CS352 Homework 1

Max sub array project

By Eric Rouse

# Mathematical Analysis:

## Algorithm 1 pseudocode:

def maxsubarray1(list):

for each element in list

for each element in list after first\_element

sum (first\_element, next\_element(s))

If sum greater than max, save sum as max

Repeat for each subsequent element

set next element as first\_element, repeat

This algorithm must pass through the list 2 times per element. So for every n elements it must pass through n-1 elements n-1 times. We are left with n\*(n-1)\*(n-1) = n3 – 2n2 + n. In the limit the n3 term dominates, we are left with O(n3)

## Algorithm 2 pseudocode:

Def maxsubarray2(list):

For each element in list

sum(first\_element)

if sum greater thanmax, save sum as max

set next element as first\_element, repeat

This algorithm passes through the list once per element. For n elements we pass through the list n-1 times. The result is n2-n, so in the limit it approaches an O(n2) running time.

## Algorithm 3 pseudocode:

Def maxsubarray3(list):

If list has zero elements, return 0

If list has one element return the greater of 0 and the single element

Else, divide the list into two halves, left and right

Sum from the middle to the left.

If sum is larger than max\_left, replace max\_left with sum

Sum from the middle to the right.

If max\_right is less than sum, replace max\_right with sum.

Return the greater of (max(maxsubarray3(left\_list),maxsubarray3(right\_list)) and max\_left + max\_right)

This algorithm uses a divide and conquer strategy, the list is divided in half for each element. So n\*log(n) – 2log(n) + 1. The result is an O(n log n) running time.

# Theoretical Correctness:

Using proof by induction, it is possible to prove the correctness of the recursive algorithm, maxsubarray3().

Since maxsubarray3() is a recursive algorithm it is best to prove with strong induction:

* Trivial case: list of zero elements, returns 0.
* The base case: list of one elements, returns the greater of 0 or the element listed.
  + Algorithm detects the largest possibility, either zero or the number.
  + Immediately returns this result, thus satisfying the requirement of finding the maximum of a single value.
* Inductive hypothesis: assume that maxsubarray3() on a region of k elements (where k >=1) returns the max array for that region.
  + k+1 > 1 by definition, executes neither base case, so the default code is executed.
  + There are three cases:
    - 1: Max array is fully enclosed in left half of the list
      * left\_list > right\_list && left\_list > max\_left + max\_right
    - 2: Max array is fully enclosed in right half of the list
      * right\_list > left\_list && right\_list > max\_left + max\_right
    - 3: Max array straddles the middle of the list
      * max\_left + max\_right > max\_right && max\_left + max\_right > max\_left
    - maxarray3() executes the same way for all these cases.
      * First it finds the greater of maxsubarray3(left\_list) vs. maxsubarray3(right\_list)
      * Since this is iterative, this will continue until we reach our base case of one element. So the maximums will be compared element by element.
      * Then it checks that max against the max of the middle list
      * It does this at each iteration step, ensuring that ONLY the highest value is cascaded forward.
      * So, on the unwind, the highest value, whether right, left or the middle, will be passed upward.
    - Thus, all cases are checked at every iteration step
* Hence the recursion returns the appropriate maximum, starting with the base case. It is proven by induction.

# Testing:

I performed the test as specified in the document, it said that the resul was recorded.

# Experimental Analysis:

The code to generate random numbers of the array size specified is attached as tester.py.

## Results:

### Algorithm 1, maxsubarray1():

|  |  |
| --- | --- |
| Size | Run Time |
| 100 | 0.000785804 |
| 200 | 0.0181916 |
| 300 | 0.141048503 |
| 400 | 0.633576894 |
| 500 | 2.139910388 |
| 600 | 5.79885602 |
| 700 | 13.63946042 |
| 800 | 29.4394686 |
| 900 | 56.0752856 |

As expected, this algorithm increases in runtime very quickly as the number of elements increase.

#### Algorithm 2, maxsubbarray2():

|  |  |
| --- | --- |
| Size | Run Time |
| 100 | 3.43E-05 |
| 200 | 0.000312591 |
| 300 | 0.00114181 |
| 400 | 0.003371 |
| 500 | 0.007214117 |
| 600 | 0.014108992 |
| 700 | 0.0253299 |
| 800 | 0.0407547 |
| 900 | 0.066061521 |
| 1000 | 0.097584081 |
| 2000 | 0.179565716 |
| 3000 | 0.362990689 |
| 4000 | 0.672370601 |
| 5000 | 1.228489614 |
| 6000 | 2.097222304 |
| 7000 | 3.481521106 |
| 8000 | 5.278179502 |
| 9000 | 7.971049905 |
| 10000 | 11.74170461 |

This algorithm performs much better than the previous, but is starting to become cumbersome around the final data points.

#### Algorithm 3, maxsubarray3():

|  |  |
| --- | --- |
| Size | Run Time |
| 100 | 3.69E-05 |
| 200 | 0.000172 |
| 300 | 0.00033 |
| 400 | 0.000883 |
| 500 | 0.001599 |
| 600 | 0.002928 |
| 700 | 0.004678 |
| 800 | 0.007172 |
| 900 | 0.011804 |
| 1000 | 0.016281 |
| 2000 | 0.028871 |
| 3000 | 0.054926 |
| 4000 | 0.099858 |
| 5000 | 0.177135 |
| 6000 | 0.302171 |
| 7000 | 0.490979 |
| 8000 | 0.794477 |
| 9000 | 1.159381 |
| 10000 | 1.728725 |
|  |  |

This is far and away the most efficient method of the three presented here.

#### Plots

All three algorithms shown on a single plot. Graphically shows how much better the second process is compared to the first.

This log-log plot demonstrates the exponential nature of these algorithms.

# Extrapolation and Interpretation:

## Question 1:

By extrapolating the data experimentally gathered, it is possible to estimate the number of elements that any given algorithm could parse in one hour.

First we use the log-log plot to determine the slope of the lines. (Tabulated in question 2). Since the formula for the log-log line is of the form Time = constant \* (elements)^(slope), we need to determine the constant to make our estimates. Using the slope from question 2 and the highest number of elements/time pair we can determine the constant.

* Constant of maxsubarray1(): c = 56.075/(9005.13) = 3 x10-14
* Constant of maxsubarray2(): c = 11.7417/(100002.22) = 2 x10-8
* Constant of maxsubarray3(): c = 56.075/(100001.60) = 1 x10-6

So, maximum elements of each can be calculated using the following, 3600/(constant)^slope:

* Elements in one hour of maxsubarray1() = 2134
* Elements in one hour of maxsubarray2() = 97864
* Elements in one hour of maxsubarray3() = 939051

## Question 2:

Log-log plot slope = log(y2/y1)/log(x2/x1)

Slope of maxsubarray1(): 5.13

Slope of maxsubarray2(): 2.22

Slope of maxsubarra3(): 1.60

There exist discrepancies between the values determined experimentally and the theoretical values above. This is due, at least in large part, to the simplification of Big O notation to focus on the dominating term.

In order to make the experiment less onerous we only ran values up to 900 elements on maxsubbary1 and 10000 elements on the other two functions. This means the dominating term hadn’t really been given enough time to fully dominate. Thus each value is somewhat higher than the predicted value.