1) a)

We use a strategy similar to quicksort and call our algorithm QUICKSORT-RANGE. So, we select a pivot and sort left and right halves recursively. To choose a pivot, we select a random interval and compare it with other intervals and find all the intersecting intervals. We take intersection of this randomly selected interval with all the intersecting intervals to create the pivot ($[a_p, b_p]$).
While comparing the pivot with other elements, we place intervals with $a_i < a_p$ and $b_i < b_p$ to the left of pivot. We consider the overlapping intervals as same, hence we cluster all the intervals overlapping with pivot and put them immediately to the right of it.

```
PARTITION(A, p, r):
    pivot = RANDOM(A) // pick a random interval as pivot
    SWAP(pivot, A[r])
    intersection = pivot

    // Find the intersection with the randomly selected element to
    // get the pivot (intersection)
    for i from p to r-1:
        if INTERSECTS(intersection, A[i]):
            if A[i].left > intersection.left:
                intersection.left = A[i].left
            if A[i].right < intersection.right:
                intersection.right = A[i].right

    s = p

    // Place intervals less than the intersection to the left of it
    for i from p to r-1:
        if A[i].right < intersection.left:
            SWAP(A[i], A[s])
            s++

    SWAP(A[r-1], A[s])

    t = s+1
    i = r-1

    // Cluster together intervals which intersect
    while t<=i:
        if INTERSECTS(A[i], intersection):
            SWAP(A[t], A[i])
```

```
                t++
        else:
                i--

    return (s, t) // return an interval


QUICKSORT-RANGE(A, p, r):
    if p < r-1:
        pivot = PARTITION(A, p, r) // returns a range
        QUICKSORT-RANGE(A, p, pivot.left) // pivot.left is start of pivot's interval
        QUICKSORT-RANGE(A, pivot.right, r) // pivot.right is end of pivot's interval
```

b)

To prove that the algorithm correctly sorts a given set of intervals, consider the invariant that we have three set of intervals at any moment. Leftmost set with elements less than the intersection (A[i].right < intersection.left), a middle set with elements intersecting with the intersection (INTERSECTS(A[i], intersection)) and a rightmost set with elements greater than the intersection (A[i].left > intersection.right).

As long as this invariant is maintained, we would have sorted array by the time algorithm ends because elements in the middle set are at the right position and we are recursively calling the algorithm on leftmost set and the rightmost set.

At any point, if the invariant breaks, we maintain it again by swapping the right pair of elements. In case of generation of leftmost set an element violates the invariant if A[i].right < intersection.left and it is not present in the leftmost set, in which case we make a swap. Similarly the invariant is violated for the middle set when an element is not present in the middle set and it intersects with the intersection (INTERSECTS(A[i], intersection)). In that case we again make a swap and maintain the invariant.

Hence, the array we get in the end is sorted.

c)

As we are making recursive calls like quicksort, we can assume the time complexity as follows:

$T(n) = n + 2*T(n/2)$

We are assuming that on an average problem is divided equally because we are choosing pivot randomly.

To solve one recursive call it takes O(n) time because finding intersection, putting elements on leftmost side and clustering intersecting elements, each take linear time.

Now, consider the case when all intervals are overlapping with one x such that x belongs to each of the intervals. This implies that the intersection that we calculate in our algorithm belongs to each of the intervals. So, the loop commented as "Place intervals less than the intersection to the left of it" would not make any swap. Loop commented as "Cluster together intervals which intersect" would make n swaps as all the intervals overlap with the intersection. All this takes O(n) time and further recursive calls would be QUICKSORT-

RANGE(A, p, p) and QUICKSORT-RANGE(A, r, r). So, our algorithm stops here, making the time complexity as O(n).

2) a)

Observe that there are n^2 possible pairs consisting of one red and one blue jug. Thus, any algorithm that compares any given pair at most a constant number of time uses $O(n^2)$ comparisons. For example, the following simple approach works: pick a red jug and compare it against all blue jugs until the matching jug is found, and then pair these two jugs. Repeat for all red jugs.

```
SORT-JUGS(Ar, Ab): // Ar and Ab represent arrays having red and blue jugs
    result = []
    i = 0
    for r in Ar: // pick a red jug from the array
        for b in Ab: // pick a blue jug from the array
            if COMPARE(r, b):
                result[i] = r
                result[i+1] = b
                break

    return result

COMPARE(r, b):
    if r = b: // r and b have same capacity
        return true
    return false
```

b)

Consider the comparisons in the form of a decision tree where each internal node makes a decision between a red and a blue jug (red jug is smaller than the blue jug, red jug is larger than the blue jug or they are of same capacity). So, each internal node has labels like (r_i, b_j) (r_i represents a red jug and b_j represents a blue jug) and a comparison is made between these two jugs. So, bifurcation of node stops (i.e. it becomes a leaf node) if both the jugs are of same capacity, else if red jug is smaller, it goes left to make another comparison with another blue node (r_i, b_k) and if red jug is larger, it goes right to make comparison with another blue node (r_i, b_m). Leaves would have labels like (r_1, b_1), (r_2, b_2) and so on, considering r_i=b_i.
Height of the tree is the number of comparisons made for a single jug and we have n such jugs. Hence, total minimum comparisons = n*(height of tree)
There are n nodes at the lowest level and suppose the height is h, we have,
$n = 3^h$
$\Rightarrow h = \log_3 n$

Comparisons = n*h = n*$\log_3 n$

c)

```
QSORT-JUGS(Ar, Ab, result, i):
    if size(Ar) = size(Ab) = 1:
        result[i] = r // remaining last element in Ar
        result[i+1] = b // remaining last element in Ab
    else:
        r = random element from Ar
        for b in Ab:
            if b = r: // pivot element
                break
        Bless = [] // stores values less than the pivot
        Bgreater = [] // stores values greater than the pivot
        for b in Ab:
            if b < r:
                insert b in Bless
            else:
                insert b in Bgreater

        Rless = [] // stores values less than the pivot
        Rgreater = [] // stores values greater than the pivot
        for r in Ar:
            if r < b:
                insert r in Rless
            else:
                insert r in Rgreater

        result[i] = r
        result[i+1] = b

        QSORT-JUGS(Rless, Bless, result, i+1)
        QSORT-JUGS(Rgreater, Bgreater, result, i+1)
```

Here we are using similar strategy as in quicksort, i.e. we get a pivot using a jug of other color (because we cannot compare jugs of same color as given in problem) and then create two arrays: array with all jugs less than the pivot's capacity and array with all jugs with jugs larger than the pivot's capacity. Then, we recursively work on two subarrays.

Consider:

Ar = {r_1, r_2, r_3, … r_n}
Ab = {b_1, b_2, b_3, … b_n}

in sorted order, hence, $r_i=b_i$ for any i.

To get the complexity of the algorithm, we calculate the number of comparisons been made by the algorithm. We define a indicator random variable as follows:
X_{ij} = 1 if r_i and b_j are compared
= 0 if not

Total comparisons(X) = $\sum_{i=1}^{n} \sum_{j=1}^{n} X_{ij}$

We have to calculate the average number of comparisons made by this algorithm. Hence, we look at the expected value of this indicator random variable:

$$E[X] = E[\sum_{i=1}^{n} \sum_{j=1}^{n} X_{ij}]$$

$$E[X] = \sum_{i=1}^{n} \sum_{j=1}^{n} E[X_{ij}]$$

$$E[X] = \sum_{i=1}^{n} \sum_{j=1}^{n} Pr(r_i \, and \, b_j \, are \, compared)$$

Assume that $i \neq j$ and R_{ij} = {r_i, r_{i+1}, … r_j}. We can see that the algorithm compares r_i and b_j only when either r_i or r_j is the first element chosen as pivot. If that is not the case then r_i and b_j would not be compared and in further recursive calls lie in different subarrays. Also, each element of R_{ij} is equally likely to be picked as pivot.

Pr(r_i and b_j are compared) = Pr(r_i is picked first) + Pr(r_j is picked first)
= 1/(|j-i| + 1) + 1/(|j-i| + 1)
= 2/(|j-i| + 1)

$$E[X] = \sum_{i=1}^{n} \sum_{j=1}^{n} E[X_{ij}]$$

$$E[X] = \sum_{i=1}^{n} E[X_{ii}] + \sum_{i=1}^{n-1} \sum_{j=i+1}^{n} (E[X_{ij}] + E[X_{ji}])$$

$$E[X] = \sum_{i=1}^{n} E[X_{ii}] + \sum_{i=1}^{n-1} \sum_{j=i+1}^{n} 2 * E[X_{ij}]$$

$$E[X] = \sum_{i=1}^{n} E[X_{ii}] + \sum_{i=1}^{n-1} \sum_{j=i+1}^{n} \frac{4}{j-i+1}$$
$$E[X] = \sum_{i=1}^{n} E[X_{ii}] + \sum_{i=1}^{n-1} \sum_{k=1}^{n-i} \frac{4}{k+1}$$
$$E[X] < n + \sum_{i=1}^{n-1} 4 * \sum_{k=1}^{n-i} \frac{1}{k}$$
$$E[X] = O(n) + O(nlogn)$$
$$E[X] = O(nlogn)$$

Worst case number of comparisons would be same as in quicksort when one of the two subarrays that are generated has single element.

3. a) Consider the definition of k-sortedness. The defining inequality (for all i = 1, . . . , n − k) can be simplified by multiplying by k to give an equivalent condition:

$$\sum_{j=i}^{i+k-1} A[j] \leq \sum_{j=i+1}^{i+k} A[j]$$

for all i = 1, . . . , n − k
Observe that for any i = 1, . . ., n − k, we have

$$\sum_{j=i}^{i+k-1} A[j] \leq \sum_{j=i+1}^{i+k} A[j]$$

Or

$$A[i] + \sum_{j=i+1}^{i+k-1} A[j] \leq \sum_{j=i+1}^{i+k-1} A[j] + A[i+k]$$

Or equating both sides - $A[i] \leq A[i+k]$
Therefore an array is k-sorted if and only if $A[i] \leq A[i+k]$ for all i = 1, . . . , n – k

3. b)
We can k-sort an n-element array in O(n lg(n/k)) time as follows:
We use the fact from 3. a). The array is k-sorted if and only if A[i] ≤ A[i + k] for all i = 1, . . . , n − k. So we partition the array into k smaller arrays, sort each subarray and merge them.
K-SORT(n,k)

1. For each j = 1, . . . , k
2. For i = 0 to 1+ ⌊(n−j)/k⌋ // Length of Bj is 1+ ⌊(n−j)/k⌋
3. Bj = A[j + k* i] // Elements A[j], A[j + k], A[j + 2k],

4. End For
5. End For
6. For j = 1 to k
7. mergeSort(Bj) // Sort each Bj
8. End For
9. For j = 1 to k
10. Final array = merge(array, Bj) // merge each element from each subarray one at a time
11. End For

Complexity analysis : It takes O((n/k)lg(n/k)) time to sort each of the arrays, O(n lg(n/k)) time overall. Then, for each j = 1, . . ., k transfer the elements of the sorted array Bi into A[j], A[j+k], . . . in their sorted order. This takes O(n) time in total for all Bj arrays. Now the if and only if condition of 3.a is satisfied, so A is k-sorted. Total run time is O(n lg(n/k)).
Correctness : Correctness can be proved by proving that line 9 – 11 works. When merging is done, one element from each k subarrays are placed together followed by the next k elements from each subarray and so on. Since, each sub array is sorted, $A[i] \leq A[i + k]$ holds.

3.c)

Basic idea

1) Create k sub- arrays from the original array
2) Create a Min Heap of size k with first k elements from each array. This will take O(k) time
3) One by one remove min element from heap, put it in result array, and add a new element to heap from remaining elements of the list from which the element was taken.
4) Replace heap root with next element from the array from which the element is extracted. If the array doesn't have any more elements, then replace root with infinite. After replacing the root, heapify the tree.

Removing an element and adding a new element to min heap will take Logk time. So overall complexity will be O(k) + O((n-k)*logK)

Pseudocode

Sort-K-SortedArray(n, k)

1. For each j = 1, . . . , k
2. For i = 0 to 1+ ⌊(n−j)/k⌋ // Length of Bj is 1+ ⌊(n−j)/k⌋
   Bj = A[j + k* i] // Elements A[j], A[j + k], A[j + 2k],
3. End For
4. End For
5. For i=1 to k

```
6.      X[i] = Bi[1];
7.  End For
8.  While (k lists are not empty)              // Until all Bj is empty , for all j list where j = 1…k
9.      BUILD-MIN-HEAP(X)                      // Build min heap
10.     l=0
11.     Min = HEAP-EXTRACT-MIN(X)              // extract min from the heap
12.     Index = j                              // return index of node from X in list Bj
13.     TargetArray[l] = min                   // copy min in the target array
14.      l++
15.     IF Bmin is not empty
16.         MIN-HEAP-INSERT(X, Bmin(t++))      Insert next element from the list in to X
17.     End If
18.  End while                                 // do this until all list or heap is empty
```

Correctness: This can be proved by proving that min heap always returns the minimum element and the heap property is maintained since the elements added to the min heap are greater than equal to the remaining elements in the subarrays.

4 a)

The situation is very similar to the quicksort analysis, although k matters. zi and zj will be compared if one of them is the first element to get picked as a pivot in the smallest interval containing i, j and k. The exact expression depends on the position of k in regards to the other two:

$$E[X_{ijk}] = \begin{cases} \frac{2}{(k-i+1)} & if\ i < j \le k \\ \frac{2}{j-i+1} & if\ i \le k \le j \\ \frac{2}{j-k+1} & if\ k \le i < j \end{cases}$$

4.b) $E[X_k] = \sum_{i=1}^{n-1}\sum_{j=i+1}^{n} E[X_{ijk}]$

$= \sum_{i=1}^{k}\sum_{j=i+1}^{n} E[X_{ijk}] + \sum_{i=k+1}^{n-1}\sum_{j=i+1}^{n} E[X_{ijk}]$

$= \sum_{i=1}^{k}(\sum_{j=i+1}^{k-1} E[X_{ijk}] + \sum_{j=k}^{n} E[X_{ijk}]) + \sum_{i=k+1}^{n-1}\sum_{j=i+1}^{n} E[X_{ijk}]$

$= \sum_{i=1}^{k}\sum_{j=i+1}^{k-1} E[X_{ijk}] + \sum_{i=1}^{k}\sum_{j=k}^{n} E[X_{ijk}]) + \sum_{i=k+1}^{n-1}\sum_{j=i+1}^{n} E[X_{ijk}]$

$= \sum_{i=1}^{k-2}\sum_{j=i+1}^{k-1} E[X_{ijk}] + \sum_{i=1}^{k}\sum_{j=k}^{n} E[X_{ijk}]) + \sum_{i=k+1}^{n-1}\sum_{j=i+1}^{n} E[X_{ijk}]$

$= \sum_{i=1}^{k-2}\sum_{j=i+1}^{k-1} \frac{2}{k-i+1} + \sum_{i=1}^{k}\sum_{j=k}^{n} \frac{2}{j-i+1}) + \sum_{i=k+1}^{n-1}\sum_{j=i+1}^{n} \frac{2}{j-k+1}$

$= 2(\sum_{i=1}^{k-2}\sum_{j=i+1}^{k-1} \frac{1}{k-i+1} + \sum_{i=1}^{k}\sum_{j=k}^{n} \frac{1}{j-i+1}) + \sum_{i=k+1}^{n-1}\sum_{j=i+1}^{n} \frac{1}{j-k+1})$

$= 2(\sum_{i=1}^{k}\sum_{j=k}^{n} \frac{1}{j-i+1}) + \sum_{i=k+1}^{n-1}\sum_{j=i+1}^{n} \frac{1}{j-k+1}) + \sum_{i=1}^{k-2}\sum_{j=i+1}^{k-1} \frac{1}{k-i+1}$

$$=2\left(\sum_{i=1}^{k}\sum_{j=k}^{n}\frac{1}{j-i+1}\right)+\sum_{i=k+1}^{n-1}\sum_{j=i+1}^{n}\frac{1}{j-k+1}\Big)+\sum_{i=1}^{k-2}\frac{k-i-1}{k-i+1}$$

$$=2\left(\sum_{i=1}^{k}\sum_{j=k}^{n}\frac{1}{j-i+1}\right)+\sum_{j=k+2}^{n}\sum_{i=k+1}^{j-1}\frac{1}{j-k+1}\Big)+\sum_{i=1}^{k-2}\frac{k-i-1}{k-i+1}$$

$$=2\left(\sum_{i=1}^{k}\sum_{j=k}^{n}\frac{1}{j-i+1}\right)+\sum_{j=k+2}^{n}\frac{j-k-1}{j-k+1}\Big)+\sum_{i=1}^{k-2}\frac{k-i-1}{k-i+1}$$

$$\leq 2\left(\sum_{i=1}^{k}\sum_{j=k}^{n}\frac{1}{j-i+1}\right)+\sum_{j=k+1}^{n}\frac{j-k-1}{j-k+1}\Big)+\sum_{i=1}^{k-2}\frac{k-i-1}{k-i+1}$$

The last noted derivation is valid because of the following iversonian equation:

$$[k+1\leq i\leq n-1][i+1\leq j\leq n]=[k+1\leq i<i+1<j\leq n]=[k+1<j<n][k+1\leq i<j\,]$$

4.c) Let's take the expressions in parts. The last two are straightforward enough:

$$=\sum_{j=k+1}^{n}\frac{j-k-1}{j-k+1}+\sum_{i=1}^{k-2}\frac{k-i-1}{k-i+1}\leq\sum_{j=k+1}^{n}1+\sum_{i=1}^{k-2}1$$

$$=n-k+k-2$$

$$\leq n$$

The term $\sum_{i=1}^{k}\sum_{j=k}^{n}\frac{1}{j-i+1}$ contains terms of the form 1/m where 1≤m≤n. It contains 1/1 at most once, 1/2 at most twice, 1/3 at most three times and so on. Thus, the sum of the expressions 1/m for each m is at most 1 and there are n such different expressions, which bounds the whole sum to n.
Both expressions are at most 2n, which means that $E[X_k]\leq 4n$

The number of operations in RANDOMIZED-SELECT are linear to the number of comparisons, and the expected number of comparisons are bound by a linear function, which means that the expected running time is O(n).

EC1)
We can follow the same strategy as in Q1.

```
PARTITION(A, p, r, q):
    pivot = RANDOM(A) // pick a random interval as pivot
    SWAP(pivot, A[r])
    intersection = pivot

    // Find the intersection with the randomly selected element to
    // get the pivot (intersection)
    for i from p to r-1:
        if INTERSECTS(intersection, A[i]):
            if A[i].left > intersection.left:
                intersection.left = A[i].left
            if A[i].right < intersection.right:
                intersection.right = A[i].right
```

```
        s = p

        // Place intervals less than the intersection to the left of it
        for i from p to r-1:
                if A[i].right < intersection.left:
                        SWAP(A[i], A[s])
                        s++

        SWAP(A[r-1], A[s])

        t = s+1
        i = r-1

        // Cluster together intervals which intersect
        while t<=i:
                if INTERSECTS(A[i], intersection):
                        SWAP(A[t], A[i])
                        t++
                else:
                        i--

        return (s, t) // return an interval

FIND-MAX-OVERLAP(A, p, r, q):
        if p < r-1:
                pivot = PARTITION(A, p, r, q) // returns a range
                q = max(q, p-r+1)
                FIND-MAX-OVERLAP(A, p, pivot.left, q) // pivot.left is start of pivot's interval
                FIND-MAX-OVERLAP(A, pivot.right, r, q) // pivot.right is end of pivot's interval
```

As we know that pivot has set of all overlapping intervals, we update q every time we get a new pivot.