1. (20 points) Professor Jones has proposed a new sorting algorithm (pseudocode is given below):

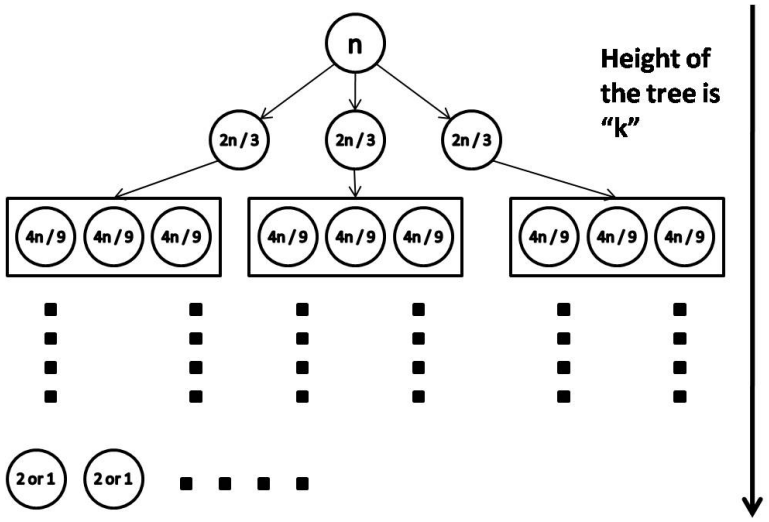
```
Jones_Sort (A, i, j)
  if (A[i] > A[j])
    A[i] ↔ A[j]                  //swap element i with element j
  if (i+1) ≥ j
    return
  k = ⌊(j-i+1)/3⌋
  Jones_Sort (A, i, j-k)         //first two-thirds
  Jones_Sort (A, i+k, j)         //last two-thirds
  Jones_Sort (A, i, j-k)         //first two-thirds again
```

1(a). (5 points) If `A[]={6, 3, 2, 5}`, show the array after each swap when `Jones_Sort(A, 0, 3)` is called.

Solution: Original array: 6, 3, 2, 5. At first function, it swaps element 0 and 3 with result: 5, 3, 2, 6. Next, in the first of the three recursive function calls, it swaps elements 0 and 2 with result: 2, 3, 5, 6. The remaining calls do not alter this array.

1(b). (10 points) Determine the worst-case time complexity of `Jones_Sort()` using `O()` notation in terms of the size of the array, `n`.

Solution: Worst-case is when a swap occurs during every function call. Each function call takes $O(1)$ time and then makes three recursive calls. So, the total time complexity can be found by finding the total number of function calls made. How many times does the function call itself? To determine this, let us examine the recursion tree. It looks like the following (with the value contained in each node representing the size of the sub-array during this recursive invocation).



Let the height of this tree be "k". At each level "i" of the tree, there are a total of 3^i nodes (you can verify this against the tree above). So, the total number of nodes in the tree will be

given by $(1+3^1+3^2+3^3+.....+3^k)$. This is a geometric progression, and its sum is given by $(3^{(k+1)}-1)/(3-1)$ = $O(3^k)$. Therefore, the worst-case time complexity is: $O(3^k)$.

We still need to find k though. To do this, consider how the problem size decreases at each recursive call. Since $2n/3$ can also be written as $(n)/(3/2)$, the problem size decreases according to the sequence: $n, n/(3/2), n/(3/2)^2, n/(3/2)^3, n/(3/2)^4, ...1$. Therefore, the total number of elements in this sequence is $O(log_{(3/2)}n)$. This is the height of the tree. Therefore, $k = O(log_{(3/2)}n)$. Therefore, time complexity of Jones sort is $O(3^{log_{(3/2)}n})$. We can simplify this expression further.

$$\begin{aligned} 3^{log_{(3/2)}n} &= 3^{(log_3 n)*(log_{3/2}3)} \\ &= (3^{(log_3 n)})^{log_{3/2}3} \\ &= n^{log_{3/2}3} \\ &= n^{2.709} \end{aligned}$$

1(c). (5 points) How does this compare with the worst-case of bubble sort and heap-sort? Is this an efficient algorithm in terms of time?

Solution: The worst-case time complexity for bubble sort is `O(`n^2`)` and for heap-sort is `O(nlog n)`. From the above, we can see that Jones sort is worse than both heap-sort and bubble sort.

2. (20 points) As we saw in class, the Mergesort algorithm relies on the `Merge()` procedure that merges two sorted arrays into a single sorted array. A commonly encountered problem is one of merging ‘‘`k`’’ sorted arrays into a single sorted array, where ‘‘`n`’’ is the total number of elements in all the input arrays together. Design the most efficient algorithm that you can to do this? (*Hint: Think about using a min-heap.*)

Solution: Initialize `k` pointers, each one pointing to the first (i.e., smallest) element of the `k` sorted arrays. Insert each of the elements pointed to by the k pointers into a min-heap (i.e., the min heap will have k elements in it). Extract the minimum value of the min heap and place it into the first position in the output array (because this element is the smallest of all the elements). If the element that was just extracted from the min heap came from array i, slide the pointer that points to an element in array i forward by one position. Insert the new element that this pointer points to into the min heap. Repeat the above procedure till all the elements are put into the output array.

Each operation on the min-heap is `O(log k)` and there are `O(n)` operations performed on the heap, so the total time complexity is `O(n log k)`.

3. (30 points)

4(a). (15 points) Illustrate the operation of Bucket Sort on the following array `A = {0.79, 0.13, 0.16, 0.64, 0.39, 0.20, 0.89, 0.53, 0.71, 0.42}`. You are free to pick the number of buckets you want as well as the function that puts data into the buckets (e.g., you can use the function given in the slides).

Solution: Pick 10 buckets (0 - 0.1 in bucket one, 0.1 - 0.2 in bucket two, and so on). After putting the data into buckets, the non-empty buckets look like:
Bucket 2: 0.13, 0.16
Bucket 3: 0.20
Bucket 4: 0.39
Bucket 5: 0.42
Bucket 6: 0.53
Bucket 7: 0.64
Bucket 8: 0.79, 0.71
Bucket 9: 0.89

Sorting each bucket and stringing the buckets back together results in the data: {`0.13, 0.16, 0.20, 0.39, 0.42, 0.53, 0.64, 0.71, 0.79, 0.89`}.

3(b). (15 points) Explain why the worst-case running time of Bucket Sort is $O(n^2)$. What simple change can you make to the algorithm to make its worst case running time $O(nlogn)$?

Solution: The standard Bucket Sort algorithm uses Insertion Sort to sort the individual buckets. In the worst-case, one or more of the buckets may have `O(n)` elements. In that case, the time to sort that bucket(s) using Insertion Sort would be $O(n^2)$.

To improve the time complexity, use Heap Sort (or any other sorting algorithm that has a worst-case time complexity of `O(nlog n)` instead of Insertion Sort to sort the buckets. This would decrease the worst case time complexity to `O(nlog n)`.

4. (30 points + 10 points extra credit) You are given a *sorted* array `A[]` that consists of `n` integer elements. You are also given an integer `z`. Design an algorithm to determine whether or not there are two elements in `A[]` whose sum is exactly `z`. What is the time complexity of your algorithm in terms of `O()` notation? **Extra credit:** You will receive **10 extra points** if your algorithm can do this in `O(n)` time.

Solutions:: Let $A[]$ contain the sorted sequence of numbers. For each element, $e \in A$, let $r = z - e$. Search for r in A using binary search. If $r \in S$ return true, otherwise return false. The running time of binary search is $O(\lg n)$. In the worst case, we perform binary search n times, requiring $O(n \lg n)$ time in all. Thus the running time of the algorithm is $O(n \lg n)$.
Extra credit: The algorithm is given below.

$i \leftarrow 0$
$j \leftarrow n-1$
while $(i < j)$ **do**
    if $(A[i]+A[j] < z)$ **then**
        $i \leftarrow i+1$
    else if $(A[i]+A[j] > z)$ **then**
        $j \leftarrow j-1$
    else
        return YES
return NO

The above algorithm exploits the fact that the array is sorted and sweeps the entire array using two variables that start from the left end and the right end of the array, respectively. If the sum of the values pointed to by the two indices is less than z, then increment the left index (since that is the only way to increase the sum). If the sum of the values pointed to by the two indices is greater than z, then decrement the right index (since that is the only way to decrease the sum). Continue until you find two indices whose values sum to z or the pointers touch each other. The algorithm runs in linear time as in each iteration of the loop, either i is incremented or j is decremented. Thus in n steps i and j would meet at which time the loop terminates.