Analysis of Code Running Time using Big-O Notation

Part I: Analyzing Code for Running Time

A.

```
int array_sum(int[] a, int n) {
    int i;
    int sum = 0;
    for (i = 0; i < n; i++) {
        sum = sum + a[i];
    }
    return sum;
}
```

Analysis:

- The loop runs from $i = 0$ to $i < n$, iterating n times.
- Each iteration involves a constant-time operation.

Time Complexity: $O(n)$ (Linear time)

B.

```
int sum = 0;
for (i = 0; i < n; i++) {
    for (j = 0; j < n; j++) {
        for (k = 0; k < n; k++) {
            if (i == j && j == k) {
                for (l = 0; l < n * n * n; l++) {
                    sum = i + j + k + l;
                }
            }
        }
    }
}
```

Analysis:

- The first three loops iterate n times each, contributing $O(n^3)$.
- The condition $i == j == k$ holds for $O(n)$ cases.
- Inside the condition, the innermost loop runs $O(n^3)$ times.
- Final complexity: $O(n^3) \times O(n^3) = O(n^6)$.

Time Complexity: $O(n^6)$ (Polynomial time)

C.

```
int sum(int a, int b) {
    int c = a + b;
    return c;
}
```

Analysis:

- This function simply performs an addition and return, both of which take constant time.

Time Complexity: $O(1)$ (Constant time)

D.

```
void fun(int n) {
    int i, j;
    for (i = 1; i <= n; i++)
        for (j = 1; j < log(i); j++)
            System.out.println("CPSC 215");
}
```

Analysis:

- The outer loop runs n times.
- The inner loop runs $O(\log i)$ times for each i.
- Summing up over all i:

$$\sum_{i=1}^{n} O(\log i) = O(n \log n)$$

Time Complexity: $O(n \log n)$

E.

```
int binarySearch(int[] arr, int target) {
    int left = 0;
    int right = arr.length - 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;
}
```

Analysis:

- Binary search repeatedly halves the search space.
- Each iteration reduces the problem size by a factor of 2.

Time Complexity: $O(\log n)$ (Logarithmic time)