1. Start with the binomial theorem, where we substitute \((x+1)^n = \sum_{k=0}^{n} \binom{n}{k} x^k\). Differentiate with respect to \(x\) and get: \(n(x+1)^{n-1} = \sum_{k=0}^{n} k \binom{n}{k} x^{k-1}\). When \(x = 1\) we get \(\sum_{k=0}^{n} k \binom{n}{k} = n 2^{n-1}\)

2. (a) To implement a stack, define a Max-Heap and an index \(i\). To push – insert an element into the heap with a priority \(i\) and increment \(i\) by 1. This way, a later element has a higher priority, so it’s further up the heap. To pop, just call Heap-Extract-Max, to extract the last element pushed. Similarly, you can access the top of the stack by calling Heap-Max.

(b) To implement a queue, define a Min-Heap and an index \(i\). To enqueue – insert an element into the heap with a priority \(i\) and increment \(i\) by 1. This way, an earlier element has a smaller priority, so it’s further up the heap. To dequeue, just call Heap-Extract-Min, to extract the last element pushed. Similarly, you can access the top of the stack by calling Heap-Min.

3. You can maintain a Min-heap \(A\) of size \(k\), that at every given moment keeps the minimum element in each of the \(k\) heaps. Each time we extract an element from the min-Heap we extract the minimum out of all these minima, and add it to the sorted list of merged items. Each time we put the minimum element in a heap we remove it from its list. We then have to trace back the list it came from and insert the next smallest element into the heap \(A\). We perform overall up to \(n\) HeapInsert and \(n\) HeapExtractMin, but the size of the heap is always at most \(k\), so the total runtime is \(O(n \log k)\)

4. Say we want to delete node \(i\). One can, for example, call IncreaseKey\((A, i, \infty)\) followed by ExtractMax\((A)\). The increase of the key to infinity will put it at the top of the heap, and the ExtractMax will take care of removing that key and arranging the heap. Both operations are \(O(\log n)\), so calling them one after the other is still logarithmic.

5. This can probably be shown in more than one way, here is the one I found easiest (there are certainly more elegant ways though): Let’s look at the first row \(a\) after the sorting of rows and then columns. Let us look at any two indices \(i, j\) such that \(i < j\). The element at position \(i\) at row \(a\) after the double sorting was originally the \(i^{th}\) smallest element, following row sorting and before column sorting, of some row \(l\) – let us call this element \(l_i\), and the element at position \(j\) was, after row sorting and before column sorting, the \(j^{th}\) smallest element at some row \(m\), denoted \(m_j\). Since this is the first row, then \(l_i\) is the smallest element in column \(i\), and in particular - \(l_i \leq m_i\) where \(m_i\) was the element that was originally the \(i^{th}\) smallest at row \(m\) after row sorting. Since \(i < j\), then after row sorting \(m_i \leq m_j\) and therefore \(m_j \geq l_i\). This is true for any \(i, j\) and therefore the top row is sorted. We can remove it from consideration and apply the same logic to the second row which is the smallest remaining etc.
6. Ex. 3.1 in Lecture notes 3. Let us prove by induction.

- Base case: When the heap has only one leaf $\text{Heapify}(A,i)$ does nothing, really, because the two children are empty. A single node is trivially a heap so we are ok.
- Let us assume by induction that both $L$ and $R$ are heaps.
- The if statements in the beginning find out the largest of $A[i]$, $A[L]$ and $A[R]$. If $A[i]$ is the largest, then it is already a heap as by inductive hypothesis $L$ and $R$ are heaps, and so the tree rooted at $A[i]$ is also a heap. Otherwise, exchanging $A[i]$ with the largest element ensures that the overall largest element is at the root of the heap (by inductive hypothesis it has to be $A[L]$ or $A[R]$). We then call $\text{Heapify}(A, \text{Largest})$, where we repeat the process until possibly hitting the base case. But since both subtrees are Heaps by inductive hypothesis, then this logic applies to all subsequent calls.

7. Ex. 6.1 in Lecture note 3: Build a Min-heap (which takes $O(n)$). You can then call $\text{Heap-Extract-Min}$ $k$ times, which is $O(k \log n)$. Overall the runtime is then $O(n + k \log n)$

8. Ex. from Lecture note 4: This is sort of like the binomial theorem. Let us look at the full levels:

- At the $k$ level, every stage is the product of $k$ stages, where each stage was either 0.1 or 0.9 of the stage above. So it is a binomial distribution with $p = 0.1$ and $q = 0.9$ and the partition cost which sums up to the overhead on the right hand side is: $\sum_{i=0}^{k} cn\binom{k}{i}(0.9)^i(0.1)^{k-i} = cn$ according to the binomial theorem. Obviously for partially full levels this is an upper bound as some of these terms are missing.

9. The answer to all the questions is "no". The reason is that for every tree with $L$ leaves, the minimum depth of the tree is $\Omega(\log L)$. So, if we only look at the half of the $n!$, their subtree has $\frac{n!}{2}$ leaves, in which case the depth is $\Omega(\log \frac{n!}{2})$ which is still $\Omega(n \log n)$... The same logic applies to $\frac{n!}{n}$ (the depth of this tree is $\Omega(\log n! - \log n) = \Omega(n \log n)$ and $\frac{n!}{n}$, where the depth of the tree is $\Omega(\log n! - n) = \Omega(n \log n)$.

10. Problem 8-5 (a-d only) (page 207).

(a) It means the array is sorted. If we substitute $k = 1$ in the formula above we get $\sum_{j=i}^{i+1} A[j] \leq A[i] \leq A[i+1]$ for each $i$, which is the definition of sorting.

(b) For example, $\{2, 1, 3, 4, 5, 6, 7, 8, 9, 10\}$. For the first pair, $\frac{2}{2} = 1.5$, and $\frac{3}{2} = 2$, so the condition holds for the first three indices. For the other consecutive pairs it’s obviously true because the rest of the array is sorted.

(c) We can show that the definition of $k$-sorting is equivalent to the condition in the question.

By definition, $\frac{i+k-1}{k} A[j] \leq \frac{i+k}{k} A[j]$, which means $\frac{i+k-1}{k} A[j] - \frac{i+k}{k} A[j] \leq 0$ or equivalently, $\sum_{j=i}^{i+k-1} A[j] - \sum_{j=i+1}^{i+k} A[j] \leq 0$ (after multiplying both sides by $k$). These two sums have most terms in common except the first of the first and the last of the last, so when subtracting one from the other most terms cancel out and we get $A[i] - A[i+k] \leq 0$ or $A[i] \leq A[i+k]$.

(d) According to (c) above, an array is $k$-sorted iff $A[i] \leq A[i+k]$ for every $i$. Therefore, to $k$-sort an array we only need to make sure that every element $A[i]$ is only sorted with respect to its $k$ neighbors. Therefore, we divide the array into $\frac{n}{k}$ subsets of $A[1+ik], A[2+ik], \ldots$ and sort them with respect to one another using merge-sort or heap-sort. Each sorting
is $O\left(\frac{n}{k} \log\left(\frac{n}{k}\right)\right)$, and we do $k$ such sorting operations, one per subset. Overall the runtime is $O\left(k \frac{n}{k} \log\left(\frac{n}{k}\right)\right) = O(n \log(\frac{n}{k})).$

11. Assume that $c \geq 0$. Assume you had some kind of super-hardware that, when given two lists of length $n$ that are sorted, merges them into one sorted list, and takes only $n^c$ steps.

(a) The recursive method can be like MergeSort: Split at the midpoint, call recursively on each half and merge the two with the super-hardware procedure. The boundary condition is of course if the lists are empty or have one item.

(b) $T(n) = 2T\left(\frac{n}{2}\right) + 2\left(\frac{n}{2}\right)^c$ – it should be noted that the two merged list are of size $\frac{n}{2}$ each.

(c) According the the master theorem, any value of $c < 1$ would basically fall in case 1 and would make $T(n) = n$. Therefore, we would be able to sort in significantly less than $O(n \log n)$, which is highly unlikely.