ex: $d_1 = 12$, $d_2 = 5$, $d_3 = 1$

Recursive step: to need:

Base case: "For which values of $n$ can we solve the problem directly? i.e. without recursion?"
$n = 12, 5, 1, \boxed{0}$

Recursive step: "Imagine coins are laid out."
Start with input $n$.
Suppose we choose $12$ & Now what does the problem input look like? $n-12$"
In general, if choose: mini-problem
\[ \begin{align*}
12 & \quad n-12 \\
5 & \quad n-5 \\
1 & \quad n-1
\end{align*} \]

Let's write pseudocode.
// Initialize Dictionary
for all i
Dictionary[i] = -1

Change(n)
if n > 0 and Dictionary[n] != -1
    return Dictionary[n]

count = 0 // variable to be returned

// Base Cases
if (n = 0)
count = 0
else if (n = 1 or n = 5 or n = 12)
count = 1
else if n < 0
    count = \infty
else // Recursive Step:
count = 1 + min(Change(n-12), 1 Solve each mini-
    \quad \text{change (n-5), } (compute #
    \quad \text{change (n-1)) coins for each denomina-
    \quad \text{increment # of coins}

    2 Choose the denum that yields minimum
    \quad \text{# of coins}

    3 Add to Dictionary
Dictionary[n] = count

return count.

"Need one more base case!"
Suppose \( n=10 \), then a call would be:

\[
\text{Change } (n-12) = \text{Change } (-2).
\]

"What does this mean?"

\[
\Rightarrow \text{Don't choose } 12 \text{! count } = \infty \text{ (or } n+1) \, \text{(add to code)}
\]

**Analyze**
- Correct? [✓]
- Efficient? [?]
- Simple? [✓]
- Useful? [✓] ATMs, currency design, financial transactions

**Efficiency**: Each recursive call takes \( O(1) \). How many calls?

\[
\begin{align*}
\text{Ch}(15) \quad \text{Ch}(15-12) \quad \text{Ch}(15-5) \quad \text{Ch}(15-1) \\
\text{Ch}(3-12) \quad \text{Ch}(3-5) \quad \text{Ch}(3-1) \\
\text{Ch}(2-12) \quad \text{Ch}(2-5) \quad \text{Ch}(2-1)
\end{align*}
\]

\[
\begin{align*}
\text{Ch}(15) & \Rightarrow 1 + \min(\infty, \infty, 2) = 3 \\
\text{Ch}(3) & \Rightarrow 1 + \min(\infty, \infty, 1) = 2 \\
\text{Ch}(2) & \Rightarrow 1
\end{align*}
\]

How many calls do you think are made? \( O(3^n) = O(2^n) \)

**H.W. Problem**

Intuitively: Tree with \( n \) levels, branching factor \( k \) has \( O(k^n) \) nodes. (Actually \( O(k^n) \) leaves)
Learning Goals:
- Speed up Change-Making
- RunTime review

COURSE INFO

"Run program. Let's look at recursive calls made to see if we can speed this up."

"Displaying only recursive calls, no Base Case calls (neg, 0, 1, 5, 12)"

"Notice anything? Many of the recursive calls are made multiple times."

\[
\begin{align*}
\text{cn(15)} \\
\text{cn(3)} \\
\text{cn(2)} \\
\vdots \\
\text{cn(3)} \\
\text{cn(2)}
\end{align*}
\]

Value is recomputed each time!

Instead, what can we do? Store these values!

memoization - store values of a function call

"Maintain dictionary of size n, initialized to -1's, when \text{Change}(i) computed, store in Dictionary[i]."

\[
\text{Dictionary[i]} = \begin{cases} 
-1 & \text{if } \text{Change}(i) \text{ not yet computed} \\
C_i & \text{fewest number of coins to make change for } i.
\end{cases}
\]

Add lines to pseudocode \(\Rightarrow\) (3 changes)
Show code + run! - Fewer calls.

"Time?" n = amount k = # of denominations

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

Ch(15)
  /  \\
 Ch(3) Ch(10) Ch(14)
   / \        \\
   /  \       \\
 3  Ch(2) 2
   \    /  \\
    \  /    \\
    \ /     \\
    / \     \\
   /  \     \\
 1   Ch(10)
  /\     \\
 /  \    \\
/    \   \\
/     \  \\
/      \ \\
/       \\
/        \\
Ch(10)

(total # calls?)

<table>
<thead>
<tr>
<th>n</th>
<th>memoized</th>
<th>non-memo</th>
</tr>
</thead>
<tbody>
<tr>
<td>15</td>
<td>3</td>
<td>3</td>
</tr>
<tr>
<td>3</td>
<td>3</td>
<td>3 + 3 + 3</td>
</tr>
<tr>
<td>2</td>
<td>3</td>
<td>3 + 3 + 3</td>
</tr>
<tr>
<td>10</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Roughly every value from \( n = 1 \) to \( n \) (all except \( d_1, d_2, \ldots, d_k \)) makes \( k \) recursive calls. \( \Rightarrow O(nk) \).

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

```
numcoins = 0
sum = 0
while (sum != n)
    x = max ? 25, 10, 5, 1 ? s.t. sum + x \( \leq \) n
    sum = sum + x
    numcoins ++
return numcoins.
```

**Runtime:** \( O(nk) \) Loop \( n \) times, each time find max of \( k \) denominations. How to make \( O(1) \)??
Summarize:

1. U.S. Denoms
2. Non-U.S. (General)
   - 3. Recursive
   - 4. Greedy
     - 5. Recursive
     - 6. Greedy
     - 7. Non-memo
     - 8. Memo
     - 9. Standard Enhanced
     - 10. Non-memo
     - 11. Memo
     - 12. Standard Enhanced

- $O(k^n)$
- $O(nk)$
- $O(n^k)$
- $O(1)$
- $O(k^n)$
- $O(nk)$
- $O(n^k)$
- $O(n)$

$\Rightarrow$ Optimal!
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:
   \[ T(n) = \text{runtime of } A \] (might be a complicated function)
   \[ f(n) = \text{some "simple" function of } n \]
2. Establish relative order of \( T(n) \) and \( f(n) \).

1. How to compare 2 functions? "Can't just say one is greater than the other".
   \[ T(n) = 1000n \quad f(n) = n^2 \]
   \[ T(n) > f(n) \text{ for } n < 1000 \]
   \[ T(n) < f(n) \text{ for } n > 1000 \]

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

\[ \text{fastest growing} \]
Typical growth rates

Run-time Notations

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

   ex. \( T(n) = O(n) \Rightarrow \) run time is at most linear (won't be worse)

What we'll mostly talk about in this class

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

   ex.: \( T(n) = \Omega(\log(n)) \) runtime is at least logarithmic (can't be better)

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

   ex.: \( T(n) = \Theta(n^2) \) \( \Rightarrow \) runtime is "exactly" quadratic

What's the difference?

ex. \( T(n) = 3n \)

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

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

- \( f(n) = 1 \) \( T(n) = O(1) \) \( T(n) = \Omega(1) \) \( T(n) = \Theta(1) \)
Typical growth rates

\[
\begin{align*}
\text{constant} & \quad c \\
\text{logarithmic} & \quad \log(n) \\
\text{linear} & \quad n \\
\text{efficient} & \quad n \log(n) \\
\text{quadratic} & \quad n^2 \\
\text{cubic} & \quad n^3 \\
\text{polynomial} & \quad n^k \\
\text{exponential} & \quad 2^n
\end{align*}
\]

\[\Rightarrow \text{Go to run-time notations}\]

"In this class, we'll typically use big-Oh but will express in tightest bound."

Tight bound - \( f(n) \) expressed in lowest correct rate.

\[\text{ex: } T(n) = 1 + 100n^2 \quad \text{which is the tight bound?}\]

\[T(n) = O(n) \quad \text{not correct}\]
\[T(n) = O(n^3) \quad \text{correct but not tight}\]
\[T(n) = \Omega(n^3) \quad \text{tight (so correct too)}\]

In CS201 analyzed runtimes of many algorithms but approach is different for recursive algorithms.

Today: analyze recursive MergeSort.