Discussion Board 2:

Non-Recursive Sorting Algorithms

As a precursor to discussing my work, I decided it would be fun to write these algorithms in C# and test them to make sure I understood how they work. So instead of pseudocode, I have some real C# code to provide to you to look through. I also have a private function used to swap indices of the array:

```
private static void Swap<T>(this T[] arr, int idx1, int idx2) =>
    (arr[idx1], arr[idx2]) = (arr[idx2], arr[idx1]);
```

The first sorting algorithm I decided to research was the BubbleSort algorithm. Here is some C# code to display BubbleSort:

```
private static int[] BubbleSort(this int[] arr)
{
    for (int i = 0; i < arr.Length - 1; i++)
    {
        bool swapped = false;
        for (int j = 0; j < arr.Length - i - 1; j++)
        {
            if (swapped = arr[j] > arr[j + 1])
            {
                arr.Swap(j, j + 1);
            }
        }
        if (!swapped) break;
    }

    return arr;
}
```

Time Complexity:

The time complexity for this algorithm is simple to analyze. The worst-case scenario would be if the array is in reverse order, meaning that it will never break out of the loop early, and every loop condition is met. This complexity could be written as:

$$n + (n - 1) + (n - 2) + (n - 3) \ldots + 1 = \sum_{k=1}^{n} k = \frac{n(n+1)}{2} = \frac{n^2 + n}{2}$$

The dominant part of this result is n^2, as the number of input items gets larger, the n^2 part of the equation becomes more significant than the rest so we round up to $O(n^2)$.

Space Complexity:

The space complexity of this algorithm is also simple. This is an in-place algorithm, meaning that it does not use any memory besides the memory already available by the input array (apart from a few loop variables, and a Boolean to determine whether items in the array were swapped). This means that our space complexity is O(1).

The second algorithm I researched was Bucket Sort. My implementation of Bucket Sort actually implements Bubble Sort, so the worst-case scenario might be a little more complex to analyze… Here is the C# implementation:

```
private static int[] BucketSort(int[] arr)
{
    List<int>[] buckets = new List<int>[arr.Length];

    for (int i = 0; i < arr.Length; i++)
        buckets[i] = new List<int>();

    int maxValue = 0;
    for (int i = 0; i < arr.Length; i++)
        maxValue = arr[i] > maxValue ? arr[i] : maxValue;

    for (int i = 0; i < arr.Length; i++)
        buckets[(arr.Length - 1) * arr[i] / maxValue].Add(arr[i]);

    for (int i = 0; i < arr.Length; i++)
        buckets[i] = buckets[i].ToArray().BubbleSort().ToList();

    int arrIdx = 0;
    for (int i = 0; i < arr.Length; i++)
    {
        for (int j = 0; j < buckets[i].Count; j++)
        {
            arr[arrIdx++] = buckets[i][j];
        }
    }

    return arr;
}
```

This algorithm works by creating an array of n "Buckets". It then finds the maximum value within the input array, and iterates through the input array and does a pre-sort by taking the array index's value, dividing it by the maximum value, and multiplying it by the number of buckets, and casting that value to an integer. This organizes the values into subsets of value ranges. Then it sorts each of those individual buckets, and flattens the two dimensional bucket array.

Time Complexity:

Each of the operations apart from the Bubble Sort piece are going to be O(n) operations since they're all a single iteration through the input array. Since Bubble Sort's worst-case time complexity is $O(n^2)$, that is going to be our biggest risk in time complexity for Bucket Sort. The worst-case scenario would again be that the array is in reverse order, and would actually give us a worse *practical* time complexity than Bubble Sort because of all of the additional operations taking place to pre-sort the array. In this time complexity analysis, I'm also ignoring the Linq functions converting the list to an array and back. In reality, we would have an implementation of Bubble Sort that would take in a List as a parameter so that this operation wouldn't be necessary, I just didn't have the time to write another implementation. The complexity of this algorithm could be written as:

$$4n + \frac{n^2 + n}{2}$$

Which again results in $O(n^2)$.

Space Complexity:

The space complexity of Bucket Sort depends on the number of Buckets (k) and the number of items in the input array (n). The worst case space complexity would be if each array item ended up in the same bucket, which would be O(n*k).