Time and space complexity

**Bubble Sort:**

The time complexity of the bubble sort algorithm is O(n^2) because it uses nested loops to traverse the array, and the worst-case scenario occurs when the array is sorted in reverse order. The space complexity is O(1) because it only requires a temporary variable to swap elements.

**Selection Sort:**

The time complexity of the selection sort algorithm is O(n^2) because it uses nested loops to traverse the array, and the worst-case scenario occurs when the array is sorted in reverse order. The space complexity is O(1) because it only requires a temporary variable to swap elements.

Insertion Sort:

The time complexity of the insertion sort algorithm is O(n^2) because it uses nested loops to traverse the array, and the worst-case scenario occurs when the array is sorted in reverse order. The space complexity is O(1) because it only requires a temporary variable to swap elements.

**Minimum and maximum**

The time complexity of the getmin and getmax functions is O(n), where n is the length of the input list. This is because the function iterates over each element in the list once to find the minimum or maximum value.

The time complexity of the main program is also O(n), as it creates a list of length n and calls the getmin and getmax functions once each, both of which have a time complexity of O(n). Therefore, the overall time complexity of the program is O(n).

The space complexity of the program is also O(n), as it creates a list of length n to store the input values. The getmin and getmax functions do not create any additional data structures that depend on the input size, so their space complexity is constant. Therefore, the overall space complexity of the program is O(n).

**Infix postfix**

The time complexity of the infix\_to\_postfix function is O(n), where n is the length of the input expression. This is because each character in the expression is processed once and is either appended to the output string or pushed or popped from the stack.

The space complexity of the function is also O(n), as the size of the stack could be up to n characters long in the worst case, and the output string will also be n characters long.

**Linear and bimnary search :**  
Time and space complexity for each function are as follows:

LinearSearch:

Time complexity: O(n) (linear time)

Space complexity: O(1) (constant space)

BinarySearch (iterative):

Time complexity: O(log n) (logarithmic time)

Space complexity: O(1) (constant space)

BinarySearch (recursive):

Time complexity: O(log n) (logarithmic time)

Space complexity: O(log n) (logarithmic space, due to the recursive calls on the stack)

**Linear based stack operation:**

The time and space complexity of the given code is as follows:

Time Complexity:

The append() function to push an element onto the stack has a time complexity of O(1).

The pop() function to remove an element from the stack has a time complexity of O(1).

The loop that pops all elements from the stack has a time complexity of O(n), where n is the number of elements in the stack.

Therefore, the overall time complexity of the code is O(n).

Space Complexity:

The space complexity of the code depends on the number of elements that are pushed onto the stack.

In the given code, three elements are pushed onto the stack, so the space complexity is O(3) or simply O(1).

**Longest comman sequesce:**

The time complexity of the given code is O(mn), where m and n are the lengths of the input strings s1 and s2, respectively. This is because the algorithm uses a nested loop that iterates over all possible combinations of characters from the two strings, and performs a constant amount of work at each iteration.

The space complexity of the code is also O(mn), because it uses a 2D array of size (m+1) x (n+1) to store the lengths of the longest common subsequence between prefixes of the input strings. This array requires O(mn) space to store.

**Multiplication of matrix:**

Time complexity:

The initialization of matrices takes O(RC + R1C1) time.

The matrix multiplication takes O(RC1C) time.

Printing the matrices and the result takes O(RC + R1C1 + R\*C1) time.

Therefore, the overall time complexity of the program is O(RC + R1C1 + RC1C).

Space complexity:

The space required to store the matrices is O(RC + R1C1).

The space required to store the result is O(R\*C1).

Therefore, the overall space complexity of the program is O(RC + R1C1 + R\*C1).

**Maximum and minimum element**

The time complexity of both functions is O(nlogn) because of the sorting operation used to find the minimum and maximum elements. The space complexity is O(n) as we are storing the list of elements in memory.

**Find pattern:**  
The time complexity of the brute\_force\_search function is O(n \* m), where n is the length of the string and m is the length of the pattern. In the worst case, it may need to compare all characters of the string with the pattern.

The space complexity of the brute\_force\_search function is O(1), as it only uses a constant amount of additional memory regardless of the size of the inputs.

**Starseens matrix:**

The time complexity of Strassen's algorithm is O(n^log7) where log7 is approximately 2.81. However, due to the high constant factors involved in the algorithm, the traditional matrix multiplication algorithm is more efficient for small matrix sizes.

The space complexity of the algorithm is O(n^2), as it creates new matrices for each recursive call.

**Sum of row column digonal element:**

The time complexity of this program is O(RC) because it involves two nested loops that iterate over all the elements in the matrix. The space complexity is O(RC) because the matrix is stored in memory as a 2-dimensional list.

**Sum of two matrices:**

The time complexity of the given code is:

For the matrix input from the user, the code has two nested loops. The outer loop runs R times and the inner loop runs C times. Therefore, the time complexity of this code segment is O(R\*C).

For the matrix addition, the code has two nested loops. The outer loop runs R times and the inner loop runs C times. Therefore, the time complexity of this code segment is also O(R\*C).

Finally, for printing the resultant matrix, the code has two nested loops. The outer loop runs R times and the inner loop runs C times. Therefore, the time complexity of this code segment is also O(R\*C).

Therefore, the overall time complexity of the given code is O(R\*C).

The space complexity of the given code is:

For storing the input matrix, the code uses a two-dimensional list of size RxC, which requires R\*C memory.

Similarly, for storing the second input matrix, the code uses another two-dimensional list of size R1xC1, which requires R1xC1 memory.

For storing the resultant matrix, the code uses another two-dimensional list of size RxC, which requires R\*C memory.

Therefore, the total space complexity of the given code is O(RC + R1C1).

Note that in the worst-case scenario, if R=C=R1=C1=N, then the time complexity is O(N^2) and the space complexity is O(N^2).

**Tower of henoi:**

Time and space complexity of factorial function:

Time complexity: O(n)

The function involves a single loop which iterates n times, so the time complexity is linear in n.

Space complexity: O(n)

The function uses the call stack to perform recursion, so the space required in the stack is proportional to n. In the worst case, the maximum depth of recursion will be n, so the space complexity is also linear in n.

Time and space complexity of fibonacci function:

Time complexity: O(2^n)

The function involves a recursive call to itself twice in each recursive call, so the number of function calls grows exponentially with n. Therefore, the time complexity is exponential in n.

Space complexity: O(n)

The function uses the call stack to perform recursion, so the space required in the stack is proportional to n. In the worst case, the maximum depth of recursion will be n, so the space complexity is also linear in n.

Time and space complexity of tower\_of\_hanoi function:

Time complexity: O(2^n)

The function involves a recursive call to itself twice in each recursive call, so the number of function calls grows exponentially with n. Therefore, the time complexity is exponential in n.

Space complexity: O(n)

The function uses the call stack to perform recursion, so the space required in the stack is proportional to n. In the worst case, the maximum depth of recursion will be n, so the space complexity is also linear in n.

Time and space complexity of iterative\_factorial function:

Time complexity: O(n)

The function involves a single loop which iterates n times, so the time complexity is linear in n.

Space complexity: O(1)

The function does not involve recursion, so the space required in the stack is constant. Only a constant amount of extra space is required to store the result and the loop variables.

Time and space complexity of iterative\_fibonacci function:

Time complexity: O(n)

The function involves a single loop which iterates n times, so the time complexity is linear in n.

Space complexity: O(1)

The function does not involve recursion, so the space required in the stack is constant. Only a constant amount of extra space is required to store the variables used in the loop.

**Counting no of even and odd elements:**  
The time complexity of the function is O(n) because it has a single for loop that iterates through the entire list.

The space complexity of the function is O(1) because it uses only a fixed amount of memory regardless of the size of the input array.

**Sum of elements in array:**  
  
Time complexity: O(n), where n is the size of the input array. The for loop iterates n times, and the operations inside the loop take constant time.

Space complexity: O(n), where n is the size of the input array. The array "lst" takes n units of space.

**To find palindrome:**

The time complexity of the given code is O(n), where n is the length of the input string 'Str'. This is because the code iterates over the string once, performing constant time operations at each iteration.

The space complexity of the code is O(1), since the amount of memory used by the program is independent of the input size. The program only stores a fixed number of variables and sets throughout the execution, regardless of the size of the input string.