Master Theorem

$$T(n) = aT(n/b) + \theta(n^k \log^p n)$$

where n = size of the problem

a = number of subproblems in the recursion and a >= 1

n/b = size of each subproblem

b > 1, k >= 0 and p is a real number.

Then,

1. If a > , then
2. If , then
    1. If p > -1, then
    2. If p = -1, then
    3. If p < -1, then
3. if , then
    1. if p >= 0, then
    2. if p < 0, then

Q - Solve for the following recursion

▼ Solution

Some popular complexities

O(1)

Constant time complexity. Example: a + b, a % b, swap(a, b)

O()

Logarithmic complexity. Example: Binary search

O()

Example: Finding divisors of a number

O(n)

Linear time complexity. Example: Linear search, finding maximum element

O()

Example: Sorting, Seive of Eratosthenes

O()

Quadratic complexity. Example: Insertion sort

O()

Cubic complexity. Example: Floyd Warshall algorithm

O()

Exponential complexity. Example: bitmasking

Problems

Q: Analyse the complexity for the following code

```
for(let i=0; i<k; i++) {
    for(let j=0; j<k; j++) {
        console.log(i + " " + j);
    }
}
```

Copy

▼ Solution

First, we need to generate a function f(n) that returns the time taken to execute for any input n.

For a given n, the outer loop of i runs for 0 to n-1.

For each value of i, the inner loop runs again for 0 to n-1

So for each i, the inner loop runs n times.

Total complexity = n + n + n + ... + n (n times as i runs for n times)

f(n) = n ^ 2

So T.C = O(k^2)

Q: Analyse the complexity for the following code

```
for(let i=0; i<k; i++) {
    for(let j=0; j<=i; j++) {
        console.log(i + " " + j);
    }
}
```

Copy

▼ Solution

For a given i, j runs from 0 to i

For i = 0: j only runs 1 time

For i = 1: j runs 2 times

For i = 2: j runs 3 times

For i = 3: j runs 4 times

..

For any general i = k: j runs for 0, 1, 2, ..., k so k times

So total complexity

1 + 2 + 3 + ... + n =

Complexity: O(k^2)

Q:

```
for(let i=1; i<n; i *= 2) {
    console.log(i);
}
```

Copy

▼ Solution

Let's see how i runs

i = 1, 2, 4, 8, 16, ..., x

where x is the highest power of 2 less than n. Let's say x = , then

So how many steps?

Step 0: i = 2 ^ 0

Step 1: i = 2 ^ 1

Step 2: i = 2 ^ 2

Step 3: i = 2 ^ 3

..

Step p: i = 2 ^ p = x

But what is p?

or p ≤

Complexity: O(

Q:

```
for(let i=1; i<n; i++) {
  for(let j=1; j<n; j+=i) {
     console.log(i + " " + j);
  }
}
```

Copy

▼ Solution

For i = 1, j runs n times (as it is same as j += 1)

For i = 2, j runs n/2 times (as j += 2)

For i = 3, j runs n/3 times

For general i = k, j runs n / k times

n + n/2 + n/3 + n/4 + ... + n/k +.. + 1

n (1 + 1/2 + 1/3 + ... 1/n)

Now we need to find the complexity of:

What is Big - O? We need to find g(n) which is greater than f(n)

We round each number k, down to its nearest power of 2 less than equal to k

i.e 3 can be rounded down to 2, 4 can be rounded down to 4, 5 can be rounded down to 4

Obviously, k > 2 ^ p, where 2 ^ p is the nearest power of 2 less than equal to k

Or,

So rewriting:

Rewriting RHS:

where

So total complexity: O(n *)

Overall Plan

Day 1: Complexity Analysis

Day 2: Arrays Continued - Sliding Window + Problems

Day 3: Recursion

Day 4: Binary Search - Part 1

Day 5: Binary Search - Part 2

Day 6: Linked Lists

Day 7: Stacks

Day 8: Queues

Day 9: Heaps

Day 10: Maps + Sets

Day 11: Doubt Clearing + Problem Solving

Day 12: Trees - 1 (Binary Trees - Beginner problems)

Day 13: Trees - 2 (Binary Trees - Intermediate to advanced problems)

Day 14: Trees - 3 (N-ary trees)

Day 15: DP - 1 (Beginner problems)

Day 16: DP - 2 (Intermediate problems)

Day 17: DP - 3 (Intermediate problems + Advanced problems + Optimisations)

Day 18: Doubt Clearing + Problem Solving

Day 19: Disjoint Set Union

Day 20: Tries

Day 21: Graphs - 1 (BFS + DFS + Problems)

Day 22: Graphs - 2 (MST + Dijkstra + FW + Problems)

Day 23: Graphs - 3 (Directed Graphs + Toplogical Sorting + SCC + Problems)

Day 24: Doubt Clearing + Problem Solving

Day 26: Articulation Points + Bridges in Graphs

Day 28: Advanced problems in Graphs (Graph DP + general advanced problems)

Day 29 - 32: Other advances concepts (Tree dp, Range queries DS, Dp optimisations) + Problem solving + Doubt clearing (Buffer period)

> 💡 We might not cover advanced concepts in week 29-32. It is dependent on how the batch is performing on an average and how much we are able to achieve our plan. It is more like a buffer period. As we have an aggressive timeline, we will move a bit fast. But in some places, we need to slow down as most students otherwise will get confused. I have designed the course aggressively but we need to move with the pace of the class.

> 💡 If we decide to proceed with C++, we will need additional 2 classes so that everyone becomes familiar with the syntax and stuff. We can cover STL whenever we are discussing the corresponding topics - for example, we can discuss priority_queues when discussing around heaps.

Complexity Analysis

Why?

Generally there can be multiple ways to solve a problem. For example, in our last class we learn't about finding intersections for multiple intervals / ranges. But we need a way to compare which one is better than the other. We can do this comparison using Complexity Analysis.

Generally, we compare 2 things:

1. Time Complexity
2. Space Complexity

Time Complexity

Time complexity of an algorithm quantifies the amount of time taken by an algorithm to run as a function of the length of the input

```
t(n) = n^2 + 2n + 6
```
Copy

Here t(n), is a function that returns the time taken by the algorithm for an input of length n.

💡 For simplicity, we have used a single parameter. It can be a function of multiple parameters as well.

Space Complexity

Space complexity of an algorithm quantifies the amount of space or memory taken by an algorithm to run as a function of the length of the input.

```
t(n) = n^2
```
Copy

Here t(n), is a function that returns the additional space used by the algorithm for an input of length n.

Note: Generally by space complexity, we mean additional space. The input can be an array of length n, but generally we do not consider that while computing space complexity

Asymptotic Notations

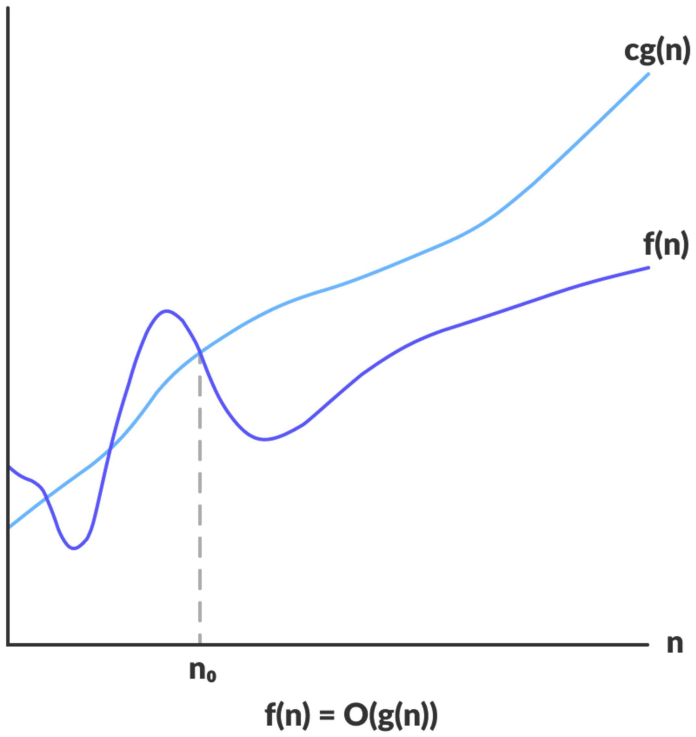
Asymptotic notations are the mathematical notations used to describe the running time of an algorithm when the input tends towards a particular value or a limiting value.

There are mainly three asymptotic notations:

- Big-O notation
- Omega notation
- Theta notation

Big-O Notation

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



We say, a function f(n) is O(g(n)) if there exists c and > 0 such that

Examples:

f(n) = n^2 + 5n - 6

g(n) = 2n ^ 2

So g(n) ≥ f(n) n ≥ 3 (Solved using quadratic equation)

So there exists, c = 2 and = 3 such that f(n) ≤ c * g(n) for all n ≥

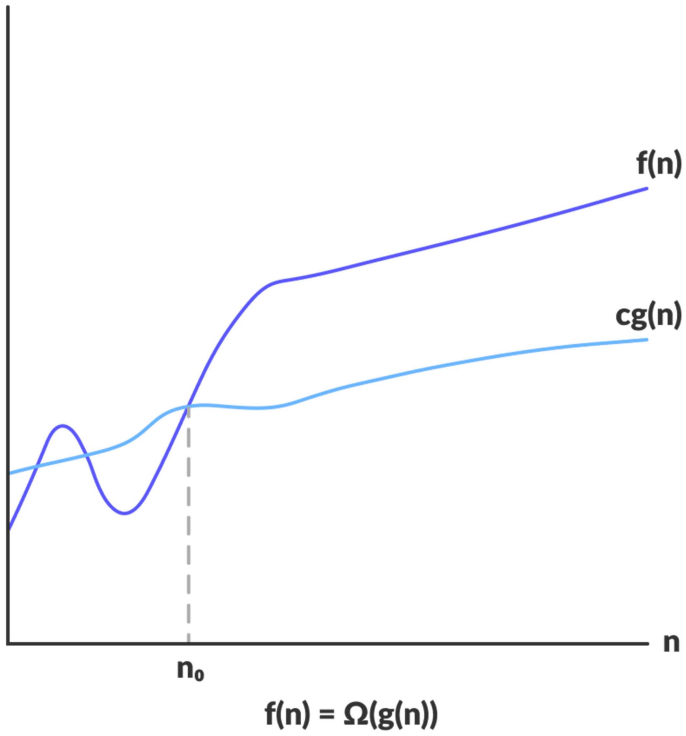
So f(n) = O(g(n))

Omega Notation

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

```
Ω(g(n)) = { f(n): there exist positive constants c and n0
            such that 0 ≤ cg(n) ≤ f(n) for all n ≥ n0 }
```

Copy



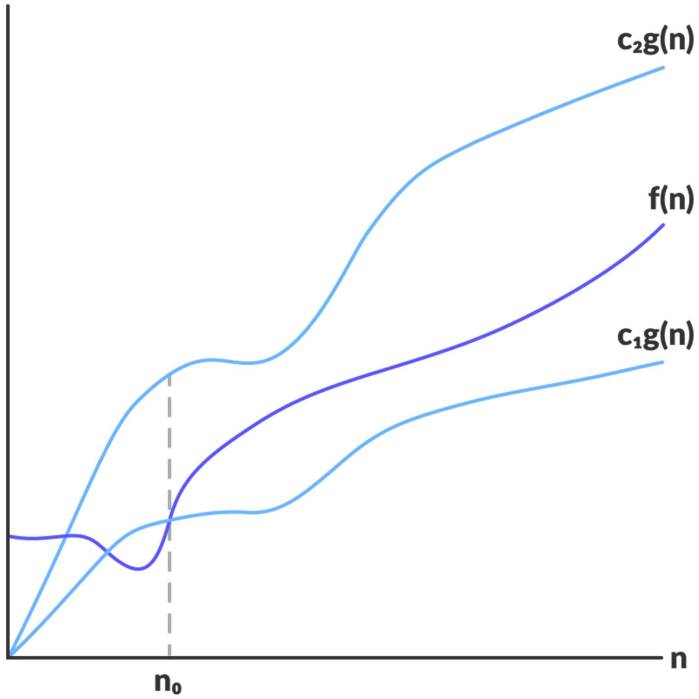
For example, f(n) = n^2 and g(n) = n + 10

Theta Notation

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

```
Θ(g(n)) = { f(n): there exist positive constants c1, c2 and n0
            such that 0 ≤ c1g(n) ≤ f(n) ≤ c2g(n) for all n ≥ n0 }
```

Copy



f(n) = Θ(g(n))