Project #3: Most Frequent Letters Advanced Algorithms

João Bernardo Tavares Farias
joaobernardo0@ua.pt

***Abstract*—This document was produced as a result of a study on the most frequent letters in documents using different algorithms. It was implemented an exact counter, an approximate counter and lastly, one algorithm to identify frequent items in data streams. These algorithms are discussed below.**

***Index Terms*—Frequent Letters, Data Stream, Text, Misra Gries, Approximate**

I. Introduction

This document was written as a result of the **Project #3** of Advanced Algorithms course and aims to explain the Most Frequent Letters Problem and the algorithms used to achieve the solutions. Some key concepts about each algorithm are explained as well.

II. Most Frequent Letters Problem

The Most Frequent Letters problem is a problem of finding the most frequently occurring letters in a given string/text. This can be useful for tasks such as analyzing text data or identifying the most common letters in a given language. Next, the algorithms used to solve this problem are presented and explained.

III. Basic Counter Algorithm

The easiest one is the **Basic Counter Algorithm**. This algorithm is a simple method for keeping track of a count or tally. It is often used in situations where it is necessary to count a specific type of item or occurrence. In the aftermath of the problem, the items are letters. To use this algorithm, the following steps were used:

1) Initialize an empty dictionary to store each letter's count.
2) Iterate over the elements in the collection.
3) For each element, check if the element is present in dictionary. If it is, increment the count by one. If not, add it to the dictionary with the value 1.
4) After iterating over the entire collection, the dictionary variable will contain the number of occurrences of the element.

A. Pseudo-code

The pseudo-code used to implement this function is described next:

Algorithm 1 Basic Count Algorithm

```
counters ← defaultdict(int)
for char in text do
    if char.isalpha() & char ≠ ” then
        char ← remove_accents(char)
        counters[char] ← counters[char] + 1
    end if
end for
sorted_counters ← sorted(counters.items(), key=lambda x: x[1], reverse=True)
```

B. Advantages

The basic count algorithm has a number of advantages:

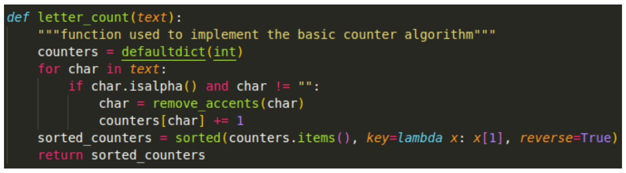
1) It is simple and clear to understand, which makes it easy to implement and debug.
2) It is fast and efficient, as it only requires a single pass through the data to count the frequency of each element.
3) It is flexible, as it can be used to count the frequency of any type of element, including integers, floating-point numbers, strings, and even complex objects.
4) It is a useful tool for data exploration and analysis, as it can help understanding the distribution and patterns in text.

C. Disadvantages

There are a few disadvantages associated with the basic count algorithm:

1) It requires additional space to store the counts of each element. This can be a problem if the number of unique elements is very large, as it may not be possible to store the counts in memory.
2) It is not suitable for streaming data, as it requires the entire dataset to be available upfront.
3) It is not very accurate for estimating the frequency of rare elements, as the basic count algorithm can only give an exact count for elements that are present in the dataset.
4) It is not suitable for counting the frequency of elements in very large datasets, as the time complexity of the algorithm is linear in the size of the dataset.
5) It is not suitable for counting the frequency of elements in distributed systems, as it requires all the data to be processed by a single machine.

Code implemented



```python
def letter_count(text):
    """function used to implement the basic counter algorithm"""
    counters = defaultdict(int)
    for char in text:
        if char.isalpha() and char != "":
            char = remove_accents(char)
            counters[char] += 1
    sorted_counters = sorted(counters.items(), key=lambda x: x[1], reverse=True)
    return sorted_counters
```

Fig. 1. Basic Counter

IV. Csuros' Counter Algorithm

The **Csuros' Counter Algorithm**, also known as Approximate Floating-point Counter (AFC), is a method for counting the number of occurrences of a specific element in a collection, using approximate floating-point arithmetic. It is a variant of the **basic counter algorithm**, but it uses approximations to reduce the number of exact floating-point operations required, which can improve performance. The basic steps of this algorithm are described next:

1) Initialize an empty dictionary to store the count.
2) Iterate over the elements in the collection.
3) For each element, check if the element is in dictionary. If it is, increment the count by an approximate value using approximate floating-point arithmetic
4) After iterating over the entire collection, the count variable will contain an approximate value for the number of occurrences of each element.

The **Csuros' Counter** can be implemented in various ways, depending on the specific requirements and constraints of the application. For example, the approximate value used for each increment could be a fixed value, or it could be based on the current count or the size of the collection. The level of approximation can also be adjusted to trade off accuracy for performance. This algorithm can be useful in situations where exact counting is not required, and the overhead of exact floating-point operations is a performance concern. However, it is important to carefully consider the trade-offs between accuracy and performance when using this algorithm, as the approximations may introduce errors that may impact the results of the count.

A. Pseudo-code

The pseudo-code used to implement this algorithm is described next:

Algorithm 2 Increment Letter

```
$t \leftarrow x \div m$
**while** $t > 0$ **do**
    **if** random.getrandbits(1) = 1 **then**
        **return** $x$
    **end if**
    $t \leftarrow t - 1$
**end while**
**return** $x + 1$
```

Algorithm 3 Estimate Frequent Letters

```
$letter\_count \leftarrow \{\}$
**for** $char \in text$ **do**
    **if** $char.isalpha()$ **then**
        $char \leftarrow remove\_accents(char)$
        **if** $char \neq$ ” **then**
            $letter\_count[char] \leftarrow incrementLetter(letter\_count[char], threshold)$
        **end if**
    **end if**
**end for**
$csuros\_counter \leftarrow sorted(letter\_count.items(), key = lambdax : x[1], reverse = True)$
**return** $csuros\_counter$
```

B. Advantages over Basic Count Algorithm

The list below presents some advantages of this algorithm over the Basic Count:

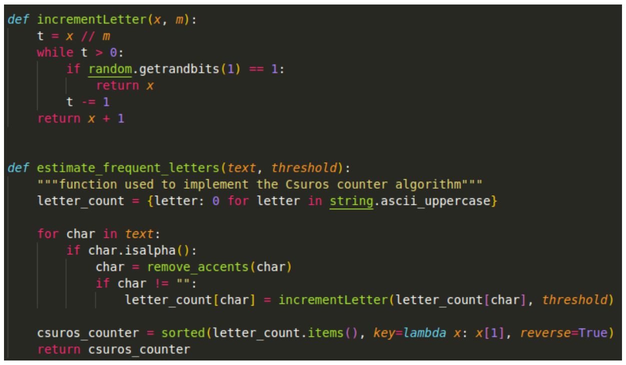
1) It requires much less space than the basic count algorithm, as it only stores a small number of hash values and count estimates for each element.
2) It can handle streaming data, as it can process elements incrementally without the need to store the entire dataset in memory.
3) It is highly accurate for estimating the frequency of common elements, as it uses a clever sampling technique to produce accurate count estimates.
4) It is suitable for very large datasets, as it has a constant space complexity and a **logarithmic** time complexity.
5) It is highly flexible, as it can be used to count the frequency of any type of element, including integers, floating-point numbers, strings, and even complex objects.

C. Disadvantages over Basic Count Algorithm

The list below presents some disadvantages of this algorithm over Basic Count:

1) It is less accurate for estimating the frequency of rare elements, as the AFC uses a sampling technique that may not accurately capture the count of infrequent elements.
2) It is more complex than the basic count algorithm, which can make it more difficult to implement and debug.
3) It is a probabilistic data structure, which means that the count estimates produced by this algorithm may have some error.
4) It may not be suitable for situations where exact counts are required, as the **Csuros' Counter** can only provide estimates.

Code implemented



```python
def incrementLetter(x, m):
    t = x // m
    while t > 0:
        if random.getrandbits(1) == 1:
            return x
        t -= 1
    return x + 1


def estimate_frequent_letters(text, threshold):
    """function used to implement the Csuros counter algorithm"""
    letter_count = {letter: 0 for letter in string.ascii_uppercase}

    for char in text:
        if char.isalpha():
            char = remove_accents(char)
            if char != "":
                letter_count[char] = incrementLetter(letter_count[char], threshold)

    csuros_counter = sorted(letter_count.items(), key=lambda x: x[1], reverse=True)
    return csuros_counter
```

Fig. 2. Csuros' Counter

V. Freq Count Algorithm

The **Frequent Count** algorithm is a streaming algorithm for finding the frequent elements in a dataset. It is based on the following idea:

1) Initialize an empty dictionary
2) Iterate the elements of the text, and for each of them if the element is in dictionary and is not full, increment the counter, if the element does not have a corresponding counter, or if the counter is already full, decrement the counter of every element.
3) After all the elements have been processed, the elements with non-zero values are the frequent elements.

A. Pseudocode

The following algorithm represents the pseudo-code:

Algorithm 4 Misra Gries Algorithm

```
$A \leftarrow \{\}$
for $j \in text$ do
    if $j.isalpha()$ then
        $j \leftarrow remove\_accents(j)$
        if $j \in A$ then
            $A[j] \leftarrow A[j] + 1$
        end if
    else
        if $|A| < k - 1$ then
            $A[j] \leftarrow 1$
        end if

        for $i \in list(A.keys())$ do
            $A[i] \leftarrow A[i] - 1$
            if $A[i] = 0$ then
                $del A[i]$
            end if
        end for
    end if
end for
return sorted($A.items()$, key=$lambdax : x[1]$, reverse=True)
```

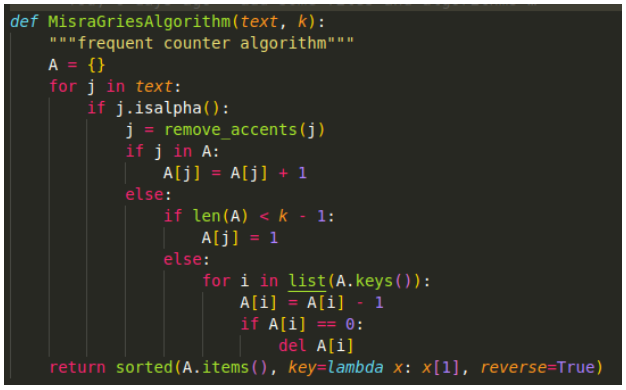
B. Advantages over Basic Count Algorithm

1) It requires much less space than the basic count algorithm, as it only stores a small number of letters for the most frequent elements.
2) It can handle streaming data, as it can process elements incrementally without the need to store the entire dataset in memory.
3) It is suitable for very large datasets, as it has a constant space complexity and a linear time complexity.
4) It is highly flexible, as it can be used to find the frequent elements of any type of element, including integers, floating-point numbers, strings, and even complex objects.
5) It is simple and easy to implement.
6) It is efficient, as it only requires a single pass through the data.

C. Disadvantages over Basic Count Algorithm

1) It is not very accurate for finding rare frequent elements, as the counters may be decremented too many times before the element is encountered again.
2) It requires a careful choice of the number of counters to achieve good accuracy.
3) It is not suitable for finding the exact frequency of elements, as it only provides approximate counts.
4) It is not as widely used or well-known as the basic count algorithm, so there may be less support and resources available for it.

Code implemented



```python
def MisraGriesAlgorithm(text, k):
    """frequent counter algorithm"""
    A = {}
    for j in text:
        if j.isalpha():
            j = remove_accents(j)
            if j in A:
                A[j] = A[j] + 1
            else:
                if len(A) < k - 1:
                    A[j] = 1
                else:
                    for i in list(A.keys()):
                        A[i] = A[i] - 1
                        if A[i] == 0:
                            del A[i]
    return sorted(A.items(), key=lambda x: x[1], reverse=True)
```

Fig. 3. Freq counter

VI. Difference between Frequent Count Algorithm and Csuros' Count Algorithm

The **Frequent Count algorithm** is a streaming algorithm for finding the most frequent elements in a stream of data, while the **Csuros' Counter** is a data structure for efficiently storing and querying the frequencies of floating-point numbers.

The **Frequent Count algorithm** algorithm maintains a fixed-size array of counters and processes each element in the stream one by one. If the element has not been seen before, it is added to the array with a count of 1. If the element has been seen before, its count is incremented. If the array is full and the element is not present in the array, the count of each element in the array is decremented. When the algorithm has processed all the elements in the stream, the elements with the highest counts are the most frequent elements.

The **Csuros' Counter**, on the other hand, is a data structure that uses hashing and approximations to store and query the frequencies of floating-point numbers. It stores the frequencies of a given number of floating-point values in a hash table, and uses approximations to estimate the frequencies of other floating-point values. The accuracy of the approximations can be controlled by adjusting the number of values stored in the hash table.

Overall, the **Frequent Count algorithm** is a general-purpose algorithm for finding the most frequent elements in a stream, while the approximate **Csuros' Counter** is a specialized data structure for efficiently storing and querying the frequencies of floating-point numbers.

VII. Code organization

The file *frequent_count.py* contains all the code used to solve this problem. The user can choose the files to analyse specifying the folder where they are stored. In this case there are **three** books: **D. Quixote**, **Lusíadas** and **Romeo and Juliet**. Each of these books is available in different languages. In order to add more books, the user just needs to store the files in a folder inside **collections** directory.

In the following example, the files inside *lusiadas* folder will be analysed.

```
$ python3 frequent_count.py -f lusiadas
```

If no folder is specified, the algorithm analyses all files inside collections folder.

The files are stored in a list and **for each**, a set of operations is applied. Firstly, the content of the file in uppercase is read and the punctuation is removed. Then, the empty spaces and the break lines characters are removed. In this moment the tokens of the file are stored in a list. Then the corresponding **stopwords** file is opened, and the tokens that are simultaneously present in the stopwords file and in the text file are removed. Once the file's name has the language of the file, it is simple to get the correct **stopwords** file.

After these text operations, each algorithm is executed and the results are stored in a text file inside **results** directory, with the same name as the corresponding file.

VIII. Results

In the following tables are represented the **top 10** most frequent letters in each language, depending on the algorithm used. The data of each language was combined depending on the algorithm used. For example, the data present in all english files was combined and the results are described in **Table III**.

The same logic was used in the other languages.

TABLE I
Spanish letters

Letter	Letter Basic Counter	Csuros' Counter	Freq Counter
E	17318	1058	5719
A	16575	1038	4975
O	11870	8480	278
S	9538	7236	5
N	8975	7022	-
R	8873	6958	-
L	7333	6199	-
I	6654	5845	-
D	5936	5479	-
U	5656	5326	1

TABLE II
Portuguese Letters

Letter	Letter Basic Counter	Csuros' Counter	Freq Counter
A	17136	10501	6299
E	16344	10386	5508
O	14119	9606	3288
S	10327	7627	45
R	8411	6699	-
N	6594	5817	-
I	6518	5751	-
D	6159	5557	2
T	5984	5495	-
M	5552	5259	-

TABLE III
English letters

Letter	Letter Basic Counter	Csuros' Counter	Freq Counter
E	88704	35475	18245
T	63812	30284	49
O	56503	28467	16
A	56746	28694	3
I	47837	25883	1
H	49622	26864	-
S	48295	26585	-
R	44769	25494	-
N	49063	26278	1
L	28590	19586	1

TABLE IV
Deutsch letters

Letter	Letter Basic Counter	Csuros' Counter	Freq Counter
E	45893	23047	22766
N	25743	16862	2628
A	17132	12909	5
D	15759	12111	2
O	14605	11082	17
I	17923	13962	1
T	15671	12494	-
R	15192	12464	2
H	10706	10074	-
S	10543	9721	-

TABLE VI
HUNGARIAN LETTERS

Letter	Letter Basic Counter	Csuros' Counter	Freq Counter
E	49550	16813	13396
A	49449	16803	13297
T	29425	13668	-
O	28457	13334	2
N	25351	12694	1
S	22896	11925	2
L	22350	11869	1
K	16957	10502	-
G	16547	10427	1
I	16418	10373	-

TABLE V
FRENCH LETTERS

Letter	**Letter Basic Counter**	**Csuros' Counter**	**Freq Counter**
E	15428	10081	22766
A	6640	5820	-
S	6547	5773	-
T	6454	5737	55
R	6368	5666	-
O	6086	5538	11
I	6010	5518	-
N	5916	5456	-
U	5758	5388	-
L	4805	4805	1

The vowels are on top 3 of the most common letters of the languages present in the tables. The use of vowels is an important aspect of the structure and grammar of European languages, and has played a significant role in the development of these languages over time. Thus, its usage is very common and every word has at least one vowel.

The letter **E** is the most frequent letter in all languages present except in Portuguese. Despite of this fact, the most common letter in portuguese is the letter **A**, a vowel as well.

IX. ERRORS

Both, **Csuros' counter** and **Freq Count** algorithms produce errors and, in this section are described the errors associated to each of these algorithms in each file. Solely three files were analyzed (Lusíadas in Portuguese, English and Spanish), but the other result can be found in the created files inside **results** directory.

Starting by the file **lusiadas_pt.txt**, the next four figures represent the errors associated to the each algorithm:

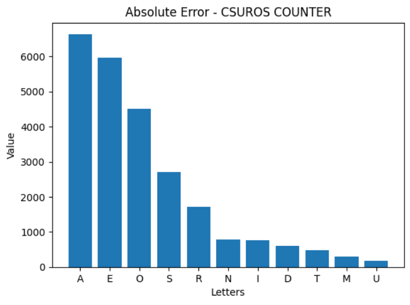


Fig. 4. Absolute Error Csuros Algorithm in lusiadas_pt.txt file

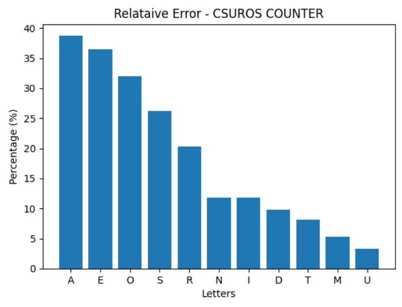


Fig. 5. Relative Error Csuros Algorithm in lusiadas_pt.txt file

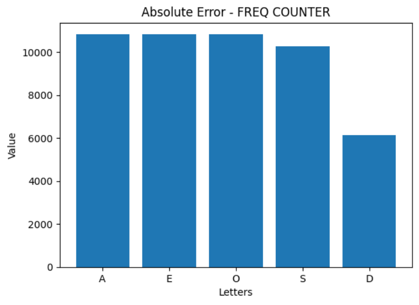


Fig. 6. Absolute Error Freq Count Algorithm in lusiadas_pt.txt file

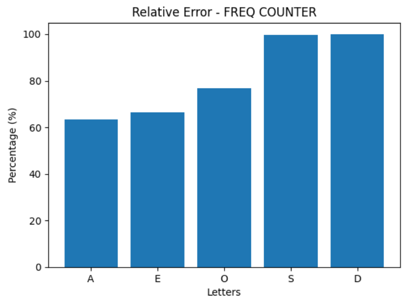


Fig. 7. Relative Error Freq Count Algorithm in lusiadas_pt.txt file

Regarding the **lusiadas_en.txt** file, the errors can be represented by the following graphs:

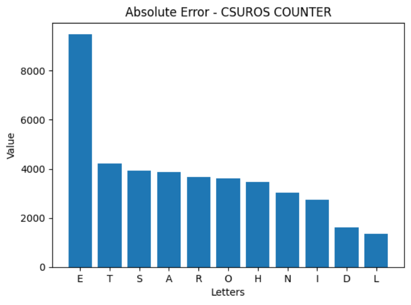


Fig. 8. Absolute Error Csuros Algorithm in lusiadas_en.txt file

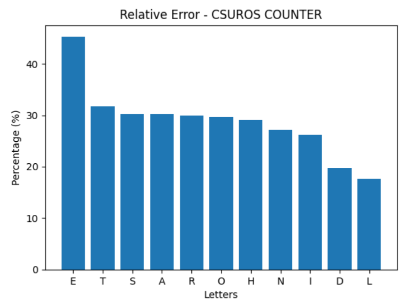


Fig. 9. Relative Error Csuros Algorithm in lusiadas_en.txt file

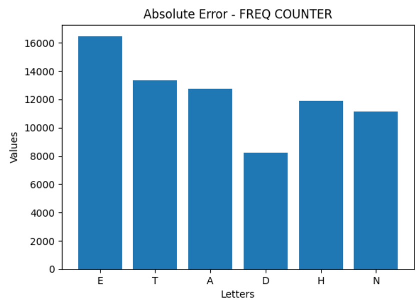


Fig. 10. Absolute Error Freq Count Algorithm in lusiadas_en.txt file

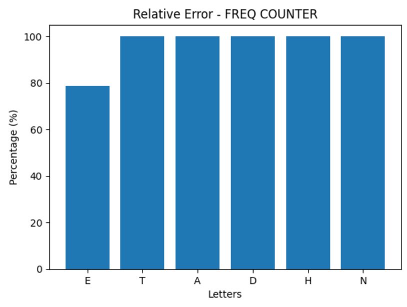


Fig. 11. Relative Error Freq Count Algorithm in lusiadas_en.txt file

Regarding the last file (**lusiadas_sp.txt**), the representative graphs are shown below:

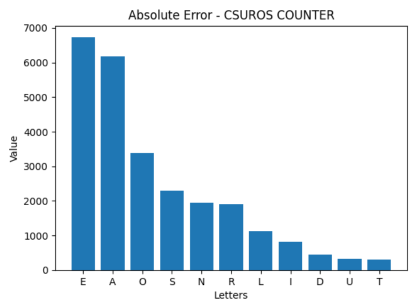


Fig. 12. Absolute Error Csuros Algorithm in lusiadas_sp.txt file

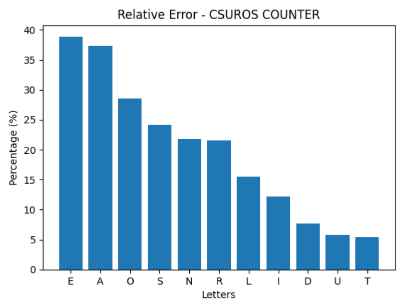


Fig. 13. Relative Error Csuros Algorithm in lusiadas_sp.txt file

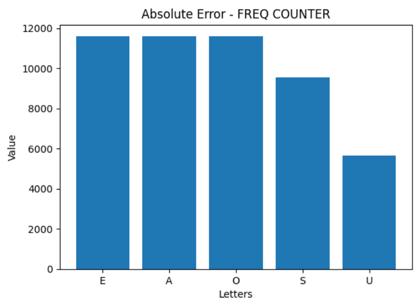


Fig. 14. Absolute Error Freq Count Algorithm in lusiadas_sp.txt file

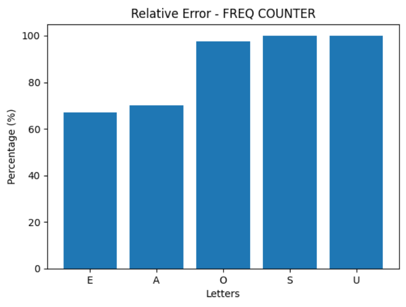


Fig. 15. Relative Error Freq Count Algorithm in lusiadas_sp.txt file

As the figures suggests, the Freq Count Algorithm is the algorithm with the highest error, nevertheless **all of them identify the same letters as the most frequent**. Thus, although the number of occurrences are not correct in **Csuros Counter** and **Freq Count** algorithms, the order of the most common letters is correctly identified.

X. Acknowledgment

In conclusion, basic counters, Csuros Counter algorithm, and Freq Counter are all useful tools for counting and analyzing data. Basic counters are simple tools that can be used to count a variety of items, such as the number of words in a document or the number of people in a room. **Csuros Counter algorithm** is a more specialized tool that is useful for efficiently identifying the most frequently occurring elements in a data set. The **Freq Counter** is another specialized tool that is useful for counting the number of floating-point numbers in a data set. All of these tools have a number of important applications and can be used in a variety of settings. Overall, these tools are essential for accurate data analysis and can be used to help understand and make sense of complex data sets.

References

[1] Graham Cormode, Marios Hadjieleftheriou, "Finding Frequent Items in Data Streams, http://hadjieleftheriou.com/papers/vldb08-2.pdf

[2] Approximate counting with a floating-point counter https://arxiv.org/abs/0904.3062

[3] Freq Count Algorithm https://www.researchgate.net/publication/221426566