Chapter 2-1, 2, & 3, Lecture notes

Recursive solutions

2. Basic understanding of recursion – assumes that you have had little or no previous introduction to recursion

   2.1 Recursion Solutions

   - 2.1.1 A power problem-solving tool
   - 2.1.2 Complicated problems solved readily by recursion
   - 2.1.3 Break the problem into smaller problems
     - 2.1.3.1 Smaller problems are identical to the larger one (reflection in a mirror)
     - 2.1.3.2 Mirror in front of a mirror
       - 2.1.3.2.1 Each image is smaller
       - 2.1.3.2.2 Problem also becomes so small that the solution is known
   - 2.1.4 Recursion can seem like a "trick" but it's an alternative to **iteration**
     - 2.1.4.1 An iterative solution involves loops
     - 2.1.4.2 Recursive solutions can be very simple and solve complex problems elegantly
     - 2.1.4.3 However, some recursive solutions are impractical when compared to iterative solutions
   - 2.1.5 Sequential search of a dictionary
     - 2.1.5.1 Example of iteration, can do the brute force method of looping through all the words in a dictionary – **a sequential search**
     - 2.1.5.2 But words in a dictionary are sorted alphabetically thus can use a "binary search"
       - 2.1.5.2.1 Binary search can also be done nicely via recursion
       - 2.1.5.2.2 The idea is to determine which "half" of the dictionary to search in and then keep dividing
       - 2.1.5.2.3 A first revision of the algorithm is:

```
// Search a dictionary for a word by using a recursive binary search
if (the dictionary contains only one page)
        scan the page for the word
else
{       Open the dictionary to a point near the middle
        Determine which half of the dictionary contains the word
        if (the word is in the first half of the dictionary)
                Search the first half of the dictionary for the word
        else
                Search the second half of the dictionary for the word
```

}

2.1.5.3 This algorithm is vague (how do you scan a page?)
2.1.5.4 The dictionary is divided again and again
2.1.5.5 When the problem cannot get any smaller, it is called the "basis or degenerate case" or "base case"
2.1.5.6 In other words the recursive code would stop being recursive (avoid going forever).
2.1.5.7 Recursive solution could be shown like so from Figure 2-1

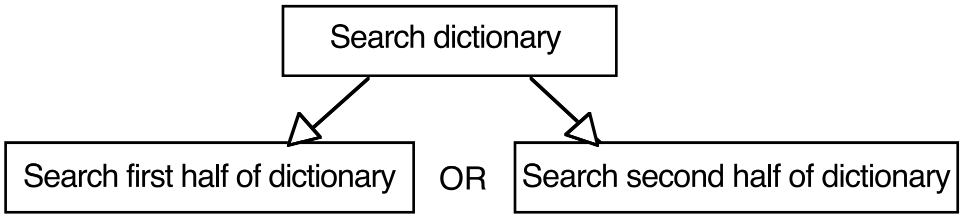

Search dictionary
Search first half of dictionary
OR
Search second half of dictionary

Figure 2-1, recursive solution

2.1.6 That solution uses the "divide and conquer" strategy
2.1.6.1 Dictionary is "divided" into two halves
2.1.6.2 "Conquer" the appropriate half
2.1.6.3 Less vague solution would be:

```
search(in aDictionary:Dictionary, in word: string)

    if (aDictionary is one page in size)
        Scan the page for word
`   else
    {   Open aDictionary to a point near the middle
        Determine which half of aDictionary contains word

        if (word is in the first half of aDictionary)
            search(first half of aDictionary, word)
        else
            search(second half of aDictionary, word)
    }
```

2.1.7 Items to note:
2.1.7.1 See where the function search would call itself (a must with recursion)
2.1.7.2 Each call to search passes the dictionary that is half of what it was – makes the problem "smaller"
2.1.7.3 When the size of the dictionary is one page, you can then switch to scan one page
2.1.7.4 When the problem gets smaller, you will hit the base page

2.1.8 There are two different kinds of recursive function:
   2.1.8.1 One kind of recursive function returns a value – **valued function**
   2.1.8.2 Recursive function that does not return a value – **void function**

2.2 Recursion That Return a Value

2.2.1 Recursive Valued Function: Factorial of n
   2.2.1.1 No advantage of doing Factorial iteratively or recursively
   2.2.1.2 But it's a good one for recursion
   2.2.1.3 Definition of factorial would be:

factorial(n) = n * (n – 1) * (n – 2) * … * 1 for any integer n > 0
factorial(0) = 1

2.2.2 The definition of *factorial(n)* in terms of *factorial(n-1)* is an example of **recurrence relation**
   2.2.2.1 Definition of *factorial(n)* lacks one key element – the base case
   2.2.2.2 A revised definition would have the base case:

$$factorial(n) = \begin{cases} 1 & \text{if n = 0} \\ n \times factorial(n - 1) & \text{if n > 0} \end{cases}$$

2.2.3 Recursive definition would be:

factorial(n) = n * [(n – 1) * (n – 2) * … * 1]
             = n * factorial(n – 1)

2.2.4 "Recurrence relation" is defining factorial in terms of factorial
   2.2.4.1 Also define factorial(n – 1) in terms of factorial(n – 2) and so on)
   2.2.4.2 BUT the definition lacks the base case: factorial(0).
   2.2.4.3 Like so:

factorial(n) = 1 if n = 0
n * factorial(n – 1) if n > 0

   2.2.4.4 factorial(4) = 4 * factorial(3), factorial(3) = 3 * factorial(2), factorial(2) = 2 * factorial(1), factorial(1) = 1 * factorial(0)
   2.2.4.5 Therefore factorial(4) = 4 * 3 * 2 * 1 * 1 = 24 and the code would be:

```
/** computes the factorial of the nonnegative integer n.
 * @pre n must be greater than or equal to 0.
 * @post None.
 * @return The factorial of n; n is unchanged */
int fact(int n)
{
        if (n == 0)
                return 1;
        else
                return n * fact(n – 1);
} // end fact
```

2.2.5 Function fact fits a recursive definition because:
  2.2.5.1 fact calls itself
  2.2.5.2 With each recursive call, factorial integer is goes down by 1
  2.2.5.3 Function handles the factorial of 0 differently (the base case) so fact(0) returns a 1
  2.2.5.4 Given that n is nonnegative, item 2 assures you will always reach the base case

2.2.6 Function fact has a precondition of a nonnegative value of n
  2.2.6.1 If n was negative, the function would not work correctly
  2.2.6.2 It would go forever – the fact(-4) would call fact(-5) and so on

2.2.7 You can use the “box trace” to clarify recursion
  2.2.7.1 Each box roughly corresponds to an **activation record** (which a compiler typically uses in its implementation)
  2.2.7.2 Let's box trace the recursive function fact:
    2.2.7.2.1 Label each recursive call in the function(A, B, C, and so on) – for example:

```
if (n == 0)
  return 1;
else
  return n * fact(n – 1);  ← (A)
```

You return to point A after each recursive call, substitute computed value for term fact(n – 1), and continue execution by evaluating expression n * fact(n – 1)

    2.2.7.2.2 Represent each call to function during execution by new box with noting **local environment** of function – each function would have

- Value arguments of formal argument list
- Function's local variables
- Placeholder for value returned by each recursive call from current box
- The value of the function itself

2.2.7.2.3 Draw an arrow from statement that first calls the recursive process to first box then continue by pointing to next box like so:

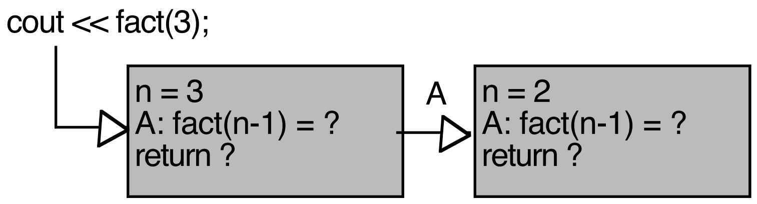
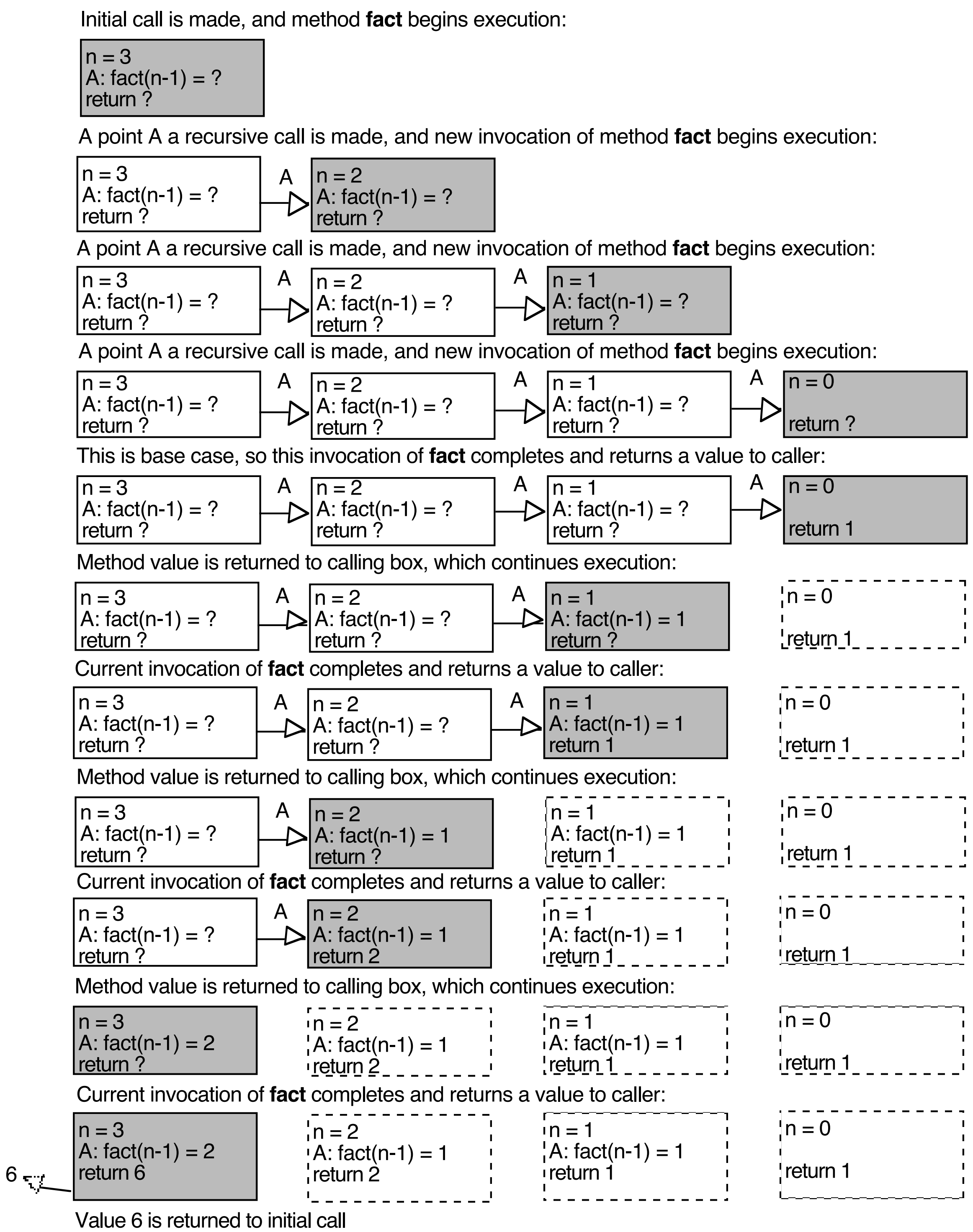


Figure 2-4, Beginning of box trace

2.2.7.2.4 Other recursive calls has an arrow from calling box (with label on arrow indicating the recursive call) to a new box. Then function starts anew.

2.2.7.2.5 On exiting the function, cross off current box and go back to the box that called it

2.2.8 We can use the box trace given by figure 2-5 like so:



Initial call is made, and method **fact** begins execution:

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

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

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

This is base case, so this invocation of **fact** completes and returns a value to caller:

Method value is returned to calling box, which continues execution:

Current invocation of **fact** completes and returns a value to caller:

Method value is returned to calling box, which continues execution:

Current invocation of **fact** completes and returns a value to caller:

Method value is returned to calling box, which continues execution:

Current invocation of **fact** completes and returns a value to caller:

Value 6 is returned to initial call

Figure 2-5, Box trace of fact(3)

2.3 Recursion That Performs an Action

2.3.1 A Recursive Void Function: Writing a String Backward

2.3.1.1 Give a string of n characters – write it in reverse order

2.3.1.2 Make the problem smaller by calling itself to write a string of n – 1 characters

- 2.3.1.2.1 Each recursive call makes the string smaller
- 2.3.1.2.2 To print "cat", you call the function to print "ca", call the function to print "c", and eventually have the base case of *write out the empty string backward*
- 2.3.1.2.3 Base case could also be print when the string has one character

2.3.1.3 To make the problem smaller, you can *strip away the last character* or *strip away the first character*

2.3.1.4 Let's do the last character solution

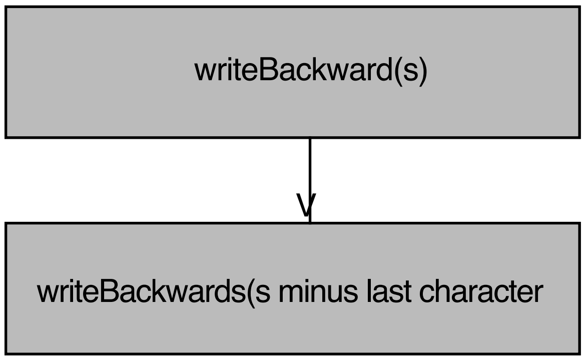


Figure 2-6, A recursive solution

- 2.3.1.4.1 Need to write the last character in string first THEN recursive call with removing a character
- 2.3.1.4.2 Like so:

```
writeBackward(s: string)

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

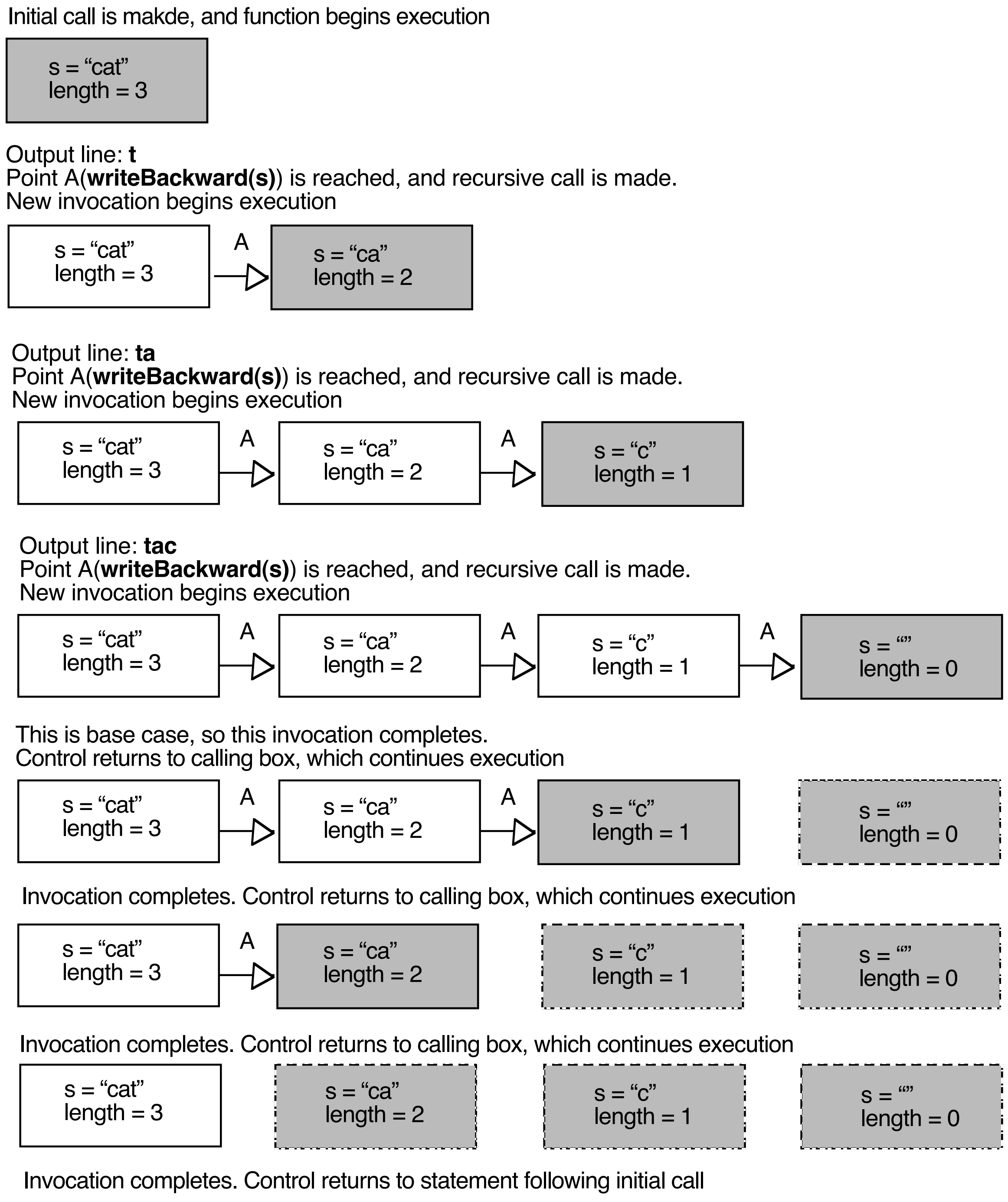
2.3.1.5 C++ substr function can be used to remove the last character
  2.3.1.5.1 The substr takes two parameters: first is the position of the characters to extract and the second is the number of characters to remove
  2.3.1.5.2 If length is the length of the string s, then s.substr(0, length-1) will give all characters minus the last character (string characters are counted from position 0, like an array)
  2.3.1.5.3 The s.substr(length – 1, 1) would give the last character
  2.3.1.5.4 Therefore the C++ function looks like:

```
/** Writes a character string backward.
 *  @pre  The string s to write backward
 *  @post  None
 *  @parm s  The string to write backward */
void writeBackward(string s)
{
        int length = s.size(): // length of string
        if (length > 0)
        {       // write the last character
                cout << s.substr(length – 1, 1);

                // write the rest of the string backward
                writeBackward(s.substr(0, length – 1));  // Point A
        } // end if

        // length == 0 is the base case – do nothing
}       // end writeBackward
```

  2.3.1.5.5 That function will not return a value, of course
  2.3.1.5.6 The box trace of writeBackward("cat") is:


Initial call is makde, and function begins execution

Output line: **t**
Point A(**writeBackward(s)**) is reached, and recursive call is made.
New invocation begins execution

Output line: **ta**
Point A(**writeBackward(s)**) is reached, and recursive call is made.
New invocation begins execution

Output line: **tac**
Point A(**writeBackward(s)**) is reached, and recursive call is made.
New invocation begins execution

This is base case, so this invocation completes.
Control returns to calling box, which continues execution

Invocation completes. Control returns to calling box, which continues execution

Invocation completes. Control returns to calling box, which continues execution

Invocation completes. Control returns to statement following initial call

Figure 2-7, Box trace of writeBackward(“cat”)

2.3.2 Let's look at another solution which uses *strip away first character*
2.3.2.1 First look at a function that prints forward first, like so:

```
writeBackward1(in s:string)
    if (the string s is empty)
        Do nothing – this is the base case
    else
```

```
        {       Write the first character of s
                writeBackward1(s minus its first character)
        }
```

2.3.2.2 Prints forward

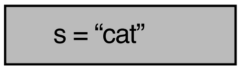
2.3.2.3 But change the order slightly and it will print backwards like so:

```
writeBackward2(in s:string)
        if (the string s is empty)
                Do nothing – this is the base case
        else
        {       writeBackward1(s minus its first character)
                Write the first character of s
        }
```

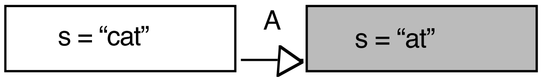
2.3.3 It writes backwards

- 2.3.3.1 Have “wound it up”
- 2.3.3.2 As it unwinds it will print the “first” character of every string it was passed – but first character had been stripped
- 2.3.3.3 This is the alternative to writing characters backwards, like so with the box trace:

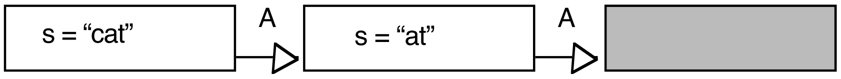
Initial call is makde, and function begins execution



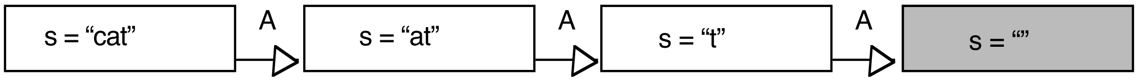
Point A is reached, and recursive call is made. New invocation begins execution:



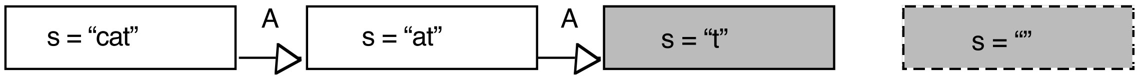
Point A is reached, and recursive call is made. New invocation begins execution:



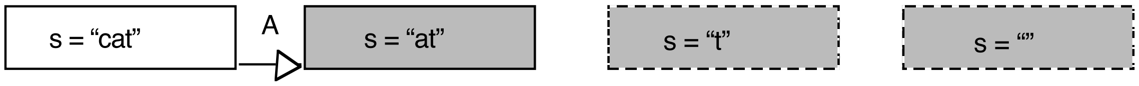
Point A is reached, and recursive call is made. New invocation begins execution:



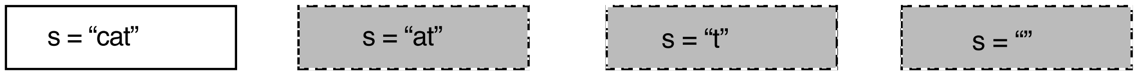
This invocation completes execution, and a return is made



This invocation completes execution, and a return is made



This invocation completes execution, and a return is made



This invocation completes execution, and a return is made

Figure 2-9, Box trace of writeBackward2(“cat”) in pseudocode