Searching & Sorting in Java – Sequential Search

A sequential search is a systematic technique where you begin your search at the beginning (of the array), and stop your search once you reach the desired value.

For example, the following code performs a sequential search on an array of `int` values, looking for the value matching `item`. If the search is successful, it returns the location of `item` in the array `list`. If the search fails, the method returns `-1`.

```
public static int seqSearch ( int[] list, int item)
{
  int location = -1;
  for (int i = 0; i < list.length; i++)
  {
    if (list[i] == item)
      location = i;
  }
  return location;
}
```

Note that the variable `location` is initialized to reflect an unsuccessful search. If item is not found, the method will return this default value to indicate a failed search.

Although this method works, it is not as efficient as we could make it. Even if it finds the search value, it continues through the entire array. It would be more efficient to stop the search once we had found the search value. We can accomplish this improvement through the use of a `boolean` variable `found`.

```
public static int seqSearch ( int[] list, int item)
{
  int location = -1;
  boolean found = false;
  for (int i = 0; i < list.length && !found; i++)
  {
    if (list[i] == item)
    {
      location = i;
      found = true;
    }
  }
  return location;
}
```

Searching & Sorting in Java – Binary Search

If the data you wish to search is already in order, a sequential search will still, on average, take the same amount of time to find the item you want. It is possible, however, to greatly improve the speed of a search on sorted data.

The *binary search* algorithm is one such way to improve performance using sorted data. This algorithm is an example of a *divide and conquer* algorithm, of which there are many other examples. This type of algorithm solves a problem by quickly reducing its size. For the binary search, at each stage of the problem we cut the size of the problem roughly in half.

To illustrate, consider the following list, and suppose we are searching for the value 47.

16	19	22	24	27	29	37	40	43	44	47	52	56	60	64
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14

To start the process, we initially examine the item in the middle of the array. The middle item, 40, is not the one we want, but it is less than the value we are looking for. Since the list is sorted, we use this information to eliminate all of the items in the lower half of the list. Our search now only looks at the remaining (upper) half of the list.

16	19	22	24	27	29	37	40	43	44	47	52	56	60	64
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14

We repeat our strategy on these items. The middle value is now 52, which is too high, so we eliminate the upper half of the remaining list.

16	19	22	24	27	29	37	40	43	44	47	52	56	60	64
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14

The middle value is now 44, which is too small. Eliminating everything below this value leaves us with only a single item that hasn't been eliminated, which is the location of our target value.

16	19	22	24	27	29	37	40	43	44	47	52	56	60	64
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14

To implement this algorithm in Java, we will search for `item` in an array called `list`. Through the process of elimination, the upper and lower bounds of the array that we need to search will change, so we will track them with `int` variables called `top` and `bottom`. Similarly, we need to track the `middle` value, also an `int`.

For each iteration, we can find the value of middle by taking the average of the top and bottom. If the value at middle is equal to item, then obviously our search is done. If our value is too low, the bottom becomes middle + 1. If our value is too high, the top becomes middle - 1.

If our search value is not in the list, this process will continue until bottom and top and middle are all equal to each other (i.e., we are looking at a single element of the array). On the next step, top or bottom will change such that top < bottom, which signals the end of our search, at which point we return a value to indicate a failure (-1).

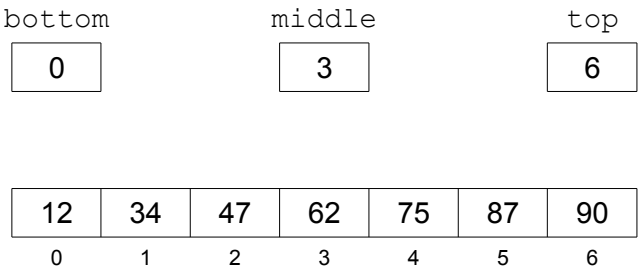
Searching & Sorting in Java – Binary Search

```
public static int binSearch ( double[] list, double item)
{
  int bottom = 0;                // lower bound of searching
  int top = list.length - 1;     // upper bound of searching
  int middle;                    // current search candidate
  boolean found = false;
  int location = -1;             // location of item, -1 for failure

  while (bottom <= top && !found)
  {
    middle = (bottom + top)/2;  // integer division, auto-truncate
    if (list[middle] == item)
    {
      location = middle;        // success!
      found = true;
    }
    else if (list[middle] < item)
    {
      bottom = middle + 1;      // look only in top half
    }
    else
    {
      top = middle - 1;         // look only in bottom half
    }
  }
  return location;
}
```

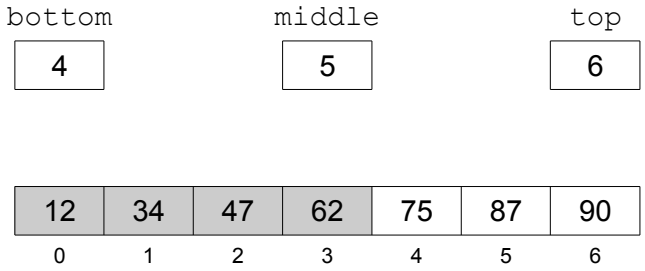
Suppose we want to perform a binary search for the value 75 on the following data.

12 34 47 62 75 87 90

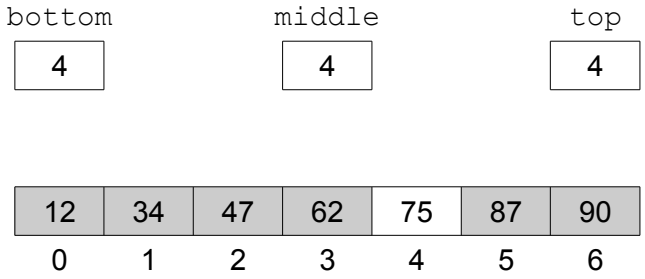
Initially, we need to search the entire array, so bottom and top are set to 0 and 6, while middle is set to 3.



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

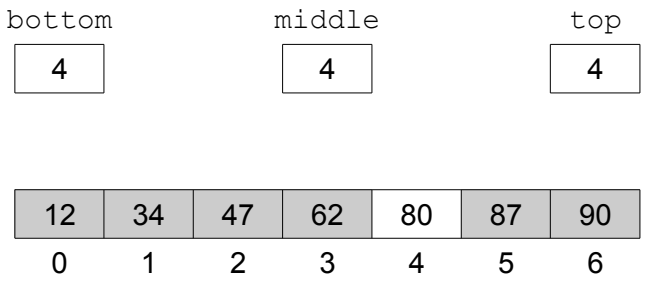


Since `87 > 75`, the value 75 cannot be in the upper half of the sublist, so we discard it by setting `top` to `middle - 1 = 4`. The new value of middle will be `(4 + 4)/2 = 4`.



Once the value has been found at middle, the search ends successfully.

Now let us consider a failed search, were the final element was not equal to our search value.



Since `80 > 75`, the value 75 cannot be in the upper half of the sublist, so we discard it by setting `top` to `middle - 1 = 3`. Now we have the situation where `top < bottom`, so our searching ends without a successful result.

Searching & Sorting in Java – Binary Search

Note: The Java libraries include methods for sorting arrays of any primitive type (int, long, float, double, char), or even objects (e.g., String). These methods are *overloaded*, which means they can be called using the same method name, `sort`. Since they are part of the `Arrays` class, the call will be:

```
Arrays.sort(<array name>);
```

For example, consider the following arrays (integers and strings) which are sorted using `sort`.

```
int[] numbers = {4, 3, 5, 6, 7, 4, 8, 3, 4, 1};
String[] names = {"Ed", "Bob", "Alice", "Rob", "Gayle"};
Array.sort(numbers);
Array.sort(names);
```

In order to use methods from the `Arrays` class, we must *import* the library into our current program. A full program, including the required import statement, is shown below. Notice that the import must come before the class declaration.

Also included is a method, toString, which allows the array to be easily displayed on a single line (if it is short enough).

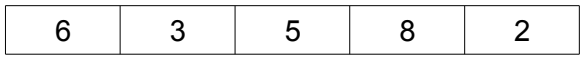
```
import java.util.Arrays;

public class ArraySortJavaLib {

    public static void main(String[] args)
    {
        //... 1. Sort strings - or any other Comparable objects.
        String[] names = {"Zoe", "Alison", "David"};
        Arrays.sort(names);
        System.out.println(Arrays.toString(names));

        //... 2. Sort doubles or other primitives.
        double[] lengths = {120.0, 0.5, 0.0, 999.0, 77.3};
        Arrays.sort(lengths);
        System.out.println(Arrays.toString(lengths));
    }
}
```

Searching & Sorting in Java – Insertion Sort

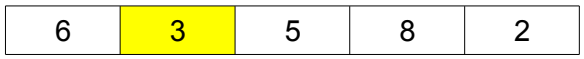
Insertion Sort

The insertion sort algorithm focuses on a single element of the list and tries to find the correct location for that element. When searching for the correct position, only positions below the current position are considered. Thus, on the first pass, the first element is always in the correct position (so it is redundant to perform this step).

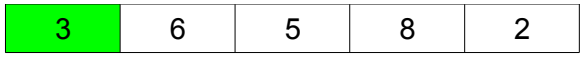
As is often the case, this is best illustrated through an example.



Pass #1:

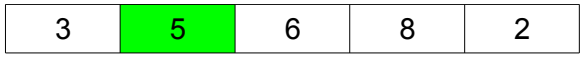


- compare 3 to all values below it (to the left); if they are larger, shift them right
- 6 is larger than 3, so move 6 to position 1
- put 3 in the empty location (which is 0).



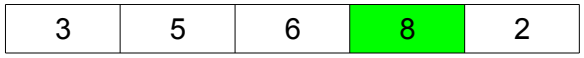
Pass #2:

- compare all values to 5
- 5 is less than 6, so shift 6 right to position 2
- 5 is greater than 3, so stop; put 5 in position 1

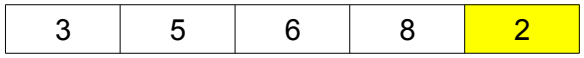


Pass #3:

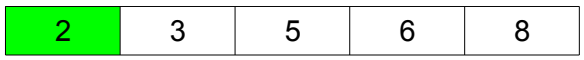
- no change; 8 is already in the correct location



Pass #4:



- compare all values to 2
- 2 is less than each value below it, so they will be shifted right by one after each comparison
- when the bottom of the list is reached, the pass ends, and 2 is put at the available location at the beginning



Searching & Sorting in Java – Insertion Sort

To implement this algorithm for an array of values, we must examine and correctly insert each of the values in the array from the second position up to the last one.

```
for (int top = 1; top < list.length; top++)
      // insert element found at top into its correct position
      // among the elements from 0 to top - 1
```

For each pass, we need to copy the element under consideration into a temporary location. Now, as we move from right to left through the array, we shift any larger values to the right. Once we have moved any larger values, we put our target element into the last empty location.

```
public static void insertSort( double[] list)
{
      for (int top = 1; top < list.length; top++)
      {
            double item = list[top];   // temporary storage of item
            int i = top;

            while (i > 0 && item < list[i-1])
            {
                  // shift larger items to the right by one
                  list[i] = list[i-1];
                  // prepare to check the next item to the left
                  i--;
            }
            list[i] = item;       // put sorted item in open location
      }
}
```

Exercises

1. An insertion sort is to be used to put the values 6 2 8 3 1 7 4 in ascending order (lowest to highest). Show the values as they would appear after each pass of the sort.

2. What changes would have to be made to the insertSort method in order to sort the values in descending order?

3. What might happen if the while statement's first line were written in the following form?
   ```
   while (item < list[i-1] && i > 0)
   ```

4. Write a program that initializes an array with the names of the planets in their typical order (from closest to furthest from the Sun) and prints them in that order on one line. The program should then use an insertion sort algorithm to arrange the names alphabetically. To trace the progress of the sort, have it print the list after each pass.

5. The *median* of an ordered list of numerical values is defined in the following way: If the number of values is odd, the median is the middle value. If the number of values is even, the median is the average of the two middle values. Write a program that prompts the user for the number of items to be processed, reads that many real values, and then finds their median.

6. A sort is said to be *stable* if it always leaves values that are considered to be equal in the same order after the sort. Is the insertion sort stable? Justify your answer.

Searching & Sorting in Java – Bubble Sort

With the bubble sort, the basic idea is to compare adjacent values and exchange them if they are not in order. Consider the following example which shows the first pass through the algorithm.

1. Compare 6 and 3	6	3	5	8	2
2. 6 is higher, so swap 6 and 3	3	6	5	8	2
3. Compare 6 and 5	3	6	5	8	2
4. 6 is higher, so swap 6 and 5	3	5	6	8	2
5. Compare 6 and 8	3	5	6	8	2
6. 8 is higher, so no swap is performed	3	5	6	8	2
7. Compare 8 and 2	3	5	6	8	2
8. 8 is higher, so swap 8 and 2	3	5	6	2	8
9. The largest value, 8, is at the end of the array	3	5	6	2	8

The result of this comparing and exchanging is that , after one pass, the largest value will be at the upper end of the list. Like a bubble, it has risen to the top. On our next pass, we can ignore this position and work with a shortened list, allowing the largest remaining value to rise to the (slightly lower) top.

The complete bubble sort is shown below in a more condensed format.

First Pass	6	3	5	8	2
	3	6	5	8	2
	3	5	6	8	2
	3	5	6	8	2
	3	5	6	2	8
Second Pass	3	5	6	2	8
	3	5	6	2	8
	3	5	6	2	8
	3	5	2	6	8
Third Pass	3	5	2	6	8
	3	5	2	6	8
	3	2	5	6	8
Fourth Pass	3	2	5	6	8
	2	3	5	6	8

Searching & Sorting in Java – Bubble Sort

To code this algorithm in Java, we start with a loop very similar to that used for the selection sort.

```
for (int top = list.length – 1; top > 0; top--)
      // compare adjacent values of the unsorted sublist
      // from 0 to top, exchanging as necessary
```

It is worth noting that, on each pass, the exchanges tend to move all values toward their final positions in the list. Thus it is possible for the list to be sorted before the final pass is completed. For the sake of efficiency, we should stop working as soon as possible to avoid any unnecessary passes.

In a sorted list, the bubble sort algorithm would not perform any exchanges, so we use a `boolean` flag to monitor this condition. The `boolean` value `sorted` is initialized to `false` so the initial pass will begin. At the beginning of each pass, the value is set to `true`, but if a single swap is required, it is set back to `false`.

Processing continues until `sorted` stays `true`, or we have completed the maximum number of passes.

The full method, for an array of strings, is as follows.

```
public static void bubbleSort (String[] list)
{
      boolean sorted = false;

      for (int top = list.length – 1; top > 0; top--)
      {
            sorted = true;
            for (int i = 1; i < top; i++)
            {
                  if (list[i].compareTo(list[i+1]) > 0)
                  {
                        sorted = false;
                        String temp = list[i];
                        list[i] = list[i+1];
                        list[i+1] = temp;
                  }
            }
      }
}
```

In general, bubble sort is not recommended for any serious applications. It is almost always slower than insertion and selection sorts because it usually involves far more data movement than they require.