BYU IDAHO

CSE 212 | Programming with Data Structures

W08 Prepare: Reading

Table of Contents

Recursion

I. RECURSIVE FUNCTION CALLS

Usually when we write functions, we design them so they call different functions. **Recursion** is a technique where a function calls itself. For example, consider the following code:

```
def say_hello():
        print("Hello")
        say_hello()  # This is the recursive call
```

This code will print "Hello" forever. Actually, Python will eventually stop with a **`RecursionError`** because the **`say_hello`** function was called too many times. Notice that in this function, the first call to **`say_hello`** never has a chance to finish. In software, when a function is called, it is put onto a stack. The stack is used to keep track of what function to go back to when a function finishes. In this case, the stack is filling up.

II. RULES OF RECURSION

When we use recursion, we need to make sure we follow two important rules:

- Smaller Problem - When we call the function recursively, we need to make sure we are calling the function on a smaller problem. Without this rule, our function will run forever.
- Base Case - As we continue to call the function on a smaller problem, we need a place to stop. We must define a scenario in which recursion is not required. This is called the **base case**.

Applying these two rules to the **`say_hello`** function, we have the following modified code which is keeping track of how many times to say "Hello":

```
def say_hello(count):
        if count <= 0:  # Base Case
                return
        else:
                print("Hello")
                say_hello(count-1)  # Smaller Problem
```

In this new code, the smaller problem is **`count-1`** and the base case is when count is equal (or less than) zero. When we look at this code,

we should probably question the use of recursion when this could have been done with a simple **for** loop. Recursion should not be used with everything. When used inappropriately, recursion can result is significant performance degradation. However, when used wisely, a simple code solution can be found for complex problems.

III. SAMPLE PROBLEMS - FACTORIALS

Solving problems using recursion requires us to state the solution of a problem in terms of the problem itself (i.e., calling the function recursively). Some problems in mathematics offer good examples of recursion (performance is questionable, but the examples are sound).

A factorial involves multiplying a series of numbers. For a positive integer **n** (greater than 0), **n**! (read as **n** factorial) is defined as **n** * (**n**-1) * (**n**-2) * ... * 3 * 2 * 1. If we wanted to calculate **n**! using recursion, we need to define the answer in terms of the problem again. The "problem" here is the factorial function. We can rewrite **n**! as follows:

n! = **n** * (**n**-1)!

This solution above satisfies the first rule of recursion. To satisfy the second rule of recursion, we need to define **n**! for some value of **n** without using recursion so that our solution does not run forever. Without much math, we can solve 1! and say that it is equal to 1. We now have a base case. With our solution and base case, we can write the code:

```
def factorial(n):
        if n <= 1:
                return 1  # 1! = 1 (no recursion)
        else:
                return n * factorial(n-1)  # n! = n * (n-1)!
```

IV. SAMPLE PROBLEMS - FIBONACCI

The Fibonacci numbers are: 1, 1, 2, 3, 5, 8, 13, 21, 34, and so forth. The sequence starts with two 1's. Each subsequent number is the sum of the two previous values. If we wanted to write a function **`fib(n)`** which would give us the **`n`**th Fibonacci number, instead of thinking about loops, let's define **`fib(n)`** in terms of the same **`fib`** problem but with smaller values:

`fib(n)` = **`fib(n-1)`** + **`fib(n-2)`**

If we implement this, eventually we will get to calls of the **`fib`** function with smaller values of **`n`**. These smaller values of **`n`** represent the base case for recursion solution. Usually we try to think about solutions that we can easily calculate such as **`fib(1)`** which will equal 1. However, if we look at our formula above, we will need more than one base case. Consider **`n`**=3 which will require us to use **`fib(2)`** and **`fib(1)`**. If we then recursively solve for **`fib(2)`**, we will need **`fib(1)`** and **`fib(0)`**. In cases like this, we will need more than one base case representing the first two Fibonacci numbers:

`fib(2)` = 1

`fib(1)` = 1

Our resulting code will be as follows:

```
def fib(n):
        if n <= 2:
                return 1    # fib(2) = 1 and fib(1) = 1
        else:
                return fib(n-1) + fib(n-2)  # fib(n) = fib(n-1) + fib(n-2)
```

It is a useful exercise to analyze what happens when we call the **`fib`** function. The diagram below shows the functions that are called when we run **`fib(6)`**. Notice that the call to **`fib(n-1)`** is called before **`fib(n-2)`** and, therefore, the **`fib(n-1)`** must finish first. Also notice that there are many duplicate calls to the **`fib`** function for the same value of **`n`**.

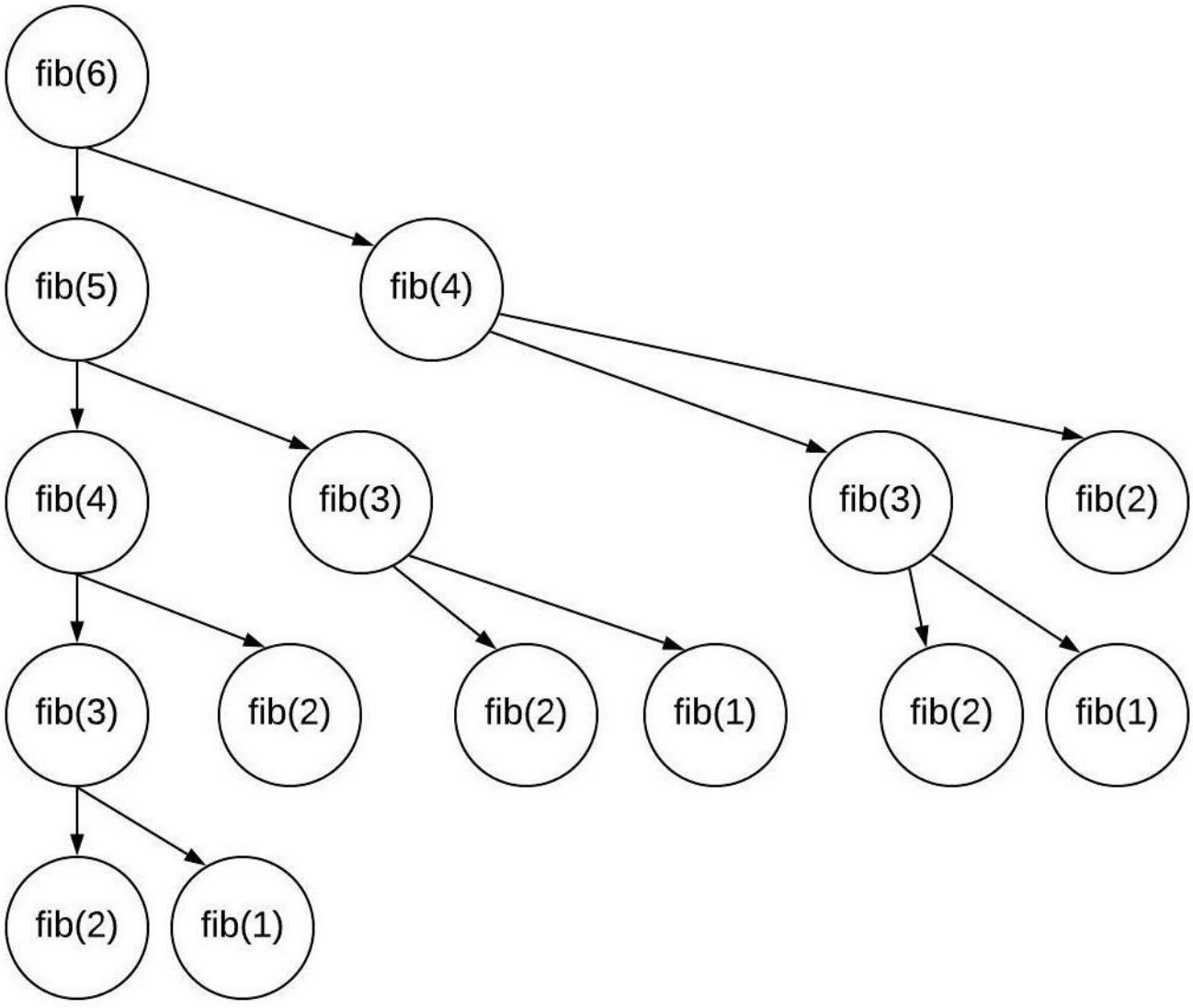

Figure 1 - Recursive Function Calls for fib(6)

The **`fib`** function was called a total of 15 times! This is an O(2^n) algorithm.

V. MEMOIZATION

We can improve the performance of the **`fib`** function by remembering previous results as we traverse through the recursive call. **Memoization** is the process of remembering these previous results so that additional recursive calls are not needed. For example, once we discover that **`fib(3)`** is equal to 2, we can store this into a Python dictionary with a key equal to 3 and the value of 2. This becomes a base case. If we need to calculate **`fib(3)`** again, we will just look up the 3 in the dictionary to get the answer.

The dictionary will only be used by the **`fib`** function and not be returned. Since the dictionary needs to be shared for all recursive calls, we will write code to create the dictionary on the first recursive call only.

```
def fibonacci(n, remember = None):
    # If this is the first time calling the function, then
    # we need to create the dictionary.
    if remember is None:
        remember = dict()

    # Base Case
    if n <= 2:
        return 1

    # Check if we have solved this one before
    if n in remember:
        return remember[n]

    # Otherwise solve with recursion
    result = fibonacci(n-1, remember) + fibonacci(n-2, remember)

    # Remember result for potential later use
    remember[n] = result
    return result

print(fibonacci(1))    # 1
print(fibonacci(2))    # 1
print(fibonacci(3))    # 2
print(fibonacci(4))    # 3
print(fibonacci(10))   # 55
```

```
print(fibonacci(100))  # 354224848179261915075 (This one will
                       # not work if you don't have the
                       # 'remember' dictionary implemented).
```

VI. SAMPLE PROBLEMS - FIND PERMUTATIONS

The problem is to calculate the number of ways to reorganize the letters in a word (i.e. the permutations). Mathematically, this should be `n!` where `n` is the number of letters in the word. However, using recursion, we can also display each of the permutations (so long as the number of letters is small, otherwise it will take a long time to display all the results).

Let's assume that our list of letters is ["A", "B", "C", "D"]. Thinking about smaller problems being solved recursively, we could say that the number of permutations would be the sum of the following four things:

- The number of permutations of A followed by all the different permutations of B, C, and D
- The number of permutations of B followed by all the different permutations of A, C, and D
- The number of permutations of C followed by all the different permutations of A, B, and D
- The number of permutations of D followed by all the different permutations of A, B, and C

Each recursive call to the `permutations` function will need to know two things: what letters have not been used yet, and the current string that has been built so far. In the four scenarios above, after we add the A, we are left with the letters B, C, and D (the letters that have not been used yet). Additionally, after we add the A, the A should be

added to the current string that we have built. The diagram below shows how these function calls will be called and how the resulting permutations will be displayed:

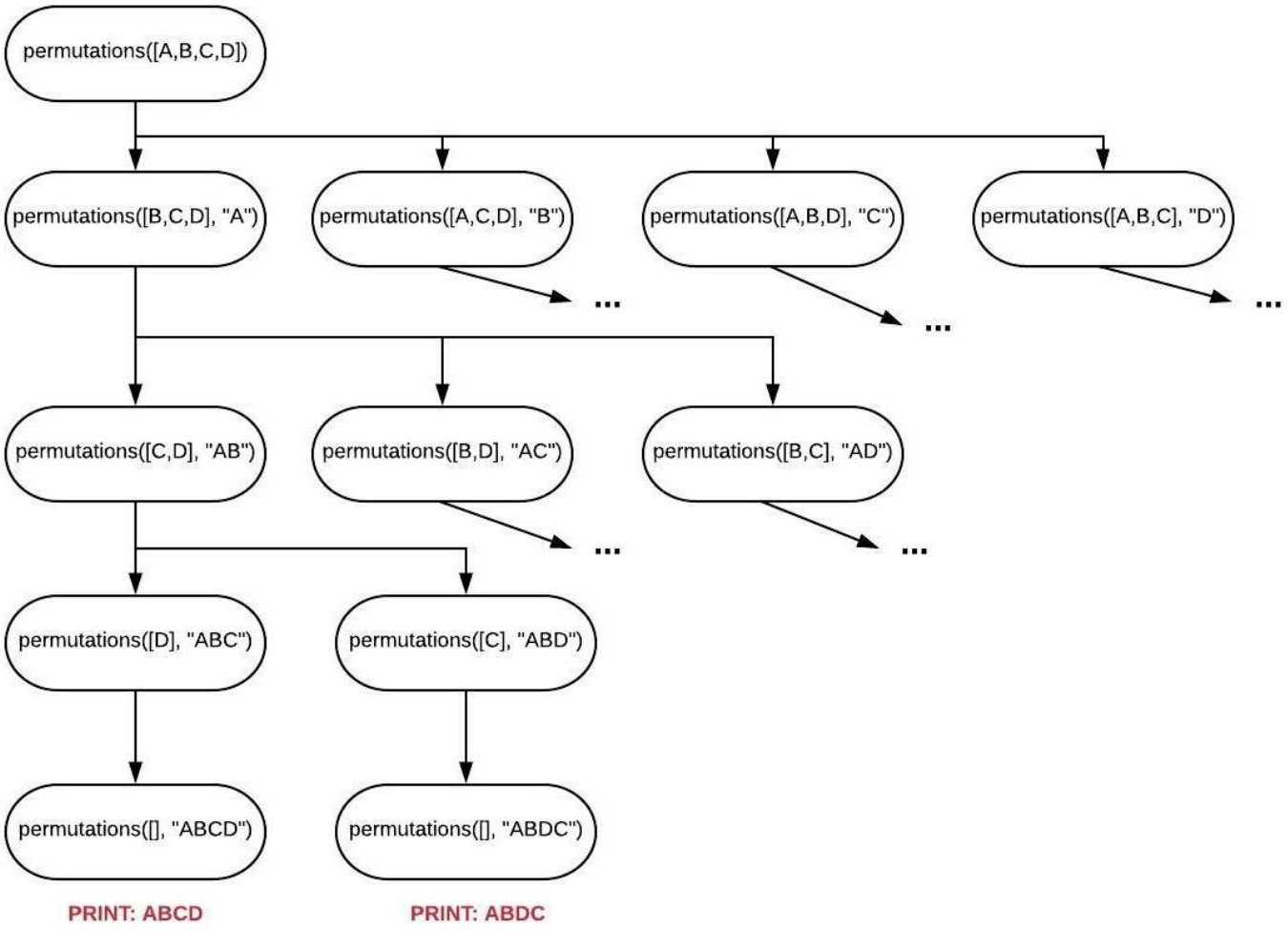

Figure 2 - Recursive Function Calls for permutations([A,B,C,D])

We also need a base case. The simplest scenario is a list with zero letters. Here is our code:

```
def permutations(letters, word=""):

        if len(letters) == 0:   # Base Case
                print(word)

        else:
                # Try adding each of the available letters
                # to the 'word_so_far' and add up all the
                # resulting permutations.

                for index in range(len(letters)):
```

```
                        # Make a copy of the letters to pass to the
                        # the next call to permutations.  We need
                        # to remove the letter we just added before
                        # we call permutations again.

                        letters_left = letters[:]
                        del letters_left[index]

                        # Add the new letter to the word we have so far
                        permutations(letters_left, word + letters[index])

permutations(list("ABC"))
"""
Results:
ABC
ACB
BAC
BCA
CAB
CBA
"""

permutations(list("ABCD"))
"""
Results:
ABCD
ABDC
ACBD
ACDB
ADBC
ADCB
BACD
BADC
BCAD
BCDA
BDAC
BDCA
CABD
CADB
CBAD
CBDA
CDAB
CDBA
DABC
DACB
DBAC
DBCA
DCAB
DCBA
"""
```

VII. SAMPLE PROBLEMS - BINARY SEARCH

Recursion plays an important role in several searching and sorting algorithms. The binary search algorithm assumes that the data is already sorted. Just like a phone book, if you had sorted data, then the best way to find something is to look in the middle of the data set. By looking in the middle of the sorted data, we can quickly exclude half of the data with a single comparison. The binary search algorithm is as follows:

- Base Case: If the list has just one item, then check it and return the result.
- Base Case: If the number in the middle of the list is what we are looking for, then the value is in the list
- Recursion: If the number in the middle of the list is not what we are looking for, then search in either the first half (lower values) or the second half (higher values). Calling the binary search function recursively on the list subset can either be done by creating a new list or by providing the function with the starting and ending index. The first approach will take more memory.

Here is the code for the binary search.

```
def binary_search(sorted_list, target):
    """
    This function uses list slicing.  A list slice will create a list from another list
    This is useful when we want to create new sublists.  Here is how list slicing works:

    data[:a] - Creates a new list from index 0 to index a-1
    data[a:] - Creates a new list from index a to len(data)-1
    data[a:b] - Creates a new list from index a to index b-1
    data[a:b:c] - Creates a new list from index a to index b-1 stepping by c
    """
    if len(sorted_list) == 1:
        # Base Case
        return target == sorted_list[0]
    else:
        # Find the middle and compare
        middle = len(sorted_list) // 2
```

```
        if target == sorted_list[middle]:
            # We got lucky and the middle was the match
            return True
        elif target < sorted_list[middle]:
            # Search the first half (index 0 to middle-1) and
            # return the result
            return binary_search(sorted_list[:middle],target)
        else:
            # Search the second half (index middle to end) and
            # return the result
            return binary_search(sorted_list[middle:],target)

print(binary_search([1, 3, 6, 18, 20, 25, 34, 38, 89, 95, 99, 100], 89)) # True
print(binary_search([1, 3, 6, 18, 20, 25, 34, 38, 89, 95, 99, 100], 1))  # True
print(binary_search([1, 3, 6, 18, 20, 25, 34, 38, 89, 95, 99, 100], 17)) # False
```

The performance of this recursive algorithm is O(log n) because we are excluding half of the list with each comparison.

Key Terms

» **base case** - The scenario that will terminate (or stop) the recursive calls. If this is not designed properly, then the recursion will run forever.

» **memoization** - The technique of remembering previous results found through recursion so that repetitive recursion can be avoided.

» **recursion** - The calling of a function with the same function. This can be used to solve problems by identifying a solution which is written in terms of solving the same problem using smaller values. A base case is needed to ensure that the recursion eventually stops. The base cases are solved in the function without using recursion.