Program Design

Lecture 5

Recursive function: Syntax

- function_name(parameter list)
- {
- …
- //*‘c’ statements*
- …
- function_name(parameter values) //recursive call
- …
- }

Fibonacci series

- A series of numbers in which each number (*Fibonacci number*) is the sum of the two preceding numbers.

- Eg of the series 1, 1, 2, 3, 5, 8,..

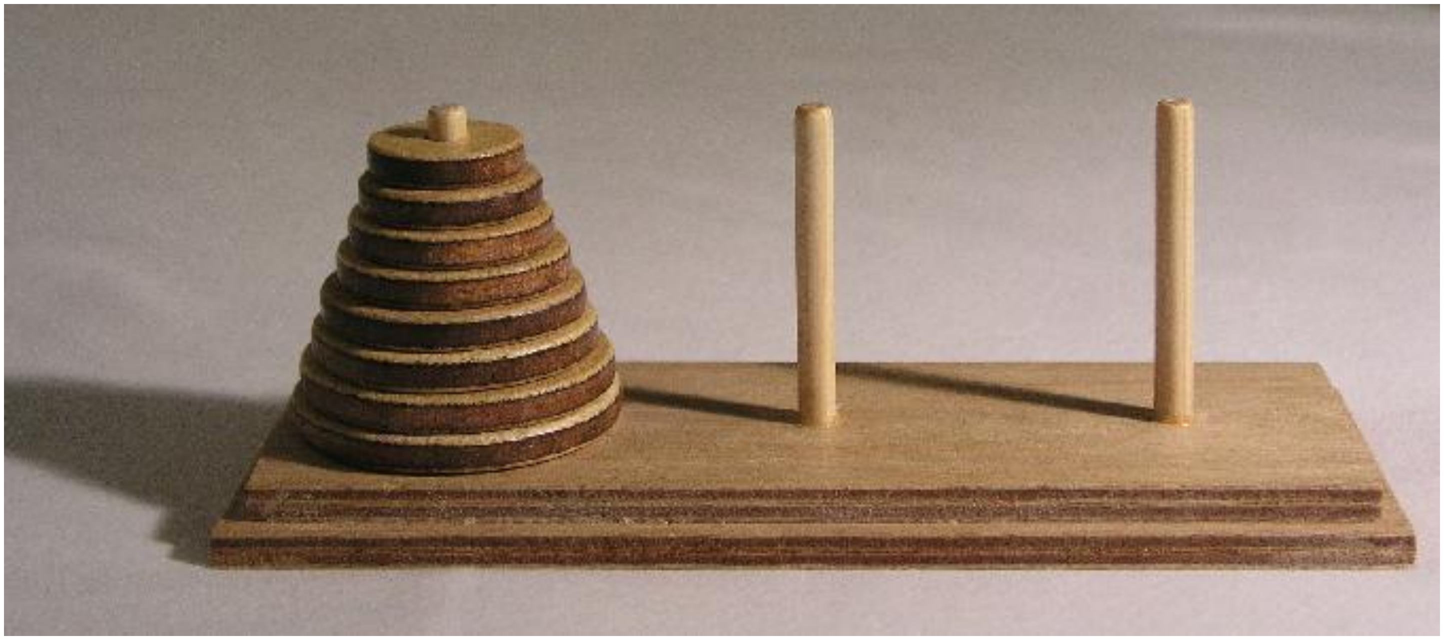
Fibonacci series

```
int fibonacci(int i) {
        if(i == 0) { return 0; }
        if(i == 1) { return 1; }
        return fibonacci(i-1) + fibonacci(i-2); }
int main() {
        int i; for (i = 0; i < 10; i++)
         { printf("%d\t\n", fibonacci(i)); } return 0; }
```

Exercise: Predict the output

```
int count = 1;
 void recurse(int sum) {
	sum  = sum + count;
	count ++;
	if(count<=9)	{
	recurse(sum);	}
	else	{
	printf("\nSum is [%d] \n", sum);	}	return; }
int main(void) {
	 int sum = 0;
	recurse(sum);
	return 0; }
```

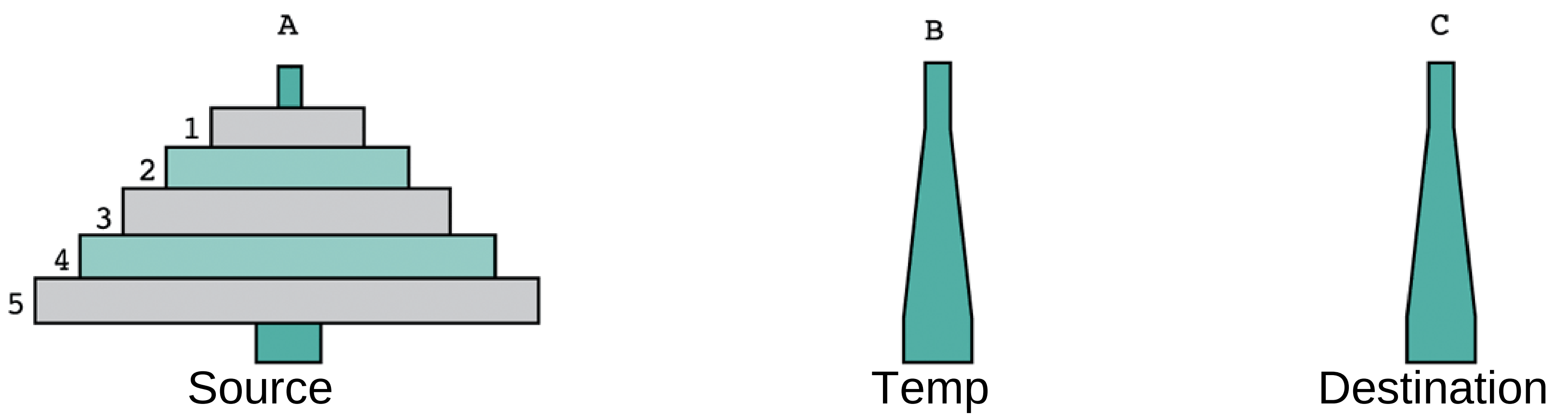
More Interesting Example
Towers of Hanoi



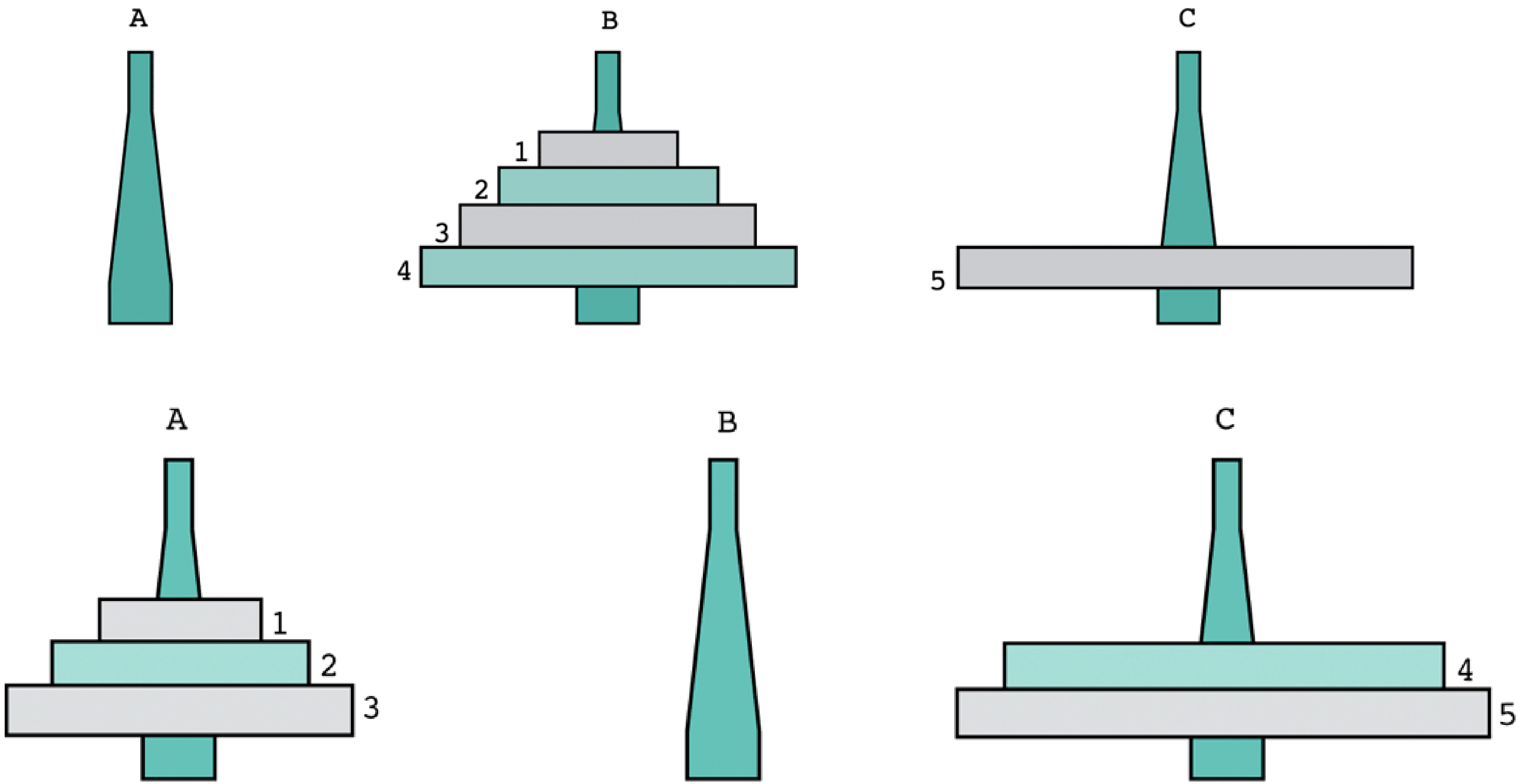
- Move stack of disks from one peg to another
- Move one disk at a time
- Larger disk may never be on top of smaller disk

A Classical Case: Towers of Hanoi

- The towers of Hanoi problem involves moving a number of disks (in different sizes) from one tower (or called “peg”) to another.
  - The constraint is that the larger disk can never be placed on top of a smaller disk.
  - Only one disk can be moved at each time
  - Assume there are three towers available.



A Classical Case: Towers of Hanoi



A Classical Case: Towers of Hanoi

- This problem can be solved easily by recursion.
- Algorithm:

    if n is 1 then

        move disk 1 from the source tower to the destination tower

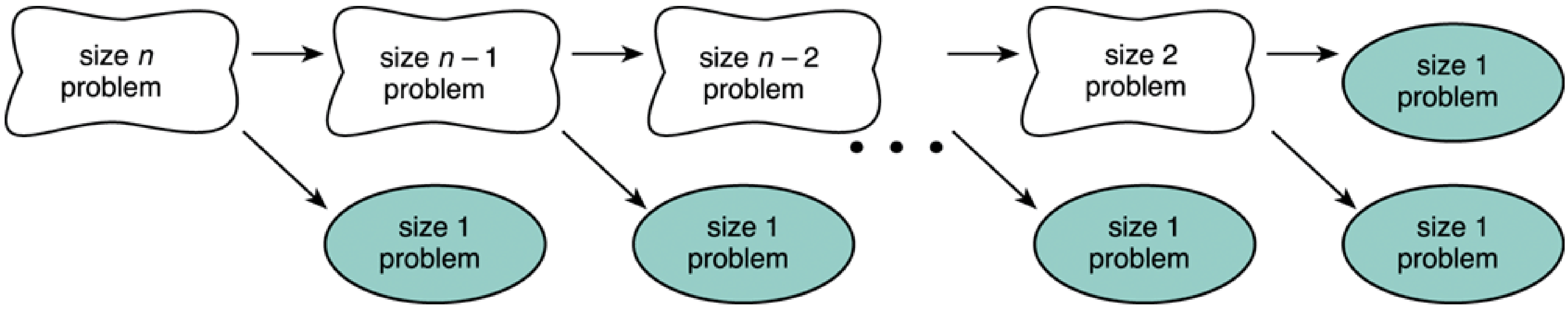
    else

        1. move n-1 disks from the source tower to the temp tower.
        2. move disk n from the source tower to the destination tower.
        3. move n-1 disks from the temp tower to the destination tower.

Problems Suitable for Recursive Functions

- One or more simple cases of the problem have a straightforward solution.
- The other cases can be redefined in terms of problems that are closer to the simple cases.
- The problem can be reduced entirely to simple cases by calling the recursive function.
  – *If this is a simple case*
    solve it
    else
    redefine the problem using recursion

Splitting a Problem into Smaller Problems



- Assume that the problem of size 1 can be solved easily (i.e., the simple case).
- We can recursively split the problem into a problem of size 1 and another problem of size n-1.

Recursion vs. iteration

- Iteration can be used in place of recursion
  - An iterative algorithm uses a *looping construct*
  - A recursive algorithm uses a *branching structure*
- Recursive solutions are often less efficient, in terms of both *time* and *space*, than iterative solutions
- Recursion can simplify the solution of a problem, often resulting in *shorter*, more easily understood source code

Recursion *vs.* Iteration (Contd…)

- Some simple recursive problems can be “unwound” into loops
  - But code becomes less compact, harder to follow!
- Hard problems cannot easily be expressed in non-recursive code
  - Tower of Hanoi
  - Robots or avatars that “learn”
  - Advanced games

Introduction to Algorithms

Computational Problem

- Statement of the problem specifies the desired input/output relationship

  Eg: Find the factorial of a given number.

- Algorithm describes a specific computational procedure to achieve the input/output relationship

  Eg: Steps to find the factorial of the input number

Examples of Problems solved by Algorithms:

- Human Genome Project
- Internet - Search Engine
- Electronic commerce
- Resource allocation
- Shortest Path problem
- Longest common subsequence

Ref: CLRS book (Chapter 1)

Types of Computational Problems:

Decision Problems: Answer for every instance is either YES or NO

Search Problems: Searching for a given value in the list of values

Optimization Problems: Find a "best possible" solution among the set of all possible solutions

Counting Problems: Number of solutions to a search problem

Classify the following problems:

1) Find a path between two nodes in a graph

2) Find the maximum value in the list of values

3)Checking whether the given number is in the list or not

4)Find the shortest path between two nodes in the given graph

5)Find the number of non-trivial prime factors of the number n

6)Check whether the number X is an Armstrong number or not

Computational Problem

- Problem specifies the desired input / output relationship

  Eg: Find the Prime factors of a given number.

Formal definition of a **Searching Problem**

Input: A sequence of n numbers $A = \langle a_1, a_2, a_3, a_4, \ldots, a_n \rangle$

and a value v

Output: An index i such that $v = A[i]$ or the

special value NIL if v does not appear in A.

Eg: What is the Input to Searching Problem?

<10,29,65,23,12,15,78>, 23

Output ?

4

Input is referred as **instance of the Searching** Problem

Instance consists of the **input** needed to compute the solution to the problem.

Instance: Satisfies the constraints imposed in the problem definition

Eg: Find the factorial of a number.

Sort the given input.

What are the reasonable constraints that can be considered?

Formal definition of a Sorting Problem

Sorting - Fundamental Operation in Computer Science

Input : A sequence of n numbers $<a_1,a_2,a_3,a_4,...,a_n>$

Output: A permutation (reordering) $<a_1',a_2',a_3',a_4',...,a_n'>$ of the input sequence such that $a_1' <= a_2' <= a_3' <= a_4' <= \ldots <= a_n'$

Eg:

(Input Sequence) **Instance**: <31, 41, 59, 26, 41, 58>

Output Sequence: <26, 31, 41, 41, 58, 59>

Algorithms

- Well defined computational procedure that takes input (some value or set of values) and produces output (some value or set of values)

- Sequence of well-defined computational steps that

  transform the input to the output

- **Tool** for solving a well-specified computational problem

Correct & Incorrect Algorithms

- **Correct** - for every input instance, it halts with the correct output
- Correct algorithm solves the given computational problem
- Incorrect algorithm - might not halt at all on some input instances or it might halt with an incorrect answer
- **Our focus is on correct algorithms**

Reading Assignment: CLRS 3rd edition- Chapter 1

Give Examples for the following:

Correct algorithm

An algorithm that does not halt on some input instances

An algorithm that gives an incorrect answer

An **algorithm** can be specified,

- English
- Computer program
- Hardware design

ONLY requirement:

Precise description of the computational procedure

Representation - Pseudocode

- **Pseudocode:** An expressive way of making things clear using English sentences

- **Pseudocode** - Not concerned with issues of software engineering such as modularity, error handling etc.

Pseudocode of Linear Search

```
LINEAR SEARCH(A, key)     // Pseudocode of Linear Search
1. found = 0
2. for i = 1 to A.length
3.         if A[ i ] = key
4.             found = 1
5.             return i
6. if found = 0
7.     return 0
```

Pseudocode conventions

- **Indentation** indicates block structure – Eg: loops and conditional constructs

<u>Eg1:</u>

```
while i > 0 and A[ i ] >key
        A[ i+1 ]=A[ i]
        i=i-1
```

<u>Eg2</u>

```
if A[ i ] =key
          found = 1
          return i
```

// This symbol indicates that the remainder of the line is a comment

2. **for** i = 1 **to** A.length
3. if A[i] = key
4. found = 1
5. return i

- Keyword ***to*** is used when a ***for*** loop increments its value by 1 in each iteration
- Keyword ***downto*** is used when a ***for*** loop decrements its value of loop counter by 1
- If the loop counter changes by an amount **greater than 1**, the amount of change follows the optional keyword ***by***

- ***i=j=e*** is equivalent to ***j=e*** followed by ***i=j***
- Variables are local to a given procedure
- We shall not use global variables without explicit indication
- A[i] indicates the i^{th} element of A
- A[i…j] indicates the element of sub array of A with elements A[i], A[i+1],.., A[j]
- We access a particular attribute of an object by an object name followed by a dot followed by the attribute name

- Parameters are passed to a procedure by **value**
- **Called procedure** receives its own copy of the parameters and, if it assigns a value to a parameter, the change is not seen by the calling procedure
- A **return** statement immediately transfers control back to the point of call in the calling procedure.
- Most **return** statements also take a value to pass back to the caller.
- Pseudocode - allows multiple values to be returned in a single **return** statement.

- The boolean operators “and” and “or” are **short circuiting.**

- When we evaluate the expression “x and y”,
  - Possible cases

- While evaluating the expression “x or y”
  - Possible cases

Binary Search

- Input : A sequence of n numbers $A = \langle a_1, a_2, \ldots, a_n \rangle$ such that $a_1 \le a_2 \le \ldots \le a_n$ and a key k
- Binary Search
  - Compare k with the middle element of the sequence, say A[mid]
  - If k = A[mid] return mid
  - If k < A [mid] search in the first half, (A[1..mid-1])
  - If k > A [mid] search in the second half (A[mid+1..n])

Example

A: 3 5 7 9 11 12 35 40 48 52 65 , k : 48

A[mid] is 12 , k > 12, search A [7 .. 11]

A: 3   5   7   9   11 12 35   40   48   52   65

A[mid]=48   matches k

Search finished with just 2 comparisons

k : 65 ? k:3 ?   k:6?

Binary Search

- Binary Search
  - `If k < A [mid] search in the first half (A[1..mid-1])`
  - `If k > A [mid] search in the second half (A[mid+1..n])`
- Size of the sequence to be searched is reduced to half
- `n -> n/2 -> n/4 -> ... -> 1`

Binary Search – Algorithm

```
BinarySearch (A, m, n, k)
   if (m>n) ……………….????  //Base Case
   mid=(m+n)/2
   if A[mid]=k  return mid;//Base Case
   else if k < A[mid]
      BinarySearch(A, m, mid-1, k)
   else if k > A[mid]
      BinarySearch(A, mid+1, n, k)
```

Binary Search – Algorithm

```
BinarySearch (A, m, n, k)
   if (m>n) return -1;    //Base Case
   mid=(m+n)/2
   if A[mid]=k  return mid;     //Base Case
   else if k < A[mid]
     BinarySearch(A, m, mid-1, k)
   else if k > A[mid]
     BinarySearch(A, mid+1, n, k)
```

Thank You