Hashing

From Programming for the Puzzled, Chapter 17
CAC 210

“the best things in life are free” -
> “nail biting refreshes the feet”
- Donald L. Holmes

Anagram

- An anagram is a word, phrase, or name formed by rearranging the letters of another
- iceman and cinema
- How can we group anagrams together?
- Given: ate but eat tub tea
- Can we produce: ate eat tea but tub

More of a Challenge

abed abet abets abut acme acre acres actors across airmen let alerted ales aligned allergy alter altered amen anew angel angle antler apt bade baste bead beast beat beats beta betas came care cares casters castor costar dealing gallery glean largely later leading learnt leas mace mane marine mean name pat race races recasts regally related remain rental sale scare seal tabu tap treadle tuba wane wean

Finding Anagram Groupings One at a Time

- Strategy:

For each word v in list:

For each word w != v in list:

Check if v and w are anagrams

If so, move w next to v

Finding Anagram Groupings One at a Time

```
def anagramGrouping(input):
    output = []
    seen = [False] * len(input)
    for i in range(len(input)):
        if seen[i]:
            continue
        output.append(input[i])
        seen[i] = True
        for j in range(i + 1, len(input)):
            if not seen[j] and anagram(input[i], input[j]):
                output.append(input[j])
                seen[j] = True
    return output
```

```
def anagram(str1, str2):
    return sorted(str1) == sorted(str2)
```

Finding Anagram Groupings One at a Time

```
def anagramGrouping(input):
    output = []
    seen = [False] * len(input)
    for i in range(len(input)):
        if seen[i]:
            continue
        output.append(input[i])
        seen[i] = True
        for j in range(i + 1, len(input)):
            if not seen[j] and anagram(input[i], input[j]):
                output.append(input[j])
                seen[j] = True
    return output
```

Creates a list the same length as input and makes all entries false

seen keeps track of words that have already been put into the output list

outer for loop: pick a word that has not already been put into output, put it into output, and look for words that are anagrams of this word

inner for loop: checks whether the word is already in output and whether it is an anagram of the chosen word. If so, it is added to the output list.

Each inner for loop produces one anagram grouping

Performance

- If we have a list of words of length n, and on average each word has m letters, we are doing roughly $n^2/2$ anagram checks
- We are comparing each distinct pair of words and don't compare v and w if we have already compared w and v - hence the 1/2
- Each anagram check requires roughly 2m log m comparisons of characters (sorting m letters in each of two words)
- Overall, $2n^2m$ log m comparisons in total

Something more efficient?

Anagram Grouping via Sorting

- anagramGrouping uses a doubly nested loop to group anagrams together
- What if we pair each word with its canonical representation
    - We create 2-tuples of the form (sorted(s), s) where the first item is a list of characters, and the second item is a string

Anagram Grouping via Sorting

- Given: [‘ate’, ‘but’, ‘eat’, ‘tub’, ‘tea’]
- We get five tuples:

([‘a’, ‘e’, ’t’], ‘ate’)

([‘b’, ’t’, ‘u’], ‘but’)

([‘a’, ‘e’, ’t’], ‘eat’)

([‘b’, ’t’, ‘u’], ‘tub)

([‘a’, ‘e’, ’t’], ‘tea’)

Anagram Grouping via Sorting

- Sort in ascending order….
- We get five tuples:

([‘a’, ‘e’, ’t’], ‘ate’)

([‘a’, ‘e’, ’t’], ‘eat’)

([‘a’, ‘e’, ’t’], ‘tea’)

([‘b’, ’t’, ‘u’], ‘but’)

([‘b’, ’t’, ‘u’], ‘tub)

Anagram Grouping via Sorting

```
def anagramSortChar(input):

    canonical = []

    for i in range(len(input)):

        canonical.append((sorted(input[i]), input[i]))

    canonical.sort()

    output = []

    for t in canonical:

        output.append(t[1])

    return output
```

Performance

- List of words of length n, and each word has m letters
- We are sorting the characters in each word, which takes a total of nm log m comparisons.
- Then, we sort the n tuples, and that takes n log n comparisons of tuples
- Overall, nm(log m + log n)

Anagram Grouping via Hashing

- What if we created a hash function using primes?

Let’s Do Together

- See Hashing/3.19.AnagramSolver.py in the shared Cloud9 space

Performance

- Given n items in a list, we will assume we need n log n comparisons to sort the list
- With m letters in each word, we only need m multiplications to compute the hash of a word
- Let's assume we dynamically compute the hashes of the two words being compared…so, the number of operations will be 2mn log n
- As opposed to $2n^2m$ log m in the first example
- Let m = 10 and n = 10,000…big difference?