1.

A)

This algorithm takes as input a graph graph, which is represented as a dictionary where the keys are the nodes of the graph and the values are the lists of neighbors of each node. The function returns a list containing the topological ordering of the graph.

DFS-based algorithms for obtaining an ordering for a given DAG (directed acyclic graph) involve traversing the graph using the DFS (depth-first search) algorithm and ordering the nodes based on the order in which they are visited. This approach is useful because it can help to identify the topological order of the nodes in the graph, which is the order in which the nodes must be processed in order to ensure that all dependencies between them are satisfied.

The worst-case time complexity of a DFS-based algorithm for obtaining an ordering for a given DAG is O(V + E), where V is the number of nodes in the graph and E is the number of edges. This is because the DFS algorithm has a time complexity of O(V + E) in the worst case, and the additional step of ordering the nodes based on the order in which they are visited does not affect the overall time complexity of the algorithm.

In summary, DFS-based algorithms for obtaining an ordering for a given DAG are useful because they can help to identify the topological order of the nodes in the graph. The worst-case time complexity of these algorithms is O(V + E).

B)

1. Initialize an empty list **order** and a set **visited** that will keep track of visited nodes.
2. Pick a starting node, let's say **node1**, and add it to **order**. Mark **node1** as visited.
3. Consider all the nodes that can be reached from **node1** by following the directed edges in the graph. For each such node, let's say **node2**, do the following:
    - If **node2** has not been visited yet, add it to **order** and mark it as visited.
    - If **node2** has been visited already, skip it.
4. Repeat steps 3 and 4 until all nodes have been visited and added to **order**.

    Here is some sample code that implements this algorithm in Python:

```
def topological_ordering(graph):
  order = []  # list to store the topological ordering
  visited = set()  # set to keep track of visited nodes

  # Function that recursively traverses the graph starting from a given node
  def traverse(node):
```

```
  # If the node has not been visited yet, add it to the order and mark it as visited
  if node not in visited:
    order.append(node)
    visited.add(node)

    # Consider all nodes that can be reached from the current node
    for neighbor in graph[node]:
      traverse(neighbor)

# Start the traversal from all nodes in the graph, one by one
for node in graph:
  traverse(node)

return order
```

This algorithm has a time **complexity** of O(V + E), where V is the number of nodes and E is the number of edges in the graph. This is because each node and edge is visited and processed at most once. The code in python file.

2.

One possible algorithm to calculate an, where a and n are integers, with a worst-case time complexity of O(log n) is the exponentiation by squaring method. This algorithm works by repeatedly squaring the result and multiplying it by a whenever the exponent n is odd.

Here is the pseudo-code for the algorithm:

```
function pow(a, n)
 if n == 0
   return 1
  if n % 2 == 1
      return pow(a, n - 1) * a
   b = pow(a, n / 2)
   return b * b
```

This algorithm has a time **complexity** of O(log n) because each recursive call to pow() reduces the value of n by at least half, so the number of recursive calls is at most log n. Additionally, the multiplications and divisions performed in the algorithm are O(1) operations, so the overall time complexity of the algorithm is O(log n).

Another possible algorithm to calculate an with a time complexity of O(log n) is the binary exponentiation algorithm, which uses a similar approach but performs the calculations using binary representation of the exponent n. The code in python file.

3.

Exhaustive search is a method of solving a problem by trying every possible solution until the correct one is found. This can be used to solve a 9x9 Sudoku puzzle by trying every possible value for each cell, and checking whether the resulting puzzle is valid (i.e. it satisfies all of the constraints). The code in python file.

4.

Insertion Sort

To sort the given array using insertion sort, we would first compare the first two elements, 6 and 8. Since 8 is greater than 6, we don't need to do anything. Then, we compare the next element, 9, to the previous two elements. Since 9 is greater than both 6 and 8, we don't need to do anything. Next, we compare the next element, 8, to the previous three elements. Since 8 is already in the correct position relative to 9, we don't need to do anything. Then, we compare the next element, 3, to the previous four elements. Since 3 is less than 8, we need to move 8 to the right and insert 3 in its place. The array now looks like this: {3,6,8,9,8,12}.

Next, we compare the next element, 3, to the previous five elements. Since there is already a 3 in the array, and since insertion sort is a stable sorting algorithm, the 3 that was originally in the array will stay in its position relative to the other elements. The array now looks like this: {3,3,6,8,9,12}. Finally, we compare the last element, 12, to the previous six elements. Since 12 is greater than all of the other elements, we don't need to do anything. The final sorted array is: {3,3,6,8,8,9,12}.

Insertion sort is a stable sorting algorithm because it maintains the relative order of elements with the same value. In other words, if there are two or more equal elements in the array, their order will be preserved after sorting.

The worst-case time complexity of insertion sort is O(n^2), which occurs when the array is in reverse order.

Quick Sort

To sort the given array using quick sort, we would first choose a pivot element. This element could be any of the elements in the array, but for the sake of this example, let's say we choose the element 9. We then move all of the elements that are less than 9 to the left of 9, and all of the elements that are greater than 9 to the right of 9. The array now looks like this: {6,8,3,3,8,12,9}.

Next, we apply the quick sort algorithm to the left and right subarrays, {6,8,3,3,8} and {12}. We again choose a pivot element for each subarray. For the left subarray, we choose 8 as the pivot element, and for the right subarray, we choose 12 as the pivot element. We then move all of the elements that are less than the pivot element to the left of the pivot element, and all of the elements that are greater than the pivot element to the right of the pivot element. The array now looks like this: {3,3,6,

Quick sort is not a stable sorting algorithm because it does not maintain the relative order of elements with the same value. In this example, the two 3s were originally next to each other in the array, but after sorting, they are not next to each other.

The worst-case time complexity of quick sort is O(n^2), which occurs when the array is already sorted in ascending or descending order.

To sort the given array using bubble sort, we would first compare the first two elements, 6 and 8. Since 8 is greater than 6, we swap their positions to get {8,6,9,8,3,3,12}. Then, we compare the next two elements, 6 and 9.

Bubble Sort

To sort the given array using bubble sort, we would first compare the first two elements, 6 and 8. Since 8 is greater than 6, we swap their positions to get {8,6,9,8,3,3,12}. Then, we compare the next two elements, 6 and 9. Since 9 is greater than 6, we swap their positions to get {8,9,6,8,3,3,12}. Then, we compare the next two elements, 8 and 6. Since 8 is greater than 6, we swap their positions to get {8,9,8,6,3,3,12}. Then, we compare the next two elements, 8 and 8. Since they are equal, we don't need to do anything. Then, we compare the next two elements, 6 and 3. Since 3 is less than 6, we swap their positions to get {8,9,8,3,6,3,12}.

Next, we compare the next two elements, 3 and 6. Since 6 is greater than 3, we swap their positions to get {8,9,8,3,3,6,12}. Then, we compare the next two elements, 3 and 3. Since they are equal, we don't need to do anything. Then, we compare the next two elements, 6 and 12. Since 12 is greater than 6, we swap their positions to get {8,9,8,3,3,12,6}.

At this point, the largest element in the array, 12, is in the correct position, so we can stop the algorithm. The final sorted array is: {3,3,6,8,8,9,12}.

Bubble sort is a stable sorting algorithm because it maintains the relative order of elements with the same value. In this example, the two 3s were originally next to each other in the array, and after sorting, they are still next to each other.

The worst-case time complexity of bubble sort is O(n^2), which occurs when the array is in reverse order.

5.

a)

As mentioned above, brute force and exhaustive search are closely related concepts in computer science. They both refer to methods of solving problems that involve trying every possible solution until the correct one is found.

The main difference between the two is that brute force is a general term used to describe any method of solving a problem by trying every possible solution, while exhaustive search is a more specific term that refers to algorithms that systematically search through a space of possible solutions.

In other words, exhaustive search is a type of brute force method, but not all brute force methods are exhaustive searches.

Both brute force and exhaustive search can be time-consuming and require a lot of computational power, but exhaustive search algorithms are typically more efficient and can find solutions more quickly than simple brute force methods.

In summary, brute force and exhaustive search are similar in that they both involve trying every possible solution to find the correct one, but exhaustive search algorithms are more efficient and can find solutions more quickly than simple brute force methods.

b)

Caesar's Cipher and AES are two different types of encryption algorithms. Caesar's Cipher is a very simple substitution cipher that replaces each letter in the plaintext with a letter that is a certain number of positions down the alphabet. For example, if the shift value is 3, then the letter "A" would be replaced with the letter "D", the letter "B" would be replaced with the letter "E", and so on.

AES, on the other hand, is a more advanced encryption algorithm that uses a fixed block size and variable key length to encrypt and decrypt data. It is considered to be very secure and is widely used in many applications, including online banking and secure communication.

Both Caesar's Cipher and AES are vulnerable to brute force attacks, which involve trying every possible combination of letters or keys until the correct one is found. In the case of Caesar's Cipher, this is relatively easy to do because there are only 26 possible shift values, so a brute force attack could try all of them in a relatively short amount of time.

AES, on the other hand, is much more difficult to crack using a brute force attack because it uses a much larger key space. However, it is still theoretically possible to crack an AES encryption by trying every possible key, it would just take a much longer time and require a lot more computational power.

In summary, both Caesar's Cipher and AES are vulnerable to brute force attacks, but AES is much more difficult to crack because it uses a larger key space.

c)

The naive solution to primality testing, which involves checking if every number from 2 to n-1 divides n, grows exponentially because the number of steps required to check all of the possible values increases rapidly as the value of n gets larger.

For example, if n is 10, then the naive solution would have to check 9 different values to see if any of them divide n. But if n is 100, then the naive solution would have to check 99 different values, and if n is 1000, then it would have to check 999 different values, and so on.

As the value of n increases, the number of values that need to be checked increases exponentially, making the naive solution to primality testing very inefficient for large values of n.

In summary, the naive solution to primality testing grows exponentially because the number of steps required to check all of the possible values increases rapidly as the value of n gets larger.