Algorithm Engineering

Lab 1 – Lecture

In this lab, you will learn-

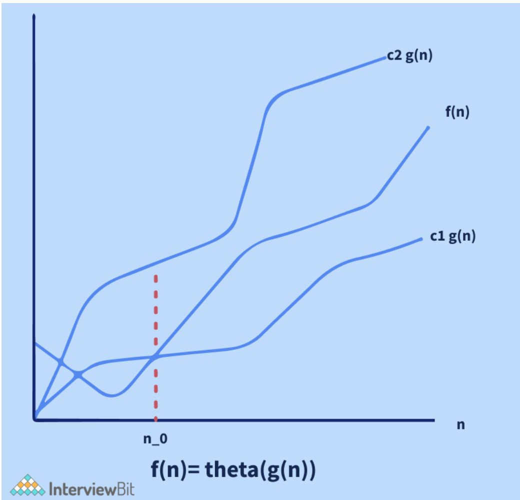
1. **design** algorithms
   a. come up with solutions
   b. generate test inputs
   c. debug
2. **analyze** algorithms
   a. time complexity
   b. space complexity
   c. compare with other approaches/algorithms
3. **optimize** algorithms
   a. make it faster
   b. make it use less space

Asymptotic Notations

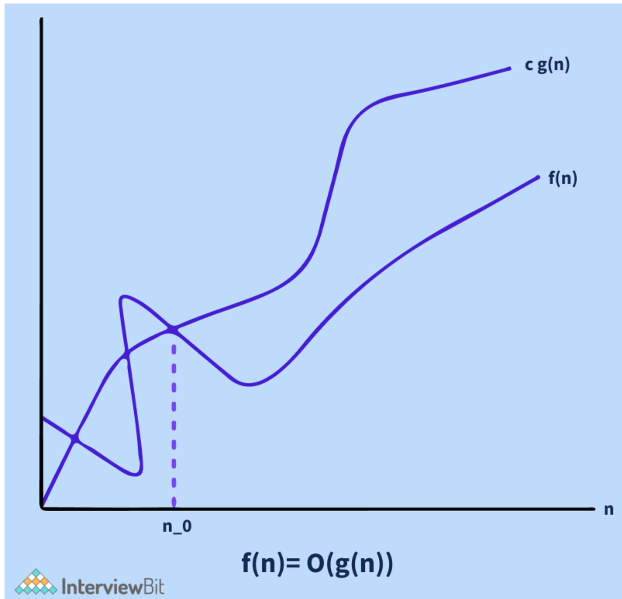
Asymptotic analysis is a technique that is used for determining the efficiency of an algorithm that does not rely on machine-specific constants and avoids the algorithm from comparing itself to the time-consuming approach. For asymptotic analysis, asymptotic notation is a mathematical technique that is used to indicate the temporal complexity of algorithms.

The following are the three most common asymptotic notations.

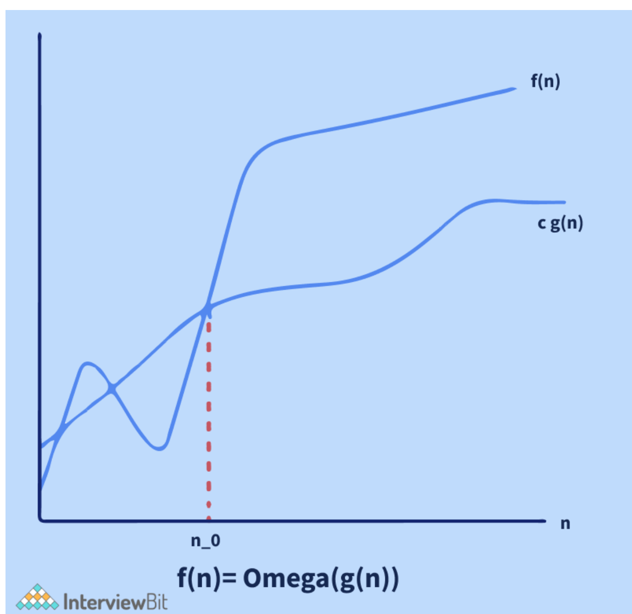
- **Big Theta Notation: (θ Notation)**
  The exact asymptotic behavior is defined using the theta (θ) Notation. It binds functions from above and below to define behavior. Dropping low order terms and ignoring leading constants is a convenient approach to get Theta notation for an expression.



- **Big O Notation:**
The Big O notation defines an upper bound for an algorithm by bounding a function from above. Consider the situation of insertion sort: in the best-case scenario, it takes linear time, and in the worst case, it takes quadratic time. Insertion sort has a time complexity O(n^2). It is useful when we just have an upper constraint on an algorithm's time complexity.



- **Big Omega (Ω) Notation:**
The Ω Notation provides an asymptotic lower bound on a function, just like Big O notation does. It is useful when we have a lower bound on an algorithm's time complexity.



Finding Time Complexity of Programs with Loops

O(n)

```
for i in range(1,N):
    print(i)
for i in range(1,N):
    print(i)
```

O(n^2)

```
for i in range(1,N):
    print(i)
    for j in range(1,N):
        print(j)
```

O(n) [as the inner loop runs finite times; the complexity is O(2n) to be exact]

```
for i in range(1,N):
    print(i)
    for j in range(1,N):
        if j>=2:
            break
        print(j)
```

Have to use summation to find out the exact complexity for the following snippet

```
for i in range(1,N):
    print(i)
    for j in range(1,i):
        print(j)
```

```
i = 1, j = 1
i = 2, j = 1, 2
i = 3, j = 1, 2, 3
...
i = N, j = 1, 2,... N
```

So, the total number of operations = 1 + 2 + 3 + … N = O(N*(N+1)/2) = **O(N^2)**

Finding Time Complexity of Programs with Recursion

Code Snippet:

```
int f(int n){
    if(n<=1)
        return 1;
    return f(n-1) + f(n-1);
}
```

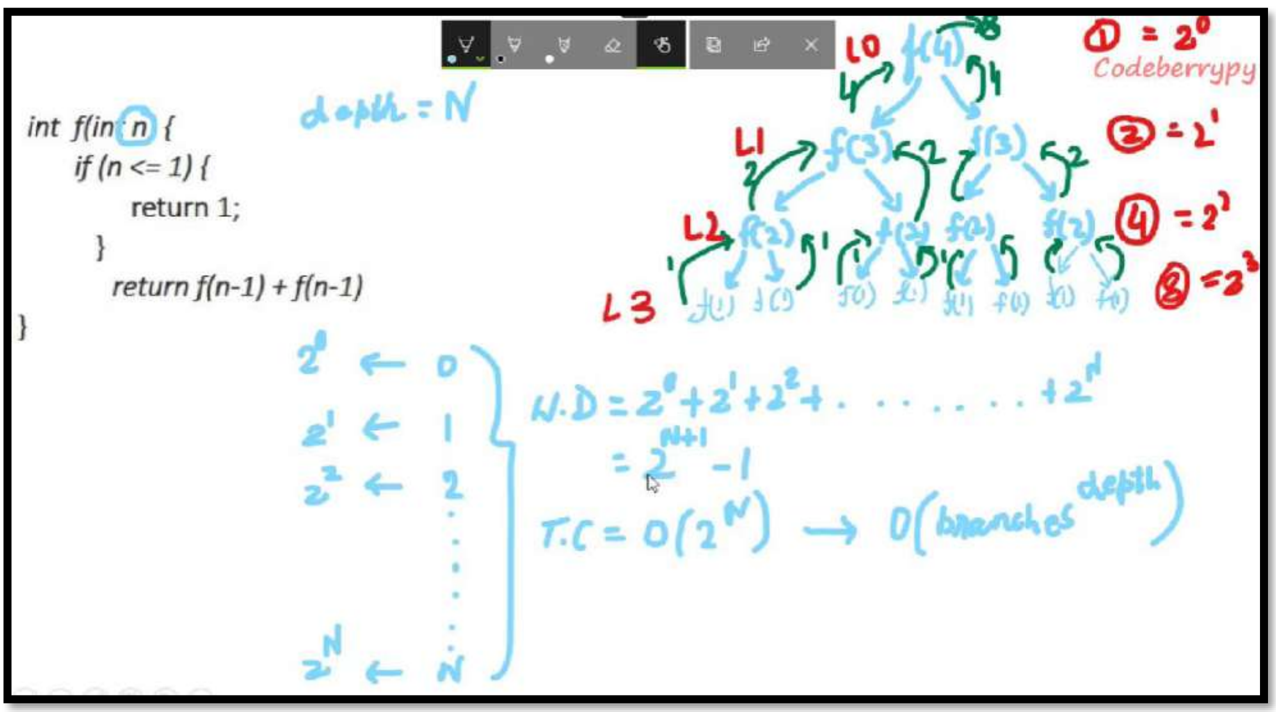
Complexity Formulation:

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

For this particular problem, the time complexity of a subproblem with size ***n*** is the 1 plus twice the time complexity of subproblem of size ***n-1.***

Solution:

Draw the recursion. Count the number of operations in each node. Add them up.



Answer: O(2^N)

Reference:

https://youtu.be/NyV0d5QadWM

https://youtu.be/ncpTxqK35PI

Tasks

Solve the attached tasks in **Lab1_Tasks.pdf** and write the solution in a report using Latex/Overleaf. Upload one pdf containing the report. Rename the submitted pdf as **LabNo_LabGroup_FullStudentID.pdf** (Lab1_1A_160041010.pdf)

Use the following template for generating report; copy the following overleaf project and edit it to prepare the report.

https://www.overleaf.com/read/kdpyrcyptkfx