

# Assignment 2: Dynamic Programming project

Francis Vo, Soo-Hyun Yoo

October 30, 2012

## 1 Recursive function

```
1 struct maxS {
2     int current;    // Current sum
3     int max;        // Overall max
4 }
5
6 struct maxS MaxSubarray(int array[], int size) {
7     struct maxS ms;
8
9     if (size == 1) {    // Base case
10        ms.current = 0;
11        ms.max = array[0];
12    } else {    // Recurse on array excluding last element.
13        ms = MaxSubarray(array, size-1);
14    }
15
16    // Find maximum.
17    ms.current += array[size-1];
18    ms.current = (ms.current > 0) ? ms.current : 0;
19    ms.max = (ms.current > ms.max) ? ms.current : 0;
20
21    return ms;
22 }
```

rec.cpp

Where `maxS` is a struct holding the running sum and the overall maximum sum and `MaxSubarray` is the recursive function. For an array  $A$  of size  $n$ , we find the maximum subarray with `MaxSubarray(A, n)`.

## 2 Pseudocode

```
1 MaxSubarray(array, size):
2     current = 0;
3     max = 0;
4     index = 0;
5
6     while index < size:
7         current = current + array[index-1]
8         if current < 0:
9             current = 0
10        else if current > max:
11            max = current
12            index = index + 1
13
14    return max
```

### 3 Running time

The code shows that we look at each element of the input array only once, so the algorithm's runtime should be  $\Theta(n)$ .

Figure 1 shows the execution time of this algorithm versus the size of the input array. The slope of the graph is 0.987956, which confirms the runtime of  $\Theta(n)$ .

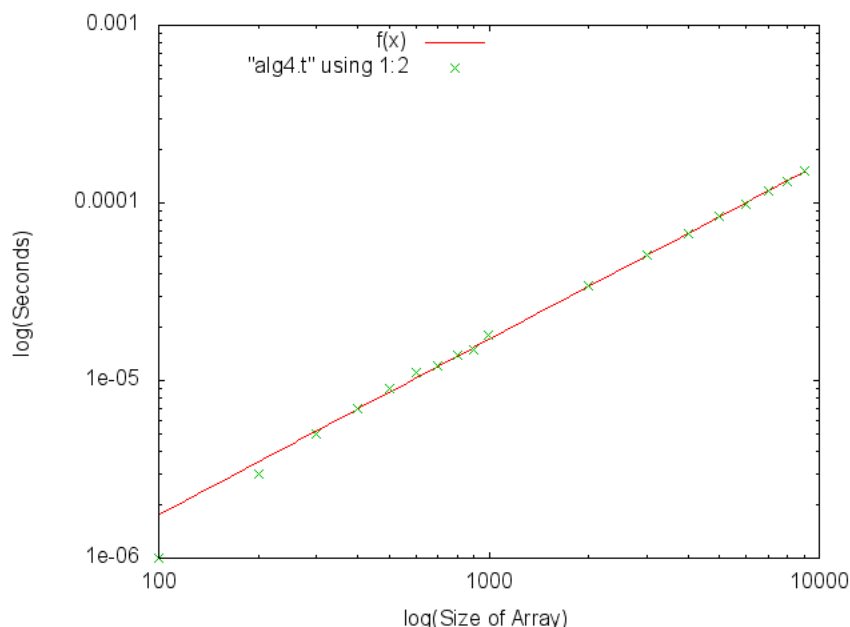


Figure 1: Algorithm 4 – execution time vs. input size

### 4 Theoretical correctness

**Claim:**  $\text{MaxSubarray}(A, n)$  will return the sum of the maximum subarray of an array  $A$  of size  $n$ .

**Proof:** As a base case, consider  $n = 0$ . Then  $\text{max} = \text{current} = 0$ , which is correct.

For the inductive hypothesis, assume that  $\text{MaxSubarray}(A, n)$  will correctly return the sum of the maximum subarray of an array  $A$  of size  $n$ . We must show that  $\text{MaxSubarray}(A, n+1)$  will do the same for an array  $A$  of size  $n + 1$ .

We have two cases to consider:

*Case 1:*  $A[n] \geq 0$ . 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 2:*  $A[n] < -\text{MaxSubarray}(A, n)$ . This will drive the running sum into the negative, so **current** will correctly “reset” to zero on line 9, effectively ignoring the sum up to element  $n$ . On the other hand, **max** retains its value, so the overall maximum sum is correctly preserved.

□

## 5 Implement

### 5.1 Algorithm 4

```
1  /*
2  * Enumeration
3  * Loop over each pair of indices i; j and compute the sum from k=i to j of a[k].
4  * Keep the best sum you have found so far.
5  */
6
7  int MaxSubarray(int array[], int size){
8      int current = 0;
9      int max = 0;
10     int i;
11
12     for (i=0; i<size; i++) {
13         current += array[i];
14         if (current <= 0) {
15             current = 0;
16         } else if (current > max) {
17             max = current;
18         }
19     }
20
21     return max;
22 }
```

alg4.cpp

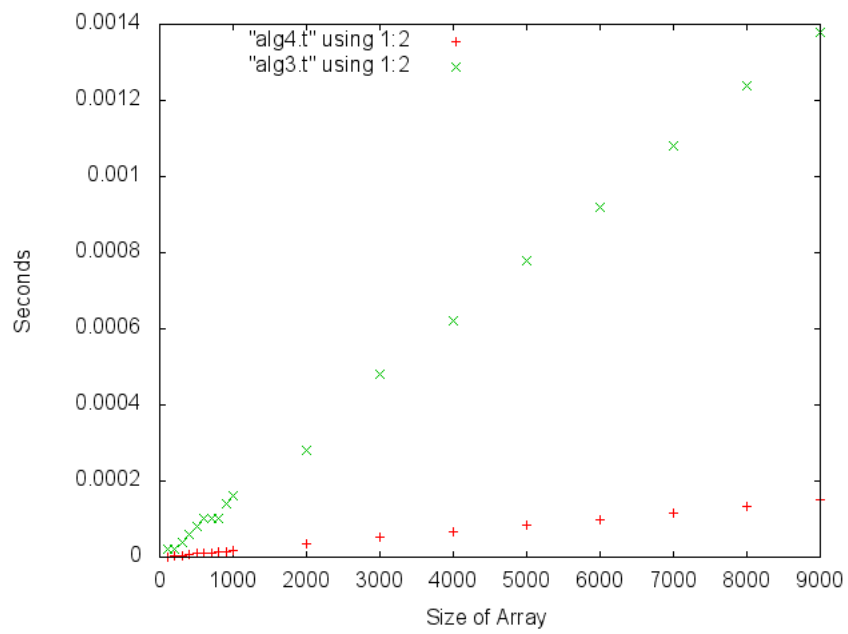
## 6 Test

Tests were run on the `ms_test.txt` file given for the last project and large arrays given by student IDs.

## 7 Compare

The two algorithms, one using divide-and-conquer and the other dynamic programming, were each tested using [TODO: How exactly was this tested?]. The results of the tests is shown in Figure 2.

The dynamic programming algorithm ( $\Theta(n)$  time) is shown to be much more efficient than the divide-and-conquer algorithm ( $O(n \log n)$  time) for a randomly sorted array. Additionally, since the DP algorithm does not have any recursive calls and needs to keep track of only two numbers (`current` and `max`), it is much less memory-intensive compared to the D&C algorithm.



**Figure 2:** Comparison of execution times between the divide & conquer and dynamic programming algorithms.