8-1

8-1.a

Each possible permutation corresponds to a order scheme, which is a leaf in the decision tree.

8-1.b

Same leaf, count down to the root of T has one more depth than root of LT or RT. There are k leaves in total, so the difference will be k.

8-1.c

Use the conclusion from 8-1.b. T can be made into a tree with i left leaves and k-i right leaves. Therefore, there is always $D(k) = D(LK) + D(RK) + k$ and you can vary i to get the minimum D(k) i.e. d(k). i is obviously not 0 otherwise there will be $d(k) = d(k) + k$

8-1.d

The derivative is

$$\lg \frac{i}{k-i}$$

Obviously reaches 0 when $i = \frac{i}{k}$ Dividing k evenly to left and right will inimize d(k), and so are left subtree d(i) and right subtree d(k-i), sub-subtrees and so on. So the minimized d(k) will be a tree that's evenly divided on every level. In total the height will be $\leq \lg(k)$ and total external path length will be klg(k). $d(k) = \Omega(k \lg(k))$

8-1.e

Use the conclusion from 8-1.d. T_A has $n!$ leaves. Expected time is

$$\Omega(n! \lg(n!)) \times \frac{1}{n!} = \Omega(\lg(n!)) = \Omega(n \lg n)$$

8-1.f

The randomized node can be trimmed down to its shortest possible deterministic subtree. This subtree is guaranteed to be better or at least equal to the randomized node performance.

8-2

8-2.a

Counting sort

8-2.b

```
    def binary_sort(A):
        index = 0
        for i in range(0, n):
            if A[i] == 0:
                swap(A[i], A[index])
                index += 1
```

8-2.c

Merge sort

8-2.d

All the stable ones can be used, i.e. either counting sort or merge sort

8-2.e

```
    def count_sort_in_place(A,k):
        C = np.zeros(k+1,dtype = np.int)
        for i in A:
            C[i] += 1
        for i in range(1,k+1):
            C[i] += C[i-1]
        swapped_index = np.empty([1,0], dtype=int)
        for i in range(np.shape(A)[0]-1, -1, -1):
            if i in swapped_index:
                continue
            else:
                while(i != C[A[i]] - 1 ) :
                    element_go = A[i] # this element will go to the right spot
                    A[C[A[i]]-1], A[i] = A[i],A[C[A[i]]-1]
                    swapped_index = np.append(swapped_index,C[element_go]-1)
                    C[element_go] -= 1
```

The idea is that with C in hand, given any element of A we can put it to the right spot. Instead of putting it to a new empty array, we swap it with the whatever element at that spot. All the elements that are moved to the right position are kept track in the array 'swapped_index'. If we met a loop index which is in this array, that represents that the element is the one we dealt before and we don't have to do anything with this element. We'll skip this one, so the worst case scenario is that we loop every element in A. Since we are using swapping, there is no stability with this algorithm.

8-3.a

First group the integers by their length and use counting sort to sort from the shortest to the longest. Then use raddix sort within each group. The time to group is O(m) where m is the number of the integers. The time for raddix sort in total is O(n). Since m is always less than n, sum up the two we have O(n).

8-3.b

Use raddix sort to sort from the left most letter. Group the strings by their first letter. After sorting, remove the first letter and recursively counting sort within each group. Treat vacancy as a letter that's lexicographically less than all other letters and groups of vacancy will be skipped. The method will go through all the letters once, so the time is O(n) in general.

8-4

8-4.a

Having one red jug, go through all the blue jugs until find the right one. In the worst case scenario each time you have to compare to the last jug. Total time cost is

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

8-4.b

Given an identical red jug array and a permutation of the blue jug array as input. Go through the comparing process. Each comparison is a node which has 2 branches: match or not. Each permutation outputs a unique matching result, meaning that there are at least $n!$ leaves in this decision tree. The height is therefore at least $\lg(n!) = \Omega(n \lg n)$.

8-4.c

Randomly choose a red jug R_i, loop through the blue jugs to find the blue jug B_i that match the volume. This process also divide the blue jugs into ones that are larger and ones that are smaller. Use the B_i to loop through the red jugs and divide them into smaller group and larger group. The partition and sort process are essentially two parallel quick sorting. So the expected running time is $O(n \lg n)$. Likewise, the worst-case scenario costs $O(n^2)$ time.

8-5

8-5.a

A is incrementally sorted.

8-5.b

1 3 2 4 6 5 8 7 10 9

5-8.c

Sufficiency: Given that for all $i = 1, 2, ..., n - k$ we have $A[i] \leq A[i + k]$ prove A is k-sorted. Compare the sum of any k consecutive elements. Since we always have $A[i] \leq A[i + k]$, obviously for any i we have $A[i] + ... + A[i + k - 1] \leq A[i + k] + ... + A[i + 2k - 1]$. On the other side, for subsets that have overlap with given subset, the substituted elements are always greater by k in index, making these subsets also larger subsets. So A is k-sorted. Necessity: Suppose there exists an i that $A[i] > A[i + k]$. Then the preceding subset $A[i]+A[i+1]+A[i+2]+...+A[i+k-1] > A[i+1]+A[i+2]+...+A[i+k-1]+A[i+k]$. A is not k-sorted.

8-5.d

Partition the array into k groups: A[1],A[1+k]...; A[2], A[2+k]...; ... A[k-1],A[2k-1].... For an array of size n, we will have n/k groups. Use quick sort or merge sort for each, time consuming is $\Theta(\frac{n}{k} \lg \frac{n}{k})$. So the total time will be $k\Theta(\frac{n}{k} \lg \frac{n}{k}) = \Theta(\frac{n}{k} \lg \frac{n}{k})$.

8-5.e

Merge the k sorted arrays. Take the first element of each array and form a reverse heap. Then remove the root (HEAP-EXTRACT-MIN, $O(\lg k)$) and insert the next element from the array which the removed MIN is from (MIN-HEAP-INSERT, $O(\lg k)$). Worst case this part will iterate for n times, which in total costs $O(n \lg k)$. Sum up the upper bound is $O(n \lg k)$

8-5.f

K-sort an array is essentially sorting k parallel arrays using comparison sort.

8-6

8-6.a

Obviously. We don't care the order. Different combinations give different arrays.

8-6.b

Best case scenario is that you can use the given order information stored in each array. In this case, you can compare the max of one array and min of another. If the max is smaller then you would only need one compare to sort them all. But this is not a general method. Merge sort is a more general method which you can draw decision tree. In the tree, the best case scenario is that each comparison one array is constantly larger, then in this case at least you can use the order information of the other, which saves you O(n comparisons. However, the worst case scenario is that each comparison gives a different output than last time. In this case, no given information is used and in total 2n comparisons are made.

8-6.c

Suppose the two elements, let's say, A_i and B_i that are consecutive in the sorted order are not compared. In the sorted order, A_i precedes B_i. Now we can swap A_i and B_i in the input two arrays, make A_i an element of array B and B_i in array A. Because they are consecutive in the sorted array, they should hold the same position in A and B with their counterparts. Now let's input these modified arrays into the decision tree, the process should be exactly the same. A_i and B_i are not compared, but this time, B_i precedes A_i. We cannot have $A_i \leq B_i$ and $B_i \leq A_i$ simultaneously. Therefore A_i and B_i must be compared to settle down their orders.

8-6.d

The worst case scenario is that all consecutive elements are one by one from opposite lists. Use the theory of 8-6.4, total 2n-1 comparisons are demanded.