**Analysis of Algorithms**

**Assignment 2**

**Ryan Farber, Zach Singleton, Huarui Wu**

**This dynamic programming algorithm is based on the following idea:**

*The maximal-sum subarray either uses the last element in the input array, or it doesn’t.*

**The tasks are:**

**Recursive function:** Describe the solution to the maximal-sum subarray problem recursively and mathematically based on the above idea.

struct maximumS {

int atPresent; // Current sum of integers

int maximum; // Overall maximum of integers

}

struct maximumS maximumSubarray (int array[ ], int size) {

struct maximumS ms;

if(size == 1) { // This is the base case

ms.atPresent = 0;

ms. maximum = array [0];

} else { // This is a recursion on the array excluding the last element

ms = maximumSubarray (array, size 1);

}

//The following code finds the maximum

ms.atPresent += array[ size 1];

ms.atPresent = (ms.atPresent > 0) ? ms.atPresent: 0 ;

ms.maximum = (ms.atPresent> ms.maximum) ? ms.atPresent: ms.maximum;

return ms;

}

Mathematical description:

* maximumS is a struct holding the running sum and the overall maximum sum
* MaximumSubarray is the recursive function
* Thus, for an array A of size n, we find the maximum subarray with MaximumSubarray(A, n).

**Pseudocode:** Give pseudocode for a dynamic programming algorithm based on this function. Your implementation should create a dynamic programming table.

maximumSubarray(array, size):

current = 0;

maximum = 0;

index = 0;

while index < size:

current = current + array[index-1]

if current < 0:

current = 0

else if current > maximum:

maximum = current

index = index + 1

return maximum

**Running time:** Analyze the time complexity of your algorithm.

The algorithm’s runtime should be Θ(n), because the code walks through each element of the input array only once.

The graph of Algorithm 4 (included at the end of this project) shows the execution time of this algorithm versus the size of the input array. The slope of Algorithm 4 is approximately 7984, which confirms the runtime of Θ(n).

Also, experimental data used for the graph of Algorithm 4 supports our theory that this graph is linear.

**Theoretical correctness:** Write a formal proof of correctness for your algorithm using induction.

**Claim:** The sum of the maximum subarray of an array A of size n will be returned by maximumSubarray(A, n).

**Proof:** Base Case - Consider n = 0. Then maximum = current = 0, which is correct.

* Inductive Hypothesis: Assume that maximumSubarray(A, n) will correctly return the sum of the maximum subarray of an array A of size n.
* We must show that maximumSubarray(A,n+1) will do the same for array A of size n+1.

**There are two cases to consider:**

***Case A:*** A[n] 0. Since we are adding each element of the array to the current running sum on line 7, current will correctly increase and be correctly captured as the maximum sum on line 11.

***Case B:*** A[n] < maximumSubarray(A, n). This drives the running sum negative. Thus, current will correctly “reset" to zero on line 9, effectively ignoring the sum up to element n.

However, because maximum retains its value, the overall maximum sum is correctly preserved.

**Implement:** Implement your algorithm and include the relevant section of code in your project report. This should not be lengthy. Your implementation should only return the value (sum) of the maximal-sum subarray and not the indices bounding the maximal-sum subarray. I’m thinking 20 lines maximum.

Javascript:

**function maxSumArray(data) {**

**var maxSum = 0;**

**var sumNow = 0;**

**var i;**

**for (i=0; i < data.length; i++) {**

**sumNow = sumNow + data[i];**

**if (sumNow < 0) {**

**sumNow = 0;**

**}**

**else if (maxSum < sumNow) {**

**maxSum = sumNow;**

**}**

**}**

**return maxSum;**

**}**

**Test:** You should make sure your program is correct, although you will not be submitting evidence (beyond the pseudocode and code above).

Tested with the datasets from Project 1. All tests were successful.

Also tested with all negative numbers. Tests were successful as well.

**Compare:** Perform tests to compare this dynamic programming algorithm to the divide & conquer algorithm of the last project. You are in charge of deciding what tests would be reasonable. Discuss the comparative benefits and drawbacks of these two algorithms in detail in your report

Divide and conquer algorithm - works by recursively breaking down a problem into two or more sub-problems of the same (or related) type, until these become simple enough to be solved directly. Then once broken down, the algorithm conquers each sub-problem recursively and combine them

Dynamic programming algorithm - a technique for solving problems with overlapping subproblems. Each sub-problem is solved only once, (unlike the D/C,) and the result of each sub-problem is stored in a table ( generally implemented as an array or a hash table) for future references.

Divide and conquer can solve difficult questions like tower of hanoi, provides a way to design efficient algorithms, offer great parallelism with multi-processor machines, and memory access mechanisms due to cache. Its disadvantage is how many times it will have to do recursion based on whether or not the subproblems are dependent or not.

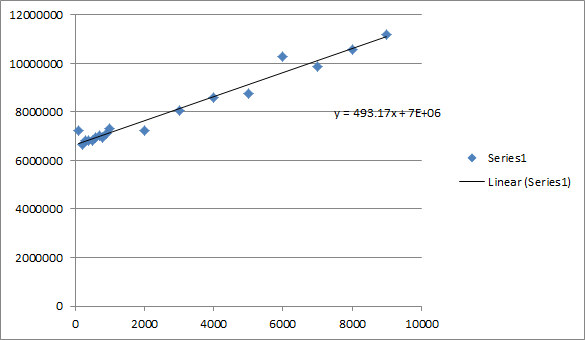
Divide and conquer works best when all problems are independent, and we know where to partition

Dynamic programming’s biggest weakness is it’s curse to dimensionality. Runtime is strongly dependent on the range of state variable. It also has a huge time and space complexity to it.

Dynamic programming works best when all problems are dependent, where we don’t know where to partition

Please Note: You may view the graphs that are part of this response on the next page.

Algorithm 3



Algorithm 4

