CSE115 / CSE503
Introduction to Computer Science I

Dr. Carl Alphonce

343 Davis Hall

alphonce@buffalo.edu

Office hours:

Tuesday 10:00 AM – 12:00 PM*

Wednesday 4:00 PM – 5:00 PM

Friday 11:00 AM – 12:00 PM

OR request appointment via e-mail

**Tuesday adjustments: 11:00 AM – 1:00 PM on 12/6*

ROADMAP

Last time

- exercise 08 solution
- exercise 09

Today

- exercise 09 solution
- exercise 10
- linear search

Coming up

- exercise 10 solution
- exercise 11 (?)
- binary search

ANNOUNCEMENTS

FINAL EXAM NOTICE

CHECK YOUR ENTIRE FINAL EXAM SCHEDULE!

http://blogs.advising.buffalo.edu/beadvised/posts/have-you-checked-your-final-exam-schedule-4/

Room assignments will be announced at a later date.

Lab 11

Due 9:00 PM on last day of classes for everyone.

This week – recitations are office hours.

EXERCISE 09
solution

EXERCISE 09

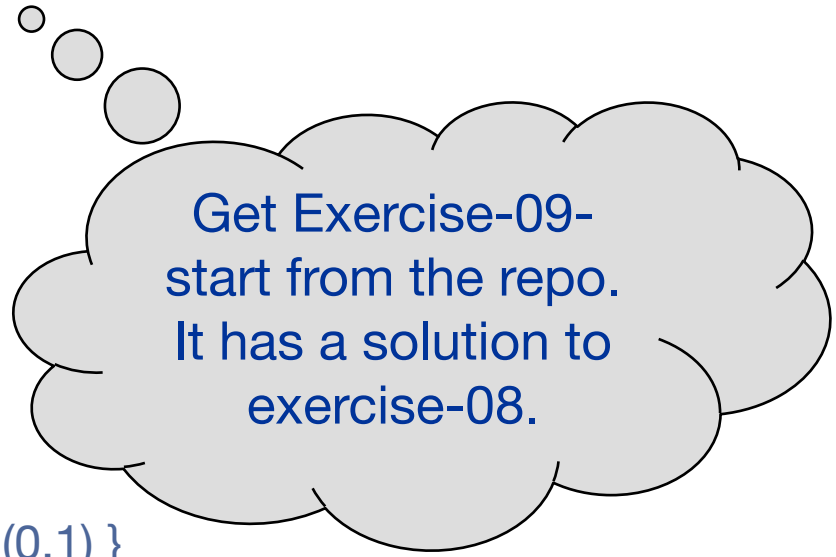
Define a method a method with this header in a class named quiz.Question:

```
public HashSet<Point> maximalMatchedRegion(Point p, ArrayList<ArrayList<String>> board)
```

Define the method so that it returns a HashSet containing those points of the largest contiguous region matching position p.

Consider this board

```
aabbc
abbcd
ddddd
efffd
```



Answer for null is { }

Answer for (0,0) is { (0,0) , (1,0) , (0,1) }

Answer for (0,3) is { (0,2) , (0,3) , (1,1) , (1,2) }

Answer for (2,4) is { (1,4) , (2,0) , (2,1) , (2,2) , (2,3) , (2,4) , (3,4) }

```
private HashSet<Point> maximalMatchedRegion(Point start, ArrayList<ArrayList<String>> board) {




}
```

```
private HashSet<Point> maximalMatchedRegion(Point start, ArrayList<ArrayList<String>> board) {

    HashSet<Point> matchedRegion = new HashSet<Point>();









    return matchedRegion;
}
```

```
private HashSet<Point> maximalMatchedRegion(Point start, ArrayList<ArrayList<String>> board) {

    HashSet<Point> matchedRegion = new HashSet<Point>();
    HashSet<Point> candidatesForInclusion = new HashSet<Point>();
    HashSet<Point> adjacentMatches = new HashSet<Point>();









    return matchedRegion;
}
```

```
private HashSet<Point> maximalMatchedRegion(Point start, ArrayList<ArrayList<String>> board) {

    HashSet<Point> matchedRegion = new HashSet<Point>();
    HashSet<Point> candidatesForInclusion = new HashSet<Point>();
    HashSet<Point> adjacentMatches = new HashSet<Point>();

    candidatesForInclusion.add(start);

    return matchedRegion;
}
```

```
private HashSet<Point> maximalMatchedRegion(Point start, ArrayList<ArrayList<String>> board) {

    HashSet<Point> matchedRegion = new HashSet<Point>();
    HashSet<Point> candidatesForInclusion = new HashSet<Point>();
    HashSet<Point> adjacentMatches = new HashSet<Point>();

    candidatesForInclusion.add(start);

    while (! candidatesForInclusion.isEmpty()) {









    }
    return matchedRegion;
}
```

```
private HashSet<Point> maximalMatchedRegion(Point start, ArrayList<ArrayList<String>> board) {

    HashSet<Point> matchedRegion = new HashSet<Point>();
    HashSet<Point> candidatesForInclusion = new HashSet<Point>();
    HashSet<Point> adjacentMatches = new HashSet<Point>();

    candidatesForInclusion.add(start);

    while (! candidatesForInclusion.isEmpty()) {
        for (Point p : candidatesForInclusion) {









        }









    }
    return matchedRegion;
}
```

```
private HashSet<Point> maximalMatchedRegion(Point start, ArrayList<ArrayList<String>> board) {

    HashSet<Point> matchedRegion = new HashSet<Point>();
    HashSet<Point> candidatesForInclusion = new HashSet<Point>();
    HashSet<Point> adjacentMatches = new HashSet<Point>();

    candidatesForInclusion.add(start);

    while (! candidatesForInclusion.isEmpty()) {
        for (Point p : candidatesForInclusion) {
            for (Point q : adjacentAndMatching(p, board)) {



            }
        }




    }
    return matchedRegion;
}
```

```
private HashSet<Point> maximalMatchedRegion(Point start, ArrayList<ArrayList<String>> board) {

    HashSet<Point> matchedRegion = new HashSet<Point>();
    HashSet<Point> candidatesForInclusion = new HashSet<Point>();
    HashSet<Point> adjacentMatches = new HashSet<Point>();

    candidatesForInclusion.add(start);

    while (! candidatesForInclusion.isEmpty()) {
        for (Point p : candidatesForInclusion) {
            for (Point q : adjacentAndMatching(p, board)) {

                adjacentMatches.add(q);

            }
        }




    }
    return matchedRegion;
}
```

```
private HashSet<Point> maximalMatchedRegion(Point start, ArrayList<ArrayList<String>> board) {

    HashSet<Point> matchedRegion = new HashSet<Point>();
    HashSet<Point> candidatesForInclusion = new HashSet<Point>();
    HashSet<Point> adjacentMatches = new HashSet<Point>();

    candidatesForInclusion.add(start);

    while (! candidatesForInclusion.isEmpty()) {
        for (Point p : candidatesForInclusion) {
            for (Point q : adjacentAndMatching(p, board)) {
                if (!matchedRegion.contains(q) && !candidatesForInclusion.contains(q)) {
                    adjacentMatches.add(q);
                }
            }
        }




    }
    return matchedRegion;
}
```

```
private HashSet<Point> maximalMatchedRegion(Point start, ArrayList<ArrayList<String>> board) {

    HashSet<Point> matchedRegion = new HashSet<Point>();
    HashSet<Point> candidatesForInclusion = new HashSet<Point>();
    HashSet<Point> adjacentMatches = new HashSet<Point>();

    candidatesForInclusion.add(start);

    while (! candidatesForInclusion.isEmpty()) {
        for (Point p : candidatesForInclusion) {
            for (Point q : adjacentAndMatching(p, board)) {
                if (!matchedRegion.contains(q) && !candidatesForInclusion.contains(q)) {
                    adjacentMatches.add(q);
                }
            }
        }
        matchedRegion.addAll(candidatesForInclusion);




    }
    return matchedRegion;
}
```

```
private HashSet<Point> maximalMatchedRegion(Point start, ArrayList<ArrayList<String>> board) {

    HashSet<Point> matchedRegion = new HashSet<Point>();
    HashSet<Point> candidatesForInclusion = new HashSet<Point>();
    HashSet<Point> adjacentMatches = new HashSet<Point>();

    candidatesForInclusion.add(start);

    while (! candidatesForInclusion.isEmpty()) {
        for (Point p : candidatesForInclusion) {
            for (Point q : adjacentAndMatching(p, board)) {
                if (!matchedRegion.contains(q) && !candidatesForInclusion.contains(q)) {
                    adjacentMatches.add(q);
                }
            }
        }
        matchedRegion.addAll(candidatesForInclusion);
        HashSet<Point> tmp = candidatesForInclusion;
        candidatesForInclusion = adjacentMatches;
        adjacentMatches = tmp;

    }
    return matchedRegion;
}
```

```
private HashSet<Point> maximalMatchedRegion(Point start, ArrayList<ArrayList<String>> board) {

    HashSet<Point> matchedRegion = new HashSet<Point>();
    HashSet<Point> candidatesForInclusion = new HashSet<Point>();
    HashSet<Point> adjacentMatches = new HashSet<Point>();

    candidatesForInclusion.add(start);

    while (! candidatesForInclusion.isEmpty()) {
        for (Point p : candidatesForInclusion) {
            for (Point q : adjacentAndMatching(p, board)) {
                if (!matchedRegion.contains(q) && !candidatesForInclusion.contains(q)) {
                    adjacentMatches.add(q);
                }
            }
        }
        matchedRegion.addAll(candidatesForInclusion);
        HashSet<Point> tmp = candidatesForInclusion;
        candidatesForInclusion = adjacentMatches;
        adjacentMatches = tmp;
        adjacentMatches.clear();
    }
    return matchedRegion;
}
```

```
private HashSet<Point> maximalMatchedRegion(Point start, ArrayList<ArrayList<String>> board) {

    HashSet<Point> matchedRegion = new HashSet<Point>();
    HashSet<Point> candidatesForInclusion = new HashSet<Point>();
    HashSet<Point> adjacentMatches = new HashSet<Point>();

    candidatesForInclusion.add(start);

    while (! candidatesForInclusion.isEmpty()) {
        for (Point p : candidatesForInclusion) {
            for (Point q : adjacentAndMatching(p, board)) {
                if (!matchedRegion.contains(q) && !candidatesForInclusion.contains(q)) {
                    adjacentMatches.add(q);
                }
            }
        }
        matchedRegion.addAll(candidatesForInclusion);
        HashSet<Point> tmp = candidatesForInclusion;
        candidatesForInclusion = adjacentMatches;
        adjacentMatches = tmp;
        adjacentMatches.clear();
    }
    return matchedRegion;
}
```

LINEAR SEARCH

Searching for a given value

Search

Computers are good at storing large amounts of data

Finding a particular value in a large collection is a typical operation

LINEAR SEARCH

how to search for a value in an unordered collection

general search

General problem: determine whether a value exists in a given collection

Assumption: to make things easier we assume that the collection contains Strings

Approach: define a method which accepts as arguments a Collection<String> and a value of type String, and returns a boolean indicating whether the value is in the collection

Pretend that 'contains' does not exist.

steps in defining method*

Step 1 – stub out the method

```
public boolean isMemberOf(String s, Collection<String> c){
    return false;
}
```

* Yes, 'contains' is a method already defined for this purpose. We are building an implementation of isMemberOf to understand how a method like contains works.

Step 2 – set up loop

Can use any of while/for/for-each.

```
public boolean isMemberOf(String s, Collection<String> c){
    for (String x : c) {
    }
    return false;
}
```

Step 3 – set up test in body of loop

```
public boolean isMemberOf(String s, Collection<String> c){
    for (String x : c) {
        if (s.equals(x)) {
        }
    }
    return false;
}
```

Step 3 – set up test in body of loop

```
public boolean isMemberOf(String s, Collection<String> c){
    for (String x : c) {
        if (s.equals(x)) {
            return true;
        }
    }
    return false;
}
```

Step 3 – set up test in body of loop

```
public boolean isMemberOf(String s, Collection<String> c){
    for (String x : c) {
        if (s.equals(x)) {
            return true;
        }
    }
    return false;
}
```

BUT WHAT IF s IS null ?

Step 3 – set up test in body of loop

```
public boolean isMemberOf(String s, Collection<String> c){
    for (String x : c) {
        if (s.equals(x)) {
            return true;
        }
    }
    return false;
}
```

AND WHAT IF c IS null ?

Step 4a – but we need to be careful, as s could be null!

```
public boolean isMemberOf(String s, Collection<String> c){
    if (c != null) {
        for (String x : c) {
           if (s==null) {
                if (s == x) {
                    return true;
                }
            }
            else {
                if (s.equals(x)) {
                    return true;
                }
            }
        }
    }
    return false;
}
```

Step 4b – but we need to be careful, as s could be null!

```
public boolean isMemberOf(String s, Collection<String> c){
    if (c != null) {
        if (s==null) {
            for (String x : c) {
                if (s == x) {  return true; }
            }
        }
        else {
            for (String x : c) {
                if (s.equals(x)) { return true; }
            }
        }
    }
    return false;
}
```

15.21.3. Reference Equality Operators == and !=

At run time, the result of == is true if the operand values are both null or both refer to the same object […]; otherwise, the result is false.

[…]

While == may be used to compare references of type String, such an equality test determines whether or not the two operands refer to the same String object. The result is false if the operands are distinct String objects, even if they contain the same sequence of characters […]. The contents of two strings s and t can be tested for equality by the method invocation s.equals(t).

http://docs.oracle.com/javase/specs/jls/se7/html/jls-15.html#jls-15.21.3

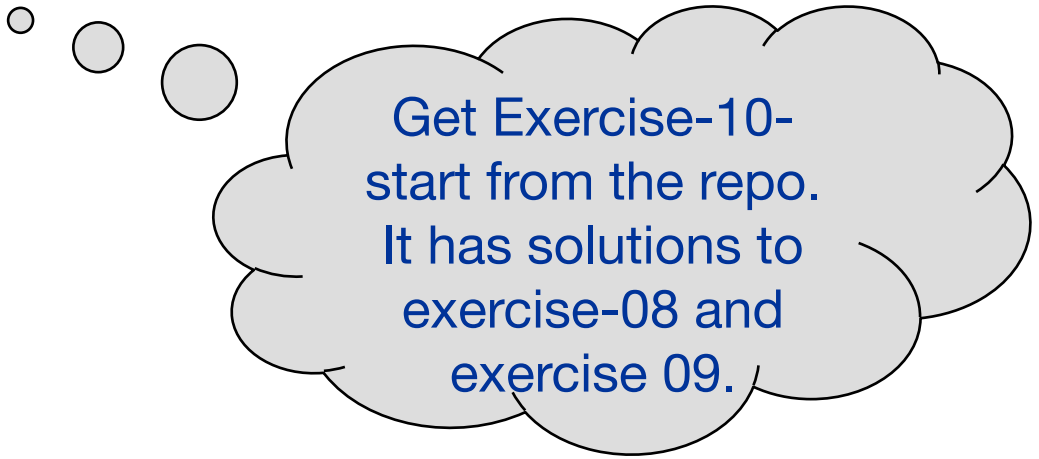
EXERCISE 10

EXERCISE 10

Define a method a method with this header in a class named quiz.Question:

```
public HashSet<HashSet<Point>> partition(ArrayList<ArrayList<String>> board)
```

Define the method so that it returns a HashSet<HashSet<Point>> that is a partition of the given board into disjoint and covering sets of contiguous and matching points from the board.

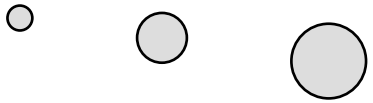


EXERCISE 10

EXAMPLE 1

```
Board: aabbc
       abbcd
       ddddd
       efffd

Partition:
{
  { (0,0) (1,0) (0,1) }
  { (0,4) }
  { (1,4) (2,0) (2,1) (2,2) (2,4) (2,3) (3,4) }
  { (3,3) (3,1) (3,2) }
  { (0,2) (1,1) (1,2) (0,3) }
  { (1,3) }
  { (3,0) }
}
```

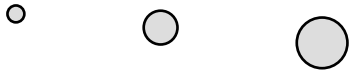


The board we've been using for several examples with exercises 8 and 9.

EXERCISE 10

EXAMPLE 2

```
Board: aba
       bab
       aba
```



A board for which every point is in its own partition.

```
Partition:
{
  { (0,0) }
  { (0,2) }
  { (1,1) }
  { (2,0) }
  { (2,2) }
  { (0,1) }
  { (1,0) }
  { (1,2) }
  { (2,1) }
}
```

EXERCISE 10

EXAMPLE 3

```
Board: aaaa
       aaaa
       aaaa
       aaaa
```

A board for which all points are in the same partition.

```
Partition:
{
  {(0,0)(0,2)(1,1)(2,0)(2,2)(1,3)(3,0)(3,2)(0,1)(1,0)(1,2)(2,1)(3,3)
(0,3)(2,3)(3,1)}
}
```