Recursion
What Is Recursion?

- **Recursive call**
  - A method call in which the method being called is the same as the one making the call

- **Direct recursion**
  - Recursion in which a method directly calls itself

- **Indirect recursion**
  - Recursion in which a chain of two or more method calls returns to the method that originated the chain
Recursion

- You must be careful when using recursion.
- Recursive solutions can be less efficient than iterative solutions.
- Still, many problems lend themselves to simple, elegant, recursive solutions.
Some Definitions

- **Base case**
  - The case for which the solution can be stated nonrecursively

- **General (recursive) case**
  - The case for which the solution is expressed in terms of a smaller version of itself

- **Recursive algorithm**
  - A solution that is expressed in terms of (a) smaller instances of itself and (b) a base case
Recursive Function Call

• A **recursive call** is a function call in which the called function is the same as the one making the call.

• In other words, *recursion occurs when a function calls itself!*

• We must avoid making an infinite sequence of function calls (infinite recursion).
Finding a Recursive Solution

- Each successive recursive call should bring you closer to a situation in which the answer is known.

- A case for which the answer is known (and can be expressed without recursion) is called a base case.

- Each recursive algorithm must have at least one base case, as well as the general (recursive) case.
General format for many recursive functions

```plaintext
if (some condition for which answer is known)
    solution statement  //base case
else
    recursive function call  //general case
```
Writing a recursive function to find n!

• Factorial(n) = n*\(n-1\)*n-2*...*1
  – Factorial(4) = 4 * 3 * 2 * 1 = 24
  – Factorial(0) = 1

• Definition of Factorial(n)
  – n*Factorial(n-1) if n>0
  – 1 if n=0
Recursive function

```cpp
int fact(int n)
{
    if (n==0)
        return 1;
    return n*fact(n-1);
}
```

- Call itself
- Parameter to recursive call is diminished by 1
- Fact(0)=1;
- Always reach base case if n>=0
Box Method

- Activation record
  - Arguments
  - Local variables
  - A place holder for value returned by recursive calls
  - The return value of the function itself

n = 3
A: fact(n-1) = ?
return ?

Activation record for Fact(3)
Box Method

The initial call is made, and method `fact` begins execution:

```
  n = 3
  A: fact(n-1)=?
  return ?
```

At point A a recursive call is made, and the new invocation of the method `fact` begins execution:

```
  n = 3
  A: fact(n-1)=?
  return ?
```

```
  n = 2
  A: fact(n-1)=?
  return ?
```

At point A a recursive call is made, and the new invocation of the method `fact` begins execution:

```
  n = 3
  A: fact(n-1)=?
  return ?
```

```
  n = 2
  A: fact(n-1)=?
  return ?
```

```
  n = 1
  A: fact(n-1)=?
  return ?
```
At point A a recursive call is made, and the new invocation of the method `fact` begins execution:

```
 n = 3
 A: fact(n-1)=?
 return ?
```

This is the base case, so this invocation of `fact` completes:

```
 n = 3
 A: fact(n-1)=?
 return ?
```

The method value is returned to the calling box, which continues execution:

```
 n = 3
 A: fact(n-1)=?
 return ?
```

The current invocation of `fact` completes:

```
 n = 3
 A: fact(n-1)=?
 return ?
```
The method value is returned to the calling box, which continues execution:

The current invocation of \texttt{fact} completes:

The method value is returned to the calling box, which continues execution:

The current invocation \texttt{fact} completes:

The value 6 is returned to the initial call.
cout << fact(3);

return 3*fact(2)

3*2

return 2*fact(1)

2*1

return 1*fact(0)

1*1

return 1
Three-Question Method of verifying recursive functions

- **Base-Case Question**: Is there a nonrecursive way out of the function?
- **Smaller-Caller Question**: Does each recursive function call involve a smaller case of the original problem leading to the base case?
- **General-Case Question**: Assuming each recursive call works correctly, does the whole function work correctly?
Stacks for Recursion

- **A stack**
  - Specialized memory structure
  - Like stack of paper
    - place new on top
    - remove when needed from top
  - Called ‘last-in/first-out’ memory structure

- **Recursion uses stacks**
  - Each recursive call placed on stack
  - When one completes, last call is removed from stack
Stack Overflow

- Size of stack limited
- Long chain of recursive calls continually adds to stack
  - All are added before base case causes removals
- If stack attempts to grow beyond limit
  - Stack overflow error
- Infinite recursion always causes this
Recursion Versus Iteration

- Recursion not always ‘necessary’
- Not even allowed in some languages
- Any task accomplished with recursion can also be done without it
  - Nonrecursive: called iterative, using loops
- Recursive
  - Runs slower, uses more storage
  - Elegant solution; less coding
Binary Search

- Recursive function to search sorted array
  - Determines IF item is in array, and if so:
  - Where in list it is
- Breaks array in half
  - Determines if item in 1\textsuperscript{st} or 2\textsuperscript{nd} half
  - Then searches again just that half
- Recursively (of course)!
Pseudocode for Binary Search

Display 13.5  Pseudocode for Binary Search

```c
int a[Some_Size_Value];

Algorithm to Search a[first] through a[last]

// Precondition:
// a[first] <= a[first + 1] <= a[first + 2] <= ... <= a[last]

To locate the value key:

if (first > last) // A stopping case
    found = false;
else
{
    mid = approximate midpoint between first and last;
    if (key == a[mid]) // A stopping case
        {  
            found = false;
            location = mid;
        }
    else if key < a[mid] // A case with recursion
        search a[first] through a[mid - 1];
    else if key > a[mid] // A case with recursion
        search a[mid + 1] through a[last];
    }
```
Checking the Recursion

- No infinite recursion:
  - Each call increases first or decreases last
  - Eventually first will be greater than last

- Stopping cases perform correct action:
  - If first > last $\rightarrow$ no elements between them
  - If key == a[mid] $\rightarrow$ correctly found!

- Recursive calls perform correct action
  - If key < a[mid] $\rightarrow$ key in 1\textsuperscript{st} half – correct call
  - If key > a[mid] $\rightarrow$ key in 2\textsuperscript{nd} half – correct call
Execution of Binary Search

Display 13.7  Execution of the function search

key is 63

<table>
<thead>
<tr>
<th>a[0]</th>
<th>15</th>
<th>first == 0</th>
<th>a[0]</th>
<th>15</th>
</tr>
</thead>
</table>

Not in this half

mid = (5 + 9)/2

first == 5

mid = (5 + 9)/2

first == 5

mid = (5 + 6)/2 which is 5

a[mid] is a[5] == 63

found = TRUE;

location = mid;

Not here
Efficiency of Binary Search

- Extremely fast
  - Compared with sequential search
- Half of array eliminated at start!
  - Then a quarter, then 1/8, etc.
  - Essentially eliminate half with each call
- For an array of 100 elements
  - Binary search never needs more than 7 compares!
  - Logarithmic efficiency (log n)
Summary

- Reduce problem into smaller instances of same problem -> recursive solution
- Recursive algorithm has two cases:
  - Base/stopping case
  - Recursive case
- Ensure no infinite recursion
- Use criteria to determine recursion correct
  - Three essential properties