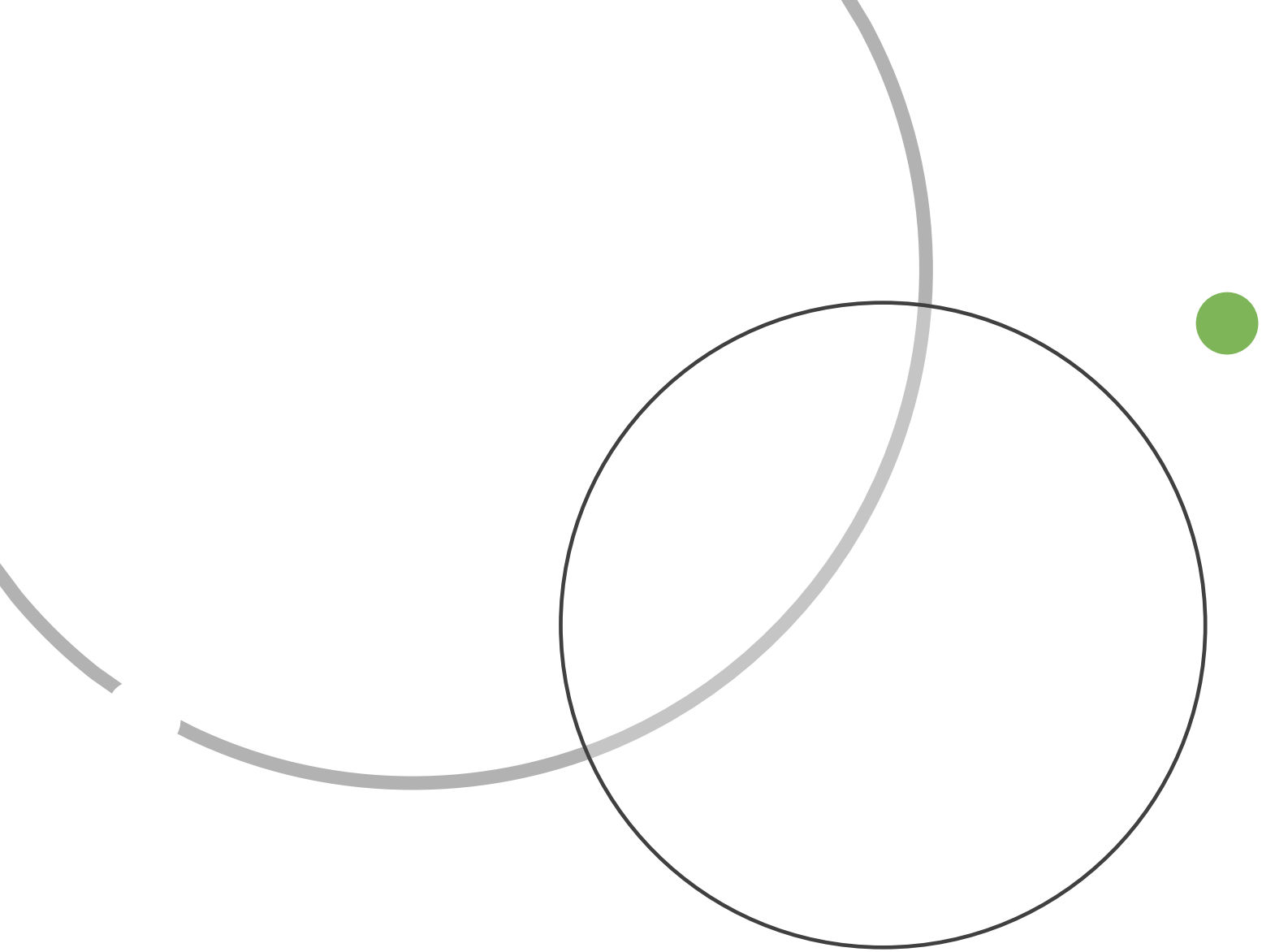


A Quickly Review of Algorithms

Carlos Jaime BARRIOS HERNANDEZ, PhD.

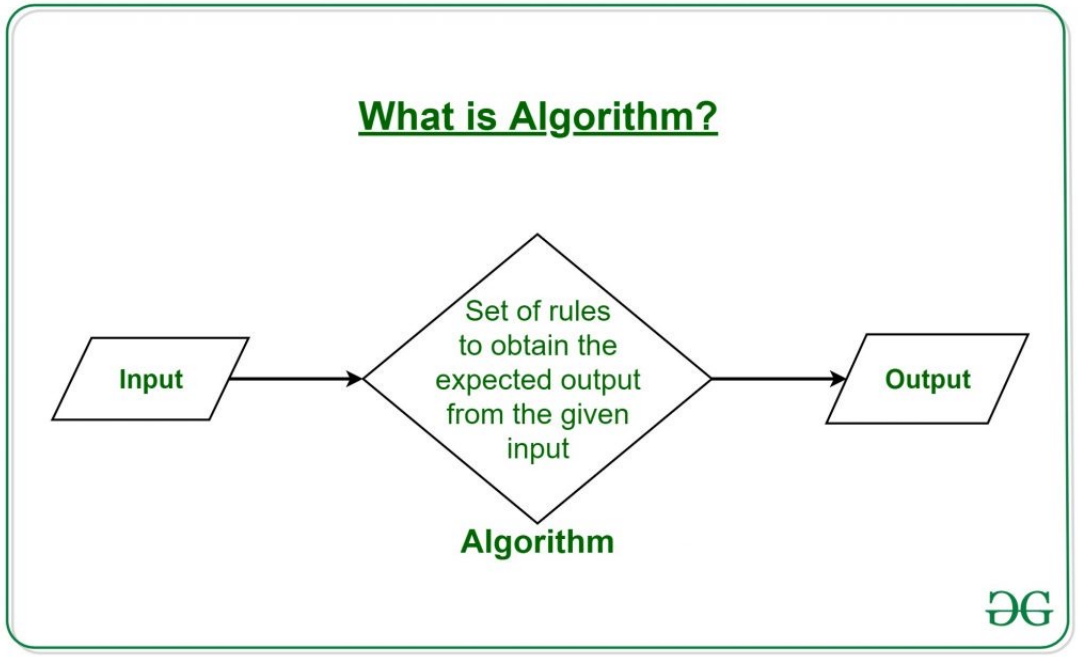
@carlosjaimebh

algorithm

noun

Word used by programmers when they do not want to explain what they did.

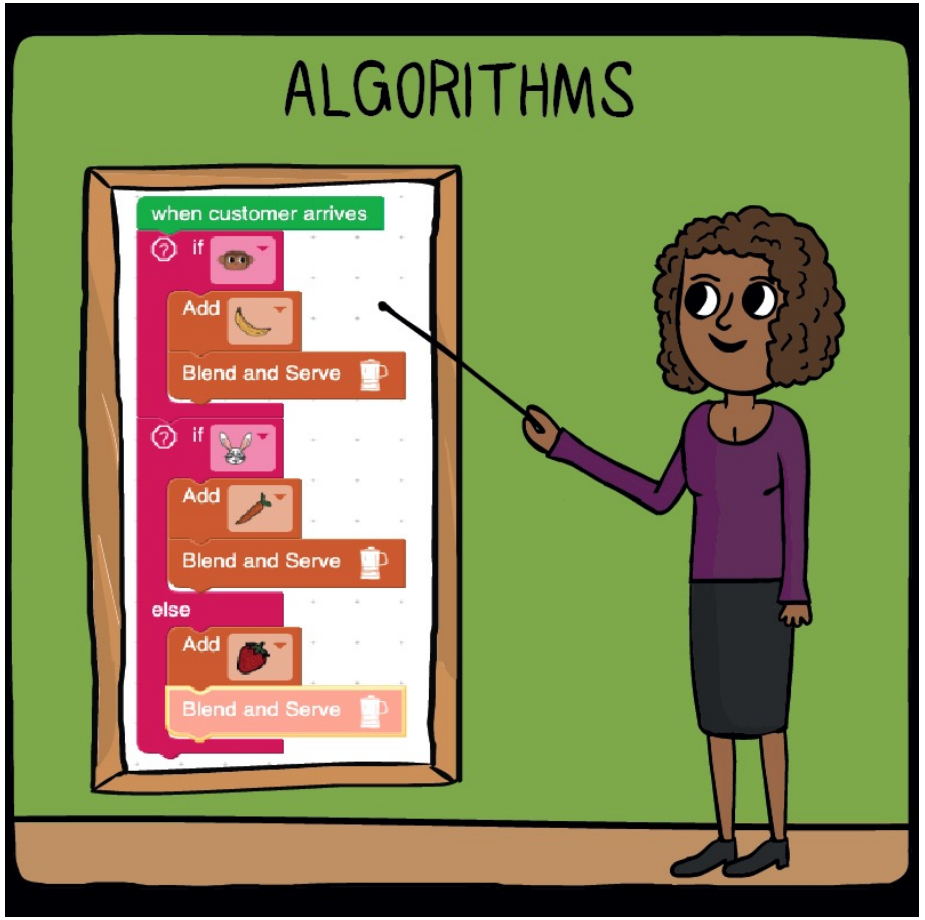
Introducing Algorithms…



- The word **Algorithm** means "a process or set of rules to be followed in calculations or other problem-solving operations". Therefore Algorithm refers to a set of rules/instructions that step-by-step define how a work is to be executed upon in order to get the expected results.

From: https://www.geeksforgeeks.org/introduction-to-algorithms/

Introducing Algorithms…



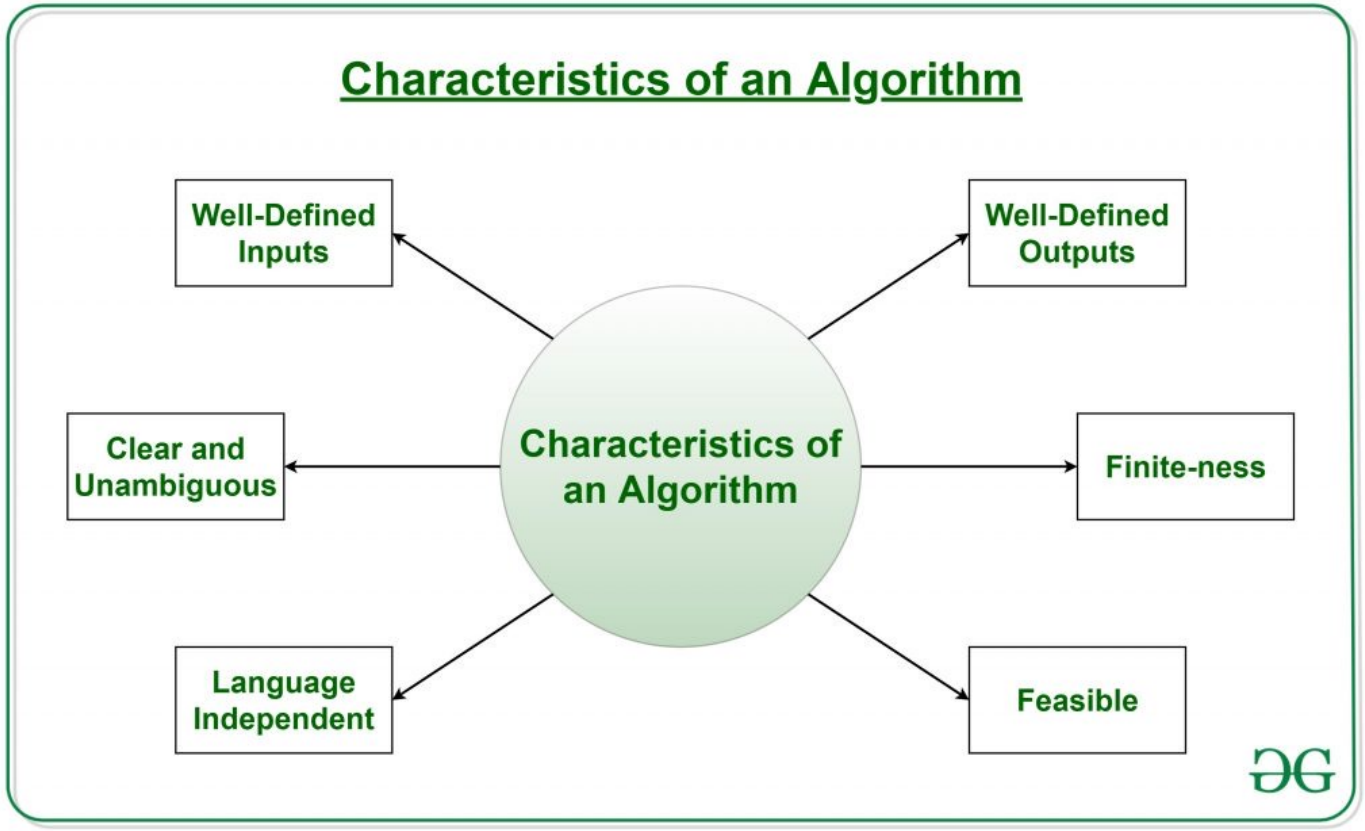
- Similarly, algorithms help to do a task in programming to get the expected output. The Algorithm designed are language or implementation independent, i.e. they are just plain instructions that can be implemented in any language, and yet the output will be the same, as expected.
  - Language implementantion (or other implementation) is a traduction to be executed in a context (or runtime).

From: https://www.geeksforgeeks.org/introduction-to-algorithms/

https://en.wikipedia.org/wiki/Algorithm

Characteristics of an Algorithm

- **Clear and Unambiguous**: Algorithm should be clear and unambiguous. Each of its steps should be clear in all aspects and must lead to only one meaning.
- **Well-Defined Inputs**: If an algorithm says to take inputs, it should be well-defined inputs.
- **Well-Defined Outputs:** The algorithm must clearly define what output will be yielded and it should be well-defined as well.
- **Finite-ness:** The algorithm must be finite, i.e. it should not end up in an infinite loops or similar.
- **Feasible:** The algorithm must be simple, generic and practical, such that it can be executed upon will the available resources. It must not contain some future technology, or anything.
- **Language Independent:** The Algorithm designed must be language-independent, i.e. it must be just plain instructions that can be implemented in any language, and yet the output will be same, as expected.

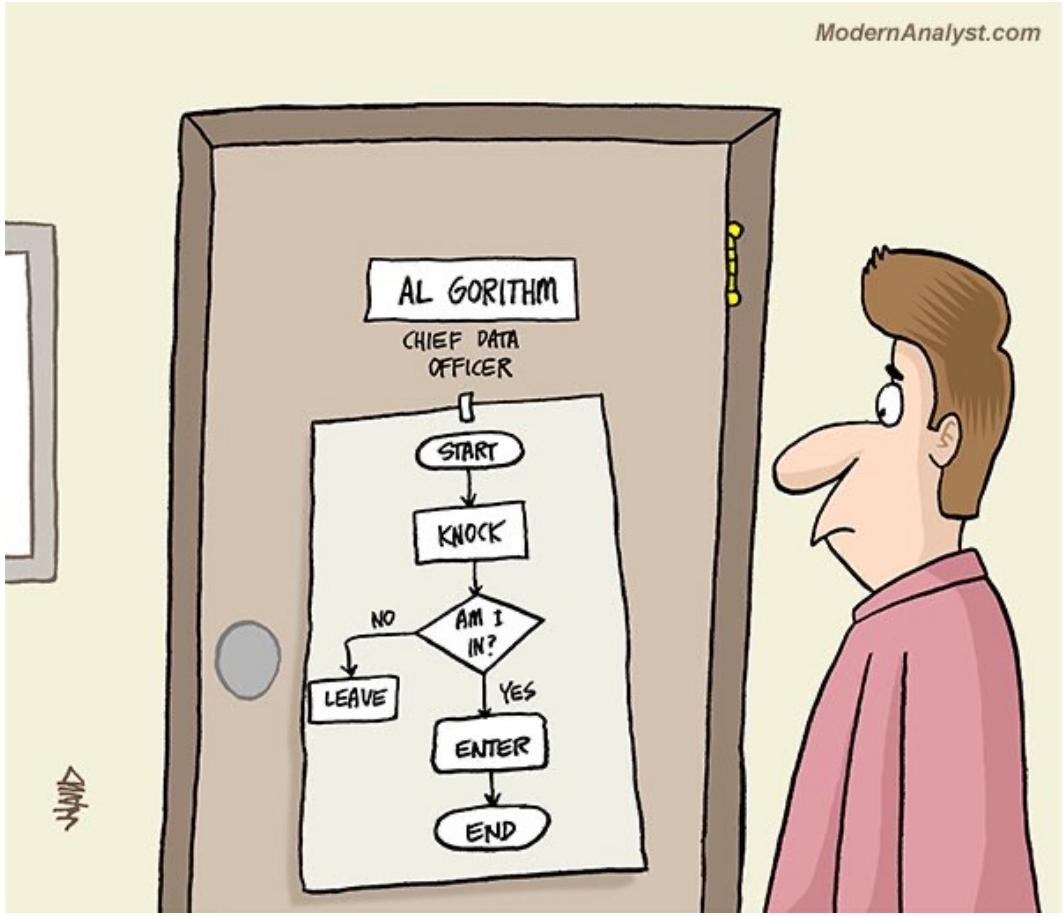


From: https://www.geeksforgeeks.org/introduction-to-algorithms/

https://en.wikipedia.org/wiki/Algorithm

Advantanges and Disadvantages

- **Advantages of Algorithms:**
- It is easy to understand.
- Algorithm is a step-wise representation of a solution to a given problem.
- In Algorithm the problem is broken down into smaller pieces or steps hence, it is easier for the programmer to convert it into an actual program.
- **Disadvantages of Algorithms:**
- Writing an algorithm takes a long time so it is time-consuming.
- Branching and Looping statements are difficult to show in Algorithms.
- Linear Thinking (However, it is more for the way to understand and to create algorithms).
  - Parallel Thinking (to see later)



From: https://www.geeksforgeeks.org/introduction-to-algorithms/

Designing algorithms

In order to write an algorithm, following things are needed as a pre-requisite:

- The **problem** that is to be solved by this algorithm.
- The **constraints** of the problem that must be considered while solving the problem.
- The **input** to be taken to solve the problem.
- The **output** to be expected when the problem the is solved.
- The **solution** to this problem, in the given constraints.

Then the algorithm is written with the help of above parameters such that it solves the problem.

However, some authors propose a common and formal « algorithm » to define (and write) algorithms:

1. Problem definition
2. Development of a model
3. Specification of the algorithm
4. Designing an algorithm
5. Checking the correctness of the algorithm
6. Analysis of algorithm
7. Implementation of algorithm
8. Program testing
9. Documentation preparation[c/

From: https://www.geeksforgeeks.org/introduction-to-algorithms/

https://en.wikipedia.org/wiki/Algorithm

Example: Consider the example to add three numbers and print the sum. (1/3) (Sum = Num_1+Num2+Num3)

- **Step 1: Fulfilling the pre-requisites**
  As discussed above, in order to write an algorithm, its pre-requisites must be fulfilled.
  - **The problem that is to be solved by this algorithm**: Add 3 numbers and print their sum.
  - **The constraints of the problem that must be considered while solving the problem**: The numbers must contain only digits and no other characters.
  - **The input to be taken to solve the problem:** The three numbers to be added.
  - **The output to be expected when the problem the is solved:** The sum of the three numbers taken as the input.
  - **The solution to this problem, in the given constraints:** The solution consists of adding the 3 numbers. It can be done with the help of '+' operator, or bit-wise, or any other method.

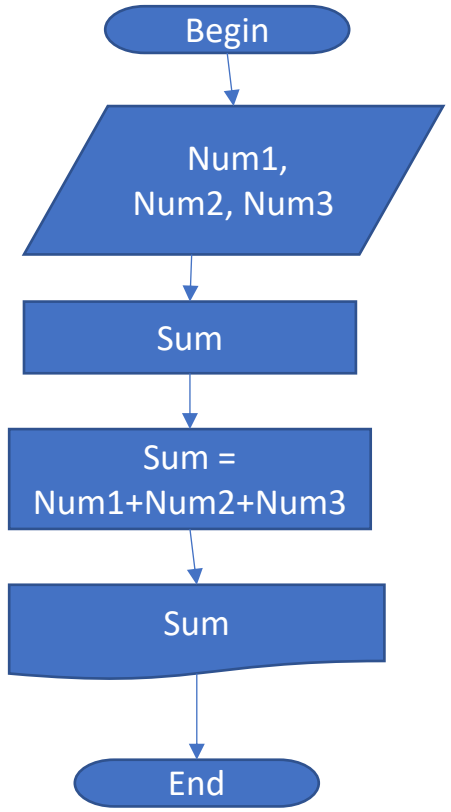
- **Step 2: Designing the algorithm**
  Now let's design the algorithm with the help of above pre-requisites:
  Algorithm to add 3 numbers and print their sum:
  - START
  - Declare 3 integer variables Num1, Num2 and Num3.
  - Take the three numbers, to be added, as inputs in variables num1, num2, and num3 respectively.
  - Declare an integer variable sum to store the resultant sum of the 3 numbers.
  - Add the 3 numbers and store the result in the variable sum.
  - Print the value of variable sum
  - END

This is a Pseudocode!

From: https://www.geeksforgeeks.org/introduction-to-algorithms/

Example: Consider the example to add three numbers and print the sum. (2/3) (Sum = Num_1+Num2+Num3)

- **Step 3: Flowchart**



(The graphic representation is very useful!)

Example: Consider the example to add three numbers and print the sum. (3/3) (Sum = Num_1+Num2+Num3)

- **Step 4: Testing the algorithm by implementing it.** (In order to test the algorithm, let's implement it in C language).

```c
// C program to add three numbers
// with the help of above designed algorithm

#include <stdio.h>

int main()
{

    // Variables to take the input of the 3 numbers
    int num1, num2, num3;

    // Variable to store the resultant sum
    int sum;

    // Take the 3 numbers as input
    printf("Enter the 1st number: ");
    scanf("%d", &num1);
    printf("%d\n", num1);

    printf("Enter the 2nd number: ");
    scanf("%d", &num2);
    printf("%d\n", num2);

    printf("Enter the 3rd number: ");
    scanf("%d", &num3);
    printf("%d\n", num3);

    // Calculate the sum using + operator
    // and store it in variable sum
    sum = num1 + num2 + num3;

    // Print the sum
    printf("\nSum of the 3 numbers is: %d", sum);

    return 0;
}
```

Output

```
Enter the 1st number: 0
Enter the 2nd number: 0
Enter the 3rd number: -1577141152
Sum of the 3 numbers is: -1577141152
```

From: https://www.geeksforgeeks.org/introduction-to-algorithms/

Possible Analysis of an Algorithm

- **Priori Analysis:** “Priori” means “before”. Hence Priori analysis means checking the algorithm before its implementation. In this, the algorithm is checked when it is written in the form of theoretical steps. This Efficiency of an algorithm is measured by assuming that all other factors, for example, processor speed, are constant and have no effect on the implementation. This is done usually by the algorithm designer. It is in this method, that the Algorithm Complexity is determined.
- **Posterior Analysis:** “Posterior” means “after”. Hence Posterior analysis means checking the algorithm after its implementation. In this, the algorithm is checked by implementing it in any programming language and executing it. This analysis helps to get the actual and real analysis report about correctness, space required, time consumed etc.
  - **Time Factor**: Time is measured by counting the number of key operations such as comparisons in the sorting algorithm.
  - **Space Factor**: Space is measured by counting the maximum memory space required by the algorithm.

From: https://www.geeksforgeeks.org/introduction-to-algorithms/

Complexity (1/2)

- **Space Complexity:** Space complexity of an algorithm refers to the amount of memory that this algorithm requires to execute and get the result. This can be for inputs, temporary operations, or outputs.

How to calculate Space Complexity?
The space complexity of an algorithm is calculated by determining following 2 components:

- **Fixed Part:** This refers to the space that is definitely required by the algorithm. For example, input variables, output variables, program size, etc.
- **Variable Part:** This refers to the space that can be different based on the implementation of the algorithm. For example, temporary variables, dynamic memory allocation, recursion stack space, etc.

From: https://www.geeksforgeeks.org/introduction-to-algorithms/

Complexity (2/2)

- **Time Complexity:** Time complexity of an algorithm refers to the amount of time that this algorithm requires to execute and get the result. This can be for normal operations, conditional if-else statements, loop statements, etc.

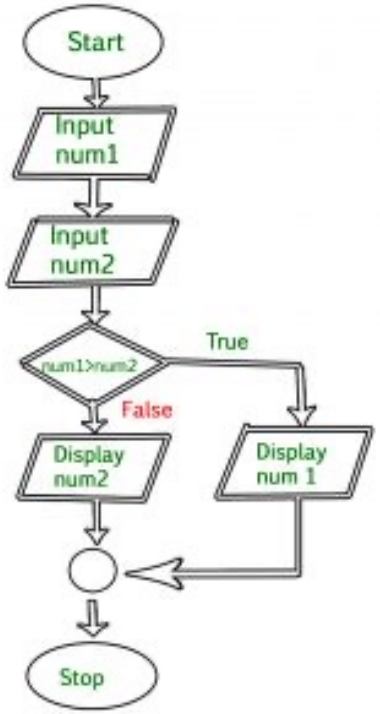
How to calculate Time Complexity?
The time complexity of an algorithm is also calculated by determining following 2 components:

- **Constant time part:** Any instruction that is executed just once comes in this part. For example, input, output, if-else, switch, etc.
- **Variable Time Part:** Any instruction that is executed more than once, say n times, comes in this part. For example, loops, recursion, etc.

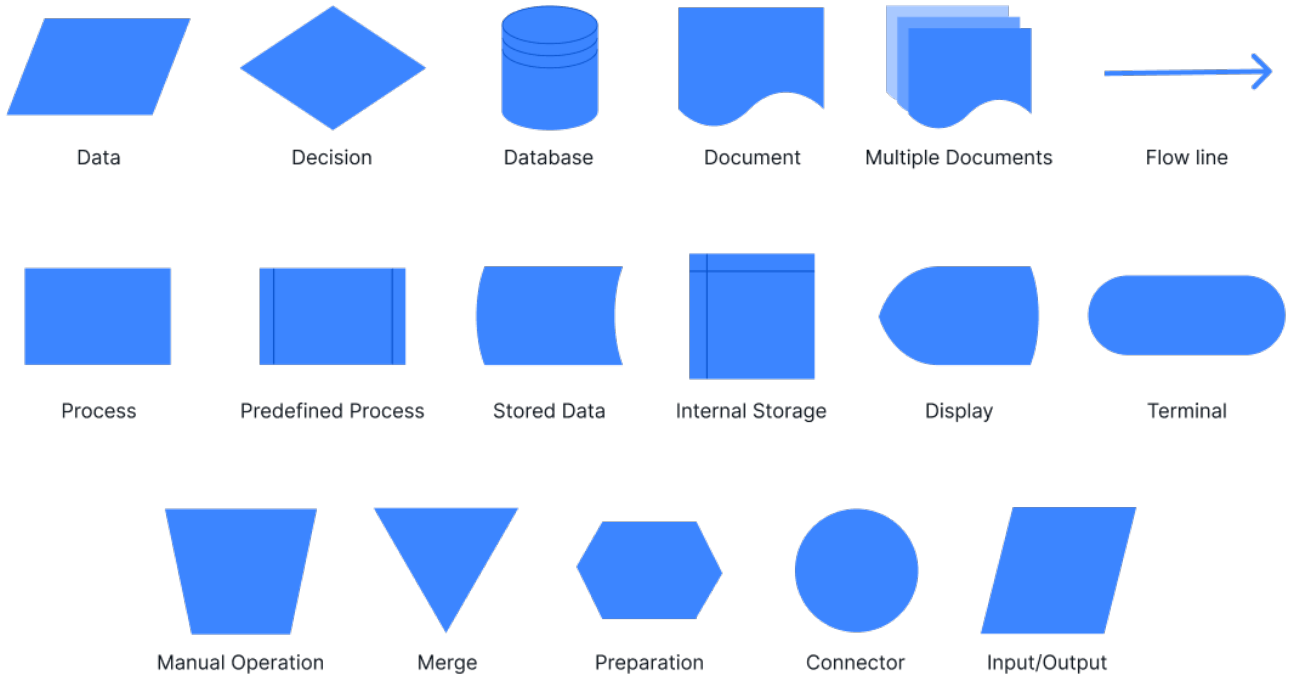
From: https://www.geeksforgeeks.org/introduction-to-algorithms/

Re-Taking FlowChart (Short Review)

- A flowchart is a diagram that depicts a process, system or computer algorithm.



Well-known symbols:



From Space Complexity to Big O Notation

- The space complexity of an algorithm or a computer program is the amount of memory space required to solve an instance of the computational problem as a function of characteristics of the input. It is the memory required by an algorithm until it executes completely.
- Similar to time complexity, space complexity is often expressed asymptotically in big O notation, such $O(n)$, $O(n\log n)$, $O(n^{\alpha})$, $O(2^{n})$, etc., where n is a characteristic of the input influencing space complexity.
  - Analogously to time complexity classes DTIME(f(n)) and NTIME(f(n)), the complexity classes DSPACE(f(n)) and NSPACE(f(n)) are the sets of languages that are decidable by deterministic (respectively, non-deterministic) Turing machines that use O(f(n)) space.
  - The complexity classes PSPACE and NPSPACE allow f to be any polynomial, analogously to P and NP. That is,

$$\mathrm{PSPACE} = \bigcup_{c \in \mathbb{Z}^+} \mathrm{DSPACE}(n^c)$$

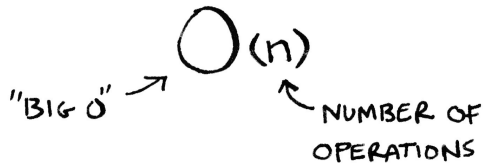
and

$$\mathrm{NPSPACE} = \bigcup_{c \in \mathbb{Z}^+} \mathrm{NSPACE}(n^c)$$

From: https://en.wikipedia.org/wiki/Space_complexity

A little about the Big O Notation

- Big O notation is special notation that tells you how fast an algorithm is.



- Big O doesn't tell you the speed in seconds. Big O notation lets you compare the number of operations. It tells you how fast the algorithm grows.
- This tell you the number of operations an algorithm will make. It's called Big O notation because you put a « big O » in front of the number of operations.
- **Also, Big O notation is called as Bachmann–Landau notation** or **asymptotic notation**.

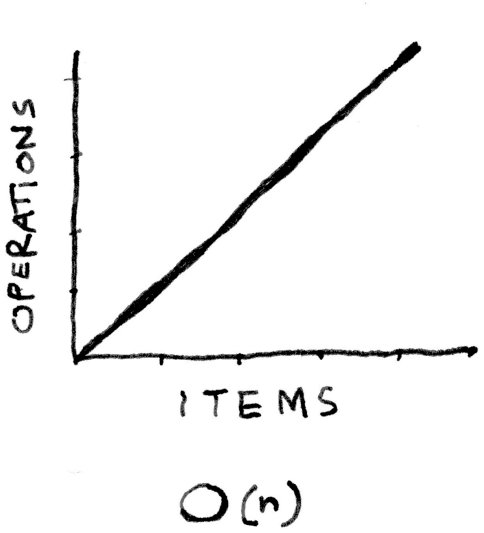
From: https://github.com/egonSchiele/grokking_algorithms

https://en.wikipedia.org/wiki/Big_O_notation

Running Time

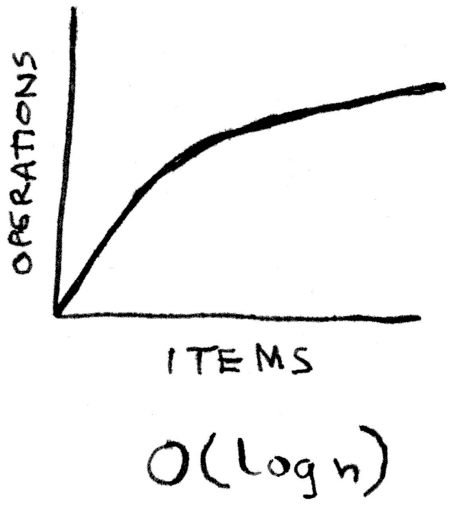
- It’s the running phase of an algorithm

Linear Time



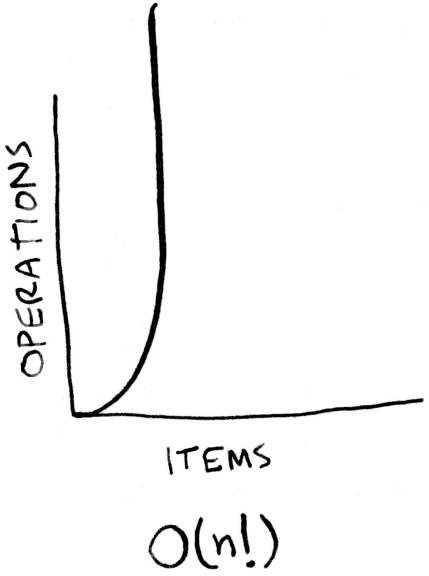
An algorithm is said to take **linear time**, or *O*(*n*) time, if its time complexity is *O*(*n*). Informally, this means that the running time increases at most linearly with the size of the input.

Logarithmic Time



An algorithm is said to take **logarithmic time** when *T*(*n*) = ***O*(log *n*).**
Algorithms taking logarithmic time are commonly found in operations on binary trees or when using binary search.

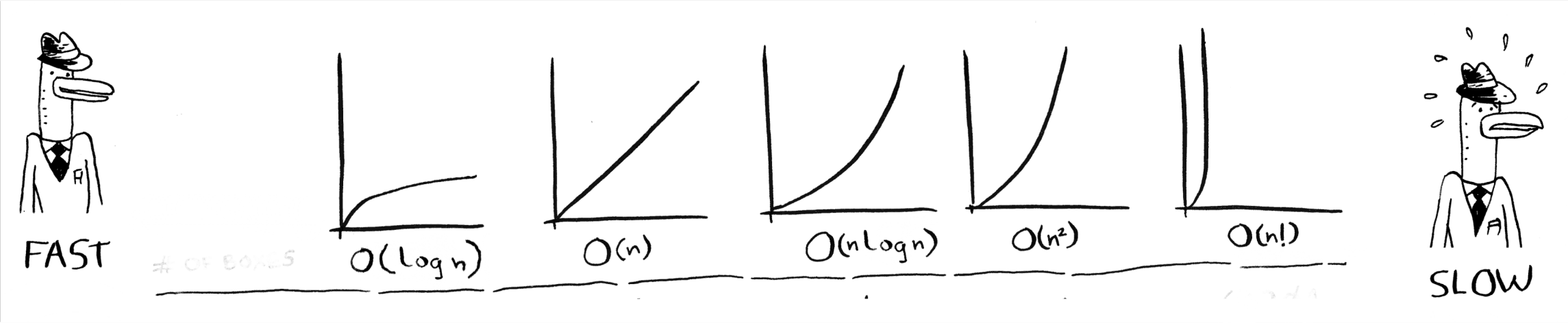
Factorial Time



Recall that a **factorial is the product of the sequence of n integers**. For example, the factorial of 5, or 5!, is: 5 * 4 * 3 * 2 * 1 = 120. We will find ourselves writing algorithms with factorial time complexity when calculating permutations and combinations.

From: https://en.wikipedia.org/wiki/Time_complexity

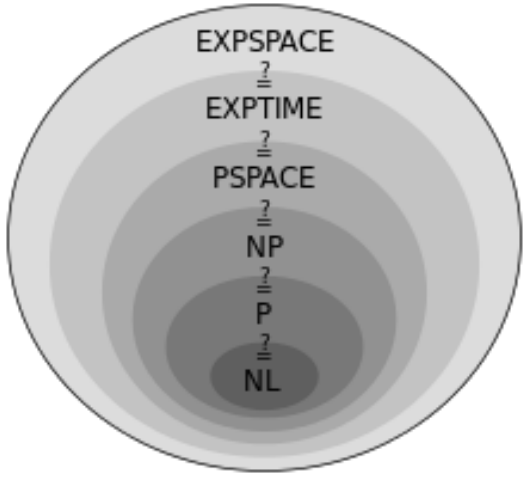
From Fast to Slow Algorithms…



- **O(log n**) also kwnon as log time . Example: Binary Search
- **O(n)** also known as linear time. Example: Simple Search
- **O(n log n)**. Example: a fast sorting algorithm like Quicksort
- **O(n^2)**. Example: a slow sorting algorithm, like Selection Sort
- **O(n!)**. Example a really slow algorithm , like the traveling salesperson.

From: https://github.com/egonSchiele/grokking_algorithms

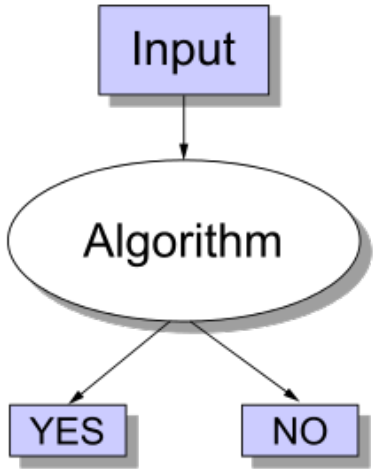
And a Little more of Complexity



A representation of the relationships between several important complexity classes

- A **complexity class** is a set of computational problems of related resource-based complexity. The two most commonly analyzed resources are time and memory.

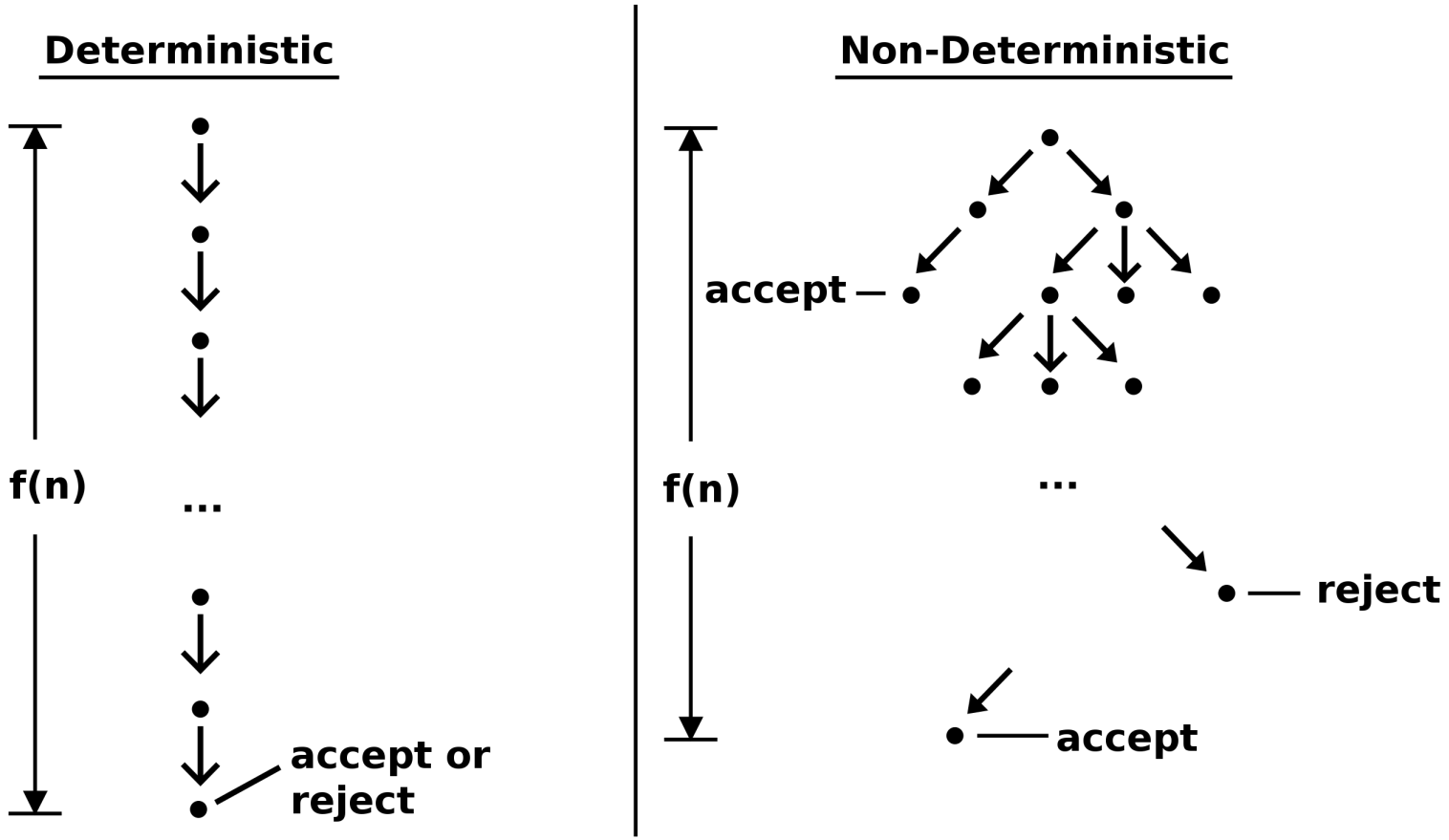
- A complexity class is defined in terms of a type of computational problem, a model of computation, and a bounded resource like time or memory.
- Complexity classes consist of decision problems that are solvable with a Turing machine, and are differentiated by their time or space (memory) requirements.
- The class **P** is the set of decision problems solvable by a deterministic Turing machine in polynomial time.
- **NP** is the class of problems that are solvable by a nondeterministic Turing machine in polynomial time.
- Many complexity classes defined in terms of other types of problems (e.g. counting problems and function problems) and using other models of computation (e.g. probabilistic Turing machines, interactive proof systems, Boolean circuits, and quantum computers).



The Decision Problem

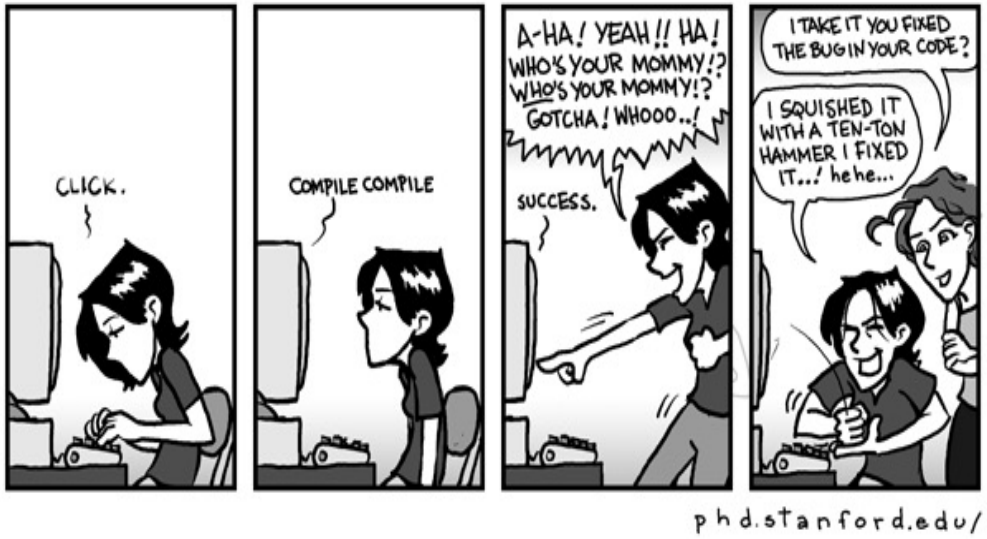
From: https://en.wikipedia.org/wiki/Complexity_class#P_and_NP

Deterministic and Non-Deterministic Turing Machines



So, why it is important to know all of this?

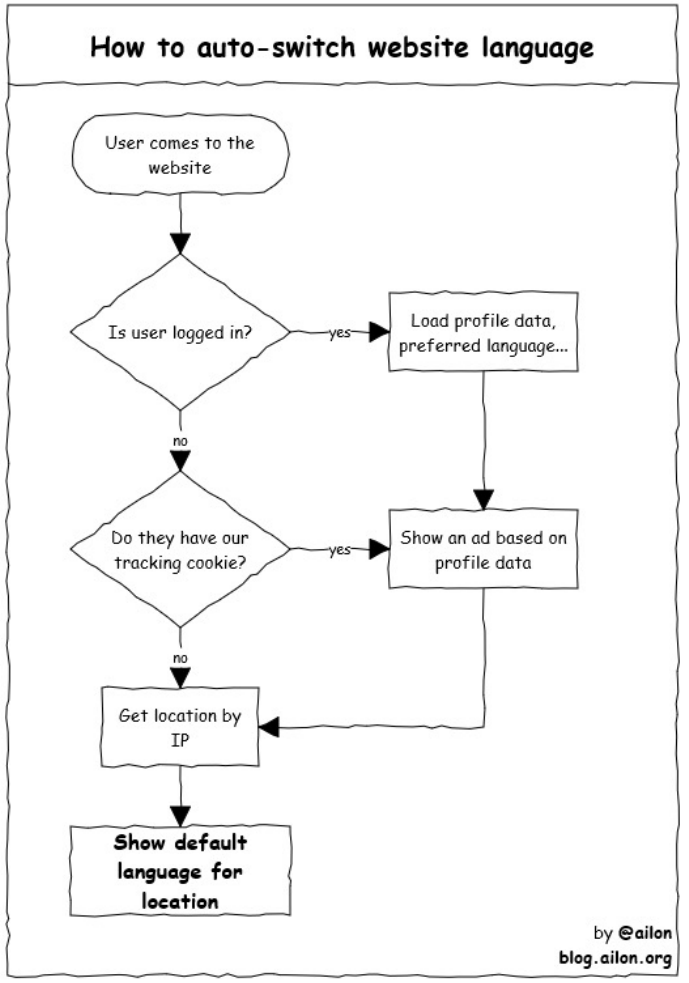
- Because you can decide how to attack a physical problem using computation
  - Selecting type of algorithm and possible design of the treatment from the mathematical representation (Remember the Big O)
  - Selecting the language and optimisation possibilities. (Or interepretators as Python)
  - Selecting the kind of computer to use (computer architecture characteristicis)
    - Classical Von-Newman Computer
    - Non Von-Newman Computer (As a Quantum Computer)
    - Variations and Hybrid Computer (i.e. using multiple processors : CPUs, GPUs, XPUs, DPUs, ASICs, etc.)



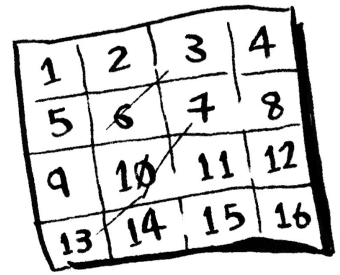
  - Selectiing Programming Paradigms (Sequential, Parallel Computing (Shared Memory, Distributed Memory, Hybrid Memory))
- Because Big Problems need Smart Solutions

Now, time to work in Class (In teams)

1. The Simple Daily Problem
   - Propose an algorithm (flowchart and pseudocode) for a simple daily task (i.e. walk to the classroom from the door of the building to your desktop, send a message by whatsapp…)
     - Try to Analize complexity and other characteristics (i.e. Number of steps, possible Big O, class of complexity)



2. Visualizing different Big O run times

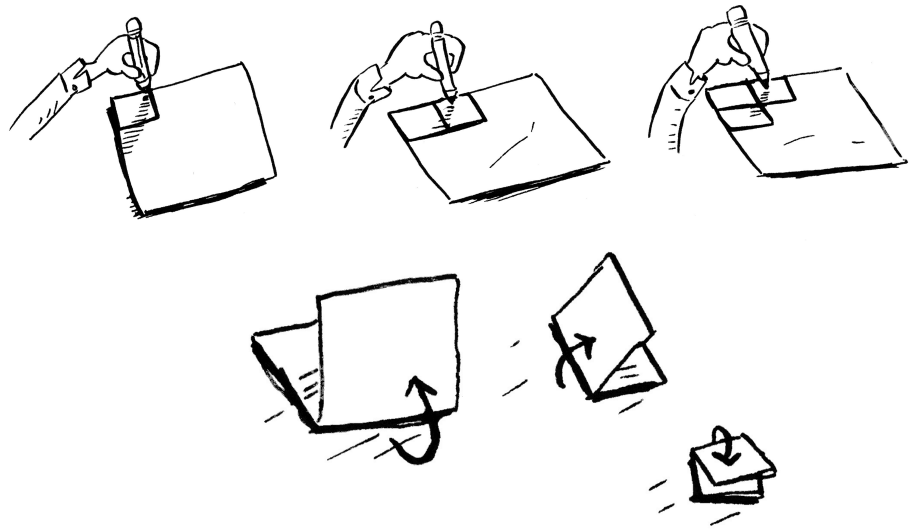


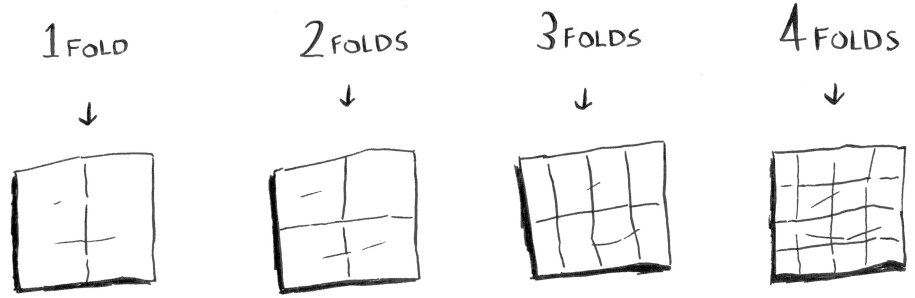
Take a Piece of paper and a pencil. Suppose you have to draw a grid of 16 boxes.

You have the possibility of two algorithms:

- **Algorithm 1:** Draw one box at time. How many operations will it take, drawing one box at time?
- **Algorithm 2:** Fold the paper, again and again, and again. Unfold it after four folds. How many operations will it tak?

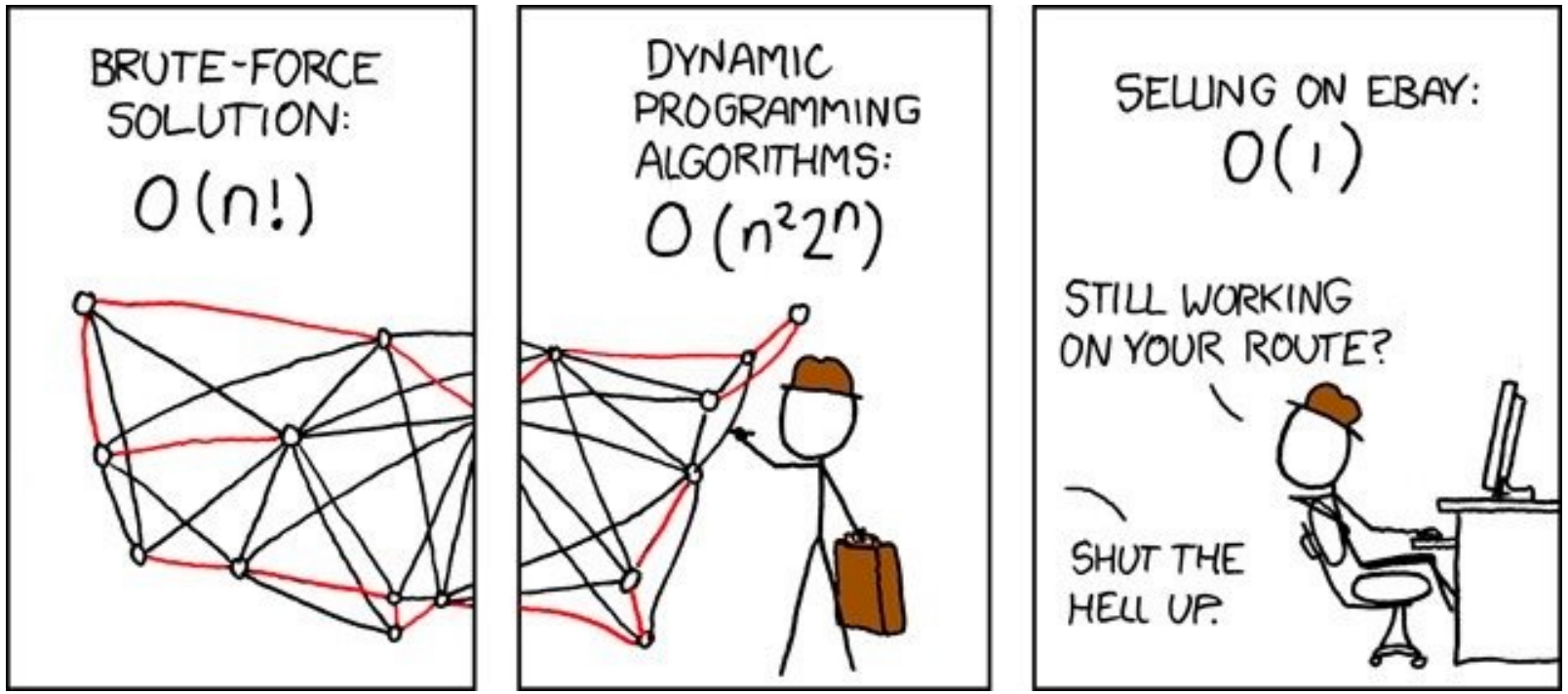
- **Taking the Big O notation, what algorithm is linear and what is logarithmic?**





From: https://github.com/egonSchiele/grokking_algorithms

Thanks



@carlosjaimebh