Show code + run! - Fewer calls!
"Time?" $n$: amount $k$: # of denominations.

COURSE INFO.

Let's look at tree of recursive calls to count # of calls.

\[ \text{total \# calls?} \]

\[ \begin{align*}
\text{memoized} & : 15 : 3 \\
\text{non-memoized} & : 15 : 3 \\
\text{3: 3 only once!} & : 3 : 3 + 3 + 3 \ldots \text{ (multiple times)} \\
\text{2: 3} & : 
\end{align*} \]
Roughly every value from \( n = 1 \) to \( n \) (all except \( d_1, d_2, \ldots, d_k \)) makes \( k \) recursive calls. \( \Rightarrow \mathcal{O}(nk) \).

Turns out: If U.S. coin denominations, can solve in \( \mathcal{O}(1) \).

\(<\text{HW. Problem}>\).

Hint: Let's review greedy algorithm from last time.

\( \text{Standard Greedy}_\text{change}(n) \) \( \text{// returns fewest number} \)
\( \text{// of coins to make change for } n \)
\( \text{// Denoms must be : 25, 10, 5, 1} \)

\[
\text{numcoins} = 0
\]
\[
\text{sum} = 0
\]
\[
\text{while (sum} \neq n)
\]
\[
x = \text{max} \{ 25, 10, 5, 1 \} \text{ s.t. } \text{sum} + x \leq n
\]
\[
\text{sum} = \text{sum} + x
\]
\[
\text{numcoins} + 1
\]

\[
\text{return numcoins.}
\]

\( \mathcal{O}(n) \text{ loop } n \text{ times, each time find max of } k \text{ denominations.} \)

How to make \( \mathcal{O}(1) \)??
Runtime Analysis: Some review, some new stuff.

Want to express running time of a program/algorithm.

Big-Oh: For an algorithm A with input size n
(1) Express runtime of A as a function of n.
   \[ T(n) = \text{runtime of A} \]
   \[ f(n) = \text{some function in terms of } n \]

(2) Establish relative order \((\text{\#}) \) \((\text{\#}) + O(\text{\#}) \) between growth rates of \( T(n) \) and \( f(n) \).

0) How to compare 2 functions? "Can't just say one is greater than the other".

\[ f(n) = 1000n \quad g(n) = n^2 \]
\[ f(n) > g(n) \text{ for } n < 1000 \]
\[ f(n) < g(n) \text{ for } n > 1000 \]

(2) So compare in terms of relative growth rate.
   Growth rate - how fast a function grows asymptotically (i.e. as \( n \to \infty \))

\[ L \]

\[ \uparrow \]

fastest growing
*Typical growth rates*

**Runtime Notations**

1. \( T(n) = O(f(n)) \) \( T(n) \) grows at a rate \( \leq f(n) \)
   - \( f(n) \) is an upper bound on \( T(n) \)

   \( \text{ex: } T(n) = O(n) \rightarrow \text{run time is at most linear} \)

2. \( T(n) = \Omega(f(n)) \) \( T(n) \) grows at a rate \( \geq f(n) \)
   - \( f(n) \) is a lower bound on \( T(n) \)

   \( \text{ex: } T(n) = \Omega(\log(n)) \rightarrow \text{runtime is at least logarithmic. (Use: can't do better)} \)

3. \( T(n) = \Theta(f(n)) \) \( T(n) \) grows at a rate \( = f(n) \)
   - \( f(n) \) is both upper and lower bound on \( T(n) \)

   \( \text{ex: } T(n) = \Theta(n^2) \rightarrow \text{runtime is "exactly" quadratic} \)

What's the difference? 

- \( T(n) = 3n \quad f(n) = n^2 \)
  
  \( T(n) = O(n^2) \quad T(n) = \Omega(n^2) \quad T(n) = \Theta(n^2) \)

- \( f(n) = n \)
  
  \( T(n) = O(n) \quad T(n) = \Omega(n) \quad T(n) = \Theta(n) \)

- \( f(n) = 1 \)
  
  \( T(n) = O(1) \quad T(n) = \Omega(1) \quad T(n) = \Theta(1) \)
Order of typical growth rates

\begin{align*}
\text{c} & \text{ constant} \\
\log(n) & \text{ logarithmic} \\
\log^2(n) & \text{ log-squared} \\
n & \text{ linear} \\
n \log n & \text{ quadratic} \\
n^2 & \text{ cubic} \\
n^3 & \text{ polynomial} \\
2^n & \text{ exponential}
\end{align*}

"In this class, we'll typically use big-on (O(f(n))) but express in terms of tightest bound.

A tight bound - f(n) expressed in lowest correct order.

ex: T(n) = 1 + 100n^2

Which is the tight f(n)?

T(n) = O(n) - not correct
T(n) = O(n^3) - correct but not tightest
T(n) = O(n^2) - " and tightest

In data structures (201), analyzed runtimes of many algorithms but this is trickier for recursive algs.

Today analyze recursive Merge-Sort."
Quick Review: First divide in half until lists of size 1, then merge

<table>
<thead>
<tr>
<th>12</th>
<th>9</th>
<th>10</th>
<th>2</th>
<th>3</th>
<th>5</th>
<th>8</th>
<th>6</th>
</tr>
</thead>
<tbody>
<tr>
<td>9</td>
<td>12</td>
<td>2</td>
<td>10</td>
<td>3</td>
<td>5</td>
<td>6</td>
<td>8</td>
</tr>
<tr>
<td>2</td>
<td>9</td>
<td>10</td>
<td>12</td>
<td>3</td>
<td>5</td>
<td>6</td>
<td>8</td>
</tr>
<tr>
<td>2</td>
<td>3</td>
<td>5</td>
<td>6</td>
<td>8</td>
<td>9</td>
<td>10</td>
<td>12</td>
</tr>
</tbody>
</table>

Divide-and-Conquer (and Combine)
- Break problem into smaller subproblems (Divide)
- Solve each subproblem (Conquer)
- Combine solutions (Combine)

"Let's look at pseudocode"

MergeSort(A)

if (|A| > 1) // |A| \geq 2 items.
   Step
   1. \text{leftA} = \text{left half of A} \quad \text{Divide} \quad \text{Step} \quad \text{Time} \quad \frac{c}{2}
   2. \text{rightA} = \text{right half of A} \quad \text{Conquer} \quad \text{(later)}
   3. \text{mergeSort(leftA)} \quad \text{Combine} \quad \text{Time} \quad \frac{c}{2}
   4. \text{mergeSort(rightA)} \quad \text{Combine} \quad \text{Time} \quad \frac{c}{2}
   5. \text{merge(leftA, rightA)} \quad \text{Combine} \quad \text{Time} \quad \frac{c}{2}

\frac{3}{2} \times n \quad \text{Scan + Copy}