

2.5

# Algorithm and Flowcharting

# Algorithm

- An algorithm is a plan, a set of step-by-step instructions to resolve a problem.
- In an algorithm, each instruction is identified and the order in which they should be carried out is planned.
- Algorithms are one of the four cornerstones of Computer Science.
- ✓ *If you can tie shoelaces, make a cup of tea, get dressed or prepare a meal then you already know how to follow an algorithm. 😊*
- In an algorithm, each instruction is identified and the order in which they should be carried out is planned.
- Algorithms are often used as a starting point for creating a computer program, and they are sometimes written as a flowchart or in pseudocode.

# Instruct a computer

- If we want to tell a computer to do something, we have to write a computer program that will tell the computer, step-by-step, exactly what we want it to do and how we want it to do it.
- **This step-by-step program will need planning, and to do this we use an algorithm.**
- Computers are only as good as the algorithms they are given.
- If you give a computer a poor algorithm, you will get a poor result — hence the phrase: ‘Garbage in, garbage out.’
- Algorithms are used for many different things including calculations, data processing and automation.

# Plan for solution

- It is important to plan out the solution to a problem to make sure that it will be correct.
- Using computational thinking and decomposition we can break down the problem into smaller parts and then we can plan out how they fit back together in a suitable order to solve the problem.
- This order can be represented as an algorithm.
- An algorithm must be clear.
- It must have a starting point, a finishing point and a set of clear instructions in between.

# Representing algorithms

- There are two main ways that algorithms can be represented – pseudocode and flowcharts.
- Most programs are developed using programming languages.
- These languages have specific syntax that must be used so that the program will run properly.
- **Pseudocode is not a programming language**, it is a simple way of describing a set of instructions that does not have to use specific syntax.
- Writing in pseudocode is similar to writing in a programming language.
- Each step of the algorithm is written on a line of its own in sequence.
- **Usually, instructions are written in uppercase, variables in lowercase and messages in sentence case.**

# A sample algorithm

*Scenario: I am walking to the shops, when a car pulls up beside me. The driver winds down the window. “Excuse me, how do I get to Kirkcroft Lane?”*

*I answer by giving an algorithm:*

1. Go down the hill.
2. Take the first major turning on the left.
3. Go up the hill.
4. Take the first turning on the left. (It is just after the Navigation pub).
5. You are there.

# A sample Pseudocode

OUTPUT 'What is your name?'

INPUT user inputs their name

STORE the user's input in the **name** variable

OUTPUT 'Hello' + name

OUTPUT 'How old are you?'

INPUT user inputs their age

STORE the user's input in the **age** variable

IF age  $\geq$  70 THEN

    OUTPUT 'You are aged to perfection!'

ELSE

    OUTPUT 'You are a spring chicken!'

✓ *What does this algorithm perform?*

*To ask someone their name and age, and to make a comment*

# Try this.....

- Write an algorithm to arrange ten children standing in the assembly in their height order.



# Representing an algorithm: Flowcharts






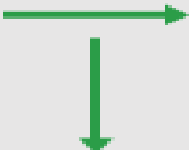
- A flowchart is a diagram that represents a set of instructions.
- Flowcharts normally use standard symbols to represent the different instructions.
- There are few real rules about the level of detail needed in a flowchart.
- Sometimes flowcharts are broken down into many steps to provide a lot of detail about exactly what is happening.
- Sometimes they are simplified so that a number of steps occur in just one step.

# Relationship between *rules* and *algorithms*

- ✓ *Not all lists of rules are algorithms.*

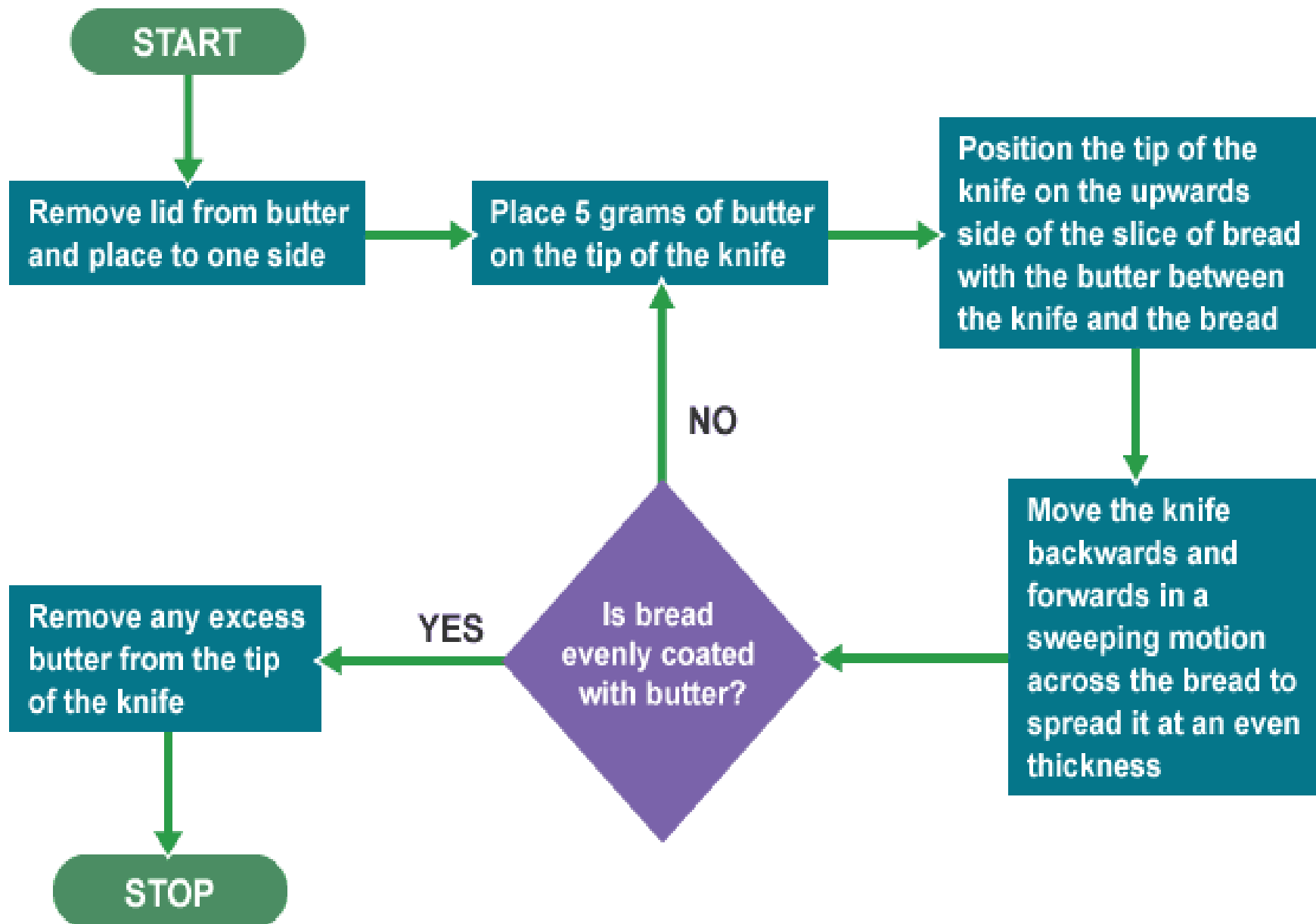
Example: Rules on toilet of in an airliner.

- No smoking.
- No eating.
- Do not use when the fasten seat belt sign is lit.
- But the following rules in an airliner exit door is an algorithm,  
*Emergency opening:*
  1. Pull cover aside.
  2. Push lever to open position and release.
  3. Push door outwards.

Name	Symbol	Usage
Start or Stop		The beginning and end points in the sequence.
Process		An instruction or a command.
Decision		A decision, either yes or no.
Input or Output		An input is data received by a computer. An output is a signal or data sent from a computer.
Connector		A jump from one point in the sequence to another.
Direction of flow		Connects the symbols. The arrow shows the direction of flow of instructions.

# A simple command like '*spread butter on bread*'

- Remove lid from butter tub and place to one side.
- Place 5 grams of butter on the tip of the knife.
- Position the tip of the knife on the upwards side of the slice of bread with the butter between the knife and the bread.
- Move the knife backwards and forwards in a sweeping motion across the bread to spread it at an even thickness.
- Repeat steps 2 to 4 until one side of the slice of bread is evenly coated with butter.
- Remove any excess butter from the tip of the knife.



A simple program could be created to ask someone their name and age, and to make a comment based on these. This program represented in pseudocode would look like this:

OUTPUT 'What is your name?'

INPUT user inputs their name

STORE the user's input in the **name** variable

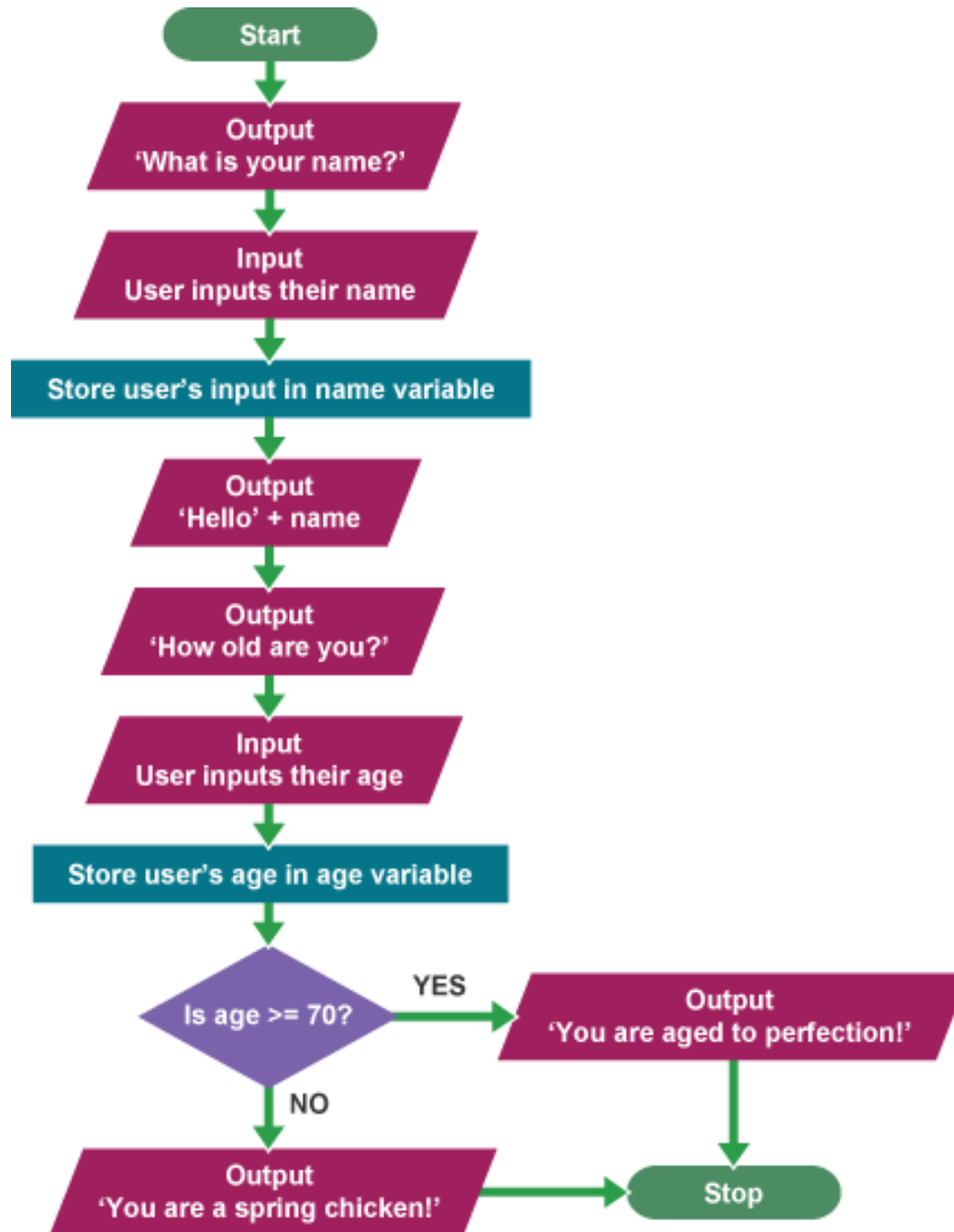
OUTPUT 'Hello' + name

OUTPUT 'How old are you?'

INPUT user inputs their age STORE the user's input in the **age** variable

IF age  $\geq$  70 THEN OUTPUT 'You are aged to perfection!'

ELSE OUTPUT 'You are a spring chicken!'



# An Algorithm for dealing with a non-functioning bulb

## *Pseudocode*

If the lamp plugged in  
Then  
    If the bulb burned out  
    Then  
        Replace bulb  
    Else  
        Buy new lamp  
Else  
    Plugin the lamp

## *Flowchart*

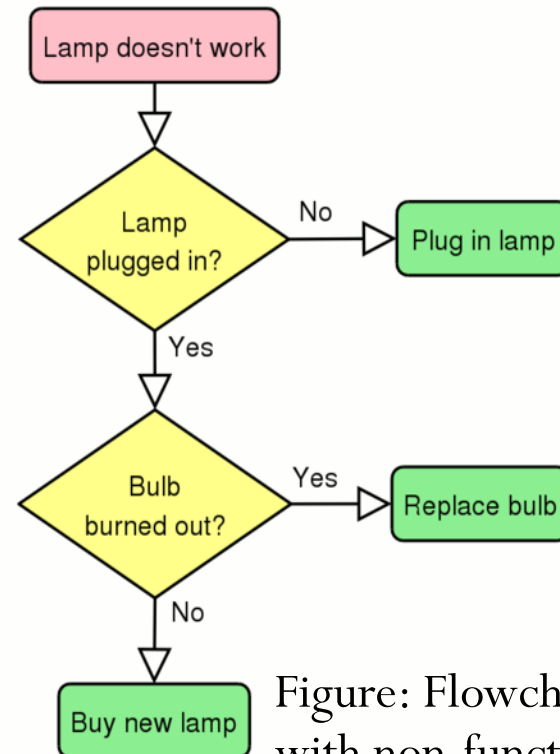
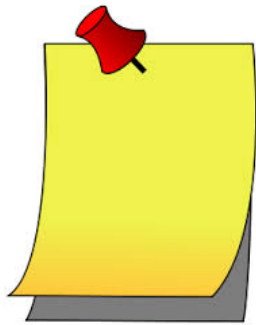


Figure: Flowchart for dealing with non-functioning bulb

✓ *Did you notice different shapes/symbols in the flowchart?*





✓ *Algorithms are about HOW we perform something.*

✓ *It is not about WHAT we want to perform...*



✓ *inputs - ingredients and quantities*

✓ *the process - recipe or method*

✓ *output - what the finished sandwich / cake will be like*



# Flowchart for a move in snake-ladder game

- *Can you fill the flowchart with the appropriate actions given below?*

- *Landed on snake head?*
- *Climb-up ladder*
- *Throw dice*
- *Move the piece*
- *At the bottom of ladder*

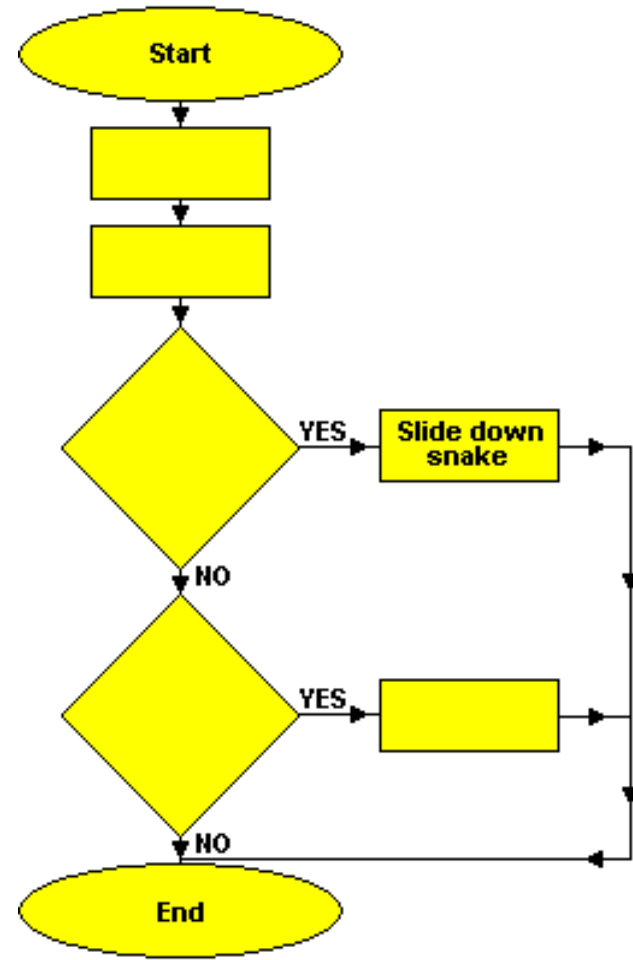


Figure: Flowchart for a move in snake-ladder game

# Control Abstraction

- In Algorithms, Control Structure a mechanism for specifying the proper order in which instructions must be performed.
- Types include:
  1. Selection
  2. Repetition
  3. Sequential control
  4. *Control abstraction* — is name which has well defined operation(use of sub problem).
  5. Concurrency (not in scope of this course)

✓ *Identify the control structures of the recipes available in next two slides*

# Flowchart structures

- Sequence
  - Series of actions performed in sequence.
- Selection
  - Selecting one of two possible actions depending on a condition.
- Iteration
  - Repeating actions.

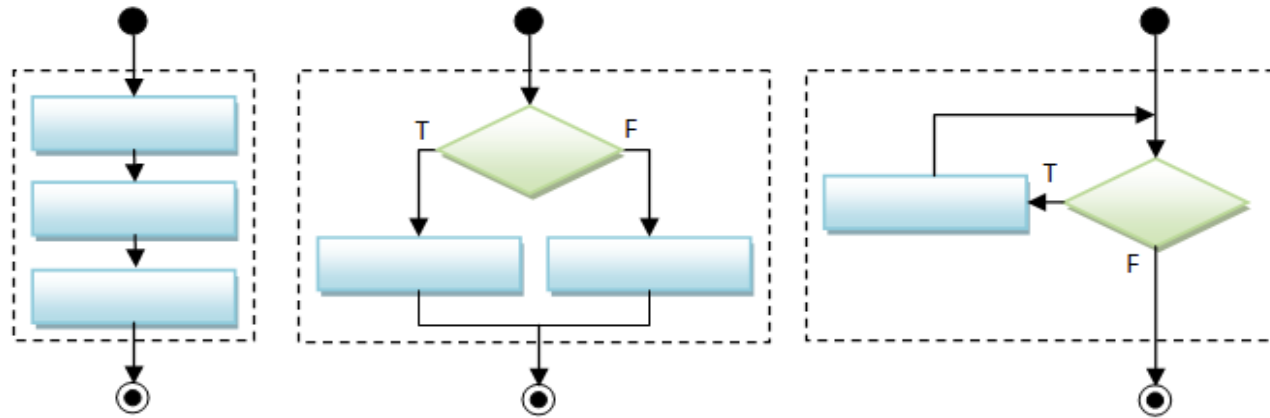
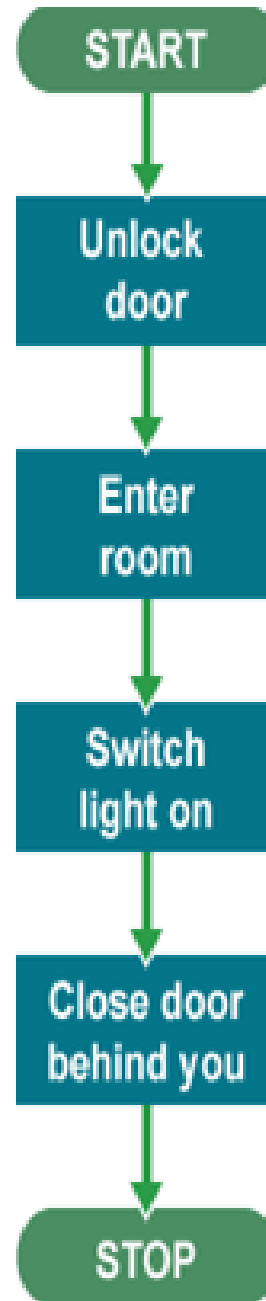


Figure: Sequential, selection and iteration structures

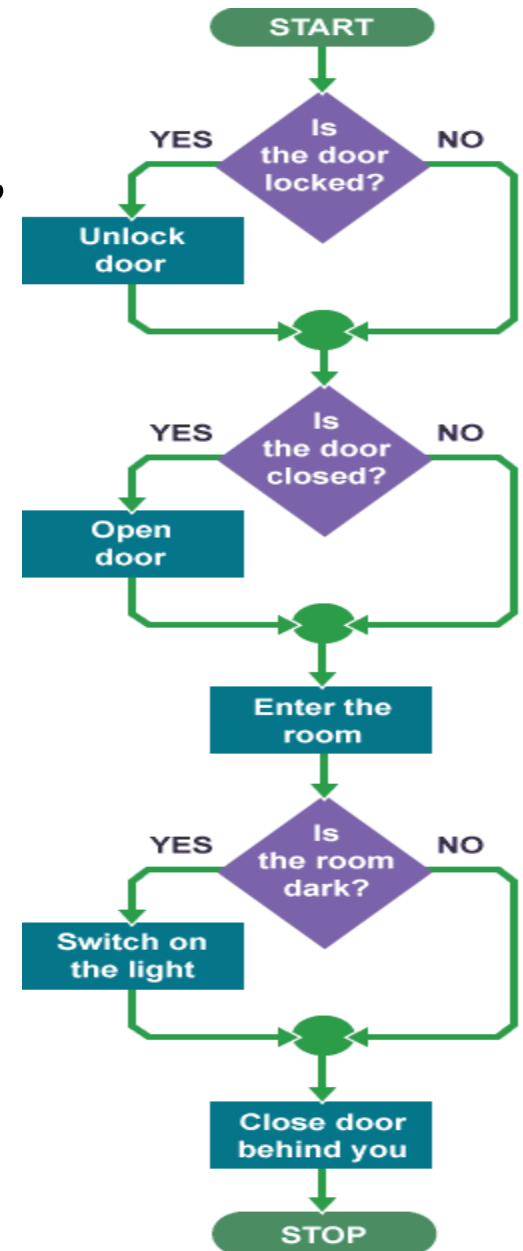
# Sequence

- unlock the door
- open the door
- enter the room
- switch on the light
- close the door behind you

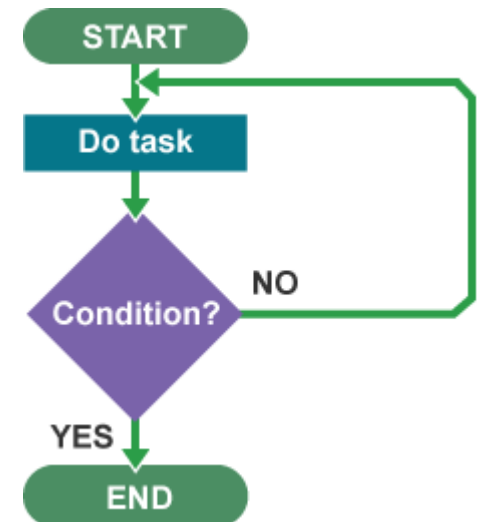
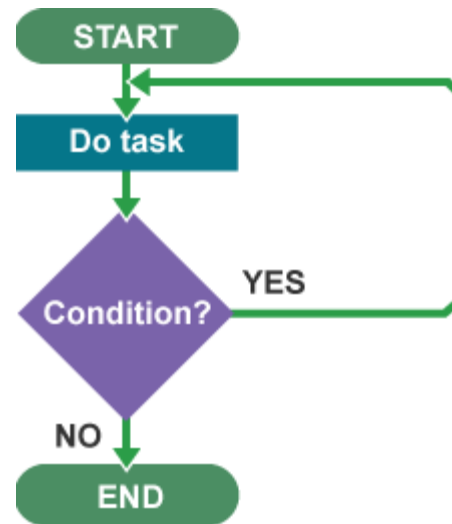
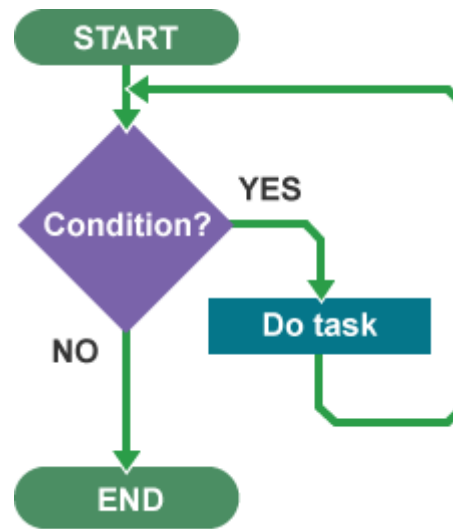
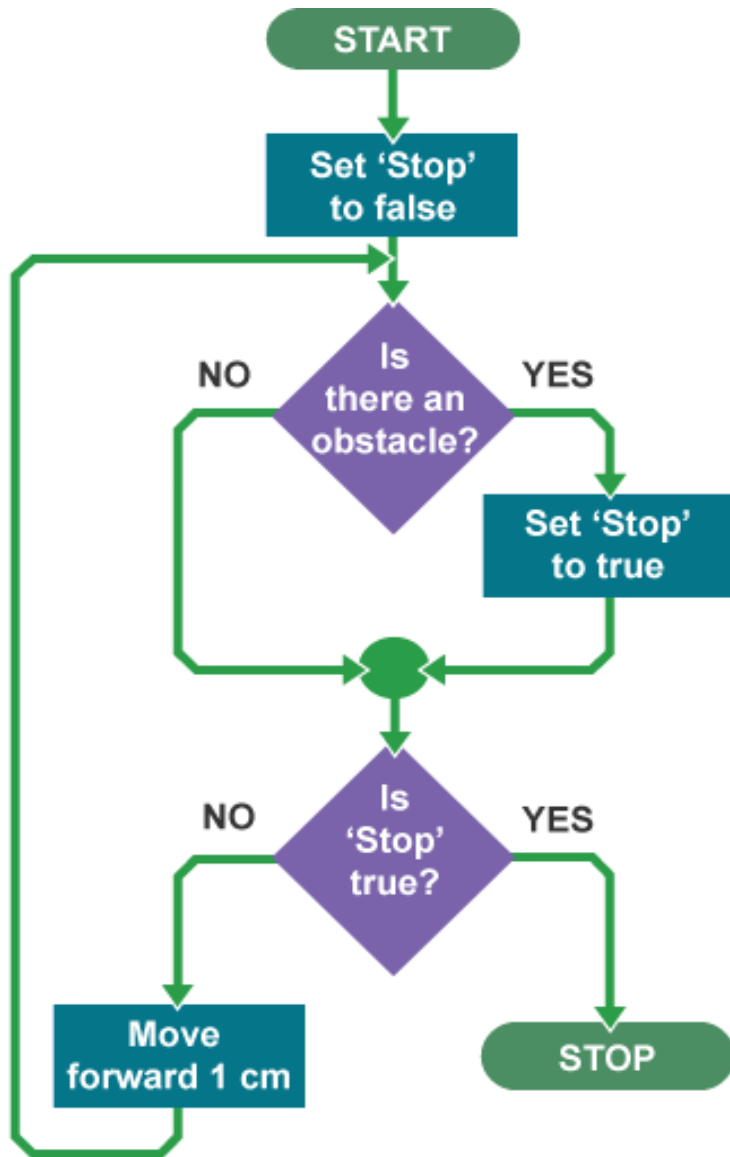


# Selection

- **IF** the door is locked, **THEN** unlock the door, **ELSE** do nothing (go to next instruction)
- **IF** the door is closed, **THEN** open the door, **ELSE** do nothing
- Enter the room
- **IF** the room is dark, **THEN** switch on the light, **ELSE** do nothing
- Close the door behind you



# Iteration





# Flowchart for classifying angles

- *Can you fill the flowchart with appropriate angles? (reflex angle, obtuse angle, angle on straight line, right angle, acute angle).*

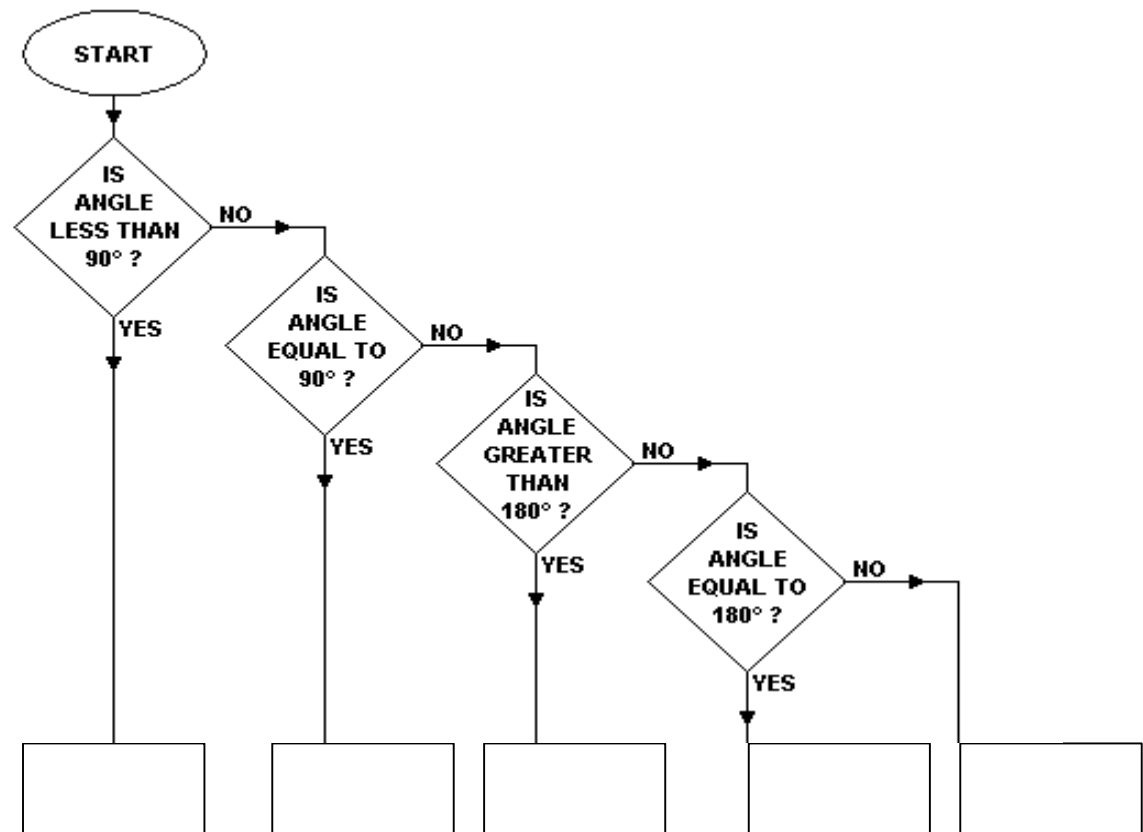


Figure: Flowchart for classifying angles

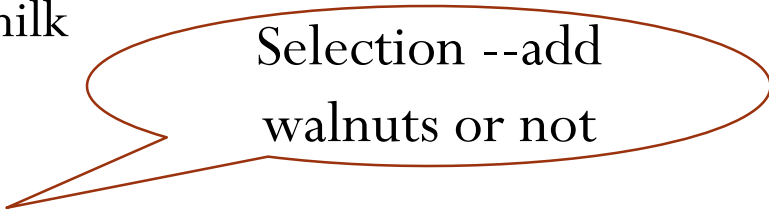


1. Place the following ingredients in a microwavable bowl:

3 cups of semisweet chocolate chips

one 14 oz. can of sweetened condensed milk

54 cup of butter



Selection --add  
walnuts or not

2. If desired. stir in 1 cup of walnut pieces.

3. Zap in microwave for one minute.



Repetition

4. Remove from microwave and stir the mixture.

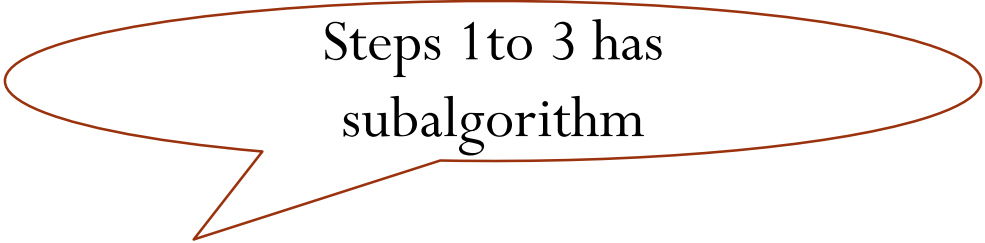
5. Repeat Steps 3 and 4 until chocolate chips are completely melted.

6. Stir 1 teaspoon of vanilla into mixture.

7. Pour mixture into a greased 8 by 8 dish.

8. Refrigerate for three hours.

9. Cut fudge into 1 inch squares.



Steps 1 to 3 has  
subalgorithm

1. Make fudge (for algorithm see previous slide).
2. Make chocolate chip cookies.
3. Make peanut butter bars.
4. Arrange fudge, cookies, and bars on a large tray.

The following algorithms each use data values.

Can you work out which of these is using fixed values and which is using variable values?



- **Algorithm A**  
Add 4 to 5  
Print the result
- **Algorithm B**  
Ask the user to type in a whole number  
Add 5 to that number  
Print the result
- **Algorithm C**  
If 4 is less than 5 Add 5 to 4  
Print the result Otherwise Print "no"
- **Algorithm D**  
Ask the user to type in a whole number  
If the number typed in is less than 5  
Add 5 to the number typed in Print the result  
Otherwise Print "no"

# Sequence structure

- Sequence
  - Series of actions performed in sequence.
- Selection
  - Selecting one of two possible actions depending on a condition.

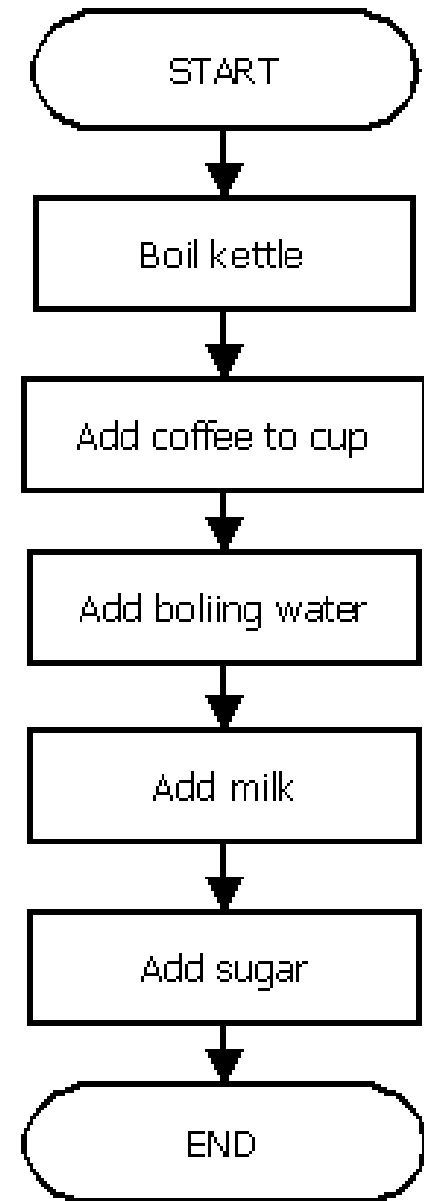
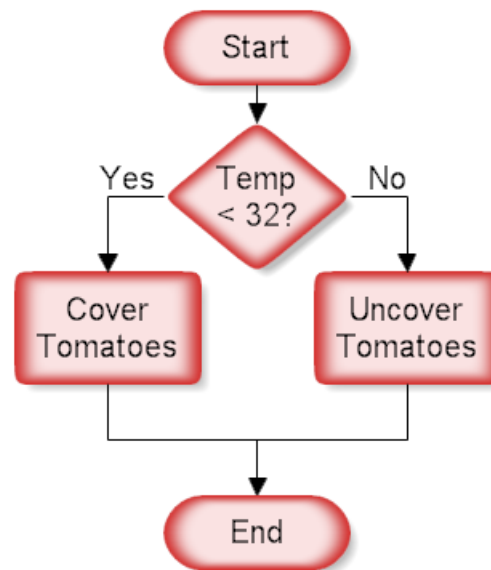


Figure: A logic requiring selection      Figure: Sequential logic algorithm

# Iteration structure

- Iteration
  - Repeating actions.
  - A loop tests a condition and if it is satisfied, performs an action. Then it tests the condition again. If the condition is still satisfied, the action is repeated. This is repeated until the condition is not satisfied.

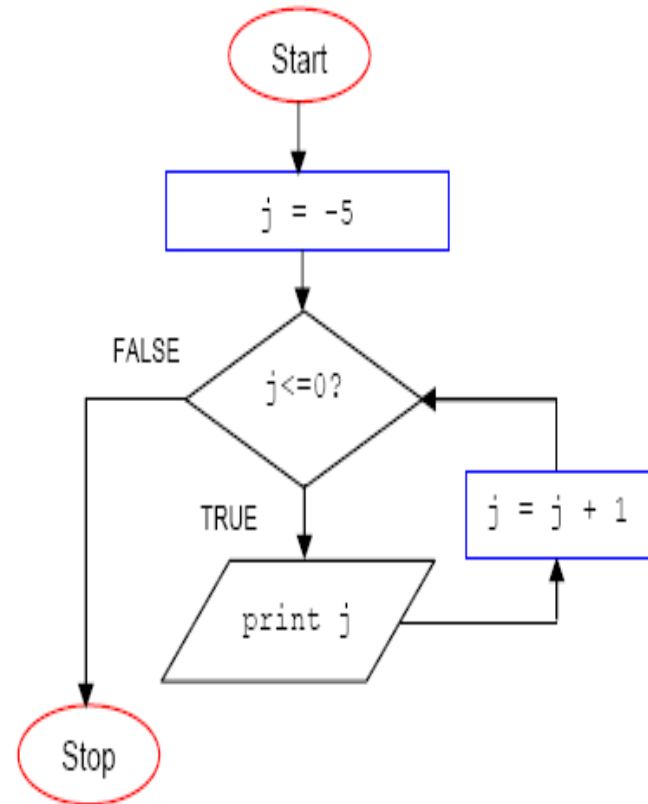


Figure: A logic requiring iteration structure

# Iteration structure – inverse condition



- Selection

- Iteration can still be realized with inverse condition satisfaction i.e. repeat action as long as the condition is not satisfied and stop once the condition is satisfied as the figure shows .

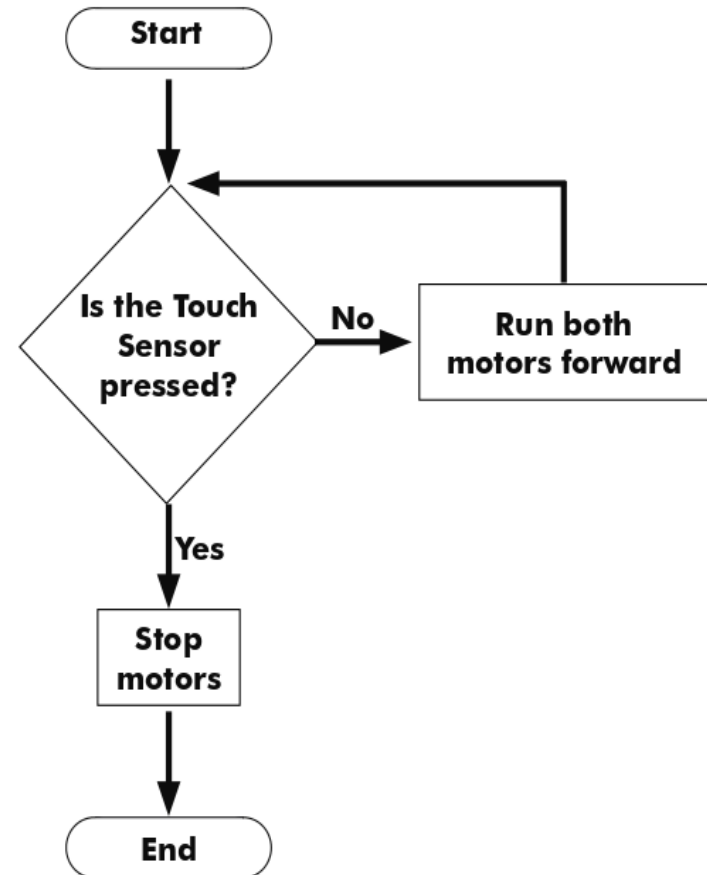


Figure: Iteration with inverse condition



# Controlling iteration structure

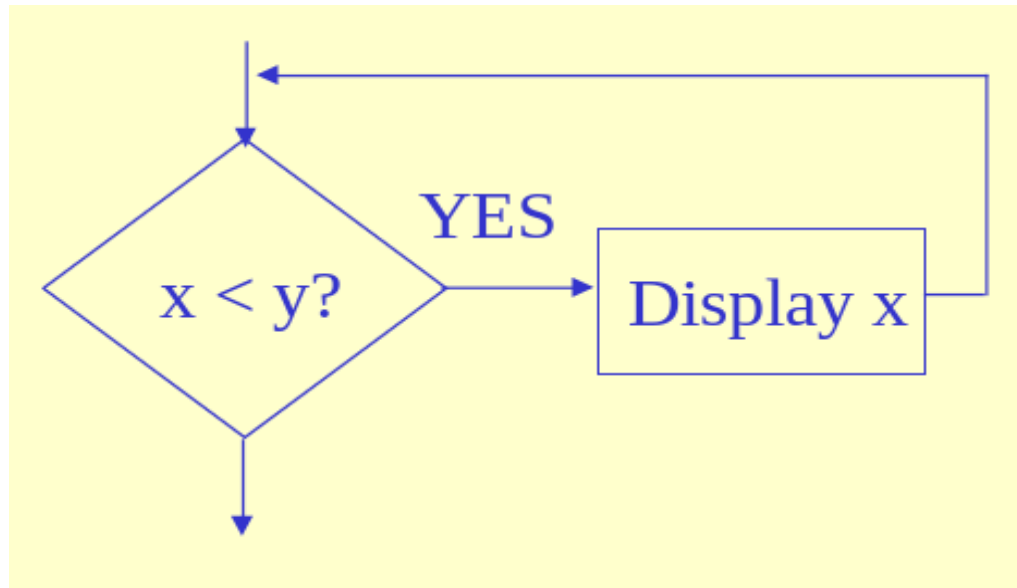


Figure: A simple task with iteration structure

✓ *Do you see any problem with the above iteration structure?*

# Connectors

- Sometimes a flowchart may exceed a page.
- Connectors allow you to connect to flowchart segments.
- Connectors need to be uniquely named for clarity.

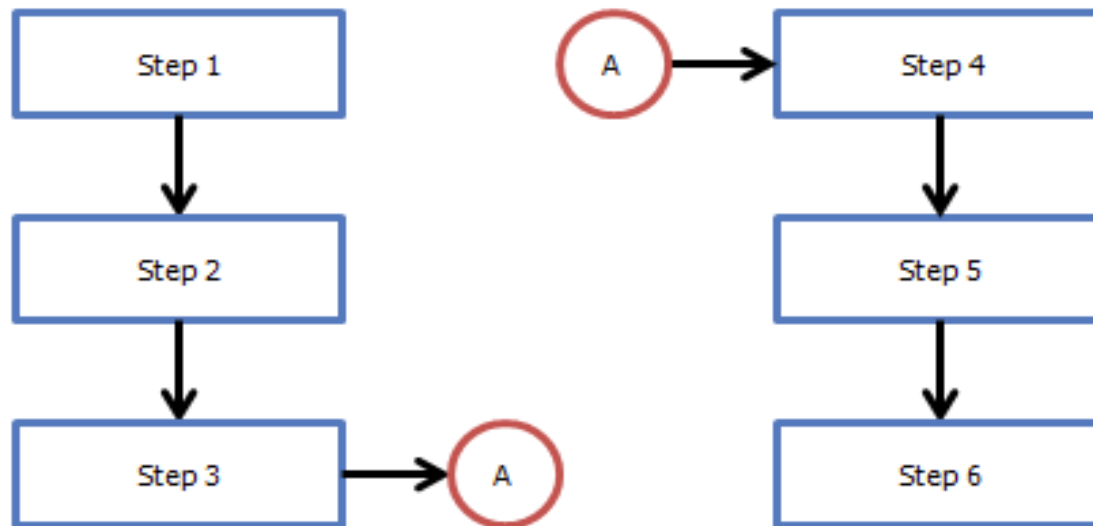


Figure: Connectors connect flowchart segments as well flowcharts across pages



# Algorithm for accept a number from user and calculate sum of square:

[Sum of square : accept number num from user, and set  $\text{sum}=0$  and calculate sum of square.]

Step 1. Start

Step 2. Read number num

Step 3. [Initialize]

$\text{sum}=0, i=1$

Step 4. Repeat step 4 through 6 until  $i \leq \text{num}$

Step 5.  $\text{sum}=\text{sum}+(i*i)$

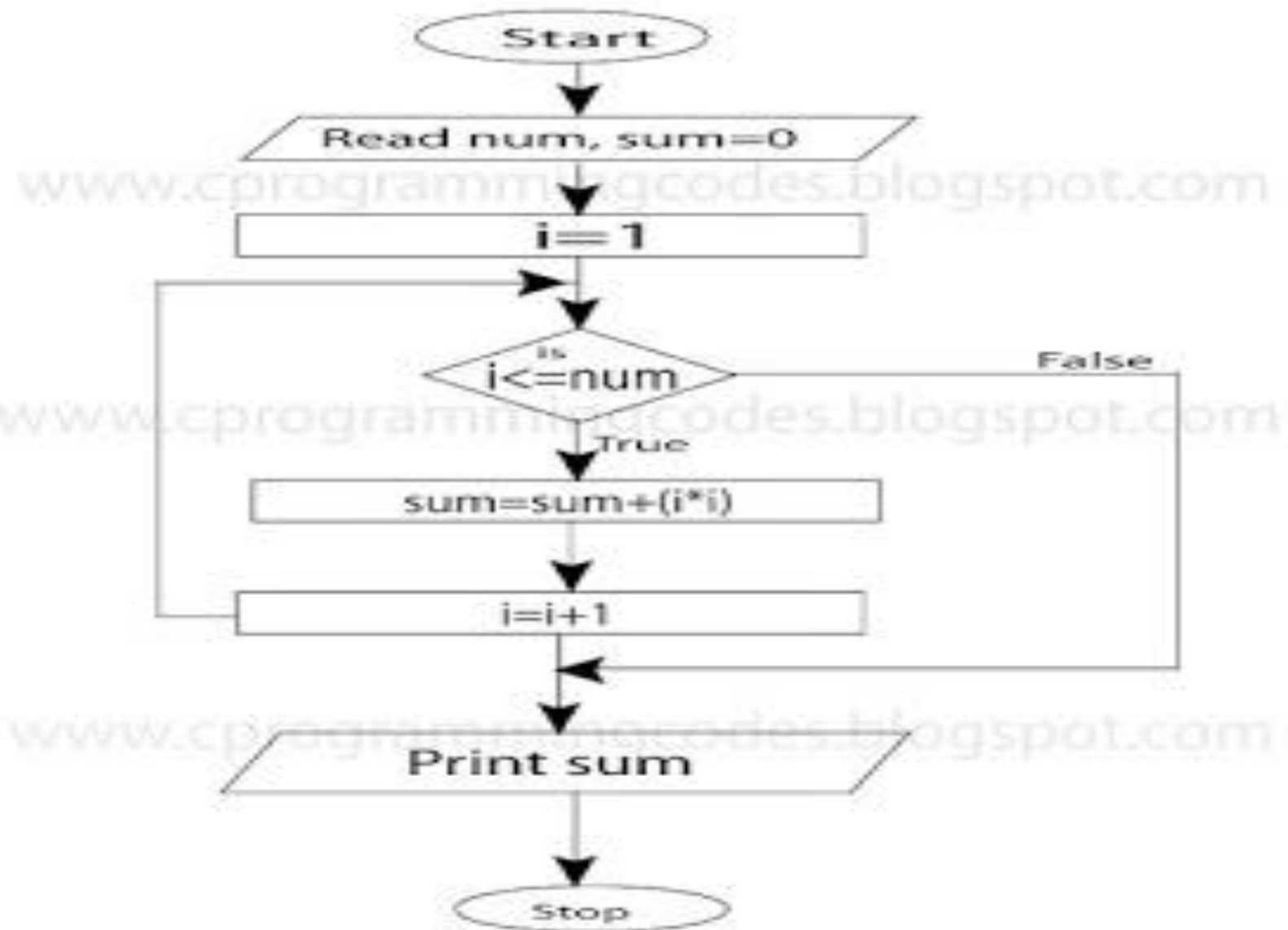
Step 6.  $i=i+1$

Step 7. print the sum of square

Step 8. stop

[end of loop step 4]

[end of sum of square]



Roots of a quadratic equation  $ax^2+bx+c=0$ , with the coefficients  $a$ ,  $b$ , and  $c$  defined from user input.

Input: quadratic equation coefficients  $a$ ,  $b$ , and  $c$  in real type

Output: the roots of the equation

Case#1: input:  $a=0, b=0, c=1$   
SOLUTION."

output: "Ill-posed equation. NO

Case#2: input:  $a=0, b=0, c=0$   
complex numbers"

output: "SOLUTION: any real or

Case#3: input:  $a=0, b=2, c=4$   
Solution: 2.0"

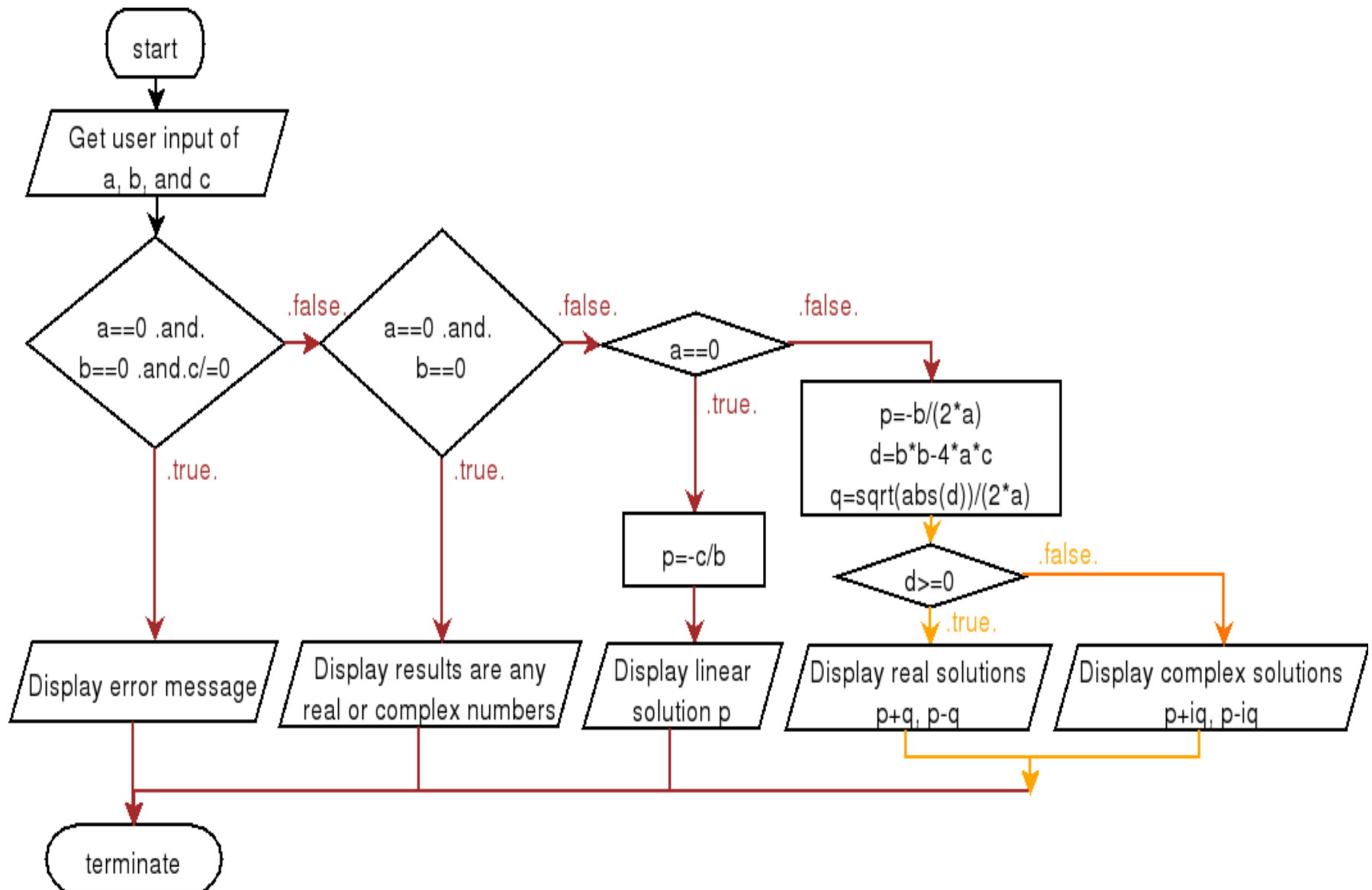
output: "Linear Equation, Unique

Case#4: input:  $a=1, b=-2, c=1$   
Solutions:  $x_1=1.0, x_2=1.0$ "!

output: "Quadratic Equation, Real

Case#5: input:  $a=1, b=0, c=1$   
Solutions:  $x_1=0.0+i1.0, x_2=0.0-i1.0$ "

output: "Quadratic Equation, Complex



# Office Assistant

Suppose that you need to hire a new office assistant. Your previous attempts at hiring have been unsuccessful, and you decide to use an employment agency. The employment agency sends you one candidate each day. You interview that person and then decide either to hire that person or not. You must pay the employment agency a small fee to interview an applicant. To actually hire an applicant is more costly, however, since you must fire your current office assistant and pay a substantial hiring fee to the employment agency. You are committed to having, at all times, the best possible person for the job. Therefore, you decide that, after interviewing each applicant, if that applicant is better qualified than the current office assistant, you will fire the current office assistant and hire the new applicant. You are willing to pay the resulting price of this strategy, but you wish to estimate what that price will be

# Algorithm

HIRE-ASSISTANT( $n$ )

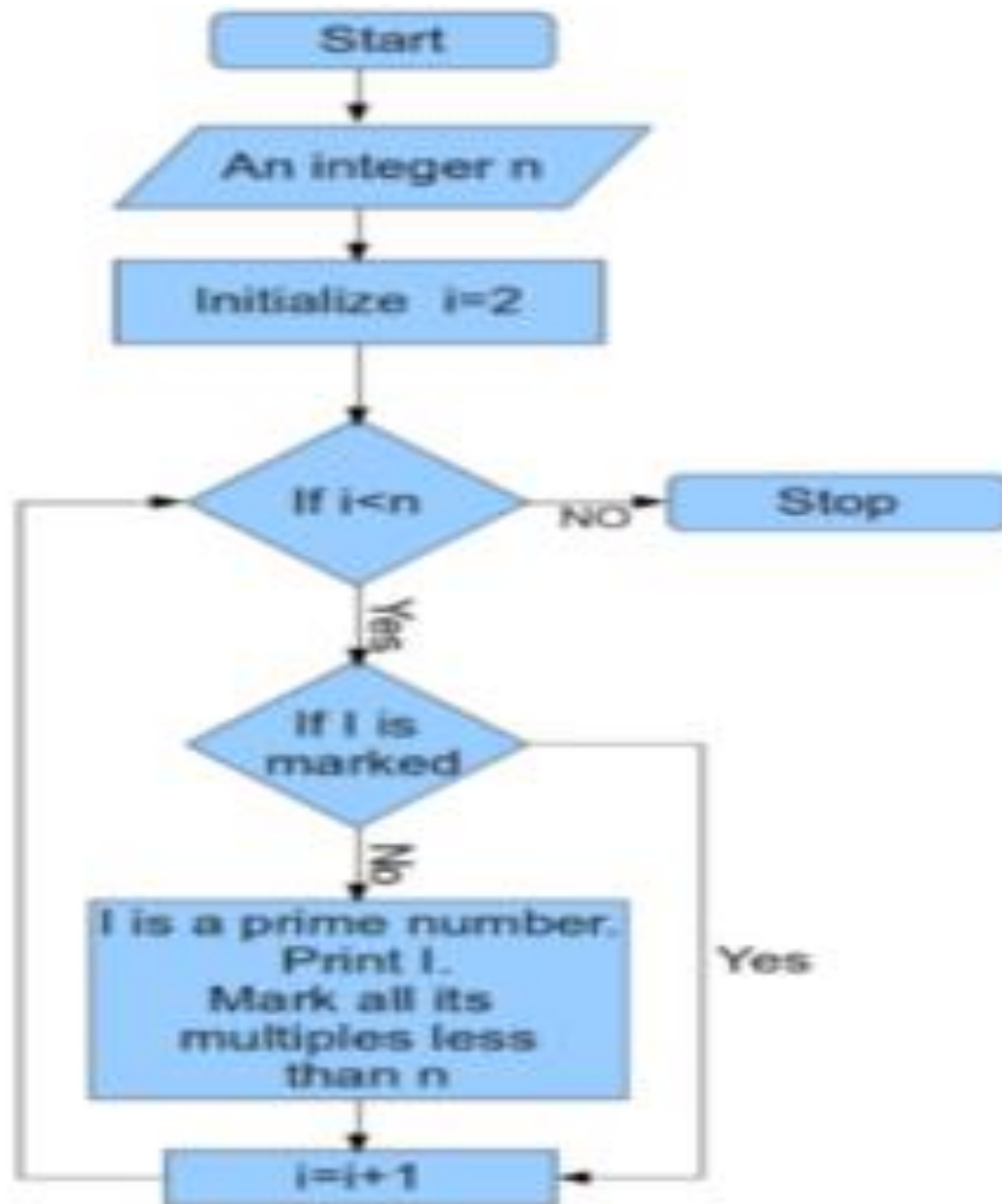
```
1   $best \leftarrow 0$        $\triangleright$  candidate 0 is a least-qualified dummy candidate
2  for  $i \leftarrow 1$  to  $n$ 
3      do interview candidate  $i$ 
4          if candidate  $i$  is better than candidate  $best$ 
5              then  $best \leftarrow i$ 
6              hire candidate  $i$ 
```

- The procedure HIRE-ASSISTANT, expresses the strategy for hiring in pseudocode.
- It assumes that the candidates for the office assistant job are numbered 1 through  $n$ .
- The procedure assumes that you are able to, after interviewing candidate  $i$ , determine if candidate  $i$  is the best candidate you have seen so far.
- To initialize, the procedure creates a dummy candidate, numbered 0, who is less qualified than each of the other candidates

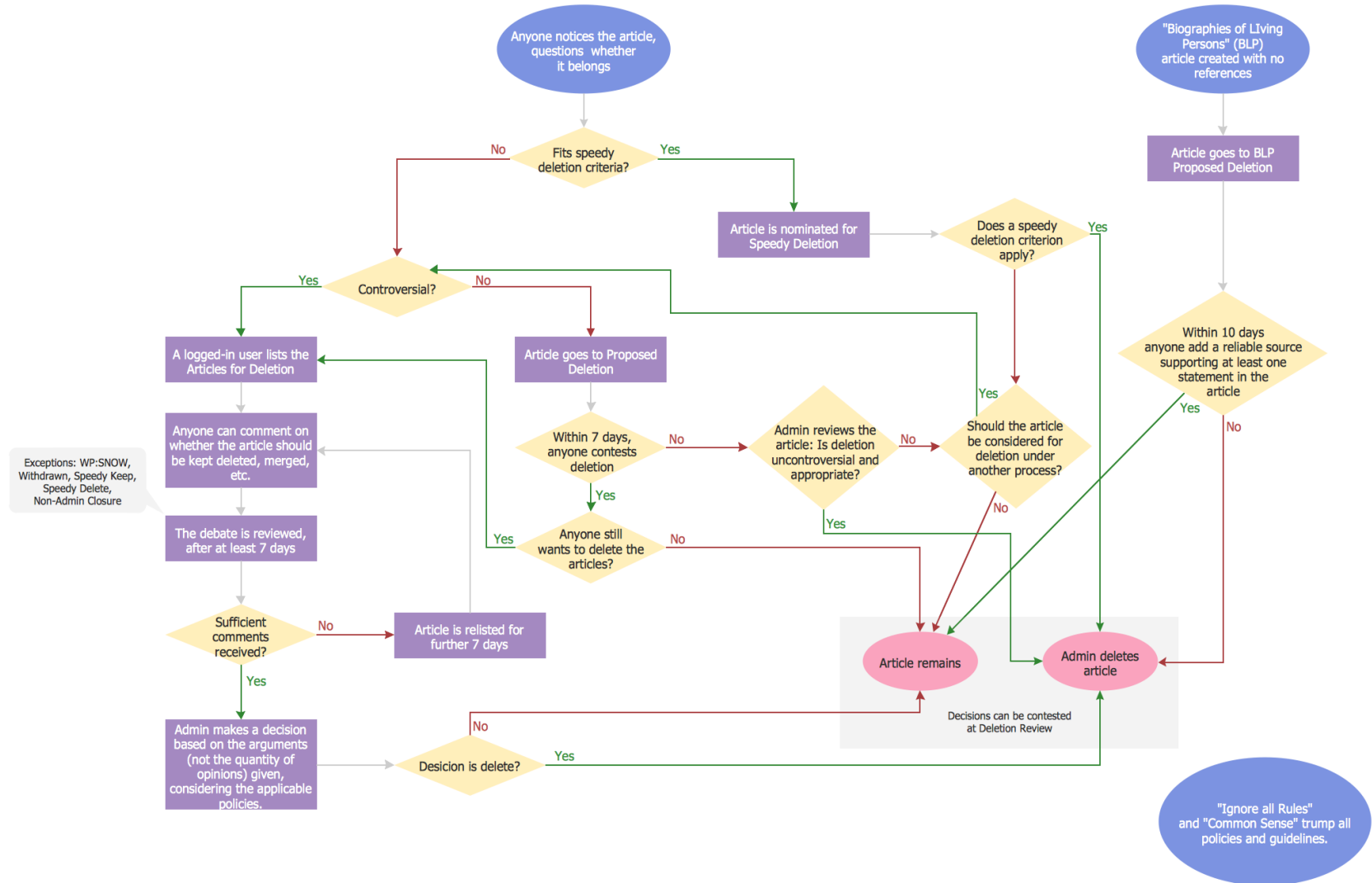
# Sieve-of-Eratosthenes-to find prime numbers up to an integer.

- Input to Sieve-of-Eratosthenes algorithm is an integer number and output is all prime numbers which is less than the input number.
- *Sample Input : 10*  
*Sample Output : 2, 3, 5, 7.*
- **Algorithm**
  - The algorithm takes a number as input. Assign this input to a variable 'limit'
  - Assign variable  $i=2$
  - Mark all numbers which is multiple of variable  $i$  and less than the 'limit'.
  - Update value of  $i$  as the next unmarked number.
  - If variable  $i$  is greater than the 'limit', then stop. Else go to step 3.





# Complex Flowcharts





## *Try it yourself*

- ✓ *Design a flowchart to convert a Fahrenheit temperature to celsius.*
- ✓ *Design a flowchart to find the largest of three numbers A, B, C.*
- ✓ *Design a flowchart for calculating a student's letter grade given the following mark ranges: (90 – 100: A; 80 – 89: B; 70 – 79: C, 60 – 69: D; below 60: F).*
- ✓ *Design a flowchart to print number sequence up to a given positive integer.*
- ✓ *Modify the flowchart that prints the number sequence to add all the numbers in the sequence.*
- ✓ *Design a flowchart to find the average age of your class.*



Case study time .....

# Drawing a Off-Road Vehicle



*Give an level-1 design to draw the Figure: Green Jeep*

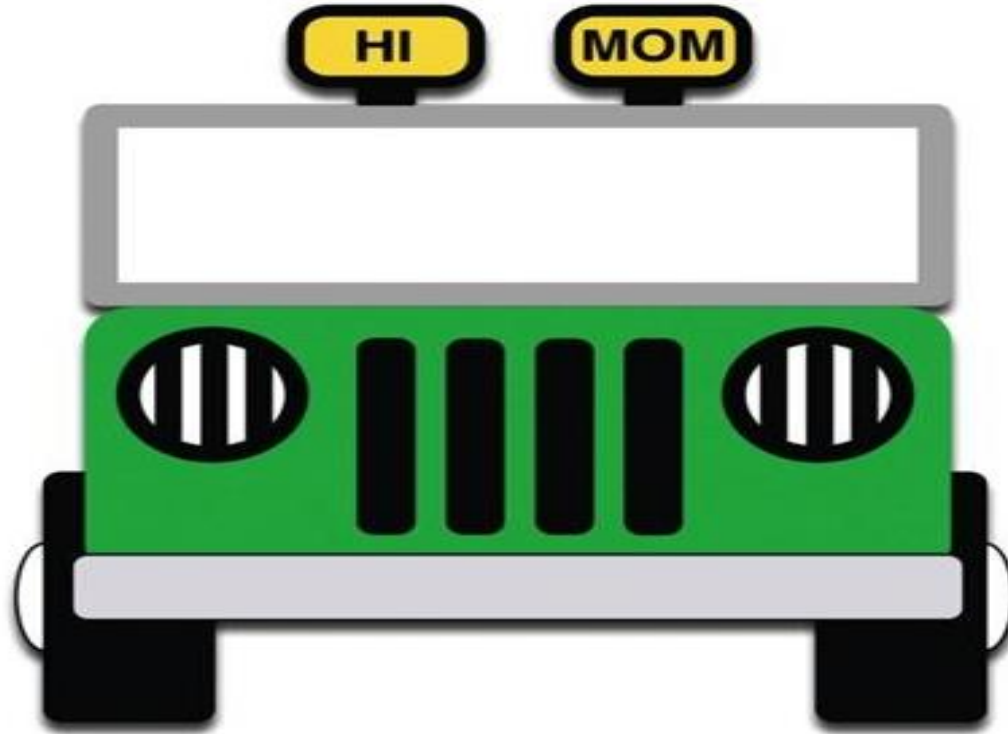


Figure: Green Jeep

# Drawing a Off-Road Vehicle- Level1

STEP-1: Draw a green grille.

STEP-2: Draw bumper just below the grille.

STEP-3: Draw the tires behind the grille and bumper.

STEP-4: Draw the windshield just above the grille.

STEP-5: Draw two auxiliary lights on top of the windshield.

# Drawing a Off-Road Vehicle- Level 2

- *Consider the instructions from the level-1 one by one and refine each instruction into parts.*
- *Consider expansion green grille drawing.*

STEP 1: Draw a green grille.

1.1 Draw a green grille background.

. 1.2 Draw the left headlight

1.3 Draw four equally spaced black vertical rectangles for grille.

- *Consider the details of drawing of tires.*

STEP 3: Draw the tires behind the grille and bumper.

- 3.1. Draw the left tire (a black rectangle) slightly protruding left of the grille and bumper.
- 3.2. Draw a half-moon hubcap extending outside the left tire.
- 3.3. Draw the right tire (a black rectangle) slightly protruding right of the grille and bumper.
- 3.4. Draw a half-moon hubcap extending outside the right tire.



- *Consider the details of drawing a windshield.*

STEP 4: Draw the windshield just above the grille.

4.1. Draw a gray rectangle for the outside of the windshield.

4.2. Draw a white rectangle for the center of the windshield.

- *Consider the details of drawing a auxiliary lights.*

STEP 5: Draw two auxiliary lights on top of the windshield.

5.1. Draw the left auxiliary light.

5.2. Draw the right auxiliary light.

# Drawing a Off-Road Vehicle- Final algorithm

STEP 1: Draw a green grille.

1.1. Draw a green grille background.

1.2. Draw the left headlight.

1.2.1. Draw black outer dot as a rim for the left headlight.

1.2.2. Draw white dot centered within the black rim.

1.2.3. Draw three equally spaced vertical black rectangles as  
headlight protectors.

1.3. Draw four equally spaced black vertical rectangles for grille openings.

1.4. Draw the right headlight.

1.4.1. Draw black outer dot as a rim for the right headlight.

1.4.2. Draw white dot centered within the black rim.

1.4.3. Draw three equally spaced vertical black rectangles as headlight protectors.

STEP 2: Draw bumper just below the grille.

STEP 3: Draw the tires behind the grille and bumper.

3.1. Draw the left tire (a black rectangle) slightly protruding left of the grille and bumper.

3.2. Draw a half-moon hubcap extending outside the left tire.

3.3. Draw the right tire (a black rectangle) slightly protruding right of the grille and bumper.

3.4. Draw a half-moon hubcap extending outside the right tire.

STEP 4: Draw the windshield just above the grille.

4.1. Draw a gray rectangle for the outside of the windshield.

4.2. Draw a white rectangle for the center of the windshield.

STEP 5: Draw two auxiliary lights on top of the windshield.

5.1. Draw the left auxiliary light.

5.1.1. Draw outer black rectangle as a rim for the left auxiliary light.

5.1.2. Draw yellow rectangle centered inside the black rim.

5.1.3. Place the word "HI" centered in the yellow rectangle.

5.1.4. Draw a small black rectangle for the base of the light.

## 5.2. Draw the right auxiliary light.

5.2.1. Draw outer black rectangle as a rim for the left auxiliary light.

5.2.2. Draw yellow rectangle centered inside the black rim.

5.2.3. Place the word "MOM" centered in the yellow rectangle.

5.2.4. Draw a small black rectangle for the base of the light.



Write the detailed algorithm for the following

Problem: An integer  $N$  is "perfect" if  $N$  is equal to the sum of the positive integers  $K$  such that  $K < N$  and  $K$  is a divisor of  $N$ .

Design an algorithm to determine if a given integer is "perfect".

The top-level sub-problems (or tasks) appear to be:

- I. Get the number which is to be tested.
- II. Determine the divisors of the number.
- III. Check if the divisors add up to the number.

# Solution

- I. Get the number which is to be tested.
  - A. Output: "Please enter a positive integer: "
  - B. Input the number; call it N.
  - C. If N isn't positive
    - i. Output: N " isn't perfect."
    - ii. STOP.
- II. Determine the divisors of the number.
  - A. Set the DivisorSum to 1. (Since 1 certainly divides N.)
  - B. Set D to 2.
  - C. While D is less than or equal to  $N / 2$ 
    - i. If D is a divisor of N
      - a. Add D to DivisorSum
    - ii. Add 1 to D.
- III. Check the results.
  - A. If DivisorSum equals N
    - i. Output: N " is perfect."Else
    - i. Output: N " isn't perfect."
  - B. STOP.



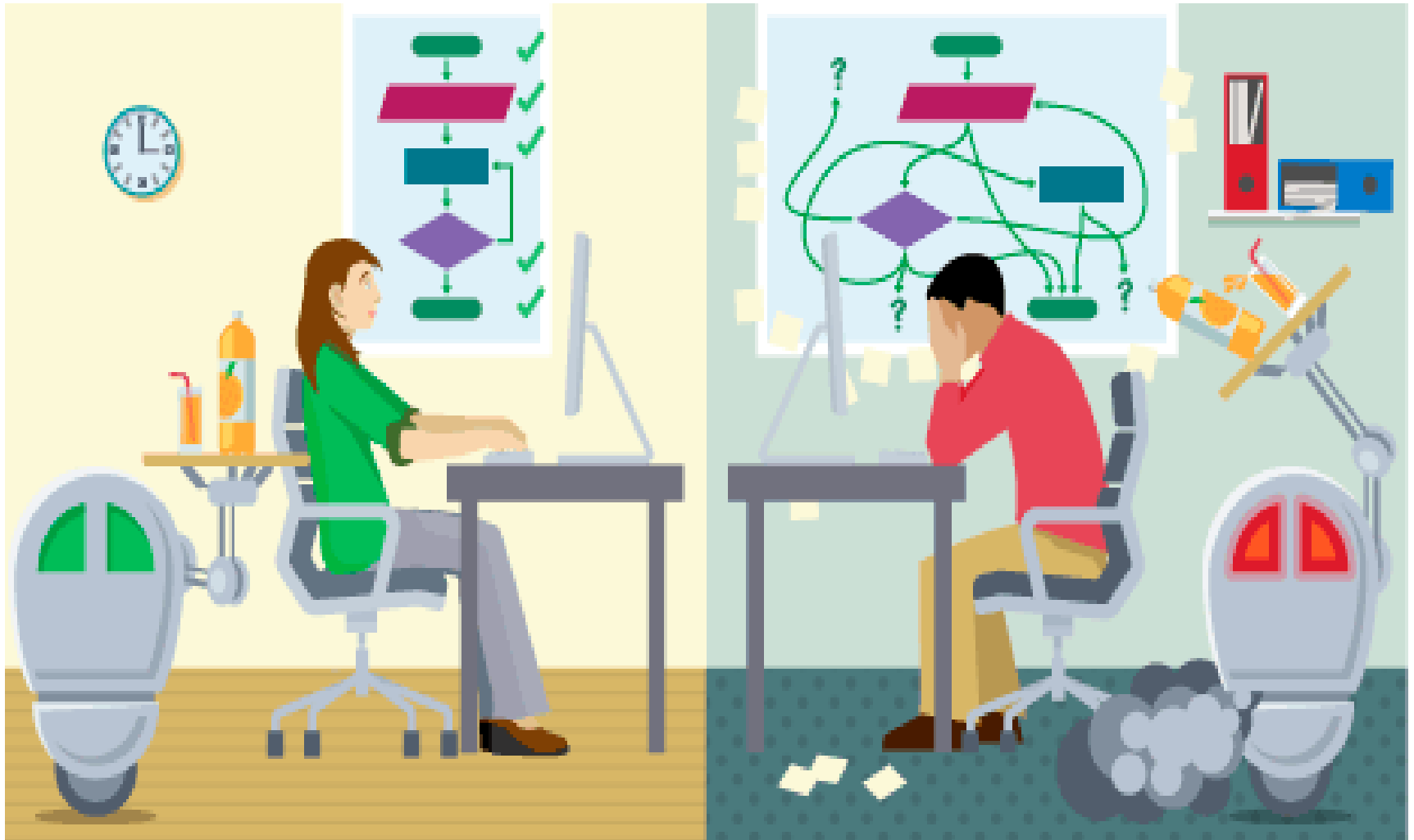
# Evaluating Solutions

- Before solutions can be programmed, it is important to make sure that it properly satisfies the problem, and that it does so efficiently.
- This is done through evaluation.

## What is evaluation?

- Once a solution has been designed using computational thinking, it is important to make sure that the solution is fit for purpose.
- Evaluation is the process that allows us to make sure our solution does the job it has been designed to do and to think about how it could be improved.

*Make sure that the solution is fit for purpose..... How ????*



# Evaluate the algorithm

Once written, an algorithm should be checked to make sure it:

- is **easily understood** – is it fully decomposed?
- is **complete** – does it solve every aspect of the problem?
- is **efficient** – does it solve the problem, making best use of the available resources (eg as quickly as possible/using least space)?
- meets any **design criteria** we have been given
- If an algorithm meets these four criteria it is likely to work well.
- The algorithm can then be programmed.

- ✓ *Failure to evaluate can make it difficult to write a program.*
- ✓ *Evaluation helps to make sure that as few difficulties as possible are faced when programming the solution.*

# Why is evaluation important?

- Computational thinking helps to solve problems and design a solution – an algorithm – that can be used to program a computer.
- However, if the solution is faulty, it may be difficult to write the program.
- Even worse, the finished program might not solve the problem correctly.
- Evaluation allows us to consider the solution to a problem, make sure that it meets the original design criteria, produces the correct solution and is fit for purpose - before programming begins.

# Faulty solutions do not solve the problem

- Once a solution has been decided and the algorithm designed, it can be tempting to miss out the evaluating stage and to start programming immediately.
- ✓ *However, without evaluation any faults in the algorithm will not be picked up, and the program may not correctly solve the problem, or may not solve it in the best way.*
- Faults may be minor and not very important.
  - For example, if a solution to the question ‘how to draw a cat?’ was created and this had faults, all that would be wrong is that the cat drawn might not look like a cat.
- However, faults can have huge – and terrible – effects, eg if the solution for an aeroplane autopilot had faults.

A faulty solution may include one or more of these errors.

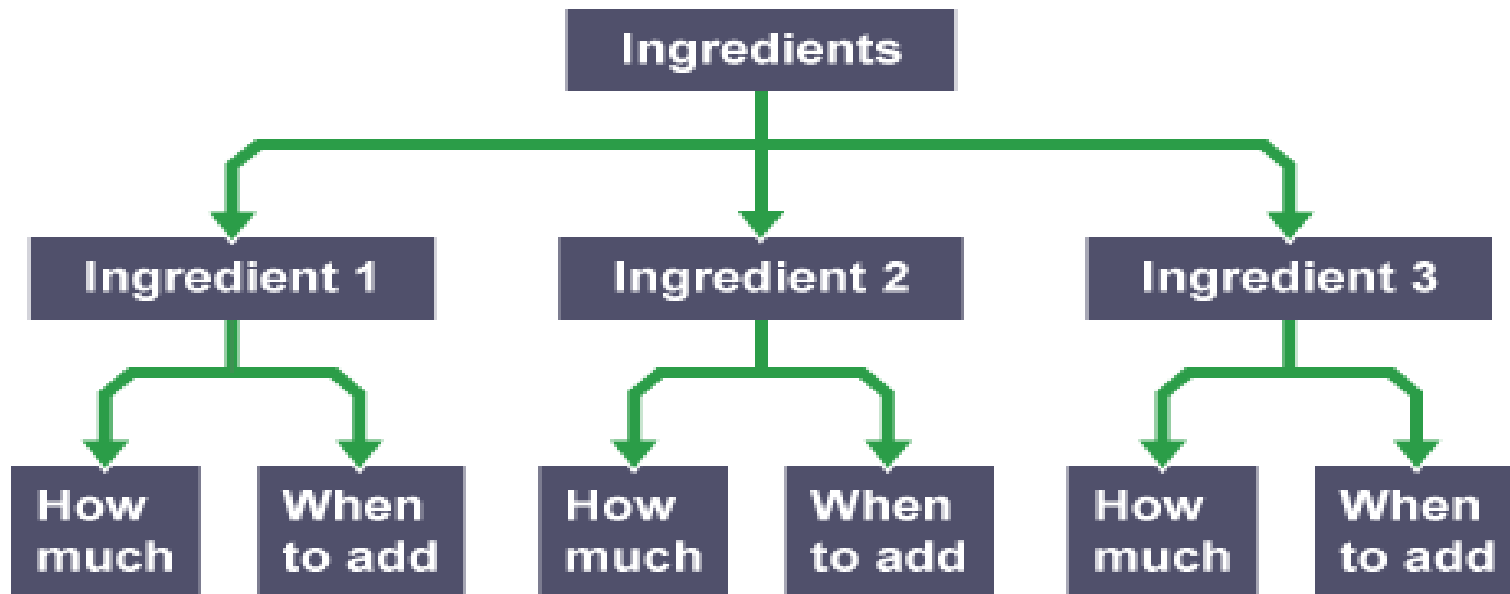
- We may find that solutions fail because:
  - it is **not fully understood** - we may not have properly **decomposed** the problem
  - it is **incomplete** - some parts of the problem may have been left out accidentally
  - it is **inefficient** – it may be too complicated or too long
  - it **does not meet the original design criteria** – so it is not fit for purpose

# A study of “Evaluation” in detail

- Consider the example “**How to bake a cake**”
- It is decomposed to:
  - what kind of cake to bake
  - what ingredients are needed, how much of each ingredient, and when to add it
  - how many people the cake is for
  - how long to bake the cake for
  - what equipment is needed

# #1 Solutions that are not properly decomposed

## ❑ The Decomposition of **ingredients**



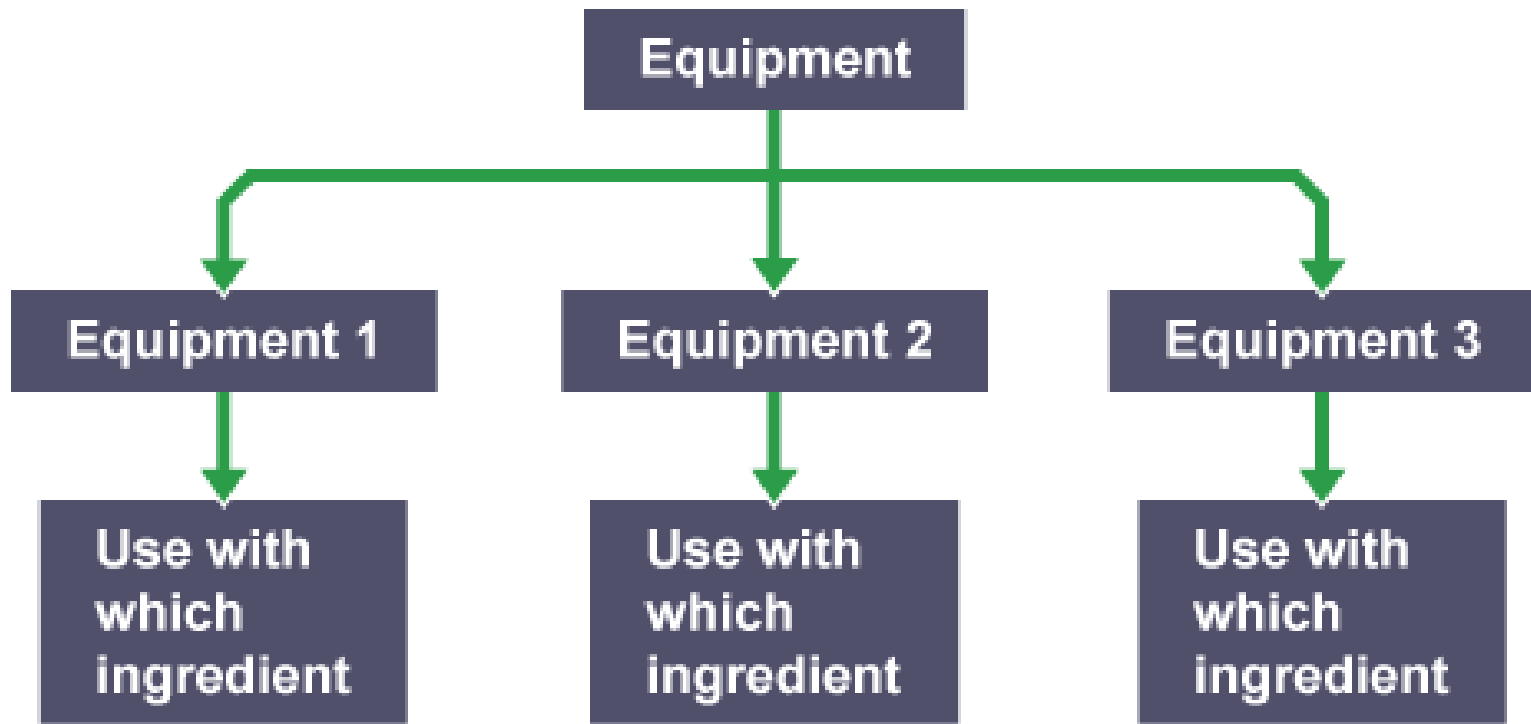
## ❑ The Decomposition of **equipment**





# Equipment Decomposition is incomplete

- The ‘Equipment’ part is not properly broken down (or decomposed).
- Therefore, if the solution - or algorithm – were created from this, baking the cake would run into problems.
- The algorithm would say what equipment is needed, but not how to use it, so a person could end up trying to use a knife to measure out the flour and a whisk to cut a lump of butter, for example.
- This would be wrong and would, of course, not work.
- Ideally, then, ‘Equipment’ should be decomposed further, to state which equipment is needed and which ingredients each item is used with.



- ✓ The problem occurred here because the problem of which equipment to use and which ingredients to use it with hadn't been fully decomposed.

## # 2 Solutions that are incomplete

- How to bake a cake decomposition **is still incomplete** – part of the problem has been left out. We still need to know:
  - where to bake the cake
  - what temperature to bake the cake at
- Therefore, if this information was used to create the solution, the **algorithm** would say how long the cake should be baked for but it would not state that the cake should be placed in the oven, or the temperature that the oven should be.
- Even if the cake made it to the oven, it could end up undercooked or burnt to a cinder.
- Very important factors have been left out, so the chances of making a great cake are slim.
- ✓ *The problem occurred here because placing the cake in the oven and specifying the oven temperature had not been included, making the solution incomplete.*



Preheat oven to 190°C

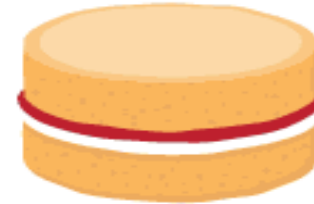
Blend butter, sugar & flour



Bake for 25 minutes



Whisk 300ml of cream



Preheat oven to 180°C



Whisk all butter and sugar

Mix in eggs



Bake for 30 minutes



Preheat oven to *temperature*



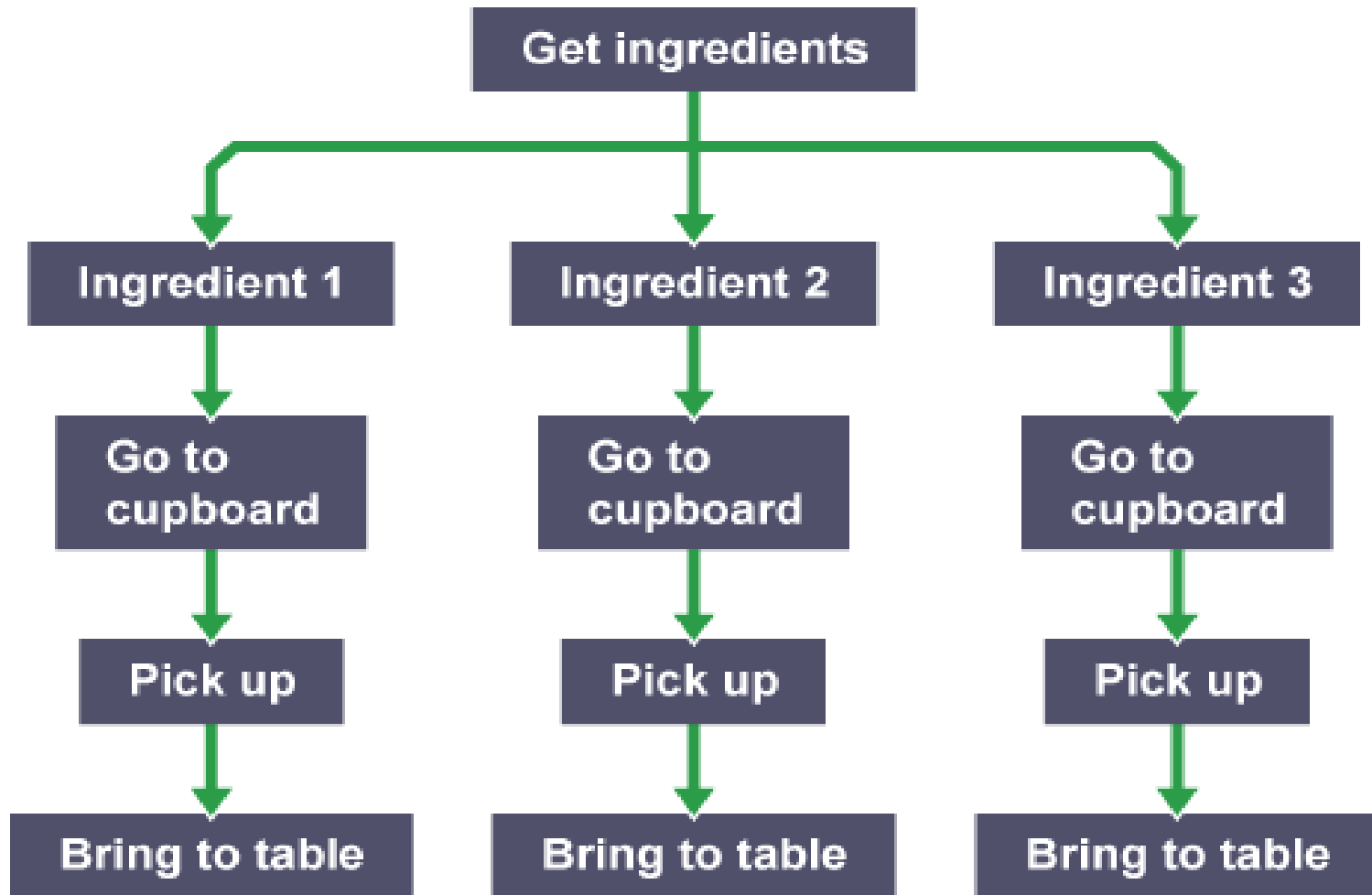
Whisk *quantity of ingredients*



Bake for *time*

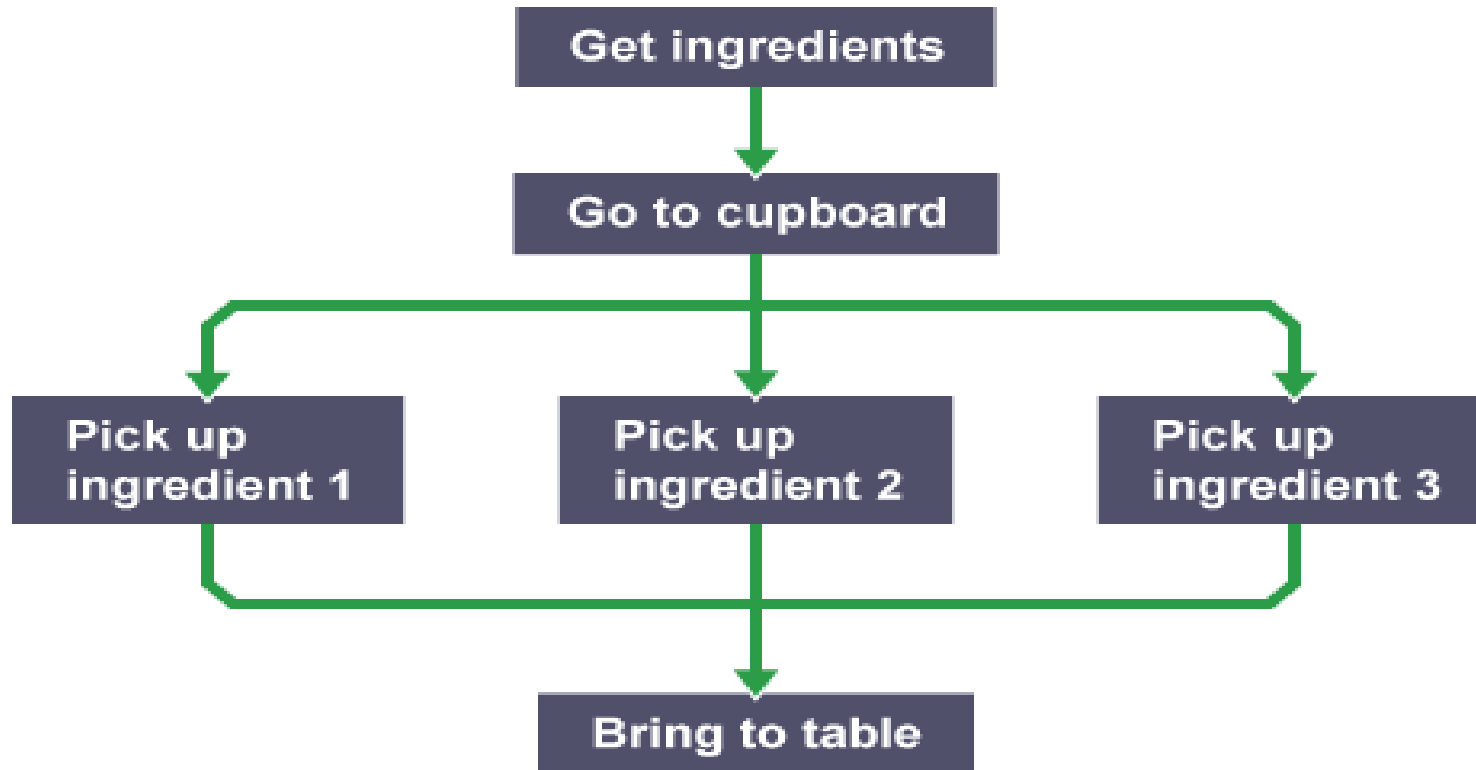
### #3 Solutions that are inefficient

- The solution would state — among other things — that certain quantities of particular ingredients are needed to make the cake.
  - For the first ingredient, it might tell us to go the cupboard, get the ingredient, and bring it back to the table.
  - For the second — and all other ingredients — It might tell us to do the same.
  - If the cake had three ingredients, that would mean three trips to the cupboard.



✓ *While this would work, it would be unnecessarily long and complicated*

- ✓ *It would be more efficient to fetch all the ingredients in one go, and the program would be shorter as a result*



- ✓ *The solution is now simpler and more efficient, and has reduced from nine steps to five.*
- ✓ *The problem occurred here because some steps were repeated unnecessarily, making the solution inefficient and overly long.*

## #4 Solutions that do not meet the original design criteria

- Solutions should be evaluated against the original specification or design criteria where possible.
- This makes sure that the solution has not strayed too much from what was originally required, that it solves the original problem and that it is suitable for users.
- The first point in the decomposition considers what kind of cake to bake.
- Often, when devising solutions to problems, a specification for the design is given.



- For example, the cake may have to be a chocolate cake, which is still quite general, or a chocolate fudge cake with chocolate icing and flakes on top, which is more specific.
  - To meet the design criteria, it is important to ensure that the exactly right kind of cake is baked.
  - Otherwise the solution may not be fit for purpose.
- ✓ *The problem occurred here because the solution did not meet the original design criteria – it was not exactly what was requested.*

# How do we evaluate our solution?

- ✓ *There are several ways to evaluate solutions.*
- ✓ *To be certain that the solution is correct, it is important to ask:*

## (1) Does the solution make sense?

- Do you now fully understand how to solve the problem?
- If you still don't clearly know how to do something to solve our problem, go back and make sure everything has been properly decomposed.
- Once you know how to do everything, then our problem is thoroughly decomposed.

## **(2) Does the solution cover all parts of the problem?**

- For example, if drawing a cat, does the solution describe everything needed to draw a cat, not just eyes, a tail and fur?
- If not, go back and keep adding steps to the solution until it is complete.

## **(3) Does the solution ask for tasks to be repeated?**

- If so, is there a way to reduce repetition?
- Go back and remove unnecessary repetition until the solution is efficient.

*✓ Once you're happy with a solution, ask a friend to look through it.*

*✓ A fresh eye is often good for spotting errors.*

# Dry runs

- One of the best ways to test a solution is to perform what's known as a 'dry run'.
- With pen and paper, work through the algorithm and trace a path through it.
- Dry runs are also used with completed programs.
- Programmers use dry runs to help find errors in their program code.

- ✓ A simple algorithm was created to ask someone their name and age, and to make a comment.
- ✓ You could try out this algorithm — give it a dry run.
  - Try two ages, 15 and 75.
  - When using age 75, where does the algorithm go?
    - Does it give the right output?
  - If you use age 15, does it take you down a different path?
    - Does it still give the correct output?
  - If the dry run doesn't give the right answer, there is something wrong that needs fixing.
  - Recording the path through the algorithm will help show where the error occurs.



- *Abstraction*: This is the sentence that will work for all monsters.
  - Draw a \_\_\_\_\_head.
  - Draw a \_\_\_\_\_ eyes.
  - Draw a \_\_\_\_\_ears.
  - Draw a \_\_\_\_\_nose.
  - Draw a \_\_\_\_\_mouth.
- *Algorithm*: These are the instructions using the abstraction adding in one variable.
  - Draw a Happy head.
  - Draw a Vegitas eyes.
  - Draw a Spritem ears.
  - Draw a Wackus nose.
  - Draw a Wackus mouth.



## *What has been described?*

- Algorithms and their automation.
- Flowcharts as a convenient way of representing algorithms.
- Various symbols that appear in a typical flowchart.
- Flowchart control structures viz. sequential, conditional and loop structures.

### *Credits*

- [www.rff.com](http://www.rff.com)
- <http://cdn.robotc.net>
- [ritterit-programming.wikispaces.com](http://ritterit-programming.wikispaces.com)
- [http://www.breezetree.com/flowcharting-software/help/split\\_connector.htm](http://www.breezetree.com/flowcharting-software/help/split_connector.htm)
- <http://www.ftms.edu.my>
- [en.wikipedia.org/wiki/Flowchart](http://en.wikipedia.org/wiki/Flowchart)
- <http://www.cimt.plymouth.ac.uk/projects/mepres/book8/>
- [www2.cs.uregina.ca/~hilder/cs115](http://www2.cs.uregina.ca/~hilder/cs115)
- [www3.ntu.edu.sg](http://www3.ntu.edu.sg)
- [www.multiwingspan.co.uk](http://www.multiwingspan.co.uk)
- [google images](http://www.google.com/images)