CSI33 Data Structures

Department of Mathematics and Computer Science
Bronx Community College

Outline

1. Chapter 6: Recursion
   - Analyzing Recursion
   - Sorting
   - The Tower of Hanoi
   - Divide-and-conquer Approach
   - In-class Work

Measuring Complexity (Running Time) Of Recursive Algorithms

Comparison With Iterative (Looping) Algorithms

- Any iterative algorithm can be transformed into a recursive one.
- Different strategies lead to different running times. (The recursive power example is more efficient than the naive loop version.)
- To measure efficiency, you must count **recursive calls** and **the depth of the call stack**.
- You must also consider the **size of the data parameters** that are passed in recursive calls.

The Fibonacci Sequence

The Fibonacci Sequence

The Fibonacci Sequence is obtained by beginning with the pair of numbers $1, 1$ and continuing indefinitely by adding the last two numbers to give the next number in the sequence, giving 1, 1, 2, 3, 5, 8, 13 and so on.

The Fibonacci Sequence

The nth Fibonacci Number: Loop Version

```
def loopFib(n):
    curr = 1
    prev = 1
    for i in range(n - 2):
        curr, prev = curr + prev, curr
    return curr
```

Analysis

To calculate `fib(n)` requires $n - 2$ iterations of the for loop, so the running time is $\Theta(n)$.

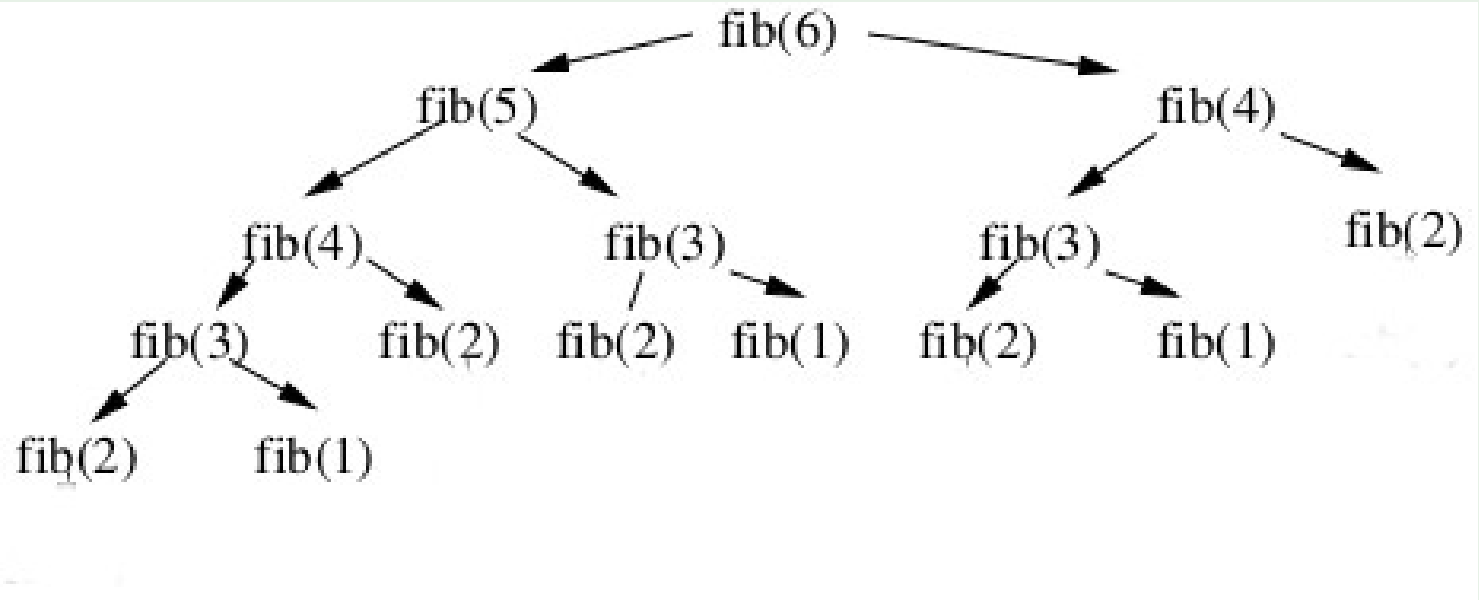
The Fibonacci Sequence

The nth Fibonacci Number: Recursive Version

```
def recFib(n):
    if n < 3:
        return 1
    else:
        return recFib(n - 1) + recFib(n - 2)
```

The Fibonacci Sequence

Analysis



The Fibonacci Sequence

Analysis

To calculate `fib(6)` is very wasteful:

- `fib(4)` is calculated 2 times
- `fib(3)` is calculated 3 times
- `fib(2)` is calculated 5 times
- `fib(1)` is calculated 8 times

To calculate `fib(n)` requires $fib(n) - 1$ steps, so the running time is $\Theta(fib(n))$, which is $\Theta(2^n)$, or exponential in n.

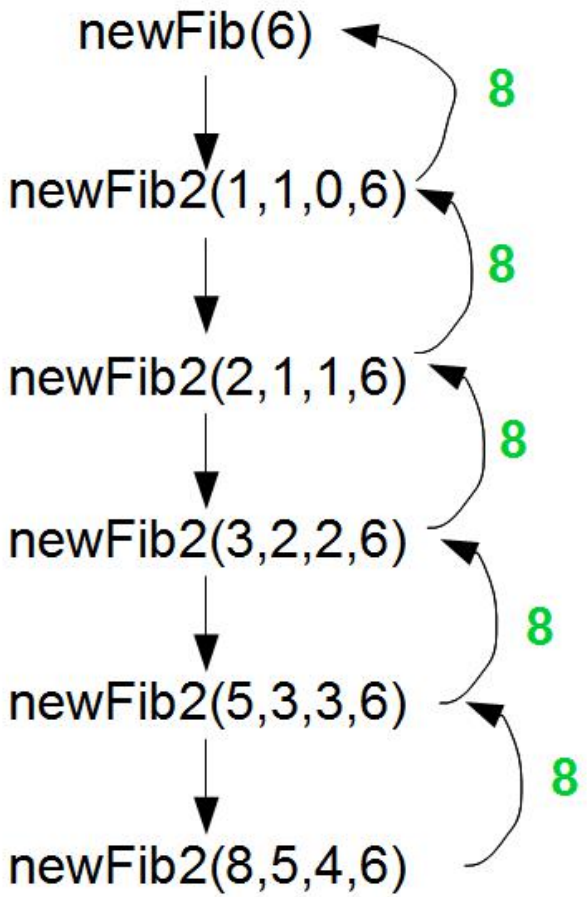
The Fibonacci Sequence

The nth Fibonacci Number: Improved Recursive Version

```
def newFib(n):
   return newFib2(1, 1, 0, n)
def newFib2(curr, prev, i, n):
   if i == n - 2:
      return curr
   else:
      return newFib2(curr + prev,curr, i + 1, n)
```

You can see that this definition won't work if a user wants to get the first Fibonacci number, i.e. 1, but it works perfectly well for all other cases. Can it be fixed? (class work)

The Fibonacci Sequence



The Fibonacci Sequence

Analysis

To calculate `fib(n)` now requires $n - 2$ recursive calls, so the running time is $\Theta(n)$, which is a big improvement.

The Fibonacci Sequence

How To Make An Iterative Function Recursive

- Write a function that calls a helper function with parameters for all local variables and parameters from the loop version.
- Pass the initial values from the loop version in this function call.
- The helper function will be recursive:
- The base case will be the negation of the loop condition.
- The recursive call will change the parameters to match one iteration of the loop version.

Selection Sort

Selection Sort

Recall Selection Sort you were asked to program in one of the hws:

```
def SelectionSort(lst):
    n = len(lst)
    for i in range(n-1):
        pos = i
        for j in range(i+1, n):
            if lst[j] < lst[pos]:
                pos = j
        lst[i], lst[pos] = lst[pos], lst[i]
```

Selection Sort

Selection Sort Analysis

- Inner loop runs n times
- First time it compares n items, then $n-1$, etc.
- Total comparisons $= n + (n-1) + (n-2) + \ldots + 1 = \frac{n(n+1)}{2}$
- Running time is $\Theta(n^2)$

Recursive Design: Mergesort

Mergesort Pseudocode

Now, let's take a look at a recursive sorting algorithm:

```
Algorithm:  mergeSort nums

   split nums into two halves (nums1, nums2)
   sort nums1 (the first half)
   sort nums2 (the second half)
   merge nums1 and nums2 back into nums
```

Recursive Design: Mergesort

Merge Pseudocode

```
Algorithm:  merge sorted lists (nums1 and nums2) into nums:
   while both nums1 and nums2 have more items:
      if top of nums1 is smaller:
         copy it into current spot in nums
      else (top of nums2 is smaller):
         copy it into current spot in nums
   copy remaining items from nums1 or nums2 to nums
```

See the definition of the `merge` function in `mergeSort.py`.

Recursive Design: Mergesort

Recursive mergeSort - with the base case

```
if len(nums) > 1:
   split nums into two halves (nums1, nums2)
   mergeSort nums1 (the first half)
   mergeSort nums2 (the second half)
   merge nums1 and nums2 back into nums
```

See the definition of the `mergeSort` function in `mergeSort.py`.

Analyzing Mergesort

Running Time of merge

- Each item gets moved exactly once back into nums
- Running time is $\Theta(n)$, where n is the size of nums

Merge Pseudocode

```
Algorithm:  merge sorted lists (nums1 and nums2) into nums:
   while both nums1 and nums2 have more items:
      if top of nums1 is smaller:
         copy it into current spot in nums
      else (top of nums2 is smaller):
         copy it into current spot in nums
   copy remaining items from nums1 or nums2 to nums
```

Analyzing Mergesort

Running Time of mergeSort

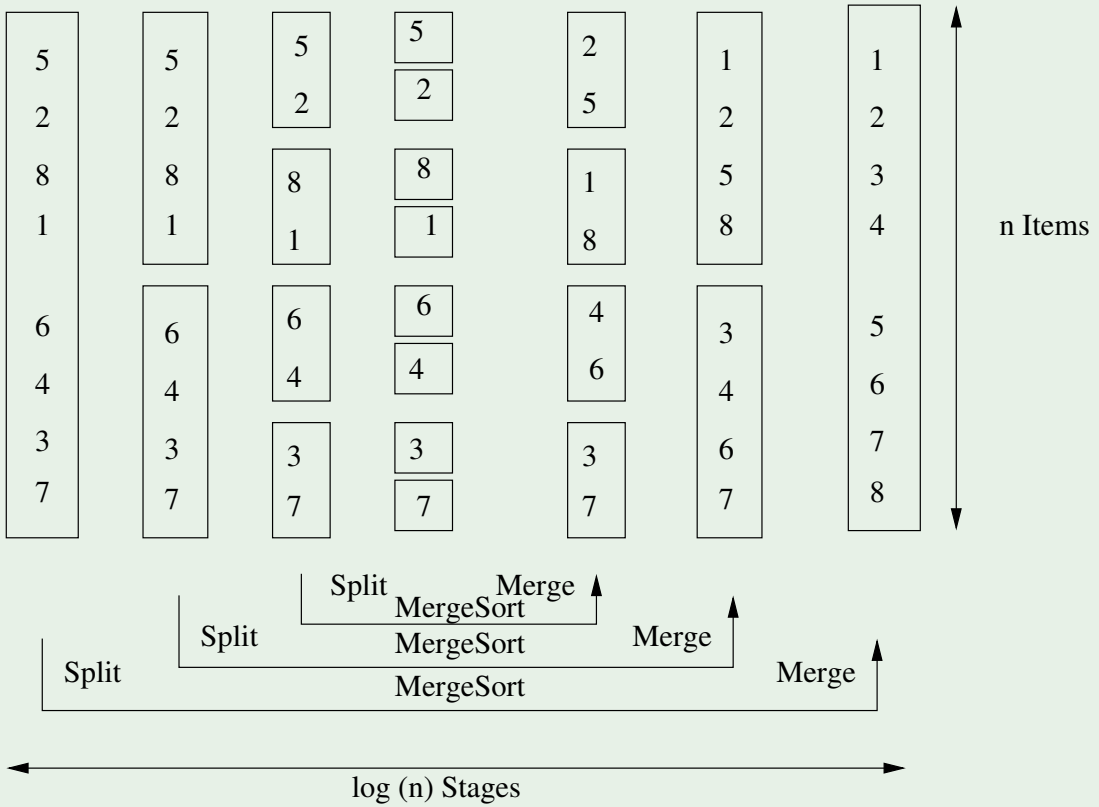
- The call stack gets as deep as $\log_2(n)$, where n is the size of `nums`
- At each stage, `mergeSort` is called twice, but for each call, the argument list is half the size as before.
- For $\log_2(n)$ stages, each of the n items is moved once per stage.
- The running time is the product, which is $\Theta(n \log n)$

Recursive mergeSort - with the base case

```
if len(nums) > 1:
   split nums into two halves (nums1, nums2)
   mergeSort nums1 (the first half)
   mergeSort nums2 (the second half)
   merge nums1 and nums2 back into nums
```

Analyzing Mergesort

Running Time of mergeSort



Tower of Hanoi rules

Tower of Hanoi or Tower of Brahma is a puzzle generally attributed to the French mathematician Édouard Lucas, who published an article about it in 1883.

Read the legend surrounding the puzzle on page 207 in the book.

Tower of Hanoi rules

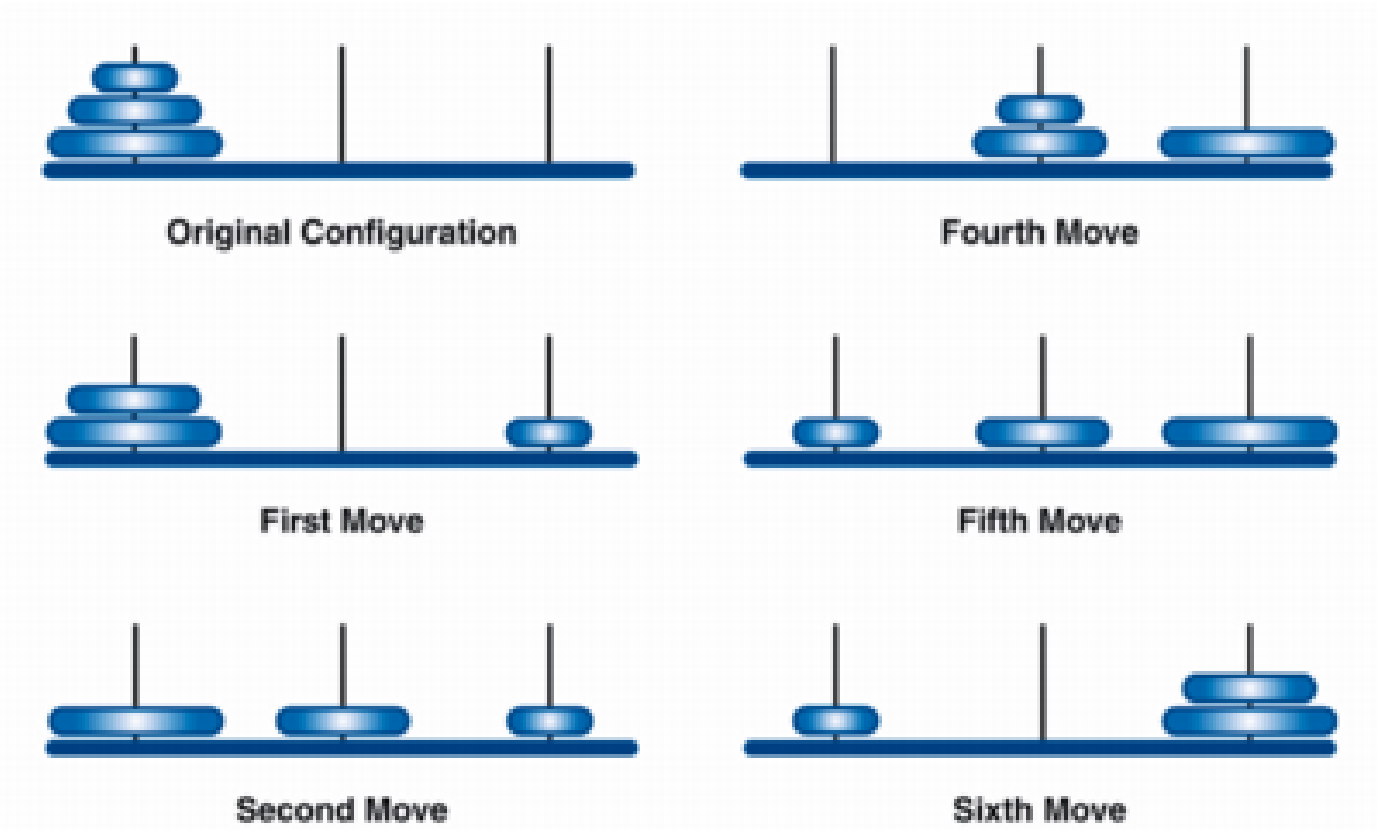
The objective of the puzzle is to move the entire stack to another rod, obeying the following simple rules:

- Only one disk can be moved at a time.
- Each move consists of taking the upper disk from one of the stacks and placing it on top of another stack.
- No disk may be placed on top of a smaller disk.

With three disks, the puzzle can be solved in seven moves. The minimum number of moves required to solve a Tower of Hanoi puzzle is $2^n - 1$, where n is the number of disks.

Tower of Hanoi rules

3-disks Tower of Hanoi with solution:

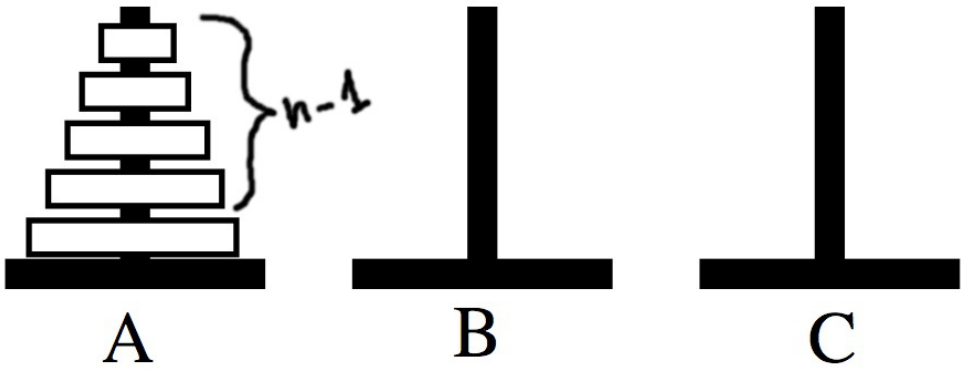


Tower of Hanoi rules

Recursive Solution

Algorithm: move n-disk tower from source to destination.

move $n-1$ disk tower from source to resting place
move 1 disk tower from source to destination
move $n-1$ disk from resting place to destination



Divide-and-conquer Approach

Divide and conquer is derived from the Latin saying *Divide et impera*.

In computer science, divide and conquer is an important algorithm design paradigm based on multi-branched recursion.

A divide and conquer algorithm works by recursively breaking down a problem into two or more sub-problems of the same (or related) type, until these become simple enough to be solved directly.

The solutions to the sub-problems are then combined to give a solution to the original problem.

Divide-and-conquer Approach

The following algorithms from the ones we covered so far employ this paradigm:

Binary Search (both versions) and Merge sort.

In-class Work

- Fix the `newFib` function to make it work when `newFib(1)` is called.
- Use Merge Sort to sort the following numbers: 5,1,6,2,8,3,9

  Show the graphical representation of the sort (use lecture slides)
- solve the Tower of Hanoi puzzle for four discs.