Learning Goals

During this lab, you will:

- Review the ideas of sorting and the run time for comparison based sorting
- Introduce the idea of linear-time sorting algorithms
- Discuss different types of Radix sort

Apply to be a Intro CIS TA!

If you are interested in TAing an intro CIS course (CIS 110, CIS 120, CIS 160, or CIS 121), please submit your application at [bit.ly/18sp_intro_app](http://bit.ly/18sp_intro_app). Applications are due on November 29th, at 11:59pm.

Background on Sorting

Limitations of comparison-based sorting

So far, we've focused on sorting elements with algorithms like Mergesort, QuickSort, and Insertion sort. These are comparison sorts, referring to the property that they sort by comparing elements to each other. With a little bit of handwaving, we argued that all comparison sorts take $\Omega(n \lg n)$ time in the worst case. There's a cool proof for this, but the 10-second version is that there are $n!$ possible permutations of your input, only one of which is the sorted output. In the worst case, you can't cut out more than half of the possible permutations with each comparison, so you'll have to compute $O(\lg(n!)) \in O(n \lg n)$ comparisons to pare down your search space to the 1 correct permutation. If you are interested in the full proof of this lower bound, you will learn it in CIS 320.

Linear-time sorting

Comparison sorting allows us to sort any elements on which we can impose a total order. However, sometimes we know some key information about the elements we're sorting, and we can use that to sort in linear time. Consider, for motivation, the following simple function. It takes a stream that will provide 26 key-value pairs. The keys are distinct lowercase letters, and the values are Objects. Its output is the 26 key-value pairs. The keys are distinct lowercase letters, and the values are Objects. Its output is the 26 key-value pairs in alphabetical order by key. Does this algorithm need to do any comparisons? Clearly not; if it sees $e$ first, it can simply put the key-value pair in the 5th slot of its output array, as $e$ is the 5th letter in the English alphabet. It is easy to see that this sort is done in linear time. Since we know exactly where each character must go in our output array, we can place it immediately into the correct location without doing any fancy comparison/sorting techniques. Generalizing this, we can accomplish linear-time sorting for any set of fixed-length strings.
**Key-indexed counting and radix sort**

Just as decimal notation uses a radix of 10, and binary a radix of 2, the radix in radix sort refers to the size of the alphabet used to express the strings we’ll be sorting.

Key-Indexed counting is a subprocess that we’ll use in radix sort. It sorts a list of single-character strings in linear time using two auxiliary arrays. It does so in three stages: First, it calculates the number of times each key shows up in the input. (Note that this is useful because there is an a-priori bounded number—the radix—of possible keys!) It then uses these counts to determine the first place in the output a given key will appear. It then walks through the input array, copying keys to an auxiliary array at the index determined in step 2, and incrementing the index at which the same key should be placed. The pseudocode is below.

```plaintext
function key_indexed_count(Array A, int radix):
    ⊲ Sorts characters alphabetically
    counts[] ← [radix + 1]
    aux[] ← [A.length]
    for i ← 0 to A.length do
    end for
    ⊳ Count[i + 1] now the number of times key i has been seen.
    for i ← 0 to count.length − 2 do
        counts[i + 1] ← counts[i] + counts[i + 1]
    end for
    ⊳ count[i] now the index to put the first key of val i
    for i ← 0 to A.length do
        aux[count[A[i]]] = A[i]
        count[A[i]] = count[A[i]] + 1
    end for
    ⊳ keys i are copied to aux; next position for key i incremented
    return aux
end function
```

**LSD radix sort**

The idea of least-significant digit radix sort is to first order your strings alphabetically by the far-right (least significant) letter. Then, move iteratively left, re-ordering your strings alphabetically by more significant bits. Because each more-significant sort is stable, no ordering ‘undoes’ the relative less-significant orderings before it. (i.e. if a sort sees two e characters, then if the less-significant letter before the e is different, the sorting of the less-significant letter will be preserved.)

**MSD Radix Sort**

MSD isn’t just the reverse of LSD, starting from the left and moving right. It shouldn’t take too much thought to convince yourself why this won’t work. Instead, MSD is a recursive algorithm that uses key-indexed counting to sort increasingly smaller subarrays of the input. For example, once the keys have been sorted by most significant digit, each set of strings with the same MSD are sorted on their second-most sig digit. The base case is hit at arrays of size 1.

**3-way string quicksort**

Empirically, quicksort is great. However, as we’ve defined it (as a comparison sort), each comparison compares full strings. Our question is, how can we optimize quicksort to work on strings? Unsurprisingly, we construct a recursive algorithm that, at each depth of the recursion, partitions strings into 3 categories, and recurses into each category.

At each depth $d$ of the recursion, we pick a pivot string, and partition the current set of strings into those with $str[d] < pivot[d]$, (less), $str[d] = pivot[d]$, (equal) and $str[d] > pivot[d]$ (more.) As discussed in the discussion section, we gain a running time benefit by comparing by character instead of by string.
Discussion

MSD v. LSD

Compare MSD and LSD radix sort. When would one be preferred over the other? What are a few reasons that support your intuition?

Solution. Students should discuss the large amount of recursive calls on small array sizes with MSD, as well as possible optimizations discussed in class with regard to a backoff to insertion sort for small array sizes. MSD considers the most significant digit/char first, so it examines just enough characters to sort the keys (not all of them all the time). As seen in the slides, the running time can become sublinear in input size if input does not have a lot of common prefixes. LSD saves more space than MSD and has the same worst case guarantee as MSD.

Quicksort running time

If you run quicksort on $N$ strings of average length $L$, what is the expected running time of quicksort in terms of $N$ and $L$?

Solution. In expectation, quicksort performs $O(N \lg N)$ comparisons (actually around $2N \lg N$). Each quicksort comparison is a string comparison, which does $O(L)$ work to compare $L$-length strings for an expected $O(LN \lg N)$.

3-way string quicksort running time

If you run 3-way string quicksort on $N$ strings of average length $L$, what is the running time in terms of $N$ and $L$?

Solution. In expectation, 3-way string quicksort performs $O(N \lg N)$ comparisons (around $2N \lg N$). Each comparison is a character comparison, so the total running time is expected $O(N \lg N)$.

Testing your Understanding

Problem 1. Which sorting method would you use to suffix sort a very long string, like Moby Dick which has about 1.2 million characters (assuming the longest repeated substring is not too long)?

Solution. Using any standard sorting algorithm, like insertion, merge, or quick sort will take a very long time as mentioned in the discussion. The candidates are now LSD, MSD and 3-way radix quicksort. Usually 3-way radix quicksort is the fastest in practice even with the cutoff optimization for MSD radix sort because no algorithm can examine fewer bits. If the longest repeated suffix is long, radix sorts take time quadratic in length of longest match. In this case, there are optimizations like Manber’s MSD, but these are not covered in this class.

Problem 2. Given an unsorted array, find the first missing non-negative number. For example, if the input is $[2, -1, 1, 0, 4]$ you should return 3.

Solution. Note that we can take the array size $len$ and make a new array (call it $arr$) of that size. For any index from 0 to $len$ we put the index’s number in that index, i.e. $i == arr[i]$. For example, any 0 seen should be moved to $arr[0]$, 1 to $arr[1]$ and so on. Note that any negative numbers and numbers greater than $len$ are simply ignored. Once we construct $arr$, then we can do a linear scan and find the first index such that $i \neq arr[i]$, and return this number. The running time is $O(N)$ and space complexity is also $O(N)$.

1See here: https://www.cs.princeton.edu/~rs/AlgsDS07/18RadixSort.pdf

3