
7
1
2
3
low
mid
high

1 of 4


Idea : Binary Search
4
5
6
7
1
2
3
4
5
6
7
1
2
3
low
mid
high
All the elements from the middle to high are decreasing so the smallest element may be in the second half.

2 of 4

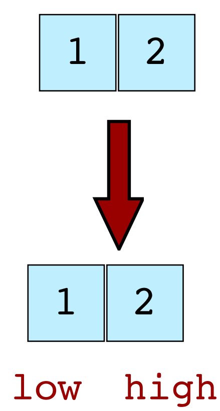
Idea : Binary Search



2 < 3,
smallest element may be to the left
so we dismiss all the elements to the right of the mid.

3 of 4

Idea : Binary Search



When we come to a case like this, all we need
to do is to return the low.

4 of 4

Implementation

Now that you have a complete idea of the algorithm, let's jump to the implementation in Python:

```
1  def find(A):
2      low = 0
3      high = len(A) - 1
4  
5      while low < high:
6          mid = (low + high) // 2
7  
8          if A[mid] > A[high]:
9              low = mid + 1
10         elif A[mid] <= A[high]:
11             high = mid
12 
13     return low
14 
15 A = [4, 5, 6, 7, 1, 2, 3]
16 idx = find(A)
17 print(A[idx])
```

Output 0.83s

```
1
```

Explanation

`low` and `high` are set to `0` and `len(A) - 1` respectively on **lines 2-3**. The code on **lines 5-6** is the same as the code in the standard binary search implementation that we covered at the beginning of the chapter. According to the algorithm, we check on **line 8** if the middle element is greater than `A[high]`. If it is, then it implies that the elements are decreasing from the middle to the high element. To reduce the search space, `low` is set equal to `mid + 1` on **line 9**. **Line 10** is evaluated in case the condition on **line 8** is not `True`, so we check if the middle element is less than or equal to `A[high]`. If this condition evaluates to `True`, it implies that the elements are increasing from `mid` position to `high` position and the smallest element may be somewhere between `low` position to `mid` position. Therefore, `high` is set to `mid` to eliminate the space from `mid` position to the previous `high` position. After the `while` loop terminates, `low` will be the index of the smallest integer in the list.

That's all on what we have for Binary Search. In the next chapter, we'll learn to solve a few problems using recursion. Stay tuned!

← Back

Next →

Exercise: Cyclically Shifted Array

Quiz

Mark as Completed

Report an Issue

Ask a Question (https://discuss.educative.io/tag/solution-review-cyclically-shifted-array__binary-search__data-structures-and-algorithms-in-python)