



## Algorithm

- *Algorithm is a step-by-step procedure, which defines a set of instructions to be executed in a certain order to get the desired output*
- *Programming language Independent*
- *Its just a set of instructions defined in English words to get the desired output*
- *Can be represented as flowchart or pseudo code*



For example, you have two integers "a" and "b" and you want to find the sum of those two number. How will you solve this? One possible solution for the above problem can be:

- Take two integers as input
  - create a variable "*sum*" to store the sum of two integers
  - put the sum of those two variables in the "*sum*" variable
  - return the "*sum*" variable
- 



```
//taking two integers as input
```

```
int findSum(int a, int b)
```

```
{
```

```
    int sum; // creating the sum variable
```

```
    sum = a + b; // storing the sum of a and b
```

```
    return sum; // returning the sum variable
```

```
}
```



In the above example, you will find three things i.e. input, algorithm, and output:



In our example,

- the input is the two numbers a and b
- we are having three steps to find the sum of two numbers. So, all three steps are collectively called an algorithm to find the sum of two numbers.
- the output is the sum of two integers a and b

- **Facebook Algorithm Example:**

- *The Facebook algorithm controls the ordering and presentation of posts, so users see what is most relevant to them.*
- *Rather than publish content chronologically, posts and ads are presented based on what Facebook sees as relevant to you, the user.*

- **Instagram Story Algorithm:**

- *The Instagram story algorithm ensures that you view the latest stories from the accounts you engage with the most.*
- *You could be engaging through likes, comments, shares, saves, tagging in photos, DMs, etc.*
- *When you are just starting out with Instagram marketing, you need to pay extra attention to posting stories.*

- **Twitter Algorithm:**

- *Twitter uses an algorithm to suggest Topics based on what it thinks someone likes.*
- *If you follow a Topic, then related Tweets, events, and ads will appear in your timeline.*
- *The Topics you follow are public.*
- *You can also tell Twitter you're not interested in a Topic*

It must have been fate that brought us together

No, just an algorithm



## ***What do you mean by a good Algorithm?***

*There can be many algorithms for a particular problem. So, how will you classify an algorithm to be good and others to be bad? Let's understand the properties of a good algorithm:*

- ***Correctness***

- *An algorithm is said to be correct if for every set of input, it should result the correct output.*
- *If you are not getting the correct output for any particular set of input, then your algorithm is wrong.*

- ***Finiteness***

- *The algorithm must always terminate after a finite number of steps.*
- *For example, in the case of recursion and loop, your algorithm should terminate otherwise you will end up having a stack overflow and infinite loop scenario respectively.*

- ***Efficiency***

- *An algorithm should efficiently*
  - *In using the resources available to the system.*
  - *the time taken to generate an output corresponding to a particular input should be as less as possible.*
  - *the memory used by the algorithm should also be as less as possible*

*So, we have seen the three factors that can be used to evaluate an algorithm. Out of these three factors, the most important one is the **efficiency of algorithms**.*

*The efficiency of an algorithm is mainly defined by two factors:*

- **Space**
- **Time**

*There can be many algorithms for a particular problem, you can pick the algorithm which is taking less time and less space*

*So, we need to analyze the algorithms, to find the best fit for our problems.*

*There are two fundamental parameters based on which we can analysis the algorithm:  
**Space Complexity and Time Complexity***



## Space Complexity

- *Space Complexity of an algorithm denotes the total space used or needed by the algorithm for its working, for various input sizes.*
- **For example:**

```
int a[] = new int[n];  
for(int i = 0; i < n; i++){  
    a[i] = sc.nextInt();  
}
```

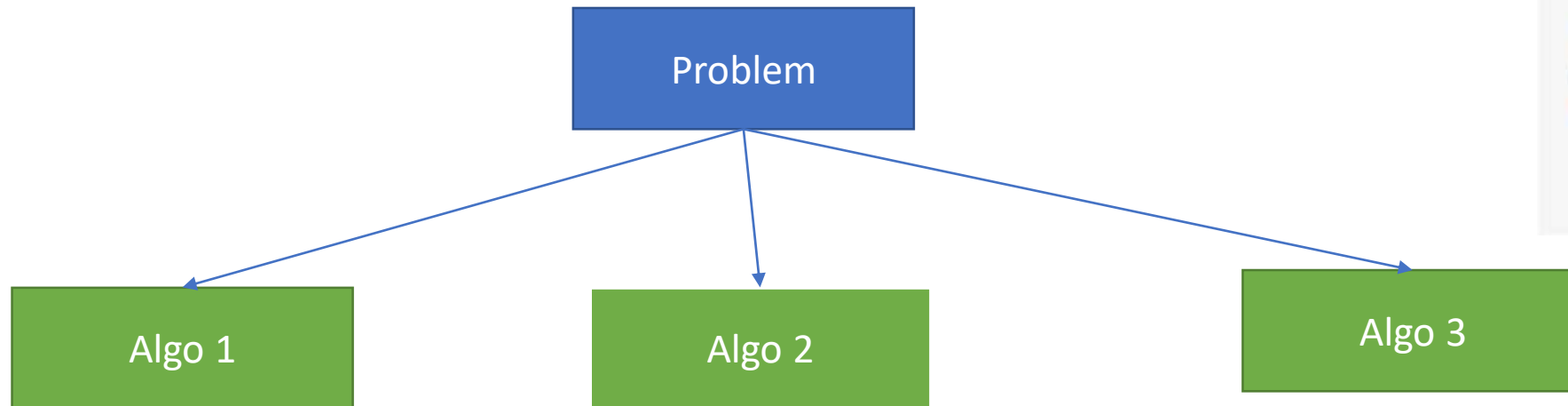
- *In the above example, we are creating an array of **size n**.*
- *So, the space complexity of the above code is in the **order of "n"** i.e., if n will increase, the space requirement will also increase accordingly.*

*Even when you are creating a variable then you need some space for your algorithm to run. All the space required for the algorithm is collectively called the Space Complexity of the algorithm.*

## Time Complexity

*Time complexity denotes the amount of time that required for the algorithm to complete the task with respect to the **input size**.*

*The time taken by an algorithm also depends on the computing speed of the system that you are using, but we ignore those external factors, and we are only concerned on the number of times a particular statement is being executed with respect to the input size.*



*Let me choose the one which takes the less execution time.*

*How can I find which is taking a less time?*

*Some approaches:*

- *Can we just run all the three algorithms on three different computers, provide same input and find the time taken by all the three algorithms and choose the one that is taking the least amount of time.*
- *Is it ok?*
  - *No, all the systems might be using some different processors.*
  - *So, the processing speed might vary.*
  - *So, we can't use this approach to find the most efficient algorithm.*

*Another approach, can we run all the three algorithms on the same computer and try to find the time taken by the algorithm and choose the best.*

- *But here also, you might get wrong results because, at the time of execution of a program, there are other things that are executing along with your program, so you might get the wrong time.*

*So, we have seen that we can't judge an algorithm by calculating the time taken during its execution in a particular system.*

*We need some standard way to analyze the algorithm.*

*We use **Asymptotic notation** to analyze any algorithm and based on that we find the most efficient algorithm.*

*Here in Asymptotic notation,*

- *We do not consider the system configuration,*
- *Rather we consider the order of growth of the input.*
- ***We try to find how the time, or the space taken by the algorithm will increase/decrease after increasing/decreasing the input size***

*There are three asymptotic notations that are used to represent the time complexity of an algorithm.*

*They are:*

- ***$\Theta$  Notation (theta)***
- ***Big O Notation***
- ***$\Omega$  Notation***

Before learning about these three asymptotic notation, we should learn about the **best, average, and the worst case** of an algorithm.

An algorithm can have different time for different inputs. It may take 1 second for some input and 10 seconds for some other input.

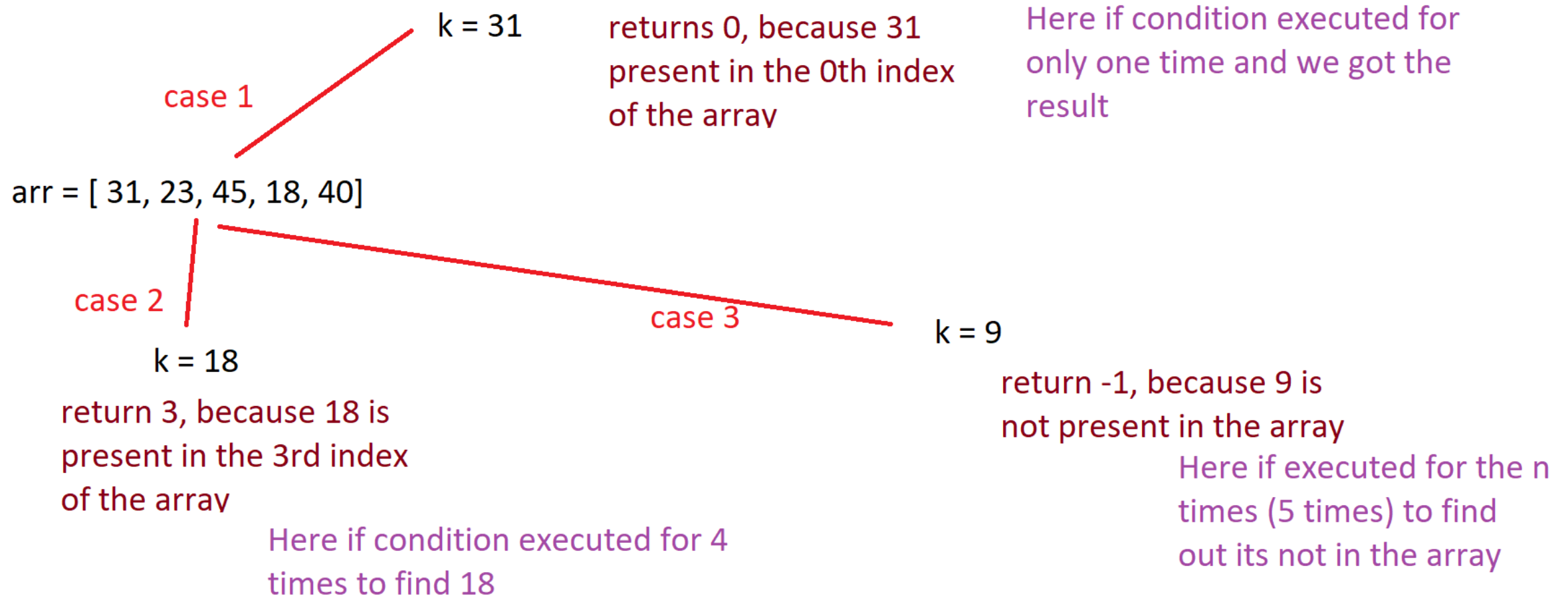
**For example:** We have one array named "arr" and an integer "k". we need to find if that integer "k" is present in the array "arr" or not? If the integer is there, then return 1 other return 0. Try to make an algorithm for this question.

- **Input** - integer array of size 'n' and integer to search 'k'
- **Output** - If the element "k" is found in the array, then we have return 1, otherwise we have to return 0.

Now, one possible solution for the above problem can be **linear search** i.e.

- we will traverse each and every element of the array and compare that element with "k".
- If it is equal to "k" then return 1, otherwise, keep on comparing for more elements in the array and if you reach at the end of the array and you did not find any element, then return 0.

```
/*  
* arr: integer array  
* n: integer (size of integer array)  
* k: integer (integer to be searched)  
*/  
int searchK(int arr[], int n, int k)  
{  
    // for-loop to iterate with each element in the array  
    for (int i = 0; i < n; ++i)  
    {  
        // check if ith element is equal to "k" or not  
        if (arr[i] == k)  
            return 1; // return 1, if you find "k"  
    }  
    return 0; // return 0, if you didn't find "k"  
}
```



- *In the case 1, we found the element present in the array, by doing the comparison for only one time*
  - *This is the best case of the algorithm*
- *In the case 3, we found the element is not present in the array, by doing the comparison for  $n$  times*
  - *This is the worst case of the algorithm*
- *In the case 2, we found the element present in the array.*
  - *Its not the worst or best, it just the average case of the algorithm*

### ***Now, Let's define***

- ***Best Case*** - Lower Bound on the running time of the algorithm
- ***Worst Case*** - Upper Bound on the running time of the algorithm
- ***Average Case*** - running time used by the algorithm averaged over all possible inputs. We calculate the running time for all possible inputs, sum all the calculated values and divide the sum by the total number of inputs.



*So, we learned about the best, average, and worst case of an algorithm.*

*Now, let's get back to the asymptotic notation where we saw that we use three asymptotic notation to represent the complexity of an algorithm i.e.,  $\Theta$  Notation (theta),  $\Omega$  Notation, Big O Notation.*

**NOTE:** *In the asymptotic analysis, we generally deal with large input size.*

## **Big O Notation**

*Big O notation denotes the upper bound of an algorithm i.e., longest time an algorithm can take for its execution, i.e., it is used for measuring the worst-case time complexity of an algorithm.*

## **$\Omega$ Notation (omega)**

*The  $\Omega$  notation denotes the lower bound of an algorithm i.e., shortest time an algorithm can take for its execution, i.e., it is used for measuring the best-case time complexity of an algorithm.*

## **$\Theta$ Notation (theta)**

*The  $\Theta$  Notation is used to find the average bound of an algorithm i.e., it defines an upper bound and a lower bound, and your algorithm will lie in between these levels.*

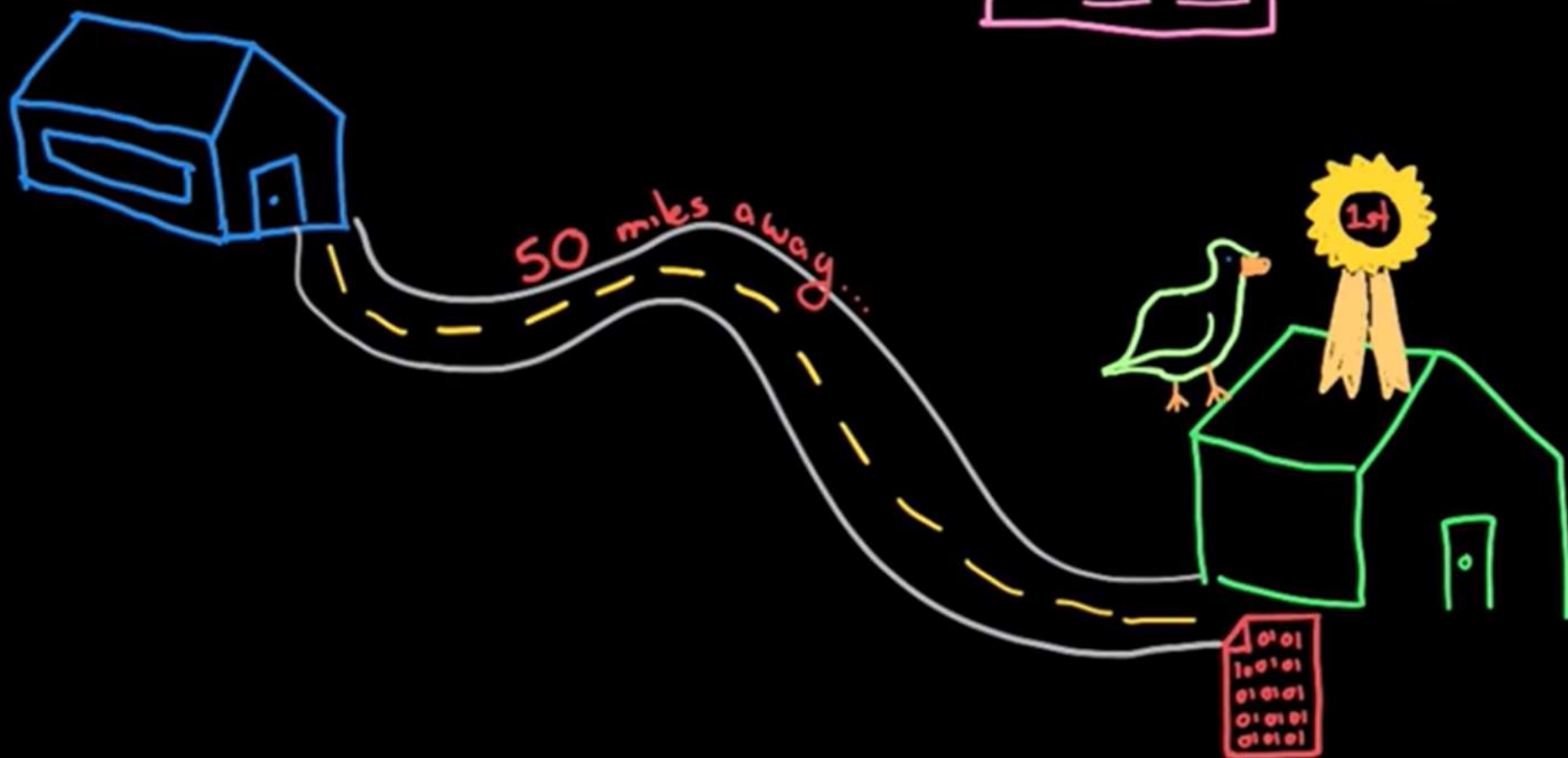
2009...



50 miles away...



2009...



# 2009...



## internet transfer

1 GB  $\rightarrow$  30 min

2 GB  $\rightarrow$  60 min

3 GB  $\rightarrow$  90 min

1000 GB  $\rightarrow$  500 hours



# 2009...



## internet transfer

1 GB  $\rightarrow$  30 min  
2 GB  $\rightarrow$  60 min  
3 GB  $\rightarrow$  90 min  
1000 GB  $\rightarrow$  500 hours (!)

## pigeon transfer (50 miles)

1 GB  $\rightarrow$  60 min  
2 GB  $\rightarrow$  60 min  
3 GB  $\rightarrow$  60 min  
1000 GB  $\rightarrow$  60 min (!)

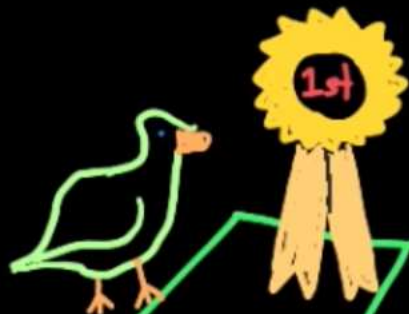
# 2009...



50 miles away...

## internet transfer

1 GB  $\rightarrow$  30 min  
2 GB  $\rightarrow$  60 min  
3 GB  $\rightarrow$  90 min  
1000 GB  $\rightarrow$  500 hours (!)



## pigeon transfer (50 miles)

1 GB  $\rightarrow$  60 min  
2 GB  $\rightarrow$  60 min  
3 GB  $\rightarrow$  60 min  
1000 GB  $\rightarrow$  60 min (!)

CONSTANT TIME

# 2009...



Constant time beats  
linear **IF** data is  
sufficiently big



50 miles away...



pigeon transfer (50 miles)

1 GB  $\rightarrow$  60 min

2 GB  $\rightarrow$  60 min

3 GB  $\rightarrow$  60 min

1000 GB  $\rightarrow$  60 min (!)

CONSTANT TIME

internet transfer

1 GB  $\rightarrow$  30 min

2 GB  $\rightarrow$  60 min

3 GB  $\rightarrow$  90 min

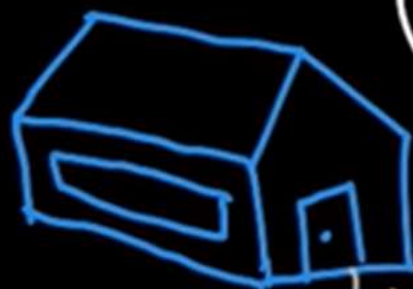
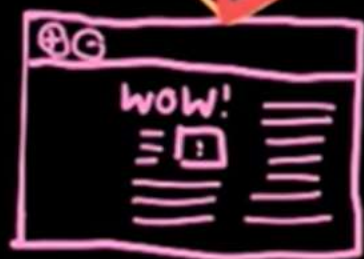
1000 GB  $\rightarrow$  500 hours (!)

LINEAR  
TIME

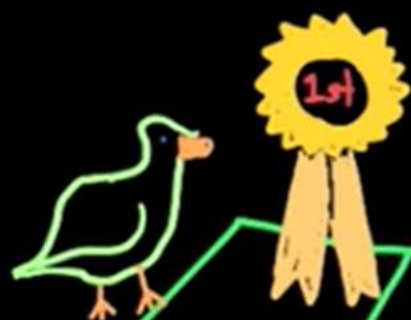


# 2009...

Constant time beats  
linear **IF** data is  
sufficiently big



50 miles away...



pigeon transfer (50 miles)

1 GB  $\rightarrow$  60 min

2 GB  $\rightarrow$  60 min

3 GB  $\rightarrow$  60 min

1000 GB  $\rightarrow$  60 min (!)

internet transfer

1 GB  $\rightarrow$  30 min

2 GB  $\rightarrow$  60 min

3 GB  $\rightarrow$  90 min

1000 GB  $\rightarrow$  500 hours (!)

LINEAR  
TIME

$O(N)$

where  $N$  = amount of data

CONSTANT TIME

$O(1)$

### Example 1: Finding the sum of the first n numbers

For example,

- if  $n = 4$ , then our output should be  $1 + 2 + 3 + 4 = 10$ .
- If  $n = 5$ , then the output should be  $1 + 2 + 3 + 4 + 5 = 15$

There are 2 approaches to find the sum

- One is using sum of n natural number formula i.e  $\text{sum} = n * (n+1)/2$
- Another is using loops and calculating the sum

### Approach 1:

```
// function taking input "n"
int findSum(int n)
{
    return n * (n+1) / 2; // this will take some constant time c1
}
```

In the above code, there is only one statement, and we know that a statement takes constant time for its execution.

It will take the same amount of time for all the input size, and we denote this as  **$O(1)$**

## Approach 2:

```
// function taking input "n"
int findSum(int n)
{
    int sum = 0;
    for(int i =1; i <=n; i++){
        sum = sum + i;
    }
    return sum;
}
```

*In the above code, we will run a loop from 1 to n and we will add these values and store it in the sum*

*the input size n increase time taken also increases i.e., it depends on the input size n and we denote this as  **$O(n)$***



HOW TIME  
SCALES WITH  
RESPECT TO SOME  
INPUT VARIABLES

## 4 Rules in Big O Notation

### *Rule 1 : Different steps gets added*


```
// This loop runs for N time
for (int i = 0; i < N; i++) {
    a = a + 5;
}
// This loop runs for M time
for (int i = 0; i < M; i++) {
    b = b + 10;
}
```



$O(N+M)$

## Rule 2: Drop Constants

```
for(int i = 0; i < n; i++){  
    if(a[i] < min)  
        min = a[i];  
}  
for(int i = 0; i < n; i++){  
    if(a[i] > max)  
        max = a[i];  
}
```


$$O(N+N) = O(2N) = O(N)$$



*Rule 3: Different Inputs has different variables*

```
for(int i = 0; i < n; i++){  
    for(int j = 0; j < m; j++){  
        System.out.println(a[i][j]);  
    }  
}
```



$O(N * M)$


```
for(int i = 0; i < n; i++){  
    for(int j = 0; j < n; j++){  
        System.out.println(a[i][j]);  
    }  
}
```



$O(N * N) = O(N^2)$

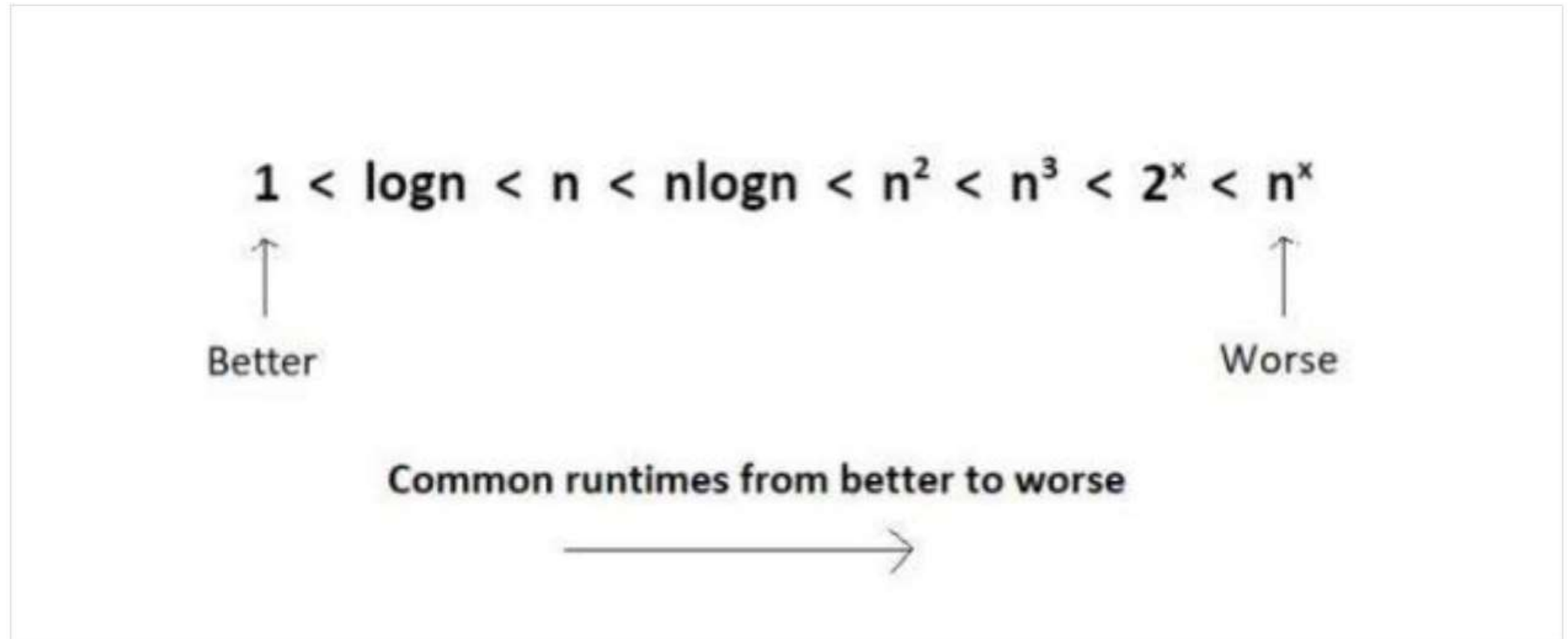
*Rule 4: Drop non dominant terms*

```
for(int i = 0; i < n; i++){  
    for(int j = 0; j < n; j++){  
        System.out.println(a[i][j]);  
    }  
}  
for(int i = 0; i < n; i++){  
    if(a[i] > max)  
        max = a[i];  
}
```

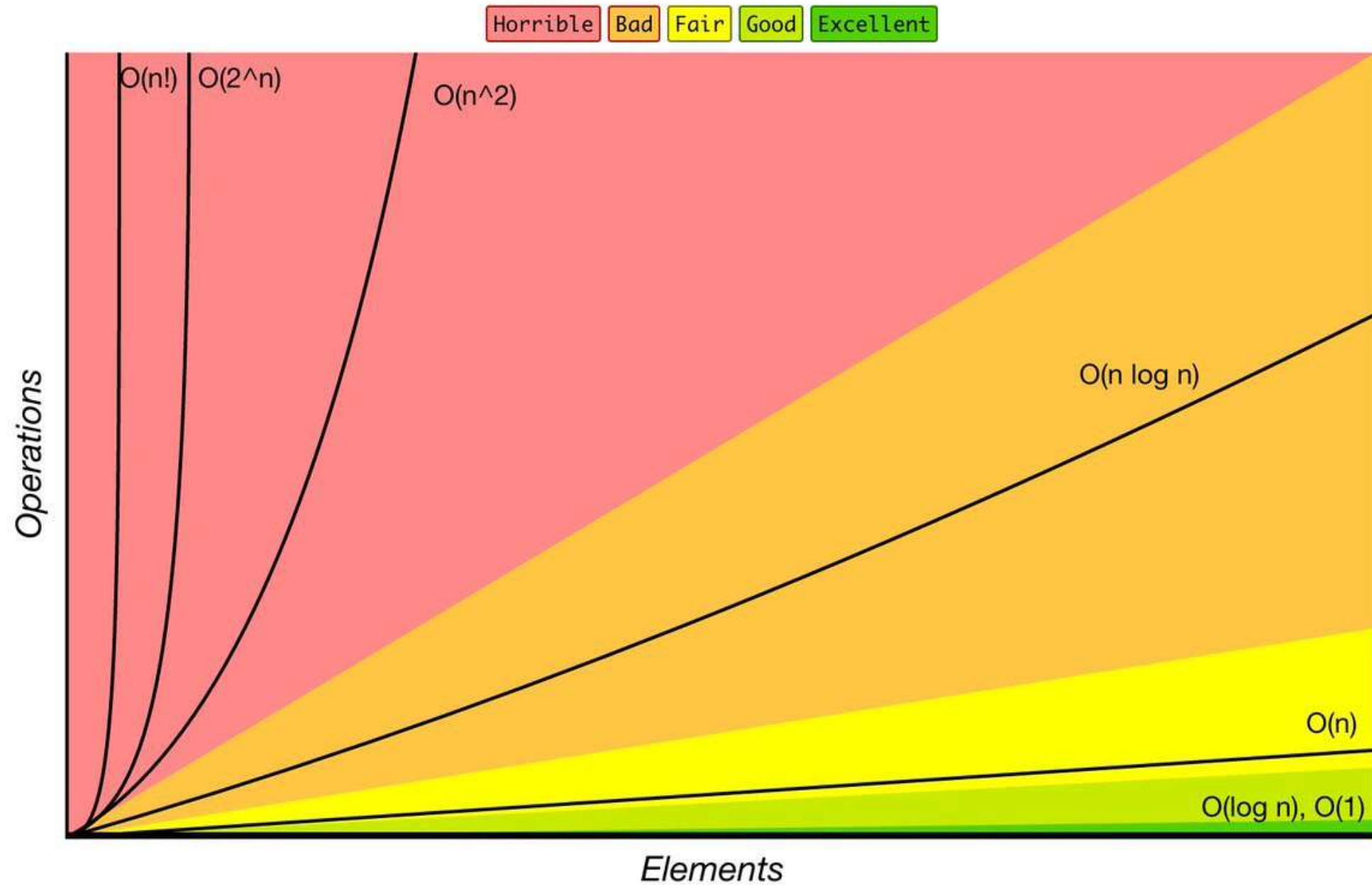

$$O(N^2 + N) = O(N^2)$$



Increasing order of common runtimes



# Big-O Complexity Chart



## **Advantages of an Algorithm**

- **Effective Communication:** Since algorithm is written in English like language, it is simple to understand step-by-step solution of the problems.
- **Easy Debugging:** Well-designed algorithm makes debugging easy so that we can identify logical errors in the program.
- **Easy and Efficient Coding:** An algorithm acts as a blueprint of a program and helps during program development.
- **Independent of Programming Language:** An algorithm is independent of programming languages and can be easily coded using any high-level language.

## **Disadvantages of an Algorithm**

- Developing algorithms for complex problems would be time consuming and difficult to understand.
- Understanding complex logic through algorithms can be exceedingly difficult.

## Types Of Algorithms:

- Brute force algorithms.
- Simple recursive algorithms.
- Divide and conquer algorithms.
- Backtracking algorithms.
- Dynamic programming algorithms.
- Greedy algorithms.
- Branch and bound algorithms.

## **Brute Force Algorithm**

- *Straight forward approach for solving problems*
- *Example:*
  - *Simple brute force attacks: hackers attempt to logically guess your credentials completely unassisted from software tools or other means.*
  - *These can reveal extremely simple passwords and PINs.*
  - *For example, a password that is set as “guest12345”*
- **Pros** - *The brute force approach is a guaranteed way to find the correct solution by listing all the possible solutions for the problem.*
- **Cons** - *The brute force approach is inefficient. For real-time problems, algorithm analysis often goes above the  $O(N!)$  order of growth.*