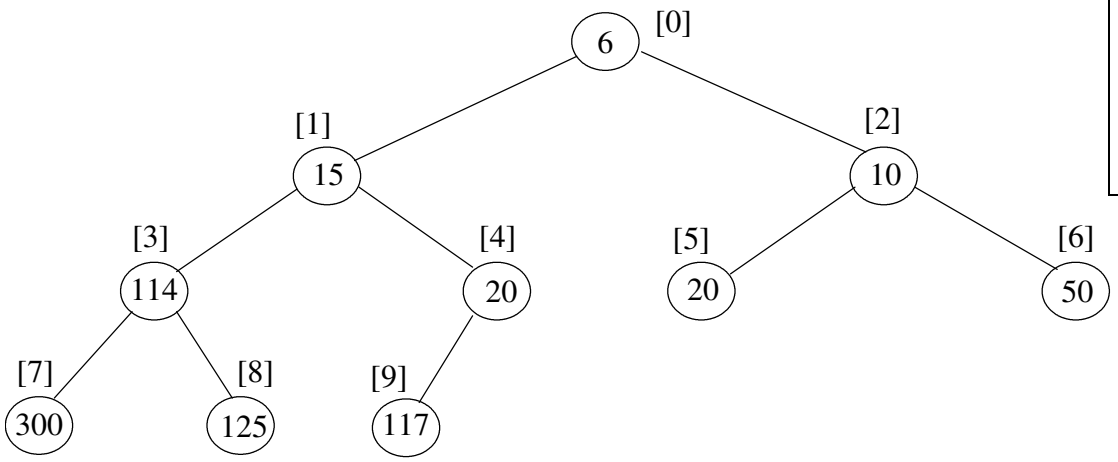
Name:_____

0. How would we write the BinaryHeap siftDown function recursively?

```
template <class T>
class BinaryHeap {
private:
      int maxSize;
      int numItems;
      T * heap;
      ...
```



Algorithm for delMin	**Algorithm for siftDown(int currentPosition)**

temp = heap[0]
numItems--
heap[0] = heap[numItems]
siftDown(0)
return temp
	while true (infinite loop) do
	  if the currentPosition has NO children then
	  return
	
	  if the currentPosition has only a left child then
	  min. child is the left child
	  else if the left child < right child then
	  min. child is the left child
	  else
	  min. child is the right child
	
	  if the item at current Position > min. child then
	  exchange these two items
	  update the currentPosition
	  else
	  return since we are done sifting down
	end while

a) What base case(s) (trival non-recursive cases) would we have?

b) What recursive case(s) would we have?

c) Write the recursive siftDown code.

Name:_____

1. Recall the iterative (i.e., using a loop) *binary search* on an array sorted in *ascending order*.

```
int binarySearch(int array[], int size, int value) {
   int first = 0,              // First array element
       last = size - 1,        // Last array element
       middle;                 // Mid point of search

   while (first <= last) {

      middle = (first + last) / 2;     // Calculate mid point
      if (array[middle] == value) {    // If value is found at mid

         return middle;

      } else if (array[middle] > value) {  // If value is in lower half

         last = middle - 1;

      } else {

         first = middle + 1;           // If value is in upper half

      } // end if
   } // end while

   return -1  // -1 indicates unsuccessful search
} // end binarySearch
```

A trace the `binarySearch` code using the following actual parameters by showing the changes to first, last, middle, position, and found.

	0	1	2	3	4	5	6		(MAX-1)
array:	2	3	4	5	7	8	9		

size: 7 value: 7

first	last	middle	
0	6	3	
4	6	5	
4	4	4	(return 4)

a) How might we think of this search problem recursively, i.e., solving a searching problem by splitting the problem into one or more simplier search problems?

b) What additional parameter(s) would be needed to specify the problem recursively? (needed to specify the smaller problem size)

c) What base case(s) are trivial enough that the answer is obvious?

d) Write the code for recursive binarySearch.

Name:_____

2. So far, we have only looked at simple sorts. Recall that all simple sorts consist of nested loops:
- an outer loop that keeps track of the dividing line between the sorted and unsorted part
- an inner loop that grows the size of the sorted part by one

Usually, the number of inner loop iterations is something like = (n-1) + (n-2) + (n-3) + ... + 3 + 2 + 1 = n * (n-1)/2 which is "theta" of n^2 (i.e., $\Theta(n^2)$).

Next, we'll look at a couple of more "advanced" sorting algorithms that are recursive: Merge sort and Quick Sort. In general, a problem can be solved recursively if it can be broken down into smaller problems that are identical in structure to the overall problem.

a) What determines the "size" of a sorting problem?

b) How might we break the original problem down into smaller problems that are identical? Are there any additional parameters that might be needed? (remember recursive binary search needed extra parameters)

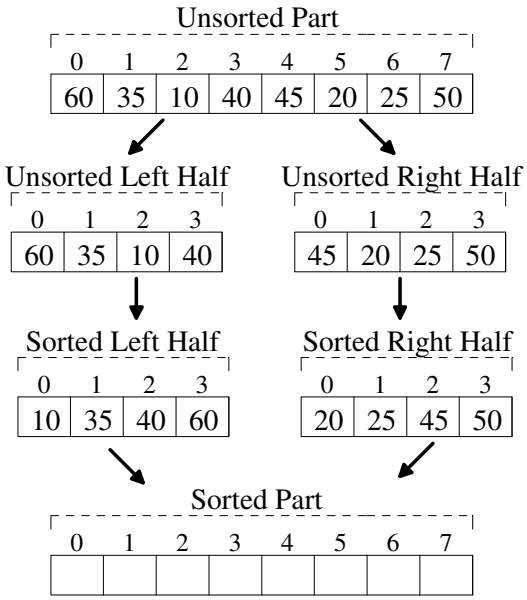
c) What base case(s) (i.e., trival, non-recursive case(s)) might we encounter with recursive sorts?

d) 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 or 10,000 amount of work. Suppose I split the problem down into two smaller 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?

- 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?

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 problem recursively using merge sort
- "Merge" the solution to the smaller problems together to solve the original sorting problem of size n

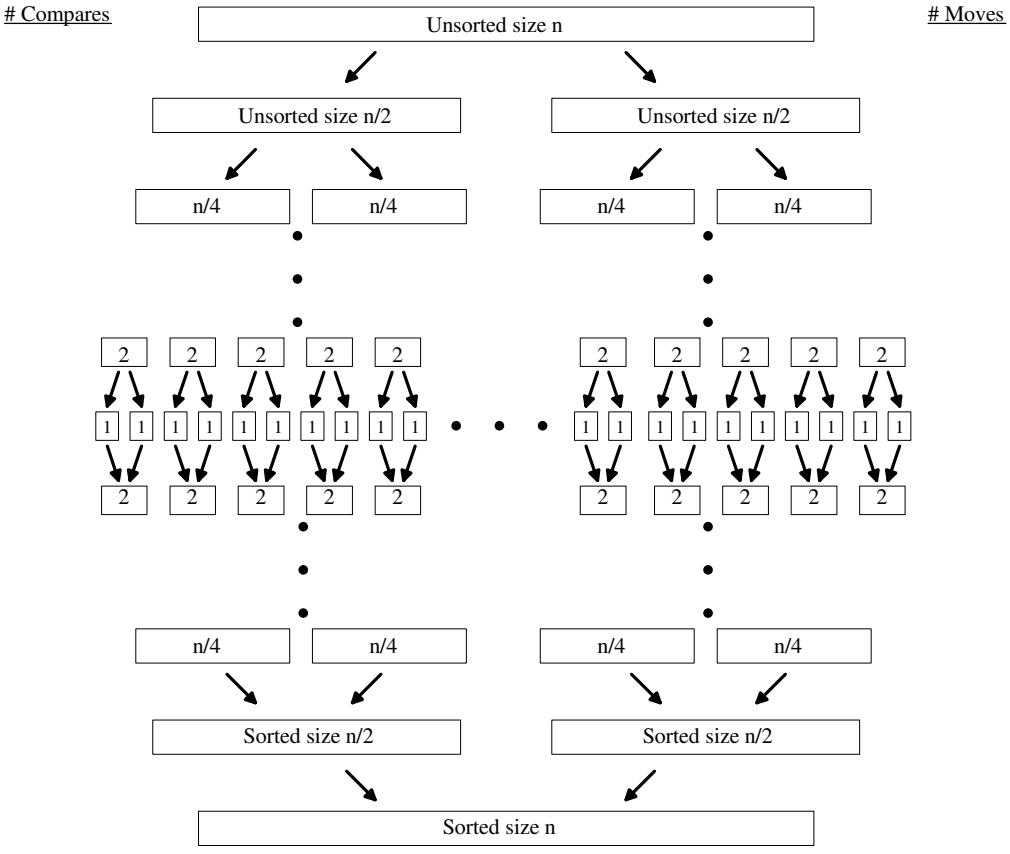
Name:_____



a) Fill in the sorted part in the above

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 total number of comparison for that level on the left, and the total number of moves for that level on the right.

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

c) What is the total number of moves for the whole algorithm (i.e., add all levels)?

d) What is BEST-CASE total number of comparisons for the whole algorithm?

5. *Quick sort* is another advanced sort that often is quicker than merge sort (hence its name). The general idea is as follows. Assume "n" items to sort.

- Select a "random" item in the unsorted part as the *pivot*
- Rearrange (called *partitioning*) the unsorted items such that:

Pivot Index

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

- Quick sort the unsorted part to the left of the pivot
- Quick sort the unsorted part to the right of the pivot

a) What base case(s) would we have?

b) Because of the recursive nature of quick sort, what "extra" parameters would we need to specify the part of the list to sort?

c) Given the following `partition` function which returns the index of the pivot after this rearrangement.

```
int partition(int set[], int start, int end) {
   int pivotValue, pivotIndex, mid;

   mid = (start + end) / 2;
   swap(set[start], set[mid]);
   pivotIndex = start;
   pivotValue = set[start];
   for (int scan = start + 1; scan <= end; scan++) {
      if (set[scan] < pivotValue) {
         pivotIndex++;
         swap(set[pivotIndex], set[scan]);
      } // end if
   }// end for
   swap(set[start], set[pivotIndex]);
   return pivotIndex;
} // end partition

// swap simply exchanges the contents of value1 and value2.
void swap(int &value1, int &value2) {
   int temp = value1;

   value1 = value2;
   value2 = temp;
} // end swap
```

Write the recursive quickSort function.