High Level Idea

For an interval $P = (a_i, b_i)$, we denote access to the element of a_i as `P[0]`, and b_i as `P[1]`. Also, if we have `a`, `b`. and we want to construct $P = (a, b)$, we denote that as `P = (a, b)`

We define for every two intervals $P = (a_i, b_i)$ and $Q = (a_j, b_j)$. If we have $a_i \leq a_j$ and $b_j \leq b_i$ we say that Q is **contained** in P.

This definition will be useful for the explanation. We define a utility function called `IsContained(ai, bi, aj, bj)` to check if $Q = (a_j, b_j)$ is **contained** in $P = (a_i, b_i)$:

```
IsContained(P, Q):
    return (P[0] ⩽ Q[0]) and (Q[1] ⩽ P[1])
```

We also have the auxiliary function `OverlapLength(P, Q)` that calculate the overlap length of interval P and Q:

```
OverlapLength(P, Q):
    return max(0, min(P[1], Q[1]) - max(P[0], Q[0]))
```

Before we begin, first check if the condition `n ⩾ 2` is true, if we are given `0` or `1` interval, it doesn't make any sense to find the maximum overlap length. In this case, we could just end our program, and possibly give / throw a exception.

We begin by sorting the array `A[1 .. n]` and permuting the array `B[1 .. n]` to maintain correspondence between endpoints, in $O(n \log n)$ time (`MergeSort`).

We iterate through these sorted `A[1 .. n]` (with associated `B[1 .. n]`). We keep a global variable `maximumLength` to track the maximum overlap length between intervals we have seen so far. This sub-process is:

```
IterateThroughIntervals(A[1 .. n], B[1 .. n]):
    maximumLength ← 0
    lastInterval ← null
    for i ← 1 to n:
        if lastInterval is not null:
            currentInterval ← (A[i], B[i])
            if IsContained(lastInterval, currentInterval):
                maximumLength ← max(maximumLength, currentInterval)
            else:
                maximumLength ← max(
                    maximumLength,
                    OverlapLength(lastInterval, currentInterval)
                )
                lastInterval ← currentInterval
    return maximumLength
```

Time Complexity Analysis

We see that the the initial sorting of `A` takes $O(n \log n)$ times, and then the `IterateThroughIntervals` takes $O(n)$ (since it has one `for` loop that loop through all intervals of size n). So, the total time complexity is dominated by the initial sorting, which is $O(n \log n)$