Programming is a process where one applies particular computations to achieve a result. Since computation has become cheep, the problems programmers solve adapted to become increasingly dynamic. This dynamic environment which allows for fast iteration times (sometimes live) has resulted in the ability to change both the problems we try solve, and their solutions quickly. Solving a single problem is not of concern, instead, we look for solutions to evolve alongside our problems.

1 Introduction

What do I mean by programming as a search problem?

I want to make my self clear here (Insert explanation).

For the time being let us ignore temporal aspect of programming concerning it's evolution. Instead for the time being, I will focus on how to solve a single problem, by searching for the appropriate solution. I will do this by first showcasing how a solution is made out of abstract parts that form a general shape, which a programmer refines into a detailed sculpture, which is of-course an analogy.

A search over the set of problems that you need to solve and a set of solutions to said problems. To reiterate: we search over a set of problems because usually the types of problems programmers face are dynamic as requirements change, so are the solutions that are required to solve said problems.

If you like, you can think of constraints like removing clay from a clay blob. The more you remove, the more detail the shape will have, eventually turning into a sculpture. If you do not remove enough, the sculpture will lack uniqueness to separate itself from just another clay blob.

This is a perfect analogy

capturing problem well and solving it and only it. Less specific constraints make a amorphic shape that captures more of the space by being less detailed. This is what allows generality to take place.

This space is constrained to all computational problems for these are the types of problems we are able to solve using our computers. Constraints are the natural way of limiting the search scope. Fuzzy search on your computer works by you giving it some key words, which function as a way to constrain the possible outputs giving ones which seem relevant. The more characters or words you give it, the more specific the output you are expecting to get as you constrain the solution space further.

The reason for why I propose to look at programming as a search problem is because it is. Programming is concerned with searching for tools to help solve the problem, certain structures, and other ways of dealing with complexity that arises or creating it when needed.

Hard constraints - these are the specific constraints Relaxed constraints - these are the general constraints that work in a relaxed manner

The model for looking at programming I am suggesting here is that of a search space and constraining using hard constraints along with relaxed constraints. This, as I will show, explains all effects programming advice bad and good has if we start with good assumptions and not invent thing out of thing air and try to say they are good because "it works for me".

NOTE: Is this required? As Allen Turing has shown, a simple Turing machine is all you need to create any sort of expected computation as it is a general purpose machine. The computers we use today are way more complex than said machine. And they are also real where as a Turing machine is commonly used as a mathematical object, a real computer is used to produce our common day computation.

Complexity - Simplicity Hard - Easy

Subdividing problems as a means to solve bigger problems. Subdivision breaks a big problem into smaller ones that you can digest individually. Sometimes the subdivision look alike which turns into a recursive algorithm that can be represented in a iterative fashion.

The bitter lesson of machine learning. Where optimizing for a certain hardware every time is a worse solution than a generic one that gives speeds up to all utilizing Moors law to full effect. This is because of the lack of wasted time in hard-coding everything, and instead using search to help find a better solution in a dynamic way. This has some upfront cost, but it can be justified if a significant speed up is found resulting from that.

People like to over complicate things by applying their domain specific knowledge to a problem even when not needed. Complex solutions to problems where a simpler search over the space although "expensive" in terms of compute, is more general and leverages gains in computation more. Brute force methods might be slow today but faster tomorrow if Moors Law is to be of any value.

From Rich Sutton's, The Bitter Lesson:

> In computer chess, the methods that defeated the world champion, Kasparov, in 1997, were based on massive, deep search. At the time, this was looked upon with dismay by the majority of computer-chess researchers who had pursued methods that leveraged human understanding of the special structure of chess. When a simpler, search-based approach with special hardware and software proved vastly more effective, these human-knowledge-based chess researchers were not good losers. They said that "brute force" search may have won this time, but it was not a general strategy, and anyway it was not how people played chess. These researchers wanted methods based on human input to win and were disappointed when they did not.

Cyber security people use the value in understanding a problem to their advantage. Understanding how the systems they want to crack work, allows them to insert values in ways that alter the expected behavior from the program, making it produce unexpected results which if inserted correctly might give a cyber cracker things he should no be able to have.

2 Randomness in exploration and generality

We can learn much from how machine learning algorithms have been benefiting from randomness in order to not overfit into a specific dataset and remain more general.

3 General things to consider

To solve problems in the domain of computation we use transformations on data. The simple transformations your computer can execute are called instructions. The model most people have of a computer is that of a Von Neuman machine. Essentially a CPU - Central Processing Unit that executes instructions, and a memory unit usually RAM - Random Access Memory that stores the result. For the time being the exact nuances in the implementation of modern hardware will not get discussed. [1]

My reasoning is simple: we are not directly exposed to said nuances in general programming, they stay hidden until we have a reason to care like in places where more performance is needed.

Because of this reality of how our machines work, I argue we understand programs best when written in the way it is executed, simple linear programs. When people avoid this reality, interesting programs arise, and ultimately make understanding of such programs a truly hard task.

For those who are unsure why we have to understand our programs well, consider what we try doing while programming. We, as stated in the first paragraph, search for solutions to problems in the domain of computation.

4 Early return (this is a continuation from linearity of code)

With the years I have converged on some ways that I like to write my code. One such way is to early return if possible at the beginning of the function. Consider the following function:

```
void compare(int *a, int len_a, int *b, int len_b) {
  // constraints on valid input
  if (0 < len_a && 0 < len_b) {
    if (len_a == len_b) {
      // logic on valid input
      while (*a == *b && (*a != 0 && *b != 0)) {
        a++; b++;
      }
      if (*a == *b) {
        return *a - *b;
      }
    }
    else {
      // handling of invalid input
      return len_a > len_b ? 1 : -1;
    }
  }
```

[1] Today we have many Out of order machines that execute many instructions in parallel so long as they don't depend on on another. The first machine to do so goes back as far as 1964. This out of order execution in time has given birth to branch prediction. There are also SIMD instructions and so on. Memory today has cache hierarchies with different policies for handling cache eviction, and more changes are to be seen as manufacturers try to make their chips fast.

```
  else {
    // handling of invalid input
    return 0;
  }
  return 0;
}
```

We have some conditions that check the validity of input. In the way it is written, the checking and handling of invalid input are separated visually while logic is put in between.

```
void compare(int *a, int len_a, int *b, int len_b) {
  // constraints on valid input + it's handling
  if (len_a < 0 && len_b < 0) return 0;
  if (len_a != len_b) return len_a > len_b ? 1 : -1;

  // logic on valid input
  while (*a == *b && (*a != 0 && *b != 0)) {
    a++; b++;
  }
  if (*a == *b) {
    return *a - *b;
  }
  return 0;
}
```

This is a rewrite of the first example. Both achieve the same but the second at least to me is significantly easier to see what is happening. The validity conditions are visually close to their handling, logic is then completely separated and put below. In this case we can see that the visual jumps you would have to do to understand it is lower and so the code is read in a more linear fashion than a none-linear one. This allows assumptions from above to propagate below, so constraints written above are assumed for code below and we no longer care for it's handling, shifting the focus to just the logic allowing us to understand it better by seeing it clearly as it is not between other not related code.