Individual Project 1

CS627

Aidan Polivka

January 7, 2024

My team has been tasked with writing an algorithm to reverse an array of bytes in an audio file. This algorithm must be an in-place algorithm, meaning that we cannot use excess memory to reverse the order of the array. We must use the input array and reverse the pointers in memory on that array without instantiating a new array. To prototype the functionality of the array reversal algorithm, we've decided to begin with creating a simple algorithm to reverse the order of integers in an array in C#. This is the implementation of this prototype array reversal algorithm:

```
private static int[] Reverse(int[] input)
{
    int iterationLength = input.Length / 2;

    // Iterate over half of the array
    for (int i = 0; i < iterationLength; i++)
    {
        // Under the hood, C# 8 allows us to swap pointers in memory doing this.
        // intput[^] syntax allows us to access references at the end of the array.
        (input[i], input[^(i + 1)]) = (input[^(i + 1)], input[i]);
    }

    return input;
}
```

This algorithm only uses a couple additional references in memory: an integer to store the iteration length, and an iteration index variable for the current position in the array. In this algorithm, we're only iterating over half the length of the input array, and swapping the respected front, and end indices. C# 8 syntax allows us to easily access indices at the end of the array using arr[^1] syntax, and it allows us to swap using .NET tuple syntax.

The space complexity of this algorithm is simple to calculate. Since we're only using the existing memory input to the function, it's O(1). The time complexity is also a simple calculation: because we're iterating over half of the array, the complexity is $\frac{n}{2}$. Because our dominant factor is n, the worst-case time complexity is O(n).

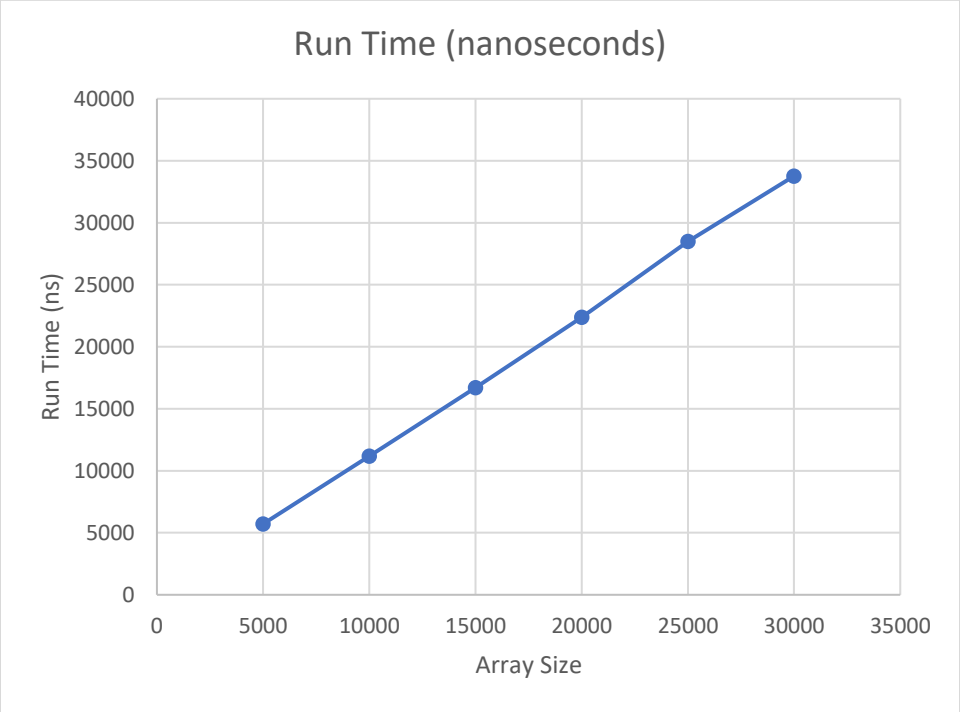
I created a couple test scenarios to prove that the reversal algorithm works, and then I created some additional tests that populate an array with configurable size, with random values between 0 and 10,000. These additional tests create a new array, then calculate the run time of the reversal algorithm. It creates a list of run times for the number of input iterations, then takes the average run time of the reversal algorithm for that input array size.

```
private static void PrintTestCase(string testCaseName, int[] input)
{
    Console.WriteLine($"** {testCaseName} **");
    Console.WriteLine($"Input:  {string.Join(", ", input)}");
    Console.WriteLine($"Output: {string.Join(", ", Reverse(input))}");
    Console.WriteLine("");
}

private static int[] FillRandom(this int[] arr, int min = 0, int max = 10000)
{
    var random = new Random();
    for (int i = 0; i < arr.Length; i++)
        arr[i] = random.Next(min, max);
    return arr;
}

private static void PrintAverageRunTime(string testCaseName, int size, int iterations)
{
    Console.WriteLine($"** {testCaseName} - iterations: {iterations} **");
    List<double> elapsedNS = new List<double>();
    for (int i = 0; i < iterations; i++)
    {
        int[] input = new int[size].FillRandom();
        Stopwatch sw = Stopwatch.StartNew();
        sw.Start();
        Reverse(input);
        sw.Stop();
        elapsedNS.Add(sw.Elapsed.TotalNanoseconds);
    }

    Console.WriteLine($"Average Elapsed Nanoseconds: {elapsedNS.Sum() / iterations}ns");
    Console.WriteLine("");
}
```

This is the output of the main function to create our test scenarios:

```
Individual Project 1

** 1: Odd Array Length **
Input:  1, 2, 3, 4, 5, 6, 7
Output: 7, 6, 5, 4, 3, 2, 1

** 2: Even Array Length **
Input:  1, 2, 3, 4, 5, 6, 7, 8
Output: 8, 7, 6, 5, 4, 3, 2, 1

** 3: Array Size 5000 - iterations: 100000 **
Average Elapsed Nanoseconds: 5712.77ns

** 4: Array Size 10000 - iterations: 100000 **
Average Elapsed Nanoseconds: 11192.489ns

** 5: Array Size 15000 - iterations: 100000 **
Average Elapsed Nanoseconds: 16713.79ns

** 6: Array Size 20000 - iterations: 100000 **
Average Elapsed Nanoseconds: 22386.42ns

** 7: Array Size 25000 - iterations: 100000 **
Average Elapsed Nanoseconds: 28491.658ns

** 8: Array Size 30000 - iterations: 100000 **
Average Elapsed Nanoseconds: 33768.178ns
```

For our specific test scenarios, we took arrays filled with random values between 0 and 10,000 of sizes 5,000, 10,000, 15,000, 20,000, 25,000, and 30,000. Each test was ran 100,000 times, and the average run time nanoseconds were captured. Below is a graph displaying the relationship between input array size and run time in nanoseconds:



From the graph, we can see that there is a linear relationship between array size and the run time of the array reversal algorithm in nanoseconds. Thus, supporting our worst-case time complexity calculation of O(n).