# Project 1

## Theoretical Run-time Analysis

### Algorithm 1: Enumeration (max\_subarray\_enum)

array\_sum, array\_start, array\_end = 0 // O(1)

**for** i = 0 to array.length // array[0, 1, …, n-1, n] = O(n)

high\_sum = 0 // O(1)

start = i // O(1)

end = null // O(1)

**for** j = i to array.length // array[i, i+1, i+2, ..., n-1, n] = O(n)

current\_sum = 0 // O(1)

**for** k = i to j // array[i, i+1, …, j-1, j] = O(n)

current\_sum = current\_sum + array[k] // O(1)

**if** current\_sum > high\_sum // O(1)

high\_sum = current\_sum // O(1)

end = j // O(1)

**if** high\_sum > array\_sum // O(1)

array\_sum = high\_sum // O(1)

array\_start = start // O(1)

array\_end = end // O(1)

**return**(array\_sum, array\_start, array\_end) // O(1)

Algorithm 1 Run-time Analysis: O(n) \* O(n) \* O(n) (the first for loop, i, traverses the entire array; the second for loop, j, traverses i to the end of the array; and the third for loop, k, traverses i to j to compute the sum); O(n3)

### Algorithm 2: Better Enumeration (max\_subarray\_better\_enum)

array\_sum, array\_start, array\_end = 0 // O(1)

**for** i = 0 to array.length // array[0, 1, …, n-1, n] = O(n)

high\_sum, current\_sum = 0 // O(1)

start = i // O(1)

end = null // O(1)

**for** j = i to array.length // array[i, i+1, …, n-1, n] = O(n)

current\_sum = current\_sum + array[j] // O(1)

**if** current\_sum > high\_sum // O(1)

high\_sum = current\_sum // O(1)

end = j // O(1)

**if** high\_sum > array\_sum // O(1)

array\_sum = high\_sum // O(1)

array\_start = start // O(1)

array\_end = end // O(1)

**return**(array\_sum, array\_start, array\_end) // O(1)

Algorithm 2 Run-time Analysis: O(n) \* O(n) \* O(1) (the first for loop, i, traverses the entire array; the second for loop, j, traverses the array from i to the end; and this time, the summation is in constant time because we’re simply adding the array[j] element to the sum of array[i, i+1, …, j-1]); O(n2)

### Algorithm 3: Divide and Conquer (max\_subarray\_divide\_and\_conquer)

**if** start >= end // O(1)

**return** array[end], start, end // O(1)

**else**

array\_start, array\_end = 0 // O(1)

mid = (start + end) / 2 // O(1)

left = max\_subarray\_divide\_and\_conquer(array, start, mid) // O(lg n)

right = max\_subarray\_divide\_and\_conquer(array, mid + 1, end) // O(lg n)

cross = max\_suffix(array, start, mid) + max\_prefix(array, mid + 1, end) // O(n) + O(n)

**if** cross > left and cross > right // O(1)

**return** cross, cross\_start, cross\_end // O(1)

**else** **if** left >= right // O(1)

**return** left, left\_start, left\_end // O(1)

**else**

**return** right, right\_start, right\_end // O(1)

#### Algorithm 3a: max\_suffix

max\_sum, current\_sum = 0 // O(1)

array\_start = end // O(1)

**for** i = end downto start // O(n)

current\_sum = current\_sum + array[i] // O(1)

**if** current\_sum > max\_sum // O(1)

max\_sum = current\_sum // O(1)

array\_start = i // O(1)

**return** max\_sum, array\_start // O(1)

#### Algorithm 3b: max\_prefix

max\_sum, current\_sum = 0 // O(1)

array\_end = start // O(1)

**for** i = start to end // O(n)

current\_sum = current\_sum + array[i] // O(1)

**if** current\_sum > max\_sum // O(1)

max\_sum = current\_sum // O(1)

array\_end = i // O(1)

**return** max\_sum, array\_end // O(1)

Algorithm 3 Run-time Analysis: O(n) \* O(log n) (we’re breaking the problem into n/2 problems and then calling the function on each half until we reach the base case of 1 element in the array – because we are therefore checking every element in the array, we’re doing n elements amount of work; the depth of the recursive calls is log n, so we’ll be doing that n amount of work log n times); O(n log n)

As we can see, max\_prefix and max\_suffix both loop through the entire range of values they are given and are therefore O(n).

The recurrence equation for max\_subarray\_divide\_and\_conquer is T(n) = 2T(n/2) + 2n + C.

Using the master theorem, we have a = 2, b = 2, f(n) = 2n + C.

n(log\_2(2)) = n1, f(n) = Θ(n), therefore T(n) = Θ(n lg n).

### Algorithm 4: Linear-time (max\_subarray\_dynamic)

running\_total, array\_sum = array[0] // O(1)

array\_start, array\_end, start, end = 0 // O(1)

**for** i = 1 to array.length // array[2, 3, …, n-1, n] = O(n)

running\_total = running\_total + array[i] // O(1)

**if** running\_total < 0 // O(1)

running\_total = 0 // O(1)

start = i + 1 // O(1)

**if** running\_total > array\_sum // O(1)

array\_sum = running\_total // O(1)

array\_start = start // O(1)

array\_end = i // O(1)

**return** array\_sum, array\_start, array\_end // O(1)

Algorithm 4 Run-time Analysis: O(n) (we’re moving through every element in the array, so we’re doing n amount of work, but each time we’re performing only constant time operations such as addition, comparison, and assignment of variable values); O(n)

## Proof of Correctness for Algorithm 3: Divide and Conquer

max\_subarray\_divide\_and\_conquer(array, start, end)

**Precondition:** The array has at least 1 element

**Post-condition:** We return the value of the largest subarray within the array, and the indices of the start and end of that subarray.

max\_suffix(array, start, end)

**Precondition:** The array has at least 1 element

**Post-condition:** Provides the maximum subarray in array for array[start … n], where start is the first element in the array and is always included in the subarray.

max\_prefix(array, start, end)

**Precondition:** The array has at least 1 element

**Post-condition:** Provides the maximum subarray in array for array[n … end], where end is the last element in the array and is always included in the subarray.

**Base case:** When start >= end, n = 1 (it is trivially correct that an array of size 1 contains a maximum subarray of size 1 (itself)).

**Inductive hypothesis:** Assume that max\_subarray\_divide\_and\_conquer correctly returns the maximum subarray for n=1, 2, …, k elements, with that subarray appearing in the first half of the divided array, the second half of the divided array, or across both divided arrays.

**Inductive step:** Show that max\_subarray\_divide\_and\_conquer correctly returns the maximum subarray for k+1 elements

First recursive call: n1 = (k+1)/2-start+1 <= k, returns the maximum subarray of [start … (k+1)/2]

Second recursive call: n2 = end-(k+1)/2 <= k, returns the maximum subarray of [(k+1)/2 + 1 … end]

All values in the original n array are accounted for, and both subarrays are divided in half until start >= end, which will satisfy the base case and provide us with a maximum subarray. We must then compare that subarray provided by the first call to the subarray provided by the second call, and to the subarray formed by the combination of max\_prefix and max\_suffix, and return the largest of the three subarrays and start/end values.

Because max\_prefix always returns the maximum subarray that also includes the final element in its array, and max\_suffix always returns the maximum subarray that includes the first element in its array, the two subarrays are contiguous and form a single subarray ranging from some index n1 … end of prefix … beginning of suffix … n2.

**Prove the program terminates:**

For array of size n, we divide the array into two subarrays of size n/2. We then call the function on each of these halves of the original array with the first call on indices from start to floor(length/2), and the second call from floor(length/2) + 1 to end. The program returns (terminates) when the base case is satisfied and start >= end. In the first case, start will always be the 0 index of the new subarray and length will be halved until it reaches 0 (integer values), satisfying the condition that start >= end and the first half will terminate successfully. In the second case, length/2 will always equal at least 1 as length/2 approaches 0, and end will eventually equal 0 as we continue to halve each array into smaller subarrays. Therefore the second half will eventually result in 1 >= end when end is 0, which will successfully terminate that half of the call.

So the recursive portion will always terminate. To show that the remainder of the program terminates, consider that all cases are satisfied – if the result of the first call is larger than or equal to the result of the second call, the program will terminate. If not (in the case where the first call is smaller than the second call) the program will terminate. In the case where the cross value is larger than either the first or second call, the program terminates.

## Testing

We tested against the MSS\_TestProblems.txt files and compared the results to MSS\_TestResults.txt.

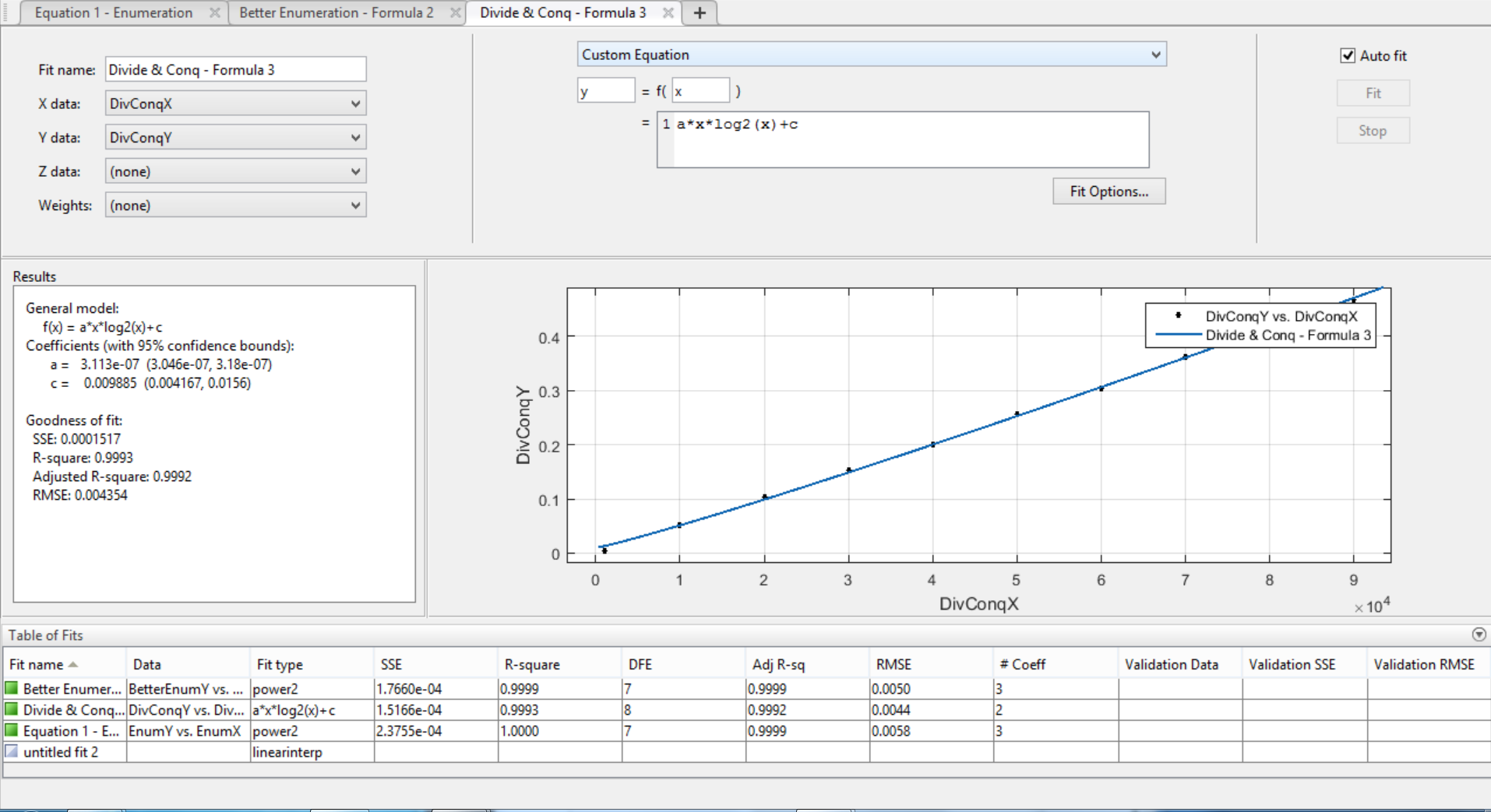
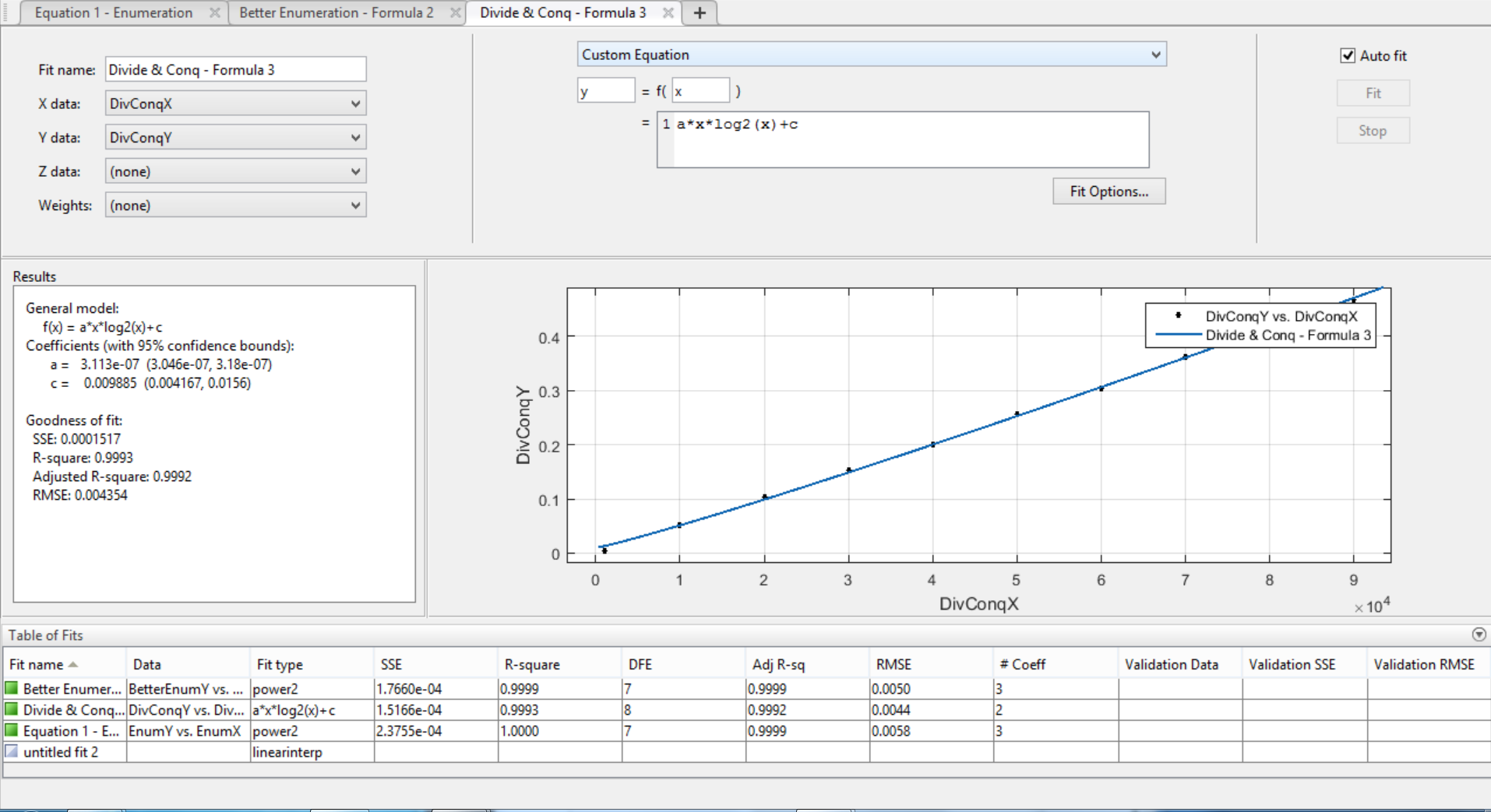
## Experimental Analysis

The average running times were calculated based on 10 runs of each n value. A sample is provided below.



### Running times as a function of input size n

### Algorithm 3: Divide and Conquer (Matlab)



One discrepancy is on equation 3. On paper the formula should be O(n\*log(n)) but the results are coming out very linear O(n). Apparently this is a result of the values for log(n) being dominated by the n portion. As a result, we don't get a very drastic change from a simple linear equation. Notice that the R-square value for the linear Excel trendline is higher than the R-square for the n log n model from Matlab.

### Largest input solvable in 10 minutes

When we solve for x using the formula from the Excel trendline equation, floor(x) for each algorithm when y = 600 is as follows. For Divide & Conquer, the Matlab equation was used because Excel trendlines do not support n log n equations.

* Enumeration: 5,836
* Better Enumeration: 86,596
* Divide & Conquer: 73,743,800
* Linear: 3,000,001,500