On your marks...

Have you ever heard the term "coding" before? It has several meanings, actually. You may have heard it on a TV show set in a hospital in which a patient is said to be "coding." Technically, it means a Code Blue and that a patient is in cardiac or respiratory arrest. Coding can also mean the act of computer programming or writing computer code. This is the kind of coding we hear about in the context of computer science. It is the art of instructing a computer to do something for us. And lately it seems as if everyone is talking about it. In education, it is almost as if coding is the new literacy: the thing that everyone should know how to do in order to be prepared for life. But coding is a very small part of computer science. Learning how to code is fine; however, it is much more powerful and fulfilling when learned alongside all of the other parts that make up computer science.

What is computer science?

This is exactly the question that we will attempt to answer in this curriculum, and at its conclusion you should be able to answer that very question. But it's not a simple one. Computer science is a large topic that is composed of a great many things, of which one small part is coding. We will try to explore as many of these things as we can. Fundamentally, though, computer science is about solving problems. The end goal of this curriculum is to understand what computer science is while simultaneously becoming better at solving problems (which, by the way, is something that we will need to do for the rest of our lives).

Coding is easy

That's what it seems like to a lot of people. In fact, this is a fallacy; however, the idea that it's easy (or that it should be) appears to be widespread. We see this on TV, in advertisements that try to get young people interested in computing, on the Internet, in social media, and so on. The problem with this is that people who start learning to code naturally assume that it will be easy for them as well. And then they hit hurdles.

Coding is a bit like learning to play a sport. Let's pick on racquetball since it's a great game that many people love to play. It takes a while at first to play well. It's not easy to hit a small blue ball that is moving across a small court very fast. By the way, the court has four walls and a ceiling – a lot of surfaces for that ball to bounce off of. After playing for some time, we become pretty good. We win many games and only lose some. We win enough to feel good about our performance. But then we hit a plateau. That plateau sometimes lasts for *years* (see Figure 1). It's not fun to play a game so much and not become any better for a long time. But if we play long enough, we do become better. If we play long enough. And if we don't quit. It's just how long it takes.

Failure

Let's face it, quitting is easy. We often assume that if we fail at things, it must mean that we're not good at them. And for some reason, we also assume that we'll *never* be any good at them. So we quit, especially if something is supposed to be easy since everyone seems to think so. And if coding is not

Failure should be our teacher, not our undertaker. Failure is delay, not defeat. It is a temporary detour, not a dead end. Failure is something we can avoid only by saying nothing, doing nothing, and being nothing. – Denis Waitley

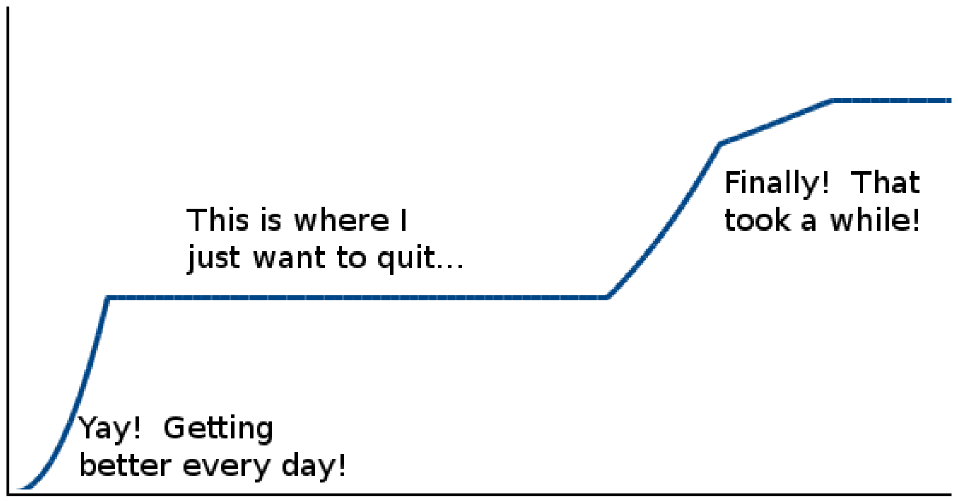
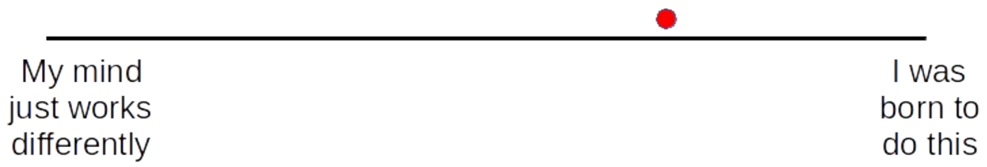


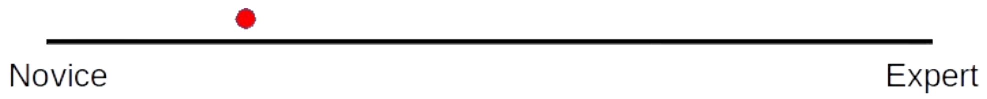
Figure 1: Racquetball learning curve. Horizontal axis is time spent playing; vertical axis is skill.

easy for us, then coding must not be for us. We must not be *built* to be a coder, and most likely our mind just *works* differently than those who are good at coding. Let's be honest, failure sucks.

But the truth is that most programming doesn't require a special brain (or something that we're somehow *born* to do). There is no such thing as "I'll never be good at coding" or "I was born to code!" We can't simply plot our *ingrained* ability on a scale like this:



In reality, it's more like this:



This scale is more realistic and provides a more reasonable meaning to quantifying one's expertise in coding. We all begin as novices with no experience. By learning, and through practice, we become better and shift to the right on this scale. It is true, however, that we are all quite different: we have different abilities, skills, learning strategies, experiences, prejudices, developed behaviors, and so on. We may not all shift to the right at the same velocity. Some of us may make progress and accelerate quickly while others may need *more practice* to get to the same point. Some of us will reach a plateau that will last a very long time, while others may blast through it in much less time. And some of us may encounter more problems along the way than others. Welcome to the real world. Remember this (attributed to Theodore Isaac Rubin): The problem is not that there are problems. The problem is expecting otherwise and thinking that having problems is a problem.

Failure is no fun. It can be awful. But living so cautiously that you never fail is worse. – J.K. Rowling

Although to the right side of the scale above is expert, there is no such thing as an expert coder. There is no final level in coding that, once reached, means that it has somehow been mastered. Becoming better at coding is something that we will work on continuously from the time that we start coding to the time that we stop coding. It's almost as if someone is constantly moving the right end. Frustrating, isn't it?

Men vs. women
There is this ridiculous perception that men have what it takes to do well in computer science and that women somehow do not. Let's be clear: research does *not* substantiate this. In fact, it is diametrically in opposition to this. It is true that men and women are different: we look different, we age differently, we have somewhat different lifespans, we have a different center of mass which affects how we walk, and so on. But it is not true that we are somehow born to have what it takes (or not) in terms of our potential to excel at coding.

However, it is much more frustrating and messy than those who do it well let on. In other words, the plateau is often at play. And when we hit the plateau, we may feel that we are not making any significant progress for quite some time. And that is just discouraging. But the thing to do is to keep trucking along, to keep coding and producing work. That's just how long it takes.

Success
Very early on we may have a pretty good idea of what good code looks like. We may be able to take a look at someone else's work and determine if and how well it solves some given problem. We may be able to comment on its beauty and efficiency. We may be able to call it a great achievement. But creating such beautiful code on our own may not come easily at first. The trick is to keep coding and producing work. And eventually, we make it through the plateau.

Some of the most successful people in the world were failures for a large majority of their lives. Yet we consider them geniuses. We have this unfortunate knack of ignoring the hard times they went through, their failures, and just looking at their successes. We also see them as they are now. Rarely do we look at how long it took them to get to this point, or how much junk they put out before finally being successful. You would not believe how much utterly lousy code so-called experts created before finally creating something worth sharing and being proud of. You would not believe how much utterly lousy

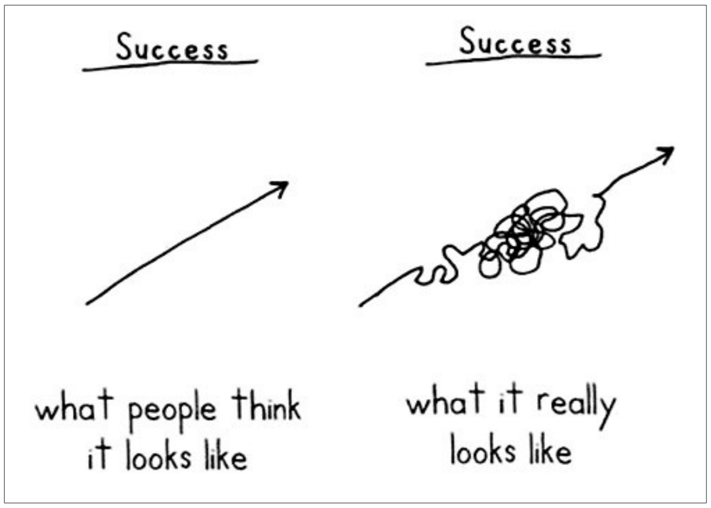


Figure 2: There are dozens of sources for this image, but none identify the original author.

Success is a journey, not a destination. – Ben Sweetland

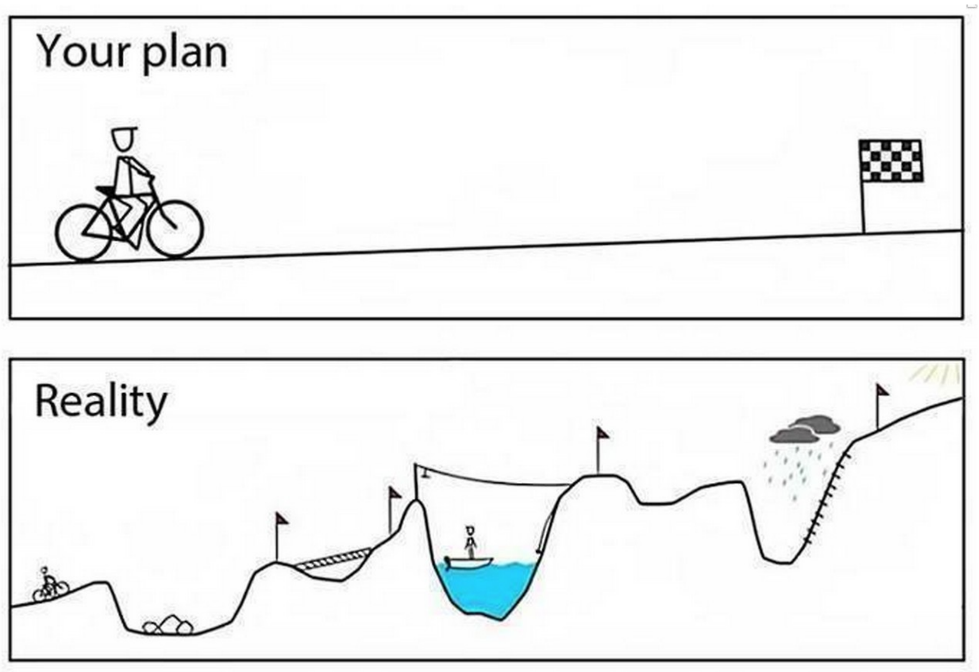


Figure 3: There are dozens of sources for this image, but none identify the original author.

code most computer science professors wrote before finally producing something that they were proud of sharing with others without worrying if they would be laughed at.

It's not always obvious how much work it takes to become good at something. The perception is that success is achieved by following some straightforward path. The reality is that it's just not so simple. In our fast-paced world, we want immediate results and solutions right away. And when the solutions don't come quickly, we become discouraged. It is therefore hard to accept the idea of failure. But to be successful, we must understand that failure is a part of the process. It is inevitably on the path to success. The key is to learn from our mistakes (and the mistakes of others) in order to become better and to grow. And this helps lead to more successes than failures in the long run.

Trying vs doing

You may have heard the phrase, "So long as you try, that's OK." In truth, it is not enough to simply try. Competition is fierce! A grade of B is not acceptable in the real world. Try it, and you will eventually be replaced by someone who cares about producing the best work...and does. This kind of person loves to solve problems. In fact, this kind of person views problems as puzzles, and puzzles are fun!

People who are successful do not just try; they *do*. That is, they achieve results (i.e., solutions that work...efficiently). People who are successful are not lazy. Lazy people do not prosper nor do they succeed at very much. It is through failure (and more importantly, by subsequently trying again), that we become better and eventually succeed. The really cool part is that we often discover useful tactics along the way that help lead us to more successes over time. It is true, however, that it may take a good number of failures and subsequent retries before success is achieved.

In the end, we must care about developing a skill set that will provide us a broad range of tactics and knowledge that, in turn, will propel us to be good at solving problems that we have never seen before. To be clear, it is not enough to just learn. To ultimately be successful, we must *desire* to learn.

Surrounding garbage
To code is one thing. It takes skill, experience, and a whole lot of practice. But it takes a bit more. There is the added tedium of setting up our environment so that we can code efficiently. This means setting up an operating system, compilers, integrated development environments, the command line (terminal), and so on. Often, no one wants to help us set this stuff up because it's usually frustrating and takes time. There is a lot of surrounding stuff to learn to do in order to just get ready to begin to code. So we need to learn other things first in order to be able to even begin coding.

A lot of coding skill is about developing a knack for asking the right questions on Google. It's about knowing which results to filter out and which to take a closer look at. It's about knowing which code that we find by clicking around is best to use (and, of course, give credit to – we obviously don't want to plagiarize). We must become good at discovering patterns.

Spectacular achievement is always preceded by unspectacular preparation. – Robert H. Schuller

Feeling stupid
When it comes to coding, we should also get used to feeling lost and stupid. In fact, the anxiety of feeling lost and stupid is not something that we learn to conquer. It's something that we learn to live with and manage. The most common state for a programmer is a sense of inadequacy. There is a limitless amount of stuff to learn. We need to *constantly* learn new tools, new techniques, new principles, new hardware, and new software. We better develop a habit of liking to learn. In the end, it helps to be mentally prepared for feeling stupid.

Learning computer science
At first, our coding skills are pretty low. We take on a lot of projects that are fairly easy and designed to get us comfortable with coding and solving problems. As our skills grow, we begin to feel good. If the problems remain simple, however, we inevitably become bored. We aren't really getting any better by continuing to code simple stuff. It's like playing racquetball. If we limit our opponents to those we can always beat, we won't become better at playing anytime soon. But as the challenges increase, we often begin to feel anxious. We feel inadequate and that our skills aren't good enough for more difficult problems. But if we keep solving more challenging problems and creating work, however, our skills will grow again. The trick is to manage our skills and challenges so that we continuously remain interested (see Figure 4). We want to tackle problems that are hard enough to test and grow our skills, but not easy enough to make us bored.

Kate Ray (technical co-founder of scroll kit, a visual web page creation tool; she now works for WordPress) has an insightful way of visualizing the learning process. She breaks it down into several steps:

1. Follow a tutorial, even if you don't understand everything that you're doing;
2. Rebuild the thing that you just made without the tutorial (at least, without using it too much);
3. Try to build something simple that you want to build that's related to the thing that you just built;

Unix for stability. Macs for productivity. Windows for solitaire.

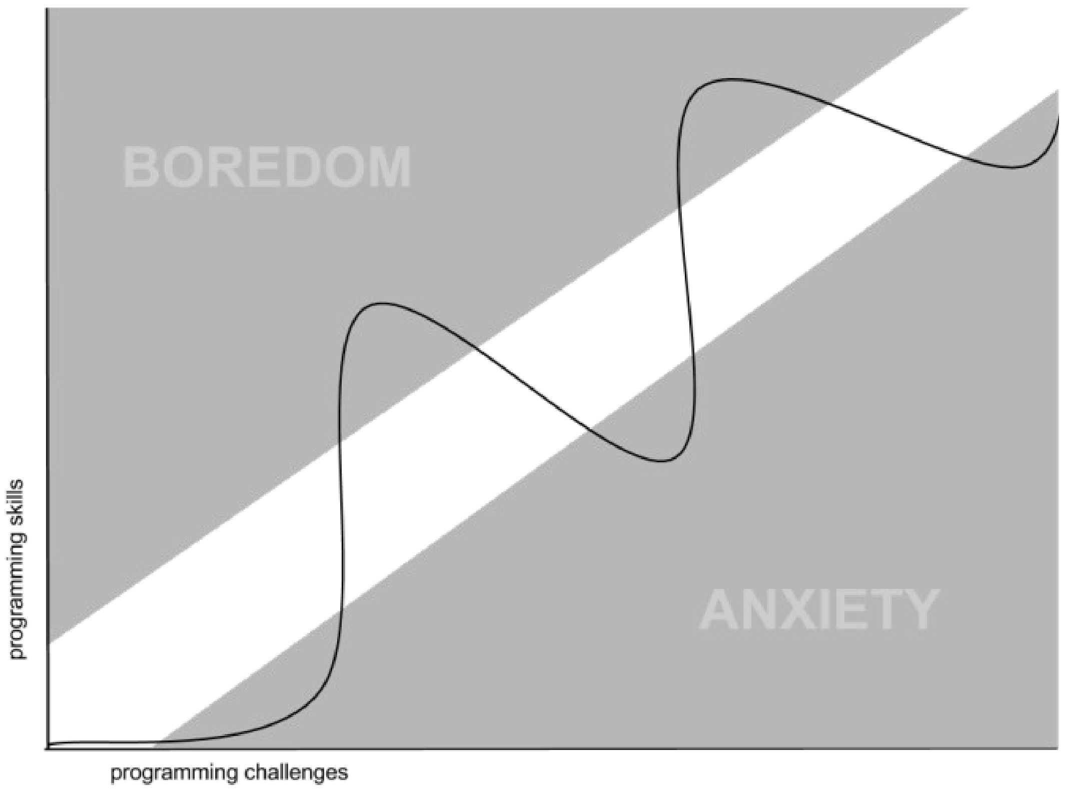


Figure 4: From an article by Kate Ray in TechCrunch.

4. Find a new tutorial related to your new thing and use it to build this new thing;
5. Rebuild the new thing yourself without the tutorial;
6. Start a new project; and
7. Repeat 1 to 6 over and over again.

These steps seem to infer that we're doing this on our own. To a large degree, this is true. There is only so much of coding that we can learn from others. It is largely a skill that is grown by locking ourselves up in our rooms and creating more work. And we need to do this a lot on our own if we really want to *accelerate* the learning curve. It can be quite frustrating. But we need to learn to grind through the frustration. It helps to like to tinker and break things. It helps to be OK with not understanding everything, but to have the desire to understand everything. It helps to aspire to intelligence rather than belittling it. It helps to be OK with not always seeing our progress.

Most people do not know how to learn, lack basic organizational skills, have absolutely no clue how to manage their time efficiently, and are grossly inefficient at processing and storing information. Truly, we all share some or all of these characteristics at one point or another in our lives, but very few people actually figure out how to regularly do these seemingly common things well. To develop these skills requires practice, often on our own. It's not particularly glamorous.

Ignorance is a state. Willful ignorance is a choice. It's the definition of stupidity. – Dan Garcia and Jean Gourd

Most people can't think, most of the remainder won't think, the small fraction who do think mostly can't do it very well. The extremely tiny fraction who think regularly, accurately, creatively, and without self-delusion – in the long run, these are the only people who count. – Robert Heinlein

Get set...

Many teachers (and even people who don't teach but who decide on policies that teachers will follow) have a tendency to mix teaching and testing; that is, to teach the test. After all, this makes it so much easier on the part of the teacher. It also manages to promote more successes than failures. You may think that sitting in class and learning this stuff should, in the end, make you good at it. You may think that you should be provided everything that you need in class, particularly in lectures. In truth, formal teaching (e.g., through lectures) should get you through the large stuff. In computer science, we call this large stuff the foundation and the pillars of computer science. From there, it's up to you to get through the little stuff that takes *repetition* and *practice*. And this is especially true of coding and many things in computer science. You can only pass the test if you get the large stuff *and* the little stuff.

But that's not all. Being good at coding is just not possible without simultaneously being good at solving problems. It means to have developed a skill set that has propelled us to grow into good problem solvers. And being good at solving problems means that we are able to make judgments about problems and about the solutions to those problems. After all, we want our final solution to a problem to be the best one. We therefore need to be able to think critically.

Go!

There are some things that we should always strive for. For example, we should try to learn about everything. Yes, this sounds a bit broad. More specifically, we should try to foster a desire to understand how things work and not just how to use them. We should try to be informed, both about things that we are interested in (like coding, maybe) and things that we know help to increase our awareness of the world and our usefulness in and contribution to society (like politics, maybe). Certainly this takes an immense amount of work on our part; but in the end, it is absolutely worth it. To help with this, the following is a little bag of tricks that I have developed over the years. It helps me solve problems, particularly those that relate to computing:

A little bag of tricks

0. *What is the problem*? If you don't have a clear understanding of the problem, you will never be able to solve it. Period.
1. *Simplicity is, well, simple*. Give yourself a simpler problem. Try to find the trivial or base case. It greatly reduces the difficulty.
2. *Doodling is fun*! Draw pictures and diagrams to help you solve the problem. Few can think in math and formulas. Most of us are visual.
3. *Split it up*. Solving parts of the problem and then putting it all together often helps if that's possible. This is exactly what we do when we design complex algorithms.
4. *Work from both ends*. If you know what the outcome or solution is supposed to look like, then use that to help solve the problem. It could reveal ideas and shortcuts.
5. *Similarity, my dear Watson*. Many problems are similar in nature. Try to see if a problem is the same thing as another you've previously solved or know something about. There's nothing wrong with reusing old material if it is relevant.
6. *Does it make sense*? Often, students accept a first hack at an answer, forget to ask themselves this important question, and do not notice how nonsensical their answer might be. Ask yourself

The world doesn't need more people with good grades. The world needs people who see the really tough problems as puzzles and have the tenacity and creative capacity to solve them. – Gever Tulley

if an answer makes sense before you commit to it. This will avoid turning in assignments that contain unedited (and often unread) cut-and-pasted material from the Web with phrases like, "Click here for a detailed explanation." And yes, this has actually happened. This is probably the most ignored trick and the most annoying to a prof who is grading your assignment.

7. *If time permits, optimize*. It's never a bad thing to eliminate redundancy and optimize your solution to the problem.
8. *Don't give up*! Seriously, don't give up. If you need help, discussing general ideas with fellow classmates is always a good thing. However, you should refrain from discussing things that are too specific (e.g., code). Furthermore, teachers should always be glad to help you **if you show them that you have reasonably thought about the problem first**.
9. *Confidence is crucial*. A lack of it can actually be quite deadly.

Thanks to Matthew Michalewicz (a researcher in puzzle-based learning), here are a few rules that can help us solve problems:

1. Be sure you that understand the problem and all the basic terms and expressions used to define it;
2. Don't rely on your intuition too much. Solid calculations are far more reliable; and
3. Solid calculations and reasoning are more meaningful when you build a model of the problem by defining its variables, constraints, and objectives.

Epilogue

You may be wondering about what makes people good at coding or, more broadly, at computer science. Here are the kinds of characteristics that make *good* computer scientists:

1. *Curiosity*. Being inquisitive about the world helps us to understand and discover problems. Curiosity is helped along if we have broad and varied interests, a desire to want to know how the world works, and a desire to constantly want to learn new things.
2. *Creativity*. Thinking *sideways* (a term that means to think "outside the box") helps us to create new problem solving approaches and useful solutions. Our creativity grows as we get better at thinking critically using careful evaluation and judgment. It is helped along if we learn to use past experiences to solve current problems, to be socially competent in order to argue for our ideas, to work well in teams so that resources and ideas are shared, and to be good at (and, more importantly, to love) solving problems.
3. *Focus*. Having perseverance and tenacity helps us work through long-lasting problems and allows us to deal with failure more easily. Focus is easier if we become good at recognizing patterns and at developing efficient solutions.
4. *Attention to detail*. In a way, this is related to focus and helps us to maintain rigor and find errors in our logic. We should assume that we aren't perfect (which we aren't), and that our solutions won't be perfect.
5. *Communication*. If life is about discovering solutions to hard problems, then we will undoubtedly have to convince others that our solutions work. We must therefore become good at communicating and defending our ideas both verbally and in writing.

So the above characterize *good* computer scientists. Well, here are the kinds of characteristics that make *great* computer scientists:

0. The kinds of characteristics that make good computer scientists.
1. *Organization*. Be organized and maintain detailed records.

2. *Time management*. Know how to manage your time efficiently.
3. *Efficiency*. Strive for efficiency when developing solutions.
4. *Improvement*. Desire to constantly improve – to become better at solving problems.
5. *The general case*. Design solutions that work for all cases instead of just one specific case. Aim for robust solutions instead of throwaway prototypes.
6. *Skepticism*. Question everything. It helps to develop and refine curiosity and creativity.
7. *Honesty*. Be honest with yourself and with others.
8. *Open mindedness*. You never know when someone else's ideas will trigger something that sparks a new (potentially better) idea.
9. *Continuously learn*. Understand that computer science is a lifelong learning process. There is no final level that denotes mastery of the discipline.
10. *Produce*. Understand that computer science is a production-oriented discipline. Trying hard is meaningless without eventually producing something that works. That takes a lot of repetition and practice.
11. *Confidence*. Although we constantly shift on the "novice-to-expert" scale, having confidence in our abilities wherever we sit on that scale goes a long way.

A final statement that is attributed to Larry F. Hodges (a fellow academic) and slightly paraphrased: a good thing to remember is that education in computer science is as much about learning how to think critically about issues and how to solve problems as it is about how to create and use technology. The technology is continually changing, but the problem-solving skills learned can serve a person throughout their life.

Bucket of quotes

I've heard a lot of quotes over the years. Here are some of my favorites:

- Despite the cost of living, have you noticed how it remains so popular?
- Nothing is foolproof to a sufficiently talented fool.
- Light travels faster than sound. This is why some people appear bright until you hear them speak.
- Everyone has a photographic memory. Some just don't have film.
- Everyone makes mistakes. The trick is to make mistakes when nobody is looking.
- If you can't convince them, confuse them.
- Support bacteria – they're the only culture that some people have.
- It may be that your sole purpose in life is simply to serve as a bad example.
- Growing old is mandatory, growing up is optional.
- Make it idiot-proof, and someone will make a better idiot.
- Those who cannot change their minds cannot change anything. – George Bernard Shaw
- Your reputation is made by others. Your character is made by you.
- Tell the truth, there's less to remember.
- If you thought before that science was certain – well, that is just an error on your part. – Richard Feynman
- Don't go around saying the world owes you a living. The world owes you nothing. It was here first. – Mark Twain
- Success is just like being pregnant. Everybody congratulates you, but nobody knows how many times you were screwed.
- Life is like a jar of jalapeños. What you do today might burn your ass tomorrow.

- A successful life is one that is lived through understanding and pursuing one’s own path, not chasing after the dreams of others. – Chin Ning Chu
- The difference between genius and stupidity is that genius has its limits. – Albert Einstein
- I have approximate answers and possible beliefs and different degrees of certainty about different things, but I'm not absolutely sure about anything. – Richard Feynman
- I would rather have questions that can't be answered than answers that can't be questioned. – Richard Feynman (attributed)
- Confidence is silent. Insecurities are loud.
- There are 10 types of people: those who know binary, those who don't, and those who weren't expecting it in base 3.
- If it is important to you, you'll find a way. If not, you'll find an excuse.
- I never lose. Either I win or I learn.
- If we don't believe in freedom of expression for people we despise, we don't believe in it at all. – Noam Chomsky
- True ignorance is not the absence of knowledge, but the refusal to acquire it. – Karl Popper
- If serving is below you, leadership is beyond you.
- You want to know the difference between a master and a beginner? The master has failed more times than the beginner has ever tried.
- You are not entitled to your opinion. You are entitled to your informed opinion. No one is entitled to be ignorant. – Harlan Ellison
- Stop Global Whining. – Jean Gourd