Arrays and Sorting Algorithms

**Arrays**

An array is a data structure used to store a collection of elements, typically of the same type, at contiguous memory locations.

**Key Operations**

1. **Access**: Access an element by its index. Time complexity: O(1)O(1)O(1).
2. **Search**: Find an element. Time complexity:
   * Linear search: O(n)O(n)O(n).
   * Binary search (sorted arrays): O(log⁡n)O(\log n)O(logn).
3. **Insert**:
   * At the end: O(1)O(1)O(1) (amortized in dynamic arrays).
   * At an arbitrary position: O(n)O(n)O(n) (requires shifting).
4. **Delete**:
   * At the end: O(1)O(1)O(1).
   * At an arbitrary position: O(n)O(n)O(n).

**Advantages**

* Fast access using indices.
* Predictable memory layout.

**Disadvantages**

* Fixed size in static arrays.
* Expensive resizing in dynamic arrays.

**Sorting Algorithms**

Sorting algorithms arrange the elements of an array in a specific order (ascending or descending). The choice of algorithm depends on factors like the size of the dataset, memory constraints, and whether the array is partially sorted.

**Common Sorting Algorithms**

1. **Bubble Sort**
   * Compare adjacent elements and swap if necessary.
   * Best Case: O(n)O(n)O(n) (already sorted).
   * Worst Case: O(n2)O(n^2)O(n2).
   * Space: O(1)O(1)O(1) (in-place).
2. **Selection Sort**
   * Select the smallest element and place it in its correct position.
   * Time: O(n2)O(n^2)O(n2) for all cases.
   * Space: O(1)O(1)O(1).
3. **Insertion Sort**
   * Build the sorted portion one element at a time.
   * Best Case: O(n)O(n)O(n) (nearly sorted).
   * Worst Case: O(n2)O(n^2)O(n2).
   * Space: O(1)O(1)O(1).
4. **Merge Sort**
   * Divide the array, sort each half, and merge.
   * Time: O(nlog⁡n)O(n \log n)O(nlogn) for all cases.
   * Space: O(n)O(n)O(n) (requires auxiliary array).
5. **Quick Sort**
   * Partition the array and recursively sort subarrays.
   * Best Case: O(nlog⁡n)O(n \log n)O(nlogn).
   * Average Case: O(nlog⁡n)O(n \log n)O(nlogn).
   * Worst Case: O(n2)O(n^2)O(n2) (when the pivot is poorly chosen).
   * Space: O(log⁡n)O(\log n)O(logn) (in-place recursion).
6. **Heap Sort**
   * Build a max-heap and extract the maximum element iteratively.
   * Time: O(nlog⁡n)O(n \log n)O(nlogn) for all cases.
   * Space: O(1)O(1)O(1).
7. **Radix Sort**
   * Sort based on individual digits/places.
   * Time: O(nk)O(nk)O(nk), where kkk is the number of digits.
   * Space: O(n+k)O(n + k)O(n+k).
8. **Counting Sort**
   * Count the occurrences of each value.
   * Time: O(n+k)O(n + k)O(n+k), where kkk is the range of input values.
   * Space: O(n+k)O(n + k)O(n+k).

**Choosing the Right Sorting Algorithm**

* **Small datasets**: Insertion sort or selection sort.
* **Large datasets**: Merge sort, quick sort, or heap sort.
* **Space constraints**: Quick sort or heap sort.
* **Nearly sorted arrays**: Insertion sort.
* **Non-comparative needs**: Radix sort or counting sort.

Would you like to dive deeper into any specific algorithm or coding examples?

4o