Finding the biggest slice

Your goal is to find the slice of a list that has the largest sum. You will find both the largest sum as well as the indices starting and ending the slice. That is, if `L[3:9]` is the largest slice and the sum of the elements is `500`, you will return the tuple `(500, 3, 9)`. Call your function `slicesum` and put it in a file called `slicesum.py`. Recall the standard python convention that slices `L[start:end]` start at `start` and go up to *but not including* `end`.

First, a short and slow solution

It's often useful to just get something working at first. In this case, it's possible to just try every slice, sum its elements, and find the max.

```
def slowslicesum(L):
    return max((sum(L[i:j]), i, j) for j in range(len(L)+1) for i in range(j))

# Let's try it a couple times to see.
print(slowslicesum([-1, 2, 3, -1]))
print(slowslicesum([-1, 2, 3, -1, 4, -10]))
```

```
(5, 1, 3)
(8, 1, 5)
```

Like a lot of one-liners, there's a lot in there. Let's break it down for practice understanding such expressions. It is taking the max of the sum of a slice for all possible slices of the list. We could have written it out more explicitly, as follows.

```
def longslowslicesum(L):
    sums = []
    for j in range(len(L) + 1):
        for i in range(j):
            sums.append((sum(L[i:j]), i, j))
    return max(sums)
```

You should be able to recognize that this function takes $O(n^3)$ time. That's pretty bad. It's not too hard to get it down to $O(n^2)$ time by precomputing the sums of some slices.

```
def longquadraticslicesum(L):
      # Precompute the sum of the first i elements
      P = [sum(L[:i]) for i in range(len(L) + 1)]
      sums = []
      for j in range(len(L) + 1):
          for i in range(j+1):
              sums.append((P[j] - P[i], i, j))
      return max(sums)
```

And you will know you understand it when you can see why it's equivalent to this shorter version.

```
def quadraticslicesum(L):
    n = len(L) + 1
    P = [sum(L[:i]) for i in range(n)]
    return max((sum(L[i:j]), i, j) for j in range(n) for i in range(j))
```

Breaking Ties

If you play around with the slow solution a bit, you might notice that often, the solution is not unique. We'd like to produce a consistent result. For this assignment, we will strive to always take the longest. Moreover, if there are multiple slices with the same sum and the same length, then return the leftmost.

This means that some extra logic would have to go into the slow solutions above. Currently, those solutions just take the max of a collection of tuples. If there is a tie in the first entry of the tuples (the sum) then the sort would order by the second entry (the left index) and then by the third entry (the right index). That won't necessarily produce the longest. If you really want to use `max`, you might consider passing a `key` function.

Let's Divide and Conquer

Once you decide to try divide and conquer on a list, you might as well try to split the list in half and try both sides. There are are only three possibilities for what the optimal solution can look like.

1. The biggest slice is in the left half.
2. The biggest slice is in the right half.
3. The biggest slice spans the two halves.

The first two cases are really easy to handle by making a recursive call. In the last case, we can consider the optimal slice as a suffix of the left half of the list and a prefix of the right half of the list. Finding the largest prefix (or suffix) slice is clearly going to be much easier.

To Do: Write a function `prefix` that takes a list `L` and returns a pair `(s, index)` such that `L[:index]` is the largest prefix slice and `sum(L[:index])` is `s`. Break ties by return the longest among the largest.

To Do: Write a function `suffix` that takes a list `L` and returns a pair `(s, index)` such that `L[index:]` is the largest suffix slice and `sum(L[index:])` is `s`. Break ties by return the longest among the largest.

For the divide and conquer approach try the left half, try the right half, and try to combine the largest suffix of the left half with the largest prefix of the right half. Then, it will return the biggest of these slices. The running time should be $O(n \log n)$. This will be *much* faster than the quadratic solution.