Test 1: CPS 006G

Owen Astrachan

September 29, 2004

Name: ____________________________________________________________

Login: ______________

Honor code acknowledgment (signature) ____________________________

<table>
<thead>
<tr>
<th>Problem</th>
<th>value</th>
<th>grade</th>
</tr>
</thead>
<tbody>
<tr>
<td>Problem 1</td>
<td>24 pts.</td>
<td></td>
</tr>
<tr>
<td>Problem 2</td>
<td>20 pts.</td>
<td></td>
</tr>
<tr>
<td>Problem 3</td>
<td>10 pts.</td>
<td></td>
</tr>
<tr>
<td>Problem 4</td>
<td>18 pts.</td>
<td></td>
</tr>
<tr>
<td>TOTAL:</td>
<td>72 pts.</td>
<td></td>
</tr>
</tbody>
</table>

This test has 14 pages, be sure your test has them all. Do NOT spend too much time on one question — remember that this class lasts 75 minutes.

In writing code you do not need to worry about specifying the proper import statements. Assume that all libraries and packages we’ve discussed are imported in any code you write.
PROBLEM 1:  \(\text{(What's what (24 points))}\)

Part A (12 points)

There are six lines printed by the statements below in \texttt{main}. Briefly explain why the output shown is generated. You can do this with a few words per line. Be precise, but complete in your explanation. Explain each line.

```java
public class Foo {
    public static void main(String[] args){
        int ai = 10;
        int bi = 3;
        int ci = Integer.MAX_VALUE;
        int di = -1;

        double ad = 10.0;
        double bd = 3.0;
        double cd = 0.0;

        System.out.println(ai/bi);
        System.out.println(ai%bi);
        System.out.println(ad/bd);
        System.out.println(ad/cd);
        System.out.println(ci+1);
        System.out.println(Math.sqrt(di));
    }
}
```

The output is

3
1
3.333333333333333
Infinity
-2147483648
NaN
Part B (6 points)
Recall that the start codon for protein-coding is "ATG". In the sequence below the start codon is found if we use an ORF of 2, but not for an ORF of 1 or 3.

"CATGCACCTTTCACTAGCA"

Similarly, in the following sequence the start codon is found if we use an ORF of -1.

"CTAGCACTTTGATAGCA"

Briefly explain what an ORF is and why six ORFs are typically tried when looking for protein-coding regions in DNA sequences.
Part C (6 points)
A year is a leap year when it has 366 days instead of 365 days. In the international Gregorian calendar a year is a leap year according to the following rules as obtained from www.timeanddate.com/date/leapyear.html

1. Every year divisible by 4 is a leap year
2. But every year divisible by 100 is NOT a leap year
3. Unless the year is also divisible by 400, then it is still a leap year

This means 1800, 1900, and 2100 are not leap years but 2000 and 2004 are leap years.

Which of the following implementations of a method isLeap returns true if year is a leap year and false otherwise. Circle those that are correct. If a method is not correct, provide a value for year for which it returns the wrong value.

```java
public boolean isLeap(int year){
    if (year % 400 == 0) return true;
    if (year % 100 == 0) return false;
    if (year % 4 == 0) return true;
    return false;
}
```

```java
public boolean isLeap(int year){
    return year % 400 == 0 || (year % 4 == 0 && year % 100 != 0);
}
```

```java
public boolean isLeap(int year){
    if (year % 100 == 0) return false;
    if (year % 400 == 0 || year % 4 == 0) return true;
    return false;
}
```
PROBLEM 2: (I’ve got a URL Jones (20 points))

Each URL on the world-wide web has a corresponding Internet Protocol (IP) address. For example, the IP address of www.duke.edu is 152.3.233.20. The table below shows some URLs and the corresponding IP address.

<table>
<thead>
<tr>
<th>URL</th>
<th>IP-address</th>
</tr>
</thead>
<tbody>
<tr>
<td><a href="http://www.parking.duke.edu">www.parking.duke.edu</a></td>
<td>152.3.9.199</td>
</tr>
<tr>
<td><a href="http://www.pastoralleadership.duke.edu">www.pastoralleadership.duke.edu</a></td>
<td>152.3.90.245</td>
</tr>
<tr>
<td>genomics.duke.edu</td>
<td>152.3.232.39</td>
</tr>
<tr>
<td><a href="http://www.events.duke.edu">www.events.duke.edu</a></td>
<td>152.3.9.120</td>
</tr>
<tr>
<td>bigbang.phy.duke.edu</td>
<td>152.3.182.5</td>
</tr>
</tbody>
</table>

In this problem URLs and corresponding IP-addresses are stored in strings separated by one space as shown below.

"www.parking.duke.edu 152.3.9.199"
"www.pastoralleadership.duke.edu 152.3.90.245"
"genomics.duke.edu 152.3.232.39"
"www.events.duke.edu 152.3.9.120"
"bigbang.phy.duke.edu 152.3.182.5"

Part A (4 points)

Write method getIPaddress which returns the IP-address portion of a String representing a URL/IP-address pair. The URL and IP-address are separated by one space. For example, getIP("www.cnn.com 64.236.16.52") returns the string "64.236.16.52".

```java
/**
 * @param s is a URL/IP-address pair, URL separated from IP-address by one space
 * @return the IP-address portion of the String (after the space)
 */
public String getIP(String s){
```
Part B (8 points)
Write method `getAllIPAddresses` which returns an array of IP-addresses extracted from each String in the array `list` which stores URL/IP-address Strings as described above. If `list` represents the five URL/IPs Strings above, the returned array should be as shown below.

`{"152.3.9.199", "152.3.90.245", "152.3.232.39", "152.3.9.120", "152.3.182.5"}`

In writing `getAllIPAddresses` you may call `getIP` from Part A. Assume `getIP` works as specified.

```java
/**
 * @param list is an array of URL/IP-address pairs
 * @return an array of just the IP-addresses represented in list
 */
public String[] getAllIPaddresses(String[] list){
```
Part C (8 points)
Complete the method `findIPAddress` which returns a String representing the IP address corresponding to the URL specified by parameter `url`. The array `list` stores Strings representing URL/IP-address pairs as described above. If the URL specified by parameter `url` is not found in the array return "0.0.0.0" for the IP-address. For example, if `list` represents the data above the calls and values below help explain the method you'll write.

<table>
<thead>
<tr>
<th>call</th>
<th>return value</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>findIPAddress(list,&quot;www.parking.duke.edu&quot;)</code></td>
<td>&quot;152.3.9.199&quot;</td>
</tr>
<tr>
<td><code>findIPAddress(list,&quot;genomics.duke.edu&quot;)</code></td>
<td>&quot;152.3.232.39&quot;</td>
</tr>
<tr>
<td><code>findIPAddress(list,&quot;cs.duke.edu&quot;)</code></td>
<td>&quot;0.0.0.0&quot;</td>
</tr>
</tbody>
</table>

Complete the method below. You may call methods specified in Part A and Part B. Assume they work as specified.

```java
public String findIPAddress(String[] list, String url) {
```

PROBLEM 3: *(The Long and Short of it (10 points))*

In this problem assume that a method convertAll is implemented. It returns an array of all the proteins in a dna-sequence (you may have written a similar method for extra credit, and we discussed a similar method in class.) The returned array may have zero proteins or 100 proteins. In the version of convertAll below, an ORF is specified.

```java
/**
 * Returns all proteins found in dna for specified orf
 * @param dna is the sequence from which proteins are obtained
 * @param orf is the open reading frame used in finding proteins
 */
public String[] convertAll(String dna, int orf){
    // implementation not shown, but convertAll can be called
}
```

Part A (6 points)

One part of determining the best ORF for a protein is the number of proteins coded using the ORF. Write a method bestORF which returns which of the ORFs 1, 2, or 3 is the best for a particular DNA sequence. The best ORF is the one for which the most proteins are found. You may call convertAll shown above, assume it works as specified.

```java
/**
 * Returns the best ORF for dna, where best decided by which ORF codes the most proteins
 * (considering only ORFs 1,2,3)
 * @param dna for which the best ORF is found
 * @return the best ORF
 */
public int bestORF(String dna){
    // implementation of bestORF method
}
```
Part B (4 points)
An alternative definition of best ORF is based on which ORF produces a group of proteins with the maximal median length. So, an ORF which codes proteins whose lengths are 20, 20, 50, 60, 70 (median length is 50) is better than an ORF which codes proteins whose lengths are 10, 10, 15, 15, 20, 20, 30, 30, 50, 50, 70 (median length is 20). The median is the middle value of a sorted list.

Don’t write code, but describe the code/method you’d write to determine the best ORF using these criteria. Be precise, but don’t write code.
PROBLEM 4: (Complementary Proteins 18 points)

In this problem you’ll need to use the class **ProteinTool** shown below by calling the static methods appropriately in solving problems related to protein *hydrophilicity* – the extent to which proteins have an affinity for water.

```java
public class ProteinTool {

    public static double hydrophilicity(String p) {
        // implementation not shown
    }

    public static double hoppWoodValue(String p) {
        // implementation not shown
    }
}
```

no code to write on this page
Part A (6 points)
Write the method `hydroCount` which calculates the number of proteins whose hydrophilicity is *above* a specified threshold and returns this number. The array parameter `plist` is an array of proteins, parameter `thresh` is the minimal value for which a protein should be considered hydrophilic. For example, if `plist` stores 10 proteins whose hydrophilicity indexes (as obtained from `ProteinTool.hydrophilicity`) are as follows:

\[
\begin{array}{cccccccc}
2.0 & 1.5 & 2.2 & 1.7 & 2.3 & -1.5 & -3.0 & -2.0 & 1.5 & 1.8 \\
\end{array}
\]

then the call `hydroCount(plist,-2.5)` should return 9, the call `hydroCount(plist,2.1)` should return 2 and the call `hydroCount(plist,2.8)` should return 0. In writing `hydroCount` you’ll need to call `ProteinTool.hydrophilicity` to determine a protein’s hydrophilicity.

```java
/**
 * Return number of proteins in plist with hydrophilicity above thresh
 * @param plist is an array of proteins
 * @param thresh is the hydrophilicity threshold
 */
public int hydroCount(String[] plist, double thresh){
```

Part B (6 points)
The hydrophilicity index of a protein is calculated using a sliding window. The window’s size specifies the number of amino acids that contribute to the average. For example, for the protein DEGFPPVCRDAE using a window of size 6 we see seven different hydrophilicity indexes/averages depending on which window is used for the calculation. The values for each amino acid are obtained using the hoppWoodValue method in ProteinTool.

<table>
<thead>
<tr>
<th></th>
<th>3.0</th>
<th>3.0</th>
<th>0.0</th>
<th>-2.5</th>
<th>0.0</th>
<th>0.0</th>
<th>-1.5</th>
<th>-1.0</th>
<th>3.0</th>
<th>3.0</th>
<th>-0.5</th>
<th>3.0</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Ultimately, a protein’s hydrophilicity is the average of the highest and lowest window-averages for a given window size.

For the protein DEGFPPVCRDAE there are seven size-six windows as shown. The minimal window-average is -0.833 and the maximal is 1.0. The method windowAverage should return the average obtained from the amino-acids in a window whose starting and ending indexes are specified for a protein as shown below.

<table>
<thead>
<tr>
<th>method call</th>
<th>window values</th>
<th>average</th>
</tr>
</thead>
<tbody>
<tr>
<td>windowAverage(p,1,6)</td>
<td>3.0 0.0 -2.5 0.0 0.0 -1.5</td>
<td>-1.0/6 = -0.166</td>
</tr>
<tr>
<td>windowAverage(p,6,11)</td>
<td>-1.5 -1.0 3.0 3.0 -0.5 3.0</td>
<td>6.0/6 = 1.0</td>
</tr>
</tbody>
</table>

Complete method windowAverage

```java
/**
 * Returns window average in window whose start and end indexes are parameters
 * both are valid indexes for parameter p which represents the protein
 * @param p is the protein whose window-average is calculated
 * @param start is first index of the window
 * @param end is the last index of the window
 * @return the window average
 */
public double windowAverage(String p, int start, int end){
    
```
Part C (6 points)
Recall that a protein’s hydrophilicity is the average of the highest and lowest window averages for some specific window size. Complete the method `hydrophilicity` below so that it works as specified. You should add code where appropriate and fill in missing parameters and loop parts to complete the method. You should not remove any code provided for you, use the code in completing the implementation.

```java
/**
 * Returns hydrophilicity for a specific window-size by averaging the minimal and maximal averages over all window sizes.
 * @param protein is the sequence of amino acids for which hydrophilicity is calculated
 * @param windowSize specifies number of amino acids contributing to window average
 * @return hydrophilicity
 */
public double hydrophilicity(String protein, int windowSize) {
    double minAvg = Double.MAX_VALUE;
    for(int start = 0; start <= ; start++) {
        double avg = windowAverage(protein, start, );
        minAvg = Math.min(avg, minAvg);
    }
    double maxAvg = windowAverage(protein, 0, );
    return (minAvg + maxAvg) / 2;
}
```