CS202 Lab Assignment 02

Abhishek, B15103

March 6, 2017

1 Theoretical Analysis through Pseudo-Code

1.1 Insertion Sort

Insertion sort is a simple sorting algorithm that builds the final sorted array (or list) one item at a time. It is much less efficient on large lists than more advanced algorithms such as quicksort, heapsort, or merge sort. The implementation of this sort is simplest as opposed to other sorting algorithm. Insertion sort, as it's name says, is the algorithm which believes in inserting elements at their true position before going to the next step.

1.1.1 Algorithm

Insertion sort iterates, consuming one input element each repetition, and growing a sorted output list. Each iteration, insertion sort removes one element from the input data, finds the location it belongs within the sorted list, and inserts it there. It repeats until no input elements remain.

1.1.2 Pseudo-Code

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

1

1.1.3 Time Complexity

The average time complexity of Insertion Sort is $\Theta(n^2)$. If the list is already sorted (,i.e., the best case time complexity) then the time complexity is $\Omega(n)$. The worst case time complexity, however, is $O(n^2)$.

1.2 Selection Sort

Selection sort is a sorting algorithm, specifically an in-place comparison sort. It has $O(n^2)$ time complexity, making it inefficient on large lists, and generally performs worse than the similar insertion sort. Selection sort is noted for its simplicity, and it has performance advantages over more complicated algorithms in certain situations, particularly where auxiliary memory is limited.

1.2.1 Algorithm

The algorithm divides the input list into two parts: the sublist of items already sorted, which is built up from left to right at the front (left) of the list, and the sublist of items remaining to be sorted that occupy the rest of the list. Initially, the sorted sublist is empty and the unsorted sublist is the entire input list. The algorithm proceeds by finding the smallest (or largest, depending on sorting order) element in the unsorted sublist, exchanging (swapping) it with the leftmost unsorted element (putting it in sorted order), and moving the sublist boundaries one element to the right.

1.2.2 Pseudo-Code

```
SELECTION-SORT(A)
for j  1 to n-1
    smallest  j
    for i  j + 1 to n
        if A[ i ] < A[ smallest ]
        smallest  i
    Exchange A[ j ]  A[ smallest ]
```

[1]Pseudo code from Wikipedia - Insertion Sort

2

1.2.3 Time Complexity

Selection sort is not difficult to analyze compared to other sorting algorithms since none of the loops depend on the data in the array. Selecting the lowest element requires scanning all n elements (this takes $n1$ comparisons) and then swapping it into the first position. Finding the next lowest element requires scanning the remaining $n1$ elements and so on, for $(n1) + (n2) + ... + 2 + 1 = n(n-1)/2 \in \Theta(n2)$ comparisons (see arithmetic progression). Each of these scans requires one swap for $n1$ elements (the final element is already in place).

1.3 Bubble Sort

Bubble sort, sometimes referred to as sinking sort, is a simple sorting algorithm that repeatedly steps through the list to be sorted, compares each pair of adjacent items and swaps them if they are in the wrong order. The pass through the list is repeated until no swaps are needed, which indicates that the list is sorted. The algorithm, which is a comparison sort, is named for the way smaller or larger elements "bubble" to the top of the list. Although the algorithm is simple, it is too slow and impractical for most problems even when compared to insertion sort. It can be practical if the input is usually in sorted order but may occasionally have some out-of-order elements nearly in position.

1.3.1 Algorithm

This sorting algorithm swaps each adjacent element until they are placed in their order. So, first, it goes and keeps on swapping until the least element is in place, the second least element is in place and so on.

1.3.2 Pseudo-Code

```
procedure bubbleSort( A : list of sortable items )
                n = length(A)
                repeat
```

[2]Pseudo code from Wikipedia - Selection Sort

```
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
```

[3]

1.3.3 Time Complexity

Bubble sort has worst-case and average complexity both (n^2), where n is the number of items being sorted. There exist many sorting algorithms with substantially better worst-case or average complexity of $O(n \log n)$. Even other (n^2) sorting algorithms, such as insertion sort, tend to have better performance than bubble sort. Therefore, bubble sort is not a practical sorting algorithm when n is large.

1.4 Rank Sort

Rank Sort is another simple sorting algorithm, which sorts on the basis of ranks of the element formed by dictionary order. The algorithm

1.4.1 Algorithm

The algorithm forms a dictionary for itself based on the ranks or the weights of the element. Each element is compared with one another, and if they are greater then the element, then their rank is increased. Otherwise, the rank is retained. At last, after doing this operation on all the elements, we obtain a sorting index of each element.

[3]Pseudo code from Wikipedia - Bubble Sort

1.4.2 Psuedo-Code

```
rankArray [n]
initialize all ranks to 0
for i = 1 to n inclusive do
    for j = i to n inclusive do
        if A[i]>A[j]
         rank[i] = rank[i]+1
sort the array based on the rank array
```

1.4.3 Time Complexity

As this algorithm has to traverse the whole list in a double loop every time to create the rank of the elements, the time complexity of this algorithm in all cases is $O(n^2)$.

1.5 Merge Sort

Merge sort (also commonly spelled mergesort) is an efficient, general-purpose, comparison-based sorting algorithm. Most implementations produce a stable sort, which means that the implementation preserves the input order of equal elements in the sorted output. Mergesort is a divide and conquer algorithm.

1.5.1 Algorithm

Conceptually, a merge sort works as follows:

- Divide the unsorted list into n sublists, each containing 1 element (a list of 1 element is considered sorted).
- Repeatedly merge sublists to produce new sorted sublists until there is only 1 sublist remaining. This will be the sorted list.

1.5.2 Pseudo-Code

```
MERGE-SORT(A, p, r)
    If p < r
    q =  [ ( p + q ) /2 ]
    MERGE-SORT(A, p, q)
    MERGE-SORT(A, q+1, r)
    MERGE(A, p, q, r)
```

```
MERGE (A, p, q, r)

    n1 = q  p +1
    n2 = r  q
    let L [1.. n1 + 1 ] and L [1.. n2 + 1 ]  be new arrays
    for i=1 to  n1
    L[ i ]  =  A [ p + i -1]
    for j=1 to n2
    R[ j ]= A[ q + j ]
    L [n1 + 1 ] =
    R [n2 + 1 ] =
    i = 1
    j = 1
    for k = p to r
    if L[ i ] < R [ j ]
    A[ k ] = L[ i ]
    i = i + 1
    else  A[ k ] = R [ j ]
    j = j + 1
```

[4]

1.5.3 Time Complexity

In sorting n objects, merge sort has an average and worst-case performance of $O(n \log n)$. If the running time of merge sort for a list of length n is $T(n)$, then the recurrence $T(n) = 2T(n/2) + n$ follows from the definition of the algorithm (apply the algorithm to two lists of half the size of the original list, and add the n steps taken to merge the resulting two lists). The closed form follows from the master theorem.

1.6 Quick Sort

Quicksort (sometimes called partition-exchange sort) is an efficient sorting algorithm, serving as a systematic method for placing the elements of an array in order. When implemented well, it can be about two or three times faster than its main competitors, merge sort and heapsort.

[4]Pseudo code from Wikipedia - Merge Sort

1.6.1 Algorithm

Quicksort is a divide and conquer algorithm. Quicksort first divides a large array into two smaller sub-arrays: the low elements and the high elements. Quicksort can then recursively sort the sub-arrays. The steps are :

- Pick an element, called a pivot, from the array.
- Partitioning: reorder the array so that all elements with values less than the pivot come before the pivot, while all elements with values greater than the pivot come after it (equal values can go either way). After this partitioning, the pivot is in its final position. This is called the partition operation.
- Recursively apply the above steps to the sub-array of elements with smaller values and separately to the sub-array of elements with greater values.

The base case of the recursion is arrays of size zero or one, which never need to be sorted. The pivot selection and partitioning steps can be done in several different ways; the choice of specific implementation schemes greatly affects the algorithm's performance.

1.6.2 Pseudo-Code

```
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
    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
```

5

1.6.3 Time Complexity

Mathematical analysis of quicksort shows that, on average, the algorithm takes $O(n \log n)$ comparisons to sort n items. In the worst case, it makes $O(n^2)$ comparisons, though this behavior is rare.

2 Practical Analysis

2.1 Ascending Order Data

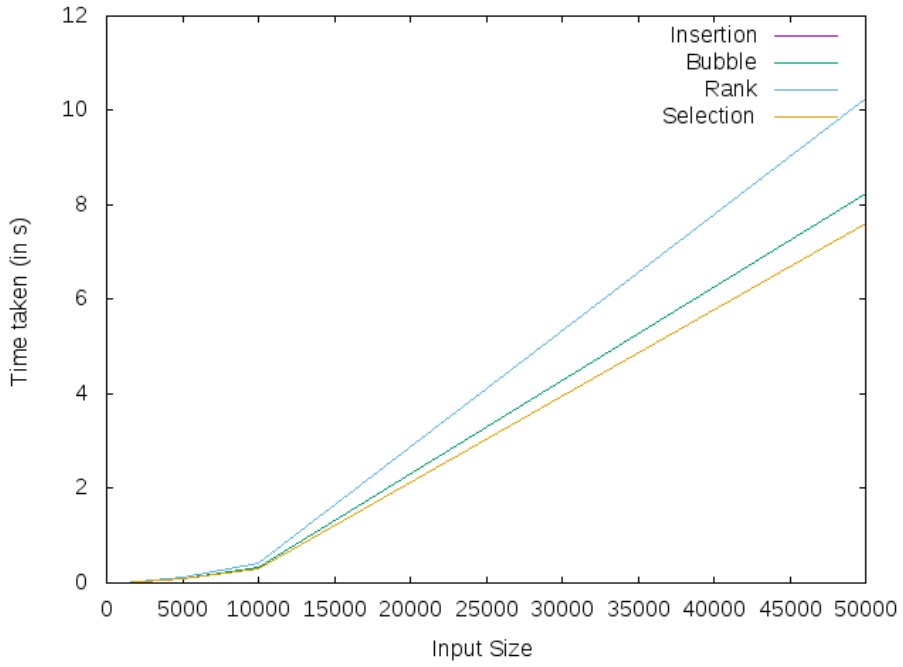

Figure 1. Ascending Order Data for $O(n^2)$ algorithms Insertion Sort happens in $O(n)$. But all other algorithms have complexity in $O(n^2)$.

[5]Pseudo code from Wikipedia - Quick Sort

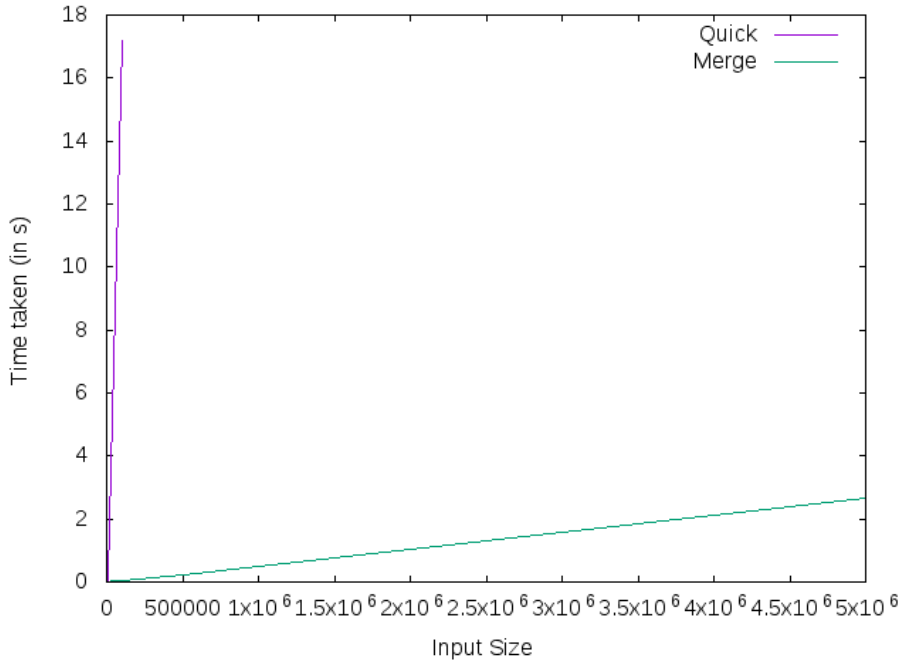

Figure 2. Ascending Order Data for $O(n \log n)$ algorithms

In ascending order inputs, insertion sort is the fastest of all sorting algorithms. The order is as follows :

$$Insertion < Merge < Selection < Rank < Bubble < Quick$$

2.2 Descending Order Data

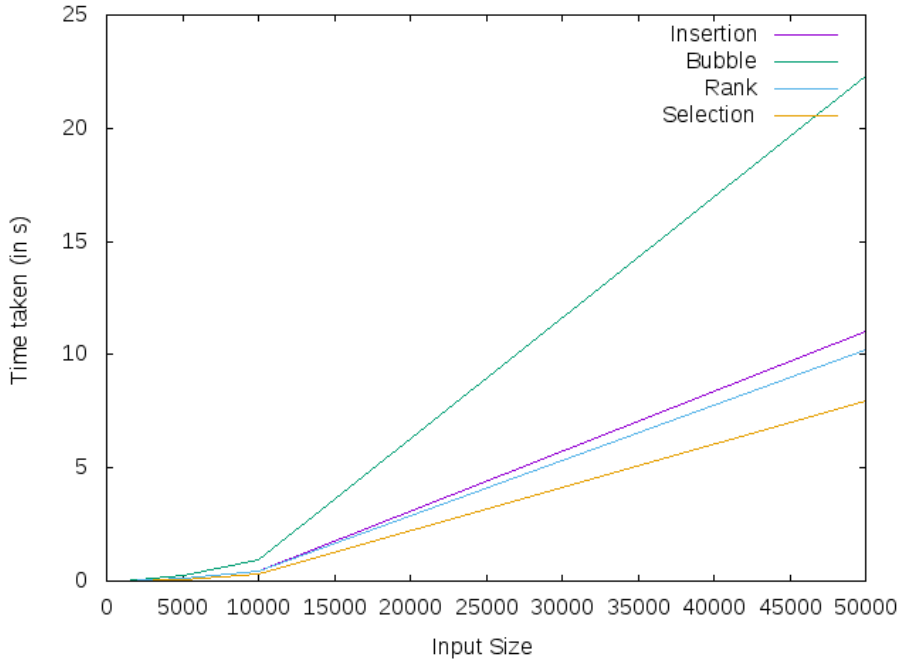

Figure 3. Descending Order Data for $O(n^2)$ algorithms

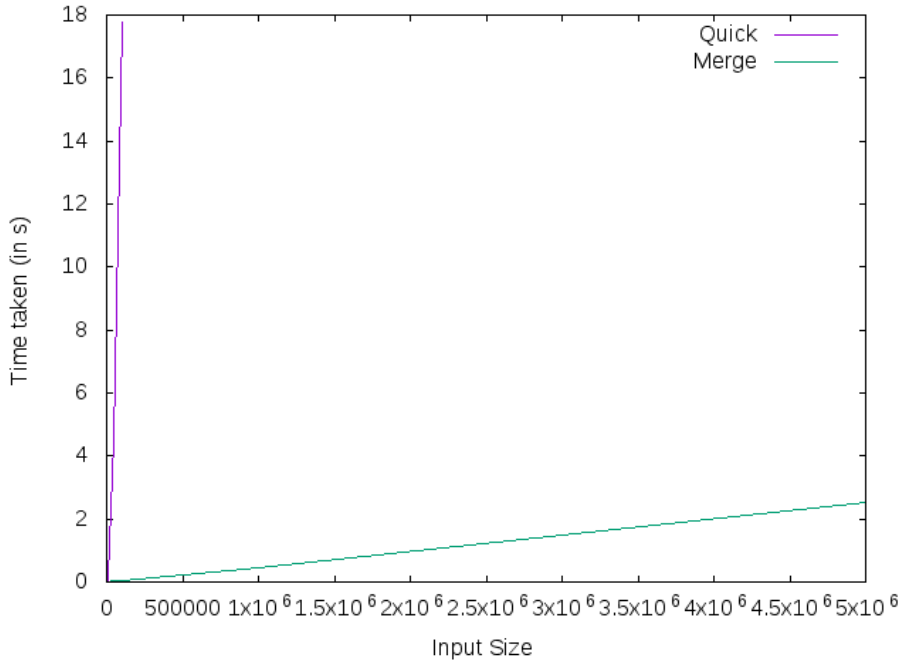

Figure 4. Descending Order Data for $O(n \log n)$ algorithms

In descending order inputs, insertion sort is the fastest of all sorting algo-

rithms. The order is as follows :

$$Merge < Insertion < Selection < Rank < Bubble < Quick$$

2.3 Random Order Data

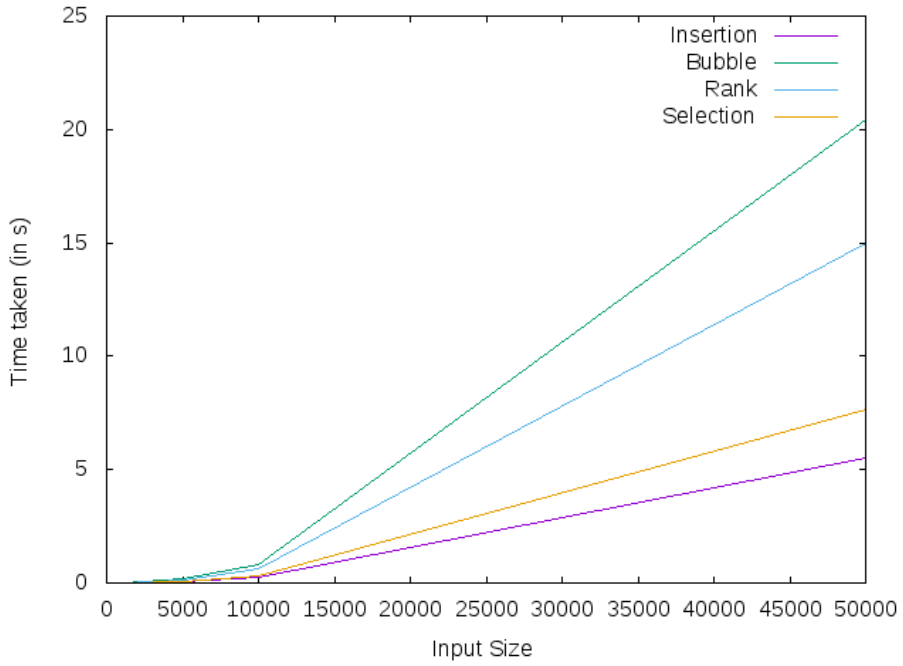

Figure 5. Random Order Data for $O(n^2)$ algorithms

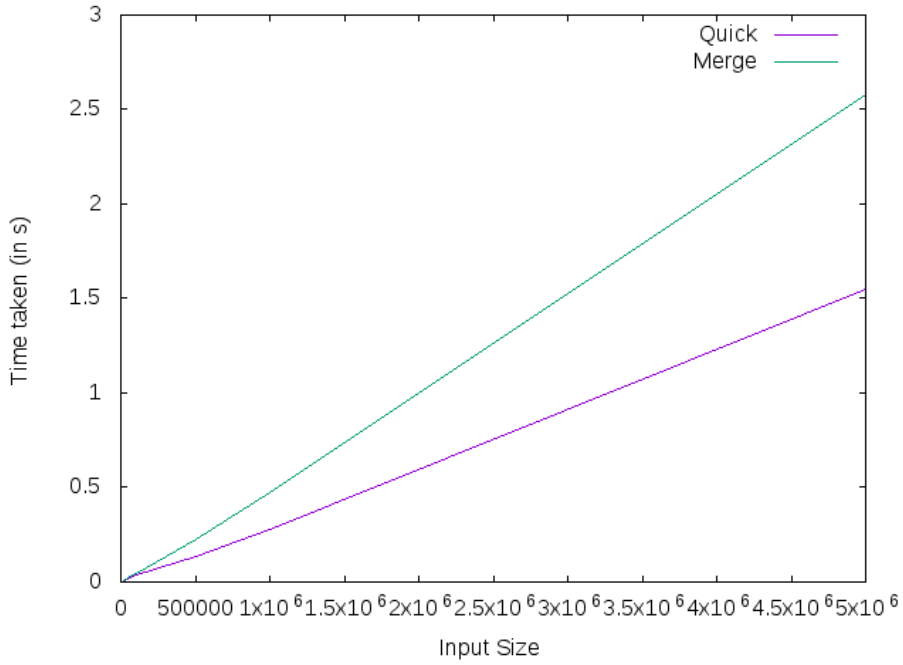

Figure 6. Random Order Data for $O(n \log n)$ algorithms

In random order inputs, quick sort is the fastest of all sorting algorithms. The order is as follows :

$$Quick < Merge < Insertion < Selection < Rank < Bubble$$