Basics of Time Complexity

○

In this tutorial, we'll learn how to calculate time complexity of a function execution with examples.

Time Complexity

Time complexity is generally represented by big-oh notation O.
If time complexity of a function is (n), that means function will take n unit of time to execute.

These are the general types of time complexity which you come across after the calculation:-

Type	Name
O (1)	Constant
O ($\log_2 n$)	Logarithmic
O ($\sqrt{n}$	Root
O (n)	Linear
O (n^2)	Quadratic
O (n^3)	Cubic
O (2^n)	Exponential
O (n^n)	Exponential

Time Complexity in the increasing order of their value:-

$1 < \log_2 n < \sqrt{n} < n < n\log_2 n < n^2 < n^3 \ldots < 2^n < 3^n \ldots < n^n$

Time Complexity Calculation

We are going to understand time complexity with loads of examples:-

`for` loop

Let's look at the time complexity of `for` loop with many examples, which are easier to calculate:-

Example 1

```
for(int i=0; i<n; i++){}
```

loop run `n` times
hence Time Complexity = O(n)

Example 2

```
for(int i=0; i<n; i=i+2){}
```

loop run `n/2` times which is still linear
hence Time Complexity = O(n)

Example 3

```
for(int i=n; i>1; i--){}
```

loop run `n` times
hence Time Complexity = O(n)

Example 4

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

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

two individual loops run `n+n = 2n` times which is still linear
hence Time Complexity = O(n)

Example 5

```
for(int i=0; i<n; i++){
    for(int j=0; j<n; j++){}

}
```

Nested loop run `nxn = n²` times which is non-linear
hence Time Complexity = $O(n^2)$

Example 6

```
for(int i=1; i<n; i=i*2){}
```

values of \`i\` in \`for\` loop
1
2
2^2
2^3
.
.
2^k

The loop will terminate when:-
$i \geq n$
$2^k \geq n$
$k \geq \log_2(n)$
so $\log_2(n)$ is total unit of time taken by the loop

hence Time complexity = $O(\log_2(n))$

Example 7

```
for(int i=1; i<n; i=i*3){}
```

Calculation:

Similar to the last example,
value of \`i\` in \`for\` loop is $i = 3^k$

The loop will terminate when:-
$i \geq n$
$3^k \geq n$
$k \geq \log_3 n$

so $\log_3 n$ is total unit of time taken by the loop

hence Time complexity = $O(\log_3 n)$

Example 8

```
for(int i=n; i>1; i=i/2){}
```

Calculation:
values of `i` in `for` loop
n
$n/2$
$n/2^2$
$n/2^3$
.
.
$n/2^k$

The loop will terminate when:-
$i \lesssim 1$
$n/2^k \lesssim 1$
$2^k \geq n$
$k \geq \log_2(n)$
so $\log_2(n)$ is total unit of time taken by the loop

hence Time complexity = $O(\log_2(n))$

Example 9

```
for(int i=0; i<n; i++){
    for(int j=1; j<n; j=j*2){}
}
```

Calculation:

outer loop complexity = O(n)
inner loop complexity = $O(\log_2(n))$

hence Time complexity = $O(n\log_2(n))$

Example 10

```
int p = 0;
for(int i=1; p<=n; i++){
    p = p + i;
}
```

Calculation:
value of `i` and `p` in `for` loop:-
i   p

1   0+1=1
2   1+2=3
3   1+2+3=6
4   1+2+3+4
.   .
.   .
k   1+2+3+4+...+k
k   k(k+1)/2
————————————————————————————

The loop will terminate when:-
p   > n
k(k+1)/2 > n

$k^2 \quad > \; n$
$k \quad > \sqrt{n}$
so $\sqrt{n}$ is total unit of time taken by the loop

hence Time complexity = $O(\sqrt{n})$

Example 11

```
int p = 0;
for(int i=1; i<n; i=i*2){
    p++;
}
for(int j=1; j<p; j=j*2){
    // statement
}
```

Calculation:

1. first `for` loop will take $\log_2(n)$ unit of time to execute.
   At the end of first loop value of p = $\log_2(n)$

2. second `for` loop will take $\log_2(p)$ unit of time to execute.

hence Time Complexity = $O(\log_2(p)) = O(\log_2\log_2(n))$

`while` loop

If you understand how to calculate the time complexity of `for` loop then `while` loop is piece of cake.

Example 1

```
int i=1;
while(i<n){
    //statement
    i=i*2;
}
```

Time Complexity = $(\log_2(n))$

Example 2

```
int i=n;
while(i>1){
    //statement
    i=i/2;
}
```

Time Complexity = $(\log_2(n))$

Example 3

```
int i=1;
int k=1;
while(k<n){
    //statement
    k=k+i;
    i++;
}
```

Time Complexity = (√n)

Variable Time Complexity

It is not necessary that function always take fixed unit of time to execute, sometime it depends on the input parameters. Here are some examples where time complexity is not fixed:-

Example 1

```
method(n, m){
    while(n!=m)
 {
        if(n>m){
            n = n-m;
        }else{
            m = m-n;
        }
    }
}
```

best case time = $O(1)$
(when n = m)

```
worst case time = O(n)
(when n is very larger then m (e.g. n=16, m=1))
```

Example 2

```
method(n){
    if(n<5){
        //statement
    }else{
        for(i=0;i<n;i++){
            //statement
        }
    }
}
```

```
best case time = O(1)
(when n < 5)

worst case time = O(n)
(when n ≥ 5)
```

Recursive Functions

Let's see the time complexity of recurring (or recursive) functions:-

Example 1

```
test(int n){
    if(n>0){
        //statement
        test(n-1);
    }
}
// T(n) = T(n-1) + 1 = O(n)
```

Calculation:

Base case
T(0) = 1

Time taken by nth task is time taken by (n-1)th task plus 1
T(n) = T(n-1) + 1   --(1)

Similarly time taken by (n-1)th task is (n-2)th task plus 1
T(n-1) = T(n-2) + 1 --(2)
T(n-2) = T(n-3) + 1 --(3)

T(n) = T(n-3) + 3   --after substituting (2),(3) in (1)
T(n) = T(n-k) + n
...
Assume (n-k)th is 0th task means n=k
T(n) = T(0) + n
T(n) = 1 + n ≌ n

hence Time Complexity = O(n)

Example 2

```
test(int n){
    if(n>0){
        for(int i=0; i<n; i++){
            //statement
        }
        test(n-1);
    }
}







// T(n) = T(n-1) + n = O(n²)
```

Calculation:

Base case
$T(0) = 1$

Time taken by nth task:-
$T(n) = T(n-1) + n$
$T(5)=T(4)+5$
$T(n-1) = T(n-2) + n-1$
$T(4)=T(3)+4$
$T(n-2) = T(n-3) + n-2$
..
..
$T(n) = T(n-3) + (n-2) + (n-1) + n$
$T(n) = T(n-k) + (n-(k-1)) + (n-(k-2)) + \dots + (n-1) + n$

Assume (n-k)th is 0th task means n=k
$T(n) = T(n-n) + (n-n+1) + (n-n+2) + \dots + (n-1) + n$
$T(n) = T(0) + 1 + 2 + 3 + \dots + n$
$T(n) = 1 + n(n+1)/2 \backsimeq n^2$

hence Time Complexity = $O(n^2)$

Example 3

```
test(int n){
    if(n>0){
        for(int i=0; i<n; i=i*2){
            //statement
        }
        test(n-1);
    }
}
// T(n) = T(n-1) + log₂(n) = O(nlog₂(n))
```

Calculation:

Base case

$T(0) = 1$

Time taken by nth task:-

$T(n) = T(n-1) + \log_2(n)$

$T(n-1) = T(n-2) + \log_2(n-1)$

$T(n-2) = T(n-3) + \log_2(n-2)$

..

..

$T(n) = T(n-3) + \log_2(n-2) + \log_2(n-1) + \log_2(n)$

$T(n) = T(n-k) + \log_2 1 + \log_2 2 + \ldots + \log_2(n-1) + \log_2(n)$

Assume (n-k)th is 0th task means n=k

$T(n) = T(0) + \log_2(n)!$

$T(n) = 1 + n\log_2(n) \backsimeq n\log_2(n)$

hence Time Complexity = $O(n\log_2(n))$

Example 4

```
test(int n){
    if(n>0){
        //statement
        test(n-1);
        test(n-1);
    }
}
// T(n) = 2T(n-1) + 1 = O(2ⁿ)
```

Calculation:

Base case
$T(0) = 1$

Time taken by nth task:-
$T(n) = 2T(n-1) + 1$
$T(n-1) = 2T(n-2) + 1$
$T(n-2) = 2T(n-3) + 1$
..
..
$T(n) = 2[2[2T(n-3) + 1] + 1] + 1$
$T(n) = 2^3T(n-3) + 2^2 + 2 + 1$
$T(n) = 2^kT(n-k) + 2^{k-1} + 2^{k-2} + .. + 2^2 + 2 + 1$

Assume (n-k)th is 0th task means n=k
$T(n) = 2^nT(0) + (2^n-1)$
$T(n) = 2^n + (2^n-1) \backsimeq 2^n$

hence Time Complexity = $O(2^n)$

Example 5

```
test(int n){
    if(n>0){
        //statement
        test(n-1);
        test(n-1);
        test(n-1);
    }
}
```

T(n) = 3T(n-1) + 1

Time Complexity = (3^n)

Example 6

```
test(int n){
    if(n>0){
        for(int i=0; i<n; i++){
            //statement
        }
        test(n-1);
        test(n-1);
    }
}
//
```

T(n) = 2T(n-1) + n

Time Complexity = $(n2^n)$

Example 7

```
test(int n){
    if(n>0){
        //statement
        test(n/2);
    }
}
```

Base case
$T(1) = 1$

$T(n) = T(n/2) + 1$
$T(n/2) = T(n/2^2) + 1$
$T(n) = [T(n/2^2) + 1] + 1$
$T(n) = T(n/2^k) + k$

Assume $(n/2^k)$th is last task means
$n/2^k = 1$
$2^k = n$
$k = \log_2(n)$
$T(n) = T(1) + \log_2(n) = 1 + \log_2(n) \backsimeq \log_2(n)$

hence Time Complexity = $O(\log_2(n))$

Example 8

```
test(int n){
    if(n>0){
        for(int i=0; i<n; i++){
            //statement
        }
        test(n/2);
```

```
    }
}
```

Base case
$T(1) = 1$

$T(n) = T(n/2) + n$
$T(n/2) = T(n/2^2) + n/2$
$T(n) = [T(n/2^2) + n/2] + n$
$T(n) = T(n/2^k) + n/2^{k-1} + n/2^{k-2} + .. + n/2^2 + n/2 + n$

Assume $(n/2^k)$th is last task means
$n/2^k = 1$
$2^k = n$
$k = \log_2(n)$
$T(n) = T(1) + n[1/2^{k-1} + 1/2^{k-2} + .. + 1/2 + 1]$
$T(n) = 1 + n[1 + 1] = 1 + 2n \backsimeq n$

hence Time Complexity = $O(n)$

Example 9

```
test(int n){
    if(n>0){
        //statement
        test(n/2);
        test(n/2);
    }
}
```

$T(n) = 2T(n/2) + 1$
$T(n) = 2^kT(n/2^k) + k + k-1 + k-2 + ... + 1$

Assume $n/2^k = 1$ means $k = \log_2(n)$

$T(n) = nT(1) + k(k+1)/2 \backsimeq n + (\log_2(n))^2 \backsimeq n$

hence Time Complexity = $O(n)$

Example 10

Quick Sort when pivot is middle element:-

```
quickSort(int[] arr, int low, int high) {
    if (low < high){
        int pi = partition(arr, low, high);  // n
        quickSort(arr, low, pi - 1);         // T(n/2)
        quickSort(arr, pi + 1, high);        // T(n/2)
    }
}
```

Base case

$T(1) = 1$

$T(n) = 2T(n/2) + n$

$T(n/2) = 2T(n/2^2) + n/2$

$T(n) = 2[2T(n/2^2) + n/2] + n$

$T(n) = 2^k T(n/2^k) + n + n + ... + n$

$T(n) = 2^k T(n/2^k) + nk$

Assume $(n/2^k)$th is last task means

$n/2^k = 1$

$2^k = n$

$k = \log_2(n)$

$T(n) = nT(1) + n\log_2(n)$

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

Time Complexity = $(n\log_2(n))$

Asymptotic Notations

We can represent the function complexity in following ways:-

Symbol	Name	Bound
O	big-oh	upper bound
Ω	big-omega	lower bound
ϴ	big-theta	average bound

Example 1

For e.g. $f(n) = 2n + 3$
can be represented as
(n) or any notation with higher weightage such as $O(n\log_2 n)$ or $O(n^2)$ or $O(n^3)$ ...
$\Omega(n)$ or any notation with lower weightage such as $\Omega(\sqrt{n})$, $\Omega(\log_2 n)$, $\Omega(1)$ ...
$\Theta(n)$ and only $\Theta(n)$ since this is average bound

Ideally you represent the function complexity to nearest type of complexity so in above case (n), $\Omega(n)$, $\Theta(n)$ are best representations.

Example 2

$f(n) = 2n^2 + 3n + 4$
$1n^2 \leq 2n^2 + 3n + 4 \leq 9n^2$
can be represented as $O(n^2)$, $\Omega(n^2)$, or $\Theta(n^2)$

Example 3

$f(n) = n^2\log_2 n + n$
$n^2\log_2 n \leq 2n^2\log_2 n + n \leq 3n^2\log_2 n$
can be represented as $O(n^2\log_2 n)$, $\Omega(n^2\log_2 n)$, or $\Theta(n^2\log_2 n)$

Example 4

$f(n) = !n = nx(n-1)x(n-2)x \ldots x2x1 = n^n$
$n \leq !n \leq n^n$
can be represented as $O(n^n)$ upper-bound, $\Omega(n)$ lower-bound
can not be represented as Θ since there is no common average-bound.

Example 5

$f(n) = \log!n = \log(nx(n-1)x(n-2)x \ldots x2x1) = \log(n^n) = n\log(n)$
$1 \leq \log!n \leq n\log(n)$
can be represented as $O(n\log(n))$ upper-bound, $\Omega(1)$ lower-bound
can not be represented as Θ since there is no common average-bound.

- It is always preferable to represent complexity in big-theta Θ, if possible, which is more accurate and tight bound.
- Big-oh (n) is the most popular notation to represent function complexity which you come across.

Note: Do not mix these notations with best case, worst case, or average case time complexity. All type of cases can be represented by O, Ω, and Θ notations.