Speed Comparison of Five Separate Sorts Given Different Ordered Elements

By Connor J. Marler

Due: 13 November, 2022

Abstract

This experiment demonstrates the speeds of several different sorting algorithms. Those include insertion sort, merge sort, heap sort, quick sort, and randomized quick sort. Merge sort came out overall the quickest on average and quick and randomized quick came out on bottom. This test was conducted by sorting a list of five hundred elements one thousand times and taking the average to create one data point. Ten data points are collected and plotted out for ease of understanding.

1 Motivation

When picking a sorting algorithm, it is important to know what the speed of sorting best, worst, and random case scenarios are prior to applying the algorithm to a problem. Certain algorithms may need tweaking to optimize on any given data set. Prior to that, it is important to see how they operate when sorting through the aforementioned cases. By picking the correct sort for a problem, there is a measurable and empirical difference in time efficiency. Broadly, sorting algorithms are invaluable to computation due to their frequent use and necessity. There are a plethora of different algorithms to sort data in a set, however, some operate differently in different scenarios. There are insertion sorts, merge sorts, bubble sorts, shell sorts, heap sorts, and many more. These are only a few of the examples of different ways to sort data, and each of them have their own unique advantages, disadvantages, and methodology for achieving the same goal of sorting data as quickly and effectively.

2 Background

The five sorting algorithms that I am looking at in this experiment are insertion, merge, heap, quick, and a modified quicksort that improves consistency. Insertion sort is one the fundamental sorting algorithms. In the best case, the time complexity of the insertion sort is O(n) and the time complexity of the average and worst case is $O(n^2)$. This best case scenario is when the list or array is already in a sorted order prior to the insertion sort being run over it. The average and worst case come from if the data structure's elements are randomly allocated with

the structure and whenever the elements are in the opposite of the order they are supposed to be in. Next, the merge sort. Broadly speaking, merge sort breaks the sorting container down in smaller pieces in order to sort them and then remerge them together. This leads to a time complexity of O(nlogn) for all three cases. This time complexity happens to also be the same for heap sort, also having a time complexity of O(nlogn). This algorithm separates off the biggest value of the heap and reassembles it. The quick sort algorithm works mildly similar to the merge sort in the sense that the container is broken up and things are sorted on a small level. When the container of elements is pre-sorted in a random order, then there is a time complexity of O(nlogn), however when the elements are already sorted or in reverse order, then the time complexity is $O(n^2)$. This is where the randomized quicksort comes into play. The randomization only runs with a time complexity of O(nlogn), due to a randomization practice that will be gone into further detail in the next section. This randomization leads to always having the same time complexity of the quicksort when the elements are in random order.

3 Procedure

First in line to be discussed is the insertion sort. Let's say for example there is an array of ten values in random order. The value we are focusing on is continually compared to the previous element if they need to be flipped, they get flipped as it travels down the set. This will happen until it gets fit into the correct spot. Then we shift the value we are focusing on to the opposite way we are comparing. This repeats until the entire list becomes sorted. The precondition that the use of this algorithm found in the appendix (InsertionSort.cpp) uses is that mainArray[] must not be empty and size must match the true size value of its paired array. It also takes in a templated array and the size of that array

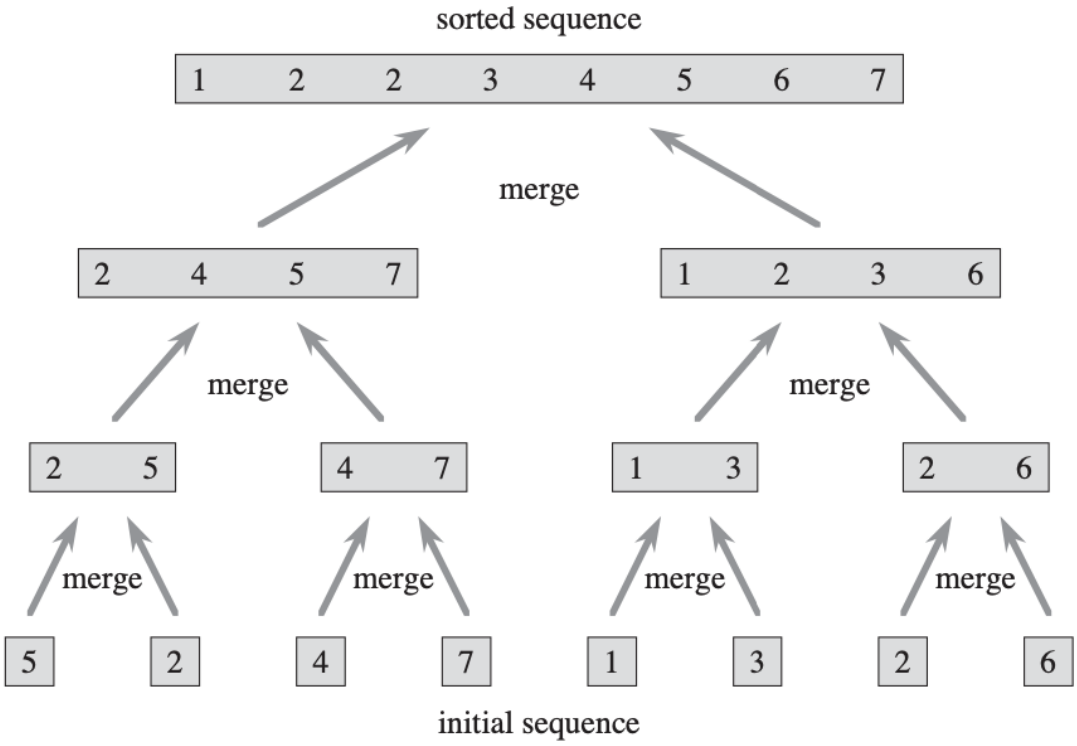
The postcondition is that mainArray[] is sorted in ascending order. The invariant that is used is: mainArray[i-1] <= mainArray[i] for all elements of mainArray[].

INSERTION-SORT(A)

```
1  for j = 2 to A.length
2      key = A[j]
3      // Insert A[j] into the sorted sequence A[1 .. j − 1].
4      i = j − 1
5      while i > 0 and A[i] > key
6          A[i + 1] = A[i]
7          i = i − 1
8      A[i + 1] = key
```

*Introduction to Algorithms 3rd Edition, CLRS pg. 18

Next, we will talk about the procedural steps with the merge sort. Merge sort will recursively call itself, and divide into smaller and smaller sections until it can be no longer divided. The elements are then compared and remerged back together whichever way that is determined by the conditional prior to the remerge.



*Introduction to Algorithms 3rd Edition, CLRS pg. 35

The precondition that was used for the merge function was assuring that the values taken in as parameters are a templated array and integer start and end points reflecting the specific subarray. The post condition used was that the result is written back to the original array. The loop invariant was hard to pin down, but was (indexOfMergedArray == left) && (indexOfSubArrayOne == 0) && (indexOfSubArrayTwo == 0).

```
MERGE(A, p, q, r)
 1  n_1 = q − p + 1
 2  n_2 = r − q
 3  let L[1 .. n_1 + 1] and R[1 .. n_2 + 1] be new arrays
 4  for i = 1 to n_1
 5      L[i] = A[p + i − 1]
 6  for j = 1 to n_2
 7      R[j] = A[q + j]
 8  L[n_1 + 1] = ∞
 9  R[n_2 + 1] = ∞
10  i = 1
11  j = 1
12  for k = p to r
13      if L[i] ≤ R[j]
14          A[k] = L[i]
15          i = i + 1
16      else A[k] = R[j]
17          j = j + 1
```

*Introduction to Algorithms 3rd Edition, CLRS pg. 31

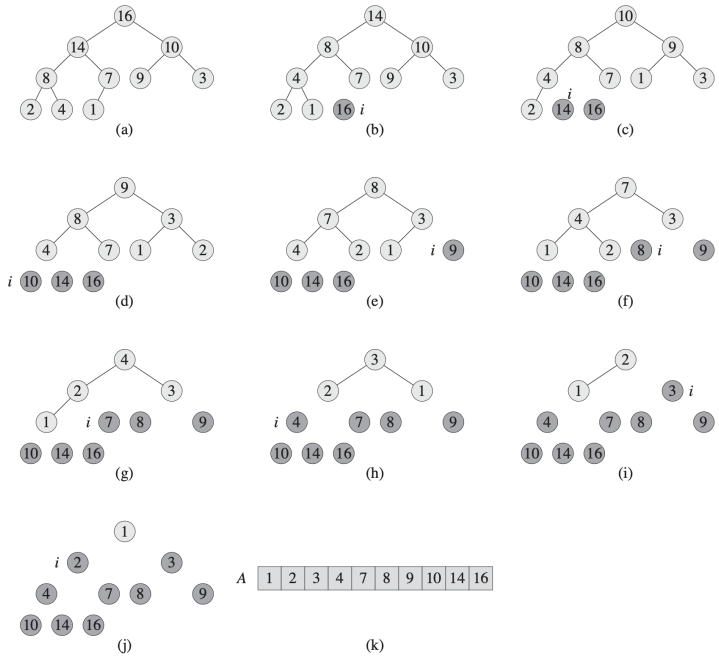
The Merge Sort function itself is what calls the merge sort function recursively and also the merge function. This function will take in as a parameter a templated array and integer start and end points reflecting the length of the array. The postcondition is that the list will now be sorted in ascending order.

```
MERGE-SORT(A, p, r)
1  if p < r
2      q = ⌊(p + r)/2⌋
3      MERGE-SORT(A, p, q)
4      MERGE-SORT(A, q + 1, r)
5      MERGE(A, p, q, r)
```

*Introduction to Algorithms 3rd Edition, CLRS pg. 34

The heap sort works by turning our array into a heap data structure or a binary tree. This is a tree data structure that satisfies the case that any parent is greater than or equal to its child. This allows for the root node to always be the largest in the tree. By doing this, we can use an

algorithm to remove the biggest node, and replace it again with the next biggest node and remove it off. All of this is moving around the elements of our array. When the root node is removed off of the tree, it becomes locked into the array in the correct spot. The heap sort algorithm is another recursive algorithm. Traditionally, this sorting algorithm is done with 3 separate functions. In my experimental implementation, it was combined down to two.



*Introduction to Algorithms 3rd Edition, CLRS pg. 161

The first of which being the Heap Sort function itself. The precondition is that it takes in a templated array and an integer of the size of the array. The postcondition is that the array that was taken in as a parameter is sorted. Our loop invariant for the heap sort function is that "For all 1 <= i < (N-1), arr[i] >= arr[i-1]".

```
HEAPSORT(A)
1  BUILD-MAX-HEAP(A)
2  for i = A.length downto 2
3      exchange A[1] with A[i]
4      A.heap-size = A.heap-size − 1
5      MAX-HEAPIFY(A, 1)
```

*Introduction to Algorithms 3rd Edition, CLRS pg. 160

The Build Max Heap function builds the heap initially. This put the largest element on top in order to then run Heapify. The Precondition is that it takes in a templated array. The post condition is that a heap will have been built from the array. The invariant can be shown as for all i, arr[i+1, i+2, ..., upperLimit] is the root of a max heap.

```
BUILD-MAX-HEAP(A)
1  A.heap-size = A.length
2  for i = ⌊A.length/2⌋ downto 1
3      MAX-HEAPIFY(A, i)
```

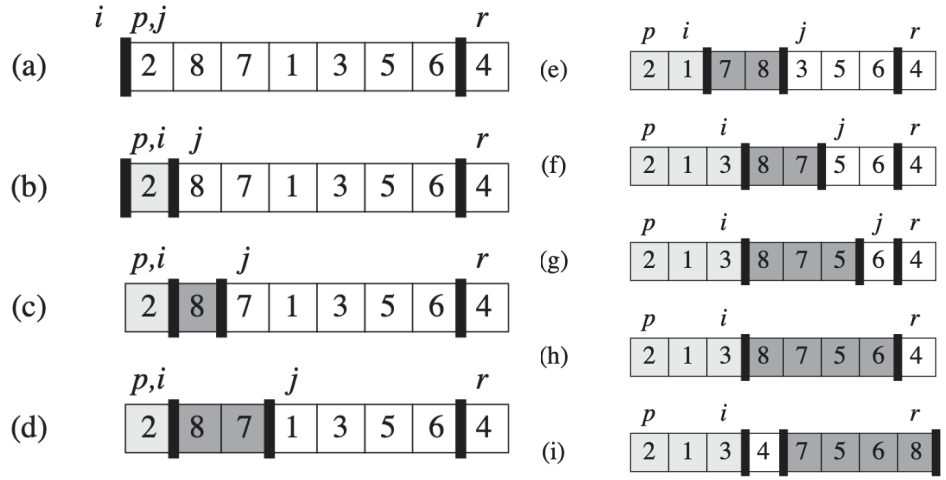
*Introduction to Algorithms 3rd Edition, CLRS pg. 167

The last part of the heap sort is the heapify. This is the heart of the heap sort. The precondition of this function is that the left and right children of A[i] are roots of max heaps. This function takes in a templated array and an int of the length n and node index of i that is also an integer. The post condition is that A[i] is the root of a max heap.

```
MAX-HEAPIFY(A, i)
1  l = LEFT(i)
2  r = RIGHT(i)
3  if l ≤ A.heap-size and A[l] > A[i]
4      largest = l
5  else largest = i
6  if r ≤ A.heap-size and A[r] > A[largest]
7      largest = r
8  if largest ≠ i
9      exchange A[i] with A[largest]
10     MAX-HEAPIFY(A, largest)
```

*Introduction to Algorithms 3rd Edition, CLRS pg. 154

Next, I will discuss a bit about the procedure of quicksort. Quick sort designates an index of the array as what is known as a pivot. This pivot is then compared to the other other values in the array and then partitioned into subarrays organized by their size compared to the pivot point. This is another recursive function.



*Introduction to Algorithms 3rd Edition, CLRS pg. 172

The first of two functions involved in this sort is the quick sort function itself. The precondition for the quick sort is that A must contain n items that can be compared, $1 \leq P \leq n$, $1 \leq R \leq n$. The post condition is that A must contain a permutation of the original items, $A[1] \leq A[2] \leq \ldots A[n]$. This function checks to see if $p < r$, and if it is then assign the partition and call quick sort recursively using that partition.

```
QUICKSORT(A, p, r)
1  if p < r
2      q = PARTITION(A, p, r)
3      QUICKSORT(A, p, q − 1)
4      QUICKSORT(A, q + 1, r)
```

*Introduction to Algorithms 3rd Edition, CLRS pg. 171

The partner function in this algorithm is called partition. This compares the array index to the pivot point and begins to partition the array based on the size of the value. In order for the partition function to operate correctly it must satisfy the precondition of taking in an array with at least two values and an integer of the starting index and the ending index. The post condition after the the values in between the indices is sorted in ascending order. There is a loop variant of once within the if statement and after the swap, arr[j] <= arr[pivot].

```
QUICKSORT(A, p, r)
PARTITION(A, p, r)
1  x = A[r]
2  i = p − 1
3  for j = p to r − 1
4      if A[j] ≤ x
5          i = i + 1
6          exchange A[i] with A[j]
7  exchange A[i + 1] with A[r]
8  return i + 1
```

*Introduction to Algorithms 3rd Edition, CLRS pg. 171

This poses an interesting modification leading to the randomized quick sort. Something that sets quicksort back is that there is a chance that depending on where the pivot is placed, issues could arise. The preconditions, postconditions, and invariants are all the same as the normal quick sort. According to Introductions to Algorithms 3rd Edition, CLS on page 179, “The changes to PARTITION and QUICKSORT are small. In the new partition procedure, we simply implement the swap before actually partitioning.”

```
RANDOMIZED-QUICKSORT(A, p, r)
1  if p < r
2      q = RANDOMIZED-PARTITION(A, p, r)
3      RANDOMIZED-QUICKSORT(A, p, q − 1)
4      RANDOMIZED-QUICKSORT(A, q + 1, r)
```

*Introduction to Algorithms 3rd Edition, CLRS pg. 179

```
RANDOMIZED-PARTITION(A, p, r)
1  i = RANDOM(p, r)
2  exchange A[r] with A[i]
3  return PARTITION(A, p, r)
```

*Introduction to Algorithms 3rd Edition, CLRS pg. 179

4 Testing

(A) Testing Plan and Results

Test Input	Expected Results	Actual Results	Corrective Action
Ascending Order	Array is sorted in ascending order.	Array is sorted in ascending order.	n/a
Descending Order	Array is sorted in ascending order.	Array is sorted in ascending order.	n/a
Random Order	Array is sorted in ascending order.	Array is sorted in ascending order.	n/a

(B) Problems Encountered

There were quite a bit of problems encountered as I performed this experiment. Throughout the development and testing phases of this project I took active note of the issues I was encountering in order to share them here. I will touch on the key complications that I experienced and resolved. The first thing I would like to mention is corrections made from my insertion sort from project one. After testing for errors, I was able to narrow down the issues to the sort function itself. After trying to correct this issue on my own, I decided to take advantage of a suggestion made in the project description. I was able to utilize online resources to piece together with my working driver code in order to correct errors made in the last assignment. Another error that I was repeatedly running into was my correcting the offsets on loops when looking at the algorithm. I frequently would arrive at a situation where my loop index would be off by 1 in either direction. What I believe to be the largest problem that I encountered was the implementation of the invariants. I did my best to go through class notes, the textbook, and by researching online, but discovering the correct way to implement the invariant in the program proved to be very difficult. I resolved this by trial and error. I would assert what I believed to be the correct code translation where I believed it should go and did my best to make it work. Lastly, the heap sort took me a bit of time to wrap my head around enough to work with. This proved to be the most difficult sort to understand.

5 Experimental Analysis

Each data point is gathered by sorting a 500 element array 1000 times and averaging the time it took to sort them. There were 10 data points gathered from each sort. The library used to do the timing is called chrono. The timing is done using nano seconds. In this section, I am going to break down each of the figures labeled in the index.

Figures 1 through 5 describe the individual sorts and the data gathered from each one. Figure 1 shows that the insertion sort operates best when the array is presorted in correct order and is the slowest when the presort order is the reverse of the end result. Differently, in figure 2, the merge sort functions at roughly the same speed when it comes to sorting an ascending and

descending presort order. When performing the merge sort, the randomized pre sort order is the slowest. Figure 3 shows that when it comes to heap sort, when the presort order is descending, the sort operates the fastest. The ascending and random presort order operate roughly at the same speed with the descending order being faster by a very small margin. Figure 4 shows that in the quick sort the fastest time was certainly when the presort order was random. Which leads to figure 5, the randomized quick sort. Which is much slower than the traditional quick sort. However, it is significantly more consistent than the traditional quick.

Figures 6 through 11 show off how each type of sort stacks up to their competitors when directly compared against each other when the presort order is the same. The fastest running algorithm for the ascending order is the insertion sort and the slowest is the quick sort. When the presort order is random, then the fastest is the quick sort and the slowest is the randomized quick sort. With respect to the descending order, the fastest was the merge sort and the slowest was quick sort.

6 Conclusion

Given the results from my experimentation, the merge sort operated the fastest on all accounts. According to my data, the quick sort had the overall slowest time. The randomized quick sort is more frequently slow. However this offers significantly more consistency than the more traditional counterpart. This is why compared to the quick sort, randomized quick sort is often the correct choice. If I was to redo this project, I would combine the files and generalize my code a bit more. I did it the way I did in order to focus very clearly on each sort in order to not confuse myself. Moving forward from this point, I believe the next course of action would be to perform the same amount of trial, but repeat this entire process for at least 5 different element container sizes. I would advise that the smallest container size would be 500 elements (which is the current size). By adding this piece to the research, we would be able to see how the amount of elements sorted will affect the speed of the sorting algorithm.

References

T. H. Cormen, C. E. LEiseron, R. L. Rivest, C. Stein., Introduction to Algorithms, 3rd Edition. MIT press Cambridge, 2001.

“C++ Program to Implement Insertion Sort Using Templates.” *SlayStudy*, 6 Aug. 2020, https://slaystudy.com/c-program-to-implement-insertion-sort-using-templates/.

“C++ Program to Implement Merge Sort Using Templates.” *SlayStudy*, 3 Aug. 2020, https://slaystudy.com/c-program-to-implement-merge-sort-using-templates/.

“C++ Program to Implement Quicksort Using Templates.” *SlayStudy*, 3 Aug. 2020, https://slaystudy.com/c-program-to-implement-quicksort-using-templates/.

Heap Sort.” *GeeksforGeeks*, 22 Sept. 2022, https://www.geeksforgeeks.org/heap-sort/.

“Quicksort Using Random Pivoting.” *GeeksforGeeks*, 20 June 2022, https://www.geeksforgeeks.org/quicksort-using-random-pivoting/.

Appendix

Insertion Sort			
	Time Measured In NanoSeconds		
Trials	Ascending	Descending	Random
1	3070	397244	397033
2	3147	397171	201989
3	2921	397142	201580
4	2980	398020	201460
5	2926	398235	201329
6	2949	397527	200928
7	2945	397520	202156
8	2933	397005	201676
9	2943	397029	201820
10	2922	397033	201374

Fig. 1

Merge Sort			
	Time Measured In NanoSeconds		
Trials	Ascending	Descending	Random
1	56339	54975	75940
2	56188	54976	75720
3	56209	54934	75768
4	56301	54988	75900
5	56148	55152	75632
6	56081	55013	75592
7	56151	55009	75677
8	56310	55134	75622
9	56257	54975	75624
10	56223	54933	75666

Fig 2.

Heap Sort			
	Time Measured In NanoSeconds		
Trials	Ascending	Descending	Random
1	114020	98101	114246
2	113181	97946	114684
3	112666	98072	114039
4	110764	97895	114014
5	110770	97869	114120
6	110583	97784	114187
7	110693	97823	114042
8	110680	97863	114089
9	111052	97766	114068
10	110976	97767	113981

Fig 3.

Quick Sort			
	Time Measured In NanoSeconds		
Trials	Ascending	Descending	Random
1	1689182	1269919	65197
2	1691725	1412792	64953
3	1684635	1298129	65149
4	1681968	1318297	65141
5	1680648	1220753	65016
6	1692546	1564820	65094
7	1693835	1627779	104796
8	1682202	1330157	65552
9	1683870	1232384	65529
10	1680073	1342578	65553

Fig 4.

Randomized Quick Sort			
	Time Measured In NanoSeconds		
Trials	Ascending	Descending	Random
1	470873	607845	613876
2	442181	597441	611029
3	486438	625988	613385
4	526038	625988	614228
5	538955	556479	614552
6	546669	574299	609339
7	464583	594938	605858
8	444372	592121	616273
9	443633	592203	614119
10	442846	600063	603004

Fig 5.

	Ascending Pre-Sort Order (Time in ns)				
Trials	Insertion	Merge	Heap	Quick	RandomQuick
1	3070	56339	114020	1689182	470873
2	3147	56188	113181	1691725	442181
3	2921	56209	112666	1684635	486438
4	2980	56301	110764	1681968	526038
5	2926	56148	110770	1680648	538955
6	2949	56081	110583	1692546	546669
7	2945	56151	110693	1693835	464583
8	2933	56310	110680	1682202	444372
9	2943	56257	111052	1683870	443633
10	2922	56223	110976	1680073	442846

Fig 6.

	Random Pre-Sort Order (Time in ns)				
Trials	Insertion	Merge	Heap	Quick	RandomQuick
1	397033	75940	114246	65197	613876
2	201989	75720	114684	64953	611029
3	201580	75768	114039	65149	613385
4	201460	75900	114014	65141	614228
5	201329	75632	114120	65016	614552
6	200928	75592	114187	65094	609339
7	202156	75677	114042	104796	605858
8	201676	75622	114089	65552	616273
9	201820	75624	114068	65529	614119
10	201374	75666	113981	65553	603004

Fig 7.

	Descending Pre-Sort Order (Time in ns)				
Trials	Insertion	Merge	Heap	Quick	RandomQuick
1	397244	54975	98101	1269919	607845
2	397171	54976	97946	1412792	597441
3	397142	54934	98072	1298129	625988
4	398020	54988	97895	1318297	625988
5	398235	55152	97869	1220753	556479
6	397527	55013	97784	1564820	574299
7	397520	55009	97823	1627779	594938
8	397005	55134	97863	1330157	592121
9	397029	54975	97766	1232384	592203
10	397033	54933	97767	1342578	600063

Fig 8.

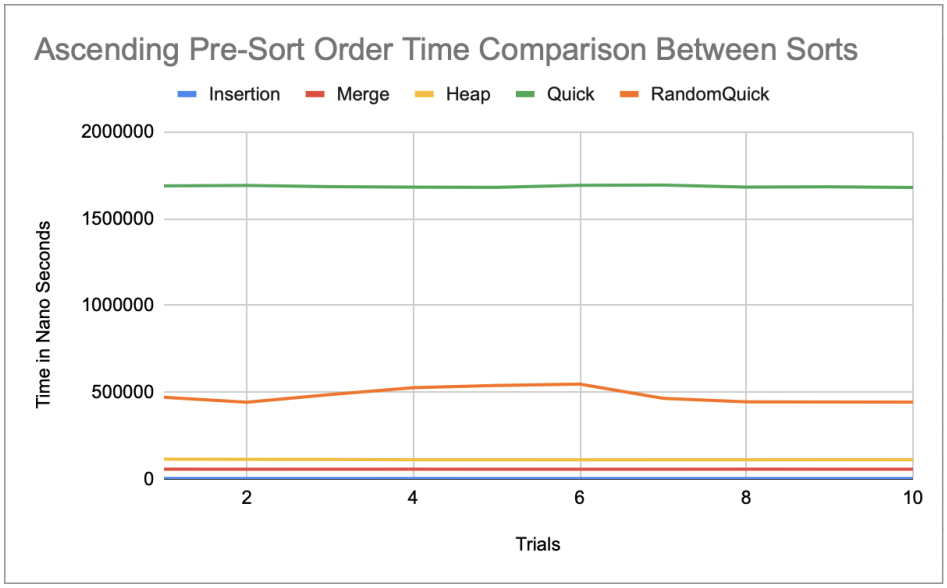

Ascending Pre-Sort Order Time Comparison Between Sorts
Insertion
Merge
Heap
Quick
RandomQuick
2000000
1500000
1000000
500000
0
Time in Nano Seconds
2
4
6
8
10
Trials

Fig 9.

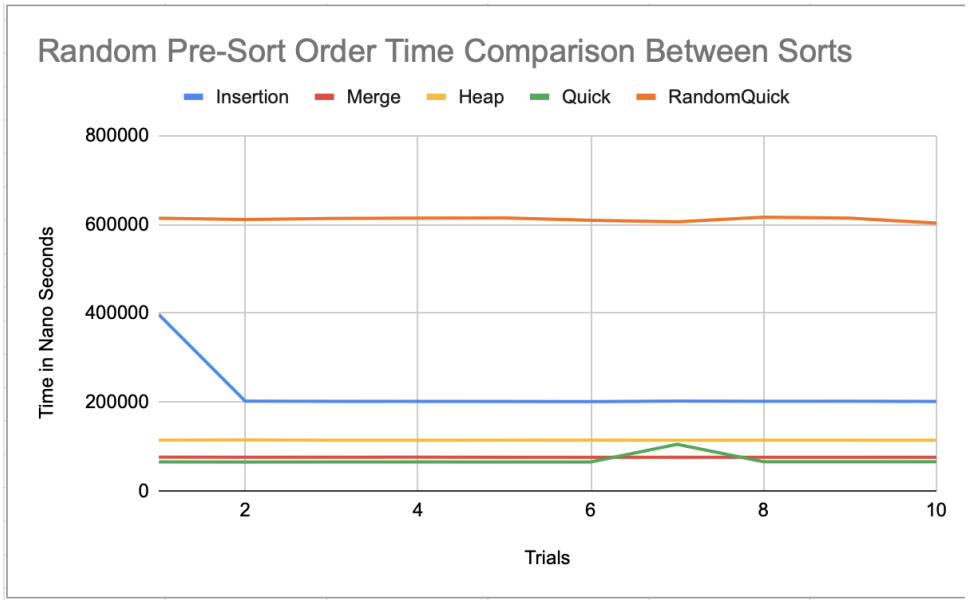

Random Pre-Sort Order Time Comparison Between Sorts
Insertion
Merge
Heap
Quick
RandomQuick
800000
600000
400000
200000
0
Time in Nano Seconds
2
4
6
8
10
Trials

Fig 10.

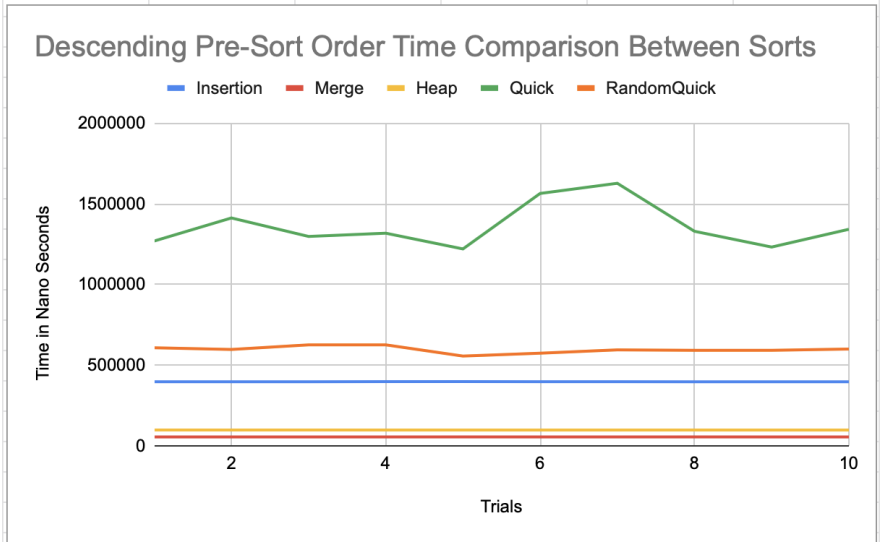


Fig 11.

C++ Source Code: **InsertionSort.cpp**

```
// Name: Connor Marler
// Instructor: Gerry Howser
// Date: 10/13/2022
// Project 2 Randomized Quick Sort Analysis


// Source: https://www.geeksforgeeks.org/quicksort-using-random-pivoting/

#include<iostream>
#include<vector>
#include <chrono>
#include <cassert>

using namespace std;


//Purpose: template function to find the position of pivot element
//PreCondition: takes in a templated function and the starting and ending index
//PostCondition: returns an interger of the pivot that is within the range of the array
//Invariant: once within the if statement and after the swap, arr[j] <= arr[pivot]
template <typename T>
int Partition(T arr[], int start, int end){
  int pivot = end;
 int j = start;

 for(int i=start;i<end;++i){
   if(arr[i]<arr[pivot]){
     swap(arr[i],arr[j]);
     assert(arr[j] <= arr[pivot]);
     ++j;
   }
 }
 swap(arr[j],arr[pivot]);
  return j;
```

```
}


//Purpose: The purpose for this is to randomize the partition in order to mitigate risk
//PreCondition: takes in an array (arr) and two intergers, one is the lower bound and one
//              is the higher bound
//PostCondition: reurns a recursive call of the Partition() function
template <typename T>
int Randomized_Partition(T arr[], int low, int high)
{
  // Generate a random number in between
  // low .. high
  srand(time(NULL));
  int random = low + rand() % (high - low);
   // Swap A[random] with A[high]
  swap(arr[random], arr[high]);
   return Partition(arr, low, high);
}

//Purpose: template function to perform quick sort on array arr
//PreCondition: takes in a templated function and the starting and ending index
//PostCondition: the array is sorted in ascending order
template <typename T>
void Randomized_Quicksort(T arr[], int start, int end ){

 if(start<end){
  int p = Randomized_Partition(arr,start,end);
  Randomized_Quicksort(arr,start,p-1);
  Randomized_Quicksort(arr,p+1,end);
 }
}

//Purpose: Template function to print array
//PreCondition: takes in an array and a value of the size of the array
//PostCondition: outputs the array givens
template<typename T>
void PrintArray(T arr[], int n)
{
  for (int i = 0; i < n; ++i)
      cout << arr[i] << " ";
  cout << "\n\n";
}


//Purpose: take in an array and fill it with a predetermined element set
//PreCondition: must receive arr[size] and a string value determining how to fill the set
//PostCondition: must fill arr[size] with the values in the appropriate as determined by preSortOrder
void populateArray(int *arr, const int size, const string preSortOrder)
{

 // This if else tree could also be re-written as a switch case
 if(preSortOrder == "assPreSortOrder")
 {
  for(int i = 0; i <= size; i++)
    {
      arr[i] = i + 1;
    }
 }
 else if(preSortOrder == "decPreSortOrder")
 {
  for(int j = 0; j <= size; j++)
```

```
    {
      arr[j] = size - j;
    }
  }
  else if(preSortOrder == "randPreSortOrder")
  {
   for(int k = 0; k <= size; k++)
    {
      arr[k] = rand() % size + 1;
    }
  }
  else // error handling
  {
   cout << "ERROR: Something broke. Please quit and try again." << endl;
  }
}

int main()
{

 // These are what we will use for the time measurement
 using chrono::high_resolution_clock;
 using chrono::duration_cast;
 using chrono::duration;
 using chrono::nanoseconds;

 srand (time(NULL));
 const int SIZE = 500;
 string preSortOrder;
 int innerLoop = 1000;
 int outerLoop = 10;
 int averageTime = 0;
 int time = 0;


 int mainArray[SIZE]; // This the hard, original copy of the array
 int workingArray[SIZE]; // This is the copy used to work with and innerLoop through


 cout << "RANDOMIZED QUICKSORT AVG TIMES" << endl;
 for(int i = 0; i <= 2; i ++) // loops through each case
 {
  if(i == 0)
  {
    preSortOrder = "assPreSortOrder";
    cout << endl << "Ascending Order Time:" << endl;
  }
  else if(i == 1)
  {
    preSortOrder = "decPreSortOrder";
    cout << endl << "Descending Order Time:" << endl;
  }
  else if(i == 2)
  {
    preSortOrder = "randPreSortOrder";
    cout << endl << "Random Order Time:" << endl;
  }

  populateArray(mainArray, SIZE, preSortOrder);
  averageTime = 0;
```

```
  for(int k = 0; k < outerLoop; k++) // takes 10 data point
  {
    averageTime = 0;
    for(int j = 0; j < innerLoop; j++) // perform the sort 1000 times and take an average for one data point
    {
      for (int a = 0; a < SIZE; a++) // This copies over the original array onto a working copy
      {
        workingArray[a] = mainArray[a];
      }
      //cout << "Before Sort" << endl;
      //PrintArray(workingArray, SIZE);
      auto time1 = high_resolution_clock::now(); // take initial time
      Randomized_Quicksort(workingArray, 0, SIZE - 1);  // do the sort
      auto time2 = high_resolution_clock::now(); // time the after time
      //cout << "After Sort" << endl;
      //PrintArray(workingArray, SIZE);
      auto nanoSeconds = duration_cast<nanoseconds>(time2 - time1); // find the difference of the times
      time = nanoSeconds.count();  // convert to an int
      averageTime += time;
    }
    averageTime /= innerLoop;
    cout << "Trial # " << k + 1 << ": " << averageTime << " ns" << endl;
  }
 }
}
```

C++ Source Code: **MergeSort.cpp**

```
// Name: Connor Marler
// Instructor: Gerry Howser
// Date: 10/13/2022
// Project 2 Merge Sort Analysis

//Source: https://slaystudy.com/c-program-to-implement-merge-sort-using-templates/

#include<iostream>
//#include<vector>
#include <chrono>
#include <cassert>

using namespace std;


//Purpose: template function to merge 2 componenets of the array, arr
//PreCondition: takes in an templated array and integer start and end points reflecting the specific subarray
//PostCondition: the result is written back to the original array
//Invariant:(indexOfMergedArray == left) && (indexOfSubArrayOne == 0) && (indexOfSubArrayTwo == 0))
template<typename T>
void Merge(T array[], int const left, int const right)
{
  auto mid = (left + right) / 2;
  auto const subArrayOne = mid - left + 1;
  auto const subArrayTwo = right - mid;
  // Create temp arrays
  auto *leftArray = new int[subArrayOne],
       *rightArray = new int[subArrayTwo];
  // Copy data to temp arrays leftArray[] and rightArray[]
  for (auto i = 0; i < subArrayOne; i++)
     leftArray[i] = array[left + i];
  for (auto j = 0; j < subArrayTwo; j++)
     rightArray[j] = array[mid + 1 + j];
```

```
  auto indexOfSubArrayOne
     = 0, // Initial index of first sub-array
     indexOfSubArrayTwo
     = 0; // Initial index of second sub-array
  int indexOfMergedArray
     = left; // Initial index of merged array
  // Merge the temp arrays back into array[left..right]
  assert((indexOfMergedArray == left) && (indexOfSubArrayOne == 0) && (indexOfSubArrayTwo == 0));
  while (indexOfSubArrayOne < subArrayOne && indexOfSubArrayTwo < subArrayTwo)
  {
   if (leftArray[indexOfSubArrayOne] <= rightArray[indexOfSubArrayTwo])
   {
    array[indexOfMergedArray] = leftArray[indexOfSubArrayOne];
    indexOfSubArrayOne++;
   }
   else {
    array[indexOfMergedArray] = rightArray[indexOfSubArrayTwo];
    indexOfSubArrayTwo++;
   }
   indexOfMergedArray++;
  }
  // Copy the remaining elements of
  // left[], if there are any
  while (indexOfSubArrayOne < subArrayOne)
  {
   array[indexOfMergedArray] = leftArray[indexOfSubArrayOne];
   indexOfSubArrayOne++;
   indexOfMergedArray++;
  }
  // Copy the remaining elements of
  // right[], if there are any
  while (indexOfSubArrayTwo < subArrayTwo)
  {
   array[indexOfMergedArray] = rightArray[indexOfSubArrayTwo];
   indexOfSubArrayTwo++;
   indexOfMergedArray++;
  }
  delete[] leftArray;
  delete[] rightArray;
}

//Purpose: template function to perform merge sort on array, arr
//PreCondition: takes in a templated array and integer start and end points reflecting the length of the array
//PostCondition: the list is now sorted in ascending orders
template<typename T>
void MergeSort(T arr[], int start, int end)
{
 if (start < end)
 {
  int mid = (start + end) / 2;
  MergeSort(arr, start, mid); // merge sort the elements in range [start, mid]
  MergeSort(arr, mid + 1, end); // merge sort the elements in range [mid+1, end]
  Merge(arr, start, end);   // merge the above 2 componenets
 }
}


//Purpose: Template function to print array
//PreCondition: takes in an array and a value of the size of the array
//PostCondition: outputs the array givens
template<typename T>
```

```
void PrintArray(T arr[], int n)
{
  for (int i = 0; i < n; ++i)
     cout << arr[i] << " ";
  cout << "\n\n";
}


//Purpose: take in an array and fill it with a predetermined element set
//PreCondition: must receive arr[size] and a string value determining how to fill the set
//PostCondition: must fill arr[size] with the values in the appropriate as determined by preSortOrder
void populateArray(int *arr, const int size, const string preSortOrder)
{

 // This if else tree could also be re-written as a switch case
 if(preSortOrder == "assPreSortOrder")
 {
  for(int i = 0; i <= size; i++)
    {
     arr[i] = i + 1;
    }
 }
 else if(preSortOrder == "decPreSortOrder")
 {
  for(int j = 0; j <= size; j++)
    {
     arr[j] = size - j;
    }
 }
 else if(preSortOrder == "randPreSortOrder")
 {
  for(int k = 0; k <= size; k++)
    {
     arr[k] = rand() % size + 1;
    }
 }
 else // error handling
 {
  cout << "ERROR: Something broke. Please quit and try again." << endl;
 }
}

int main()
{

 // These are what we will use for the time measurement
 using chrono::high_resolution_clock;
 using chrono::duration_cast;
 using chrono::duration;
 using chrono::nanoseconds;

 srand (time(NULL));
 const int SIZE = 500;
 string preSortOrder;
 int innerLoop = 1000;
 int outerLoop = 10;
 int averageTime = 0;
 int time = 0;


 int mainArray[SIZE]; // This the hard, original copy of the array
```

```
int workingArray[SIZE]; // This is the copy used to work with and innerLoop through


cout << "MERGE SORT AVG TIMES" << endl;
for(int i = 0; i <= 2; i ++) // loops through each case
{
 if(i == 0)
 {
  preSortOrder = "assPreSortOrder";
  cout << endl << "Ascending Order Time:" << endl;
 }
 else if(i == 1)
 {
  preSortOrder = "decPreSortOrder";
  cout << endl << "Descending Order Time:" << endl;
 }
 else if(i == 2)
 {
  preSortOrder = "randPreSortOrder";
  cout << endl << "Random Order Time:" << endl;
 }

 populateArray(mainArray, SIZE, preSortOrder);
 averageTime = 0;

 for(int k = 0; k < outerLoop; k++) // takes 10 data point
 {
  averageTime = 0;
  for(int j = 0; j < innerLoop; j++) // perform the sort 1000 times and take an average for one data point
  {
   for (int a = 0; a < SIZE; a++) // This copies over the original array onto a working copy
   {
    workingArray[a] = mainArray[a];
   }
   //cout << "Before Sort" << endl;
   //PrintArray(workingArray, SIZE);
   auto time1 = high_resolution_clock::now(); // take initial time
   MergeSort(workingArray, 0, SIZE - 1);         // do the sort
   auto time2 = high_resolution_clock::now(); // time the after time
   //cout << "After Sort" << endl;
   //PrintArray(workingArray, SIZE);
   auto nanoSeconds = duration_cast<nanoseconds>(time2 - time1); // find the difference of the times
   time = nanoSeconds.count();  // convert to an int
   averageTime += time;
  }
  averageTime /= innerLoop;
  cout << "Trial # " << k + 1 << ": " << averageTime << " ns" << endl;
 }
}
}
```

C++ Source Code: **HeapSort.cpp**

```
// Name: Connor Marler
// Instructor: Gerry Howser
// Date: 10/13/2022
// Project 2 Heap Sort Analysis

#include<iostream>
```

```
#include<vector>
#include <chrono>
#include <cassert>

using namespace std;

//Source: https://www.geeksforgeeks.org/heap-sort/

#include <iostream>
using namespace std;


//Purpose: To heapify a subtree rooted with node i which is an index in arr[].
//PreCondition: takes in a templated array and, an int of the length n and node
//              index of i that is also an interger
//PostCondition: A heap is created
//Invariant: no loop
template<typename T>
void heapify(T arr[], int N, int i)
{
 int largest = i; // Initialize largest as root
 int l = 2 * i + 1; // left = 2*i + 1
 int r = 2 * i + 2; // right = 2*i + 2

  if (l < N && arr[l] > arr[largest]) // If left child is larger than root
   largest = l;

 if (r < N && arr[r] > arr[largest]) // If right child is larger than largest so far
   largest = r;

 if (largest != i) { // If largest is not root
   swap(arr[i], arr[largest]);
   heapify(arr, N, largest);// Recursively heapify the affected sub-tree
 }
}

//Purpose: Main function to do heap sort
//PreCondition: takes in a templated array and an int of the size of the array
//PostCondition: The array taken in is sorted
//Invariant: For all 1 <= i < (N-1), arr[i] >= arr[i-1]
template<typename T>
void heapSort(T arr[], int N)
{
 for (int i = N / 2 - 1; i >= 0; i--) // Build heap (rearrange array)
   heapify(arr, N, i);

 for (int i = N - 1; i > 0; i--) { // One by one extract an element from heap
   swap(arr[0], arr[i]); // Move current root to end
   assert(arr[i] >= arr[i-1]);
   heapify(arr, i, 0); // call max heapify on the reduced heap
 }
}

//Purpose: Template function to print array
//PreCondition: takes in an array and a value of the size of the array
//PostCondition: outputs the array givens
template<typename T>
void PrintArray(T arr[], int n)
{
   for (int i = 0; i < n; ++i)
       cout << arr[i] << " ";
```

```
  cout << "\n\n";
}


//Purpose: take in an array and fill it with a predetermined element set
//PreCondition: must receive arr[size] and a string value determining how to fill the set
//PostCondition: must fill arr[size] with the values in the appropriate as determined by preSortOrder
void populateArray(int *arr, const int size, const string preSortOrder)
{

 // This if else tree could also be re-written as a switch case
 if(preSortOrder == "assPreSortOrder")
 {
   for(int i = 0; i <= size; i++)
     {
       arr[i] = i + 1;
     }
 }
 else if(preSortOrder == "decPreSortOrder")
 {
   for(int j = 0; j <= size; j++)
     {
       arr[j] = size - j;
     }
 }
 else if(preSortOrder == "randPreSortOrder")
 {
   for(int k = 0; k <= size; k++)
     {
       arr[k] = rand() % size + 1;
     }
 }
 else // error handling
 {
   cout << "ERROR: Something broke. Please quit and try again." << endl;
 }
}

int main()
{

 // These are what we will use for the time measurement
 using chrono::high_resolution_clock;
 using chrono::duration_cast;
 using chrono::duration;
 using chrono::nanoseconds;

 srand (time(NULL));
 const int SIZE = 500;
 string preSortOrder;
 int innerLoop = 1000;
 int outerLoop = 10;
 int averageTime = 0;
 int time = 0;


 int mainArray[SIZE]; // This the hard, original copy of the array
 int workingArray[SIZE]; // This is the copy used to work with and innerLoop through


 cout << "HEAP SORT AVG TIMES" << endl;
```

```
for(int i = 0; i <= 2; i ++) // loops through each case
{
 if(i == 0)
 {
  preSortOrder = "assPreSortOrder";
  cout << endl << "Ascending Order Time:" << endl;
 }
 else if(i == 1)
 {
  preSortOrder = "decPreSortOrder";
  cout << endl << "Descending Order Time:" << endl;
 }
 else if(i == 2)
 {
  preSortOrder = "randPreSortOrder";
  cout << endl << "Random Order Time:" << endl;
 }

 populateArray(mainArray, SIZE, preSortOrder);
 averageTime = 0;

 for(int k = 0; k < outerLoop; k++) // takes 10 data point
 {
  averageTime = 0;
  for(int j = 0; j < innerLoop; j++) // perform the sort 1000 times and take an average for one data point
  {
   for (int a = 0; a < SIZE; a++) // This copies over the original array onto a working copy
   {
    workingArray[a] = mainArray[a];
   }
   //cout << "Before Sort" << endl;
   //PrintArray(workingArray, SIZE);
   auto time1 = high_resolution_clock::now(); // take initial time
   heapSort(workingArray, SIZE);  // do the sort
   auto time2 = high_resolution_clock::now(); // time the after time
   //cout << "After Sort" << endl;
   //PrintArray(workingArray, SIZE);
   auto nanoSeconds = duration_cast<nanoseconds>(time2 - time1); // find the difference of the times
   time = nanoSeconds.count();  // convert to an int
   averageTime += time;
  }
  averageTime /= innerLoop;
  cout << "Trial # " << k + 1 << ": " << averageTime << " ns" << endl;
 }
}
}
```

C++ Source Code: **QuickSort.cpp**

```
// Name: Connor Marler
// Instructor: Gerry Howser
// Date: 10/13/2022
// Project 2 Quick Sort Analysis

//Source: https://slaystudy.com/c-program-to-implement-quicksort-using-templates/
```

```
#include<iostream>
#include<vector>
#include <chrono>
#include <cassert>

using namespace std;

//Purpose: template function to find the position of pivot element
//PreCondition: takes in a templated function and the starting and ending index
//PostCondition: returns an interger of the pivot that is within the range of the array
//Invariant: once within the if statement and after the swap, arr[j] <= arr[pivot]
template <typename T>
int Partition(T arr[], int start, int end){
 int pivot = end;
 int j = start;

 for(int i=start;i<end;++i){
  if(arr[i]<arr[pivot]){
   swap(arr[i],arr[j]);
   assert(arr[j] <= arr[pivot]);
   ++j;
  }
 }
 swap(arr[j],arr[pivot]);
 return j;
 }

//Purpose: template function to perform quick sort on array arr
//PreCondition: takes in a templated function and the starting and ending index
//            A must contain n items that can be compared, 1 ≤ P ≤ n, 1 ≤ R ≤ n
//PostCondition: the array is sorted in ascending order
//            A must contain a permutation of the original items, A[1] ≤ A[2] ≤. . . A[n]
template <typename T>
void Quicksort(T arr[], int start, int end ){

 if(start<end){
  int p = Partition(arr,start,end);
  Quicksort(arr,start,p-1);
  Quicksort(arr,p+1,end);
 }
 }

//Purpose: Template function to print array
//PreCondition: takes in an array and a value of the size of the array
//PostCondition: outputs the array givens
template<typename T>
void PrintArray(T arr[], int n)
{
  for (int i = 0; i < n; ++i)
      cout << arr[i] << " ";
  cout << "\n\n";
}


//Purpose: take in an array and fill it with a predetermined element set
//PreCondition: must receive arr[size] and a string value determining how to fill the set
//PostCondition: must fill arr[size] with the values in the appropriate as determined by preSortOrder
void populateArray(int *arr, const int size, const string preSortOrder)
{

 // This if else tree could also be re-written as a switch case
```

```
if(preSortOrder == "assPreSortOrder")
{
  for(int i = 0; i <= size; i++)
    {
      arr[i] = i + 1;
    }
}
else if(preSortOrder == "decPreSortOrder")
{
  for(int j = 0; j <= size; j++)
    {
      arr[j] = size - j;
    }
}
else if(preSortOrder == "randPreSortOrder")
{
  for(int k = 0; k <= size; k++)
    {
      arr[k] = rand() % size + 1;
    }
}
else // error handling
{
  cout << "ERROR: Something broke. Please quit and try again." << endl;
}
}

int main()
{

 // These are what we will use for the time measurement
 using chrono::high_resolution_clock;
 using chrono::duration_cast;
 using chrono::duration;
 using chrono::nanoseconds;

 srand (time(NULL));
 const int SIZE = 500;
 string preSortOrder;
 int innerLoop = 1000;
 int outerLoop = 10;
 int averageTime = 0;
 int time = 0;


 int mainArray[SIZE]; // This the hard, original copy of the array
 int workingArray[SIZE]; // This is the copy used to work with and innerLoop through


 cout << "QUICK SORT AVG TIMES" << endl;
 for(int i = 0; i <= 2; i ++) // loops through each case
 {
  if(i == 0)
  {
    preSortOrder = "assPreSortOrder";
    cout << endl << "Ascending Order Time:" << endl;
  }
  else if(i == 1)
  {
    preSortOrder = "decPreSortOrder";
    cout << endl << "Descending Order Time:" << endl;
```

```
  }
  else if(i == 2)
  {
   preSortOrder = "randPreSortOrder";
   cout << endl << "Random Order Time:" << endl;
  }

  populateArray(mainArray, SIZE, preSortOrder);
  averageTime = 0;

  for(int k = 0; k < outerLoop; k++) // takes 10 data point
  {
   averageTime = 0;
   for(int j = 0; j < innerLoop; j++) // perform the sort 1000 times and take an average for one data point
   {
    for (int a = 0; a < SIZE; a++) // This copies over the original array onto a working copy
    {
     workingArray[a] = mainArray[a];
    }
    //cout << "Before Sort" << endl;
    //PrintArray(workingArray, SIZE);
    auto time1 = high_resolution_clock::now(); // take initial time
    Quicksort(workingArray, 0, SIZE - 1);  // do the sort
    auto time2 = high_resolution_clock::now(); // time the after time
    //cout << "After Sort" << endl;
    //PrintArray(workingArray, SIZE);
    auto nanoSeconds = duration_cast<nanoseconds>(time2 - time1); // find the difference of the times
    time = nanoSeconds.count();  // convert to an int
    averageTime += time;
   }
   averageTime /= innerLoop;
   cout << "Trial # " << k + 1 << ": " << averageTime << " ns" << endl;
  }
 }
}
```

C++ Source Code: **RandomizedQuicksort.cpp**

```
// Name: Connor Marler
// Instructor: Gerry Howser
// Date: 10/13/2022
// Project 2 Randomized Quick Sort Analysis


// Source: https://www.geeksforgeeks.org/quicksort-using-random-pivoting/
```

```
#include<iostream>
#include<vector>
#include <chrono>
#include <cassert>

using namespace std;


//Purpose: template function to find the position of pivot element
//PreCondition: takes in a templated function and the starting and ending index
//PostCondition: returns an interger of the pivot that is within the range of the array
//Invariant: once within the if statement and after the swap, arr[j] <= arr[pivot]
template <typename T>
int Partition(T arr[], int start, int end){
  int pivot = end;
 int j = start;

 for(int i=start;i<end;++i){
   if(arr[i]<arr[pivot]){
     swap(arr[i],arr[j]);
     assert(arr[j] <= arr[pivot]);
     ++j;
   }
 }
 swap(arr[j],arr[pivot]);
  return j;
 }


//Purpose: The purpose for this is to randomize the partition in order to mitigate risk
//PreCondition: takes in an array (arr) and two intergers, one is the lower bound and one
//              is the higher bound
//PostCondition: reurns a recursive call of the Partition() function
template <typename T>
int Randomized_Partition(T arr[], int low, int high)
{
   // Generate a random number in between
   // low .. high
   srand(time(NULL));
   int random = low + rand() % (high - low);
    // Swap A[random] with A[high]
   swap(arr[random], arr[high]);
    return Partition(arr, low, high);
}

//Purpose: template function to perform quick sort on array arr
//PreCondition: takes in a templated function and the starting and ending index
//PostCondition: the array is sorted in ascending order
template <typename T>
void Randomized_Quicksort(T arr[], int start, int end ){

 if(start<end){
   int p = Randomized_Partition(arr,start,end);
   Randomized_Quicksort(arr,start,p-1);
   Randomized_Quicksort(arr,p+1,end);
 }
}

//Purpose: Template function to print array
//PreCondition: takes in an array and a value of the size of the array
//PostCondition: outputs the array givens
```

```
template<typename T>
void PrintArray(T arr[], int n)
{
  for (int i = 0; i < n; ++i)
      cout << arr[i] << " ";
  cout << "\n\n";
}


//Purpose: take in an array and fill it with a predetermined element set
//PreCondition: must receive arr[size] and a string value determining how to fill the set
//PostCondition: must fill arr[size] with the values in the appropriate as determined by preSortOrder
void populateArray(int *arr, const int size, const string preSortOrder)
{

 // This if else tree could also be re-written as a switch case
 if(preSortOrder == "assPreSortOrder")
 {
  for(int i = 0; i <= size; i++)
    {
     arr[i] = i + 1;
    }
 }
 else if(preSortOrder == "decPreSortOrder")
 {
  for(int j = 0; j <= size; j++)
    {
     arr[j] = size - j;
    }
 }
 else if(preSortOrder == "randPreSortOrder")
 {
  for(int k = 0; k <= size; k++)
    {
     arr[k] = rand() % size + 1;
    }
 }
 else // error handling
 {
  cout << "ERROR: Something broke. Please quit and try again." << endl;
 }
}

int main()
{

 // These are what we will use for the time measurement
 using chrono::high_resolution_clock;
 using chrono::duration_cast;
 using chrono::duration;
 using chrono::nanoseconds;

 srand (time(NULL));
 const int SIZE = 500;
 string preSortOrder;
 int innerLoop = 1000;
 int outerLoop = 10;
 int averageTime = 0;
 int time = 0;

 int mainArray[SIZE]; // This the hard, original copy of the array
```

```
int workingArray[SIZE]; // This is the copy used to work with and innerLoop through


cout << "RANDOMIZED QUICKSORT AVG TIMES" << endl;
for(int i = 0; i <= 2; i ++) // loops through each case
{
 if(i == 0)
 {
  preSortOrder = "assPreSortOrder";
  cout << endl << "Ascending Order Time:" << endl;
 }
 else if(i == 1)
 {
  preSortOrder = "decPreSortOrder";
  cout << endl << "Descending Order Time:" << endl;
 }
 else if(i == 2)
 {
  preSortOrder = "randPreSortOrder";
  cout << endl << "Random Order Time:" << endl;
 }

 populateArray(mainArray, SIZE, preSortOrder);
 averageTime = 0;

 for(int k = 0; k < outerLoop; k++) // takes 10 data point
 {
  averageTime = 0;
  for(int j = 0; j < innerLoop; j++) // perform the sort 1000 times and take an average for one data point
  {
   for (int a = 0; a < SIZE; a++) // This copies over the original array onto a working copy
   {
    workingArray[a] = mainArray[a];
   }
   //cout << "Before Sort" << endl;
   //PrintArray(workingArray, SIZE);
   auto time1 = high_resolution_clock::now(); // take initial time
   Randomized_Quicksort(workingArray, 0, SIZE - 1);  // do the sort
   auto time2 = high_resolution_clock::now(); // time the after time
   //cout << "After Sort" << endl;
   //PrintArray(workingArray, SIZE);
   auto nanoSeconds = duration_cast<nanoseconds>(time2 - time1); // find the difference of the times
   time = nanoSeconds.count();  // convert to an int
   averageTime += time;
  }
  averageTime /= innerLoop;
  cout << "Trial # " << k + 1 << ": " << averageTime << " ns" << endl;
 }
}
}
```