Lecture 7: Linear-Time Sorting

Lecture Overview

- Comparison model
- Lower bounds
  - searching: $\Omega(\lg n)$
  - sorting: $\Omega(n \lg n)$
- $O(n)$ sorting algorithms for small integers
  - counting sort
  - radix sort

Lower Bounds

theorem
proof
counterexample

Claim

- searching among n preprocessed items requires $\Omega(\lg n)$ time
  $\Longrightarrow$ binary search, AVL tree search optimal
- sorting n items requires $\Omega(n \lg n)$
  $\Longrightarrow$ mergesort, heap sort, AVL sort optimal

. . . in the comparison model

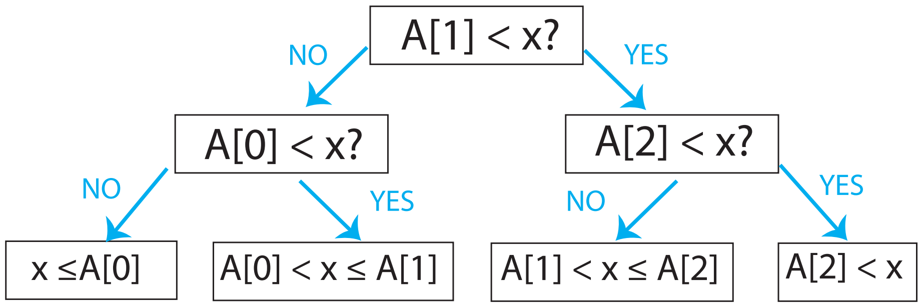
Comparison Model of Computation

- input items are black boxes (ADTs)
- only support comparisons ($<, >, \leq$, etc.)
- time cost = # comparisons

Decision Tree

Any comparison algorithm can be viewed/specified as a tree of all possible comparison outcomes & resulting output, for a particular n:

- example, binary search for $n = 3$:



- internal node = binary decision
- leaf = output (algorithm is done)
- root-to-leaf path = algorithm execution
- path length (depth) = running time
- height of tree = worst-case running time

In fact, binary decision tree model is more powerful than comparison model, and lower bounds extend to it

Search Lower Bound

- # leaves $\geq$ # possible answers $\geq n$ (at least 1 per $A[i]$)
- decision tree is binary
- $\Longrightarrow$ height $\geq \lg \Theta(n) = \lg n \underbrace{\pm \Theta(1)}_{\lg \Theta(1)}$

Sorting Lower Bound

- leaf specifies answer as permutation: $A[3] \leq A[1] \leq A[9] \leq \ldots$
- all $n!$ are possible answers

- # leaves $\geq n!$

$$\begin{aligned}
\implies \text{height} &\geq \lg n! \\
&= \lg(1 \cdot 2 \cdots (n-1) \cdot n) \\
&= \lg 1 + \lg 2 + \cdots + \lg(n-1) + \lg n \\
&= \sum_{i=1}^{n} \lg i \\
&\geq \sum_{i=n/2}^{n} \lg i \\
&\geq \sum_{i=n/2}^{n} \underbrace{\lg \frac{n}{2}}_{= \lg n - 1} \\
&= \frac{n}{2} \lg n - \frac{n}{2} = \Omega(n \lg n)
\end{aligned}$$

- in fact $\lg n! = n \lg n - O(n)$ via Stirling's Formula:

$$n! \sim \sqrt{2\pi n}\left(\frac{n}{e}\right)^n \implies \lg n! \sim n \lg n - \underbrace{(\lg e)n + \tfrac{1}{2}\lg n + \tfrac{1}{2}\lg(2\pi)}_{O(n)}$$

Linear-time Sorting

If n keys are integers (fitting in a word) $\in 0, 1, \ldots, k-1$, can do more than compare them

- $\implies$ lower bounds don't apply
- if $k = n^{O(1)}$, can sort in $O(n)$ time
  OPEN: $O(n)$ time possible for all k?

Counting Sort

```
L = array of $k$ empty lists                                   } $O(k)$
    — linked or Python lists
for $j$ in range $n$:                                          } $O(n)$
    $L[\underbrace{\text{key(A[j])}}_{\text{random access using integer key}}]$.append(A[j])   $\rightarrow O(1)$
output = [ ]                                                   } $O(\sum_i(1+|L[i]|)) = O(k+n)$
for $i$ in range $k$:
    output.extend($L[i]$)
```

Assume $key(A[i]) \neq key(A[j])$ iff $A[i] \neq A[j]$. k can be large (e.g., 2^{32}) and counting sort will require a lot of storage that is sparsely used.
Time: $\Theta(n+k)$ — also $\Theta(n+k)$ space

Intuition: Count key occurrences using RAM
output <count> copies of each key in order . . . but item is more than just a key

CLRS has cooler implementation of counting sort with counters, no lists — but time bound is the same

Anagram Puzzle

You are given a long list of English words in alphabetical or random order. Your task is to sort the words such that sets of anagrams are next to each other in the sorted list minimizing worst-case complexity.[1]

For example, given `ate`, `dog`, `eat`, `god`, `tea`, we want `ate`, `eat`, `tea`, `dog`, `god`.

We create a histogram for letters in a word. For each word, we populate an array of size σ, where σ is the size of the alphabet. For English, $\sigma = 26$. For our list we get:

`ate`	`10001000000000000001000000`
`dog`	`00010010000000100000000000`
`eat`	`10001000000000000001000000`
`god`	`00010010000000100000000000`
`tea`	`10001000000000000001000000`

Note that the count for every letter is bounded by l which is less than the length of the longest word, but in reality will be much smaller, since it is bounded by

[1]One can use hashing to solve this problem, but we are focused on a different solution that does not require hashing.

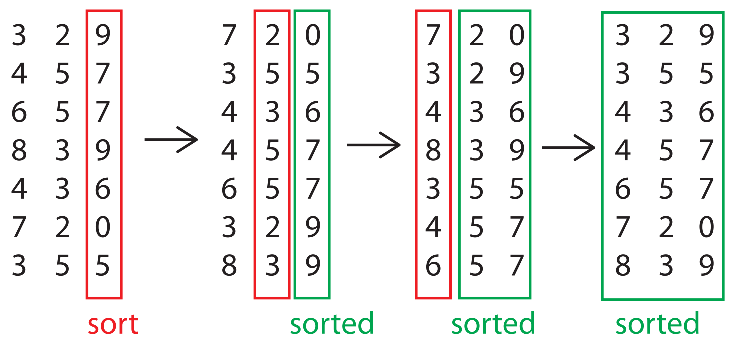
the maximum number of occurrences of the same letter in any word. For example, `possesses` contains 5 `s`'s.

It is pretty clear that sorting the above list will give us what we want but what sorting algorithm should we use? These are large numbers and counting sort will require a lot of space. We want a complexity of $O((n+l)\cdot\sigma)$, where n is the number of words.

Radix Sort

Radix sort is a digit-by-digit sorting algorithm that works even for large integers like those in our anagram example above. Radix sort is built on top of counting sort.

- imagine each integer in base b
  $\implies$ $d = \log_b k$ digits $\in \{0, 1, \dots, b-1\}$. (In our anagram puzzle, we have $d = \sigma$, and the base $b = l + 1$.)
- sort (all n items) by least significant digit $\to$ can extract in $O(1)$ time
- $\cdots$
- sort by most significant digit $\to$ can extract in $O(1)$ time
  sort must be stable: preserve relative order of items with the same key
  $\implies$ don't mess up previous sorting
  For example:



- use counting sort for digit sort
  - $\implies$ $\Theta(n+b)$ per digit
  - ($\implies$ $\Theta((n+l)\sigma)$ total time for the anagram puzzle)

- $\implies \Theta((n+b)d) = \Theta((n+b)\log_b k)$ total time
- minimized when $b = n$
- $\implies \Theta(n \log_n k)$
- $= O(nc)$ if $k \leq n^c$