We introduce the Quick Sort algorithm. The Quick Sort algorithm is a divide-and-conquer strategy to sort an array efficiently while making the least number of comparisons between array elements. Our results show that for arrays with large numbers of array elements, Quick Sort is more efficient than three other comparison sort algorithms, Bubble Sort, Insertion Sort, and Selection Sort. Our theoretical evaluation shows that Quick Sort beats a quadratic time complexity, while our empirical evaluation shows that Quick Sort at times was 32 times faster than Insertion Sort, the current recognized most efficient comparison algorithm, with one real data set.

**Keywords**
Quick Sort, sorting, comparisons, Selection Sort, time complexity

1. **INTRODUCTION**

The ability to arrange an array of elements into a defined order is very important in Computer Science. Sorting is heavily used with online stores, were the order that services or items were purchased determines what orders can be filled and who receives their order first. Sorting is also essential for the database management systems used by banks and financial systems, such as the New York Stock Exchange, to track and rank the billions of transactions that go on in one day. There are many algorithms which provide a solution to sorting arrays, including algorithms such as Bubble Sort[1], Insertion Sort[2], and Selection Sort[3]. While these algorithms are programmatically correct, they are not efficient for arrays with a large number of elements and exhibit quadratic time complexity.

We are given an array of comparable values. We need to arrange these values into either an ascending or descending order.

We introduce the Quick Sort algorithm. The Quick Sort algorithm is a divide-and-conquer algorithm. It takes an array and divides that array into sub arrays based on a pivot value. Values less than the pivot value are shifted to within the array to the left of the pivot while values greater are shifted to the right of the pivot in a process called partitioning. These elements are then divided into two sub arrays, one of the elements to the left of the pivot and the other the right elements. The partitioning process is done recursively until we have arrays of single elements. A single element is already sorted, and so the elements are combined back into sorted arrays two sub-arrays at a time, until we are left with a final sorted array. We contribute the following:

1. We introduce the Quick Sort algorithm.

2. We show that theoretically Quick Sort performs no worse than$O(n^2)$ in the worst case, being equivalent to current solutions.

3. We show that empirically Quick Sort is on average faster than Insertion Sort.

This paper will discuss in Section 2 comparison sort algorithms related to the problem, followed by the detailed approach of our solution in Section 3, the evaluation of our results in Section 4, and our final conclusion in Section 5.

2. **RELATED WORK**

The three algorithms that we will discuss are Bubble Sort, Selection Sort, and Insertion Sort. All three are comparison sort algorithms, just as Quick Sort.

The Bubble Sort algorithm works by continually swapping adjacent array elements if they are out of order until the array is in sorted order. Every iteration through the array places at least one element at its correct position. Although algorithmically correct, Bubble Sort is inefficient for use with arrays with a large number of array elements and has a$O(n^2)$ worst case time complexity. Knuth observed[1], also, that while Bubble Sort shares the worst-case time complexity with other prevalent sorting algorithms, compared to them it makes far more element swaps, resulting in poor interaction with modern CPU hardware. We intend to show that Quick Sort needs to make on average fewer element swaps than Bubble Sort.

The Selection Sort algorithm arranges array elements in order by first finding the minimum value in the array and swapping it with the array element that is in its correct position depending on how the array is being arranged. The process is then repeated with the second smallest value until the array is sorted. This creates two distinctive regions within the array, the half that is sorted and the half that has not been sorted. Selection Sort shows an improvement over Bubble Sort by not comparing all the elements in its unsorted half until it is time for that element to be placed into its sorted position. This makes Selection Sort less affected by the input’s order. Though, it is still no less inefficient with arrays with a large number of array elements. Also, even with the improvements Selection Sort still shares the same worst-case time complexity of$O(n^2)$. We intend to show that Quick Sort will operate on average faster Selection Sort.

The Insertion Sort algorithm takes elements from the input array and places those elements in their correct place into a new array, shifting existing array elements as needed. Insertion Sort improves over Selection Sort by only making as many comparisons as it needs to determine the correct position of the current element, while Selection Sort makes comparisons
against each element in the unsorted part of the array. In the average case, Insertion Sort's time complexity is $O(n^2)$, but its worst case is $O(n^2)$, the same as Bubble Sort and Selection Sort. The tradeoff of Insertion Sort is that on the average more elements are swapped as array elements are shifted within the array with the addition of new elements. Even with its limitations, Selection Sort is the current fastest comparison based sorting algorithm since it equals Bubble Sort and Selection Sort in the worst case, but is exceptionally better in the average case. We intend to show that Quick Sort operates on average faster than Insertion Sort.

3. APPROACH
A large array with an arbitrary order needs to be arranged in an ascending or descending order, either lexicographically or numerically. Quick sort can solve this problem by using two key ideas.

The first key idea of Quick sort is that a problem can be divided and conquered. The problem can be broken into smaller arrays, and those arrays can be solved. This is done by picking an element from the array, called the pivot, and reordering the array so that all elements which are less than the pivot come before the pivot and all elements greater than the pivot come after it. This process is called partitioning. Second, by dividing the array into two halves, elements less than the pivot and elements greater than the pivot, then partitioning those halves recursively into arrays of single elements, two sorted arrays are concatenated on either side of the pivot value into one array, as a single element array is already sorted. Refer to the following pseudocode:

```plaintext
Input - A: array of n elements, left: left sub array from the pivot, right: right sub array from the pivot
Output - array A sorted in ascending order
1. proc quicksort(A,left,right)
2. if right > left
3. pivot = random(A)
4. newPivot=partition(A,left,right,pivot)
5. quicksort(A, left, newPivot - 1)
6. quicksort(A,newPivot+1,right)
7. return A
```

Figure 1: Pseudocode of the Quick Sort algorithm

As the pseudocode shows, after a new pivot element is randomly chosen (lines 3-4), the array is broken up into a left half and a right half and sorted recursively (lines 5-6). In the partition algorithm, the two sub arrays are compared to determine how they should be arranged (lines 12-15). In the following examples, using the given input, the partitioning of the array (Figure 3) and how the sub arrays are concatenated back into a sorted array (Figure 4) are illustrated.

```
Inputs - A: array of n elements
(33 55 77 11 66 88 22 44)
Output - array A in ascending order
```

![Figure 3: Shows the partitioning of the input array into single element arrays. The red element is the pivot element of the array.](image)

![Figure 4: From the bottom up, shows the concatenating of the single element arrays. The red element is the pivot element.](image)

As the example shows, array A is broken in half based on the pivot until they are in arrays of only a single element, then those single elements are concatenated together until they form a single sorted array in ascending order.

4. EVALUATION
4.1 Theoretical Analysis
4.1.1 Evaluation Criteria
All comparison based sorting algorithms count the comparisons of array elements as one of their key operations. The Quick Sort algorithm can be evaluated by measuring the number of comparisons between array elements. As the key operation, we can measure the number of comparisons made to determine the overall efficiency of the algorithm. We intend to show that because the Quick Sort algorithm makes less comparisons over the currently acknowledged most efficient algorithm, Insertion Sort[Sec 2], Quick Sort is the most efficient comparison sort algorithm.
4.1.2 Quick Sort Case Scenarios

4.1.2.1 Worst Case

Quick Sort makes the element comparisons we want to measure during the partition step, where pairs of arrays are recursively ordered. Quick Sort’s worst case, depicted in Figure 5, is the scenario where each partition creates one sub array of n-1 elements and one sub array with zero elements. This is caused either by the smallest or largest element in the original array being chosen as the pivot. This forces the maximum number of comparisons to occur. In this case, the Quick Sort algorithm’s efficiency can be represented by the number of comparisons made during each partitioning, which is described in the equation below:

$$T(n) = T(k) + T(n-k) + n$$

Equation (1) gives the total number of comparisons that occur in Quick Sort dependent on the number of array elements. The T(k) refers to the comparisons made to the left sub array from the pivot. The T(n-k) refers to the total comparisons on the right sub array. Let us assume that the pivot was chosen to be the smallest element of the array. We get Equation (2). With this equation, we can analyze the time complexity by solving the recurrence. We use substitution to substitute n-1 instead of n and continue with substitution to get Equations (3) – (5).

$$T(n) = T(1) + T(n-1) + n$$  \hspace{1cm} (2)
$$T(n) = T(n-2) + 2T(1) + (n-1 + n)$$  \hspace{1cm} (3)
$$T(n) = T(n-3) + 3T(1) + (n-2 + n-1 + n)$$  \hspace{1cm} (4)
$$T(n) = T(n-4) + 4T(1) + (n-3 + n-2 + n-1 + n)$$  \hspace{1cm} (5)

It is at this point, that we can determine a pattern in the recurrence and get the following generalized equation:

$$T(n) = T(n-i) + iT(1) + (n-i+1 + \cdots + n-1 + n)$$  \hspace{1cm} (6)

Equation (6) can be further decomposed into the following summation:

$$T(n) = T(n-i) + iT(1) + \sum_{j=0}^{i-1}(n-j)$$  \hspace{1cm} (7)

It is easy to see that the recurrence from Equation 8 can only go on until i=n-1 since otherwise n-i would be less than 1. We substitute i=n-1 to get Equation (8).

$$T(n) = T(1) + (n-1)T(1) + \sum_{j=0}^{n-2}(n-j)$$  \hspace{1cm} (8)

We can further simply Equation (8) to see that for Quick Sort in the worst case its time complexity is $O(n^2)$.

4.1.2.2 Best Case

The best case of Quick Sort, depicted in Figure 5, occurs when the each partition steps creates sub arrays of equal size. In this scenario, only $\frac{n^2}{2}$ comparisons of array elements are made. Using the same process that we used to determine the total number of comparisons for the worst case, we get the following recurrence:

$$T(n) = 2T\left(\frac{n}{2}\right) + n$$  \hspace{1cm} (9)

Solving this recurrence results in a Big O time complexity of $O(n \log n)$.

4.1.3 Analysis

Comparing the time complexities of Quick Sort and Insertion Sort, Insertion Sort equals Quick Sort in the worst case. However, taking the best cases, Quick Sort is faster than Insertion Sort with a time complexity of still $O(n \log n)$ over Insertion Sort’s, which is equivalent to $O(n)$. This adds to the theory that Quick Sort will overall be a better algorithm over Insertion Sort with less than optimum input.

Figure 5: Image (a) illustrates the partitioning of the sub arrays in the best case of Quick Sort, while Image (b) shows the worst case.

4.2 Empirical Analysis

4.2.1 Evaluation Criteria

The goal of this evaluation is to demonstrate the improved efficiency of the Quick Sort algorithm based on the execution’s CPU runtime. We also intend to show that the CPU runtimes for each run are significantly different between Quick Sort and Insertion Sort within a 95% confidence interval.

4.2.2 Evaluation Procedures

The efficiency of Quick Sort can be measured by determining the CPU runtime of an implementation of the algorithm to sort a number of elements versus the CPU runtime it takes an implementation of the Insertion Sort algorithm. The dataset used is a server log of connecting IP Addresses over the course of one day. All the IP Addresses are first extracted from the log, in a separate process, and placed within a file. Subsets of the IP Addresses are then sorted using both the Quick Sort algorithm and the Insertion Sort algorithm with the following array sizes: 5, 10, 15, 20, 30, 50, 100, 5000, 10000, 15000, and 20000. Each array size was run ten times and the average of the CPU
runtimes was taken. Both algorithms take as its parameter an array of IP Addresses. For reproducibility, the dataset used for the evaluation can be found at "http://cs.fit.edu/~pkc/pub/classes/writing/httpdJan24.log.zip". The original dataset has over thirty thousand records. For the purposes of this experiment, only twenty thousand were used. The tests were run on a PC running Microsoft Windows XP with the following specifications: Intel Core 2 Duo CPU E8400 at 3.00 GHz with 3 GB of RAM. The algorithms were implemented in Java using the Java 6 API.

4.2.3 Results and Analysis

Compared to the Insertion Sort algorithm, the Quick Sort algorithm shows marginally faster CPU runtimes for array sizes over 10, increasing drastically for arrays over 100. A summary of the results is contained in Figure 6.

![Figure 6: Depicts the slower speed of Insertion Sort vs. Quick Sort.](image)

With Quick Sort being red and Insertion Sort being blue, Figure 6 shows the relative execution speed curves of the two algorithms. It can be seen that for array sizes under 10, the Insertion Sort algorithm has much faster execution times, showing a clear advantage. This is probably due to the overhead incurred during the shifting of elements during partitioning. Around array sizes of 10, the algorithm’s curves intersect, and Quick Sort begins to show a very small advantage over Insertion Sort. As the size of the array increases pass 100, Insertion Sort’s execution runtime increases at a faster pace than Quick Sort. This shows that Insertion Sort will progressively get worse than Quick Sort as you increase the array size, while Quick Sort’s efficiency will decrease at a slower rate. It can be concluded from the results that for array sizes over 100 Insertion Sort will be unsuitable compared to the efficiency presented when using the Quick Sort algorithm. We performed a two tail t test to test the significance of the differences in CPU runtime. As can be seen in Table 1, we found a significant difference in the CPU runtimes between array sizes when the array sizes were 5 or less ($t=12.91$, d.f.=9, p<0.05) and when the array sizes were 15 or more ($t=17.93$, d.f.=9, p<0.05).

5. CONCLUSION

The purpose of this paper was to introduce the Quick Sort algorithm and show its improvements over its predecessors. Our theoretical analysis shows that compared to the Bubble Sort, Insertion Sort, and Selection Sort algorithms, Quick Sort has a faster Big O worst-case time complexity of $O(n \log n)$. Our empirical analysis shows that compared to Insertion Sort, Quick Sort is 32 times faster for arrays larger than 100 elements. This makes Quick Sort far more efficient than Insertion Sort for array sizes larger than 100. For array sizes less than and greater than 100, the Quick Sort and Insertion Sort algorithms’ CPU runtimes were found to be statistically different within a 95% confidence interval.

Although Quick Sort has been shown to be better in the worst case for array sizes larger than 100, it was still slower than Insertion Sort with array sizes less than 100. This can be explained by the overhead required for the shifting of elements during the partition step. A future improvement to the Quick Sort algorithm would be to determine a way to reduce this overhead.

6. REFERENCES


<table>
<thead>
<tr>
<th>Array size</th>
<th>P-value</th>
<th>T-value</th>
</tr>
</thead>
<tbody>
<tr>
<td>5</td>
<td>4.13E-07</td>
<td>12.907001</td>
</tr>
<tr>
<td>10</td>
<td>0.4656399</td>
<td>-0.761858</td>
</tr>
<tr>
<td>15</td>
<td>5.141E-12</td>
<td>-17.9256</td>
</tr>
<tr>
<td>20</td>
<td>8.597E-18</td>
<td>-47.66376</td>
</tr>
<tr>
<td>30</td>
<td>4.756E-21</td>
<td>-58.5233</td>
</tr>
<tr>
<td>50</td>
<td>6.998E-16</td>
<td>-46.82916</td>
</tr>
<tr>
<td>100</td>
<td>2.015E-09</td>
<td>-23.71021</td>
</tr>
<tr>
<td>1000</td>
<td>7.626E-18</td>
<td>-66.38865</td>
</tr>
<tr>
<td>5000</td>
<td>7.503E-20</td>
<td>-344.217</td>
</tr>
<tr>
<td>10000</td>
<td>8.277E-18</td>
<td>-204.1008</td>
</tr>
<tr>
<td>15000</td>
<td>2.593E-21</td>
<td>-500.2812</td>
</tr>
<tr>
<td>20000</td>
<td>7.378E-24</td>
<td>-959.619</td>
</tr>
</tbody>
</table>