Assignment 2: Dynamic Programming project

Francis Vo, Soo-Hyun Yoo

October 30, 2012

1 Recursive function

```cpp
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;

    for i=0 to size:
        current = current + array[i]
        if current < 0:
            current = 0
        else if current > max:
            max = current

    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 3 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 $\Theta(n)$.

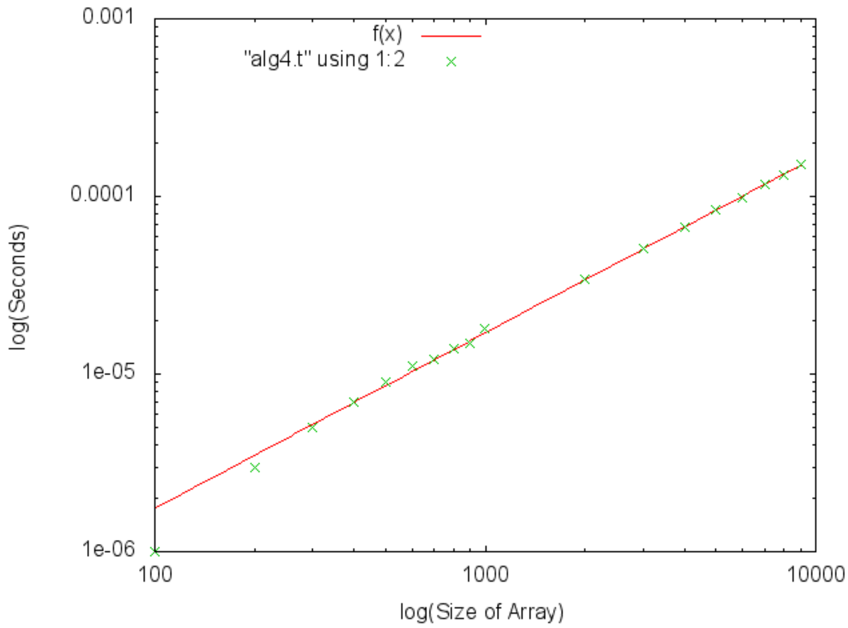


Figure 1: Algorithm 4 – execution time vs. input size

4 Theoretical correctness

Induction Proof. $MS(k)$ will return the maximum subarray sum for the array $A[0:k]$
Base case: If $n = -1$ then $max = current = 0$
Inductive Step: $maxSubarray(n-1).current + A[n]$ or 0 is the current largest sum starting from the left
Proof:
Case if $A[n] > 0$ then $current = MS(n-1).current + A[n] > MS(n-1).current$.
This number might also be the max value. So $max = Greater(max, current)$
Case if $A[n] > -maxSubarray(n-1)$ then $maxSubarray(n-1) + A[n] < 0$ making the Null set greater.
$max = MS(n-1).max$ and $current = 0$
Case else making $A[n]$ negative but $maxSubarray(n-1) + A[n] >= 0$
so it is still good to use for the next current: $current + A[n+1] > A[n+1]$
$max = MS(n-1).max$ and $current = maxSubarray(n-1) + A[n]$
$MS(n).max = max$ and $MS(n).current = current$

□

5 Implement

5.1 Algorithm 4

```cpp
/*
 * 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.
 */

using namespace std;


int MaxSubarray(int a[], int n){
  int current = 0;
  int max = 0;
  int i;
  for(i = 0;i < n; i++){
    current += a[i];
    if(current <= 0){
      current = 0;
    }else if(current > max){
      max = current;
    }
  }
  return max;
}
```

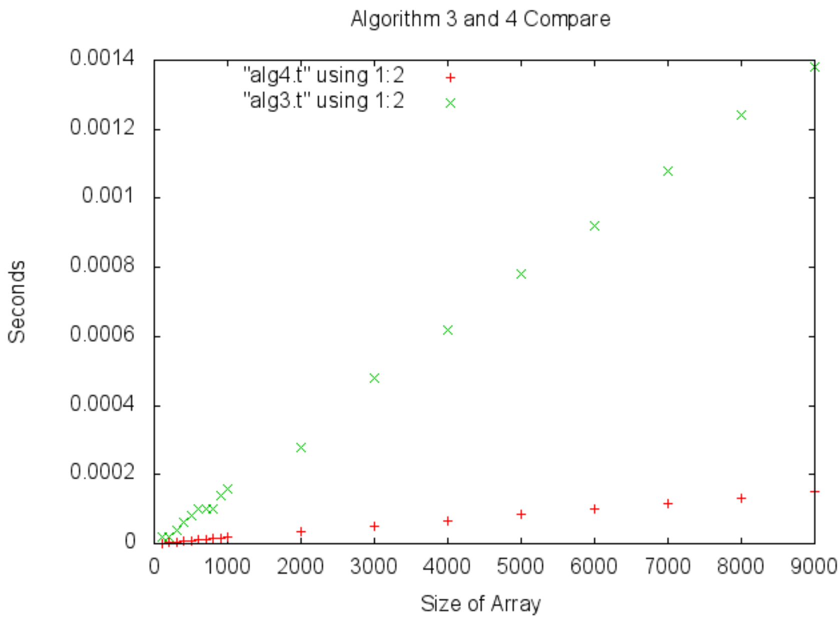
alg4.cpp

6 Test

Test were run on the ms_test.txt file given 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
Size of Array
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