Recursion

CS 16: Solving Problems with Computers I
Lecture #16

Ziad Matni
Dept. of Computer Science, UCSB

Announcements

- Lab #9 is due on the last day of classes: Friday, 12/2
- Homework #15 is due on Tuesday, 11/29

Lecture Outline

- A Word About Lab 9 / Lab 10
- The Point of Pointers!

CH. 14

- Recursive Functions

About Lab9 / Lab10

- Lab 9 is a equal to the last 2 labs for the quarter
- It is worth 2x the other individual labs
  - 5 exercises utilizing vectors, dynamic arrays, and recursive functions
- Pair programming is REQUIRED!
  - Will not grade labs that are not from a pair
    - Deadline to pair up is Monday 11/28
  - The only deviations from this requirement are:
    - You are the last person to pair-up and everyone else has
    - You have *extenuating* circumstances – if so, the **instructor has to approve**.
- Lab is due on **FRIDAY, Dec. 2nd**

Remaining To-Dos

M	T	W	Th	F
11/21	**11/22** **HW #14** *Recursive functions*	**11/23**	**11/24** ***THANKSGIVING BREAK*** 	**11/25**
11/28	**11/29** **HW #15** *Structures*	**11/30**	**12/1** **HW #16** *Structures + Review for Final Exam*	**12/2** **LAB #9**
12/5	**12/6** **FINAL EXAM At 4 PM**			

Why Pointers?

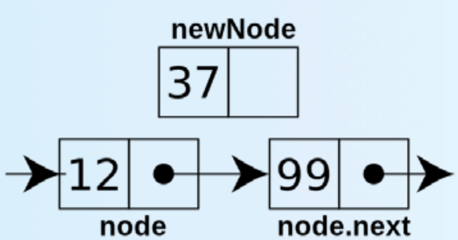
- With the creation of object-oriented programming, using pointers is not as useful as it used to be
- Use pointers mostly if you're writing a C++ program that references C libraries or older C programs
- Pointers/references are very useful when passing variables in a function that you want changed outside the function
  - a.k.a call-by-reference functions

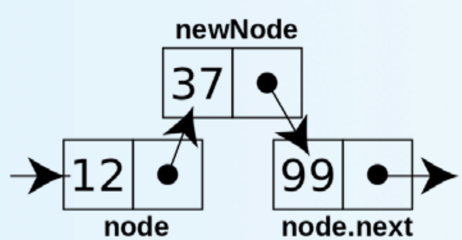
Pointers and Linked Lists

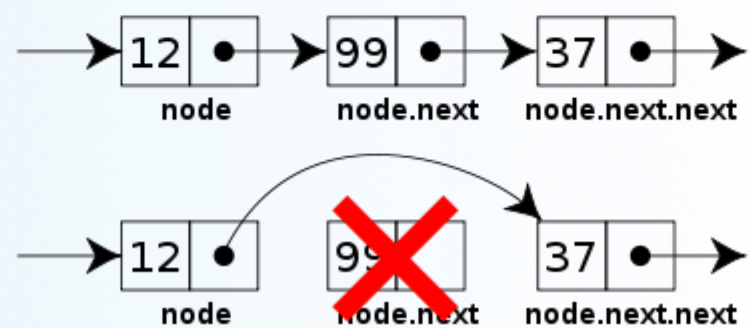
- Pointers are very useful when creating *linked lists*
- Linear collection of data elements, called *nodes*, each pointing to the next node by means of a pointer
- List elements can easily be inserted or removed without reorganization of the entire structure (unlike arrays)
- Data items in a linked list do not have to be stored in one large memory block (again, unlike arrays)

Linked Lists

- You can build a list of “nodes” which are made up of variables and pointers to create a chain.
- Adding and deleting nodes in the link can be done by “re-routing” pointer links.
- Chapter 13 in your books explains this further, but we won’t cover it in CS16







Recursive Fuctions

Recursive Functions for Tasks

- **Recursive: (adj.) Repeating unto itself**
- **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

- For example: **Searching an array**
  - Could be divided into searching the 1st, then 2nd halves of array
  - Searching each half is a smaller version of searching the whole array

Example: The Factorial Function

Recall: x! = 1 * 2 * 3 … * x

You could code this out as either ***(the following is pseudocode)***:

- A for-loop:

```
(for k=1; k < x; k++) { factorial *= k; }
```

- Or a recursion/repetition:

```
factorial(x) = x * factorial(x-1)
             = x * (x-1) * factorial (x-2)
             = etc…
               until you get to factorial(1)
```

Example: Recursive Formulas

- Recall from Math, that you can create a recursive formula from a sequence

Example:

- Consider the arithmetic sequence:

5, 10, 15, 20, 25, 30, …

- If I call $a_1 = 5$, then I can write the formula as:

$$\mathbf{a_n = a_{n-1} + 5}$$

Case Study: Vertical Numbers

- Problem Definition:
  Write a function that takes an integer number and prints it out one digit at a time vertically :

```
write_vertical(3):
3
write_vertical(12):
1
2
write_vertical(123):
1
2
3
```

```
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

Analysis:

- Take a number, like 543.
- How do I separate the digits from each other?
  - So that I can print out **5**, then **4**, then **3**?

- Note that 543 = 500 + 40 + 3

Case Study: Vertical Numbers

Algorithm design

- *Simplest case*:
  If **n** is 1 digit long, just write the number

- *More typical case*:
  1) Output all but the last digit vertically (recursion!)
  2) Write the last digit
  - *Step 1 is a smaller version of the original task*
    - *The recursive case*
  - *Step 2 is the simplest case*
    - *The base case*

Case Study: Vertical Numbers

*The **write_vertical** algorithm (in pseudocode):*

```
void write_vertical( int n ) {

if (n < 10)  cout << n << endl;
 // n < 10 means n is only one digit

else  // n is two or more digits long
  {
        write_vertical(n with the last digit removed);
        cout << the last digit of n << endl;
  }

}
```

Case Study: Vertical Numbers

- Note that: **n / 10** returns **n** with *the least-significant digit removed*
  - So, for example, **124 / 10** = **12**
- Whereas: **n % 10** returns the *last digit of **n***
  - In this example, **124 % 10 = 4**

I've separated the last digit from the other digits!

- *Another way to do this:*
  Remove the first (most-significant) digit would be just as valid for defining a recursive solution
  - However, this would be more difficult to translate into C++

DEMO!

A Closer Look at Recursion

- The function **write_vertical** uses recursion
  - Used no new keywords or anything "new"
  - It simply called itself with a different argument

- If you want to ***track*** a recursive call:
  - Temporarily stop the execution *at* the recursive call
  - Show or save the result of the call before proceeding
  - Evaluate the recursive call
  - Resume the stopped execution

How Recursion Ends

- Recursive functions have to stop eventually
  - One of the recursive calls must not depend on another recursive call
  - Usually, it's the last 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 (i.e. memory usually) and the program will terminate abnormally

Example: Infinite Recursion

- What if we wrote the function **write_vertical**, *without the base case*

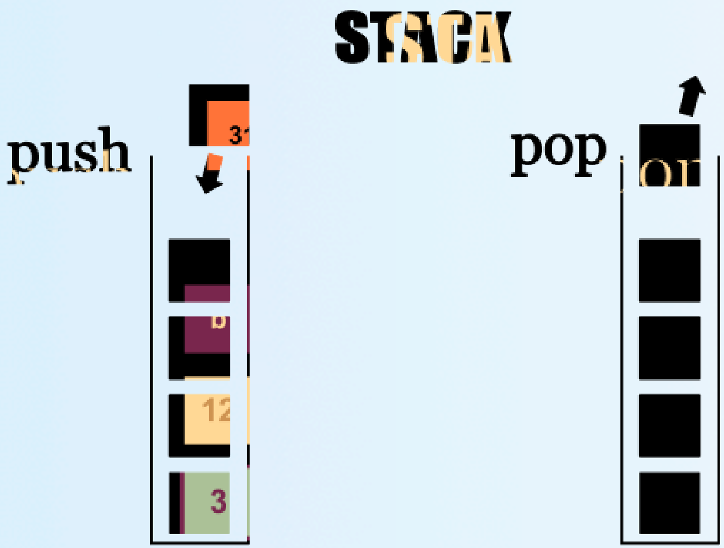
```
void write_vertical(int n) {
    write_vertical (n / 10);
    cout << n % 10 << endl;  }
```

- Will *eventually* call **write_vertical(0)**,
  which will call **write_vertical(0)**,
  which will call **write_vertical(0)**,
  which will call **write_vertical(0)**, …etc…

Stacks for Recursion

- Computers use a structure called a ***stack*** to keep track of recursion
- **Stack**:
  a memory structure analogous to a ***stack of paper***
  - To place information on the stack,
    write it on a piece of paper and place it on **top** of the stack
  - To **insert *more*** information on the stack,
    use a clean sheet of paper,
    write the information, and place it on the **top** of the stack
  - To **retrieve** information, only the top sheet of paper can be read,
    and then thrown away when it is no longer needed

LIFO



LIFO (Last In First Out)

- This scheme of handling sequential data in a stack is called:

  Last In-First Out (**LIFO**)

- The other common scheme in CS data organization is FIFO (First In-First Out)

Stacks & Making the Recursive Call

- When execution of a function definition reaches a recursive call
  1. Execution is halted
  2. Then, data is saved on a “clean sheet of paper” to enable resumption of execution later
  3. This sheet of paper is placed *on top of the stack*
  4. Then a *new* sheet is used for the recursive call
     a) A new function definition is written, and arguments are plugged into parameters
     b) Execution of the recursive call begins
  5. And it goes on…

Stacks & 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
  - Resumes computation based on the information on the sheet
- When that computation ends, that sheet of paper is discarded
- 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

- Instead of “paper”, think “memory”…

- Portions of memory are used for the stack
  – 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 (instantiation) of the function

Stack Overflow

- Because each recursive call causes an **activation frame** to be placed on the stack
  - Infinite recursions can force the stack to grow **beyond** its limits

- The result of this erroneous operation is called a ***stack overflow***
  - This causes abnormal termination of the program

Image from stackoverflow.com

Recursion versus Iteration

Algorithmic Truism:

- Any task that can be accomplished using recursion can also be done without recursion
  - Recall the 2 demos I showed you…

- A non-recursive version of a function typically contains loop(s)

- 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*

Recursive Functions for Values

Recursive Functions for *Values*

- Recursive functions don't have to be **void** types
  - They can also return values
- The technique to design a recursive function that returns a value is basically the same…
  - 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

Example: Define a new **power** function (not the one in <cmath>)

- Let it return an integer, 2^3 ,when we call the function as:
  int y = power(2,3);
  - Use the following definition:
    $x_n = x_{n-1} * x$ *i.e.* $2^3 = 2^2 * 2$
    - Note that this only works if n is a positive number
  - Translating the right side of that equation into C++ gives:
    power(x, n-1) * x
  - The base/stopping case:
    when n is 0, then power() should return 1

```
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;
}
```

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
```

Stopping case

```
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

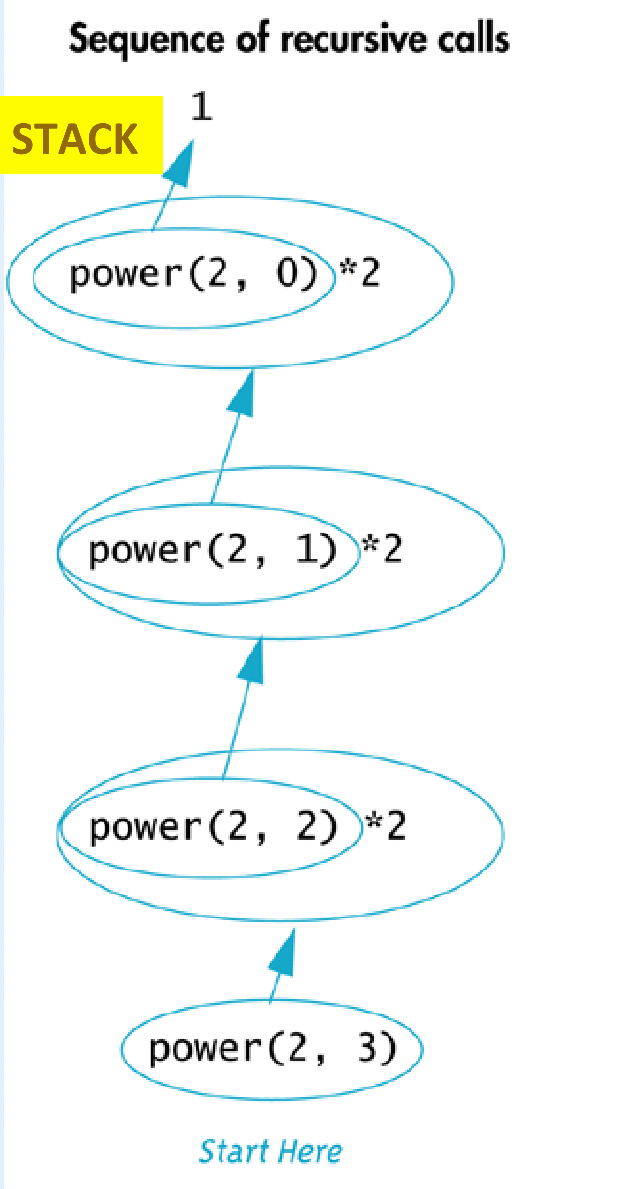
```
3 to the power 0 is 1
3 to the power 1 is 3
3 to the power 2 is 9
3 to the power 3 is 27
```

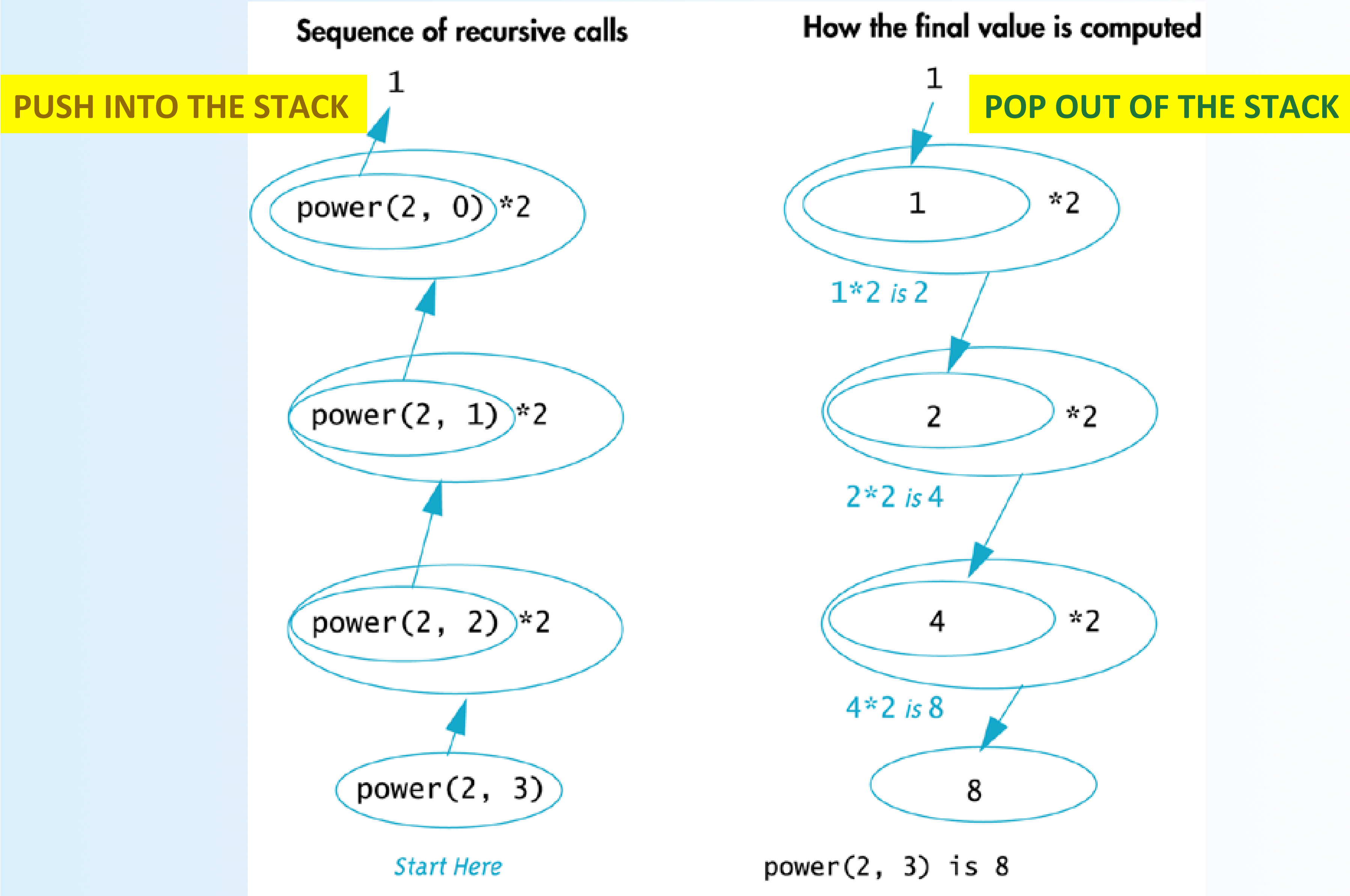
Stopping case

Tracing *power(2, 3)*

- **power(2, 3)** results in the following recursive calls:
  - power(2, 3) is power(2, 2) * 2
  - power(2, 2) is power(2, 1) * 2
  - power(2, 1) is power(2, 0) * 2
  - power (2, 0) is 1 (stopping case)

PUSH INTO THE STACK




Sequence of recursive calls
PUSH INTO THE STACK
1
power(2, 0) *2
power(2, 1) *2
power(2, 2) *2
power(2, 3)
Start Here
How the final value is computed
POP OUT OF THE STACK
1
1 *2
1*2 is 2
2 *2
2*2 is 4
4 *2
4*2 is 8
8
power(2, 3) is 8

Thinking Recursively

Thinking Recursively

- When designing a recursive function, you do not need to trace out the entire sequence of calls
  - Check that there is no infinite recursion: i.e. that, eventually, a stopping case is reached
  - Check that each stopping case returns the correct value
  - For cases involving recursion: if all recursive calls return the correct value, then the final value returned is the correct value

Reviewing the **power** function

- There is no infinite recursion in that function ✓
- Notice that the 2nd argument is decreased at each call. ✓
  - Eventually, the 2nd argument must reach 0, the stopping case

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

- Each stopping case returns the correct value ✓
  - Example: Does **power(x, 0)** return $x^0 = 1$?

Case Study: Binary Search

- A binary search (not to be confused with binary numbers) 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

Binary Search: Problem Definition

- The function will use 2 call-by-reference parameters to return the outcome of the search
  - One parameter, ***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
  - We will call this parameter: ***key***

Binary Search: Problem Definition

- Pre and Postconditions for the function:

```
//precondition:  a[0] through a[final_index] are
//              sorted in increasing order

//postcondition: if key is not in a[0] thru a[final_index]
//          found == false;   otherwise found == true
```

Binary Search: Algorithm Design

Our algorithm:

N1	N2	N3	N4	N5	N6	N7	N8	N9	N10	N11	N12	N13
first						middle						last

- Start by looking at the item in the middle of the list:
  - 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 1st half of the list**
  - If it is less than the number we are looking for, **look in the 2nd half of the list**

Binary Search: Algorithm Design

1st attempt at the 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

- Since searching each of the shorter lists is a smaller version of the task we are working on, a recursive approach is natural
  - Keep dividing list in half and go again until you find it

- We must refine the recursive calls in our algorithm
  - Because we will be searching sub-ranges of the array, we need additional parameters to specify the sub-range to search
  - We will add parameters ***first*** and ***last*** to indicate the first and last indices of the sub-range

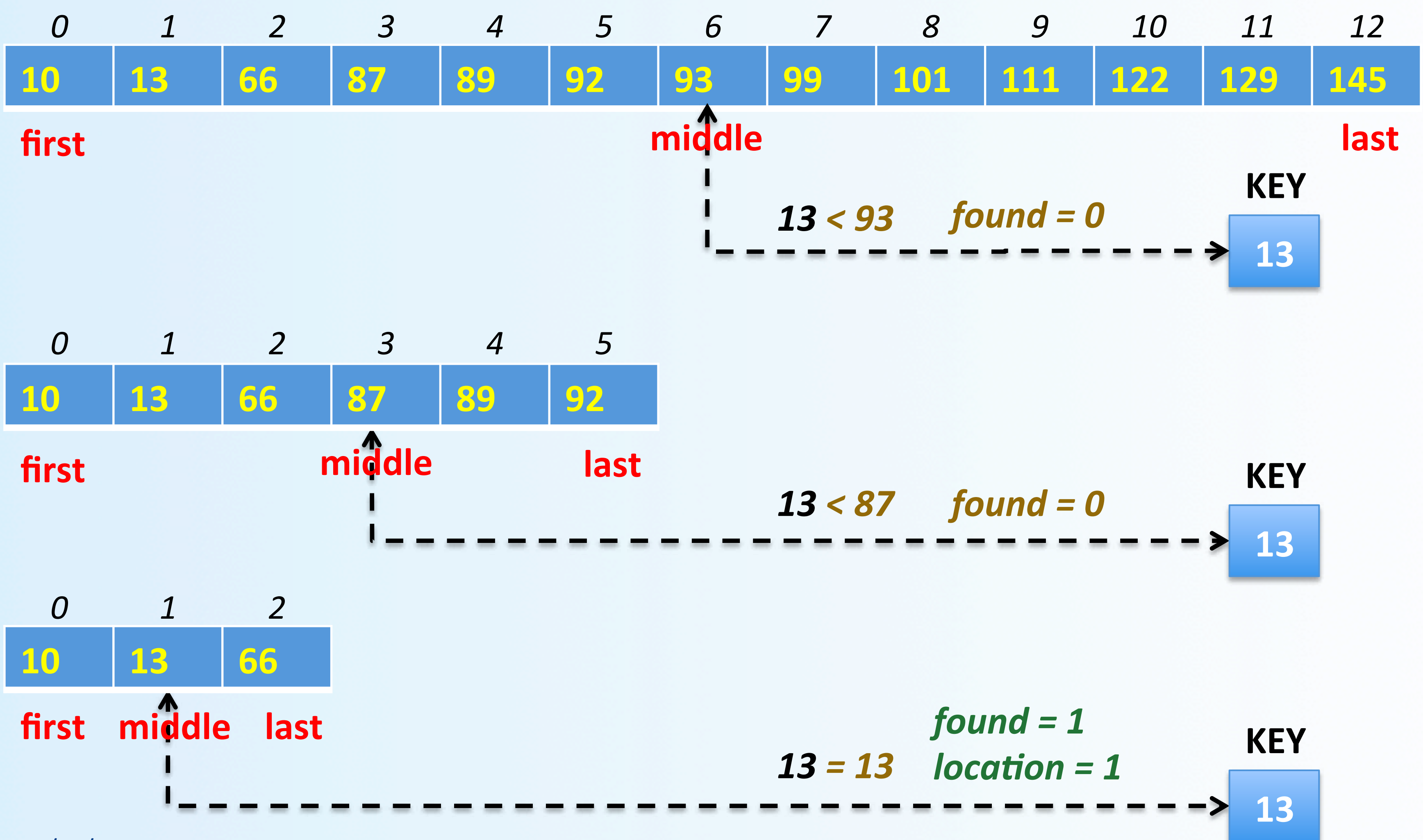
Binary Search: Algorithm Design

Here is our first refinement:

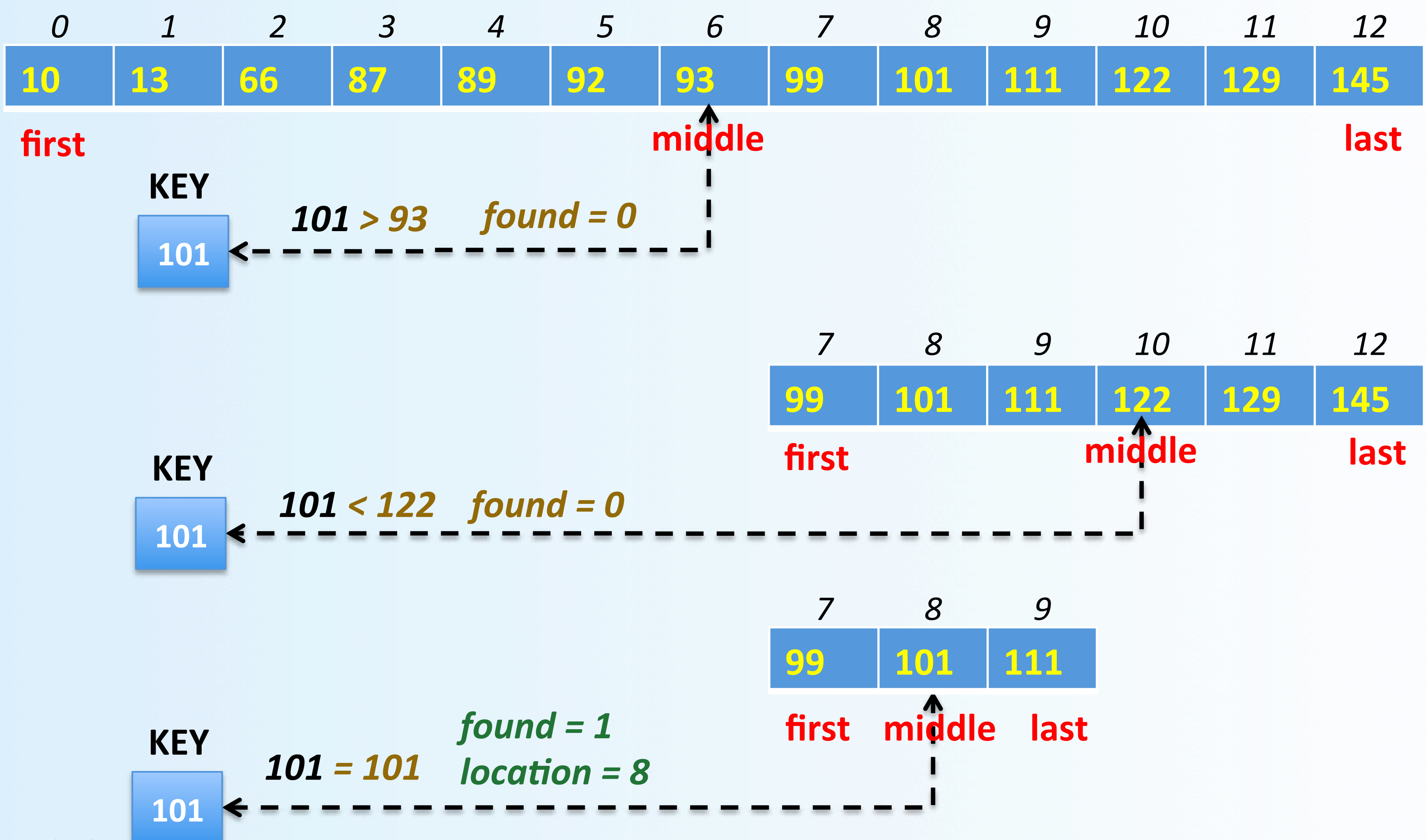
```
found = false;
mid = approx. midpoint between 0 and final_index;

if (key == a[mid])  {
   found = true;
   location = mid;
}

else if (key < a[mid])
        search a[first] through a[mid -1]

else if (key > a[mid])
        search a[mid +1] through a[last];
```

Instead of going from
0 to mid-1
and then from
mid-1 to final_index

Binary Search: A Visualization 1



Binary Search: A Visualization 2



Binary Search: Algorithm Design

- We must ensure that our algorithm eventually ends
  - No infinite recursions!
- 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

Binary Search: Writing the Code

- Function **search** implements the algorithm:

```
void search(const int a[ ], int first, int last,
            int key, bool& found, int& location);

//precondition:  a[0] through a[final_index] are
//                          sorted in increasing order

//postcondition: if key is not in a[0] - a[final_index]
//                          found = = false; otherwise
//                          found = = true
```

- See **Display 14.6** in Chapter 14 for full program

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

Binary Search:
Checking the Recursion

- Each stopping case performs the correct action

  - If **first** > **last**, then there are no more 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

- 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]**, then **key** is in one of elements **a[first]** through **a[mid-1]** if it is in the array.
    No other elements need 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** algorithm 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, a simple serial search will average **50** comparisons and may do as many as **100**!
  - N items, N max. comparisons

- For an array of **100** items, the binary search algorithm never compares more than **7** elements to the key!
  - N items, $\log_2 N$ max. comparisons

Binary Search: An Iterative Version

- The iterative version of the binary search may run faster on some systems
  - Iterative vs Recursive is not always a decisive speed decision
- The algorithm for the iterative version is shown in Display 14.8 of the textbook
  - It was created by mirroring the recursive function
- Even if you plan an iterative function, it may be helpful to start with the recursive approach

Iterative Version of Binary Search

Function Declaration

```
void search(const int a[], int low_end, int high_end,
                           int key, bool& found, int& location);
//Precondition: a[low_end] through a[high_end] are sorted in increasing
//order.
//Postcondition: If key is not one of the values a[low_end] through
//a[high_end], then found == false; otherwise, a[location] == key and
//found == true.
```

Function Definition

```
void search(const int a[], int low_end, int high_end,
                           int key, bool& found, int& location)
{
    int first = low_end;
    int last = high_end;
    int mid;

    found = false;//so far
    while ( (first <= last) && !(found) )
    {
        mid = (first + last)/2;
        if (key == a[mid])
        {
            found = true;
            location = mid;
        }
        else if (key < a[mid])
        {
            last = mid - 1;
        }
        else if (key > a[mid])
        {
            first = mid + 1;
        }
    }
}
```

To Dos

- Homework #14 for next Tuesday

- Lab #9:
  Make sure you pick your partner by Monday!!

HAPPY THANKSGIVING!

</LECTURE>