Recursion

Modified from Chapter 14

Recursive Functions Basics

Modified from Sections 14.1, 14.2

Recursive Functions for Tasks

- A recursive function contains a call to itself
- When breaking a task into subtasks, it may be that the subtask is a smaller example of the same task
  - Searching an array could be divided into searching the first and second halves of the array
  - Searching each half is a smaller version of searching the whole array
  - Tasks like this can be solved with recursive functions

Case Study:
Vertical Numbers

- Problem Definition:

  -
    ```
    void write_vertical( int n );
    // Precondition:  n >= 0
    // Postcondition: n is written to the screen vertically
    //                with each digit on a separate line
    ```

Case Study: Vertical Numbers

- Algorithm design:
  - Simplest case:

    If n is one digit long, write the number
  - Typical case:

    1) Output all but the last digit vertically
    2) Write the last digit
    - Step 1 is a smaller version of the original task
    - Step 2 is the simplest case

Case Study: Vertical Numbers (cont.)

- Translating the algorithm into C++
  - n / 10 returns n with the last digit removed
    - 124 / 10 = 12
  - n % 10 returns the last digit of n
    - 124 % 10 = 4
- Removing the first digit would be just as valid for defining a recursive solution
  - It would be more difficult to translate into C++

Case Study: Vertical Numbers (cont.)

- The write_vertical algorithm:

  -
    ```
    if (n < 10)
    {
         cout << n << endl;
    }
    else  // n is two or more digits long
    {
         write_vertical(n / 10);
         cout << n % 10 << endl;
    }
    ```

Tracing a Recursive Call

- write_vertical(123)

```
if (123 < 10)
{
        cout << 123 << endl;
}
else // n is more than two digits
{
        write_vertical(123/10);
        cout << (123 % 10) << endl;
}
```

Calls write_vertical(12)

resume

Function call ends

Output 3

Tracing write_vertical(12)

- write_vertical(12)
```
if (12 < 10)
{
        cout << 12 << endl;
}
else // n is more than two digits
{
        write_vertical(12/10);
        cout << (12 % 10) << endl;
}
```

Calls write_vertical(1) ▶

resume

Output 2

◀ **Function call ends**

Tracing write_vertical(1)

- write_vertical(1)

```
if (1 < 10)
{
        cout << 1 << endl;
}
else // n is more than two digits
{
        write_vertical(1/10);
        cout << (1 % 10) << endl;
}
```

Simplest case is now true

Output 1

Function call ends

A Closer Look at Recursion

- write_vertical uses recursion
  - Used no new keywords
  - It simply called itself with a **different** argument
- Recursive calls are tracked by
  - Temporarily stopping execution at the recursive call
    - The result of the call is needed before proceeding
  - Saving information to continue execution later
  - Evaluating the recursive call
  - Resuming the stopped execution

How Recursion Ends

- Eventually one of the recursive calls must not depend on another recursive call
- Recursive functions are defined as
  - One or more cases where the task is accomplished by using recursive calls to do a smaller version of the task
  - One or more cases where the task is accomplished without the use of any recursive calls. These are called base cases or stopping cases.

"Infinite" Recursion

- A function that never reaches a base case, in theory, will run forever
- In practice, the computer will often run out of resources and the program will terminate abnormally

Example: Infinite Recursion

- Function write_vertical, without the base case

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

  will eventually call write_vertical(0), which will call write_vertical(0),which will call write_vertical(0), which will call write_vertical(0), which will call write_vertical(0), which will call write_vertical(0), which will call write_vertical (0), …

Stacks for Recursion

- Computers use a structure called a stack to keep track of recursion
- Whenever a function is called, the computer uses a "clean sheet of paper"
  - The function definition is copied to the paper
  - The arguments are plugged in for the parameters
  - The computer starts to execute the function body

Stacks and The Recursive Call

- When execution of a function definition reaches a recursive call
  - Execution stops
  - Information is saved on a "clean sheet of paper" to enable resumption of execution later
  - This sheet of paper is placed on top of the stack
  - A new sheet is used for the recursive call
    - A new function definition is written, and arguments are plugged into parameters
    - Execution of the recursive call begins

The Stack and Ending Recursive Calls

- When a recursive function call is able to complete its computation with no recursive calls
  - The computer retrieves the top "sheet of paper" from the stack and resumes computation based on the information on the sheet
  - When that computation ends, that sheet of paper is discarded and the next sheet of paper on the stack is retrieved so that processing can resume
  - The process continues until no sheets remain in the stack

Activation Frames

- The computer does not use paper
- Portions of memory are used
  - The contents of these portions of memory is called an activation frame
  - The activation frame does not actually contain a copy of the function definition, but references a single copy of the function

Stack Overflow

- Because each recursive call causes an activation frame to be placed on the stack
  - infinite recursion can force the stack to grow beyond its limits to accommodate all the activation frames required
  - The result is a stack overflow
  - A stack overflow causes abnormal termination of the program

Recursion versus Iteration

- Any task that can be accomplished using recursion can also be done without recursion
  - A non-recursive version of a function typically contains a loop or loops
  - A non-recursive version of a function is usually called an iterative-version
  - A recursive version of a function
    - Usually runs slower
    - Uses more storage
    - May use code that is easier to write and understand

Display 14.2

Display 14.2

Back Next

DISPLAY 14.2 Iterative Version of the Function in Display 14.1

```
//Uses iostream:
void write_vertical(int n)
{
    int tens_in_n = 1;
    int left_end_piece = n;
    while (left_end_piece > 9)
    {
        left_end_piece = left_end_piece/10;
        tens_in_n = tens_in_n*10;
    }
    //tens_in_n is a power of ten that has the same number
    //of digits as n. For example, if n is 2345, then
    //tens_in_n is 1000.

    for (int power_of_10 = tens_in_n;
           power_of_10 > 0; power_of_10 = power_of_10/10)
    {
        cout << (n/power_of_10) << endl;
        n = n % power_of_10;
    }
}
```

Recursive Functions for Values

- Recursive functions can also return values
- The technique to design a recursive function that returns a value is basically the same as what you have already seen
  - One or more cases in which the value returned is computed in terms of calls to the same function with (usually) smaller arguments
  - One or more cases in which the value returned is computed without any recursive calls (base case)

Program Example: A Powers Function

- To define a new power function that returns an int, such that

  int y = power(2,3);

  places 2^3 in y
  - Use this definition:

    $$x^n = x^{n-1} * x$$
  - Translating the right side to C++ gives:

    power(x, n-1) * x
  - The base case: n == 0 and power should return 1

Display 14.3

Back   Next

The Recursive Function power

```
//Program to demonstrate the recursive function power.
#include <iostream>
#include <cstdlib>
using namespace std;

int power(int x, int n);
//Precondition: n >= 0.
//Returns x to the power n.

int main( )
{
    for (int n = 0; n < 4; n++)
        cout << "3 to the power " << n
             << " is " << power(3, n) << endl;

    return 0;
}

//uses iostream and cstdlib:
int power(int x, int n)
{
    if (n < 0)
    {
        cout << "Illegal argument to power.\n";
        exit(1);
    }

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

Sample Dialogue

```
3 to the power 0 is 1
3 to the power 1 is 3
3 to the power 2 is 9
3 to the power 3 is 27
```

Thinking Recursively

Modified from Section 14.3

Design Recursive Functions

- When designing a recursive function, you do not need to trace out the entire sequence of calls
  - Check that there is no infinite recursion
  - Check that each stopping case performs the correct action for that case
  - Check that for each recursive case: if all recursive calls perform their actions correctly, then the entire case performs correctly

Case Study: Binary Search

- A binary search can be used to search a sorted array to determine if it contains a specified value
  - The array indexes will be 0 through final_index
  - Because the array is sorted, we know
    a[0] <= a[1] <= a[2] <= … <= a[final_index]
  - If the item is in the list, we want to know where it is in the list
  - If the target value appears more than once, we only need to know one of its indices

Binary Search: Problem Definition

- The function will use two call-by-reference parameters to return the outcome of the search
  - One, called found, will be type bool. If the value is found, found will be set to true. If the value is found, the parameter, location, will be set to the index of the value
- A call-by-value parameter is used to pass the value to find
  - This parameter is named key

Binary Search Algorithm Design

- Our algorithm is basically:
  - Look at the item in the middle
    - If it is the number we are looking for, we are done
    - If it is greater than the number we are looking for, look in the first half of the list
    - If it is less than the number we are looking for, look in the second half of the list

Binary Search Algorithm Design (cont.)

- Here is a first attempt at our algorithm:

  -
    ```
    found = false;
    mid = approx. midpoint between 0 and final_index;
    if (key == a[mid])
       {
            found = true;
            location = mid;
        }
    else if (key < a[mid])
            search a[0] through a[mid -1]
    else if (key > a[mid])
            search a[mid +1] through a[final_index];
    ```

Binary Search Algorithm Design (cont.)

- Since searching each of the shorter lists is a smaller version of the task we are working on, a recursive approach is natural
  - We must refine the recursive calls
    - Because we will be searching subranges of the array, we need additional parameters to specify the subrange to search
    - We will add parameters first and last to indicate the first and last indices of the subrange

Binary Search
Algorithm Design (cont.)

- Here is our first refinement:

```
found = false;
mid = approx. midpoint between first and last;
if (key == a[mid])
{
    found = true;
    location = mid;
}
else if (key < a[mid])
   search a[first] through a[mid -1]
else if (key > a[mid])
   search a[mid +1] through a[last];
```

Binary Search Algorithm Design (cont.)

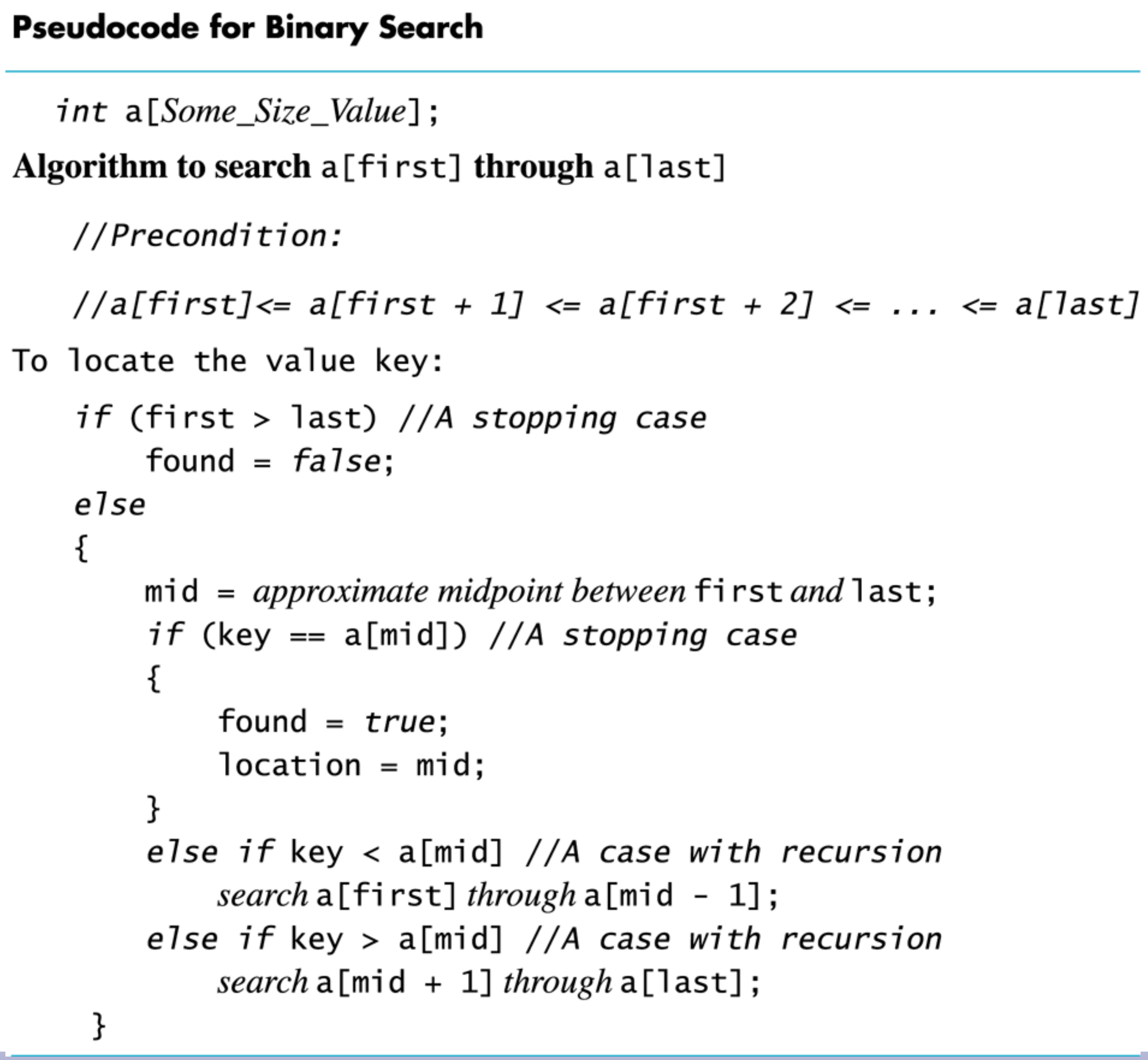
- We must ensure that our algorithm ends
  - If key is found in the array, there is no recursive call and the process terminates
  - What if key is not found in the array?
    - At each recursive call, either the value of first is increased or the value of last is decreased
    - If first ever becomes larger than last, we know that there are no more indices to check and key is not in the array
- The final pseudocode is shown in

Display 14.5

Display 14.7

Display 14.5

Back Next



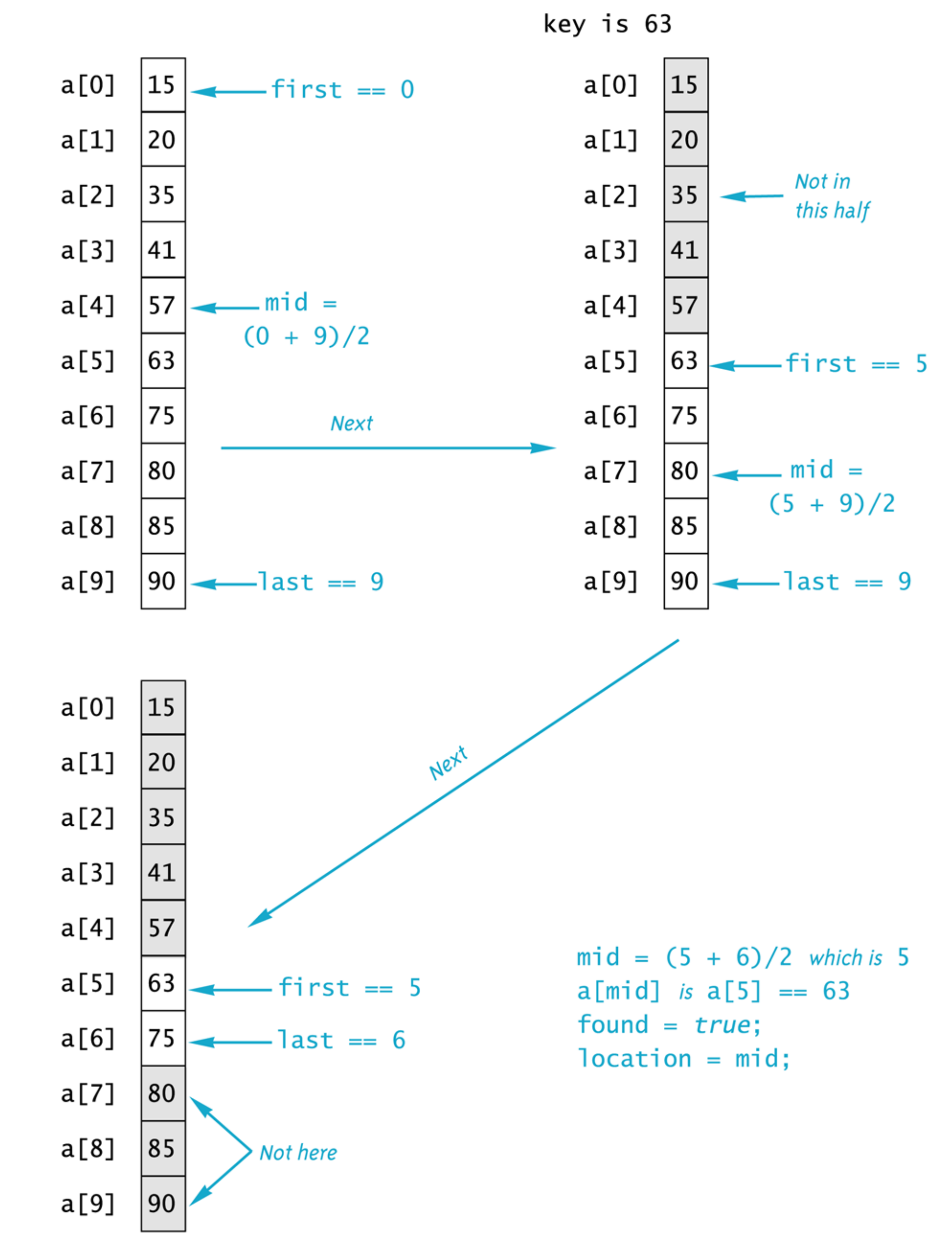
Pseudocode for Binary Search

```
  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 = 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];
     }
```

Display 14.7

Back Next

Execution of the Function search



Display 14.6

Back Next

```
void search(const int a[], int first, int last,
                                int key, bool& found, int& location)
{
    int mid;
    if (first > last)
    {
        found = false;
    }
    else
    {
        mid = (first + last)/2;

        if (key == a[mid])
        {
            found = true;
            location = mid;
        }
        else if (key < a[mid])
        {
            search(a, first, mid - 1, key, found, location);
        }
        else if (key > a[mid])
        {
            search(a, mid + 1, last, key, found, location);
        }
    }
}
```

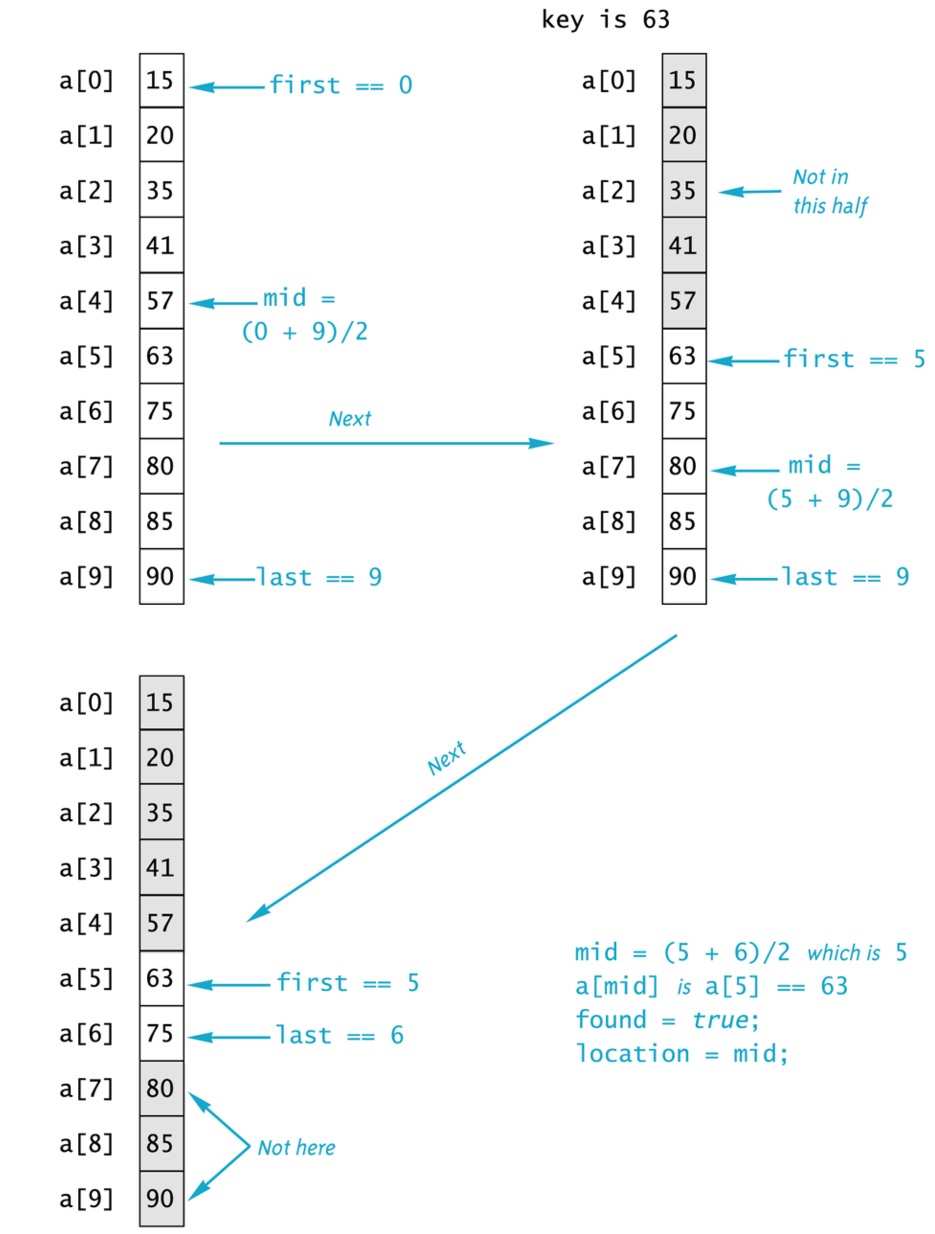
Binary Search
Checking the Recursion

- There is no infinite recursion
    - On each recursive call, the value of first is increased or the value of last is decreased. Eventually, if nothing else stops the recursion, the stopping case of first > last will be called

Display 14.7

Back Next

Execution of the Function search



Binary Search
Checking the Recursion (cont.)

- Each stopping case performs the correct action
  - If first > last, there are no elements between a[first] and a[last] so key is not in this segment and it is correct to set found to false
  - If k == a[mid], the algorithm correctly sets found to true and location equal to mid
  - Therefore both stopping cases are correct

Binary Search
Checking the Recursion (cont.)

- For each case that involves recursion, if all recursive calls perform their actions correctly, then the entire case performs correctly
  Since the array is sorted…
  - If key < a[mid], key is in one of elements a[first] through a[mid-1] if it is in the array. No other elements must be searched…the recursive call is correct
  - If key > a[mid], key is in one of elements a[mid+1] through a[last] if it is in the array. No other elements must be searched… the recursive call is correct

Binary Search Efficiency

- The binary search is extremely fast compared to an algorithm that checks each item in order
    - The binary search eliminates about half the elements between a[first] and a[last] from consideration at each recursive call
    - For an array of 100 items, the binary search never compares more than seven elements to the key
    - For an array of 100 items, a simple serial search will perform average 50 comparisons and may do as many as 100!