Assignment 2: Dynamic Programming project

Francis Vo, Soo-Hyun Yoo

October 30, 2012

1 Recursive function

```
struct maxS {
  int current;    // Current sum
  int max;        // Overall max
}

struct maxS MaxSubarray(int array[], int size) {
  struct maxS ms;

  if (size == 1) {    // Base case
    ms.current = 0;
    ms.max = array[0];
  } else {    // Recurse on array excluding last element.
    ms = MaxSubarray(array, size-1);
  }

  // Find maximum.
  ms.current += array[size-1];
  ms.current = (ms.current > 0) ? ms.current : 0;
  ms.max = (ms.current > ms.max) ? ms.current : 0;

  return ms;
}
```

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

```
MaxSubarray(array, size):
    current = 0;
    max = 0;
    index = 0;

    while index < size:
        current = current + array[index-1]
        if current < 0:
            current = 0
        else if current > max:
            max = current
        index = index + 1

    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 $\Theta(n)$.

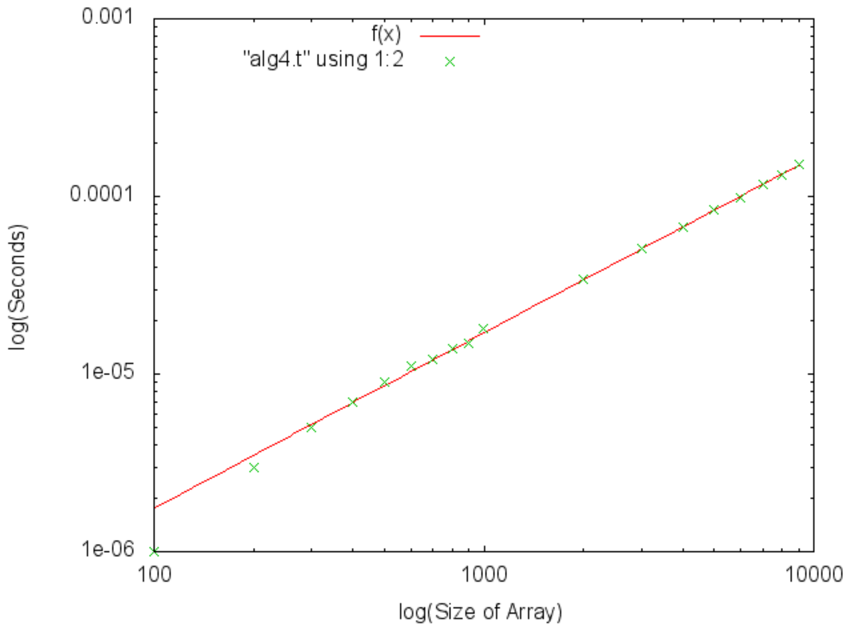


Figure 1: Algorithm 4 – execution time vs. input size

4 Theoretical correctness

Claim: `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 $max = current = 0$, which is correct.

For the inductive hypothesis, assume that `MaxSubarray(A, n)` will correctly return the sum of the maximum subarray of an array A of size n. We must show that `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: $A[n] < -$`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.

$\square$

5 Implement

5.1 Algorithm 4

```
/*
 * Enumeration
 * Loop over each pair of indices i; j and compute the sum from k=i to j of a[k].
 * Keep the best sum you have found so far.
 */

int MaxSubarray(int array[], int size){
  int current = 0;
  int max = 0;
  int i;

  for (i=0; i<size; i++) {
    current += array[i];
    if (current <= 0) {
      current = 0;
    } else if (current > max) {
      max = current;
    }
  }

  return max;
}
```

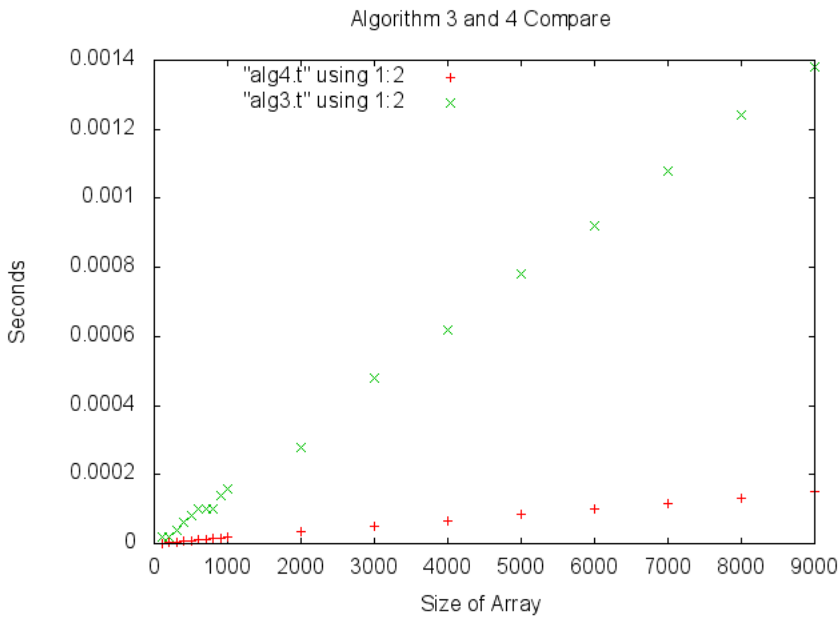
alg4.cpp

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

Well, there is a huge difference as seen on the Compare Plot. Algorithm 4, Dynamic Programming, is great because doesn't use any recurvsive calls and doesn't need to hold much data. Algorithm 4 only needs to hold onto 2 integers (max and current) and the input integer array. Whereas algorithm 3, divide & conquer, needs to use memory on the stack for each recurvsive call and needs to pass 4 integers back to the parent function.
WHAT THE HELL IS GOOD ABOUT D&C???


Algorithm 3 and 4 Compare
"alg4.t" using 1:2
"alg3.t" using 1:2
Seconds
0.0014
0.0012
0.001
0.0008
0.0006
0.0004
0.0002
0
0
1000
2000
3000
4000
5000
6000
7000
8000
9000
Size of Array