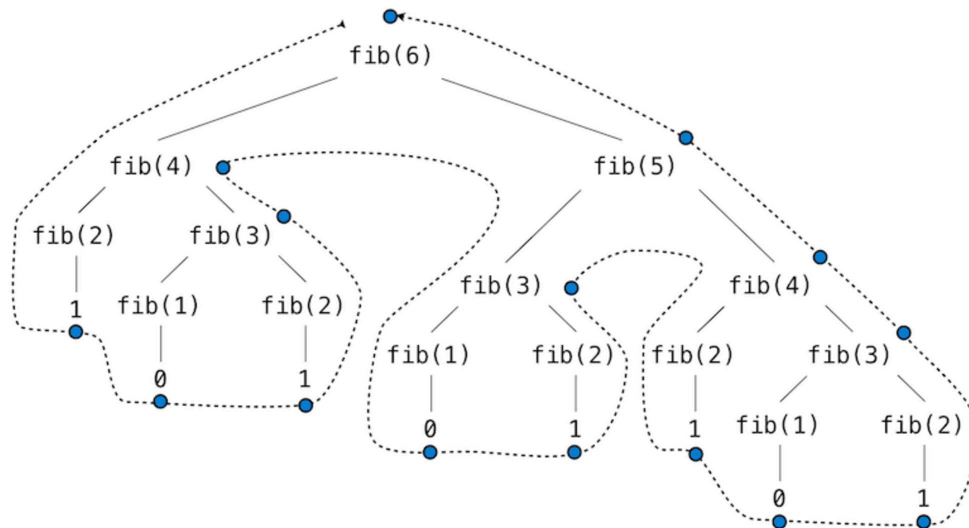
Memoization in Python

we saw multiple implementations of a function to compute Fibonacci numbers. The recursive version was as follows:

```
1  def fib(n):
2      if n == 1:
3          return 0
4      if n == 2:
5          return 1
6      return fib(n-2) + fib(n-1)
7
8  result = fib(6)
```

The recursive definition is tremendously appealing, since it exactly mirrors the familiar definition of Fibonacci numbers. However, consider the pattern of computation that results from evaluating `fib(6)`, shown below. To compute `fib(6)`, we compute`fib(5)` and `fib(4)`. To compute `fib(5)`, we compute `fib(4)` and `fib(3)`.



This recursive implementation is a terribly inefficient way to compute Fibonacci numbers because it does so much redundant computation. Notice that the entire computation of `fib(4)` -- almost half the work -- is duplicated. In fact, it is not hard to show that the number of times the function will compute `fib(1)` or `fib(2)` (the number of leaves in the tree, in general) is precisely`fib(n+1)`. To get an idea of how bad this is, one can show that the value of `fib(n)` grows exponentially with `n`. `fib(40)` is 63,245,986! The function above uses a number of steps that grows exponentially with the input.

We have also seen an iterative implementation of Fibonacci numbers, repeated here for convenience.

```
>>> def fib_iter(n):
        prev, curr = 1, 0  # curr is the first Fibonacci
number.
        for _ in range(n-1):
             prev, curr = curr, prev + curr
        return curr
```

The state that we must maintain in this case consists of the current and previous Fibonacci numbers. Implicitly, the `for` statement also keeps track of the iteration count. This definition does not reflect the standard mathematical definition of Fibonacci numbers as clearly as the recursive approach. However, the amount of computation required in the iterative implementation is only linear in `n`, rather than exponential. Even for small values of `n`, this difference can be enormous.

One should not conclude from this difference that tree-recursive processes are useless. When we consider processes that operate on hierarchically structured data rather than numbers, we will find that tree recursion is a natural and powerful tool.

Furthermore, tree-recursive processes can often be made more efficient through *memoization*, a powerful technique for increasing the efficiency of recursive functions that repeat computation. A memoized function will store the return value for any arguments it has previously received. A second call to `fib(4)` would not evolve the same complex process as the first, but instead would immediately return the stored result computed by the first call. If the memoized function is a pure function, then memoization is guaranteed not to change the result.

Memoization can be expressed naturally as a higher-order function, which can also be used as a decorator. The definition below creates a *cache* of previously computed results, indexed by the arguments from which they were computed. The use of a dictionary will require that the argument to the memoized function be immutable.

```
>>> def memo(f):
        """Return a memoized version of single-argument function f."""
        cache = {}
        def memoized(n):
            if n not in cache:
                cache[n] = f(n)
            return cache[n]
        return memoized
>>> @memo
    def fib(n):
        if n == 1:
            return 0
        if n == 2:
            return 1
        return fib(n-2) + fib(n-1)
>>> fib(40)
63245986
```

The amount of computation time saved by memoization in this case is substantial. The memoized, recursive *fib* function and the iterative *fib_iter* function both require an

amount of time to compute that is only a linear function of their input `n`. To compute`fib(40)`, the body of `fib` is executed 40 times, rather than 102,334,155 times in the unmemoized recursive case.

Space. To understand the space requirements of a function, we must specify generally how memory is used, preserved, and reclaimed in our environment model of computation. In evaluating an expression, we must preserve all *active* environments and all values and frames referenced by those environments. An environment is active if it provides the evaluation context for some expression being evaluated.

For example, when evaluating `fib`, the interpreter proceeds to compute each value in the order shown previously, traversing the structure of the tree. To do so, it only needs to keep track of those nodes that are above the current node in the tree at any point in the computation. The memory used to evaluate the rest of the branches can be reclaimed because it cannot affect future computation. In general, the space required for tree-recursive functions will be proportional to the maximum depth of the tree.

The diagram below depicts the environment created by evaluating `fib(3)`. In the process of evaluating the return expression for the initial application of `fib`, the expression `fib(n-2)` is evaluated, yielding a value of `0`. Once this value is computed, the corresponding environment frame (grayed out) is no longer needed: it is not part of an active environment. Thus, a well-designed interpreter can reclaim the memory that was used to store this frame. On the other hand, if the interpreter is currently evaluating`fib(n-1)`, then the environment created by this application of `fib` (in which `n` is `2`) is active. In turn, the environment originally created to apply `fib` to `3` is active because its return value has not yet been computed.

```
1  def fib(n):
2      if n == 1:
3          return 0
4      if n == 2:
```

```
5            return 1
6        return fib(n-2) + fib(n-1)
7
8    result = fib(3)
```

In the case of `memo`, the environment associated with the function it returns (which contains `cache`) must be preserved as long as some name is bound to that function in an active environment. The number of entries in the `cache` dictionary grows linearly with the number of unique arguments passed to `fib`, which scales linearly with the input. On the other hand, the iterative implementation requires only two numbers to be tracked during computation: `prev` and `curr`, giving it a constant size.

Memoization exemplifies a common pattern in programming that computation time can often be decreased at the expense of increased use of space, or vis versa.

Source : http://inst.eecs.berkeley.edu/~cs61A/book/chapters/interpretation.html#memoization