Remove Multiple Entries in Huge Lists

Huge test file: rockyou.7z with 14' 344' 365 lines and 139,9 MB (139'921'497 bytes)

List vs. Initial Character Dictionary

Huge files tasks, like looking for double entries in files, can take a very long time, if applied conservatively. For example, we can collect all lines in a simple array

```
array = [line 0, line 1, ..., line n]
```

and then check each time when reading a new line, if it is already in the array. The longer the array gets, the longer it takes for the check.

So for this kind of problem we can make a dictionary where the lines are stored according to their initial character:

```
charDict = { a: [line a_0, line a_1, ... ], b: [line b_0, line b_1, ...], ...}
```

This already makes big difference in the performance:

Test with 71'552 lines:

simple array	character dictionary
18 seconds	< 1 second

Now we can apply the principle of the character dictionary also to the dictionary items themselves and we will get character dictionaries of certain depths. For example, a character dictionary of depth 2 would look like this:

```
charDict_2 = {
  a: {a: [line aa_0, line aa_1, ...], b: [line ab_0, line ab_1, ...], ...},
  b: {a: [line ba_0, line ba_1, ...], b: [line bb_0, line bb_1, ...], ...},
  c: {a: [line ca_0, line ca_1, ...], b: [line cb_0, line cb_1, ...], ...},
}
```

Where all lines of the form line aa_n start with the string "aa", lines of the form line ba_n start with the string "ab" and so on.

The algorithm to generate a character dictionary of depth 3 would look like this:

```
def dict_depth_3():
    wrapDict = {}
    for char in ["a", "b", "c", ...]:
        wrapDict[char] = {}
        for char2 in ["a", "b", "c", ...]:
            wrapDict[char][char2] = {}
            for char3 in ["a", "b", "c", ...]:
                wrapDict[char][char2][char3] = []
    return wrapDict
```

We want to be able to define the depth of the dictionary on the fly, so depth has to be an input argument. It is easier to start at the core of the dictionary with defining the function for the inner dictionary and then iteratively build wrapper dictionaries around it:

```
def simpleDict():
    simpleDict = {}
```

```
    for char in ["a", "b", "c", ...]:
        simpleDict[char] = []
    return simpleDict


def wrapperDict(depth):
    wrapperDicts = [simpleDict()] + [{} for i in range(depth - 1)]
    for i in range(depth - 1):
        for char in ["a", "b", "c", ...]:
            wrapperDicts[i + 1][char] = wrapperDicts[i]
    return wrapperDicts[depth - 1]
```

The function call wrapperDict(2) returns charDict_2. Let us look at the following case:

We have a character set of 121 characters and the file rockyou.7z with 14' 344' 365 lines and 139,9 MB (139'921'497 bytes). So a line consists of approximately ten bytes in average. The implementation is the python script removeDoubleLines.py.

Timing Test Results:

lines	simple array	character directory	character dictionary of depth 2	depth 2 : depth 1
100'000	84 seconds	> 2 seconds	41 seconds	20
500'000	2215 seconds	48 seconds	1765 seconds	30
4'000'000	-	4554 seconds	-	-
6'300'000	-	3:15:04 hours	-	-
9'600'000	-	5:30:16 hours	-	-
< 9'700'000	-	killed process	-	-

We see that in this setting, the character dictionary of depth 1 is the best choice. The dictionary of depth 2 has $121^2 = 14641$ sub arrays which slows down the process again considerably compared to the 121 sub arrays in the dictionary of depth 1. But it is still remarkably faster that the simple array.

For charset with less characters and the same file a dictionary of depth 2 might make more sense.

A good solution for very hug files would probably to divide them into manageable chunks and iteratively compare the chunks against each other.

Initial Character Distribution

The checking of initial character distribution in huge files is quickly done. The Python script initialCharDistribution.py counts the occurences of initial characters of the lines in a file. For the unzipped file of rockyou.7z it only took 9 seconds to get the results initial_character_distribution.txt.

Dividing Huge Files

To avoid memory errors, the algorithm is based on dividing files into smaller files, then compare them in pairs against each other:

```
for baseFileName in distributedFiles:
        baseIndex = distributedFiles.index(baseFileName)
        with open(baseFileName, "rb") as baseFile:
            baseFile = baseFile.readlines()
```

```
for compareFileName in distributedFiles[baseIndex + 1:]:
    with open(compareFileName, "rb") as compareFile:
        compareFile = compareFile.readlines()
        newContent = set(baseFile + compareFile) - set(baseFile)
    rewritePart(newContent, compareFileName)
```

If we got n files after dividing the huge files, then this algorithm will consist of *n*(n-1)/2* steps. For the unzipped file of rockyou.7z plus myspace.txt it only took **02:36.566385 minutes** to get the cleaned files.

So the algorithm of choice is: removeMultiLinesInHugeDir.py !!!

Tests with different params

size of directory	max size of distributed files	number of distributed files	comparing steps	timing
141 MB	500'000	41	820	3:45.965
141 MB	1'000'000	27	351	2:25.671