**10 practice problems**

Some of these problems would be quite challenging on an exam, or have solutions that are simply too lengthy for an exam. Others are relatively easier. They are all good to get you thinking about the material.

1. Someone hacks into your sorted array of *n* distinct elements, and randomly displaces one element (shifting everything in between its initial and final position). For example, displacing 2 in 1,2,3,4,5 and positioning it between 4 and 5 would result in 1,3,4,2,5. Between standard Insertion-sort and Merge-sort, which would you use to minimize the worst-case time of getting the array back in sorted order? Why?

Insertion-sort is O(n2) based on the worst case scenario. In order to find the displaced element, the algorithm has to travel across the n-size array. Upon discovering the first unsorted number, it will have to perform j checks between the sorted array and the unsorted element and swap them. There will be n comparisons done to check the array, but when the unsorted number is found, it will have to swap j times to return back to its original position. This would be around 2-3 swaps necessary to restore the original position

Merge-sort has O(n log n) time based on worst case scenario which is a lower time complexity than insertion-sort. Merge sort will break up the array as many times as needed and reassemble the array by comparing the value and merging them back together. The best and worst-case scenario of merge-sort would be O(n log n) because it will have to check the array of size n and merge it back together in O(log n) time.

1. The analysis of Merge sort in the course notes assumes that *n* is even. Write down a recurrence for Merge sort that is more accurate, without this assumption. Without actually analyzing the more accurate recurrence directly, explain why the time complexity is still *O*(*n*log*n*).

*In other words, don’t use induction or trees, instead use one of my favorite techniques.*

* For merge sort, each array is split into two smaller arrays, creating two leaves for every parent array: 2T(n/2)
* Even arrays can be split in half
* Odd-sized arrays will have to split where one array will contain an additional element until it reaches either a size of 1 or an even sized array.

It is still n constant time. Because n/2 is <= n and we’re checking for n log n time complexity, we can exaggerate both n values to just be the size of n/2

The end result is that T(n) is still O(n log n) time complexity because the combined exaggeration of n/2 is still less than or equal to n. Having an odd-sized array doesn’t greatly affect the time complexity of splitting and merging an array together.

1. Given a list of *n* distinct real numbers and a value *k*, how can you report all pairs in the list that have a difference of *k*? Find two different ways to solve this.

The slow method is to use a form of selection sort where you go through the list of n from the beginning. For each number encountered, search the rest of the list for a number that yields a difference of k. When finding that number, mark off the pair of numbers being used and check for the next distinct real number. This would be O(n2) complexity.

The alternative method is to treat the list of n as though it is a tree with hashes. Create a hash table that stores all the n values and an output table where the pairs would be put together. Build the heap tree through the recursive heapify function to create a complete valid heap tree of n numbers. To search for any difference of k, start with the root node at n then check the largest of the two children of the root node. If the difference is less than k, then check children of the left child node. If the difference is greater than k, check the children of the right child node.

Repeat the process of checking the children until a node that contains the proper difference has been found. Remove both the root and the matching node from the heap table and place it into its own output table. Upon removing the two nodes, reheapify the heap tree and repeat the process with the next root node until all elements have been paired up. When all elements are paired up, the list of distinct pairs has been found within O(n) time because the heap procedure requires only O(n) time.

1. You are given an array *A*, containing *n* real numbers without duplicates. For every pair of elements, we say that the pair is “in order” if the smaller value is found somewhere to the left of the larger one. For example, in the array [13*,*11*,*14*,*12] there are three pairs that are in order: {13*,*14}, {11*,*14}, {11*,*12}. Show how to find the number of such pairs in *A*, in *O*(*n*log*n*) time.

There is a form of pattern where in the pairs of (a, b) and (b, c) that exhibits the transitive property. If (a, b) and (b, c) then that must also mean (a, c) because a < b and b < c meaning that a must also be a < c.

Using divide-and-conquer, split the array until there is a pair of two elements in the broken-up arrays.

1. You have *k* sorted lists, each of size *n*. How fast can you sort all this data, in terms of *k* and *n*?
2. Show that we don’t really need to worry about the Case 3 technicality of the master method (see course notes), if *f*(*n*) is *nk* or 2*n* or log*k n*. Here, *k* is a positive constant.
3. *This problem is described in section* 4*.*1 *of CLRS (p.68). Here’s a summary:*

You are given an array A of *n* real numbers. You get to select two indices of A, *i* ≤ *j*, and your score will be the sum of all values stored between A[i] and A[j] (inclusive). For example if you select 3 and 5, your score will be A[3] + A[4] + A[5].

* 1. The book says that a brute-force algorithm would run in quadratic time. I disagree.

What do you think brute-force would be?

* 1. Can you solve the problem in quadratic time?
  2. Can you solve the problem in *O*(*n*log*n*) time?

1. Considering the two heap-build methods covered in class:
   1. Given an array with distinct elements, will both methods give the same heap, not give the same heap, or does it depend on the input? Justify your answer.
   2. We say an algorithm is *stable* if any pair of elements with equal value appear in the same order in both the input and output. Assume that both heap-building methods will not swap elements that have equal value. Which heap-building method is stable, if any? What about the extraction phase of heapsort that follows heap-building?
2. Suppose that you have *n* real numbers, listed in groups of size *k*. The groups are already in “sorted” order in the sense that every number from one group is smaller than every number from the next. However within each group nothing is sorted. For example, [3,6,2,5],

[12,15,11,19], [25,21,28,27], etc.

You must return all numbers in sorted order.

* 1. What would you do to sort this input, and what would the time complexity be?
  2. How many possible outputs (permutations of the input) are there?
  3. Provide a lower bound for the worst case time complexity of fully sorting the input, usingyour answer from (b). Do this without Θ-notation.
  4. Convert the answer that you get in (c) to an expression that is more “user-friendly”,using Θ-notation.

1. Provide your own binary decision tree that can sort any 5 distinct numbers, with the smallest possible depth.

*Note*: this tree will be large, so if there are similar subtrees, just draw one of them and explain that others are similar. For instance if two subtrees have exactly the same structure, but their indices are a permutation of each other, then it suffices to provide one subtree and the permutation that encodes the similarity. For example, see Figure 1.

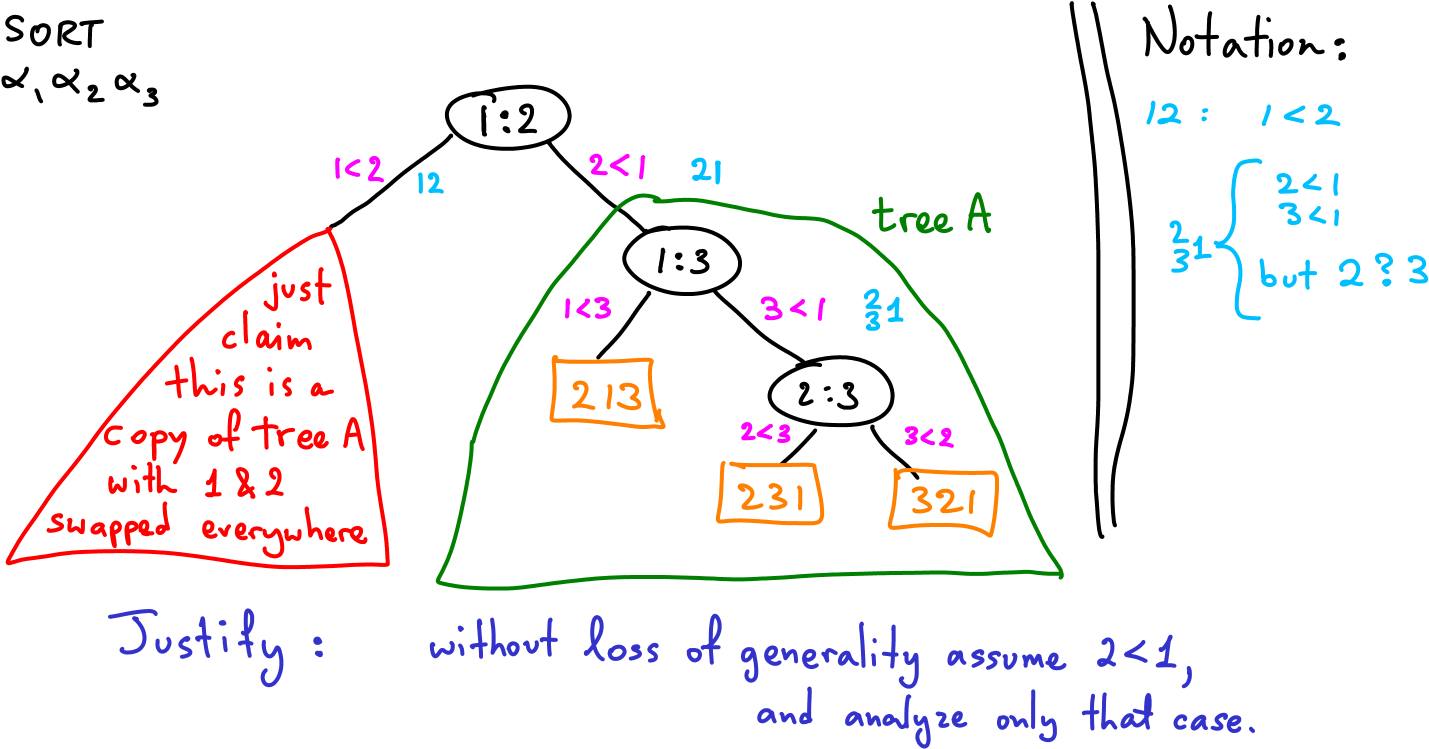


Figure 1: Some warmup and notation that can be used for the problem of sorting 5 numbers.