RECURSION

Suppose you have a sequence of numbers such as the powers of two, $2^k$: 1, 2, 4, 8, 16, … for $k = 0, 1, 2, 3, 4, \ldots$ Each number is called a term of the sequence and has an ordinal position. The 0th term is 1, the 1st term is 2, the 2nd term is 4, etc.

Recursion is a computational technique whereby you compute each term of a sequence by somehow manipulating one or more previous terms. The nonrecursive computation of $2^k$ is to multiply 2 by itself $k-1$ times. Thus, $2^4 = 2 \times 2 \times 2 \times 2 = 16$. The recursive computation of $2^k$ is to double the previous term: $2^k = 2 \times 2^{k-1}$. Thus, $2^4 = 2 \times 2^3 = 2 \times 8 = 16$.

In mathematics, a formula that expresses a computation recursively is called a recurrence relation. In computer science a routine that does so is called a recursive routine. Both have a similar structure, consisting of two parts:

1. One or more initial conditions that define the initial terms of the sequence.
2. A recursive clause that defines the next term of the sequence using computations on one or more previous terms. This clause is self-referential, meaning that the formula defines itself in terms of itself.

Below is the computation of the powers of a number $x$ expressed as a recurrence relation and the recursive routine in Java that computes the same result:

\[
egin{array}{c}
x^n = x \cdot x^{n-1} \\
x^1 = x \\
x^0 = 1
\end{array}
\]

\[
\text{double power ( double x, int n )}
\{
    \text{if ( n == 0 )} \quad // 1 \\
    \quad \text{return 1;} \quad // 2 \\
    \text{else if ( n == 1 )} \quad // 3 \\
    \quad \text{return x;} \quad // 4 \\
    \text{else} \quad // 5 \\
    \quad \text{return x * power( x, n - 1 );} \quad // 6
\}
\]

The sequence of computations invoked by recursion can be likened to the action of a yo-yo. The recursive clause of the relation or routine is repeatedly invoked, thus stretching the computation out into smaller and smaller pieces. The initial condition is finally reached giving a partial solution. That partial solution is combined with the previous small piece to give a bigger solution. This process works backwards until ultimately giving the final solution.
To illustrate, let's follow the computation of $2^5$ using the recurrence relation:

\[
2^4 = 2 \cdot 2^3 \quad \text{recursive clause}
\]
\[
= 2 \cdot 2^2 \quad \text{recursive clause}
\]
\[
= 2 \cdot 2 \cdot 2^1 \quad \text{recursive clause}
\]
\[
= 2 \cdot 2 \cdot 2 \quad \text{initial condition}
\]
\[
= 2 \cdot 2 \cdot 4
\]
\[
= 2 \cdot 8
\]
\[
= 16
\]

And by following the sequence of calls of the recursive routine `power`

\[
p = \text{power}(2, 4)
\]
\[
\text{power} ( x, n ) \quad \text{return} \quad 2 \cdot \text{power} ( 2, 3 )
\]
\[
\quad \quad \text{power} ( x, n ) \quad \text{return} \quad 2 \cdot \text{power} ( 2, 2 )
\]
\[
\quad \quad \quad \quad \text{power} ( x, n ) \quad \text{return} \quad 2 \cdot \text{power} ( 2, 1 )
\]
\[
\quad \quad \quad \quad \quad \text{power} ( x, n ) \quad \text{return} \quad 2
\]
\[
\quad \quad \quad \quad \quad \quad \quad \text{return} \quad 2 \cdot 2
\]
\[
\quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \text{return} \quad 2 \cdot 4
\]
\[
\quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \text{return} \quad 2 \cdot 8
\]
\[
p = 16
\]

To follow the sequence of recursive calls, remember that every call to a recursive routine is invoking the routine afresh. Thus, you get new copies of the routine's parameters and local data (pushed onto the run-time stack) and a new return address. When the routine ends, it returns to the calling point, which may be within the previous execution of the recursive routine. The "last in" copies of the parameters and local data are popped from the run-time stack making the previous copies available again.
**Factorial**

A well-known recurrence relation is that of the *factorial* function. This function is often taught in its nonrecursive form:

\[ n! = n \times (n-1) \times (n-2) \times \ldots \times 2 \times 1 \]

So that:

\[ 4! = 4 \times 3 \times 2 \times 1 = 24 \]

Here is its definition using a recurrence relation and recursive routine:

<table>
<thead>
<tr>
<th>( n! = n \times (n-1)! )</th>
<th>( 0! = 1 )</th>
</tr>
</thead>
</table>
| \begin{align*} \text{long factorial}( \text{int } n) \{} \\
| \quad \text{if ( } n == 0 \) \} & \quad \text{// 1} \\
| \quad \text{return 1;} & \quad \text{// 2} \\
| \quad \text{else} & \quad \text{// 3} \\
| \quad \text{return } n \times \text{factorial}( n-1 ); & \quad \text{// 4} \\
| \\end{align*} |

We illustrate the computation of 4! using the recurrence relation:

\[ 4! = 4 \times 3! \quad \text{recursive clause} \]
\[ = 4 \times 3 \times 2! \quad \text{recursive clause} \]
\[ = 4 \times 3 \times 2 \times 1! \quad \text{recursive clause} \]
\[ = 4 \times 3 \times 2 \times 1 \times 0! \quad \text{recursive clause} \]
\[ = 4 \times 3 \times 2 \times 1 \times 1 \quad \text{initial condition} \]
\[ = 4 \times 3 \times 2 \times 1 \]
\[ = 4 \times 6 \]
\[ = 24 \]
And the recursive routine:

\[
f = \text{factorial}(4)
\]

\[
\text{factorial}(n)
\]

\[
\begin{align*}
\text{return} & \quad 4*\text{factorial}(3) \\
\text{factorial}(n) & \quad \text{return} \quad 3*\text{factorial}(2) \\
\text{factorial}(n) & \quad \text{return} \quad 2*\text{factorial}(1) \\
\text{factorial}(n) & \quad \text{return} \quad 1*\text{factorial}(0) \\
\text{factorial}(n) & \quad \text{return} \quad 1 \\
\end{align*}
\]

\[
\text{return} \quad 1*1 \\
\text{return} \quad 2*1 \\
\text{return} \quad 3*2 \\
\text{return} \quad 4*6
\]

\[
f = 24
\]
O(\lg n) Power
The power function calculating \(x^n\) given earlier runs in \(O(n)\) time. You can justify that by considering that each recursive call takes the computation one step closer to the final answer. For example in the computation of \(\text{power}(2, 4)\) given above, \(\text{power}(2, 4)\) calls \(\text{power}(2, 3)\) calls \(\text{power}(2, 2)\) calls \(\text{power}(2, 1)\). Four calls to the routine followed by four returns and three multiplications. The total number of computation steps is \(9 = 2n + 1\). Thus, the number of computations steps is \(O(n)\).

We can improve the performance of the recursive power routine by doing some of the computation as the recursion expands rather then waiting to do it all as the recursion returns. The following routine is \(O(\lg n)\).

\[
x^n = (x^2)^{n/2}
\]
\[
x^0 = 1
\]

\begin{verbatim}
double power( double x, int n )
{
    if ( n == 0 )
        return 1;
    else if ( n == 1 )
        return x;
    else if ( n % 2 == 0 ) // n even
        return power( x * x, n / 2 );
    else // n odd
        return x * power( x, n - 1 );
}
\end{verbatim}

Here is a trace of the algorithm for \(n=16\):

\[
p = \text{power}(2, 16)
power( x, n )
return \text{power}(4, 8)
power( x, n )
return power(16, 4)
power( x, n )
return power(256, 2)
power( x, n )
return power(65536, 1)
power( x, n )
return 65536
return 65536
return 65536
return 65536
return 65536
p = 65536
\]
You can see that the number of computation steps increases by 2 when \( n \) is doubled. Thus, the routine has an \( O(\lg n) \) running time.

Euclid’s Algorithm

<table>
<thead>
<tr>
<th>Euclid’s Algorithm</th>
</tr>
</thead>
<tbody>
<tr>
<td>( \text{GCD}(a, b) = \text{GCD}(b, a % b) )</td>
</tr>
<tr>
<td>( \text{GCD}(a, 0) = a )</td>
</tr>
</tbody>
</table>

```c
int gcd( int a, int b )
{
    if ( b == 0 )
        return a;
    else
        return gcd( b, a % b );
}
```
Inefficient Recursion
Your textbook makes the point that calling a routine requires some memory and run-time overhead. Memory is needed for the routine's parameters, local variables and return address. Execution time is used to push the parameters, return address and memory locations onto the run-time stack. This overhead, however, is not significant enough to avoid using recursion unless each recursive call results in a large amount of data being pushed onto the run-time stack.

Moreover, your textbook makes the point that recursion is used because it simplifies a problem conceptually. Usually this simplicity is worth a little bit of run-time overhead.

Sometimes, however, a recursive solution is a grossly inefficient one. This happens not because of the extra run-time overhead but because the recursive solution results in the same computations being done over and over again. To illustrate, consider the Fibonacci sequence. Each term is the sum of the previous two terms: 0, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, … Here is the recurrence relation and recursive routine:

<table>
<thead>
<tr>
<th>Fibonacci Sequence</th>
</tr>
</thead>
<tbody>
<tr>
<td>F(n) = F(n−1) + F(n−2)</td>
</tr>
<tr>
<td>F(0) = 0</td>
</tr>
<tr>
<td>F(1) = 1</td>
</tr>
</tbody>
</table>

```c
long fibonacci( long n )
{
    if ( n == 0 || n == 1 )
        return n;
    else
        return fibonacci(n−1) + fibonacci(n−2);
}
```

A useful device for tracing the calls of this recursive computation is a recursion tree. The recursion tree on the next page shows the computation of Fibonacci(6). The number inside each circle represents the value of the parameter n in the given call of the routine; the number above the circle represents the value returned by the call. Look how many times the computation repeats the calls to Fibonacci(1) and Fibonacci(0). It is that repeated calculation of the same intermediate result that makes this algorithm perform so poorly.

Just how poorly? A call to Fibonacci(n) runs in exponential time. You can justify this by considering the growth of the recursion trees. Each node represents three computation steps – recursive call, + and return. The tree for Fibonacci(2) has 3 nodes; Fibonacci(3) has 5 nodes; Fibonacci(4) has 5 + 3 + 1 = 9 nodes; Fibonacci(5) has 9 + 5 + 1 = 15 nodes; Fibonacci(6) has 15 + 9 + 1 = 25 nodes and so on. Increasing the argument to Fibonacci by 1, multiplies the number of computation steps by roughly \( \frac{1}{2} \). Thus, the routine's running time is \( O(1.6^n) \).
Recursion