Experiment 1: Quicksort Pivot Selection

- The pivot selection schemes I chose: 1) choose A[hi] as the pivot; 2) choose the middle element in the array as the pivot, which is A[(hi + lo)/2]; The types of inputs are: 1)reverse order input; 2)random input; The sizes I used in experiment 1 are 10000, 20000, 30000, 40000 and 50000.
- The tables and plots of run times are shown below.

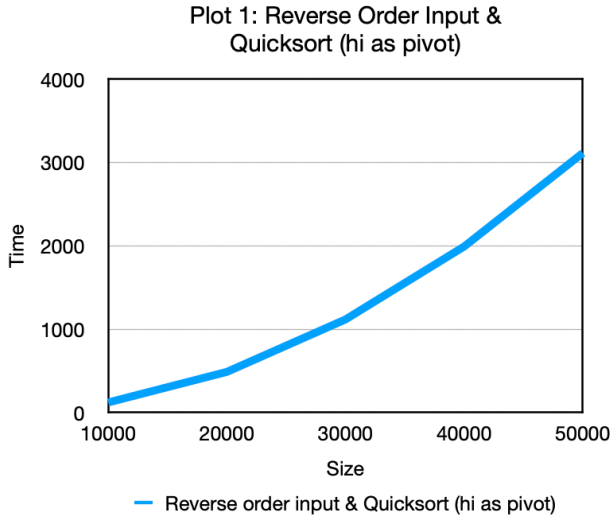
Table 1: Reverse order

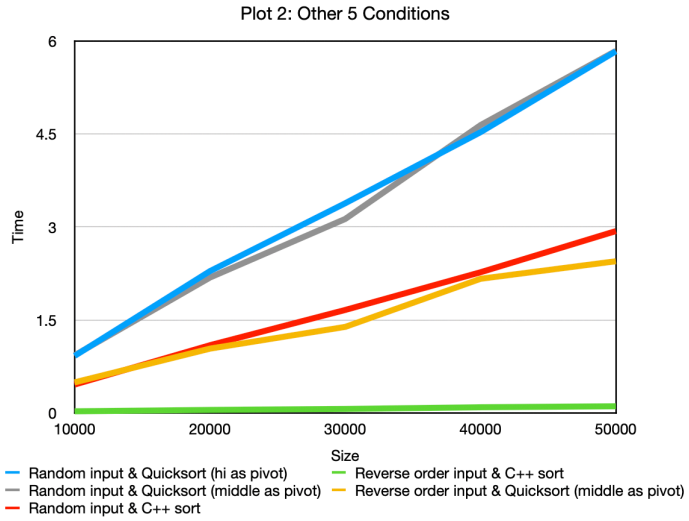
	10000	20000	30000	40000	50000
Quicksort (hi as pivot)	121.728	488.151	1117.6	1992.28	3113.64
Quicksort (middle as pivot)	0.4928	1.036	1.3873	2.1629	2.4459
C++ sort	0.0258	0.048	0.0635	0.0912	0.1072

Table 2: Random order

	10000	20000	30000	40000	50000
Quicksort (hi as pivot)	0.9189	2.2862	3.3817	4.5275	5.8326
Quicksort (middle as pivot)	0.9293	2.1833	3.1284	4.6413	5.8387
C++ sort	0.4571	1.0917	1.6604	2.2685	2.9326

- As the running time of Quicksort (hi as the pivot) using reverse order input is significantly longer than the other 5 conditions, so I plot it individually (as seen in plot 1). Plot 2 shows the running time versus size under the other 5 conditions.





- By observation, the condition "Reverse order input & Quicksort (hi as the pivot)" is extremely time-consuming, and the reason is this is the worst case for Quicksort, and the time complexity is $O(n^2)$, which is pretty large. And the yellow line in Plot 2 shows the best case of Quicksort, for which the time complexity is O(nlog(n)). The C++ sort seems to be more efficient and more stable than Quicksort. As the green line shows, it can even be faster than the best performance of Quicksort. When the input is random, both the two sorting algorithms slow down a bit.
- Purpose of files:

1) exp1.h: contains the Quicksort algorithm for use;
2) test1.cpp: test the running time.

Experiment 2: IntroSort

- The algorithms I chose: 1) Insertion Sort; 2) Standard Quicksort; 3) Modified Quicksort; 4) C++ sort. The types of inputs are: 1)reverse order input; 2)random input; The sizes I used in experiment 2 are 9, 10, 11, 12 for reverse order, and 40, 45, 50, 55 for random order.
- The tables and plots of run times are shown below.

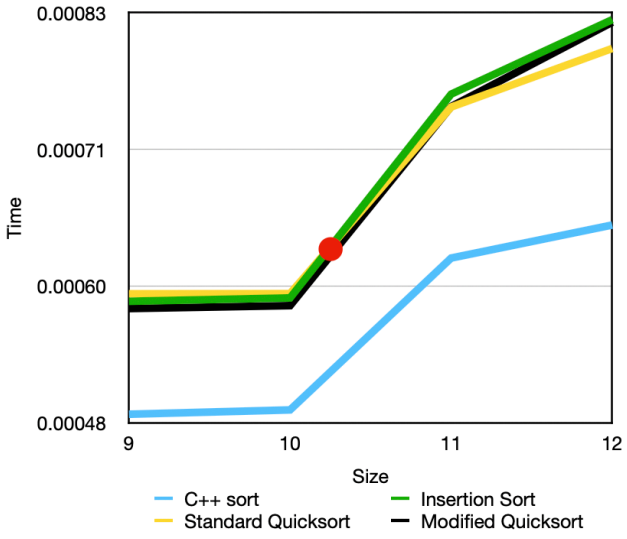
Table 3: Reverse order

	9	10	11	12
Insertion Sort	0.0005837	0.0005866	0.0007605	0.0008241
Standard Quicksort	0.0005901	0.0005903	0.0007495	0.0007995
Modified Quicksort	0.0005775	0.00058	0.0007499	0.0008219
C++ sort	0.0004873	0.000491	0.0006206	0.0006488

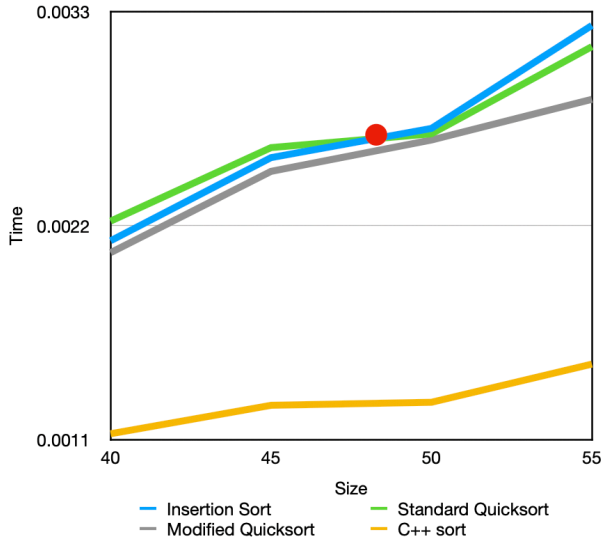
Table 4: Random order

	40	45	50	55
Insertion Sort	0.002122	0.0025493	0.0026996	0.003229
Standard Quicksort	0.002223	0.0026008	0.0026718	0.00312
Modified Quicksort	0.002061	0.0024788	0.00264	0.002849
C++ sort	0.00113	0.0012762	0.0012918	0.001488

Plot 3: Reverse order



Plot 4: Random order



- Marked by the red node in the plots, the S is the x-axis data of the intersections of Insertion Sort curves and Standard Quicksort curves. By doing lots of testing and observation of plot, I finally found **the S for reverse order input is about 10, and for random input, it is approx 50**. When size < S, Insertion Sort is more efficient as its algorithm is much simpler than Standard Quicksort. It doesn't involve recursion while Standard Quicksort does. Also, the number of variables is very less in Insertion Sort, and it spares a lot of time on creating new variables. When size > S, especially when size >> S, Quicksort shows an advantage as its time complexity is closer and closer to O(nlog(n)) when size grows. And the C++ sort still shows the highest efficiency as it does in experiment 1.
- Purpose of files:

1) exp2.h: contains the algorithms (Insertion Sort, Standard Quicksort, Modified Quicksort) for use;
2) test2.cpp: test the running time.