UMEÅ UNIVERSITET **Sunday 17th June, 2018**

Assignments

Efficient algorithms, 57209HT17

Final report

Name	Username	E-mail
Kristen Viguier	ens17kvr	ens17kvr@cs.umu.se

Graders
Frank Drewes

Introduction

In computing science we are often wondering about how to solve a problem by designing an algorithm in a way that is not too much complicated. But sometimes designing a simple algorithm is not enough, we need to optimize it in a way to maximize efficiency we wish and to minimize resource usage. This can be a tough exercise and will require to take time, think about the problem and acquire efficient algorithm skills.

The aim of this document is an introduction to algorithm efficiency. The first part is about the two ways of designing algorithms, Divide and Conquer and Dynamic Programming. The second part is an introduction to formal language theory. The third part is an experimental part where we implement the Chomsky Kasami Younger algorithm with Divide and Conquer and Dynamic Programming. Finally, the last part is about analysis and performance comparisons between the different algorithm designs.

Contents

1 Divide and Conquer and Dynamic Programming

We are now going to introduce the two widely used algorithms in efficient algorithm. These are Divide and Conquer and Dynamic Programming. Both are different but their aim still the same : divide a problem into sub-problems in order to solve efficiently an initial problem by combination of trivial cases.

Sub-problems : the problem can be broken down into sub-problems which are reused several times or a recursive algorithm for this problem solves the same sub-problem over and over.

The difference, between these two methods, that allows us to determine which one to used is the type of sub-problems. Indeed, if by dividing the initial problem into distinct sub-problems, not dependent and they are not overlapping in this case the method that has to be used is Divide and Conquer. However, if the division of an initial problem into sub-problems that are dependent of each others and are overlapping, in this case the method that has to be used is Dynamic Programming.

For instance, if we decide to solve dependent problems with the Divide and Conquer method then we will have an exponential time computation because we need to solve every overlapping sub-problems each time. This is why Dynamic Programming is appropriated because we store computed results, within an adapted data structure according the need, in order to enhance running time. Thank to that, when we are solving a sub-problem we check first whether it has been already computed. If so, we just return the stored value . If not, we solve this sub-problem and store the result within the data structure.

To sum it up, Dynamic Programming has to be preferred over Divide and Conquer if and only if there is no way to avoid overlapping sub-problems, but if you use Dynamic Programming for no overlapping sub-problems, then the data structure used to store computed value is useless overhead.

1.1 Divide and Conquer

Divide and Conquer is an algorithm which permits to resolve recursively a problem. The process is to divide our initial problem into smaller sub-problems. Then we resolve those recursively and combined the obtained results to an overall result. If the obtained sub-problems are trivial then we solve them in a straightforward manner.

This is an often used technic which provides efficient algorithms such as binary search, merge sort... It is very efficient, its complexity is somewhere between $O(n\log n)$ and $O(n\log n^3)$

It is quite easy to remember how Divide and Conquer works. We can easily remember its steps as divide, conquer, combine. For instance you can see below

Figure 1 how one step works :

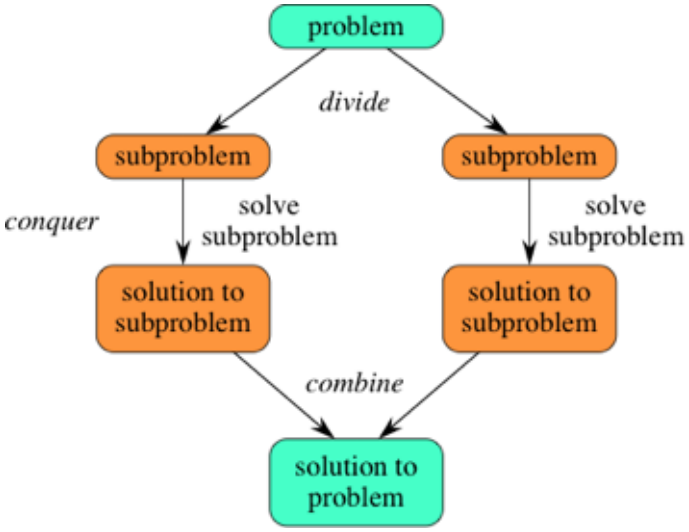

Figure 1: Divide and Conquer step process

1.2 Dynamic Programming

Dynamic Programming is quite similar to the Divide and Conquer because the process is the same. The difference is based on the fact of remembering the results, this is called **memoïzation** for Top-Down or **tabulation** for Bottom-up. This is due to overlapping problems to avoid repeated computations by the use of a table or a specific data structure (the choice of a data structure is important because this enhance the running time).

There are two ways of designing algorithm with Dynamic Programming :

- **Top-Down :** is very similar to the Divide and Conquer process because this is a recursive algorithm. The main difference is the used of memoïzation.
- **Bottom-up :** is different to the Divide and Conquer because it uses iteration to fill a table of results for all sub-problems starting from the smallest.

1.2.1 Top-Down

Our point of departure is a general problem that we divide into sub-problems and, moreover, their resolutions allow us to find a solution to this problem. Either we use memoïzation principle (it is an optimization technique used to speed up computer programs by storing results of expensive functions) or we store solutions inside a table. Thanks to this, each time we have to solve a

sub-problem we will check if we have already stored a solution. If so, then we can use it, otherwise we have to compute and add the solution to the table. Top-down is a recursive method.

Here a little example of Fibonacci number with Top-down pseudocode version. If you don't know anything about Fibonacci number I suggest you to search it on Wikipedia.

Algorithm 1: Fibonacci with Top-Down version

```
input: The parameter is the rank of the search term : n
begin
    initialize table with NIL values
    if table[n] == NIL then
        if n <= 1 then
            table[n] = n;
        else
            table[n] = fib(n-1) + fib(n-2)
        end
    end
    return table[n]
end
```

The difference with the naive algorithm of Fibonacci is that we check first inside the table if we have already computed the solutions. This algorithm first initializes the table of size n with NIL values. Then, when we are solving the Fibonacci problem, we look inside the table whether or not the value exist. If so we just return that value otherwise we compute it and store the value inside the table.

1.2.2 Bottom-up

Bottom-up is quite different to Top-Down because of its approach. We need to know every sub-problems that has to be solved in order to solve the initial problem. The Bottom-up version is represented by several layers where one layer represents a sub-problem to solved. We always start from the bottom and by solving every layers and combination with previous results we find our initial problem. We can deduct that this method is an iterative algorithm that is different to Top-down which is recursive.

Here a little example of Fibonacci number with Bottom-up pseudocode version.

Algorithm 2: Fibonacci with Bottom-Up version

```
input: The parameter is the rank of the search term : n
begin
    initialize table of size N. Set table[0] = 0 and table[1] = 1
    for i ← 2 to n do
        table[i] = fib(i-1) + fib(i-2)
    end
    return table[n]
end
```

As we said above, Bottom-up is an iterative algorithm. We can see that we build a table and we return the last cell within the table. Indeed, in this Fibonacci bottom-up design we compute Fibonacci 0, then Fibonacci 1, Fibonacci 2 ... until Fibonacci n.

2 Comparison of the methods

We can't discard one method rather than the other because the choice will depend on the kind of problems we have to solve. One of these methods will be more efficient, adapted or easier to implement depending on our need and by the kind of problems we have to solve. In the case of Dynamic Programming, problems are overlapping, are dependent whereas the Divide and Conquer method solves fully independent problems. This means that, in the case of dynamic programming, the resolution of a problem will depend on the solution or solutions of sub-problems. Another interesting difference is that we store sub-problems' solutions in order to avoid to compute again a solved sub-problem. This saves time at the expense of the memory. It is not something we do for the Divide and Conquer method where we compute each time the solution of a sub-problem.

Top-Down :

- the reasoning for finding recurrence is generally easier if the problem is not to much complex,
- recursive method is faster than iteration method for smaller problems,
- recursive method avoids unnecessary calls. Hence the solving of the problem is faster than iteration way because we don't compute all the sub-problems,
- as it is recursive if the input is too large we can have a stack buffer overflow,
- it consumes more memory because of recursivity,
- it is not easy to debug mostly determining where the solution went wrong is complex.

Bottom-Up :

- implementation is more difficult than Top-Down because we need to determine every sub-problems for every layer and how to store results in table and combined them to solve the initial problem,
- implementing optimization and parallelization is easier,
- bugs can be easily found,
- allows us a certain freedom in the way to create variables, codes and choice of data structure,
- big problems are less time consuming than Top-down and this method avoid stack buffer overflow.

Personally, I prefer the Top-Down method because we start with an initial problem then we divide it into sub-problems. This reasoning seems to me more natural, easier to visualize, also this is simpler to represent it, for example, with a tree representation. But as it was said before, the choice of the method will depend on which kind of problems we have and what is our need.

3 Grammars and CKY Algorithm

In this part, we are going to put into practice the previous algorithm methods seen before through the Cocke Younger Kasami algorithm which is a parsing algorithm for context-free grammars. But before the practical part, we are going to define several notions about formal language theory in order to have the requirement to understand the CYK algorithm.

3.1 Formal language theory

Formal language theory studies all the purely syntactical aspects of such languages, that is their internal structural patterns. In computer science, formal language are often used as the basis for defining programming languages and others systems. Words of a language have a meaning and a semantic. We are going to see a small part of this area because the aim is to know necessary concepts for the understanding and implementation of the CYK algorithm.

Definition : In mathematics and computer science, a formal language is a set of words and it is a subset of an alphabet. It is often defined by a grammar.

A **word** is a finite sequence of elements of an alphabet. It has a length k w={a1, a2, ..., ak}. ϵ represents an empty word w. It is made up of letters.

A word ω=**(a1,a2,...,an)** can be written in a simple way like : ω=**a1a2...an**

We can concatenate 2 words. Let say we have u and v, 2 words, $u = abraca$ and $v = dabra$ the concatenation gives the word $uv = abracadabra$

3.1.1 Production rule

We call a production rule (rewrite rule) a couple **S**=<**V**,**R**>. **V** represents the vocabulary and **R** represents a finite set of string pairs on **V***.

Every element (α, β) of **R** is called a rewrite rule and is under the form $\alpha \rightarrow \beta$. If we have rules that are often repeated like this :

$\alpha \rightarrow \beta$**1**, $\alpha \rightarrow \beta$**2**, ..., $\alpha \rightarrow \beta$**n**

then we can write them like this $\alpha \rightarrow \beta$**1** | β**2** | β**n**

3.1.2 Grammar

We call grammar every quadruple **G=(N, T, P, S)** :

- **T** and **N** are 2 disjoint vocabularies. **T** represents terminal vocabulary (its elements are called **terminal symboles**). **N** represents non terminal vocabulary (its elements are called **non-terminal symbols**). **V**=**T**∪**N** is the vocabulary of the grammar.
- **S** is an element of **N**, this is the start symbol of the grammar.
- **P** is the rule set of the grammar which are pairs formed of a non-terminal and a sequence of terminals and non-terminals..

3.1.3 Generated language

The generated language of a grammar **G**, written **L(G)**, is the set of terminals that we can derived from the start symbol **S**.

$L(G) = \{w \in T^* | S \Rightarrow^* w\}$

3.1.4 Context-free grammar

Context-free grammar is a certain type of formal grammar in which each production rules is under this form : $\mathbf{X} \rightarrow \alpha$

X represents a non-terminal symbol and α is a string composed of terminals and non-terminals.

3.1.5 Chomsky normal form

A context-free grammar is under a Chomsky normal form if all its rules are under one of the following forms :

- **X** $\rightarrow$ **AB** where **X, A, B** are non-terminals.
- **X** $\rightarrow$ **a** where **X** is a non-terminal and **a** is a terminal.

Theorem : Let **G**, a free-context grammar, can build a grammar **G'** under Chomsky natural form such as : **L(G')** = **L(G)** - $\{\epsilon\}$

3.2 Cocke–Younger–Kasami Algorithm

Cocke–Younger–Kasami algorithm is a parsing algorithm for context-free grammars. It allows to determine if a word is generated by a grammar. It is named like this because of its inventors, John Cocke, Daniel Younger and Tadao Kasami. It supports only grammar under the Chomsky Natural Form. However, we can transform any context-free grammar into the CNF. This algorithm employs bottom-up parsing and dynamic programming.

The first part will introduce the Bottom-up version followed with an example to understand exactly how it works. Then, we will see the naive and the top-down pseudocode.

3.2.1 Bottom-up method with memoïzation

In the bottom-up we need to know the length of the input string **n** and the size of the grammar **r**. The first step consists of initializing all the elements of the table[n][n][r]. The second step is divided into 2 parts. The first part is about the first layer which works with a word of size 1 to determine terminal symbols. This means we have to find a rule that matches with the word size of 1 and if so we store it. The second part will increase each time the length of the sub-string. We will treat a word of size 2, then 3, then 4... until the size n of the word.

Finally, we check the last cell of the table in order to know whether the word belongs to this grammar.

Algorithm 3: Bottom-Up version of CKY Algorithm

```
begin
    Create a table[n][n][r] where n represents the length of the input
    string and r the grammar size.
    for i=1 to n do
        for j=1 to n do
            for k=1 to r do
                table[i][j][k] = false;
            end
        end
    end
    for i=0 to n do
        for all rules Na → input[i][i] do
            table[i][i][Na] = true;
        end
    end
    for l=2 to n do
        for i=1 to n - l + 1 do
            for k=i to i + l - 2 do
                for all production Na→ Nb Nc do
                    if table[i][k][Nb] and table[k + 1][i + l - 1][Nc] then
                        table[i][i + l - 1][Na] = true;
                    end
                end
            end
        end
    end
    return table[n]
end
```

3.2.2 Example

This part will present the CKY algorithm step by step, from the bottom up method, for a given input and grammar. For this example let's say we have defined our grammar :

- **S** → **AB** | **BC**
- **A** → **BA** | **a**
- **B** → **CC** | **b**
- **C** → **AB** | **a**

The input word is : **baaba** with 5 length.

First layer : The first layer works with a word of length 1. So, it will be **b**, then **a**, then **a**... We have to check in the first layer part if we can find a non terminal

for each letter of the word.

For example, we are looking for X1,1 non-terminal rules. This corresponds to the letter **b**. By looking at the grammar we can find B as unique rule for the letter **b**. So, we store B in X1,1

For X2,1, this corresponds to the letter **a**. By looking at the grammar we can find A and C as rule for the letter **a**. So, we store A,C in X2,1

We repeat this process until the first layer is completed.

j i	1	2	3	4	5
5	?	X	X	X	X
4	?	?	X	X	X
3	?	?	?	X	X
2	?	?	?	?	X
1	**B**	**A,C**	**A,C**	**B**	**A,C**
input	b	a	a	b	a

Second layer : The second layer consists to increase the length of the sub string to take from the word. The sub string length will be 2, then 3, then 4 until the length n of the input word. Doing that we check every combination possible with the rule grammar to know whether we can find a non-terminal rules.

For example, with a sub string of length 2, we take the word **ba**. For the letter **b** we stored **B** in X1,1. For the letter **a** we stored **A,C** in X2,1.

X2,1 $= \{B\} \cup \{A, C\} =$ **BA, BC** the results of the combination. Now we check in our grammar whether a non terminal rule gives BA or BC. We find **A,S**. So, we store in **X2,1** $\leftarrow$ **A,S**.

The next sub string of the word is **aa**.

X2,2 $= \{A, C\} \cup \{A, C\} =$ **AA, AC, CA, CC** the results of the combination. Now we check in our grammar whether a non terminal rule gives AA, AC, CA or CC. We find **X2,2** $\leftarrow$ **B**

The next sub string of the word is **ab**.

X2,3 $= \{A, C\} \cup \{B\} =$ **AB, CB** the results of the combination. Now we check in our grammar whether a non terminal rule gives AB or CB. We find **X2,3** $\leftarrow$ **S,C**

The next sub string of the word is **ba**.

X2,4 $= \{B\} \cup \{A, C\} =$ **BA, BC** the results of the combination. Now we check in our grammar whether a non terminal rule gives BA or BC. We find **X2,4** $\leftarrow$ **A,S**

j i	1	2	3	4	5
5	?	X	X	X	X
4	?	?	X	X	X
3	?	?	?	X	X
2	**A,S**	**B**	**S,C**	**A,S**	X
1	B	A,C	A,C	B	A,C
input	b	a	a	b	a

Now we did all the sub string of length 2, we need to do the same process for sub string of length 3.

The next sub string of the word is **baa**.

X3,1 $= \{A,S\}\{A,C\} \cup \{B\}\{B\} =$ **BB, AA, AC, SA, SC** the results of the combination. Now we check in our grammar whether a non terminal rule gives BB, AA, AC, SA or SC. We find no rule **X3,1** $\leftarrow \phi$.

The next sub string of the word is **aab**.

X3,2 $= \{A,C\}\{S,C\} \cup \{B\}\{B\} =$ **AS, AC, CS, CC, BB** the results of the combination. Now we check in our grammar whether a non terminal rule gives AS, AC, CS, CC or BB. We find **X3,2** $\leftarrow$ **B**.

The next sub string of the word is **aba**.

X3,3 $= \{A,C\}\{A,S\} \cup \{S,C\}\{A,C\} =$ **AA, AS, CA, CS, SA, SC, CA, CC** the results of the combination. Now we check in our grammar whether a non terminal rule gives AA, AS, CA, CS, SA, SC, CA or CC. We find **X3,3** $\leftarrow$ **B**.

j i	1	2	3	4	5
5	?	X	X	X	X
4	?	?	X	X	X
3	ϕ	**B**	**B**	X	X
2	A,S	B	S,C	A,S	X
1	B	A,C	A,C	B	A,C
input	b	a	a	b	a

Now we did all the sub string of length 3, we need to do the same process for sub string of length 4.

The next sub string of the word is **baab**.

X4,1 $= \{B\}\{B\} \cup \{A,S\}\{S,C\} \cup \{\phi\}\{B\} =$ **BB, AS, AC, SS, SC,** ϕ the results of the combination. Now we check in our grammar whether a non terminal rule gives BB, AS, AC, SS, SC or ϕ. We find no rule **X4,1** $\leftarrow \phi$.

The next sub string of the word is **aaba**.

X4,2 $= \{A,C\}\{B\} \cup \{B\}\{A,S\} \cup \{B\}\{A,C\} =$ **AB, CB, BA, BS, BA, BC** the results of the combination. Now we check in our grammar whether a non terminal rule gives AB, CB, BA, BS, BA or BC. We find **X4,2** $\leftarrow$ **S, A, C**.

j i	1	2	3	4	5
5	?	X	X	X	X
4	**-**	**S, A, C**	X	X	X
3	ϕ	B	B	X	X
2	A,S	B	S,C	A,S	X
1	B	A,C	A,C	B	A,C
input	b	a	a	b	a

Repeat the same process as before for the last case. If we have the last symbol in **X5,1** it means that the word belongs to this grammar. In our case the input **baaba** belongs to this grammar.

j i	1	2	3	4	5
5	**S,A,C**	X	X	X	X
4	-	**S, A, C**	X	X	X
3	ϕ	B	B	X	X
2	A,S	B	S,C	A,S	X
1	B	A,C	A,C	B	A,C
input	b	a	a	b	a

3.2.3 Naive method

This is the CKY algorithm under the naive method that is equivalent to the Divide and Conquer process. Our trivial case is when $i==j$ and this case has to determine whether the input[i][j] is a rule.

The second part of the algorithm consists to browse every rule $X \rightarrow AC$ to check whether A or C matches with all the partitions of the word. To do this, we call recursively the function for the non terminal A and the non terminal B. If our two recursive functions return true for both A and B that means X is a rule for the partitions of the word (input[i][j]). Then we can say that X $\rightarrow input[i][j]$.

In the case that the two recursive functions returned false that means we did not find any rule that matches for any partition.

Algorithm 4: The Naive version of CKY Algorithm

```
input: A represents the non terminal, i the start index, j the length index
begin
    if i==j then
        return isRule(A, i);
    end
    for k = i to j - 1 do
        for all the rule Na → Nb Nc do
            if naive(Nb, i, k) and naive(Nc, k + 1, j) then
                return true;
            end
        end
    end
    return false
end
begin
    Function : isRule(A, i)
    for all rules A -> a do
        if A == a then
            if a == input[i] then
                return true
            end
        end
    end
    return false
end
```

3.2.4 Top-down method with memoïzation

This is the CKY algorithm in top down version with memoïzation. We use a table to store the computed results. We check whether the result has been already computed inside the table. If not, we compute it and then store it.

The first step is to check whether we have the result inside the table. If so, return the computed result. The second step : if(i==j) this is the trivial case we check whether it is a rule and store the computed result. The third step : we browse every partition and every rule and we call recursively the topDown function.

Algorithm 5: The Top-Down version of *CKY* Algorithm

```
input: A represents the non terminal, i the start index, j the length index
begin
    tmp = table[i][j][A];
    if tmp != null then
        return tmp;
    end
    if i==j then
        isRule = isRule(A, i);
        table[i][j][A] = isRule;
        return isRule;
    end
    for k = i to j - 1 do
        for all the rule Na → Nb Nc do
            if TopDown(Nb, i, k) and TopDown(Nc, k + 1, j) then
                table[i][j][A] = true
                return true;
            end
        end
    end
    table[i][j][A] = false
    return false
end
begin
    Function : isRule(A, i)
    for all rules A -> a do
        if A == a then
            if a == input[i] then
                return true
            end
        end
    end
    return false
end
```

4 Performance Metrics

In computer science, we evaluate the complexity of an algorithm in order to know the running time. It will quantify the amount of time taken by an algorithm to solve a problem. The complexity of an algorithm is commonly expressed using big **O** notation which excludes coefficient and lower order terms. In this section we are going to study the theoretical complexity and then we are going to do some practical measurements for the three methods seen previously.

4.1 Theoretical Efficiency

4.1.1 Bottom-up

We have a word length **n** and a grammar G length **N**. The first set nested loops of the algorithm is $\mathbf{O(n)}$ because we iterate to **n** once. The second set nested loops of the algorithm is $\mathbf{O(n^3)}$.

```
for each s = 1 to n
...
for each l = 2 to n -- Length of span
  for each s = 1 to n-l+1 -- Start of span
    for each p = 1 to l-1 -- Partition of span
```

A way more mathematical :

$$t = \sum_{l=2}^{n} \sum_{s=1}^{n-l+1} \sum_{p=2}^{l-1} 1$$

Finally, the complexity of CYK algorithm is $\mathbf{O(n^3N)}$, **n** represents the length of a word to parse and **N** represents the size of the grammar.

4.1.2 Naive

This algorithm works but it is not the most efficient because of a lot of redundant calls. Suppose **n** is the length of the input string and we assume that the algorithm is called with a string of length n (for any non-terminal for which there is at least one non-terminal rule). Then the algorithm checks, in its main loop, recursive calls in which the argument strings are of length :

- 1 and n-1
- 2 and n-2
- ...
- n-1 and 1.

We are going to establish a lower bound. We can notice that the two calls whose argument strings are of length n-1. They prove that the running time f(n) satisfies the inequality $f(n) \geq 2 * f(n-1)$. Thus, the function defined by $f(n) = 2 * f(n-1)$ and $f(1) = 1$ gives us a lower bound on the worst-case running time.

The name of the function f is exponential. Indeed, $f(n) = 2 * f(n-1)$ is the same as f(n) = $e^{(n-1)*ln(2)}$.

4.1.3 Top-Down

As explained before, the Top-Down version uses memoïzation to enhance running time. As we store the computation's result we avoid to compute it again and we avoid some redundant calls so we earn some times. According to me this algorithm running time is $\mathbf{O(n^3N)}$ where $\mathbf{n}$ is the length of the input string, $\mathbf{N}$ is the number of non-terminals in our grammar and for each production tried we have $\mathbf{O(n)}$ work.

4.2 Practical

In this part we will continue on efficiency measures but at the practical level. Having implemented the algorithm of CKY under the three previously seen methods (bottom-up, top-down with and without memoïzation) we will perform performance measurements based on different criteria. Obviously several aspects are to be taken into consideration (operating system, programming language, processor, compiler, data structure ...) because they can have an impact on the evaluation time and consequently lead to a more significant difference on the theoretical evaluation of the complexity of these three methods. We will first see how I implemented these three algorithms and then realized measurements and arguments on the different results.

4.2.1 Implementation

These three algorithms were implemented with Java because I am familiar with this language. The input grammar is stored inside a text file with the following conventions :

- Non terminals and terminals are individual ASCII symbols.
- All non terminals are uppercase letters of the Latin alphabet, and all remaining ASCII symbols occurring in the rules are terminals.
- The left-hand side of the first rule is the initial non terminal.

So, each specific grammar is inside a unique file. The word to analyze is set inside the Java program. For maybe a future enhancement we can either store the word to analyze inside the grammar that we want to test with or ask the user from a console which word he wants to test (like a menu).

My development environment was Eclipse Neon and every measurements were done with the same computer.

Representation of rules in memory : I have created an object called **Grammar** that contains several characteristics. My reasoning is : Grammar contains a start symbol and a set of productions rules then when I parse a grammar input

file I need to store the start symbol corresponding to this grammar and to store the production rules.

But what about the production rules ?

We know production rules can be either under the form $S \rightarrow AC$ or $C \rightarrow c$. We can see that we have two parts : the left part (non terminal) and the right part (either non terminals or terminal). According to me, the good data structure that represents this is a Map under this form : *Map< Non Terminal, Set<Right Part> >*. The key represents the left part (non terminal) and the value represents the set of terminals and non terminals for this given left part.

Thanks to this data structure it is quite easy to access quickly every rule having a specific left part.

Storing computed results : my choice of data structure to store the computed results was for bottom-up a boolean table of three dimensions and for the top-down it was a Boolean table of three dimensions.

The primitive boolean is by default to false and we have only 2 states either true or false. We know for the bottom-up we have to fill every cell either to true or false.

Unlike bottom-up, top down required 3 states : true, false or null and Boolean object allows this. This allows us to know whether the result was already computed or not.

We could have very well used int instead of boolean or Boolean with something like 0 for false 1 for true and -1 for not yet computed. But to me, it was more significant to use boolean.

4.2.2 Impact of the input size n

If we increase the size of the word to test then the number of iterations will increase for bottom-up and the number of calls will also increase of top-down. The number of iterations or calls are related with the running time because we will have more cell to compute.

As we mentioned before we have a table[n][n][r] where **n** represents the input size and **r** represents the grammar size. For instance, if we take a word of size 5 we will have a table[5][5][r] and if we increase that word size to 30 we will have a table[30][30][r]. Then, the number of cells to compute increase and the number of calls increase too according the specificity of each method. Thus this impacts the running time.

More specifically for the bottom-up, as we explained before the number of iterations is not "dependant" of the word. Why ? Because with this version we need to compute every cell of the table whether or not the word belongs to the language. Let's provide a example to illustrate this with the language 3. If we take a word of size 30 the number of iterations will be 35960 whether or not the

word belongs to the grammar.

Regarding the top-down approach, the running time will depend on the grammar but also the word. Let's take the same example above to illustrate the specific cases of top-down. If we take a word of size 30 the number of calls can be either 144 or 16301 whether the word belongs to the language.

We can notice that top-down calls are always lower (for best and worst case) than the bottom-up's number of iterations.

4.2.3 Naive approach

As we illustrated it before we can guess that this method is not effective. Indeed, we can easily guess that the execution time will increase if the size n of the input increases and will become very quickly too long.

Example with language 1 :

Word size	4	10	14	20	> 20
Time	3 ms	13 ms	160 ms	5334 ms	> 20 min

4.2.4 Measurements

We are going to do some running time comparisons between Bottom-up and Top-down approach with different languages. Every run test will be done with considerably longer input strings and equal spacing between the lengths.

Language 2

Let's do some measurements with the language 2 which is a linear grammar. We can notice that this language has a number of non-terminals and of rules more important than the others. We are going to see the behavior of both methods according to several and different input cases.

The grammar is defined as below :

- R $\rightarrow$ SP
- P $\rightarrow$ BP | DN
- N $\rightarrow$ BN | EX
- X $\rightarrow$ SY | S
- Y $\rightarrow$ BY | 0 | 1
- S $\rightarrow$ + | -
- B $\rightarrow$ 0 | 1

- D → .
- E → e

The first test is based on valid input (see Figure 2). That means they belong to the grammar. We noticed that Bottom-up method runs in cubic time while Top-down runs in linear time.

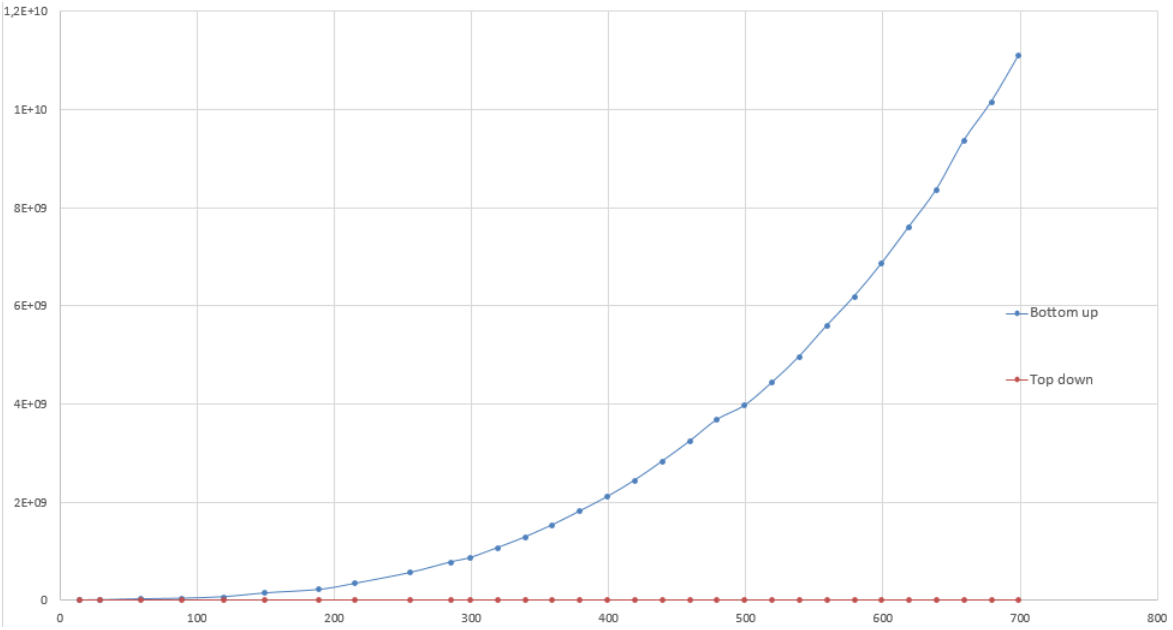

Figure 2: Comparison between Bottom-up and Top-down with valid input

NB : the curve of the top down is not 0. As the bottom-up running time is much higher it seems that top-down is 0 on the schema. I will show a schema that represents different cases between top-down to see what really happens.

The second test is based on no valid inputs. I added an incorrect character at the beginning of the inputs to see different behaviors. Here is what happened on the Figure 3.

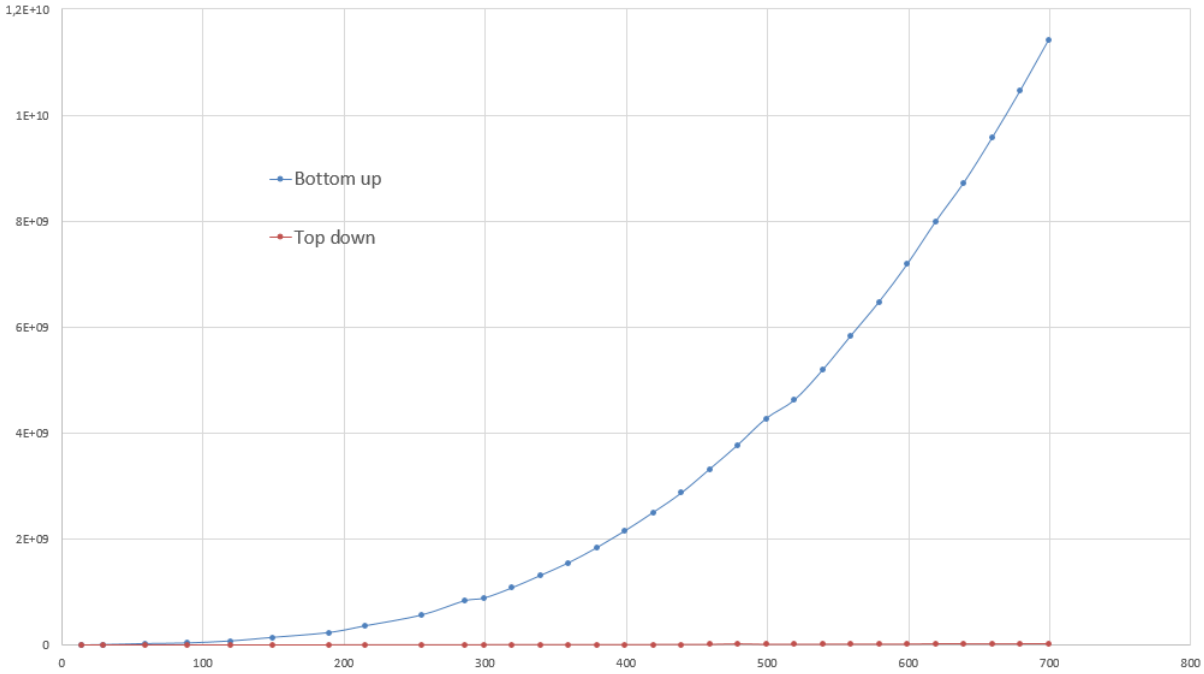

Figure 3: Comparison between Bottom-up and Top-down with no valid inputs - error at the beginning of the inputs

We notice that the Bottom-up running time still is cubic. Regarding the running of the Top-down it is quite low : quadratic.

The third test is also based on no valid inputs. I added an incorrect character at the end of the inputs to see different behavior. Here is what happened on the Figure 4

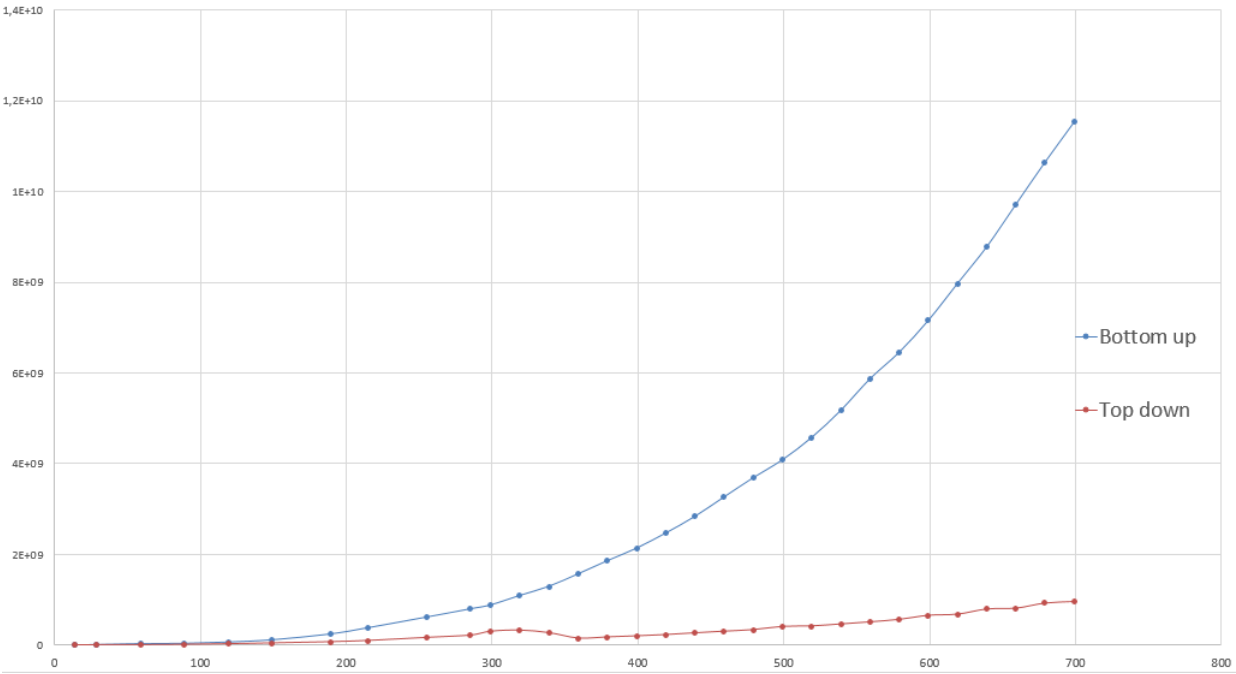

Figure 4: Comparison between Bottom-up and Top-down with no valid inputs - error at the end of the inputs

We notice that the Bottom-up running time still is cubic. Regarding the running of the Top-down it is quite low : cubic.

The Figure 5 is a comparison between Top-down to see the three curves guessed before with the same tests.

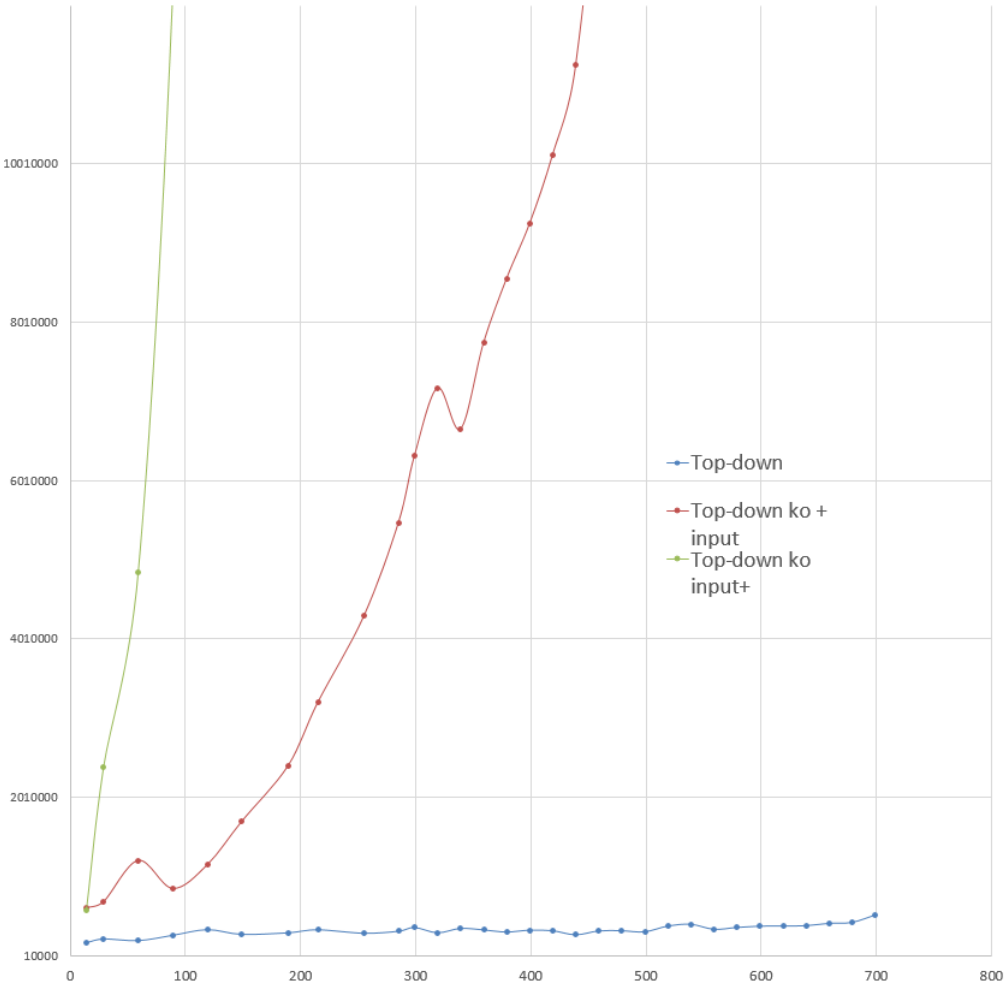

Figure 5: Comparison between Top-down valid inputs, error at the beginning of the inputs and error at the end of the inputs

To sum it up and give explanations

We can notice that every word are formed in the same way. For instance, let's take this input : *+010101010101.10101e-10101010*. We already know that this word belongs to the grammar by reading from the left to the right. We notice that it begins with the symbol *+*, then some *0* or *1* then a dot then others *0* or *1* then the exponent *e*, then a sign - then *0* or *1*. All the words are structured, formed in the same way so we can easily detect whether the word belongs to the language.

Because of that and the fact that Top-down approach calls itself recursively and avoids unnecessary calls this explains different running time behaviors.

Let's take two inputs that don't belong to the grammar : *+-010101010101.10101e-10101010* and *+010101010101.10101e-10101010-*

As we know, the Bottom-up version proceeds layer by layer. This means we compute every cell even the word belongs or not to the grammar. This means either the word belongs or not to the grammar, we know that the running time will be quite the same. Here in every case the running time will be cubic.

Regarding Top-down approach, we notice that the running time will depend on the fact that the word belongs to the grammar or not and also on the fact where the error is located.

As explained previously the Top-down approach will call itself recursively on partitions with a partition index going from the left to the right (remember about sub string). Moreover we do two recursive calls. The first call will detect very fast the first input error at the beginning of the word whereas the second call will detect the error at the end of the input in a bit longer time.

We also explained that Top-down version fills much less cells that the bottom-up so it spends less time in the main part of the algorithm. It avoids unnecessary calls. We can except the Top-down to be more efficient than the Bottom-up thank to the structure of the grammar and the specificity of Top-down approach.

Language 1

Let's look at the language 1. The grammar is defined as below :

- S → SS | LA | LR
- A → SR
- L → (
- R →)

It seems that this language is quite similar to the language 2. The word is always read from the left to the right and if we detect an unexpected additional symbol *"("* or *")"* than we know that the input doesn't belong to the grammar.

If the input begins by the symbol *")"* the Top-down approach will detect immediately that this input doesn't belong to the grammar whereas the Bottom-up will need to compute every cell.

Running time measurements :

- A word of size 100 that belongs to the grammar will take 59 ms for the Top-down and 63 ms for the Bottom-up.
- A word of size 100 that doesn't belong to the grammar and begins by *")"* will take 7 ms for the Top-down and 56 ms for the Bottom-up.
- A word of size 100 that doesn't belong to the grammar and ends by an unexpected additional parenthesis will take 12 ms for the Top-down and 61 ms for the Bottom-up

We notice, as the language 2, that the Top-down approach has a huge gain of running time when there is an error. Regarding the Bottom-up there is a little gain but it is not significant.

Running time according to the location of the wrong character

We see that languages can have some specific effects on the Top-down approach. I am going to do some measurements with the language 1 about the impact that a wrong character can have on the running time.

For this measurements I will create an input of size 100 and put a wrong character at the beginning and shift it until the end. I will add 50 loops on the same error index in order to have an average running time express in ms. Results appear on Figure 6.

We notice that the first half of the input's running time tends to grow until to reach the half of the input. After that the running time becomes more constant and takes the same amount of time. This is due to the fact that when an error is after the half of the word the Top-down approach has to analyze the first half part before detecting it in the second half.

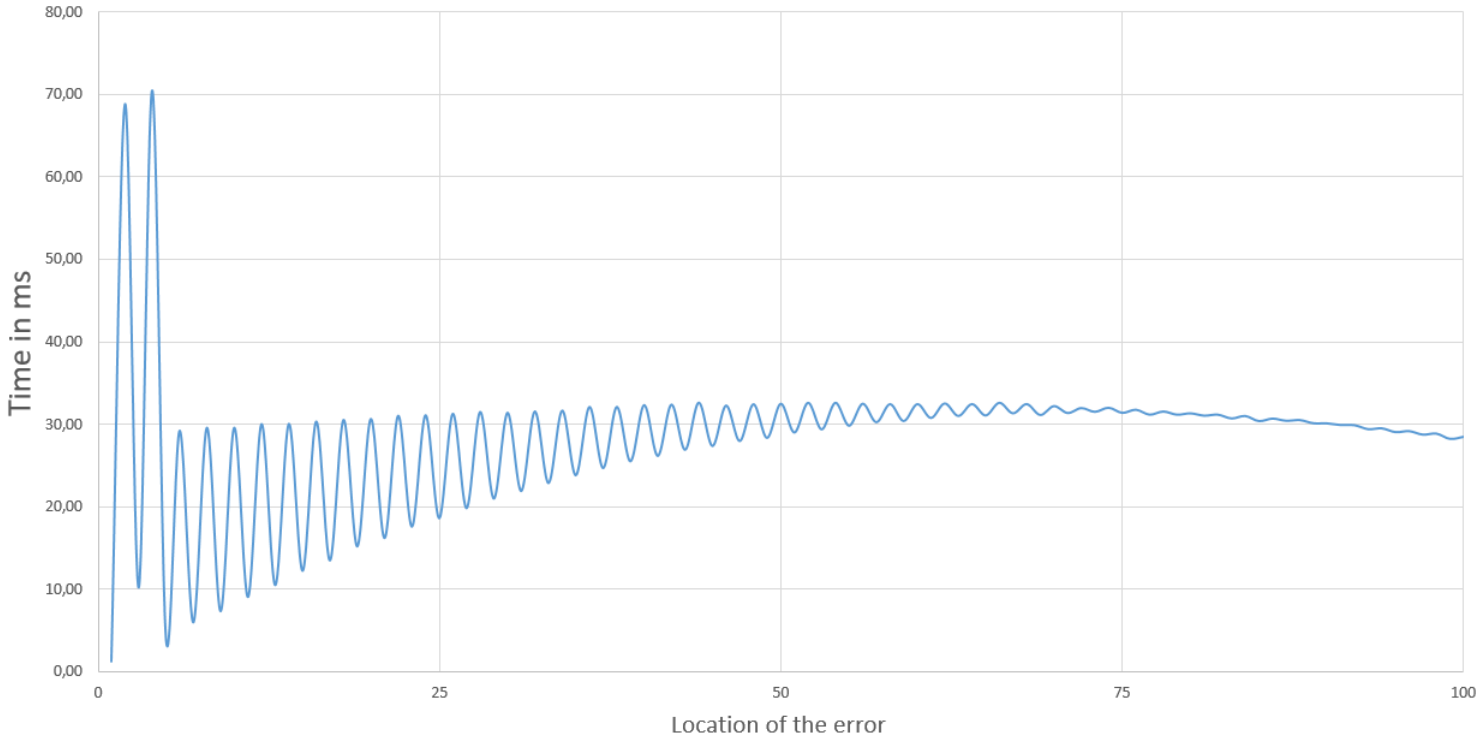

Figure 6: Measurement of the running time with Top-down approach about the error location

5 Conclude

In this section we are going to conclude about the computed performance measurements and speak about the differences between the Top-down and the Bottom-up approach.

5.1 Difference between Bottom-up and Top-down approach

5.1.1 The size of the input

The Bottom-up approach is more efficient than Top-down because of the size of the inputs. Indeed, if we have a huge input Top-down could end with a stack buffer overflow because of recursivity.

5.1.2 Number of iterations VS number of calls

We see that the size of the input will impact the number of iterations or calls and as the running time is related to this a larger input will take more time.

5.1.3 Impact of the specific grammar

Taking in account the two previous cases and what we said about the specificity of grammars we showed that sometimes the behaviors change. The way grammars are defined can change the assumption for Top-down. We see that Top-down, depending on the location of the error, earns a huge running time efficiency whereas Bottom-up still is quite the same.

5.2 Improvement suggestions

The strength of Bottom-up allows us to define specific data-structure according to the problem we have to solve. We can find a way to change the table that stores results to reduce unused cells and makes it work in parallel. The parallel implementation will work on the computation of cells by layer only because of the fact that upper layer needs lower layer to compute its own layer.

5.3 Finally

Thank to this practical experience with these given grammars we see that Top-down is really better than Bottom-up.

Next section, we will see practical problems with linear grammars under the Chomsky Normal Form. We will need to adapt the actual algorithms, study

behaviors of Bottom-up and Top-down approach and maybe Top-down will be less efficient than Bottom-up with such grammars ?

6 Linear grammar

In this section, we will try to restrict standard algorithms to special cases in order to gain efficiency. We will focus a specific type of context-free grammars which is linear grammars. We will first introduce linear grammars and understand them. In a second time we will adapt, modify our existing algorithms and finally see the impact of the running time of the CKY algorithm.

6.1 Linear grammars

6.1.1 Definitions

In computer science, a linear grammar is a context-free grammar that has at most one non terminal in the right hand side of each of its productions. A linear language is a language generated by some linear grammar.

A grammar is right-linear if all its rules are one of the two following forms :

- A $\rightarrow \omega$B with A and B $\in$ N and $\omega \in$ T
- A $\rightarrow \omega$ with A $\in$ N and $\omega \in$ T

A grammar is left-linear if all its rules are one of the two following forms :

- A $\rightarrow$ Bω with A and B $\in$ N and $\omega \in$ T
- A $\rightarrow \omega$ with A $\in$ N and $\omega \in$ T

These grammars are also called regular grammars.

6.1.2 How to turn a linear grammar into a Chomsky Normal Form

As we know a context-free grammar is under a Chomsky Normal Form if all its rules are under one of the following forms :

- **X** $\rightarrow$ **AB** where **X, A, B** are non-terminals.
- **X** $\rightarrow$ **a** where **X** is a non-terminal and **a** is a terminal.

So, to turn a linear grammar into a Chomsky Normal form the aim is to determine whether the right part of a rule contains a terminal and a non-terminal. If so, then we need to replace the terminal by a non-terminal and create a new production rule where the non-terminal yields the terminal.

example : if S → aA then it will become S → BA and we create a new production rule B → a.

6.2 CKY algorithm with Linear Grammars

Hopefully we can turn a Linear Grammar into Chomsky Normal Form. In this section, we will apply our designed CKY algorithms with Linear Grammars turned into Chomsky Normal Form. First, we will analyze them and see the impact. Then, we will do an empirical evaluation with new and previous designed algorithm and compare it with different grammars.

6.2.1 An example is

Let's take this linear grammar as example :

- S → Ac | b
- A → aS | aB
- B → bS

The language is infinite because you can always get back to S from A or B.

- S → Ac
- → aSc
- → aAcc
- → aaScc
- → aaAccc
- → aaaBccc
- → aaabSccc
- → aaabAcccc
- → aaabaBcccc
- → aaababSccccc
- → aaababScccc

6.2.2 Bottom-up Approach

According to me, I do not think that the Bottom-up approach will be significantly enhanced with Linear Grammars turned into CNF. Indeed, we see that the Bottom-up approach consists of filling a table layer by layer. As we know we have an array of 3 dimensions that is filled : one dimension is for input length, second is for input length, last is for the grammar length (so the number of rules). If we test similar grammar with a equivalent input and same size the only thing that can modify a little bit the running time is the number of rules of the grammar.

6.2.3 Top-down Approach

The Top-down approach is, according to me, more interesting. Indeed, we have to take in account the specificity of the linear grammar either left or right. If we look at our current Top-down designed algorithm, we notice that every partition is analyzed from left to right. Moreover if we remember the specificity of Right-linear grammar defined above (example : S $\rightarrow$ aB) we can notice we can quickly validate the tested words because non-terminals are on the right part. Thank to the structure of Right-linear grammar we can reduce the amount of calls and enhance running time.

Example :

Let's take this rule from a Right-linear grammar : **A** $\rightarrow$ **abcB** and this string : **abcabb**. We can see that the first part of the string **"abc"** matches the first part of the rule. Now we need to determine whether or not the rule **B** can recognize the string **abb**.

6.3 "Enhancing", "Optimizing" the current CKY algorithms with both approach

In this part, we will see whether we can optimize, enhance our current designed algorithms for Top-down and Bottom-up approach.

6.3.1 Changes

We know that a linear grammar on the right side of a rule contains either a single terminal symbol or two symbols composed of one terminal and a non-terminal.

We also know that CNF contains, on the right side of a rule, either two non-terminals or one terminal.

Thank to this difference we can notice what we should modify for both algorithms. Indeed, the for loop used to browse every partition is now "useless". We

will need to remove the partitions loop and keep the cases where the terminal is the first or last symbol of the word. The rest will be a single symbol handle by a non-terminal.

6.3.2 Efficiency

As we said above we remove the partition loop. Doing that will enhance the running time of Bottom-up approach and also the Top-down approach.

With the Top-down approach we do 4 recursive calls instead of 2 * number of partitions. On the other hand, for the Bottom-up this is the "same thing". Filling a cell of a layer will need 2 cells instead of 2 * number of partitions.

The new running time for Top-down is $\mathbf{O(n^2N)}$ and for Bottom-up is $\mathbf{O(n^2N)}$ because we remove the loop of partitions of length n.

6.3.3 Pseudocode of optimized algorithm

Algorithm 6: Bottom-Up version of CKY Algorithm with Linear-Grammar

```
begin
    Create a table[n][n][r] where n represents the length of the input
    string and r the grammar size.
    for i=1 to n do
        for j=1 to n do
            for k=1 to r do
                table[i][j][k] = false;
            end
        end
    end
    for i=0 to n do
        for all rules Na → input[i][i] do
            table[i][i][Na] = true;
        end
    end
    for l=2 to n do
        for i=1 to n - l + 1 do
            for all production Na→ Nb Nc do
                if table[i][i][Nb] and table[i + 1][i + l - 1][Nc] then
                    table[i][i + l - 1][Na] = true;
                end
                if table[i][i + l - 2][Nb] and table[i + l - 1][i + l - 1][Nc]
                 then
                    table[i][i + l - 1][Na] = true;
                end
            end
        end
    end
    return table[n]
end
```

Algorithm 7: The Top-Down version of CKY Algorithm with Linear-Grammar

```
input: A represents the non terminal, i the start index, j the length index
begin
    tmp = table[i][j][A];
    if tmp != null then
        return tmp;
    end
    if i==j then
        isRule = isRule(A, i);
        table[i][j][A] = isRule;
        return isRule;
    end
    for all the rule Na → Nb Nc do
        if TopDown(Nb, i, i) and TopDown(Nc, i + 1, j) then
            table[i][j][Na] = true
            return true;
        end
        if TopDown(Nb, i, j - 1) and TopDown(Nc, j, j) then
            table[i][j][Na] = true
            return true;
        end
    end
    table[i][j][Na] = false
    return false
end
begin
    Function : isRule(A, i)
    for all rules A -> a do
        if A == a then
            if a == input[i] then
                return true
            end
        end
    end
    return false
end
```

6.3.4 Experimental measurements

In this section we are going to do some performance measurements with the previous given grammar. We will compare the old and new algorithm with the same grammar and with input of different lengths.

The first test is based on the evolution of iteration (see Figure 7). We can notice that the new optimize algorithm is far better than the old one. Moreover, the new Top-down approach still better than the new Bottom-up approach.

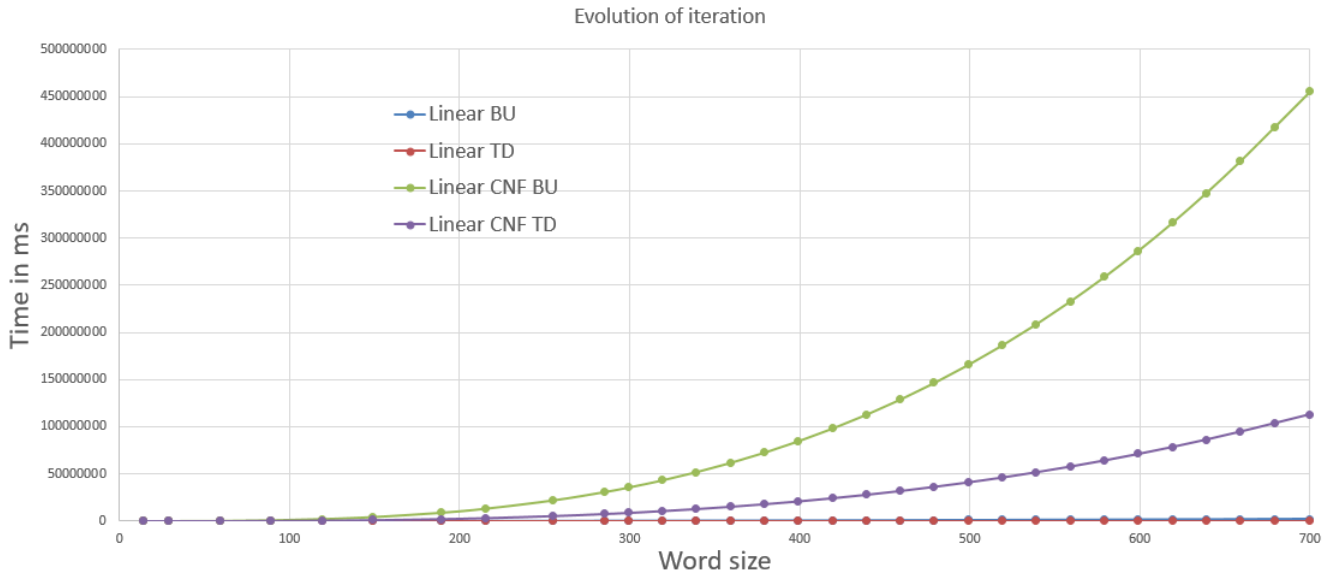

Figure 7: Evolution of iteration and comparison between old and new CKY algorithm

The second test is based on the running time (see Figure 8). This test confirms our theoretical analyze. The running time of new algorithms is significantly faster than old algorithms.

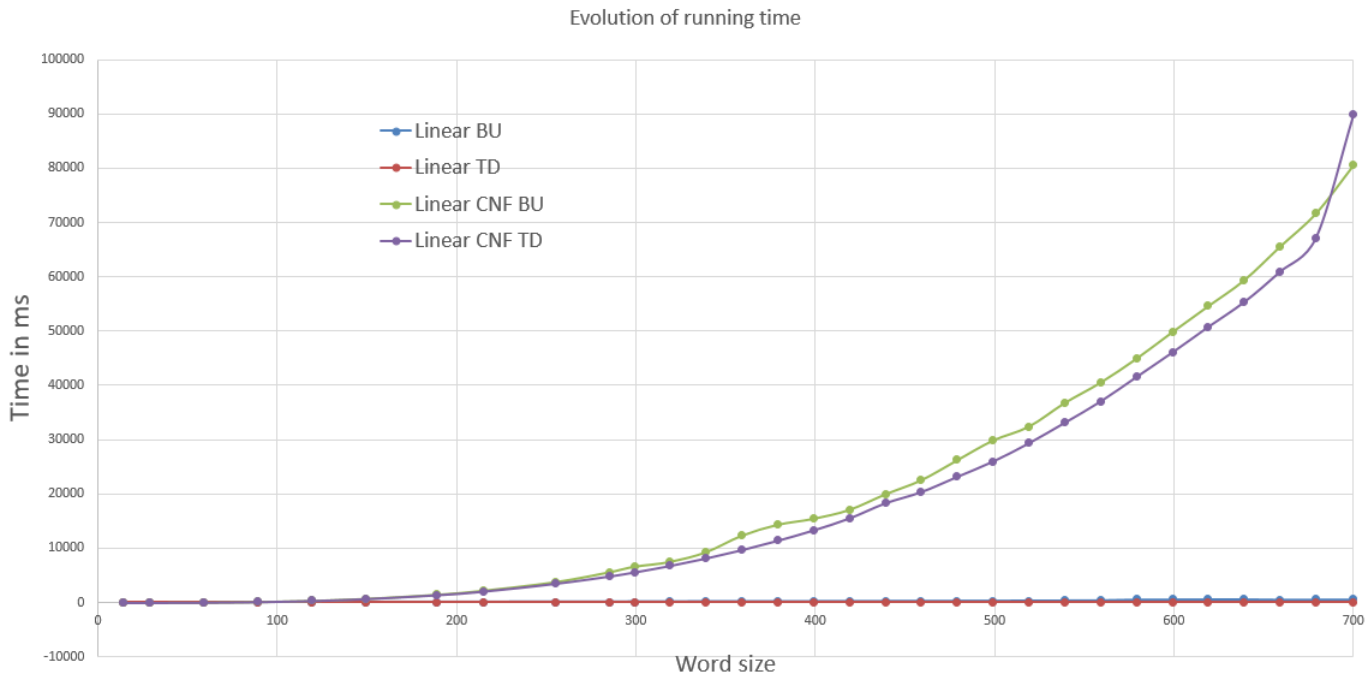

Figure 8: Running time comparison between old and new CKY algorithm

Thank to this two tests we see that using an adapted algorithm allows us now to parse important words' size in a very short time.