Breaking Down Racket Problems: A Step-by-Step Guide

1. Sum of a List

Problem: Write a function that takes a list of numbers and returns their sum.

Concepts Used: Recursion, Base case, car, cdr

Breakdown:

- Think about the base case. When the list is empty, what should the function return?
- To break down the problem, consider taking the first element (car lst) and adding it to the sum of the rest (cdr lst).
- Recursively call the function on the rest of the list.
- How does the function eventually terminate?

Racket Hint:

```
(define (sum-list lst)
 (if (null? lst)
   0
   (+ (car lst) (sum-list (cdr lst)))))
```

---

2. Filtering Even Numbers

Problem: Implement a function that removes odd numbers from a list.

Concepts Used: filter, Predicate functions

Breakdown:

- The filter function takes a predicate (a function that returns #t or #f).
- The predicate function should check if a number is even (even?).
- filter will keep elements that return #t when passed to the predicate.

Racket Hint:

```
(define (filter-even lst)
 (filter even? lst))
```

---

3. Mapping Over a List

Problem: Write a function that doubles every element in a list.

Concepts Used: map, Higher-order functions

Breakdown:

- The map function applies a given function to each element of a list.
- You need to define a function that doubles a number.
- The result should be a new list with transformed elements.

Racket Hint:

```
(define (double-list lst)
  (map (lambda (x) (* 2 x)) lst))
```

---

4. Finding the Maximum Element

Problem: Find the largest number in a list.

Concepts Used: Recursion, Comparison, Base case

Breakdown:

- Base case: If the list has one element, return that element.
- Compare the first element with the maximum of the rest of the list.
- Recursively reduce the problem size.

Racket Hint:

```
(define (max-list lst)
  (if (null? (cdr lst))
    (car lst)
    (max (car lst) (max-list (cdr lst)))))
```

---

5. Reversing a List

Problem: Reverse a given list.

Concepts Used: Recursion, List construction

Breakdown:

- Base case: An empty list should return an empty list.
- Append the first element to the reversed rest of the list.

Racket Hint:

```
(define (reverse-list lst)
 (if (null? lst)
   '()
   (append (reverse-list (cdr lst)) (list (car lst)))))
```

---

6. Counting Elements in a List

Problem: Count how many elements are in a list.

Concepts Used: Recursion, Base case, Accumulator pattern

Breakdown:

- Base case: An empty list has a count of 0.
- Recursively call the function on the rest of the list, adding 1 at each step.

Racket Hint:

```
(define (count-list lst)
 (if (null? lst)
   0
   (+ 1 (count-list (cdr lst)))))
```

---

7. Checking if a List is Sorted

Problem: Write a function that checks if a list is sorted in ascending order.

Concepts Used: Recursion, Pairwise comparison

Breakdown:

- Base case: A single element or an empty list is always sorted.
- Compare the first two elements; if they are in the wrong order, return #f.
- Recursively check the rest of the list.

Racket Hint:

```
(define (sorted? lst)
  (or (null? (cdr lst))
     (and (<= (car lst) (cadr lst))
        (sorted? (cdr lst)))))
```

---

8. Flattening a Nested List

Problem: Convert a nested list into a single-level list.

Concepts Used: Recursion, append

Breakdown:

- Base case: An empty list returns an empty list.
- If the first element is a list, recursively flatten it.
- Use append to merge results.

Racket Hint:

```
(define (flatten lst)
  (cond [(null? lst) '()]
      [(list? (car lst)) (append (flatten (car lst)) (flatten (cdr lst)))]
      [else (cons (car lst) (flatten (cdr lst)))]))
```

---

9. Generating Factorials

Problem: Compute the factorial of a number.

Concepts Used: Recursion, Base case

Breakdown:

- Base case: 0! is 1.
- Recursive case: n! = n * (n-1)!.

Racket Hint:

```
(define (factorial n)
 (if (= n 0)
   1
   (* n (factorial (- n 1)))))
```

---

10. Fibonacci Sequence

Problem: Generate the nth Fibonacci number.

Concepts Used: Recursion, Overlapping subproblems

Breakdown:

- Base cases: fib(0) = 0, fib(1) = 1
- Recursive relation: fib(n) = fib(n-1) + fib(n-2)
- Why is naive recursion inefficient? (Think about memoization.)

Racket Hint:

```
(define (fib n)
 (if (<= n 1)
   n
   (+ (fib (- n 1)) (fib (- n 2)))))
```

Example Breakdown: Lambda Calculus Reductions

Problem 1: Reducing (λx. x x) (λy. y)

Step 1: Understanding the Expression

- We have a function **λx. x x** applied to another function **λy. y**.
- In lambda calculus, function application means substituting the argument into the function.

Step 2: Applying Beta Reduction

- The function **λx. x x** expects an input and applies it to itself.
- We substitute **(λy. y)** for **x** in **x x**:

(λx. x x) (λy. y) → (λy. y) (λy. y)

Step 3: Evaluating the New Expression

- Now we have **(λy. y) (λy. y)**.
- The function **λy. y** is the identity function, meaning it returns whatever is given to it.
- Applying it to **λy. y** results in:

(λy. y) → λy. y

- The expression reduces to **λy. y**, which is the simplest form (normal form).

Key Concepts in This Reduction:

- **Beta Reduction:** Replacing a bound variable with its argument.
- **Identity Function: λy. y** always returns its input.
- **Normal Form:** When no further reductions can be applied.

Problem 2: Reducing (λx. x (λz. z)) (λy. y y)

Step 1: Understanding the Expression

- The function **λx. x (λz. z)** is applied to **λy. y y**.
- This means **x** will be replaced with **(λy. y y)** in the expression **x (λz. z)**.

Step 2: Applying Beta Reduction

- Substitute **(λy. y y)** for **x** in **x (λz. z)**:

(λx. x (λz. z)) (λy. y y) → (λy. y y) (λz. z)

Step 3: Evaluating the New Expression

- Now we have **(λy. y y) (λz. z)**.
- **λy. y y** applies itself to its argument, which is **λz. z**.
- This results in:

(λz. z) (λz. z)

- The function **λz. z** is the identity function, so it just returns **λz. z**.
- The final result is **λz. z**, which is the normal form.

Key Concepts in This Reduction:

- **Function Substitution:** Replacing **x** with its argument.
- **Self-Application: λy. y y** applies itself to an argument.
- **Normal Form:** The simplest form after reduction.

General Notes for Understanding Lambda Calculus:

- **Alpha Conversion:** Renaming bound variables to avoid confusion.
- **Beta Reduction:** The main step of computation in lambda calculus.
- **Eta Reduction:** Simplifying functions when possible (e.g., **λx. f x → f** if **x** does not appear in **f**).

This document should help you practice breaking down lambda expressions step by step!