Time Complexity Notes

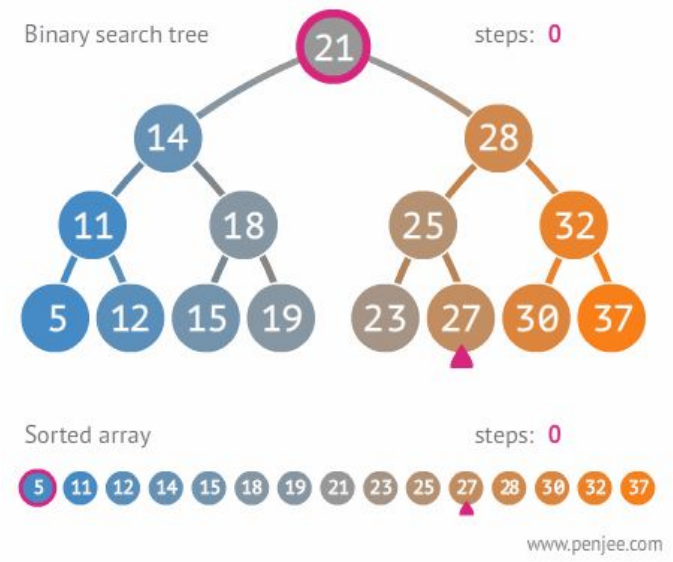
Binary Search

The binary search algorithm is a very efficient way to search for an element in a sorted array or list. Its time complexity is O(log n), where "n" is the number of elements in the array. This means that the algorithm's performance improves as the size of the array increases, but the increase in search time is relatively slow compared to linear search (O(n)).

Binary search works by repeatedly dividing the search interval in half. At each step, the algorithm compares the middle element of the interval with the target value. If the middle element is equal to the target, the search is successful. If the middle element is greater than the target, the search continues in the left half of the interval. If the middle element is smaller than the target, the search continues in the right half of the interval. By repeatedly halving the interval, binary search quickly narrows down the possible locations of the target element until it's found or the interval becomes empty.

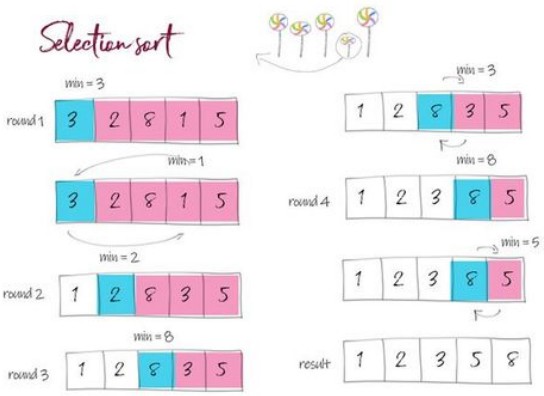
The logarithmic time complexity of binary search is achieved because with each step, the size of the search interval is halved. This effectively eliminates half of the remaining elements, leading to a significant reduction in the number of comparisons needed to find the target element compared to linear search.

To summarize, the time complexity of binary search is O(log n), making it a highly efficient algorithm for searching in sorted collections.



Selection Sort

Selection sort is a simple sorting algorithm that iteratively finds the minimum element from the unsorted portion of the array and swaps it with the first unsorted element. This process continues until the entire array is sorted. Despite its simplicity, selection sort's time complexity of O(n^2) makes it inefficient for large datasets.



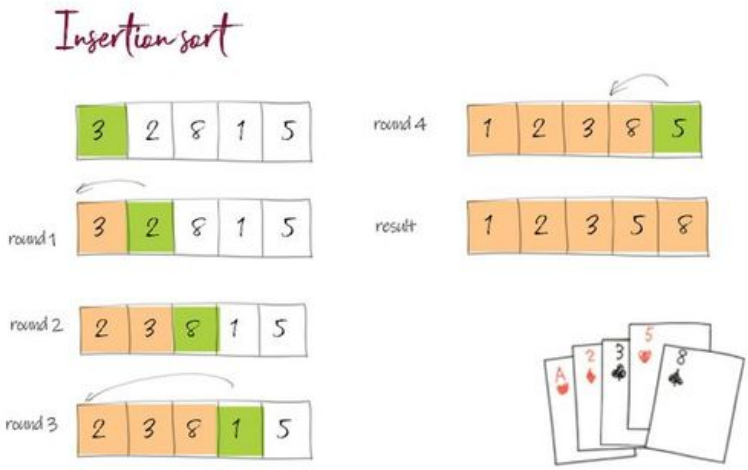
Insertion sort

Insertion sort is a sorting algorithm that builds the final sorted array one element at a time. In the best case, when the input array is already nearly sorted, insertion sort performs efficiently with a time complexity of O(n). It involves minimal swaps and

comparisons as elements are inserted into their correct positions in the sorted part of the array.

In the worst case, when the input array is sorted in reverse order, each new element needs to be moved to the beginning of the sorted part of the array, resulting in a time complexity of O(n^2). This occurs because all existing sorted elements need to be shifted to make room for the new element being inserted.

In summary, insertion sort shines when dealing with almost sorted data (best case), but its performance degrades significantly for fully unsorted data (worst case) due to the need for multiple shifts.



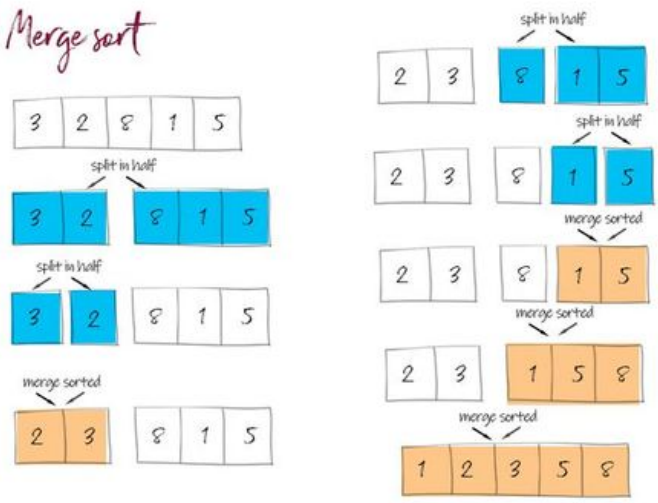
Merge Sort

Merge sort's time complexity of O(n log n) stems from its divide-and-conquer approach. Here's a brief explanation of why:

1. **Divide**: The array is repeatedly divided into halves until single-element subarrays are formed. This process takes logarithmic time, specifically O(log n), where n is the number of elements in the array.
2. **Conquer (Merging)**: Merging the subarrays back together while maintaining their sorted order requires linear time, O(n), where n is the number of elements in the array.

Since the divide and conquer phases are done separately, the total time complexity is the product of the two: O(log n) for the divide phase and O(n) for the merge phase. Multiplying these two complexities together gives the overall time complexity of O(n

log n). This efficiency makes merge sort preferable for larger datasets compared to algorithms like insertion sort or bubble sort, which have worse time complexities for larger inputs.



Quick Sort

Quick sort is a widely used sorting algorithm with an average-case time complexity of O(n log n), making it efficient for large datasets. Here's a quick overview of its time complexity:

1. **Partitioning**: In each iteration, quick sort selects a pivot element and rearranges the array so that all elements smaller than the pivot are on its left and all elements greater are on its right. This partitioning step takes O(n) time.
2. **Recursion**: After partitioning, quick sort recursively sorts the subarrays on the left and right of the pivot. On average, the array is divided into roughly equal parts, leading to a logarithmic number of divisions. Each division takes O(n) time for partitioning.

Combining these two steps, the average-case time complexity of quick sort is O(n log n). However, in the worst case (when the pivot is consistently a minimum or maximum element), the algorithm can degrade to O(n^2). To mitigate this, various techniques like randomized pivot selection or using a median-of-three pivot can be employed.

In summary, quick sort's average-case efficiency, memory usage, and relatively simple implementation have made it a popular choice for sorting tasks.

Heap Sort

Heap sort is a comparison-based sorting algorithm with an average and worst-case time complexity of O(n log n), making it efficient for large datasets. It involves two main steps:

1. **Heapification**: Building a max-heap (or min-heap) from the input array takes O(n) time.
2. **Sorting**: Extracting the maximum (or minimum) element from the heap and restoring the heap property takes O(log n) time, and since this process is repeated for all n elements, the sorting step takes O(n log n) time.

In summary, the combination of heapification and the repeated extraction of elements from the heap results in an overall time complexity of O(n log n) for heap sort. While this complexity is favorable for large inputs, heap sort's main drawback is its lack of stable sorting (equal elements might change their relative order) and its slightly higher overhead due to maintaining the heap structure.