CS 395 Homework 5

Colby Blair

Due February 29th, 2012

Grade: _____

PROBLEMS

1.

If all entries in the subarray A are negative, then the array is split into subarrays until a subarray with one element is found. This element is the maximum, least negative, or closest to zero value of any elements.

The function $FIND\_MAXIMUM\_SUBARRAY(A, low.high)$ returns the beginning index of the subarray (low), the end index of the subarray (high), and the sum of the elements in the subarray. In the case of all negatives, low and high both equal the maximum element, so the subarray is of length 1. The sum is then, obviously, just the one value of the subarray.

For example, consider the subarrays $A = [-6, -3, -2, -5, -4, -7, -12, -4, -3, -6, -5]$, $B = [-9, -5, -4, -7, -2, -4, -5]$, and $C = [-22, -43, -21, -67, -99, -10]$. $FIND\_MAXIMUM\_SUBARRAY()$ would then be $(3, 3, -2)$, $(5, 5, -2)$, and $(6, 6, -10)$, respectively.

2.

Pseudocode for the naive $FIND\_MAXIMUM\_SUBARRAY()$ function is as follows:

	FIND_MAXIMUM_SUBARRAY()	C	T
1	max_sum = -infinity	C1	1
2	left_index = 0	C2	1
3	right_index = 0	C3	1
4	for j in 1 to n	C4	n
5	sum = 0;	C5	n - 1
6	for (int i = start; i < 16; i++)	C6	(n-1)(n)
7	sum += a[i];	C7	(n-1)(n-1)
8	if (sum > max_sum)	C8	(n-1)(n-1)
9	max_sum = sum;	C9	(n-1)ti
10	leftIndex = start;	C10	(n-1)ti
11	rightIndex = i;	C11	(n-1)ti

In the worst case scenario, each element in the array is positive, so sum is always $> max_sum$. Thus:

$$t_i = \quad n \tag{1}$$

This doesn't really mattter, however, because there is always at least the operation on line 7 that executes.
For the worst case, the total run time $T(n)$ is as follows:

$$T(n) = C_1 + C_2 + C_3 + C_4n + C_5(n-1) + C_6(n-1)n + C_7(n-1)(n-1) + C_8(n-1)(n-1) + C_9(n-1)n + C_{10}(n-1)n + C_{11}(n-1)n \tag{2}$$

$$= C_1 + C_2 + C_3 - C_4n + C_5n - C_5 + C_6n - C_6 + C_7n^2 - 2C_7n + C_7 + C_8n^2 - 2C_8n + C_8 + C_9n^2 - C_9n + C_{10}n^2 - C_{10}n + C_{11}n^2 - C_{11}n \tag{3}$$

$$= C_1 + C_2 + C_3 - C_5 - C_6 + C_7 + C_8 - C_9 + n(-C_4 + C_5 + C_6 - 2C_7 - 2C_8 - C_9 - C_{10}) + n^2(C_7 + C_8 + C_9 + C_{10} + C_{11}) \tag{4}$$

$$= an^2 + bn + c \tag{5}$$

Even in the best case, C_7 ends up with a n^2. With $T(n) = an^2 + bn + c$, $T(n) = \Theta(n^2)$.

3.

Python code for the naive function $FIND\_MAXIMUM\_SUBARRAY()$:

```
#!/usr/bin/env python2.7

import time

def find_max_subarray_naive(A):
        sum = 0
        max_sum = 0
        low = 0
        high = 0
        for j in range(0, len(A)):
                sum = 0
                for i in range(j, len(A)):
                        sum += A[i]
                        if sum > max_sum:
                                max_sum = sum
                                low = j
                                high = i
        return (low, high, max_sum)


#main

A = [-1 ,3, 2, 13, -4, 7, -12, 4, -3 , 6, -5]
#A = [-6 ,-3, -2, -5, -4, -7, -12, -4, -3 , -6, -5]
#A = [-9, -5, -4, -7, -2, -4 ,-5]
#A = [-22, -43, -21, -67, -99, -10]

# test all
print "Results: ",
print A
stime = time.time()
print find_max_subarray_naive(A)
etime = time.time()
ttime = etime - stime
print " took %s seconds" % ttime
```

The output is:

```
$ python find_max_subarray_naive.py
Results:  [-1, 3, 2, 13, -4, 7, -12, 4, -3, 6, -5]
(1, 3, 18)
 took 0.000226974487305 seconds
```

Python code for the recursive function $FIND\_MAXIMUM\_SUBARRAY()$:

```
1  #!/usr/bin/env python2.7
2
3  import time
4
5  def find_max_crossing_subarray(A, low, mid, high):
6          max_left = 0
7          max_right = 0
8          #Find a max subarray of the form A[i..mid]
9          left_sum = -float("inf")
10         sum = 0
11         for i in range(mid, low - 1, -1): #for i = mid downto low
12                 sum = sum + A[i]
13                 if sum > left_sum:
14                         left_sum = sum
15                         max_left = i
16         #Find a max subarray of the for A[mid + 1 .. high]
17         right_sum = -float("inf")
18         sum = 0
19         for j in range(mid+1, high):
20                 sum = sum + A[j]
21                 if sum > right_sum:
22                         right_sum = sum
23                         max_right = j
24         return (max_left, max_right, left_sum + right_sum)
25
26
27 def find_max_subarray(A, low, high):
28         #best case, one element
29         if high == low:
30                 return (low, high, A[low])
31         else:
32                 mid = (low + high) / 2
33                 (left_low, left_high, left_sum) = \
34                         find_max_subarray(A,low, mid)
35                 (right_low, right_high, right_sum) = \
36                         find_max_subarray(A, mid + 1, high)
37                 (cross_low, cross_high, cross_sum) = \
38                         find_max_crossing_subarray(A, low, mid, high)
39                 if left_sum >= right_sum and \
40                         left_sum >= cross_sum:
41                         return (left_low, left_high, left_sum)
42                 elif right_sum >= left_sum and \
43                         right_sum >= cross_sum:
44                         return (right_low, right_high, right_sum)
45                 else:
```

```
46                          return (cross_low, cross_high, cross_sum)
47
48 #main
49
50 A = [-1 ,3, 2, 13, -4, 7, -12, 4, -3 , 6, -5]
51 #A = [-6 ,-3, -2, -5, -4, -7, -12, -4, -3 , -6, -5]
52 #A = [-9, -5, -4, -7, -2, -4 ,-5]
53 #A = [-22, -43, -21, -67, -99, -10]
54 low = 0
55 high = len(A)
56 mid = (low + high) / 2
57
58 # test cross search
59 '''
60 print "Results: ",
61 print A
62 print find_max_crossing_subarray(A, low, mid, high)
63 '''
64
65 # test all
66 print "Results: ",
67 print A
68 stime = time.time()
69 print find_max_subarray(A, low, high-1)
70 etime = time.time()
71 ttime = etime - stime
72 print " took %s seconds" % ttime
```

The output is:

```
1 $ python find_max_subarray.py
2 Results:  [-1, 3, 2, 13, -4, 7, -12, 4, -3, 6, -5]
3 (1, 3, 18)
4  took 0.000158786773682 seconds
```

The naive approach never beats the recursive approach, with $\Theta(n^2)$ and $\Theta(nln(n))$, respectively. See Figure 1:

n	n^2	n lg n
1	1	0
2	4	2
3	9	4.7548875022
4	16	8
5	25	11.609640474
6	36	15.509775004
7	49	19.651484454
8	64	24
9	81	28.529325013
10	100	33.219280949
11	121	38.053747805
12	144	43.019550009
13	169	48.105716336
14	196	53.302968909
15	225	58.603358934
16	256	64
17	289	69.486868301
18	324	75.058650026
19	361	80.710622755
20	400	86.438561898
21	441	92.238665878
22	484	98.10749561
23	529	104.04192499
24	576	110.03910002
25	625	116.09640474
26	676	122.21143267
27	729	128.38196256
28	784	134.60593782
29	841	140.88144886
30	900	147.20671787
31	961	153.58008562
32	1024	160
33	1089	166.46500594
34	1156	172.9737366
35	1225	179.52490559
36	1296	186.11730005
37	1369	192.74977453
38	1444	199.42124551
39	1521	206.13068654

Figure 1: Naive vs Recursive Max Subarray Search comparison

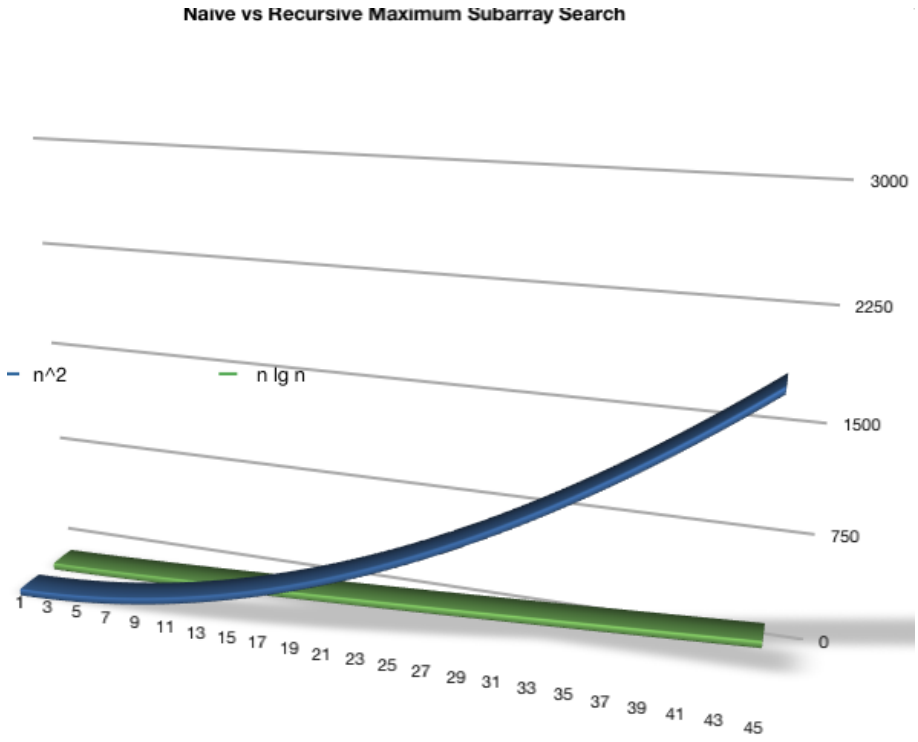

Figure 2: Naive vs Recursive Max Subarray Search comparison graph