Designing Generative Recursion

When you discover that the design recipe isn't working, **stop writing code**

Instead, figure out the *algorithm*

- What is the trivial case?

- What are the smaller sub-problems, and how are their solutions combined?

  Generating sub-problems or combining the answers may require additional functions
Generating Sub-Problems

The key to a sub-problem is that it looks like the original problem (only smaller)

Example: In `odd-items`, the sub-problem is a smaller list from which we want the odd items

Homework: In `colors->list`, the sub-problem should be a smaller list from which to extract rows

Guideline: When the result is a list, try to generate the first item in the list, then create a sub-problem for the rest of the list
New Example

Suppose that instead of rows, we want to convert an image into a list of columns

\[
\text{(colors→columns (list color1 color2 color3 color4 color5 color6)) 3)
\]

"should be" (list (list color1 color4)
                 (list color2 color5)
                 (list color3 color6))

Structural recursion doesn't work well
Designing the Column Converter

(colors->columns (list color1 color2 color3
    color4 color5 color6)
3)
"should be" (list (list color1 color4)
    (list color2 color5)
    (list color3 color6))

The result is a list of columns:

• Can we get the first column?
• Can we create a list with only the other columns?
Designing the Column Converter

(colors->columns (list color1 color2 color3
                   color4 color5 color6)
                    3)
"should be" (list (list color1 color4)
                   (list color2 color5)
                   (list color3 color6))

(colors->columns (list color1 color2 color3
                   color4 color5 color6)
                    3)
→
(cons (list color1 color4)
       (colors->columns (list color2 color3
                        color4 color5 color6)
                        2)))
Designing the Column Converter

(colors→columns (list color1 color2 color3 color4 color5 color6) 3)
"should be" (list (list color1 color4)
(list color2 color5)
(list color3 color6))

; extract-first-column : list-of-color num → list-of-color

; drop-first-column : list-of-color num → list-of-color
Implementing the Column Converter

(define (colors->columns l n)
  (cond
   [(empty? l) empty]
   [else
    (local [(define cl
              (extract-first-column l n))
            (define rl
              (drop-first-column l n))]
      (cons cl
        (colors->columns rl (sub1 n))))]))

With two pending wishes...
Designing Extract

Now to satisfy our wish for \texttt{extract-first-column}...

\[
\text{(extract-first-column (list color1 color2 color3 color4 color5 color6)) 3)}
\]

"should be" \texttt{(list color1 color4)}

Again, structural recursion doesn't work well

- Can we get the first item in the column?
- Can we create a list whose first column is the rest of the column?
Designing Extract

Now to satisfy our wish for \texttt{extract-first-column}...

\begin{verbatim}
(extract-first-column (list color1 color2 color3
                       color4 color5 color6) 3)
"should be" (list color1 color4)

(extract-first-column (list color1 color2 color3
                       color4 color5 color6) 3)
→ (cons color1
     (extract-first-column
      (list color4 color5 color6) 3))

; skip-n : list-of-X nat -> list-of-X
\end{verbatim}
Implementing Extract

(define (extract-first-column l n)
  (cond
    [(empty? l) empty]
    [else
      (cons
        (first l)
        (extract-first-column (skip-n l n) n))])))

Implementing \texttt{skip-n} is an exercise in structural recursion on \texttt{nat}.
Designing Drop

Finally, to satisfy our wish for \texttt{drop-first-column}...

\begin{verbatim}
(drop-first-column (list color1 color2 color3 color4 color5 color6) 3)
"should be" (list color2 color3 color5 color6)
\end{verbatim}

Yet again, structural recursion doesn't work well

- Can we get the first item in the result?
- Can we create a list where dropping the first column is the rest of the answer?
Designing Drop

Finally, to satisfy our wish for `drop-first-column`...

```
(drop-first-column (list color1 color2 color3 color4 color5 color6) 3)
"should be" (list color2 color3 color5 color6)
```

```
(drop-first-column (list color1 color2 color3 color4 color5 color6) 3)
```

→

```
(cons color2
  (drop-first-column ??? 3))
```
Designing Drop

Finally, to satisfy our wish for `drop-first-column`...

```
(drop-first-column (list color1 color2 color3
color4 color5 color6) 3)

"should be" (list color2 color3
color5 color6)
```

• Can we create a list where dropping the first column is the rest of
the answer?

No – getting just the first item doesn't make a similar sub-problem
Designing Drop

Finally, to satisfy our wish for `drop-first-column`...

```
(drop-first-column (list color1 color2 color3
                    color4 color5 color6)
                   3)
"should be" (list color2 color3
              color5 color6)
```

Need to grab an entire row, then skip the row to recur

```
(drop-first-column (list color1 color2 color3
                    color4 color5 color6)
                   3)
→
(append (list color2 color3)
        (drop-first-column (list color4 color5 color6) 3))
```
Implementing Drop

(define (drop-first-column l n)
  (cond
    [(empty? l) empty]
    [else
     (append
      (first-n (rest l) (sub1 n))
      (drop-first-column (skip-n l n)))]))

; first-n : list-of-X nat -> list-of-X
; snip-n : list-of-X nat -> list-of-X

The leftover wishes are straightforward
Another Example

• Implement \texttt{replace-range}, which takes a list, two numbers \textit{start} and \textit{end}, and a value \( v \); the result is a list like the given one, except that \( v \) replaces the elements in positions \textit{start} to \textit{end} inclusive.

\[
\text{\texttt{replace-range}} : \quad \text{list-of-X num num X -> list-of-X}
\]

\[
(\text{\texttt{replace-range}} \ ' (a b c d e) 1 3 'x) \quad "\text{should be}" \\
' (a x x x x e)
\]
Designing Replacement

\[(\text{replace-range } '(a \ b \ c \ d \ e) \ 1 \ 3 \ 'x)\]

"should be"

'\( (a \ x \ x \ x \ e) \)

\(\text{---------------------------}\)

\[(\text{replace-range } '(a \ b \ c \ d \ e) \ 1 \ 3 \ 'x)\]

\[\rightarrow\]

\[(\text{cons } 'a\]

\[(\text{replace-range } '(b \ c \ d \ e) \ 0 \ 2 \ 'x)\)]
Designing Replacement

\[
\begin{align*}
(\text{replace-range} &\ '\ (a\ b\ c\ d\ e)\ 1\ 3\ 'x) \\
\text{"should be"} \\
&\ '(a\ x\ x\ x\ x\ e) \\
\hline
(\text{replace-range} &\ '\ (a\ b\ c\ d\ e)\ 1\ 3\ 'x) \\
\rightarrow \\
(\text{cons} &\ 'a \\
(\text{replace-range} &\ '\ (b\ c\ d\ e)\ 0\ 2\ 'x)) \\
\rightarrow \\
(\text{cons} &\ 'a \\
(\text{cons} &\ 'x \\
(\text{cons} &\ 'x \\
(\text{replace-range} &\ '\ (c\ d\ e)\ -1\ 1\ 'x)))))
\end{align*}
\]
Designing Replacement

\[(\text{replace-range} \ (\text{a b c d e}) \ 1 \ 3 \ 'x)\]

"should be"

\[\ (a \ x \ x \ x \ e)\]

\[\rightarrow \rightarrow\]

\[(\text{cons} \ 'a)\]

\[(\text{cons} \ 'x)\]

\[
\ldots
\]

\[(\text{replace-range} \ (e) \ -3 \ -1 \ 'x))\]

\[\rightarrow\]

\[(\text{cons} \ 'a)\]

\[
\ldots \ldots \ldots
\]

\[(\text{cons} \ 'e)\]

\[(\text{replace-range} \ \text{empty} \ -4 \ -2 \ 'x)))\]
(define (replace-range l s e v)
  (cond
    [(empty? l) empty]
    [else (cons (cond
                  [(and (< s 1) (> e -1)) v]
                  [else (first l)])
         (replace-range (rest l)
                        (sub1 s)
                        (sub1 e)
                        v))])))
Designing Generative Recursion

Finding the recursive sub-problem is the key

- Think first, write code second
- Writing down example steps can help