An Analysis of My Sorting Algorithms

Preface: Sorting is one of the most fundamental parts of Computer Science. There are a myriad of algorithms that will only work if the data is sorted, meaning that it is important to find efficient methods of sorting data. Over winter break, I implemented seven of the most popular sorting algorithms, and put them to the test. In this short paper, I will discuss what I found.

Setup: In order to test a sorting algorithm, I first needed data to sort. In C++, an easy way to generate unsorted data can be done using the `rand()` function. This is how I generated my unsorted data:

```
    //Randomize seed
    srand(time(0));
    // Initialize 5 (SIZE) unsorted vectors of unsigned ints, each of which
       containing 10^n unsigned ints (n >= 1 && n <= 5)
    std::vector<std::vector<unsigned int>> v;
    v.reserve(SIZE);
    for (int n = 10; n <= 100000; n*=10) {
        std::vector<unsigned int> vect;
        vect.reserve(n);
        for (int i = 0; i < n; i++) {
            vect.push_back(rand());
        }
        v.push_back(vect);
    }
```

On my machine, `rand()` returns a number between `0` and `32767`, which is an important thing to note. Also, I did not generate new data for each sort, I used the same data on each sort. Finally, I ran each of the seven sorts on each list **one hundred times** (this program took about 11 hours to run!), and took the average [$(t_1 + t_2 + ... + t_{100})/100$] runtime of each sort on each list. To record the runtime of each sort, I used the `chrono` library:

```
template<class item_t>
float timeOfSort(std::vector<item_t> v, const std::string& s)
{
```

```cpp
    std::chrono::time_point<std::chrono::system_clock> start =
std::chrono::system_clock::now();
    switch (hash(s)) {
        case bubble:
            mysorts::bubbleSort(v);
            break;
        case shortbubble:
            mysorts::shortBubbleSort(v);
            break;
        case selection:
            mysorts::selectionSort(v);
            break;
        case insertion:
            mysorts::insertionSort(v);
            break;
        case merge:
            mysorts::mergeSort(v, 0, v.size()-1);
            break;
        case quick:
            mysorts::quickSort(v, 0, v.size()-1);
            break;
        case radix:
            mysorts::radixSort(v, 5, 10);
            break;
    }
    std::chrono::time_point<std::chrono::system_clock> end =
std::chrono::system_clock::now();
    std::chrono::duration<float> elapsed = end-start;
    return elapsed.count();
}
```

O(n^2) Sorts:

1. **Bubble Sort:**

```
Set current to index of first element

while more elements in unsorted section of list
```

```
    “Bubble up” smallest element to the top of unsorted section
    by swapping values as necessary
    Increment current
```

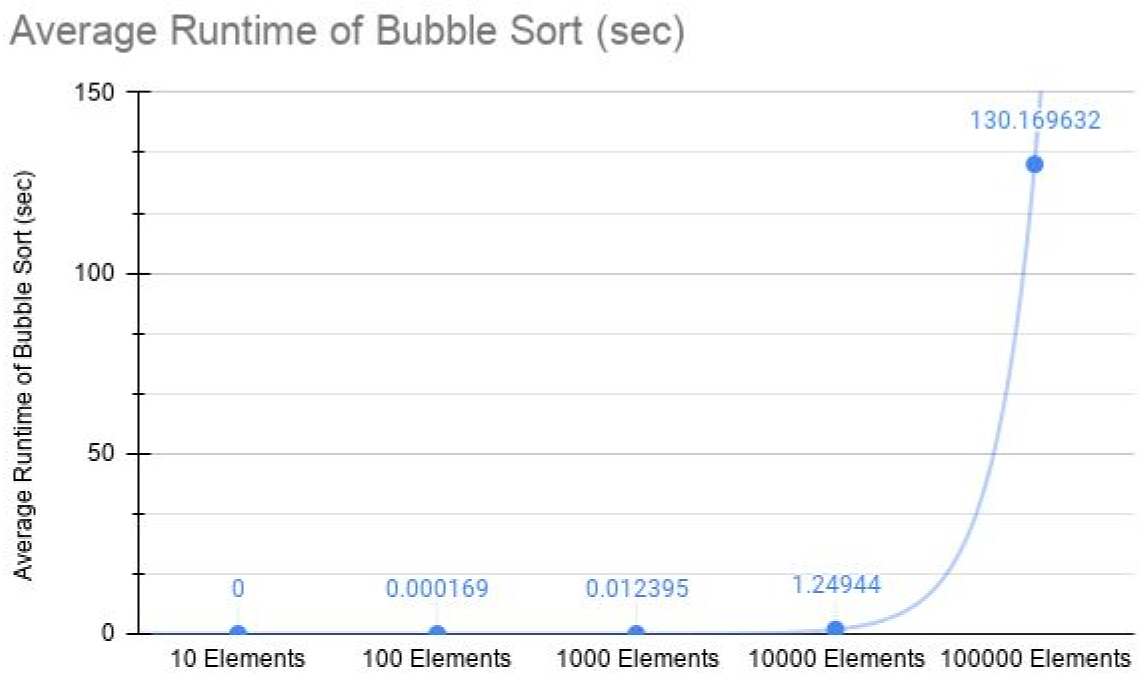
My Implementation in C++:

```cpp
    //Bubble Sort
    template <class item_t>
    void bubbleUp (std::vector<item_t>& values, int start)
    {
        /*
            Precond: values is a reference to a vector of "item_t". start
is of type int. start refers to the index of the first unsorted position
of the vector.
            Postcond: Starting at the last element of the vector, if the
current element at index i is less than the previous element, swap these
two elements. Otherwise, do
            nothing. Do this while i is greater than start.
        */
        int end = values.size()-1;
        for (int i = end; i > start; i--) {
            if (values.at(i) < values.at(i-1)) swap(values.at(i),
values.at(i-1));
        }
    }

    template<class item_t>
    void bubbleSort(std::vector<item_t>& values)
    {
        /*
            Precond: values is a reference to a vector of "item_t". item_t
is a type that can be compared using logical operators.
            Postcond: values is sorted using the Bubble Sort algorithm.
        */
        int n = values.size();
        for (int current = 0; current < n-1; current++) {
            bubbleUp(values, current);
        }
    }
```

Runtimes:

	Average Runtime of Bubble Sort (sec)
10 Elements	0
100 Elements	0.000169
1000 Elements	0.012395
10000 Elements	1.24944
100000 Elements	130.169632



2. **Short Bubble Sort:**

```
Set current to index of first element
while more elements in unsorted section of list AND not sorted
    "Bubble up" smallest element to the top of unsorted section
    by swapping values as necessary; if no swaps are made, set
    isSorted to true
    Increment current
```

My Implementation:

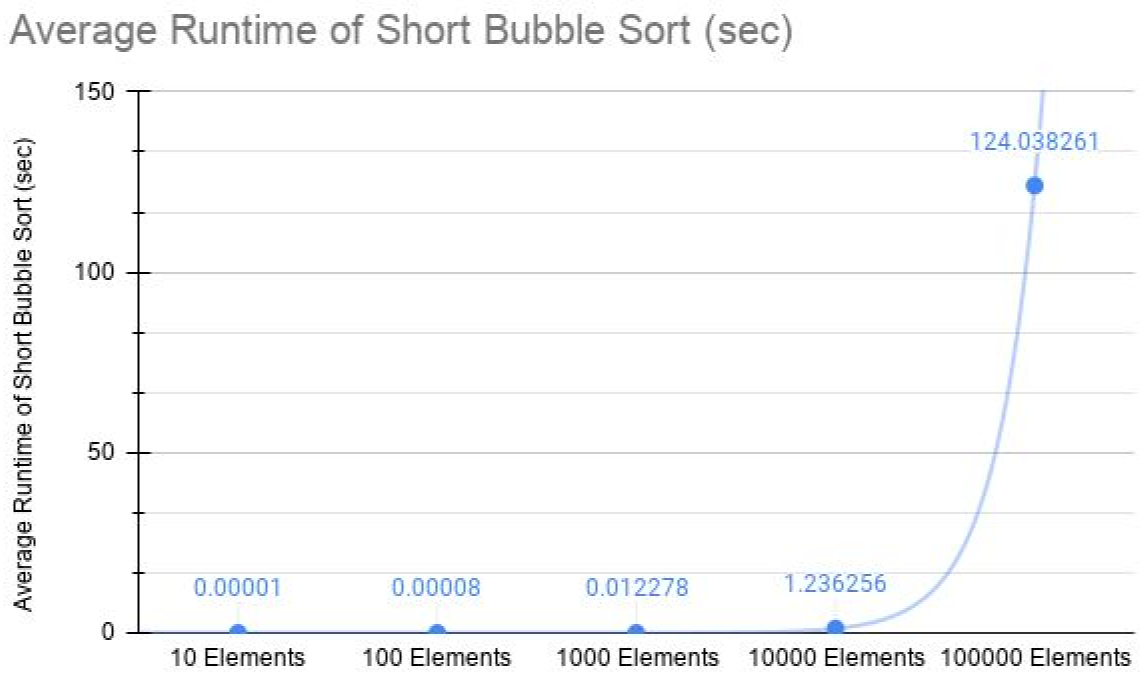
```
//Short Bubble Sort
    template <class item_t>
    void bubbleUpShort(std::vector<item_t>& values, int start, bool&
isSorted)
    {
        /*
            Precond: values is a reference to a vector of values. start is
of type int. isSorted is a reference to a boolean
            flag, which determines whether or not values is sorted or not.
            Postcond: Starting at the last element of the vector, if the
current element at index i is less than the previous element, swap these
two elements. Otherwise, do
            nothing. Do this while i is greater than start. If no swaps
occur throughout the bubbleUp process, isSorted will be true, since this
indicates
            that the list is sorted. Otherwise, reset isSorted back to
false.
        */
        isSorted = true;
        int end = values.size()-1;
        for (int i = end; i > start; i--) {
            if (values.at(i) < values.at(i-1)) {
                swap(values.at(i), values.at(i-1));
                isSorted = false;
            }
        }
    }

    template <class item_t>
    void shortBubbleSort(std::vector<item_t>& values)
    {
        /*
```

```
            Precond: values is a reference to a vector of "item_t". item_t
is a type that can be compared using logical operators.
            Postcond: values is sorted using the Short Bubble Sort
algorithm.
        */
        bool isSorted = false;
        int n = values.size(), current = 0;
        while (current < n-1 && !isSorted) {
            bubbleUpShort(values, current, isSorted);
            current++;
        }
    }
```

Runtimes:

	Average Runtime of Short Bubble Sort (sec)
10 Elements	0.00001
100 Elements	0.00008
1000 Elements	0.012278
10000 Elements	1.236256
100000 Elements	124.038261



3. Selection Sort

```
Set current to index of first element in list

while more items in unsorted section of list

    Swap the element at current with the minimum value of

    the unsorted section of the array

    Increment current
```

My Implementation:

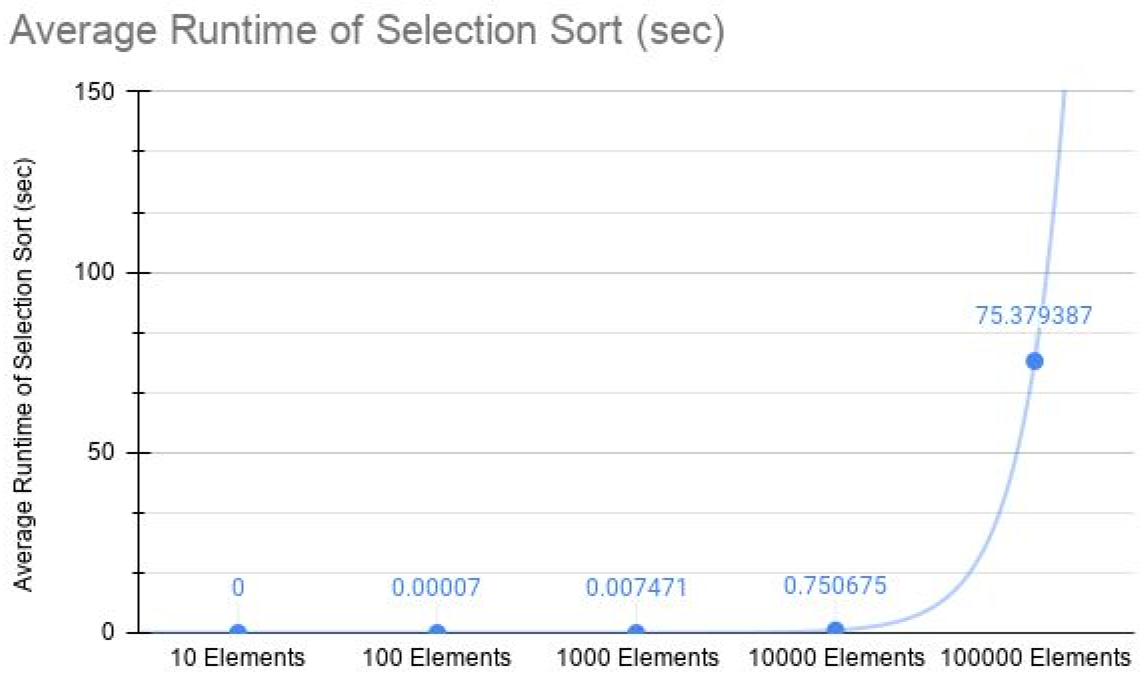
```
    //Selection Sort
    template <class item_t>
    int indexOfMin(const std::vector<item_t>& values, int start)
    {
        /*
            Precond: values is a const reference to a vector of "item_t".
start is of type int. start refers to the index of the first
            unsorted element in the vector.
            Postcond: The minimum element of the unsorted section of the
vector is found, and returned.
        */
        int minIndex = start;
        for (int i = start+1; i < values.size(); i++) {
            if (values.at(i) < values.at(minIndex)) minIndex = i;
        }
        return minIndex;
    }

    template<class item_t>
    void selectionSort(std::vector<item_t>& values)
    {
        /*
            Precond: values is a reference to a vector of "item_t". item_t
is a type that can be compared using logical operators.
            Postcond: values is sorted using the Selection Sort algorithm.
        */
        int lastIndex = values.size()-1;
        for (int current = 0; current < lastIndex; current++) {
            swap(values.at(current), values.at(indexOfMin(values,
current)));
        }
    }
```

Runtimes:

	Average Runtime of Selection Sort (sec)
10 Elements	0
100 Elements	0.00007
1000 Elements	0.007471
10000 Elements	0.750675

100000 Elements	75.379387



4. **Insertion Sort**

```
Set current to index of first element in the list
while more items in unsorted section
    Swap current with elements in sorted section of list until
    current is in its proper location
    Increment current
```

My Implementation:

```
//Insertion Sort
    template<class item_t>
    void insertItem(std::vector<item_t>& values, int current)
    {
        /*
```

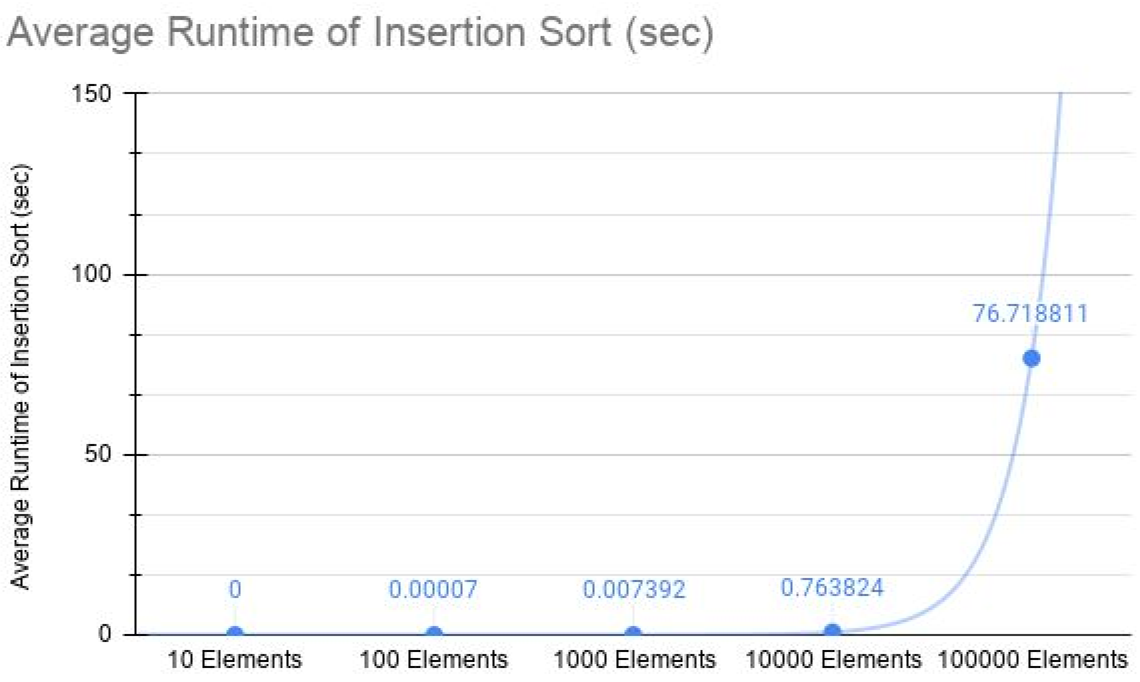
```
            Precond: values is a reference to a vector of "item_t".
current is of type int. current refers to the index of the first item
            of the non-sorted section of the vector.
            Postcond: The value at index is inserted into the sorted
section of values in its proper location.
        */
        bool isFinished = false;
        //Loop runs until either current has reached the first index of
vector values or until item has been placed correctly in vector
        while (current != 0 && !isFinished) {
            if (values.at(current) < values.at(current-1)) {
                swap(values.at(current), values.at(current-1));
                current--;
            } else {
                isFinished = true;
            }
        }
    }

    template <class item_t>
    void insertionSort(std::vector<item_t>& values)
    {
        /*
            Precond: values is a reference to a vector of "item_t". item_t
is a type that can be compared using logical operators.
            Postcond: values is sorted using the Insertion Sort algorithm.
        */
        for (int current = 0; current < values.size(); current++) {
            insertItem(values, current);
        }
    }
```

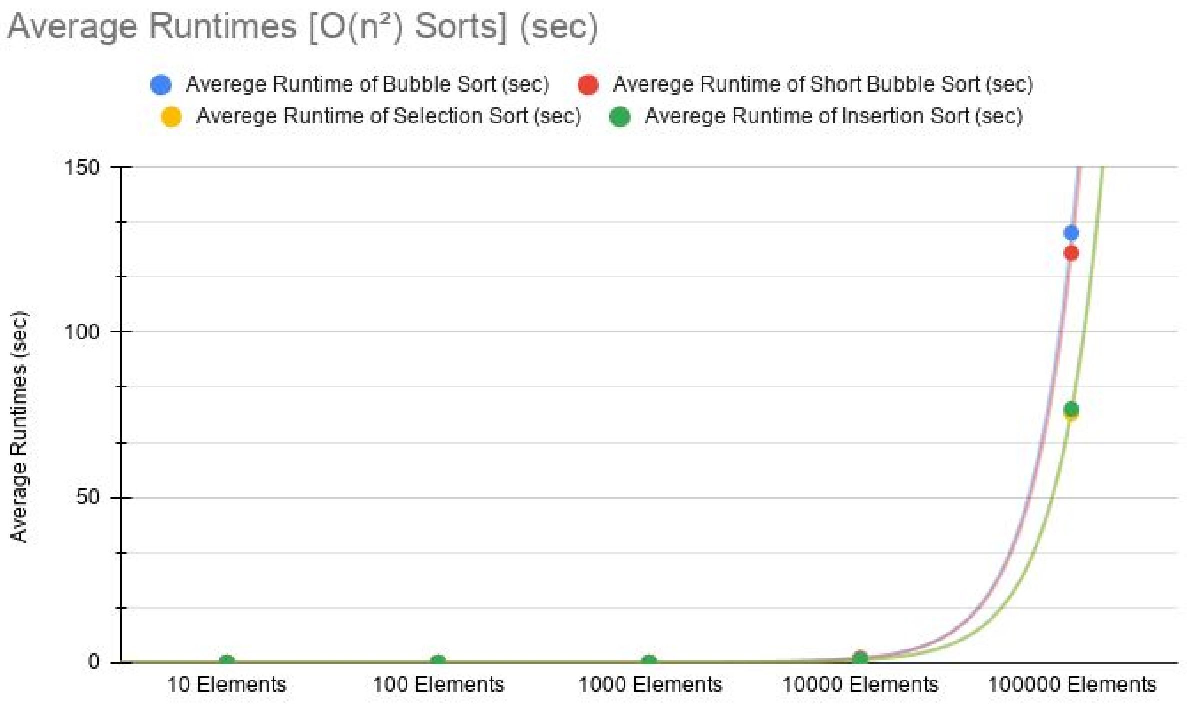
Runtimes:

	Average Runtime of Insertion Sort (sec)

10 Elements	0
100 Elements	0.00007
1000 Elements	0.007392
10000 Elements	0.763824
100000 Elements	76.718811



Comparing O(n²) Sorts:



O(n log(n)) Sorts:

1. **Merge Sort**

```
Set middle equal to the firstIndex plus lastIndex divided by
two
Call mergeSort on first half of list [firstIndex, middle]
Call mergeSort on second half of list [middle+1, lastIndex]
Merge the two lists back into one sorted list
```

My Implementation:

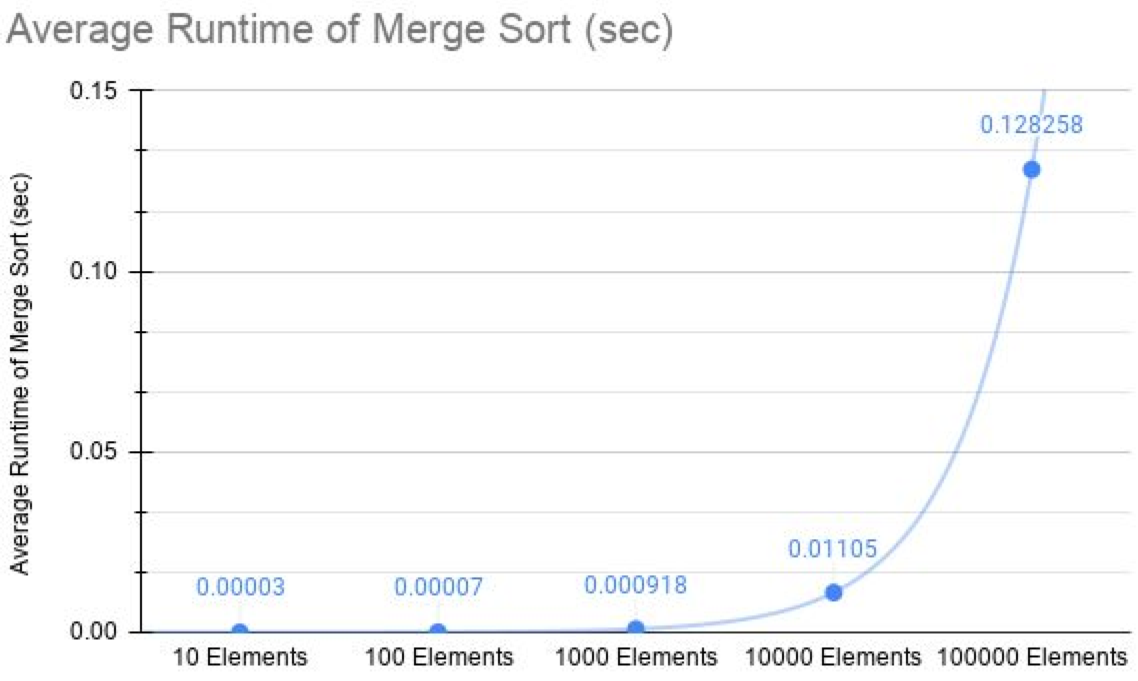
```
//Merge Sort
    template <class item_t>
    void merge(std::vector<item_t>& values, int left, int middle,
int right)
    {
        /*
            Precond: values is a reference to a vector of "item_t".
left, middle, and right are all of type int. left refers to the
            left most index of the vector that we plan to merge,
middle refers to the middle index of the vector that we plan to
merge,
            while right refers to the right index of the vector that
we plan to merge.
            Postcond: The two subvectors formed by the split at
middle are merged back together into values.
        */
        //Create two subvectors, which represent the two halfs of a
particular section in values.
        int leftSize = middle-left+1, rightSize = right-middle;
        std::vector<item_t> leftSub, rightSub;
        leftSub.reserve(leftSize);
        rightSub.reserve(rightSize);
        for (int i = 0; i < leftSize; i++) {
            leftSub.push_back(values.at(i+left));
        }
        for (int i = 0; i < rightSize; i++) {
            rightSub.push_back(values.at(i+middle+1));
        }
        //Merge Process:
        int totalItr = left, leftItr = 0, rightItr = 0;
        while (leftItr < leftSize && rightItr < rightSize) {
            if (leftSub.at(leftItr) <= rightSub.at(rightItr)) {
                values.at(totalItr) = leftSub.at(leftItr);
                leftItr++;
            } else {
                values.at(totalItr) = rightSub.at(rightItr);
                rightItr++;
            }
            totalItr++;
        }
        //Merge remaining elements back into values
        while (leftItr < leftSize) {
            values.at(totalItr) = leftSub.at(leftItr);
            leftItr++;
            totalItr++;
        }
        while (rightItr < rightSize) {
            values.at(totalItr) = rightSub.at(rightItr);
            rightItr++;
            totalItr++;
        }
    }

    template <class item_t>
    void mergeSort(std::vector<item_t>& values, int left, int right)
    {
        /*
            Precond: values is a reference to a vector of "item_t".
left and right are both of type int. left represents the leftmost
            index of the array to be sorted, while right represents
the rightmost index of the array to be sorted.
            Postcond: This method recursively breaks down the values
vector, and then merges these divided parts into one sorted vector.
        */
        //Base Case: values has been broken down to a size of 1 or
less elements at a particular point in the vector.
        if (left >= right) return;
        //Recursive Case: Break down the vector into halves: left
and right half, and merge together each half in a sorted manner.
        int middle = (left+right)/2;
        mergeSort(values, left, middle);
        mergeSort(values, middle+1, right);
        merge(values, left, middle, right);
    }
```

Runtimes:

	Average Runtime of Merge Sort (sec)
10 Elements	0.00003

100 Elements	0.00007
1000 Elements	0.000918
10000 Elements	0.01105
100000 Elements	0.128258



2. **Quick Sort**

```
Set split equal to first plus last divided by two
Using swaps, move all elements less than or equal to split
to the left side of list, and all elements greater than
split to the right side of the list
Set splitPoint1 equal to the first element in right side of
list, and splitPoint2 equal to the last element in left side
of list
```

```
If splitPoint1 is less than right, call quickSort on the
left side of list
If splitPoint2 is greater than left, call quickSort on the
right side of list
```

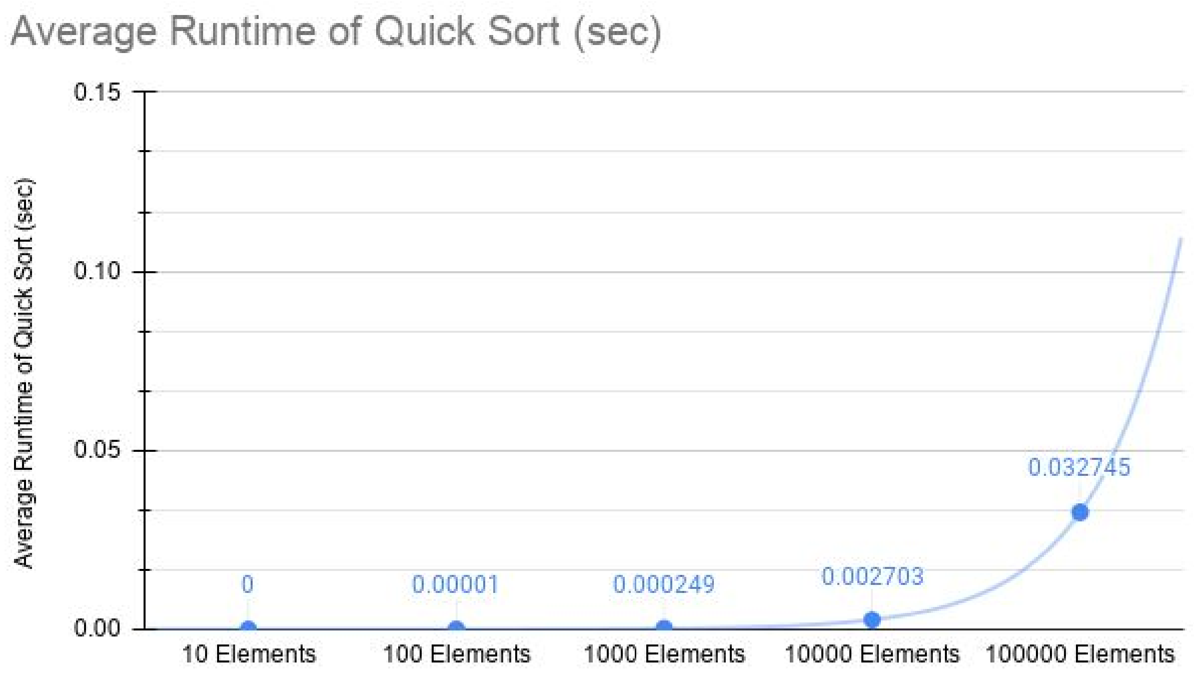
My Implementation:

```
    //Quick Sort
    template <class item_t>
    void split(std::vector<item_t>& values, int left, int right,
int& splitPoint1, int& splitPoint2)
    {
        /*
            Precond: values is a reference to a vector of "item_t".
left and right are both of type int, while splitPoint1 &
            splitPoint2 are references to ints. left represents the
leftmost index of the particular section of the vector to
            be split, while right represents the rightmost index of
the particular section of the vector to be split. Both
            split points will be determined once the splitting
process has completed.
            Postcond: All elements greater than the value of the
index of the average of left and right are shifted to the
            right of splitPoint1, while all elements less or equal
to the value of the index of the average of the left and
            right are shifted to the left of splitPoint2.
        */
        item_t splitValue = values.at((left+right)/2);
        bool notFound;
        do {
            notFound = true;
            while (notFound) {
                if (values.at(left) >= splitValue) {
                    notFound = false;
                } else {
                    left++;
                }
            }
            notFound = true;
            while (notFound) {
                if (values.at(right) <= splitValue) {
                    notFound = false;
                } else {
                    right--;
                }
            }
            if (left <= right) {
                swap(values.at(left), values.at(right));
                left++;
                right--;
            }
        } while (left <= right);
        splitPoint1 = left;
        splitPoint2 = right;
    }

    template <class item_t>
    void quickSort(std::vector<item_t>& values, int left, int right)
    {
        /*
            Precond: values is a reference to a vector of "item_t".
left and right are both of type int. left represents the leftmost
            index of the array to be sorted, while right represents
the rightmost index of the array to be sorted.
            Postcond: This method recursively breaks down the values
vector by using a splitting point. Once values has been fully broken
            down, it will be sorted.
        */
        //Base Case: values has been broken down to a size of 1 or
less elements at a particular point in the vector.
        if (left >= right) return;
        //Recursive Case: The section of the vector is broken down
on a split value.
        int splitPoint1, splitPoint2;
        split(values, left, right, splitPoint1, splitPoint2);
        if (splitPoint1 < right) quickSort(values, splitPoint1,
right);
        if (splitPoint2 > left) quickSort(values, left,
splitPoint2);
    }
```

Runtimes:

	Average Runtime of Quick Sort (sec)
10 Elements	0
100 Elements	0.00001
1000 Elements	0.000249
10000 Elements	0.002703
100000 Elements	0.032745



O((n+b) * log$_b$(k)) Sorts [B = Number base, k = max possible value]:

1. **Radix Sort**

```
for position going from 1 to numPositions
    for counter going from 0 to numValues-1
        Set val equal to digit at position “position” of list
        Enqueue val at counter into queues[val]
    Collect queues
```

My Implementation:

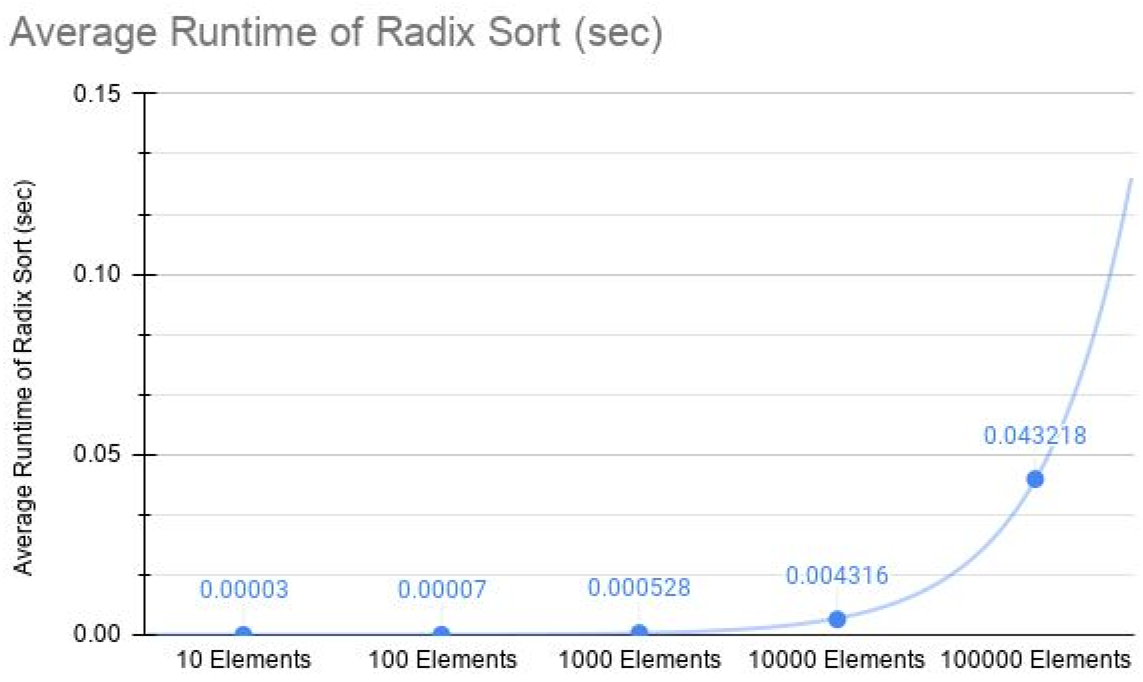
```
   //Radix Sort
    template<class item_t>
    void collectQueues(std::vector<item_t>& values, std::queue<item_t>
queues[], int radix)
    {
        /*
```

```
            Precond: values is a reference to a vector of "item_t". queues
is a vector of queues of size radix. radix is of type int,
            and is the radix of the data.
            Postcond: The queues are emptied, and determine the new order
of values, which is based on the ordering of one of the positions
            of the data.
        */
        int i = 0;
        for (int j = 0; j < radix; j++) {
            while (!queues[j].empty()) {
                item_t val = queues[j].front();
                queues[j].pop();
                values.at(i) = val;
                i++;
            }
        }
    }

    template <class item_t>
    void radixSort(std::vector<item_t>& values, int numPositions, int
radix)
    {
        /*
            Precond: values is a reference to a vector of "item_t".
numPositions is of type int, and refers to the max length of the elements
            in values. radix is of type int, and refers to the number of
unique digits that can represent a number. NOTE: THIS IMPLEMENTATION
            ONLY WORKS FOR INTEGER TYPES!!!
            Postcond: values is sorted using the radix sort algorithm.
        */
        for (int i = 1; i <= numPositions; i++) {
            std::queue<item_t> queues[radix];
            for (int j = 0; j < values.size(); j++) {
                item_t val = (int)(values.at(j)/pow(10, i-1))%10;
                queues[val].push(values.at(j));
            }
            collectQueues(values, queues, radix);
        }
    }
```

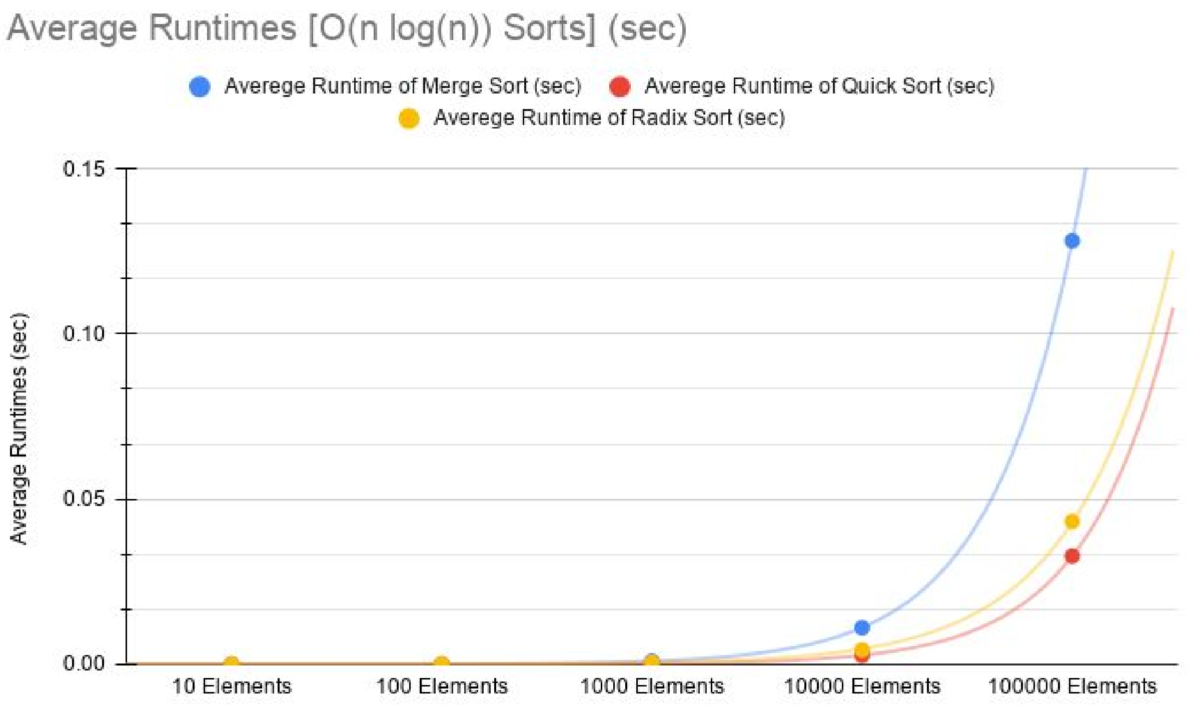
Runtimes:

	Average Runtime of Radix Sort (sec)
10 Elements	0.00003
100 Elements	0.00007
1000 Elements	0.000528
10000 Elements	0.004316
100000 Elements	0.043218

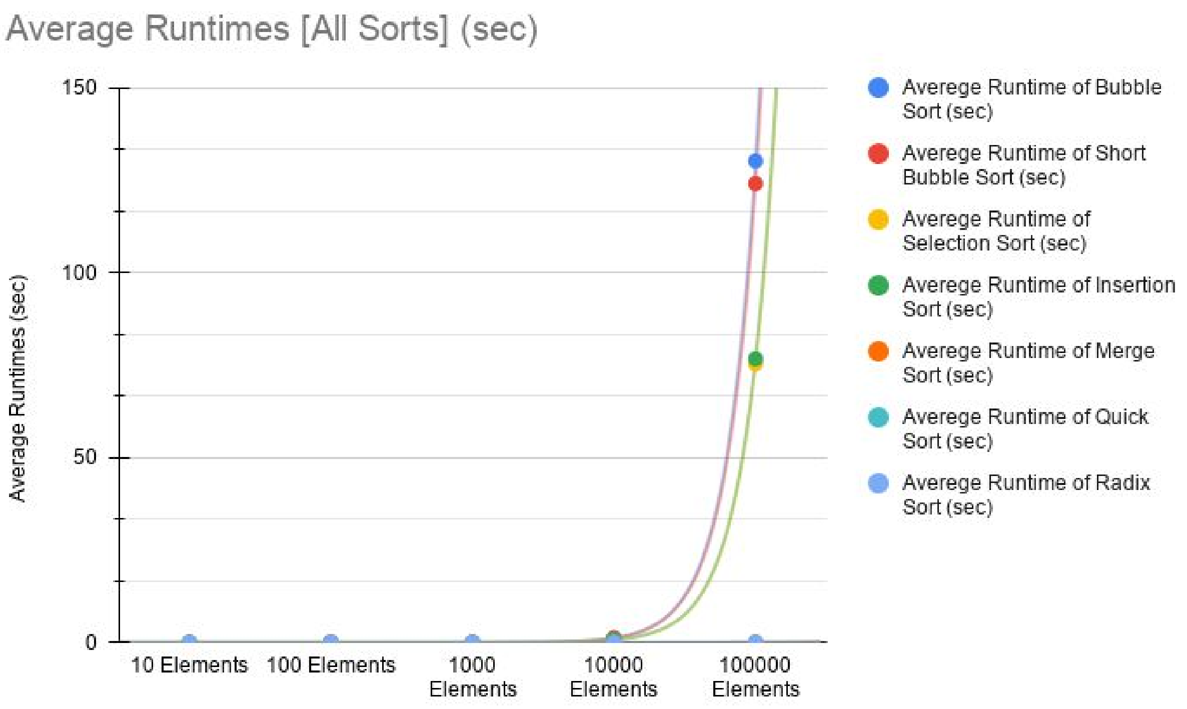


Comparing O(n log(n)) Sorts:

NOTE: Radix Sort is included, even though it is not an O(n log(n)) sort, since it has comparable runtimes.



Comparing All Sorts:



Conclusions:

All in all, the results of these tests were unsurprising: as the number of elements increased, the $O(n^2)$ suffered relative to the $O(n \log(n))$ sorts (and Radix Sort). However, when I took a closer look at the data, I noticed a few interesting things.

1. *Both Bubble Sort and Short Bubble Sort should* ***never*** *be used on randomized, unsorted data.*

I could discuss the differences between the data for Bubble Sort and Short Bubble Sort, but there is not much point: both of these sorts are outperformed by every other sort, including the other $O(n^2)$ sorts.

2. *For smaller data sets (<1000 elements), Selection Sort and Insertion Sort suffice.*

Both Selection Sort and Insertion Sort had similar average runtimes for all lists, with Selection Sort being a hair faster for each list. For the 10 and 100 element lists, the average runtime per sort were pretty similar results for each sort: each had an average runtime of less than a millisecond. Therefore, because of the complexity of Merge, Quick, and Radix Sort, it may be easier to simply implement one of the elementary sorts, as they have less overhead. However, once the data exceeds about 1000 elements, the $O(n^2)$ sorts really begin to suffer (as can be seen by the graph comparing all the sorts).

3. *Overall, Quick Sort reigns supreme for randomized, unsorted data, but Radix Sort is a close second.*

To me, this is perhaps the most interesting conclusion I was able to draw. I expected Quick Sort to suffer with lower amounts of elements due to the amount of overhead, but to my surprise, it was still faster than every other sort **at all list sizes**. I wasn't sure what to expect from Radix Sort, due to its wacky nature relative to the other sorts, but I knew it would be faster than the

$O(n^2)$ sorts. To my surprise, it was on par with Merge Sort until 1000 elements, which it became about 4 times faster than Merge Sort for the rest of the lists. Radix Sort definitely defied my expectations!