Special Topics in Computer Science- CSC 4992

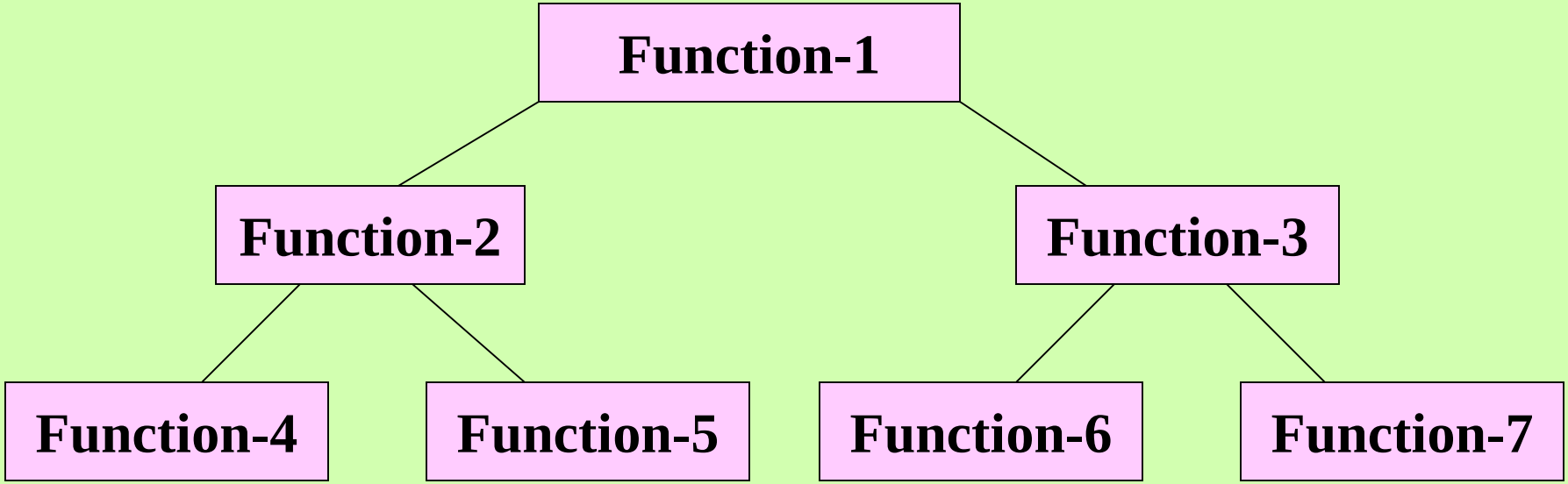
Design with Functions

Why Design?

- As problems become more complex, so do their solutions
- There is more to programming than just hacking out code
- We can decompose a complex problem into simpler subproblems and solve each of these
- Divide up the work and assign responsibilities to individual actors (functions)

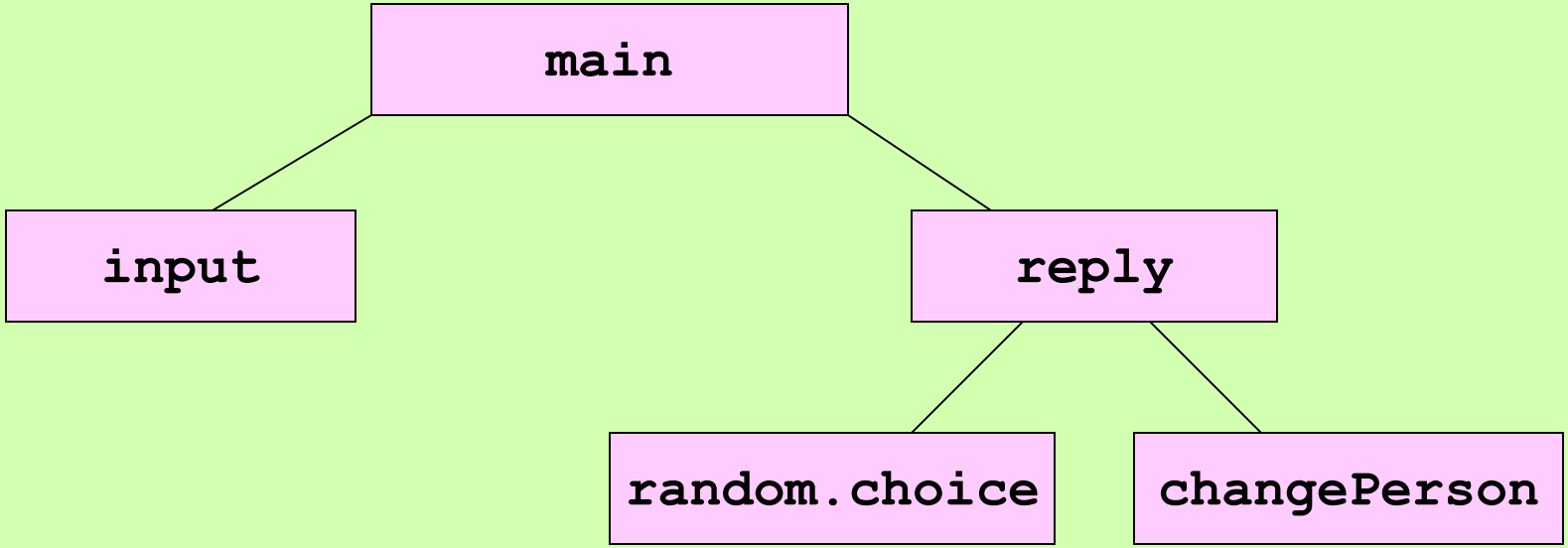
Top-Down Design

In top-down design, we decompose a complex problem into simpler subproblems, and solve these with different functions.



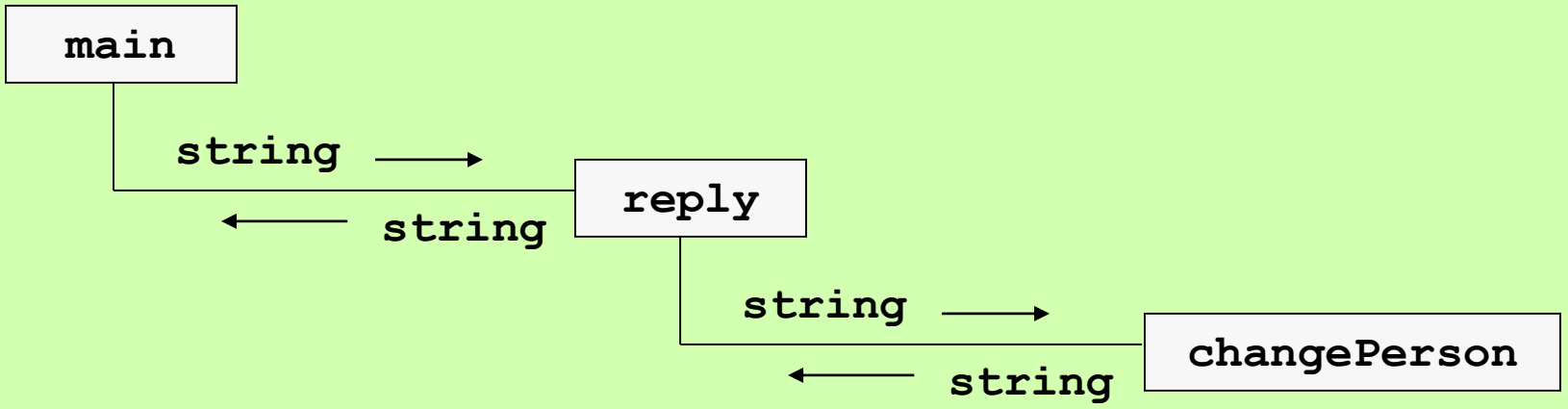
Example: The **`doctor`** Program

Each function has its own responsibilities; in a well-designed program, no function does too much



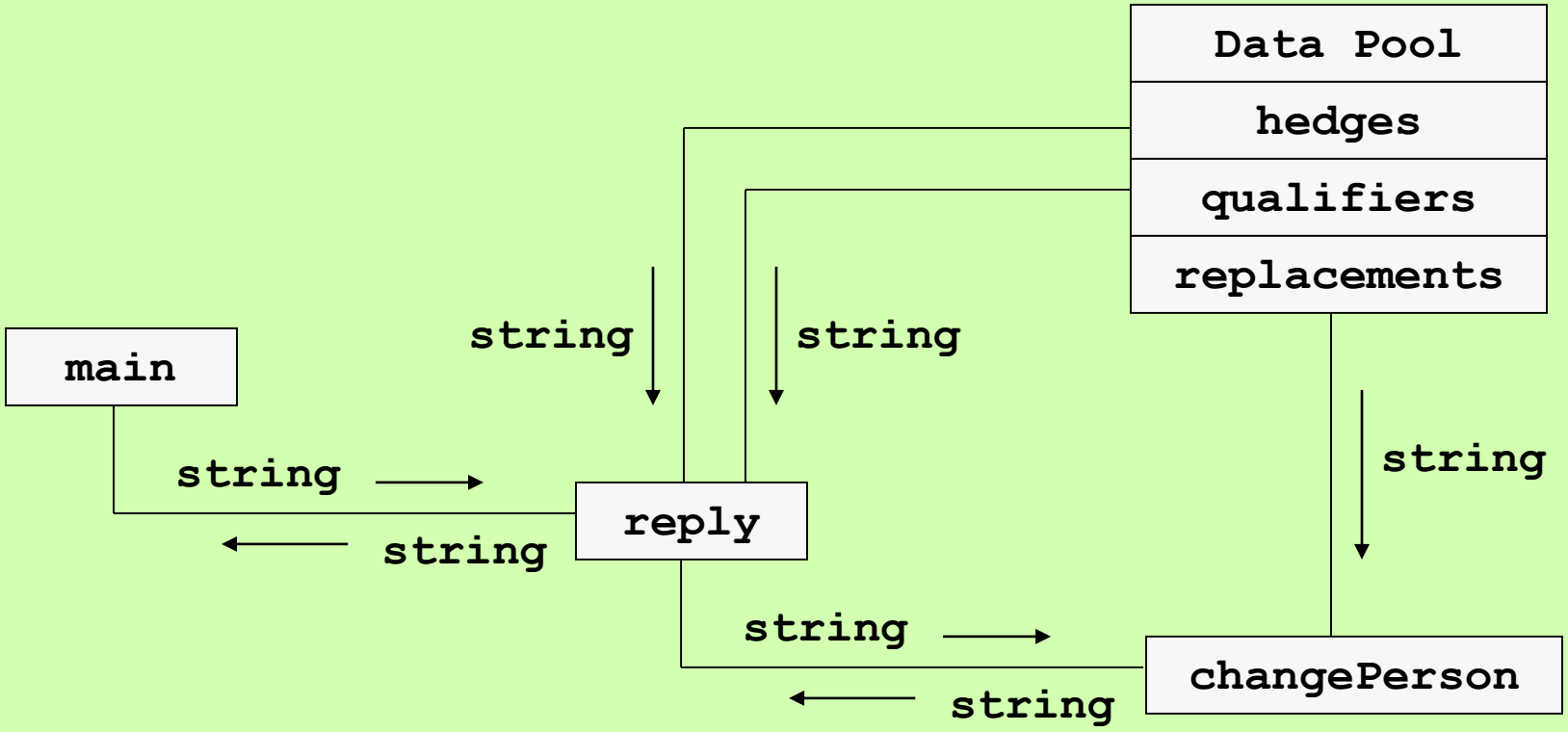
Example: The **`doctor`** Program

Functions communicate via arguments and returned values

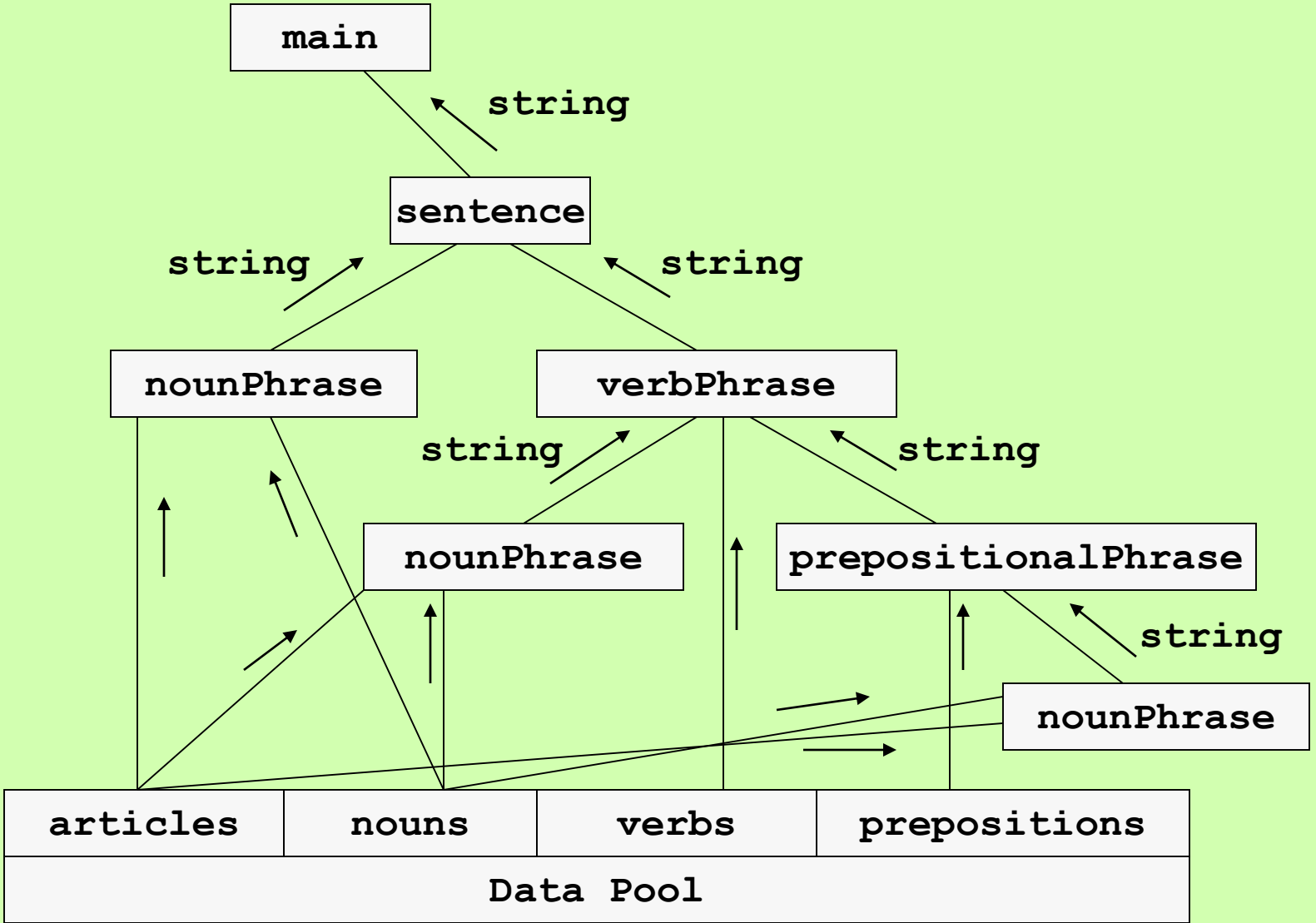


Example: The **doctor** Program

Functions also go to shared data pools for information



Example: The Sentence Program



Design Strategies: Top Down

- Start with the **`main`** function and pretend that the functions that it calls are already defined
- Work your way down by defining those functions, etc.
- Cannot test anything until they're all finished

Drivers and Stubs

- Start with the **`main`** function and pretend that the functions that it calls are already defined
- Define these functions using simple headers and almost no code
  - If a function returns a value, return a reasonable default (0 or empty string)
  - If a function does not return a value, return **`None`**
- The **`main`** function is known as a *driver*, and the other functions are called *stubs*

Skeletal Design

- Drivers/stubs allow you to sketch out the structure of a program in terms of cooperating functions without finishing everything at once
- Set up their communication links, which are the arguments and return values
- Run the program to check these before filling in the details

Design Strategies: Bottom Up

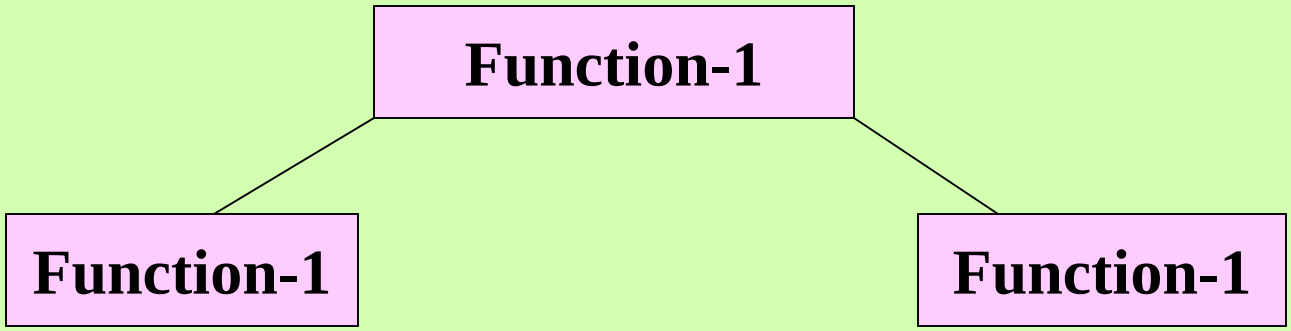
- Start with simple functions at the bottom of the chart and work your way up
- Each new function can be tested as soon as it's defined
- Easier to get the little pieces of a program up and running, then integrate them into a more complete solution

Good Design?

- Do you divide up the work so that each function does one coherent thing?
- Do the functions communicate via arguments and return values rather than a common pool of data?
- When a common pool of data seems necessary, do you confine access to just a few functions?
- Do you name the functions and arguments to reflect their purpose in the program?
- Do you document your design?????

Recursive Design

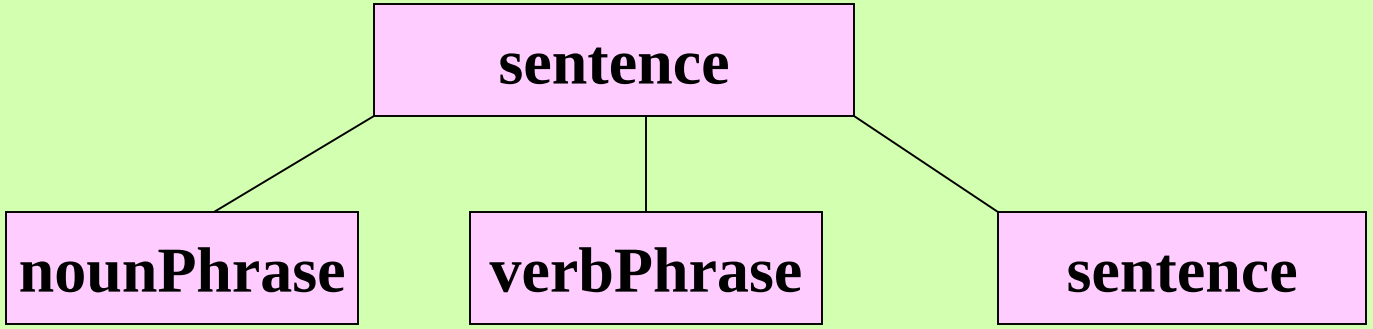
As a special case of top-down design, we decompose a problem into smaller subproblems that have the *same form*, and solve these with the *same function*.



Recursive Design

As a special case, we decompose a problem into smaller subproblems that have the *same form*, and solve these with the *same function*.

sentence = nounPhrase verbPhrase { “and” sentence }



Example: Get a Valid Integer

- The function **`getValidInteger`** prompts the user for an integer and inputs it
- The integer is returned if it is within the specified range
- Otherwise, the function displays an error message and calls **`getValidInteger`** to try again

Define a Recursive Function

```
def getValidInteger(prompt, low, high):
    number = int(input(prompt))
    if number >= low and number <= high:
        return number
    else:
        print('ERROR: Input number is out of range')
        return getValidInteger(prompt, low, high)
```

A *recursive function* calls itself

There will be 0 or more *recursive calls* of this function

A *recursive process* is similar to an iterative process (the same thing is done repeatedly, 0 or more times)

The Base Case

```
def getValidInteger(prompt, low, high):
    number = int(input(prompt))
    if number >= low and number <= high:
        return number
    else:
        print('ERROR: Input number is out of range')
        return getValidInteger(prompt, low, high)
```

A recursive process stops when a *base case* is reached

In this function, a valid input number is simply returned

The Recursive Step

```
def getValidInteger(prompt, low, high):
    number = int(input(prompt))
    if number >= low and number <= high:
        return number
    else:
        print('ERROR: Input number is out of range')
        return getValidInteger(prompt, low, high)
```

Otherwise, a *recursive step* drives the recursion forward, until a base case is reached

Computing Factorial (!)

- 4! = 4 * 3 * 2 * 1 = 24
- N! = N * (N - 1) * (N - 2) * … * 1
- *Recursive definition* of factorial:
  - N! = 1, when N = 1
  - N! = N * (N - 1)!, otherwise

Define `factorial` Recursively

```
# N! = 1, when N = 1
# N! = N * (N - 1)!, otherwise

def factorial(n):
    if n == 1:
        return 1
    else:
        return n * factorial(n - 1)
```

What is the base case?

What is the recursive step?

Does the recursive step advance the process toward the base case?

Tracing `factorial`

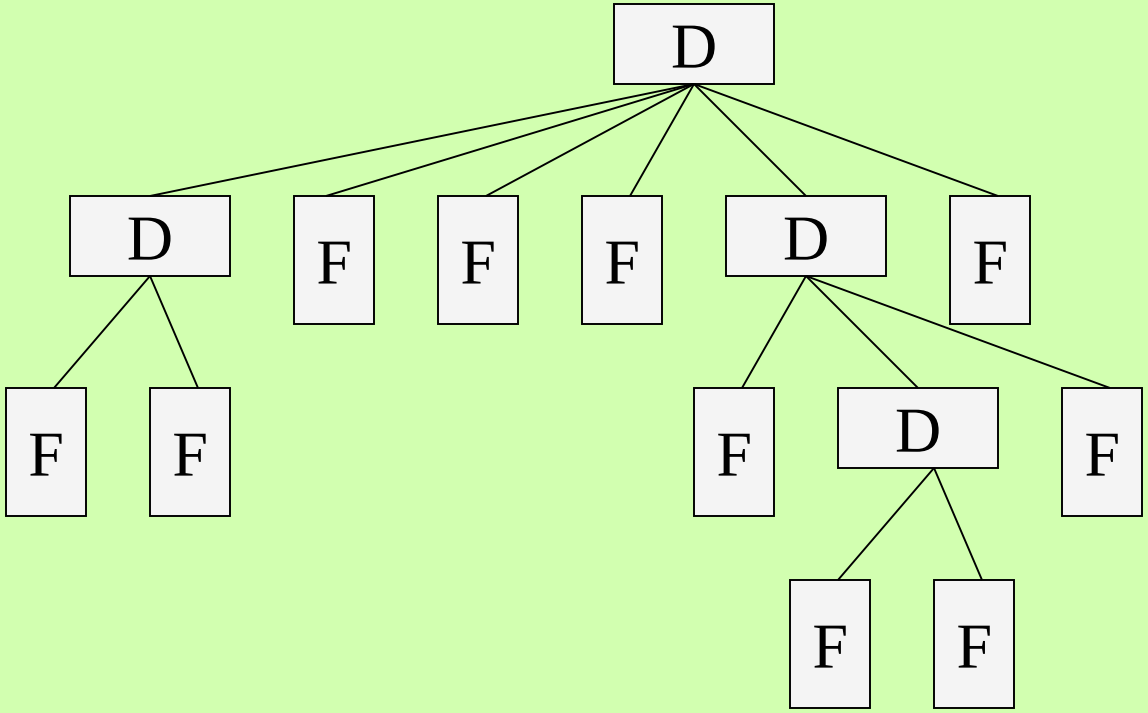
```
# N! = 1, when N = 1
# N! = N * (N - 1)!, otherwise

def factorial(n):
    if n == 1:
        return 1
    else:
        return n * factorial(n - 1)
```

```
>>> factorial(4)          # With a trace of the process
n = 4
    n = 3
        n = 2
            n = 1
            factorial(1) = 1
        factorial(2) = 2
    factorial(3) = 6
factorial(4) = 24
```

Gathering File System Stats

- Count the files
- Size of a directory (number of bytes)



Modules `os` and `os.path`

```
os.getcwd()

os.listdir(dirname)

os.chdir(dirname)

os.path.isfile(name)

os.path.isdir(name)

os.path.getsize(filename)


Define functions:

countFiles(dirname)

getSize(dirname)
```

Counting the Files

- Use a recursive strategy
- Sum all of the items in the current directory
- If the item is a file, add 1
- Otherwise, the item is a directory, so add the count obtained from a recursive call of the function

Define and Use **countFiles**

```
import os
import os.path

def countFiles(dirname):
    """Counts the files in a directory and its subdirectories."""
    count = 0
    listOfItems = os.listdir(dirname)
    for item in listOfItems:
        if os.path.isfile(item):
            count += 1                                  # It's a file
        else:
            os.chdir(item)                          # It's a directory
            count += countFiles(os.getcwd())
            os.chdir("..")
    return count



countFiles(os.getcwd())
```