Today: → Sorting
  → Insert sort
  → Merge sort (divide & conquer)
  → Solving recurrences (Master theorem)

Sorting: given an array $A[0...n-1]$ of #s,
  find a permutation $B$ of $A$ such that $B[0] \leq B[1] \leq \ldots \leq B[n-1]$

E.g. $A = [18 \ 2 \ 4 \ 5 \ 3]$
     ↓
     $B = [2 \ 3 \ 4 \ 5 \ 18]$

Why sorting?
  → Makes search easier!
     (Power of binary search)
  → Processing sorted data is easier too
    → Find a median ($\Theta(1)$ vs. $\Theta(n)$ time)
    → Find a closest pair/duplicates (see 6.046)

How to sort efficiently?
  First idea: Let's do it iteratively
Insertion sort: Compute iteratively increasingly longer sorted prefixes of A.

\[ \text{For } 1 \leq i \leq n-1: \]
\[ \text{Assume } A[0..i-1] \text{ already sorted (trivially true for } i=1) \]
\[ \text{Make } A[0..i] \text{ sorted by swapping } A[i] \text{ into the correct position:} \]
\[ j = i \]
\[ \text{While } A[j] < A[j-1]: \]
\[ \text{Swap } A[j] \leftrightarrow A[j-1] \]
\[ j = j - 1 \]

\[ \text{Output } A \]

**Example:**

\[
\begin{array}{cccccc}
5 & 3 & 6 & 4 & 12 & \\
\downarrow & & & & & \\
3 & 5 & 6 & 4 & 12 & \\
\downarrow & & & & & \\
3 & 5 & 6 & 4 & 12 & \\
\downarrow & & & & & \\
1 & 3 & 4 & 5 & 6 & 2 \\
\downarrow & & & & & \\
1 & 2 & 3 & 4 & 5 & 6
\end{array}
\]

**Correctness?**
Each phase increases the length of sorted prefix by one.

**Running time?**
\[ O(n^2) \text{: two nested loops of } \leq n \text{ iterations each} \]
\[ O(n^2) \text{: Consider reverse sorted input } A = [m, m-1, \ldots, 1] \]
\[ \Rightarrow \text{Phase } i: \ (i-1) \text{ comparisons & swaps} \]
\[ \Rightarrow \text{Total time: } \sum_{i=1}^{n} (i-1) = \frac{n(n-1)}{2} = O(n^2) \]

\[ \Rightarrow \text{Overall: } T(n) = \Theta(n^2) \]
Can we do better?

**Binary insertion sort:** Insert keys into already sorted prefix via binary search. See Recitation 2 notes.

- Only $\Theta(\log n)$ comparisons per key insertion phase.
- Total # of comparisons $\Theta(n \log n)$ (vs. $\Theta(n^2)$ before).
- But: # of swaps still $\Theta(n^2)$ (in the worst-case) to place the keys in the correct places of the array.

**Exercise:** Would using a doubly linked list instead of an array help here?

Can we get a truly $\Theta(n \log n)$ time algorithm?

**Divide & Conquer approach:**

1. Divide input into part(s)
2. Conquer (solve) each part recursively
3. Combine results(s) to solve the original instance (crucial part: where the "real" work is done)

- If size $n$ input divided into $n_1, \ldots, n_k$ then
  \[ T(n) = \text{divide time} + T(n_1) + \ldots + T(n_k) + \text{combine time} \]

**Recurrence**

- Stock gain problem \([L1]\): $n_1 = n_2 = \frac{n}{2}$, divide $\Theta(n)$, combine: $\Theta(n)$

Assume some base case $T(i) = \text{Const.}$
Merge Sort:
- If \( n = 1 \): done
- Recursively sort \( A[0..\frac{n}{2}] \to L \)
- Recursively sort \( A[\frac{n}{2}+1..n-1] \to R \)
- Merge \( L \) & \( R \) into a single sorted array \( \to output \)

Merge (key routine): Combine two sorted arrays \( L \) & \( R \) into a single sorted array
- Define \( L[\text{len}(L)] = R[\text{len}(R)] = \infty \)
- \( L = r = 0 \)
- If \( L[i] \leq R[r] \): output \( L[i] \)
  \[ L = L + 1 \]
  else: output \( R[r] \)
  \[ r = r + 1 \]
Repeat until would output \( \infty \)

Example:
- \( L: 2 \quad 8 \)
- \( R: 3 \quad 4 \quad 9 \)
Output: \( 2, \quad 3, \quad 4, \quad 8, \quad 9 \)

Correctness of Merge sort?
- Easy proof by induction

Running time of Merge Sort?
1. Divide: \( \Theta(n) \) (\( \Theta(1) \) via implicit splicing)
2. Recursion (Conquer): \( n_1 = n_2 = \frac{n}{2} \) \( \to T(n_1) + T(n_2) = 2T(\frac{n}{2}) \)
3. Merge (Combine): \( \Theta(1) \) per output element \( \to \Theta(n) \) in total
\[ T(n) = 2T(\frac{n}{2}) + \Theta(n) = \Theta(n \log n) \]
Note: Insertion sort works in-place (i.e., needs only $O(n)$ auxiliary space, but Merge Sort requires $O(n)$ extra space. (There is an in-place variant of Merge Sort but is more complicated.)

Many recurrences that arise in context of divide & conquer have the following form:

$$T(n) = a \cdot T\left(\frac{n}{c}\right) + f(n)$$

$a, b$ - constants
(assuming $b > 1$)

How to solve such recurrences?

Recursion tree method:

$$T(n) = a \cdot T\left(\frac{n}{c}\right) + f(n)$$

$$= \begin{cases} 
  f(n) & \text{(a times)} \\
  T\left(\frac{n}{2}\right) & \\
  T\left(\frac{n}{3}\right) & \\
  T\left(\frac{n}{c}\right) &
\end{cases}$$

Total work at each level:

$$f(n) \quad \rightarrow \quad a \cdot f\left(\frac{n}{c}\right)$$

$$a \cdot f\left(\frac{n}{c}\right) \quad \rightarrow \quad a^2 f\left(\frac{n}{c^2}\right)$$

$$\log_b n = O(\log n)$$

levels

$$T(1) \quad T(1) \quad \cdots$$

$$\rightarrow a^{\log_b n} \cdot \Theta(1)$$
Total running time: \[ \text{Sum work on each level} \]
Then sum these sums

\[ T(n) = \sum_{j=0}^{\log_b n} a^j f\left(\frac{n}{b^j}\right) \]

Fun fact: For most such recurrences level sums are either

1. \( \Theta(\text{same}) \Rightarrow \text{total} = \Theta(\text{level sum \# levels}) \)

2. Geometrically decreasing
   \[ \Rightarrow \text{total} = \Theta(\text{top level}) = \Theta(1(n)) \]

3. Geometrically increasing
   \[ \Rightarrow \text{total} = \Theta(\text{bottom level}) = \Theta(\text{\# leaves}) \]
   (Note: \( T(1) = \Theta(1) \) always)

Master theorem: Makes the above more precise

\[ T(n) = a T\left(\frac{n}{b}\right) + f(n) \quad a, b, \text{const}^+ \]

\[ h = \# \text{ of levels} = \log_b n = \Theta(\log n) \]
\[ L = \# \text{ of leaves} = a^h = a^{\log_b n} = n \log a \]

(See next page)
Master Theorem (cont’d)

1. \( f(n) = O(L^{n-c}) = O(n^{\log_a L - c}) \quad \text{(geometrically increasing)} \)
   \[ \Rightarrow T(n) = \Theta(L) = \Theta(n^{\log_a L}) \]

2. \( f(n) = \Theta(L) = \Theta(n^{\log_a L}) \quad \text{(equal levels)} \)
   \[ \Rightarrow T(n) = \Theta(L \cdot h) = \Theta(n^{\log_a L \cdot \log n}) \]

2'. \( f(n) = \Theta(L \log^k n) \quad k > 0 \quad \text{(more general)} \quad \text{(almost equal levels ⇒ log factors carry through)} \)
   \[ \Rightarrow T(n) = \Theta(L \cdot \log^{k+1} n) \]

3. \( f(n) = \Omega(L^{n-c}) = \Omega(n^{\log_a L + c}) \)
   & \( a \cdot f(n/6) \leq (1 - \delta) \cdot f(n) \)
   \[ \Rightarrow T(n) = \Theta(f(n)) \quad \text{(bound on growth of second level)} \]

See textbook for details & proof.

Note: Master theorem is not always applicable

E.g. \( T(n) = 2^n T(n/2) + n^n \quad \text{(a non-const)} \)
\( T(n) = 0.5 T(n/2) + 4 \cdot n \quad \text{(a < 1)} \)
\( T(n) = 64 \cdot T(n/8) - n^2 \log n \quad \text{(f(n) not positive)} \)