Announcements

- Project 5 is on the street. Second part is essay questions for CoS teaming requirements.
  - The first part you do as a team
  - The CoS essay gets individually answered and has separate submission instructions
- No office hours tomorrow

Recursion

- It is OK for a function to call yourself, but you need some skill:
  - Identify self-similarity in the problem
  - Know when to stop
- Think of it as cloning the function.

Key Insight

- To understand and be able to program recursively, you must
  - Break down the problem into sub problems and
  - Join the solution of those sub problems back to get the solution of the original problem.

CQ:

Merge sort can be done using recursion

A. True

B. False

C. Depends

Recall Binary Search

- The basic idea of the binary search algorithm was to iteratively divide the problem in half.
- This technique is known as the *divide and conquer* approach in algorithm design
- Divide and conquer divides the original problem into sub-problems that are smaller versions of the original problem.

Recursive Problem-Solving

- In the binary search, the initial range is the entire list. We look at the middle element… if it is the target, we're done. Otherwise, we continue by performing a binary search on either the top half or bottom half of the list.
- In the iterative version of Binary Search, you split the list into half after each iteration. If the element is less than the middle, then you search for the lower half of the list in the next iteration, otherwise you search for the upper half of the list.
- There is another way to similarly solve Binary Search.

Recursive Algorithm for Binary Search

```
def binarySearch(key, low, high, numlist):
    mid = (low + high)//2
    if low > high:
        return -1
    if key == numlist[mid]:
        return mid
    elif key < numlist[mid]:
        return binarySearch(key, low, mid-1, numlist)
    else:
        return binarySearch(key, mid+1, high, numlist)
```

Recursive Definitions

- A description of something that refers to itself is called a *recursive* definition.
- In the last example, the binary search algorithm uses its own description – a “call” to binary search “recurs” inside of the definition – hence the label “recursive definition.”
- The function is calling itself with new parameters. If you notice, the parameters low and high change in every recursive call.
- The parameters are local to the particular instantiation of the function body.

Recursive Definitions Rules

- All good recursive definitions have these two key characteristics:
  - There are one or more base cases for which no recursion is applied.
  - All chains of recursion eventually end up at one of the base cases.
- The simplest way for these two conditions to occur is for each recursion to act on a smaller version of the original problem. A very small version of the original problem that can be solved without recursion becomes the base case.

Recursive Definitions: Math Example

- In mathematics, recursion is frequently used. The most common example is the factorial:
- For example, 5! = 5(4)(3)(2)(1), or
  5! = 5(4!)

$$n! = n(n-1)(n-2)...(1)$$

Factorial Recursive Definition

- In other words,

$$n! = n(n-1)!$$

- Or

$$n! = \begin{cases} 1 & \text{if } n = 0 \\ n(n-1)! & \text{otherwise} \end{cases}$$

- This definition says that 0! is 1, while the factorial of any other number is that number times the factorial of one less than that number.

Factorial Recursive Definition

- Our definition is recursive, but definitely not circular.
- Circular definition will keep on going indefinitely. Recursive definitions on the other hand stops at one point of its execution
- Consider 4!
  - 4! = 4(4-1)! = 4(3!)
  - What is 3!? We apply the definition again
    4! = 4(3!) = 4[3(3-1)!] = 4(3)(2!)
  - And so on…
    4! = 4(3!) = 4(3)(2!) = 4(3)(2)(1!) = 4(3)(2)(1)(0!) = 4(3)(2)(1)(1) = 24

Factorial Recursive Definition

- Factorial is not circular because we eventually get to 0!, whose definition does not rely on the definition of factorial and is just 1. This is called a *base case* for the recursion.
- When the base case is encountered, we get a closed expression that is computed directly.

Recursive Algorithm for Factorial

- We've seen previously that factorial can be calculated using a loop accumulator.
- Factorial can be written recursively as:

```
def factorial(n):
    if n == 0:
        return 1
    else:
        return n * factorial(n-1)
```

Factorial: Calling recursive function

```
>>> factorial(4)

24

>>> factorial(10)

3628800
```

Remember:

each call to a function starts that function anew, with its own copies of local variables and parameters.

Example: String Reversal

- Python lists have a built-in method that can be used to reverse the list. What if you wanted to reverse a string?
- If you wanted to program this yourself, one way to do it would be to convert the string into a list of characters, reverse the list, and then convert it back into a string.

Example: String Reversal

- Using recursion, we can calculate the reverse of a string without the intermediate list step.
- Think of a string as a recursive object:
  - Divide it up into a first character and “all the rest”
  - Reverse the “rest” and append the first character to the end of it

Example: String Reversal

-
  ```
  def reverse(s):
      return reverse(s[1:]) + s[0]
  ```
- The slice **`s[1:]`** returns all but the first character of the string.
- We reverse this slice and then concatenate the first character (**`s[0]`**) onto the end.

Example: String Reversal

-
  ```
  >>> reverse("Hello")

  Traceback (most recent call last):
    File "<pyshell#6>", line 1, in -toplevel-
      reverse("Hello")
    File "C:/Program Files/Python 2.3.3/z.py", line 8, in
  reverse
      return reverse(s[1:]) + s[0]
    File "C:/Program Files/Python 2.3.3/z.py", line 8, in
  reverse
      return reverse(s[1:]) + s[0]
  …
   File "C:/Program Files/Python 2.3.3/z.py", line 8, in
  reverse
      return reverse(s[1:]) + s[0]
  RuntimeError: maximum recursion depth exceeded
  ```
- What happened? There were 1000 lines of errors!

Example: String Reversal

- Remember: To build a correct recursive function, we need a base case that doesn’t use recursion.
- We forgot to include a base case, so our program is an *infinite recursion*. Each call to **`reverse`** contains another call to **`reverse`**, so none of them return.

Example: String Reversal

- Each time a function is called it takes some memory. Python stops it at 1000 calls, the default “maximum recursion depth.”
- What should we use for our base case?
- Following our algorithm, we know we will eventually try to reverse the empty string. Since the empty string is its own reverse, we can use it as the base case.

Example: String Reversal

-
  ```
  def reverse(s):
      if s == "":
          return s
      else:
          return reverse(s[1:]) + s[0]
  ```

-
  ```
  >>> reverse("Hello")
  'olleH'
  ```

Example: Anagrams

- An *anagram* is formed by rearranging the letters of a word.
- Anagram formation is a special case of generating all permutations (rearrangements) of a sequence, a problem that is seen frequently in mathematics and computer science.

Example: Anagrams

- Let’s apply the same approach from the previous example.
  - Slice the first character off the string.
  - Place the first character in all possible locations within the anagrams formed from the “rest” of the original string.

Example: Anagrams

- Suppose the original string is “abc”. Stripping off the “a” leaves us with “bc”.
- Generating all anagrams of “bc” gives us “bc” and “cb”.
- To form the anagram of the original string, we place “a” in all possible locations within these two smaller anagrams: [“abc”, “bac”, “bca”, “acb”, “cab”, “cba”]

Example: Anagrams

- As in the previous example, we can use the empty string as our base case.
-
```
def anagrams(s):
    if s == "":
        return [s]
    else:
        ans = []
        for w in anagrams(s[1:]):
            for pos in range(len(w)+1):
                ans.append(w[:pos]+s[0]+w[pos:])
        return ans
```

Example: Anagrams

- A list is used to accumulate results.
- The outer **`for`** loop iterates through each anagram of the tail of **`s`**.
- The inner loop goes through each position in the anagram and creates a new string with the original first character inserted into that position.
- The inner loop goes up to **`len(w)+1`** so the new character can be added at the end of the anagram.

Example: Anagrams

- **`w[:pos]+s[0]+w[pos:]`**
  - **`w[:pos]`** gives the part of **`w`** up to, but not including, **`pos`**.
  - **`w[pos:]`** gives everything from **`pos`** to the end.
  - Inserting **`s[0]`** between them effectively inserts it into **`w`** at **`pos`**.

Example: Anagrams

- The number of anagrams of a word is the factorial of the length of the word.

-
  ```
  >>> anagrams("abc")
  ['abc', 'bac', 'bca', 'acb', 'cab', 'cba']
  ```

Example: Ackermann Function

$A(0, j) = j + 1$ for $j \geq 0$
$A(i, 0) = A(i - 1, 1)$ for $i > 0$
$A(i, j) = A(i - 1, A(i, j - 1))$ otherwise

Function grows extremely fast; $A(k, k)$ grows faster than iterated exponentiation:

$$E(0) = 2$$
$$E(n + 1) = 2^{E(n)}$$

$$E(1) = 4$$
$$E(2) = 16$$
$$E(3) = 64K$$
$$E(4) = 2^{64K}$$

…

McCarthy’s 99 Function

$$M(n) = n - 10 \;\; if \;\; n > 100$$
$$M(n) = M(M(n + 11)) \;\; if \;\; n \le 100$$

Not obvious that the computation terminates…