Program Design
Problem solving and Implementation

Course: CPSC 1150
Instructor: Dr. Bita Shadgar

Lecture 6

Learning Outcomes

- Identify the program design process
- Recognize the input, process, and output of any program or function
- Calculate the output(s) using the input(s)
- Design the algorithm using a flowchart
- Apply pseudo-code to represent an algorithm
- Test your algorithm with different possible inputs
- Convert the algorithm to the programming code

Programming Design

A. Problem Solving
   1. Define the problem
   2. Design an algorithm
   3. Test the algorithm

B. Implementation
   1. Write the code (program)
   2. Correct syntax errors
   3. Test the program
   4. Correct logic errors

A.1 Problem Definition

- Understand the task that the program is to perform
- To make sure you have understood the problem itself, answer the following questions:
    - What are the inputs, and what are their types?
    - What are the outputs, and what are their types?
    - What is the source(s) of input(s)?
    - How the output(s) should be delivered?

Example#1 - Problem Definition

- Write a program to calculate the perimeter and the area of a circle.
  - Inputs?
    - The **radius** of the circle: **One real** number
    - It can be read from the **keyboard**.
  - Outputs?
    - The calculated **perimeter** and **area**: **Two real** numbers
    - They can be reported as a message on the **screen**.

Example #2

- Develop an algorithm to find out the average of a class for an exam?
  - Input(s):
    - grades of students : zero or more float number(s)
    - can be read from keyboard
  - Output(s):
    - average grade of class: a float number
    - can be displayed on the screen

A.2 Algorithm Design

Definition

Algorithm is an ordered set of precise instructions that leads to a solution to a problem.

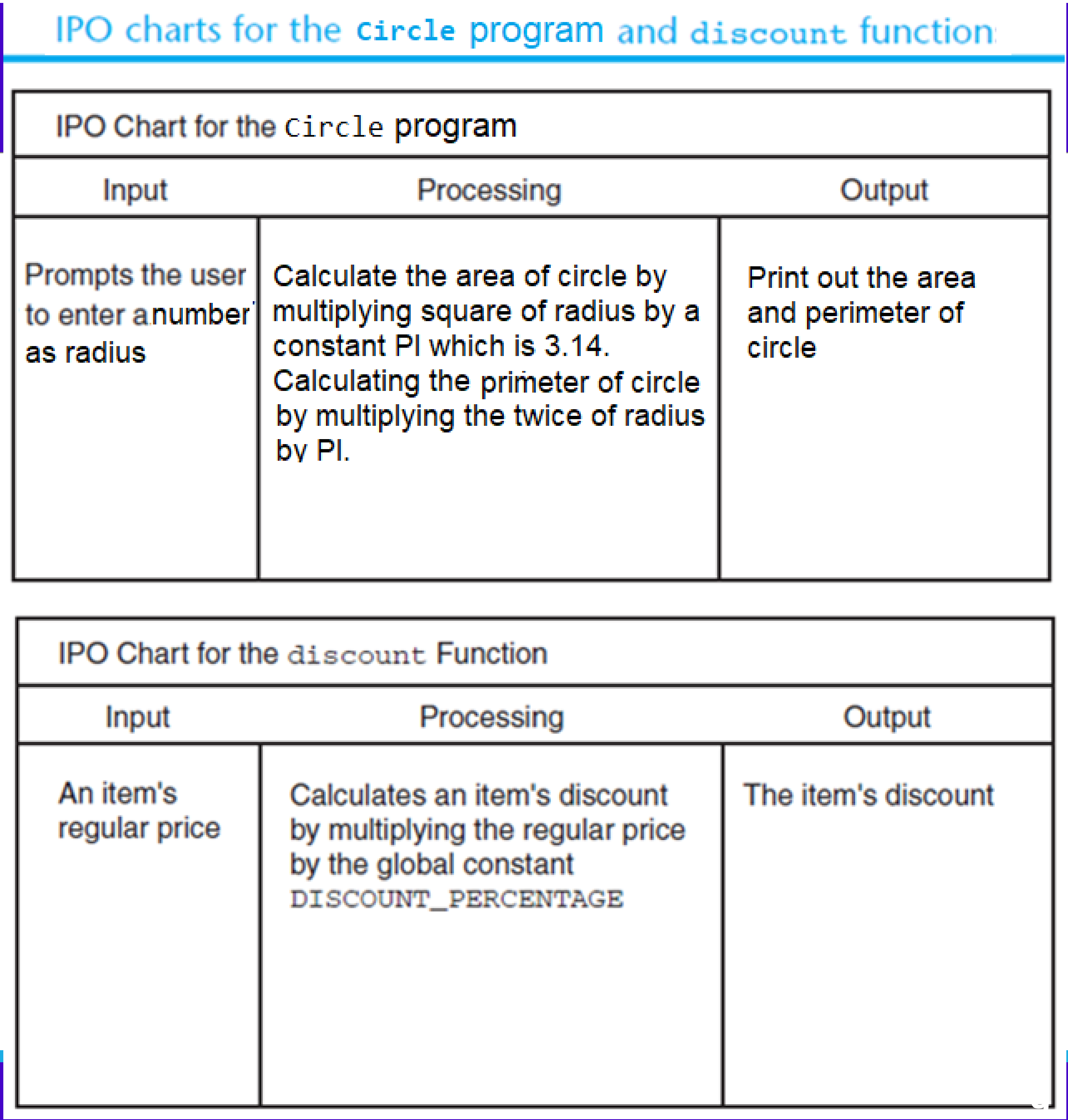
- Programs must be designed before they are written.
  - Steps to produce output(s) from the input(s).
- Features of an algorithm
  - Correct: Provide correct output
  - Unambiguous: meaning is clear
  - Deterministic: Unique action specified
  - Feasible: Can be performed
  - Finite: Come to an end.

A.2 Algorithm Design (cont'd.)

- There are three general steps usually available in all algorithms:
  1. Read inputs from their appropriate sources.
  2. Process your inputs to calculate the results (outputs).
  3. Deliver the outputs.
- At any of the above steps, you need to identify:
  1. If some task needs to be done conditionally or at all circumstances?
  2. If some tasks is repeated? If so, identify the number of times it is repeated, or the condition(s) need(s) to be held to repeat.

IPO Charts

– IPO charts is the essence of system analysis that describes the input, processing, and output of a program or a task



IPO charts for the circle program and discount function

IPO Chart for the Circle program		
Input	Processing	Output
Prompts the user to enter a number as radius	Calculate the area of circle by multiplying square of radius by a constant PI which is 3.14. Calculating the primeter of circle by multiplying the twice of radius by PI.	Print out the area and perimeter of circle

IPO Chart for the discount Function		
Input	Processing	Output
An item's regular price	Calculates an item's discount by multiplying the regular price by the global constant DISCOUNT_PERCENTAGE	The item's discount

A.2 Algorithm Design: naming variables

– Consider a name for any of the following values (data) in your algorithm; names will be later used to reference their corresponding values:

1. All the **input data**
2. All the **calculation results** that are supposed to be used more than once.

   - Data is used in different ways like being printed on the screen, participating in a calculation, etc.

Example - An algorithm to calculate the perimeter and the area of a circle

1. Read the radius of the circle from keyboard (radius).
2. Calculate the result of the following formulas
   - $\text{area} = 3.14 \times \text{radius}^2$
   - $\text{perimeter} = 2 \times 3.14 \times \text{radius}$
3. Print area and perimeter on screen

NOTE:

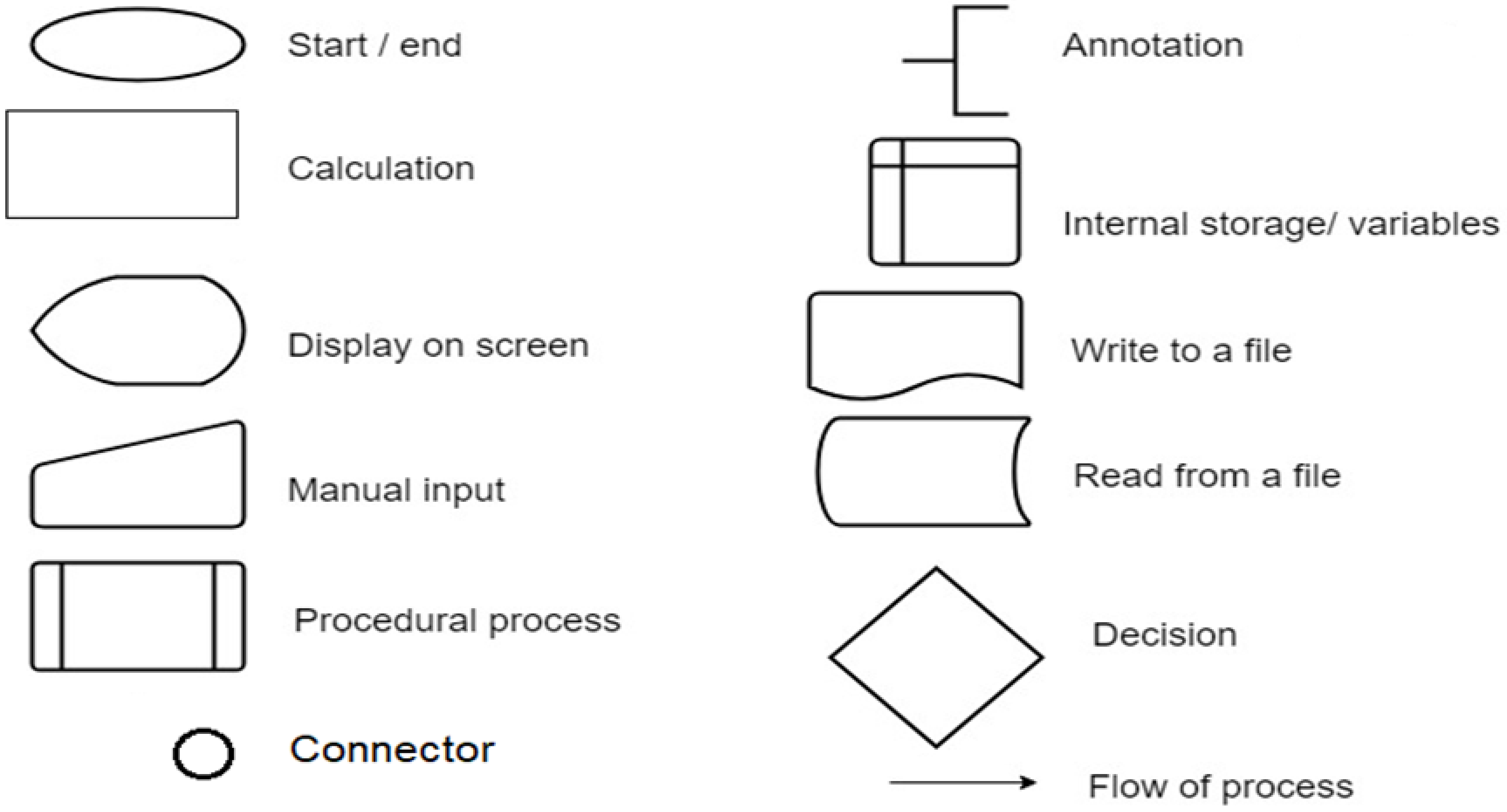
All the above operations need to be done only once at all circumstances; no conditional or repeated operation exists. done only once

Representing an Algorithm

- An algorithm needs to be represented in a very clear, unambiguous way so it can be easily converted to a program.
- There are different ways to state an algorithm:
  - Informal tool: **English**
    - Naturally ambiguous, not suitable for complex algorithms
  - Standard tools: **Flowcharts, Pseudocode**
    - No ambiguity, suitable for more complex algorithms

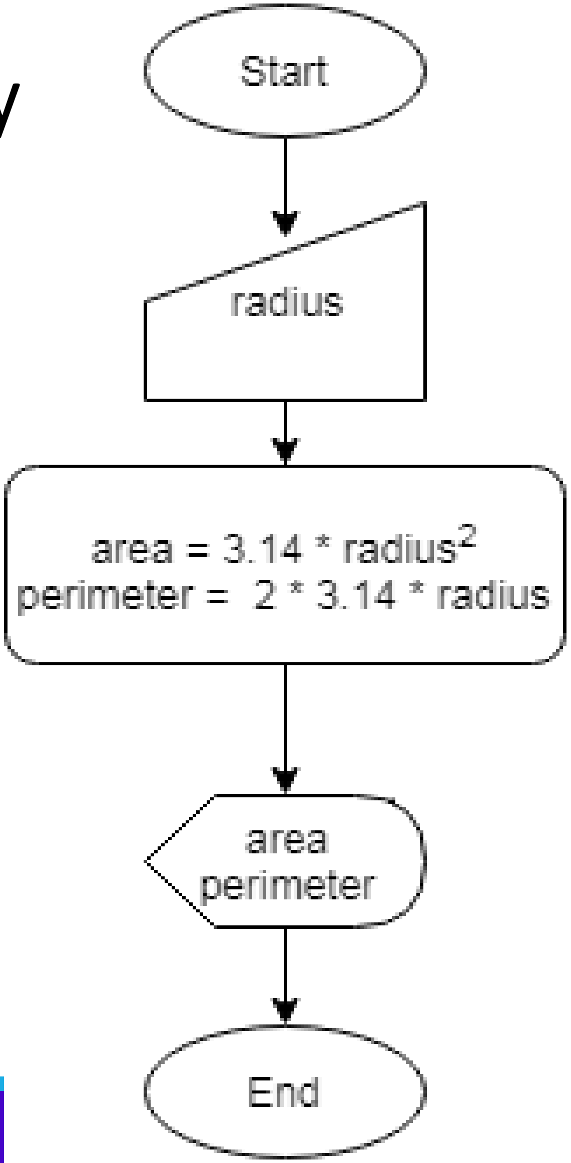
Representing Algorithm - Flowcharts

– diagram that graphically depicts the steps in a program



Representing Algorithm - Flowcharts

- Flowchart is too primitive to display details, but good for high level representation, or small programs.
- Use online software https://www.draw.io/



Representing Algorithm - Pseudocode

Definition

Pseudocode (fake code) is an informal language that has no syntax rule

- Not meant to be compiled or executed
- Used to create a model program
  - No worry about syntax errors, focuses on program's design
  - Can be translated directly into actual code in any programming language

Pseudo code primitives (Assignment)

name ← *expression*

– Gives a symbolic *name* to an *expression*

– A *name* can be reassigned to another value during the execution of the algorithm

Pseudo code primitives (Selection)

```
If condition is true then
      activity 1
otherwise
      activity 2
```

– *condition* is an expression that evaluates to true or false

– If the *condition* is true execution continues with *activity 1* otherwise *activity 2*

Pseudo code primitives (Loop)

```
Repeat while condition is true
    activity
```

– *Activity* is executed repeatedly until the *condition* expression is false

```
Repeat n times
    activity
```

Example 1

Algorithm: find area and perimeter of a circle

(Define terms)

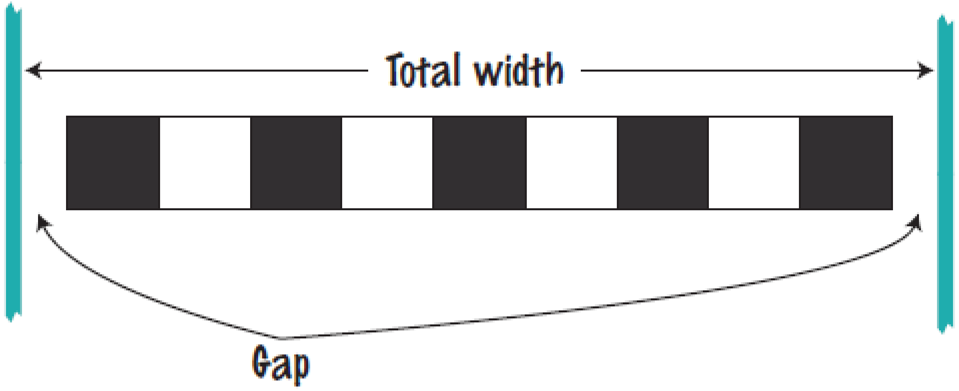
0.1 Let radius be the radius of a circle.

0.2 let area and perimeter be the area and perimeter of the circle.

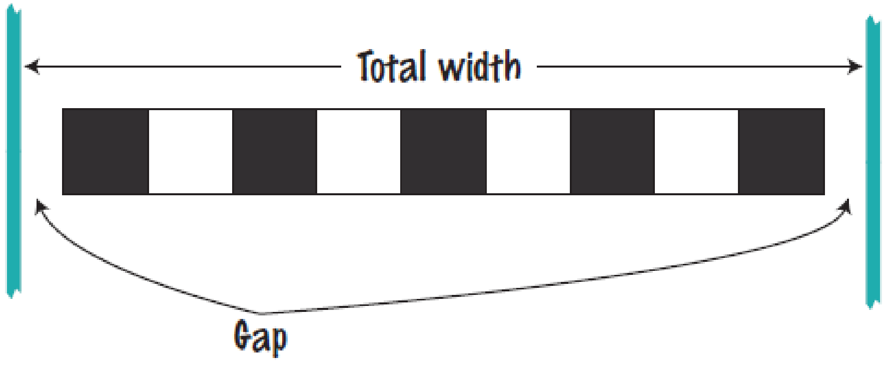
```
START
1.  Read radius from keyboard
2.  area ← radius * radius * PI
3.  perimeter ← 2 * perimeter * PI
4.  Print area and perimeter
END
```

A. Problem Solving: First by Hand

- A very important step for developing an algorithm is to first carry out the computations *by hand*.
- Example Problem:
  - A row of black and white tiles needs to be placed along a wall. For aesthetic reasons, the architect has specified that the first and last tile shall be black.
  - Your task is to compute the number of tiles needed and the gap at each end, given the space available and the width of each tile.



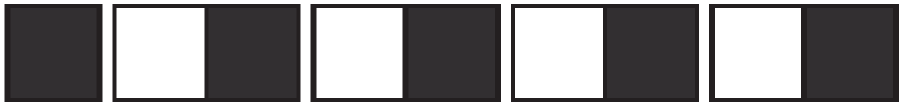
Start with example values



Given
- Total width: 100 inches
- Tile width: 5 inches

Test your values
- Let's see… 100/5 = 20, perfect! 20 tiles. No gap.
- But wait… BW…BW "…first and last tile shall be black."

Look more carefully at the problem….
- Start with one black, then some number of WB pairs

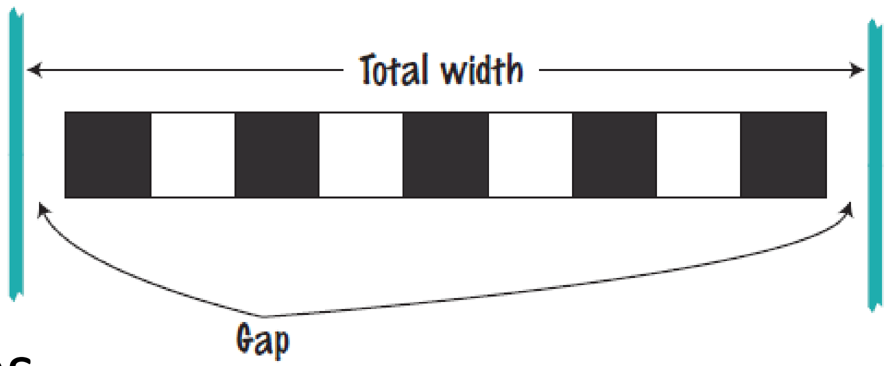


- Observation: each pair is 2x width of 1 tile
  - In our example, 2 x 5 = 10 inches

Keep applying your solution

Total width: 100 inches

Tile width: 5 inches



- Calculate total width of all tiles
  - One black tile: 5”
  - 9 pairs of BWs: 90”
  - Total tile width: 95”
- Calculate gaps (one on each end)
  - 100 – 95 = 5” total gap
  - 5” gap / 2 = 2.5” at each end

Now devise an algorithm

- Use your example to see how you calculated values
- How many pairs? (number of pairs)
  - Note: must be a whole number

Integer part of:

(total width – tile width) / (2 x tile width)

- How many tiles? (number of tiles)

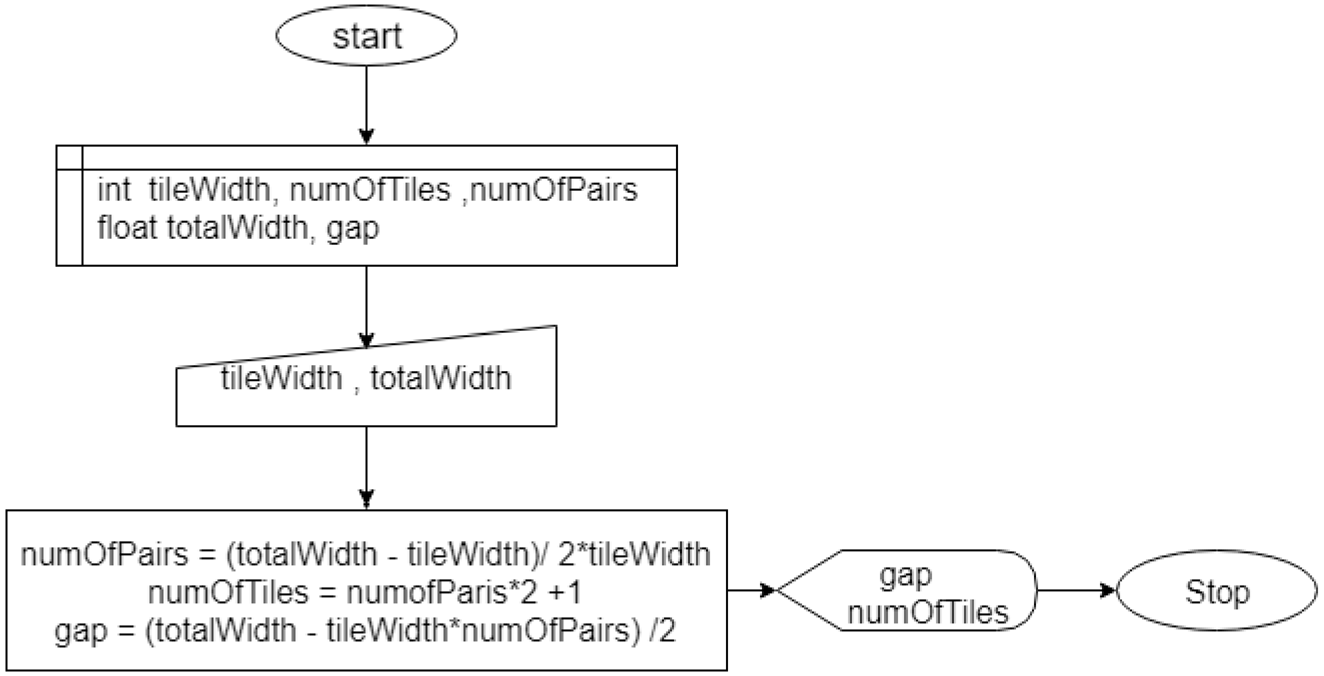
1 + 2 x the number of pairs

- Gap at each end (gap)

(total width – number of tiles x tile width) / 2

In class exercise

– Represent your algorithm of tiling by flowchart


start
int tileWidth, numOfTiles ,numOfPairs
float totalWidth, gap
tileWidth , totalWidth
numOfPairs = (totalWidth - tileWidth)/ 2*tileWidth
numOfTiles = numofParis*2 +1
gap = (totalWidth - tileWidth*numOfPairs) /2
gap
numOfTiles
Stop

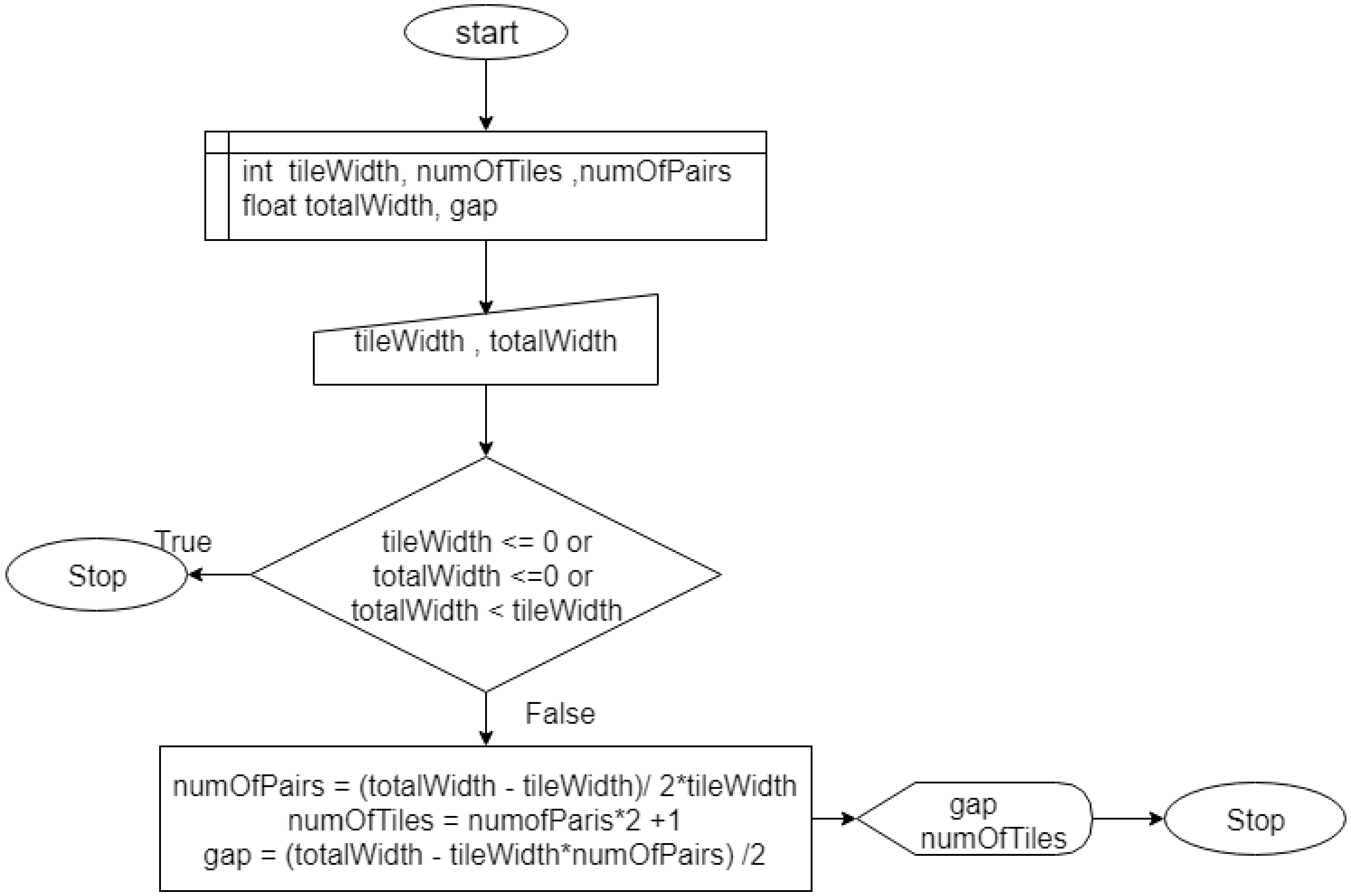
In class exercise

– Represent your algorithm of tiling by pseudo-code

Test the algorithm

- Test the algorithm with different inputs and make sure that algorithm works correctly
- Make sure different data cause no runtime error (logical or exception error)
- Example:
  - what if the total width is less than a tile width?
  - what if the total width is less than twice of a tile width?
  - what if the inputs are negative?

Modify the algorithm


start
int tileWidth, numOfTiles ,numOfPairs
float totalWidth, gap
tileWidth , totalWidth
tileWidth <= 0 or
totalWidth <=0 or
totalWidth < tileWidth
True
Stop
False
numOfPairs = (totalWidth - tileWidth)/ 2*tileWidth
numOfTiles = numofParis*2 +1
gap = (totalWidth - tileWidth*numOfPairs) /2
gap
numOfTiles
Stop

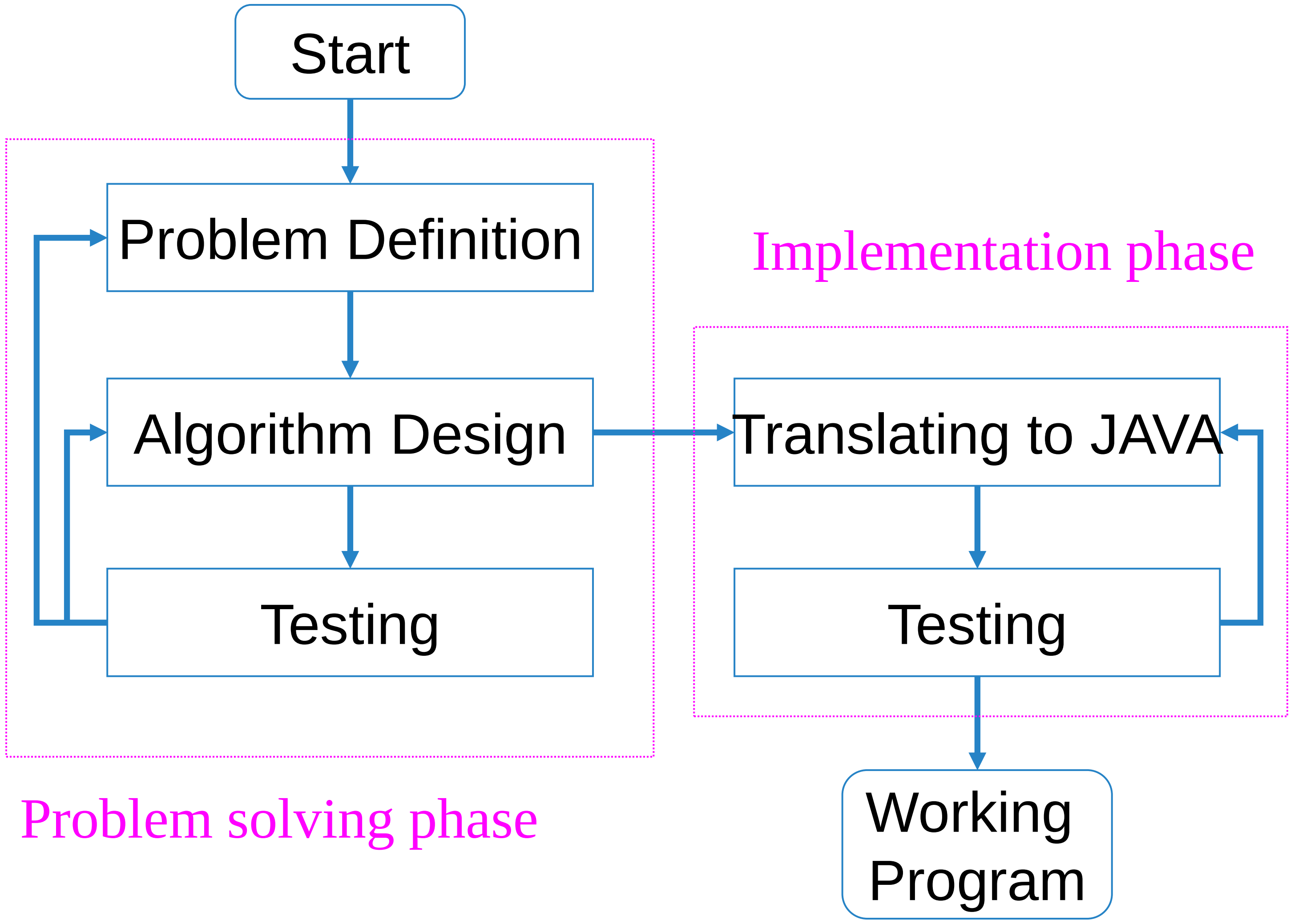
B. From Algorithm to Program

Definition

A **program** is an algorithm stated in a given programming language.

- Each step in the algorithm needs to be converted to an equivalent statement in the underlying programming language (e.g. Java).
- Each programming language provides a set of *operators*, *structures*, *functions*, etc. to do different tasks.
- The **syntax** and **semantic** of a given programming language must be followed.

Program Design Process



Example 2

Algorithm: Find average of a class for an exam.

(Define terms)

0.1 Let numOfStudents be the number of the students in the class.

0.2 Let grade be the grade of each students.

```
START
1.  numOfStudents ← 0
2.  Total ← 0
3.  Repeat for all students
        3.1 read grade of the student.
        3.2 total ← total + grade
        3.3 Increment numOfStudents by 1
4. average ← total / numOfStudents
5. Print average
END
```

Example 3

Algorithm: Find number of the students passed or failed in a class

```
(Define terms)
  0.1 Let passed be the number of the students passed.
  0.2 Let failed be the number of the students failed.
  0.3 Let grade be the grade of each students in the exam.
START
1.  passed ← 0
2.  failed ← 0
3.  Repeat for all students
        3.1  read grade of the student.
        3.2  if grade >= 50 then
                 3.2.1 passed ← passed+1
             Otherwise
                 3.2.2 failed ← failed+1
4. Print passed and failed
END
```

Example 4 - Factorial

The factorial of a number **`X`**`,` **`X!`** is the product of all integers from 1 up to and including X

- e.g. 10! = 10(9)(8)(7)…(3)(2)(1) = ?
  4! = 4(3)(2)(1) = 24

– Write an algorithm to compute **`X!`** (use flowchart and pseudo-code to show your algorithm)

Pseudo code for Factorial

Algorithm: Find n! (Factorial of n)

(Define terms)

0.1 Let *n* be the number input by user.

0.2 Let *factorial_n* be the factorial of *n*

```
START
1.   Input n
2.   factorial_n ← 1
3.   Repeat while n > 0
         3.1   factorial_n ← factorial_n * n
         3.2   n ← n -1
4.   Print factorial_n
END
```

Test using Trace Table

n	factorial_n	Output
5	1	

Example 5 - Fibonacci

The Fibonacci sequence is the sequence of integers such that every number in the sequence is the sum of the previous two numbers in the sequence.

- e.g. 1, 1, 2, 3, 5, 8, 13, 21, …

– Write an algorithm to display the first X values of the Fibonacci sequence. (use flowchart and pseudo-code to show your algorithm)

Pseudo code for Fibonacci

Algorithm: Print first x Fibonacci numbers

```
(Define terms)
  0.1 Let x be the number input defined by user.
START
1. Input x
2. num1 ← 0
3. num2 ← 1
4. newNum ← 1
5. counter ← 0
6. Print newNum,
7. Repeat while x>= counter
      7.1 newNum ← num1 + num2
      7.2 Print newNum,
      7.3 num1 ← num2
      7.4 num2 ← newNum
      7.5 counter ← counter +1
END
```

Test using Trace Table

x	num1	num2	newNum	counter	output
5					

Is this algorithm Correct?

More Practice

A prime number is a number that is dividable by only 1 and by itself.

- Example of prime numbers: 1, 2, 3, 5, 7, 11, 13, 17, 19, . . .

– Develop an algorithm to find out if a given integer number is a prime number. Represent your algorithm using a flowchart and a pseudocode.