AP Computer Science A

KEY

DO's & DON'Ts

IMPORTANT CONCEPTS TO REVIEW AND REMEMBER

Random Numbers:

```
double ranNum = Math.random();                 interval: [0.0, 1.0)
int ran = (int)(Math.random() * n) + start;    interval: [start, start+n-1]
```

- Write a statement that will assign a double random number in the interval [1, 5) to `ranNum`.

  ranNum = Math.random() * 4.0 + 1.0

- Assume some names have been added to `nameList`. Assign a randomly selected value from `nameList` to `name`. In writing this statement, you must be sure that `name` could be assigned any name that is stored in `nameList`.

  ```
  List<String> nameList = new ArrayList<String>();
  String name;
  int i = (int)(Math.random() * nameList.size());
  name = nameList.get(i);
  ```

- Write a statement that will produce a random integer value in the range of 1 to 20 and store it in `numChips`.

  ```
  int numChips;
  numChips = (int)(Math.random() * 19 + 1);
  ```

Using REM (`%`) and DIV (/ with `ints`):

The `%` operator returns the remainder of a dividend and a divisor. When used with integers, / operator returns the quotient of a dividend divided by and a divisor.

These operators can be used to isolate digits in a number or to convert from one number base to another.

Example:

```
int number = 1035;
int onesDigit = number % 10;
int restOfDigits = number / 10;
int tensDigit = restOfDigits % 10;
restOfDigits = restOfDigits / 10;
```
and so on . . .

Initializing private instance variables:

Initializing private instance variables in a class is the responsibility of the constructor. When initializing these variables, it is important to remember that they have already been declared. **DO NOT REDECLARE PRIVATE INSTANCE VARIABLES**!

```
public class Date
{
   private int month;
   private int day;
   private int year;

   public Date(int m, int d, int y)
   {
         month = m;
         day = d;
         year = y;


   }
   . . .
}
```

Initializing arrays and lists in constructors:

When an array or a list is a private instance variable in a class, initializing the array or list is the responsibility of the constructor (or constructors). This usually involves instantiating the array or list.

```
public class HorseBarn
{
   private Horse[] barn;

   public HorseBarn(int numStalls)
   {
      barn = new
         Horse[numstalls];
   }
      . . .
}
```

```
public class CustomerList
{
   private List<Customer> customers;

   public CustomerList()
   {
      customers = new
         ArrayList<Customer>();
   }
   . . .
}
```

```
public class AnswerSheets
{
   private boolean[][] sheets;

   public AnswerSheets(int nr, int nc)
   {
      sheets =
         new boolean[nr][nc];
   }
   . . .
}
```

```
public class CustomerList
{
   private List <Customer> customers;

   public CustomerList(Customer[] list)
   {
      customers = new
         ArrayList<Customer>();
      for (Customer c : list)
         customers.add(c);
   }
   . . .
}
```

```
public class StudentRoster
{
   private String[] roster;

   //copy the names from chart to roster
   public StudentRoster(String[][] chart)
   {
      roster = new String[chart.length * chart[0].length];
      int index = 0;
      for (String[] row : chart)
         for (String s : row)
         {
            roster[index] = s;
            index++;
         }
   }
   . . .
}
```

Loops and Lists/Arrays: `for` vs `while`

When using a `for` loop, the `for` loop heading contains the loop control variable initialization, the test, and the loop control variable update. It is **bad form** to adjust the value of the loop control variable in the `for` loop. If the update in the body is conditional, consider using a `while` loop or in the case of removing items from a list, go backwards!

```
public class NameList
{
   private List<String> names;
   . . .

   public void removeAll(String name)
   {
      for (int k = 0; k < names.size(); k++)
      {
         if (name.equals(names.get(k))
         {
            names.remove(k);
            k--;  //bad form!!!!!!!!!
         }
      }
   }
}
```

```
public class NameList
{
   private List<String> names;
   . . .

   public void removeAll(String name)
   {
      int i = 0;
      while (i < names.size())
      {
         if (name.equals(names.get(i))
            names.remove(i);
         else
            i++; //conditional update
      }
   }
}
```

```
public class NameList
{
   private List<String> names;

   . . .

   //going backwards
   public void removeAll(String name)
   {
      for (int k = names.size() - 1; k >= 0; k--)
      {
         if (name.equals(names.get(k))
            names.remove(k);
      }
   }
}
```

Why does this code sometimes fail to remove all the occurrences of `name`?

```
public void removeAll(String name)
{
   for (int k = 0; k < names.size(); k++)
   {
      if (name.equals(names.get(k))
      {
         names.remove(k);
      }
   }
}
```

BECAUSE WHEN AN ELEMENT IS DELETED, THE NEXT ONE "SLIDES DOWN" AND ISN'T CHECKED.

Common Algorithms: Lists and Arrays

Inserting a new item into a sorted list: This is a search algorithm. You need to search the list to find where to insert a new item so that the list remains sorted after the insertion is done.

Any search in an array or list must check:

- is there more data in the array/list to process
- has the target item been found

If there is no more data left in the list to search, the search must stop. If there is more data, then the search continues and you must compare the target to the list's current item to see if the target should be inserted at the item's index or not. Notice that the check for more data **MUST** be done before comparing an item at a given index in the list to the target. Why?

THE TEST MUST BE BEFORE THE ACCESS SO THAT SHORT-CIRCUITING CAN PREVENT THE OBOE.

What is short-circuiting and how does it work in Java?

JAVA IS LAZY WHEN EVALUATION EXPRESSIONS WITH || OR &&. ONCE IT DETERMINES THE FINAL VALUE OF THE EXPRESSION, IT STOPS EVALUATING THE EXPRESSION.

Explain how short circuiting will avoid a runtime exception in the following example.

Assume that a, b, and n are int variables and have been initialized.

```
if (a != b && (n / (a - b)) > 90)
```

IF a==6 THEN WE HAVE false && ... WHICH IS false. JAVA USES SHORT-CIRCUITING TO PREVENT THE DIVISION BY ZERO (WHEN a==6, a-6 IS ZERO)

```
public class NameList //while loop implementation
{
   private List<String> names;

   // precondition: names is in ascending order
   // postcondition: newName has been inserted into names, names is in ascending order
   public void insert(String newName)
   {
      int index = 0;
      while (index < names.size() && newName.compareTo(names.get(index)) > 0)
         index++;
      names.add(index, newName);
   }
   . . .
}
```

What makes this while loop stop?

IT STOPS WHEN (index >= names.size() || newName.compareTo(names.get(index)) <= 0)

IN OTHER WORDS, IT STOPS WHEN index CONTAINS THE POSITION WHERE newName SHOULD BE INSERTED.

Where is newName inserted and how can you be sure that the list is still sorted once the insertion has been done?

IT IS INSERTED AT POSITION index. index IS THE CORRECT POSITION BECAUSE
- THE FACT THAT THE while CONDITION WAS true FOR POSITION index -1 MEANS THAT IT GOES AFTER index -1.
- THE FACT THAT THE while CONDITION IS false FOR POSITION index, MEANS THAT IT GOES BEFORE OR AT index.

General algorithm of the while loop version of the insert:

- INITIALIZE index TO 0.
- WHILE index IS NOT THE CORRECT POSITION TO INSERT, INCREMENT index;
- INSERT AT index.

```
public class NameList //for loop implementation
{
   private List<String> names;

   // precondition: names is in ascending order
   // postcondition: newName has been inserted into names, names is in ascending order
   public void insert(String newName)
   {
      for (int k = 0; k < names.size(); k++)
      {
         if (newName.compareTo(names.get(k) <= 0)
         {
            names.add(k, newName);
            return;
         }
      }
      names.add(newName);
   }
   . . .
}
```

What makes this for loop stop?

IT STOPS WHEN !(k < names.size()). IN OTHER WORDS WHEN k >= names.size() WHICH MEANS THAT k IS OUT OF BOUNDS.

Where is newName inserted and how can you be sure that the list is still sorted once the insertion has been done?

IF IT IS INSERTED INSIDE THE LOOP, IT'S IN THE CORRECT PLACE. THE JUSTIFICATION IS SIMILAR TO THE ONE ON THE PREVIOUS PAGE.

IF IT IS INSERTED AFTER THE LOOP, THEN IT BELONGS AT THE END WHICH IS WHERE IT IS INSERTED.

General algorithm of the for loop insert:

FOR EACH k FROM 0 TO names.size() EXCLUSIVE, IF k IS THE CORRECT POSITION TO INSERT, THEN INSERT AND RETURN.

IF THE ITEM DOESN'T GO BEFORE ANY ITEM, THEN IT GOES AT THE END. SO PUT IT THERE.

Compare the two implementations of the insert method. Which implementation has fewer special cases to code?

THE WHILE LOOP HAS FEWER SPECIAL CASES.

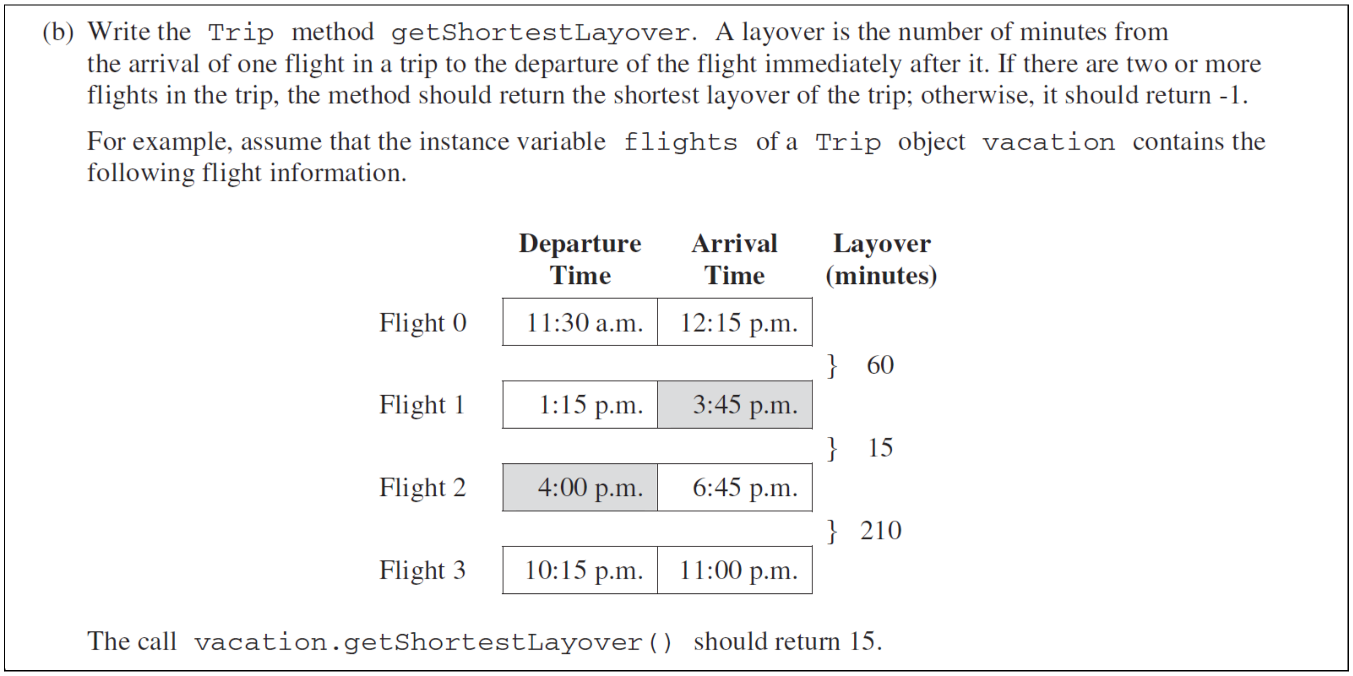
Finding the min or the max in a list or array: This is a type of search algorithm.

To find the min (or max) value in a list or array:

- Assume that the first item in the list or array is the min and assign that value to a variable that will store the current min value
- Go through the list and compare the current min value to each item in the list or array. If an item in the array is smaller than the min, set the current min value to that item.

```
//precondition: temperatures.length > 0
public static double findMin(double[] temperatures)
{
   double min = temperatures[0];
   for (double temp : temperatures)
   {
      if (temp < min)
         min = temp;
   }
   return min;
}
```

The find min/find max algorithm frequently shows up on the AP CS A exam. Here are some recent free response examples.



(b) Write the `Trip` method `getShortestLayover`. A layover is the number of minutes from the arrival of one flight in a trip to the departure of the flight immediately after it. If there are two or more flights in the trip, the method should return the shortest layover of the trip; otherwise, it should return -1.

For example, assume that the instance variable `flights` of a `Trip` object `vacation` contains the following flight information.

	Departure Time	Arrival Time	Layover (minutes)
Flight 0	11:30 a.m.	12:15 p.m.	
			} 60
Flight 1	1:15 p.m.	3:45 p.m.	
			} 15
Flight 2	4:00 p.m.	6:45 p.m.	
			} 210
Flight 3	10:15 p.m.	11:00 p.m.	

The call `vacation.getShortestLayover()` should return 15.

(b) Write the `BatteryCharger` method `getChargeStartTime` that returns the start time that will allow the battery to be charged at minimal cost. If there is more than one possible start time that produces the minimal cost, any of those start times can be returned.

For example, using the rate table given at the beginning of the question, the following table shows the resulting minimal costs and optimal starting hour of several possible charges.

Hours of Charge Time	Minimum Cost	Start Hour of Charge	Last Hour of Charge
1	40	12	12
2	110	0	1
		or	
		23	0 (the next day)
7	550	22	4 (the next day)
30	3,710	22	3 (two days later)

Assume that `getChargingCost` works as specified, regardless of what you wrote in part (a).

Complete method `getChargeStartTime` below.

```
/** Determines start time to charge the battery at the lowest cost for the given charge time.
 *  @param chargeTime the number of hours the battery needs to be charged
 *        Precondition: chargeTime > 0
 *  @return an optimal start time, with 0 ≤ returned value ≤ 23
 */
public int getChargeStartTime(int chargeTime)
```

(b) Write the method `getLongestRun` that takes as its parameter an array of integer values representing a series of number cube tosses. The method returns the starting index in the array of a run of maximum size. A run is defined as the repeated occurrence of the same value in two or more consecutive positions in the array.

For example, the following array contains two runs of length 4, one starting at index 6 and another starting at index 14. The method may return either of those starting indexes.

If there are no runs of any value, the method returns `-1`.

Index	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17
Result	1	**5**	**5**	4	3	1	**2**	**2**	**2**	**2**	6	1	**3**	**3**	**5**	**5**	**5**	**5**

Complete method `getLongestRun` below.

```
/** Returns the starting index of a longest run of two or more consecutive repeated values
 *  in the array values.
 *  @param values an array of integer values representing a series of number cube tosses
 *        Precondition: values.length > 0
 *  @return the starting index of a run of maximum size;
 *          -1 if there is no run
 */
public static int getLongestRun(int[] values)
```

Algorithms that require comparing neighbors in an array/list:

Consider writing a method that will return `true` if values in a given array are in increasing order; `false` otherwise. To determine this, each pair of neighbors must be compared to confirm that the left neighbor is less than or equal to the right neighbor. If just one of these tests is fails, a value of `false` should be returned. To return `true`, you must compare all neighbor pairs and each of those tests must confirm that the left neighbor is less than or equal to the right neighbor.

To code this algorithm, we will first concentrate on accessing each neighbor pair and printing them without causing a boundary error.

To do this, a loop will be required to access all neighbor pairs and you must be sure to adjust the loop boundaries so that an `ArrayIndexOutOfBoundsException` will not occur.

```
public static void printAllNbrs(int[] nums)
{
   for (int k = 0; k < nums.length - 1; k++)
   {
      System.out.println(nums[k] + " " +
                         nums[k + 1]);
   }
}
// using nums[k] and nums[k + 1]
```

```
public static void printAllNbrs (int[] nums)
{
   for (int k = 1__; k < nums.length________; k++)
   {
      System.out.println(nums[k - 1] + " " +
                         nums[k]);
   }
   return true;
}
// using nums[k - 1] and nums[k]
```

Not adjusting the loop boundaries is a common mistake when writing algorithms that involve using neighboring values in an array and will in some cases cause an out of bounds error.

Now we turn our attention to proving that for every neighbor pair, the left neighbor is less than the right neighbor. In cases such as this, it turns out that it is easier to test the opposite; that for at least one neighbor pair, the left neighbor is greater than or equal to the right neighbor. If this proves to be true, we return `false`, because the list is not in increasing order. If we are not able to find any neighbor pair where the left neighbor is greater than or equal to the right neighbor, the list must be in increasing order and we return `true`. You cannot return `true` until AFTER the loop completes and all pairs have been compared.

```
public static boolean isIncreasing(int[] nums)
{
   for (int k = 0; k < nums.length - 1; k++)
   {
      if (nums[k] >= nums[k + 1])
         return false;
   }
   return true;
}
// comparing nums[k] and nums[k + 1]
```

```
public static boolean isIncreasing(int[] nums)
{
   for (int k = 1; k < nums.length; k++)
   {
      if (nums[k - 1] >= nums[k])
         return false;
   }
   return true;
}
// comparing nums[k - 1] and nums[k]
```

Look at these attempts to write the `isIncreasing` method and find the intent (logic) error in each.

```
public static boolean isIncreasing(int[] nums)
{
   for (int k = 0; k < nums.length - 1; k++)
   {
      if (nums[k] >= nums[k + 1])
         return false;
       else
         return true;
   }
   return true;
}
```

ALWAYS RETURNS. ONLY THE FIRST PAIR IS TESTED. LOOP DOES NOT COMPLETE.

```
public static boolean isIncreasing(int[] nums)
{
   for (int k = 0; k < nums.length - 1; k++)
   {
      if (nums[k] < nums[k + 1])
         return true;
   }
   return false;
}
```

RETURNS true IF ANY PAIR IS CORRECT. BUT ALL PAIRS NEED TO BE CORRECT.

String Advice: When solving `String` problems, stay away from the `char` data type.

Using a `char` is tricky, especially when you try to concatenate two characters.

`String s = 'a' + 'b';` will not create the `String` `"ab"`. It will cause an "incompatible type" compiler error because an `int` is being assigned to a `String`.

If you need to process each character of a given `String`, use the `substring` method and create substrings of length 1.

Example: `String letter = word.substring(index, index + 1);`
Creates a one letter substring of the character found at `index` in `word`.

Complete the following method that returns a `String` with changes all occurrences of `sourceLetter` in `str` to `targetLetter`.

```
public static String changeSource(String str, String sourceLetter, String targetLetter)
{
   String result = "";
   for (int i = 0 ; i < str.length() ; i++)
      if (str.substring(i, i+1).equals(source letter))
            result += targetLetter;
      else
            result += str.substring(i, i+1);

   return result;
}
```

`null` References and the `NullPointerException`:

A reference variable contains the address of an object or `null`. If the variable contains `null`, you cannot dereference the variable, i.e. call an object's method.

```
String s = null;
System.out.println(s.length()); //will generate a NullPointerException at
runtime
```

Special care should be taken when searching an array or list that contains objects to be sure that the array does not contain any `null` values. If that possibility exists, you must check for `null` **BEFORE** calling an object's method.

Here is an example from the `HorseBarn` free response.

2012 AP® COMPUTER SCIENCE A FREE-RESPONSE QUESTIONS

Complete method `findHorseSpace` below.

```
/** Returns the index of the space that contains the horse with the specified name.
 *  Precondition: No two horses in the barn have the same name.
 *  @param name  the name of the horse to find
 *  @return  the index of the space containing the horse with the specified name;
 *           -1  if no horse with the specified name is in the barn.
 */
public int findHorseSpace(String name)
{
    for (int k = 0; k < spaces.length; k++)
    {
        if (spaces[k] != null &&
              name.equals(spaces[k].getName()))
        {
            return k;
        }
    }
    return -1;
}
```

Using For-Each Loops (Enhanced for Loops) to traverse arrays and lists:

For-Each Loops access the elements of the array/list “for free,” which often provides advantages over using indexed loops to traverse arrays/lists:

- Less code is required.
- The code is easier to read.
- A partial Free Response exam solution might score an additional point.

Here are some examples of indexed loop vs. For-Each loop traversals.

```
Indexed Loop Traversals                                   | For-Each Loop Traversals
----------------------------------------------------------+-----------------------------------------------
public void printAll(int[] nums)                          | public void printAll(int[] nums)
{                                                         | {
   // k is an index                                       |    // n is an element of nums
   for (int k = 0; k < nums.length; k++)                  |    for (int n : nums)
   {                                                      |    {
      int n = nums[k];                                    |       System.out.println(n);
      System.out.println(n);                              |    }
   }                                                      | }
}                                                         |
----------------------------------------------------------+-----------------------------------------------
public void printAll(List<String> names)                  | public void printAll(List<String> names)
{                                                         | {
   // k is an index                                       |    // n is an element of nums
   for (int k = 0; k < names.size(); k++)                 |    for (String n : names)
   {                                                      |    {
      String n = names.get(k);                            |       System.out.println(n);
      System.out.println(n);                              |    }
   }                                                      | }
}                                                         |
----------------------------------------------------------+-----------------------------------------------
public void printAll(String[][] names)                    | public void printAll(String[][] names)
{                                                         | {
   // r & c are indexes                                   |    // row is an element of String[]
   for (int r = 0; r < names.length; r++)                 |    for (String[] row : names)
   {                                                      |    {
      for (int c = 0; c < names[0].length; c++)           |       // n is an element of row
      {                                                   |       for (String n : row)
         String n = names[r][c];                          |       {
         System.out.print(n + "\t");                      |          System.out.print(n + "\t");
      }                                                   |       }
      System.out.println(); // New line                   |       System.out.println(); // New line
   }                                                      |    }
}                                                         | }
```

For-Each loops have some restrictions though. Never use a For-Each loop when you:

- need the indexes of elements of the array/list.
- want to traverse the array/list in a different order than front to back (lowest index to highest).
- want to add or delete elements of a list inside the loop (change the size of the list). This will result in a `ConcurrentModificationException` at runtime.

Consider each of the paired traversal examples below.

Which is the best choice of loop to use for the task? Identify any errors.

Indexed Loop Traversals	**For-Each Loop Traversals**

`public void printAll(int[] nums)` BEST, BUT
`{`
`   for (int k = 0; k < nums.length; k++)`
`   {`
`      System.out.println(nums[k]);`
`   }`
`}`
	`public void printAll(int[] nums)`
	`{`
	`   for (int n : nums)`
	`   {`
	`      System.out.println(nums[n]);` n
	`   }`
	`}`
	DOESN'T WORK. n IS A VALUE, NOT AN INDEX

`public int search(String[] names,` BEST
`                  String target)`
`{`
`   for (int k = 0; k < names.length; k++)`
`   {`
`      if (names[k].equals(target))`
`         return k;`
`   }`
`   return -1;`
`}`
	`public int search(String[] names,`
	`                  String target)`
	`{`
	`   int index = 0;`
	`   for (String n : names)`
	`   {`
	`      if (n.equals(target))`
	`         return index;`
	`      index++;`
	`   }`
	`   return -1;`
	`}`

`//` Return first name with less than 3 characters;
`//` Return `null` if there are no short names.
`public String findShort(String[] names)`
`{`
`   for (int k = 0; k < names.length; k++)`
`   {`
`      if (names[k].length() <= 3)`
`         return names[k];`
`   }`
`   return null;`
`}`
	`//` Return first name with less than 3 characters; BEST
	`//` Return `null` if there are no short names.
	`public String findShort(String[] names)`
	`{`
	`   for (String n : names)`
	`   {`
	`      if (n.length() <= 3)`
	`         return n;`
	`   }`
	`   return null;`
	`}`

`//` Remove all names with less than 3 characters. BEST!!!
`public void removeShort(List<String> names)`
`{`
`   for (int k = names.size() - 1; k >= 0; k--)`
`   {`
`      if (names.get(k).length() <= 3)`
`         names.remove(k);`
`   }`
`}`
	`//` Remove all names with less than 3 characters.
	`public void removeShort(List<String> names)`
	`{`
	`   for (String n : names)`
	`   {`
	`      if (n.length() <= 3)`
	`         names.remove(n);`
	`   }`
	`}`

`//` Duplicate all elements in nums creating consecutive pairs.
`public void doubleUp(List<Integer> nums)`
`{`
`   for (int k = 0; k < nums.size(); k += 2)`
`   {`
`      nums.add(k, nums.get(k));` BEST !!!
`   }`
`}`
	`//` Duplicate all elements in nums creating consecutive pairs.
	`public void doubleUp(List<Integer> nums)`
	`{`
	`   int k = 0;`
	`   for (Integer n : nums)`
	`   {`
	`      nums.add(k, n);`
	`      k += 2;`
	`   }`
	`}`

NEITHER OF THESE WORK. YOU ARE NOT ALLOWED TO CHANGE THE SIZE OF A LIST INSIDE OF A FOR-EACH LOOP. THIS CAUSES A ConcurrentModification Exception RUNTIME ERROR.