



COMPSCI 105 S1 2017 Principles of Computer Science

20-Recursion(1)



Agenda & Readings

- ▶ **Agenda**
 - ▶ What is recursion?
 - ▶ Recursive solutions, examples:
 - ▶ The Factorial of N
 - ▶ Box Trace Example
 - ▶ Write a String Backward
 - ▶ Tail Recursion
- ▶ **Reference:**
 - ▶ Textbook:
 - ▶ Problem Solving with Algorithms and Data Structures
 - Chapter 4 – Recursion



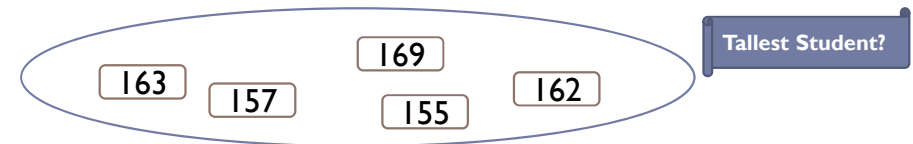
20.1 Introduction Definitions

- ▶ **Problem Domain:**
 - ▶ The space consisting of all elements for which the problem is solved
 - ▶ Examples: An array of integers, all people in this room, the days of the month, all “All Blacks” rugby games
- ▶ **Problem Size:**
 - ▶ The number of elements of the problem domain
 - ▶ Examples: An array with N elements, the number of people in this room, a list of N cities, the number of games played by the “All Blacks”
- ▶ **GOAL: Design algorithms to solve problems!**



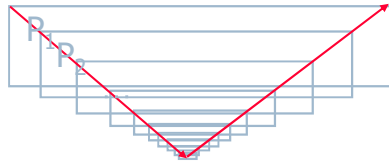
20.1 Introduction Iterative Algorithm

- ▶ **Algorithm which solves a problem by applying a function to each element of the problem domain**
 - ▶ Example: Find the tallest person in a group of N>0 students



```
def Student FindTallestStudent(Group_of_students)
  TallestStudent = Take any student from group;
  Repeat until nobody left
    Take next student from group
    If student is taller than TallestStudent then
      TallestStudent = student
  Return TallestStudent
```

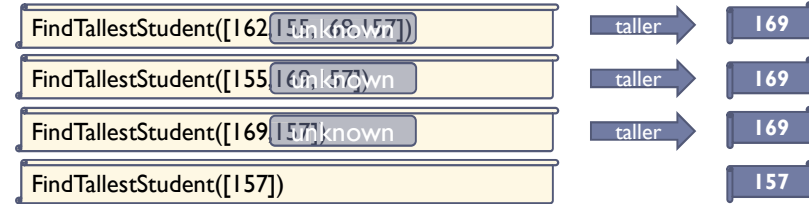
- Recursion is a powerful problem solving technique where a problem is broken into smaller and smaller identical versions of itself until a smaller version is small enough that it has an obvious solution



A **base case** is a special case whose solution is known

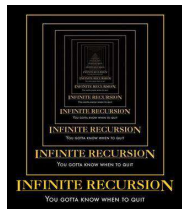
- Note:**
 - Complex problems can have simple recursive solutions It is an alternative to iteration (involves loops)
 - BUT:** Some recursion solutions are inefficient and impractical!

- Recursion involves a function calling itself
 - Example: Find the tallest person in a group of $N > 0$ students



```
def FindTallestStudent(Group of students)
    If only one student in group
        return this student
    else
        StudentA = Take any student from group
        StudentB = FindTallestStudent(Remaining Group)
        return the taller person of StudentA and studentB;
```

- Properties of a recursive solution
 - A recursive method calls itself
 - Each recursive call solves an identical, but smaller, problem
 - A test for the base case enables the recursive calls to stop
 - Base case: a known case in a recursive definition
 - Eventually, one of the smaller problems must be the base case (problem not allowed to become smaller than base case)



- Four questions for constructing recursive solutions
 - How can you define the problem in terms of a smaller problem of the same type?
 - How does each recursive call diminish the size of the problem?
 - What instance of the problem can serve as the base case?
 - As the problem size diminishes, will you reach this base case?



Example – Calculate the Sum

▶ Get the sum by:

- ▶ Taking the first number + the sum of the rest of the list

recursive_sum([2, 1, 5, 6]) 2+ 14

recursive_sum([1, 5, 6]) 1+ 12

recursive_sum([5, 6]) 5+ 11

recursive_sum([6]) 6+ 6

```
def recursive_sum(num_list):
    if len(num_list) == 0:
        return 0
    return num_list[0] + recursive_sum(num_list[1:])
```



Example – – Bad Recursion 1

▶ Problem:

- ▶ Compute the sum of all integers from 1 to n

```
def bad_sum(n):
    return n + bad_sum(n-1)
```

No base case!!!



Example – – Bad Recursion 2

▶ Problem:

- ▶ If n is odd compute the sum of all odd integers from 1 to n, if it is even compute sum of all even integers

```
def bad_sum(n):
    if (n == 0):
        return 0
    return n + bad_sum(n-2)
```

Base case cannot be reached!!!



Definition

▶ Problem

- ▶ Compute the factorial of an integer $n \geq 0$

▶ An iterative definition of factorial(n)

- ▶ If $n = 0$, factorial(0) = 1
- ▶ If $n > 0$, factorial(n) = $n * (n-1) * (n-2) * \dots * 1$

▶ Examples:

- ▶ $4! = 4 * 3 * 2 * 1 = 24$
- ▶ $7! = 7 * 6 * 5 * 4 * 3 * 2 * 1 = 5040$

```
def factorial(n):
    result = 1
    for i in range(n, 1, -1):
        result = result * i
    return result
```



Definition

- ▶ A recurrence relation
 - ▶ A mathematical formula that generates the terms in a sequence from previous terms
 - ▶ $\text{factorial}(n) = n * [(n-1) * (n-2) * \dots * 1]$
 - ▶ $\text{factorial}(n) = n * \text{factorial}(n-1)$
- ▶ A recursive definition of $\text{factorial}(n)$
 - ▶ $\text{factorial}(n) = \begin{cases} 1, & \text{if } n = 0 \\ n * \text{factorial}(n-1), & \text{if } n > 0 \end{cases}$

```
def fact (n):
  if n <= 0:
    return 1
  return n * fact(n-1)
```



Box Trace

- ▶ A systematic way to trace the actions of a recursive method
 - ▶ Create a new box for each recursive method call
 - ▶ Describe how return value is computed
 - ▶ Provide link to box (or boxes) for recursive method calls within the current method call
 - ▶ Each box corresponds to an activation record
 - ▶ Contains a method's local environment at the time of and as a result of the call to the method

The local environment contains:
Value of argument, local variables, return value,
address of calling method, ..., etc.



Four Criteria

- ▶ **fact(n)** satisfies the four criteria of a recursive solution
 - ▶ **fact(n)** calls itself
 - ▶ At each recursive call, the integer whose factorial to be computed is diminished by 1
 - ▶ The method handles the factorial 0 differently from all other factorials, where $\text{fact}(0)$ is 1
 - ▶ Thus the base case occurs when n is 0
 - ▶ Given that n is non-negative, item 2 of this assures that the computation will always reach the base case

```
def fact (n):
  if n <= 0:
    return 1
  return n * fact(n-1)
```

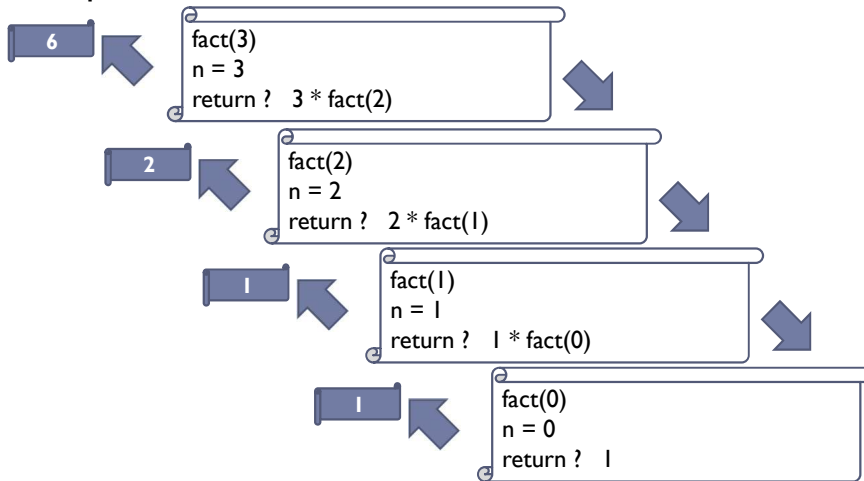


Box Trace

- ▶ A method's local environment includes:
 - ▶ The method's local variables
 - ▶ A copy of the actual value arguments
 - ▶ A return address in the calling routine
 - ▶ The value of the method itself

```
fact(3)
n = 3
A: fact(n-1) = ?
return ?
```

▶ Example



▶ Draw a call tree of the following method call: **fact(4)**

▶ Problem:

- ▶ Given a string of characters, write it in reverse order

▶ Recursive solution:

- ▶ Each recursive step of the solution diminishes by 1 the length of the string to be written backward
- ▶ Base case:
 - ▶ Write the empty string backward

▶ Examples:

```

print(writeBackward("cat")) tac
print(writeBackward("cat")) tac
    
```

▶ Two approaches

▶ writeBackward(s)

call method recursively for the string minus the **last** character

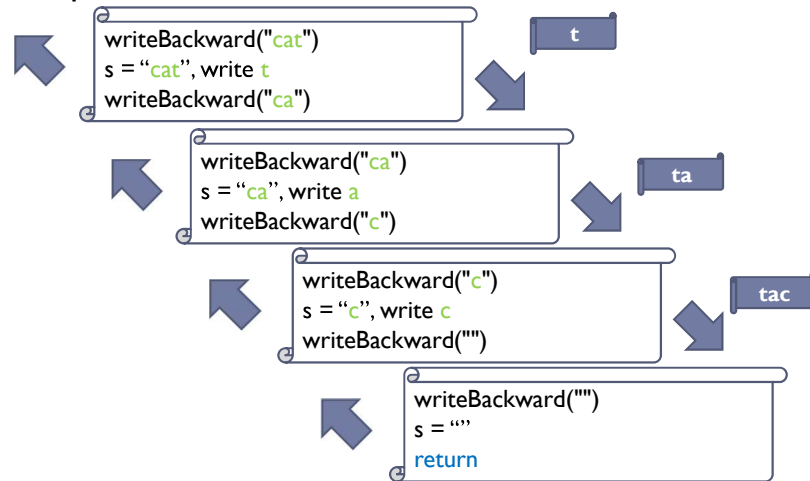
if the string s is empty:
Do nothing – base case
else:
write the last char of s
writeBackward(s minus its **last** char)

▶ writeBackward2(s)

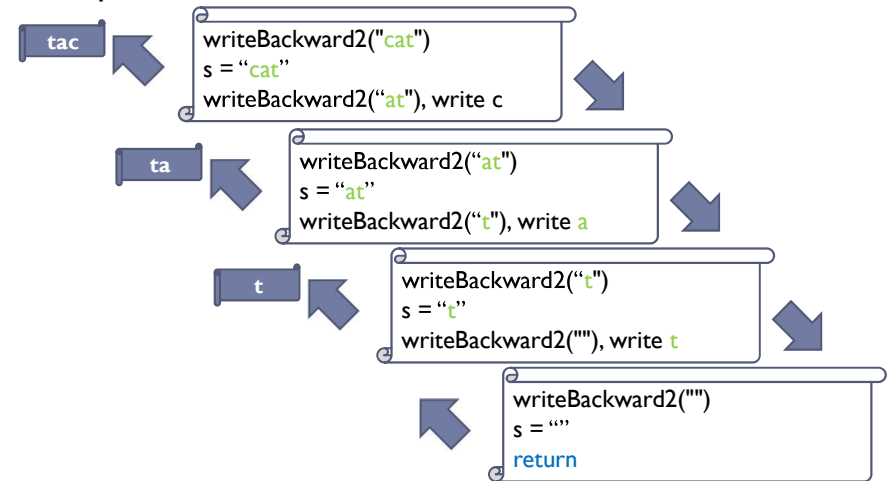
call method recursively for the string minus the **first** character

if the string s is empty:
Do nothing – base case
else:
writeBackward2(s minus its first char)
write the first char of s

▶ Example



▶ Example



Summary

- ▶ A recursive algorithm passes the buck repeatedly to the same function
- ▶ Recursive algorithms are well-suited for solving problems in domains that exhibit recursive patterns
- ▶ Recursive strategies can be used to simplify complex solutions to difficult problems