1. So far, we have looked at simple sorts consisting of nested loops. The \( n \times (n-1)/2 \) is \( O(n^2) \). Consider using a min-heap to sort a list. (methods: `BinHeap()`, `insert(item)`, `delMin()`, `isEmpty()`, `size()`) 

   a) If we insert all of the list elements into a min-heap, what would we easily be able to determine?

General idea of Heap sort:

1. Create an empty heap
   \[ \text{myHeap} = \text{BinHeap()} \in O(1) \]
2. Insert all \( n \) list items into heap
   \[ \text{for item in myList: } \text{myHeap.insert(item)} \in O(n \log n) \]
3. delMin heap items back to list in sorted order
   \[ \text{for index in range(len(myList))}{}, \text{myList[index]} = \text{myHeap.delMin()} \in n \log n \]

b) What is the overall \( O(\cdot) \) for heap sort?

\( O(n \log n) \)

2. Another way to do better than the simple sorts is to employ divide-and-conquer (e.g., Merge sort and Quick Sort). Recall the idea of Divide-and-Conquer algorithms. Solve a problem by:

- dividing problem into smaller problem(s) of the same kind
- solving the smaller problem(s) recursively
- use the solution(s) to the smaller problem(s) to solve the original problem

In general, a problem can be solved recursively if it can be broken down into smaller problems that are identical in structure to the original problem.

a) What determines the “size” of a sorting problem? \( \text{size of list} \in n \)

b) How might we break the original problem down into smaller problems that are identical?

split in half

c) What base case(s) (i.e., trivial, non-recursive case(s)) might we encounter with recursive sorts?

\( O(1) \) items in list

d) How do you combine the answers to the smaller problems to solve the original sorting problem?

\[ \frac{50^2}{2} + \frac{50^2}{2} = 2500 \]

\[ \frac{25^2}{2} - \frac{25^2}{2} + \frac{25^2}{2} + \frac{25^2}{2} = 2 \times 25^2 = 1250 \]

\[ \frac{5^2}{2} + \frac{5^2}{2} = 25 \]

\[ \frac{5^2}{2} = 12 \frac{1}{2} \]

\[ \frac{25^2}{2} + \frac{25^2}{2} + \frac{25^2}{2} + \frac{25^2}{2} = 2 \times 25^2 = 1250 \]

\[ \frac{25^2}{2} \]
3. The general idea of merge sort is as follows. Assume "n" items to sort.
   - Split the unsorted part in half to get two smaller sorting problems of about equal size = n/2
   - Solve both smaller problems recursively using merge sort
   - "Merge" the solutions to the smaller problems together to solve the original sorting problem of size n

a) Fill in the merged Sorted Part in the diagram.

b) Describe how you filled in the sorted part in the above example?

4. Merge sort is substantially faster than the simple sorts. Let's analyze the number of comparisons and moves of merge sort. Assume "n" items to sort.

a) On each level of the above diagram write the WORST-CASE number of comparisons and moves for that level.

b) What is the WORST-CASE total number of comparisons and moves for the whole algorithm (i.e., add all levels)?

c) What is the big-oh for worst-case?
5. Quick sort general idea is as follows:
   - Select a "random" item in the unsorted part as the pivot
   - Rearrange (partitioning) the unsorted items such that:
   - Quick sort the unsorted part to the left of the pivot
   - Quick sort the unsorted part to the right of the pivot

   a) Given the following partition function which returns the index of the pivot after this rearrangement, complete the recursive quicksortHelper function.

   ```python
   def partition(lyst, left, right):
       middle = (left + right) // 2
       pivot = lyst[middle]
       lyst[middle] = lyst[right]
       lyst[right] = pivot
       boundary = left
       for i in range(left, right):
           if lyst[i] < pivot:
               temp = lyst[i]
               lyst[i] = lyst[boundary]
               lyst[boundary] = temp
               boundary += 1
       temp = lyst[boundary]
       lyst[boundary] = lyst[right]
       lyst[right] = temp
       return boundary
   
   def quicksort(lyst):
       quicksortHelper(lyst, 0, len(lyst) - 1)
   
   def quicksortHelper(lyst, left, right):
       if left < right:
           pivotIndex = partition(lyst, left, right)
           quicksortHelper(lyst, left, pivotIndex - 1)
           quicksortHelper(lyst, pivotIndex + 1, right)
       
   quicksort(lyst)
   ```

   b) For the list below, trace the first call to partition and determine the resulting list, and value returned.

   ```plaintext
   lyst: 34 26 17 50 23 54 41 55 20
   left 0 8 6 7 8 5 0 0 0 0
   right 8 8 7 7 7 5 0 0 0 0
   index 0 1 1 0 0 0 0 0 0 0
   boundary 0 1 1 0 0 0 0 0 0 0
   pivot 50 50 50 50 50 50 50 50 50 50
   
   b) What initial arrangement of the list would cause partition to perform the most amount of work?

   - If pivot is largest item < (\( \frac{3n}{\max} \)) \( \frac{\max}{n - 1} \) comparisons

   c) Let "n" be the number of items between left and right. What is the worst-case \( O(\ ) \) for partition?

   \( O(n) \)
d) What would be the overall, worst-case $O(\cdot)$ for Quick Sort?

\[ O(n) \]

\[ O(n^2) \]

\[ O(n \log n) \]

e) Ideally, the pivot item splits the list into two equal size problems. What would be the big-oh for Quick Sort in the best case?

\[ n \]

\[ \frac{n}{2} \]

\[ \frac{n}{4} \]

\[ \frac{n}{8} \]

\[ \frac{n}{16} \]

\[ \log_2 n \]

\[ O(n \log n) \]

f) What would be the big-oh for Quick Sort in the average case?

\[ O(n \log n) \]

g) The textbook's partition code (Listing 5.15 on page 225) selects the first item in the list as the pivot item. However, the above partition code selects the middle item of the list to be the pivot. What advantage does selecting the middle item as the pivot have over selecting the first item as the pivot?

In books code, if list already sorted, then selecting pivot from first item cause worst case $O(n^2)$ performance, so $n$ levels and $O(n \log n)$ performance.