Now consider the behaviour of the Binary Search algorithm when it is given a value key to search for and it is being used to search in a sorted array A with positive length n.

Look at slides 7-8 for preconditions, add slide 15 for condition d).

Precondition

a. A is an array with length A.length = n ≥ 1 storing values of some ordered type T
   A is not empty
b. A[i] < A[i + 1] for every integer i such that 0 ≤ i < n − 1
   A is a sorted array
c. Either:
   i. Case 1: key is a value of type T that is stored in A OR
   ii. Case 2: key is a value of type T that is ***not*** stored in A
   Loop invariant that proves your search algorithm is working.
d. low and high are integers such that
   i. 0 ≤ low ≤ n
   ii. −1 ≤ high ≤ n− 1
   iii. low ≤ high + 1
   iv. A[h] < key for 0 ≤ h < low
   5a) low = 0 therefore no values of h exist
   v. A[h] > key for high < h ≤ n − 1
   5a) high = n-1 therefore no values of h exist

5b

Looking at the last two conditions, if we assume high = **low - 1**, then:

A[h] < key for 0 ≤ h < low
A[h] > key for **low - 1** < h ≤ n − 1

For all values h < low, A[h] is not the key. For all values h ≥ low, A[h] is not the key.

5c

If low = high, then:

A[h] < key for 0 ≤ h < low
A[h] > key for **low** < h ≤ n − 1

For all values h < low, A[h] is not the key. For all values h > low, A[h] is not the key. Then either A[low] = key, or A[h] ≠ key for every h between 0 and n-1.

5d

Given low ≥ high, we know either low > high or low = high. From the above, we know that if low > high, then the key is not an element of the array. If low = high, then either A[low] = key OR key does not exist in the array.

5e

Assuming either precondition for the current execution is satisfied, then:

- Conditions a), b), and c) are trivial, as long as A and key do not change.
- low and high are integers such that
  - $0 \leq low \leq n$
  - $-1 \leq high \leq n-1$
  - $low \leq high + 1$
  - $A[h] < key$ for $0 \leq h < low$
  - $A[h] > key$ for $high < h \leq n - 1$

If a recursive call is made, then:

- $high \geq low$
- $mid = \lfloor (low + high) / 2 \rfloor$

Case 1: A[mid] > key

- newLow = low
- newHigh = mid - 1
- Since $mid = \lfloor (low + high) / 2 \rfloor$, and low and high are positive integers, then $mid \geq 1$. Then newHigh +1 ≥ 1, and **newHigh ≥ 0**.
- Since low < high, and $mid = \lfloor (low + high) / 2 \rfloor$, then mid < high, and mid < n-1. Because newHigh = mid - 1, **newHigh < mid < n-1**.
- Since low < high, and $mid = \lfloor (low + high) / 2 \rfloor$, then mid > low, and mid > newLow. We know mid = newHigh + 1, so **newHigh + 1 > newLow**.
- If A[mid] > key, then A[newHigh+1] > key. Condition a) tells us the array is sorted, so we know for all values h > newHigh + 1, A[h] > key. Then we can say the same for **all values h ≥ newHigh + 1, or h > newHigh**.

Case 2: A[mid] < key

- newHigh = high
- newLow = mid + 1
- Since $mid = \lfloor (low + high) / 2 \rfloor$, and low and high are positive integers, then $mid \geq 1$. Then newLow - 1 ≥ 1, and **newLow ≥ 0**.
- Since low < high, and $mid = \lfloor (low + high) / 2 \rfloor$, then mid < high, and mid < n-1. Then newLow - 1 < n - 1, and **newLow ≤ 0**.
- Since low < high, and $mid = \lfloor (low + high) / 2 \rfloor$, then mid < high, and mid < newHigh. We know mid = newLow - 1, so newLow - 1 < newHigh, and **newLow < newHigh + 1**.
- If A[mid] < key, then A[newLow-1] < key. Condition a) tells us the array is sorted, so we know for all values h < newLow - 1, A[h] < key. Then we can say the same for **all values h ≤ newLow - 1, or h < newLow**.

If the postconditions of the recursive (inner) application are correct, then…

Case 1: In the original (outer) application of bsearch, A[mid] > key.
- Since the array is sorted, then A[h] ≠ key for all h ≥ mid.
- The postcondition of the inner application tells us that, either:
  - Case 1a: A[h] = key for some value h, where low ≤ h < mid-1
    - Then A[h] is found for some value h, where low ≤ h ≤ mid-1 < high.
  - Case 1b: A[h] ≠ key for all values h, where low ≤ h < mid-1
    - Then A[h] ≠ key for all h, where low ≤ h ≤ mid-1 < high
  - Either case satisfies the postcondition for the outer application of bsearch

Case 2: In the original (outer) application of bsearch, A[mid] < key.
- Since the array is sorted, then A[h] ≠ for all h ≤ mid.
- The postcondition of the inner application tells us that, either:
  - Case 2a: A[h] = key for some value h such that mid+1 ≤ h ≤ high
    - Then A[h] is found for some value h, where low < mid + 1 < h ≤ high.
  - Case 2b: A[h] ≠ key for all values h, where mid+1 ≤ h ≤ high
    - Then A[h] ≠ key for all h, where low < mid + 1 < h ≤ high.
  - Either case satisfies the postcondition for the outer application of bsearch

2. Continue to consider the Binary Search algorithm and the bsearch subroutine that does all the work.
   a. Suppose the subroutine is called with (the first two) inputs low and high such that
      high = low + 2h − 1
      for some positive integer h, so that the part of the array that remains to be searched has length high − low + 1 = 2h.

      Show that if bsearch calls itself recursively, then it does to so search a part of the array that has length at most h.

      Assuming bsearch calls itself recursively, then:
      Case 1: Search from low to (mid-1)
      - mid = ⌊(low + high) / 2 ⌋
        = ⌊(low + low + 2h - 1) / 2 ⌋
        = low + h - ⌊½⌋
        = low + h
      - # of elements = (mid - 1) - low
        ≤ low + h - ½ - 1 - low
        = h - 1½
        ≤ h

      Case 2: Search from (mid+1) to high

- mid = ⌊(low + high) / 2⌋
  = ⌊(high - 2h +1 + high) / 2⌋
  = high - h + ⌊½⌋
  = high - h
- # of elements = high - (mid + 1)
  = high - mid - 1
  = high - (high - h) - 1
  = h - 1
  ≤ h

b. Suppose, instead, that bsearch is called with inputs low and high such that
high = low + 2h,
for a positive integer h, so that the part of the array that remains to be searched has length 2h + 1.

Show that if bsearch calls itself recursively, then it does so to search a part of the array that has length at most h in this case, as well.

Assuming bsearch calls itself recursively, then:
mid = ⌊(low + high) / 2⌋
= ⌊(low + low + 2h) / 2⌋
= low + h = high - h
Case 1: Search from low to (mid-1)
# of elements = (mid - 1) - low
= low + h - 1 - low
= h - 1
≤ h
Case 2: Search from (mid+1) to high
# of elements = high - (mid+1)
= high - (high - h + 1)
= h - 1
≤ h

c. Notice that if bsearch is called when low = high (to search a part of an array with length one) then it calls itself at most once more.
If low = high, then mid = low.
If A[mid] = key, 0 recursive calls.
If A[mid] > key, bsearch(low, low - 1), throw KeyNotFoundException.
If A[mid] < key, bsearch(low + 1, low), throw KeyNotFoundException.

d. Use all of the above to show that if the Binary Search is run on an array of length n, for n ≥ 1, then the total number of calls made to bsearch is at most $\log_2 n + 2$.
We make one call at the beginning.

In the worst case, the key does not exist in the array.
Then we have to make $\log_2 n$ recursive calls before low = high. (a diagram such as a binary tree can easily show this)
From above, we can see that when low = high, we make at most one more call.
Thus the worst case is $\log_2 n + 2$.

3. Now consider the use of Binary Search to search for a key in an array A with positive length n, when the key is greater than all of the values stored in A Use mathematical induction to show that bsearch is called at least $\log_2 n + 1$ times during this search.

4. Use the results of the last three questions to argue that the Binary Search algorithm is correct and, furthermore, that it can be used to search in a sorted array of length n using $\Theta(\log_2 n)$ operations in the worst case.

   Questions 3 and 4 are covered from slides 18-21 in lecture 15 notes.