**Experiment #1**

For this portion of the research, I chose a list size of 100 (X = 100) as my base number. I ran the programs and timers 10 times each and then averaged the times out to see the difference. The results are below:

X = 100

- Insertion Sort: AVG 3,074,449 NS (nano-seconds)
- Quick Sort: AVG 325,037 NS
- Difference: 2,749,412 NS

X = 200

- Insertion Sort: AVG 10,062,366 NS
- Quick Sort: AVG 543,521 NS
- Difference: 9,518,845 NS

X = 300

- Insertion Sort: AVG 22,843,833 NS
- Quick Sort: AVG 753,220.9NS
- Difference: AVG 22,090,613 NS

As this experiment was only done ten times for each respective list size, the results show that, on average, quick sort is faster than insertion sort at each level. Also, as the list gets bigger, quick sort does better and better. I also tried it on huge lists (1000000+) and found the same results. Since we know that insertion sort does better with small lists, these large list sizes that utilize a short quick sort time make a lot of sense. Also, the fact that the lists get bigger and the quick sort does even better supports this.

| X = 1000 | Insertion Sort (Nano-Seconds) | Quick Sort (Nano-Seconds) |
|---|---|---|
| 1 | 2853042 | 364376 |
| 2 | 2993208 | 314625 |
| 3 | 2843458 | 320875 |
| 4 | 3364000 | 308291 |
| 5 | 2943791 | 316667 |
| 6 | 2846292 | 320416 |
| 7 | 2988667 | 319292 |
| 8 | 2912541 | 312208 |
| 9 | 2877125 | 318458 |
| 10 | 4122375 | 355166 |

| X = 2000 | Insertion Sort (Nano-Seconds) | Quick Sort (Nano-Seconds) |
|---|---|---|
| 1 | 11371750 | 563959 |
| 2 | 10631083 | 563875 |
| 3 | 11091333 | 571833 |
| 4 | 11174250 | 594292 |
| 5 | 11745208 | 608333 |
| 6 | 11526708 | 707542 |
| 7 | 11233042 | 589750 |
| 8 | 10799333 | 582084 |
| 9 | 11935667 | 585958 |
| 10 | 10688000 | 597584 |

| X = 3000 | Insertion Sort (Nano-Seconds) | Quick Sort (Nano-Seconds) |
|---|---|---|
| 1 | 22245583 | 920042 |
| 2 | 23899958 | 743458 |
| 3 | 22857459 | 802000 |
| 4 | 22276500 | 780958 |
| 5 | 23299542 | 778209 |
| 6 | 22806208 | 821042 |
| 7 | 22982958 | 380584 |
| 8 | 22710750 | 764208 |
| 9 | 22377292 | 778750 |
| 10 | 22982083 | 762958 |

**Experiment #2**

For this experiment, I took the random generated lists formula and then sorted them before running them through the sorting methods once more and timed it. The averages are below:

Insertion Sort: AVG 149,249 NS
Quick Sort: AVG 107,219,666 NS
Difference: AVG107,070,417 NS

| | Insertion Sort (NanoSeconds) | Quick Sort (NanoSeconds) |
|---|---|---|
| 1 | 15500 | 11501125 |
| 2 | 14875 | 10611917 |
| 3 | 14792 | 11009125 |
| 4 | 14792 | 10073459 |
| 5 | 14750 | 10949250 |
| 6 | 14875 | 10642375 |
| 7 | 14875 | 10407083 |
| 8 | 14916 | 10784708 |
| 9 | 14916 | 10582708 |
| 10 | 14958 | 10657916 |

From this data, it's clear that on a sorted list, insert sort is much, much faster than quick sort. This is interesting especially because insert sort on the un-sorted lists was slower every single time. However, it makes sense, as Insertion sorts best case run time is faster than quick sort's (constant vs log(n)). In class, we talked about how insertion sorts best case runtime is constant with a list that is already sorted, so this falls in line with what we've been taught. Another important point to note is that quick sort will be very slow if the pivot point is an extreme value compared to the rest of the list. Since the values are sorted, the pivot point with no random attribute will always be almost as extreme as possible, further slowing quick sort down. Also, something interesting is that the timings for insertion sort are incredibly close to each other, more so than the other experiments.

**Experiment #3**
For this experiment, I conducted it the same way as above but I reversed the list that was previously sorted. The averages are below:

Insertion Sort: AVG 53,801,753 NS
Quick Sort: AVG 62,743,500 NS
Difference: AVG 8,941,747 NS

| | Insertion Sort (NanoSeconds) | Quick Sort (NanoSeconds) |
|---|---|---|
| 1 | 6455458 | 7285958 |
| 2 | 5725709 | 6489792 |
| 3 | 2775417 | 3777750 |
| 4 | 5719959 | 6420750 |
| 5 | 5735292 | 6404750 |
| 6 | 5311083 | 6426833 |
| 7 | 5741084 | 6421791 |
| 8 | 4729417 | 6448084 |
| 9 | 5863000 | 6686375 |
| 10 | 5745334 | 6381417 |

Based off of the data, in this reversed list format, the quick sort method is marginally slower than the insertion sort method. Again, the problem with the extreme value pivot point comes into play here, justifying the slow speed of quick sort. Also, a reverse sorted list represents insertion sorts worst case run time (O(n^2)) as it will have to most every single element without a doubt, so it makes sense that insertion sort is slower as well.

**Experiment #4**
For this experiment, I took the randomly generated lists, sorted them, and then using the random_shuffle method in c++, used a for-loop to shuffle the elements in groups of ten so that each element was at most 10 spots away from its original spot. This created a partially sorted list. The results are below:

Insertion Sort: AVG 76,620.7 NS
Quick Sort: AVG 302,437.4 NS
Difference: AVG 225,816.7 NS

| X = 1000 | Insertion Sort (NanoSeconds) | Quick Sort (NanoSconds) |
|---|---|---|
| 1 | 99333 | 420833 |
| 2 | 78500 | 322041 |
| 3 | 77500 | 346333 |
| 4 | 79875 | 322958 |
| 5 | 77125 | 333375 |
| 6 | 77125 | 326584 |
| 7 | 46625 | 213542 |
| 8 | 76875 | 348625 |
| 9 | 76208 | 345292 |
| 10 | 77041 | 334791 |

From these results, we can see that, on average, insertion sort was much faster than quick sort. This is because (1) we used a relatively small input for the lists, so insertion sort will normally run faster than quick sort and (2) a partially sorted list is closer to insertion sorts best-case scenario than it is to quick sorts.

**Experiment #5**

For this experiment, I changed the minsize variable that I was working with for the quick sort methods so as to observe the effects of a “hybrid” quick sort rather than a “pure” one. I tried 3 different minsizes: 50, 25, and 5. The results are below:

Minsize = 50: AVG 126,649.9 NS
Minsize = 25: AVG 168,287.5 NS
Minsize = 5: AVG 205,937.5 NS

| Minsize = 50, X = 1000 | Quick Sort (Nano Seconds) |
|---|---|
| 1 | 125125 |

| | |
|---|---|
| 2 | 135417 |
| 3 | 130208 |
| 4 | 153791 |
| 5 | 114167 |
| 6 | 121666 |
| 7 | 120167 |
| 8 | 108000 |
| 9 | 139958 |
| 10 | 118000 |

| Minsize = 25, X = 1000 | Quick Sort (Nano Seconds) |
|---|---|
| 1 | 152542 |
| 2 | 168041 |
| 3 | 176833 |
| 4 | 129250 |
| 5 | 161875 |
| 6 | 173917 |
| 7 | 168667 |
| 8 | 178000 |
| 9 | 191584 |
| 10 | 182166 |

| Minsize = 5, X = 1000 | Quick Sort (Nano Seconds) |
|---|---|
| 1 | 161375 |
| 2 | 204541 |

| | |
|---|---|
| 3 | 200875 |
| 4 | 239000 |
| 5 | 207875 |
| 6 | 211041 |
| 7 | 183875 |
| 8 | 196917 |
| 9 | 238417 |
| 10 | 215459 |

After all the other experiments, this obviously makes tons of sense. Since we know that insertion sort is better suited for smaller lists and that quick sort is better suited for larger lists, a larger minsize net will allow the algorithm to outsource the smaller work to insertion sort. The overall effect on the efficiency of the quick sort method allows it to deal with smaller lists without sacrificing time.