

2011 Iverson Computing Science Competition

May 31, 2011

Name _____

School _____

City _____

Grade _____

CS Teacher _____

Are you taking AP or IB Computer Science? (Yes/No) _____

Have you taken Advanced Level courses? These are courses at the 3000 level such as CSE3110 Iterative Algorithms 1.

(Yes/No/Taking one or more now) _____

Please write clearly!

If we cannot read your writing, we cannot give you marks.

Question	Question Name	Part a	Part b	Part c	Total
		Marks out of			
		2	3	5	10
1	Traffic Lights				
2	Pig Latin				
3	Circle on a Chess Board				
4	Bulls and Cows				
TOTAL					

Exam Format

This is a traditional 2 hour paper and pencil exam. It consists of four questions with each question consisting of three parts. Part a) of each question typically asks you to solve a specific instance of the problem by hand, and it is recommended for everyone. Parts b) and c) are more challenging.

Even students who have just started their work in Computing Science should be able to do reasonably well on part a) of each question. Parts b) and c) of each question require some more problem solving skills. Solve as many parts of as many questions as you can.

Programming Language

The parts of the questions that require programming can be answered using any programming language you wish (for example, VB, C/C++, Java, or Perl). You can also use pseudo-code. Be sure to provide an adequate amount of detail so the markers can determine if your solution is correct. **Pseudo-code should be detailed enough to allow for a near direct translation into a programming language.**

Our primary interest is in your higher order thinking skills rather than your code wizardry. A demonstration of logical thinking and systematic problem solving approaches will count for more than a mastery of syntax of one particular language. Still, you will have to use some type of programming language or pseudo-code to demonstrate what you can do. Minor syntactical errors will be ignored. Add comments where needed to clarify your code.

Suggestions

1. Read the problem descriptions carefully. To get full marks you have to meet all problem specifications for the parts of the questions you attempt. You can assume that only valid input will be entered by the user. You do not need to include out-of-range or data type error checks for the input to your programs.
2. Where feasible, sample executions of the desired program have been included. Review the samples carefully to make sure you understood the specifications. The samples can also give you hints as to how to proceed.
3. We strongly recommend that you do some design work prior to writing any code. Use pseudo-code, diagrams, tables or any other aid to help you plan your code. We will be looking at your rough work and you can get marks for it. We are looking for the key computing ideas, not specific coding details. In particular you can invent your own “built-in” functions for subtasks such as reading the next number, or the next character in a string, or loading an array. Just make sure you specify those functions by giving a relationship between their inputs and outputs.
4. Take a look at all of the questions before deciding which ones to attempt, and in which order. Start with the easiest parts of each question. It is OK to do the questions out of the presented order.
5. Make sure to include English language comments to explain non-obvious or “clever” parts of your solution.

Question 1: Traffic Lights

Eight traffic lights in the small city of Iversonia are controlled by a central computer system. Special messages are sent to each traffic light to change it to a new status, one of **red**, **green**, **yellow** and **off**. Each message consists of five binary digits (bits) in the following format: the first three bits identify the traffic light. The eight lights have codes 000, 001, 010, 011, 100, 101, 110 and 111. The last two of the five bits in a message encode the status that the light should switch to: 00 = **off**, 01 = **red**, 10 = **yellow**, 11 = **green**. For example, the message 01110 can be seen as consisting of two parts, 011 and 10. It changes light 011 to **yellow** (code 10). At the start, the status of each traffic light is **off**.

Question 1, Part a:

a1) What is the status of all traffic lights after the computer sends the following messages:

01101 01001 01010 11010 10101 11011

Write the answers in the table below, i.e. if a light is **off**, then write **off** in the corresponding cell.

light	000	001	010	011	100	101	110	111
status								

a2) Write a sequence of messages that sets all traffic lights in the city to green assuming we start in the initial state when all lights are **off**.

Question 1, Part b:

b1) The *state* of the whole traffic system is given by the status of all lights. For example, here is one possible state of all the traffic lights.

light	000	001	010	011	100	101	110	111
status	red	yellow	off	off	red	red	green	green

How many different states for the whole traffic light system are possible? Two states are considered different if the status of at least one of the traffic lights differs.

b2) Would the message system above still work if the city installed a ninth traffic light? If yes, show how to send information to the new light. If no, how would you redesign the message system to account for the new light?

In order to answer Part c) of Question 1, you will need some

Basic terminology for finite state machines

A *finite-state machine* (FSM) is a kind of computation device. Although you don't usually encounter them directly, finite-state machines exist inside devices such as kitchen appliances, remote control toys, simple phones (the complex ones have full computers), and so on.

FSM can be represented by a diagram. Three basic concepts are associated with parts of an FSM diagram.

A **state** is written as a circle and typically each state has a label that tells what it means for the FSM to be in the state.

A **transition** is written as an arrow and indicates a potential change from one state to another or possibly the same state. The transition is labeled with the events that cause the transition to occur.

An **event** is a cause of a transition. For example, pushing a button, or reading a character from the input, or the ticking of a clock are events.

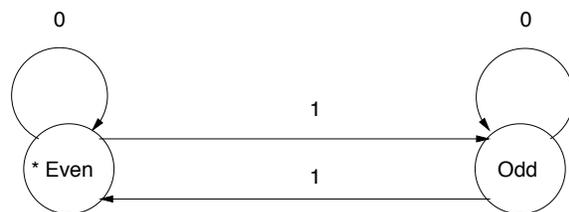
An FSM does a computation by starting in a specific *start state*, marked with a *, and then waiting for events to occur. Each event causes the machine to change state by following the transition that matches the event. The computation stops when there are no more events or an event occurs that does not have a matching transition. In the latter case, we say that the FSM gets stuck.

A state of an FSM is called *accepting* or *final*, written as a double circle in the diagram, if the state corresponds to some desired property of the sequence of events that has occurred so far.

A good physical intuition for these notions is: a state is a possible location that you can be at. 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.

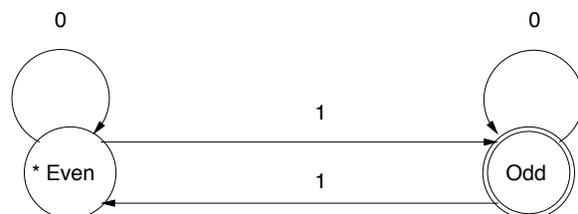
When you trace an FSM computation, think of putting a marker (like a pebble) on the current state, and when an event occurs, moving the marker to the next state along the transition caused by the incoming event.

Here is a diagram of a sample FSM that reads characters from the input and depending whether it reads a 0 or a 1, it switches between states.



This FSM starts in state **Even** (the * means start in this state). Every time a 1 is read, the FSM makes a transition to the other state. But every time a 0 is read the FSM makes a transition 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 (state **Odd**) or even (state **Even**). None of the states is marked as the accepting state.

The machine to the right is similar and accepts inputs with odd numbers of 1's. Note the accepting state **Odd**.



Question 1, Part c:

Consider the original traffic system with 8 lights. Traffic light 011 is not working and the technicians want to find out if any messages are being sent to this light. Your job is to draw a finite state machine (FSM) that reads all broadcast messages sent to all traffic lights, and ends up in an accepting state if and only if there was some message sent to traffic light 011.

For this FSM, an event is reading one character (a single 0 or 1) which has been broadcast from the central computer.

Example 1.

00101 01101 01010 11010 10101 11011

After these messages, your FSM should be in an accepting state, since the second message was for traffic light 011.

Example 2.

00000 11101 01010

Assume the system only received these messages (not the previous ones from Example 1). Then the FSM should not be in an accepting state, since no message was sent to 011.

Extra space

Question 2: Pig Latin

The Pig Latin translation of a word is defined as follows.

- Words that begin with one of the vowels a, e, i, o, u should just have the string ay appended to the end of it.

For example, **apple** becomes **appleay**.

- Words that begin with any other letter should have that first letter removed and appended to the end of the word, and then ay appended after that.

For example, **hello** becomes **ellohay**.

Note that “words” do not have to be valid English words here, they can be any sequence of lowercase letters.

Question 2, Part a:

What was the input that was translated to the following Pig Latin sentence? If there is more than one possible input for a word that produces this translation, then list all such inputs.

histay isay aay hortsay exampleay

Question 2, Part b:

Given the Pig Latin translation of a text, is it possible to translate it back to the original text? If your answer is “yes, always”, then describe how to do it. If your answer is “no, never”, then explain why. If your answer is “sometimes”, then describe exactly when it is possible to translate a word back.

Question 2, Part c:

Write a program that will read in an arbitrary number of lines of text and output its Pig Latin translation. To keep things simple, each line of text will contain exactly one word. Each word is a sequence of lowercase letters. Stop when the input line contains the word `piggy`. Do not translate this last line.

Sample Input

```
good  
morning  
to  
all  
piggy
```

Sample Output

```
oodgay  
orningmay  
otay  
allay
```

Extra space

Question 3: Circle on a Chess Board

A circle with diameter $2n - 1$ has been drawn centered on a $2n$ by $2n$ chess board. The case $n = 3$ is illustrated below.

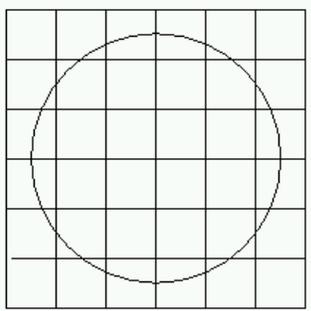


Figure 1: Circle on a chess board.

Given n , you should determine $b(n)$, the number of cells on the board which contain a segment of the circle, and $i(n)$, the number of cells on the board which lie entirely inside the circle.

Note that the circle will never go through an intersection point of the lines on the grid, no matter which integer value n has. (This fact can be proven but we do not show the proof here.)

Question 3, Part a:

Solve the case $n = 2$ by hand. What are the values of $b(2)$ and $i(2)$?

$$b(2) = \qquad i(2) =$$

Question 3, Part b:

This question is about the growth of functions $b(n)$ and $i(n)$. For large n , what do you expect the ratio of the function values for $3n$ and n to be approximately? Justify your answer.

b1) Ratio of cells that contain a segment, $\frac{b(3n)}{b(n)} \approx$

b2) Ratio of cells that lie inside the circle, $\frac{i(3n)}{i(n)} \approx$

Question 3, Part c:

Write a program that solves the *Circle on a Chess Board* problem as follows.

Input and Output

Each line of the input file will contain a positive integer n , $0 < n \leq 150$. For each input value n , write one line of output with the two numbers $b(n)$ and $i(n)$.

Sample Input

3

4

Output for Sample Input

20 12

28 24

Extra space

Question 4: Bulls and Cows

Bulls and Cows is an old code-breaking game, similar to the better known Mastermind game. One player makes up a secret 4-digit code using the digits 0, 1, 2, 3, 4, 5, 6, 7, 8, 9. In a *valid* code, all four digits must be different. For example, 1234, 8093 and 0582 are valid codes, but 1111, 1231 and 0570 are not.

The other player must find the secret code through a series of guesses. To keep things simple, all secret codes and guesses must be valid codes. A guessed digit is called a “bull” if it is a correct digit in the correct location, and is called a “cow” if it is a correct digit, which occurs somewhere in the secret code, but it is in the wrong location in the guess.

Example of a game.

```
(The secret code is 0582)
Guess #1: 1234: 0 bulls, 1 cow (the number 2)
Guess #2: 5678: 0 bulls, 2 cows (5 and 8)
Guess #3: 0928: 1 bull (the 0), 2 cows (2 and 8)
Guess #4: 2850: 0 bulls, 4 cows
Guess #5: 5082: 2 bulls, 2 cows
Guess #6: 0582: 4 bulls
The player has now cracked the code.
```

Question 4, Part a:

Here is the transcript of the start of another game:

```
Guess #1: 1234: 0 bulls, 1 cow
Guess #2: 5678: 1 bull, 1 cow
Guess #3: 5719: 0 bulls, 2 cows
Guess #4: 8607: 1 bull, 0 cows
Guess #5: 9654: 2 bulls, 2 cows
```

Given the five guesses and answers above, what are all the secret codes that are still possible?

Question 4, Part b:

Implement a program which given a secret code and a guess, computes the number of bulls and cows. The program input are two 4-digit numbers: the secret code, followed by your guess. You can assume that both the secret code and the guess are valid codes. The program output should be two numbers: the number of bulls followed by the number of cows in the guess.

Sample Input

0582 1234

Output for Sample Input

0 1

Question 4, Part c:

Write a program that can play Bulls and Cows as a player, and can find out the secret code by making smart guesses. Just like a human would, your program should make repeated guesses, and it can check the number of bulls and cows for each guess until the problem is solved.

Assume your program can call two functions `CountBulls(guess)` and `CountCows(guess)`, which evaluate the number of bulls and cows respectively in a given guess. Your program should continue until it finds the correct code, for which `CountBulls(guess)` returns 4.

`CountBulls(guess)` and `CountCows(guess)` may only be called with valid guesses that have four different digits. For full marks, your program should implement some “smart” form of guessing, not just try all possible guesses from 0123 to 9876. Write some documentation to explain your guessing strategy.

Extra space
