A. M. Turing Award Oral History Interview with Richard Karp

by Christos Papadimitriou

April 23, 2012

Berkeley, California

Papadimitriou: Dick, did it all start?

Karp: I grew up in a suburb of Boston, actually close to the center of the city. My father was a middle school math teacher. My mother was a high school graduate, but later went back and earned a degree at Harvard through extension courses. Proudest moment of her life was when, at age 57, she stepped up to receive her diploma at Harvard. Education was very much paramount in my parents' worldview. There were four of us, I'm the oldest of four, and they spaced us out at intervals of a college career so that they could have one kid in college at a time.

I think the greatest admiration I felt for my father was when I visited his class in the middle school where he taught, and he was very much in command in the classroom, teaching math, and he could draw near-perfect circles freehand, which I thought was extremely impressive.

Papadimitriou: Funny. My father was also a middle school math teacher. Did you end up talking with him about math?

Karp: We did. But he really didn't have very advanced mathematical knowledge. He was pretty much limited to what he was teaching. But his presence and sense of command in the classroom was something that I wanted to emulate, and I think that it's not an accident that I went into teaching eventually. I always was good at mental arithmetic somehow. That's not math, as I explain to my friends. But I was able to multiply three- or even four-digit numbers in my head to amuse my friends.

Papadimitriou: Math helps, but not much.

Karp: It doesn't help much. I didn't have any special tricks. But I really understood what math was about only in the tenth grade when I took Euclidean geometry and I fell in love with the subject. It was so pure, so pristine, so clean, and so amazing that you could prove things through pure logical reasoning. I used to pretend to be sick so that I could stay home and solve geometry problems. [laughs]

The other subject that I particularly liked in high school was Latin, which again I think had a nice logical structure that I enjoyed.

I was always terrible at anything that involved mechanical aptitude. Even today, I have to lean on friends to figure out how to install a piece of software or how to screw in a light bulb and so forth. [laughs]

I had a young aunt, only a few years old than I, who taught me to read, so I read quite early. For that reason I skipped a grade, and so I ended up being a year and a half younger than my classmates, which probably affected my social development. I think I was a bit of an introvert, and I clearly was behind my classmates in things like art and carpentry and the like. That feeling of weakness in that area has stuck with me throughout my entire life. Many years later I got an honorary degree at the University of Pennsylvania, and the person who introduced me at that event had been in a lab with me at Harvard, an electronics lab, where I was the klutziest fellow in the whole class. So he was totally astonished that anybody like me could possibly get an honorary degree from any institution.

It was geometry that turned me on. I did well in high school in mathematics, and didn't get so terribly advanced though. My high school didn't offer calculus and I had to wait until college to take calculus.

Papadimitriou: Harvard was an obvious top choice I guess.

Karp: My parents directed me to go to Harvard. They directed me also to apply to MIT as a backup, but it was clear that they wanted me to go to Harvard. I think it was a good choice. The only thing that was problematic was that there were a lot of very bright kids there, and particularly…

Papadimitriou: That's a problem, yeah.

Karp: That's a problem. [laughs] It's nice, but it's also a problem if you aren't quite so sure of yourself. And in particular, there were some math prodigies who were way ahead of me. I didn't realize that one of them was going to win the Fields Medal, David Mumford, and one of them was going to win a Nobel Prize, Ken Wilson. I just knew that they were a lot better than I was.

Papadimitriou: That must have made you feel better, but too late.

Karp: [laughs] Too late.

I also found that it was very difficult for me to get excited about a purely axiomatic subject if I couldn't relate it to anything else. For example, I was very interested in game theory because I saw that it related to situations of conflict. But when I was

playing around with let's say the axioms of topology, manipulating the axioms just was not something that I enjoyed. I needed a reference point.

Papadimitriou: Not necessarily outside math? Even inside math it would suffice?

Karp: Yeah, even inside. Like I didn't have any trouble appreciating abstract algebra, because they were the specific instances of the general axiomatic systems. Even though I enjoy math for its beauty, I think I always feel the need for a reference point, which could be an application in the world or unification of two subjects. Something beyond the stripped-down formal content.

It was really in my senior year that I encountered people like Mumford and Ken Wilson.

Papadimitriou: These are colleagues…?

Karp: They were actually younger than I.

Papadimitriou: Younger than you. I see.

Karp: A bit younger, but more advanced in mathematics. I really was reluctant to pursue a career that would have led to an academic position as a math department member. There were no computer science departments at that time.

Papadimitriou: True.

Karp: Toward the end of my undergraduate career, I took courses at the Harvard Computation Lab, which had a more pragmatic orientation – numerical analysis and the like. I also took some courses in probability and statistics, and I found that I seemed to have a particular affinity for those subjects. In particular, Professor Hartley Rogers who taught the probability course gave me a great deal of encouragement, because I did well in his course, and that to some extent mitigated my feeling that had been arising from my comparison of myself with these advanced young mathematicians.

So I decided to stay on at Harvard and work in the Computation Lab. It was actually my mother who said to me, "Young man, go into data processing." [laughs]

Papadimitriou: Is that right?

Karp: She would have liked me to be a nuclear physicist, but I didn't really have much of a leaning towards physics. I found that instruction in physics that I received was not very satisfying. It was too non-rigorous I think. So I didn't want to be a physicist. I wanted to do something with math. My mother told me that

data processing was the key to the future, I had enjoyed my applied math courses in the Computation Lab, so I stayed on there for the PhD. And gradually I began to develop a sense of my own powers, that I really could do something.

Papadimitriou: But computation was not something that fascinated you at first sight?

Karp: Even from the early days when I was multiplying numbers in my head, algorithmic processes really interested me. I remember there's a famous algorithm called the Hungarian algorithm for the so-called matching problem, which I learned about. And the way it sort of proceeds inexorably toward the solution very beautifully doing nothing but addition and subtraction operations and comparisons appealed to me very much. I enjoyed other aspects of mathematics, but less so. I really had more of a feeling for discrete mathematics and constructed mathematics, algorithmic mathematics. I know that Don Knuth has expressed – a famous computer scientist, maybe the most famous – has expressed similar feelings about his own mathematical leanings. So I was lucky to come along at a time when discrete mathematics was something that one could make his specialty.

Over the course of my graduate career, I began to get a stronger grip. I began to feel that I had some capabilities. I was successful in some summer jobs at outside laboratories and I found that I seemed to have a knack for lecturing. I had a few occasions to give seminar talks and to fill in occasionally for an instructor, and I discovered that the process of putting a lecture together and delivering it really appealed to me very much.

Papadimitriou: But you didn't go into academia. *[0:10:00]*

Karp: One couldn't in those days. My interests were not along the main lines of a mathematics department, and there were no computer science departments.

Papadimitriou: Nowadays, mathematics departments do hire computer scientists, but at that time it was probably unheard of.

Karp: Well, the term "computer science" was coined after I got my PhD as far as I know. I'm not sure, maybe around the same time. I began reading the meagre literature that existed on combinatorial mathematics and theory of computation. There wasn't much. There was the graph theory book by Kőnig, there was… I think Berge's book had come out by that time. A few other references. Kleene's book on *Introduction to Metamathematics*. But there wasn't really very much to dig one's teeth into at the time.

So I did a PhD dissertation that was concerned with the application of fairly elementary graph theory to the analysis of computer programs, figuring out which variables could be deleted and which variables could share a storage location if

they didn't conflict. Things like that that became something of a fashion in computer science later. I anticipated some of that. That was my thesis.

But at the time I got my PhD, I didn't really feel that I had a truly advanced set of tools in mathematics. I was lucky enough to get a wonderful job at IBM, the IBM Research Center, the Watson Research Center outside of New York City. And I consider the nine years that I spent at IBM really a continuation of my education. I was lucky to have some wonderful mentors. Upon graduating, I had a number of offers from different branches of IBM, and it ended up being a choice between joining the Mathematical Sciences Department at the Yorktown Heights lab where in fact I did go, or being part of the team that was developing an advanced computer system in Poughkeepsie, the so-called Stretch computer. The same team eventually was responsible for developing the System/360 computers. And I was tempted, but I sort of had a gut feeling that I would do better and have more fun doing mathematical work, so I gave up the opportunity to the design operator console of the Stretch computer and joined instead a group that was doing logical design of digital circuits, what we call switching theory.

I felt that I was able to penetrate into that field, had wonderful mentors and wonderful opportunities to expand my knowledge at IBM. I got there in 1959, and in 1960, my boss, a man named Paul Roth, had orchestrated a wonderful series extending over several weeks in which many prominent discrete mathematicians came together. These were the people who had really developed discrete optimization, network flow theory, linear programming – George Dantzig, Ray Fulkerson, Ralph Gomory, and others. So I was quite excited to meet these famous people from the RAND Corporation, Princeton University, and the other centers where this kind of work was going on.

A very influential figure for me was Alan Hoffman, who joined IBM a little after I did but is about 10 years older than I am. Alan had been one of the leading figures in the development of linear programming and combinatorial optimization up to that point.

Papadimitriou: He came from RAND?

Karp: No. He had been at some center at UCLA where Derrick Lehmer was also operating, as I recall. Alan was known among other things for his example of the phenomenon that the famous simplex method of linear programming could cycle, could get into a loop. Although it never happened in practice, he constructed an artificial example. So he had some interest in algorithms, but basically he was more interested in mathematical structures from a non-algorithmic point of view. So he and I had very different styles, but he became a model for me in terms of the elegance of his abilities at mathematical exposition and also certain bodies of knowledge. He taught me what he knew about linear programming and network flows, flow of commodities through networks, and a little bit about eigenvalues of graphs and that sort of thing. So basic discrete mathematics.

He also used to set me up with dates with girls who were friends of his wife or his family. So he was an all-purpose mentor.

The period at IBM was extremely enjoyable. It was really a dream job. Every now and then, I was asked to do something for the company, which I was willing to do of course. But 90% of the time, I was on my own to work with colleagues at the lab, and they were very good people.

There were a couple of things I worked on that seemed to have had an impact from that period. One of them was some work on the traveling salesman problem, the famous problem where you have a salesman who knows the distances between every pair of cities in his territory and he wants to make a tour of his territory while minimizing the total distance traveled. With a colleague named Michael Held, we developed some algorithmic approaches that for a time made us the world-champion solvers of traveling salesman problems. We lost that position very soon, and by now of course we've been left in the dust by much more sophisticated methods. Some of the tools that we developed, what's called a lower bound on the cost of the optimal tour, is still investigated.

Papadimitriou: It's still the champion of lower bound?

Karp: Yeah, I suppose so. Yes. I also worked with a long-term colleague, Ray Miller, who was at the IBM Research Lab with me, and we got interested in parallel computation, partly motivated by the desire of IBM to develop some algorithms that could be submitted to the Patent Office so that IBM would have some standing in the area of patented algorithms in case that became a competitive issue. So we developed some formal models that we could use to develop parallel algorithms. In particular, we created an abstract model of computation that could be asynchronous, meaning that it wasn't entirely predictable whether one step would precede another. And we…

Papadimitriou: Vector addition systems?

Karp: Yes. And that led to a nice mathematical formalism called vector addition systems.

Vector addition system was a kind of rule for generating a subset of the lattice points in a *d*-dimensional space. And the decision question that we studied was whether a particular vector addition system generated an infinite set of reachable points so-called or just a finite set.

Papadimitriou: The boundedness thing.

Karp: The boundedness question. The interesting thing was that Ray and I developed, showed that it was algorithmically solvable. Later, other investigators

showed that while it was algorithmically solvable, the best algorithm for it would inevitably require an astoundingly high number of computation steps. This was something that we hadn't really thought about. It was indicative of the level of development of the field, that nowadays you always ask for…

Papadimitriou: Automatically.

Karp: …automatically, "What is the time bound?" But there, at that point, we satisfied ourselves just showing that there was a finite, a terminating algorithm for the problem.

Papadimitriou: And that was what was known also for much easier problems like linear programming, max flow, and so on. People seemed to be content that the algorithm was finite.

Karp: That's right.

There's a fundamental algorithm that was… it's called Ford–Fulkerson algorithm. It solves the following problem. You have a network, and the edges of the network are like channels for sending some commodity. It could be oil, bits of information, water, whatever. And the question is, given the structure of the network and the capacities of these channels or edges, exactly at what rate can you pump the commodity from a given source to a given destination? It's called the max flow problem. There's a beautiful theorem called the max-flow min-cut theorem which characterizes it. *[0:20:00]* It's related to linear programming. And there was a very nice algorithm called the Ford–Fulkerson algorithm for solving this problem. The Ford–Fulkerson algorithm could run for a very long time. In fact, if you had irrational capacities, it could even be non-terminating.

Papadimitriou: Fulkerson knew that, actually.

Karp: Fulkerson did know that. It didn't occur to them to derive a variant of the algorithm that would have a nice upper bound on the number of computation steps. So I discovered two variants. One of them worked in what we call strongly polynomial time. The other worked in a running time that depended on the number of bits in the data.

So there were two basically efficient algorithms whose efficiency we could prove. And it turned out that a brilliant young mathematician named Jack Edmonds at the National Bureau of Standards had also worked along the same lines that we were. It was a case of simultaneous discovery.

This was very lucky for me, because that gave me entrée to visit Jack at the National Bureau of Standards. My first visit to him was one of the most inspiring moments of my entire career. Jack was not part of the establishment of the field. He didn't hold a professorship at a major university. He didn't even have a PhD.

He was just some guy who had a master's degree, had dropped out of the PhD program at the University of Maryland, somehow gotten himself a job at the Bureau of Standards, but he was doing absolutely brilliant work. He was also hindered a little bit by having shall we say a highly opinionated manner and personality [chuckles] which sometimes offended some of the powers that be at the RAND Corporation and Princeton University and such places.

But he did work of historic value in the field of combinatorial optimization. And I visited him in order to pursue this joint work that we were developing on the network flow problem, but we spent a day together where he told me about the concept of polynomial-time computation, the idea that you would like to show that your algorithm runs in a time bound that's only bounded by some fixed power of the size of the input. And he discussed how you could attack these combinatorial problems by formulating them properly and in a non-obvious way as linear programs, and how that led to beautiful combinatorial algorithms for standard problems called matching, matroid intersection, minimum arborescence. Jack just blew me away. It was like the most eye-opening single day I think I've ever spent.

Papadimitriou: Interesting you say that. I mean in a famous interview later, much later to George Nemhauser, Jack when asked, "So what are you telling me, Jack? Are you saying that this is like Einstein's special theory or something?" and Edmonds said, "Yeah, it's somewhere up there."

Karp: [laughs]

Papadimitriou: I know. At that time, I laughed, I smiled and laughed. But the more I think about it, he does have a point. I mean it was fundamental, transformative, and completely out of the blue.

Karp: Right. And it definitely was one of the developments that prepared the way for my later work on NP-completeness, which has to do with the issue of figuring out which problems are solvable in polynomial time. That came a few years a later.

So Jack and I ended up writing a paper together on these network flow problems. That's pretty much the story of my time at IBM. It was just a wonderful interlude in my life, running from 1959 to 1968.

_____

Papadimitriou: Rabin was not an employee of IBM. He visited.

Karp: Another one of my most important mentors was Michael Rabin. Michael is a very famous computer scientist who is only a few years older than I am, but

when I was really a beginner in the field, he was already famous for his work with Dana Scott on finite-state machines, an abstract model of computation.

It turned out that Michael was visiting the IBM Research Center, and we happened to be living in the same apartment complex on the outskirts of New York City, and so we began to commute together up to the lab. So every day I had a little over an hour of Michael's time. And Michael, who had a very wide knowledge of mathematics and logic and a very sharp sense of where the field was going, basically became my mentor and told me about all kinds of things from logic and mathematics, theory of computation. Those trips up the Saw Mill River Parkway were also quite an inspiration for me in the late '60s.

Papadimitriou: The landscape there is inspiring if I remember. You know, this is…

Karp: Well, I wasn't noticing the landscape because I was mulling over what Michael was telling me.

Papadimitriou: And you stayed at IBM for nine years?

Karp: I stayed at IBM for nine years. During that period, as a kind of hobby, I would teach courses at universities in New York City – NYU, Columbia, Brooklyn Polytechnic Institute. And it was great fun. It was just a kind of addition to my activities, just broadened my activities a little bit. It was a learning experience. And I discovered that I had a gift for imparting knowledge and was successful in teaching and moreover enjoyed it tremendously, the interaction with the students. Teaching has always been a vehicle for learning on my part. It was a way in which I was motivated to expand my knowledge, and it was just totally delightful. The only downside of it was that the wind that whipped down Riverside Drive when I went over to Columbia to teach evening courses was a little hard to take in the wintertime. But apart from small details like that, it was just a wonderful additional experience to do this teaching.

So I began to turn my thoughts toward making teaching a more important part of my activities and began to think about looking for a position at a university. But in 1968, when I learned that Berkeley was setting up a computer science department, I felt somehow that I was ready to make the move. My colleague Michael Harrison called me up. He regaled me with tales of the beauties of California, the exciting atmosphere on the Berkeley campus, full of revolutionary ideas. And the package of living in Berkeley, being able to witness some of the social movements of the time, and being part of a great university's computer science department appealed to me, and I gave up my position at IBM and moved to Berkeley in the fall of 1968.

Papadimitriou: Must have been an interesting time at Berkeley. You used the verb "witness" for the social movements.

Karp: Yes. I have to admit that I was you might say a voyeur, that I really wasn't throwing myself into for example the anti–Vietnam War movement or any of the other…

Papadimitriou: Free Speech Movement.

Karp: Free Speech Movement was a bit earlier. That had happened in '64 or '65. But I was very interested in the atmosphere around the antiwar movement and the various socially revolutionary things that were happening, and Berkeley was certainly one of the focal points of all of that. But I was basically a voyeur, a witness. I wasn't really deeply involved. I did a couple of minor things when the campus was closed. I would teach, give the lectures at my home. Once one of my best… Perhaps my best friend from that period of time was a colleague, Gene Lawler, who was more active than I in these movements, and I had to go out to Santa Rita Prison near Berkeley to bail him out of prison after he was arrested for participating in an unauthorized march through the streets. So I had some tangential involvement in all of that. *[0:30:00]*

Papadimitriou: Even 30 years later when I chose to come to Berkeley, the memory of that period was one of the attractions. So it must have been even more attractive when it was going on.

Karp: Yes, but Berkeley has changed. I remember when Mario Savio, who was the leader of the Free Speech Movement, emerged from hibernation, he was really avoiding the public eye, but he did come to Berkeley for the twentieth anniversary of the Free Speech Movement. I happened to be teaching a class that met at the same time as the lecture that he was going to be giving on Sproul Plaza. So I told this class of 70 students that they really should go and listen to Mario Savio and I would excuse them from the lecture. Out of the 70, just one person stood up and left the room. I think most of them were mystified. They had never heard of the Free Speech Movement or Mario Savio.

Part of the appeal of coming to Berkeley was that teaching would become a greater element in my life. And I took it very seriously, in fact perhaps too seriously. I think I sort of had the attitude that every lecture in a class was like a performance of *King Lear* and I had to, [chuckles] you know…

Papadimitriou: I thought it was. It's not?

Karp: It's not. I've gradually taken a more relaxed attitude towards teaching. As one gets more experienced in the field, there are more demands on one's time and there are certain moments when you have to go into class without full preparation and make the best of it. But in the early days, I was never unprepared. I always prepared thoroughly and, if anything, my lectures were too polished. I think sometimes it's good to show some of the rough edges. In fact,

some of my colleagues were very good at appearing to get stuck. Whenever you seem to be getting stuck of course, it excites the class. But I didn't rely on such cheap tricks. I'd like to be extremely well prepared and to lay things out in a very organized way.

Papadimitriou: You are a legendary teacher, both around the department and in the field. I mean, so are you telling me it's preparation?

Karp: Yes. Preparation, preparation, preparation. These are the three foundations of good teaching.

Well, I should say that by now the level of teaching in the department that I'm in is phenomenal. The young people that we've hired are really setting higher standards than I think I ever enjoyed.

But there are different approaches, different styles. My most respected colleague from that period of time, from the first dozen years at Berkeley was Manuel Blum, a famous computer scientist. And Manuel…

Papadimitriou: I doubt that "Preparation, preparation, preparation" was his motto.

Karp: No. I think my greatest strength was preparation and precision. Manuel's greatest strength was first of all human warmth and additionally intuition. Manuel didn't pretend to come into class with pedantic preparation, but he was able to convey the excitement of what he was doing. He was a divergent thinker, he originated many new lines of investigation, and he was able to inspire students who went on to do terrific things.

Papadimitriou: I recently read something written by him. He says that if you can prove that a statement is true and at the same time you can prove that the same statement is false, then you know you're onto something.

Karp: I understand. But that wasn't my style. My style was to decide whether it was true or false before I went into class. [chuckles] But we had a wonderful time during the '70s and '80s when Manuel and I were together and building up Berkeley's activity in theory of computing. Gene Lawler, whom I mentioned earlier, was also a part of that.

Papadimitriou: You became chair at that time. There were administrative changes or cataclysms going on.

Karp: Yes. When I joined Berkeley, I became the member of two departments. One of them was related to operations research in the College of Engineering, and the other one, which I thought of as my main department, was a fledgling computer science department in the College of Letters and Science, which had

split off from the electrical engineering department. So Berkeley had a situation where we had these two departments establishing themselves – the older electrical engineering department, which had a computer science section of 12 to 15 faculty, and a small new department in the College of Letters and Science that was trying to define itself in a slightly different way relating more to the social sciences and the humanities. But it turned out that the differences were very superficial, and it became clear after a while that the two departments were really duplicating each other and the administration decided that something had to be done about this duplication. There were also very bitter feelings on the part of many of the faculty in the two departments. There were a number of people making personal innuendos and it was a messy situation.

And I was selected to be the leader of this activity after it was merged back into the electrical engineering department as a supposedly autonomous division of the department. My main job was to sort of mend the fences, to get people past the rivalries that had been going on. And I think in that we were successful. I think people were really ready to give up their grudges, so from that point of view things went pretty smoothly.

The Computer Science Division that I was heading was in some fuzzy way supposed to have some degree of autonomy, and the…

Papadimitriou: Still fuzzy.

Karp: Still fuzzy after all these years. But the chairman that I worked under, at least for the first year of my term, was very understanding of our need for autonomy. He gave me some room and I was very motivated then to do my best.

I wasn't a born administrator though, and after a while I started sneaking off to draw little mathematical diagrams on my blackboard when I should have been dealing with the budget and such things. So I don't think that the later period of my administration was so notable, but at least by the time I stepped down from that job after a couple of years, I think we had reached a more peaceful environment.

———-

Karp: Let's take a particular computational problem as an example. Let's take the problem of coloring a graph. A graph is just a structure where you have a set of points called vertices and lines called edges connecting them. So it's a pattern of interconnections between the vertices. And if you're given a graph, you can ask the following question. Can you put a color on each vertex, either red, blue, or green, in such a way that no two vertices that are joined by an edge have the same color? That's called the graph coloring problem.

There are a couple of things to note about this problem. The first thing to note is that there are really an infinite number of instances of this problem, because you can be given any graph of any size.

Papadimitriou: And any number of colors to…

Karp: And any number of colors. But for simplicity, let's just say red, blue, and green.

Papadimitriou: Three. Okay.

Karp: Then there's a decision problem. For which graphs can you actually assign the colors so that you never get two greens adjacent to each other or two reds or blues adjacent to each other? How could you prove that the graph could be colored?

Well, you could prove it very easily. You just exhibit the coloring and anybody could check. So this is an example of first of all a combinatorial problem, combinatorial decision problem. It has the property that a brute-force approach would be to just try all possible assignments of the colors, and that would be hugely expensive. So the obvious way to attack the problem would lead to a tremendous combinatorial explosion in the running time of the algorithm.

So the question is, is there an algorithm that can reliably answer this question of whether the graph can be colored properly with three colors while running rapidly? *[0:40:00]* And our notion of rapid computation is by convention the property that the running time grows as some fixed power of the size of the graph rather than exponentially in the size of the graph – in other words, if the running time doubled every time you had one more vertex…

Papadimitriou: Or tripled as is the case with the coloring problem.

Karp: Or tripled as is the case with the brute-force approach to the coloring problem. That's called exponential time, and that's not satisfactory. So we're looking for something that runs in time proportional to the number of vertices or the square or the third power or some fixed power of the number of vertices.

We have many, many such problems, and we like to classify them. We would like to identify those problems that can be solved in polynomial time. And we have a name for that class of problems. We call it P for "polynomial."

Now you could also ask, for which problems is there a quick way of demonstrating… once you have decided that it's solvable, a quick way of demonstrating that it's solvable? Well, in the case of the coloring problem, that's easy. You just exhibit the coloring. So the class of problems where there's a quick way of presenting a solution once you have it is called NP for

"nondeterministic polynomial time." I won't explain why the term "nondeterministic" is appropriate, but that's what we call it.

It turns out that the great majority of the combinatorial problems that arise in applications, whether in pure mathematics or in scheduling factories or designing computer chips or making timetables for schools or dispatching taxicabs or – an enormous number of everyday problems – packing your suitcases into the trunk of your car…

Papadimitriou: Analyzing programs, which was the subject of your thesis.

Karp: Analyzing programs. There's this vast terrain of combinatorial decision problems and related also so-called optimization problems. Most of them, if they're formulated as yes-no decision problems like "Can you or can you not color the graph with three colors?" most of them lie in this class of nondeterministic polynomial-time problems.

Papadimitriou: NP.

Karp: The class NP.

Papadimitriou: Which is also – isn't it? – a natural limit of our ambition, because if you cannot even recognize a solution if they gave it to you, if you bumped into it, then how can you expect to find it?

Karp: That's right. But it's interesting that the property is not symmetric. It's easy to demonstrate that a graph *can* be colored with three colors, but there's no obvious way and in fact no way known to succinctly demonstrate that it *cannot* be colored with three colors. So there's also that aspect of it.

But basically you're right. You could, with proper formulation…

Papadimitriou: As Edmonds would have put it, salesmen of colorable graphs have an easy time because they can demonstrate to customers that they are colorable, but salesmen of uncolorable graphs don't because there's no obvious way to…

Karp: Right. Exactly, exactly.

Taking a slight technical liberty, NP represents essentially all of the usually… the combinatorial problems that arise in practice. And within that, there is P, the seemingly much smaller class of problems for which we have polynomial-time algorithms rather than exponential-time algorithms. So there are many problems are in P – the problem of sorting a list of numbers, the problem of finding the shortest path through a graph, the network flow problems that we talked about earlier are examples of problems that are in the class P. But these are really

exceptional. The great majority of the problems that arise in applications and which lie properly described in NP do not appear to have such polynomial-time algorithms. The great majority of problems that we have to solve in practice don't appear to be in the class P. But we have at the present time no proof of that. It still remains a legitimate possibility that P is equal to NP, that the class of problems that can be solved efficiently is the same as the class of problems that can be verified efficiently once you have found a solution.

Nobody, none of us I think really believe that those two classes are the same. If the two classes were the same, it would be tantamount to saying that finding a proof is as easy as verifying a proof, which violates all of our intuitions about mathematics. But at the present time there is no rigorous demonstration that the classes P and NP are different from each other, and that's generally considered the biggest open problem in computer science and maybe one of the half dozen most significant open problems in all of mathematics. And in fact among the open problems in mathematics, it may be the one that has the greatest philosophical significance because it deals with the very nature of proof and whether finding a solution is harder than exhibiting a solution.

Papadimitriou: Interestingly, it was discovered that in 1954, Kurt Gödel himself had proposed it, had actually stated…

Karp: Right. Yes, a letter by Gödel to…

Papadimitriou: Von Neumann, yes.

Karp: Yeah, in an informal way raised this question. And later it was found that John Nash had speculated along these lines as well.

Papadimitriou: That's right.

Karp: And I discovered in a trip to Czechoslovakia back in the '70s that this same question essentially was current in the Soviet Union. I met the mathematician Trakhtenbrot.

Papadimitriou: The *perebor* problem.

Karp: The *perebor* problem, yeah, they called it in Russia.

Papadimitriou: "Search." It means "search," unless I'm mistaken.

Karp: I guess so. I don't know. And in particular, the developments that I'll describe in a minute regarding this P versus NP problem were carried out in slightly different forms but essentially equivalently both in the Soviet Union by Leonid Levin and in the West by Stephen Cook and myself.

Papadimitriou: Did you hear from Edmonds about the P versus NP problem?

Karp: No. I first became aware of it through a paper by Stephen Cook, a former colleague at Berkeley who had moved to the University of Toronto. It was in that paper that he gave a formal definition of NP, the class of problems for which a solution can easily be verified. One could argue that Jack Edmonds had this concept but hadn't quite formalized it in a way that theoretical computer scientists would expect. But Cook did that.

Then he did one additional amazing thing. He showed that a particular problem in mathematical logic was as hard as any problem in the class NP in the sense that it had sufficient expressive power to enable any problem in NP to be described. This is what's called the satisfiability problem of propositional logic. You have variables that can assume the value true or false and you have a collection of conditions that those variables have to satisfy. Those conditions are called "clauses." A typical clause might be the condition that "Either A is true or B is false or C is true," and you've given a collection of these clauses involving a set of variables, and the question then is whether you can satisfy those clauses. What that means is that you can assign the value true or false to each of these variables so that every one of the conditions is satisfied. That's call the satisfiability problem of propositional logic.

Cook showed that that problem actually held the key to the question of whether P is equal to NP, because given any other problem that lies in NP with a suitable description of the problem, *[0:50:00]* you can write down any instance of such a problem. For example, the graph coloring problem. You can rewrite any instance of the graph coloring problem as a set of clauses. The clauses would say, "Every vertex has to be either red or blue or green," "It cannot be both red and blue," etc., etc., "If you have two vertices that are adjacent to each other, they can't both be green." In that way, you can write down a finite list of clauses that capture this particular instance of the graph coloring problem. And that essentially amounts to a reduction of the graph coloring problem to this problem in logic of deciding whether a set of logical clauses can be satisfied.

Cook demonstrated that given any problem in the class NP, which is essentially the universe of typically occurring combinatorial problems, you can rewrite it in polynomial time as a problem of satisfiability. You can take any instance of it and in a uniform way rewrite it as an instance of the satisfiability problem. What that means is that the satisfiability problem is in some sense the most general, the most universal, the most expressive problem in the class NP, and if you could have a polynomial-time algorithm for solving that problem, in other words if that problem is in the class P, then every problem in NP would be in the class P.

Papadimitriou: So it is the least likely problem to be in P.

Karp: So it's the least likely problem to be in P. If that problem fell, then every other… all of the problems, all of these diverse problems that arise in engineering and commerce and the natural sciences and computer programming and so on, all of them would be solvable in polynomial time. Too much for anyone to believe, but… So what that means is that we can focus on one particular problem if we wish, and that problem captures the question about the whole class, whether the whole class NP lies in P, is solvable efficiently.

This was a terrific achievement by Stephen Cook. He was coming at it from the viewpoint of a logician, but when I saw his paper, it evoked a sort of different line of thinking on my part. Because it occurred to me, I felt convinced of it even from the start, that there were many specific problems that had the same universality, that it wasn't a peculiar characteristic of just this satisfiability problem of propositional logic, but the down-to-earth problems of routing, packing, matching, scheduling, satisfying constraints, design of circuits, a host of problems from different areas of application would have also this same universality.

So how does one prove something like that? The way you start from Cook's result that the satisfiability problem has this universal quality, that everything in the class can be reduced to the satisfiability problem, now if we could show that the satisfiability problem in turn could be rewritten as a graph coloring problem, then the graph coloring problem would also be universal. Following that line of thought, I created a set of polynomial-time reductions showing that something like 21 different problems from various domains, but problems that are well recognized as being central in different application areas, also had the same universality property. And we call problems of this kind of NP-complete problems.

So I presented a paper at a conference at IBM in which I unveiled my list of 21 problems. And it was perhaps the first occasion where the theory of computing directly had bearing on problems out there in the real world. It happened to be sort of the sweet spot you might say where the typical difficulty of optimization problems in the real world was captured by the class NP. So starting from my list of 21 problems, people in various fields ranging from physics to biology to different branches of engineering began constructing reductions working from the reductions that I had given and developing more and more until thousands of problems eventually were shown to be NP-complete, to have this universal character. What this means in practice is that when a problem is shown to be NP-complete, as most practical problems can be, it's incredibly unlikely that we will ever find a fast algorithm that succeeds in solving all instances of the problem. So it's a kind of impossibility result.

Unfortunately, we have a strong belief that NP is not contained in P but we don't know it for sure. This is still the P versus NP problem.

Papadimitriou: But our ignorance has let us go a long way.

Karp: But our ignorance has let us go a long way because we now know that all of these NP-complete problems are equivalent – either all of them or none of them are solvable in polynomial time.

Papadimitriou: And also I think that, something that always surprised me in NP-completeness, is its broad applicability, that it's not like some problems are in P, some problems are NP-complete, and for most problems we don't know.

Karp: Right. Most problems are NP-complete.

Papadimitriou: Most problems can be classified. If they're not NP-complete, you start suspecting that maybe there's a polynomial-time algorithm. Lo and behold, eventually you may find it. Amazing foresight from your part is that, reading your paper, in the end you seem between astonished and annoyed that three problems, which one of them was factoring, the other was primality, the other was graph isomorphism, resisted your classification. I guess linear programming was classified soon after that, but the other two still resisted.

Karp: From the starting point of my paper, people developed a kind of cornucopia of NP-complete problems using very refined methods far beyond the relatively simple methods that I had used. And Christos, you were very influential in that line work.

Papadimitriou: I started my graduate career a few months after you published your paper. This means that I never had trouble really choosing a thesis topic.

Karp: I was in awe of the depth of some of the reductions that you and other colleagues like Larry Stockmeyer and Mike Garey and David Johnson and various other people produced. So it became a very refined area where not only were broad problems shown to be NP-complete but even very restricted special cases of them could be shown to be general, to be universal.

Papadimitriou: Also you spoke about the breadth of the applicability of this notion. I probably have told you this, but I looked up at a search engine the term "NP-complete," how often does it appear in scientific papers, and I got over 2,000 papers that contain "NP-complete" in their title and abstract. That may not be surprising until I tell you I had restricted the search to physics and chemistry.

Karp: I see, I see. I wonder how many of those authors actually knew the definition of NP-completeness and how many of them were using it in a kind of informal way synonymous with "hard."

Papadimitriou: This would speak even louder for its impact.

Karp: Yeah, yeah.

Papadimitriou: The more an achievement, a scientific achievement is abused in the literature, the higher it is.

Karp: Yeah, that's interesting.

So this was a peak moment of my career, but it left us with a question. How do we cope with these problems? We still have to schedule our factories and our schools. We still have to ship cement from factories to the battlefront and so forth. We still have to optimize production in different ways. And not only that. There are many problems in pure mathematics that we would like to attack from this point of view. So how is it that the world doesn't grind to a halt? How is it that we can still dispatch taxicabs *[1:00:00]* and pack the suitcases into the trunk of the car? And partly it's because some of these are fairly small instances, but it's more than that. We seem to be able to do well on quite huge ones as well, even though the problems are NP-complete.

So what is a possible methodology for demonstrating that all is not lost, that we can really do well? In practice, people seem to muddle through. They can take some big, very complicated problem like scheduling the classes in the school and they'll somehow be able to play around and eventually get a solution that satisfies almost all of the constraints that are imposed. The standard way that people in theoretical computer science attack this is to formulate the problem as an optimization problem, to ask not only "Is there a solution or isn't there a solution?" but to in some way attach a cost to different possible solutions. A good example would be the traveling salesman problem where you have distances between cities and you want to make a tour that minimizes the total distance.

Papadimitriou: Or graph coloring where you are not given *a priori* a number of colors but you want to minimize the number of colors.

Karp: Exactly. So you can define the notion of an approximate solution if maybe you're willing to be off by 5% or 10% in the tour of the salesman or in the cost of production in a factory or the area of the chip. That still could be quite satisfactory in practice. So people started proving theorems of the form "Here's a polynomial-time algorithm that solves problem X within 10% of the optimum." This led to a great extension of the theory of combinatorial optimization that pioneers like George Dantzig and Fulkerson and Edmonds and other people had developed using methods in many cases based on linear programming to get these approximations.

Unfortunately, practitioners were not terribly satisfied with these solutions, with these bounds. For example, a famous worker in computational biology once said to me, "Why do I care that they can prove it within a factor of 10 or log n in the worst case when I'm solving instances every day within 1% of optimal?" In short, it has turned out that in most cases the bounds that one can prove about the

degree of approximability using a polynomial-time algorithm are a bit too pessimistic to really satisfy practitioners.

Papadimitriou: I totally agree. And also in some sense this unfortunate effect can also be seen in a different way I think – that with using mathematics as you understand them today, the mathematics you are using, it's very hard to bring closed upper and lower bounds on the approximability of a problem that we can prove. In other words, the upper bound, how well you can do, and the lower bound, what you know to be NP-complete to do. And that's in contrast to the original NP-completeness theory where there was a very fine dividing line between the problems that we can solve and the problems that we cannot solve exactly.

Karp: There's another very beautiful development that I'll just mention parenthetically, which is the area of hardness of approximation. People developed very powerful theorems in I guess it was the late '80s, enabling them to prove the limits of approximate algorithms, that you were able to prove in some cases that the degree of approximation that can be achieved in polynomial time is very poor, that you have to accept a very large ratio between the cost of the approximate solution and the cost of the optimal solution. So briefly, one can say that people were showing that for many of these problems, getting any kind of decent approximate solution is as hard as solving the problem exactly. That's become one of the mainstream areas of the theory of computational complexity and one in which Berkeley students and faculty have had a lot of influence.

Getting back to the birth of NP-completeness a little bit with my personal history on this, first of all I became distracted for a couple of years because I was appointed to head this new Computer Science Division where we were attempting to heal the wounds of the rivalry that had been created. And unlike some others who have had such positions and, like yourself, were able to continue their research, I didn't seem to be able to juggle things well enough. So I had to really concentrate on my administrative job for a couple of years and didn't make progress on my theoretical work. And by the time I left my administrative position, the techniques of proving NP-completeness had moved well beyond what I had been able to do earlier.

I should mention that there was… I think I did say something about it a few moments ago, but there was in parallel with the work that Stephen Cook did in the original NP-completeness and followed on by further reductions, Leonid Levin, a brilliant mathematician in the Soviet Union, did very much the same thing around the same time. And Levin eventually came to the United States and has become a prominent member of our theoretical computer science community.

When I came off my administrative post and had a little bit of time to think, I began to ask myself, "How can we bridge the gap? How can we get some

constructive results to demonstrate that all is not lost with these NP-complete problems?" And it occurred to me that maybe the thing to do was to show that some of these problems were efficiently solvable on the average. So I began to delve into average-case complexity.

What do we mean by "solving a problem on the average"? You have to define some probability space of problem instances. For example, if you're looking at a problem on graphs, you might assume that all the graphs with a given number of vertices and edges are equally likely, a very simplistic assumption. Or if you're dealing with a problem where the entry, the data in the problem consists of the numbers in a matrix, you might assume that these numbers are drawn independently from some simple distribution. And indeed, if you're willing to make assumptions like this, you can prove nice theorems, you can prove that it's easy for example to find what's called a Hamiltonian circuit in a graph, a tour that visits all the cities without repetition. And many of these problems become easy. You can show that that traveling salesman problem can be solved with very small relative error.

So I reveled in this and had a great time proving results of this kind for a few years in the mid to late '70s. The reception for this kind of work by the community was at best mixed, because the simple probability distributions that we were assuming didn't strike people as being particularly descriptive of the kinds of problems that would come up in practice.

Papadimitriou: Perhaps the intellectual tradition of computer science has been to be agnostic about where the inputs come from.

Karp: Yes, yes. You'd like something that works for all inputs, and that was not being provided by this probabilistic analysis. So I had a great time with it, but it was not necessarily a landmark achievement in terms of the progress of the field.

However, a related direction became very prominent. In the kind of probabilistic analysis that I was engaged in, a probability is in the generation of the input. So you're assuming that the input is a random graph or a random matrix or something of that nature. There's another way to inject probability, which is to assume that the input can be any graph, any matrix, any data object, but that the algorithm is allowed to generate random numbers or allowed to have a source of random numbers, random zeros and ones for example. *[1:10:00]*

In a 1976 paper presented at a conference at Carnegie Mellon University, Michael Rabin, the hero with whom I had commuted to IBM back in those days, again inspired me and the whole community by giving some beautiful examples of algorithms which depended on a source of random numbers and could solve problems that we didn't know how to solve without randomization. And he gave two very striking examples. One of them is the problem of finding the closest pair of points in *d*-dimensional space without exhaustively looking at all pairs, and the

other one was testing whether a number is prime, whether a number has no divisors except itself and one.

So Rabin, building on some earlier work by Gary Miller, who was a student at Berkeley at the time, gave a randomized polynomial-time algorithm for testing whether a number is prime. The algorithm depends on random coin tosses, random bits. It generates random tests that can be applied to the given number. If any of these tests fail, then you have a demonstration that the number is not prime. But if these random tests succeed, then you're left in doubt as to whether the number is prime or not.

Papadimitriou: But tiny doubt.

Karp: But very small doubt, because every time you generate a new random test, if the number is not prime, you'll have at least a 50% chance of showing that it's not prime. So if you apply a series of such tests, the chance that a number that's not prime would survive and remain in doubt would become vanishingly small.

So in practice, this kind of randomized algorithm is really quite useful and quite valuable. The first polynomial-time randomized algorithm for primality was actually due to the mathematicians Solovay and Strassen, but the version that was due to Miller and Rabin somehow, through the force of Michael Rabin's character in part, led to a revolution in the way we think about algorithms. There is the philosophical question of how do you actually get random bits, and there's a whole body of work on that. But in practice, one can basically get sufficient degree of random bits for practical purposes. So, many, many problems were solved more efficiently or more simply by using randomization.

Papadimitriou: You did a lot of work on this.

Karp: Yes, I did. There are a couple of… I can give a couple of examples. One of them, in one of the stories, Michael Rabin reappears.

Michael came to Berkeley one day and of course we were overjoyed to see him and took him out to lunch and were having a nice discussion in which Michael revealed a new algorithm that he had developed for the following problem. Suppose you have a large body of text and you're given a particular word, and you want to test whether that word occurs somewhere in the body of text. Well, you can do it by a brute-force method of sliding the word across the text and checking in each position whether you get an exact match with the word. But if the word is very long, that's not a very efficient way to do it. What Rabin had was a randomized algorithm which would achieve the same effect of comparing the word with every position in the text but at the cost of only a constant number of operations for every position in the text, what we call a linear-time algorithm.

The basic idea is what we call fingerprinting – that you take the word and you compute a kind of random function, which we call the fingerprint of the word. It's just like a fingerprint for a human. A fingerprint doesn't describe you completely, but it's extremely unlikely that two different individuals would have exactly the same fingerprint. You can use that same philosophy using random bits to generate a fingerprint. And Rabin had a particular way of doing that which allowed you to take the fingerprint of the word whose occurrences you are searching for and to slide across the text and compute the fingerprint of every stretch of text in such a way that when you advance from one position to the next, you'll do only a constant amount of work. And Rabin did this with a certain formalism involving random matrices of determinant 1.

Michael showed us all of this, and it occurred to me that there was a slightly simpler way to do it. So I shouted out, "Michael, you don't need the matrices!" Michael doesn't like to have somebody one-up him in that way, but he thought for a little bit and he had to agree that yes, that was a somewhat cleaner way to do it. So, very generously he made me a co-author of what is now known as the Rabin–Karp string-matching algorithm.

Another example is the problem of finding a maximal independent set of vertices in a graph. This is perhaps a little bit abstruse, so let me try to explain it.

Papadimitriou: You can relate it to coloring very naturally. You are trying to find a set of vertices that can be colored by one, by the same color.

Karp: The problem can be stated that way. You want to find a maximal set of vertices that can be colored with the same number… same color so that no two of them are adjacent.

Now it turns out that the problem of getting the largest possible number of vertices that are independent in this sense is NP-complete. But you can ask for a weaker condition. You can ask for a set of vertices which is independent – that means that no two of them have an edge between them – but is what we call *maximal*, which is different from maximum and means simply that no single vertex can be added to the set. Every vertex that's left out of the set is adjacent to some vertex in the set.

Now that problem is easy to solve by a sequential algorithm – you just pick a vertex and scratch out all the vertices adjacent to it and then pick another one and so forth. But what about doing it through a parallel algorithm? How can you do it very, very rapidly in parallel? People had conjectured that there was really no way to use the parallelism because you would then have to coordinate the actions of different processors in solving the problem. But Avi Wigderson, a brilliant postdoc with me at the time in the late… mid '80s I guess it was, together with me devised a fast parallel algorithm for doing this, and this was later improved by another colleague, Michael Luby, who was my PhD student.

The moral of the story is that randomization, a use of probability different from what I had been doing with the probabilistic analysis, turned out to be an extremely powerful tool in the kit bag of computer scientists.

Papadimitriou: Also in the same spirit, there was the random-walks algorithm.

Karp: This was worked on with Richard Lipton. Richard spent about a year or two at Berkeley in the late '70s. He's a brilliant researcher. He's currently well known for the blog that he's been producing. And we did a couple of things together. One of them had to do with searching a maze.

Imagine that you are trying to wander around a maze to see if somewhere in the maze there's a pot of gold. However, you have a very poor memory. When you reach an intersection in the maze, you can't even recall whether you've been there before. All that you can do is just pick some new edge in the maze and follow it. So we asked ourselves the question, "What if you make a random choice at every intersection as to how you're going to leave that intersection, and you can't even remember anything else except where you are and what the incident corridors of the maze are? How long would it take you to cover the whole maze?" We showed that it can be done in time that's only proportional to the third power of the number of nodes. In other words, it would be a polynomial-time algorithm *[1:20:00]* for searching the entire maze without any memory except the ability to know at any point which corridors are incident with your current location.

This can be rephrased in terms of a randomized algorithm for searching a graph with a very small amount of storage. And it's remained an open question whether problems that can be solved with that amount of storage using random bits can also be solved deterministically. It remains an open problem to this day.

Dick Lipton is an interesting character. He had an interesting way of working. He would do his thinking at night and his communicating during the day. His lecture preparation he just didn't do. He managed to do a pretty good job just using his wits without any time for preparing his lectures. But he'd think very hard about research problems in the evening, and then he'd come and circulate around the department, having discussions with various people. And I was typically on his circuit, so every day or two he would come by and reveal his latest thoughts. That's what led us into the random walk work, which turned out to eventually lead to a joint paper with some other colleagues at Toronto and elsewhere.

But he also proposed another interesting line of investigation. We have every reason to believe that the NP-hard problems are very hard, but suppose you were focusing on, in the case of a graph problem, just graphs of a certain size, a certain number of vertices and edges. Would you be able to devise a special-purpose algorithm that would allow you to handle all of those graphs? And we could formulate that in terms of logical circuits that would take the description of

the graph or the combinatorial input and feed it through a series of gates and end up with an output which would say yes or no to determine whether the input had the desired property. Could you formulate such a logical circuit tailored just for inputs of a certain size? And we showed that in the case of NP-hard problems or NP-complete problems, this would not be possible unless some very unlikely discovery event occurred in complexity theory that we believe is not possible. This ruled out one possible avenue for attacking NP-complete problems, namely focusing one at a time on different sizes of inputs.

It turned out that the result that Richard Lipton and I got could actually be improved and simplified by another very powerful researcher, Michael Sipser, who's now chairman of the mathematics department at MIT. Mike was and is a good friend, and we worked together on a certain problem about matchings in random graphs. A matching in a graph is just a set of edges that don't touch each other. So no vertex occurs in two edges of a matching. The problem was to find very large matchings in random graphs. The ability to do that would of course depend on the density of the graph. And what Mike and I discovered through simulations was that a certain algorithm would find a matching that covered almost all of the vertices of the graph provided the ratio of the number of edges to the number of vertices was above a critical value given by the transcendental number *e*.

And we proved various other things about certain algorithms to find matchings in random graphs. One of the techniques we used was to approximate a sort of discrete process by a continuous one that could be described by a differential equation. And there was a certain technical theorem that we depended on in making this transformation to the differential equation, and we depended on a theorem due to the mathematician Thomas Kurtz to justify the use of the differential equation. We had a very hard time applying the theorem, and if truth be told, there was even a slight gap in our final presentation of the result, and we labored over this difficult work for quite some time.

After the work was completed, Mike sent me a present. You may recall that the central character in Joseph Conrad's *Heart of Darkness* was named Kurtz, a very nefarious and mysterious character. So Mike sent me *Heart of Darkness* as a present to remind me of the difficulty we had with Kurtz's theorem.

Mike was one of the favorite students that passed through Berkeley during the era when Manuel and I were working together leading the activity at Berkeley. Two of the other people who passed through and went on to distinguished careers were Silvio Micali, a leading cryptography researcher at MIT, and Vijay Vazirani, a combinatorial mathematician at Georgia Tech. Silvio and Vijay entered our graduate program at the same time, and in their first year at Berkeley, the first semester at Berkeley, they took my graduate course on combinatorial algorithms. Unfortunately, neither of them passed in any homework

solutions during the entire term nor did they bother to hand in the take-home final examination at the end of the course. What were they doing?

They were working on a very difficult problem. They were trying to extend the result that John Hopcroft and I had obtained for a particular class of graphs to the full class of graphs having to do with computing a matching of maximum size. Edmonds had solved the problem in polynomial time as part of the brilliant work that I described earlier, but there was a certain conjecture about the improved running time that might be possible. And Silvio and Vijay dropped all of their coursework to concentrate on this and were very close to proving this very important result about matchings but hadn't quite completed it at the end of the semester.

So here I was faced with a dilemma. They were obviously brilliant. They had done no work in the course. It appeared that they might be solving this problem, but we didn't know for sure until they wrote everything down. What was the grade that I should give them at the end of the course?

Well, I couldn't give them a really good grade, but on the other hand I didn't want them to be kicked out of graduate school, so I decided to give them a B-minus, the lowest grade that they could get without getting into trouble with the dean. So I gave them a B-minus in the course. They never complained. They completed the solution. And about 20 years later when Vijay came to town, I told him that I was going to change the grade to an A.

Papadimitriou: And it was about that time that we had the Complexity Year at MSRI. *[Mathematical Science Research Institute – ed.]*

Karp: Yeah. That was wonderful. That was a great experience. MSRI is one of the mathematics institutes, I think the first one to be set up as a quasi-permanent institute for the study of mathematics with National Science Foundation sponsorship. And I think it was quite early in the history of MSRI. It was 1985 when Steve Smale asked me to join forces with him in running a year-long research program in computational complexity at MSRI. So I was very happy to join with Steve in proposing this, and during that year I worked intensively with the staff at MSRI, and in particular with the mathematician Calvin Moore, who was the associate director at the time. He was incredibly helpful and selfless in handling all of the details, all of the logistic details of selecting and bringing in postdoctoral fellows and visiting faculty. We had a glorious year of research on a daily basis. Always something new for the entire year with an all-star cast of postdocs and senior scientists.

I actually played Cupid that year. I introduced David Shmoys to Éva Tardos, who were both postdocs at the time at the institute. I asked them to run the colloquium, to run the weekly lectures at the institute. And…

Papadimitriou: They run a household now.

Karp: They soon became attached to each other and now for many years they've been running a joint household in Ithaca, New York. So that was a desirable outcome of the Complexity Year. *[1:30:00]*

I didn't get a lot of work done that year. I was too busy going to talks and enriching my background. But it was a great success and many of the postdocs and young faculty who participated said afterwards that it really broadened their horizons.

Papadimitriou: On the other side of the campus, there was ICSI. *[International Computer Science Institute – ed.]*

Karp: Yes. That was another one of my very fortunate external involvements. Around 1988, a group of people from industry and universities in Germany decided that they would like to sponsor a new institution that would receive postdoctoral researchers from Germany and integrate them into research projects. They had found that the usual channels of bringing postdocs from Germany into American universities didn't always guarantee a warm reception or a good connection for the postdocs with the ongoing activities. So the charter of this new institution was to provide a home for these postdocs where they could be well-integrated into the activities.

I became attached to this new enterprise and led a small group in theory of computation that also included Michael Luby, who later went on to found a successful company, and Lenore Blum, who has worked for years with Stephen Smale and is now a professor along with her husband Manuel and her son Avrim at Carnegie Mellon.

So with substantial funding from the German side, we were able to operate an ongoing algorithms research activity with many participants particularly from Germany, also from other countries such as Israel, and we became one of the obligatory stops that travelling theoreticians would make in their circuits around the country. It was very wonderful for about a decade. After a while, support from Germany was not renewed.

After several years at ICSI, I succumbed to an early retirement offer from the University of California. It was a famous moment in 1994 when the pension system of the university was rich and the university was poor, and so they decided to pension off, to provide inducements for people to retire. So even though I was nowhere close to ending my career, I succumbed to this offer and moved a year later to the University of Washington.

The reason I chose the University of Washington was that I was very interested in computational biology. In the early '90s, the Human Genome Project was

getting into high gear and there was a feeling that first of all biology would be the most important science of the coming generation and also I had the feeling that there was a great opportunity for work in combinatorial algorithms related to biology, because we were dealing basically with combinatorial objects. We were working with the genomes, which are strings of symbols. We were working with networks of interacting proteins and trying to figure out which subsets of the interacting proteins worked together to regulate the activities of cells. There were just many combinatorial problems. There were problems of figuring out the phylogenetic trees that indicate the descent of different species from ancestral species. It seemed like a very rich area where I felt that my combinatorial skills would give me an edge. So starting around 1991, I began to devote myself to combinatorial problems arising in computational biology, and I've continued to do that for over 20 years now.

So coincident with the extension of the early retirement offer came the opportunity to move to the University of Washington for a time, where computational biology was more advanced than it was at that time at Berkeley. So I spent four very happy years at the University of Washington, sitting at the feet of biologists and trying to learn the trade. It was a paradoxical situation because I would wander over to the labs where the various postdocs in molecular biology were doing their work and trying to understand what they were doing more or less as a humble student, and then when they decided to move on from their postdocs, they would ask me to write a letter of recommendation. So it wasn't clear which of us was the teacher and which of us was the student. But in any case, I benefitted greatly from that exposure to computational molecular biology and learned a little bit about what goes on in the labs and what kinds of experiments and measurements are feasible.

In 1999, after my four years in Seattle, I was invited back to Berkeley.

Papadimitriou: I hired you back. I was the chair at the time. Yeah.

Karp: I see. Well, I owe it all to you.

Papadimitriou: We had missed you.

Karp: Well, I'm glad that I was missed. I think part of the reason that the opportunity came to me was that Berkeley was trying to get into high gear in computational biology at that time. So I also became attached to a very new department, the bioengineering department. Over the next several years, collectively we were very successful in hiring some of the best young minds in the field, and now we have a thriving computational biology center spanning the campus.

When I returned to play that role at Berkeley, there were certain technicalities that required me to keep my appointment under 50%. So I relied again on the

International Computer Science Institute to be my second home. But this time, there was a different environment. We weren't receiving the ample funding from Germany but instead several colleagues in the area of computer networking, including Scott Shenker and Sally Floyd and Vern Paxson, were founding an outstanding networking research group funded by AT&T. And even though I hadn't done any work in computer networking, they invited me to join the team, which I was delighted to do. And so for the next several years, in parallel with my work in computational biology, I worked with you on one occasion and with Scott Shenker and others on problems in networking.

The most significant outcome of that period of research had to do with the concept of a peer-to-peer network. A peer-to-peer network is a system of computers which collectively share their common data. And the fundamental… So a computer in a peer-to-peer network may well be holding data that doesn't belong to it so to speak but is being stored on behalf of some other computers in the system. One of the problems in the peer-to-peer network is the navigation problem of locating the computer in the system that contains a given piece of data given the name of that data. This is a difficult problem because the system is dynamic, computers are entering and leaving, and so the assignment of data to computers is constantly changing. And we developed a particular method for solving the navigation problem. In other words, using this method, a query could enter the system at an arbitrary computer and, given only the name of the data that was being searched for, the computers could pass their query on from one to another until it reached the appropriate target where the answer could be found.

That paper turned out to be my second most cited paper, the most cited one being the initial paper on "Reducibility among Combinatorial Problems."

Papadimitriou: Your work in computational biology, I remember finding that fascinating. I was particularly moved and inspired by the faithfulness with which you wanted to approach biology. It's easy to become, to solve silly problems that you can then claim relate to biology. It's also easy to become the research assistant of a biologist. But to cut your own path and still be faithful to both computation and biology, I found that very inspiring.

Where do you see this computational biology work going?

Karp: I think that it's really of fundamental importance for the future of science and medicine. *[1:40:00]* Because we're entering a phase of personalized medicine now where it may be possible to tailor medical treatments according to the particular genetic structure of an individual, the particular mutations and variations in an individual's genome. For that, we have to analyze complete genomes to understand which genetic mutations or variants create susceptibilities to disease. It's a huge combinatorial problem, particularly since most diseases are multifactorial – in other words, there's not usually a single mutation or a single event that induces the disease but some kind of cascade of

events. So I think the profound problem of finding these combinations of mutations and tracing the causal path by which a disease is induced and possibly finding ways to intervene with those chains of causation, these are really fundamental questions that will change the face of biology and medicine and involve very complicated combinatorial work, which I hope to be a part of as we go along.

I will say though that one needs a certain mentality to work in computational biology, and I haven't completely succeeded in crossing that bridge. I still find myself seduced by the idea of creating beautiful algorithms. And that's part of the game… Very unfortunate. [laughs] That's part of the game, but there's much more to the game than finding wonderful algorithms. First of all, one has to be very faithful to the kinds of measurement methods that are available for biologists to acquire data. One has to work closely with biologists to understand what they really want to know, what they really want to find out. And once you produce a piece of software that can be used by a biologist to solve a problem, you have to convince them that they would like to use it, you have to produce highly tailored software and human–computer interfaces that make it easy for the biologists to use, and have to live with the unfortunate fact that the biologists really don't care very much how elegant your algorithms are. They're judgement of an algorithm is based entirely on whether they find the results it produces agreeable.

So it's a very complicated business. I can't claim to have had as much impact as I would have liked, because I haven't been as adept at playing all of the dimensions of this game as one might want.

One thing I do feel very good about, however, is having blazed a trail that has led many other people with training in theoretical computer science into computational biology, where they've been very productive in some cases, more productive than I have been. Several of my former students and postdocs who are now at places like Tel Aviv University or the University of Texas or UC San Diego have been very successful in working in computational biology.

Papadimitriou: You blazed a trail for computer scientists to open up in different sciences also. That was around the time when quantum computation was being formalized, and also it was just around this time, early-mid 1990s, when the Web and the economics of the Internet were knocking on our door. So it was the time where computer scientists started working on problems of the other sciences and of the world.

Karp: Berkeley has really been one of the focal points for that kind of outlook on the sciences. Back around the year 2001, a group of us in the theoretical computer science group at Berkeley realized that we had something in common – that in our own individual ways, we were reaching out to other scientific fields and examining fundamental processes that occur in those fields from the point of view of their algorithmic content. Typically, people in the physical sciences would

analyze a process in terms of the energy balances and the energy consumption, but often you could think of the logic of the process as well and look at how the process evolves as a kind of computation.

For example, our colleague Umesh Vazirani working in quantum computing took that point of view. Another colleague, Alistair Sinclair, applied the same point of view to statistical physics. You yourself ventured into economic mechanisms and the analysis of processes on the web. And I continue to investigate regulatory pathways and other aspects of computational biology. So we collectively view this idea of theory of computation as a lens on the sciences as really a fundamental departure that provides many new directions for the field of theory of computing. And we feel that by considering the implications for other scientific fields, we will discover new foundational problems that will enrich the theory of computation.

In fact, that's one of the guiding ideas behind the new institute that has just come into existence at Berkeley supported by the Simons Foundation. We're just inaugurating the Simons Institute for the Theory of Computing. It's a major enterprise for the campus. In the course of the last year and a half, trying to develop our plans for this institute and eventually winning the competition for Berkeley to be selected as the site of the institute, we've established extensive ties with people across different scientific fields on the campus ranging from climate science to astronomy to neuroscience to economics and cognitive science.

So I think we're entering a new era. You and I are both intimately involved in this institute. For me, it represents a departure in the role that I see for myself.

Papadimitriou: You are going to be the leader, the director of the institute.

Karp: Yeah. I'm going to be serving as the director. What this means is that at this late stage in my career, I'll be entering a new phase where my focus will be not as much as before on my own individual research but rather in creating an environment that will foster the ambitions and activities of younger scientists who will be flowing into our institute for various programs over the next decade or longer.

Papadimitriou: I predict that you're still going to draw combinatorial diagrams on your blackboard.

Karp: I hope to sneak away and do a little bit of doodling on the blackboard. But every day when I show up at the office, my first responsibility will be to put out any fires that may arise at the institute.

Papadimitriou: You started your scientific career when computer science was an embryo.

Karp: That's right.

Papadimitriou: In some sense, your career sort of helped shape and define our science. I mean, was it a choice? Would you say that you went in that direction because you saw something in the future? Or what can you say about computer science, how you compare computer science as it is now with how it was then?

Karp: Well, as I told you earlier, my mother told me that [laughs] data processing…

Papadimitriou: Amazing foresight. I can't believe that… I don't think that many mothers at that time knew this.

Karp: Right. Well, she was an unusual mother. But speaking seriously, I really feel that for somebody who is mathematically inclined and who is particularly drawn to discrete processes and dynamical systems where you look at systems not statically but in terms of how they change in the manner that an algorithm is executed, one can still find enormous challenges. I think I was very fortunate. I could not have chosen a better time to begin my research career, because computer science I believe had not even been given a name *[1:50:00]* at the time that I completed my PhD, but in effect we were doing it at Harvard. And so there were just fundamental problems that were sitting on the ground waiting to be snatched.

Papadimitriou: It's funny. My PhD is probably 17 or 18 years after yours, and still I feel that I was at the very beginning of computer science, that there was low-hanging fruit there, there was a lot of… more paths had to be cut than followed and so on.

Karp: Yeah. I think the frontiers are being pushed out, and it's probably not possible anymore for somebody to have a detailed knowledge of all of the different branches.

Papadimitriou: I think on the eve of my qualifying exam at Princeton, I was the last person who knew all of computer science. [laughs] That was 1974.

Karp: It's become more specialized, but compared to mathematics, the frontiers are still accessible. It's still possible for a young scientist to quickly move to the forefront of a subarea. And in fact I've been addressing this question in a very personal way because my son is now entering graduate school. After a career at college where he first studied neuroscience, then economics, he decided in his senior year that he would like to be a mathematician. So I've been counseling him over the past year or two about what he should do, how he should pursue it, and it turns out – quite by chance I think, although maybe it has something about the genes as well – that his interests are very parallel to mine and he'll be

entering a program in the fall in algorithms, combinatorics, and optimization, areas that are very closely related to computer science.

So yes, I still think that the opportunities are great. And in view of the immense sophistication of pure mathematics, one can probably get the best mileage from his limited abilities by working in computer science as compared with a branch of pure mathematics.

Papadimitriou: Computer is many things. It's an artifact. It is an industry. It is the object of intellectual study. It has immense impact on society. How do all these aspects of computation affect computer science and especially the theory?

Karp: We need to turn our attention to models different from those we used before. For many decades, the Turing machine model held center stage and it was adequate for most of our investigations. In the case of a Turing machine, the task is very simple – you have a single user computing a single function. By contrast, with the World Wide Web, what we have is a dynamic community of agents, partly cooperative, partly adversarial, conducting economic and social transactions. It requires a totally different emphasis and a whole new set of problems that involve economic mechanisms, communication, questions of inducement and motivation that do not arise in the classical theory of computation. You yourself have pointed out that we have to view the web as an artifact to be understood empirically, much as in the past we've studied the brain, society, and physical systems. So I think the opportunities are tremendous.

Papadimitriou: What would you say is the difference between the worldviews and ways of approaching problems between a mathematician and a computer scientist?

Karp: I think the most fundamental notion is that a computer scientist will tend to look at processes, at change, at dynamics, and at effective computation, whereas a mathematician might look at objects described by a fixed set of axioms and be interested often in non-constructive solutions. So I think the main differences are the notion of effective computation and the notion of dynamical processes.

[1:55:05]

[end of recording]