goo.gl/plnAFk

Lab #14

Date: 2014/05/28

goo.gl/plnAFk

plnAFk

Recursive

Recursive Function(1/2)

- A recursive function calls itself.
- Each recursive call solves an identical, but smaller problem.
- A test for the base case enables the recursive calls to stop.

Recursive Function (2/2)

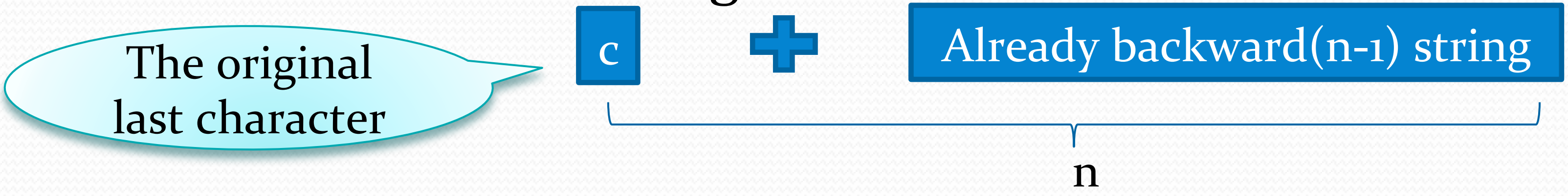
- Four questions:
  1. How can you define the problem in terms of a smaller problem of the same type?
  2. How does each recursive call diminish the size of the problem?
  3. What instance of the problem can serve as the base case?
  4. As the problem size diminishes, will you reach this base case?

A Recursive `void` Function: Writing a String Backward (1/11)

- **Problem**:
  - Given a string of characters, write it in reverse order.
    - E.g., "cat" → "tac".
- How can you write an n-character string backward, if you can write an (n-1)-character string backward?
- The base case: Write the empty string backward

A Recursive `void` Function: Writing a String Backward (2/11)

- Consider stripping away the ***last*** character:
  - For the solution to be valid, you must write the last character in the string first.


The original last character

c + Already backward(n-1) string

n

```
writeBackward(in s:string)

   if (the string is empty)
          Do nothing - this is the base case
   else
   {
          Write the last character of s
          writeBackward(s minus its last character)
   }
```

strip away the last character

A Recursive `void` Function: Writing a String Backward (3/11)

WriteBackward.cpp

- The C++ function `writeBackward`:

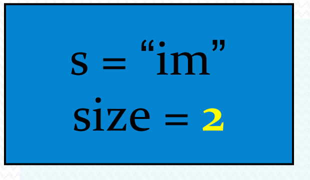
```
/** Writes a character string backward.
 * @param s  The string to write backward.
 * @param size  The length of s. */
void writeBackward(string s, int size)
{
   if (size > 0)
   {  // write the last character
      cout << s.substr(size-1, 1);
   //印出從0算起第(size-1)個字元開始,長度為1個字元的子字串

      // write the rest of the string backward
      writeBackward(s.substr(0, size-1), size-1);  // Point A,
                          //means call the recursive function
   }  // end if
   else ;
   // size == 0 is the base case - do nothing
}  // end writeBackward
```

A Recursive `void` Function: Writing a String Backward (4/11)

- Box trace of `writeBackward(“im”,2)`:
    - Each box contains the local environment of the recursive call:
        - The input arguments `s` and `size`.

A Recursive `void` Function: Writing a String Backward (5/11)

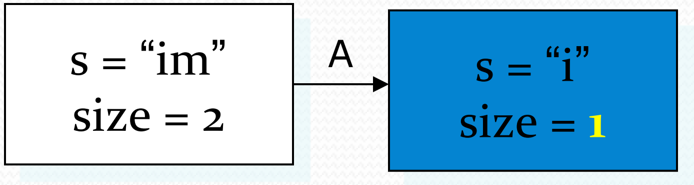
The initial call is made, and the function begins execution:


s = “im”
size = 2

Output line: m

Point A (`writeBackward(s, size-1)`) is reached, and the recursive call is made.

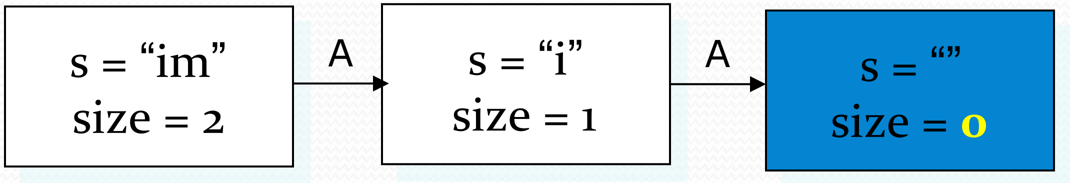
The new invocation begins execution:


s = “im”
size = 2
A
s = “i”
size = 1

Output line: mi

Point A is reached, and the recursive call is made.

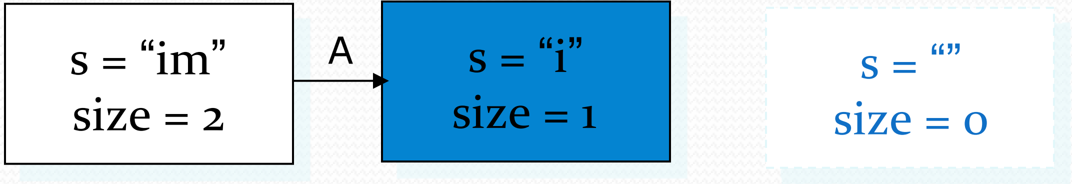
The new invocation begins execution:


s = “im”
size = 2
A
s = “i”
size = 1
A
s = “”
size = 0

A Recursive `void` Function: Writing a String Backward (6/11)

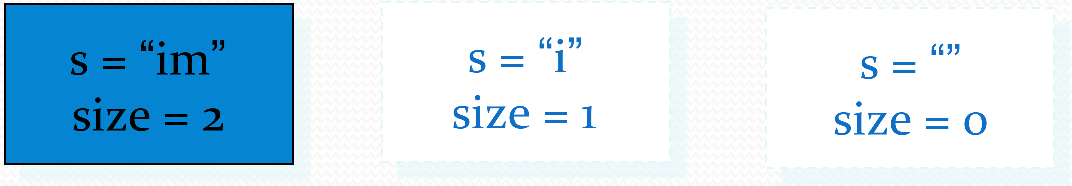
This is the base case, so this invocation completes.

Control returns to the calling box, which continues execution:



This invocation completes.

Control returns to the calling box, which continues execution:



This invocation completes. Control returns to the statement following the initial call.

Search

Searching an Array:
Binary Search (1/6)

- **A high-level binary search**:
  - Search a **sorted** array of integers for a given value.
    - *anArray[0] <= anArray[1] <= … <= anArray[size-1]*

```
    binarySearch(in anArray:ArrayType, in value:ItemType)
      if (anArray is of size 1)
         Determine if anArray’s item is equal to value
      else
      {  Find the midpoint of anArray
         Determine which half of anArray contains value
         if (value is in the first half of anArray)
           binarySearch(first half of anArray, value)
         else
           binarySearch(second half of anArray, value)
      }
```

Searching an Array:
Binary Search (2/6) Implementation details

- How will you pass "half of `anArray`" to the recursive calls to `binarySearch`?
  - `binarySearch(anArray, from, to, value)`
    - Pass the entire array at each call, but have `binarySearch` search only `anArray[from…to]`.

  - The midpoint in `[from…to]` is:
    - `mid = (from + to) / 2`

  - Then `binarySearch(first half of anArray, value)`: `binarySearch(anArray, from, mid-1, value)`

  - `binarySearch(second half of anArray, value)`: `binarySearch(anArray, mid+1, to, value)`

Searching an Array:

Binary Search (3/6) Implementation details

- How do you determine which half of the array contains `value`?
  - The first half will be: `if (value < anArray[mid]).`
  - Remember to test the equality between *`value`* and *`anArray[mid]`*.

- What should the base case(s) be?
  - `Length of search space == 1`
    - `value == anArray[mid]`
      - When value is in the array.
    - `Value != anArray[mid]`
      - When value is not in the array.

Binary Search

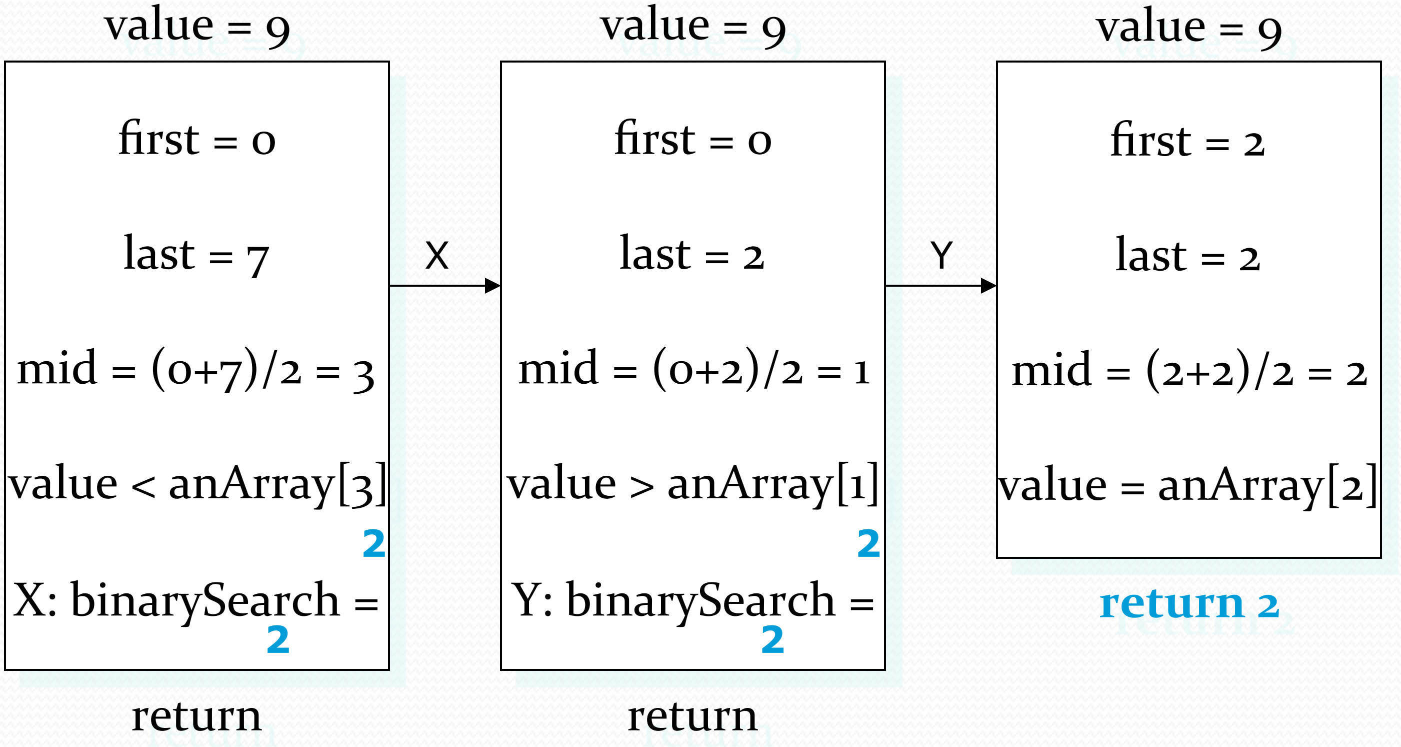
- `binarySearch.cpp`

Searching an Array:
Binary Search (4/6) Implementation details

- How will `binarySearch` indicate the result of the search?
  - A negative value if it does not find value in the array.
  - The index of the array item that is equal to value.

Searching an Array:
Binary Search (5/6)

- Box trace of `binarySearch`:
  - `anArray = <1, 5, 9, 12, 15, 21, 29, 31>`
  - Search for `9`.



value = 9

first = 0

last = 7

mid = (0+7)/2 = 3

value < anArray[3]

2

X: binarySearch =

2

return

X

value = 9

first = 0

last = 2

mid = (0+2)/2 = 1

value > anArray[1]

2

Y: binarySearch =

2

return

Y

value = 9

first = 2

last = 2

mid = (2+2)/2 = 2

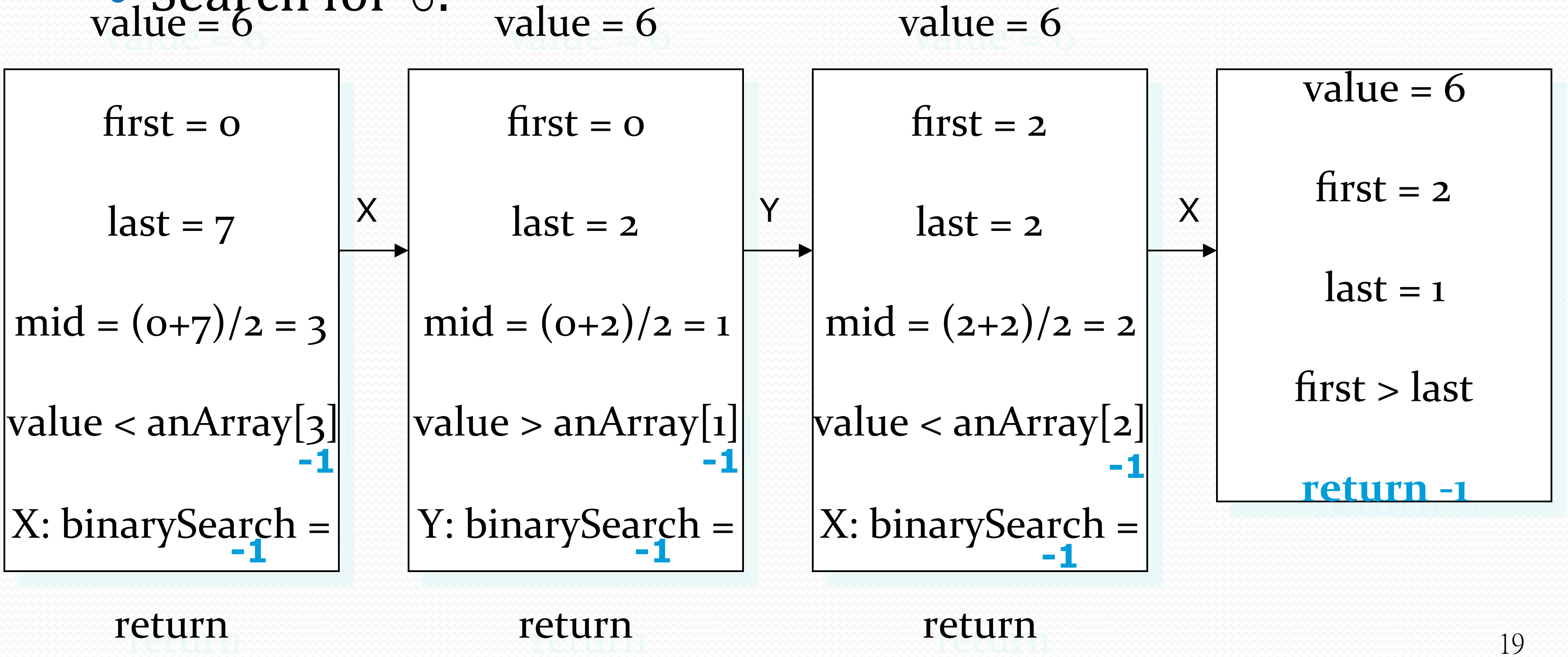
value = anArray[2]

return 2

Searching an Array:

Binary Search (6/6)

- Box trace of `binarySearch`:
  - `anArray = <1, 5, 9, 12, 15, 21, 29, 31>`
  - Search for `6`.



value = 6
first = 0
last = 7
mid = (0+7)/2 = 3
value < anArray[3]
-1
X: binarySearch = -1
return

X →

value = 6
first = 0
last = 2
mid = (0+2)/2 = 1
value > anArray[1]
-1
Y: binarySearch = -1
return

Y →

value = 6
first = 2
last = 2
mid = (2+2)/2 = 2
value < anArray[2]
-1
X: binarySearch = -1
return

X →

value = 6
first = 2
last = 1
first > last
return -1