Big O Notation

Big O notation is used to describe the performance or complexity of an algorithm in terms of time or space requirements as the input size grows.

1. O(1) - Constant Time Complexity

An algorithm with O(1) complexity executes in the same amount of time regardless of the input size.

```
int getFirstElement(int arr[], int size) {
    return arr[0];  // Always takes constant time
}
```

- **Explanation**: This function returns the first element of an array, and the time it takes does not depend on the size of the array.

2. O(n) - Linear Time Complexity

An algorithm with O(n) complexity scales linearly with the input size.

```
void printElements(int arr[], int size) {
    for (int i = 0; i < size; i++) {
        std::cout << arr[i] << " ";  // Takes time proportional to 'n'
    }
}
```

- **Explanation**: The function iterates over all elements in the array, so if the size doubles, the time taken also doubles.

3. O(n^2) - Quadratic Time Complexity

An algorithm with O(n^2) complexity involves a nested loop where the number of operations scales with the square of the input size.

```
void printPairs(int arr[], int size) {
```

```
    for (int i = 0; i < size; i++) {
        for (int j = 0; j < size; j++) {
            std::cout << "(" << arr[i] << ", " << arr[j] << ") ";  // Nested loops
        }
    }
}
```

- **Explanation**: The function prints all possible pairs in the array, and the number of operations grows quadratically with the array size.

4. O(log n) - Logarithmic Time Complexity

An algorithm with O(log n) complexity reduces the problem size by a constant factor at each step.

```
int binarySearch(int arr[], int size, int target) {
    int left = 0, right = size - 1;
    while (left <= right) {
        int mid = left + (right - left) / 2;
        if (arr[mid] == target) {
            return mid;
        } else if (arr[mid] < target) {
            left = mid + 1;
        } else {
            right = mid - 1;
        }
    }
    return -1;  // Not found
}
```

- **Explanation**: The binary search algorithm repeatedly halves the search space, so the number of operations grows logarithmically with the input size.

5. O(n log n) - Linearithmic Time Complexity

An algorithm with O(n log n) complexity is common in efficient sorting algorithms like Merge Sort or Quick Sort.

```
void merge(int arr[], int left, int mid, int right) {
  // Merging two halves (simplified)
}


void mergeSort(int arr[], int left, int right) {
  if (left < right) {
    int mid = left + (right - left) / 2;
    mergeSort(arr, left, mid);
    mergeSort(arr, mid + 1, right);
    merge(arr, left, mid, right);
  }
}
```

- **Explanation**: The merge sort algorithm divides the array into halves recursively, and then merges them back together, leading to a time complexity of O(n log n).

6. O(2^n) - Exponential Time Complexity

An algorithm with O(2^n) complexity has a growth rate that doubles with each additional element in the input.

```
int fibonacci(int n) {
  if (n <= 1) return n;
  return fibonacci(n - 1) + fibonacci(n - 2);
}
```

- **Explanation**: The recursive Fibonacci function computes the result by making two recursive calls for each input, leading to exponential growth in the number of operations.