Recursion

Thomas Schwarz

Recursion

- A function that calls itself
  - Example: the Fibonacci numbers
    - $f_0 = 0, \quad f_1 = 1, \quad f_n = f_{n-1} + f_{n-2}$

Recursion

- Implementation

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

Recursion

- In order to calculate f_7
  - Need f_6 and f_5
  - Need f_5, f_4 and f_4, f_3
    - Need f_4, f_3, f_3, f_2 and $f_3, f_2, f_2, f_1 = 1$
    - …
- So, we calculate the same value many times.

Memoization

- A simple trick to speed up recursive functions is to remember values that we have already calculated.
- Create a dictionary (possibly global) that stores values already calculated
    - Before any calculation check whether the desired value is in the dictionary
    - If we calculate something, we put the value into the dictionary

Memoization

```
fdic={0: 0, 1:1}

def fibonacci2(number):
    if number in fdic:
        return fdic[number]
    else:
        retval = fibonacci2(number-1)+fibonacci2(number-2)
        fdic[number] = retval
        return retval

for i in range(41):
    measure(fibonacci2, i*50)
```

Memoization

```
fdic={0: 0, 1:1}

def fibonacci2(number):
    if number in fdic:
        return fdic[number]
    else:
        retval = fibonacci2(number-1)+fibonacci2(number-2)
        fdic[number] = retval
        return retval

for i in range(41):
    measure(fibonacci2, i*50)
```

- **Defining the dictionary**

Memoization

```
fdic={0: 0, 1:1}

def fibonacci2(number):
    if number in fdic:
        return fdic[number]
    else:
        retval = fibonacci2(number-1)+fibonacci2(number-2)
        fdic[number] = retval
        return retval

for i in range(41):
    measure(fibonacci2, i*50)
```

- Check whether value is in the dictionary

Memoization

```
fdic={0: 0, 1:1}

def fibonacci2(number):
    if number in fdic:
        return fdic[number]
    else:
        retval = fibonacci2(number-1)+fibonacci2(number-2)
        fdic[number] = retval
        return retval

for i in range(41):
    measure(fibonacci2, i*50)
```

- **Calculation is necessary**

Memoization

```
fdic={0: 0, 1:1}

def fibonacci2(number):
    if number in fdic:
        return fdic[number]
    else:
        retval = fibonacci2(number-1)+fibonacci2(number-2)
        fdic[number] = retval
        return retval

for i in range(41):
    measure(fibonacci2, i*50)
```

- But we store the result in the dictionary in case we use it in the future

Memoization

```
fdic={0: 0, 1:1}

def fibonacci2(number):
    if number in fdic:
        return fdic[number]
    else:
        retval = fibonacci2(number-1)+fibonacci2(number-2)
        fdic[number] = retval
        return retval

for i in range(41):
    measure(fibonacci2, i*50)
```

- And now we measure

Decorators

- Python uses decorators to allow changing functions
- A decorator is implemented by:
  - Creating a function of a function that returns the amended function

Decorators

```
def timeit(function):
    def clocked(*args):
        start_time = time.perf_counter()
        result = function(*args)
        duration = (time.perf_counter() - start_time)
        name = function.__name__
    arg_string = ', '.join(repr(arg) for arg in args)
    print('Function {} with arguments {} ran
           in {} seconds'.format(
        name, arg_string, duration))
    return result
return clocked
```

Decorators

- Decorator takes a function with positional arguments as function
- Decorator defines a new version of the argument function
- And returns it.

Decorators

```
def timeit(function):
    def clocked(*args):
        start_time = time.perf_counter()
        result = function(*args)
        duration = (time.perf_counter() - start_time)
        name = function.__name__
    arg_string = ', '.join(repr(arg) for arg in args)
    print('Function {} with arguments {} ran
           in {} seconds'.format(
        name, arg_string, duration))
    return result
return clocked
```

Decorators

- To use a decorator, just put its name on top of the function definition
  - Decorator generator is executed when module is imported (or generator is defined)
  - When decorated function is defined, the modified version is created

Decorators

```
@timeit
def fibonacci(n):
    if n == 0:
        return 0
    if n == 1:
        return 1
    else:
        return fibonacci(n-1)+fibonacci(n-2)
```

Decorators

- If we execute this function, we get to see how often fibonacci is called on arguments already executed

```
>>> fibonacci(10)
Function fibonacci with arguments 1 ran in 5.140000070014139e-07 seconds
Function fibonacci with arguments 0 ran in 1.0870000011209413e-06 seconds
Function fibonacci with arguments 2 ran in 0.1692790839999958 seconds
Function fibonacci with arguments 1 ran in 1.2330000060956081e-06 seconds
Function fibonacci with arguments 3 ran in 0.2676633440000131 seconds
Function fibonacci with arguments 1 ran in 9.8000001003129e-07 seconds
Function fibonacci with arguments 0 ran in 1.0470000120221812e-06 seconds
Function fibonacci with arguments 2 ran in 0.09880945999999824 seconds
Function fibonacci with arguments 4 ran in 0.4692909440000079 seconds
Function fibonacci with arguments 1 ran in 6.51999997103303e-07 seconds
Function fibonacci with arguments 0 ran in 1.0500000087176886e-06 seconds
Function fibonacci with arguments 2 ran in 0.11281222700000626 seconds
Function fibonacci with arguments 1 ran in 1.958000012791672e-06 seconds
Function fibonacci with arguments 3 ran in 0.21685028000000273 seconds
Function fibonacci with arguments 5 ran in 0.7868284680000102 seconds
Function fibonacci with arguments 1 ran in 5.699999742402e-07 seconds
Function fibonacci with arguments 0 ran in 1.0729999928571488e-06 seconds
Function fibonacci with arguments 2 ran in 0.11366798399998856 seconds
Function fibonacci with arguments 1 ran in 1.2930000110600304e-06 seconds
Function fibonacci with arguments 3 ran in 0.2176230820000029 seconds
Function fibonacci with arguments 1 ran in 5.839999914769578e-07 seconds
```

Memoization with lru_cache

- We can define our own memoization decorator
  - But Python has one that uses an LRU cache
    - Memoization is LRU cache with an infinite cache size
  - Import from functools lru_cache

```
@functools.lru_cache
def fib(n):
    if n <= 1:
        return n
    else:
        return fib(n-1)+fib(n-2)
```

Memoization

- A more complicated example:
  - How many stamps are needed to make various postages
  - Assume we have three types of stamps: ones, fours, and fives
  - To lick 27 cents, we can:
    - five 5c + 2 1c stamps: total of 7 stamps
    - four 5c, one 4c and 3 1c stamps: total of 8 stamps
    - three 5c and three 4c stamps: total of six stamps

Memoization

- Given an amount *n* we want to calculate the minimum number of stamps needed
- An inane method: Try out all possibilities

```
def inane(n):
    best_seen = n
    for ones in range(n+1):
        for fours in range(n+1):
            for fives in range(n+1):
                if ones*1+fours*4+fives*5 == n:
                    if ones+fours+fives < best_seen:
                        best_seen = ones+fours+fives
    return best_seen
```

Memoization

- Why is it inane?
  - It takes $n \times n \times n$ iterations to try out all possibilities

Memoization

- A better way?
  - Given an amount *n*, we can try out the best ways to lick *n*-5, *n*-4, and *n*-1, corresponding to deciding to first lick a five cent, a four cent, and a one cent stamp
  - Find the best among those three possibilities and add one to it, because we already licked one stamp
  - $$\texttt{lick}(n) = \min(\begin{cases} 0 & \texttt{if } n = 0 \\ 1 + \texttt{lick}(n-5) & \texttt{if } n \geq 5 \\ 1 + \texttt{lick}(n-4) & \texttt{if } n \geq 4 \\ 1 + \texttt{lick}(n-1) & \texttt{if } n \geq 1 \end{cases})$$

Memoization

- This gives immediately a nice recursive implementation

```
def lick(amount):
    if amount == 0:
        return 0
    if amount < 4:
        return amount
    if amount == 4:
        return 1
    if amount == 5:
        return 1
    return min([lick(amount-1), lick(amount-4), lick(amount-5)])+1
```

Just some special base cases

Memoization

- But this recursive version is very slow
  - That is because the same value `link(x)` can be calculated many times
  - So we put the intermediate results into a dictionary to remember them
  - This is called memoization

Memoization

- Memoization:
  - Maintains a **cache** of intermediate results
  - If the function is called on a value:
    - We first check whether the value is in the cache
    - Otherwise: we calculate it and put it into the cache

Memoization

```
lick_dictionary = {0:0, 1:1, 2:2, 3:3, 4:1, 5:1}

def mlick(amount):
    if amount in lick_dictionary:
        return lick_dictionary[amount]
    else:
        result = min([mlick(amount-1), mlick(amount-4), mlick(amount-5)])+1
        lick_dictionary[amount] = result
        return result
```