**What is time complexity and space complexity?What is best case, worst case and average  
case time complexity?**

Time complexity is a measure of how much time an algorithm takes to solve a problem as a function of the input size. In other words, it measures how much the running time of an algorithm grows as the input size grows. Time complexity is usually expressed using "Big O" notation, which describes the upper bound of the running time in terms of the input size. For example, an algorithm with a time complexity of O(n) means that its running time grows linearly with the input size.

Space complexity, on the other hand, is a measure of how much memory an algorithm uses to solve a problem as a function of the input size. Space complexity is also expressed using "Big O" notation, which describes the upper bound of the memory usage in terms of the input size. For example, an algorithm with a space complexity of O(n) means that its memory usage grows linearly with the input size.

Both time complexity and space complexity are important factors to consider when designing and evaluating algorithms, as they help to determine whether an algorithm is practical and scalable for a given problem.

Best case, worst case, and average case time complexity are different ways to measure the performance of an algorithm in different scenarios.

Best case time complexity is the time complexity of an algorithm when the input is already in the best possible state or when the algorithm is presented with a specific input that makes it run with the lowest possible number of operations. This is often an unrealistic scenario, but it is useful for establishing a lower bound on the running time of an algorithm.

Worst case time complexity is the time complexity of an algorithm when the input is in the worst possible state or when the algorithm is presented with a specific input that makes it run with the maximum possible number of operations. This is often the most important scenario to consider, as it establishes an upper bound on the running time of an algorithm and helps to ensure that the algorithm will not fail in any scenario.

Average case time complexity is the time complexity of an algorithm when the input is randomly generated or when the algorithm is presented with a variety of inputs. This is often the most realistic scenario to consider, as it takes into account the range of inputs that the algorithm is likely to encounter in real-world usage. Average case time complexity is often used to compare the performance of different algorithms on a particular problem.

In practice, it's important to consider all three measures of time complexity when analyzing an algorithm's performance. The worst case time complexity ensures that the algorithm will always work correctly, while the average case time complexity can provide insight into the algorithm's performance in real-world scenarios. The best case time complexity is less important, but it can provide a useful baseline for understanding the algorithm's performance in ideal scenarios.

1. **Implement quick sort.**

**public class QuickSort {**

**public static void sort(int[] arr) {**

**quickSort(arr, 0, arr.length - 1);**

**}**

**private static void quickSort(int[] arr, int left, int right) {**

**if (left < right) {**

**int pivotIndex = partition(arr, left, right);**

**quickSort(arr, left, pivotIndex - 1);**

**quickSort(arr, pivotIndex + 1, right);**

**}**

**}**

**private static int partition(int[] arr, int left, int right) {**

**int pivot = arr[right];**

**int i = left - 1;**

**for (int j = left; j < right; j++) {**

**if (arr[j] <= pivot) {**

**i++;**

**swap(arr, i, j);**

**}**

**}**

**swap(arr, i + 1, right);**

**return i + 1;**

**}**

**private static void swap(int[] arr, int i, int j) {**

**int temp = arr[i];**

**arr[i] = arr[j];**

**arr[j] = temp;**

**}**

**}**

1. **Implement heap sort.**

**public class HeapSort {**

**public static void sort(int[] arr) {**

**int n = arr.length;**

**for (int i = n / 2 - 1; i >= 0; i--) {**

**heapify(arr, n, i);**

**}**

**for (int i = n - 1; i > 0; i--) {**

**swap(arr, 0, i);**

**heapify(arr, i, 0);**

**}**

**}**

**private static void heapify(int[] arr, int n, int i) {**

**int largest = i;**

**int left = 2 \* i + 1;**

**int right = 2 \* i + 2;**

**if (left < n && arr[left] > arr[largest]) {**

**largest = left;**

**}**

**if (right < n && arr[right] > arr[largest]) {**

**largest = right;**

**}**

**if (largest != i) {**

**swap(arr, i, largest);**

**heapify(arr, n, largest);**

**}**

**}**

**private static void swap(int[] arr, int i, int j) {**

**int temp = arr[i];**

**arr[i] = arr[j];**

**arr[j] = temp;**

**}**

**}**

**What is hashtable?Implement hashtable using chaining method.**

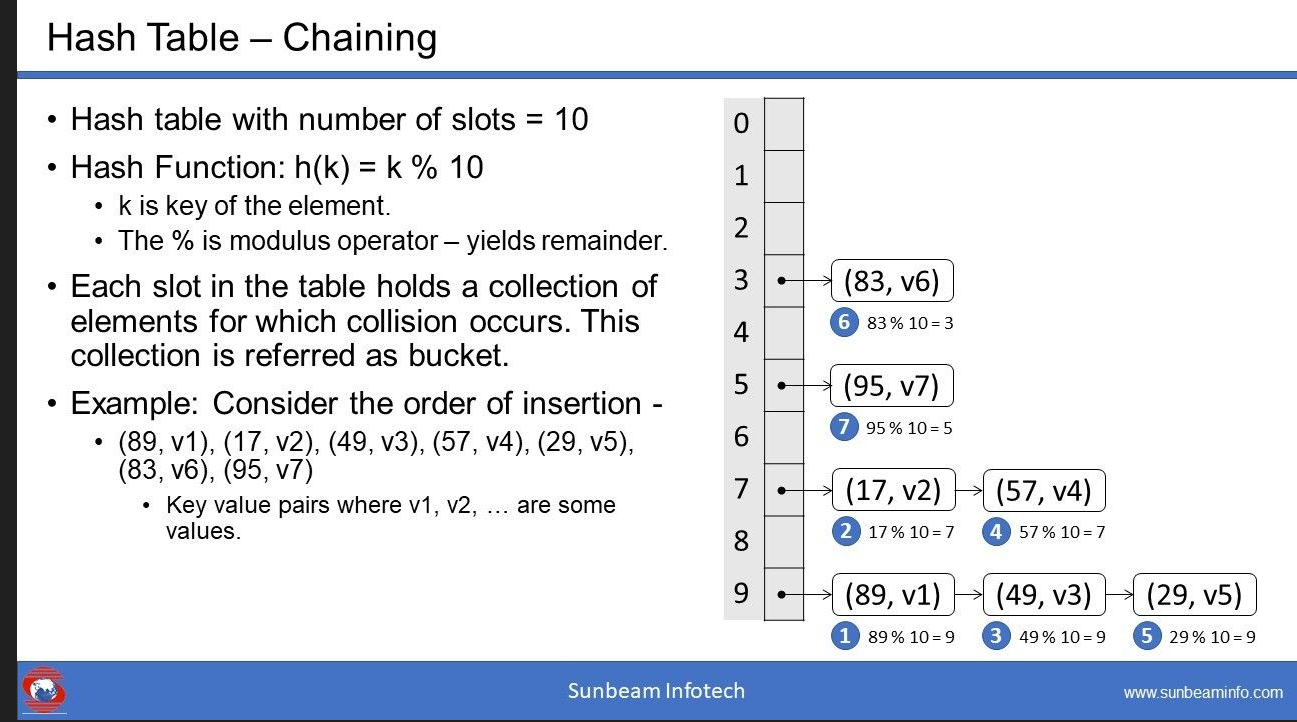
**A hashtable (or hash map) is a data structure that maps keys to values for efficient lookup and retrieval. It uses a hash function to compute an index into an array of buckets or slots, from which the desired value can be found.**

**The basic idea of a hashtable is to store values based on keys rather than their position in a list or array. To store and retrieve values using a hashtable, the key is first hashed by a hash function, which converts the key into a numeric value. This numeric value is then used to determine the index of the bucket or slot in the array where the value will be stored.**

**When retrieving a value, the key is hashed again to find the corresponding index in the array. If there are multiple values stored in the same bucket (known as a collision), the hashtable may use a separate data structure, such as a linked list, to store and retrieve all values associated with that bucket.**

**The primary advantage of hashtables is their efficient lookup time, which is typically O(1) on average. This means that as the size of the hashtable grows, the amount of time required to retrieve a value remains relatively constant, making hashtables a popular choice for implementing lookup tables and associative arrays.**

**However, hashtables can be less efficient than other data structures, such as balanced trees, in certain scenarios. In particular, when the number of collisions is high, the performance of a hashtable can degrade significantly. Additionally, hashtables typically do not preserve the order of the keys, which can be important in some applications.**

****

**What is load factor of hashtable? How it is related to collision handling technique?**

The load factor of a hashtable is a measure of how full the hashtable is. It is defined as the ratio of the number of elements stored in the hashtable to the number of buckets available.

For example, if a hashtable has 100 buckets and contains 50 elements, the load factor is 0.5 (50/100). As the number of elements in the hashtable increases, the load factor also increases.

The load factor is important because it affects the performance of the hashtable. When the load factor is low, there are fewer elements stored in each bucket, which reduces the likelihood of collisions and makes lookups faster. However, as the load factor approaches 1 (i.e., the hashtable is almost full), the likelihood of collisions increases, which can slow down lookups and insertions.

To ensure good performance, it is common practice to set an upper bound on the load factor (e.g., 0.75). When the load factor exceeds this threshold, the hashtable is resized (typically by doubling the number of buckets) to reduce the load factor and improve performance. Resizing the hashtable involves rehashing all the keys and values, which can be a costly operation, so it is important to choose an appropriate initial capacity and load factor to minimize the frequency of resizes.