## Recursion

### Example: $n!$

<table>
<thead>
<tr>
<th>Iterative Version</th>
<th>Recursive Version</th>
</tr>
</thead>
<tbody>
<tr>
<td>read $n$</td>
<td>function FACT($n$)</td>
</tr>
<tr>
<td>$x := 1$</td>
<td>if $n \leq 1$</td>
</tr>
<tr>
<td>for $i := 2$ to $n$ do $x := x \ast i$</td>
<td>then return 1</td>
</tr>
<tr>
<td>write $x$</td>
<td>else return $n \ast FACT(n-1)$</td>
</tr>
<tr>
<td></td>
<td>end</td>
</tr>
</tbody>
</table>

### Idea:

- Recursion works so long as we can count on the compiler to generate code that saves variable values when a recursive call is made and restores them when control returns.
- So from our point of view, the recursive call might as well be to a completely different piece of code that just happens to have the same name.
Example: Recursive Binary Search

We wish to determine a position in the sorted array $A[a] ... A[b]$ where a given value $x$ occurs, or return $-1$ if $x$ is not found.

```plaintext
function RBS(a, b)
    if $a < b$ then begin
        $m := \lfloor (a+b)/2 \rfloor$
        if $x \leq A[m]$ then return RBS(a, m) else return RBS(m+1, b)
    end
    else if $x = A[a]$ then return $a$
    else return $-1$
end
```

**Time:** $O(\log(n))$ time.

**Space:** Like factorial, this is an example of tail recursion. Space may be $O(1)$, or as much as $O(\log(n))$, depending on how well the compiler implements "hidden" bookkeeping.
Example: Search a Linked List

To avoid the list header, define a function that searches from a vertex forward and then call it on the first vertex of the list.

function SEARCHTAIL(d,v,L)
    if v=nil then return nil
    else if d=DATA(v) then return v
    else return SEARCHTAIL(d,NEXT(v),L)
end

function SEARCH(d,L)
    SEARCHTAIL(d,FIRST(L),L)
end
Example: Sum Vertices $v$ Through $w$ of a Linked List

function SUM($v, w, L$)
  if $v = w$
    then return DATA($w$)
  else return DATA($v$) + SUM(NEXT($v$), $w, L$)
end
Example: Reverse a Singly Linked List

procedure REVERSE(L)
    if (SIZE(L)>1) then begin
        d := DELETE(FIRST(L),L)
        REVERSE(L)
        INSERT(d,LAST(L),L)
    end
end
Example: Fast Computation of Integer Exponents

**Idea:** \( x^i \) can be expressed in terms of \( x^{i/2} \):

\[
x^i = \begin{cases} 
(x^{i/2})^2 & \text{if } i \text{ is even} \\
 x(x^{(i-1)/2})^2 & \text{if } i \text{ is odd}
\end{cases}
\]

The equivalent recursive program is:

```plaintext
function POWER(x, i):
    if i = 0 then return 1
    else if i is even then return POWER(x, i/2)^2
    else return x*POWER(x, (i−1)/2)^2
end
```
(integer exponents continued)

**Equivalent iterative program:**

**Idea:** If $x$ has $k$ bits, then the recursive call processes the first $k-1$ bits, then that value is squared, and finally it is multiplied by $x$ if the $k^{th}$ bit is 1. By "unwinding" these recursive calls, we see that we want to visit the bits of $x$ from left to right (high order to low order), and at each iteration, square and then multiply by $x$ if the bit is one. We initialize the value $v$ to 1. On the first iteration squaring does nothing, and then since the leading bit of $x$ is always 1, this first iteration simply has the effect of doing $v = x$.

```plaintext
function iterativePOWER(x,i):
    Let $s[1]...s[m]$ be the bits of $i$ written left to right in binary.
    $v := 1$
    for $k := 1$ to $m$ do begin
        $v := v^2$
        if $s[k] = 1$ then $v := x*v$
    end
    return $v$
end
```
Time: Both the recursive and iterative versions perform at most 2 multiplications per bit in the binary representation of $i$. Hence the time is $O(\log(i))$. This analysis makes the standard unit-cost assumption that a multiplication can be done in $O(1)$ time, which may not be realistic as $i$ gets large. In any case, $O(\log(i))$ multiplications is better than the $O(i)$ multiplications that the straightforward approach uses.