Recipe from HTDP sections I and II

Step 1 - Problem Analysis and Data Definitions

- Identify the information that must be represented and how it is represented in the chosen programming language. Formulate data definitions and illustrate them with examples.
- Define the program constants, everything that doesn't change over time when the program is running is a constant.
- Everything that changes over time, needs a **Data Definition**.
- If a problem statement is about information of arbitrary size, you need a self-referential data definition to represent it.

Step 2 - Signature, Purpose Statement, Header

- State what kind of data the desired function(s) consumes and produces. Formulate a concise answer to the question *what* the function computes. Define a stub (header) that lives up to the signature.
- Make a **Wish List**: Maintain a list of function headers that must be designed to complete a program. Each entry on a wish list should consist of three things: a meaningful name for the function, a signature, and a purpose statement.

Step 3 - Functional Examples

- Work through examples that illustrate the function's purpose. (Most of the time, this is Unit Tests)
- If the function's argument is a Data containing **intervals**, be sure to make examples for the extremes of the interval, also cover all **Items** on the Data Definition if any.

Step 4 - Function Template

- Translate the data definitions (from function arguments) into an outline of the function.
- Put constants that might be used to compute the answer on the outline of the function.

Question	Answer
Does the data definition distinguish among different sub-classes of data?	Your template needs as many 'cond' clauses as sub-classes that the data definition distinguishes.
How do the sub-classes differ from each other?	Use the differences to formulate a condition per clause.
Do any of the clauses deal with structured values?	If so, add appropriate selector expressions to the clause.
Does the data definition use self-references?	Formulate "natural recursions" for the template to represent the self-references of the data definition.
If the data definition refers to some other data definition, where is this cross-reference to another data definition?	Specialize the template for the other data definition. Refer to this template.

Step 5 - Function Definition

- Fill in the gaps in the function template. Exploit the purpose statement and the examples.

- The more complex you make your data definitions, the more complex this step becomes. If the function argument is a Structure, Determine what the pieces of the template compute from the given inputs. Then consider **how to combine these pieces** (plus some constants) to compute the desired output. Keep in mind that you might need an *auxiliary function*.
- If the function argument is a Self-referential Data, follow the question-answer game:

Question	Answer
What are the answers for the non-recursive 'cond' clauses?	The examples should tell you which values you need here. If not, formulate appropriate examples and tests
What do the selector expressions in the recursive clauses compute?	The data definitions tell you what kind of data these expressions extract, and the interpretations of the data definitions tell you what this data represents.
What do the natural recursions compute?	Use the purpose statement of the function to determine what the value of the recursion means, **not how it computes this answer**. If the purpose statement doesn't tell you the answer, improve the purpose statement.
How can the function combine these values to get the desired answer?	Find a function that combines the values. Or, if that doesn't work, make a wish for a helper function. For many functions, this last step is straightforward. The purpose, the examples, and the template together tell you which function or expression **combines** the available values into the proper result. We refer to this function or expression as a ***combinator***.
So, if you are stuck here, ...	... arrange the examples from the third step in a table. Place the given input in the first column and the desired output in the last column. In the intermediate columns enter the values of the selector expressions and the natural recursion(s). Add examples until you see a pattern emerge that suggests a combinator.
If the template refers to some other template,what does the auxiliary function compute?	Consult the other function's purpose statement and examples to determine what it computes, and assume you may use the result even if you haven't finished the design of this helper function.

- The discover of the need for auxiliary functions during the function definition (step 5) is a natural thing. To help with that, follow:
    1. If the composition of values requires knowledge of a particular domain of application—for example, composing two (computer) images, accounting, music, or science—design an auxiliary function.
    2. If the composition of values requires a case analysis of the available values—for example, depends on a number being positive, zero, or negative— use a cond expression. If the cond looks complex, design an auxiliary function whose arguments are the template's expressions and whose body is the cond expression.

3. If the composition of values must process an element from a self-referential data definition—a list, a natural number, or something like those—design an auxiliary function.
4. If everything fails, you may need to design a **more general** function (in other words, *solve a more general problem*, based on the original problem statement) and define the main function as a specific use of the general function. This suggestion sounds counterintuitive, but it is called for in a remarkably large number of cases.

Step 6 - Testing

- Articulate the examples as tests (if you used unit tests, they are already defined on step 3) and ensure that the function passes all. Doing so discovers mistakes. Tests also supplement examples in that they help others read and understand the definition when the need arises.
- If the result doesn't match the expected output (from Step 3), consider the following three possibilities:
  a. You miscalculated and determined the wrong expected output for some of the examples.
  b. Alternatively, the function definition computes the wrong result. When this is the case, you have a *logical error* in your program, also known as a *bug*.
  c. Both the examples and the function definition are wrong.

Notes

Step 1

- If it is impossible to generate examples from the data definition, it is invalid. If you can generate examples for self-referential data but you can't see how to generate increasingly larger examples, the definition may not live up to its interpretation.

Step 2

- When you do formulate the purpose statement, focus on what the function computes not how it goes about it, especially not how it goes through instances of the given data.