Oral History of Alfred Aho
A.M. Turing Award Winner, 2020

Interviewed by:
Hansen Hsu

Recorded June 13, 2022
Chatham, NJ

CHM Reference number: 2022.0084

Hsu: So, today is Monday, June 13th, 2022, and I am Hansen Hsu, and I am here with Alfred Aho. And so, to start with, tell us when and where were you born.

Aho: I was born in a small mining town in northern Ontario called Timmins. Timmins had the property that there was a fortnight where it never got above 20 below Fahrenheit. And so, when I was two years old, I persuaded my parents to move to a warmer place. They moved to Toronto, and I grew up in Toronto and went to high school and took engineering physics at the University of Toronto. Then, after Toronto, I didn't know what I wanted to do with my life, so I decided a good strategy is to stay in school as long as possible. I had done a senior thesis in minimizing Boolean circuits, and I became fascinated with how Boolean circuits could be modeled with the formalism of Boolean algebra. There was a professor at Princeton by the name of Edward McCluskey, who was one of the world's experts on minimizing Boolean functions at that time. I didn't know much about graduate schools at that time. There weren't any computer science departments when I was growing up, but I had heard of MIT, and I thought MIT might be a good place to apply to, and then I started getting these personal letters from Professor McCluskey at Princeton. I initially accepted MIT, but all they sent me were forms to fill out. And I kept getting these personal letters from McCluskey. So, after a while, I said why do I want to go to a university that treats me like a form when I can go to Princeton and study under this great man. So, I said to Ed I'd be delighted to come to Princeton. And so, I got my PhD in a program called digital systems in the electrical engineering department at Princeton, but my graduate education at Princeton really consisted of having one course in formal language theory from John Hopcroft and six courses from the Princeton math department. There was an interesting story about McCluskey because, shortly after I got to Princeton, Stanford hired McCluskey, the reason I went to Princeton. But Princeton retaliated and hired this young new PhD from Stanford by the name of John Hopcroft. John Hopcroft inherited the two students that McCluskey had. And he only had one research problem. He gave that research problem to the other McCluskey student, and I'm glad he did that because he told me, "Al, find your own research problem to work on." It took thirty years for the research community to solve the problem that he gave to the other McCluskey student. I spent my third year tearing my hair out trying to find a suitable PhD thesis topic. I can talk about, if you're interested, some of the work that I embarked on during and after Princeton.

Hsu: Sure. Yeah, let me go way, way back to-- we'll go back to your sort of early childhood. So, actually, you were mentioning to me sort of your parents' backgrounds. Could you talk about your parents' backgrounds and occupations?

Aho: My father was a carpenter. He had not gone to university. My mother was an executive secretary. She worked for the general manager of a pharmaceutical company in Toronto. So, none of my parents had even gone to college, but my mother was an avid reader. When I was in elementary school, we didn't have a cafeteria at school. I would walk home for lunch, and during the lunchtimes, I'd read some of the books that my mother was reading. She liked Shakespeare and Dickens and the classics. So, I got a very good background in Shakespeare, Dickens, the classics, Jules Verne, and that stood me in good stead in getting good marks in my English courses as I was going through school. The area that I grew up in in Toronto had a lot of immigrants, many people fleeing Europe because of World War II. The school that I was in had a lot of children of immigrants, and the children were very interested in academic things.

I noticed that some of the mothers of my friends were particularly delighted that their son was going out with me because I was good in school. I also played the violin. I ended up playing the violin for the bar mitzvah of one of my close friends in elementary school at the time. I thought it was an interesting occasion because I really wasn't all that good. When I started at the University of Toronto, I took engineering physics because it was very mathematically and scientifically oriented, and I liked mathematics and science. I had a very good time at the University of Toronto. Maybe I shouldn't tell this story, but as I was going to the University of Toronto, and the University of Toronto is a great university, I didn't think there were any smart people in the world until I came to Princeton. One of the first people that I met at Princeton was a Columbia graduate by the name of Jeffrey Ullman. He had just gotten his undergraduate degree from Columbia University and also had come to study digital systems in the EE department at Princeton. So, he and I became close friends. When we graduated from Princeton, we both joined the newly formed Computing Sciences Research Center at Bell Labs. There we developed a lifelong collaboration on subjects ranging from algorithms, programming languages, to the very foundations of computer science. I was very fortunate to have met some of the greatest people in the field and to have gotten to know them and work with them. You learn so much by working with the best people in the field. So, I felt very blessed because I had this kind of background.

Hsu: Thank you. Could you tell us about your family's ethnic background?

Aho: My family's technical back--?

Hsu: Ethnic background.

Aho: Ethnic background. My mother was born in the United States. My father was born in Finland. My mother had Finnish parents, so that's how I learned Finnish at home. I was raised by a grandmother, since my mother was working. My grandmother was from Finland. As I mentioned to you earlier, the first language that I learned was Finnish. When I went to Kindergarten, I couldn't speak a word of English. My Kindergarten teacher would write on her report card, "Alfred does not speak any English." And then the subsequent report card said, "Alfred speaks too much English." So, I learned English somewhat late in life, when I was five or six, but I really have forgotten all of the Finnish that I once used to know.

Hsu: And did your family have any particular political or religious beliefs?

Aho: Well, I guess Finns are Lutherans. My family had friends who were Lutherans and occasionally went to a Lutheran church, but I wasn't particularly religious until I got to high school. When I got to high school, my parents moved to another part of Toronto that had a very good high school with a musical program in it. A lot of parents, in fact, would relocate to that area, that part of Toronto, so they could send their kids to North Toronto Collegiate Institute. The school had an orchestra, senior orchestra, junior orchestra. It had an orchestra for the parents. It had a band, all sorts of quartets and other ensemble groups. I had a very good time at the school because I played in the orchestra. The person that I played beside was the concert mistress of the orchestra who went on to play for the Toronto Symphony Orchestra after she graduated. I wasn't nearly in that class, but I found music as a very good background activity when I was going to high school. The school had a music program that had a better reputation than the football team. We would go out and show what an instrumental music program could be like to other towns in and around southern Ontario. We would be able to go on weekend trips to local towns, and everybody thought that was cool. So, this was a very good activity. Then when I got to the University of Toronto, I started playing with the University of Toronto symphony orchestra, but I found that I could not compete with the music majors who were practicing four and five hours a day with my program in engineering physics. I just contented myself to play with the engineering band, which was called the Lady Godiva Memorial Band. It had some very eclectic instruments, like it would use garbage can lids as drums and timpani. On one occasion, we got a gig at the Bohemian Embassy, a nightclub in Toronto, where I got dressed up as a Gypsy and played Monti's Czardas late at night to the patrons of the nightclub. The band had red handkerchiefs in their costumes. They would take their handkerchiefs out at appropriate times and weep profusely as I played Czardas to the people in the restaurant. They wondered what in the world was going on, but we were undergraduate students; we were having fun. Playing the violin allowed me to meet some very interesting people. When I went to Stanford for a sabbatical, Donald Knuth mentioned that some very good mathematicians and computer scientists are also very good musicians. He said that somehow if the brain is wired for mathematics and computer science, it's also wired for music. He invited me to come and play violin and piano sonatas at his house. Don Knuth had designed a two-story pipe organ for his house, which he had some company in Los Angeles come and build. In his music room, not only did he have this pipe organ, but had a Bosendorfer grand piano and a Steinway. He, for our first occasion, presented me with the music for Grieg's Violin and Piano Sonata No. 3. This is a virtuoso piece for the violin. He said this was given to him by Dahl, a Norwegian computer scientist. He said Dahl gave him a recording of it. On the recording, it goes about like this. This is the tempo. So, he plays a few bars at the beginning to set the tempo. I look at my piece. I realized it was a virtuoso piece for the violin. And he expects me to sight read it at concert speed! That was the longest evening of my life, but somehow, I survived it. Later, he invited me again to come and play some pieces that I could play with him. So, music was a very good way to get to know Don Knuth. I had a very good time on my sabbatical at Stanford just because of this musical connection. And of course, Stanford was full of some of the brightest people in computer science at the time. I've played violin all my life. I've played in a string quartet since I started working at Bell Labs, and have continued that until just a few years ago when I found my shoulder was getting too tired to hold up the violin for three or four hours in an evening.

Hsu: Oh, wow. A couple years ago, we were interviewing Alan Kay in his home in L.A. And he had a pipe organ built in his home. So, that was reminding me of that.

Aho: Yes, I met Alan Kay. I didn't realize he also had a pipe organ in his house.

Hsu: Yeah, so music has been-- so, your mother was into music from--?

Aho: No, my grandmother--

Hsu: Oh, right.

Aho: My grandmother bought me a Delivet violin. I started taking lessons at the Royal Conservatory of Music in Toronto in elementary school. I guess I was in grade three or four when I started, so maybe I should have started when I was 18 months old. I might have been good, but I was able to play well enough to play in the senior orchestra beside the concert mistress in high school, so that got me to a very good start. I kept taking lessons from the Royal Conservatory of Music, which was very helpful. I've enjoyed music all my life. If we talk about some technical things, I could mention a few of the things that this interest in music has gotten me into.

Hsu: Oh, that will be fascinating. Do you have any siblings?

Aho: No, I don't. I'm an only child.

Hsu: And so, in school, would I assume that music and science of math were your favorite subjects?

Aho: Well, I found all subjects interesting. As I mentioned, throughout high school and college, I found the academic work quite easy. I don’t know what accounted for that. In high school, we had a math teacher, who was a veteran of World War II. He felt that he had to periodically go into the basement and have a cigar when he was teaching us math classes. He would say, "And if anybody has any questions about the math assignment, just ask Al Aho. He'll help you out." I found that helping other people with their math problems was also a good way to learn mathematics because you'd learn something much more deeply if you try to explain and teach it to others.

Hsu: Yeah. You mentioned earlier reading Shakespeare and Dickens. What other books or media were you really into as a child?

Aho: I was very big into science fiction. In fact, in high school, I wrote a hundred-page science fiction story. It was awful, but my mother kept it. When I was cleaning out her house when she moved out of her house, I reread it. It was a story about a future in which criminals were punished by recursively putting them into some sub-universe, and then they had to live in that sub-universe, but they didn’t know how deep the recursion was. So, if they wanted to escape from it, they didn’t know how many times they should try to pop up to get out of their universe.

Hsu: It's kind of like "Inception" a little bit.

Aho: Well, I've discovered that this is a plot that's been used over and over again. It was original with me at the time, and I enjoyed writing it. I enjoyed science fiction. One of my favorite science fiction movies of all time is "Forbidden Planet." I don’t know whether you've seen that or not. It's a story of an expedition that leaves Earth to visit a planet that has had a scientific colony set up on it some years ago. The expedition hadn’t heard from that colony, so when they go and visit this planet, they discover that there are only two people from the original scientific expedition left, the head of the scientific expedition and his beautiful 20-year-old daughter. What's interesting about this planet is that the head of the expedition was a scientist, and he had created a gigantic computer out of the core of the planet. This computer had this property that it would create and implement whatever thought he had. The theme of the movie was the monsters from the Id. His thoughts created these monsters that destroyed his fellow crewmen on the scientific expedition. Finally, the only people that were left on the planet were him and his daughter. There was also a very charming robot called Robbie the Robot that would do anything for you, create food, clothing, and so on. This was a movie from the 1950s, but it was very prescient of themes in science fiction movies today and also of AI, although, it far exceeds our abilities in AI at this point. And there's a good lesson in this as to whether you actually want to turn your thoughts into actions and concrete things that will implement whatever you are thinking. The monsters from the Id was a very apt paraphrase of this movie.

Hsu: Yeah, wow. Earlier you mentioned that story that you wrote that had recursion in it, that was in high school. So, you had already understood the concept of recursion? Had you had any actual formal exposure to that?

Aho: I can't remember. I picked up a lot of math on my own. One of the things I've been very interested in is precise definitions. I have a fetish for precision in communication. And this starts from talking with one another. If we use words whose meanings the other person doesn’t understand, we don’t have communication. So, it's very important to use words that both the speaker and the listener understand. The same is true for writing so that, in your writing, you should use words that you know your audience are going to understand. This gives to rise to the question of how do you define words that everyone will understand. If you're creating mathematics, how do you create definitions so that everything is built on top of well-understood concepts? Even today, there are questions of what should be the axioms that define mathematics. I'm sure you've heard of set theory, but what kind of set theory? Do you use naïve set theory or Zermelo-Fraenkel set theory? A lot of work was done trying to establish set theory, or naïve set theory, as the basis of mathematics, but then Bertrand Russell came up with this, "What about the set of all sets that are not members of themselves? Is that a set?" This question caused naïve set theory to be discarded and replaced by other forms of set theory for the foundations of mathematics. I was intrigued by proofs that we had to do in high school in geometry. I was also intrigued by what is a proof. I became interested in axiom systems and abstract data types. This actually is very good background for computer science, especially if you're into programming languages because you want a precise definition for a programming language, especially if you're going to create a standard for your language. This precision in communication has been a fetish of mine. A lot of my friends say that I ask too many questions because I often ask them to define what they're talking about. Sometimes, they realize that they don’t understand what they're talking about when they think about it more deeply. And that's disconcerting to them. And then I don’t feel so bad because I wasn’t understanding what they were talking about either.

Hsu: Did you have any influential teachers or mentors growing up?

Aho: Betty Bealey my English teacher in high school was a fan of precise communication. She mentioned the definition of the word irony, which I still remember. Maybe you'll remember this now. “Irony is the juxtaposition of incongruous elements.” I thought that was beautiful. How did she think of that? I haven’t seen that as a dictionary definition anywhere, but learning how to define things precisely and sometimes in a colorful way is very good. The professors in the Princeton math department that I took courses from were formidable, awesome. I had a course on probability theory from Feller, who was one of the giants of the field. I took a course on game theory from Morgenstern. I took on course on logic from Alonzo Church. I didn’t realize how good these guys were until long afterwards, but those courses were very much to my liking. And then John Hopcroft taught a course on computer science theory, it was really automata and language theory, and that really got me started on my interest in formal languages, programming languages, theory, algorithms and catapulted me into these domains of computer science that still fascinate me.

Hsu: So, you definitely entered into computer science from the math side. When was your first actual experience with a computer?

Aho: Well, I loved building things when I was in high school. I created, built, radios and amplifiers using Heathkits. I don't know whether you've experienced Heathkits where you get a kit, and you solder the parts together. I got to know what transistors looked like and what capacitors looked like and how circuits behaved. I had to know what an AND gate, OR gate, and so on was. My first real computer program was when I was going to University of Toronto. They had a 7090. Maybe it was a 7094. I had to write an assembly language program for the 7090. I really didn't think much of assembly language as a programming language. I preferred to think in higher level abstractions like Boolean algebra when I was couching solutions to problems. This gets into this whole area of abstractions and algorithms, which I've been a big believer in all my life. You notice in every area of human endeavor, you use abstractions, by and large, to talk about what your field is about. When we talk about computers, we don't talk about computers as collections of atoms unless we're talking about quantum computers, but to describe computers and digital circuits, you talk about adders, you talk about CPUs. These are abstractions that are used to build computers. If you look at programming languages, programming languages are built out of abstractions. If you look at any field of scientific endeavor, it starts off with a base of fundamental abstractions, and then the field is built up on top of that. Mathematics is particularly notorious for this. Physics has been trying to develop a theory of everything for as long as I can remember. They seem to be getting, or they say they're getting, closer and closer to it, but sometimes, they have to say, "Well, what about these new particles? Where do they fit into the standard model of physics?" Black holes are also very intriguing objects that stretch the limits of physics. I have met Brian Greene, and I like talking to him about string theory, a potentially good theory for physics. But it's tough to do experiments to validate string theory. Abstractions are what we like to use to talk about things, but there are many different kinds of abstractions. For example, the abstractions that we use for music are very different than the abstractions that we use for talking about food or for chemistry or for psychology. Sometimes, we can get very precise definitions of abstractions like integers, but how many people do you think have a good definition for real numbers?

Hsu: Uhm-- probably a mathematician would, the average person, I don't know.

Aho: Would you get into Dedekind cuts?

Hsu: I'm not sure if I even know what that is.

Aho: For some of the things that we take for granted, we have a great deal of difficulty coming up with a precise definition. Although the mathematicians have done a pretty good job of defining the real numbers and complex numbers, but some of these concepts are much more difficult for people who aren't grounded in mathematics. One of the areas that still baffles me is the abstractions that are used in quantum mechanics because one of my most recent research interests was in quantum computing. I thought some of the principles of quantum mechanics that underlie the field are very, very weird, particularly the phenomenon of entanglement. I don't know whether you've looked at entanglement or the formal definition for entanglement. If you know complex linear algebra, quantum mechanics is easy. Scott Aaronson, who is a famous computer scientist in quantum computing, says that quantum mechanics is easy if you take the physics out of it. If you just look at it purely as complex linear algebra, no big deal, but if you try to look at the collapse of the wave function, I don't think people really understand what happens in the collapse of the wave function, although the physicists love to talk about it.

Hsu: So, you mentioned-- actually, let me go back to what year did you start attending the University of Toronto?

Aho: 1959.

Hsu: '59, okay. And so, your major was engineering physics?

Aho: Yes.

Hsu: Okay. And so, that course that you mentioned where you programmed the 7090 or 7094, was that part of that major?

Aho: I think many engineering students had to write a computer program. Engineering physics was particularly interesting in that it was a special program. You had to stand first or second in your high school class to get into it, and they gave us the math courses of the math majors, the physics courses of the physics majors, and then they invented special engineering courses to give us. It was quite comparable to the kind of engineering program that you have at MIT or Caltech. It was highly mathematical, heavily rooted into physics and science, and at Toronto, it was particularly brutal because the entering class of engineering physics had about 150 students in it. Four years later, 39 of these bright students actually graduated. The rest of them were asked to go into some of the other engineering departments, where they often stood first or second in their class. Toronto brutalized the students in engineering physics, but maybe that's good experience for real life because you don't know when you're going to meet the best people in the field. As I mentioned, when I first came to Princeton, I met Jeff Ullman, and I thought, "What planet am I on? Where do they make people this smart," because he was one of the brightest people that I have ever met. When I joined Bell Labs at Murray Hill, it had all these Nobel Prize winners, brilliant mathematicians, and it was just like Princeton, where you had all these brilliant faculty members, math people and the people in the engineering school. So, I was ratcheting up. Although soon after I got to Bell Labs, I said, "Gee, great, I made it in here." Then after a while, I said, "How do I survive here because I have to show something that will let my management say we need to keep Aho, here." They did not have a tradition of tenure in Bell Labs Research. So if you did not perform, after a while, you found jobs in other parts of Bell Labs and AT&T. It was a way to keep the research area vibrant, but on the other hand, it kept people on their toes.

Hsu: I sort of want to move a little bit. So, how did you choose the engineering physics major when you first got to Toronto?

Aho: Since I was interested in math and science, It was a choice between engineering physics in the engineering school, and math, physics, and chemistry in the arts and sciences school. There was a student in my high school class, who I couldn’t stand, who was very bright. He went into MPC, math, physics, and chemistry. So, I said, "That does it. I'm going to take engineering physics," a personal decision.

Hsu: Okay.

Aho: I don’t regret it at all.

Hsu: Earlier, you mentioned starting at Princeton. So, remind me again what major or field was your PhD at Princeton?

Aho: It was in electrical engineering. They had a computer science/computer engineering option in electrical engineering. It was called digital systems at the time. This was long before Princeton got its own computer science department. Sometime ago, I was on the advisory committee for a combined department of electrical engineering at Princeton called electrical engineering and computer science. After a decade of trying to persuade the presidents of Princeton University that computer science deserved a department of its own, the committee finally convinced a new president that they should do this. And so, he promptly built a building for the new computer science department at Princeton. The department at Princeton in computer science is now one of the top departments in the country. It's amazing how long it takes academia to respond to changing conditions. In Europe, sometimes the charter of the university is embodied in the state constitution. In order to set up a computer science program in European universities, you had to modify the state charter, the state constitution. So, it took a long time for some universities in Europe to create computer science programs. They called these programs informatics, by and large.. I don't know whether we want to get into this subject. Since technology is changing so rapidly and is becoming more and more computerized, how do ordinary people keep up with the world and find jobs in the new world in which the technology is changing so rapidly. If I only know what I knew when I graduated from Princeton, I'd be obsolete. Lifelong learning is the adage for survival these days, but I don't know that people are trained to do lifelong learning.

Hsu: Earlier, you mentioned that-- so, when you ended up working with Hopcroft that he assigned another graduate student this research problem, and you had to find your own, who was that other graduate student?

Aho: Al Korenjak.

Hsu: And then you also met Ullman right from the beginning at--

Aho: Yes, he entered Princeton the same year I did. We met at Princeton, and then he graduated, took a job in the Computing Sciences Research Center at Bell Labs, and a few months later, I took a job in the same center at Bell Labs. There were only very few opportunities for people interested in computer science at that time because academia hadn't yet set up computer science departments very widely. Bell Labs, as a research institution, was legendary. It had this huge list of inventions and innovations that have changed the world ranging from the transistor to discovery of the Big Bang to inventing information theory. It was just a logical choice, if you were interested in research, to go to Bell Labs.

Hsu: Before we jump into Bell Labs more deeply, could you maybe explain-- talk about your PhD thesis, but try to explain it to somebody who, maybe like a museum goer who doesn't really know much about computer science and linguistics.

Aho: This is interesting. As I mentioned, Hopcroft told me, "Find your own research problem." He did teach a course in automata and language theory, so I got introduced to formal language theory and automata theory, at least, as it was known at that time. I was interested in programming languages and compilers. What I noticed was that a programming language has a syntax and a semantics. All languages have a syntax and a semantics. If you want to write a translator for a programming language, or even a natural language, you have to understand the syntax and semantics of your source language and the target language. Specifying the syntax of programming languages was done with some type of grammatical construction at the time, and the primary grammatical constructs that were being used were context-free grammars. In the programming language [field] context-free grammars were called BNF, short for Backus Naur form. Chomsky, a linguist at MIT, had created a hierarchy of grammars and the context-free grammars were the second level in the Chomsky hierarchy. They could specify some, but not all of the syntax of [a] programming language, and there were constructs in programming languages that could not be specified using context-free grammars. I was interested, can we extend context-free grammars so they can specify more of the syntax of programming languages. I created a generalization of context-free grammars I called indexed grammars, and then I found an automaton analogue for indexed grammars just the same way as pushdown automata are an automaton analogue for context-free languages. I devised an automaton I called the nested stack automaton, and the nested stack automata can recognize precisely the indexed languages. Now we have this same duality that we had with context-free languages. You have a grammatical specification and an automaton specification for indexed languages. This became my PhD thesis and you can read about indexed grammars and nested stack automata on Wikipedia if you want to delve into the details of this. They have closure properties and decidability properties that are similar to those of context free languages, which makes them useful for more practical applications, although they aren't widely used for programming language specification. They've been used for some biological specifications by some people and they also formed the first level in a hierarchy of languages between the context-free languages and the linear bounded automaton languages, also known as the context-sensitive languages. You can view the indexed languages as applying a duplicate operator to the memory structure of a pushdown automaton: this is a nested stack. Memory becomes an ability to create a stack of stacks, and then for the next level in the hierarchy, you can have stacks of stacks of stacks, and so on. So indexed languages were an instance of an abstract family of languages, a research effort launched by Ginsberg and Greibach at the time. Hopcroft and Ullman were looking at a related topic called balloon automata which are generalizations of automata.

Indexed languages fit nicely into their hierarchy as well. So it's a somewhat natural class of languages if you like formal languages. This then became my PhD thesis. I was tearing my hair out in my third year trying to find a PhD thesis topic, and it took me about six months to come up with the idea of indexed grammars and nested stack automata, but after I got the idea, I was able to write up a PhD thesis out of them within another six months. So I escaped from Princeton in three years and three months, roughly.

Hansen: Wow.

Aho: Ullman did it in three years, but he's much smarter than I am.

Hansen: How did you get interested in programming languages and compilers in the first place?

Aho: I've been interested in languages all along, as I mentioned, dictions to specify abstractions for whatever you're doing, and I thought how do you specify abstractions when you're talking to computers, and particularly about algorithms? I was very interested in algorithms. My definition of an algorithm for my mother was an algorithm is just a recipe for doing something. The computer science definition of the term algorithm is a Turing machine that halts on all inputs. What's kind of interesting is this is Knuth's definition of an algorithm too, that an algorithm has to halt on all inputs. But I discovered that there are some authors who don't require an algorithm to halt on all inputs. So these authors have bastardized the definition of the fundamental term algorithm, and I think that's a great crime, and Knuth also agrees with me. We called, in our Design and Analysis of Computer Algorithms book, a recipe that doesn't have to halt on all inputs, a procedure. In mathematics, in the area of recursive function theory, we have the distinction between recursive functions, which correspond to the class of functions you can define with Turing machines that halt on all the inputs, and partial recursive functions, which are functions that can be defined by Turing machines that don't have to halt on all inputs. That's a natural distinction from recursive function theory and I think it's nice to have these parallels with mathematics. I think formal proofs, specifications, languages, they're all neatly interwoven.

Hansen: Okay, let's get to Bell Labs. You joined Bell Labs in 19…

Aho: '67.

Hansen: 1967, and you followed Ullman there. He had already joined Bell Labs before.

Aho: A few months before me.

Hansen: A few months before. And what group was it that you joined?

Aho: I was interviewed by a department head by the name of Doug McIlroy. He was an applied mathematician from MIT. He had been at Bell Labs for a few years before me. Amongst other things, he had coinvented macros for programming languages and he's also in this class of one of the smartest people I've ever met. But when he interviewed me, and decided to hire me, I show up for work, and he says, "Al, why don't you do what you think is important?" I said, "I think I can handle that job charter." So I was left to work on whatever I thought was important, and since Jeff Ullman was there, I had a person who understood computer science. He and I embarked on a bunch of projects related to automata theory, language theory, algorithms, and programming languages. Ultimately, it went into the very foundations of computer science. Ullman and I also did some work on database theory. We went through a whole bunch of fields in computer science that had what we thought were interesting problems for us, and in the first 10 years at Bell Labs, I wrote 40 papers and 5 books, several of them coauthored with Jeff Ullman, and many of the papers were coauthored with Jeff Ullman, but also with other people at Bell Labs. What I found interesting about Bell Labs, it was teeming with interesting, talented people, and interesting research problems, and coming from the field of computer science, I would think about problems in a different way than some of the people working on these problems. There were stories where I would be sitting in a seminar and doodling on a piece of paper. The person next to me would look at my doodles and say, “Where did you get that?”, and then by the end of the day, we'd write a joint paper. I had a number of stories of this nature that collaborating with interesting people often results in solving problems that you may not have thought of, or you didn't have a solution to. Jeff wanted to go to academia a little bit earlier than I did, like many years earlier. He stayed at Bell Labs for a few years and went to Princeton University where he joined the faculty of the electrical engineering department, but he would come and spend one day a week consulting at Bell Labs. His consulting stint was he would come Fridays and sit in my office all day. The conversations that we'd have would range over all sorts of topics, and sometimes he'd mentioned that he was working on a problem with a colleague at Princeton, and after describing the problem, I might say, “You're kidding,” and he said, “Oh, you're right. The solution is obvious, isn't it?” I don't know whether I would say dynamic programming or whatever, but several papers came out of this intense collaboration, and we got to the point where we could communicate with just a few words. We had a very large, shared symbol table.

Hansen: In your minds.

Aho: In our minds, yes. But it was a very productive relationship. Another Bell Labs anecdote. I mentioned that in the first 10 years, I wrote a bunch of books, 5 books. One of the books was called the Design and Analysis of Computer Algorithms with my adviser John Hopcroft, and with Jeff Ullman, and it codified some of the work that we were doing on algorithms at the time. I found myself teaching at a local university at Stevens [Institute of Technology]. I would use the notes for the book in my lectures, and I found that doing research, teaching, and writing were very synergistic. The president of Bell Labs heard that I was writing a book on algorithms. This leads to a story associated with algorithms. I was giving a seminar on algorithm design techniques at Bell Labs and at the end of the seminar, a young woman came up to me and said she had just written a bibliographic search program. Bell Labs used to get a tape of current technical papers and reports that had been written under government contracts, and she had written a computer program that could search this tape for keywords and phrases, so that scientists at Bell Labs could see what kind of work was being done under government contract. She mentioned to me right after my talk that some gung-ho bibliographer had specified a very large search specification, a query that had scores of keywords and phrases in it. The program that she had written had a $600 limit on how much computer time would be spent on one search, and after spending the $600 on that big search, it had not yet finished printing out all of the answers to the search. So she said, “Can you help?” I asked her, “How are you doing the search?” She explained the algorithm she was using, that she'd read a buffer full from the tape, look for the first keyword, the next keyword, and so on, and then repeat that on the next buffer full that was read. I said, “Well, what you could do is you could construct an automaton that will search for the keywords all at once. It would do it in parallel, and by the way, here's a spiffy way of constructing this automaton in linear time.” I mentioned this to her at the end of the talk. A couple of weeks later, she comes into my office, and says, “You remember that search that used to take $600? It now costs $25. Every search cost $25.” That was the cost of reading the tape. What had been a computationally bound problem had turned into just the cost of reading the tape. I mentioned this to my boss, Doug McIlroy with maybe a little less description than I already gave you. It was on a Friday afternoon. I discovered you should manage your boss. It's a good strategy no matter where you are. My job of managing my boss was every Friday afternoon or every second Friday afternoon, I'd go into his office and tell him what I've done in the past week. When Midge Corasick told me about what happened with her computer program, I mentioned this to Doug McIlroy in about two or three minutes. I outlined how I had constructed the automaton and done the search, but not in any detail. The next Monday at around 11:00 or 12:00, the head of the math center at Bell Labs comes into my office, and says, “How do you compute that failure function?” I said, “I beg your pardon? What failure function?” He said, ”Your boss just told us about this new algorithm you have for fast bibliographic search, and it uses a failure function. How do you compute it?” So McIlroy in two or three minutes understood technically what I had done, created a talk for the three-level management seminar that was held in research describing the algorithm to mathematicians and physicists, and gave me the title for the paper that we should write, Fast String Pattern Matching, an Aid to Bibliographic search. Somehow the president of the Bell Labs heard about this. He stopped me on the aisle as I was walking to lunch one day, and said, “Oh, Al, you should keep working on those algorithms. They'll be important someday.” This president was Bill Baker, and he was notorious for saying things in a very cryptic, but encouraging way. He also got me to teach a course on algorithm design to engineers at Bell Labs. I gave a televised course on algorithm design techniques from the Design and Analysis of Computer Algorithms book and many people then at Bell Labs got to know me through watching these videos of my algorithm lectures. At the same time, I should perhaps mention that there was a scientist at Bell Labs by the name of Richard Hamming, of Hamming Code fame. He had an opinion on everything. He was a very interesting person, a very talented person. In the first week that I was at Bell Labs, on that Friday afternoon, he came into my office, put his feet up on my desk. I had two desks. One desk like this and another one at 90-degrees to it. There was a chair next to the angled desk. He sat in the chair, put his feet up on that desk, wouldn't allow me to do any work, and said, “Al, it's Friday afternoon. It's time to think great thoughts.” What he said to me at that time was, if you write a book on your scientific work that becomes popular, this becomes a very good way to develop a scientific reputation. He was a big fan of encouraging people to write books on whatever they were working on. This may have been in the back of my mind as Hopcroft and I were working on algorithms, and writing this book, the Design and Analysis of Computer Algorithms. This was back in 1974. When we wrote the book, algorithms and computer science departments were brand new in universities around the world. This book was used at many of the new computer science departments around the world, and it became one of the most cited books in computer science, and hundreds, maybe thousands of people took courses out of that book, and sort of created the reputation for us of being in algorithms. One of the things that Ullman and I get cited for in our Turing Award is our work on algorithms. A few years ago, Hopcroft won the Turing award with Tarjan for his work on algorithms, which also mentioned the Aho, Hopcroft, and Ullman book. I was very happy to have had the association with Hopcroft. It also shows the impact that writing books and teaching can have on your scientific career. Now I always recommend the importance of writing books to new scientists: if you think you have something worthwhile to say, say it in such a way so that you can get many people around the world to talk about your scientific accomplishments. This same story is particularly true about the dragon books that Jeff and I wrote, which, if you want to talk about programming languages and their translators, we can do that at however long a length you want.

Hansen: I mean, that's a natural segue about the dragon books. Talk about how those came about.

Aho: Okay. So in the 1970s…

Hansen: Well, first explain that they're about compilers, correct?

Aho: Okay, so maybe I should start with the center that I joined at Bell Labs in 1967. It was newly created. It got to be called the Computing Sciences Research Center. It is the lab that invented Unix, C, C++, plus a whole bunch of other things. The Unix operating system was just being invented and developed at that time by Ken Thompson, Dennis Ritchie, and a bunch of the people that had joined the Computing Sciences Research Center. I discovered that I could write scientific papers using the new programs that were being created on the Unix operating system. It was much better than writing a paper longhand, giving the longhand to a secretary, who would then type up the longhand into words, and then the author would have to correct the words, and so on. I was one of the early users of Unix because of its capabilities for being able to typeset scientific documents.

Hansen: So you were using a text editor and typing your papers directly.

Aho: Yes. That's the only way to do anything these days.

Hansen: Right, and was there like automatic formatting? I mean, was this probably before TeX had been invented. But were you using some sort of format?

Aho: Long before TeX was invented, long before LaTeX. Have you ever heard of roff?

Hansen: Yes.

Aho: Have you ever used roff?

Hansen: No.

Aho: Roff was one of the early editing programs and typesetting programs on Unix. For the terminals, we had teletype machines, machines that weighed not 10 or 20 pounds, but 100 or so pounds. I had a teletype machine in my office here.

Hansen: Right here in this office?

Aho: Yeah, with a T1 connection to Bell Labs for high-speed connection. This was one of the benefits of working in the Computing Sciences Research Center. But we went through a whole progression of computer terminals, big, clunky computer terminals, before we got to personal computers, iPads, cell phones, and so on. But as Unix was being developed, Ken Thompson created the first two versions of Unix using assembly language. He had joined Bell Labs at roughly the same time I had. He was there maybe six months or so ahead of us, and he had been assigned to work on the Multics project that Bell Labs was part of with MIT and GE. When Bell Labs got tired of pouring money into Multics and not getting the operating system that it had wanted, it abandoned the project and left Ken Thompson to his own devices. Ken thought there were some good ideas in Multics. Being the genius that he was, he said, I can do it much more simply and much more elegantly. So he created a rudimentary version of Unix and then kept writing and polishing it. Dennis Ritchie came on the scene. Ken had also created a programming language, B. The B was maybe the first letter of BCPL. Who knows? But when Dennis Ritchie looked at it, he said, what B needs is a decent type system. So he put a decent type system on B, and created the C programming language. Thompson and Ritchie wrote the third version of Unix using the newly created C programming language. I became an early adopter of C, and I had C wired in my fingertips, so I could write C programs quite readily, and of course, there were all these neat tools that accompanied the programming environment on Unix. There were the text editors. I don't know whether you've ever heard of the ED editor or the QED editor that was at MIT as part of Multics. QED had regular expressions in it. This triggered my interest in regular expressions. Ken Thompson had written a program called grep for doing pattern matching on text files, and it had a very limited form of regular expressions when I encountered it. Having studied regular expressions, I thought Unix deserved complete regular expressions, so I created a generalization of the grep program called egrep and put in a fast pattern matching algorithm into egrep.

Hansen: Was this the same algorithm that you had created for the bibliographer, the Aho-Corasick algorithm?

Aho: With the Aho-Corasick algorithm, I had created a tool called fgrep for fast grep, but it only did keywords. It did not do full regular expressions, but it had the spiffy algorithm that I had created on the spur of the moment with Midge Corasick and had put it into the fgrep program. I wrote a paper with Midge on bibliographic search that became one of my most widely cited papers, and fgrep became part of the set of tools that appeared with Unix. Now fgrep is part of the grep utility, and egrep is also part of the grep utility. They're all combined into this grep program, and there's a lot of interesting anecdotes associated with these programs. Doug McIlroy liked the full, regular expressions that I had put into egrep, and he was using them in a calendar program. As you know, dates around the world have many different styles for representing dates. We could talk about 6/13/22, or June 13th, 2022, or in other countries of the world, you have a different order between the month and the day, and so on. Doug wanted to have a calendar program that dealt with dates in many different formats, and so he wanted to do the search for the dates using egrep. But egrep initially constructed a deterministic finite automaton from the regular expression, and then it could execute the deterministic finite automaton very quickly, very efficiently with only three machine instructions per input character. But this was a little bit like constructing a Boeing 747 to go across the road. You spend all of your time constructing the airplane and very little time flying it. So he mentioned that the time to do searches wasn't that fast. Then, I said, "Well, let me construct the deterministic finite automaton incrementally and on demand." I put in an incremental on-demand algorithm into egrep, and it made regular expression pattern matching instantaneous, and it made my boss very happy too, which is a good thing to do. Just the feedback you get from users and the interactions that you have with users often help you create a much better product than you initially had in mind. This happened repeatedly with the Unix operating system: it was written and rewritten. Ken Thompson didn't mind throwing out an entire operating system and rewriting it if he thought he knew how to do it better. Since the Unix operating system was written in C, there was a colleague of mine in the Computing Sciences Research Center, somebody who had done his PhD thesis at Columbia in category theory. I don't know whether you're familiar with category theory. It's one of the most abstract theories in mathematics there is. But when Steve Johnson came to see me, he wanted to write a C compiler for a different machine, and he had trouble writing the syntax analyzer, the parser for C. He had heard that I was into formal languages and parsing algorithms. He said, “How would you create this parser for C?” and I said, “Well I’d first write a grammar for it and then I'd use Knuth’s LALR parsing algorithm for it.” He said, “Would you mind showing me how to do this?” I said, “Not at all.” So I got a big sheet of cardboard from the stock room, sort of two by four, and I did the sets of items construction for the LR parsing algorithm on this sheet of cardboard from the grammar for the language that Johnson had given me. I'd usually do it watching television over the weekend and I'd come on Monday morning and present Johnson with the sheet of cardboard with the specification of the parsing algorithm, the automaton. He'd implement it, and of course, it wouldn't work properly because there were mistakes in it. The next weekend I’d take home the automaton, correct the mistakes, and then present him with the corrected automaton. He'd put it into the machine and it still had mistakes. I think on the third iteration, he said, “Why don't you tell me how you're actually constructing this?” So I explained the LR parsing algorithm to him and what I was doing. He implemented it, and it became the parser generator, Yacc on Unix. Jeff Ullman, Steve Johnson, and I worked on making the construction of LR parsers much more efficient. We wrote a number of papers on this, and Steve Johnson just kept rewriting Yacc to make it better and faster all the time. Yacc became the tool of choice to construct parsers, and it was based on formal language theory at its heart. Going back to its roots, the original idea of the LR parsing algorithm was due to Don Knuth. I can go into much more detail on how to do this. Many people actually wanted some help with the explanation of the shift-reduce and reduce-reduce conflicts that resulted from the grammar that they used. One of the nice things about Yacc is that it would tell users if there were any difficult to parse constructs in the grammar that they had written for their programming language, so it allowed users to get much more accurate grammatical specifications for their languages. Over a period of several years, Yacc became one of the most widely used tools for constructing parsers, particularly in computer science courses. The fast regular expression pattern matching algorithm that I had created got incorporated into a tool for creating another component of a compiler called the lexical analyzer. There was one interesting summer where Eric Schmidt was a summer intern at Bell Labs, and he took one of my fast regular expression pattern matching algorithms and put it into this lexical analyzer generator program called Lex that Michael Lesk had first developed. Using Lex to develop the lexical analyzer and using Yacc to generate the syntax analyzer became a very quick way of constructing the frontend of a compiler. The combination allowed you to experiment with programming language design. Many of our colleagues at Bell Labs started using Lex and Yacc in constructing little languages. Brian Kernighan and Lorinda Cherry were amongst the early users. Kernighan and Cherry developed a little language for specifying mathematics called EQN using these tools. Brian Kernighan’s model for how you should design a language for specifying mathematics is based on how a mathematician would describe an equation to you over the telephone. You could say A squared is equal to B squared plus C squared, or just A sup 2, using the superscript operator sup for exponentiation. A sup 2 represents A squared. You could put subscripts in and so on. People started using the Kernighan and Lorinda Cherry EQN tool to specify mathematics in their documents and in the research papers that they were writing. They would feed the EQN specification into the typesetting program roff. We finally got a phototypesetter and Kernighan developed a phototypesetter version of EQN that allowed us to produce professional looking papers instantaneously. Sometimes when we submitted a paper to a journal or a conference, the conference chair would say, “Did you already publish this because this looks professional?” One of the purposes of these document preparation tools was to allow Bell Labs to be able to type patent applications without having to retype them. Because if the typist made a single mistake on the patent application, the typist would have to retype that page. But with the computer editing tools and typesetting tools, it became very easy to create perfect patent documents. Also, if you were willing to diligently proofread your papers, you could get all the typos out of your papers. The first books that I wrote were done the old-fashioned way of writing them out in longhand, giving them to a typist, then sending the typed pages to the publisher. The publisher would then hire basically, a typesetter to take that and typeset the documents, and you’d get the galley proofs, and then page proofs. You'd correct those, and then finally, you’d get to see the book. The first books that I wrote took almost 18 months to go from manuscript to the finished book. For the subsequent books that I wrote, it just took a few weeks from submitting the typeset papers to the publisher to get the physical book in my hand. In addition, I could apply all these correction tools for both the grammar and the language that took out grammatical mistakes from the manuscripts. There was a huge increase in productivity in writing papers and patents. We also developed tools not only for typesetting mathematics, but for typesetting figures and pictures. Knuth adopted the EQN language to include in the TeX typesetting system, and in LaTeX. It’s basically Kernighan and Cherry’s way of specifying mathematics. These software tools had a great deal of influence, and Kernighan and Cherry enjoyed the fruits of parsing theory and formal language theory in using the tools Lex and Yacc to create their EQN typesetting language. Knuth has this saying that the best theory is motivated by practice and the best practice by theory. I internalized that with my early experience in the Computing Sciences Research Center because I found that the theory that we were developing in computer science could be applied to document preparation systems, programming languages, compilers, and so on. It was really a very productive environment. I taught courses on compiler design at local universities, and then when I went to Columbia, I would teach the course on programming languages and their translators. This is a senior/graduate level course, and I would have the students not only create a translator, but they’d first have to create a new programming language of their own design. They’d work in a small team, and after creating the new programming language, they'd have to write a translator for it. Sometimes the students would say, “This professor is out of his gourd. Not only does he want us to create a new programming language, he expects us to write a translator for it.” But what I noticed was that with the tools that we had and the experience that I got at working at Bell Labs on creating these tools and some languages of my own with some others like AWK, it became routine to be able to do this in a 15-week course at Columbia. I might point out that the first Fortran compiler developed by IBM in the 1950s took 18 staff years to create. In my programming languages and compilers course, I organized the students into teams of four or five. Each team had to create their own programming language, and then write a translator for it, and in all the time that I taught the course for almost 25 years at Columbia to thousands of students, never did a team failed to deliver a working compiler in the 15-week course, and I attribute that to the abstractions and algorithms that we put into our dragon books and to the tools that we created to support these abstractions and algorithms. I can go into much more detail on the abstractions and algorithms, but this would be an entire semester course.

Hsu: We covered a lot there. First, I want to clarify, we were talking about fgrep and egrep. What exactly is the relationship between the two? Is egrep a generalization of fgrep?

Aho: No. They're two separate programs.

Hsu: They're two separate. Okay.

Aho: And they were separate utilities. They were written in the early 1970s and they were put onto the very first versions of the Unix operating system.

Hsu: Right. Okay. So egrep is the one that that parses regular expressions and fgrep…

Aho: Just does keywords.

Hsu: Just does keywords and fgrep uses the Aho-Corasick algorithm.

Aho: That is correct.

Hsu: But egrep does not or does it also?

Aho: egrep uses my own algorithm for dynamic, lazy construction of the deterministic finite automaton, and going back to formal language theory, constructing the deterministic automaton can take time that's exponential in the length of the regular expression. So if you have a long regular expression, and you have an algorithm that runs in time, two to the length of the regular expression, it can take a long time to produce that. Doing it dynamically, the observed time was just you read in the regular expression, and you construct the state transitions of the automaton as they're needed. You construct those parts of the automaton that are only needed for matching the regular expression.

Hsu: Okay, so that brings it down to…

Aho: Linear and this is observed. Linear in the length of the regular expression plus the length of the input, rather than exponential in the length of the regular expression plus linear in the length of the input. But that exponential part of it dominated. That's constructing the Boeing 747 to go across the street. Now you construct a motorcycle to go across the street. It's very efficient.

Hsu: Okay, and we talked about Yacc and Lex, but we actually never got to the actual story of the dragon books themselves. Could you talk more about how that started?

Aho: Jeff had bought into this idea that it's good for your career to write a book about what you're working on. In the ’70s, with all this work on Unix and C, there was a lot of interest in creating new programming languages and compilers. As with the algorithms book, what we did was we performed research on efficient algorithms for parsing and for some of the other phases of compilation, wrote papers on those and presented them at conferences. But we took the important ideas that we developed and the community had developed over several decades and codified them into what are now called the dragon books. The first dragon book was published in 1977. It incorporated the experiences that we had at Bell Labs in creating the Lex and Yacc tools, and some of the experiences we had in using the Lex and Yacc tools to create programming languages, also what people like Kernighan and Cherry had done in creating the typesetting EQN language and other languages. It had relevant theory that could be useful in practice to create compilers and programming language translators. It was theoretically sound. Jeff and I are theoreticians at heart. We did have theorems and proofs in the book, and Jeff had this brilliant idea that the book should have a cover with a fierce dragon on it representing the complexity of compiler design, and then a knight in armor with a lance. The armor and the lance were emblazoned with techniques from formal language theory and compiler theory to slay the complexity of compiler design. The first book was written in 1977, and it had just Jeff and me as coauthors, and it was 603 pages in length. It represented what we knew about translating programming language with compilers. In the 1980s, more was known about how to construct efficient compilers. We invited Ravi Sethi as a third coauthor, he was at Bell Labs at the time, to join us in creating the second version of the dragon book. In the first version, the dragon was in red. This second version, the dragon was-- sorry. In the first version it was in green. In the second version the dragon was in red. What was interesting about the red dragon book was there was a movie that was created in 1995 titled Hackers with a young Angelina Jolie in it, and in the movie, there is the uber hacker that's explaining to the new hackers what you have to read to become an uber hacker. He shows them 10 papers and books that you must read, and one of them was the red dragon book. When my two children saw this movie, and they had seen the red dragon book at home, this is the first time they thought their old man was really something because he had one of his books in a Hollywood movie. It shows what you have to do to impress your kids these days. The red dragon book was 800 pages. In 2007, we invited Monica Lam as a fourth coauthor to create a third version of the dragon book that had a purple dragon on the cover and it was close to a thousand pages. None of us had the heart to write a fourth book at this point because it just shows how much new knowledge had been created in the area of programming languages and compilers and their translators, and we continued to do research in this area to keep up with it. I haven't really done that much research in compiler design in the last part of my career. But there are still lots of interesting open problems left and one of the most intriguing aspects of compiler design is can we use AI, machine learning, and large language models like GPT-3 to create code automatically from written or spoken specifications. That's still an unfolding story and I'm not willing to trust any program created by an AI program at this point. I wouldn't want it in my pacemaker. I wouldn't want it in my self-driving car or in my airplane. But maybe for a computer game, it's okay. This is what they're creating with these at this time. So even the area of programming language translation is undergoing new approaches and how successful they will be is yet to be determined. But one of the things that intrigues me greatly and has intrigued me from since I was a little kid was how does the human brain work-- what kind of algorithms does it use to process ideas? You hear my voice. What kind of algorithm does your brain use to convert the sound waves hitting your eardrum into meaning in your brain? The brain is a fantastic compiler, language translator and nobody knows what algorithms it really uses. So there's that interesting aspect yet of compiler design. How does the human brain do it, and I don't know, when, if ever we'll really understand that because this shows you how many interesting problems are left in programming language translation, depending on what your model of computation is, and what kind of computing device you use to do it with.

Hsu: excellent. Can you tell us what is AWK and how did it come about?

Aho: Okay. AWK is a programming language that was created by me, Brian Kernighan, and Peter Weinberger.

Hsu: And it's your three initials that are in.

Aho: Yes. I'm the A in AWK. Weinberger is the W in AWK and Kernighan is the K in AWK. Brian and I had been talking, we had offices next door to one another at Bell Labs for a long time, of maybe we could create a generalization of the grep program that would do more than just string pattern matching. That maybe we could have patterns that match numbers, arithmetic, strings, and Boolean combinations of these things and patterns, and maybe the action could be more than just printing the line that matches that pattern. We wanted to be able to do more sophisticated things. Our concept was let's create a pattern-action program. I liked the pattern-action paradigm of programming because human beings are pattern-action creatures. If we hear a loud noise, we look where the loud noise comes from. We're attracted by a pattern and then we want to take some action associated with this. If the lion is pouncing and growling, we definitely want to take an action after hearing that growl. This was the motivation for the design of AWK, that it would be a tool for solving routine data processing applications. I found myself teaching courses, so I had to keep track of student grades. I found myself having to manage budgets, so I had to keep track of budgets. I was an editor of a journal. I had to keep track of editorial correspondence. I wanted a little language where I could write one- or two-line programs to do the administrative data processing. Kernighan had the similar idea. Peter Weinberger was interested in databases at that time. He understood the importance of being able to do routine data processing easily and quickly. We created a language whose sole function was to write throwaway programs, one- or two-line programs to do useful, but routine data processing problems. This was the genesis of AWK. I was particularly interested in regular expression pattern matching at the time. Kernighan was interested in looking at things like associative arrays, generalizing the language and pattern-matching tools. We constructed a grammatical specification for this language that we were thinking of, and then showed it to Weinberger who was in our area, and in a weekend he created the first version of AWK.

Hsu: So he did the implementation.

Aho: Well, we had Lex and Yacc there. I actually wrote the pattern matching algorithms that are in AWK, and Kernighan decided that it needed more comments on it because you should comment your code, and his comment was, "Abandon hope, all ye who enter here." That's what he thought about my algorithm. He said it was too complicated for a mortal to understand. I disagreed with him, but it's basically the algorithm from egrep. I had written the code for egrep and fgrep, and it was in C.

Hsu: Okay. So you contributed that part of the code.

Aho: Yeah, our approach was divide and conquer; Weinberger's contribution was the runtime. Since we had Lex and Yacc programs for the syntax, all we had to do is run them through Lex and Yacc to create the frontend. When we created the first version of AWK, people started using it immediately. We thought that it was just a throwaway tool for us, nobody really would be interested in it. But it's amazing how much routine data processing there is in the world. In language design, what's perhaps just as important is what you leave out of the language as what you choose to include in the language. When we created the first version of AWK, we spent a lot of time debating what features AWK should have. The reason the language got to be known as AWK was because when our colleagues would see the three of us in one office or another, and when they'd walk past the open door, they'd say, AWK, AWK, AWK as they were going down the corridor. So we had no choice but to call it AWK because of the good-natured ribbing we got from our colleagues, and because at some Unix conference, they passed out t-shirts that had AWK, and the error message saying "bailing out on or near line five" on them. AWK got enshrined as a tool on Unix because it's part of the POSIX specification of Unix. So wherever you have a Unix system or Linux system these days, you get AWK with it. It's one of the most widely used tools on Unix and I still write AWK programs to do my routine data processing. I just recently was talking to somebody about the Russian peasant multiplication algorithm, and I wrote and sent them the five-line program to do this in AWK. I could have used Python but AWK is in my fingertips, so why not? So this is the birth of AWK. What astonished me was that I initially felt that nobody would write any big AWK programs. One day, I came into Bell Labs, and here was this guy, an engineer from the microelectronics unit of Bell Labs, who had written a several thousand line AWK program, and created a design automation system in it because he was doing digital circuit design. He found that creating a tool to take specifications of digital circuits and transform them into the components of the circuit was enormously productive. The reason he was in my office was that he had spent, I think, a week or two trying to debug his design automation tool, only to discover there was a bug in AWK. He accused me of wasting two weeks of his life by writing an incorrect processor. At that point, I decided, talking to Brian, that maybe we should practice good software engineering in the design of the translator. So Brian instituted a unit test for all of the syntactic constructs in AWK, and I got my students at Columbia, to follow this practice of one click, design-build-test. This is why they were able to design or deliver a working compiler at the end of the 15-week course because they delivered what was working at the end of the course. They had to write a language specification patterned after the appendix A of Kernighan and Ritchie's C language book for their language. They also had to deliver what was in the language specification. All this experience and the tools that I got experience with at Bell Labs got fed into the compiler design course. I could mention a couple of the languages that were designed in the course because I found that the most interesting part of my career at Columbia was teaching and working with the Columbia students. In this course, I had a student who had come to Columbia for her PhD, and she was interested in quantum computing. In her first year she took this course, and she and a small group of colleagues had designed a little language for learning about quantum computing. After taking the course, she said, "These programming language translators are really fascinating. Will you be my thesis advisor?" I said to her, "Krysta, I know nothing about quantum mechanics, but if you teach me quantum mechanics, I'll teach you about compilers, and then I'll be happy to be your thesis advisor." She did her PhD thesis on programming languages and tools for quantum computing, and in the process, taught me quantum mechanics. Through her, I said that if you're a compiler designer, not only do you have to understand the source language, you have to understand the target machine. I had her spend a year at MIT in Ike Chuang's lab learning about how quantum computers are really built, and at the end of this experience, we had this experience of creating an article on tools for quantum computing. It became a paper as part of her thesis. It also became a paper that was featured as the cover article of the Computer Magazine for IEEE. After she graduated, she went to Microsoft to work, and she is now the vice president in charge of quantum computing at Microsoft. She goes around talking about the benefits of quantum computing all over the world. She and her colleagues developed a language at Microsoft called Q# that is now part of the Microsoft Quantum Development Kit. You can experiment with quantum algorithms using Q# and you can then have them compiled into code that will run on a variety of elementary quantum computers that have been created by the community already. There are 15 different programming languages listed on Wikipedia’s quantum programming languages page, including Q#. That was an interesting experience and it also reinforces the comment that to be with it, it’s lifelong learning. There's nothing better than getting great people to teach you about their field. She taught me the little I know about quantum mechanics, but I know enough about how to represent quantum programs in a programming language to have had very fruitful collaboration with her on quantum programming design tools and programming languages.

Hsu: What’s her name again?

Aho: Krysta, K-R-Y-S-T-A, Svore, S-V-O-R-E. She's well known in the quantum computing programming circles, and you can see all sorts of talks by her at various conferences and fora on YouTube if you want to follow up on this. The other interesting language-- well, almost all of the languages far exceeded my expectations on what I thought students would produce. The other language that I thought was particularly noteworthy was a language called Upbeat, where the ‘b’ is a flat sign. We’re going back to music now. The Upbeat is a language for auralizing data or putting music to data. An Upbeat program will allow you to take a data stream, and it will then put sound to it of whatever kind you want. In my course, students had to give a demo of their programming language at the end of the course. For their demo the Upbeat team took the New York Stock Exchange ticker tape feed and set it to music. When the market was going up, the music was lively and bouncy. When the market was going down, it was gloomy and somber. The language guru on this team was a person by the name of Adrian Weller. He had been a stockbroker when he graduated from college, and so he had an experience of what stockbrokers do. He said that stockbrokers would much rather listen to music to see what direction the market is going in, rather than staring at the ticker tape all day. I thought this was a very useful and unusual language. Adrian is now a professor at Cambridge University. He just got the MBE award from Queen Elizabeth for his work. Being associated with talented students is not only good for one's career, but also a very satisfying and gratifying experience. I recommend that to people going into academia. Go to a place where you get exceptional students if you can and you can try to make the students even more exceptional in their areas with what you teach them. There was a third language that might be interesting. It was called What to Wear. With this language, you take the clothes that you have in your closet, put them in a database. You write a program that describes your fashion style, what kind of fashion image you want to exude, and you specify where or what conference you're going to. Then what the program does is make recommendations of what you should wear when you travel to that conference. There were two women on this team, because I can't imagine a team of five guys creating a language like this. But it's a great language if you're in the retail business because then you can make fashion recommendations out of your fashion offerings to prospective customers. Programming languages are being used for many more purposes other than just computing numbers these days and I'm sure you're well aware of that.

Hsu: Yeah. I had been wanting to ask this question before. Given how many programming languages we have today already, why is it necessary or interesting to create new ones?

Aho: Have you heard of the Tower of Babel in the Bible? It's human nature and it's been with us since humanity began. People like to be individuals in what they say, what they think, and how they communicate it. A long time ago, I wrote a paper for Science magazine entitled “Software and the Future of Programming Languages”. For the introduction of this paper, I tried to estimate are there more programming languages than natural languages in the world? You may have heard of a database called Ethnologue. It catalogs all of the extant natural languages in the world and I think there are about 7,000 or so spoken natural languages still in existence today. I once saw a website at a university in Australia that had pointers to about 7,000 programming languages. So maybe number of programming languages to the number of natural languages are roughly equal. But I think there are several hundred more commonly used programming languages. Remember, in my course, every semester, there'd be more than a hundred students in the class, assuming they're in teams of five. So 20 new programming languages got created and after 25 years, you'd have hundreds of new programming languages. Most of the languages were just created for the course. But some of them led to more interesting languages like Q#. So what do you want in a programming language? This gets down to this issue of how do you communicate. What's the purpose of a programming language? It’s to communicate an algorithm to a computer and the compiler translates the algorithm from the source language into a semantically equivalent algorithm in the target language. The target language, by the way, could be how your brain works, but let's just deal with the silicon computers. People get passionate about their programming languages. If you want to create World War III amongst a group of programmers, just ask them, “What's the world's best programming language?” That's almost akin to asking, “What's the world's best natural language?” One that you probably speak, and you can't think of a better language than what you speak. Of course, a lot of abstractions have gone into the creation of programming languages and now we've entered the world of distributed computing. Not just computing a function. We deal with distributed software systems that maintain an ongoing interaction with their environment. Things like the internet. Things like an operating system. Things like the human brain. So if you want to describe an ongoing interaction with a distributed system, what kind of formalism should you use to describe this ongoing interaction? What branch of mathematics is well suited for doing this? Or it doesn't have to be mathematics, but you'd like to have a precise specification because if you're dealing with the internet, you need precise specifications of the protocols that are used in the internet, in order to achieve interoperability. You want a global information infrastructure that operates seamlessly. Many people have strong feelings about what a protocol should be and how it should be designed. Many people have strong feelings of how arithmetic should be specified and how much mathematics should you put into your programming language. What type system should you use? Should every programming language know about complex numbers? How about quaternions? It's so much fun designing a programming language but it's awfully hard to design one that is elegant, simple, and gets widely adopted. One of the things I noted with C is that C has been one of the most popular programming languages in the world since its inception and I attribute that to the good taste of Dennis Ritchie. Good taste -- I don't know -- can that be taught? It's a question that intrigues me, and certainly, I learned an awful lot about language design by interacting with Brian Kernighan and Peter Weinberger in the design of AWK. Kernighan has a number of very popular little languages to his credit. I mentioned EQN for specifying mathematics. He has created a language for mathematical programming called AMPL. He also created a language called Ratfor [Rational Fortran] and I think he agrees with the principle that languages should be simple, elegant, and easy to teach. Well, one of the things I did in my compilers course was that, in addition to creating a new programming language, the students had to write a user manual for their programming language before they implemented it. Because what good is a programming language if you can't teach it to others? You learn so much about how to get rid of the crud from your language when you have to explain to others how to program in it. There are three skills that I thought my course instilled on students. One was communication, both oral and written. You should be able to talk about what you're doing in ways that other people can understand you. You should be able to write about what you're trying to do in ways that people can understand you. A second important skill is project management. That's why I had five people on a programming team. Managing one person is tough. Managing two is a little harder. Managing five -- you have to start thinking about serious project management. How do you get five people to implement something that's changing as it's being developed? This is the epitome of collaborative software design of evolving systems. How do you keep everybody on the team abreast of what you're doing and what kind of process do you use to ensure that it gets correctly produced, that the output is what you had in mind at the beginning? That was the function of the language guru on each team. The third life skill is teamwork. How do you work with other people? I'd assign a TA to each team and I'd TA a few teams myself, just to make sure there wasn't a meltdown on the team. One of the comments that I often heard from talking to the individual team workers is, "I really can't work with these turkeys." These are Columbia students, and I said to them, "You haven't seen turkeys. Wait until you get out into the real world. There you'll meet some real turkeys. This is just practice." Teamwork is a very important skill, especially in this world, where you deal with systems that are so large that they can't be created by a single individual anymore. Computer science has got all sorts of interesting technical problems. It's also got interesting people problems and it's got interesting philosophical problems like are human beings just computers? The Turing test or there's the Church-Turing thesis that states anything that can be computed can be computed by a Turing machine. If we're just biological computers, can we be simulated by a Turing machine? Can emotion be simulated by a Turing machine? This gets down to the monsters from the Id. Can that be simulated by a Turing machine, and are computers the salvation of humanity, or the destruction of humanity? A lot of science fiction movies have played on this theme. I think we need good leadership through this phase of human development for humanity to survive.

Hsu: Yeah. Could you talk a little bit more about being part of that Unix group in the early days and the culture of the group?

Aho: Oh, it was nirvana. Here you're given a job where you could do whatever you thought was important. You could work with the brightest people in the field. They take an interest in what you're doing. When I created egrep and fgrep, people used the tools immediately, and there's nothing more satisfying than somebody paying attention to what you're doing. Even the president of Bell Labs said algorithms are important. Has the president of your company said, go teach it to everyone else? I thought that was a very heady experience. What was also interesting about the Unix group was that there was an expectation of you learning from others. The original Unix operating system was very open, which if kept the way it was, would not be suitable in this world environment. But I learned so much by being able to read other people's code and by being able to use other people's programs. I mentioned being a very early user of the typesetting programs to be able to write my papers. The other aspect of the Unix group or the Computing Sciences Research Center is every week we had a seminar that some member of the center had to give. It was the most terrifying event of working at Bell Labs. Because once a year, you had to get up in front of your peers and explain to them what you're doing, and these people did not suffer fools very graciously. So I would have to say, “What can I do to impress Ken Thompson?” Screw impressing my boss. Impressing my peers was much more important to me because I wanted to be considered one of the guys and to get into the Unix room. Ken Thompson had his desk right at the entrance, and I know people who felt intimidated that they'd have to pass by Ken Thompson to walk into the Unix room. Finally, I got used to it, and they more or less accepted me in the Unix room. There was a certain culture, but the culture was one of excellence. One of caring for what people did and it was one of conviviality. Have you ever seen a punch card?

Hsu: Mm-hm.

Aho: Have you ever created a punch card on a key punch machine?

Hsu: Yes, at the museum.

Aho: In the early days, that was the way we created input to the very first computers we had. We had these key punch machines in the computer room, and with the key punch machines, depending on what kind of pattern you put into the punches you put on the punch cards, you could develop a certain resonance of the key punch machine, so you could get them to start walking across the room. We had contests to see who could make our key punch machine walk the fastest across the room. This was not necessarily viewed very favorably with management. But on the other hand, it shows what some of the things very bright people, creative people do when they get tired of doing what they're supposed to be doing. I don't think I should go into some of the other stories of what we did, but there's a lot of conviviality and camaraderie amongst the people in the Unix team.

Hsu: Earlier, you mentioned some other collaborations with some of the other team members. Would you mind discussing a few of them?

Aho: Well, I mentioned I was in a seminar where I was doodling on some recurrence relations and solutions to the recurrence relations. Sitting next to me was Neil Sloane, who had written the bible called, "The Handbook of Integer Recurrences." He was looking at my recurrence relation. I was looking at how fast a particular parsing algorithm worked, and then when he saw the recurrence relation, he took the sheet of paper out from under my pencil and in a loud voice said, "Where did you get this recurrence," because this was a recurrence that wasn't in his book at the time. And he was writing THE handbook of integer recurrences. This recurrence was just of the form x sub n is equal to x sub n minus 1 squared plus some function of the previous elements of the recurrence. Amongst this class of recurrences, you could count the number of binary trees of height n, and that comes up in parsing theory. What we did was we generalized it to a very general class of recurrences and showed that all the solutions to this generalized class of recurrences were doubly exponential. If you want to count the number of Boolean functions on n variables, there are 2 to the 2 to the n Boolean functions on n variables. They're doubly exponential. We had the solution to our generalized recurrences by four o'clock in the afternoon. He wanted to make sure he had the literature search right, so, he wanted to go to the Brown library and look at some of the original papers by some of the mathematical greats like Euler or Leibniz or Gauss, to make sure that he had the historical treatment of the evolution of this class of recurrences correct. By the end of the week, we had the paper written and submitted to the Fibonacci Quarterly. That was an experience that I probably could not have had anywhere. The other experience of this nature was working on database systems. There was a lot of interest in database systems in the '70s. You may have heard of relational databases and languages like SQL. It was Jeff Ullman who was really spearheading this effort into database systems and database queries. We did some work with some students that were at Princeton at the time like Catriel Beeri and wrote some fundamental papers on the complexity of database queries and how to optimize database queries. I got to meet some of the people like Ted Codd, who founded relational database theory, in some of the seminars that we had orchestrated. This brought people at Bell Labs into the world of relational databases and database queries, another way of keeping up with the times and learning new fields. I sort of categorize my technical interests in the '70s as algorithms and compilers. The '80s were the beginning of databases. In the 2000s, when I got to Columbia, quantum computing, particularly with Krysta Svore and some other students. I did a brief digression into computational thinking. I don't know whether you've heard of that term "computational thinking." In the 1990s, I wrote two books with Jeff Ullman on the foundations of computer science. They're somewhere up there, the “Foundations of Computer Science." They have a turtle on top of elephants on the front cover. Jeff liked making fanciful front covers. There's this phrase, "It's turtles all the way down." On top of the turtle were things like a baseball player, a teddy bear, and so on. In the book, the first chapter is entitled "Computer Science: The Mechanization of Abstraction." I created this title because I think of computer science as the mechanization of abstraction. Abstractions are important in any field. In 2006, Jeannette Wing wrote a paper for the Communications of the ACM entitled "Computational Thinking." It became very influential in suggesting that computational thinking should become part of every school kid's repertoire of tools for approaching problems. My thought on computational thinking was that too, and I noticed as I was going through school, that if you have the right way of thinking about a problem, some ways of thinking about a problem make the solution appear very elementary and very simple. My idea of computational thinking is you have to come up with the right abstractions and the right computational model for a problem so you can then create the solutions out of computational steps and algorithms. This is my view of computational thinking: what are the right abstractions that we should be using to think about problems. First of all, thinking of the right problem is more than half the battle, but coming up with the right abstraction is then a very important part of going to a solution. If we look at Unix, Ken Thompson had this abstraction that everything should be a file, and the file should be the unit of interoperability amongst the programs. That gave enormous leverage in the design of the software that sits in an operating system. If you looked at some of the IBM systems in the past, every program had its own file format, and it was a nightmare to get the output of one program to be adopted as the input to another program. This was another motivation for us creating AWK: we wanted a tool that could be used to transform the output of one program very easily so that it could be fed into another program. Doug McIlroy had come up the idea of pipes long before as a way of connecting programs together. He thought that programs should be connectible just like you can connect pieces of garden hoses together, screwing in one piece of hose into another. That gives enormous leverage in piecing together programs. One famous example of this I used when I talked about computer science and algorithms to a high school math club, was making a talking desk calculator. We had a program that behaved as a desk calculator. It would add, subtract, multiply numbers, arbitrary precision numbers. They could be arbitrarily long integers if you like. There was a program that would translate numbers into words. So, 1 would become O-N-E. There was a program called Speak that would translate words into phonemes, spoken phonemes. So, all you had to do to get a talking desk calculator is take the desk calculator program, pipe its output into words, pipe its output into Speak, and voila, you have a speaking desk calculator. All you had to do was compose three existing programs together. Think of how hard this would be if you had to do it from scratch. Having the right abstractions can greatly facilitate the design of programs and systems. I think finding the right abstractions is an essential component of solving problems, and that's why I'm a big fan of computational thinking.

Hsu: You left Bell Labs and joined Columbia in 1995, but then you returned to Bell Labs in 1997, and then you were there until 2002.

Aho: I was on a leave of absence.

Hsu: Ah, right.

Aho: I asked Columbia for a one year leave of absence. I was having such a good time, and then wrote Columbia and said, "Can I extend that to two years?" Iterated this. When it came up to the end of the fifth year, the provost said, "Are you coming back, because if you're not coming back, we'll get rid of your position." I said, "No, no, I'm coming back," because I still had this dream of wanting to teach and work with students. So, I came back. When I was at Bell Labs on that leave of absence, I was part of the senior management of research. I was an assistant vice president for research and the vice president of the Computing Sciences Research Center. Bell Labs had the good philosophy of appointing as research managers people who had done significant research, so they'd understand what it takes to do research. This is a very good philosophy for the people who are actually doing the research but a very bad philosophy for the people who are the research managers because then, all of a sudden, you get into management questions, and that takes away time from doing your first love, pure research. That was another inducement to go back to Columbia where I could do research, work with students, and teach. One of the experiences that I had when I had my stint at Bellcore, between Bell Labs and Columbia was to learn how software was done in practice. A good friend of mine was the vice president in charge of software development at Bellcore. So, I thought if I worked at Bellcore and watched how he managed software development, that would allow me to better teach software engineering at Columbia. I have a strong feeling that people who teach should know what they're talking about. That's not often true, unfortunately, but it was a good experience for me. It also facilitated why the students in my programming languages and compilers course could finish building their compiler in 15 weeks. When the students graduated from Columbia, the job interviewers asked them about what courses they had taken. The students would talk about the programming languages and compilers course. Often the interviewers would say, "Gee, I wish our software developers would do software that well," because there is a history of software being buggy and not delivered on time. I did not want to teach such bad habits. But I learned a lot of things about the software industry through being in senior management. Also, I was on a number of external committees for the government for NSF and for the national academies. On one occasion I had been invited to chair an evaluation committee for computer science research and teaching in Finland, which was an interesting experience in international relations.

Hsu: Right. Can you talk quickly about your decision to leave Bell in the first place and go to Columbia?

Aho: I think everybody should have a life plan, and my life plan was do research at Bell Labs and then go teach and work with students for the last third of my career. So, that's part of my life plan.

Hsu: Yeah, you could have gone anywhere, though. How did you end up at Columbia?

Aho: Do you want me to be honest?

Hsu: <laughs>

Aho: The previous chair, Joe Traub, had set up computer science at Carnegie Mellon University and done a very good job of it. Columbia hired him to set up the computer science department at Columbia, and he recognized the importance of getting good people. He had also worked at Bell Labs at one time, and he knew me. So, he got me to serve on his search committee for the next chair of the computer science department. We went through candidate after candidate after candidate. He kept saying, "Oh, Al, you're so much better than these candidates." After saying this often enough, he persuaded me to enter Columbia as chair of the computer science department, which is a suboptimal way to enter academia, but I said let me do my public service. I was managing an organization of about 200 people at Bellcore. How hard it could be to manage 25 people at academia? The answer is much harder.

Hsu: <laughs>

Aho: But I'm very happy to say that computer science has done quite well at Columbia. It's gone way up in the rankings from when I started, and it's got some of the most exceptional people in the field at this point. What I learned at Bell Labs is that if you want to be the best research organization in the world, you better hire the best researchers in the world. So, my philosophy was always to go after the best students that were graduating. Hiring the very best is a challenge. You have to nurture the candidates long before they graduate because they're spoken for well before they graduate.

Hsu: So, when you went back to Bell Labs in '97, it was part of Lucent at the time?

Aho: Yes. As you know, AT&T was like a mothership that kept spawning organizations over the passage of time. It had a microelectronics manufacturing division, Western Electric. That was spun out of AT&T, and there was a divestiture in I think it was '84 where AT&T kept the long distance business. It spun out the local businesses through the regional Bell operating companies, and it spun out the manufacturing arm. Bellcore was created as the research arm for the local operating companies. It was split out of Bell Labs Research and other parts of Bell Labs so that it would have the technical base to provide the advice and guidance to the local telephone companies. So, it had some very good people. My good friend, Bob Martin, had gone off to Bellcore. He knew that I was interested in going to academia sometime and also interested in teaching software engineering, so he recruited me from Bell Labs to go to Bellcore. Maybe I shouldn't say this. It wasn’t as hard leaving Bellcore to go to Columbia as it was leaving Bell Labs because I already made one major transition in my life. And it came at the right time because Bellcore was being sold off to SAIC, and I'd also discovered what venture capital can do to an organization. You see this in Silicon Valley all the time. The second owner of a startup company isn’t necessarily like the first owner. Many of the people who set up startup companies are not capable of managing more established companies.

Hsu: Yeah.

Aho: You can provide your own examples of this.

Hsu: Yeah. Could you talk about labs like Bell Labs and Xerox PARC don't-- I mean I guess even if they formally exist, they're not as dominant as-- in terms of like turning out the industry changing innovations the way that they used to. Could you speak to maybe why that is or what's happened to these types of corporate labs?

Aho: Well, an enormous amount has been written about these labs. Bell Labs was created I think in 1925, formally. It was split off part of Western Electric Engineering. The leaders of AT&T had this idea that in order to have universal service, that was their credo, we don’t understand enough of the technology to create reliable, efficient universal service. The charter for the Bell Labs was reliable efficient universal service across the United States, and around the world eventually. So, they created this organization that had this long term view and stable funding, and the people who started Bell Labs also realized that they needed scientists who could create the technology to support this vision of universal service. A lot of the work allowed the researchers to spend some time thinking about what kind of technology they should create and develop. It wasn’t what can impact the next quarter's financial statement. It's what kind of technology can we develop in a period of years to improve things. It was that backdrop that created an atmosphere for creating impactful, high impact, scientific and technological innovation. Maybe one of the most significant things that came out of Bell Labs was the transistor. The transistor was a cost reduction for the electronic equipment that was being used to run the telephone industry, but, on the other hand, it created the global information infrastructure or facilitated the creation of the global information infrastructure. UNIX revolutionized operating systems. It was a follow-on to the Multics project, the purpose of which was to help develop the operation support systems for the telephone industry. Well, it turned out that UNIX did a lot more. Fundamental inventions can be used for many more purposes than what originally people thought that they were for or the funding was for. Did ARPA know that ARPAnet would become the Internet? They say yes now, but having worked with some of the predecessors of the Internet like NSFnet, nobody realized the wealth of information and technology that's present on the Internet today and how it's changing the world and has changed the world. Essays are being written on whether it's for good or for ill. So, good people, long term vision, stable funding are essential. Also being in the right area at the right time is, at least from my personal perspective, a very good thing because it's much easier to innovate and solve important problems as the field is developing rather than in a field that's several hundred years old. If we take number theory in mathematics, you probably have to spend several years of hard studying to get to the research frontiers in order to make a contribution. Then when you do make a contribution, maybe there'll be three or four people in the world who'll understand the proof of your theorem. In the early days of computer science, it was very easy to innovate and come up with new algorithms for important problems. Some of these algorithms are still used. So, having the right timing is also good. In many ways, I think that maybe understanding how the human body processes information is ripe for innovation and scientific improvement, but the problems are really hard. How does the brain implement algorithms to do what it does? What is consciousness? Do you have a good definition for consciousness?

Hsu: I don't, but there are ones out there.

Aho: Uh uh. The medical profession does not have a good definition for consciousness or for thinking, a good definition in the same sense that mathematics, physics, and computer science demand good definitions. Do you know a good definition for love? Hate?

Hsu: I mean I would wonder is everything-- is it possible even to strictly define everything?

Aho: Well, if you can't define it, you don’t know it. Your concept of love may be very different than mine.

Hsu: <laughs> Yeah. That goes back to the communication question--

Aho: Exactly. You know the physicists say if you can't measure it, it's not science.

Hsu: Right. Okay, I'm going to start wrapping up.

Aho: Please do.

Hsu: Last few questions. Can you talk about receiving the Turing Award and what it has meant to you?

Aho: I don’t deserve it. Why I got it is still a mystery to me because I don’t think I've done anything that is really that important. But, I'm happy to have received it. Some people say I’ve had a big impact, or at least the work that I've done, I think particularly in the area of algorithms and algorithm design techniques--when I wrote *The Design and Analysis of Computer Algorithms* book. My favorite chapter that I wrote was "Algorithm Design Techniques." My boss, Doug McIlroy, said, "This is a great contribution because, at that point, people have created good algorithms for specific problems like the fast Fourier transform or sorting, but here, you've created techniques that apply to a broad class of problems so that, when you're exploring a new area, you can immediately ask such questions as what are the brute force methods, does dynamic programming apply, can you use hill climbing techniques, and so on." You have these general classes of techniques that can immediately be used to try to find that algorithm that gives the improvement that you're looking for. Maybe you can get a Turing Award or a Nobel Prize for the algorithm design techniques used by whoever created humans because we have some pretty good algorithm design techniques. What gives people creativity? Take evolution, I think of that as a magnificent algorithm. Who thought of evolution? We ramp up the food chain and then ultimately come up to questions of who created God. I don’t have an answer to this, but there are some truly mysterious open problems confronting humankind, philosophical problems. I think the philosophers have recognized this many millennia ago because the questions that they look at confront what is reality, what's the essence of being. Will we ever have precise answers for these? Certainly, not in my lifetime, maybe not even in yours. But you could certainly make some award-winning contributions by coming up with measurable definitions or measurable metrics for some of these big ticket questions.

Hsu: Thank you. And last question, what advice would you give to a young person starting out in your field today?

Aho: An excellent question. My advice is you can never learn too much mathematics. Mathematics is going to be important. Learn the fundamentals of whatever field you're in because the fundamentals don’t go out of style nearly as quickly as the technology of the nonce. When you graduate, try to get a job working with the best people in the field because you learn so much from these people. Then you'll be well-positioned for your next job. So, learn mathematics, learn fundamentals, try to work with the best people in the field, and write a book. I maybe should add that about what you're doing, have you thought of writing a book on interviewing people effectively, efficiently?

Hsu: Not yet.

Aho: Not yet. That's an excellent answer, but you might want to think about it. I noticed in the New York Times today, there's a person who is advising the government on cybersecurity who is a journalist, and one of the things that she's saying is that the government needs to attract the best people, technical people, in the field to combat the cyberterrorists. I think that's an important observation on how should the government go about being able to do this. There was also another article in today's New York Times about the love guru. I haven’t finished reading this, but she runs an advisory service for rich clients on how to find dates. She's got a methodology on what you should do to find that one. Interesting areas of opportunity, and whether computer science or computational thinking works for these remains to be seen, but I'm looking at where some of the targets of opportunity for new people are. Notice that a lot of these areas involve how does technology interact with people and society. One of the most interesting books or journals that I recently read was "AI and Society," the latest journal of the American Academy of Arts and Sciences. It has 25 articles talking about AI and its interactions with society and asking such questions as how should we regulate AI, who should regulate AI, ethics for AI. I'm astonished at what these large language models like GPT-3 are now capable of. I don’t know whether you've read any of the stories that it generates. The last article in this journal was giving GPT-3 the questions on an Oxford final exam and looking at the essays it wrote as answers. Is the Turing test a good test for artificial intelligence? I was amazed at the erudition of the answer and the reasonableness of the answer for comparing two philosophies. Finally, did you see that somebody at Google has been put on administrative leave for saying that their latest large language system exhibits sentience?

Hsu: Yeah. I saw that.

Aho: So, do you think we'll ever see sentience in AI? You've interviewed a lot of interesting people.

Hsu: Ever? I mean forever, that's a long time. In my lifetime, I don’t know.

Aho: Yeah. These are some of the questions that the current technology engenders and maybe with more pungency than before. A question I asked yesterday since I interact with a bunch of lawyers in a discussion group, should the law sanction marriage between a human being and an AI. It's pride month, so let's just extend it a little. What do you think?

Hsu: Well, that depends on if an AI is considered equivalent to a human in certain ethical terms.

Aho: The law has nothing to do with ethics. It's whatever the politicians decide it should be.

Hsu: That's true.

Aho: If we got a bunch of computer scientists in Congress, we could pass such a law and then amend the Constitution. By the way, this question was asked in a PhD thesis that I read many decades ago coming out of Denmark saying if marriage gets sanctioned between humans and an AI, Denmark will be the first country to do it. He was Danish.

Hsu: My last question actually is-- we were just talking about AI and ethics and AI and society. There's been a lot of-- the word algorithm has kind of taken on a negative connotation in the last couple of years.

Aho: That's because they don’t know the definition.

Hsu: Right.

Aho: So, what is an algorithm?

Hsu: You mentioned earlier it's a recipe.

Aho: Precise definition: A Turing machine that halts on all inputs.

Hsu: Right.

Aho: So, that has negative connotations? It's a mathematical definition. No emotion, just mathematics.

Hsu: Right, so that really comes down to when people are saying the world algorithm--

Aho: Why don’t they say this about a recipe?

Hsu: Yeah, well that goes back to the communication issue.

Aho: Yeah, exactly.

Hsu: We're defining the words in two different ways.

Aho: You made my point eloquently.

Hsu: <laughs>

Aho: At the University of Helsinki, when you get a PhD, you have to buy a sword to defend truth and knowledge. So, they gave me this sword for my honorary degree. I didn't have to buy it. At the end of the graduation ceremony, the PhDs form an arch of swords, and then the master's students go walking under the arch, an impressive ceremony. They don't have such impressive ceremonies at American universities for the PhD. The Europeans are much more flamboyant in their awards. Their costumes tend to be much more colorful and so on. Although, American universities have pretty interesting graduation gowns these days, too. But that's been one of the more interesting trinkets that I've gotten.

END OF THE INTERVIEW