Data Structures & Algorithms Problem Sheet 1

Worth 7% of your final grade
Due Friday 19/10, 7pm in Moodle
Submit the following files to Moodle: `StringStack.java`, `Question2.java`, `StringRepeater.java` (your modified version), `Question3.pdf` and `DNABook.java`.

Note:

- Marks will be given based on correctly working code and correct answers submitted. You do not need to comment your code, unless there is a problem with it (then comments may give you partial marks).
- Do not copy code or answers from others. More about plagiarism and how to avoid it here: http://www.bath.ac.uk/library/help/infoguides/plagiarism.html
- If you submit code that does not compile, you can only get a maximum of 60% of the marks for it. If your code does not work, it is much better to make it compile and leave comments why you think it does not work in the code.
- For this problem sheet you should implement your own algorithms and data structures and not use pre-defined ones such as the ones from the Java API.
- Please respect the Java conventions for naming of methods and classes, e.g. the use of upper/lower case: http://www.oracle.com/technetwork/java/codeconventions-135099.html
- Please ensure that the files you submit are recognized as correct text files (beware of copy-pasting special characters from the lecture slides as they can confuse the Java compiler).
- If you are struggling with Java or programming tools such as Eclipse, have a look for video tutorials on YouTube. Many students found them useful.
- After submitting, please check your submission on Moodle to make sure it is there and you have submitted the right files.
- We aim to provide feedback no later than three weeks after the submission date.

The questions start on the next page…

Question 1: Abstract Data Type `StringStack` (25% of marks)

Given the following incomplete implementation of the abstract data type `StringStack` in the file `StringStack.java` (download it from Moodle):

```
/** A stack abstract data type that contains Strings. */
public class StringStack {

    // TODO add variables for data here

    /**
     * Constructor for creating a new StringStack with a certain capacity.
     * @param capacity the maximum number of strings the stack can hold
     */
    public StringStack(int capacity) {
        // TODO implement this
    }

    /**
     * Puts the given String on top of the stack (if there is enough space).
     * @param s the String to add to the top of the stack
     * @return false if there was not enough space in the stack to add the string;
                otherwise true
     */
    public boolean push(String s) {
        // TODO implement this
    }

    /**
     * Removes the String on top of the stack from the stack and returns it.
     * @return the String on top of the stack, or null if the stack is empty.
     */
    public String pop() {
        // TODO implement this
    }

    /**
     * Returns the number of Strings in the stack.
     * @return the number of Strings in the stack
     */
    public int count() {
        // TODO implement this
    }
}                                                    // question continued on next page…
```

Complete the implementation of `StringStack` using an array to store the stack elements and test it with the executable class `StringStackTest.java` given on Moodle. **Submit your file `StringStack.java` to Moodle.**

Question 2: Measuring Performance (25% of marks)

Given the following source code file **`StringRepeater.java`** (download it from Moodle):

```
public class StringRepeater {

    public String repeatString(String s, int n){
        String result = "";
        for(int i=0; i<n; i++) {
            result = result + s;
        }
        return result;
    }
}
```

a) (10% of marks) Create an executable class **`Question2.java`** with a main method that does the following:
it measures and prints a **"naïvely" measured runtime** (i.e. only one run measured, as described in the lecture) for `repeatString()`, with the string "hello" for n=1, n=100, n=1000 and n=10000. That is, four different runtimes should be printed, in the format "`T(…) = … seconds`" (the dots will be the numbers for n and the time).

b) (5% of marks) Now we improve our measurements. After the code from part a), add more code to the main method of your class **`Question2.java`** that does the following:
it measures and prints a **fairly precise runtime** (as described in the lecture) for `repeatString()`, with the string "hello" for n=1, n=100, n=1000 and n=10000. That is, four different runtimes should be printed, in the format "`T(…) = … seconds`" (the dots will be the numbers for n and the time). Please make sure that the naïve measurement from part a) is still printed before printing the more precise runtime in this part, as it will be marked separately. **Submit your file `Question2.java` to Moodle.**

c) (10% of marks) Let's try and improve the performance a bit by optimizing the code of `StringRepeater`. Modify **`StringRepeater.java`** so that the `repeatString()` method uses the **`StringBuffer`** class instead of working directly with `String`s. Google for `StringBuffer` to find out more about it and learn how it can be used to improve the performance of `repeatString()`, e.g. by reading the answer here:
http://stackoverflow.com/questions/2439243/what-is-the-difference-between-string-and-stringbuffer-in-java
Measure the improved performance using your class `Question2.java`. **Submit the modified file `StringRepeater.java` to Moodle.**

Question 3: Analysing Time Complexity (20%)

a) (10% of marks) Analyse the time complexity of the unoptimised `repeatString()` method from Question 2. That is, **list the elementary operations and then sum them all up to come up with a formula for T(n)**, as illustrated in the lecture. Important: The operation `result + s` in the code has a special runtime (not just 1 elementary operation), so note down the time for this operation as an unknown **X** in your formula for T(n).

b) (5% of marks) Look at the runtime you measured for the unoptimised `repeatString()` method in Question 2 b). Which complexity class in terms of Big-Theta does `repeatString()` have (e.g. $\Theta(\log n)$, $\Theta(n)$, $\Theta(n^2)$ or $\Theta(n^3)$, *etc.*)? If the pattern is not clear, you may want to do some extra runtime measurements, e.g. for n=20,000 or even larger n. **Name the right complexity class and provide a single sentence explaining your choice.**

c) (5% of marks) Look at the runtime you measured for the optimised `repeatString()` method from Question 2 c), i.e. the one which uses `StringBuffer`. Which complexity class in terms of Big-Theta does the optimized `repeatString()` have (e.g. $\Theta(\log n)$, $\Theta(n)$, $\Theta(n^2)$ or $\Theta(n^3)$, *etc.*)? If the pattern is not clear, you may want to do some extra runtime measurements, e.g. for n=100,000 or even larger n. **Name the right complexity class and provide a single sentence explaining your choice.**

Submit all your answers for Question 3 in a single PDF file called `Question3.pdf` to Moodle.

Question 4: Abstract Data Type `SocialNetwork` (30%)

Imagine you are developing a new social network for all participants of the Data Structures & Algorithms unit: D&A ~~Face~~book! As a starting point you are given the following Java interface `SocialNetwork.java` (download it from Moodle):

```
public interface SocialNetwork {

    void registerUser(String name);
    void becomeFriends(String name1, String name2);
    boolean areTheyFriends(String name1, String name2);

}
```

For each new user, first method `registerUser` has to be called to register the user in the network. Once users are registered, they can become friends by calling method `becomeFriends`. The network can check whether two users are friends with method `areTheyFriends`. All methods receive the names of the users as arguments (you can assume that two users cannot have the same name). For this implementation, you may assume that there are at most 100 users in a social network.

a) (15% of marks) Create a class `DNABook.java` that implements the `SocialNetwork` interface. Test it with the executable class `DNABookTest.java` given on Moodle.

b) (15% of marks) Optimise your implementation in `DNABook.java` so that the runtimes of the methods `becomeFriends` and `areTheyFriends` grow logarithmically with the number of registered users. For this implementation you can assume that all users have been registered in alphabetical order, as done in the test class `DNABookTest.java`.

Submit your file `DNABook.java` to Moodle. If you are completing part b) as well as part a) only the optimised version needs to be submitted.