The 2007 Iverson Computing Science Competition
June 9, 2007

Name: _____

Question	Difficulty Level	Mark
1	1	
2	2	
3	1	
4	2	
5	2	
6	1	
7	2	
TOTAL		

Overview:

Time and Place: 9:00 to 11:00 AM in Room 3-33, Computing Science Centre, University of Alberta

Format:

This is a traditional 2 hour paper and pencil exam. It is divided into two sections: *required*, and *selected*. All answers, including rough work, are to be written in this booklet.

Everyone should do the two required questions. The required questions do not require any particular programming language or advanced knowledge. They test your ability to read and understand computation. Please ask for assistance if you do not understand some aspect of the question, or the language.

The selected questions section constitutes the main part of the competition. Do as many of these questions as you can, in any order that you choose. Each question is marked with a difficulty level of 1 or 2, where 1 is less difficult than 2. The level is based on the opinion of our question reviewers, and so is probably not that accurate. There are two possible relative weights for questions: a question of difficulty level 2 is worth 1.3 times a question of difficulty level 1.

Since this is the first time we have conducted this exam, there are more questions than can possibly be answered in the time limit. Do not expect to complete all the questions.

Programming Language:

The programming questions can be answered using any programming language you wish (for example, VB, C/C++, Java, or Perl). Minor syntax errors will be ignored. Put in comments where needed to clarify your code.

Suggestions:

1. Read the problem descriptions carefully. To get full marks all program specifications have to be met. You can assume that only valid input values will be entered by the user. You do not need to include out-of-range or data type error checks in the input section of your programs.

2. Sample executions of the desired program have been included for each problem. Review them carefully to make sure you haven't missed any specifications and to get hints as to how to proceed. In these sample executions the sample data entered by the user are underlined.

3. It is recommended that you rough out your program prior to writing any code. Use pseudo-code, diagrams, screen displays, tables or any other aid to help you plan your code. We will be looking at your rough work. Thus you don't have to have a completely coded solution in order to get substantial marks. We are looking for the key computing idea, not specific details which are impossible to remember. In particular you can introduce "built-in" functions for reading a the next number, or the next word, or the whole next line.

4. Take a look at all of the questions before deciding on the ones to attempt as you will not likely be able to finish all of them.

Required Questions

Question 1 (Required) - Difficulty Level 1 – Buggy Code

Modern technology enables us to obtain various kinds of geographical data, thus enabling us to construct accurate maps of the earth. With this data we can compute many interesting things, such as the amount of water in a lake, size of an island, length of a shore, and so on.

Geographical information is typically arranged in a grid of cells, where each cell contains some data associated with the (x,y) coordinate of the cell.

For example, here is a grid map of a small lake with an island in it, with depth measurements taken at intervals of 1m. For example, the depth at position (2,1) is 3m. At (1,1) the value is -1 indicating the presence of land 1m above the water.

y					
3	1	1	2	3	2
2	2	-2	-2.5	3	1
1	1	-1	3	2.5	2
0	2	1	3.5	3	2
	0	1	2	3	4
			x		

This information can be accessed via a function called **GetDepth.** The value of **GetDepth(x,y)** is the depth of the lake (in m) at the grid point (x,y). Note that the depth cannot be 0. It is either positive if the grid square is water, or negative if the grid square is land. There is no limit on the values of x and y that you supply to **GetDepth**, imagine you are on an infinitely large flat earth.

Problem: The following part of a program is supposed to compute the approximate volume of the lake using the simple approach of adding up the volumes of each 1m x 1m column of water at each grid point. For simplicity we know that the ranges of x and y are as in the data above (0 <= x <= 4, and 0 <= y <= 3)

```
int x;
int y;
float Volume = 0.0;

for (x=0; x <= 4; x = x+1) {

    for (y=0; y <= 3; y = y+1) {

        Volume = Volume + GetDepth(x,y);

    }
}
```

When the program is run on the depth data above, it computes 29.5 cubic m as the volume of the lake, but we expected 35 cubic m.

Task: Fix the program by writing your changes directly on the code above. You can either modify the code directly or use comments to indicate what you would change.

Question 2 (Required) – Difficulty Level 2 – String Edit Machine

There are many kinds of programming languages and computation styles. Some are very general (like Java and C++), others are specifically designed for certain kinds of problem. In this question we are going to explore the String Edit Machine (SEM).

A SEM machine or program is very simple. The machine contains only one variable, called S (for state) which is a character string of potentially unlimited length. When the machine starts, S contains the input, and when the machine stops S contains the output.

During computation, the value is S is altered by a program that consists of an ordered list of editing rules. Each editing rule says how to replace a substring of S by another substring. A typical editing rule looks like:

Rule 1: "xyz" => "ab"

This rule says to replace the first "xyz" substring of S by the substring "ab". So if S was:
"dogxyzoutxyz"
then after applying Rule 1 above, S becomes
"dogaboutxyz"
and after one more use of Rule 1, S becomes
"dogaboutab"
Then, since the substring "xyz" is no longer present in S, Rule 1 no longer applies.

The replacement string on the right hand side of the => can be the empty string "", which means to just delete the characters from the left hand side and do not insert any new ones.

A SEM program consists of a bunch of these rules in an ordered list. The program is run by going through the list in order, attempting to find the first rule that applies, and using it to edit S. After applying that rule, the process is restarted at the beginning of the list of rules. If no rule applies, the process stops. It is possible that a rule always applies and the program never stops.

Example: Here is a simple program that takes input S as any string of "X" and "Y" characters, and edits S to move all the "X" to the left and all the "Y" to the right, and then changes all the "Y" characters to Z:

Rule 1: "YX" => "XY"
Rule 2: "Y" => "Z"

If you run this program on the input S = "XYXYXXY" you get the following sequences of edits:

"XYXYXXY"
Apply Rule 1: "XXYYXXY"
Apply Rule 1: "XXYXYXY"
Apply Rule 1: "XXXYYXY"
Apply Rule 1: "XXXYXYY"
Apply Rule 1: "XXXXYYY"
Note: At this point, Rule 1 no longer applies, so Rule 2 is the next one to try.
Apply Rule 2: "XXXXZYY"
Apply Rule 2: "XXXXZZY"
Apply Rule 2: "XXXXZZZ"
Note: At this point, no rule applies, so the machine stops.

Task: Write a SEM program (an ordered list of rules as in the previous example) that tells you whether the original input string has the same number of "X" and "Y" characters in it. The original string S is guaranteed to contain only "X" and "Y" characters. When the machine stops, if S is the empty string "" then the original S had the same number of "X" and "Y" characters; and if S is "0" then the original S had a different number of "X" and "Y" characters.

For example, started with "XYXYXXY", the program stops with S as "0".
But if it started with "XYYX", the program stops with S equal to "".

Selected Questions

Question 3 – Difficulty Level 1 - The Sign of the 'X'

A somewhat inept local graffiti artist can only draw dots of paint. Known as Mr. X, he leaves the letter X in various sizes around town. The size of the 'X' varies depending on circumstances, but always fills a square grid.

Task: Write a program that allows Mr. X to enter the length of a side of a the grid and then generate the pattern of dots that he should paint. The size is to be an odd positive integer no smaller than 3. Hint: think about the relationship between the horizontal and vertical positions of the * characters.

Note values entered by the user are underlined. Assume that only valid input values will be entered.

A Sample run of the program

```
N ? 5

*   *
 * *
  *
 * *
*   *
```

Question 4 – Difficulty Level 2 - Text Compression Using a Dynamic Dictionary

Dynamic dictionary coding is a type of data compression that reduces size by replacing duplicated words with numbers. In dictionary coding each word in a text is replaced by a number that represents that word's place in a list or "dictionary". In the dynamic version of dictionary coding the dictionary or list of words is created from the text being compressed.

In this approach the program begins by creating an empty "dictionary". Then the program processes the text word by word from start to finish. For each word the program first scans its dictionary to see if the word is present. If the word is already present in the dictionary it is replaced with the number of that word's position in the dictionary. If the word is not in the dictionary, it is left as is in the text and a copy of the word is added to the end of the dictionary.

Task: Write a program that accepts lines of text from the keyboard. As the line is read, the dynamic dictionary is updated, the line of text is compressed, and then displayed. This process continues until a blank line is entered and the program stops. You can assume that no numbers or punctuation will appear in the text and that all letters will be in lower case. You may also assume that words are 15 or fewer letters in length, and that the input line is at most 80 characters long.

Here is a sample run of the required program. Note how the second line adds to the existing dictionary, and does not start fresh. **Note: Data entered by the user are underlined. Assume that only valid input values are entered.**

```
Dynamic Dictionary Compression

Text?
water water every where but where is the boat

Compressed text:
water 0 every where but 2 is the boat

Text?
is the row boat floating on the water
Compressed text:
4 5 row 6 floating on 5 0

Text?

_
Bye
```

Question 5 – Difficulty Level 2 - Finite-State Machines

A finite-state machine (FSM) is another kind of computation device. Although you don't actually see them very much, finite state machines are inside almost any device that does something interesting: kitchen appliances, remote control toys, simple phones (the complex ones have full computers), and so on.

There are three primitive concepts associated with a FSM, each associated with part of a FSM diagram:

State – a circle that indicates the current settings of the parts of the machine. Typically the state has a label that tells you what it means.

Transition - a potential change from one state to another (possibly the same) state. The transition is labeled with the events that cause the transition to occur.

Event - something that causes a transition. For example, pushing a button, or reading a character from the input, or the ticking of a clock are events.

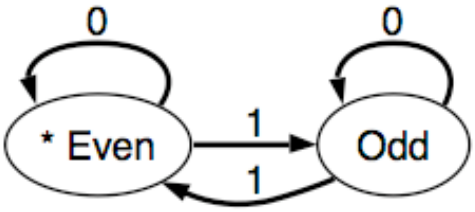
A FSM does a *computation* by starting in a specific start state, and then waiting for events to occur, the events cause the machine to change states. It stops when there are no more events.

A good physical intuition for these notions is: A state is a possible location that you can be at on a map. When you are at one location a transition is a road that can take you to another location. An event causes you to choose a particular road to travel. A computation is a road-trip, or path, between a start location and an end location.

So when you trace a FSM computation, think of putting a marker (like a coin) on the current state, and when an event occurs, moving the marker to the next state along the transition that matches the incoming event.

Examples:

Example 1: Here is a FSM that processes two possible kinds of events: 0 and 1

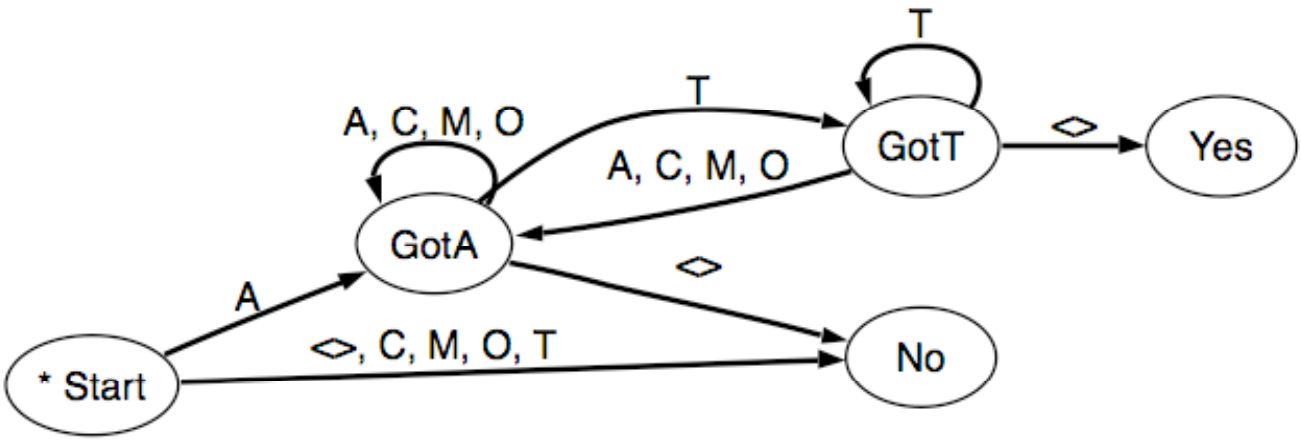


This FSM starts in state Even (the * means start here). Every time a 1 event occurs the FSM makes a transition to the other state. But every time a 0 event occurs the FSM makes a transition back to the same state. Thus, the name of the current state tells you whether the number of 1's that have arrived so far is odd (Odd) or even (Even).

Example 2: One of the most common applications of FSMs is in string pattern matching. The characters of a string come in one by one as events, starting with the left most character of the string. The end of the string is signaled by the special event <>. Pattern machine machines have a starting state called Start; a Yes state that signals if the pattern is found in the string; and a No state if the pattern is not present. Other states are then added as necessary. When the <> event occurs, this means no more events will arrive, so the machine can stop. It can stop earlier if it chooses.

For example, suppose the possible characters are A, C, M, O, T. Here is a machine that looks for strings that start with A and end with T. On the string ATOOT this machine makes the following transitions:

Start --A--> GotA --T--> GotT --O --> GotA --O--> GotA --T--> GotT --<>--> Yes



Task: Assume that, like the previous example, the only characters are A, C, M, O, T. Draw the diagram for a pattern matching machine that can detect when the input string contains the sequence CAT but **not** the sequence ATOM.

For example, on these strings the machine says Yes: CAT, MOOCATCAT, MCCATTOO

While on these strings it says No: ATOMCAT, CATOM, ATOM, MOTO, OOCATOOATOMOO

Question 6 – Difficulty Level 1 – Robot Navigation

Suppose that you have a remote controlled robot that follows instructions in a simple Robotic Command Language (RCL). There are only three commands in the RCL. They are F (Forward), L (Left Turn) and R (Right Turn). The Forward command is always followed by a number indicating how many units the robot is to move. The robot can move 1 or 2 units. The L and R commands cause the robot to turn 90 degrees to the Left and 90 degrees to the Right respectively.

Here is a simple program that makes the robot travel in a rectangular path, reverse itself, retrace its steps and turn to point in its original direction:

F1-R-F2-R-F1-R-F2-L-L-F2-L-F1-L-F2-L-F1-R-R

In general, the human operator can issue a long sequence of commands to the robot that make it wander about. Eventually the robot needs to get back to the base where it started. This could easily be done by simply reversing the sequence of commands issued to the robot so that it retraces its steps. But when finished exploring, the robot should get back to base using the shortest possible path. So it is useful to have a program that takes the sequence of commands that have been sent to the robot, figures out where it is, and then finds the shortest sequence of commands to get back home. (Note: the shortest sequence is also the shortest distance.)

Task: Write a program that does the following. It reads in a sequence of commands that takes a robot on an exploration mission. It figures out where the mission ends relative to the starting position, and then outputs a shortest possible sequence of commands necessary to get back to the starting point.

You may assume that the robot always starts its travels facing to the north. On return it should be also be facing north.

The screen input and output for a sample run of the program is given below. **Note: Data entered by the user are underlined. Assume that only valid input values are entered.**

```
Enter the exploration command sequence:
F1-R-F2-L-F1-F2-L-F2-F1-L-F1-R-F2

The command sequence to return to base is:
L-F2-F1-L-F2-F1-L
```

Question 7 – Difficulty Level 2 – Advanced Robot Navigation

Note: This question uses the scenario from Question 6

In Question 6, the assumption was that there were no obstacles for the robot to navigate around on the return home. This is of course unrealistic. If there were obstacles, then the only safe path home might be to retrace all the steps. This is not very efficient. One improvement on simply retracing steps is for the robot to recognize that it has been to the same spot before, and eliminate the sequence of commands that form the loop.

For example, the sequence of commands

F1-R-F2-R-F1-R-F2-L-L-F2-L-F1-L-F2-L-F1-R-F1-R

can be shortened to the sequence

R-F2-L-F1-L-F2-L-F1-R-F1-R

since the initial sequence F1-R-F2-R-F1-R-F2-L-L returns to the starting point with the robot facing East. The sequence can be further simplified to

L-F1-R

Task: Write a program that does the following. It reads in a sequence of commands that takes a robot on an exploration mission. It then outputs a sequence of commands that follows the original path except that all diversionary loops are removed. This path could then be reversed to return back to the starting point, but do not do this in this program. Hint: you can make an initial assumption that there is at most one loop in the path, and explain how you would handle multiple loops.

The screen input and output for a sample run of the program is given below. **Note: Data entered by the user are underlined. Assume that only valid input values are entered.**

```
Enter the exploration command sequence:
F2-F2-R-F1-R-F2-R-F2

The simplified command sequence is:
F2-L-F1
```