Sorting Techniques

This C++ file is an exceptional and comprehensive library of **sorting algorithms**, covering the full range from simple, intuitive methods to highly efficient, advanced techniques. It's perfectly structured to provide a side-by-side comparison of different sorting philosophies.

The file begins with the foundational **comparison-based sorts**: **Bubble Sort**, **Insertion Sort**, and **Selection Sort**. These are easy to understand but are generally less efficient for large datasets.

It then progresses to the more powerful **divide-and-conquer** algorithms: **Quick Sort** and **Merge Sort**. Both recursive and iterative versions of Merge Sort are provided, offering a clear look at two ways to implement the same logic. These algorithms are the workhorses of modern sorting due to their excellent average-case performance.

Finally, the file explores a fascinating category of **non-comparison-based sorts** that are extremely fast for specific types of data. This includes **Count Sort**, **Bucket/Bin Sort**, and **Radix Sort**, which sort integers by using their actual values or digits as indices. The file concludes with **Shell Sort**, a clever enhancement of Insertion Sort that improves its performance by sorting elements that are far apart first.

**Bubble Sort**

This is the simplest sorting algorithm. It repeatedly steps through the list, compares adjacent elements, and swaps them if they are in the wrong order. The passes through the list are repeated until the list is sorted. It's like bubbles in a drink rising to the top.

* for(int j = 0; j < n-i-1; j++){ ... } This inner loop is responsible for one "pass" through the array. It compares each element A[j] with its neighbor A[j+1] and swaps them if A[j] is larger. After the first pass, the largest element is guaranteed to be at the very end.
* if(flag == 0){ break; } This is a crucial optimization. The flag variable checks if any swaps were made during a pass. If a full pass is completed with no swaps (flag remains 0), it means the array is already sorted, and the algorithm can **terminate early**. This makes Bubble Sort adaptive.

**Insertion Sort**

This algorithm builds the final sorted array one item at a time. It iterates through the input elements and, for each element, it finds the correct position in the sorted part of the array and inserts it there by shifting other elements to the right. It's like sorting a hand of playing cards.

* for(i = 1; i<n; i++){ ... } The main loop starts from the second element (i=1), assuming the first element is a sorted sub-array of size one.
* x = A[i]; The current element to be inserted is stored in a temporary variable x.
* while(j>=0 && A[j] > x){ A[j+1] = A[j]; j--; } This while loop is the core of the algorithm. It moves backward through the already sorted part of the array, **shifting every element that is greater than x one position to the right**. This creates an empty slot for x.
* A[j+1] = x; The element x is placed into the empty slot it belongs in.

**Selection Sort**

This algorithm divides the input list into two parts: a sorted sublist which is built up from left to right, and a sublist of the remaining unsorted items. It proceeds by finding the smallest element in the unsorted sublist, swapping it with the leftmost unsorted element, and moving the sublist boundaries one element to the right.

* for(i = 0; i<n-1; i++){ ... } The outer loop iterates through the array to place the correct element at each position i.
* for(j=k=i; j<n; j++){ if(A[j] < A[k]){ k = j; } } This inner loop is responsible for finding the **minimum element** in the unsorted part of the array (from index i to the end). The index of this minimum element is stored in k.
* swap(&A[i], &A[k]); After the minimum element is found at index k, it is **swapped** with the element at the current position i. This places one more element correctly in the sorted part of the array.

**Quick Sort**

This is a highly efficient, "divide and conquer" sorting algorithm. It works by selecting a 'pivot' element from the array and partitioning the other elements into two sub-arrays, according to whether they are less than or greater than the pivot. The sub-arrays are then sorted recursively.

* int pivot = A[low]; In this implementation, the **first element** of the sub-array is chosen as the pivot.
* do{i++;}while(A[i]<=pivot); do{j--;}while(A[j]>pivot); This is the partitioning logic. Two pointers, i and j, start from opposite ends and move towards each other. i moves right, skipping over elements smaller than the pivot, and j moves left, skipping over elements larger than the pivot.
* swap(&A[low], &A[j]); When i and j cross, the partitioning is complete. The pivot element is swapped with the element at index j, placing the pivot in its final sorted position.
* QuickSort(A, low, j); QuickSort(A, j+1, high); The algorithm then makes two **recursive calls** to sort the sub-array to the left of the pivot and the sub-array to the right of the pivot.

**Merge Sort (Iterative Version)**

This is an iterative, "bottom-up" implementation of the Merge Sort algorithm. Instead of using recursion, it sorts the array by merging progressively larger and larger sub-arrays in a series of passes.

* void Merge(int A[], int low, int mid, int high){ ... } This is the crucial **Merge function**. It takes two adjacent sorted sub-arrays (low to mid and mid+1 to high) and merges them into a single sorted sub-array using a temporary array B.
* for(p = 2; p<=n; p = p\*2){ ... } This outer loop controls the **size of the sub-arrays** to be merged in each pass (p). It starts with merging sub-arrays of size 1 (to create sorted lists of size 2), then merges those to create lists of size 4, then 8, and so on, until the entire array is sorted.

**Recursive Merge Sort**

This is the classic, more common implementation of Merge Sort. It's a "top-down," divide-and-conquer algorithm that recursively breaks the array down into sub-arrays of size one, and then merges them back together in sorted order.

* if(l<h){ ... } This is the **base case** for the recursion. The recursion continues as long as the sub-array has more than one element.
* mid = (l+h)/2; MergeSort(A, l, mid); MergeSort(A, mid+1, h); These are the two **recursive calls**. The algorithm splits the current array into two halves at the midpoint and recursively calls itself to sort the left half and then the right half.
* Merge(A, l, mid, h); After the two halves have been sorted by the recursive calls, the **Merge function** is called to combine them back into a single, larger sorted array.

**Count Sort**

This is a non-comparison-based sorting algorithm that is extremely fast for sorting integers within a known, limited range. It works by counting the number of occurrences of each distinct element.

* max = findMax(A, n) + 1; C = (int \*)malloc(sizeof(int) \* max); First, the algorithm finds the **maximum value** in the input array. It then creates an auxiliary "count" array C with a size equal to max+1.
* for(int j = 0; j<n; j++){ C[A[j]]++; } This loop populates the count array. It iterates through the input array A, and for each element A[j], it uses that element's value as an **index** into C and increments the count. This builds a **frequency map**.
* while(j<max){ if(C[j]>0){ A[i++] = j; C[j]--; } ... } This final loop rebuilds the original array A in sorted order. It iterates through the count array C. For each index j, it writes the value j back into A as many times as its count in C.

**Bucket/Bin Sort**

This is another non-comparison sort that distributes elements into a number of "buckets" or "bins." Each bucket is then sorted individually, either using a different sorting algorithm or by recursively applying the bucket sort algorithm. This implementation uses a linked list for each bucket.

* Node\*\* bins = new Node\* [max + 1]; The algorithm first finds the maximum value and then creates an array of pointers to Node. This bins array acts as the set of buckets, where each index corresponds to a value, and the pointer at that index is the **head of a linked list** for all elements with that value.
* Insert(bins, A[i]); The code iterates through the input array A and inserts each element A[i] into the correct bucket by calling the Insert function. This function adds a new node to the end of the linked list at bins[A[i]].
* A[j++] = Delete(bins, i); Finally, it iterates through the bins array from index 0 to max. For each bucket, it drains the linked list, adding the elements back into the original array A in order.

**Radix Sort**

Radix Sort is a non-comparison integer sorting algorithm that sorts data with integer keys by grouping keys by the individual digits which share the same significant position and value. It processes the numbers from the least significant digit to the most significant digit.

* int nPass = countDigits(max); First, it determines how many "passes" are needed by finding the **number of digits** in the largest number in the array.
* for (int pass=0; pass<nPass; pass++){ ... } The main loop iterates once for each digit's place (once for the ones place, once for the tens, etc.).
* int binIdx = getBinIndex(A[i], pass); Insert(bins, A[i], binIdx); In each pass, it iterates through the input array A. The getBinIndex function calculates the correct bin (0-9) for a number based on its digit in the current pass. For example, in pass 0, the number 237 would go into bin 7. The number is then inserted into that bin's linked list.
* while (bins[i] != nullptr){ A[j++] = Delete(bins, i); } After all numbers have been distributed into the bins for the current pass, the code collects them back into the array A in order (from bin 0 to bin 9). This re-collection process ensures the numbers are sorted based on the current digit's place.

**Shell Sort**

Shell Sort is an optimization over Insertion Sort. The idea is to allow the exchange of items that are far apart. It starts by sorting pairs of elements far apart from each other, then progressively reduces the gap between elements to be compared.

* for (gap = n/2; gap >= 1; gap /= 2) { ... } This outer loop controls the **gap size**. It starts with a large gap (n/2) and reduces it by half in each iteration until the gap is 1.
* for (i = gap; i < n; i++) { ... } This inner loop iterates through the array, starting from the first element of the current gapped sub-array.
* while (j >= 0 && A[j] > temp) { A[j + gap] = A[j]; j = j - gap; } This is essentially an **insertion sort performed on a gapped sub-array**. Instead of comparing A[j] with A[j-1], it compares it with A[j-gap]. This allows elements to make big jumps towards their correct positions in the early passes, making the final pass (which is a standard insertion sort with gap=1) very fast.