# Sorting Algorithms Cheat Sheet

A concise cheat sheet covering common sorting algorithms, their time complexities, and pseudocode for quick reference during coding interviews and algorithm analysis.

## Basic Sorting Algorithms

### Bubble Sort

| | |
|---|---|
| **Description:** | Repeatedly steps through the list, compares adjacent elements and swaps them if they are in the wrong order. |
| **Time Complexity:** | Worst/Avg: O(n^2), Best: O(n) (when nearly sorted) |
| **Space Complexity:** | O(1) |
| **Pseudocode:** | `for i = 0 to n-1:`<br>`  for j = 0 to n-i-1:`<br>`    if arr[j] > arr[j+1]:`<br>`      swap(arr[j], arr[j+1])` |
| **Use Cases:** | Rarely used in practice due to its inefficiency on large datasets. Good for small, nearly sorted datasets. |

### Selection Sort

| | |
|---|---|
| **Description:** | Finds the minimum element in each iteration and places it at the beginning. |
| **Time Complexity:** | O(n^2) (always) |
| **Space Complexity:** | O(1) |
| **Pseudocode:** | `for i = 0 to n-1:`<br>`  min_idx = i`<br>`  for j = i+1 to n:`<br>`    if arr[j] < arr[min_idx]:`<br>`      min_idx = j`<br>`  swap(arr[i], arr[min_idx])` |
| **Use Cases:** | Simple to implement but generally inefficient for large datasets. Performs well compared to bubble sort. |

### Insertion Sort

| | |
|---|---|
| **Description:** | Builds the final sorted array one item at a time. It is much less efficient on large lists than more advanced algorithms such as quicksort, heapsort, or merge sort. |
| **Time Complexity:** | Worst/Avg: O(n^2), Best: O(n) (when nearly sorted) |
| **Space Complexity:** | O(1) |
| **Pseudocode:** | `for i = 1 to n-1:`<br>`  key = arr[i]`<br>`  j = i-1`<br>`  while j >= 0 and arr[j] > key:`<br>`    arr[j+1] = arr[j]`<br>`    j = j-1`<br>`  arr[j+1] = key` |
| **Use Cases:** | Efficient for small datasets or nearly sorted data. Often used as a subroutine in more complex sorting algorithms. |

## Divide and Conquer Sorting

### Merge Sort

| | |
|---|---|
| **Description:** | Divides the array into halves, recursively sorts each half, and then merges the sorted halves. |
| **Time Complexity:** | O(n log n) (always) |
| **Space Complexity:** | O(n) |
| **Pseudocode:** | `mergeSort(arr, l, r):`<br>`  if l < r:`<br>`    m = (l + r) / 2`<br>`    mergeSort(arr, l, m)`<br>`    mergeSort(arr, m+1, r)`<br>`    merge(arr, l, m, r)` |
| **Use Cases:** | Guaranteed O(n log n) performance, suitable for large datasets. Used in external sorting. |

### Quick Sort

| | |
|---|---|
| **Description:** | Picks an element as a pivot and partitions the array around the pivot. Average case is very efficient. |
| **Time Complexity:** | Worst: O(n^2), Avg: O(n log n), Best: O(n log n) |
| **Space Complexity:** | O(log n) average, O(n) worst (due to recursion stack) |
| **Pseudocode:** | `quickSort(arr, low, high):`<br>`  if low < high:`<br>`    pi = partition(arr, low, high)`<br>`    quickSort(arr, low, pi - 1)`<br>`    quickSort(arr, pi + 1, high)` |
| **Use Cases:** | Generally the fastest sorting algorithm in practice. Sensitive to pivot selection. |

## Advanced Sorting Algorithms

## Heap Sort

| | |
|---|---|
| **Description:** | Uses a binary heap data structure to sort the array. In-place algorithm. |
| **Time Complexity:** | O(n log n) (always) |
| **Space Complexity:** | O(1) |
| **Pseudocode:** | `heapSort(arr):`<br>`  buildMaxHeap(arr)`<br>`  for i = n-1 to 0:`<br>`    swap(arr[0], arr[i])`<br>`    heapify(arr, 0, i)` |
| **Use Cases:** | Guaranteed O(n log n) performance, in-place, but generally slower than quicksort in practice. |

## Radix Sort

| | |
|---|---|
| **Description:** | Sorts integers by processing individual digits. Non-comparison based sorting. |
| **Time Complexity:** | O(nk) where k is the number of digits in the largest number. |
| **Space Complexity:** | O(n+k) |
| **Pseudocode:** | `radixSort(arr, n):`<br>`  for digit = 0 to k:`<br>`    countSort(arr, n, digit)` |
| **Use Cases:** | Efficient for integers when the range of digits is known. Can be faster than comparison sorts under certain conditions. |

# Sorting Algorithm Summary

## Time and Space Complexity Comparison

| Algorithm | Best Case | Average Case | Worst Case | Space Complexity |
|---|---|---|---|---|
| Bubble Sort | O(n) | O(n^2) | O(n^2) | O(1) |
| Selection Sort | O(n^2) | O(n^2) | O(n^2) | O(1) |
| Insertion Sort | O(n) | O(n^2) | O(n^2) | O(1) |
| Merge Sort | O(n log n) | O(n log n) | O(n log n) | O(n) |
| Quick Sort | O(n log n) | O(n log n) | O(n^2) | O(log n) avg, O(n) worst |
| Heap Sort | O(n log n) | O(n log n) | O(n log n) | O(1) |
| Radix Sort | O(nk) | O(nk) | O(nk) | O(n+k) |

## Choosing the Right Sorting Algorithm

- **Small Datasets:** Insertion sort is often the fastest.
- **Large Datasets:** Merge sort or quicksort are generally preferred.
- **Nearly Sorted Data:** Insertion sort or bubble sort (with optimization) can be very efficient.
- **Memory Constraints:** Heap sort is an in-place algorithm.
- **Specific Data Types:** Radix sort can be very efficient for integers.