Publications and Research

New York City College of Technology

2021

An Analysis of Comparison-based Sorting Algorithms

Jacob M. Gomez
CUNY New York City College of Technology

Edgar Aponte
CUNY New York City College of Technology

Brad Isaacson
CUNY New York City College of Technology

"An Analysis of Comparison-based Sorting Algorithms"

By: Jacob Gomez , Edgar Aponte, Brad Isaacson

Abstract

Sorting algorithms put elements of a list into an order (e.g., numerical, alphabetical). Sorting is an important problem because a nontrivial percentage of all computing resources are devoted to sorting all kinds of lists. For our project, we implemented 24 comparison-based sorting algorithms from pseudocode and compared them. The algorithms all have their advantages and disadvantages as well as their unique features. We found that introspective sort (which is a modified version of heapsort) and timsort (which is a modified version of mergesort) were the most efficient.

Introduction

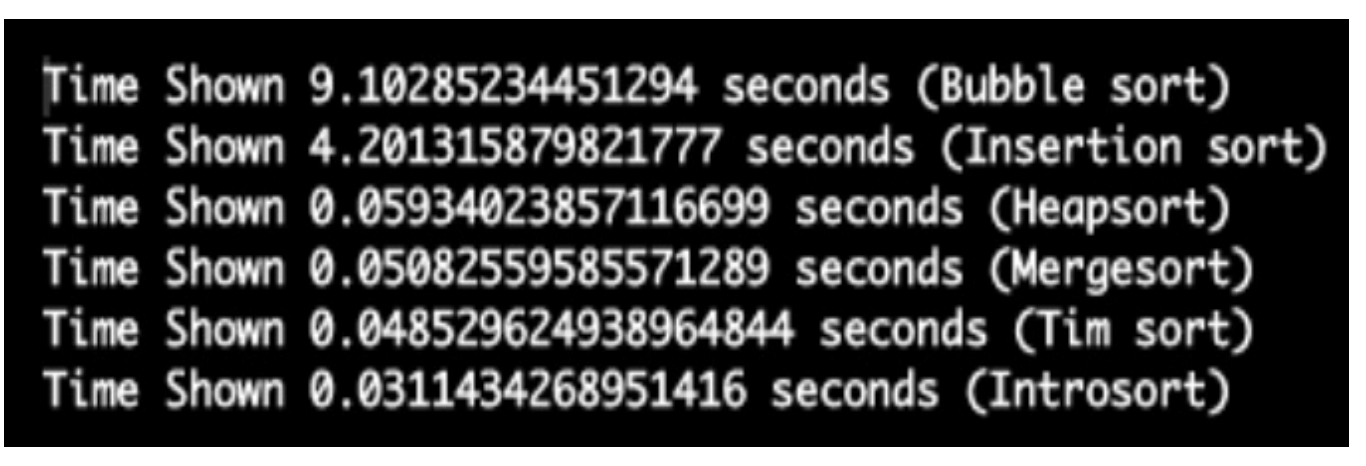
A number of different algorithms have been invented for sorting. Their relative advantages and disadvantages have been studied extensively. Out of the 24 that we implemented, we chose 6 to discuss in this poster: (1) bubble sort, (2) insertion sort, (3) heapsort, (4) mergesort, (5) timsort, and (6) introspective sort. Although the latter four sorting algorithms are the fastest by virtue of having the optimal worst-case time complexity of *O(n log(n))*, bubble sort and insertion sort are straightforward algorithms in which all Computer Science and Applied Mathematics majors should know. For our project, we created an application which included 24 sorting algorithms. We analyzed the running times of these sorting algorithms with various sets of unsorted data. Through this process, we gained a better understanding of the various sorting algorithms and their time complexities. In conclusion, introspective sort and timsort were the fastest and most efficient sorting algorithms we tested, with introspective sort being the very fastest.

Process

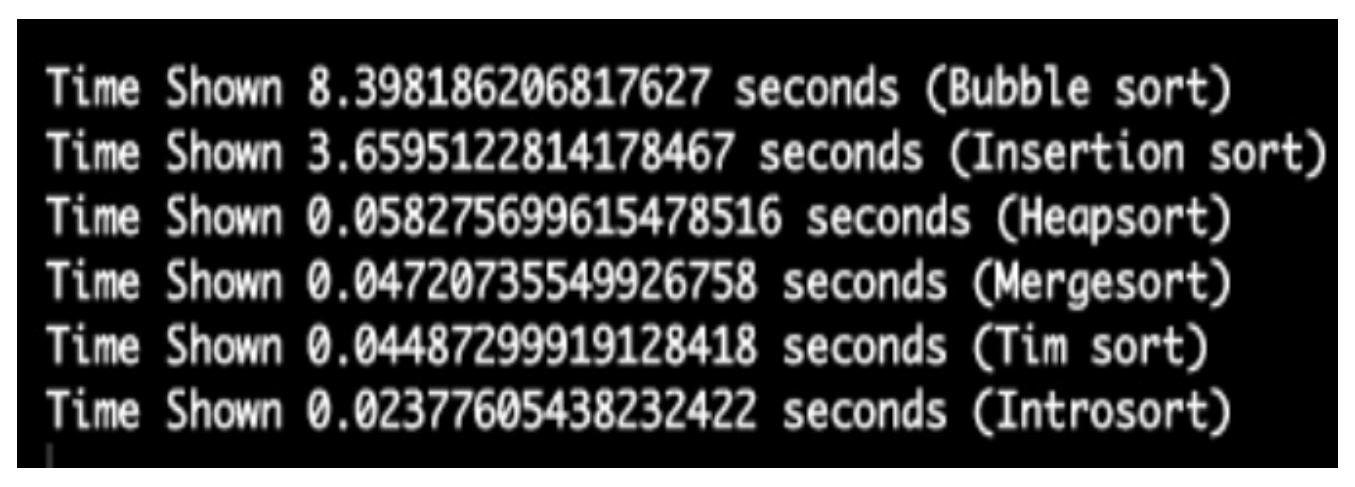
The application we used to compile the algorithms was JDoodle. This is an online site where you can run and write out different languages of code like Java, C++, Python and many more. The only limitation on this site is the amount of memory you're given, so printing large arrays caused run-time errors. Nonetheless, we were able to calculate the running time for the various sorting algorithms we implemented. Although we tested various inputs, we preferred to use randomly generated lists of 10,000 integers. The pseudocodes for the 6 aforementioned sorting algorithms we implemented can be found on the right side of this poster.

Results

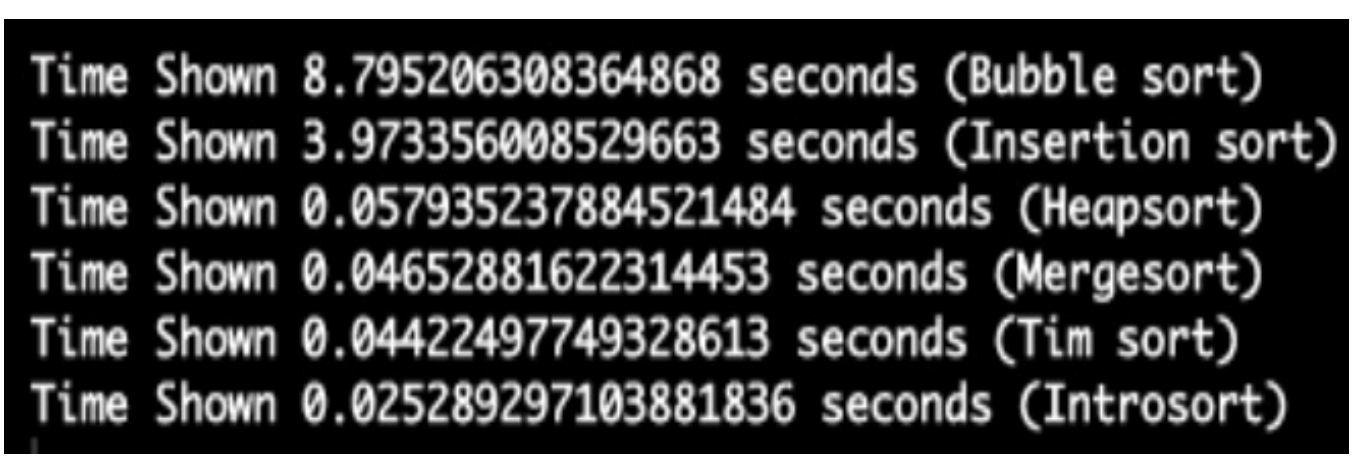
We tested 6 sorting algorithms: (1) bubble sort, (2) insertion sort, (3) heapsort, (4) mergesort, (5) timsort, and (6) introspective sort. Below are the running times of the 6 sorting algorithms using randomly generated unsorted lists of 10,000 integers as input. Each box corresponds to the same input.



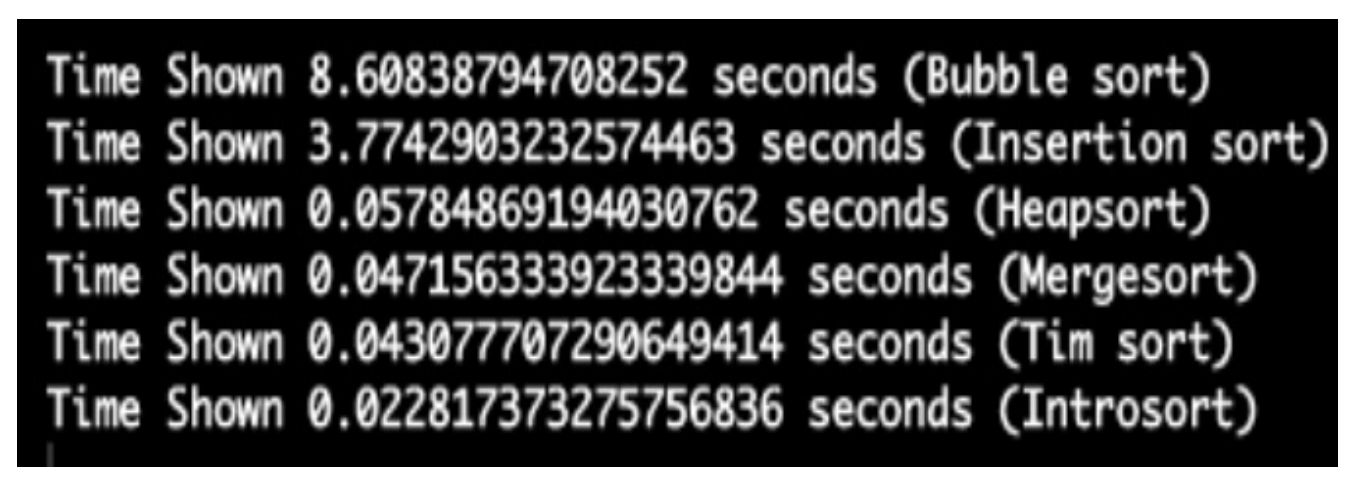
```
Time Shown 9.10285234451294 seconds (Bubble sort)
Time Shown 4.201315879821777 seconds (Insertion sort)
Time Shown 0.05934023857116699 seconds (Heapsort)
Time Shown 0.05082559585571289 seconds (Mergesort)
Time Shown 0.04852962493896484 seconds (Tim sort)
Time Shown 0.0311434268951416 seconds (Introsort)
```



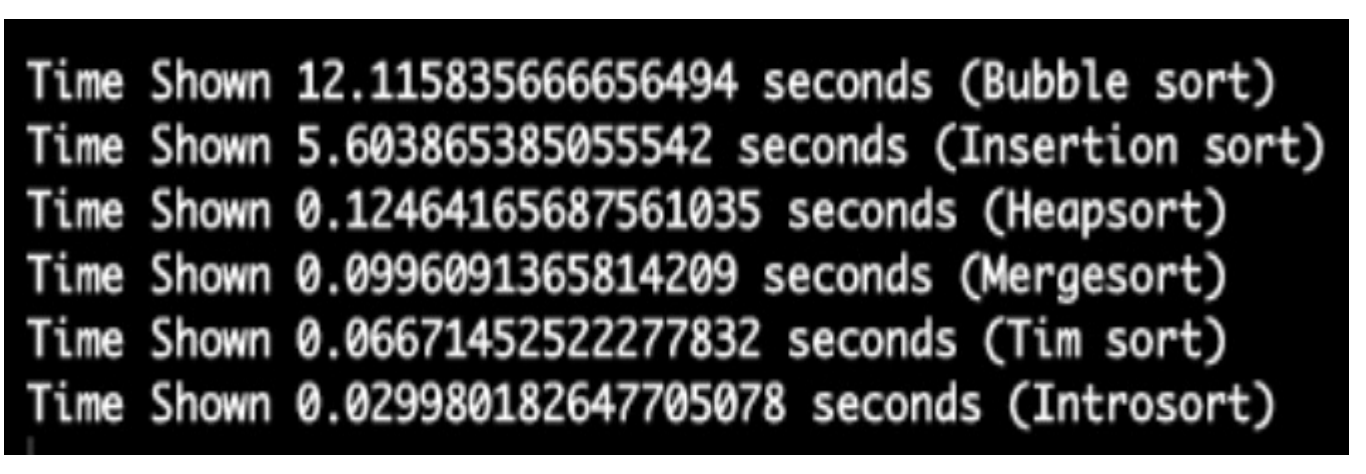
```
Time Shown 8.398186206817627 seconds (Bubble sort)
Time Shown 3.6595122814178467 seconds (Insertion sort)
Time Shown 0.058275699615478516 seconds (Heapsort)
Time Shown 0.04720735549926758 seconds (Mergesort)
Time Shown 0.04487299919128418 seconds (Tim sort)
Time Shown 0.02377605438232422 seconds (Introsort)
```



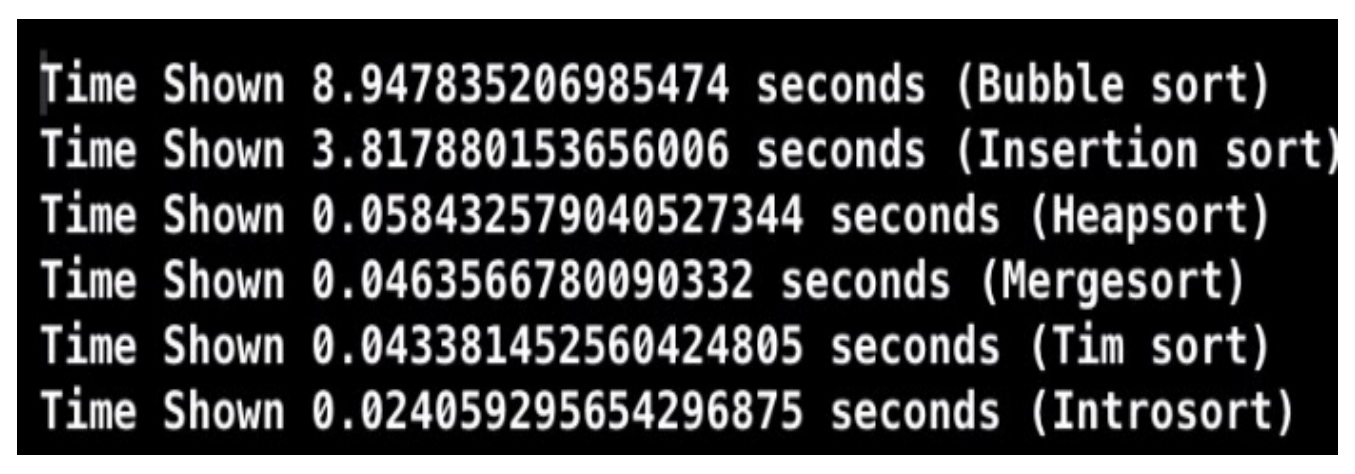
```
Time Shown 8.795206308364868 seconds (Bubble sort)
Time Shown 3.973356008529663 seconds (Insertion sort)
Time Shown 0.05793523788452148 seconds (Heapsort)
Time Shown 0.04652881622314453 seconds (Mergesort)
Time Shown 0.04422497749328613 seconds (Tim sort)
Time Shown 0.025289297103881836 seconds (Introsort)
```



```
Time Shown 8.60838794708252 seconds (Bubble sort)
Time Shown 3.7742903232574463 seconds (Insertion sort)
Time Shown 0.05784869194030762 seconds (Heapsort)
Time Shown 0.047156333923339844 seconds (Mergesort)
Time Shown 0.04307770729064941 seconds (Tim sort)
Time Shown 0.022817373275756836 seconds (Introsort)
```



```
Time Shown 12.115835666656494 seconds (Bubble sort)
Time Shown 5.603865385055542 seconds (Insertion sort)
Time Shown 0.12464165687561035 seconds (Heapsort)
Time Shown 0.0996091365814209 seconds (Mergesort)
Time Shown 0.06671452522277832 seconds (Tim sort)
Time Shown 0.029980182647705078 seconds (Introsort)
```



```
Time Shown 8.947835206985474 seconds (Bubble sort)
Time Shown 3.817880153656006 seconds (Insertion sort)
Time Shown 0.05843257904052734 seconds (Heapsort)
Time Shown 0.046356678009033203 seconds (Mergesort)
Time Shown 0.043381452560424805 seconds (Tim sort)
Time Shown 0.024059295654296875 seconds (Introsort)
```

Discussion

Based on the results, we found that introspective sort and timsort were the fastest and most efficient. We also found that bubble sort and insertion sort were the slowest and least efficient. Interestingly, bubble sort performed much worse than insertion sort despite them both having the same worst-case time complexity of *O(n^2)*. Heapsort and mergesort were average. Timsort and introspective sort were impressive because they were much faster than heapsort and mergesort for most inputs despite them all having worst-case time complexity of *O(n log(n))*. Our findings are consistent with timsort and introspective sort being used as the industry standard. What we found most surprising is that introspective sort was the fastest algorithm of them all, being roughly twice as fast as timsort in all of our experiments!

Timsort is a hybrid sorting algorithm derived from mergesort and insertion sort and is designed to perform well on many kinds of real-world data. This is the default sorting algorithm in Python and Java.

Introspective sort is also a hybrid sorting algorithm that provides both fast average performance and optimal worst-case performance. Our implementation essentially calls quicksort (which is a divide-and-conquer sorting algorithm using pivot points and partitions) until the recursion depth reaches a certain level, and then calls heapsort. Further refinements incorporate a third sorting algorithm, insertion sort. Introspective sort is the default sorting algorithm in C++.

Bubble Sort

```
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
```

It repeatedly steps through the list, compares adjacent elements and swaps them if they are in the wrong order. It repeats until the list is sorted.

Heapsort

```
//(Put elements of 'a' in heap order, in-place)
procedure heapify(a, count) is
    //(start is assigned the index in 'a' of the last parent node)
    //(the last element in a 0-based array is at index count-1; find the parent of that element)
    start ← iParent(count-1)

    while start ≥ 0 do
        //(sift down the node at index 'start' to the proper place such that all nodes below
        //the start index are in heap order)
        siftDown(a, start, count - 1)
        //(go to the next parent node)
        start ← start - 1
    //(after sifting down the root all nodes/elements are in heap order)

//(Repair the heap whose root element is at index 'start', assuming the heaps rooted at its children are valid)
procedure siftDown(a, start, end) is
    root ← start

    while iLeftChild(root) ≤ end do    //(While the root has at least one child)
        child ← iLeftChild(root)   //(Left child of root)
        swap ← root                //(Keeps track of child to swap with)

        if a[swap] < a[child] then
            swap ← child
        //(If there is a right child and that child is greater)
        if child+1 ≤ end and a[swap] < a[child+1] then
            swap ← child + 1
        if swap = root then
            //(The root holds the largest element. Since we assume the heaps rooted at the
             //children are valid, this means that we are done.)
            return
        else
            swap(a[root], a[swap])
            root ← swap            //(repeat to continue sifting down the child now)

procedure heapsort(a, count) is
    input: an unordered array a of length count

    //(Build the heap in array a so that largest value is at the root)
    heapify(a, count)

    //(The following loop maintains the invariants that a[0:end] is a heap and every element
     //beyond end is greater than everything before it (so a[end:count] is in sorted order))
    end ← count - 1
    while end > 0 do
        //(a[0] is the root and largest value. The swap moves it in front of the sorted elements.)
        swap(a[end], a[0])
        //(the heap size is reduced by one)
        end ← end - 1
        //(the swap ruined the heap property, so restore it)
        siftDown(a, 0, end)
```

It divides its input into a sorted and an unsorted region and it repetitively shrinks the unsorted region by extracting the largest element from it and inserting it into the sorted region. Heapsort maintains the unsorted region in a heap data structure by calling heapify() to more quickly find the largest element in each step.

Timsort

```
// Iterative Timsort function to sort the
// array[0...n-1] (similar to merge sort)
def timSort(arr):
    n = len(arr)
    minRun = calcMinRun(n)

    # Sort individual subarrays of size RUN
    for start in range(0, n, minRun):
        end = min(start + minRun - 1, n - 1)
        insertionSort(arr, start, end)

    // Start merging from size RUN (or 32). It will merge
    // to form size 64, then 128, 256 and so on ....
    size = minRun
    while size < n:

        // Pick starting point of left sub array. We
        // are going to merge arr[left..left+size-1]
        // and arr[left+size, left+2*size-1]
        // After every merge, we increase left by 2*size
        for left in range(0, n, 2 * size):

            // Find ending point of left sub array
            // mid+1 is starting point of right sub array
            mid = min(n - 1, left + size - 1)
            right = min((left + 2 * size - 1), (n - 1))

            // Merge sub array arr[left.....mid] &
            // arr[mid+1....right]
            if mid < right:
                merge(arr, left, mid, right)

        size = 2 * size
```

It iterates over the data collecting elements into runs and simultaneously puts those runs in a stack. Whenever the runs on the top of the stack match a merge criterion, they are merged. This goes on until all data is traversed. All runs are then merged two at a time until one sorted run remains.

Insertion Sort

```
function insertionSortR(array A, int n)
    if n > 0
        insertionSortR(A, n-1)
        x ← A[n]
        j ← n-1
        while j >= 0 and A[j] > x
            A[j+1] ← A[j]
            j ← j-1
        end while
        A[j+1] ← x
    end if
end function
```

It iterates, consuming one input element each repetition and grows a sorted output list. At 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.

Mergesort

```
function merge_sort(list m) is
    // Base case. A list of zero or one elements is sorted, by definition.
    if length of m ≤ 1 then
        return m

    // Recursive case. First, divide the list into equal-sized sublists
    // consisting of the first half and second half of the list.
    // This assumes lists start at index 0.
    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

    // Recursively sort both sublists.
    left := merge_sort(left)
    right := merge_sort(right)

    // Then merge the now-sorted sublists.
    return merge(left, right)

function merge(left, right) is
    var result := empty list

    while left is not empty and right is not empty do
        if first(left) ≤ first(right) then
            append first(left) to result
            left := rest(left)
        else
            append first(right) to result
            right := rest(right)

    // Either left or right may have elements left; consume them.
    // (Only one of the following loops will actually be entered.)
    while left is not empty do
        append first(left) to result
        left := rest(left)
    while right is not empty do
        append first(right) to result
        right := rest(right)
    return result
```

It divides the unsorted list into *n* sublists, each containing one element. It repeatedly merges sublists to produce new sorted sublists until there is only one sublist remaining.

Introspective Sort

```
procedure sort(A : array):
    let maxdepth = ⌊log(length(A))⌋ × 2
    introsort(A, maxdepth)

procedure introsort(A, maxdepth):
    n ← length(A)
    if n ≤ 1:
        return  // base case
    else if maxdepth = 0:
        heapsort(A)
    else:
        p ← partition(A)  // assume this function does pivot selection, p is the final position of the pivot
        introsort(A[0:p-1], maxdepth - 1)
        introsort(A[p+1:n], maxdepth - 1)
```

It selects a 'pivot' element from the array and partitions 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 by calling heapsort() when the recursion depth reaches a level based on the logarithm of the number of elements to be sorted.