Programming for Engineers

Lecture 7 - Trees and Recursion

Radoslav Škoviera

Table of contents

Recursion

Introduction to Recursion

Recursion is a programming technique where a function calls itself. It can be used to solve problems that can be broken down into smaller sub-problems. The function calls itself until a base case (trivial case) is reached, at which point the function returns a value and "retraces its steps" to reach the original call.

Simple example:

```
def count_down(n):
    if n == 0:  # base case
        return
    print(n)
    count_down(n-1)  # recursive call

count_down(5)
```

```
5
4
3
2
1
```

Simple example with return value:

```
def sum_numbers(n):
    if n == 0:  # base case
        return 0
    return n + sum_numbers(n-1)  # recursive call

print(sum_numbers(5))
```

```
15
```

Roll-out of calls:

```
sum_numbers(5)
5 + sum_numbers(4)
5 + 4 + sum_numbers(3)
5 + 4 + 3 + sum_numbers(2)
5 + 4 + 3 + 2 + sum_numbers(1)
5 + 4 + 3 + 2 + 1 + sum_numbers(0)  # base case returns 0
5 + 4 + 3 + 2 + 1 + 0  # now everything is computed backwards
5 + 4 + 3 + 2 + 1
5 + 4 + 3 + 3
5 + 4 + 6
5 + 10
15
```

Or, we can actually print the roll-out with a recursive function:

```python
def print_numbers(n, s=""):
    if n == 0:  # base case
        return 0
    print(f"{s}{n} + sum_numbers({n-1})")
    v = print_numbers(n-1, s + f"{n} + ")
    print(f"{s}{n} + {v}")
    return n + v

print("print_numbers(5)")  # the first call is "external"
print_numbers(5)
```

```
print_numbers(5)
5 + sum_numbers(4)
5 + 4 + sum_numbers(3)
5 + 4 + 3 + sum_numbers(2)
5 + 4 + 3 + 2 + sum_numbers(1)
5 + 4 + 3 + 2 + 1 + sum_numbers(0)
5 + 4 + 3 + 2 + 1 + 0
5 + 4 + 3 + 2 + 1
5 + 4 + 3 + 3
5 + 4 + 6
5 + 10


15
```

Order of recursion:

```python
def print_sum_numbers(n):
    print(f"visiting {n}")
    if n == 0:  # base case
        print(0)
        return "0"
    print(str(n), "+", print_sum_numbers(n-1))
    return str(n)

print_sum_numbers(5)
```

```
visiting 5
visiting 4
visiting 3
visiting 2
visiting 1
visiting 0
0
1 + 0
2 + 1
3 + 2
4 + 3
5 + 4

'5'
```

Somewhat useful example: Factorial

```python
def factorial(n):
    if n == 0:
        print("Returning 1")
        return 1
    print(f"Computing {n} * factorial({n-1})")
    return n * factorial(n - 1)

print(factorial(5))
```

```
Computing 5 * factorial(4)
Computing 4 * factorial(3)
Computing 3 * factorial(2)
Computing 2 * factorial(1)
```

```
Computing 1 * factorial(0)
Returning 1
120
```

Types of Recursion

Tail vs. Non-Tail Recursion

- **Tail Recursion:** Recursion where the recursive call is the last expression in the function.
- **Non-Tail Recursion:** Recursion where the recursive call is not the last expression in the function.

Tail Recursion

```python
def count_down(n):
    if n == 0:
        return
    print(n)
    # recursion is the last expression in the function
    count_down(n-1)

count_down(5)
```

```
5
4
3
2
1
```

Non-Tail Recursion

```python
def count_down(n):
    if n == 0:
        return
    count_down(n-1)
    print(n)  # print after the recursive call

count_down(5)
```

```
1
2
3
4
5
```

Direct Recursion vs. Indirect Recursion

All above cases are direct recursion - the function calls (only) itself. Indirect recursion is when the function calls another function that calls the original function.

```python
def is_even(n):
    print(f"is {n} even?")
    return n == 0 or is_odd(n - 1)

def is_odd(n):
    print(f"is {n} odd?")
    return n != 0 and is_even(n - 1)

print(is_even(5))
print("---")
print(is_odd(5))
```

```
is 5 even?
is 4 odd?
is 3 even?
is 2 odd?
is 1 even?
is 0 odd?
False
---
is 5 odd?
is 4 even?
is 3 odd?
is 2 even?
is 1 odd?
is 0 even?
True
```

Yes, multiple indirect recursion is also possible but let's not do that.

Nested Recursion

Nested recursion is when a function calls itself inside a recursive call - the inner recursive call needs to return before the outer recursive call can continue.

```python
def ackermann(m, n):
    if m == 0:
        print(f"{n} + 1")
        return n + 1
    elif n == 0:
        print(f"ackermann({m} - 1, 1)")
        return ackermann(m - 1, 1)
    else:
        print(f"ackermann({m} - 1, ackermann({m}, {n} - 1))  # nested")
        return ackermann(m - 1, ackermann(m, n - 1))  # Nested call

print(ackermann(1, 2))
```

```
ackermann(1 - 1, ackermann(1, 2 - 1))  # nested
ackermann(1 - 1, ackermann(1, 1 - 1))  # nested
ackermann(1 - 1, 1)
1 + 1
2 + 1
3 + 1
4
```

Linear vs. Tree (Branched) Recursion

- **Linear Recursion:** Recursion where the function calls itself only once at each step.
- **Tree Recursion:** Recursion where the function calls itself multiple times during a single step (but not nested).

Fibonacci is a typical example of a simple tree recursion.

```python
def fibonacci(n):
    print(f"fibonacci({n})")
    if n <= 1:
        return n
    return fibonacci(n-1) + fibonacci(n-2)

print(fibonacci(5))
```

```
fibonacci(5)
fibonacci(4)
fibonacci(3)
fibonacci(2)
fibonacci(1)
fibonacci(0)
fibonacci(1)
fibonacci(2)
fibonacci(1)
fibonacci(0)
fibonacci(3)
fibonacci(2)
fibonacci(1)
fibonacci(0)
fibonacci(1)
5
```

However, tree recursions are mostly useful to traverse trees or graphs in general (or grids, which are basically also graphs). Be careful as tree recursions can easily "explode", consuming too much memory.

Recursion vs. Loops

Most things done recursively can also be done with loops. Let's compare:

Iterative Factorial

```python
def factorial_iterative(n):
    result = 1
    for i in range(1, n + 1):
        result *= i
    return result

print(factorial_iterative(5))
```

```
120
```

Iterative Fibonacci

```
def fib_iterative(n):
    a, b = 0, 1
    for _ in range(n):
        a, b = b, a + b
    return a

print(fib_iterative(10))
```

```
55
```

Issues with Recursion

- **Stack Overflow:** Too many recursive calls can exceed the call stack. There is a way to increase the call stack size in Python but then you risk running out of memory.
- **Performance:** Redundant computations if not optimized, especially for nested or tree recursions (e.g., naive Fibonacci).
- **Mitigation:** Use **memoization** or **tail recursion** (Python doesn't natively optimize tail recursion).
- **Python frames** Python, as an interpreted language, has especially bad memory management for recursive calls (each call creates a new "frame").

Python Frames

```
import inspect

def show_frames():
    frame = inspect.currentframe()
    depth = 0
    while frame:
        print(f"Frame {depth}: {frame.f_code.co_name}")
        frame = frame.f_back
        depth += 1

def factorial(n):
    if n == 0:
        show_frames()
```

```
        return 1
    return n * factorial(n - 1)

factorial(5)
```

```
Frame 0: show_frames
Frame 1: factorial
Frame 2: factorial
Frame 3: factorial
Frame 4: factorial
Frame 5: factorial
Frame 6: factorial
Frame 7: <module>
Frame 8: run_code
Frame 9: run_ast_nodes
Frame 10: run_cell_async
Frame 11: _pseudo_sync_runner
Frame 12: _run_cell
Frame 13: run_cell
Frame 14: run_cell
Frame 15: do_execute
Frame 16: execute_request
Frame 17: execute_request
Frame 18: dispatch_shell
Frame 19: process_one
Frame 20: dispatch_queue
Frame 21: _run
Frame 22: _run_once
Frame 23: run_forever
Frame 24: start
Frame 25: start
Frame 26: launch_instance
Frame 27: <module>
Frame 28: _run_code
Frame 29: _run_module_as_main

120
```

Memoization

Memoization is a technique to store the results of expensive function calls and reuse them when the same input is encountered again. This can be used to cache results in recursion.

```
cache = {}

def fib_memo(n):
    if n in cache:
        return cache[n]
    if n <= 1:
        return n
    cache[n] = fib_memo(n-1) + fib_memo(n-2)
    return cache[n]

def fib_normal(n):
    if n <= 1:
        return n
    return fib_normal(n-1) + fib_normal(n-2)

print("Fibonacci runtime without memoization:")
%timeit fib_normal(10)
print("Fibonacci runtime with memoization:")
%timeit fib_memo(10)
```

```
Fibonacci runtime without memoization:
7.1  s ± 266 ns per loop (mean ± std. dev. of 7 runs, 10,000 loops each)
Fibonacci runtime with memoization:
57.3 ns ± 2.01 ns per loop (mean ± std. dev. of 7 runs, 10,000,000 loops each)
```

Alternatively, use `functools.lru_cache` decorator.

```
from functools import lru_cache

@lru_cache(maxsize=None)
def fib_memo(n):
    if n <= 1:
        return n
    return fib_memo(n-1) + fib_memo(n-2)

%timeit fib_memo(10)
```

```
48.2 ns ± 1.59 ns per loop (mean ± std. dev. of 7 runs, 10,000,000 loops each)
```

Uses of Recursion

General rule is if you can do it nicely with a loop, do it with a loop. There are, however, some problems that are better done recursively.

For example: - tree and graph traversals (search) - divide-and-conquer algorithms (merge sort, quick sort) - backtracking (e.g., maze solving)

Trees

Introduction to Trees

A tree is a non-linear data structure composed of nodes connected by edges. One node is the root, and others branch off. It is basically a directed acyclic graph (DAG).

Binary Tree

The following requires `graphviz` to be installed: https://graphviz.org/

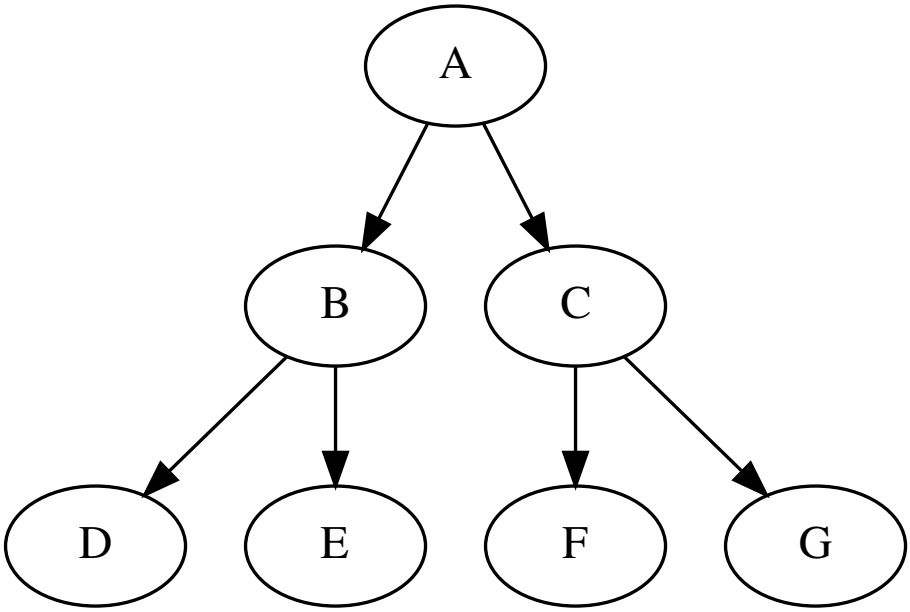
```
!pip install graphviz
```

```
from graphviz import Digraph

def draw_simple_tree():
    tree = Digraph(format='pdf')
    tree.node('A')
    tree.node('B')
    tree.node('C')
    tree.node('D')
    tree.node('E')
    tree.node('F')
    tree.node('G')
    tree.edge('A', 'B')
    tree.edge('A', 'C')
    tree.edge('B', 'D')
    tree.edge('B', 'E')
    tree.edge('C', 'F')
    tree.edge('C', 'G')
```

```
    return tree

draw_simple_tree()
```



There are many different types of trees - they don't have to be binary.

Binary Search Tree

```
class Node:
    def __init__(self, key):
        self.left = None
        self.right = None
        self.val = key

# Insert and Inorder Traversal

def insert(root, key):
    if root is None:
        return Node(key)
    if key < root.val:
        root.left = insert(root.left, key)
    else:
```

```
        root.right = insert(root.right, key)
    return root

def inorder(root):
    if root:
        inorder(root.left)
        print(root.val, end=' ')
        inorder(root.right)

r = Node(50)
insert(r, 30)
insert(r, 20)
insert(r, 40)
insert(r, 70)
insert(r, 60)
insert(r, 80)

inorder(r)
```

```
20 30 40 50 60 70 80
```

More on trees next time.

Trees and Recursion

Tree Traversals

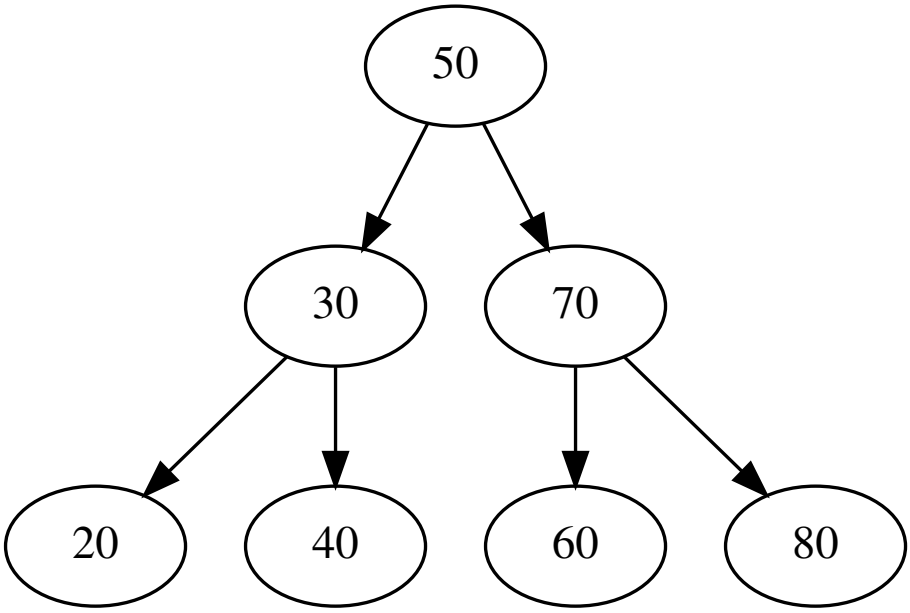
Trees are inherently recursive data structures - recursion is a natural way to traverse them. Most operations on trees would be very difficult to do with loops.

Example problem: Drawing a Tree

```
def draw_tree(node, tree=Digraph(format='pdf')):
    if node:
        tree.node(str(node.val))
        if node.left:
```

```
            tree.edge(str(node.val), str(node.left.val))
            draw_tree(node.left, tree)
        if node.right:
            tree.edge(str(node.val), str(node.right.val))
            draw_tree(node.right, tree)
    return tree

draw_tree(r)
```



Example problem: Height of a Tree

```
def height(root):
    if root is None:
        return 0
    return 1 + max(height(root.left), height(root.right))

height(r)
```

```
3
```

Problem: Sum of All Nodes

```
def sum_nodes(root):
    if root is None:
        return 0
    return root.val + sum_nodes(root.left) + sum_nodes(root.right)

sum_nodes(r)
```

```
350
```