AP Computer Science B

Java Object-Oriented Programming [Ver. 2.0]

Unit 5: Algorithm Study

WEEK 14: CHAPTER 17 RECURSION (PART 2: RECURSIVE PROCESSING)

DR. ERIC CHOU

IEEE SENIOR MEMBER

Objectives

- Problem Solving using Recursion
- Recursive Processing I: Statistics
- Recursive Processing II: Text Processing
- Recursive Processing III: Algorithms
- Recursion versus Iteration
- Tower of Hanoi

Problem Solving using Recursion

LECTURE 1

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.**
- In general, to solve a problem using recursion, you break it into subproblems. If a subproblem resembles the original problem, you can apply the same approach to solve the subproblem recursively. This subproblem is almost the same as the original problem in nature with a smaller size.

Problem Solving Using Recursion

Let us consider a simple problem of printing a message for n times. You can break the problem into two subproblems: one is to print the message one time and the other is to print the message for n-1 times. The second problem is the same as the original problem with a smaller size. The base case for the problem is n==0. You can solve this problem using recursion as follows:

```
public static void nPrintln(String
message, int times) {
  if (times >= 1) {
    System.out.println(message);
    nPrintln(message, times - 1);
  } // The base case is times == 0
}
```

Think Recursively

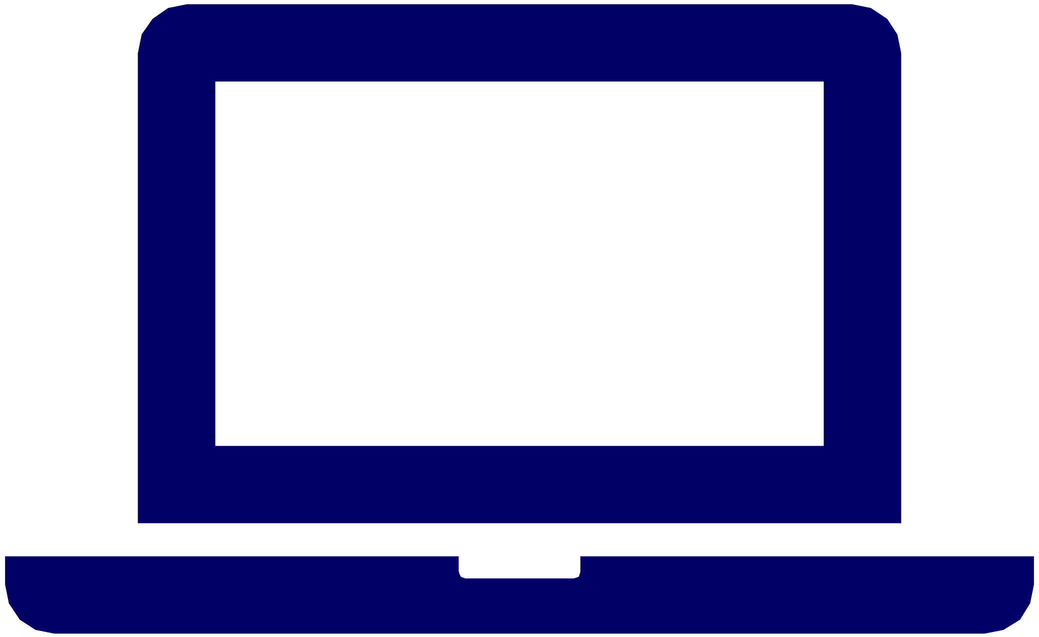
Many of the problems presented in the early chapters can be solved using recursion if you *think recursively*. For example, the palindrome problem can be solved recursively as follows:

```
public static boolean isPalindrome(String s) {
  if (s.length() <= 1) // Base case
    return true;
  else if (s.charAt(0) != s.charAt(s.length() - 1)) // Base case
    return false;
  else
    return isPalindrome(s.substring(1, s.length() - 1));
}
```

Recursive Helper Methods

- The preceding recursive **isPalindrome** method is not efficient, because it creates a new string for every recursive call. To avoid creating new strings, use a helper method:

```
public static boolean isPalindrome(String s) {
  return isPalindrome(s, 0, s.length() - 1);
}
public static boolean isPalindrome(String s, int low, int high) {
  if (high <= low) // Base case
    return true;
  else if (s.charAt(low) != s.charAt(high)) // Base case
    return false;
  else
    return isPalindrome(s, low + 1, high - 1);
}
```



Demonstration Program

PALINDROME.JAVA

PALINDROMETEST.JAVA

Recursive Processing I

LECTURE 1

Algorithms in Recursive Processing I

1) Recursive Traversal
2) Maximum
3) Minimum
4) Sum
5) Average
6) Example Array:

```
static int[] a1 = {5, 3, 6, 7, 9, 24, 27, 1, 0 , 16};
```

(1) Recursive Traversal

```
public static void traverse(int[] a){
     traverse(a, 0);
}

public static void traverse(int[] a, int current){
     if (current == a.length) {
        System.out.println();
        return;
     }
     System.out.print("("+a[current]+") ");
     traverse(a, current+1);
}
```

(2) Maximum

```
public static int max(int[] a){
      return max(a, 0, Integer.MIN_VALUE);
 }

 public static int max(int[] a, int current, int result){
      if (current == a.length) return result;
      if (a[current] > result) result = a[current];
      return max(a, current+1, result);
 }
```

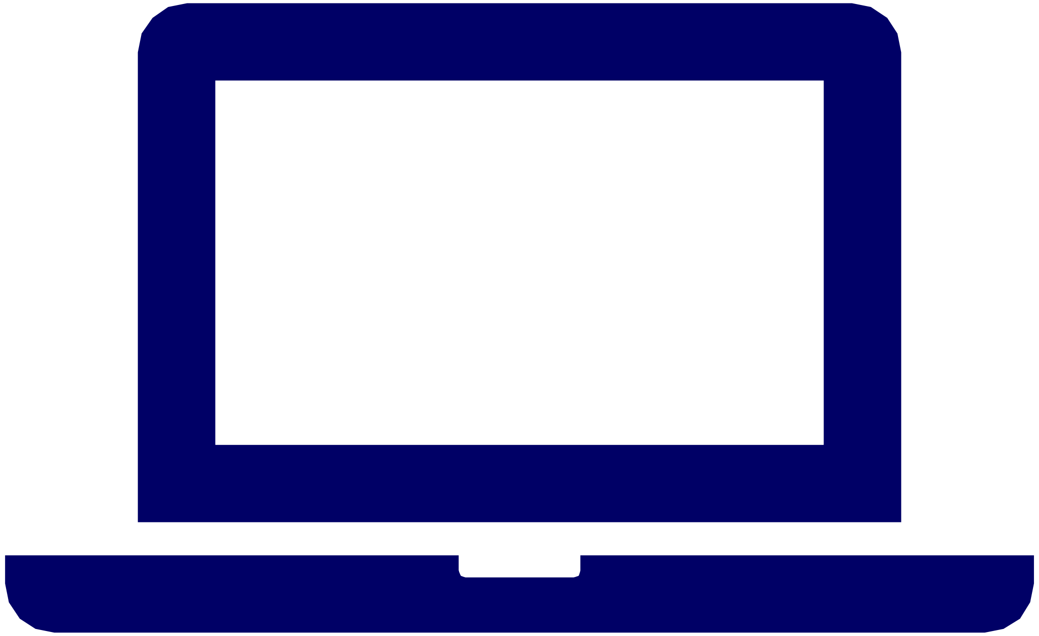
(3) Minimum

```
public static int min(int[] a){
    return min(a, 0, Integer.MAX_VALUE);
}

public static int min(int[] a, int current, int result){
     if (current == a.length) return result;
     if (a[current] < result) result = a[current];
     return min(a, current+1, result);
}
```

Sum and Average

```
public static int sum(int[] a){
   return sum(a, 0, 0);
}
public static int sum(int[] a, int current, int result){
     if (current == a.length) return result;
     result += a[current];
     return sum(a, current+1, result);
}
public static double avg(int[] a){
    return (double) sum(a)/a.length;
}
```



Demonstration Program

RECURSIVEPROCESSINGI_STATS.JAVA

Recursive Processing II

LECTURE 1

Algorithms in Recursive Processing II

1) Palindrome Check (already discussed)
2) Reverse of String
3) Permutation of String

Reverse of String

```
public static String reverse(String a){
     if (a.length()==1) return a;
     return a.charAt(a.length()-1) +
                 reverse(a.substring(0, a.length()-1));
   }
```

Permutation of String

```
public static void permute(String a, ArrayList<String> alist){
      permute(a, "", alist);
   }
  public static String cut(String a, int i){
      return a.substring(0, i)+a.substring(i+1);
   }
  public static void permute(String a, String result, ArrayList<String> alist){
        if (a.length() == 1) { alist.add(a.charAt(0)+ result); return; }
        for (int i=0; i<a.length(); i++)
              permute(cut(a, i), a.charAt(i)+result, alist);
   }
```

Demonstration Program

RECURSIVEPROCESSINGII_TEXT.JAVA

Recursive Processing III

LECTURE 1

Algorithms in Recursive Processing III

(1) Linear Search

(2) Recursive Selection Sort (Linear Traversal Version calling minIndex())

(3) Recursive Binary Search

(4) Recursive Selection Sort

(5) Directory Size (Finding files and directory recursively.)

Recursive Linear Search

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

public static int search(int[] a, int key, int current, int result){
     if (current == a.length) return result;
     if (a[current] == key) return current;
     return search(a, key, current+1, result);
 }
```

Selection Sort with minIndex()

minIndex(): finding the element with minimum value

```
public static int minIndex(int[] a){
     return minIndex(a, 0, Integer.MAX_VALUE, -1);
 }

public static int minIndex(int[] a, int current, int result, int index){
      if (current == a.length) return index;
      if (a[current] < result) {
          result = a[current];
          index = current;
      }
      return minIndex(a, current+1, result, index);
 }
```

Selection Sort with minIndex()

sort(): using the concept of available list (MAX_VALUE unavailable elements)

```
public static void sort(int[] a){
      int[] c = new int[a.length]; int[] b = new int[a.length];
      for (int i=0; i<a.length; i++) c[i] = a[i];
      sort(c, b, 0);
      for (int i=0; i<a.length; i++) a[i] = b[i];
  }
public static void sort(int[] c, int[] b, int current){
      if(current == c.length) return;
      b[current] = c[minIndex(c)];
      c[minIndex(c)] = Integer.MAX_VALUE;
      sort(c, b, current+1);
  }
```

Recursive Binary Search

- Case 1: If the key is less than the middle element, recursively search the key in the first half of the array.
- 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.

- Demo Program: RecursiveBinarySearch.java

Recursive Implementation

```
/** Use binary search to find the key in the list */
public static int recursiveBinarySearch(int[] list, int key) {
  int low = 0;
  int high = list.length - 1;
  return recursiveBinarySearch(list, key, low, high);
}

/** Use binary search to find the key in the list between
    list[low] list[high] */
public static int recursiveBinarySearch(int[] list, int key,
  int low, int high) {
  if (low > high)  // The list has been exhausted without a match
    return -low - 1;

  int mid = (low + high) / 2;
  if (key < list[mid])
    return recursiveBinarySearch(list, key, low, mid - 1);
  else if (key == list[mid])
    return mid;
  else
    return recursiveBinarySearch(list, key, mid + 1, high);
}
```

Demonstration Program

RECURSIVEPROCESSINGIII_ALGORITHMS.JAVA

Recursive Selection Sort (by swap)

- Find the smallest number in the list and swaps it with the first number.
- Ignore the first number and sort the remaining smaller list recursively.

Demonstration Program

RECURSIVESELECTIONSORT.JAVA

Sort

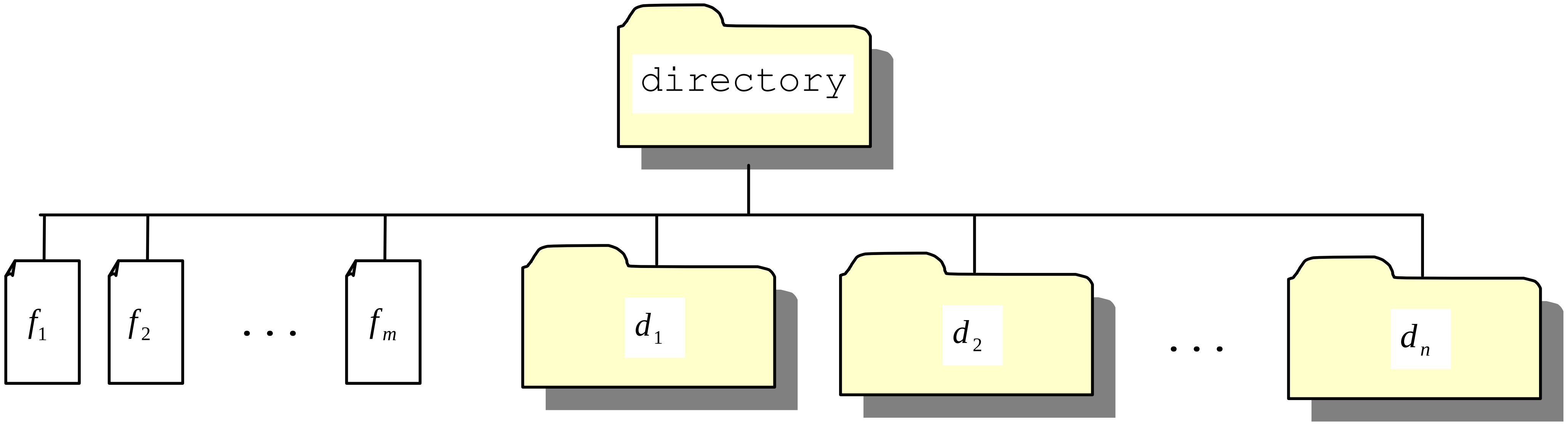
```
public static void sort(double[] list) {
   sort(list, 0, list.length - 1); // Sort the entire list
 }

private static void sort(double[] list, int low, int high) {
   if (low < high) {
     // Find the smallest number and its index in list(low .. high)
     int indexOfMin = low;
     double min = list[low];
     for (int i = low + 1; i <= high; i++) {
       if (list[i] < min) {
         min = list[i];
         indexOfMin = i;
       }
     }
     // Swap the smallest in list(low .. high) with list(low)
     list[indexOfMin] = list[low];
     list[low] = min;
     // Sort the remaining list(low+1 .. high)
     sort(list, low + 1, high);
   }
 }
```

Directory Size

- The preceding examples can easily be solved without using recursion. This section presents a problem that is difficult to solve without using recursion. The problem is to find the size of a directory.
- The size of a directory is the sum of the sizes of all files in the directory.
- A directory may contain subdirectories. Suppose a directory contains files , , ..., , and subdirectories , , ..., , as shown below.

Directory Size



DirectorySize

```
public static void main(String[] args) {
    String directory =
       "C:\\Eric_Chou\\Udemy\\APCSB\\BlueJ\\Chapter14";

    // Display the size
    System.out.println(getSize(new File(directory)) + " bytes");
  }
```

DirectorySize

```
public static long getSize(File file) {
  long size = 0; // Store the total size of all files
  if (file.isDirectory()) {
    File[] files = file.listFiles(); // All files and subdirectories
    for (int i = 0; i < files.length; i++) {
      size += getSize(files[i]); // Recursive call
    }
  }
  else { // Base case
    size += file.length();
  }
  return size;
}
```

Directory Size

The size of the directory can be defined recursively as follows:

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

Demonstration Program

DIRECTORYSIZE.JAVA

Recursion vs. Iteration

LECTURE 1

Recursion vs. Iteration

- Recursion is an alternative form of program control. It is essentially repetition without a loop.

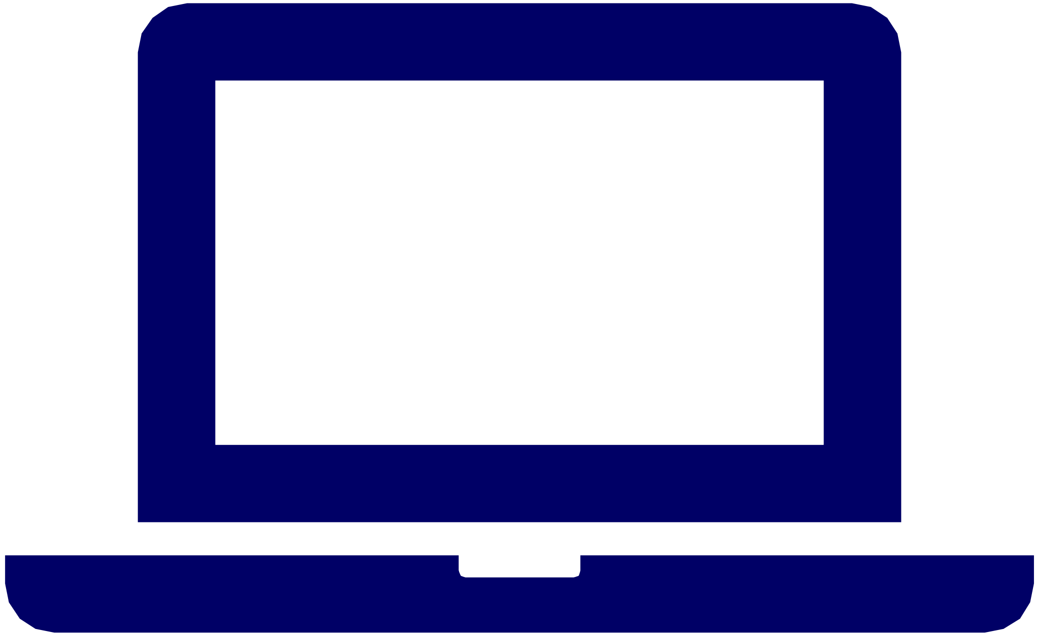
- Recursion bears substantial overhead. Each time the program calls a method, the system must assign space for all of the method’s local variables and parameters. This can consume considerable memory and requires extra time to manage the additional space.

	Iteration	Recursion
Control	Loop Statement	Recursive Call
Local Variables	Required	Not Required
Assignments	Required	Not Required
Style	Imperative	Declarative
Size	Larger	Smaller
Nontermination	Infinite Loop	Infinite Recursion

Fibonacci Number

```
int fibonacci(int n){
   int[] a = new int[n];
   a[0] = 1;
    a[1] = 1;
    for (int i = 2; i<=n; i++){
       a[i] = a[i-1] + a[i-2];
     }
    return a[n];
} // runs faster
```

```
int fibonacci(int n){
   if (n == 0 || n == 1) return 1;
   return fibonacci(n-1) +
          fibonacci(n-2);
} // shorter code
```



Demonstration Program

FIBONACCINUMBERS.JAVA

BlueJ: Terminal Window - Chapter14

Options

```
n=1  Iterative Time: 0  Recursive Time: 0   Difference: 0
n=2  Iterative Time: 0  Recursive Time: 0   Difference: 0
n=3  Iterative Time: 0  Recursive Time: 0   Difference: 0
n=4  Iterative Time: 0  Recursive Time: 0   Difference: 0
n=5  Iterative Time: 0  Recursive Time: 0   Difference: 0
n=6  Iterative Time: 0  Recursive Time: 0   Difference: 0
n=7  Iterative Time: 0  Recursive Time: 0   Difference: 0
n=8  Iterative Time: 0  Recursive Time: 1   Difference: 1
n=9  Iterative Time: 0  Recursive Time: 0   Difference: 0
n=10  Iterative Time: 0  Recursive Time: 0   Difference: 0
n=11  Iterative Time: 0  Recursive Time: 0   Difference: 0
n=12  Iterative Time: 0  Recursive Time: 1   Difference: 1
n=13  Iterative Time: 0  Recursive Time: 2   Difference: 2
n=14  Iterative Time: 0  Recursive Time: 1   Difference: 1
n=15  Iterative Time: 0  Recursive Time: 1   Difference: 1
n=16  Iterative Time: 0  Recursive Time: 0   Difference: 0
n=17  Iterative Time: 0  Recursive Time: 1   Difference: 1
n=18  Iterative Time: 0  Recursive Time: 1   Difference: 1
n=19  Iterative Time: 0  Recursive Time: 7   Difference: 7
n=20  Iterative Time: 0  Recursive Time: 4   Difference: 4
n=21  Iterative Time: 0  Recursive Time: 11   Difference: 11
n=22  Iterative Time: 0  Recursive Time: 12   Difference: 12
n=23  Iterative Time: 0  Recursive Time: 28   Difference: 28
n=24  Iterative Time: 0  Recursive Time: 46   Difference: 46
n=25  Iterative Time: 0  Recursive Time: 147   Difference: 147
n=26  Iterative Time: 0  Recursive Time: 160   Difference: 160
n=27  Iterative Time: 0  Recursive Time: 488   Difference: 488
n=28  Iterative Time: 0  Recursive Time: 618   Difference: 618
n=29  Iterative Time: 0  Recursive Time: 2113   Difference: 2113
n=30  Iterative Time: 0  Recursive Time: 2478   Difference: 2478
```

Recursive program for fibonacci is not efficient.

Advantages of Using Recursion

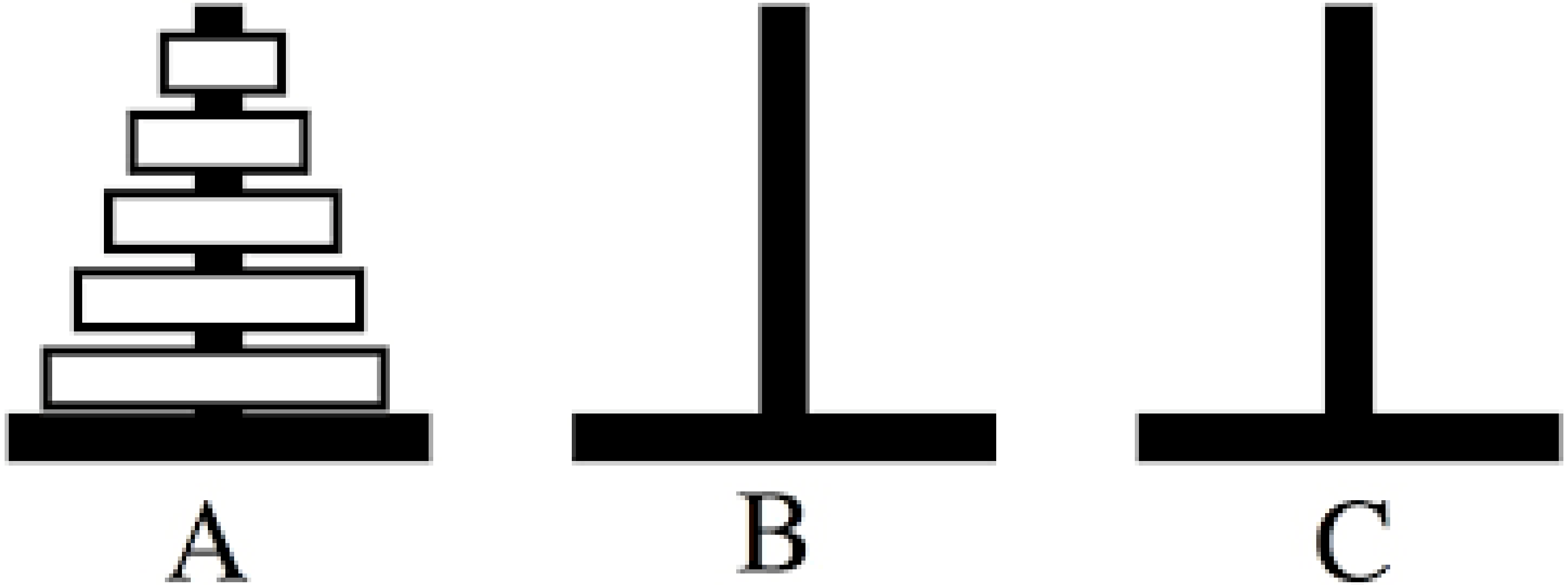
Recursion is good for solving the problems that are inherently recursive. (Especially 1st order)

Tower of Hanoi

(Non-AP Topic)

LECTURE 1

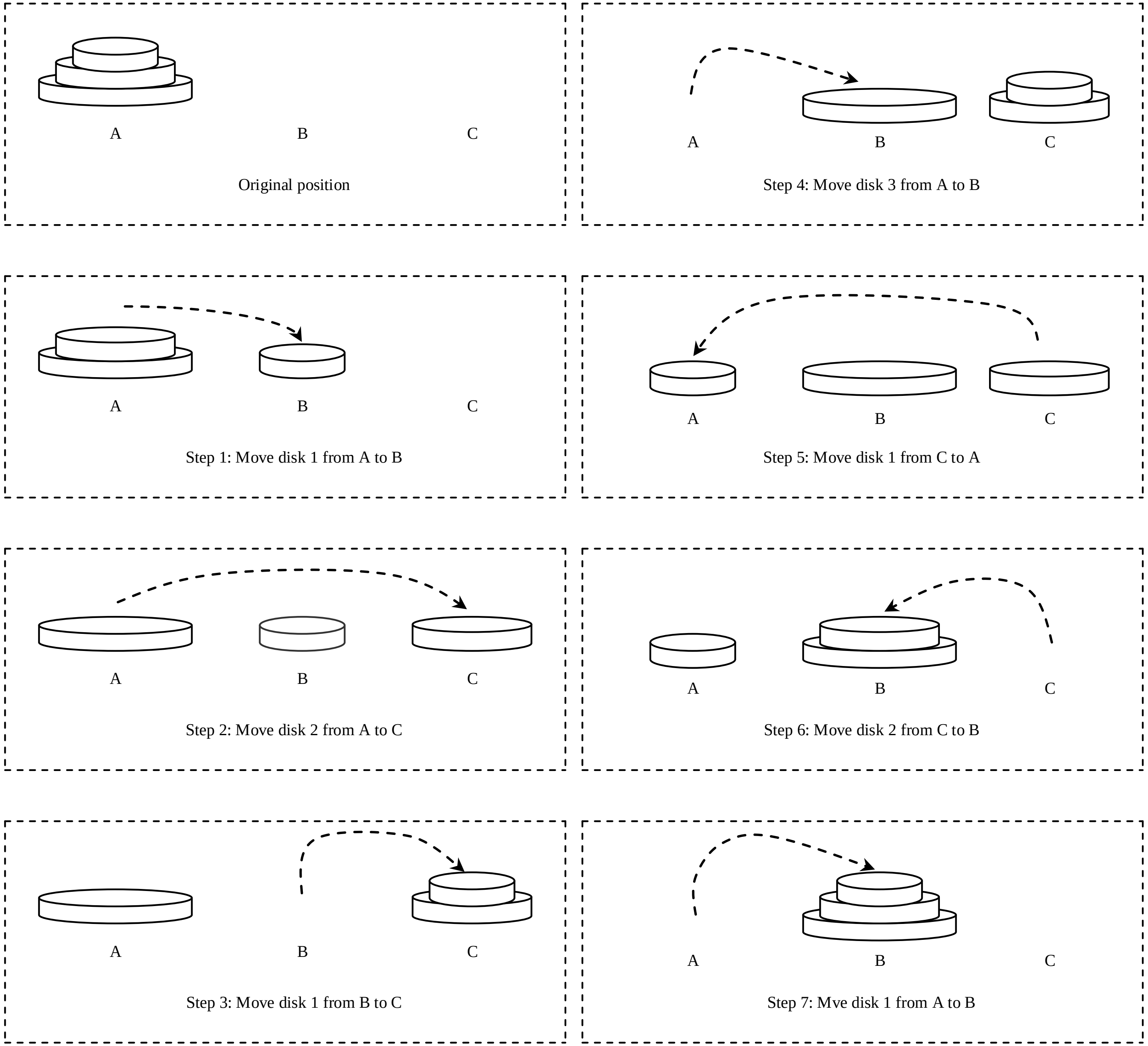
Towers of Hanoi



Towers of Hanoi

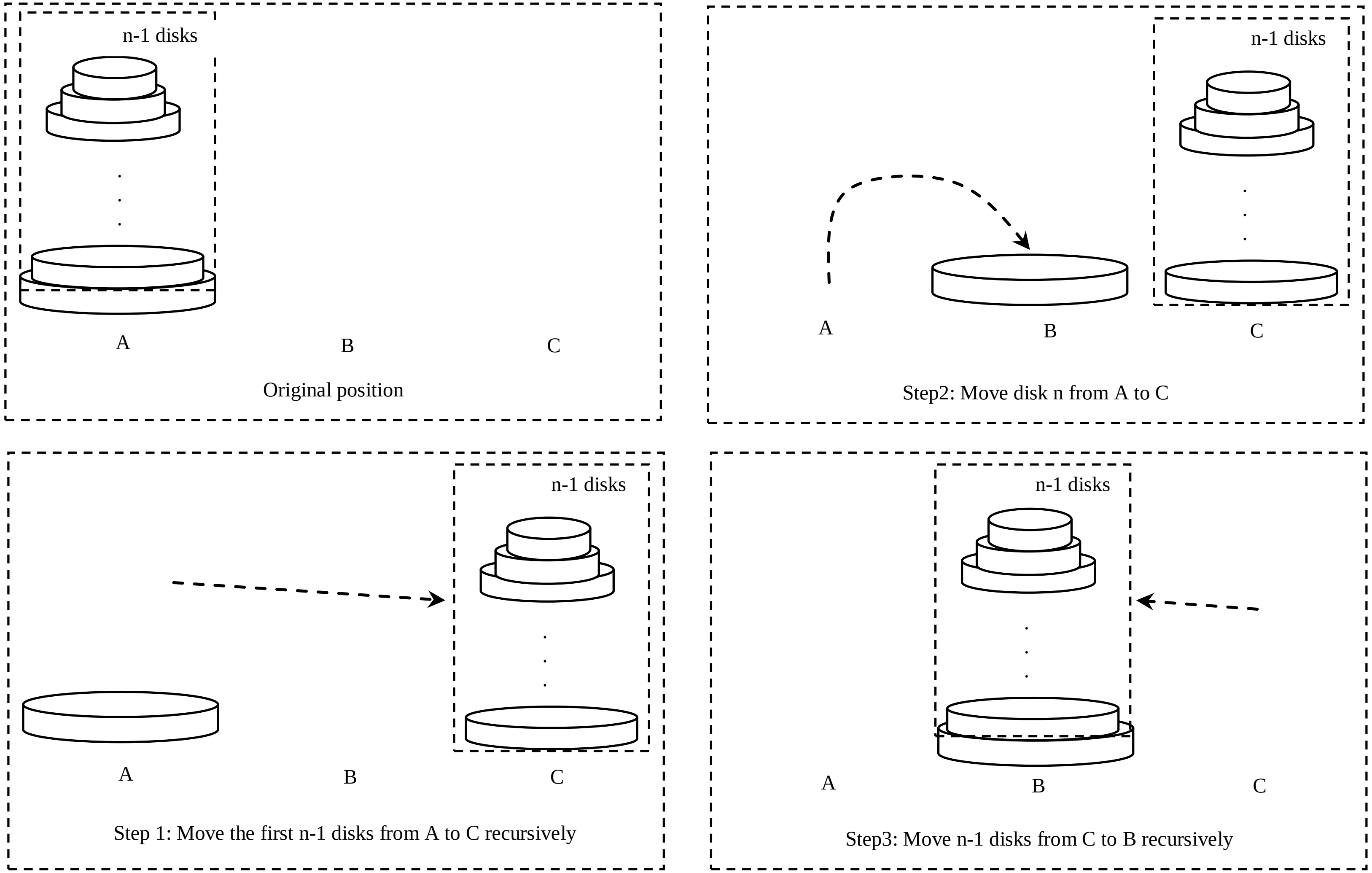
- There are n disks labeled 1, 2, 3, . . ., n, and three towers labeled A, B, and C.
- No disk can be on top of a smaller disk at any time.
- All the disks are initially placed on tower A.
- Only one disk can be moved at a time, and it must be the top disk on the tower.

Towers of Hanoi



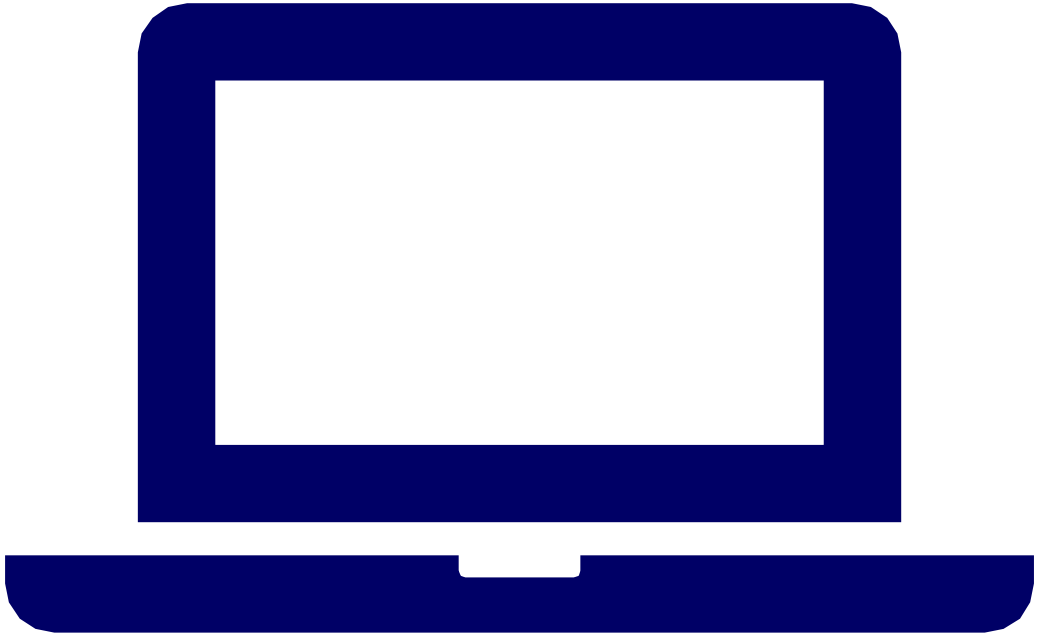
Solution to Towers of Hanoi

The Towers of Hanoi problem can be decomposed into three subproblems.



Solution to Towers of Hanoi

- Move the first **n - 1** disks from A to C with the assistance of tower B.
- Move disk **n** from A to B.
- Move **n - 1** disks from C to B with the assistance of tower A.



Demonstration Program

TOWERSOFHANOI.HAVA