Suppose \( n = 10 \), First call would be
\[ \text{Change} (n-12) = \text{Change} (-2) . \]

"What does this mean?"
\[ \Rightarrow \text{Don't choose 12!} \quad \text{count} = \infty \text{ (or } n+1) \text{ (add to code)} \]

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

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

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

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)
"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."

\[ \text{ch}(15) \]
\[ \text{ch}(3) \]
\[ \text{ch}(2) \]

\[ \text{Value is recomputed each time!} \]

\[ \text{ch}(3) \]
\[ \text{ch}(2) \]

"Instead, what can we do?" Store these values!

\textbf{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].

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

⇒ Add lines to pseudocode \( \implies (2 \text{ changes}) \)
"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.

total # calls?
memorized  non-memorized
15: 3 15: 3
3: 3 only once! 3: 3 + 3 + 3 ... (multiple times)
2: 3
Roughly every value from \( n = 1 \) to \( n \) (all except \( d_1, d_2, \ldots, d_k \)) makes \( K \) recursive calls. \( \Rightarrow O(n^k) \).

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

HW Problem:

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

```
not most efficient!
StandardGreedyChange(n); // returns fewest number
   // of coins to make change for n.
   // Denoms must be : 25, 10, 5, 1.
   numcoins = 0
   sum = 0
   while (sum != n)
      x = max of 25, 10, 5, 1 s.t. sum + x <= n
      sum = sum + x
      numcoins ++

   return numcoins.
```

RunTime? \( O(n^k) \) loop \( n \) times, each time find max of \( k \) denominations.

How to make \( O(1) \)??
Summarize:

1. U.S. Demands
   3. Recursive
   4. Greedy
      7. Non-memo
      8. Memo
      10. Standard
      11. Enhanced

2. Non-U.S.
   5. Recursive
   6. Greedy
      13. Non-memo
      9. Memo

3. \( O(k^n) \)
4. \( O(n \cdot k) \)
5. \( O(n \cdot k) \)
6. \( O(1) \)
12. \( O(n) \)
14. \( O(k^n) \)
15. \( O(nk) \)
16. \( O(n) \)

No solution