PE04: PE of 19/12/2019 (solutions)

Master in Informatics and Computing Engineering
Programming Fundamentals
Instance: 2019/2020

An example of solutions for the 5 questions in this Practical on computer evaluation.

1. Count common items

Write a Python function **`common_items(alist, aset)`** that, given a list **`alist`** and a set **`aset`**, returns how many elements from **`alist`** appears in **`aset`**.

```
def common_items(alist, aset):
    r = 0
    for i in alist:
        if i in aset:
            r += 1
    return r
```

2. Process commands on sparse vectors

The vector (0, 5, 0, 4) can be represented in many forms. Typically, it is represented by a list or a tuple. For example: `v=[0, 5, 0, 4]`.

But, in some engineering tasks, vectors have too many zeros which occupy too much memory. In those cases, we can specify only positions that are non-zero using a dictionary. For example, the previous vector would be `v={1: 5, 3: 4}`. This is known as *sparse vector*.

Write a Python function `process(l)` that, given a list `l` that alternates between sparse vectors and operations to apply between the vectors, returns the resulting sparse vector.

The following operations are available:

Name	**Example**
`"add"`	`{1: 5} + {2: 7} = {1: 5, 2: 7}`
`"sub"`	`{1: 5} - {1: 3} = {1: 2}`
`"mul"`	`{1: 5} * {1: 3} = {1: 15}`

Remember that in sparse vectors: (i) all elements not shown are zero; (ii) zeros are not stored.

```
def process(l):
    res = l[0]       # 1st operand or the final result
    for i in range(1, len(l), 2):
        part = {}
        o = l[i]     # operation
        v = l[i+1]   # 2nd operand
        for k in v.keys() | res.keys():
            if o == 'add':
                elem = res.get(k, 0) + v.get(k, 0)
            if o == 'sub':
                elem = res.get(k, 0) - v.get(k, 0)
            if o == 'mul':
                elem = res.get(k, 0) * v.get(k, 0)
            if elem != 0:  # keep the vector sparse
                part[k] = elem
        res = part
    return res
```

3. Repeated odd and even numbers

Write the Python function **`repeated(nlist)`** that, given a list of integers in **`nlist`**, returns the difference between the number of repetitions of *even* numbers and the number of repetitions of *odd* numbers. A number that occurs *n* times in the list counts as *n*-1 repetitions. Zero is an even number.

You **cannot** use cycles, only map/filter/reduce or list comprehensions.

```
def repeated(list):
    # Filter even & odd numbers
    even = [el for el in list if el % 2 == 0]
    odd = [el for el in list if el % 2 != 0]

    # Compute number of repetitions
    rep_even = len(even) - len(set(even))
    rep_odd = len(odd) - len(set(odd))

    return rep_even - rep_odd
```

4. Generator from an interval list

An interval list is a list of tuples that represent intervals. Each tuple represents an interval in the form (min, max). Tuples appear in increasing order in the list and the intervals are disjoint.

For example, the list of intervals `intlist=[(1,1), (3,5), (10,15)]` represents the full sequence `1, 3, 4, 5, 10, 11, 12, 13, 14, 15`.

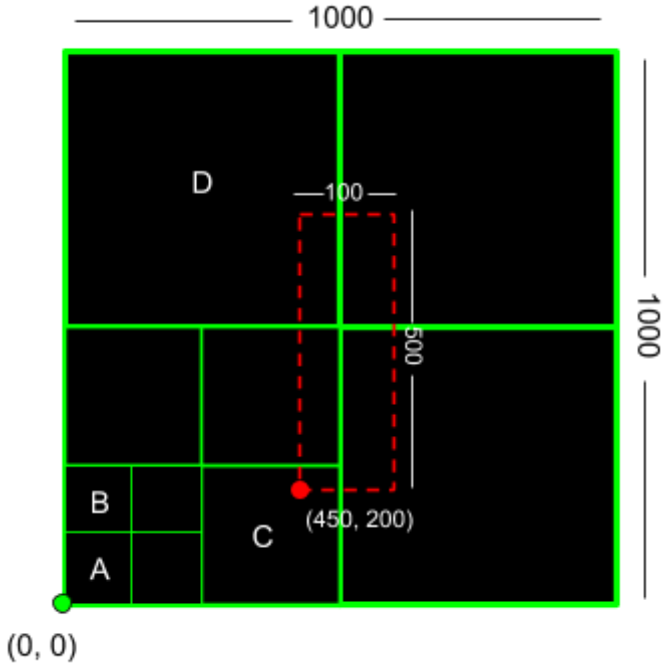
Write a generator function `generator(intlist)` that iteratively *yields* the next term in the full sequence, encoded in the list `intlist`.

```
def generator(intlist):
    while intlist != []:
        # Yield the 1st of the list
        yield intlist[0][0]
        # Go to next of the sequence
        if intlist[0][0] == intlist[0][1]:  # If min = max
            intlist = intlist[1:]           # Go to next interval
        else:  # Go to next in the same interval
            intlist = [(intlist[0][0]+1, intlist[0][1])] + intlist[1:]
```

5. Search map

For efficiency reasons, a map or screen can be divided into quadrants:

Each one of these quadrants may be further divided as illustrated:



Write a Python function `search_map(map, map_rectangle, search_rectangle)` that is given the map as `map`, the rectangle of that map as `map_rectangle` and the search rectangle (in red above) as `search_rectangle`. The function returns the objects inside the search rectangle as a set. In the example above, `{"C", "D"}` is returned.

The `map` is a nested tuple containing the four quadrants `(Q1, Q2, Q3, Q4)`. Each one of these quadrants can either be another quadrant tuple if there are several objects, or the object name as a string if there is one object, or `None` if there are no objects. Rectangles are tuples containing the coordinates as `(x, y, width, height)`, all positive, as represented in the figure above.

```
def search_map(amap, map_rectangle, search_rectangle):
    # Base condition
    if amap is None:       # no objects
        return set()
    if type(amap) == str:  # one object
        return {amap}

    # Explore each of the four quadrants recurrently.
    # For each, adjust the map boundaries (origin+size).
    mx, my, mw, mh = map_rectangle
    sx, sy, sw, sh = search_rectangle
    res = set()
    # Q1
    if sx <= mx+mw/2 and sy <= my+mh/2:
        res |= search_map(amap[0], (mx, my, mw/2, mh/2), search_rectangle)
    # Q2
    if sx+sw >= mx+mw/2 and sy <= my+mh/2:
        res |= search_map(amap[1], (mx+mw/2, my, mw/2, mh/2),
                          search_rectangle)
    # Q3
    if sx+sw >= mx+mw/2 and sy+sh >= my+mh/2:
        res |= search_map(amap[2], (mx+mw/2, my+mh/2, mw/2, mh/2),
                          search_rectangle)
    # Q4
    if sx <= mx+mw/2 and sy+sh >= my+mh/2:
        res |= search_map(amap[3], (mx, my+mh/2, mw/2, mh/2),
                          search_rectangle)
    return res
```

The end.

FPRO, 2019/20