Computer Science is more important than Calculus: The challenge of living up to our potential

By Mark Guzdial and Elliot Soloway

In 1961, Alan Perlis made the argument that computer science should be considered part of a liberal education, and that *everyone* should learn to program. M. Mitchell Waldrop in his book *The Dream Machine* (Viking: 2001) says that he made the argument that programming was a fundamental intellectual skill, like mathematics. He argued that computers "will participate in almost every intellectual transaction that goes on in the university." Calculus is generally considered part of a liberal education—truly educated people know something significant about calculus. Calculus is the study of rates, and rates are important to many fields. Perlis argued that computer science is about *process*: Its specification, its execution, its composition, and its limitations. And process is important to *everybody*.

What would computer science education look like if we took Perlis' argument seriously? We can set up some expectations on what we would hope for a core liberal arts course. Almost everyone should be successful at the introductory course in the field. Students should emerge from the course with a sense about what's interesting in the field, and they should have some practical knowledge that they can apply in their fields. Both men and women should be equally successful at it.

Anyone who's taught a first course in computing knows that we don't come close to those goals. Withdraw-or-failure rates of 50% are not uncommon. Even at the recent SIGCSE conference, the best practices completion rates for non-majors in CS1 (the ACM/IEEE curriculum term for a first course in computing) were around 65%. The best we can do is only lose 35% of the students? The reports from the American Association of University Women (AAUW) and the book *Unlocking the Clubhouse* (MIT Press: 2001) by Margolis and Fisher paint the picture clearly with respect to gender in computer science. Women are a scarce resource in computer science, and the retention rates are even worse for women than men. It's fair to say that our current state is that many students emerging from CS1 hit the ground running—away, with no desire to ever return.

Alan Kay likes to argue these days that "the Computer Revolution hasn't happened yet." He says that the potential impact of the computer across our society is much greater than what we've seen so far. With CS1 in the state that it is, is it any surprise that we don't find professionals of all fields steeped in computer science and ready to apply its lessons to their fields?

Perlis places a high bar for us. Let's consider some of the implications of taking him seriously and think about building introductory computer science education that can be part of a liberal education. The argument that we are going to make is this: If we want to attract and retain a greater proportion of the students, we need to change our pedagogy, which includes changing what we teach. What we teach now is not interesting and motivating the majority of students—the raw data tell us that. We are going to provide concrete examples of the kinds of things that we feel can be taken out of introductory CS courses and what kinds of authentic content might fill a new kind of introductory CS.

Is sorting fundamental?

Much of what we teach in introductory computer science is completely practical and relevant for students who will one day be software developers, but is completely impractical and irrelevant for everyone else. This point was struck home for us recently in a conversation with a Chemical Engineering professor who had spent several years in industry doing molecular modeling. He reported that a program of 100 lines was enormous for him, and the longest program he ever wrote had 200 lines in it. He makes extensive use of libraries and external modules so that he writes as little as possible. Most of his programs are about 10 lines of code. "If I followed the commenting requirements that you guys have," he said. "My programs would be twice as large!" This is someone who does a lot of programming—sometimes several programs in a week. But we might call him a "tool builder" rather than a "software developer." A great many of the non-majors who take our introductory courses will end up being like him, not like us. What do we need in introductory computing courses to help this kind of future professional?

Most introductory computing curricula include a variety of concepts that are important not for their direct relevance to the tasks of the students, even as professionals, but for their value in demonstrating concepts or in serving as exemplars for important ideas. Many of the more esoteric data structures that we teach will never actually be implemented by any of our students. (Let's count the number of AVL and red-black trees we've implemented in the last year, shall we?) But even the uncommon data structures are helpful in describing how complex data structures should be constructed, if need be.

Sorting is a good case in point. Very few programmers will ever implement even half of the sorting algorithms we talk about. Most programmers, we argue, will never write a sort at all! Sorting facilities are so common in modern libraries that you will almost always reuse someone else's, rather than build your own. Yet, still we teach every student who walks in our doors about bubble sort and quicksort and heapsort. The argument is sound. Sorting is a good example of something which can be done slowly and poorly, or quickly and well, and thus serves to show the difference between $O(n \log n)$ and $O(n^2)$. But can't we make similar points without spending weeks teaching something that no one, especially not the tool builders, will ever use?

At Georgia Tech, we are now teaching an introductory computing course, *Introduction to Media Computation*. The focus of the course is to learn programming and computing concepts in a context of manipulating media. We splice sounds (then tweak the volume to make them sound right), teach chromakey like the weather forecasters use it, and explain how Photoshop filters work. We started with 120 non-CS and non-Engineering students, 2/3 female. For these students, the computer is a tool for communication much more than calculation, so the media context feels relevant and important to them.

In the Media Computation course, the difference between algorithm complexity comes through very naturally. Sound processing algorithms tend to run very quickly. Even though there are thousands of samples (16 bit values that each represent 1/44,100th of a second of sound) in each sound, they're only a 1-D array and most processing takes only a single loop. Picture processing takes longer. 2-D matrices of pixels (picture elements) almost always require nested loops. Movie processing takes *forever*, because we have to do the 2-D processing for each one of the *n* frames. It's a simple argument that matches their experience with writing and running these kinds of programs and that makes clear an important point about the limitations of computation.

Sorting is just a case in point. The point is that thinking about computer science as part of a liberal education will require us to re-think what we teach and where we teach it in the curriculum. Some of the interesting but uncommon issues may belong later in the curriculum, after more relevant and motivating issues come first.

Sampling and other new fundamentals

The media computation course is really only one way to make CS1 more practical and relevant to non-majors. We can imagine many others. We could build a course around web harvesting and visualization of the harvested data, for example. Students could build web spiders, use various pattern matching and parsing tools to pull out the data, then build three-dimensional visualizations of what they found. The *Virtual Worlds* course that Randy Pausch has built at CMU with his programming environment Alice demonstrates that 3-D visualizations are well within the range of a non-major undergraduate.

As we think about introductory computing courses that are about something, not just about the abstractions, we may find that there are new concepts that need to be added to our courses. There are fundamental concepts and algorithms that arise in these concrete domains that we are currently not teaching. That observation raises the question of what is fundamental and how do we decide.

Let's take the media computation context, since it's one with which we've become the most familiar. Below we are going to provide examples that lead us to a concept that is fundamental to media computation, but doesn't make it into most CS1 courses. The examples are in the programming language Python (http://www.python.org). The version of Python we're using is Jython (http://www.jython.org). That's Python implemented in Java which makes the Java class libraries accessible, and allows us to create a cross-platform, multimedia API for the students. For the most part, Python works as a kind of pseudo-code. The only prefatory statement that you may need is that blocks are delimited in Python by indentation: If it looks like a block, it is one.

We'll start with a program to crop a woman's face out of a picture and paste it into a canvas image.

```
def copyBarbsFace2():
  # Set up the source and target pictures
  barbf=getMediaPath("barbara.jpg")
  barb = makePicture(barbf)
  canvasf = getMediaPath("7inX95in.jpg")
  canvas = makePicture(canvasf)
  # Now, do the actual copying
  sourceX = 45
  for targetX in range(100,100+(200-45)):
    sourceY = 25
    for targetY in range(100,100+(200-25)):
      color = getColor(getPixel(barb,sourceX,sourceY))
      setColor(getPixel(canvas,targetX,targetY), color)
      sourceY = sourceY + 1
    sourceX = sourceX + 1
  show(barb)
  show(canvas)
  return canvas
```

This function gets the filename of our source picture "barbara.jpg" from a globally defined media folder. The canvas is a blank 7x9.5 inch JPEG image. (It's easier to provide blank files and associate all picture objects with JPEG files than to deal with the abstractions of constructors that allow creation of invisible pictures.) We then construct a loop to copy the picture starting at (45,25) and extending to (200,200). We copy into (100,100) and then as many pixels over as we need. At each pixel, we get the color from the source picture, then set the color of the corresponding pixel in the target. At the end of the loop we increment the **sourceY** and **sourceX** indices to synchronize with the **targetY** and **targetX** index variables in our **for** loops. At the end, we display each picture.

This is a fairly simple example of copying from one matrix into another. We don't typically spend much time in any of introductory computing courses on copying elements of matrices and arrays around—it's simply boring when the data is made up. But when the elements are colors in pixels, or samples in sounds, it's suddenly interesting to do all kinds of copying and variations on those copies.

Here's another version of the same code, with a couple of small but significant changes underlined. Now, instead of incrementing the source indices by 1, we increment by 0.5 and take the integer (**int** function) of the result. The sequence of 45, 45.5, 46, 46.5… becomes now 45, 45, 46, 46… We end up taking every pixel *twice*. This is the process of scaling up a picture: Doubling its size. Notice that the target size necessarily doubles.

```
def copyBarbsFaceLarger():
  # Set up the source and target pictures
  barbf=getMediaPath("barbara.jpg")
  barb = makePicture(barbf)
  canvasf = getMediaPath("7inX95in.jpg")
  canvas = makePicture(canvasf)
  # Now, do the actual copying
  sourceX = 45
  for targetX in range(100,100+((200-45)*2)):
    sourceY = 25
    for targetY in range(100,100+((200-25)*2)):
      color = getColor(getPixel(barb,int(sourceX),int(sourceY)))
      setColor(getPixel(canvas,targetX,targetY), color)
      sourceY = sourceY + 0.5
    sourceX = sourceX + 0.5
  show(barb)
  show(canvas)
  return canvas
```

A similar change is needed to scale *down* a picture. We need *fewer* pixels in the target, so skip a few. If you skip every other pixel (increment the source indices by 2), you shrink the image by ½.

Let's shift gears to sound. The example below copies a sound backwards. We open the same sound as a **source** and **target**. (That's an easy way to assure that two arrays are of the same size.) We start the **sourceIndex** at the end of the **source**, and move the **targetIndex** from the start to the end of the **target**. Each time through the loop, we get one sample value from the **source** and set the corresponding value in the **target**. We then decrement the **sourceIndex** and increment the **targetIndex**. The result is a reversal of the sound.

```
def backwards(filename):
  source = makeSound(filename)
  target = makeSound(filename)

  sourceIndex = getLength(source)
  for targetIndex in range(1,getLength(target)):
    sourceValue = getSampleValueAt(source,sourceIndex)
    setSampleValueAt(target,targetIndex,sourceValue)
    sourceIndex = sourceIndex - 1

  return target
```

Again, this is a simple loop copying from one array to another—relatively simple and boring for most CS1 classes. The result, though, is an interesting and concrete effect: You hear yourself or your teacher sounding funny. (Mark likes to use Elliot's recording of "Hello, world!" from Elliot's SIGCSE2002 keynote.)

Let's make a small tweak to the backwards function. We'll move both indices forward now, instead of the source decrementing while the target increases. But we'll increment the source index by 0.5 and take the integer of the source index when accessing the sample value. We've seen this pattern: The result is taking every sample twice. The concrete result is that the sound slows down, and the frequency of the sound drops by ½. If we applied this function to a sound at 440 Hz, the resultant sound is at 220 Hz.

```
def half(filename):
  source = makeSound(filename)
  target = makeSound(filename)

  sourceIndex = 1
  for targetIndex in range(1, getLength( target)):
    sourceValue=getSampleValueAt(source,int(sourceIndex))
    setSampleValueAt( target, targetIndex, sourceValue)
    sourceIndex = sourceIndex + 0.5

  play(target)
  return target
```

You can probably figure out how to do the other sound transformation, the doubling of the frequency: Simply take every-other sample. By skipping every other sample, we speed up the sound and double the frequency.

These examples demonstrate the concept of *sampling*, a fundamental concept in media computation. There are lots and lots of samples in a sound (44,100 per second in CD-quality sound), and lots of lots of pixels in a picture. Skipping a few or doubling a few still allows us to recognize the sound or the image. Sampling is a rich subject with the potential for a lot of exploration and opportunities for learning.

- Do distortions occur in sampling? Of course, and there are algorithms for blurring that were invented to deal with these kinds of distortions.
- If you shift the frequency of a source sound up (skip samples), and the target is long enough, you will run out of source samples. Should you start over at the first sample? That will probably work, but there may be problems with having "breaks" in the sound. Detecting and fixing breaks leads to opportunities to think about pattern matching in large data sets.
- There are other algorithms for sampling which lead to fewer distortions and provide an opportunity to introduce concepts like averaging or interpolating.

Is sampling fundamental computer science? That's a hard question. Is sampling a useful concept to include in a CS1 course? That's a much easier question. Shifting sounds to different frequencies is what every sampling keyboard does. Resizing images is what people do all the time in applications like Photoshop and Word. It's generalizable, concrete, relevant, and understandable. It's *authentic*. Sampling is a pedagogically-rich concept that is attractive for both majors and non-majors.

Of course, we can't just add sampling to every CS1 curriculum. CS1 courses are over-full with concepts as it is. But if we get rid of some of those concepts that aren't relevant for everyone, we have room for others that are more relevant.

Are there concepts like sampling that are key in other potential CS1 contexts, like web harvesting and visualization? We don't know—there may certainly be. We can only know by trying these additional contexts. The opportunity is ours to re-think what "fundamental" means and what belongs in an introductory course.

Conclusion

Alan Perlis' challenge to become part of the core of the University is over 40 years old now. To meet that challenge, we argue that we need to re-think our introductory computer science content in order to remove the old and broken and replace it with the new and relevant. Computer science is broader, deeper, and more interesting than it was 40 years old. We don't have to use the same examples to illustrate the same concepts — the range of algorithms we have to choose from, the examples we can draw on, and the references to real applications that we can make give us enormous flexibility. As we re-think introductory computer science, we can use these opportunities to create a curriculum that is authentic and motivating. And in so doing, we would rise to Perlis' challenge – computer science can become a core part of the intellectual life of the university.

P.S. As of the last day to drop courses, our *Introduction to Media Computation* course had only lost two of the original 120 students.