Algos & Their Complexities Explained

Searching Algorithms

Algorithm	Best Case	Average Case	Worst Case	Space Complexity	Reason
Linear Search	O(1)	O(n)	O(n)	O(1)	Scans each element sequentially; best case when the element is first.
Binary Search	O(1)	O(log n)	O(log n)	O(1)	Repeatedly divides the search interval in half; requires sorted array.
Jump Search	O(√n)	O(√n)	O(√n)	O(1)	Jumps ahead by fixed steps, then does linear search in block.
Interpolation Search	O(1)	O(log log n)	O(n)	O(1)	Estimates position based on distribution; works well with uniform distribution.
Exponential Search	O(1)	O(log n)	O(log n)	O(1)	Expands range exponentially, then does binary search within the range.

Sorting Algorithms

Algorithm	Best Case	Average Case	Worst Case	Space Complexity	Reason
Bubble Sort	O(n)	O(n^2)	O(n^2)	O(1)	Repeatedly swaps adjacent elements; best case when array is already sorted.
Selection Sort	O(n^2)	O(n^2)	O(n^2)	O(1)	Selects the minimum element in each pass and moves it to its correct position.
Insertion Sort	O(n)	O(n^2)	O(n^2)	O(1)	Inserts each element into its correct position; best case when

					array is already sorted.
Merge Sort	O(n log n)	O(n log n)	O(n log n)	O(n)	Divides array into halves, recursively sorts them, and merges; always divides array in half.
Quick Sort	O(n log n)	O(n log n)	O(n^2)	O(log n)	Divides array around a pivot; best case when pivots divide array into equal halves.
Heap Sort	O(n log n)	O(n log n)	O(n log n)	O(1)	Builds a heap and repeatedly extracts the maximum element; ensures O(log n) time for heap operations.
Shell Sort	O(n log n)	O(n log n)^2	O(n^2)	O(1)	Sorts elements with diminishing gaps; best case when initial gaps yield small subarrays.
Counting Sort	O(n + k)	O(n + k)	O(n + k)	O(k)	Counts occurrences of each element; efficient when range k is not significantly larger than n.
Radix Sort	O(nk)	O(nk)	O(nk)	O(n + k)	Sorts elements by individual digits; efficient when number of digits (k) is small.
Bucket Sort	O(n + k)	O(n + k)	O(n^2)	O(n + k)	Distributes elements into buckets and sorts each bucket; efficient when input is uniformly distributed.

Notes:

- **n** is the number of elements in the array.
- **k** is the range of the input (for Counting Sort and Radix Sort).
- The best case for Quick Sort occurs when the pivot element divides the array into two equal halves. The worst case occurs when the pivot elements are either the largest or smallest element.
- For Radix Sort, **k** is the number of digits.

- Space complexities assume in-place sorting algorithms where applicable. For example, Merge Sort requires O(n) additional space for the temporary arrays.