CISC 3115 TY3

C22a: Problem Solving using Recursion

Hui Chen

Department of Computer & Information Science

CUNY Brooklyn College

Outline

- Characteristics of recursion
- Recursion as problem solving strategy
  - Examples
- Recursive helper method/function
- Tail and non-tail recursion

Characteristics of Recursion

- All recursive methods have the following characteristics:
  - One or more base cases (the simplest case) are used to stop recursion.
  - Every recursive call reduces the original problem, bringing it increasingly closer to a base case until it becomes that case.

Recursion as Problem Solving Strategy

- Break the problem into subproblems such that one or more subproblems resembles the original problem
  - These subproblems resembling the original problem is almost the same as the original problem in nature with a smaller size.
- Apply the same approach to solve the subproblem recursively to reach the base case

Problem Solving Example: Print Message Many Times

- Problem: print a message n times.
- Can we solve it using recursion?
- Original problem: print a message n times
- Subproblems with smaller size
  - print a message 1 time
  - print a message (n-1) times (the same problem as the original problem, but smaller size)
- Base case: print the message 0 times (or 1 time)

Problem Solving Example: Print Message Many Times: Solution

- Base case: n = 0 (how about the n = 1 base case? Which one yields better code?)

```
public class PrintMsg {
  public static void main(String[] args) {
    nPrintMsg("Hello, World!", 5);
  }
  public static void nPrintMsg(String msg, int n) {
    if (n == 0) return;   // base case
    System.out.println(msg); // subproblem 1
    nPrintMsg(msg, n-1);     // subproblem 2
  }
}
```

Problem Solving Example: Print Message Many Times: Revisit

- Wait! Can we just write a loop to print a message n times? (solving problem iteratively)
- Remarks
    - However, it is sometimes easier to think recursively to solve a complex problems.
    - Many problems we solve iteratively can also be solved recursively.
        - Example: the palindrome problem (e.g., madam, nursesrun)

Problem Solving Example: Is It a Palindrome?

- Problem: is a given string a palindrome?
- Recursive solution:
    - 1) Compare the first and last character of the string. If not equal, not palindrome; 2) otherwise, repeat for the substring less the first and the last character (the same problem whose size is the original size – 2)
    - Base case: a single character or empty string, and the single character string or the empty string is always a palindrome.

Problem Solving Example: Is It a Palindrome? Solution

- An example realization of the solution

```
public static boolean isPalindrome(String s) {
  // base case
  if (s.length() <= 1) return true;

  // subproblem 1
  if (s.charAt(0) != s.charAt(s.length()-1)) return false;

  // subproblem 2
  return isPalindrome(s.substring(1, s.length()-1));
}
```

Problem Solving Example: Is It a Palindrome? Discussion

- Is the solution efficient, in particular, when the string is very long? Hint: a Java string is immutable? How many string objects are being coreated?

```
public static boolean isPalindrome(String s) {
  // base case
  if (s.length() <= 1) return true;

  // subproblem 1
  if (s.charAt(0) != s.charAt(s.length()-1)) return false;

  // subproblem 2
  return isPalindrome(s.substring(1, s.length()-1));
}
```

Problem Solving Example: Is It a Palindrome? Recursive Helper

- Rewrite it by introducing a new method that uses parameters to indicate subproblem size

```
public static boolean isPalindrome(String s) {
    return isPalindrome(s, 0, s.length()-1);
}


// The recursive helper method
public static boolean isPalindrome(String s, int beginIndex, int endIndex) {
  if (endIndex - beginIndex <= 1) return true;    // base case
  if (s.charAt(beginIndex) != s.charAt(endIndex)) return false; // subproblem 1
  return isPalindrome(s, beginIndex+1, endIndex-1); // subproblem 2
}
```

Problem Solving Example: Selection Sort

- Problem: sort a list
- Recursive solution: divide the problem into two subproblems
  - 1. Find the smallest number in the list and swaps it with the first number.
  - 2. Ignore the first number and sort the remaining smaller list recursively (subproblem is the same problem as the original problem with the size – 1).

Problem Solving Example: Selection Sort: Solution

- The sample solution includes two realizations
  - Sort integers
  - Sort any objects with the Comparator interface

Problem Solving Example: Searching

- Problem: search an item (using its key) in a sorted list
- Recursive solution: divide the problem into subproblems, one or more are essentially the original problem
    - 1. Find the middle element in the list
    - 2. The list becomes three parts. Determine which part contains or may contain the item. Search the item the part that may contain the item (the subproblem identical to the original problem but with smaller size)
        - Case 1: If the key is less than the middle element, recursively search the key in the first half of the list.
        - Case 2: If the key is equal to the middle element, the search ends with a match.
        - Case 3: If the key is greater than the middle element, recursively search the key in the second half of the array.
    - Base case: the list becomes empty (not found); or it is the middle element (found).

Problem Solving Example: Searching: Implementation

- An example implementation using a helper method

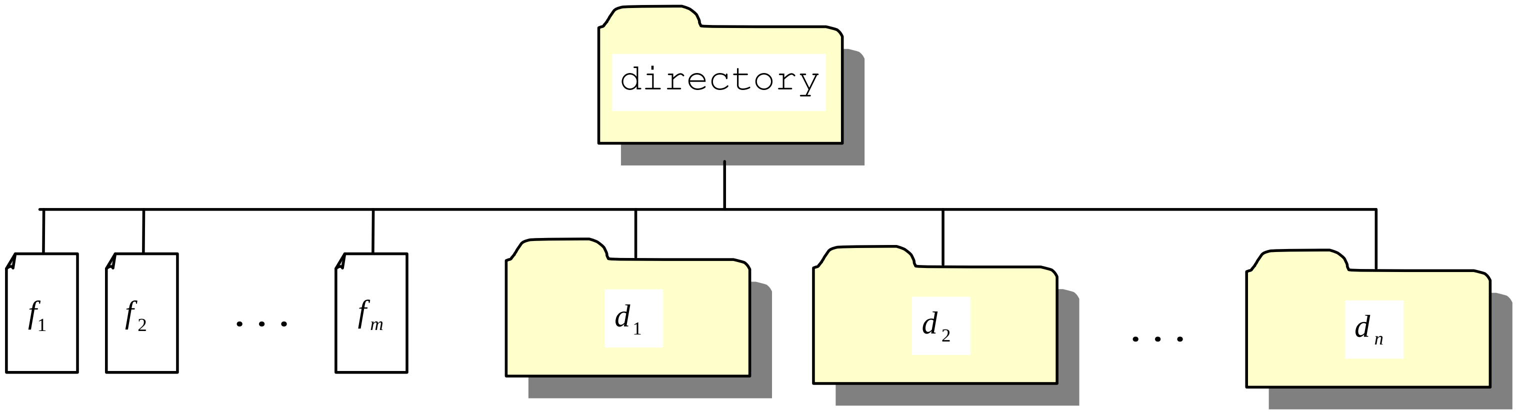
```
public static int search(int[] numbers, int key) {     return search(numbers, key, 0, numbers.length-1);     }


private static int search(int[] numbers, int key, int beginIndex, int endIndex) {
  int mid = (endIndex + beginIndex) / 2; // observe when mistakenly wrote - instead
  if (beginIndex > endIndex) return - beginIndex - 1; // base case (not foudn)
  if (numbers[mid] == key)  return mid;                // base case (found)
  if (key < numbers[mid]) { // subproblem, the same problem but smaller size
    return search(numbers, key, beginIndex, mid-1);
  } else { // subproblem, the same problem but smaller size
    return search(numbers, key, mid+1, endIndex);
  }
}
```

Iteration or Recursion?

- Some problems appear to be easily solved using iteration, while others recursion.
- Question: can you solve preceding examples using iteration?
- Example problems (recursion is easier)
  - Search files containing a word in a directory (the search file problem, already discussed)
  - Find directory size (the total size in bytes of all files under a directory, a revision of the search file problem)
  - Solve the "Tower of Hanoi" problem
  - Quick sort

Problem Solving Example: Directory Size

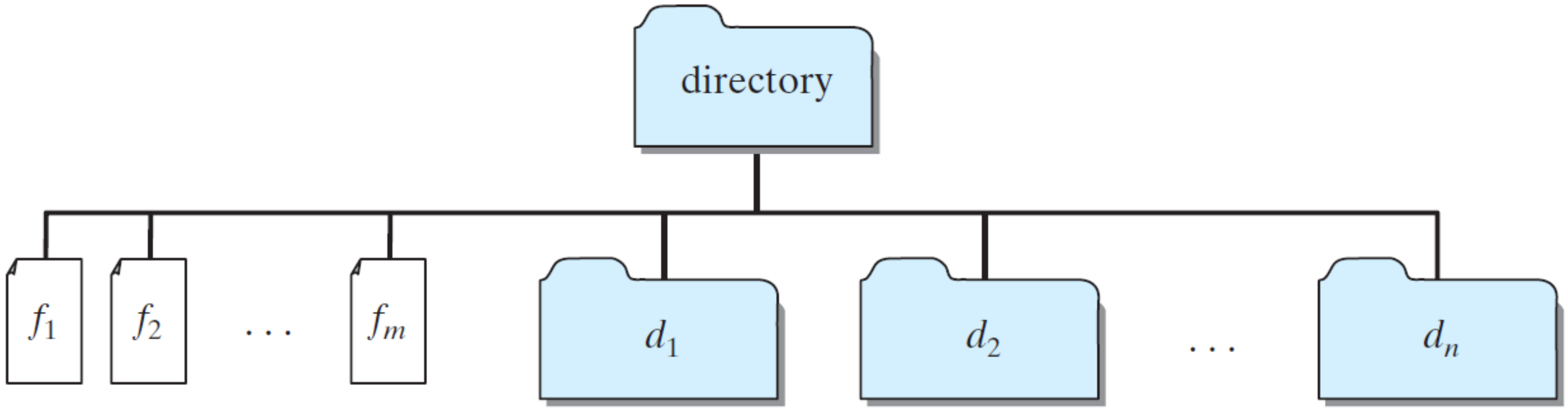
- Problem: to find the size of a directory, i.e., the sum of the sizes of all files in the directory.
- The challenge: a directory may contain subdirectories and files.



Directory Size: Thinking Recursively

- The size of the directory can be defined recursively as follows,

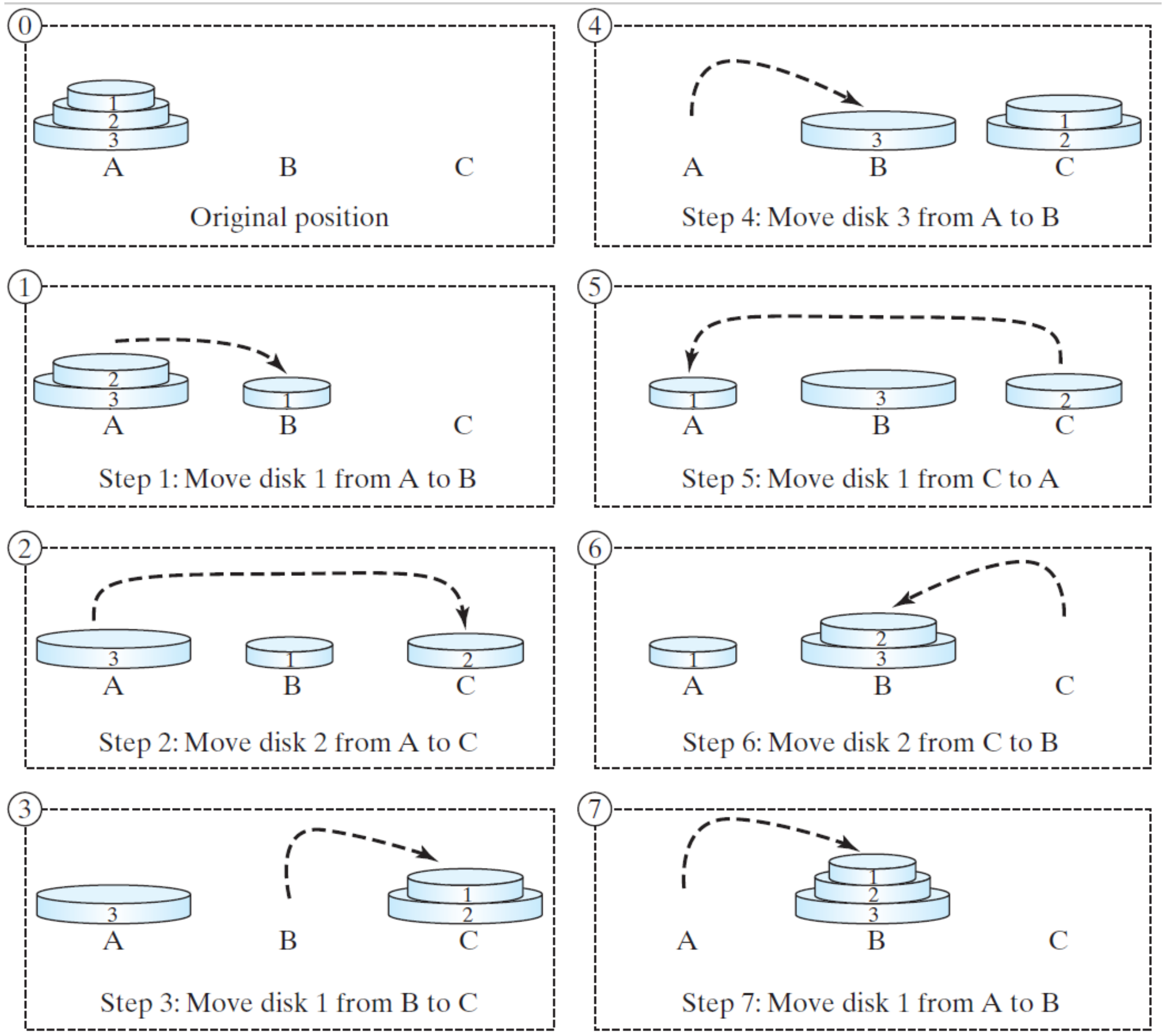
$$size(d) = size(f_1) + size(f_2) + \ldots + size(f_m) + size(d_1) + size(d_2) + \ldots + size(d_n)$$



Problem Solving Example: Tower of Hanoi

- Problem:
    - There are n disks labeled 1, 2, 3, . . ., n, and three towers labeled A, B, and C.
    - All the disks are initially placed on tower A.
    - No disk can be on top of a smaller disk at any time.
    - Only one disk can be moved at a time, and it must be the top disk on the tower.
- See https://liveexample.pearsoncmg.com/dsanimation/TowerOfHanoi.html

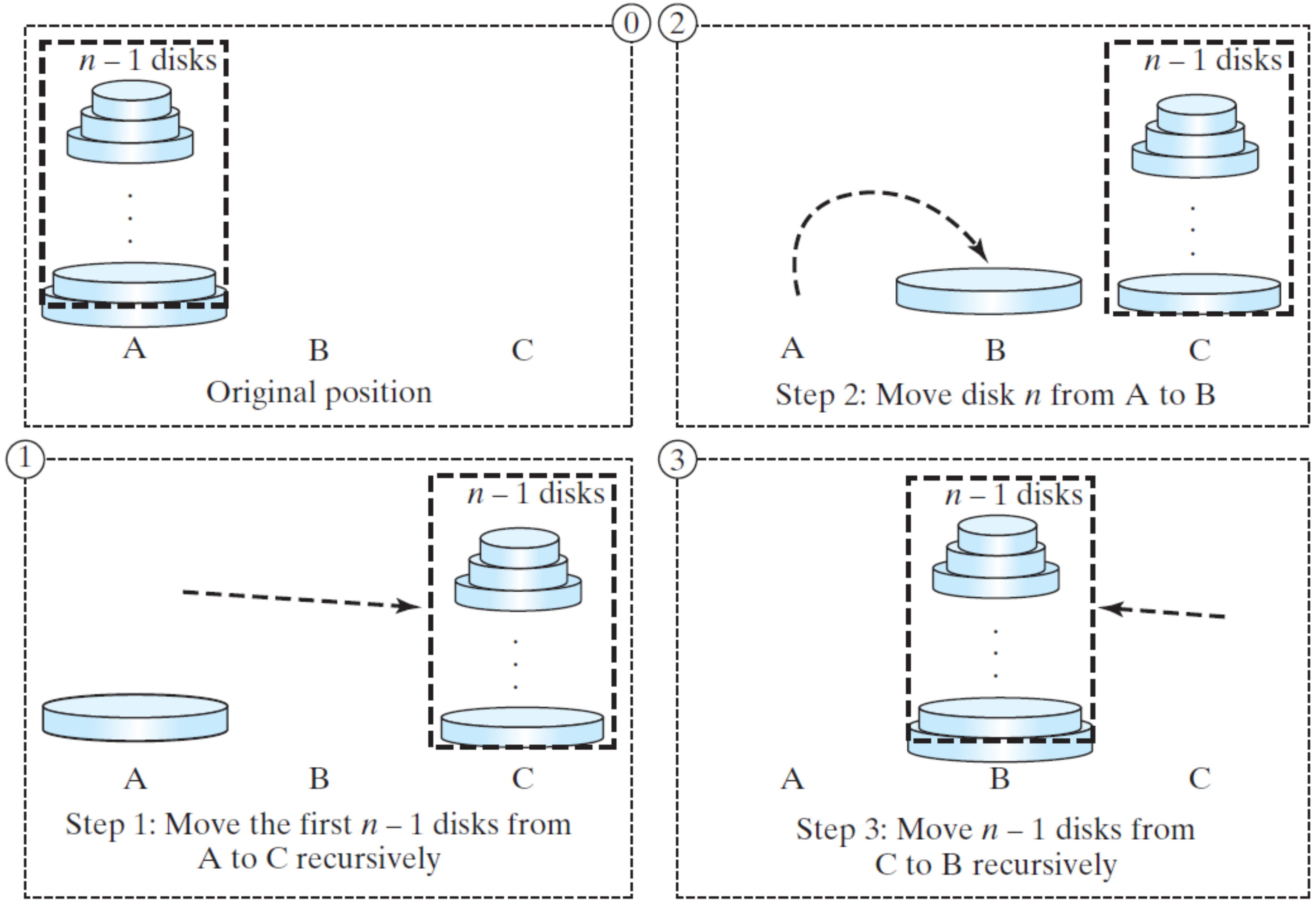
Examine it at Size = 3


0
A
B
C
Original position
1
A
B
C
Step 1: Move disk 1 from A to B
2
A
B
C
Step 2: Move disk 2 from A to C
3
A
B
C
Step 3: Move disk 1 from B to C
4
A
B
C
Step 4: Move disk 3 from A to B
5
A
B
C
Step 5: Move disk 1 from C to A
6
A
B
C
Step 6: Move disk 2 from C to B
7
A
B
C
Step 7: Move disk 1 from A to B

How about Large Size?

- Starting with n disks on tower A. The Tower of Hanoi problem can be decomposed into three subproblems:
  - Move n-1 disks from tower A to tower C
  - Move disk n from tower A to tower B
  - Move n-1 disks from tower C to tower B

How about Large Size?



0
n – 1 disks
A
B
C
Original position
2
n – 1 disks
A
B
C
Step 2: Move disk n from A to B
1
n – 1 disks
A
B
C
Step 1: Move the first n – 1 disks from A to C recursively
3
n – 1 disks
A
B
C
Step 3: Move n – 1 disks from C to B recursively

Questions?

- Problem solving using recursion
  - Divide big problem into smaller subproblems some of which are the same problem as the original one with smaller size
- Examples
  - Sorting, searching, and others
- More examples in the textbook

Tail Recursion

- A recursive method is said to be *tail recursive* if there are no pending operations to be performed on return from a recursive call.
- Tail recursions can be realized by complier efficiently.

Tail and Non-tail Recursion: Compute Factorial

Non-tail recursion

```
public static int factorial(int n) {

   if (n == 0) {  // base case

     return 1;

   } else { // recursive call or method invocation

     // non-tail recursion, because we have to multiple
factorial(n-1) by n, a pending operation

     return n * factorial(n - 1);

   }

 }
```

Tail recursion

```
public static int factorial(int n) {

   return factorial(n, 1);

 }

 private static int factorial(int n, int result) {

   if (n == 0) { // base case

     return result;

   } else { // recursive call

     // tail recursion, no pending operation after
returning from the recursive call

     return factorial(n - 1, n * result);

   }

 }
```

Questions

- Concept of tail and non-tail recursions
- Can you identify non-tail/tail-recursive methods in preceding examples?
- Write tail-recursive methods