socrative

MergeSort Part 2 Quiz Score:

1. **The exact opposite of concatenation (i.e. joining, or merging) is**

A) Appending
B) Inserting
C) Cutting, Splitting, Slicing
D) Deleting

2. **Slicing** (□□□□□□) **is the opposite of Concatenating.**

A) True
B) False

3. **To divide a string into multiple pieces, we use .split() method.**
To divide a list into one or more pieces, we use slice operator ([ ]).

A) True
B) False

4. **HEAD PROBLEM**
al = [5, 6, 7, 8]
To get the "head" of alist, we can use the slice al[:1].
What will be the value of it?

A) [5]
B) [5, 6, 7]
C) [ ] (list with no elements)
D) 5
E) None of the above

5. **With reference to HEAD PROBLEM, Option A and Option D are the same.**

A) True
B) False

6. al = [1, 2, 3, 4].
To rotate **al** left is to end up with **al = [2, 3, 4, 1]**.
That is, take the head element and place it at the end of the list.
What is the code which will make this happen?

A) **al = [2, 3, 4, 1]**

B) **al = [2, 3, 4] + [1]**

C) **al = al[1:] + al[:1]**

D) **al[:] = al[1:] + al[:1]**

E)
```
head = al[:1]
rest = al[1:]
al = rest + head
```

F)
```
head = al.pop(0)
al.append(head)
```

G) None of the above

7. HEAD AND REST PROBLEM

al = [1, 2, 3, 4].
To divide al into **'head' ([1]) and 'rest' ([2, 3, 4])**, what are the slices to use?

A) **al[0] , al[0:]**

B) **al[:1], al[1:]**

C) **al[0], al[1:]**

D) None of the above

8. With reference to **HEAD AND REST PROBLEM**,
compare the output of Option C with Option B.
There are at least two reasons why one is more preferable. What are they?

9. al = [1, 2, 3, 4]
What is the result of **al[:2]?**

A) [1, 2]

B) [1, 2, 3]

C) [ ] (list with no elements)

D) None of the above

10. If **al** = [1, 2, 3, 4], then **al[2:]** is equal to

A [2, 1]

B [3, 4]

C [2, 3, 4]

D [2]

E None of the above

11. al = [1, 2, 3, 4]
What is the result of **al[:2] + al[2:]**?

A [1, 2]

B [1, 2, 3]

C [2, 2]

D [1, 2, 3, 4]

E None of the Above

12. □□□□□□□ : □□□□
-> CUTTING using [] : JOINING using '+'
-> SLICING : CONCATENATING
-> RECURSIVE SLICING : RECURSIVE MERGING
-> **DIVIDE** : **CONQUER**
-> □□□□ : □□□□□□□
MERGE SORT

A True

B False

வெட்டு : *சேர்*
-> CUTTING using [] : JOINING using '+'
-> SLICING : CONCATENATING
-> RECURSIVE SLICING : RECURSIVE MERGING
-> **DIVIDE** : **CONQUER**
-> *பிரி* : *அடக்கு*
MERGE SORT

13. The value of **mid** in the given code snippet is

A 0

B 1

C 2

D 1.5

```
2   al = [1, 2, 3, 4]
3   mid = len(al) // 2
4   newlist = al[:mid] + al[mid:]
5   assert newlist == al
```

14. If you replace Line 3 with **mid = int(len(al)/2)**, the value of **mid** will remain the same.

A True

B False

```
2   al = [1, 2, 3, 4]
3   mid = len(al) // 2
4   newlist = al[:mid] + al[mid:]
5   assert newlist == al
```

15. **newlist** will ***not*** contain the same number of elements as **al**.

A True

B False

```
2   al = [1, 2, 3, 4]
3   mid = len(al) // 2
4   newlist = al[:mid] + al[mid:]
5   assert newlist == al
```

16. **The assertion in Line 5 will not produce an error.**

A) True

B) False

```
2  al = [1, 2, 3, 4]
3  mid = len(al) // 2
4  newlist = al[:mid] + al[mid:]
5  assert newlist == al
```

17. **Line number 14 will produce what output?**

A) [1, 2], [1, 2]

B) [1, 2], [4, 5]

C) [1, 2], [3, 4, 5]

D) None of the above

```
11  al = [1, 2, 3, 4, 5]
12  mid = len(al) // 2
13  left, right = al[:mid], al[mid:]
14  print(left, right)
```

18. **The list al is an example of nested list. It has a length of**

A) 5

B) 2

C) 1

D) 4

```
al = [1, [2, [3, [4, [5, None]]]]]
```

19. **The printRec is a valid recursive function and it has one terminal case.**

A) True

B) False

```
13  al = [1, [2, [3, [4, [5, None]]]]]
14  def printRec (alist):
15    if not alist[1]:
16      print(alist[0], end=".\n")
17      return
18
19    print(alist[0], end=", ")
20    printRec(alist[1])
21
22  printRec(al)
```

20. **The line number 22 will produce what output?**

A) **1, 2, 3, 4, 5.**

B) **5, 4, 3, 2, 1.**

C) None of the above.

```
13  al = [1, [2, [3, [4, [5, None]]]]]
14  def printRec (alist):
15    if not alist[1]:
16      print(alist[0], end=".\n")
17      return
18
19    print(alist[0], end=", ")
20    printRec(alist[1])
21
22  printRec(al)
```

21. **The sum function is**

A) a **recursive** function but not a fruitful function

B) **a recursive** and **fruitful** function

C) non-recursive function

D) none of the above

```
16  def sum(alist):
17    if not alist:
18      return 0
19    if len(alist) == 1:
20      return alist[0]
21
22    remaining = alist[1:]
23    return alist[0] + sum(remaining)
24
25  print(sum([1, 2, 3, 4, 5]))
```

22. The **sum** function has one terminal case.

A True

B False

```
16  def sum(alist):
17    if not alist:
18      return 0
19    if len(alist) == 1:
20      return alist[0]
21
22    remaining = alist[1:]
23    return alist[0] + sum(remaining)
24
25  print(sum([1, 2, 3, 4, 5]))
```

23. The recursive **some_func** has only one terminal case.

A True

B False

```
27  def merge(A, B):
28    return [
29      (A if A[0] < B[0] else B).pop(0)
30      for _ in A+B if A and B
31      ] + A + B
32
33  def some_func(ulist):
34    if len(ulist) < 2:
35      return ulist
36
37    mid = len(ulist)//2
38    left = ulist[:mid]
39    right = ulist[mid:]
40
41    sorted_left = some_func(left)
42    sorted_right = some_func(right)
43    print(sorted_left, sorted_right)
44
45    slist = merge(sorted_left, sorted_right)
46    return slist
47
48  print(some_func([5, 0, 2, 1, 3, 4]))
```

24. The output from Line 48 will be **[5, 4, 3, 2, 1, 0]**

A True

B False

```
27  def merge(A, B):
28    return [
29      (A if A[0] < B[0] else B).pop(0)
30      for _ in A+B if A and B
31      ] + A + B
32
33  def some_func(ulist):
34    if len(ulist) < 2:
35      return ulist
36
37    mid = len(ulist)//2
38    left = ulist[:mid]
39    right = ulist[mid:]
40
41    sorted_left = some_func(left)
42    sorted_right = some_func(right)
43    print(sorted_left, sorted_right)
44
45    slist = merge(sorted_left, sorted_right)
46    return slist
47
48  print(some_func([5, 0, 2, 1, 3, 4]))
```

25. The intermediary output caused by Line 43 will be as shown here.

A True

B False

```
[0] [2]
[5] [0, 2]
[3] [4]
[1] [3, 4]
[0, 2, 5] [1, 3, 4]
```

26. The most appropriate name that can replace **some_func()** is

A **insertion_sort**

B **selection_sort**

C **merge_sort**

D **histogram**

```
27  def merge(A, B):
28    return [
29      (A if A[0] < B[0] else B).pop(0)
30      for _ in A+B if A and B
31      ] + A + B
32
33  def some_func(ulist):
34    if len(ulist) < 2:
35      return ulist
36
37    mid = len(ulist)//2
38    left = ulist[:mid]
39    right = ulist[mid:]
40
41    sorted_left = some_func(left)
42    sorted_right = some_func(right)
43    print(sorted_left, sorted_right)
44
45    slist = merge(sorted_left, sorted_right)
46    return slist
47
48  print(some_func([5, 0, 2, 1, 3, 4]))
```

27. Assume **al =** [2, 1, 4, 3, 6, 5, 8, 7] and we want to mergesort it.
After the first conquer step, we will have **[1, 2], [3, 4]**, and **[5, 6], [7, 8].**

On visual examination, it is obvious all that needs to be done is merge the sublists to get the sorted list.

This illustrative example can be a source for inspiration to improve the **merge algorithm** so that it can be very efficient when dealing with almost sorted lists. What will you do?

28. Modify the **mergesort** algorithm to eliminate duplicate elements during the process of sorting.
If **al = [4, 5, 6, 1, 2, 1, 2, 3],** then after the sorting is complete the result must be al = [1, 2, 3, 4, 5, 6].