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

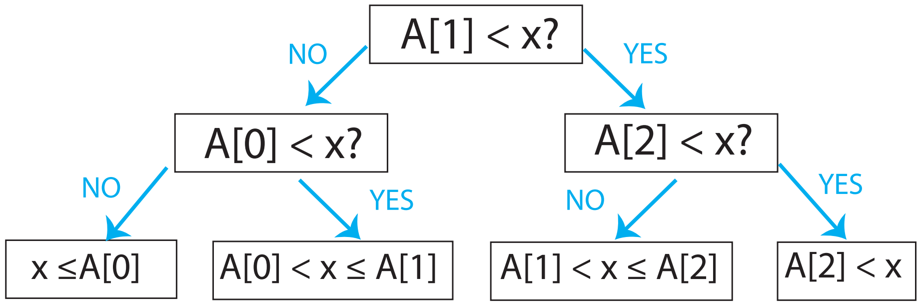
Comparison Model of Computation

- 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
- $\implies$ 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 Sterling's Formula:

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

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
— linked or Python lists
$\Big\}$ $O(k)$

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

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

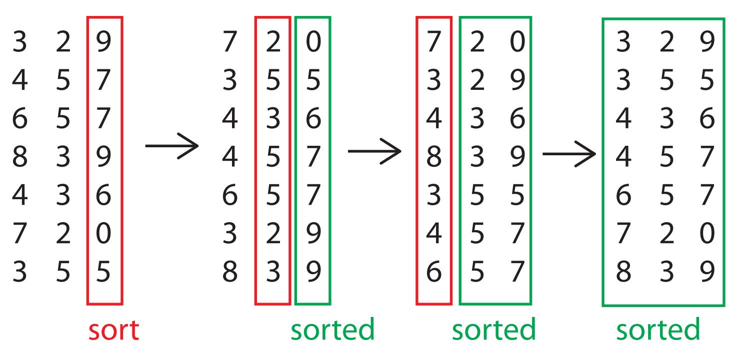
<u>Time:</u> $\Theta(n+k)$ — also $\Theta(n+k)$ space

<u>Intuition:</u> 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
  $\Longrightarrow d = \log_b k$ digits $\in \{0, 1, \ldots, b-1\}$
- <u>sort</u> (all n items) by least significant digit $\rightarrow$ can extract in $O(1)$ time
- $\cdots$
- <u>sort</u> by most significant digit $\rightarrow$ can extract in $O(1)$ time
  sort must be <u>stable</u>: preserve relative order of items with the same key
  $\Longrightarrow$ don't mess up previous sorting
  For example:



- use counting sort for digit sort
  - $\Longrightarrow \Theta(n+b)$ per digit
  - $\Longrightarrow \Theta((n+b)d) = \Theta((n+b)\log_b k)$ total time
  - minimized when $b = n$
  - $\Longrightarrow \Theta(n \log_n k)$
  - $= O(nc)$ if $k \leq n^c$