This homework is due at the start of class on Friday, February 22. In all cases, remember to show your work and justify your answers.

1. In class, we discussed a "tournament" approach to finding the maximum in an array. Consider the following recursive version of that algorithm: Divide the array into two halves, run a tournament on each half to find the maximum element in that half. Then compare the two winners to find the overall maximum in the array. Write a recurrence relation for the run time of this algorithm, and use the Master Theorem to solve that recurrence relation. (This is a short exercise, and the answer should of course be $\Theta(N)$.)

2. Let $A[0], A[1], \ldots, A[N-1]$ be an array such that the first $N - \sqrt{N}$ elements are **already** sorted into non-decreasing order. Nothing is known about the last $\sqrt{N}$ elements.

a) Insertion sort has better performance when applied to an array that is already almost sorted. If you apply *Insertion Sort* to the array in this problem, what will its worst-case run time be? (Why?)

b) Devise an algorithm that sorts the array with run time $\Theta(N)$. (Hint: You can use extra space. You might encounter an unusual run-time function.)

3. Heaps are often used in algorithms that require repeatedly taking the maximum or minimum of a set, over and over. This program gives an example. Suppose that you have K lists that are **already** sorted into increasing order. The total number of items in all the lists combined is N. The goal is to merge the K sorted lists into a single sorted list of length N. You can do something similar to the merge part of merge sort. That is, find the list that has the smallest first element, remove that element, and place it at the end of the big sorted list.

a) It is easy to determine which list contains the smallest first element with an algorithm that has run time $\Theta(K)$. Describe an algorithm for the list-merging problem that uses this idea, and explain why the total run time for the algorithm is $\Theta(N * K)$ in the worst case.

b) Now, find another algorithm that solves the merging problem with worst-case run time $\Theta(N * \log(K))$. Use a MinHeap to speed up the algorithm from part **a**. The items in the heap can be lists. Explain why the run time is $\Theta(N * \log(K))$.

4. We have seen how dynamic arrays enable arrays to grow while still achieving constant time amortized performance. This problem concerns extending dynamic arrays to let them both grow and shrink as items are added and removed over time.

a) Consider an "underflow" strategy that cuts the array size in half whenever the number of items falls below half of the length of the array. Give an example sequence of insertions and deletions where this strategy gives a really bad amortized cost.

b) Then give a better underflow strategy than the one suggested in part **a)**. Your strategy should be one that achieves constant amortized cost per deletion.

5. Consider this problem: Given an array of N different numbers, find the M largest values in the array, in sorted order. (You might, for example, be looking for the best score, second best score, third best score, ... M-th best score, in a video game.) Consider the following three algorithms for solving this problem:

a. Sort the array, then get the M numbers from the last M places in the array.

b. Build a max-heap from all the values in the array, and call *removeMax*() M times on the heap.

c. Use an order statistic algorithm to find the $(N-M)$-th smallest number. Use that number as a pivot in the Quicksort partition algorithm. Then sort the M numbers that end up to the right of the pivot.

Give a run time analysis of each algorithm, in terms of N and M. If you think of M as being a constant, which algorithm would you expect to be asymtotically fastest (that is, fastest at least for very large values of N)?

More ideas for projects. Here are some general ideas for things to look at as you are thinking about term projects. Some of these are covered in the textbook, some you can search on Google. Many of them are important algorithms that are used out in the real world. (And check out https://en.wikipedia.org/wiki/List_of_algorithms) You do not necessarily need to implement or even use an algorithm—you could just discuss what it does and how it is used.

Fast Fourier Transform, for signal processing.

Data compression algorithms, for compressing audio and video.

Numerical methods for solving differential equations, for modeling.

Linear Programming, for resource allocation.

Algorithmic stock trading.

Steganography, for hidden messages.

Encryption algorithms.

Digital signatures, and blockchain.

Stable Marriage Problem.

Geometric algorithms, such as convex hull and closest pair of points.

A^* algorithm, for path finding.

Page rank, Google's original algorithm for ranking search results.

Artificial Intelligence algorithms.

Computer Graphics algorithms, such as ray-tracing and collision detection.

Fractal algorithms, such as finding fractal dimension or the Chaos Game.

Primality testing algorithms.