Binary Search

The `binary search` algorithm is one of the most well known searching algorithms. It is a fast algorithm, running in `O(log n)` time, that has seen many uses across the field of Computer Science. However, there is one major drawback with the algorithm, the input **must be sorted**. The efficiency of the algorithm comes from the sorted nature of the input. It allows the search space of the algorithm to be reduced significantly each iteration.

Implementation

Pseudocode

```
binary_search(A, n, T):
    L := 0
    R := n - 1
    while L ≤ R do
        m := floor((L + R) / 2)
        if A[m] < T then
            L := m + 1
        else if A[m] > T then
            R := m - 1
        else:
            return m
    return unsuccessful
```

Java Implementation

Based on the above pseudocode we can start to implement the binary search algorithm in Java. We'll start off with the method signature since we would like to be able to perform a binary search on any type of array or list input. Note that the return type of the method should be the index, i.e. an int, of the target element.

```java
public static <T extends Comparable<? super T>> int binary_search(final T[] data,
final T elem, final Comparator<T> comparator) {}
public static <T extends Comparable<? super T>> int binary_search(final List<T> data,
final T elem, final Comparator<T> comparator) {}
```

We will need to decide what value to return if we don't find the target element. This varies across implementations /languages so best to always check the documentation. For example, in Java, the built-in binary search algorithm returns the negation of index that corresponds to where the value would be in the array. For simplicity, we will return `-1`. This indicates that the value is not in the input.

```
public static <T extends Comparable<? super T>> int binary_search(final T[] data,
final T elem, final Comparator<T> comparator) {
    int left = 0;
    int right =  data.length - 1;
    while (left <= right)  {
        int m = (left + right) / 2;
        if (comparator.compare(data[m], elem)  < 0) {
            left = m + 1;
        } else if (comparator.compare(data[m], elem) > 0) {
            right = m - 1;
        } else {
            // we found it
            return m;
        }
    }
    return -1;
}
```

That's the full algorithm, it short and simple but very powerful.

We can then test our code using the following:

```
public static void main(final String[] args) {
    final List<Integer> data = IntStream.range(0,
100).boxed().collect(Collectors.toList());
    final Integer[] arr_data = data.toArray(new Integer[0]);
    final boolean contains_20 = binary_search(arr_data, 20,
Comparator.comparing(Integer::intValue)) != -1;
    final boolean contains_1000 = binary_search(arr_data, 1000,
Comparator.comparing(Integer::intValue)) != -1;
    System.out.println("Is 20 contained in the list: " + contains_20);
    System.out.println("Is 1000 contained in the list: " + contains_1000);
}
```

With an expected output of:

```
Is 20 contained in the list: true
Is 1000 contained in the list: false
```

Complexity

The time complexity of the `binary search` algorithm is `O(log n)` and comes from the fact that in each iteration of the loop, we assign the mid index to either the left or right index, thus cutting our search space in half each iteration.