Time and space complexity

Time Complexity:

1. Definition: Time complexity measures how the running time of an algorithm grows as the input size increases.
2. It answers the question: "How much longer will my algorithm take if I give it more data?"

Space Complexity:

1. Definition: Space complexity measures how much additional memory an algorithm needs as the input size increases.
2. It answers the question: "How much more memory will my algorithm need if I give it more data?"

Common Time Complexities (from fastest to slowest):

1. O(1) - Constant time (best)
   - Example: Accessing an array element by index
2. O(log n) - Logarithmic time
   - Example: Binary search
3. O(n) - Linear time
   - Example: Linear search
4. O(n log n) - Linearithmic time
   - Example: Efficient sorting (Merge sort, Quick sort)
5. $O(n^2)$ - Quadratic time
   - Example: Nested loops, bubble sort
6. $O(2^n)$ - Exponential time (worst)
   - Example: Recursive Fibonacci

Common Space Complexities:

1. O(1) - Constant space
   - Example: Variables, simple loops
2. O(n) - Linear space
   - Example: Arrays, lists of size n
3. $O(n^2)$ - Quadratic space
   - Example: 2D arrays of size $n \times n$

Let's look at some practical examples:

Example 1:
Time complexity O(1)
Space complexity O(1)
Accessing an element in an array

```
// C++ Example
#include <iostream>
using namespace std;
int main() {
   int arr[] = {10, 20, 30, 40};
   cout << arr[2]; // Access is constant time O(1)
   return 0;
}
// Java Example
public class Main {
   public static void main(String[] args) {
      int[] arr = {10, 20, 30, 40};
      System.out.println(arr[2]); // Access is constant time O(1)
   }
}
```

Example 2:
Time complexity O(log(n))
Space complexity O(1)
Binary search

```
#include <iostream>
#include <vector>
using namespace std;

int binarySearch(vector<int>& arr, int target) {
   int left = 0, right = arr.size() - 1;

   while (left <= right) {
      int mid = left + (right - left) / 2; // Avoid overflow
      if (arr[mid] == target) {
         return mid; // Target found
      } else if (arr[mid] < target) {
         left = mid + 1; // Search right half
      } else {
         right = mid - 1; // Search left half
      }
   }
   return -1; // Target not found
}

int main() {
   vector<int> arr = {2, 4, 6, 8, 10, 12, 14};
   int target = 10;

   int result = binarySearch(arr, target);
```

```
    if (result != -1)
        cout << "Element found at index: " << result << endl;
    else
        cout << "Element not found!" << endl;

    return 0;
}
```

```
import java.util.*;

public class Main {
    public static int binarySearch(int[] arr, int target) {
        int left = 0, right = arr.length - 1;

        while (left <= right) {
            int mid = left + (right - left) / 2; // Avoid overflow
            if (arr[mid] == target) {
                return mid; // Target found
            } else if (arr[mid] < target) {
                left = mid + 1; // Search right half
            } else {
                right = mid - 1; // Search left half
            }
        }
        return -1; // Target not found
    }

    public static void main(String[] args) {
        int[] arr = {2, 4, 6, 8, 10, 12, 14};
        int target = 10;

        int result = binarySearch(arr, target);
        if (result != -1)
            System.out.println("Element found at index: " + result);
        else
            System.out.println("Element not found!");
    }
}
```

Each iteration of the binary search reduces the size of the search space by half.
For an array of size nnn, the maximum number of iterations is log(n).
Thus, the time complexity of binary search is O(logn).

Example 3:
Time complexity O(n)
Space complexity O(1)

Simple Array Sum

```
int sum(int[] arr) {
    int total = 0;                  // Space: O(1) - just one variable
    for(int i = 0; i < arr.length; i++) {  // Time: O(n) - loops through each element once
        total += arr[i];
```

```
  }
  return total;
}
```

Time Complexity: O(n) - linear time, because it processes each element once
Space Complexity: O(1) - constant space, because it only uses one variable regardless of input size

Example 4:
Time complexity O(n)
Space complexity O(n)
Creating a Copy of an Array

```
int[] copyArray(int[] arr) {
  int[] copy = new int[arr.length];     // Space: O(n) - creates new array
  for(int i = 0; i < arr.length; i++) { // Time: O(n) - loops through each element
    copy[i] = arr[i];
  }
  return copy;
}
```

Time Complexity: O(n) - linear time
Space Complexity: O(n) - linear space, because it creates a new array of size n

Example 4
Time Complexity: O(nlog(n))
Space Complexity: O(n)
Merge sort

```
#include <iostream>
#include <vector>
using namespace std;

// Merge two sorted subarrays
void merge(vector<int>& arr, int left, int mid, int right) {
  vector<int> temp(right - left + 1);
  int i = left, j = mid + 1, k = 0;

  while (i <= mid && j <= right) {
    if (arr[i] <= arr[j]) {
      temp[k++] = arr[i++];
    } else {
      temp[k++] = arr[j++];
    }
  }

  while (i <= mid) temp[k++] = arr[i++];
  while (j <= right) temp[k++] = arr[j++];

  for (int p = 0; p < temp.size(); p++) {
    arr[left + p] = temp[p];
  }
}

// Merge sort implementation
```

```
void mergeSort(vector<int>& arr, int left, int right) {
   if (left < right) {
      int mid = left + (right - left) / 2;

      mergeSort(arr, left, mid);        // Sort left half
      mergeSort(arr, mid + 1, right);  // Sort right half
      merge(arr, left, mid, right);    // Merge sorted halves
   }
}

int main() {
   vector<int> arr = {38, 27, 43, 3, 9, 82, 10};
   mergeSort(arr, 0, arr.size() - 1);

   cout << "Sorted Array: ";
   for (int x : arr) {
      cout << x << " ";
   }
   return 0;
}

import java.util.Arrays;

public class Main {
   // Merge two sorted subarrays
   public static void merge(int[] arr, int left, int mid, int right) {
      int[] temp = new int[right - left + 1];
      int i = left, j = mid + 1, k = 0;

      while (i <= mid && j <= right) {
         if (arr[i] <= arr[j]) {
            temp[k++] = arr[i++];
         } else {
            temp[k++] = arr[j++];
         }
      }

      while (i <= mid) temp[k++] = arr[i++];
      while (j <= right) temp[k++] = arr[j++];

      for (int p = 0; p < temp.length; p++) {
         arr[left + p] = temp[p];
      }
   }

   // Merge sort implementation
   public static void mergeSort(int[] arr, int left, int right) {
      if (left < right) {
         int mid = left + (right - left) / 2;

         mergeSort(arr, left, mid);        // Sort left half
         mergeSort(arr, mid + 1, right);  // Sort right half
```

```
            merge(arr, left, mid, right);    // Merge sorted halves
        }
    }

    public static void main(String[] args) {
        int[] arr = {38, 27, 43, 3, 9, 82, 10};
        mergeSort(arr, 0, arr.length - 1);

        System.out.println("Sorted Array: " + Arrays.toString(arr));
    }
}
```

**Explanation of O(nlogn) Complexity**

1. **Divide-and-Conquer**:
    - The array is divided into two halves repeatedly, which takes log(n) steps.
2. **Merging**:
    - In each step, all elements are processed (merged) in O(n) time.
3. **Total Complexity**:
    - O(nlog(n)).

Example 5
Time Complexity: $O(n^2)$
Space Complexity: O(1)
Nested Loops Example

```
void printPairs(int[] arr) {
    for(int i = 0; i < arr.length; i++) {        // Time: O(n²) - nested loops
        for(int j = 0; j < arr.length; j++) {
            System.out.println(arr[i] + "," + arr[j]);
        }
    }
}
```

Time Complexity: $O(n^2)$ - quadratic time, because for each element, it loops through all elements
Space Complexity: O(1) - constant space, uses only loop variables

Example 6
Time Complexity: O(2^n) - exponential time
Space Complexity: O(n) - recursive call stack

```
int fibRecursive(int n) {
    if (n <= 1) return n;
    return fibRecursive(n-1) + fibRecursive(n-2);
}
```