University Of New York Tirana
Algorithms & Complexity

Assignment 2

Vikensa Grabocka

Contents

1 Introduction **1**
1.1 Part 1-Entropy of Language 1
1.2 Part 2-Commit Owners 1

2 Part 1- Entropy of Language **2**
2.1 Data-Structures 2
2.2 Pseudocode 2
2.3 Proof of correctness 5
2.4 Time Complexity 7
2.5 Results 9

3 Part 2-Commit Owners **11**
3.1 Data Structures 11
3.2 Pseudocode 11
3.3 Proof of correctness 13
3.4 Time Complexity 13

Bibliography **15**

1 Introduction

This assignment consists of two parts: Entropy of Language and Commit Owners. It is implemented in Java and consists of two packages that contain the solution of each part, respectively. Moreover, a folder is part of the project. It contains the text files that are required for the first part of the assignment.

1.1 Part 1-Entropy of Language

The first part of the assignment measures the entropy of the Albanian language. For this purpose, it uses two text files: one that belongs to the encyclopedia category and the other that belongs to the newspaper category. Both these files are located in a separate folder inside the project. These text files are used to construct the n-gram models of Albanian language, where n lies between 0 and 10 inclusive. For each model it is displayed the entropy, the total number of tokens,five most frequent tokens and five least frequent tokens. These metrics are displayed for each category and for the entire language. Afterwards, the user can enter a sentence and the program will find if it is in Albanian or English language.

This part is constructed from two classes, one that contains the logic related to the calculation of entropy and the other that contains the main class, which displays the results and performs the interactions with the user.

1.2 Part 2-Commit Owners

The second part of the assignment is designed to solve the problem of extracting the employees that have performed commits in a distributed version control platform, given as input the weld, which is the concatenated value of committers' id and a text file which contains the necessary data related to the employees' names, surnames and ids.

This part is constructed from 2 classes, one that contains the logic that is related to the solution of the problem and the other which contains the main class the performs the interaction with the user. The user is prompted to insert the path of the file that contains the data related to the employees and the value of the weld. Afterwards, the configuration with the most commits is displayed and the number of all possible decompositions.

2 Part 1- Entropy of Language

2.1 Data-Structures

The data structures used for the purpose of this part of the assignment are:

1) Map <String,Long> , more specifically a HashMap <String,Long> which is used to keep track of the frequencies of the tokens that appear in the n-gram model. The Map data structure is used to associate keys to values, therefore it is the most suitable data structure for the purpose of this part of the assignment where each token should be associated with its corresponding frequency. Long data type is used to avoid overflow that might result from large values. Moreover, the HashMap data structure allows constant time complexity O(1) for accessing the elements. This feature is very important, especially in this context where large text files are scanned.

2) List <String> is used to collect the five most frequent tokens and the 5 least frequent tokens. Lists can be easily iterated and the elements can be easily accessed when printing them. For the purpose of storing the 5 most/least frequent tokens the list data structure is a suitable one.

3) String[] . An array of strings is used to store the words of one line of text. This data structure is the most suitable data structure for this purpose since it is compatible with the split method that the String data type has. Moreover, when splitting a line of text the number of words is known immediately, therefore arrays can be used, avoiding in this way using extra memory, which is needed for example in the case of ArrayLists.

2.2 Pseudocode

Algorithm 1 Normalize Text

Input: String line, a line of text
Output: a normalized line of text that contains only lowercase letters.

```
1: function NORMALIZE-TEXT(line)
2:     line ← (lowercase(line))
3: if line contains non-letter characters then
4:         non − letter character ← " "
5: end if
6:     return line
```

Algorithm 2 Calculate Token Frequency

Input: String line, a line of text, nGram, a number that represents the n-gram size
Output: a map that contains the frequencies of each token

```
1: function CALCULATE-TOKEN-FREQUENCY(line, nGram)
2:     line ← (normalize − text(line))
3:     tokenFrequency ← []
4:     words[] ← split(line)
5: for word in words do
6:    if length(word) > nGram then
7:       for i=0...length(word)-nGram do
8:             token ← word[i : i + nGram]
9:          if  token in tokenFrequency then
10:            tokenFrequency[token] ← tokenFrequency[token] + 1
11:         else
12:            tokenFrequency[token] ← 1
13:         end if
14:      end for
15:   else
16:      token ← word
17:      if token in tokenFrequency then
18:         tokenFrequency[token] ← tokenFrequency[token] + 1
19:      else
20:         tokenFrequency[token] ← 1
21:      end if
22:   end if
23: end for
24:    return tokenFrequency
```

Algorithm 3 Calculate Entropy

Input: numLetters, a number that corresponds to the letters of the alphabet of a particular language, tokenFreq, a map that contains the frequencies of each token, nGram, the number that corresponds to the n-gram model size
Output: the value of entropy

```
1: function CALCULATE-ENTROPY(numLetters, tokenFreq, nGram)
2:     entropy ← 0.0
3: if nGram = 0 then
4:        prob ← 1/numLetters
5:    entropy ← log_2 prob
6: else
7:    totalFreq ← sum(values of tokenFreq)
8:    for token in tokenFreq do
9:       prob ← tokenFreq[token]/totalFreq)
10:      entropy ← entropy + prob · log_2 prob
11:   end for
12: end if
13:    return −entropy
```

Algorithm 4 Get 5 most frequent tokens

Input: freqToken, a map that contains the token values along with their corresponding frequencies
Output: a list that contains 5 most frequent tokens.

```
1: function GET-FIVE-MOST-FREQUENT-TOKENS(freqToken)
2:     top5Tokens ← []
3:     sortedTokens ← SORT BY VALUE DESC(freqToken)
4:     top5Tokens ← first five keys(sortedTokens)
5:     return top5Tokens
```

Algorithm 5 Get 5 least frequent tokens

Input: freqToken, a map that contains the token values along with their corresponding frequencies
Output: a list that contains 5 least frequent tokens.

```
1: function GET-FIVE-MOST-FREQUENT-TOKENS(freqToken)
2:     5lessFrequent ← []
3:     sortedTokens ← SORT BY VALUE (freqToken)
4:     5lessFrequent ← first five keys(sortedTokens)
5:     return less5Frequent
```

Algorithm 6 Get total number of tokens

Input: freqToken, a map that contains the token values along with their corresponding frequencies
Output: a number that corresponds to the total number of tokens

```
1: function GET-TOTAL-NUMBER-OF-TOKENS(freqToken)
2:     totalFreq ← 0
3: for token in freqToken do
4:         totalFreq ← totalFreq + tokenFreq[token]
5: end for
6:     return totalFreq
```

Algorithm 7 Calculate resemblance

Input: unkownLanguageToken, a map that contains n-gram frequencies of an unkown language, freqToken, a map that contains the n-gram frequencies of a known language
Output: a value that indicates the resemblance between two languages

```
1: function CALCULATE RESEMBLANCE(unknownLanguageToken, freqToken)
2:     componentProduct ← unknownLanguageTokens
3: for token in componentProduct do
4:   if token in freqToken then
5:         componentProduct[token] ← componentProduct[token] · tokenFreq[token]
6:   else
7:     componentProduct[token] ← 0
8:   end if
9: end for
10: numerator ← sum(values componentProduct)
11: sum1 ← sum((values freqToken)^2)
12: sum2 ← sum((values unknownLanguageToken)^2)
13: resemblance ← numerator/(√sum1 · √sum2)
14:     return resemblance
```

2.3 Proof of correctness

The entropy is a statistical parameter which measures, how much information is produced on the average for each letter of a text in the language. If the language is translated into binary digits (0 or 1) in the most efficient way, the entropy is the average number of binary digits required per letter of the original language [1]. The entropy is calculated using the following formula:

$$E(X) = -\sum_{x \in X} p(x) \cdot \log_2 p(x)$$

Since we are not interested in punctuation and may treat the text as case insensitive, first of all we need to normalize the text (to convert it to lowercase letters and to remove non-letter characters. This is done in Algorithm 1.

Afterwards, the n-gram tokens should be generated. An n-gram is an adjacent sequence of n letters in the words of the text. In Algorithm 2 the tokens for each line of text are generated along with their corresponding frequencies.

Firstly, the line of text is normalized, afterwards it is split into words and then for each word is checked whether its length is greater than the size of n-gram model. If the length of the word is greater than the size of the n-gram model, an inner loop iterates starting from the first character up to the length of the word minus the size of n-gram character. In this loop all adjacent sequences of n letters are computed, starting from characters 0 to n, 1 to n+1 until the last sequence of adjacent n letters is constructed: the characters in the last n positions of the word. Then, it is checked whether the token already exists in the map that keeps track of the tokens and their corresponding frequencies or not. If it exists the corresponding frequency is incremented, otherwise a new entry is added with frequency 1. If the length of the word is less than the size of the n-gram the entire word is considered as a token and again is performed a check in the map that contains all the tokens along with their corresponding frequencies. If the token exists its frequency is incremented, otherwise a new entry is inserted. This process is repeated for all the text lines. In this way the n-gram tokens are constructed, which complies with their definition.

After the tokens and their corresponding frequencies are computed entropy is calculated (Algorithm 3). Firstly, it is checked whether the n-gram size is equal to 0. Given that the 0-gram model does not take into consideration the frequencies of each letter, the letters of the alphabet are assumed to be normally distributed, therefore the probability of each letter is equal to

$$p(X = x) = \frac{1}{n}$$

where n are the number of letters of the alphabet. Therefore the entropy formula becomes:

$$E(X) = -\sum_{x \in X} \frac{1}{n} \cdot \log_2(\frac{1}{n})$$

$$E(X) = -(n \cdot \frac{1}{n} \cdot \log_2(\frac{1}{n}))$$

$$E(X) = -(\log_2(p(X = x)))$$

which is exactly what it is returned in Algorithm 3 when n-gram size is equal to 0.

When n-gram size is not equal to 0, firstly is calculated the total number of frequencies by summing the frequency of each token. The probability of an event is found by the formula:

$$P(A) = \frac{Number\ of\ favorable\ outcomes}{Total\ number\ of\ outcomes}$$

In this case, the probability of each token is found by dividing its frequency by the total number of frequencies. A for loop is used to iterate over all tokens and the probability of each token is calculated by dividing its frequency by the total number of frequencies. After each iteration the product of the probability of each token with the logarithm of the probability base 2 is added to value of entropy. At the end the negation value of the running total of entropy is returned, which complies with the entropy formula.

In order to get the 5 most frequent tokens, the map that contains the tokens along with their corresponding frequencies is sorted in reverse order according to the values of the frequencies and first five elements are retrieved.

In a similar way, to get the 5 least frequent tokens the map that contains the tokens along with their corresponding frequencies is sorted according to the values of the frequencies and first five elements are retrieved.

To obtain the total number of tokens, the frequencies of all the tokens are summed.

The actual implementation of this problem consists of a class that contains as attributes the n-gram size, the number of letters in the alphabet and a Map that contains the tokens along with their frequencies for the corresponding n-gram. In order to obtain the desired results for the n-gram models starting from 0 up to 10 both inclusive, 11 objects are used.

In order to find which language belongs to an unknown sentence a method called Cosine Similarity is used.

The calculation of cosine similarity (COS) between two texts starts with the conversion of the texts into a Euclidean space. Assuming the two texts together have n unique words, the space will have n dimensions and each text is a vector in that space. The value of a text vector on any particular dimension reflects the frequency of the corresponding word in the text. Computationally, the COS score s of two texts A and B is given as:

$$s = \cos\theta = \frac{A \cdot B}{\|A\| \times \|B\|} = \frac{\sum_{i=1}^{n} a_i b_i}{\sqrt{\sum_{i=1}^{n} a_i^2}\sqrt{\sum_{i=1}^{n} b_i^2}}$$

Let θ be the angle between the two vectors $\mathbf{A}$ and $\mathbf{B}$ in an n-dimensional space. $\mathbf{A} \cdot \mathbf{B}$ is the dot product, and $\|\mathbf{A}\|$ and $\|\mathbf{B}\|$ are the lengths (norms) of vectors $\mathbf{A}$ and $\mathbf{B}$, respectively. The terms a_i and b_i denote the i-th elements of $\mathbf{A}$ and $\mathbf{B}$, respectively.

Given that all a_i and b_i are positive, the cosine similarity (COS) scores are bounded between 0 and 1, where:

- COS = 0 means the two texts are completely different (i.e., the two vectors are orthogonal),
- COS = 1 means they are identical in terms of word composition (i.e., the two vectors overlap but may have different lengths) [2].

In Algorithm 7, given two maps that consist of tokens along with their corresponding frequencies, the COS value is calculated. First of all is calculated the dot product of two vectors, which is calculated as the sum of products of their components (frequency of tokens that appear in both texts), if a token exists in the n-gram model of one language but not in the other, the product with respect to that component is 0 since the corresponding frequency in the model that does

not contain it is 0 and the entire product is 0. This is performed in lines 3-10 of Algorithm 7. Afterwards, the sum of squares of the components is computed (the frequencies of the tokens for each n-gram model are raised to the power 2 and summed). Their values are used to find the length of each vector used to calculate the COS value in line 13. This approach complies exactly with the above-mentioned formula.

This algorithm is implemented in the LanguageModel class. In the Main class the user is prompted to enter a sentence and a LanguageModel object is instantiated, which contains as attribute the token frequency for the bigram model of the sentence. On the other hand, a LanguageModel object that contains the token frequency for the bigram model in Albanian language is instantiated, and another one that contains the token frequency for the bigram model in English language is instantiated. This method is used to calculate the COS value of the sentence model and English model, and the Albanian model and the sentence model. If the values that result from this method are both 0, it means that the text is neither in Albanian nor in English. Otherwise the sentence is in the language that the highest COS value was obtained. (Note, to obtain the bigram of the English language and Albanian language the text files located in a folder inside the project are used.)

2.4 Time Complexity

Let c, be the total number of characters that the text contains, let w the number of words that it contains and let n be the n-gram size of the model.

Firstly, the text should be normalized meaning that each character should be processed, so the time complexity of Algorithm 1 is O(c).

In order to construct the n-gram tokens, the text should be split into words, splitting the text into words has O(c) cost since each character should be scanned to check whether it is a space character or not.

Afterwards, for each word, n-gram tokens should be generated . In the best case the length of the word is less than the size of the n-gram model, therefore there will be only one token for that particular word (the entire word itself). The token will be checked in the map whether it already exists or not, operation that has O(1) time complexity since a HashMap is being used and accessing and inserting an element in the HashMap has O(1) time complexity. Therefore in the best case the body of the loop in line 5 of Algorithm 2 will have O(1) time complexity. It will run w times for each word and the time complexity will be:

$$w \cdot O(1) = O(w)$$

In the worst case the length of the word is greater than the size of the n -gram model. Therefore, each word will be traversed number of characters that it contains minus n-gram size plus 1 to generate the n-gram tokens. Let's denote the length of each word by

$$l$$

. Therefore the for loop in line 7 will be executed

$$O(l - n + 1)$$

times. Afterwards each token will be checked if it already exists in the HashMap, an operation that has O(1) time complexity. Therefore, the for loop in line 7 will have

$$O(l - n + 1)$$

time complexity. The for loop in line 5 will run w times (one time for each word).

$$\sum_{i=1}^{w}(\ell_i - n + 1) = \sum_{i=1}^{w}\ell_i - w(n-1) = c - w(n-1)$$

The time complexity of Algorithm 2 will be:

$$O(c - w \cdot n + w)$$

which is linear in terms of the size of the text and the n- gram size.

The time complexity of entropy calculation (Algorithm 3) in the best case is O(1), when n=0 the entropy is calculated according to the above-mentioned formula (Algorithm 3, line 5). The frequency of the tokens is not taken into consideration. In the worst-case it's time complexity will be

$$O(c - w \cdot n + w)$$

since for each token the probability will calculated and the entropy formula will be applied. Basically, the number of tokens in the worst case is

$$\sum_{i=1}^{w}(\ell_i - n + 1) = \sum_{i=1}^{w}\ell_i - w(n-1) = c - w(n-1)$$

Therefore, the time complexity of Algorithm 3 is linear in terms of the text size and n-gram size.

Regarding the Algorithm 4 and 5, getting the 5 most frequent and 5 least frequent tokes, the map that contains the tokens is firstly sorted according to the values of frequencies. This operation has

$$O((c - w \cdot n + w)\log n)$$

time complexity. Afterwards 5 first elements are retrieved. This operation has

$$5 \cdot O(1) = O(1)$$

time complexity. Overall, getting five mos/least frequent tokens has

$$O((c - w \cdot n + w)\log n)$$

time complexity.

The algorithm 7 which calculates the resemblance between two language models of the same n-gram size, has time complexity linear in terms of the number of tokens. If we take into consideration that this method is used only with the bigram model, then in the worst case both language models will have

$$\sum_{i=1}^{w}(\ell_i - 2 + 1) = \sum_{i=1}^{w}\ell_i - w = c - w$$

tokens. The for loop in line 3 will run c-w times. The calculations in the if-else block have O(1) time complexity. Lines 10 up to 12 have time complexity linear in terms of number of tokens:

$$O(c - w)$$

Line 13 has O(1) time complexity. Therefore, taking into consideration that Algorithm 7 will use bigram models, its time complexity is

$$O(c - w)$$

Otherwise, if it is used with an n-gram model, its time complexity in the worst case will be:

$$O(c - w \cdot n + w)$$

2.5 Results

The results obtained from the text files are as follows:

n-gram size	Entropy
0	5.17
1	4.27
2	7.67
3	10.36
4	11.99
5	12.67
6	12.71
7	12.47
8	12.10
9	11.73
10	11.42

Table 2.1: Entropy of encyclopedia category in Albanian Language

n-gram size	Entropy
0	5.17
1	4.27
2	7.66
3	10.33
4	11.88
5	12.47
6	12.47
7	12.20
8	11.82
9	11.46
10	11.16

Table 2.2: Entropy of news category in Albanian Language

n-gram size	Entropy
0	5.17
1	4.28
2	7.66
3	10.38
4	12.01
5	12.72
6	12.78
7	12.54
8	12.18
9	11.81
10	11.49

Table 2.3: Entropy of Albanian Language

What is noticed is that for n-gram sizes greater than 6, the entropy decreases. This decreasing trend of entropy continues even for n-gram sizes larger than 10. When the size of the n-gram model is increased, the predictability of the text increases; basically, the model improves as the

size of the n-gram is increased. However, its complexity increases as the size of the n-gram model increases because more unique tokens are created.

Another important point to note is that the entropy of the news category is less than the entropy of the encyclopedia category. This means that the text that belongs to the news category is more predictable than the text that belongs to the encyclopedia category.

3 Part 2-Commit Owners

3.1 Data Structures

The data structures used for the purpose of this part of the assignment are:

1) Map <Long,String> , more specifically a HashMap <Long,String> which is used to keep track of the ids that correspond to each employee. Long data type is used so that large id values can be properly handled. The Map data structure is used to associate keys to values, therefore it is the most suitable data structure for the purpose of this part of the assignment where each id should be associated with the corresponding employee. Moreover, the HashMap data structure allows constant time complexity O(1) for accessing the elements.

2) List <String> more concretly LinkedList <String> is used to collect the commit owners. The LinkedList uses as much space as needed to store the objects, in contrast to ArrayLists that use more space that needed. Moreover, arrays could not be used since they have fixed size and the number of commit owners is not known in advance. This LinkedList is used to iterate over all commit owners that it stores, therefore its linear time complexity for accessing a particular element it is not a problem, since this operation is not required.

3) String[] . An array of strings is used to store each piece of data related to an employee. This data structure is the most suitable data structure for this purpose since it is compatible with the split method that the String data type has. Moreover, when splitting a line of text the number of words is known immediately, therefore arrays can be used, avoiding in this way using extra memory, which is needed for example in the case of ArrayLists.

4) char[] . An array of characters is used to store each character from which the weld that contains the employee ids is constructed. This data structure is the most suitable data structure for this purpose since it is compatible with the toCharArray method that the String data type has.

5) long[] . An array of long is used to store for each index i, the number of ways the first i characters of the weld can be interpreted. The long data type is used in order to handle large numbers of decompositions. The array data structure is the most suitable data structure for the purpose of this task, since each element can be access in O(1) time and it is efficient even in terms of memory.

3.2 Pseudocode

Algorithm 8 Create Dictionary

Input: String line, a line of text that contains the data for each employee
Output: a map that contains employee id values along with the corresponding name and surname of the employee

```
1: function CREATE -DICTIONARY(line)
2: employees ← []
3: parts ← []
4: for word in line do
5:     parts ← word
6: end for
7: id ← parts[0]
8: name ← parts[1]
9: surname ← parts[2]
10: employees[id] ← name + surname
11: return employees
```

Algorithm 9 Find Commit Owners

Input: Employees, a map that contains the id for each employee along with the corresponding name and surname, weld a string that contains the concatenated value of the employees' id
Output: a list of commit owners

```
1: function FIND-COMMIT-OWNERS(employees, weld)
2: commitOwners ← []
3:   endIndex ← 0
4:   idCand ← ""
5: while (endIndex < length(weld) ) do
6:     idCand ← idCand + weld(endIndex)
7:   if (idCand in employees) then
8:     commitOwners ← employees[id]
9:     idCand ← ""
10:  end if
11:  endIndex ← endIndex + 1
12: end while
13: return commitOwners
```

Algorithm 10 Calculate number of decompositions

Input: Employees, a map that contains the id for each employee along with the corresponding name and surname, weld a string that contains the concatenated value of the employees' id
Output: the number of possible decompositions

```
1: function CALCULATE-NUM-DECOMPOSITIONS(employees, weld)
2: numDecompositions ← []
3: numDecompositions[0] ← 1
4: for i=1....length(numDecompositions) do
5:   for id in employees do
6:       idLength ← length(id)
7:     if (i >= idLength & weld[i-idLength : i] == id) then
8:       numDecompositions[i] ← numDecompositions[i] + numDecompositions[i −
         idLength]
9:     end if
10:  end for
11: end for
12: return numDecompositions[length(numDecompositions) − 1]
```

3.3 Proof of correctness

Firstly, the dictionary which contains the id values of each employee and the corresponding full name of the employee should be constructed. This is performed in Algorithm 8. Given that each line consists of id, name,surname of one employee, each line is split into its parts so that id, name and surname of each employees is extracted and this data is inserted into the map.

Afterwards, given the weld value, the configuration with the most commits is found using Algorithm 9. Each character of weld is traversed and a pointer keeps track of the end index. The end index refers to the character that should be appended to the variable that holds the id value that should be checked if it is a valid employee id or not. Starting from the first character it is checked whether exists an employee with that id, if yes the corresponding full name of the employee is added to the list that contains the committers and the variable that contains the candidate id is cleared because an employee is found with that particular id and there is no longer need to append other values to it. If the particular id is not found in the map then the consecutive character in the weld is appended to the candidate id and the process starts again checking if an employee exists with that id or not. The process finishes when the last character of the weld is reached. In this way even if more than one configuration exists only the one with the most commits will be computed.

In order to calculate the number of decompositions Algorithm 10 is used. It evaluates in how many ways a concatenation of ids can be configured by breaking the problem into subproblems. It evaluates the number of configurations by taking into consideration for each index i that corresponds to a character of the weld in how many different ways the first i values can be configured. numDecompositions array keeps track of the numbers of ways the first i digits of the weld can be interpreted. The base case is when i = 0, numDecompositions[i]= 1 since there is only one way to interpret the empty string.

Let f(i) be the number of ways the first i characters of the weld can be interpreted:

$$f(i) = \sum f(j) \ where \ j < i \ and \ weld[j : i] \in \ employees$$

A partition should end with a valid employee id, if the employee id has a length k, then it occupies weld[i - k : i]. The remaining part weld[0 : i - k] should also be partitioned in a valid way. According to the above-mentioned formula the weld[0:i-k] can be partitioned into f(i-k) ways. Any valid employee id can be chosen so we can use the sum rule [1] to find all the possibilities. Therefore, the number of possible configurations to interpret the first i characters (weld[0:i]) is:

$$f(i) = \sum f(i - len(id)) \ \forall id \ weld[i - len(id) : i] = id$$

In order to find the possible ways that the weld can be configured, we need to find the possible ways that weld[0: n] (where n is the last character of the weld) can be interpreted. This value is stored in the last index of the numDecompositions array. This is exactly what is performed in Algorithm 10.

3.4 Time Complexity

Algorithm 8 which is used to create the map that consists of employee ids and their corresponding full names runs one time for each line of input, basically it runs one time for each employee.

[1] Rule of Sum states that if there are A ways to do one thing and B ways to do another, and these two actions cannot occur simultaneously, then there are A + B ways to choose one of the actions.

Considering the number of employees as the higher level of abstraction creating the map has O(e) time complexity. Algorithm 9 is used to find the commit owners. The while loop in line 5 runs the length of weld times. Let l be the length of the weld (number of digits that it contains). The body of the while loop has O(1) time complexity since it consists of checking whether the id is a valid employee id or not, this check is performed in a HashMap. Accessing an element of the HashMap has O(1) time complexity. If the id is a valid one, the full name of the corresponding employee is inserted into the list, this operation has O(1) time complexity. Therefore, the time complexity of Algorithm 9 is O(l). Algorithm 10 is used to calculate the number of possible decompositions.The loop in line 4 runs length of weld times (l times), the inner loop in line 5 runs 1 time for each employee, it runs e times. The condition in line 7 checks whether the i value is greater than the length of the current id and whether the substring from index i- length of id up to i is equal to the valid id, extracting this substring requires an iteration that has size length of the id, the maximum value that the length of the id can have is l, therefore, extracting the substring might result in O(l) time complexity in the worst case. The overall time complexity of the Algorithm 10 is

$$O(l \cdot e \cdot l) = O(l^2 e)$$

,where l is the length of the weld (number of digits that it contains) and e is the number of employees.

Bibliography

[1] C. Shanon, "Prediction and entropy of printed english," pp. 50–52, 1950.

[2] K. Guo, "Testing and validating the cosine similarity measure for textual analysis," 2022.