



University of Maryland College Park

Department of Computer Science

CMSC132 Summer 2023

Exam #2

FIRSTNAME, LASTNAME (PRINT IN UPPERCASE):

KEY

STUDENT ID (e.g., 123456789):

Instructions

- Please print your answers and use a pencil.
- This exam is a closed-book, closed-notes exam with a duration of 50 minutes and 100 total points.
- **Do not remove the exam's staple.** Removing it will interfere with the scanning process (even if you staple the exam again).
- Write your directory id (e.g., terps1, not UID) at the bottom of pages with **DirectoryId**.
- Provide answers in the rectangular areas.
- Do not remove any exam pages. Even if you don't use the extra pages for scratch work, return them with the rest of the exam.
- Your code must be efficient and as short as possible.
- If you continue a problem on the extra page(s) provided, make a note on the particular problem.
- You don't need to use meaningful variable names; however, we expect good indentation.
- **You must write your name and id at this point (we will not wait for you after time is up).**
- You must stop writing once time is up.

Grader Use Only

#1	Part #1 (Short Answer)	30	
#2	Part #2 (Code)	70	
Total	Total	100	

Part #1 (Short answers 3 points each)

1. List the following Big O expressions in order of asymptotic complexity (lowest complexity first).

$O(n \log(n))$

$O(1)$

$O(\log(n))$

$O(n^2)$

$O(n!)$

$O(1)$ $O(\log(n))$ $O(n \log(n))$ $O(n^2)$ $O(n!)$

2. Write the formal definition of Big O as given in lecture. That is, given a time function **$f(n)$** and a simple function **$g(n)$** , write the condition that implies **$f(n)$** is in **$O(g(n))$** . In your definition use **M** as a positive constant, **N_0** as the variable for when you start to get an upper bound, and **n** as the size of the input. As all components are already defined, you just need to write one mathematical expression.

$M * g(n) \geq f(n)$ for all $n \geq N_0$

3. Indicate the algorithm complexity of the following expression using the best bound: **$5n \log(n) + n^2 + 30n + 671263$**

$O(n^2)$

4. In the code section of this exam, you will write a method called `addORReplacE`. (Answer when done with code part)

What is the best case runtime for this method in terms of number of comparisons made, assuming **n** is the number of strings in the field bank? **$O(1)$**

Explain the scenario that leads to the best case? **find and replace at first element is constant work**

What is the worst case runtime in terms of number of comparisons made, assuming **n** is the number of strings in the field bank? **$O(n)$**

For question 5 to 10, assume the following classes all in the same package.

<pre> public class Animal { @Override public String toString() { return "I am an Animal"; } public void whoAmI() { System.out.println(this); } } </pre>	<pre> public class Bird extends Animal { @Override public String toString() { return "I am a Bird"; } public void whoAmI() { System.out.println(this); } } </pre>
<pre> public class Heron extends Bird{ @Override public String toString() { return "I am a Heron"; } public void whoAmI() { System.out.println(this); } } </pre>	<pre> public class BlueHeron extends Heron { @Override public String toString() { return "I am a Blue Heron"; } public void whoAmI() { System.out.println(this); } } </pre>

For questions 5 to 10, assume the code given is the only code placed in a main method. Simply write **NC** if it would not compile, **CE** if it would compile but throw an exception, **C** if it would compile but no output to the console, or write the output if it will compile and produce output.

5. NC, C, CE, or output? **I am a Blue Heron**

```

ArrayList <Heron> List = new ArrayList<>();
List.add(new BlueHeron());
List.get(0).whoAmI();

```

6. NC, C, CE, or output? **I am a Heron**

```

ArrayList <? super Heron> exampleSuper = new ArrayList<Bird>();
exampleSuper.add(new Heron());
System.out.println(exampleSuper.get(0));

```

7. NC, C, CE, or output? **NC**

```

ArrayList <? extends Heron> exampleExt= new ArrayList<BlueHeron>();
exampleExt.add(new Heron());
System.out.println(exampleExt.get(0));

```

8. NC, C, CE, or output? **I am a Blue Heron**

```

BlueHeron bh [] = {new BlueHeron()};
Heron h[] =bh;
System.out.println(h[0]);

```

9. NC, C, CE, or output? **CE**

```

BlueHeron bh [] = {new BlueHeron()};
Heron h[] =bh;
h[0]= new Heron();

```

10. NC, C, CE, or output? **C**

```

ArrayList <BlueHeron> bhList =new ArrayList<BlueHeron>();
bhList.add(new BlueHeron());
ArrayList <? extends Animal> exampleExt=bhList;
Animal a =bhList.get(0);

```

Part #2 (Code)

Given the code below, finish the missing methods. All code is in the same package.

```
import java.util.ArrayList;
import java.util.Arrays;

public class WordBank implements Cloneable{

    private ArrayList <String> bank;

    //data in non-null argument array assigned to bank field
    //assume at least one non-null String
    public WordBank(String[] bank) {
        this.bank = new ArrayList <String>();
        this.bank.addAll(Arrays.asList(bank));
    }

    //if find is in bank, replace first instance
    //if not, add find at end of bank
    //assume neither argument is null
    public void addORReplace(String find, String replace){
        //YOU WILL CODE THIS
    }

    //private helper method called in the 3 public makeStr
    private String makeStr(ConcatenateInterface c){
        return c.concatenate();
    }

    //local class syntax must be used
    public String makeStr1(){
        //YOU WILL CODE THIS — ONLY LAST STATEMENT GIVEN
        return makeStr(new localClass());
    }

    //anonymous class syntax
    public String makeStr2(){
        //YOUR CODE WILL REPLACE ?
        return makeStr(?);
    }

    //lambda expression syntax
    public String makeStr3(){
        //YOUR CODE WILL REPLACE ?
        return makeStr(?);
    }

    @Override
    public String toString() {
        return bank.toString();
    }

    @Override
    public WordBank clone() {
        //YOU WILL CODE THIS
    }

    public CSWordIterator geCSWordIterator() {
        return new CSWordIterator();
    }

    //public inner class
    public class CSWordIterator{
        private int index =0; //no other fields, constructors, or methods
        public String next() {
            //YOU WILL CODE THIS
        }
    }
}
```

```
public interface ConcatenateInterface {
    public String concatenate();
}
```

Driver

```
public class Driver {

    public static void main(String[] args) {

        WordBank wb = new WordBank(new String[] {"cat",
            "if*", "red", "else*", "phone"});

        System.out.println(wb);

        System.out.println(wb.makeStr1());
        System.out.println(wb.makeStr2());
        System.out.println(wb.makeStr3());

        wb.addORReplace("bird", "java*");
        System.out.println(wb); //no bird, so added

        wb.addORReplace("red", "blue");
        System.out.println(wb); //blue in place of red

        WordBank wbClone = wb.clone();

        wb.addORReplace("while*", "purple");
        System.out.println(wb);
        System.out.println(wbClone); //while* not in clone

        WordBank.CSWordIterator csI = wb.geCSWordIterator();

        //let us just loop 10 times to see it cycle a few times
        for (int i=0; i<10; i++ )
            System.out.print(csI.next() + "    ");

    }

}
```

Driver Output

```
[cat, if*, red, else*, phone]
cat if* red else* phone
?cat?
$phone$
[cat, if*, red, else*, phone, bird]
[cat, if*, blue, else*, phone, bird]
[cat, if*, blue, else*, phone, bird, while*]
[cat, if*, blue, else*, phone, bird]
if*    else*    while*    THE END    if*    else*    while*    THE END    if*    else*
```

1. Write the code for the `addORReplace` method. Assume both arguments are not null. If `find` is in `bank`, replace the first instance with `replace`. If not, add `find` to the end of `bank`. Only allowed library methods are `ArrayList` methods: `size`, `add`, `set`, and `get`. Also allowed is the `equals` method for `Strings`.

```
public void addORReplace(String find, String replace){

    for (int i = 0; i<bank.size(); i++)
    {
        if(bank.get(i).equals(find))
        {
            bank.set(i, replace);
            return;
        }
    }

    //never found find, add at end
    bank.add(find);
}
```

2. Write the code for the `makeStr1` method. Use local class syntax to make a class called `localClass` that implements `ConcatenateInterface`. In the class implement `concatenate` by concatenating all strings in `bank` (with one space after each string) and return the string. No library methods allowed. Must use a for-each loop. Last line is given.

```
public String makeStr1(){

    class localClass implements ConcatenateInterface{
        public String concatenate() {
            String str = "";
            for (String s: bank)
                str+=s+" ";
            return str;
        }
    }

    return makeStr(new localClass()); }
}
```

3. Write the code for the `makeStr2` method. Use anonymous class syntax to make an object that *IS A* `ConcatenateInterface`. In the class implement `concatenate` by concatenating `?` before and after the first element in `bank`. Only library method allowed is `get` of `ArrayList`. Put code in argument to private `makeStr`.

```
public String makeStr2(){
    return makeStr(
        new ConcatenateInterface() {
            public String concatenate() {
                return "?" + bank.get(0) + "?";
            }
        }
    );
}
```

4. Write the code for the `makeStr3` method. Use lambda expression syntax (without explicit `return`) to make an object that *IS A* `ConcatenateInterface`. In the class implement `concatenate` by concatenating `$` before and after the last element in `bank`. Only library method allowed is `get` and `size` of `ArrayList`. Put code in argument to private `makeStr`.

```
public String makeStr3(){
    return makeStr(
        () -> "$" + bank.get(bank.size() - 1) + "$"
    );
}
```

5. Complete the `try` part of the `clone` method. The resulting object must be independent of the original. Only methods allowed is the **default constructor** of `ArrayList` and `add`. You can also call `clone` of `Object` class.

```
public WordBank clone() {
    WordBank obj = null;
    try {

        obj = (WordBank) super.clone();

        obj.bank = new ArrayList <String>();
        for(String w : bank)
        {
            obj.bank.add(w);
        }

    } catch (CloneNotSupportedException e) {
        e.printStackTrace();
    }
    return obj;}
}
```

6. Complete the `next` method of the `CSWordIterator` inner class. Every call to `next` should return a `String` from `bank` that ends with an `*`. Once you run out of such strings, the next call to `next` should return "THE END". The next call after that will restart the pattern. See sample code for an example. Only methods allowed are `ArrayList` methods: `size` and `get`. String methods `charAt` and `length` are also allowed. Remember you have the inner class field `index` you can use.

```
public class CSWordIterator{
    private int index =0; //no other fields, constructors, or methods

    public String next() {

        for(int i = index; i < bank.size(); i++)
        {
            String temp = bank.get(i);
            if (temp.charAt(temp.length()-1) == '*')
            { index =(i+1);
              return temp;
            }
        }

        index =0;
        return "THE END"; }

}
```