Analysis of Algorithm

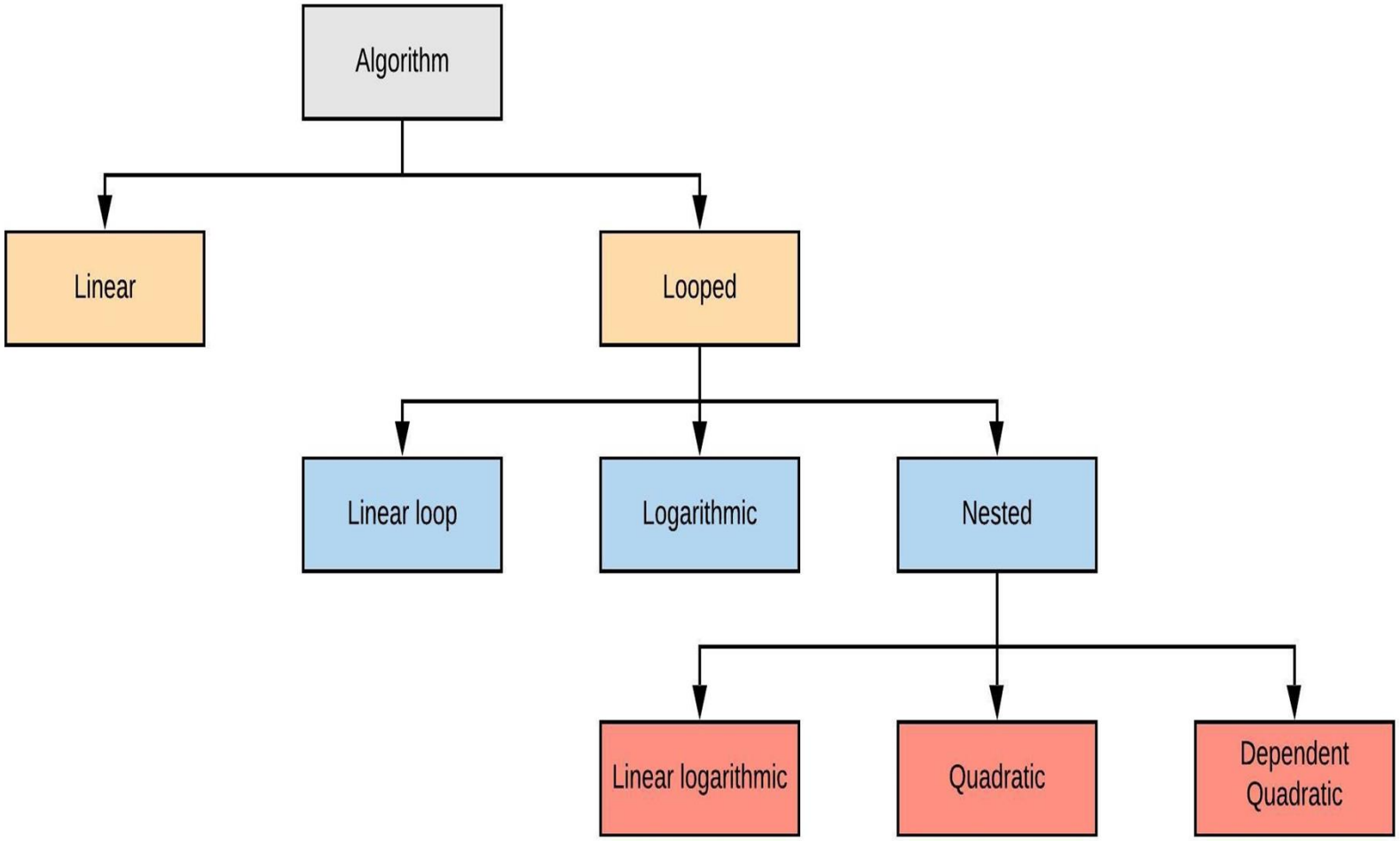
- **Time Complexity**
  - Best Case
  - Average Case
  - Worst Case
  - Amortized Case
- **Space Complexity**
  - Fixed Part
  - Variable Part

How to find Complexity?

- Linear Function: No Loops, No call to functions
  - Running Time = Number of Instructions

- Loops
  - Running Time depends upon
    - number of loops
    - Complexity of loops

Analyzing Time Complexity



Analyzing Time Complexity

Linear Loops

```
for(i=0;i<100;i++)
        statement block;
for(i=0;i<100; i+=2)
        statement block;
```

➔ f(n) = n

➔ f(n) = n/2

Logarithmic Loops

```
for(i=1;i<1000;i*=2)
    statement block;
for(i=1000;i>=1;i/=2)
    statement block
```

➔ f(n) = $\log_2 n$

i=1,2,4,8,16,32,64

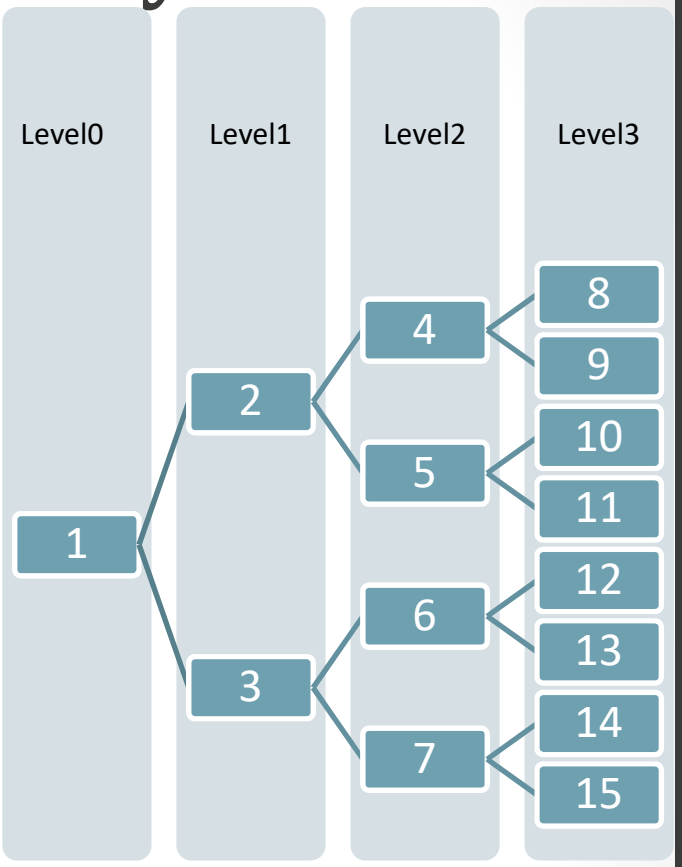
Analyzing Time Complexity

Example 1:

```
int i = 1;
while (i <= n)
{
        System.out.println("*");
        i = 2 * i;
}
```

No. of times * prints

$1+ \log_2 n$



Analyzing Time Complexity: ***Nested Loops***

Linear logarithmic loop

```
for(i=0;i<10;i++)
        for(j=1; j<10;j*=2)
                statement block;
```

➔ f(n) = n logn

Quadratic loop

```
for(i=0;i<10;i++)
        for(j=0; j<10;j++)
                statement block;
```

➔ $f(n) = n^2$

Dependent quadratic loop

```
for(i=0;i<10;i++)
        for(j=0; j<=i; j++)
                statement block;
```

➔ f(n) = n (n + 1)/2

1 + 2 + …. + 9 + 10 =55

Arithmetic Series: 1 + 2 + 3 + … + n = $n*(n+1)/2$

Analyzing Time Complexity

Example 2:

```
1.          for (int i = 1; i <=m; i += c)
2.                      {   Statement-1   }
3.          for (int i = 1; i <=n; i += c)
4.                      {   Statement-2 }
```

Complexity:= m + n

if n=m then 2n

Example 3:

```
1.    int i = n;
2.    while (i > 0) {
3.                      for (int j = 0; j < n; j++)
4.                              System.out.println("*");
5.                       i = i / 2;
6.          }
```

Outer Loop: $\log_2 n$

Inner Loop: n

$F(n) = n * \log_2 n$

Analyzing Time Complexity

Example 4:

```
for (int i = 0; i < n; i++) // loop 1
        for (int j = i+1; j > i; j--) // loop 2
                for (int k = n; k > j; k--) // loop 3
                        System.out.println("*");
```

Loop1:- n

Loop2:- 1

Loop 3:- Dependent

Follows arithmetic Series:- 1 + 2 + 3 + ... + (n-2) + (n-1) + n

Total= $(n + 1)(n)/2$

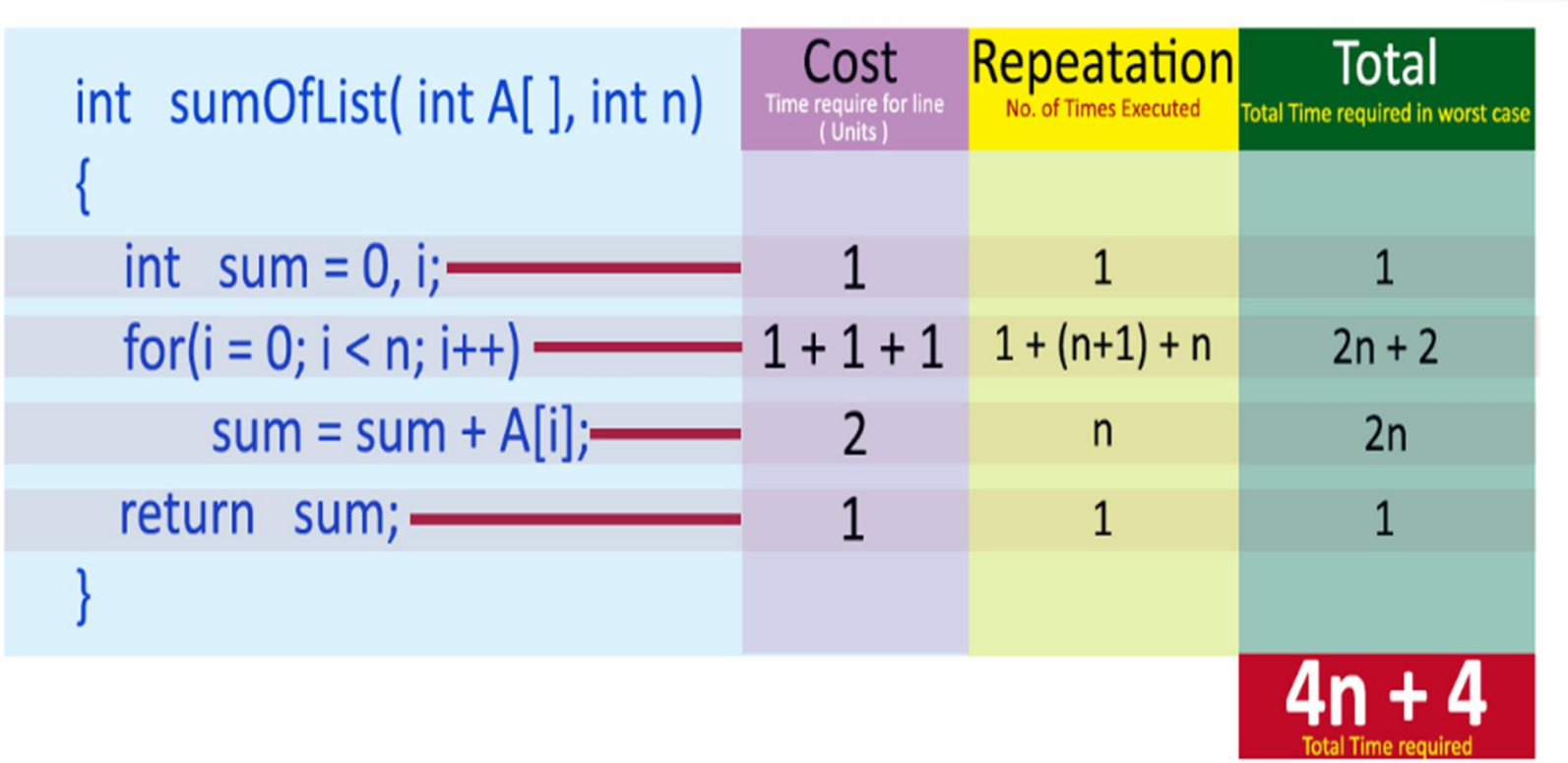
Analyzing Time Complexity

Example 5:

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

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 $O(N)$.


int sumOfList(int A[ ], int n)
{
int sum = 0, i;
for(i = 0; i < n; i++)
sum = sum + A[i];
return sum;
}
Cost
Time require for line (Units)
1
1 + 1 + 1
2
1
Repeatation
No. of Times Executed
1
1 + (n+1) + n
n
1
Total
Total Time required in worst case
1
2n + 2
2n
1
4n + 4
Total Time required

Notations for Complexity

BIG O Notation: O

- a dominant factor in the expression is sufficient to determine the order of the magnitude of the result
- O stands for 'order of'
- When using the Big O notation, constant multipliers are ignored
- If f(n) and g(n) are the functions defined on a positive integer number n, then

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

if and only if positive constants c and n exist,

$$f(n) <= cg(n).$$

- TIGHT UPPER BOUND

BIG O Notation: O

Constant “c” which depends upon the following factors

- the programming language used,
- the quality of the compiler or interpreter,
- the CPU speed,
- the size of the main memory and the access time to it,
- the knowledge of the programmer, and
- the algorithm itself, which may require simple but also time-consuming machine instructions.

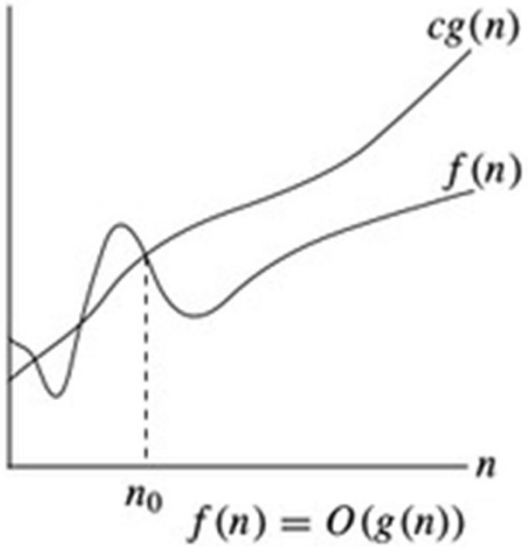
- $f(n) = O(g(n))$

BIG O Notation: O

g(n)	f(n) = O(g(n))
10	O(1)
$2n^3 + 1$	$O(n^3)$
$3n^2 + 5$	$O(n^2)$
$2n^3 + 3n^2 + 5n - 10$	$O(n^3)$

Example

- $O(n^3)$ will include ➔ $n^{2.9}$, n^3, $n^3 + n$, $540n^3 + 10$.
- $O(n^3)$ will not include ➔ $n^{3.2}$, n^2, $n^2 + n$, $540n + 10$, $2n$



(b)

BIG O Notation: O

- Best case O describes an upper bound for all combinations of input.
- It is possibly lower than the worst case.
- For example, when sorting an array the best case is when the array is already correctly sorted.
- Worst case O describes a lower bound for worst case input combinations. It is possibly greater than the best case.
- For example, when sorting an array the worst case is when the array is sorted in reverse order.
- **If we simply write O, it means same as worst case O.**

Limitations of Big O Notation

- Many algorithms are simply too hard to analyse mathematically.
- There may not be sufficient information to calculate the behavior of the algorithm in the average case.
- **Big O analysis only tells us how the algorithm grows with the size of the problem**, not how efficient it is, as it does not consider the programming effort.
- It ignores important constants.
- For example, if one algorithm takes $O(n^2)$ time to execute and the other takes $O(100000n^2)$ time to execute, then as per Big O, both algorithm have equal time complexity, but this may be a serious consideration.

Categories of Algorithms

According to the Big O notation, we have five different categories of algorithms:

- **Constant time algorithm: O(1)**
- **Linear time algorithm: O(n)**
- **Logarithmic time algorithm: as O(log n)**
- **Polynomial time algorithm: $O(n^k)$ where k > 1**
- **Exponential time algorithm: as $O(2^n)$**

OMEGA NOTATION (Ω)

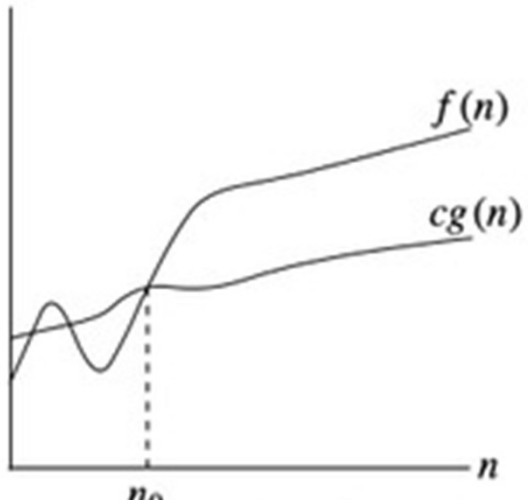
- provides a tight lower bound for f(n)
- Ω notation is simply written as, $f(n) \in \Omega(g(n))$
- $\Omega(g(n)) = \{h(n): \exists$ positive constants $c > 0$, n0 such that

$$0 \leq cg(n) \leq h(n), \forall\ n \geq n0\}.$$

- If $cg(n) \leq f(n)$, $c > 0$, $\forall\ n \geq n_0$, then $f(n) \in \Omega(g(n))$ and g(n) is an asymptotically tight lower bound for f(n).

- Examples of functions in $\Omega(n^2)$ include: n^2, $n^{2.9}$, $n^3 + n^2$, n^3
- Examples of functions <u>not</u> in $\Omega(n^3)$ include: n, $n^{2.9}$, n^2

- Best case Ω describes a lower bound for all combinations of input
- Worst case Ω describes a lower bound for worst case input combinations
- **If we simply write Ω, it means same as <u>best case</u> Ω.**



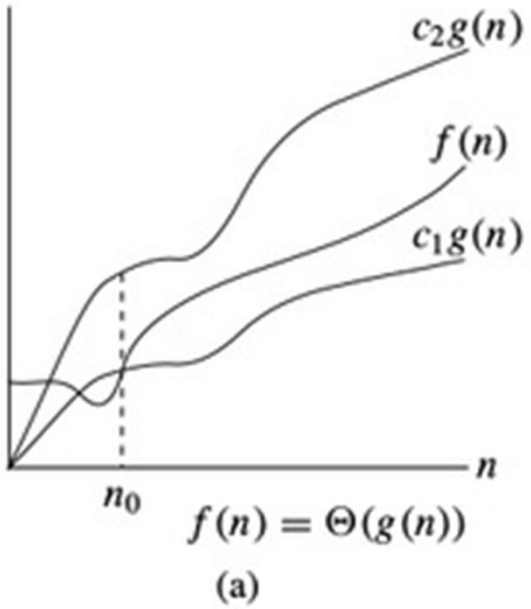
(c)

THETA NOTATION (Θ)

- Theta notation provides an asymptotically tight bound for f(n).
- Θ notation is simply written as, $f(n) \in \Theta(g(n))$
- $\Theta(g(n)) = \{h(n): \exists$ positive constants c_1, c_2, and n_0 such that $0 \leq c_1 g(n) \leq h(n) \leq c_2 g(n), \forall\ n \geq n_0\}$.

- The best case in Θ notation is not used.
- • Worst case Θ describes asymptotic bounds for worst case combination of input values.
- • If we simply write Θ, it means same as worst case Θ.

Time Complexity



Space Complexity

- When we design an algorithm to solve a problem, it needs some computer memory to complete its execution. For any algorithm, memory is required for the following purposes...
- To store program instructions.
- To store constant values.
- To store variable values.
- And for few other things like funcion calls, jumping statements etc,.
- Space complexity of an algorithm can be defined as follows...

Total amount of computer memory required by an algorithm to complete its execution is called as space complexity of that algorithm.

Space Complexity

Generally, when a program is under execution it uses the computer memory for THREE reasons. They are as follows...

- **Instruction Space:** It is the amount of memory used to store compiled version of instructions.
- **Environmental Stack:** It is the amount of memory used to store information of partially executed functions at the time of function call.
- **Data Space:** It is the amount of memory used to store all the variables and constants.

Space Complexity : Example

```
int square(int a)
{
        return a*a;
}
```

Memory Required is 2*size of int

```
int sum(int A[ ], int n)
{
   int sum = 0, i;
   for(i = 0; i < n; i++)
      sum = sum + A[i];
   return sum;
}
```

Memory Required is n*size of int + 4