Lecture 7: Linear-Time Sorting

Linear time sorting is possible sometimes. This lecture shows when

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

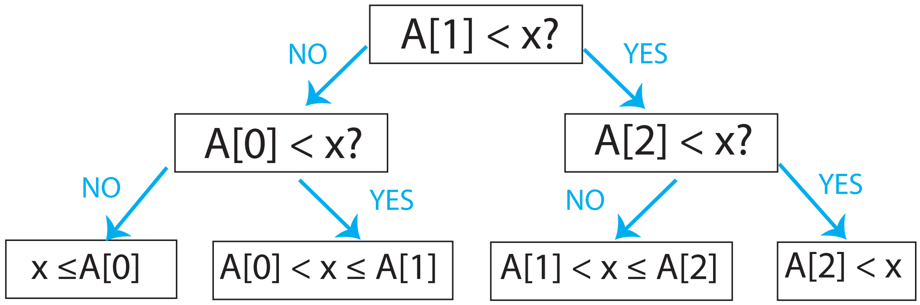
Comparison Model of Computation

- 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

You don’t normally write a binary search as tree
But its useful to show via tree
“this is all this possibilities this algorithm can do”

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

Search Lower Bound

Assumes you have n preprocessed items initially. Do anything with them at t=0
No matter what you do, to find a new item, it will take log n time

- $\#$ leaves $\geq$ $\#$ possible answers $\geq n$ (at least 1 per $A[i]$)
- decision tree is binary And it must have n leaves, 1 for each answer. Maybe more than 1 leaf correspond to 1 answer. (multiple paths to same answer). But leaves have to be atleast n
- $\implies$ height $\geq \lg \Theta(n) = \lg n \underbrace{\pm\Theta(1)}_{\lg\Theta(1)}$

And height is the worst case running time

This extends to finding successor, predecessor etc. All of them must take logn time. Searching takes logn time. Hence BST are very efficient
(Yes you can find pred/succ in O(1) time in sorted arrays, but search is still O(logn) sorted array insertion is O(n) time)

logn is the best you can do
in comparison model, and BST archives that

Sorting Lower Bound

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

Just like case of search;
decision tree is binary; #leaves>=#answers
#leaves>=n!

- $\#$ leaves $\geq n!$ height is log(leaves) in binary tree

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

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

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

Further algos won't be based on comparison model

Linear-time Sorting (integer 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 O(k) always. Also O(n) if k=n^O(1)
  OPEN: $O(n)$ time possible for all k?

Counting Sort Makes no comparisons

L = array of k empty lists
  — linked or Python lists
} $O(k)$

for j in range n:
  $L[\underbrace{\text{key}(A[j])}_{\text{random access using integer key}}].\text{append}(A[j])$ $\rightarrow O(1)$
} $O(n)$

output = [ ]
for i in range k:
  output.extend($L[i]$)
} $O(\sum_i (1 + |L[i]|)) = O(k + n)$

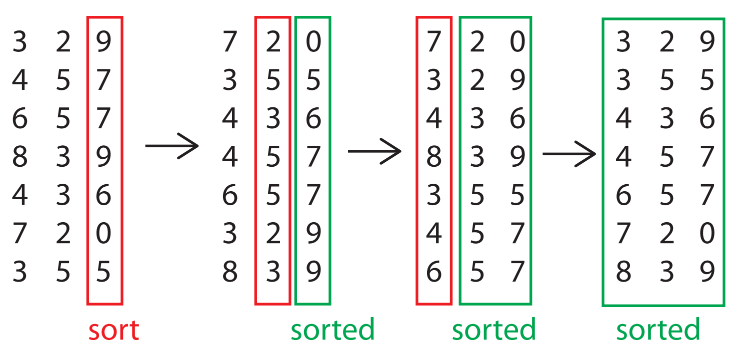
Time: $\Theta(n+k)$ — also $\Theta(n+k)$ space

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

Radix Sort

- imagine each integer in base b
  $\implies d = \log_b k$ digits $\in \{0, 1, \ldots, b-1\}$ d represents the number of digits
- sort (all n items) by least significant digit $\rightarrow$ can extract in $O(1)$ time
- $\cdots$
- sort by most significant digit $\rightarrow$ 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
  - $\implies \Theta((n+b)d) = \Theta((n+b)\log_b k)$ total time
  - minimized when $b = n$
  - $\implies \Theta(n \log_n k)$
  - $= O(nc)$ if $k \leq n^c$