Complexity

General Guideline

Session Plan

The complexity will cover following topics:-

- What are various characteristics of algorithms?
- What are various steps to solve a problem?
- What is time space tradeoff?
- Asymptotic Notations

Module Objective

- Complexity analysis allows us to measure how fast a program is when it performs computations.

Complexity

Representation of a Problem

- The problem can be written in a story/case study form or in technical terms.

Example:

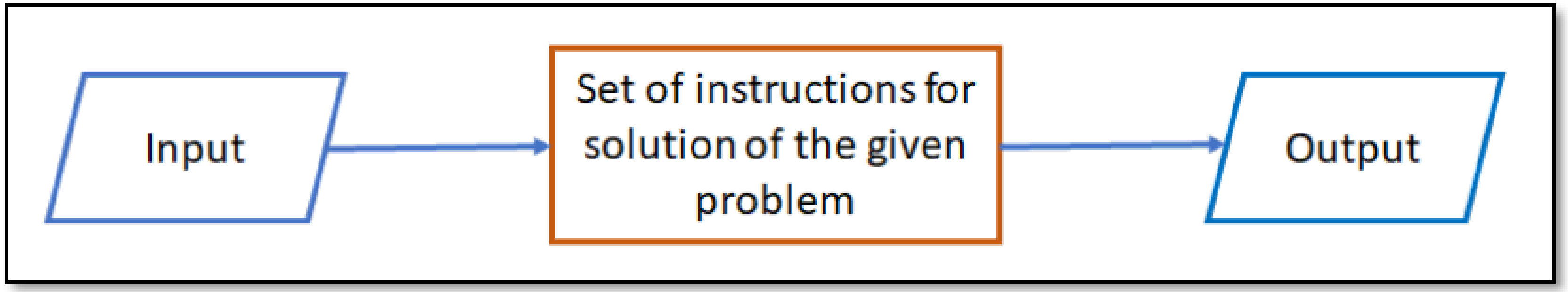
In story/Case study form: The usual chess board consists of 8x8 board with white and black chess pieces. There is one white Queen and one black. The queen can attack in horizontal, vertical and diagonal directions. Consider a customized chess board of size 4x4 with 4 Queens. The problem is to place these queens such that no one is in the attacking position.

	Q		
			Q
Q			
		Q	

In Technical Term: Given an array, write an algorithm to reverse the array elements.

Algorithm

An algorithm is a finite set of steps to solve a problem.



Characteristics of an Algorithm

1- Definiteness

2- Finiteness

3- Input

4- Output

5. Feasible

6. Language Independent

Characteristics of an Algorithm

1- Definiteness

Every instruction in the Algorithm must have the clear meaning without any ambiguity.

2- Finiteness

Every instruction in the Algorithm should terminate in finite amount of time.

```
e.g.    i=1;
   WHILE (1) DO
        i=i+1
```

Characteristics of an Algorithm

3- Input

Every instruction must accept well defined inputs. An algorithm may contain 0 input as well.

4- Output

The Algorithm is designed for performing a specific task. Hence the algorithm should generate some output (at least 1)

Characteristics of an Algorithm

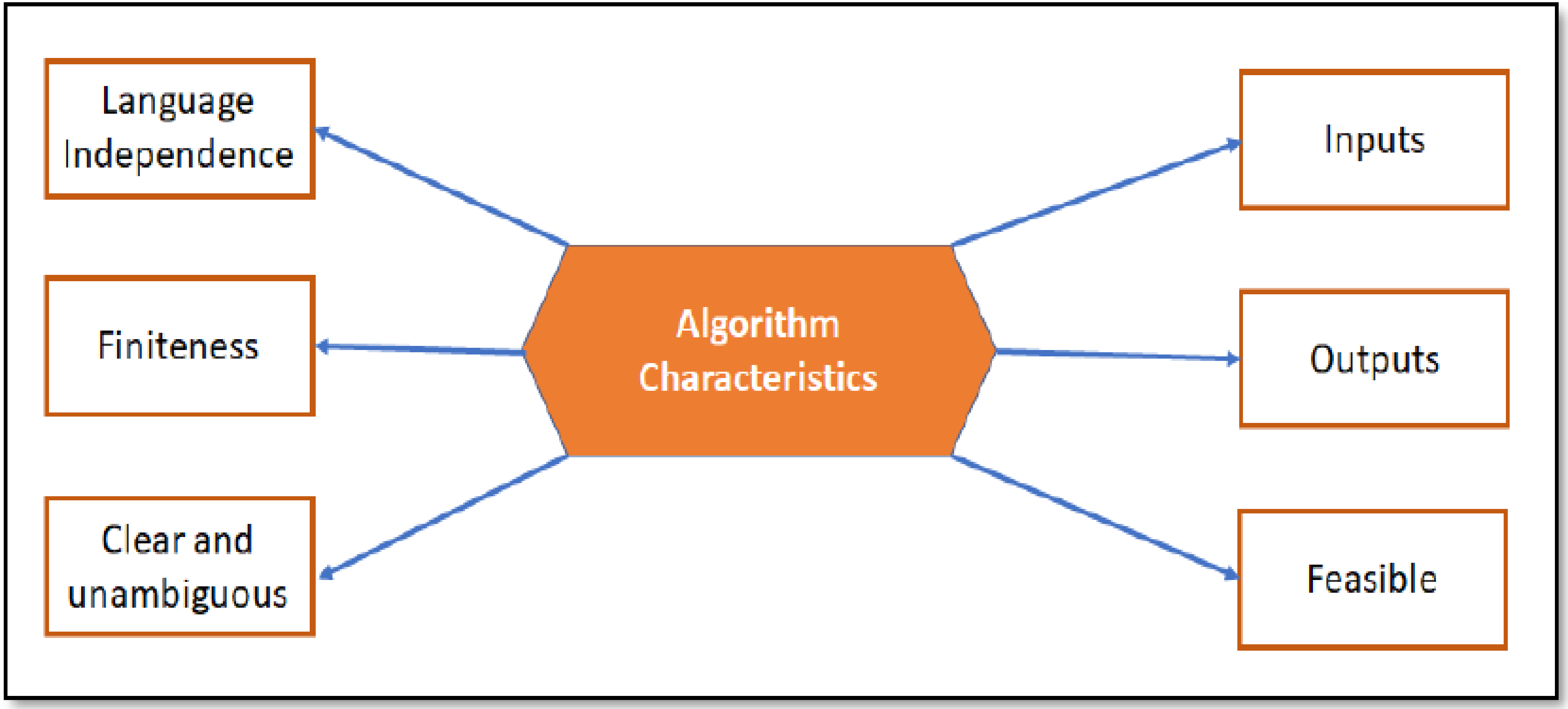
5. Feasible

The Algorithm must be simple, generic and practical, such that it can be executed upon with the available resources.

6. Language Independent

The algorithm should be written free from any programming language. It should be general which can be implemented in any programming language.

Characteristics of an Algorithm



Algorithm Analysis

Analysis of efficiency of an algorithm can be performed at two different stages, before implementation and after implementation, as

A priori analysis − This is defined as theoretical analysis of an algorithm. Efficiency of algorithm is measured by assuming that all other factors e.g. speed of processor, are constant and have no effect on implementation.

A posterior analysis − This is defined as empirical analysis of an algorithm. The chosen algorithm is implemented using programming language. Next the chosen algorithm is executed on target computer machine. In this analysis, actual statistics like running time and space needed are collected.

Algorithm analysis is dealt with the execution or running time of various operations involved. Running time of an operation can be defined as number of computer instructions executed per operation.

Generally, there is always more than one way to solve a problem in computer science with different algorithms. Therefore, it is highly required to use a method to compare the solutions in order to judge which one is more optimal. The method must be:

- **Independent of the machine and its configuration, on which the algorithm is running on.**
- **Shows a direct correlation with the number of inputs.**
- **Can distinguish two algorithms clearly without ambiguity.**

Suppose X is treated as an algorithm and N is treated as the size of input data, the time and space implemented by the Algorithm X are the two main factors which determine the efficiency of X.

Time Factor − The time is calculated or measured by counting the number of key operations such as comparisons in sorting algorithm.

Space Factor − The space is calculated or measured by counting the maximum memory space required by the algorithm.

The complexity of an algorithm f(N) provides the running time and / or storage space needed by the algorithm with respect of N as the size of input data.

Space Complexity

Space complexity of an algorithm represents the amount of memory space needed the algorithm in its life cycle.

Space needed by an algorithm is equal to the sum of the following two components

A fixed part that is a space required to store certain data and variables (i.e. simple variables and constants, program size etc.), that are not dependent of the size of the problem.

A variable part is a space required by variables, whose size is totally dependent on the size of the problem. For example, recursion stack space, dynamic memory allocation etc.

Example

Algorithm

SUM(P, Q)
Step 1 - START
Step 2 - R $\leftarrow$ P + Q + 10

Step 3 - Stop

Here we have three variables P, Q and R and one constant. Hence S(p) = 1+3. Now space is dependent on data types of given constant types and variables and it will be multiplied accordingly.

Time Complexity

Time Complexity of an algorithm is the representation of the **amount of time required by the algorithm to execute to completion**. Time requirements can be denoted or defined as a numerical function t(N), where t(N) can be measured as the number of steps, provided each step takes constant time.

Asymptotic Notations

Asymptotic notations are used to represent the running time for an algorithm.
the time required by an algorithm falls under three types −

- **Best Case** − Minimum time required for program execution.
- **Average Case** − Average time required for program execution.
- **Worst Case** − Maximum time required for program execution.

"In asymptotic notations, we derive the complexity concerning the size of the input. (Example in terms of n)"
"These notations are important because without expanding the cost of running the algorithm, we can estimate the complexity of the algorithms."

Following are the commonly used asymptotic notations to calculate the running time complexity of an algorithm.

- O Notation
- Ω Notation
- θ Notation

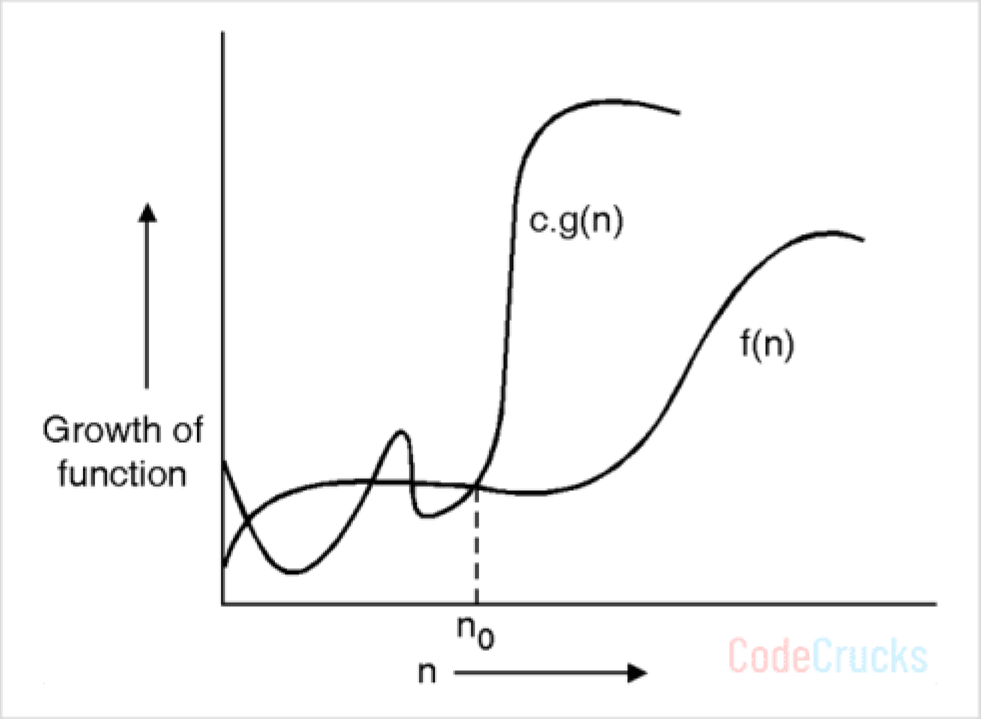
Big-oh notation

The notation O(n) is the formal way to express the upper bound of an algorithm's running time. It measures the worst case time complexity or the longest amount of time an algorithm can possibly take to complete.

Let f(n) and g(n) are two nonnegative functions indicating the running time of two algorithms. We say, g(n) is upper bound of f(n) if there exist some positive constants c and n_0 such that

$0 \le f(n) \le c.g(n)$ for all $n \ge n_0$.

It is denoted as f(n) = O(g(n)).



Examples on Upper Bound Asymptotic Notation

Example: Find upper bound of running time of constant function $f(n) = 6993$.

To find the upper bound of $f(n)$, we have to find c and $n0$ such that $0 \leq f(n) \leq c.g(n)$ for all $n \geq n0$

$0 \leq f(n) \leq c \times g(n)$

$0 \leq 6993 \leq c \times g(n)$

$0 \leq 6993 \leq 6993 \times 1$

So, $c = 6993$ and $g(n) = 1$

Any value of c which is greater than 6993, satisfies the above inequalities, so all such values of c are possible.

$0 \leq 6993 \leq 8000 \times 1 \rightarrow$ true

$0 \leq 6993 \leq 10500 \times 1 \rightarrow$ true

Function $f(n)$ is constant, so it does not depend on problem size n. So $n0 = 1$

$f(n) = O(g(n)) = O(1)$ for $c = 6993$, $n0 = 1$

$f(n) = O(g(n)) = O(1)$ for $c = 8000$, $n0 = 1$ and so on.

Example 2

Find upper bound of running time of a linear function $f(n) = 6n + 3$.

To find upper bound of f(n), we have to find c and n_0 such that $0 \le f(n) \le c \times g(n)$ for all $n \ge n_0$

$0 \le f(n) \le c \times g(n)$

$0 \le 6n + 3 \le c \times g(n)$

$0 \le 6n + 3 \le 6n + 3n$, for all $n \ge 1$ (There can be such infinite possibilities)

$0 \le 6n + 3 \le 9n$

So, $c = 9$ and $g(n) = n$, $n_0 = 1$

Tabular Approach

Tabular Approach

$0 \leq 6n + 3 \leq c \times g(n)$

$0 \leq 6n + 3 \leq 7n$

Now, manually find out the proper n0, such that $f(n) \leq c.g(n)$

n	f(n) = 6n + 3	c.g(n) = 7n
1	9	7
2	15	14
3	**21**	**21**
4	27	28
5	33	35

From Table, for $n \geq 3$, $f(n) \leq c \times g(n)$ holds true. So, c = 7, g(n) = n and n0 = 3, There can be such multiple pair of (c, n0)

$f(n) = O(g(n)) = O(n)$ for c = 9, n0 = 1

$f(n) = O(g(n)) = O(n)$ for c = 7, n0 = 3

and so on.

Example 3

Find upper bound of running time of quadratic function f(n) = 3n2 + 2n + 4.

To find upper bound of f(n), we have to find c and n0 such that $0 \leq f(n) \leq c \times g(n)$ for all $n \geq n0$

$0 \leq f(n) \leq c \times g(n)$

$0 \leq 3n^2 + 2n + 4 \leq c \times g(n)$

$0 \leq 3n^2 + 2n + 4 \leq 3n^2 + 2n^2 + 4n^2$,

for all $n \geq 1$:

$0 \leq 3n^2 + 2n + 4 \leq 9n^2$

$0 \leq 3n^2 +2n + 4 \leq 9n^2$

So, c = 9, g(n) = n2 and n0 = 1

Tabular approach:

$0 \le 3n2 + 2n + 4 \le c.g\ (n)$

$0 \le 3n2 + 2n + 4 \le 4n2$

Now, manually find out the proper n0, such that $f(n) \le c.g(n)$

n	f(n) = 3n2 + 2n + 4	c.g (n) = 4n2
1	9	4
2	20	16
3	37	36
4	60	64
5	89	100

From Table, for $n \ge 4$, $f(n) \le c \times g\ (n)$ holds true. So, c = 4, g(n) = n2 and n0 = 4. There can be such multiple pair of (c, n0)

f(n) = O (g(n)) = O (n2) for c = 9, n0 = 1
f(n) = O (g(n)) = O (n2) for c = 4, n0 = 4
and so on.

Example

Find upper bound of running time of a cubic function $f(n) = 2n^3 + 4n + 5$

Tabular approach

0 ≤ 2n^3 + 4n + 5 ≤ c × g(n)

0 ≤ 2n^3 + 4n + 5 ≤ 3n3

Now, manually find out the proper n0, such that f(n) ≤ c × g(n)

n	f(n) = 2n3 + 4n + 5	c.g(n) = 3n^3
1	11	3
2	29	24
3	71	81
4	149	192

From Table, for n ≥ 3, f(n) ≤ c × g(n) holds true. So, c = 3, g(n) = n3 and n0 = 3.

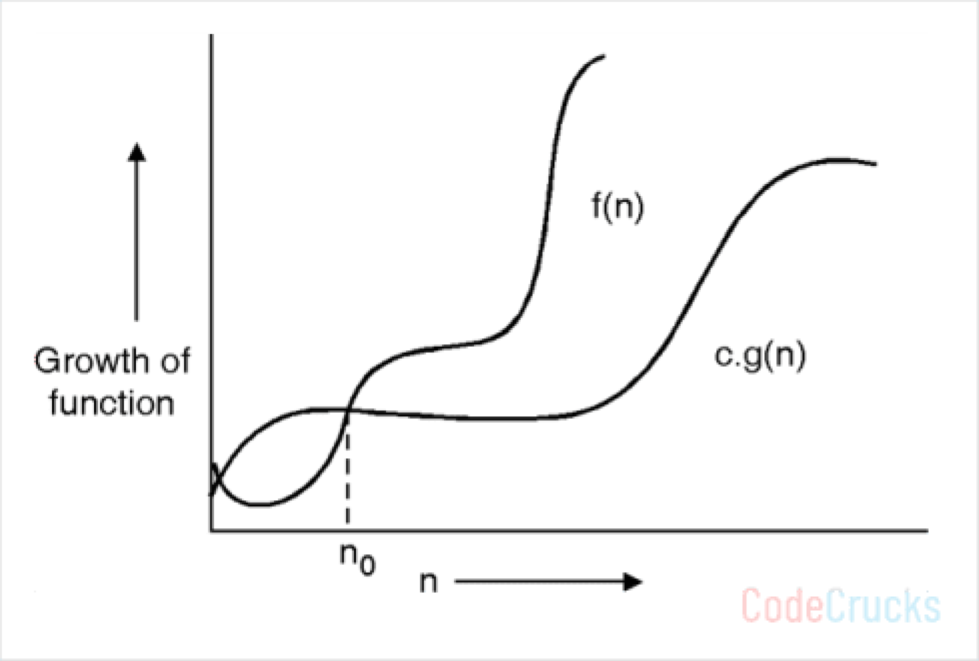
There can be such multiple pair of (c, n0)

f(n) = O(g(n)) = O(n^3) for c = 11, n0 = 1

f(n) = O(g(n)) = O(n^3) for c =3, n0 = 3 and so on.

Omega Notation, Ω

The notation Ω(n) is the formal way to express the lower bound of an algorithm's running time. It measures the best case time complexity or the best amount of time an algorithm can possibly take to complete.



The function g(n) is lower bound of function f(n) if there exist some positive constants c and n_0 such that

$0 \le c.g(n) \le f(n)$ for all $n \ge n_0$.

It is denoted as $f(n) = \Omega(g(n))$.

Examples on Lower Bound Asymptotic Notation

Find lower bound of running time of constant function $f(n) = 23$.

To find lower bound of f(n), we have to find c and n_0 such that $\{ 0 \le c \times g(n) \le f(n)$ for all $n \ge n_0 \}$

$0 \le c \times g(n) \le f(n)$

$0 \le c \times g(n) \le 23$

$0 \le 23 X 1 \le 23 \rightarrow$ true

$0 \le 12 X1 \le 23 \rightarrow$ true

$0 \le 5X1 \le 23 \rightarrow$ true

Above all three inequalities are true and there exists such infinite inequalities

So c = 23, c = 12, c = 5 and g(n) = 1. Any value of c which is less than or equals to 23, satisfies the above inequality, so all such value of c are possible. Function f(n) is constant, so it does not depend on problem size n. Hence $n_0 = 1$

$f(n) = \Omega (g(n)) = \Omega (1)$ for $c = 23$, $n_0 = 1$

$f(n) = \Omega (g(n)) = \Omega (1)$ for $c = 12$, $n_0 = 1$ and so on.

Find lower bound of running time of a linear function $f(n) = 6n + 3$.

To find lower bound of f(n), we have to find c and n_0 such that $0 \leq c.g(n) \leq f(n)$ for all $n \geq n_0$

$0 \leq c \times g(n) \leq f(n)$

$0 \leq c \times g(n) \leq 6n + 3$

$0 \leq 6n \leq 6n + 3 \rightarrow$ true, for all $n \geq n_0$

$0 \leq 5n \leq 6n + 3 \rightarrow$ true, for all $n \geq n_0$

Above both inequalities are true and there exists such infinite inequalities. So,

$f(n) = \Omega\ (g(n)) = \Omega\ (n)$ for $c = 6$, $n_0 = 1$

$f(n) = \Omega\ (g(n)) = \Omega\ (n)$ for $c = 5$, $n_0 = 1$

and so on.

Find lower bound of running time of quadratic function $f(n) = 3n^2 + 2n + 4$.

To find lower bound of f(n), we have to find c and n_0 such that $0 \le c.g(n) \le f(n)$ for all $n ³ n_0$

$0 \le c \times g(n) \le f(n)$

$0 \le c \times g(n) \le 3n^2 + 2n + 4$

$0 \le 3n^2 \le 3n^2 + 2n + 4$, → true, for all $n \ge 1$

$0 \le n^2 \le 3n^2 + 2n + 4$, → true, for all $n \ge 1$

Above both inequalities are true and there exists such infinite inequalities.

So, $f(n) = \Omega(g(n)) = \Omega(n^2)$ for $c = 3$, $n_0 = 1$

$f(n) = \Omega(g(n)) = \Omega(n^2)$ for $c = 1$, $n_0 = 1$

and so on.

Steps for Problem Solving

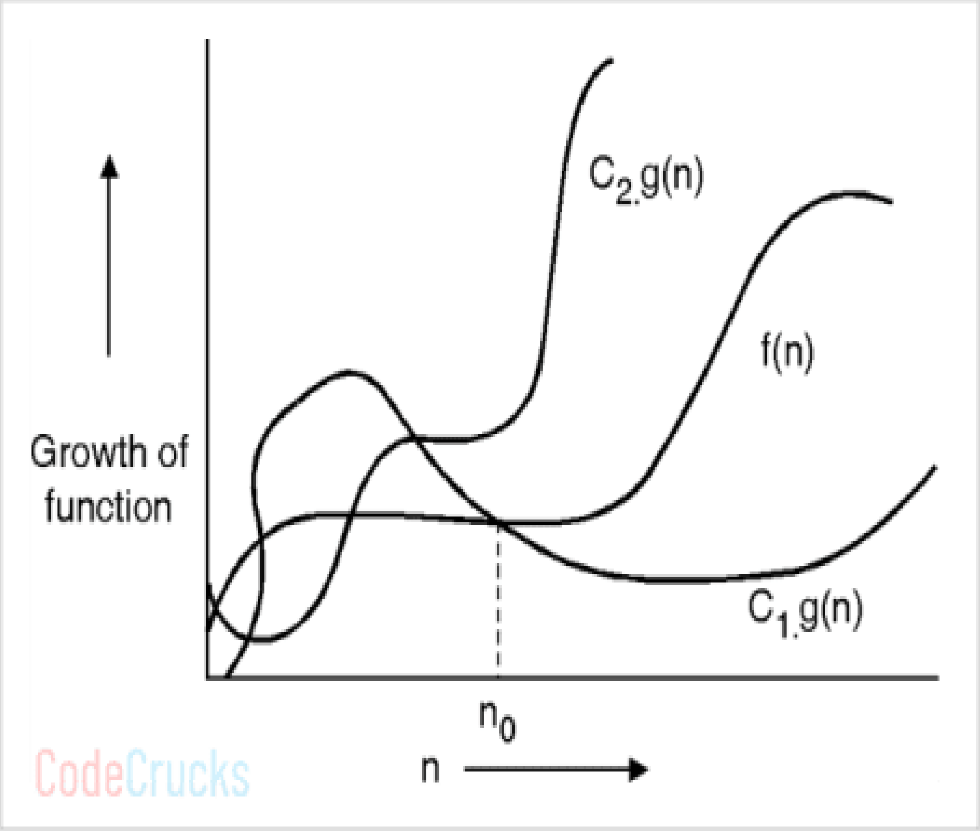
1. **Identifying Problem Statement (problem reframing stage)**
2. **Identifying Constraints**
3. **Design Logic**
4. **Validation**
5. **Analysis: a. Priori Analysis**

   b. Posterior Analysis
6. **Implementation**
7. **Testing and Debugging**

Theta Notation, θ

The notation θ(n) is the formal way to express both the lower bound and the upper bound of an algorithm's running time. It is also known as Tight bound.

We say the function g(n) is tight bound of function f(n) if there exist some positive constants c_1, c_2, *and* n_0 *such that* $0 \le c_1 \times g(n) \le f(n) \le c_2 \times g(n)$ *for all* $n \ge n_0$. *It is denoted as* $f(n) = \Theta(g(n))$.



Example: Find tight bound of running time of constant function f(n) = 23.

To find tight bound of f(n), we have to find c_1, c_2 and n_0 such that, $0 \le c_1 \times g(n) \le f(n) \le c_2 \times g(n)$ for all $n \ge n_0$

$0 \le c_1 \times g(n) \le 23 \le c_2 \times g(n)$

$0 \le 22 \times 1 \le 23 \le 24 \times 1$, → true for all $n \ge 1$

$0 \le 10 \times 1 \le 23 \le 50 \times 1$, → true for all $n \ge 1$

Above both inequalities are true and there exists such infinite inequalities.

So, $(c_1, c_2) = (22, 24)$ and $g(n) = 1$, for all $n \ge 1$

$(c_1, c_2) = (10, 50)$ and $g(n) = 1$, for all $n \ge 1$

$f(n) = \Theta(g(n)) = \Theta(1)$ for $c_1 = 22$, $c_2 = 24$, $n_0 = 1$

$f(n) = \Theta(g(n)) = \Theta(1)$ for $c_1 = 10$, $c_2 = 50$, $n_0 = 1$

and so on.

Example: Find tight bound of running time of a linear function $f(n) = 6n + 3$.

To find tight bound of $f(n)$, we have to find c_1, c_2 and n_0 such that,

$0 \leq c_1 \times g(n) \leq f(n) \leq c_2 \times g(n)$ for all $n \geq n_0$

$0 \leq c_1 \times g(n) \leq 6n + 3 \leq c_2 \times g(n)$

$0 \leq 5n \leq 6n + 3 \leq 9n$, for all $n \geq 1$

Above inequality is true and there exists such infinite inequalities.

So, $f(n) = \Theta(g(n)) = \Theta(n)$ for $c_1 = 5$, $c_2 = 9$, $n_0 = 1$

Find tight bound of running time of quadratic function $f(n) = 3n^2 + 2n + 4$.

To find tight bound of f(n), we have to find c_1, c_2 and n_0 such that, $0 \le c_1 \times g(n) \le f(n) \le c_2 \times g(n)$ for all $n \ge n_0$

$0 \le c_1 \times g(n) \le 3n^2 + 2n + 4 \le c_2 \times g(n)$

$0 \le 3n^2 \le 3n^2 + 2n + 4 \le 9n^2$, for all $n \ge 1$

Above inequality is true and there exists such infinite inequalities. So,

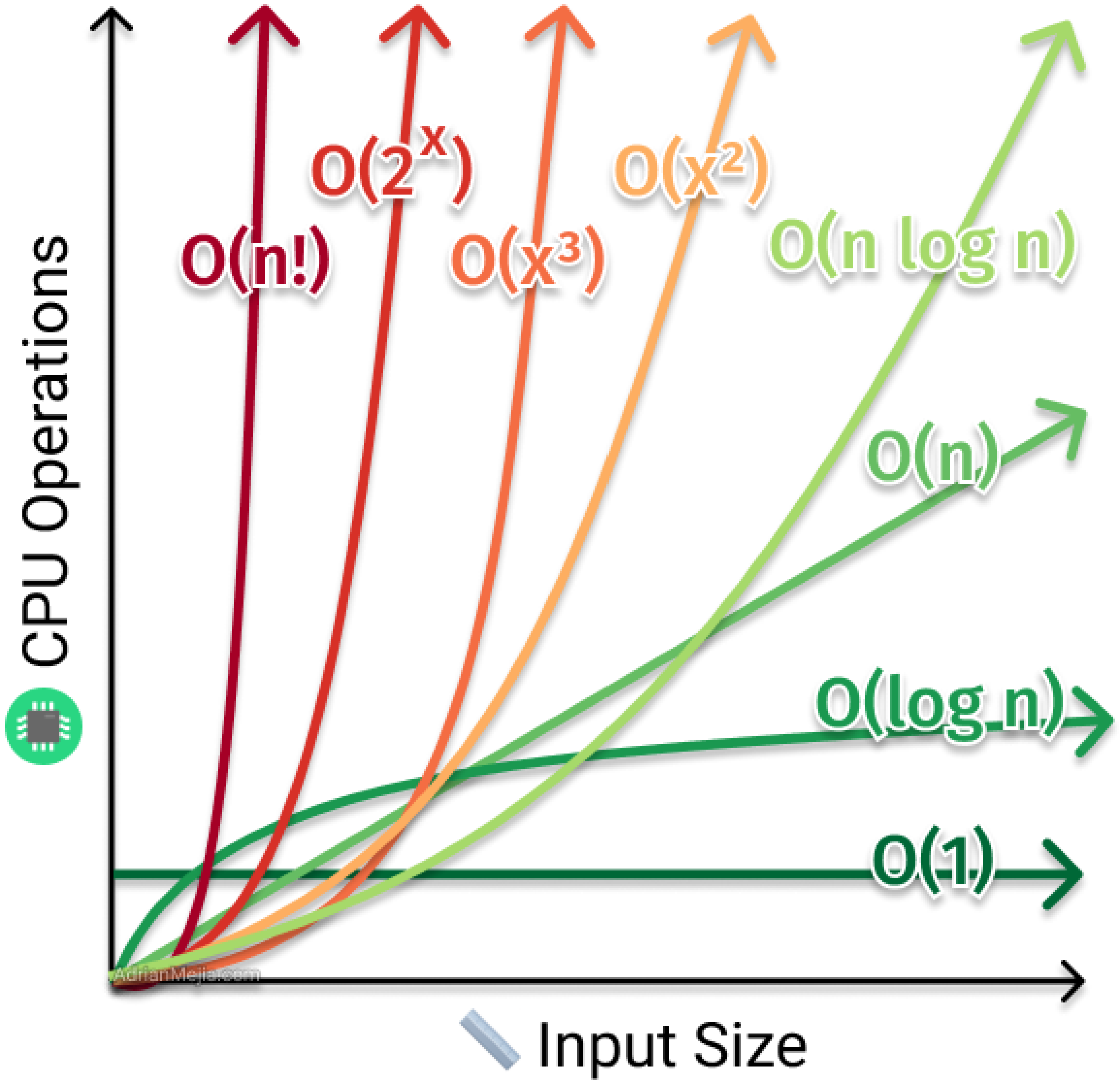
$f(n) = \Theta(g(n)) = \Theta(n^2)$ for $c_1 = 3$, $c_2 = 9$, $n_0 = 1$

constant	−	$O(1)$
logarithmic	−	$O(\log n)$
linear	−	$O(n)$
n log n	−	$O(n \log n)$
quadratic	−	$O(n^2)$
cubic	−	$O(n^3)$
polynomial	−	$n^{O(1)}$
exponential	−	$2^{O(n)}$

The following is the relationship between the order of growth rate:

$$O(1) < O(\log n) < O(n) < O(n \log n) < O(n^2) < O(n^3) < O(2^n) < n! < n^n$$

Time Complexity



Sequential Statements

If we have statements with basic operations like comparisons, assignments, reading a variable. We can assume they take constant time each **O(1)**

```
statement1;
statement2;
...
statementN;
```

If we calculate the total time complexity, it would be something like this:

```
T(n) = t(statement1) + t(statement2) + ... + t(statementN);
```

If each statement executes a basic operation, we can say it takes constant time O(1). As long as you have a fixed number of operations, it will be constant time, even if we have 1 or 100 of these statements.

Example of Sequential Statements

Let's say we can compute the square sum of 3 numbers.

```
function squareSum(a, b, c) {
  const sa = a * a;
  const sb = b * b;
  const sc = c * c;
  const sum = sa + sb + sc;
  return sum;
}
```

As you can see, each statement is a basic operation (math and assignment). Each line takes constant time `O(1)`. If we add up all statements' time it will still be `O(1)`. It doesn't matter if the numbers are `0` or `9,007,199,254,740,991`, it will perform the same number of operations.

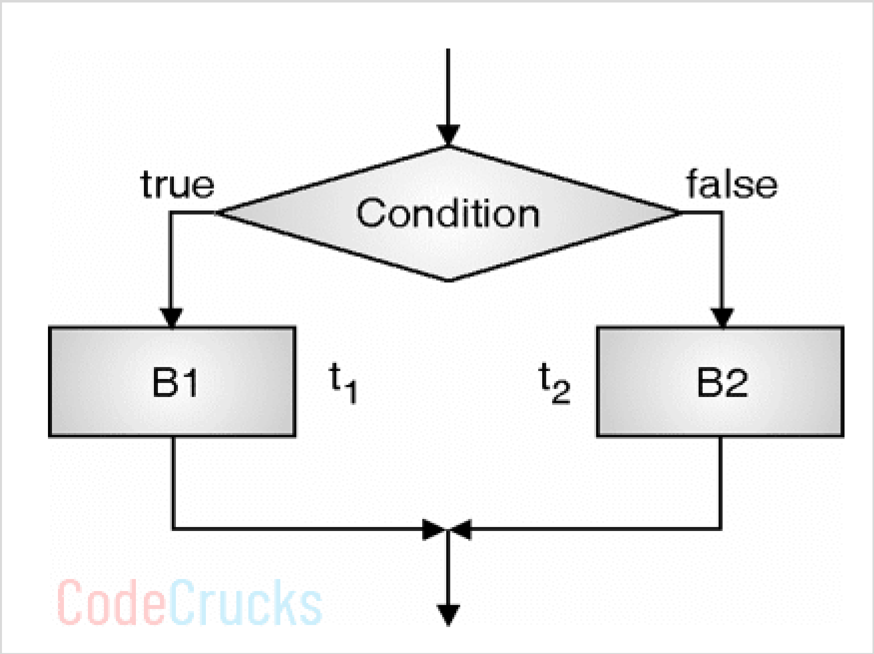
Conditional Statements

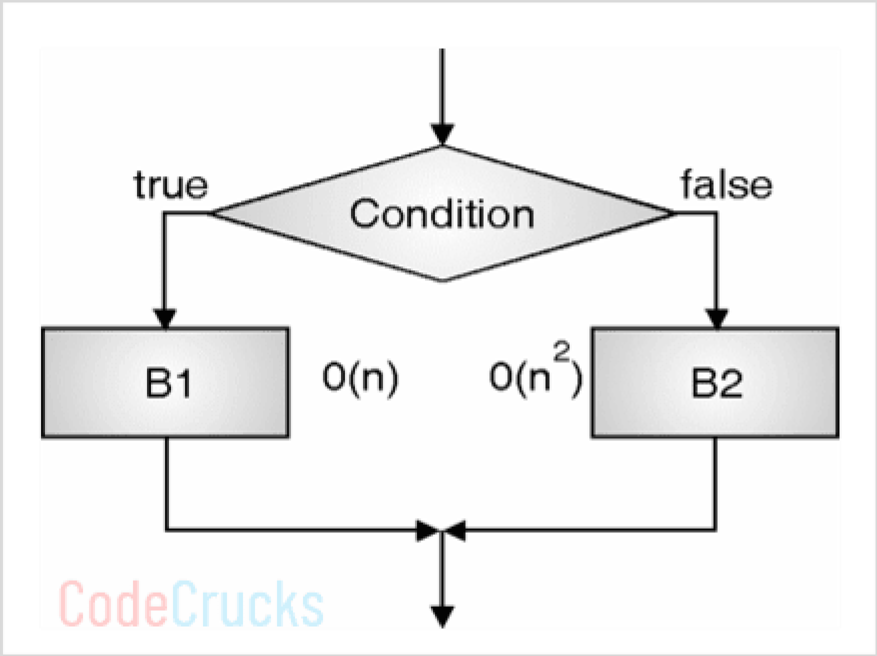
Very rarely, you have a code without any conditional statement. How do you calculate the time complexity? Remember that we care about the worst-case with Big O so that we will take the maximum possible runtime.

```
if (isValid) {
  statement1;
  statement2;
} else {
  statement3;
}
```

Since we are after the worst-case we take whichever is larger:

```
T(n) = Math.max([t(statement1) + t(statement2)], [time(statement3)])
```





$T(n) = \max(t_1, t_2)$

$T(n) = \max(O(n), O(n^2)) = O(n^2)$

Example:

```
if (isValid) {
  array.sort();
  return true;
} else {
  return false;
}
```

What's the runtime? The `if` block has a runtime of `O(n log n)` (that's common runtime for efficient sorting algorithms). The `else` block has a runtime of `O(1)`.

So we have the following:

```
O([n log n] + [n]) => O(n log n)
```

Since `n log n` has a higher order than `n`, we can express the time complexity as `O(n log n)`.

Linear Time Loops

For any loop, we find out the runtime of the block inside them and multiply it by the number of times the program will repeat the loop.

```
for (let i = 0; i < array.length; i++) {
  statement1;
  statement2;
}
```

For this example, the loop is executed `array.length`, assuming `n` is the length of the array, we get the following:

```
T(n) = n * [ t(statement1) + t(statement2) ]
```

All loops that grow proportionally to the input size have a linear time complexity `O(n)`. If you loop through only half of the array, that's still `O(n)`. Remember that we drop the constants so `1/2 n => O(n)`.

Nested loops statements

Sometimes you might need to visit all the elements on a 2D array (grid/table). For such cases, you might find yourself with two nested loops.

```
for (let i = 0; i < n; i++) {
  statement1;

  for (let j = 0; j < m; j++) {
    statement2;
    statement3;
  }
}
```

For this case, you would have something like this:

```
T(n) = n * [t(statement1) + m * t(statement2...3)]
```

Assuming the statements from 1 to 3 are **O(1)**, we would have a runtime of **O(n*m)**. If instead of m
you had to iterate on n, again, then it would be **O(n^2)**

Logarithmic Time Loops

Consider the following code, where we divide an array in half on each iteration (binary search):

```
function fn1(array, target, low = 0, high = array.length - 1) {
  let mid;
  while ( low <= high ) {
    mid = ( low + high ) / 2;
    if ( target < array[mid] )
      high = mid - 1;
    else if ( target > array[mid] )
      low = mid + 1;
    else break;
  }
  return mid;
}
```

This function divides the array by its `mid`dle point on each iteration. The while loop will execute the amount of times that we can divide `array.length` in half. We can calculate this using the `log` function. E.g. If the array's length is 8, then we the while loop will execute 3 times because $\log_2(8) = 3$.

Time Complexity Analysis

Let us assume that we have an array of length **32**. We'll be applying **Binary Search** to search for a random element in it. At each iteration, the array is halved.

- Iteration 0:
  - Length of array = 32
- Iteration 1:
  - Length of array = 32/2 = 16
- Iteration 2:
  - Length of array = 32/2^2 = 8
- Iteration 3:
  - Length of array = 32/2^3 = 4
- Iteration 4:
  - Length of array = 32/2^4 = 2
- Iteration 5:
  - Length of array = 32/2^5 = 1

Another example would be that for an array of size **1024**, only **10** iterations are needed to approach unity. For an array size of **32768**, we'll need only 15 iterations. Thus we can see that the number of operations grows at a very small rate compared to the size of the input array while complexity is logarithmic.

To generalize, after k iterations, our array size approaches 1.

Hence, n/2^k = 1 => n = 2^k

Applying logarithmic function on both sides, we get

=> log2 (n) = log2 (2^k) => log2 (n) = k log2 (2)

or, => k = log2 (n)

Hence, the time complexity of Binary Search becomes ***log2(n), or O(log n)***

```
while(n > 0)
{
  n = n / 2;
}
```

Here, decrement of n is not in linear order. Value of n is reduced by factor 2, maximum log2n divisions are possible before n reduce to 0. So,

$T(n) = O(\log_2 n)$

```
while(n > 0)
{
  n = n / m;
}
```

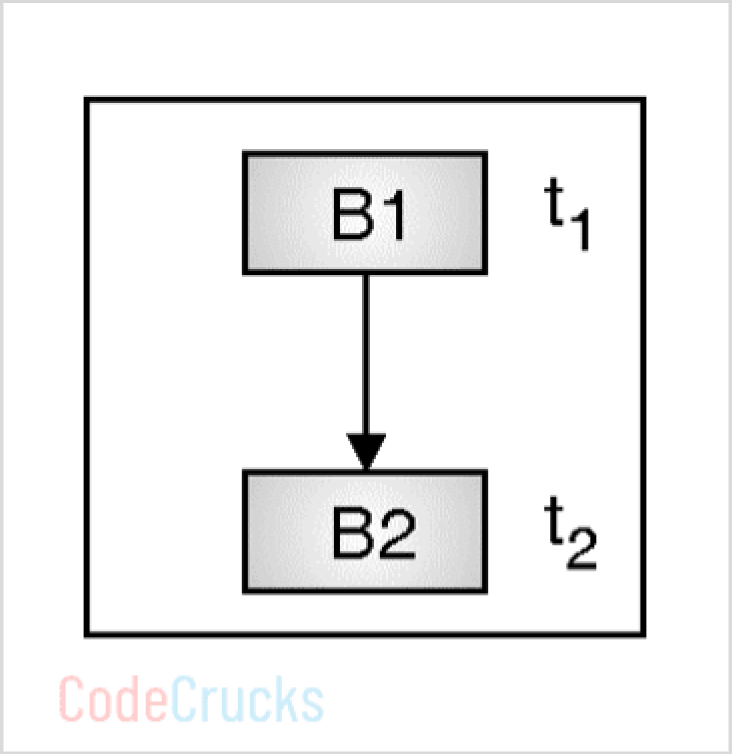
In the above code, n reduces by factor m in every iteration, so while loop can iterate maximum logmn times.

$T(n) = O(\log_m n)$

Sequential execution

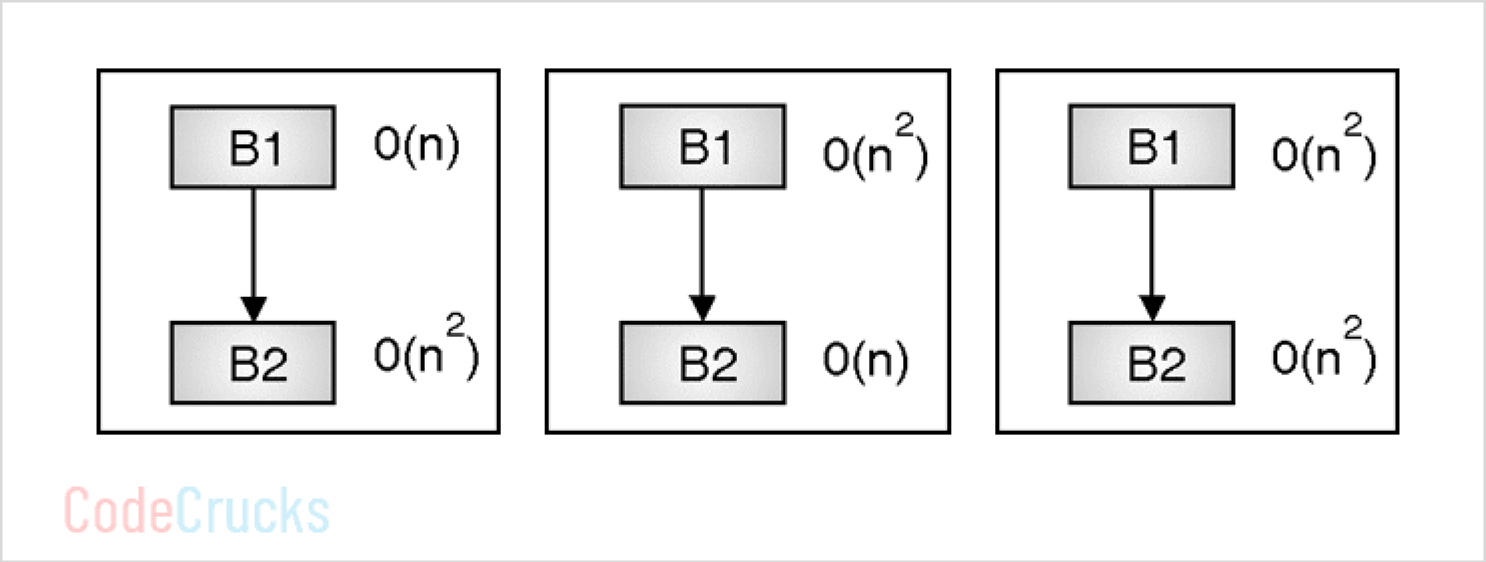
Statements or blocks of statements appear one after the other in sequential structures. Assume B1 and B2 are two blocks of instructions, each of which might contain a single instruction, several sequential instructions, or a collection of complicated instructions.

The execution of B1 and B2 are indicated in the figure to be consecutive. Assume that the time taken by code of B1 is t_1 and the time taken by code of B2 is t_2.

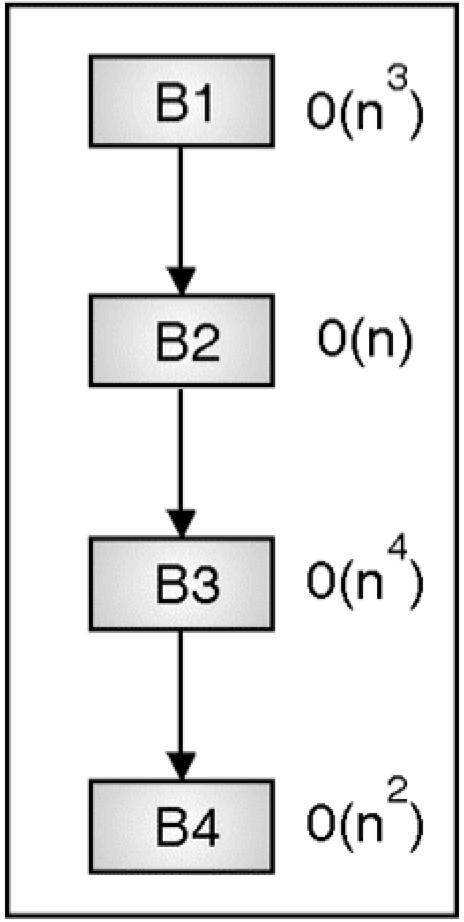


If B1 and B2 are executed in sequential order than the complexity of the entire program will be,

$$T(n) = t_1 + t_2 = \max(t_1, t_2)$$



$T(n) = O(n^2)$



In a complexity study, the order of code blocks is irrelevant. In general, if the program has m modules (or m functions), then the total complexity of the program is determined by first determining the complexity of each module, i.e. t_1, t_2,…, t_m. Find the maximum time for all of them; this is the overall program's complexity.

$$T(n) = \max (t_1, t_2, t_3, \ldots, t_m)$$

$T(n) = \max (t_1, t_2, t_3, \ldots, t_m) = \max (t_1, t_2, t_3, t_4)$
$= \max (O(n^3), O(n), O(n^4), O(n^2))$
$= O(n^4)$

Examples

```
int count = 0;
for (int i = 0; i < N; i++)
    for (int j = 0; j < i; j++)
        count++;
```

Lets see how many times **count++** will run.

When $i = 0$, it will run 0 times.
When $i = 1$, it will run 1 times.
When $i = 2$, it will run 2 times and so on.

Total number of times **count++** will run is $0 + 1 + 2 + \ldots + (N-1) = \frac{N*(N-1)}{2}$. So the time complexity will be $O(N^2)$.

```
int count = 0;
for (int i = N; i > 0; i /= 2)
    for (int j = 0; j < i; j++)
        count++;
```

This is a tricky case. In the first look, it seems like the complexity is $O(N * logN)$. N for the $j's$ loop and $logN$ for $i's$ loop. But its wrong. Lets see why.

Think about how many times **count++** will run.

When $i = N$, it will run N times.
When $i = N/2$, it will run $N/2$ times.
When $i = N/4$, it will run $N/4$ times and so on.

Total number of times **count++** will run is $N + N/2 + N/4 + \ldots + 1 = 2 * N$. So the time complexity will be $O(N)$.

Example

```
// function taking input "n"
int findSum(int n)
{
    int sum = 0; // ------------------> it takes some constant time "c1"
    for(int i = 1; i <= n; ++i) // --> here the comparision and increment will take place
                                        n times(c2*n) and the creation of i takes place
                                        with some constant time
        sum = sum + i; // -----------> this statement will be executed n times i.e. c3*n
    return sum; // -------------------> it takes some constant time "c4"
}
/*
* Total time taken = time taken by all the statments to execute
* here in our example we have 3 constant time taking statements i.e. "sum = 0", "i = 0", and
"return sum", so we can add all the constatnts and replacce with some new constant "c"
* apart from this, we have two statements running n-times i.e. "i < n(in real n+1)" and "sum =
sum  + i" i.e. c2*n + c3*n = c0*n
```

- Total time taken = c0*n + c
- The big O notation of the above code is O(c0*n) + O(c), where c and c0 are constants. So, the overall time complexity can be written as ***O(n)***.

Time Space Trade off

- A tradeoff is a situation where one thing increases and another thing decreases. It is a way to solve a problem in:
  - Either in less time and by using more space, or
  - In very little space by spending a long amount of time.
- **Types of Space-Time Trade-off**
  - Compressed or Uncompressed data
  - Smaller code or loop unrolling
  - Lookup tables or Recalculation
- In Time Space Trade off Time is inversely proportional to space and vice versa.

Thank You