Computer Science Concepts

* Algorithm Techniques
  + Brute Force
    - Go through ALL possible opportunities, without really skipping any possibilities and determining at each step along the way if the current possibility is a solution
      * Can either stop if a goal or look for all solutions
    - Most useful with a small state-space
    - Advantage(s):
      * Simplicity
      * Complete if a solution exists
    - Disadvantage(s):
      * Poor run time generally
      * Not optimal
    - **General approach**:
      * Generate first candidate
      * Get next candidate(s)
      * Check if current candidate is a solution
      * Output if solution is found
    - Examples of these algorithms:
      * Selection Sort
      * Insertion Sort
      * Bubble Sort
  + Divide and Conquer
    - Divide the problem into several sub-problems (usually smaller instances of the same problem) where we solve the sub-problems separately when they are much easier to solve and combine at the end to get the solution of the original problem
    - Best when the problem CAN be broken down into smaller parts, BUT the sub-problems are not evaluated many times (sub-problems aren’t very necessary to be reused else dynamic programming is better)
    - Advantages:
      * Does well with complicated problems
      * Efficient run-time and space usage
    - Disadvantages:
      * Recursion is slow
      * Simple problems could have easier and cleaner solutions
    - Examples of these algorithms:
      * Binary Search
      * Quick Sort
      * Merge Sort
  + Greedy Programming
    - Decision of algorithm is always based on what is the current best choice
    - Used for optimization problems where we have different values for choices that can be made
    - Advantages:
      * Simplicity (just go towards cheapest choice)
      * Easier to analyze run time and space usage
      * If solvable, then it is the best algorithm choice
    - Disadvantages:
      * Not necessarily optimal
      * Many issues of correctness to take into consideration
    - Examples of these algorithms:
      * Prim’s
      * Djikstra’s
      * Kruskal’s
  + Dynamic Programming
    - Splits the central problem into multiple sub-problems- and after solving each sub-problem will combine the solutions to get the final solution
    - Differs from Divide & Conquer because in here we reuse the solutions to the sub-problems multiple times
      * The sub-problems are **not** independent in Dynamic, unlike D&C
      * We will cache the results of the previous sub-solutions to the sub-problems, and when calculating towards the bigger goal, we will save time not having to recalculate the sub-problem
        + Memoization = having a table of all solved sub-problems to use towards getting the final solution (Top-Down)
        + Dynamic Programming = Analyze the problem and see the order sub-problems are solved, solving from the sub-problem TOWARDS the actual problem
    - Often used for optimization problems
* To-do
  + Data Structures
  + Work through various algorithms and problems utilizing the techniques above
* Recursion
  + Always requires at least one base case, but there can be more!
  + Remember parameters and a value equal to the recursion invocation can BOTH change
  + Without Loop:
    - Update parameters as required (like param – 1 for calculating Fibonacci)
    - Don’t think too hard when generating the solution
      * Function(A) = A + Function (B) = A + B + Function(C)
        + Couple of things

The plus is just a placeholder, you could have any operation- including multiplication, division, subtraction, and perhaps a connective for a stack

Think of it as ‘Function of A = A + The return value of B = A + B + The return value of C = A + B + C.’ This could be a path, a sum, multiplication, just about anything

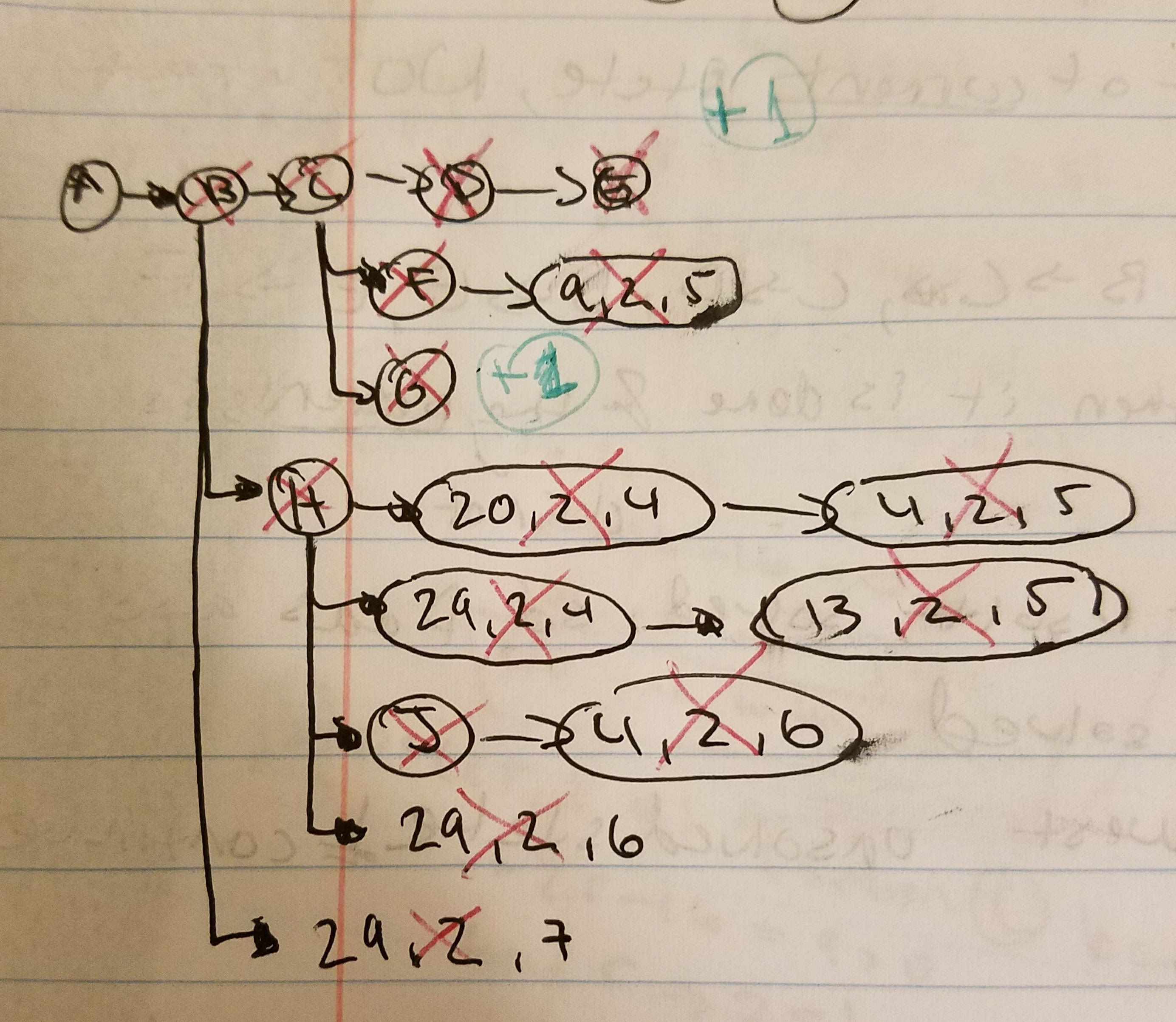
While it is helpful to think of it like this, remember that recursion utilizes a stack, and so in the case where Function(C) ends and is not a goal or a specified value- it is removed from the stack and we go to another child of B (like D)

This results in:

Function(A) = A + Function (B) = A + B + Function(D) where D is a solution and thus the path is A + B + D

* + - * + Form: The function’s own value (A), operator (+), next function call (Function B: B, +, Function C)

Until we return a value from the next function

* + With Loop:
    - Much trickier
    - The **current state** will have a child when it invokes the child in the loop with the recursive call
      * If this child returns a value, then BOTH child and parent will be taken out of the stack/considered solved (Ex E then D)
      * Else, this child becomes the new current state and will now be the parent and have its own child (Ex: F and C)
        + This creates a chain until the **newest unsolved parent** is solved by a child (F solved, so C is now newest again)
    - If you come back to a parent and must iterate, then you update the values based on the child that was solved before it
      * Ex: In the picture F was a child of C. F’s own child returned 0, so it was solved- thus resolving F. We went back up to C and generated child G, which had parameters based on the loop from F. The parameter I for F was 4, but for G because it came after F, I = 5.