Asymptotic Analysis:

This is the theoritical way to find the time complexity by finding the order of growth of an Algorithm or Program. It doesn't depend on any compiler, Machine, Programming Language, Here, We Measure order of growth. We don't have to implement this, We can simply analyze algorithms.

Example:

1. **sum of n natural numbers:**

// 1st Approach:

```
#include<bits/stdc++.h>
using namespace std;
int main()
{
       int n;
       cin >> n;
       cout << (n*(n+1))/2 << endl;
     return 0;
 }
```

Here, The order of growth = **c1*n** (where c1 assumed as constant)

Overall order of growth = n

Average case Time Complexity: **Θ(1)**

// 2nd Approach:

```
#include<bits/stdc++.h>
using namespace std;
int main()
{
    int n;
    cin >> n;
    int sum = 0;
    for(int i = 1; i<=n; i++){
    sum = sum+i;
}
cout << sum << endl;
    return 0;
}
```

Here the Order of growth = c1 + c2*(n) = n
Overall order of growth = n

Average case Time Complexity = $\Theta(n)$

Worse case Time Complexity = $O(n)$

// 3rd Approach:

```
#include<bits/stdc++.h>
using namespace std;

int32_t main()
{
int n;
cin >> n;//3
int sum = 0;

    for(int i = 1; i<=n; i++){          //n times
        for(int j = 1; j<=i; j++){      //n times
            sum++; //1+(1+1)+(1+1+1)
        }
        cout << sum << endl;
    }
    return 0;
}
```

Here, The order of growth = c1 + c2*(n)^2

Overall order of growth = n^2

Average case Time Complexity = $\Theta(n^2)$

Worse case Time Complexity = $O(n^2)$

1st approach - Time taken = c1 - constant order of growth

2nd approach - Time taken = c1+c2*n - linear order of growth
overall order of growth = n

3rd approach - Time taken = c1 + c2*(n*n) - quadratic order of growth
overall order of growth = n^2

Direct way to find the order of growth

1.) Ignore lower order Terms.

2.) Ignore Leading Terms Constant.

How do we compare terms?

CHART:

c < log(log(n)) < log n < (n)^(1/3) < (n)^(1/2) < (n) < (n)^2 <

(n)^3 < (n)^4 < (2)^n < (n)^n

NOTE: n < nlogn

Asymptotic Notations:

Asymptotic notations are mathematical tools to represent the time complexity of algorithms for asymptotic analysis. The following 3 asymptotic notations are mostly used to represent the time complexity of algorithms:

1.) Theta Notation Θ() :

It is used to find the Exact Bound.A simpler way to get Theta notation of an expression is to ignore low order terms and ignore leading constants.

2.) Omega Notation :

It is used to find the Exact or Lower Bound. A simpler way to get Omega notation of an expression is to ignore low order terms and ignore leading constants.

3.) Big O Notation O():

It is used to find the Exact or Upper Bound. A simpler way to get Big O notation of an expression is to ignore low order terms and ignore leading constants.

Worst Case | Best Case | Average Case Time Complexity

- In Order to find the Worst Case Time Complexity, We can use Big O Notation, that shows the upper bound of an Algorithm or Program.

- Similarly, For the Average case Time complexity, We can use theta Notation, that shows the exact bound of an Algorithm or Program.

- For the Best case Time complexity, We can use Omega Notation, that shows the lower bound of an Algorithm or program.

NOTE:
Mostly We are interested to know the upper or exact bound of any algorithm.

Important Points:

- Most of the times, we do the worst case analysis to analyze algorithms. In the worst analysis, we guarantee an upper bound on the running time of an algorithm which is a good piece of information.

- The average case analysis is not easy to do in most of the practical cases and it is rarely done. In the average case analysis, we must know (or predict) the mathematical distribution of all possible inputs.

- The Best Case analysis is bogus. Guaranteeing a lower bound on an algorithm doesn't provide any information as in the worst case, an algorithm may take years to run.

Some Important Things which we have to keep in mind.

Analysis of Loops:

O(Log n):

The Time Complexity of a loop is considered as O(Logn) if the

loop variables is divided / multiplied by a constant amount.

Example:

NOTE: C is considered as constant and n is the input variable.

```
for (int i = 1; i <=n; i *= c) {
        // constant work
}
```

```
for (int i = n; i > 0; i /= c) {
        // constant work
}
```

O(n):

The Time complexity of Loop is considered as O(n) if the loop variables is incremented / decremented by a constant amount.

Example:

NOTE: C is considered as constant and n is the input variable.

```
for (int i = 1; i <=n; i++) {
        // constant work
}
```

```
for (int i = n; i > 0; i--) {
        // constant work
}
```

O(n^2):

The Time complexity of Loop is considered as O(n^2), if nested loops is equal to the number of times the innermost statement is executed.

Example:

NOTE: C is considered as constant and n is the input variable.

```
for (int i = 1; i <=n; i++) {
```

```
        for (int j = 1; j <=n; j++) {
            // constant work
        }
}
```

```
for (int i = n; i > 0; i--) {
        for (int j = n; j > 0; j--) {
            // constant work
        }
}
```

O(1):

The Time complexity of Loop is considered as O(1) if the loop runs a constant number of times.

Example:

NOTE: C is considered as constant.

```
for (int i = 1; i <=c; i++) {
        // constant work
}
```

```
for (int i = c; i > 0; i--) {
        // constant work
}
```

O(Log(Log n))

Time Complexity of a loop is considered as **O(Log(Log n))** if the loop variables is incremented / decremented exponentially by a constant amount.

Example:

NOTE: C is considered as constant which is always greater than 1. and n is considered as input variable.

```
for (int i = 2; i <=n; i = pow(i,c)) {
        // constant work
```

}

//Suppose there is a function that returns the square root of a number.

//Suppose there is a function that returns the cube root of a number.

//Suppose there is any other function that returns the constant root of a number.

```
for (int i = n; i > 1; i = fun(i)) {
    // constant work
}
```

That’s all for Now.