Class 6 - Recursion

MISE Summer Programming Camp 2023

Intro

Viknesh ("Vik") Krishnan

Software Engineer @ Google

BS in Mathematics & Computer Science from UMichigan

Recap of Class 5

- Lists as “references” (will review again later)
  - A list variable “refers” to an actual list
  - Two variables can point to the same actual list
- 2 dimensional / multidimensional lists
  - Lists of lists to record tables of data
- Extra features of lists
  - List comprehension
  - List slicing

Recursion

Simple Recursive Functions: Example 1

Let us consider the problem of **adding all the numbers in a list**.

If we implement a iterative (i.e. nonrecursive) solution for the problem, it would look like this:

```
def listSum(nums):
    numsum = 0
    for num in nums:
        numsum += num
    return numsum
```

How would a recursive version of this function look like?

Simple Recursive Functions: Example 1

A recursive solution for the problem would look like:

```
def recListSum(nums):
    if len(nums) == 0:
        return 0
    return nums[0] + recListSum(nums[1:])
```

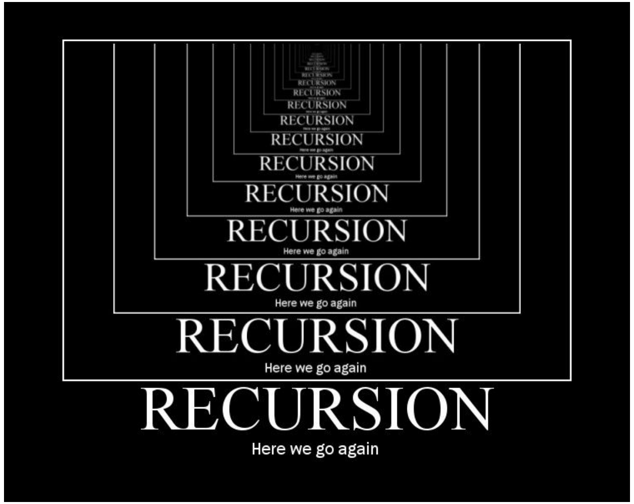
List slicing: all elements of *nums* except the first one

Recursion: What is it?

A problem solving approach where we break a problem into smaller versions of the same problem.

Technically, we can think of recursion as being a **function that calls itself**.

However, in reality, it turns out to be a powerful way to solve problems.



Recursion: What is it?

We often divide a recursive function in two parts:

- A **base case**: returns a result for a known value;
- A **recursive case**: computes a result calling the same function for a different value.

In other words, with recursion, we solve a problem by assuming it is already solved :)

Recursion: Code example

A template for simple recursive functions can be achieved as follows:

```
def recursiveTemplate(value):
    if baseCase == True:
        return knownValue
    else:
        return recursiveTemplate(modify(value))
```

itempool.com/mise23/live

Pop Quiz 1:

What is the output of the following code:

```python
def f(x):
    if x == 0:
        return 0
    return 1 + f(x - 1)
print(f(5))
```

pythontutor

Pop Quiz 2:

What is the output of the following code:

```python
def f(a, b):
    if b == 0:
        return 1
    return a * f(a, b - 1)

print(f(3, 2))
```

pythontutor

On the previous example:

The previous pop quiz is a function that computes the power of a number!

Here is a better code:

```
def recPower(base,exponent):
    if exponent == 0:
        return 1
    return base * recPower(base,exponent-1)
```

Challenge: Fibonacci!

Now let us consider the problem of computing the nth Fibonacci number.

The Fibonacci numbers are defined as follows:

$$F_0 = 0, \quad F_1 = 1, \qquad F_n = F_{n-1} + F_{n-2}$$

So,

$F_2 = F_0 + F_1 = 1$

$F_3 = F_2 + F_1 = 2$

$3, 5, 8, \ldots$

Challenge: Fibonacci!

Now let us consider the problem of computing the nth Fibonacci number.

The Fibonacci numbers are defined as follows:

$$F_0 = 0, \quad F_1 = 1, \qquad F_n = F_{n-1} + F_{n-2}$$

Let's try solving this problem two different ways, using **iteration** and using **recursion**.

Fibonacci: Solutions

Iterative Solution:

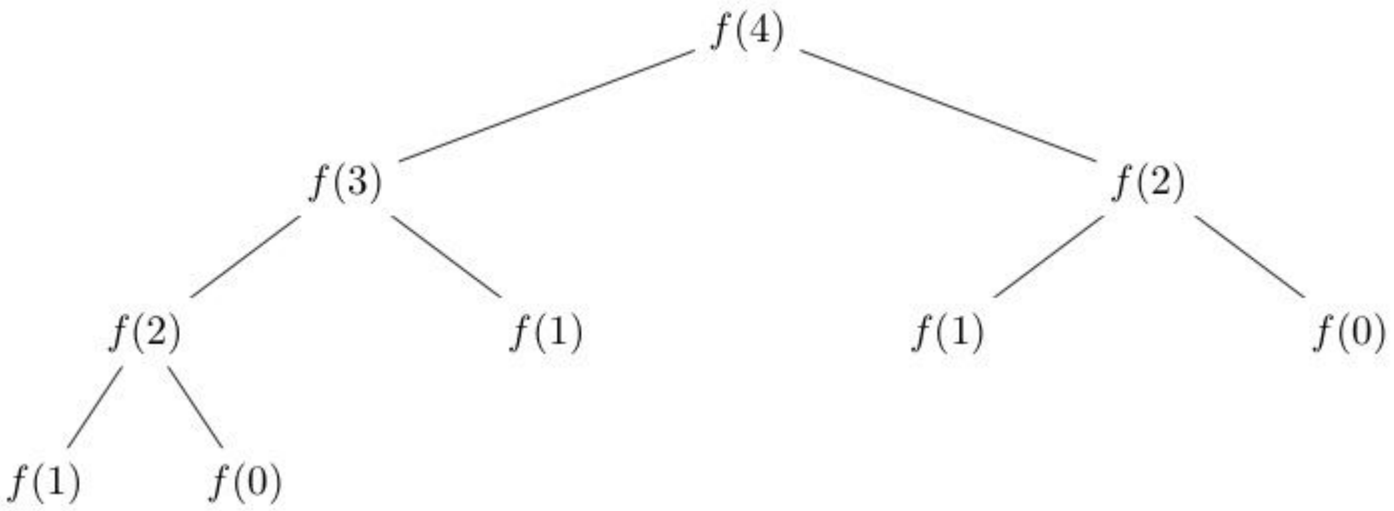
```python
def iterative_fibonacci(n):
    if n <= 1:
        return n
    f_n_2 = 0
    f_n_1 = 1
    for i in range(n - 1):
        f = f_n_2 + f_n_1
        f_n_2, f_n_1 = f_n_1, f
    return f_n_1
```

Recursive Solution:

```python
def recursive_fibonacci(n):
    if n <= 1:
        return n
    return recursive_fibonacci(n - 1) + recursive_fibonacci(n - 2)
```

Visualize this in Python Tutor!

Recursion tree



Pop Quiz 3:

Which of the following mimics what the *range()* function does:

```
def my_range1(n):
    if n == 1:
        return []
    return my_range1(n - 1) + [n - 1]
```

```
def my_range2(n):
    if n == 1:
        return []
    return my_range2(n) + [n]
```

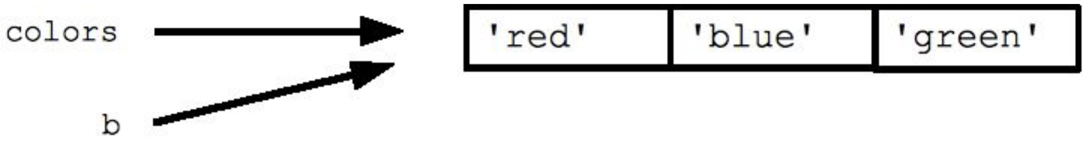
```
def my_range3(n):
    if n == 0:
        return []
    return my_range3(n - 1) + [n - 1]
```

```
def my_range4(n):
    if n <= 0:
        return []
    result = my_range4(n - 1)
    result.append(n)
    return result
```

Backtracking

Review: List References

```
colors = ["red", "blue", "green"]
b = colors
```



References are essentially pointers that allow variables to refer to an actual list

```
def f(l):
    l[0] = 5
    print(l)

l = [1, 2, 3, 4, 5]
f(l)
print(l)
```

(reference)

```
[5, 2, 3, 4, 5]
[5, 2, 3, 4, 5]
```

[pythontutor]

```
def f(l):
    l = 5
    print(l)

l = 1
f(l)
print(l)
```

(not reference)

```
5
1
```

What is backtracking?

Strategy where we enumerate all possible solutions to a problem by **incrementally building** candidates to solutions

Very useful to find solutions to combinatorial problems (we'll see examples)

Generating all DNA strings of length n

```python
def gen_strs(n):
    if n == 0:
        return ['']

    sol = []
    partial = gen_strs(n - 1)
    for base in ['A', 'C', 'G', 'T']:
        for dna in partial:
            sol.append(base + dna)
    return sol
```

Alternate solution using backtracking

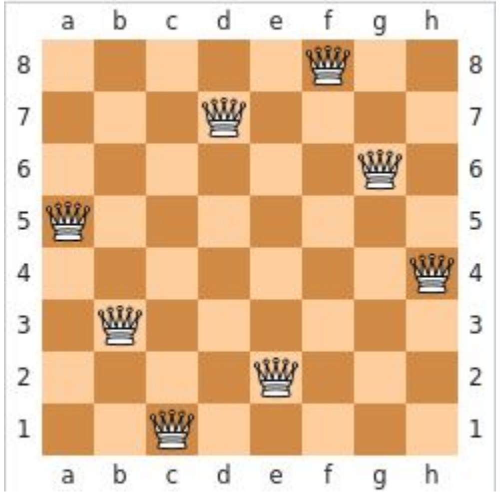
```
def gen_strs(current, n, sol):
    if n == 0:
        sol.append(current)
        return

    for base in ['A', 'C', 'G', 'T']:
        gen_strs(base + current, n - 1, sol)
```

Notice how we build partial solutions (the parameter ‘current’) incrementally

Counting problems: the n-queens problem

Consider a **n** by **n** chessboard where we want to place **n** queens such that they don't attack other (example on the right)

How many different ways are there to do so?

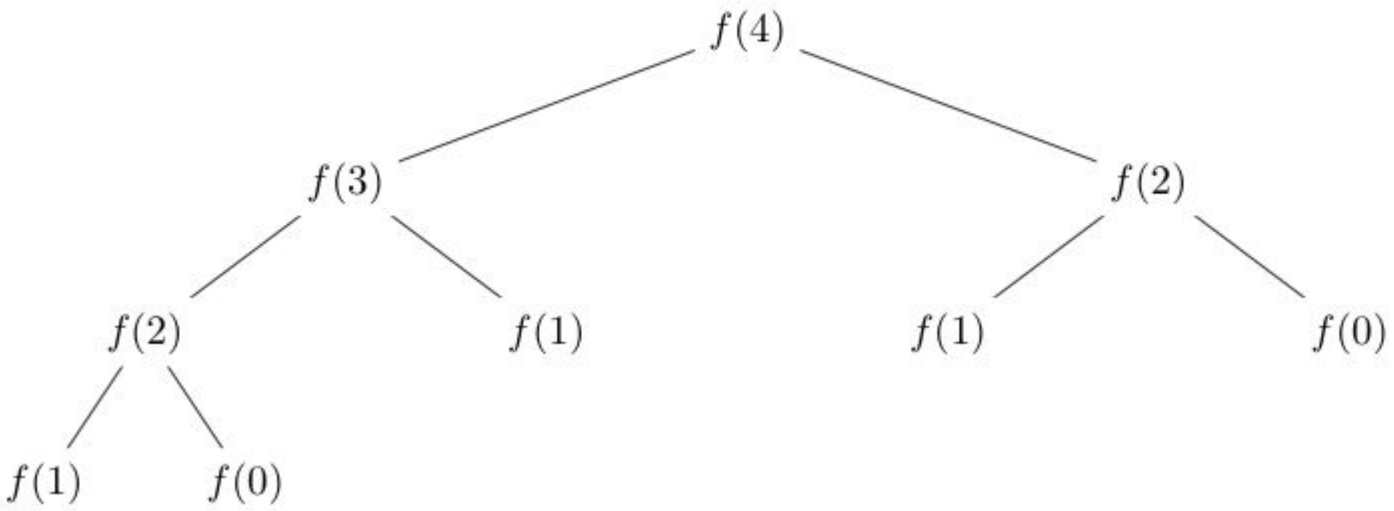


```python
def solve(board, placed):
    n = len(board)
    if placed == n:
        return 1

    ct = 0
    for i in range(n):
        if isSafe(board, i, placed):
            board[i][placed] = 1
            ct += solve(board, placed + 1)
            board[i][placed] = 0
    return ct

n = 8
board = [[0 for j in range(n)] for i in range(n)]
print(solve(board, 0))
```

```python
def isSafe(board, row, col):
    for i in range(col):
        if board[row][i] == 1:
            return False
    for i, j in zip(range(row, -1, -1), range(col, -1, -1)):
        if board[i][j] == 1:
            return False
    for i, j in zip(range(row, len(board), 1), range(col, -1, -1)):
        if board[i][j] == 1:
            return False
    return True
```

Dynamic Programming

Avoiding repeating actions



Avoiding repeating actions

- When we write recursive code we subdivide a problem into smaller subproblems
- Often there are a lot of repeated subproblems (like in the previous example)
- We can avoid having to recompute the solution to subproblems by storing it
- This is called Dynamic Programming

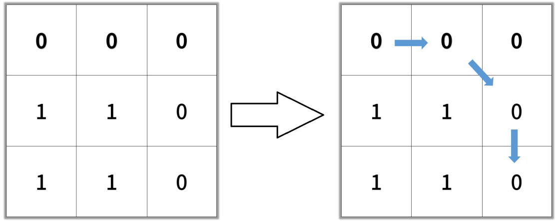
Back to the Fibonacci example

```python
dp = [-1 for i in range(10)]

def fib(n, dp):
    if dp[n] != -1:
        return dp[n]
    if n <= 1:
        dp[n] = n
    else:
        dp[n] = fib(n - 1, dp) + fib(n - 2, dp)
    return dp[n]

print(fib(9, dp))
print(dp)
```

```
34
[0, 1, 1, 2, 3, 5, 8, 13, 21, 34]
```

Challenge - Paths on a grid

- Let's suppose we have a **n** by **n** grid with integers
- We start at the top left corner of the grid and we want to go to the lower right
- We can move down or to the right
- Everytime we step on a grid cell we pay a cost equal to the cell's value
- What is the minimum cost path?



Complete the following

```python
def path(grid, row, col):
    n = len(grid)
    if row == n and col == n:
        return grid[row - 1][col - 1]

    return grid[row - 1][col - 1] + min(path(???), path(???))
```

Solution

```python
def path(grid, row, col):
    n = len(grid)
    if row == n and col == n:
        return grid[row - 1][col - 1]

    sol = 10000000000
    if row < n:
        sol = min(sol, path(grid, row + 1, col))
    if col < n:
        sol = min(sol, path(grid, row, col + 1))
    return sol + grid[row - 1][col - 1]
```

this is correct, but … what’s the problem with code?

How do we store repeated computation?

```python
dp = [[-1 for i in range(n)] for j in range(n)]

def path(grid, row, col):
    n = len(grid)
    if row == n and col == n:
        return grid[row - 1][col - 1]
    if dp[row - 1][col - 1] != -1:
        return dp[row - 1][col - 1]

    sol = 10000000000
    if row < n:
        sol = min(sol, path(grid, row + 1, col))
    if col < n:
        sol = min(sol, path(grid, row, col + 1))
    dp[row - 1][col - 1] = sol + grid[row - 1][col - 1]
    return dp[row - 1][col - 1]
```