Making Change

Goal. Given currency coin denominations, e.g., \{100, 25, 10, 5, 1\} devise a method to pay an integer amount using the fewest coins.

Example: 34¢.

Cashier’s algorithm. At each iteration, add coin of the largest value that does not take us past the amount to be paid.

Example: $2.89 = 289¢.
**Greedy Algorithm**

*Cashier’s algorithm.* Use the maximal number of the largest denomination coin 

\[ x - \text{amount to be changed} \]

**Sort** coins denominations by value: \( c_1 < c_2 < \ldots < c_n \).

\( S \leftarrow \text{empty} \) coins selected

**while** (\( x > 0 \)) {

\( \text{let } k \text{ be largest integer such that } c_k \leq x \)

\( \text{if } (k == 0) \# \text{all } c_k > x \)

\( \text{return } \text{"no solution found"} \)

\( x \leftarrow x - c_k \)

append(\( S,k \))

}\n
**return** \( S \)

Does this Greedy algorithm always work?

---

**Greedy doesn’t always work**

1. **Greedy fails changing 30 optimally** with coin set \{25, 10, 1\} as it produces \[25,1,1,1,1,1\] instead of \[10,10,10\]

2. **Greedy fails changing 30 at all** with coin set \{25, 10\} even though there is a solution: \[10,10,10\]

3. But the Greedy algorithm works for US coin set

   **Proof:** number theory (canonical coin systems)
Different problem: number of ways to pay

Given a sorted coin set coins = \{c_0, c_1, ..., c_{d-1}\} c_0 the smallest coin value, and c_{d-1} the largest coin value, and an amount M

how many different ways can M be paid?

One possible recursive either / or solution: go backwards through coins and choose to use the largest remaining coin or not

\texttt{mkCh}(n, c):
\begin{itemize}
  \item \# \text{n}: amount still to be paid
  \item \# \text{c}: index of coins value currently considered
  \item \textbf{Base}: 
    \begin{itemize}
      \item if \text{c} == 0, how many ways? (is there always a way?)
    \end{itemize}
  \item \textbf{Step}:
    \begin{itemize}
      \item if \text{c}>0
        \begin{itemize}
          \item if largest coin cannot be used: consider \text{coin}_{c-1}
          \item else: # it can be used
            \begin{itemize}
              \item \textbf{either} use one \text{coin}_c and keep considering \text{coin}_c
              \item or don't use \text{coin}_c and thus consider \text{coin}_{c-1}
            \end{itemize}
        \end{itemize}
    \end{itemize}
\end{itemize}

Making change vs. knapsack

\textbf{Recurrence}:

\begin{equation}
\text{ways}(\text{amount}, \text{i}) = \\
1. \text{Base case?} \\
2. \text{If amount < coin[i]: ways(i-1, amount)} \\
3. \text{Else: ways(amount-coin[i],i) + ways(amount, i-1)}
\end{equation}

\textbf{Making change is very similar to knapsack, but}:

\begin{itemize}
  \item 1. We take the sum, not the maximum, of the two options.
  \item 2. We must use the same coin value a number of times. How this is reflected in the recurrence?
\end{itemize}
Example of the recursive solution

coins = [1, 5, 10, 25] M = 29

use

4,3

Quarters

29,3

don't use

29,2

Dimes

19,2

9,2

Nickels

19,1

4,1

Pennies

14,1

9,1

Complete this call tree

Making Change Dynamic Programming

Go through the state space bottom-up: i=0 to n-1

- select coin type
  - first 1 coin type, then 1&2, ......, finally all coin types
  - what does the first column look like?
- use solutions of smaller sub-problems to compute solutions of
  larger ones by storing previous values. Which values do you
  need to preserve?

In the recursive solution (DC) there are
2 (recursive) sub-problems. In the dynamic
programming solution there are 2 reads:

don't use current coin

use current coin
Programming Assignment

1. Write a recursive mkChange function based on the either or choices from slide 6, then turn it into a Dynamic Programming function.
   - Do you need a 2 D table here?

2. Determine the performance of the two algorithms. Later, in a written assignment, you will plot your data, and infer $O$ complexity:
   - Recursive: count number of calls
   - Dynamic programming: count number of table reads