Data Structures and Algorithms

Sorting Algorithms

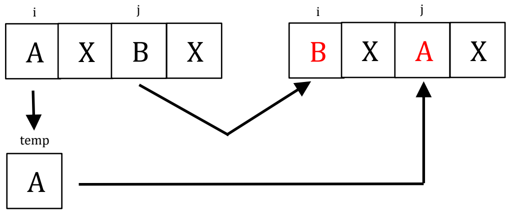
Contents

1. Bubble Sort

1.1 Swap and Swap and Swap

Here's the swap algorithm for arrays. Nothing much needed to explain here.
Note: this is very useful to remember when implementing certain comparison algorithms.

```
public void swap(int[] arr, int i, int j) {
    int temp = arr[i];
    arr[i] = arr[j];
    arr[j] = temp;
}
```



1.2 Original Bubble Sort Algorithm

```
public int[] bubblesort(int[] arr) {
    // go through r rounds where r = n - 1
    // given n = # of elements
        // for every round, go through from index 0 to index i - r - 1
            // if arr[i] > arr[i + 1], swap them
        // this pushes the largest element to the back
}
```

```
Original Array:
[9, 3, 1, 6, 4]

Round: 0
9 is bigger than 3 ! Swap!
[3, 9, 1, 6, 4]
9 is bigger than 1 ! Swap!
[3, 1, 9, 6, 4]
9 is bigger than 6 ! Swap!
[3, 1, 6, 9, 4]
9 is bigger than 4 ! Swap!
[3, 1, 6, 4, 9]
Biggest element for this round is: 9

Round: 1
3 is bigger than 1 ! Swap!
[1, 3, 6, 4, 9]
6 is bigger than 4 ! Swap!
[1, 3, 4, 6, 9]
Biggest element for this round is: 6

Round: 2
Biggest element for this round is: 4

Round: 3
Biggest element for this round is: 3

Final Array:
[1, 3, 4, 6, 9]
```

Note: What is the worst case time complexity of this algorithm?

Exercise 1:

Exercise 1.1: Implement Bubble Sort

```
public void bubblesort(int[] arr) {
    // using the algorithm shown above!
}
```

Exercise 1.2: What if the array is sorted already after a certain iteration? In the example above, at Round 2, no swaps has been made as the array is already sorted after Round 1.
Can we do better? Implement a better Bubble Sort where you can stop after there is a round where no swaps have been made

```
public void improvedbubblesort(int[] arr) {
    // Can we find ways to do better?
}
```

2. Selection Sort

2.1 Selection Sort Algorithm

```
public void selectionsort(int[] arr) {
    // go through the array
        // find the largest element
        // swap it element at the end of the array
    // go through the array iteratively, exclude the last element each time
}
```

```
Original Array:
[9, 3, 1, 6, 4]

Round 1:
Largest Element is: 9
[4, 3, 1, 6, 9]
Round 2:
Largest Element is: 6
[4, 3, 1, 6, 9]
Round 3:
Largest Element is: 4
[1, 3, 4, 6, 9]
Round 4:
Largest Element is: 3
[1, 3, 4, 6, 9]

Final Array:
[1, 3, 4, 6, 9]
```

Note: What is the worst case time complexity of this algorithm?

Exercise 2:

Exercise 2.1: Implement the Selection Sort Algorithm.

```
public void selectionsort(int[] arr) {
    // do it yourself!
}
```

3. Insertion Sort

3.1 Insertion Sort Algorithm

```
public void insertionsort(int[] arr) {
    // go through the array n - 1 times
        // every round,
        // insert the element into its relevant sorted position
}
```

```
Original Array:
[9, 3, 1, 6, 4]

To Be Inserted: 3
[3, 9, 1, 6, 4]

To Be Inserted: 1
[1, 3, 9, 6, 4]

To Be Inserted: 6
[1, 3, 6, 9, 4]

To Be Inserted: 4
[1, 3, 4, 6, 9]


Final Array:
[1, 3, 4, 6, 9]
```

Note: What is the worst case time complexity of this algorithm?

Exercise 3:

Exercise 3.1: Implement the Insertion Sort algorithm

```
public void insertionsort(int[] arr) {
    // do it yourself!
}
```

4. Merge Sort

4.1 Divide and Conquer

```
public void mergesort(int[] arr, int i, int j) {
    if (i < j) { // end-condition, means cannot be divided further
        int mid = (i + j) / 2; // divide into two parts
        mergesort(arr, i, mid); // keep dividing
        mergesort(arr, mid + 1, j); // keep dividing
        merge(arr, i, mid, j); // merge the two halves
    }
}
```

4.2 How to merge arrays

```
public void merge(int[] arr, int left, int mid, int right) {
    // Idea: to merge the two sorted subarrays arr[left...mid] and arr[mid+1....right]
    // For example:
    L: [1,5,8] R: [2,4,7] temp: []
    L: [5,8] R: [2,4,7] temp: [1]
    L: [5,8] R: [4,7] temp: [1,2]
    L: [5,8] R: [7] temp: [1,2,4]
    L: [8] R: [7] temp: [1,2,4,5]
    L: [8] R: [] temp: [1,2,4,5,7]
    L: [] R: [] temp: [1,2,4,5,7,8]
    // Note: what happens if there are still elements remaining in the sub-arrays?
    // Note: remember to copy the temp array back into the main array
}
```

Note: What is the worst case time complexity of this algorithm?

Exercise 4:

Exercise 4.1: Implement the merge algorithm

```
public void merge(int[] arr, int left, int mid, int right) {
    // do it yourself!
}
```

5. Quick Sort

5.1 Divide and Divide and Divide and ....

```
public void quicksort(int[] arr, int i, int j) {
    if (i < j) { // end-condition. if i >= j, means the array has 1 or 0 elements
        int pivot = partition(arr, i, j); // split the array into 3 parts
        // left: elements < pivot
        // middle: pivot
        // right: elements >= pivot
        quicksort(arr, i, pivot - 1); // divide
        quicksort(arr, i, pivot + 1); // divide
    }
}
```

5.2 Partition it!

```
public int partition(int[] arr, int i, int j) {
    // take the first element to be the pivot
    // make a pointer for the pivot
    // push all elements smaller than the pivot to the left by swapping
    // finally put the pivot in its proper position
}
```

For example:

```
Original Array:
[3, 7, 8, 2, 5, 0]

Pivot: 3
Found smaller element: 2
[3, 2, 8, 7, 5, 0]
Found smaller element: 0
[3, 2, 0, 7, 5, 8]
Pivot to be put in proper position:
[0, 2, 3, 7, 5, 8]
```

Exercise 5:

Exercise 5.1: Implement the partition algorithm

```
public int partition(int[] arr, int i, int j) {
    // do it yourself!
}
```

6. Radix Sort

6.1 Idea behind Radix Sort

Sort by the ones, tens, hundreds digit... until all digits of each element has been accounted for

```
public void radixsort(int[] arr) {
    int m = max(arr); // find the largest element in the array
    for (int e = 1; (m/e) > 0; e *= 10) { // arranges elements based particular digit
        arrange(arr, arr.length, e); // arrange
    }
}
```

```
Original Array: [30, 27, 111, 92, 356, 7, 8, 10]
Group by Ones: [30, 10] [111] [92] [356] [27, 7] [8]

New Array: [30, 10, 111, 92, 356, 27, 7, 8]
Group by Tens: [07, 08] [10, 111] [27] [30] [356] [92]

New Array: [7, 8, 10, 111, 27, 30, 356, 92]
Group by Hundreds: [007, 008, 010, 027, 030, 092] [111] [356]

New Array: [7, 8, 10, 27, 30, 92, 111, 356]
```

6.2 Sample Algorithm

```
public void arrange(int[] arr, int len, int e) {
    int[] output = new int[len]; // initialize output array
    int[] count = new int[10]; // intialize digits from 0 to 9
    for (int a = 0; a < 10; a ++) {
        count[a] = 0; // count = 0 for digits 0 to 9
    }

    for (int i = 0; i < len; i ++) {
        int digit = (arr[i] / e) % 10;
        count[digit] ++; // add count based on the digit value
    }

    for (int j = 1; j < 10; j ++) {
        count[j] += count[j - 1]; // iteratively add the counts to get the indexes
    }

    for (int k = len - 1; k >= 0; k --) { // backwards for-loop
        int digit = (arr[k]/e) % 10; // find the digit of the element
        int ind = count[digit] - 1; // get the index from the count array
        output[ind] = arr[k]; // place the element into the output array
        count[digit] --; // remember to minus one from the count
    }

    for (int l = 0; l < len; l ++) {
        arr[l] = output[l]; // copy output array to main array
    }
}
```

7. Sorting Groups

7.1 Stable Sorting

If a sorting algorithm is stable if the relative order of elements with the same key value is preserved by the sorting algorithm used.

2	1	3	1	Stable Sort →	1	1	2	3

Note: Quick Sort and Selection Sort are not stable. Find out why!

So how do we make these sorts stable?
We do **NOT SWAP,** instead, we **SHIFt** these elements

```
public void swap(int[] arr, int i, int j) {
    int temp = arr[i];
    arr[i] = arr[j];
    arr[j] = temp;
}

public void shift(int[] arr, int i, int j) {
    int temp = arr[i];
    for (int t = i + 1; t < j; t ++) {
        arr[i] = arr[i + 1];
    }
    arr[j] = temp;
}
```

7.2 In-Place Sorting

If a sorting algorithm is in-place, no extra space is required to sort the elements.
Note: Radix Sort and Merge Sort are not stable. Find out why!

8. Sorting Algorithms Summary

Sorting Algorithm	Worst Time Complexity	Best Time Complexity	Worst Space Complexity	In-place	Stable
Selection	$O(n^2)$	$O(n^2)$	$O(1)$	YES	NO
Insertion	$O(n^2)$	$O(n)$	$O(1)$	YES	YES
Bubble	$O(n^2)$	$O(n^2)$	$O(1)$	YES	YES
Merge	$O(n^2)$	$O(n \log n)$	$O(n)$	NO	YES
Quick	$O(n^2)$	$O(n \log n)$	$O(\log n)$	YES	NO
Radix	$O(nd)$	$O(n)$	$O(n+d)$	NO	YES

Note: d represents number of digits

9. Sorting and Comparators

9.1 Comparators and Comparable<T>

How to implement: (a compare b)
1) a is to have higher priority than b, return a negative number;
2) a is to have equal priority than b, return a 0;
3) a is to have lower priority than b, return a positive number;

```
class MyComparator implements Comparator {
    public int compare(MyClass a, MyClass b) {
        // implement it here
    }
}

class MyClass implements Comparable<MyClass> {
    public int compareTo(MyClass b) {
        // implement it here
    }
}
```

9.2 Sorts

https://docs.oracle.com/javase/7/docs/api/java/util/Arrays.html
https://docs.oracle.com/javase/7/docs/api/java/util/Collections.html

*Note: If your Elements already implements Comparable<T>, you do not need a Comparator Otherwise, you **NEED** a Comparator.*

```
class MyClass {
    public int compareTo(MyClass b) {
        // implement it here
    }
}

class MyComparator implements Comparator {
    public int compare(MyClass a, MyClass b) {
        // implement it here
    }
}

MyClass[] myArray;
ArrayList<MyClass> myArrayiList;

Arrays.sort(myArray); // wrong!
Arrays.sort(myArray, new MyComparator());
Collections.sort(myArrayList); // wrong!
Collections.sort(myArrayList, new MyComparator());
```

Suggestion Solutions

Exercise 1:

```
public void bubblesort(int[] arr) {
    for (int r = 0; r < arr.length - 1; r ++) {
        for (int i = 0; i < arr.length - r - 1; i ++) {
            if (arr[i] > arr[i+1]) {
                swap(arr, i, i + 1);
            }
        }
    }
}
```

```
public void improvedbubblesort(int[] arr) {
    for (int r = 0; r < arr.length - 1; r ++) {
        boolean isSorted = true;
        for (int i = 0; i < arr.length - r - 1; i ++) {
            if (arr[i] > arr[i+1]) {
                swap(arr, i, i + 1);
                isSorted = false;
            }
        }

        if (isSorted) {
            return; // end if the array is already sorted
        }
    }
}
```

Exercise 2:

```
public void selectionsort(int[] arr) {
    for (int i = arr.length - 1; i >= 1; i --) {
        int maxInd = i;
        for (int j = 0; j < i; j ++) {
            if (arr[j] > arr[maxInd]) {
                maxInd = j;
            }
        }
        swap(arr, i, maxInd);
    }
}
```

Exercise 3:

```
public void insertionsort(int[] arr) {
    for (int i = 1; i < arr.length; i ++) {
        int toBeInserted = arr[i];
        int j;
        for (j = i - 1; j >= 0 && arr[j] > toBeInserted; j --) {
            arr[j + 1] = arr[j];
        }
        arr[j + 1] = toBeInserted;
    }
}
```

Exercise 4:

```java
public void merge(int[] arr, int left, int mid, int right) {
    int[] temp = int[right-left+1];
    int l = left;
    int r = mid + 1;
    int i = 0;
    while (l <= mid && r <= right) {
        if (arr[l] < arr[r]) {
            temp[i++] = arr[l++];
        } else {
            temp[i++] = arr[r++];
        }
    }

    while (l <= mid) {
        temp[i++] = arr[l++];
    }

    while (r <= right) {
        temp[i++] = arr[r++];
    }

    for (int j = 0; j < right-left+1; j ++) {
        arr[left+j] = temp[j];
    }
}
```

Exercise 5:

```
public int partition(int[] arr, int i, int j) {
    int pivot = arr[i];
    int pos = i;
    for (int k = i + 1; k <= j; k ++) {
        if (arr[k] < pivot) {
            pos ++;
            swap(arr, k, pos);
        }
    }
    swap(arr, i, pos);
    return pos;
}
```