Lecture 2: Examples of algorithms. Abstraction

Doina Precup
With many thanks to Prakash Panagaden and Mathieu Blanchette

January 8, 2014

1 Example: Intersection of student lists

Often, a certain problem can be solved by several different correct algorithms, which might have different efficiency. For example, suppose I want to find a good time to schedule a tutorial for COMP-250. To do this, I might want to know how many students from COMP-250 also take COMP-206, MATH-133 etc., in order to avoid conflicts. So now suppose that I have two lists of students (for two courses) in arrays a and b, and I want to compute the number of elements in common. We assume, for simplicity, that there are no duplicate elements in any array (this is true if they are indeed student lists). Ideally, we would like to get an algorithm that also runs fast (for instance, in terms of the number of comparisons it requires).

The simplest solution is to loop through a, and for each of its elements, look it up in b. In pseudocode, this would look as follows:

Algorithm: listIntersection(a,n,b,m)
Input: An array a of n elements (in our case, strings) and an array b of m elements of the same type as a. The elements of a are assumed to be distinct. The same is true for b.
Output: The number of elements present in both a and b
int i,j //indices for the two arrays
int $intersect \leftarrow 0$ //this variable will hold the result
for $i \leftarrow 0$ **to** $n-1$ **do**
  for $j \leftarrow 0$ **to** $m-1$ **do**
    if $a[i] = b[j]$ **then**
      $intersect \leftarrow intersect + 1$
      break
return $intersect$

Now let's think about the running time of this algorithm. Of course, the running time depends on the size of the two arrays, m and n: two big lists will take more time to intersect than two small ones. So we will have to **describe the running time as a function of the size of the input**. There is an additional problem, though: the running time may depend not only on the size of the input, but also on the **content** of the input. For instance, in our case, if we have arrays {"Alice", "Bob",

"Carl"} and {"Alice", "Bob", "Carl"}, we find the solution with 6 comparisons. But if the second array is {"Doina, "John", "Wayne"}, it will take 9 comparisons.

In principle there are three possible ways of measuring the running time of an algorithm:

1. **Best case:** This is the running time of the "easiest" input, and it is usually meaningless.

2. **Worst case:** This is the running time on the worst input. This is often not so hard to compute, and we will focus on it in this course. We denote this by O (read "Big-Oh"), and we will formalize this notion in an upcoming lecture.

   For instance, in the case of the list intersection, the worst case is the one in which the lists do not have any overlap, in which case we will do a number of comparisons of $m \cdot n$.

3. **Average case:** This is the running time on an "average" input. The trouble is that when we say "average", we must know what kind of inputs we are expecting (i.e., what is the input distribution) and this is often very hard to anticipate. Even when this is known, the average time may be hard to compute.

   In our example, suppose that we expect the lists to have an overlap of k students. Then for the $n-k$ students that are in a but not in b, we will need to go to the end of the array b, doing $(n-k)m$ comparisons. For the students that are in both, we need to be able to say how many comparisons we expect to make on the average. Maybe $\frac{m}{2}$ would be a good estimate, but this is highly dependent on the assumptions we make about the arrays we will receive. If this were true, the average running time would be $(n-k)m + k\frac{m}{2} = nm - k\frac{m}{2}$. Note that this is a "tighter" estimate than the worst-case, but depends on the expected overlap k. To simplify it, we will need to make further assumptions on what we expect k to be.

In this course we will focus on analyzing worst-case running time (complexity) of algorithms, in a way that is as independent as possible of the computer used, the programming language, the compiler and other such details.

2 Example: Finding an element in an array

An important part of list intersection is **finding a given value in an array**. The pseudocode for the algorithm is as follows:

Algorithm search(a,n,val)
Input: An array a of n elements and a value val that we want to find
Output: True if a contains val, false otherwise
int i; //this is an index in the array
$i \leftarrow 0$
while $(i < n)$ **do**
  if $a[i] = val$ **then**
    return true
  $i \leftarrow i + 1$
return false;

In the worst case, this algorithm will go through its internal loop n times (if the element is not in the array, or the element is in the last position), so it is "$O(n)$".

3 Abstraction

Now suppose that I wanted to **use** the algorithm above for some other purpose (e.g., to find the intersection of two lists). In order to call it, I only need to know the first three lines, which tell me the name of the algorithm, what arguments it expects and what it will return. We will call this the **interface** of the algorithm. Think of this as a **contract** specifying what you provide, and what the algorithm provides. It is extremely useful to have such contracts in order to be able to build and use complex programs. Technically, the interface only contains the types of the inputs and outputs. However, there may be other important information that is not specified directly by the data type. For instance, the output may always be positive, or always odd, or have some other special property. Or the method may have side effects in terms of modifying data that is not specifically sent as an input (changing "global" variables). In such cases, it is important to **write comments** which specify such "fine print" of the contract.

By analogy, consider the case of driving a car. In order to do this, you only need to know how to use the ignition key, steering wheel and brakes. You do *not* need to know how the engine works, how the electric wiring is done etc. Indeed, knowing all these details is useless if you just want to drive! These details are hidden behind the car's *interface*.

So, in computer science, like in daily life, we will rely a lot on **abstraction**, in order to handle complexity. In other words, we will need to **let go of irrelevant details**. One example is **procedural abstraction**: once an algorithm is understood, and written down precisely (either in pseudocode or in a programming language), it can be thought of as a method. This method can then be called from a more complex algorithm **as if it were a primitive step**. All we need in order to call algorithms is their **interface**, not their implementation. Another example is **data abstraction**: we can hide the details of complex data structures and only keep in mind how to use them (forget how they work internally), like in the example of the car. We can think of the data structure as having a certain behavior, without worrying anymore about how this behavior is produced. In this course, we will learn the internal workings of different data structures, but then forget about this and think of how they can be used to build large, interesting algorithms and programs.

4 Example: Using search for list intersection

The search procedure above can be used as a building block for the list intersection example, as follows:

Algorithm: listIntersection(a,n, b,m)
Input: An array a of n strings and an array b of m strings. The elements of a are assumed to be distinct. The same is true for b.
Output: The number of elements present in both a and b
int $intersect \leftarrow 0$
for $i \leftarrow 0$ **to** $n-1$ **do**

```
  if search(b,m,a[i]) then
      intersect ← intersect + 1
return intersect
```

Note that this code is slightly easier to understand and "cleaner" than the previous version. Also, note that the "search subroutine" can be used by many other algorithms. In general, we will make it our goal to write code that is **re-usable** and **modular** (i.e. made of small, understandable, easy-to-debug parts). And, if we manage to improve the search subroutine (as we will do in the next lecture) all algorithms using it will profit.

In terms of the running time, the new version of listIntersetction written above runs in $n\cdot$ (running time of search on an array of size m). So, even the running time is expressed in a modular way. This is nice, because once we analyze how long the search takes, the result can be used to figure out the running time of algorithms that call it. As it turns out, since we call search on an array of size m, it will run in $O(m)$, so the total time for the new version of listIntersection is the same as before, $O(nm)$. The main benefit in this version is not in a faster running time, but in having a program that is easier to understand, analyze and debug, and which re-uses other programs.