Group Members: Barbara Hazlett, Wing Lee Zerlina Yeung, Ethan Spiro
CS 325:Project 2 - Max Sub Array with dynamic programming
Language: C++

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

Idea above “The maximal-sum subarray either uses the last element in the input array, or it doesn’t”

Say we have an array of integers of size n, A = [A1, …., An] that contains positive and negative numbers and we want to find the maximum value contiguous subsequence. We want MSA(ai...aj) where 1 <= i <= j <= n. Suppose we have an MSA(j) which is the maximum sum over all windows, ending at the jth element.

We have the recursive function MSA(j) = max{MSA(j -1) + A[j], A[j] }.

To find the optimal window ending at position j, we need to extend the optimal window ending at position j-1, or we won’t extend anything and start a brand new window consisting only a[j]. We need to either extend the window or start a new window, hence only 2 options to consider.

If MSA(j - 1) will be included in the window with A(i), but if MSA(j - 1) + A[j] gives us a smaller sum than just A[j], then we will start a new window. If MSA(j - 1) + A[j] gives us a bigger sum than just A[j], we will extend the window to include A[j].

We keep going until we reach the last element of the input array. At that point we have again our two options to consider, whether adding that last element will give us a larger subsequence or not. If it does, we extend the window, if not we start a new window. This concurs with the idea, the max subarray either uses the last element in the input array or it doesn’t. If it uses the last element then the last element is part of the max sub array, if it doesn’t use it, it means the last element is bigger than the sub sequence before it, and thus the last element would become the first element in the max sub array sequence.

To show this in mathematical way I have this proof:

For an array of integers of size n A = [A1, …, An], let MSA(j) be the maximum subarray ending in Aj. We want to to prove that MSA(j+1) will give us the maximum subarray ending in MSA(j+1)

Base case:
We can prove trivially that when MSA(1), since there is only one element in the array MSA(1) = A1.

Recursive case:

Adding Aj+1 to MSA(j) will give us two possibilities.

1) First case, after adding Aj+1 to MSA(j), MSA(j) gives us a greater value than when just considering Aj+1. In this case Aj+1 must be part of this larger window or sub array preceding itself. Thus MSA(j+1) is the sum of Aj + 1 and MSA(j).

2) Second case, after adding Aj + 1 to MSA(j), we get a lesser value than when just considering Aj+1. Thus Aj+1 must be larger than the sub sequence preceding itself. Thus MSA(j + 1) is Aj+1.

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

```
Declare a single dimension array (size of search array)
Initialize table[0] to first element of search array
for i = 1 … end of array[]
     DP table[i] = max of (array[i] OR array[i] + DP table[i - 1]
end for
Declare an int max (set equal to 0 temporarily)
for i = 0 … end of DP table
     if (DP table > max)
          max = DP table[i];
     end if
end for
return max
```

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

The time complexity of this algorithm is O(n). It has two loops that are O(n) because one traverses the test array and one traverses the DP table. However, they are not nested so they are not multiplied together to get the time complexity.

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

Claim: Our function alogrithm_dp(int *arr, int sizeOfArray) returns the maximum sum of contiguous array elements, as our table[n] contains this maximum sum.

Base Case: When n = 0, A0 is trivially the maximum subarray as table[0] = a[0].

Inductive hypothesis: Assume for any value k, using algorithm_dp(int *arr, int k) will produce a table[k] that contains the maximum subarray in a[k] terminating in an element Aj. We must prove that alogrithm_dp(int *arr, int k + 1) will also produce a table[k + 1] that contains the maximum subarray a[k+1] terminating in an element Aj.

Inductive step: Using the inductive hypothesis, we want to prove that algorithm_dp(int *arr, int k + 1) returns the sum of the maximum array.

We have two cases to consider.
1) Window extension: If table[k] + a[k + 1] > a[k + 1], then A[k + 1] must be part of the maximum subarray terminating in Aj, so table [k + 1] = table[k] + a[k + 1].

2) New window: If table[k] + a[k + 1] < a[k + 1], then Aj+1 must be be small enough to cause table[k] + a[k + 1] to have smaller value than a[k + 1]. Thus we begin a new window with table[k + 1] = a[k + 1].

Since we use our variable max to store the maximum sum in table[k], our algorithm must return the maximum sum. Our proof is complete.

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 MAX.*

```
int algorithm_DP(int *arr, int sizeOfArray) {

        int dp_table[sizeOfArray];
        dp_table[0] = arr[0];

        for(int i = 1; i < sizeOfArray; i++) {
                dp_table[i] = max(arr[i], arr[i] + dp_table[i-1]);
        }

        int max = 0;
```

```
        for(int i = 0; i < sizeOfArray; i++){
                if(dp_table[i] > max) {
                max = dp_table[i];
                }
        }

        return max;
}
```

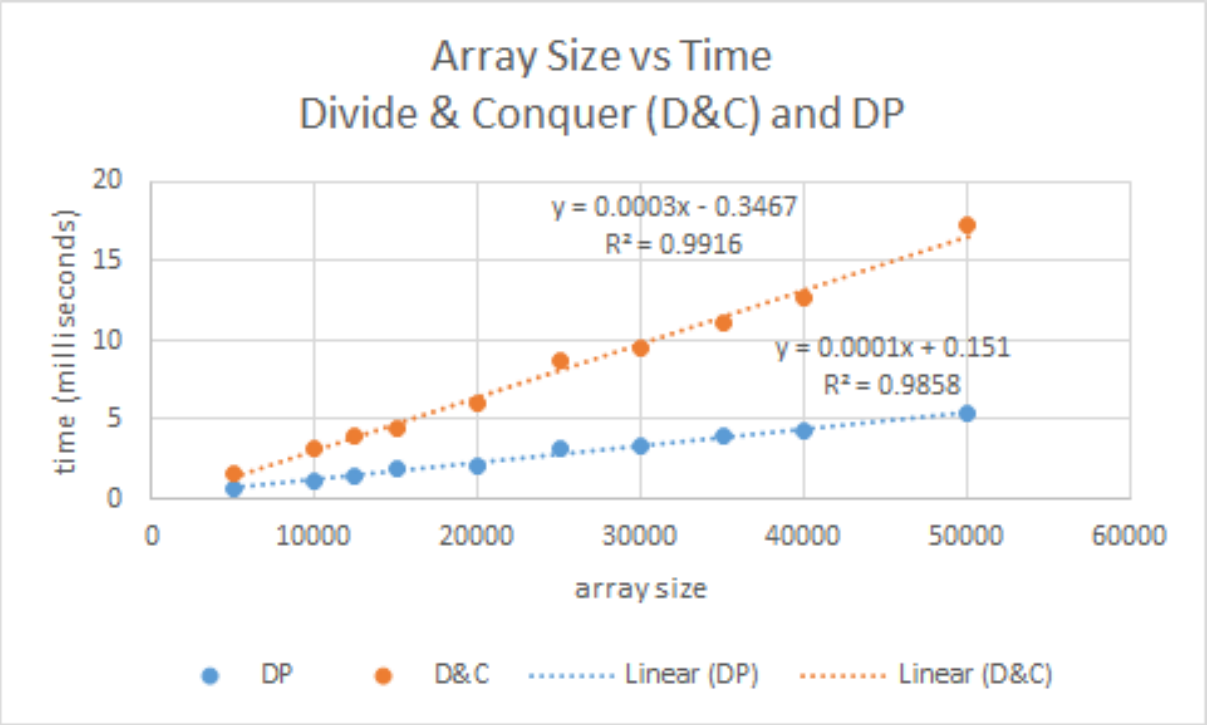
Test: *You should make sure your program is correct, although you will not be submitting evidence (beyond the pseudocode and code above).*

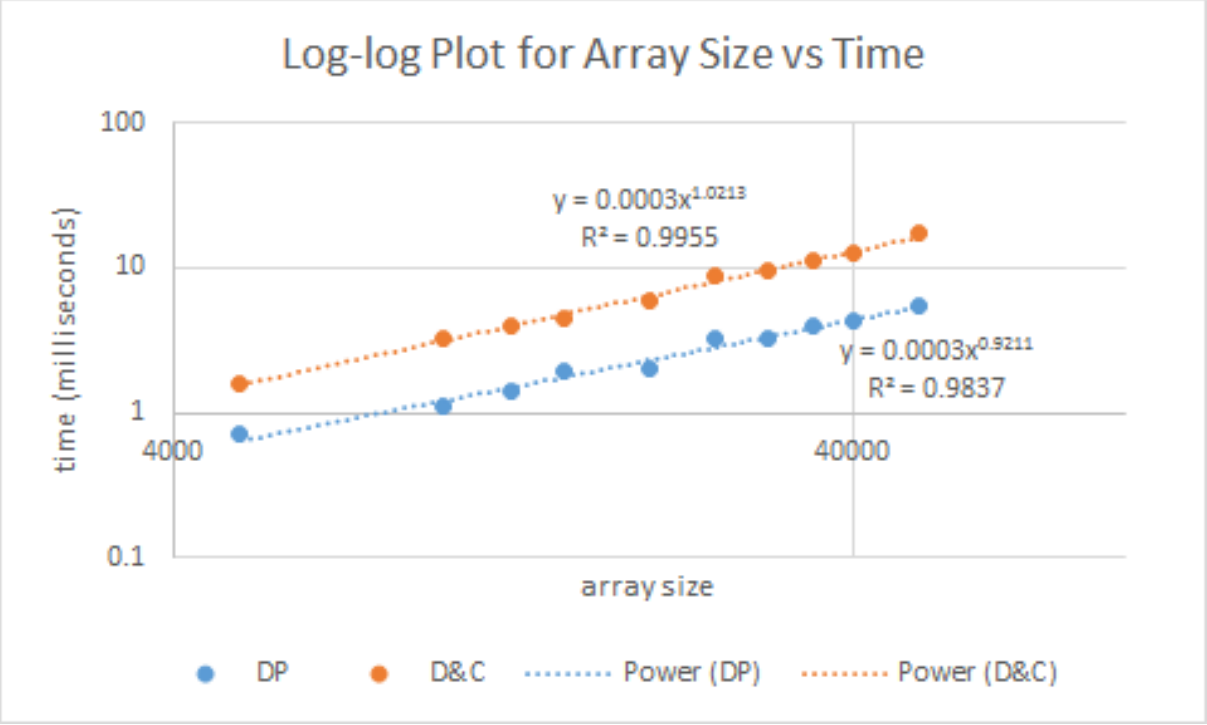
We tested the code with all of the test arrays provided for project 1. In addition we verified that an array of all negative numbers returned a max subarray of 0. All tests passed.

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.*

We revised the code from project 1 and timed the two algorithms with arrays of size:
5000, 10000, 12500, 15000, 20000, 25000, 30000, 35000, 40000 and 50000

We ran these tests on a Dell Inspiron laptop with an i7 processor and used Excel to plot the data. Below are normal and log-log plots of array size versus runtime:





Experimental versus theoretical run times:

Looking at the graphs it is clear that the data is noisy, probably due to the fact that both algorithms run so quickly with the test array sizes that we chose. From the best fit lines in the log-log plot we can see that the slope for the D&C algorithm is 1.0213. For a nlogn plot using the same input array sizes we expect the slope to be 1.1033. From the best fit line for the DP algorithm the slope is .9211 and we expect it to be 1.0. Both algorithms are close to the theoretical and we would expect them to be even closer with larger size arrays but it was not feasible to test huge arrays due to system constraints.

Speed comparison:

From the graphs we can see the speed start to affect run time at the higher numbers, although both are fast.

We can calculate the biggest instance that each algorithm can solve in an hour using the best fit curve for each from the log-log plot. This will give us a better idea of the speed differences between the two algorithms.

Divide and Conquer:
trendline equation: y = .0003x^1.0213
let y = 3600000 milliseconds/hour = .0003x^1.0213
3600000/.0003 = x^1.0213
1.20E10 = x^1.0213
take log of both sides: log(3.6E11) = log(x^1.0213)
10.07918 = 1.0213logx
logx = 9.86897
x = 10^9.86897 = 7,395,578,300
The biggest instance for algorithm 1 is an array of ~7,395,578,300 values.

Dynamic Programming:
The best fit equation is y = .0003x^.9211. Using the same technique as above, the biggest instance is an array of ~87,608,916,150 values. This is over ten times the biggest instance for the D&C algorithm.

~~~Dynamic Programming~~~

Benefits: Easier to understand, less code, faster. Per our calculations, in one hour we can process arrays over ten times as large. As as the arrays or processing time get larger this difference will continue to increase.
Drawbacks: Have to maintain an auxiliary table which takes up extra space.

~~~Divide and Conquer~~~

Benefits: Don't have to maintain a table, some consider recursion elegant.

Drawbacks: Longer code, we found it harder to understand, slower.