# ICS-33: In-Lab Programming Exam #3

## 1: Details of max_skip_ties:

The **max_skip_ties** generator function (a decorator for iterables) has zero or more iterable arguments, which are passed to its single parameter named **iterables**. If this generator function is passed no arguments, it produces no values, terminating the first time that **next** is called on it.

When we call **max_skip_ties**, it returns a result that is an **iterator**

- ...that generally produces the maximum value of the first values produced by all the iterables; then the maximum value of the second values produced by all the iterables; etc. This process stops when any of its iterable arguments cannot produce another value.
- ...but the produced value must have a **unique** maximum. So, if **multiple** iterables produced the **same** maximum value, then the values in only those iterables are **skipped** (the values for the other iterables remain the same). Thus, each of the multiple maximums is replaced by the next value in its iterable and this process is repeated.

Here is one simple example that illustrates some possibilities:

**max_skip_ties( [1, 3, 1] , [1, 2] , [3, 3, 1, 4] )**

would produce **3** and then **2** as follows:

- The initial values produced by all three iterables are **1**, **1**, and **3** respectively, which produces their unique maximum value: **3**.
- Next, **all** the iterables are advanced. Now their current values are **3**, **2**, and **3** respectively. But, because the maximum value **3** is not unique, it is not produced...
- ...instead, the first and third values (the duplicate maximums) are skipped: their iterables are advanced. Now their current values are **1** (the value after **3** in the first iterable), **2** (its value, not being a duplicate maximum, stays the same), and **1** (the value after the second **3** in the third iterable). Among these values, the iterator produces their unique maximum value: **2**.

  This same process could repeat multiple times before a value is produced, with some iterables advanced whenever their values are tied at the maximum. Repetition does not happen in this example.
- Next, **all** the iterables are advanced: but the first and second iterable cannot produce another value, so this iterator finishes.

Your code must work for any number of iterables (zero or more). Do not assume anything about the **iterable** arguments, other than they are **iterable**; the testing code uses the **hide** function to "disguise" a simple **iterable** (like a **list**). Don't even assume that any **iterable** is finite: so, don't try iterating all the way through any **iterable** to acquire all its values.

**Hint**: Because the iterators are not synchonously advanced, think about using a **list** of iterators and a **list** of their current values to process: advance the iterators in the first **list**, when necessary, to compute new values for the second **list**. My code comprises a dozen statements, including multiple comprehensions.

Finally, **You may NOT import any functions from itertools or functools** to help you solve this problem. Solve it with the standard Python tools that you know.

## 2: Details of can_sum_to:

The **can_sum_to** function computes whether **int** values in a **set** can sum to a specific amount. Only a subset of the values need to participate in the sum: each value can be **used once** or **not used** in the sum. The set values and specific amount can be positive, zero, or negative. The amount **0** can always be summed, by using **no** values in the set.

In the function header **can_sum_to ( pool : {int}, value : int ) -> bool:** the parameter **pool** is the **set** of values to choose from; and **value** is the specific amount that they must sum to.

Here are a few simple examples.

1. **can_sum_to( {1}, 1 )** returns **True**.
2. **can_sum_to( {1}, 2 )** returns **False**.
3. **can_sum_to( {1, 5, 8, -2}, 4)** returns **True: 1 + 5 - 2 = 4** (**8** is not used).
4. **can_sum_to( {1, 5, 8, -2}, 10)** returns **False**.
5. **can_sum_to({32, 7, -85, 83, -44, 87, -70, 94}, 14)** returns **True: 32 -85 - 44 + 87 -70 + 94 = 14** (**7** and **83** are not used).
6. **can_sum_to({32, 7, -85, 83, -44, 87, -70, 94}, 6)** returns **False:**.

For small arguments, like the ones I will test, such a function can compute its result in under a few seconds.

You must write the **can_sum_to** function **recursively**, but you can use any Python features (not just those use in functional programming). Do not import/use any other functions from any other modules. Think carefully about how to write this function. Using its specification above (its arguments, result, and their types), think about how to write the base case and how to break down the problem into similar/strictly smaller subproblems and successfully combine the solutions of these recursively solved subproblems. There are multiple ways to do so. "It's elephants all the way down."

**Hint**: It is always a good idea to avoid mutating parameters: you might find it useful to first copy a parameter and then mutate the copy. You can find information about **set** methods and operators in the **Set Type documentation** pdf that is distributed with this exam (4 pages). Note that Python operations like **union**, **intersection**, and **difference** (using the operators |, **&**, and **-**) do not mutate their operands: they each produce a new **set** object as a result.

## 3: Details of sorted_dict:

In this problem you will define a class named **sorted_dict**, derived from the **dict** class. Programmers can specify **key** and **reverse** information for **sorted_dict** objects, which they store as attributes. When Python iterates over the keys in a **sorted_dict**, it uses these attributes to determine the iteration order.

A **sorted_dict** has an attribute that stores the **temporal index** in which keys are associated with values indicating when the key was first used: **1** for the first key used, **2** for the second key used, etc. This specifies the default iteration order for its keys; but we can supply a **key** (function) and **reverse** (**bool**) -as we can when calling the **sorted** function- to specify a different order. All **key** functions take one argument, which is **3-tuple** of

- Index 0: the **temporal index** of the **key**
- Index 1: the **key** itself
- Index 2: the **value** currently associated with the **key**

Here is an illustration of the use of a **sorted_dict**. The **show_info** method returns the **_temporal_index** and **_temporal_keys** attributes, as well as the keys/values printed using the standard iteration order of a **dict**.

```
sd = sorted_dict()     # Construct an empty sorted_dict
print(sd.show_info()) # _temporal_index/_temporal_keys = 0/{}
                       # keys/values in dict order      = {}

sd['a'] = 10           # 1st association; 'a' temporal index is 1
print(sd.show_info()) # _temporal_index/_temporal_keys = 1/{'a': 1}
                       # keys/values in dict order      = {'a': 10}

sd['c'] = 30           # 2nd association: 'c' temporal index is 2
print(sd.show_info()) # _temporal_index/_temporal_keys = 2/{'a': 1, 'c': 2}
                       # keys/values in dict order      = {'a': 10, 'c': 30}


sd['b'] = 20           # 3rd association: 'b' temporal index is 3
print(sd.show_info()) # _temporal_index/_temporal_keys = 3/{'a': 1, 'c': 2, 'b': 3}
                       # keys/values in dict order      = {'a': 10, 'c': 30, 'b': 20}

print(sd)              # {'a': 10, 'c': 30, 'b': 20} appear in temporal (default) order

sd.set_sorting(key  = (lambda x : x[1]), reverse = True)  # specify key and reverse
print(sd)              # {'c': 30, 'b': 20, 'a': 10} appear in order of keys, reversed

sd.set_sorting(key  = (lambda x : x[2]), reverse = False) # Specify key and reverse
print(sd)              # {'a': 10, 'b': 20, 'c': 30} appear in order of increasing values
                       #                            HERE AND IN FOLLOWING 2 PRINTS

del sd['c']            # NO temporal data changes with deletions! 'c': 2 is retained
print(sd.show_info()) # _temporal_index/_temporal_keys = 3/{'a': 1, 'c': 2, 'b': 3}
                       # keys/values in dict order      = {'a': 10, 'b': 20}

print(sd)              # {'a': 10, 'b': 20} appear again in order of increasing values

sd['c'] = -7           # 4th association: NO temporal change: 'c' already in _temporal_keys
print(sd.show_info()) # _temporal_index/_temporal_keys = 3/{'a': 1, 'c': 2, 'b': 3}
                       # keys/values in dict order      = {'a': 10, 'b': 20, 'c': -7}

print(sd)              # {'c': -7, 'a': 10, 'b': 30} appear again in order of increasing values
```

**IMPORTANT**: Define **sorted_dict** by inheritance, so that it operates as specified below, producing the same results as in the examples shown above. By using inheritance, other methods like **__len__** or **__contains__** should be inherited and work correctly, without you having to write any code for them. Some calls to these (and other) methods may appear in the testing code for this derived class. Note: the **.keys()**, **.values()**, and **.items()** views will **not** automatically work correctly (and you don't have to implement them).

I have written two methods in the class, which you can use unmodified. Both are used in the example above.

- **show_info**: displays two attrributes and the keys/values in the **sorted_dict**, but in the standard **dict** order, not in the correct order for **sorted_dict**; printing this information is useful for debugging.

- **set_sorting**: sets the two attributes used to determine the iteration order in **__iter__** for a **sorted_dict**.

You will write only four methods: **__init__**, **__setitem__**, **__str__**, and **__iter__**. You should write **__init__** and **__setitem__** first. Then write **__str__**, using the inherited **__iter__**, so its associations appear in whatever order the standard **dict** iterates over its keys. Finally write **__iter__**: now that **dict**'s **__iter__** is overridden, **__str__** should show the keys in the correct order.

1. The **__init__** method should initialize the **sorted_dict** definining the four attributes listed below, using the exact names shown. When a **sorted_dict** object is constructed, it is passed no arguments.

    - **_temporal_index** stores a count of how many times any "new" keys have been used in a **sorted_dict**; a key is "new" only the first time it is used.

    - **_temporal_keys** stores a **dict** associating each key to a unique temporal index: the value of **_temporal_index** when the key was first used in the **sorted_dict**. **Important**: No information is ever removed from this **dict**: after deleting a key from a **sorted_dict**, its **_temporal_keys** will still contain that key, which is no longer in the **sorted_dict**. If that key is reused in the **sorted_dict**, its retains the same association in **_temporal_keys**.

    - **_sorter_key** stores a reference to a function object that determines the iteration order in **__iter__** (which is used in **__str__**). It should be initialized so that keys are iterated in temporal order. Recall that **_sorter_key** expects to work on the **3-tuple** (temporal index, key, value) specified above.

    - **_sorter_reverse** stores a **bool** that determines whether the order created by the **_sorter_key** should be reversed in the iteration order in **__iter__**. It should be initialized to **False**.

2. The **__setitem__** method should associate a never-used key in **_temporal_keys** with an updated **_temporal_index**; and, it should update the key/value association in the **sorted_dict**. See the earliest parts of the example shown previously.

3. The **__str__** method should return a **str** that looks like a **dict** but whose order of key/value associations is ultimately determined by **__iter__** (when it is written: which is affected by the attributes **_sorter_key** and **_sorter_reverse**). The returned string should show the the **repr**esentation of the keys and values: e.g., strings show within quotes. Note **repr('a')** is a string containing 3 characters: an **a** between single quotes (**'**).

4. The **_iter__** method should yield all the **keys** in the **sorted_dict** in the order specified by the attributes **_sorter_key** and **_sorter_reverse**. Recall that **_sorter_key** expects to work on the **3-tuple** (temporal index, key, value) specified above. **Hint**: yield the keys from a sorted **list** of the required **3-tuple**s. **IMPORTANT**: Be very careful about what you iterate over and how you iterate over it in **__iter__** to avoid attempting "recursive" iteration. There are multiple ways to avoid this problem, but doing so requires some thinking and knowledge of overridding methods.