Collin O'Connor

Comp 3270 Algorithms

November 16, 2016

Algorithms Project

STRATEGIES(1.1):

1. **Strategy-1:** This strategy first checks if the array of numbers from the input is equal to one. If it is, then that number will be printed as well as both indices being the first index for the start and end of the subset. If not, the algorithm will go through the array, checking if every integer is positive, and if so, it will print the smallest integer and its index. If not, then check if the array is all negative integers, if so then it will use the original set as the subset, summing all of it, and using the starting and ending indices of the original set. If all of these fail then, start at the beginning of the array, and sum each consecutive integer, however, if the sum reaches 0 or greater (or reaches the end of the array), back track to the indices in which the subset had the lowest value, set as the lowest sum if one has not been set, otherwise compare to the current lowest sum, if it is lower, then set as the new lowest sum, the indices will be saved as well. Continue this process, beginning with the index that is after the last index to fail. After the end of the array has been reached, the lowest sum will be printed, along with its start and end indices. (Source: myself)

2. **Strategy-2:** This strategy walks through the array of numbers from the input, creating all of the possible consecutive subsets that can be created from the original set, summing them and saving the indices. It will keep track of the subset that has the lowest possible value. After this, it will print the sum of the selected subset, along with two integers corresponding to the start index and end index. (Source: Brute Force strategy that is hinted at in the project assignment.)

3. **Strategy-3:** This strategy is a slightly modified version of the Dynamic Solution Strategy to the maximum contiguous sum sub-array problem. It walks through the numbers from the input array, creating subarrays, and saving the minimum subarray, however, it will ignore the sum of the previous n-1 elements if the nth element is less than the sum, it will save the indices of the minimum sum along the way. Once the entire array has been walked through the sum and indices are printed. (Source: Dynamic Solution presented here: http://www.programcreek.com/2013/02/leetcode-maximum-subarray-java/)

ALGORITHMS(1.2):

1. **Algorithm-1:** (r is the input size, n)

```
Algorithm-1(Arr[1…r]: array of integers r ≥ 1)
1)  if 1 = = r then print Arr[1], 1, 1 and return
2)  minLeft = 1, minRight = 1, minSum = Arr[1]
3)  for i = 1 to r
4)      if Arr[i] > 0 and Arr[i] < minSum
5)          then minSum = Arr[i], minLeft = i, minRight = i
6)      if Arr[i] < 0 then negCount++;
7)      if negCount == r then break loop
8)      if i = = r and negCount < 0 then print minSum, minLeft, minRight, and return
9)  if allNeg = true then minSum = Arr[1]
10)     for j = 2 to r
11)         minSum += Arr[j]
12)     print minSum, 1, r, and return
13) minLeft = 1, minRight = 1, minSum = Arr[1]
14) for j = 1 to r
15)     minSumNew += Arr[j]
16)     minRightNew = j
17)      if minSumNew < minSum
18)         then minSum = minSumNew,
19)         minLeft = minLeftNew, minRight = minRightNew
20)     if minSumNew >= 0
21)         then minSum = 0, minLeftNew = j + 1
22) print minSum, minLeft, minRight
```

2. **Algorithm-2:** (r is the input size, n)

```
Algorithm-2(Arr[1…r]: array of integers r ≥ 1))

1)  if 1 = = r then print Arr[1], 1, 1
2)  minSum = Arr[1], minLeft = 1, minRight = 1
3)  for i = 1 to r – 1
4)      minSumNew = arr[i];
5)      minLeftNew = i
6)      if minSumNew < minSum
7)              then minSum = minSumNew
8)              minLeft = minLeftNew
9)              minRight = j
10)     for j = i + 1 to r
11)         minSumNew = minSumNew + Arr[j]
12)         if minSumNew < minSum
```

```
13)            then minSum = minSumNew
14)            minLeft = minLeftNew
15)            minRight = j
16) if arr[r] > minSum then minSum = arr[r], minRight = r, minLeft = r
17) print minSum, minLeft, minRight
```

3. **Algorithm-3:** (r is the input size, n)

```
Algorithm-3(Arr[1…r]: array of integers r ≥ 1)
1) min = Arr[1], minLeftNew = 1
2) sum = sum[1…r]
3) sum[1] = Arr[1]
4) for i = 2 to r
5)     sum[i] = Math.min(Arr[i], sum[i-1] + Arr[i])
6)     min = Math.min(min, sum[i])
7)     if min == sum[i] then minRight = i, minLeft = minLeftNew
8)     if sum[i] > 0 then minLeftNew = i + 1
9) if minRight ==1 AND arr[1] > 0 then minLeft = 1
10) prints min, minLeft, minRight
```

T(n) CALCULATIONS(1.3):

1. **Algorithm-1:** (where r = n, n is the size of the input array)
   1) 1(read r) + 1(compare = =) + 1(access Arr[1]) + 1(read Arr[1]) = **3**
   2) 1(write minLeft) + 1(write minRight) + 1(access Arr[1]) +1(read Arr[1]) + 1(write minSum) = **5**
   3) 1(for loop) * n+1(amount of times for loop condition is checked) = **n+1**
   4) 1(read i) + 1(access Arr[i]) +1(read Arr[i]) + 1(compare >) + 1(compare =)+ 1(read i) + 1(access Arr[i]) +1(read Arr[i]) + 1(read minSum) + 1(compare <) * n(for loop executions) = **10n**
   5) 1(read i) + 1(access Arr[i]) +1(read Arr[i]) + 1(write minSum) + 1(read i) + 1(write minLeft) + 1(read i) + 1(write minRight) * n(for loop executions) = **8n**
   6) 1(write false) * n(for loop executions) = **n**
   7) 1(read i) + 1(access Arr[i]) + 1(read Arr[i]) + 1(compare <) + 1(read negCount) + 1(add 1 to negCount) * n(for loop executions) = **6n**
   8) 1(read negCount) + 1(read r) + 1(compare ==) + 1(read negCount) + 1(compare <) * n(for loop executions) = **5n**
   9) 1(read i) + 1(read r) + 1(compare ==) * n(for loop executions) = **3n**
   10) 1(read allNeg) + 1(compare =) + 1(access A[1]) + 1(read A[1]) = **4**
   11) 1(for loop) * n+1(amount of times for loop condition is checked) = **n+1**
   12) 1(read minSum) + 1(read j) + 1(access Arr[j]) + 1(read Arr[j]) + 1(add Arr[j]) + 1(write minSum) * n(for loop executions) = **6n**
   13) 1(read minSum) = **1**

14) 1(write minLeft) + 1(write minRight) + 1(access Arr[1]) + 1(read Arr[1]) + 1(write minSum) = **5**
15) 1(for loop) * n+1(amount of times for loop will be checked) = **n+1**
16) 1(read minSumNew) + 1(read j) + 1(access Arr[j]) + 1(read Arr[j]) +1(add Arr[j]) + 1(write minSumNew) * n(for loop executions) = **6n**
17) 1(read j) + 1(write minRightNew) * n(for loop executions) = **2n**
18) 1(read minSumNew) + 1(read minSum) + 1(compare >) * n(for loop executions) = **3n**
19) 1(read minSumNew) + 1(write minSum) * n(for loop executions) = **2n**
20) 1(read minLeftNew) + 1(write minLeft) + 1(read minRightNew) + 1(write minRight) *n(for loop executions) = **4n**
21) 1(read minSumNew) + 1(compare >) + 1(compare ==) *n(for loop executions) = **3n**
22) 1(write minSum) + 1(read j) +1(add 1) + 1(write minLeftNew) *n(for loop executions) = **4n**
23) 1(read minSum) + 1(read minLeft) + 1(read minRight) = **3**

T(n) = 3 + 5 + n + 1 + 10n + 8n + n + 6n + 5n + 3n + 4 + n + 1 + 6n + 1 + 5 + n + 1 + 6n + 2n + 3n + 2n + 4n + 3n + 4n + 3

T(n) = 66n + 23

2. **Algorithm-2:** (where r = n, n is the size of the input array)
   1) 1(read r) + 1(compare ==) + 1(access Arr[1]) + 1(read Arr[1]) = **4**
   2) 1(access Arr[1]) + 1(read Arr[1]) + 1(write minSum) + 1(write minLeft) + 1(write minRight) = **5**
   3) 1(for loop) *n(amount of times for loop condition will be checked) = **n**
   4) 1(read i) + 1(access arr[i]) + 1(read arr[i]) + 1(write minSumNew) * n-1(for loop execs) = **3n-3**
   5) 1(read i) + 1(write minLeftNew) * n-1(for loop execs) = **2n – 2**
   6) 1(read minSumNew) + 1(read minSum) + 1(compare <) * n-1(for loop execs) = **3n – 3**
   7) 1(read minSumNew) + 1(write minSum) * n-1(for loop execs) = **2n-2**
   8) 1(read minLeftNew) + 1(write minLeft) * n-1(for loop execs) = **2n -2**
   9) 1(read j) + 1(write minRight) * n-1(for loop execs) = **2n-2**
   10) 1(for loop) *n(amount of times for loop condition will be checked) *n-1(outer for loop) = **n^2-n**
   11) 1(read minSum) + 1(read j) + 1(access Arr[j]) + 1(read Arr[j]) + 1 (add minSum and Arr[j]) + 1(write minSumNew) * n-1(for loop execs) *n-1(outer loop) = 6**n^2-12n+6**
   12) 1(read minSumNew) + 1(read minSum) + 1(compare <) *n-1(for loop execs) *n-1(outer loop) = **3n^2-6n+3**
   13) 1(read minSumNew) + 1(write minSum) *n-1(for loop execs) *n-1(outer loop) = **2n^2-4n+2**
   14) 1(read minLeftNew) + 1(write minLeft) *n-1(for loop execs) *n-1(outer loop) = **2n^2-4n+2**

15) 1(read minRightNew) + 1(write minRight) *n-1(for loop execs) *n-1(outer loop) = **$2n^2$-4n+2**

16) 1(read r) + 1(access arr[r]) + 1(read arr[r]) + 1(read minSum) + 1(compare >) + 1(read r) + 1(access arr[r]) + 1(read arr[r]) + 1(write minSum) + 1(read r) + 1(write minRight) + 1(read r) + 1(write minLeft) = **13**

17) 1(read minSum) + 1(read minLeft) + 1(read minRight) = **3**

T(n) = 4 + 5 + n + 2n – 2 + n^2 – n + $6n^2$ – 12n + 6 + $3n^2$ – 6n + 3 + $2n^2$ – 4n + 2 + $2n^2$ – 4n + 2 + $2n^2$ – 4n + 2 + 3 + 13 + 2n – 2 + 2n – 2 + 2n – 2 + 3n - 3 + 3n - 3

T(n) = $16n^2$ – 16n + 24

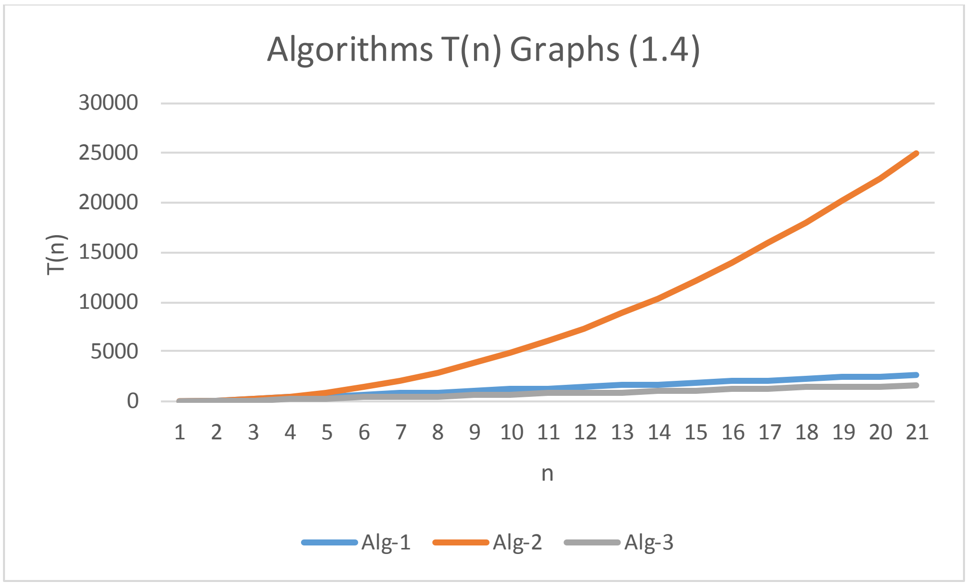
3. **Algorithm-3:** (where r = n, n is the size of the input array)

1) 1(access Arr[1]) + 1(read Arr[1]) + 1(write min) + 1(write minLeftNew) = **4**
2) 1(read r) + 1(create new array, length r, sum) + 1(write sum) = **3**
3) 1(access sum[1]) + 1(access Arr[1]) + 1(read Arr[1]) + 1(write sum[1]) = **4**
4) 1(for loop) *n(amount of times loop condition will be checked) = **n**
5) 1(read i) + 1(access sum[i]) + 1(write sum) + 1(read i) + 1(access Arr[i]) + 1(read Arr[i]) + 1(read i) + 1(i sub 1) + 1(access sum[i-1]) + 1(read sum[i-1]) + 1(read i) + 1(access Arr[i]) + 1(read Arr[i]) + 3(Math.min method) * n-1(for loop execs) = **16n - 16**
6) 1(read min) + 1(read i) + 1(access sum[i]) + 1(read sum[i]) +3(Math.min) + 1(write min) *n-1(for loop execs) = **8n-8**
7) 1(read min) + 1(read i) + 1(access sum[i]) + 1(read sum[i]) + 1(compare ==) + 1(read i) + 1(write minRight) + 1(read minLeftNew) + 1(write minLeft) *n-1(for loop execs) = **9n-9**
8) 1(read i) + 1(access sum[i]) + 1(read sum[i]) + 1(compare >) + 1(read i) + 1(add i+1) + 1(write minLeftNew) * n-1(for loop execs) = **7n-7**
9) 1(read minRight) + 1(compare ==) + 1(access Arr[1]) + 1(read Arr[1]) + 1(compare >) + 1(write minLeft) = **6**
10) 1(read min) + 1(read minLeft) + 1(read minRight) = **3**

T(n) = 4 + 3 + 4 + n + 16n – 16 + 8n – 8 + 9n – 9 + 7n – 7 + 6 + 3

T(n) = 41n - 20

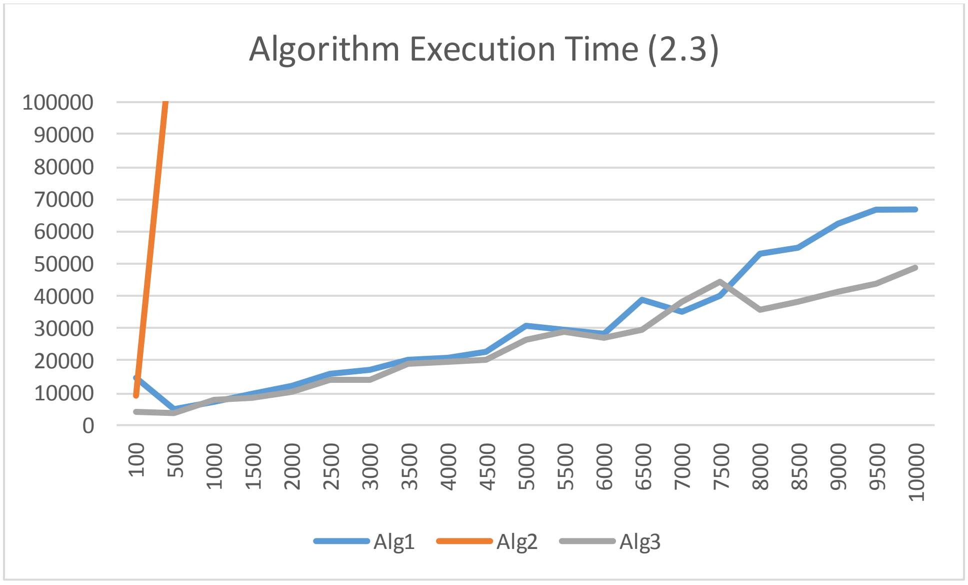
GRAPH OF T(n)’S (1.4):



EVALUATION(1.5):

1. The most efficient algorithm is algorithm-3. This algorithm has O(n) time complexity which is lower than the others.
2. Algorithm-1 is the second most efficient algorithm of the three. Even though the algorithm's time complexity is O(n), its T(n) is greater than that of algorithm-3. It is also less elegant.
3. The least efficient algorithm is algorithm-2. It has O(n^2) for its time complexity. Which is drastically worse than the other two algorithms.

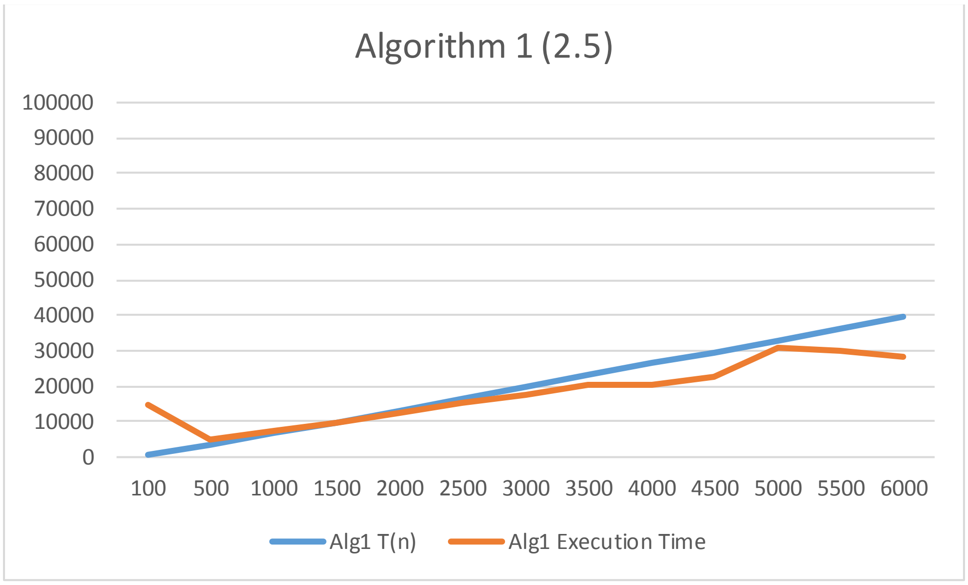
GRAPH AFTER IMPLEMENTATION (2.3)

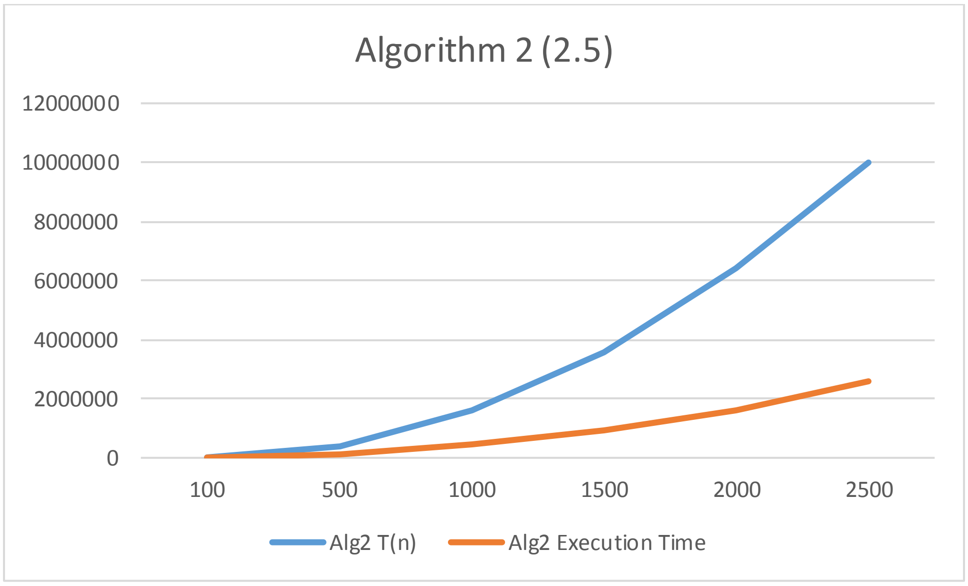


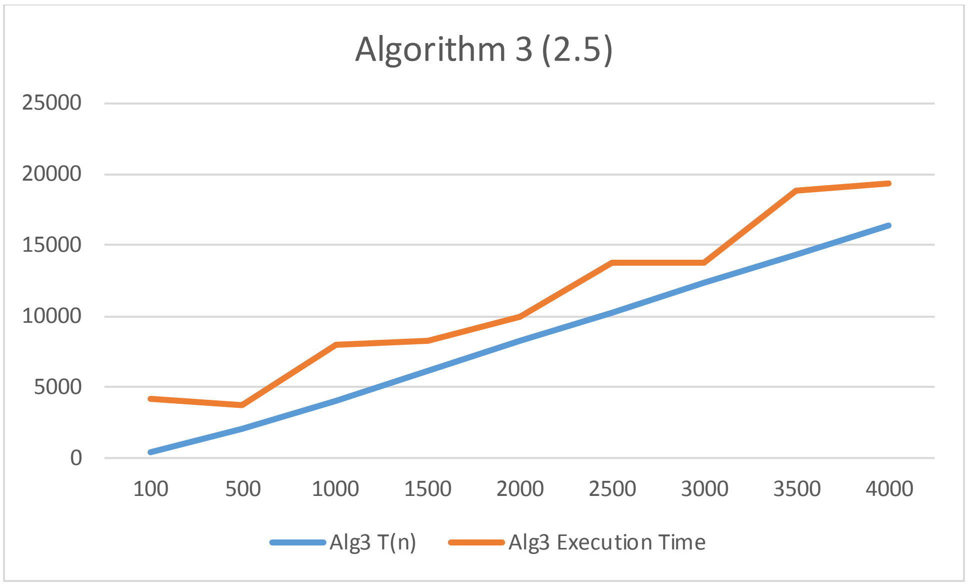
EVALUATION AFTER EMPIRACAL ANALYSIS(2.4):

1. The most efficient algorithm after empirical analysis is still algorithm-3. This algorithm, while close to algorithm-1, is still the most efficient after being tested on 21 different size arrays, 5000 times each. It performed the best on 18 out of 21 sizes of n, in this sample set.
2. Algorithm-1 is the second most efficient algorithm of the three. Even though the algorithm's time is close to algorithm-3, it starts to become vastly slower around n=8000.
3. The least efficient algorithm is algorithm-2. It is exponentially larger than the other two algorithms, even as early as n=500.

GRAPHS T(n) VS EXECUTION TIME(2.5):







The algorithms graphed by T(n) vs recorded execution time are not an exact match. This is because T(n) is the calculation of steps usually accepting the worst case scenario. Whereas execution time is recorded over several sample sizes of which may be best, average, or worst case. However, the overall trend of the curves match. Algorithms 1 and 3 are both linear as they are supposed to be, and Algorithm 2's execution time is graphed quadratic, as the T(n) predicted.