Chapter 2 Recursion

Fei Tan

Chaifetz School of Business
Saint Louis University

Computer Science Fundamentals
(Source: brilliant.org)

December 6, 2018

The Road Ahead...

- When problems are just a bigger version of themselves, recursion is always the answer
- What we'll accomplish
  - understand & apply Fibonacci sequence
  - conquer Towers of Hanoi

Algorithm: Recursion

- Recursion is self-embedded structure or process
  - useful in CS: simplify code; speed up program by requiring less information to be stored
  - e.g., Fibonacci sequence: $F_n = F_{n-1} + F_{n-2}$, $n \geq 2$

Python code

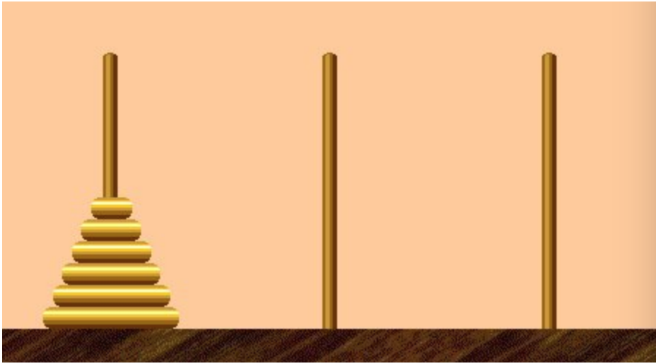
```python
def fibonacci(n):
    if n < 0:
        raise ValueError("invalid index!")
    # define base case
    if n == 0:
        return 0
    if n == 1:
        return 1
    return fibonacci(n - 1) + fibonacci(n - 2)
```

Algorithm: Binary Search Revisited

Python code

```
def binarySearch(A, item):
    if len(A) == 0:
        return False
    else:
        middle = len(A) // 2
        if A[middle] == item:
            return True
        if item<A[middle]:
        #A[:middle] is a copy of elements to the
            left of middle of A
            return binarySearch(A[:middle], item)
        else:
        #A[middle+1:] is a copy of elements to the
            right of middle of A
            return binarySearch(A[middle + 1:],
                item)
```

Algorithm: Divide and Conquer



- **Divide** problem into sub-problems; **conquer** each problem; **combine** results
- Useful for matrix multiplication, sorting algorithms, calculating Fourier transform, etc.
- How to solve Towers of Hanoi?

Pseudo Code

Solving Towers of Hanoi

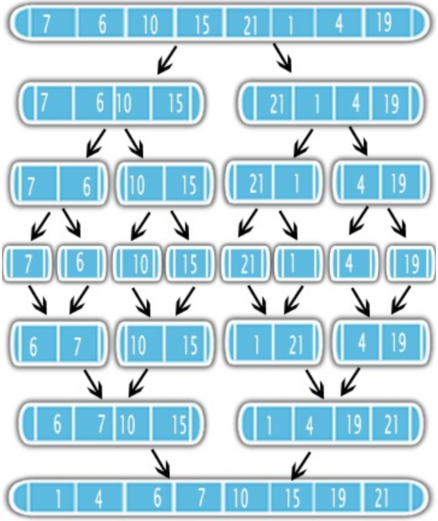
```
move(n,A,B):

Let C be the third tower (that isn't A or B)

If (n == 1):
    move top disk on A to B

If (n>1):
    move(n-1,A,C)
    move nth smallest disk from A to B
    move(n-1,C,B)
```

Algorithm: Mergesort



- Mergesort applies divide & conquer to sorting

Python Code

```
def merge(left, right):
    result = []
    left_idx, right_idx = 0, 0
    while left_idx < len(left) and right_idx < len
        (right):
        # change comparison direction to change
            sort direction
        if left[left_idx] <= right[right_idx]:
            result.append(left[left_idx])
            left_idx += 1
        else:
            result.append(right[right_idx])
            right_idx += 1
    if left:
        result.extend(left[left_idx:])
    if right:
        result.extend(right[right_idx:])
    return result
```

Python Code (Cont'd)

```python
def mergeSort(m):
    if len(m) <= 1:
        return m

    middle = len(m) // 2
    left = m[:middle]
    right = m[middle:]

    left = mergeSort(left)
    right = mergeSort(right)
    return list(merge(left, right))
```

- Time complexity
  - division into two parts: $O(1)$
  - solving subproblems: $2T(N/2)$
  - combining subproblems: $O(N)$ (N comparisons)
  - Master Theorem: $T(N) = 2T(N/2) + O(N) = O(N \log N)$

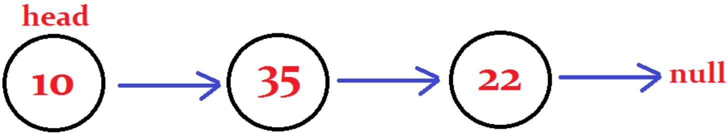
Algorithm: Quicksort

- Framed in ‘divide and conquer’ framework
  - divide/partition: pick pivot element; all numbers $<$ pivot go to left, all elements $>$ pivot go to right
  - conquer: feed all left/right elements recursively into algorithm
  - combine: no need
- Common approaches to pick pivot
  - select random pivot
  - select first/last element of array
  - median of three: median of first, middle, and last elements
  - use algorithm to find median
- Time complexity
  - random data: quicksort 2–3 times faster than mergesort
  - quicksort: $O(N^2)$–$O(N \log N)$; mergesort: $O(N \log N)$

Python Code

```python
def quickSort(arr):
    less = []
    pivotList = []
    more = []
    arr_length = len(arr)
    if arr_length <= 1:
        return arr
    else:
        pivot = arr[0]
        for i in arr:
            if i < pivot:
                less.append(i)
            elif i > pivot:
                more.append(i)
            else:
                pivotList.append(i)
        less = quickSort(less)
        more = quickSort(more)
        return less + pivotList + more
```

Data Structure: Linked Lists



- Linked list holds data in nodes; each node holds both data and pointer to another node (i.e., link)
- Time complexity:
  - array: $O(1)$ access, $O(N)$ insertion
  - linked list: $O(N)$ access, $O(1)$ insertion
- Lists can be doubly linked/bi-directional to simplify deletion

Python Code

Implementing linked list (F)–>(I)–>(S)–>(H)–>null

```
Node1.data = "F"
Node1.next = Node2
Node2.data = "I"
Node2.next = Node3
Node3.data = "S"
Node3.next = Node4
Node4.data = "H"
Node4.next = null

def traverse(node):
    if node.next != null:
        traverse(node.next)
    print node.data

#output of traverse(Node1): HSIF
```