Designing Generative Recusion

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

```
(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
                             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 c1
               (extract-first-column l n))
             (define rl
               (drop-first-column l n))]
       (cons c1
             (colors->columns rl (sub1 n))))]))
```

With two pending wishes...

Designing Extract

Now to satisfy our wish for **`extract-first-column`**...

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

```
(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
```

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 **skip-n** is an exercise in structural recursion on **nat**

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)
```

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 strightforward

Another Example

- Implement **`replace-range`**, which takes a list, two numbers *start* and *end*, and a value *v*; the result is a list like the given one, except that *v* replaces the elements in positions *start* to *end* inclusive

```
; replace-range :
;   list-of-X num num X -> list-of-X

(replace-range '(a b c d e) 1 3 'x)
"should be"
'(a x x x e)
```

Designing Replacement

```
(replace-range '(a b c d e) 1 3 'x)
"should be"
'(a x x x e)
```

```
(replace-range '(a b c d e) 1 3 'x)
→
(cons 'a
      (replace-range '(b c d e) 0 2 'x))
```

Designing Replacement

```
      (replace-range '(a b c d e) 1 3 'x)
      "should be"
      '(a x x x e)
```

```
(replace-range '(a b c d e) 1 3 'x)
→
(cons 'a
      (replace-range '(b c d e) 0 2 'x))
→
(cons 'a
      (cons 'x
            (replace-range '(c d e) -1 1 'x)))
```

Designing Replacement

```
(replace-range '(a b c d e) 1 3 'x)
"should be"
'(a x x x e)
```

```
→ →
(cons 'a
   (cons 'x
      ...
      (replace-range '(e) -3 -1 'x)))
→
(cons 'a
   ... ...
   (cons 'e
      (replace-range empty -4 -2 'x)))
```

Implementing Replacement

```
(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