Recursive Algorithms

Objectives

- After you have read and studied this chapter, you should be able to
  - Write recursive algorithms for mathematical functions and nonnumerical operations.
  - Decide when to use recursion and when not to.
  - Describe the recursive quicksort algorithm and explain how its performance is better than selection and bubble sort algorithms.

Recursion

- The *factorial of N* is the product of the first N positive integers:

$$N * (N-1) * (N-2) * \cdots * 2 * 1$$

- The factorial of N can be defined *recursively* as

$$\texttt{factorial(N) =} \begin{cases} \texttt{1} & \texttt{if N = 1} \\ \texttt{N * factorial(N-1)} & \texttt{otherwise} \end{cases}$$

Recursive Method

- An *recursive method* is a method that contains a statement (or statements) that makes a call to itself.
- Implementing the factorial of N recursively will result in the following method.

Test to stop or continue.

End case: recursion stops.

Recursive case: recursion continues.

```
public int factorial( int N ) {

   if ( N == 1 ) {
      return 1;
   }
   else {
      return N * factorial( N-1 );
   }
}
```

Anagram

- List all anagrams of a given word.

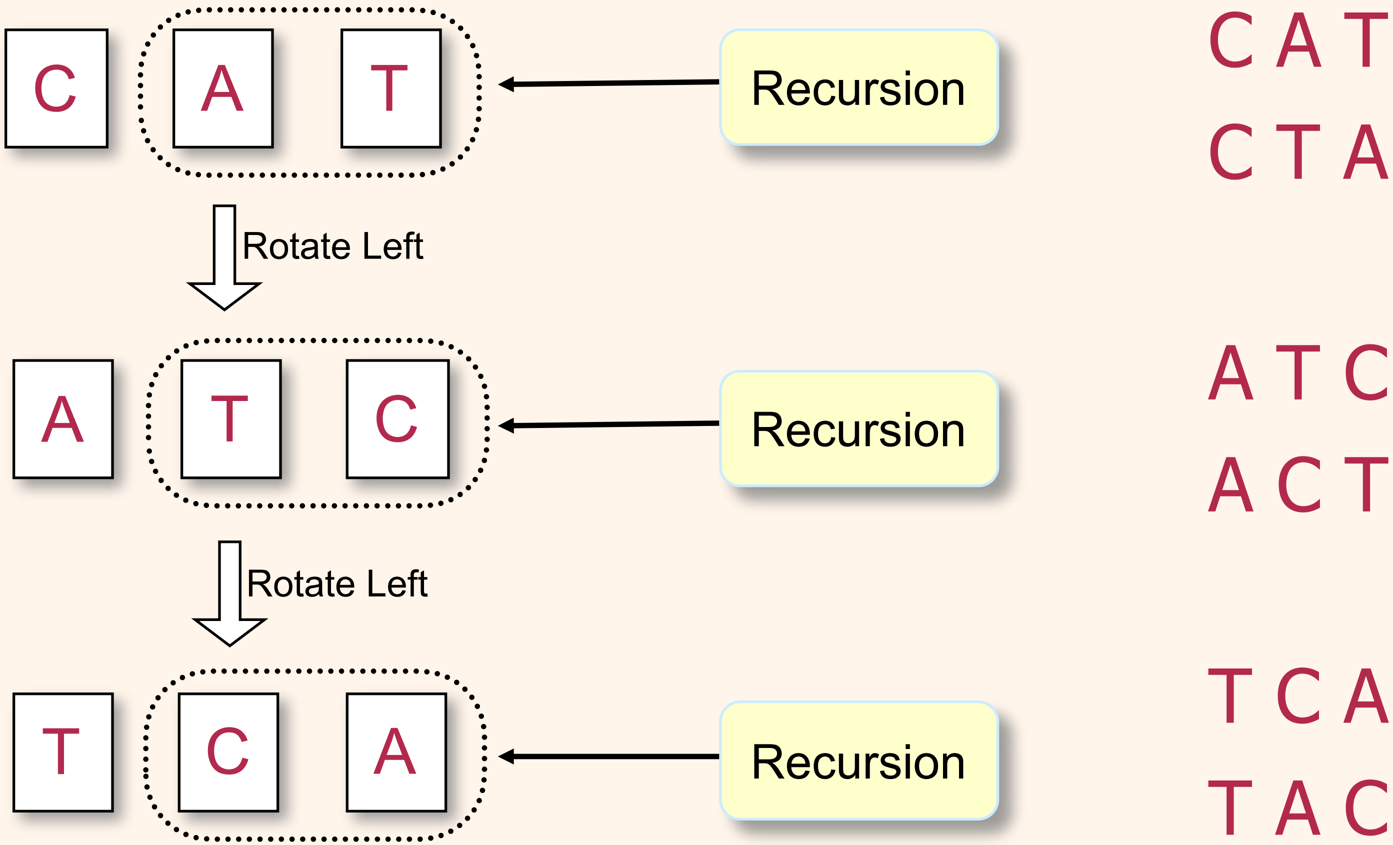
Word → C A T

C T A
A T C
A C T
T C A
T A C

← Anagrams

Anagram Solution

- The basic idea is to make recursive calls on a sub-word after every rotation. Here's how:



Anagram Method

```
public void anagram( String prefix, String suffix ) {
   String newPrefix, newSuffix;
   int numOfChars = suffix.length();

   if (numOfChars == 1) {
      //End case: print out one anagram
      System.out.println( prefix + suffix );

   } else {
      for (int i = 1; i <= numOfChars; i++ ) {
         newSuffix = suffix.substring(1, numOfChars);
         newPrefix = prefix + suffix.charAt(0);
         anagram( newPrefix, newSuffix );
         //recursive call
         //rotate left to create a rearranged suffix
         suffix = newSuffix + suffix.charAt(0);
      }
   }
}
```

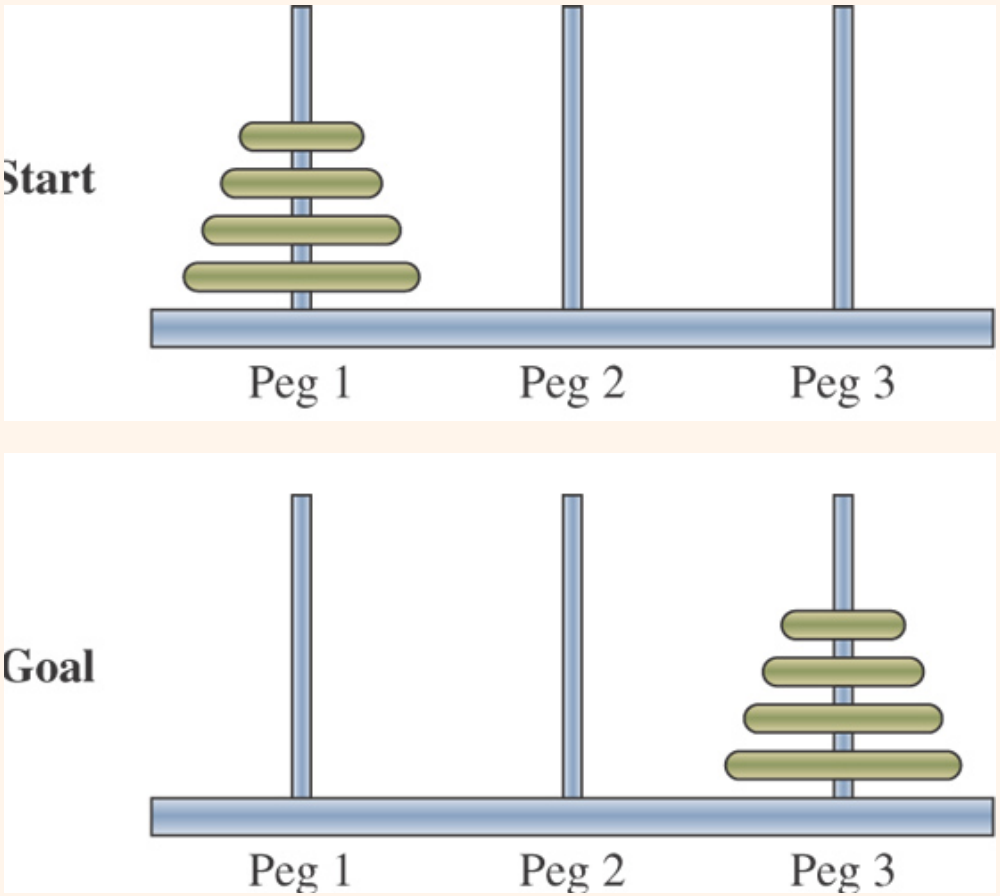
Test

End case

Recursive case

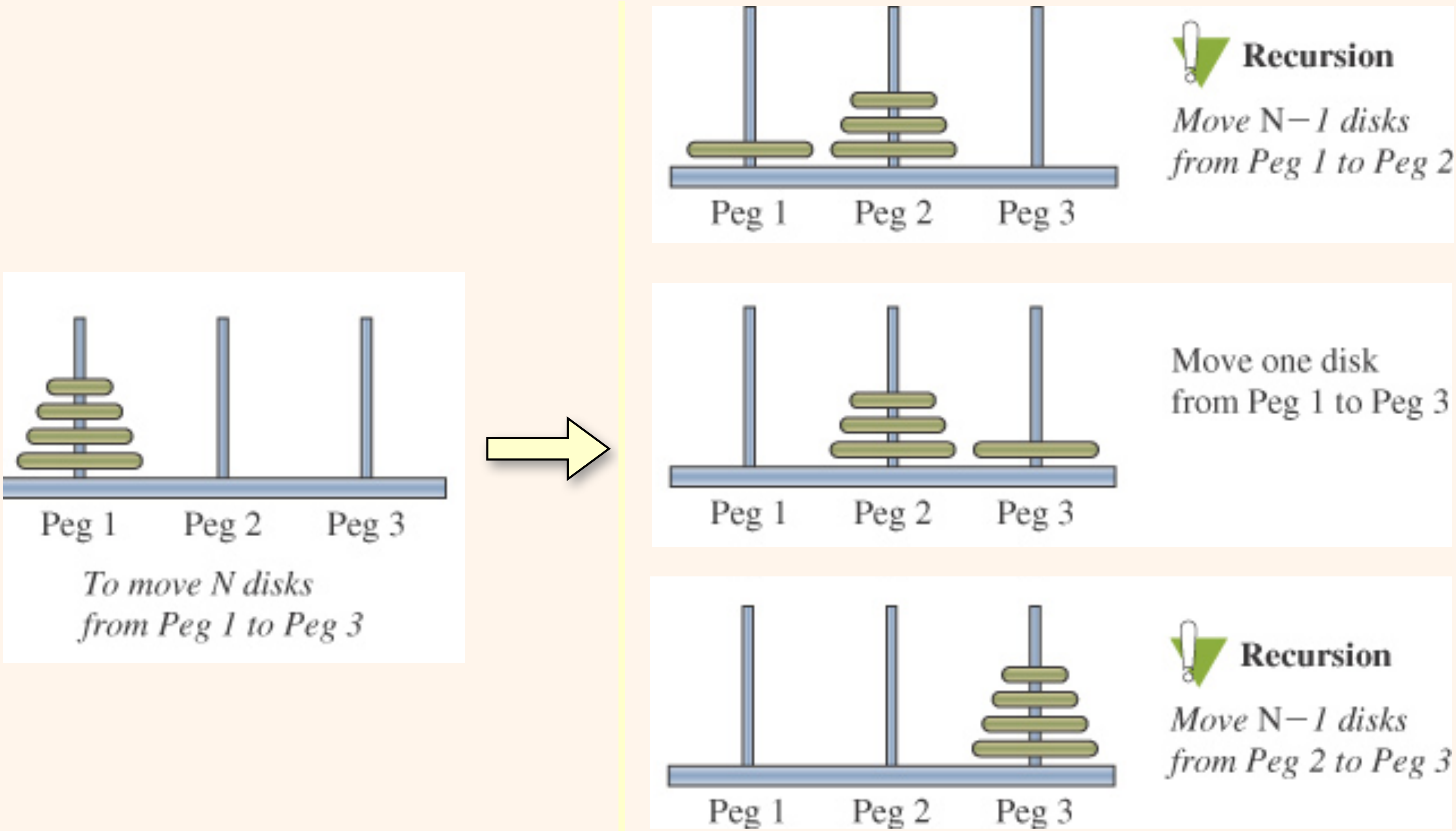
Towers of Hanoi

- The goal of the Towers of Hanoi puzzle is to move N disks from peg 1 to peg 3:



  - You must move one disk at a time.
  - You must never place a larger disk on top of a smaller disk.

Towers of Hanoi Solution



towersOfHanoi Method

Test

End case

Recursive case

```
public void towersOfHanoi(int N,        //number of disks
                          int from,   //origin peg
                          int to,     //destination peg
                          int spare ){//"middle" peg

  if ( N == 1 ) {

     moveOne( from, to );

  } else {
     towersOfHanoi( N-1, from, spare, to );
     moveOne( from, to );
     towersOfHanoi( N-1, spare, to, from );
  }
}

private void moveOne( int from, int to ) {
   System.out.println( from + " ---> " + to );
}
```

When Not to Use Recursion

- When recursive algorithms are designed carelessly, it can lead to very inefficient and unacceptable solutions.
- For example, consider the following:

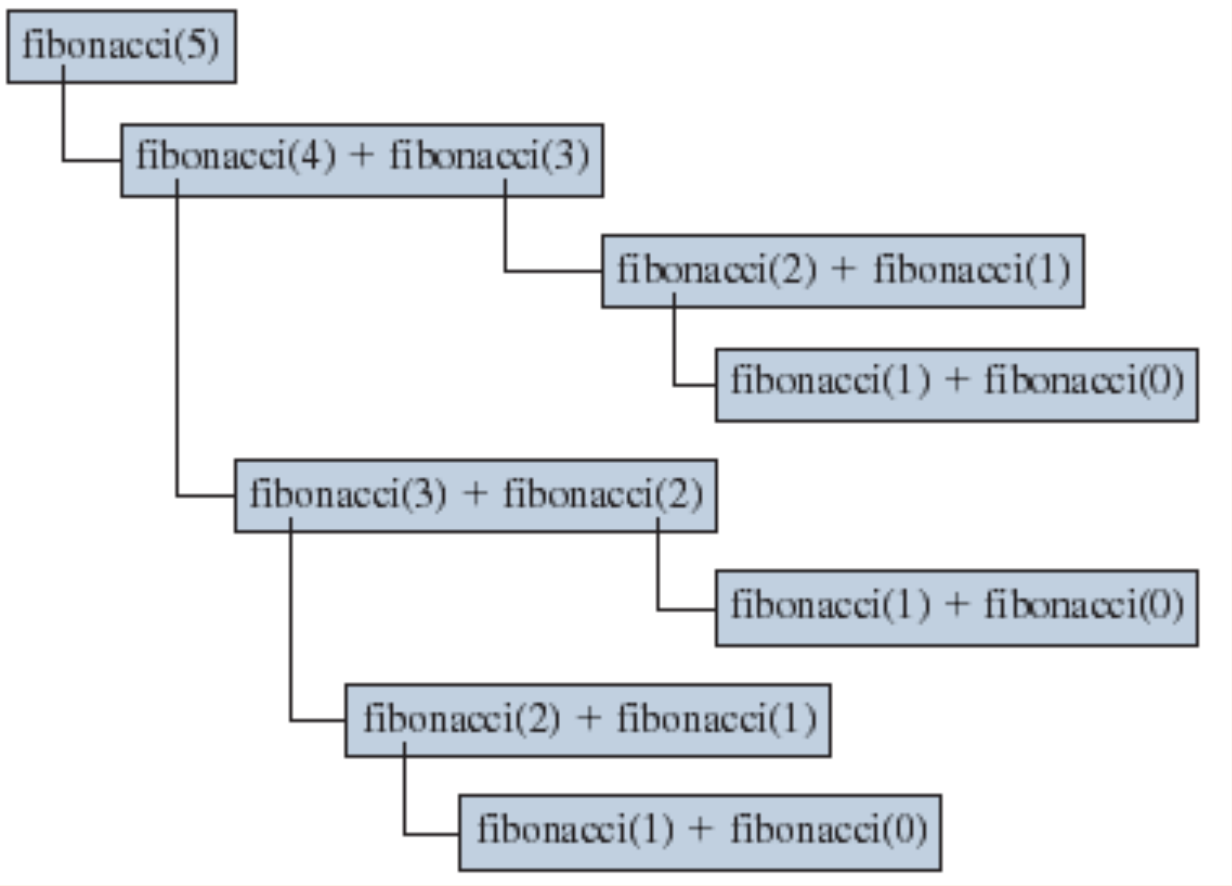
```
public int fibonacci( int N ) {

   if (N == 0 || N == 1) {
      return 1;

   } else {
      return fibonacci(N-1) + fibonacci(N-2);
   }
}
```

Excessive Repetition

- Recursive Fibonacci ends up repeating the same computation numerous times.



Nonrecursive Fibonacci

```
public int fibonacci( int N ) {

   int fibN, fibN1, fibN2, cnt;

   if (N == 0 || N == 1 ) {
      return 1;

   } else {

      fibN1 = fibN2 = 1;
      cnt = 2;
      while ( cnt <= N ) {
         fibN = fibN1 + fibN2; //get the next fib no.
         fibN1 = fibN2;
         fibN2 = fibN;
         cnt ++;
      }
      return fibN;
   }
}
```

When Not to Use Recursion

- In general, use recursion if
  - A recursive solution is natural and easy to understand.
  - A recursive solution does not result in excessive duplicate computation.
  - The equivalent iterative solution is too complex.