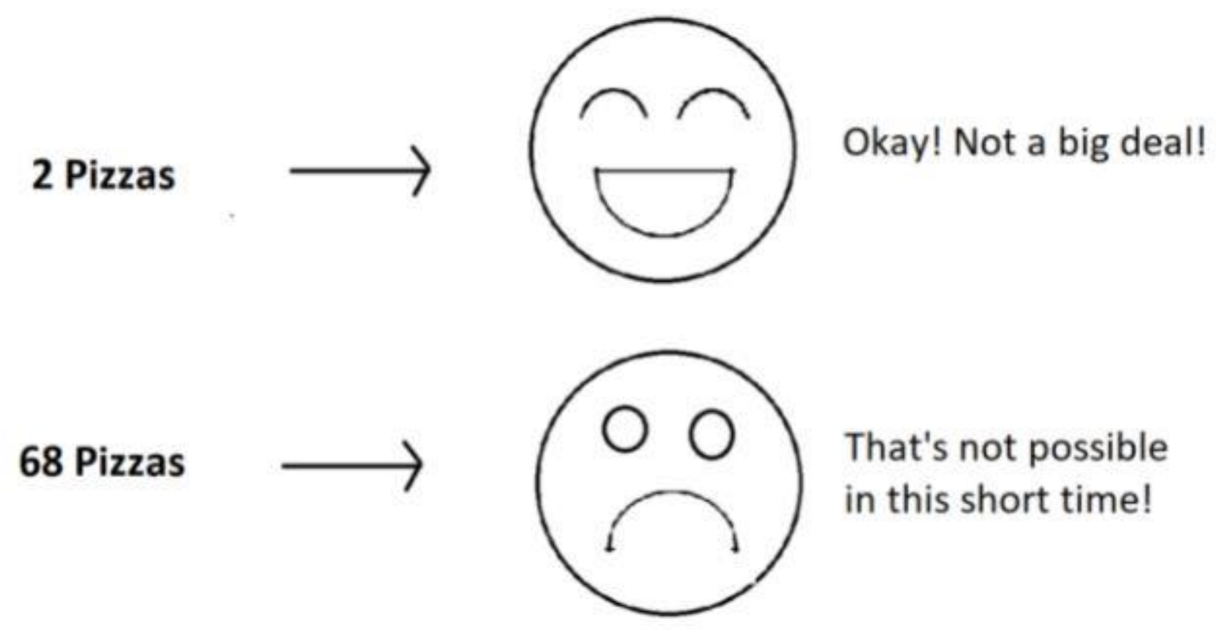
Time Complexity and Big O Notation (with notes)

An analogy to a real-life issue:

- This morning I wanted to eat some pizza; So, I asked my brother to get me some from Dominos, which is 3 km away.
- He got me the pizza, and I was happy only to realize it was too little for 29 friends who came to my house for a surprise visit!
- My brother can get 2 pizzas for me on his bike, but pizza for 29 friends is too huge of an input for him, which he cannot handle.



What is Time Complexity?

Time Complexity is the study of the efficiency of algorithms. It tells us how much time is taken by an algorithm to process a given input. Let's understand this concept with the help of an example:

Consider two developers Shubham and Rohan, who created an algorithm to sort 'n' numbers independently. When I made the program run for some input size n, the following results were recorded:

No. of elements (n)	Time Taken By Shubham's Algo	Time Taken By Rohan's Algo
10 elements	90 ms	122 ms
70 elements	110 ms	124 ms
110 elements	180 ms	131 ms
1000 elements	2s	800 ms

We can see that at first, Shubham's algorithm worked well with smaller inputs; however, as we increase the number of elements, Rohan's algorithm performs much better.

Quick Quiz: Who's algorithm is better?

Time Complexity: Sending GTA 5 to a friend:

- Imagine you have a friend who lives 5 km away from you. You want to send him a game. Since the final exams are over and you want him to get this 60 GB file worth of game from you. How will you send it to him in the shortest time possible?
- Note that both of you are using JIO 4G with a 1 Gb/day data limit.
- The best way would be to send him the game by delivering it to his house. Copy the game to a hard disk and make it reach him physically.
- Would you do the same for sending some small-sized game like MineSweeper which is in KBS of size? Of Course no, because you can now easily send it via the Internet.
- As the file size grows, the time taken to send the game online increases linearly – O(n) while the time taken by sending it physically remains constant. O(n0) or O(1).

Calculating Order in terms of Input Size:

In order to calculate the order(time complexity), the most impactful term containing n is taken into account (Here n refers to Size of input). And the rest of the smaller terms are ignored.

Let us assume the following formula for the algorithms in terms of input size n:

Algo 1: $$\underbrace{k_1n^2}_{\text{Highest Order Term}} + \underbrace{k_2n + 36}_{\text{Can ignore other lower order terms}} = O(n^2)$$

Algo 2: $$k_1k_2^2 + k_3k_2 + 8 = O(n^0) = O(1)$$

Here, we ignored the smaller terms in algo 1 and carried the most impactful term, which was the square of the input size. Hence the time complexity became n^2. The second algorithm followed just a constant time complexity.

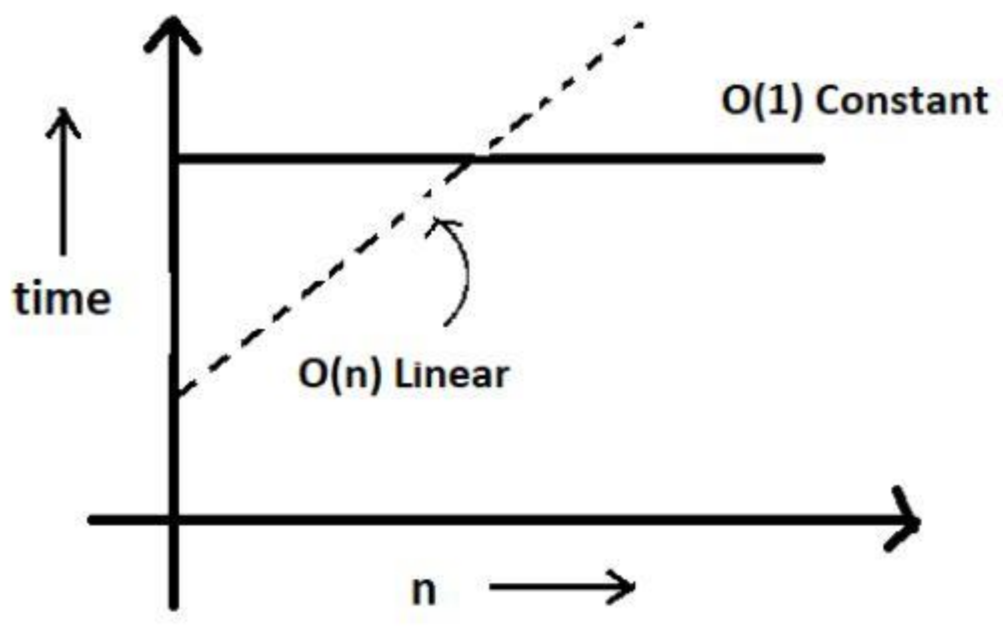
Note that these are the formulas for the time taken by their program.

What is a Big O?

Putting it simply, big O stands for 'order of' in our industry, but this is pretty different from the mathematical definition of the big O. Big O in mathematics stands for all those complexities our program runs in. But in industry, we are asked the minimum of them. So this was a subtle difference.

Visualizing Big O:

If we were to plot O(1) and O(n) on a graph, they would look something like this:



So, this was the basics of time complexities. You don't have to worry about things passing you by. We'll learn more deeply in the coming videos. For now, just stay with me.

Thank you for being with me throughout the tutorial. I hope you enjoyed it. If you appreciate my work, please let your friends know about this course too. Make sure to download the notes linked below. See you all in the next tutorial, where we'll learn more about time complexities, Big O, and other asymptotic notations. Till then, keep learning.

Asymptotic Notations: Big O, Big Omega and Big Theta Explained (With Notes)

Asymptotic notation gives us an idea about how good a given algorithm is compared to some other algorithm.

Now let's look at the mathematical definition of 'order of.' Primarily there are three types of widely used asymptotic notations.

1. Big oh notation (O)
2. Big omega notation (Ω)
3. Big theta notation (θ) – Widely used one

Big oh notation (O):

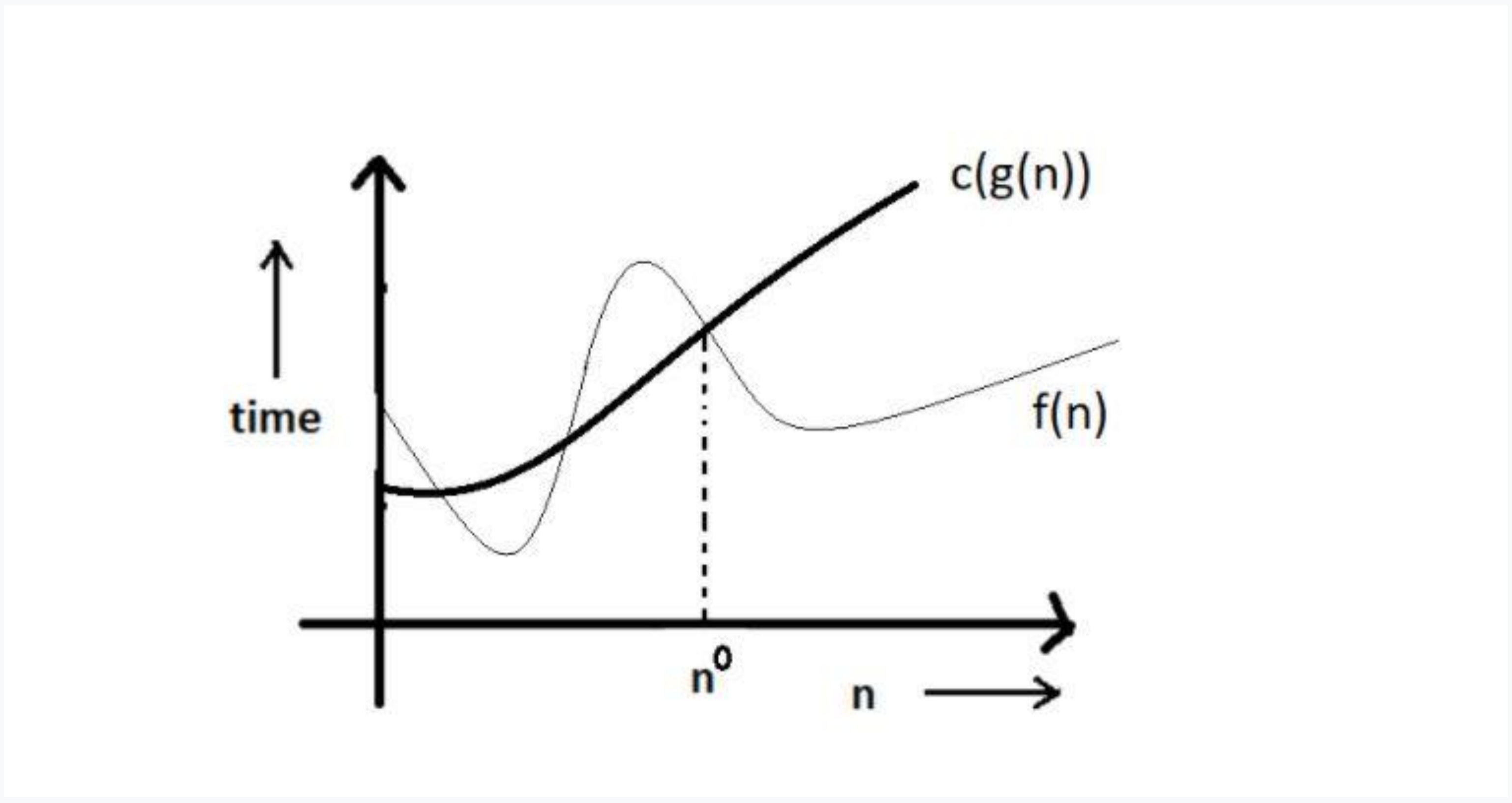
- Big oh notation is used to describe an asymptotic upper bound.
- Mathematically, if f(n) describes the running time of an algorithm; f(n) is O(g(n)) if and only if there exist positive constants c and n° such that:

```
0 ≤ f(n) ≤ c g(n)        for all n ≥ n°.
```

Copy

- Here, n is the input size, and g(n) is any complexity function, for, e.g. n, n2, etc. (It is used to give upper bound on a function)
- If a function is O(n), it is automatically O(n2) as well! Because it satisfies the equation given above.

Graphic example for Big oh (O):



Big Omega Notation (Ω):

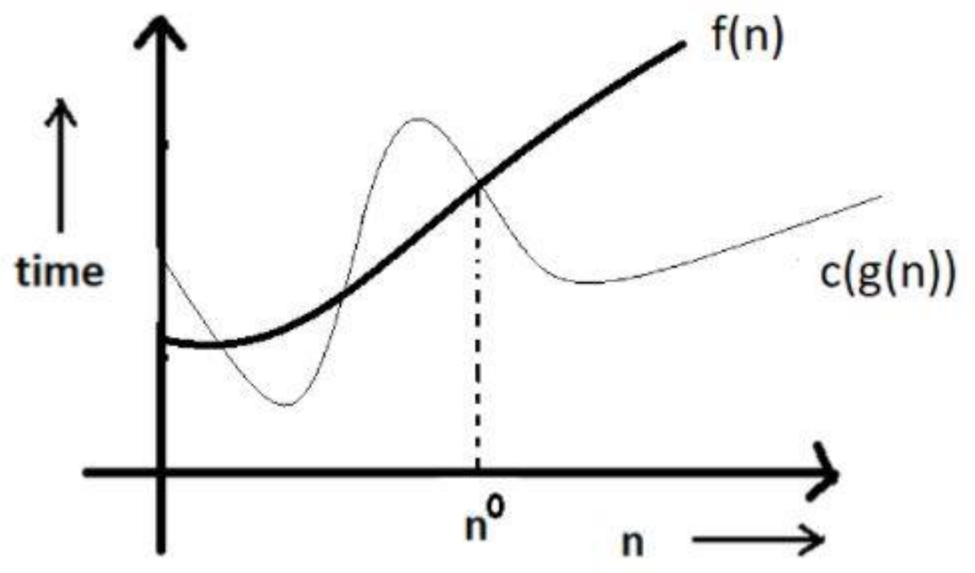
- Just like O notation provides an asymptotic upper bound, Ω notation provides an asymptotic lower bound.
- Let f(n) define the running time of an algorithm; f(n) is said to be Ω (g(n)) if and only if there exist positive constants c and n° such that:

```
0 ≤ c g(n) ≤ f(n)        for all n ≥ n°.
```

Copy

- It is used to give the lower bound on a function.
- If a function is Ω (n2) it is automatically Ω (n) as well since it satisfies the above equation.

Graphic example for Big Omega (Ω):



Big theta notation (θ):

- Let f(n) define the running time of an algorithm.
- F(n) is said to be θ (g(n)) if f(n) is O (g(n)) and f(x) is Ω (g(n)) both.

Mathematically,

$$0 \leq f(n) \leq c_1 g(n) \qquad \forall \quad n \geq n_o$$

$$0 \leq c_2 g(n) \leq f(n) \qquad \forall \quad n \geq n_o$$

for sufficiently large value of n

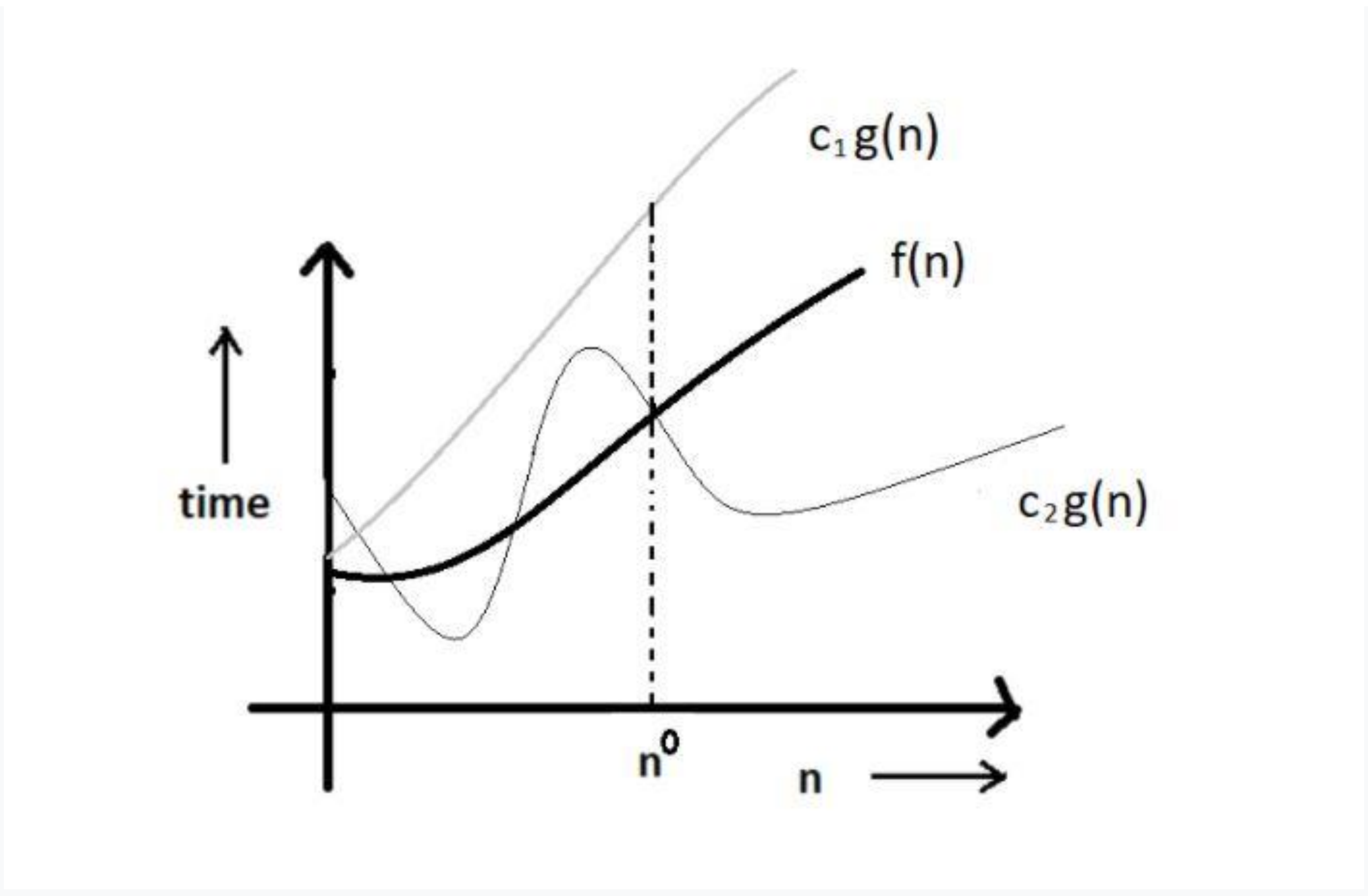
Merging both the equations, we get:

```
 0  ≤ c2 g(n)  ≤  f(n) ≤ c1 g(n)      ∀    n ≥ no.
```

Copy

The equation simply means that there exist positive constants c1 and c2 such that f(n) is sandwiched between c2 g(n) and c1 g(n).

Graphic example of Big theta (θ):



Which one of these to use?

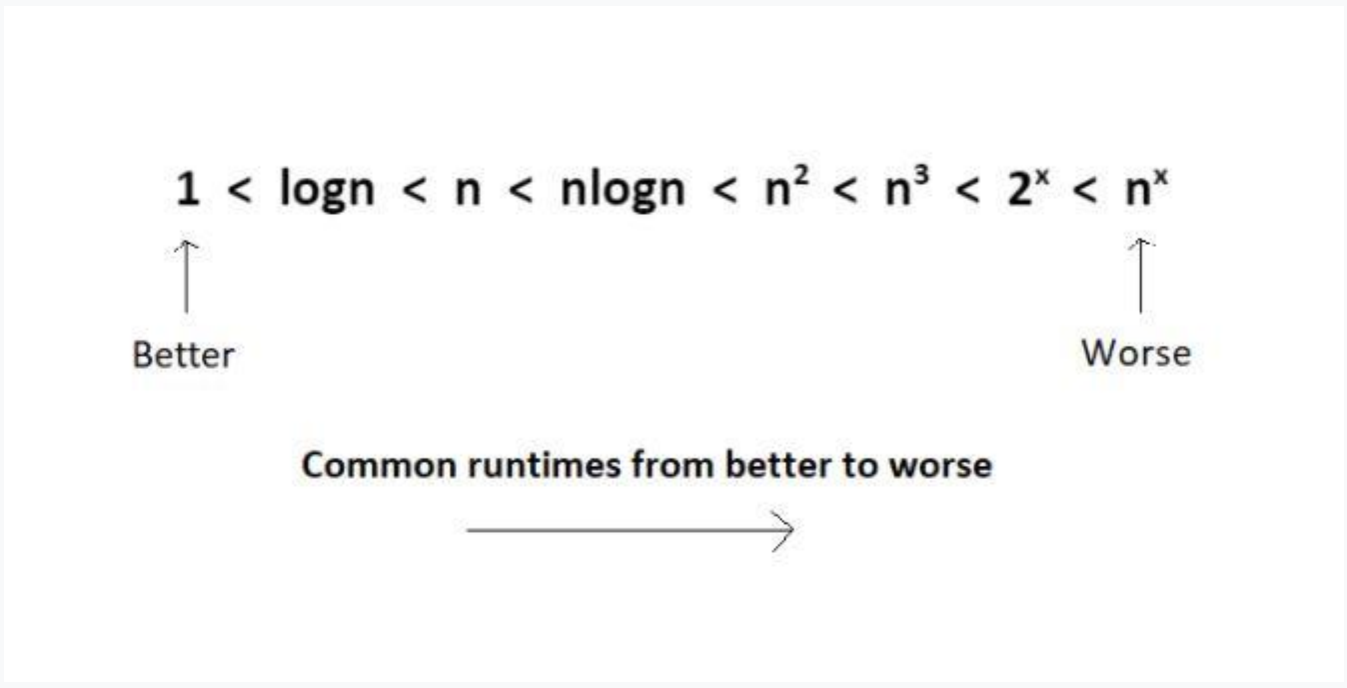
Big theta provides a better picture of a given algorithm's run time, which is why most interviewers expect you to answer in terms of Big theta when they ask "order of" questions. And what you provide as the answer in Big theta, is already a Big oh and a Big omega. It is recommended for this reason.

Quick Quiz: Prove that n2+n+1 is O(n3), Ω(n2), and θ(n2) using respective definitions.

Hint: You can approach this both graphically, making some rough graphs and mathematically, finding valid constants c1 and c2.

Increasing order of common runtimes:

Below mentioned are some common runtimes which you will come across in your coding career.



$1 < \log n < n < n\log n < n^2 < n^3 < 2^x < n^x$
Better
Worse
Common runtimes from better to worse

So, this was all about the asymptotic notations. We'll come across these a lot in future. However, there's no need for concern. Just stay with me.

Thank you for being with me throughout the tutorial. I hope you enjoyed it. If you appreciate my work, please let your friends know about this course too. Make sure to download the notes linked below. If you haven't checked out the whole playlist yet, move on to codewithharry.com or my YouTube channel to access it. See you all in the next tutorial here we'll learn analysing an algorithm and differentiating them on the basis of their time complexities. Till then keep learning.

Best Case, Worst Case and Average Case Analysis of an Algorithm (With Notes)

Life can sometimes be lucky for us:

- Exams getting canceled when you are not prepared, a surprise test when you are prepared, etc. → **Best case**

Occasionally, we may be unlucky:

- Questions you never prepared being asked in exams, or heavy rain during your sports period, etc. → **Worst case**

However, life remains balanced overall with a mixture of these lucky and unlucky times. → **Expected case**

Those were the analogies between the study of cases and our everyday lives. Our fortunes fluctuate from time to time, sometimes for the better and sometimes for the worse. Similarly, a program finds it best when it is effortless for it to function. And worse otherwise.

By considering a search algorithm used to perform a sorted array search, we will analyze this feature.

Analysis of a search algorithm:

Consider an array that is sorted in increasing order.

1	7	18	28	50	180

We have to search a given number in this array and report whether it's present in the array or not. In this case, we have two algorithms, and we will be interested in analyzing their performance separately.

1. **Algorithm 1** – Start from the first element until an element greater than or equal to the number to be searched is found.
2. **Algorithm 2** – Check whether the first or the last element is equal to the number. If not, find the number between these two elements (center of the array); if the center element is greater than the number to be searched, repeat the process for the first half else, repeat for the second half until the number is found. And this way, keep dividing your search space, making it faster to search.

Analyzing Algorithm 1: (Linear Search)

- We might get lucky enough to find our element to be the first element of the array. Therefore, we only made one comparison which is obviously constant for any size of the array.
- Best case complexity = O(1)
- If we are not that fortunate, the element we are searching for might be the last one. Therefore, our program made 'n' comparisons.
- Worst-case complexity = O(n)

For calculating the average case time, we sum the list of all the possible case's runtime and divide it with the total number of cases. Here, we found it to be just O(n). (Sometimes, calculation of average-case time gets very complicated.)

Analyzing Algorithm 2: (Binary Search)

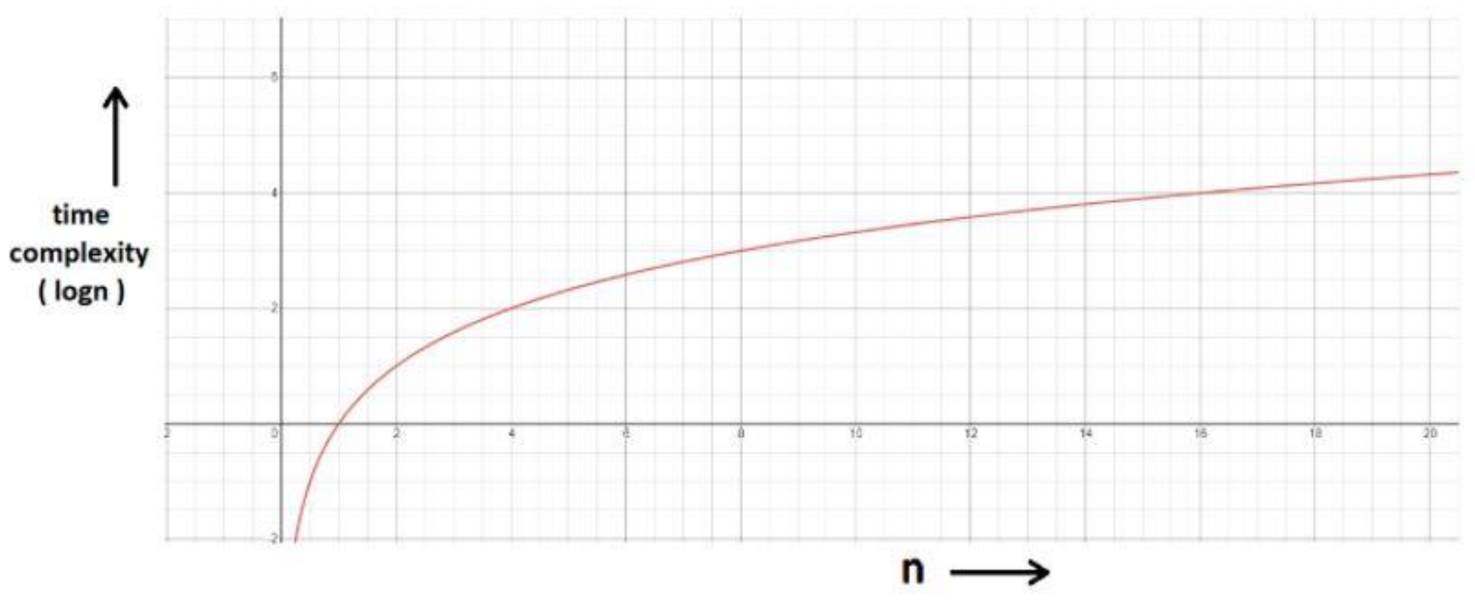
- If we get really lucky, the first element will be the only element that gets compared. Hence, a constant time.
- Best case complexity = O(1)
- If we get unlucky, we will have to keep dividing the array into halves until we get a single element. (that is, the array gets finished)
- Hence the time taken : n + n/2 +n/4 + ... + 1 = logn with base 2
- Worst-case complexity = O(log n)

What is log(n)?

Logn refers to how many times I need to divide n units until they can no longer be divided (into halves).

- log8 = 3 ⇒ 8/2 + 4/2 + 2/2 → Can't break anymore.
- log4 = 2 ⇒ 4/2 + 2/2 → Can't break anymore.

You can refer to the graph below, and you will find how slowly the time complexity (Y-axis) increases when we increase the input n (X-axis).



Space Complexity:

- Time is not the only thing we worry about while analyzing algorithms. Space is equally important.
- Creating an array of size n (size of the input) → O (n) Space
- If a function calls itself recursively n times, its space complexity is O (n).

Quiz Quiz: Calculate the space complexity of a function that calculates the factorial of a given number n.

Hint: Use recursion.

You might have wondered at some point why we can't calculate complexity in seconds when dealing with time complexities. Here's why:

- Not everyone’s computer is equally powerful. So we avoid handling absolute time taken. We just measure the growth of time with an increase in the input size.
- Asymptotic analysis is the measure of how time (runtime) grows with input.

Thank you for being with me throughout. I hope you enjoyed the tutorial. If you appreciate my work, please let your friends know about this course too. Make sure to download the notes linked below. If you haven’t checked out the whole playlist yet, move on to codewithharry.com or my YouTube channel to access it. See you all in the next tutorial, where we’ll learn how to calculate these time complexities with a few solved examples. Till then, keep learning.

How to Calculate Time Complexity of an Algorithm + Solved Questions (With Notes)

In previous videos, we had discussed what time complexity is and how it helps in dealing with what is most efficient for our programs. Our task today will be to find out how to calculate the time complexity of our programs. Here are some tips and tricks about the same, followed by a discussion of some questions.

Techniques to calculate Time Complexity:

Once we are able to write the runtime in terms of the size of the input (n), we can find the time complexity. For example:

```
 T(n) = n2 → O(n^2)
T(n) = logn → O(logn)
```
Copy

Here are some tricks to calculate complexities:

- **Drop the constants:**

  Anything you might think is O(kn) (where k is a constant) is O(n) as well. This is considered a better representation of the time complexity since the k term would not affect the complexity much for a higher value of n.

- **Drop the non-dominant terms: :**

  Anything you represent as O(n2+n) can be written as O(n2). Similar to when non-dominant terms are ignored for a higher value of n.

- **Consider all variables which are provided as input:**

  O (mn) and O (mnq) might exist for some cases.

In most cases, we try to represent the runtime in terms of the inputs which can even be more than one in number. For example,

The time taken to paint a park of dimension m * n → O (kmn) → O (mn)

Time Complexity – Competitive Practice Sheet:

Question1: Fine the time complexity of the *func1* function in the program shown in the snippet below:

```c
#include<stdio.h>

void func1(int array[], int length)
{
    int sum=0;
    int product =1;
    for (int i = 0; i <length; i++)
    {
        sum+=array[i];
    }

    for (int i = 0; i < length; i++)
    {
        product*=array[i];
    }
}

int main()
{
    int arr[] = {3,4,66};
    func1(arr,3);
    return 0;
}
```
Copy

Question 2: Find the time complexity of the *func* function in the program from program2.c as follows:

```c
void func(int n)
{
    int sum=0;
    int product =1;
    for (int i = 0; i <n; i++)
    {
        for (int j = 0; j < n; j++)
        {
            printf("%d , %d\n", i,j);
        }
    }
}
```
Copy

Question 3: Consider the recursive algorithm below, where the random(int n) spends one unit of time to return a random integer where the probability of each integer coming as random is evenly distributed within the range [0,n]. If the average processing time is T(n), what is the value of T(6)?

```
int function(int n)
{
    int i = 0;
    if (n <= 0)
    {
        return 0;
    }
    else
    {
        i = random(n - 1);
        printf("this\n");
        return function(i) + function(n - 1 - i);
    }
}
```

Copy

Question 4: Which of the following are equivalent to O(N) and why?

1. O(N + P), where P < N/9
2. 0(9N-k)
3. O(N + 8log N)
4. O(N + M2)

Question 5: The following simple code sums the values of all the nodes in a balanced binary search tree (don't worry about what it is, we'll learn them later). What is its runtime?

```
int sum(Node node)
{
    if (node == NULL)
    {
        return 0;
    }
    return sum(node.left) + node.value + sum(node.right);
}
```

Copy

Question 6: Find the complexity of the following code which tests whether a given number is prime or not?

```
int isPrime(int n)
{
    if (n == 1)
    {
        return 0;
    }
    for (int i = 2; i * i < n; i++)
    {
        if (n % i == 0)
        {
            return 0;
        }
    }
    return 1;
}
```

Copy

Question 7: What is the time complexity of the following snippet of code?

```
int isPrime(int n)
{
    for (int i = 2; i * i < 10000; i++)
    {
        if (n % i == 0)
        {
            return 0;
        }
    }

    return 1;
}
isPrime();
```

Copy

So, these were the few questions I felt like discussing. Try them on your own first. And if you have already given them enough thought, then move on to the video above where we have discussed their solutions in detail. Hopefully, they were able to make you learn how to find the time complexities for different programs.