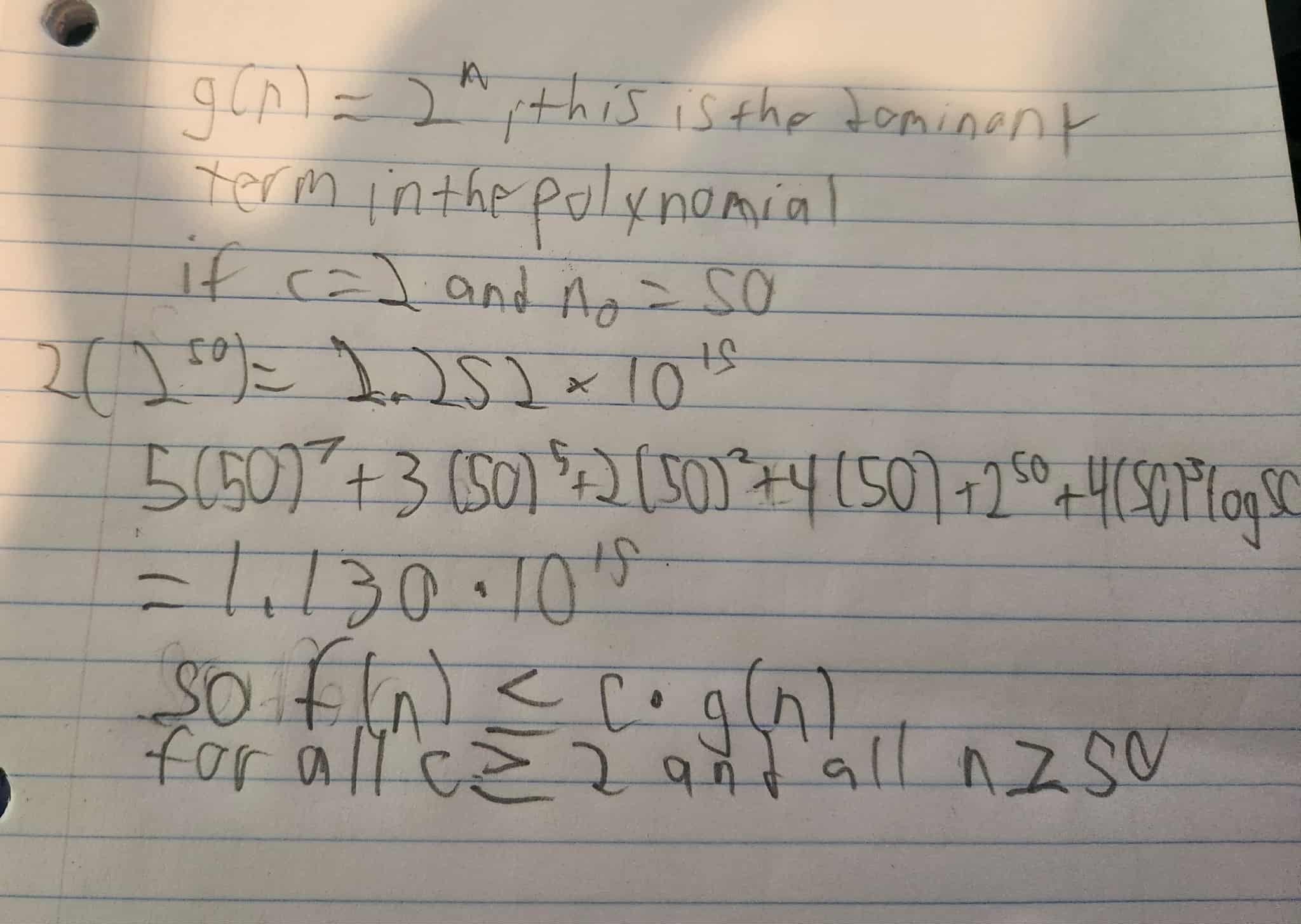
Problem 1:

(from lowest to highest)

1. 2^10 (O(1))
2. 2^log(n) (this is equal to n) (O(n))
3. 4n (O(n))
4. 3n + 100logn (O(n))
5. nlogn (O(nlogn))
6. 4nlogn + 2n (O(nlogn))
7. n^2 +10n (O(n^2))
8. n^3 (O(n^3))
9. 2^n (O(2^n))

Problem 2:

1. O(2^n)



1. O(n^d). n^d will always be the dominant term in the polynomial.

|  |  |
| --- | --- |
| C1 | n |
| C2 | n |
| C3 | n |
| C4 | n^2 |
| C5 | n^2 |
| C6 | n^2 |
| C7 | n |

C1-C3 and C7, in the outer loop, will have to be done one time for each item in the array as it is iterated through.

C4-C6, in the nested loop, will have to be done n^2 times, as the nested loop iterates through the complete array once for each item in the array.

A close up of writing on a piece of paper

AI-generated content may be incorrect.

Problem 3:

3.1:

**Algorithm** RecursiveSort(array, k, l 🡨 0, r 🡨 len(array) - 1):

**if** l > r:

**then** **return** array

{base case}

**if** array[l] <= k:

**then** return RecursiveSort(array, k, l +1, r)

{item is on the correct side, increment the left pointer}

**else**:

{item is on the wrong side}

array[l],array[r] 🡨 array[r],array[l]

{switch the item with the item at the right pointer}

return RecursiveSort(array, k, l, r-1)

{decrement the right pointer}

3.2

Algorithm IterativeSort(array, k)

l 🡨 0

r 🡨 len(array) – 1

{initialize pointers}

**while** l < r **do :**

{same as recursive solution, just using a while loop instead of recursion}

**if** array[l] <= k:

**then** l 🡨 l - 1

{item is on the correct side, just move the left pointer}

**else:**

{item is on the wrong side, switch it to the other side and move the

right pointer}

array[l],array[r] 🡨 array[r], array[l]

r 🡨 r-1

**return** array

3.3:

For both algorithms, the big O runtime is O(n). Worst case the list will have to be iterated through completely once in both algorithms, so the worst case runtime is the length of the list.

3.4: