**Time Complexities**

**Constant Time - O(1)**

* **Definition**: The number of operations does not depend on n; it remains constant.
* **Example**: Accessing an element in an array by index.

**Example**

public class ConstantTimeExample {

public static void main(String[] args) {

int[] arr = {10, 20, 30, 40, 50};

System.out.println(arr[2]); // Always takes the same time, regardless of array size

}

}

**Time Complexity:** O(1) (Single operation, independent of n)

**2. Linear Time - O(n)**

* **Definition**: The number of operations grows proportionally with n.
* **Example**: Traversing an array or a linked list.

**Example**

public class LinearTimeExample {

public static void main(String[] args) {

int[] arr = {10, 20, 30, 40, 50};

for (int i = 0; i < arr.length; i++) {

System.out.println(arr[i]); // Runs `n` times

}

}

}

**Time Complexity:** O(n) (Loop runs n times)

**3. Quadratic Time - O(n²)**

* **Definition**: The number of operations grows as n².
* **Example**: Nested loops, like checking all pairs in an array.

**Example**

public class QuadraticTimeExample {

public static void main(String[] args) {

int[] arr = {10, 20, 30};

for (int i = 0; i < arr.length; i++) {

for (int j = 0; j < arr.length; j++) {

System.out.println(arr[i] + ", " + arr[j]); // Runs `n \* n` times

}

}

}

}

**Time Complexity:** O(n²) (Nested loops)

**4. Logarithmic Time - O(log n)**

* **Definition**: The number of operations increases logarithmically with n.
* **Example**: Binary search (dividing search space in half).

**Example**

public class LogarithmicTimeExample {

public static int binarySearch(int[] arr, int target) {

int left = 0, right = arr.length - 1;

while (left <= right) {

int mid = left + (right - left) / 2;

if (arr[mid] == target) return mid;

if (arr[mid] < target) left = mid + 1;

else right = mid - 1;

}

return -1;

}

public static void main(String[] args) {

int[] arr = {10, 20, 30, 40, 50, 60, 70};

System.out.println(binarySearch(arr, 40)); // Logarithmic divisions

}

}

**Time Complexity:** O(log n) (Halving search space)

**5. Linearithmic Time - O(n log n)**

* **Definition**: More efficient than quadratic but worse than linear. Common in sorting algorithms.
* **Example**: Merge Sort or Quick Sort.

**Example (Merge Sort)**

public class LinearithmicTimeExample {

public static void mergeSort(int[] arr, int left, int right) {

if (left < right) {

int mid = left + (right - left) / 2;

mergeSort(arr, left, mid);

mergeSort(arr, mid + 1, right);

merge(arr, left, mid, right);

}

}

public static void merge(int[] arr, int left, int mid, int right) {

int n1 = mid - left + 1;

int n2 = right - mid;

int[] leftArr = new int[n1];

int[] rightArr = new int[n2];

for (int i = 0; i < n1; i++) leftArr[i] = arr[left + i];

for (int i = 0; i < n2; i++) rightArr[i] = arr[mid + 1 + i];

int i = 0, j = 0, k = left;

while (i < n1 && j < n2) {

if (leftArr[i] <= rightArr[j]) arr[k++] = leftArr[i++];

else arr[k++] = rightArr[j++];

}

while (i < n1) arr[k++] = leftArr[i++];

while (j < n2) arr[k++] = rightArr[j++];

}

public static void main(String[] args) {

int[] arr = {38, 27, 43, 3, 9, 82, 10};

mergeSort(arr, 0, arr.length - 1);

for (int num : arr) System.out.print(num + " ");

}

}

**Time Complexity:** O(n log n) (Recursive divide-and-merge)

**6. Exponential Time - O(2ⁿ)**

* **Definition**: The number of operations doubles with each additional input size.
* **Example**: Recursive Fibonacci.

**Example**

public class ExponentialTimeExample {

public static int fibonacci(int n) {

if (n <= 1) return n;

return fibonacci(n - 1) + fibonacci(n - 2); // Recursive calls grow exponentially

}

public static void main(String[] args) {

System.out.println(fibonacci(5)); // Takes a lot of time for large `n`

}

}

**Time Complexity:** O(2ⁿ) (Each call spawns two new calls)