Revised: 6/20/2020

Sorting I

Michael C. Hackett

Assistant Professor, Computer Science

Community College of Philadelphia

Lecture Topics

- Bubble Sort
  - Bubble Sort Improvements
  - Cocktail Shaker Sort
- Insertion Sort
- Selection Sort
- Comparable Interface (Java)
- Comparator Interface (Java)
- Recursive Sorting
  - Bubble Sort
  - Merge Sort
  - Quicksort

Sorting Algorithms

- A **sorting algorithm** is an algorithm that organizes a sequence of data (like an array) into some type of order.
  - Usually ascending or descending order.

- An **ascending sort** means arranging the values from smallest to largest.
  - {3, 4, 2, 5, 1} {1, 2, 3, 4, 5}

- A **descending sort** means arranging the values from largest to smallest.
  - {3, 4, 2, 5, 1} {5, 4, 3, 2, 1}

Sorting Algorithms

- A **comparative sorting algorithm** sorts the contents of a sequence through a repeated process of comparing values in the sequence.
  - Usually less-than or greater-than comparisons.

- A **non-comparative sorting algorithm** sorts the contents of a sequence using characters from the values to be sorted.
  - Non-comparative sorting is the next lecture.

Bubble Sort Algorithm

- The Bubble Sort algorithm is a comparative sorting algorithm.
- Neighboring pairs of values are compared and swapped so that they are in the correct order.
- The algorithm repeats this process *n*-number of times, where *n* is the length of the array.

Bubble Sort Algorithm

- Here is a link to a video with a visualization of the Bubble Sort sorting a short array of numbers:

https://www.youtube.com/watch?v=xli_FI7CuzA

- The pseudocode at the end of the video is a little different from the C++ implementation on the next slide.

Bubble Sort (C++ Function)

```
void bubbleSort(int a[], int length) {
    for(int i = 0; i < length; i++) {
        for(int j = 1; j < length; j++) {
            if(a[j-1] > a[j]) {
                int temp = a[j-1];
                a[j-1] = a[j];
                a[j] = temp;
            }
        }
    }
}
```

Bubble Sort Algorithm Improvements

- One variation of the Bubble Sort allows it to end early if the sequence is in order.
  - The algorithm begins by assuming the sequence is in order.
  - It traverses the sequence, making any necessary swaps.
  - If it made at least one swap, the process repeats.
  - If it made no swaps, the algorithm terminates.

Bubble Sort (Improved 1) (C++ Function)

```
void bubbleSort(int a[], int length) {
    bool sorted;
    do {
        sorted = true;
        for(int i = 0; i < length-1; i++) {
            if(a[i] > a[i+1]) {
                int temp = a[i];
                a[i] = a[i+1];
                a[i+1] = temp;
                sorted = false;
            }
        }
    } while(!sorted);
}
```

Bubble Sort Algorithm Improvements

- Another improvement of the original Bubble Sort allows it to only traverse up to the last element that was sorted.
  - The algorithm is almost identical to the original, but the inner loop only continues while the inner loop's counter is < *length – the outer loop's counter*
  - For example, when the sequence is half-way sorted there is no need to check the second half of the sequence, since those elements are already in order.
    - The original Bubble Sort (and the alternative implementation previously shown) unnecessarily checks those already sorted elements.

Bubble Sort (Improved 2) (C++ Function)

```
void bubbleSort(int a[], int length) {
    for(int i = 0; i < length; i++) {
        for(int j = 1; j < length - i; j++) {
            if(a[j-1] > a[j]) {
                int temp = a[j-1];
                a[j-1] = a[j];
                a[j] = temp;
            }
        }
    }
}
```

Bubble Sort Variations Compared

- Original Bubble Sort

  Total Comparisons **= n * (n-1)**
  - Length of 5: Total Comparisons = 5 * (5-1) = 20
  - Length of 10: Total Comparisons = 10 * (10-1) = 90
- Bubble Sort Improved 1

  n-1 <= Total Comparisons **<= n * (n-1)**
  - Length of 5: 4 <= Total Comparisons <= 20
  - Length of 10: 9 <= Total Comparisons <= 90
- Bubble Sort Improved 2

  Total Comparisons **=** $\sum_{i=1}^{n-1} i$
  - Length of 5: Total Comparisons = 1 + 2 + 3 + 4 = 10
  - Length of 10: Total Comparisons = 1 + 2 + 3 + 4 + 5 + 6 + 7 + 8 + 9 = 45

Bubble Sort Variations Compared

- The following examples use small sequences, but should still illustrate the amount of work required by each algorithm.
  - Each assumes sorting in ascending order.

- {5, 1, 2, 3, 4}
  - Original Bubble Sort: 20 comparisons
  - **Bubble Sort Improved 1: 8 comparisons**
  - Bubble Sort Improved 2: 10 comparisons

Bubble Sort Versions Compared

- {1, 2, 3, 4, 5}   (Already sorted)
  - Original Bubble Sort: 20 comparisons
  - **Bubble Sort Improved 1: 4 comparisons**
  - Bubble Sort Improved 2: 10 comparisons

- {5, 4, 3, 2, 1}   (Worst Case)
  - Original Bubble Sort: 20 comparisons
  - Bubble Sort Improved 1: 20 comparisons
  - **Bubble Sort Improved 2: 10 comparisons**

Cocktail Shaker Sort Algorithm

- The Cocktail Shaker Sort algorithm is another variation of the Bubble Sort algorithm.
- Neighboring pairs of values in sequence are compared and then swapped, moving the largest values to the right.
  - If performing an ascending sort.
- But then it goes backwards, moving the smallest values to the left.

https://www.youtube.com/watch?v=njClLBoEbfI

Cocktail Shaker Sort (C++ Function)

```
void cocktailSort(int a[], int length) {
     bool sorted;
     int i;
     do {
```

Forward

```
          sorted = true;
          for(int i = 1; i < length; i++) {
               if(a[i-1] > a[i]) {
                    int temp = a[i-1];
                    a[i-1] = a[i];
                    a[i] = temp;
                    sorted = false;
               }
          }
          if(sorted) {
               break;
          }
```

Reverse

```
          sorted = true;
          for(int j = length-1; j > 0; j--) {
               if(a[j] < a[j-1]) {
                    int temp = a[k-1];
                    a[k-1] = a[k];
                    a[k] = temp;
                    sorted = false;
               }
          }
     } while(!sorted);
}
```

Cocktail Shaker Sort Algorithm

- The Cocktail Shaker Sort algorithm behaves similarly to the (Improved 1) Bubble Sort algorithm.
  - Terminates early if it didn't make any swaps

- A variation of this algorithm (which sets high and low boundaries) skips the already sorted values at the beginning and end of the sequence.
  - Similar to the behavior of the (Improved 2) Bubble Sort algorithm

Cocktail Shaker Sort (Improved) (C++ Function)

```
void cocktailSort(int a[], int length) {
     int lowBound = 0;
     int highBound = length - 1;

     while(lowBound < highBound) {
```

Forward

```
          for(int i = lowBound; i < highBound; i++) {
               if(a[i] > a[i+1]) {
                    int temp = a[i+1];
                    a[i+1] = a[i];
                    a[i] = temp;
               }
          }
          highBound--;
```

Reverse

```
          for(int i = highBound; i > lowBound; i--) {
               if(a[i-1] > a[i]) {
                    int temp = a[i-1];
                    a[i-1] = a[i];
                    a[i] = temp;
               }
          }
          lowBound++;
     }
}
```

Cocktail Shaker Sort Variations Compared

- Original Cocktail Shaker Sort

  Odd length: **n-1 <=** Total Comparisons **<= n * (n-1)**

  Even length: **n-1 <=** Total Comparisons **<=** (**n+1) * (n-1)**

  - Length of 5: 4 <= Total Comparisons <= 20
  - Length of 10: 9 <= Total Comparisons <= 99

- Cocktail Shaker Sort Improved

  Total Comparisons **=** $\sum_{i=1}^{n-1} i$

  - Length of 5: Total Comparisons = 1 + 2 + 3 + 4 = 10
  - Length of 10: Total Comparisons = 1 + 2 + 3 + 4 + 5 + 6 + 7 + 8 + 9 = 45

Cocktail Shaker Sort Variations Compared

- The following examples use small sequences, but should still illustrate the amount of work required by each algorithm.
  - Each assumes sorting in ascending order.

- {5, 1, 2, 3, 4}
  - **Original Cocktail Shaker Sort: 8 comparisons**
  - Cocktail Shaker Sort Improved: 10 comparisons

Cocktail Shaker Sort Versions Compared

- {1, 2, 3, 4, 5} (Already sorted)
  - **Original Cocktail Shaker Sort: 4 comparisons**
  - Cocktail Shaker Sort Improved: 10 comparisons
- {5, 4, 3, 2, 1} (Worst Case)
  - Original Cocktail Shaker Sort: 20 comparisons
  - **Cocktail Shaker Sort Improved: 10 comparisons**

Selection Sort Algorithm

- The Selection Sort algorithm is a comparative sorting algorithm.
- The algorithm divides the array into sorted and unsorted partitions.
- The algorithm searches (linearly) through the unsorted partition to find the smallest value.
  - If performing an ascending sort.
- The smallest value found is swapped to the beginning of the unsorted partition, thus extending the sorted partition by 1.

Selection Sort Algorithm

- Here is a link to a video with a visualization of the Selection Sort sorting a short array of numbers.

https://www.youtube.com/watch?v=g-PGLbMth_g

- The pseudocode at the end of the video is a little different from the C++ implementation on the next slide.

Selection Sort (C++ Function)

```
void selectionSort(int a[], int length) {
    for(int i = 0; i < length-1; i++) {
        int smallest = i;
        for(int j = i+1; j < length; j++) {
            if(a[j] < a[smallest]) {
                smallest = j;
            }
        }
        if(smallest != i) {
            int temp = a[smallest];
            a[smallest] = a[i];
            a[i] = temp;
        }
    }
}
```

Insertion Sort Algorithm

- The Insertion Sort algorithm is a comparative sorting algorithm.
- Like the Selection Sort, it divides an array into sorted and unsorted partitions.
- The algorithm takes the first element in the unsorted portion, and keeps swapping backwards until it finds a smaller value in the sorted portion.
  - If performing an ascending sort

Insertion Sort Algorithm

- Here is a link to a video with a visualization of the Insertion Sort sorting a short array of numbers.

https://www.youtube.com/watch?v=JU767SDMDvA

- The pseudocode at the end of the video is a little different from the C++ implementation on the next slide.

Insertion Sort (C++ Function)

```
void insertionSort(int a[], int length) {
    for(int i = 1; i < length; i++) {
        int value = a[i];
        int j = i-1;
        while(j >= 0 && a[j] > value) {
            a[j+1] = a[j];
            j--;
        }
        a[j+1] = value;
    }
}
```

Selection Sort and Insertion Sort Compared

- Selection Sort

  Total Comparisons **=** $\sum_{i=1}^{n-1} i$
  - Length of 5: Total Comparisons = 1 + 2 + 3 + 4 = 10
  - Length of 10: Total Comparisons = 1 + 2 + 3 + 4 + 5 + 6 + 7 + 8 + 9 = 45

- Insertion Sort

  n-1 <= Total Comparisons **<=** $\sum_{i=1}^{n-1} i$
  - Length of 5: 4 <= Total Comparisons <= 10
  - Length of 10: 10 <= Total Comparisons <= 45

Selection Sort and Insertion Sort Compared

- The following examples use small sequences, but should still illustrate the amount of work required by each algorithm.
  - Each assumes sorting in ascending order.

- {5, 1, 2, 3, 4}
  - Selection Sort: 10 comparisons
  - **Insertion Sort: 7 comparisons**

Selection Sort and Insertion Sort Compared

- {1, 2, 3, 4, 5}   (Already sorted)
  - Selection Sort: 10 comparisons
  - **Insertion Sort: 4 comparisons**

- {5, 4, 3, 2, 1}   (Worst Case)
  - **Selection Sort: 10 comparisons**
  - **Insertion Sort: 10 comparisons**

These Algorithms Compared

- {5, 1, 2, 3, 4}
    - **Insertion Sort: 7 comparisons**
    - Original Cocktail Shaker Sort: 8 comparisons
    - Bubble Sort Improved 1: 8 comparisons
    - Selection Sort: 10 comparisons
    - Cocktail Shaker Sort Improved: 10 comparisons
    - Bubble Sort Improved 2: 10 comparisons
    - Original Bubble Sort: 20 comparisons

These Algorithms Compared

- {1, 2, 3, 4, 5}   (Already sorted)
  - **Insertion Sort: 4 comparisons**
  - **Original Cocktail Shaker Sort: 4 comparisons**
  - **Bubble Sort Improved 1: 4 comparisons**
  - Selection Sort: 10 comparisons
  - Cocktail Shaker Sort Improved: 10 comparisons
  - Bubble Sort Improved 2: 10 comparisons
  - Original Bubble Sort: 20 comparisons

These Algorithms Compared

- {5, 4, 3, 2, 1}   (Worst Case)
  - **Insertion Sort: 10 comparisons**
  - **Cocktail Shaker Sort Improved: 10 comparisons**
  - **Bubble Sort Improved 2: 10 comparisons**
  - **Selection Sort: 10 comparisons**
  - Original Cocktail Shaker Sort: 20 comparisons
  - Original Bubble Sort: 20 comparisons
  - Bubble Sort Improved 1: 20 comparisons

- Of *these* algorithms, **insertion sort will generally be the best choice**.

Complexity

- All seven algorithms shown have polynomial time complexity.
  - $O(n^2)$

- All seven algorithms shown have linear space complexity.
  - $O(n)$
  - Determined by the length of the sequence
  - Auxiliary space needed for all seven is constant
    - $O(1)$

Sorting Arrays in Java

- Of course, all of these algorithms could be implemented using Java (or any other programming language)

- Many languages have sorting methods built-in.

- For example, Java's Arrays.sort() method can be used to sort an array of numbers or Strings.
  - (in ascending order)
  - Must be imported from **`java.util.Arrays`**

Sorting Arrays in Java

```
int[] a = {5, 4, 3, 2, 1};
Arrays.sort(a);
```

- This is helpful (and quick) way to use sorting in a Java program.
- Though, how would this method know how to sort custom objects that you defined?
- If the object has multiple fields, how will the method know which one to use in its sorting process?

Comparable Interface

- Java’s Comparable interface contains one abstract method that it requires implementing classes to define.
  - Syntax: **public class MyObj implements Comparable<MyObj>**

- Required method:

  public int compareTo(MyObj x)
  - Returns:
    - A positive number if >= x
    - A negative number if <= x
    - 0 if == x

Comparable Interface

```
public int compareTo(MyObj x) {
    return this.field – x.field;
}
```

- If this object’s field was 7 and x’s field was 5, the result is 2
  - Positive, so this object is “greater than” x
- If this object’s field was 4 and x’s field was 8, the result is -4
  - Negative, so this object is “less than” x
- If this object’s field was 6 and x’s field was 6, the result is 0
  - Zero, so this object is “equal to” x

Comparable Interface

- If the object is implementing Comparable, then methods like Arrays.sort() will be able to sort an array of these objects.
  - It knows to use the object's compareTo methods, which defined how the objects are to be sorted.

- A drawback is this only allows us one way to sort the objects.
  - There may be multiple fields we wish to sort the objects by.
  - We may want to sort by one field but later sort the same objects using a different field.

Comparator Interface

- The Comparator interface is used to make multiple *comparators*
  - Classes that implement a method to compare two objects.
  - Syntax: `public class myObjComp implements Comparator<MyObj>`

- Required method:

```
public int compare(MyObj x1, MyObj x2)
```

  - Returns:
    - A positive number if x1 >= x2
    - A negative number if x1 <= x2
    - 0 if x1 == x2

Comparator Interface

```
public class myObjComp implements Comparator<MyObj> {
    public int compare(MyObj x1, MyObj x2) {
        return x1.field – x2.field;
    }
}
```

- If this x1’s field was 7 and x2’s field was 5, the result is 2
  - Positive, so x1 is “greater than” x2
- If this x1’s field was 4 and x2’s field was 8, the result is -4
  - Negative, so x1 is “less than” x2
- If this x1’s field was 6 and x’2s field was 6, the result is 0
  - Zero, so x1 is “equal to” x2

Comparator Interface

- We can have many Comparators that handle different comparisons:

```
public class myObjComp implements Comparator<MyObj> {
    public int compare(MyObj x1, MyObj x2) {
        return x1.field – x2.field;
    }
}

public class myObjComp2 implements Comparator<MyObj> {
    public int compare(MyObj x1, MyObj x2) {
        return x1.field2 – x2.field2;
    }
}
```

Comparator Interface

- When calling a method like Arrays.sort() we will need to provide it both the array to sort and a comparator to tell the method how to compare/sort the objects.

```
Arrays.sort(myobjs, new MyObjComp());
Arrays.sort(myobjs, new MyObjComp2());
```

- The first sort would use the MyObjComp comparator.
- The second sort would use the MyObjComp2 comparator.

Using Recursion to perform a Bubble Sort

- You’ve seen the bubble sort (and its variants) implemented using an iterative algorithm.

- Since any problem that can be solved iteratively can be solved recursively, we can design a recursive replacement for the iterative bubble sort.

Using Recursion to perform a Bubble Sort

- For an ascending sort, the first pass will move the largest value to the end of the array (length - 1).
  - The next pass will move the second largest value to index length - 2.
  - The next pass will move the third largest value to index length - 3.
  - And so on…

- The last pass of the iterative algorithm sorts for index 0.
  - At this point, the smallest value is guaranteed to already be in index 0.

Using Recursion to perform a Bubble Sort

- The base case is when the algorithm sorts for index 0.
  - An array with a length of 1
  - An array of length 1 implies the value at index 0 is already in the correct position (because it is the only value).

- The recursive case is for sorting an array with a length > 1.

Bubble Sort (Recursive Algorithm)

```
void bubbleSort(int a[], int length) {
    if(length == 1) {
        return;
    }

    for(int i = 0; i < length-1; i++) {
        if(a[i] > a[i+1]) {
            int temp = a[i+1];
            a[i+1] = a[i];
            a[i] = temp;
        }
    }

    bubbleSort(a, length-1);
}
```

Calls the method again, but one index less than the index we just sorted for (Similar to what we did for Bubble Sort Improved 2)

Using Recursion to perform a Bubble Sort

- This recursive bubble sort algorithm, like the iterative version, also performs in polynomial time.
- Let's see if there is any difference in the number of comparisons made between a recursive bubble sort and the iterative bubble sorts.

Using Recursion to perform a Bubble Sort

- First, how many times will the function call itself?
  - The base case is reached when length = 1
  - Each recursive call subtracts one from the length
  - This means the method will call itself *length* times

- The number of repetitions in the for loop decreases with each recursive call
  - length causes length-1 repetitions in the for loop
  - length-1 causes length-2 repetitions
  - length-2 causes length-3 repetitions
  - and so on…

Using Recursion to perform a Bubble Sort

- Let's say we have an array with a length of five
  - First call (length = 5), for loop repeats length-1 (4) times
  - Second call (length = 4), for loop repeats length-1 (3) times
  - Third call (length = 3), for loop repeats length-1 (2) times
  - Fourth call (length = 2), for loop repeats length-1 (1) time
  - Fifth call (length = 1), method returns (base case)

- Each iteration of the for loop performs one comparison
  - Total Comparisons **=** $\sum_{i=1}^{n-1} i$
  - **Same as Bubble Sort Improved 2**

Merge Sort and Quicksort

- We’ll now see a pair of comparative sorting algorithms, Merge Sort and Quicksort, that are typically implemented using recursion.
  - They can be done iteratively, but the code is much easier to read when implemented recursively.

- Both algorithms take a “divide and conquer” approach to sorting, much like the how the binary search algorithm performs its searches.

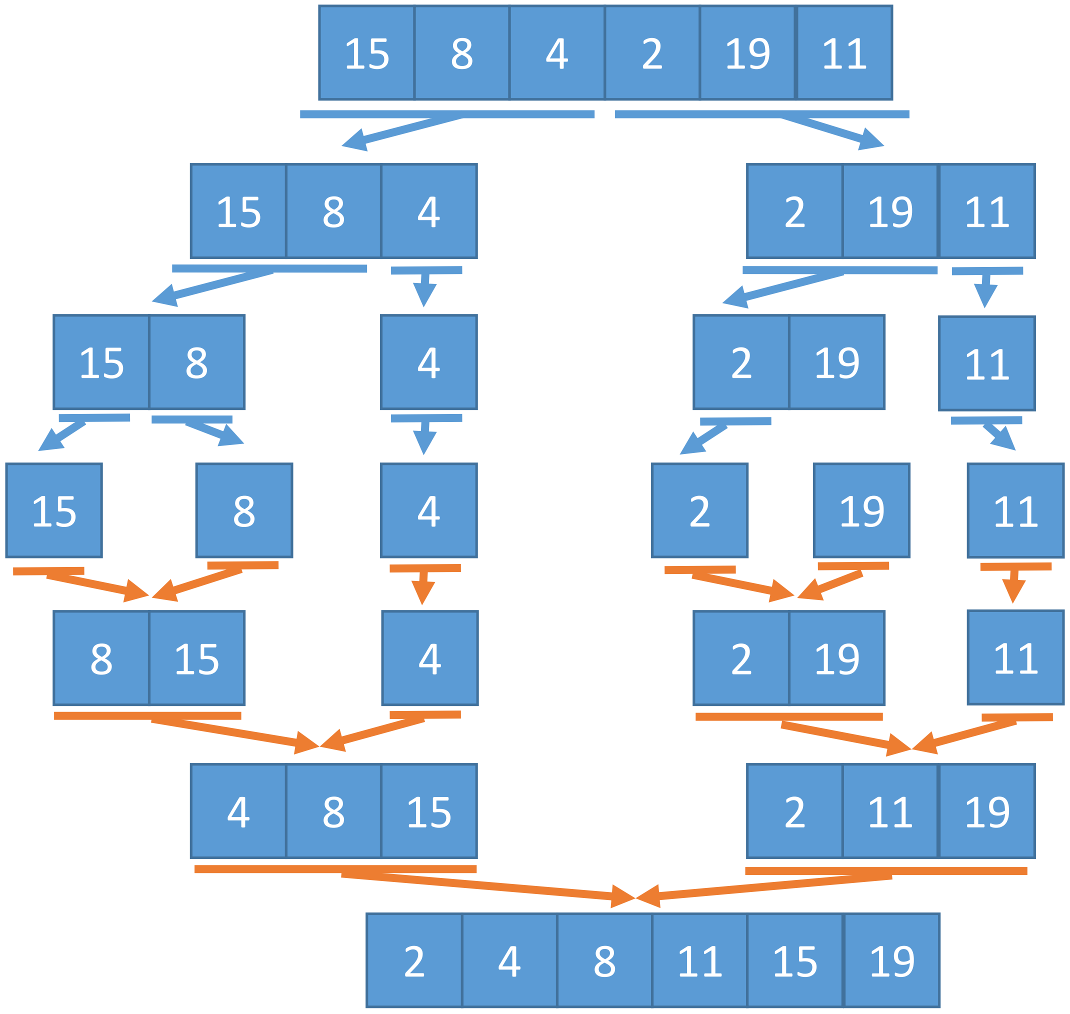
Merge Sort Algorithm

- In the Merge Sort algorithm, the array is repeatedly (recursively) split in half until it reaches halves that only contain one element.
  - At this point, the lowest depth has been reached.
- Then, working backwards, it sorts/merges the smaller arrays back together

Merge Sort Algorithm

- The image on the right gives the basic idea of how it works.
  - It divides up the array (Blue lines)
  - Then merges the array back together (Orange lines)

- Each merge involves two, sorted arrays.
  - Since the arrays to merge are in order, merging them is not computationally difficult.



Merge Sort Algorithm

- The C++ functions are a bit too long to put here in their entirety.
  - See the Sample Code provided.

- The next slides explain, *very briefly*, what is needed.
  - Two functions:
    - One function for the algorithm
    - Another function that handles the merging process

Merge Sort

mergesort(*array*, *left*, *right*):
    If *left* boundary < *right* boundary :
        Find the middle, *m*
        mergesort(*array*, *left*, *m*)
        mergesort(*array*, *m*+1, *right*)

        merge(*array*, *left*, *m*, *right*)

Merge Sort

merge(*array*, *left*, *middle*, *right*) :
    leftArray[*middle* - *left* + 1]
    rightArray[*right* - *middle*]
    Copy left side of *array* to *leftArray*
    Copy right side of *array* to *rightArray*
    Put the values of *leftArray* and *rightArray* back into *array*, in the correct order
    Put the remaining value, leftover in either *leftArray* or *rightArray*, into *array*

Quicksort

- In the Quicksort algorithm, the array is repeatedly (recursively) split into two smaller partitions, until the partitions only contain one element.
    - At this point, the lowest depth has been reached.
- The algorithm chooses a value in each partition, called the **pivot**.
    - One of the two partitions will contain any values less than the pivot.
    - The other partition will contain any values greater than the pivot.
- The process repeats recursively until partitions of length 1 are reached.
    - At which point, the array will have been sorted through the pivot processes.

Quicksort

- There are a few ways of selecting the pivot:
  - Always use the last element.
  - Always use the middle element.
  - Always use the first element.
  - Always use a randomly chosen element.
- The Sample Code provided uses the middle element as the selected pivot value.
- The next slide explains, *very briefly*, what is needed.

Quicksort

```
quicksort(array, start, end) :
        Select the pivot (the value in the middle)
        Partition the array
        quicksort(lower half)
        quicksort(upper half)
```

Merge Sort and Quicksort

- Both algorithms use the divide and conquer process like a binary search.
  - Which performs in logarithmic time.

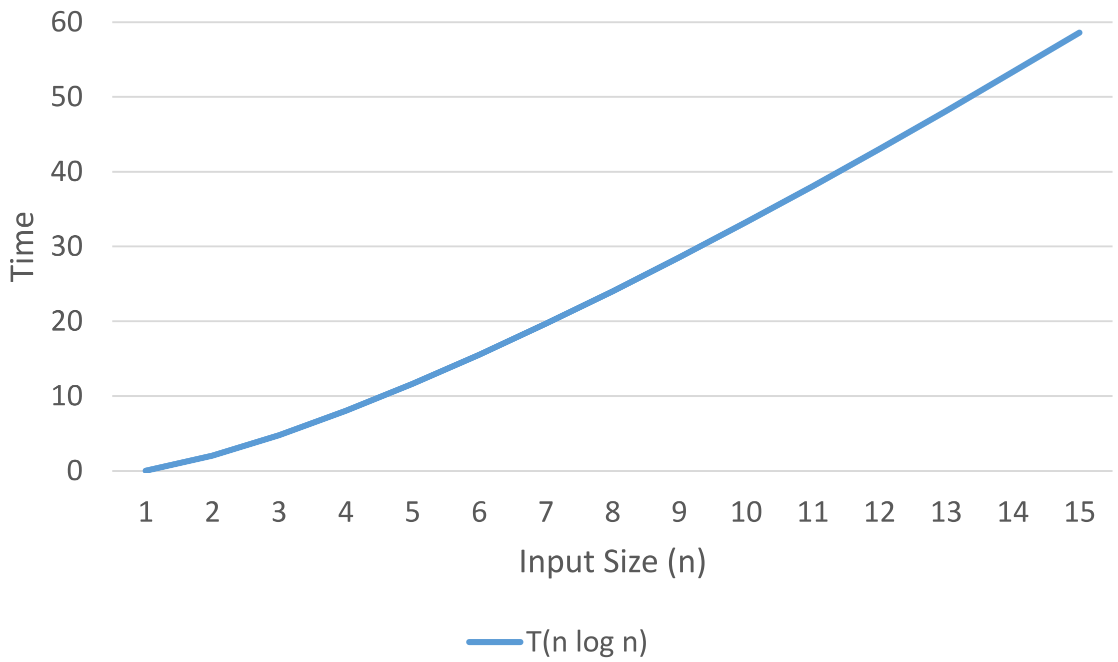
- Both algorithms, at one point or another, will have partitions with a length of 1.
  - The algorithms will perform the logarithmic divide and conquer operations for as many elements that exist in the array.

- Their time complexity is a mix of logarithmic and linear time.

Quasi-Linear Time

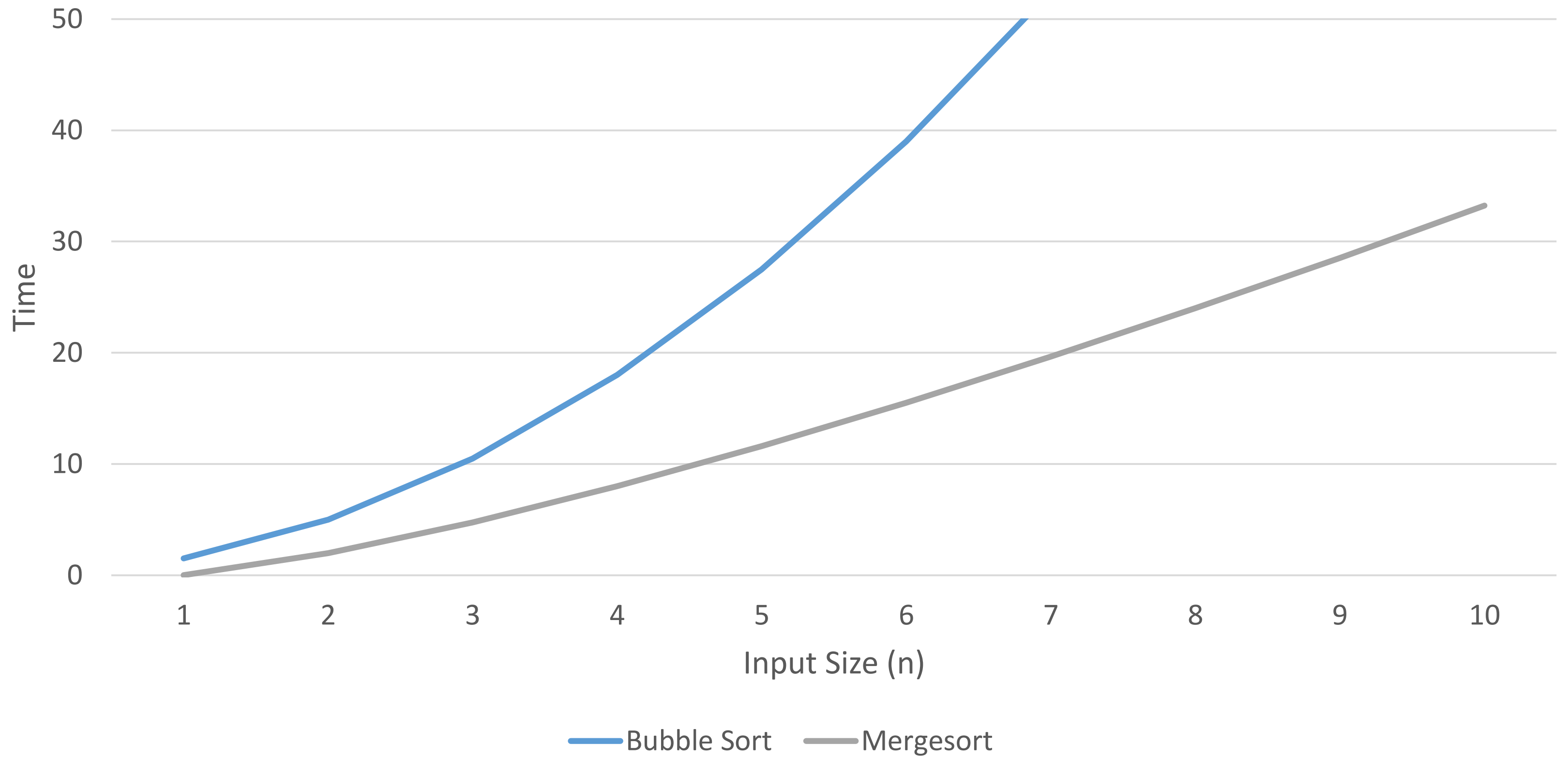
- Quasi-Linear (or Log-Linear) time is when an algorithm executes n-number of operations, where each operation performs in logarithmic time.
  - $T(n) = n \log_2 n$

- $T(n) = n \log_2 n$
  - $T(4) = 4 \log_2 4$
  - $T(8) = 8 \log_2 8$
  - $T(12) = 12 \log_2 12$



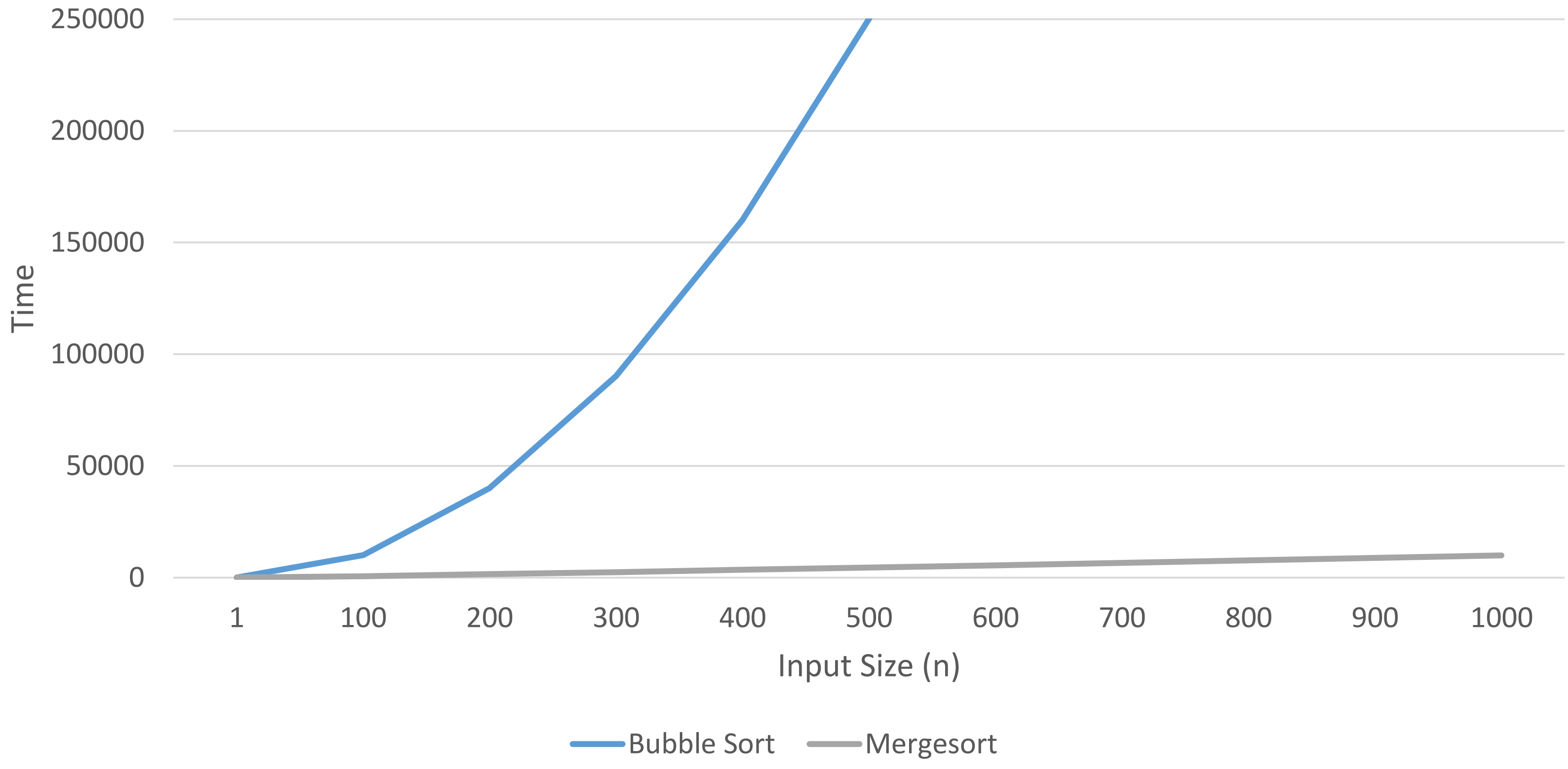
Bubble Sort vs Merge Sort

- Array length: 10

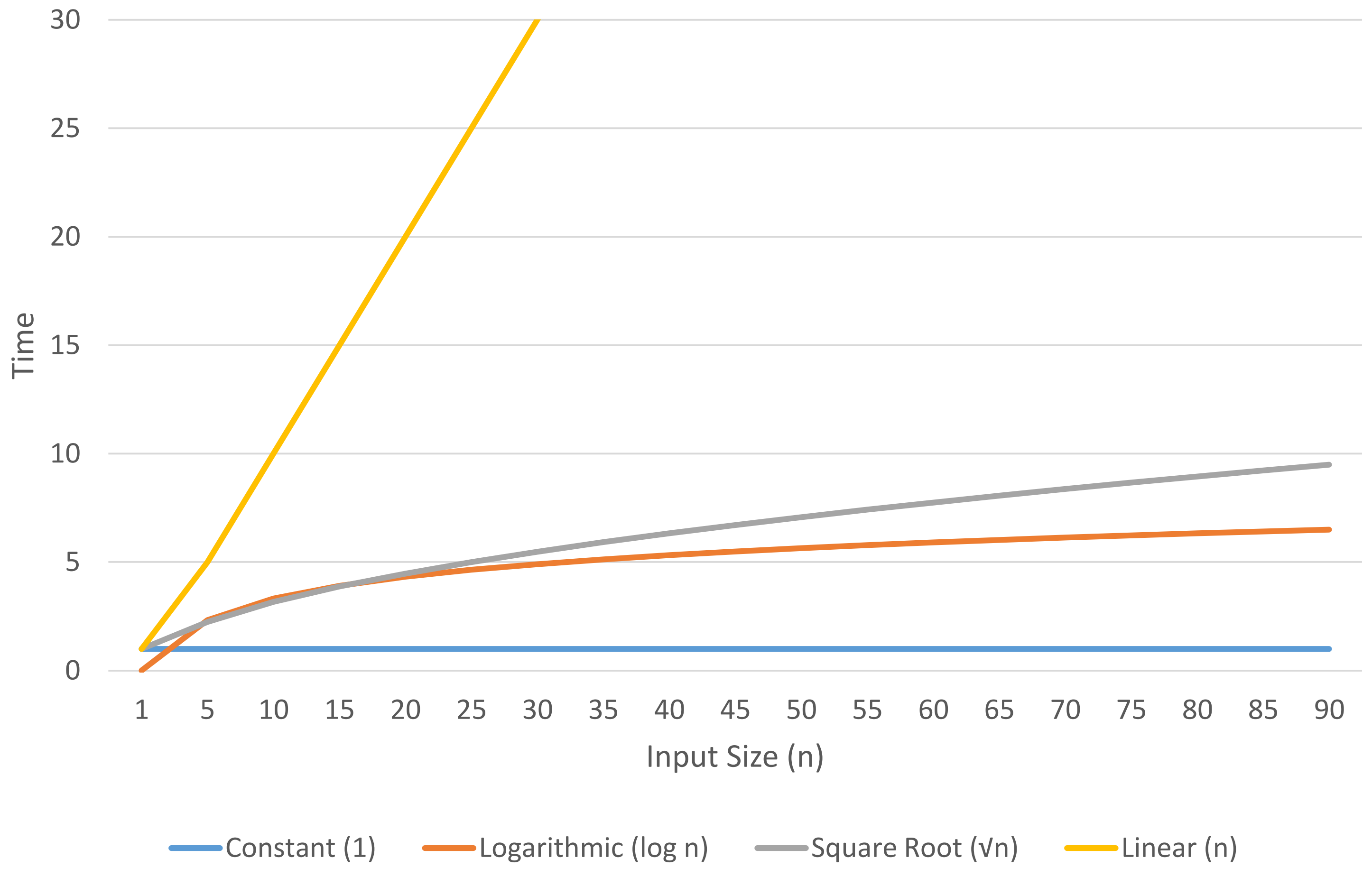


Bubble Sort vs Merge Sort

- Array length: 1000



Complexities



Complexities