Lecture 3: More on list intersection

Doina Precup
With many thanks to Prakash Panagaden and Mathieu Blanchette

January 10, 2014

1 Example: Computing the intersection of two lists of students

In lecture 2, we started discussing the problem of computing the number of common elements in two arrays. We described an algorithm which loops over the two arrays and computes this in time $O(mn)$, where m and n are the sizes of the two arrays. What if the array is sorted? Can we do any better?

2 Binary Search

The answer is yes, by using an algorithm called **binary search**. The basic idea is to look at the middle of the array. If we found the name (or student id) we searched for, we are done. Otherwise, we look to the right or left part of the array, depending on whether we have a name that occurs before or after the one in the middle, in terms of alphabetic order (or whatever ordering we have). The pseudocode for binary search in general is as follows:

```
Algorithm: binarySearch ($a$,$n$,$val$)
Inputs: A sorted array $a$ of $n$ elements, and a value $val$ that we want to find
Output: True if $a$ contains $val$, false otherwise
int $l$, $r$; // these are indices that delimit the part of the array where we are searching
$l \leftarrow 0$
$r \leftarrow n - 1$
while ($r \geq l$) do
    int $mid \leftarrow \lfloor \frac{l+r}{2} \rfloor$ // this the index of the middle element
    if ($a[mid] = val$) return true
    else if ($a[mid] < val$) then $l \leftarrow mid + 1$
    else $r \leftarrow mid - 1$
return false
```

Now given this routine, the list intersection algorithm can be written 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 $intersect \leftarrow 0$
sort($b$,$m$)
for $i \leftarrow 1$ to $n$ do
   if binarySearch(b,m,a[i])
      $intersect \leftarrow intersect + 1$
return $intersect$
```

How much time will this algorithm take? Its running time will be the same as the running time of the sorting algorithm plus the running time of the binary search. We will see later that sorting takes roughly $m \log_2 m$, where m is the size of the array we sort. But how long does binary search take?

To reason about this, suppose that we are searching through an array of 7 elements. How many comparisons will we make, in the worst case? We will look at the middle of the array (element of index 3, assuming that we are indexing from 0). Then we will look at a chunk of size 3 elements, and after that at a single element. So in the end, we will make 3 comparisons in the worst case. This is roughly $\log_2 7$. How can we reason about this more generally? Think of how the size of the part of the array that is of interest to us decreases over time. Initially we have m elements, then $m/2$, $(m/2)/2 = m/2^2$ etc. In the worst case, the element is not there, and we will have to go all the way to a single element, i.e. $n/2^{\log_2 m}$. We do one comparison for each size of the array, so the total number will be $O(\log_2 m)$. Hence, binary search takes $O(\log_2 m)$.

Now going back to our list intersection algorithm, binary search is called in a loop which executes n times, so this will take $O(n \log_2 m)$. So it is best actually to sort one of the arrays, then run this algorithm. Since listIntersection has to sort then execute the loop, its total time is $O((n + m) \log_2 m)$. Is this kind of improvement important compared to the previous solution? To see this, consider how the running time grows with the sizes of the two arrays:

(m, n)	Naive solution	Binary search	
(8,8)	8*8=64	16*log2(8)=48	
(16,16)	16*16=256	32*log2(16) =128	
(32,32)	32*32=1024	64*log2(32)=320	
(64,64)	64*64=4096	128*log2(64)=1024	
...	...	...	
(1024,1024)	1 048 576	20 480	51 times faster!
(10^6, 10^6)	10^{12}	$4 * 10^7$	25000 times faster!

3 More on binary search. Recursion

The version of binary search that we wrote above is **iterative**: we have a loop and indices which tell us in which part of the array too look. But perhaps a more natural way to think about the problem is the following: we look at the middle element, and if it is not the value we want, we will now search either to the left of it, or to the right of it, **using the same procedure**. This is essentially what we are doing with our indices. But we can write this in a more direct way as well:

Algorithm: binarySearch (a,l,r,val)
Inputs: A sorted array a, indices l and r which delimit the part of interest, and a value val that we want to find
Output: True if the portion of a between l and r contains val, false otherwise

```
if ($l > r$) then return false
int $mid \leftarrow \lfloor \frac{l+r}{2} \rfloor$ // this the index of the middle element
if ($a[mid] = val$) then return true
else if ($a[mid] < val$) then return binarySearch($a$, $mid + 1$, $r$, $val$)
else return binarySearch($a$, $l$, $mid - 1$, $val$)
```

Note that we will call this algorithm on and array a of size n as:
binarySearch(a,0,$m - 1$,val)

Also note that the **else** is optional, we could remove them and the flow of the algorithm would not change. Finally, notice that the structure of the algorithm is somewhat different, we start by stating the "exit" conditions (the easy cases in which the algorithm terminates) then proceed with the others.

Note that this binary search algorithm actually calls itself! An algorithm which calls itself, on a different set of attributes, is called **recursive**. In this case, the execution of both the iterative and the recursive versions will proceed exactly the same. However, sometimes it is more natural to think of an algorithm recursively. As we will see later, it may also be easier to prove the correctness of a recursive algorithm, and to analyze its running time. We will see lots of examples of recursive algorithms later in this class.