Ch. 5 Recursion

Thursday, May 7, 2020 8:41 AM

Fundamentals Ch. 5

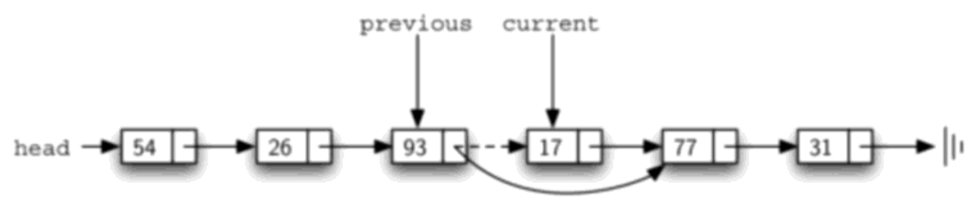


Figure 13: Removing an Item from the Middle of the List

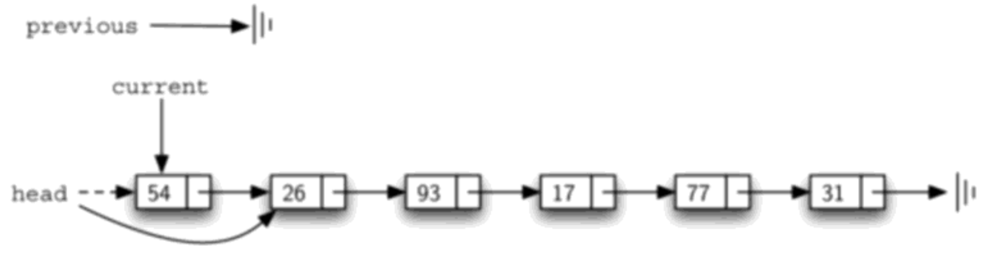


Figure 14: Removing the First Node from the List

5.2 Recursion

Recursion is a method of solving problems that involves breaking a problem down into smaller and smaller subproblems until you get to a small enough problem that it can be solved trivially. Usually recursion involves a function calling itself. While it may not seem like much on the surface, recursion allows us to write elegant solutions to problems that may otherwise be very difficult to program

In [1]:
```
# Addition with recursion instead of using for/while loops

def listsum(numList):
   if len(numList) == 1:
        return numList[0]

   else: # This function strips the first item of the list until you get a large
        return numList[0] + listsum(numList[1:])

print(listsum([1,3,5,7,9]))
```

```
25
```

When we reach the point where the problem is as simple as it can get, we begin to piece together the solutions of each of the small problems until the initial problem is solved. Figure 2 shows the additions that are performed as listsum works its way backward through the series of calls

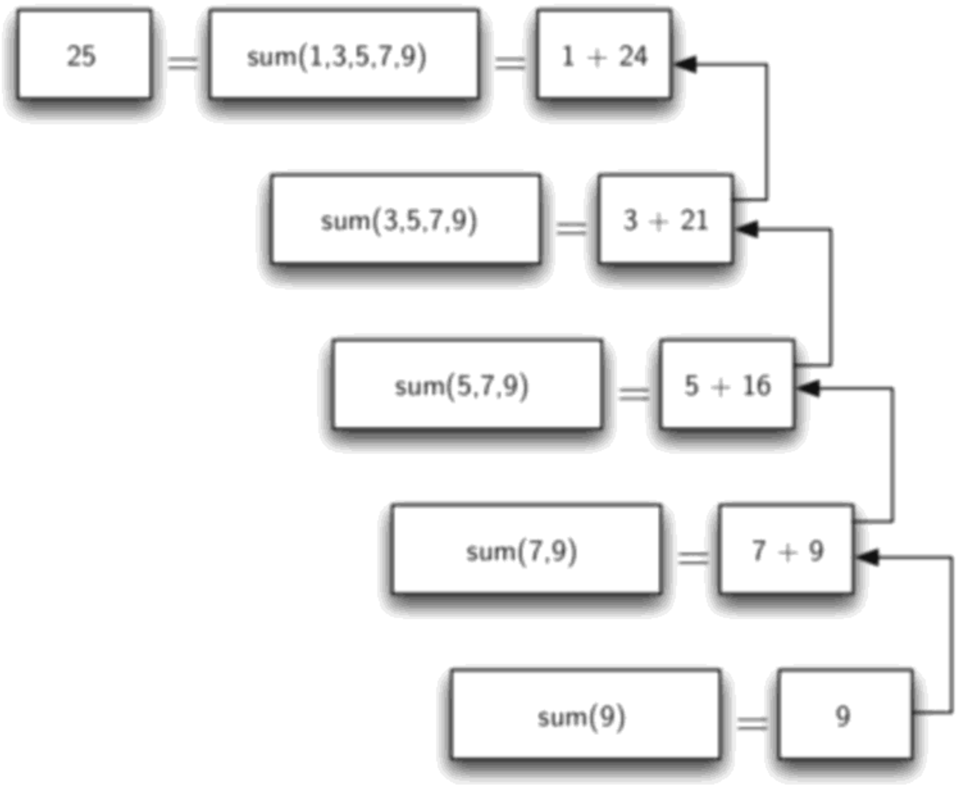


Figure2: Series of Recursive Returns from Adding a List of Numbers

Going backwards, recursion adds up all the function in the reverse order that they were called in

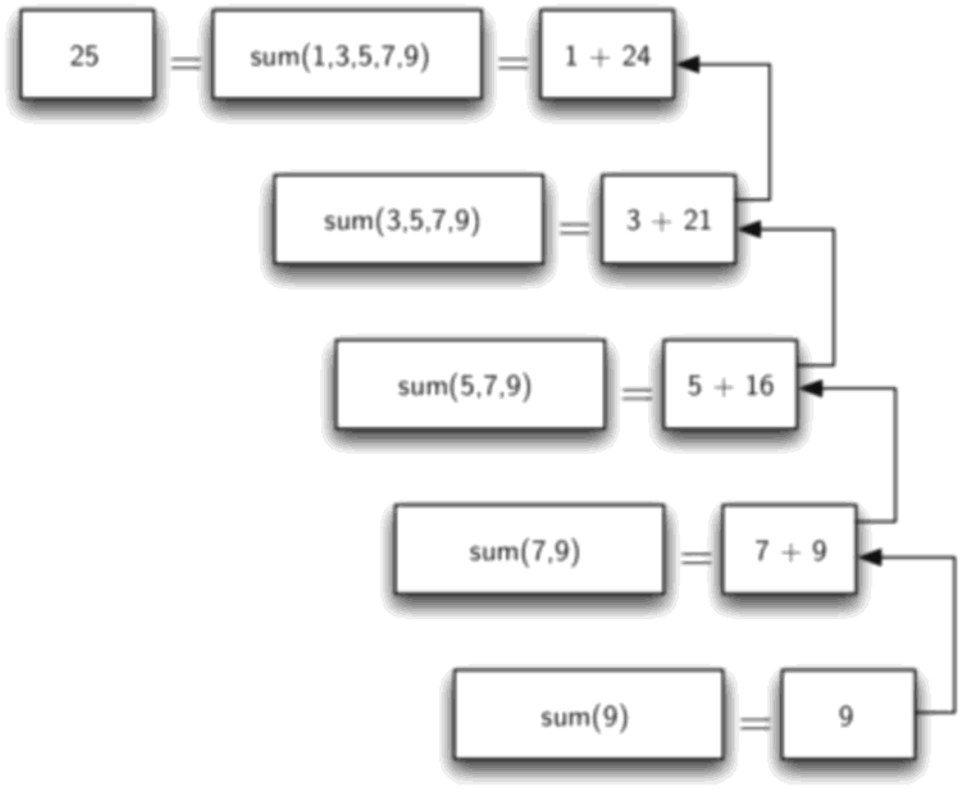


Figure2: Series of Recursive Returns from Adding a List of Numbers

base case: A list of length 1 (the problem is smalll enough to solve directly so recursion can stop) **recursion**: Shortening the list with each iteration making the problem smaler **calls itself**: Each iteration called a smaller problem

The Three Laws of Recursion

1. A recursive algorithm must have a base case
2. A recursive algorithm must change its state and move toward the base case
3. A recursive algorithm must call itself recursively

```
In [12]: #Converting an Integer into a String to Any Base

         def toStr(n,base):
            convertString = "0123456789ABCDEF" # Base Case: Convert the single digit-numbe
            if n < base:
               return convertString[n] # Reduce the original number to a series of single-
            else:
               return toStr(n//base,base) + convertString[n%base] # Concatenate the single

         print(toStr(1453,16))

         # Internal Regression Work

         # toStr(90) + "13"
         # toStr(5) + '10' + '13'
         # '5' + 'A' + 'D'

         # The stack frames also provide a scope for the variables used by the function. E
```

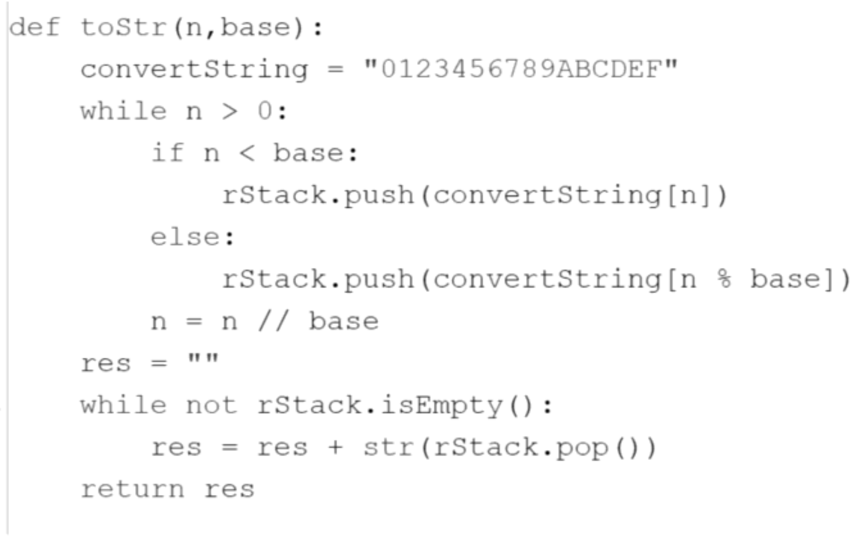
```
5AD
```

```
Out[12]: '\ntoStr(90) + "13"\n\ntoStr(5) + \'10\' + \'13\'\n\n\'5\' + \'A\' + \'D\'\n\n'
```

Thinking about toStr in terms of Stacks

A stack frame is allocated to handle the local variables of the function. When the function returns, the return value is left on top of the stack for the calling function to access. Figure 6 illustrates the call stack after the return statement on line 4.

The code above can be reinterpreted by the stack framework:



```
def toStr(n,base):
    convertString = "0123456789ABCDEF"
    while n > 0:
        if n < base:
            rStack.push(convertString[n])
        else:
            rStack.push(convertString[n % base])
        n = n // base
    res = ""
    while not rStack.isEmpty():
        res = res + str(rStack.pop())
    return res
```

Visualization

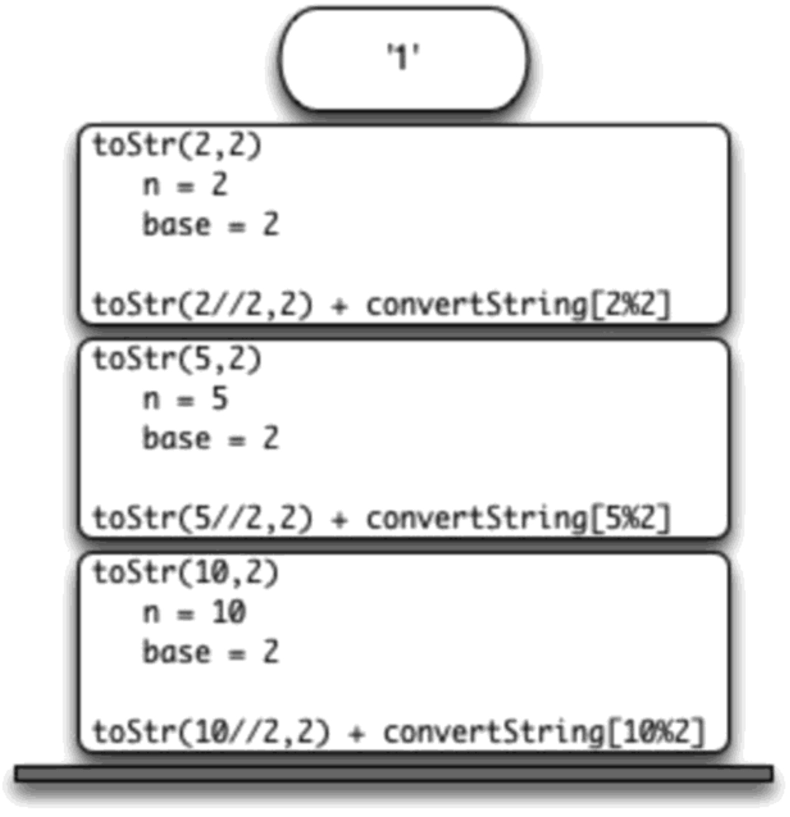

'1'
toStr(2,2)
n = 2
base = 2
toStr(2//2,2) + convertString[2%2]
toStr(5,2)
n = 5
base = 2
toStr(5//2,2) + convertString[5%2]
toStr(10,2)
n = 10
base = 2
toStr(10//2,2) + convertString[10%2]

Figure 6: Call Stack Generated from `toStr(10,2)`

5.8 Fractals

Sierpinski Triangle

Is a visual representation of a recurisive function. The procedure for drawing a Sierpinski triangle by hand is simple. Start with a single large triangle. Divide this large triangle into four new triangles by connecting the midpoint of each side. Ignoring the middle triangle that you just created, apply the same procedure to each of the three corner triangles. Each time you create a new set of triangles, you recursively apply this procedure to the three smaller corner triangles

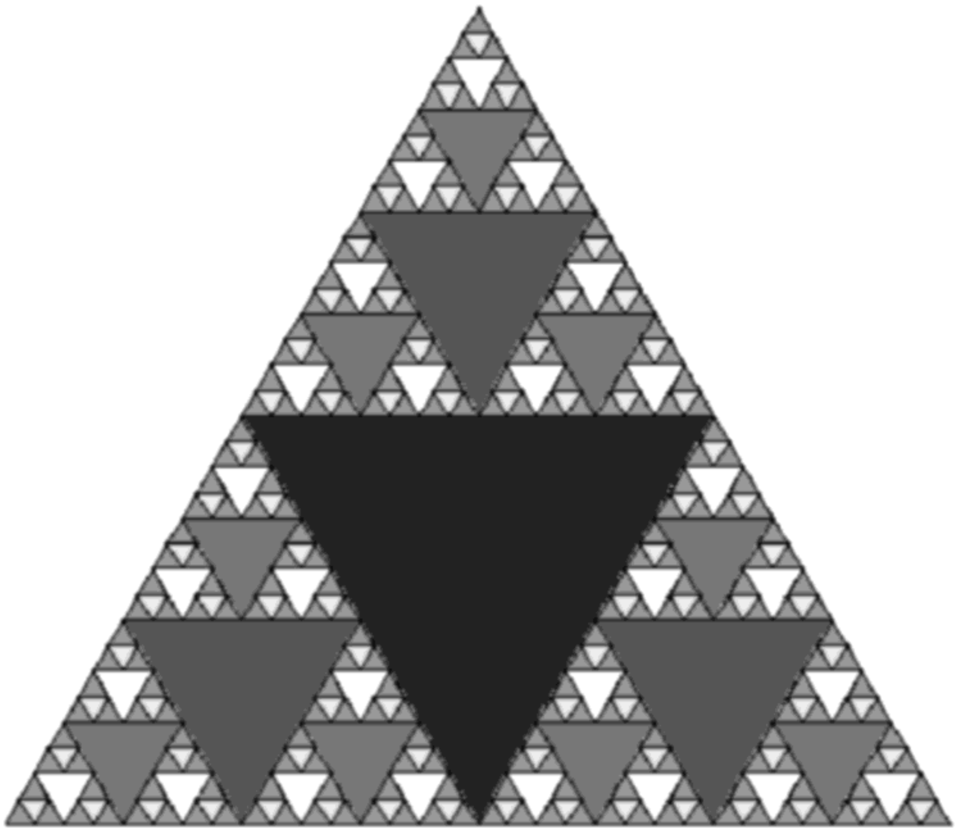


Figure 3: The Sierpinski Triangle

The base case is set arbitrarily as the number of times we want to divide the triangle into pieces. Sometimes we call this number the `“degree” of the fractal.`

Recursive algorithm can also be seen as a diagram of function calls. the recursive calls are always made going to the left. The active functions are outlined in black, and the inactive function calls are in gray. The farther you go toward the bottom of Figure 4, the smaller the triangles. The function finishes drawing one level at a time; once it is finished with the bottom left it moves to the bottom middle, and so on.

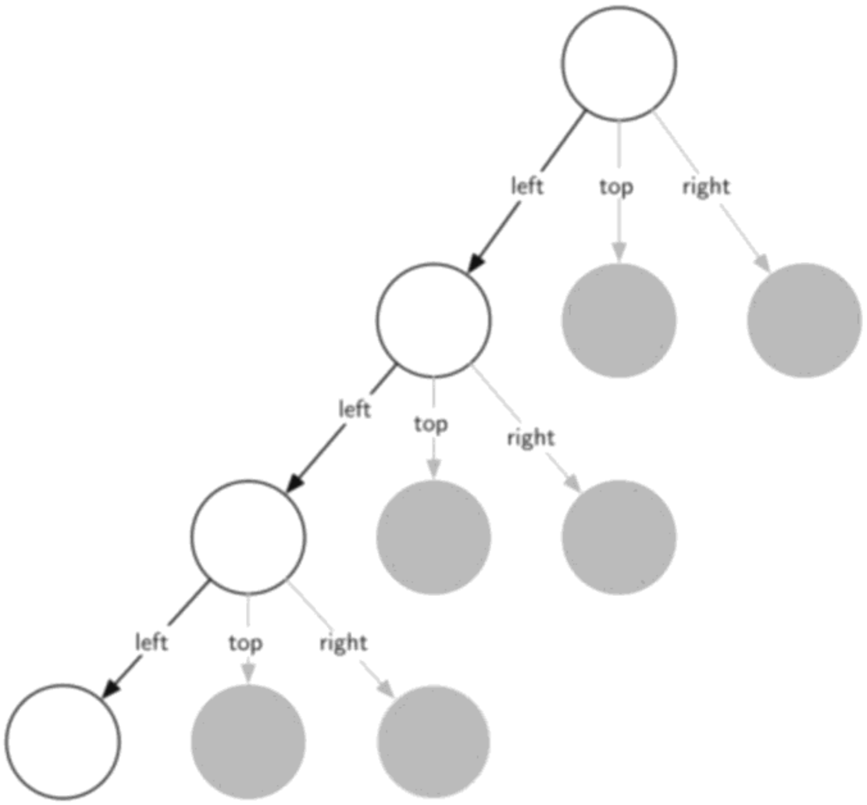


Figure 4: Building a Sierpinski Triangle

5.12 Dynamic Programming

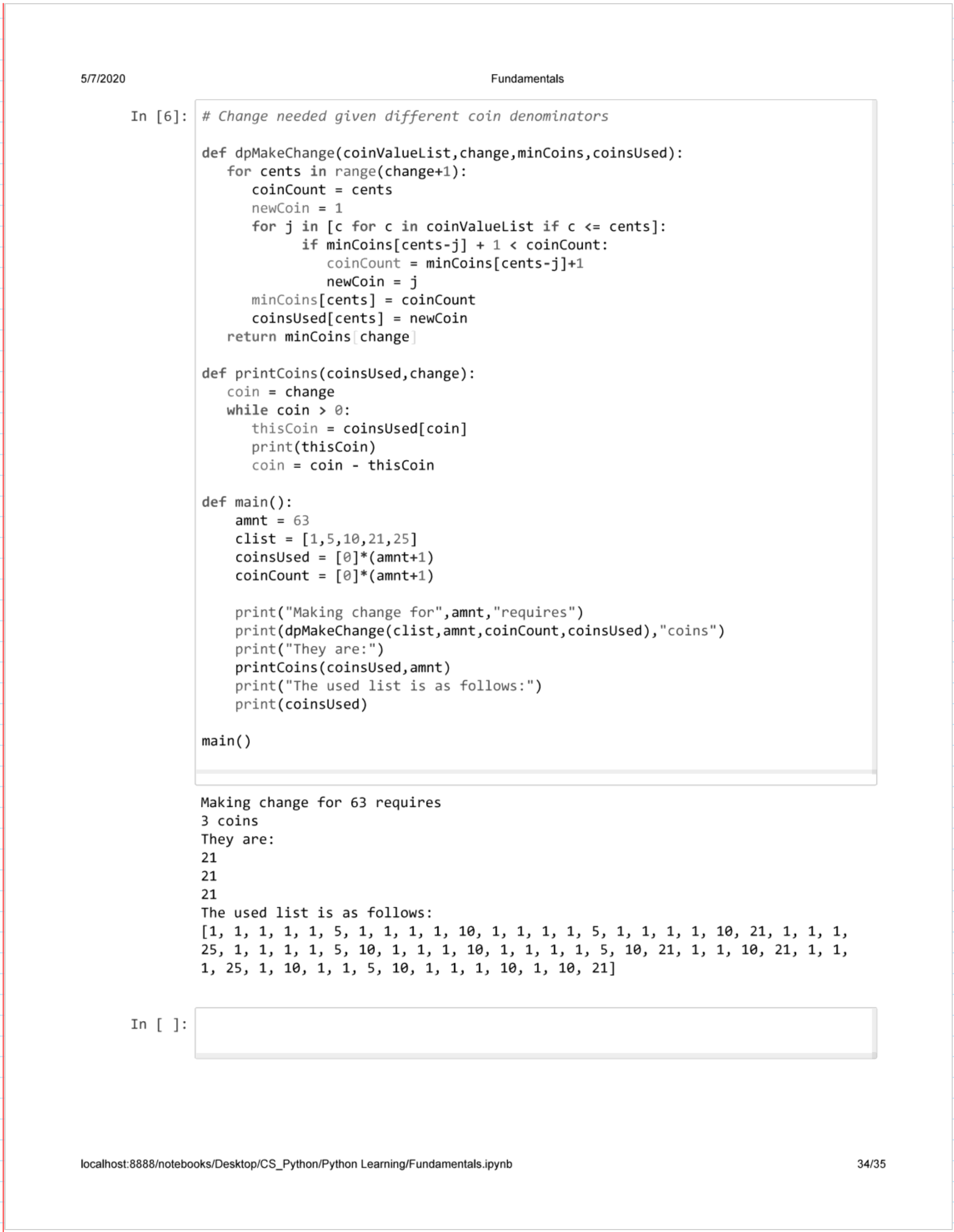
Dynamic programming solves problems regarding optimization using computer science

Greedy Method Solving the biggest problem first

It is important to realize that just because you can write a recursive solution to a problem does not mean it is the best or most efficient solution.

Summary

- All recursive algorithms must have a base case
- A recursive algorithm must change its state and make progress toward the base case
- A recursive algorithm must call itself (recursively)
- Recursion can take the place of iteration in some cases
- Recursive algorithms often map very naturally to a formal expression of the problem you are trying to solve
- Recursion is not always the answer



In [6]:
```
# Change needed given different coin denominators

def dpMakeChange(coinValueList,change,minCoins,coinsUsed):
   for cents in range(change+1):
      coinCount = cents
      newCoin = 1
      for j in [c for c in coinValueList if c <= cents]:
            if minCoins[cents-j] + 1 < coinCount:
               coinCount = minCoins[cents-j]+1
               newCoin = j
      minCoins[cents] = coinCount
      coinsUsed[cents] = newCoin
   return minCoins[change]

def printCoins(coinsUsed,change):
   coin = change
   while coin > 0:
      thisCoin = coinsUsed[coin]
      print(thisCoin)
      coin = coin - thisCoin

def main():
    amnt = 63
    clist = [1,5,10,21,25]
    coinsUsed = [0]*(amnt+1)
    coinCount = [0]*(amnt+1)

    print("Making change for",amnt,"requires")
    print(dpMakeChange(clist,amnt,coinCount,coinsUsed),"coins")
    print("They are:")
    printCoins(coinsUsed,amnt)
    print("The used list is as follows:")
    print(coinsUsed)

main()
```

```
Making change for 63 requires
3 coins
They are:
21
21
21
The used list is as follows:
[1, 1, 1, 1, 1, 5, 1, 1, 1, 1, 10, 1, 1, 1, 1, 5, 1, 1, 1, 1, 10, 21, 1, 1, 1,
25, 1, 1, 1, 1, 5, 10, 1, 1, 1, 10, 1, 1, 1, 1, 5, 10, 21, 1, 1, 10, 21, 1, 1,
1, 25, 1, 10, 1, 1, 5, 10, 1, 1, 1, 10, 1, 10, 21]
```

In [ ]: