Project 2 Report

Paul Chon

October 2020

Note:

I mainly used Intellij to code, but I used Eclipse for the JUnit testing and exporting the project.

1 Brute Force

1.1 OBJECTIVE:

The objective of the Brute Force Algorithm is to implement an O(n^2) algorithm that can find the maximum subarray by looking at all possible pairs of arriving and departing dates and tries to figure out the arriving and departing date that gives the maximum sum.

1.2 IMPLEMENTATION:

I implemented this algorithm by using a nested for loop where the outer loop picks the arriving date and the inner loop picks the departing date, which should be greater than or equal to the arriving date. The inner loop picks the departing date by adding each date after the arriving date and comparing it with the overall max sum. So, the arrive date is the index of the outer loop and the departing date is the index of the inner loop, but are only set when a max sum is found. The method for finding this max sum is called findMaximumSubArray() seen in Figure 1 and the class is called BruteForceAlgorithm.

```java
public MaxSubArrayResults findMaximumSubArray() {
    int maxSum = 0;
    int sum = 0;
    int arrive = 0;
    int depart = -1;

    for (int i = 0; i < data.length; i++) {
        sum = 0;
        for (int j = i; j < data.length; j++) {
            sum += data[j];
            if (sum > maxSum) {
                maxSum = sum;
                arrive = i;
                depart = j;
            }
        }
    }

    return new MaxSubArrayResults(maxSum, arrive, depart);
}
```

Figure 1: The implementation of the Brute Force Algorithm

1.3 TESTING:

I tested this algorithm by testing 10 sets of 100 numbers that was given in the maxSumtest.txt file, testing 2 sets of 8 numbers that was given in the instruction for the project, testing a random array of a random size using nextInt() from the Random class, and testing the running time with input sizes of: 100, 200, 500, and 1000. The input size for this algorithm only goes up to 1000 because any more is too slow. Also, for random testing for the Brute Force Algorithm, I only went up to 1000 for the same reason.

2 Divide and Conquer

2.1 OBJECTIVE:

The objective of the Divide and Conquer Algorithm is to implement a O(n lg n) algorithm that can find the maximum subarray by using a Divide and Conquer approach, which is to divide the array into two halves and then find the maximum of three sums: the left half of the array, the right half of the array, and the subarray that can cross the midpoint.

2.2 IMPLEMENTATION:

I implemented this algorithm by having a base case where, if the index of the first number and last number are the same, then return the number at the index of the first number as the sum and the index of the first number and last as the arriving and departing date respectively. Then, I calculate the midpoint and find the left, right and crossing sum. The left and right sum just recursively call the same method seen in Figure 2, but the crossing sum calls a separate method seen in Figure 3. The crossing sum is calculated by adding the left sum and the right sum.

```
public MaxSubArrayResults findMaxCrossingSubArray(int low, int mid, int high) {
    int leftSum = Integer.MIN_VALUE;
    int sum = 0;
    int maxLeft = 0;

    for (int i = mid; i >= low; i--) {
        sum += data[i];
        if (sum > leftSum) {
            leftSum = sum;
            maxLeft = i;
        }
    }

    int rightSum = Integer.MIN_VALUE;
    sum = 0;
    int maxRight = 0;

    for (int i = mid + 1; i <= high; i++) {
        sum += data[i];
        if (sum > rightSum) {
            rightSum = sum;
            maxRight = i;
        }
    }

    return new MaxSubArrayResults( maxSum: leftSum + rightSum, maxLeft, maxRight);
}
```

Figure 2: Main implementation of the Divide and Conquer Algorithm that will find left and right sum and also find the maximum subarray

```java
public MaxSubArrayResults findMaximumSubArray(int low, int high) {
    if (high == low) {
        return new MaxSubArrayResults(data[low], low, high);
    }

    int mid = (low + high) / 2;
    MaxSubArrayResults left = findMaximumSubArray(low, mid);
    MaxSubArrayResults right = findMaximumSubArray( low: mid + 1, high);
    MaxSubArrayResults crossing = findMaxCrossingSubArray(low, mid, high);

    if (left.getMaxSum() < 0 && right.getMaxSum() < 0 && crossing.getMaxSum() < 0) {
        return new MaxSubArrayResults( maxSum: 0,  arrive: 0,  depart: -1);
    }

    if (left.getMaxSum() >= right.getMaxSum() && left.getMaxSum() >= crossing.getMaxSum()) {
        return left;
    } else if (right.getMaxSum() >= left.getMaxSum() && right.getMaxSum() >= crossing.getMaxSum()) {
        return right;
    } else {
        return crossing;
    }
}
```

Figure 3: Implementation of the Divide and Conquer Algorithm that will find crossing sum

2.3 TESTING:

I tested this algorithm by testing 10 sets of 100 numbers that was given in the maxSumtest.txt file, testing 2 sets of 8 numbers that was given in the instruction for the project, testing a random array of a random size using nextInt() from the Random class, and testing the running time with input sizes of: 100, 200, 500, 1000, 2000, 5000, and 10000.

3 Dynamic Programming: Kadane's Algorithm

3.1 OBJECTIVE:

The objective of Kadane's Algorithm is to implement a O(n) algorithm that can find the maximum subarray by add one element to a temporary variable and then at the end keep the max sum.

3.2 IMPLEMENTATION:

I implemented this algorithm by creating a temporary variable called sum and I used a for loop that will add each element of the array to sum and there are two if statements: the first checks to see if sum is less than 0 and if it is, then the sum is set to 0 and the arrive date is set to the index of the current element plus 1; the second checks to see if the sum is greater than the current maxSum and if it is, then the maxSum is set to the sum and the depart date is set to

the index of the current element. In the second check there is also a tempArrive variable that is used to correct the arrive variable to the right number. The implementation can be seen in Figure 4

```
public MaxSubArrayResults findMaximumSubArray() {
    int maxSum = 0;
    int sum = 0;
    int arrive = -1;
    int depart = -1;
    int tempArrive = 0;

    for (int i = 0; i < data.length; i++) {
        sum += data[i];

        if (sum < 0) {
            sum = 0;
            arrive = i + 1;
        }

        if (maxSum < sum) {
            maxSum = sum;
            depart = i;
            tempArrive = arrive;
        }
    }

    arrive = tempArrive;
    return new MaxSubArrayResults(maxSum, arrive, depart);
}
```

Figure 4: The implementation of the Dynamic Programming Algorithm or Kadane's Algorithm

3.3 TESTING:

I tested this algorithm by testing 10 sets of 100 numbers that was given in the maxSumtest.txt file, testing 2 sets of 8 numbers that was given in the instruction for the project, testing a random array of a random size using nextInt() from the Random class, and testing the running time with input sizes of: 100, 200, 500, 1000, 2000, 5000, and 10000.

4 TIMING

I used nanoTime() rather than currentTimeMillis(), so I got much larger numbers rather than just using currentTimeMillis(). The average of 10 runs of different input sizes for each algorithm can be seen in Figure 5, Figure 6, and Figure 7. The linear plot is Figure 8 and the log-log plot is Figure 9. Also, for

the Brute Force Algorithm, I only went up to 1000 inputs because any more takes too long to calculate.

Input Size	Average (in ns)
100	4260
200	10680
500	56210
1000	363440

Figure 5: Data used to plot the times for the Brute Force Algorithm

Input Size	Average (in ns)
100	10510
200	13030
500	34200
1000	73920
2000	137020
5000	346100
10000	580160

Figure 6: Data used to plot the times for the Divide and Conquer Algorithm

Input Size	Average (in ns)
100	840
200	1770
500	3230
1000	6300
2000	10410
5000	24440
10000	44250

Figure 7: Data used to plot the times for the Dynamic Programming Algorithm or Kadane's Algorithm

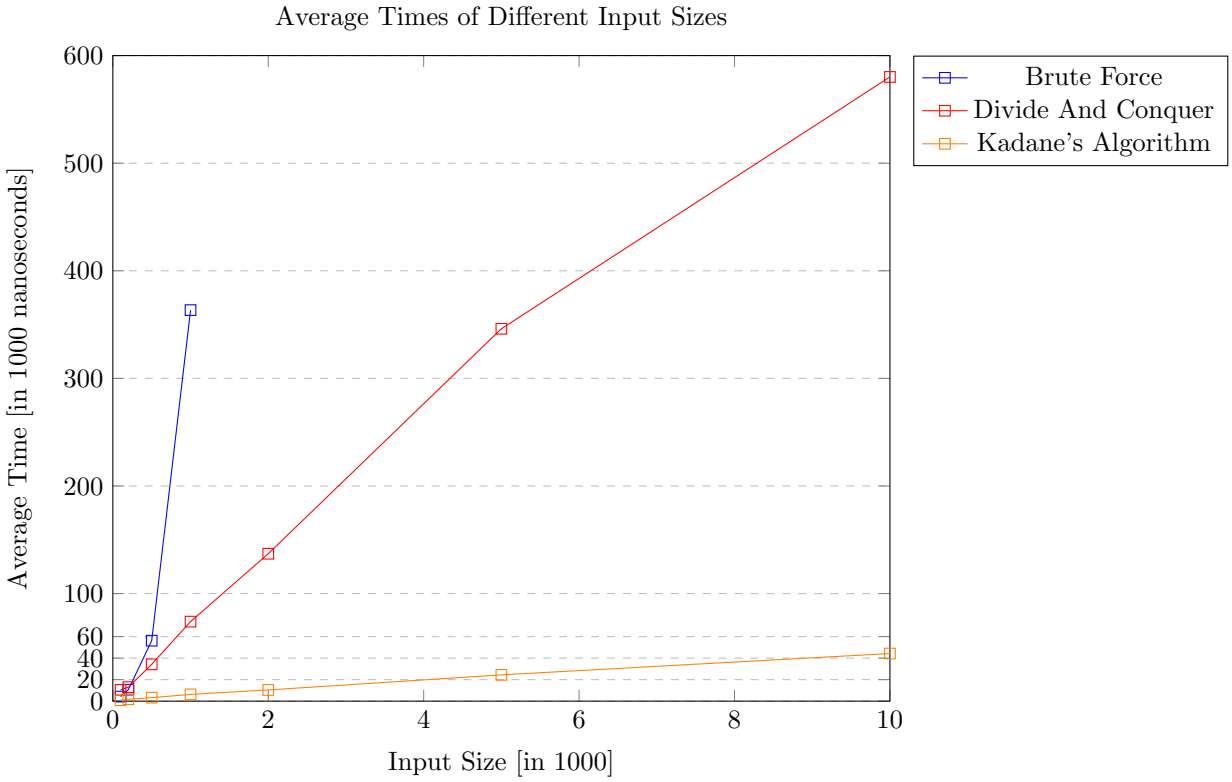

Figure 8: Linear plot of the average times of different input sizes

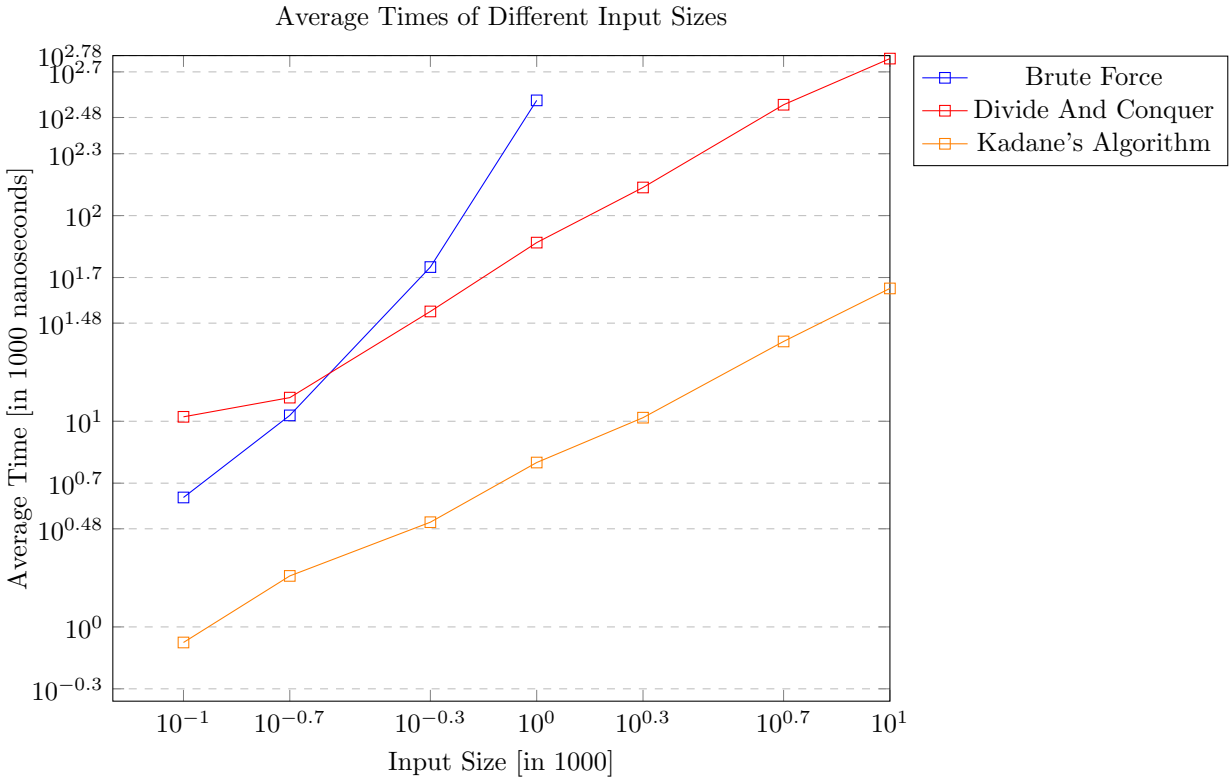

Figure 9: Log-log plot of the average times of different input sizes