Davis Reid Watson
COMP 3270 Programming Project Homework
7/20/2021

Algorithm-1

Step	Cost of each execution	Total # of times executed
1	1	1
2	1	n+1
3	1	$\sum_{i=P}^{n} (n - i + 2)$
4	1	$\sum_{i=P}^{n} (n - i + 1)$
5	1	$\sum_{i=P}^{n} (\sum_{i=P}^{n} (n - i + 1)+1)$
6	6	$\sum_{i=P}^{n} (\sum_{i=P}^{n} (n - i + 1))$
7	8	$\sum_{i=P}^{n} (n - i + 1)$
8	1	1

Multiply col.1 with col.2, add across rows and simplify
$T_1(n) = 7n^3+10n^2+n+18$
So, $O(n^3)$

Algorithm-2

Step	Cost of each execution	Total # of times executed
1	1	1
2	1	n+1
3	1	n
4	1	$\sum_{i=P}^{n} (n - i + 2)$
5	6	$\sum_{i=P}^{n} (n - i + 1)$
6	8	$\sum_{i=P}^{n} (n - i + 1)$
7	1	1

Multiply col.1 with col.2, add across rows and simplify
$T_2(n) = 15n^2+2n+19$
So, $O(n^2)$

Algorithm-3

Step	Cost of each execution	Total # of times executed in any single recursive call
1	3	1
2	7	1
Steps executed when the input is a base case:1 or 1,2		
First recurrence relation: T(n=1 or n=0) = 10 so O(1)		
3	5	1
4	2	1
5	1	(n+1)/2
6	6	n/2
7	7	n/2
8	2	1
9	1	(n+1)/2
10	6	n/2
11	7	n/2
12	4	1
13	4	(cost excluding the recursive call)T(n/2)
14	5	(cost excluding the recursive call)T(n/2)
15	7	1
Steps executed when input is NOT a base case: 1,2,3,4,5,6,7,8,9,10,11,12,13,14,15		
Second recurrence relation: T(n>1) = 9T(n/2) + 52n/2 + 28		
Simplified second recurrence relation (ignore the constant term): $T(n>1) = n^2$		

Solve the two recurrence relations using any method (recommended method is the Recursion Tree). Show your work below:

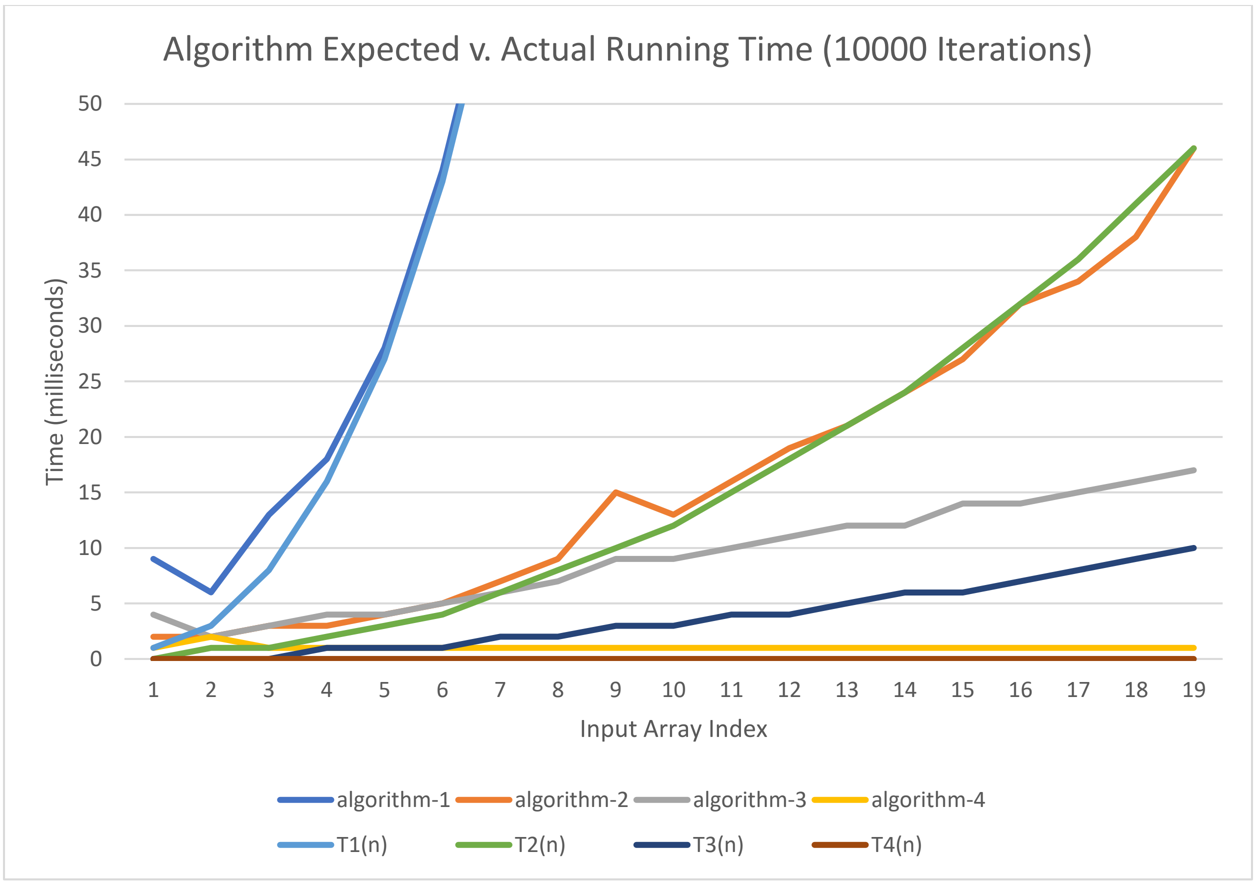
$T_3(n) = n^2$

Algorithm-4

Step	Cost of each execution	Total # of times executed
1	1	1
2	1	1
3	1	n+1
4	10	n
5	6	n
6	1	1

Multiply col.1 with col.2, add across rows and simplify
$T_4(n) = 17n+4$
So, O(logn)

Results:



Explanation:

- Algorithm 1's upper bound of O(n^3) can clearly be seen to follow the expected output once the array size increases past the first few cases. It is by far the slowest algorithm of the 4, so we can only see the results of execution up to Array 6 in a scale that allows us to evaluate the other algorithms.
- Algorithm 2's upper bound of O(n^2logn) can also clearly be seen to follow its estimation line closely. It has some slight variances, but overall, the actual running time shown in the dark orange does not stray far from the green expected time curve.
- Algorithm 3's upper bound of O(n^2) varies the greatest of the 4 algorithms, but overall shows a trend mirroring its estimated output. The gray line of Algorithm 3 runs consistently above the dark blue line of its time estimation, but shows a similar curve and increases at a similar rate. This shows that there are slight discrepancies between the two, which can be attributed to my implementation and language-specific features between the pseudocode and the Javascript implementation. This algorithm uses recursion, unlike the rest, and therefore performs faster than Algorithms 1 and 2.
- Algorithm 4's upper bound of O(logn) follows closely to the bottom of the graph, following a close trend to its time estimation. This algorithm blows the other algorithms out of the water in performance speed. When testing my program, I increased the number of algorithms up to 100,000 just to get this curve further off the ground. Almost all entries for the currently specified number of executions (10,000) are either 1 or 0 for the implemented algorithm and its time estimation. This can be attributed to this algorithm only using one loop over the arrays elements to calculate the max consecutive summation of the array, whereas the others all contain multiple.