CS101 Mid-Semester Examination

General instructions: Time: 2 hours

- There are 6 questions in this exam. Write your answers directly on this paper in the spaces provided.
- Do not use a more general type where a simpler type would work (eg. do not use a **float** type where **int** will work.)
- Follow C++ syntax strictly when completing missing parts of code.

Q1) **[10 + 5 = 15 Marks]** Consider the Peasants' Algorithm for multiplication of two positive integers. It works in the following manner.

- Write the two numbers in two columns. Keep updating according to the following procedure until the number in the first (i.e., left) column becomes 1.
- Halve the number in first column (integer division), double the number in the second column.
- At the end, sum up all the numbers in the second column, for which, corresponding number in the first column is odd.

Example of multiplication using Peasants' algorithm:

1. 13 x 8:

13	8	←
6	16	
3	32	←
1	64	←

Answer: 64 + 32 + 8 = 104

(Note: This algorithm is related to binary multiplication procedure. It can be seen by writing both the numbers in binary form and then applying standard multiplication.)

Now answer the following questions.
(a). Following is the code for calculating multiplication using Peasants' Algorithm. Fill in appropriate blanks. The variable **`num_iterations`** is just to count number of iterations of the while loop. Assume that values of **`num1,num2`** and **`product`** fit in **`int`** type. **[2*5 = 10]**

```
#include <simplecpp>
void peasantMultiplication(int num_in_col1, int num_in_col2,int& answer){
   int numIterations=0;
   while(1){
       numIterations++;
       if(num_in_col1 % 2 == 1) answer = answer + num_in_col2;
       if(num_in_col1 == 1) return;
       num_in_col1 = num_in_col1 / 2;
       num_in_col2 = num_in_col2*2;
   }
}

main_program{

      int num1,num2;
      cin>>num1>>num2;
      int product = 0;
      peasantMultiplication(num1,num2,product);
      cout << product << endl;
}
```

(b). In the following two cases, what will be the final value of num_iterations when the function `peasantMultiplication` returns ? **[2.5 + 2.5 = 5]**

1. peasantMultiplication(32,2): **6**
2. peasantMultiplication(2,32): **2**

Evaluation Instructions:

- **Part (a) : Each blank carries 2 marks.**
- **Part (b) : Both the answers carry 2.5 marks each.**

Q2) .[**14 marks**] Write the output of the code shown below.

```
#include <simplecpp>
int f(int x,int y)
{
      if( x < y)
      {
      return x*x;
      }
      else return y*y*y;
}
int g(int a)
{
      for(int i=0; i<=a; i++)
      {
      if(a == i*i*i)return true;
      }
      return false;
}
int h(int a)
{
  for(int i=2; i*i<=a; i++)
  {
      if(a%i == 0)return true;
  }
  return false;
}
main_program{
   int j=1;
   int x=4;
   for(x=0;x<12;x++)
   {
      if(h(x))continue;
      j++;
      cout<<f(x+1,j)<<" ";
      if(g(j))break;
   }
   cout<<endl;
   cout<<x<<" "<<j<<endl;
}
```

Answer:

1 4 9 16 216 343 512

11 8

Evaluation instructions:

1 4 9 16 : For first 4 numbers in line 1: 4 marks, marks will be awarded only when all the 4 numbers match, else awarded 0 marks.

216 343 512: For next 3 numbers in line 13: 4 marks, marks will be awarded only when all the 3 numbers match, else awarded 0 marks.

11 8: 6 marks. 3 marks for each correct number. Partial marks will be given here if 1 of the 2 blanks is correct.

Q3). **[15 marks]**

The source code for two programs series1.cpp and series2.cpp are given as follows. The code for series2.cpp contains a few blanks.

(a). Complete the blanks with valid C++ expressions (consisting of constants, variables and/or arithmetic operators used in the correct syntax of C++) such that both the programs print the same output, given the same input. You are not allowed to use any math functions like sqrt() or pow(). [**12 marks = 3+ 6+ 3**]

[series1.cpp]

```
#include <simplecpp>

main_program {
	double z;
	cin >> z
	double sum = 0, term1 = 0, term2 = 1, term3 = z;

	for (int i = 1; i <= 10; i++) {
		term1 = i * i;
		term2 *= i;
		if (i % 2 == 1)
			sum = sum + term1 * term3 / term2;
		else
			sum = sum - term1 * term3 / term2;
		term3 *= z;
	}
	cout << sum << endl;
	return 0;
}
```

[series2.cpp]

```
#include <simplecpp>

double series_term(int k, double z) {
	double sign = -1, temp_term = 1;

	for(int j = 0; j < k; j++) {
		temp_term = temp term * z / (j + 1);
		sign *= -1;
	}
	return sign * temp_term * k * k;
}

main_program {
	double z;
	cin >> z;
	double sum = 0;

	for (int i = 1; i <= 10; i++) {
		sum += series_term(i, z);
	}
	cout << sum << endl;
	return 0;
}
```

(b). Write the invariants for variables **term2, term3 and sum** at the start of the loop?**[1+1+1 marks]**

Answer:

1) term2 stores (i-1)! (i-1 factorial) 1 mark (no partial marking)
2) term3 stores z^i (z power i) 1 mark (no partial marking)
3) sum stores sum of (i-1) terms of the series. 1 mark (no partial marking)

Evaluation Instructions:

Part (a).
3 marks - blank 1
6 marks - blank 2
3 marks - blank 3

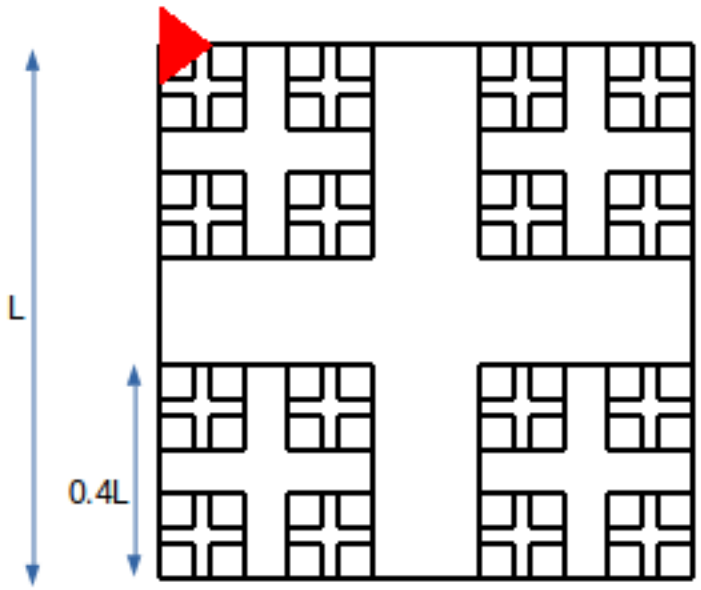
If the expression is mathematically correct and not a valid C++ syntax, half the marks will be awarded.
Marks will be awarded only if the complete blank is matched, else award 0.

Part (b).
Note: For this part, c++ syntax need not be followed

Q4)

[8 + 8 = 16 marks]

Using recursion helps in avoiding duplication of code and makes the code easier to write. In this question, we try to verify it by writing iterative and recursive code for the following image.



Consider the functions **drawSquareIter** and **drawPatternRec**. Both create the same final image. **drawSquareIter** is an iterative function whereas **drawPatternRec** is a recursive one.

(a). Following is the incomplete code for the function **drawSquareIter**. Fill in the blanks. Also, what should be the values of X and Y? **[8 Marks]**

```
#include <simplecpp>
void drawSquareIter(double side){
   repeat(X){
       repeat(X){
             repeat(X){
                   repeat(X){
                         forward(0.064*side);
                         right(Y);
                   }
                   forward(0.16*side);
                   right(Y);
             }
             forward(0.4*side);
             right(Y);
       }
       forward(side);
       right(Y);
   }
}
Value of X = 4
Value of Y = 90
```

(b). Now, following is the incomplete code for function **drawPatternRec**. To draw the given figure, the function will be called with level=4. Value of **sideLen** will be the length of outermost square. **[8 Marks]**

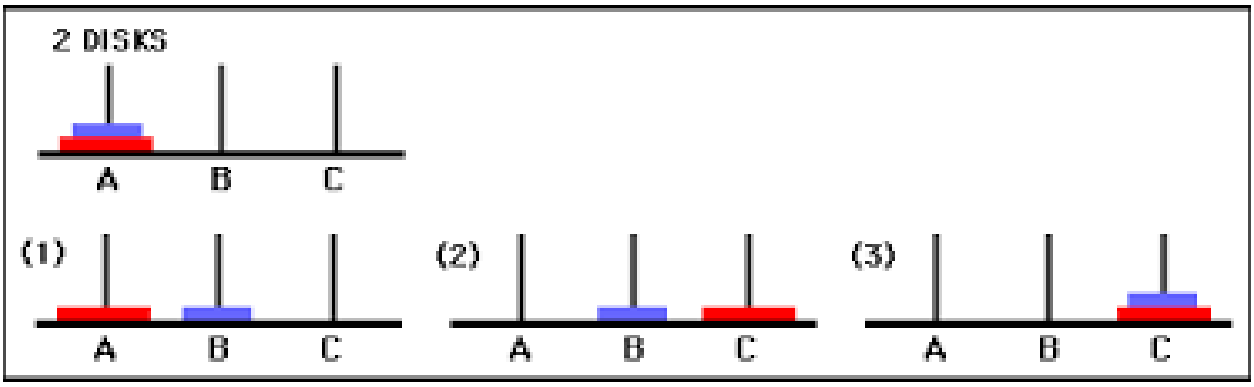
```
void drawPatternRec(double side,int level){
   if(level==0) return;

   repeat(4){
       drawPatternRec(0.4*side,level-1);
       forward(side);
       right(90);
   }
}
```

Evaluation instructions:

- **Part (a). marks distribution:**
  - **Value of X : 1 mark**
  - **Value of Y : 1 mark**
  - **All other blanks : 2 marks each**
- **Part (b). marks distribution:**
  - **All blanks : 2 marks each**

Q5). **[14 Marks]** The Tower of Hanoi is a puzzle which consists of three rods, and a number of disks of different sizes which can slide onto any rod. The puzzle starts with the disks in a neat stack in ascending order of size on one rod, the smallest at the top, thus making a conical shape. The objective of the puzzle is to move the entire stack from first rod to the third rod, obeying the following simple rules:

1. Only one disk can be moved at a time.
2. Each move consists of taking the upper disk from one of the stacks and placing it on top of another stack i.e. a disk can only be moved if it is the uppermost disk on a stack.
3. No disk may be placed on top of a smaller disk.



The above figure is a solution for the Tower of hanoi puzzle consisting of two disks.

E.g: Below are the sequence of steps to solve for a tower consisting of 2 disks.

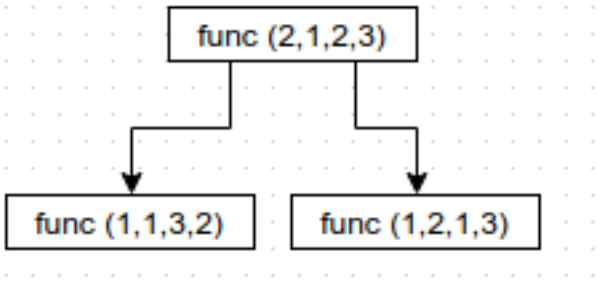
Move the top disk from rod-1 to rod-2
Move the top disk from rod-1 to rod-3
Move the top disk from rod-2 to rod-3

`solve_tower_of_hanoi(int n, int source, int helper, int destination)` is a function which prints the above steps. Here, **source** is the rod where all the disks are initially present in the valid order. Now **destination** is the rod where all disks present on source are to be placed in the same order. **helper** is the remaining rod which can be used as a helper rod for intermediate steps.

Following is the program which solves the puzzle:

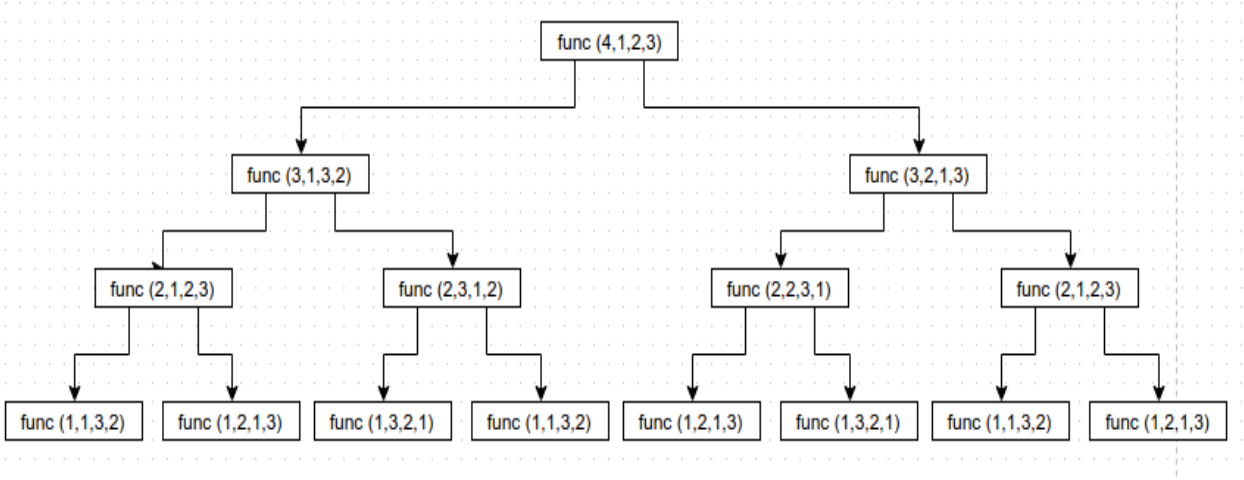
```
#include <simplecpp>

void solve_tower_of_hanoi(int n, int source, int helper, int destination){
	if(n == 1){
		cout << "Move the top disk from rod-" << source << " to rod-" <<
		destination << "\n";
	}else{
		solve_tower_of_hanoi(n - 1, source, destination, helper);
		cout << "Move the top disk from rod-" << source << " to rod-" <<
		destination << "\n";
		solve_tower_of_hanoi(n - 1, helper, source, destination);
	}
}

main_program{
	int n;
	cin >> n;
	solve_tower_of_hanoi(n,1,2,3);
}
```

(a). Draw the call tree for the function call **solve_tower_of_hanoi(4,1,2,3).** Call tree for the function call **solve_tower_of_hanoi(2,1,2,3)** is given below, where the name **func** has been used to indicate the function **solve_tower_of_hanoi** for brevity.



Note that the left function call represented by the left node executes completely before the right function call. This property should be preserved by your solution.

Answer:



(b). Number of function calls of solve_tower_of_hanoi for n = 4 is 15 and closed form expression for a generalized n is 2^n-1.

Evaluation Comments :

(a) 10 marks - 1 mark penalty for each incorrect node. Minimum marks awarded is 0.

Q6). **[Marks 14]** Given below are two recursive functions to calculate the value of nCr. nCr is the number of ways of selecting r items given n items.

The recurrence for calculating ${}^{n}C_{r}$ is given by: ${}^{n}C_{r} = {}^{n-1}C_{r} + {}^{n-1}C_{r-1}$.

Assume that n>=r in this question.

The function '**nCr_erroneous'** has some errors in it and the function '**nCr_correct**' has some missing parts. Read the code below and proceed to answer some questions below.

Code:

```
int nCr_erroneous(int n, int r) {
	if(r==0)
		return n;
	else if(n<=r)
		return n;
	return nCr_erroneous(n-1,r+1) + nCr_erroneous(n-1,r-1);
}
```

Now, answer the following questions:

(a). Give the return value of the nCr_erroneous function for the following function calls

nCr_erroneous(2,0): 2

nCr_erroneous(2,2): 2

nCr_erroneous(5,3): 11

(b) Fill in the provided blanks in the nCr_correct function

```
int nCr_correct(int n, int r) {
	if(r==0)
		return  1;
	else if(n<=r)
		return  1;
	return nCr_correct(n-1, r) + nCr_correct(n-1, r-1);
}
```

Note: Use the outputs from the part (a) as clues for solving this.

Evaluation Instructions:

Part (a): 2 marks per correct answer[2*3 = 6].

Part (b): 2 marks for the first two blanks and 1 mark each for other four blanks[2*2 + 4 = 8]

Note that the two recursive calls may be written in different order.