Problem 4-1. (Counting Sort) Recall the counting-sort algorithm from class:

```
function counting-sort(A, B, k):
    initialize C[0...k] as an array of zeros
    for j=1 to A.length:
        C[A[j]] = C[A[j]] + 1
    for i=1 to k:
        C[i] = C[i] + C[i-1]
    for j=A.length down to 1:
        B[C[A[j]]] = A[j]
        C[A[j]] = C[A[j]] - 1
```

(a) What is the runtime of counting-sort?

Solution: $O(n+k)$, or $O(n)$ if $k = O(n)$.

(b) Argue that counting-sort is stable.

Solution: Suppose two elements at indices $i_1 < i_2$ are equal. In the sorted array, they will appear at indices $j_1 + 1 = j_2$. The last `for` loop puts $A[i_2]$ in $B[j_2]$ first and then $A[i_1]$ in $B[j_1]$. The two elements preserve their order in the sorted list. Hence, the algorithm is stable.

(c) Would the algorithm still be correct if the last `for` loop (`for j=A.length down to 1`) went forward? How would this affect the result?

Solution: The algorithm is still correct. The correctness argument we saw in lecture did not depend on the order in which elements were processed.

However, the sort is no longer stable. Consider the elements in the proof from the previous question. With the forward loop, the two elements at $A[i_1]$ and $A[i_2]$ are swapped in B. In fact, all duplicates will appear in reverse order in the sorted array.

(d) How is counting sort affected in practice if the input contains few integers over a large range (i.e. $k >> A.length$)?

Solution: It will require a lot of memory, and most will be unused (sparse C).

Also, counting sort runs in $O(n+k)$, so if $k = \omega(n)$, then k will dominate the runtime and the sort will no longer be linear.

(e) Is there a linear time sorting algorithm that addresses this problem?

Solution: Radix sort could outperform counting-sort for this case. Recall radix sort depends on the number of digits and their possible values, which could be far smaller than the full data range.

Also, bucket sort does not make assumptions about the range of data, only that the data's values are fairly evenly distributed. Hence, it might do better in practice than counting-sort on the $k >> A.length$ case.

Problem 4-2. (Heap Review)

(a) Give the runtimes for the basic max-heap algorithms we've covered: `Heapify()`, `HeapSort()`, `HeapMaximum()`, `HeapExtractMax()`.

Solution: $O(lgn), O(nlgn), O(1), O(lgn)$.

Recall that `HeapMaximum()` is just a getter, while the `HeapExtractMax()` function actually removes the heap element.

(b) At what **height** are the leaves of a heap?

Solution: The height of leaf nodes is 0.

(c) What is the maximum number of nodes at height h?

Solution: $\lceil n/2^{h+1} \rceil$

(d) What is the time complexity of `BuildHeap()`?

```
function BuildHeap(A):
    A.heapSize = A.length
    for j = A.length / 2 down to 1:
        Heapify(A, j)
```

Solution:

We call `Heapify()` on $\lceil n/(2^{h+1}) \rceil$ nodes per level. `Heapify()` is $\Theta(h)$ at height h.

$$\sum_{h=0}^{\log n} \left\lceil \frac{n}{2^{h+1}} \right\rceil \Theta(h) = \Theta\left(n \sum_{h=0}^{\log n} \frac{h}{2^h} \right)$$

Note that for $x < 1$:

$$\sum_{k=0}^{\infty} k x^k = \frac{x}{(1-x)^2}$$

We apply the above formula:

$$\begin{aligned} \sum_{h=0}^{\infty} \frac{h}{2^h} &= \frac{1/2}{(1-1/2)^2} \\ &= 2 \end{aligned}$$

Thus, we see that our expression for the time complexity above resolves:

$$\Theta\left(n \sum_{h=0}^{\log n} \frac{h}{2^h} \right) = \Theta(2n) = \Theta(n)$$

Problem 4-3. (Sorting In Linear Time)

Suppose that you are given n red and n blue water jugs, all of different shapes and sizes. All red jugs hold different amounts of water, as do the blue ones. Moreover, for every red jug, there is

a blue jug that holds the same amount of water, and vice versa.
Your task is to find a grouping of the jugs into pairs of red and blue jugs that hold the same amount of water. To do so, you may perform the following operation: pick a pair of jugs in which one is red and one is blue, fill the red jug with water, and then pour the water into the blue jug. This operation will tell you whether the red or the blue jug can hold more water, or that they have the same volume. Assume that such a comparison takes one time unit. Your goal is to find an algorithm that makes a minimum number of comparisons to determine the grouping. Remember that you may not directly compare two red jugs or two blue jugs.

(a) Describe a deterministic algorithm that uses $\Theta(n^2)$ comparisons to group the jugs into pairs.
Solution: . Compare each red jug with each blue jug. Since there are n red jugs and n blue jugs, that will take $\Theta(n^2)$ comparisons in the worst case.

(b) Prove a lower bound of $\Omega(n \lg n)$ for the number of comparisons that an algorithm solving this problem must make.
Solution: To solve this problem, an algorithm has to perform a series of comparisons until it has enough information to determine the mathings. We can view the computation of an algorithm as a decision tree. Every internal node is labeled with two jugs(one red, one blue) which we compare, and has three outgoing edges (red smaller, same size, red larger). The leaves are labeled with a unique matching of jugs.
The height of the decision tree is equal to the worst-case number of comparisons the algorithm has to make to determine the matching. To bound that size, let us first compute the number of possible matchings for n red and n blue jugs.
If we label the red jugs from 1 to n and we also label the blue jugs from 1 to n before starting the comparisons, every outcome of the algorithm can be represented as a set

$$\{(i, \pi(i)) : 1 \leq i \leq n \text{ and } \pi \text{ is a permutation on } \{1, ..., n\}\}$$

which contains the pairs of red jugs (first component) and blue jugs (second component) that are matched up. Since every permutation π corresponds to a different outcome, there must be exactly $n!$ different results.
Now we can bound the height h of our decision tree. Every tree with a branching factor of 3 (every inner node has at most three children) has at most 3^h leaves. Since the decision tree must have at least $n!$ children, it follows that

$$3^h \leq n! \leq (n/e)^n \Rightarrow h \leq n \log_3 n - n \log_3 e = \Omega(n \lg n)$$

Problem 4-4. (`Build-Max-Heap()` with insertion)

Consider a modification to `Build-Max-Heap()` that uses the `Max-Heap-Insert()` function repeatedly.

```
function Build-Max-Heap'(A):
    A.heapSize = 1
    for i=2 to A.length:
        Max-Heap-Insert(A, A[i])
```

(a) What is the worst-case running time to build an n-element heap with this algorithm?

Solution: $\Theta(nlgn)$

(b) But the normal algorithm has runtime $O(n)$! Why is this algorithm different?

Solution: In the normal algorithm, most nodes actually don't "bubble down" very far, even in the worst case. (The sum in Problem 2 illustrates why.) However, the insert-based version forces every node to the bottom of the heap, and then those nodes potentially have to "bubble" all the way up the heap.

(c) Let $A = [3,2,1,4,5]$. Trace through each algorithm and state the two outputs.

Solution: `Build-Max-Heap` will return $[5,4,1,3,2]$. The modified algorithm will return $[5,4,1,2,3]$.