**CSC 212**

**GOOD PROGRAMMING PRINCIPLES**

**1. Keep It Simple, Stupid (KISS)**

It sounds a little harsh, but it's a coding principle to live by. What does this mean?

It means you should be writing code as simple as possible. Don't get caught up in trying to be overly clever or showing off with a paragraph of advanced code. If you can write a script in one line, write it in one line.

Here's a simple function:

function addNumbers(num1,num2){

return num1 + num2;

}

Pretty simple. It's easy to read and you know exactly what is going on.

Use clear variable names. Take advantage of coding libraries to use existing tools. Make it easy to come back after six months and get right back to work. Keeping it simple will save you the headache.

**2. Write DRY Code**

The Don't Repeat Yourself (DRY) principle means, plainly, not repeating code. It's a common coding mistake. When writing code, avoid duplication of data or logic. If you've ever copied and pasted code within your program, it's not DRY code.

Take a look at this script:

function addNumberSequence(number){

number = number + 1;

number = number + 2;

number = number + 3;

number = number + 4;

number = number + 5;

return number;

}

Instead of duplicating lines, try to find an algorithm that uses iteration. For loops, and while loops are ways to control code that needs to run multiple times.

DRY code is easy to maintain. It's easier to debug one loop that handles 50 repetitions than 50 blocks of code that handle one repetition.

**3. Open/Closed**

This principle means you should aim to make your code open to extension but closed to modification. This is an important principle when releasing a library or framework that others will use.

For example, suppose you're maintaining a GUI framework. You could release for coders to directly modify and integrate your released code. But what happens when you release a major update four months later?

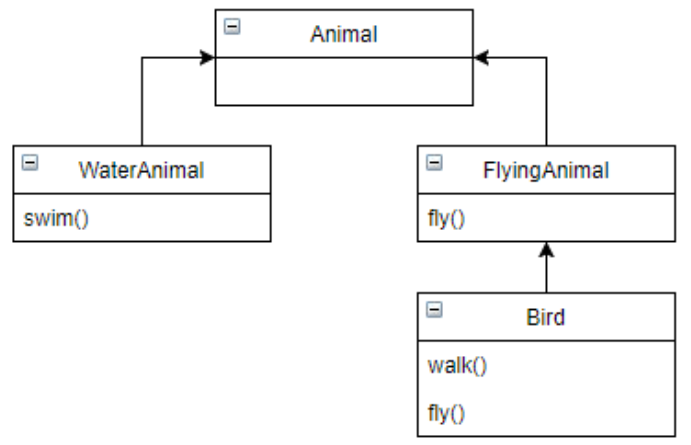
Their code will break. This will make engineers unhappy. They won't want to use your library for much longer, no matter how helpful it may be.

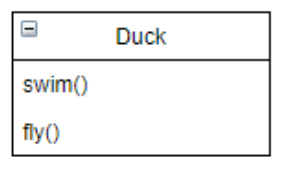
Instead, release code that prevents direct modification and encourages extension. This separates core behavior from modified behavior. The code is more stable and easier to maintain.

**4. Composition Over Inheritance**

If you write code using [object-oriented programming](https://www.makeuseof.com/tag/python-object-oriented-programming/) you're going to find this useful. The **composition over inheritance**principle states: objects with complex behaviors should contain instances of objects with individual behaviors. They should not inherit a class and add new behaviors.

Relying on inheritance causes two major issues. First, the inheritance hierarchy can get messy in a hurry. You also have less flexibility for defining special-case behaviors. Let's say you want to implement behaviors to share:





Composition programming is a lot cleaner to write, easier to maintain and allows flexibility defining behaviors. Each individual behavior is its own class. You can create complex behaviors by combining individual behaviors.

**5. Single Responsibility**

The single responsibility principle says that every class or module in a program should only provide one specific functionality. As Robert C. Martin puts it, "A class should have only one reason to change."

[Classes and modules](https://www.makeuseof.com/tag/python-instance-static-class-methods/) often start off this way. Be careful not to add too many responsibilities as classes get more complicated. Refactor and break them up into smaller classes and modules.

The consequence of overloading classes is twofold. First, it complicates debugging when you're trying to isolate a certain module for troubleshooting. Second, it becomes more difficult to create additional functionality for a specific module.

**6. Separation of Concerns**

The **separation of concerns principle** is an abstract version of the single responsibility principle. This idea states that a program should be designed with different containers, and these containers should not have access to each other.

A well-known example of this is the model-view-controller (MVC) design. MVC separates a program into three distinct areas: the data (model), the logic (controller), and what the page displays (view). Variations of MVC are common in today's most popular web frameworks.

For example, the code that handles the database doesn't need to know how to render the data in the browser. The rendering code takes input from the user, but the logic code handles the processing. Each piece of code is completely independent.

The result is code that is easy to debug. If you ever need to rewrite the rendering code, you can do so without worrying about how the data gets saved or the logic gets processed.

**7. You Aren't Going to Need It (YAGNI)**

Thisprinciple means you should never code for functionality on the chance that you may need in the future. Don't try and solve a problem that doesn't exist.

In an effort to write DRY code, programmers can violate this principle. Often inexperienced programmers try to write the most abstract and generic code they can. Too much abstraction causes bloated code that is impossible to maintain.

Only apply the DRY principle only when you need to. If you notice chunks of code written over and over, then abstract them. Don't think too far out at the expense of your current code batch.

**8. Document Your Code**

Any senior developer will stress the importance of documenting your code with proper comments. All languages offer them and you should make it a habit to write them. Leave comments to explain objects, enhance variable definitions, and make functions easier to understand.

Here's a JavaScript function with comments guiding you through the code:

//This function will add 5 to the input if odd, or return the number if even

function evenOrOdd(number){

//Determine if the number is even

if(number % 2 == 0){

return number;

}

//If the number is odd, this will add 5 and return

else {

return number + 5;

}

}

Leaving comments is a little more work while you're coding, and you understand your code pretty well right?

Leave comments anyway!

Try writing a program, leaving it alone for six months, and come back to modify it. You'll be glad you documented your program instead of having to pour over every function to remember how it works. Work on a coding team? Don't frustrate your fellow developers by forcing them to decipher your syntax.

**9. Refactor**

It's hard to accept, but your code isn't going to be perfect the first time. Refactoring code means reviewing your code and looking for ways to optimize it. Make it more efficient while keeping the results exactly the same.

Codebases are constantly evolving. It's completely normal to revisit, rewrite, or even redesign entire chunks of code. It doesn't mean you didn't succeed the first time you wrote your program. You're going to get more familiar with a project over time. Use that knowledge to adjust your existing code to be DRY, or following the KISS principle.

**10. Clean Code At All Costs**

Leave your ego at the door and forget about writing clever code. The kind of code that looks more like a riddle than a solution. You're not coding to impress strangers.

Don't try to pack a ton of logic into one line. Leave clear instructions in comments and documentation. If your [code is easy to read](https://www.makeuseof.com/how-to-keep-your-code-clean-with-object-encapsulation/) it will be easy to maintain.

Good programmers and readable code go hand-in-hand. Leave comments when necessary. Adhere to style guides, whether dictated by a language or your company.

**ALGORITHM**

* An algorithm is the list of instructions and rules that a computer needs to do to complete a task.
* Algorithms are in everything we do. In fact, the joke runs that developers use the word ‘algorithm’ when they don’t want to explain what they’ve done.
* In essence, algorithms are simply a series of instructions that are followed, step by step, to do something useful or solve a problem. You could consider a cake recipe an algorithm for making a cake, for example.
* In computing, algorithms provide computers with a successive guide to completing actions. They’re comprised of a precise list of instructions that outline exactly how to complete a task.
* An algorithm is a set of step-by-step procedures, or a set of rules to follow, for completing a specific task or solving a particular problem. ... The recipe for baking a cake, the method we use to solve a long division problem, and the process of doing laundry are all examples of an algorithm.
* An algorithm is a set of well-defined instructions in sequence to solve a problem.

**Qualities of Good Algorithms**

* Input and output should be defined precisely.
* Each step in the algorithm should be clear and unambiguous.
* Algorithms should be most effective among many different ways to solve a problem.
* An algorithm shouldn't include computer code. Instead, the algorithm should be written in such a way that it can be used in different programming languages.

**EXAMPLES OF ALGORITHM**

**Algorithm 1: Add two numbers entered by the user**

Step 1: Start

Step 2: Declare variables num1, num2 and sum.

Step 3: Read values num1 and num2.

Step 4: Add num1 and num2 and assign the result to sum.

sum←num1+num2

Step 5: Display sum

Step 6: Stop

**Algorithm 2: Find the largest number among three numbers**

Step 1: Start

Step 2: Declare variables a,b and c.

Step 3: Read variables a,b and c.

Step 4: If a > b

If a > c

Display a is the largest number.

Else

Display c is the largest number.

Else

If b > c

Display b is the largest number.

Else

Display c is the greatest number.

Step 5: Stop

**Algorithm 3: Find Root of the quadratic equation ax2 + bx + c = 0**

Step 1: Start

Step 2: Declare variables a, b, c, D, x1, x2, rp and ip;

Step 3: Calculate discriminant

D ← b2-4ac

Step 4: If D ≥ 0

r1 ← (-b+√D)/2a

r2 ← (-b-√D)/2a

Display r1 and r2 as roots.

Else

Calculate real part and imaginary part

rp ← -b/2a

ip ← √(-D)/2a

Display rp+j(ip) and rp-j(ip) as roots

Step 5: Stop

**Algorithm 4: Find the factorial of a number**

Step 1: Start

Step 2: Declare variables n, factorial and i.

Step 3: Initialize variables

factorial ← 1

i ← 1

Step 4: Read value of n

Step 5: Repeat the steps until i = n

5.1: factorial ← factorial\*i

5.2: i ← i+1

Step 6: Display factorial

Step 7: Stop

**Algorithm 5: Check whether a number is prime or not**

Step 1: Start

Step 2: Declare variables n, i, flag.

Step 3: Initialize variables

flag ← 1

i ← 2

Step 4: Read n from the user.

Step 5: Repeat the steps until i=(n/2)

5.1 If remainder of n÷i equals 0

flag ← 0

Go to step 6

5.2 i ← i+1

Step 6: If flag = 0

Display n is not prime

else

Display n is prime

Step 7: Stop

**Algorithm 6: Find the Fibonacci series till the term less than 1000**

Step 1: Start

Step 2: Declare variables first\_term,second\_term and temp.

Step 3: Initialize variables first\_term ← 0 second\_term ← 1

Step 4: Display first\_term and second\_term

Step 5: Repeat the steps until second\_term ≤ 1000

5.1: temp ← second\_term

5.2: second\_term ← second\_term + first\_term

5.3: first\_term ← temp

5.4: Display second\_term

Step 6: Stop

**PSEUDOCODE**

* Pseudocode is an informal way of programming description that does not require any strict programming language syntax or underlying technology considerations. It is used for creating an outline or a rough draft of a program. Pseudocode summarizes a program’s flow, but excludes underlying details. System designers write pseudocode to ensure that programmers understand a software project's requirements and align code accordingly.
* Pseudocode is lines of instructions written in a language close to English but with common programming terms used where possible (selection and iteration etc).
* Pseudocode is an artificial and informal language that helps programmers develop algorithms. Pseudocode is a "text-based" detail (algorithmic) design tool.
* Pseudocode is an informal high-level description of the operating principle of a computer program or other algorithm.
* Pseudocode is a simple way of writing programming code in English. Pseudocode is not actual programming language.
* Pseudocode is a method of planning which enables the programmer to plan without worrying about synta
* Description: Pseudocode is not an actual programming language. So it cannot be compiled into an executable program. It uses short terms or simple English language syntaxes to write code for programs before it is actually converted into a specific programming language. This is done to identify top level flow errors, and understand the programming data flows that the final program is going to use. This definitely helps save time during actual programming as conceptual errors have been already corrected. Firstly, program description and functionality is gathered and then pseudocode is used to create statements to achieve the required results for a program. Detailed pseudocode is inspected and verified by the designer’s team or programmers to match design specifications. Catching errors or wrong program flow at the pseudocode stage is beneficial for development as it is less costly than catching them later. Once the pseudocode is accepted by the team, it is rewritten using the vocabulary and syntax of a programming language. The purpose of using pseudocode is an efficient key principle of an algorithm. It is used in planning an algorithm with sketching out the structure of the program before the actual coding takes place.

**Advantages of pseudocode**

* Pseudocode is understood by the programmers of all types.
* it enables the programmer to concentrate only on the algorithm part of the code development.
* It cannot be compiled into an executable program. Example, Java code : if (i < 10) { i++; } pseudocode :if i is less than 10, increment i by 1.

**EXAMPLES OF PSEUDOCODE**

**Task 1: Write a program that asks the user for a temperature in Fahrenheit and prints out the same temperature in Celsius.**

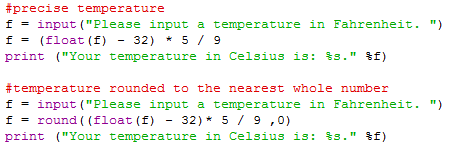
Example Pseudocode:

x = Get user input

x = Convert X to Celsius

Output message displaying Celsius temperature

Example code:



The first part of the program rounds gets the precise temperature, contrary to the second program which uses the round function to round the result to the nearest whole number. The round function works as shown below:

round(x,y)

"x" is what you're rounding and "y" is how many decimal places you want to round up to.

**Task 2: Write a program that converts from Fahrenheit to Celsius or from Celsius to Fahrenheit, depending on the user's choice.**

Pseudocode:

x = input "Press 1 to convert from Fahrenheit to Celsius or Press 2 to convert from Celsius to Fahrenheit. "

y = input ask what number?

if 1 is pressed

do f to c conversion

print output

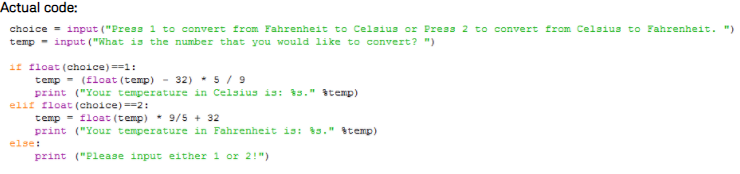
elif 2 is pressed

do c to f conversion

print output

else

print "Please enter either 1 or 2"



**Task 3: Write a program that lets the user type words and when they press 'x', it prints how many words the user inputted then quits program.**

Pseudocode:

while true

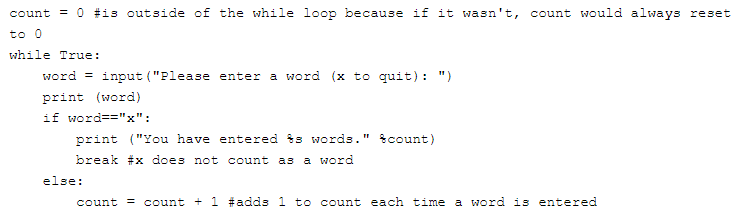
ask user to enter a word

print word

if word = x

print number of words entered

break



**Task 5: Write a program that asks for a user to enter a number, and then print all those numbers after 10 numbers are entered.**

Pseudocode:

numbercount is 0

list

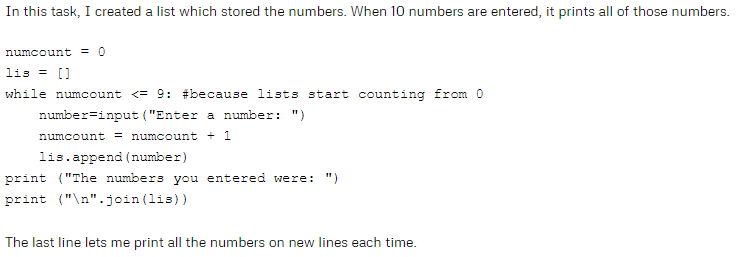
while the number count <= 9

ask for input

add one to number count

add number to list

print numbers on a separate line each time

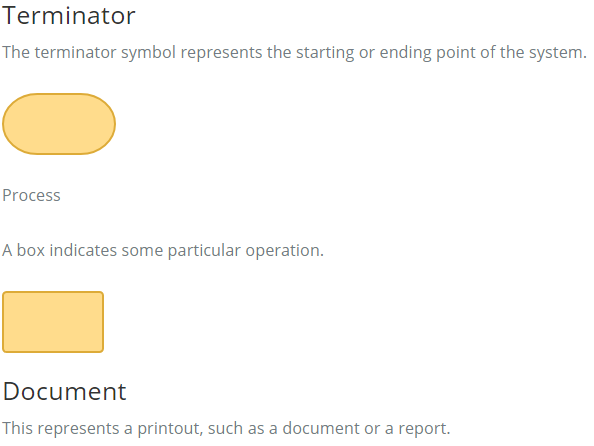


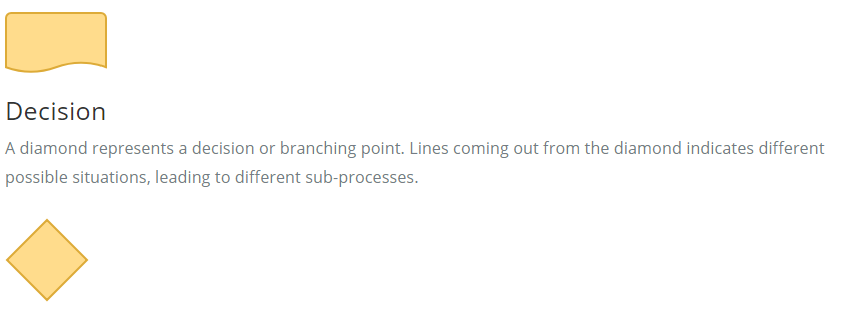
**FLOWCHART**

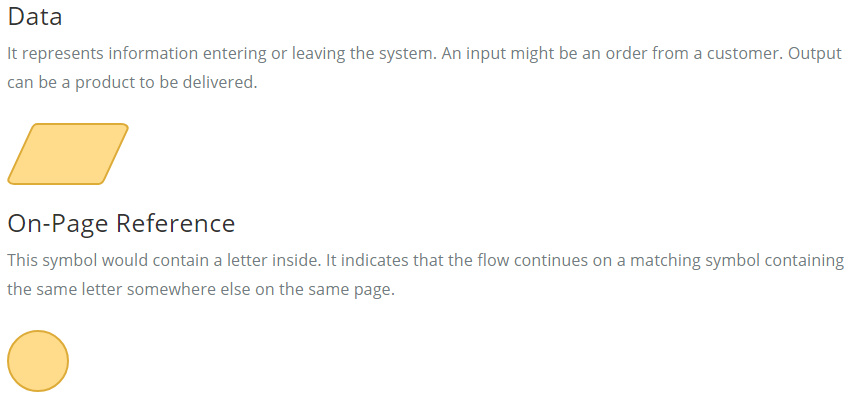
* A flowchart is a type of diagram that represents a workflow or process. A flowchart can also be defined as a diagrammatic representation of an algorithm, a step-by-step approach to solving a task. The flowchart shows the steps as boxes of various kinds, and their order by connecting the boxes with arrows.
* A flowchart is simply a graphical representation of steps. It shows steps in sequential order and is widely used in presenting the flow of algorithms, workflow or processes. Typically, a flowchart shows the steps as boxes of various kinds, and their order by connecting them with arrows.
* A flowchart is a graphical representations of steps. It was originated from computer science as a tool for representing algorithms and programming logic but had extended to use in all other kinds of processes. Nowadays, flowcharts play an extremely important role in displaying information and assisting reasoning. They help us visualize complex processes, or make explicit the structure of problems and tasks. A flowchart can also be used to define a process or project to be implemented.

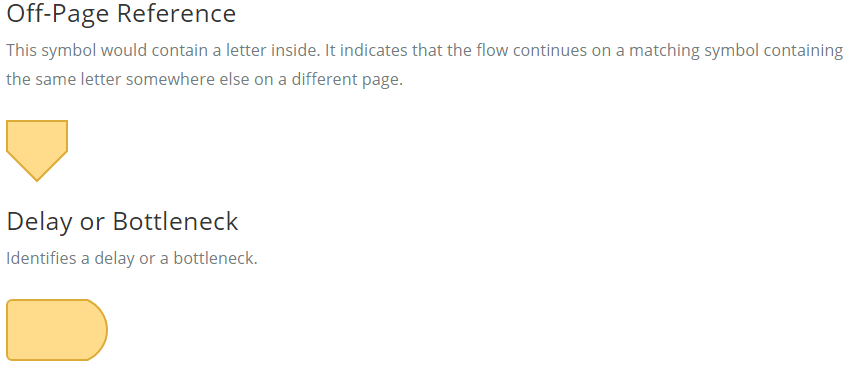
**Flowchart Symbols**

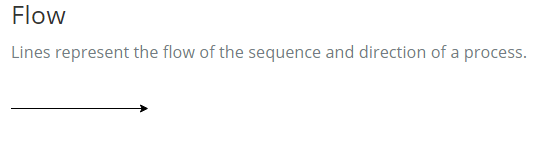
* Different flowchart shapes have different conventional meanings. The meanings of some of the more common shapes are as follows:











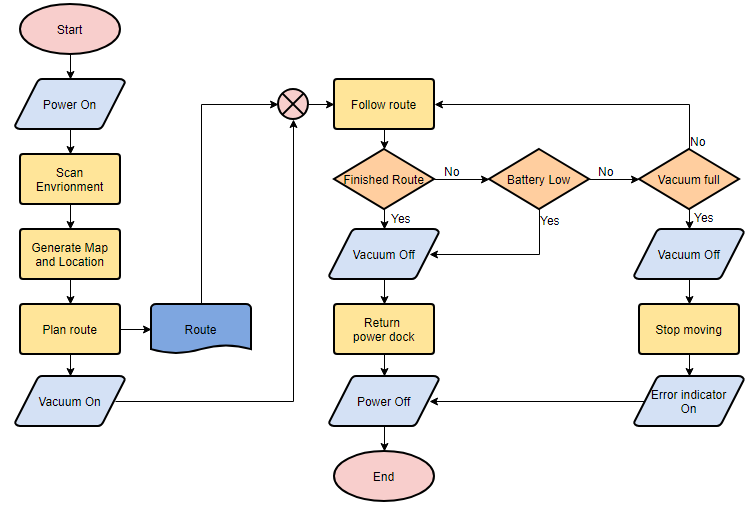
**When to Draw Flowchart?**

Using a flowchart has a variety of benefits:

* It helps to clarify complex processes.
* It identifies steps that do not add value to the internal or external customer, including delays; needless storage and transportation; unnecessary work, duplication, and added expense; breakdowns in communication.
* It helps team members gain a shared understanding of the process and use this knowledge to collect data, identify problems, focus discussions, and identify resources.
* It serves as a basis for designing new processes.

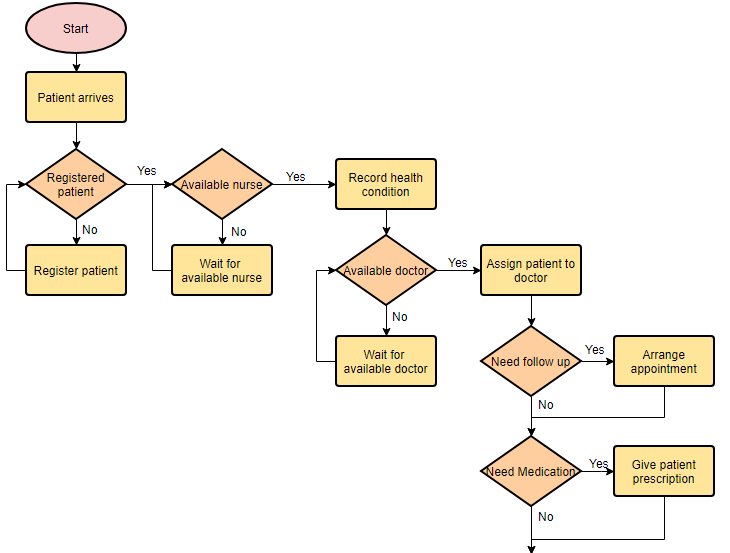
**Flowchart examples**

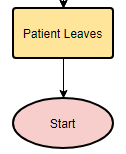
Here are several flowchart examples. See how you can apply a flowchart practically.



**Flowchart Example – Medical Service**

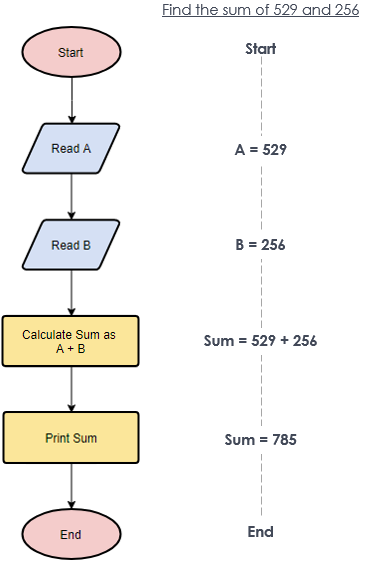
This is a hospital flowchart example that shows how clinical cases shall be processed. This flowchart uses decision shapes intensively in representing alternative flows.





**Flowchart Example – Simple Algorithms**

A flowchart can also be used in visualizing algorithms, regardless of its complexity. Here is an example that shows how flowchart can be used in showing a simple summation process.



**Flowchart Example – Calculate Profit and Loss**

The flowchart example below shows how profit and loss can be calculated.

Find the profit / loss when income = 1000 and loss = 800

