Question 1

There are three obvious ways of merging two similarly sized heap trees s and t into a single heap tree:

1. *Move the items one at a time from the smaller heap tree into the larger heap tree using the insert algorithm studied in lecture notes section 6.4.*
2. *Repeatedly move the last items from one heap tree to the other, with bubbling up, until the new binary tree makeTree(0,t,s) is complete. Then move the last item of the new tree to replace the dummy root "0", and bubble down that new root.*
3. *Concatenate the array forms of s and t and use the heapify algorithm from lecture notes section 6.6 to convert it into a new heap tree.*

Comment on the average-case time complexities of each approach, and thus suggest which approach would be the best in practice to use in general. Are there any special cases for which a different choice might be appropriate?

The average-case time complexities (m is the number of nodes in the smaller heap tree, and n is the number of nodes in the larger heap tree):

1. Inserting a node into the larger tree is $O(\log_2 n)$. We have to do this m times, however, hence the whole thing is $O(m \log_2 n)$
2. At first we must make the trees be the same height (or differ by one if the smaller tree is full), since only then will `makeTree(0, t, s)` produce a complete tree. The difference in height is $\log_2 n - \log_2 m$, i.e. $\log_2 \frac{n}{m}$. Each insertion of a node from the smaller subtree will be will be $O(\log_2 m)$. Hence, the work to make the trees be the same height is $O\left(\log_2 m \times \log_2 \frac{n}{m}\right)$. The movement of the last node to the root can be done in constant time, and the subsequent bubbling down is $O(\log_2(m+n))$. Hence the whole thing is $O\left(\log_2 m \times \log_2 \frac{n}{m}\right)$.
3. Since the heap trees are backed by arrays no work needs to be done to convert them to arrays. Array concatenation can be done in linear time, though this is only relevant if we care about actually having the array's elements in contiguous memory, which we don't. We can simulate array concatenation in constant time. Running `heapify` is $O(m+n)$, so the whole thing is just $O(m+n)$.

In general, the 2nd approach will work best.

In cases where m is very small, the 3rd approach could be better.

Question 2

Outline in words an efficient algorithm for merging two heap trees equal height into a single heap tree using approach (ii) from Question 1. You can assume that you know which is the smaller of the two trees, and have called that one s and the other one t.

Now convert your algorithm into an efficient pseudocode procedure mergeHT(s,t) that returns the merged heap tree. You can use the standard primitive binary tree procedure makeTree(r,s,t), and any of the following heap tree procedures:

- *size(t) – returns the size of heap tree t*
- *height(t) – returns the height of heap tree t*
- *full(t) – returns true if heap tree t has full last level, false otherwise*
- *lastLeaf(t) – returns the right most node of the bottom row of heap tree t*
- *bubbleDown(i,t) – bubbles down the node i of heap tree t to its correct position*
- *delete(i,t) – deletes node i from heap tree t, by replacing it with the last leaf, bubbling it up or down, and updating the tree size*
- *insert(v,t) – inserts value v into heap tree t immediately after the current last leaf, bubbling it up, and updating the tree size*

Include comments in your code that explain how your algorithm works.

Until the heights of s and t are the same, or if s is full, differ by at most one, get and remove the last leaf of t and insert it into s. Then join the two trees by creating a tree which has s and t as its left and right subtrees respectively. Get and remove the last leaf of t, and set the value of the root equal to its value. Finally, bubble down the root.

```
equalHeights(s, t) {
    difference = height(t) - height(s)
    // Return true if either they have the same height
    // or s is full and only one less.
    return difference == 0 or (difference == 1 and full(s))
}

getLastAndDelete(t) {
    last = lastLeaf(t) // Get the last item
    delete(size(t), t) // Remove it (1-based indexing)
    return last        // Return it
}

mergeHT(s, t) {
    // Until the resultant tree will be complete
    while(not equalHeights(s, t)) {
        // Move the last item from the
        // larger heap to the smaller one
        insert(getLastAndDelete(t), s)
    }
    // Merge s and t into one tree, with
    // the last item in the root position
    last = getLastAndDelete(t)
    tree = makeTree(last, s, t)
    // Bubble down the root (1-based indexing)
    bubbleDown(1, tree)
    // Return the merged heap tree
    return tree
}
```

Question 3

Often one needs to determine whether an array contains any duplicate items. Write a procedure duplicates(a), which doesn't involve sorting, that returns true if integer array a contains duplicates, and false otherwise. You may assume that you have access to a procedure size(a) that returns the size of an array a.

What is the worst case time complexity of your algorithm?

What about the average case time complexity of your algorithm?

```
// Uses 0-based indexing
duplicates(a) {
    n = size(a)
    for(i = 0; i < n-1; i++) {
        for(j = i + 1; j < n; j++) {
            if(a[i] == a[j]) {
                return true
            }
        }
    }
    return false
}
```

Both the worst and average case complexities are $O(n^2)$.

Question 4

Suppose you have access to a procedure Xsort(a) that returns a sorted version of array a, that has average and worst case time complexity of at least O(n log n). Write an efficient new procedure duplicates2(a), to perform the test for duplicates, that begins by calling Xsort(a) to sort the given array. You may again assume that you have access to a procedure size(a) that returns the size of an array a.

What is the overall worst case time complexity of your algorithm?

What about the overall average case time complexity of your algorithm?

Thus, comment on when it is worth using Xsort in a duplicates testing algorithm.

```
// Uses 0-based indexing
duplicates2(a) {
    a = Xsort(a)
    n = size(a)
    for(i = 0; i < n-1; i++) {
        if(a[i] == a[i+1]) {
            return true
        }
    }
    return false
}
```

The overall worst and average cases are the same as those of `Xsort`. This is because the extra work we do is $O(n)$ and hence (because it is of a lower order) will not affect the overall time complexity.

It could be worth using `Xsort` if its average or worst case complexities are less than $O(n^2)$, for example $O(n \log n)$. It could also be worth using `Xsort` if both the average and worst case complexities are $O(n^2)$ but the associated constants and lower order terms are fewer than with the original `duplicates` procedure.

Question 5

Shaker Sort (also called Bidirectional Bubble Sort) successively compares adjacent pairs of items in an array, and exchanges them if they are in the wrong order. It alternatively passes forwards and backwards through the array until no more exchanges are needed. Explain why this algorithm is guaranteed to terminate with the array sorted.

Work through the sorting of array [5, 3, 7, 8, 1, 9] using this algorithm, writing down the direction of the pass at each swap, the swaps that occur, and the array after each swap.

TODO

Swapped Numbers	Pass Direction	Resultant Array
5 and 3	Forwards	[3, 5, 7, 8, 1, 9]
8 and 1	Forwards	[3, 5, 7, 1, 8, 9]
7 and 1	Backwards	[3, 5, 1, 7, 8, 9]
5 and 1	Backwards	[3, 1, 5, 7, 8, 9]
3 and 1	Backwards	[1, 3, 5, 7, 8, 9]