CS325 Project 2 – Maximum Sub-Array; Dynamic Programming

Project Team: Eric Rouse

Resources:

1) Matt Schriever on Piazza

Recursive Function

Assuming the maximum subarray uses the last element means that we must start from the back of the array and work down. Since we are building a table of maximums we put the last item in the array into the table at position zero. This is the current maximum.

Now we can recurrsively work our way down the rest of the array, reducing the array size by on at each call. At each call we compare the new last element of the array to the last element of the table. The larger of the element or the result of adding the element to the current maximum is appended to the end of the table for the new local maximum.

When the recurrsion is done the maximum element of table will be the maximum subarry.

Psuedocode:

```
DYNAMIC_MAX_SUB(ar)
    #verify array is not empty or one element long
    if |array| = 0, return 0
    if |array| = 1, return that element
    #initialize a dynmaic programming table
    dyn_table = [ ]
    #max is either the last element, or it isn't
    append array[last element] to dyn_table
    return max (TABLE_GENERATOR(ar[all but last element],dyn_table))
TABLE_GENERATOR(array, dyn_table)
    if |array| <= 1, return dyb_table
#always append (to the dynamic table) the greater of the current element or the
addition of the current element to the current max.
   append max(array[last element], array[last element] + dyn_table[last element]
   return RECURSION(ar[all but last element],dyn_table)
```

Running Time:

T(n) = T(n-1) + c, because the recursive step operates on an array reduced in size by one element every recursive step. After K steps we are left with T(n-k)+k*c, solving for depth at k reveals k = n-1. Substitute: T(n-n+1)+ c*(n-1), simplify T(1) + c*n – c. So the running time works out to be O(n).

Theoretical Correctness

Given an array A of n elements, let recurrsive_algorithm return the sum of the maximum sub-array.

Claim

Recurrsive_algorithm(A) correctly returns the maximum sum of the sub-elements of A.

Proof

For an array A, let P(A) be the statement that recurrsive_algorithm(A) correctly returns the maximum sum of all sub-arrays of A.

As a base case, consider when |A| = 1. This one-element array is already the maximum possible sub array and the algorithm correctly returns the value of A as the maximum sum.

For the induction hypothesis, suppose that P(A) is true for all array of length < n; that is, suppose that for any array A of length < n, recurrsive_algorithm(A) correctly sums A. Now consider an array A of length n. Our algorithm reduces A by n - 1 which is less than n; and is therefore summed properly by the induction hypothesis.

The maximum of either the current element or the current element plus the current maximum is stored. This is repeated until the base case is reached. The maximum of that area will be the maximum sub array. Therefore, by induction, recurrsive_algorithm(A) correctly sums any sub-array in any array.

Implement

```
#
#   Name:            Eric Rouse
#   Language:        Python 3, tested in Python 3.3.3
#   Email Address:   rousee@onid.orst.edu
#   Class Name:      CS325
#   Assignment:      Project #2

def maxsub_dynamic(ar):
    if len(ar) <= 0:
        return 0
    if len(ar) == 1:
        return ar[0]
    #dynamic table
```

```
    dtbl = []
    #max is either the last element, or it isn't
    dtbl.append(ar[len(ar)-1])
    return max(table_generator(ar[:-1],dtbl))

def table_generator(ar,tb):
    if len(ar) <= 0:
        return tb
    el = ar[len(ar)-1]
    tb.append(max(el, el+tb[len(tb)-1]))
    return table_generator(ar[:-1],tb)
```

Test

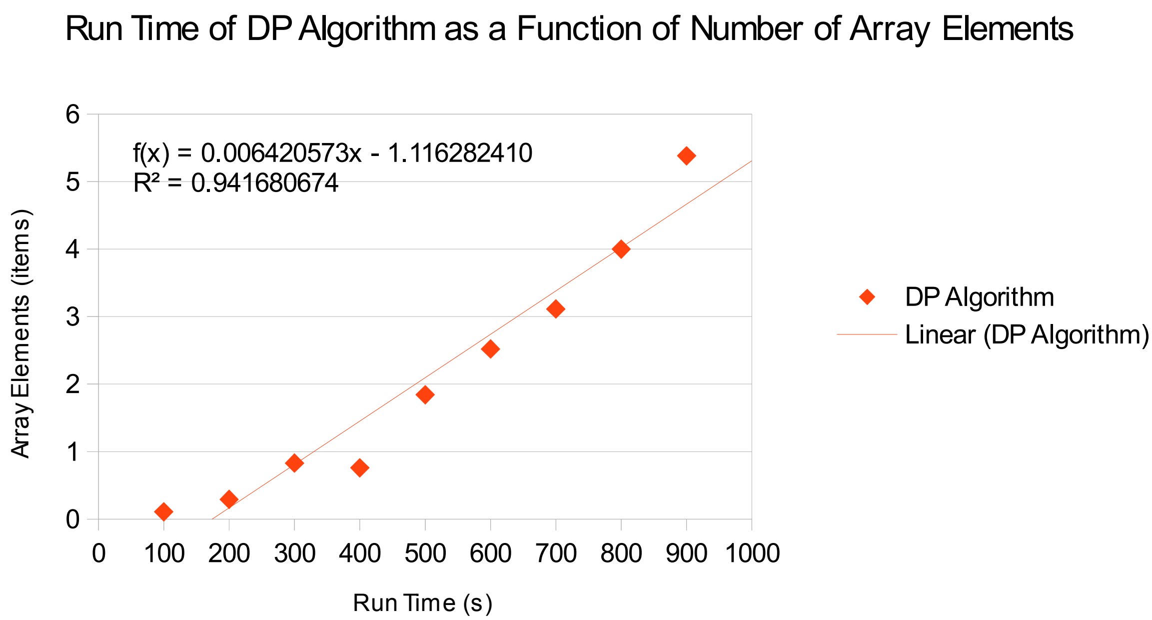
Code was tested to the same arrays as Project 1 and passed all with flying colors.

Tabulated data:

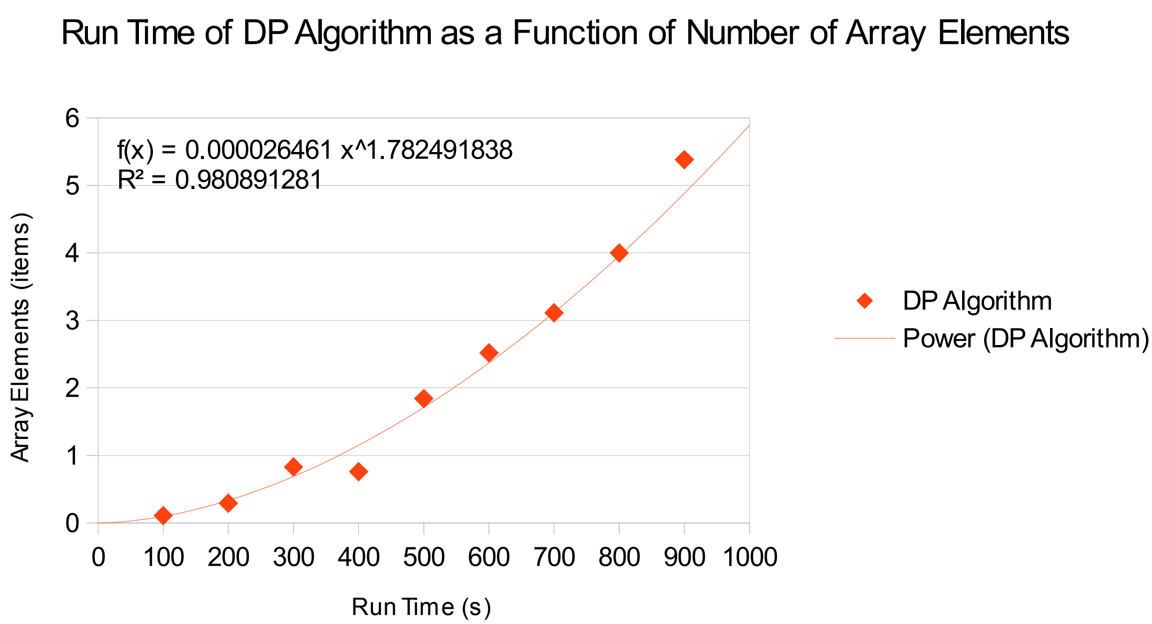
length of array (items):	Algorithm 1	Algoritm 2	Algorithm 3	DP Algorithm
100	2.7716159821	0.0872612	0.7102489471	0.1094341278
200	5.4697990417	0.0882148743	0.7722377777	0.2927780151
300	11.5578174591	0.1192092896	1.1301040649	0.8301734924
400	20.049571991	0.1575946808	1.9469261169	0.7607936859
500	37.1966362	0.1928806305	1.9254684448	1.8424987793
600	45.1765060425	0.2381801605	2.3641586304	2.5193691254
700	59.3752861023	0.2632141113	2.8069019318	3.1123161316
800	82.5231075287	0.3051757813	3.1435489655	3.998041153
900	104.8910617828	0.3345012665	3.5936832428	5.3806304932

Compare

Although the theoretic runtime is linear, th data doesn't line up with O(n) perfectly, the R^2 0.94 is OK, but could be better.



It turns out that the constant time operations (mostly splitting the array, I'm guessing) have an increasingly important effect. Using a power fit (with an R^2 of 0.98) the constant term is very, very small. So even though this is $O(n^2)$ it is kind of close to linear.



For comparrions sake, let's look at how many values our algorithm could solve in an hour versus the Project 1 algoritms.

Algorithm 1

n = 283,443

Algorithm 2

n = 9,316,543

Algorithm 3

n = 752,992

Dynamic Programming Algorithm – upper bound best fit curve, linear fit: f(n) = 0.006420473*n - 1.116282410, R^2 of 0.9417, solve for n:

n = (f(n) + 1.116282410)/0.006420473

let f(n) = 1 hour = 3600 seconds

n = (3600 - 1.116282410)/0.006420473

n = 560,532

Dynamic Programming Algorithm – upper bound best fit curve, power fit: f(n) = $0.000026461n^{1.782491838}$, R^2 of 0.9809, solve for n:

n = $\log_{1.782491838}$(f(n)/0.000026461)

let f(n) = 1 hour = 3600 seconds

n = $(3600/0.000026461)^{1/1.782491838}$

n = 36,568

The dynamic programming algorithm, in this implementation, is much slower that both Algorithm2 and Algorithm3 from Project 1. In the linear fit case, it performs better than Algorithm 1.

That all didn't really sit right with me, so I did some research. It turns out Python (my chosen language for algorithm implimentation) is kind of terrible at recurrsion and list iterations are prefered. So, rewriting my algorithm to use list iteration I get:

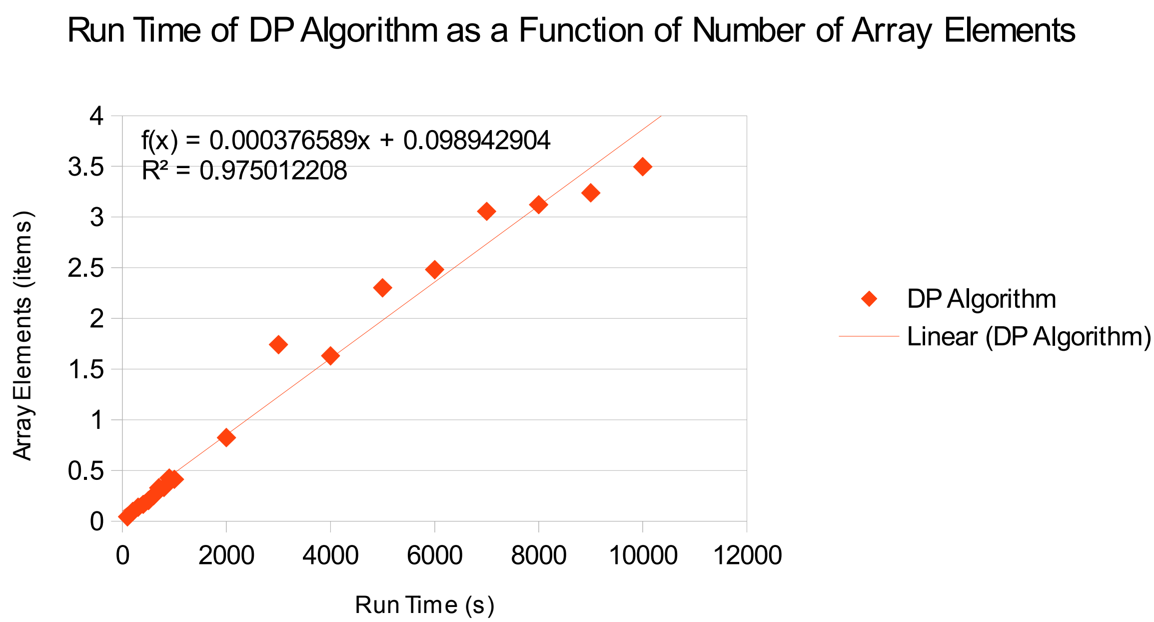
```
def maxsub_dynlist(ar):
   #base cases
   if len(ar) <= 0:
      return 0
   if len(ar) == 1:
      return ar[0]
   #start at the back
```

```
ar.reverse()
#dynamic table
dtbl = []
#max is either the last element, or it isn't
dtbl.append(ar[0])
for i, el in enumerate (ar[0:]):
    dtbl.append(max(el+dtbl[i], el))
return max(dtbl)
```

The run times are MUCH faster:

n (items)	time (s)
100	0.0429153442
200	0.0965595245
300	0.1366138458
400	0.1649856567
500	0.2026557922
600	0.2508163452
700	0.3290176392
800	0.333070755
900	0.4251003265
1000	0.4131793976
2000	0.8239746094
3000	1.7416477203
4000	1.6305446625
5000	2.3019313812
6000	2.480506897
7000	3.0560493469
8000	3.1216144562
9000	3.2386779785
10000	3.4971237183
10000000	38333.309

And much more linear:



Resulting in processing **9,559,230** items in an hour. So, the dynamic programing method is far superior, if implimented correctly in the chosen language! This could be why my data from Project 1 was so weird!