10/10

Dan Ports
6.046 PS2-1

Your TIM
Collab: ∅

a. Algorithm: This is a variant on randomized quicksort. We randomly select a pivot bolt and compare every nut to it in order to partition the nuts into a set A of nuts too small for the bolt, and a set B of nuts too large for the bolt, as well as the one nut that fits the bolt. We can then use that one nut to partition the bolts into the set A' that are too small for the nut, and the set B' of bolts that are too large. We can then recursively apply the algorithm to the nut set A with bolt set A', and then B with B'.

Proof of correctness: use QS

Given a set of one nut and bolt, it is trivial to match the nut and bolt. By induction, we can show that it is correct for any set. Suppose it works for a set of size $\lceil n/2 \rceil$ and $\lfloor n/2 \rfloor$. Then we can take a set of size n, randomly select a pivot bolt, and partition it using the procedure above. Each pair of sets (A A') and (B B') contains either $\lceil n \rceil$ or $\lceil n/2 \rceil$ nuts and bolts, and each nut in each set is matched with the appropriately sized bolt: all of the nuts too small to fit on the pivot bolt are matched with the smaller bolts, and likewise for the larger nuts and bolts. From the inductive hypothesis, the algorithm will work for the two subsets, and we know that it correctly matches the pivot bolt to its nut. Thus it works in the n nuts and bolts case and, by induction, it works for all n.

Runtime analysis: The algorithm has expected runtime $\Theta(n \log n)$. This is clear by comparison to randomized quicksort. The algorithm randomly selects a pivot, partitions the problem into two subproblems, and recursively applies to itself exactly like quicksort. The only difference is that partitioning takes twice as much work, but this is just a constant factor, and hence since randomized quicksort has expected worst-case runtime $\Theta(n \log n)$ (CLRS p. 156), so does this algorithm.

b. Given any fixed ordering of the bolts, this problem is isomorphic to finding the appropriate permutation of the nuts. There are $n!$ possible permutations, each of the $n!$ possible inputs must map to a different output permutation of the nuts, so the decision tree must have $n!$ leaves. Each comparison has only three possible outcomes, so the height of the tree must be at least $\log_3(n!) \Rightarrow \log_3((n/e)^n)$ asymptotically $\Rightarrow n \log_3(n)$. So any algorithm that provides correct output must be $\Omega(n \log n)$. If it does not, it will map two inputs to the same output permutation, giving incorrect results.

Dan Ports
6.046 PS2-2

14/15

4oav 2PM
Collab: ∅

CLRS 2-4

a. (1,5) (2,5) (3,5) (4,5), (3,4) 0p

b The worst case is when the array is in reverse-sorted order. Then every pair of elements will be an inversion. This is $\binom{n}{2}$ total inversions ✓ 2p

c Insertion sort will run in $\Theta(n+k)$ time, where k is the number of inversions. It is clear that n will be a factor because the outer loop of insertion sort runs n times. The inner loop swaps one element with the previous one if they are out of order. This eliminates one inversion. Thus each time the outer loop runs, the inner loop removes all inversions containing the selected element. To be sorted, all k inversions must be removed. New inversions are never added, and inversions are only removed by the swap in the inner loop. So the inner loop runs k times total ✓ 3p

d. This can be done by modifying mergesort. We keep a growing counter of the number of inversions found so far. In the Merge procedure, whenever we take an element from the right array, we know that it is less than every remaining element in the left array, so we increment the number of inversions by the number of remaining elements in the left

array. (vague, you may want to be more precise).

Pf of correctness: By induction. We split an array of size n into two arrays of size $\lfloor n/2 \rfloor$ and $\lceil n/2 \rceil$. Inductively, we suppose that we can sort and count inversions in the left and right subarrays. Then the only inversions yet to be counted are the ones with one element in each array. Any element in the right array was originally to the right of the ones in the left array, so for each element in the right array, we have one inversion per element in the left that is larger. We count this by noting that when we select an element from the right array to be added to the merged list, it is less than any elements from the left that have not yet been selected, so we increment the number of inversions by the number of remaining elements on the left. Once we have done this for every element on the right, and we add in the previously-found number of inversions in each subarray, we have the number of inversions of a size n set. ✓

Runtime analysis: This algorithm has runtime $\Theta(n \lg n)$. It is identical to mergesort except for the code that increments the counter. But this only affects the constants, it clearly does not change the asymptotics, so the algorithm has the same runtime as mergesort. ✓ 9p

10/10

Dan Ports
6.046 PS2-3

Yoav ~2PM
Collab: ∅

a. Each matrix-vector product can be performed using n^2 multiplications and n additions (a linear combination of n vectors of length n). To perform most of these takes m times as long: it is $\Theta(mn^2)$

b. We can perform this multiplication faster by using a variant on the exponentiation algorithm, noting that

$$A^m = \begin{cases} (A^{m/2})^2 & , m \text{ even} \\ A(A^{\frac{m-1}{2}})^2 & , m \text{ odd} \end{cases}$$

From the fast-exponentiation algorithm analysis, or by setting up a recurrence, we know that this requires $\Theta(\lg m)$ multiplications. These are matrix multiplications which can be performed in time $O(n^{\lg 7})$ by Strassen's algorithm. So we can find the matrix A^m in time $O(n^{\lg 7} \lg m)$. We then need to multiply A^m by x, a matrix-vector product requiring time $\Theta(n^2)$, but n^2 is asymptotically smaller than $n^{\lg 7}$, so it is not significant. The entire algorithm runs in time $O(n^{\lg 7} \lg m)$.

This algorithm will be asymptotically faster than the first when $n^{\lg 7} \lg m < mn^2$

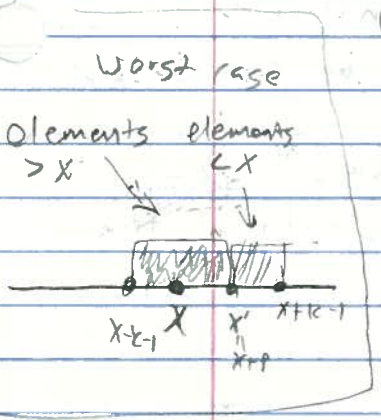
$$\Rightarrow \frac{m}{\lg m} > \frac{n^{\lg 7}}{n^2} \Rightarrow \boxed{\frac{m}{\lg m} > n^{(\lg 7 - 2)}}$$

Dan Porty
6.046 PS2-4

(14/10)

Your ZPM
Collab: Ø

a. From problem 2, we know insertion-sort requires time $\Theta(n+m)$, where m is the number of inversions in the array. Consider an arbitrary element of the array whose sorted position is x. Call its position in the unsorted array x' and note that $x' \in [x-k, x+k]$. There will be an inversion between x and another element y if $x<y$ and $y'<x'$ or $y<x$ and $x'<y'$. The set of elements whose value is greater than x can lie in the interval $[x-k+1, n]$ in the unsorted array, and the elements whose value is less than x can lie in the interval $[0, x-k-1]$. So if we let $x' = x+p$, for $-k \le p \le k$, then the worst case is when as many elements whose value is less than x lie in positions greater than $x+p$, and as many elements whose value is greater than x lie in positions less than $x-p$. So there can be elements greater than x in positions $[x-k-1, x+p-1]$ and elements less than x in positions $[x+p+1, x+k+1]$. This is a total of $(p+k+1)+(k+1-p) = 2k+2$ inversions. This is $O(k)$. (This is in fact an upper bound, since there are less inversions if we are near one end of the array but this can be neglected.)



There are a total of n elements that can be chosen as x, so the total number of inversions is $O(kn)$. Hence the runtime of insertion-sort is $O(n+kn)$
$= O(kn)$

5. Consider the subset of inputs generated by dividing the sorted array into blocks of length k and permuting them. These inputs clearly satisfy the requirement that the array is k-almost-sorted with k. The total number of inputs is bounded below by $(k!)^{\lfloor n/k \rfloor} = \Omega((k!)^{n/k})$ because there are more than $\lfloor n/k \rfloor$ blocks that can have $k!$ permutations. The sorting algorithm must output a unique output permutation for each input, since all inputs are distinct, so the total number of outputs must also be bounded below by $(k!)^{\lfloor n/k \rfloor}$.

Each comparison can produce at most 3 results, so the height of the worst-case decision tree, which is the number of comparisons that need to be performed in the worst case

$$= \Omega(\lg[k!)^{\lfloor n/k \rfloor}]) = \Omega(\lg[(k!)^{n/k}])$$
$$= \Omega(\lg[(\sqrt{k}(\tfrac{k}{e})^k)^{n/k}]) \text{ via Stirling's thm}$$
$$= \Omega(\lg(k^n))$$
$$= \Omega(n \lg k)$$

Since this particular subset of inputs requires at least $\Omega(n \lg k)$ time, any algorithm that handles the full set of inputs clearly must be $\Omega(n \lg k)$.