CS125 : Introduction to Computer Science

Lecture Notes #1
Computer Science and Software Design

Lecture 1 : Computer Science and Software Design

An introductory discussion

Imagine you walk into a room and there are 100 small boxes laid out in a single-file row on the floor. The lids of the boxes are closed. You are told that either exactly one of the boxes contains a penny, or else all the boxes are empty. Your task is to determine which box, if any, holds the penny.

Now, suppose that there isn't a penny inside any of the 100 boxes. However, you don't know that yet; you've only just now walked into the room, and you are asked to figure out which box – if any – holds a penny. How many of the boxes would you have to open before you could be certain there was no penny in any box? For example, if you open just one box, and it was empty, could you then be certain there was no penny in *any* of the 100 boxes? If not, how many boxes would you have to open before you *could* be certain that none of the 100 boxes held a penny? (Assume you can only tell if a penny is in a box, by opening the box and looking inside – i.e. you don't have metal detectors or X-ray machines or any such thing.)

Well, since you have not been given any extra information about the boxes, it will be necessary to look inside *every* box to see whether or not that box holds a penny. If you only check half the boxes, the penny could be in the boxes you didn't check. If you check all but one of the boxes, the penny could be in that last box you didn't check. You can only say for certain that no box contains a penny if you've checked every box.

Certainly, if you knew something extra about the boxes, that might help. For example, perhaps half the boxes are green, and half are red. If you were told that the penny – if there is one – is in a green box, then you would not need to check the red boxes, since you've been told the penny would not be in a red box. However, you *would* need to look *at* every box, since you at least need to determine if a box is green or red before deciding whether or not you need to bother opening it to look inside. On the other hand, if you are told that if there is a penny, it would only be in one of the three boxes closest to the door, then in that case, not only would you not need to look *in* most of the boxes, but you wouldn't even need to look *at* most of the boxes – you could just look inside the three boxes closest to the door, and ignore every other box in the room.

But if you are given no such additional information – if it's possible for the penny to be in any of the boxes – then you would need to check all of the boxes in order to be sure that there was no penny in any box.

Now, imagine a second scenario. What if I printed out the name of every single student at the University of Illinois, in a list that is *not* alphabetized, and gave that printout to you? Imagine that I then ask you to search that list to see if there is any student named "Abraham Lincoln" on the list. Perhaps there is such a student on the list, and perhaps there is not. However, if you only look at the first page of names, find no one named "Abraham Lincoln", and then quit, you can't be sure there is no such student at the University – maybe there's a student with that name on one of the other pages you didn't check. If you check all the pages except the last one, or if you check every name on every page except for the last name on the last page, you face a similar problem. As long as there are names you haven't looked at yet, one of those names could have been "Abraham Lincoln" and you'd have no way of knowing. You can only be sure there is no student with that name if you've checked every name.

Now, certainly, if the list is ordered in some manner, that would help. If the names are sorted alphabetically by last name, for example, you'd only need to check a certain part of the list – namely, where "Lincoln" would occur in alphabetical order. But if there is no order to the list for you to take advantage of, then you'd be forced to check every name if you wanted to be sure that no student named “Abraham Lincoln” existed.

Note that the problem of searching boxes for a penny, and the problem of searching a list of names for a student with a particular name, are very similar problems. In both problems, you have a number of similar items, in no particular order, and you are searching the collection of items for one which matches a description exactly. In the first case, you are looking at boxes one by one to find the box that matches the description “there’s a penny inside”; in the second case, you are looking at names one by one to find the name that is “Abraham Lincoln”. And in both problems, if the only information you have is the description you are trying to match, you need to examine *all* the items before you can be certain that no item matches the description. However, it is also true that in both problems, if you had additional information that you could use to rule out some items without examining them in detail, then it might have saved you some work.

This is computer science!!

Now, that might sound strange, since we have not even discussed computers yet. However, what we are doing here is conveying an important principle: *computer science is not about computers.*

Computer Science

Computer Science is the study of *computation.* (The discipline really should have been called “Computation Science” from the start, but “Computer Science” got chosen way back when, and now we’re stuck with that name, even though it’s a little bit misleading.) The study of computation encompasses a great deal of material. Among the more theoretical aspects of this material is the study of *algorithms*. An *algorithm* is a step-by-step procedure (with a finite number of steps) that, when followed from beginning to end, solves a particular problem – i.e., performs a specific computation. Note that these steps should be well-defined steps that can be easily and correctly followed every time the algorithm is run; for example, you can’t say “Step 5: meditate until the answer pops into your head” and count that as a legitimate step.

Note that we have basically come up with an algorithm for solving each of our earlier problems, even though we didn’t state the algorithm in any sort of formal way. Let’s do that now, for the box-and-penny problem:

```
To determine which of a finite number of boxes, if any, holds a penny:

   step 1) Start at the first box in the collection of boxes; call this
              your ``current box''; go to step 2.
   step 2) Open the ``current box'' to see if it holds a penny; if it
              does, go to step 6; otherwise go to step 3.
   step 3) If your ``current box'' is the last box in your collection of
              boxes, go to step 5, otherwise, go to step 4.
   step 4) Consider the next box after your ``current box'' to be the new
              ``current box'', and go back to step 2.
   step 5) It is confirmed that no box in the collection of boxes has
              a penny inside it, since we have now checked every box
              in the collection. Stop.
   step 6) You have found the box that contains a penny. Stop.
```

Note that as you run through this algorithm, step 4 takes you back to step 2. So, if the box does NOT hold a penny in step 2 (thus leading you to step 3), and if that is NOT your last box in step 3 (thus leading you to step 4), you end up at step 2 again, and might run through the series of steps 2, 3, and 4 over and over again. You only stop running steps 2, 3, and 4 repeatedly, when you either find a box with a penny in step 2, or you run out of boxes in step 3. And that should make sense, given our problem – we only stop looking for a penny, when we've found the penny, or run out of boxes to look at.

Note that this algorithm is very similar to the one we might have used to find "Abraham Lincoln" on our list of unalphabetized names:

```
To determine which of a finite number of student names, if any,
is ``Abraham Lincoln'':

   step 1) Start at the first name in the list of names; call this
              your ``current name''; go to step 2.
   step 2) Examine the ``current name'' to see if it is ``Abraham Lincoln'';
              if it is, go to step 6; otherwise go to step 3.
   step 3) If your ``current name'' is the last name in your list of
              names, go to step 5, otherwise, go to step 4.
   step 4) Consider the next name after your ``current name'' to be the new
              ``current name'', and go back to step 2.
   step 5) It is confirmed that no name in the list of names is
              ``Abraham Lincoln'', since we have now checked every name
              in the list. Stop.
   step 6) You have found the name that is ``Abraham Lincoln''. Stop.
```

In fact, you could generalize this, and have a procedure that work no matter what collection you were searching and no matter what particular value you were searching for:

```
To determine which of a finite number of values in a collection C,
matches some description D:
   step 1) Start at the first value in the collection; call this your
            ``current value''; go to step 2.
   step 2) Examine the ``current value'' to see if it matches your
             description; if it does, go to step 6; otherwise go to step 3.
   step 3) If your ``current value'' is the last value in your collection,
             go to step 5, otherwise, go to step 4.
   step 4) Consider the next value after your ``current value'' to be the new
             ``current value'', and go back to step 2.
   step 5) It is confirmed that no value in the collection matches the
             description, since we have now checked every value in the
             collection. Stop.
   step 6) You have found the value that matches your description. Stop.
```

The above is an algorithm known as "linear search" – and the purpose of the algorithm is to search a collection of items, one by one, from the first item to the last item, trying to find one that matches a particular description. As we have seen, it can be applied to searching boxes for a penny, or searching a list of names for a particular name. It can be applied to many other kinds of specific problems as well – any time we have a collection and need to search it for a particular item.

We will explore linear search and other kinds of searching, later in the semester. For now, we simply wanted to present it as one example of an algorithm – a finite, step-by-step procedure designed to solve a particular problem.

We'll talk about computers, too

Now, when we said "Computer science is not about computers", perhaps that was a little bit inaccurate. Usually, when we discuss algorithms, our intention is to eventually have computers, not people, run through the algorithms step by step. So, the design of computers, and the programming of computers, are also aspects of computer science. In fact, you will begin your study of the programming of computers, in this very course; later CS courses will introduce you to the design of computers as well.

The important idea behind the earlier discussion, though, was to point out that we can discuss *what* work needs to be done, without discussing *how* that work needs to be done. For example, based on our earlier discussion, we know that if you are searching through one hundred different integers, for one that equals 47, then – assuming that none of the integers *are* equal to 47, even if you don't know that yet – you would have to look at all one hundred integers to be certain that none of them are equal to 47. This is an inherent property of the problem. Even if you search through the collection of integers using a computer, the computer program you write will still have to look through all one hundred different integers. If you buy a computer tomorrow that is twice as fast, that doesn't mean you suddenly only need to look at fifty integers – because looking at every single integer is an inherent property of the computation itself, and has nothing at all to do with the speed of the computer or person doing that computation. If in the future, you buy a computer 100 times as fast as the one you use today, that computer will still need to look at all one hundred integers. A faster computer can only perform each individual step of an algorithm faster; it cannot *eliminate* any of the steps of the algorithm, since every one of those steps still needs to be performed in order for the algorithm to be run correctly. If your fast computer only looks at the first ninety-nine integers, the last integer could have been `47` and neither you or the program would have any way of knowing for sure – except by continuing onward and inspecting that final integer out of the one hundred integers.

There are many other sorts of advanced theoretical issues that can be discussed as well – and you will discuss many of them in later courses! CS125 is meant to introduce you to some introductory theoretical issues, as well as giving you some beginning instruction in how to program computers and design software. The important thing to realize, here, is that – though programming is *part* of computer science – a proper exposure to computer science involves more than just programming, and though we will not touch on every aspect of computer science in this course, we will indeed be doing more than just teaching you how to program in the Java language. We will be touching on some more general computer science issues as well – among them, algorithm design and analysis, and some program design techniques. All of these things are part of the discipline of computer science.

Conceptual Tools

In the coming lectures, you will learn about various computational ideas, and the particular Java syntax that you use to express those ideas in a Java program. Other programming languages might express those ideas with different syntax than Java uses, just like different spoken languages would have different words for the concept of "water" or "computer". However, just as the *idea* of "water" is the same whether the word you use to represent the idea is the English word "water", or the Spanish word for that concept, or the Japanese word for that concept, likewise, the computational ideas we will learn will will be the same, regardless of what language we happen to be using to express those ideas. In this course, we will use Java to express our ideas, and if this is the first time you've done any programming, it might be easy to think we are learning Java ideas instead of general programming ideas. That is not the case, though. We are learning general programming ideas which you will find in many different programming languages, and the particular symbols which represent those ideas in Java, won't necessarily be the same as the particular symbols that represent those ideas in other languages.

However, all programs basically consist of (1) obtaining data and (2) running instructions that manipulate that data. In that respect, there are three kinds of conceptual ideas we can introduce right now, which you will make heavy use of as you design programs throughout the semester:

1. *primitives* – these are the lowest-level data values and procedures provided by a programming language. Primitives are not things you need to define, since they are built right into the language; similarly, they can't really be broken down into smaller parts without losing their meaning. (The concept is similar to that of an "atom" in chemistry or physics class.)

    - When talking about data, the primitives are the built-in data values that your program can use without any further definition. For example, you can type numbers such as `5` or `-2` or `38.6` directly into your Java program; the language is designed to understand numerical values like that automatically. Similarly, characters such as a lowercase 'h' or the dollar sign '$', are understood by the language.
    - When talking about procedures, the primitives are small, basic operations such as simple arithmetic. Adding two numbers, or multiplying two numbers, would be considered

   primitive operations – they are built right into the language and you can use them without any further definition.

2. *composition* – this is the idea of taking smaller ideas, and putting them together to build a larger idea which is effectively just the collection of the smaller ideas

   - When we talk about data composition, we mean building larger pieces of information by collecting together smaller pieces of information. For example, when you take your name, address, social security number, GPA, and the names of the classes you took and the grades you got in those classes, and put all that information together, you have your personal record in the student database. In this case, the chunk of data known as a "student record", is in reality composed of many smaller pieces of data, such as your name, GPA, and so on. The student record is a *composition* of other, smaller data values, which together form a larger, more complex piece of data.
   - When we talk about procedural composition, we mean building larger procedures out of smaller ones. For example, it is possible to add two numbers together, and it is possible to divide a number by 3. So let's apply that first procedure (addition) twice, and then apply the second procedure (division by 3) after that. Given three numbers:

     (a) add the first two numbers together
     (b) add the sum from the previous step, to the third number
     (c) divide the sum from the previous step, by 3

     Now, in that case, we've applied some small, primitive procedures – addition and division. However, the result of the last step will be the average of the three numbers we were given. So, what we have really done above is to create a procedure for finding the average of three numbers. By combining some primitive mathematical operations – addition and division – we were able to produce a procedure that is slightly more complex. This more complex procedure is a *composition* of other, smaller procedures that together form a more larger, more complex procedure.

3. *abstraction* – this is the idea of focusing on the "big picture" of an idea, and ignoring the details that implement that big picture. The advantage to doing this is that worrying about those details often bogs us down in a lot of little issues that we really don't need to worry about. When you speak into a telephone, you might know that the phone is encoding sound waves as electrical signals, but you don't really worry too much about that when you speak into a phone, and certainly you don't perform that encoding on your own! You simply trust that the phone will perform the encoding properly, forget about it, and just pick up the phone and speak. You worry about the overall purpose of the phone, rather than the details of how that purpose is achieved. Certainly, someone or something has to handle those details, but that someone or something doesn't have to be you.

We make use of this idea when writing computer programs, as well:

- When we talk about data abstraction, we are referring to the idea of viewing a data composition in terms of what the overall collection of data is trying to represent, rather than focusing on the details of what pieces of data make up the composition. For example, we might have procedures that "print a student record" or "copy a student record"; in those cases, we don't need to worry about sending a lot of little pieces of data to those procedures; we simply send one piece of data – a student record – and don't need to worry that a student record is "really" lots of smaller pieces of data if you look at it closely.
- When we talk about procedural abstraction, we mean viewing a composition of smaller procedures in terms of what the overall goal is, rather than viewing it in terms of the smaller procedures that make up the larger one. For example, once we have written a procedure to take the average of three values, we can make use of it whenever we have three values we want the average of. We can send the three values to our procedure, and get the average back, and we don't have to worry about the details of how the average is calculated.

All these ideas fit together to aid us in program design. Quite often, we will often compose a larger piece of data out of smaller pieces of data, and then focus from then on, on the larger data concept rather than worrying about what smaller pieces of data form the larger one. Sometimes, those smaller pieces of data that form the larger piece of data, will be primitives, and sometimes they will instead be other data compositions we already created earlier. Ultimately, though, if you break any piece of data down into small enough detail, you will find that it is composed of the same limited set of data primitives that the language (and processor) supports. We just choose to focus on the "big picture" when we can, instead of always peeking into a data composition to see the smaller pieces of data from which it is made.

Similarly, quite often, we will compose a larger procedure out of smaller ones, and then focus from then on, on the larger procedure and what the overall task is that it accomplishes, rather than worrying about all the smaller little procedures that form it. Sometimes, the smaller procedures we use to form the larger procedure, will be primitives, and sometimes they will instead be other procedural compositions we already created earlier. Ultimately, though, if you break any procedure down into small enough detail, you will find that it is composed of the same limited set of procedural primitives that the language (and processor) supports. We just choose to focus on the "big picture" when we can, instead of always peeking into a procedural composition to see the smaller procedures from which it is made.

Over the first half of CS125, you will learn about many of the primitives available to you in the Java programming language, and you will also learn about the syntax available in Java for dealing with data composition and abstraction, and with procedural composition and abstraction. In addition, you will see the concept of abstraction in use in other ways as well.