CS4Teachers Data Structures and Algorithms for K-12 Education: Sorting[*]

Robbie Weber

Abstract

Data structures and algorithms is about finding better ways of getting things done. A very common thread in elementary data structures and algorithms is that sorted data is easier to use than unsorted data. In this document, I'll outline ideas for activities related to data structures and algorithms for students from elementary school through high school. With each activity, I have listed the main takeaway for students.

1 Why Does Sorting Help (All ages!)

Takeaways Intuition for binary search. Along with the idea that if something is in order, you know where to look for it. Hopefully, the ability to search for things in real-life (say for books in a library) a little bit more efficiently.

Activity Take 20 notecards, and label each of them with one of the numbers from 1-20. Show the students the notecards, then place 15 face-down, and keep 5 in your back pocket (without letting the students see which are in which group). Ask the students to determine whether one of the cards is in your pocket or on the table by turning over the fewest number of cards on the table.

For this first round, the students will have no information about which cards are where until they are flipped over. There is no strategy that works better than the obvious one: flip over any card until you find the one you're looking for, or until all on the table are flipped (in which case the card can only be in the teacher's pocket).

You may want to try one round with the card in your pocket and one on the table.

Now, it's time to see the benefits of sorting. Repeat the setup, but place face-down cards in sorted order, and promise the students that the list it is sorted. I'd recommend putting a marker "small numbers" on one end and "big numbers" on the other. With the same goal, suggest they flip over middle card. Ask where desired card could be/couldn't be. Help them realize that they don't have to flip over one half of the cards to know that the target card is not there.

[*]This document gives more details on three possible activities involving data structures and algorithms content for K-12 education. It was originally written for a session as part of the CS4Teachers summer program.

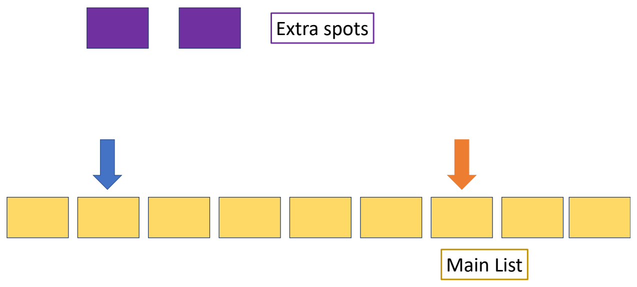
Depending on age of your students, you might want to continue having them take middle card or see if they come up with idea.

Real world uses Follow-up the activity with a trip to the library, and look for a particular book. What does it mean to be "in the middle" now? Will it be faster than just checking each individual book?

2 How Do We Sort? (Middle School and up)

Have a mat with ten[1] marked spots for cards in a line, and two extra marked spot above the line of ten. Have each student make 10 notecards, each with a different integer on them[2]. Your students will start with the cards (unsorted) in the 10 spots, and execute various algorithms to sort the list. Humans will try to take shortcuts or make intuitive "jumps" to sort faster – that's normal! But it's not something that computers can do. We give computers the steps to do, and it does them all. No shortcuts (except those we clearly define in the algorithm).

To sort a little more similarly to the way computers do, your students will have to follow these rules: You can only touch one card at a time. Cards have to be in you hand (only one) or in one of the 10 spots of the array (the "main list" of numbers) , or in one of the two extra spots.



The sorting humans can do under these rules match very closely to what computers can do with a single "temporary" variable Have students execute these sorts:

[1]Or some slightly larger/smaller number, depending on their age.

[2]The exact numbers do not really matter, though I'd recommend not just using numbers 1-10. If you do, students can too easily predict the final spot of each of the numbers.

2.1 Selection Sort

Students will make a “sorted area” on the left, and an “unsorted area” on the right. At the start, the sorted area is empty. Place an arrow to the left of the first card to mark the border.
Until the “sorted area” is the full line:

1. Place the first (i.e., left-most) element of the unsorted area in the first extra spot.
2. Moving from left to right, if an element in the main area is smaller than the one in the extra spot, swap[3] them.
3. When you reach the end of the main area, take whatever is in the first extra spot, and place it in the empty location in the array.[4].
4. If you’ve done it right, your “sorted area” is now one spot larger. Move the arrow one spot to the right.
5. If your arrow is at the end of the list, you’re done! Otherwise go back to step 1.

2.2 Insertion Sort

Students will again make a “sorted area” on the left, but will do it differently with this algorithm. Start with a single card as your sorted area, so place an arrow between spots one and two. Until the sorted area is the full line:

1. Move the first element from the unsorted array to the first extra spot (this is the card you will insert).
2. Compare the element in the extra spot to the current “gap” in the sorted area – if the extra spot card is less than the card to the left of the gap, move the gap to the left (by moving the card to the left of the gap one spot to the right).
3. Repeat the step above until the gap is all the way at the left of the main list, or the card in the extra spot is greater than the one to the left of the gap. At that point, move the card from the extra spot down to fill the gap.
4. If you’ve done it right, your “sorted area” is now one spot larger. Move the arrow one spot to the right.
5. If your arrow is at the end of the list, you’re done! Otherwise go back to step 1.

[3]swapping is hard when you can only have one in your hands! You’ll have to use the second extra spot as a temporary location to perform the swap

[4]if you’ve swapped correctly, it will be the smallest number that isn’t already in the sorted area

2.3 Quicksort

A recursive algorithm. Start with the full list.

1. Place the leftmost card in the first extra spot[5]. Place the blue arrow on spot 2 (i.e., pointing directly at the first remaining element in the list) and the orange arrow on the last spot.
2. Until the orange and blue arrows point to the same spot: look at the card the blue arrow is pointing to:
    - If the arrow is pointing to a smaller card than the card in the first extra spot, move the blue arrow one spot to the right.
    - Otherwise, Swap the card the blue arrow is pointing to with the card the orange arrow is pointing to, then move the orange arrow to the left.
    - Repeat this step.
3. When the orange and blue arrows point to the same spot, compare the pointed to card and the card in the extra spot:
    - If the pointed to card is larger, take the card to the left of the arrows, and move it to the gap at the far left. Then move the card in the extra spot to the newly-created gap.
    - If the pointed to card is smaller, move the pointed to card to the far left. Then move the card in the extra spot to the newly-created gap.
4. If you've followed these steps correctly, the area you just worked on is now: everything smaller than the pivot (possibly not in order), the pivot (that is, the card that spent the algorithm in the first extra spot), then everything larger than the pivot (again, not necessarily in order).
5. Now, repeat this full algorithm on the left-side only (that is go back to step 1), pretending the pivot and everything larger don't exist.[6]
6. Then, repeat these steps on the right-side only.

[5]This card is usually called the "pivot"

[6]An algorithm like this, that refers to itself, is called "recursive." You'll probably want to keep track of which areas have and haven't been sorted, as you'll have a few "levels" of recursion waiting to be done.

2.4 Merge sort

Execute as a class! Give most of your students[7] numbers to wear, and place them into individual "lists" of size 1. (You might want spots marked on the ground for this). Then pair the lists, and for every pair of lists execute this "merge" step:

1. Each list starts sorted smallest to largest.
2. Until one list is empty, look at the smallest element remaining in each list, take the smaller of the two and place them in the lowest open spot in the merged array.
3. When one list is empty, but the other isn't, take someone from the non-empty array.
4. The new list is sorted in order!

Repeat the "pair lists off and merge" process (you'll halve the number of lists each time) until you have a fully sorted list.

Merge sort is a great example of a *parallelizable* algorithm – if you have more computers, you can sort faster.[8] Once you've run the algorithm with just one person doing all the merging one at a time, try it again with (say) 4 "computers" each taking 1/4 of the lists until there are 4 lists left (at the end, have two people merge from 4 lists to 2, and one merge from 2 to 1). See if this version goes any faster.

3 A Data Structure for Sorting (high school students)

Why do programmers use data structures? We saw with binary search (the algorithm in section 1) that having sorted data makes it easier to find what you're looking for. A data structure will make it easier to maintain that being-sorted-benefit under updates (can we still use the data structure after we've added or removed many elements) – if you add something to a sorted list, you have to put it in the correct spot or the list won't be sorted anymore!

Our scenario is choosing the next TikTok to show to a user; our job is to maintain a set of all the videos we might recommend, where each video has a "priority" – a score that corresponds to how much we think our user will like the video. The higher the number, the more we want to show the video.

[7]preferable, a power-of-two number of students

[8]something like selection sort isn't – it's hard to get separate jobs for each computer to do where they wouldn't interfere with each other. With mergesort, you can break up the lists and have multiple processors working until the very end.

3.1 Designing a Heap

A heap is a data structure that follows the following rules:

1. A Binary Tree
2. Every node is greater than or equal to all its children (in particular, the largest element must be the root!)
3. The tree is complete That is, every level of the tree is completely filled, except possibly the last row which is filled from left to right.

In the session we discussed how to `removeMax` from a heap (that is to remove the maximum-valued node and then restore the data structure to being a valid heap) and how to `insert` into a heap. The basic ideas are:

`removeMax`

1. Remove the root.
2. Take the farthest right node from the bottom row, and put it in the root's location.
3. Percolate the new root down where it belongs, by repeated swapping the node with the larger of its children until the node is larger than both of its children.

`insert`

1. Create a new node with the new datapoint at the bottom-right of the heap.
2. Percolate the new node up to where it belongs, by repeated swapping the node with its parent until the node is smaller than its parent.

See the slides from the session for some examples.

3.2 Analyzing the heap

Big idea: How many data points fit in a heap?

How long does it take to `insert` and `removeMax`? Well, the slowest they could be would be to swap from the top-to-the-bottom (or bottom-to-top). So the number of operations to do is at most O(height of the heap) [9]

But we don't want to leave our running time in terms of the height of the heap. Imagine if another programmer, someone who wants to determine the number of content-creators a person could safely subscribe to, comes to you and asks "hey, how many videos can a user safely have in their list before it takes too long to get a video?" Telling them "Oh, it's fine as long as the height of the video heap is at most 15." might be a *true* statement, but it won't be particularly useful statement for the other programmer. They might never have even learned about heaps, let alone know how to translate that into an answer to their question – someone shouldn't need to know details about your implementation to know how to use it! We want our running times in terms of n, the total number of videos in the data structure.

3.2.1 Converting from h to n

So, let's say we have a tree with h levels in it. How many videos will fit in those h levels?
The first level (the root) has room for 1 node (the root!)
The second level (children of the root) has room for 2 nodes (the two children the root is allowed)
The third level (grandchildren of the root) has room for 4 nodes (two children each for the two nodes at the second level)
The fourth level (great-grandchildren) has room for 8 nodes
Continuing the pattern, at level h there will be 2^{h-1} nodes (if it's totally full).

So the most nodes we can fit is:

$$\sum_{i=1}^{h} 2^{i-1} = 2^h - 1$$

What does that mean for us? Well, we can set that expression equal to n for the most nodes we could fit.[10], so we'll have $h \approx \log_2(n)$. This might make intuitive sense if you think about what logs do – the number of nodes we can fit is about doubling at every level, so asking "what power do I have to raise 2 to get the number of nodes? it should be about the height, if each new level has about as many nodes as all the levels before it combined."

With this calculation, we can say that the worst-case running times for `insert` and `removeMax`

[9] If you haven't seen big-O notation before, the $O()$ means "don't worry about constant factors or lower order terms" so $2\log(n)$, $3\log(n)+5$, and $\log(n-1)$ would all be "$O(\log n)$", because that's the "leading term." A lot of implementation details are being swept under the rug for this document. If you want to know enough to write this code yourself, you should learn some common implementation tricks, especially how to store the heap in an array instead of using pointers.

[10] If you're worried about the least nodes that could be there, remember that we have to completely fill a level before going to the next one, so we have at least 2^{h-1} nodes

for a heap with n elements are both $O(\log n)$.

Takeaways: Practice with finding closed forms of summations and logarithms; also potential motivation for these topics when first introduced in algebra courses.

3.3 Compare and Contrast

A heap is a great data structure for these two operations, but it's not the only way of organizing this data! There are a few other very reasonable choices, each with their own pros and cons. In a data structures class, we spend a lot of time discussing benefits and drawbacks and practicing how to frame discussions and rather little time giving easy answers.

As an additional activity, have students analyze how they would perform the same tasks (`removeMax` and `insert`) if they were using an array that they kept sorted, and an array that they just left unsorted.

Here is a table comparing three possible implemtnations, and their worst-case running times with n videos.[11]

Data Structure	`removeMax`	`insert`
Heap	$O(\log(n))$	$O(\log(n))$
Sorted Array	$O(1)$	$O(n)$
Unsorted Array	$O(n)$	$O(1)$

So which should you use? It depends! The heap offers the best balance of running times, but it isn't strictly better than either of the other two options. It's very easy to maintain an unsorted array because it has minimal structure (insertion is very fast, because there aren't strict sorting rules to follow. Just put it anywhere, it's fine). But without much structure, it's very slow to find the next video to show. Conversely, a sorted array has insert being quite slow because it has too much structure – you can find where to put it with binary search, but you might have to shift a lot of elements to make room. The "partially sorted" nature of a heap balances the amount of structure so you can remove the max and insert significantly faster than $O(n)$, but neither is $O(1)$. If for some reason you absolutely need constant time on one of those operations, you should use the other implementations!

[11] Students might get slightly different times for slightly different implementations (for example, the running time of `insert` in an unsorted array depends on where the student chooses to insert the new element), but I don't know a way to do better than these times.

3.4 Extensions

There are lots of other operations we might want our data structure to perform; here are some common ones, along with comparisons to other options.

`DecreaseKey` given an element, change its priority to a smaller number.
Motivation: we dont want to show videos once they stopped being viral. Over time, we should decrease the "priority" of the videos.

`Build` given an unordered list of videos, turn them into the data structure.
Motivation: we have to start somewhere! When someone makes a new account, they'll probably start with a few recommended videos, use those as the starting point (we could `insert` them one at a time, but maybe there's a faster way).

Expanding our table:

Data Structure	`removeMax`	`insert`	`decreaseKey`	`Build`
Heap	$O(\log(n))$	$O(\log(n))$	$O(\log(n))$	$O(n)$
Sorted Array	$O(1)$	$O(n)$	$O(n)$	$O(n \log n)$
Unsorted Array	$O(n)$	$O(1)$	$O(1)$	$O(1)$

The details of these running times depend quite a bit on the exact implementations, and might need some extra data structures.[12]

To `build` on a heap quickly, use Floyd's BuildHeap. Multiple sorting algorithms are available to sort an array efficiently. As before, the less structured a data structure is, the faster you can insert, but the slower it is to use it.

3.5 Some other uses of heaps

If your students don't like TikTok, here are some other scenarios where heaps might be useful:

- Keep track of a list of your homework assignments, and decide which to work on next. The priority can be a combination of the work required and the time until due date.
- Keep track of the features you want to add to your app.
- At Hogwarts, professors can take points away from houses when students misbehave. Some professors (Snape) have been accused of removing more points from rival houses

[12]To `decreaseKey`, for example, you would need to find the node to move it. That may require extra dictionar(ies). And `build` speed might depends on whether you need to copy over all the datapoints first.

than their own. Dumbledore decides to keep track of all the incidents involving the removal of points, ordered by how many were lost, for him to investigate on slower days.

- Instead of answering emails in the order they arrive, you decide to order them by when you need to respond.