Practical 6: Polymorphism and Complexity

Due Monday by 23:59 **Points** 7 **Submitting** an external tool

General Instructions

All submissions for the practical assignments should be under version control. Review practical 1 for setting up your repository for this practical.

Submission is through Gradescope.

If you get stuck on one of the hidden test cases and really cannot resolve it yourself, please feel free to ask the practical tutors for guidance as to how to fully test your code. You will need to explain what testing you have already done (boundary cases, unit testing, etc)

We encourage you to finish your work before the practical session and take the session as consulting time.

Problem Description

Objective

This practical will give you further practice on polymorphism.
Testing Hint: it's easier if you test things as a small piece of code, rather than building a giant lump of code that doesn't compile or run correctly. As part of your design, you should also sketch out how you are going to build and test the code.

Problem

DNA contains the genetic code that defines the structure of every organism on Earth. The information in this DNA is copied and inherited across generations from individual to individual, but may change over generations due to crossover and mutation. A more successful individual is more likely to survive to breed, increasing the likelihood that it will be able to pass on its particular DNA encoding.

In this practical, we are going to represent an individual with a binary "DNA" strand and mutate it over a number of generations to get a better quality individual. The concepts in this practical are related to Evolutionary Computation, a field of Artificial Intelligence.

Evolutionary computation has been used to solve a number of problems, including making virtual creatures, reducing race time for athletes, designing strategies for satellite coverage, designing

turbines... the list goes on. The following articles provide some kind of overview on evolutionary algorithms. Please have a read if you are interested.
http://www.perlmonks.org/?node_id=298877
http://www.genetic-programming.com/published/usnwr072798.html

Representation of binary strings

In this practical, we use a class called **Individual** to represent the DNA which can be represented by a list of binary digits. **Individual** has a variable called *binaryString* which stores the value of genes.

Your **Individual** class should at least have the following functions:

- string getString() : The function outputs a binary string representation of the bitstring list (e.g.“01010100”).
- int getBit(int pos) : The function returns the bit value at position pos. It should return -1 if pos is out of bound..
- void flipBit(int pos) : The function takes in the position of the certain bit and flip the bit value.
- int getMaxOnes() : The function returns the longest consecutive sequence of ‘1’ digits in the list (e.g. calling the function on “1001110” will obtain 3).
- int getLength() : The function returns the length of the list.
- A constructor that takes in the length of the binary DNA and creates the binary string. Each binary value in the list should be given a value of 0 by default.
- A constructor that takes in a binary string and creates a new Individual with an identical list. Note that this involves creating a new copy of the list.

Smooth Operator

In order to mutate the DNA, we need a class called **Mutator**. The **Mutator** class has a function *mutate* that takes in an **Individual** and an integer index *k* as parameter and returns the offspring after mutation. This function will be defined in the derived classes. You are required to derive three classes from **Mutator**:

- BitFlip: The *mutate* function in this class “flips” the *k*th binary digit. If *k* is greater than the length of the list, we will count in circles. For example, if the length of the list is 10 and *k*= 12, then the mutate function will flip the second digit.
- BitFlipProb: The *mutate* function in this class goes through every digit in the binary string and “flips” each of the binary digit with probability *p*. The probability *p* is of type double and in the range of (0,1). *p* should be given as a parameter of the constructor.
- Rearrange: In this class, the *mutate* function rearranges the list. The function will select the *k*th digit in the bitstring (again, counting in circles). This digit and all digits after it (all the way to the tail) will be moved to the start of the list. For example, if you were rearranging the list (a,b,c,d,e) and *k*= 3, the function would return an Individual with the list (c,d,e,a,b).

In your *main.cpp*, create a function

```
Individual * execute(Individual * indPtr, Mutator * mPtr, int k)
```

which calls the *mutate* function on the Individual object and returns the offspring. Your execute function should decide on which mutator to use based on the actual type of the Mutator.

Complexity

You will not need to submit anything for this part, but think about the computational complexity of your mutate functions. Find the Big-Oh notation for each of these functions. If in doubt, ask your practical supervisors for feedback. It's very important that you be able to answer a simple complexity question like this.

Main function

The test script will compile your code using
g++ -o main.out -std=c++11 -O2 -Wall *.cpp

It is your responsibility to ensure that your code compiles on the university system. g++ has too many versions, so being able to compile on your laptop does not guarantee that it compiles on the university system. You are encouraged to debug your code on a lab computer (or use SSH).
You are asked to create a main function (main.cpp). It takes in one line of input.
binarystr1 k1 binarystr2 k2
Two Individual objects should be created using *binarystr1* and *binarystr2*. The BitFlip mutation and Rearrange mutation are invoked on the first and the second **Individual** with index *k1* and *k2* respectively by calling the *execute* function with the appropriate arguments. The output of your main function should be the two resulting binary string and the longest consecutive sequence of 1 bits of the second offspring. *k1* and *k2* are both positive integers. Please separate the results using one space.

Sample input: 000000 2 0111 2
Sample output: 010000 1110 3
Sample input: 001100 7 011100 3
Sample output: 101100 110001 2

Marking Scheme

- Functionality (7 marks):
  - Matches interface (1 mark)
  - Passing public test cases (3 mark)
  - Passing hidden test cases (3 mark)

Submit Programming Assignment

Upload all files for your submission

SUBMISSION METHOD

GitHub Bitbucket

REPOSITORY

Select a repository...

BRANCH

Select a branch...

Upload Cancel