Interval Breaks

Reasoning

We can use a similar approach we did to the weighted interval scheduling problem, using dynamic programming. This time however, instead of simply checking if the current interval overlapps with the last ones, we have to check if it overlaps with the others with the result from our new 2d array of travel times. So our new dynamic programming setup is as follows:

What: $S[j] =$ max weight of the non overlapping intervals from the first j intervals.

How: $S[j] = \begin{cases} S[j] = 0 \text{ if } j = 0 \\ S[j] = \max\{S[j-1], 1 + S[k]\} \text{ if } j > 0 \end{cases}$ where k is the last previous value where the end time plus the travel time doesn't overlap with the start of the current element.

Where: $S[n]$

Psudocode

```
def find_max_classes(A:array<interval>, B:array<array<int>>):
    sort A by finishing time (increasing)
    S[0] = 0
    for j from 1 to n:
        k = j
        while A[k].end + B[k][j] > A[j].start: k --
        S[j] = max{S[j-1], 1 + S[k]}
    return S[n]
```

Proof

First of all, assuming that the erlier elements of the S array are correct, we know that $S[j]$ will be the maximum possible value, since the maximum value must be either the previous value, or a pervious maximum, that doesn't overlap with the current element plus the current element, so we know that $S[j]$ is correct as long as all elements before it are correct. The first elements therefore are all we need to prove to be correct for the algorithm to be correct. The first element is going to be the first interval, since it cannot overlap, and having 1 interval is more then 0 intervals (the previous element to the first). Therefore since this is correct, the algorithm is correct.

Running Time

Estimate

The running time is $O(n^2)$

Reasoning

```
def find_max_classes(A:array<interval>, B:array<array<int>>):
    sort A by finishing time (increasing) # O(n log n)
    S[0] = 0 # O(1)
    for j from 1 to n: # O(n^2)
        k = j # O(1)
        while A[k].end + B[k][j] > A[j].start: k -- # O(n)
        S[j] = max{S[j-1], 1 + S[k]}# O (1)
    return S[n] # O(1)
```

Since the longest part of the algorithm is $O(n^2)$, the algorithm takes $O(n^2)$ time.