Algorithm - set of steps that solve a problem

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

"Typically we will describe a problem, with input and design an algorithm to solve it"

"Put terms (problem, input, algorithm into context by relating to some algorithm 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>sorted list</td>
<td>Binary Search</td>
<td>true/false, index</td>
</tr>
<tr>
<td>searching</td>
<td>graph</td>
<td>BFS</td>
<td>true/false, path</td>
</tr>
</tbody>
</table>

"Another problem"

Change-Making Problem
Input - coins (25¢, 10¢, 5¢, 1¢) (unlimited amount)
Value n
Output - fewest number of coins to make change of n

Volunteer? n = 41
Algorithm? Continuously choose largest coin s.t. n not exceeded until n is obtained.

Ex: $n = 41 \Rightarrow \text{soln: } 25, 10, 5, 1 \quad 4 \text{ coins}$

Example of a Greedy Algorithm - at every stage, choose what appears to be the best current choice.

"Doesn't think about the future - Party Algorithm"

"Let's change coin denominations"

Ex: Coins $2\$ , $1\$, 5, 1

Ex: $12, 5, 1\$

$n = 15$ Greedy: $12, 1, 1, 1 \quad 4 \text{ coins}$

Optimal: $5, 5, 5 \quad 3 \"$

For general coin denominations $d_1 > d_2 > \ldots > d_k$

Greedy doesn't work.

"Instead, we use a recursive algorithm."

Recall: solve the problem by solving smaller versions of the problem first."

Idea - Find optimal number of coins for $n$ by finding optimal number for smaller values first.
ex: \( d_1 = 12 \), \( d_2 = 5 \), \( d_3 = 1 \)

Recursive Step: need:

Base Case: "For which values of \( n \) can we solve the problem directly? i.e. without recursion?"
\( n = 12, 5, 1, 10 \)

Recursive Step: "Imagine coins are laid out"
Start with input \( n \)
Suppose we chose 12¢. Now what does the problem input look like? \( n = 12 \)
In general, if choose: mini-problem
\[
\begin{align*}
12 \& n - 12 \,
5 \& n - 5 \,
1 \& n - 1
\end{align*}
\]
Recurrsively solve these mini-problems first.

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

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 = ∞
  else // Recursive Step:
    count = 1 + min(change(n-12),
                     change(n-5),
                     change(n-1))

  // add to Dictionary
  Dictionary[n] = 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?"

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

Analyse
Correct? \( \sqrt{\square} \)
Efficient? \( \square \)
Simple? \( \sqrt{\square} \)
Useful? \( \sqrt{\square} \)

ATMs, currency design, financial transactions

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

\[
\begin{align*}
\text{Ch}(15) &= 1 \\
\text{Ch}(15-12) &= \text{Ch}(3) = 1 + \min(\infty, \infty, 2) = 2 \\
\text{Ch}(15-5) &= \text{Ch}(10) = 1 + \min(\infty, \infty, 2) = 2 \\
\text{Ch}(15-1) &= \text{Ch}(14) = 1 + \min(\infty, \infty, 2) = 2 \\
\text{Ch}(3-12) &= \infty \\
\text{Ch}(3-5) &= \text{Ch}(2) = 1 + \min(\infty, \infty, 1) = 2 \\
\text{Ch}(3-1) &= \text{Ch}(2) = 2 \\
\end{align*}
\]

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."

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

\[
\begin{align*}
\text{Value is recomputed each time!} \\
\end{align*}
\]

"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,j]."

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

⇒ Add lines to pseudocode ⇐ (3 changes)