**Radix-sorting.** Suppose we refactor LSD sort so that it uses a different sorting algorithm other than key-indexed counting. For each of the algorithms below, state whether or not LSD would work if that sort was used instead of key-indexed counting. Why?

**Any stable sorting algorithm will work. Thus, mergesort and insertion sort are yes, and the others are no.**

**Mergesort:** \_\_\_\_\_\_\_\_**, Quicksort:** \_\_\_\_\_\_\_\_**, Heapsort:** \_\_\_\_\_\_\_\_**, Insertion sort:** \_\_\_\_\_\_\_\_\_

**LSD vs. MSD**. Why is it that LSD can re-use one big count array, but MSD needs D count arrays, (where D is the depth of the recursion)?

**We use these count arrays to track the locations of our recursive subproblems. We can avoid this memory cost by adding code that scans for the next subproblem. This will slightly increase the runtime and save us memory (though memory usage will still be proportional to D, as is the case with any recursive algorithm).**

**MSD vs. 3-way radix quicksort.** The fundamental difference between MSD and 3-way radix quicksort is simple (once you’re quite comfortable with the material at least): In MSD, we use key-indexed counting and recurse on the resulting subproblems. In 3-way radix quicksort, we use 3-way partitioning and recurse on the resulting subproblems. Both of these basic operations are linear time, but key-indexed counting brings us closer to our goal (since it actually sorts instead of just partitioning) – this results in having to reanalyze the same character of the same key potentially as many as R times! Succinctly state why partitioning instead of sorting ends up being a net win despite this fact.

Even though key-indexed-counting is linear time, it is much slower than partitioning. Partitioning doesn’t involve count or aux arrays and has better cache performance. Also, we generate only 3 subproblems instead of N subproblems (saving memory, which is important if we keep count arrays).

**Fattest path.** Given an edge-weighted digraph and two vertices s and t, design an algorithm to find a fattest path from s to t. The bottleneck capacity of a path is the minimum weight of an edge on the path. A fattest path is a path such that no other path has a higher bottleneck capacity.

**First create a pathExists(T) algorithm that determines whether or not a path exists of fatness T. We can do this by simply removing all edges of weight less than T, then running BFS from s to t, taking E+V time. Given this routine, we then need to simply perform a binary search on our edge weight values. Sorting the edge weights is time ElogE, and running log(E) pathExists is also ElogE.**

**2-sum.** Given an array of N 64-bit integers and a target value T, determine whether there are two distinct integers I and j such that . Your algorithm should run in linear time in the worst case.

**We sort the integers in linear time using LSD (or one of our other string sorting algorithms). We then apply the trick from one of our first precepts, starting with two pointers on either end of the sorted array. If the sum of the pointed-to-numbers is < T, we move the left pointer to the right and vice versa.**

**Manichaean heteronormative dystopian mate assignment problem.** Suppose we have N men and N women. Each person has M binary attributes (tall vs. short, hirsute vs. hairless, etc). Each person specifies their desire for each attribute. Design an algorithm to find a perfect matching such that the most unlucky person gets no more than B bad attributes, where B is some specified constant.

***For an extra challenge*:** Let X be the number of bad attributes that the most unlucky person has to deal with. Design an algorithm to find a perfect matching such that X is minimized.

**We convert this into a flow problem where we draw an edge from person X to person Y with ‘weight’ equal to the degree of X’s disapproval of Y and capacity 1. For example, if X wants traits short, hirsute, and clean, and person Y is tall, hirsute, and dirty, then we’d draw an edge of weight 2.**

**To solve the first problem, we use the same idea from the fattest paths problem, throwing away all edges of weight greater than B, and run Ford-Fulkerson to see if we find a flow with value equal to N.**

**To solve the second problem, we simply binary search using the algorithm from the first problem (similar again to the fattest paths problem).**

**Alphabet selection.** Warning: I got a bit stuck on this yesterday. Good luck! Given some fixed N and W, explain a strategy for selecting an alphabet size R to minimize the number of array accesses used by LSD. Extra challenging: Find the alphabet size that minimizes the number of array accesses for N=1,000,000,000, W=9.