Algorithm - set of steps that solve a problem

Analyzing Algorithms
1. correctness
2. efficiency (time/space)
3. simplicity (clarity)
4. applications (usefulness)

"Typically, we'll describe a problem, with input, and design an algorithm to solve."

"Some problems/alg. you've already seen:"

<table>
<thead>
<tr>
<th>Problem</th>
<th>Input</th>
<th>Alg</th>
<th>Output/Solution</th>
</tr>
</thead>
<tbody>
<tr>
<td>Sorting</td>
<td>list</td>
<td>Quicksort</td>
<td>sorted list</td>
</tr>
<tr>
<td>Searching</td>
<td>list</td>
<td>Binary Search</td>
<td>true/false, index</td>
</tr>
<tr>
<td>&quot;</td>
<td>graph</td>
<td>BFS</td>
<td>true/false, path</td>
</tr>
</tbody>
</table>

Another Problem + Algorithm
Change-Making Problem

Input - coins (25¢, 10¢, 5¢, 1¢) (unlimited amount)

amount n

Output - Fewest number of coins to make change of n

Volunteer?
Algorithm? Keep choosing largest coin s.t. n not exceeded until n obtained.

ex: $n = 41 \Rightarrow$ Soln: 25, 10, 5, 1 4 coins

Example of "Greedy" Algorithm - at every stage, choose what appears to be best current choice. Doesn't think about the future. Party algorithm?

Let's change coin denominations:

ex: Coins: 28, 18, 5, 1

\{ 12, 5, 1 \}

$n = 15$

Greedy: 12, 1, 1, 1 4 coins

Optimal: 5, 5, 5 3 "

For general coin denominations $d_1 > d_2 > d_3 \ldots d_k$, greedy doesn't work.

Instead use a recursive algorithm.

Idea: Find the optimal number of coins for n by finding optimal number for smaller amounts first

ex: $d_1 = 12$, $d_2 = 5$, $d_3 = 1$
Recursive, so need:

Base case: For which values of \( n \) can we solve the problem directly? Without recursion? \( n = 12, 5, 1, 0 \)  

Recursive step: Imagine coins are laid out. Start with \( n \).

Suppose we choose 12¢. Now what does the problem input look like? \( n - 12 \)

In general, if chose:

\[
12 \times \Rightarrow \quad n - 12 \\
5 \times \Rightarrow \quad n - 5 \\
1 \times \Rightarrow \quad n - 1
\]

Let’s write out pseudocode.
Change(n)
  if (n > 0) and Dictionary[en] != -1)
  return Dictionary[en]

  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 (\text{change(n-12)}, \text{compute # coins increment # coins by 1 change(n-5)}, \text{for each den change(n-1)})

  // Add to Dictionary
  Dictionary[en] = count

  return count

Need one more base case!
Suppose n=10. First call would be:

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

"What does this mean?"

Don't choose !2&!; count = 0 or n+1 (add to code)

Analyze:
Correct? ✓
Efficient? ?
Simple? ✓
Applications?

Efficiency: let's look at calls made. What calls made for n = 15?

\[
\begin{align*}
\text{ch(15)} & \\
| & \\
\text{ch(15-12)} & \text{ch(15-5)} \text{ ch(15-1)} \\
\text{ch(3)} & = 1+2+3 \\
\infty & \infty = \text{ch(2)} = 1+1+1+2 \\
& \\
\text{ch(-10)} & \text{ch(3)} \text{ ch(1)} \\
& = \infty = \infty = 1
\end{align*}
\]

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 \( n(2^k) \) 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.

\[
\begin{align*}
\text{ch}(15) \quad \text{value is recomputed each time!} \\
\text{ch}(3) \\
\{ \\
\text{ch}(2) \\
\} \\
\text{ch}(3) \\
\text{ch}(2)
\end{align*}
\]

Instead, what can we do? "Store this information".

Memoization - store values of a function call.

Maintain dictionary of size \( n \), initialized to -1's.

When \( \text{change}(i) \) computed, store in Dictionary \( \text{Eij} \).

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

⇒ Add lines to pseudocode ⇒