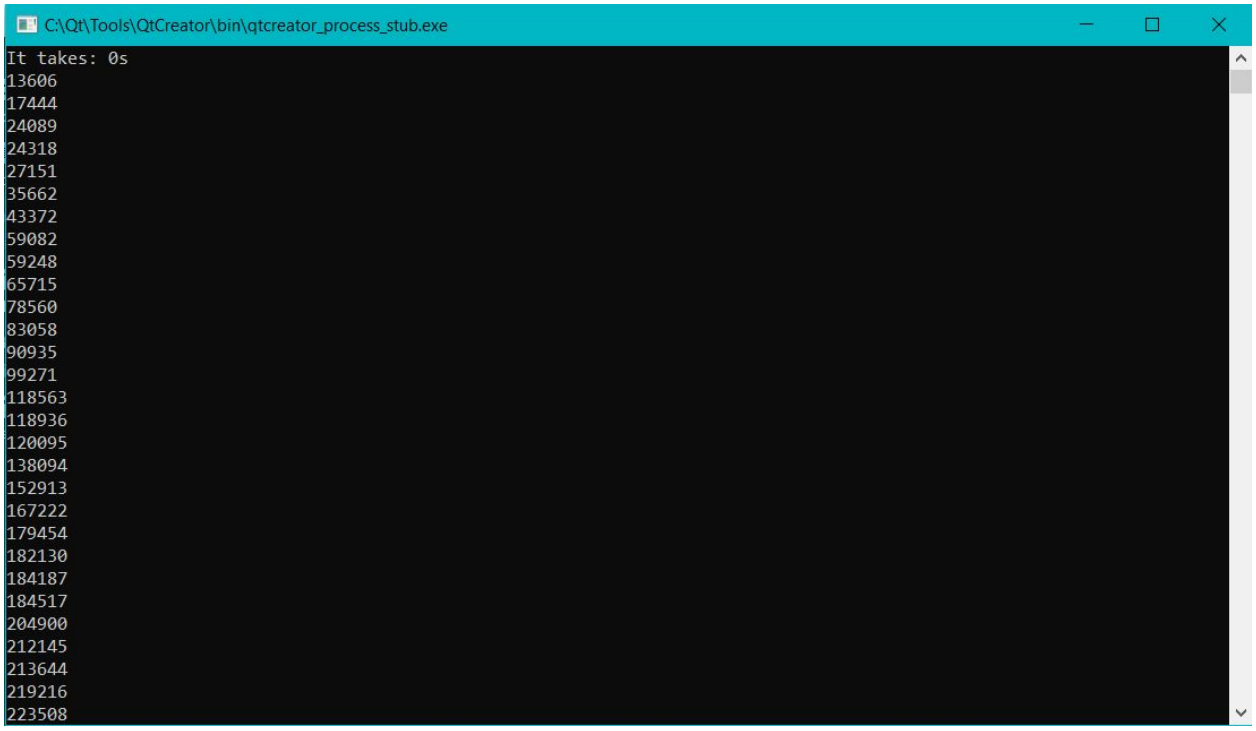
YunHo(Louis) Law
CS 361
Lab 2 Report
02/10/2019

Code for Radix Sort:

```
10  void CountingSort(int arr[], int size, int placeValue){
11      int* B = new int[size]; // Array B stores the output
12      int* C = new int[10]; //Array C stores the counter
13      //initialize the Array C to 0
14      for(int i = 0; i < 10; i++){
15          C[i] = 0;
16      }
17      //store the # of each value in Array C
18      for(int j = 0; j < size; j++){
19          C[(arr[j] / placeValue) % 10] += 1;
20      }
21      // add up the value in Array C which will give us the index that we should put into our Array B
22      for(int k = 1; k < 10; k++){
23          C[k] += C[k - 1];
24      }
25      // put the numbers into array B base on the Index in array C
26      for(int l = size - 1; l >= 0; l--){
27          B[C[(arr[l] / placeValue) % 10] - 1] = arr[l];
28          C[(arr[l] / placeValue) % 10] -= 1;
29      }
30      // Copy the whole output array back into the original
31      for(int m = 0; m < size; m++){
32          arr[m] = B[m];
33      }
34  }
35
36  void RadixSort(int arr[], int size, int LargestDigit){
37      int placevalue = 1; // initial place value from the right to the left
38      // counting sort each place value
39      for(int i = 1; i <= LargestDigit; i++){
40          CountingSort(arr, size, placevalue);
41          placevalue *= 10;
42      }
43  }
```

Output for sorted array (first 1000 items)

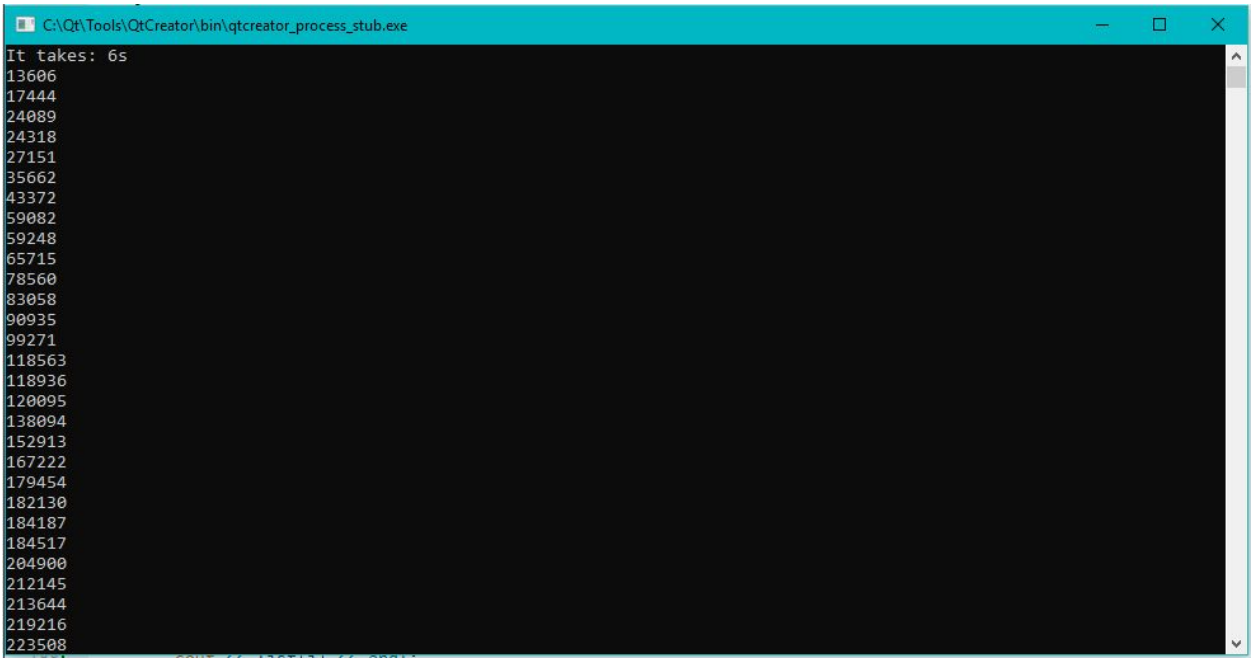


BucketSort Code

```
97  void InsertionSort(vector<int>& list){
98      int i;
99      int k;
100     int temp;
101     for(i = 1; i < list.size(); i++){
102         k = list.at(i);
103         temp = i - 1;
104         while(temp >= 0 && list.at(temp) > k){
105             list.at(temp + 1) = list.at(temp);     implicit conversion changes signe
106             temp--;
107         }
108         list.at(temp + 1) = k;
109     }
110 }
111
112 void BucketSort(int arr[], int size){
113     int bucketSize = 100000; // causing memory problems with too many buckets
114     vector<int> bucket[bucketSize]; //initialize buckets
115     // puting all the items into its designated buckets
116     for(int i = 0; i < size; i++){
117         int bucketIndex = arr[i] / bucketSize;
118         if(bucketIndex >= bucketSize){
119             bucketIndex = bucketSize - 1;
120         }
121         bucket[bucketIndex].push_back(arr[i]);
122     }
123     // insertion sort each buckets
124     for(int i = 0; i < bucketSize ; i++){
125         InsertionSort(bucket[i]);
126     }
127     int k = 0; // array index
128     // concatenate each bucket back into an original array
129     for(int i = 0; i < bucketSize; i++){
130         for(auto it = bucket[i].begin(); it != bucket[i].end(); ++it){
131             arr[k] = *it;
132             k++;
133         }
134     }
135
136 }
```

I am still having problem with the bucket size and memory control. Still working on the computation of the bucketIndex.

BucketSort Output (the first 1000 sorted items)



Since the bad implementation of the bucketSize and bucketIndex, the algorithm does not take any advantage out of Θ(n) complexity. Instead, it does a lot more insertion sort which makes the algorithm complexity be Θ(n^2). I am still in process optimizing the implementation and the computation.

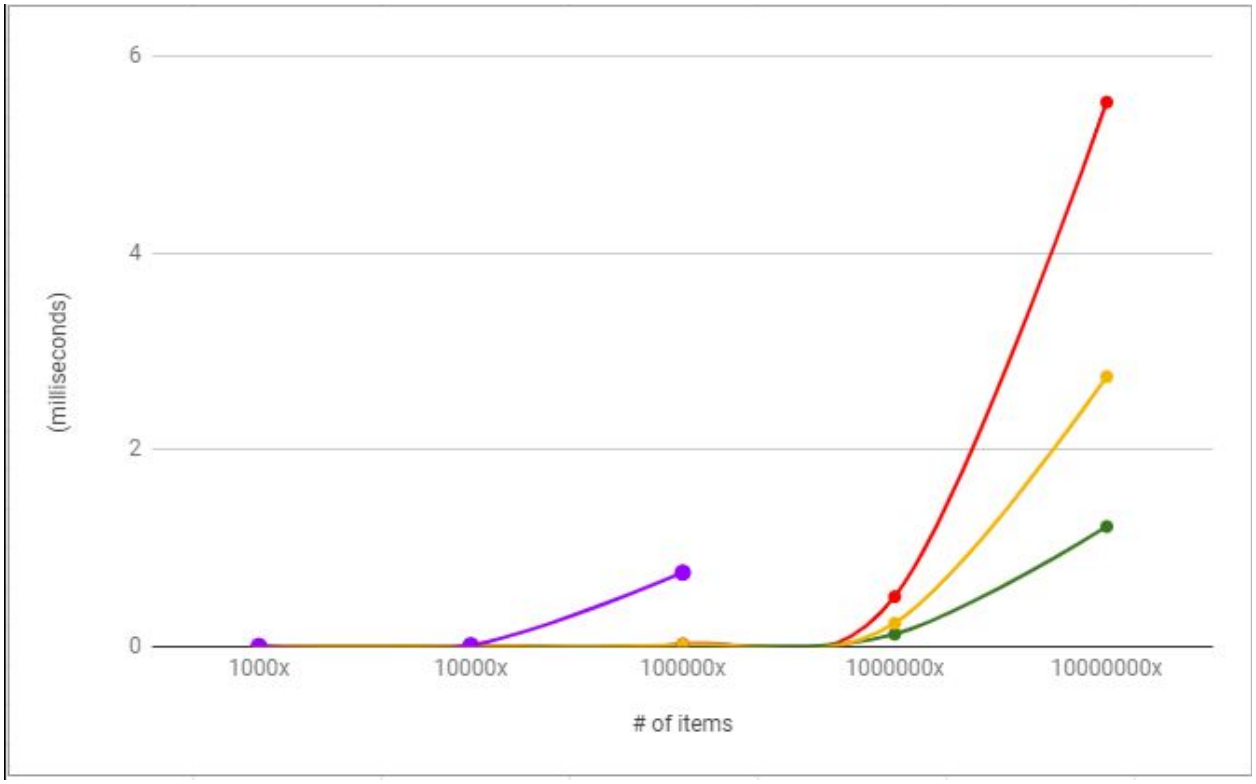
PrintTenLargest Code

```
vector<int> FindTenLargest(int arr[], int startIndex, int endIndex){
    vector<int> largest = {0};
    if(endIndex == 0 && largest.size() == 10){
        return largest;
    }
    else{
        largest = FindTenLargest(arr, startIndex, endIndex - 1);
        if(arr[endIndex] > arr[endIndex - 1]){
            largest.push_back(arr[endIndex]);
        }
    }
}

void PrintTenLargest(int arr[], int startIndex ,int endIndex){
    vector<int> largest = FindTenLargest(arr, startIndex, endIndex);
    InsertionSort(largest);
    for(auto i = largest.rend(); i != largest.rbegin(); --i){
        cout << *i << endl;
    }
}
```

I think using divide and conquer should be easier to implement this solution since we can break down the array and then find maybe 10 largest items for each subarray and then sort each subarray, finally, combine.

Table and Graph



MergeSort	1000x	10000x	100000x	1000000x	10000000x
1st Run	0	0.003	0.025	0.495	5.584
2nd Run	0	0.002	0.026	0.49	5.516
3rd Run	0	0.002	0.026	0.535	5.5
AVG:	0	0.002333333333	0.02566666667	0.5066666667	5.533333333
RadixSort	1000x	10000x	100000x	1000000x	10000000x
1st Run	0	0.001	0.011	0.126	1.126
2nd Run	0	0.002	0.013	0.127	1.266
3rd Run	0	0.001	0.012	0.126	1.267
AVG:	0	0.001333333333	0.012	0.1263333333	1.219666667
BucketSort	1000x	10000x	100000x	1000000x	10000000x
1st Run	0.005	0.014	0.756		
2nd Run	0.006	0.014	0.749		
3rd Run	0.006	0.014	0.751		
AVG:	0.005666666667	0.014	0.752		
QuickSort	1000x	10000x	100000x	1000000x	10000000x
1st Run	0	0.002	0.02	0.236	2.739
2nd Run	0	0.002	0.02	0.237	2.768
3rd Run	0	0.002	0.021	0.238	2.723
AVG:	0	0.002	0.02033333333	0.237	2.743333333

RadixSort has better complexity than any other Sorting Algorithm which give us linear running time.
BucketSort does not have the expected complexity due to the implementation.