Problem 2:

- There are 3 *for loops* in the algorithm so the time complexity would be *O(N)* for each of the loops. For the worst case the *if- else statement* would be making 2 comparisons for every run of the first two loops and there would be 4 comparisons for the last *for loop* for each run which would be *O(N)* for each loop. In the best case there would only be one comparison for every time the *for loop* runs.

```
    for (int i = 0; i < middle; i++) {
        if(a[i] > a[i+1]) {
            temp = a[i];
            a[i] = a[i+1];
            a[i+1] = temp;
        }
        else {
            a[i] = a[i];
            a[i+1] = a[i+1];
        }
} // sorted the first half of the array

    for (int j = middle + 1; j < high; j++) {
        if (a[j] > a[j+1]) {
            temp = a[j];
            a[j] = a[j+1];
            a[j+1] = temp;
        }
        else {
            a[j] = a[j];
            a[j+1] = a[j+1];
        }
    } // sorted the second half of the array

int [] temporary = new int [M];
int b = low;
int c = middle;

for (int k = 0; k < M; k++) {
    if (b == middle) {
        temporary[k] = a[c++];
    }
    else if (c == high) {
        temporary[k] = a[b++];
    }
    else if (a[c] < a[b]) {
        temporary[k] = a[c++];
    }
    else {
        temporary [k] = a[b++];
    }
} // combined the two sorted halves
```

- The bottleneck, shown by the highlighted parts, occurs at the increments of each loop since the frequency for each increment is approximately N^2. Since there are three loops and there are 7 increments in total this bottleneck occurs at 7 places.
- This is a stable sort because it will sort each element in the order that it receives it without the order of the sort being interchangeable since it's mergesort. The elements when swapped won't be interchangeable in the swapped positions.