Worksheet #12 Recursion Answer Key =)

TRUE/FALSE

1. (12 points: 1 point each) The following are true/false questions. **Write either T or F in the boxes at the bottom of page 1.** If there's any counterexample, it's false.

   (a) Recursion is a programming technique where a function calls itself.

   (b) A recursive function must have a base case to stop the recursion.

   (c) Any recursive function using a list can be adjusted to use a set.

   (d) Recursive functions are always more efficient than iterative solutions.

   (e) Recursive functions can be called with different parameters in each recursive call.

   (f) A recursive function can have multiple base cases.

   (g) Recursive functions can only call themselves once within their body.

   (h) Recursion is the only way to implement functions that solve certain mathematical problems efficiently, such as factorials.

   (i) If function A calls function B, and function B calls function A, then neither function is recursive because they do not directly call themselves.

   (j) Recursion always leads to infinite recursive calls if not implemented correctly.

   (k) Recursion is generally recommended for problems that can be easily solved using loops.

   (l) Recursive functions in Python cannot be optimized for better performance.

a	b	c	d	e	f	g	h	i	j	k	l
T	T	F	F	T	T	F	F	F	F	F	F

Notes:
(C) – not true! If the recursive function indexes into the list, it cannot be adjusted to use a set, because sets do not allow indexing.
(H) – totally silly ;) If we can solve a problem using recursion, we can solve it iteratively, too.
(J) – not true! Obviously, if you're a sloppy programmer (no offense to y'all out there!), you can get tons of other problems, index out of bounds errors.
(K) – if we could easily solve them with loops, then why use recursion? It's also not guaranteed that the problem is as easily solvable with a recursive approach.

(I) Refer to slide 44 of recursion lectures, a recursive approach to Fibonacci is optimized.

MULTIPLE CHOICE

2	3	4	5	6	7	8	9
C	D	C	E	A/B	A	A	C

2. What is the purpose of a base case in a recursive function?
   A. To make the code more readable and maintainable.
   B. To handle errors and exceptions that may occur during recursion.
   C. To define the initial condition that stops the recursive calls.
   D. To ensure that the function returns a value at every step of the recursion.
   E. Base cases are not required in recursive functions.

Correct answer explanation:
(C) Recursive functions repeatedly call themselves with modified parameters until a certain condition is met. The base case is that condition. It represents the simplest scenario where the function does not need to make a recursive call and can provide a result directly. When the base case is met, the recursion stops. In other words, the base case provides the exit strategy for the recursive process. Without a base case, the recursive function would keep calling itself indefinitely.

Wrong answer explanations:
(A) While having a base case can arguably contribute to readability of recursive code by providing a clear termination condition, this is not the primary purpose of a base case, and it still doesn't relate to maintainability.
(B) A base case also doesn't necessarily handle errors and exceptions, like checking for parameter size to avoid exceeding maximum recursion depth.
(D) A base case is about determining when the recursion should stop, not necessarily about returning values at every step.
(E) Without a base case, a recursive function may continue calling itself indefinitely, leading to a stack overflow or infinite recursion.

3. Which of the following is true about the relationship between recursive and iterative solutions?

   A. Recursive solutions are always clearer than iterative solutions.
   B. Iterative solutions are always more efficient than recursive solutions because the function does not call itself repeatedly.
   C. Iterative solutions are prone to errors such as infinite loops, while recursive solutions are not.
   D. Recursion and iteration are different techniques with their own strengths and weaknesses, although they can be used interchangeably.

Correct answer explanation:
(D) Yes! Slide 36 from the recursion lecture states "You can always convert a recursive solution to an iterative solution, and vice versa. But it may not be easy!" Whether you should take a recursive or an iterative approach depends on the problem, but it is possible to switch one out for the other.

Wrong answer explanations:
(A) Not true! We saw in the recursion lecture videos that Towers of Hanoi had quite a concise recursive solution, and it was stated that the iterative approach would have been much more complex.
(B) Not true! Too generalized. Recursive function calls can introduce additional inefficiencies, but it doesn't necessarily mean that iterative solutions are always more efficient. Some problems are naturally suited to recursion (like Towers of Hanoi).
(C) Iterative solutions are no more error-prone than recursive solutions -- it simply depends upon the skill of the programmer. Recursive solutions also face their own unique errors, such as exceeding maximum recursion depth.

4. What is the primary advantage of using recursion in programming?

   A. Recursion allows for more efficient memory utilization.
   B. Recursive implementations are always simpler because they do not use loops.
   C. Recursion can solve complex problems concisely by breaking them down into smaller instances.
   D. Recursion improves the speed of program execution.
   E. Recursive solutions are easier to debug.

Correct answer explanation:

(C) Yes! Recursion is intended to solve one big problem, by solving multiple smaller instances of it, and building a way up to the original problem.

Wrong answer explanations:
(A) Not guaranteed. This depends not only on the problem but also the specific recursive implementation.
(B) Not necessarily! It's easy to see how some recursive functions on homework 12 (such as integerToList) might be simpler with an iterative approach.
(D) Not guaranteed. Again, this depends on the problem and on the implementation.
(E) Not guaranteed! Iterative solutions tend to be more straightforward, so it can be easier to use print statements in certain parts and understand what the program is doing wrong, than navigating through multiple levels of nested calls in recursive functions.

5. What is a potential drawback of using recursion in programming?

   A. Recursion can lead to infinite function calls and stack overflow errors.
   B. Recursion is only applicable to mathematical calculations.
   C. Recursion can be challenging with some data structures, like dictionaries.
   D. Using recursion makes the code more difficult to read and understand.
   E. Both A and C

Correct answer explanation:
(E) Yes! Both A and C are true. When using recursion, you should be wary of having a correct base case (so the recursion eventually terminates), exceeding the maximum recursion depth (stack overflow errors), and that recursive approaches using some objects (such as sets, which have no indexes) can be tricky.

Wrong answer explanations:
(B) Not true! Homework 12 has recursive functions on strings (palindrome, find first uppercase, etc.)
(D) Not always the case! Homework 12 has some recursive functions that aren't difficult to read.

6. Which of the following is a correct recursive implementation of the power function?

A.
```
def power(x, n):
    if n == 0:
        return 1
    return x * power(x, n - 1)
```

B.
```
def power(x, n):
    if n == 1:
        return x
    else:
        return x * power(x, n - 1)
```

C.
```
def power(x, n):
    return x ** n
```

D.
```
def power(x, n):
    if n == 0:
        return 1
    else:
        return power(x, n) * x
```

Correct answer explanation:
Either (A) or technicallyyyy (B) will work – the only difference is that (A), because its base case is n == 0, will avoid crashing when 0 is given for n. (B)'s base case checks for n == 1, meaning that if n is less than 1 originally, the function will keep decrementing n endlessly and thus run into a maximum recursion depth exceeded error. I should probably patch this so that only (A) is correct.

Example of (A) in action:

Recursive calls made when x == 5 and n == 3:
power(5, 3) → 5 * power(5, 2)
power(5, 2) → 5 * power(5, 1)
power(5, 1) → 5 * power(5, 0)
power(5, 0) → 1 *(base case hit)*

Now we can plug in, from bottom to top.

If we know power(5, 0) is 1, then power(5, 1) must be 5 * 1, which is 5.
If we know power(5, 1) is 5, then power(5, 2) must be 5 * 5, which is 25.

If we know power(5, 2) is 25, then power(5, 3) must be 5 * 25, which is 125 – our final answer.

Wrong answer explanations:
(B) See above. =)
(C) This approach isn't recursive!!
(D) This approach doesn't change the parameter in any way for the next recursive call, meaning that the function (assuming n is originally > 0) will call itself forever.

7. What does the term "recursion depth" refer to?

   A. The number of recursive calls made by a function before a base case triggers.
   B. The depth of nested loops in a recursive algorithm.
   C. The total number of lines in a recursive function.
   D. The level of indentation in a recursive function.

Correct answer:
(A): Recursion depth refers to the amount of recursive calls on the 'stack' at once (in limbo, essentially, because we have not yet hit a base case and thus cannot return an actual value to earlier calls).

That is to say: recursion depth is the number of times a recursive function calls itself before reaching a base case (our stopping condition). Each time a function calls itself, the recursion depth increases by one.

(B), (C), and (D) are just silly answers.

8. Under what circumstances is it opportune to use recursion in programming?

   A. When the problem can be naturally divided into smaller instances of the same problem, and requires a parameter that can be minimized or truncated
   B. Only when the problem is impossible or too complex to solve iteratively.
   C. When aiming for the most memory-efficient solution due to the inherent nature of recursive algorithms.
   D. Whenever recursion is available as an option, as it often simplifies code and enhances maintainability.

Correct answer explanation:
(A) Yes! This is the intent behind recursion. Solve smaller instances of the same problem, but by truncating the parameter a step at a time (example: with strings or lists, perhaps by removing the last element, and having our base case be when the string or list has length 0).

Wrong answer explanations:

(B) Not true. If the problem is impossible to solve iteratively, then we couldn't solve it recursively, either. (Slide 36 from the recursion lecture states "You can always convert a recursive solution to an iterative solution, and vice versa. But it may not be easy!" So if one approach is impossible, the other must be too.)

(C) Not true! Again, this depends not only on the problem but also the specific recursive implementation.

(D) Not true! Obviously, there are some problems where an iterative approach would be simplest.

9. What is the correct recursive implementation for a function that takes two ordered strings as input and returns their (ascending) ordered concatenation?

A.
```
def ordered_concat(str1, str2):
    if not str1:
        return str2
    elif not str2:
        return str1
    else:
        return ordered_concat(str1[1:], str2[1:]) + max(str1[0], str2[0])
```

B.
```
def ordered_concat(str1, str2):
    if len(str1) == 0:
        return str2
    elif len(str2) == 0:
        return str1
    else:
        return ordered_concat(str1[1:], str2[1:]) + min(str1[0], str2[0])
```

C.
```
def ordered_concat(str1, str2):
    if not str1:
        return str2
    elif not str2:
        return str1
    elif str1[0] < str2[0]:
        return str1[0] + ordered_concat(str1[1:], str2)
    else:
        return str2[0] + ordered_concat(str1, str2[1:])
```

D.
```
def ordered_concat(str1, str2):
    if len(str1) == 0:
        return str2
    elif len(str2) == 0:
        return str1
    elif str1[0] > str2[0]:
        return max(str1[0], str2[0]) + ordered_concat(str1[1:], str2)
    else:
        return str2[0] + ordered_concat(str1, str2[1:])
```

Correct answer:

(C) This implementation is correct because in each call, the first characters of str1 and str2 are compared (str1[0] and str2[0]), and whichever one is less, is put at the front (lesser items at the front adheres with ascending ordering). We then truncate the string with this character, so the next recursive call compares a different pair of characters. This process continues until one of the strings becomes empty, at which point the concatenation is performed with the remaining characters in the non-empty string.

Recursive calls made when str1 = “acf” and str2 = “bde”:

ordered_concat(“acf”, “bde”) → “a” + ordered_concat(“cf”, “bde”)

- Remember this function compares the first character of each string – “a” is less than “b”

ordered_concat(“cf”, “bde”) → “b” + ordered_concat(“cf”, “de”)
ordered_concat(“cf”, “de”) → “c” + ordered_concat(“f”, “de”)
ordered_concat(“f”, “de”) → “d” + ordered_concat(“f”, “e”)
ordered_concat(“f”, “e”) → “e” + ordered_concat(“f”, “”)
ordered_concat(“f”, “”) → “f” *(base case triggered)*

Now we can plug in, from bottom to top.

If we know that ordered_concat(“f”, “”) = “f”, then ordered_concat(“f”, “e”) must be “e” + “f” = “ef”.
If we know that ordered_concat(“f”, “e”) = “ef”, then ordered_concat(“f”, “de”) must be “d” + “ef” = “def”.
If we know that ordered_concat(“f”, “de”) = “def”, then ordered_concat(“cf”, “de”) must be “c” + “def” = “cdef”.
If we know that ordered_concat(“cf”, “de”) = “cdef”, then ordered_concat(“cf”, “bde”) must be “b” + “cdef” = “bcdef”.
If we know that ordered_concat(“cf”, “bde”) = “bcdef”, then ordered_concat(“acf”, “bde”) must be “a” + “bcdef” = “abcdef”.

Wrong answer explanations:
(A) + (B): Both implementations are incorrect because in the 'else', the recursive calls truncate both str1 and str2 by removing their first characters. However, when combining the results of these recursive calls, only one of the truncated characters (determined by either max(str1[0], str2[0]) or min(str1[0], str2[0])) is added. Thus, the character that is not selected is discarded, as both strings are shortened in the recursive call (via ordered_concat(str1[1:], str2[1:])). This results in the exclusion of characters from the final resulting string.
(D) this implementation results in a descending (not ascending) string. The second ‘elif’ here looks tricky, with the max(str1[0]. str2[0]) but note that the condition is ‘elif str1[0] > str2[0]’, so we already know that the max between str1[0] and str2[0] should be str1[0], otherwise this branch would not have triggered.

TRACING

10. (3 points)

```
def coraline(wybie):
    if wybie <= 0:
        return
    print(wybie, end = " ")
    coraline(wybie - 2)
    print(wybie, end = " ")

print(coraline(5))
```

5 3 1 1 3 5 None

Recursive calls made:

coraline(wybie = 5) → does not trigger base case, prints 5, calls coraline(wybie = 3), prints 5
coraline(wybie = 3) → does not trigger base case, prints 3, calls coraline(wybie = 1), prints 3
coraline(wybie = 1) → does not trigger base case, prints 1, calls coraline(wybie = -1), prints 1
coraline(wybie = -1) → hits base case, returns None

From this:

We call coraline(5) which prints **5** then calls coraline(3).

- coraline(3) will print **3**, then call coraline(1).
    - coraline(1) will print **1**, then call coraline(-1).
        - coraline(-1) returns None.
    - coraline(1) prints **1** again.
- coraline(3) prints **3** again

coraline(5) prints **5** again
coraline(5) implicitly returns None. Because we ran the code print(coraline(5)) and not just coraline(5), we will also display the return value. So, our final result is

5 3 1 1 3 5 None

11. (3 points)

```
def lulu(jerry):
    if not jerry:
        return 0
    elif jerry[0] in "aeiouAEIOU":
        return 1 + lulu(jerry[1:])
    else:
        return lulu(jerry[1:])

print(lulu("Buttercup"))
```

3

This is recursively counting vowels!

Recursive calls made:

lulu(“Buttercup”) → lulu(“uttercup”)
lulu(“uttercup”) → 1 + lulu(“ttercup”)
lulu(“ttercup”) → lulu(“tercup”)
lulu(“tercup”) → lulu(“ercup”)
lulu(“ercup”) → 1 + lulu(“rcup”)
lulu(“rcup”) → lulu(“cup”)
lulu(“cup”) → lulu(“up”)
lulu(“up”) → 1 + lulu(“p”)
lulu(“p”) → lulu(“”)
lulu(“”) → 0

Working from bottom up:

If lulu(“”) is 0, then we know lulu(“p”) will be 0.
If lulu(“p”) is 0, then we know lulu(“up”) will be 1 + 0 = 1.
If lulu(“up”) is 1, then we know lulu(“cup”) will be 1.
If lulu(“cup”) is 1, then we know lulu(“rcup”) will be 1.
If lulu(“rcup”) is 1, then we know lulu(“ercup”) will be 1 + 1 = 2.
If lulu(“ercup”) is 2, then we know lulu(“tercup”) will be 2.
If lulu(“tercup”) is 2, then we know lulu(“ttercup”) will be 2.
If lulu(“ttercup”) is 2, then we know lulu(“uttercup”) will be 1 + 2 = 3.
If lulu(“uttercup”) is 3, then we know lulu(“Buttercup”) will be 3.

12. (3 points)

```
def fanfiction(trope):
    if len(trope) == 0:
        return ""
    else:
        return fanfiction(trope[1:]) + trope[0]

print(fanfiction("revol2ymene"))
```

enemy2lover

This is reversing a string!

Recursive calls made:
fanfiction("revol2ymene") → fanfiction(“evol2ymene”) + “r”
fanfiction("evol2ymene") → fanfiction(“vol2ymene”) + “e”
fanfiction("vol2ymene") → fanfiction(“ol2ymene”) + “v”
fanfiction("ol2ymene") → fanfiction(“l2ymene”) + “o”
fanfiction("l2ymene") → fanfiction(“2ymene”) + “l”
fanfiction("2ymene") → fanfiction(“ymene”) + “2”
fanfiction("ymene") → fanfiction(“mene”) + “y”
fanfiction("mene") → fanfiction(“ene”) + “m”
fanfiction("ene") → fanfiction(“ne”) + “e”
fanfiction("ne") → fanfiction(“e”) + “n”
fanfiction("e") → fanfiction(“”) + “e”
fanfiction("") → "" *(base case hit)*

Working from bottom up:

If we know fanfiction(“”) is “”, then fanfiction(“e”) is “e”.
If we know fanfiction(“e”) is “e”, then fanfiction(“ne”) is “en”.
If we know fanfiction(“ne”) is “en”, then fanfiction(“ene”) is “ene”.
If we know fanfiction(“ene”) is “ene”, then fanfiction(“mene”) is “enem”.
If we know fanfiction(“mene”) is “enem”, then fanfiction(“ymene”) is “enemy”.
If we know fanfiction(“ymene”) is “enemy”, then fanfiction(“2ymene”) is “enemy2”.
If we know fanfiction(“2ymene”) is “enemy2”, then fanfiction(“l2ymene”) is “enemy2l”.
If we know fanfiction(“l2ymene”) is “enemy2l”, then fanfiction(“ol2ymene”) is “enemy2lo”.
If we know fanfiction(“ol2ymene”) is “enemy2lo”, then fanfiction(“vol2ymene”) is “enemy2lov”.
If we know fanfiction(“vol2ymene”) is “enemy2lov”, then fanfiction(“evol2ymene”) is “enemy2love”.
If we know fanfiction(“evol2ymene”) is “enemy2love” then fanfiction(“revol2ymene”) is “enemy2lover”.

So, the final reversed string is "enemy2lover". (my favorite trope, but I also like slow burn!)

13. (3 points)

```
def peanuts(snoopy, woodstock):
    if woodstock == 0:
        return snoopy
    else:
        return peanuts(woodstock, snoopy % woodstock)

print(peanuts(15, 10), peanuts(12, 18), peanuts(49, 77))
```

5 6 7

This is finding the greatest common divisor between two numbers!

Recursive calls made for peanuts(15, 10):

peanuts(15, 10) → peanuts(10, 5)
peanuts(10, 5) → peanuts(5, 0)
peanuts(5, 0) → 5.
Since peanuts(5, 0) is 5, we know peanuts(10, 5) is 5. Thus, we know peanuts(15, 10) is 5.

Recursive calls made for peanuts(12, 18):

peanuts(12, 18) → peanuts(18, 12)
peanuts(18, 12) → peanuts(12, 6)
peanuts(12, 6) → peanuts(6, 0)
peanuts(6, 0) → 6.
Since peanuts(6, 0) is 6, we know peanuts(12, 6) is 6. Thus, we know peanuts(18, 12) is 6, and peanuts(12, 18) is 6.

Recursive calls made for peanuts(49, 77):

peanuts(49, 77) → peanuts(77, 49)
peanuts(77, 49) → peanuts(49, 28)
peanuts(49, 28) → peanuts(28, 21)
peanuts(28, 21) → peanuts(21, 7)
peanuts(21, 7) → peanuts(7, 0)
peanuts(7, 0) → 7.

Since peanuts(7, 0) is 7, we know peanuts(21, 7), peanuts(28, 21), peanuts(49, 28), peanuts(77, 49), and peanuts(49, 77) are all 7.

14. (3 points)

```
def mystery(gang):
    clues = []
    for who in gang:
        if type(who) == list:
            clues.extend(mystery(who))
        else:
            clues.append(who)
    return clues

inc = ["scooby", ["shaggy", ["daphne", "velma"]], ["fred"]]
print(mystery(inc))
```

['scooby', 'shaggy', 'daphne', 'velma', 'fred']

This recursive function 'flattens' nested lists!

- Iteration one in loop – item: "scooby"
  - "Scooby" is not a list, so the "else" triggers – "scooby" is simply appended to clues. Current clues: ["scooby"]
- Iteration two – item: ["shaggy, ["daphne", "velma"]]
  - This item is a list, so the 'if' triggers. We recursively call mystery() on this item and extend the result to clues (which is currently only ["scooby"]).
    - **mystery(["shaggy", ["daphne", "velma"]])**
      - clues = []
      - Iteration one in loop – item: "shaggy"
        - "shaggy" is not a list, so it is appended to clues. clues: ["shaggy"]
      - Iteration two in loop – item: ["daphne", "velma"]
        - This item is a list, so the "if" triggers. We recursively call mystery() on this item, and we extend clues with this result.
          - **mystery(["daphne", "velma"])**
            - clues = []
            - Iteration one in loop – item: "daphne":
              - "daphne" is not a list, so the "else" triggers – it is appended to clues. clues: ["daphne"]
            - Iteration two in loop – item: "velma":
              - "velma" is not a list, so the "else" triggers – it is appended to clues. clues: ["daphne", "velma"]
            - return clues – ["daphne", "velma"]
        - We obtain ["daphne", "velma"], extend it to clues. clues is now ["shaggy", "daphne", "velma"]
      - We return ["shaggy, daphne, "velma"].
  - clues.extend(["shaggy", "daphne", "velma"]) will be ["scooby", "shaggy", daphne, "velma"]

- Iteration three – item: [“fred”]
    - This item is a list, so the “if” triggers. We recursively call mystery() on this item and extend it to clues.
        - **mystery([“fred”])**
            - clues = []
            - Iteration one in loop – item: “fred”
                - “fred” is not a list, so the “else” triggers – “fred” is appended to clues. clues: [“fred”]
            - return clues – [“fred”]
    - We obtain [“fred”] and extend it to clues. clues is now [“scooby”, “shaggy”, daphne, “velma”, “fred”].
- Our loop finishes. We return [“scooby”, “shaggy”, daphne, “velma”, “fred”]

15. (3 points)

```
def girlPower( arr, l, r, x):
   if r < l:
      return -1
   if arr[l] == x:
      return l
   if arr[r] == x:
      return r
   return girlPower(arr, l+1, r-1, x)

princesses = ["cinderella", "snowWhite", "moana", "jasmine", "aurora", \
            "merida", "anna", "elsa"]
print(girlPower(princesses, 0, len(princesses) - 1, princesses[3]))
```

3

This function is a recursive implementation of linear search! arr is the list to search through. L is our starting (left) index, R is our ending (right) index, and x is the item to search for.

This implementation of linear search searches both the front and back of the list each iteration. We’ll compare the first and last items, checking to see if either is x. If we don’t find it, we increase L by one (so we move our “pointer” at the front of the list to the right), and we decrease R by one (so we move our “pointer” at the end of the list to the left).

So we have three base cases: either the item is at the left pointer (arr[l] == x), right pointer (arr[r] == x), or, we’ve checked all elements – in this case, R will be less than L (because we continuously increase L but decrease R). In this case, we’ll return -1 to indicate we did not find x in our list.

In this example, we give it the princess list to search through. Our start index is the beginning of the list, 0, our end index is the last item of the list (len(princesses) - 1, in this case 7), and we specify “jasmine” (princesses[3]) for the princess to find.

girlPower(princesses, 0, 7, “jasmine”)

- princesses[0] and princesses[7] are “cinderella” and “elsa”. We don’t find “jasmine”, so we shift our pointers and call girlPower(princesses, 1, 6, “jasmine”)

girlPower(princesses, 1, 6, “jasmine”)

- princesses[1] and princesses[6] are “snowWhite” and “anna”. We don’t find “jasmine”, so we shift our pointers and call girlPower(princesses, 2, 5, “jasmine”)

girlPower(princesses, 2, 5, “jasmine”)

- princesses[2] and princesses[5] are “moana” and “merida”. We don’t find “jasmine”, so we shift our pointers and call girlPower(princesses, 3, 4, “jasmine”)

girlPower(princesses, L =3, R = 4, “jasmine”) → 3 *(one base case triggered)*

- We find “jasmine” when testing princesses[L], so we return L, which is 3.

This value of 3 is returned back to girlPower(princesses, 2, 5, “jasmine”), which returns it back to girlPower(princesses, 1, 6, “jasmine”), which returns it back to girlPower(princesses, 0, 7, “jasmine”).

16. (3 points)

```
def spookyMath(scaryNums):
    if scaryNums == []:
        return 0
    elif scaryNums[0] % 2 == 0:
        return -scaryNums[0] + spookyMath(scaryNums[1:])
    else:
        return scaryNums[0] + spookyMath(scaryNums[1:])

scaryNums = [7, -42, 17, -99, 2, -8]
print(spookyMath(scaryNums))
```

-27

This function subtracts all even numbers in the list, and adds all the odd numbers! In the following traceback, note that when we subtract a negative number, we're essentially adding it (and when we add a negative number, we're essentially subtracting it).

Recursive calls made:
spookyMath([7, -42, 17, -99, 2, -8]) → 7 + spookyMath([-42, 17, -99, 2, -8])
spookyMath([-42, 17, -99, 2, -8]) → 42 + spookyMath([17, -99, 2, -8])
spookyMath([17, -99, 2, -8]) → 17 + spookyMath([-99, 2, -8])
spookyMath([-99, 2, -8]) → -99 + spookyMath([2, -8])
spookyMath([2, -8]) → -2 + spookyMath([-8])
spookyMath([-8]) → 8 + spookyMath([])
spookyMath([]) → 0

Working from bottom up:

If we know spookyMath([]) is 0, then spookyMath([-8]) is 8 + 0 = 8
If we know spookyMath([-8]) is 8, then spookyMath([2, -8]) is -2 + 8 = 6
If we know spookyMath([2, -8]) is 6, then spookyMath([-99, 2, -8]) is -99 + 6 = -93
If we know spookyMath([-99, 2, -8]) is -93, then spookyMath([17, -99, 2, -8]) is 17 + -93 = -76
If we know spookyMath([17, -99, 2, -8]) is -76, then
spookyMath([-42, 17, -99, 2, -8]) is 42 + -76 = -34
If we know spookyMath([-42, 17, -99, 2, -8]) is -34, then
spookyMath([7, -42, 17, -99, 2, -8]) is 7 + -34 = -27

17. (3 points)

```
def toyStory(woody, buzz):
    if buzz == 0:
        return 1
    elif buzz % 2 == 0:
        jessie = toyStory(woody, buzz // 2)
        return jessie * jessie
    else:
        return woody * toyStory(woody, buzz - 1)

print(toyStory(3, 4), toyStory(4, 3))
```

81 64

This function recursively calculates the power of a number, but only by squaring. That is to say, in toyStory(3, 4) we calculate 3^4 by squaring 3^2 as many times as needed.

Recursive calls made by toyStory(3, 4):

toyStory(3, 4)

- Because buzz (4) % 2 == 0, the ‘elif’ triggers. We call toyStory(3, 2) & square the result.
    - toyStory(3, 2)
        - Because buzz (2) % 2 == 0, the ‘elif’ triggers. We call toyStory(3, 1) and square the result.
            - toyStory(3, 1)
                - buzz (1) is not 0, but buzz % 2 is also not 0. So the ‘else’ triggers. We return
                  woody * toyStory(woody, buzz -1).
                  In this case, that is: 3 * toyStory(3, 0)
                    - toyStory(3, 0)
                        - buzz == 0, so we return 1
                - toyStory(3, 0) returned 1, so the result of toyStory(3,1) is 3 * 1 = 3.
            - We get back 3 from toyStory(3, 1). This result squared is 9.
        - toyStory(3, 2) returns 9.
- We get back 9 from toyStory(3, 2). We square this and obtain 81.
    - 81 is our final answer.

toyStory(4, 3)

- Because buzz (3) is not 0, but buzz % 3 is also not 0, the “else” triggers. We return woody * toyStory(woody, buzz - 1). That is, 4 * toyStory(4, 2)
    - toyStory(4, 2)
        - Because buzz % 2 == 0, the ‘elif’ triggers. We call toyStory(4, 1) and square the result.
            - toyStory(4, 1)
                - buzz (1) is not 0, but buzz % 2 is also not 0. So the ‘else’ triggers. We return
                  woody * toyStory(woody, buzz - 1)
                  In this case, that is: 4 * toyStory(4, 0)
                    - toyStory(4, 0)
                        - buzz == 0, so we return 1
                - toyStory(4, 0) returned 1, so the result of toyStory(4, 1) is 4 * 1 = 4.
            - We get back 4 from toyStory(4, 1). This result squared is 16.
        - toyStory(4, 2) returns 16.
- We multiply 4 * toyStory(4, 2), or 4 * 16. The final result is 64.