Unit 4: Data Collections
Searching and Sorting

Adapted from:

1) Building Java Programs: A Back to Basics Approach
by Stuart Reges and Marty Stepp

2) Runestone CSAwesome Curriculum

https://longbaonguyen.github.io

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

i

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

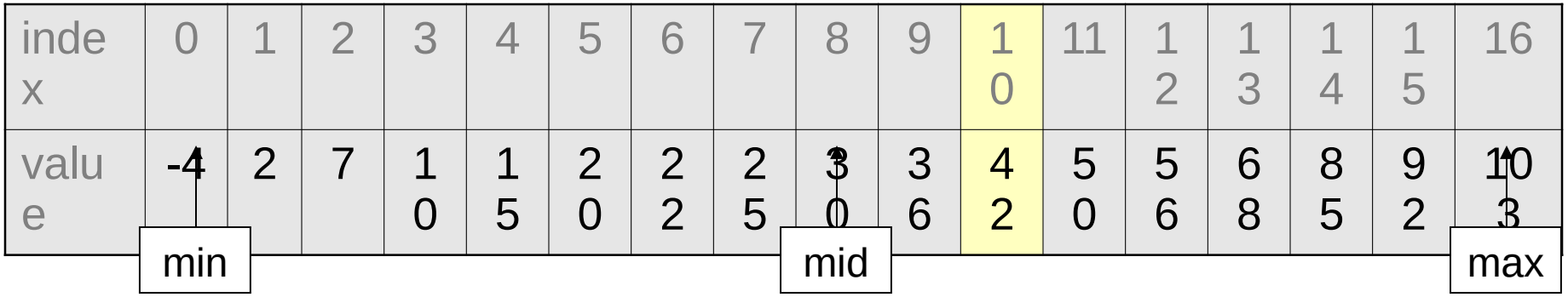
Sequential search

Implement sequential search using arrays which returns the index of the target or -1 if it is not found.

```
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**:



Binary search

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

```
public int binarySearch(int[] sortedArray, int target){
      int min = 0, max = sortedArray.length – 1;
      while (min <= max) {
        int mid = (min + max) / 2;
        if (sortedArray[mid] < target)
            min = mid + 1;
        else if (sortedArray[mid] > target)
            max = 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	23	18	12	-4	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
valu	**-4**	1	1	**22**	2	3	3	5	7	6	9	5	2	8	4	9	2

index	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
valu	-4	**2**	1	22	2	3	3	5	7	6	9	5	**18**	8	4	9	2

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

2) After the kth pass, the first k elements are in their final sorted positions.

Selection Sort

Implement selection sort.

```
public static void selectionSort(int[] elements){
  for (int j = 0; j < elements.length - 1; j++){
    int minIndex = j;
    for (int k = j + 1; k < elements.length; k++){
      if (elements[k] < elements[minIndex])
        minIndex = k;
    }
    if (j != minIndex){ // swap only if different
      int temp = elements[j];
      elements[j] = elements[minIndex];
      elements[minIndex] = 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
  - 58 less than 64
  - 58 greater than 54
  - Insert 58 before 64
- 54 58 64 **87** 55
  - 87 greater than 64
  - Go to next element
- 54 58 64 87 **55**
  - 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

Insertion Sort

This implementation uses ArrayLists. This code has appeared on MC questions on the AP exam. Understand it!

```
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);
  }
}
```

Insertion Sort

This implementation uses arrays. This code has appeared on MC questions on the AP exam. Understand it!

```
public static void insertionSort(int[] elements){
 for (int j = 1; j < elements.length; j++){
   int temp = elements[j];
   int possibleIndex = j;
   while (possibleIndex > 0 && temp <
                              elements[possibleIndex - 1]){
     elements[possibleIndex] = elements[possibleIndex - 1];
     possibleIndex--;
   }
   elements[possibleIndex] = temp;
 }
}
```

Insertion Sort

Some properties of insertion sort:

1) For an array of *n* elements, the array is sorted after *n* − 1 passes.

2) After the *k*th pass, a[0], a[1], ..., a[k] are sorted with respect to each other but not necessarily in their final sorted positions.

3) The worst case for insertion sort occurs if the array is initially sorted in reverse order, since this will lead to the maximum possible number of comparisons and moves.

4) The best case for insertion sort occurs if the array is already sorted in increasing order. In this case, each pass through the array will involve just one comparison, which will indicate that “it” is in its correct position with respect to the sorted list. Therefore, no elements will need to be moved.

Note: For selection sort, there is no worst or best case. Finding the smallest at each iteration requires traversing the array to the end.

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 of the worst case scenario.

Example 1

How many times as a function of n does the computation x++ executed?

```
int x = 0;
for(int i = 0; i < n; i++){
  x++;
}
```

Answer: n(linear function of n)

Example: Sequential/Linear Search has complexity n.

Example 2

How many times as a function of n does the computation x++ executed?

```
int x = 0;
for(int i = 0; i < n; i++){
  x++;
}

for(int j = 0; j < n; j++){
  x++;
}
```

Answer: 2n
Example: Sequential Search

Example 3

How many times as a function of n does the computation x++ executed?

```
int x = 0;
for(int i = 0; i < n; i++){
  for(int j = 0; j < n; j++){
        x++;
  }
}
```

Answer: n^2

Example: Selection and Insertion are both quadratic in complexity.

Example 4

How many times as a function of n does the computation x++ executed?

```
int x = 1;
while((int)(Math.pow(2, x)) <= n){
  x++;
}
```

Answer: log_2(n)

Example: Binary Search has complexity log_2(n).

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.

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.