**Part 1:**

**Algorithm Analysis: Theoretical vs Experimental**

**Team Members:** Carlo Quitorio, Jacob Whitlow**,** Juan Mendoza, Julieta Mendez, Malee Seechan, Marc Andrews Bantay, Ryan Anderson

In the study of algorithms, we develop mathematical expressions to describe the expected growth rate of runtimes of algorithms as the input size grows. Sorting algorithms provide a great example of when and how we do this, as well as showing how the same result can be achieved by drastically different approaches. Within the scope of this paper, we are looking at these well known sorting algorithms:

1. Insertion Sort
2. Selection Sort
3. Bubble Sort
4. Merge Sort
5. Quick Sort
6. Heap Sort
7. Counting Sort
8. Radix Sort

The sorts labeled 1-6 are known as comparison based sorting algorithms. There is a predetermined “ranking order” of each element to be sorted, and that all of those sorts follow that ordering. For simplicity, we have chosen the set of whole numbers, with an arbitrarily chosen upper bound of 49,999 for randomized inputs.

**Theoretical Question:**

What are the best, worst, and average case inputs, theoretically for each sorting algorithm.

**Insertion Sort:**

Best Case: Insertion Sort works by taking an element and continually swapping it with its partner to its left as long as it violates the predetermined order of the sort. This means, if the input is in order, there are no swaps made at all causing a runtime of n as it still needs to progress through the entire list, therefore best case is **O(n)**

Worst Case: As stated above, insertion sort will continually swap an element with the element to its left if the order is violated. If the input was reversed, every item would violate this order causing n(n-1)/2 comparisons and swaps necessary to complete the sort. This would be **O(n2)**

Average Case: If we assume about half of the items violate the predetermined order, then the worst case being cut in half would be a good indicator of the average: n(n-1)/4. This however still has a leading term of n2 therefore the running time would be **O(n2)**

**Selection Sort:**

Best Case/Worst Case/ Average Case: Selection sort, as the name suggests, selects the “next minimum” element from the remainder of the list, and inserts it in its respective place. To do this, it must iterate to the end of the list from the current start of the list, so regardless of the input, it will always iterate fully to the end to find the minimum element. Therefore, selection sort will always have n(n-1)/2 comparisons needed to sort the list, and any differences in sorting times will be caused by assigning the value of the minimum as iteration through the list progresses. Running time is **O(n2)**

**Bubble Sort:**

Best Case: The inputs are already sorted, meaning that no swapping takes place. Depending on the implementation, best case can be expressed as **O(n)**.

Worst Case: The inputs are in reverse order. This would cause the first element, which would be the largest, to be repeatedly swapped n-1 times, and this would repeat n-1 times, or n(n-1), which can be expressed as **O(n2).**

Average Case: The input would be random, meaning that some elements are already in order, while others are not. We can expect that about half of the items are in order, meaning that our worst case running time can be cut approximately in half, or n(n-1)/2 -> n2/2 which can also be expressed as **O(n2)**.

**Merge Sort:**

Best Case/ Worst Case/ Average Case: Merge Sort is another interesting sorting algorithm. It works by creating smaller and smaller subproblems and utilizes auxiliary arrays to help sort the input. Using this method, the order of the input does not greatly affect the run-time. Merge Sort’s comparison loop will undoubtedly cover all of the inputs, but only as many times as the input can be divided in half due to its divide and conquer strategy for sorting. This leads to a **𝝧(n lg n)** running time.

**Quick Sort:**

Best Case: Quick Sort is tricky since its best case depends on how the partition is chosen inside the “inner loop” of the function. Ideally the partition would be a value perfectly in the middle of the greatest and least values of the sub-list being sorted. This would cause a recurrence relation that looks like: T(n) = 2T(n/2) + O(n) which we know comes out to O(n lg n). As this would be the best case, this would give a lower bound of **𝛀(n lg n).**

Worst Case: The given worst case happens when the RNG gods look down on you, and the partition is chosen as either the greatest, or least, value of the sublist, which causes the d d recurrence relation to be T(n) = T(1) + T(n-1) + O(n), which can be shown to evaluate out to **O(n2).**

Average Case: Depending on the method to calculate the partition, quick sort can be non-input sensitive. This means that for an already ordered list, or a reversed list, the partition will cause a recurrence relation between T(n) = 2T(n/2) + O(n) and T(n) = T(1) + T(n-1) + O(n), and can reasonably be expected to evaluate out to **O(n lg n)**.

**Heap Sort:**

Best Case/Worst Case/Average Case: Heap sort works differently than most other sorting algorithms, as Heap sort starts by pre-sorting the input into a Heap data structure. Without delving too deep into that process, we can say that this occurs in O(n) time. From there, we repeatedly “pop” off the top element of the heap, and re-heapify the remaining elements. This can be expressed as O(1) for the “pop” off, and O(lg n) for the re-heapify function. All together we can express this as: (n \* lg n) {which represents the repeated calls to re-heapify} + n {denotes the amount of “pop”s that occur} + O(n) { from the original call to Build-Heap}: which can be simplified to **O(n lg n)**

**Counting Sort:**

Best Case/Worst Case/Average Case: Counting Sort is non-comparison based, and performs in a manner that is nearly independent of the input. However, when the range of input is smaller, meaning you have a smaller value of n, you can expect slightly better runtimes. Otherwise, Counting sort can be expected to have an amortized runtime of **Θ(n+r)** where n denotes the input size, and r denotes the range of the input.

**Radix Sort:**

Best Case/Worst Case/Average Case: Radix sort is another non-comparison based sorting algorithm. It works by sorting the input by values at a given radix, starting with the least significant element, and repeating the sort until it reaches the most significant element. Radix sort is open for interpretation, however, a popular choice for the inner sorting algorithm is counting sort, which we know to have a O(n + r) run time. Since radix sort will repeat this sort over every significant element we can express the runtime as a multiple of counting sorts runtime:**Θ(d(n+r))** where d represents the greatest amount of significant elements in the input.

**Data generation and experimental setup:**

Machines Used:

CPU | Ram | IDE

1. Ryzen 5 3600 6-Core/12-thread @3.6 Ghz | 16GB DDR4-2133 MHz-CL16 | Dev-C++
2. Ryzen 5 3600 6-core/12-thread @ 4.2 GHz | 16GB DDR4-3600MHz-CL18 | Microsoft Visual Studio
3. Ryzen 5 3600 6-core/12-thread @ 4.2 GHz | 16GB DDR4-3600MHz-CL18 | Vs Code

Timing Mechanism: Seconds, calculated via clock()/CLOCKS\_PER\_SEC equation in C++

Each sorting algorithm was tested with 5 input sizes, each with 3 different orders, resulting in 15 different recorded times per algorithm. To ensure stability, each sort was run and timed 5 times for each type of input, and the average of those times were taken as the recorded time. In addition to this, these tests were run across 3 different machines, with coincidentally near identical specifications, but different IDE’s. This was done to ensure that even though the run-times differ, the growth rates were the same. Recorded times will be added at the end of the document.

Input sizes were chosen according to the expected runtime order of a given algorithm. For example, algorithms on the order of O(n2) were given input sizes between 1,000 and 200,000, while algorithms on the order O(n lg n) were given inputs ranging from 1 million to 200 million. However, due to system limitations and the space complexity of certain algorithms, the inputs greater in size than 10 million varied per algorithm, and per machine. Due to the nature of comparing the experimental run-time to the theoretical runtime, this variance was considered acceptable, so long as we can show that the relationship between the two.

One more thing of note, to help reduce variables when testing the randomized inputs, the same “randomized” input was used across machines. C++’s rand() function produces pseudo-random numbers, and the same set of random numbers could be produced across machines if given the same “seed” value for the related srand() function. Using this concept, the same seed value was chosen for all randomized inputs ensuring that the same “random” list was being tested and timed.

**Experimental Data vs Theoretical Expectations**

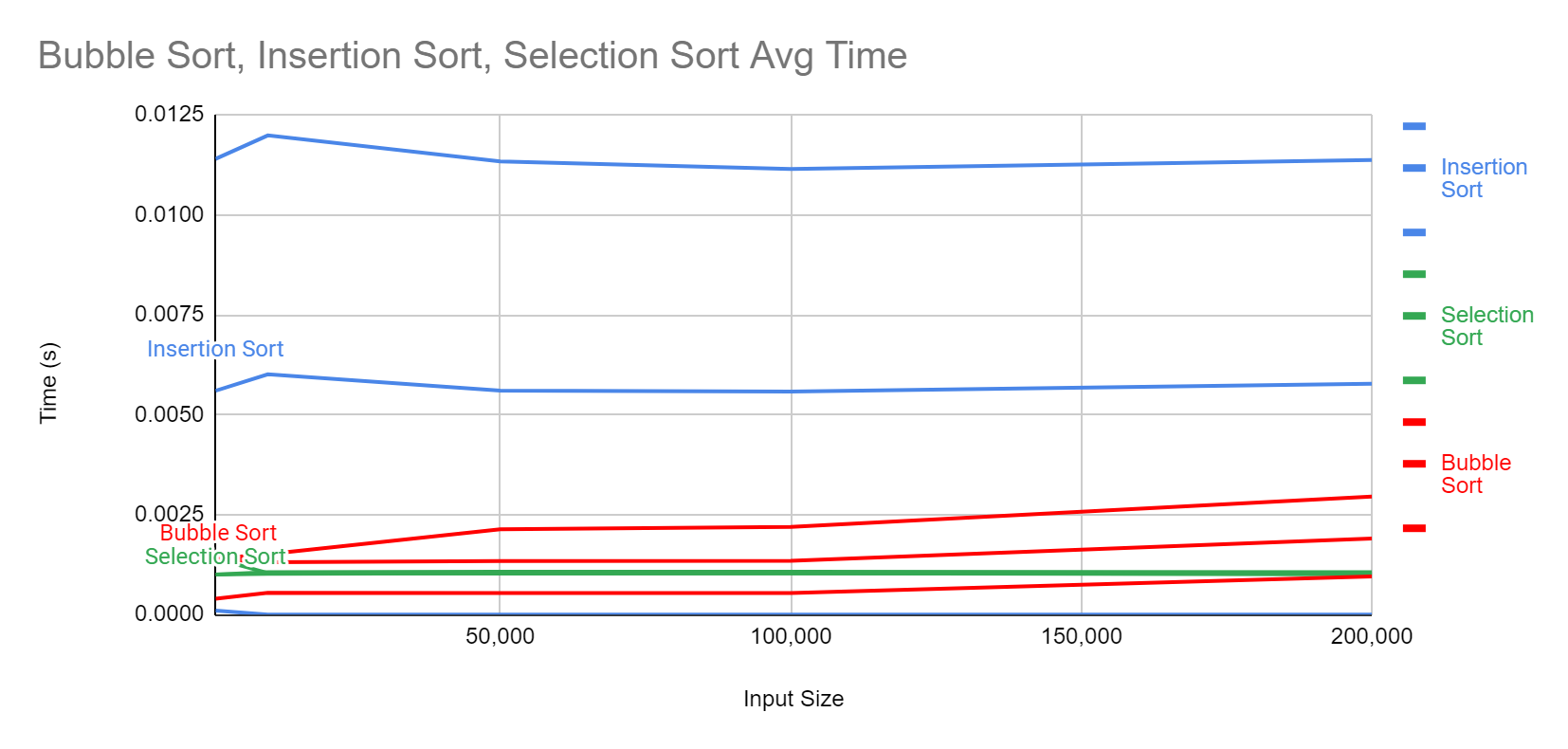
To determine to what extent the Best/Worst/Average cases align with the theoretical calculations for each algorithm, we are attempting to get the “average time per operation” for each sort and inputs. To do this, we simply divide the experimental time taken, by the value given when the input size is plugged into the theoretical formula for the sort. For example, if we had an experimental time of 20 seconds on a 1 million element input for an algorithm that is said to have an **n lg n** runtime, we would calculate:

20.0/(1,000,000 \* lg 1,000,000) = 1.003 \* 10-6 s

If this value is constant, or near constant across all input sizes for that algorithm, then it would be a good indicator that the given growth rate is accurate for that algorithm.

**Bubble Sort, Insertion Sort, Selection Sort,:**

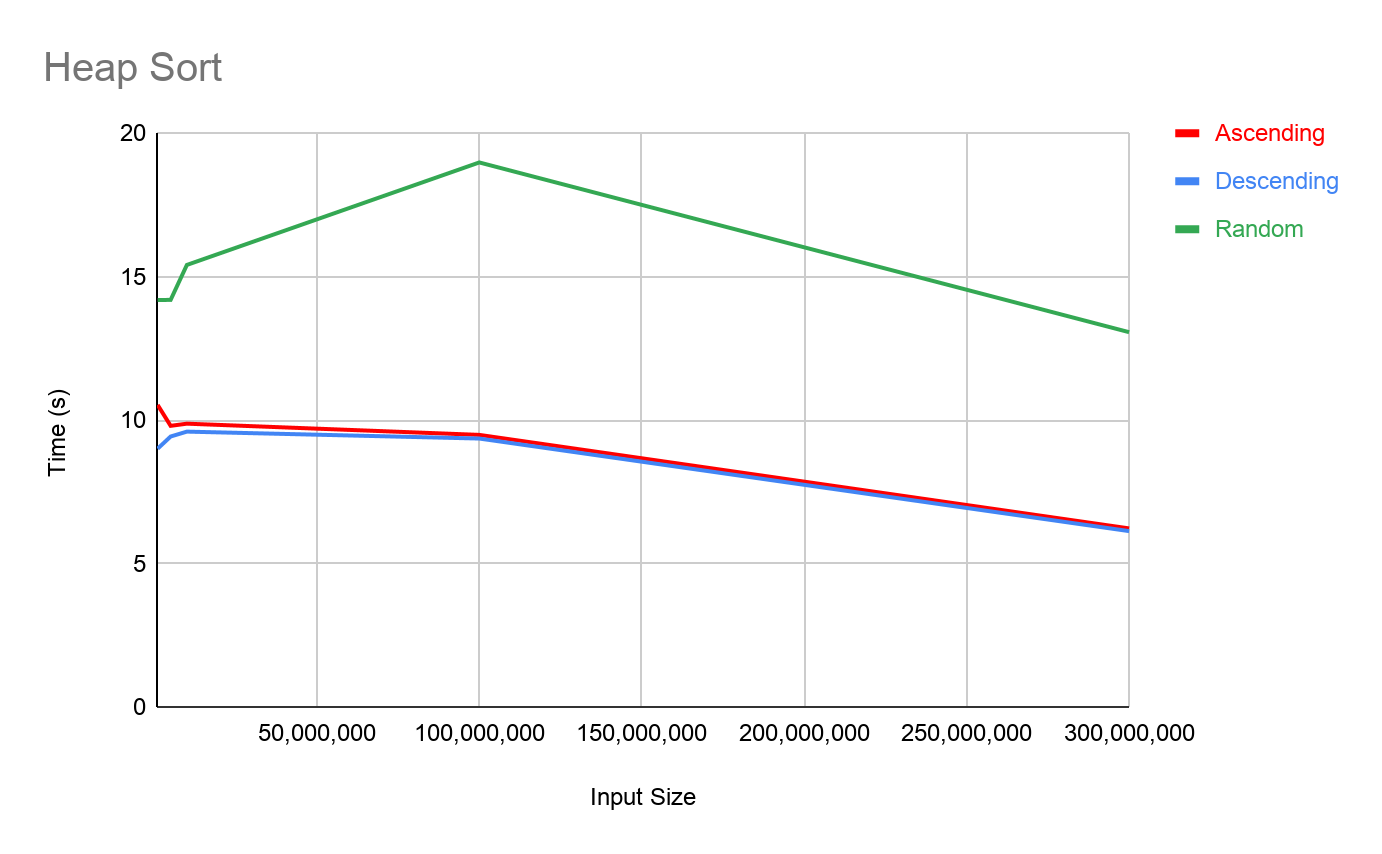
Bubble Sort, Insertion Sort, and Selection Sort all have an expected run time of O(n2).



For this graph, all the y values are calculated by using the above formula. Since for each sorting algorithm the “average time per operation” remains relatively consistent, which is an indicator that O(n2 ) is an accurate descriptor for the running time. For the most part, the experimental results agree with the theoretical calculations.

**Heap Sort:**

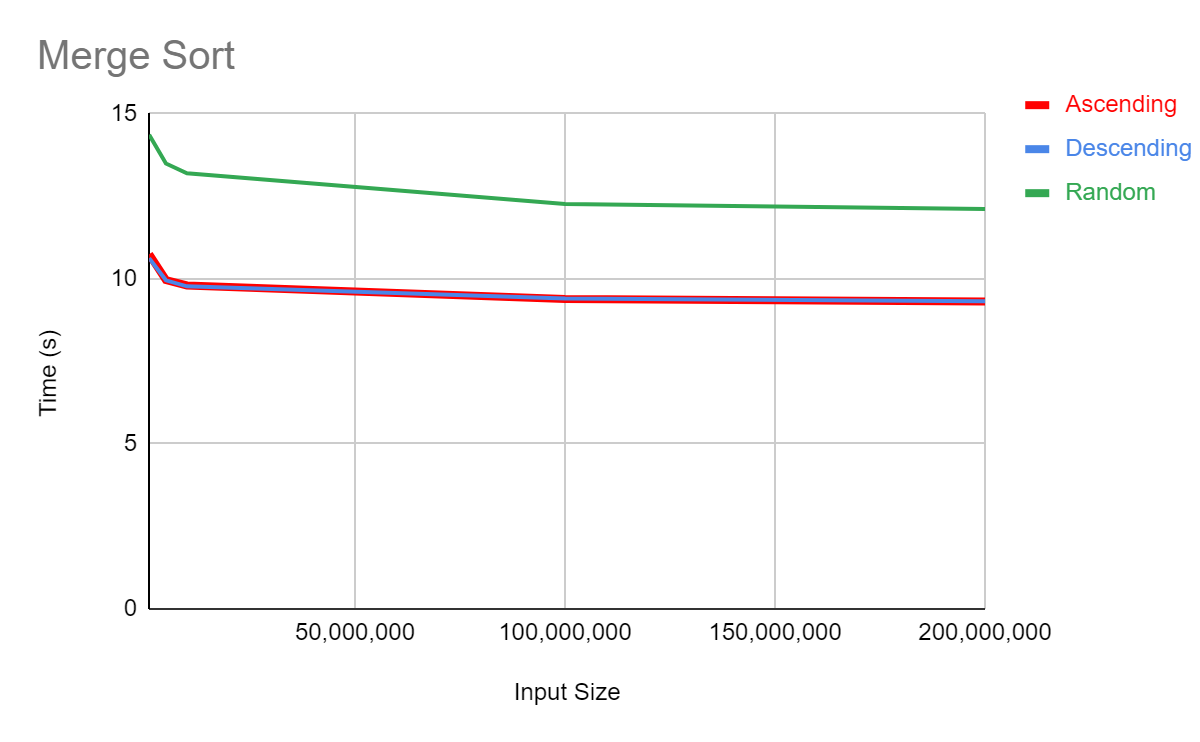
Heap Sort has two main parts to its algorithm: the first being the “Build-Heap” step, and the second “Heapify” is to sort the heap into an ordered list. The “Build-Heap” part of the sort can be shown to run in O(n) time, while the second part of the algorithm is shown to run in O(n lg n) time, which dominates the total run time. Here are the graphed results of the “average time per operation” for Heap Sort:



The red and blue lines represent ascending and descending inputs respectively, while the green line represents the randomized input. It is worth noting that the red and blue lines are nearly identical and have only a slight downward slope. However, looking at the green line, you notice a “hill” like line that has a more pronounced downward slope towards the end. This can be due to system cache that helped speed up the algorithm as the input grew incredibly large. If we take that into account, we can see that it is otherwise nearly horizontally linear with a small slope magnitude. This would indicate that the experimental results, for the most part, are in line with the theoretical runtime state previously.

**Merge Sort:**

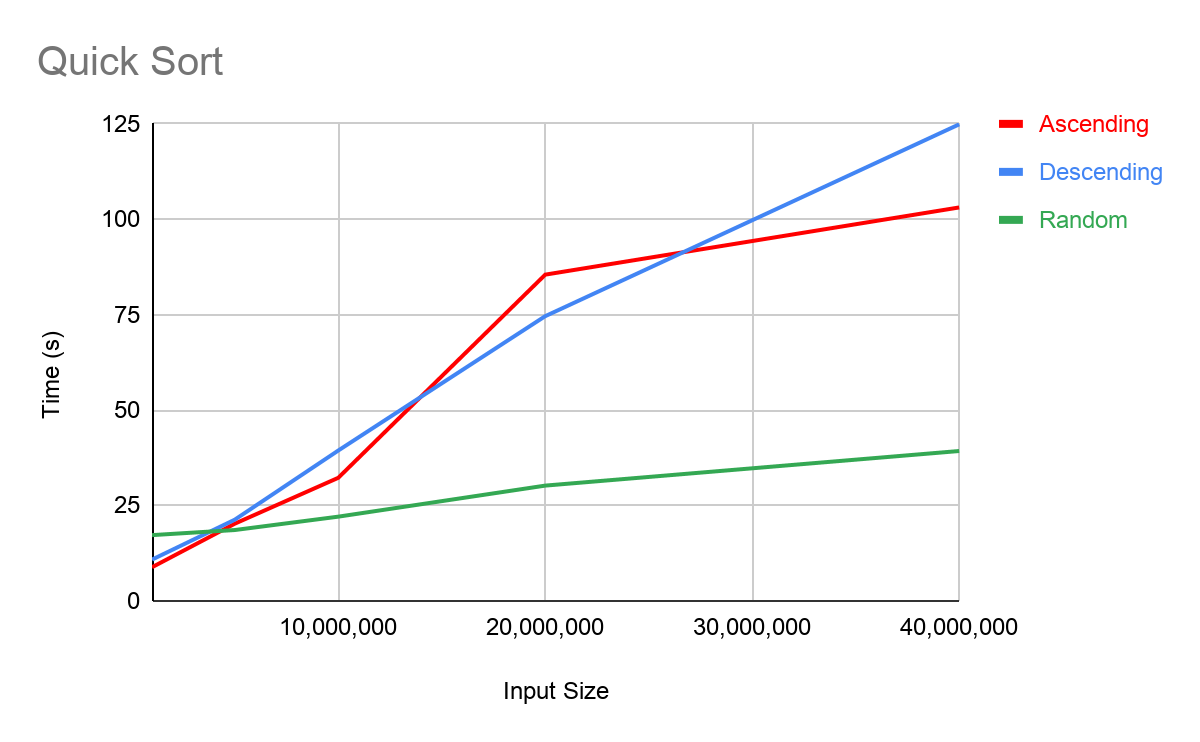
Merge sort is a bit different than most of the sorts we’ve covered so far, in that its running time is guaranteed. At each step it recursively splits the list into two lists of equal, or near equal size, until the base case condition occurs in which the list is of size 1. This is guaranteed to be the same size split every time, unlike quicksort where the sub-list size can differ. Since the split is guaranteed to be half, we are able to express Merge Sort’s running time as **𝝧(n lg n).**

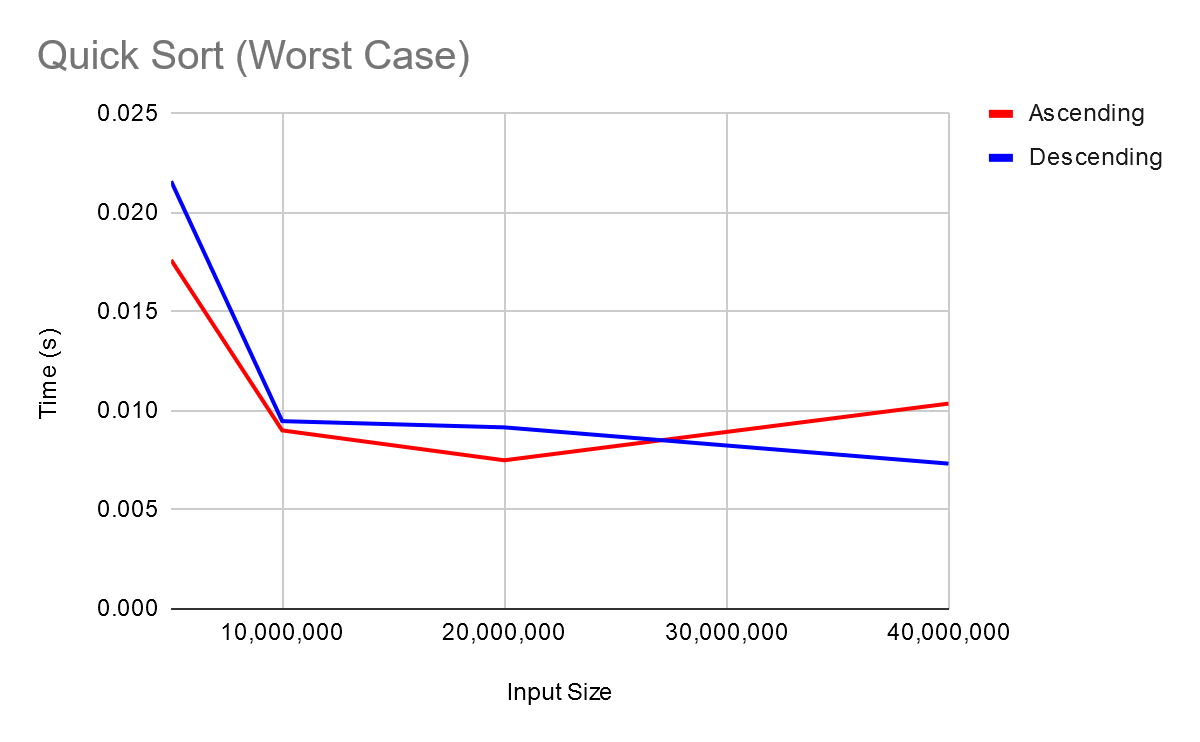


Looking at the plotted data for merge sort, we see a small drop at the beginning, but then it evens out to a nearly horizontal line. As we have said previously, when the line is flat with no slope, then the experimental time matches pretty well with the theoretical calculations.

**Quick Sort:**

Quick Sort is usually the goto algorithm for most cases when a list needs to be sorted. Quick Sort, like Merge Sort, splits a list into two smaller sub-lists. However, unlike Merge Sort, Quick Sort splits at a seemingly random index. Merge Sort consistently splits the list in half while Quick Sort can make a ¾ to ¼ split, a ½ and ½ split, or even a maximally unbalanced split such as 1 and n-1. In the worst case Quick Sorts recurrence relation looks like: T(n) = T(1) + T(n-1) + O(n), which we know comes out to O(n2). Given a good method for choosing the splitting, or pivot point, it can help avoid the worst case. The method implemented uses a “random” partition, which means it will randomly choose a pivot point between both ends of the current list and use that pivot to split the list. This helps avoid a worst case split when dealing with pre-sorted input lists and often does not present a drop in performance for randomized lists. The the data is shown below:



Like before, the red and blue lines represent the Ascending and Descending lists respectively, and the green represents the randomized list. It is apparent that the presorted lists have a growing length of time per operation as the input increases. This indicates that the run time for those two inputs is on a higher order than the estimated n lg n. This data reinforces the fact that the worst case is on the order of n2. 

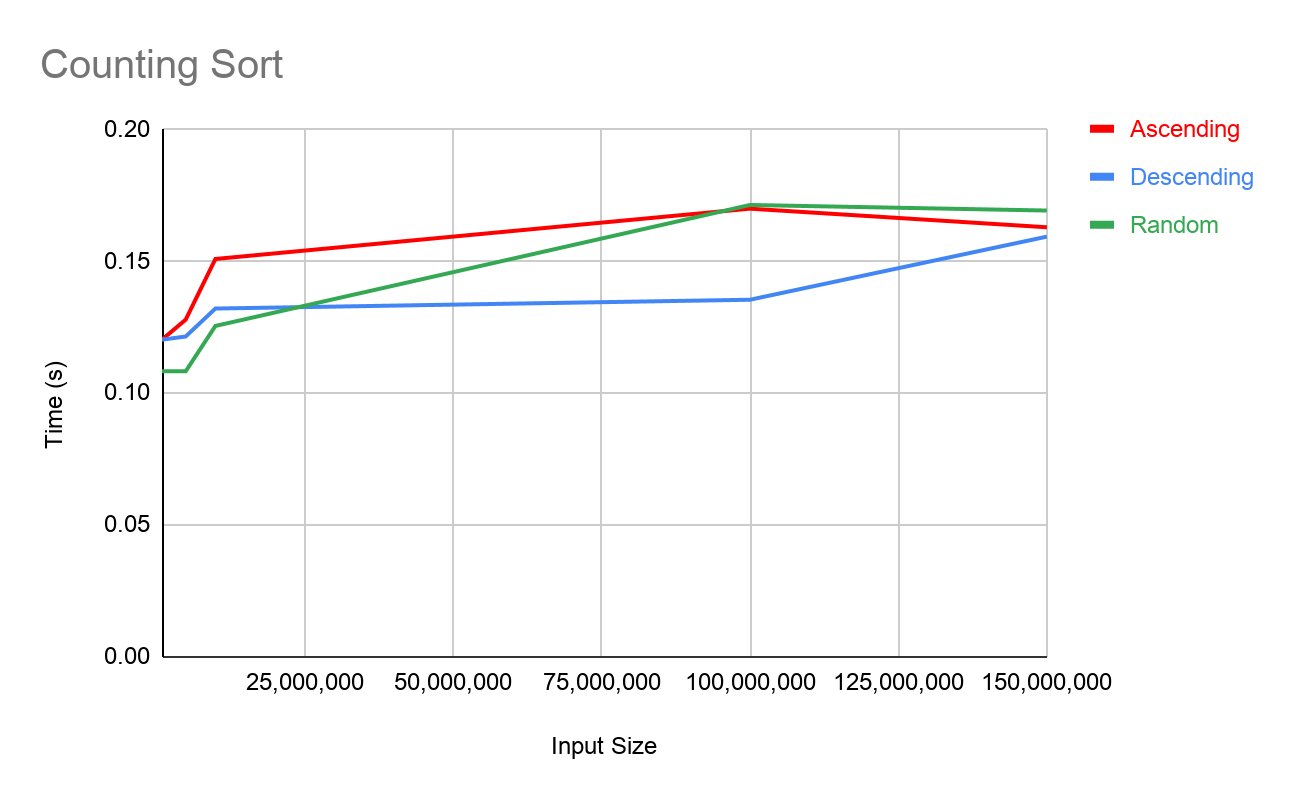
We can see from plotting the data from the ascending and descending lists against n2, and see that the line tends to flatten a lot more, indicating that these two types of inputs cause a worst case scenario. This could be due to the fact that even a randomized choice would cause a less than ideal pivot choice. In the case of the descending input, it could also be that for the first several partitions, all of the elements, minus the pivot, had to be relocated.

However, using a randomized partition should help prevent this, and we can see that as the ascending list times tend to even out after 20 million in the first graph, and the continual decrease in the second graph. Even using a randomized partition, we cannot fully prevent bad splits, only make them less likely, and in the case of the descending list, even with a good split, several elements are relocated.

The data gained experimentally, for the randomized, or “average” input agrees with the above calculations. The data for the pre-ordered lists, doesn’t necessarily disagree with the above statements either, but their outcomes were still a bit unexpected.

**Counting Sort:**

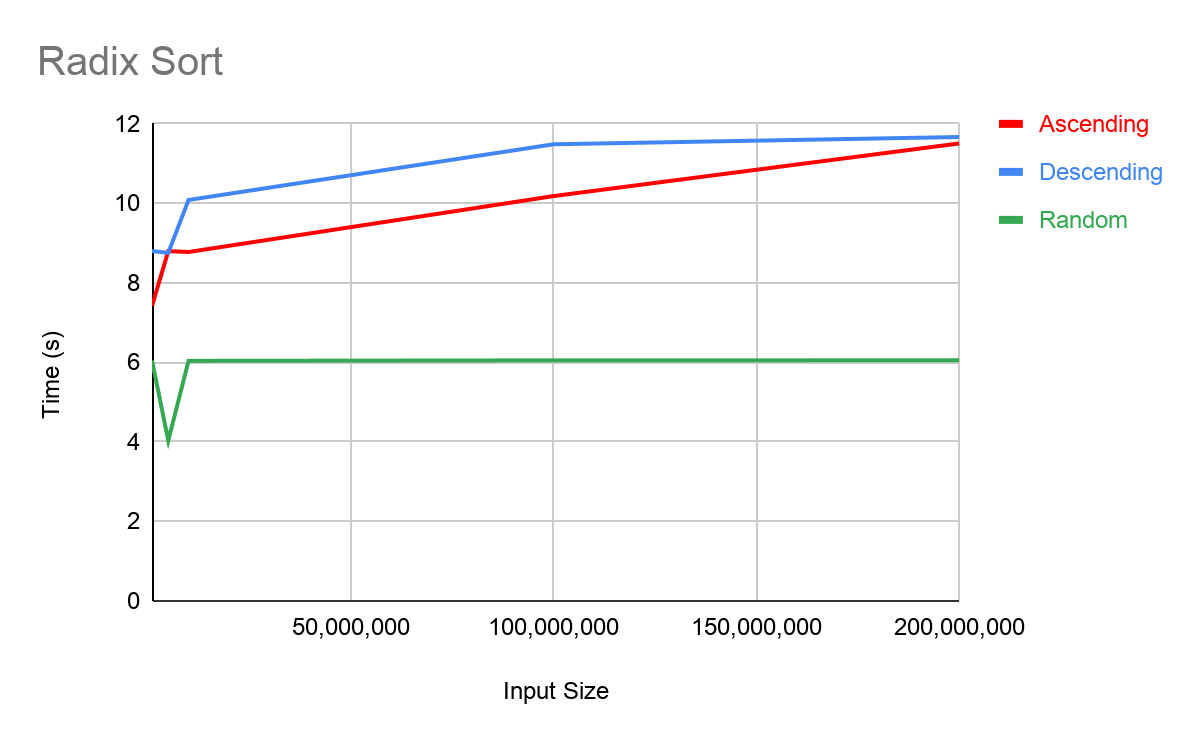
Counting Sort is the first Non-comparison based sorting algorithm on our list. It works by counting the number of occurrences each element has, and uses that value to place the element into its final position. The trick to this algorithm is mapping elemental values to list index numbers, and determining a range for the input. The given runtime for Counting Sort is O(n + r) where n is the input size, and r is the range of the input. The range of the can drastically change the speed of the algorithm, due to how its auxiliary list is used. If you have 5 elements, but one of them is a high extreme value, the auxiliary list becomes much larger than your normal list. To accommodate this issue, and prevent unnecessary running time, the size of the original list was used as a range boundary when getting the runtimes. Here is the “average cost per operation” as a function of input size for counting sort:



We can see some variance as the input size grows, and as stated before, the input size was used as a range boundary. Since that is the case, the auxiliary list used did variate as the input size grew, and this could affect the given runtime of the algorithm. Upon further investigation however, we see that all three input types start to plateau or average out to similar times as the input size reaches its highest point. If we were to increase the input size, it is reasonable to expect this trend to continue and to continue to provide a fairly evened out set of data points. While it is a little naive to assume this behavior is going to continue, this is a good first step into seeing that both the experimental data and theoretical calculations are in agreement.

**Radix Sort:**

Radix Sort utilizes an inner sorting method, in this case Counting Sort, to effectively sort a list. What makes Radix Sort different is that it sorts a list of elements by least significant digits first, then it repeats the sort for the next least significant value of an element. This process repeats until all significant values have been respectively sorted. A key element to this sort is that the inner sorting method must be stable, hence the reason Counting Sort is used. Radix sort also utilizes an auxiliary list, but the range of that list may be substantially reduced, as the range for integers is 0-9 or 10 elements, and the range for letters A-Z is 26 elements. This improves on space complexity, but due to the repeated calls to the inner sorting method, the runtime gets increased by a factor equivalent to the number of significant digits/characters in the largest/longest element. This is expressed as O(d(n+r)), where we recognize the n+r from Counting Sort, and d is the amount of times the sorting method will ultimately get called. Here is the plotted data:



The red and blue lines represent the Ascending and Descending lists respectively. We can see that the green line representing the randomized data is horizontal after a small dip. The red and blue lines are trending to meet at a certain point. Upon closer inspection, we see that the times for the descending list evens out after 100 million elements and remains constant. This implementation of Radix Sort has an additional step, it must find the maximum value first. In an ascending list, this means that every element it sees gets assigned to the maximum value, repeatedly until it reaches the end, which could help explain the constant increase in time. As for the descending list, it assigns this value exactly once.

Taking all this into account, we can reasonably conclude that both the experimental data and the theoretical runtimes are in agreement with each other. One thing of note, Radix Sort ran considerably longer than counting sort for the same inputs. This makes sense as Radix Sort has to repeatedly use Counting Sort, so while the Auxiliary list is smaller and the effective runtime is longer. Non-comparison based sorts, specifically Radix sort, is much more viable for sorting words, as the range of words is much much greater, on average, than the range of numbers to be sorted.

**Part 2:**

The problem presented is to find if there exists a pair of elements in a list that add up to a target value. The first approach is a brute force approach, it checks every pair using a doubly nested for loop.

Here is the pseudo code:

**1 twoSum(arrayA, target)**

**2 for** i ← 1 **to** arrayA.length-1 //iterate through the entire array

**3 for j ←** i + 1 to arrayA.length // starting at i iterate through the array

**4** **if** (arrayA[i] + arrayA[j] == target) //if sum is found

**5** **then return true** // return true

**6 return false** // otherwise end of list is reached

// indicating no sum

Lines 2 and 3 are used to iterate over the list, line 3 starts at the index directly to the right of the index of the first for loop, and repeatedly iterates to the end of the list. Line 4 checks to see if the elements at the two indexes add to the given target sum, and if they are equal, it breaks the loops and returns true on line 5. If the end of the list is reached, then that indicates there are no pairs that add to the sum.

Starting at line 2, it will run n times, and for each iteration of the outer loop, the inner loop iterates n-i times where i is the index of the outer loop. This creates a runtime that can be represented by n(n-1)/2. This, as we know, is in the family of O(n2) functions. This is considered a Brute Force approach that processes every single pair of elements, and is rather slow. It is efficient on memory, but in-efficient on speed.

Below are two other approaches that demonstrate a more efficient and creative non-brute force approach that solves the same problem. These two algorithms are a classic example of a trade off between space and time efficiency.

**Approach # 1: Runtime O(n lg n), requires no extra space**

**//This returns a set of 2 elements whose sum is equal to the target sum, or returns**

**// the empty set if there is no solution**

**1 sumOfTwoNumbers(A, target)**

**2** sort(A) // assumes use of quick Sort O(n lg n) with minimal extra memory

**3 i** ← 1 // beginning index

**4 j ←**size(A) // last index

**5 while** i < j

**6 sum ←** A[i] + A[j]

**7 if** sum < target // if sum is less than target, increase the smaller element

**8** i ← i + 1

**9 else if** sum > target // else if sum is greater, reduce the larger element

**10** j ←j - 1

**11 else** // else the sum is equal, exit the loop maintaining indexes

**12** exit while loop

**13** if A[i] + A[j] == target

**14 return** A[i], A[j]

**15 else**

**16** no solution

This approach starts by sorting the list prior to iterating through the list. This sort, in line 2, is preferably Quick Sort, which runs in O(n lg n) time. The next step for this algorithm is to start iterating through the array starting at each end of the list. The pointer i starts at the lowest end of the list, and j starts at the highest end of the list. While these indexes do not cross, it processes the given sum for the two elements at these indexes. If the sum of the current two elements being pointed at is less than the target sum, then the lower pointer is incremented to the next element. Otherwise if the current sum is greater than the target sum, the pointer to the greater element is decremented to the next smallest value. This process continues until either a match is found, or the pointers cross. Analyzing this while loop, we see that each element will be iterated over one time, until the final element is reached when i and j cross. This is represented by n+1 iterations.

Adding both of these runtimes together we get the expression f(n) = (n lg n) + n + 1. We can see that this approach is dominated by the sorting algorithm used, in this case Quick Sort, and the whole algorithm has a runtime of O(n lg n).

We chose Quick Sort because it utilizes very little excess memory, and is rather efficient.

This next approach runs faster, but at the cost of using more memory.

**Approach # 2: Runtime O(n), requires up to 2n for space complexity**

**1 twoSum (numList, listLength, sum)**

**2 If listLength < 1**

**3 Then return false // if empty list return false**

**4 EndIf**

**5** Initialize a hash table **T** // create a hash table, preferably with chaining

**6 for** i ← 1 **to** listLength //for each element in the list

**7 If** T **does not** contain (sum - numList[i]) //check for its counterpart

**8 then** insert numList[i] **into** T// if no counterpart, insert into hash table

**9 Else return** true // else the counter part was found

**10 EndIf** // meaning two elements add to “sum”

**11 return false** // if end of list is reached, return false

In this approach, we utilize a familiar data structure called a Hash Table. For this approach, we are using two functions from the hash table, an insert function and a find function. Hash tables are known to run most of their operations in O(1) or a constant time. This implementation uses a hash table that uses Chaining as a method of collision resolution, this allows it to be able to grow dynamically as needed.

Line 6 starts at the first index of the list, and iterates through to the end, which can be expressed as n comparisons. Line 7 checks to see if the needed counterpart to the current value has already been seen, this function for a hash table is known to run in O(1) time. If we have seen its counterpart already, we return true, otherwise we push the current value into the hash table in line 8. This insertion function is known to run in O(1) time. As we progress through the list, and we push more elements into the hash table, the more space we use. This space can run up to another n elements if we iterate through the entire list.

Adding all these runtimes together we get the expression f(n) = n + (n\*1). This expression is on the order of O(n) which is faster than the O(n lg n) approach above, however to achieve this run-time, we use twice the amount of memory. This second approach can also be described as a type of Dynamic Programming approach to the problem.