1.a)

Function(int[] A, int targetValue)

Start

Set counter = 0

For each element i in arr:

If i equals targetValue:

Increment counter by 1

If counter equals 4:

Print "The value was found exactly 4 times"

Else:

Print "The value was found counter times", where counter is the value of counter

End

1.b) the time complexity is O(n)

1.c) O(n)

1.d) No having a sorted array would not improve the time complexity, because for example if the 4 occurrences of the value x are at the end of the array. The program would still need to traverse through the entirety of the array of size n, which would take O(n) time in the worst case.

2.a) n5 logn is O(n7), because in the polynomial (n7)(n2) the n2 term grows much faster than the logn term in the polynomial n5logn. N5logn is indeed less than n7 as n becomes large.

2.b)f(n)=10n2 + 5n4 5000000n3 + n is Θ(n4) by proving that the function grows as fast as n4. To prove this, the function must have an upper bound of O(n4) and a lower bound of Ω(n4)

The dominant term in f(n) is 5n4. The dominant term grows no faster than cn4 for some constant c. As n approaches infinity the function behaves like 5n4 meaning that f(n) cannot grow faster than cn4.

Since 5n4 is the dominant term the function will grow at least as fast as n4 as n approaches infinity. Therefore f(n) >= cn4 for some constant

This shows that 10n2 + 5n4 5000000n3 + n is Θ(n4)

2.c) f(n)= nn is not O(n!) because for this to be true f(n) must grow no faster than n!. Even though factorial growth is fast, exponential grows faster.

For example, as n-> inf:

|  |  |  |
| --- | --- | --- |
| n | nn | n! |
| 3 | 27 | 6 |
| 4 | 256 | 24 |
| 10 | 10,000,000,000 | 3,628,800 |

As n grows, the gap between nn and n! becomes larger, exponential functions like nn grow faster than factorial functions.

2.d) f(n)= 0.01n2+0.0000001n4 is not Θ(n2 ) because f(n) must grow no faster or slower than n2.

To prove this, we must evaluate the dominant term in the function. When evaluating the function as n grows larger, the term 0.0000001n4 will be the dominant term in the function as the overall function will grow like n4. As n-> inf, the term n4 will grow much faster than n2. Therefore, the function cannot be bounded by n2. This means that f(n) is not Θ(n2).

2.e) 1000000n2 + 0.0000001n5 is Ω(n3) because f(n) grows at least as fast as n3. To prove this, we must analyze the dominant term in the function as n -> inf. The term with n5  controls the growth rate f the entire function. When analyzing n5 we can see that it will grow much faster n3 which satisfies the requirement for Ω(n3)

2.f) n! is not O(4n) because n! grows faster than 4n for large n. While both grow quickly, factorial growth outpaces exponential growth with base 4.

For example, for n=3,4 & 10:

|  |  |  |
| --- | --- | --- |
| n | 4n | n! |
| 3 | 64 | 6 |
| 4 | 256 | 24 |
| 10 | 1,048,576 | 3,628,800 |

As n grows larger, n! will outpace 4n thus disproving that n! is O(4n)

3.a)

|  |  |  |
| --- | --- | --- |
| Done | J | a |
| T | 0 | 3,10,5,2,1 |
| F | 1 | 3,5,10,2,1 |
| F | 2 | 3,5,2,10,1 |
| F | 3 | 3,5,2,1,10 |
| F | 4 | 3,5,2,1,10 |
| F | 3 | 3,5,1,2,10 |
| F | 2 | 3,1,5,2,10 |
| F | 1 | 1,3,5,2,10 |
| T | 0 | 1,3,5,2,10 |
| T | 1 | 1,3,5,2,10 |
| F | 2 | 1,3,2,5,10 |
| F | 3 | 1,3,2,5,10 |
| F | 4 | 1,3,2,5,10 |
| F | 3 | 1,3,2,5,10 |

Resulting A=[1,2,3,5,10]

b)

Big O(n2):

For each forward or backward pass the algorithm performs O(n) comparisons and possibly O(n) swaps. For one full cycle of a forward and backward pass it is O(n) + O(n) = O(2n) = O(n). The key part is that the algorithm will possibly repeat itself in the worst-case n times because in each pass , the algorithm may only move one element either largest or smallest once. The worst case being the array is completely unsorted. Therefore the time complexity will be O(n) x O(n) = O(n2)

Big Omega(n):

The algorithm is bigOmega(n) because in the best case scenario the array is already completely sorted, and the algorithm will only need to do one cycle of a forward and backward pass with a time complexity of O(n)+O(n) = O(2n) = O(n). Since the array is already sorted no additional passes are required after the first cycle.

4.

Function(input \_string)

Convert input\_string to char array and call it input

Initialize an empty string 'output'

Set 'currentChar' to the first character in 'input'

Initialize 'counter' to 1

For each character 'nextChar' in 'input', starting from index 1:

If 'nextChar' is equal to 'currentChar':

Increment 'counter'

Else:

Call appendCharWithCount(output, currentChar, counter)

Set 'currentChar' to 'nextChar'

Reset 'counter' to 1

Call appendCharWithCount(output, currentChar, counter) # Append the last character

Print 'output'

End Function

Function appendCharWithCount(output, character, count):

Append 'character' to 'output'

If 'count' is greater than 1:

Append 'count' to 'output'

End Function

1. The algorithm processes each character in the string exactly once and it iterates from the beginning to the end. The loop runs from 1 to n-1 and each iteration involves constant time operations: comparing characters, counter increment and appending. The constant time operations are O(1). Since each iteration of the loop is O(1) and the loop runs n times, where n is length of the string, O(n) x O(1) = O(n). The time complexity of my algorithm is O(n).
2. The space complexity is O(n). To Prove this we must analyze the different variables being stored in memory. The input array of size n, which requires O(n) space where n is the length of the input string. The String builder output stores the processed version of the input string and in the worst case will be the same length as the input string, so the space required is O(n). The helper variables like the counter and char are all constant in terms of space, so those are O(1). The total space complexity is O(n) because the algorithm requires space proportional to the size of the input and output string.

5.

Function(string[] array)

Set 'arrLength' to the length of the array

// Define boundaries for the left and right halves

Set 'endLeft' to arrLength / 2

Set 'endRight' to endLeft if arrLength is even, otherwise set 'endRight' to endLeft + 1

Initialize 'left' to 0

Initialize 'right' to arrLength - 1

// Loop through the array and perform operations

While 'left' is less than or equal to endLeft - 1 and 'right' is greater than or equal to endRight + 1:

If array[left] equals array[left + 1]:

Call Negate(array, left, left + 1)

Call reverse(array, right, right - 1)

// Move the left and right pointers

Increment 'left' by 2

Decrement 'right' by 2

Print the modified 'array'

Function reverse(array, leftIndex, rightIndex):

Swap array[leftIndex] and array[rightIndex]

End Function

Function Negate(array, i, j):

Multiply array[i] and array[j] by -1

End Function

1. The time complexity of my algorithm is **O(n)**. To demonstrate this, we need to analyze the loop and the operations inside it. The loop uses two pointers that start at opposite ends of the array and move towards the middle, processing two elements per iteration. This ensures that the loop traverses the array exactly once, resulting in **O(n)** iterations, where n is the length of the array. Within each iteration of the loop the operations are comparisons, increments, swaps and arithmetic operations. All of these operations are constant time O(1). Since the loop iterates n times, and the operations are of O(1), the total time complexity of the algorithm is O(n) x O(1) = O(n)
2. The Space complexity of my algorithm is O(1). It uses a constant amount of extra memory regardless on the input size. The algorithm uses only a few auxiliary variables like left, right endLeft and endRight, which are all constant space variables. The helper functions also do not require and space as they work directly on the array. Since my algorithm does not allocate any extra space, the space complexity is O(1).