COMPLEXITY PROJECT
Splitting sentences into words

Yunhan Ma, Liliana Jalpa Pineda, Elsa Schalck

February 2020

Contents

1 Introduction **3**

2 A summary on existing string matching algorithms **3**
- 2.1 Naive algorithm . . . 3
- 2.2 Rabin-Karp . . . 4
- 2.3 Knuth-Morris-Pratt . . . 5
- 2.4 Conclusion on string matching algorithms . . . 5

3 Word segmentation **5**
- 3.1 Algorithms to browse the sentence . . . 6
  - 3.1.1 Recursion algorithm and backtracking - a naive approach . . . 6
  - 3.1.2 Dynamic programming and pruning . . . 6
  - 3.1.3 Sliding window approach . . . 7
- 3.2 Word research in the dictionary : the necessity of a well constructed dictionary . 7
  - 3.2.1 The specificity of the Chinese dictionaries . . . 7
  - 3.2.2 Dictionary retrieval and description . . . 7
  - 3.2.3 Sorting and using dichotomy . . . 7
  - 3.2.4 The prefix tree structure . . . 7
  - 3.2.5 The radix tree structure . . . 8

4 Our approach to Chinese word segmentation **9**
- 4.1 First implementation : using recursion and backtracking with dichotomy . . . 9
  - 4.1.1 Implementation of the dictionary (script list_dictionary.py) . . . 9
  - 4.1.2 Description of the algorithm (script naive_algorithm.py) . . . 9
  - 4.1.3 Performance . . . 9
- 4.2 Second implementation : Word search guided by a well constructed tree, dynamic programming and pruning . . . 10
  - 4.2.1 Implementation of the dictionary (script tree_dictionary.py) . . . 10
  - 4.2.2 Description of the algorithm (script tree_pruning_algorithm.py) . . . 11
  - 4.2.3 Performance . . . 12
- 4.3 A quick recap on the complexity of these two algorithms . . . 14
- 4.4 Going a step further : Searching the best cut in Chinese Word Segmentation . . . 14

5 Conclusion **15**

1 Introduction

The problem we are going to be focusing on this paper is splitting sentences into words. This research topic is also known as the *Word Break Problem* and is a frequent interview question for famous Google and Amazon. The objective of the algorithm is to return all possible cuts of a sentence into words from a specific dictionary, regardless of the meaning of the sentence.

We decided to focus on the processing of texts written in languages that do not use delimiters between the words, like Chinese. Our goal is to be able to split sentences into words with a focus on time complexity of the algorithms.

2 A summary on existing string matching algorithms

To address this problem, we first studied string matching algorithms: their implementation consists in finding a pattern in a larger ensemble. Pattern recognition has, indeed, many applications:

- Text processing, information retrieval, information analysis
- Bioinformatics
- Detect plagiarism
- Security

And the list goes on and on [10].

Researchers have been focusing on string matching algorithms for more than half a century, leaving aside the naive algorithm, enhanced performance algorithms have been proposed ever since 1969. We will explain some of these algorithms and summarize on their associated time complexity.

2.1 Naive algorithm

The naive algorithm, also known as naive brute force algorithm, is the simplest algorithm to find a string on a large body of text. The inputs for the algorithm are: a large body of text T and string P for which we will try to find a match. The algorithm tests every possible position where the word can be.

Algorithm 1 Naive brute force algorithm - pseudo code [10]

```
1: procedure NaiveStringMatching(T,P)
2:     n ← length of T
3:     m ← length of P
4:     for  s = 0 > to (n − m)
5:     if P[1...m] == T[s+1...s+m] then
6:         return True
```

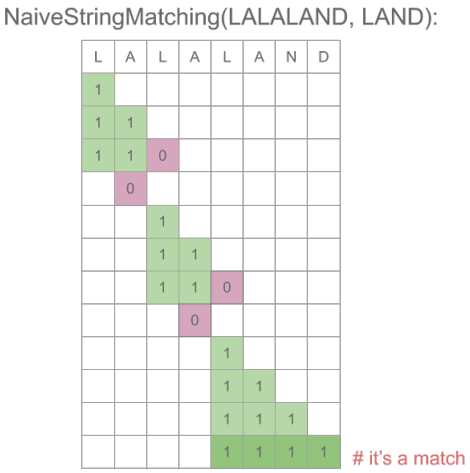

Figure 1: Illustration of how the naive brute force algorithm works

The Naive Brute Force Algorithm gets the work done, but the time complexity of this algorithm is $O(nm)$ *(m being the length of the searched word and n being the length of the text)*, the worst case scenario being the research of one character that is to be placed at the end of the sentence.

2.2 Rabin-Karp

The Rabin-Karp algorithm is in a certain way similar to the Naive algorithm, because it looks for all the possible combinations. Yet, this algorithm has two interesting attributes, a sliding window and a hash value function. The algorithm calculates a hash value for the given word to find and processes the whole body of text thanks to the sliding window that has the same length as the searched word. Having determined all combination of letters having the same length as the searched word, it then computes the hash value for each of these possibilities, the algorithm returns the index for the words that match the hash value of the input word [10].

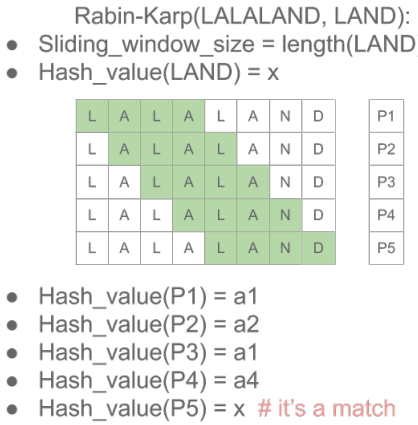

Figure 2: Illustration of how the Rabin-Karp algorithm works

One of the strengths and weaknesses of the Rabin-Karp algorithm is that we can get the position of many patterns. In other words, we can see in the example above that the "LALA" pattern appears twice in the input text which is "LALALAND". Nevertheless, as it analyses all of the possibilities the time complexity is yet again is $O(nm)$ *(m being the length of the searched word and n being the length of the text).*

2.3 Knuth-Morris-Pratt

The Knuth-Morris-Pratt (KMP) algorithm's performance is enhanced by the fact that it keeps an information on the given string, determining prefix of the word which will later, in case of a mismatch, give an information on where the next possible match can be. **[Explain here how the prefix values are calculated]** The KMP algorithm is, as a matter of fact, the first linear time string matching algorithm, but if there is no prefix the time complexity equals that of the naive brute force algorithm.

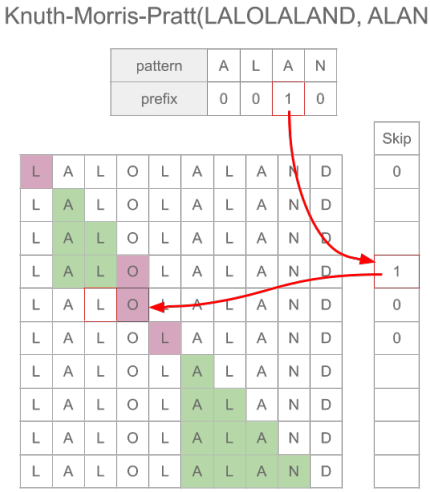

Figure 3: Illustration of how the Knuth-Morris-Pratt algorithm works

Given the construction of the failure function table, and therefore of the prefixes, this algorithm could be very interesting and useful in the fields of genomics and/or proteomics because they usually involve repetitive patterns.

2.4 Conclusion on string matching algorithms

Algorithm	Preprocessing time	Matching time
Naive	0	$O((n-m+1)m)$
Rabin-Karp	$\Theta(m)$	$O((n-m+1)m)$
Knuth-Morris-Pratt	$\Theta(m)$	$\Theta(n)$

Figure 4: A summary of string matching algorithms complexity [1]

In conclusion, we can see that pre-treating the words that are in the search stack, we can help minimize the processing time or as it is called in figure 4 the "matching time", but in the worst case scenario, the time complexity of these three algorithms is comparable.

3 Word segmentation

String matching is one part of the solution for the word segmentation problem, the other part being the dictionary where the actual words are being stocked. In the first section of this paper

we analyzed algorithms to browse a large text; in this section we will discuss different possible ways of using these algorithms when there is no input word to look for, which implies looking for them in another structure that is the word container. Therefore, we will also discuss of the need of a well constructed structure to stock the words.

3.1 Algorithms to browse the sentence

3.1.1 Recursion algorithm and backtracking - a naive approach

One of the simplest approaches would be, as in the naive algorithm, to check letter by letter if there is a match with a word stocked in a dictionary. As we would like the algorithm to return all possible cuts of a sentence, we test if each prefix of the sentence is included in the dictionary (Figure 5). If the prefix is included in the dictionary, recursion is used for the remaining part of the sentence. Then, we test the following prefix, which contains one more character, until we reach the end of the sentence. Once the end of the sentence is reached, the result is stored if the last prefix tested is in the dictionary. This approach is known as the **backtracking approach** [2].

The complexity in time of this algorithm can be high and is strongly related to the length of sentence. Indeed, given a sentence of size n, (n-1) tests to check if a word is in the dictionary are computed, and in the worst case, the recursion is called (n-1) times. Consequently, the complexity in time is exponential.

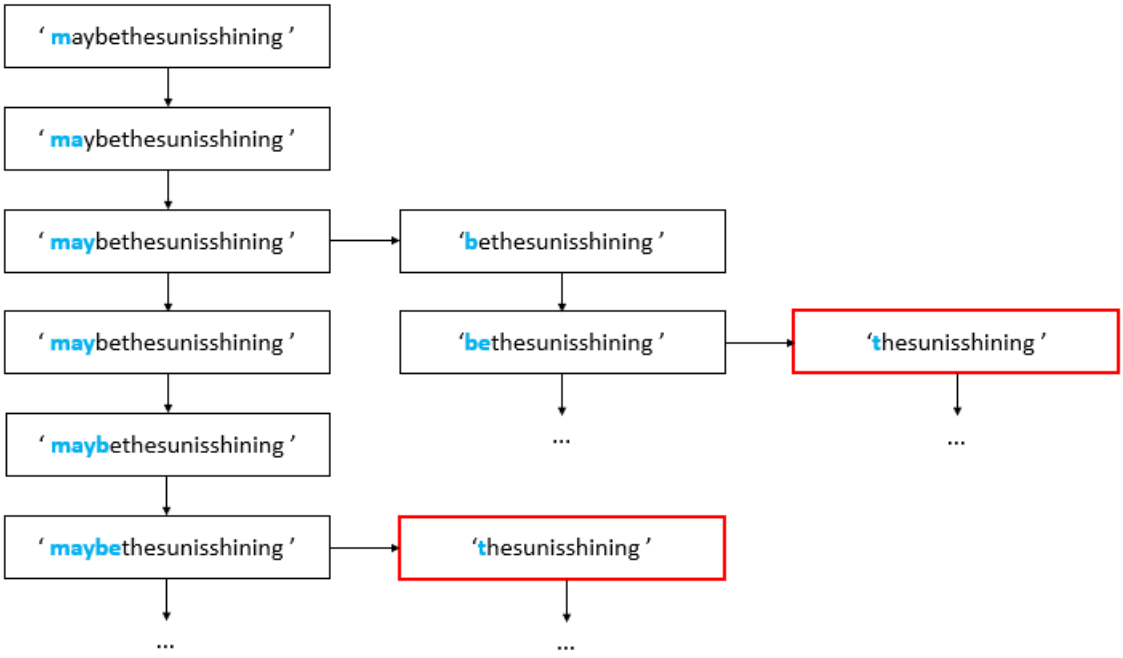

Figure 5: Graphic of the naive algorithm

3.1.2 Dynamic programming and pruning

An idea to reduce the complexity of the naive algorithm is to use dynamic programming [3][9], which consists in splitting a complex problem into several simpler sub-problems. The solution of the complex problem is then computed thanks to the intermediate results of the sub-problems.

Dynamic programming is interesting here as the previous algorithm contains several so-called "overlapping sub-problems"[3][9]. In our example (Figure 5), the algorithm processes for instance several times the same sub-sentence *'thesunisshining'*. The idea of dynamic programming is

to store the results of each sub-sentence, so that each sub-sentence is only tested completely one time. A subsentence is considered each time recursion is called. We'll detail the pratical implementation of dynamic programming in the section 4.

3.1.3 Sliding window approach

Using the sliding window approach and inspired by Rabin-Karp algorithm : an approach could be to build up a table or a dictionary that contains all possible words associated to a hash value. The algorithm would consist in passing a sliding window through a sentence and splitting the sentence according to the hash values. The aim is to go through the sentence multiple times, increasing the length of the sliding window at every round.

3.2 Word research in the dictionary : the necessity of a well constructed dictionary

3.2.1 The specificity of the Chinese dictionaries

Chinese orthography is different from English orthography because it is based on the construction of words using characters instead of using alphabet, therefore words could be considered shorter. Fluent Chinese requires the knowledge of 3,000 to 5,000 characters, and the number of existing characters can be counted in tens of thousands, it is estimated that it ranges from 40,000 to over 60,000. Chinese dictionaries consequently contain a huge amount of data therefore our approach has to be adapted.

3.2.2 Dictionary retrieval and description

Jieba is the most famous library in Python for Chinese text segmentation[6]. Two dictionaries are available in this library : the first one is smaller and adapted to common Chinese language, the second one is bigger and *"has better support for traditional Chinese"*[6]. We decided to use the small dictionary for this project. The main features of this dictionary are the followings:

- it contains 109 750 different words
- for each word, we have its character composition, its frequency and the semantic class it belongs to
- the dictionary is sorted according to the frequency of the words

3.2.3 Sorting and using dichotomy

A first approach, is to build a **sorted list** based on the content of the dictionary. There is no alphabetical order in Chinese, *"the sequence of head characters in a dictionary is based on either the graphic structure of the character or its phonetic properties."*[7]. However, as Chinese characters have a *Unicode* value, it is possible to use a lexicographical order to sort the dictionary in Python. Once the dictionary is sorted, the word research in the dictionary can be optimized using a binary search algorithm.

3.2.4 The prefix tree structure

Another approach is to structure the dictionary into a **prefix tree**, thi approch is also used in the Jieba library [6]. The dictionary contains all the single elements of a language as tree roots. Each node contains a word or part of a word, the nodes one step lower in the hierarchy contain a their parent's string and one element plus in the end. The node elements in the same hierarchy have the same length, the length is the range in the hierarchy. All the leaves must be words, but other nodes can be a word or part of word.

If we take English as an example, in the dictionary there would be 26 main branches whose node elements are *'a','b','c' ... 'z'*. In the main branch *'a'*, as there are no words beginning with *'aa'*, the main branch *'a'* has as children *'ab','ac','ad', ... , 'az'*, and node *'ab'* has children *'aba', 'abb',... , 'abz'*, and so on, until this branch contains all the words begin with *'a'*.

The longest word in Oxford English Dictionary is 'pneumonoultramicroscopicsilicovolcanoconiosis' which counts 45 letters. Therefore the main branch 'p' shall have 45 hierarchies. Chinese words consist of 1 to 7 characters, most of the words being 2 characters long. But, each character has its own meaning, in a way, so nearly every character along can be considered as a word. In the dictionary that we build, in the first hierarchy there are around **8000 main branches**, most main branches have depths of less than 4 hierarchies and most nodes contain a word.

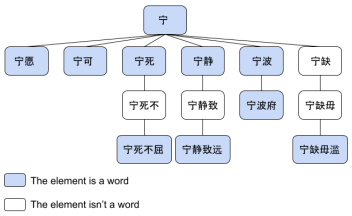

Figure 6: A main branch of Chinese prefix tree

3.2.5 The radix tree structure

A **radix tree** is a space-optimized prefix tree in which each node that is the only child is merged with its parent, this results in a **reduction of the overall depth**. This makes radix trees much more efficient for small sets of long strings and for sets of strings that share long prefixes. However, the node elements in the same hierarchy don't have the same length anymore.

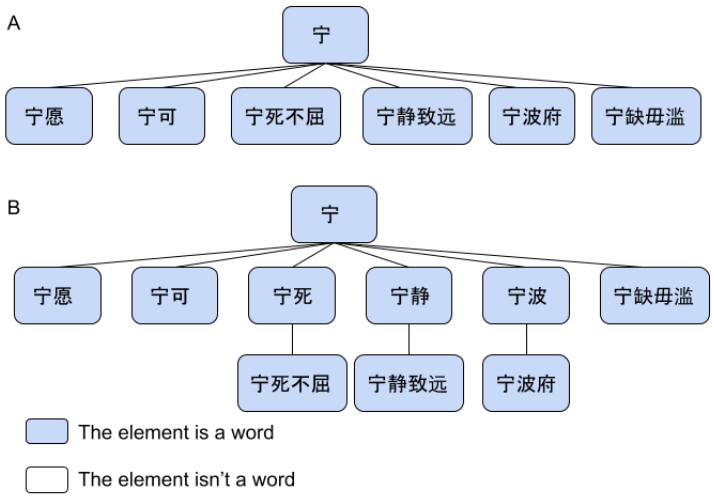

Figure 7: A main branch of Chinese radix tree

In Chinese it is often the case that a 2 character word is a part of 4 character word. From second hierarchy, most nodes have only one child. If we merge the node who is the only child with its parent, we end up loosing words, as shown in Figure7 A. Instead, we merge the node who is the only child and isn't a word with its parent, as shown in Figure 7 B.

Since Chinese words are short strings, the radix tree's advantage of reduction depth are not

efficient. Besides, the same length of elements in the same hierarchy will be useful in future word segmentation. We decided to build a prefix tree.

4 Our approach to Chinese word segmentation

4.1 First implementation : using recursion and backtracking with dichotomy

4.1.1 Implementation of the dictionary (script list_dictionary.py)

In this first approach, the words of the dictionary are **stored in a sorted list**. Two functions are used to compute the list : the first one loads the dictionary (*load_ dictionary*) and the second one sorts the dictionary (*quicksort*). We decided to use the **Quick Sort** to sort the dictionary, as this sorting algorithm is optimal on average. The complexity in time of this algorithm is *O(nlog(n))*, with n=109750.

In practice, the processor's time to sort the dictionary is 2.544e-05 s (average of 5 runs).

4.1.2 Description of the algorithm (script naive_algorithm.py)

The algorithm is composed of two functions:

- *contains(dictionary, word)* : this function is used to check if a word is in the dictionary and uses dichotomy.
- *cut_ 1(sentence, dictionary, branch=[ ])* : the heart of the algorithm. The cut currently explored by the algorithm is stored through the recursion thanks to the list *branch*. This list contains the different words discovered and is returned by the algorithm once the whole sentence has been tested successfully. The list *result* stores all the complete branches returned, which means all the possible cuts.

4.1.3 Performance

Theoretical performance In this case, let m be the size of the dictionary and n the size of the sentence to split.

First we will suppose that we are only looking for one of the many possible splittings, also, lets suppose the worst case scenario where every character of the sentence is a word, first we search, using a binary search algorithm, the first character of the sentence in the dictionary, and restart the process with the second character and so on. In this case we dive n times into the dictionary : the complexity of a binary search being of *O(log(m))* then in this case the complexity of the naive algorithm is *O(n log(m))*.

But, lets not forget that we want to get all the possible splittings. This means that eventually the whole sentence has to be tested as an only word. To illustrate this, lets have a look at the short word *"notes"* and its possible splittings, we see in figure 8 that many subsets of strings will be analyzed, in fact, the number of times a subset of a certain length will be analyzed is given by :

$$\sum_{i=1}^{n}(n-i+1)*i = \frac{n(n+1)(n+2)}{6}$$

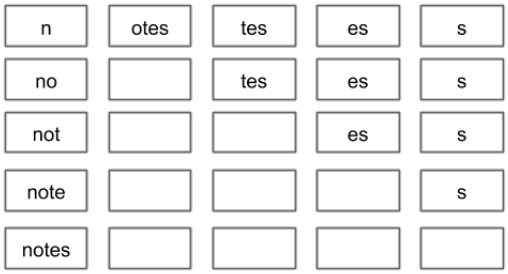

Figure 8: Countdown of all the splittings the naive algorithm will analyse

To conclude, the complexity of the naive search algorithm would then be $O(n^3 log(m))$. The algorithm has significantly better performance thanks to the binary search, if the search was not binary the complexity would be $O(n^3 m)$.

Practical performance In addition, we studied the performance of our algorithm for different sentence lengths (Figure 9). The processing time increases quickly over 30 characters, and the algorithm can not be used for sentences longer than 40 characters. We will try to improve the complexity in time of the algorithm in a second approach.

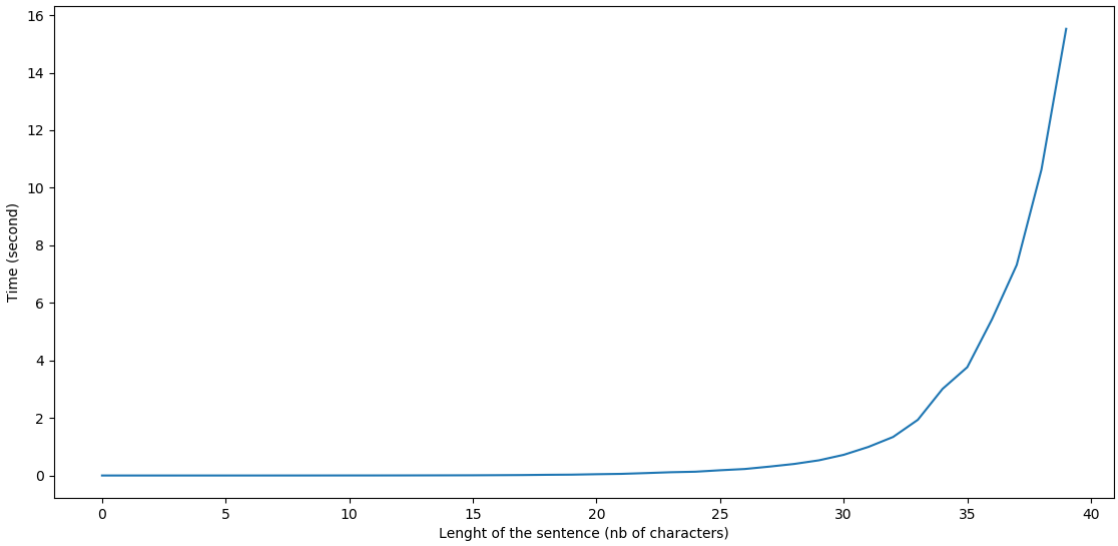

Figure 9: Average of the measured processor time for the algorithm to compute all the possible cuts of a sentence, for different sentence lengths (5 runs)

4.2 Second implementation : Word search guided by a well constructed tree, dynamic programming and pruning

4.2.1 Implementation of the dictionary (script tree_dictionary.py)

This method sets up a **prefix tree** with all words in a dictionary.
We define two classes: *Character* class and *Dict_tree* class. Each node of the prefix tree is a *Character* class object, with 3 attributes:

- *char* is a character or characters, the element of this node
- *is_word* is a Boolean value, which stores if the element of this node is a word or not

- *words* is a Python dictionary object, whose keys are the characters of its children, the value is the corresponding *Character* class object.

Elements in this class have two functions:

- *find_ branch(self, char)* returns to the *Character* class object corresponding to the given characters (*char*), if it's in the subbranch of the element (*self*);
- *add_ word(self, word)* add the given new word (*word*) to the branch of this element (*self*)

The root of prefix tree is a *Dict_ tree* class object. It has only one attribute: *character_ dict* is a Python dictionary object, whose keys are the character of its children (its children are the elements in the first hierarchy, have only a single character) , the value is the corresponding *Character* class object.
There are three functions in this class:

- *add_ character(self,characters)* adds the given new word (*characters*) to this tree (*self*)
- *find_ tree(self, char)* given a character string with only one character, return to its *Character* class object
- *find_ char(self, char)* given a character string with one or many characters, return to its *Character* class object.

In practice, the processing time to build the tree is 1.190 s (average of 5 runs), which is a little bit longer than the time to sort the dictionary in our previous approach.

4.2.2 Description of the algorithm (script tree_pruning_algorithm.py)

The actual Word Break Algorithm is composed of one function, described below :

- *cut_ all(dict,text,dict_ p=None,p=0,i=1)* is a recursive function that gives all possible segmentation from each possible cut position of the string.
  - *dict* is the prefix tree
  - *text* is an input string containing only Chinese characters. For each character in the string, we name the position *0 to l-1* (*l*: length of string), a sub string from position *p* is the string from position p to the end
  - *dict_p* is a Python dictionary, the keys are : position *p*, value : a list of all the possible segmentation from the position, empty as input, will store all segmentation and will be the return value
  - *p*: the starting position
  - *i*: the length of word, 1 as default
- For longer texts, the function *remove_ punctuation(string)* is used to split the sentences into several sub sentences, before calling *cut_ all* for each sub sentence. The simplest way to split a sentence into sub sentences is using punctuation. *In general, in Chinese, a sub sentences has less than 20 characters.*

Here we note a sub string from position ***p*** *with length* ***i*** *as* ***text(p,i)****.*

The algorithm implementation takes into account various conditions and processes in order to increase the performance of the splitting :

- **The stop condition:** When the sub string from position *p* is a word of length *i*, we add this word to the *dict_p[p]*.

- **Recursive Process for *p*:** if from a position p, a word with length i is found, recall the function from position $p+i$ and word length i.
- **Recursive Process for *i*:** After coming out of Recursive Process for p, recall the function from position p and word length *i+1*. Here we can see the advantage of using prefix tree, if *text(p,i+n+1)* is not a child of *text(p,i+n)*, Recursive Process for i can be end. We don't need to check *text(p,i+n+2)* and the longer strings.
- **Avoiding repetition:** If in the *dict_ p[p]*, the result of a position p is already exist, in the Recursive Process for p the function from position p wouldn't be recalled. Instead, the result of this function will be called and join with the new word just before position p.

4.2.3 Performance

Theoretical performance Let l be the length of the input text. If all positions are a possible positions to cut, the number of segmentation should be $2^{(l-1)}$, the number of segmentation is exponent of the length of input text. In reality there are the positions that must be cut and positions that can not be cut, the number of possible segmentation is in fact smaller than $2^{(l-1)}$. We suppose the number of possible segmentation is a function of *l*, *f(l)*, $f(l) < 2^{(l-1)}$.

Python dictionary is a hash table, a structure that can map keys to values. The complexity of find a value by given key is *O(1)*.

If the string of characters of this element is m, the string characters to find and the word to add has length n, The complexity of *find_ branch(self, char)* and *add_ word(self, word)* is *O(n-m)*, *add_ character(self,characters)* and *find_ char(self, char)* is *O(n)*. Since the length of Chinese words are one or two in most case and can't be bigger than seven, I will ignore the complexity of those functions when calculate the complexity of function *cut_ all(dict,text,dict_ p=None,p=0,i=1)*. The largest complexity of this function lies in join a new word before position p with all the possible segmentation of sub string from position p, *f(lp)* (*lp* is the length of sub string from position p). The complexity can be calculated as $a_{l-1}f(l-1) + a_{l-2}f(l-2) + ... + a_1 f(1)$,where a_i is the number of possible words just before position p, it can be *0*, if this position can't be cut from what's before it, in most case it should be less than four. The complexity is *O(f(l))* where $f(l) < 2^{(l-1)}$, which is $O(e^l)$.

Practical performance We studied the performance of this second algorithm for different sentence lengths and compared the results to the first algorithm (Figure 10). We successfully managed to decrease the complexity in time of the algorithm. However, the processor time is still very high over 35 to 40 characters.

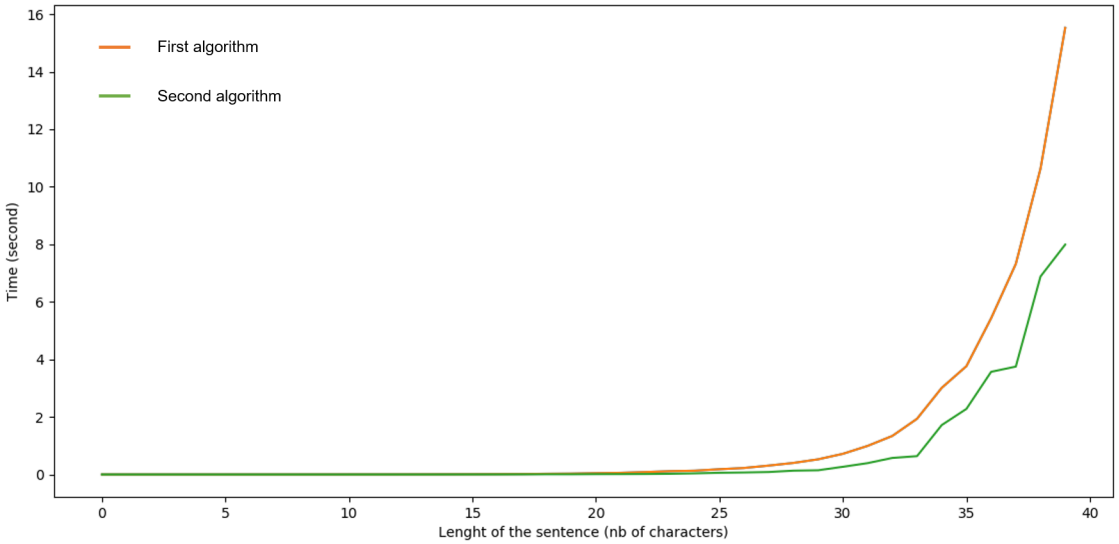

Figure 10: Average of the measured processor time for the two algorithms to compute all the possible cuts of a sentence for different sentence lengths (5 runs), orange : naive algorithm, green : search guided by a tree

The complexity in time is strongly related to the number of possible segmentation, as shown on Figure 11. With 30 characters there are already more than 10 000 possible segmentation, with 40 characters there are more than 200 000 possible segmentation.

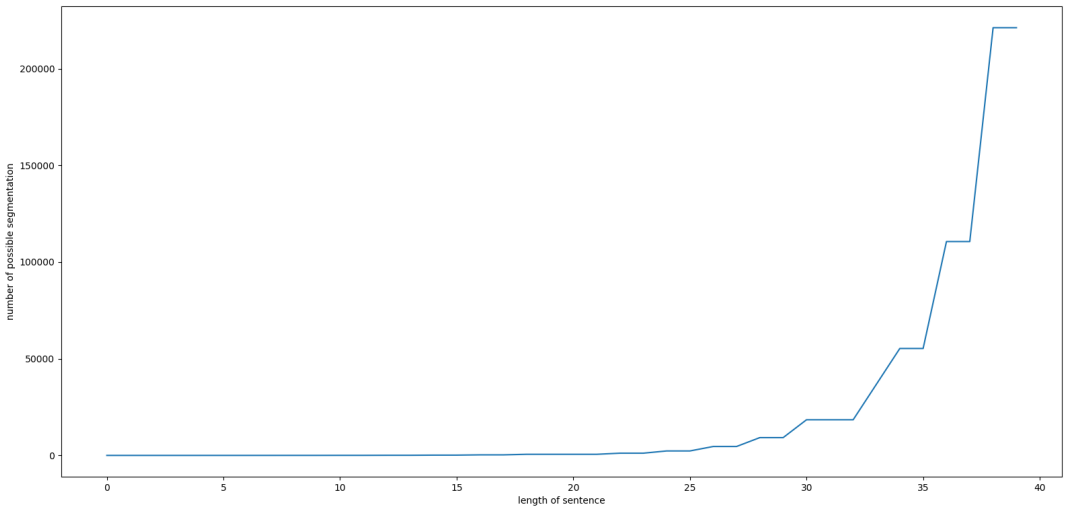

Figure 11: The relation between number of possible segmentation and length of input

4.3 A quick recap on the complexity of these two algorithms

Algorithm	Preprocessing time	Splitting time
Naive	*0*	*O(n^3m)*
Naive + binary search	*O(mlog(m))*	*O(n^3log(m))*
Pruning + prefix tree	*O(m) ?*	*O(exp(n)) ?*

Figure 12: A summary of the complexity of our word breaking algorithms

As for the string matching algorithms we can conclude that pre-treating the word container structure, having a well organized word container, can help minimize the processing time or as it is called in figure 12 the "splitting time". An interesting thing to see, is that when we use the prefix tree, the size of the dictionary is no longer relevant to the splitting time. In this case, the *jieba* dictionary is considered to be a small dictionary, so we can guess that using a larger dictionary would prove the pruning+prefix tree algorithm even faster than the naive algorithm that uses a binary search.

N.B : There might be a mistake in the calculation of the processing time to build a prefix tree, because in practice building a prefix tree takes longer than sorting the dictionary, theoretically 12 it is not the case (probably because of the miscalculation), the table 12 needs to be reviewed.

4.4 Going a step further : Searching the best cut in Chinese Word Segmentation

Finding all possible cuts of a sentence into words of a dictionary is the first step in language processing of Chinese texts. The second step is to find the best split, according to the meaning of the sentence.

This task is much more difficult, and many different approaches have been proposed in Chinese Word Segmentation. Some team of researches use grammatical relations to tackle the problem[4] [6], some others use word frequency [6]. Others use models *"learned from punctuation marks which are perfect word delimiters"* [8].
But the real goal is to be have

> *"a robust segmentation that requires no prior lexical knowledge and no extensive training to adapt to new types of data [5]"*

This is not an easy task in any of the known languages, Huang et al. [5] propose an approach where they first classify Chinese words into character-boundaries (CB's) and word-boundaries (WB's), given the distribution of CB's and WB's there is a possibility to obtain a robust segmentation.

This is an example among many of the research that has been going for years on proper Chinese word segmentation, yet no optimal solution requiring no lexical knowledge has been found. In the next paragraph we detail our simple approach that uses backtracking.

Our implementation : Backtrack word search (script best_cut.py) The idea is to start from the beginning of a text, cut at the position of longest word, and start again from next position, if in the following part, no solution can be found, go one step backwards, if there is a solution go forwards, if not keep going backwards.

Why cut by longest word? As we discussed in **Our approach to Chinese word segmentation**, most Chinese characters can be considered as a whole word, the number of possible segmentation is exponent of the length of input text. Therefore we try to find a rather good segmentation without having to check all the possibilities. As described in **the prefix tree structure**, the first hierarchy of the Chinese prefix tree has 8000 nodes, in the second hierarchy each branch has several nodes, and afterwards each node has in most cases only one child or no children. For a position, if it is not with the character before it, there are 8000 characters possible, if it is with the character before it, there are very few possibilities. So if the character in this position happens to be able to make a word with the character or characters before it, with great chance, that is the word we are looking for.

Go forwards: With given a text and a dictionary, from a certain position p of the text, the tree shall be the character of this position text[p], the idea is to find the longest word (with length of word i) *text(p,i)* in the tree, and then go to the next position p+i+1.

Go backwards: If from a position p, no word can be find, go to position *p-1*, the word before position p is text[p-l-1:p-1] its length $l > 1$, if text[p-l-1:p-2] is a word and a word can be found from *p-1*, finish Go backwards, Go forwards from *p-1*, if not keep Go backwards.

This algorithm works efficiently, but it gives only one solution of segmentation (a rather good one in my opinion).

5 Conclusion

In conclusion, we successfully managed to compute all possible cuts of a sentence written in Chinese. We improved our first approach, which used **backtracking, recursion and a list to structure the dictionary**, thanks to the **Prefix Tree** structure and **dynamic programming**. By doing so, we decreased the time complexity of the algorithm. However, because of the very high number of possible segmentation, both of our algorithms still need very long to deal with sentences longer than 40 characters. In practice, our algorithms could be used, as longer texts and sentences can be cut in smaller sub sentences thanks to punctuation. Indeed, in Chinese, a sub sentence between two punctuation has less than 20 characters.

Finally, as an extension of our work, we tried a first implementation of searching the best cut among all possibilities. Another possible extension of our project would have been to improve our algorithms so that they resist to noisy data.

References

[1] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. *Introduction to algorithms. 2nd ed.* Cambridge, MA: MIT Press, 2nd ed. edition, 2001. ISBN 0-262-03293-7/hbk; 0-262-53196-8/pbk.

[2] Unknown Geek for geeks. Word break problem using backtracking, consulted on february 2020. URL `https://www.geeksforgeeks.org/word-break-problem-using-backtracking/`.

[3] Unknown Geek for geeks. Word break problem | dp-32, consulted on february 2020. URL `https://www.geeksforgeeks.org/word-break-problem-dp-32/`.

[4] The Stanford Natural Language Processing Group. Chinese word segmentation, Unknown. URL `https://nlp.stanford.edu/projects/chinese-nlp.shtml#cws`.

[5] Chu-Ren Huang, Petr Šimon, Shu-Kai Hsieh, and Laurent Prévot. Rethinking chinese word segmentation: tokenization, character classification, or wordbreak identification. In *Proceedings of the 45th Annual Meeting of the Association for Computational Linguistics Companion Volume Proceedings of the Demo and Poster Sessions*, pages 69–72, 2007.

[6] Sun Junyi. jieba, Consulted in February 2020. URL `https://github.com/fxsjy/jieba;https://pypi.org/project/jieba/#description`.

[7] Henning Klöter. Chinese lexicography, published in 2013, consulted on february 2020. URL `https://www.academia.edu/20535422/Chinese_lexicography`.

[8] Zhongguo Li and Maosong Sun. Punctuation as implicit annotations for chinese word segmentation. *Computational Linguistics*, 35(4):505–512, 2009. URL `https://www.aclweb.org/anthology/J09-4006.pdf`.

[9] Sadanand Vishwas OpenGenus. Word break problem o(n * s) time complexity, consulted on february 2020. URL `https://iq.opengenus.org/word-break-problem/`.

[10] Rafid Asrar Prottoy, Munshi Allama Rumi, et al. *Study On String Matching Algorithm.* PhD thesis, United International University, 2018.