

A lower time bound for comparison based sorting

Given an input of n numbers to sort, they could be arranged in $n!$ different orders. Clearly, for each of those $n!$ cases, a sorting algorithm **MUST** "act" differently. In particular, if a sorting algorithm makes a certain number of comparisons, when running to each of these $n!$ inputs, no two will give the same exact results for every comparison. (If they did, could you sort those two different data sets differently?)

Thus, we must make enough comparisons, in any comparison based sorting algorithm to distinguish between all $n!$ inputs. A single comparison can distinguish between 2 separate inputs. (Because either an element is greater than the other, or less than the other. For the moment we are assuming distinct elements.) In general, k comparisons can distinguish between at most 2^k inputs. To see this, imagine making a chart like so:

Input	$a[1]>a[2]$	$a[1]>a[3]$	$a[2]>a[3], \dots$	$a[n-2]>a[n-1]$
2,1,4,3	T	F	F	T
2,3,1,4	F	T	T	F
4,3,2,1	T	T	T	T

As you can see, if there are k columns in the chart, each with 2 possible answers, there are a total of 2^k possible distinct rows that could be on the chart. If two distinct inputs gave rise to the same **EXACT** row of answers, it would be impossible for our sorting algorithm to distinguish between the two inputs. Thus, if a sorting algorithm is to work, each input **NEEDS** to lead to a distinct row of answers.

This leads us to the equation:

$$2^k > n!$$

We need to find the smallest value of k which satisfies this equation above. Using logarithms we find this value to be $\log_2 n!$

Now, what is this value equal to approximately? It turns out that $n! > (n/e)^n$. So we find that

$$2^k > n! > (n/e)^n$$

$$\log_2(2^k) > \log_2(n/e)^n$$

$$k > n(\log_2 n - \log_2 e)$$

$$k > n \log_2 n - n \log_2 e > n \log_2 n - 2n = \Omega(n \log_2 n).$$

Thus, we have shown that it is necessary to make at least $\Omega(n \log_2 n)$ comparisons to sort n values in a purely comparison based sorting algorithm.

Bucket Sort

Although no comparison sort is faster than $O(n \lg n)$, if we do assume some information about the input that allows us to sort with extra information (not just comparisons), we can indeed improve our sorting time to $O(n)$. (Clearly we can not do any better asymptotically because we must "look" at every number at least once to guarantee that the output list is sorted.)

In Bucket Sort, we will assume that the input values are randomly distributed in a range $[0, N)$. (Our book says the values have to be integers, but the sort as well as the analysis will still work if we only assume that the values being sorted are real.)

Assume that we have n values to sort. We will create n different buckets to hold all the values. We can implement each bucket with a linked list. Each bucket will store values within a range of N/n .

Perhaps the easiest way to get a grasp of the algorithm is to go through an example:

Consider sorting a list of 10 numbers known to be in between 0 and 2, not including 2 itself. Thus, each bucket will store values in a range of $2/10 = .2$ In particular, we have the following list:

Bucket	Range of Values	Bucket	Range of Values
0	[0, .2)	5	[1, 1.2)
1	[.2, .4)	6	[1.2, 1.4)
2	[.4, .6)	7	[1.4, 1.6)
3	[.6, .8)	8	[1.6, 1.8)
4	[.8, 1)	9	[1.8, 2)

Consider sorting the following list: 1.3, 0.7, 0.1, 1.8, 1.1, 1.0, 0.5, 1.7, 0.3, and 1.5. Here is a picture of what happens during the sort. Based upon this, imagine how one would write code to implement this sort. (The data structure to use would be an array of linked lists.)

Also, consider that it is not necessary for the input range to start at 0. As long as you have both a lower and upper bound for input, the bucket sort algorithm can be adapted.

Given that the range of inputs is $[L, L+N)$, (where L stands for lowest possible value, and N stands for the range), and there are n values to be sorted, if we are given the value v , our first goal is to determine WHICH linked list to insert the value into. Once we do that, then we can simply call our linked list insert method to insert a value into a linked list in sorted order.

The correct array index is $\text{floor}(n(v - L)/N)$, using integer division. To see this, consider that the value $v-L$ is from the range $[0, N)$. Now, we must map this to the correct array index from 0 to $n-1$. Each array index has a range of N/n values. Thus, what we want to do is divide our value $v-L$ by the range of each bucket to give us the array index of the appropriate bucket. For example, using our previous list, if we were to sort 1.3, it would go to the bucket $\text{floor}(10(1.3 - 0)/2) = 6$ as desired.

Although I will skip the mathematics as to why this sort is $O(n)$, I will give you the intuition as to why that is the case. Since we have as many buckets and values to sort, IF the values to sort are randomly distributed, then the length of each linked list will be constant with very high probability. Each insert in a constant sized linked list will take constant time, just the total amount of time for the sort will be linear in the number of items to sort.

Radix Sort

The input to this sort must be non-negative integers all of a fixed length of digits. Let each number be k digits long. The sort works as follows:

- 1) Sort the values using a $O(n)$ stable sort on the k th most significant digit.
- 2) Decrement k by 1
- 3) Repeat step 1. (Unless $k=0$, then you're done.)

Once again, this sort is much more easily understood with a demonstration. The running time of this sort should be $O(nk)$, since we do k stable sorts that each run in $O(n)$ time.

Depending on how many digits the numbers are, this sort can be more efficient than any $O(n \lg n)$ sort, depending on the range of values.

A stable sort is one where if two values being sorted, say v_i and v_j are equal, and v_i comes before v_j in the sorted list, then v_i will STILL come before v_j in the sorted list.

This sort does seem quite counter intuitive. One question to ask: would it work the other way around (namely from most significant to least significant digit)? If not, could we adapt the sort to work the other way around? Why does this always work? Why does the intermediate sort need to be stable?

The key to it is as follows: After the m th iteration, the values are sorted in order with respect to the m least significant digits. Clearly, when $m=k$, this means the list is just plain sorted!

Here is an illustration of Radix sort:

unsorted

-----	v	v	v
235	162	628	162
162	734	734	175
734	674	235	235
175	235	237	237
237	175	162	628
674	237	674	674
628	628	175	734

The second column is sorted by the units digit, the third by the tens digit and the last column by the hundreds digit. Notice how each sort is stable, (when there are ties, the order of the elements is NOT switched. Only in this case is correctness guaranteed.)