1. (10 pts) *You are given n metal balls $B_1, \ldots, B_n$, each having a different weight. You can compare the weights of any two balls by comparing their weights using a balance to find which one is heavier.*

   (a) *Consider the following algorithm to find the heaviest ball:*

      i. *Divide the n balls into $\frac{n}{2}$ pairs of balls.*

      ii. *Compare each ball with its pair, and retain the heavier of the two.*

      iii. *Repeat this process until just one ball remains.*

   Illustrate the comparisons that the algorithm will do for the following $n = 8$ input:

$$B_1 : 3, \quad B_2 : 5, \quad B_3 : 1, \quad B_4 : 2, \quad B_5 : 4, \quad B_6 : \tfrac{1}{2}, \quad B_7 : \tfrac{5}{2}, \quad B_8 : \tfrac{9}{2}$$

   For the example,

$$B_1 : 3, B_2 : 5, B_3 : 1, B_4 : 2, B_5 : 4, B_6 : 0.5, B_7 : 2.5, B_8 : 4.5$$

   At first round, the weighings are

$$(B_1, B_2), (B_3, B_4), (B_5, B_6), (B_7, B_8)$$

   The balls that remain are

$$B_2, B_4, B_5, B_8$$

   The second round we have 2 weighings

$$(B_2, B_4), (B_5, B_8)$$

   The balls that remain are

$$B_2, B_8$$

   The last round has one more weighing and B_2 is the heaviest ball.

   (b) *Show that for n balls, the algorithm* (1a) *uses at most n comparisons.*

   At each round, the number of balls is halved. The algorithm stops when there is just one ball left.
   At the first round, we have $\frac{n}{2}$ weighings, $\frac{n}{4}$ for the second round and so on until $\frac{1}{2} \leq \frac{n}{2^k} \leq 1$. The number of rounds is therefore $k = \log_2(n)$ and the total number of weighings is

$$\left(\frac{n}{2} + \frac{n}{4} + \ldots + 1\right) \leq n\left(\frac{1}{2} + \frac{1}{4} + \ldots\right) \leq n\left(\frac{\frac{1}{2}}{1-\frac{1}{2}}\right) \leq n$$

(c) *Describe an algorithm that uses the results of* (1a) *to find the second heaviest ball, using at most* $\log_2 n$ *additional comparisons. There is no need for pseudocode; just write out the steps of the algorithm like we have written in* (1a).
Hint: if you follow sports, especially wrestling, read about the repechage.

(d) *Show the additional comparisons that your algorithm in* (1c) *will perform for the input given in* (1a).

(C and D): Let B_j be the heaviest ball found in (A). While carrying out (A), let us record all the weighings performed. In particular, collect all balls that were compared against B_j and discarded at some point when solving (A).
There are at most $\log_2(n)$ such balls. Now find the heaviest ball in this set by rerunning the scheme in (A). This yields the second heaviest ball.

2. (10 pts) *An array is almost* k *sorted if every element is no more than* k *positions away from where it would be if the array were actually sorted in ascending order.*

   (a) *Write down pseudocode for an algorithm that sorts the original array in place in time* $n\,k\log k$. *Your algorithm can use a function* `sort`(A, ℓ, r) *that sorts the subarray* $A[\ell], \ldots, A[r]$.

   The pseudocode simply goes through the array and sorts every subarray of size k as follows:

```
def sortAlmostKSorted(A, k):
    n = len(A)
    for i in range(0, n - k):
        sort(A, i, i + k)
    return
```

3. (20 pts) *Consider the following strategy for choosing a pivot element for the* `Partition` *subroutine of QuickSort, applied to an array* A.

   - *Let* n *be the number of elements of the array* A.
   - *If* $n \leq 15$, *perform an Insertion Sort of* A *and return.*

- *Otherwise:*
  - *Choose* $2\lfloor\sqrt{n}\rfloor$ *elements at random from* n*; let* S *be the new list with the chosen elements.*
  - *Sort the list* S *using Insertion Sort and use the median* m *of* S *as a pivot element.*
  - *Partition using* m *as a pivot.*
  - *Carry out QuickSort recursively on the two parts.*

(a) *If the element* m *obtained as the median of* S *is used as the pivot, what can we say about the sizes of the two partitions of the array* A*?*

The median m has at least $\sqrt{n}$ elements less than or equal to it and at least $\sqrt{n}$ elements greater than or equal to it. Therefore the partition of A will have at least $\sqrt{n}$ elements on one partition and at most $n - \sqrt{n}$ on the other.

(b) *How much time does it take to sort* S *and find its median? Give a* Θ *bound.*

To sort $2\sqrt{n}$ elements using insertion sort requires $\Theta(n)$ time. Finding median of a sorted array is just constant time.

(c) *Write a recurrence relation for the worst case running time of QuickSort with this pivoting strategy.*

The recurrence for worst case running time will be

$$T(n) = \begin{cases} C_0 & n \leq 15 \\ T(n - \sqrt{n}) + T(\sqrt{n}) + C_1 n & n > 15 \end{cases}$$

4. (20 pts) *Let* A *and* B *be arrays of integers. Each array contains n elements, and each array is in sorted order (ascending).* A *and* B *do not share any elements in common. Give a* $O(\lg n)$*-time algorithm which finds the median of* $A \cup B$ *and prove that it is correct. This algorithm will thus find the median of the* $2n$ *elements that would result from putting* A *and* B *together into one array. (Note: define the median to be the average of the two middle values of a list with an even number of elements.)*

Here's an algorithm that satisfies the requirement, in two parts:

```
TWO-ARRAY-MEDIAN(X, Y)
   n = length[X]                       // n also equals length[Y]
   median = FIND-MEDIAN(X, Y, n, 1, n)
   if median == NOT-FOUND
      then median = FIND-MEDIAN(Y, X, n, 1, n)
   return median

FIND-MEDIAN(A, B, n, low, high)
   if low > high
       then return NOT-FOUND
   else if n==2
       return ( max(A[1], B[1]) + min(A[2], B[2]) ) / 2
   else k = (low+high)/2
        if k == n and A[n] <= B[1]
           then return A[n]
        elseif k < n and B[n - k] <= A[k] <= B[n - k + 1]
            then return A[k]
        elseif A[k] > B[n - k + 1]
           then return FIND-MEDIAN(A, B, n, low, k - 1)
        else return FIND-MEDIAN(A, B, n, k + 1, high)
```

Let us start out by supposing that the median (the lower median, since we know we have an even number of elements) is in X. Let us call the median value m, and suppose that it is in $X[k]$. Then k elements of X are less than or equal to m and $n-k$ elements of X are greater than or equal to m. We know that in the two arrays combined, there must be n elements less than or equal to m and n elements greater than or equal to m, and so there must be $n-k$ elements of Y that are less than or equal to m and $n-(n-k) = k$ elements of Y that are greater than or equal to m.

Thus, we can check that $X[k]$ is the lower median by checking whether $Y[n-k] \leq X[k] \leq Y[n-k+1]$. A boundary case occurs for $k = n$. Then $n - k = 0$, and there is no array entry $Y[0]$; we only need to check that $X[n] \leq Y[1]$. Now, if the median is in X but is not in $X[k]$, then the above condition will not hold. If the median is in $X[k']$, where $k' < k$, then X[k] is above the median, and $Y[n-k+1] < X[k]$. Conversely, if the median is in $X[k'']$, where $k'' > k$, then $X[k]$ is below the median, and $X[k] < Y[n-k]$.

Thus, we can use a binary search to determine whether there is an X[k] such that either $k < n$ and $Y[n-k] \leq X[k] \leq Y[n-k+1]$ or $k = n$ and $X[k] \leq Y[n-k+1]$;

if we find such an $X[k]$, then it is the median. Otherwise, we know that the median is in Y, and we use a binary search to find a $Y[k]$ such that either $k < n$ and $X[n-k] \leq Y[k] \leq X[n-k+1]$ or $k = n$ and $Y[k] \leq X[n-k+1]$; such a $Y[k]$ is the median. Since each binary search takes $O(\lg n)$ time, we spend a total of $O(\lg n)$ time.