CS 1110 Final Exam Solutions, May 2023

1. [12 points] **Strings.** Implement the following function. Do not use iteration in your solution.

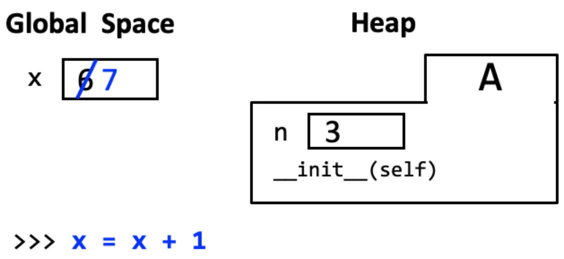
```
def happify(s):
    """
    Returns a new string, that is the "happy version" of `s`

    the happy version is a copy of `s` with the following changes:
    1. the second word is replaced with the string `LOVE`
    2. the last character of the third word is replicated twice
        (this character can be a letter or punctuation)
    3. if it doesn't already, the string should end in an exclamation point

    Examples:
    happify('i like you lots!')       returns 'i LOVE youuu lots!'
    happify('i like you. lots')       returns 'i LOVE you... lots!'
    happify('I hate taking exams.')   returns 'I LOVE takinggg exams.!'
    happify('a e i o!')               returns 'a LOVE iii o!'

    Preconditions:
        `s` is a string of at least four words separated by spaces
        the only whitespaces in `s` are single spaces (' ')
    """
    # STUDENTS: assume preconditions are met. No need to assert them.

    if s[-1] != '!':
        s = s + "!" # take care of change 3 up front

    # Solution with split & join
    slist = s.split(" ")
    slist[1] = 'LOVE'
    dup = slist[2][-1]
    slist[2] = slist[2]+dup+dup
    result = " ".join(slist)
    return result

    # Solution with index and cocatenation
    space1 = s.index(" ")
    space2 = s.index(" ",space1+1)
    space3 = s.index(" ",space2+1)
    dup = s[space3-1]
    result = s[0:space1+1] + "LOVE" + s[space2:space3]+dup+dup+s[space3:]
    return result
```

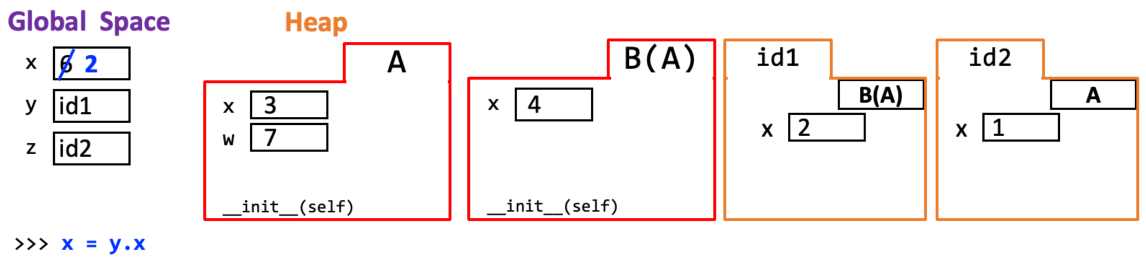
2. [12 points] **Dictionaries.** Implement this function according to its specification using any (and only) tools you learned in CS 1110.

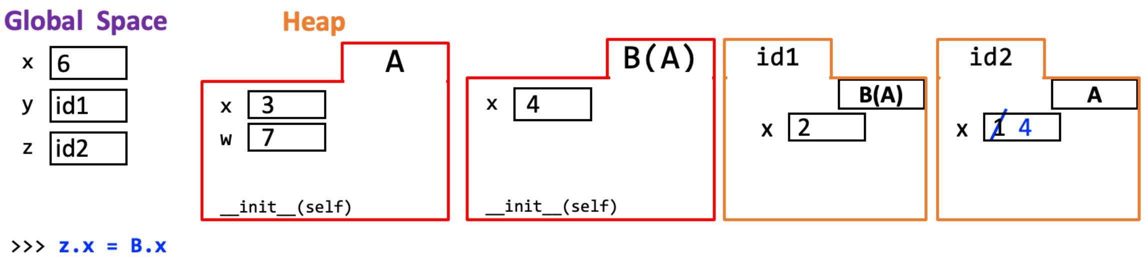
```
def merge_dict(input_dict):
    """ Given an `input_dict` with the following properties:
          - keys are strings of repeating letters ('aaa' or 'bbbbbbbbbbb')
          - values are ints

    Returns: a new dictionary that is the merged version of the input_dict:
    - the keys are the 1 character version of input_dict's keys
    - the values are the combined values across all merged entries,
          weighted by the number of characters in the original key

    EXAMPLES:
       {'a': 6, 'aa': 5, 'aaa': 4} → {'a':28}
         1 x 6 +  2 x 5 +   3 x 4        =28
       {'bb':6 , 'aa': 5, 'aaa': 4} → {'b':12, 'a':22}
         2 x 6                            =12
                   2 x 5 +   3 x 4                =22
       {'a': 2, 'b': 2, 'c': 3} → {'a': 2, 'b': 2, 'c': 3}
       {'zzzzzzz': 1} → {'z': 7}
       {} → {}

    Precondition: input_dict is a possibly empty dictionary that will only have:
    - strings of repeating lower case characters (a-z) as keys
    - ints as values
    """
    # STUDENTS: assume preconditions are met. No need to assert them.

    newdict = {}
    for key in input_dict: # or input_dict.keys() (see reference sheet)
        merged_key = key[0]
        val = input_dict[key]

        new_val = val * len(key)
        if merged_key not in newdict:
            newdict[merged_key] = new_val
        else:
            newdict[merged_key] += new_val
    return newdict
```

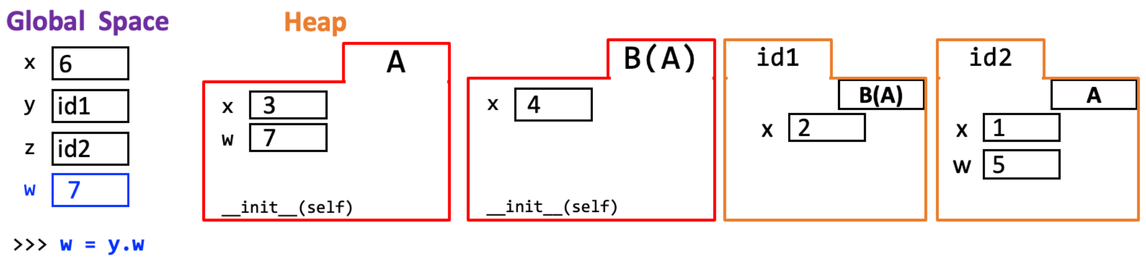
3. [10 points] **Visualizing Inheritance.** For this question, you will be shown the state of memory before a single assignment statement is executed. Modify the drawing to show how memory changes after that single assignment statement has been executed. If at any point an error is thrown, please write **ERROR** next to the assignment statement; only draw the changes to memory that would occur before the error occurs.

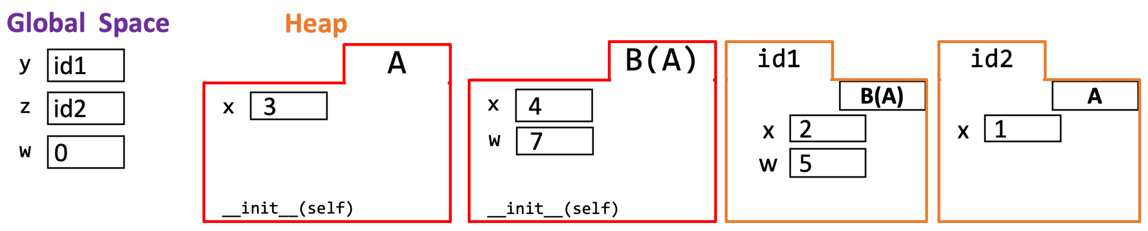
Each part is independent.
Notice: there is no Call Stack.
To the right is an example:



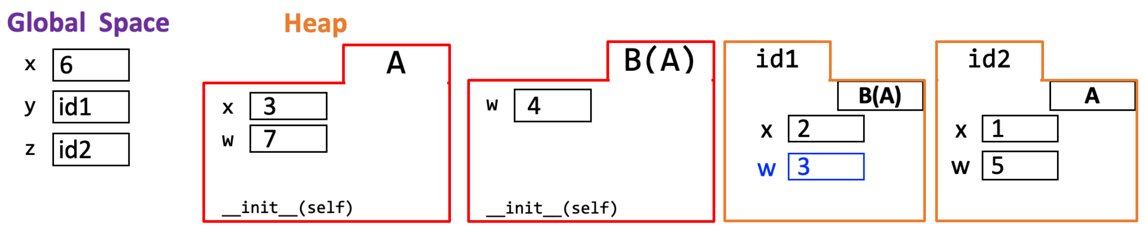








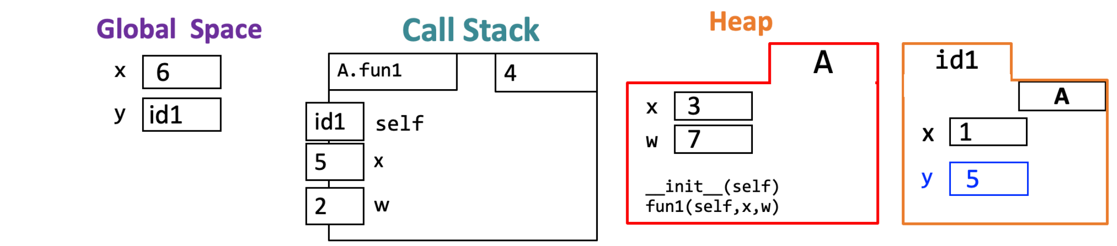
```
>>> a = z.w     ERROR
```



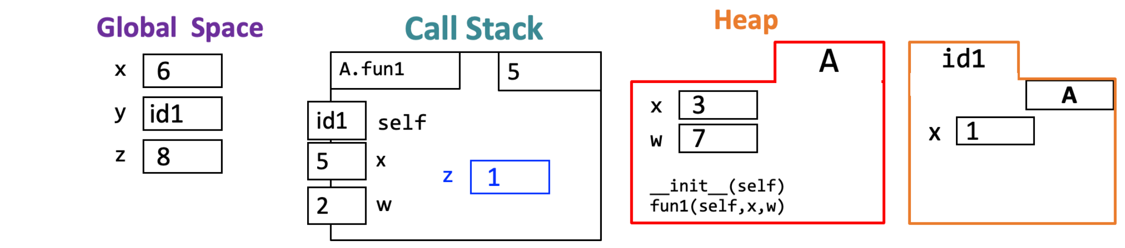
```
>>> y.w = B.x
```

---

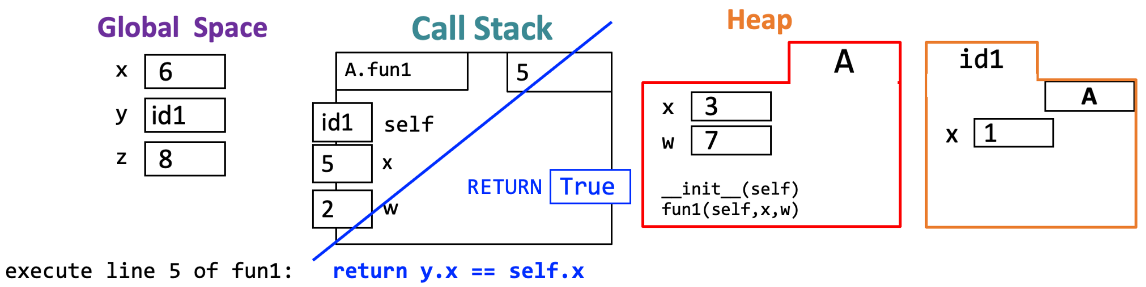
4. [7 points] **Visualizing Methods.** For this question, you will be shown the state of memory before a single Python statement is executed. Modify the drawing to show how memory changes after that single statement has been executed. If at any point an error is thrown, please write **ERROR** next to the assignment statement; only draw the changes to memory that would occur before the error occurs. Do not worry about changing the Program Counter in the top right corner of the call frame. (Since we are not showing you the code, you can't know what the next line of executable code will be). Once again, each part is independent. Notice that there is a call stack: each line being executed exists inside a method.



execute line 4 of fun1: `self.y = x`



execute line 5 of fun1: `z = self.x`



execute line 5 of fun1: `return y.x == self.x`

5. [16 points] **Recursion.** Let `Person` be a class as defined below:

```
class Person:
    """An instance represents a person who may or may not be CPR certified.
    (CPR certification is useful during a medical emergency.)

    Instance attributes:
       name             [str] - unique non-empty name of a person
       cert            [bool] - whether the person is CPR certified or not
       ec1   [Person or None] - emergency contact 1
       ec2   [Person or None] - emergency contact 2
    """
    def __init__(self, name, cert=False, c1=None, c2=None):
        """Create new Person with a name, CPR certification status, and
        up to 2 emergency contacts"""
        self.name = name
        self.cert = cert
        self.ec1 = c1
        self.ec2 = c2
```

Question begins on the next page.

(Continued from previous page.)

Implement the the `Person` class' instance method `can_help`, making effective use of recursion.

```
def can_help(self):
    """Returns: True if the person has someone who is CPR certified in
    their emergency contact network. This means either they or someone
    they can reach through their emergency contacts are CPR certified.
    Otherwise returns False.

    Examples:

    p1 = Person("BoA", False)                        p6:Irene-F
    p2 = Person("Ailee", True)                        /      \
    p3 = Person("Jisoo", False, p1, p2)        p4:HyunA-F  p5:Tzuyu-F
    p4 = Person("HyunA", False, p3)                 /
    p5 = Person("Tzuyu", False)                p3:Jisoo-F
    p6 = Person("Irene", False, p4, p5)           /     \
                                             p1:BoA-F  p2:Ailee-T
                                                       ^^ certified!
    ===IMPORTANT===
    p1.can_help() and p5.can_help() return False; they have
       no emergency contacts and are not themselves CPR certified
    p2.can_help(), p3.can_help(), p4.can_help(), p6.can_help() all
       Return True because they or someone they can reach via their
       emergency contacts are CPR certified
    """
    # STUDENTS: assume all Person objects are well formed with attributes
    # as described above

    if self.cert:
        return True
    c1 = False
    c2 = False
    if self.ec1 is not None:
        c1 = self.ec1.can_help()
    if self.ec2 is not None:
        c2 = self.ec2.can_help()
    return c1 or c2
```

```python
class Course:
    def __init__(self, name, n_credit):
        """
        Precondition: name is unique string identifier
                      n_credit is an int
        """
        self.name = name
        self.n_credit = n_credit

class Student:
    max_credit = 20

    def __init__(self, netID, courses):
        """
        Precondition: netID is unique string identifier
                      courses is a list of Course
        """
        self.netID = netID
        self.courses = courses
        # Add up credits
        for one_course in self.courses:
            self.n_credit += one_course.n_credit

    def enroll(self, new_course):
        """
        Precondition: new_course is a Course
        """
        if new_course.n_credit + self.n_credit <= self.maxcredit:
            self.courses.append(new_course)
            self.n_credit += new_course.n_credit

    def drop(self, course_name):
        """
        Precondition: course_name is the name of the course to drop
        """
        for one_course in self.courses:
            if one_course.name == course_name:
                self.n_credit -= one_course.n_credit
                self.courses.remove(one_course)

c1 = Course("CS 1110", 4)
c2 = Course("HADM 1810", 3)
s1 = Student("mep1", [c1])       # enroll in first course
s1.enroll(c2)                    # enroll in second course
assert len(s1.courses) == 2
s1.drop(c1)                      # drop a course
assert len(s1.courses) == 1      # should be down to 1 course...
```

6. **Debugging.** On the previous page is the code for two new classes and 7 lines of code that use them. Keep in mind that specifications are always correct and should not be changed.

   When the given code is run in Python, the following error is reported:

```
Traceback (most recent call last):
  File "college.py", line 43, in <module>
    s1 = Student("mep1", [c1])      # enroll in first course
  File "college.py", line 22, in __init__
    self.n_credit += one_course.n_credit
AttributeError: 'Student' object has no attribute 'n_credit'
```

   (a) [2 points] Fix the code to remove only the above error. **Fix only the problem that directly causes the above error message.** Mark your fix(es) with the label **FIX1**.

   Before the for loop in line 13, need to add the line `self.n_credit = 0`

   Now that you have fixed the error, you rerun the code and now a new error is reported:

```
Traceback (most recent call last):
  File "college.py", line 44, in <module>
    s1.enroll(c2)                # enroll in second course
  File "college.py", line 28, in enroll
    if new_course.n_credit + self.n_credit <= self.maxcredit:
AttributeError: 'Student' object has no attribute 'maxcredit'
```

   (b) [2 points] Fix the code to remove only this new error. **Fix only the problem that directly causes to the new error message.** Mark your fix(es) with the label **FIX2**.

   Line 17 has a typo, should read `self.max_credit` not `self.maxcredit`

   Now that you have fixed the error, you rerun the code and now a new error is reported:

```
Traceback (most recent call last):
  File "college.py", line 47, in <module>
    assert len(s1.courses) == 1        # should be down to 1 course...
AssertionError
```

   (c) [2 points] Is there a bug in the `drop` method?

   Circle One: Yes No

   (d) [2 points] If you answered No, explain what the problem is. If you answered Yes, fix the bug. Label this as **FIX3**.

   No, the problem is that the precondition is violated! `drop` is being called with a course, not a course name. (So there won't be a name match, so the course will not be dropped.)

7. **Classes and Subclasses.** Here are two classes `FoodItem` and `Cart`.

```
class FoodItem:
    """
    Represents some food available for sale at a store.

    Instance attributes:
        name (str):     name of the food item
        weight (float): how much the food weighs (in pounds); based on how
            much of the item there is.
        price (float):  how much the food costs
        flat_price (bool): indicates whether the price is the total price
            for the food item (if True) or the price per pound (if False).
    """
    def __init__(self, name, weight, price, flat_price=True):
        self.name = name
        self.weight = weight
        self.price = price
        self.flat_price = flat_price


class Cart:
    """
    Represents a shopping cart which holds food items.

    Instance attributes:
        contents (list of FoodItem): (possibly empty) list of all FoodItems
            in the cart.
    """
    def __init__(self):
        self.contents = []
```

Question begins on the next page.

(Continued from previous page.)

(a) [5 points] Implement the `add_item` method of class `Cart` so that it meets its specification.

```
def add_item(self, name, weight, price, flat_price=True):
    """This function makes a new FoodItem object (with name `name`,
    weight `weight`, price `price`, and flat_price `flat_price`) then
    adds the new FoodItem to the contents of the cart.

    Parameters:
        `name`: name of the FoodItem being created/added
        `weight`: the weight of the FoodItem being created/added
        `price`: the price of the FoodItem being created/added
        `flat_price`: flat_price value for the FoodItem
    """
    item = FoodItem(name, weight, price, flat_price)
    self.contents.append(item)
```

(b) [9 points] Implement the `calculate_total` method of class `Cart` so that it meets its specification.

```
def calculate_total(self):
    """
    Calculates the total cost to purchase all food items in the cart.

    The total should correctly account for the price of all items,
    including calculating the cost based on the weight of food items
    as necessary.

    Returns: the total cost (as a float; doesn't need to be rounded)
    """
    total = 0
    for item in self.contents:
        if item.flat_price:
            total += item.price
        else:
            total += (item.price * item.weight)
    return total
```

(Continued from previous page.)

Consider a subclass of `Cart` called `MembersCart`, which offers discounts to those who have purchased a store membership.

```
class MembersCart(Cart):
    """ Represents a shopping cart belonging to a customer who is a member,
    and has a corresponding membership discount rate on all items.  """

    discount = 0.10  # membership discount rate
```

(c) [8 points] Implement class `MembersCart`'s `calculate_total` method so that it meets specification.

```
    def calculate_total(self):
        """
        Calculates the total cost to purchase all food items in the cart.
        The total should correctly account for the price of all items,
        including calculating the cost based on the weight of food items
        as necessary.

        Additionally, apply the membership discount rate to adjust the total
            cost. (ie: if the discount is 0.25, the total should be 25% less)

        Returns: the discounted total cost (as a float; no rounding needed)
        """
        # Call super to calculate total
        total = super().calculate_total()
        total = total * (1-MembersCart.discount)
        return total
```

(d) [2 points] Will your `calculate_total` method work correctly, even though the `MembersCart` class does not have its own `__init__` method?

Circle One: Yes No

(e) [2 points] Explain.

Yes. First, it's fine for a class to not have an `__init__` method. In this particular instance, all the instance attributes that `MembersCart` needs are correctly initialized by the parent class' `__init__` method, which will be called by the constructor since `MembersCart` does not have its own.

8. [12 points] **While loops.** Implement this **function**, using a while-loop. Do not use `break`.

```
def filter_and_sum(mylist, n):
    """Returns the sum of the elements in a list. This function
    stops adding the elements when the nth zero is reached.

    Examples:
    filter_and_sum([1,0,2,3,0,4,0,5], 1) returns 1
      (1 then stops when it encounters the 1st 0 @ index 1)
    filter_and_sum([1,0,2,3,0,4,0,5], 2) returns 6
      (1+2+3 then stops when it encounters the 2nd 0 @ index 4)
    filter_and_sum([1,0,2,3,0,4,0,5], 4) returns 15
      (1+2+3+4+5 reaches the end of the list, never sees 4 0s)
    filter_and_sum([], 3) returns 0
    filter_and_sum([0], 1) returns 0

    Preconditions:
        mylist:  a (possibly empty) list of integers
        n:       an int, value >= 1
    """

    result = 0
    n_seen = 0
    i = 0
    while (n_seen < n and i < len(mylist)):
        result += mylist[i]
        if mylist[i] == 0:
            n_seen += 1
        i += 1
    return result
```

9. For each question, provide **only one answer**. If you provide 2, we will only grade the first.

   (a) [2 points] Which of the following statements about types in Python is true?

      (A) Python will never automatically convert a value from a narrower type to a wider type. For example, from an `int` to an `float`.

      (B) An operator (like `+`), has the same meaning regardless of the types of the values it operates on.

      (C) Once a variable has a value of a certain type, it can only ever have a value of that type assigned to it.

      (D) A class is a user-defined type.

      (E) Variables have types.

      Correct Answer: D

   (b) [2 points] Which of the following statements about testing and debugging is true?

      (A) A programmer should first make their code efficient and then test it for correctness.

      (B) Using `print` statements is a good way to find syntax errors in your code.

      (C) A good test suite will include test cases that violate the precondition.

      (D) Using `print` statements after `if` expressions is a good way to examine program flow.

      (E) A good test suite includes a test case for each possible input length.

      Correct Answer: D

   (c) [2 points] Which of the following statements about equality and identity is true?

      (A) The `==` operator should compare identity; the `is` operator should compare equality.

      (B) When used to compare two instances of a newly-defined class that has no overwritten version of the special method `__eq__`, the `==` operator will compare identity.

      (C) The `isinstance` method can be used interchangeably with the `type` method.

      (D) The `is` operator invokes the `__eq__` method.

      (E) The `==` operator invokes the `isinstance` method.

      Correct Answer: B

   (d) [2 points] Which of the following statements about **Linear Search** is **true**?

      (A) Linear Search is faster than Binary Search.

      (B) With each step of Linear Search, you can rule out half of the search space.

      (C) It's always better to sort your list so you can use Binary Search over Linear Search.

      (D) In Linear Search, doubling the list size quadruples the expected time of the search.

      (E) Linear Search works on any list, sorted or not.

      (F) Binary Search's complexity is on the order of n^2.

      Correct Answer: E

In class we looked at `insertion sort` and `merge sort`. Here is the implementation of a third sorting algorithm, called `selection sort`:

```python
def selection_sort(mylist):
    """Sorts a list of integers by repeatedly looking for the smallest integer
    from the unsorted part of the list (on the right) and swapping it with the
    integer at the beginning of the unsorted part. The sorted part of the list
    keeps growing until all of mylist is sorted.
    """
    n = len(mylist)

    for i in range(n):
        # Find the smallest element in the unsorted part of mylist
        min_idx = i
        for j in range(i+1, n):
            if mylist[j] < mylist[min_idx]:
                min_idx = j

        # swaps values of mylist[i] and mylist[min_idx]
        swap(mylist, i, min_idx)
```

(e) [2 points] If the size of the list that needs to sorted were to *double*, how would the work performed by `selection sort` change?

(A) The work would double, just like it does for `insertion sort`.
(B) The work would a little more than double, just like it does for `insertion sort`.
(C) The work would quadruple, just like it does for `insertion sort`.
(D) The work would double, just like it does for `merge sort`.
(E) The work would a little more than double, just like it does for `merge sort`.
(F) The work would quadruple, just like it does for `merge sort`.

Correct Answer: C

(f) [2 points] How would you describe the space requirements of `selection sort`?

(A) Like `insertion sort`, `selection sort` sorts the list in place, so the space requirements are *not* costly.
(B) Like `merge sort`, `selection sort` sorts the list in place, so the space requirements are *not* costly.
(C) Like `insertion sort`, `selection sort` requires that you make many temporary, new lists, so the space requirements are costly.
(D) Like `merge sort`, `selection sort` requires that you make many temporary, new lists, so the space requirements are costly.

Correct Answer: A