**Runtime Overviews**

**Observed Execution Times (s)**

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Sorting**  **Algorithm** | **Number of Entries to Sort (N)** | | | |
| **10** | **100** | **1,000** | **10,000** |
| **Selection Sort** | 0.001 | 0.001 | 0.004 | 0.348 |
| **Bubble Sort** | 0.001 | 0.001 | 0.013 | 1.156 |
| **Merge Sort** | 0.001 | 0.001 | 0.003 | 0.227 |
| **Quick Sort** | 0.001 | 0.001 | 0.001 | 0.004 |
| **Bucket Sort** | 0.001 | 0.001 | 0.001 | 0.003 |

|  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| **Sorting Algorithm** | **Number of Entries to Sort (N)** | | | | | | | | | |
| **1000** | **2000** | **3000** | **4000** | **5000** | **6000** | **7000** | **8000** | **9000** | **10000** |
| **Selection Sort** | 0.004 | 0.014 | 0.032 | 0.055 | 0.087 | 0.124 | 0.169 | 0.221 | 0.279 | 0.348 |
| **Bubble Sort** | 0.013 | 0.048 | 0.104 | 0.185 | 0.285 | 0.417 | 0.563 | 0.745 | 0.948 | 1.156 |
| **Merge Sort** | 0.003 | 0.01 | 0.023 | 0.038 | 0.059 | 0.082 | 0.111 | 0.145 | 0.183 | 0.227 |
| **Quick Sort** | 0.001 | 0.001 | 0.001 | 0.002 | 0.002 | 0.002 | 0.002 | 0.003 | 0.004 | 0.004 |
| **Bucket Sort** | 0 | 0.001 | 0.001 | 0.001 | 0.001 | 0.002 | 0.002 | 0.002 | 0.002 | 0.003 |

**Analysis:**

When the number of data entries to be sorted is small (<1000), the performance efficiencies of the various sorting algorithms are virtually indistinguishable. On the ‘scale of N’, these differences are still existent, but have little significance in terms of overall computation time. As the number of entries increases, the performance efficiency of the sorting algorithm rapidly becomes more important, as the relationship between the execution time and the number of entries becomes more empirically evident. Once the number of entries exceeds 2000, there is a clearly observable divergence in computation time between the various sorting algorithms. In particular, for Quick Sort and Bucket Sort, the fastest sorting algorithms, there is close to no change in execution time as the number of entries increases from 1000 to 10,000; the execution time is very close to 0 for all N within that range.

While the execution time for all sorting algorithms is dependent on the size of N, there is a smaller growth variance for Quick Sort and Bucket Sort with respect to the scaling of N. In fact, the trend lines for both of these algorithms are linear, from inspection. Conversely, the relationship between execution time and the number of entries to be sorted for Bubble Sort is much more sensitive to the size of N; Bubble Sort takes exponentially more computation time for more and more entries. From the graph, it is clear that there is a sharp deviation from linearity all the way through. Merge Sort and Selection Sort are intermediary between these two extremes for computation time and have moderate performance efficiencies, being less sensitive to N scaling than Bubble Sort, but growing more quickly than either Quick Sort or Bucket Sort in computation time. In particular, Bubble Sort exhibited very poor performance, being more than 3X slower than the second slowest sorting algorithm in the list, Selection Sort.

For all test cases from 10 to 10,000, the order of performance efficiencies was the same, from fastest to slowest:

1. Bucket Sort
2. Quick Sort
3. Merge Sort
4. Selection Sort
5. Bubble Sort

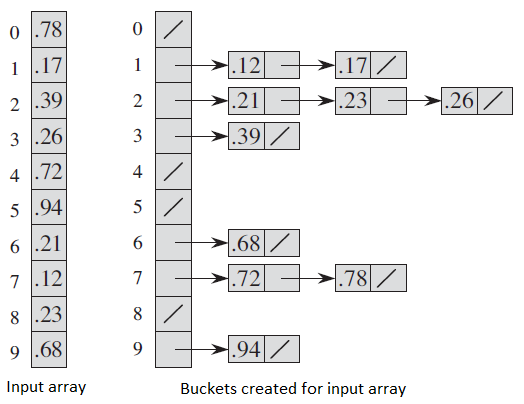
**Scalable Computational Efficiency: Number of Comparisons**

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Sorting Algorithm** | **Number of Comparisons** | | | |
| **Best** | **Average** | **Worst** | **Comments** |
| **Selection Sort** | N2 | N2 | N2 | Selection Sort is a highly simplistic sort, but is inefficient for sorting larger collections of data. In Selection Sort, the unsorted elements are compared one at a time, with the first element being considered the smallest as a starting point. If an element is smaller than the smallest element in the sorted set, these elements are then swapped, so that the newly inserted element is redefined to be the smallest. This process carries on until all the elements in the set are sorted (in ascending order).  When the sorting is run, finding the lowest element requires scanning through the entire array of n elements, which takes n-1 comparisons. These elements must then be swapped into place in ascending order. Once this smallest element has been found and inserted into the sorted data collection, finding the next smallest element out of the remaining n-1 elements takes n-2 comparisons. This is continued until the data collection has been completely sorted. The sum of these comparisons for all n elements can be written as follows:  As the leading power is of degree-2, the time complexity of Selection Sort is quadratic. This behavior is consistently the same for all test cases. |
| **Bubble Sort** | N | N2 | N2 | In our table of observed execution times, Bubble Sort is less efficient than Selection Sort, although similarly difficult to implement and with the same class of time complexity. The premise of Bubble Sort is that pairs of adjacent elements in the data collection are compared until all elements are in the correct order. The unsorted collection might have to be run through more than once to ensure the correct ordering; must be run until no swaps occur in a run of the entire sequence.  In the average- and worst-case scenarios, Bubble Sort has quadratic time complexity. This is due to the need to compare all pairs of adjacent elements in the unsorted sequences and swap accordingly. In the first run through, it would have to compare n elements (with n-1 comparisons) and, in general, on the kth run through, it would have to compare n-k+1 elements (with n-k comparisons) until the sequence is completely sorted. The sum of these comparisons for all n elements can be written as follows:  In the best-case scenario (which is rather trivial and unlikely), all n elements are already sorted in ascending order and Bubble Sort only has to do n comparisons to determine that the list is correctly sorted. Thus, in the best-case scenario, the time complexity is linear. |
| **Merge Sort** | NlogN | NlogN | NlogN | While Merge Sort is considerably more complicated and difficult to implement than either Selection Sort or Bubble Sort, it is much more efficient and general-purpose. Conceptually, Merge Sort uses a ‘divide-and-conquer’ strategy where the data collection of n elements is first divided into n sub-collections (each containing 1 element), which are then repeatedly merged to produce new sorted sub-collections until there is only 1 sub-collection remaining of equal size to the base data collection. The sub-collection will then be sorted. Splitting and remerging of the sub-collections is done recursively.  Merge Sort is written in terms of recursive calls between the main function, Merge Sort and an auxiliary function. This auxiliary function sorts ‘left and right’ portions of the data collection specified by Merge Sort and merges the portions of the data collection together. In the ‘divide-step’, Merge Sort recursively calls itself to divide the array into two halves until all portions contain 1 element. In the ‘conquer-step’, Merge then uses a temporary array to store items moved from the sorted portions of A so that the temporary array is sorted. This is done by comparing the elements of separate halves; elements are allocated from the two portions to the temporary array from smallest to largest until all items have been moved from those portions. Lastly, in the ‘combine-step’, these values of the temporary array are then copied back in the base data collection within the index bounds specified by the Merge Sort function, merging halves in increasing powers of 2. This is done until all of the portions have been merged and the resulting collection is completely sorted.  Finding the time complexity of Merge Sort is a bit trickier. We can begin with the definition that for a collection containing only 1 element, the time complexity for Merge Sort is constant:   * T(1) = 1   Building on this:   * The time to Merge Sort n elements is the time it takes to Merge Sort n/2 elements twice (n = twice n/2) plus the time it takes to Merge the two portions each containing n/2 elements, which takes linear time (merging n elements together) * Thus, T(N) = 2T(N/2) + N  |  |  | | --- | --- | | **Portions** | **Recurrence Relation** | | 2 | T(N) / N = T(N/2) / (N/2) + 1 | | 4 | T(N/2) / (N/2) = T(N/4) / (N/4) + 1 | | 8 | T(N/4) / (N/4) = T(N/8) / (N/8) + 1 | | … | | | N/portions = 2 | T(2) / 2 = T(1) / 1 + 1 |  * If all the terms are added together, the left-hand side will equal the right-hand side, which then reduces to T(N)/N = T(1)/1 + LogN; LogN is derived from the fact that there are LogN 1’s on the right-hand side of the equation (base 2) * T(1), as mentioned earlier, is just 1, so multiplying both sides by N to isolate T yields T(N) = N + NlogN * The dominant terms in this expression is NlogN; therefore, the time complexity of Merge Sort in Big Oh notation is O(NlogN) or log-linear time |
| **Quick Sort** | NlogN | NlogN | N2 | Similarly to Merge Sort, Quick Sort also uses a ‘divide-and-conquer’ strategy and is very efficient. Quick Sort essentially consists of three steps:   * Choosing a pivot value; often the midpoint, but this is not required * In the partition step, rearranging elements in such a way that all elements which are less than the pivot go to the left of the array and all elements which are greater go to the right of the array (swapping) * Recursively calling Quick Sort to sort both left and right parts until the sequence is completely sorted; recursion continues as long as the indexes for i and j do not cross   Time complexity is a little tricky here as well. In the partition step of the sort, the for loop stops when the indexes i and j cross; thus, there are N iterations and this step occurs in cN linear time. In the partition step, swapping is one operation and occurs in constant time. With the recursion calls, things become a bit more complicated. There are two possible distinct cases:   * Best-case: each recursion call is on one part (left or right); thus T(N) = 2T(N/2) as N is split into two sub-collections of size N/2, each of which takes T(N)/2 time (although this disregards the -1 from the exclusion of the pivot point, this is not too important) * Worst-case: one portion is empty and the other contains N-1 elements (excluding the pivot point); thus T(N) = T(N-1)   In general, the total time to sort the collection is comprised of the time it takes to sort the left side with i elements, the time it takes to sort the right half with N-i-1 elements (minus the elements in left side and the pivot point) and the time for partitioning the collection. Thus, T(N) = [T(i) + T(N-i-1)] + cN, where c is a constant multiplier.  What is the time complexity for the best-case scenario, generalizable to the average-case scenario? Similar thing we did with Merge Sort here.   * T(N) = 2T(N/2) + cN  |  |  | | --- | --- | | **Portions** | **Recurrence Relation** | | 2 | T(N) / N = T(N/2) / (N/2) + c | | 4 | T(N/2) / (N/2) = T(N/4) / (N/4) + c | | 8 | T(N/4) / (N/4) = T(N/8) / (N/8) + c | | … | | | N/portions = 2 | T(2) / 2 = T(1) / 1 + c |  * If all the terms are added together, the left-hand side will equal the right-hand side, which then reduces to T(N)/N = T(1) + cLogN; LogN is derived from the fact that there are LogN c’s on the right-hand side of the equation (base 2) * Multiplying both sides by N, we obtain the equation T(N) = N + cNlogN * Same method can be applied with slight modification to show that the average-case scenario is also cNLogN log-linear time; have to use the average value for T(i), where T(i) is 1/N times the sum of T(0) through T(N-1)   What about the worst-case scenario?  Here, the pivot is the smallest element.   * T(N) = T(N-1) + cN, N > 1 * T(N-1) = T(N-2) + c(N-1) * T(N-2) = T(N-3) + c(N-2) * T(N-3) = T(N-4) + c(N-3) * … * T(2) = T(1) + c.2   If we add together the terms on both sides of the equation, the left-hand and right-hand sums should be the same:   * T(N) + T(N-1) + T(N-2) + … + T(2)   = T(N-1) + T(N-2) + … + T(2) + T(1) + c(N) + c(N-1) + c(N-2)  + … + c.2  Cancelling out common terms, we are left with c(N(N+1)/2 -1); note that all the T(N-1) … T(2) terms are eliminated and the arithmetic progression of terms with c can be rewritten as a triangular sum:   * T(N) = c(N(N+1)/2 -1) = cN2/2 + cN/2 – c   Thus, Quick Sort has a worst-case behavior of quadratic time complexity. |
| **Bucket Sort** | N + K | N + K | N2 | This is provided below in the explanation of the additional sorting algorithm. N is the number of elements in the collection. K represents the number of buckets in the collection. |

**Explanation of Additional Sorting Algorithm**

Bucket Sort, also commonly known as Bin Sort, is a very efficient distribution sort where:

1. K empty bins are created (and initialized to 0), where K is determined by the range of elements
2. N allocated entries in the data collection (e.g. array or vector) are distributed to bins in an auxiliary data collection
3. These entries are then sorted using another sorting algorithm, typically Insertion Sort
4. Once the entries are sorted in the auxiliary data collection, they are then merged into a sorted list



For the interval of data to be sorted, Bucket Sort divides the interval into K equally-sized subintervals or buckets into which the N elements of the unsorted list can be distributed, each with range N/K. Optimally, Bucket Sort assumes the entries in the unsorted data collection are uniformly distributed in the interval. Thus, when the unsorted elements are distributed across the K subintervals, there should be about the same number of elements in each bucket. Although Bucket Sort has a very quick runtime, it is heavier in terms of memory consumption, as an auxiliary data collection for the buckets must be allocated. Thus, Bucket Sort is optimized when allocating additional memory to the sorting is not an issue.

The size of the bucket can be adjusted, per the ‘density’ of the data collection to be sorted, for further refinement. If the elements of the collection are ‘dense’ with smaller differences between elements, a smaller bucket size is favoured. If the elements of the collection are ‘sparse’ with larger differences between elements, a larger bucket size, one which can hold multiple values, is favoured. If elements are sufficiently ‘far apart’, then it is statistically unlikely any one bucket that can hold multiple values will contain more than one element, owing to the assumption that the entries are uniformly distributed as opposed to clumped. There will be less buckets to sort and each bucket will have as many (or as few) elements as there would be for more buckets in a ‘dense’ data collection, thus minimizing runtime. In this program, the bin size is given to be 1, as many of the data entries are close to each other and the data collection is very ‘dense’.

In general, Bucket Sort is very efficient and has linear time complexity. Inserting an unsorted entry into a bin takes constant time. If the unsorted entries in the data collection are uniformly distributed, the inner Insertion Sort would take linear time N on average due to the small number of elements per bin. Finally, merging the bins into the resultant sorted list is of N linear time complexity, as there will be N elements across all buckets to be merged together. The net time complexity for the whole process is therefore linear.

More rigorously, it takes, on average, cN/K time to sort each bucket, where c is a constant multiplier. A more complete and robust picture of time complexity also accounts for the constant time required to scan the bucket; even if the bucket is empty, the algorithm must check if there are any elements in the bucket to sort. Thus, it takes cN/K + d time on average to sort each bucket. Multiplying the average time to sort a bucket by the number of buckets, K, yields the total time for sorting:

For the best-case scenario, the elements are uniformly distributed into the buckets, while there would be deviations from uniformity in the average case. For the best- and average- case scenarios, Bucket Sort has O(N+K) time complexity.

However, it can behave poorly in certain scenarios with a worst-case quadratic time complexity. This happens when all the N elements in the unsorted data collection are allocated to the same bucket in the auxiliary data collection, which then has to be sorted using the inner sorting algorithm. When this is the case, Bucket Sort degenerates into the inner sorting algorithm and takes on its time complexity. For the more commonly used Insertion Sort, this worst-case time complexity is quadratic. It is possible to minimize this worst-case behavior by using different inner sorting algorithms depending on the number of elements in the bucket. If there are many elements, it would be preferable to use Merge Sort or Quick Sort rather than Insertion Sort, which have an average linear-logarithmic time complexity (although Quick Sort has a worst-case quadratic time complexity). For a smaller number of elements, Insertion Sort is easy to implement and is relatively efficient.

**C++ implementation of Bucket Sort**

//Bucket Sort

//Splits the N elements of the unsorted Vector into m buckets, each of which stores a range of data with range N/m

//These elements are then sorted using Insertion Sort and merged together to form the sorted list; in this implementation, the data will be sorted in ascending order

void BucketSort(vector<int> &dataVector, int vectorSize)

{

//Number of buckets, m, is defined as one greater than the range of data; in this case, we are assuming from 0 – 10,000

int m = 10001;

//Create m empty buckets

int buckets[m];

//Initialize all buckets to 0

for (int i = 0; i < m; i++)

buckets[i] = 0;

//Puts elements into buckets

for (int i = 0; i < vectorSize; i++)

buckets[dataVector[i]]++;

//Sort using Insertion Sort and merge

for (int i = 0, j = 0; j < m; j++)

for (int k = buckets[j]; k > 0; k--)

dataVector[i++] = j;

}