Computational Structures in Data Science

Tree Recursion
Learning Objectives

• Write Recursive functions with multiple recursive calls
• Understand Recursive Fibonacci
• Understand the quicksort algorithm
Tree Recursion

• Recursion which involves multiple recursive calls to solve a problem.
• Drawing out a function usually looks like an “inverted” tree.
Example I

List all items on your hard disk

- Files
- Folders contain
  - Files
  - Folders

```python
def process_directory(directory):
    for item in directory:
        if is_file(item):
            process_file(item)
        else:
            process_directory(item)
```
The Fibonacci Sequence

0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89...

\[ F_0 = 0, \quad F_1 = 1 \]
\[ F_n = F_{(n-1)} + F_{(n-2)} \]
Golden Spirals Occur in Nature

GO BEARS
Fibonacci Code

fibonacci(n) = fibonacci(n-1) + fibonacci(n-2)
    where fibonacci(1) == 1 and fibonacci(0) == 0

def fib(n):
    """
    >>> fib(5)
    5
    """
    if n < 2:
        return n
    return fib(n - 1) + fib(n - 2)
Visualizing Fib Recursion:

Interactive View
Counting Change

• **Problem Statement:**
• Given (an infinite number of) coins, (25¢, 10¢, etc) how many different ways can I represent 50¢?
  – e.g. 5¢ can be made 2 ways: 1 nickel, or 5 pennies
  – 10¢ can be made 4 ways: [1x 10¢, 2x 5¢, 1 5¢ + 5 1¢, 10x 1¢]
  – Order doesn’t matter, 5¢ + 5 1¢ is the same as 5 1¢ + 5¢

• **How do we solve this?**
Counting Change

• change for 25¢ using [25, 10, 5] → 4

• What do we return?
  • 1 if valid count
  • 0 otherwise

• What are possible “smaller” problems?
  • Smaller amount of money → use coin
  • Fewer coins → “discard” coin

• What is our base case?
  – valid count: value is 0
  – invalid count: value is < 0, or no coins left

• Recursion:
  • Divide: split into two problems (smaller amount & fewer coins)
  • Combine: addition (# of ways)
def count_change(value, coins):
    """
    >>> denominations = [50, 25, 10, 5, 1]
    >>> count_change(7, denominations)
    2
    """
    if value < 0 or len(coins) == 0:
        return 0
    elif value == 0:
        return 1
    using_coin = count_change(value - coins[0], coins)
    not_using_coin = count_change(value, coins[1:])
    return using_coin + not_using_coin
Visualizing Count Change

- Interactive view
What’s the point?

• Explore of problem like `count_partitions`
• Many tree recursive questions follow a similar step
  – Notice how instead of a conditional, we combine the results of two possible options
Bonus: Quicksort
Quicksort

• A fairly simple to sorting algorithm
• Goal: Sort the list by breaking it into partially sorted parts
  – Pick a “pivot”, a starting item to split the list
  – Remove the pivot from your list
  – Split the list into 2 parts, a smaller part and a bigger part
  – Then recursively sort the smaller and bigger parts
  – Combine everything together: the smaller list, the pivot, then the bigger list
QuickSort Example

[3, 3, 1, 4, 5, 4, 3, 2, 1, 17]

[3, 1, 3, 2, 1]  [4, 5, 4, 17]

[1, 3, 2, 1]  []  [4]  [5, 17]


[1]  [1]  [2, 3]  [4, 4, 5, 17]

[1, 1, 2, 3]  [4, 4, 5, 17]

[1, 1, 2, 3, 3]  [1, 1, 2, 3, 3, 3, 4, 4, 5, 17]
Tree Recursion

• Break the problem into multiple smaller sub-problems, and Solve them recursively

```
def split(x, s):
    return [i for i in s if i <= x], [i for i in s if i > x]

def quicksort(s):
    """Sort a sequence - split it by the first element, sort both parts and put them back together.""
    if not s:
        return []
    else:
        pivot = s[0]
        smaller, bigger = split(pivot, s[1:])
        return quicksort(smaller) + [pivot] + quicksort(bigger)

>>> quicksort([3,3,1,4,5,4,3,2,1,17])
[1, 1, 2, 3, 3, 3, 4, 4, 5, 17]
```
Quicksort Visualization

- Interactive View