Binary Search

The Problem

The General Problem

Search for an answer(entry/position) that meets some designated properties in a sequential list.

The Specific Problem

Search for an integer in a sorted array with distinct integers and return its index if it's present or -1 if it's absent.

This is a classic problem of binary search that will be used as a specific running example to illustrate many apsects of the algorithm. Other applications and variants of the problem will be discussed in the section **Generalization and Extension**.

Core Idea

Halving the Problem Size

The core idea underlying binary search is to solve a problem, which is typically searching for a given key in a sequential list, by continually **halving the problem size**. The problem size is often reflected as the length of the subsequence derived from the original list. We ensure that each time we reduce the problem size, the correct answer to the problem, if present, is somewhere in the subsequence but absent from the rest of the list. Because we reduce the problem size in each iteration, eventually we terminate the algorithm in one of the two ways: either we find the answer in one of the iterations or the problem size is reduced to 0, for which we conclude the answer is absent from the list.

The Premise of Binary Search

Many search algorithms solve the problem by reducing the problem size, e.g. sequential/linear search reduces the problem size by 1 for each iteration and it's a universal method because we examine every entry without omission. Binary search does the problem size reduction by halving the current problem size, but it depends on an indispensable premise:

If the correct answer(entry/position) is present in the list, every entry that comes before it shares a common property, and every entry that comes after it shares another common property, and the two properties of the two subsequences must be exclusive, that is no entry can have two of them at once. These properties are often expressed as predicates.

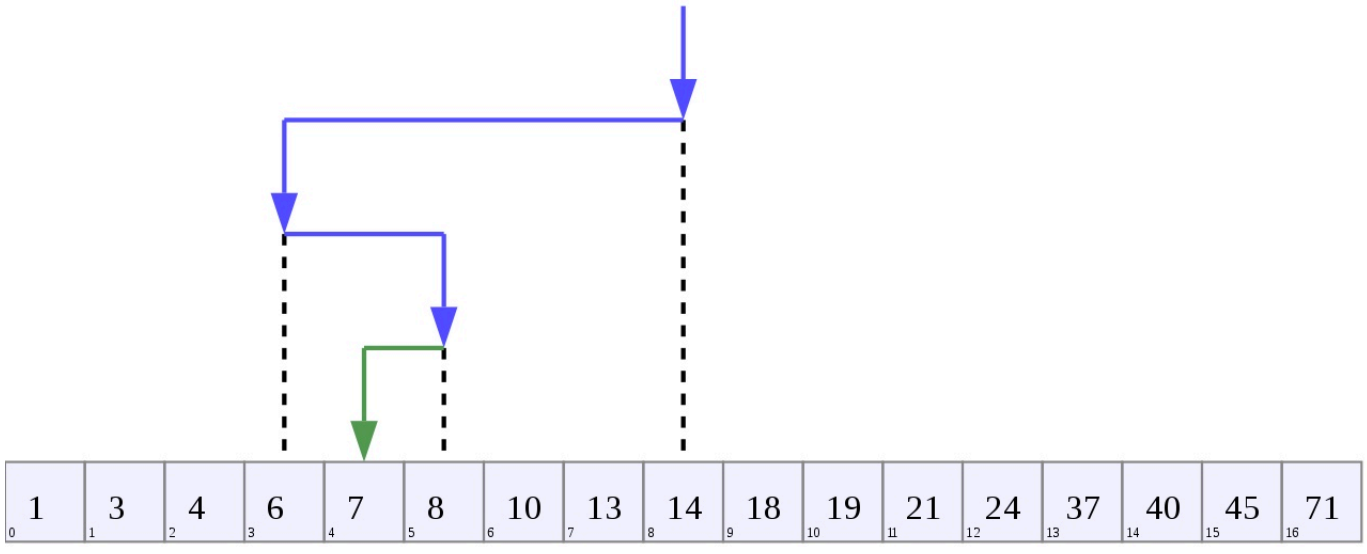
This is the premise of binary search, and many lists of interest we need to perform the search on have more specific properties that satisfy this premise. A generalized version of this premise, which extends binary search to a broader range of problems is presented in the section **Generalization and Extension**.

Solution Analysis

Applying the Core Idea

For a sorted array, the premise of binary search is satisfied : **If the target key is present in the array $a[]$, every entry that comes before is smaller than it and every entry that comes after is larger than it.** This follows naturally from the monotonicity of the entries in the array: **Every pair of $(a[i], a[i+1])$ always have some common transitive relation, in this case, $a[i] < a[i+1]$.**

Applying the core idea, we examine the entry on the midpoint of a given subsequence and use its relation with $\boldsymbol{key}$ to halve the subsequence for the next iteration, reducing the problem size in half.



From a mathematical perspective, given a sorted array, $A = [a_0, a_1, \ldots, a_{n-1}]$, binary search essentially tries to narrow the sequence that a specific key might fall into, thus creating a subsequence each iteration, outside of which key is impossible to exist. Once the subsequence become empty, we conclude that key is not in A.

Because the array is sorted in ascending order, $\forall a_i \in A,\ a_i <= a_{n-1}\ and\ a_i >= a_0$. We can first check if key satisfy this predicate, and if not, that is either $key < a_0\ or\ key > a_{n-1}$, we don't need to perform any search. Having excluded this marginal case, we know that key is possibly present in the array, then we do the binary search:

Four Equivalent Methods to Represent a Subsequence

To narrow the subsequence of interest, we need a pair of variables that store the lower bound and upper bound of the subsequence. Typically we use two variables: low and $high$, or $left$ and $right$ equivalently by some literature , with $low \leq high$ for this one dimensional sequence.

There are actually four possible intervals that can be represented by these two variables:

1. $[low..high]$, becomes empty when $low > high$, typically when $low = high + 1$
2. $[low..high)$, becomes empty when $low = high$
3. $(low..high]$, becomes empty when $low = high$
4. $(low..high)$, becomes empty when $low \geq high - 1$, typically when $low = high - 1$

Each corresponds to a slightly different implementation, with perhaps different ways to calculate the midpoint and different exit conditions for the loop. After some experiment, I find that the second is the fastest and uses the least number of comparison, but I'll stick to the first one as it's the easiest to implement and understand.

Note that we don't necessarily exit the loop when the subsequence is empty because for some problems our interest lies in the last entry.

Three Important Elements of Implementation

There are three important elements of implementation of binary search:

1. **Midpoint Index Calculation**: How to calculate the index of midpoint?
2. **Search Boundaries Updates**: How and when do we update low and $high$?
3. **Exit Conditions for Loop**: When do we terminate the loop?

To ensure the correctness of the implementation we scrutinize these correlated elements in the following sections.

Midpoint Index Calculation

For two points x_1 and x_2 on the number line of real numbers, supposing without loss of generality: $x_1 < x_2$, the midpoint x_m can be derived from its definition: $x_m - x_1 = x_2 - x_m$, so we have

$$x_m = \frac{x_1 + x_2}{2} \tag{1}$$

However, using this formula to calculate the midpoint in many programming languages is vulnerable to potential **addition overflow**, as $x_1 + x_2$ may be larger than **Integer.MAX_VALUE**, which is $2^{31} - 1$ in Java.

This problem motivate us to find another formula that is more suitable for computer programming, which avoid the direct addition of x_1 and x_2. We can work on the original formula as follows: $\frac{x_1+x_2}{2} = \frac{x_1+x_2+x_1-x_1}{2}$, so we have

$$x_m = x_1 + \frac{x_2 - x_1}{2} \tag{2}$$

$\frac{x_2-x_1}{2}$ is half the distance between x_2 and x_1, which is guaranteed to be positive and because x_1 is no less than 0, it won't be larger than $x_2/2$. Adding x_1 to it yields an integer that is less than x_2, thus prevents overflow.

Note that when using formula(1), we don't need to care which of the two indexes is larger, but when applying formula(2) we do.

In the context of integer arithmetic, $\frac{x_1+x_2}{2}$ is also equivalent to $x_1 + \frac{x_2-x_1}{2}$ when no overflow occurs, and they both evaluate to $x_1 + \lfloor\frac{x_2-x_1}{2}\rfloor$.

Thus in programming languages such as C, C++ and Java, the expressions:

```
int mid = low + (high - low) / 2;
int mid = low + ((high - low) >> 1);
int mid = (low + high) >>> 1; // utilize java's arithmetic right shift
```

will always yield an integer that is equal to $low + \lfloor\frac{high-low}{2}\rfloor$. I call this the **floor midpoint**.

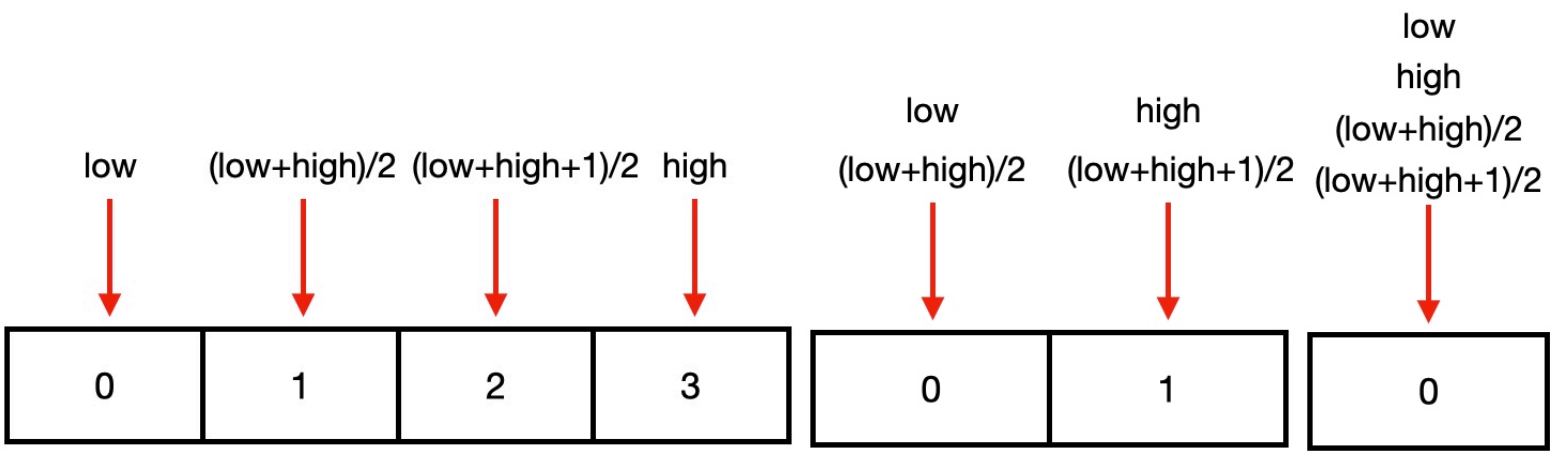
There is an alternative way to calculate the index of the midpoint that always evaluates closer to $high$:

```
int mid = low + (high - low + 1) / 2;
int mid = low + ((high - low + 1) >> 1);
int mid = (high - low + 1) >>> 1; // utilize java's arithmetic right shift
```

will always yield an integer that is essentially $low + \lceil\frac{high-low}{2}\rceil$. I call this the **ceiling midpoint**.

The length of the subsequence determined by low and $high$ is $length = high - low + 1$.

1. When $length$ is odd, $high - low$ will be even, so $(high - low)/2$ is an integer, which means $\lfloor\frac{high-low}{2}\rfloor = \lceil\frac{high-low}{2}\rceil$, so mid will be exactly the midpoint of low and $high$.
2. When $length$ is even, $high - low$ will be odd, so in integer arithmetic $\lfloor\frac{high-low}{2}\rfloor = \lceil\frac{high-low}{2}\rceil - 1$, which means mid will be closer to low if we use the **floor midpoint**, and will be closer to $high$ if we use the **ceiling midpoint**.



Search Boundaries Update

Having obtained the index of the midpoint mid, we check the input key against $a[mid]$, and of course if they are equal we return mid, otherwise we need to update low or $high$ to halve the subsequence.

There are typically two ways to update the boundary pointer low, when $a[mid] < key$ or $a[mid] \geq key$:

1. $low := mid$, is often paired up with $a[mid] \geq key$
2. $low := mid + 1$, is often paired up with $a[mid] < key$

There are typically two ways to update the boundary pointer $high$, when $a[mid] > key$ or $a[mid] \leq key$:

1. $high := mid$, is often paired up with $a[mid] \leq key$
2. $high := mid - 1$, is often paired up with $a[mid] > key$

There are other ways to update the boundary pointers, such as $low := mid - 1$, $high := mid + 1$, and etc. They may also lead to a correct implementation but the codes will be clumsy and hard to understand.

Exit Conditions for Loop

Typically there are three ways to code the loop condition for the **while** loop:

1. $while(low \leq high)$, the loop will terminate when the subsequence becomes empty
2. $while(low < high)$, the loop will terminate when the subsequence has one entry
3. $while(high - low > 1)$, the loop will terminate when the subsequence has two entry

Note that methods terminating the loop without finding key when there are more than one entries, such as $while(high - low > 1)$, is rarely used, because it means we have to examine these entries outside the loop.

Correlated Combination

A complete procedure of binary search is a combination of correlated choices of the above three elements: we can use either the **floor midpoint** or the **ceiling midpoint**, which will impact how and when we update low and $high$ and in turn impacts when we exit the loop. There are many combinations that lead to a correct implementation, thus a complete enumerations is tedious and unnecessary. I only document the crux and a few caveats here:

The crux of a correct implementation of binary search lies in two essential requirements:

1. We always examine the possible interval the correct answer might be in, thus never miss it.
2. We always reduce the length of the subsequence in each iteration, thus never loop infinitely.

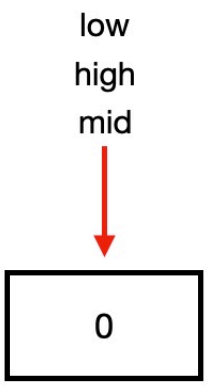
The caveats of an incorrect implementation are violations of any of the two requirements above.

There are some examples of combinations that illustrate the common issues:

1. Using $low := mid$ or $high := mid$ but not both, with $while(low \leq high)$ is a wrong idea, because we may be stuck in an infinite loop when there is only one entry in the subsequence.

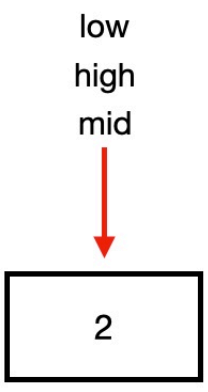
```
while (low <= high) {
    final int mid = low + (high - low) / 2;
    final int entry = array[mid];
    if (key == entry) {
        return mid;
    } else if (key > entry) {
        low = mid;
    } else { // key < entry
        high = mid - 1;
    }
}
```

key=1




```
while (low <= high) {
    final int mid = low + (high - low) / 2;
    final int entry = array[mid];
    if (key == entry) {
        return mid;
    } else if (key > entry) {
        low = mid + 1;
    } else { // key < entry
        high = mid;
    }
}
```

key=1

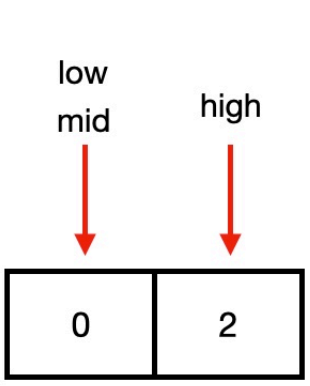


The correct choice is to pair them with $while(low < high)$ which exits the loop when there are one remaining entry. If we check $entry = key$ in the loop we can conclude that it's absent from the array, but if we use $a[mid] \leq key$ or $a[mid] \geq key$ instead we need to check the equality outside the loop.

2. Pairing $low := mid$ with the floor midpoint or $high := mid$ with the ceiling midpoint is also wrong, even when the loop condition is $while(low \leq high)$

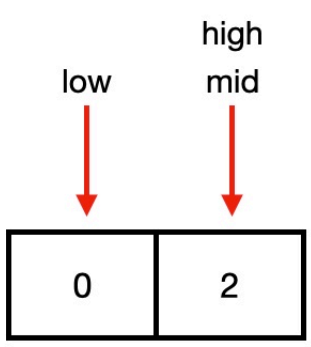
```
while (low < high) {
    final int mid = low + (high - low) / 2;
    final int entry = array[mid];
    if (key == entry) {
        return mid;
    } else if (key > entry) {
        low = mid;
    } else { // key < entry
        high = mid - 1;
    }
}
```

key=1




```
while (low < high) {
    final int mid = low + (high - low + 1) / 2;
    final int entry = array[mid];
    if (key == entry) {
        return mid;
    } else if (key > entry) {
        low = mid + 1;
    } else { // key < entry
        high = mid;
    }
}
```

key=1



The correct choice is to pair $low := mid$ with the ceiling midpoint and $high := mid$ with the floor midpoint.

3. Using both $low := mid$ and $high := mid$ paired with $while(high - low > 1)$ is workable, but not a good idea in general and can be easily replaced with better solution, so we don't further analyze this combination.

4. Using both $low := mid + 1$ and $high := mid - 1$ can be paired with either $while(low < high)$ or $while(low \leq high)$ as the loop conditions and either the floor or ceiling midpoint. But as for when to update them, pairing $low := mid + 1$ with $a[mid] \leq key$ or $high := mid - 1$ with $a[mid] \geq key$ is obviously wrong, because we may be skipping the correct answer.

How do we avoid the wrong combinations? One trick is to examine the **marginal cases**, the iterations when the loop are supposed to terminate, typically thing may go wrong when there are one or two entries left in the subsequence.

Common Standard Templates

Although there are many correct implementation of binary search as illustrated in the previous section, it's a good idea to stick to only a few useful standard templates that are efficient and easy to understand and debug.

Following are two standard templates that not only are useful in finding a given key but also serves as templates for many other problems.

Inclusive Template

Compare the middle entry of the subsequence, if it's key return mid directly, otherwise update the subsequence to $[low..mid - 1]$ or $[mid + 1..high]$ according to its relation with key. We repeat this process to narrow the subsequence that may contain key, and if it's in the array it will definitely be found. Otherwise the subsequence will eventually become empty and we know that it's absent.

```java
public static int binarySearchInclusive(final int[] array, final int key) {
    int low = 0;
    int high = array.length - 1;

    while (low <= high) {
        final int mid = low + (high - low) / 2;
        final int entry = array[mid];
        if (entry < key) {
            low = mid + 1; // subsequence becomes [mid+1,high]
        } else if (entry > key) {
            high = mid - 1; // subsequence becomes [low,mid-1]
        } else {
            return mid;
        }
    }

    return -1;
}
```

The inclusive template is intuitive and efficient. If the key is not present in the array, the low and $high$ pointers will have some useful properties that we will consider later.

Implementation Note: We consider the probability of each branch and see that in most cases **entry==key** is the most unlikely branch, so we move it to the last branch giving the two other branches higher priorities.

Exclusive Template

Compare the middle entry of the subsequence, if it's larger than or less than key, trim down the subsequence by one of the following two methods:

1. **Lower Method**: Partition the subsequence into two exclusive parts, the entries smaller than key and the entires no less than the key. If the midpoint entry is smaller than the key, than the index of the key is impossible in $[low..mid]$, we trim down the subsequence to $[mid + 1..high]$ by updating $low := mid + 1$; else (the midpoint entry is larger than or equal to the key), the index of the key might be in $[low..mid]$, but is impossible in $[mid - 1..high]$, thus we trim down the subsequence to $[low..mid]$ by updating $high := mid$.

2. **Upper Method**: Partition the subsequence into two exclusive parts, the entries larger than key and the entires no greater than the key. If the midpoint entry is larger than the key, than the index of the key is impossible in $[mid..high]$, we trim down the subsequence to $[low..mid - 1]$ by updating $high := mid - 1$; else (the midpoint entry is smaller than or equal to the key), the index of the key might be in $[mid..high]$, but is impossible in $[low..mid - 1]$, thus we trim down the subsequence to $[mid..high]$ by updating $low := mid$.

We repeat this process to narrow the subsequence until its length becomes one and exit the loop. If the only remaining entry equals key, we have its index, otherwise in **Lower Method** we have the index of the smallest(first) number that is larger than key(its **successor**); and in **Upper Method** we have the index of the largest(last) number that is smaller than key(its **predecessor**).

```java
public static int binarySearchExclusiveLower(final int[] array, final int key) {
    int low = 0;
    int high = array.length - 1;
```

```
    while (low < high) {
        final int mid = low + (high - low) / 2;
        final int entry = array[mid];
        if (entry < key) {
            low = mid + 1; // subsequence becomes [mid+1,high]
        } else {
            high = mid; // subsequence becomes [low,mid]
        }
    }

    return array[low] == key ? low : -1;
}
```

```
public static int binarySearchExclusiveUpper(final int[] array, final int key) {
    int low = 0;
    int high = array.length - 1;

    while (low < high) {
        final int mid = low + (high - low + 1) / 2;
        final int entry = array[mid];
        if (entry > key) {
            high = mid - 1; // subsequence becomes [low,mid-1]
        } else {
            low = mid; // subsequence becomes [mid,high]
        }
    }

    return array[low] == key ? low : -1;
}
```

The exclusive template has a faster comparison loop and although it's on average less effective in finding a given key but becomes useful when we need to find the first or the last index than meets some demand, such as the successor or the predecessor of that key. Another reason to use the exclusive template is that we have only one entry to consider after the loop exits which free us from dealing with two pointers, simplifying the codes.

Implementation Note: We consider that $key \geq entry$ is always a more expensive operation than $key > entry$, and the same goes for $key \leq entry$ and $key < entry$. Thus when coding a binary **if-else** branching structure we always choose the simpler one as the branch condition, which is faster to run and easier to code.

Recursive Implementation

Above are the iterative implementation of binary search, all having their recursive counterparts. I post the recursive codes for inclusive template here and the ones for exclusive templates are very similar.

```
public static int binarySearchRecursive(final int[] array, final int key) {
    return binarySearchRec(array, key, 0, array.length - 1);
}

public static int binarySearchRec(final int[] array, final int key, final int low, final int high) {
    if (low > high) return -1;

    final int mid = low + (high - low) / 2;
    final int entry = array[mid];
    if (entry < key) {
        return binarySearchRec(array, key, mid + 1, high);
    } else if (entry > key) {
        return binarySearchRec(array, key, low, mid - 1);
    } else {
        return mid;
    }
}
```

Binary search when implemented correctly is promised to have $O(\log N)$ iterations, and thus at most $O(\log N)$ recursive calls, which is unlikely to cause a stack-overflow even for very large N.

Out-of-Bound Cases Check

We omitted the following codes that checks for the marginal cases, which is common to all implementations:

```
        if (array == null || array.length == 0) return -1;
        if (array[0] > key || array[array.length - 1] < key) return -1;
```

The first check is indispensable, but the second is to improve performance, without which all the different versions of implementations above can still solve the problem correctly.

In the following analysis I'll refer to $key < a[0]$ and $key > a[n-1]$ as **out-of-bound cases**.

Invariants and Properties

Predecessor and Successor

The largest entry that is smaller than **key** is its **predecessor**.

The smallest entry that is larger than **key** is its **successor**.

The **predecessor** and the **successor** of a **key** are its approximate matches in an array, and they are the closest entries to **key**. Note that when **key** is larger than every entry in the array it doesn't have a **successor**, and when it's smaller than every entry in the array it doesn't have a **predecessor**. For a sorted array we can check key against $a[0]$ and $a[n-1]$ to test the existence of its **predecessor** and **successor**:

1. If $key <= a[0]$, it doesn't have a **predecessor**
2. If $key >= a[n-1]$, it doesn't have a **successor**

Common Invariants and Properties

Invariants between Key and the Entries

If we exclude the **out-of-bound cases** , we have several invariants that hold during the loop with subsequence determined by $[low, high]$, which are useful properties we can utilize in some problems:

$$key > a[i], \forall i < lowkey < a[i], \forall i > highkey \geq a[low] \ \wedge \ key \leq a[high]$$

Complete Split of Subsequence

The different methods to update low and $high$ aims to maintain one invariant:
The subsequence before split equals the two disjoint subsequences after split plus a_{mid}.
Failure to maintain this invariant will induce bugs that skip certain entries in the array.

Invariants and Properties for Inclusive Template

Non-empty Subsequence

The loop conditions of the while loop signals when the subsequence becomes empty, which is an invariant:
The subsequence is not empty when the loop condition is TRUE.
Failure to maintain this invariant will cause the loop to terminate prematurely or out-of-bound access to the array.

Marginal Cases of Loop Exit

The loop exits when the subsequence becomes empty or the **key** is found. Note that **key == a[mid]** may happen in any iteration of the loop so it has nothing to do with the length of the subsequence. Marginal cases where the subsequence becomes empty are when the subsequence is of size 1 or 2, and both cases may produce an empty subsequence for the next iteration.

1. When **subsequence.length == 1**, we have $low = high$
   i. If $a[mid] = key$, return mid, the loop exits.
   ii. If $a[mid] > key$, $high := mid - 1$ which makes $low > high$. Besides, **a[mid/low]** is the **successor**, and **a[mid-1/high]** the **predecessor**.
   iii. If $a[mid] < key$, $low := mid + 1$ which makes $low > high$. Besides, **a[mid+1/low]** is the **successor**, and **a[mid/high]** is the **predecessor**
2. When **subsequence.length == 2**, we have $low = mid = high - 1$.
   i. If $a[mid] = key$, return mid, the loop exits.
   ii. If $a[mid] > key$, $high := mid - 1$, which makes $low > high$. Besides, **a[mid/low]** is the **successor** and **a[mid-1/high]** the **predecessor**.
   iii. If $a[mid] < key$, $low := mid + 1$, we need to do one more iteration.

To see that the length of the subsequence won't become empty in other situations, we need to prove that if the key is not in the array, the loop won't exit when **subsequence.length > 2**. Equivalently, we need to prove that :

$$\textbf{If } high - low + 1 \geq 3, mid - 1 \geq low \textbf{ and } mid + 1 \leq high$$

Inversely, we can start from $low > high$ to deduce the properties of values of low and $high$ one iteration before the loop exits, to prove that for $low > high$ in the next iteration, $high - low + 1 \leq 2$ must hold. The proof is easy so we omit it here.

Indexes of Predecessor and Successor

From the previous properties we conclude that If **key** is absent from the array, inclusive template will eventually converge on the entries that are closest to it, that are its **predecessor** or **successor**. Furthermore, both entries(if present) will be compared to **key** before the loop exits.
If we dive deeper, after the loop exits if **key** is not in the array, we always have:

$$successor = low = high + 1 = predecessor + 1predecessor = high = low - 1 = successor - 1$$

These formula only holds when the **key** has a **predecessor** or **successor**:

1. If it doesn't have a **successor**, we have $low = n$, which is its supposed insert position;
2. If it doesn't have a **predecessor**, we have $high = -1$;

Invariants and Properties for Exclusive Template

Subsequence of More than One Entry

The loop conditions of the while loop signals when the subsequence has only one entry, which is an invariant:
The subsequence has more than one entry when the loop condition is TRUE.
Failure to maintain this invariant will cause the loop to terminate prematurely because we only manually examine the last remaining entry.

Marginal Cases of Loop Exit

If we exclude the cases where the input array size is less than 2, the loop will exit when the length of the subsequence is exactly 1. Marginal cases are when the subsequence is of size 2 or 3, and both cases may produce a subsequence of single entry for the next iteration.

1. When **subsequence.length == 2**
    i. For method1, we have $low = mid = high - 1$
        a. If $a[mid] < key$, $low := mid + 1$, which makes $low = high$
        b. If $a[mid] \geq key$, $high := mid$, which makes $low = high$
        c. For either case, if key is in the array, we have $a[low/high] = key$. Otherwise if key has a **successor**, **a[low/high]** is its **successor**, else **a[low/high]** is its **predecessor**
    ii. For method2, we have $low + 1 = mid = high$
        a. If $a[mid] \leq key$, $low := mid$, which makes $low = high$
        b. If $a[mid] > key$, $high := mid - 1$, which makes $low = high$
        c. For either case, if key is in the array, we have $a[low/high] = key$. Otherwise if key has a **predecessor**, **a[low/high]** is its **predecessor**, else **a[low/high]** is its **successor**
2. When **subsequence.length == 3**
    i. For method1, we have $low = mid - 1 = high - 2$
        a. If $a[mid] < key$, $low := mid + 1$, which makes $low = high$. If key is in the array, we have $a[low/high] = key$. Otherwise if key has a **successor**, **a[low/high]** is its **successor**, else **a[low/high]** is its **predecessor**
        b. If $a[mid] \geq key$, $high := mid$, which makes $low = high - 1$, we need one more iteration
    ii. For method2, we have $low = mid - 1 = high - 2$
        a. If $a[mid] \leq key$, $low := mid$, which makes $low = high - 1$, we need one more iteration
        b. If $a[mid] > key$, $high := mid - 1$, which makes $low = high$. If key is in the array, we have $a[low/high] = key$. Otherwise if key has a **predecessor**, **a[low/high]** is its **predecessor**, else **a[low/high]** is its **successor**

To see that the length of the subsequence won't become empty in other situations, the deduction is similar to that of inclusive template.

Indexes of Predecessor and Successor

From the previous properties we conclude that If **key** is absent from the array, exclusive templates will eventually converge on the entries that are closest to it, that is its **predecessor** or **successor** if they exist, depending on which method is used. The discussion for the indexes below are based on the case when **key** is not in the array.

1. Using **Lower Method**
    i. If **key** has a **successor**, we have $successor = low = high = predecessor + 1$
    ii. If **key** doesn't have a **successor**, we have $predecessor = low = high = n - 1$
2. Using **Upper Method**
    i. If **key** has a **predecessor**, we have $predecessor = low = high = successor - 1$
    ii. If **key** doesn't have a **predecessor**, we have $successor = low = high = 0$

Note that these properties only hold when **key** is absent from the array, because when it is in the array, it's obvious that $predecessor = low - 1 = high - 1$ and $successor = low + 1 = high + 1$.
If we want to use the exclusive template to find the predecessor or successor for a given key, we better first check that it actually has a predecessor or successor and then use a modified implementation discussed in **Generalization and Extension** to solve this task.

Mathematics Analysis

Proof of Correctness

Informal Proof

The informal but intuitive proof of binary search is that we are always reducing the length of the subsequence while keep the invariant that if key is somewhere in the array, $a[low] \leq key \leq a[high]$ must be **TRUE** in every iteration. This eventually leads to one of the following cases:

1. During a certain iteration we find that $key = a[mid]$, so **mid** is exactly the answer.
2. When one of the marginal cases arises, leaving us with an empty subsequence, the loop terminates and we conclude that key is absent from the array, because we have narrowed the possible location of key into an empty range.

Formal Proof

TODO

https://www.cs.cornell.edu/courses/cs211/2006sp/Lectures/L06-Induction/binary_search.html
https://stackoverflow.com/questions/13696185/how-can-we-prove-by-induction-that-binary-search-is-correct
https://www.geneseo.edu/~baldwin/csci141/spring2004/0316binsearch.html
https://math.stackexchange.com/questions/117078/proof-of-correctness-of-binary-search

Complexity Analysis

TODO

Generalization and Extension

General Search

Binary search is a specific example of **Search By Reduction**, and thus we can utilize the idea behind it, **reducing the problem size in half**, to solve problems with certain property that does not necessarily resemble fully sorted array. Fully sorted ordering is a specific case of monotonicity, which is also a specific case of mutually exclusive property. To generalized this idea, we extend the premise, discuss dealing with multiple answers and how to use binary search partially. We call this the general application of binary search, or in short, general search, on which we are going to define some important elements below to help tackle with a broader range of problems.

Three Sections of Search

The Premise of Binary Search can be further generalized and abstracted. For a given array with some properties, the target entry/position of a search problem may not be unique, that is there may be multiple correct answers for that particular problem. These correct answers form an **answer section**, which may have 0, 1, or more than 1 entries. There are two typical cases for the answer section:

1. The answer section is always continual for the input array, which means all correct answer positions are adjacent to each other.
    - As a result of the continuity, we can partition the array into three sections: **lower section**, **answer section** and **upper section**, with each one having some properties related to our problem.
    - For this partition, the **Premise** can be generalized as every entry comes before **answer section**, **lower section**, has some common property; every entry comes after **answer section**, **upper section**, has some common property; and **answer section** itself has some common property. If these properties are mutually exclusive, we can apply binary search; and if it's not case we need to look closer to see if we can apply binary search partially.
2. The answer section may be partitioned into multiple continual subsequences, which means we may find non-answer entries between two correct answer positions.
    - We don't have **lower section** or **upper section** in this case, where the property of **answer section** is, generally, not related to all the entries come before and after it but only related to a portion of entries come before and after it, and in most cases it's related to its predecessor and successor.
    - For this partition, the **Premise** can be generalized as some entries comes before **answer section** have some common property; some entries come after **answer section**, have some common property; and **answer section** itself has some common property. If these

properties are mutually exclusive, we can apply binary search partially, which means we might not be able to halve the problem size in every iteration.

Properties and Predicates

We used the word **property** to denote the condition that the entries from answer/lower/upper sections meet. For typical binary search problems, these properties are mutually exclusive, which means they are the sufficient and necessary conditions for determining which section an entry belongs to.

In mathematical terms, each property corresponds to a **predicate** $P(i)$, and we use the predicate of a section as its branch condition:

1. $P_A(i) = T \leftrightarrow a[i] \in Answer$
2. $P_L(i) = T \leftrightarrow a[i] \in Lower$
3. $P_U(i) = T \leftrightarrow a[i] \in Upper$

When coding the branch conditions in the loop, we can split the predicate of a section into multiple sufficient condition to avoid redundant calculation. And the negation nature of **if-else** block allow us to explicitly express the condition that's easier to write.

Defining the Answer Section

Answer section is equivalent to the set of entries/positions that the problem demands, **so even for the same array, answer section varies as the search query varies**. Thus answer section is a relative concept, and defining the property of the answer section according to the problem description is the most important step for applying binary search.

To appreciate the cases we need to define the answer section multiple times for a single problem, notice how these two problem differs:

1. Search for a key in a sorted array that may have duplicate entries. All occurrence of the target key form the answer section, and we only need to find one of them.
2. In contrast, if the problem asks to find all the occurrences of a key in a sorted array that may have duplicates. All occurrence of the target key form the answer section, and we have to locate all of them.

There are multiple strategies to solve the second problem**(LC34)**, and if we try to solve it by finding the first and the last position of the answer section, which are two subproblems of the original problem with their own answer section(the first and the last position for the input key). For this approach, there are two layers of answer sections.

Property of Answer Section

Often we neglect the property of entires in **answer section**, focusing only on the properties of **lower section** or **upper section**. In many cases it's a simple equality relation but sometimes they have more complex properties. Notice that the property of **answer section** is typically exclusive to that of **lower section** or **upper section** and the entries need to be continual, otherwise we may need to do some modification to make the search work(such as including linear search).

During the binary search, we can always examine the entry pointed by **mid** to see if it satisfy the property of **answer section** and if it does we don't have to search anymore. This is the idea underlying the **Inclusive Template**, and can be incorporated into **Exclusive Templates** too.

For example, in **LC278**, typically we can use the **Exclusive Template Lower Method** to narrow down the subsequence of the bad version into one entry, which is the answer. But we can also check, when **mid** currently points to a bad version, whether its predecessor is a good version, and if it is, return the index directly.

Search inside a Subsequence with Common Property

For the problem of searching for an entry that has common properties with a subsequence that contains it, we can use binary search to locate the the subsequence, by finding its first and last entries with **Exclusive Template**. **These are the two entries that be directly found, while the entries between them not so, unless we use a long and complex predicate**. We typically first find the first or the last entries and then use the offset between it and the target key.

For example in a **sorted array with duplicate numbers(LC34)**, finding any occurrence of that number is easy(Inclusive or both Exclusive Templates will work), and the first and the last are also easy, because the entries come before and after them have exclusive properties, for example the entries coming before the first index are strictly smaller than key, and the entries come after it are larger than or equal to it. But finding other **Nth** cannot be directly done. Because the overlapping properties of entries that come before and after them, so that we cannot move **low** or **high** pointers when **a[mid]=key**: the correct index could come before or after it.

Disjunction of Properties

An interesting case of the premise: **The list is partitioned into multiple disjoint subsequences that satisfy mutually exclusive properties with respect to a certain target key.** This is essentially equivalent to the **premise** because we can take the disjunction of the predicates of the properties of the subsequences, which are mutually exclusive, that come before the key and that come after it to derive two subsequences with

disjoint predicates.

Moreover, the properties of the subsequences inside the lower section or the upper section may not be mutually exclusive properties, but as long as all the entries come before the key and all the entries come after it are exclusive, we can apply binary search. This usually results in somehow complex branching conditions for update **low** and high pointers.

Problems involving searching in a rotated array, such as **LC33**, **LC81**, **LC153**, **LC154**, are examples of properties disjunction. For an entry in the first or the second half of the array, its lower and upper sections have exclusive properties with regard to the target key and the pivot(first entry of the rotated array).

Intersection between Properties

Before we assume that only the properties between **lower section** and **upper section** are mutually exclusive. Well if they are exclusive we can apply binary search, but sometimes even when whey have some intersection we can apply binary search partially, for some iterations. The intersection between two properties can be thought of a predicate, which evaluates TRUE for some entries from both **lower section** and **upper section**, which essentially corresponds to a set of entries. If **mid** doesn't point to this set of entries, we can still move **low** or **high** pointers to narrow the subsequence, because it points to a entry that is only possible from **lower section** or **upper section** but not both.

This is the case of **LC81**, where we move **low** and **high** pointers by comparing the entries pointed by them with **key**, if **mid** points to **pivot**. Note that we only move one position at a time and then perform the binary search again instead of degenerating into linear search(see the implementation to get a better idea).

Binary Search with Linear Search is usually a bad idea but sometimes inevitable, but still we need to try to use binary search whenever possible.

Searching for Arbitrary Answer

There are variants of problems that have non-continual answer sections, e.g. **LC162**. For these problems, we are often asked to find any occurrence of the answer, which means we are asked to search for one arbitrary answer. In this case, binary search might be applicable, where we don't have a clear definition of lower or upper sections, but by examining an entry(the midpoint one if we were to use binary search) and some entires situated near it, we might be able to reduce the problem size(not necessary by half though). This is a more general case of **Search by Reduction**.

LC162, a variant of **LC852**, has multiple potential peaks(mountains) in the input array. The two constrains, $a[i] \neq a[i+1]$ for all valid i and $a[-1] = a[n] = -\infty$ are essential to make binary search possible. The subsequence of interest, denoted as $[j, k]$ always has the property that $a[j-1] < a[j]$ and $a[k] > a[k+1]$, this is TRUE even when $j = 0$ or $k = n-1$. For each iteration, we maintain this invariant by examining $a[mid-1], a[mid], a[mid+1]$ and depending on their relative relation updating j or k. The size of the subsequence shrinks and eventually we will end up with only one entry, a certain i that makes $a[i-1] < a[i] > a[i+1]$.

For this kind of problem, the key is to make sure each time we reduce the size of the problem, there is alway at least one answer in the subsequence after the shrink(there might be answers outside the shrunk subsequence but that is ok). How to ensure this? Mathematical deduction.

Presence of Single Answer

After checking the out-of-bound cases before the major loop, if we are certain that answer section must be present and has only one element, we can simplify the predicates as the following ways

Using the Predicate of Lower Section:

- $P_L(i) = L(i)$
- $P_A(i) = !L(i) \wedge (i = 0 \vee L(i-1))$
- $P_U(i) = i > 0 \wedge !L(i-1)$

Using the Predicate of Upper Section:

- $P_L(i) = i < n-1 \wedge !U(i+1)$
- $P_A(i) = !U(i) \wedge (i = n-1 \vee U(i+1))$
- $P_U(i) = U(i)$

Using the Predicate of Answer Section:

- $P_L(i) = i < n-1 \wedge A(i+1)$
- $P_A(i) = A(i)$
- $P_U(i) = i > 0 \wedge A(i-1)$

Applying General Search

The crux of general search is the identification of properties of answer section, lower section and upper section.

1. Define the answer section
2. Identify the properties of lower section, answer section and upper section
3. Choose a variant of a template to find the answer

Inclusive and Exclusive Templates

Defining the three sections and describing their properties as predicates gives us a good idea whether binary search is applicable to the problem at hand. What comes next is to combine the three elements of binary search in a certain fashion to give a workable implementation.

There are many ways to do this, and I find that the Inclusive Templates and the Exclusive Templates are easiest to write and efficient enough for most problems. For the specific problem of searching in a fully sorted array with distinct values, they are intuitive, but to apply them to a broader range of problem, we need to delve deeper, to see why they work and what is the distinction.

The Inclusive or the Exclusive Templates work because the input array meet the **Premise** of binary search with typically three sections that have mutually exclusive properties. Every time we pick the midpoint entry to see which section it falls in, we can trim down about half of the subsequence thus leaving a smaller subproblem. Based on this premise, the Inclusive Templates and the Exclusive Templates use different conditions/properties to halve the problem:

Inclusive Templates

```
int Inclusive_Template_Answer(Array<Entry> array, Entry key) {
    int low = 0;
    int high = array.length() - 1;
    while(low <= high) {
        int mid = low + (high - low) / 2;
        Entry entry = array.get(mid);
        if (P_L(entry)) {               // entry is in lower section
            low = mid + 1;
        } else if (P_U(entry)) { // entry is in upper section
            high = mid - 1;
        } else {                                // entry is in answer section
            return mid;
        }
    }
    return NOT_PRESENT;
}
```

```
int Inclusive_Template_Lower(Array<Entry> array, Entry key) {
    int low = 0;
    int high = array.length() - 1;
    while(low <= high) {
        int mid = low + (high - low) / 2;
        Entry entry = array.get(mid);
        if (P_U(entry)) {               // entry is in upper section
            high = mid - 1;
        } else if (P_A(entry)) { // entry is in answer section
            return mid;
        } else {                                // entry is in lower section
            low = mid + 1;
        }
    }
    return NOT_PRESENT;
}
```

```
int Inclusive_Template_Upper(Array<Entry> array, Entry key) {
    int low = 0;
    int high = array.length() - 1;
    while(low <= high) {
        int mid = low + (high - low) / 2;
        Entry entry = array.get(mid);
        if (P_L(entry)) {               // entry is in lower section
            low = mid + 1;
        } else if (P_A(entry)) { // entry is in answer section
            return mid;
        } else {                                // entry is in upper section
            high = mid - 1;
        }
    }
    return NOT_PRESENT;
}
```

Exclusive Templates

```
int Exclusive_Template_Lower(Array<Entry> array, Entry key) {
    int low = 0;
```

```
    int high = array.length() - 1;
    while(low < high) {
        int mid = low + (high - low) / 2;
        Entry entry = array.get(mid);
        if (P_L(entry)) { // entry is in lower section
            low = mid + 1;
        } else {                  // entry is in upper section or answer section
            high = mid;
        }
    }
    return isAnswer(array.get(low)) ? low : NOT_PRESENT;
}
```

```
int Exclusive_Template_Upper(Array<Entry> array, Entry key) {
    int low = 0;
    int high = array.length() - 1;
    while(low < high) {
        int mid = low + (high - low + 1) / 2;
        Entry entry = array.get(mid);
        if (P_U(entry)) { // entry is in upper section
            high = mid - 1;
        } else {                  // entry is in lower section or answer section
            low = mid;
        }
    }
    return isAnswer(array.get(low)) ? low : NOT_PRESENT;
}
```

Principle of Applying Inclusive and Exclusive Templates

I didn't enumerate all variants of Inclusive and Exclusive Templates, just five typical ones of them. In practical use, we may need to tweak them to achieve simpler codes or better performance. For example, the order of branches in Inclusive Templates can be adjusted according to the probability of execution. Moreover, using the idea of complement, after writing out the predicate of lower section in **Exclusive_Template_Lower**, we can take its negation and use that as the disjunction of the predicate of upper and answer sections as the branch condition if it's simpler(e.g. property for lower section is **entry<=key**, so the negation of it is entry>key which is simpler). The same is true for **Exclusive_Template_Upper**.

The core distinction between three variants of Inclusive Templates and two variants of Exclusive Templates is whether we have a dedicate branch for entries in answer section, which in turn determines how many predicates we need to explicitly express. In Inclusive Templates we do but in Exclusive Templates we don't. As a result of this distinction, Exclusive Templates have faster loops than their Inclusive counterparts because of fewer branches, which ease the demand of coding predicates, while Inclusive Templates may exit the loops earlier as soon as it locate the answer section, but it require us to express two predicates.

If the answer section is continual for the input arrays under a particular search query, we can use any of the above five equivalent implementations to solve the problem. As a result of their equivalence, they can be converted to one another interchangeably. But among these five, which template and which variant is best for the problem? It depends on the predicates of the three sections.

To program a binary search problem we need to convert the property of three sections into predicates that serve as branch conditions, encoding them with a boolean expression that returns TRUE or FALSE. Some predicates are expensive to check or harder to code compared to others. **The best practice of good binary search implementation is to always stick to the predicates that are easiest to code, simplest in terms of computation and have the largest probability with regard to the data set**. More complex expression often leads to more codes, more checks on out-of-bound cases and are thus more error-prone.

In practice, Exclusive Templates are more useful and easier to apply, where we only need to precisely express the predicate of either lower or upper section, because in most programming problems that utilize binary search only one of the three sections has a relatively simpler predicate. Precisely defining all three predicates $P_A(i), P_L(i), P_U(i)$ help us to have a rigorous mathematical definition of the problem, but a correct and efficient implementation does not need all of them.

We will analyze this implementation choice for every problem we encounter.

Implementation Notes

1. For a problem where the correct answer is always present in the array, or we can check the marginal cases where the answer is absent from the array easily before the loop leaving a problem for binary search where the answer is always present, the exclusive templates promise to find that answer, that is **isAnswer(array.get(low)**) is always TRUE. Thus we can return **low** directly.
2. Exclusive Templates are often more easy to code than Inclusive Templates but less effective when the predicate of the answer section is easy to express. On the other hand, if the predicate of the answer section is hard to express and only one of the lower and upper sections' is easy to express, using the corresponding Exclusive Template is preferable.
3. Choose the computationally cheapest one when they are all easy to code.
4. The condition for the binary if-else is usually the exclusive part(lower section or the upper section) of the partition, because the inclusive part includes two sections(lower section/upper section+answer section), so its branch condition, translated from the disjunction of predicates of the two sections is usually more complicated

5. Check the out-of-bound and marginal cases before the major loops can rule out many cases that may cause a headache when programming the predicates of sections and thus result in simpler codes as the branch conditions and perhaps a smaller subsequence to search. An alternative approach is to include the checks, typically checking the when **mid=0** or **mid=n-1** inside the loop but before the main branches, which is also applicable but slower than its counterpart outside the loop.
6. We can't use the disjunction of predicates of lower section and upper section, for an entry lying in one of them gives us no clue on how to update **low** or **high**.
7. Beware of marginal cases: **Absence of the Answer Section**! We can translate the properties of three sections into predicates, but we cannot guarantee that each of them is present in the array. That means after the loop exits, for Inclusive Templates **low** is larger than **high**(both of them may be out-of-bound indexes) indicating the answer section is not present. While for Exclusive Templates, the last remaining entry may not be the answer section because unlike Inclusive Templates we never explicitly test the predicate of answer section. There are typically two ways to handle this:
    i. We explicitly check whether the remaining entry is the answer like we did in the templates. This approach is preferable when the predicate of answer section is simple to check, and it’s a general approach, applicable to all kinds of problems.
    ii. We check whether there is at least one answer in the array before the major loop thus ensuring that we are always doing binary search on an array that has an answer. This approach is preferable when the property of the answer section is hard to check but only applicable for some problems.
8. In cases where answer section has multiple elements, such as searching for a key in an array with duplicate entries, the following properties hold:
    i. The Inclusive Templates returns any of the occurrence.
    ii. The Exclusive_Template_Lower always return the leftmost occurrence. Generally, if the entires of a subsequence share a common property, Exclusive_Template_Lower can be used to find its first member.
    iii. The Exclusive_Template_Upper always return the rightmost occurrence. Generally, if the entires of a subsequence share a common property, Exclusive_Template_Upper can be used to find its last member.

Typical Extension

Symbol Table Queries

Rank queries, which ask about the number of entries come before or after a given **key**, can be answered by using the position of the leftmost or the rightmost position, if **key** is present in the array.

TODO, need to update.

Search for First and Last Position

Problem Description: **Search for the starting and ending position of a given target value in a sorted array and return [-1, -1] if it's absent.**

```java
public int[] searchRangeByFirstAndLastOccurrence(final int[] array, final int key) {
    if (array == null || array.length == 0) return new int[]{-1, -1};
    if (array[0] > key || array[array.length - 1] < key) return new int[]{-1, -1};

    final int firstIndex = BinarySearch.binarySearchExclusiveLower(array, key);
    if (firstIndex == -1) {
        return new int[]{-1, -1};
    }

    final int lastIndex = BinarySearch.binarySearchExclusiveUpper(array, key);
    return new int[]{firstIndex, lastIndex};
}
```

Search for the Kth Position

Problem Description: **Search for the position of the Kth occurrence of a given target value in a sorted array and return -1 if it's absent.**

```java
public int searchKthOccurrence(final int[] array, final int key, final int k) {
    if (k <= 0) return -1;
    if (array == null || array.length == 0) return -1;
    if (array[0] > key || array[array.length - 1] < key) return -1;

    final int firstIndex = BinarySearch.binarySearchExclusiveLower(array, key);
    if (firstIndex == -1) {
        return -1;
    }
    if (firstIndex + k - 1 > array.length || array[firstIndex + k - 1] != key) {
        return -1;
    }
    return firstIndex + k - 1;
}
```

Search for Ceiling

Problem Description: **Search for the smallest entry larger than or equal to a given key. If it's absent, return the size of the array.**

This implementation also returns the rank of a given key(the number of entries smaller than it) no matter whether it is present or not.

```java
public static int binarySearchCeiling(final int[] array, final int key) {
    if (key > array[array.length - 1]) return array.length;
    if (array[0] >= key) return 0;

    int low = 0;
    int high = array.length - 1;

    while (low < high) {
        final int mid = low + (high - low) / 2;
        final int entry = array[mid];
        if (entry < key) {
            low = mid + 1; // subsequence becomes [mid+1,high]
        } else {
            high = mid; // subsequence becomes [low,mid]
        }
    }

    return low;
}
```

Search for Floor

Problem Description: **Search for the largest entry smaller than or equal to a given key. If it's absent, return -1.**

```java
public static int binarySearchFloor(final int[] array, final int key) {
    if (key >= array[array.length - 1]) return array.length - 1;
    if (key < array[0]) return -1;

    int low = 0;
    int high = array.length - 1;

    while (low < high) {
        final int mid = low + (high - low + 1) / 2;
        final int entry = array[mid];
        if (entry > key) {
            high = mid - 1; // subsequence becomes [low,mid]
        } else {
            low = mid; // subsequence becomes [mid,high]
        }
    }

    return low;
}
```

Search for Predecessor

Problem Description: **Search for an entry in a sorted array that is the largest number smaller than a given key and return its index if it's present or -1 if it's absent.**

The following three implementations work when there are duplicate entries in the array.

```java
public static int binarySearchPredecessor(final int[] array, final int key) {
    int low = 0;
    int high = array.length - 1;

    while (low < high) {
        final int mid = low + (high - low + 1) / 2;
        final int entry = array[mid];
        if (entry < key) {
            low = mid; // subsequence becomes [mid,high]
        } else {
            high = mid - 1; // subsequence becomes [low,mid-1]
        }
    }

    return low;
}
```

```java
public static int binarySearchPredecessor2(final int[] array, final int key) {
    int low = 0;
    int high = array.length - 1;

```

```
    while (low < high) {
        final int mid = low + (high - low + 1) / 2;
        final int entry = array[mid];
        if (entry > key) {
            high = mid - 1; // subsequence becomes [low,mid-1]
        } else if (entry < key) {
            low = mid; // subsequence becomes [mid,high]
        } else {
            return mid - 1;
        }
    }

    return low;
}
```

```
public static int binarySearchPredecessor3(final int[] array, final int key) {
    int low = 0;
    int high = array.length - 1;

    while (low < high) {
        final int mid = low + (high - low + 1) / 2;
        final int entry = array[mid];
        if (entry > key) {
            high = mid - 1; // subsequence becomes [low,mid-1]
        } else {
            low = mid; // subsequence becomes [mid,high]
        }
    }

    return array[low] == key ? low - 1 : low;
}
```

The codes handling **out-of-bound cases** for finding **predecessor**, which is common to all implementations above:

```
    if (array == null || array.length == 0 || array[0] >= key) return -1;
    if (array[array.length - 1] < key) return array.length - 1
```

Search for Successor

Problem Description: **Search for an entry in a sorted array that is the smallest number larger than a given key and return its index if it's present or -1 if it's absent.**

The following three implementations work when there are duplicate entries in the array.

```
public static int binarySearchSuccessor(final int[] array, final int key) {
    int low = 0;
    int high = array.length - 1;

    while (low < high) {
        final int mid = low + (high - low) / 2;
        final int entry = array[mid];
        if (entry > key) {
            high = mid; // subsequence becomes [low,mid]
        } else {
            low = mid + 1; // subsequence becomes [mid+1,high]
        }
    }

    return low;
}
```

```
public static int binarySearchSuccessor2(final int[] array, final int key) {
    int low = 0;
    int high = array.length - 1;

    while (low < high) {
        final int mid = low + (high - low) / 2;
        final int entry = array[mid];
        if (entry < key) {
            low = mid + 1; // subsequence becomes [mid+1,high]
        } else if (entry > key) {
            high = mid; // subsequence becomes [low,mid]
        } else {
            return mid + 1;
        }
    }

    return low;
}
```

```java
public static int binarySearchSuccessor3(final int[] array, final int key) {
    int low = 0;
    int high = array.length - 1;

    while (low < high) {
        final int mid = low + (high - low) / 2;
        final int entry = array[mid];
        if (entry < key) {
            low = mid + 1; // subsequence becomes [mid+1,high]
        } else {
            high = mid; // subsequence becomes [low,mid]
        }
    }

    return array[low] == key ? low + 1 : low;
}
```

The codes handling **out-of-bound cases** for finding **successor**, which is common to all implementations above:

```java
    if (array == null || array.length == 0 || array[array.length - 1] <= key) return -1;
    if (array[0] > key) return 0;
```

Search for Closest Elements

Problem Description: **Given a integer array $a[]$, two integers k and x, return the k closest integers to x in the array.**

Supposing $n = a.length$, this problem only makes sense when $k < n$. Three candidate solutions:

1. Brute-Force solutions are easy, we simply calculate the distance between every entry in the array and x and return the k smallest ones, whose complexity is $O(n \log n)$, $O(n)$ to calculate n absolute values and another $O(n \log n)$ to sort them or storing them in a priority queue.
2. A $O(\log n + k)$ solution with two phases is intuitive. We first find the floor or the ceiling of the input key with binary search, and then use two pointers to find the k closest elements. This is effective in most scenarios, but gets slow when k is large. For example, if k is fixed as a constant factor of n, the complexity will become $O(n)$. Note that if the input key is absent from the array, the floor/ceiling we find with binary search might not be the element closest to it, which is common pitfall, although the closest element must be one of them.
3. A $O(log(n-k))$ solution which also uses binary search is the main concern of this section. Rather than search for the floor/ceiling of the input key, we search for the answer section directly. Although being faster and arguably the optimal solution under most scenarios, this solution is not so general as the previous two, as the implementation details are somehow related to the nuance of the problem description, which we shall see.

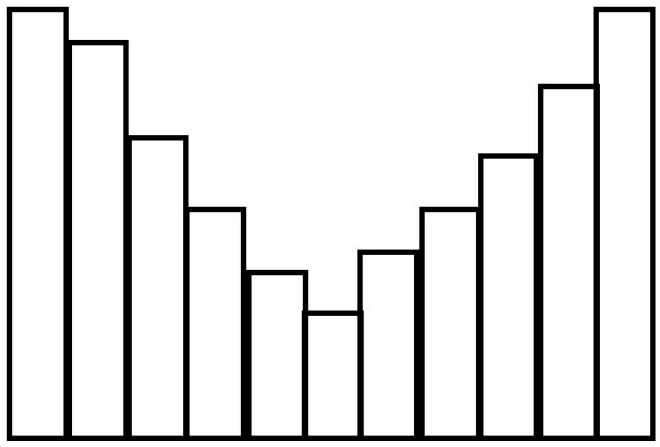
The problems have several variants and are closely tied to applying binary search, which we consider below:

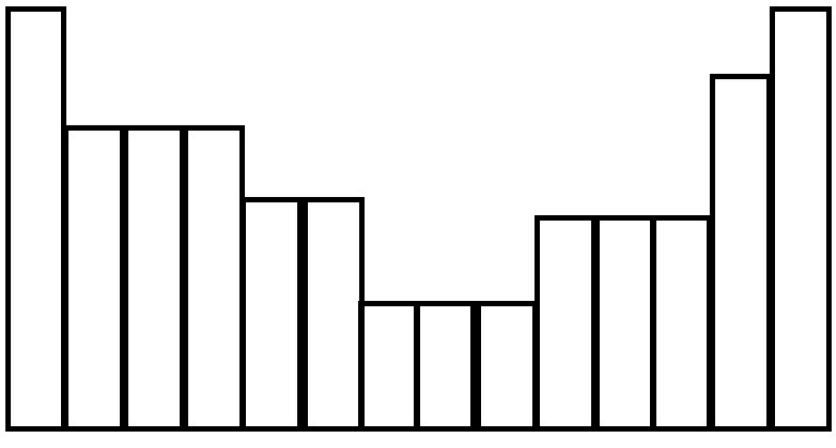
1. The input array can be sorted or unsorted. If it's unsorted we need to sort it first in order to apply binary search, in which case the performance gain of applying binary search is not so obvious.
2. The result, which is a set of integers, may be required to be sorted too. This is trivial as we can return them as a combination(where order doesn't matter) or permutation(where order matters).
3. The problem may ask us to exclude the presence of x, if present, in the return result. This requires us to first search for the first and the last occurrence of x to know the number of its occurrence $xCount$ and search for the $k + xCount$ closest entries then.
4. A tricky aspect of the return result is whether duplicate elements are allowed. If the problem asks us to return k unique entries in the array that are closest to a given x and there might be duplicate entries in the input array, the $O(log(n-k))$ solution is unproductive because the answer section might not be continuous, use the $O(\log n + k)$ solution instead. From now on we suppose duplicate elements are allowed in return result.
5. When $k = 1$, the $O(\log n + k)$ is preferable because we only need constant operations to locate the closest element after we locate the floor/ceiling of x.
6. How we break the tie when two distinct elements have the same distance from x greatly impacts the implementation of the $O(log(n-k))$ solution. An integer a is closer to x than an integer b if $|a-x| < |b-x|$ or
   i. $|a-x| = |b-x|$ and $a < b$. We call this the **left-closer case**.
   ii. $|a-x| = |b-x|$ and $a > b$. We call this the **right-closer case**.
7. The input array may contain duplicate entries, which further complicates the analysis of the $O(log(n-k))$ solution. In short, if the duplicate entries are absent, the answer section is a unique subsequence; otherwise the answer section have multiple subsequence and we are searching for an arbitrary answer.

Following are some useful notation for analyzing the problem:

- Suppose the entries that are closest to x comprise a set, and the corresponding array indexes comprise another set M. We refer to any one of them as $a[m]$, thus $max(m)$ has the largest index and $min(m)$ has the smallest index.
- Suppose the answer section is a subsequence of the input array, denoted as $a[l..h]$, where $h - l + 1 = k$. We will explain later why the closest entries must be a continuous. As we shall see, for **left-closer case**, we denote $ans = l$, and for **right-closer case**, we denote $ans = h$.
- Suppose $dis[i] = |x - a[i]|$. How $dis[i]$ varies as i varies depends on whether duplicate entries are allowed in the input array.

We document the analysis of the **left-closer case** first. The problem essentially asks us to find k entries that have the smallest dis. **A very important insight is that such k entries must exist in the array once we are certain that $k \leq n$.** The certainty of the presence of the answer section allow us to simplify the branch conditions.

- If there are no duplicate entries.
  - The plot of $dis[i]$ resembles a valley. The adjacent entries won't have same value of dis except for one case, where the the input key is missing and its successor and predecessor are present and have the same distance from it. Overall, one or two entries(the aforementioned case) has the smallest value of dis, that is the size of M is either 1 or 2.
  - For $i \in [0..min(m)]$, $dis[i]$ is **monotonically decreasing**. For $i \in [max(m)..n-1]$, $dis[i]$ is **monotonically increasing**. As a result of this monotonicity, the closest entries must comprise a continuous subsequence of length k.
  - Because the first index plus the length determines a unique subsequence. Thus we can simplify finding a subsequence into finding the first index of the subsequence, denoted as ans, making range of interest for this problem $[0..n-k]$, the subsequence we want to find becomes $[ans..ans+k-1]$. And we know for sure that ans is unique and must be present in $[0..n-k]$.
  - 
  - Although it's not obvious, this equivalent problem satisfy the premise of binary search due to the distribution of $dis[i]$. It takes a little thinking to see that choosing the first index as the answer section greatly simplifies the property of the lower section:
    - The lower section $[0..ans-1]$ is **monotonically decreasing**, because this is **left-closer case**, we have $ans \leq min(m)$.
    - The answer section has only one entry $a[ans]$. It has many interesting properties but to find a necessary and sufficient condition for them is not so obvious.
    - The upper section $[ans+1..n-1]$ seems to have more complicates properties as it may have both **monotonically decreasing** and **monotonically increasing** subsequences simultaneously.
  - From the **Principle** we know that if we can explicitly code the sufficient and necessary condition for the properties of either lower or upper section, when the answer section is definitely present, we can use **Exclusive Template** to solve the problem. In this problem, explicitly coding the property of the lower section seems most promising, and the following is how we come to it:
    - We know that dis in the lower section is **monotonically decreasing**, but $dis[i] < dis[i+1]$ cannot be its property because it may also be TRUE for the answer section and some entries in the upper section. The property of subsequence $[ans..ans+k+1]$ might give us some inspiration : any entry outside of it has a larger value of dis. Thus $dis[ans] \leq dis[ans+k]$, the equality is possible because this is **left-closer case**, but for $i \in [ans+1..ans+k+1]$, we have $dis[i] < dis[i+k]$. It can be deduced that dis in $[ans+k..n-1]$ is **monotonically decreasing**, because $ans+k \geq ans+1 \geq max(m)$, which makes $dis[i] < dis[i+k]$ TRUE in $[ans+1..n-1]$.
    - A few sample test show that $dis[i] \leq dis[i+k]$ is FALSE for sample entries in lower section, making $dis[i] > dis[i+k]$ a tempting choice for describing the property of lower section. We have already show that $dis[i] \leq dis[i+k]$ is TRUE for $[ans..n-1]$, which means $dis[i] > dis[i+k]$ is FALSE for it, thus for an i to make $dis[i] > dis[i+k]$ TRUE, i must belong to the lower section. This is equivalent to say $dis[i] > dis[i+k] \rightarrow i \in [0..ans-1]$, making it a sufficient condition for lower section.
    - To prove it's also a necessary condition for lower section, we need to prove $i \in [0..ans-1] \rightarrow dis[i] > dis[i+k]$. There are two cases: For $i+k < ans$, this is obvious TRUE because $[0..ans-1]$ is **monotonically decreasing** for dis; For $i+k \geq ans$, we already know that $i < ans$, thus $ans \leq i+k \leq ans+k-1$, which means $a[i+k]$ is among the k entries with smallest dis, and $dis[i] > dis[i+k]$ must be TRUE.
  - Now we have the predicate for lower section, we can apply the **Exclusive Template**. In fact we can write down the properties of three sections explicitly:
    - Lower Section: $L(i) = dis[i] > dis[i+k]$
    - Answer Section: $!L(i) \wedge (i = 0 \wedge L(i-1))$, or equivalently,
      $dis[i] = dis[i+k] \ \vee (dis[i] < dis[i+k] \wedge (i = 0 \vee dis[i-1] > dis[i+k-1])$
    - Upper Section: $i > 0 \wedge !L(i-1)$, or equivalently, $i > 0 \wedge dis[i-1] \leq dis[i+k-1]$(the equality is dues to choosing the smaller entry to break the tie)
  - The implementation can be further condensed and simplified: we can replace $L(i) = |x - a[i]| > |x - a[i+k]|$ with $P(i) = x - a[i] > a[i+k] - x$. Because we know $a[i] < a[i+k]$(the array is sorted and has no duplicates), thus for all three cases of i these two inequities are equivalent:
    - $a[i] < a[i+k] \leq x$, $P(i) = L(i) = T$.
    - $a[i] \leq x \leq a[i+k]$, $P(i) = L(i)$.
    - $x \leq a[i] < a[i+k]$, $P(i) = L(i) = F$.
- If there might be duplicate entries.
  - For an array with possible duplicates, the distribution of $dis[i]$ is not necessarily **monotonically decreasing** before $min(m)$, nor necessarily **monotonically increasing** after $max(m)$; any adjacent entires could have equal values of dis. We start by looking at the graph of $dis[i]$.

- It's still a valley but with potentially many equal adjacent entries, and the equality may occur anywhere in the array, not just between the predecessor and the successor. Thus the size of M could be any number in $[1..n]$.
- For $i \in [0..max(m)]$, $dis[i]$ is **non-increasing**; for $i \in [min(m)..n-1]$, $dis[i]$ is **non-decreasing**. Another useful property is that for $a[i] < x$, $dis[i]$ is **non-increasing**; for $a[i] > x$, $dis[i]$ is **non-decreasing**.
- Notice something troublesome: as a result of duplicate entries, the index interval $[l, h]$ of the answer is not unique, which means multiple answer sections is possible. If we were to use binary search, we can search for arbitrary answer. For a concrete example : [1,1,2,2,2,2,2,4,4] with $k = 3$, $x = 3$, [2,4], [3,5], [4,6] are all correct answers.
- 
- We can still search for the first index ans of the subsequence directly in $[0..n-k]$, but with a slightly different properties of three sections. Because of the presence of duplicate entries, we need to treat equality carefully.
- Notice that $ans \le max(m)$ and $ans + k - 1 \ge min(n)$ are always TRUE for any viable ans, not just the one with the largest index. Because if $ans > max(m)$, or $ans + k - 1 < min(n)$, the subsequence doesn't contain the entries that are closest to x. This tells us that lower section is **non-increasing**.
- As with the case where duplicates are not allowed, we are reasonable to believe that the property of lower section is the simplest. The reasoning is similar so we omit it here. Our main concern is to find its predicate. We start immediately from the relation between $dis[i]$ and $dis[i+k]$:
  - We first try to prove that $dis[i] > dis[i+k] \rightarrow i < ans$, which is equivalent to $i \ge ans \rightarrow dis[i] \le dis[i+k]$, its contrapositive.
    - If $ans \le i \le ans + k - 1$, we have $dis[i] \le dis[i+k]$, which is obvious.
    - If $ans + k - 1 < i \le n - 1$, we have $i \ge min(n)$, $dis[i]$ is **non-decreasing**, so $dis[i] < dis[i+k]$.
  - However, $i < ans \rightarrow dis[i] > dis[i+k]$ is not always TRUE. Because for $i < ans$, dis is **non-increasing** but not necessarily **monotonically increasing**, thus $dis[i] = dis[i+k]$ is possible. $dis[i] = dis[i+k]$ is the most complicated case that require our most attention:
    - $a[i] = a[i+k] < x$, we have $dis[i] = dis[i+1] = \ldots = dis[i+k-1] = dis[i+k]$. As dis could potentially be further decreased as i increases, $[i+1..i+k]$ is better than $[i..i+k-1]$, so we conclude that i belongs to lower section.
    - $a[i] = x = a[i+k]$, we have $dis[i] = dis[i+1] = \ldots = dis[i+k-1] = dis[i+k] = 0$. $[i..i+k-1]$ and $[i+1..i+k]$ are both correct answers. We can conclude i belongs to either lower or answer section.
    - $x < a[i] = a[i+k]$, we have $dis[i] = dis[i+1] = \ldots = dis[i+k-1] = dis[i+k]$. As dis could potentially be further decreased as i decreases, we conclude that i belongs to either upper or answer section.
    - $a[i] < x < a[i+k]$, we know that dis in $[0..i-1]$ are no less than $dis[i]$ and dis in $[i+k+1..n-1]$ are no less than $dis[i+k]$.
      - If $i = 0$ or $a[i-1] < a[i]$, which means $dis[i-1] > dis[i]$, i is the answer section.
      - If $a[i-1] = a[i]$, there are two more cases depending on $a[i+k-1]$
        - If $a[i+k-1] > a[i]$, $[i-1..i+k-2]$ is better than $[i..i+k-1$, so i belongs to upper section.
        - If $a[i+k-1] = a[i]$, $[i-1..i+k-2]$ and $[i..i+k-1]$ are both correct answers.
  - The above analysis gives us the predicate of lower section as the union of all of its sufficient conditions:
    - $L(i) = dis[i] > dis[i+k] \vee (a[i] = a[i+k] \wedge a[i] \le x)$
    - $L'(i) = dis[i] > dis[i+k] \vee (a[i] = a[i+k] \wedge a[i] < x)$
- Can we code the predicate of answer section and upper section explicitly? Yes but it will be pretty complicated:
  - Answer section: $!L(i) \wedge (i = 0 \vee L(i-1))$
  - Upper section: $i > 0 \wedge !L(i-1)$
- $L'(i)$ is equivalent to $P(i) = x - a[i] > a[i+k] - x$, but $L(i)$ isn't:
  - $a[i] = a[i+k]$
    - $a[i] < x$, $P(i) = T$, $L(i) = T$, $L'(i) = T$
    - $a[i] = x$, $P(i) = F$, $L(i) = T$, $L'(i) = F$
    - $a[i] > x$, $P(i) = F$, $L(i) = F$, $L'(i) = F$
  - $a[i] \ne a[i+k]$
    - $a[i] < a[i+k] \le x$, $P(i) = T$, $L(i) = T$, $L'(i) = T$
    - $a[i] \le x \le a[i+k]$, $P(i) = L(i) = L'(i)$
    - $x \le a[i] < a[i+k]$, $P(i) = F$, $L(i) = F$, $L'(i) = F$

```java
public static List<Integer> binarySearchClosestElementsWithoutDuplicates(final int[] array, final int k, final int x) {
    if (x >= array[array.length - 1]) return getSubList(array, array.length - k, array.length - 1);
    if (x <= array[0]) return getSubList(array, 0, k - 1);

    int low = 0, high = array.length - k;
    while (low < high) {
        final int mid = low + (high - low) / 2;
```

```
        if (Math.abs(x - array[mid]) > Math.abs(x - array[mid + k])) {
            low = mid + 1;
        } else {
            high = mid;
        }
    }

    return getSubList(array, low, low + k - 1);
}
```

```
public static List<Integer> binarySearchClosestElementsWithDuplicates(final int[] array, final int k, final int x) {
    if (x >= array[array.length - 1]) return getSubList(array, array.length - k, array.length - 1);
    if (x <= array[0]) return getSubList(array, 0, k - 1);

    int low = 0, high = array.length - k;
    while (low < high) {
        final int mid = low + (high - low) / 2;
        if (Math.abs(x - array[mid]) > Math.abs(x - array[mid + k])
            || (array[mid] == array[mid + k] && array[mid] <= x)) {
            low = mid + 1;
        } else {
            high = mid;
        }
    }

    return getSubList(array, low, low + k - 1);
}
```

```
    public static List<Integer> binarySearchClosestElementsUnified(final int[] array, final int k, final int x) {
        if (x >= array[array.length - 1]) return getSubList(array, array.length - k, array.length - 1);
        if (x <= array[0]) return getSubList(array, 0, k - 1);

        int low = 0, high = array.length - k;
        while (low < high) {
            final int mid = low + (high - low) / 2;
            if (x - array[mid] > array[mid + k] - x) {
                low = mid + 1;
            } else {
                high = mid;
            }
        }

        return getSubList(array, low, low + k - 1);
    }
```

```
public static List<Integer> getSubList(final int[] array, final int low, final int high) {
    final List<Integer> subList = new ArrayList<>(high - low + 1);
    for (int i = low; i <= high; i++) {
        subList.add(array[i]);
    }
    return subList;
}
```

Why did we choose the first index of the subsequence as the answer section? Because it makes the predicate of the lower section much easier. For the same reason, we choose the last index of the subsequence as the answer section in the **right-closer case** and end up with a simple predicate of the upper section. We omit the analysis of the **right-closer case**, because everything is symmetrical to that of the **left-closer case**, simply substituting $>$ with $<$ and $<$ with $>$.

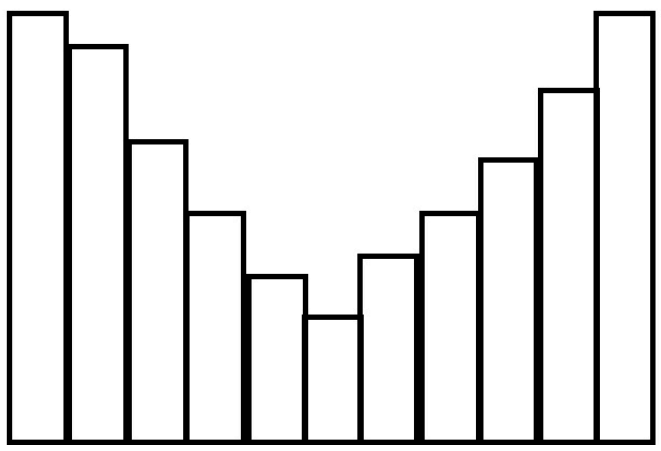
Notable Programming Problems

- LeetCode704 Binary Search
  - Basic Implementation. **We need to implement more bounded versions of binary search to enrich our source code library.**
  - Properties of three sections of search($0 \leq i < n$), if the input array has distinct values:
    - Lower Section: $a[i] < key$
    - Answer Section: $a[i] = key$
    - Upper Section: $a[i] > key$
- LeetCode35 Search Insert Position
  - **#SearchForSuccessor**, **#SearchForCeiling**
  - A combination of answers under different situations, and can be merged into one unified implementation.
    a. Return the index of **key** if it's present
    b. Return the index of the **successor** of **key** if it's absent
    c. Return the size of the array if **key** is absent and it doesn't have a **successor**
  - We can translate the the problem into: **search for the smallest index that points to an entry no less than an input key(the ceiling of the key), and if such index doesn't exist return the size of the array**.

    - Properties of three sections of search($0 \le i < n$), if the input array has distinct values:
        - Lower Section: $a[i] < key$
        - Answer Section: $a[i] = key \ \vee (a[i] > key \ \wedge (i = 0 \ \vee \ a[i-1] < key))$
        - Upper Section: $i > 0 \ \wedge \ a[i-1] \ge key$
        - Marginal Case: $a[n-1] < key$, return n
    - Properties of three sections of search($0 \le i < n$), if the input array may have duplicate values:
        - Lower Section: $a[i] < key$
        - Answer Section: $a[i] \ge key \ \wedge (i = 0 \ \vee \ a[i-1] < key)$
        - Upper Section: $i > 0 \ \wedge \ a[i-1] \ge key$
        - Marginal Case: $a[n-1] < key$, return n
    - It can be easily seen that the property of lower section is the simplest, so **Exclusive_Template_Lower** is the most intuitive choice to pick. Also note that if we check the cases of $a[0]$ outside the loop we can omit the check $i > 0$ inside the loop.
- LeetCode278 First Bad Version
    - **#SubsequenceWithCommonProperty**, **#SearchForFirstAndLastPosition**, **#PropertyOfAnswerSection**
    - Two solutions:
        a. The entries come before it are good versions and the entries come after it are bad versions, which determines an subsequence that includes both the last good version and the first bad version. Using the exclusive templates to find the first or the last occurrence of the the sequence can find both of them easily. In this case, only the first bad version is considered as the answer section.
        b. Alternatively we can utilize the property of the first bad version: 1. It's a bad version; 2. Version that comes before it is a good version.
    - For the first bad version problem, properties of three sections of search($1 \le i \le n$):
        - Lower Section: $a[i] = Good$
        - Answer Section: $a[i] = Bad \ \wedge \ a[i-1] = Good$
            - We omit checking $i > 1$ because the problem says there must be at least one good version
        - Upper Section: $i > 1 \ \wedge \ a[i-1] = Bad$
    - For the last good version problem, properties of three sections of search($0 \le i < n$):
        - Lower Section: $i < n \ \wedge \ a[i+1] = Good$
        - Answer Section: $a[i] = Good \ \wedge \ a[i+1] = Bad$
            - We omit checking $i < n$ because the problem says there must be at least one bad version
        - Upper Section: $a[i] = Bad$
- LeetCode34 Find First and Last Position of Element in Sorted Array
    - **#SubsequenceWithCommonProperty**, **#SearchForFirstAndLastPosition**, **#SearchForSuccessor**, **#SearchForPredecessor**
    - A brute-force way to combine binary search with linear search but may degenerate into O(n) linear search.
    - Two solutions:
        a. We find the first and the last position directly using the two Exclusive Templates.
        b. We find the predecessor and the successor using the modified Exclusive Templates and then find the first and last position.
    - The first approach is simpler and more straightforward while the second one needs to consider the marginal cases where one or both of the predecessor and the successor is absent.
    - To find the predecessor, properties of three sections of search($0 \le i < n$):
        - Lower Section: $i < n-1 \ \wedge \ a[i+1] < key$
        - Answer Section: $a[i] < key \ \wedge \ (i = n-1 \ \vee \ a[i+1] \ge key)$
        - Upper Section: $a[i] \ge key$
        - Marginal Case: $a[0] \ge ket$, key has no predecessor
    - To find the successor, properties of three sections of search($0 \le i < n$):
        - Lower Section: $a[i] \le key$
        - Answer Section: $a[i] > key \ \wedge \ (i = 0 \ \vee \ a[i-1] \le key)$
        - Upper Section: $i > 0 \ \wedge \ a[i-1] \ge key$
        - Marginal Case: $a[n-1] \le key$, key has no successor
    - To find the first occurrence, properties of three sections of search($0 \le i < n$):
        - Lower Section: $a[i] < key$
        - Answer Section: $a[i] = key \ \wedge \ (i = 0 \ \vee \ a[i-1] < key)$
        - Upper Section: $a[i-1] \ge key$
    - To find the last occurrence, properties of three sections of search($0 \le i < n$):
        - Lower Section: $a[i+1] \le key$
        - Answer Section: $a[i] = key \ \wedge \ (i = n-1 \ \vee \ a[i+1] > key)$
        - Upper Section: $a[i] > key$
- LeetCode33 Search in Rotated Sorted Array
    - **#DisjunctionOfProperties**, **#PropertyOfAnswerSection**
    - Two solutions:
        - Find the largest or the smallest entry (that is not equal to key) in the array, by binary search, to partition it into two subsequences and preform the typical binary search on one of the subsequences. (We can add codes to check whether pivot has a predecessor or a successor before the search by checking a[1] and a[n-1])
        - We don’t partition the array but instead search for key directly in the array with disjunctive multiple properties depending on which subsequence it falls in.
    - The implementations of solution 1 are good examples that illustrates how to deal with marginal cases. It's preferable to check first that

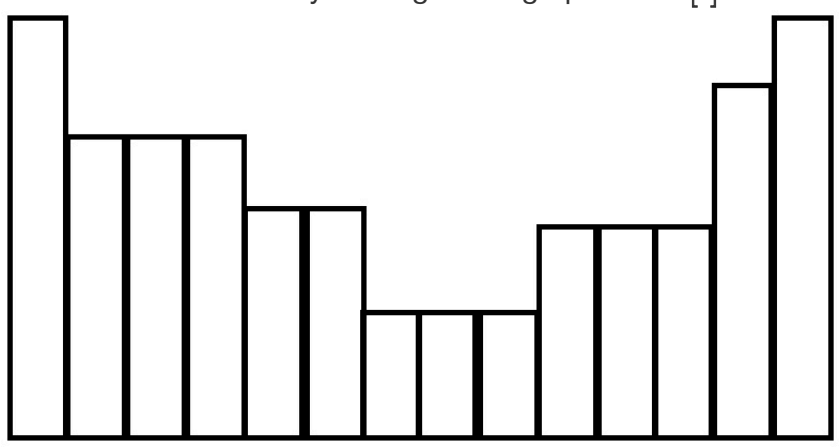
there is at least one entry in the array that is larger or smaller than pivot, otherwise we can apply the binary search directly on [1,n].
    - The direct seach approach actually disjunct multiple properties, and thus is a good example that illustrates that the three sections could have somehow complex properties.
    - Marginal Case: $pivot = key$, return 0
    - To find the largest entry, properties of three sections of search($1 \leq i < n$), if the input array has distinct values:
        - Lower Section: $i < n - 1 \ \wedge \ a[i + 1] > pivot$
        - Answer Section: $a[i] > pivot \ \wedge \ (i = n - 1 \ \vee \ a[i + 1] < pivot)$
        - Upper Section: $a[i] < pivot$
        - Marginal Case: $a[1] < pivot$, there is no entry larger than $pivot$
    - To find the smallest entry, properties of three sections of search($1 \leq i < n$), if the input array has distinct values:
        - Lower Section: $a[i] > pivot$
        - Answer Section: $a[i] < pivot \ \wedge \ (i = 1 \ \vee \ a[i - 1] > pivot)$
        - Upper Section: $i > 1 \ \wedge \ a[i - 1] < pivot$
        - Marginal Case: $a[n - 1] > pivot$, there is no entry smaller than $pivot$
    - $key > pivot$, search in the first half of the subsequence, properties of three sections of search($1 \leq i < n$), if the input array has distinct values:
        - Lower Section: $a[i] < key \ \wedge \ a[i] > pivot$
        - Answer Section: $a[i] = key$
        - Upper Section: $a[i] > key \ \vee \ a[i] < pivot$
    - $key < pivot$, search in the second half of the subsequence, properties of three sections of search($1 \leq i < n$), if the input array has distinct values:
        - Lower Section: $a[i] < key \ \vee \ a[i] > pivot$
        - Answer Section: $a[i] = key$
        - Upper Section: $a[i] > key \ \wedge \ a[i] < pivot$
- LeetCode81 Search in Rotated Sorted Array II
    - **#DisjunctionOfProperties**, **#IntersectionBetweenProperties**
    - Also two solutions but because of duplicate entries, we need to combine linear search with binary search.
    - For duplicate smallest or largest entries, we need to find the leftmost smallest entry or the rightmost largest entry. And the methods for LC33 work here.
    - To find the largest entry, properties of three sections of search($1 \leq i < n$), if the input array may have duplicate values:
        - Lower Section: $i < n - 1 \ \wedge \ a[i + 1] \geq pivot$
        - Answer Section: $a[i] > pivot \ \wedge \ (i = n - 1 \ \vee \ a[i + 1] \leq pivot)$
        - Upper Section: $a[i] \leq pivot$
        - We cannot easily check whether there is a least one entry larger than **pivot** in the array.
        - If we were to use the property of upper section as the branch condition, we cannot update **low** or **high** when $a[i] = pivot$, because it's the intersection between lower and upper section. Use a one step linear search instead. So in the worst case it takes O(n) time to finish the search. The refined partition is as follows:
        - Unsearchable Section: $a[i] = pivot$
        - Lower Section and Answer Section: $a[i] > key$
        - Upper Section: $a[i] < key$
    - To find the smallest entry, properties of three sections of search($1 \leq i < n$), if the input array may have duplicate values:
        - Lower Section: $a[i] \geq pivot$
        - Answer Section: $a[i] < pivot \ \wedge \ (i = 1 \ \vee \ a[i - 1] \geq pivot)$
        - Upper Section: $i > 1 \ \wedge \ a[i - 1] \leq pivot$
        - We cannot easily check whether there is a least one entry smaller than **pivot** in the array.
        - If we were to use the property of lower section as the branch condition, we cannot update **low** or **high** when $a[i] = pivot$, because it's the intersection between lower and upper section. Use a one step linear search instead. So in the worst case it takes O(n) time to finish the search. The refined partition is as follows:
        - Unsearchable Section: $a[i] = pivot$
        - Upper Section and Answer Section: $a[i] < key$
        - Lower Section: $a[i] > key$
    - $key > pivot$, search in the first half of the subsequence, properties of three sections of search($1 \leq i < n$), if the input array has distinct values:
        - Unsearchable Section: $a[i] = pivot$
        - Lower Section: $a[i] < key \ \wedge \ a[i] > pivot$
        - Answer Section: $a[i] = key$
        - Upper Section: $a[i] > key \ \vee \ a[i] < pivot$
    - $key < pivot$, search in the second half of the subsequence, properties of three sections of search($1 \leq i < n$), if the input array has distinct values:
        - Unsearchable Section: $a[i] = pivot$
        - Lower Section: $a[i] < key \ \vee \ a[i] > pivot$
        - Answer Section: $a[i] = key$
        - Upper Section: $a[i] > key \ \wedge \ a[i] < pivot$
- LeetCode153 Find Minimum in Rotated Sorted Array

    - A subproblem of LC33
- LeetCode154 Find Minimum in Rotated Sorted Array II
    - A subproblem of LC81
- LeetCode274 H-Index
    - **#SubsequenceWithCommonProperty**
    - First note the range of h: $0 \leq H \leq n = a.size$, which is not directly related with the entries of the array.
    - After sorting the array, properties of three sections of search($0 \leq i < n$), if the input array might have duplicate values:
        - Suppose $count[i] = a.length - i$
        - Lower Section: $a[i] < count[i]$
        - Answer Section: $a[i] \geq count[i] \ \wedge (i = 0 \ \vee \ a[i-1] < count[i-1])$
        - Upper Section: $i > 0 \ \wedge \ a[i-1] \geq count[i-1]$
        - Marginal Case: if $a[n-1] = 0$, return 0
    - Is multiple H possible? Yes, only with duplicate entries larger than H.
    - Because we need to sort the array first, binary search and linear search doesn't impact the overall run time very much.
    - What makes this problem different from many other is that the value we seek is not the index or the content pointed by that index(the citation) but the distance between it and the end of the array. It's always a good habit to think about the possible range of the answer first.
    - Also note that $a[i] = count[i]$ is a sufficient but not necessary condition for $count[i]$ to be the answer.
- LeetCode275 H-Index II
    - A subproblem of LC274. Binary search is the fastest approach as the array is already sorted.
- LeetCode540 Single Element in a Sorted Array
    - **#PropertyOfAnswerSection**
    - Properties of three sections of search($0 \leq i < n$), assuming the single element is present in the array:
        - Lower Section: $(i > 0 \ \wedge \ a[i] = a[i-1] \ \wedge \ i\%2 \neq 0) \vee (i < n-1 \ \wedge \ a[i] = a[i+1] \ \wedge \ i\%2 = 0)$
        - Answer Section:
          $(n = 1) \vee (0 < i < n-1 \ \wedge \ a[i] \neq a[i-1] \ \wedge \ a[i] \neq a[i+1]) \vee (i = 0 \ \wedge \ a[i] \neq a[i+1]) \vee (i = n-1 \ \wedge \ a[i] \neq a[i-1])$
        - Upper Section: $(i > 0 \ \wedge \ a[i] = a[i-1] \ \wedge \ i\%2 = 0) \vee (i < n-1 \ \wedge \ a[i] = a[i+1] \ \wedge \ i\%2 \neq 0)$
    - None of the properties of three sections are simple, because we always need to make sure that $i-1$ and $i+1$ don't go out-of-bound, that is when $i = 0$ and $i = n-1$. It's desirable to check these marginal cases separately in order to get simpler expressions for three sections:
        - If $n = 1$, return $a[0]$
        - If $i = 0$, we have $low = 0$, $high = 1$, since the single element must present, return $a[0]$
        - If $i = n-1$, which is impossible, because this implies $low = high = n-1$, we are outside the loop.
    - Another way to help ease the pain of checking $i-1$ and $i+1$ don't go out-of-bound is to explicitly check whether $a[0]$ or $a[n-1]$ is the single element, which simply means check them against $a[1]$ and $a[n-2]$ respectively. Either we find the single element or we continue to use binary search with the subsequence $[1, n-2]$, where $i-1$ and $i+1$ will never go out-of-bound.
    - Checking the marginal cases before the branch conditions give us simpler expressions inside the loop:
        - Lower Section: $(a[i] = a[i-1] \ \wedge \ i\%2 \neq 0) \vee (a[i] = a[i+1] \ \wedge \ i\%2 = 0)$
        - Answer Section: $a[i] \neq a[i-1] \ \wedge \ a[i] \neq a[i+1]$
        - Upper Section: $(a[i] = a[i-1] \ \wedge \ i\%2 = 0) \vee (a[i] = a[i+1] \ \wedge \ i\%2 \neq 0)$
    - A variant of this problem is that the single element may be absent from the array, which leads to more complicated codes checking marginal cases inside the loop, but the branch conditions stay the same for three sections:
        - We use **low<=high** as the loop condition, so that we don't have to check the last remaining entry separately
        - If $n = 1$, return $a[0]$
        - If $i = 0$, we have $low = 0$, $high = 1$
            - If $a[0] = a[1]$, return -1
            - Else return $a[0]$
        - If $i = n-1$, we have $low = high = n-1$
            - If $a[n-1] = a[n-2]$ return -1
            - Else return $a[n-1]$
    - Alternatively we can check $a[0]$ and $a[n-1]$ outside the loop, which is better.
- LeetCode852 Peak Index in a Mountain Array
    - **#PropertyOfAnswerSection**
    - We know from description that the peak index is in $[1, n-2]$, and $n \geq 3$, which helps us to avoid some annoying marginal cases as $a[0]$ must belong to lower section and $a[n-1]$ must belong to upper section.
    - Properties of three sections of search($1 \leq i \leq n-2$):
        - Lower Section: $a[i-1] < a[i] < a[i+1]$
        - Answer Section: $a[i-1] < a[i] > a[i+1]$
        - Upper Section: $a[i-1] > a[i] > a[i+1]$
    - For a more general problem where the peak index is promised to be present but could be the first or the last entry, it's recommended that we check $a[0]$ and $a[n-1]$ before the loop(checking when $mid = 0$ or $mid = n-1$ is also correct but results in a slower loop).
- LeetCode162 Find Peak Element
    - **#PropertyOfAnswerSection**, **#SearchForArbitraryAnswer**
    - A variant of LC852, with multiple potential peaks(mountains) in the input array. The two constrains, $a[i] \neq a[i+1]$ for all valid i and $a[-1] = a[n] = -\infty$ are essential to make binary search possible.

    - The invariant we try to maintain is the subsequence of interest, denoted as $[low, high]$ always has the property that $a[low-1] < a[low]$ and $a[high] > a[high+1]$, this is already TRUE when $low = 0$ or $high = n-1$ because $a[-1] = a[n] = -\infty$.
    - For each iteration, we maintain this invariant by examining $a[mid-1], a[mid], a[mid+1]$. There are 4 possible cases:
        - $a[mid-1] < a[mid] > a[mid+1]$, a peak, return mid directly
        - $a[mid-1] < a[mid] < a[mid+1]$, set low as $mid+1$(use mid is also ok, but beware of the loop condition)
        - $a[mid-1] > a[mid] > a[mid+1]$, set $high$ as $mid-1$(use mid is also ok, but beware of the loop condition)
        - $a[mid-1] > a[mid], a[mid] < a[mid+1]$, mid is a trough, we can update either low or $high$
    - For any subsequence $[low, high]$ where $a[low-1] < a[low]$ and $a[high] > a[high+1]$, there is at least one peak in it. This can be proved by contradiction. Supposing that there is no peak in the subsequence, we have $\forall i \geq low,\ a[i] < a[i+1]$, because as i increases, the entry $a[i]$ can only be larger(equality of neighboring entries is impossible according to the description), otherwise a peak will occur. Thus we have $a[high] > a[high-1]$ and $a[high] > a[high+1]$, making $high$ a peak, a contradiction.
    - Because we always maintain the invariant while still reduce the problem size for each iteration, eventually we will end up with a peak when there is only one entry in the subsequence.
    - Turns out the code is identical to the general version of LC852
- LeetCode1095 Find in Mountain Array
    - Another variant of LC852, where we need to find the peak index first, which is unique. Then find the input key in the first half ascending subsequence and if it's absent search in the second half descending subsequence.
    - It's impossible to search for the given key in one pass of binary search, unlike searching in rotated array, because only the first half ascending and the second half descending arrays meet the premise of binary search respectively, but the entire array doesn't.
- LeetCode658 Find K Closest Elements
    - **#SearchForClosestElements**, **#SearchForCeiling**, **#SearchForFloor**, **#SearchForArbitraryAnswer**
    - This problem only makes sense when $k < n$. Brute-Force solutions are easy, so omitted.
    - A $O(\log n + k)$ solution with two phases is intuitive. We first find the floor or the ceiling of the input key with binary search, and then use two pointers to find the k closest elements. This is effective in most scenarios, but gets slow when k is large. For example, if k is fixed as a constant factor of n, the complexity will become $O(n)$. Note that if the input key is absent from the array, the floor/ceiling we find with binary key might not be element that is closest to it, which is common pitfall, although the closest element must be one of them.
    - A $O(log(n-k))$ solution also uses binary search, but rather than search for the floor/ceiling of the input key, we search for the answer section directly. Suppose $dis[i] = |x - a[i]|$, the problem essentially asks us to find k entries that have the smallest dis, so how does $dis[i]$ vary as i varies? We first consider the easier problem where there are no duplicate entries in the array.
        - If there are no duplicate entries, the plot resembles a valley. The adjacent entries won't have same value of dis except for one case, where the the input key is missing and its successor and predecessor are present and have the same distance from it. Overall, one or two entries(the aforementioned case) has the smallest value of dis, and to its left $dis[i]$ is **monotonically decreasing**, and to its right $dis[i]$ is **monotonically increasing**.
        - 
        - Deduced from the distribution of $dis[i]$, the k entries comprise a subsequence $[l, h]$ where $max(dis[in]) \leq min(dis[out]), in \in [l, h],\ out \in ([0, n-1] - [l, h])$ and because we break the tie with the smaller entry, $dis[in] = dis[out] \rightarrow a[in] < a[out]$.
        - A property about $max(dis[in])$ is that either $in = l$ or $in = h$, that is either the first or the last entry has the largest distance in the subsequence. Also note that the closest entry to x denoted as $closest = a[m]$, must belong to $[l, h]$($k = 1$ is special, if the predecessor and the successor have the same distance, only the former is picked). To make it more clear, we have $l \leq min(m)$ and $h \geq max(m)$.There are four cases for m:
            - $a[m] = x$
            - $a[m]$ is the predecessor of x
            - $a[m]$ is the successor of x
            - m is in a set of two values {predecessor(x), successor(x)}
        - We have $h - l + 1 = k$, so the first index l determines a unique subsequence $[l, l+k-1]$. Thus we can simplify finding a subsequence into finding the first index of the subsequence, making range of interest for this problem $[0, n-k]$.
        - Although it's not obvious, the distribution of the $dis[i]$ satisfy the premise of binary search. Properties of three sections of search($0 \leq i \leq n-k$) are as follows, with the rigid mathematical proof omitted:
            - Supposing the correct answer is $[ans, ans+k-1]$
            - Lower Section: $dis[i] > dis[i+k]$
                - $dis[i] > dis[i+k]$ says i cannot be ans, but how do we know it belong to lower section or upper section(and not both)? because $dis[j]$ is **monotonically increasing** for $j \geq max(m)$, so we know $i < max(m)$. for $ans \leq j < max(m)$, $j+k$ is outside of $[ans, ans+k-1]$($j+k > ans+k-1$), so $dis[j]$ is impossible to be larger than $dis[j+k]$. How to we know this is always true for $j < ans$? There are two cases: 1. for $j+k \geq ans$, we also have $j+k \leq ans+k-1$, as

$j < ans$, it cannot has a larger distance than $j + k$. 2. for $j + k < ans$, we also have $j + k < min(m)$ because of the **monotonically decreasing** for $[0, min(m) - 1]$, we have $dis[j] > dis[j + k]$. From this analysis we know that this is the property of lower section.
    - Note that $dis[i - 1] > dis[i + k - 1]$ is always true for lower section(except 0), but it cannot be used as its property.
- Answer Section: $dis[i] = dis[i + k] \ \vee (dis[i] < dis[i + k] \wedge (i = 0 \vee dis[i - 1] > dis[i + k - 1]))$
    - The case $dis[i] = dis[i + k]$ happens only when $a[i] < x < a[i + k]$, so $i \leq min(m)$ and $i + k \leq max(m)$, and thus for all $j < i \vee j > i + k$, $dis[j] < dis[o], o \in [i, i + k - 1]$. i is the answer.
    - If $dis[i] \neq dis[i + k]$, we know that $dis[i] > dis[i + k]$ is the property of lower section, so how about $dis[i] < dis[i + k]$? To better understand the problem, suppose that we expand the answer section to $[ans, ans + k - 1]$ not just ans, what is the properties of three sections? Lower sections stays the same, answer section is hard to tell at a glance, but the upper section is easy: $dis[i] > dis[i - k]$, because $dis[i]$ is **monotonically increasing** for $i > l \geq max(m)$, plus $dis[i] = dis[i - k]$ for how the problem breaks the tie. Now we have the property of the answer section: $dis[i] \leq dis[i + k] \wedge dis[i] < dis[i - k]$, which is say that i is not in the lower section nor the upper section. The first index of the subsequence must satisfy this property, but this property is a necessary but not sufficient condition for it, so we cannot use it as the property of the first entry). If we want to locate the lower bound of the subsequence, we need to determine a property that it has but the entries in $[l + 1, h]$ don't. It's actually not hard to see that only the first entry has a predecessor belonging to the lower section, that is to say $dis[i - 1] > dis[i + k - 1]$ is only satisfied by l in $[l, h]$.
    - A good way to describe the answer section: it doesn't belong to the lower section, and it's either the first entry of the array or its predecessor is in the lower section.
- Upper Section: $i > 0 \wedge dis[i - 1] \leq dis[i + k - 1]$(the equality is dues to choosing the smaller entry to break the tie)
    - A good way to describe the upper section: its predecessor doesn't belong to the lower section.
    - It's obvious that Exclusive Template is the better choice and the implementation can be further condensed and simplified: we can replace $|x - a[i]| > |x - a[i + k]|$ with $x - a[i] > a[i + k] - x$. Because we know $a[i] < a[i + k]$(the array is sorted and has no duplicates), thus for all three cases of i these two inequities are equivalent:
        - a[i]---a[i+k]---x, both $|x - a[i]| > |x - a[i + k]|$ and $x - a[i] > a[i + k] - x$ are both TRUE, equivalent.
        - a[i]---x---a[i+k], $|x - a[i]| = x - a[i]$, $|x - a[i + k]| = a[i + k] - x$, equivalent.
        - x---a[i]---a[i+k], both $|x - a[i]| > |x - a[i + k]|$ and $x - a[i] > a[i + k] - x$ are both FALSE, equivalent.

◦ Now we consider about the harder problem where there might be duplicate entries in the array.

- For an array with possible duplicates, some properties of the easier version of the problem is invalidated: the distribution of $dis[i]$ is not **monotonically decreasing** before m, nor **monotonically increasing** after m; any adjacent entires could have equal values of dis and etc. We start by looking at the graph of $dis[i]$.
- 
- It's still a valley but with potentially many equal adjacent entries, and it may occur anywhere in th array, not just between the predecessor and the successor. The number of entries with smallest $dis[i]$ could be any number in $[1, n]$, which is also the set of values of m:
    - $a[m] = x$
    - $a[m]$ is the set of all occurrence of the predecessor of x
    - $a[m]$ is the set of all occurrence of the successor of x
    - m is in a set of values consisting of all occurrence of the successor and the predecessor of x
- For $i \in [0, max(m)]$, $dis[i]$ is **non-increasing**; for $i \in [min(m), n - 1]$, $dis[i]$ is **non-decreasing**. Perhaps a more useful property is that for $a[i] < x$, $dis[i]$ is **non-increasing**; for $a[i] > x$, $dis[i]$ is **non-decreasing**.
- Notice something troublesome: as a result of duplicate entries, the index interval $[l, h]$ of the answer is not unique, which means multiple answer sections is possible. If we were to use binary search, the process is similar to searching for arbitrary answer, but unlike LC162, it always returns $[l, h]$ with the largest l and h when there are multiple answers. For a concrete example : [1(0),1(1),2(2),2(3),2(4),2(5),2(6),4(7),4(8)] with k=3 x=3, [2,4], [3,5], [4,6] are all correct answers, but binary search can only find [4,6].
- We can still search for the first index l of the subsequence directly in $[0, n - k]$, but with a slightly different properties of three sections. Because of the presence of duplicate entries, we need to treat equality carefully. For example $dis[i] = dis[i + k]$ can happen for three cases:
    - $a[i] = a[i + k] < x$
    - $a[i] < x < a[i + k]$
    - $x < a[i] = a[i + k]$
- As with the case where duplicates are not allowed, we are reasonable to believe the property of lower section is the simplest, so we consider it first.(The relation between ans and $max(m)$ and $min(m)$ is nondeterministic):
    - If $dis[i] > dis[i + k]$, we know that i is not ans, and because $dis[j]$ is **non-decreasing** for $j \geq min(m)$, we know $i < min(m)$. For $ans \leq j \leq max(m)$, $j + k$ is outside of $[ans, ans + k - 1]$($j + k \geq ans + k > ans + k - 1$), so $dis[j]$ is

impossible to be larger than $dis[j+k]$. Thus we have $i < ans$. Notice that $dis[i] > dis[i+k] \to i < ans$ but $i < ans \to dis[i] > dis[i+k]$ is not true.
    - For $i < ans$, $dis[i] < dis[i+k]$ is impossible but what about $dis[i] = dis[i+k]$? For $i < ans$ we have $a[i] \le x$, and the dis is **non-increasing**, so we have $dis[i] = dis[i+1] = \ldots = dis[i+k-1] = dis[i+k]$, because we aims to find the largest $[l, h]$, $[i+1, i+k]$ is better than $[i, i+k-1]$, we conclude that i belongs to lower section. Actually if $a[i] = x \wedge dis[i] = dis[i+k]$, i is a correct answer but not necessarily the largest one. Also note that we need $a[i] = a[i+k]$, because if $a[i] \ne a[i+k] \wedge dis[i] = dis[i+k]$, i is the answer section, not in the lower section.
    - The above analysis gives us the property of lower section: $dis[i] > dis[i+k] \vee (dis[i] = dis[i+k] \wedge a[i] = a[i+k] \wedge a[i] \le x)$. We denote this boolean expression as a predicate $L(i)$.
    - $L'(i) = dis[i] > dis[i+k] \vee (dis[i] = dis[i+k] \wedge a[i] = a[i+k] \wedge a[i] < x)$ is also ok, but the ans is not guaranteed to be the largest, it might be random.
- Can we write the property of answer section and upper section explicitly? Yes but it will be pretty complicated, so we use $L(i)$ to represent them:
    - Answer section: $L(i) = F \ \wedge \ (i = 0 \ \vee \ L(i-1) = T)$
    - Upper section: $i > 0 \ \wedge \ L(i-1) = F$
- We still compare $dis[i]$ with $dis[i+k]$, $dis[i-1]$ with $dis[i+k-1]$, but meanwhile takes the relation between $a[i]$ and x into consideration because $dis[i] = dis[j]$ has more cases than before.
    - Lower Section: $dis[i] > dis[i+k] \vee (dis[i] = dis[i+k] \ \wedge \ a[i] < x)$
    - Answer Section: $(dis[i] < dis[i+k] \wedge (i = 0 \vee dis[i-1] > dis[i+k-1])) \vee (dis[i] = dis[i+k] \wedge a[i] = x)$
    - Upper Section: $(i > 0 \wedge dis[i-1] \le dis[i+k-1]) \vee (dis[i] = dis[i+k] \wedge \ a[i] > x)$(the equality is dues to choosing the smaller entry to break the tie)
    - Is $L(i)$ or $L'(i)$ equivalent to $P(i) = x - a[i] > a[i+k] - x$?
        - $a[i] = a[i+k]$
            - $a[i] < x$, $P(i) = T$, $L(i) = T$, $L'(i) = T$
            - $a[i] = x$, $P(i) = F$, $L(i) = T$, $L'(i) = F$
            - $a[i] > x$, $P(i) = F$, $L(i) = F$, $L'(i) = F$
        - $a[i] \ne a[i+k]$
            - $a[i] < a[i+k] \le x$, $P(i) = T$, $L(i) = T$, $L'(i) = T$
            - $a[i] \le x \le a[i+k]$, $P(i) = L(i) = L'(i)$
            - $x \le a[i] < a[i+k]$, $P(i) = F$, $L(i) = F$, $L'(i) = F$
  - An example of complicated categorized discussion if we want to simplify the codes of the most efficient approach, and the result is remarkably simple.
  - One of **a[first]** or **a[last]** has the largest distance from key, any element outside the interval [first, last] should has a larger distance.
  - Turning searching a range of fixed size into searching the first index of the range.
  - https://leetcode-cn.com/problems/find-k-closest-elements/solution/pai-chu-fa-shuang-zhi-zhen-er-fen-fa-python-dai-ma/
  - https://leetcode.com/problems/find-k-closest-elements/discuss/106426/JavaC%2B%2BPython-Binary-Search-O(log(N-K)-%2B-K)

HotFixes

There are many varaints of binary search, I noted down some of their properties and come back later to summarize:
How many items are smaller than a given item? Which items fall within a given range?

5. **Combination of Answers**: For some problems, the desired answer is a combination of different situations. For example, the answer to **LC35** has differnt property under different circumstances, it the input key is in the array, we return its index; if the input key we return its desired insert position, which can be further cut down into two cases: if it has a successor, we return the index of that successor; if it doesn't have one, we return the size of the array. These are different search objectives so we may need to use three different procedures to handle each of them! But luckily with a little modification of the Exclusive Procedure Method1, we can solve it using a unified procedure. But note that this may be always be the case, as sometimes we can't merge the code of different procedures finely, in which case we need to write them separately.
6. **Disjunction of Multiple Properties**: The second solution of **LC33** is a good example of dealing with multiple properties : If an entry is present in the rotated array(excluding the first entry, the pivot), it falls in either the first half subsequence or the second half subsequence but not both. Although the entries of the two subsequences are disjoint, we cannot only use the relation between the entry pointed by **mid** and the **key** to determine how to update **low** and **high**, because from the perspective of the target **key**, the subsequence comes before it and after it doesn't have exclusive properties with regard to **key**. For example, if the **key** is in the first half, and its index is **i**. It's possible that entries comes before it and the entries of the second half are both smaller than **key**, so we cannot move **low** or **high** when **mid** points to a entry smaller than **key**. The same is true for entries larger than **key** when **key** belongs to the second half. To make the properties exclusive we need to use their properites with regard to **pivot**, and the case when **key** is in the first half have different properties from when it is in the second half so we need two different implementation. We use the case when **key** falls in the range of first half(but not necessarily present in the array) to illustrate how ot get exclusive properties : The **lower section** satisfy the condition $a[i] < key \wedge a[i] > pivot$; The **upper section** satisfy the condition $a[i] > key \vee a[i] < pivot$. This two properties are excluisve. Alternatively, we can see this from another perspective: All entires with $a[i] > key$ belong to the upper section. For entries with $a[i] < key$, those who $a[i] > pivot$ belong to the **lower section** and those who $a[i] < pivot$ belong to the **upper section**. Thus the property of **upper section** is actually a disjunction of two exclusive properties.

7. **The Commonly Seen Marginal Cases**: The commonly seen marginal cases of binary search is that something(usually an index) you want to find and use later to solve the problem is not present in the array, for example the predecessor, successor or the entries that satisfy certain properties. This usually happens when we partition the search into multiple stages, that is our solution has more than one phase. If we don't consider this cases, the end result is potentially buggy. The first solution of **LC33** is a good example, where finding the largest or the smallest entry is the first step. From what I have seen, there are usually two ways to solve this:
    i. Explicitly checking whether that something we try to find is in the array or is not in the array and only when it's in the array and we have found it then we use it, otherwise we use the shortcut to solve the problem(usually a few lines of codes)
    ii. Set the initial value of the variable that is to hold that something we are going to find to a reasonable value that will still work even it's absent from the array. This correct reasonable value depends on how you are going to use that value later, and if that something is absent, that variable won't be updated.
8. **General Cases and Marginal Cases**: After some practice, I notice that general cases and marginal cases are relative concepts. In **LC33**, **LC34** and **LC81**, finding the successor and the predecessor is relative tedious and error-prone compared to better solutions but they nicely illustrate commonly seen marginal cases as discussed in the above section. For these problems, marginal cases usually mean those cases where we cannot find what we set out to find, the successor or the predecessor. For these cases we usually need to write more codes or change the initial values for some variables and we call the cases where they are present general cases. **General cases are cases we need to write most of our codes to tackle with, while the marignal cases are simpler cases that can often be dealt with a few lines of codes**. They are relative to the specific approach we choose to solve the problem. When coding an algorithmic implementation, I find several ways to tackle these two different kinds of cases:
    i. Write codes for each cases individually, and then try to unified them into one implementation
    ii. Write codes for the general cases and simultaneously adding codes that tackles marginal cases
    iii. Write complete codes for the general cases first and then try to modify it to accommodate mariginal cases
    iv. The last two cases will typically eventually end up with same codes but go through different thinking process. I personally adore the last one because it helps me to focus on the main procedure. If we use the last method, we need to be very clear about the preconditions we make for the general cases(writing them down in the comments!) and make adjustments for marginal cases later.
9. **Categorized Discussion**: A single problem may be categorized into different situations/cases that require somehow different approaches to solve. We can solve the whole problem with codes for each case but we can also try to unified the different codes into one unified version. There are pros and cons for doing so. The merit is a simpler and sometimes just a few lines of codes, but the demerit is that it may be hard to read as we mix up codes of different cases. This is the extension of the discussion of general and marginal cases, as somtimes there may be multiple general and marginal cases that require a lot of codes. **LC658** is a good example.
10. **Trick without Mathematics** : **LC35** is a typical problem that can use a trick to quickly find the correct answer. We know when the key is absent from the array, after the Inclusive Approach exits the loop, low or high will have relation to it, and very likely the relation is a fixed equation. How to derive the equation? At least two ways, one is like what I have done in this document, that is pure mathematical analysis; another is to use a simple example to see how it works out! There are risks involved in the second approach as the cases we choose may not cover all the potential cases. But in practice, this usually works!
11. binary search can also be used to search for the answer section and many related properties of , such as the index of its first and last entries.

How many keys fall within a given range (between two given keys)? Which keys fall in a given range
The basic operations for determining where a new key fits in the order are the rank operation (find the number of keys less than a given key) and the select operation (find the key with a given rank)

Remark

Number of Comparison: It is interesting to study for a sorted list of size N, how many times of comparison is needed for a particular key.

The difference between examining the last entry and having an empty subsequence

The famous bug of calculating the index of midpoint of a subsequence in binary search : https://ai.googleblog.com/2006/06/extra-extra-read-all-about-it-nearly.html

From Wikipedia

https://en.wikipedia.org/wiki/Binary_search_algorithm

> Binary search compares the target value to the middle element of the array. If they are not equal, the half in which the target cannot lie is eliminated and the search continues on the remaining half, again taking the middle element to compare to the target value, and repeating this until the target value is found. If the search ends with the remaining half being empty, the target is not in the array.
>
> Binary search is faster than linear search except for small arrays. However, the array must be sorted first to be able to apply binary search. There are specialized data structures designed for fast searching, such as hash tables, that can be searched more efficiently than binary search. However, binary search can be used to solve a wider range of problems, such as finding the next-smallest or next-largest element in the array relative to the target even if it is absent from the array.
>
> There are numerous variations of binary search. In particular, fractional cascading speeds up binary searches for the same value in multiple arrays. Fractional cascading efficiently solves a number of search problems in computational geometry and in numerous other fields. Exponen

> tial search extends binary search to unbounded lists. The binary search tree and B-tree data structures are based on binary search.
>
> An infinite loop may occur if the exit conditions for the loop are not defined correctly. Once **L** exceeds **R**, the search has failed and must convey the failure of the search. In addition, the loop must be exited when the target element is found, or in the case of an implementation where this check is moved to the end, checks for whether the search was successful or failed at the end must be in place. Bentley found that most of the programmers who incorrectly implemented binary search made an error in defining the exit conditions.

1. **Duplicate Elements**: Common implementation deals with lists that don't have duplicate entries. For these lists, returning the first and last occurrence(indexes) will be useful.
    i. Range Search
    ii. Any match
    iii. Leftmost element
    iv. Rightmost element
2. **Searching Insert Position**: Returning the insert index when the queried key is absent from a sorted input list is useful. Note that although this approach could utilize binary search, it doesn't answer the caller whether the key is present in the list.
3. Searching for the Predecessor or the Successor, nearest neighbor
4. Rank Queries, seeking the number of elements between two values

if we are looking for the first entry of some subsequence of common property, the property of the lower section is simpler; if we are looking for the last entry, the upper section has simpler property

the method of classification discussion

categorized discussion method

排除法，减治法： searching by exclusion

K-way search, more general than binary search?