These problems will help you understand the Basic data structure (queues, stacks, …etc.) and Quick-Union. This Homework should prepare you for Project as well.

Submitting Information:

- DO NOT post your code on Piazza
- Use the code I provided for each problem. DO NOT delete any function
- Submit your work on Canvas.
- DO NOT change the name of .java files.
- Put all the .java files in one zip file and name it <last_Name><first_initial>.zip for example ChandakA.zip for Name Aniket Chandak
- For theory answer, you are allowed to write your answer in word or handwritten on paper. You create pdf for your word/scanned copy of handwritten answer and put in the zip file
- The deadline is Tuesday, Feb 17th at 11:55PM
- Follow the guidelines in homework rubric

Problem 1. Give a counterexample that shows why this intuitive implementation of `union()` for quick-find is not correct:

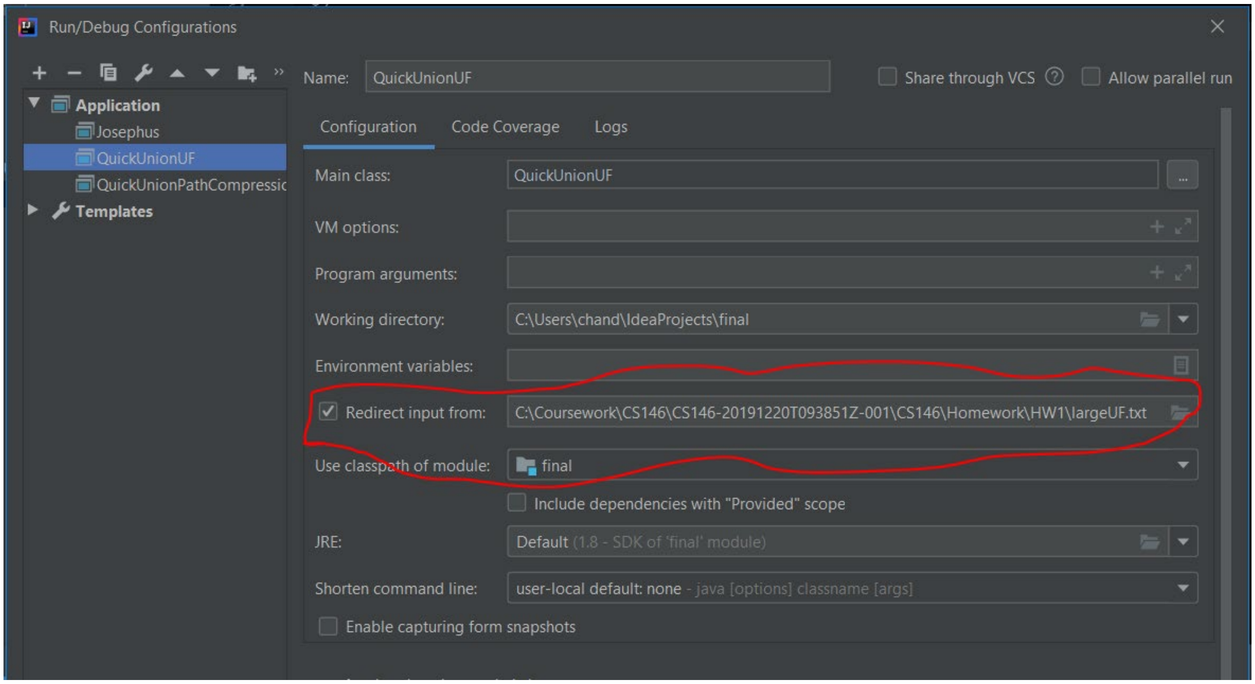
```
public void union(int p, int q) {
   if (connected(p, q)) return;
   for (int i = 0; i < id.length; i++)
      if (id[i] == id[p]) id[i] = id[q];
   count--;
}
```

Problem 2: Consider for union function, p is part of smaller tree and q is part of bigger tree. As per the weighted quick-union, we change the id[root(p)] to id[root(q)] to make a union. If we change id[root(p)] to q instead of id[root(q)], would the resulting algorithm be still correct? Give reason for your answer. Also are there any performance consequences?

Problem 3: Two skeleton code are given for Quick-Union implementation, QuickUnionUF.java and QuickUnionPathCompressionUF.java. Add the missing code for functions in both programs. QuickUnionUF.java is normal implementation with no improvement where as the other code should have improvement for path compression (Two pass). Three different inputs are provided to run a program smallUF.txt, mediumUF.txt and LargeUF.txt You can test your implementation using any of the given input file.
Following command is example to run a code:

```
$ java QuickUnionUF < smallUF.txt
```

If you are using IntelliJ, you need to edit the configuration with input file before running program.
Below is screenshot for my configuration:



Problem 4: Compare the performance of the codes from Problem 3 for each input size. Give your analysis. [which one is faster and what is the reason for faster performance.]

Problem 5. *(Josephus.java)* In the Josephus problem from antiquity, N people are in dire straits and agree to the following strategy to reduce the population. They arrange themselves in a circle (at positions numbered from 0 to N −1) and proceed around the circle, eliminating every Mth person until only one person is left. Legend has it that Josephus figured out where to sit to avoid being eliminated. Write a Queue client Josephus.java that takes N and M from the command line and prints out the order in which people are eliminated (and thus would show Josephus where to sit in the circle).

```
$ java Josephus 7 2
1 3 5 0 4 2 6
$ java Josephus 20 3
2 5 8 11 14 17 0 4 9 13 18 3 10 16 6 15 7 1 12 19
```

Problem 6. *(KthString.java)* Write a Queue client KthString.java that takes a command-line argument k and prints the kth string from the end found on standard input, assuming that standard input has k or more strings

```
$ java KthString 9
it was the best of times it was the worst of times best
<ctr-d>
best
```

Problem 7. *(Parantheses.java)* Implement the static method match() in Parentheses.java that takes a string s as argument and uses a stack to determine whether its parentheses are properly balanced, and returns true if they are and false otherwise. You may assume that s only consists of parentheses (curly, square, and round).

```
$ java Parentheses
[()]{}{[()()]()}
true
$ java Parentheses
[(])
<ctr-d>
false
```