# *COMP 340 - Lecture Notes - 02 - Skiena Chapter 1*

*Spring 2014*

In these notes we discuss the content of chapter 1 of *The Algorithm Design Manual*.

## *The Design Process*

Let's recall the process we're putting to work here:

1. Identify the real-world problem, give it a *name*, and identify the problem's *inputs and outputs*. Choose an abstract model[1] for it's input and output. Check models against concrete instances of the problem.
2. Formulate a high-level strategy for solving the model of the problem.
3. Write a complete pseudo-code algorithm.
4. Evaluate algorithm correctness by finding counter-examples to prove it incorrect or proving correctness mathematically[2]. Use concrete examples to develop proofs or to motivate improvements to algorithm for the sake of correctness.
5. Evaluate algorithm efficiency under the RAM computing model[3] using asymptotic[4] worst-case[5] analysis. Use concrete example to motivate improvements for the sake of efficiency.
6. Generate examples of implementations of the algorithm on a real-world computing system. Evaluate the algorithm for ease of implementation. Evaluate the implementations for correctness and efficiency[6].

[1] combinatorial, recursive, etc.

[2] probably with induction

[3] or a suitably chosen model of computation

[4] big Oh

[5] average-case analysis is a field of research in and of itself

[6] This is the field of Experimental Algorithmics

## *Well-specified problems and Models*

The study of algorithms might, at first glance, seem to be about solutions to problems. It turns out that you really cannot study solutions without also systematically studying the problems themselves. Why is that so? There are two big reasons we need to consider:

1. The structure and shape of the problem often lends itself to well known models with well-known algorithms and implementations. Thus, we can avoid reinventing some algorithmic and programmatic wheels.
2. Well-specified problems have a well-defined criteria for correctness. Correctness for poorly-specified problems is often fuzzy at best.[7]

[7] Garbage in, garbage out.

Honestly, this all boils down to the fact that you can't expect to develop a reasonable solution to a problem when you don't understand the problem to begin with. The real difficulty is that we like to think we understand the problem when, in fact, we really don't. So, how can one make sure the really understand a problem? You guessed it: *concrete examples of instances of the problem*! Easier said than done though. What exactly constitutes an instance of a problem? How do we come up with them? How many do we need? Are some instances "better" than others? Let's start at the beginning: what is a problem and how can we define them in a way that is conducive to examples.

A well-specified problems has three parts:

1. a name
2. a well-defined set of possible inputs
3. well-defined required properties of the output

Names should be short, descriptive, and specific. In giving your problem a name it become more tangible. Good names go a long way.

Problem inputs are the information that vary from one instance of the problem to the next. We should describe the details of how we are modeling them and find some kind of Goldilocks zone where we're not super restrictive but also not too general. When faced with a difficult problem, it is often a good idea to restrict the input in order to simplify the problem[8]. So, think of it this way. If your definition of the inputs doesn't also imply ways to restrict or generalize the inputs and thereby modify the problem, then it may be the case that your definition isn't refined enough. *Once you think you've settled on a good definition, play around with it and define more restrictive and more generalized versions of you input as a check on how well-formed your definition is.*

[8] pg 13

Skiena goes the Jeopardy route with statements about algorithm output and frames them in the form of questions. From this perspective output descriptors become the questions answered by the algorithm[9]. A good check you might make is to look at the problem name and output descriptor together. If they seem clearly related, then you at least have a good name for your problem. Skiena points out two pitfalls for output descriptors[10].

[9] This is why they're called problem descriptions in Part II of the text

[10] pg 13

1. Ill-defined questions
2. Compound goal

The first pitfall often boils down to being vague and saying things like "best" without being clear what best means. So, try to leave zero

ambiguity in your questions. The extreme on this is to pose your question using formal mathematics rather than natural English. A nice medium you might play with if you're unsure if your question is ill-defined would be to attempt to capture some of the key properties of your question with Math. Assuming you're math is correct, it lets you get away with much less than English. The second pitfall would seem to be a case of doing too much and is the same as writing function in our programs that do too many things. If the goal of your algorithm really has multiple distinct goals, then you're probably dealing with multiple problems! Don't be afraid to step back and see if you can't rethink your problem as several problems working together. In this case, you either simplify your problem or set out to design multiple algorithms.

There's more to problem specification and example generation though. Problems exist in the real-world, algorithms exist in model-world. This means the understanding we gain in the early parts of design is textured by these layers. The real goals of the first phase of the design process are:

- Understand the problem as it exists in the real-world[11]
- Understand the model of the problem [12]
- Understand the relationship[13] between the two.

[11] Domain Knowledge

[12] Math and Computing Knowledge

[13] mapping

The real messy part, and the reason for iterative refinement through exploring concrete examples, is the fact that real problems are often vague and poorly-specified[14] at first, and so our first crack at a model might not work out. We could fail to capture crucial information or not capture enough. In the end, you'll often find that by exploring the real-world to model mapping, you learn more about he real-world, which in turn leads to better models, which in turn leads to better mapping, which in turn leads to a better understanding of the problem, which in turn leads to...[15]

[14] at least in ways that allow verifiable correctness

[15] you get it

*Models*

When you're designing programs, then you model real-world information with program-world data structures. When you're designing algorithms you need to model with something at least a structured but not tied to a specific programming language or programming paradigm[16]. That something is mathematics. Skiena offers the following list:

[16] I'm looking at you OOP

1. Permutations
2. Subset

3. Trees
4. Graphs
5. Points
6. Polygons
7. Strings

Most of these should be familiar to you, but we'll review a little or a lot as needed. You not only need to know what these things are, but ways in which you might restrict an instance of one of these structures by places conditions on its elements[17].

[17] Thankfully, half your textbook is full of examples of this as inputs/outputs of algorithms

There's one big pattern underlying the structure of all of these objects and that is *recursion*. You should not be surprised by this. You've been recursively defining structures since semester one. These recursive decompositions can sometimes lead to efficient algorithms. They always lead to a place to start and always keep you safe from having nothing to try. Dust off your recursive thinking, you're going to need it.

## *An Idea*

So you think you have a good handle on your problem and have defined it in sufficient detail to start taking a crack at designing the algorithm. Now what? Now you come up with an idea, a plan for how to achieve your solution, a *strategy*. Remember this stage is comparable to jotting down a template? Strategies are about outlines of how you'll get form input to output. The good news is that there are a lot of general strategies for us to work with, we just need to learn them. *If our goal is to be algorithm designers, then this is why we study existing algorithms. We want to learn about the strategies they use.*[18].

[18] Read this again. Drill it into your head. Always be on the lookout for algorithmic strategies

Skeina doesn't seem to give explicit statements of strategy when presenting algorithms. When we design algorithms, we will always write write a short statement of the strategy we plan to use in our algorithm. I can imagine quiz and exam questions focused solely on the efficacy of different strategies for a particular problem. The point is, *we will be explicit and regular in attaching strategies to our algorithms*. As we learn some standard strategies, it is not sufficient to just drop their names as declarations of strategy for our algorithm. We need to contextualize them within our problem. We'll cover strategies encountered in the first chapter later. For now, just store away in your long term memory that no algorithm should be designed without first clearly identifying the overall strategy that the algorithm will carry out.

## *Psuedocode*

When it comes to choosing a language for specifying algorithms, there are really lots of choices. Many of them just don't do what we need them to. English[19] is too vague and leaves too much room for interpretation. Programming languages and formal mathematics are too strict and too fine-grained and force us to delve into details that distract from the the core logic of the problem. As Skiena points out however, we'll ultimately use all three in the specification of our algorithms. English is best for describing the problem and our overall algorithmic strategy. Mathematics is perfectly well suited for laying out the properties of our inputs and outputs. Programming languages are a must if we want to have anything more than a theoretical discussion about the ease of implementing our algorithms. But, when it comes to specifying the algorithm itself, *psuedocode* is usually the hands down winner.

[19] any natural langauge

There is one problem with psuedocode. It's not really a thing, but an idea. There's no set standard for psuedocode, just generally agreed upon conventions that might vary from one community to another. So, your goal is to mimic the style and conventions of our textbook. Just remember that the goal is clarity. If you don't truly know what you're trying to express with your psuedocode[20], then you're probably not going to write acceptable psuedocode. One way of checking yourself is to translate your pseudocode into something more precise and something less precise, say, Code[21] and English. If you can't imagine how to say it in one or both of those languages, rethink what you're doing and see if you can say it in a way that's more clear.

[20] or English, or Code, or Mathematics

[21] or Math

## *Correctness*

An algorithm is not correct until it has been mathematically proven to be correct. This is not always an easy task, but the big thing we have going for us is recursion. If you've done your job right and identified a well-defined input, then that input is likely to have some underlying recursive structure. *Where there is recursion, there is mathematical induction*[22]. Before you set out to do an inductive proof of the correctness of your algorithm, you must first explore some concrete instances of your algorithm for two reasons: the examples will help you find the pattern for the induction or *they will help you find an instance where the algorithm does not work*. So, your best bet is usually to try and prove the algorithm incorrect by counter-example, and wait to attempt a proof of correctness only when you're super-extra convinced its correct[23]

[22] we'll cover that part of chapter 1 in the next set of notes

[23] Haven't looked at any examples but you're convinced it's correct anyway? Be ready to be wrong.

### *Incorrectness*

Section 1.3.3 discusses the search for counter-examples to your algorithm's claim of correctness. *Go read it again.*[24]. Skiena's advice on page 14 should be committed to memory. However, if you really and truly understand the conditions placed on a proper counter-examples, then his advice seems like the logical consequences. So, let's repeat those conditions here:

[24] Then read it 10 more times

1. *Verifiability* You must show exactly what your algorithm does for your counter-example input **and**[25] show that the result can be bettered by a different input.
2. *Simplicity* Listen. If the problem instance is complicated and large, you probably can't get your head around it enough to demonstrate verifiability so just don't try. The best counter-examples don't just show that the algorithm fails, they clearly illustrate what about their logic caused them to fail. Nothing is clear with giant inputs.

[25] this part is pretty optimization focused

### *Examples*

Skiena gives us two examples and one war story[26]in this chapter.

[26] those things are gold, read and re-read them

1. Robot Tour (aka Traveling Salesmen)
2. Move Scheduling
3. Psychic Modeling

We should note that all three of these problems are Optimization problems[27]. Not every problem is an optimization problem so let's be on the look out for something different later. Before we break these down let's be sure we know what the big lessons were:

[27] Minimize/Maximize something

- Algorithms are always correct. Heuristics are sometimes correct.
- Correctness is hard. Don't make assumptions about correctness.
- Good modeling is key.

Great. So the only one I want to talk about is the first. Absent a proof of correctness, how do you know something is a heuristic or an algorithm? You don't. Heuristics are really that happy medium between correct and crap. The difference between algorithm and heuristic is clear, but the difference between heuristic and crap might be less so. How often do you need to be right[28] for something to get promoted to heuristic? I'll leave you to think that our for now. Let's look at the design process at work and break down the develop of the algorithms for these problems step-by-step.

[28] or how close to right do you need to be

*Robot Tour Optimization*

We begin, as all algorithms should, with defining the problem.

> *Problem:* Robot Tour Optimization *Input:* A set $S$ of $n$ points in the plane. *Problem Description:* What is the shortest cycle tour that visits each point in the set S?

[29]

[29] pg 5.

Now the author just jumps right to coming "up with an algorithm to solve this problem", but that's easy to say and less easy to do when you're new to this game. What we would do next[30] is play around with some concrete instances of RTO.

[30] or concurrently with the problem defining stage

Where would you start? What's the first kind of example you'd come up with? Me? I'm lazy. I'd find all the trivial examples I could. It's clear that for $n < 3$ this problem is very trivial[31]. So now what? Well, write out a few trivial examples and their solutions. Now, let's build on top of that. Add to those and see what we see. The point here is start easy, not hard[32]. Make absolutely certain that what you think is easy is in fact easy. Explore any possible nuance of easy. All that easily gotten success will keep you moving when you get to the not so easy stuff and will hopefully make that stuff a little easier.

[31] I see recursive base cases

[32] Go small or go home

Ok. We have examples. Now, it's time to develop and state a strategy. Skiena just pulled strategies out of the ether. And didn't state them formally. Let's go ahead and do that for him[33]. We begin with:

[33] We'll always do this

> *Strategy:* Move to the nearest neighboring point. Thus if we're at point $p_i$ at step $i$. Then, point $p_{i+1}$ is the point closest to $p_i$. The starting location, $p_0$, is chosen at random.

Several things to note about this declaration of strategy. We start with something plain and English, and then provide a bit more formalism by effectively defining an iterative process that corresponds to our strategy. The first sentence is great, but lacks sufficient specificity. Remember, the HtDP process required that we write a template, which outlined the overall structure of the code. So our statements of strategy need to at least provide some kind of high-level, semi-formalized outline for what our pseudocode needs to do.

So, how does he know that nearest-neighbor is a heuristic? The answer is hindsight, of course. The problem is that he announces it as a heuristic before proving that it's not an algorithm. It's probably best if we imagine starting with the assumption that our strategy is crap. We can then show some cases[34] where it's correct. From there, we need to explore some complex cases and test their correctness. In some cases, we discover that cases exist where the our algorithmic process leads an incorrect solution. At this point we're looking at a heuristic at best[35]. We might have to accept the fact that our strategy

[34] probably trivial

[35] if we deem it to be "correct enough"

failed. No biggie. At least we know something that doesn't work and can analyze it for where exactly it goes wrong![36].

[36] Something wrong is better than nothing

Back to the process. We defined the problem and established a strategy for our algorithm. All the while, we should have been working concrete instances of the problem to test our intuition. Now it's psuedocoding time. The trick to psuedocode is to finding the right amount of stuff to *not say*. Let's see what kind of logic Skiena[37] left out.

[37] pg 6

1. how to select $p_0$
2. a specific loop control variable
3. variable type declarations
4. how to compute and select $p_i$ given $p_{i+1}$
5. what it means to visit $p_i$

This might seem like a lot of crucial information, and it is if you're setting out to implement this algorithm[38]. So let's recall the goal of this step of the process. *To clearly highlight and convey the general idea laid out in our strategy*. In many respects, our strategy statement was quite clear; it clearly laid out the iterative process carried out by the loop. So what do we gain from Skiena's pseudocode? I'll argue the following:

[38] programmer's problem. not algorithm designers

1. clear sense of sequencing.
2. A more imperative[39] presentation for the idea
3. A few steps closer to code

[39] as opposed to declarative

In the end, we get more specificity without also losing sight of the overall strategy. This is a tricky tightrope to walk, but we'll rely on our authors and each other to find that sweet spot between too much and not enough detail.

Skiena works one more strategy[40] before landing on an actual algorithm. The algorithm is dead simple: use BRUTE FORCE and try all possible solutions. Computers are really excellent at this. Don't every forget this strategy because it almost always works. The problem is it's usually painfully inefficient and impractical. So what. This makes it a terrible place to end the process, but a wonderful place to begin. They tell you that at least one algorithm exists and they give you an upper bound on your algorithm; they give you a target to beat[41]

[40] closest-pair. pg 7

[41] always look for a way to get a win when solving problems

I personally like to make sure I know how to brute-force a problem before I attempt to be clever. It's a good check on your understanding of the problem model. Here's why. Notice that Skiena's algorithm clearly states the number of possible problems and doesn't

just say, "try all possible solutions." The later lacks specificity, the former clearly imposes order on the problem space, order that can be leveraged in to programs[42]. In fact, there are well understood ways of ordering, and therefore stepping through, a set of permutations.

So, I might go about solving ROBOT OPTIMIZATION differently. The first strategy I'd try would be [43]. Seeing that this algorithm works but is entirely intractable, I'd look to improve *the strategy*. What does brute-force do wrong? Well, one way to look at brute force is that the choice of what to do next is made based off nothing other than structures in the model[44]. So why not try to dictate the choice based off some knowledge of the problem. This is, in my mind, what leads to things like nearest-neighbor and closest-pair[45]. Models have very well define structures absent the problem to which you're applying the model; when in doubt, use them as the basis for your strategy.

[42] if you can count it, you can problem encode it as numbers

[43] brute-force

[44] why hello basic HtDP recipe...

[45] this is all classic AI as well