Program 1

Code: approxPi.c

```
#include <stdio.h>
#include <stdlib.h>
#include <math.h>

typedef int bool;
#define true 1
#define false 0

int main(void)
{
	bool validInput = false;
	int input;
	int inCircle;
	double x;
	double y;

	while (!validInput)
	{
		printf("Enter the number of interations: ");

		scanf("%d", &input);

		if (input > 0)
			validInput = true;
	}
```

```
	int i;


	srand(time(NULL));


	for (i = 0; i < input; i++)
	{
		x = (double)rand()/RAND_MAX;
		y = (double)rand()/RAND_MAX;


		if (x * x + y * y < 1)
			inCircle++;
	}


	printf("After %d iterations, pi has been approximated to: %lf\n",
input, (double)inCircle * 4 / input);
}
```

Code: approxMorePi.c

```
#include <stdio.h>
#include <stdlib.h>
#include <math.h>

typedef int bool;
#define true 1
#define false 0
#define MAX 10

int main(void)
{
      bool validInput = false;          // Flag for input correctness
      int input;                        // Number of iterations for the
                                        approximation of pi
      int inCircle;                     // Number of values of x and y
                                        that lie within the circle
      double x;                         // The x co-ordinate
      double y;                         // The y co-ordinate
      double avgSum, stdDevSum;         // Variables holding the total
                                        summation values of the
                                        approximations
      double results[MAX];              // An array containing the ten
                                        approximations
```

```
	// Reads input and validates it
	while (!validInput)
	{
		printf("Enter the number of interations: ");
		scanf("%d", &input);

		// Checks to make sure that the number of iterations is
		positive
		if (input > 0)
			validInput = true;
	}

	srand(time(NULL));			// Seeding the random function

	int j;					// Counter for the outer loop
	int i;					// Counter for the for the inner
loop

	for (j = 0; j < MAX; j++)
	{
		inCircle = 0;
		for (i = 0; i < input; i++)
		{
			// Generate random x and y co-ordinates
			x = (double)rand()/RAND_MAX;
			y = (double)rand()/RAND_MAX;
```

```
            // If the point (x^2,y^2) lies in the circle,
            increment the counter
            if (x * x + y * y < 1)
                  inCircle++;
      }
      results[j] = (double)inCircle * 4 / input;
      printf("%d: %lf\n", j, results[j]);
      avgSum += results[j];
   }

   avgSum = avgSum/MAX;

   for (i = 0; i < 10; i++)
      stdDevSum += (results[i] - avgSum) * (results[i] - avgSum);

   stdDevSum = sqrt(stdDevSum / MAX);

   printf("After %d iterations, the average approximation of pi
   is: %lf, with a standard deviation of %lf. \n", input * 10,
   avgSum, stdDevSum);
}
```

Cases:

N = 10

obelix[51]% appPi
Enter the number of interations: 10
0: 2.000000
1: 3.600000
2: 3.200000
3: 2.800000
4: 2.400000
5: 2.800000
6: 3.200000
7: 2.400000
8: 3.200000
9: 2.400000
After 100 iterations, the average approximation of pi is: 2.800000, with a standard deviation of 0.473286.

N = 100

obelix[49]% appPi
Enter the number of interations: 100
0: 3.320000
1: 2.920000
2: 2.760000
3: 3.120000
4: 3.280000
5: 3.320000
6: 3.080000
7: 3.160000
8: 3.080000
9: 3.120000
After 1000 iterations, the average approximation of pi is: 3.116000, with a standard deviation of 0.167284.

N = 1,000

obelix[50]% appPi
Enter the number of interations: 1000
0: 3.160000
1: 3.160000
2: 3.068000
3: 3.212000
4: 3.220000
5: 3.076000
6: 3.084000
7: 3.156000
8: 3.120000
9: 3.048000
After 10000 iterations, the average approximation of pi is: 3.130400, with a standard deviation of 0.057444.

N = 10,000

obelix[52]% appPi
Enter the number of interations: 10000
0: 3.127200
1: 3.126400
2: 3.118400
3: 3.119600
4: 3.169200
5: 3.138800
6: 3.121200
7: 3.134800
8: 3.108400
9: 3.150800
After 100000 iterations, the average approximation of pi is: 3.131480, with a standard deviation of 0.016898.

N = 100,000

obelix[47]% appPi
Enter the number of interations: 100000
0: 3.147760
1: 3.139920
2: 3.146720
3: 3.135800
4: 3.129400
5: 3.146040
6: 3.141040
7: 3.142480
8: 3.142000
9: 3.140240
After 1000000 iterations, the average approximation of pi is:
3.141140, with a standard deviation of 0.005198.

N = 1,000,000

obelix[46]% appPi
Enter the number of interations: 1000000
0: 3.144972
1: 3.139204
2: 3.139452
3: 3.141520
4: 3.140340
5: 3.144264
6: 3.141348
7: 3.140256
8: 3.139896
9: 3.141216
After 10000000 iterations, the average approximation of pi is:
3.141247, with a standard deviation of 0.001846.

N = 10,000,000

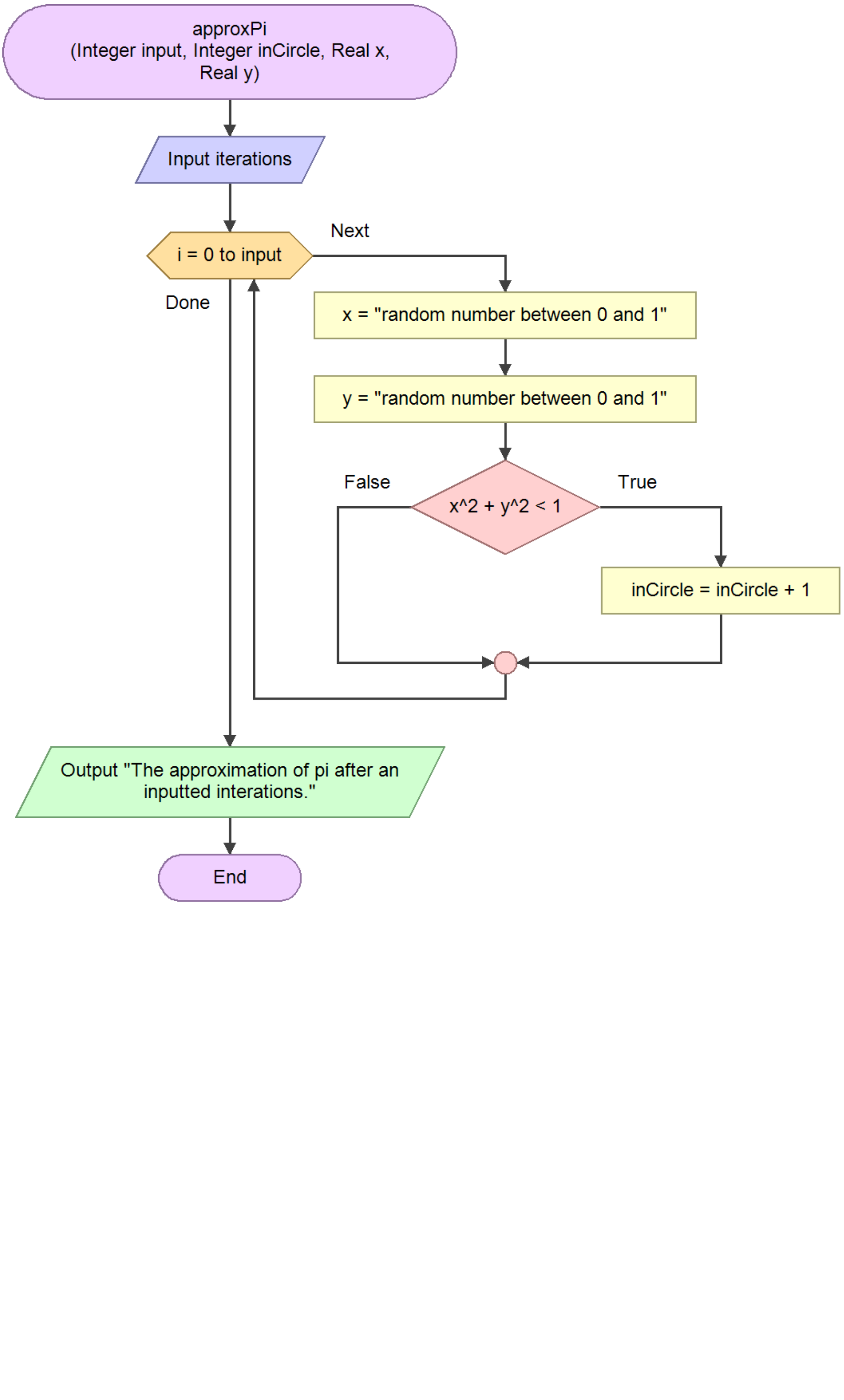
obelix[45]% appPi

Enter the number of interations: 10000000
0: 3.141478
1: 3.141121
2: 3.141416
3: 3.141950
4: 3.142189
5: 3.141014
6: 3.141794
7: 3.141735
8: 3.141001
9: 3.141734
After 100000000 iterations, the average approximation of pi is:
3.141543, with a standard deviation of 0.000386.

N = 100,000,000

obelix[44]% appPi
Enter the number of interations: 100000000
0: 3.141892
1: 3.141657
2: 3.141550
3: 3.141164
4: 3.141399
5: 3.141513
6: 3.141613
7: 3.141468
8: 3.141739
9: 3.141330
After 1000000000 iterations, the average approximation of pi is:
3.141532, with a standard deviation of 0.000198.

For the sake of this flowchart, consider “Real” as double.



Program 2

Code:

```
#include<stdio.h>

typedef int bool;
#define true 1
#define false 0

int main(void)
{
	bool validInput = false;
	int size;


	// Input validation
	while (!validInput)
	{
		printf("Enter the size of the magic square: ");

		scanf("%d", &size);

		if (size > 0 && size < 99 && size %2 != 0)
			validInput = true;
		else
			printf("Invalid size.\n");
	}
		// Declare and initialize the two-dimensional array,
filling all slots with 0s
```

```
            int square[size][size];

            // Declaring column, row and the value to add
            int col;
            int row;
            int val;

            for (col = 0; col < size; col++)
            {
                  for (row = 0; row < size; row++)
                        square[col][row] = 0;
            }

            // Reset row and column
            col = 0;
            row = size / 2;
            val = 2;

            // Set the middle value of the first row to 1 to mark the
starting position
            square[col][row] = 1;

            while(val < size * size + 1)
            {
                  // To move up a row and one column to the right
                  col--;
                  row++;
```

```
                // Make sure the pointers do not exceed the limits of
the square
                if (col == -1)
                      col = size - 1;
                if (row == size)
                      row = 0;

                // If there is no valid value in the slot:
                if (square[col][row] == 0)
                {
                      // Inserts the value into the slot
                      square[col][row] = val;

                      // Increments the value
                      val++;
                }

                else
                {
                      col += 2;
                      row--;

                      // Make sure the pointers do not exceed the
limits of the square
                      if (col >= size)
                            col -= size;

                      if (row == -1)
```

```
                    row = size -1;

                    // Inserts the value into the slot
                    square[col][row] = val;

                    // Increments the value
                    val++;
               }
          }

          // Print out the array
          for (col = 0; col < size; col++)
          {
               for (row = 0; row < size; row++)
                    printf("\t %d", square[col][row]);
               printf("\n");
          }
}
```

Cases:

Size 3

obelix[31]% magicSquare

Enter the size of the magic square: 3

```
      8     1     6

      3     5     7

      4     9     2
```

Size 5

obelix[33]% magicSquare

Enter the size of the magic square: 5

```
      17    24    1     8     15

      23    5     7     14    16

      4     6     13    20    22

      10    12    19    21    3

      11    18    25    2     9
```

Size 7:

obelix[34]% magicSquare

Enter the size of the magic square: 7

```
      30    39    48    1     10    19    28

      38    47    7     9     18    27    29

      46    6     8     17    26    35    37

      5     14    16    25    34    36    45

      13    15    24    33    42    44    4

      21    23    32    41    43    3     12

      22    31    40    49    2     11    20
```

Size 9:

obelix[35]% magicSquare

Enter the size of the magic square: 9

```
      47    58    69    80    1     12    23    34    45
      57    68    79    9     11    22    33    44    46
      67    78    8     10    21    32    43    54    56
      77    7     18    20    31    42    53    55    66
      6     17    19    30    41    52    63    65    76
      16    27    29    40    51    62    64    75    5
      26    28    39    50    61    72    74    4     15
      36    38    49    60    71    73    3     14    25
      37    48    59    70    81    2     13    24    35
```

Size -1 (values less than 0):

obelix[36]% magicSquare

Enter the size of the magic square: -1

Invalid size.

…

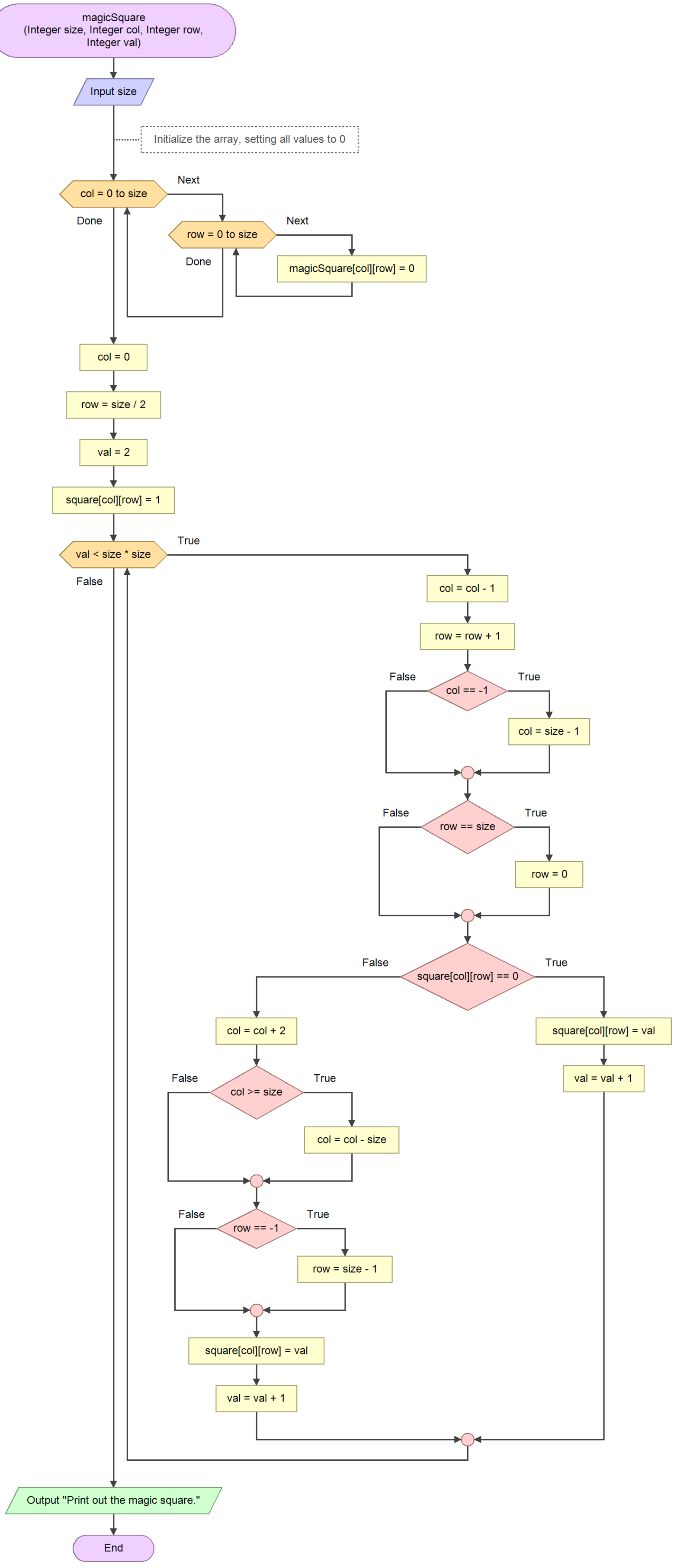
Size 6 (even numbers):

obelix[36]% magicSquare

Enter the size of the magic square: 6

Invalid size.

…


magicSquare
(Integer size, Integer col, Integer row,
Integer val)
Input size
Initialize the array, setting all values to 0
col = 0 to size
Next
Done
row = 0 to size
Next
Done
magicSquare[col][row] = 0
col = 0
row = size / 2
val = 2
square[col][row] = 1
val < size * size
True
False
col = col - 1
row = row + 1
col == -1
False
True
col = size - 1
row == size
False
True
row = 0
square[col][row] == 0
False
True
col = col + 2
square[col][row] = val
val = val + 1
col >= size
False
True
col = col - size
row == -1
False
True
row = size - 1
square[col][row] = val
val = val + 1
Output "Print out the magic square."
End

Problem 3

Code:

```
#include <stdio.h>

void payMethod(int dollars, int *twenties, int *tens, int *fives, int
*toonies, int *loonie);

int main(void)
{
	int dollars;
	int twenties;
	int tens;
	int fives;
	int toonies;
	int loonies;

	printf("Enter the payment value: $");
	scanf("%d", &dollars);

	payMethod(dollars, &twenties, &tens, &fives, &toonies, &loonies);
	printf("The fewest bills and coins necessary to pay $%d.00 is: %d
twenties, %d tens, %d fives, %d toonies, %d loonies.\n", dollars,
twenties, tens, fives, toonies, loonies);

	return 0;
}

void payMethod(int dollars, int *twenties, int *tens, int *fives, int
*toonies, int *loonies)
{
	// How many twenties
	*twenties = dollars / 20;

	// Updating the dollar value
	dollars %= 20;

	// How many tens
	*tens = dollars / 10;

	// Updating the dollar value
```

```
    dollars %= 10;

    // How many fives
    *fives = dollars / 5;

    // Updating the dollar value
    dollars %= 5;

    // How many toonies
    *toonies = dollars / 2;

    // Updating the dollar value
    dollars %= 2;

    // How many loonies
    *loonies = dollars;
}
```

Cases:

Case 1

obelix[21]% paymentAmount
Enter the payment value: $1
The fewest bills and coins necessary to pay $1.00 is: 0 twenties, 0 tens, 0 fives, 0 toonies, 1 loonies.

Case 2

obelix[22]% paymentAmount
Enter the payment value: $2
The fewest bills and coins necessary to pay $2.00 is: 0 twenties, 0 tens, 0 fives, 1 toonies, 0 loonies.

Case 3

obelix[23]% paymentAmount
Enter the payment value: $3
The fewest bills and coins necessary to pay $3.00 is: 0 twenties, 0 tens, 0 fives, 1 toonies, 1 loonies.

Case 4

obelix[24]% paymentAmount
Enter the payment value: $4
The fewest bills and coins necessary to pay $4.00 is: 0 twenties, 0 tens, 0 fives, 2 toonies, 0 loonies.

Case 5

obelix[25]% paymentAmount
Enter the payment value: $5
The fewest bills and coins necessary to pay $5.00 is: 0 twenties, 0 tens, 1 fives, 0 toonies, 0 loonies.

Case 6

obelix[26]% paymentAmount
Enter the payment value: $11
The fewest bills and coins necessary to pay $11.00 is: 0 twenties, 1 tens, 0 fives, 0 toonies, 1 loonies.

Case 7

obelix[27]% paymentAmount
Enter the payment value: $29
The fewest bills and coins necessary to pay $29.00 is: 1 twenties, 0 tens, 1 fives, 2 toonies, 0 loonies.

Case 8

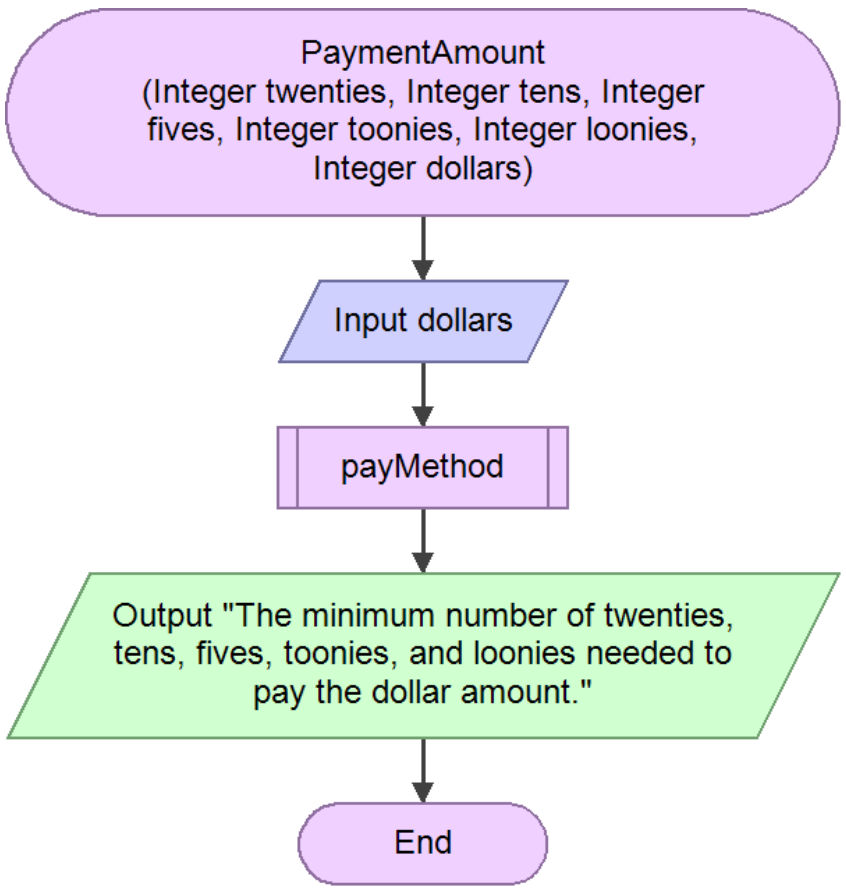
obelix[28]% paymentAmount
Enter the payment value: $39
The fewest bills and coins necessary to pay $39.00 is: 1 twenties, 1 tens, 1 fives, 2 toonies, 0 loonies.

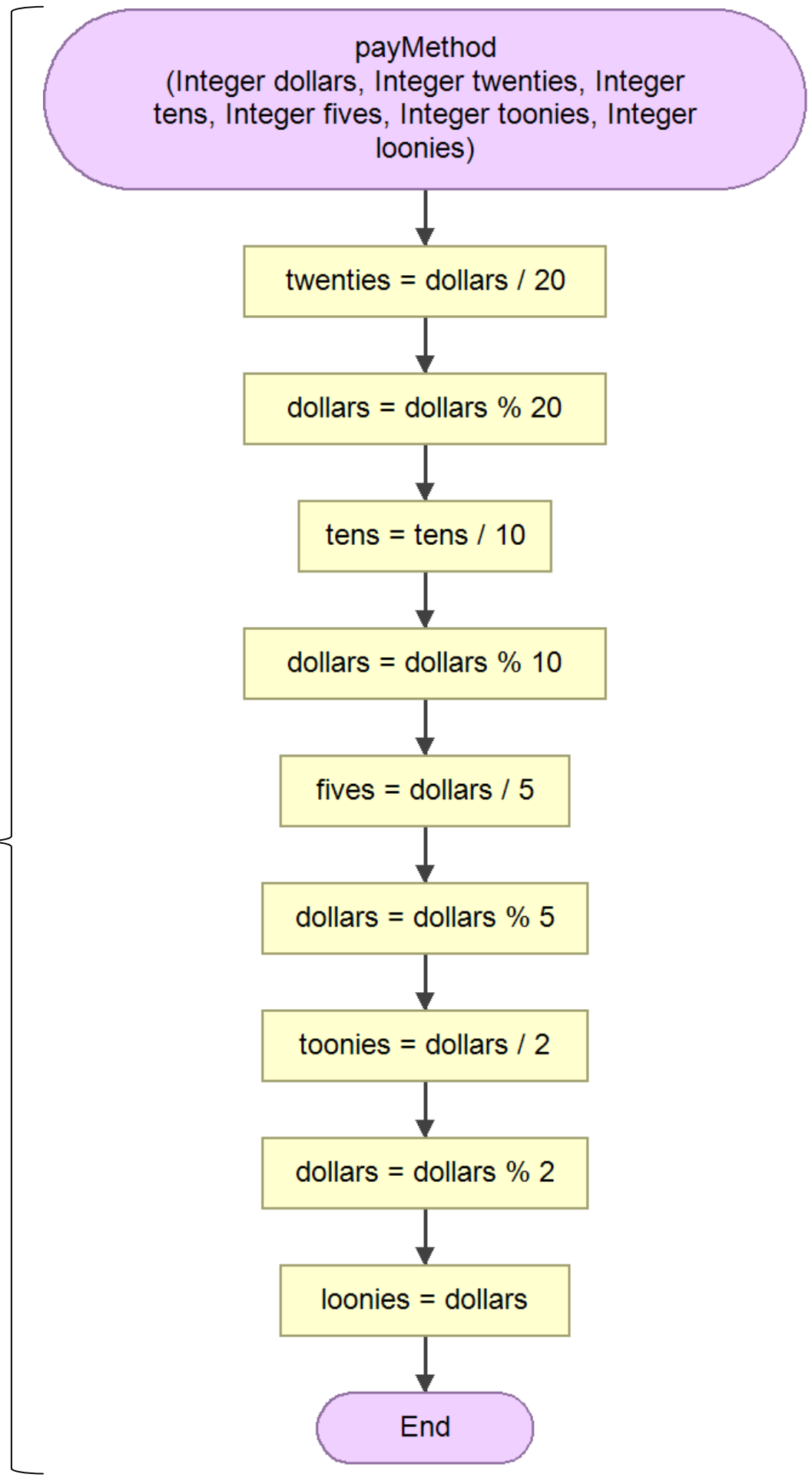
Case 9

obelix[29]% paymentAmount
Enter the payment value: $37
The fewest bills and coins necessary to pay $37.00 is: 1 twenties, 1 tens, 1 fives, 1 toonies, 0 loonies.

Case 10

obelix[30]% paymentAmount
Enter the payment value: $38
The fewest bills and coins necessary to pay $38.00 is: 1 twenties, 1 tens, 1 fives, 1 toonies, 1 loonies.


PaymentAmount
(Integer twenties, Integer tens, Integer fives, Integer toonies, Integer loonies, Integer dollars)
Input dollars
payMethod
Output "The minimum number of twenties, tens, fives, toonies, and loonies needed to pay the dollar amount."
End


payMethod
(Integer dollars, Integer twenties, Integer tens, Integer fives, Integer toonies, Integer loonies)
twenties = dollars / 20
dollars = dollars % 20
tens = tens / 10
dollars = dollars % 10
fives = dollars / 5
dollars = dollars % 5
toonies = dollars / 2
dollars = dollars % 2
loonies = dollars
End