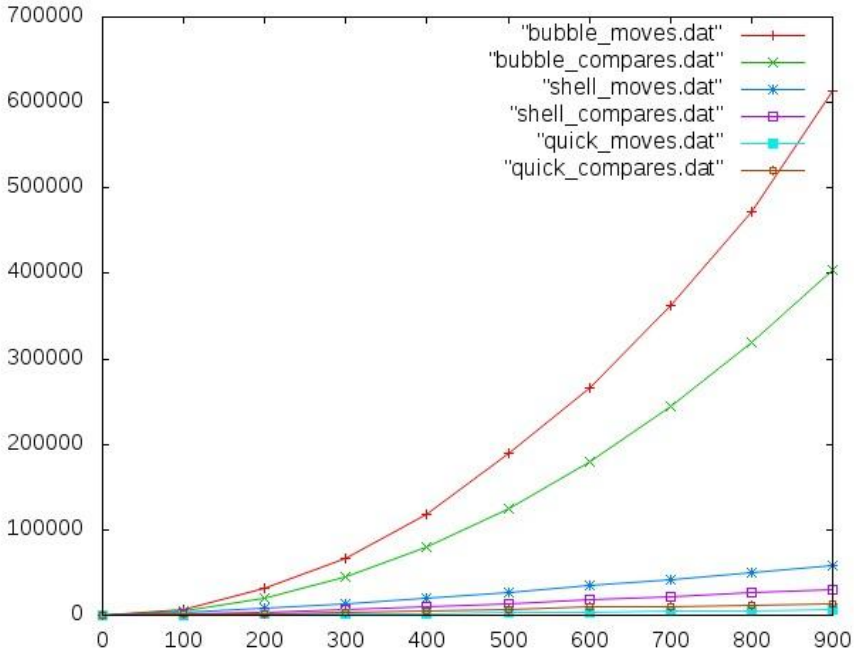
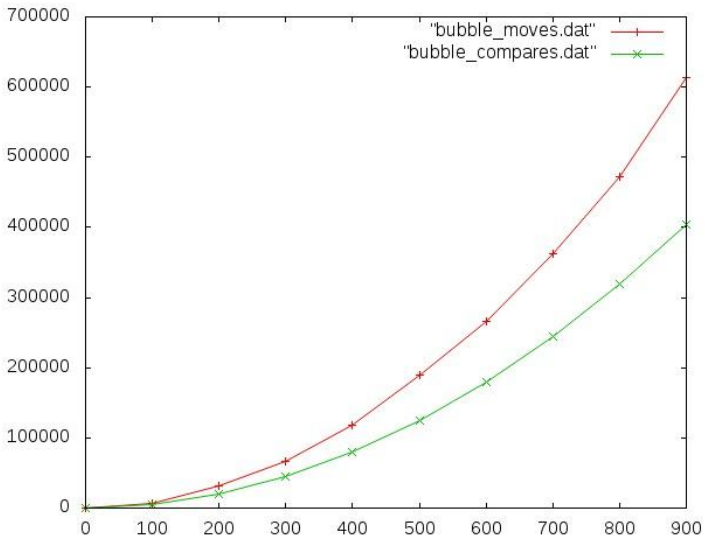
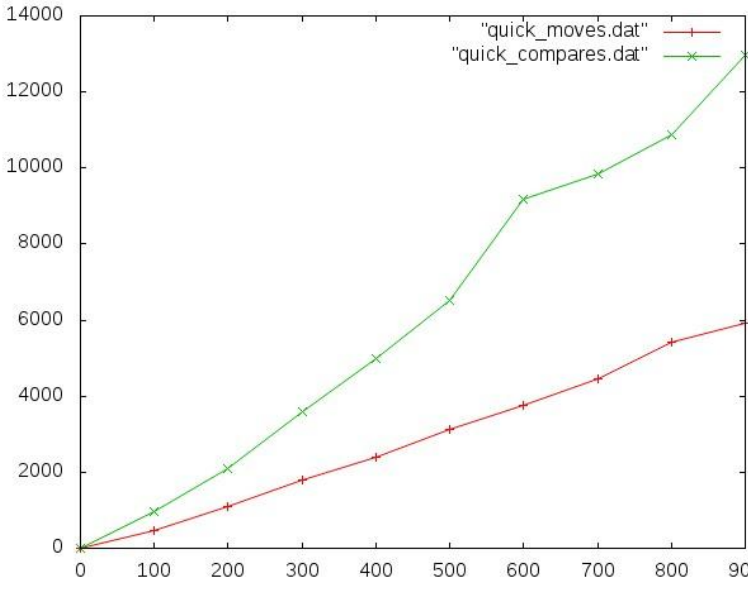
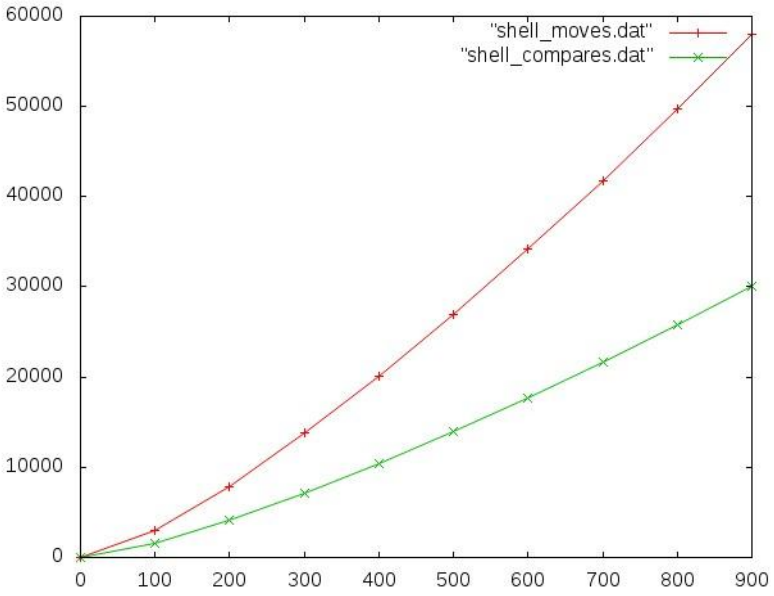

"bubble_moves.dat"
"bubble_compares.dat"
"shell_moves.dat"
"shell_compares.dat"
"quick_moves.dat"
"quick_compares.dat"
700000
600000
500000
400000
300000
200000
100000
0
0
100
200
300
400
500
600
700
800
900


"bubble_moves.dat"
"bubble_compares.dat"
700000
600000
500000
400000
300000
200000
100000
0
0
100
200
300
400
500
600
700
800
900


"quick_moves.dat"
"quick_compares.dat"
14000
12000
10000
8000
6000
4000
2000
0
0
100
200
300
400
500
600
700
800


"shell_moves.dat"
"shell_compares.dat"
60000
50000
40000
30000
20000
10000
0
0
100
200
300
400
500
600
700
800
900

What I learned from different sorting algorithms + analyze graph:
Based on the graph and results from my program, I found out that in an average and usual situation, quick sort is the quickest and least moves + compares sort(meaning most efficient). In contrast, bubble sort is the slowest sort with most moves + compares. Shell sort is in the middle, quicker than bubble sort but slower than quick sort.

Usually more elements will cause bigger time efficiency. As the graph shows, quick sort is much faster during 1000 elements, but less faster during 10 elements.

However, each sort has its own worst case. For example, the worst case for bubble sort is reverse order. I talked about the worst case for these 3 sorts in design.pdf, feel free to check it. There is a huge difference between worst case run complexity and average complexity. For example, the base case performance for quick sort is o(n log n), but the worst case performance is O(n^2).

Time complexity:
Using Big O notation to analyze time complexity, it needs to remove all constant factors so that the running time is estimated in relation to N.

How big do the stack and queue get? Does the size relate to the input? How?
Size is highly related to the input(size will get larger if n is getting larger). Size will get larger when push/enqueue recursion runs (push and enqueue lo and hi until lo = hi), the size is maximum during the last recursion.

Bubble sort:

Bubble Sort in Python

```
1  def bubble_sort(arr):
2      n = len(arr)
3      swapped = True
4      while swapped:
5          swapped = False
6          for i in range(1, n):
7              if arr[i] < arr[i - 1]:
8                  arr[i], arr[i - 1] = arr[i - 1], arr[i]
9                  swapped = True
10         n -= 1
```

Time complexity:
In LIne 4 and Line 6, it represents On because the running time of the loop is directly proportional to N, it will run N times if it has n number.

The rest of the lines are O1(constant), because it will only run 1 time no matter how N values will change.
It will be O(1+1+n(1+n(1+1+1)=1)

Since On^2(lne 4 and line 6) already dominant the constants running time, so the average running time for bubble sort will be **On^2**

Worst case: reverse order(has to compare and move every elements)

Shell sort:

Shell Sort in Python

```
1 def shell_sort(arr):
2     for gap in gaps:
3         for i in range(gap, len(arr)):
4             j = i
5             temp = arr[i]
6             while j >= gap and temp < arr[j - gap]:
7                 arr[j] = arr[j - gap]
8                 j -= gap
9             arr[j] = temp
```

Time Complexity:
Line 3 and Line 6: Both represent On, because for loop will be affected by N, because if array length is longer, then runtime will be slower.

The rest of the lines are O1, because they are all statements, won't be affected by N.

Average time complexity:
O(1+(n+1+1+n(1+1)+1)
= O(n^2)

Worse case: O(n*log^2 n)

If the gap is 1(meaning split whole data to 1 group and sort), then it will be the worst case(you will find out that, it's actually an insertion sort). So different gaps will have different performance based on the data. However, if we improve our gap based on the data(for example, make sure gaps are decreasing until 1), it will be much more efficient.

Quicksort:

Partition in Python

```
1  def partition(arr, lo, hi):
2      pivot = arr[lo + ((hi - lo) // 2)]; # Prevent overflow.
3      i = lo - 1
4      j = hi + 1
5      while i < j:
6          i += 1 # You may want to use a do-while loop.
7          while arr[i] < pivot:
8              i += 1
9          j -= 1
10         while arr[j] > pivot:
11             j -= 1
12         if i < j:
13             arr[i], arr[j] = arr[j], arr[i]
14     return j
```

```
1  def quick_sort_stack(arr):
2      lo = 0
3      hi = len(arr) - 1
4      stack = []
5      stack.append(lo) # Pushes to the top.
6      stack.append(hi)
7      while len(stack) != 0:
8          hi = stack.pop() # Pops off the top.
9          lo = stack.pop()
10         p = partition(arr, lo, hi)
11         if lo < p:
12             stack.append(lo)
13             stack.append(p)
14         if hi > p + 1:
15             stack.append(p + 1)
16             stack.append(hi)
```

Quicksort in Python with a queue

```
1  def quick_sort_queue(arr):
2      lo = 0
3      hi = len(arr) - 1
4      queue = []
5      queue.append(lo) # Enqueues at the head.
6      queue.append(hi)
7      while len(queue) != 0:
8          lo = queue.pop(0) # Dequeues from the tail.
9          hi = queue.pop(0)
10         p = partition(arr, lo, hi)
11         if lo < p:
12             queue.append(lo)
13             queue.append(p)
14         if hi > p + 1:
15             queue.append(p + 1)
16             queue.append(hi)
```

Time complexity:

The run time complexity for partition part should be O(log n), because the running time of the algorithm is directly related to N (which can divided by 2 low and high), algorithm divides the working area in half with each iteration

Both the rest of the lines, stack and queue part are statements(O1). But in line9, it's O(n), because it repeats iteration N times to call partition at line 9. **N * logn(call partition n times)**

So the formula should be **O(nlog(n))**

Worst case: Usually, quicksort is a fast sort, however, if the pivot is the smallest number in the array, like 0, then it doesn't help at all to sort. In contrast, if the pivot is 0, it wastes one round to compare and move and slows down the run time.