Converting a number to its binary representation

Back in lecture 2, I gave you an algorithm for converting a decimal number \( n \geq 1 \) to binary. Here we write this algorithm recursively.

```java
toBinary(n){ // algorithm assumes input n >= 1
    if n >= 1{ // otherwise base case, and do nothing
        print n % 2
        toBinary( n/2 )
    }
}
```

Note that the above algorithm prints \( b_0, b_1, \ldots, b_{m-1} \) in that order.

We discussed back in the lecture 2 that the iterative version of the algorithm loops about \( \log_2 n \) times. For the same reason, the recursive version has about \( \log_2 n \) recursive calls, one for each bit of the binary representation of the number.

In this lecture, we’ll look at another algorithm that runs in \( O(\log_2 n) \) time. This algorithm is closely related to the binary conversion algorithm, as we will discuss later.

Binary search in a sorted (array) list

Suppose we have an array list of \( n \) elements which are already sorted from smallest to largest. These could be numbers or strings sorted alphabetically. Consider the problem of searching for a particular element in the list, and returning the index in \( 0, \ldots, n-1 \) of that element, if it is present in the list. If the element is not present in the list, then we return -1.

One way to do this would be to scan the values in the array, using say a `while` loop. In the worst case that the value that we are searching for is the last one in the array, we would need to scan the entire array to find it. This would take \( n \) steps. Such a linear search is wasteful since it doesn’t take advantage of the fact that the array is already sorted.

A much faster method, called binary search takes advantage of the fact that the array is sorted. You are familiar with this idea. Think of when you look up a word in an index in the back of a book. Since the index is sorted alphabetically, you don’t start from the beginning and scan. Instead, you jump to somewhere in the middle. If the word you are looking for comes before those on the page, then you jump to some index roughly in the middle of those elements that come before the one you jumped to, and otherwise you jump to a position in the middle of those that come after the one you jumped to. The binary search algorithm does essentially what I just described.

Let’s first present an iterative (non-recursive) version of binary search algorithm.

---

1Of course, if you are looking for a word that starts with ”b”, then you don’t just into the middle, but rather you start near the beginning. But let’s ignore that little detail here.
binarySearch(list, value){  // iterative
  low = 0
  high = list.size - 1
  while low <= high {
    mid = low + (high - low)/ 2  // so mid == low, if high - low == 1
    if list[mid] == value
      return mid
    else{ if value < list[mid]
          high = mid - 1  // high can become less than low
        else
          low = mid + 1 }
  }
  return -1  // value not found
}

For each pass through the while loop, the number of elements in the array that still need to be examined is cut by at least half. Specifically, if [low, high] has an odd number of elements (2k+1), then the new [low, high] will have k elements. If [low, high] has an even number of elements (2k), then the new [low, high] has either k or k-1 elements. Note that the new [low, high] does not contain the mid element.

It follows that for an input array with n elements, there are about log₂n passes through the loop. This is the same idea as converting a number n to binary, which takes about log₂n steps to do, i.e. the number of times we can divide n by 2 until the answer is 0.

The details of the algorithm are a bit tricky and it is common to make errors when coding this up. e.g. There are two inequality tests: one is ≤ and one is <, and the way the low and high variables are updated is also rather subtle. For example, note that the while loop exits when high < low which indeed can and does occur.

Also note I have updated the mid value by using low + (high-low)/2 rather than (low+high)/2. The two are equivalent, but I find the former is easier to think about, vis-a-vis off-by-one errors and integer division (which rounds down).

binarySearch( list, value, low, high ){  // recursive
  if low > high {
    return -1
  else{
    mid = low + (high - low) / 2
    if value == list[mid]
      return value
    else if value < list[mid]
      return binarySearch(list, value, low, mid - 1)
    else
      return binarySearch(list, value, mid+1, high)
  }
}

Notice how the iterative and recursive versions of the algorithm are nearly identical.
Tail Recursion

The two algorithms I’ve mentioned today (converting to binary, and binary search) have iterative and recursive versions which seem to be nearly identical. For these algorithms, you will note that the recursive call for the recursive version is the last thing that the algorithm does before exiting. This is no accident. As it turns out (you’ll have to take my word for this now), whenever the recursive call is the last thing a recursive algorithm does, it is straightforward to convert the algorithm to a non-recursive algorithm. This situation is called tail recursion. You will learn more about it in COMP 302.

```java
factorial(n){ // algorithm assumes argument: n >= 1
    if (n == 1)
        return 1
    else
        return n * factorial(n - 1)
}
```

Note that the factorial algorithm above is not tail recursive (even though it can be turned relatively easily into a non-recursive algorithm). The reason it isn’t tail recursive is that the last thing the algorithm does is the multiplication, after the recursive call to `factorial(n-1)` has returned its value.

Recursion and the Call Stack

Back in lecture 7, we discussed stacks and I mentioned the ”call stack”. Each time a method calls another method (or a method calls itself, in the case of recursion), the computer needs to do some administration to keep track of the “state” of method at the time of the call. This information is called a “stack frame”.

For example, suppose the program calls `factorial(6)`. This leads to a sequence of recursive calls and subsequent returns from these calls. For example, right before returning from the `factorial(3)` call, we have made the following sequence of calls and returns:

`factorial(6), factorial(5), factorial(4), factorial(3), factorial(2), factorial(1), return from factorial(1), return from factorial(2)`

the call stack looks like this,

- frame for `factorial(3)`: [factn = 6, n=3] <---- top of stack
- frame for `factorial(4)`: [factn = 0, n=4]
- frame for `factorial(5)`: [factn = 0, n=5]
- frame for `factorial(6)`: [factn = 0, n=6] <---- bottom of stack

Using the Eclipse debugger and setting breakpoint within the `factorial()` method, you can see how the stack evolves. I strongly recommend that you do this and verify how this works.

```
http://www.cim.mcgill.ca/~langer/250/TestFactorial.java
```

I would do the same for the Tower of Hanoi.

```
http://www.cim.mcgill.ca/~langer/250/TestTowerOfHanoi.java
```

See slides for screen shots.
Stack frames

The call stack is not some abstract idea, but rather it is a real data structure that the software that runs your program (called the "Java Virtual Machine") uses to keep track of the methods.

The stack consists of *stack frames*, one for each method that is called. In the case of recursion, there is one stack frame for each time the method is called. The stack frame contains all the information that is needed for that method. This includes local variables declared and used by that method, parameters that are passed to the method, and information about where the method returns when it is done, that is, who called the method.

You will learn about the call stack and stack frames work in much more detail in COMP 273. I mention it here because I want you to get familiar with the idea, and because I want you to be aware that the call stack and stack frame really exist. Indeed most decent IDEs will allow you to examine the call stack (see slides) and at least the current stack frame, i.e. the frame on top of the call stack.