

# **Unit 7: ArrayList**

## **Searching and Sorting**

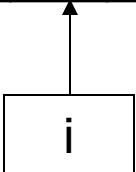
Adapted from:

- 1) Building Java Programs: A Back to Basics Approach  
by Stuart Reges and Marty Stepp
- 2) Runestone CSAwesome Curriculum

# Sequential search

- **sequential search:** Locates a target value in an array/list by examining each element from start to finish. If found, return index of first occurrence. Otherwise, return -1.
  - How many elements will it need to examine?
  - Example: Searching the array below for the value **42**:

index	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
value	-4	2	7	10	15	20	22	25	30	36	42	50	56	68	85	92	103



- Notice that the array is sorted. Could we take advantage of this?

# Sequential search

Implement sequential search using arrays.

```
public int sequentialSearch(int[] array, int target){  
    for(int i = 0; i < array.length; i++){  
        if(array[i] == target)  
            return i;  
    }  
    // target not in array  
    return -1;  
}
```

# Binary search (13.1)

- **binary search:** Locates a target value in a *sorted* array/list by successively eliminating half of the array from consideration.
  - How many elements will it need to examine?
  - Example: Searching the array below for the value **42**:

index	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
value	-4	2	7	10	15	20	22	25	30	36	42	50	56	68	85	92	103

min

mid

max

# Binary search

Implement binary search using arrays. Assume that the array is sorted.

```
public int binarySearch(int[] sortedArray, int target){
    int low = 0, high = sortedArray.length - 1;
    while (low <= high) {
        int mid = (low + high) / 2;
        if (sortedArray[mid] < target)
            low = mid + 1;
        else if (sortedArray[mid] > target)
            high = mid - 1;
        else if (sortedArray[mid] == target)
            return mid;
    }
    return -1;
}
```

# Sorting

- **sorting**: Rearranging the values in an array or collection into a specific order (usually into their "natural ordering").
  - one of the fundamental problems in computer science
  - can be solved in many ways:
    - there are many sorting algorithms
    - some are faster/slower than others
    - some use more/less memory than others
    - some work better with specific kinds of data
    - some can utilize multiple computers / processors, ...
  - *comparison-based sorting* : determining order by comparing pairs of elements:
    - `<`, `>`, `compareTo`, ...

# Sorting algorithms

There are many sorting algorithms. Wikipedia lists over 40 sorting algorithms.

The following three sorting algorithm will be on the AP exam.

**selection sort:** look for the smallest element, swap with first element. Look for the second smallest, swap with second element, etc...

**insertion sort:** build an increasingly large sorted front portion of array.

**merge sort:** recursively divide the array in half and sort it. Merge sort will be discussed in Unit 10.

# Selection sort

**selection sort:** Orders a list of values by repeatedly putting the smallest or largest unplaced value into its final position.

The algorithm:

- Look through the list to find the smallest value.
- Swap it so that it is at index 0.
- Look through the list to find the second-smallest value.
- Swap it so that it is at index 1.
- ...
- Repeat until all values are in their proper places.



# Selection sort example

- Initial array:

index	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
value	22	18	12	-4	27	30	36	50	7	68	91	56	2	85	42	98	25

- After 1st, 2nd, and 3rd passes:

index	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
value	<b>-4</b>	18	12	<b>22</b>	27	30	36	50	7	68	91	56	2	85	42	98	25

index	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
value	-4	<b>2</b>	12	22	27	30	36	50	7	68	91	56	<b>18</b>	85	42	98	25

index	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
value	-4	2	<b>7</b>	22	27	30	36	50	<b>12</b>	68	91	56	18	85	42	98	25

# Selection Sort

Implement selection sort.

```
public static void selectionSort(int arr[]){  
    for (int i = 0; i < arr.length - 1; i++){  
        // find smallest from i to end of array  
        int min_idx = i;  
        for (int j = i+1; j < arr.length; j++){  
            if (arr[j] < arr[min_idx])  
                min_idx = j;  
        }  
        // swap minimum with element at index i  
        int temp = arr[min_idx];  
        arr[min_idx] = arr[i];  
        arr[i] = temp;  
    }  
}
```

# Insertion Sort

**insertion sort:** Shift each element into a sorted sub-array

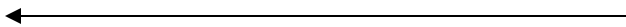
The algorithm: To sort a list of  $n$  elements.

Loop through indices  $i$  from 1 to  $n - 1$ :

- For each value at position  $i$ , inserted into correct position in the sorted list from index 0 to  $i - 1$ .

index	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
value	-4	12	18	22	27	30	36	50	7	68	91	56	2	85	42	98	25

sorted sub-array (indexes 0-7)



# Insertion Sort Algorithm

- 64 **54** 58 87 55
  - 54 less than 64
  - Insert 54 before 64
- 54 64 **58** 87 55(1<sup>st</sup> pass)
  - 58 less than 64
  - 58 greater than 54
  - Insert 58 before 64
- 54 58 64 **87** 55(2<sup>nd</sup> pass)
  - 87 greater than 64
  - Go to next element
- 54 58 64 87 **55**(3<sup>rd</sup> pass)
  - 55 less than 87
  - 55 less than 64
  - 55 less than 58
  - 55 greater than 54
  - Insert 55 before 58
- 54 55 58 64 87(4<sup>th</sup> pass)



# Insertion Sort

Since insertion sort involves inserting and shifting elements, let's use an arraylist for the sort.

```
public void insertionSort(ArrayList<Integer> list) {  
    for(int i = 1; i < list.size(); i++) {  
        int current = list.remove(i); // removes & returns  
        int index = i - 1;  
        while(index >= 0 && current < list.get(index))  
            index--;  
        list.add(index+1, current);  
    }  
}
```

Note that implementing this sort using an array will require manually shifting elements. See lab 1.

# The Complexity of An Algorithm

The **complexity** of an algorithm is the amount of resources (elementary operations or loop iterations) required for running it. (lower the complexity = faster algorithm)

The complexity  $f(n)$  is defined in terms of the input size  $n$ . For example, sorting an array should take more resources for larger arrays.

We can approximate the complexity of simple algorithms by counting the number of iterations in the algorithm.

# Complexity of Algorithms

Assume the following algorithms are performed on an array of size  $n$ .

Sequential Search: for loop in the algorithm requires approximately  $n$  iterations. This is a linear complexity algorithm.

Binary Search: This is a bit harder. Since each comparison eliminates half of the array, it takes approximately  $\log_2(n)$  iterations. For example, if an array has length 32, it takes about 5 comparisons since  $2^5 = 32$ .

# Complexity of Algorithms

Assume the following algorithms are performed on an array of size  $n$ .

Selection Sort: nested for loops = approximately  $n^2$  iterations.  
(Quadratic complexity).

Insertion Sort: nested for loops = approximately  $n^2$  iterations.  
(Quadratic complexity).

Note: Searching is much faster than sorting.



# Complexity of Algorithms

The performance of some algorithms can depend on the data of the array.

What is the best-case scenario for insertion sort?

Answer: the array is already sorted

What is the worst-case scenario for insertion sort?

Answer: the array is reverse sorted

What's the best-case, worst-case scenario for selection sort?

Answer: None, all cases are the same.

Can you explain why?

# Lab 1

Write the following methods.

- 1) Reimplement sequentialSearch using an arraylist of Integers.
- 2) Write the insertion sort method using arrays instead of arraylist.

# Lab 2(Processing)

## Resolving Wall Collisions.

This lab will allow us to resolve wall collisions. After this lab, we can write top-down games(view is from the ceiling) where characters walk through a map, collecting objects and are not allowed to walk through walls/obstacles.

You should have completed the previous Processing lab: Collision Detection where we discussed how to detect collision between two Sprites. The template for this lab will contain the solution to that previous lab.