Today: Linear-Time Sorting
- comparison model
- lower bounds:
  - searching: $\Omega(lg n)$
  - sorting: $\Omega(n lgn)$
- $O(n)$ sorting algorithms
  - counting sort
  - radix sort

Lower bounds: claim
- searching among n preprocessed items requires $\Omega(lg n)$ time
  $\Rightarrow$ binary search, AVL tree search optimal
- sorting n items requires $\Omega(n lgn)$
  $\Rightarrow$ mergesort, heap sort, AVL sort optimal
... in the comparison model

Comparison model of computation:
- input items are black boxes (ADTs)
- only support comparisons ($<, >, \leq, \text{etc.}$)
- 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 \( n \):

- e.g. binary search for \( n=3 \):

```
A[1] < x?
  NO  A[0] < x?
    NO  x <= A[0]
    YES A[0] < x <= A[1]
```

- 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
  \[ \Rightarrow \text{height} \geq \log \Theta(n) = \log n + O(1) \]

Sorting lower bound:
- leaf specifies answer as permutation:
- all \( n! \) are possible answers
  \[ \Rightarrow \# \text{leaves} \geq n! \]
  \[ \Rightarrow \text{height} \geq \log n! \]
  \[ = \log (1 \cdot 2 \cdot \ldots \cdot (n-1) \cdot n) \]
  \[ = \log 1 + \log 2 + \ldots + \log(n-1) + \log n \]
  \[ = \sum_{i=1}^{n} \log i \]
  \[ \geq \sum_{i=\lfloor \frac{n}{2} \rfloor}^{n} \log i \]
  \[ = \left\lfloor \frac{n}{2} \right\rfloor \log n - \frac{n}{2} = \Omega(n \log n) \]
- in fact \( \log n! = n \log n - O(n) \) via:
  \[ \text{Sterling's formula:} \quad n! \sim \sqrt{2\pi n} \left(\frac{n}{e}\right)^n \]
  \[ \Rightarrow \log n! \sim n \log n - (\log e) n + \frac{1}{2} \log n + \frac{1}{2} \log (2\pi) \]
Linear-time sorting: fitting in a word

if n keys are integers \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 \( \implies O(k) \)

- for \( j \) in range(n):
  \( L[\text{key}(A[j])].append(A[j]) \) \( \implies O(1) \) \( \implies O(n) \)
  \( \uparrow \) random access using integer key

- output = []
- for \( i \) in range(\( k \)):
  \( \text{output}.extend(L[i]) \)

\( \sum_i (1 + |L[i]|) \) = \( O(k + n) \)

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 $b$
  \[ d = \log_b k \text{ digits } \in \{0, 1, \ldots, b-1\} \]
- Sort by least significant digit \[ \Rightarrow \text{all } n \text{ items} \]
- Sort by most significant digit
  - Sort must be stable:
    - Preserve relative order of items with the same key
  \[ \Rightarrow \text{don't mess up previous sorting} \]

\[ \begin{array}{cccc}
3 & 2 & 9 & \text{sort} \\
4 & 5 & 7 & \Rightarrow \\
6 & 5 & 7 & 7 & 2 & 0 & \Rightarrow \\
8 & 3 & 9 & 4 & 3 & 6 & 8 & 3 & 9 & \Rightarrow \\
4 & 3 & 6 & 6 & 5 & 7 & 3 & 5 & 5 & 4 & 5 & 7 & 7 & 2 & 0 & \Rightarrow \\
3 & 5 & 5 & 8 & 3 & 9 & 3 & 2 & 9 & 4 & 3 & 6 & 6 & 5 & 7 & 8 & 3 & 9 & \Rightarrow \\
\end{array} \]

- Use counting sort for digit sort
  \[ \Theta(n+b) \text{ per digit} \]
  \[ \Rightarrow \Theta((n+b)d) = \Theta((n+b) \log_b k) \text{ total time} \]
- Minimized when $b = n$
  \[ \Theta(n \log_n k) \]
  \[ = O(n^c) \text{ if } k \leq n^c \]