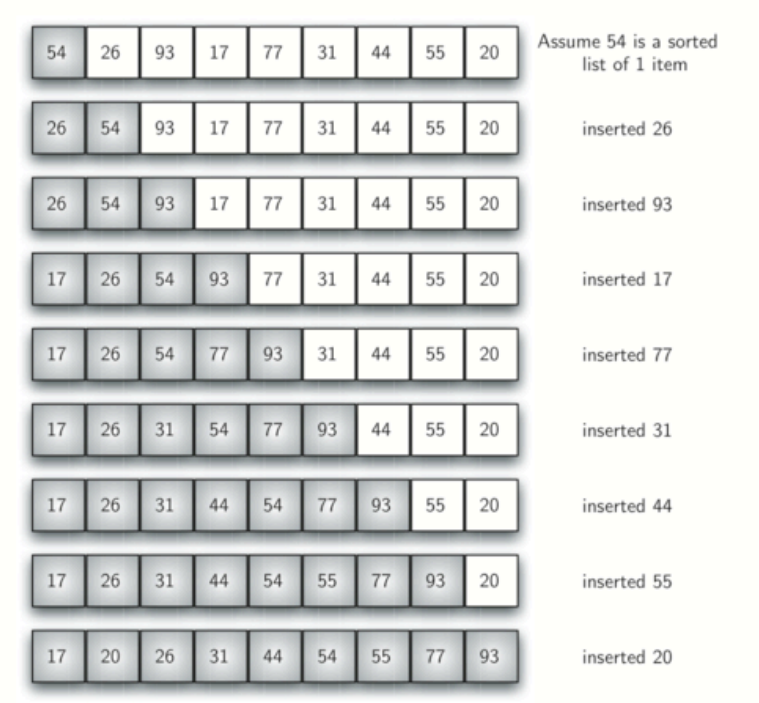
CSCI-SHU 210 Data Structures

Homework Assignment9 Sorting Algorithms

- **You can use Gradescope to check your output's correctness, but you still have to use the required algorithms to get full credit.**
- **Please also provide a .txt file to answer non-coding questions.**
- **For runtime complexity, please give the tightest bound for Big O.**

1. Insertion sort

To warm up, let us start with `insertion sort`.



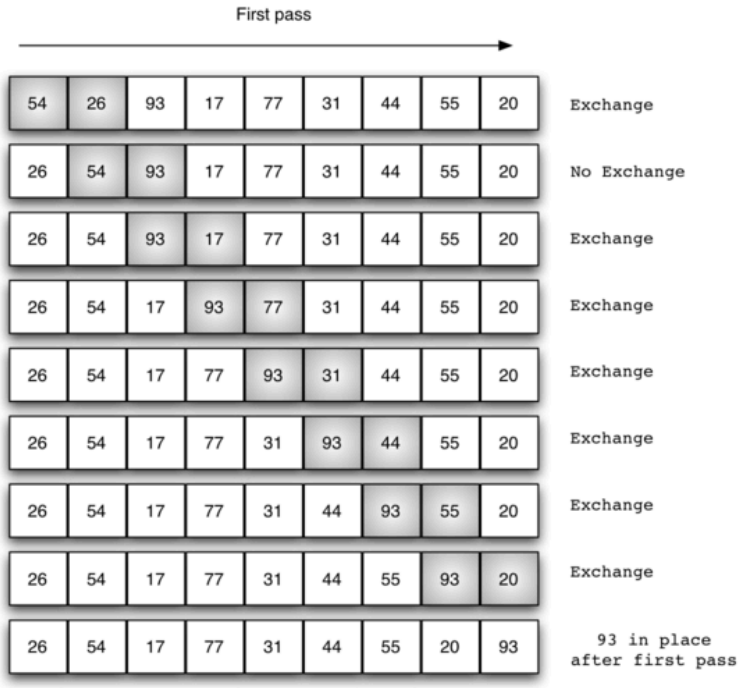
Question 1.1. Code your implementation of `insertion sort`.

Question 1.2. What is the best-case runtime complexity of `insertion sort`? What does the best-case input look like? Answer this in .txt file.

Question 1.3. What is the worst-case runtime complexity of `insertion sort`? What does the worst-case input look like? Answer this in .txt file.

2. Bubble sort

Next, let's implement `bubble sort.`

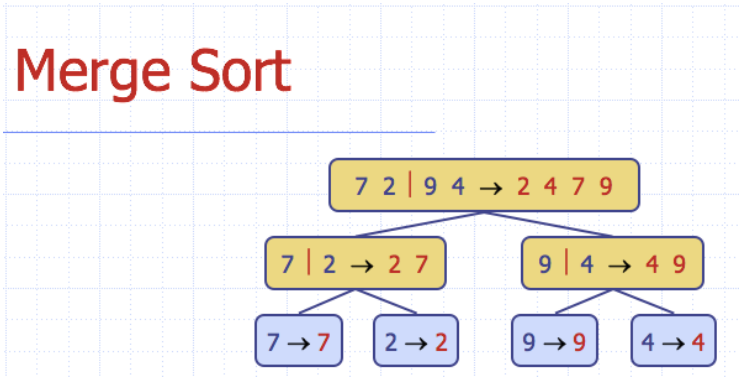


Question 2.1. Code your implementation of `bubble sort`.

Question 2.2. What is the worst-case runtime complexity of `bubble sort`? Answer this in the .txt file.

3. Merge sort

Another sorting algorithm required to know is the `merge sort.`



Question 3.1. Code your implementation of `merge sort`.

Question 3.2. What is the worst-case runtime complexity of `merge sort`? Answer this in the .txt file.

4. Summer sort

Design a sorting algorithm, called the "`SummerSort`", that uses only the two following operations:

- `findMax(array, a, b)` returns the index of the largest value in the array between index positions a and b
- `reverse(array, a, b)` reverses all the values in the array from index position a to index position b

Both of these operations are already available in the skeleton code.

Question 4.1. Code your implementation of `SummerSort`.

Question 4.2. What is the best-case runtime complexity of `SummerSort`? What does the best-case input look like? Answer this in .txt file.

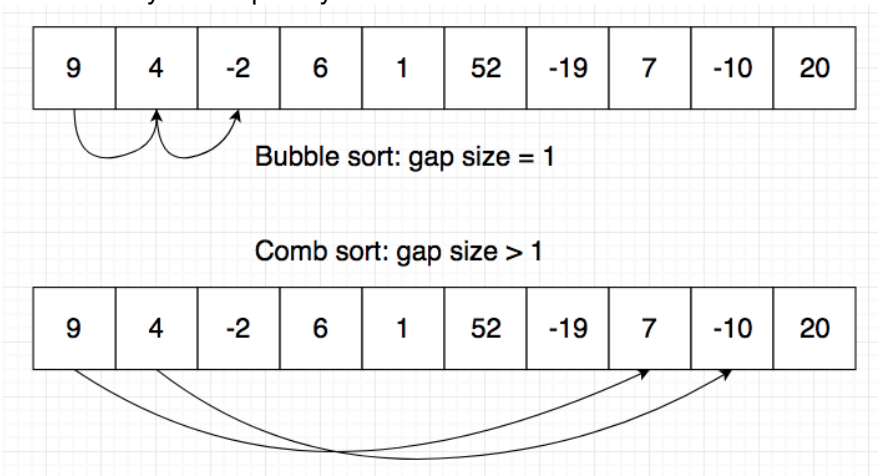
Question 4.3. What is the worst-case runtime complexity of `SummerSort`? What does the worst-case input look like? Answer this in .txt file.

5. Comb sort (Improved bubble sort)

`Comb sort` is mainly an improvement over Bubble Sort. Bubble sort always compares adjacent values. So, all inversions are removed one by one. `Comb sort` improves on Bubble Sort by using gap of size more than 1. The gap starts with a large value and shrinks by a factor of 1.3 in every iteration until it reaches the value 1. Thus, `Comb sort` removes more than one inversion counts with one swap and performs better than Bubble Sort.

The shrink factor has been empirically found to be 1.3 (by testing `Comb sort` on over 200,000 random lists)

- Bubble sort uses gap size = 1, which means always compare adjacent elements.
- `Comb sort`, start with gap size len(array) // 1.3, swap elements that are gap size away, then use the next gap size.
- The larger gap allows `Comb sort` moving elements faster than Bubble sort.
- When `Comb sort` reaches gap size 1, it will perform regular bubble sort step until the array is completely sorted.



If array size = 10, then
Gap size 1st = 10 // 1.3 = 7 Gap size 2nd = 7 // 1.3 = 5
Gap size 3rd = 5 // 1.3 = 3 Gap size 4th = 3 // 1.3 = 2
Gap size 5th = 2 // 1.3 = 1 then keep using gap size = 1 until list is fully sorted.

Question 5.1. Code your implementation of `Comb sort`.
Question 5.2. What is the best-case runtime complexity of `Comb sort`? Answer this in .txt file.

6. Binary radix sort

During the recitation, we have implemented the decimal radix sort. In decimal radix sort, ten LinkedQueues were used to store ten possible digits.

Now, your task is to implement binary radix sort. In binary radix sort, only two queues are used and every integer number is treated as a binary number.

To get started, I recommend understanding decimal radix sort first, then make modifications from there.

Question 6.1. Code your implementation of binary radix sort.