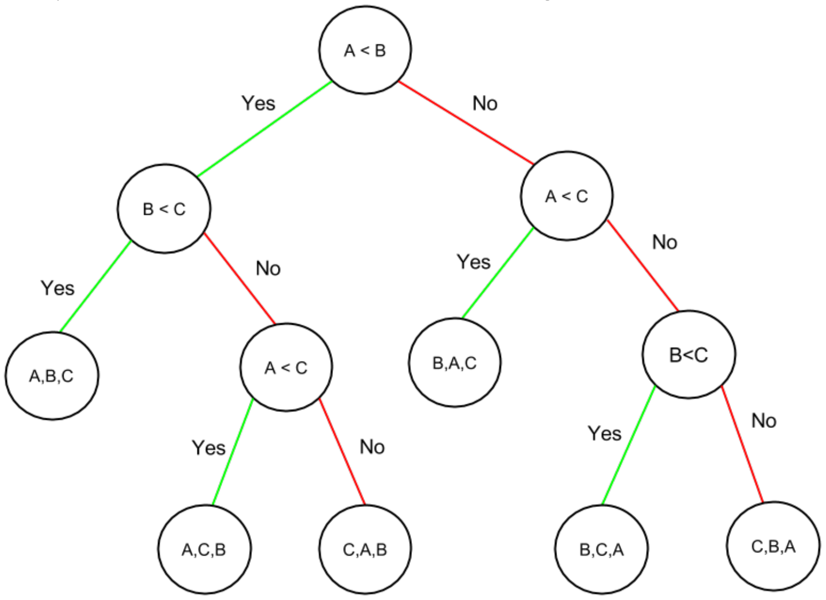
Radix Sort

1 Radix Sort

Before we look at this next sort, let's take a small detour. The prior sorts we went over were comparison based sorts. These sorts have a lower bound of $O(Nlog(N))$. Why? We can visualize a decision tree for this mini proof.

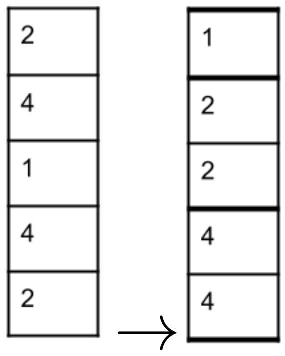
Proof. In the below decision tree, we are seeing how we can order A,B, and C.



The amount of children that each node can have is equal to the amount of decisions we can make at that node. In our normal sorting, we can either say yes or no so we have 2 or 0 children per node. We know that nodes with 0 children are only those that have no more decisions to be made, which means they are fully sorted. The amount of ways we can order the lists is thus equal to the amount of nodes with 0 children, or leaves. In this case, we have 6, or 3!, ways to order our list. Since there are only 2 children per node, we can see that the height of our tree would be dependent on how many leaves we have; this leads us to $lg(6)$ in our case or, rounding up, 3 levels.
We can generalize this for all inputs. The amount of ways we can order any input would be $N!$ and the height of our tree would be upperbounded $lg(N!)$ (we will use $log(N)$ for ease of calculations). We can see that $log(N!) = log(N) + log(N-1) + log(N-2) \leq log(N) + log(N) + log(N) = Nlog(N)$. □

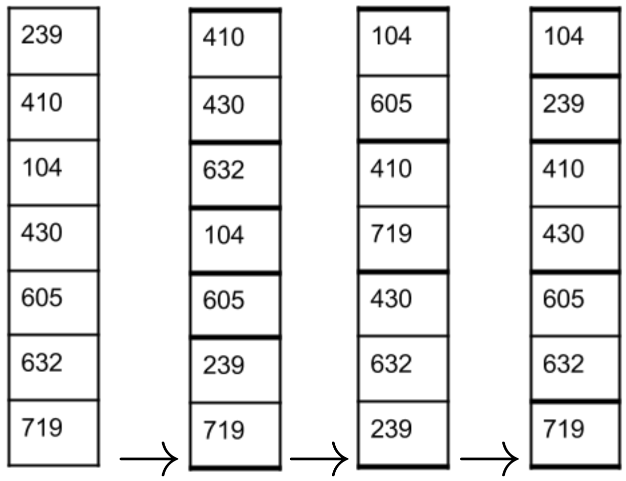
Sometimes, we may want something faster than $Nlog(N)$, but how can we get around this? Is there any way we can avoid "comparing" items? Well if there wasn't there this section of the book wouldn't be written- we can use a category of sort called ***Radix Sort*** to get around this lower bound.
Instead of looking at the full item we are comparing, we look at a certain digit/index of all our elements and

put them into "buckets". We make these buckets by assigning indices based off the amount of elements that are in the bucket. We will briefly go over an example.



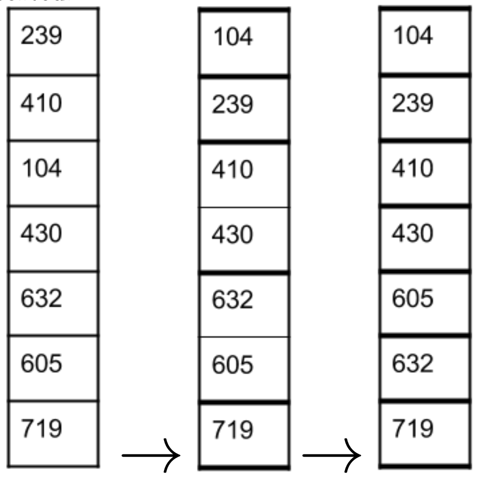
We start off with an unsorted array and then we end up with a sorted array. How is this done? We will start out by finding the amount of times each element appears in our unsorted array. We see that there is 1 1, 2 2's, and 2 4's. Using this information, we will calculate the starting index for each element. We know that 1 is the smallest element, so it will be the first set of buckets, which means that it starts at the 0 index. 2 is the next largest element, and since there is only 1 1, 2 will start at index 1. There are 2 elements in 2, so there will be 2's until index 2- this means that we should start our last bucket 4, at the index after, so 3. The basic idea that we can use for this is that there you keep a starting index for a set of elements that, in our eyes, are equal. We then add our size to the bucket's index to find the end of it. The next bucket will be found by adding 1 to the end of the previous bucket.

There are 2 specific radix sorts that we will go over, ***LSD*** (Least Significant Digit) and ***MSD*** (Most Significant Digit). Least Significant Digit works as follows: we sort from right to left, at each step placing items in buckets that are ordered from least to greatest. We will go over an example:



The bolded lines refer to the "boundaries" of each digit in the process. The first list is unsorted, the second list is sorted by first digit {0, 2, 4, 5, 9 }, the third list is sorted by second digit {0,1,3}, the last list is sorted by final digit {1,2,4,6,7}. How do the items stay in order? After all, sorting with just 1 digit in mind does not make sense. To keep things as sorted as possible we go from top to bottom. For example, say we were on the second digit and we had 2 numbers, 239 and 231. Using the previous step, we would place 231 earlier in the list than 239 because 1 comes before 9. The general rule is that, within the same bucket, items that appeared earlier in a previous iteration will keep their same relative position.

MSD can be thought of as the reverse process as LSD sort. We start from the leftmost digit then we move to the right, making a new subbucket each time. Once every element is in its own element, the entire list is sorted.



We start off with an unsorted list then we move every item into buckets based off their highest digit. The way we order within buckets is based off relative position in the list from the previous iteration. After 2 iterations, every item is in its own bucket so the algorithm terminates.

Now why would we use LSD over MSD and vice versa? MSD tends to be more useful because LSD must use every single digit. MSD on the other hand has the chance of terminating earlier.
In general, the worst case runtime for both LSD and MSD is $\Theta(WN + WR)$ where W is the size of the longest string, R is the size of the alphabet (for the creation of buckets) and N is the amount of strings. As we said earlier, MSD can also have terminate early, this means that it has a best case performance- this is $\Theta(N + R)$ in the case everything gets done in 1 iteration so the width of the key doesn't matter. The runtime of radix sort does not come without cost. There is a much higher memory cost for radix sort- LSD uses $\Theta(N + R)$ space and MSD uses $\Theta(N + WR)$ the extra cost for MSD comes from the usage of collecting the buckets in the recursion of the algorithm.
(Radix sort is a stable sort, in fact that is the whole base of how radix sorts (from top to bottom). If this property did not exist radix sort would return the wrong result.