Worksheet #10 More on Lists Answer Key =)

TRUE/FALSE

1. (10 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) The expression `myList[2][3]` is valid for a nested list with three inner lists, each having at least four elements.

   (b) In a linear search, if the specified item is found, the search algorithm returns the index of the found item plus 1 to account for 0-based indexing.

   (c) Iterating through a nested list requires nested loops to access each individual element.

   (d) Binary search is guaranteed to return the index of the first instance of an item in a list.

   (e) Lists within a list can have a different length from each other.

   (f) In selection sort, the minimum element is repeatedly selected from the unsorted part of the list and swapped with the first element, while in insertion sort, elements are compared and inserted into their correct positions in the sorted part of the list.

   (g) The `len()` function can be used to find the total number of elements in a nested list.

   (h) When using the `.sort()` method on nested lists in Python, the sorting is applied only to the outermost list, leaving the inner lists unaltered.

   (i) In a linear search, one item is checked in the unexplored portion of the list each step, while in binary search, the search space is cut in half with each step.

   (j) In binary search, the number of "searches" required is at most half the length of the list.

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

Notes:

(A) True! Index 2 is the third item in myList, and index 3 of that gets us the fourth element.

(B) This is nonsensical! Our search algorithm should return the index itself, so that way we know where the item is

in our list. Returning the index + 1 simply means we're getting the position of the item in a 1-based (instead of 0-based) indexing system.
(D) False! Refer to Q2 of multiple choice answer explanations.
(J) False! Binary search halves the search space, but this occurs each iteration. Because we repeatedly halve the length of the list, binary search will only require up to log_2(list length) searches. This helps!: https://www.khanacademy.org/computing/computer-science/algorithms/binary-search/a/running-time-of-binary-search

MULTIPLE CHOICE

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

2. In which scenario would using binary search be more advantageous than linear search?

   A. Searching through a small, unsorted list.
   B. Looking for the first occurrence of an item in a list.
   C. Searching through a large, sorted list.
   D. Finding the maximum value in an unsorted list.
   E. None of the above.

Correct answer explanation:
(C) Yes! Binary search is more efficient than linear search, but it requires a sorted list. But if we knew the list was sorted, then binary search would be a better choice than linear search, especially on larger lists (because linear search would check each and every item, whereas binary search halves the search space each iteration).

Wrong answer explanations:
(A) Binary search requires a sorted list, because the algorithm compares the target value to the middle element of the list and eliminates half of the remaining elements based on this comparison, repeating this process until the target value is found or the search space is empty.
(B) Binary search is not guaranteed to get us the first occurrence of an item in a list, because it checks the middle index first. For example: if we provide the list [5, 5, 5] to binary search and specify that we want to find 5, then we will get back index 1 instead of index 0.

(D) With linear and binary search, we provide the item we want the index of. (D) implies we would simply get the max value back, instead of an index. In this scenario, it'd be preferable to use max().
(E) because (C) is correct, (E) cannot be true.

3. How many iterations does a linear search require to find the value 616 in the list `[9, 10, 21, 41, 98, 123, 364, 616, 1218]`?

A. 8 B. 7 C. 6 D. 9

Correct answer explanation:
(A) Linear search simply checks the items in our list, left to right, in order to find a specific item. Because 616 is the eighth item in our list, a linear search will require eight iterations to find it.

4. How many iterations does a binary search require to find the value 616 in the list `[9, 10, 21, 41, 98, 123, 364, 616, 1218]`?

A. 1 B. 2 C. 3 D. 4 E. 2 or 3

Correct answer explanation:
(E): Binary search compares the item we're looking for, to the middle of the list, to decide whether we need to cut the lower half or upper half of the list. To calculate the middle of the list (let's call this our pivot), we add the index of our start (0) and our end (8) and divide by 2. This means our pivot starts at index 4 – this is 98 in our list.

We compare 616 to 98 – we are over, so we eliminate 98 and below. We set our 'start' to one after our previous middle. This would set our start to be index 5. We then recalculate the middle. Our start is index 5, our end is index 8. 5 + 8 = 13, divided by 2 results in either 6 (round down) or 7 (round up) for our middle.

If our binary search rounds up the middle when there is a decimal, middle will be index 7, then we compare 616 to 616 and see we have a match. This would be the second iteration.

If our binary search rounds down the middle when there is a decimal, middle is index 6, which will be 364 in our list. We compare 616 to 364 – we are still over. So, we eliminate 364 and below. We set our 'start' to one index after our previous middle, which was index 6 – so now our 'start' is 7.

Using a start of 7 and end of 8, we get 7.5 for our middle – we know our binary search rounds

down, so the middle will be 7, which is 616. Now, we will compare 616 to 616, and see we have a match. This would be the third iteration.

5. What will the list `[86485, 42, 1337, 404, 777, 9000, 24601]` look like after three iterations of the selection sort algorithm?

   A. `[42, 404, 777, 1337, 9000, 24601, 86485]`
   B. `[42, 404, 777, 86485, 1337, 9000, 24601]`
   C. `[42, 404, 1337, 86485, 777, 9000, 24601]`
   D. None of the above.

Correct answer explanation:
(B) Selection sort works by finding the minimum of the list and swapping it with the element at the front, then finding the next minimum and swapping it with the element at the second-most front, etc. So, we will find the first minimum of the list, which is 42, and swap it with the front, which is 86485. Our list now is [42, 86485, 1337, 404, 777, 9000, 24601].

We then find the second minimum of the list, which is 404, and swap it with the item at the second-most front position (86485). Our list now is [42, 404, 1337, 86485, 777, 9000, 24601].

We then find the third minimum of the list, which is 777, and swap it with the item at the third-most front position (1337). Our list is now [42, 404, 777, 86485, 1337, 9000, 24601], which matches with (B).

6. What will the list `[86485, 42, 1337, 404, 777, 9000, 24601]` look like after three iterations of the insertion sort algorithm?

   A. `[42, 404, 777, 1337, 9000, 24601, 86485]`
   B. `[42, 1337, 86485, 404, 777, 9000, 24601]`
   C. `[42, 404, 1337, 86485, 777, 9000, 24601]`
   D. None of the above.

Correct answer explanation:
(C) Insertion sort will start by comparing the first two elements of the list, and swapping them to be sorted. This means we will compare 86485 and 42 – we swap 42 to be in front. Iteration one, done! Our list is currently [42, 86485, 1337, 404, 777, 9000, 24601].

We then consider the next item in the list, 1337. We will compare it with our two sorted elements from before. 1337 is greater than 42 but less than 86485, so it will go between them. Our list is now [42, 1337, 86485, 404, 777, 9000, 24601]. Iteration two, check!

We now consider 404. It is greater than 42, but less than 1337, so we place it there. Our list becomes [42, 404, 1337, 86485, 777, 9000, 24601], which matches (C).

7. How do you remove `"dumbo"` from the following nested list?:

```
100acreWood = [["pooh", "tigger", "piglet"], ["eeyore", "rabbit", "owl",
"dumbo"], ["kanga", "roo", "gopher"]]
```

A. `100acreWood.remove("dumbo")`
B. `100acreWood.pop(1)`
C. `100acreWood[1].remove("dumbo")`
D. `100acreWood[1].pop("dumbo")`

(fun fact – 100acreWood is actually an invalid name, because it starts with a number. I should fix this LOL)

Correct answer explanation:
(C) yes! We index 1 into 100acreWood to get the list [“eeyore”, “rabbit”, “owl”, “dumbo”], and we call .remove(“dumbo”) on this list. This properly removes “dumbo” from the inner list.

Wrong answer explanations:
(A) “dumbo” is not a direct item of 100acreWood, because it is in one of the inner lists. .remove() checks the items of 100acreWood, to see if any are equal to “dumbo”, and it will remove the first match. .remove() won’t find a match, so this causes an error..
(B) This would remove the item at index 1 of 100acreWood, which would be the whole list [“eeyore”, “rabbit”, “owl”, “dumbo”]. We only want to remove “dumbo”.
(D) pop() takes in an index as a parameter, not the item itself.

8. How do you check if the string `"mew"` is present in the following nested list?:

```
myTeam = [["eevee", "togepi"], ["mew", "mewtwo"], ["dragonite", "gengar"]]
```

A. `"mew" in` [redacted] myTeam
B. `True if "mew" in [item for item in` [redacted] `] else False`
C. `"mew" in` [redacted] myTeam[1]
D. `"mew" in` [redacted] myTeam[1:1]

This is a fun question!

Correct answer explanation:
(C) Yes! We get the item at index 1 of myTeam ([“mew”, “mewtwo”]). And then we check to see if “mew” is in this list – this will evaluate to True.

Wrong answer explanations:
(A) “mew” is an item in one of the inner lists of myTeam – it is not a direct item of myTeam, so the expression “ ‘mew’ in myTeam ” will be False.
(B) [item for item in myTeam] will simply recreate myTeam. So, because the expression “ ‘mew’ in myTeam ” is False, the expression “True if ‘mew’ in [item for item in myTeam] else False ” will also be False.
(D) myTeam[1:1] will be an empty list, because we specify the same starting and ending index. So, there is no way “mew” will be in myTeam[1:1].

9. On a list that is already mostly sorted, which sorting algorithm is likely to perform worse, selection sort or insertion sort?

   A. Insertion sort, because it will cause unneeded swaps as it sorts through the mostly sorted list.
   B. Selection sort, because it will unnecessarily search for minimum elements, which are likely to be in the correct place already in a mostly sorted list.
   C. The two algorithms will perform exactly the same.
   D. This cannot be determined from the given information.

Correct answer explanation:
(B) Yes! Selection sort will repeatedly scan through the list to find the minimum element to place at the front. For example, on list [1, 2, 3, 4, 5, 6, 8, 7], it will scan all 4 items to find the minimum (1) to try to place it at the front, even though it is already there. Then it scans through the next 7 items to find the next minimum (2) to place, even though 2 is already sorted, and so on…

In contrast, insertion sort starts at the front of the list and will decide – 1 or 2, which is the minimum? 1 is, but it’s already sorted. Neat. So then insertion sort looks at 2 or 3 – minimum is 2, but it’s already sorted. Then, 3 or 4 – 3 is lower, but it’s already in-place. 4 or 5 – 4 is lower, already sorted. Etc.

Note that a question like this would NOT pop up on the exam. This is quite advanced. But for you kiddos who want to make SURE that you understand the sorts, or you kiddos going into CS 313E, here you go! =)

Wrong answer explanations:
(A) Not true. Neither algorithm would cause unneeded swaps – the only difference is how they go about selecting and or placing elements.
(C) Not true! The algorithms take different approaches to sorting, so one must be more efficient than the other in some cases.
(D) Not true!

TRACING

10. (3 points)

```
queensLists = [["ariel"], ["belle", "tiana"], \
               ["mulan", "pocahantas", "rapunzel"]]
result = [homegirl for lst in queensLists for homegirl in lst]
print(result)
```

['ariel', 'belle', 'tiana', 'mulan', 'pocahontas', 'rapunzel']

A double list comprehension!!! GASP!

I think list comprehension is easier to read from left to right, starting at the first 'for'. And we can nest each expression. So our first expression is 'for lst in queensList', our next is 'for homegirl in lst'. So that will be

for lst in queensList
—for homegirl in lst
——homegirl

lst will refer to the items in queensLst – that is, ["ariel"], ["belle", "tiana], ["mulan", "pocahontas", "rapunzel"]. Those are all the items in queensLst.

For each lst, we go through their items (each item is referred to as homegirl). In our list comprehension, we simply have 'homegirl'. So, that will just be the items in lst, with no modification made.

So, our new list called result will have all the individual items from the nested lists in queensLists. So, essentially, queensList was simply 'flattened', and we get back ["ariel", "belle", "tiana", "mulan", "pocahontas", "rapunzel"].

11. (3 points)

```
inputNums = [13, 17, 25, 28, 10, 30, 21]
myLst = [ [0, 0, 0], [0, 0, 0], [0, 0, 0] ]

for num in inputNums:
    result1 = num % 3
    result2 = (num // 10) % 3
    myLst[result1][result2] += 1
print(myLst)
```

[[1, 0, 1], [0, 2, 2], [0, 1, 0]]

For each number in the list inputNums, we calculate two things: num % 3 (stored as result1), and (num // 10) % 3 (stored as result2). We then use result1 and result2 as indexes into myLst (a nested list – result1 tells us which list, result2 tells us which index), and we increment that corresponding item.

inputNum number	result1 (num % 3)	result2 (num // 10) % 3	myLst
13	1	1	[[0, 0, 0,], [0, **1**, 0,], [0, 0, 0,]]
17	2	1	[[0, 0, 0,], [0, 1, 0,], [0, **1**, 0,]]
25	1	2	[[0, 0, 0,], [0, 1, **1**,], [0, 1, 0,]]
28	1	2	[[0, 0, 0,], [0, 1, **2**,], [0, 1, 0,]]
10	1	1	[[0, 0, 0,], [0, **2**, 2,], [0, 1, 0,]]
30	0	0	[[**1**, 0, 0,], [0, 2, 2,], [0, 1, 0,]]
21	0	2	[[1, 0, **1**,], [0, 2, 2,], [0, 1, 0,]]

Thus, our final result is [[1, 0, 1], [0, 2, 2], [0, 1, 0]].

12. (3 points)

```
powerpuff = [["blossom", "mojoJojo"], \
             ["bubbles", "utonium"], \
             ["buttercup", "bunny"]]
powerpuff.sort()
print(powerpuff)
```

[['blossom', 'mojoJojo'], ['bubbles', 'utonium'], ['buttercup', 'bunny']]

When sorting nested lists, Python compares the first item of each list to each other, considering only later items in case of a tie. In our case, the first items are "blossom", "bubbles," and "buttercup". This essentially boils down to a string comparison. Among the three, "blossom" has the lowest lexicographic order, so the list containing "blossom" will be the first in the sorted list. Although "bubbles" and "buttercup" share the same first two characters, when comparing the third, 'b' in "bubbles" is less than "t" in "buttercup". Consequently, "bubbles" has a lower lexicographic order than "buttercup", so we place the list with "bubbles" in it next, followed by the list with "buttercup".

13. (3 points)

```
lst1 = [[1, -100], [5, 10]]; lst2 = [[1, -900], [999, 999]]
print(lst1 > lst2)
```

True

When comparing nested lists to find which is greater, Python will compare the items at each index. Thus, the first comparison it makes will be between [1, -100] and [1, -900]. These two items are equal to each other in their first index, so we look at the second index. The first list is greater than the second here (-100 > -900), which means that ls1 > ls2 will evaluate to True.

14. (3 points)

```
board = [["H", "X", "Z", "C"], ["A", "O", "T", "J"], \
         ["K", "P", "N", "D"], ["L", "M", "U", "K"]]

for i in range(len(board)):
    for j in range(len(board)):
        if i == j:
            print(board[i][j], end = "")
```

HONK

Because we have a double for loop, both going from 0 to 3 (because range(len(board)) is simply range(4) → [0, 1, 2, 3]), this code snippet iterates over all rows and columns of board. Our condition is 'if i == j', so this will print when i and j are both 0, 1, 2, and 3. We print board[0][0], board[1][1], board[2][2], and board[3][3], which are "H", "O", "N", and "K".

15. (3 points)

```
sanrio = [["keroppi", "helloKitty"], \
          ["pompompurin", "myMelody", "cinnamoroll"], \
          ["chococat", "badtzmaru"]]
cuties = [group[0] for group in sanrio]
print(cuties)
```

['keroppi', 'pompompurin', 'chococat']

This is list comprehension! Again, we can maybe 'read' this easier if we read starting from the first 'for'.

for group in sanrio
—group[0]

So, for each item (referred to as 'group') in the list sanrio, we will index into it at position[0]. This will refer to "keroppi", "pompompurin", and "chococat" – these items are all put into a new list from our list comprehension.

16. (3 points)

```
looney = [["bugs", "daffy"], [1930, 4], ["tweety", "sylvester", "lola"]]
print(looney[-1][-2][0])
```

s

looney[-1] will get us the last item in the looney list. This will be [“tweety”, “sylvester”, “lola”]. Indexing [-2] will get us back “sylvester”. This item can be indexed, because it is a string, so indexing [0] will get us back the first character, “s”.

17. (3 points)

```
myNums = [[1, 2, 3, 4], [5, 6, 7, 8], [9, 10, 11, 12]]
total = 0
for i in range(len(myNums)):
    for j in range(4):
        if not j % 2:
            total += myNums[i][j]
print(total)
```

36

This question looks daunting, but I promise that it isn’t!

Let’s walk through it, step by step. Our double for loop will simply iterate through each individual item in myNums. Our outer loop is range(len(myNums)), or range(3) → [0, 1, 2]. Our inner loop is range(4) → [0, 1, 2, 3]. And each iteration, we test the condition ‘if not j % 2’. j%2 gets us either 0 or 1, and we use this result as a boolean. When we apply ‘not’ to 0 (False), we get 1 (True), meaning that this ‘if not j %2’ condition will trigger only when j % 2 evaluates to 0.

if not j % 2
if not 0
if not False
if True

So, for each item in our list, we check to see if its index % 2 is 0. If so, we add it to total. Because j can take on values [0, 1, 2, 3], the only numbers that get us 0 when we do % 2 will be 0 and 2. So throughout the loop, we simply sum the items in the inner lists that are at index 0 or 2. This is 1 + 3 + 5 + 7 + 9 + 11, which is 36.