Background on Sorting

Comparison Sort Limitations

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 $O(n \log n)$ 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 $O(\log(n!)) \in O(n \log n)$ comparisons to pare down your search space to the 1 correct permutation.

Linear-Time Sorting

Comparison sorting allows us to sort any elements on which we can impose a total order. However, sometimes we know more 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 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. It is easy to see that this sort is done in linear time. Generalizing this, we can accomplish linear-time sorting for any set of fixed-length strings.

Key-Indexed counting, 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.

LSD Radix Sort

The idea of LSD 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. 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.
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
  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

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 $str[d] < \text{pivot}[d]$, (less), $str[d] = \text{pivot}[d]$, (equal) and $str[d] > \text{pivot}[d]$ (more.) As discussed in the discussion section, we gain a runtime benefit by comparing by character instead of by string.

Tries (A Quick Summary)

1. A trie is a special kind of tree that is used to store strings. With an alphabet size $R$, each node in a trie has $R$ children. A node $v$’s descendants are the set of all strings input into the trie that have the prefix corresponding to the path to $v$ from the root of the trie. (If root $\rightarrow_p v$, $v$’s descendants could include pizza and piazza. )

2. When implementing a trie, we want basic functionality, such as: inserting a word, looking up a word, and deleting a word. Typically, we can use an array of size $R$ to keep track of children. Any node that is the end of a string can be marked with some count variable, a null, or some other special character. Search can simply just follow up the characters in the query through the trie and see if a terminal node is reached. Insertion and deletion are fairly straightforward; additionally, they should handle the case where a substring of the word is already present or needs to stay present in the trie (for example, say the trie contained the word ”baggage” and ”bag” and you wanted to delete ”baggage”).

3. Check out the lecture notes and slides for visualizations of what a trie would look like. Tries are used for a variety of cases, particularly when we are interested in properties (occurrences, lengths, etc) of substrings of given words.

Discussion

MSD v LSD
Compare MSD and LSD radix sort. Intuitively, which one seems like it should be faster? What are a few reasons that support your intuition?
QuickSort runtime

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

3-way String Quicksort runtime

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

Testing your Understanding

**Problem 1.** Given a set of $N$ strings, how can we find the longest common prefix between any two strings? Analyze the runtime of your algorithm and give a short proof of correctness.

**Problem 2.** As discussed, the space usage of a trie depends on the prevalence of common prefixes between input strings. For this problem, we define *average space usage* as the total space usage of a trie divided by the number of strings in the trie. Given alphabet of size $R$ and strings of length $L$, determine:

1. The number of strings $N$ (in terms of $R$ and/or $L$), and characteristics about each string $n_i$ such that the average space usage of the trie is maximum.

2. The number of strings $N$ (in terms of $R$ and $L$) such that the average space usage of the trie is minimum.

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

**Problem 4.** Given some arbitrarily long string, how can we find the longest repeated substring? What about if we want the longest that is repeated $k$ times?

**Problem 5.** What is the *average space usage* of a complete trie? That is, a trie with as many strings as it can hold for its values of $R$ and $L$?