Problem 1

An array A of n distinct numbers are said to be *unimodal* if there exists an index k, $1 \leq k \leq n$, such that $A[1] \leq A[2] \leq \cdots \leq A[k-1] \leq A[k]$ and $A[k] \geq A[k+1] \geq \cdots \geq A[n]$. In other words A has a unique peak and are monotone on both sides of the peak. Design an $\mathcal{O}(\log n)$ algorithm that finds the peak in a unimodal array of size n.

```
low = 0
high = A.length // this is n

while(1)
    i = (high - low) / 2
    if( A[i-1] < A[i] && A[i] < A[i+1] )
        low = i
    else if( A[i-1] > A[i] && A[i] > A[i+1] )
        high = i
    else
        return A[i]
```

$$T(n) = T(n/2) + c$$
$$T(n) = T(n/4) + c + c$$
$$T(n) = T(n/2^k) + ck$$
$$\text{if } k = log_2 n$$
$$T(n) = T(n/2^{log_2 n}) + c\ log_2 n$$
$$T(1) + c\ log_2 n$$
$$= \mathcal{O}(log\ n)$$

Problem 2

Let A be a heap of size n. Let T be the binary tree corresponding to A (that is, $A[1]$ is the root node of T, $A[2]$ and $A[3]$ are the left child and right child nodes of $A[1]$, etc). Prove the following properties of T (you need to pay attention to the floor notation in the equations).

(a) The height of T is $\lfloor log\ n \rfloor$;

The maximum number of nodes in a complete binary tree of height h is, $2^{h+1} - 1$

The minimum number of nodes in a binary tree of height h is, 2^h this is because the maximum nodes in a binary tree of height $h - 1$ is, $2^h - 1$.

Thus, $2^h \leq n < 2^{h+1}$

Then by taking the log of all sides, $h \leq log(n) < h + 1$.

Since both h and $h + 1$ are integers, it follows thats $h = \lfloor log(n) \rfloor$

(b) T is a *balanced* binary tree, namely if we denote the two subtrees of the root node of T by T_L and T_R respectively, then $|T_R| \leq |T_L| \leq 2n/3$ (In fact, you have just proved a more general statement: the inequality holds for any non-leaf node in the tree);

The left child of the tree must fill up before the right child does as a result of being a balanced binary tree. The largest possible imbalance that can occur is the final row of the left subtree being completely populated and the final row of the right subtree being completely empty. The left subtree will be bounded by 2n/3.

(c) The leaf nodes of T are $A[\lfloor n/2 \rfloor + 1]$, $A[\lfloor n/2 \rfloor + 2]$, $\ldots$, $A[n]$.

When heaps are stored as arrays non-leaf nodes are stored before leaf nodes.

Problem 3

Suppose you are given an unsorted list of n distinct numbers. However, the list is close to sorted in the sense that each number is at most k positions away from its position in the sorted list. For example, suppose the sorted list is in ascending order, then the smallest element in the original list will lie between position 1 and position $k + 1$, as position 1 is its position in the final sorted list. Design an efficient algorithm that sorts such a list. Your running time should depend on both n and k.

```
A[] // unsorted list of distinct numbers
H   // min-heap

H = new Heap(k+1) // create min-heap of size k+1
for( i=0, i<=k, i<n, i++)
    H[i] = [i]
H.heapify()

// A will continue where it left off
// j is the index for the min element in H
for( i=k+1, j=0, j<n, i++, j++)
    if i<n
        A[j] = H.replaceMinimum(A[i])
    else
        A[j] = H.removeMinimum()
```

Make heap of size k will take $\mathcal{O}(k)$ time. Remove the minimum element from the heap and add it to the array then from the left over elements add one to the heap.

Thus, $\mathcal{O}(k) + \mathcal{O}((n - k) * log\ k)$

Problem 4

Using the approach of Divide-and-Conquer, design an algorithm that finds the *median* of an unsorted list (the *median* of a list of n number is the $\lceil n/2 \rceil$-th smallest number in the list; that is, the one that lies in the middle). State both the worst-case running time and the average-case running time of your algorithm. You do not need to give rigorous mathematical proofs of your claims but you should provide reasoning to support your conclusions. (hint: you will find the approach of **QuickSort** useful for this problem as well. Also, when applying the method of recursion, you will often find that working with a more *general* problem is a lot easier.)

In order to find the k-th smallest element of an unsorted list (k being the median) an algorithm similar to **QuickSort** is used. However, the array only need be partially sorted as we will know which side to work on through comparing the pivot element to the k-th element.

```
k = ceil(list.length / 2)
select (list, start, end, k)
    if start==end
        return list[start]
    pivotIndex = [random number between start and end]
    pivotIndex = partition(list, start, end, pivotIndex)

    // after partition the pivot is in its final position
    if k == pivotIndex
        return list[k]
    else if k < pivotIndex // median is below the pivot
        return select( list, start, pivotIndex-1, k)
    else
        return select( list, pivotIndex+1, end, k)
```

Worst Case: $\mathcal{O}(n^2)$
If bad pivots are chosen each time then it could be possible that no work will be done and the list will be sorted in a worst case QuickSort style.

Average Case: $\mathcal{O}(n)$
If good pivots are chosen (those which decrease problem size) then the search set decreases in size exponentially. The running time looks like $T(n) = T(n/b) + \mathcal{O}(n)$, however, since the work is divided by some fraction we will only be working on one side. Here Case 1 of the Master Theorem always applies since $log_b 1 = 0$ and $c = 1$ meaning $c > log_b a$ in other words the running time is $\mathcal{O}(n)$

Problem 5

You have n pairs of nuts and bolts (the bolts screw into the nuts) that have been dropped onto a table with the nuts on one side and the bolts on another side; however you cannot tell whether one nut is bigger than another nut or whether one bolt is bigger than another bolt. You can attempt to screw a bolt into a nut and this will tell you either that the nut matches the bolt, or that the bolt is too large or too small for the nut. Call this a *nut/bolt comparison.* Give a randomized algorithm that will match all nuts with their corresponding bolts with at most $\mathcal{O}$(n log n) nut/bolt comparisons.

To solve this problem you essentially do a quick sort on two arrays at the same time, using corresponding elements in one array as a pivot for the other array. You begin by picking any random bolt (you can begin with a random nut, but this explanation will assume you begin with a bolt). Now you can treat that bolt as the pivot for the nuts and partition the nuts by which are bigger than the bolt and which are smaller than the bolt. At some point while portioning the nuts, you will also find the nut which matches the bolt you are currently using to sort the nuts, give that nut the same placement as the bolt you are currently using. Once you are doing partitioning the nuts, you can now use the nut that corresponds to the bolt you were just using to partition nuts in order to partition the bolts. By doing this you will have three things. You will have a nut/bolt pair, you will have a set of unpaired nuts and bolts that are smaller than the pair you used for partitioning, and you will have a set of unpaired nuts and bolts that are bigger than the pair you used for partitioning. Now you just recursively work on the two sets of unpaired nuts/bolts using the same algorithm and you will end with all pairs of nuts and bolts in sorted order. Because you are using a randomized version of **QuickSort** (since you are picking a random bolt to start the comparisons in each quick sort) the worst case $\mathcal{O}(n^2)$ is extremely unlikely to show up and you will deal with at most $\mathcal{O}(n\ log\ n)$ comparisons to find the corresponding pairs of nuts and bolts(which will also be sorted).