Assignment 4: Recursion and Polymorphism

Required Reading

Problem Solving with C++

Chapter 14 – Recursion
Chapter 15.3 – Polymorphism

Optional Reading

Chapter 13.1 – Binary Trees (p761)
Chapter 18.3 – Running Times and Big O Notation

Instructions – PLEASE READ *(notice **bold** and underlined phrases)*

This assignment has four parts:

A. Written – Answer the questions, **submit to CourSys**
B. Programming – Code must be well commented, compile/test, then submit
C. Bonus – Optional Programming Assignment for Bonus Marks
D. Submission – Submit specified assignment files by deadline

1. **This is an individual assignment. You may *NOT* work in teams for this assignment. You may discuss ideas with others, but you must answer all questions and write all the code yourself. Plagiarism by students will result penalties that may include receiving zero for the assignment.**

2. **All Files Submitted for Programming Assignments must include block comments for the file, each class, and each function or method. Block comments at the top of each file must include the name of the file plus a short description of what the file does, and the Student's name, Number, and school email address.**

3. **Submission deadline: 11:59pm Monday Apr 11th (last day of class). You should be able to complete the work if you have completed the required reading for the assignment. More material is available in the class Slides. While you have seen most of what is discussed here, some topics discussed in this assignment and needed for your submission may not be seen until another class! You may submit before the deadline if you prefer. You may resubmit again later without penalty provided you resubmit before the deadline.**

4. ***No Late submissions will be accepted*.**

This assignment is due on the last day of class.

A. Written Assignment – Submit in CourSys

Students may discuss the problems with fellow students, but <u>MUST</u> complete the work individually. Submitted answers must be your own work.

This section does not require any programming. Write your answers neatly in a document such as a Word Document that will be submitted to CourSys as part of the assignment.

Your document must be called a4Writeup.doc or a4Writeup.pdf and must include your name, student number, and email address. Use Courier font and double-spacing in your write up. Image files of handwritten work is not acceptable unless otherwise specified.

1. Chapter 14 – Recursion 5 Marks

(a) What is a Recursive Method?

(b) What is Infinite Recursion and how do we avoid it?

(c) When is a base case needed for Recursive Programming?

(d) Is Recursion necessary? Why or why not?

(e) What is Indirect Recursion?

2. Chapter 19 – Trees 5 Marks

(a) What is a Tree?

(b) What is the root of a Tree?

(c) What is a leaf of a Tree?

(d) Define the height of a tree?

(e) Given a complete tree with Height N, how many comparison steps does it take to search for a specified element within the tree?

3. Analysis of Algorithms

Using Big-O notation, analyze the following code fragments and specify what their run-time complexity are. In your write-up, label loops and loop bodies that are included in your calculation with the appropriate Big-O notation. Then show your calculations to compute the overall run-time cost using Big-O notation. Your Big-O notation should be specified in terms of N which is the length of the input array `data`**.**

Reminder: Analyzing the run-time cost of an algorithm first requires that you estimate the cost of the method body using Big-O notation, and then multiply that by the number of times the loop (or loops) is executed with respect to the size of the array N.

(a) Sorting Numbers — 5 Marks

One way to sort an array of numbers is to scan through the array looking for pairs of numbers that are in the wrong order and swapping them. Once all pairs have been compared, and possibly swapped, then the entire array is sorted. Using Big-O notation, analyze the following sort algorithm.

```
// Sort data[] array of length N from smallest to largest
void sort(int data[], int N)
{
 int position;
 int scan;
 int temp;

 for (position=N-1; position>=0; position--)
    for (scan = 0; scan <= position - 1; scan++)
       if (data[scan] > data[scan+1]) {
          /* Elements are in wrong order, swap them */
          temp         = data[scan];
          data[scan]   = data[scan+1];
          data[scan+1] = temp;
       }
    }
```

(b) Searching for a Target in an unsorted Array of Numbers 5 Marks

Searching involves looking for some target in a collection of elements and returning true if the target is found, and false otherwise. When we have an unsorted array of objects, each element must be compared with the target until we find the target element, or until we hit the end of the array.

Examine the following code fragment for searching for a target in an unsorted array of ints of length N. Analyze the algorithm to determine its run-time cost using Big-O notation when it is invoked with min set to 0 and max set to N-1.

```
bool slowSearch(int data[], int min, int max, int target)
   {
        bool found = false;

        for(int i=min; !found && i<=max; i++)
            if (data[i] == target)
               found = true;

        return found;
   }
```

(c) Searching for a Target in a Sorted Array of Numbers 10 Marks

When we have a sorted array, we can search the list much faster by using recursion. To start the search, we would provide the following function which takes the int array data[], the length of the array N, and the `target` to search for:

```
bool search(int data[], int N, int target)
{
  return fastSearch(data, 0, N-1, target);
}
```

Search uses the helper function `fastSearch` to recursively search the array for the target value, passing the ranged to be searched from 0 to N-1. This is the index range of the entire array.

After `fastSearch` compares the target value to the `int` at the midpoint in the array, we can rule out one half of the array based on whether our target is smaller or larger than the value stored in `data[midpoint]`.

The algorithm performs a recursive step to search for the target in the remaining half of the array. Each recursive step involves searching through a portion of the array containing only half the number of elements as the previous step.

The recursion stops when the target is found, or there are no further elements to check between min and max. This occurs when min becomes larger than max.

```
bool fastSearch(int data[], int min, int max, int target)
{
 bool found    = false;
 int  midpoint = (min+max)/2; // determine the midpoint

 if (data[midpoint] == target)
    found = true;
 else
    if (data[midpoint] > target)
    {
     if (min <= midpoint - 1)
        // Recursion on the right half of the array
        found = fastSearch(data, min, midpoint-1, target);
    }
    else
     if (midpoint + 1 <= max)
        // Recursion on the left half of the array
        found = fastSearch(data, midpoint+1, max, target);

 return found;
}
```

B. Programming – To be completed by Students Individually

You must *not* use the vector class for this assignment.

1. Practice with Recursion — 30 Marks

Gain practice with writing Recursive functions by completing the following recursive problems. Remember that designing a recursive function is like forming an inductive proof, design your Base case first then design your recursive case. Solve an instance of the big problem using a recursive solution that breaks it into one or more smaller instances of problem.

You are not permitted to use loops to solve these, only use recursion.

For each problem below, there will be an associated header file which your C++ file must include. For example, in the first problem, the header file factorial.h will be provided, and your solution should be called factorial.cpp.

A single test program called A4B1.cpp will also be provided. It will include the header file A4B1.h that specifies which problems you wanted tested. You may modify the constants in this header file to specify which files you have implemented and want tested. You will submit your .cpp files, but you may not submit the A4B1.h header file.

Example Problem and Recursive Solution:

(x) bunnyEars

We have a number of bunnies and each bunny has two big floppy ears. We want to compute the total number of ears across all the bunnies recursively (without loops or multiplication).

Your function must have the following signature:

```
int bunnyEars(int numBunnies);
```

For example:

```
bunnyEars(0) → 0
bunnyEars(1) → 2
bunnyEars(2) → 4
```

Example Solution:

```
int bunnyEars(int numBunnies)
{
  // Check for Base Case(s)
  if (numBunnies == 0)
     // Base case
     return 0;
  else
     // Recursive case:
     // Solve using solution(s) to smaller subproblem(s)
     return 2+bunnyEars(numBunnies-1);
}
```

(a) factorial

Given n of 1 or more, return the factorial of n, which is n * (n-1) * (n-2) ... 1. Compute the result recursively (without loops).

Your function must have the following signature:

```
int factorial(int n);
```

For example:

```
factorial(1) → 1
factorial(2) → 2
factorial(3) → 6
```

(b) fibonacci

The fibonacci sequence is a famous bit of mathematics, and it happens to have a recursive definition. The first two values in the sequence are 0 and 1 (essentially 2 base cases). Each subsequent value is the sum of the previous two values, so the whole sequence is: 0, 1, 1, 2, 3, 5, 8, 13, 21 and so on. Define a recursive fibonacci(n) method that returns the nth fibonacci number, with n=0 representing the start of the sequence.

Your function must have the following signature:

```
int fibonacci(int n);
```

For example:

```
fibonacci(0) → 0
fibonacci(1) → 1
fibonacci(2) → 1
fibonacci(3) → 2
fibonacci(4) → 3
fibonacci(5) → 5
```

(c) bunnyEars2

We have bunnies standing in a line, numbered 1, 2, 3, etc... The odd bunnies (1, 3, ...) have the normal 2 ears. The even bunnies (2, 4, ...) we'll say have 3 ears, because they each have a raised foot. Recursively return the number of "ears" in the bunny line 1, 2, ... n (without loops or multiplication).

Your function must have the following signature:

```
int bunnyEars2(int numBunnies);
```

For example:

```
bunnyEars2(0) → 0
bunnyEars2(1) → 2
bunnyEars2(2) → 5
```

(d) triangle

We have triangle made of blocks. The topmost row has 1 block, the next row down has 2 blocks, the next row has 3 blocks, and so on. Compute recursively (no loops or multiplication) the total number of blocks in such a triangle with the given number of rows.

Your function must have the following signature:

```
int triangle(int rows);
```

For example:

```
triangle(0) → 0
triangle(1) → 1
triangle(2) → 3
```

(e) powerN

Given base and n that are both 1 or more, compute recursively (no loops) the value of base to the n power, so powerN(3, 2) is 9 (3 squared).

Your function must have the following signature:

```
int powerN(int base, int n);
```

For example:

```
powerN(3, 1) → 3
powerN(3, 2) → 9
powerN(3, 3) → 27
```

(f) sumDigits

Given a non-negative int n, return the sum of its digits recursively (no loops). Note that mod (%) by 10 yields the rightmost digit (126 % 10 is 6), while divide (/) by 10 removes the rightmost digit (126 / 10 is 12).

Your function must have the following signature:

```
int sumDigits(int n);
```

For example:

```
sumDigits(126) → 9
sumDigits(49)  → 13
sumDigits(12)  → 3
```

(g) array6

Given an array of ints and its length, compute recursively if the array contains a 6. We'll use the convention of considering only the part of the array that begins at the given index. In this way, a recursive call can pass in the array of ints and length unchanged, but pass index+1 to move down the array. The initial call will pass in index as 0.

Your function must have the following signature:

```
bool array6(const int nums[], int length, int index);
```

For example:

```
array6({1, 6, 4}, 3, 0) → true
array6({1, 4}   , 2, 0) → false
array6({6}      , 1, 0) → true"
```

(h) array11

Given an array of ints, compute recursively the number of times that the value 11 appears in the array. We'll use the convention of considering only the part of the array that begins at the given index. In this way, a recursive call can pass in the array of ints and length unchanged, but pass index+1 to move down the array. The initial call will pass in index as 0.

Your function must have the following signature:

```
int array11(const int nums[], int length, int index);
```

For example:

```
array11({1, 2, 11}  , 3, 0) → 1
array11({11, 11}    , 2, 0) → 2
array11({1, 2, 3, 4}, 4, 0) → 0
```

(i) array220

Given an array of ints, compute recursively if the array contains somewhere a value followed in the array by that value times 10. We'll use the convention of considering only the part of the array that begins at the given index. In this way, a recursive call can pass in the array of ints and length unchanged, but pass index+1 to move down the array. The initial call will pass in index as 0.

Your function must have the following signature:

```
bool array220(const int nums[], int length, int index);
```

For example:

```
array220({1, 2, 20}, 3, 0) → true
array220({3, 30}   , 2, 0) → true
array220({3}       , 1, 0) → false
```

(j) nestParen

Given a string, return true if it is a nesting of zero or more pairs of parenthesis, like "(())" or "((()))". Suggestion: check the first and last chars to determine if they are '(' and ')' respectively, and then recur on what's inside them.

Hint: string method `substr(pos,len)` will take an existing string and return a newly constructed string object len characters long that contains a copy of the characters starting at index pos. If string s is `"A Good Day!"`, then `s.substr(2,4)` instantiates a new string object with value `"Good"`.

Your function must have the following signature:

```
bool nestParen(const string str);
```

For example:

```
nestParen("")        → true
nestParen("6")       → false
nestParen("(())")    → true
nestParen("((()))") → true
nestParen("(((x))") → false
```

2. Inheritance and Polymorphism 40 Marks

We will explore inheritance and polymorphism through a game of Tic-Tac-Toe. As you recall from lectures, when a child class extends a parent class, that child inherits all the members of the parent class. This is the case with the Player class in this simple Tic-Tac-Toe game.

As implemented, the tic-tac-toe game defaults to playing a single game against a human player and a computer player that plays random moves. As it plays the game, it alternatively waits for either the human player or computer player to select their moves. This continues until the game is complete and a winner is announced, or the game is declared to be a draw.

Using command line parameters, the user can invoke the program and choose which types of players should play (human or computer) and how many games the tournament should last.

The command syntax is defined in the Usage Statement as follows:

The command syntax is:
`TicTacToe PlayerType1 PlayerType2 NumberOfGames wantGUI`

Where:
- PlayerType1 - specifies which Player derived class should play as player 1 (default is human)
- PlayerType2 - specifies which Player derived class should play as player 2 (default is random)
- NumberOfGames - specifies how many games to play in the tournament (default is 1)
- wantGUI – if specified, then we should use a Graphical User Interface (default is no GUI)

Valid choices for PlayerType are: Human, Random, Smarter, Learner
Note that only Human and Random are implemented in the sample implementation

Currently there are two player classes defined:

1) **HumanPlayer – prompts the user to enter moves manually**
2) **RandomPlayer – picks any empty square at random**

Notice that invoking these classes involves polymorphism. While the object reference within Main refers to a Player class, the actual `getNextMove` method called depends upon the class of the player object used at run-time.

You must create a new `Smarter` computer player class for the tic-tac-toe game. You can choose how smart to make this player, but it must move to a winning square if it exists immediately, otherwise it must block an immediate win by its opponent if it can, and if neither case exists, your smarter player must select an empty square based on your move selection algorithm. Your class will be tested by playing a 100-game tournament with the `Random` computer player. Your class must consistently defeat the `Random` player by over 50 times out of 100 to be declared smarter by our test program.

The class `Player` defines an abstract class that represents a tic-tac-toe player. Being abstract, one cannot instantiate an object of this class. One

must create a child class that extends `Player` and provides implementations for all its abstract methods.

The `Player` Class defines only one abstract method. This method accepts one parameter that is the current board position, and it returns the player's next move. The signature for this method is defined in the `Player` Class as follows:

```
virtual Move getNextMove(const Board& board);
```

The Board class represents the 9 positions on the tic-tac-toe board as a 3x3 matrix of ints. The upper left square is board[0][0] while the lower right square is board[2][2]. Each square is numbered from 1 to 9 with position board[0][0] being square 1 and board[2][2] being square 9. An empty square is represented by a zero in the matrix, while a 1 represents a square marked by player number 1 and a 2 represents a square marked by player number 2.

To select a move, Player.getNextMove(board) examines the current board position and selects an empty square to place the player's token in, which thus marks the square as being owned by that player. The returned Move object represents the selected move by the player.

Players take turns moving until either one player wins by getting three squares in a row, or it is declared a draw when all squares are occupied and no player has won.

You may not modify the Player class. The class PlayerTypes contains method `createPlayer` that will instantiate an object instance of your `Smarter` child class when requested to do so via the command line or through the Graphical User Interface or GUI.

The Tic-Tac-Toe program provided contains the following files:

TicTacToe.cpp	**- The main() function for playing the game**
Board.cpp	**- Records the current state of play on the board**
Move.cpp	**- Move object used for creating a new move**
Player.cpp	**- Abstract Class defining Player child class method**
PlayerType.cpp	**- Defines playerTypeNames and `createPlayer`**
HumanPlayer.cpp	**- The Human player class**
RandomPlayer.cpp	**- The Random Computer player class**

You must implement the `SmarterPlayer` class in file `SmarterPlayer.cpp`. Your new class is derived from the `Player` class, and thus your child class will inherit all its public methods and public local variables defined within the `Player` class. You may use them as you see fit within your own child class to help implement your `SmarterPlayer` class.

C. Programming for Bonus Marks – To be completed individually

These programming problems are optional. You may complete them if you choose and will receive bonus marks that may be used to top up your overall Assignment portion of your course grade.

1. Backtracking with Recursion 10 Bonus Marks

The problems in B1 required simple straightforward recursive solutions. The use of recursion for such simple problems may seem inappropriate, as often the iterative solution is simpler, faster, and more intuitive.

So why bother with practicing recursion? Because many complex problems become much easier when solved with Recursion! Problems such as Compiling code, natural language understanding, Chess, The Towers of Hanoi, problems in Bioinformatics, and various optimization and routing problems become much easier to code when we include recursion in the solution.

These problems are all recursive backtracking problems with arrays. See if you can solve them!

As with question B1, for each problem below, there will be an associated header file which your C++ file must include. A test program called A4C1.cpp will also be provided.

(a) groupSum

Given an array of ints, is it possible to choose a group of some of the ints, such that the group sums to the given target? This is a classic backtracking recursion problem. Once you understand the recursive backtracking strategy in this problem, you can use the same pattern for many problems to search a space of choices. Rather than looking at the whole array, our convention is to consider the part of the array starting at index start and continuing to the end of the array. The caller can specify the whole array simply by passing start as 0. No loops are needed -- the recursive calls progress down the array.

Your function must have the following signature:

```
bool groupSum(const int nums[],int length,int start,int target);
```

For example:

```
groupSum(0, {2, 4, 8}, 10) → true
groupSum(0, {2, 4, 8}, 14) → true
groupSum(0, {2, 4, 8},  9) → false
```

(b) groupSum6

Given an array of ints, is it possible to choose a group of some of the ints, beginning at the start index, such that the group sums to the given target? However, with the additional constraint that all 6's must be chosen. (No loops needed nor permitted.)

Your function must have the following signature:

```
bool groupSum6(const int nums[],int len,int start,int target);
```

For example:

```
groupSum6({5, 6, 2}, 3, 0, 8) → true
groupSum6({5, 6, 2}, 3, 0, 9) → false
groupSum6({5, 6, 2}, 3, 0, 7) → false
```

(c) groupNoAdj

Given an array of ints, is it possible to choose a group of some of the ints, such that the group sums to the given target with this additional constraint: If a value in the array is chosen to be in the group, the value immediately following it in the array must not be chosen. (No loops needed nor permitted.)

Your function must have the following signature:

```
bool groupNoAdj(const int nums[],int len,int start,int target);
```

For example:

```
groupNoAdj({2, 5, 10, 4}, 4, 0, 12) → true
groupNoAdj({2, 5, 10, 4}, 4, 0, 14) → false
groupNoAdj({2, 5, 10, 4}, 4, 0,  7) → false
```

(d) splitArray

Given an array of ints, is it possible to divide the ints into two groups, so that the sums of the two groups are the same. Every int must be in one group or the other. Write a recursive helper method that takes whatever arguments you like, and make the initial call to your recursive helper from splitArray(). (No loops needed nor permitted.)

Your function must have the following signature:

```
bool splitArray(const int nums[], int length);
```

For example:

```
splitArray({2, 2}   , 2) → true
splitArray({2, 3}   , 2) → false
splitArray({5, 2, 3}, 3) → true
```

(e) split53

Given an array of ints, is it possible to divide the ints into two groups, so that the sum of the two groups is the same, with these constraints: all the values that are multiple of 5 must be in one group, and all the values that are a multiple of 3 (and not a multiple of 5) must be in the other. Write a recursive helper method that takes whatever arguments you like, and make the initial call to your recursive helper from split53(). (No loops needed nor permitted.)

Your function must have the following signature:

```
bool split53(const int nums[], int length);
```

For example:

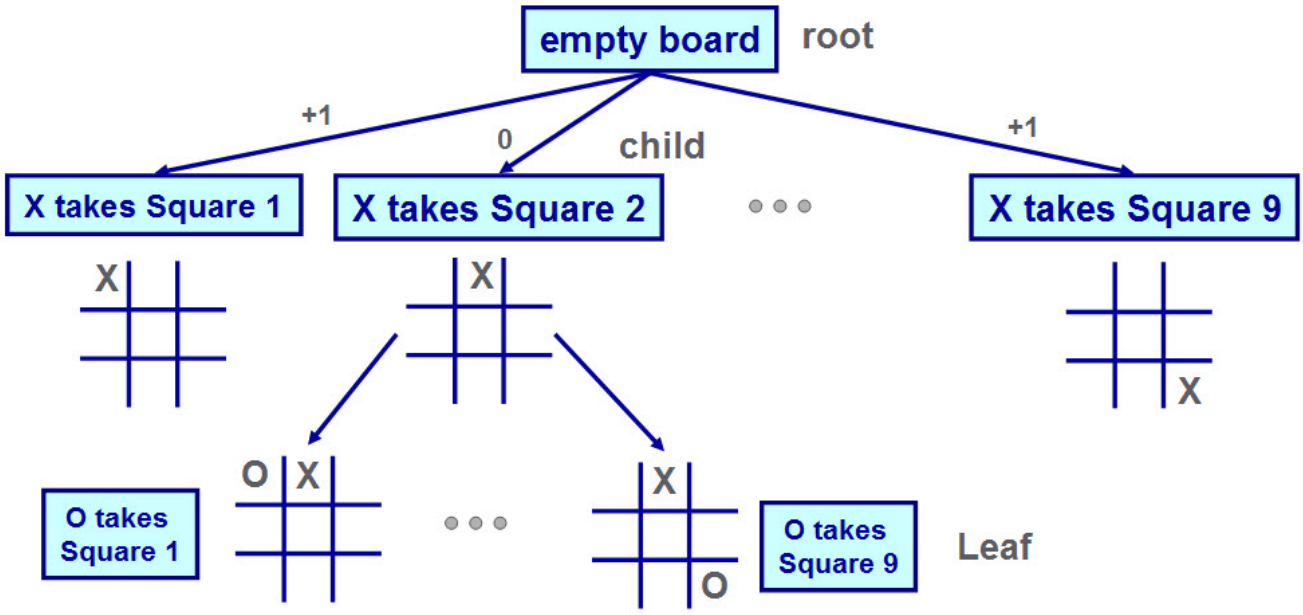
```
split53({1,1}      ,2) → true
split53({1, 1, 1} ,3) → false
split53({2, 4, 2} ,3) → true
```

2. Learning to Play Tic-Tac-Toe 20 Bonus Marks

While it is easy to create a tic-tac-toe program that plays perfectly, it is more interesting to create a player class that *learns* how to play perfectly through trail and error. Artificial Intelligence is a huge field within computer science and there are many interesting ways to develop a learning program.

We will look at a simple method using decision trees that are modified by the program to capture new information. Eventually, these trees will capture all tactical possibilities and your class will play a perfect game of tic-tac-toe.

An int array is a linear structure of elements. A tree on the other hand, creates a two dimensional structure which can be used to store a decision tree for games like tic-tac-toe and even chess. Each element in the array can represent a possible move, and each element can point to another array that points to the set of counter moves by the opponent, and so on. Tree structures made up of arrays like this are called B-Trees. The array at the top of the tree is known as the root node, while the arrays that the root points to are called child nodes. Nodes without any children are called leaf nodes.



For example, at the start of a game there are 9 possible squares to move to. You can create an int array that represents these 9 moves with the int value representing their perceived move strength. For example, one could represent "guaranteed win" by setting the int value to 1, unknown and draw could be set to 0, and "guaranteed loss" by setting the value to -1. This initial node would be the root node of the decision B-Tree.

At the start of your program, you initialize the root node to contain 9 elements with each int initialized to zero. Setting the int to zero indicates that your program currently does not know if the move associated with that square is a winning move, a losing move, or a drawing move.

As your program makes its moves, the number of remaining empty squares on the board where one can move is reduced by one after each move. As these new board positions are encountered, a new int[] array should be created to capture the knowledge of this board position (which will initially be all zeroes) and the parent node should be made to point to this new node. If this new board position was reached by selecting the i^{th} possible move from the previous position, then the i^{th} entry in the parent node array should point to this new node in the tree (to do so will require arrays of objects rather than simple ints).

This process of building the B-Tree can continue until the game is either won, lost, or drawn. At this point, the leaf node contains only a single element and it can be labeled with -1, 0, or 1 for guaranteed loss, draw, or win respectively.

To build on this knowledge, one must then go to the parent node and also record this int value there. If it is marked as a -1, then your program should never make this move again since it is a guaranteed loss. If it is marked with a +1, then your program should always select this move since it is guaranteed to win by doing so.

To complete the algorithm, one must also check if all the choices in the parent node all equal -1 or +1. Assume that they all equal -1, then all such moves at that position are a guaranteed loss, and so the program should avoid this position by avoiding the move that got it here. To do so, we recursively go to the parent node and mark the move that got us here as being bad (-1) as well. If all the moves at that node are marked as bad, repeat the process with its parent and so on.

As your program continues to win or lose, it will continue to update its B-Tree with knowledge about the various board positions until it will eventually play the perfect tic-tac-toe game.

The learning algorithm described above is one possible implementation using arrays and ints. A more elegant solution would involve encapsulating both the B-Tree Nodes and the learned quality of each move into a new Class. Advanced students are encouraged to do so.

While your program learns, convergence may take a long time since the search space for Tic-Tac-Toe is large. The convergence can be accelerated by realizing that decisions are independent of move order, and thus many nodes in the tree can be merged.

Creating a LearnerPlayer class is not easy! If you are not already familiar with tree structures and recursion from previous experience, you may find this too difficult. If the above explanation is not clear or you are not familiar with tree structures and recursion, you should not attempt this question until you have learn more about these concepts in Cmpt 225.

3. Playing with Graphics 20 Bonus Marks

One command line option is to provide the user with a graphical user interface. Implement a graphical user interface for the TicTacToe program that allows the game to be run from a GUI and provides the same options as the command line parameters using buttons and other widgets.

One recommended toolset for creating graphics in C++ on Ubuntu is SFML. It is already installed on CSIL machines. More information and references are listed at the end of this question.

Your GUI should provide radio buttons for selecting the player types, and display statistics on number of games won, lost, and drawn by each player type.

The GUI should provide a MODE button to allow the user to specify whether the tournament should be played continuously or should be paused after each game until a CONTINUE TOURNAMENT button is pressed. This mode only applies to tournaments where both players are computer players. If one or both players are human, the games already pauses and waits for the player to click on the square on which to place the player's token, so there is no need to provide a MODE button in that case.

If the wantGUI flag is set to true on the command line, your code will be invoked to initialize itself. Your code will have an opportunity to re-display the board after every move, and once the game is complete. Interface details will be provided in the TicTacToeGUI.h header file.

The amount of bonus marks awarded will be based on the quality of your GUI. The best GUI's will be presented in class and the very best will win a prize. To be eligible for presentation in class, your GUI must be submitted by April 9th, and an email sent to cmpt-135-help@sfu.ca requesting that your GUI be entered into the competition. You may continue to work on your GUI and resubmit until April 11th, but only the version submitted by April 9th is eligible for prizes.

SFML - A C++ Based Graphics Systems

To install SFML on your home Ubuntu system

1) Install sfml-dev onto Ubuntu from the command line:

```
sudo apt-get install libsfml-dev
```

2) Add this to the Linker "Additional Options" for the project:

```
-lsfml-graphics -lsfml-window -lsfml-system
```

The SFML Homepage: http://www.sfml-dev.org/

SFML and Linux Getting Started Guide: http://www.sfml-dev.org/tutorials/2.3/start-linux.php

Here is the SFML faq: http://www.sfml-dev.org/faq.php

D. Submission – To be completed by Students Individually

Due date: Monday Apr11th 11:59pm (No Late Submissions Accepted!)

Students are responsible for submitting the requested work files by the stated deadline for full marks. It is the student's responsibility to submit on time. Submissions via email will *NOT* be accepted.

You must submit your final version of the following files before the deadline. The written answers for Part A must be submitted in a separate Word or PDF documents which are double spaced and include the Student's Name and Student Number at the top, along with the Course Number and Assignment Number. Students must ensure that all submitted code compiles and is properly commented and formatted for readability.

Submit into CourSys:

Part A	**: a4Writeup.doc, a4Writeup.docx, or a4Writeup.pdf**
Question B1	**: factorial.cpp, fibonacci.cpp, … , nestParen.cpp**
Question B2	**: smarterPlayer.cpp**
Question C1	**: groupSum.cpp, groupSum6.cpp, … , split53.cpp**
Question C2	**: LearnerPlayer.cpp**
Question C3	**: TicTacToeGUI.cpp**

Late Submissions will not be Accepted or Marked for this Assignment.

Files are to be submitted into CourSys under Assignment 4.