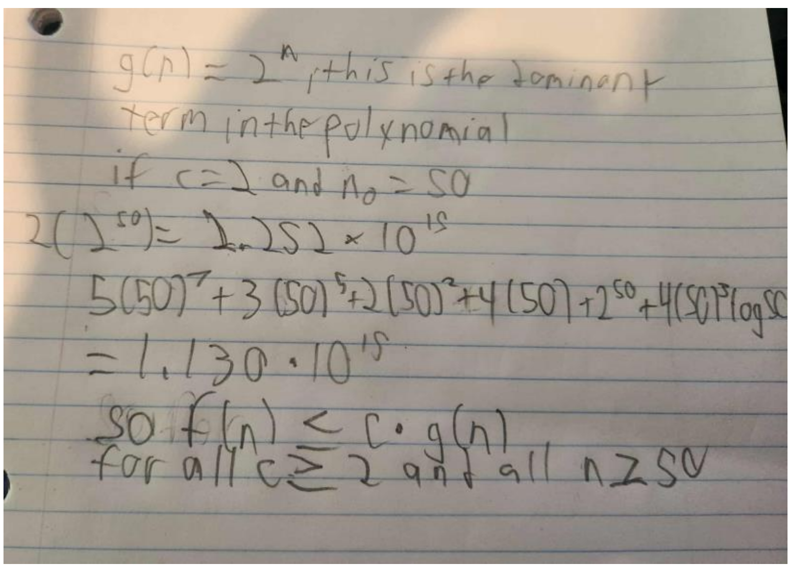
Problem 1:

(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:

A. O(2^n)



$g(n) = 2^n$, this is the dominant term in the polynomial

if $c=2$ and $n_0 = 50$

$2(2^{50}) = 2.252 \times 10^{15}$

$5(50)^7 + 3(50)^5 + 2(50)^2 + 4(50) + 2^{50} + 4(50)^3 \log 50$

$= 1.130 \cdot 10^{15}$

so $f(n) \leq c \cdot g(n)$

for all $c \geq 2$ and all $n \geq 50$

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

C.

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.

2. $f(n) = 3n^2 + 4n$
3. $g(n) = n^2$, the dominant term

For every $c \geq 5$ and $n_0 \geq 1$

$f(n) \leq c \cdot g(n)$

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:

Algorithm Name	Algorithm Type	Input Array Size	Algorithm runtime
RecursiveSort Runs in O(n)	recursive		
		10	0.0
		100	0.0
		500	0.001
IterativeSort	iterative		
		10	0.0
		100	0.0
		500	0.0

4.1:

Algorithm recursive_sum(array, k, l ← 0,r ←len(array) - 1):

if l > r:

then return False

{base case 1, the pointers have crossed and there is no match)

else:

if array[l]+array[r] = k:

then return array[l], array[r]

{base case 2, a sum is found. return the correct pair}

if array[l]+array[r < k:

then return recursive_sum(array, k, l +1, r)

{the sum is too small, move the left pointer right to increase the sum}

else:

{the sum is too big, so move the right pointer left to decrease the sum}

return recursive_sum(array,k,l,r-1}

4.2:

Algorithm iterative_sum(array,k):

l ← 0

r ← len(array) -1

while l < r **do:**

if array[l]+array[r] = k:

then return array[l], array[r]

{ a sum is found. return the correct pair}

if array[l]+array[r < k:

then l+1

{the sum is too small, move the left pointer right to increase the sum}

else:

{the sum is too large, so move the right pointer left to decrease the sum}

r+1

return False

{if no sum is found before the loop exits none exists, return false}

4.3:

Again, for both algorithms the runtime is O(n). both methods need to iterate through the list once completely worst-case to find a solution.

4.4:

Algorithm Name	Algorithm Type	Input Array Size	Algorithm runtime
RecursiveSum Runs in O(n)	recursive		
		10	0.0
		100	0.0
		500	0.0
IterativeSum	iterative		
		10	0.0
		100	0.0
		500	0.0