Chapter 1: Principles of Analyzing Algorithms and Problems

Prepared By:
Amrit Poudel

Introduction to Algorithm

- An **algorithm** is a finite set of computational instructions that, if followed, accomplishes a particular task.
- You need **designing concepts** of the algorithms because if you only study the algorithms then you are bound to those algorithms and selection among the available algorithms. However if you have knowledge about design then you can attempt to improve the performance using different design principles.
- The **analysis of the algorithms** gives a good insight of the algorithms under study. Analysis of algorithms tries to answer few questions like;
    - is the algorithm correct? i.e. the Algorithm generates the required result or not?,
    - does the algorithm terminate for all the inputs under problem domain?
    - other issues like analysis of efficiency, optimality, etc.

Algorithm Properties

- An algorithm must satisfy following criteria:
  - **Inputs:**
    - Zero or more quantities are externally supplied.
  - Output:
    - At least one quantity is produced.
  - Definiteness:
    - Each instruction is clear and unambiguous.
  - Finiteness:
    - Algorithm must terminate after finite number of steps.
  - Effectiveness:
    - Every instruction must be very basic so that it can be carried out, in principle, by a person using pencil and paper.
    - It deals with feasibility of algorithm

Research Areas of Algorithms

- How to devise algorithms
  - Creating an algorithm is an art which may never be fully automated.
  - There are different algorithm design techniques.
    - Dynamic Programming
    - Linear Programming
    - Operation Research
  - Mastering these techniques help anyone to devise new and useful applications

Research Areas of Algorithms

- How to validate algorithms?
  - Once an algorithm is devised, it is necessary to show that it computes the correct answer for all possible legal inputs.This process is known as algorithm validation.
  - The purpose of the validation is to assure that the algorithm will work correctly independently of the issues concerning the programming language it will eventually written in.
- How to analyze algorithms?
  - Once the algorithm is executed, analysis of algorithm refers to the task of determining how much computing time and storage an algorithm requires.

Research Areas of Algorithms

- How to test a program?
  - Consists of 2 phases:
  - Debugging:
    - The process of executing programs on sample data to determine whether faulty results occur, and , if so, to correct them.
  - Profiling(a.k.a performance measurement)
    - Process of executing a correct program on data sets and measuring the time and space it takes to compute the result.

Pseudocode

It is an approach to describe algorithm.

Some of the **conventions** used to write the pseudocodes are:

- Comments begin with // and continue until the end of the line.
- Blocks are indicated with matching braces: { and }.
- An identifier begins with letter.
- The data types of variables are not explicitly declared.
- The variable type(local and global) are not explicitly declared.
- Assignment of values to variables is done using the assignment statement.

  (variable) := (expression)

(cont…)

Convention to write pseudocode(contd..)

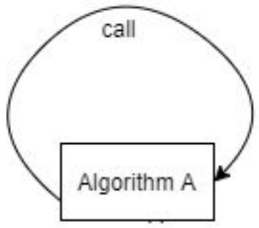
- There are two boolean values **true** and **false.** In order to produce these values, the logical operators and,or, not and the relational operators <,<=,≠,>= and > are provided.
- Elements in multidimensional arrays are accessed using [and]. For example, if A is a multidimensional array, the *(i, j)th* element of the array is denoted by A[*i*,*j*], Array index start from zero.
- The for, while and repat-until statements are employed for looping.
- The if..then, if...then..else and case statements are used to handle conditions.
- Inputs and Outputs are done using instructions **read** and **write. N**o format to specify the size of the input and output quantities.

Convention to write pseudocode(contd..)

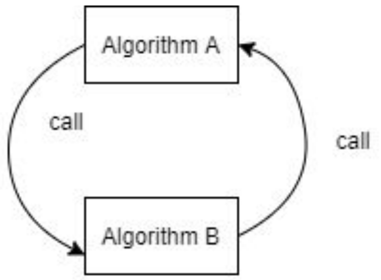
- There is only one type of procedure: **Algorithm.**
  - Algorithm consists of heading and body.
  - Heading takes the form Algorithm Name((parameter list))
    - Syntax: Algorithm Name(parameter list)\

Recursive Algorithms

- An algorithm is said to be recursive if the same algorithm is sinvoked in the body.
- Two types:
    - Direct Recursive
        - An algorithm that call itself
    - Indirect Recursive
        - An algorithm that calls another algorithm that in turns calls A



Direct Recursion

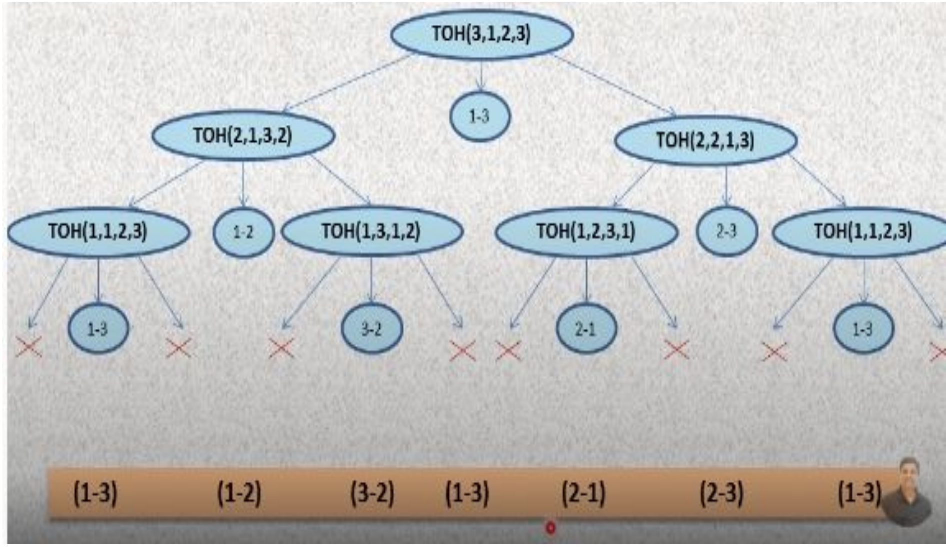


Inidrect Recursion

Recursive Algorithm Examples:

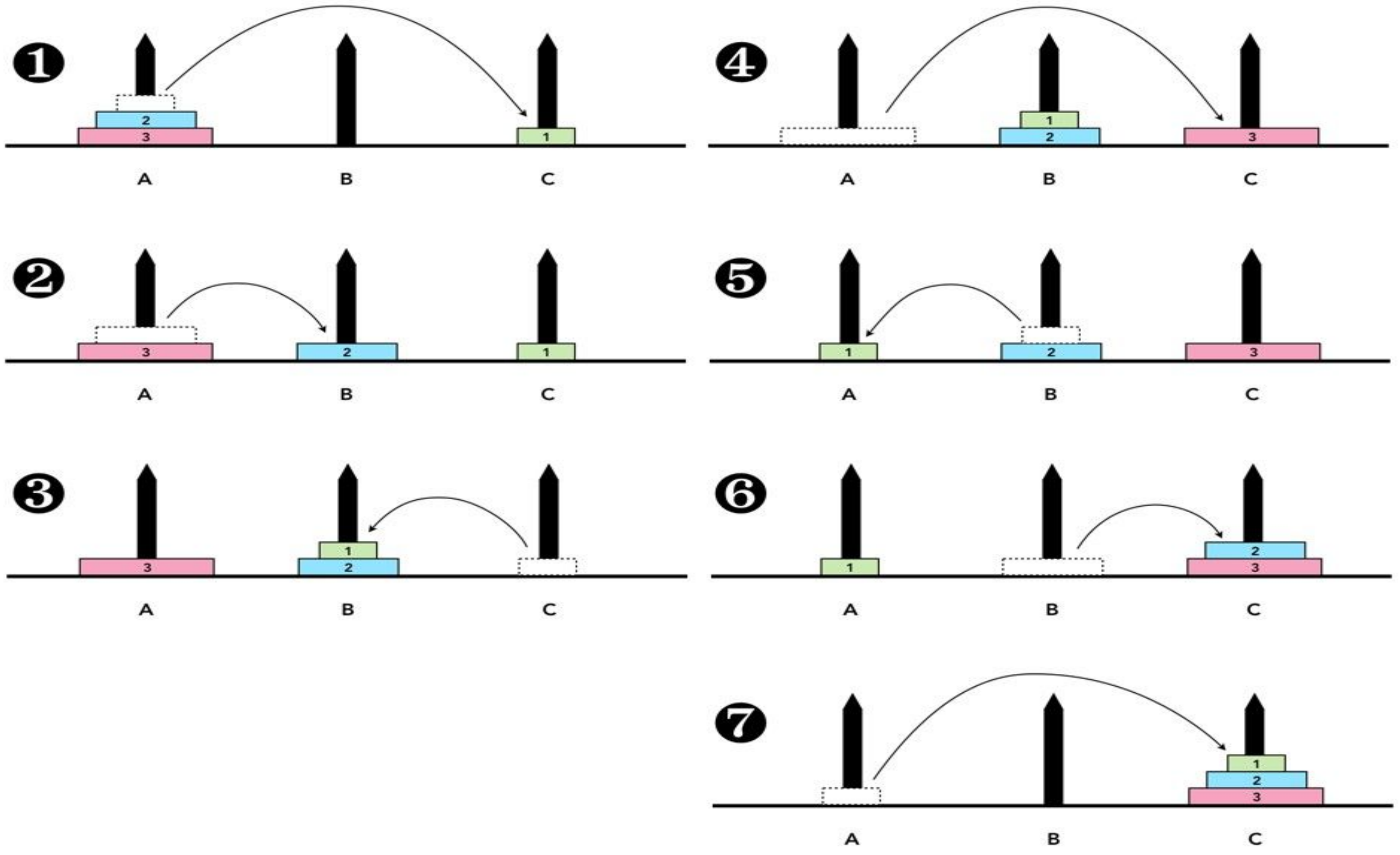
Tower of Hanoi

```
Algorithm TOH( n, A, B,C){
  If (n>0) then{
          TOH(n-1, A, C, B);
          write("Move a disc from tower", A, "to tower ", C);
          TOH(n-1, B, A, C)
     }
}
```



Tracing for 3 Discs

(1-3) (1-2) (3-2) (1-3) (2-1) (2-3) (1-3)



Performance Analysis

Performance Analysis of algorithm can be done using:

1. Space Complexity
2. Time Complexity

Space Complexity

Defn: The space complexity of an algorithm is the amount of memory it needs to solve the problem.

S(P) = C + Sp(instance)

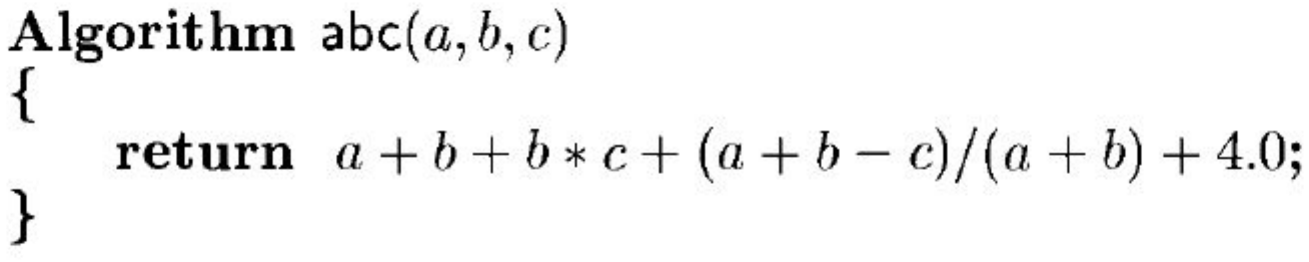
S(P) = space complexity of algorithm a

C = fixed part(e.g: space to store code of algorithm and so on)

Sp(instance) = space to store variable part(this space is not fixed)

Note: *When analyzing the space complexity of an algorithm , we concentrate solely on estimating Sp*

Space Complexity Examples



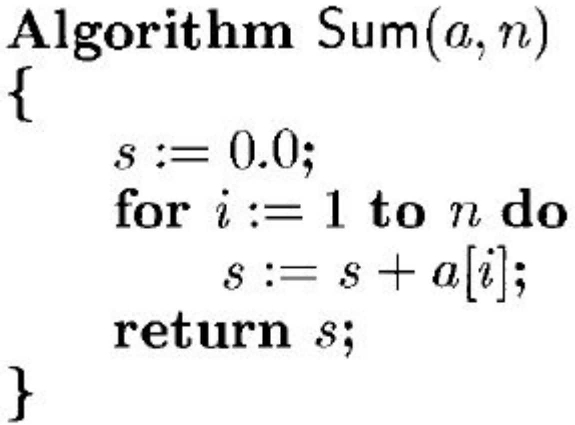
```
Algorithm abc(a, b, c)
{
    return $a + b + b * c + (a + b - c)/(a + b) + 4.0$;
}
```

For every instance, 3 words are required to store variables a,b and c.

S(P) = C + Sp(instance)

S(P) = 3 + 0 = 3

Example 2: Iterative function for sum



```
Algorithm Sum($a, n$)
{
    $s := 0.0$;
    for $i := 1$ to $n$ do
        $s := s + a[i]$;
    return $s$;
}
```

For every instance,

To store a = n words

To store n = 1 word

To store i and s = 2 words

S(P) = C + Sp(instance)

S(P) = (1+2) + n

= 3 +n

Example 3: Recursive function for sum

```
Algorithm RSum(a, n)
{
    if ($n \le 0$) then return 0.0;
    else return RSum(a, $n - 1$) + $a[n]$;
}
```

To store **n** = 1 word

To store **return address** = 1 word

To store a **pointer** to a= 1 word

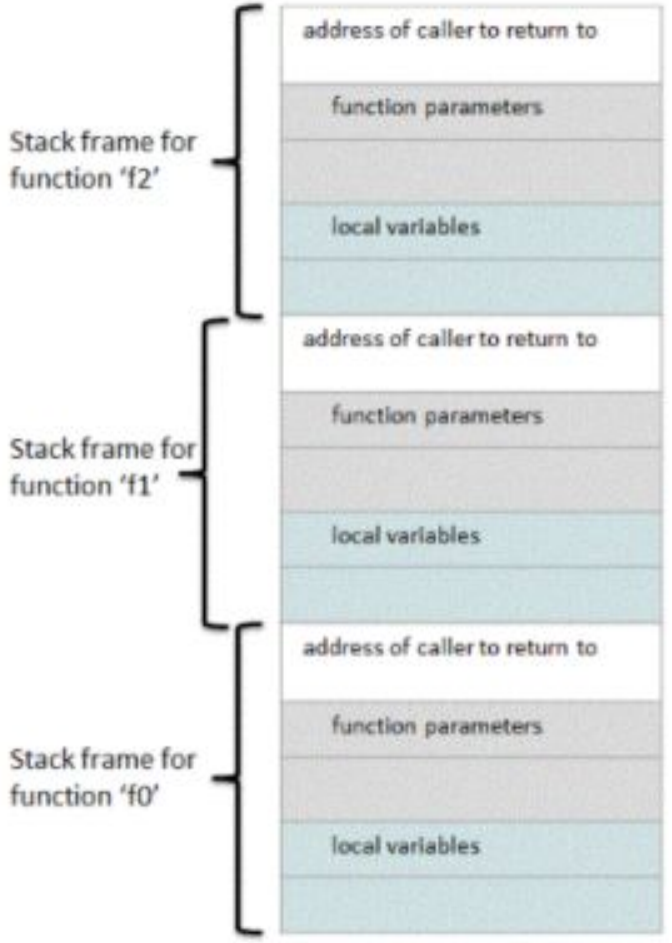
Since the **depth of recursion** is n + 1,

The recursion stack space needed is

3(n + 1)

How to calculate space complexity in recursive algorithm?

- To calculate space complexity, one need to understand how the stack frames are generated in memory for recursive call sequence.
- In above algorithm, when a function RSum(1) is called from function RSum(0), stack frame corresponding to this function RSum(1) is created.
- This stack frame is kept in the memory until call to function RSum(1) is not terminated.
- This stack frame is responsible for saving the parameters of function RSum(1), local variables in the function 'RSum(1)' and return address of caller function(function 'RSum(0)').
- Now when this function RSum(1),calls another function RSum(2), stack frame corresponding to 'RSum(2)' is also generated and is kept in the memory until call to 'RSum(2)' is not terminated and so on.


Stack frame for function 'f2'
address of caller to return to
function parameters
local variables
Stack frame for function 'f1'
address of caller to return to
function parameters
local variables
Stack frame for function 'f0'
address of caller to return to
function parameters
local variables

Call stack in memory for call sequence f0->f1->f2

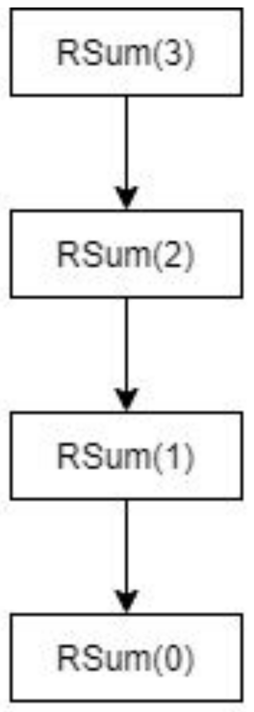
Let,

f0 = RSum(0)

f1 = RSum(1)

f2 = RSum(2)

- Now when call to function 'f2' returns, the stack frame corresponding to 'f2' is deleted from memory since it is no longer required. Same is the case for stack frames of function 'f1' and function 'f0'.
- Using this analogy for recursive call sequence, it should follow that maximum number of stack frames that could be present in memory at any point of time is equal to maximum depth of recursion tree.



Recursion Tree generated for computing sum upto 3

Time Complexity

Defn: The time complexity of an algorithm is the amount of time it needs to solve the problem.

T(P) = Compile Time + Run Time

T(P) = C + tp(instance)

Compile time

- does not depend on the instance characteristics.
- can also assumed that a compiled program can be ruined several times without recompilation

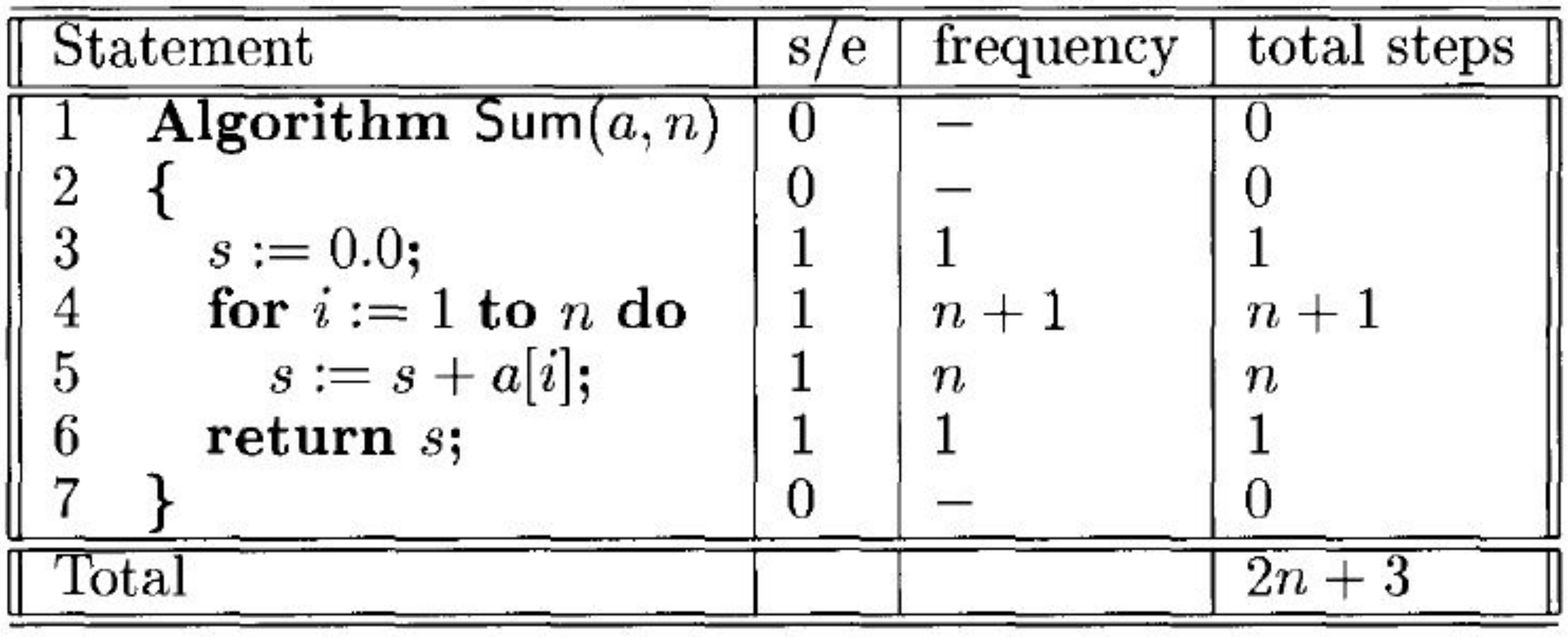
Run Time:

- Depends on the instance characteristics
- Thus, time complexity is focused on run time.

How time complexity calculated?

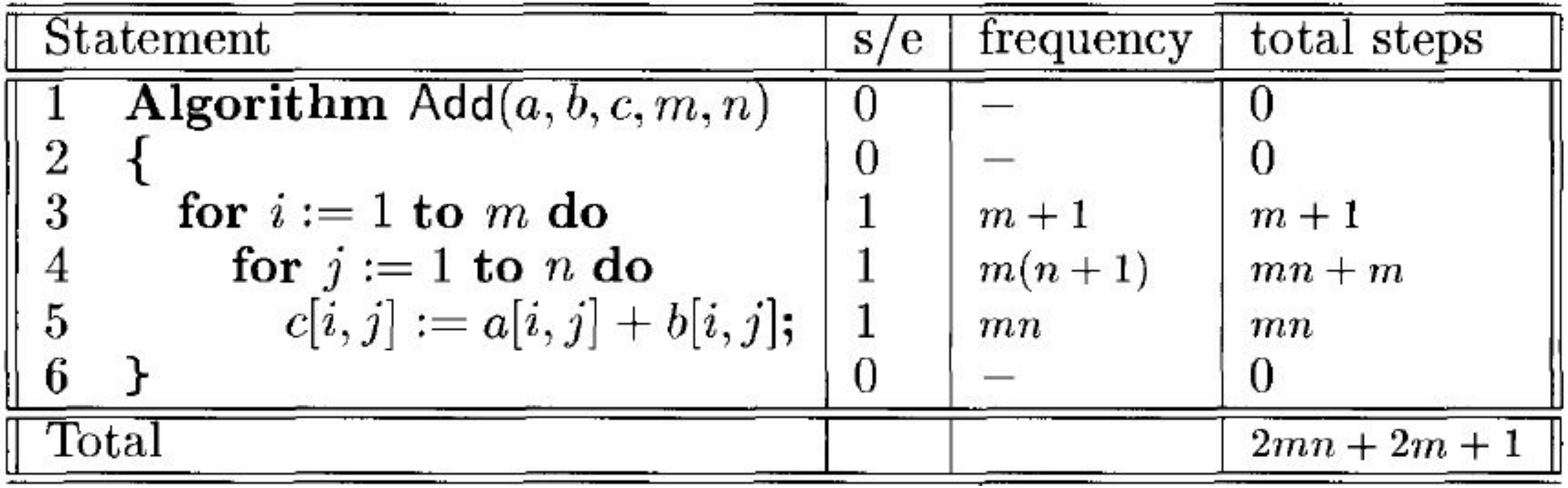
- Time complexity can be determined by analyzing the program’s statements (go line by line).
- However, one need to be mindful how are the statements arranged. Suppose they are inside a loop or have function calls or even recursion. All these factors affect the runtime of your code.

Example:



Statement	s/e	frequency	total steps
1 **Algorithm** Sum(a, n)	0	–	0
2 {	0	–	0
3 $s := 0.0;$	1	1	1
4 **for** $i := 1$ **to** n **do**	1	$n + 1$	$n + 1$
5 $s := s + a[i];$	1	n	n
6 **return** s;	1	1	1
7 }	0	–	0
Total			$2n + 3$

Example 2:



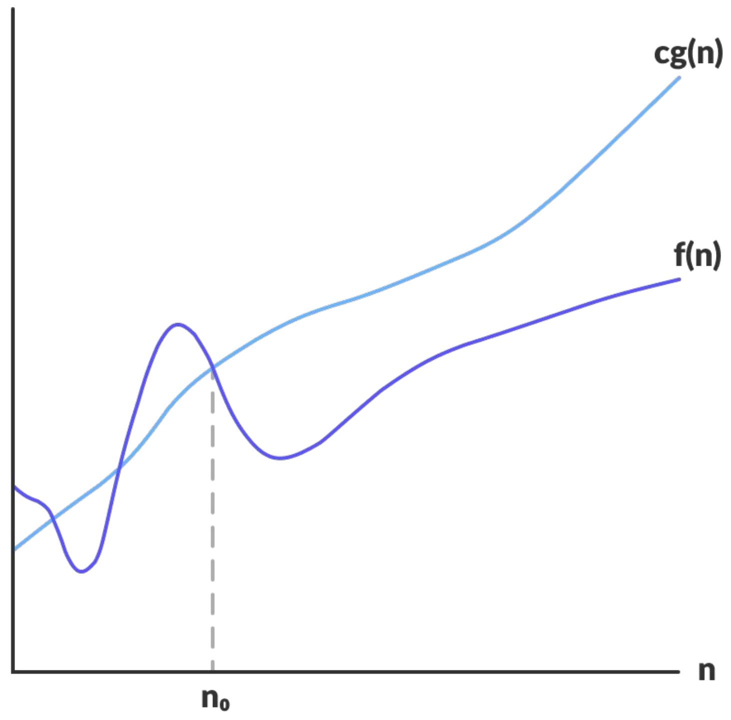
Statement	s/e	frequency	total steps
1 **Algorithm** Add(a, b, c, m, n)	0	—	0
2 {	0	—	0
3 **for** $i := 1$ **to** m **do**	1	$m + 1$	$m + 1$
4 **for** $j := 1$ **to** n **do**	1	$m(n + 1)$	$mn + m$
5 $c[i, j] := a[i, j] + b[i, j]$;	1	mn	mn
6 }	0	—	0
Total			$2mn + 2m + 1$

Asymptotic Notation

- used to describe the running time of an algorithm - how much time an algorithm takes with a given input, n.
- three different notations:
    - big O,
    - big Theta (Θ), and
    - big Omega (Ω)
- big-Θ is used when the running time is the same for all cases,
- big-O for the worst case running time, and
- big-Ω for the best case running time.

Big-O 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.
- the Big-O of an algorithm can be computed by counting how many iterations an algorithm will take in the worst-case scenario with an input of N.
- the Big-O is typically consulted because one must always plan for the worst case.

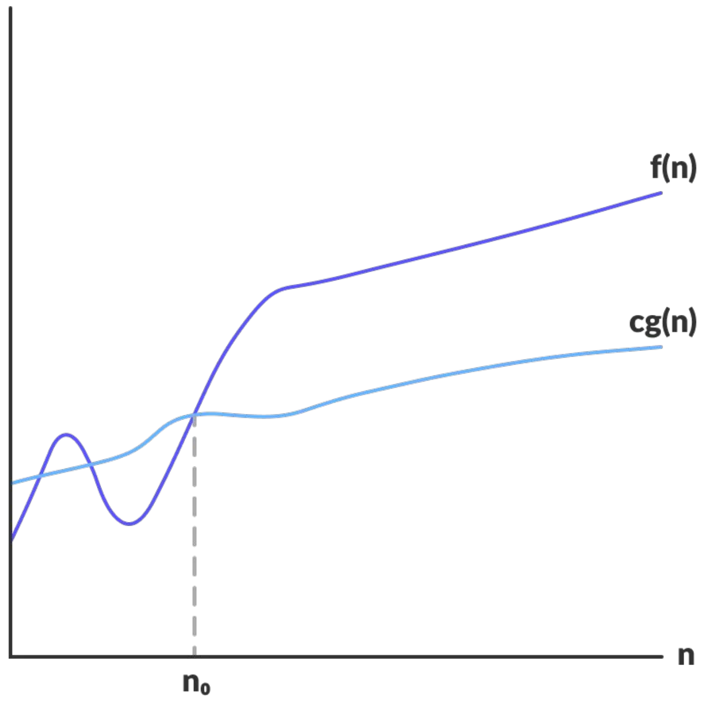


f(n) = O(g(n))

```
O(g(n)) = { f(n): there exist positive constants c and n0
            such that 0 ≤ f(n) ≤ cg(n) for all n ≥ n0 }
```

Big 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

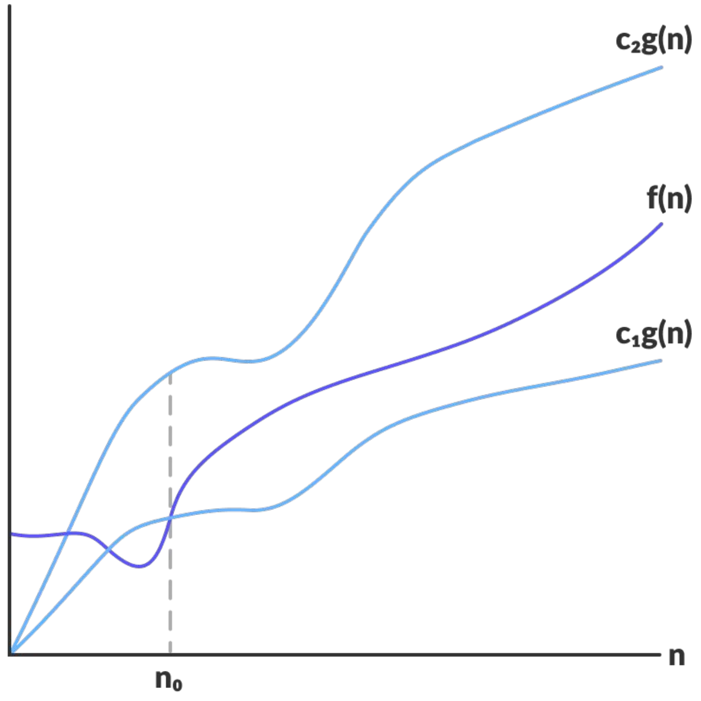


f(n) = Ω(g(n))

$\Omega(g(n)) = \{ f(n)$: there exist positive constants c and n_0 such that $0 \le cg(n) \le f(n)$ for all $n \ge n_0$ $\}$

Big 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.



$f(n) = \Theta(g(n))$

$\Theta(g(n)) = \{ f(n)$: there exist positive constants c_1, c_2 and n_0
such that $0 \le c_1g(n) \le f(n) \le c_2g(n)$ for all $n \ge n_0$ $\}$

Calculating Time Complexity

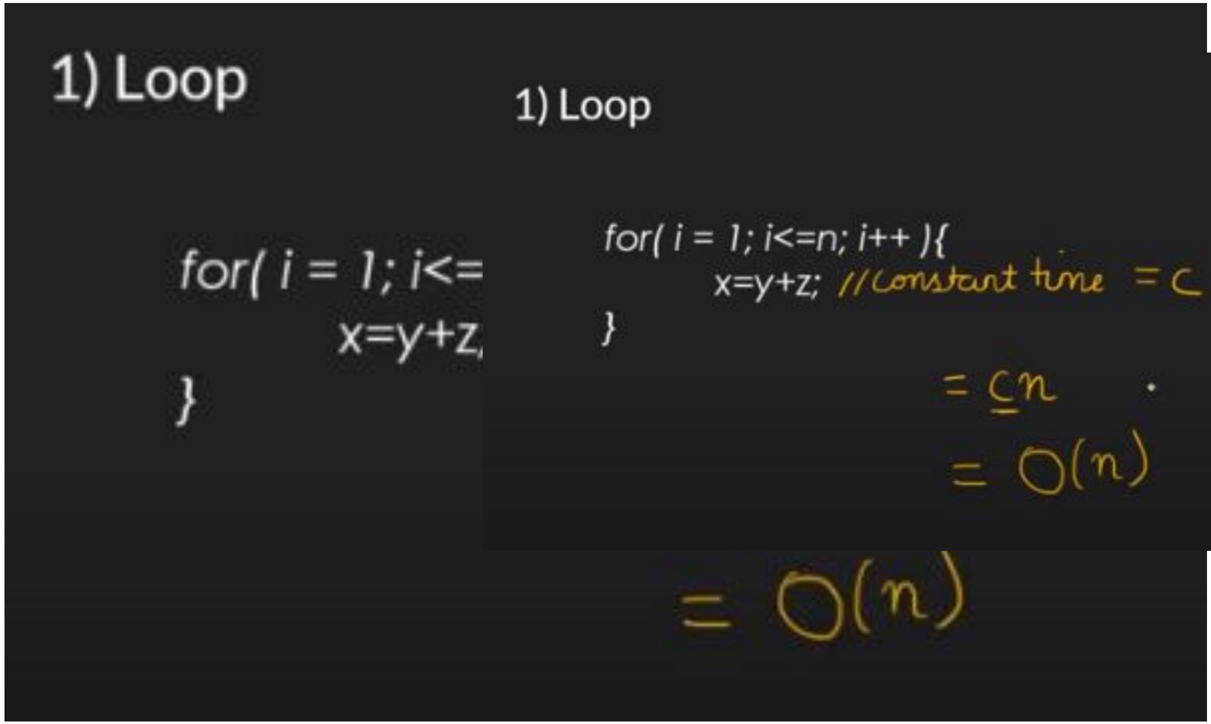
Steps:

- Drop Lower order terms
- Drop all constant multipliers

$$T(n) = \cancel{2}\underline{n^2} + \underline{\cancel{3n+1}}$$

$$= O(n^2)$$

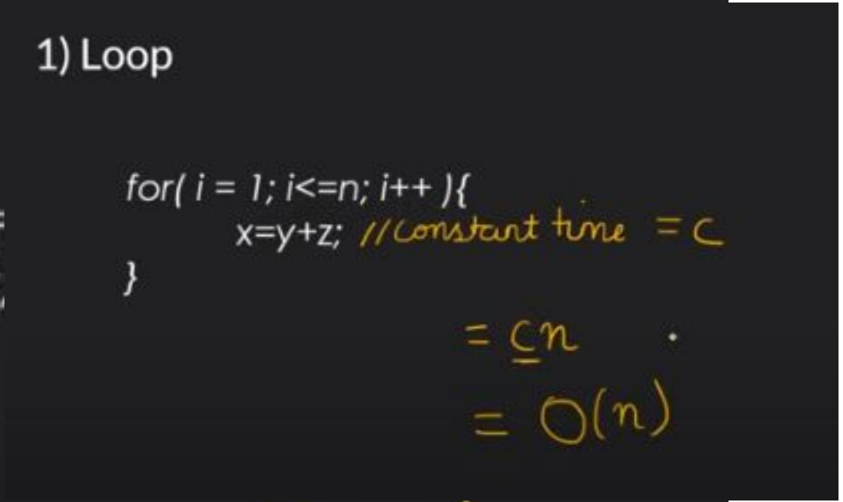
Calculating time complexity for single loop



1) Loop

```
for( i = 1; i<=
        x=y+z;
}
```

$= O(n)$



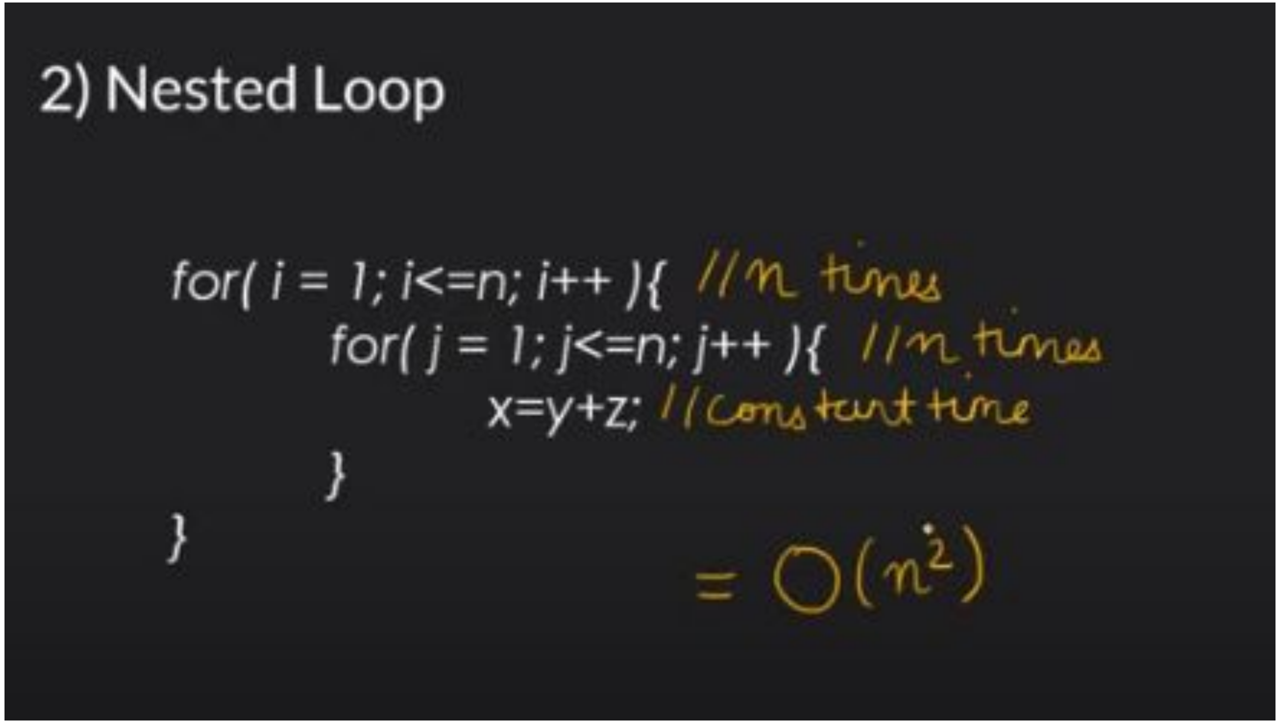
1) Loop

```
for( i = 1; i<=n; i++ ){
        x=y+z; //constant time = c
}
```

$= cn$

$= O(n)$

Time complexity for nested loop



2) Nested Loop

```
for( i = 1; i<=n; i++ ){ //n times
        for( j = 1; j<=n; j++ ){ //n times
                x=y+z; //constant time
        }
}
```

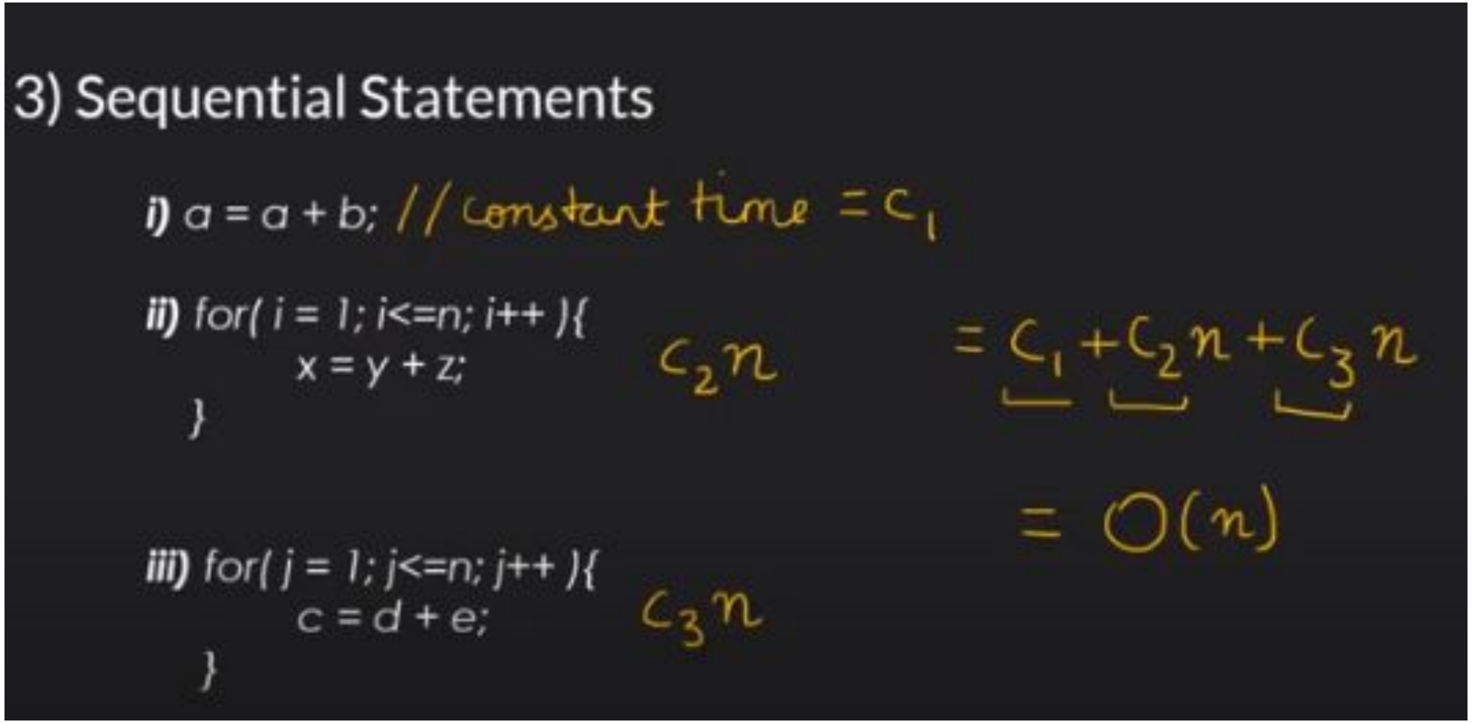
$= O(n^2)$

Time Complexity for sequential statements

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

Time for each statement is constant and the total time is also constant: O(1)

Time Complexity for sequential statements



3) Sequential Statements

```
i) a = a + b;
```
// constant time $= c_1$

```
ii) for( i = 1; i<=n; i++ ){
          x = y + z;
    }
```
$c_2 n$

```
iii) for( j = 1; j<=n; j++ ){
          c = d + e;
     }
```
$c_3 n$

$= \underbrace{c_1} + \underbrace{c_2} n + \underbrace{c_3} n$

$= O(n)$

Conditional Statements

```
if (isValid) {
  statement 1;
  statement 2;
} else {
  statement 3;
}
```

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

```
T(n) = Math.max([t(statement 1) + t(statement 2)], [time(statement 3)])
```

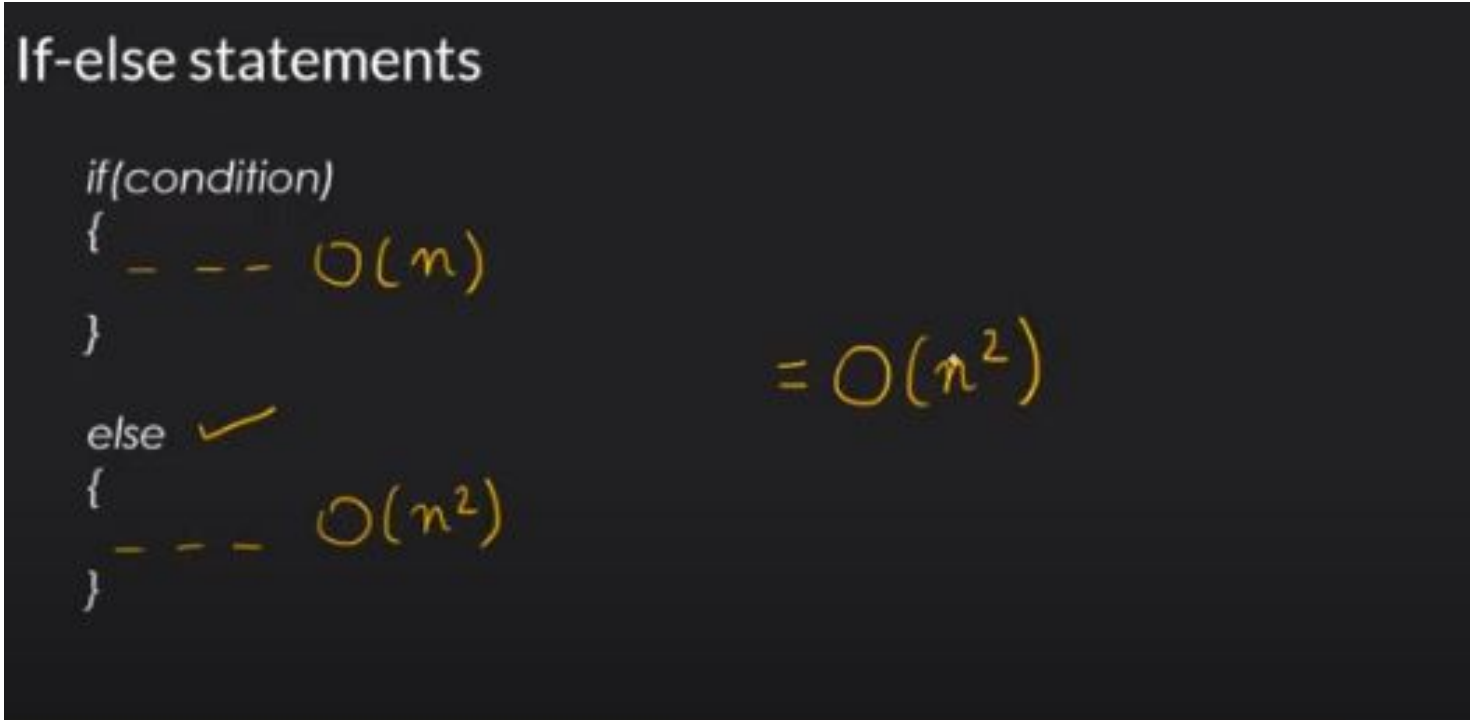
Conditional Statements

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

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)`

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

Conditional Statements

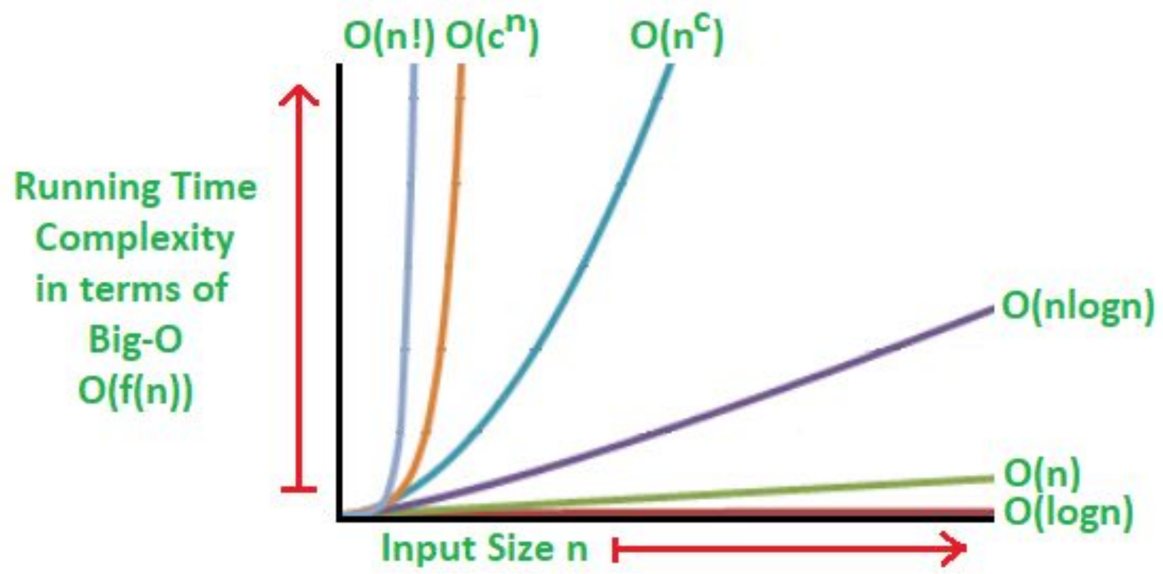


If-else statements

```
if(condition)
{
  - - -  O(n)
}

else ✓
{
  - - -  O(n^2)
}
```

$= O(n^2)$



$O(n!), O(c^n), O(n^c)$ - Worst

$O(n\log n)$ - Bad

$O(n)$ - Fair

$O(\log n)$ - Good

$O(1)$ - Best

$O(1) < O(\log n) < O(n) < O(n\log n) < O(n^c) < O(n!$

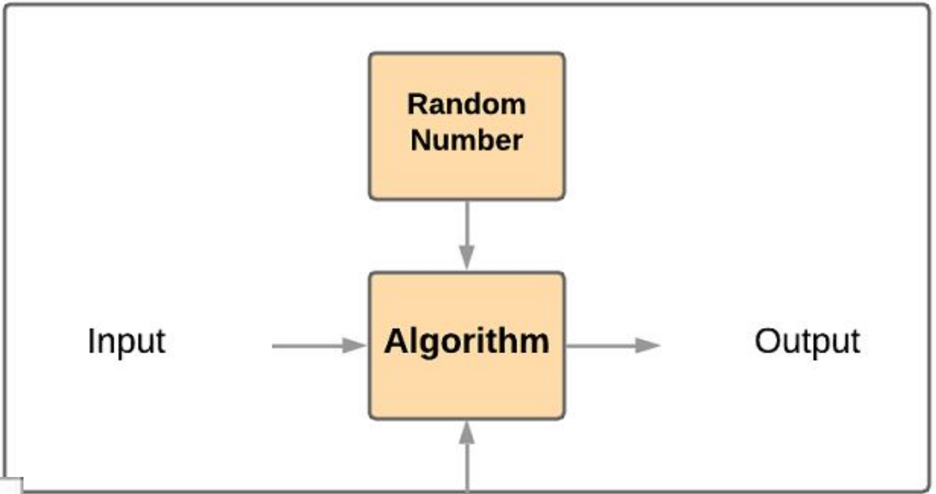
Randomized Algorithm

- Makes use of randomizer(such as a random number generator)
- Some of the decisions made in the algorithm depend on the output of the randomizer.
- Since the output of any randomizer differ from run to run, the output of the randomized algorithm could also differ from run to run for same input.
- The execution time of the randomized algorithm could also differ from run to run for the same input.
- They are usually simpler and more efficient(time and space complexity) than its deterministic algorithm

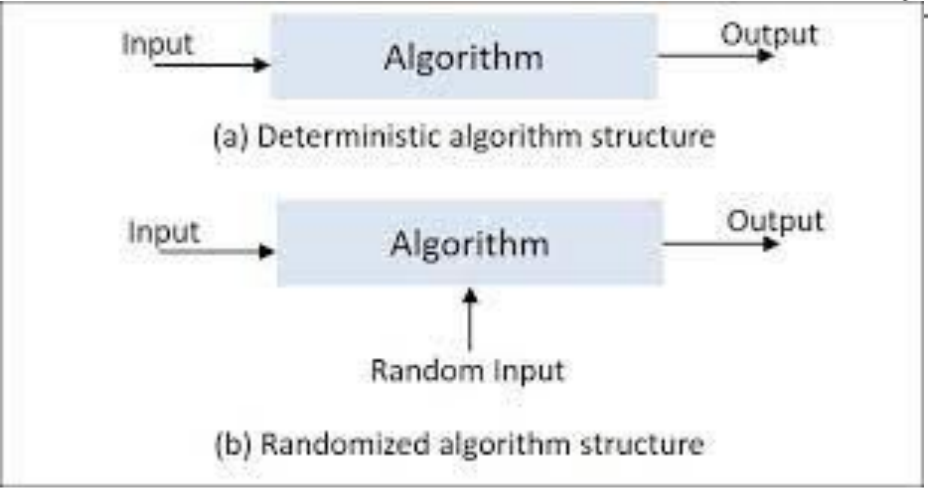
Randomized Algorithm

Types:

- Las Vegas Algorithm
- Monte Carlo Algorithm



Randomized Algorithm Flowchart


Input
Algorithm
Output
(a) Deterministic algorithm structure
Input
Algorithm
Output
Random Input
(b) Randomized algorithm structure

Las Vegas Algorithm

- Algorithms that use the random input so that they always terminate with the correct answer but the running time is not fixed.
- **Algorithm LasVegas**(A, n, a)

```
{
    While (true)
    {
        i:= Random();//randomly select an element out of n elements
        if(i = a)
            Return i;
    }
}
Here, no of iterations is not fixed, but the result is always
obtained
```

Monte Carlo Algorithm

- Algorithms which have a chance of producing an incorrect result, but the running time is fixed.
- `Algorithm MonteCarlo(A, n, a, x)`

```
{
    i := 0;
    tag := False
    while (i<= x)
     {
        b:= Random();//randomly select an element out of n elements
         i := i+1
        if(b = a)
              tag = True;
     }
     return tag
}
Here, no of iterations are fixed, but the result may not be always obtained
```