

Section, Name and ID#:

AMERICAN UNIVERSITY OF ARMENIA  
College of Science and Engineering  
CS 120 Introduction to Object-Oriented Programming  
MIDTERM EXAM

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  $\pi$  by the following formula:

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

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.

```
float pi(int n) {
    float factorial = 1.0; float result; float pi;
    if (n % 2)
        for (int k = 2; k < n; k += 2)
            factorial = factorial * (2 * k - 1) / (2 * k);
        result = factorial / (2 * k + 1); // calculated the terms that we need to sum up
        pi = 2 * kahan(result, result, compensation);
    else if (n < 0)
        result = 1 / (2 * k + 1); pi = 2 * kahan(result, result, compensation);
    else
        for (int k = 1; k < n; k += 2)
            factorial = factorial * (2 * k - 1) / (2 * k);
        result = factorial / (2 * k + 1); pi = 2 * kahan(result, result, compensation);
    return pi;
}
```

Use the backside, if needed

Problem 1 of 4

00P.MT.170317.11003



Section, Name and ID#:

**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

$$y = kx + b,$$

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) {  
    double[] result = double new double[2];  
    for (int i = 0; i < data.length; i++)  
    mean(data);  
    double[] x = new double[data.length];  
    for (int i = 0; i < data.length; i++)  
        x[i] = i;  
    double[] xy = new double[data.length];  
    double[] xy = new double[data.length];  
    for (int i = 0; i < data.length; i++)  
        xy[i] = x[i] * data[i];  
    double[] squared = new double[data.length];  
    for (int i = 0; i < data.length; i++)  
        squared[i] = i * i;  
    result[0] = (mean(xy) - mean(x) * mean(data)) /  
    double result[0] = (mean(xy) - mean(x) *  
        * mean(data)) /  
        (mean(squared) - mean(x) * mean(x));  
    result[1] = mean(data) - result[0] * mean(x);  
    return result;  
}
```

Use the backside, if needed

Problem 2 of 4

OOP.MT.170317.11003

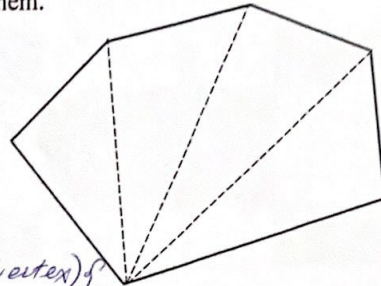


Section, Name and ID#:

**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  $n^{\text{th}}$  and  $(n+1)^{\text{st}}$  vertices;
2. Adds the areas of the constructed triangles using the formula  $\text{area} = \sqrt{p(p-a)(p-b)(p-c)}$ , 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) {  
    for (int i = 0, i < vertex.length; i++)  
    double a = dist(vertex[0][i], vertex[1][i],  
                vertex[0][i+1], vertex[1][i+1]);  
    double b = dist(vertex[0][i], vertex[1][i], vertex[0][i+2], vertex[1][i+2]);  
    double c = dist(vertex[0][i], vertex[1][i], vertex[0][i+2], vertex[1][i+2]);  
}
```

Use the backside, if needed

Problem 3 of 4

00P.M.T. 170315.H003



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 another 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.

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

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

- ① `public static void magic4N(int[][] square) {`  
`int[][] forward = new int[square.length][square.length];`  
`for (int row = 0; row < forward.length; row++)`  
`for (int col = 0; col < forward.length; col++)`  
`forward[row][col] = row * forward.length + col + 1;`  
`int[][] backward = new int[square.length][square.length];`  
`for (int row = 0; row < backward.length; row++)`  
`for (int col = 0; col < backward.length; col++)`  
`backward[row][col] = backward.length * (backward.length`  
`- row) - col; (-1?);`  
 ③

2

Use the backside, if needed

Problem 4 of 4

GOP. MT. 170317.11003