David Rigert

Isaiah Perrotte-Fentress

Adam McDaniel

Group 1

# Project 1

## Theoretical Run-time Analysis:

Algorithm 1: Enumeration (max\_subarray\_enum)

array\_sum, array\_start, array\_end = 0

**for** i = 0 to array.length

high\_sum = 0

start = i

end = null

**for** j = 0 to array.length

current\_sum = sum(array[i], array[j+1])

**if** current\_sum > high\_sum

high\_sum = current\_sum

end = j

**if** high\_sum > array\_sum

array\_sum = high\_sum

array\_start = start

array\_end = end

**return**(array\_sum, array\_start, array\_end)

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 the entire array; and the summation moves through the array and computes the sum on each element); O(n3)

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

array\_sum, array\_start, array\_end = 0

**for** i = 0 to array.length

high\_sum = 0

current\_sum = 0

start = i

end = null

**for** j = i to array.length

current\_sum = current\_sum + array[j]

**if** current\_sum > high\_sum

high\_sum = current\_sum

end = j

**if** high\_sum > array\_sum

array\_sum = high\_sum

array\_start = start

array\_end = end

**return**(array\_sum, array\_start, array\_end)

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 our running total); O(n2)

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

**if** start >= end

**return** array[end], start, end

**else**

array\_start, array\_end = 0

mid = (start + end) / 2

left = max\_subarray\_divide\_and\_conquer(array, start, mid)

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

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

**if** cross > left and cross > right

**return** cross, cross\_start, cross\_end

**else** **if** left >= right

**return** left, left\_start, left\_end

**else**

**return** right, right\_start, right\_end

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)

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

running\_total, array\_sum = array[0]

array\_start, array\_end, start, end = 0

**for** i = 1 to array.length

running\_total = running\_total + array[i]

**if** running\_total < 0

running\_total = 0

start = i + 1

**if** running\_total > array\_sum

array\_sum = running\_total

array\_start = start

array\_end = i

**return** array\_sum, array\_start, array\_end

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:

## Testing:

## Experimental Analysis: