CSCI 1300 - Spring 2018
Instructor: Fleming/Gupta
Recitation 10
Due Saturday, March 24th, by 6:00pm

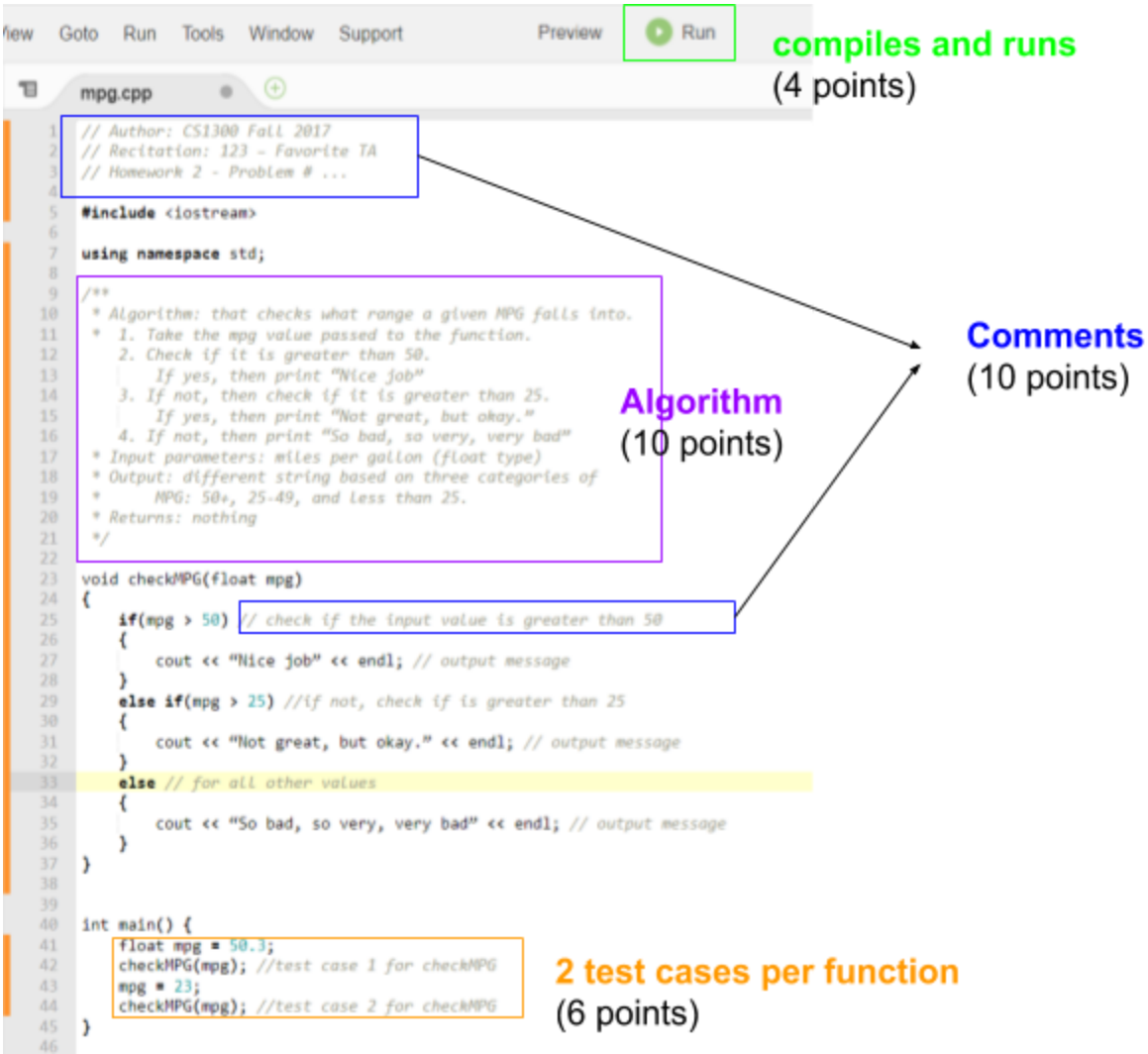
- All components (Cloud9 workspace, moodle quiz attempts, and zip file) must be completed and submitted by **Saturday, March 24th 6:00 pm** for your solution to receive points.
- Recitation attendance is required to receive credit.
- **Develop in Cloud9:** For this recitation assignment, write and test your solution using Cloud9.
- **Submission:** All three steps must be fully completed by the submission deadline for your homework to receive points. Partial submissions will not be graded.
    1. Make sure your Cloud9 workspace is shared with your TA: Your recitation TA will review your code by going to your Cloud9 workspace. TAs will check the last version that was saved before the submission deadline.
        - Create a directory called *Rec10* and place all your file(s) for this assignment in this directory.
    2. **Submit to the Moodle Autograder:** Head over to Moodle to the link *Rec 10*. You will find one programming quiz question for each problem in the assignment. You can modify your code and re-submit (press Check again) as many times as you need to, up until the assignment due date.
    3. **Submit a zip file to Moodle:** After you have completed all the questions and checked them on Moodle, you must submit a zip file with the .cpp file(s) you wrote in Cloud9. Submit this file going to *Rec 10 (File Submission)* on moodle.

Please follow the same submission guidelines first outlined in Homework 3. Here is a summary:

- 4 points for code compiling and running
- 6 points for two test cases for each function in main.
- 10 points for an algorithm description for each function
- 10 points for comments
    - required comments at top of file

      // CS1300 Spring 2018
      // Author:
      // Recitation: 123 – Favorite TA
      // Cloud9 Workspace Editor Link: https://ide.c9.io/…
      // Recitation 10 - Problem # ...

    - comments for functions / test cases



Sorting Algorithms

There are many different ways of sorting lists of numbers. Probably the simplest is the *bubble sort*, which is less efficient but easier to understand than many of the more complex sorting algorithms.

In general, the *bubble sort* algorithm works by repeatedly looping over the array, swapping adjacent elements if they are out of order. Here's a step-by-step description of the *bubble sort* algorithm:

The *bubble sort* algorithm:

1. Compare adjacent elements. If the first is greater than the second, swap them.
2. Do this for each pair of adjacent elements, starting with the first two and ending with the last two. At this point the last element should be the greatest.
3. Repeat again from the beginning for one less element each time, until there are no more pairs to compare.

The *bubble sort* algorithm got its name from the way bubbles rises to the surface:

- The ***largest* bubble** will **reach** the surface **first.**
- The ***second largest* bubble** will reach the surface **next.**
- And so on.

Similarly, in the **bubble sort** algorithm:

- The ***largest* value** will **reach** its ***correct* position first.**
- The ***second largest* bubble** will reach its ***correct* position next.**
- And so on.

For example:

- Items that are already sorted are highlighted.
- Items currently being compared are **bolded**.
- Example from *Bubble Sort* article on Wikipedia: https://en.wikipedia.org/wiki/Bubble_sort

---

Pass #1

(**5** **1** 4 2 8) → (**1** **5** 4 2 8), Here, algorithm compares the first two elements, and swaps since 5 > 1.
(1 **5** **4** 2 8) → (1 **4** **5** 2 8), Swap since 5 > 4
(1 4 **5** **2** 8) → (1 4 **2** **5** 8), Swap since 5 > 2
(1 4 2 **5** **8**) → (1 4 2 **5** **8**), Now, since these elements are already in order (8 > 5), algorithm does not swap them.

Pass #2

(**1** **4** 2 5 8) → (**1** **4** 2 5 8)
(1 **4** **2** 5 8) → (1 **2** **4** 5 8), Swap since 4 > 2
(1 2 **4** **5** 8) → (1 2 **4** **5** 8)

Now, the array is already sorted, but the algorithm does not know it is completed.

Pass #3

(**1** **2** 4 5 8) → (**1** **2** 4 5 8)
(1 **2** **4** 5 8) → (1 **2** **4** 5 8)

Pass #4

(**1** **2** 4 5 8) → (**1** **2** 4 5 8)

Done!

(1 2 4 5 8)

One of the "imperfections" of the *bubble sort* algorithm is that the "bubbles" are not treated equally; while the large bubbles are ascending rapidly, the small bubbles descend only one position per iteration.

The *cocktail sort* algorithm (also known as shaker sort) mitigates this problem. *Cocktail sort* is essentially the same as *bubble sort* but instead of starting at the beginning of the array, it alternates between going from left-to-right to right-to-left on each pass of the array. Hence, every iteration of the algorithm consists of two phases. In the first one the largest bubble ascends to the end of the array, in the second phase the smallest bubble descends to the beginning of the array.

The *cocktail sort* algorithm:

1. **Forward pass**: Starting at the beginning of the array, compare adjacent elements. If the first is greater than the second, swap them. By the end of this pass, the largest element should be at the end of the array (as it was in bubble sort).
2. **Backwards pass:** Start at the opposite end of the array at the element just before the item that was sorted in the forward pass. Compare adjacent elements. If the first

is greater than the second, swap them. By the end of this pass, the smallest element should be at the beginning of the array.
3. Repeat 1 and 2 again, moving the start and end positions to account for the newly sorted items in each pass.

For example:

- Items that are already sorted are highlighted.
- Items currently being compared are **bolded**.

Pass #1

Forward Pass

(**5** **1** 4 2 8) → (**1** **5** 4 2 8), Here, algorithm compares the first two elements, and swaps since 5 > 1.
(1 **5** **4** 2 8) → (1 **4** **5** 2 8), Swap since 5 > 4
(1 4 **5** **2** 8) → (1 4 **2** **5** 8), Swap since 5 > 2
(1 4 2 **5** **8**) → (1 4 2 **5** **8**)

The largest element (8) is at the end of the array at the end of the forward pass.

Backward Pass (Start at **5**, since 8 is already sorted from the forward pass.)

(1 4 **2** **5** 8) → (1 4 **2** **5** 8), Here, the elements are already in order (2 > 5), so the algorithm does not swap them
(1 **4** **2** 5 8) → (1 **2** **4** 5 8), Swap since 4 > 2
(**1** **2** 4 5 8) → (**1** **2** 4 5 8)

The smallest element (1) is at the beginning of the array at the end of the backward pass.

Pass #2

Forward Pass (Start at **2**, since 1 is already sorted from the backward pass.)

(1 **2** **4** 5 8) → (1 **2** **4** 5 8)
(1 2 **4** **5** 8) → (1 2 **4** **5** 8)

Backward Pass (Start at **4**, since 5 is already sorted from the forward pass)

(1 **2** **4** 5 8) → (1 **2** **4** 5 8)

Done!

(1 2 4 5 8)

Optimizing Bubble Sort with a Flag:

You might have noticed for bubble sort and for cocktail sort that there were passes at the end where nothing changed.

In fact, even if we had a *perfectly sorted* array like (1 2 3 4 6), the algorithm would still do 10 comparisons!

If we go through an *entire pass* with *no elements being swapped*, we know the array is already fully sorted. One common optimization for bubble sort is to add a boolean *flag* that can be used to check if the array has already been sorted. We can then use this boolean flag to terminate early.

The *bubble sort* algorithm with a flag:

1. Set the flag to true (assume array is sorted).
2. Compare adjacent elements. If the first is greater than the second, swap them and set the flag to false (array is not fully sorted).
3. Do this for each pair of adjacent elements, starting with the first two and ending with the last two. At this point the last element should be the greatest.
4. If the flag is true, the array is sorted, so stop. Otherwise, go back to step 1 and repeat for one less element each time, until there are no more pairs to compare.

The *cocktail sort* algorithm with a flag:

1. Set the flag to true (assume array is sorted).
2. **Forward pass**: Starting at the beginning of the array, compare adjacent elements. If the first is greater than the second, swap them and set the flag to false. By the end of this pass, the largest element should be at the end of the array (as it was in bubble sort).
3. If the flag is true, the array is sorted, so stop. Otherwise continue to step 4.
4. Set the flag to true (assume array is sorted).
5. **Backwards pass:** Start at the opposite end of the array at the element just before the item that was sorted in the forward pass. Compare adjacent elements. If the first is greater than the second, swap them and set the flag to false. By the end of this pass, the smallest element should be at the beginning of the array.

6. If the flag is true, the array is sorted, so stop. Otherwise continue to step 7.
7. Go back to step 1, moving the start and end positions to account for the newly sorted items in each pass.

Here's the bubble sort example from earlier with a flag named *sorted*.

Pass #1

sorted = true

(**5** **1** 4 2 8) → (**1** **5** 4 2 8), Swap since 5 > 1. sorted = false (since a swap occurred)
(1 **5** **4** 2 8) → (1 **4** **5** 2 8), Swap since 5 > 4. sorted = false
(1 4 **5** **2** 8) → (1 4 **2** **5** 8), Swap since 5 > 2. sorted = false.
(1 4 2 **5** **8**) → (1 4 2 **5** **8**)

sorted == false, so we don't know if the array is fully sorted and have to another pass

---

Pass #2

sorted = true

(**1** **4** 2 5 8) → (**1** **4** 2 5 8)
(1 **4** **2** 5 8) → (1 **2** **4** 5 8), Swap since 4 > 2. sorted = false
(1 2 **4** **5** 8) → (1 2 **4** **5** 8)

sorted == false, so we don't know if the array is fully sorted and have to another pass

---

Pass #3

sorted = true

(**1** **2** 4 5 8) → (**1** **2** 4 5 8)
(1 **2** **4** 5 8) → (1 **2** **4** 5 8)

sorted == true, so we know the array is fully sorted

Done!

(1 2 4 5 8)

What is given:
An array containing 20 positive integer numbers, unsorted.

Problem 1

Write a function **bubbleSort** that takes an array of integers and the number of elements in the array. Your function should sort the array using bubble sort and print the number of comparisons and swaps (a "swap" is whenever two numbers change places).

You should implement the *bubble sort* algorithm with a flag.

Note: You might also want to write and use a function that swaps two values.

Your function should not return anything but should display the sorted array and the number of swaps executed. Your output should look like:

Bubble Sort:
1 1 2 3 3 3 3 4 4 5 6 7 7 7 8 9 10 13 15 17
Number of comparisons: <comparisons>
Number of swaps: <swaps>

Problem 2

Write a function **cocktailSort** that takes an array of integers and the number of elements in the array. Your function should sort the array using cocktail sort and print the number of comparisons and swaps (a "swap" is whenever two numbers change places).

You should implement the *cocktail sort* algorithm with a flag.

Note: You might also want to write and use a function that swaps two values.

Your function should not return anything but should display the sorted array and the number of swaps executed. Your output should look like:

Cocktail Sort:
1 1 2 3 3 3 3 4 4 5 6 7 7 7 8 9 10 13 15 17
Number of comparisons: <comparisons>
Number of swaps: <swaps>