Big O Notation for Coding Interviews

Constant Time O(1)

- Input size has no impact on algorithm growth
- The total number of operations is always going to remain the same

Code Implementation

```
def get_first(nums):
    return nums[0]      # Always 1 operation
```

Logarithmic Time O(log n)

- Algorithm grows proportionally to logarithm of input size

Code Implementation

```
def binary_search(arr, target):
    low, high = 0, len(arr) - 1
    while low <= high:
        mid = (low + high) // 2
        if arr[mid] == target:
            return mid
        elif arr[mid] < target:
            low = mid + 1
        else:
            high = mid - 1
    return -1
```

Linear Time O(n)

- Algorithm grows linearly to the size of the input
- Linear time does not always mean the slope is always 1

Code Implementation

```
def find_max(nums):
    max_val = nums[0]
    for n in nums:
        if n > max_val:
            max_val = n
    return max_val
```

LogLinear Time O(n log n)

- Algorithm growth combines log n and linear time

Code Implementation

```
def merge_sort(arr):
    if len(arr) <= 1:
        return arr

    mid = len(arr) // 2
    left = merge_sort(arr[:mid])
    right = merge_sort(arr[mid:])

    result = []
    i = j = 0
    while i < len(left) and j < len(right):
        if left[i] < right[j]:
            result.append(left[i]); i += 1
        else:
            result.append(right[j]); j += 1

    result.extend(left[i:])
    result.extend(right[j:])
    return result
```

O(n * m) Complexity

- Algorithm grows proportionally to n and m's product

Code Implementation

```
def multiply_lists(a, b):
    for x in a:
        for y in b:
            print(x, y)
```

Quadratic Time O(n^2)

- Algorithm grows quadratically to the size of the input

Code Implementation

```
def print_pairs(nums):
    for i in nums:
        for j in nums:
            print(i, j)
```

Exponential Complexity O(2^n)

- Algorithm grows exponentially relative to the size of the input
- Incrementing the input size by 1 DOUBLES the number of operations

Code Implementation

```
def fib(n):
    if n <= 1:
        return n
    return fib(n-1) + fib(n-2)
```

Factorial Complexity O(n!)

- Algorithm grows proportionally to the factorial of n

Code Implementation

```
def permutations(arr, path=[]):
    if not arr:
        print(path)
        return
    for i in range(len(arr)):
        permutations(arr[:i] + arr[i+1:], path + [arr[i]])
```

Practice Questions

What is the Big O time complexity when you have a loop within a loop (assuming each loop runs n times)?

- o(n^2) because the outer loop runs n times. The inner loop also runs n times for each outer loop iteration. Total operations = n * n = n^2. Runtime grows quadratically as n increases.

How would the following be written: O(100n^2)?

- O(n^2) in Big O notation, constant multipliers are ignored. The 100 is a constant factor, so we removed it. The exponent on n stays the same, so the simplified form is O(n^2).

What Big O is associated with Divide and Conquer?

- Many divide and conquer algorithms repeatedly split the problem in half. Each split reduces the problem size exponentially, leading to logarithmic growth in the number of steps. Examples: Binary search, balanced binary tree operations.

What is the correct way to write: O(n^2 + n)

- In Big O notation, we keep only the dominant term, the one that grows the fastest as n increases. n^2 grows faster than n, so n becomes insignificant for large inputs. Constants are also dropped, leaving O(n^2).

The most efficient Big O is?

- Constant time, the number of operations stays the same no matter how large the input gets. It's the fastest possible runtime because execution time does not grow with n. Examples: Accessing an element in an array by index, pushing to the top of a stack.