Analysis of Common Sorting Algorithms

Utkrisht Dhankar

March 6, 2017

1 Abstract

There are many sorting algorithms available today, and sorting is one of the most used operations in modern computing. Therefore, it is worthwhile to examine common sorting algorithms, and see their strengths and weeknesses. Here, I'll be analysing 6 common sorting algorithms: Bubble Sort, Insertion Sort, Selection Sort, Rank Sort, Merge Sort, and Quick Sort.

2 Theoretical Analysis

2.1 Bubble Sort

Bubble sort is theoretically the slowest algorithm in the list of algorithms analysed here. The basic idea is to go through the list, swapping elements every time two adjacent out of order elements are seen. This algorithm is always $\mathcal{O}(n^2)$, and the constants associated are generally high. It tends to be very slow as in each iteration there are very few swap operations but many comparisions. The pseudocode for Bubble Sort is as follows[1].

```
procedure bubbleSort( A : list of sortable items )
   n = length(A)
   repeat
     swapped = false
     for i = 1 to n-1 inclusive do
       /* if this pair is out of order */
       if A[i-1] > A[i] then
         /* swap them and remember something changed */
         swap( A[i-1], A[i] )
         swapped = true
       end if
     end for
   until not swapped
end procedure
```

[1] from Bubble Sort on Wikipedia `https://en.wikipedia.org/wiki/Bubble_sort`

2.2 Insertion Sort

Insertion sort is also theoretically quite a slow algorithm, but it has some advantages, especially when sorting arrays that are almost sorted already. The idea with Insertion Sort is to keep a list of sorted elements in place at the start of the array being sorted, and insert the first unsorted element into the sorted array in its correct place. In such cases, Insertion Sort's running time is close to or even equal to $\mathcal{O}(n)$ as it does not modify the array on each iteration if it finds an already sorted element in place, and so it is quite useful in such cases. It's worst case time complexity is $\mathcal{O}(n^2)$, just like Bubble Sort, but the constants associated are lower. The pseudocode is as follows [2]:

```
for i = 1 to length(A)
    x = A[i]
    j = i - 1
    while j >= 0 and A[j] > x
        A[j+1] = A[j]
        j = j - 1
    end while
    A[j+1] = x[3]
 end for
```

2.3 Selection Sort

Selection Sort is a similar sorting algorithm to Insertion Sort, in that we maintain a list of sorted elements, but this time, we select the smallest element from the unsorted list and append it to the sorted list each iteration. This does not have the advantage that Insertion Sort has when it comes to close to sorted arrays however, as finding the minimum element in the unsorted array always takes a lot of time. It's running time is always $\mathcal{O}(n^2)$. The pseudocode for Selection Sort is as follows [3] :

```
SELECTION-SORT(A, n)
    for j = 1 to n-1
```

[2]Cormen, Thomas H.; Leiserson, Charles E.; Rivest, Ronald L.; Stein, Clifford (2001) [1990]. *Introduction to Algorithms (2nd ed.)*. MIT Press and McGraw-Hill. Section 2.1: Insertion sort, pp. 15 to 21.

[3]Modified from `http://freefeast.info/general-it-articles/selection-sort-pseudo-code-of-selection-sort-selection-sort-in-data-structure/`

```
        smallest = j
        for i = j + 1 to n
             if A[ i ] < A[ smallest ]
                 smallest = i
            Exchange A[ j ] and A[ smallest ]
```

2.4 Rank Sort

Rank Sort maintains a seperate list of the rank of the elements, where the rank is the number of elements smaller than or equal to an element. This creates some ambiguity when two numbers are the same, but this can be avoided by creating a loop that only looks forward for elements, and not backwards. Rank Sort is always $\mathcal{O}(n^2)$, and also requires $\mathcal{O}(n)$ extra space to store the auxiliary list. The Pseudocode for Rank Sort is as follows:

```
RANK-SORT(A, n)
    ranks = zeros(n)
    for (i = 1 to n - 1)
        for (j = i to n - 1)
            if (A[i] > A[j])
                rank[i]++
            else
                rank[j]++

    copy = A
    for (i = 1 to n-1)
        A[rank[i]] = copy[i]
```

2.5 Merge Sort

Merge Sort is the first of the algorithms I'll be analysing here which uses the divide and conquer methodology to sort the array. The idea is to divide the array to be sorted into two halves, sort the left and right halves recursively, and then merge the two sorted arrays using a standard merging procedure. This algorithm always takes $\mathcal{O}(nlog(n))$ time, but it requires some extra space to store the copied subarrays. It is in general much faster than the $\mathcal{O}(n^2)$ algorithms. The pseudocode for this algorithm is as follows [4]

[4]Modified from `https://en.wikipedia.org/wiki/Merge_sort`

```
function merge_sort(list m)
    if length of m  1 then
        return m

    var left := empty list
    var right := empty list
    for each x with index i in m do
        if i < (length of m)/2 then
            add x to left
        else
            add x to right

    left := merge_sort(left)
    right := merge_sort(right)

    return merge(left, right)
```

2.6 Quick Sort

Quick Sort is a very fast sorting algorithm that uses the divide and conquer methodology to achieve $\mathcal{O}(nlog(n))$ average case running time. The reason for it being quick is that the constants associated with it are much smaller than those associated with Merge Sort. This is because the merge operation, a heavy operation that slows Merge Sort down, is trivial in Quick Sort. The idea in Quick Sort is to select a pivot element, put all elements larger than it to its right, and smaller than it to its left, and then recursively sort the two subarrays. The pseudocode for Quick Sort is as follows [5] :

```
algorithm quicksort(A, lo, hi) is
    if lo < hi then
        p := partition(A, lo, hi)
        quicksort(A, lo, p  1)
        quicksort(A, p + 1, hi)

algorithm partition(A, lo, hi) is
```

[5]Cormen, Thomas H.; Leiserson, Charles E.; Rivest, Ronald L.; Stein, Clifford (2009) [1990]. "Quicksort". *Introduction to Algorithms (3rd ed.)*. MIT Press and McGraw-Hill. pp. 170 to 190

```
pivot := A[hi]
i := lo - 1
for j := lo to hi - 1 do
    if A[j]  pivot then
        i := i + 1
        swap A[i] with A[j]
swap A[i+1] with A[hi]
return i + 1
```

3 Observations

I implemented all the six algorithms and ran them on datasets ranging as high as 5 million integers, with random, ascending, and descending order elements. Here are the results, plotted in Google Sheets [6].

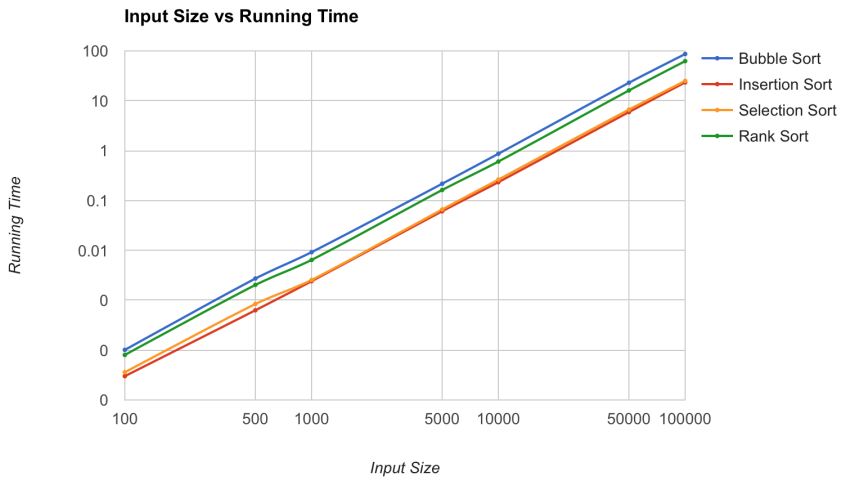

Figure 1. The $\mathcal{O}(n^2)$ on a random input

[6] For the raw data, see `https://docs.google.com/spreadsheets/d/1SNPT1LTcow5kVDeKIXEFPN6cPsf4cfh08tgTT1_FfjA/edit?usp=sharing`

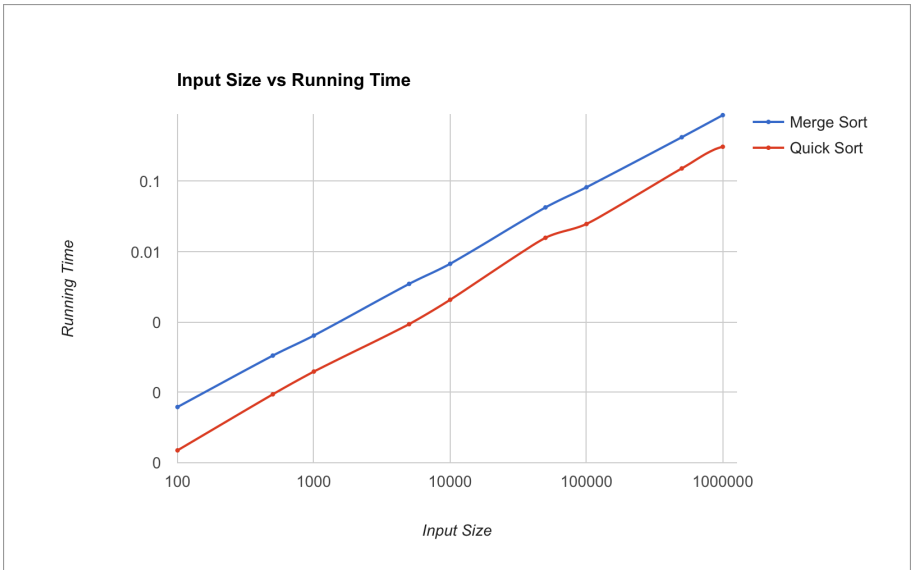

Figure 2. The $\mathcal{O}(nlog(n))$ on a random input

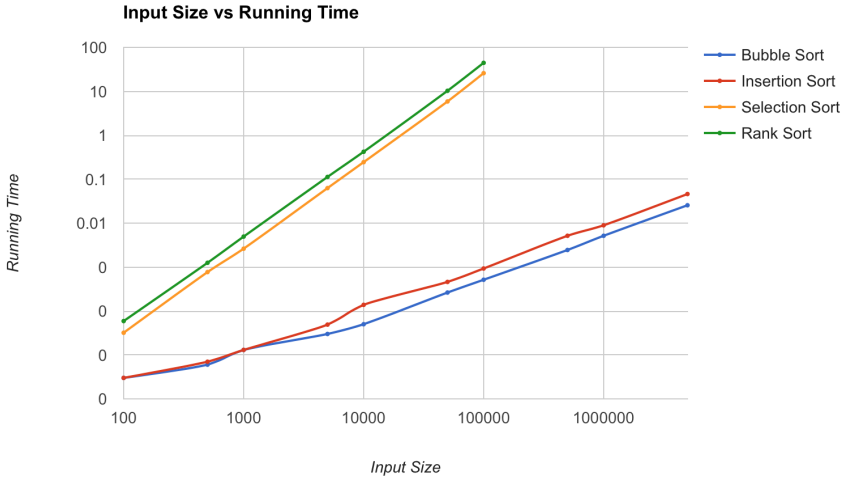

Figure 3. The $\mathcal{O}(n^2)$ on a ascending input

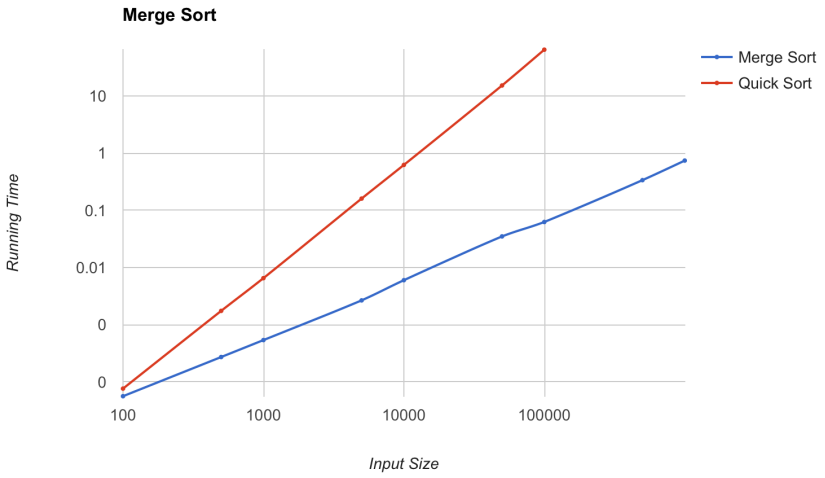

Figure 4. The $\mathcal{O}(nlog(n))$ on a ascending input

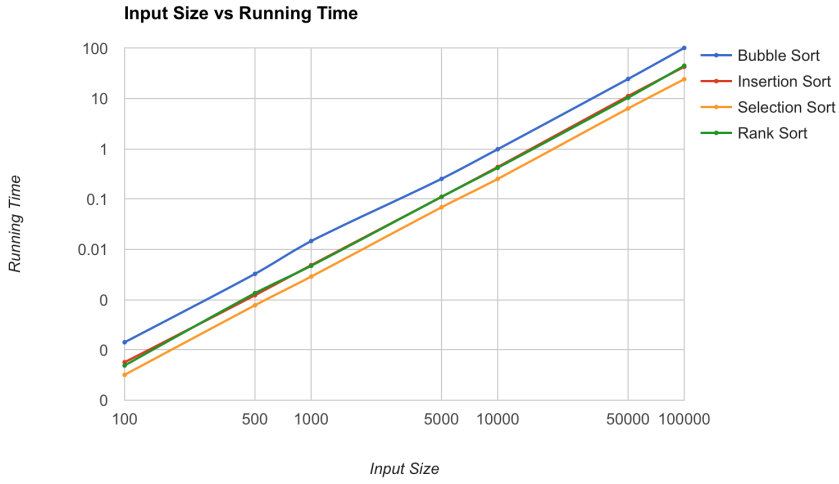

Figure 5. The $\mathcal{O}(n^2)$ on a descending input

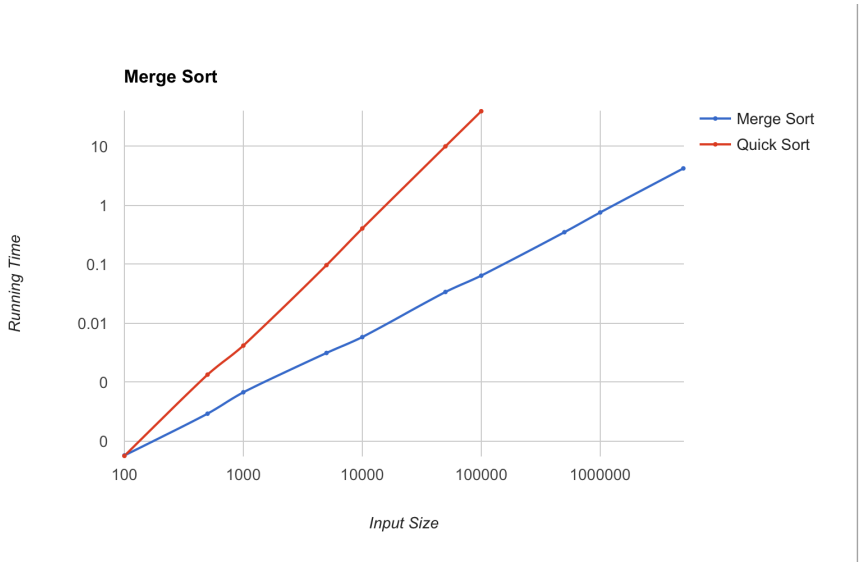

Figure 6. The $\mathcal{O}(nlog(n))$ on a descending input

Some things of note: (a) The graphs have been plotted on *log* vs *log* scale, as any other method would have resulted in very hard to read graphs. (b) the results which took longer than 100 seconds have been removed. However, as in the things tend to form straight lines, it is easy to project forward, and (c) Google Sheets does not allow labels for numbers smaller in magnitude than 0.01, so some of the labels say 0 when they should say 0.001, 0.0001 etc.

When running the random input on the $\mathcal{O}(n^2)$ algorithms, it is clear that Insertion sort is the quickest. Slightly slower than it on random input is Selection Sort. After a substantial gap is Rank Sort, followed by a somewhat smaller gap to Bubble Sort, the slowest of the four algorithms. The reasons for this were analysed in the *Theoretical Analysis* section.

Of the $\mathcal{O}(nlog(n))$ algorithms, on random input merge sort was clearly the quickest by a fairly large margin. However, both of these algorithms are much faster than the $\mathcal{O}(n^2)$ algorithms on random input, as they even managed to completely sort 5000000 integers very quickly.

When sorting ascending input (already sorted), Insertion Sort's advantage over Selection Sort is clear. It acts like an $\mathcal{O}(n)$ algorithm rather than an $\mathcal{O}(n^2)$ algorithm. Interestingly, Bubble Sort was the quickest of the $\mathcal{O}(n^2)$ algorithms, as it's comparisons are very simple and it doesn't do any other operations.

In the $\mathcal{O}(nlog(n))$ algorithms, Merge Sort acts like an $\mathcal{O}(nlog(n))$ algorithm while Quick Sort is much slower, and acts like an $\mathcal{O}(n^2)$ algorithm. The reason for Quick Sort's slowness is that every time it divides the array into sub-arrays where one sub-array only has 1 element, and the rest go into the other sub-array.

On Descending Inputs, in the $\mathcal{O}(n^2)$ algorithms, Selection Sort is clearly the quickest. The reason for this is that in Insertion sort, on Descending input, it has to swap many elements to get the element being inserted into it's correct position, where in Selection Sort this swap operation is simple, and only the comparisons take long. Rank Sort is interestingly enough similarly quick as Insertion Sort, and Bubble Sort is the slowest.

In the $\mathcal{O}(nlog(n))$ algorithms, Merge Sort is again much quicker than Quick Sort, for the same reasons as in the case of the ascending (already sorted) inputs.

4 Conclusions

Many of these sorting algorithms have practical applications. Quick Sort is the best general purpose sorting algorithm for random elements, as it is quickest by far. Therefore, many languages bundle Quick Sort as their default sorting algorithm. Merge Sort quite useful in cases where given arrays have a high likelihood of being both nearly sorted or being in random order as on average it is quicker when this is the case, because of Quick Sort's slowness in the already sorted cases. Both of these algorithms are also highly parallelisable, as they divide their tasks into sub tasks, which makes for a straightforward multithreading application.

Insertion Sort is very useful for simple, quick sorting of nearly sorted arrays, and is also useful when previous order needs to be maintained between equal elements (it is a stable sort, where the $\mathcal{O}(nlog(n))$ algorithms are unstable. Rank Sort is useful when the ranking data is needed after sorting. Bubble Sort can be useful when sorting almost sorted data exclusively, but can take up a lot of CPU time if a random array is given, and thus is usually avoided.

Miscellaneous References

1. All time and space complexities taken from class notes on given algorithms.

2. All execution of the algorithms was done on a MacBook Pro (Retina, 15-inch, Mid 2014).

3. Complete implementations of the algorithms and the data structures involved can be found in the attached source files.

4. The input data used was generated by a script. Both the data and the script are available at `https://github.com/sahilarora535/cs202-assignment02`.