Problem Solving: AFK[*]

Adrian German

June 24, 2019

Abstract

It's often said that education is what's left when you forget everything you learned in school. Here's a working document prepared for Min Lee, Rafa Daney and James Lewis in the Summer of 2019, as written then. Its purpose is to help you train your ability to monitor, plan and control your mental processes so that you can become a successful problem solver. In the process you will learn to become self-aware, to better monitor your own thinking and set appropriate goals, and (equally important) how to self-correct (usually in response to self-assessment).

1 Cracking the Coding Interview

Consider the following problems:

Check Permutation	Given two strings, write a method to decide if one is a permutation of the other.
No Duplicates	Given a string, write a program to determine if it has all unique characters[1].
Check Palindrome	Given a string, write a function to determine if it is a palindrome.
Almond Pier	Given a string, design an algorithm to determine if it is a permutation of a palindrome.

[*]Away from the keyboard.

[1]What if you cannot use additional data structures?

One Away	There are three types of edits that can be performed on strings: insert a character, remove a character, or replace a character. Given two strings, write a function to check if they are one edit (or zero edits) away.
Zero Matrix	Write an algorithm such that if an element in an $m \times n$ matrix is 0, its entire row and column are set to 0.
Remove Dups	Write code to remove duplicates from an unsorted linked list.
Return k-th to Last	Implement an algorithm to find the k-th to last element of a singly linked list.
Partition	Write code to partition a linked list around a value x, such that all nodes less than x come before all nodes greater than or equal to x.
Three in One	: Describe how you could use a single array to implement three stacks.

How do we go about solving problems like these[2]?

2 A Discipline of Problem-Solving

The first thing you need to do is to set an operational goal: a goal that can be checked. If you are working with a programming language you are already in the right neighborhood. Start by describing in that language how things will look when/if you're done[3]. In the process you will be forced to clarify any terms that you're not familiar with and choose a data representation. Once you have that under control you can reach back into your recollection of the examples you've seen, problems you've solved, or seen being solved, and try to adapt something similar to the problem at hand. Let's see how this works for the problems above.

2.1 The Miracle Question

The miracle question or "problem is gone" question is a method of questioning that a coach, therapist, or counselor can utilize to invite the client

[2]These are interview questions, in fact they're the simplest of their kind. They are taken from Gayle Laakmann McDowell's book "Cracking the Coding Interview".

[3]TDD resembles Solution-Based (Brief) Therapy.

to envision and describe in detail how the future will be different when the problem is no longer present. A traditional version of the miracle question would go like this:

> "I am going to ask you a rather strange question [pause]. The strange question is this: [pause] After we talk, you will go back to your work (home, school) and you will do whatever you need to do the rest of today, such as taking care of the children, cooking dinner, watching TV, giving the children a bath, and so on. It will be time to go to bed. Everybody in your household is quiet, and you are sleeping in peace. In the middle of the night, a miracle happens and the problem that prompted you to talk to me today is solved! But because this happens while you are sleeping, you have no way of knowing that there was an overnight miracle that solved the problem. [pause] So, when you wake up tomorrow morning, what might be the small change that will make you say to yourself, 'Wow, something must have happened—the problem is gone!' "?

So start by assuming each problem solved and write that in Java.

2.2 A Few More Problems

Before we go any further let's increase the sample size[4]:

Word Parity	How would you compute the parity of a very large number of 64-bit words?
Swap Bits	Design and implement code that takes as input a 64-bit integer and swaps the bits at indices i and j.
Reverse Bits	Write a program that takes a 64-bit unsigned integer and returns the 64-bit unsigned integer consisting of the bits of the input in reverse order.
Closest Integer	Define the *weight* of a nonnegative integer x to be the number of bits that are set to 1 in its binary representation. Write a program which

[4]These are from "Elements of Programming Interviews in Java" by Adnan Aziz, Tsung-Hsien Lee and Amit Prakash.

takes as input a nonnegative integer x and returns a number y which is not equal to x, but has the same weight as x and their difference $|y - x|$ is as small as possible. You can assume x is not 0, not all 1s.

Bitwise Product Write a program that multiplies two nonnegative numbers. The only operators you are allowed to use are: assignment, bitwise operators, equality checks and Boolean combinations thereof.

Bitwise Quotient Given two positive integers, compute their quotient, using only addition, subtraction and shifting.

So now ask the miracle question (and answer it) for each one of these.

2.3 The Categories

As you may have noticed, the second batch contains questions that are very similar (consistent, uniform) in the kind of knowledge they test: the category to which they belong could be entitled "Primitive Types". In this document we are going to survey a very large number of sample problems/questions. To make sure we build the right kind of expectations we enumerate here the categories (other than `1.` Primitive Types) in which we will group the questions: `2.` Arrays, `3.` Strings, `4.` Linked Lists, `5.` Stacks and Queues, `6.` Binary Trees, `7.` Heaps, `8.` Searching, `9.` Hash Tables, `10.` Sorting, `11.` Binary Search Trees (BSTs), `12.` Recursion, `13.` Dynamic Programming, `14.` Greedy Algorithms and Invariants, `15.` Graphs and `16.` Parallel (Concurrent) Programming. In addition, we will also have a section on `17.` Object-Oriented Design (i.e., Design Patterns).

2.4 A Sampling

This will give you an idea of the basic area we want to cover:

Arrays Write a program which takes as input a sorted array and updates it so that all duplicates have been removed and the remaining elements have been shifted left to fill the emptied indices. Return the number of valid elements. Many languages have library functions for performing this operation—you cannot use these functions. Please provide an $O(n)$ time and $O(1)$ space solution.

	Image rotation is a fundamental operation in computer graphics. Write a function that takes as input an $n \times n$ two dimensional array, and rotates the array by 90 degrees cloclwise. The time complexity of your solution should be $O(n^2)$ and the additional space complexity is $O(1)$.
Strings	Reverse all the words in one sentence.
Linked Lists	Delete a node from a singly linked list.
Stacks/Queues	Evaluate Reverse Polish Notation (RPN) Expressions.
Binary Trees	Compute the k-th node in an inorder traversal. How fast is your code?
Heaps	Write a program that takes as input a set of sorted sequences and computes the union of these sequences as a sorted sequence. If there are k sequences and n elements in total describe the time and space complexity of your solution.
Searching	Search a sorted array for an entry equal to its index.
Hash Tables	Find the nearest repeated entries in an array.
Sorting	Computer the intersection of two sorted arrays.
BSTs	Find the first key greater than a given value in a BST.
Recursion	Compute all mnemonics for a phone number.
	Implement a Sudoku solver.
Dynamic Programming	Count the number of ways to traverse a two dimensional array.
Greedy Algorithms	You are given a sequence of adjacent buildings (each has unit width and an integer height, like a histogram). These buildings form the skyline of a city. Compute the largest rectangle under the skyline.
Graphs	Model a maze as a graph and search it.
Parallel Programming	Implement caching for a multithreaded dictionary.
Domain Specific	Design an efficient algorithm that takes as input a set of text files and returns pairs of files which have substantial commonality.

Design Patterns Explain the difference between the template method pattern and the strategy pattern with an example. / Explain the observer pattern with an example. / Explain the difference between the singleton pattern and the flyweight pattern. Use concrete examples. / What is a difference between a class adapter and an object adapter. / Compare and contrast the main creational patterns: builder, static factory, factory method, and abstract factory.

For each problem ask the “miracle question”. What do you need[5]?

3 PEBKAC (SCNR)

Take a look at this example of how to approach the very first problem we mentioned (“Check Permutation”):

```
public class One {
  public static void main(String[] args) {
    String a = "how", b = "who", c = "wow";
    System.out.println(isPermutation(a, b)); // true
    System.out.println(isPermutation(a, c)); // false
  }
  public static boolean isPermutation(String a, String b) {
    // purpose statement: method returns true if one of its inputs (a) can
    // be obtained through a rearrangement of the characters in the other (b)
    boolean answer = false; // so we can compile (for now)
    // ...
    return answer;
  }
}
```

So at this point one needs to access some templates (if they exist) from past (hopefully successful) experience(s). But before you can do that one needs to come up with some concrete examples describing the desired behavior. For example one can argue along the following lines:

> “what” and “thaw” are identical modulo a permutation of their letters, but “what” and “hat” are not (because ‘w’ does not appear in both). Likewise “what” is not a permutation of “wrath” because ‘r’ appears in one but not the other. However “froth” is

[5]In each case. What is your data representation?

a permutation of “forth” and “silence” is a permutation of “license”. And so on (preferrably with additional examples).

Perhaps as you read the paragraph above you verified that our statements were in fact accurate. How did you check?

You can check letter by letter but that might not be easy if the strings get longer. For example, the phrases ”teaching and learning” and ”rain, the dancing angel” are an anagram of each other, that is, they have the exact same composition, if we exclude the comma and the spaces. But is it easy to check? (No, the straightforward way is a bit tedious, isn’t it?) So, instead, we can arrange the letters in each word in alphabetical order and report if the two orderings are identical or not.

Thus “silence” would be first converted to “ceeilns” and then “license” would be converted to (or sorted into) “ceeilns” (which is the same thing). As a result they’re “the same”. But “what” would be sorted into “ahtw” while “wrath” would be sorted into “ahrtw” and because the two orderings don’t coincide (one is longer than the other, for starters) the two strings are not “the same”. So this thinking relies on the “functional decomposition” template and it could result into the following refined goal/target:

```
public class One {
  public static void main(String[] args) {
    String a = "how", b = "who", c = "wow";
    System.out.println(isPermutation(a, b)); // true
    System.out.println(isPermutation(a, c)); // false
  }
  public static boolean isPermutation(String a, String b) {
    // purpose statement: method returns true if one of its inputs (a) can
    // be obtained through a rearrangement of the characters in the other (b)
    boolean answer;
    a = One.sort(a);
    b = One.sort(b);
    answer = a.equals(b);
    return answer;
  }
  public static String sort(String a) {
    String result = "";
    // this in Python can be achieved thusly: result = ''.join(sorted(a))
    // how do we do it in Java?
    // ...
    return result;
  }
}
```

So now we know what the end of the road (the target) looks like. We could even delegate the development reliably to a team member (if we happen to be the coordinator).

```
import java.util.Comparator;
import java.util.stream.Collectors;
import java.util.stream.Stream;

public class One {
  // ...
  public static String sort(String a) {
    String result = "";
    // this in Python can be achieved thusly:
    //   result = ''.join(sorted(a))           // how do we do it in Java?
    result = Stream.of(a.split("")).
      sorted(Comparator.comparingInt(o -> Character.toLowerCase(o.charAt(0)))).
      collect(Collectors.joining());
    // System.out.println(result); // [1]
    return result;
  }
}
```

Now the question is: would you have come up with the same exact solution? Keep the same basic steps but develop your own solution and share it with me. Let me therefore remind you the steps, as we learned them in C211.

3.1 The Design Recipe

Every time we need to design a function we go through these steps:

1. Choose data representation
2. Give some examples (of your data representation choice)
3. Name your function, write the signature (its contract)
4. Write purpose statement (for your function)
5. Give some examples (describing how your function will work)
6. Choose a template[6]

[6]Templates are constructed in a step by step fashion from the data definition. In the same spirit, I prefer to say we develop a data representation for the information that our program will process (with the implication we may have to backtrack and fix it). The result is a data definition.

7. Write the code (based on the template)
8. Add check-expect statements (tests)

The design recipe is a way of structuring (not eliminating) your creative process (as I am sure you know).

3.2 Isograms

We can start in DrJava this time:

```
import org.junit.Test;
import org.junit.Assert;

public class TwoTest
{
  @Test public void noIsogram() {
    Assert.assertFalse(Two.fun("banana"));
    Assert.assertFalse(Two.fun("apple"));
    Assert.assertFalse(Two.fun("whatever"));
  }
  @Test public void isogram() {
    Assert.assertTrue(Two.fun("a"));
    Assert.assertTrue(Two.fun("able"));
    Assert.assertTrue(Two.fun("octane"));
    Assert.assertTrue(Two.fun("vampire"));
    Assert.assertTrue(Two.fun("captions"));
  }
}
```

So we're using JUnit now and we just asked (and answered) "The Miracle Question". All we have to do (after we write the code) is: compile, then press the "Test" button. So let's solve the problem, see in what additional complications we get ourselves in:

```
public class Two {
  public static boolean fun(String a) {
    for (int i = 0; i < a.length(); i = i + 1) {
      if (Two.count(a.charAt(i), a) > 1) {
        return false;
      }
    }
    return true;
  }
}
```

Now we have a “wish list” of one: `count`.

Miracle question leads us to this battery of tests:

```
import org.junit.Test;
import org.junit.Assert;

public class TwoTest
{
  @Test public void countNone() {
    Assert.assertEquals(Two.count('a', "onion"), 0);
    Assert.assertEquals(Two.count('i', "team"), 0);
  }
  @Test public void countOne() {
    Assert.assertEquals(Two.count('a', "what"), 1);
    Assert.assertEquals(Two.count('i', "pneumatic"), 1);
  }
  @Test public void countMany() {
    Assert.assertEquals(Two.count('a', "whaat"), 2);
    Assert.assertEquals(Two.count('o', "onomatopoeia"), 4);
    Assert.assertEquals(Two.count('w', "Bow-wow!"), 3);
  }
  @Test public void noIsogram() {
    Assert.assertFalse(Two.fun("banana"));
    Assert.assertFalse(Two.fun("apple"));
    Assert.assertFalse(Two.fun("whatever"));
  }
  @Test public void isogram() {
    Assert.assertTrue(Two.fun("a"));
    Assert.assertTrue(Two.fun("an"));
    Assert.assertTrue(Two.fun("bye"));
    Assert.assertTrue(Two.fun("able"));
    Assert.assertTrue(Two.fun("handy"));
    Assert.assertTrue(Two.fun("octane"));
    Assert.assertTrue(Two.fun("vampire"));
    Assert.assertTrue(Two.fun("captions"));
    Assert.assertTrue(Two.fun("magnitude"));
    Assert.assertTrue(Two.fun("binoculars"));
    Assert.assertTrue(Two.fun("geophysical"));
    Assert.assertTrue(Two.fun("productively"));
    Assert.assertTrue(Two.fun("unpredictably"));
    Assert.assertTrue(Two.fun("hydromagnetics"));
    Assert.assertTrue(Two.fun("abcdefghijklmnop"));
  }
}
```

So now let’s write a solution to this by fleshing out `count`.

We obviously could have[7] written the purpose statement etc. instead we simply show you the final code:

```
public class Two {
  public static boolean fun(String a) {
    for (int i = 0; i < a.length(); i = i + 1) {
      if (Two.count(a.charAt(i), a) > 1) {
        return false;
      }
    }
    return true;
  }
  public static int count(char c, String word) {
    int howMany = 0;
    for (int i = 0; i < word.length(); i++) {
      if (c == word.charAt(i)) {
        howMany += 1;
      }
    }
    return howMany;
  }
}
```

We need to add this approach to our toolbox; and we move on.

[7]But did not, and left it as an exercise for you