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

Claim

- searching among $n$ preprocessed items requires $\Omega(lg\ n)$ time
  $\implies$ binary search, AVL tree search optimal
- sorting $n$ items requires $\Omega(n\ lg\ n)$
  $\implies$ 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 ≥ # possible answers ≥ n (at least 1 per $A[i]$)
- decision tree is binary
- $\implies$ height ≥ $\lg \Theta(n) = \lg n \pm \Theta(1)$

Sorting Lower Bound

- all $n!$ are possible answers
• # leaves ≥ n!

⇒ height ≥ lg n!
= lg(1 · 2 · · · (n − 1) · n)
= lg 1 + lg 2 + · · · + lg(n − 1) + lg n
= \sum_{i=1}^{n} lg i
≥ \sum_{i=n/2}^{n} lg i
≥ \sum_{i=n/2}^{n} \frac{lg n}{2}
= \frac{n}{2} lg n - \frac{n}{2} = \Omega(n lg n)

• in fact lg n! = n lg n − O(n) via Sterling’s Formula:

\[ n! \sim \sqrt{2\pi n} \left(\frac{n}{e}\right)^n \Rightarrow lg n! \sim n lg n - \left(lg e\right)n + \frac{1}{2} lg n + \frac{1}{2} lg(2\pi) \]

Linear-time Sorting

If n keys are integers (fitting in a word) ∈ 0, 1, . . . , k − 1, can do more than compare them

• ⇒ 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

\[ L = \text{array of } k \text{ empty lists} \]
\[ \text{— linked or Python lists} \]
\[ \text{for } j \text{ in range } n: \]
\[ L[\text{key}(A[j])].append(A[j]) \rightarrow O(1) \]
\[ \text{random access using integer key} \]
\[ \text{output} = [ ] \]
\[ \text{for } i \text{ in range } k: \]
\[ \text{output.extend}(L[i]) \]
\[ O(\sum_i (1 + |L[i]|)) = O(k + n) \]
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 $\in \{0, 1, \ldots, b - 1\}$

- sort (all $n$ items) by least significant digit $\implies$ can extract in $O(1)$ time

- . . .

- sort by most significant digit $\implies$ 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:

$$
\begin{array}{cccccc}
3 & 2 & 9 & 7 & 2 & 0 \\
4 & 5 & 7 & 3 & 5 & 5 \\
6 & 5 & 7 & 4 & 3 & 6 \\
8 & 3 & 9 & 4 & 5 & 7 \\
4 & 3 & 6 & 6 & 5 & 7 \\
7 & 2 & 0 & 3 & 2 & 9 \\
3 & 5 & 5 & 8 & 3 & 9 \\
\end{array}
\
\begin{array}{cccccc}
\text{sort} & \text{sorted} & \text{sorted} & \text{sorted} \\
7 & 2 & 0 & 3 & 2 & 9 \\
8 & 3 & 9 & 6 & 5 & 7 \\
4 & 5 & 7 & 3 & 5 & 5 \\
6 & 5 & 7 & 4 & 3 & 6 \\
4 & 3 & 6 & 6 & 5 & 7 \\
7 & 2 & 0 & 3 & 2 & 9 \\
3 & 5 & 5 & 8 & 3 & 9 \\
\end{array}
$$

- 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
  $- \text{minimized when } b = n$
  $- \implies \Theta(n \log_n k)$
  $- = O(nc)$ if $k \leq n^c$