Section, Name and ID#: [redacted]

AMERICAN UNIVERSITY OF ARMENIA

College of Science and Engineering

CS 120 Introduction to Object-Oriented Programming

MIDTERM EXAM

13

Date / Time: Friday, March 17 2017 at 17:30

Duration: 2 hours

Attention: **ANY TYPE OF COMMUNICATION IS STRICTLY PROHIBITED**

Write down your section, name and ID# at the top of all used pages

Participation:

Problem 1: Consider below a C++ function ***float kahan(float num1, float num2, float& compensation)*** that implements the ***Kahan Summation Algorithm*** for high-precision compensated summation of two float arguments ***float num1*** and ***float num2***:

```
float kahan(float num1, float num2, float &compensation)
{
        float result;
        num2 -= compensation;
        result = num1 + num2;
        compensation = (result - num1) - num2;
        return result;
}
```

Using this function, write a C++ function ***float pi(int n)*** that computes the value π by the following formula:

$$\pi = 2\sum_{k=0}^{n}\frac{(2k-1)!!}{(2k)!!(2k+1)} = \frac{2}{1*1} + \frac{1}{2}*\frac{2}{3} + \frac{1*3}{2*4}*\frac{2}{5} + \frac{1*3*5}{2*4*6}*\frac{2}{7} + \cdots$$

Recall that **n!!** is the product of odd numbers from *1* to ***n***, if ***n*** is odd; and is the product of even numbers from ***2*** to ***n***, if ***n*** is even. The double factorial of non-positive numbers equals to ***1*** by definition.

The initial value of ***float compensation*** is ***0.0***.

suppose we have function factorialDouble

```
float pi(int n){
int  π, π1 = 0;
  for(int k=0; k<=n; k++)
  π1 += (factorialDouble(2*k-1))/factorialDouble(2*k)*
                                    * (2*k+1)

  π = 2π1;
  return π;
}
```

int division

: Kahan?

3

Use the backside, if needed

Problem 1 of 4

OOP.MT.170317.H056

Section, Name and ID#: [redacted]

Problem 2: Write a Java method ***public static double[] lin(double[] data)*** that takes as its argument an array of data points ***double[] data***, and returns a two-element array – the first element being the slope of the linear regression and the second element being the intercept. The linear regression approximates the data points by the linear formula

$$y = k\,x + b,$$

where the slope ***k*** and the intercept ***b*** are computed as

$$k = \frac{\overline{xy} - \bar{x}\,\bar{y}}{\overline{x^2} - \bar{x}^2}, b = \bar{y} - k\,\bar{x}$$

Here $\bar{x}$ is the mean of the *x* coordinates, $\bar{y}$ is the mean of the *y* coordinates, $\overline{x^2}$ is the mean of the squares of the *x* coordinates, and $\overline{xy}$ is the mean of the products of the *x* and *y* coordinates. Use the element indices of the array ***double[] data*** as *x* coordinates and the element values as *y* coordinates. You may assume and use the method double ***mean(double[] a)***.

```
public    static double[]  lin(double[] data) {
   int K=0,b=0;  int x, y, xMean, yMean, xSMean, ySMean, xy, Mxy =0;
   for (int i =0; i< data.length; i++)
        x += i;
   xMean =   x / data.length;
   for (int i =0; i< data.length; i++)
        y += data[i];
   y Mean =   y / data.length;
   for (int i =0; i< data.length; i++)
        X1 += i*i;
   x SMean  =  X1 / data.length;
   for (int i =0; i< data.length; i++)
        y1  += data[i] * data[i];
   yS Mean =   y1 / data.length;
   for (int i =0; i< data.length; i++)
        xy  += i * data[i];
   M x y  =  xy / data . length ;
   k  = (M xy - x Mean * yMean) / (xSMean - ySMean);
   b  = yMean  -  K * x Mean;
double []      result  = new   double [2];
             double [0] = K;
```

Use the backside, if needed

```
             double [1] =b;
       return   double;
       }
```

Problem 2 of 4

4

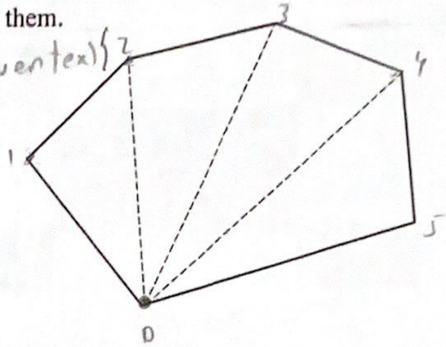
OOP.MT.170317.H056

Section, Name and ID#: ______________________________

Problem 3: Write a Java function ***public static double area(double[][] vertex)*** that takes as its argument a ***2-by-n*** array of a convex polygon's vertex coordinates ***double[][] vertex*** – the ***x*** coordinates in the first row and ***y*** coordinates in the second row. It returns polygon's area as follows:

1. Divides the polygon into triangles by connecting the ***first*** vertex with the ***n***th and ***(n+1)***st vertices;
2. Adds the areas of the constructed triangles using the formula $area = \sqrt{p(p-a)(p-b)(p-c)}$, where ***a***, ***b*** and ***c*** are the sides and ***p = (a + b + c) / 2***.

You may assume and use a method ***double dist(double x1, double y1, double x2, double y2)*** that takes as its arguments coordinates of two points and returns the distance between them.



```
public static double area(double[][] vertex){
    int i = 0; result = 0 ; S = 0 ;
for (int j = 0 ; j < vertex[i].length-2; j++) {
    a = dist (vertex [0][0], vertex[1][0]
              vertex[i][j+1], vertex(i+1](j+1]);

    b = dist( vetex[0][0], vertex[1][0];
              vertex[i][j+2], veter[i+1][j+2])

    c = dist (vertex[i][j+1], vetex[i+1][j+1],
              vertex [i][j+2] , vertex[i+1][j+2]);

      p = (a+b+c)/2;
   S = sqrt(p*(p-a)*(p-b)*(p-c));
       result += S ;

     }

   return result;

}
```

The 0 vertex is constant
& change the other 2 vertices
using for loop.

x_0	x_1	x_2	x_2	
y_0	y_1	y_2	y_3	

0 0
1 1
0 2
1 2

$i = 0$ $j = 0$

0 0 1 0 0 1 1 1

x_0 y_0 x_1 y_1

0 0 1 0 0 2 1 2

x_0 y_0 x_2 y_2

0 1 1 1 0 2 1 2

x_1 y_1 X_2 y_2

$i = 0$ $j = 1$

0 0 1 0 0 2 1 2

0 0 1 0 0 3 1 3

$j = 4$

4

Use the backside, if needed

Section, Name and ID#:_

Problem 4: Write a Java method ***public static void magic4N(int[][] square)*** that creates a magic square of a *4N-by-4N* size using the following algorithm:

1. Creates an array of the same size as ***int[][] square*** and fills it forward with successive integers assigning *1* to the top-left element;
2. Creates anther array of the same size as ***int[][] square*** and fills it backward with successive integers assigning *1* to the bottom-right element;
3. Divides the original ***int[][] square*** into 16 blocks of the same size – 4 blocks per row and column. In the on-diagonal (shaded) blocks copies the elements from the first array, and in the off-diagonal blocks copies the elements from second array.

0 0	1	2	0,1				7	8
1 0	9	10	1 1	1, 3			15	16
		2, 2	19	20	21	22		
4 0		3, 2	27	28	29	30		
5 0			35	36	37	38		
		7 1	43	44	45	46		
7. 0	49	50					55	56
8 0	57	58					63	64

8 1

	0 0	0 1					0 n-2	0 n-1
			62	61	60	59		
	1 0	1 1	54.	53	52	51	1 n-2	1 n-1
	48	47					42	41
	40	39					34	33
	32	31					26	25
	24	23					18	17
n-2 0			14	13	12	11		n-2 n-2
n-1 0		n-1 1	6	5	4	3		

```
public static void magin4N(int[][] square){
int[][] array1 = new int[square.lengh][square.length];
 fill Forward(array1); //suppose we have such function
int[][]. array2 = new int[square.length][square.length];
 fill Backward(array 2); // suppose we have such method

 for(int row=0; row<square.lengh, row++)
   for(int col=0; col<square.lengh, col++)
     if(
```

0 0
0 1
0 2

take the first block copy the corresspoding part of arrat. change row+2 col+2 and copy until you complet the first diagonal.

1	2
9	10

7	8
15	16

then take the last block copy the corresponding part of array1 and move by row+2, col-2 by filling the blocks.

The remaining part will be filled by the array 2.

2