Advanced topics in TCS

Exercise sheet 2.

Majority, Misra-Gries, Counting

Raphaël Clifford

For the implementation questions, please use any language of your choosing. The snippets I give are in Python. None of the code should take long to implement and doing so I hope will help clarify the algorithms.

Question 1.

1. Implement the majority algorithm as a function majority() and test your code on two different inputs. Example inputs:
   - $AAACCBBCCCBCC$
   - $AAABBBC$

2. Implement the second pass method as a function majoritysecondpass() to check your answers are correct.

```
# Check to see if alpha is really the majority item
def majoritysecondpass(sequence, alpha):
        # missing code
        if count > m/2:
                print(alpha, "is the majority item")
        else:
                print(alpha, "is not the majority item")

majoritysecondpass("AAABBBC", "C")
```

Question 2. Misra-Gries

Implement the Misra-Gries algorithm as a function misragries(). Test your code on two different inputs with different values of k. Example inputs:

- $ACABACBB$
- $AAACCBBCCCBCCAAC$

```
def misragries(sequence, k):
        # missing code
        return S

for k in range(1, 5):
        print(misragries("ACABACBB", k))

for k in range(1, 7):
        print("k=",k, misragries("AAACCBBCCCBCC", k))
```

Question 3.

Let $\hat{m}$ be the sum of all counters maintained by the Misra-Gries algorithm after it has processed an input stream, i.e., $\hat{m} = \sum_{\ell \in \text{keys}(A)} A[\ell]$. Prove that the bound from the lecture notes can be sharpened to

$$f_j - \frac{m - \hat{m}}{k} \leq \hat{f}_j \leq f_j$$

Question 4. Heavy hitters

Items that occur with high frequency in a dataset are sometimes called heavy hitters. Accordingly, let us define the HEAVY-HITTERS problem, with real parameter $\varepsilon > 0$, as follows. The input is a stream σ. Let m, n, f have their usual meanings. Let

$$\text{HH}_\varepsilon(\sigma) = \{j \in [n] : f_j \geq \varepsilon m\}$$

be the set of ε-heavy hitters in σ. Modify Misra-Gries to obtain a one-pass streaming algorithm that outputs this set "approximately" in the following sense: the set H it outputs should satisfy

$$\text{HH}_\varepsilon(\sigma) \subseteq H \subseteq \text{HH}_{\varepsilon/2}(\sigma)$$

Your algorithm should use $O\left(\varepsilon^{-1}(\log m + \log n)\right)$ bits of space.

Question 5. Misra-Gries
Suppose we have run the (one-pass) Misra-Gries algorithm on two streams σ_1 and σ_2, thereby obtaining a summary for each stream consisting of k counters. Consider the following algorithm for merging these two summaries to produce a single k-counter summary.

1. Combine the two sets of counters, adding up counts for any common items.
2. If more than k counters remain:
   (a) $c \leftarrow$ value of $(k+1)$ th counter, based on decreasing order of value.
   (b) Reduce each counter by c and delete all keys with non-positive counters.

Prove that the resulting summary is good for the combined stream $\sigma_1 \circ \sigma_2$ (here $\circ$ denotes concatenation of streams) in the sense that frequency estimates obtained from it satisfy the bounds from Question 3., where m is the combined length of the two streams.

Question 6.
This question is about the HyperLogLog (HLL) algorithm. You may find it helpful to refer to this paper (clickable link) and in particular Figure 1.

Randomised cardinality estimators are always described in academic contexts in terms of pairwise independent hash function. However, in practice a non-random hash function such as MD5 of SHA256 can be used instead and may be more practical in some circumstances. HLL turns out to be one of those situations. The explanation below will use the MD5 hash although we will only use the first 32 bits of the hash.

1. First pick a small positive integer p. For this experiment we can set $p = 5$.
2. Initialise an array M of length 2^p to be all zeros.
3. For each token in the stream:
   (a) Compute the MD5 hash of the token. Look at the first p bits in the hash, convert it into an integer and call this idx.
   (b) Starting at the $p + 1$th bit of the hash we just computer, count the number of leading zeros and call this ρ. If there are more than $32 - p$ leading zeros then set $\rho = 32 - p$.

(c) If $\rho + 1 > M[idx]$, set $M[idx] = \rho + 1$.

In order to get the full binary representation of a 128-bit MD5 hash you can use:

```
import hashlib
bin(int.from_bytes(hashlib.md5(x.encode()).digest(),
                                    "little"))[2:].zfill(128)
```

The method so far looks similar to the Tidemark algorithm although using the MD5 hash as a proxy for random bits. The main sophistication is in how these 2^p different estimates are combined. The method described in the original paper is as follows:

1. Let $m = 2^p$.
2. Compute $\alpha_m = \left(m \int_0^\infty \left(\log_2 \left(\frac{2+u}{1+u}\right)\right)^m \mathrm{d}u\right)^{-1}$
3. The cardinality esimate is $E = \alpha_m m^2 \left(\sum_{j=0}^{m-1} 2^{-M[j]}\right)^{-1}$.

Implement the HyperLogLog algorithm and try it on simulated data. In Python this will make some suitable random data.

```
rangeofvals = 20000
random.choices([*map(str, range(rangeofvals))], k = 5000)
```

You can use $\alpha_{2^5} \approx 0.69712$ or your code can numerically compute the integral using scipy.integrate.

Question 7.
The linked paper has some corrections for cases where the estimate is very small or large. If you have not done so already, implement these corrections and add them to your HyperLogLog code.