**Data Structure Algorithm(DSA)**

Algorithms are an integral part of a computer program, and in the real world, every software application uses an algorithm to perform operations. For example, the Ola app uses algorithms for operations such as the following:

1. Ola Select: On a customer's request, the app finds a cab nearby and follows the shortest route between the pickup and drop locations.
2. Ola Share: On multiple requests from different customers, the app follows the shortest route for different pickup and drop locations.

#### Q1. Algorithm

What is the function of the count[id[i]]++ instruction in this algorithm?

public void findDuplicates(int[] id) {

System.out.println("Duplicate data : ");

int count[] = new int[10000];

for (int i = 0; i < id.length; i++) {

count[id[i]]++;

if (count[id[i]] == 2)

System.out.print(id[i] + " ");

}

System.out.println();

}

Ans: The count[id[i]]++ instruction will increment the value at the index location id[i], i.e., the student ID of the count array for each iteration. If certain student ID occurs twice in the given data, then the value at the count[id[i]] is incremented to two. In this way, the count array keeps track of each student ID and helps in finding duplicate IDs.

#### Q2. Algorithm

The id [ ] array is as follows:

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| |  |  |  |  |  |  |  | | --- | --- | --- | --- | --- | --- | --- | | 3 | 5 | 2 | 4 | 6 | 5 | 2 |   choose the correct count [ ] array after executing the program. Note that the values in the array id [ ] act as the index values for the count [ ] in which the values get incremented as per the number of occurrences. |

Ans: count[id[i]] - [0, 0, 2, 1, 1, 2, 1]

**✓ Correct**

**Feedback:**

The count array is initialised with 0 and gets incremented in the cell corresponding to the student ID number. If the same student ID is repeated twice, then the corresponding cell of the count array is incremented to 2.

#### Q3: Algorithm

In your own words, explain ‘algorithm 2’ — how it helps in finding the duplicate data from a given database.

Ans: A count array is used to track the number of times you see a particular student ID. Specifically, the count array variable is defined by the maximum possible size of the input data, and with respect to the input data, the corresponding array index locations of the count variable are incremented by 1. If the student ID repeats, then the if condition, i.e. “count [id[i]] is equal to 2”, is satisfied and prints the duplicate student ID data. Unlike the previous algorithm, if a specific student ID occurs multiple times, i.e. count[id[i]] > 2, then algorithm 2 does not print that student ID.

Q4: List the major differences between the two algorithms discussed so far.

Ans: In algorithm 1, you use only one array variable for registered student IDs and two for loops to iterate across the data and find duplicate student IDs. However, in algorithm 2, an extra count array variable is declared to keep the count of the duplicate student IDs, and one for loop is used to iterate through the input data.

Q5: Give examples in which the same computational problem can be solved by two different algorithms.

Ans: One example can be the sum of the first n consecutive natural numbers calculated either using a single for loop or a simple arithmetic formula such as n(n+1)/2.

Q6: **In your own words, explain the function of the ‘break’ instruction in an algorithm.**

Ans: In a scenario wherein the same student registers multiple times for a course, the specific student ID occurs multiple times in the given data.In such a scenario, the ‘break’ instruction is executed for each iteration when the duplicate student ID is found and breaks out of the inner loop(j). In a way, the 'break' instruction helps to print the student ID equal to the number of times for which the student registers and avoids printing the duplicate student ID multiple times for each iteration.

# Parameters for Algorithm Efficiency

the parameters that determine the efficiency of an algorithm are the **time taken** (running time) and the **memory space** required to execute an algorithm.

The time taken and the memory space required to execute an algorithm are calculated as the functions of the input size. Since you typically do not know the input size beforehand, you simply use the variable ‘n’ to represent the potential input size of the algorithm.

Now, the question that arises is **how do you calculate the time taken to execute an algorithm?**

* Each instruction in an algorithm takes a specific amount of time to execute, and a certain instruction set is executed for more than once, i.e., the instruction set inside a 'for' or 'while' loop.
* To analyse an algorithm, you must calculate the number of times for which an instruction set is executed with respect to the input size (n) rather than the exact time values.
* This is because the time taken to execute an algorithm depends on various external factors such as processor speed, compiler, etc. because these external factors can vary from computer to computer.

**Note:** When calculating the number of times for which an instruction set is executed, software developers and computer scientists tend to consider the worst case, which refers to the specific case in which the instruction set is executed for a maximum number of times.

There are three cases in the analysis of an algorithm, which are as follows:

1. Best case
2. Average case
3. Worst Case

For Demo program in dsa folder:

**Best case:**

* The lower bound on the time taken by an algorithm is the best case. In the program given above, for the Input 1, the output was ‘Entered no found at position:1’, which means that the number was found at the first position itself.
* The ‘for’ loop in the program iterates for n times until it finds the required value in the array at the ith index.
* Since the required value is found at the first index, the ‘for’ loop terminates in the first iteration itself; hence, this is the best case for this program.
* The number of instructions that get executed in the best case is constant.

**Average case:**

* In this case, consider all possible different types of input data and calculate the time taken by the algorithm for all those inputs.
* Now, add all those calculated time taken values and divide it by the total number of inputs. Then, the obtained value is the average time.

**Worst case:**  
The upper bound of the time taken by an algorithm is the worst case.

* In the program given above, for the Input 2, the output was ‘Entered number found at position:10’ and that for Input 3 was ‘Sorry! Entered number not found’.
* For both inputs, the program searches throughout the array; here, it will take the maximum time for the algorithm to execute. Hence, this is the worst case.
* For most of the times, the worst case time complexity of an algorithm is used for the algorithm analysis because the worst case time complexity guarantees the upper bound of the time that the algorithm takes, i.e., the maximum time taken by the algorithm.

In the next segment, we will see how to apply this complexity function to analyse algorithms.

Before moving on, please answer the question below to understand the worst case in an algorithm.

#### Q7: Algorithm

According to you, what is the worst case of algorithm 1?

public void findDuplicates(int[]id){

System.out.println("Duplicate student id : ");

for(int i=0;i<id.length;i++){

for(int j=i+1;j<id.length;j++){

if(id[i]==id[j]){

System.out.print(id[i]+" ");

break;

}

}

}

}

Ans: In algorithm 1,if (id[i] == id[j]) {System.out.print(id[i] + " ");break;}The above inner loop(j) instruction set is executed maximum no of times(worst case) when there are no duplicate student IDs in the given data.If there are any duplicate IDs, then the inner loop prints the student ID and breaks out of the inner loop(j).

# Best, Average, Worst Case

**So what actually IS the difference between Best, Average & Worst Case Time Complexity?**

In simple words, the best case refers to the best case scenario wherein the resources or the time required to perform a given operation is the least. As the name suggests, the worst case refers to the MOST PESSIMISTIC scenario wherein the highest amount of resources or time may be required to perform a given task or operation. An average case time complexity, as the name suggests, refers to how an algorithm may perform on an average.

1. **Best Case:** An algorithm's behavior under optimal conditions. For example, given a simple list and we need to find an whether an element exist inside that list, the best case would occur when the element is found right in the first iteration itself, i.e. in the first shot!

2. **Worst Case:** Using the worst case scenario often gives you the safest guarantee within which the algorithm may work. For example, when designing an algorithm, the safest bet would be to assume that the worst happens and thereby worst case time analysis comes into picture here. For example, while searching for an element in a very long list of numbers, the worst case would be to assume that the element does not exist inside the list at all and that the procedure needs to check each and every element of that long list to finally conclude that the element does not exist!

3. **Average Case:** While the worst case scenario is mostly highly useful to determine the performance of an algorithm, it often gives us an overly pessimistic performance overview of an algorithm. A more realistic estimate of the performance of an algorithm would be to use the average case time complexity i.e. to calculate how an algorithm performs on an average.

# Complexity Functions

#### Q8: Algorithms

Can you explain how the sum of n - 1 consecutive natural numbers is equal to n(n - 1)/2?

Ans: Let us consider the following:Sn = 1 + 2 + 3 + 4 + 5 … (n - 1) → (eq 1)Sn = (n - 1) + (n - 2) + (n - 3) + (n - 4) ... + 1 → (eq 2)Now, on adding the corresponding terms of both eq 1 and eq 2, the equation obtained is as follows:2Sn = n + n + n + n … + nn - 1 terms have been added in the equation.So, 2Sn = n(n - 1)Sn = n(n - 1)/2Therefore, the sum of n - 1 consecutive natural numbers is n(n - 1)/2.

#### Q9: Algorithms

How many times does the for loop(i) instruction set execute in algorithm 2?

public void findDuplicates(int[] id) {

System.out.println("Duplicate data : ");

int count[] = new int[10000];

for (int i = 0; i < id.length; i++) {

count[id[i]]++;

if (count[id[i]] == 2)

System.out.print(id[i] + " ");

}

System.out.println();

}

Ans: The for loop(I) instruction set executes for n times, as the variable i increments from 0 to n -1(id.length). If the count[id[i]] is equal to two, then it prints the duplicate student ID.

Now, having understood the number of times for which a specific instruction set is executed in each algorithm, let’s follow an analytical approach to calculate the total time taken by both algorithms.

**Time Complexity of Algorithm 1**

|  |  |
| --- | --- |
| Function | Time taken |
| **public** void findDuplicates(int[] id) {  System.out.println("Duplicate student id : "); | * This instruction set is executed once. * Assume that this instruction set takes a constant time of c1 to execute. |
| **for** (int i = 0; i < id.length; i++) {  **for** (int j = i+1; j < id.length; j++){  **if** (id[i] == id[j]) {    System.out.print(id[i] + " ");  **break**;    }  }  } | * The outer loop iterates n (array size) times. * In the worst case, the inner loop instruction set is executed n - (i + 1) times for each outer loop iteration. * In total, it performs n(n-1)/2 steps. * Assume that it takes a constant time of c2 to check the **if**condition and print the duplicate data. |

Therefore, the total time taken to execute algorithm 1 for an input size n is as follows:

Total time taken to execute algorithm 1 = Time taken for the step 1 + Time taken for step 2

                                                              T(n)=c1+c2∗n(n−1)/2

Here, T(n) represents the time complexity; it is the total time taken to execute an algorithm as a function of the input size (n). The time complexity function gives you an idea of how long an algorithm takes to process the output.

**Time Complexity of Algorithm 2**

|  |  |
| --- | --- |
| Function | Time taken |
| **public** void findDuplicates(int[] id) {    System.out.println("Duplicate data : ");  int count[] = **new** int[10000];  } | * This instruction set is executed once. * Assume that the declaration of the count array variable takes a constant time of c1. |
| **for** (int i = 0; i < id.length; i++){  count[id[i]]++;  **if** (count[id[i]] == 2){  System.out.print(id[i] + " ");  }  System.out.println();  } | * The for loop iterates n times. * Assume that the instruction set in the for loop takes a constant time of c2. |

Therefore, the total time taken to execute algorithm 2 for an input size n is as follows:

Total time taken to execute algorithm 2 = Time taken for the step 1 + Time taken for step 2

                                                          T(n)=c1+n∗c2

**Space complexity of algorithm 1**

The space complexity is the additional memory space needed to execute an algorithm as a function of the input size (n). In general, you only calculate the extra memory required, without including the memory needed to store the input. The space complexity is represented by S(n).

In algorithm 1, except for the input array, the remaining variables are independent of the input size and consume a fixed amount of memory, irrespective of the input size. So, algorithm 1 requires a constant memory space of m units. Therefore, the space complexity of algorithm 1 is a constant function, i.e., S(n)=m.

**Space complexity of algorithm 2**

With respect to the space complexity, assume that there are only 10,000 students in the university, and their IDs are in the range of 1-10,000. The count array variable size needs to be the highest student ID, i.e., 10,000, to keep a count of all the student IDs registered. Therefore, the space complexity of algorithm 2 increases linearly as the number of students increases in the university or is equal to the highest student ID.

The space complexity is linearly proportional to the number of students in the university, as the length of count array is equal to the highest student ID.

Note: In this case, irrespective of the input size, i.e., the number of students registered, the required memory space depends on the highest student ID.

Now, having obtained a functional form for both parameters, which are the time taken and the memory space required for both algorithms, you need a mathematical tool to compare the obtained complexity functions about which we will learn in the next segment.

#### Q10: Algorithm

What is the total number of times for which the inner loop(j) instruction set is executed in the given algorithm?

Note: The 'Break' instruction is removed, and the remaining code remains the same.

public void findDuplicates(int[] id) {

System.out.println("Duplicate student id : ");

for (int i = 0; i < id.length; i++) {

for (int j = i +1; j < id.length; j++) {

if (id[i] == id[j])

System.out.print(id[i] + " ");

}

}

}

Ans: n(n−1)/2

**✓ Correct**

**Feedback:**

The break instruction is executed when there is duplicate data. On removing the instruction, the inner loop iterates through each data cell until the value of j equals the length of the array, irrespective of the input data. So, in total, this algorithm takes n(n-1)/2steps. This is the worst case of algorithm 1, when there are no duplicate student IDs in the given data.

#### Q11: Space Complexity

Between algorithm 1 (iterating using two for loops) and algorithm 2 (using count [ ] array), which algorithm needs more memory space to find the duplicate student IDs?

Ans: Algorithm 2

**✓ Correct**

**Feedback:**

Algorithm 2 needs more memory space than algorithm 1. This is because algorithm 2 declares a count array of size 10,000 besides the input array id[]. Also, the count array size increases as the number of students in the university increases. The count array size depends on the highest possible student ID, whereas algorithm 1 requires a constant memory space, irrespective of the number of students in the university.

# Asymptotic Notations - I

#### Q12: Worst Case

What is the possible worst case while searching for a specific element in an array if an algorithm looks for the element sequentially from the start of the array?

Ans: There are two possible worst cases:1. The specific element is not present in a given array; in such a case, the program checks the entire array. The algorithm executes the maximum number of steps in the possible worst case.  So if the array size is n, the worst case will be n.2. The specific element is the last element in a given array; in such a case, the program will check the entire array before it finds the element that it is looking for. The algorithm executes the maximum number of steps in the possible worst case. So if the array size is n, the worst case will be n.

#### Q13: Big O

Assume that an algorithm takes T(n)=2n2+4 time, where n is the input size. What is the Big O of T(n)?

Ans: O(n2)

**✓ Correct**

**Feedback:**

On dropping the constant multipliers and less significant terms, you can observe that the order of the function is 2; hence, it belongs to O(n2).

**Note:** In the video given above, at 4:58, the professor intends to say “The actual answer deviates from 10n3 by only 0.01%” not “n3 by only 0.01%”.

T(n) = 2n2−3n+6   
T(n) ≤2n2 for n > 2   has been shown in the video.

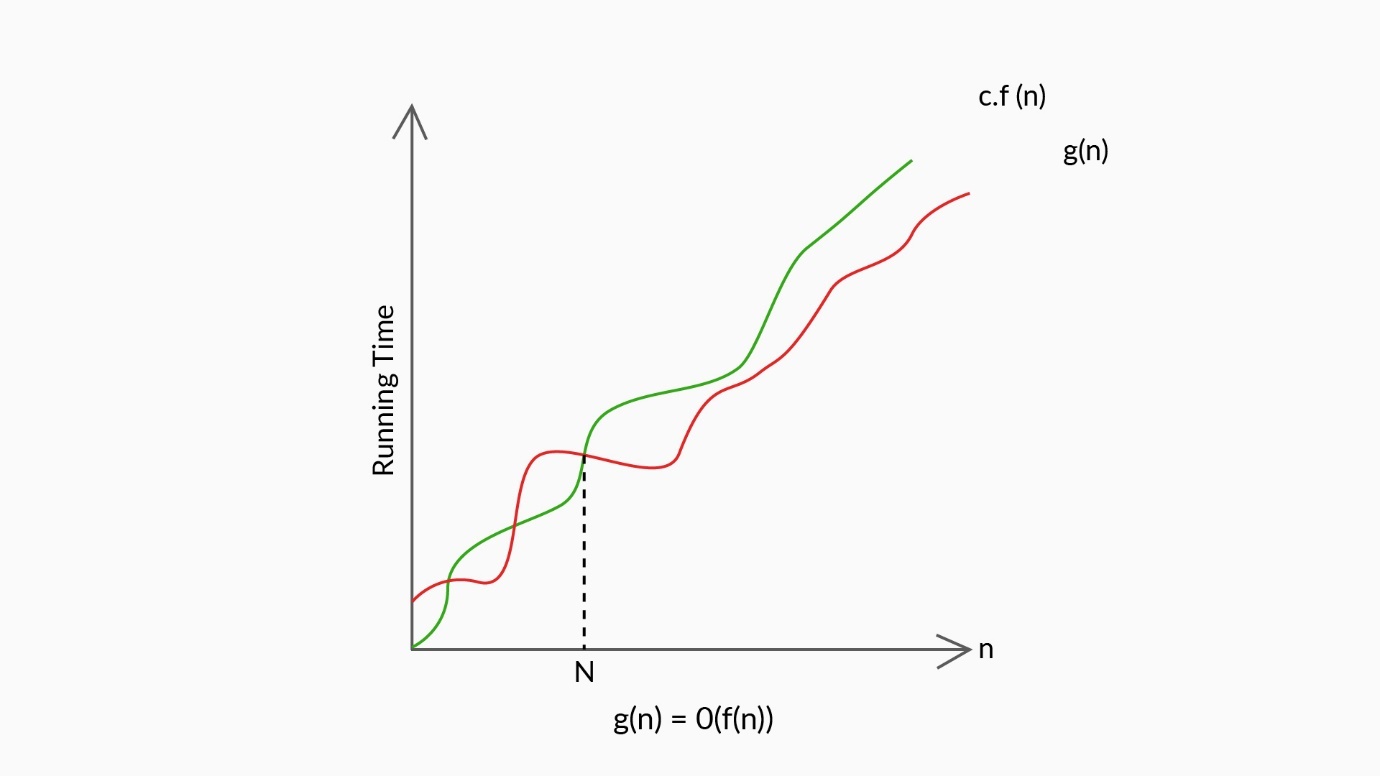
**Big-O notation** is a relative representation of the complexity of an algorithm. Here there are some important terms being used, namely "relative", "representation" and "complexity".

1. By relative, we try to say that ideally, it makes sense to compare only those two or more algorithms that perform a similar kind of function. For example, if we have an Algorithm X which sorts a list of integers, Algorithm Y which again sorts a list of integers and Algorithm Z which performs arithmetic multiplication, it makes sense to compare only X with Y since they are both performing similar kind of operations.
2. By representation, we try to say that Big-O reduces the comparison between algorithms to a single variable. This variable is often chosen based on observations. For example, when comparing sorting algorithms, we use the criteria of "number of comparisons" to assess which algorithm performs better. It may so happen that comparison may be cheap but swapping may be expensive? In that case, our criteria or variable changes.
3. By complexity, we mean to say something like: if it takes 1 second to sort 1000 integers, how many seconds will it take to sort 1 million integers? By complexity, here we mean a relative measure to something else.

Now, before reading about the graphical representation of asymptotic notations, Vishwa will explain what these graphs represent and how they relate to the run-time analysis that we have been discussing so far.

**Big**O

Big O refers to the order of growth of a function; essentially, the growth of any complexity function for large input size(n) values depends on the most significant term of the function. So, less significant terms are dropped from the function to calculate the Big O.



Big O

So when calculating Big O, you can drop the lower valued terms and only keep the highest terms.

For example:

The time complexity of a function is defined as 2n2−3n+6

As the value of n increases, only the term  2n2 and 3n will contribute to the increase in the time since 6 is constant.

Now, it is clearly understood that upon further increasing the value of n, the change brought by 3n will be insignificant in 2n2

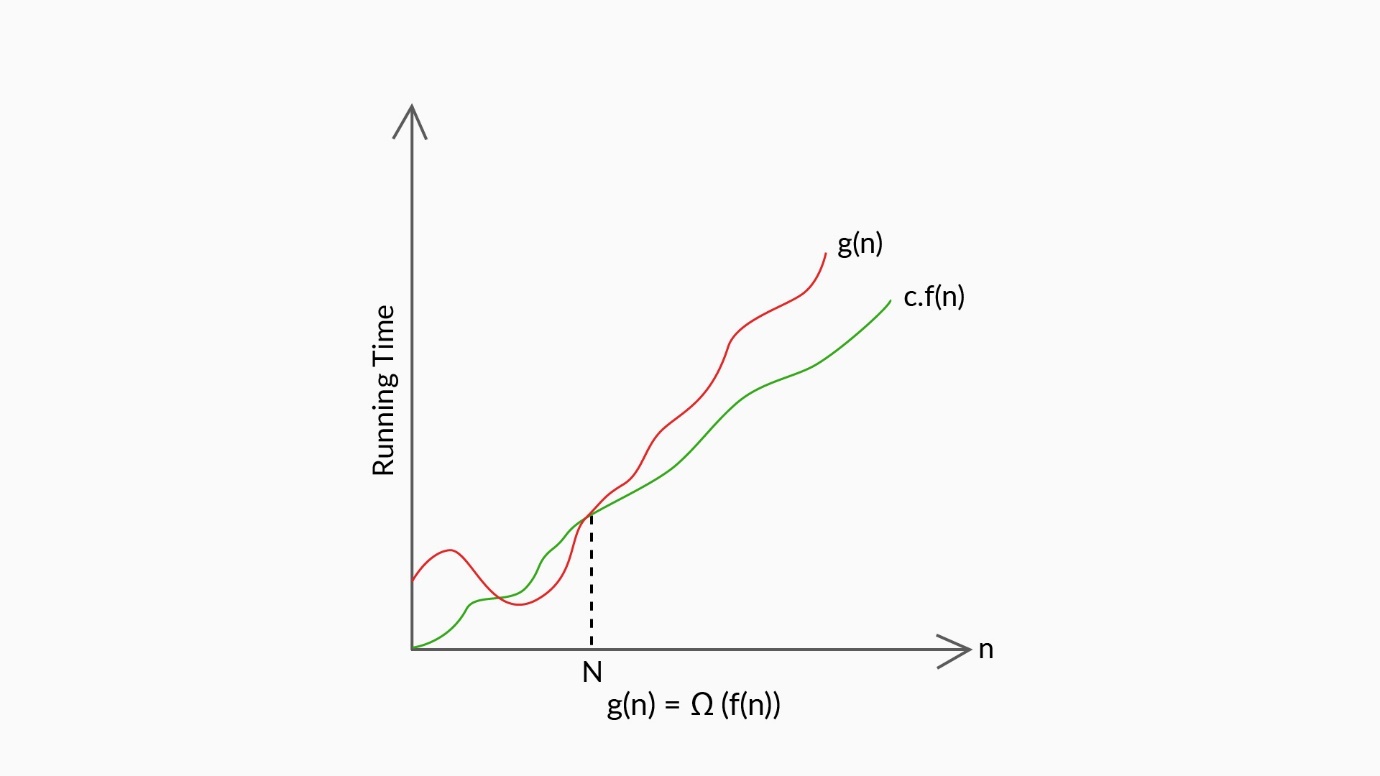
Therefore you can drop the lower value term 3n and only consider 2n2. For further refining this complexity, you can drop off 2, since it is a constant value.

Finally, the time complexity would be of the order O(n2)

**Big Omega**

Big Omega indicates the lower bound of the running time of an algorithm. In the earlier scenario of opening a lock, if you were able to undo the lock on your first attempt itself, then the first attempt will be the lower bound. In any other case, the number of attempts to unlock will be **greater than** the lower bound.

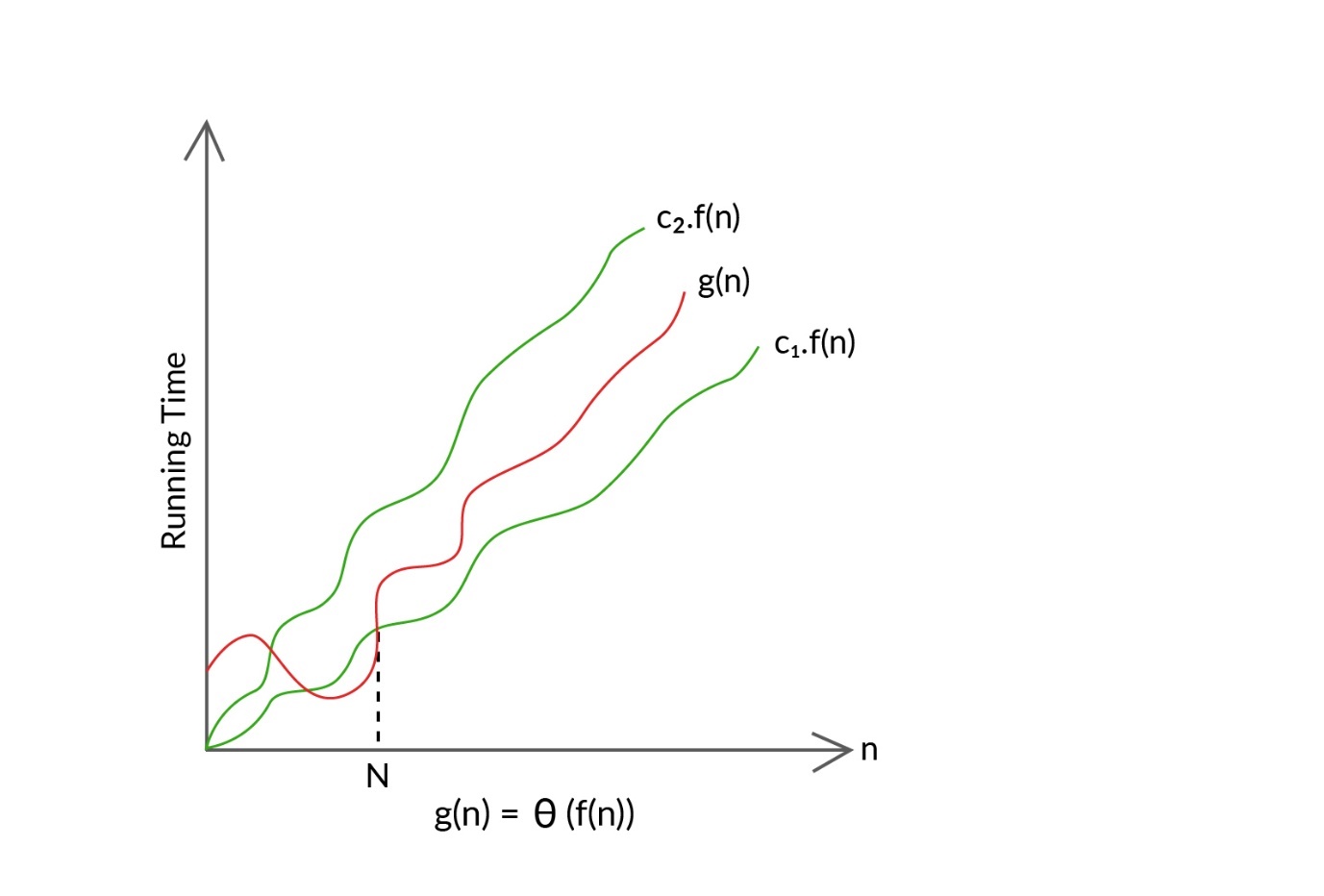
When calculating Big Omega, the time complexity function of an algorithm is **greater than or equal** to the lower bound.



Omega

**Big Theta**

Big Theta represents an in-between case, bounded both above (upper bound) and below (lower bound) the running time of an algorithm.



Theta

From the three asymptotic notations that we discussed, the most used notation to compare algorithms is Big O. So, we tend to find the worst case of an algorithm with respect to the input size (n). In the next segment, we will learn more about it.

#### Q14: Big O

Reduce the time complexity function of the following algorithm using the Big O notation:

T(n)=5n3+3n2+4

Ans: O(n3)

**✓ Correct**

**Feedback:**

On dropping the constant multipliers and less significant terms, you can observe that the order of the function is 3; hence, it belongs to O(n3).

#### Q15: Time Complexity

Simplify the following functions and choose the correct option.

i) T(n)=2n2+3

ii)T(n)=3n3+5n2+2n+7

iii)T(n)=6n+1

Ans: O(n2),O(n3),O(n)

**✓ Correct**

**Feedback:**

On dropping the constant multipliers and less significant terms, the order of the first function is quadratic, that of the second function is cubic and that of the third function is linear. So, O(n2),O(n3),andO(n).

# Asymptotic Notations - II

A very simplistic understanding of the complexity of algorithms can be understood from this distribution:

**O(n): known as Linear complexity**

1 item: 1 operation  
10 items: 10 operations  
100 items: 100 operations

**O(n^2): known as Quadratic complexity**

1 item: 1 operation  
10 items: 100 operations  
100 items: 10,000 operations

**O(1): known as Constant complexity**

1 item: 1 operation  
10 items: 1 operation  
100 items: 1 operation

**O(log n): known as Logarithmic complexity**

1 item: 1 operation  
10 items: 2 operations  
100 items: 3 operations  
1000 items: 4 operations  
10,000 items: 5 operations

... and so on.

# Common Rules for Calculating Big O - I

 when analysing an algorithm:

**Rule of sums:** If two loops come one after another, you need to add the number of execution steps for each loop.  
**Rule of products:** If two loops are nested, then you need to multiply the number of execution steps for each value.

The analysis of different types of iterative programs can be conducted as follows:

1. The time complexity of a single instruction is O(1).

**Example:**int a = b + c;

Why is that so? That is because no matter what is the value of b and c, int a will simply store the value of addition of b and c. Since the input itself is fixed and the operation of addition takes constant time, the overall time complexity will be O(1).  
  
Now you know that adding two numbers take constant time i.e. O(1), how much time do you think will be taken for adding n numbers? Well yes, you guessed it right! It will be O(n) because the larger the n will become, the more time it will take.

2. The time complexity of a function is O(1) if it contains only simple instructions without any loops or recursion and does not call to any other function (whose time complexity is not constant).

**Example:**

// returns the minimum of two numbers

public static int minimum\_of(int number1, int number2) {

int minimum;

if (number1 > number2)

minimum = number2;

else

minimum = number1;

return minimum;

}

Let us take a look one by one at each step:  
Step 1: **int minimum;**  
Step 2: **if (number1 > number2)**  
Step 3: **minimum = number2;**  
Step 4: **else minimum = number1;**  
Step 5: **return minimum;**

Again,  
1. Step 1 is simply a declaration so it happens in constant O(1) time [no loops, no repetition, constant/fixed input]  
2. Step 2 is simply a comparison operation and since comparison itself takes constant time and the input (number1 and number2) is fixed, so this step also takes O(1) time. [no loops, no repetition, constant/fixed input]  
3. Step 3 is again simply an assignment of values operation, so it takes constant time [no loops, no repetition, constant/fixed input]  
4. Step 4 is also assigning of values which takes O(1) time [no loops, no repetition, constant/fixed input]  
5. Step 5 is simply returning a value and so is O(1) [no loops, no repetition, constant/fixed input]

3. If there is a loop that iterates some constant number of times, then the time complexity of that loop is O(1).

**Example:**

//c here is some constant

for (int i = 0; i < c; i++) {

//Some instructions with time complexity O(1)

}

C is a constant value. So basically, you are performing constant number of operations irrespective of the value of n. There is no varying input n. The loop run is fixed - c times. So it has O(1) time complexity.

4. The loop given below has the time complexity O(n) because the loop variable is incremented by constant amounts from 1 to n or decremented by constant amounts from n to 1, which means that the innermost instructions get executed n times.

**Example:**

//incremented

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

//Some instructions with time complexity O(1)

}

//decremented

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

//Some instructions with time complexity O(1)

}



In the first loop, i starts from 1, keeps increasing by 1 in each step till it attains the value n. Therefore, what is the total no. of times the loop is run? You are right! It is n times and thus the time complexity of the first loop is O(n).

In the second loop, i starts from n, decreases by 1 in each iteration till it reduces back to the value of 1. Therefore, what is the total no. of times the loop is run? You are right! It is n times and thus the time complexity of the second loop is also O(n).

Since both these loops are run in conjunction with each other, their total time complexity will be Time(loop 1) + Time (loop 2) i.e. O(n) + O(n) which is O(n) itself.

# Common Rules for Calculating Big O - II

you learnt to calculate the time complexities for the following code snippets.

Example 1:

**int** a[] = **new** **int**[n];

**for**(**int** i = **1**; i <= n; i += **1**)

{

**for**( **int** j = **1**; j <= n; j += **1**)

{

a[i] = i + j;

}

}

The statement a[i]=i+j runs in constant time. Both the inner and the outer loops run ‘n’ times. However, for every outer loop iteration, the linear loop runs ‘n’ times. Therefore, the total time complexity for the program is O(n2).

Example 2:

**int** a[] = **new** **int**[n];

**for**(**int** i = n; i >= **0**; i -= **1**)

{

**for**( **int** j = i + **1**; j <= n; j += **1**)

{

a[i] = i \* j;

}

}

In this code snippet, you calculated the time complexity by actually substituting the value for n and calculating the number of iterations. You discovered that for ‘n = 3’, the program has six iterations. This pattern when expressed in general terms led you to arrive at the following formula to calculate the number of iterations for a given ‘n’:

(n2+n)/2

While calculating time complexities, you know that the term with the highest power is considered and the rest of the terms are dropped. Hence, the time complexity is O(n2).

Example 3:

**for**(**int** i = **1**; i < n; i \*= **2**)

{

System.out.println("Value of i is" + i);

}

From the video, you also learnt that whenever an iterator is multiplied or divided by a constant, the time complexity of the program is O(logn).

On substituting for n = 9, you discovered that for i = 16, the loop ends. The total number of iterations that happened for ‘i’ to reach 16 is 4.

Example 4:

**int** i = n;

**while** (i > **0**)

{

**for**(**int** j = **0**; j < n; j++)

{

System.out.println("test");

}

i = i / **2**

}

You also learnt that on every iteration of the outer loop, ‘i’ is divided by 2. This is a situation where an iterator is divided by a constant. The inner loop runs ‘n’ times for each iteration of the outer loop. Therefore, the time complexity is O(nlogn).

Example 5:

**while** (n > **0**)

{

**for**(**int** j = **0**; j < n; j++)

{

System.out.println(\*);

}

n = n / **2**

}

This code snippet looks similar to Example 4, however, it results in very different time complexities. The value of n (on each iteration of the outer loop) is divided by 2. This results in the inner loop having a different number of iterations for every iteration of the outer loop.

Vignesh generalised the number of iterations for this program to  2n−1; however, as we remove the constants while expressing time complexities, the time complexity is O(n).

# Common Rules for Calculating Big O – III

1. The time complexity of the below loop is O ( n2).

**Example 1:**

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

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

//Some instructions with time complexity O(1)

}

}

In the instruction set given above, the inner loop gets executed n times, and the outer loop also gets executed n times; hence, the innermost instruction gets executed n2 times. Therefore, the time complexity of the program is O(n2).

**Example 2:**

for (int i = n; i > 0; i -= 1) {

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

//Some instructions with time complexity O(1)

}

}

Here also, the outer loop starts with the value of i as n, gets decreased by 1 in each step and finally terminates when it attains the value of 0. Thus, it runs a total no. of n times.

Similarly, in the inner loop, j starts from a value of the outer i in each step (which itself varies between 1 to n), increases in each step and goes on till it attains the value n. Thus the time complexity of the inner loop is also O(n).

Now since these two loops are placed one inside the other, for each iteration of i, the inner loop with j will execute n times, and thus, the total time complexity will be O(n\*n) i.e. O(n^2)

2. The time complexity of the below loops is O(logn).

**Example 1:**

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

//Some instructions with time complexity O(1)

}

    The value i for each iteration is as follows:

        1   2   4   8   …..    n     
     Let's assume that the loop given above gets executed k times. Then,

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| I values | 1 | 2 | 4 | 8 | ....... | n |
| No of iterations | 1 | 2 | 3 | 4 | ....... | k |
| I values | 20 | 21 | 22 | 23 | ....... | 2k |

     2k = n  
     k =log2n  
     The number of iterations = k = log2n. Therefore, the time complexity = log2n.

**Example 2:**

for (int i = n; i > 0; i /= 2) {

//Some instructions with time complexity O(1)

}

     Similarly as above

3. The time complexity of the below loop is O(loglogn).

**Example 1:**

// pow(i, c) is a function which calculates ic, c>0

for (int i = 2; i <= n; i = pow(i, c)) {

//Some instructions with time complexity O(1)

}

    Let's assume that n=22k.

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| k values | 1 | 2 | 3 | ........ | k |
| i values | 2, 4 | 2,4,16 | 2,4,16,256 | ........ | 2,...22k |
| No of iteration | 2 | 3 | 4 | ........ | k+1 |

    22k = n  
    k =log2log2n  
    Number of iterations = k+1 = k (by excluding constant)  
    Because the number of iterations = k =log2log2n, the time complexity =log2log2n.

**Example 2:**

//sqrt can be replaced by any other constant root

for (int i = n; i > 0; i = sqrt(i)) {

//Some instructions with time complexity O(1)

}

    Similarly as above.

4. The time complexity of the below function is O(n!).

**Example:**

void timenfact ( int n){

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

timenfact(n - 1);

}

}

 In the next segment, we will learn more about Big O functions.

#### Q16: Rule of Products

What is the Big O of the following function?

public void function(int n){

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

int j = 0;

while(j < n){

j++;

}

}

}

Ans: O(n2)

**✓ Correct**

**Feedback:**

The for loop (i) is iterated n times, and for every iteration of the for loop (i), j is initialised with 0 and the while loop is executed n times. So, this function has a time complexity of O(n2).

#### Q17: Rule of Products

What is the Big O of the following function? Hint: Look out for the iterations of the loops.

public void function(int n) {

for(int i = 0; i < n\*n; i++){

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

System.out.println("Hello");

}

}

}

Ans: O(n3)

**✓ Correct**

**Feedback:**

The outer loop (i) is iterated n2 times, whereas the inner loop (j) is executed n times for every single iteration of the outer loop. So, the time complexity belongs to O(n3).

#### Q18: Algorithm

Now that you have learnt about the Big O notation, find the Big O of both the algorithms that were explained to find duplicate student IDs.

**Algorithm 1**

public void findDuplicates(int[] id) {

System.out.println("Duplicate student id : ");

for (int i = 0; i < id.length; i++) {

for (int j = i+1; j < id.length; j++) {

if (id[i] == id[j]) {

System.out.print(id[i] + " ");

break;

}

}

}

}

**Algorithm 2**

public void findDuplicates(int[] id) {

System.out.println("Duplicate data : ");

int count[] = new int[10000];

for (int i = 0; i < id.length; i++) {

count[id[i]]++;

if (count[id[i]] == 2)

System.out.print(id[i] + " ");

}

System.out.println();

}

Ans: Algorithm 1: O( n2 ), algorithm 2: O(n)

**✓ Correct**

**Feedback:**

Algorithm 1 has two for loops, in which a certain instruction set is executed n(n - 1)/2 times. Algorithm 2 has only one for loop, in which a certain instruction set is executed n times. Therefore, algorithm 1 is O(n2) and algorithm 2 is O(n).

Q19: **More Practice!**  
1. What is the time of the following code :

**int** a = **0**, b = **0**;

**for** (i = **0**; i < N; i++) {

**for** (j = **0**; j < N; j++) {

a = a + j;

}

}

**for** (k = **0**; k < N; k++) {

b = b + k;

}

**Solution:**  
The first set of nested loops is O(N^2) and the second loop is O(N).  
This is O( max(N^2, N) ) which is O(N^2).

2. What is the time complexity of the following code :

**int** a = **0**;

**for** (i = **0**; i < N; i++) {

**for** (j = N; j > i; j--) {

a = a + i + j;

}

}

**Solution:**  
Total number of runs = N + (N - 1) + (N - 2) + ... 1 + 0  
= N \* (N + 1) / 2  
= 1/2 \* N^2 + 1/2 \* N  
= O(N^2) times.

3. What does it mean when we say that an algorithm X is asymptotically more efficient than Y?  
**Solution:**  
In asymptotic analysis we consider growth of algorithm in terms of input size. An algorithm X is said to be asymptotically better than Y if X takes smaller time than y for all input sizes n larger than a value n' where n' > 0

# Big O Functions

**Note:**

In the video given above, at 2:15, professor intends to say “As seen in this table, it takes a few minutes to run”, not “It takes few days to run”.

Also, at 2:39, professor intends to say “O(log n) algorithm which is faster than O(n log n)”, not “O(log n) algorithm which is faster than O(logn)”.

You now know why you need an efficient algorithm to accomplish any task. This is because, as Prof. Murali explained, for large input size (n) values, algorithms may take years to process an output, making the entire process highly inefficient.

**Different types of and most often encountered asymptotic notations are as follows:**

1. O(1)
2. O( logn)
3. O(n)
4. O(n logn)
5. O(n2)
6. O(n2 logn)
7. O(n3)
8. O(2n)
9. O(n!)

**The examples of different Big O functions are as follows:**

T(n)=6∈O(1)—  Constant function

T(n)=logn+6∈O(logn)— Logarithmic function

T(n)=3n+4∈O(n)— Linear function

T(n)=3n2+5n+4∈O(n2)—  Quadratic function

T(n)=5n3+4n2+3n+2∈O(n3)— Cubic function

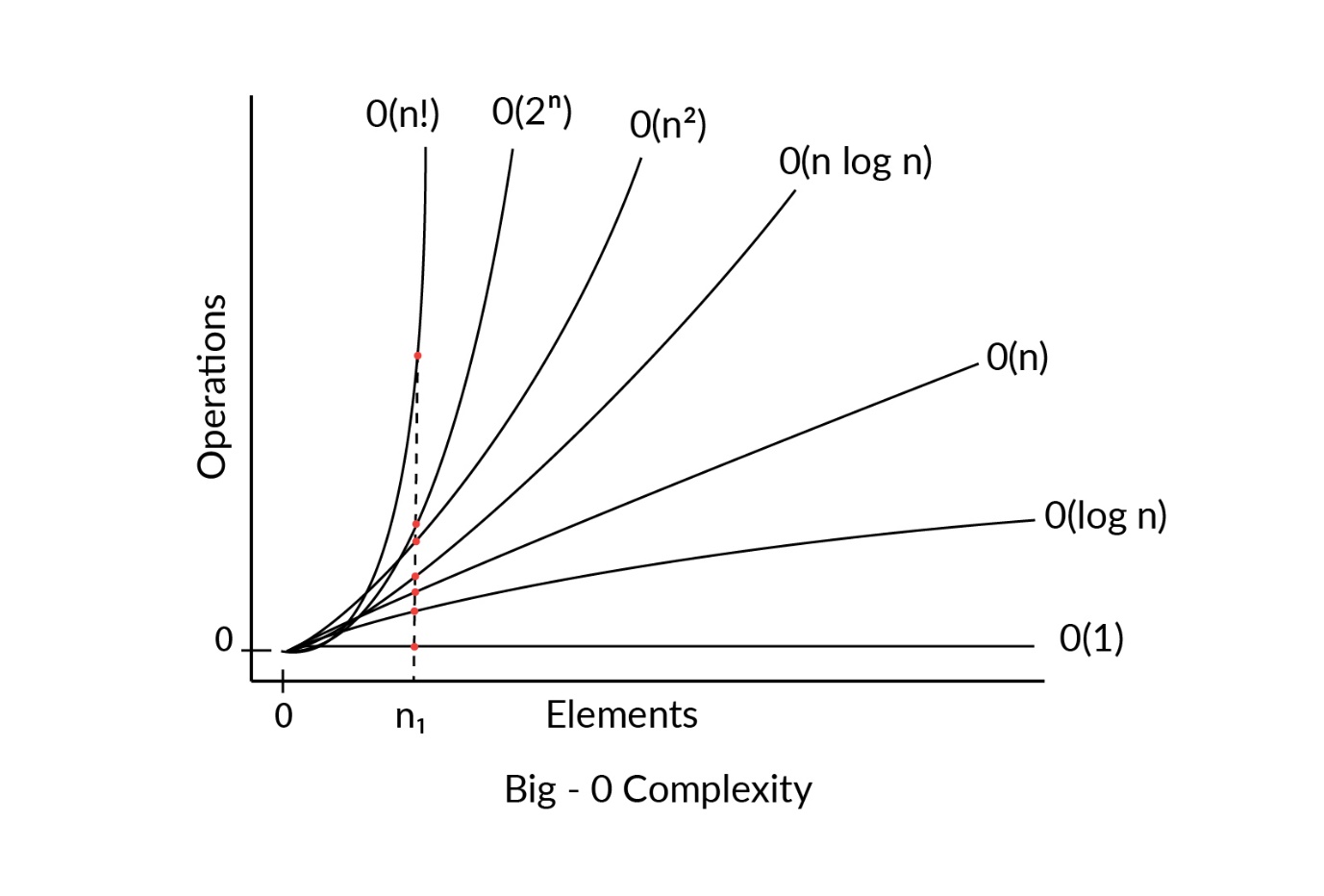
T(n)=2n+n2+4∈O(2n)— Exponential function

Any polynomial function with the degree k∈O(nk)

**The relative efficiencies of the different Big O functions are given below.**

O(1)>O(logn)>O(n)>O(nlogn)>O(n2)>O(n3)>O(2n)>O(n!)

**Graphs comparing different types of asymptotic notations are given below.**



Comparison of different Big O functions

In the image given above, when the number of elements is n,  we can conclude that the relative time complexity of different big O functions is  O(n!)>O(2n)>O(n3)>O(n2)>O(nlogn)>O(n)>O(logn)>O(1). Also, the number of operations made by the program follow the same order.

In the next segment, you will learn about how to decide between two algorithms with different time and space complexities.

#### Q20: Algorithms

You learnt about the order of efficiency of different functions. Which of the two algorithms that are used to find duplicate elements in an array is more efficient with respect to the time taken?

Ans: Algorithm 2 : O(n), using count array variable

**✓ Correct**

**Feedback:**

Let us assume one operation takes m seconds  
O(n) means for one input element, it performs one operation. Similarly, for n input elements, it performs n operations, and so, the time taken to perform n operations is n \* m seconds.  
O(n2) means that for n input elements, it performs n2 operations, and so, the time taken here is n2 \* m seconds.  
O(n2) takes more time than O(n) despite both being given the same number of input elements. Therefore, O(n) is efficient.

#### Q21: Time Complexity

Arrange the following functions in the order of **decreasing** efficiency, and choose the correct option:

1. T(n)=6n2+2n+3
2. T(n)=45
3. T(n)=5n+4

Hint: Recall the order of efficiency discussed in this segment and the rules for big calculating Big O.

Ans: 2, 3, 1

**✓ Correct**

**Feedback:**

On dropping the constant multiplier and less significant terms, the order of the first function is O(n2), that of the second function is O(1) and that of the third function is O(n). So, O(1),O(n)  
and O(n2).

# Time vs Space Complexity Trade-off

In the previous segment, you learnt how to find the relative efficiency of multiple functions. In this segment, you will compare the complexity functions for two algorithms, which will help you in finding the efficient algorithm to identify duplicate student IDs from the given data.

**Time complexity:**

Algorithm 1:  T(n)=C1+n(n−1)2∗C2  ⇒ T(n) ∈ O(n2) - Quadratic function

Algorithm 2:  T(n)=C1+n∗C2 ⇒ T(n) ∈ O(n) - Linear function

You can observe that algorithm 2 with time complexity of  O(n) is more efficient than algorithm 1with time complexity of  O(n2).

**Space complexity:**

Algorithm 1: S(n) ∈ O(1) - Constant function

Algorithm 2: S(n) ∈ O(n) - Linear function

You can observe that algorithm 1**—**O(1) is more efficient than algorithm 2**-**O(n).

As you have seen, the two algorithms for identifying duplicate student IDs have made different trade-offs in terms of time and space.

Specifically:

* Algorithm 1 runs slower but uses less memory.
* Algorithm 2 runs faster but uses more memory.

As a software developer, you will often face this kind of dilemma while designing programs and creating software. Do you want to write programs that run fast but use lots of memory space? Or do you want to write programs that use less memory space but run slower?

The answer to this question is circumstantial.

For example, if you are creating a piece of software to perform high-frequency stock trading wherein every microsecond can be the difference between earning and losing hundreds of thousands of dollars, you will likely want to design programs that can execute extremely quickly at the expense of using excessive memory space.

On the other hand, you may be creating a piece of software that runs on smartphones, wherein the memory allocated to the software is limited. In this situation, you may want to create a piece of software that uses less memory at the expense of running it slightly more slowly.

Therefore, use your best judgement when it comes to ‘time vs space complexity’ trade-offs. Identify your business needs and constraints and then determine whether you should trade space for time or vice versa.

# Summary

You learnt how to analyse algorithms using mathematical tools such as Big O and how to compare the Big Os of two different algorithms and find the more efficient of the two.

The parameters that determine the efficiency of an algorithm are the **time taken** (running time) and the **memory space** required to execute an algorithm and are calculated as functions of the input size. Also, when faced with 'time vs space complexity’ trade-offs, identify your business needs and constraints and then determine whether you should trade space for time or vice versa.

The three cases in the analysis of an algorithm are as follows:

1. Best case: Lower bound on the time taken by an algorithm **(Big Omega (Ω))**
2. Average case: Average of time taken by the algorithm for all the inputs **(Big theta**(θ)**)**
3. Worst case: Upper bound of the time taken by an algorithm **(Big O)**

**Most of the times, the worst case time complexity (Big O) of an algorithm is used for algorithm analysis because it guarantees the upper bound of the time that the algorithm takes.**

**The common rules for calculating Big O are as follows:**

* **Rule of sums:** If two loops come one after another, you need to add the number of execution steps for each loop.
* **Rule of products:** If two loops are nested, then you need to multiply the number of execution steps for each value.

The relative efficiencies of the different Big O functions, as discussed in the previous segment, are as given below.

O(1)>O(logn)>O(n)>O(nlogn)>O(n2)>O(n3)>O(2n)>O(n!)

With this, you have successfully completed this session. In the following video, you will hear from Rachit about the next session.

#### Q22: Algorithms

The id[ ] array is as follows:

id [ ]

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| 5 | 1 | 3 | 5 | 6 | 2 | 4 |

Choose the correct count [ ] array after executing the program using algorithm 2 (as discussed in segment 2).

Ans: count [id[i]] - [0, 1, 1, 1, 1, 2, 1]

**✓ Correct**

**Feedback:**

The count array is initialised with 0 and gets incremented in the cell corresponding to the student ID number. If the same student ID is repeated twice, then the corresponding cell of the count array is incremented to 2.

#### Q23: Rule of Products

What is the Big O of the following function?

public void function(int n){

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

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

System.out.println("Hello");

}

}

}

Ans: O(n2)

**✓ Correct**

**Feedback:**

For each iteration of the outer loop (i), the inner loop instruction set is executed i times. The outer loop (i) is executed n times. So in total, the inner loop (j) instruction set is executed n(n+1)/2 times. Therefore, the Big O of this function is O(n2).

#### Q24: Rule of Products

What is the Big O of the following function? (Hint: Carefully observe the declaration of variables in the for loops and evaluate the number of iterations accordingly.)

public void function(int n){

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

for(int j=2; j < n; j\* =2){

System.out.println("Hello");

}

}

}

Ans: O(n log2n)

**✓ Correct**

**Feedback:**

In this algorithm, the outer loop (i) is iterated n times, whereas the inner loop (j) is multiplied by two after each iteration. So, the inner loop instruction set executes when the variable j is an integer power of 2, i.e., j = 2, 4, 8, 16… for every iteration of the outer loop(i). Therefore, the inner loop(j) instruction set is executed log2n times; hence, the Big O of this function is O(n log n).

#### Q25: Time Complexity

Arrange the following functions in the descending order of their efficiencies, and choose the correct option:

1. T(n)=logn+n2+3
2. T(n)=2n3+3n2+5n+6
3. T(n)=logn+2

Hint: Recall the order of efficiency of different Big O functions and the rules for big calculating Big O as discussed in the previous segments.

Ans: 3, 1, 2

**✓ Correct**

**Feedback:**

On dropping the constant multipliers and less significant terms, the order of the first function is O(n2), that of the second function is O(n3) and that of the third function is O(log n). So, O(log n), O(n2), and O(n3).

#### Q26: Algorithms

Define the best case for algorithm 1 and algorithm 2 in finding duplicate student IDs. What are their best case-time complexities?

And: The best case for algorithm 1 is when only one student ID is repeated n times throughout the count array. For each iteration of the outer loop (i), the inner loop (j) is executed once; it prints the duplicate student IDs and then breaks out of the inner loop (j). In total, the instruction set is executed n times. Therefore, the best case-time complexity of algorithm 1 is T(n)∈O(n). The best case for algorithm 2 is the same as its worse case for any given data. The instruction set inside the loop is always executed n times. Therefore, T(n) ∈O(n).

**Run-time Analysis**

# Solving Sample Problems 1

Now that we have gained sufficient understanding of the concepts of time complexity as well as asymptotic notations, let's try to calculate the time and space complexity of some examples.

**Example 1**

What would be the time and space complexity of the following code?

**int** p = **0**, q = **0**;

**for** (**int** i = **0**; i < N; i++) {

p = p + **1**;

}

**for** (**int** j = **0**; j < M; j++) {

q = q - **1**;

}

**Solution:**

* If we look at the code carefully, we can see there are two loops running one after the other. The first loop can run a maximum of N times, so the time complexity for loop 1 would be O(N). Similarly, the time complexity of loop 2 would be O(M).
* Since we don't know which of the two time complexities is bigger and both of them run one after the other, the total time complexity of the code would be O(N+M).
* For calculating the space complexity, we can see apart from the declared variables, nothing additional is being stored and it is independent of the size of the input. So the space complexity of the code would be constant or O(1).

**Example 2**

What would be the time complexity of the following code?

**int** p = **0**, q = **0**;

**for** (**int** i = **0**; i < N; i++) {

**for** (**int** j = **0**; j < N; j++) {

p = p + j;

}

}

**for** (**int** k = **0**; k < N; k++) {

q = q + k;

}

**Solution:**

* Here again there are two loops running one after the other.
* However, the first loop (loop 1.1) is a nested loop which further contains another loop (loop 1.2). The time complexity of loop 1.2 would be O(N) since it can run a maximum of N times. Now we know one instance of loop 1.2 will run a maximum of N times. But loop 1.2 itself can be called a maximum of N times by loop 1.1. So the time complexity of loop 1 will be O(N∗N) or O(N2).
* Loop 2 is a simple loop with a time complexity of O(N).
* So the total complexity of the code would become O(N2)+O(N).
* We know O(N2) is exponentially more that O(N), so for higher values of N, the value of O(N) will be neglible compared to the value of O(N2). So the time complexity of the code will become O(N2).

**Example 3**

What would be the time complexity of the following code?

**int** p = **0**;

**for** (**int** i = **0**; i <N; i++) {

**for** (**int** j = **1**; j < N; j = j \* **2**) {

p = p + N;

}

}

**Solution:**

* In the code given above, we see there is a nested loop.
* If we see the internal loop, we can understand that the internal loop will not run a maximum of N times since the iterator is doubling itself instead of getting increased by 1
* Since the increase in the iterator is exponential, the loop will only run a maximum of O(logN) times. The outer loop can run a maximum of N times (calling inner loop on every iteration).
* Therefore the time complexity of the code will be O(N∗logN)or O(NlogN).

# Solving Sample Problems 2

**Example 1**

What would be the time complexity of the following code?

**int** p = **0**, i = N;

**while** (i > **0**) {

p = p\*i;

i /= **2**;

}

Since there is only 1 while loop in the code, the time complexity of this loop will be the time complexity of the code. If we look carefully, we find that the iterator i will not iterate N times since it is getting reduced by a factor of 2. We know that if the loop variable is multiplied or divided by a constant amount on every iteration, the time complexity of that loop becomes O(logN). Therefore the time complexity of the given code will be O(logN).

**Example 2**

What would be the time complexity of the following code?

**int** a = **0**;

**for** (**int** i = N; i > **0**; i /= **2**) {

**for** (**int** j = **0**; j <N; j++) {

a+=j;

}

}

In the code given above, the iterator in outer loop is getting divided by a constant value, so the time complexity of the outer loop will be O(logN). The inner loop would run a maximum of N times for every iteration of the outer loop. Therefore the total number of times the loop is running is logN∗N. The time compelxity of the code would therefore be O(NlogN).

**Example 3:**

What would be the time complexity of the following code?

**int** a = **0**;

**for**(**int** i = N; i > **0**; i /= **2**)

{

a += j;

}

**for**(**int** j = **0**; j < N; j++)

{

a += j;

}

This example is similar to Example 2, however, in this example, the loops run sequentially instead of being nested. The first loop has the time complexity O(logn), while the second loop has the time complexity O(n). As the loops are sequential, you will add their time complexities and choose the term that has the maximum impact on the running time of the program. In this scenario, the second loop impacts the running time of the program the most; hence, the time complexity is O(n).

**Example 4**

What would be the time complexity of the following code?

**int** a = **0**;

**for** (**int** i = N; i > **0**; i /= **2**) {

**for** (**int** j = **0**; j < i; j++) {

a+=j;

}

}

On the first glance this code looks familiar. There are two loops, one nested inside the other. The iterator in the outer loop is getting divided by a constant value so the time complexity of outer loop will be O(logN). The inner loop is a fairly simple loop with the time complexity O(N). Therefore the time complexity of the code should be O(logN∗N) or O(NlogN), right?

Unfortunately this is wrong.

Although the time complexity of outer loop is O(logN), we see the iterator of inner loop (j) depends upon the value of outer loop iterator (i).

The value i goes like N, N/2, N/4, N/8..... etc, so the inner loop does not run a fixed number of times.

By dry running the loop for a couple of values for N, we find that the code will loop for a maximum of 2N times. Therefor the time complexity of the code is O(N).

# Solving Sample Problems 3

**Example 1**

What would be the time complexity of the following code?

**int** **recursiveFun**(**int** n)

{

**if** (n <= **0**)

**return** **1**;

**else**

**return** **1** + recursiveFun(n-**1**);

}

In the given code, the stopping condition is n<=0. This means the function will call itself recursively unless the value of n is less than or equal to 0. The function calls itself n times before reaching the stopping criteria therefore the time complexity of the function will be O(n).

**Example 2**

What would be the time complexity of the following code?

**int** **recursiveFun**(**int** n)

{

**if** (n <= **0**)

**return** **1**;

**else**

**return** **1** + recursiveFun(n/**5**);

}

In the given code, the stopping condition is n<=0. This means the function will call itself recursively unless the value of n is less than or equal to 0. If we look carefully, the function is again dividing the iterator (n) by a constant value (5). Therefore the time complexity of the function will become O(logN).

**Example 3**

What would be the time complexity of the following code?

**void** **recursiveFun**(**int** n, **int** m, **int** o)

{

**if** (n <= **0**)

{

cout<<m<<o;

}

**else**

{

recursiveFun(n-**1**, m+**1**, o);

recursiveFun(n-**1**, m, o+**1**);

}

}

In the given code, the stopping condition is n<=0. This means the function will call itself recursively unless the value of n is less than or equal to 0. Here we can see the function is calling itself twice recursively. The time complexity of both the recursive calls would be O(n)  because the stopping condition is only based on value of n. Due to two recursive calls on every recursion, the time complexity of the code will be O(2n)

**Example 4**

What would be the time complexity of the following code?

**int** **recursiveFun**(**int** n)

{

**for** (i = **0**; i < n; i += **2**) {

cout<<"Linear function called";

}

**if** (n <= **0**)

**return** **1**;

**else**

**return** **1** + recursiveFun(n-**5**);

}

In the given code there is a for loop in addition to a recursion. Let's proceed to calculate the time complexity step by step.

In the for loop, the iterator is getting incremented by 2, which means the loop will run a maximum of n/2 times. Therefore the time complexity of for loop would be O(n/2) or simplyO(n).

For the recursive call, the stopping condition is n<=0. This means the function will call itself recursively unless the value of n is less than or equal to 0. If we look carefully, the function is getting reduced by a constant value of 5. Therefore the time complexity of the function will become O(n/5) or simply O(n).

The time complexity of the function would be O(n/5∗n/2) i.e. O(n2/10) or simply O(n2).

# Fibonacci Sequence

Let's begin with the Fibonacci algorithm. In the coming few segments , we will see how we can optimise it using different algorithms.

Before learning how to devise an algorithm for Fibonacci sequence, go through the role of modulo(%) operator given below, which is used by the professor in the video given below.

**The role of the %(modulo) operator:**

The modulo operator gives the remainder when a number is divided by another number. **a%b**gives the remainder when **a** is divided by**b**,for example, 7%2 gives 1 because when 7 is divided by 2, the remainder will be 1. When a number is divided by 10, the remainder will always be the last digit of that number. So, any number %10(modulo 10) gives the last digit of that number as the output.

**Examples:**  
132%10 gives an output of 2.  
45%10 gives an output of 5.

#### Q27: Fibonacci Sequence

Calculate F(8) for the following sequence: F(n) = F(n-1) + F(n-2),

where F(0) = 0 and F(1) = 1.

**Note:**The function given above does not have the %(Modulus) operator.

Ans: 21

**✓ Correct**

**Feedback:**

You know that F(0) = 0 and F(1) = 1. Using the equation F(n) = F(n - 1) + F(n - 2), you can find F(2) = 1, F(3) = 2, F(4) = 3, F(5) = 5, F(6) = 8, F(7) = 13, and F(8) = 21.

#### Q28: Recursion

Explain, in your own words, how recursion works in the above algorithm to generate the Fibonacci number.

Ans: For an intenger n for which we require the fibonacci number, the  function calls itself recursively until n>2 and returns the value forsum of  f(n-1) and f(n-2). So when n reaches 2, f(2) i.e 1 is returned. That implies for n=3, f(2) +f(1) will be returned which is 1+0=1; for n=4 , f(3)+f(2) will be returned, which is now 1+1=2; for f(4), f(3)+f(2) is returned in which f(3) and f(2) are already calculated and hence the value for f(4) will be 2+1= 3.

#### Q29: Recursion

Discuss what happens if the terminating condition n < 2 is removed from the algorithm

Ans: The terminating condition helps to give a definite answer, i.e., either F(0) = 0 or F(1) = 1, and breaks the recursive loop caused, as the function is not called again inside the same function. If there is no terminating or base case condition, then it is called infinite recursion, which causes stack overflow. The terminating or base case condition is a must for any recursive program to execute.

# Complexity Analysis of Algorithm 1

**Space complexity**

To conduct a space complexity analysis of a recursive algorithm, you need to first learn how a recursive program executes in memory to analyse the memory space required to analyse the algorithm.

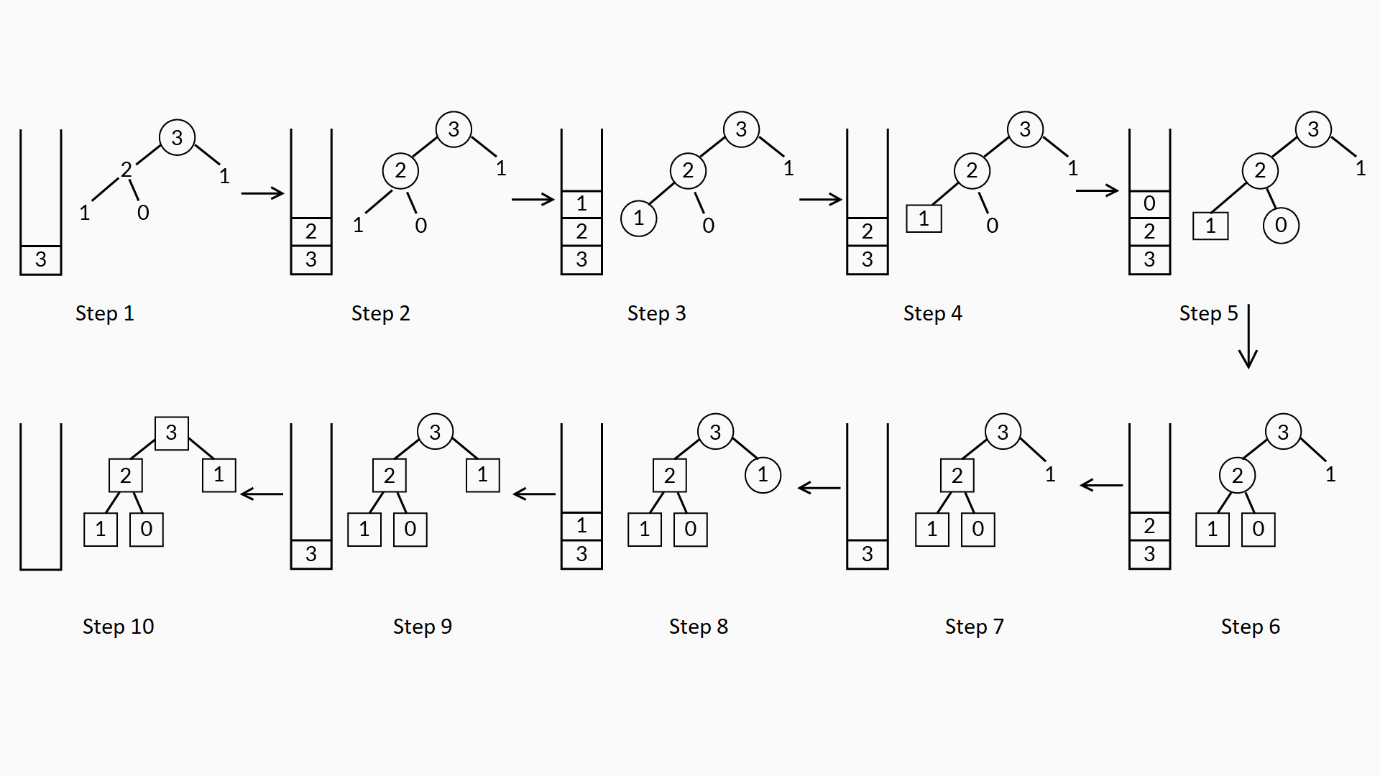
Suppose you have some partitioned memory as shown, and each part is one unit of memory.

|  |
| --- |
|  |
|  |
|  |
|  |
|  |
|  |

Assume that you are generating only three Fibonacci numbers using the function fibonacci(3), and this function is called from the main function. When a method is executed currently, it is saved in the memory so that all the information of the variables and operations is also saved.

Currently, the main function is being executed, excluding the main function. Let's learn how the Fibonacci function is executed in the memory.

**Information related to the image given below:**  
Circles indicate that the function is in the memory stack.  
Squares indicate that the function completed its execution and popped out of the memory stack.



Example

When the main function calls fibonacci(3), this function is executed in the memory, and main() is paused.  
Given below are the steps that correspond to the steps shown in the image given above:

1. fibonacci(3) starts executing and calls fibonacci(2).
2. fibonacci(2) now calls for fibonacci(1).
3. In fibonacci(1), since the function variable is 1, the base case of the fibonacci function is executed, and it terminates the recursion.
4. fibonacci(1) gets popped from the memory stack, then fibonacci(2) calls fibonacci(0)
5. In fibonacci(0) since function variable is 0, the base case of the fibonacci function is executed, and it terminates the recursion.
6. fibonacci(0) gets popped from the memory stack.
7. Since the fibonacci(2) completes its execution, it gets popped out of the memory stack, and then, fibonacci(3) calls fibonacci(1).
8. In fibonacci(1), since the function variable is 1, the base case of the fibonacci function is executed, and it terminates the recursion.
9. fibonacci(1) gets popped from the memory stack.
10. Since the fibonacci(3) completes its execution, it gets popped out of the memory stack.

Finally, the main function resumes with all the called functions popped out of the memory.

The same process of memory allocation to generate three numbers can be extended to n numbers of the Fibonacci series. In this recursion algorithm, the maximum memory used is proportional to the number generated in the function:

F(0) = 0

F(1) = 1

F(n) = [F(n - 1) + F(n - 2)]%10, n > 1

Therefore, the space complexity is a O(n)-linear function with respect to the input size.

#### Q30: Time Complexity

In your own words, explain how the equation T(n) = T(n - 1) + T(n - 2) + 1 is formed.

Ans: T(n): No. of additions to calculate F(n); when calculating F(n), the function recursively calls F(n - 1) and F(n - 2), which will take T(n -1 ) and T(n - 2) respectivelyT(n - 1): No. of additions to calculate F(n - 1)T(n - 2): Number of additions to calculate F(n - 2)Also, when calculating F(n), you need to add the previous two numbers F(n - 1) and F(n - 2). The result is counted as one more addition besides T(n - 1) and T(n - 2). So, you get the following equation:

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

#### Q31: Time Complexity

In your own words, explain the change in the inequality from T(n)≤2∗T(n−1) to T(n)≤2∗2∗T(n−2).

Ans:

3776789

**Question 1/1**

Mandatory

#### Time Complexity

In your own words, explain the change in the inequality from T(n)≤2∗T(n−1) to T(n)≤2∗2∗T(n−2).

Word Count **9**Word Limit **1 - 500**



**Suggested Answer**

As you have learnt previously, to calculate upper bound of the equation: T(n)=T(n−1)+T(n−2)+1

To calculate T(n−1), substitute n−1 in the place of n in the above equation.

Consider the equation T(n−1)=T(n−2)+T(n−3)+1

                                                                    ⩾T(n−2)+1(BecauseT(n−3)⩾0)

Now, in the equation T(n)=T(n−1)+T(n−2)+1, replace T(n−2)+1 with T(n−1) Since   
T(n−2)+1≤T(n−1),

                                    so, T(n)≤T(n−1)+T(n−1)

                                                        ≤2∗T(n−1)

Similarly, to calculate upper bound for the equation T(n−1)=T(n−2)+T(n−3)+1

To find the value ofT(n−2), substitute n-2 with n−2 in the first equation.

Consider the equation T(n−2)=T(n−3)+T(n−4)+1

                                                                    ⩾T(n−3)+1(BecauseT(n−4)⩾0)

Now, in the equation T(n−1)=T(n−2)+T(n−3)+1, replace T(n−3)+1 with  T(n−2)because T(n−3)+1  T(n−2).

                                    so, T(n−1)≤T(n−2)+T(n−2)

                                                        ≤2∗T(n−2)

In the inequality T(n)≤2∗T(n−1), substitute T(n−1) with 2∗T(n−2)

T(n−1)≤2∗T(n−2)  from above; multiply with 2 on both sides  
2T(n−1)≤2∗2∗T(n−2)  
T(n)≤2∗T(n−1)  
By observing the two expressions given above, we get the following:  
T(n)≤2∗2∗T(n−2)    since(if a≤b and b≤c then a≤c)

Therefore, T(n)≤2∗2∗T(n−2).

#### Q32: Time Complexity

Can you explain the reason why the algorithm has not given any output for n = 100 for so long when executed by the professor in the previous segment?

Ans: As you learnt already, the algorithm follows an exponential time of O(2n). For an input size of n = 100, this will take around 2100 additions, which will take some time to process the output. Most compilers have some time limit to process the output after which the time limit given will be exceeded. Therefore, using this algorithm, you will not be able to find the outputs for large values.

# Recursion

Recursion is an important programming technique that is used to solve many critical real-world problems. Now that you have understood and analysed the recursive ‘algorithm 1’ in detail, let’s consider an example.

**The program to find the factorial of a number using recursion is given below.**

**import** **java.util.Scanner**;

**public** **class** **Demo** {

**public** **static** **int** factorial(**int** n) {

**if** (n == 0)

**return** 1;

**else**

**return** n \* factorial(n - 1);

}

**public** **static** **void** main(String args[]) {

Scanner sc = **new** Scanner(System.in);

System.out.println(*"Enter a positive no: "*);

**int** n = sc.nextInt();

System.out.println(*"Factorial of "* + n + *" is : "* + factorial(n));

}

}



Let's assume that the total time taken by the factorial function given above to calculate the factorial oinf n  is  T(n).  
On analysing the factorial function given above:  
In 'if (n==0)', comparison takes a constant time ‘c’.  
In 'return n∗ factorial(n−1);', multiplication takes a constant execution time ‘c’, and subtraction in (n−1) takes a constant execution time ‘c'.  
Since the time taken by the factorial function given above to calculate the factorial of n is T(n), the time taken by the factorial function given above to calculate the factorial of n−1 is T(n−1).  
   
When n=0 in the above factorial function, T(n)=c, because only comparison is done in 'if (n==0)'. This implies that T(0)=c.  
   
So, the total time complexity of the above factorial function is as follows:  
T(n)=c+c+c+T(n−1)  
=3∗c+T(n−1)       ----------------- 1st step  
On substituting n with n−1 in the equation given above, you will obtain the following:  
T(n−1)=T(n−2)+3∗c  
On substituting this T(n−1) in the first equation, you will obtain the following:  
T(n)=6c+T(n−2)      ----------------- 2nd step  
          
Similarly,  
T(n)=T(n−k)+3∗k∗c      ----------------- kth step  
   
Let's assume that the kth step is the final step.  
The recursion will terminate only when n=0. So, T(n−k) should be equal to T(0).  
⟹n−k=0  
⟹n=k  
By substituting n=k in the kth step, you will obtain the following:  
You get T(n)=T(0)+3∗n∗c  
T(n)=c+3∗n∗c (since T(0)=c)   
Let constant time c=1, then:  
T(n)=1+3n  
By excluding the constant terms you will obtain the total time complexity as follows:  
O(n)

Now, let’s take a look at a real-world problem where recursion plays a critical role.

Q33: Write an algorithm regarding how a file can be searched for in a file directory(folder) using recursion.

Ans: You can use recursion to search for a file in its directory.First, pass the file that you are looking for and the file directory path(folder path) where you want to start the search to the recursive function.Then, follow these steps inside the recursive function:1. List all the contents of the file directory that are passed to the function.2. Loop through the content inside the file directory.a. If any part of the content is another file directory(folder), recursively call the function and pass the file directory(folder) and its path through.b. If the content is a file, check whether it matches the file you are searching for. If it does, then return 'file is found' with the file path.

# Algorithm 2

Let’s get back to the mathematical function to generate numbers according to the following function:

F(0) = 0

F(1) = 1

F(n) = [F(n - 1) + F(n - 2)]%10

Typically, there is more than one way to solve the same problem, similar to how you found the duplicated IDs at the beginning of this course. Having understood and analysed the recursive algorithm in detail, let’s explore and check whether there is any other way to generate Fibonacci numbers.

Q34: Identify the major differences between the two algorithms that are used to generate the defined computational problem.

Ans: In the first algorithm, the function follows a recursive procedure to generate the Fibonacci number, but it does not store any of the values. We recalculated the same values at different instances, which increase the number of operations while executing this algorithm. However, in the second algorithm, declaring an extra array variable to store all the Fibonacci numbers reduces the redundant operations while executing the algorithm.

Algorithm 2 uses an array variable f[], stores the calculated values for each iteration of the for loop (i) and, finally, prints the nth number of the function. The time complexity of algorithm 2 is a linear function that is relatively more efficient than the exponential function of algorithm 1. Therefore, algorithm 2 is able to process the output for n = 100 in an instant.

Okay, so we have compared Algorithm 1, and Algorithm 2, but we can still do better. In the next segment we will look at another algorithm that will make our lives much easier.

# Algorithm 3

Though algorithm 2 is better than algorithm 1, the constraint with memory has to be overcome, and the algorithm should be able to calculate the function value for n = 109 and higher values.

In this video, Vishwa will explain and analyse another algorithm to calculate the values of a function for higher input values.

The run-time and space complexity of algorithm 3 is O(n) and O(1), respectively, making algorithm 3 much more efficient than the other two algorithms. Therefore, it can process the nth number of the function F(n) = [F(n - 1) + F(n - 2)] for large values of n that would otherwise be quite large for algorithm 1 and algorithm 2.

The given file contains the code for algorithm 3; in the next video, Vishwa will explain this in detail.

Q35: How did algorithm 3 overcome the memory space constraint compared with algorithm 2?

Ans: Algorithm 2 stores all the values in an array variable f[ ], whereas in algorithm 3, there are only three variables, and algorithm 3 only stores the required values F(n), F(n - 1) and F(n - 2). This overwriting of F(n - 2) at every iteration allows the algorithm to save memory space and manage with only three variables.

**Summary**

Algorithm 1: T(n) ∈ O(2n) – Exponential time  
                         S(n) ∈ O(n) – Linear space  
  
Algorithm 2: T(n) ∈ O(n) – Linear time  
                         S(n) ∈ O(n) – Linear space  
  
Algorithm 3: T(n) ∈ O(n) – Linear time  
                         S(n) ∈ O(1) – Constant space

Algorithm 3 is more efficient than algorithm 2 and algorithm 1 in terms of both time complexity which is O(n) and space complexity which is O(1) .

#### Q36: Big O

In comparison with the usual Big O functions, where can you position the exponential time in the decreasing order of efficiency?

Ans: O(1) > O(log n) > O(n) > O(n2) > O(2n)

**✓ Correct**

**Feedback:**

Consider the input size n = 100. Then, substitute it in all of the functions given above and arrange them in order from the smallest to the largest.

#### Q37: Algorithms

Check the following algorithms to calculate the sum of the even numbers until n and answer the questions given below.

Algorithm 1

public static int evenSum(int n) {

if(n <= 1)

return 0;

else if (n % 2 == 0)

return (n + evenSum(n - 1));

else

return evenSum(n - 1);

}

Algorithm 2

private static int evenSum(int n){

return (n/2)\*(1 + n/2);

}

Ans: Both algorithms are correct.

**✓ Correct**

**Feedback:**

In algorithm 1, for a given n value, the first if condition verifies whether the value is an even or an odd number. If it is an even number, it is added to the sum and recursively calls the function evenSum(n - 1). If it is an odd number, then it recursively calls the function evenSum(n - 1) and does not add the odd numbers. In algorithm 2, you use a formula (n/2)\*(1+n/2) to calculate the sum of the even numbers.

#### Q38: Agorithms

Go through the algorithms mentioned below and answer the question that follows.

Algorithm 1

public static int evenSum(int n) {

if(n <= 1)

return 0;

else if (n % 2 == 0)

return (n + evenSum(n - 1));

else

return evenSum(n - 1);

}

Algorithm 2

private static int evenSum(int n){

return (n/2)\*(1 + n/2);

}

Which of the following options is correct based on the algorithms given above?

Ans:   
Algorithm 2 is faster than algorithm 1 (O(1) is better than O(n)).

**✓ Correct**

**Feedback:**

Algorithm 2’s instruction set uses a simple formula and calculates the sum of the even numbers in constant time or O(1). Algorithm 1’s instruction set calls the function recursively n times to calculate the sum of the even numbers or O(n).

#### Q39: Algorithms

Algorithm 1

public static int evenSum(int n) {

if(n <= 1)

return 0;

else if (n % 2 == 0)

return (n + evenSum(n - 1));

else

return evenSum(n - 1);

}

Algorithm 2

private static int evenSum(int n){

return (n/2)\*(1 + n/2);

}

Which of the following options is correct based on the algorithms given above?

Ans: Algorithm 1 has a higher space complexity than algorithm 2 (O(n) compared with O(1)).

**✓ Correct**

**Feedback:**

Algorithm 1 calls the function evenSum recursively n times. So, the maximum memory required is the linear space complexity or O(n). Algorithm 2 requires a constant space to calculate the sum of the even numbers or O(1).

# Prime Numbers Less Than Or Equal To n (Part I)

You will be given an integer n. You need to print all the prime numbers that are less than or equal to n, starting from the smallest prime number.

**Input:**

19

**Output:**

2 3 5 7 11 13 17 19

#### Q41: Time Complexity - Approach I

What is the time complexity of the following pseudocode snippet?

PRIMALITY\_CHECK (num)  
1. for each i in [2, num)  
     A. if num is divisible by i  
          i) return false

2. return true  
  
PRINT\_PRIMES(n)  
1. for each num in [2, n]  
     A. if PRIMALITY\_CHECK(num) returns true  
          i) Print num

(Hint: Check for the range across which the loop iterates and compute the number of iterations.)

Ans: O(n2)

**✓ Correct**

**Feedback:**

The outer loop goes from num = 2 to n. The inner loop goes from 2 to num-1, thus being executed num-2 times.   
For num = 2, the inner loop is executed (num-2) = (2-2) = 0 times.  
For num = 3, the inner loop is executed (num-2) = (3-2) = 1 time.  
For num = 4, the inner loop is executed (num-2) = (4-2) = 2 times.  
For num = 5, the inner loop is executed (num-2) = (5-2) = 3 times.  
...  
For num = n, the inner loop is executed (num-2) = (n-2) times.  
  
Thus, the time complexity is the sum of all the times the inner loop is executed.  
= 1 + 2 + 3 + ... + (n-2)  
= (n−2)(n−1)2  
= n2−3n+22

= n22−3n2+1

The time complexity is greater of O(n2) and O(n).

Therefore, the time complexity is O(n2), and this option is the correct answer.

#### Q42: Printing all prime numbers less than or equal to n - Approach I

Write the pseudocode to print all prime numbers less than or equal to the given number n.

Ans: PRIMALITY\_CHECK (num)  
1. for each i in [2, num)  
     A. if num is divisible by i  
          i) return false

2. return true  
  
PRINT\_PRIMES(n)  
1. for each num in [2, n]  
     A. if PRIMALITY\_CHECK(num) returns true  
          i) Print num

#### Q43: Printing all prime numbers less than or equal to n - Approach II

Based on the hint given above, write the pseudocode to print all the prime numbers that are less than or equal to the given number n.

Ans: PRIMALITY\_CHECK (num)  
1. for each i in [2, √num)  
     A. if num is divisible by i  
          i) return false

2. return true  
  
PRINT\_PRIMES(n)  
1. for each num in [2, n]  
     A. if PRIMALITY\_CHECK(num) returns true  
          i) Print num

#### Q44: Time Complexity - Approach II

What is the time complexity of the following pseudocode snippet?

PRIMALITY\_CHECK (num)  
1. for each i in [2, √num]  
     A. if num is divisible by i  
          i) return false

2. return true  
  
PRINT\_PRIMES(n)  
1. for each num in [2, n]  
     A. if PRIMALITY\_CHECK(num) returns true  
          i) Print num

(Note the change in the range in which the loop is iterating.)

Ans: O(n√n)

**✓ Correct**

**Feedback:**

The outer loop goes from num = 2 to n. The inner loop goes from 2 to √num, thus being executed √num-1 times. Please check your answer again.  
For num = 2, the inner loop is executed (√num-1) = (√2-1) times.  
For num = 3, the inner loop is executed (√num-1) = (√3-1) times.  
For num = 4, the inner loop is executed (√num-1) = (√4-1) times.  
...  
For num = n, the inner loop is executed (√num-1) = (√n-1) times.  
  
Thus, the time complexity is the sum of all the times the inner loop is executed.  
= (√2-1) + (√3-1) + (√4-1) + ... + (√n-1)

= (√2 + √3 + √4 + ... + √n) - (n-1)

= C1+23n√n+12√n+1√n(124−11920n2+19216n3+....)  
(as per [this](http://ramanujan.sirinudi.org/Volumes/published/ram09.pdf) link)

Therefore, the time complexity is O(n√n), and this option is the correct answer.

# Prime Numbers Less Than Or Equal To n (Part II)

In the last segment, you learnt about two approaches to solve the problem of printing prime numbers that are less than or equal to n. Can you think of another approach to solve this problem? The hint is to use basic mathematics.

If you clearly observe, you will notice that every prime will be of the form 6k±1 except 2 and 3. As you know, every number can be represented as 6k+i for some integer k and i=0,1,2,3,4,5. In these six forms, 6k+0, 6k+2, 6k+4 will be divided by 2 and 6k+3 will be divided by 3. So, you need to check whether a number is prime or not only for 6k+1 and 6k+5, i.e., 6k-1.

**Example:**Take numbers from 6 to 11.

6 = 6 \* 1+0

7 = 6 \* 1+1

8 = 6 \* 1+2

9 = 6 \* 1+3

10 = 6 \* 1+4

11 = 6 \* 1+5

Here, 6,8 and 10 will be divided by 2 because they are even numbers, and 9 can be divided by 3, So, you need to check only 7 and 11.

#### Q45: Integer Representation

As you learnt in the video, any integer can be expressed in either of the following representations:  
6k, 6k+1, 6k+2, 6k+3, 6k+4, 6k+5

This is because the remainder of any integer, when divided by the number 6, is in the set {0, 1, 2, 3, 4, 5}.  
  
**Can you think why have we used the number 6 only?**  
  
Why didn't you use another number (for example, the representation of the format 2k and 2k+1 or the representation of the format 3k, 3k+1 and 3k+2)?

Ans: ?

you learnt that every prime number will be either of the form **6k+1** or of the form **6k-1**. Based on this fact let's devise an approach to print all prime numbers less than or equal to n.

#### Q46: Checking if a number is prime or not

Based on the hint given above, write a pseudocode for checking whether the given number is prime or not, which is more efficient than the pseudocode provided previously.

Ans: function(n)

     if n is less than or equal to 1

          return false

     if n is less than or equal to 3

          return true

     if n is divisible by 2 or 3

           then return false

     initialise i equal to 5

     while

i

 is less than or equal to n

2

          if n is divisible by i or i+2

               then return false

          add 6 to i

     return true

#### Q48: Time Complexity

What is the time complexity of the following pseudocode snippet?

function(n)

     if n is less than or equal to 1

          return false

     if n is less than or equal to 3

          return true

     if n is divisible by 2 or 3

           then return false

     initialise i equal to 5

     while

i

 is less than or equal to n

2

          if n is divisible by i or i+2

               then return false

          add 6 to i

     return true

(Hint: Recall the definition of a Big O notation.)

Ans: O(sqrt(n))

**✓ Correct**

**Feedback:**

For checking whether a number is prime or not, time complexity is O(sqrt(n)). Even though we are checking only for one in every six values of i <= √n, the time complexity cannot be O(sqrt(n)/6) because based on the [definition of Big O notation](https://stackoverflow.com/questions/18572522/why-o2n2-and-o100-n2-same-as-on2-in-algorithm-complexity), O(sqrt(n)/6) should be written as O(sqrt(n)). So, the time complexity is O(sqrt(n)).

**Algorithm:**

1. Create a list of consecutive integers from 2 to n.
2. Initially, let p be equal to 2, which is the smallest prime number.
3. Mark all the numbers that can be divided by p i.e 2p, 3p, 4p, etc.
4. Find the first number that is greater than p that is not marked. If there is no such number, stop. Otherwise, let p now equal this new number (which is the next prime), and repeat from step 3.
5. When the algorithm terminates, the numbers that are not marked are all the prime numbers below n.

#### Q50: Simple Approach

Write the function of the approach given above. Assume that the max limit point is the maximum value that an int can store. Return it as true if they meet at some point before Integer.MAX\_VALUE; otherwise, return it as false.

Ans: **static** **boolean** **check**(**int** p1, **int** s1, **int** p2, **int** s2) {

**int** max = Integer.MAX\_VALUE;

**while** (p1 < max && p2 < max) {

**if** (p1 == p2)

**return** **true**;

p1 += s1;

p2 += s2;

}

**return** **false**;

}

#### Q51: Efficient Algorithm

Can you think of a more efficient approach to find out if the two persons meet each other? Write the logic here.

Ans: As already mentioned,  the starting points of two persons will be different always. Check the following:

* The speeds of two persons are different as well
* Initially, if the first person is ahead of the second person, then the speed of the second person should be greater than that of the first person; otherwise, the second person cannot catch up to the first person and meet the first person ever or vice versa.
* The difference between the initial starting points is divisible by the difference of speeds of two people.

Refer to the example given below to gain a better understanding.

**Example:** 's' is the difference between the speeds of two people, and 'p' is the difference between the initial points of two people. So, after the first jump, the difference between two people will be p-s. After the second jump, the difference between them will be p-2s. So, for the two persons to meet the difference between then should be 0 after some n jumps or algorithmically, this implies that p-n\*s=0 => p=n \* s => p/s = n where n is integer. So, p must be divisible by s.

**Example:**

p1 = 6

s1 = 3

p2 = 8

s2 = 2

p2-p1 = 8-6 = 2

s1-s2 = 3-2 = 1

1 divides 2; so, two persons will meet after at a point after some jumps.

For the two people to  meet after the same number of jumps, the following conditions should be satisfied:

* The speeds of two persons must be different.
* If, initially, the first person is ahead of the second person, the speed of the second person should be greater than that of the first person; otherwise, the second person will not be able to catch up to the first person ever or vice versa.
* The difference of speeds should divide the difference of initial starting points of two people.

You learnt about the three conditions for the two people to meet after the same number of jumps. Now, try to code using the efficient approach.

#### Correctness

Q52: Why do we need to set the max limit point while solving the problem via the simple approach? Why don’t we need to set the max limit while solving the problem via an efficient approach?

Ans: Let's assume there is no max limit point while solving the problem via the simple approach.

Let’s assume that p1 and p2  are starting from different points and their speeds are equal. In this case, p1 and p2 will never meet because p2 will never catch up to p1.  In addition, if there is no max limit point, the algorithm will run for infinite times because p1 and p2 will never meet, as mentioned previously.

For example, If p1’s starting point is ahead of p2's starting point and p1 is faster than p2, they will not meet at any point, and the algorithm will run infinitely. So, there should be a max limit point while solving the problem via the first algorithm.

However, there is no need of a max limit point in the second algorithm because this algorithm will  compare the starting points and speeds of p1 and p2 and immediately determine whether they  will meet at any point.

#### Q54: Growth Rate

Arrange the following functions in the increasing order of rates of growth for a large value of n.

2n, n2, n, √n, n!, 4n, log(n!), log(n), 1, log(log(n)), log2n , n\*log(n)

Ans: Let’s consider n = 11

2n = 211 = 2048

n2 = 112 = 121

n = 11 = 11

√n = √11 = 3.316

n! = 11! = 39916800

4n = 411 = 4194304

log(n!) = log(39916800) = 7.601

log(n) = log(11) = 1.041

1=1

log(log(n)) = log(log(11)) = 0.0176

log2n = log211 = 1.0844

n\*log(n) = 11\*log(11) = 11.455

Order: log(log(n)) < 1 < log(n) < log2n < √n < log(n!) < n < n\*log(n) < n2 < 2n < 4n < n!

* If n>1010+1, then log(log(n))>1
* If n>25, then log(n!)>n

So, for extremely large values of n, the actual order will be as follows: 1 < log(log(n)) < log(n) < log2n < √n < n < log(n!) < n\*log(n) < n2 < 2n < 4n < n!

#### Q55: Time Complexity - I

What is the time complexity and space complexity of the following code snippet? (Hint: Recall the definitions of time and space complexities as discussed in the session previously.)

**int** a = **0**, b = **0**;

**for** (i = **0**; i < n; i++)

a++;

**for** (j = **0**; j < m; j++)

b++;

Ans: O(n+m), O(1) respectively

**✓ Correct**

**Feedback:**

Notice that the two loops are independent and not nested. So, we can consider them separately and then add the number of times for which each for loop instruction set is executed. The first loop runs for n times, and the second one runs for m times. If any information is provided regarding max(n,m), then the time complexity will be O(max(n,m)), but we do not know which one is greater. So, the time complexity is O(n+m). Since there is no additional space being utilised, the space complexity is constant O(1).

#### Q55: Time Complexity - II

What is the time complexity of the following code snippet? (Hint: Recall the definition of Big O notation, as discussed in this session previously.)

**int** a = **0**;

**for** (i = **0**; i < n/**2**; i++)

a++;



Ans:   
O(n)

**✓ Correct**

**Feedback:**

Notice that there is one loop that iterates for n/2 times. So, the time complexity is O(n). The fact that it only runs for n/2 times does not affect the big O time.

#### Q56: Time Complexity - I

What is the time complexity of the following code snippet?

**int** a = **0**, b = **0**;

**for** (i = **0**; i < n; i++)

**for** (j = **0**; j < n; j++)

a++;

**for** (k = **0**; k < m; k++)

b++;



Ans: O(n2)

**✓ Correct**

**Feedback:**

There are three for loops–loop(i), loop(j) and loop(k)–in the code snippet in which loop(i) and loop(j) are nested and loop(k) is independent. So, find the time complexities of the nested loop and loop(k) separately and then return the maximum of the two. loop(i) iterates for n times from i=0 to i=n-1. For every iteration of loop(i), the loop(j) iterates for n times. So, loop(j) iterates for n \* n times, i.e., n2 times. loop(k) iterates for n times. So, the time complexity is O(max(n2,n)) i.e O(n2).

#### Q57: Time Complexity - II

What is the time complexity of the following code snippet? (Hint: Evaluate the loop iterations. Then, using the definition of the Big O notation, answer the question given below.)

**int** a=**0**;

**for**(**int** i = **1**; i <= n/**3**; i++)

**for**(**int** j = **1**; j <= n; j += **4**)

a++;

Ans:   
O(n2)

**✓ Correct**

**Feedback:**

There are two for loops nested. The outer(i) loop will iterate for n/3 times from i=1 to i=n/3. For every iteration of outer(i) loop, the inner(j) loop iterates for n/4 times because in every iteration, the j value will increase by 4, and j can have a value less than or equal to n. So, the total number of times for which the j loop executed is n/4+n/4+.... (n/3 times) = (n/3)(n/4) = n2/12 times. Hence, the time complexity is O(n2).

#### Q58: Time Complexity - I

What is the time complexity of the following code snippet? (Hint: Observe the definition of the variable in the for loop carefully and then evaluate the number of iterations.)

**int** a = **0**;

**for** (i = n; i > **0**; i /=**2**)

a++;

Ans: O(log(n))

**✓ Correct**

**Feedback:**

There is one ‘for’ loop in the given code snippet. Observe how the 'i' value is changing after every iteration. After every iteration, the value of 'i' will be halved. So, after x iterations, the value of 'i' will be n/2x. However, the minimum integer value 'i' can attain is 1; so, n/2x = 1 implies x = log(n). Therefore, the loop iterates for log(n) times. So, the time complexity is O(log(n)).

#### Q59: Time Complexity - II

What is the time complexity of the following code snippet? (Hint: Observe the definition of the variable in the for loop carefully and then evaluate the number of iterations,)

**int** a = **0**

**for**(**int** i = **0**; i < n; i++)

**for**(**int** j = **1**; j < n; j \*=**2**)

a++;

Ans:   
O(n\*log(n))

**✓ Correct**

**Feedback:**

There are two for loops nested. The outer(i) loop iterates for n times. For every iteration of the outer(i) loop, the inner(j) loop executes for logn times. So, the total number of times for which the j loop executed is logn+logn+logn+...ntimes = n \* logn. So, the time complexity is O(n\*log(n)).

#### Q60: Upper Bound

What is the upper bound of n4 + 10n3 + 100n2 + 1000n?

Ans: n4+ 10n3 + 100n2 + 1000n <= 4n4 for all n>=10

n4 + 10n3 + 100n2 + 1000n = O(n4) for c = 4 and n0 = 10.

#### Q61: Time Complexity

What is the time complexity of the following code snippet?  (Hint: Observe the definition of the variable in the for loop carefully and then evaluate the number of iterations.)

**int** a = **0**;

**for** (**int** i = **1**; i <= n; i++)

**for** (**int** j = **1**; j <= n; j += i)

     a++;

Ans: O(n\*log(n))

**✓ Correct**

**Feedback:**

There are two nested for loops. The outer(i) loop will iterate for n times from i=1 to i=n. For every iteration of the outer(i) loop, the inner(j) loop iterates for n/i times because in every iteration, the j value will increase by i, and j can have a value that is less than or equal to n. So, the total number of times for which the j loop executed is n/1+n/2+n/3+.....+n/n = n(1+½+⅓+....+1/n) = n(logn) times. So, the time complexity is O(nlogn).

#### Q62: Time Complexity

What is the time complexity of the following code snippet? (Hint: What are the number of times for which the while loop is executed?)

Note: The initial value of the sum is zero, and max and b are positive integers.

**int** sum=**0**, max, b;

**while**(sum < max)

sum += b;

Ans:   
O(max/b)

**✓ Correct**

**Feedback:**

There is one while loop. This loop will end if sum >= max. At each iteration of the loop, the value of the sum increases by b. The while loop will execute for (max - initial value of sum) / b) times, i.e., (max/b) times because the initial value of sum = 0. So, the time complexity is O(max/b).

#### Q63: Recurrence Relation

#### Find the time complexity of the following recurrence: T(n) = 3T(n-1) if n>0

                                                                              = 1 otherwise

Ans: Solve the function:

T(n)=3T(n−1)  
        =3(3T(n−2))=32T(n−2) because T(n−1)=3T(n−2)  
        = 32(3T(n−3))=33T(n−3) because T(n−2)=3T(n−3)  
        = 33(3T(n−4))=34T(n−4) because T(n−3)=3T(n−4)  
.  
.  
.  
.  
On generalising this equation in terms of k, you will obtain the following:  
T(n)=3kT(n−k) for some positive integer k.  
   
For the base case, the equation will get reduced to 1. Thus, T(n−k) will become equal to 1. This implies the following:  
n–k=0  
This implies that n–k=0.  
   
On substituting the value of k=n in the generalised equation, you will obtain the following:  
T(n)=3nT(n−n)=3nT(0)=3n  
   
Therefore, the time complexity of the given function is O(3n).

Q64: Find the time complexity of the following recurrence: T(n) = 2T(n-1)-1 if n>0

                                                                              = 1 otherwise

Ans: Solve the function:  
T(n)=2T(n−1)−1  
        = 2(2T(n−2)−1)−1=22T(n−2)−2−1=22T(n−2)−3 because T(n−1)=2T(n−2)−1  
        = 22(2T(n−3)−1)−3=23T(n−3)−4−3=23T(n−3)−7 because T(n−2)=2T(n−3)−1  
        = 23(2T(n−4)−1)−7=24T(n−4)−8−7=24T(n−4)−15 because T(n−3)=2T(n−4)−1  
.  
.  
.  
.

On generalising this equation in terms of k, you will obtain the following:  
T(n)=2kT(n−k)–(2k−1) for some positive integer k.  
   
For the base case, the equation will get reduced to 1. Thus, T(n−k) will become equal to 1. This implies the following:  
n–k=0  
This implies that k=n.

On substituting the value of k=n in the generalised equation, you will obtain the following:  
T(n)=2nT(n−n)−(2n−1)  
        = 2nT(0)−2n+1  
        = 2n.1−2n+1  
        = 2n.1−2n+1  
        = 2n−2n+1  
        = 1

Therefore, the time complexity of the given function is O(1).

# Conceptual Questions - 3

**Master theorem for divide-and-conquer recurrences:**

If the recurrence relation is of the form T(n) = aT(n/b) + Θ(nklogpn) where a>=1, b>1, k>=0 and p is real number, then:

1. If a > bk, then T(n) = Θ(nlogba)
2. If a = bk, then
   1. If p>-1, then T(n) = Θ(nlogbalogp+1n)
   2. If p= -1, then T(n) = Θ(nlogbalog(log(n)))
   3. If p < -1, then T(n) = Θ(nlogba)
3. If a < bk, then
   1. If p>= 0, then T(n) = Θ(nklogpn)
   2. If p < 0, then T(n) = O(nk)

Note:

* The master theorem always yields asymptotically tight bounds to recurrences from the divide and conquer algorithms that partition an input into smaller subproblems of equal sizes, solve the subproblems recursively, and then combine the subproblem solutions to give a solution to the original problem.
* There are no separate set of equations for the master theorem for BigO notation.
* f(x) = Θ(g(x)) iff f(x) = O(g(x)) and f(x) = Ω(g(x))
* The derivation and proof of this theorem are out of scope for this module.

**Master theorem for subtract-and-conquer recurrences:**

If the recurrence relation is of the form T(n) = c, if n<=1

                                                                                           = aT(n-b)+f(n) if n>1

where c,a > 0, b>0, k >= 0,

If f(n)=O(nk) then:

                                           T(n) = O(nk),          if a<1

                                                     = O(nk+1),      if a=1

                                                     = O(nkan/b),   if a>1

#### Q65: Time Complexity

What is the time complexity of the following code snippet? (Hint: Answer using the master theorems that were discussed in this segment.)

fun(**int** n) {

**if** (n <= **0**)

**return** **0**;

**else** **if** (n == **1**)

**return** **1**;

**return** fun(n - **1**) + fun(n - **1**);

}

Ans: O(2n)

**✓ Correct**

**Feedback:**

The recurrence formula is as follows:

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

According the master theorem for subtract-and-conquer recurrences:

a=2, b=1, k=0

Therefore, T(n) = O(n02n/1) i.n/1(2n).

So, the time complexity is O(2n).

#### Q68: Time Complexity

What is the time complexity of the following code snippet? (Hint: Answer using the master theorems discussed in this segment.)

fib(**int** n) {

**if** (n <= **0**)

**return** **0**;

**else** **if** (n == **1**)

**return** **1**;

**return** fib(n - **1**) + fib(n - **2**);

}



Ans:   
O(2n)

**✓ Correct**

**Feedback:**

The recurrence formula is as follows:

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

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

According to the master theorem for subtract-and-conquer recurrences:

a=2, b=1, k=0

Therefore, T(n) = O(n02n/1), giving O(2n).

So, the time complexity is O(2n).

#### Q69: Time Complexity

What is the time complexity of the code snippet given below?

fun(**int** n) {

**int** a = **0**, b = **1**, c = **0**;

**for** (**int** i = **2**; i <= n; i++) {

c = a + b;

a = b;

b = c;

}

**return** c;

}

Ans: O(n)

**✓ Correct**

**Feedback:**

There is one for loop that is iterated n-2 times for n>2. So, the time complexity is O(n-2), giving O(n).

#### Q70: Time Complexity

What is the time complexity of the code snippet given below? (Hint: Answer using the master theorems discussed in this segment.)

fun(**int** n) {

**if** (n < **1**)

**return** **0**;

**else** **if** (n == **1**)

**return** **1**;

**return** **2** \* fun(n / **2**);

}

Ans:   
θ(log(n))

**✓ Correct**

**Feedback:**

If you observe carefully, the value of n is halved after each call. It will continue to call the recurrence function until the value reaches 1. Suppose x recurrence calls were made for the value of n to become 1. This gives n/2x = 1, implying x= logn. So, the time complexity is θ(logn).

#### Q71: Time Complexity

Find the recursive function for the running time T(n).

fun(**int** n) {

**int** a = **0**;

**if** (n == **1**)

**return**;

**for** (**int** i = **1**; i <= n; i++)

**for** (**int** j = **1**; j <= n; j++)

       a++;

   fun(n - **3**);

}

Ans: If fun(n) is called, the given two loops are executed, and then, fun(n-3) is called. The outer loop and the inner loop iterate for n times each. So, the two loops execute for cn2 times. Thus, T(n) = cn2+T(n-3).

#### Q72: Time Complexity

Solve the recurrence relation T(n) = 2T(√n)+logn and find the time complexity.

Ans: Assume n = 2m.

Apply log on both sides

=> logn = m\*log2

=> m = logn

Now, the recurrence relation is converted as follows:

T(2m) = 2T(√2m)+m = 2T(2m/2)+m

Assume T(2m) = S(m)

=> T(2m/2) = S(m/2) and S(m) = 2S(m/2)+m

According to the master theorem for divide-and-conquer recurrences:

p=0, a=2, b=2, k=1 and a=bk and p>-1 so,

T(n)=Θ(nlogbalogp+1n) => S(m) = Θ(mlogbalogp+1m) => Θ(mlog22log0+1m)

=> Θ(mlogm)

Substitute m = logn

=> T(n) = O(logn\*loglogn).

So, the time complexity is O(logn\*loglogn).

#### Q68: Time Complexity

What is the time complexity of the code snippet given below? (Hint: Answer using the master theorems discussed in this segment.)

fun(**int** n) {

**if** (n <= **1**)

**return**;

**for** (**int** i = **1**; i <= **3**; i++)

    fun(n - **1**);

}

Ans: O(3n)

**✓ Correct**

**Feedback:**

If you call fun(n), then for loop will execute three times, and each time it then calls fun(n-1). Thus, T(n) = c + 3T(n-1).

According to the master theorem for subtract-and-conquer recurrences:

a=3, b=1, k=0

Therefore, T(n) = O(n03n/1) i.e (3n).

So, the time complexity is O(3n).

# Sum of n-terms of a GP (to be deleted)

GP stands for Geometric Progression. A GP is a sequence of numbers where each new term is found by multiplying the previous term by a fixed number (called the common ratio, denoted by 'r'). The first term of a GP is denoted by 'a'.

For example, the GP for a = 3, r = 2 looks like this:  
3, 6, 12, 24, 48...

You need to find the sum of the first n terms of a GP and return the result mod m (result % m).

**Approach**

Here, you need to find the sum of the following series.  
a+ar+ar2+ar3 +....+ arn

Taking ‘a’ common, you get:

=a(1+r+r2+r3 +....+ rn)

Let sum(r,n)=1+r+r2+r3 +....+ rn

Observe that,  
[1+r+r2+r3 +...+ r2∗n+1]=(1+r)∗(1+(r2)+(r2)2+(r2)3+...+(r2)n)

So,

sum(r,2∗n+1)=(1+r)∗sum(r2,n)

Also,

sum(r,2∗n)=1+(r∗(1+r)∗sum(r2,n−1))

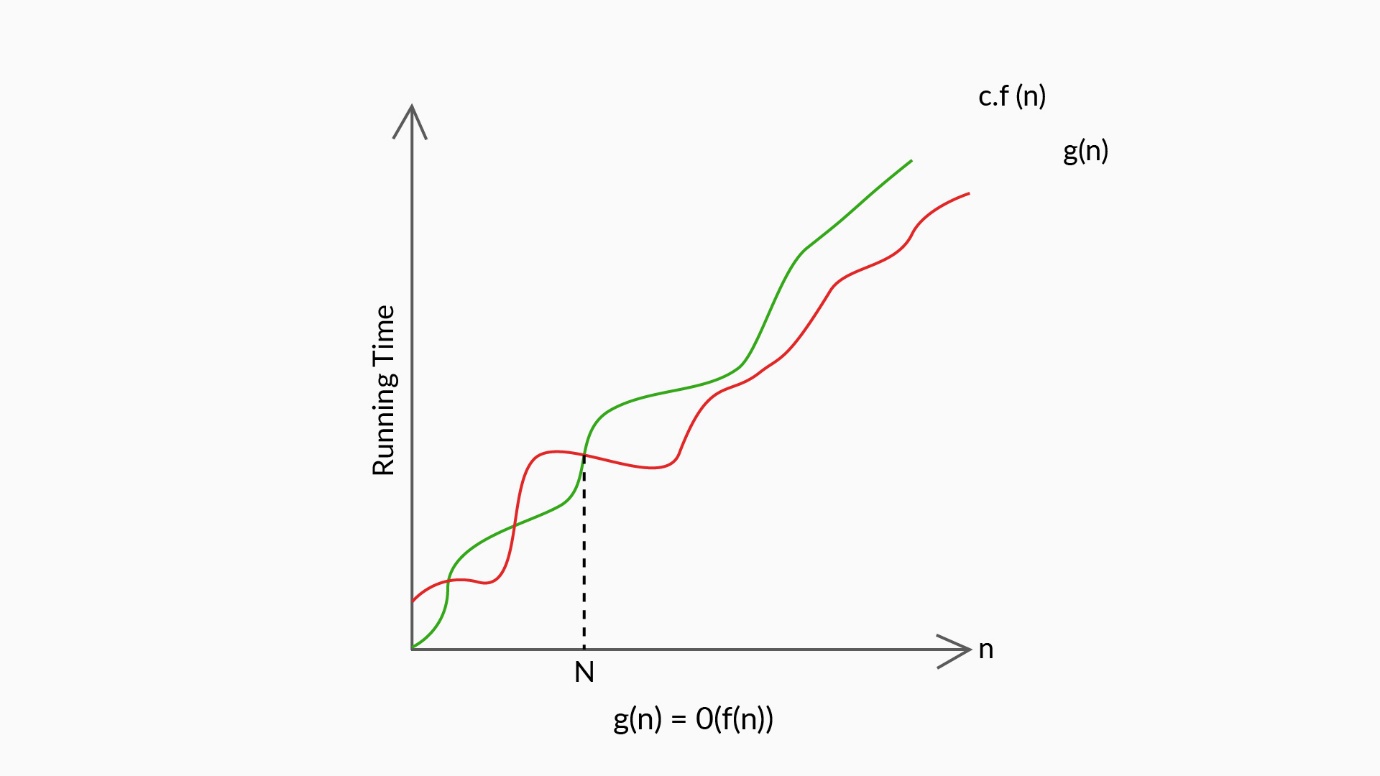
So, you have got your recurrence relations.

The base cases will be:  
sum(r,0)=1  
sum(r,1)=1+r

# More on Asymptotic Notations

**Big**O

Big O refers to the order of growth of a function; essentially, the growth of any complexity function for large input size(n) values depends on the most significant term of the function. So, less significant terms are dropped from the function to calculate the Big O.



Big O

**Definition:** Big O (O) is 'bounded above by' (upper bound); there exist constants c>0 and

N>0.  
                    T(n)≤c.f(n), for all n>N, T(n)**∈ O(**f(n)**)**

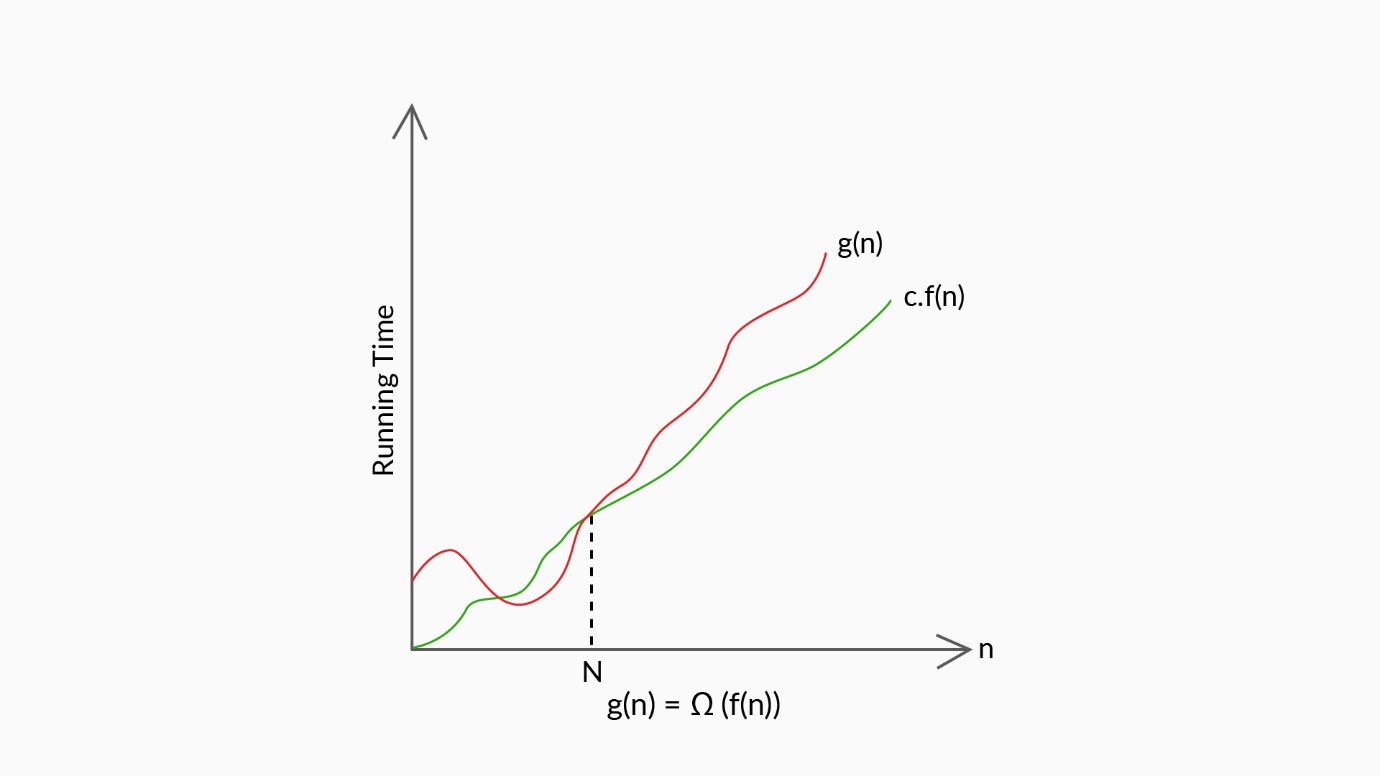
As shown in the graph given above, only for n>N ,***g(n)*** is bounded by c.f(n). It can be clearly seen from the graph that for all n<N, ***g(n)***  is not bounded  by c.f(n).  Hence the upper bound of ***g(n)*** is ***O(f(n))***. Which means that for any value of n>N, the running time of an algorithm does not cross time provided by ***O(f(n))***.

Now that you have understood the upper bound (Big O) of an algorithm, let’s take a look at the following two other mathematical notations: Big Omega and Big Theta.

**Big Omega**

Big Omega indicates the lower bound of the running time of an algorithm. In the earlier scenario of opening a lock, if you were able to undo the lock on your first attempt itself, then the first attempt will be the lower bound. In any other case, the number of attempts to unlock will be **greater than** the lower bound.

When calculating Big Omega, the time complexity function of an algorithm is **greater than or equal** to the lower bound.



Omega

**Definition:** Big Omega (Ω) is 'bounded below by' (lower bound). There exists constants c>0 and N>0.  
                    c.f(n)≤T(n), for all n>N, T(n)**∈ Ω(**f(n)**)**

As shown in the graph given above, for  all n>N, **g(n)**is bounded below by c.f(n). It can be clearly seen from the graph that for all n<N, **g(n)**is not bounded below by c.f(n). Hence the lower bound of ***g(n)*** is **Ω**(f(n)). Which means that for any value of n>N, the minimum time required by the algorithm is given  by **Ω**(f(n)).

Suppose an algorithm takes the time complexity function as follows:

T(n)=2n2−3n+6, for all n>N

          ⩾2n2−3n

          ⩾n2, forn⩾3because 2n2−3n⩾n2

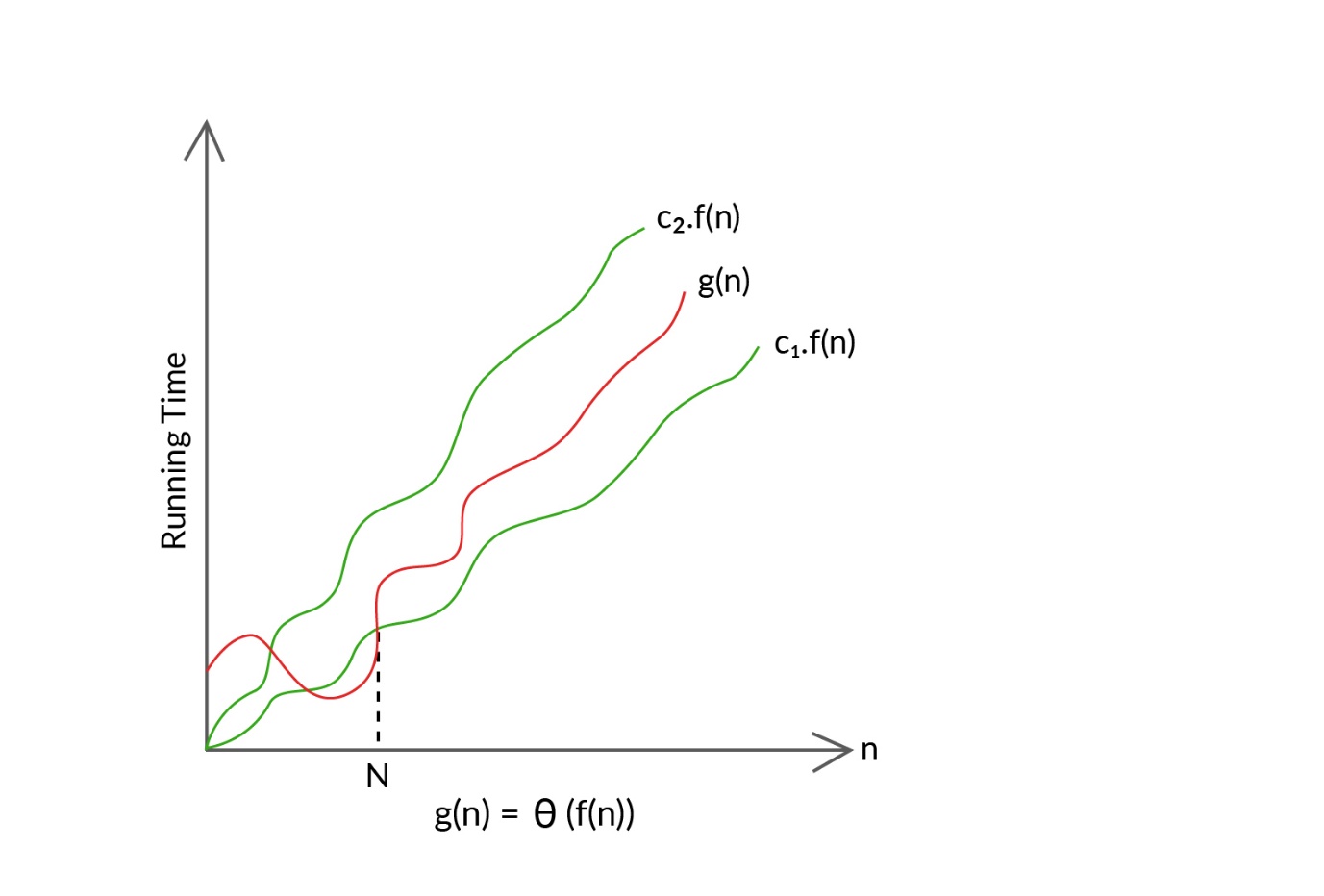
                                                                     n2⩾3n

                                                                      n⩾3

So, n2≤T(n), for all n⩾3             
T(n)**∈ Ω(**n2**)**

**Big Theta**

Big Theta represents an in-between case, bounded both above (upper bound) and below (lower bound) the running time of an algorithm.



Theta

**Definition:** Big Theta (θ) is 'bounded above and below'. There exists constants c1>0,c2>0, and N>0.

                    c1.f(n)≤T(n)≤c2.f(n), for all n>N, T(n)**∈** θ(f(n))

As shown in the above graph, for n>N then  g(n)is bounded above and below by c2.f(n) and c1.f(n). It can be clearly seen from the graph that for all n<N,g(n) is not bounded above and below by c2.f(n) and c1.f(n).

Hence, the average case running time of g(n) lies between c1.f(n) and c2.f(n) which is given by θ(f(n)) .

Suppose an algorithm takes the time complexity function as T(n)=2n2−3n+6 , as already discussed.

Upper bound:  2n2−3n+6≤2n2

Lower bound:  n2≤2n2−3n+6

then, n2≤2n2−3n+6≤2n2

n2≤T(n)≤2n2, for all n≥3

 T(n)**∈**θ(n2)

From the three asymptotic notations that we discussed, the most used notation to compare algorithms is Big O. So, we tend to find the worst case of an algorithm with respect to the input size (n).