Recursion

- Recursive void Functions
- Recursive Functions that Return a Value
- Thinking Recursively
- Binary Search and Merge Sort

Introduction to Recursion

- C++ allows recursion
  - as do most high-level languages
  - can be useful programming technique
  - has limitations
- Recursion is implemented in functions
- A recursive function
  - **a function that calls itself**

Recursive Function

- Divide and Conquer
  - break large task into subtasks
  - basic design technique
- <u>Subtasks are smaller versions of the original task!</u>
- Example -- search list for a value
  - Subtask 1: search 1st half of list
  - Subtask 2: search 2nd half of list
  - Subtasks are smaller versions of original task
- recursive function can be used.
  - Usually results in "elegant" solution

Recursive `void` Function: Vertical Numbers

- Task: display digits of integer vertically, one per line
- Example call:

```
writeVertical(1234);
```

Produces output:

1

2

3

4

Vertical Numbers: Recursive Definition

- Break problem into two cases
- Simple/base case: if n < 10

  -- simply output n to screen
- Recursive case: if n >= 10, two subtasks:

  1) output all digits except last digit – smaller subtask

  2) output last digit – smaller subtask
- Example: argument `1234`:
  - 1st subtask displays 1, 2, 3 vertically
  - 2nd subtask displays 4

writeVertical Function Definition

```
void writeVertical(int n)
{
    if (n < 10)  //Base case
        cout << n << endl;

    else  //Recursive step
    {
        writeVertical (n / 10);
        // call itself on smaller argument!

        cout << (n % 10) << endl;
    }
}
```

writeVertical Trace

- Example call:

```
writeVertical(123);
    writeVertical(12);          // (123/10)
        writeVertical(1);     // (12/10)
            cout << 1 << endl;
        cout << 2 << endl;
    cout << 3 << endl;
```

- Altogether, calling `writeVertical` three times
- Notice first two calls call again (recursive)
- Last call on argument (1) displays, "ends“ the series of calls, and starts “backtracking”

Recursion—A Closer Look

- Computer tracks recursive calls
  - Stops current function
  - Must know results of new recursive call before proceeding
  - Saves all information needed for current call
    - To be used later
  - Proceeds with evaluation of new recursive call
  - When THAT call is complete, returns to "outer" computation/statements

Recursion Big Picture

- Outline of successful recursive function:
  - **One or more** cases where function accomplishes its task by:
    - making one or more recursive calls to solve smaller versions of original task
    - called "**recursive case(s)**“
  - **One or more** cases where function accomplishes its task without recursive calls
    - Called "**base case(s)**" or stopping case(s)

Infinite Recursion

- Base case MUST eventually be entered
- If it doesn't → infinite recursion
  - recursive calls never end!
- Recall `writeVertical` example:
  - base case happened when down to 1-digit number
  - that's when recursion stopped

Infinite Recursion Example

- Consider alternate function definition:

```
void newWriteVertical (int n)
{
    newWriteVertical (n / 10);
    cout << (n % 10) << endl;
}
```

- Seems "reasonable“, but missing "base case"!
- Recursion never stops

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
  - memory is finite
- 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“
- Any task accomplished with recursion can also be done without it
  - non-recursive, also called iterative -- using loops
- Recursive:
  - Runs slower, uses more storage
  - Elegant solution; less coding

Recursive Functions that Return a Value

- Recursion not limited to void functions
- Can return value of any type
- Same technique, outline:
  1. One+ cases where value returned is computed by recursive calls
     - Should be "smaller" sub-problems
  2. One+ cases where value returned is computed without recursive calls
     - Base case

Return a Value Recursion Example: Powers

- Recall predefined function **`pow`** :

  ```
  result = pow(2.0, 3.0);
  ```

  - returns 2 raised to power 3 (8.0)
  - takes two `double` arguments
  - returns `double` value

- Let's define a *similar* function recursively
  - integer arguments, not double
  - negative exponent not considered

Function Definition for `power()`

```
// precondition: n >= 0

int power(int x, int n)
{
    if (n == 0)
        return 1;
    else
        return power(x, n-1) * x;
}
```

Calling Function `power()`

Example calls:

```
power(2, 0);
```

→ returns 1

```
power(2, 1);
```

→ returns `( power(2, 0) * 2 );`

→ returns `1`

- value `1` multiplied by `2` , then returned to original call

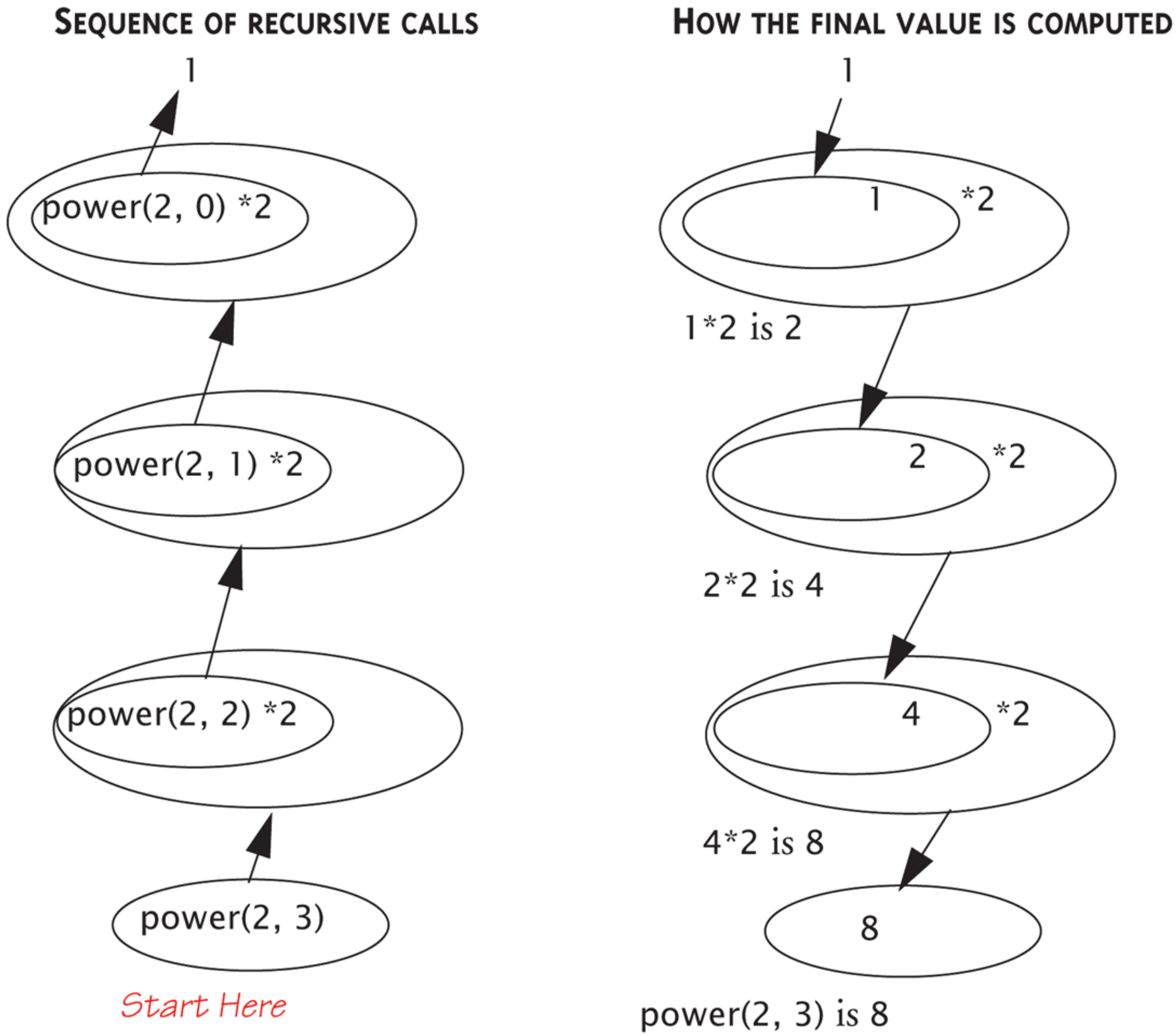
Calling Function `power()`

Larger example call:

```
power(2,3);
   → power(2,2) * 2
         → power(2,1) * 2
               → power(2,0) * 2
                     →1
```

- reaches base case
- recursion stops
- values "returned back" up stack
- See next slide for tracing the function call

Display 13.4 **Evaluating the Recursive Function Call power(2,3)**



Thinking Recursively

When designing recursive solution:

- Ignore details
  - forget how stack works
  - forget the suspended computations
  - yes, this is an "abstraction" principle!
  - and encapsulation principle!
- **Programmer just think “big picture”**
  - Let computer do "bookkeeping"

Thinking Recursively: `power`

- Consider `power` function again
- To calculate x^n :

$$\overbrace{x * x * x * \ldots * x * x}^{n}$$

  A smaller subtask is x^{n-1}

$$\underbrace{x * x * x * \ldots * x}_{n-1} * x$$

- suppose x^{n-1} is computed, how to compute x^n?
  - In other words, **how to go from the smaller subtask to the original one?**
  - Easy! $x^{n-1} * x$
- And ensure base case will be met

Recursive Design Techniques

- Don’t trace entire recursive sequence!
- Just check 3 properties:

  1. no infinite recursion
     - at least one base case
     - the size of the argument is smaller in the recursive call
  2. stopping/base case(s) return correct value(s)
  3. recursive case(s) return correct value(s)

Recursive Design Check: `power()`

- Check **`power()`** against 3 properties:

1. No infinite recursion:
    - 2^{nd} argument decreases by 1 each call
    - Eventually must get to base case of 1
2. Stopping case returns correct value:
    - power(x,0) is base case
    - Returns 1, which is correct for x^0
3. Recursive calls correct:
    - For n>1, power(x, n) returns power(x, n-1) * x
    - Plug in values → correct

Binary Search

- Recursive function to search array
  - determines IF item is in list, and if so:
  - where in list it is
- Assumes **array is sorted**
- Breaks list in half
  - determines if item in 1st or 2nd half
  - then searches again just that half
    - Recursively (of course)!

Pseudocode for Binary Search

```
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:

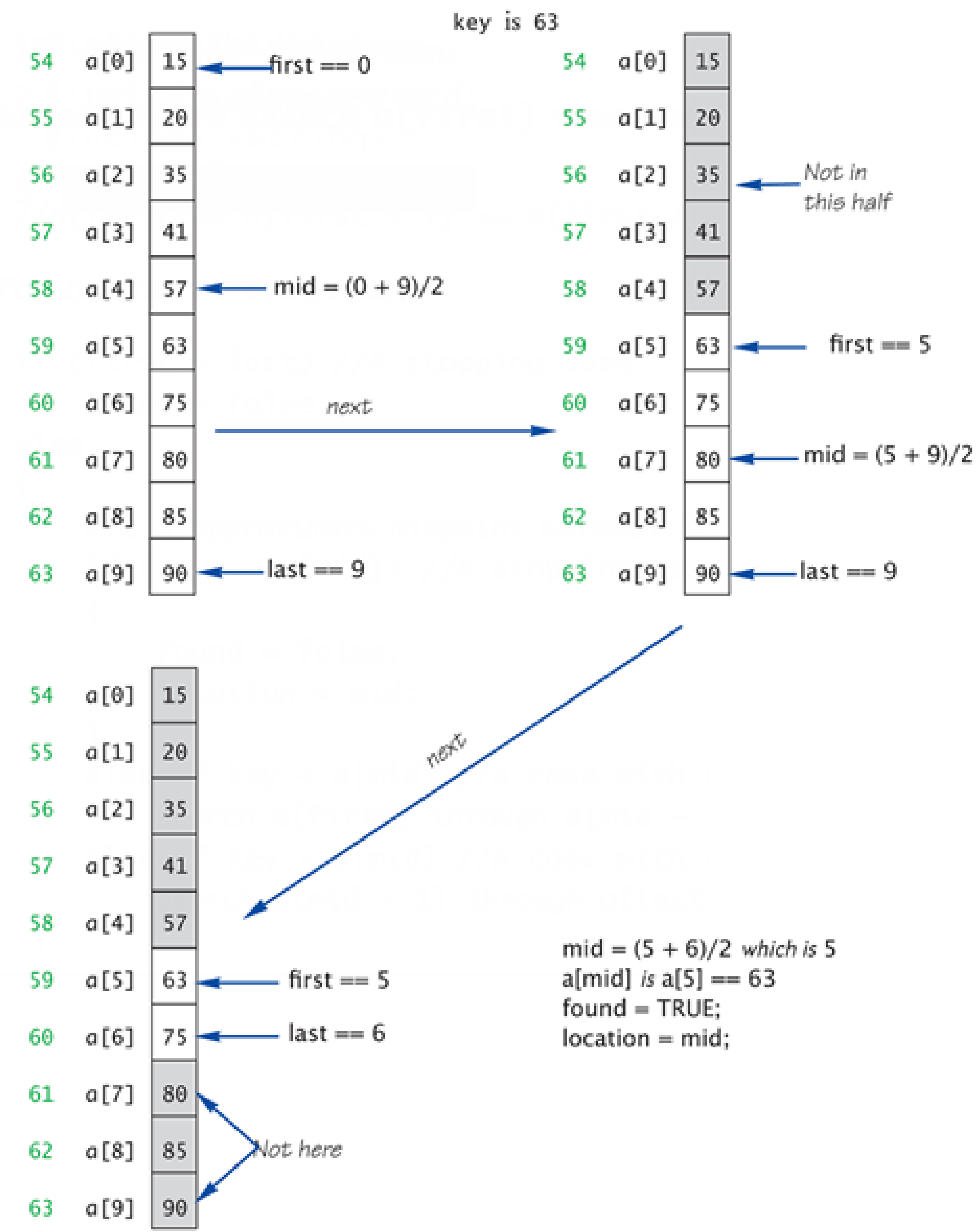
```
if (first > last) //A stopping case
    found = false;
else
{
    mid = approximate midpoint between first and last;
    if (key == a[mid]) //A stopping case
    {
        found = false;  true
        location = mid;
    }
    else if key < a[mid] //A case with recursion
        search a[first] through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

Checking the Recursion

- Check binary search against criteria:

1. No infinite recursion:
    - Each call increases first or decreases last
    - Eventually first will be greater than last
2. Stopping cases perform correct action:
    - If first > last → no elements between them, so key can't be there!
    - if key == a[mid] → correctly found!
3. Recursive calls perform correct action
    - If key < a[mid] → key in 1st half – correct call
    - If key > a[mid] → key in 2nd half – correct call

Execution of the Function search



Efficiency of Binary Search

- Compared with linear search, binary search is very efficient (But it requires that the list is sorted!)

- Half of array eliminated at start!
  - Then a quarter, then 1/8, etc
  - Essentially eliminate half with each call

- Example:

  Array of 100 elements (problem size n = 100)
  - Binary search never needs more than 7 compares!
    - Logarithmic efficiency (**log n**)

Efficiency of Algorithms

- **time efficiency** and space efficiency
- **How can we measure how long it takes for an algorithm to solve a problem of a certain size?**
  - Usually we measure the number of critical steps needed based on the problem size n, which can be expressed as a function on n: f(n)
  - Usually, we investigate the algorithm efficiency in the worst case scenario. That is, we seek a function f(n) to describe the time efficiency of an algorithm when the algorithm is used to solve a problem in the worst case.
  - Our major concern is when problem size n gets large, whether or not the algorithm is workable on the problem

Efficiency of Algorithms (cont)

- We use a series of reference functions in terms of **big *O*** notation to measure the efficiency of individual algorithms

Function	Name of function
O (1)	constant
O (log n)	logarithmic
O (n)	linear
O (n log n)	linearithmic
O (n^2)	quadratic
O (n^c); c > 1	polynomial
O (2^n)	exponential
O (n!)	factorial
O (n^n)	n to the n

- A description of a function in terms of big *O* notation provides an upper bound on the growth rate of the function

Other Recursive Algorithms

- Exercises on Recursion …

- Now, let’s check out another recursive algorithm:

  `merge sort`

  (<u>Quick sort</u> is also recursive, and usually it is very fast.)