Intro to Computer Science

Lesson 07 Notes

Past, Present, and Future of Computing

Past Present And Future

Welcome to unit 7. This unit will start with a brief summary of the course, recapping the major themes we've seen. Then the rest of the unit will be all fun field trips and interviews. We'll start by looking at the past of computing by visiting the computer history museum. Then we'll look at the present and how you can contribute by visiting SLAC National Accelerator Laboratory, and talking to the folks at Mozilla and Benetech. Finally we'll get a glimpse into the future of computer by visiting UC Berkeley and looking at what people there are doing research on. We'll start by summarizing the course, and I want to highlight the three main themes that I think are most important in computing.

Themes

I think there are three main themes that really pervade the course that we've seen over and over again, but I haven't mentioned them explicitly until now. These are themes that aren't just pervasive throughout this course but are pervasive throughout all computer science. The first one is abstraction. Abstraction is all about hiding details. The point of abstraction is to make it so you can have one thing that is used in many different ways, and you can use it without necessarily understanding all the details behind it. We've seen lots and lots of examples of abstraction in this class. The most important one is procedural abstraction. We can write one procedure, use it to do lots of different things depending on what the inputs are, and when we use that procedure we don't need to know exactly what sequence of instructions will execute. What we know is what the procedure does. The second main theme is universality. We talked about this early in the class about how computers are universal machines. I want to show you how much better I've gotten at drawing. If you thought my drawing of a toaster in Unit 1 was not recognizable, I hope you can see how much better I can do now.

This is a toaster, and a toaster is very different from a computer. A computer is a universal machine. We've seen lots of examples of how a computer is universal-- that we only need a few simple

constructs to be able to define every possible program. We saw that if we had a way to make decisions-- and we saw we have that with if in Python; we saw that if we have a way to keep track of things-- and in Python we can do that using variables and data; and if we have a way to keep going-- and we first saw that using while--with a while loop we can make something keep going as long as we want. But in fact, we don't even need while. We could do that with procedures, and that's what we saw with recursive definitions in Unit 6.

Once we have those 3 things, we can write any computer program. So we have much more power with a computer than we do with a toaster. The final major theme I want to summarize as our third main theme is one we've already mentioned here. It's recursive definitions. We introduced those explicitly in Unit 6, but you saw them many times before that. The first time you saw them was actually in Unit 1. We saw these rules that said we can make an expression by taking another expression, using the plus operator, and combining it with another expression. That was not enough by itself for a recursive definition. We also needed a base case. We saw a base case, like we can have an expression that's a number. So the big idea here is that with very simple rules we can define infinitely many things and very complex things by using rules to define things in terms of simpler versions of themselves.

Overview

These are our 3 big themes in the class: abstraction, universality, and recursive definitions. What I want to do now is look at various things that we've covered throughout the course and see how they fit into these 3 main themes. In Unit 1 we introduced the concept of variables. We can use a name to refer to some value. So that's definitely a type of abstraction. It allowed us to use the name x to refer to different things. It's helpful for universality, but by itself it doesn't really provide us that. And it's not really a recursive definition, although we can define variables in terms of other variables. In Unit 2 we introduced procedures. A procedure is definitely a kind of abstraction. By making the parameter to a procedure, we can have code that we write once that does many different things, and we can hide all those details when we use the procedure. Procedures also give us a form of universality.

We can use the same procedure to do many different things, and we can define procedures to do everything, as we've seen. We didn't really understand that yet in Unit 2, though, but it is enough to define every possible computer program. We certainly use procedures to make recursive definitions, and we've certainly defined recursive procedures--not until Unit 6, though. And this gives us a way to define a procedure to break a problem down by seeing it in terms of a smaller version of itself. The main new thing in Unit 3 was lists. This is a kind of data abstraction. You can use a list without knowing the details of how Python implements a list. It also can be a recursive definition. We can have lists that have elements that are other lists, so it certainly is useful for recursive definitions. And it's universal in the sense that we can put any element we want in a list. So in Unit 4 the main things we did were finish the search index by providing a way to produce an index of data. This certainly involved a lot of abstraction. We found a way to represent the data from the web pages in an index

and respond to queries. The other thing we talked about in Unit 4 was how networks work. Networks are all about abstraction. We want to find ways to make requests where we're thinking about what we're requesting as a web page. All the details of how we actually do that are hidden, and unless we need to worry about that, it's much better to think about that abstractly as this is sending a request for a web page. In Unit 5 the main new idea we introduced was how to measure cost. That's a kind of abstraction. We want to measure cost in terms of understanding how the cost scales with the size of the input rather than the details of the cost. It's also related to universality. Understanding the cost of a procedure depends on having a fairly universal model of computing.

We want to understand how much a given algorithm costs without depending on the details of how our particular computer works. And then we introduced the hash table. This was a kind of data abstraction. It was universal in the sense that it could contain any kind of element as its values. We didn't define it recursively. We could certainly have hash tables containing other hash tables, and we have dictionaries of dictionaries, so this also certainly has aspects of recursive definitions in place. And finally, in Unit 6 the main big idea was recursive procedures. Obviously, that fits in to our theme of recursive definitions. It also relates to universality. We showed how to make code that runs forever or keeps on going as long as it needs to without using a while loop, so that gives some support for the idea that all we really need are procedures, if, and a way to keep track of things to be able to define every computer program. We saw that any program that we could write as a recursive procedure we could also write as a while loop. So if you've followed everything that we've done in this class, you've really learned an amazing amount. And as you learn more and more about computing, you'll see these 3 themes of abstraction, universality, and recursive definitions all over the place. But you've already seen them many, many times just in the units of this course.

Quiz: Computer Science

We're only going to have 1 quiz in this unit, and the quiz is something that should sound like a fairly simple question, although it's actually quite difficult to answer. The question is, what kind of thing is computer science? Your choices are engineering, science, and liberal art. Check as many as you want.

[x] Engineering
[x] Science
[x] Liberal Art

Computer Science

This is definitely a subjective question, and it's kind of strange that a discipline like computer science is so poorly defined that if you ask many computer scientists what it is you'll get very different answers. But my answer to this is it is engineering, it is a science, and it's also a liberal art, that all 3

are true, and I'll explain why next. This is the way I tend to think about engineering is it's building big, complex structures that solve physical problems like getting cars across a bay of water. With that narrow view of building physical things, most of what we do in computing is not about building physical things, it's about building virtual things, it's about solving problems. A broader view of engineering views it as design under constraint. When you're building a bridge, the constraints are the laws of physics. You need the bridge to stand up, but you want to limit the cost of the bridge, the amount of material that you use, you want it to not sway too much in the wind. These are all the kinds of constraints that you have to think about in designing a bridge. When we build computer programs, we don't usually have to worry too much about the laws of physics. We've seen that they do affect us-- that the speed it takes for light to travel affects how fast our programs will run-- but for the most part when we're designing programs, this is not the main thing that we're thinking about. The main constraint that we have to worry about is the limitations of human minds. What we can do is limited by our creativity, but it's also limited by how much we can keep in mind at one time. The main tool that we have to solve this is abstraction. The point of abstraction is to allow us to limit what we have to keep in our mind to be able to solve problems by hiding those details and thinking about things at the right level and designing programs in a way that we can have a procedure that solves the problem that we want, and then we can forget about the details while we move on to the next part. That's all about using abstraction to overcome the limits of human minds in terms of how much we can keep in mind and understand at one time. So from that perspective, computer science is definitely about engineering. We certainly want to build things. We're not building things like bridges, and we're doing that under constraints, but the main constraints are not from the laws of physics-- because we're not building physical things--but from the limits of our own minds. We tend to think about science as understanding nature through observation, and our favorite example of this is Sir Isaac Newton. If you can't recognize this, this is obviously Newton because he's sitting under an apple tree and the probably untrue story of him understanding gravity because of an apple falling on his head. In computer science we're mostly not focused on understanding nature, at least not in a very direct way. We're thinking about abstract things. We're writing programs to solve problems, but they're dealing with abstract representations of those problems. Certainly computing is used all over the place to build better models of the universe and to test those models, but that's a little different from saying that computer science itself is a science. But there is a lot of computing in nature. Certainly physical processes involve computation. The other big example today is really in biology. If you think about DNA as programming organisms, it's a very complex way of programming, but it's also very computational. And if you think about biology, biology is really all about computation today. So certainly if we want to understand nature better, both how human minds work--that's largely about computation-- and how DNA produces biological organisms and how DNA evolves over time, that's really about understanding computation in nature. We didn't get to this too much in this class, but this question about universality, questions about what can be computed and how fast it can be computed are really fundamental questions about science. They are questions about our universe. They are abstract questions in the sense that we can form them as mathematical, precise questions, but they are also really questions about what are the limits and what are the possibilities in our

universe. The final correct answer was liberal art. In some sense this is arguably the easiest one to defend, but it's probably the one that was most surprising to many of you. Understanding why computer science is a liberal art depends on actually understanding what the liberal arts are. Here's a picture of the medieval view of the liberal arts. There's grammar, rhetoric, dialect, which we think of as logic today. Those are the three that have to do with language, and then there are four that have to do with number. The four that have to do with number are arithmetic, geometry, music, which is number in time, and astronomy, which is number in time and space. So how do each of those relate to computing? We've certainly seen a lot with grammars. We've seen rules like this. Rhetoric is about using language to communicate between people, especially to persuade people. We've actually seen rhetoric a little bit. This is a little less clear, but we've seen it in terms of network protocols. What protocols are are ways of computers communicating with each other and making sure that they can understand each other. So arguably, that's a form of rhetoric. Logic is the art of thinking in the medieval view. We've certainly used logic all over the place in this class. All of our decision constructs are logical constructs, so we've certainly used a lot of logic. So those are the three traditional liberal arts, known as the trivium, focused on language. And then we have the quadrivium, which are the four focused on numbers. Arithmetic. We've certainly seen lots of arithmetic in this class, and we can do lots of arithmetic in Python. Geometry. We haven't used too much directly in this class. There certainly is lots of use of geometry in computing. The two that I'll mention most-- In computer graphics that's all about understanding the geometry of light well enough to be able to simulate images and to be able to draw things on the screen. The other place where geometry comes up a lot is in networking, and we saw that a little bit in Unit 4. If we want to design a network, we care about the topology of the network-- how things are connected--and that depends on understanding geometry. But we didn't get to either of those things too much in this class. Music, which is about number in time. We haven't really seen how computing relates to music very much in this class, but certainly recursion occurs all over the place in music. If you don't believe me, I would encourage you to read this book by Douglas Hofstadter called Godel, Escher, Bach. It's all about how the logic of computing relates to things in both the art of Escher and the music of Bach. And the final traditional liberal art was astronomy. We haven't seen that directly in this class either, but certainly almost all astronomy today depends on computing.

Past Of Computing

So we've seen science, engineering, and five of the seven traditional liberal arts. That's pretty good. We should go on some field trips to celebrate. Let's start by visiting the Computer History Museum in Mountain View.

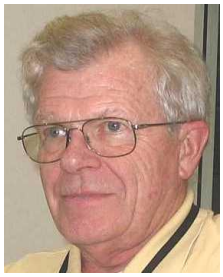
Computer History Museum

Hi, I'm Alex Bochannek, curator at the Computer History Museum in California. Welcome to Revolution: The First 2,000 Years of Computing, an exhibition that covers the history of computing from the abacus to the smart phone and everything in between.. The Computer History Museum was started in the 1970s in Boston, has been in California since the '90s, and since 2011 this new permanent exhibition has been open to the public with 23,000 square feet, 19 galleries, and over 1100 artifacts.

Babbage Engine

Folks, you're looking at the Babbage engine. It does what Charles Babbage envisioned it would do-- produce mathematical tables without any human intervention all the way over to the print shop. It produces a stereotype try, which then goes to a printer to print a book of logarithmic tables for example. This was envisioned by Charles Babbage 1849 but never built in Charles Babbage's lifetime.

First Hard Drive



My name is Dave Bennet. I'm a member of the team that restored this machine--the world's first hard disk drive. It was announced in 1956 it was the first product of IBM's laboratory in San Jose, California. This is a big machine with a very small capacity. However, in 1956 it was very important. This machine had 50 disks 24 inches in diameter. It stores a total of 5 million 6-bit characters, 3.75 megabytes in today's world. Any disk drive that doesn't have at least gigabytes in it is obsolete by modern standards. It was actually designed to run a request from the Air Force for storage capacity for their 50,000 spare parts inventory. It was designed to hold 50,000 character records with access to any records in less than a second.

Search Before Computers

How did we search before we had computers?

People with clay tablets, which have writing on them, would sometimes have cross-references to other clay tablets. That's kind of the origin of Bing. Telegraphy--you could send in a query, like to a search engine, as a telegram and get it back by post. There were a lot of people that were thinking about how to use technology to search. The search engine is basically where you're looking through every word in every document that might be up on the web that's accessible to find a match. It's really just the same as if you went into a library and you spent, in this case, it would take days to sit there and go through every page of book, looking for all the occurrences of--I don't know--the word "aardvark." Obviously, that's something that's really painful to do in the real world, but powerful computers can make that so fast that it actually becomes a reasonable way to search for information.

Search On The Web

Our students have all the knowledge they need to build a search engine, but what are they missing?

Well, a few thousand of these. We're standing in front of the Google server rack. This is a web server, but it's using Ethernet to connect these different boards, which are actually separate computers that they've put together in a rack. These are just generic PCs, because you can run a search engine at home on one computer, but it's not going to have a very big index.

How many are we talking about--10s?

MW: Many, many, many thousand. Google was not a particularly well-known one it its early years. Most of these search engines--today we think of them as being incredibly wealthy companies. At the time they were mostly losing money. They had venture capitalists put money in but they basically didn't make enough money from selling ads to pay their bills. Search engines actually go back before the web. On the Internet there were several search engines. The closest to what we have today might be WAIS, done in the late '80s, and that let you search different servers over the Internet. Then we have, for instance, the CERN Virtual Library, which was done by Tim Berners-Lee, the man who invented the Web, but that wasn't really like a modern search engine. It was more like a library cart catalog where you have categories. Early Yahoo was like that as well. The one that got the most recognition was Alta Vista, and this was kind of like the Google of its day in the late '90s. One that was a little bit different was Ask Jeeves, which was a search engine, but it was also meant to be a natural language query. You could say, "What's the capital of Spain," and it would give you the answer. Google borrowed an idea from a little search engine called GoTo.com, which was to not just have ads but to sell the search terms themselves. So like when you go to Google now, and you see the "sponsored results," those are actual terms related to what you're searching for, but they're also ads. That just took off. People really responded to that. That actually brought the entire search

industry from on its way down to being one of the most profitable parts of web commerce.

Present Of Computing

I hope everyone enjoyed the visit to the Computer History Museum. Maybe one day, there'll be something in the museum built by a CS101 student. If you're ever in the area, it's a really cool place to visit, and you can also drop by our offices in Palo Alto, only a short distance from there. What we're going to do next is learn about the present of computing. We're going to visit SLAC National Accelerator Laboratory, and see how they're using computing to advance science today.

Slac And Big Data

DE: We're here at SLAC National Accelerator Lab, and we're going to see how they use computing to understand the mysteries of the universe.

SG: We're standing in the klystron gallery, formerly the longest building in the world.

RM: You're here at SLAC National Accelerator Laboratory. This is a 50-year-old laboratory, as all the flags on the lampposts around the lab are telling you. It was founded to build a 2-mile-long linear accelerator. SLAC is an accelerator laboratory still. Its main science is based on accelerating particles and creating new states of matter or exploring the nature of matter with the accelerated particles. This always has generated a lot of data, a lot of information. It's very data-intensive experimental science. From the earliest days of SLAC computing to analyze data has been a major part of the activity here. You really can only study the cosmos by studying it in a computer. You get one chance to look at it, but to understand how it evolved into the state it is now, you have to do all this in the computer. There are massive computations going on for that sort of simulation, massive computations in catalysis and material science and massive data analysis going on here as well. The particular particle physics experiment that I am involved in right now has some 300 petabytes of disk space-- some 300,000 terabytes, some 300 million gigabytes of disk space around the world to do this analysis. Of course, we are far from understanding everything about the universe, but this is probably one of the most data-intensive activity in science today. The raw data rate coming out of the ATLAS detector that I'm involved in is about a petabyte a second. That's 1 million gigabytes a second. You can't store that with any budget known to man, so most of it is inspected on the fly and reduced to a much smaller, but still large, storable amount of data. Right now we are sifting through these many,

many petabytes of data to look for signals of the Higgs boson, as no doubt people have heard in the news. There are tantalizing hints that I'm not holding my breath about at all right now, but this is the way we do it. You need to have those vast amounts of data just to pick out the things that will really revolutionize physics in there, and you need to understand all of it in detail, because what you're looking for is something slightly unusual compared with everything else. If you don't understand everything else perfectly then you don't understand anything.

MS: We're looking at one of the racks that contains the ATLAS proof buster at SLAC. ATLAS is an experimental Large Hadron Collider in Geneva, Switzerland, that collides protons, fundamental building blocks of nature, traveling at very, very, very close to the speed of light with trillions of times the energy that they have at room temperature. You get many and many of these collisions happening at once and this enormous machine that reads out trillions of data channels. At the end of the day, you have this enormous amount of data--petabytes of data-- that you have to analyse looking for very rare, very particular signatures inside of that. If I want to look for a rare signature--something that had a lot of energy and a lot of really strange particles at once-- there are trillions and trillions of these events stored on this machine. To look for them in any reasonable amount of time, I have to do many searches at once. I have to use all the cores on the computers-- the hundreds of cores on the machine all running at full-speed at the same time-- to have any hope of doing it in any reasonable amount of time.

RM: This isn't the sort of thing that search engines currently do. They're looking for text strings and indexing all the text strings that they find in some way like this. What we have is very, very structured. We know the structure of these data. We know exactly how to go to anything that we want to get to in these data, because the way in which everything is linked together is very well understood. Things will go wrong all the time. You cannot assume you won't lose data from the disk. You send it by network from one computer center to another. You cannot assume it arrives undamaged. You cannot assume your computers don't die in the middle of calculations. Everything can go wrong, so the computing we do for the LHC has many layers of error correction and retry. Some of the basic failure rates are quite high, but by the time everything has been fairly automatically retried and errors have been corrected, we get high throughput and a high success rate.

Mozilla

For our next stop, I'm really happy to be here at Mozilla where they make the Firefox web browser. One of the really critical things about Firefox is it is open-source.

Open Source

DE: Can you tell us more about what does it mean for software to be open source?

AD: For software to be opensource it means a couple of things. First, it means that the actual underlying source code to the piece of software is freely available, that people can acquire that from a website or off of a DVD or in some other form and see the underlying source code that makes up that program. But that's actually insufficient, because opensource also requires that not only can people see it but they can do what they want with it. They can take that piece of software, make modifications to it, develop their own products based on top of that. There are different licenses that open source software operates under that have different requirements and responsibilities for those who take that source code and make modifications to it. At Mozilla with the Firefox browser, our license is called the Mozilla Public License, and it says you're free to take the source code to Firefox and go build your own browser with it. You can do whatever you want to that source code and you can turn it into Walmart-Fox, if you're Walmart and wanted to. There's no requirement that you pay us for that, that you have any involvement with Mozilla. But if you make changes that improve the software that you got from Mozilla, you have to return those changes. If you take a feature and you make it faster or you add a new feature, making the browser more capable, you're free to ship that and under your own brand for whatever your own business model is, but you have to return those changes. This allows Mozilla to take the work that you've done, reincorporate it into Firefox and ship a new version that's as good as the version you shipped. Different licenses behave in different ways. Our license is pretty generous in that it says you can take our product and combine it with other products, and you don't have to disclose any of your proprietary information. You only have to return to the Mozilla Project the pieces of our code that you altered.

Getting Involved

DE: I'm here with Dave Herman who works at Mozilla and directs research here. Tell us how you go involved in Mozilla and open-source projects.

DH: I was actually a graduate at the time that I first got involved. I was interested in JavaScript, and I was doing research in programming languages. I was spending some time learning about JavaScript, and I wrote web pages about. Out of the blue, Brendan Ike, the CTO at Mozilla and inventor of JavaScript, found the web pages that I had written and sent me an email and said that he was interested in talking to me about some of the things that I had learned and was doing. That experience alone was incredibly eye-opening to me. It showed me that the assumptions that I had had about the boundaries between research and industry, the boundaries between academia and industry. The boundaries between just a person on the internet and the movers and shakers in industry were a lot more porous than I realized, and it showed that the web is this place that brings people together who might otherwise not have had any way of getting to know each other. That was very exciting, and I started talking with Brendan and talking with Mozilla and got involved with the ECMAScript Standards committee, which is the standards committee that standardizes the JavaScript programming language. I was doing that while I was in grad school, and when I finished grad school I came to Mozilla full time.

DE: A lot of the students in our class have only been programming for about 7 weeks. Can they still get involved in open source projects or do they need a lot more experience than that?

DH: There are all sorts of ways of getting involved in open source projects. You don't have to be a computer science wizard to contribute. One of our largest sources of community volunteer work is localization, which is basically translation. Firefox is developed in English and our official release is in English, but we actually have close to 100 localized versions of Firefox in countries all over the world. We have volunteers in communities that could be as large as many countries or could be as small as a small community in one country where anybody can simply provide translations of the menu items and the windows and dialogue boxes--all of the stuff where we have to present something to the user we do in English, and then you can translate it. Now you can have contributed to your native languages version of Firefox. Another project that we started is a project called universal subtitles where you can take YouTube videos that were recorded in one language, and you can provide translation and the software will automatically create subtitles based on the translation that you provided. There are all sorts of ways that people contribute to Mozilla software and to open source software in general, and it doesn't always have to be technical. It can be code, and sometimes we end up hiring people who started out contributing code and ended up working on some of the most advanced parts of our projects. But it also can be as simple as editing documentation or providing translations.

DE: We have a real wide range of different types of people in our class. Can you tell us is it possible for anyone to get involved in open source software?

AD: One of the things that really appeals to me about open source software is the opportunity for a whole variety of people with backgrounds and different experiences and different interests to get involved with the project. A good example of this is if you look back to the Mozilla Project in 2002 and

2003, we were sort of struggling along with this next general Netscape communicator browser suite. And one day a young 15-year-old high school student showed up in one of our IRC channels where we all get together as engineers and testers and the people working all across the project and talk about the work we're doing, and he came in and said, "I don't like the way this particular button behaves in Mozilla." "Can someone fix that?" I reached out to him, and I said, "That button is actually powered by some simple JavaScript." "You could look at the source code and maybe you could fix it yourself." He said, "Well, I've put together a couple of web pages. I know some simple JavaScript." And before you know it he was a dedicated member of the team. Only a few short months later, he was one of the founding members of the new Firefox project that would go on to reach 500 million users today. This is a 15-year-old high school student with no background in computer sciences. He got involved. He had mentors who helped him learn the pieces he didn't know. He went and read books, and before you knew it he was building one of the world's most successful software products. We also have a number of students who come here, either for weekend work as something to compliment their education. We have people who are in the industry and have been developing software for as far back as the Fortran and COBOL days who are looking for something that's fresh and exciting and want to renew their skillset and see participating in open source software as a way to work with some of the younger people on some of the newer technologies and to refresh and get revitalized with their computer software skillset and interests. We have people who are part-time Mozilla contributors working on it because they're interested in a particular feature set. We have retirees and people who have part-time jobs trying to fill the rest of their day and who happen to know something about what we're doing somewhere on the Mozilla project. Sometimes that's writing client software code in C++, but other times that's manning our volunteer support desk and helping Firefox users fix problems with their browsing experience. So opensource really does, and many opensource projects really do, have multiple points where one can attach themselves to that project and find support and mentorship. Some of the places you attach yourself can be very simple. We'll often put some of our most valuable programming resources on very challenging projects and leave some of the easier or simpler features or smaller bug fixing unowned, as it were, by a full-time staff member with the idea that that staff member can instead mentor a volunteer or a student or a retiree to go finish off that piece of the work. This is a great learning experience and an opportunity to develop a set of skills that you can advance and go and become more deeply with the project over time. Ultimately, this young 15-year-old, Blake Ross who helped launch the Firefox project moved from fixing a couple of style issues using some JavaScript and CSS on toolbar buttons to being the authority on a bunch of the C++ code that made up Firefox. That was probably in a short time period, maybe less than 2 years that he was able to make that progression almost exclusively through opensource mentoring relationships on the Mozilla project.

Having An Impact

I hope everyone enjoyed the visit to Mozilla and is excited about the things you might be able to do to contribute to open-source software. It's really a great way to learn more about programming as well

as to work with expert programmers and contribute to making software that millions of people will use.

Benetech

DE: I'm here with Jim Fruchterman who is the director of Benetech. Tell us about what Benetech does.

JF: Benetech is Silicon Valley's non-profit technology company, and we exist to combat market failure. Basically, when something doesn't make enough money but does a lot of social good, we want to develop software that delivers solutions for that social problem.

DE: What kinds of things do you actually build?

JF: We write software in three different major areas. The first one is education, especially for kids with disabilities. Like we run the largest library for the print disabled, the blind, in the world online. We also write software for the human rights movement, whether you're a grassroots group or a truth commission or a war crimes tribunal. Our third area is the environment. We write project management software for environmentalists that are running campaigns or managing programs on the ground to actually have the same kind of tools you would have in a for-profit but because you're in the environment now you've got one.

DE: And heard you're starting a new project for coding for social good.

JF: Yeah, it's in a pilot phase right now. It's called "Social Coding for Good," and you can kind of think of it as an online matching site for geeks that want to do social good. You tell us your passion, your technical skill, and your time availability, and we'll hopefully match you up with a project like Wikimedia or the Guardian Project helping human rights groups or whatever other sort of project that matches up with your goals. It's just in the pilot phase, but we're hoping that in the next year it goes to scale so that geeks all over the world that want to do social good can get a chance to go to a place like GitHub, do some volunteering, and get some good cred for having done something socially good.

DE: How much experience would students need to have to be able to start doing this?

JF: Well, I think that people in different levels of the professional area of software development can actually help. I think our initial target audience is to work on people who are in the profession, people who are working for high-tech companies or software developers, but the human rights, the environment, and the education need people at all sorts of different levels. It could be characterizing a bug or writing some documentation, and our goal, of course, is people can increase and improve their skills by doing volunteering for social good, and then hopefully that leads to a job either doing social good or maybe a regular job.

DE: It sounds like you found ways to do lots of really beneficial things with technology. Can you tell us what are the social responsibilities that someone who gets power because they learned about technology should have?

JF: You know, I think that being educated in a profession like software development you have a responsibility to give back to society. Many people do that by actually getting a job and creating products that have value. The thing I want to talk to engineers is as you go out there and you see problems that are important to society and they can take advantage of your skills, don't ignore them just because they don't make a lot of money. I think that there are ways to make a living or volunteer to actual help these issues, and I think that that's really an area of terrific satisfaction as a toolmaker when you see the great things that people do with your tools. Solve bit social problems and actually find a way to change the world for the better, not just for more money.

Future Of Computing

For our next stop, we're going to get a glimpse into the future of computing. I'm here at the University of California Berkeley, which has one of the top computer science departments in the world. Lots of exciting things are going on. Let's go inside and see what people are working on.

Text Analysis


DE: This is Brian Gawalt who works on text analysis, which is a really important problem for search engines if we want to be able to do a better job of identifying the right documents. Tell us how that works.

BG: There are lots of different approaches, but the model that we tend to take transforms them into a really nice hashtable representation. Text is long and unwieldy and unstructured and prose flows and goes, but computers like things that are nice and structured and amenable in familiar things like lists, hashtables, etc. What we've got is a way to turn unstructured text into a data structure that computers are really happy with. One of those is the hashtable and the ???. We can start with a document that says, "Obama announces candidacy," and we can turn that into a representation of that document, and imperfect representation that just says let's keep track of every word and the number of times that word appeared in the document. We would just go through and say "Obama"--go to the hashtable and hash Obama and make sure the value associated with it is 1.
Similarly, "announces" hashes to 1, and "candidacy hashes to 1. If we were dealing with longer documents, these values would grow, and the number of elements in the hashtable would grow as well. Then we come across document 2, and we can represent that with a whole new hashtable. "Barak Obama elected"--Barak maps to 1, Obama maps to 1, and elected maps to 1. If it also said "President" down here, we'd add more and more keys and keep track of the value associated with each one. When it comes time to relate documents to each other, like we would do with our search engine, we can look and say, hey, there's an overlap between these two documents and the fact that they both used "Obama." If we've got enough documents and there are enough patterns between them, then we can start saying I'm seeing this pattern of "Barak" shows up in a lot of them or "candidacy" or "announcements" seem to come from him a lot. Now you've got a way of expanding this key word that maybe we're searching for "Obama" and the search engine can think when you say "Obama" you're really talking about this wide pantheon of topics that all of the do with the 44th presidency.

DE: How did you get started working on this?

BG: Well, I was an electrical engineer, and I went from circuits to signals to statistical signal processing, and it turns out I was working on how to tell if a cell phone is sending a 1 or a 0, and the math behind that wound up being broadly applicable to general problems, including how to tell if an email is spam or not. That's kind of the entrance to text processing in general from my background.

DE: I see. So what's the next step then? Well, a lot of the stuff that we work with all comes down to being able to compare two documents and then finding out how that comparison matches to whatever you're target is--detecting spam or not. All that works really well if the documents are next to each other, living close to each other within a single system, but that gives you a maximum of how many documents get analyzed. So distributing the solutions to these problems is, I think, where a lot of energy is being put forth.

Energy Aware Computing

DE: I'm here with Sara Alspaugh, who is a PhD student at Berkeley, in the lab where she works on energy-aware computing. So tell us about this here.

SA: I'm working on a project in which we try to create electric loads whose power consumption can be scheduled at a given point in time in order to make use of renewable supplies of energy like wind and solar which are only available when there's wind or sun and not available when we necessarily need them to be. And so on this project we tried to create a computing cluster in which we scheduled the work that the cluster did at times when there was wind energy available. So for instance, the types of work this cluster might be doing is you might have some set of these machines which act as web frontends and respond to search queries, and they would need to respond to the search queries as soon as the query came in, and you wouldn't be able to schedule this work any other time. But some of the other stuff that the computing cluster might be doing is building the index for the search engine, and that sort of work has some flexibility in when you actually can do it, so you can wait until there's wind energy available to schedule this work. And so these machines can do their work powered off of wind energy, and the rest of the time you can keep them in a low power state so they're not consuming energy.

Computer Security

DE: This is Adrienne Felt. Adrienne is a PhD student at Berkeley, and she's done a lot of interesting work on computer security. So what do our students need to understand about security?

AF: One of the first projects I looked at, which I actually worked here with Dr. Evans, was I use Facebook and I install and use Facebook applications. Many of you may be familiar with the Washington Post Social Reader. And at the time I was just curious. What are these doing with my data? How are they getting my data? How do these things work? And so I decided to learn more about how they work, and from there it led to basically what ended up becoming my thesis topic in a PhD

program. I'm still interested in how do applications get data and how do they use the data. When developers are writing applications that access users' private information or resources like their location or their camera, they need to keep in mind that they need to a) respect users' privacy and only use the resources that they need to use, and b) they have to be careful to build their applications in such a way that they don't accidentally leak this data to other applications. So I primarily focus on Android development and security of Android applications. One thing that's really important for an Android application developer to know is about the privacy and security of their applications. So when a user installs an application from the Android Market, they're shown things called permissions, which are small warnings that tell the user about all of the resources that the application gets access to on their phone. For example, for this application I'm looking at, the application can get access to location, the Internet, the list of contacts, camera, etc. So the user is warned before the application installation is completed that the application can do all of these things. So as a developer, it's important to only ask for access to resources that your application really needs. Otherwise you might unnecessarily scare off users who might otherwise want to install your application. So one of the projects we built was a static analysis tool that analyzes the permission usage of Android applications so that it can warn developers if developers are requesting more permissions than they actually need for their applications to work.

Theory Of Computation

DE: I'm here with Isabelle Stanton, who is a PhD student at Berkeley, and she works on theory of computation. So what are you working on?

IS: Lately I've been working in computational economics, which is this exciting thing that really Berkeley focuses on, so looking at problems that economists face and see whether or not we can develop algorithms and ways to solve this. So particularly the thing that I've been working on has been computational social choice. Social choice functions are ones that aggregate people's preferences. So think about this as a voting mechanism. Everybody gets to say, "I prefer Candidate A to Candidate B to Candidate C," and then we take everyone's votes and we output some kind of winner. And you'd really like this winner to be the one most people prefer or something. The problem that I've been specifically looking at is that back in the 1970s Gibbard and Satterthwaite showed that for any kind of reasonable voting mechanism it's going to be susceptible to manipulation.

DE: So is this like when someone doesn't vote for their preferred candidate because they think it's going to be better for them if they vote for sort of the more popular, less preferred candidate?

IS: Exactly. So if you knew that your favorite person was the underdog and there was no way, you might change your vote so that way your second favorite person would be the one that's elected. We've been specifically looking at Binary Cup voting protocols, which are familiar to people as single elimination tournaments.

DE: So how could someone manipulate a tournament like this?

IS: The tournament organizer is allowed to pick the seeding of the players, and that's the bracket, that's in each round who plays against who. For example, I can show you over here on the computer. Here on this slide we have actually the Women's World Cup. The seeding is this picking of Germany versus Japan, Sweden versus Australia. If I wanted something else to happen, maybe I could have picked a different initial seeding here, like maybe I have Sweden play England, and that would end up with a different result out of the tournament. Let's go over a simple example. Everyone is familiar with the game rock, paper, scissors. And in computer science we like things to be a power of 2, so let's add in hand grenade so that way we have 4 players. So we've got rock, paper, scissors, hand grenade. Now we can look at 2 different tournament brackets that have different winners. So if we have rock, hand grenade, paper, scissors, first rock wins this matchup and then scissors wins here and then rock wins the total bracket. If, on the other hand, we have rock face paper and scissors face hand grenade, then paper knocks out rock, hand grenade still wins, and hand grenade is the final winner. So the tournament manipulation problem is to figure out if there's a bracket so that your favorite player wins, like this, given information about the match outcomes.

DE: I'm not sure if this explains how the US lost the Women's World Cup, but we'll leave that as an open problem.

Quantum Computing

I'm here with Brielin Brown, who works on quantum computing. So Brielin, what does quantum physics have to do with computing?

BB: Quantum physics actually has a lot to do with computing. If you think about the algorithms that you've been learning about in your course for searching the Web and doing things like this, everything that you program is built on a fundamental model of computing that manipulates a registry of 0s and 1s. Everything is transformed into machine instructions to manipulate these bits. But in quantum computing, you actually get to replace these bits with something called qubits, which give you a lot more power.

DE: What can you do with a qubit that you can't do with a regular bit?

BB: With regular bits they're either 0 or 1 and there's no middle ground, but with a quantum bit you can actually exist in a superposition of 0 and 1. So right now I have this coffee cup which the viewer doesn't know whether or not it has any coffee in it. So right now it exists in a superposition of being filled and unfilled until I measure, a.k.a. take a sip. And so you can see that there is coffee in this cup. The cool thing about quantum information is that it takes a lot more classical information to describe the same system. But at the end of a quantum algorithm, to get the access for this information you have to take a measurement which actually destroys the system. So if you're really clever with how you devise your algorithm, you can sort of access this exponential amount of information in a way that allows you to solve a problem faster by making a correct measurement at the end. And so this allows you to do things like factor integers faster or simulate quantum systems, which gives you a lot of progress in physics that we haven't been able to see recently.

DE: Wow. All this talk about physics is making me thirsty. I think it's time to go get a coffee.

BB: Absolutely.

Stay Udacious

Ah! That's good. Or is it empty? You can't really tell. I hope everyone has enjoyed our field trips as much as I have. It's really been a great time doing this, and it's been an amazing experience to teach this class. You're nearing the end of the class but I hope just the beginning of your journey in computing. There's a lot more exciting things to do after this, including taking the courses that we'll be offering starting in April, but I hope all of you will find more exciting things to do in computing after this class. It's really been an amazing and wonderful experience for me to teach it, and it's been a privilege to have so many students from so many different places in the class and to see how much you've all contributed to make the class great and the effort that you've put into it. So I hope everyone has had a great experience. Stay udacious, and I look forward to seeing you soon.

Continue Learning

Whoa! We're not quite done yet. This was the original end of the class. And if you've made it this far and learned everything that we've covered, you definitely accomplished a lot. And you've learned a lot about computer science, as well as Python programming. But we've learned that it's very hard for students based on just this to be ready for the next level of classes. So we added a few extra units that I think will prepare you for that. As well as get you to the next level as a programmer. The main focus of the extra units. Is on how to manage complexity. This is something that becomes more and more important as your programs get longer and more complicated. As well as if you start working in teams to build larger programs. Where you have to understand other peoples code, and they have to understand your code. As you develop as a programmer, learning how to manage complexity becomes more and more important, and these new units will teach you tools and techniques for how to do that. I congratulate you on getting to the end of the original course, but hope you'll stick with it, and join me for the new units.