Recursive Definition of a Set

To define a set S recursively, we specify

-- one or more initial elements of S
-- a rule for constructing new elements of S given those already in the set

Example:

$3 \in S$
If $x, y \in S$, then $x + y \in S$

Recursive Definition of a Set

To define a set S recursively, we specify

-- one or more initial elements of S
-- a rule for constructing new elements of S given those already in the set

Example:

$3 \in S$
If $x, y \in S$, then $x + y \in S$
Nothing else is in S. (Extremal or limiting clause)

Recursive Definition of a Set

To define a set S recursively, we specify

-- one or more initial elements of S
-- a rule for constructing new elements of S given those already in the set

Example:
S is the smallest set such that
$3 \in S$
If $x, y \in S$, then $x + y \in S$

Recursive Definition of a Set

To define a set S recursively, we specify

-- one or more initial elements of S
-- a rule for constructing new elements of S given those already in the set

Example:

$3 \in S$
If $x, y \in S$, then $x + y \in S$

What is S?

Recursive Definition of a Set

To define a set S recursively, we specify

-- one or more initial elements of S
-- a rule for constructing new elements of S given those already in the set

Example:

p, q are WFF's

If x is a WFF, then $\neg x$ is a WFF.

If x and y are WFF's, then

$(x \vee y)$, $(x \wedge y)$, $(x \rightarrow y)$, and $(x \leftrightarrow y)$ are WFF's

Recursive Definition of a Set

Example:

The set of binary numbers with more 0's than 1's

0
00
000
001
010
100
0000
0001
0010
0100
1000

Proving Properties of Recursively Defined Sets

$3 \in S$
If $x, y \in S$, then $x+y \in S$

Prove: If $w \in S$, then $3 \mid w$.

Every element of S is either the base case or is produced by some number of applications of the recursive rule.

We can do a proof by strong induction on the number of applications of the rule to show that for $n \geq 1$, if $w \in S$ is produced by n applications, then $3 \mid w$.

The main argument: if w is produced by k + 1 applications of the recursive rule, then $w = x + y$ where x and y are produced by $i \leq k$ applications. Since these are covered by the inductive hypothesis, $3 \mid x$, $3 \mid y$, and so $3 \mid w$.

Proving Properties of Recursively Defined Sets

Principle of Structural Induction (Generalized Induction)

Suppose S is a set defined by a set of base cases B and a set of recursive rules R, and that P(s) is a proposition involving a member s of S.

If
P(s) is true for every $s \in B$,
and
P(s) is true whenever s is produced, using a rule from R, from members $t_1, \ldots, t_n$ of S for which $P(t_1) \ldots P(t_n)$ are true,
then
P(s) is true for every $s \in S$.

If the primitive elements have the property, and if building new elements preserves the property, then all elements have the property.

Proving Properties of Recursively Defined Sets

$3 \in S$
If $x, y \in S$, then $x + y \in S$

Prove: If $w \in S$, then $3 \mid w$.

We will show that P(s): $3 \mid s$ is true for all $s \in S$.

BASE CASE: $3 \mid 3$, so P(3) is true.

INDUCTIVE STEP: We need to show that applying the recursive rule preserves the property P.

Suppose $x, y \in S$, $3 \mid x$, $3 \mid y$, and $w = x + y$.

Proving Properties of Recursively Defined Sets

$3 \in S$
If $x, y \in S$, then $x + y \in S$

Prove: If $w \in S$, then $3 \mid w$.

We will show that P(s): $3 \mid s$ is true for all $s \in S$.

BASE CASE: $3 \mid 3$, so P(3) is true.

INDUCTIVE STEP: We need to show that applying the recursive rule preserves the property P.

Suppose $x, y \in S$, $3 \mid x$, $3 \mid y$, and $w = x + y$.
Then by Theorem 32.1, $3 \mid w$, so P(w) is true.

Thus applying the recursive rule preserves the property, and all members of S are divisible by 3.

The set S of binary numbers with more 0's than 1's

0
00
000
001
010
100
0000
0001
0010
0100
1000

S {
$0 \in S$
If $r, s \in S$, then the following are in S:
rs
rs1
r1s
1rs

Recursive Functions

A recursive function is one that does some of the work in such a way as to reduce the problem to a simpler one of exactly the same form. Ultimately the problem becomes so simple that it can be solved directly.

Since the subproblem is of the same form as the original, the function can call itself to solve it.

Recursive Functions

A recursive function is one that does some of the work in such a way as to reduce the problem to a simpler one of exactly the same form. Ultimately the problem becomes so simple that it can be solved directly.

Since the subproblem is of the same form as the original, the function can call itself to solve it.

The following need to be true:

> There must be some base condition for which the subprogram does not call itself.
>
> Each time the subprogram calls itself, it must converge on the base condition.

Pseudocode Example

To print the characters of a string:

```
function PrintString(string s)
{

}
```

Pseudocode Example

To print the characters of a string:

```
function PrintString(string s)
{
     if (Length(s) = 0) then return      base case
     printchar(First(s))                 do some work
     PrintString(Rest(s))                recursive call on simpler problem
}
```

where

- Length is a function that returns the length of a string
- printchar is a system routine that prints a character
- First is a function that returns the first character of a string
- Rest is a function that returns a string with the first character removed

Pseudocode Example

What does this do?

```
function PrintString(string s)
{
     if (Length(s) = 0) then return      base case
     PrintString(Rest(s))                recursive call on simpler problem
     printchar(First(s))                 do some work
}
```

Pseudocode Example

What does this do?

```
function PrintString(string s)
{
 if (Length(s) = 0) then return          base case
     printchar(First(s))                 do some work
     PrintString(Rest(s))                recursive call on simpler problem
     printchar(First(s))                 do some work
}
```

Pseudocode Example

Back to the original version:

```
function PrintString(string s)
{
     if (Length(s) = 0) then return      base case
     printchar(First(s))                 do some work
     PrintString(Rest(s))                recursive call on simpler problem
}
```

Tail recursion : the only recursive invocation is the last step of the algorithm

Pseudocode Example

To sum the first n integers, add n to the sum of the first n-1 integers.

```
function Sum(integer n)
{
    if (n = 0)
      then
          return 0                    Base case
      else
          return n + Sum(n – 1)       Do some work after
                                      a recursive call
}
```

Pseudocode Example

Don't do this:

```
function Sum(integer n)
{
    if (n = 0)
        then
            return 0              Base case
      else if (n = 1)
        then
            return 1              Another base case
      else if (n = 2)
        then
            return 3              Yet another base case
      else if (n = 3)
        then
            return 6              Still another base case
      else
            return n + Sum(n – 1) Do some work after a recursive call
}
```

Pseudocode Example

a^n can be defined by: $a^0 = 1$, $a^n = a\,(a^{n-1})$

```
function power(real a, non-negative integer n)
{
   if (n = 0)
       then
           return 1                    Base case
       else
           return a · power(a, n – 1)  Do some work after
                                       a recursive call
}
```

n!

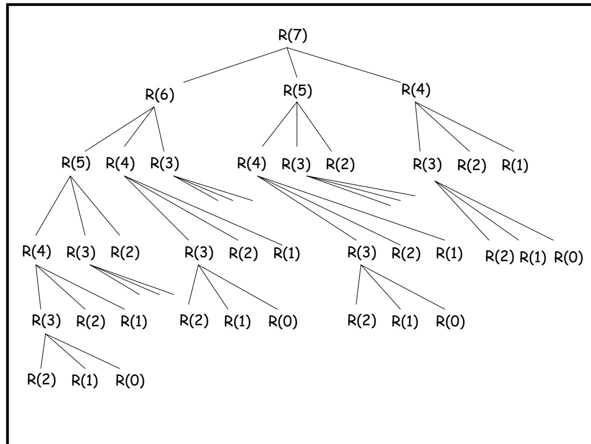
```
function factorial(n)
{
   if (n = 1)
    then
       return 1
    else
       return n · factorial(n – 1)
}
```

```
function nfact(n)
{
   nf := n
   while (n ≠ 1)
   {
      n := n – 1
      nf := n · nf
   }
   return nf
}
```

Give iterative and recursive algorithms for finding the nth term of the sequence defined by: $a_0 = 1$, $a_1 = 3$, $a_2 = 5$, $a_n = a_{n-1} + (a_{n-2})^2 + (a_{n-3})^3$. Which is more efficient?

```
function R(non-negative integer n)
{
        if (n < 3) return 2n + 1
        else
                return R(n-1) + power(R(n-2), 2) + power(R(n-3), 3)
}

function I(non-negative integer n)
{
        if (n < 3) return 2n + 1
        x := 1
        y := 3
        z := 5
        for (i := 1 to n-2)
        {
                w := z + y·y + x·x·x
                x := y
                y := z
                z := w
        }
        return z
}
```



Binary Search of a Sorted Array

low — high

Is the number x in the array ?

bool BinarySearch(int x, int values[], int low, int high)

If x is outside the given range, the answer is no.
Find the middle element.
If it is x, the answer is yes.
Otherwise, search the left part if middle > x
or the right part if middle < x.

How Does Recursion Work Inside a Computer?

We'll start by examining how non-recursive function calls work.

```
  R(a,b)
  {
1    if (a = 0) return 0
2    ...
3    F(a+b)
4    ...
5    ...
6 }

  F(x)
  {
1   ...
2 }
```

```
  R(a,b)
  {
1    if (a = 0) return 0
2    ...
3    F(a+b)
4    ...
5    ...
6 }

  F(x)
  {
1   ...
2 }
```

R(3, 4)

a	3
b	4
ret	OS

Stack frame

Stack

PC R,1

```
  R(a,b)
  {
1    if (a = 0) return 0
2    ...
3    F(a+b)
4    ...
5    ...
6 }

  F(x)
  {
1   ...
2 }
```

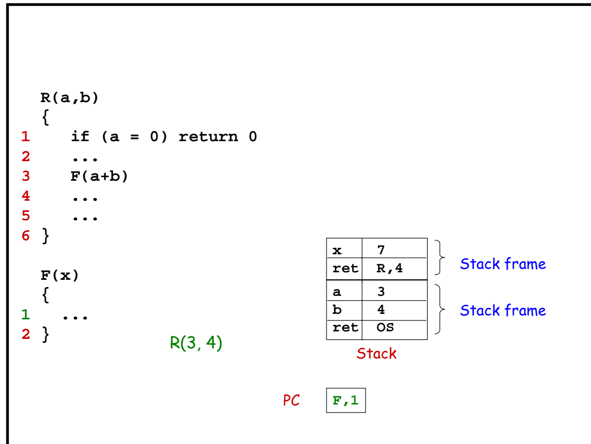
R(3, 4)

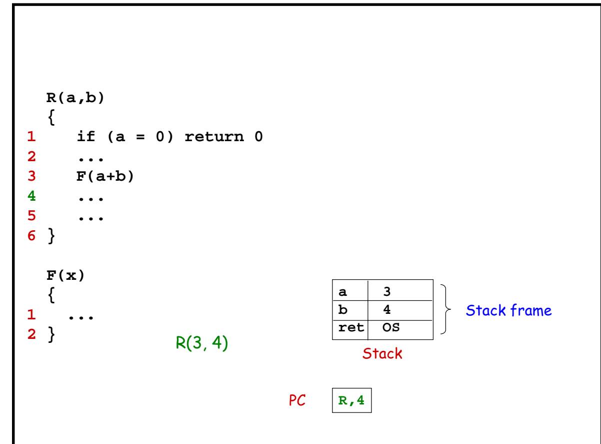
a	3
b	4
ret	OS

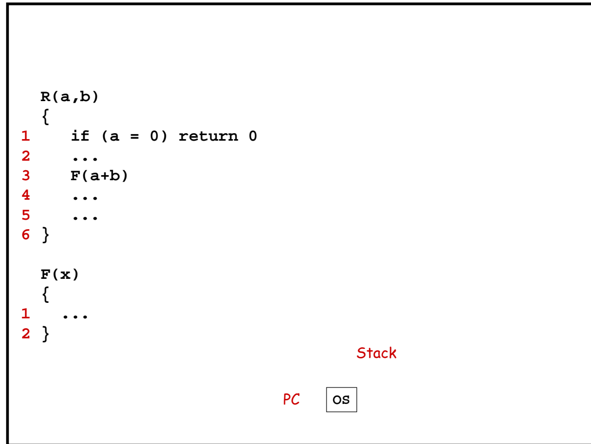
Stack frame

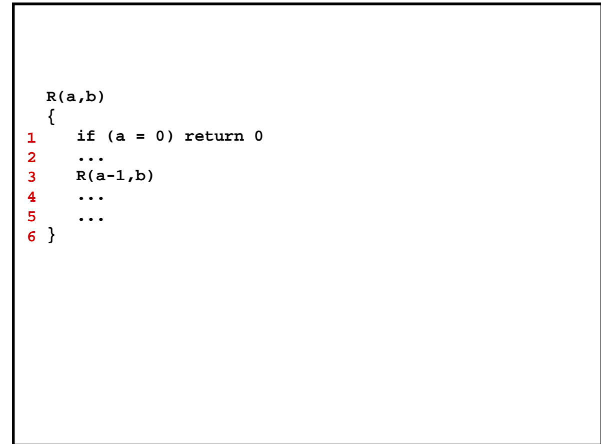
Stack

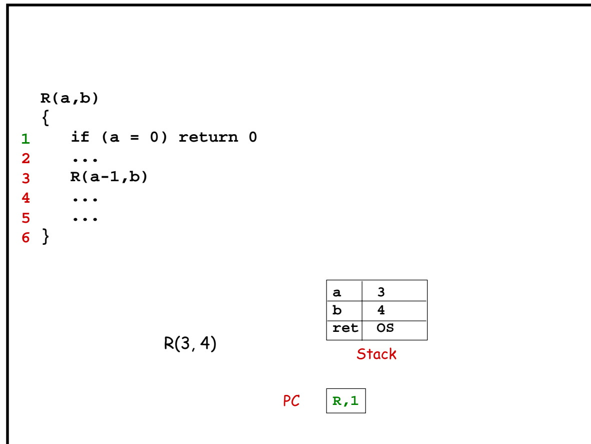
PC R,3

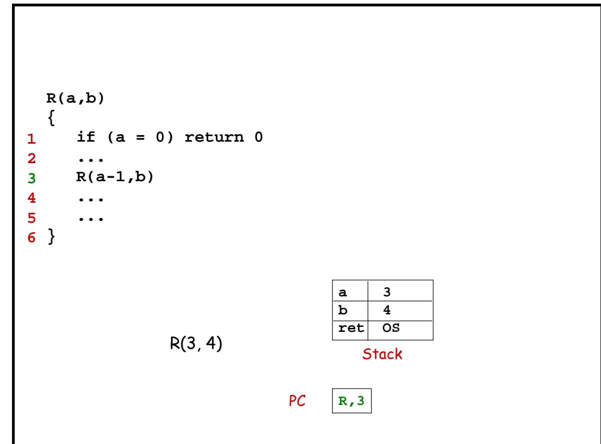

R(a,b)
{
1 if (a = 0) return 0
2 ...
3 F(a+b)
4 ...
5 ...
6 }
F(x)
{
1 ...
2 }
R(3, 4)
x 7
ret R,4
Stack frame
a 3
b 4
ret OS
Stack frame
Stack
PC F,1


R(a,b)
{
1 if (a = 0) return 0
2 ...
3 F(a+b)
4 ...
5 ...
6 }
F(x)
{
1 ...
2 }
R(3, 4)
a 3
b 4
ret OS
Stack frame
Stack
PC R,4


R(a,b)
{
1 if (a = 0) return 0
2 ...
3 F(a+b)
4 ...
5 ...
6 }
F(x)
{
1 ...
2 }
Stack
PC OS


R(a,b)
{
1 if (a = 0) return 0
2 ...
3 R(a-1,b)
4 ...
5 ...
6 }


R(a,b)
{
1 if (a = 0) return 0
2 ...
3 R(a-1,b)
4 ...
5 ...
6 }
R(3, 4)
a 3
b 4
ret OS
Stack
PC R,1


R(a,b)
{
1 if (a = 0) return 0
2 ...
3 R(a-1,b)
4 ...
5 ...
6 }
R(3, 4)
a 3
b 4
ret OS
Stack
PC R,3

```
  R(a,b)
  {
1    if (a = 0) return 0
2    ...
3    R(a-1,b)
4    ...
5    ...
6 }
```

a	2
b	4
ret	R,4
a	3
b	4
ret	OS

R(3, 4)

Stack

PC R,1

```
  R(a,b)
  {
1    if (a = 0) return 0
2    ...
3    R(a-1,b)
4    ...
5    ...
6 }
```

a	2
b	4
ret	R,4
a	3
b	4
ret	OS

R(3, 4)

Stack

PC R,3

```
  R(a,b)
  {
1    if (a = 0) return 0
2    ...
3    R(a-1,b)
4    ...
5    ...
6 }
```

a	1
b	4
ret	R,4
a	2
b	4
ret	R,4
a	3
b	4
ret	OS

R(3, 4)

Stack

PC R,1

```
  R(a,b)
  {
1    if (a = 0) return 0
2    ...
3    R(a-1,b)
4    ...
5    ...
6 }
```

a	0
b	4
ret	R,4
a	1
b	4
ret	R,4
a	2
b	4
ret	R,4
a	3
b	4
ret	OS

R(3, 4)

Stack

PC R,1

```
  R(a,b)
  {
1    if (a = 0) return 0
2    ...
3    R(a-1,b)
4    ...
5    ...
6 }
```

a	1
b	4
ret	R,4
a	2
b	4
ret	R,4
a	3
b	4
ret	OS

R(3, 4)

Stack

PC R,4

```
  R(a,b)
  {
1    if (a = 0) return 0
2    ...
3    R(a-1,b)
4    ...
5    ...
6 }
```

a	1
b	4
ret	R,4
a	2
b	4
ret	R,4
a	3
b	4
ret	OS

R(3, 4)

Stack

PC R,6

```
  R(a,b)
  {
1    if (a = 0) return 0
2    ...
3    R(a-1,b)
4    ...
5    ...
6 }
```

R(3, 4)

a	2
b	4
ret	R,4
a	3
b	4
ret	OS

Stack

PC R,4

```
  R(a,b)
  {
1    if (a = 0) return 0
2    ...
3    R(a-1,b)
4    ...
5    ...
6 }
```

R(3, 4)

a	3
b	4
ret	OS

Stack

PC R,4

```
  R(a,b)
  {
1    if (a = 0) return 0
2    ...
3    R(a-1,b)
4    ...
5    ...
6 }
```

R(3, 4)

Stack

PC OS

Tail Recursion

```
  R(a,b)
  {
1    if (a = 0) return 0
2    ...
3    ...
4    ...
5    R(a-1,b)
6 }
```

R(3, 4)

a	0
b	4
ret	R,6
a	1
b	4
ret	R,6
a	2
b	4
ret	R,6
a	3
b	4
ret	OS

Stack

PC R,1

Tail Recursion

```
  R(a,b)
  {
1    if (a = 0) return 0
2    ...
3    ...
4    ...
5    R(a-1,b)
6 }
```

R(3, 4)

a	1
b	4
ret	R,6
a	2
b	4
ret	R,6
a	3
b	4
ret	OS

Stack

PC R,6

Tail Recursion

```
  R(a,b)
  {
1    if (a = 0) return 0
2    ...
3    ...
4    ...
5    R(a-1,b)
6 }
```

R(3, 4)

a	2
b	4
ret	R,6
a	3
b	4
ret	OS

Stack

PC R,6

Tail Recursion

```
  R(a,b)
  {
1    if (a = 0) return 0
2    ...
3    ...
4    ...
5    R(a-1,b)
6 }
```

R(3, 4)

a	3
b	4
ret	OS

Stack

PC R,6

```
  R(a,b)
  {
1    if (a = 0) return 0
2    ...
3    ...
4    ...
5    R(a-1,b)
6 }
```

R(3, 4)

Stack

PC OS

Tail Recursion

```
  R(a,b)
  {
1    if (a = 0) return 0
2    ...
3    ...
4    ...
5    R(a-1,b)
6 }
```

R(3, 4)

a	3
b	4
ret	OS

Stack

PC R,6

Recursive Functions

A recursive function is one that does some of the work in such a way as to reduce the problem to a simpler one of exactly the same form. Ultimately the problem becomes so simple that it can be solved directly.

Since the subproblem is of the same form as the original, the function can call itself to solve it.

The following need to be true:

There must be some base condition for which the subprogram does not call itself.

Each time the subprogram calls itself, it must converge on the base condition.