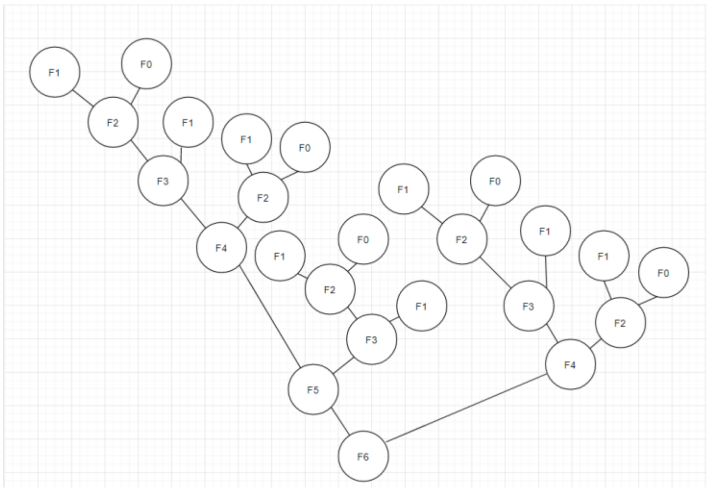
1.

a) Algorithm 1 performs redundant calculations as it only terminates at the base case.

b)

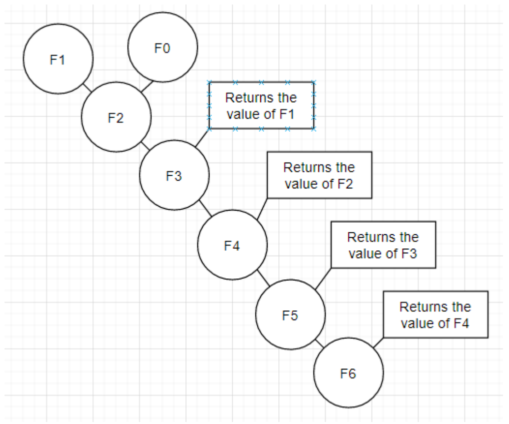


c) For F6 F5 is called once, F4 is called twice, F3 is called three times and F2 is called five times.

2.

a) Algorithm 2 doesn’t perform redundant calculations as it calculates each branch once and than returns the value of the branch.

b)



c) For F6 F5 is called once, F4 is called once, F3 is called once and F2 is called once.

3. Algorithm 2 is more cost efficient as it only calls each branch once and avoids redundant calculations while Algorithm 1 repeats calculations already made.

4. It is O(n) because when we look at our code and count the operations, we arrive at the equation from our code we can get the equation 7n + 6. The 7 in front of n comes from needing to compare, incrementing k and the math inside the loop itself. The constant comes form defining our array, setting an array element to zero, comparisons that are made, setting k and returning. We find our c by adding our constants to that $7n+6 \leq 13n$ when $n \geq n_0 = 1$.

5. It is O(n) because when we look at our code and count the operations, we arrive at the equation from our code we can get the equation 5n + 7. The 6 in front of n comes from needing to compare, incrementing k and the switching of variables. The constant comes from defining variables, our base

cases, initial loop setup and returning. We find our c by adding our constants so that $5n+7 \leq 12n$ when $n \geq n_0 = 1$.

6. Algorithm 4 is more cost efficient as it doesn't require retrieving values from an array and instead relies on switching local variables. We also see this from our O(n). While both are linear algorithm 4 has a lower constant meaning asymptotically it is more efficient than algorithm 3.

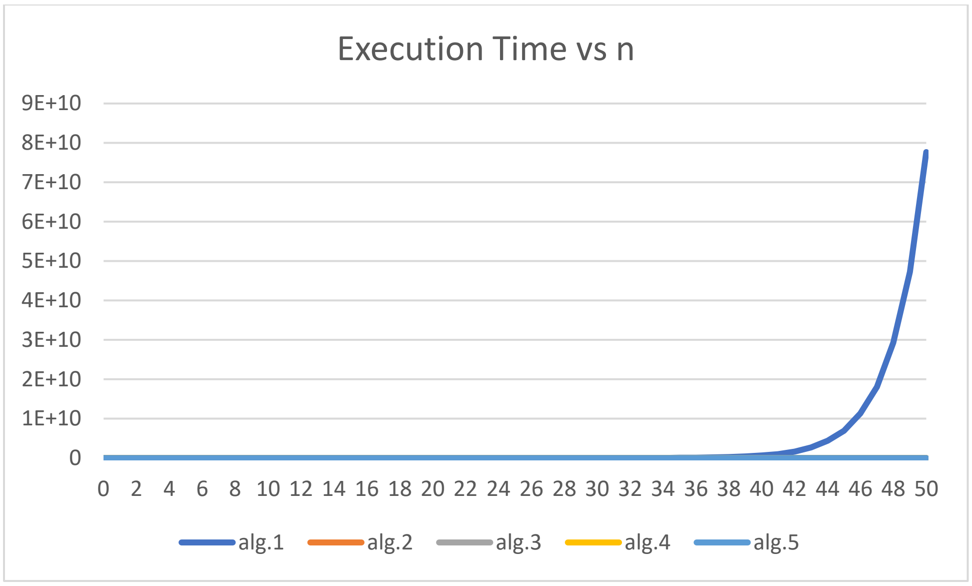
7. It is log(n) as n is halved every time. We can also see this relationship from graph 5 as running time increases as n increases making it not constant but it doesn't increase linearly. From the code we can get the algorithm

$$f(n) = \begin{cases} 1 & \text{if n} = 1 \\ 27 + f(n/2) & \text{n} \geq 1 \end{cases}$$

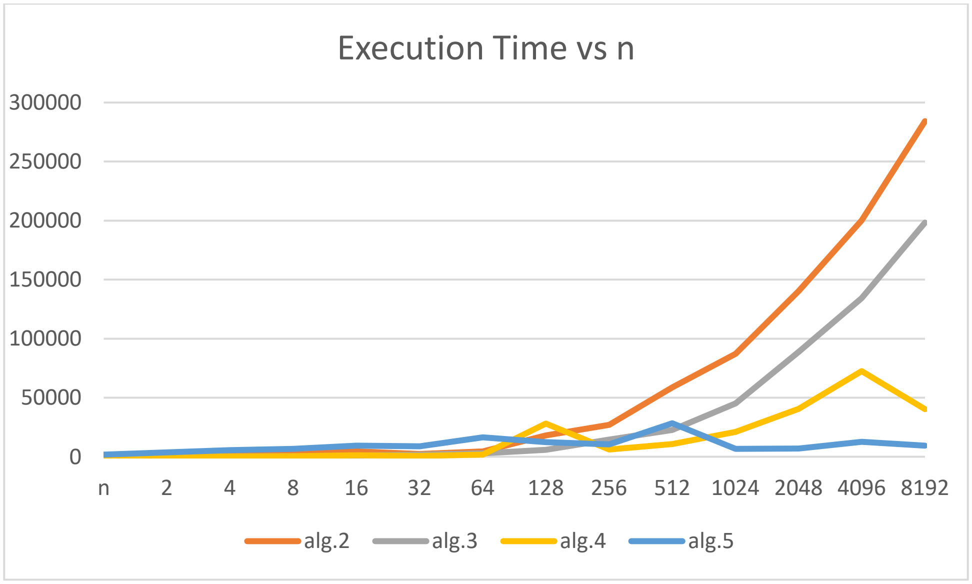
where 27 comes from the matrix multiplication and switching values in the arrays. To solve we try to get our base case. We use substitution by subbing f(n/2) = 27 + f(n/4) into f(n) = 54 + f(n/4). We than sub f(n/4) = 27 + f(n/8) to get f(n) = 81 + f(n/8). After our substitutions we can derive the pattern $f(n) = 2k + f(n/2^k)$ where k is an integer greater than 1. To solve for our base case, we have $n/2^K = 1$ solving for k we see that k = log(n). When we sub back into our f(n) we derive the equitation f(n) = 27log(n) + 1. We find our c by adding our constants so that $27\log(n) + 1 \leq 28\log(n)$ when $n \geq n_0 = 2$.

8. I would use algorithm 1 for very small value of n as any larger value becomes to inefficient for the algorithm to calculate. Algorithm 2 while being more efficient begins encountering stack overflow errors making it unable to calculate values above n roughly around 32768 as it still relies on recursion. Algorithm 3 is more efficient than both and can be used efficiently for all values of n. Algorithm 4 is highly efficient at both a high and lower values of n in comparison to the other above. Asymptotically algorithm 5 is the most efficient meaning it is the best for calculating large values of n. However, it does have a higher constant than algorithm 4 meaning that for smaller values of n algorithm 4 is more efficient. Mathematically from our O(n) equations around n = 6 algorithm 5 becomes more efficient. However, from graph 2 it seems that around n=512 algorithm 5 begins to always be more efficient. Therefore, base off our timing execution to make the most efficient algorithm I would use algorithm 4 for n less than 512 and algorithm 5 for any higher values.

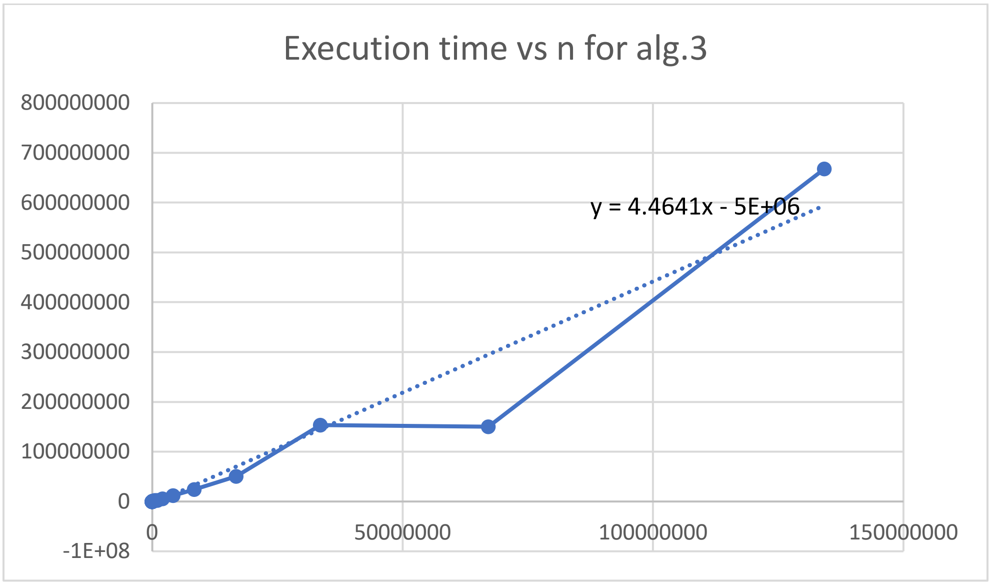
Graph 1



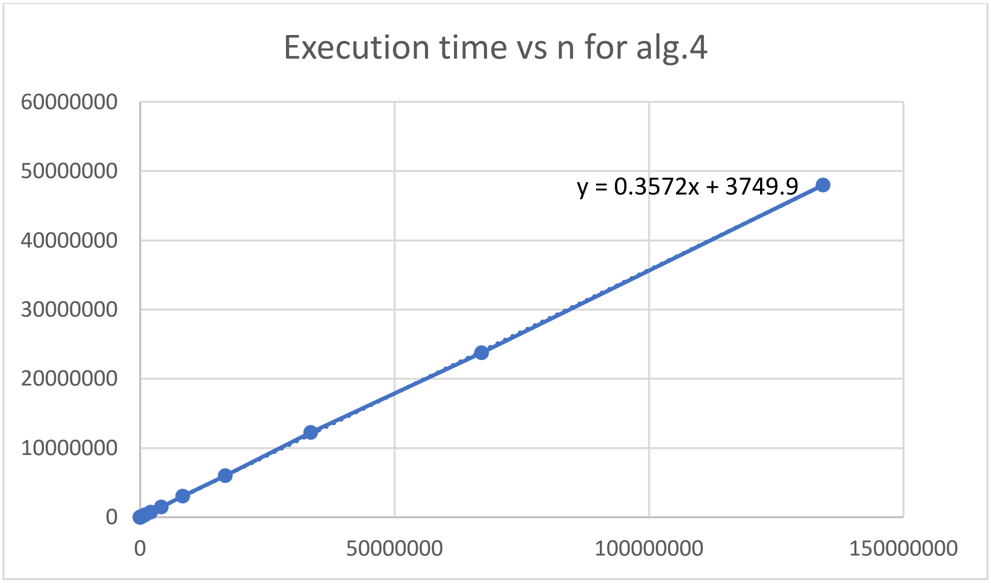
Graph 2



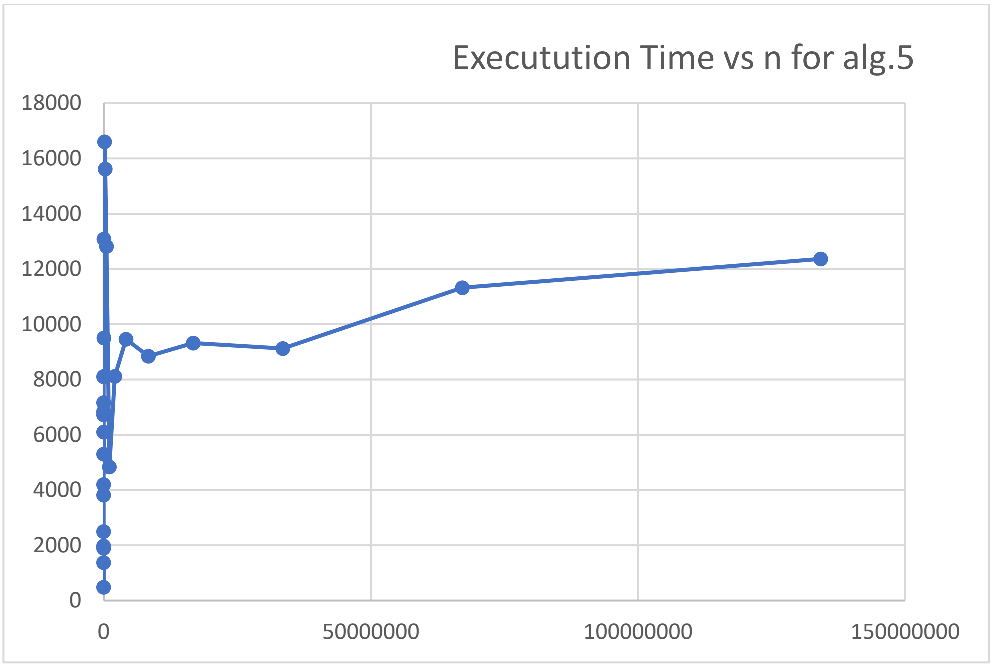
Graph 3



Graph 4



Graph 5


Executution Time vs n for alg.5
18000
16000
14000
12000
10000
8000
6000
4000
2000
0
0
50000000
100000000
150000000