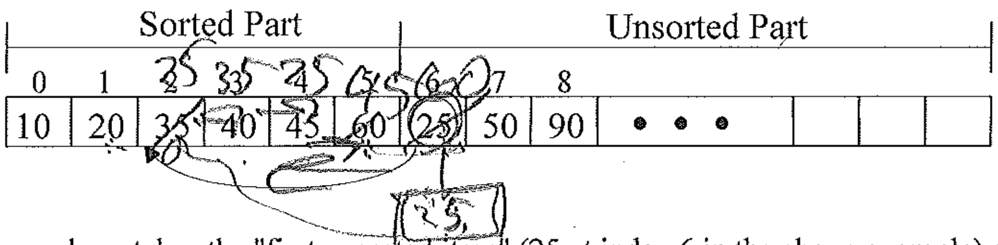
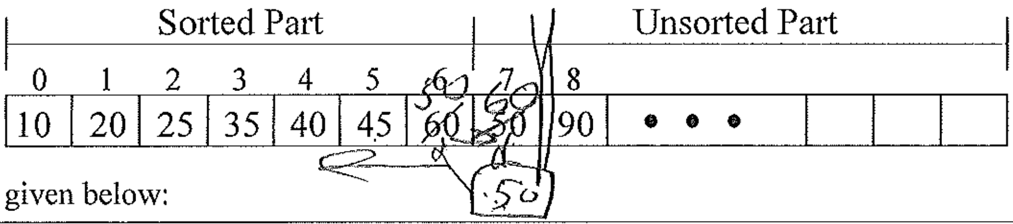
4. Another simple sort is called insertion sort. Recall that in a simple sort:

- the outer loop keeps track of the dividing line between the sorted and unsorted part with the sorted part growing by one in size each iteration of the outer loop.
  - the inner loop's job is to do the work to extend the sorted part's size by one.

After several iterations of insertion sort's outer loop, a list might look like:



In insertion sort the inner-loop takes the "first unsorted item" (25 at index 6 in the above example) and "inserts" it into the sorted part of the list "at the correct spot." After 25 is inserted into the sorted part, the list would look like:



Code for insertion is given below:

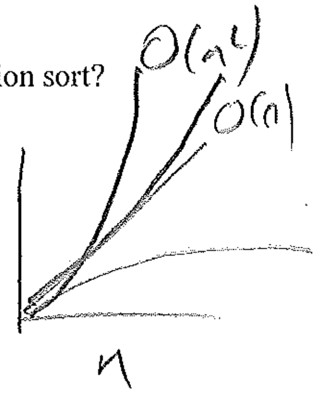
```
def insertionSort(myList):
    """Rearranges the items in myList so they are in ascending order"""

    for firstUnsortedIndex in range(1,len(myList)):
        itemToInsert = myList[firstUnsortedIndex]

        testIndex = firstUnsortedIndex - 1

        while testIndex >= 0 and myList[testIndex] > itemToInsert:
            myList[testIndex+1] = myList[testIndex]
            testIndex = testIndex - 1

        # Insert the itemToInsert at the correct spot
        myList[testIndex + 1] = itemToInsert
```

a) What is the purpose of the `testIndex >= 0` while-loop comparison?

Make sure we have run off left end of list

b) What initial arrangement of items causes the is the overall worst-case performance of insertion sort?

descending order initially



c) What is the worst-case $O()$ notation for the number of item moves?

$O(n^2)$ (see next page)

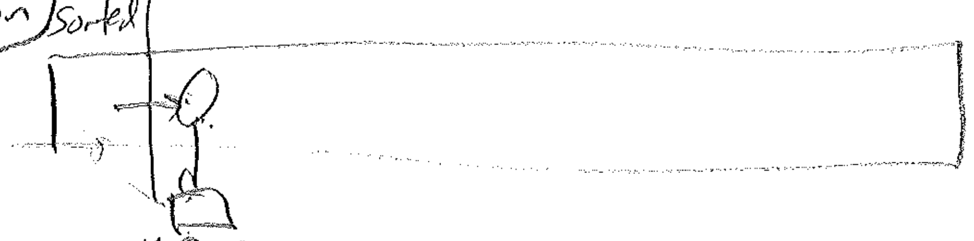
d) What is the worst-case $O()$ notation for the number of item comparisons? $O(n^2)$

e) What initial arrangement of items causes the is the overall best-case performance of insertion sort?

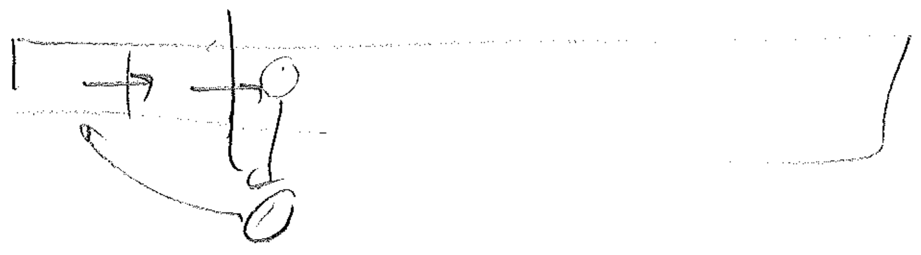
$O(n)$ if already in ascending order

f) What is the best-case $O()$ notation for insertion sort?

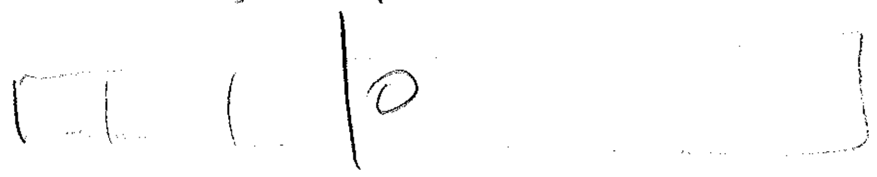
Insertion Sort



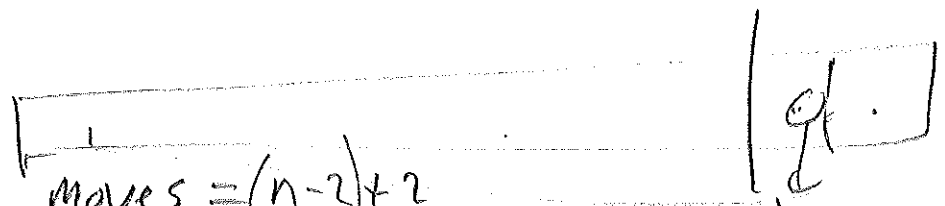
moves = 1+2



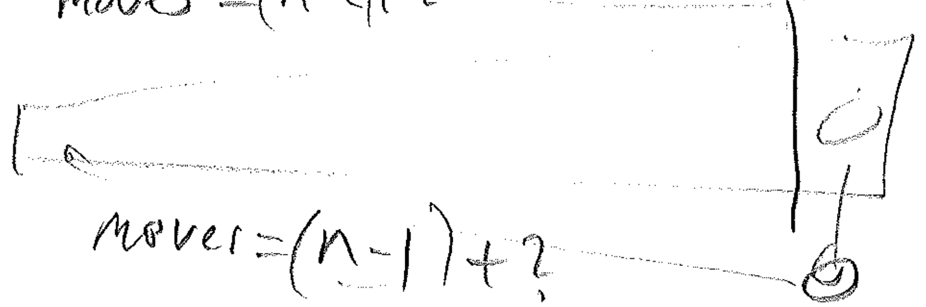
moves = 2+2



Moves = 3+2



moves = (n-2)+2



moves = (n-1)+?

$$(1+2+3+\ \ldots\ +(n-2)+(n-1)) + 2\times(n-1)$$

$$n$$

$$n\times\frac{(n-1)}{2} + 2\times(n-1) \quad O(n^2)$$ moves compares

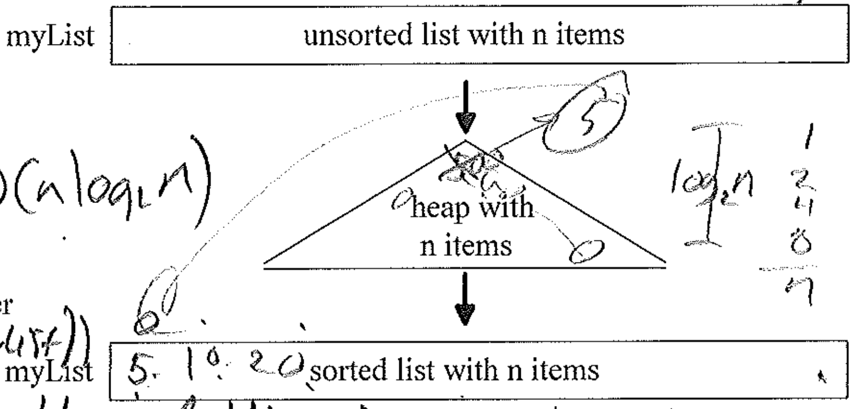
Name: $\frac{n^2}{2}$

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

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

General idea of Heap sort:

1. Create an empty heap $O(1)$
   myHeap = BinHeap()
2. Insert all n list items into heap
   for item in myList $O(n \log_2 n)$
       myHeap.insert(item)
3. delMin heap items back to list in sorted order
   for index in range(len(myList))
       myList[index] = myHeap.delMin() $O(n \log_2 n)$



b) What is the overall $O(\,)$ for heap sort? $O(n \log_2 n)$

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:

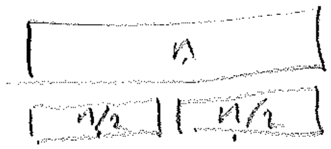
- dividing problem into smaller problem(s) of the same kind
- solving the smaller problem(s) recursively
- use the solution(s) to the smaller problem(s) to solve the original problem

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? size of list being sorted = "n"

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

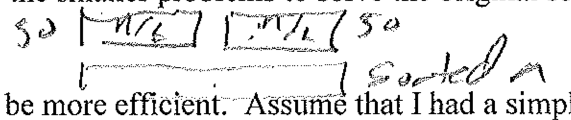
split list in half



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

lists of size 1 or 0 are already sorted

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



e) Consider why a recursive sort might be more efficient. Assume that I had a simple n^2 sorting algorithm with n = 100, then there is roughly $100^2 / 2$ or 5,000 amount of work. Suppose I split the problem down into two smaller sorting problems of size 50.

- If I run the ~~n²~~ n^2 algorithm on both smaller problems of size 50, then what would be the approximate amount of work?

$$\frac{50^2}{2} + \frac{50^2}{2} = 50^2 = 2500$$

- If I further solve the problems of size 50 by splitting each of them into two problems of size 25, then what would be the approximate amount of work?

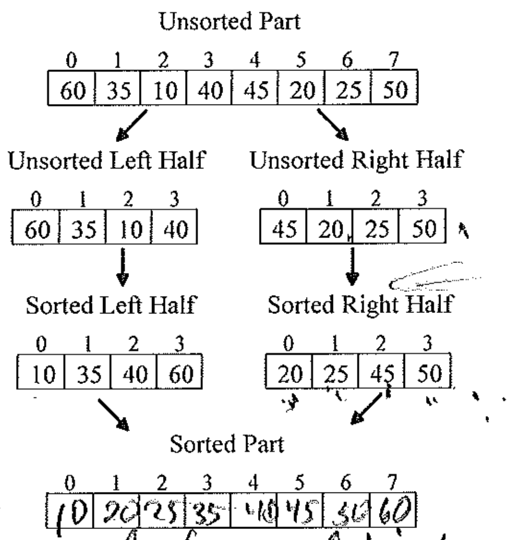
$$\frac{25^2}{2} + \frac{25^2}{2} + \frac{25^2}{2} + \frac{25^2}{2} = 2 \times 25^2 = 1250$$

3. The general idea 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

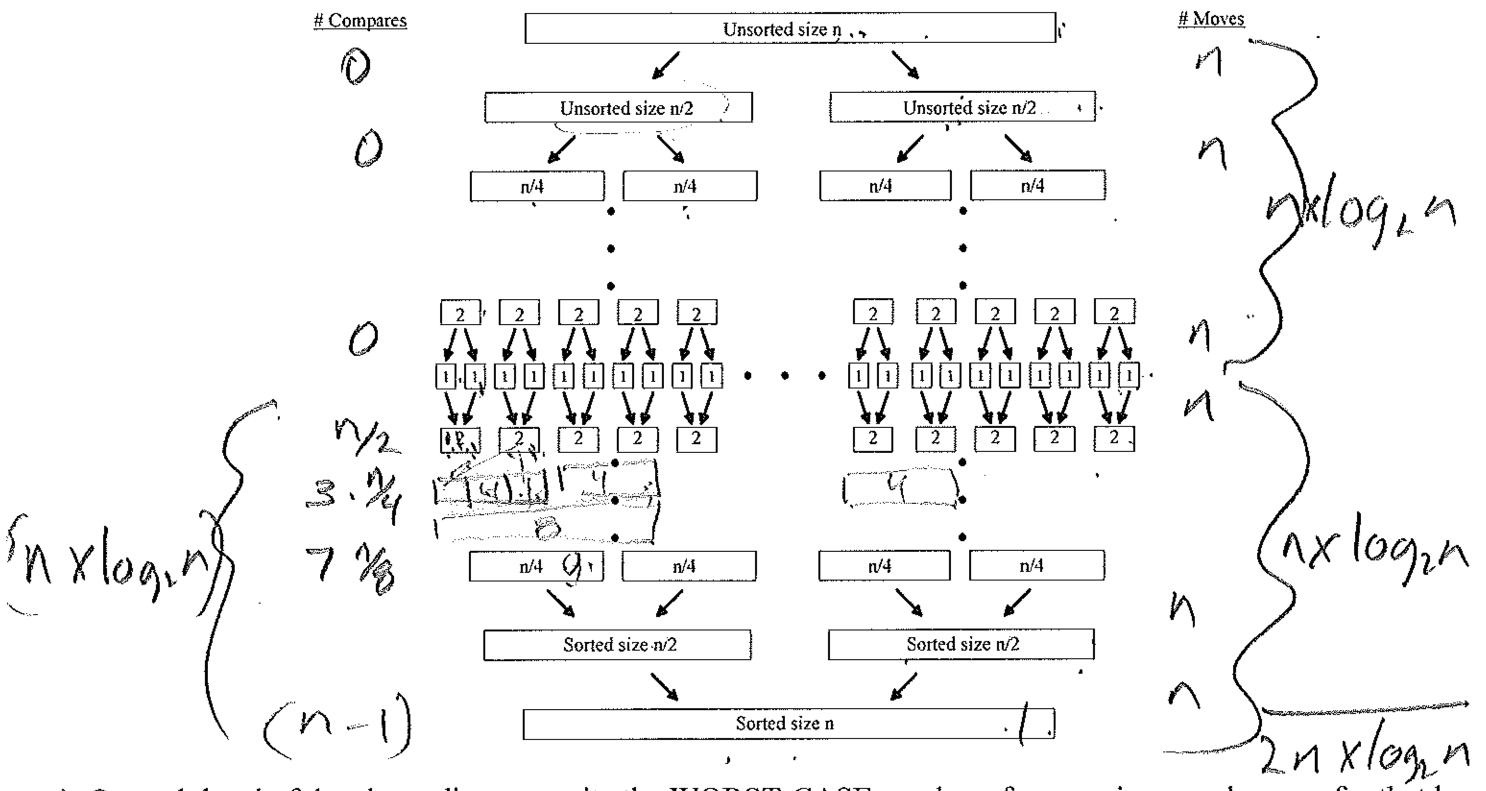
a) Fill in the merged Sorted Part in the diagram.

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

Keep track of 3 indexes: nextLeft, nextRight, nextSorted all starting at 0 compare nextLeft item from leftHalf to nextRight item from rightHalf and move smaller to nextSorted spot of Sorted list. Increment nextLeft or nextRight depending on which is moved. If a half list runs out, copy remaining part of other half to end of sorted list



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)?

compares $\approx n \log_2 n$ moves: $2n \times \log_2 n$

c) What is the big-oh for worst-case? $O(n \log_2 n)$

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

left Pivot Index right

All items < to Pivot	Pivot Item	All items >= to Pivot

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

```
def partition(lyst, left, right):
    # Find the pivot and exchange it with the last item
    middle = (left + right) // 2
    pivot = lyst[middle]
    lyst[middle] = lyst[right]
    lyst[right] = pivot
    # Set boundary point to first position
    boundary = left
    # Move items less than pivot to the left
    for index in range(left, right):
        if lyst[index] < pivot:
            temp = lyst[index]
            lyst[index] = lyst[boundary]
            lyst[boundary] = temp
            boundary += 1
    # Exchange the pivot item and the boundary item
    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(list,pivotIndex+1,right)
```

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

	0	1	2	3	4	5	6	7	8	left	right	index	boundary	pivot
lyst:	54	26	93	17	~~50~~ 20	31	44	55	~~20~~ 50	0	8			

b) What initial arrangement of the list would cause partition to perform the most amount of work?

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