Individual Project 5

CS627

Aidan Polivka

February 4, 2024

Portfolio

Array Reversal Algorithm

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).

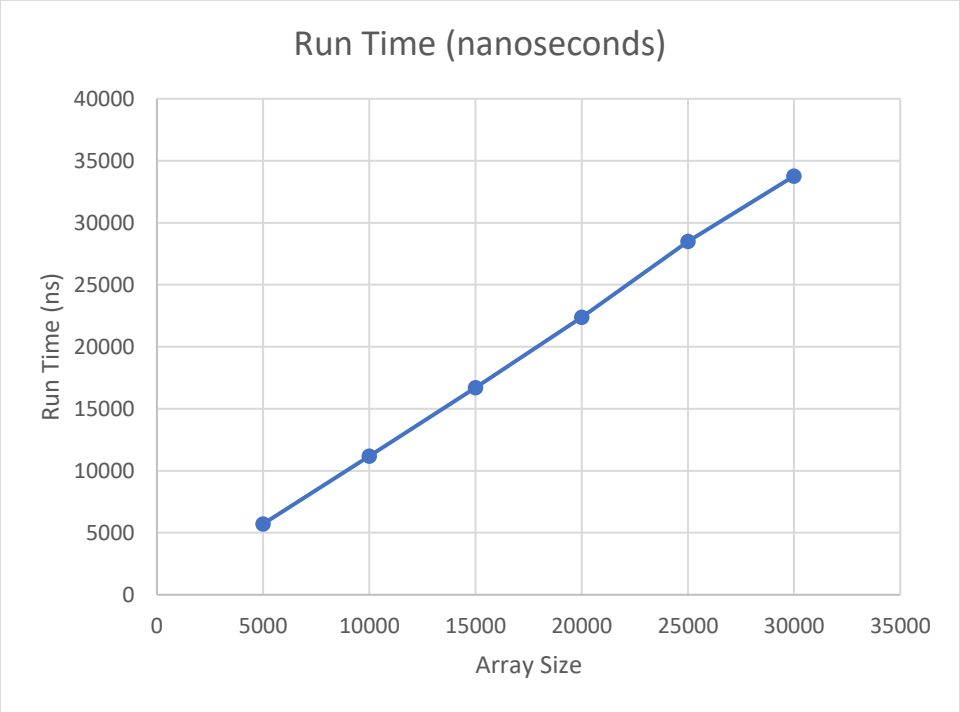
I created a couple test scenarios to prove that the reversal algorithm works, and then I created some additional tests that populate an array with configurable size, with random values between 0 and 10,000. These additional tests create a new array, then calculate the run time of the reversal algorithm. 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).

Divide and Conquer

My team has been tasked with building a search algorithm to find a song in a sorted list of song titles. We've decided to explore an implementation of the binary search algorithm with a twist: instead of using the divide and conquer approach on 2 sub-arrays, we'll be splitting the original array into 3 sub-arrays. This effectively could be called a "Ternary Search" algorithm. This is the implementation we've built in C#:

```
/// <summary>
/// Ternary Searching Algorithm - Splits array into three segments <br />
/// Recursively calls TernarySearch on the segment the search term should be in
/// </summary>
/// <param name="arr">pre-sorted list of strings</param>
/// <param name="searchTerm">search term</param>
/// <param name="first">optional left boundary</param>
/// <param name="last">optional right boundary</param>
/// <returns>index of search term in array, or -1 if not found</returns>
private static int TernarySearch(
    string[] arr,
    string searchTerm,
    int first = 0,
    int? last = null)
{
    // Set operating variables
    last = last ?? arr.Length - 1;
    int left = first + ((int)(last - first) / 3);
    int right = first + ((int)Math.Ceiling((double)(last - first) * 2 / 3));

    // Recursion ending cases
    if (first == last && arr[first] != searchTerm)
    {
        return -1;
    }else if (arr[left] == searchTerm)
    {
        return left;
    } else if (arr[right] == searchTerm)
    {
        return right;
    }

    // Recursive cases
    if (string.Compare(searchTerm, arr[right]) > 0)
    {
        return TernarySearch(arr, searchTerm, right + 1, arr.Length - 1);
    }
    else if (string.Compare(searchTerm, arr[left]) < 0)
    {
        return TernarySearch(arr, searchTerm, 0, left - 1);
    }

    return TernarySearch(arr, searchTerm, left + 1, right - 1);
}
```

So, the question is, what is the worst-case time complexity of this algorithm, and how does it compare to binary search? This algorithm's time complexity is $\log_3 n$, which boiled down to its dominant factor is equal to log(n). Therefore, the complexity of Ternary Search is the same as Binary Search. In my mind, the follow up question would be: Does this statement remain true in practice? And to that I'd say no. Because Ternary Search must perform so many more comparisons than Binary Search, it makes it measurably slower than Binary Search. Therefore, it's impractical to implement Ternary Search over Binary Search.

Greedy Paradigm

For this week's individual project, we were asked to use the greedy paradigm to develop an algorithm to fill an array of disks with an array of files. The goal is to use the fewest number of disks, and to optimize the amount of space used on the disks. Any additional disks will be returned to the store. Input into this algorithm will be an array of file names [0 – n], an array of file sizes [0 – n], an array of disk names [0 – m] and an array of disk sizes [0 – m]. The output should be a mapping of file names to the disk index. Being an object-oriented engineer, I think this would be best handled using custom objects to keep the disk names tied to their corresponding sizes, and the file names tied to their corresponding sizes, but for the sake of the assignment I decided to follow instructions as close as possible. Here is the pseudocode for this greedy algorithm:

```
/* This algorithm's purpose is to organize files onto the minimum number of disks.
 * With this pseudocode, we are making a number of assumptions:
 *  1. input array lengths for fileNames and fileSizes are the same
 *  2. input array lengths for diskNames and diskSizes are the same
 *  3. units for the fileSizes and diskSizes arrays are the same
 *  4. we are receiving valid input, without a need for error handling
 */
private Dictionary<string, int> OrganizeFiles(
    string[] fileNames, decimal[] fileSizes,
    string[] diskNames, decimal[] diskSizes)
{
    // Output dictionary of fileNames and disk indicies
    Dictionary<string, int> mappings = new Dictionary<string, int>();

    // Sort the files list by size descending
    MergeSort(fileNames, fileSizes); // O(nlogn)

    // Iterate through list of files
    for (int i = 0; i < fileNames.Length; i++) { // O(n * m)

        // Find the first disk that the file can fit onto
        for (int j = 0; j < diskNames.Length; j++) {
            if (fileSizes[i] <= diskSizes[j]) {

                // Create a mapping for the file name and disk index
                mappings.Add(fileNames[i], j);

                // Update the disk size at disk index
                diskSizes[j] -= fileSizes[i];

                // Break from this loop when the file is allocated
                break;
            }
        }
    }

    return mappings;
}
```

This algorithm will not always provide the optimal solution. Because of the nature of the greedy algorithm, it does not evaluate all the files at once. It evaluates each file largest to smallest and finds the first disk in the list that has space for it. Consider this scenario:

fileSizes	diskSizes
50	90
30	90
25	90
15	90

Using the greedy algorithm, disk 0 would store files 0 and 1, which would leave 10 mb unused. If the algorithm had a global context, disk 0 would store files 0, 2, and 3, maximizing the amount of space on the disk.

The time complexity of the greedy algorithm is the complexity of the sorting algorithm + the complexity of the file-to-disk mapping algorithm. Merge Sort's time complexity is O(nlogn), and the file-to-disk mapping algorithms complexity is O(number of files * number of disks). The worst-case scenario would be for all the drives to have 0 remaining space except for the last disk. So, for each file we would need to iterate through the entire list of drives. Therefore, the complexity of the file-to-disk mapping algorithm is O(n*m), and the overall time complexity is O(nlogn + nm).

The Merge Sort algorithm complexity dominates the nested for loops for files and drives, so the end time complexity is O(nlogn). Something to consider would be the possibility that the number of drives is equal to the number of input files. This is an unrealistic scenario, and out of character for the file organizing algorithm. If the possibility of this scenario were more realistic, it would be worth considering this algorithm to be $O(n^2)$, but here I think it is acceptable to use the Merge Sort complexity as the dominant factor.

To brute force a solution to this problem we would need to evaluate all permutations of the files (n!) for each disk (m). So, the complexity of such an algorithm would be O(mn!). This is drastically more complex than the greedy algorithm, and in large data sets of n and m it's impractical. There may be another way to optimize the space on these drives without brute force which is overly complex, or the greedy paradigm which is sub-optimal. But, the greedy paradigm provides a simple, maintainable solution that is acceptable enough for this scenario.

Dynamic Programming

For this assignment we've been tasked to implement an image comparison program in pseudocode. We're operating under the assumption that we're dealing with purely black and white images, and that the input images have the same number of rows, with potentially differing numbers of columns. We're not working with a grey scale, but rather pure black and white, so we're using 0 to signify black, and 1 to signify white. The images are represented as 2 dimensional arrays of integers. The comparison algorithm has to compare to an input threshold of difference to compare similarity, and return a Boolean value for whether or not the difference between the images meets, or is lower than the input threshold. Here is the pseudocode I've generated for this assignment:

```
public bool IsSimilar(int[][] imgx, int[][] imgy, int threshold) {
    // initializes D as 2d array of size J,K filled with zeros
    int[,] D = new int[imgx[0].length, imgy[0].length];

    // iterate through rows
    for (int i = 0; i < imgx.length; i++) {

        // compare columns of imgx and imgy
        for (int j = 1; j < imgx[i].length; j++) {
            for (int k = 1; k < imgy[i].length; k++) {
                int relationA = imgx[i][j] == imgy[i][k]
                    ? D[j-1, k-1]
                    : D[j-1, k-1] + 1;

                // relation B is obsolete if j is within k
                int relationB = j >= imgy[i].length
                    ? D[j-1, k] + 1
                    : int.MaxValue;

                // relation C is obsolete if k is within j
                int relationC = k >= imgx[i].length
                    ? D[j, k-1] + 1
                    : int.MaxValue;
                D[j, k] = Math.Min(relationA, Math.Min(relationB, relationC));
            }
        }
    }

    // iterate over bottom line of D
    int minValue = int.MaxValue;
    for (int k = 0; k < D[0].length; k++) {
        minValue = Math.Min(minValue, D[D.length - 1, k]);
    }

    return minValue <= threshold;
}
```

This pseudocode utilizes dynamic programming principles by leveraging matrix D for memoization, which stores the results of previously computed edit distances. By iterating over the bottom row of stored subproblem solutions, the algorithm efficiently finds the minimum edit distance. This approach minimizes redundant calculations and optimizes the overall computation. The time complexity is O(I * J * K), where I is the number of rows, J is the number of columns in imgx, and K is the

number of columns in imgy. This complexity is essential for comparing the two images, making the algorithm both optimal and efficient in determining image similarity.

Heuristic Approach

State Space

Our goal for this project is to come up with a plan to brute force a solution for a given starting state of a Sudoku puzzle. I think the best approach would be to iterate over the matrix top-down left-right recursively. Here are our base cases:

If the current cell is not empty, we'll continue to the next column in the row. If we're at the last column in the row, we'll move to the first column of the next row. If we're past the last column of the last row, we'll return true.

Our primary operating case is if we land on a cell that is empty. In this case, we'll iterate over the possible values of that cell (1-9) and determine if that is a valid entry (this is where heuristics comes into play). If the current number in iteration is a valid entry, we'll call the solve function recursively on that state of the matrix. If it's not a valid entry, we'll continue in iteration between 1-9. If we've tried every possibility for that cell and not found a valid solution, we'll reset the value of the cell to empty and return false. If the solve function returns true, we'll return true and propagate that answer up through the recursive instances. I think this would make each node of our state traversal tree a state snapshot of the entire puzzle state. I also think this would result in a depth-first search algorithm, so our goal with heuristics would be to break out of the depth search as soon as possible if we're going down a path that is not valid.

Traversal Time Complexity

The very worst case would be if we weren't determining validity at each vertex and had to traverse the depth of every node until we got to the furthest right, furthest bottom vertex of the state tree. This would be an approach without heuristics or domain understanding, we're simply testing every permutation of the tree that matches the input state. We'd also be assuming that every input cell is blank, which is somewhat unrealistic that both worst-case scenarios come true. If this was the case, we'd have to test each cell for the 9 possible options. So, our time complexity would be $O(9^{NxN})$. This is strictly for the traversal of the state tree, we could run into additional complexity when we go about determining valid input into the current cell. The heuristics approach is going to make this more complex, but should hopefully make the average run time much shorter.

Heuristics Search

The heuristic aspects of this algorithm would go into each iteration of the 1-9 for the empty cell. We would determine if the entry is valid on a few different domain-oriented constraints:

1. Is the entry already in the row?
2. Is the entry already in the column?
3. Is the entry already in the 3x3 block?

There are additional rules to Sudoku/more complex ways to determine if the input is valid that I don't fully know, but for the sake of this assignment we'll stick to the simple rules of Sudoku.

Pseudocode

```
    internal class Program
    {
        static void Main(string[] args)
        {
            int[,] startState = {
                { 5, 1, 0, 0, 7, 0, 0, 0, 0 },
                { 0, 0, 0, 0, 0, 0, 8, 0, 0 },
                { 0, 4, 0, 2, 0, 0, 0, 9, 1 },
                { 0, 0, 0, 3, 0, 0, 6, 0, 0 },
                { 0, 8, 0, 0, 0, 0, 2, 0, 0 },
                { 4, 0, 0, 0, 0, 5, 0, 8, 3 },
                { 0, 9, 0, 1, 0, 0, 0, 3, 4 },
                { 0, 0, 0, 0, 0, 0, 0, 0, 6 },
                { 0, 0, 1, 0, 0, 2, 0, 0, 0 }
            };

            Console.WriteLine("Solving Puzzle from start state:");
            PrintSudoku(startState);
            Console.WriteLine("");

            SolveSudoku(startState);

            Console.ReadKey();
        }

        public static int[,]? SolveSudoku(int[,] startState)
        {
            if (SolveSudokuState(startState, 0, 0))
            {
                Console.WriteLine("Puzzle Solved!");
                PrintSudoku(startState);
                return startState;
            } else
            {
                Console.WriteLine("Puzzle cannot be solved :(");
                return null;
            }
        }

        public static void PrintSudoku(int[,] sudokuState)
        {
            for (int i = 0; i < sudokuState.GetLength(1); i++)
            {
                int[] row = Enumerable.Range(0, sudokuState.GetLength(1))
                    .Select(x => sudokuState[i, x])
                    .ToArray();

                Console.WriteLine(string.Join(", ", row));
            }
        }
```

```
    private static int N = 9;
    private static bool SolveSudokuState(int[,] sudokuState, int row, int col)
    {
        if (row == N - 1 && col == N) {
            return true;
        }

        if (col == N)
        {
            return SolveSudokuState(sudokuState, row + 1, 0);
        }

        if (sudokuState[row, col] != 0)
        {
            return SolveSudokuState(sudokuState, row, col + 1);
        }

        for (int i = 1; i <= N; i++)
        {
            if (IsValidEntry(sudokuState, row, col, i))
            {
                sudokuState[row, col] = i;
                if (SolveSudokuState(sudokuState, row, col + 1))
                {
                    return true;
                }
            }
            sudokuState[row, col] = 0;
        }

        return false;
    }

    private static bool IsValidEntry(int[,] sudokuState, int row, int col, int entry)
    {
        int boxColStart = col / 3 * 3;
        int boxRowStart = row / 3 * 3;
        for (int i = 0; i < N; i++)
        {
            // heuristic check 1: Same row
            if (sudokuState[row, i] == entry)
                return false;

            // heuristic check 2: Same column
            if (sudokuState[i, col] == entry)
                return false;

            // heuristic check 3: Same 3x3 grid
            if (sudokuState[boxRowStart + (i / 3), boxColStart + (i % 3)] == entry)
                return false;
        }
        return true;
    }
}
```

```
Solving Puzzle from start state:
5, 1, 0, 0, 7, 0, 0, 0, 0
0, 0, 0, 0, 0, 0, 8, 0, 0
0, 4, 0, 2, 0, 0, 0, 9, 1
0, 0, 0, 3, 0, 0, 6, 0, 0
0, 8, 0, 0, 0, 0, 2, 0, 0
4, 0, 0, 0, 0, 5, 0, 8, 3
0, 9, 0, 1, 0, 0, 0, 3, 4
0, 0, 0, 0, 0, 0, 0, 0, 6
0, 0, 1, 0, 0, 2, 0, 0, 0

Puzzle Solved!
5, 1, 3, 8, 7, 9, 4, 6, 2
9, 6, 2, 4, 3, 1, 8, 5, 7
8, 4, 7, 2, 5, 6, 3, 9, 1
1, 7, 5, 3, 2, 8, 6, 4, 9
3, 8, 6, 7, 9, 4, 2, 1, 5
4, 2, 9, 6, 1, 5, 7, 8, 3
2, 9, 8, 1, 6, 7, 5, 3, 4
7, 5, 4, 9, 8, 3, 1, 2, 6
6, 3, 1, 5, 4, 2, 9, 7, 8
```