Problem Solving with Algorithms
Correct Thinking leads to correct Code!
Michael Mattie(codermattie@gmail.com)

Behavior – STATE & CLARIFY

- **do** – effects in state or behavior
- **return** – answers in computational results

Inputs – STATE & CLARIFY

- types and scale

Definition – paradigm and design from Behavior & Inputs

- **Scenarios** – All of the **expectations** of the program based on Behavior and Inputs
- **Paradigm** (Comprehension)
    - what model is best suited to framing the problem (UI, System Design, Data Flow, Query/Relational, REST, Use Case, Sequence, UML)
    - spot check the model to see if it adequately describes the problem.
    - Find key characteristics and underlying properties

  Solution (Design)
    - Conduct thought experiments, Give up on bad ideas quickly
    - Manage complexity of solution and code
    - **Identify** essential state (objects) - **maximize** idempotent functions (API)
    - **sketch the code** in functions, loops, with comments
    - **decide what techniques** will be used to optimize and implement the algorithm: dynamic, recursion, linguistic (DSL), query, logic, single pass, multi-pass, pre-compute, multi-process)

ALGORITHM – **scenarios, objects, functions, and loops**

- **Objects** represent state **(modality)**

- **Functions** represent idempotent computations **(API)**

- **Code Sketch**
  - INITIALIZE: establish a **return value**, empty containers over nulls
  - TERMINATE - determine the base case. **When is it done**?
  - FIRST, MIDDLE, LAST **Cases**
  - CORNER cases - Input **validation**, System errors, stale state, deadlocks**,** and sync errors, timeouts
  - INVARIANTS – statements always true in the procedure's execution
    - defined - on computational cases or state
    - Computational cases – what holds true in loops and logic
    - State - initialize, maintain, terminate

SOLUTION MODELS

RECURSION – n*log$_x$(n) {where x is the partition size}
- REDUCE: to the recurrence of the essence. Recurse to the depth of the solution data structures, never to the input
- Define: TERMINATION as return combining the recurrence with the recursive call
- SOLVE: the problem by computing part of the problem

DIVIDE & CONQUER – n*log$_x$(n) {where x is the partition size}
- DIVIDE the problem into *n/x* parts.
- SOLVE each part
- COMBINE the solutions for the final solution

Parts of the problem must not be interdependent.

Divide and conquer is excellent for parallelization

DYNAMIC PROGRAMMING:
- Applied to recursion is descent + memoization
- recursively can be no cycles in the DAG of the recursion, or it will get into an infinite loop
- Is fundamentally a brute force approach
- Good for computing min/max style answers
- A pre-compute pass can speed things up immensely preventing $\sum n + 1..n$ searches
- Can use significant memory
- Caching combined with LAZY

GREEDY PROGRAMMING
- Packing algorithms, like the parser compiler function packer

LAZY PROGRAMMING
- When the computation may not be needed
- When the problem cannot fit into memory it can be lazy loaded as needed

STREAMS
- A finite sequence of discrete elements of the same type processed in a linear way
- Can be infinite with two arrays, one being processed, the other being loaded
- Good for representing large data sets coming out of storage

DATA STRUCTURES

ARRAY
- Typed and RAM indexed they are extremely fast with O(1) read for any element
- Insert is very slow as the array elements have to be copied to make room for the element
- allows the use of fast algorithms like binary search

LIST
- single or double linked for traverse forward and traverse back, fast inserts
- can only efficiently access in a linear way random access is O(n)
- counting length is 0(n)
- double linking requires twice as much overhead

TREES
- good for storing hierarchal data
- natural fit for recursive algorithms
- good for indexes requires only O $\log_x(n)$ to find an element
- performance is maintained only when the tree is balanced, re-balancing on insert can be an expensive operation
- recursion is practical to the logarithmic complexity of traversal

STACK
- LIFO (Last in First out) push on the end, pop by removing from end
- Fast implementation in arrays
- Good for tracking advancing through a problem, and returning to previous steps

QUEUE
- FIFO (First in First out)
- Good for processing in chronological ordering
- Can be used to do a breadth traversal of a tree

HASHES:
- Can index with non-numeric types
- Fast read and write

- Can use compound keys in certain situations

Algorithm Diagram