Programming in C

By

Dr Ramkumar Krishnamoorthy

kramdharma@gmail.com

Introduction to Computer Based Problem Solving

Unit I

Contents

Program Design

Implementation Issues

Prepare Flow Charts

Develop Algorithms

Top wise Design

Stepwise Refinement

Programming Environment

Machine Language

Assembly Language

High Level Language

Assemblers

Compilers

Interpreters

Unit I – Chapter I

> “Computer Science is a science of abstraction -creating the right model for a problem and devising the appropriate mechanizable techniques to solve it

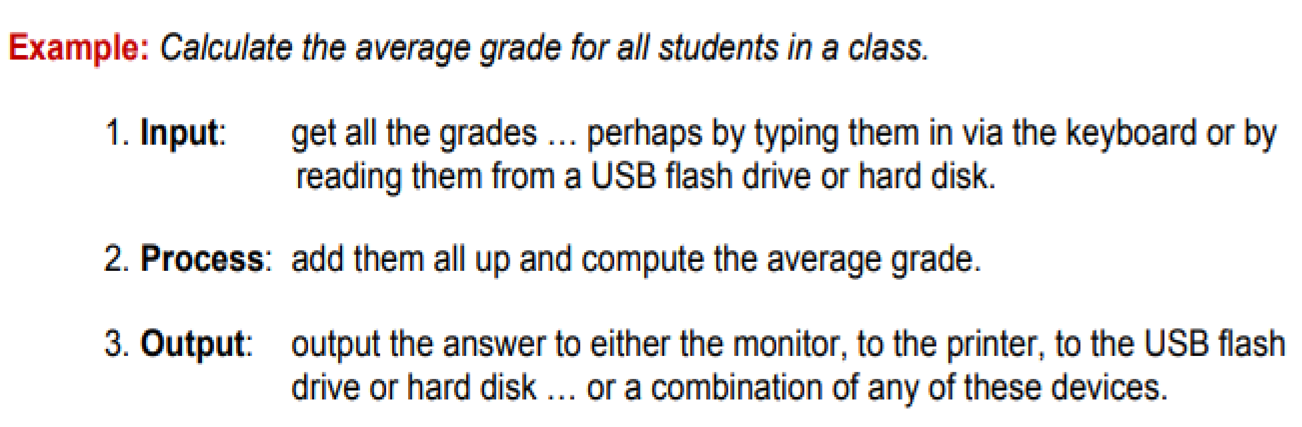
Problem Analysis

- Problem analysis is the process of defining a problem and decomposing overall system into smaller parts to identify possible inputs, processes and outputs associated with the problem.
- It is the sequential process of analyzing information related to a given situation and generating appropriate response options.

Steps to Solve a Problem

- Understand the Problem
- Formulate a Model
- Develop an Algorithm
- Write the Program
- Test the Program
- Evaluate the Solution

Example Flow



Example: *Calculate the average grade for all students in a class.*

1. **Input**: get all the grades … perhaps by typing them in via the keyboard or by reading them from a USB flash drive or hard disk.

2. **Process**: add them all up and compute the average grade.

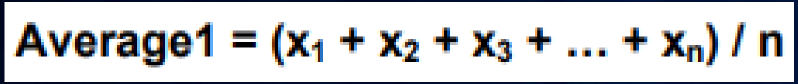
3. **Output**: output the answer to either the monitor, to the printer, to the USB flash drive or hard disk … or a combination of any of these devices.

Step 1: Understand the Problem

- It sounds strange, but the first step to solving any problem is to make sure that you understand the problem that you are trying to solve.
  - What input data/information is available ?
  - What does it represent ?
  - What format is it in ?
  - Is anything missing ?
  - Do I have everything that I need ?
  - What output information am I trying to produce ?
  - What do I want the result to look like … text, a picture, a graph … ?
  - What am I going to have to compute ?

Step 2: Formulate a Model

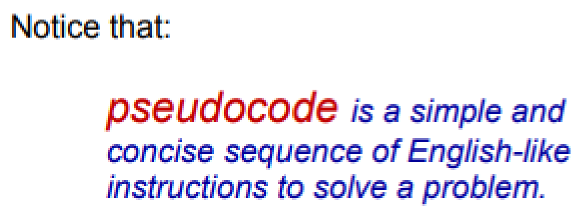
- Many problems break down into smaller problems that require some kind of simple mathematical computations in order to process the data.



$$\text{Average1} = (x_1 + x_2 + x_3 + \ldots + x_n) / n$$

Step 3: Develop an Algorithm

- An algorithm is a precise sequence of instructions for solving a problem

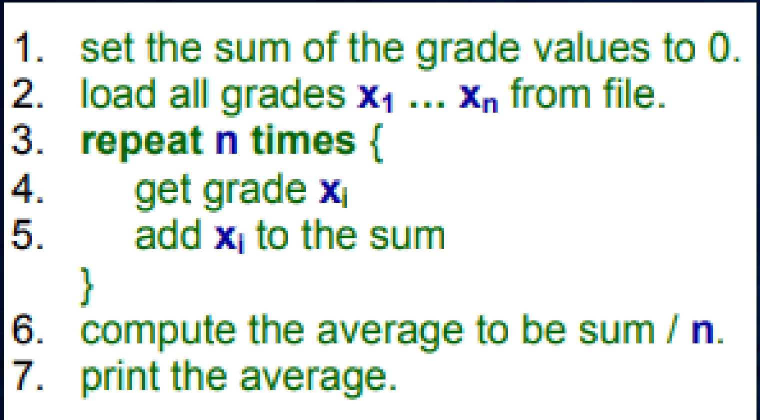


Notice that:

pseudocode is a simple and concise sequence of English-like instructions to solve a problem.

Step 4: Write the Program

- Writing a program is often called "writing code" or “implementing an algorithm”. So the code (or source code) is actually the program itself



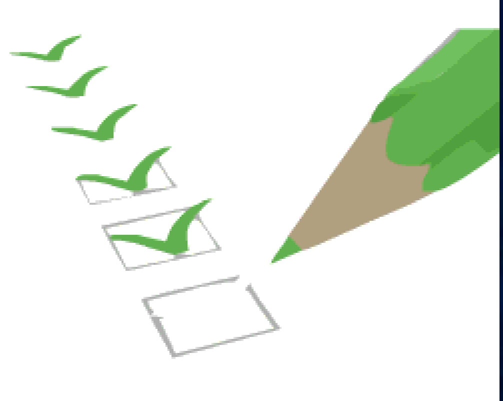
```
set the sum of the grade values to 0.
load all grades $x_1$ … $x_n$ from file.
repeat n times {
    get grade $x_i$
    add $x_i$ to the sum
}
compute the average to be sum / n.
print the average.
```

Step 5: Test the Program

- Once you have a program written that compiles, you need to make sure that it solves the problem that it was intended to solve and that the solutions are correct
- Running a program is the process of telling the computer to evaluate the compiled instructions

Step 6: Evaluate the Solution

- It is also possible that when you examine your results, you realize that you need additional data to fully solve the problem. Or, perhaps you need to adjust the results to solve the problem more efficiently



Program Design

Program Design

- Program design consists of the steps a programmer should do before they start coding the program in a specific language. These steps when properly documented will make the completed program easier for other programmers to maintain in the future. There are three broad areas of activity:
  - Understanding the Program
  - Using Design Tools to Create a Model
  - Develop Test Data

Understanding the Problem

- If you are working on a project as one of many programmers, the system analyst may have created a variety of documentation items that will help you understand what the program is to do. These could include screen layouts, narrative descriptions, documentation showing the processing steps, etc. If you are not on a project and you are creating a simple program you might be given only a simple description of the purpose of the program.
- Understanding the purpose of a program usually involves understanding its:

  - Inputs
  - Processing
  - Outputs

Use Design to Create a Model

- Pseudocode
- Algorithm
- Data

Develop Test Data

- Test data consists of the programmer providing some input values and predicting the outputs. This can be quite easy for a simple program and the test data can be used to check the model to see if it produces the correct results.

Program

Program

- Set of statements/instructions to achieve some specific task is called a program
- Task should be clear and concise
- To write a correct program, a programmer must write each and every instructions in the correct sequence
  - Logic (instruction sequence) of a program can be very complex
- Hence, programs must be planned before they are written to ensure program instruction are:
  - Appropriate for the problem
  - In the correct sequence

Program

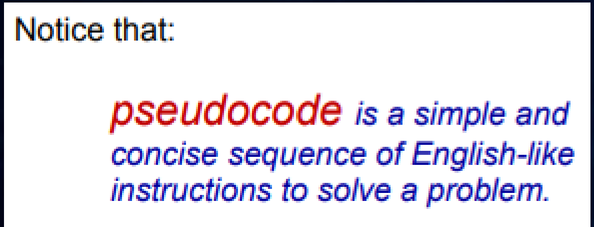
- Program writing is definitely a technique. One has to master it by experience. One should always start with writing small and simple programs and then graduate to complex and complicated programs.
- Whatever he may write, it should be written in a manner which is understood by others. God forbid if he is not around the others should be able to run and modify it. Here the documentation parts becomes very necessary.
- A good program should have

1. Readability
2. Efficiency
3. Reliability
4. Meaningfulness
5. Portability
6. User-friendly

Algorithm Development

Algorithm

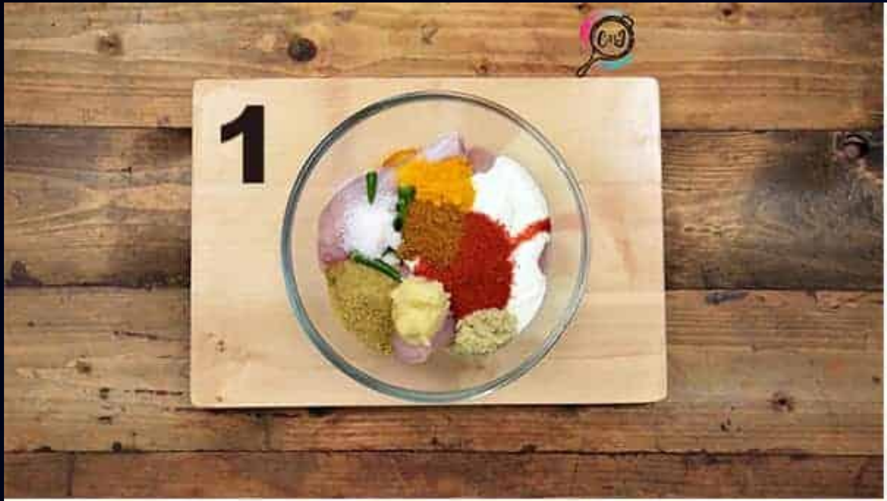
- An algorithm is a precise sequence of instructions for solving a problem

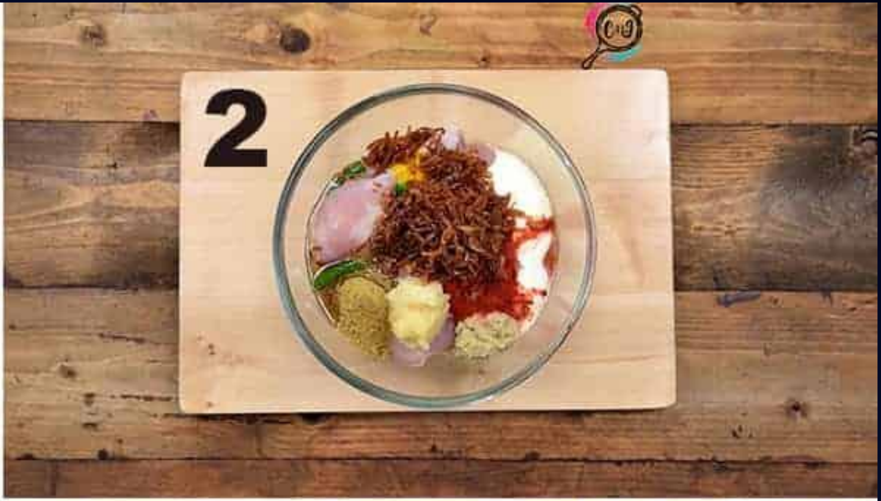


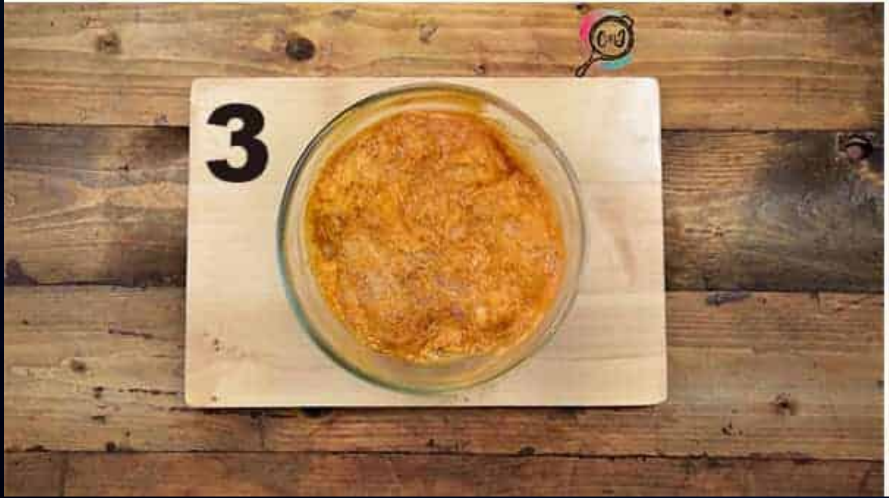
Notice that:

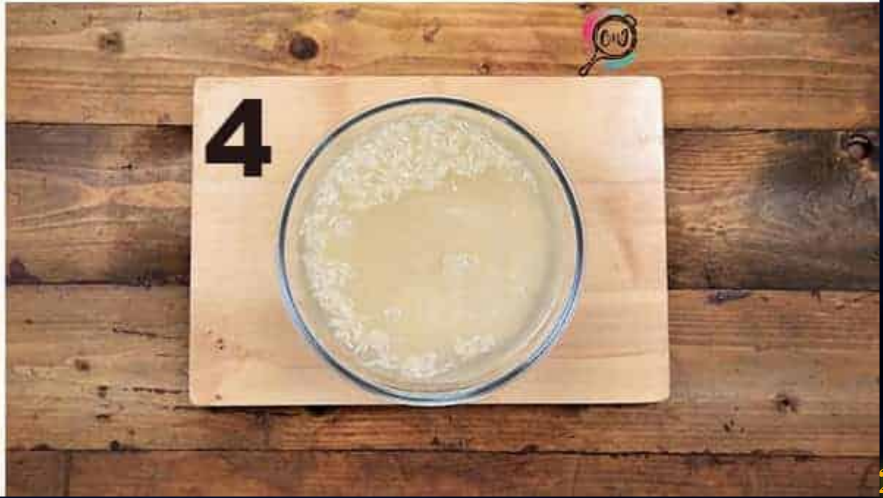
pseudocode is a simple and concise sequence of English-like instructions to solve a problem.

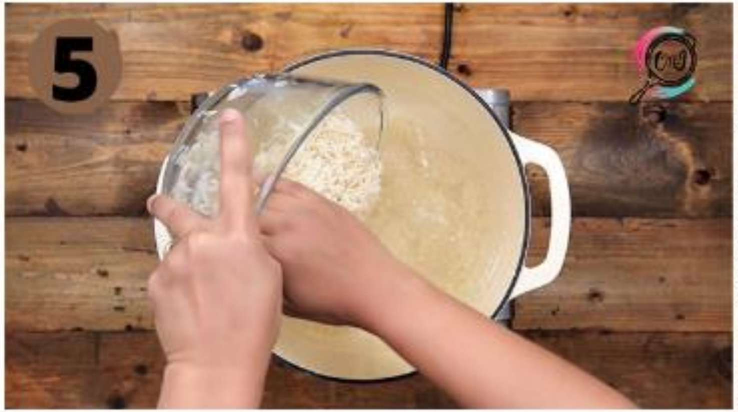
Example Algorithm for Making Chicken Briyani

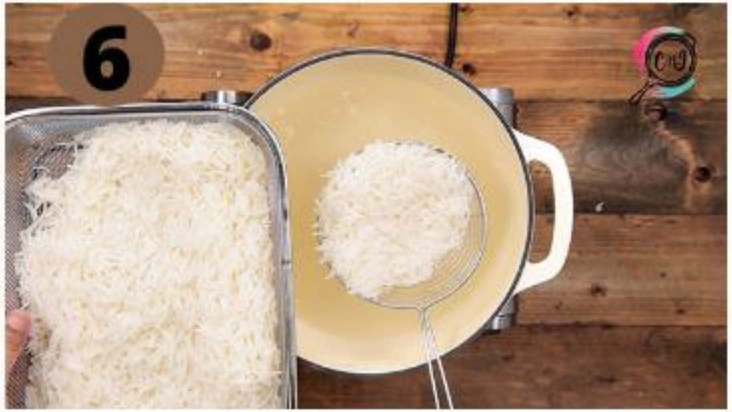

1


2

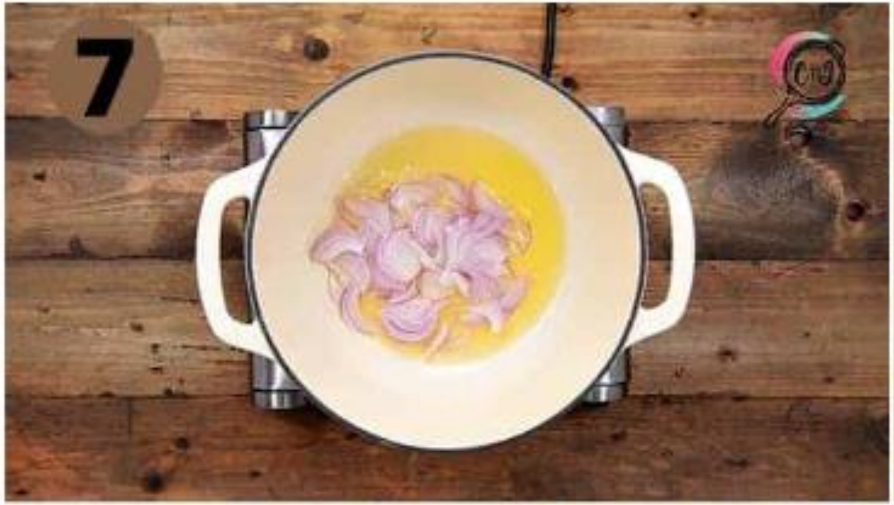

3


4

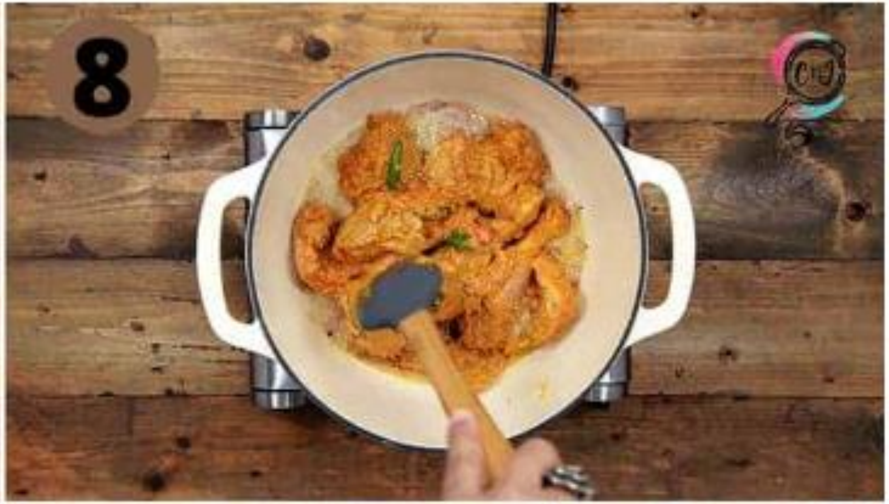

5


6

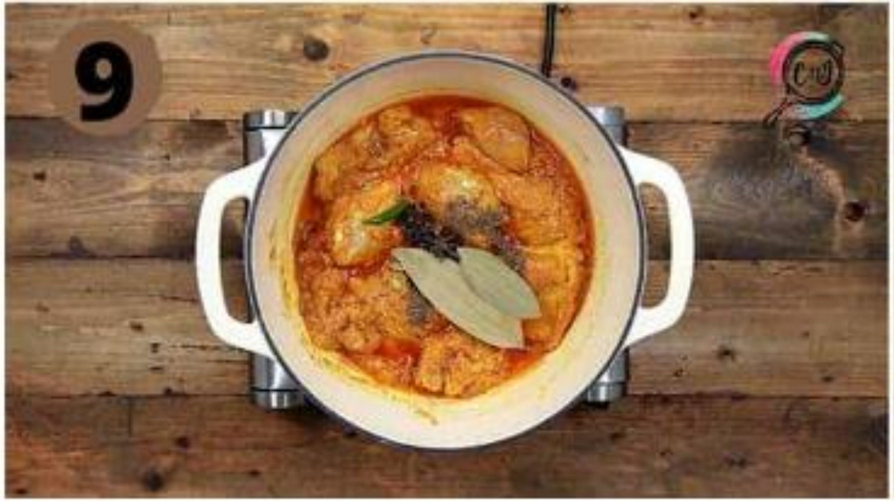
7



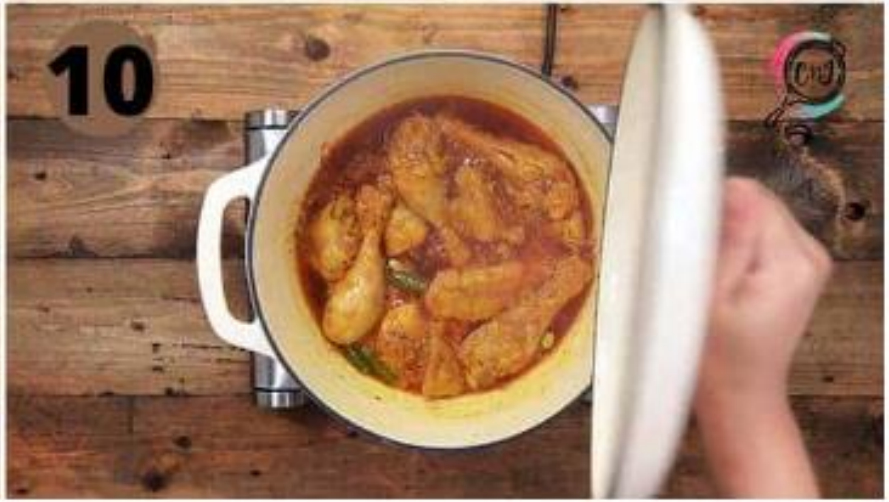
8

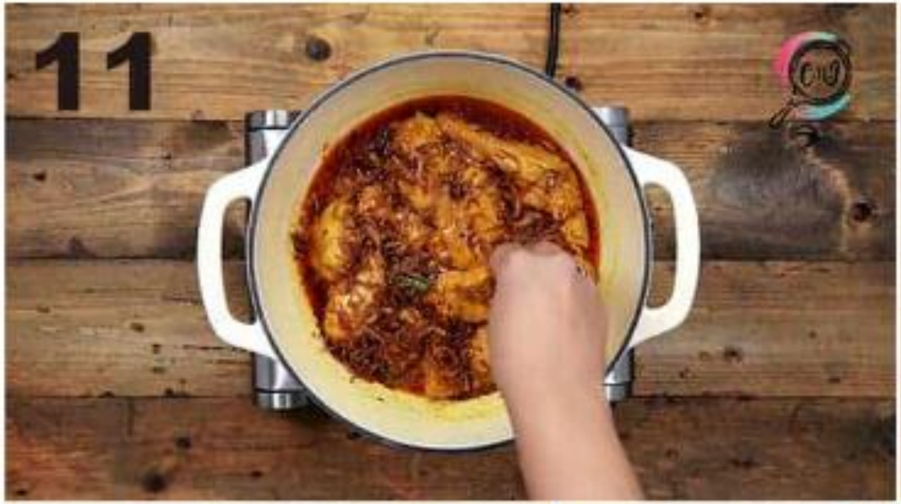


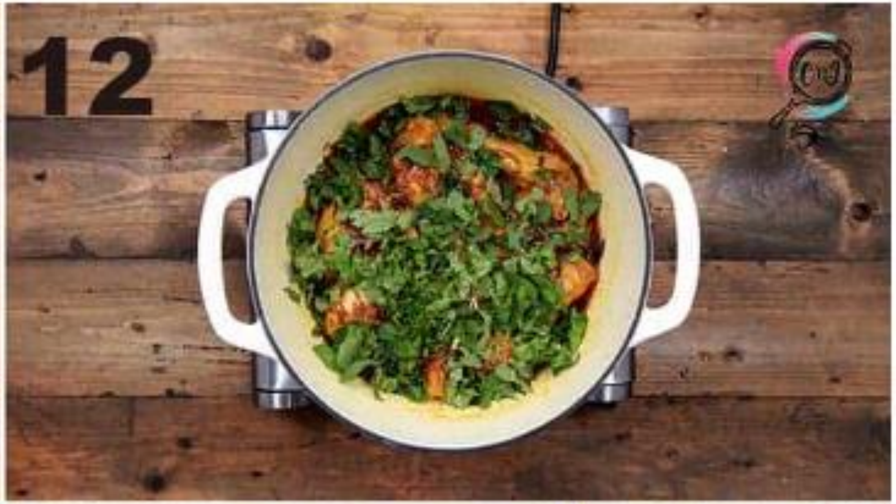
9

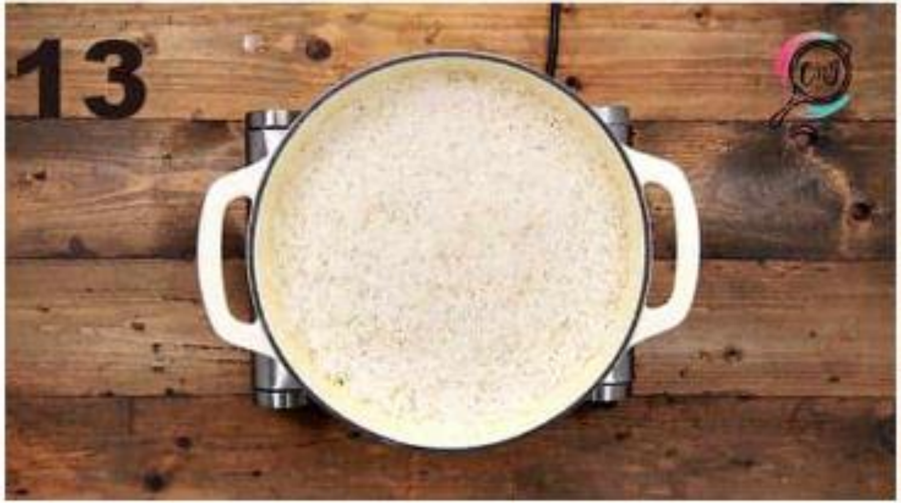


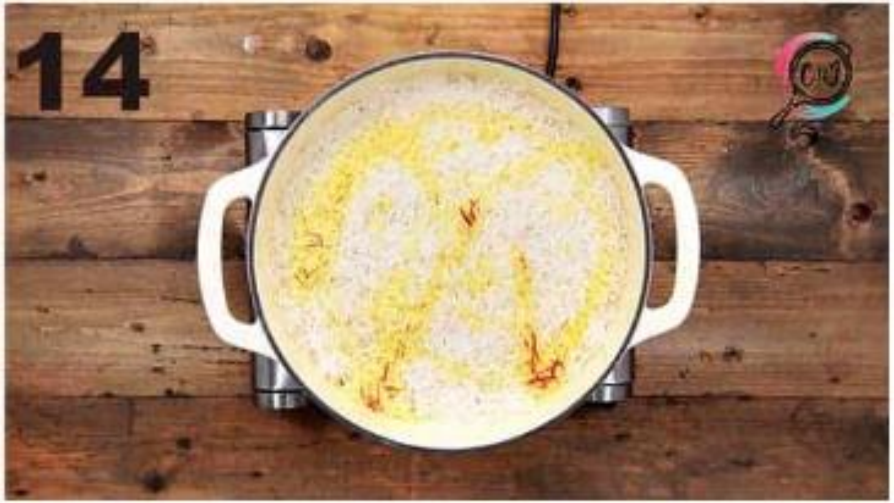
10




11

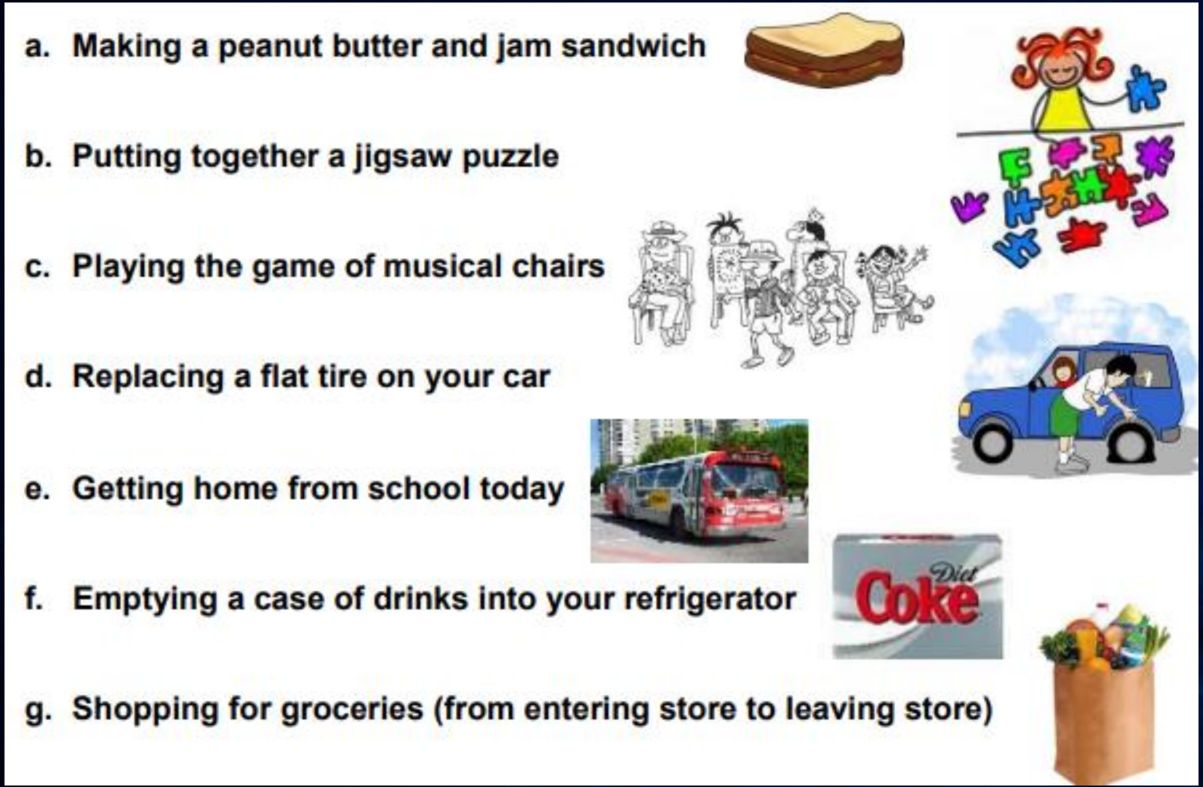

12


13


14

kulfyapp.com

Sample Problems for Practices



a. Making a peanut butter and jam sandwich

b. Putting together a jigsaw puzzle

c. Playing the game of musical chairs

d. Replacing a flat tire on your car

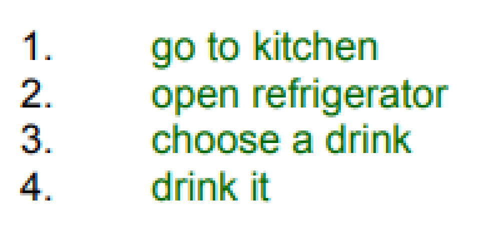
e. Getting home from school today

f. Emptying a case of drinks into your refrigerator

g. Shopping for groceries (from entering store to leaving store)

Sample Algorithms

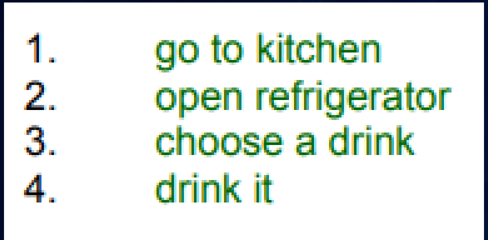
- Support You are thirsty and at Couch (Bed)



1. go to kitchen
2. open refrigerator
3. choose a drink
4. drink it

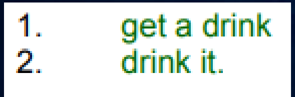
Sample Algorithms

- Support You are thirsty and at Couch (Bed)



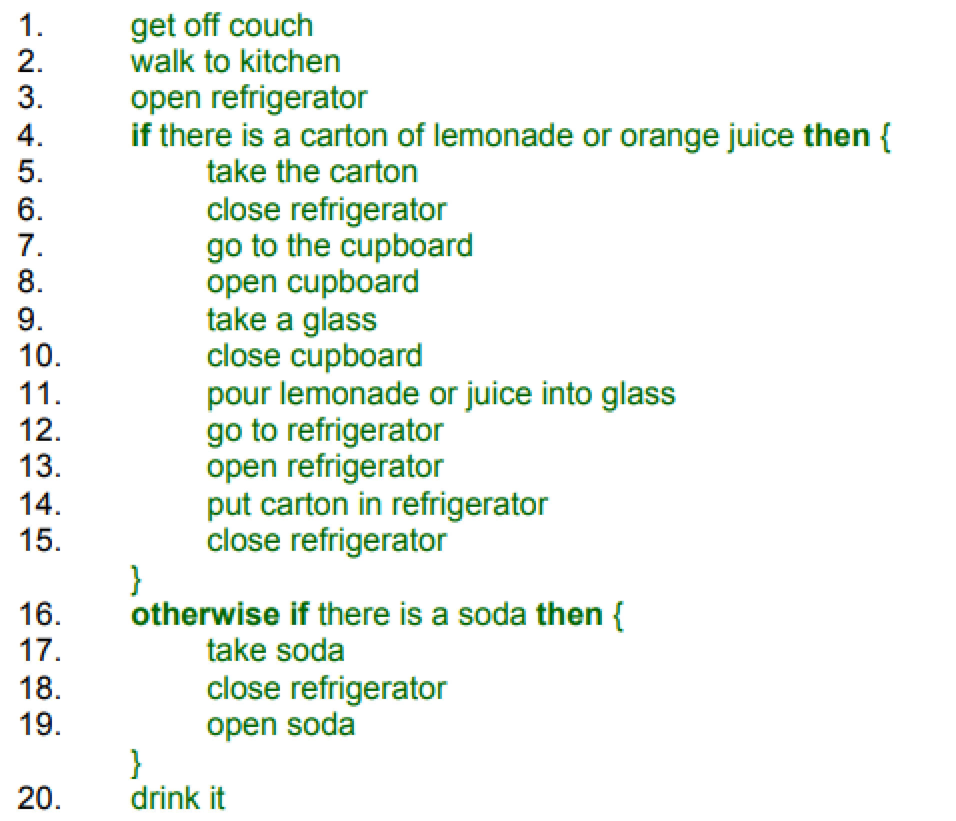
1. go to kitchen
2. open refrigerator
3. choose a drink
4. drink it

- Short (Abstract)



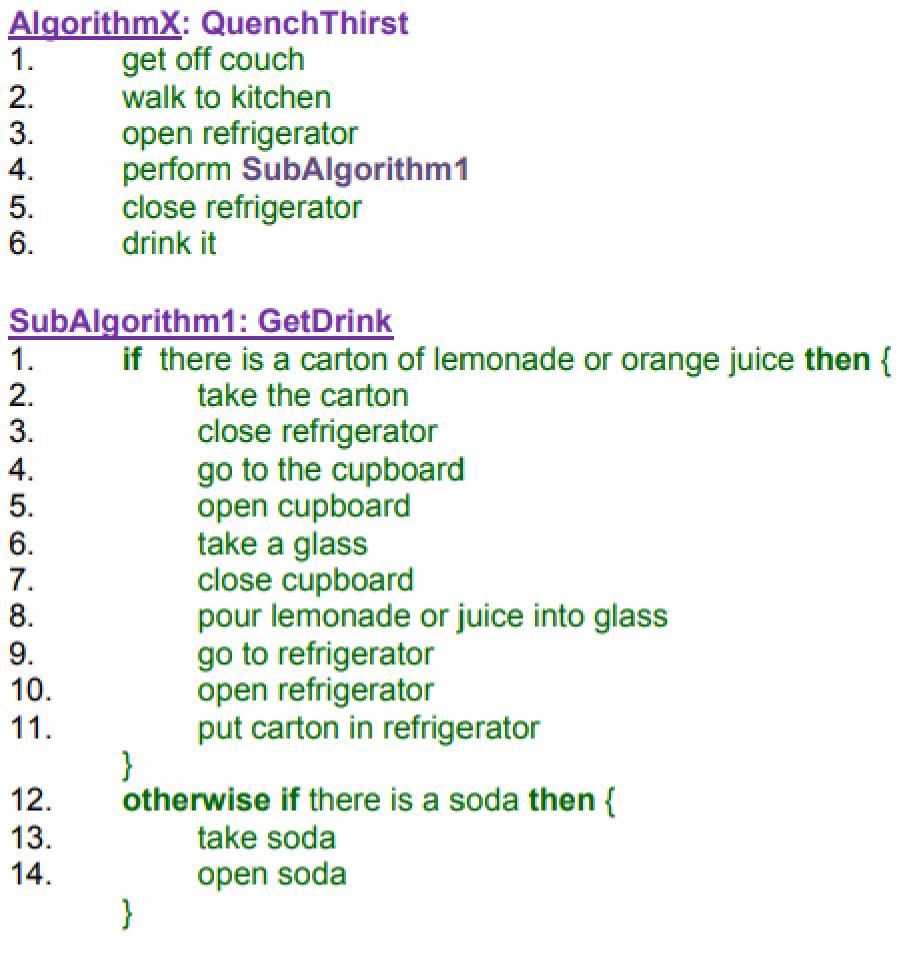
1. get a drink
2. drink it.

Sample Algorithms



```
1.    get off couch
2.    walk to kitchen
3.    open refrigerator
4.    if there is a carton of lemonade or orange juice then {
5.        take the carton
6.        close refrigerator
7.        go to the cupboard
8.        open cupboard
9.        take a glass
10.       close cupboard
11.       pour lemonade or juice into glass
12.       go to refrigerator
13.       open refrigerator
14.       put carton in refrigerator
15.       close refrigerator
      }
16.   otherwise if there is a soda then {
17.       take soda
18.       close refrigerator
19.       open soda
      }
20.   drink it
```

Sample Algorithms

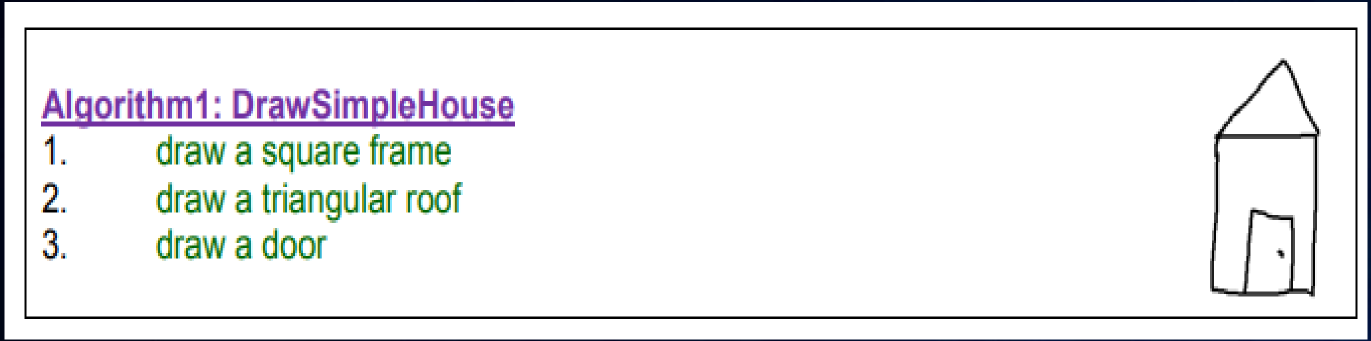


```
AlgorithmX: QuenchThirst
1.      get off couch
2.      walk to kitchen
3.      open refrigerator
4.      perform SubAlgorithm1
5.      close refrigerator
6.      drink it

SubAlgorithm1: GetDrink
1.      if  there is a carton of lemonade or orange juice then {
2.          take the carton
3.          close refrigerator
4.          go to the cupboard
5.          open cupboard
6.          take a glass
7.          close cupboard
8.          pour lemonade or juice into glass
9.          go to refrigerator
10.         open refrigerator
11.         put carton in refrigerator
        }
12.     otherwise if there is a soda then {
13.         take soda
14.         open soda
        }
```

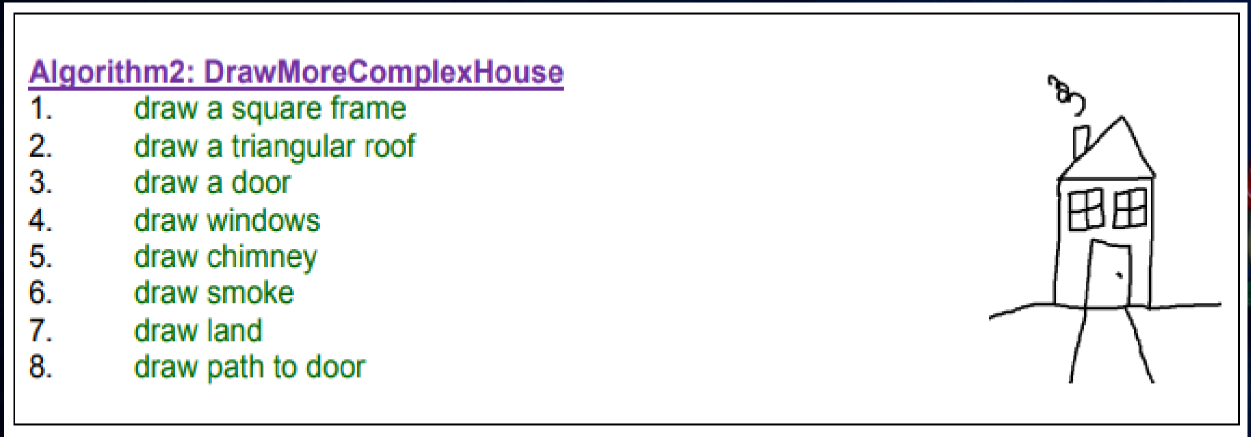
- By having Sub Algorithm

Sample Algorithms



Algorithm1: DrawSimpleHouse

1. draw a square frame
2. draw a triangular roof
3. draw a door



Algorithm2: DrawMoreComplexHouse

1. draw a square frame
2. draw a triangular roof
3. draw a door
4. draw windows
5. draw chimney
6. draw smoke
7. draw land
8. draw path to door

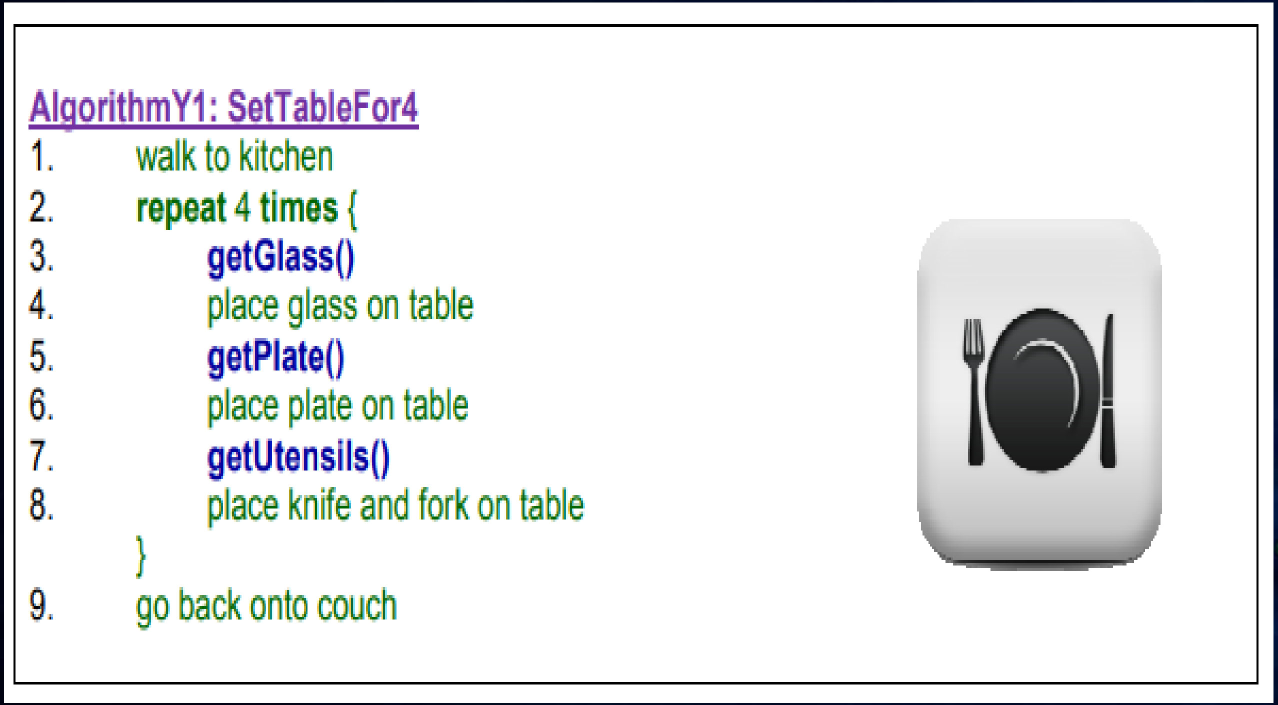
Algorithm Efficiency

- It is used to describe properties of an algorithm relating to how much of various types of resources it consumes

The **runtime complexity** (a.k.a. running time) of an algorithm is the amount of time that it takes to complete once it has begun.

The **space complexity** of an algorithm is the amount of storage space that it requires while running from start to completion

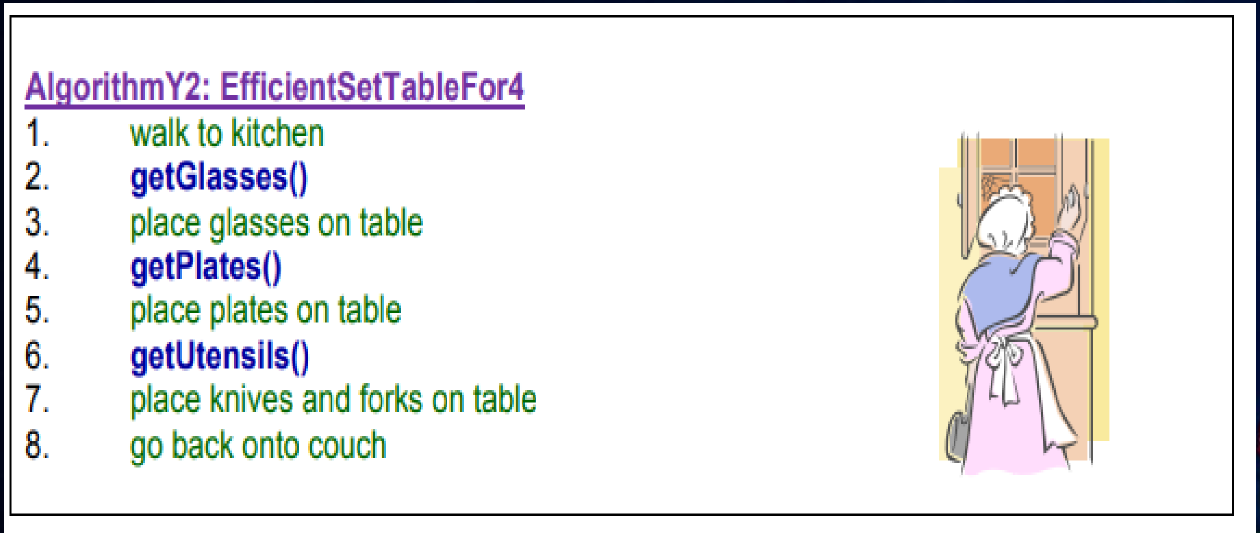
Algorithm Efficiency



AlgorithmY1: SetTableFor4

```
walk to kitchen
repeat 4 times {
    getGlass()
    place glass on table
    getPlate()
    place plate on table
    getUtensils()
    place knife and fork on table
}
go back onto couch
```

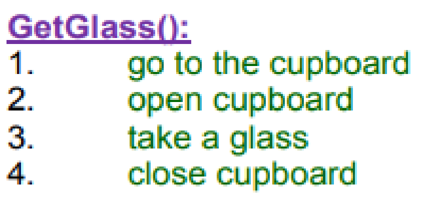
Algorithm Efficiency



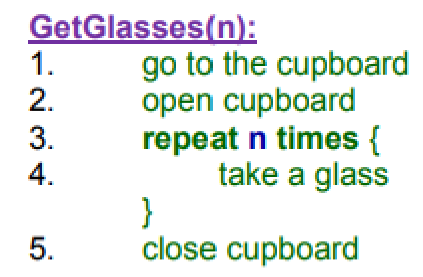
AlgorithmY2: EfficientSetTableFor4

```
1.  walk to kitchen
2.  getGlasses()
3.  place glasses on table
4.  getPlates()
5.  place plates on table
6.  getUtensils()
7.  place knives and forks on table
8.  go back onto couch
```

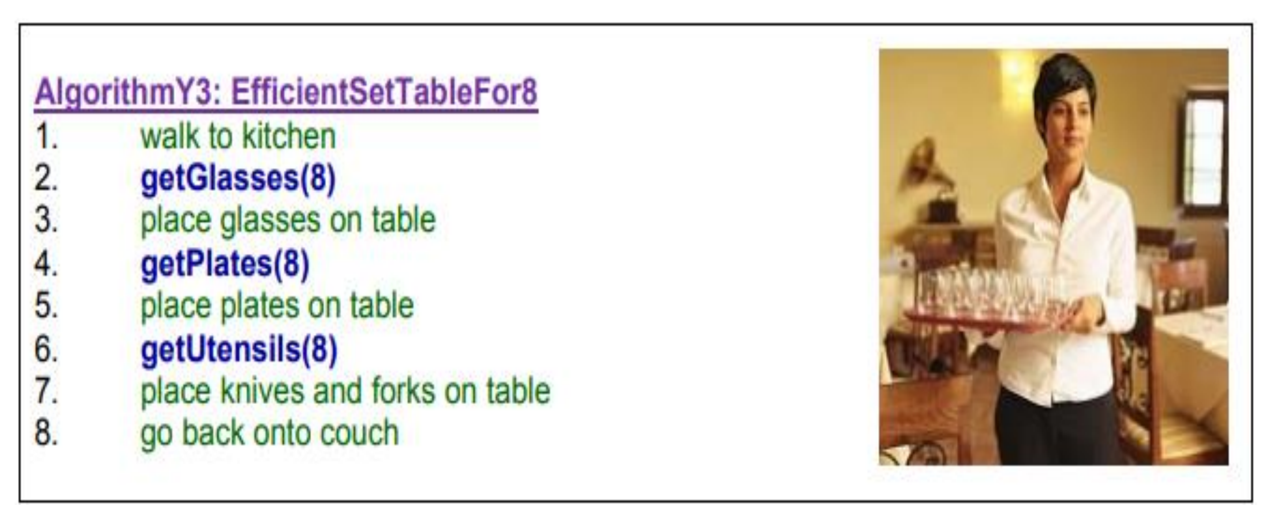
Algorithm Efficiency



```
GetGlass():
1.        go to the cupboard
2.        open cupboard
3.        take a glass
4.        close cupboard
```

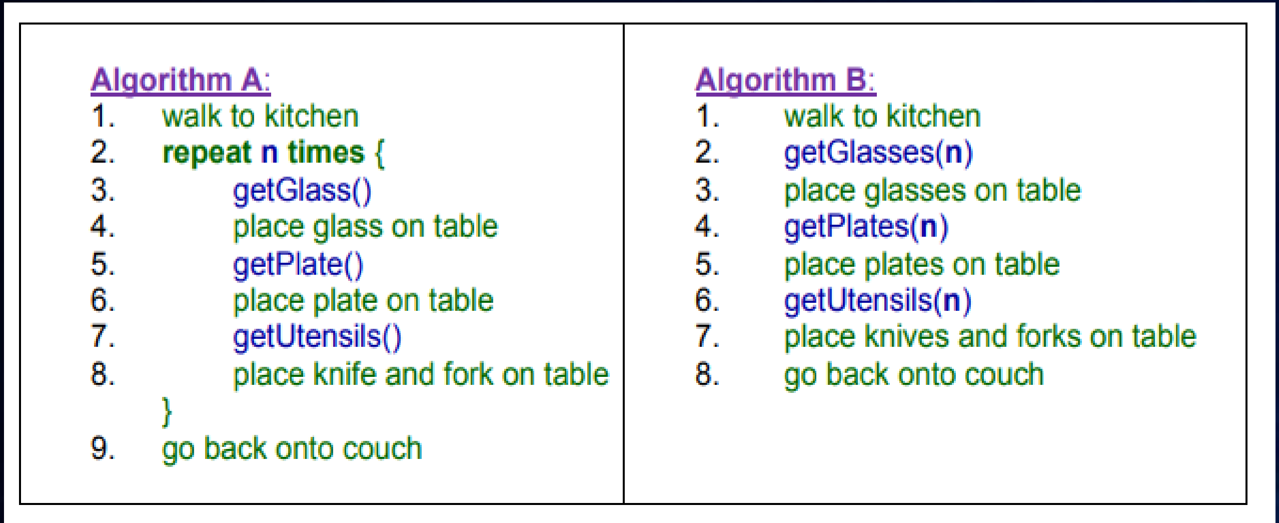


```
GetGlasses(n):
1.        go to the cupboard
2.        open cupboard
3.        repeat n times {
4.            take a glass
          }
5.        close cupboard
```



```
AlgorithmY3: EfficientSetTableFor8
1.        walk to kitchen
2.        getGlasses(8)
3.        place glasses on table
4.        getPlates(8)
5.        place plates on table
6.        getUtensils(8)
7.        place knives and forks on table
8.        go back onto couch
```

Algorithm Efficiency Comparison



Algorithm A:

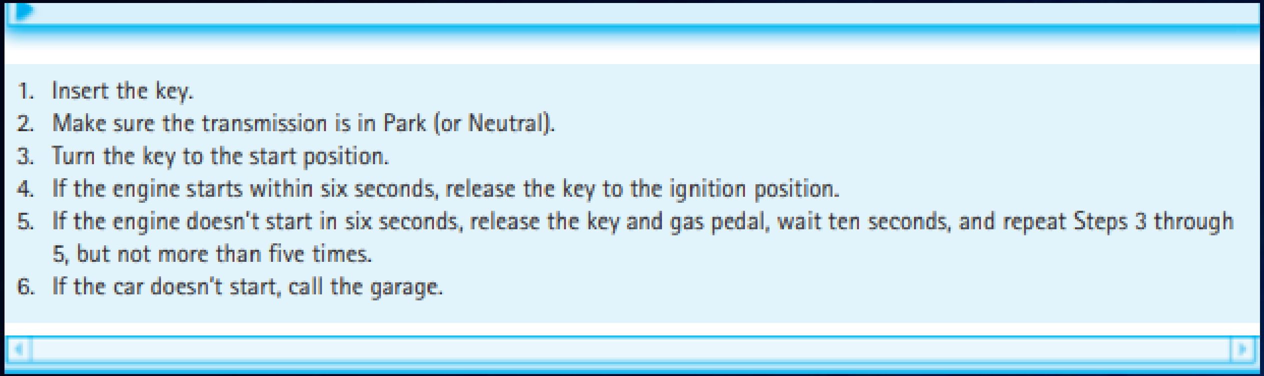
```
1.    walk to kitchen
2.    repeat n times {
3.          getGlass()
4.          place glass on table
5.          getPlate()
6.          place plate on table
7.          getUtensils()
8.          place knife and fork on table
      }
9.    go back onto couch
```

Algorithm B:

```
1.    walk to kitchen
2.    getGlasses(n)
3.    place glasses on table
4.    getPlates(n)
5.    place plates on table
6.    getUtensils(n)
7.    place knives and forks on table
8.    go back onto couch
```

Algorithm to start a Car

Algorithm to start a Car



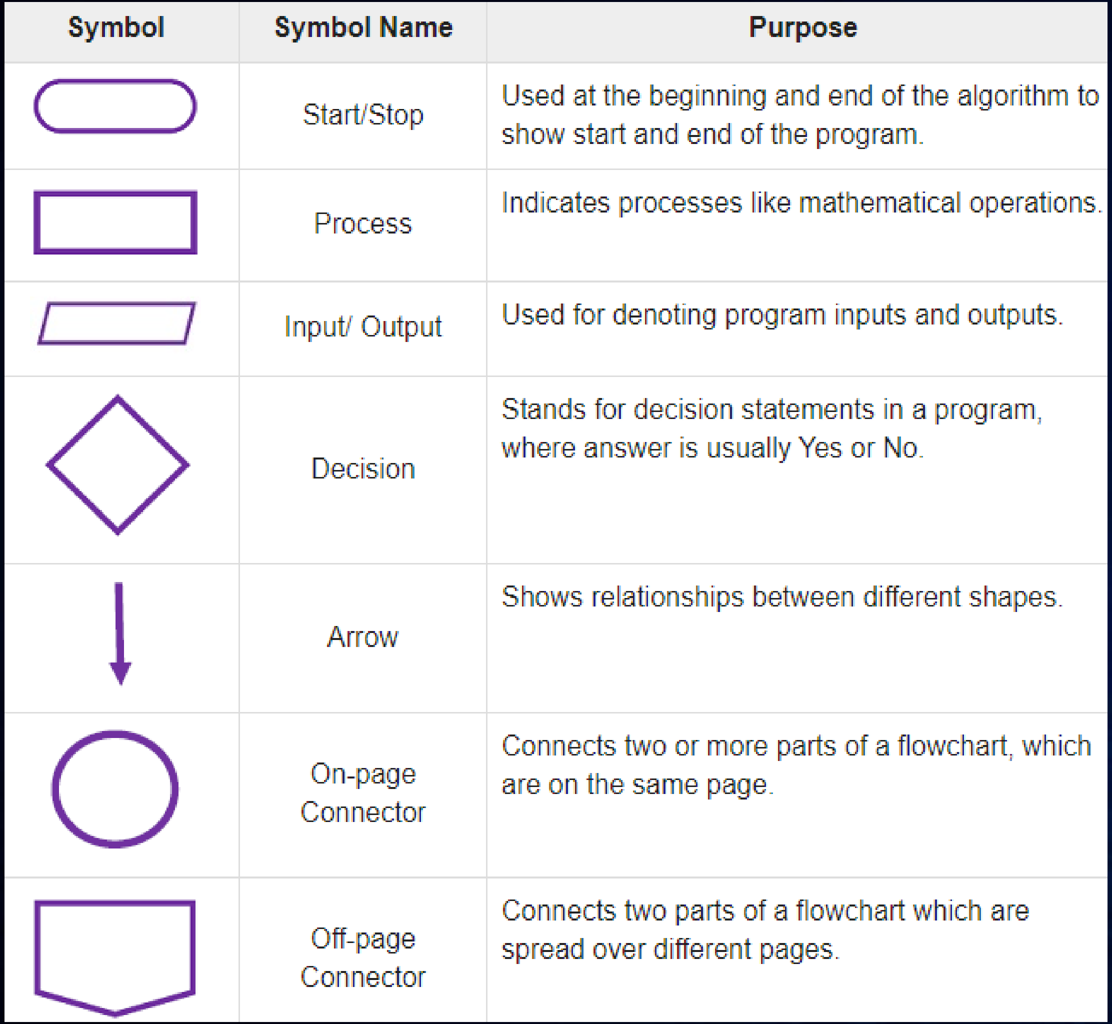
1. Insert the key.
2. Make sure the transmission is in Park (or Neutral).
3. Turn the key to the start position.
4. If the engine starts within six seconds, release the key to the ignition position.
5. If the engine doesn't start in six seconds, release the key and gas pedal, wait ten seconds, and repeat Steps 3 through 5, but not more than five times.
6. If the car doesn't start, call the garage.

Flowchart

Flowchart

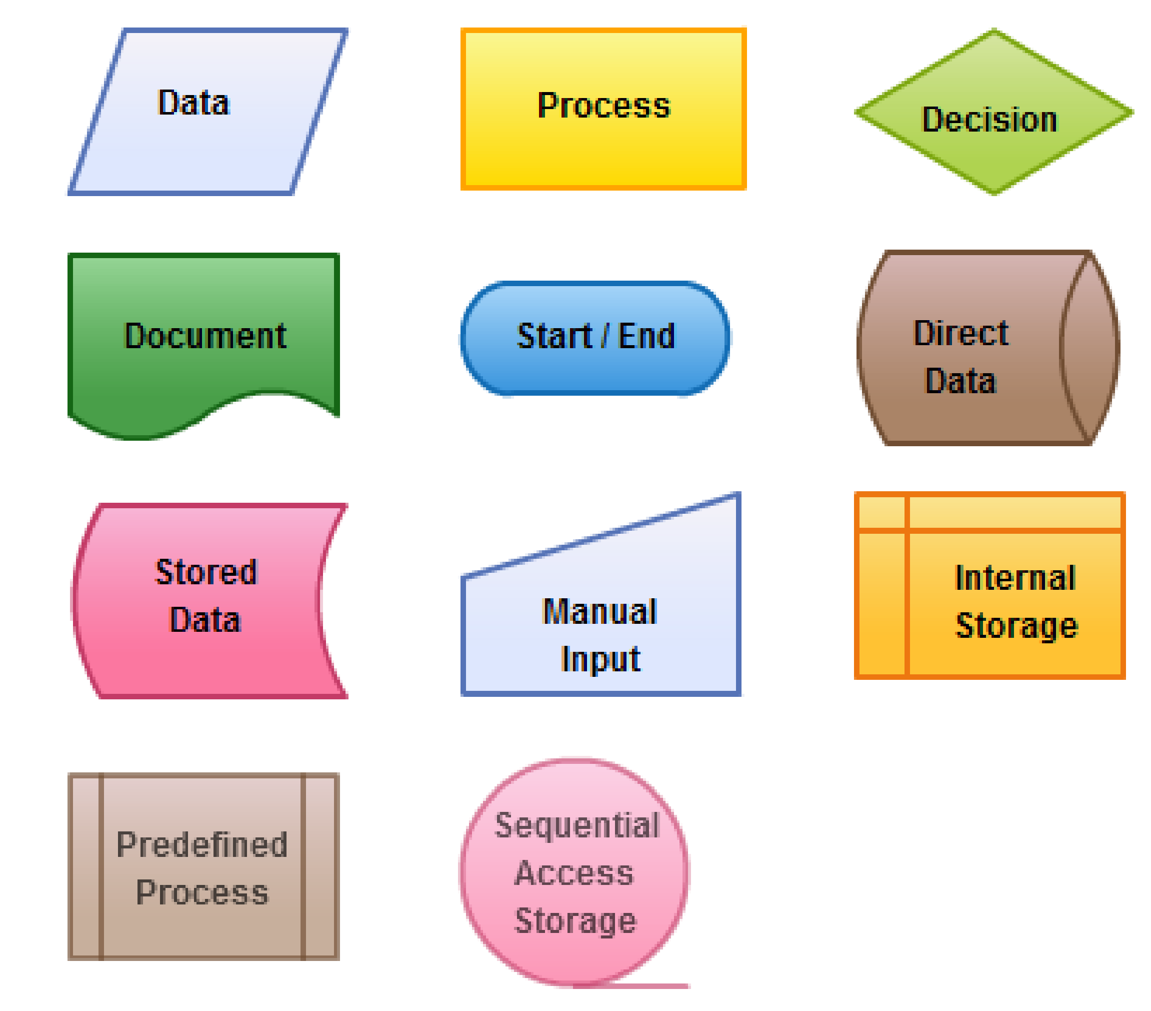
- A flowchart is simply a graphical representation of steps.
- It shows steps in sequential order and is widely used in presenting the flow of algorithms, workflow or processes.
- Typically, a flowchart shows the steps as boxes of various kinds, and their order by connecting them with arrows

Flowchart



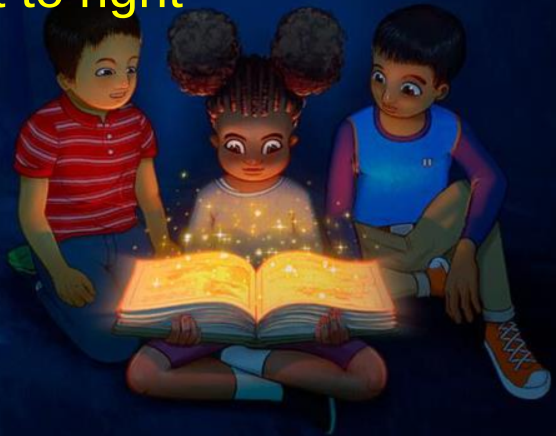
Symbol	Symbol Name	Purpose
⬭	Start/Stop	Used at the beginning and end of the algorithm to show start and end of the program.
▭	Process	Indicates processes like mathematical operations.
▱	Input/ Output	Used for denoting program inputs and outputs.
◇	Decision	Stands for decision statements in a program, where answer is usually Yes or No.
↓	Arrow	Shows relationships between different shapes.
○	On-page Connector	Connects two or more parts of a flowchart, which are on the same page.
⬠	Off-page Connector	Connects two parts of a flowchart which are spread over different pages.

Elements

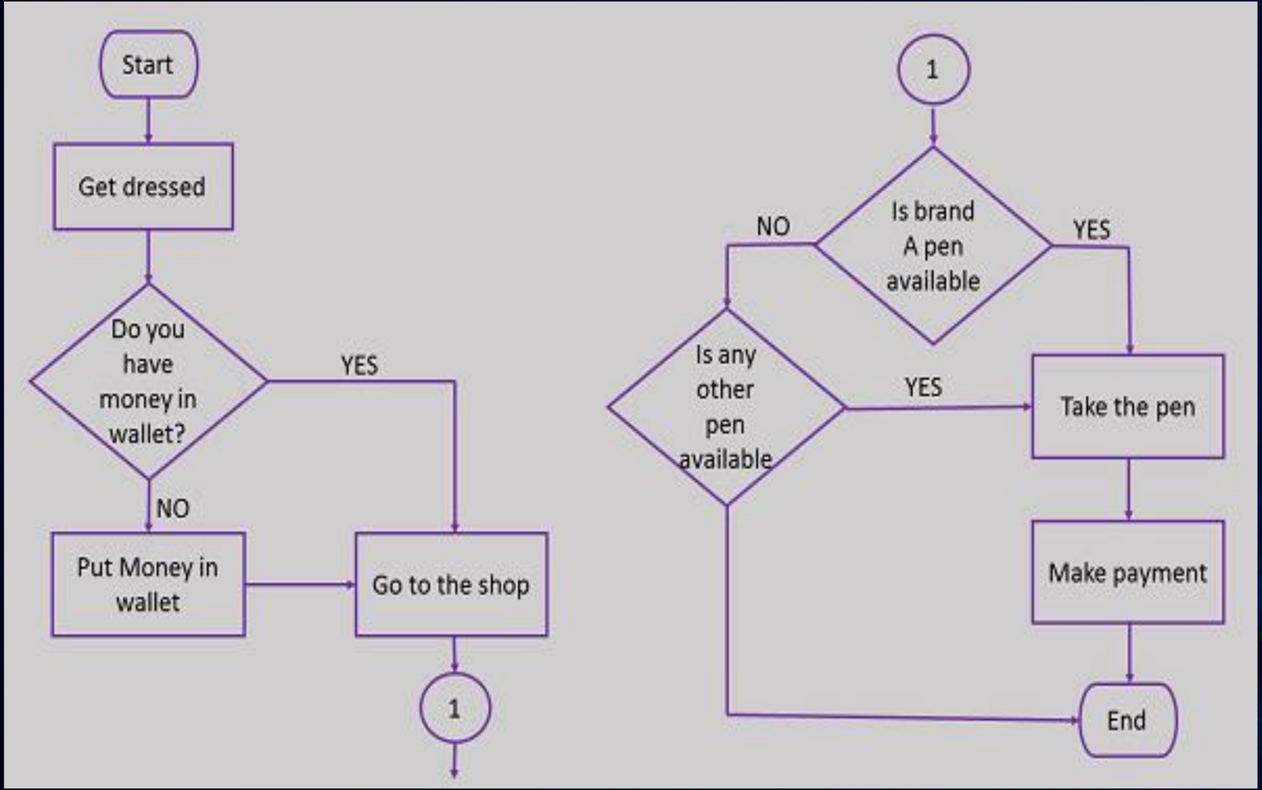

Data
Process
Decision
Document
Start / End
Direct Data
Stored Data
Manual Input
Internal Storage
Predefined Process
Sequential Access Storage

Developing Guidelines

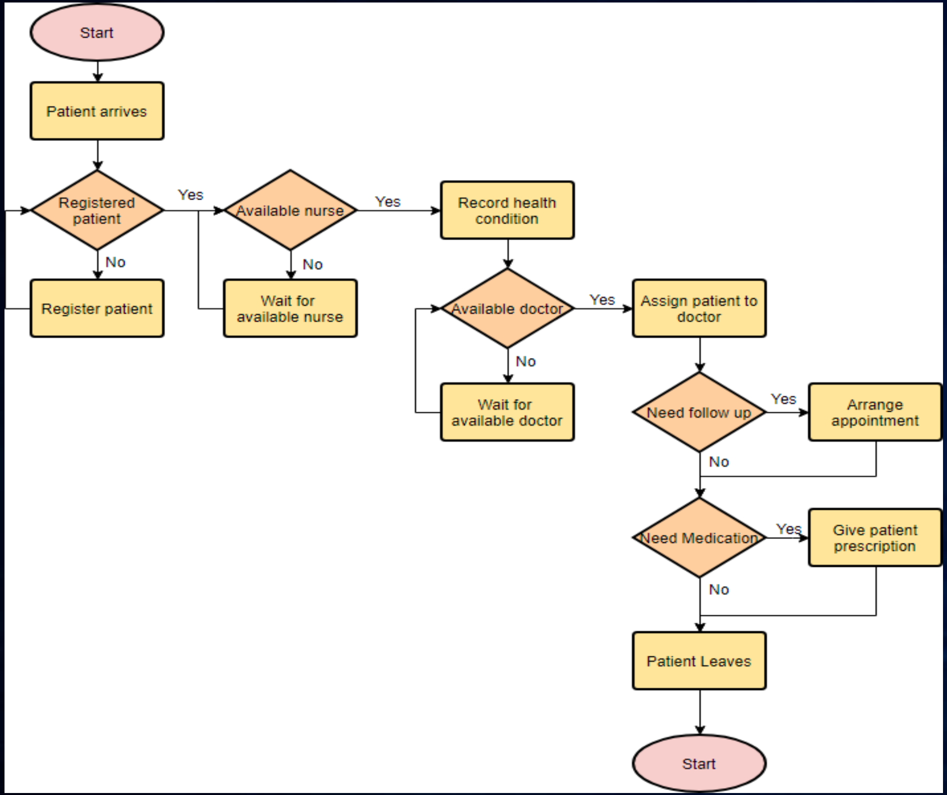
- Flowchart can have only one start and one stop symbol
- On-page connectors are referenced using numbers
- Off-page connectors are referenced using alphabets
- General flow of processes is top to bottom or left to right
- Arrows should not cross each other



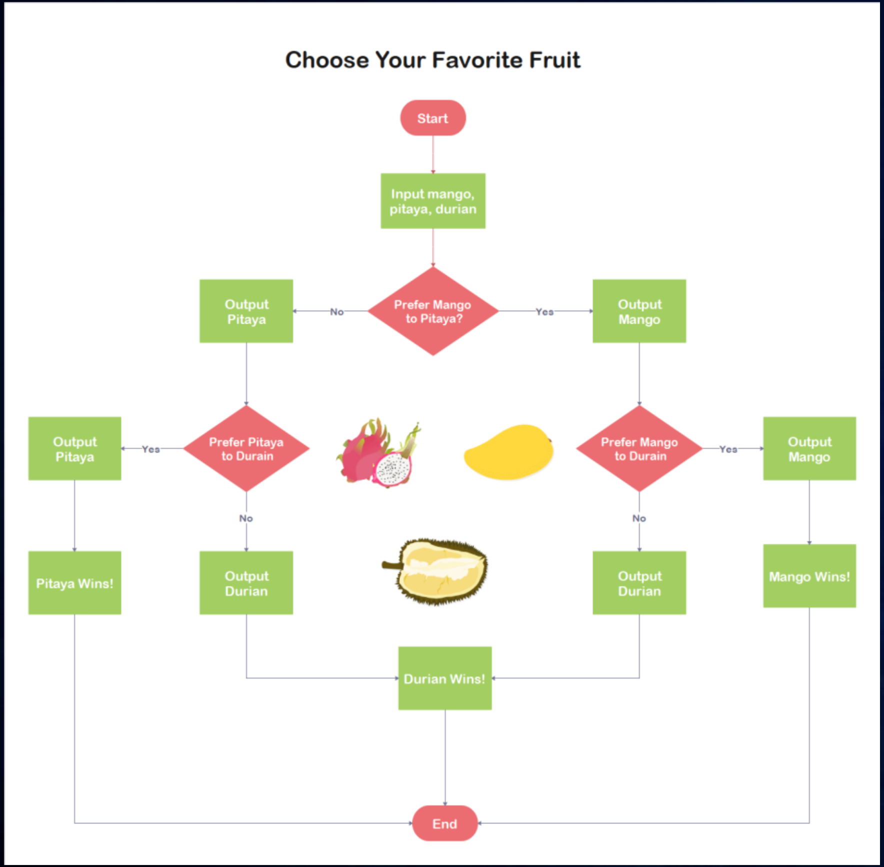
Sample


Start
Get dressed
Do you have money in wallet?
YES
NO
Put Money in wallet
Go to the shop
1
1
Is brand A pen available
NO
YES
Is any other pen available
YES
Take the pen
Make payment
End

Sample


Start
Patient arrives
Registered patient
Yes
No
Register patient
Available nurse
Yes
No
Wait for available nurse
Record health condition
Available doctor
Yes
No
Wait for available doctor
Assign patient to doctor
Need follow up
Yes
No
Arrange appointment
Need Medication
Yes
No
Give patient prescription
Patient Leaves
Start

Sample



Choose Your Favorite Fruit

Start

Input mango, pitaya, durian

Prefer Mango to Pitaya?

No

Output Pitaya

Yes

Output Mango

Prefer Pitaya to Durain

Yes

Output Pitaya

No

Output Durian

Prefer Mango to Durain

Yes

Output Mango

No

Output Durian

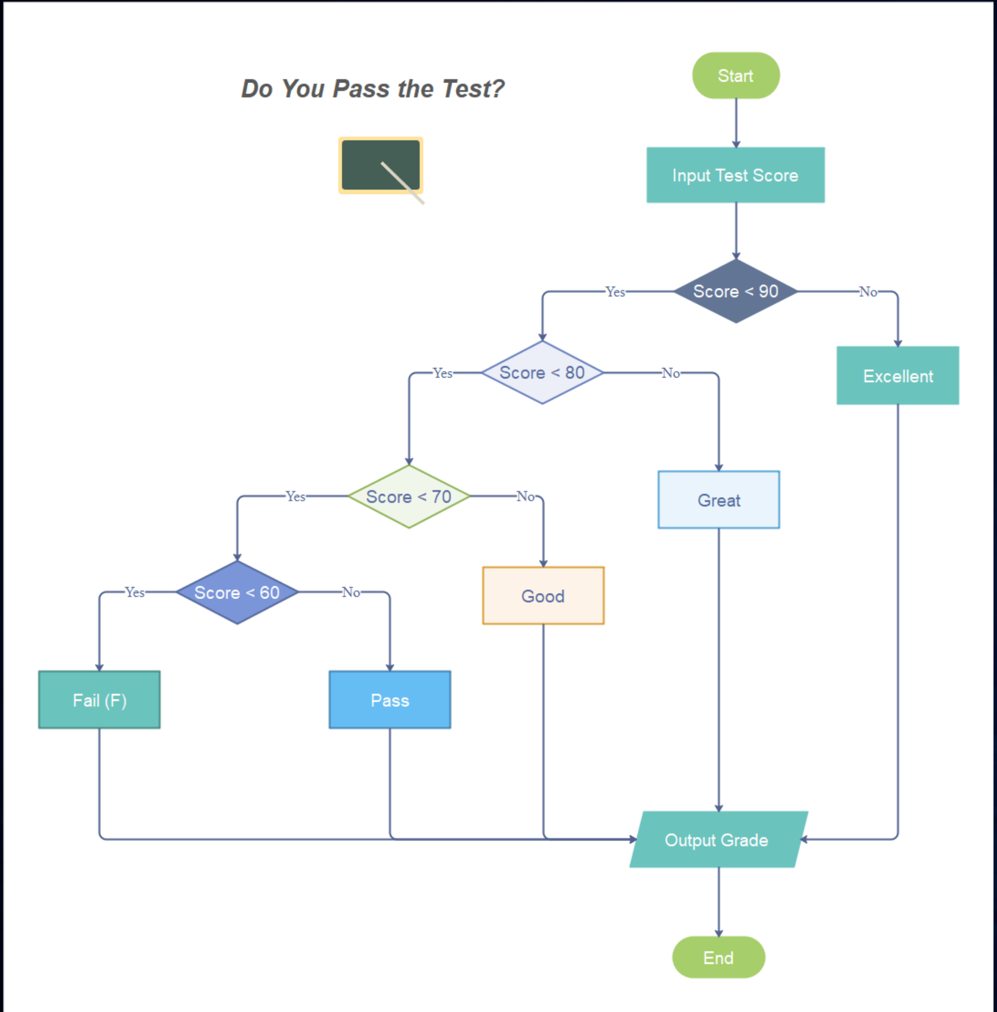
Pitaya Wins!

Durian Wins!

Mango Wins!

End

Sample



Do You Pass the Test?

Start

Input Test Score

Score < 90

Yes

No

Excellent

Score < 80

Yes

No

Great

Score < 70

Yes

No

Good

Score < 60

Yes

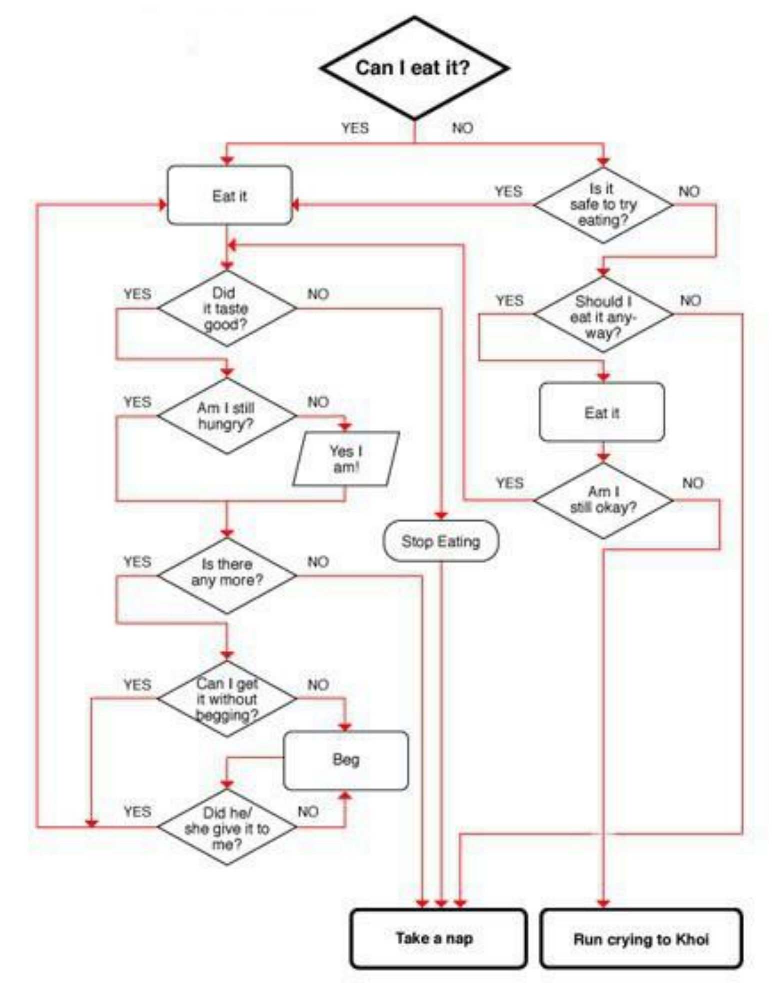
No

Fail (F)

Pass

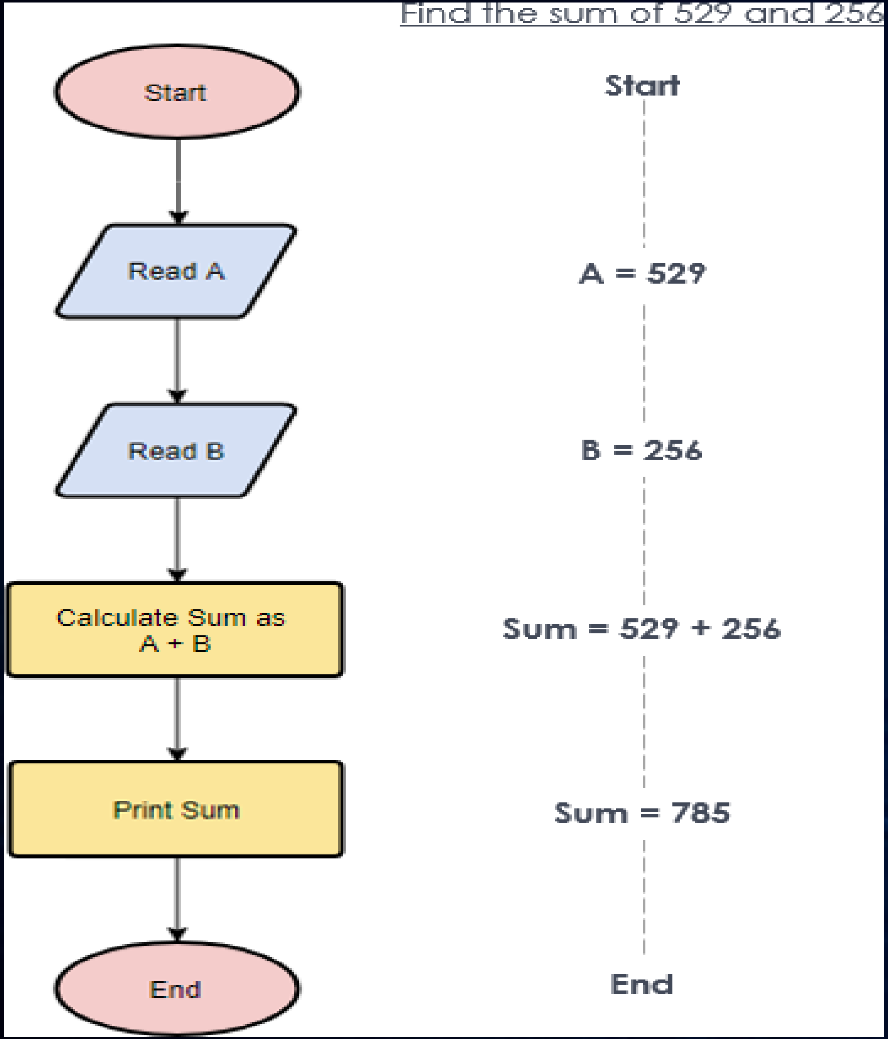
Output Grade

End


Can I eat it?
YES
NO
Eat it
YES
Is it safe to try eating?
NO
YES
Did it taste good?
NO
YES
Should I eat it any-way?
NO
YES
Am I still hungry?
NO
Yes I am!
Eat it
YES
Am I still okay?
NO
Stop Eating
YES
Is there any more?
NO
YES
Can I get it without begging?
NO
Beg
YES
Did he/ she give it to me?
NO
Take a nap
Run crying to Khoi

Sample

Sample



Find the sum of 529 and 256

Start

Read A

Read B

Calculate Sum as A + B

Print Sum

End

Start

A = 529

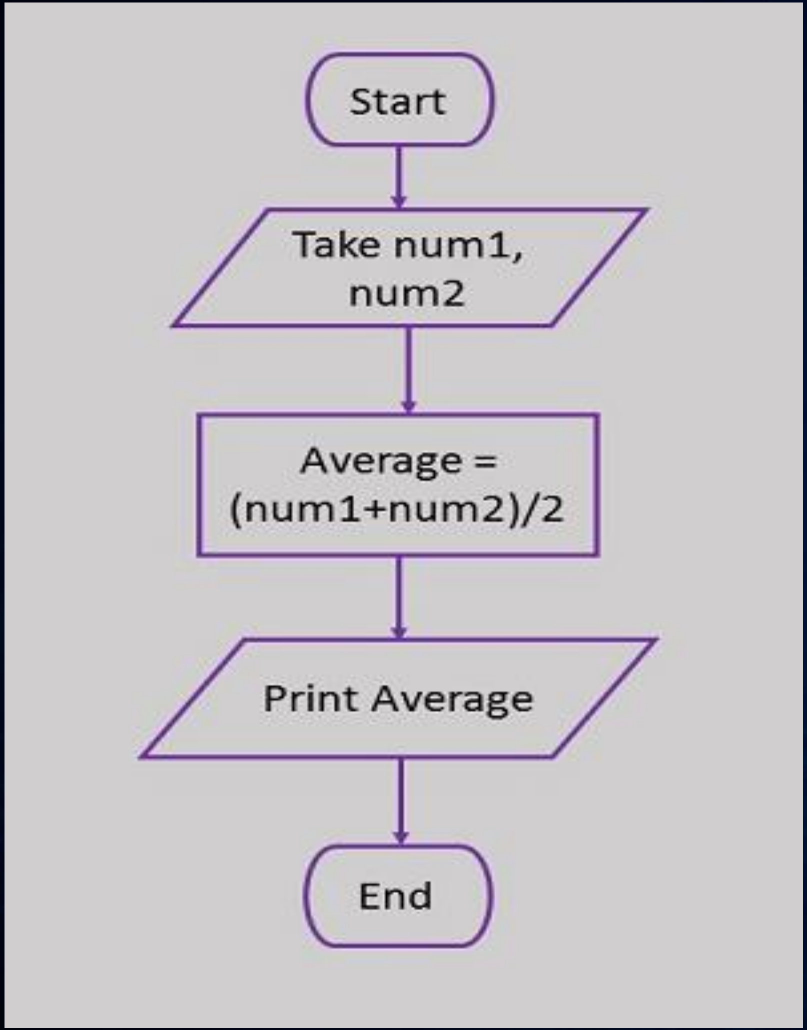
B = 256

Sum = 529 + 256

Sum = 785

End

Sample



Start

Take num1, num2

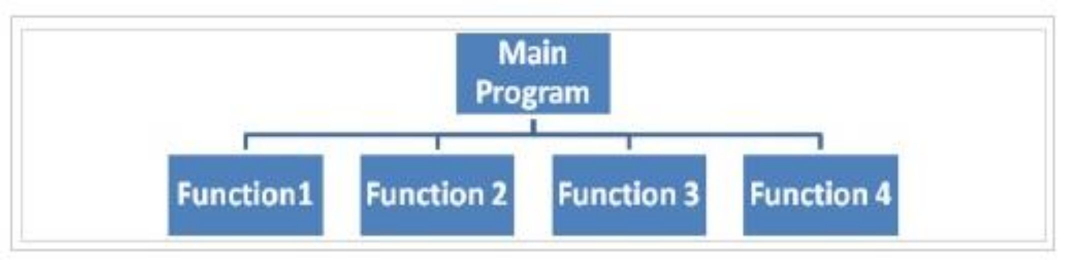
Average = (num1+num2)/2

Print Average

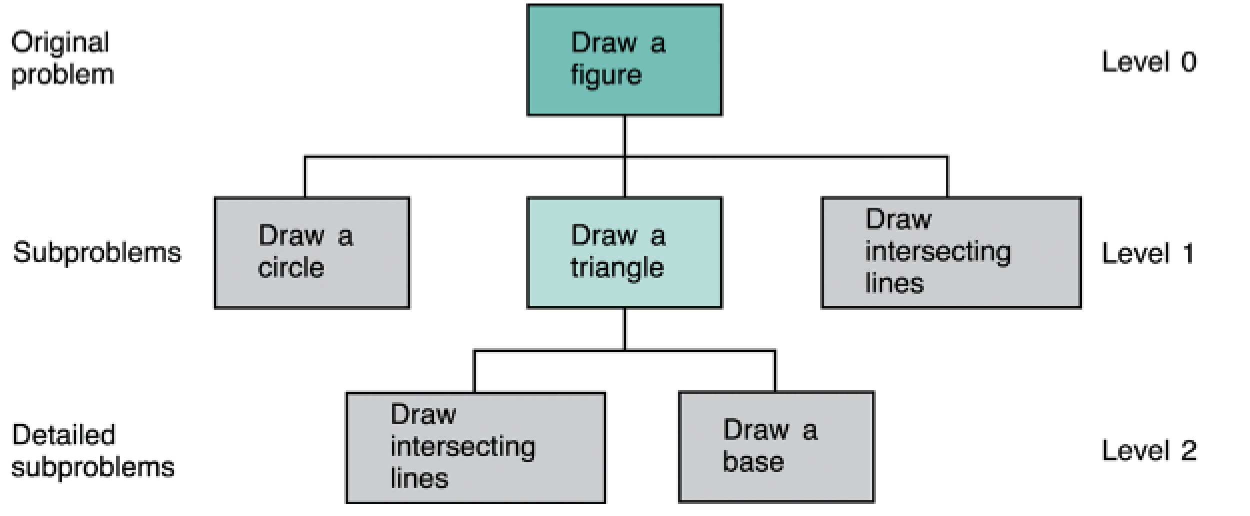
End

Topdown Design

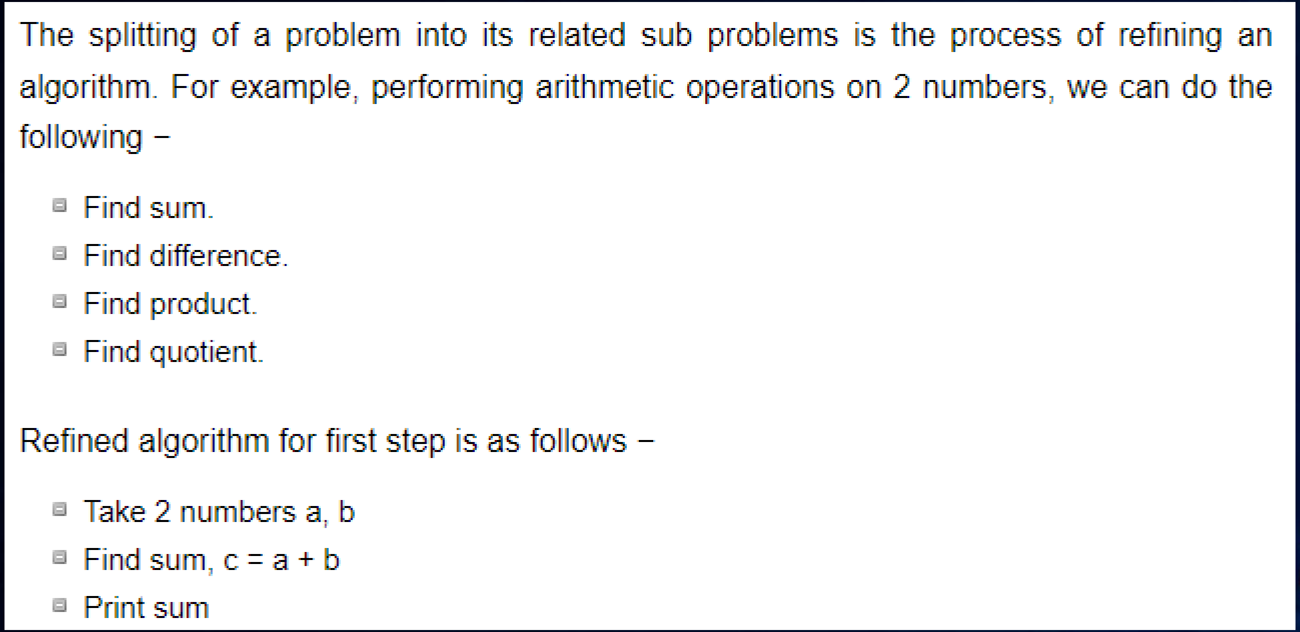
- It is a problem solving method in which a complex problem is solved by splitting into sub problems.
- Structure chart is a documentation tool that shows the relationships among the sub problems of a problem.



Topdown Design - Example



Topdown Design



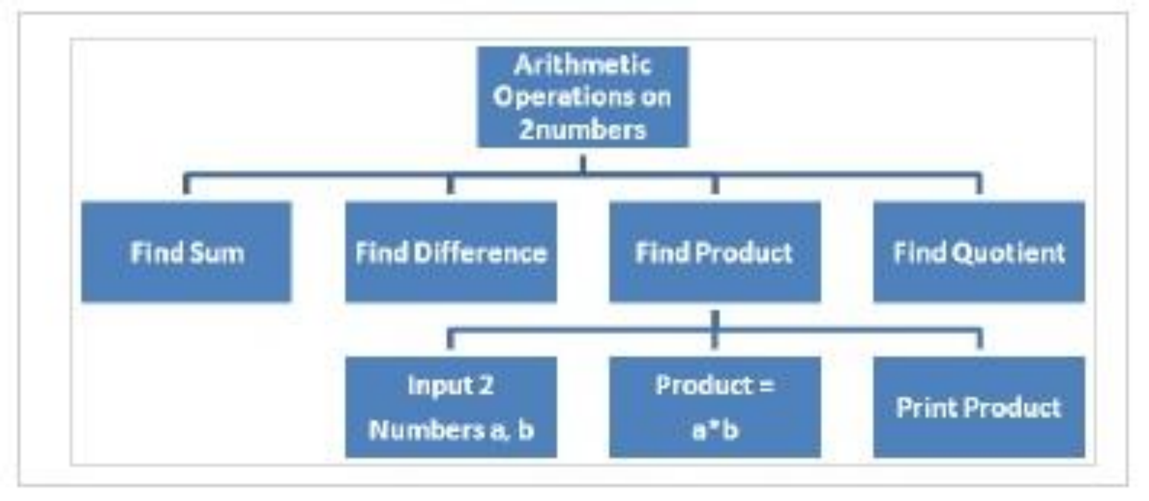
The splitting of a problem into its related sub problems is the process of refining an algorithm. For example, performing arithmetic operations on 2 numbers, we can do the following −

- Find sum.
- Find difference.
- Find product.
- Find quotient.

Refined algorithm for first step is as follows −

- Take 2 numbers a, b
- Find sum, c = a + b
- Print sum

Structure Chart



Arithmetic Operations on 2numbers

Find Sum

Find Difference

Find Product

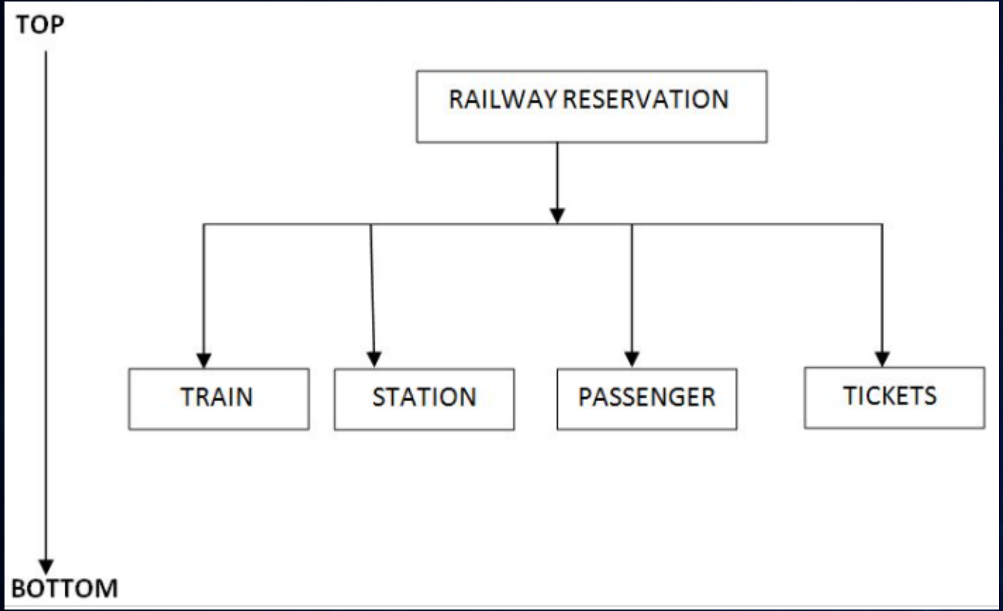
Find Quotient

Input 2 Numbers a, b

Product = a*b

Print Product

Structure Chart



TOP

RAILWAY RESERVATION

TRAIN

STATION

PASSENGER

TICKETS

BOTTOM

Stepwise Refinement

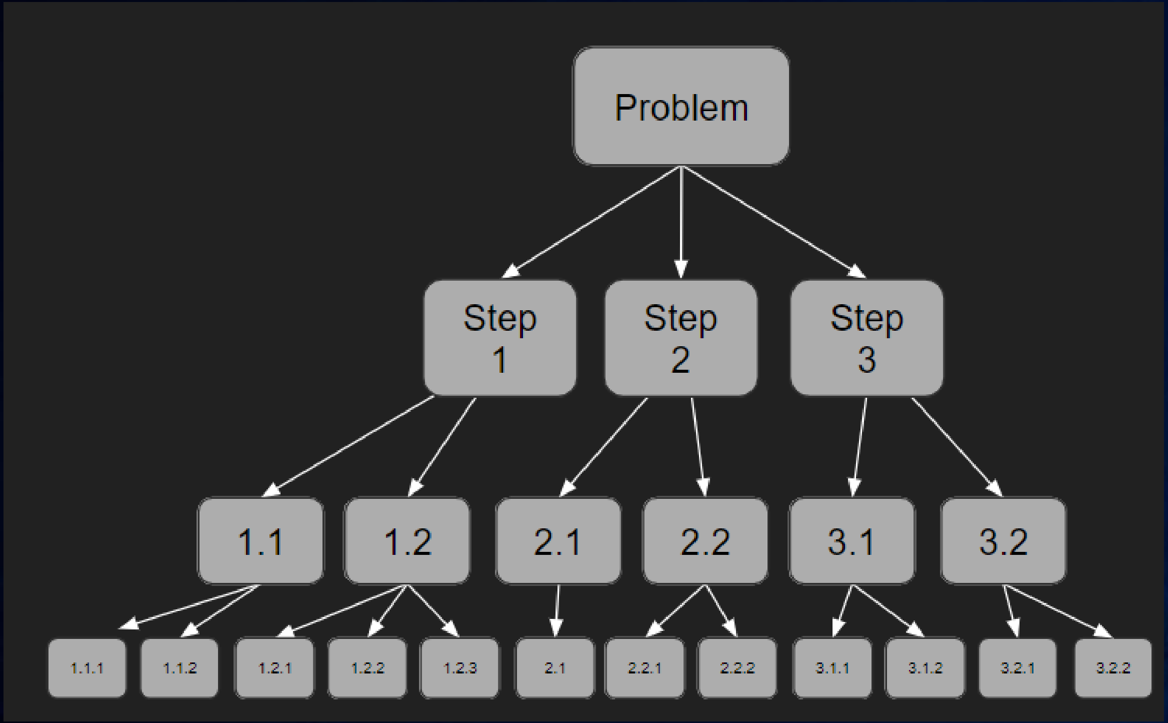
- Stepwise refinement is a basic technique for **low-level design**.

- Invented by Niklaus Wirth in 1971, it may be the oldest systematic approach to software design still in use. But it remains quite relevant to modern programming.

- Stepwise refinement is a discipline of taking small, easily defended steps from a very generic view of an algorithm, adding a few details at each step, until the path to an actual program becomes clear.

- It’s a great answer to that awful, hollow feeling that you may get when staring at a blank sheet of paper or editor window and wondering, “how in the world am I supposed come up with a program to do …?”

Stepwise Refinement



Problem

Step 1

Step 2

Step 3

1.1

1.2

2.1

2.2

3.1

3.2

1.1.1

1.1.2

1.2.1

1.2.2

1.2.3

2.1

2.2.1

2.2.2

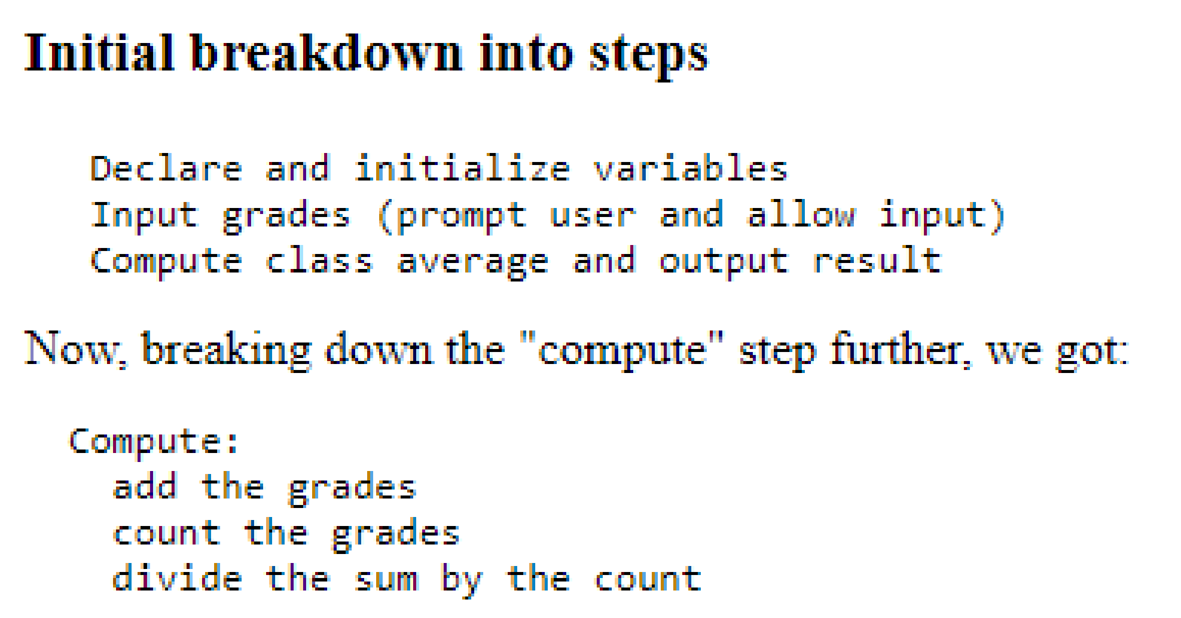
3.1.1

3.1.2

3.2.1

3.2.2

Stepwise Refinement



Initial breakdown into steps

```
Declare and initialize variables
Input grades (prompt user and allow input)
Compute class average and output result
```

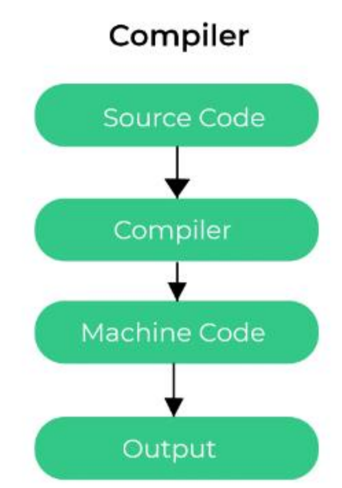
Now, breaking down the "compute" step further, we got:

```
Compute:
  add the grades
  count the grades
  divide the sum by the count
```

Programming Environment

- Text Editor Windows Notepad, OS Edit command, Brief, Epsilon, EMACS, and vim or vi.
- The C Compiler
- Installation on UNIX/Linux
- Installation on Mac OS -Xcode GNU compiler
- Installation on Windows - MinGW (gcc, g++, ar, ranlib, dlltool)

Compilers



Compiler

Source Code

Compiler

Machine Code

Output

Definition

- A Compiler is a program that translates source code from a high-level programming language to a lower level language computer understandable language(e.g. assembly language, object code, or machine code) to create an executable program

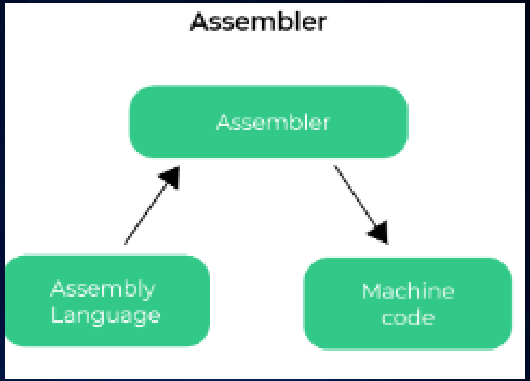
Features

- It is more intelligent than interpreter because it goes through the entire code at once
- It can tell the possible errors and limits and ranges.
- But this makes it’s operating time a little slower
- It is platform-dependent
- It help to detect error and get displayed after reading the entire code by compiler.

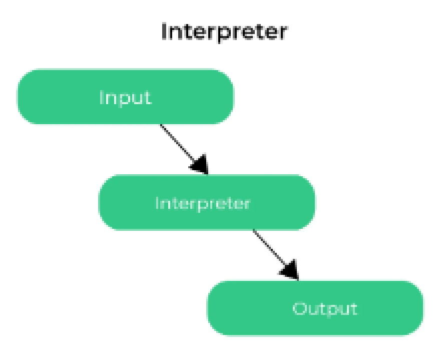
Assemblers

- In computer science, an assembler is a program that converts the assembly language into machine code.

- The output of an assembler is called an object file, which contains a combination of machine instructions as well as the data required to place these instructions in memory



Interpreter

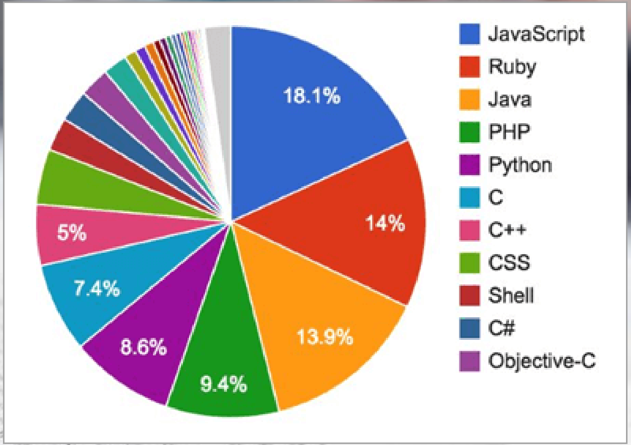


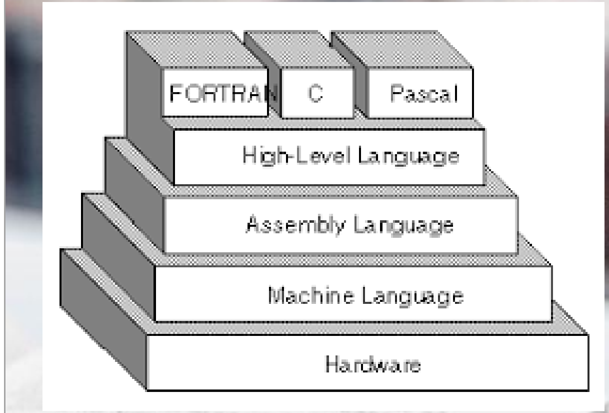
- An interpreter is also a program like a compiler that converts assembly language into Machine Code

- But an interpreter goes through one line of code at a time and executes it and then goes on to the next line of the code and then the next and keeps going on until there is an error in the line or the code has completed.

Implementation Issues

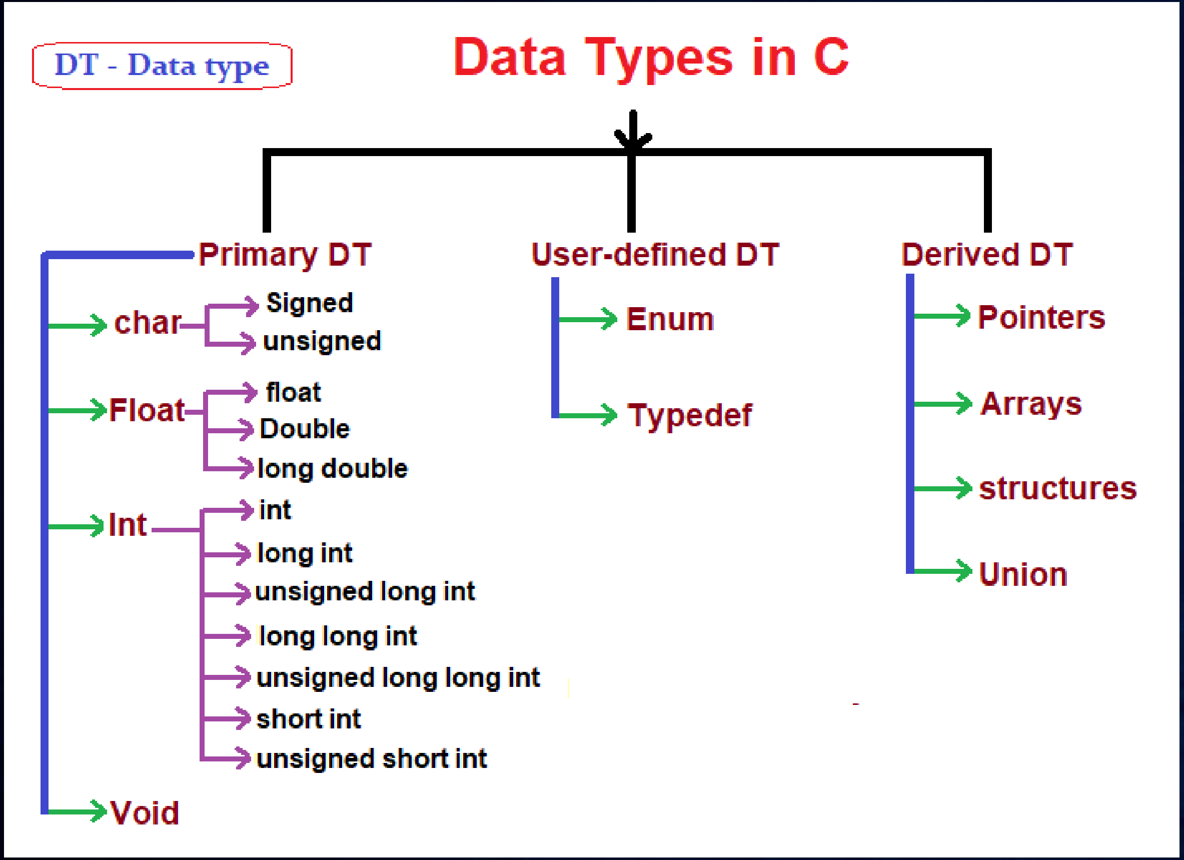
- Buffer And Memory Related Operations
- Handling Warnings
- Command Execution Vulnerabilities
- Format String Vulnerabilities
- Null Pointers / Dangling Pointer Access
- Code reuse
- Version Management





High Level Languages

Datatypes

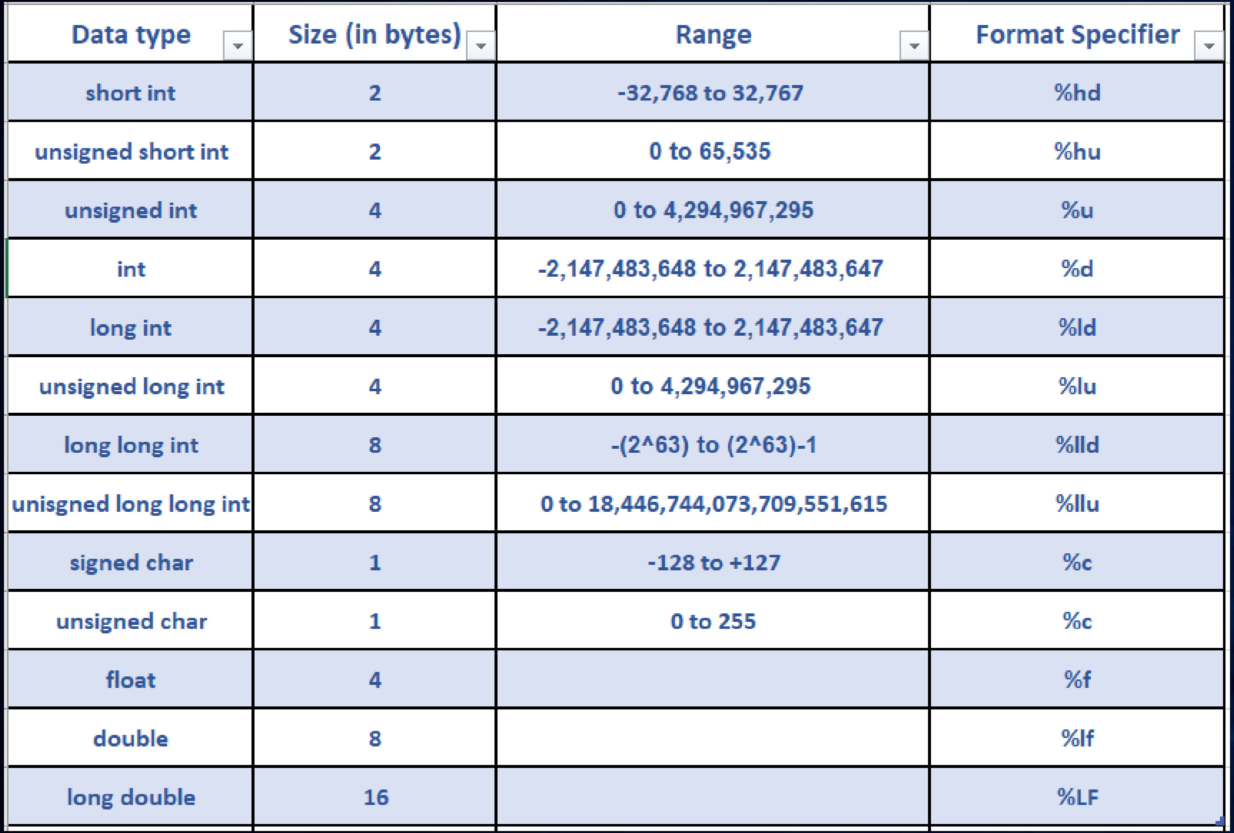


DT - Data type

Data Types in C

- Primary DT
  - char
    - Signed
    - unsigned
  - Float
    - float
    - Double
    - long double
  - Int
    - int
    - long int
    - unsigned long int
    - long long int
    - unsigned long long int
    - short int
    - unsigned short int
  - Void
- User-defined DT
  - Enum
  - Typedef
- Derived DT
  - Pointers
  - Arrays
  - structures
  - Union

Datatypes



Data type	Size (in bytes)	Range	Format Specifier
short int	2	-32,768 to 32,767	%hd
unsigned short int	2	0 to 65,535	%hu
unsigned int	4	0 to 4,294,967,295	%u
int	4	-2,147,483,648 to 2,147,483,647	%d
long int	4	-2,147,483,648 to 2,147,483,647	%ld
unsigned long int	4	0 to 4,294,967,295	%lu
long long int	8	-(2^63) to (2^63)-1	%lld
unisgned long long int	8	0 to 18,446,744,073,709,551,615	%llu
signed char	1	-128 to +127	%c
unsigned char	1	0 to 255	%c
float	4		%f
double	8		%lf
long double	16		%LF

Credits

- https://people.scs.carleton.ca/~lanthier/teaching/ProcessingNotes/COMP1405_Ch1_IntroductionToComputerScience.pdf

Thanks!

Any questions?

You can find me at:

- kramdharma@gmail.com