Algorithm Analysis

In previous section we learn about what is algorithm? And Why algorithm is primary factor and What is the role of an algorithm in programming and much more. In this section I am going to teach you how to choose a best algorithm for any problem and how to compare two algorithm, how to calculate running time of a algorithm and asymptotic notations and much more.

What is Algorithm Analysis?

Analysis of an algorithm is the process of analyzing the problem-solving capability of the algorithm in a different factors like- running time, space consumption, code-implementation, computer hardware and much more. To trace the behaviour of an algorithm like- how much time is taken by an algorithm to execute a operation?. But in general we are mostly concern about the running time. Analyzing several algorithms can help us to find out the efficient algorithm for a problem.

Why the Analysis of Algorithms?

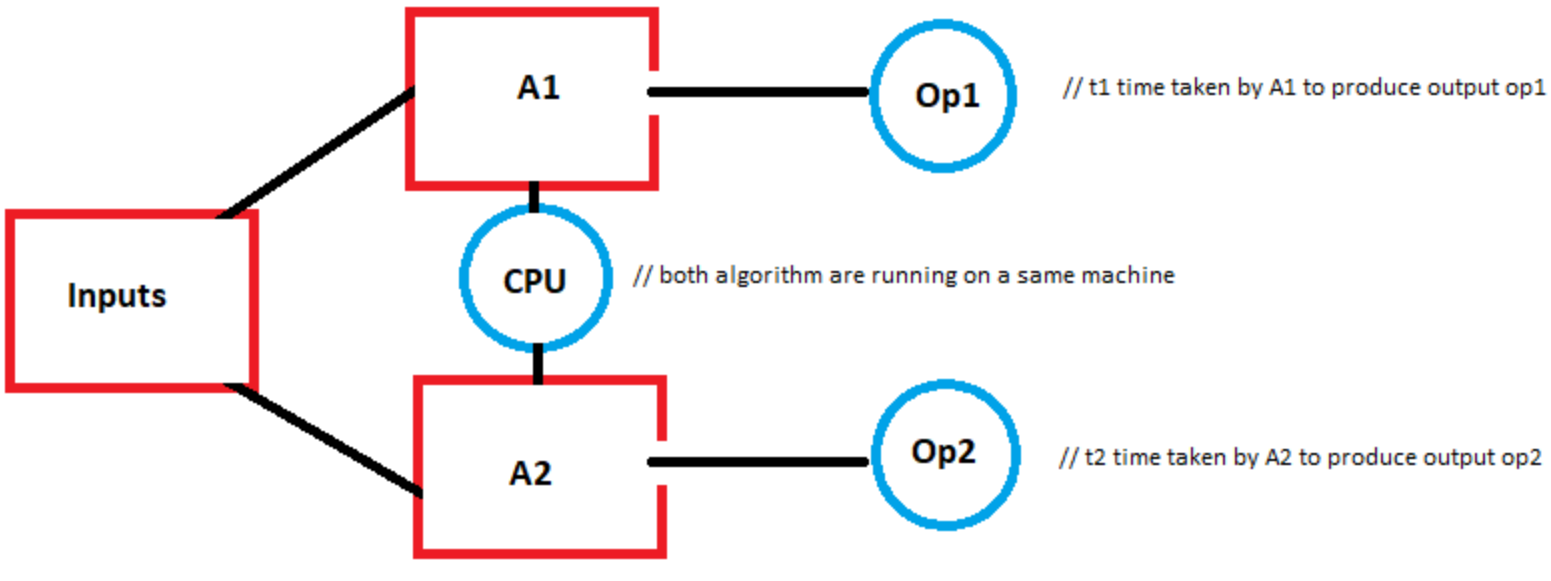
To go from city “A” to city “B”, there can be many ways of accomplishing this: by flight, by bus, by train and also by bicycle. Depending on the availability and convenience, we choose the one that suits us. Similarly, in computer science, multiple algorithms are available for solving the same problem (for example, a sorting problem has many algorithms, like insertion sort, selection sort, quick sort, merge sort and much more). Algorithm analysis help us to determine which algorithm is most efficient in terms of time and space consumed.

Goal of the Analysis of Algorithms

The goal of the analysis of algorithms is to compare the algorithms mainly in terms of running time but also in terms of other factors like- memory consumption, code implementation, programming language, developer effort etc.

How do we choose which algorithm is best?

Suppose there are two different algorithms A1 and A2 that are running on a same CPU with same input.



Now the question is which algorithm is faster among both of them and what inputs should we take (whether, the inputs are big or small inputs) and what is the implementation of an algorithm (whether, it’s efficient or inefficient implemetation) and in which programming language(like- C, C++ or Python etc.) it’s implemented and so on? Are some common questions that we encounter during comparison of algorithms.

As we can see that comparing two algorithms and deciding which algorithm is best is not an easy task. To answer these questions we have to first find out running time of an algorithm.

Running Time

Suppose you developed a program that finds the shortest distance between two major cities of your country. You showed the program to your friend and he/she asked you “What is the running time of your program”. You answered promptly and proudly “Only 3 seconds”. It says more practical to say the running time in seconds or minutes but is it sufficient to say the running time in time units like *seconds and minutes*? Did this statement fully answer the question? The answer is NO. Measuring running time like this raises so many other questions like-

- What’s the speed of the processor of the machine the program is running on?
- What is the size of RAM?

- What is the programming language? and much more.

In order to fully answer your friend's question, you should say like "My program runs in 3 seconds on Intel Core i7 8-cores 4.7GHz processor with 16 GB memory and is written in C++ 14". Who would answer this way? Of course no one. Running time expressed in time units has so many dependencies like a computer being used, programming language, a skill of the programmer and so on. Therefore, expressing running time in seconds or minutes make so little sense in programminng.

You are now convinced that "seconds" is not a good choice to measure the running time. Now the question is how to calculate the running time of a algorithm. To answer this question we first have to find out the factors that algorithms running depends upon.

Factors that algorithm running time depends upon

There could be many factors where running time (Time Complexity) of an algorithm depends upon and some of them are mentioned below.

- Input
- Single Processor vs Multi Processor **X**
- Computer Architecture like- 32 bit or 64 bit **X**
- Hardware **X**
- Programming languages **X**
- Read/Write speed to memory or disk **X**

When we talk about running time of an algorithm, we consider Input as a primary factor among all the factors because we only bother about how our algorithm behaves for various inputs or how the time taken by an algorithm grows with the input to the algorithm. So In conclusion we are mostly interested in the growth rate of time of a algorithm with respect to inputs.

Now the question is how do we find out the growth rate of time of a algorithm with respect to inputs?

To find out the growth rate of time of a algorithm, let's create a model machine and see how the growth rate of time of a algorithm behaves in our model machine.

Before creating a model machine let me introduce, how can we represents running time of an algorithm.

If the input size is n (which is always postive), then the running time is some function f of n i.e. Running Time = f(n). The functional value of f(n) gives the number of operations required to process the input with size n. So the running time would be the number of operations (instructions) required to carry out the given task.

Note

- We know that inputs are the major factor but in a algorithm there could be various operations that an algorithm executes and time taken by these operations may vary on a different machines. So when we calculate the running time of a algorithm we consider that each operations takes 1 unit of time to get execute for any machine(Computer).
- addition(a+b), subtraction(a-b), assigining a value(a=10), comparison(a ≥ 10), checking the condition with if-statement is one operation, In for-loop there are three operations present (intialization, condition checking, increment/decrement) all these operation are indiviual operations which takes 1 unit of time.
- The running time is also called a time complexity.

Create a Model Machine

Let's understand more clearly how to calculate the running time with respect to inputs of a algorithm by creating a our very own model machine.

Suppose the above image is our Model Machine consist of single processor, 32-bit architecture, sequential execution, takes 1-unit of time for each operation like- arithmetic and logical operations, assignment and return value.

Example- 1

Suppose you created a function “sum” which will calculate the sum of two numbers.

```
// calculate the running time sum function

1. sum(a, b)
2. {
3.  return a + b; // operation count = 2
4. }

a+b is one operation and return statement is one operation i.e total 2 operations are executed for this function.
Therefore, running time of this function will be:

f(a, b)) = 1 + 1 = 2 // constant time
```

Example- 2

Suppose you create a function “array_sum” which will calculate the sum of elements of an array.

```
// Calculate the running time of a array_sum function

1.  array_sum (int arr[], int n) {
2.  int sum = 0;
3.  for i = 0 to n-1
4.    sum = sum + arr[i]
5.  print(sum)
6.  }

operation          operation count      no. of times operations executed

int sum = 0              1                              1
for i = 0 to n-1         3                              n+1 // +1 for condition get failed
sum = sum + arr[i]       2                              n
print(sum)               1                              1

f(n) = 1*1 + 3*(n+1) + 2*n + 1*1
f(n) = 1 + 3n + 3 + 2n + 1
f(n) = 5n + 5
f(n) = n // linear time

// Note-
// 1. we ignore the lower order terms and interested in higher order terms while finding the running time of a algorithm.
// 2. you can write f(n) as T(n). Most programmers prefer writting T(n) over f(n).
```

Example- 3

Suppose you create a function “matirx_sum” which will calculate the sum of elements of a matrix.

```
1. matrix_sum(int matrix[][], int rows, int cols) {
2.  int sum = 0;
3.  for i = 0 to rows - 1 {
4.    for j = 0 to cols - 1
5.    sum = sum + matrix[i][j];
6.  }
7.  print(sum);
8. }

f(rows, cols) = 1*1 + 3*(rows+1)*(cols+1) + 2*rows*cols + 1*1
let rows, cols <=> n // for simplicity
f(n) = 1 + 3n^2 + 3 + 3n + 3 + 2n^2 + 1 // ignore the lower order terms
f(n) = n^2 // quadratic time
```

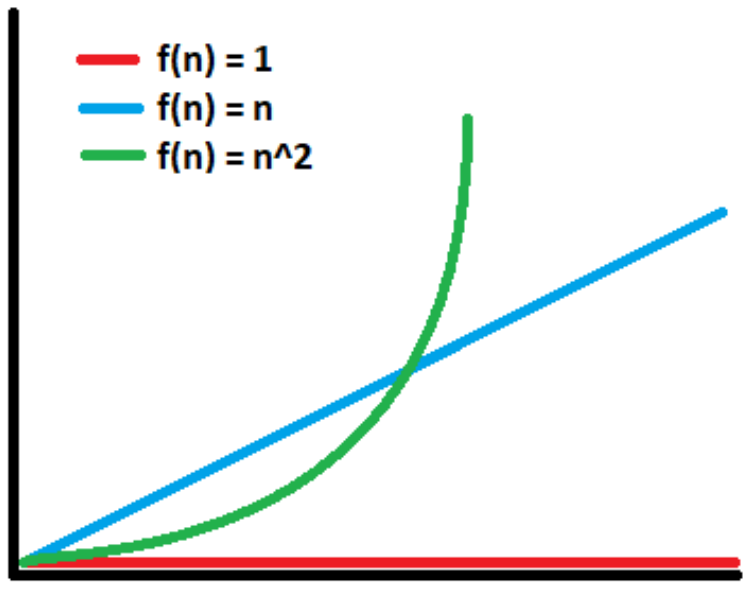
From the above examples we learn how to calculate the running time of a program. In next, section we are going to learn about “Growth of functions” which will help us to find out which running time is best among these three running times.

Growth of Functions

In the previous section, I said that the running time are expressed in terms of the input size (n). we have three running times (1, n, n^2). Among these three running times, which one is better? In other words, which function grows slowly with the input size as compared to others? To find this out, we need to analyze the growth of the functions i.e we want to find out, if the input increase, how quickly the running times goes up.

Plot a graph

One easiest way of comparing different running times is to plot them and see the natures of the graph. The following figure shows the graphs of 1, n and n^2. (x-axis represents the size of the input and y-axis represents the number of operation get executed i.e. running time)



Looking at the figure above, we can clearly see the function f(n) = n^2 is growing faster than functions f(n) = 1 and f(n) = n. Therefore, running time 1(constant time) is better than running times n, n^2 . One thing to note here is the input size is very small. I deliberately use the small input size only to illustrate the concept. In computer science especially in the analysis of algorithms, we do the analysis for **very large** input size.

Take a limit

Another way of checking if a function f(n) grows faster or slower than another function g(n) is to divide f(n) by g(n) and take the limit $n \to \infty$ as follows.

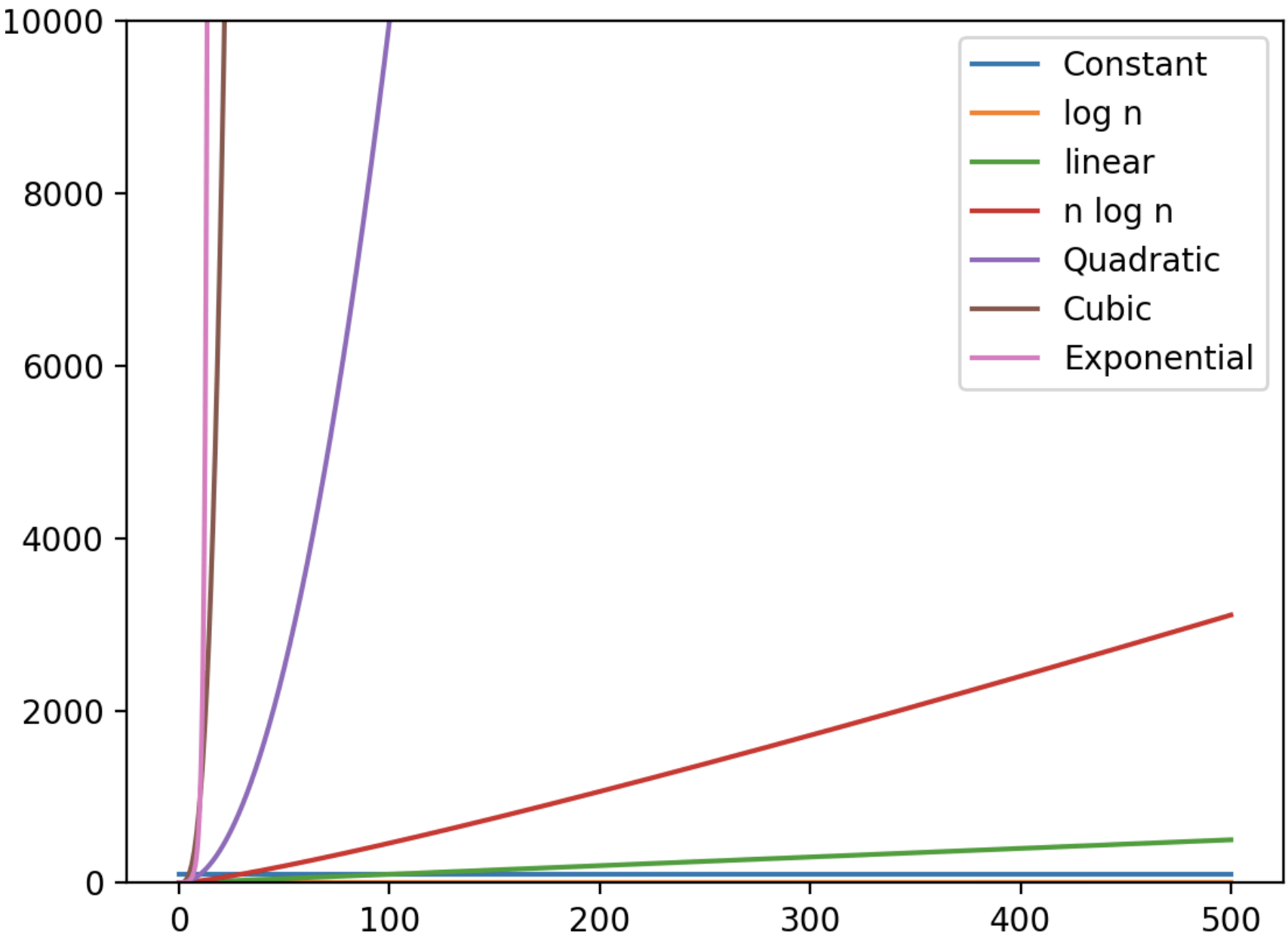
$$\lim_{n \to \infty} \frac{f(n)}{g(n)}$$

If the limit is 0, f(n) grows faster than g(n). If the limit is ∞, f(n) grows slower than g(n)

The table below shows common running times in algorithm analysis. The arrow in the table is representing slower to quicker (best to worst) running times.

Running Time	Examples
Constant	1, 2, 100, 300, …
Logarithmic	$\log n$, $5 \log n$, …
Linear	n, $n + 3$, $2n + 3$, …
$n \log n$	$n \log n$, $2n \log n + n$, …
Polynomial	Quadratic, Cubic, or higher order
Exponential	2^n, 3^n, $2^n + n^4$, …
Factorial	n!, n! + n, …

The figure below shows the graphical representations of these functions (running times).



Types of analysis

To analyze the given algorithm, we need to know with which inputs the algorithm takes less time (performing well) and with which inputs the algorithm takes a long time. We have already seen that an algorithm can be represented in the form of an expression. That means we represent the algorithm with multiple expressions: one for the case where it takes less time and another for the case where it takes more time.

Worst Case

- Input for which the algorithm takes a long time (Slowest time to complete).
- Example- Array is sorted in reverse order.

Best Case

- Input for which the algorithm takes a least time (Fastest time to complete)
- Example- Array is sorted.

Average Case

- Run the algorithm many times, using many different inputs that come from some distribution that generates these inputs, compute the total running time (by adding the individual times), and divide by the number of trials. Assumes that the input is random.
- Example- Elements of an array are in random order.

In general the Worst Case is called as Best Case and Best Case as a Worst Case of a algorithm. Because when you increase the input size you will going see the actual problem of a algorithm like- How much time a algorithm is actually taking to execute a operation for higher inputs?

When we do algorithm analysis we are actually concern about worst case and average case. But sometimes calculating the running time of a algorithm in average case is a complex work for many algorithms.

Asymptotic Notations

Asymptotic notations are the mathematical notations used to describe the running time of an algorithm when the input tends towards a particular value or a limiting value.

For example: In Insertion sort, when the input array is already sorted, the time taken by the algorithm is linear i.e. the best case.

But, when the input array is in reverse condition, the algorithm takes the maximum time (quadratic) to sort the elements i.e. the worst case.

When the input array is neither sorted nor in reverse order, then it takes average time. These durations are denoted using asymptotic notations.

Types of asymptotic notations:

- **Big-O notation**
- **Omega notation**
- **Theta notation**
- **Little-o**
- **Small-omega**

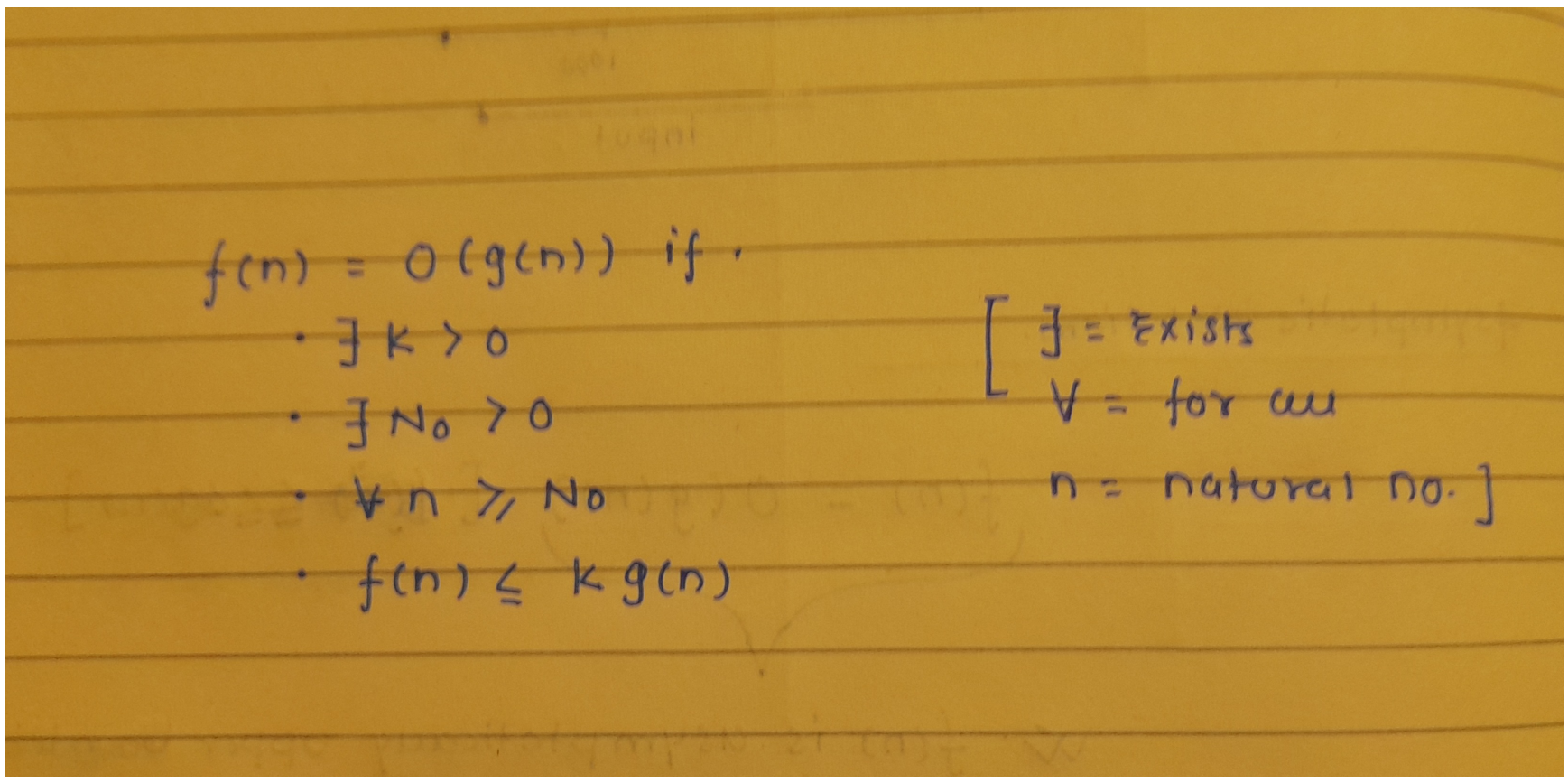
Goal of asymptotic notations

We know that running time are expressed as n, n^2, nlogn etc. These are called exact running time or exact complexity of an algorithm. We are rarely interested in the exact complexity of the algorithm rather we want to find the approximation in terms of upper, lower, and tight bound.

Big-O notation (≤)

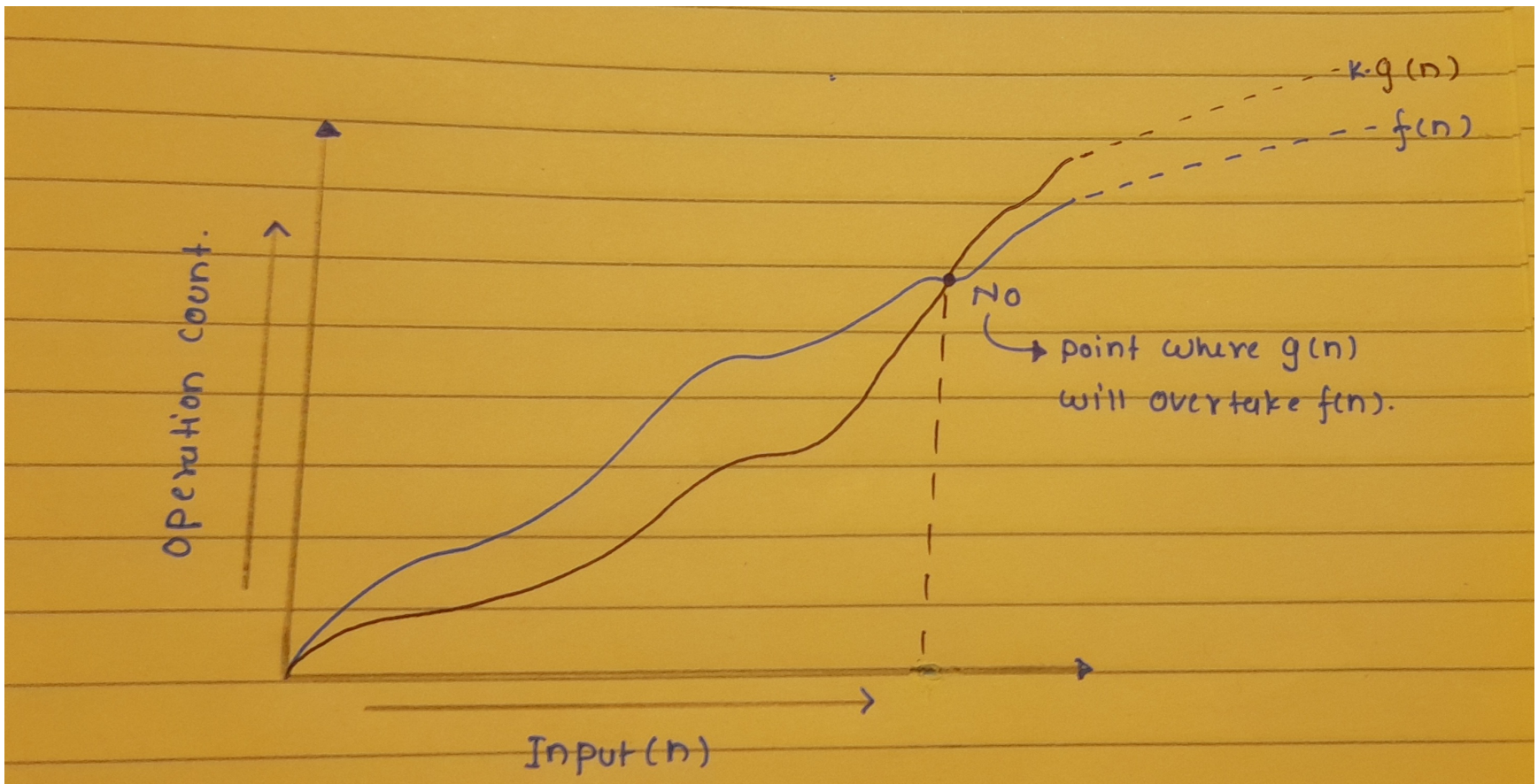
Big-O notation represents the upper bound of the running time of an algorithm. Thus, it gives the worst-case complexity of an algorithm.

Generally, it is represented as **f(n) = O(g(n))**



Note- f(n) and g(n) are two different functions.

Let’s plot a graph that will show you how **f(n) = O(g(n)).**



The graph shows that Nnot is the point from where g(n) overtakes f(n) and f(n) stays overtaken. Multiplying K with g(n) helps g(n) to overtake f(n). Note that K could be any number but it must be greater than 0, As a result we can see that f(n) is bounded by the g(n) i.e. the time complexity of f(n) will not going to cross the time complexity of g(n) no matter how bigger the input is.

So we can say that f(n) ≤ c(g(n)) i.e. that means f(n) is asymptotically upper bounded by g(n) or f(n) = O(g(n))

Note-

Here f(n) is the function whose time complexity we are finding. We assume g(n) as a nearest function with help of some constant (K) to bound the f(n). Remember if you take g(n) = n^100 for f(n) = n^2 doesn't make any sense it's like you are in a mobile shop and asking for a phone of price 5 cr. there should be a order in your price range, Same applies here you should always take g(n) as a nearest function of f(n) like- f(n) = n^2 then g(n) = 10n^2 (k= 10) or g(n) = kn^3 etc.

In Asymptotic Notation we ignore constant terms of a function. Considering constant terms of a function doesn't make any sense if the function is of higher order than other function because no matter how big constant you are using the higher order function will going to overtake the lower order function.

For Example-

Example- 1

```
f(n) = 3log n + 100
g(n) = log n

Is f(n) O(g(n))? Is 3 log n + 100 O(log n)? Let's look to the definition of Big-O.

3log n + 100 <= K * log n
Is there some pair of constants K, n0 that satisfies this for all n >= n0?

3log n + 100 <= 150 * log n  // n=2 i.e. is the point from where g(n) overtakes f(n)

Therefore, f(n) is O(g(n)).
```

Example- 2

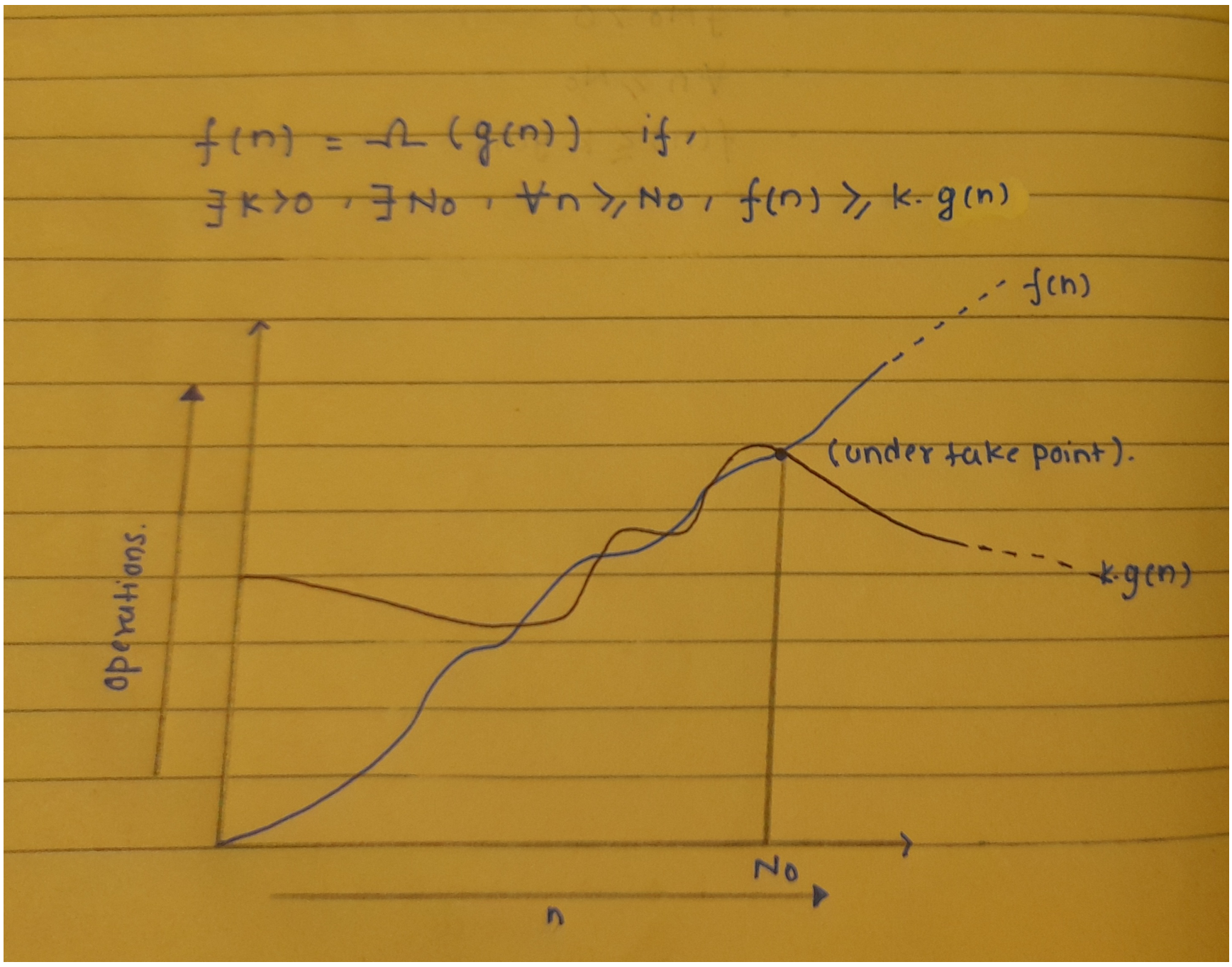
```
f(n) = 3*n^2
g(n) = n

Is f(n) O(g(n))? Is 3 * n^2 O(n)? Let's look at the definition of Big-O.
Is there some pair of constants c, n0 that satisfies this for all n > 0? No, there isn't. f(n) is NOT O(g(n)).
```

Big-Ω notation (≥)

Big-Ω notation represents the lower bound of the running time of an algorithm. Thus, it provides the best case complexity of an algorithm.

Generally, it is represented as **f(n) = Ω(g(n)).**

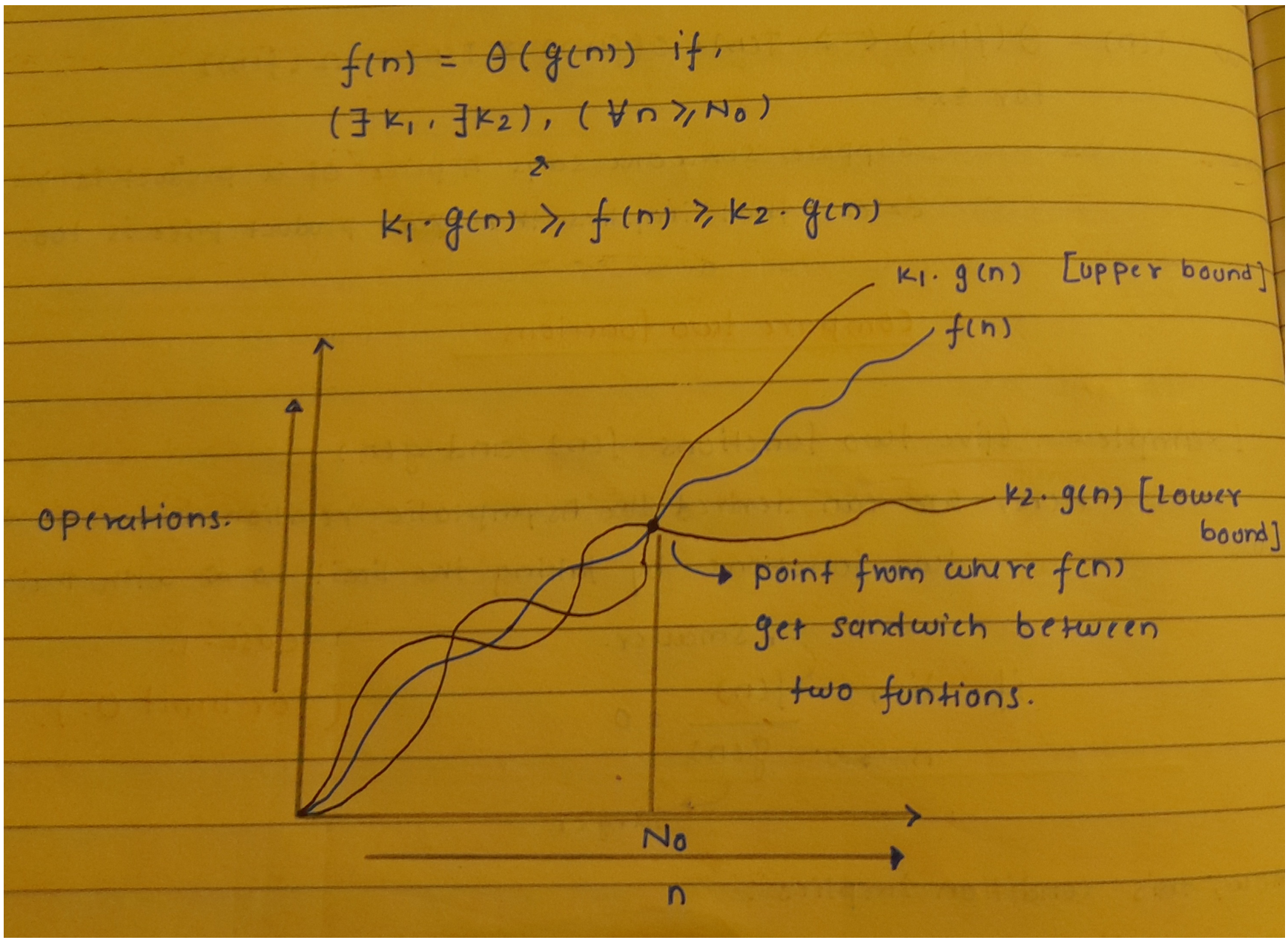


We can say that f(n) = O(g(n)) <=> g(n) = Ω(f(n))

Θ-notation (=)

Theta notation encloses the function from above and below. Since it represents the upper and the lower bound of the running time of an algorithm, it is used for analyzing the average-case complexity of an algorithm.

Generally, it is represented as **f(n) = Θg(n)**



We can say that f(n) = Θg(n) <=> f(n) = O(g(n) AND f(n) = Ω(g(n))

Example

```
f(n) = 2n + logn
g(n) = 2n + 3

Q. Is f(n) = 0g(n)?

Let's solve:
let k1 = 10 and k2 = 1
Therefore, k1.g(n) = 10*(2n + 3), k2.g(n) = 1*(2n+3) and f(n) = 2n + logn

Now, consider the leading terms and ignore constants and lower order terms only. From,
f(n) = n // is the leading term
g(n) = n // is the leading term

So, we can see that f(n) and g(n) both have same leading terms. Therefore we can say that f(n) = g(n) or f(n) = 0g(n).
```

Little o (<)

Little-o notation is used to describe an upper-bound that cannot be tight.

Let f(n) and g(n) be functions. We say that f(n) is o(g(n)) if for **any real** constant c > 0, there exists an integer constant n0 ≥ 1 such that 0 ≤ f(n) < c*g(n).

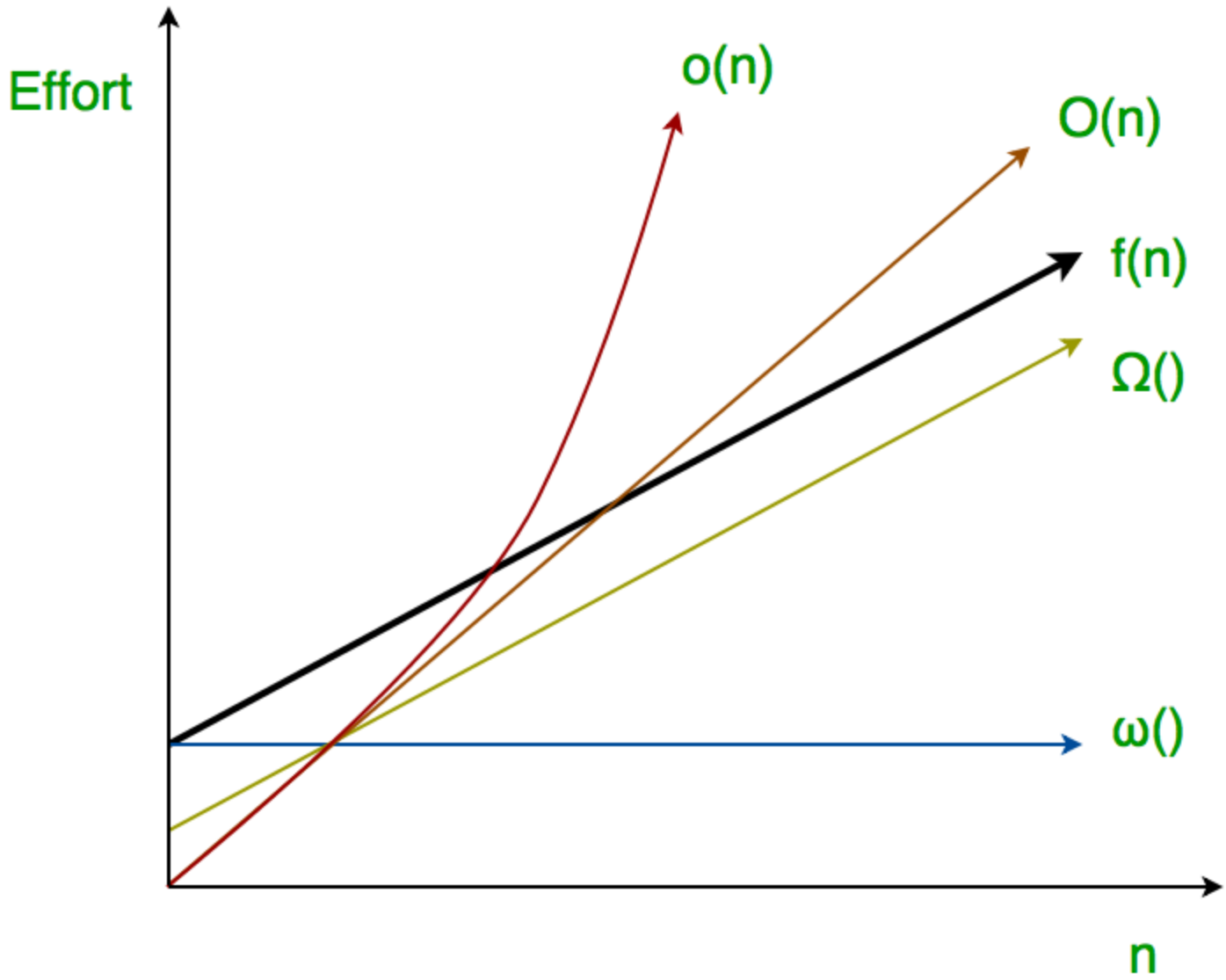
Little o is a rough estimate of the maximum order of growth whereas Big-O may be the actual order of growth.

Little Ω (>)

Little-Ω notation is used to describe an lower-bound that cannot be tight.

Let f(n) and g(n) be functions. We say that f(n) is Ω(g(n)) if for **any real** constant c > 0, there exists an integer constant n0 ≥ 1 such that 0<=c*g(n)<f(n).

Little Omega (ω) is a rough estimate of the order of the growth whereas Big Omega (Ω) may represent exact order of growth.


Effort
o(n)
O(n)
f(n)
Ω()
ω()
n