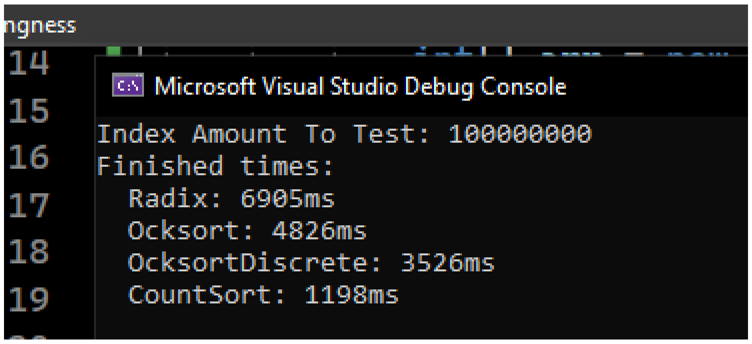
Ocksort

A really fast sorting algorithm, not the fastest in the world, but definitely the stupidest for its performance.

Big O time complexity: O(n+k)

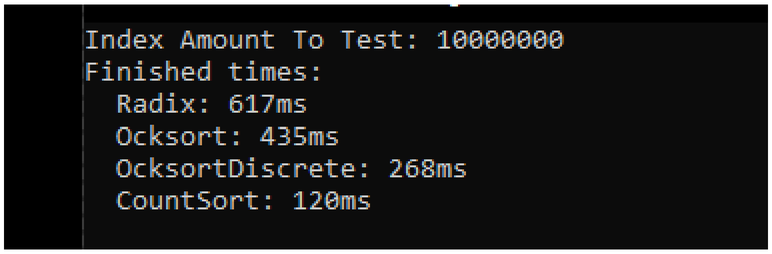
Where k is the difference between the mix and max values of the data set

Performance Stats with 100 million indexes



```
Index Amount To Test: 100000000
Finished times:
  Radix: 6905ms
  Ocksort: 4826ms
  OcksortDiscrete: 3526ms
  CountSort: 1198ms
```

Performance Stats with 10 million indexes



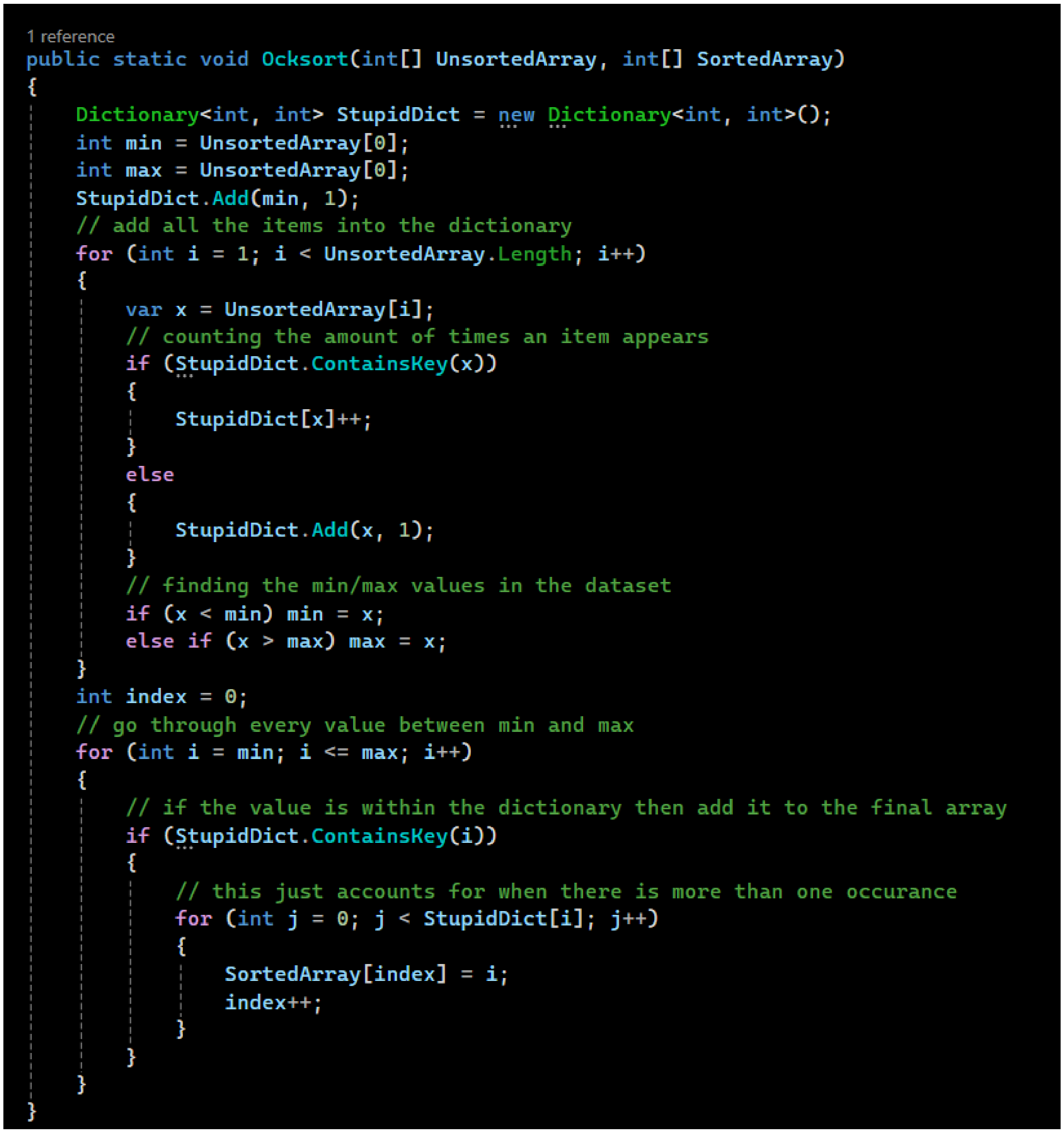
```
Index Amount To Test: 10000000
Finished times:
  Radix: 617ms
  Ocksort: 435ms
  OcksortDiscrete: 268ms
  CountSort: 120ms
```

So what is this crackpot sorting algorithm I concocted at 2 am? I am calling it by the name of Ocksort, and a slight variant of it as OcksortDiscrete. The discrete version as I call it tends to perform around 50% better compared to the normal version, although it only works with unique numbers. The normal sorting algorithm allows for duplicates but is slower.

The exact method I used to achieve results outplaying Radix Sort is honestly incredibly stupid, and I have a strong feeling that someone much smarter than myself could improve the process. In the normal algorithm, the first thing I do is go through every item in the list and add them to a dictionary, the dictionary key storing the item in the array, and the dictionary value storing the number of times it is present in the array. During this loop, I also store the lowest and highest values contained within the dataset.

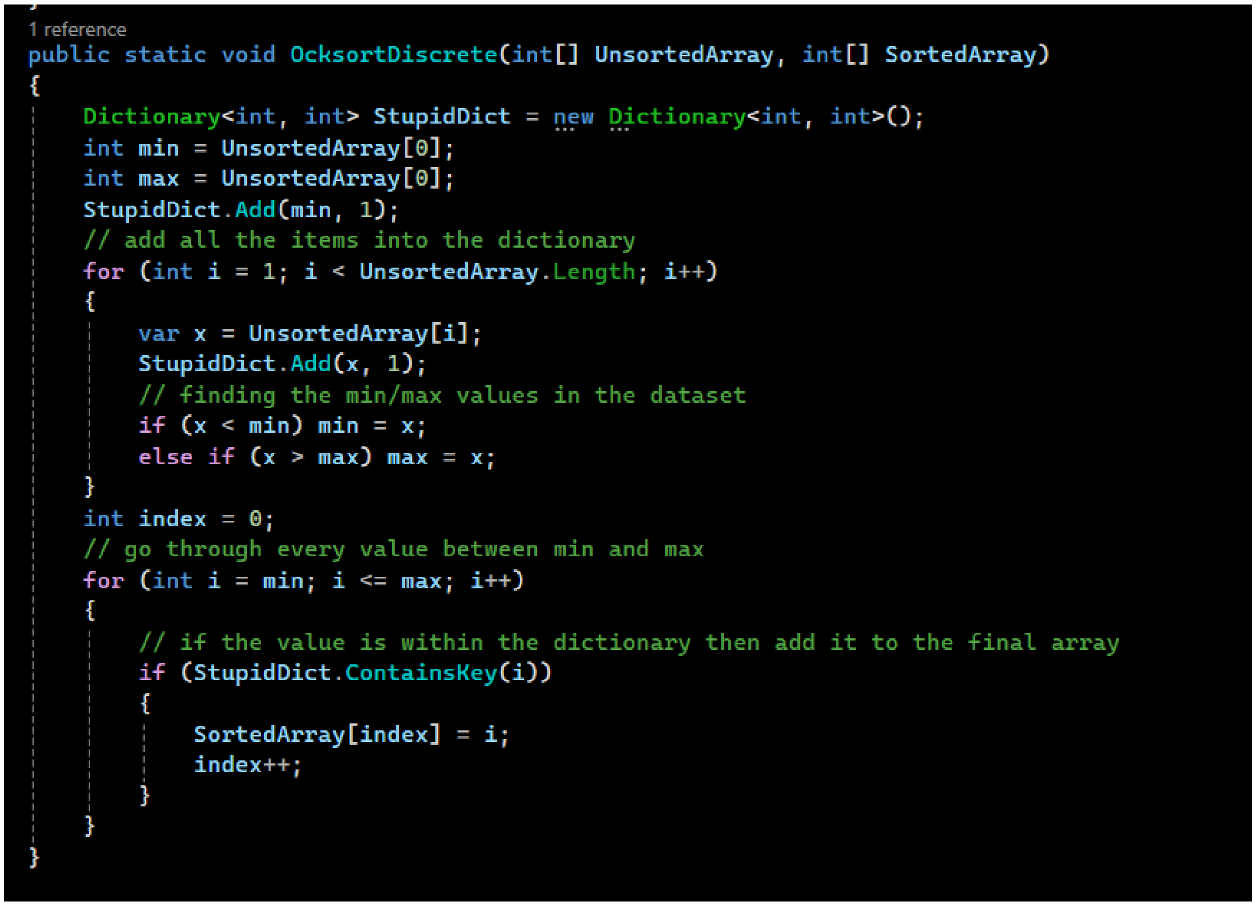
Then it moves on to a separate loop which goes from the min up to the max value, checking if the key is present in the dictionary, and if it is then adding the relevant amount of itself into the output array. And that's it.

Below you will find the implementation I used for the original testing labeled as “Ocksort”



```
1 reference
public static void Ocksort(int[] UnsortedArray, int[] SortedArray)
{
    Dictionary<int, int> StupidDict = new Dictionary<int, int>();
    int min = UnsortedArray[0];
    int max = UnsortedArray[0];
    StupidDict.Add(min, 1);
    // add all the items into the dictionary
    for (int i = 1; i < UnsortedArray.Length; i++)
    {
        var x = UnsortedArray[i];
        // counting the amount of times an item appears
        if (StupidDict.ContainsKey(x))
        {
            StupidDict[x]++;
        }
        else
        {
            StupidDict.Add(x, 1);
        }
        // finding the min/max values in the dataset
        if (x < min) min = x;
        else if (x > max) max = x;
    }
    int index = 0;
    // go through every value between min and max
    for (int i = min; i <= max; i++)
    {
        // if the value is within the dictionary then add it to the final array
        if (StupidDict.ContainsKey(i))
        {
            // this just accounts for when there is more than one occurance
            for (int j = 0; j < StupidDict[i]; j++)
            {
                SortedArray[index] = i;
                index++;
            }
        }
    }
}
```

There is also the modified version of this with less logic, which runs faster, but also does not allow for duplicate values in the dataset. I quite literally just gutted out the pieces of the code responsible for the duplicate checking/handling, which you will find below.



```
1 reference
public static void OcksortDiscrete(int[] UnsortedArray, int[] SortedArray)
{
    Dictionary<int, int> StupidDict = new Dictionary<int, int>();
    int min = UnsortedArray[0];
    int max = UnsortedArray[0];
    StupidDict.Add(min, 1);
    // add all the items into the dictionary
    for (int i = 1; i < UnsortedArray.Length; i++)
    {
        var x = UnsortedArray[i];
        StupidDict.Add(x, 1);
        // finding the min/max values in the dataset
        if (x < min) min = x;
        else if (x > max) max = x;
    }
    int index = 0;
    // go through every value between min and max
    for (int i = min; i <= max; i++)
    {
        // if the value is within the dictionary then add it to the final array
        if (StupidDict.ContainsKey(i))
        {
            SortedArray[index] = i;
            index++;
        }
    }
}
```

Clearly, there is a gaping flaw in this algorithm, which is also where I think improvements that I am not smart enough to find can be implemented. This massive flaw is the fact that for large gaps in the dataset (such as an input of [0, -1000000, 98765432123]) the runtime can get quite terrible. I haven't thought of a great way to mitigate this, but it would be amazing if anyone could further expand upon this stupid idea of a sorting method to make it less idiotic. Thankfully for datasets where the range is the same or less than the index count, the algorithm performs outstandingly well.

So as far as I have been able to test, my sorting method(s) outperform almost everything else aside from Countsort, which is why I included it on the testing sheet at the very opening of this document. If you have any ways you can think of to improve this then have at it, and if you want to talk you can message me a few ways.

Email: ocks.dev@gmail.com
Discord: ocks_dev