# 10.2 Binary Search

Suppose that you are, once again, looking for a CD in your collection. If you have gone to the trouble of putting the CD's in order, your search can be speeded up considerably and you should be able to locate the one that you want very quickly. Our next algorithm, called a *binary search*, provides an efficient method for searching a list in which the items are ordered. The algorithm is an example of a large class called *divide and conquer* algorithms. These algorithms employ strategies that solve the problem by quickly reducing its size. For a binary search, at each stage of the solution, we cut the size of the problem roughly in half. To illustrate the way that the algorithm works, suppose that we are searching for the item 47 in the sorted list shown below.

```
16 19 22 24 27 29 37 40 43 44 47 52 56 60 64 71
```

To start the process, we initially examine the item in the middle of the array. The middle item, 40, is not the one that we want but, because it is less than the value that we are looking for and because the list is sorted, we know that we can eliminate all the items in the lower half of the list. Our search can now continue looking only at the remaining half of the original list, as shown.

```
43 44 47 52 56 60 64 71
```

We now repeat our strategy on these items. Since there are now eight items left, there is no exact middle; we can choose either 52 or 56. If we look at 52, we see that it is larger than the value for which we are searching. Again taking advantage of the fact that the items are ordered, we can eliminate 52 and all items above it from our search. This leaves us with only the following values.

```
43 44 47
```

We again look at the middle item, this time finding 44, which is too small. We can therefore discard both it and the item below it, leaving us with only one value.

```
47
```

One final probe, looking at 47, produces a successful search. In only four comparisons, we have been able to find the item that we want. Let us now see how we can implement a binary search in Java. Suppose that we are searching for `item` in an array called `list`. At the start of the search, `item` could be anywhere in `list` but, as the search proceeds, the interval that is being searched is reduced over and over again. To keep track of the bounds of this interval, we use two `int` variables: `bottom`, containing the index of the lower bound of the current interval and `top`, containing the index of the upper bound. The easiest way to find the index of the middle (or near middle) of the interval is to take the average of `bottom` and `top`. Once we have found the middle, we examine the value located there. If it is equal to `item`, the value we are seeking, we are done. If it is smaller than `item`, we know that we can discard both the value at `middle` and all values below this. To do so, we simply change the value of `bottom` to `middle + 1`. Similarly, if the value located at `middle` is larger than item we discard the upper values of the interval by setting `top` to `middle - 1`.

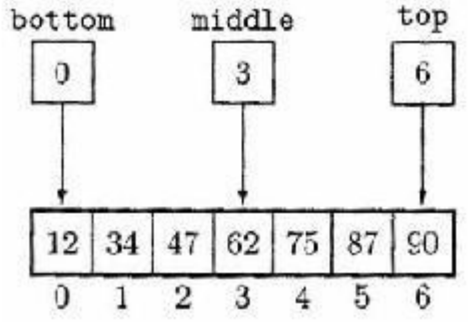
## Example 1

Suppose that we want to perform a binary search for the value 75 on the following data.

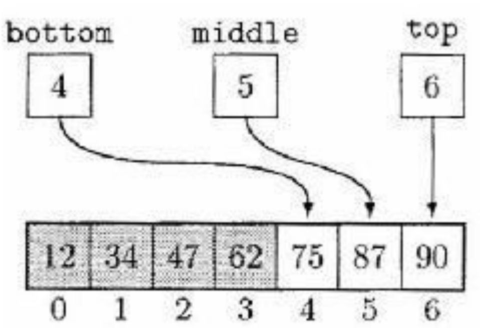
12 34 47 62 75 87 90

Initially, we want to examine the entire array so the variables `bottom` and `top` are initialized to

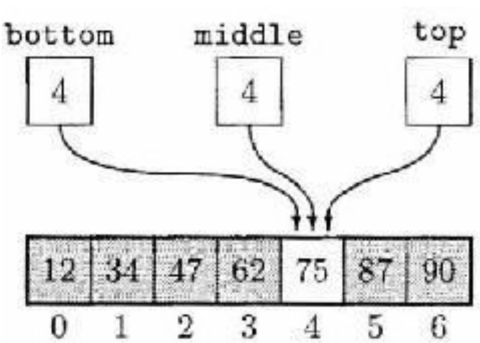
0 and 6, the indices of the first and last elements, respectively while `middle` is set to `(0 + 6) /2 = 3`. (In the diagrams, we use arrows to emphasize the purpose of `bottom`, `middle`, and `top`; they should not be confused with the arrows that we use for reference variables.)




Since 62 < 75, the item that we are seeking cannot be in the left half of the list. We discard this half of the list by setting `bottom` to `middle + 1`.
The middle of the remaining interval is `(4 + 6) /2 = 5`, as shown in the next diagram.




Since 87 > 75, the value 75 cannot be in the upper half of this sublist so we discard it by setting `top` to `middle - 1`. Since `bottom` and `top` are now both equal to 4, the value of `middle` will be `(4 + 4) /2 = 4`.




Once `middle` has found the value, the search ends successfully.

The next example implements a binary search for the value `item` in an array `list` of `double` values. It returns the index in `list` of `item` (if `item` is in `list`) and `-1` otherwise. Notice how the search terminates if the item that we are looking for is *not* in the list. We saw in the last example how the values of `bottom` and `top` converge as the search proceeds with `bottom` getting larger and `top` getting smaller. If the item that we are seeking is not in the list, eventually we will either set `bottom` to a value larger than `top` or we will set `top` to a value smaller than `bottom`. Thus, we only keep searching as long as `bottom <= top` and the item that we are seeking has not been found.

# Example 2

```
public static int binSearch (double[] list, double item)
{
   int bottom = 0;                      // lower bound of subarray
   int top = list.length - 1;             // upper bound of subarray
   int middle;                            // middle of subarray
   boolean found = false;                // to stop if item found
   int location = -1;                   // index of item in array

   while (bottom <= top && !found)
   {
      middle = (bottom + top)/2;
      if (list[middle] == item)     // success
      {
         found = true;
         location = middle;
      }
      else if (list[middle] < item) // not in bottom half
         bottom = middle + 1;
      else                                // item cannot be in top
half
         top = middle - 1;
   }
   return location;
}
```

# Exercise 10.2

1. Suppose that an array contains the following elements.

   23 27 30 34 41 49 51 55 57 60 67 72 78 83 96

   Trace the execution of the method `binSearch` shown in this section as it searches for

the following values of `item`. In each trace, show the progress of the search by using diagrams similar to those in Example 1.
(a) 72 (b) 41 (c) 62

```
public class Question1 {

	public static void main(String[] args) {
		// TODO Auto-generated method stub
		int [] list =
{23,27,30,34,41,49,51,55,57,60,67,72,78,83,96};
		System.out.println(binSearch(list, 72));
		System.out.println(binSearch(list,41));
		System.out.println(binSearch(list,62));

	}

	public static int binSearch (int [] list, int item) {
		int bottom = 0;
		int medium;
		int top = list.length-1;
		int location = -1;
		boolean found = false;

		while (!found) {
			medium = (top + bottom)/2;
			if (item == list[medium]) {
				found = true;
				location = medium;
			}
			else if (item > list[medium]) {
				bottom = medium + 1;
			}
			else if (item < list[medium]) {
				top = medium - 1;
			}
		}

		return location;
	}

}
```

2. What changes would have to be made to `binSearch` so that it will search an array that is sorted in *descending* order.

The < and > sings will be reversed in the method. Since the bigger values lies toward the lower bound and smaller values lies towards the upper bound, we can then switch the method such that

- When the item is bigger than the medium, we can discard both the middle value and the values above it.
- When item is smaller than the medium, we can discard both the middle and the values below it.

```
public class Question2 {

	public static void main(String[] args) {
		// TODO Auto-generated method stub
		int [] list =
{96,83,78,72,67,60,57,55,51,49,41,34,30,27,23};
		System.out.println(binSearch(list, 72));
		System.out.println(binSearch(list,41));
		System.out.println(binSearch(list,62));


	}
	public static int binSearch (int [] list, int item) {
		int bottom = 0;
		int medium;
		int top = list.length-1;
		int location = -1;
		boolean found = false;

		while (!found) {
			medium = (top + bottom)/2;
			if (item == list[medium]) {
				found = true;
				location = medium;
			}
			else if (item < list[medium]) {
				bottom = medium + 1;
			}
			else if (item > list[medium]) {
				top = medium - 1;
			}
		}

		return location;
	}

}
```

3. Rewrite binSearch so that, if a search is unsuccessful, the method will return the index of the value *nearest* to item, instead of returning -1. (If there is a tie, return the smaller index.)

```java
public class Question3 {

	public static void main(String[] args) {
		// TODO Auto-generated method stub
		int [] list =
{96,83,78,72,67,60,57,55,51,49,41,34,30,27,23};
		System.out.println(binSearch(list, 72));
		System.out.println(binSearch(list,41));
		System.out.println(binSearch(list,62));

	}

	public static int binSearch (int [] list, int item) {
		int bottom = 0;
		int medium;
		int top = list.length-1;
		int location = -1;
		boolean found = false;

		while (!found && bottom <= top) {
			medium = (top + bottom)/2;
			if (item == list[medium]) {
				found = true;
				location = medium;
			}
			else if (item < list[medium]) {
				bottom = medium + 1;
			}
			else if (item > list[medium]) {
				top = medium - 1;
			}
		}
		int difference = list[0];
		 if (location == -1) {
			 for (int x = 0; x < list.length; x++) {
				int temp = Math.abs(list[x] - item);
				if (difference > temp) {
					difference = temp;
					location = x;
				}
			 }
		 }
		return location;

	}
}
```

4. What is the maximum number of comparisons that might be necessary to perform a binary search on a `list` containing seven items?

It will maximum number of 3 comparisons to perform a binary search on a list containing 7 items.

```
public class Question4 {

	public static void main(String[] args) {
		// TODO Auto-generated method stub
		int items = 7;
		Comparisons(items);



	}

	public static void Comparisons (int items) {
		int comparisons = 0;
		int temp = items;
		while (temp >= 1) {
			temp = temp/2;
			comparisons ++;
		}
		System.out.println("It will maximum number of " + 
comparisons + " comparisons to perform a binary search on a list 
containing " + items + " items");
	}
}
```

5. Repeat the previous question for lists with the indicated sizes.
(a) 3 (b) 15 (c) 31 (d) 63 (e) 100 (f) 500 (g) 1 000 (h) 10 000

```
public class Question5 {

	public static void main(String[] args) {
		// TODO Auto-generated method stub
		//PART A
		Comparisons(3);

		//PART B
		Comparisons(15);

		//PART C
		Comparisons(31);

		//PART D
		Comparisons (63);

		//PART E
		Comparisons(100);
```

```java
		//PART F
		Comparisons(500);

		//PART G
		Comparisons (1000);

		//PART H
		Comparisons (10000);
	}

	public static void Comparisons (int items) {
		int comparisons = 0;
		int temp = items;
		while (temp >= 1) {
			temp = temp/2;
			comparisons ++;
		}
		System.out.println("It will maximum number of " + 
comparisons + " comparisons to perform a binary search on a list 
containing " + items + " items");
	}

}
```