

Today we will look at three recursive algorithms which have a similar behavior, namely they all require time proportional to  $\log n$  to compute. (I am presenting these algorithms within the broad topic of recursion, though some of the algorithms just as easily be expressed non-recursively.)

### Twenty questions (with numbers), and decimal-to-binary conversion

Let's play a game. I am thinking of a number  $n$  between 0 and  $2^{20} - 1$  (inclusive). Your task is find this number, by asking a sequence of 20 questions that have yes/no answers. How would you do it? Easy. Your  $i^{\text{th}}$  question is, "does bit  $b_i$  of the binary representation of  $n$  have value 1?" Once I give you all the bits, you can convert from binary to decimal and get the number  $n$ . (Note that for me to answer your questions, I need to convert  $n$  from decimal to binary.)

We have seen the algorithm for converting a decimal number  $n$  to binary. We can rewrite this algorithm so that it is recursive.

---

#### Algorithm: DecimalToBinary( $n$ )

**Input:** a decimal number  $n$

**Output:** bit sequence  $b_i$  from least ( $i = 0$ ) to most significant, representing  $n$  in binary

```

if  $n \geq 1$  then
    print  $n \% 2$ 
    DecimalToBinary(  $n / 2$  )
end if

```

---

Note that if you switch the order of the print and recursive call, then you will print the bits in the opposite order, namely from most significant to least significant. (i.e. the first print only occurs after you have done all the recursive calls and the parameter  $n$  has been reduced to 0.)

The number of steps needed to compute decimal to binary has the same dependence on  $n$  whether one uses a recursive or non-recursive algorithm, namely about  $\log_2 n$ . The reason is that this is about the number of bits that are needed to represent  $n$  in binary.

To be precise, a positive integer  $n$  requires  $\text{floor}(\log_2 n) + 1$  bits. This can be easily seen with the following derivation. Let  $m$  be the number of bits needed to represent  $n$ , that is,

$$n = b_{m-1}2^{m-1} + b_{m-2}2^{m-2} + \dots b_12 + b_0$$

where  $b_{m-1} = 1$ . Since each of these  $b_i$  is either 0 or 1, we have

$$\begin{aligned} n &\leq 2^{m-1} + 2^{m-2} + \dots + 2 + 1 \\ &= 2^m - 1 \\ &< 2^m \end{aligned}$$

Taking the log (base 2) of both sides gives:

$$\log n < m$$

We also have a lower bound

$$n \geq 2^{m-1}$$

and so

$$\log n \geq m - 1.$$

Noting that  $m$  is an integer, it follows immediately that  $m - 1 = \text{floor}(\log_2 n)$  and so

$$m = \text{floor}(\log_2 n) + 1.$$

## Power ( $x^n$ )

Our next example looks very different at first glance, but we will see that it is in fact very similar. Consider an algorithm for computing  $x$  raised to some power  $n$ . An iterative (non-recursive) method for doing it is:

---

### Algorithm: Power( $x, n$ )

**Input:** a real number  $x$  and a positive integer  $n$

**Output:**  $x^n$

```

result ← 1
for  $i = 1$  to  $n$  do
    result ← result *  $x$ 
end for
return result

```

---

A faster way to compute  $x^n$  is to use recursion. For example, 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.

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  $x$ ).

The recursive algorithm is shown on the next page. Notice that the number of recursive calls will be about  $\log n$ , 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.

The number of instructions (or, specifically, the number of multiplications) that is executed for each recursive call depends on the original  $n$ . The reason is that the number of multiplications in any single call will be either 1 or 2, depending on whether the  $n$  parameter passed in that call is even or odd. For example, if the original  $n$ 's binary representation has all 1's, e.g.  $63 = (111111)_{two}$ , then two multiplications will be executed at each recursive call since the parameter  $n$  will always be odd.

**Algorithm: power(x,n) – recursive****Input:** a real number  $x$  and a non-negative integer  $n$  (incl. 0)**Output:**  $x^n$ 

```
if  $n = 0$  then
    return 1
else
     $tmp \leftarrow \text{power}(x, n/2)$ 
    if  $n$  is even then
        return  $tmp * tmp$ 
    else
        return  $tmp * tmp * x$ 
    end if
end if
```

---

**Binary search in a sorted array**

Let's now look at a third problem in which this  $\log n$  behavior appears. Suppose we have an *array* of elements which are sorted from smallest to largest. These could be numbers or strings sorted alphabetically as in last names in a phone book. Now we would like to search for a particular element (number or string) and return the location (index) of that value in the array. If that value is not present in the array, then it should return the index -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 method is called *linear search*.

A much faster way to search takes advantage of the ordering of the elements. You are familiar with this idea. Think of when you look up a phone number in a telephone book. You don't start from the first page and scan. Instead, you pick a page somewhere in the middle. If the name you are looking for comes before those on the page, then you continue your search in only pages that come before the chosen one, otherwise you continue your search in the pages that come after the chosen one. The *binary search* algorithm is similar to this.

Each time the recursion is called, the number of elements in the array that still need to be examined is cut approximately in half. I say "approximately" because if  $[low, high]$  has an odd number of elements, then we cannot cut this odd number exactly in half. (Note:  $(low + high)/2$  is an integer division, and so the remainder is ignored.) For an input array with  $n$  elements, there are approximately  $\log_2 n$  recursive calls.

The algorithm is shown on the next page. Several observations can be made. First, if  $a[mid] == v$  before  $low == high$ , then the algorithm keeps going recursively, even though we have found the element.<sup>1</sup> Second, if the element appears more than once in the array, then algorithm will return the index of one of elements but not all. Third, if the original number of elements is a power of 2, then the search can be interpreted in terms of the binary representation of the indices, namely each

---

<sup>1</sup>This seems inefficient, at first glance. However, it is not so bad. In order to make the algorithm stop when  $a[mid] == v$ , you would need to test for this explicitly. And you would need to do this explicit test every time the recursion is called. This would be extra work and would cancel out (to some extent) the savings you would get by stopping the recursion early.

**Algorithm:** `binarySearch`( $a, v, low, high$ )

**Input:** array  $a$ , value  $v$ , lower and upper bound indices  $low, high$  ( $low = 0, high = n - 1$  initially)

**Output:** the index  $i$  of element  $v$  (if it is present), -1 (if  $v$  is not present)

```

if  $low == high$  then
  if  $a[low] == v$  then
    return  $low$ 
  else
    return  $-1$ 
  end if
else
   $mid \leftarrow (low + high)/2$ 
  if  $v \leq a[mid]$  then
    return binarySearch( $a, v, low, mid$ )
  else
    return binarySearch( $a, v, mid + 1, high$ )
  end if
end if

```

decision of which recursive call to make amounts to choosing the next bit of the binary representation of the index (from most significant to least significant). (See example below.)

In the example below, suppose  $n = 16$  and the item we are searching for happens to be at index 5. The succession of  $low, high$  values will be  $(0,15), (0,7), (4,7), (4,5), (5,5)$ . See figure below, with the low and high values marked L and H. In the first call, you determine that  $b_3 = 0$ . In the second call you determine that  $b_2 = 1$ . In the third call, you determine that  $b_1 = 0$ . And the final call determines that bit  $b_0 = 1$ .

0	0000	L	L		
1	0001				
2	0010				
3	0011				
4	0100		L	L	
5	0101			H	L=H
6	0110				
7	0111	H	H		
8	1000				
9	1001				
10	1010				
11	1011				
12	1100				
13	1101				
14	1110				
15	1111	H			

Note that this interpretation of the algorithm only makes sense if the original  $n$  is a power of 2.