CSC8012 Software Development Techniques and Tools — Part 3

Dr Konrad Dabrowski
konrad.dabrowski@newcastle.ac.uk

School of Computing
Newcastle University

Link to AVD:
https://rdweb.wvd.microsoft.com/arm/webclient/index.html

Searching and Sorting Algorithms: Topics

- Big-O notation
- Sorting algorithms
  - Selection sort
  - Insertion sort
- Searching algorithms
  - Sequential Search
  - Binary Search

Big-O Notation (1)

- We can describe an algorithm's performance by considering how many basic operations (assignments, number of comparisons etc.) are required for a given input of size n (e.g. an array with n items).
- Example: Consider an algorithm for displaying an array of numbers. For an input array of size n, how many operations are required?
- Big-O is a notation that gives a rough upper bound on an algorithm's performance.
- It only considers dominant terms (as n approaches infinity):

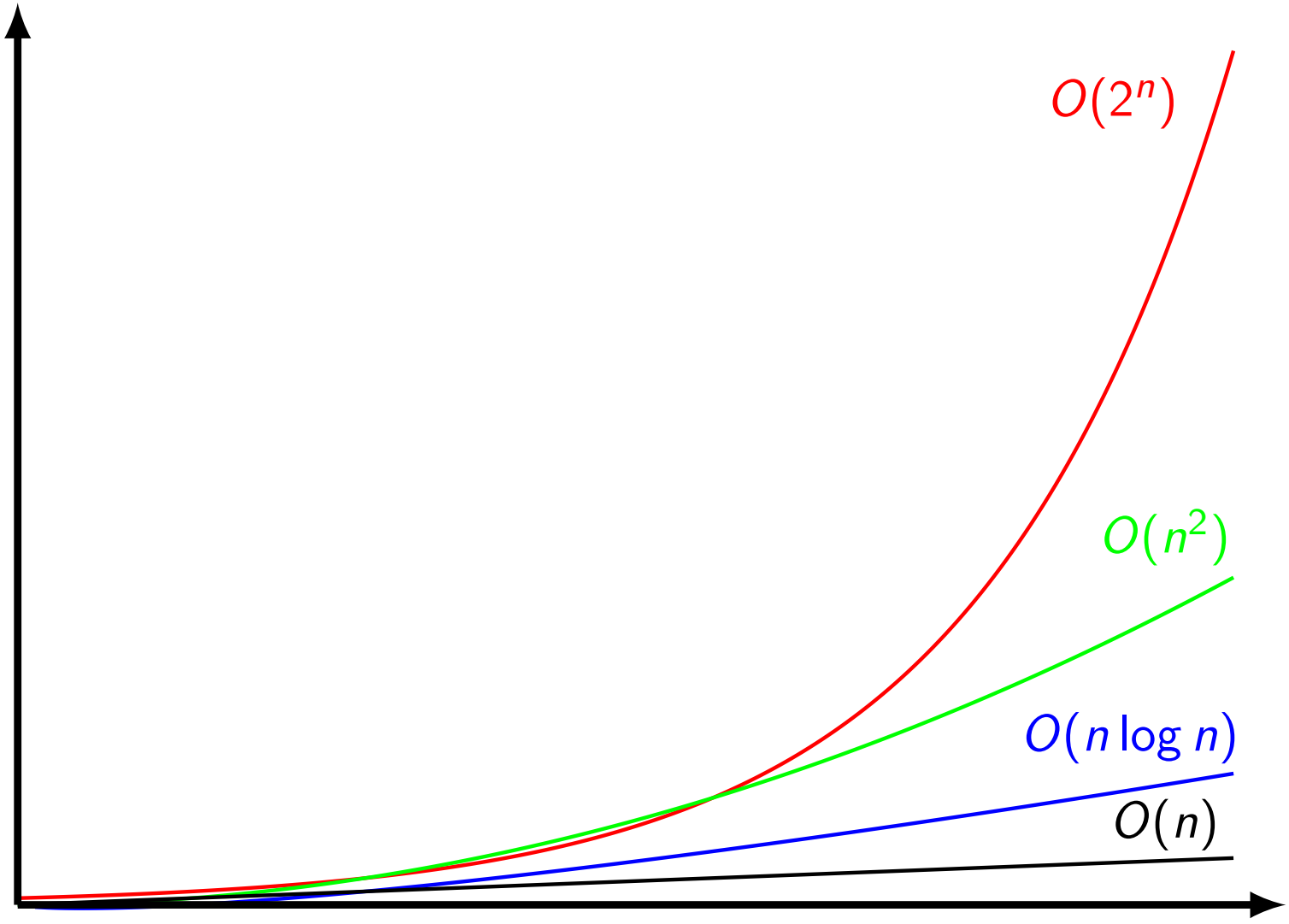
$$2n + 5 = O(n)$$
$$n^2 - 100n = O(n^2)$$
$$2^n + 5n^4 = O(2^n)$$

Big-O Notation (2)

- Consider what happens if we double input size n for different Big-O bounds:

$O(1)$	Nothing (constant)
$O(n)$	Doubles
$O(n \log n)$	Slightly more than double
$O(n^2)$	Factor of 4
$O(n^3)$	Factor of 8
$O(2^n)$	Squared

Growth Rate of Various Functions

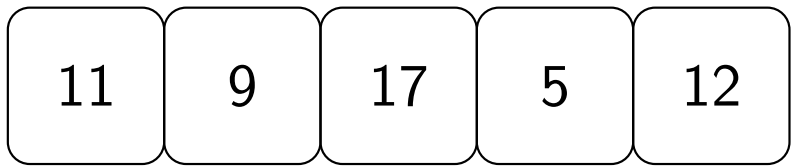


Selection Sort

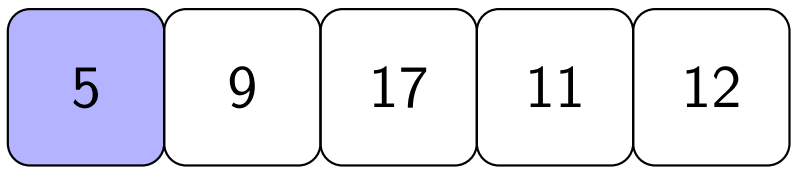
- Suppose we have an array with n items.
- *Selection sort* works by selecting the sorted items one at a time.
- The smallest item belongs in position 0 of the sorted array, so exchange it with whatever item is stored there.
- Now select the next smallest item from the $n - 1$ remaining items and store it at position 1.
- On the each subsequent iteration, select the smallest item from the remaining items and store it at the right position.
- After i iterations, the first i array locations will contain the first i smallest items in sort order. Thus, after $n - 1$ iterations, the whole array will be sorted.

Selection Sort Example

- Array in original order

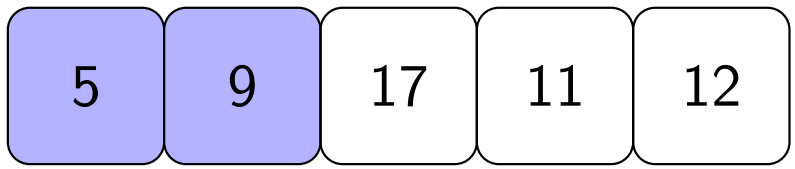


- Find the smallest and swap it with the first item.

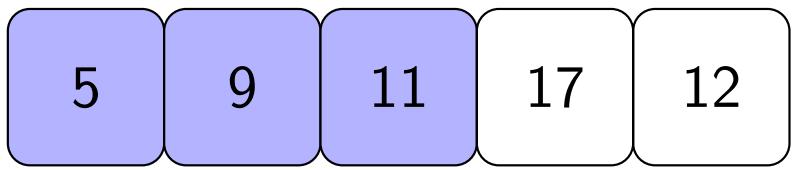


Selection Sort Example (Continued)

- Find the next smallest. It is already in the correct place.



- Find the next smallest and swap it with the first item of unsorted portion.



Selection Sort Algorithm

```
public static <E extends Comparable<? super E>> void selectionSort(E[] a)
{
    E temp;
    int minIndex;

    for (int j=0; j<a.length-1; j++)
    {
        // this is the outer loop
        minIndex = j;
        for (int k=j+1; k<a.length; k++)
        {
            if (a[k].compareTo(a[minIndex])<0)
            {
                minIndex = k;
            }
        }
        temp = a[minIndex];
        a[minIndex] = a[j];
        a[j] = temp;
    }
}
```

Selection Sort Performance

- For the array with n items we have:
  - The number of assignments related to swaps: $3(n-1) = O(n)$
  - The number of comparisons: $(n-1) + (n-2) + \cdots + 2 + 1 = n(n-1)/2 = O(n^2)$
- The performance of the algorithm is therefore $O(n^2)$
- This algorithm is good only for small size arrays because $O(n^2)$ grows rapidly as n grows.
- The selection sort algorithm does not depend on the initial arrangement of data. The number of comparisons is always $O(n^2)$ and the number of assignments is $O(n)$.

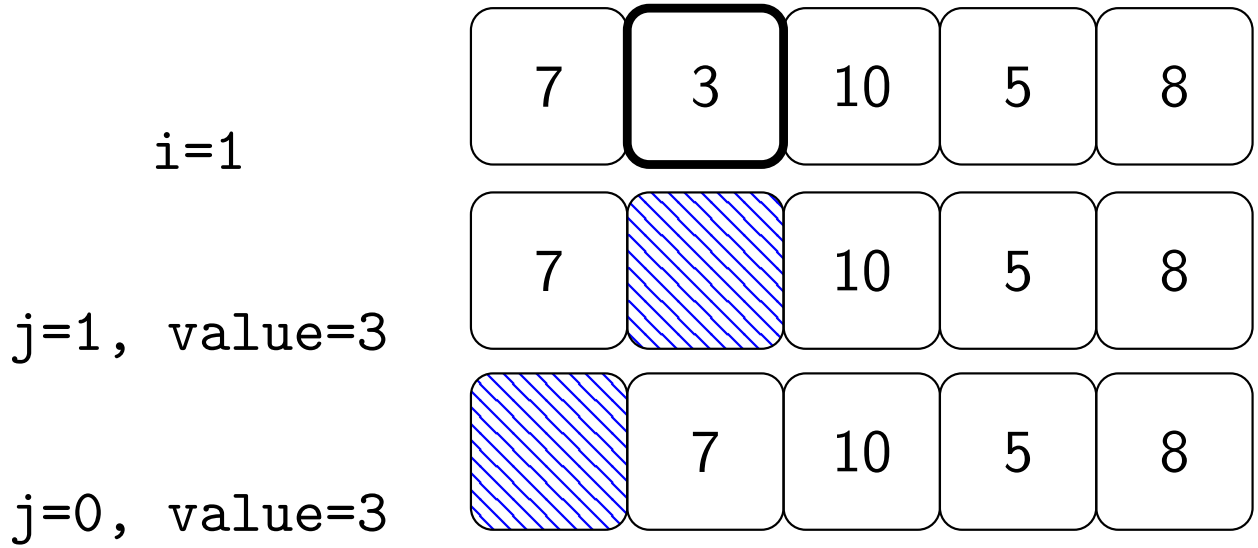
Insertion Sort

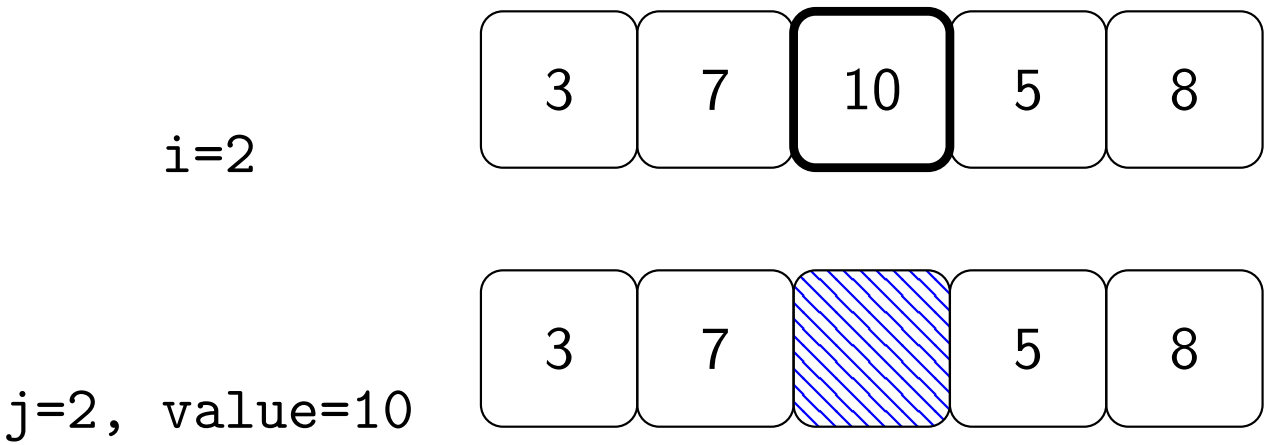
- The idea of *Insertion sort* is to sort the array from left to right.
- The array can be viewed as containing two parts: sorted and unsorted. At the beginning the sorted part contains only the item at position 0. The rest is considered unsorted.
- The items in the unsorted part are to be moved one at a time to their proper places in the sorted part.
- In each iteration, we take the first item of the unsorted part (called `value`) and all the items from the sorted part that should follow `value` are shifted by one position to the right to make room for `value`.
- After i iterations, the first $i + 1$ array items are sorted, so after $n - 1$ iterations the whole array is sorted.
- If the array is already nearly sorted, *Insertion sort* is very efficient.

The Insertion Sort Algorithm

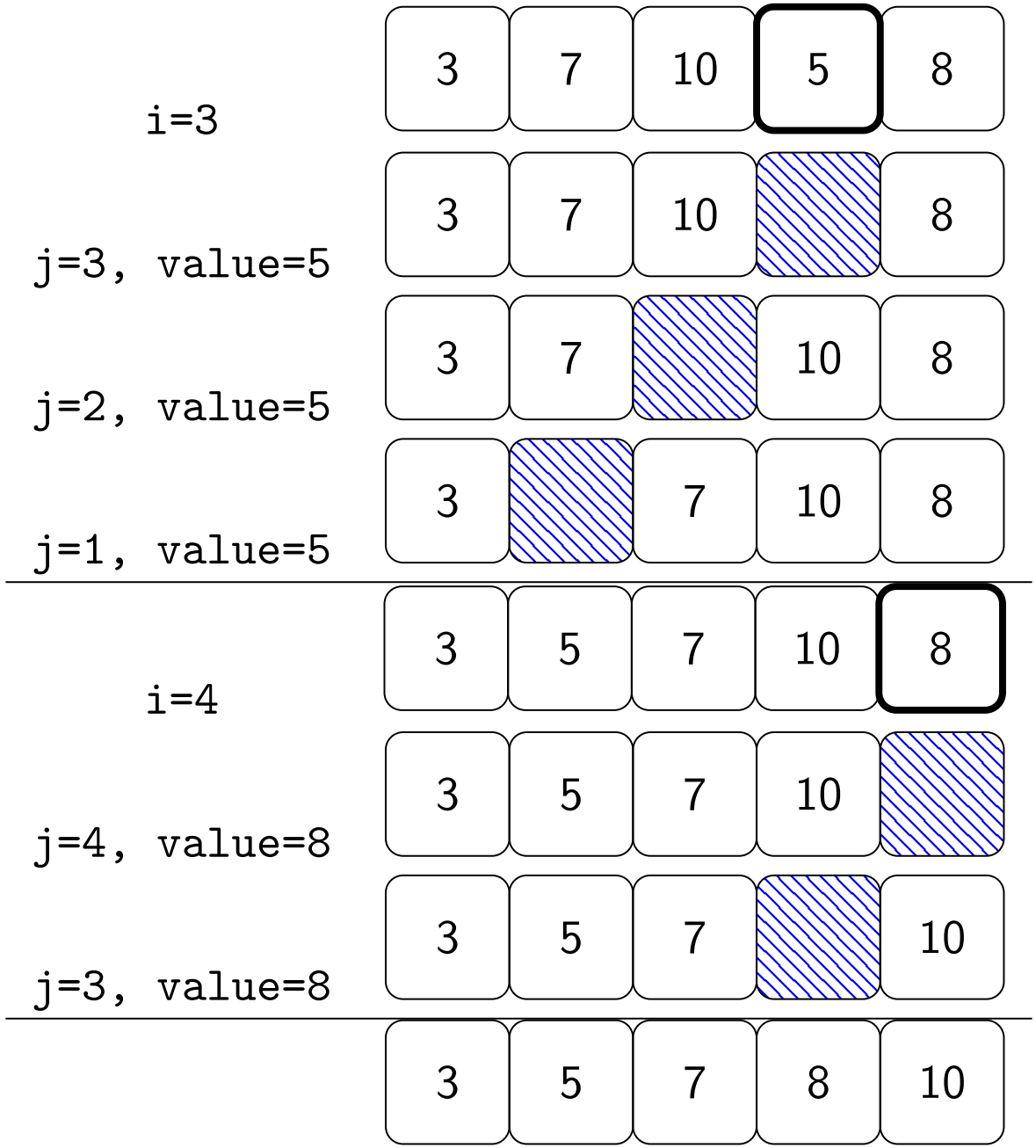
```
public static <E extends Comparable<? super E>> void insertionSort(E[] a)
{
    for (int i = 1; i < a.length; i++)
    {
        E value = a[i];
        int j;
        for (j = i; j > 0; j--)
        {
            if (a[j-1].compareTo(value)<0)
            {
                break;
            }
            else
            {
                a[j] = a[j-1];
            }
        }
        a[j] = value;
    }
}
```

Insertion Sort Example





Insertion Sort Example (Continued)



Insertion Sort Performance

- We analyse behaviour in terms of the number of comparisons C_n needed for an array of size n.
- We always execute the outer loop $n-1$ times.
- *Best case:* array is already sorted
  - The inner loop exits after the first comparison.
  - So $C_n = n-1$.
  - Performance is $O(n)$.
- *Worst case:* array is in reverse order (i.e. descending)
  - The inner loop is executes i times for $i = 1, \ldots, n-1$.
  - So $C_n = n(n-1)/2$
  - Performance is $O(n^2)$

Insertion Sort Performance (Continued)

- *Average case:* behaviour on random data
  - The inner loop is executed $i/2$ times for $i = 1, \ldots, n-1$.
  - Approximately $C_n = n^2/4$ comparisons.
  - Performance is $O(n^2)$.

Other Sorting Algorithms

- There are other sorting algorithms. For example: *Bubble sort*, *Shell sort*, *Merge sort*, *Quick sort*.
- *Merge sort* and *Quick sort* are examples of *divide-and-conquer* algorithms: they partition the array and recursively sort the partitions.
- Elementary sorting algorithms which are based on comparing adjacent array items are usually $O(n^2)$.
- However, "divide-and-conquer" algorithms can have better performance:
  - *Merge sort* is $O(n \log n)$.
  - *Quick sort*, on average, makes $O(n \log n)$ comparisons to sort n items. In the worst case, however, it makes $O(n^2)$ comparisons, though this behaviour is rare. *Quick sort* is often faster in practice than algorithms like *Merge sort*.
- It can be proved that any sorting algorithm which uses comparisons requires at least $O(n \log n)$ operations.

Sorting in a Java Program

- The `Arrays` class of `java.util` package contains static sort methods for primitive types and objects.
- Example - to sort an `Array` of integers

```
int[] intArray = {7, 3, 10, 5, 8};
Arrays.sort(intArray);
```

- A tuned *Quick sort* method is used for the sorting.

Sequential Search

- A *sequential search* of an array examines each item in turn, looking for a specific item.
- The search terminates when a matching item is found or when there are no items left.
- If the array is ordered, then it is possible to terminate the search sooner.
- Sometimes it is possible to rearrange the items in the array to speed up subsequent searches.
- However, all sequential searching algorithms are $O(n)$ in the worst-case.

Sequential Search Algorithm

```
public static <E extends Comparable<? super E>>
                                    int seqSearch(E[] a, E searchItem)
{
    int loc;
    boolean found = false;

    for (loc = 0; loc < a.length; loc++)
    {
        if (a[loc].compareTo(searchItem) == 0)
        {
            found = true;
            break;
        }
    }

    if (found)
        return loc;
    else
        return -1;
}
```

Ordered Searching

- A disadvantage of a sequential search is that it is necessary to examine every item in case the item we are looking for is not present. If the items of the array are sorted into order, then the search can be terminated early.
- Assuming the items are sorted in increasing order, once a "bigger" item (according to the `compareTo` method) is found, there is no point in looking any further.
- But this is still an $O(n)$ algorithm.
- If we are searching an ordered array, we can do much better using *Binary Search*.

Binary Search

- A sequential search of an ordered array is still $O(n)$ even though only half the items have to be examined on average. However, an important property of an array is that you do not have to access its items in sequence.
- You can access any array item at constant cost i.e. array access is $O(1)$.
- It is possible to exploit this property to develop an $O(\log_2 n)$ algorithm for searching a sorted array.

Binary Search: Explanation

- At each stage, the array is divided into two roughly equal halves.
- By examining the item stored at the mid-point, it is easy to determine which half of the array might contain an item being sought (assuming such an item is present).
- Thus, at each stage the number of items to be searched is halved, so that after $\log_2 n$ stages there is only one item left to be examined.

Binary Search Algorithm

```
public static <E extends Comparable<? super E>>
                                    int binarySearch(E[] a, E searchItem)
{
    int first = 0;
    int last = a.length - 1;
    int mid = -1;
    boolean found = false;

    while (first <= last && !found)
    {
        // comparing searchItem with the middle element of the
        // currently examined portion of the array
    }

    if (found)
        return mid;
    else
        return -1;
}
```

Binary Search Algorithm: the `while` Loop

```
while (first <= last && !found)
{
    mid = (first + last) / 2;
    int result = a[mid].compareTo(searchItem);
    if (result == 0)
        found = true;
    else
        if (result > 0)
            last = mid - 1;
        else
            first = mid + 1;
}
```

Binary Search Performance

- Suppose a is a sorted array of size n.
- Moreover, suppose that n is a power of 2, for example: $n = 2^m$ ($m = \log_2 n$).
- After each iteration of the `while` - loop about half the items are left to search. This means the search sub-array for the next iteration is half the size of the current sub-array. For example, after the first iteration, the search sub-array is of size $n/2 = 2^{m-1}$.
- The maximum number of iterations for the `while` loop is $m + 1$, and each iteration makes two comparisons
- So, the maximum number of comparisons to determine whether an item `x` is in a is: $2(m + 1) = 2(\log_2 n + 1) = 2 \log_2 n + 2 = O(\log_2 n)$.

Sorting and Searching ArrayLists

- The Collections class of java.util package contains static sort and binarySearch methods:
  - The sort method sorts the specified list into ascending order, according to the natural ordering of its elements. All elements in the list must implement the Comparable interface.
  - The binarySearch method searches the specified list for the specified object using the binary search algorithm. The list must be sorted into ascending order according to the natural ordering of its elements prior to making this call. It returns the index of the search object, if it is contained in the list.
- These methods can be used to sort an ArrayList of objects:

```
int[] intArray = {13, 2, 10, 7, 1};
ArrayList<Integer> integers = new ArrayList<Integer>();
for (Integer i: intArray) integers.add(i);
Collections.sort(integers);
int index = Collections.binarySearch(integers, 10);
```