Q1 What do you understand by Asymptotic notations. Define different Asymptotic notation with examples.

Asymptotic notations are a set of mathematical tools used to describe the behavior of functions as their input sizes approach infinity. They are often used to analyze the time and space complexity of algorithms.

There are three main types of asymptotic notation:

Big O notation (O): This notation provides an upper bound on the growth rate of a function. It represents the worst-case running time of an algorithm, which is the maximum amount of time it could take to complete. For example, if we say that an algorithm has a time complexity of O(n), we mean that the algorithm's running time grows at most linearly with the size of its input.
Example: consider the following code that sums the first n natural numbers:

```
int sum = 0;
for (int i = 1; i <= n; i++) {
sum += i;
}
```

The time complexity of this algorithm is O(n), since the running time grows linearly with the input size.

Q1

Omega notation (Ω): This notation provides a lower bound on the growth rate of a function. It represents the best-case running time of an algorithm, which is the minimum amount of time it could take to complete. For example, if we say that an algorithm has a time complexity of $\Omega(n)$, we mean that the algorithm's running time grows at least linearly with the size of its input.

Example: consider the following code that finds the minimum element in an array:

```
int min = array[0];
for (int i = 1; i < n; i++) {
if (array[i] < min) {
min = array[i];
    }
}
```

The time complexity of this algorithm is $\Omega(n)$, since we need to look at each element in the array at least once to find the minimum.

Theta notation (Θ): This notation provides both an upper and a lower bound on the growth rate of a function. It represents the average-case running time of an algorithm, which is the expected amount of time it would take to complete. For example, if we say that an algorithm has a time complexity of $\Theta(n)$, we mean that the algorithm's running time grows linearly with the size of its

Q1

input, and there are no faster or slower growth rates.

Example: consider the following code that reverses a string:

```
char[] reversed = new char[s.length()];
for (int i = 0; i < s.length(); i++) {
reversed[i] = s.charAt(s.length() - i - 1);
}
String result = new String(reversed);
```

The time complexity of this algorithm is $\Theta(n)$, since we need to visit each character in the string once to reverse it.

Asymptotic notation allows us to compare the performance of different algorithms and make informed decisions about which one to use for a particular problem.

Q2 What should be time complexity of - for(i=1 to n){ i=i*2;}

The time complexity of the loop for(i=1 to n){ i=i*2;} can be determined by counting the number of iterations that the loop will execute as a function of the input size n.

Here, the value of i is being doubled in each iteration. So, the loop will terminate when i becomes greater than n. We can express this mathematically as 2^k = n, where k is the number of iterations required for i to become greater than n. Solving for k gives us k = log(n).

Therefore, the loop will execute for log(n) iterations. Since each iteration takes constant time, the time complexity of the loop is O(log n).

Q3 What should be time complexity of - $T(n) = \{3T(n-1)$ if $n>0$, otherwise $1\}$

The time complexity of the recursive function $T(n) = \{3T(n-1)$ if $n>0$, otherwise $1\}$ can be determined by analyzing the number of function calls it makes as a function of the input size n.

Each call to T(n) results in 3 calls to T(n-1), until n reaches 0, at which point the function returns 1. We can represent this recurrence relation using a tree:

```
              T(n)
              /|\
    T(n-1) T(n-1) T(n-1)
      /|\    /|\    /|\
T(n-2) T(n-2) T(n-2) T(n-2) T(n-2) T(n-2) T(n-2) T(n-2) ...
  /|\    /|\    /|\    /|\
```

The height of the tree is n, and at each level, there are 3 nodes. Therefore, the total number of function calls is 3^n.

However, each call to the function takes constant time, so the time complexity of the function is O(3^n), which is an exponential time complexity. This means that the function's running time grows very quickly with the size of the input, making it very inefficient for large

Q4 What should be time complexity of - $T(n) = \{2T(n-1)-1$ if $n>0$, otherwise $1\}$

The time complexity of the recursive function $T(n) = \{2T(n-1)-1$ if $n>0$, otherwise $1\}$ can be determined by analyzing the number of function calls it makes as a function of the input size n.

Each call to T(n) results in 2 calls to T(n-1), until n reaches 0, at which point the function returns 1. We can represent this recurrence relation using a tree:

```
T(n)
  \     /
T(n-1) T(n-1)
  \  /X
T(n-2) T(n-2) T(n-2) T(n-2)
  X   X   X
.
```

The height of the tree is n, and at each level, there are 2 nodes. Therefore, the total number of function calls is 2^n.

However, each call to the function takes constant time (specifically, O(1) time), so the time complexity of the function is O(2^n) * O(1) = O(2^n), which is an exponential time complexity. This means that the function's running time grows very quickly with the size of the input, making it very inefficient for large inputs.

Q5 What should be time complexity of -

```
int i=1, s=1;
while(s<=n){
i++;
s=s+i;
printf("#");
}
```

The time complexity of the given while loop is O(sqrt(n)).

The loop iterates until the value of s becomes greater than n. At each iteration, i is incremented by 1, and s is updated to s+i. The number of iterations required to reach s>n can be approximated by solving the equation s + (i+1) > n, which simplifies to i^2 + i - 2(n-s) > 0.

using the quadratic formula, we can solve for the positive root of this equation to get i > (-1 + sqrt(1 + 8(n-s))) / 2. This means that the loop will iterate for O(sqrt(n)) iterations, since s starts at 1 and i starts at 2.

Since each iteration of the loop takes constant time (specifically, O(1) time), the overall time complexity of the loop is O(sqrt(n)).

Q6

Time complexity of -

```
void function(int n){
int i, count=0;
for(i=1; i*i<=n; i++)
count++
}
```

The time complexity of the given function is O(sqrt(n)).

The for loop iterates from i=1 to i such that i*i<=n. This means that the loop will execute for all values of i from 1 to the largest integer less than or equal to the square root of n. Since the loop iterates sqrt(n) times, the time complexity of the loop is O(sqrt(n)).

Each iteration of the loop takes constant time (specifically, O(1) time), since it only involves simple arithmetic operations and comparisons. Therefore, the overall time complexity of the function is O(sqrt(n)).

Q7

Time complexity of -

```
void function(int n){
int i, j, k, count=0;
for(i=n/2; i<=n; i++)
for(j=1; j<=n; j=j*2)
for(k=1; k<=n; k=k*2)
count++
}
```

The time complexity of the given function is O(n^2 * log(n)).

The function consists of three nested loops that iterate over the variables i, j, and k. The first loop iterates from i=n/2 to i<=n, which takes n/2 iterations. The second loop iterates over the variable j from 1 to n in powers of 2, which takes log2(n) iterations. The third loop iterates over the variable k from 1 to n in powers of 2, which also takes log2(n) iterations.

The total number of iterations of the loop can be calculated by multiplying the number of iterations of each loop together:

n/2 * log2(n) * log2(n)

= (1/2 * n) * (log2(n))^2

= O(n * log^2(n))

Each iteration of the loop takes constant time (specifically, O(1) time), since it only involves simple arithmetic operations and

Q7

comparisons. Therefore, the overall time complexity of the function is O(n^2 * log(n)).

Q8

Time complexity of -

```
function(int n){
if(n==1) return;
for(i=1 to n){
for(j=1 to n){
printf("*");
}
}
function(n-3);
}
```

The time complexity of the given function is O(n^2 * (n/3)^2).

The function is a recursive function that is called with the argument n-3. The function also contains two nested loops that iterate over the variables i and j. The outer loop iterates n times, and the inner loop also iterates n times. The print statement within the loops will execute n*n times.

At each recursive call, the value of n is decreased by 3. Therefore, the function will be called a total of n/3 times recursively until n=1.

The total number of times the innermost loop will be executed can be calculated by multiplying the number of iterations of the outer

Q8

and inner loops together, and then multiplying that by the number of recursive calls:

$n * n * (n/3) + (n/3)^2 * n/3 * (n/3)$

$= n^2 * (n/3) + (n/3)^4$

$= O(n^2 * (n/3)^2)$

Since each iteration of the loops and the recursive function call takes constant time (specifically, $O(1)$ time), the overall time complexity of the function is $O(n^2 * (n/3)^2)$.

Q9

Time complexity of -

```
void function(int n){
for(i=1 to n){
for(j=1; j<=n; j=j+i)
printf("*")
}
}
```

The time complexity of the given function is O(n * log(n)).

The function consists of two nested loops that iterate over the variables i and j. The outer loop iterates n times, and the inner loop iterates n/i times, where i is the current value of the outer loop variable. Therefore, the total number of iterations of the inner loop can be calculated as follows:

n + n/2 + n/3 + ... + 1

This is a harmonic series, which can be approximated as log(n) + 0.5772 + O(1/n). Therefore, the total number of iterations of the inner loop is O(n * log(n)).

Since each iteration of the inner loop takes constant time (specifically, O(1) time), the overall time complexity of the function is O(n * log(n)).