Big O Notation - Detailed Notes

Author: Shekh Mostofa Abedin

Key Concepts

Common Big O Notations

1. **O(1) - Constant Time**
   - The algorithm's runtime does not change with the size of the input.
   - Example: Accessing a specific element in an array by index.
2. **O(N) - Linear Time**
   - The runtime grows linearly with the size of the input.
   - Example: Iterating through an array of size N.
3. **$O(N^2)$ - Quadratic Time**
   - The runtime is proportional to the square of the input size. Often occurs with algorithms involving nested loops.
   - Example: A double loop over an array.
4. **O(log N) - Logarithmic Time**
   - The runtime grows logarithmically as the input size increases. Common in algorithms that divide the problem in half at each step (e.g., binary search).
   - Example: Binary search in a sorted array.
5. **O(N log N) - Log-Linear Time**
   - The runtime is slightly worse than linear but better than quadratic. Common in efficient sorting algorithms.
   - Example: Merge sort and quicksort.
6. **O(2^N) - Exponential Time**
   - The runtime doubles with each additional element in the input. Extremely inefficient for large inputs.
   - Example: Recursive algorithms that solve subproblems multiple times.
7. **O(N!) - Factorial Time**
   - The runtime is proportional to the factorial of the input size. Very poor performance, typically seen in algorithms that generate all possible permutations.
   - Example: Brute-force solutions to the traveling salesman problem.

How to Calculate Big O Notation

1. **Identify the Input Size (N):**

   - Determine what the input is and define its size as `N`. The input size could be the length of an array, the number of elements in a list, or the number of nodes in a tree.

2. **Analyze the Loops:**

   - **Single Loops:** If an algorithm has a loop that iterates `N` times, the time complexity is typically O(N).
   - **Nested Loops:** If there are nested loops, multiply the complexities of the loops. For example, a loop inside another loop that both run `N` times results in $O(N^2)$.
   - **Logarithmic Loops:** If the loop decreases the problem size by half each time (e.g., `i = i * 2`), the time complexity is O(log N).

3. **Analyze Conditional Statements:**

   - Conditional statements like `if-else` do not usually affect the time complexity unless the conditions contain loops or recursive calls. In such cases, analyze the code within those conditions.

4. **Focus on the Most Significant Term:**

   - When combining terms, keep only the most significant one (i.e., the one that grows the fastest). For example, if an algorithm has steps that take O(N) and $O(N^2)$ time, the overall time complexity is $O(N^2)$.

5. **Drop Constants and Lower-Order Terms:**

   - Big O notation describes the upper bound, so drop any constants or less significant terms. For instance, O(3N) simplifies to O(N), and $O(N^2 + N)$ simplifies to $O(N^2)$.

Examples of Big O Calculation

1. **Example 1: Single Loop**

   ```python
   def example_function(arr):
       for i in range(len(arr)):
           print(arr[i])
   ```

   - **Analysis**: The loop runs `N` times (where `N` is the size of the array).
   - **Big O Notation**: O(N).

2. **Example 2: Nested Loop**

   ```python
   def example_function(arr):
       for i in range(len(arr)):
           for j in range(len(arr)):
               print(arr[i], arr[j])
   ```

- **Analysis**: The outer loop runs N times, and for each iteration of the outer loop, the inner loop also runs N times.
- **Big O Notation**: O(N) * O(N) = O(N^2).

3. **Example 3: Logarithmic Loop**

```
def example_function(n):
    while n > 1:
        n = n // 2
```

- **Analysis**: The loop decreases n by half each time, resulting in a logarithmic number of iterations.
- **Big O Notation**: O(log N).

4. **Example 4: Multiple Terms**

```
def example_function(arr):
    for i in range(len(arr)):   # O(N)
        print(arr[i])

    for j in range(len(arr)):   # O(N)
        for k in range(len(arr)):  # O(N)
            print(arr[j], arr[k])
```

- **Analysis**: The first loop runs in O(N), and the nested loop runs in O(N^2).
- **Big O Notation**: O(N) + O(N^2) simplifies to O(N^2) since the quadratic term dominates.

Big O Notation Comparison

- **Hierarchy of Common Complexities**:
  - O(1) < O(log N) < O(N) < O(N log N) < O(N^2) < O(2^N) < O(N!)

Common Algorithms and their Big O Notations

1. Bubble Sort - O(n^2)

```
def bubble_sort(arr):
    n = len(arr)
    # O(n^2) - Overall Big O
    for i in range(n):                 # O(n) - Outer loop runs n times.
        for j in range(0, n-i-1):      # O(n) - Inner loop runs (n-i-1) times,
which is approximately n times.
            if arr[j] > arr[j+1]:      # O(1) - Comparison is a constant time
operation.
                arr[j], arr[j+1] = arr[j+1], arr[j]  # O(1) - Swapping elements is
```

```
a constant time operation.
    return arr
```

2. Merge Sort - O(n log n)

```
def merge_sort(arr):
    # O(n log n) - Overall Big O
    if len(arr) > 1:
        mid = len(arr) // 2
        L = arr[:mid]
        R = arr[mid:]

        merge_sort(L)  # O(log n) - Recursive call on the left half.
        merge_sort(R)  # O(log n) - Recursive call on the right half.

        i = j = k = 0

        while i < len(L) and j < len(R):  # O(n) - Merging the sorted halves.
            if L[i] < R[j]:               # O(1) - Comparison is a constant time
operation.
                arr[k] = L[i]
                i += 1
            else:
                arr[k] = R[j]
                j += 1
            k += 1

        while i < len(L):                 # O(n) - Merging remaining elements.
            arr[k] = L[i]
            i += 1
            k += 1

        while j < len(R):                 # O(n) - Merging remaining elements.
            arr[k] = R[j]
            j += 1
            k += 1
```

Quick Sort - O(n log n) on average, O(n^2) in the worst case

```
def quick_sort(arr):
    # O(n log n) on average, O(n^2) in the worst case - Overall Big O
    if len(arr) <= 1:
        return arr
    pivot = arr[len(arr) // 2]
    left = [x for x in arr if x < pivot]
    middle = [x for x in arr if x == pivot]
```

```python
    right = [x for x in arr if x > pivot]
    return quick_sort(left) + middle + quick_sort(right)
```

Linear Search - O(n)

```python
def linear_search(arr, x):
    # O(n) - Overall Big O
    for i in range(len(arr)):         # O(n) - Looping through n elements in the array.
        if arr[i] == x:               # O(1) - Comparison is a constant time operation.
            return i
    return -1
```

Binary Search - O(log n)

```python
def binary_search(arr, x):
    # O(log n) - Overall Big O
    left, right = 0, len(arr) - 1
    while left <= right:               # O(log n) - The loop runs log(n) times, halving the search space each time.
        mid = (left + right) // 2      # O(1) - Calculating the midpoint is a constant time operation.
        if arr[mid] == x:              # O(1) - Comparison is a constant time operation.
            return mid
        elif arr[mid] < x:             # O(1) - Comparison and reassignment are constant time operations.
            left = mid + 1             # O(1) - Reassignment is a constant time operation.
        else:
            right = mid - 1            # O(1) - Reassignment is a constant time operation.
    return -1
```