Sorting Algorithms in C

This document explains eight sorting algorithms (Selection, Insertion, Bubble, Shell, Merge, Quick, Heap, Radix) with simple descriptions, stability, time complexities, and iterative C code. For unstable algorithms, stable versions are provided using index tracking.

1. Selection Sort

Explanation: Scans the unsorted portion to find the smallest element and swaps it with the first unsorted element, repeating until sorted.

- **Stability**: Not stable (can reorder equal elements).
- **Time Complexity**:
  - Best: $O(n^2)$
  - Average: $O(n^2)$
  - Worst: $O(n^2)$

C Code (Iterative):

```c
void selection_sort(int arr[], int n) {
    for (int i = 0; i < n - 1; i++) {
        int min_idx = i;
        for (int j = i + 1; j < n; j++) {
            if (arr[j] < arr[min_idx]) {
                min_idx = j;
            }
        }
        int temp = arr[i];
        arr[i] = arr[min_idx];
        arr[min_idx] = temp;
    }
}
```

2. Insertion Sort

Explanation: Builds a sorted portion by inserting each element into its correct position in the sorted part.

- **Stability**: Stable (preserves order of equal elements).
- **Time Complexity**:: Best: O(n) (nearly sorted)
    - Average: $O(n^2)$
    - Worst: $O(n^2)$ **C Code** (Iterative):

```c
void insertion_sort(int arr[], int n) {
    for (int i = 1; i < n; i++) {
        int key = arr[i];
        int j = i - 1;
        while (j >= 0 && arr[j] > key) {
            arr[j + 1] = arr[j];
            j--;
        }
        arr[j + 1] = key;
    }
}
```

3. Bubble Sort

Explanation: Repeatedly steps through the array, swapping adjacent elements if out of order, until no swaps are needed.

- **Stability**: Stable.
- **Time Complexity**:: Best: O(n) (already sorted)
  - Average: $O(n^2)$
  - Worst: $O(n^2)$ **C Code** (Iterative):

```c
void bubble_sort(int arr[], int n) {
    for (int i = 0; i < n - 1; i++) {
        int swapped = 0;
        for (int j = 0; j < n - i - 1; j++) {
            if (arr[j] > arr[j + 1]) {
                int temp = arr[j];
                arr[j] = arr[j + 1];
                arr[j + 1] = temp;
                swapped = 1;
            }
        }
        if (!swapped) break;
    }
}
```

4. Shell Sort

Explanation: Enhances insertion sort by comparing elements separated by a gap, reducing the gap each pass until it's 1.

- **Stability**: Not stable.
- **Time Complexity**: (using n/2 gap sequence): Best: O(n log n) (nearly sorted)
  - Average: O(n^1.3) (approximate)
  - Worst: O(n^2) **C Code** (Iterative):

```c
void shell_sort(int arr[], int n) {
    for (int gap = n / 2; gap > 0; gap /= 2) {
        for (int i = gap; i < n; i++) {
            int temp = arr[i];
            int j = i;
            while (j >= gap && arr[j - gap] > temp) {
                arr[j] = arr[j - gap];
                j -= gap;
            }
            arr[j] = temp;
        }
    }
}
```

5. Merge Sort

Explanation: Divides the array into halves, sorts them, and merges by taking the smallest element from each half.

- **Stability**: Stable.
- **Time Complexity**:: Best: O(n log n)
  - Average: O(n log n)
  - Worst: O(n log n) **C Code** (Iterative):

```c
void merge_sort(int arr[], int n) {
    int *temp = (int *)malloc(n * sizeof(int));
    for (int width = 1; width < n; width *= 2) {
        for (int left = 0; left < n; left += 2 * width) {
            int l = left;
            int r = left + width < n ? left + width : n;
            int m = left + 2 * width < n ? left + 2 * width : n;
            int i = l, j = r, k = l;
            while (i < r && j < m) {
                if (arr[i] <= arr[j]) {
                    temp[k++] = arr[i++];
                } else {
                    temp[k++] = arr[j++];
                }
            }
            while (i < r) temp[k++] = arr[i++];
            while (j < m) temp[k++] = arr[j++];
            for (i = left; i < m; i++) arr[i] = temp[i];
        }
    }
    free(temp);
}
```

6. Quick Sort

Explanation: Chooses a pivot, partitions the array so smaller elements are on the left and larger on the right, then repeats.

- **Stability**: Not stable.
- **Time Complexity**:: Best: O(n log n)
    - Average: O(n log n)
    - Worst: O(n^2) (unbalanced partitions) **C Code** (Iterative with stack):

```c
void quick_sort(int arr[], int n) {
    int stack[n][2], top = -1;
    stack[++top][0] = 0;
    stack[top][1] = n - 1;
    while (top >= 0) {
        int low = stack[top][0];
        int high = stack[top--][1];
        if (low < high) {
            int pivot = arr[high];
            int i = low - 1;
            for (int j = low; j < high; j++) {
                if (arr[j] <= pivot) {
                    i++;
                    int temp = arr[i];
                    arr[i] = arr[j];
                    arr[j] = temp;
                }
            }
            int temp = arr[i + 1];
            arr[i + 1] = arr[high];
            arr[high] = temp;
            int pivot_idx = i + 1;
            if (pivot_idx - 1 > low) {
                stack[++top][0] = low;
                stack[top][1] = pivot_idx - 1;
            }
            if (pivot_idx + 1 < high) {
                stack[++top][0] = pivot_idx + 1;
                stack[top][1] = high;
            }
        }
    }
}
```

7. Heap Sort

Explanation: Builds a max-heap, then swaps the root (largest) to the end and re-heapifies the rest.

- **Stability**: Not stable.
- **Time Complexity**:: Best: O(n log n)
  - Average: O(n log n)
  - Worst: O(n log n) **C Code** (Iterative):

```
void heap_sort(int arr[], int n) {
    for (int i = n / 2 - 1; i >= 0; i--) {
        int j = i;
        while (1) {
            int largest = j;
            int left = 2 * j + 1;
            int right = 2 * j + 2;
            if (left < n && arr[left] > arr[largest]) largest = left;
            if (right < n && arr[right] > arr[largest]) largest = right;
            if (largest == j) break;
            int temp = arr[j];
            arr[j] = arr[largest];
            arr[largest] = temp;
            j = largest;
        }
    }
    for (int i = n - 1; i > 0; i--) {
        int temp = arr[0];
        arr[0] = arr[i];
        arr[i] = temp;
        int j = 0;
        while (1) {
            int largest = j;
            int left = 2 * j + 1;
            int right = 2 * j + 2;
            if (left < i && arr[left] > arr[largest]) largest = left;
            if (right < i && arr[right] > arr[largest]) largest = right;
            if (largest == j) break;
            int temp = arr[j];
            arr[j] = arr[largest];
            arr[largest] = temp;
            j = largest;
        }
    }
}
```

8. Radix Sort

Explanation: Sorts numbers digit by digit from least to most significant using a stable subroutine (counting sort).

- **Stability**: Stable.
- **Time Complexity**: (for d digits, k=10 for decimal): Best: O(d(n + k))
  - Average: O(d(n + k))
  - Worst: O(d(n + k)) **C Code** (Iterative, assumes non-negative integers):

```c
void radix_sort(int arr[], int n) {
    int max_val = arr[0];
    for (int i = 1; i < n; i++) if (arr[i] > max_val) max_val = arr[i];
    int *output = (int *)malloc(n * sizeof(int));
    for (int exp = 1; max_val / exp > 0; exp *= 10) {
        int count[10] = {0};
        for (int i = 0; i < n; i++) count[(arr[i] / exp) % 10]++;
        for (int i = 1; i < 10; i++) count[i] += count[i - 1];
        for (int i = n - 1; i >= 0; i--) {
            output[count[(arr[i] / exp) % 10] - 1] = arr[i];
            count[(arr[i] / exp) % 10]--;
        }
        for (int i = 0; i < n; i++) arr[i] = output[i];
    }
    free(output);
}
```

Stable Versions of Unstable Algorithms

The unstable algorithms (Selection Sort, Shell Sort, Quick Sort, Heap Sort) are modified to be stable by storing elements with their original indices and using indices to resolve ties for equal values. This adds O(n) space complexity.

Stable Selection Sort

C Code:

```c
typedef struct { int value, index; } Pair;
void stable_selection_sort(int arr[], int n) {
    Pair *pairs = (Pair *)malloc(n * sizeof(Pair));
    for (int i = 0; i < n; i++) {
        pairs[i].value = arr[i];
        pairs[i].index = i;
    }
    for (int i = 0; i < n - 1; i++) {
        int min_idx = i;
        for (int j = i + 1; j < n; j++) {
            if (pairs[j].value < pairs[min_idx].value ||
                (pairs[j].value == pairs[min_idx].value && pairs[j].index <
pairs[min_idx].index)) {
                min_idx = j;
            }
        }
        Pair temp = pairs[i];
        pairs[i] = pairs[min_idx];
        pairs[min_idx] = temp;
    }
    for (int i = 0; i < n; i++) arr[i] = pairs[i].value;
    free(pairs);
}
```

Stable Shell Sort

C Code:

```c
typedef struct { int value, index; } Pair;
void stable_shell_sort(int arr[], int n) {
    Pair *pairs = (Pair *)malloc(n * sizeof(Pair));
    for (int i = 0; i < n; i++) {
        pairs[i].value = arr[i];
        pairs[i].index = i;
    }
    for (int gap = n / 2; gap > 0; gap /= 2) {
        for (int i = gap; i < n; i++) {
            Pair temp = pairs[i];
            int j = i;
            while (j >= gap && (pairs[j - gap].value > temp.value ||
                                (pairs[j - gap].value == temp.value && pairs[j -
gap].index > temp.index))) {
                pairs[j] = pairs[j - gap];
                j -= gap;
            }
            pairs[j] = temp;
        }
    }
    for (int i = 0; i < n; i++) arr[i] = pairs[i].value;
    free(pairs);
}
```

Stable Quick Sort

C Code:

```c
typedef struct { int value, index; } Pair;
void stable_quick_sort(int arr[], int n) {
    Pair *pairs = (Pair *)malloc(n * sizeof(Pair));
    for (int i = 0; i < n; i++) {
        pairs[i].value = arr[i];
        pairs[i].index = i;
    }
    int stack[n][2], top = -1;
    stack[++top][0] = 0;
    stack[top][1] = n - 1;
    while (top >= 0) {
        int low = stack[top][0];
        int high = stack[top--][1];
        if (low < high) {
            Pair pivot = pairs[high];
            int i = low - 1;
            for (int j = low; j < high; j++) {
                if (pairs[j].value < pivot.value ||
                    (pairs[j].value == pivot.value && pairs[j].index <=
pivot.index)) {
                    i++;
                    Pair temp = pairs[i];
                    pairs[i] = pairs[j];
                    pairs[j] = temp;
                }
            }
            Pair temp = pairs[i + 1];
            pairs[i + 1] = pairs[high];
            pairs[high] = temp;
            int pivot_idx = i + 1;
            if (pivot_idx - 1 > low) {
                stack[++top][0] = low;
                stack[top][1] = pivot_idx - 1;
            }
            if (pivot_idx + 1 < high) {
                stack[++top][0] = pivot_idx + 1;
                stack[top][1] = high;
            }
        }
    }
    for (int i = 0; i < n; i++) arr[i] = pairs[i].value;
    free(pairs);
}
```

Stable Heap Sort

C Code:

```
typedef struct { int value, index; } Pair;
void stable_heap_sort(int arr[], int n) {
    Pair *pairs = (Pair *)malloc(n * sizeof(Pair));
    for (int i = 0; i < n; i++) {
        pairs[i].value = arr[i];
        pairs[i].index = i;
    }
    for (int i = n / 2 - 1; i >= 0; i--) {
        int j = i;
        while (1) {
            int largest = j;
            int left = 2 * j + 1;
            int right = 2 * j + 2;
            if (left < n && (pairs[left].value > pairs[largest].value ||
                             (pairs[left].value == pairs[largest].value &&
pairs[left].index < pairs[largest].index)))
                largest = left;
            if (right < n && (pairs[right].value > pairs[largest].value ||
                              (pairs[right].value == pairs[largest].value &&
pairs[right].index < pairs[largest].index)))
                largest = right;
            if (largest == j) break;
            Pair temp = pairs[j];
            pairs[j] = pairs[largest];
            pairs[largest] = temp;
            j = largest;
        }
    }
    for (int i = n - 1; i > 0; i--) {
        Pair temp = pairs[0];
        pairs[0] = pairs[i];
        pairs[i] = temp;
        int j = 0;
        while (1) {
            int largest = j;
            int left = 2 * j + 1;
            int right = 2 * j + 2;
            if (left < i && (pairs[left].value > pairs[largest].value ||
                             (pairs[left].value == pairs[largest].value &&
pairs[left].index < pairs[largest].index)))
                largest = left;
            if (right < i && (pairs[right].value > pairs[largest].value ||
                              (pairs[right].value == pairs[largest].value &&
```

```
    pairs[right].index < pairs[largest].index)))
                    largest = right;
                if (largest == j) break;
                Pair temp = pairs[j];
                pairs[j] = pairs[largest];
                pairs[largest] = temp;
                j = largest;
            }
        }
        for (int i = 0; i < n; i++) arr[i] = pairs[i].value;
        free(pairs);
    }
```

Notes

- **Stability**: Unstable algorithms are made stable by using a Pair struct to track original indices, comparing indices for equal values. This adds O(n) space complexity.
- **Iterative Conversion**: Merge Sort and Quick Sort are implemented iteratively using loops (and a stack for Quick Sort) to avoid recursion.
- **Radix Sort Limitation**: Assumes non-negative integers. For negative numbers or non-integers, preprocessing (e.g., offsetting) is required.
- **Memory Management**: C code uses malloc and free for dynamic arrays. Ensure proper memory handling in practice.
- **Space Complexity**: Stable versions, Merge Sort, and Radix Sort use O(n) extra space. Others are in-place (O(1) extra, excluding stack space for Quick Sort).
  (* **Generated by Grok 3**)