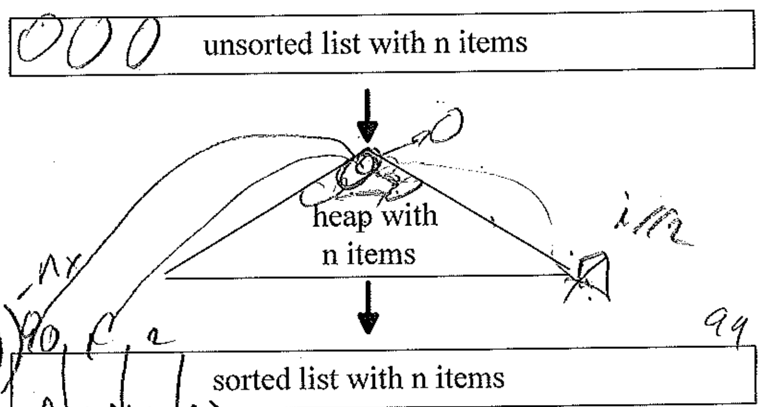
1. So far, we have looked at simple sorts consisting of nested loops. The # of inner loop iterations $n*(n-1)/2$ is $O(n^2)$. Consider using a min-heap to sort a list. (methods: `BinHeap()`, `insert(item)`, `delMin()`, `isEmpty()`, `size()`)

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:(myList)



1. Create an empty heap
   myHeap = BinHeap()
2. Insert all n list items into heap
   n log₂n { for item in myList: — n x
   myHeap.insert(item) ← log₂n
3. delMin heap items back to list in sorted order
   n log₂n { for index in range(len(myList)):
   myList[index] = myHeap.delMin() — log₂n

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? # items to sort

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

[n]

[n/2][n/2]

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

n = 0

n = 1

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

[n/2 sorted][n/2 sorted]

[n sorted] merge sort

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 n^2 algorithm on both smaller problems of size 50, then what would be the approximate amount of work?

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

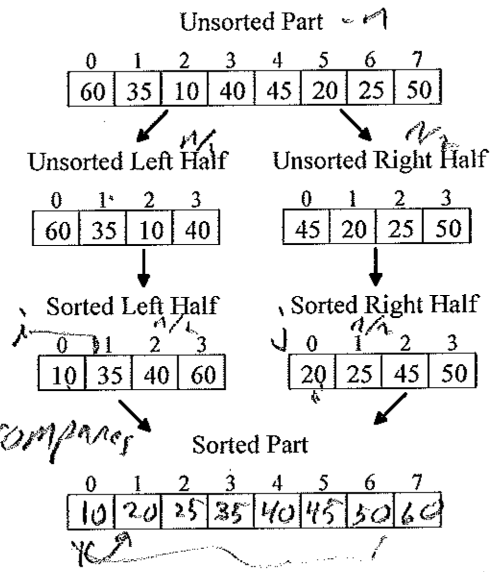
- If I further solve the problems of size 50 by splitting each of them into two problems of size 25, then what would be the approximate amount of work?

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

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

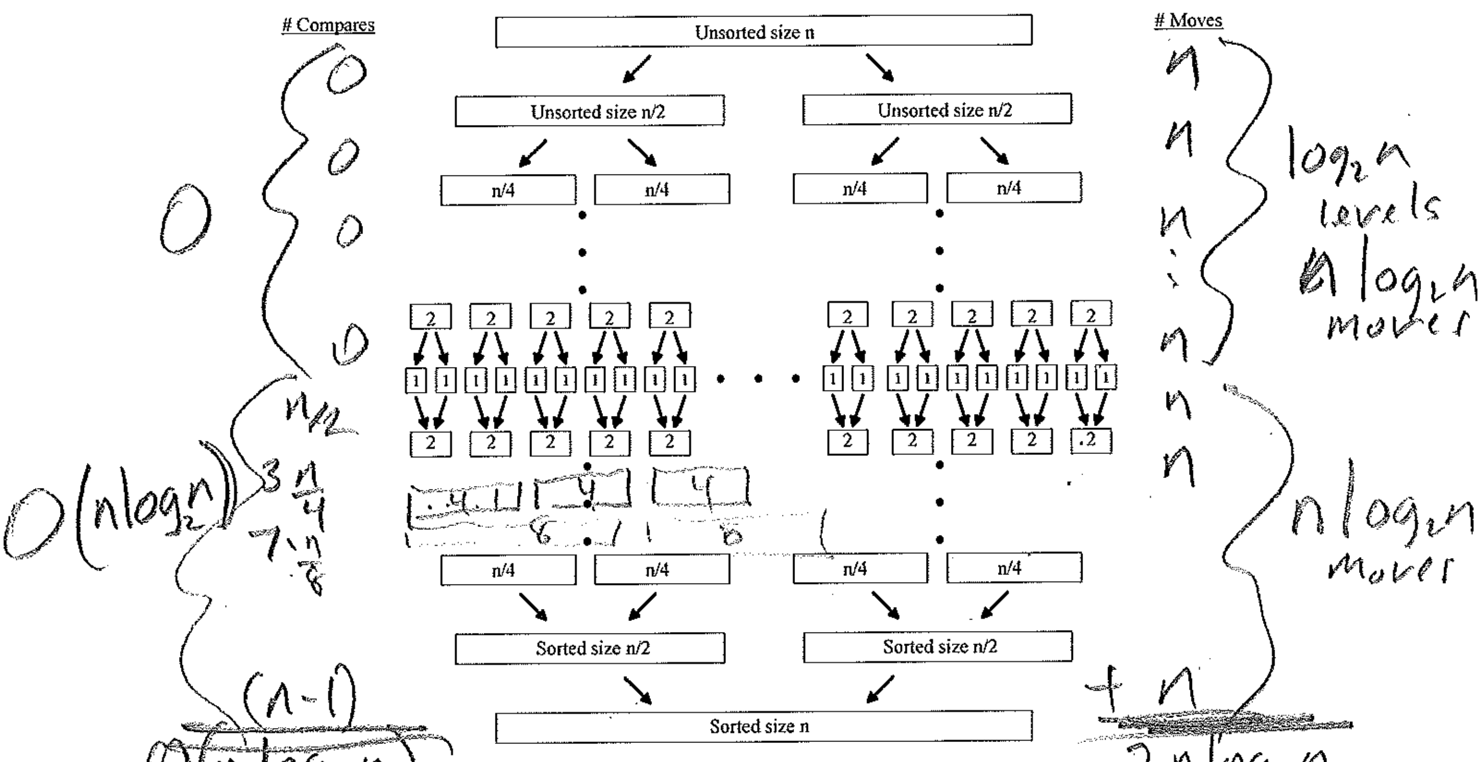
a) Fill in the merged Sorted Part in the diagram.

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



worst case # compares (n-1)
best-case n/2
O(n)

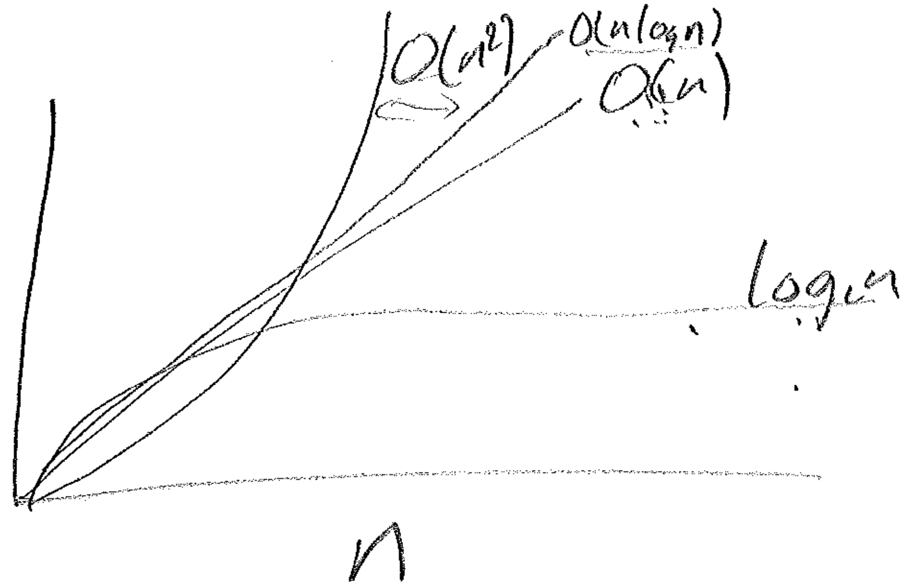
4. Merge sort is substantially faster than the simple sorts. Let's analyze the number of comparisons and moves of merge sort. Assume "n" items to sort.



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? $O(n \log_2 n)$



O(n²)
O(n log n)
O(n)
log n
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(lyst, 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	middle
lyst:	~~54~~	26	~~93~~	~~17~~	~~50~~	~~31~~	~~44~~	55	~~20~~	0	8	~~0~~	~~0~~	50	4

index: 1 2 3 4 5 6 7

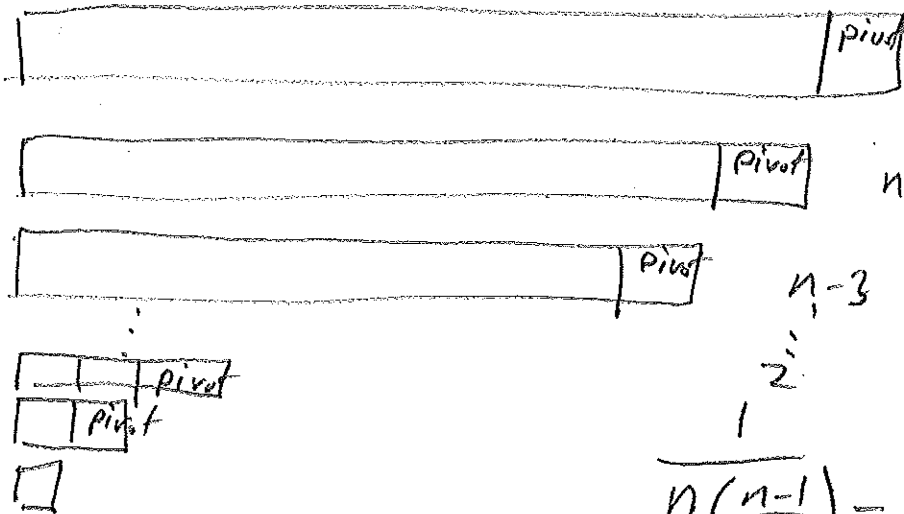
boundary: 1 2 3 4 5

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?

$O(n)$

d) What would be the overall, worst-case $O(\ )$ for Quick Sort? Repeatedly pick pivot that's largest



$n-1$ compares + swaps

$n-2$ compares + swaps

$n-3$ "

⋮

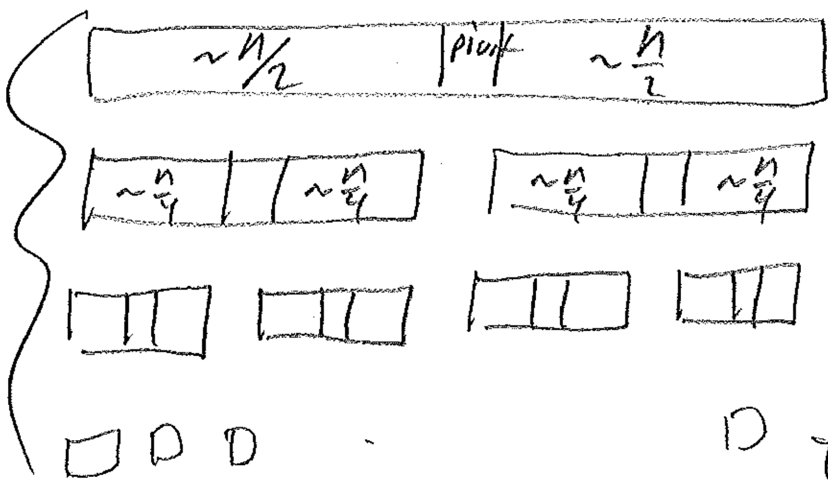
2

1

$n\left(\frac{n-1}{2}\right) = O(n^2)$

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?

$\log_2 n$ levels



$\sim n$ compares

$\sim n$ compares

$\sim n$ compares

⋮

$0 + n$ compares

$O(n \log_2 n)$

f) What would be the big-oh for Quick Sort in the average case? - picking a pivot at random should roughly partition list into equal pieces, So $O(n \log_2 n)$ on average.

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?

If list already sorted, picking first item as pivot leads to worst case $O(n^2)$ compares since pivot always on left end