Manjesh Prasad

Nov 26, 2020

Purpose: The main purpose of this lab is to use a set of integers to find out the algorithm

Background: In this lab, we apply our theoretical understanding of sort detection using various sorting algorithms which include quick, bubble, insertion, selection, merge, and heap sort. In order to apply our understanding, I was given 6 mysterious algorithms and detect what sorting algorithm goes with which mysterious algorithm. Although all 6 sorting algorithms do eventually organize the set of integers in ascending groups, they differ in terms of how many comparisons and the moves. In this lab, although time complexity is important to distinguish between each sort, it is not sufficient in the lab because it varies among every computer that is being tested. For example, the time complexity when I run the test on my computer is different than if I use a school desktop. In this lab, I tested various sets of integers in order to analyze the data performance of the mysterious algorithm. In order to understand this lab, we must distinguish every single sort based on its performance, ‘how they achieve their sorts,’ and how it makes the comparisons.

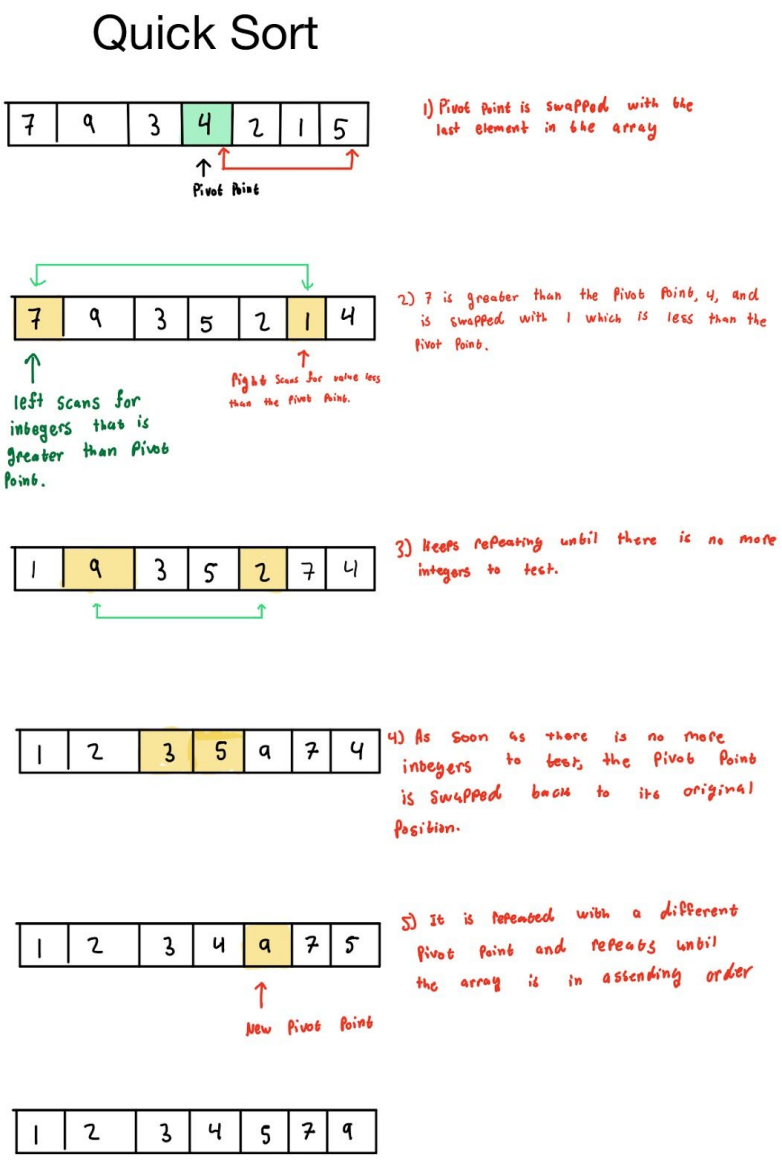
In a **quick sort**, is a recursive sorting algorithm that uses the method of “dividing-and-conquering" in order to organize the set of integers into an ascending group. In quick sort, a pivot point is chosen among the set (usually the integer in the center) and is temporarily swapped with the last integer on the set. It then compares the integers from the left to see if it is greater than the integer, and compares the integer from the right, and detects if it is

less than the pivot point. It is chosen true, it swaps both values from the point and repeats until everything is tested. According to the data, Quick Sort tends to be the fastest sorting algorithm when being compared to the other 6

Average big O: (n log n).

Worst big O: O(n^2)

Best case: O(n log n)



The figure above indicates a visual representation of what quicksort is.

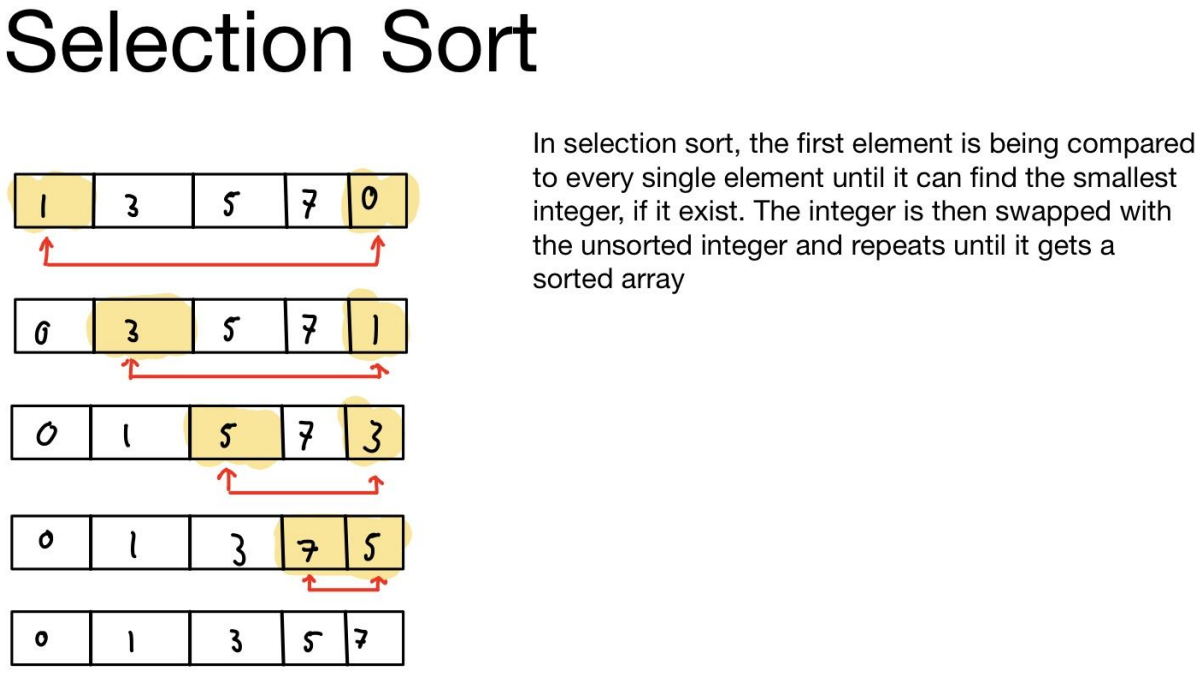
In **heap sort**, it utilizes the heap in order to organize elements, and uses a tree in search of the maximum value. The value of the parent nodes is greater than the value of the child nodes and organizes the tree in the ‘max heap’ in order to organize elements from the largest to smallest. It then takes the maximum node from the tree and deletes it from the tree, adds the value onto the head, and repeats the process all over again until there are no more nodes left.

The average case: O(n log n)

Best case: O(n log n)

Worst case: O(n log n)

In **selection sort**, the array is iterated in order to find the smallest value. As soon as the smallest value is found, the integer is swapped to the integer that is unsorted. A better way to understand this is is the example I have drawn:



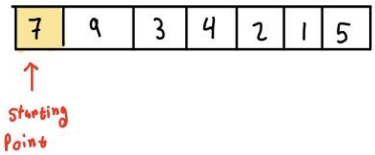
Insertion sort is one of the simplest sorting algorithms that is commonly used to organize arrays. During iteration, the first array is compared to its processors, and it it;s processors are smaller than it is placed on the appropriate placement.

Average Big O: O(n^2)

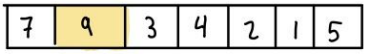
Worst Case: O (n^2)

Best Case O(n)

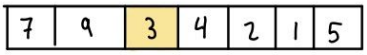
Insertion sort



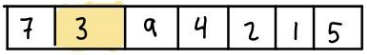
During insertion, it starts off the first integer and organize the element to it's left value, if it out of place, then 'inserts' to appropriate value. In this case, 7 has no value to it's left, therefore, it is in a 'correct' position.



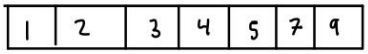
9 is greater than it's left element, 7, so therefore it is in a correct position as well.



3 is less than it's left element, 9, so it 'inserts' and swap with 9



3 is still less than it's left element, 7, so it 'insert' and swap with the position of 7. This repeats until the entire list in assenting order.

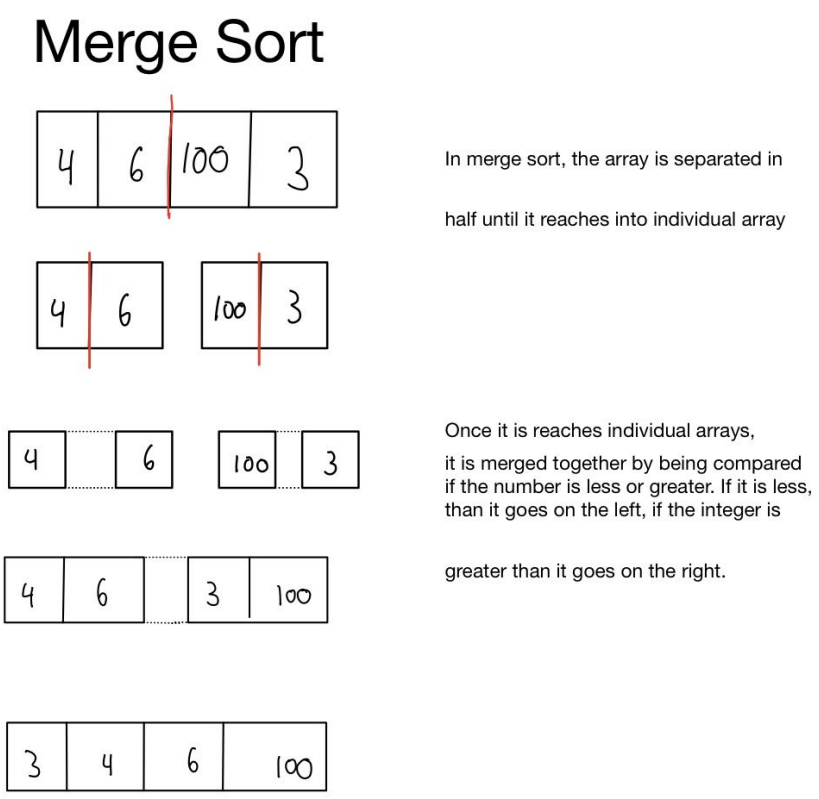


In a **merge sor**t, usually done recursively, where it uses a strategy of “divide-and-conquer.” We divide the entire array in half until we are left with individual items. Every individual item are compared and are sorted and merge into temporary arrays until we are left with an entire sorted array.

Average case: O(n log n)

Best case: O(n log n)

Worst case: O(n log n)



In a Bubble sort, compare consecutive items. The highest number will partition it's way to the right side, and repeat in the beginning until the entire list is sorted.

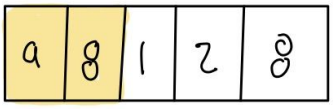
In my attempt, I used a sorted set of integers of:

Average Big O: O(n^2)

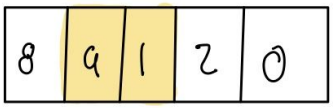
Worst case: O(n^2)

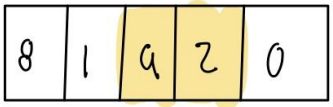
Best case: O(n)

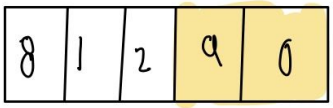
Bubble sort

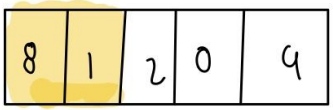


Bubble sorting is when the left of the element is being compared to it's right. If the left element is greater than the element on the right, then it is swapped. It repeats until the element that is greater is at the end of the array.









After the 9 goes at the end, it starts over until the list is organized increasingly.

By theory, the sorting algorithms that are simpler to implement and compare are Bubble, Insertion, and Selection; while the algorithms that require more of a recursive call are Heap, Quick, and Merge. When I was conducting the sort detection, I paid close attention to the number of moves and comparisons using different sets of integers. In other words, I tested sets that were already sorted, a set that had the same integers as the first set, but unsorted, and sets that contain a different set of integers. Here I will present all the trials that signify my conclusion.

“Dumb” Vs. “Smart”:

Although every sorting algorithms are not dumb or smart, they are seperated according to their reliability:

Dumb	**Smart**
Usually the in average case, the Big O is: O(n^2) Generally slower than Usually passes through the array once Bubble, Insertion, and Selection sort	Usually the in average case, the Big O is: O(n log n) Usually less moves and comparisons (at least in my data) Quick, Heap, and Insertion sort

Based on my results I conclude that :

Algorithm 1: Quicksort

Algorithm 2: Heapsort

Algorithm 3: Insertion sort

Algorithm 4: Merge sort

Algorithm 5: Bubble sort

Algorithm 6: Selection sort

Trial 1:

In this test, I used a simply ordered set of integers to indicate an idea of what possible sort is being used.

{ 1, 2, 3, 4, 5, 6 }

	ms	mv	comps
alg1	0	5	25
alg2	0	42	16
alg3	0	0	5
alg4	0	32	9
alg5	0	1	6
alg6	0	15	15

According to the data, Alg 5 is the second to least amount of comparisons; in bubble sort, every item is being compared to its neighboring element (both right and left) and are swapped if the items are misplaced. Since the list that is being tested is already sorted, then little to no moves are taken. In theory, Alg 5 should be bubble sort.

Based on my data, alg1 is making comparisons and moves in order to receive an ordered list. In this algorithm, the amount of moves is smaller than the number of comparisons. This has to be an algorithm that has to be recursive dependent. I theorized that if this is quicksort, then pivot-point is selected and uses ‘divide and conquer ’ has to be and contribute to

the number of comparisons. Since the entire array is sorted, quick sort still has to compare every single integer to see whether it is greater than or less than the pivot point. Therefore, I believe algorithm 1 is quicksort.

In Alg6, its moves and comparisons are equal to each other. In theory, I speculated that Algorithm 6 cannot possibly be a heap or merge sort where its believed to use the most moves in a sorted set to see if the integers are in increasing order. Selection sort iterates through the entire array to see if there is a minimum value in comparison to the element it is comparing to. For example, when it starts at 1, it still has to compare 1 to everything within the array to ensure it is the minimum. If it's not the minimal value, it is swapped with the minimal value. Therefore, with that said, It has to be a selection sort because it iterates through the entire list in search of a smaller value than the current value. The current pointer starts in the left and it iterates through the entire list and compares the integer to see if it is smaller.

Based on the table, Alg 3 has to be insertion sort, because although there is nothing in the list is unsorted, each element still has to be compared to its left-hand neighbors. Since nothing in the list is out of place, no element has to move to a new index making the moves 0 and comparisons 6.

In Alg 4, I theorized it is Merge sort because the index is broken down into individual arrays and gets compared to each other again. Even though the array is not unsorted, not too many comparisons have to be made, making the number of moves significantly greater than the number of comparisons.

Using the process of elimination, Alg 2 has to be Heap sort.

Trial 2:

In this trial, I tested out every single algorithm with the same integers as trial 1, but in a different order. I chose this in order to determine orthogonality within every single sort that is being used and compare the relationship as well as the differences between trial 1 and trial 2.

Set that is being used: { 2 1 4 3 6 5 }

	ms	mv	comp
alg1	0	14	28
alg2	0	39	14
alg3	0	9	7
alg4	0	32	10
alg5	0	11	12
alg6	0	15	15

Like trial 1, alg6 moves and comparisons are completely equal, which means as it continues to iterates, the moves and comparisons increase. Since I am using the same set of integers as trial 1 but unsorted; then it iterates no matter what and makes the same amount of comparisons and moves just like in trial 1.

With the same set of integers, algorithm 4 should be merged search, because the amount of moves is the same as trial 1, but the number of comparisons differs by 1.

Alg3 has the least amount of comparisons and the least amount of moves. In insertion, the amount of comparisons is determined by how unorganized the entire list is. Each move within the list is about 3 moves in order to make the swap, which is why it has to be insertion

As an example:

```
Int a = 2, b = 1, temp;
If ( a > b)
{
        Temp = a;
        A = b;
        B = temp;
}
```

Alg 5 should be bubble sort because the number of comparisons it has to make until it reaches the first element is 12. In bubble sort, it continues to compare every integer until it reaches the first element whether or not the list is sorted. Since the list was initially unsorted, bubble sort continues to iterate and compare from the beginning of the list and sort the list which is not too reliable.

Trial 3 - Testing out a set of random integers

Unlike my first 2 trials where I tested the same set of integers sorted and/or unsorted, in this trial; I increased the size of the array, and shuffled every single integer in order to see the relationship between every single algorithm. In this trial, it is a little bit different, because the amount of moves and comparisons is what separates the ‘smart’ versus the ‘dumb’ sorting algorithms.

{ 100, 90, 4, 5, 60, 75, 3, 2, 0, 55, 45, 67, 43, 6, 7, 8, 9, 10}

	Ms	mv	comp
alg1	0	68	133
alg2	0	171	91
alg3	0	267	101
alg4	0	152	47
alg5	0	276	162
alg6	0	51	153

According to my data, if we increase the size of the set, then the amount of comparisons and moves among every mysterious algorithm differs in the number of comparisons and/or moves. However, since this list is unsorted, it still supports my theory from alg1 (quicksort) because the amount of moves and comparisons is generally smaller between the other 5 algs. Quicksort tends to place “out of place” integers in order with the least amount of moves for comparison that the other sorts.

Alg5 should be bubble sort because the number of comparisons and moves are significantly higher than the other comparisons. Whether the integers are in order or unsorted, it is still being compared. Just like insertion, it takes 3 moves just to make 1 swap. Alg3 should be inserted because the entire array shifts if there is an out-of-place integer.

Alg 6 should continue to be a selection sort because the amount of comparison is illustrated, because as it continues to iterate.

Original test list

		Ms	mv	comp
test.txt	alg1	6	8130	14765
test.txt	alg2	12	27222	16814
test.txt	alg3	160	719634	240871
test.txt	alg4	18	19952	8732
test.txt	alg5	338	720555	921000
test.txt	alg6	123	2997	499500

Even though time complexity is not as important when being compared to the moves and comparisons category, in this trial; it indicates which sorting algorithm is used. Quicksort tends to be the fastest sorting algorithm, where it is represented in alg1, while bubble sort is the slowest (of the 5 other) sorting algorithms (which is indicated in alg5).

Even though the amount of moves is not significantly high on alg2 and alg4, it does indicate that they are ‘smart’ algorithms. According to the data, Alg2 looks like it should be heap sort, because of the process of reheapification and pushing the values onto the heap tends generally to be higher in terms of the moves. Thus making comparisons a bit higher as well. In Alg4, every integer within the array is being broken down into individual arrays and gets compared again in order to make the sorted group of integers.

Just like trail one, two, and 3; alg 3 should be inserted, and alg6 is selection sort.

What separates this list from the other 3 trials I have tested is how it makes it obvious which algorithm is a ‘smart’ algorithms and which once are ‘dumb’ algorithms. The mysterious

algorithms that have a significantly large time complexity, a larger amount of moves, and significantly large comparisons tend to be the ‘dumb’ algorithms. This set supports the 3 trials that I have tested because alg3 makes the most moves and smaller comparison in an unsorted list just like the insertion sort. Whereas alg4 is merge sort where it is a ‘smart’ sorting algorithm that has greater moves than comparisons.

Conclusion:

The main purpose of this lab is to use our theoretical knowledge of sorting algorithms in order to inspect and figure out which of the 6 sorting algorithm goes with the mysterious algorithms. Based upon the data, I conclude that Algorithm 1 is quick, Algorithm 2 is heap, Algorithm 3 is Insertion, Algorithm 4 is Merge, Algorithm 5 is Bubble, and Algorithm 6 is Selection Sort. Although I used the same set of integers in my first two trials, the two sorted/ unsorted sets distinguished which of the sorting algorithms is a ‘smart’ and ‘dumb’ algorithms. The general intention I have captured is which of the sorting algorithms can possibly be dumb or smart.

Even though time complexity did not matter in the first three trials (considering it is zero), it is more significant in the original test file where it is different among every algorithm that is beginning tested. Quicksort happens to be the fastest sorting algorithm while bubble sorts are the slowest sorting algorithm. Even though every single sorting algorithm ends up being sorted; in the end, the type of sort we use tends to be more reliable in time complexity. In a bubble, selection, and insertion sort; it only iterates through the entire array once while in quick, merge, and heap sort goes through the arrays multiple. This is important to know because the among of comparisons that is used in the ‘smart’ algorithm is not significantly higher when being compared to ‘dumb’ algorithms.