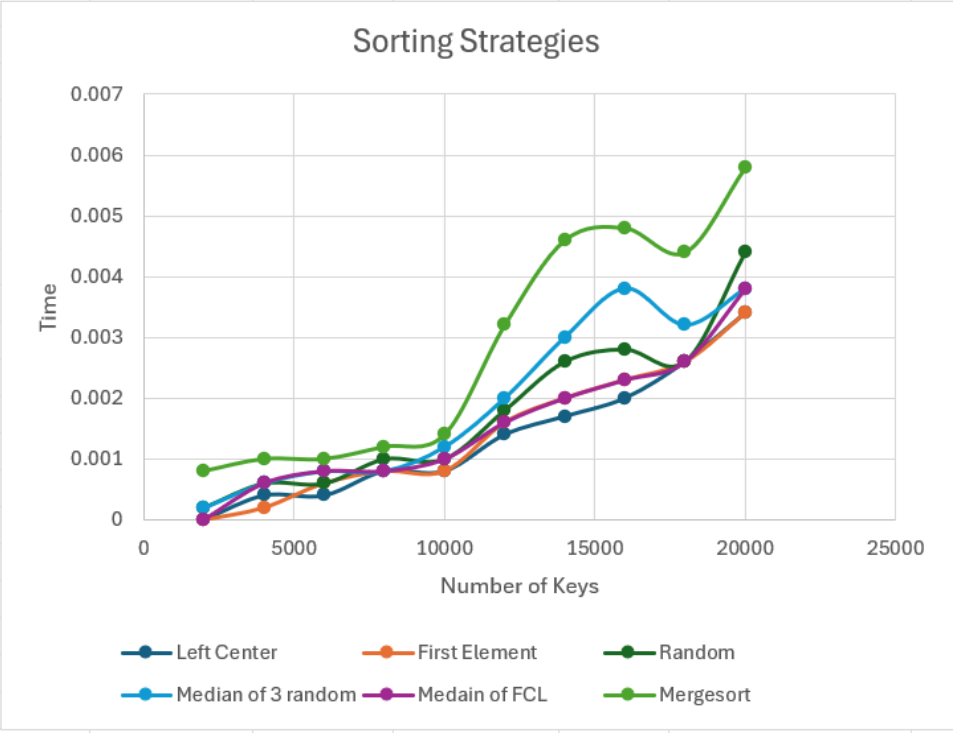
Author 1:	Jonathan Mabery
Email 1:	jmabery@uco.edu
Author 2:	Andrew Aprile
Email 2:	aaprile@uco.edu
Course:	ADS. CMSC3613. CRN10040
Assignment	Project 5. P05

Give a brief discussion of your implementation: just like an explanation of your idea in an interview.
Answer:
In this project,we built a program to compare different sorting strategies. There are 6 total. The first was the left center given in the program. The 2-5 strategies were different pivot strategies. The last one was mergesort. We had to modify main to allow for 6 strategies instead of just 5.

Draw a Graph to compare the performance of different pivot strategies:

Data Number	Left Center	First Element	Random	Median of 3 random	Medain of FCL	Mergesort
2000	0	0	0.0002	0.0002	0	0.0008
4000	0.0004	0.0002	0.0006	0.0006	0.0006	0.001
6000	0.0004	0.0006	0.0006	0.0008	0.0008	0.001
8000	0.0008	0.0008	0.001	0.0008	0.0008	0.0012
10000	0.0008	0.0008	0.001	0.0012	0.001	0.0014
12000	0.0014	0.0016	0.0018	0.002	0.0016	0.0032
14000	0.0017	0.002	0.0026	0.003	0.002	0.0046
16000	0.002	0.0023	0.0028	0.0038	0.0023	0.0048
18000	0.0026	0.0026	0.0026	0.0032	0.0026	0.0044
20000	0.0034	0.0034	0.0044	0.0038	0.0038	0.0058
Outliers changed to the average of one above and one below						



Please also answer the following questions:

1. **Which strategy do you think is the best among the four above? Show your analysis.**
   Merge sort appears to be the best method of all of them. It's very close with the left center strategy.

2. **What's the worst case for quicksort using the first element as the pivot?**
   The worst case time complexity for Quicksort when using the first element as pivot would be O(n^2)

   Because when the input array is already sorted, or reverse sorted, it can cause the pivot to be the maximum or minimum element, which means the array would not be split into equal halves, leading to N recursive calls.

3. **Do you think Quicksort is faster than Insertion Sort? Please briefly justify it.**
   Yes. Quicksort has a time complexity of O(n log(n)). This is due to it breaking up the list into smaller lists that it solves recursively.
   Insertion sort time complexity is O(n^2). This is due to needing to shift elements around to insert into the correct spot.

4. **Some people argue that on average Mergesort could be faster than Quicksort when the input array is big. Do you agree? Please share your thoughts.**
   According to the graph above, Mergesort would likely not be faster on average due to the movement of data and constant resizing of the array using temporary variables. However like mentioned in question 2, mergesort could beat out quicksort if the array is already sorted, or sorted in reverse order, causing the quicksort algorithm to have a time complexity of O(n^2), while Mergesort has a time complexity of O(n log(n)).

5. **Can you propose a better pivot strategy for Quicksort than these given options?**
   Choosing the middle element or the middle between middle and last might be a good option. Lists may be already sorted somewhat so choosing the middle would give you an even number of keys less than it and greater than it.

Source Code:

To Do 1

```
template <class Entry>
void Sortable<Entry>::recursive_quick_sort(int low, int high, int option)
/*
Pre:  low and high are valid positions in the Sortable.
Post: The entries of the Sortable have been
      rearranged so that their keys are sorted into nondecreasing order.
      The pivot option 1: first element as pivot.
      The pivot option 2: randomly choosing the pivot.
      The pivot option 3: choose the median of 3 randomly selected elements as pivot.
      The pivot option 4: choose the median of first, center, and last elements as
pivot.
*/
```

```
{
   // TODO 1: implement quicksort with different pivot strategies

   if (low >= high) {
      return;
   }

   int pivot_position;
   int middle = (low + high) / 2;

   switch (option){
      case 1: {
         //choose the first element as our pivot point
         pivot_position = low;
         break;
      }
      case 2: {
         //choose a random element as our pivot point

         pivot_position = low + rand() % (high - low + 1);
            // the rand() function generates a random integer from 0 to RAND_MAX
(32767 in most systems)
            // add to low to make sure the random number is at least low
            // taking mod from the difference of high and low makes sure the random
number is within the range
         break;
      }
      case 3: {
         // choose the median (middle) of three elements chosen at random as our
pivot point

         // get three random positions
         int r1 = low + rand() % (high - low + 1);
         int r2 = low + rand() % (high - low + 1);
         int r3 = low + rand() % (high - low + 1);

         //get the three entries
         Entry e1 = entry[r1];
         Entry e2 = entry[r2];
         Entry e3 = entry[r3];

         //find median of the entries
         if ((e1 <= e2 && e1 >= e3) ||
             (e1 >= e2 && e1 <= e3)){
```

```
            pivot_position = r1;
         } else if ((e2 <= e1 && e2 >= e3) ||
                    (e2 >= e1 && e2 <= e3)){
            pivot_position = r2;
         } else {
            pivot_position = r3;
         }


         break;
      }
      case 4: {
         // choose the median (middle) of the first, middle, and last elements as our
pivot point
         Entry first = entry[low];
         Entry center = entry[middle];
         Entry last = entry[high];

         if ((first <= center && first >= last) ||
             (first >= center && first <= last)){
            pivot_position = low;
         } else if ((center <= first && center >= last) ||
                    (center >= first && center <= last)){
            pivot_position = middle;
         } else {
            pivot_position = high;
         }
         break;
      }
      default: {
         // should not reach here
         pivot_position = middle;
         break;
      }
   }


   std::swap(entry[middle], entry[pivot_position]); // put pivot at the center
   int new_pivot = partition(low, high);

   recursive_quick_sort(low, new_pivot - 1, option);
   recursive_quick_sort(new_pivot + 1, high, option);
}
```

To Do 2: Merge Sort

```
template <class Entry>
void Sortable<Entry>::merge_sort()
/*
Post: The entries of the Sortable have been rearranged so
      that their keys are sorted into nondecreasing order.
*/
{
   // TODO 2: implement merge_sort
   recursive_merge_sort(0, count-1);
}


template <class Entry>
void Sortable<Entry>::recursive_merge_sort(int low, int high){
   if(low < high){
      int mid = (low + high)/2;
      recursive_merge_sort(low, mid);
      recursive_merge_sort(mid + 1, high);
      merge(low, high);
   }
}


template <class Entry>
void Sortable<Entry>::merge(int low, int high){
   Entry* temp = new Entry[high - low + 1];
   int index = 0;
   int index1 = low;
   int mid = (low + high) / 2;
   int index2 = mid + 1;
   while (index1 <= mid && index2 <= high){
      if (entry[index1] < entry[index2]){
         temp[index] = entry[index1];
         index++;
         index1++;
      } else {
         temp[index] = entry[index2];
         index++;
         index2++;
      }
   }
```

```
    while(index1 <= mid){
        temp[index] = entry[index1];
        index++;
        index1++;
    }
    while(index2 <= high){
        temp[index] = entry[index2];
        index++;
        index2++;
    }
    for(index = low; index <= high; index++){
        entry[index] = temp[index - low];
    }
    delete[] temp;
}
```

To Do 3

Todo3. In Main Function
Modify main to also use Merge_sort() for comparison.
Changed the max option to 6. This produces 6 sorted output files and 6 time files.

```
  const int max_option = 6; // option 0 = left center quick sort from lecture. options 1-

4 = different pivot strategies. option 5 = merge sort
```

Down below in the for loop, we changed the structure to include idx of 5 which chooses merge_sort.

```
118       for (int idx = 0; idx < max_option; idx++) {
            Delegate to agent
119         // TODO 3: modify this part to include merge_sort in the comparison
120
121         Sortable<int> sortable(arr, size);
122         start = clock();
123         if (idx == 0) {
124           sortable.quick_sort();
125         }
126         else if(idx >=1 && idx <=4){
127           sortable.quick_sort(idx);
128         } else if(idx == 5){
129           sortable.merge_sort();
130         }
131         finish = clock();
```

A screenshot of a test run

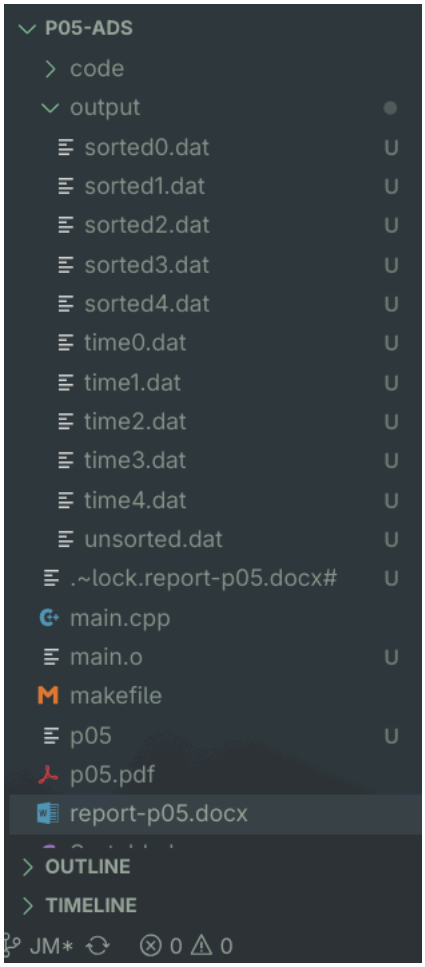
This is our terminal output:

```
p05-ads JM ? ❯ make
rm -f *.o
g++ -c -g main.cpp
g++ -o p05 main.o

p05-ads JM ? ❯ ./p05
Directory exists.

p05-ads JM ? ❯
```

Output File Generation:



Sorted File 0 (all sorted files are sorted properly):