Recursion

Slides and figures have been collected from various publicly available Internet sources for preparing the lecture slides of IT2001 course. I acknowledge and thank all the original authors for their contribution to prepare the content

What is recursion?

- Sometimes, the best way to solve a problem is by solving a ***smaller version*** of the exact same problem first
- Recursion is a technique that solves a problem by solving a ***smaller problem*** of the same type

Functions that call themselves (*recursive functions*)

```
int f(int x)
{
 int y;
  if(x==0)
    return 1;
  else {
    y = 2 * f(x-1);
    return y+1;
 }
}
```

Problems defined recursively

- There are many problems whose solution can be defined recursively

Example: *n factorial*

$$n! = \begin{cases} 1 & \text{if } n = 0 \\ (n-1)! * n & \text{if } n > 0 \end{cases} \quad \text{(}\textit{recursive}\text{ solution)}$$

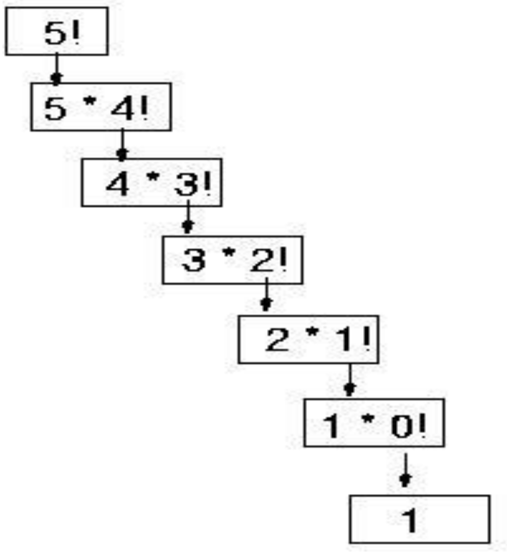
$$n! = \begin{cases} 1 & \text{if } n = 0 \\ 1*2*3*\ldots*(n-1)*n & \text{if } n > 0 \end{cases} \quad \text{(}\textit{closed form}\text{ solution)}$$

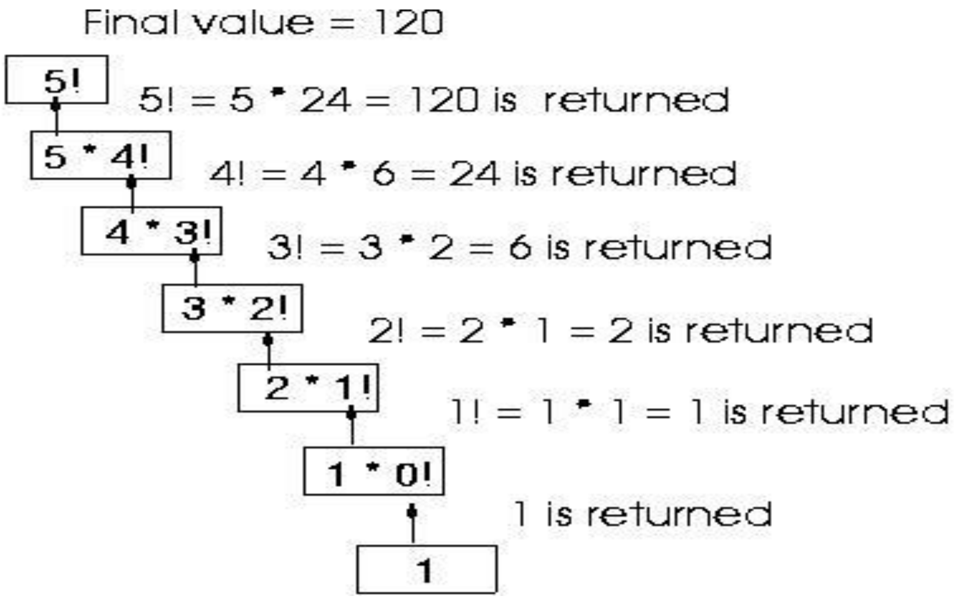
Factorial function

- Recursive implementation

```
int Factorial(int n)
{
 if (n==0)  // base case
   return 1;
 else
   return n * Factorial(n-1);
}
```

Recursive calls





Factorial function (cont.)

- Iterative implementation

```
int Factorial(int n)
{
 int fact = 1;
  for(int count = 2; count <= n; count++)
     fact = fact * count;
 return fact;
}
```

Another example: *n* choose *k* (combinations)

- Given *n* things, how many different sets of size *k* can be chosen?

$$\binom{n}{k} = \binom{n-1}{k} + \binom{n-1}{k-1} \quad , \ 1 < k < n \qquad \textit{(recursive solution)}$$

$$\binom{n}{k} = \frac{n!}{k!(n-k)!} \quad , \ 1 < k < n \qquad \textit{(closed-form solution)}$$

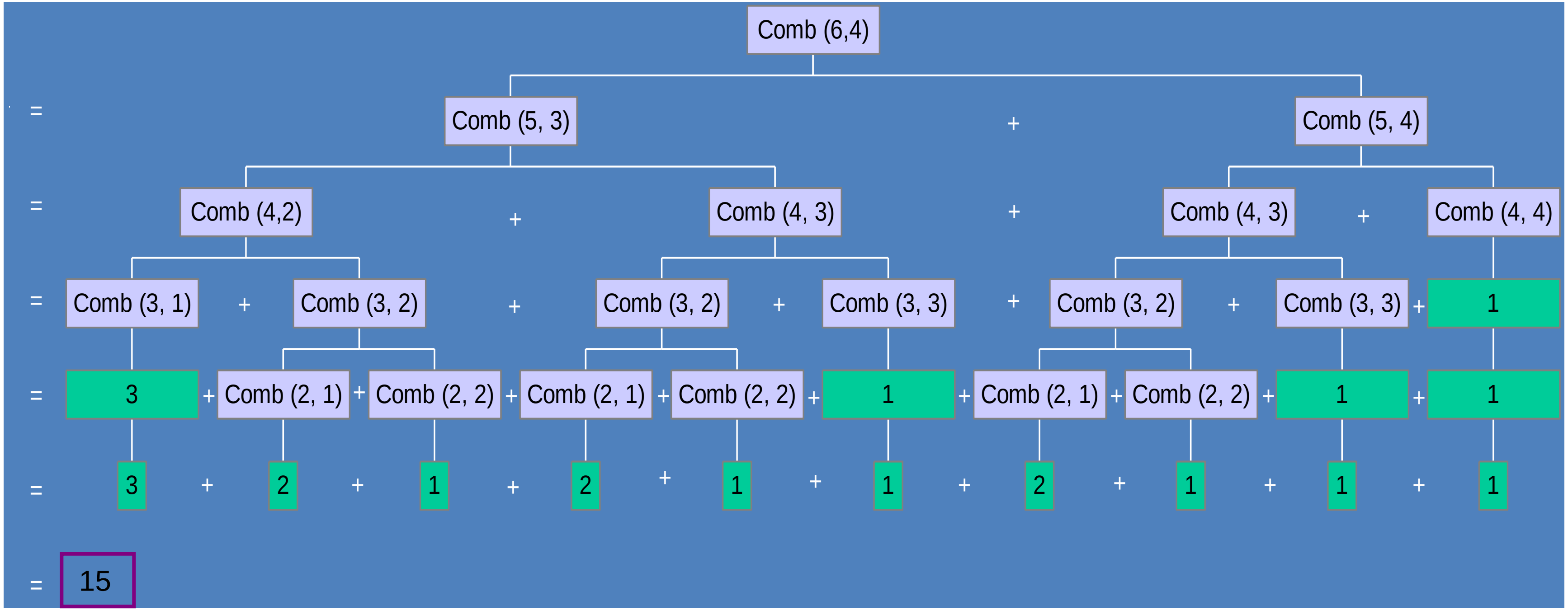
with base cases:

$$\binom{n}{1} = n \ \ (k = 1), \qquad \binom{n}{n} = 1 \ (k = n)$$

n choose *k* (combinations)

```
int Comb(int n, int k)
{
 if(k == 1)  // base case 1
   return n;
 else if (n == k)  // base case 2
   return 1;
 else
   return(Comb(n-1, k) + Comb(n-1, k-1));
}
```

Recursion can be very inefficient is some cases



Recursion vs. iteration

- Iteration can be used in place of recursion
  - An iterative algorithm uses a *looping construct*
  - A recursive algorithm uses a *branching structure*
- Recursive solutions are often less efficient, in terms of both *time* and *space*, than iterative solutions
- Recursion can simplify the solution of a problem, often resulting in shorter, more easily understood source code

How do I write a recursive function?

- Determine the <u>size factor</u>
- Determine the <u>base case(s)</u>
  - the one for which you know the answer
- Determine the <u>general case(s)</u>
  - the one where the problem is expressed as a smaller version of itself

Three-Question Verification Method

The Base-Case Question:

- Is there a nonrecursive way out of the function, and does the routine work correctly for this "base" case?

The Smaller-Caller Question:

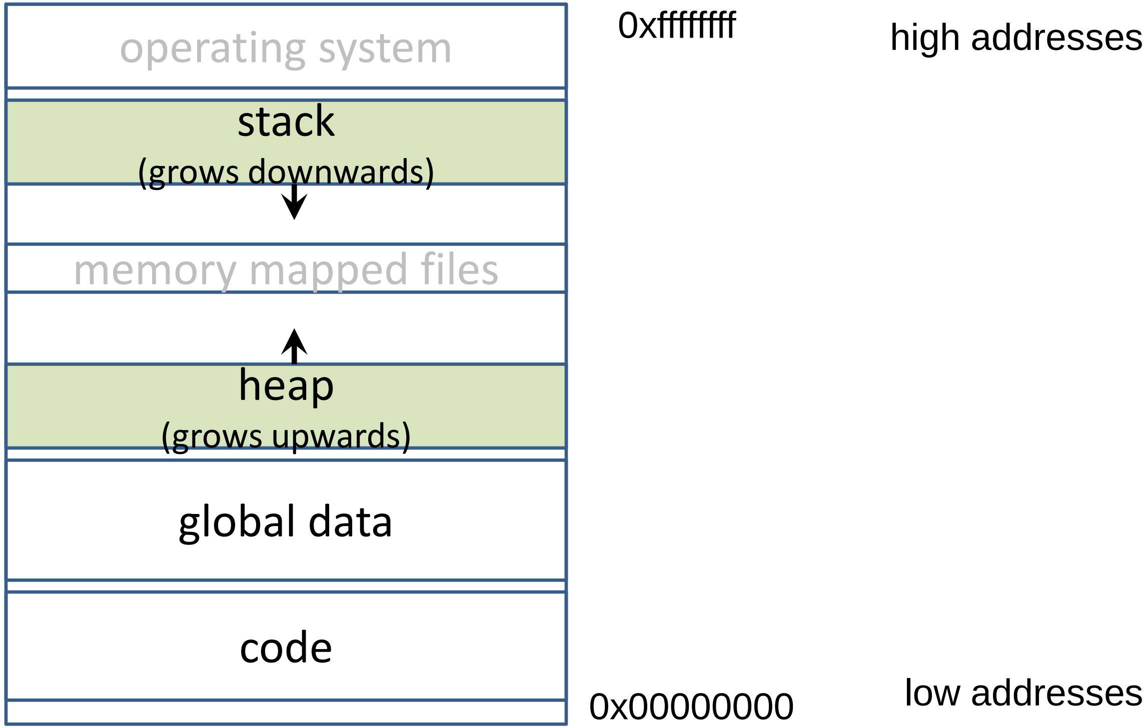
- Does each recursive call to the function involve a smaller case of the original problem, leading inescapably to the base case?

The General-Case Question:

- Assuming that the recursive call(s) work correctly, does the whole function work correctly?

Background: Runtime Memory Organization

Layout of an executing process's virtual memory:



Background: Runtime Memory Organization

Code:

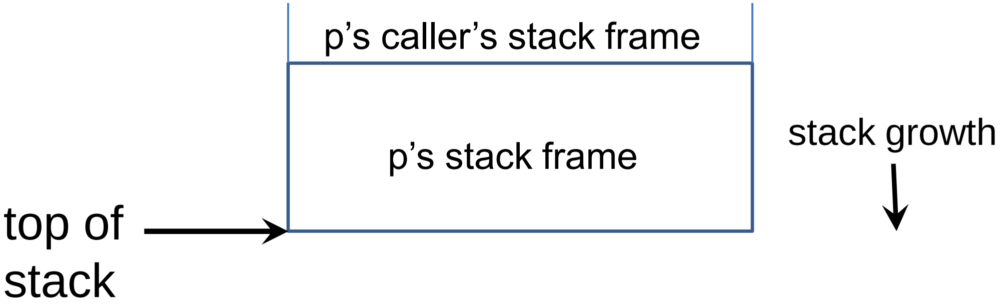
```
p(…) {
   …
   q(…);
   s(…);
}

q(…) {
   …
   r(…);
}

r(…)
{

s(…) {
   …
}
```

Runtime stack:



Background: Runtime Memory Organization

Code:

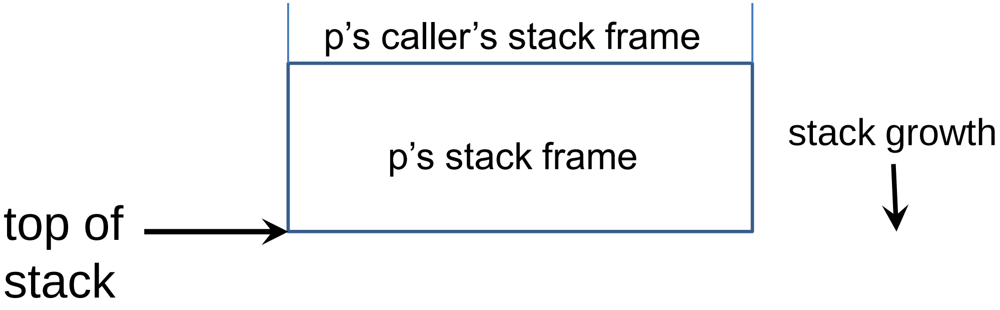
```
p(…) {
  …
  q(…);
  s(…);
}

q(…) {
  …
  r(…);
}

r(…)
{
```

```
s(…) {
  …
}
```

Runtime stack:



Background: Runtime Memory Organization

Code:

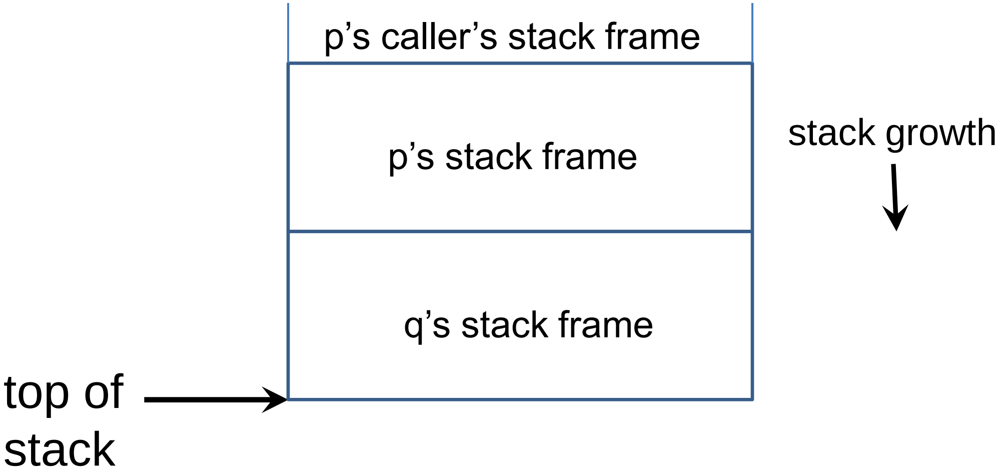
```
p(…) {
  …
  q(…);
  s(…);
}

q(…) {
  …
  r(…);
}

r(…)
{
```

```
s(…) {
  …
}
```

Runtime stack:



Background: Runtime Memory Organization

Code:

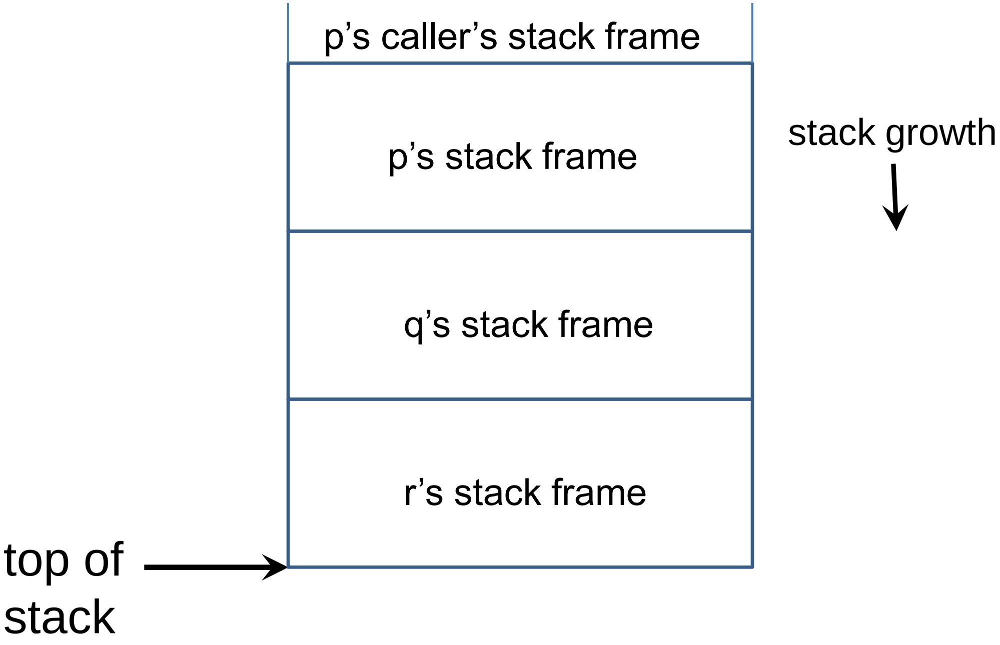
```
p(…) {
  …
  q(…);
  s(…);
}

q(…) {
  …
  r(…);
}

r(…)
{
```

```
s(…) {
  …
}
```

Runtime stack:



Background: Runtime Memory Organization

Code:

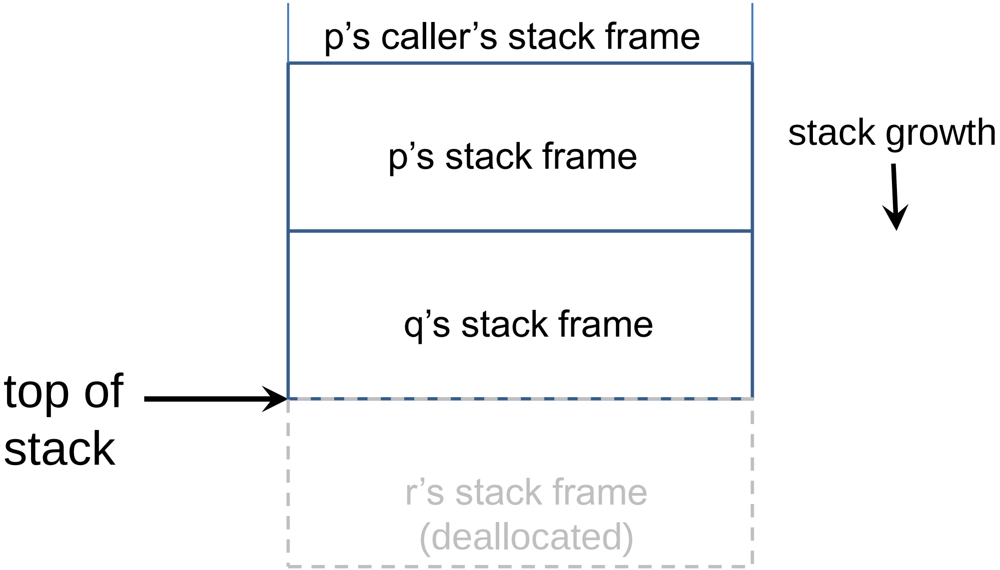
```
p(…) {
  …
  q(…);
  s(…);
}

q(…) {
  …
  r(…);
}

r(…)
{
```

```
s(…) {
  …
}
```

Runtime stack:



Background: Runtime Memory Organization

Code:

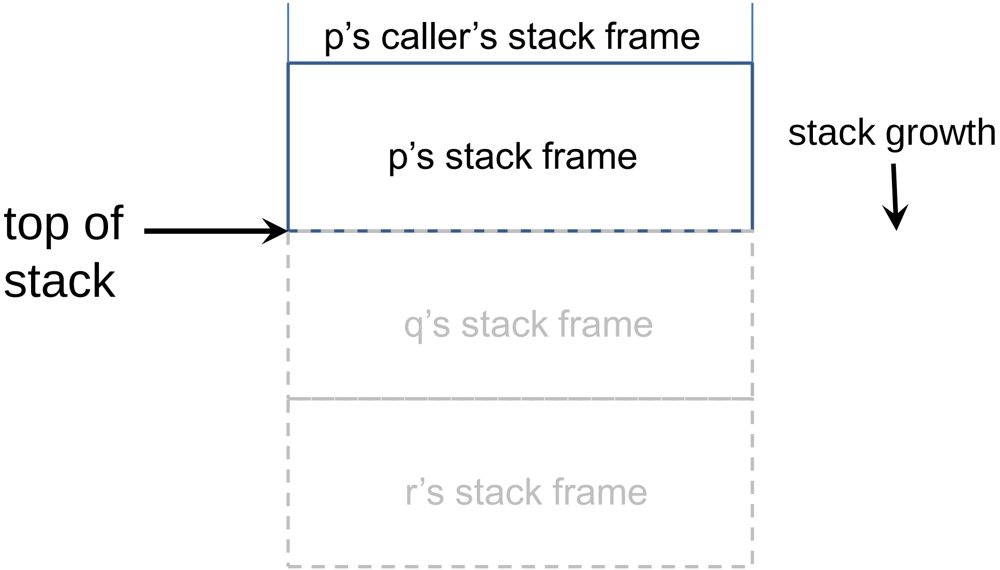
```
p(…) {
  …
  q(…);
  s(…);
}

q(…) {
  …
  r(…);
}

r(…)
{
```

```
s(…) {
  …
}
```

Runtime stack:



Background: Runtime Memory Organization

Code:

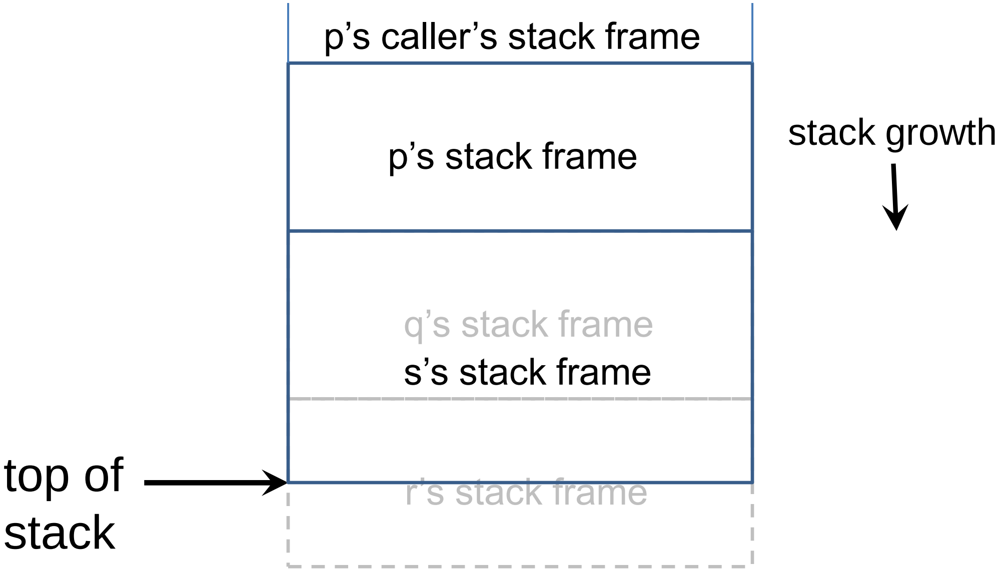
```
p(…) {
  …
  q(…);
  s(…);
}

q(…) {
  …
  r(…);
}

r(…)
{
```

```
s(…) {
  …
}
```

Runtime stack:



How is recursion implemented?

- What happens when a function gets called?

```
int A(int w)
{
 return w+w;
}

int B(int x)
{
 int z,y;
 ……………… // other statements
 z = A(x) + y;

 return z;
}
```

What happens when a function is called? (cont.)

An **activation** record is stored into a stack **(run-time stack)**

1) The computer has to stop executing function ***B*** and starts executing function **A**
2) Since it needs to come back to function ***B*** later, it needs to store everything about function **B** that is going to need (**x, y, z**, and the place to start executing upon return)
3) Then, **x** from **B** is bounded to **w** from **A**
4) Control is transferred to function **A**

What happens when a function is called? (cont.)

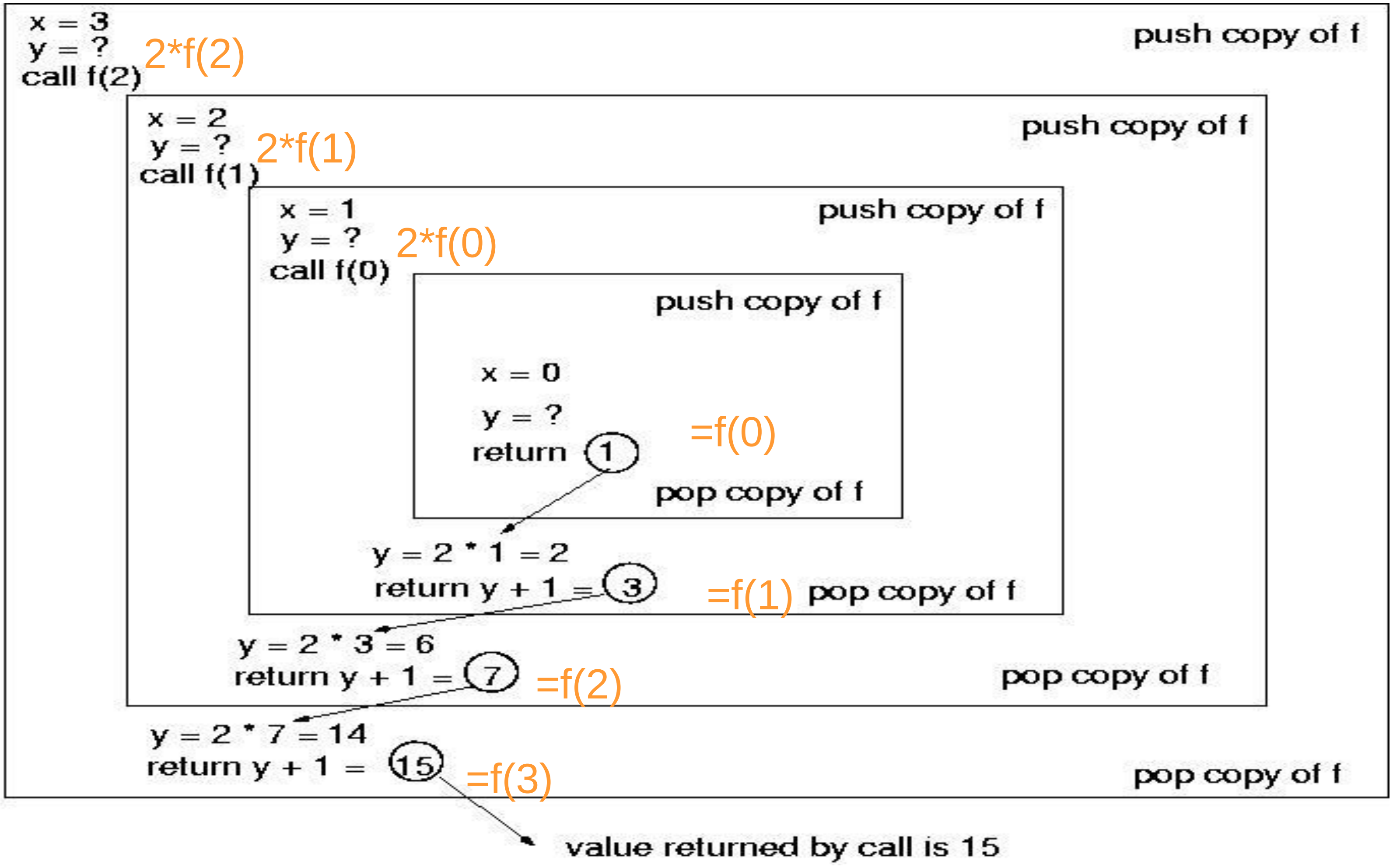
- After function **A** is executed, the activation record is popped out of the run-time stack
- All the old values of the parameters and variables in function **B** are restored and the return value of function **A** replaces **A(x)** in the assignment statement

What happens when a recursive function is called?

- Except the fact that the calling and called functions have the same name, there is really no difference between recursive and nonrecursive calls

```
int f(int x)
{
    int y;

    if(x==0)
            return 1;
    else {
            y = 2 * f(x-1);
            return y+1;
    }
}
```



x = 3
y = ?
call f(2)
2*f(2)
push copy of f
x = 2
y = ?
call f(1)
2*f(1)
push copy of f
x = 1
y = ?
call f(0)
2*f(0)
push copy of f
push copy of f
x = 0
y = ?
return 1
=f(0)
pop copy of f
y = 2 * 1 = 2
return y + 1 = 3
=f(1)
pop copy of f
y = 2 * 3 = 6
return y + 1 = 7
=f(2)
pop copy of f
y = 2 * 7 = 14
return y + 1 = 15
=f(3)
pop copy of f
value returned by call is 15

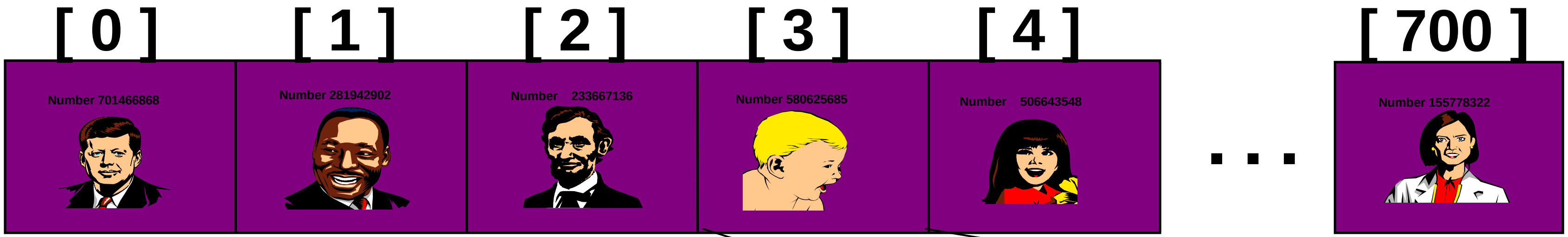
Deciding whether to use a recursive solution

- When the depth of recursive calls is relatively "shallow"
- The recursive version does about the same amount of work as the nonrecursive version
- The recursive version is shorter and simpler than the nonrecursive solution

Problem: Search

- We are given a list of records.
- Each record has an associated key.
- Give efficient algorithm for searching for a record containing a particular key.
- Efficiency is quantified in terms of average time analysis (number of comparisons) to retrieve an item.

Search



Each record in list has an associated key.
In this example, the keys are ID numbers.

Given a particular key, how can we efficiently retrieve the record from the list?

Serial Search

- Step through array of records, one at a time.
- Look for record with matching key.
- Search stops when
  - record with matching key is found
  - or when search has examined all records without success.

Pseudocode for Serial Search

```
// Search for a desired item in the n array elements
// starting at a[first].
// Returns the position of the desired record if found.
// Otherwise, return “not found”
…
for(i = 0; i < n; ++i )
        if(a[i] == desired item)
                return i+1;
```

Serial Search Analysis

- What are the worst and average case running times for serial search?
- Number of operations depends on *n*, the number of entries in the list.

Worst Case Time for Serial Search

- For an array of n elements, the worst case time for serial search requires n array accesses: $O(n)$.
- Consider cases where we must loop over all n records:
  - desired record appears in the last position of the array
  - desired record does not appear in the array at all

Average Case for Serial Search

Assumptions:

1. All keys are equally likely in a search
2. We always search for a key that is in the array

Example:

- We have an array of 10 records.
- If search for the first record, then it requires 1 array access; if the second, then 2 array accesses. *etc.*

The average of all these searches is:

$$(1+2+3+4+5+6+7+8+9+10)/10 = 5.5$$

Average Case Time for Serial Search

Generalize for array size *n.*

Expression for average-case running time:

$(1+2+\ldots+n)/n = n(n+1)/2n = (n+1)/2$

Therefore, average case time complexity for serial search is $O(n)$.

Binary Search

- Perhaps we can do better than O(n) in the average case?
- Assume that we are give an array of records that is sorted. For instance:
  - an array of records with integer keys sorted from smallest to largest (e.g., ID numbers), or
  - an array of records with string keys sorted in alphabetical order (e.g., names).

Binary Search

Example: sorted array of integer keys. Target=7.

[0]	[1]	[2]	[3]	[4]	[5]	[6]
3	6	7	11	32	33	53

Binary Search

Example: sorted array of integer keys. Target=7.

[0]	[1]	[2]	[3]	[4]	[5]	[6]
3	6	7	11	32	33	53

Binary Search

Example: sorted array of integer keys. Target=7.

[0]	[1]	[2]	[3]	[4]	[5]	[6]
3	6	7	11	32	33	53

Binary Search

Example: sorted array of integer keys. Target=7.

[0]	[1]	[2]	[3]	[4]	[5]	[6]
3	6	7	11	32	33	53

Binary Search

Example: sorted array of integer keys. Target=7.

[0]	[1]	[2]	[3]	[4]	[5]	[6]
3	6	7	11	32	33	53

Binary Search

Example: sorted array of integer keys. Target=7.

[0]	[1]	[2]	[3]	[4]	[5]	[6]
3	6	7	11	32	33	53

Binary Search

Example: sorted array of integer keys. Target=7.

[0]	[1]	[2]	[3]	[4]	[5]	[6]
3	6	7	11	32	33	53

Binary Search

Example: sorted array of integer keys. Target=7.

[0]	[1]	[2]	[3]	[4]	[5]	[6]
3	6	7	11	32	33	53

Binary Search

Example: sorted array of integer keys. Target=7.

[0]	[1]	[2]	[3]	[4]	[5]	[6]
3	6	7	11	32	33	53

Binary Search

Example: sorted array of integer keys. Target=7.

[0]	[1]	[2]	[3]	[4]	[5]	[6]
3	6	7	11	32	33	53

Binary Search

Example: sorted array of integer keys. Target=7.

[0]	[1]	[2]	[3]	[4]	[5]	[6]
3	6	7	11	32	33	53

Binary Search (non-recursive)

```
int BinarySearch ( array[ ], first, last, target) {
  // first = 0 and last = array.length-1  for searching the entire list
   while ( first <= last ) {
            mid = (first + last) / 2;
            if ( target == array[mid] ) return mid;  // found it
            else if ( target < array[mid] )  // must be in 1st half
                      last = mid -1;
            else // must be in 2nd half
                      first = mid + 1
     }
     return -1; // only got here if not found above
}
```

Binary Search (recursive)

```
int BinarySearch ( array[ ], first, last, target) {
    if ( first <= last ) {              // base case 1
        mid = (first + last) / 2;
        if ( target == array[mid] ) // found it! // base case 2
            return mid;
        else if ( target < array[mid] )  // must be in 1st half
                return BinarySearch( array, first, mid-1, target);
        else // must be in 2nd half
                return BinarySearch(array, mid+1, last, target);
    }
    return -1; // only got here if not found above
}
```

- No loop! Recursive calls takes its place
- Base cases checked first? (Why? Zero items? One item?)

Recursive binary search (cont'd)

- What is the *size factor*?

  The number of elements in (array*[first]* ... *array[last]*)

- What is the *base case(s)*?

  (1) If *first* > *last*, return -1

  (2) If target==*array[mid]*, return mid

- What is the *general case*?

  if target < *array[mid]* search the first half

  if target > *array[mid]*, search the second half

Binary Search: Analysis

- Worst case complexity?
- What is the maximum depth of recursive calls in binary search as function of *n*?
- Each level in the recursion, we split the array in half (divide by two).
- Therefore, maximum recursion depth is floor($\log_2 n$) and worst case = $O(\log_2 n)$.
- Average case is also = $O(\log_2 n)$.

Can we do better than $O(\log_2 n)$?

- Average and worst case of serial search = $O(n)$
- Average and worst case of binary search = $O(\log_2 n)$
- Can we do better than this?

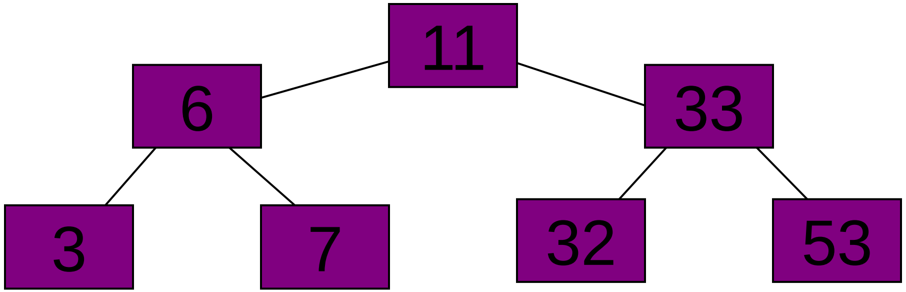
YES. Use a hash table! (Will be taught later)

Relation to Binary Search Tree

Array of previous example:

3	6	7	11	32	33	53

Corresponding complete binary search tree

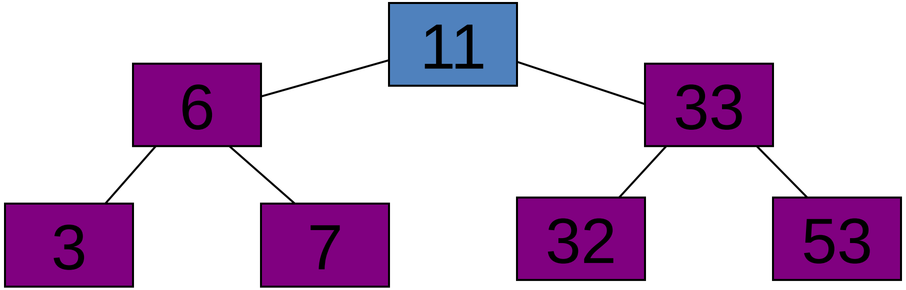


Search for target = 7

Find midpoint:

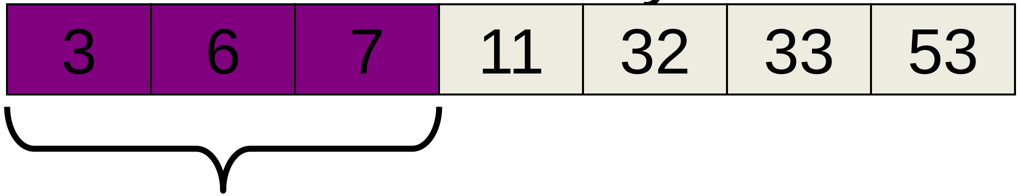
3	6	7	11	32	33	53

Start at root:

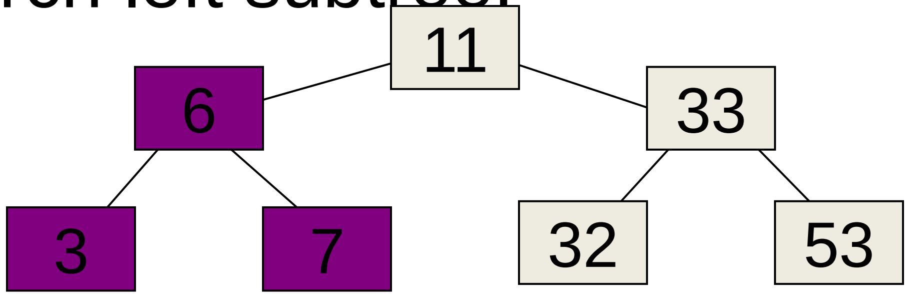


Search for target = 7

Search left subarray:

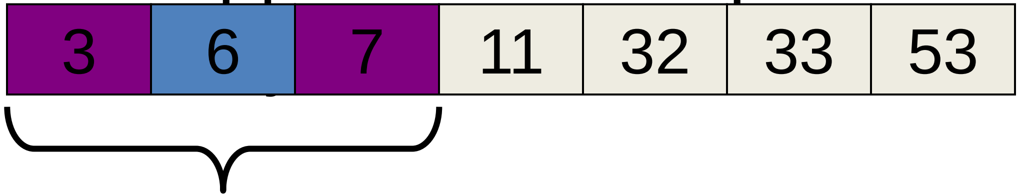


Search left subtree:

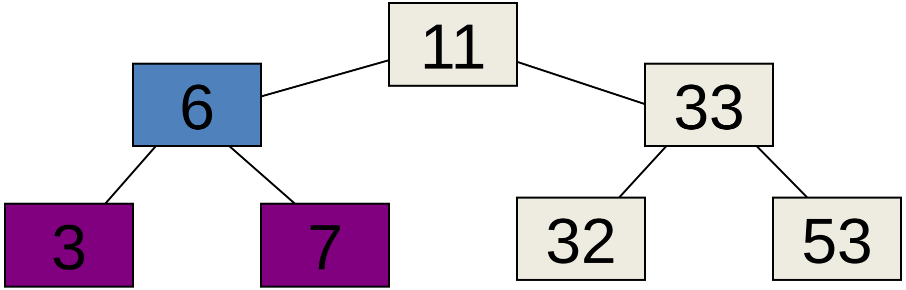


Search for target = 7

Find approximate midpoint of



Visit root of subtree:



Search for target = 7

Search right subarray:

3	6	7	11	32	33	53

Search right subtree: