**Part I- Entropy of Language**

1. **General Overview**

The problem presented in this first part of our assignment falls well within the scope of **Natural Language Processing (NLP)**. NLP represents one of the most important subfields of Artificial Intelligence, as it allows machines to understand, interpret, and communicate in human language. Before taking the AI course, our perception of human languages was no more than just a collection of random words, especially when we heard a foreign one. But as we delved deeper into the AI course, we saw that languages are not random. On the contrary, there are many grammatical rules and statistical structures that every human language follows. The simplest observation that can assure us of this is the fact that some words or letter combinations appear more frequently than others. So, when we think of letter and word distributions, we are quick to realize that they are not uniform, and as we will soon see, this non-uniformity can be measured and modeled. It is this simple observation that stands at the foundation of the NLP field, which is especially crucial in powering large language models (LLMs) like ChatGPT that are changing how we humans interact with the digital world.

The scope of the NLP field is large, but many of its core techniques and concepts remain as the foundation of our project. We will establish those connections later when we talk about our general algorithm, but for now, it is crucial to understand what the task at hand is. Our primary goal is to find the entropy of our native language, which in this case for us is Albanian. In general terms, the entropy of a distribution is a measure of how much randomness it contains, how predictable that distribution is. But this definition can seem rather confusing. What are our distributions, and what constitutes randomness and predictability in our case? To answer this, we need to introduce the concepts of n-gram models, which allow us to understand the statistical structure of our language. In this context, our “distribution” refers to how frequently different sequences of characters, known as n-grams, appear in the body of a text. An n-gram is just an adjacent sequence of n characters, and their frequencies allow us to build the probability distributions p(x) over all these sequences. For example, we established before that distributions are not uniform, and in the case of the Albanian language, we see that in a 2-gram model(bigram), the pair “sh” appears more frequently than “xq”. And this way we have explained the other concept “predictability” because, as we saw, some sequences of letters are more predictable or structured (sh is an actual letter) compared to others.

In our case, the project asks that we calculate these entropies for two large Albanian texts (5MB each), and if possible, they should be of different complexities(categories). It also specifies that we need to analyze entropy for both texts across a range of n-gram models, starting from 0 to 10-gram. So when we talk about 0-grams, we assume all tokens of our distribution have equal probability, but this is a theoretical one, as we have already mentioned that this is not the case. In general, an n-gram model considers the frequency of n-grams (subsequences of length n of adjacent letters in the words of the text), and the higher it gets more context is incorporated. We will talk a bit later about what problems arise as n gets too big and will try to explain other important observations when we talk about the algorithm in the next section.

Another way of viewing entropy is as a measure of the average number of bits (theoretically) needed to represent a token of our distribution. But entropy gives us a theoretical measure, and our project asks that we provide the practical one as well (optimal coding). So, to translate this into something practical, we need to apply the concept of Huffman coding introduced in our DPV book. It tells us how much our language distributions can be compressed, and we can use it to see how close our real encoding length approaches the theoretical entropy. Their relationship is well established by the horse race example in our book, which argues that the more compressed a distribution, the less random and more predictable it is (more compressible ≡ less random ≡ more predictable).

The bonus part of our project revolves around the concept of language identification, which utilizes binary classification. Classification (supervised learning) is again a very important concept of AI. A classifier is a function y =f(x), where x=features, y = true label, 𝑦(hat)=estimated label. So, to build this classifier, we need to use training data that is accompanied by labels, and during classification, we predict the class label of some unknown token. In our specific case, we first need to build bigram models for both Albanian and English, which enable us to construct the classifier that predicts the language of an unknown token(sentence). In general, in our project, we don’t consider punctuation and we treat the texts as case-insensitive.

1. **General Algorithm**

We were able to establish in the first section the connection between NLP and the requirements of our project. In this section, we first introduce the key techniques of NLP used in the algorithm, and then we will reason about some important observations. Many of the methods we use, like text preprocessing, n-gram modeling, entropy calculation, and probabilistic classification, are direct implementations of core NLP techniques.

* 1. Preprocessing

Before we could perform any analysis on our texts, we needed to ensure that we considered valid and meaningful data. That included: 1. case normalization, where we converted all characters to lowercase, 2. alphabet filtering, where we made sure to retain only valid letters of our native alphabet, including ë and ç, 3. punctuation and whitespaces removal, where we exclude all non-letter symbols to focus on the language only. This step was implemented by the method TextProcessor.clean().

* 1. N-gram modeling

N-gram models are the foundations of language modeling. In our project, we built n-gram models from n=0 to n=10. These are implemented by the method TextProcessor.extractNGrams(text, n), which returns all overlapping sequences of length n(n-grams) from the cleaned text.

* 1. Entropy calculation

Entropy calculation represents the main goal of our project, and in our implementation, we calculated it using the Shannon formula given: H(X)=−x∑​p(x)log2​p(x). In our code, this entropy calculation is handled by the method NGramModel.computeEntropy(), which treats as x the n-grams we find. P(x) is just count(x) over the total number of tokens.

* 1. Language classification (Bonus)

This is a classic problem of NLP, which in our case, we needed to find the language a given sentence belonged to. We did this using probabilistic bigram models for both the Albanian and English languages, and the concept is similar to that of Naïve Bayes classifiers introduced in our AI course. We implemented this classifier in the LanguageClassifier class, which manages multiple language models and uses BigramLanguageModel.score(sentence) to compute probabilities.

The last key point of our project was not directly related to NLP concepts but to Huffman coding, a classical compression algorithm which we discussed in class. The finding of optical codes is handled by the Huffman Tree class that has methods to build the binary tree, compute code length, and the weighted average length.

Testing our program, we were able to make several important observations, which we will try to present here. One of the key observations was the change in entropy as n increased. Initially, as n increased from 0 to 3 or 4 (depending on the text), we saw a significant drop in entropy. However, as n continued to increase, the rate at which entropy decreased slowed and, in some cases, even increased slightly. Same pattern for Huffman code length. However, when we tried larger values of n, like 9 or 10, we saw that the entropy and length of the Huffman code began to increase. This is because longer n-gram models result in a larger number of possible token sequences. Also, we saw that when comparing the Huffman code length and entropy of each n-gram model, they were actually really close. We were not able to use different text categories, but we believe that if we did so, we would find that the higher the complexity of the text, the higher the entropy and Huffman code length would be, because complex text like scientific papers don’t use many repetitive words as opposed to for example, news.

1. **Algorithm pseudocode and analysis**

The algorithms presented in this section use a similar syntax and reasoning style that is familiar to us from the *Algorithms* book by Dasgupta, Papadimitriou, and Vazirani. In one case, for the Huffman algorithm, we added on top of the existing algorithm to make it work for our specific case. But in general, we try our best to follow as closely as possible the material presented in the book, more specifically the one in section 5.2.

* 1. **Procedure: clean(text)**

clean(text)

result = empty string

for each character c in text:

if LOWERCASE(c) ∈ Alnanian\_Alphabet:

append LOWERCASE(c) to result

return result

This procedure is an implementation of the logic behind NLP text preprocessing that we introduced in the section above. So, this method includes the 3 steps we introduced: 1. case normalization, where we convert all characters to lowercase, 2. alphabet filtering, where we made sure to retain only valid letters of our native alphabet, including ë and ç, 3. punctuation and whitespaces removal, where we exclude all non-letter symbols to focus on the language itself. Its input is a text that contains mixed cases, punctuation, digits, symbols, foreign letters(maybe), and in the end, it returns a normalized text.

**Correctness:** The proof of correctness for this procedure is trivial. It is correct because it makes sure to treat the text as case-insensitive by converting letters to lowercase. Also, it filters out all the other characters that are not present in the Albanian alphabet. Also, this check ensures that punctuations are removed as well.

**Time Complexity:** Let's denote with n the number of characters on our text. The algorithm iterates once through each character. During each iteration, the amount of work it does is constant O(1) because it just converts characters to lowercase, checks if they are in the alphabet(comparison), and appends if so. So, the overall time complexity is O(n).

* 1. **Procedure:** extractNGrams(text, n)

generate\_ngrams(text, n)

ngrams = empty list

for i from 0 to Length(text) − n:

token = Substring(text, i, i + n)

append token to ngrams

return ngrams

This procedure implements the logic behind NLP n-gram modeling. It takes a cleaned text and extracts all overlapping substrings of length n. In the end, it returns a list containing all n-grams. Our n-grams computed here become the tokens x that we need for the entropy formula.

**Correctness:** First, it is correct because our loop condition ensures that all substrings don’t exceed their bounds. The substrings from i to i + n to guarantee us that each n-gram is contiguous (composed of adjacent letters) and overlaps (ex: “abc” = “ab”, “bc”). Also, the method makes sure that all possible substrings of length n are included once in order.

**Time Complexity:** Let n be the size of each n-gram. Our loop runs from 0 to the length of the text – n. At each iteration, the operation is performed substring and appended, which in Java takes constant time. So the overall time complexity is linear O(n).

* 1. **Procedure: count\_freqs(ngrams)**

count\_Freqs(ngrams)

freq = empty map

for token in ngrams:

freq[token] = freq.get(token, 0) + 1

return freq

So, this procedure takes a list of n-grams as input, which we extracted from our previous method, and it counts how many times each unique n-gram appears. It returns a hash map where the key is the n-gram, and the value is the number of times it appears.

**Correctness:** At the start, the freq is initialized as empty. Then we enter a loop, where for each token in the list of n-grams, we return its count. This way we make sure each token is counted once per appearance and all unique keys are stored as keys.

**Time Complexity:** Let n be the number of n-grams on the list. Then our loop iterates over each token of the list, and it performs two constant-time operations, one retrieval of the map and insertions. So, the time complexity is linear, n iterations, and a constant amount of work in each iteration.

**4. Procedure: computeEntropy(),**

computeEntropy(freq, totalTokens)

entropy = 0

for token in freq:

p = freq[token] / totalTokens

entropy = entropy − p \* log₂(p)

return entropy

This procedure is the one that implements the Shannon formula to calculate the entropy. It takes a frequency map we found in the previous method and uses those values to calculate the entropy. In the end, it returns the entropy as a double value.

**Correctness:** It is correct because it is mostly a code implementation of the Shannon formula. It calculates the probability of each n-gram token as the number of that taken over the total tokens. Then it uses these values in the formula. If p=0, we skip it even though we know the frequency of a token is > 0.

**Time complexity:** Let n be the number of unique tokens in the list. We iterate once for each key in freq, and the amount of work done in each iteration is constant. So, the overall time complexity is linear O(n).

**5. Procedure: huffman(freq)**

huffman(freq)

Q = priority queue of all tokens by frequency

while Q.size > 1:

x = Q.extractMin()

y = Q.extractMin()

z = new node with freq = x.freq + y.freq

z.left = x

z.right = y

Q.insert(z)

return Q.extractMin()

This method is the one that gives us the optimal coding. It takes a frequency map of tokens and builds a binary tree (Huffman tree) where each leaf is a token, and an internal node is a combination of two subtrees. It returns an optimal prefix-free code.

**Correctness:** The correctness of this algorithm follows from the explanation given in the book. It has been proven that it generates prefix-free code with the lowest average code length.

**Time Complexity:** Let n be the number of unique tokens in the frequency maps. We perform 2 crucial operations. One is the construction of the priority queue which takes linear O(n) time and the other merges steps(extract/insert), each of these steps taking O(log n) time. So the overall time of our algorithm is O(n log n ).

**6. Procedure: code\_Lengths (tree, depth, map)**

Code\_Lengths(node, depth, map)

if node is a leaf:

map[node.symbol] = depth

else:

Code\_Lengths(node.left, depth + 1, map)

Code\_Lengths(node.right, depth + 1, map)

This procedure comes after we build the Huffman tree using our previous method. It translates the tree structure into code lengths for each token. The method traverses the Huffman tree and assigns a code length for every token by recording the depths of its corresponding leaf in tree. It also builds a map: token: code\_length (eg. “sh” :2).

**Correctness:** Its correctness follows closely from the correctness of DFS, because our method is a recursive DFS traversing a binary tree. At each call, we increment the depth by 1, and when we reach a leaf, we record its depth. If only one token exists, its depth is 0.

**Time Complexity:** Let n be the number of tokens (leaves on our Huffman tree). During iteration, each node in the binary tree is visited only once. So the overall time complexity is linear O(n).

**7. Procedure: avg\_Huffman\_Code\_Length (freq, codeLen, totalTokens)**

avg\_Huffman\_Code\_Length(freq, codeLen, totalTokens)

avg = 0

for token in freq:

p = freq[token] / totalTokens

avg = avg + p \* codeLen[token]

return avg

This procedure helps us calculate the average code length using the previously built frequency maps and code lengths. It receives freq: map of tokens and frequencies, codeLen: map of tokens to their Huffman code lengths (in bits), totalTokens: total number of n-grams. The procedure then calculates the average code length for encoding a token of our distribution. In the end, it returns a single value, which is the average number of bits per token.

**Correctness:** It is correct because we first compute p(x) and then we multiply each of these probabilities by its assigned code length. In the end, we accumulate the weighted sum into the result.

**Time Complexity:** Let n be the number of unique tokens in the frequency map. We iterate over all of them, and during each iteration, we perform constant-time operations. So, the overall time complexity is O(n).

**8. Procedures: TOP\_K(freq, k) / BOTTOM\_K(freq, k)**

TOP\_K(freq, k)

return top k from freq sorted by descending count

BOTTOM\_K(freq, k)

return top k from freq sorted by ascending count

These two simple methods allow us to output the statistics required by the project. Their input is a freq map of tokens to their frequencies and k: the number of results to return. The output for TOP\_K is the k most frequent tokens in descending order, while for BOTTOM\_K, the opposite in ascending order.

**Correctness:** They work by sorting map entries by their frequency values. Then they either pick the top or the bottom k. Two edge cases exist:1. if k > the number of tokens, they return all sorted, and 2. if k = 0, they return an empty list.

**Time Complexity:** Let n be the number of unique tokens in the frequency map. We perform sorting of all these tokens, which takes O(n log n) time, and slicing the top or bottom, which takes linear time. But sorting dominates, so the overall complexity is O(n log n).

**9. Bonus Procedure: score(sentence, freq, totalTokens)**

score(sentence, freq, totalTokens)

bigrams = generate\_Ngrams(clean(sentence), 2)

score = 0

for b in bigrams:

p = freq.get(b, 0) / totalTokens if b exists else ε

score = score + log₂(p)

return score

This procedure computes the log likelihood score, representing how well a given sentence matches a trained bigram model. It is used for the bonus classification task. Its input is a sentence to classify, a frequency map of bigrams from the training corpus, and the total number of bigrams in the model. In the end, it returns a real number representing the log probability score.

**Correctness:** The correctness of this algorithm follows from the correctness of the methods it calls, which we have already proven. Also, we calculate the scoring value by iterating over each bigram. In each iteration, we look up its count and compute its score.

**Time Complexity:** Let n be the number of characters in our cleaned sentence. Then, from previous analysis, we know that cleaning has time complexity O(n), bigram extraction O(n), and scoring linear as well as O(n). So, the overall time complexity is linear O(n).

**4. Data structure analysis**

In this section, we will try to provide all the data structures we used in our program and indicate where in the program we used them(in which classes). We will consider each class separately, and for each class, we will show the key data structures(as they are implemented in code) and then provide the reason (justification) why we chose them.

**1. Frequency Table (HashMap)**

This data structure appears in class NgramModel and is expressed as **Map<String, Integer>**. Its main purpose is to store how many times each n-gram appears. Also, it allows us to increment the count and retrieve the frequencies. HashMap was preferred here because it allows for fast average-case access and update times. We use it not only for computing entropy but for Huffman code calculation as well.

**2. Token Sequences (ArrayList)**

This data structure appears in the class TextProcessor and is expressed as **List<String>.** The main purpose of it is to collect all n-grams from the cleaned text. This is needed in the extractNGrams() method. Also, some of the key operations we utilize for this are append and iterate. ArrayList was perfect here because it preserves the order of appearance and allows sequential processing of all tokens.

**3. Huffman Tree (PriorityQueue)**

This data structure is part of the class HuffmanTreeand and is expressed in this form: **PriorityQueue<HuffmanNode>.** It is used in the build method of this class. We chose the PriorityQueue because it efficiently picks the two least frequent tokens for merging. For this, we utilize insert and extractMin operations. PriorityQueue is essential here because it enables efficient merging and is essential for the Huffman algorithm logic.

**4. Code Length Table (HashMap)**

This data structure is part of class HuffmanTree and is used in the method to compute the code lengths. It appears like this: **Map<String, Integer>.** HasMap is essential here because is allows us to store the bit-length, the depth of each symbol in the tree. For this we utilized its insert and reading methods.

**5. Albanian Alphabet (HashSet)**

This data structure appears in the class TextProcessor and is expressed as **Set<Character>.** The main purpose of it is to filter only valid Albanian letters. This is needed in the clean() method of this class. HashSet made sense here because it allows for fast membership check (contains()), and this makes it possible to filter unwanted characters.

**6. Language Model Registry (HashMap) -Bonus**

This data structure is part of the class LanguageClassifier and is expressed in this form: **Map<String, BigramLanguageModel>.** It is used to associate each language name (e.g., "Albanian") with a model. For this, we utilize its put, get and iterate operations. HashMap is essential here because it enables fast lookup and can scale to multiple languages(ex: English).

**7. Bigram Frequency Table (HashMap)**

This data structure is used in BigramLanguageModel class and is expressed as **Map<String, Integer>.** It is used to store bigram frequencies for scoring. For this, we used lookup and division operations. HashMap was reasonable here because with it we could reuse this structure, which is crucial for computing token probability in test sentence.

**8. Sentence Tokens (ArrayList)**

This data structure is used in the score() procedure called from BigramLanguageModelclass and is expressed as **List<String>.**  It is used to tokenize and score the sentence by iterating. We use ArrayList here because it is efficient, simple and suitable for a token list of sentences.

**Part 2- Entropy of Language**

1. **General Overview**

The second part of our assignment represents a beautiful problem that is rooted in key algorithmic ideas like dynamic programming, recursion, and string manipulation, all of which we were introduced to from our DPV book. At first glance, the problem may seem like a simple case involving file parsing and string manipulations. However, as we will soon see, this is not the case. It represents a string segmentation problem that can be solved using dynamic programming (DP) techniques.

So, our project revolves around a version control platform (like GitHub) that a company uses to host the code of their software development. The employees of this software company can commit their work to the platform. We have access to the employee's name, surname, and ID. Each commit made is associated with the employee ID. For a specific time period, the platform produces the weld, which is the concatenated value of the committer’s ID. Our main goal is to find which employees have contributed (committed) to a project based on this weld. The weld is not separated in any way to indicate to us which employee contributed; it is just a numeric value.

Our input for this project consists of two parts: 1. a file that has one employee per line, and each of these employees has an ID (numeric), surname, and name separated by a single comma; 2. a weld value. The problem then asks us to extract the employees who performed the commits. If more than one configuration is possible, we need to find out the one with the most commits. So, simply put, we need to decompose the weld into valid IDs, so the number of commits is maximized. But this is not just a parsing(decomposition) problem because there are a bunch of hidden complex subproblems that we need to address. The first being that the ID's value can be of different lengths. Another complexity is the fact that we mentioned in describing the weld, which we said had no separators, meaning there could be multiple valid ways to split it. And the third is that we need to return the split with the maximum number of commits, not just one of the valid ones. The bonus part asks us to count all these valid splits(decompositions), which is an enumeration problem similar to the BigWeather one.

Another way for us to see this problem is to imagine the weld as being a DAG. Our main problem is similar to finding the longest path from node 0 to node n. And the bonus part asks us to count all the paths that start at node 0 and end at n. Also, our problem is similar to other dynamic problems introduced in chapter 6, especially 6.2- 6.4, in which we try to solve the problem by first breaking it down into sub-problems, saving their results, and using those to solve the overall problem. As we will see, this is the best approach, and trying to solve this by brute force will result in exponential time complexity. As we will explain later, to solve this problem, we need to use a combination of strategies introduced in chapters 6 and 8.

1. **General Algorithm**

In our previous section, we established the fact that our problem is fundamentally a string decomposition problem that requires a dynamic programming solution. We will try to give a general description of our algorithm here, emphasizing how we used what we learned from the DPV book to solve our problem. As such, our solution can be broken down into five core components: input parsing, recursive formulation, dynamic programming, path reconstruction, and valid decomposition counting.

* 1. **Input Parsing**

The first thing we need to do is to read the employee file and gather from it all the IDs. We store those IDs in a hash data structure because it gives us constant time for lookups. This is essential because, as we will see in our recursive segment, we will need to frequently check whether a substring is a valid ID.

* 1. **Recursive Formulation**

The main idea comes from chapter 6 editDistance(x, y), LIS(i), and the knapsack(i, w) functions, where problems are solved by breaking them down into subproblems and then combining them recursively to find the solutions. In our program, we have a recursive method, dp(i), that calculates the maximum number of valid IDs that can be formed from the weld[i….n-1]. To compute dp(i), we consider every substring weld[i..j-1] where j goes from i+1 to n. We then check whether the substring is a valid ID. If so, we recursively compute dp(j) and then we take the maximum of 1 +dp(i) over all j.

* 1. **Dynamic Programming and Memorization**

Since the first chapter with the example of computing Fibonacci by using the recursion naively we saw how it leads to exponential time complexity, because it recomputes the same subproblems many times. In order not to fall for this, we need to use memorization, which in our case, means the results of dp(i) are stored in an array and reused if the same problem is encountered again later. As we will see in the next section, this brings the complexity down to O(n \* L) where n is the weld length and L is the maximum ID length. The main idea behind this is that we have n subproblems and during each iteration we try at most L substrings.

* 1. **Path Reconstruction**

Besides computing the maximum number of commits by using dp(i) we also want to reconstruct the actual sequence of IDs that gives that maximum result. We do this by simply keeping track of the best path during the recursive calls. This allows us to recover the optimal decomposition at the end.

* 1. **Counting all Valid Decompositions**

In the bonus part, we are asked to find how many ways we can split the weld string into valid IDs. Here, we sum up all possible valid paths instead of maximizing the number of commits. So, this represents an enumeration problem. We solve this problem by creating a method named count(i) that returns the number of valid decompositions of our weld. This approach kind of mirrors the knapsack solution, because there we ask how many ways instead of what the maximum value is.

**3. Algorithm pseudocode and analysis**

In this section, we present all the core algorithms we need to solve our Commit Owner problem. As always, we tried to follow the syntax and logic introduced in our “Algorithms” book by Dasgupta, Papadimitriou, and Vazirani (DPV). Also in this section, we tried to present the input and output sections in our pseudocode definitions in a similar way to the book. For each pseudocode we present we will give a general description of it and then reason about its correctness and time complexity.

* 1. **Procedure: readEmployeeFile(filename)**

procedure readEmployeeFile(filename)

**Input**: filename, path to employee file

**Output**: validIDs, a set of all employee IDs

validIDs = empty set

open file at filename

for each line in file:

tokens = split line by ','

id = tokens[0]

add id to validIDs

return validIDs

This procedure is the first to be executed. It is responsible for reading the employee file and extracting from that file all the employee IDs and storing them in a set. This is crucial because we will need to lookup this set(table) for validating substrings during weld decomposition, as we mentioned earlier.

**Correctness:** Our procedure is correct because we know that each line of the file contains the employee ID, name, and surname. We split each line by a comma(because in the file they are separated by a comma) and we store the first token, only the ID, in our set. If both file reading and string splitting are done correctly when we implement it in Java, then our procedure will accurately collect all IDs.

**Time complexity:** Let n be the number of lines in our employee files, basically the number of employees. During each iteration, reading the file and processing each line takes constant time. So, the overall running time is linear O(n).

* 1. **Procedure: maxCommits(weld, validIDs)**

procedure maxCommits(weld, validIDs)

**Input**: weld, string of digit validIDs, a set of valid employee IDs

**Output**: list of IDs forming the maximum commit sequence

n = length(weld)

memo = array[0..n] of UNDEFINED

function dp(i):

if i = n:

return (0, empty list) // base case: end of string

if memo[i] is defined:

return memo[i]

maxCommitsHere = -infinity

bestPath = empty list

for j = i+1 to n:

token = weld[i..j-1]

if token in validIDs:

(commits, path) = dp(j)

if 1 + commits > maxCommitsHere:

maxCommitsHere = 1 + commits

bestPath = [token] + path

memo[i] = (maxCommitsHere, bestPath)

return memo[i]

(\_, resultPath) = dp(0)

return resultPath

This procedure closely mirrors editDistance, knapsack and LIS methods from our DPV book. It represents the core method for solving the base case of our problem. We used top down DP- programming to find all the decompositions of the weld string. This returns the maximum number of valid IDs. We are not going to explain the dp(i) function again here as we did in great detail in the previous section; however is important to remember that it returns the best decomposition, which for us is the one containing the largest number of valid IDs.

**Correctness:** Our procedure is correct as it makes sure to try all substrings. When it finds a valid ID, it calls the dp(j) method recursively to compute the optimal decomposition of the remaining string. As we mentioned earlier, it ensures that we keep track of the decomposition with the highest number of commits between recursive calls. This way we know that we consider every possible decomposition and return the best one. This is thanks to memorization, which ensures that all subproblems are solved exactly once.

**Time Complexity:** Let n be the length of our weld string and L be the maximum length of an employee(ID). We know there are n subproblems for each position, say x, and for each of x positions we (during iteration) check up to L substrings. But checking just takes constant time, so the overall running time is O(n\*L).

* 1. **Procedure: countDecompositions(weld, validIDs)**

procedure countDecompositions(weld, validIDs)

**Input**: weld, string of digits validIDs, set of valid employee IDs

**Output**: integer count of all valid decompositions

n = length(weld)

memo = array[0..n] of UNDEFINED

function count(i):

if i = n:

return 1 // base case: one valid decomposition

if memo[i] is defined:

return memo[i]

total = 0

for j = i+1 to n:

token = weld[i..j-1]

if token in validIDs:

total = total + count(j)

memo[i] = total

return total

return count(0)

This procedure is used for the bonus part of our assignment. As we mentioned in the general algorithm section, to find the number of all valid decompositions. Here, this is done by utilizing the count(i) method that returns as a value how many valid ways are there to split our weld string.

**Correctness:** Just like the previous method we explored dp(i), count(i) considers all substrings. For each of these substrings, we count how many of valid decompositions there are starting from the next position. This is done recursively, and our base case returns 1 when the end of the string is reached. This means that we found a valid decomposition path. Her again to avoid recomputing the same subproblems, we use memorization.

**Time Complexity:** Following the same analogy as for the dp(i) method, we find that the time complexity is also O(n\*L).

* 1. **Procedure: countDecompositions(weld, validIDs)**

procedure printCommitIDs(commitIDs)

for id in commitIDs:

print id

This is a simple procedure that satisfies the output condition given in the project. It just prints the sequence of committers, one per line.

**Correctness:** Its correctness is trivial; we just perform a loop that iterates over valid IDs. This being said, if IDs are valid, it is guaranteed to work as it should.

**Time Complexity:** Let n be the number of valid IDs. We iterate n times over all IDs, and during each iteration, we just print something, which is a basic computer operation, performed in constant time. So, the overall running time is linear O(N).

**4. Data structure analysis**

In this section, like the one for Part 1, we will try to present all the data structures we used and also, we will try to justify our reason for choosing them. We will try to specify where in our code they appear and how they appear (their specific name).

**1. Valid ID Storage (HashSet) - Set<String>**

Hashset is used throughout our program in 3 important methods: InputProcessor.readEmployeeFile(), CommitParser.maxCommits(), CommitParser.countDecompositions(). Its main role is to store all the valid IDs. HashSet represented the best choice because it allows for constant time substring lookups. The main reason we chose HashSet is that the main operation of our program was checking whether a substring of the weld string was indeed a valid employee ID. We mentioned that we needed to access them repeatedly, and this needs acsses need to be efficient, which HashSet provides. That is why we didn’t consider other storage options like Lists or ArrayLists.

**2. Memoization (Array) - Result[] or Integer[]**

This data structure is used as Result[] memo in the method maxCommits() and as Integer[] memo in countDecompositions() method. We use Arrays because, in dynamic programming, we need to store solutions of overlapping subproblems. Arrays also allow for constant access and update time, and the size is linear, which is okay as the length of weld n is not too large.

**3. Custom Result Class – Tracking Commit Count and Path**

This is a composite class used in CommitParser.maxCommits() (internal class Result). We do this because the dp(i) method returns both the count of comments and the actual list of IDs used to get that count. This avoid many parallel data structures and make it so our recursive logic(dp) is easy understandable.

**4. List<String> – For Constructing Paths**

This data structure is used in Results. path to store the sequence of IDs that make up the current optimal path. We choose List because it allows for dynamic resizing, as we know the values of the optimal path stored there can increase as the iterations progress.