Converting a number to its binary representation

Back in lecture 2, we saw how to convert a decimal number $n \geq 1$ to binary. Here I'll write the algorithm slightly differently so we are printing out the bits from low to high.

```
toBinary(n){  //  iterative
    i = 0
    while n > 0 {
      print  n % 2     //  bit i
      n = n/2
      i = i+1
    }
}
```

Next we write the algorithm recursively.

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

Note that this indeed prints the bits from the lowest order to highest order. However, if we were two switch the print statement with the `toBinary` call, then we would print from highest order to low. In that case, the printing wouldn't start until we have exited the `toBinary(n)` where `n` is either 1 or 2.

Finally, recall from lecture 2 that the iterative version of the algorithm loops about $\log_2 n$ times, where n is the original number. 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. Today we will see a few more algorithms that have an $O(\log_2 n)$ property.

Power (x^n)

Let x be some number (say a positive integer for simplicity) and consider how to compute x raised to some power n. An iterative (non-recursive) method for doing it is:

```
power(x,n){    //  iterative
    result = 1
    for i = 1 to n
       result = result * x
    return  result
}
```

A more interesting way to compute x^n is to use recursion. We could do what we did last lecture and rite $x^n = x^{n-1}x$, but let's do it in a way that is more interesting. Suppose we wish to compute x^{18}. We can write

$$x^{18} = x^9 * x^9$$

So, to evaluate x^{18}, we could evaluate x^9 and then perform *only one more* multiplication, i.e. $x^9 * x^9$. But how do we evaluate x^9 ? Because 9 is odd, we cannot do the same trick exactly. Instead we compute

$$x^9 = (x^4)^2 * x.$$

Thus, *once we have* x^4, two further multiplications are required to compute x^9.

Breaking it down again, we write $x^4 = (x^2)^2$ and so we have

$$x^{18} = (((x^2)^2)^2 * x)^2.$$

Thus we see that we need a total of 5 multiplications (4 squares and one multiplication by x).

```
power(x,n){    //  recursive
   if n == 0
        return 1
   else if n == 1
      return x
   else{
       tmp = power(x,n/2)
       if n is even
          return  tmp*tmp
       else
          return  tmp*tmp*x
    }
}
```

The number of multiplications that is executed for each recursive call will be either 1 or 2, depending on whether the n parameter passed in that call is even or odd. Note the similarity to converting a number (in this case 18) to binary where we decide if each bit is either 0 or 1. (In the slides, I used 243 as an example.)

Notice that the number of recursive calls will be $O(\log_2 n)$, for the same reason as in the decimal-to-binary algorithm, namely each call divides n by 2 and so the number of calls is the number of times you can divide the original n by 2 until you get to 0.

For the same reason, the number of multiplies that are executed will be between $\log_2 n$ and $2 \log_2 n$. Why? If the original n's binary representation has all 1's, e.g. $63 = (111111)_2$, then two multiplications will be executed at each recursive call since the parameter n will always be odd. If the original n's binary representation is all 0's (except the high order bit), e.g. $n = (100000000)_2$, then only one multiplication will be computed at each recursive call. [ASIDE: Note that the highest order bit is a 1. This bit corresponds to the base case of the recursion, and no multiplication is computed in that case. In the Sec. 001 lecture, I mistakenly forgot about that.]

As I discussed in the lecture slides and I will discuss in an exercise, we should not be fooled by the above argument into thinking that we can compute x^n in time $O(\log_2 n)$. The issue here is that we have only discussed the number of multiplications. But the time taken for a multiplication depends on the number of digits in the two numbers being multiplied. It turns out that the recursive algorithm – although quite interesting – takes just as long as the if we compute x^n by successive multiplications by x. But as I as briefly will mention in the exercise, we can use this recursion trick to compute something similar which is very useful, and which we can indeed compute in time $O(\log_2 n)$.

Let's now turn to a different problem, which also is $O(\log_2 n)$ for similar reasons as what we've discussed today.

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.[1] 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.

```
binarySearch(list, value){          // iterative
    low = 0
    high = list.size - 1
    while  low <= high  {
       mid = (low + high)/ 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
}
```

[1]Of 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.

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, or less than half. If `[low, high]` has an even number of elements (2k), then the new `[low, high]` has either k or k-1 elements, whih is at most half. 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 $\log_2 n$ passes through the loop. This is the same idea as converting a number n to binary, which takes about $\log_2 n$ steps to do, i.e. the number of times we can divide n by 2 until we get 0.

The details of the above algorithm are a bit tricky and it is common to make errors when coding it up. There are two inequality tests: one is $\leq$ and one is $<$, and the way the `low` and `high` variables are updated is also rather subtle. For example, note that if the item is not found then the while loop exits when `high < low`. There is no explicit check for the case that `high == low`.

Here is a recursive version. Notice how the iterative and recursive versions of the algorithm are nearly identical. One difference is that the recursive version passes in the `low` and `high` variables.

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

If you were to implement this in Java, you probably wouldn't pass the `list` in as parameter since that would mean you were creating many new lists as you go. Let's not concern ourselves with this issue now, since I want you to focus here more on how `low` and `high` work and how the size of the list gets chopped in half each time. These are the main points to understand.

Today we looked at a few problems that all had $O(\log_2 n)$ behavior. The base 2 of the logarithm is due to chopping something in *half*, whether it is a number we are converting to binary or the exponent of a power, or the size of a sorted list. It is nice to see problems that having some similarities in their behavior but that also have some subtle differences. Next class we will look at another way in which we can chop a problem in half and we will see that a $\log_2 n$ factor arises there as well.