# Most Frequent Letters

Rafael Gil

Abstract –This report presents a study on three counting algorithms, developed with the aim to count the frequency of each individual letter in a literary work. The first algorithm is an exact counter algorithm, the second is an approximate counter algorithm and the third is an implementation of the lossy counter algorithm. The article begins with a presentation of the problem to be addressed, followed by an explanation of the algorithms developed. After this, an experimental analysis will be carried out for each of the algorithms, where the results obtained will be compared. Finally, the conclusions are presented.

Resumo –Este relatório apresenta um estudo sobre três algoritmos de contagem, desenvolvidos com o objetivo de contar a frequência de cada letra individual numa obra literária. O primeiro algoritmo é um algoritmo de contagem exacta, o segundo é um algoritmo de contagem aproximada e o terceiro é uma implementação do algoritmo de contagem lossy. O artigo começa com uma apresentação do problema a resolver, seguida de uma uma explicação dos algoritmos desenvolvidos. De seguida, é feita uma análise experimental de cada um dos algoritmos, onde serão comparados os resultados obtidos. Por fim, são apresentadas as conclusões.

Keywords –Algorithm, Counters, Exact, Approximate, Lossy

Palavras chave –Algoritmo, Contadores, Exato, Aproximado, Lossy

## I. Introduction

Our objective is to implemented three different algorithms, in order to count the frequency of each letter in a literary text. The literary text being used is "Romeo and Juliet", in three different languages: English, French and German.

## II. Algorithms implemented

### A. Exact Counter Algorithm

This is a simple algorithm, which goes through every individual letter and increments its designated counter, storing an exact counter for each letter, as we can see in algorithm 1.

Algorithm 1 Exact Count

```
function Exact_Count($text$)
    $letter\_counters \leftarrow \{\}$
    for $char$ in $text$ do
        if $char$ in $letter\_count$ then
            $letter\_count[char]$ += 1
        else
            $letter\_count[char] \leftarrow 1$
        end if
    end for
    return $letter\_counters$
end function
```

This algorithm runs in linear time, $O(n)$ where $n$ is the length of the text, as its time complexity grows larger for every single interaction it has to go through.

### B. Approximate Counter Algorithm

Similarly to the exact counter algorithm, this algorithm will also go through every individual letter in order to increment its dedicated counter. But the main difference is that it will only increment the counter with a set probability, in this case being 1/16. This way, we can count large streams of data without taking up too much memory.
This algorithm can be implemented in a similar way than the algorithm 1, changing only the way the counters are incremented. We can use the analogy of tossing a coin, deciding whether to increment the counters or not, based on the result of the tossing. To do this, we generated a random number between 1 and 16, choosing to increment the counters only when the random number equalled 1, as we can see in algorithm 2.

Algorithm 2 Approximate Counter

```
function Aprox_Counter($text$)
    $letter\_counter \leftarrow \{\}$
    for $char$ in $text$ do
        if Random.randint(1, 16) == 1 then
            if $char$ in $letter\_count$ then
                $letter\_count[char]$ += 1
            else
                $letter\_count[char] \leftarrow 1$
            end if
        end if
    end for
    return $letter\_counter$
end function
```

Similarly to the algorithm 1, this algorithm also runs in linear time for the same reasons.

### C. Lossy Counter Algorithm

This algorithm, proposed by Manku and Motwani [1], is a deterministic algorithm that computes frequency counts over a stream of data, using at most $\frac{1}{e}\log_e(N)$ space, where $N$ is the length of the data stream. The user specifies two parameters: support $s$ and error $e$. The algorithm consists of dividing the data stream into *buckets* of width $w = \lfloor\frac{1}{e}\rfloor$, labeling each of these *buckets* with an *id*, starting at 1, as we can see in algorithm 3.

Algorithm 3 Process Data Stream

```
1:  function ProcessDataStream($datastream$)
2:      for $data$ in $datastream$ do
3:          $currentBucket.append(data)$
4:          if len($currentBucket$) $\geq (1/error)$ then
5:              $processCurrentBucket()$
6:              $currentBucket \leftarrow []$
7:              $idCurrentBucket$ += 1
8:          end if
9:      end for
10: end function
```

Each *bucket* will be analyzed separately from the others.
For each individual cell of data in a *bucket*, the algorithm checks if that cell of data already has its counter. It it has its counter, it will be incremented; if not, a new counter is created with value 1 and a maximum error, equal to $idCurrentBucket - 1$, is registered for this data entry, as seen in algorithm 4 and algorithm 5.

Algorithm 4 Process Bucket

```
1: function processCurrentBucket
2:     for $data$ in $currentBucket$ do
3:         $processedDataCounter$ += 1
4:         $processIndividualData(data)$
5:     end for
6:     $decrementCounters()$
7:     $pruneFrequencies()$
8: end function
```

Algorithm 5 Process Individual Data

```
1: function processIndividualData($data$)
2:     if $data$ in $frequency\_counter$ then
3:         $frequency\_counter[data]$ += 1
4:     else
5:         $frequency\_counter[data] \leftarrow 1$
6:         $maxError[data] \leftarrow idCurrentBucket - 1$
7:     end if
8: end function
```

At the end of processing a *bucket*, there are two actions that take place. Firstly, every counter is decremented, seen in algorithm 6. Lastly, a process of pruning is conducted on the counters, in order to remove every counter where $frequency - maximumError <= idCurrentBucket$, as seen in algorithm 7.

Algorithm 6 Decrement Counters

```
1: function decrementCounters
2:     for $data$ in $frequency\_counter$.keys() do
3:         $frequency\_counter[data]$ -= 1
4:     end for
5: end function
```

Algorithm 7 Prune Frequencies

```
1:  function pruneFrequencies
2:      $to\_remove \leftarrow []$
3:      for $key, value$ in $frequency\_counter$.items() do
4:          if $value + maxError[key] \leq idCurrentBucket$ then
5:              $to\_remove$.append($key$)
6:          end if
7:      end for
8:      for $key$ in $to\_remove$ do
9:          delete $frequency\_counter[key]$
10:         delete $maxError[key]$
11:     end for
12: end function
```

This process is repeated until the data stream ends, giving an extremely decent approximation of the frequencies in the data stream. This approximation grows smaller in accuracy the bigger the data stream is, but it compensates in space efficiency, proving to be a really efficient algorithm to use when dealing with counters in immensely large data streams.

## III. Results

To conduct this experimental analysis, three versions of the same literary piece were used, in order to not only compare the efficiency between the three algorithms, but also to compare the frequencies of the letters between three different languages.
Every one of the algorithms has a very fast execution time, as they all take less than a second to process the entirety of the literary work.

### A. Algorithms comparison

Using only the english version, we will compare the efficiency between the algorithms by analysing the frequency counters for the $n = 3, 5, 10$ most frequent letters.

#### A.1 Results for top 3 letters

Starting by checking the three most frequent letters, we can see, using the exact count algorithm, that the most frequent letters are $E$, $T$ and $O$.

| Letter | Frequency Counter |
|---|---|
| E | 12839 |
| T | 9785 |
| O | 8847 |

TABLE I
Results of exact counter algorithm - top 3 letters

For the approximate counter algorithm, we also obtained these same letters, in the same order, but, obviously, with different counters.

| Letter | Frequency Counter |
|---|---|
| E | 771 |
| T | 629 |
| O | 573 |

TABLE II
Results of approximate counter algorithm - top 3 letters

For the lossy counter algorithm, we, once again, obtained the correct frequency order for the letters, but this time with a much more accurate representation of the real values of their frequencies, being off only by a couple hundred.

| Letter | Frequency Counter |
|---|---|
| E | 12621 |
| T | 9570 |
| O | 8632 |

TABLE III
Results of lossy counter algorithm - top 3 letters

### A.2 Results for top 5 letters

Similarly to the results obtained from the tests conducted before, we obtained the same letters with the highest frequency, across every algorithm.

| Letter | Frequency Counter |
|---|---|
| E | 12839 |
| T | 9785 |
| O | 8847 |
| A | 8300 |
| I | 6949 |

TABLE IV
Results of exact counter algorithm - top 5 letters

| Letter | Frequency Counter |
|---|---|
| E | 779 |
| T | 587 |
| O | 565 |
| A | 560 |
| I | 456 |

TABLE V
Results of approximate counter algorithm - top 5 letters

| Letter | Frequency Counter |
|---|---|
| E | 12621 |
| T | 9570 |
| O | 8632 |
| A | 8086 |
| I | 6735 |

TABLE VI
Results of lossy counter algorithm - top 5 letters

### A.3 Results for top 10

For the top 10 letters, we can start to see an inefficiency in the approximate counter algorithm, as it fails to get the correct order of the most frequent letters, even though it still manages to identify the most frequent letters. The lossy algorithm continues to accurately determine the most frequent letters, proving its efficiency.

| Letter | Frequency Counter |
|---|---|
| E | 12839 |
| T | 9785 |
| O | 8847 |
| A | 8300 |
| I | 6949 |
| H | 6802 |
| S | 6647 |
| N | 6541 |
| R | 6533 |
| L | 5006 |

TABLE VII
Results of exact counter algorithm - top 10 letters

| Letter | Frequency Counter |
|---|---|
| E | 769 |
| T | 609 |
| O | 547 |
| A | 505 |
| I | 462 |
| H | 405 |
| S | 402 |
| R | 398 |
| N | 371 |
| L | 311 |

TABLE VIII
Results of approximate counter algorithm - top 10 letters

| Letter | Frequency Counter |
|---|---|
| E | 12621 |
| T | 9570 |
| O | 8632 |
| A | 8086 |
| I | 6735 |
| H | 6588 |
| S | 6434 |
| N | 6326 |
| R | 6318 |
| L | 4792 |

TABLE IX
Results of lossy counter algorithm - top 10 letters

### A.4 Approximate vs Lossy

As we can see, both the approximate counter and the lossy counter algorithms perform relatively well, with the lossy counter being the clear winner. This is due to it being more accurate in determining the actual order of the frequencies and the true values of the counters, while still conserving space in memory.

### B. Comparing between languages

The pattern observed, between the performance of the algorithms, previously, can still be seen in both the French and German version of the literary work, with the lossy counter algorithm performing better than the approximate counter algorithm, with the latter showing a small inefficiency in determining the correct order of the frequencies.

## IV. Conclusion

In conclusion, it is important to notice the importance that these algorithms have in dealing with complex problems in the real world, such has monitoring network links, counting distinct flows, dealing with some specific problems in databases, etc.
Having these topics in consideration, having had to study and implement this algorithms, will only improve the understanding of such complex topics.

## References

[1] Rajeev Motwani Gurmeet Singh Manku, “Approximate frequency counts over data streams”, VLDB ’02: Proceedings of the 28th International Conference on Very Large Databases, 2002.