Matthew Virgin

Dr. Sudarshan Chawathe

COS 554

25 September 2023

Homework 1

The implementation for my algorithm is largely the same as described in the pseudocode for question 4 of the homework. Some lines have been added to calcPogos() and pogo() to handle edge cases where D(which my code named myPogos for clarity) may have contained 0. The most major addition is the main() function, which handles the user input as well as the output of the program. Other important information can be found in the comments of hw1.py.

To test the runtime of the algorithm across inputs I created hw1testing.py, which is largely the same as hw1.py with a few added lines and print statements that reveal the memory usage and runtime (in seconds) for a given input (n, D). Because the inputs tested can produce extremely large output, hw1testing.py does not produce the lexicographically ordered list of possibilities, listing only the memory usage, input, number of possibilities, and runtime instead. Twenty example inputs representing the worst case where the elements of D range from 1 to n are contained within tst1.txt (i.e the inputs are as follows: (0 [0]) (1 [1]) (2 [1 2]) … (20 [1 2 3 … 20])). The output of running hw1testing.py on tst1.txt is contained within tstout1.txt. The output indicates that after about (5 [1 2 3 4 5]), the runtime doubles for each consecutive input.

(n, [D])	Runtime (seconds)
(0, [0])	0.00002340000000
(1, [1])	0.00001760000000
(2, [1, 2])	0.00002200000000
(3, [1, 2, 3])	0.00002370000000
(4, [1, 2, 3, 4])	0.00003210000000
(5, [1, 2, 3, 4, 5])	0.00004790000000
(6, [1, 2, 3, 4, 5, 6])	0.00008580000000
(7, [1, 2, 3, 4, 5, 6, 7])	0.00016640000000
(8, [1, 2, 3, 4, 5, 6, 7, 8])	0.00033280000000
(9, [1, 2, 3, 4, 5, 6, 7, 8, 9])	0.00071570000000
(10, [1, 2, 3, 4, 5, 6, 7, 8, 9, 10])	0.00143830000000
(11, [1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11])	0.00304890000000
(12, [1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12])	0.00684700000000
(13, [1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13])	0.01366610000000
(14, [1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14])	0.02950020000000
(15, [1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15])	0.06335120000000
(16, [1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16])	0.13620510000000
(17, [1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17])	0.29373620000000
(18, [1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18])	0.62835750000000
(19, [1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19])	1.41786160000000
(20, [1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19, 20])	3.03676130000000

Figure 1: Runtimes

Figure 1 demonstrates a rough doubling in runtime for every input following n = 5. This pattern indicates a “Big O” of O(2^n), or exponential runtime.

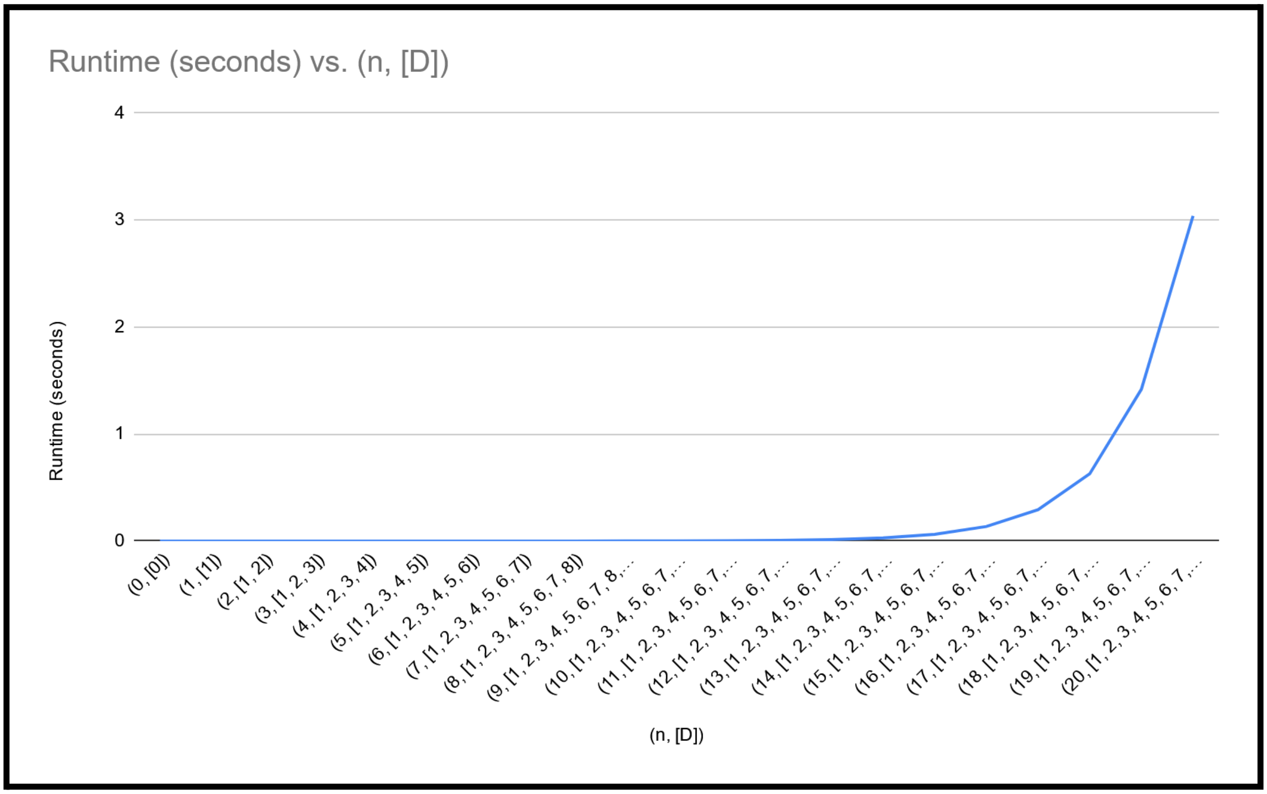

Figure 2: Runtimes Graph

Figure 2 supports the fact that the algorithm is O(2^n), as the curve seen on the chart is indicative of an exponential function. These results would indicate that my runtime estimation for question 5 was incorrect. I had guessed that the runtime would be a summation of the cost of each recursive call, which would not be an exponential function.