Step 4

Jules Sang
jules.sang@grenoble-inp.org

June 25, 2021

Contents

Abstract

In this paper, the efficiency of different parsing algorithms is analyzed, both for Chomsky normal form grammars and linear grammars.

First part investigates the parsing of Chomsky normal form grammars with (i) a naive top-down algorithm, (ii) a memoization algorithm, and (iii) the widely-known Cocke-Younger-Kasami tabulation algorithm.

Second part investigates linear grammars' parsing with (i) converting linear grammars to Chomsky normal form, then applying the Cocke-Younger-Kasami algorithm, (ii) directly adapting the Cocke-Younger-Kasami algorithm to linear grammars.

Both parts show theorical and empirical views of the algorithms' efficiency.

Introduction

Parsing is the process of analysing a string of symbols, conforming to the rules of a formal gramar. It is a fundamental computer science notion: The parser is one of the main parts of compilers and interpreters, which describe how programs should be executed by the machine. Since most modern languages are context-free (and thus expressible by context-free grammars 1.1), parsers for context-free languages can be used to check that programs are syntactically correct. Having efficient parsing algorithms hence allows a faster compilation or interpretation of programs.

1 Central notions

1.1 Context-free grammar (CFG)

Parsers need set of formal rules in order to interpret the symbols of input strings, and define syntactic relations between symbols. This set of formal rules is called a grammar, and is used to express a language's semantics. A context-free grammars is a special type of grammar, that can express most modern programming languages.

1.1.1 Definition

A context-free grammar is defined by $G = (N, \Sigma, P, S)$, where:

1. N is a finite set, $A \in N$ is called a nonterminal element.
2. Σ is a finite set, $N \cup \Sigma = \emptyset$, $\sigma \in \Sigma$ is called a terminal element.
3. $P : N \to (N \cup \Sigma)^*$ is a finite set of production rules. The "*" symbol expresses a multiplicity. Here it means "the concatenation of any number of elements from N and Σ".
4. $S \in N$ called the starting symbol of G.

$L(G) = \{w \in \Sigma^* | S \text{ generates}^1 \ w\}$ is called the language generated by G, and corresponds to the set of terminal strings that can be generated starting from S with production rules from P.

Example of CFG

$$G = (N, \Sigma, P, A)$$
$$N = \{A, B\}$$
$$\Sigma = \{\alpha, \beta\}$$
$$P = \{A \to \alpha B \beta | \alpha, B \to \beta A \alpha | \beta\}$$

For example, $\{\alpha, \alpha\beta\alpha, \alpha\beta\alpha\alpha\} \subset L(G)$

1.2 Chomsky normal form (CNF) grammars

A context-free grammar G is said to be in Chomsky normal form if all of its production rules are of the form

- $A \to BC$ or
- $A \to \alpha$ or
- $S \to \epsilon$

Where A, B, C are nonterminals, S is the starting symbol of G, and ϵ denotes the empty string. Also, if $S \to \epsilon$ is a production rule of G, S may not appear in the right-hand side of a production rule. Any CFG can be transformed into a CNF grammar, with a size no larger than the square of the initial grammar's size.

1.3 Linear grammars

A context-free grammar is said to be linear if there is at most one nonterminal symbol in the right-hand side of its production rules.

1.4 Some common algorithmic techniques

Here are the main idea of the algorithmic techniques used in this paper.

[1] Given a production rule $A \to \alpha$, we say that A generates α. If $A \to B$, and $B \to \alpha$, A generates α by transitivity.

1.4.1 Tabulation

Tabulation is a dynamic programming technique that focuses on building up larger and larger subsolutions to a problem until the target solution has been reached. Each iteration uses the results of the previous ones to compute the solution faster. Tabulation algorithm are bottom-up and iterative by nature.

Example of tabulation algorithm Here is how one would use tabulation for computing fibonacci:

```
function fibo(n):
    mem[0] <- 0
    mem[1] <- 1

    for i in 2...n:
        mem[i] = mem[i-1] + mem[i-2]

    return mem[n]
```

1.4.2 Memoization

Memoization is a dynamic programming technique where the result of time-expansive function calls are stored in a data structure, in order to avoid recomputing them if the same inputs occur again. Memoization algorithm are top-own and recursive by nature.

Example of memoization algorithm Here is how one would use memoization for computing fibonacci:

```
function fibo(n):
    if n < 2:
        return n

    if mem[n] is null:
        mem[n] = fibo(n-1) + fibo(n-2)

    return mem[n]
```

2 Parsing algorithms

2.1 Cocke-Younger-Kasami (CYK) algorithm

The CYK algorithm is a parsing algorithm: in its standard version, the input is G, a context-free grammar in CNF, and α a string. The output is true if and only if $\alpha \in L(G)$. It is easy to tweak the algorithm so that it returns the rules of G used for generating α, without additional space or memory costs.

The original version of the CYK algorithm is bottom-up. The two other versions presented are variants.

2.1.1 Main idea

2.1.2 Naive top-down implementation

Let G be the grammar, Σ the set of nonterminals symbols of G, P the rules of G, α the string to parse and $S \in \Sigma$ the starting symbol of G.
Here is how *parse(S, α)* behaves:

- If $|\alpha| = 1$: Return true if $(S \to \alpha) \in P$ else return false
- If $|\alpha| > 1$: For each $(S \to AB) \in P$, and for each possible partition of α into two parts β and δ, return true if both *parse(A, β)* and *parse(B, δ)* return true. If no such combination of (A, B, β, δ) is found, return false

This algorithm consists of a recursive enumeration of each appliable rule.

```
function naive(G: the grammar, S: the
    starting symbol, alpha[i, j]: the
    string):
    if j = 1:
        return G.contains(S -> alpha)

    for each P production rule of G, of
        the form S -> AB:
        for k = i...j:
            if naive(G, A, alpha[i, k]) and
                naive(G, B, alpha[k, j]):
                return true

    return false
```

Complexity: Note that recursive calls with substring alpha[i, j] imply a copy of that substring, and hence increases the spatial complexity. Also note that G is not considered to be part of the input, and $|G|$ can hence be disregarded when computing the complexity.

The exact time cost is hard to compute, but we can get a worst case lower time bound which will be sufficient for comparing this algorithm with the other ones:
Given an instance of size n, if we consider the two recursive calls of depth $n-1$ (rooted in the instances $[1, n-1]$, and $[2, n]$), we get a recursion tree of height n and branching factor 2. At level i of input size m, the time complexity excluding the recursive calls is $O(m)$, and level $i+1$ has input size $m-1$. We can hence express our lower bound as follows:

$$T(n) < n + 2(n-1)|G| + 4(n-2)|G| + \cdots + 2^{n-1}|G|$$
$$\Leftrightarrow T(n) < |G| \sum_{i=0}^{n-1} 2^i (n-i)$$
$$\Leftrightarrow T(n) = \Omega(2^{n-1}) = \Omega(2^n)$$

Note that $\Omega(2^n)$ is a worst-case lower time bound. If the first leaf of the recursion tree is true, $T(n) = n$.

In the next sections, we will see how dynamic programming can reduce the execution time.

2.1.3 Top-down implementation with memoization

The Top-down implementation of the CYK algorithm works just like the naive implementation but maintains a global three-dimensional boolean table Z.
Each time a recursive call such as "*parse(A, $\alpha[i, j]$)*" is made (A being the starting symbol, and α the string to parse from index i to j), the result is stored in $Z[A, i, j]$, so that if the result is needed again, it can be accessed in constant time.
That programming technique is called memoization and has the following characteristics:

- It stops on the first positive result, instead of going through all of the possible subproblems, which can save time
- It has a recursive nature, which makes it easy to write but can cause excessive memory use with unadapted compilers
- It only requires a small adaptation of the naive top-down method, which is maintain and use the global table Z

Pseudocode Here is the pseudocode for the top-down implementation:

```
global Z
function TD(G: the grammar, S: the starting
    symbol, alpha[i, j]: the string):
```

```
    if j = 1:
        return G.contains(S -> alpha)

    for each P production rule of G, of the
        form S -> AB:
        for k = 0...j:
            if Z[A, i, k] is null:
                Z[A, i, k] <- TD(G, A, alpha[i,
                    k]

            if Z[B, k, j] is null:
                Z[B, k, j] <- TD(G, B, alpha[k,
                    j]

            if Z[A, i, k] and T[B, k, j]:
                return true

    return false
```

Complexity: At most, there will be as many recursive calls as there are cells in T, i.e. $|G|n^2$. Since the body of the function, except the recursive calls has a time complexity of $O(n)$, the total time complexity is $T(n) = O(n) * O(|G|n^2) = O(|G|n^3) = O(n^3)$

2.1.4 Bottom-up implementation with tabulation

Let G be the grammar, S the starting symbol, α the string to parse.

The following bottom-up implementation of the CYK algorithm is the original version that can be found in textbooks. The idea is to maintain a table of substrings that can be parsed by rules of the grammar, storing the possible starting symbol(s). The iterations start at substring size 1, and end at substring size $|\alpha|$. If S is stored in the table for the size $|\alpha|$, $\alpha \in L(G)$. Note that previous iterations are used for building the next ones.
Let P be the table of substrings. $P[a, b, c]$ is true iff the substring of α starting at index c and of length a can be parsed with the production rule of index b.
That programming technique is called tabulation and has the following characteristics:

- It has an iterative nature, which does not cause excessive memory use with any compiler, but it is usually harder to program
- It computes all of the possible problems, which can cost a bit of compututational time (without increasing the asymptotic complexity), but allows to give instantly the solution to any subproblem afterwards ($\bigcup_b P[a, b, c] \equiv parse(\alpha[c, c + a])$.

Pseudocode Here is the pseudocode for the bottom-up implementation:

```
function TD(G: the grammar, alpha[1, n]: the
    string):
    R[1, r] <- G.nonterminal_symbols // with
        R[1] the starting symbol of G
    let P[n, n, r] an array of boolean with
        all values initialized to false
    for s in 1...n:
        for each production rule R[v] -> a[s]:
            P[1, v, s] <- true

    for l in 2...n:
        for s in 1...n-l+1:
            for p in 1...l-1:
                for each production rule R[a]
                    -> R[b]R[c]:
                    if P[p, s, b] and P[l-p,
                        s+p, c]:
                        P[l, s, a] <- true

    return P[n, 1, 1]
```

Complexity: Most of the work is done in four imbricated loops, three of them are bounded by n, and the last one bounded by $|G|$. We hence have $T(n) = O(n^3|G|) = O(n^3)$

3 Empirical measurements

3.1 Comparison with theoritical running time estimations

3.1.1 Well-balanced parentheses

The grammar used for the next tests only accepts well balanced parentheses:

$$S \to SS|LA|LR$$
$$A \to SR$$
$$L \to ($$
$$R \to)$$

Given the respective theorical complexities, we should expect a running time growing extremely fast as n grows for the naive algorithm, and similar running times for the top-down and the bottom-up dynamic programming variants.

$\frac{n}{2}$ opening followed by $\frac{n}{2}$ closing, i.e. (((...)))

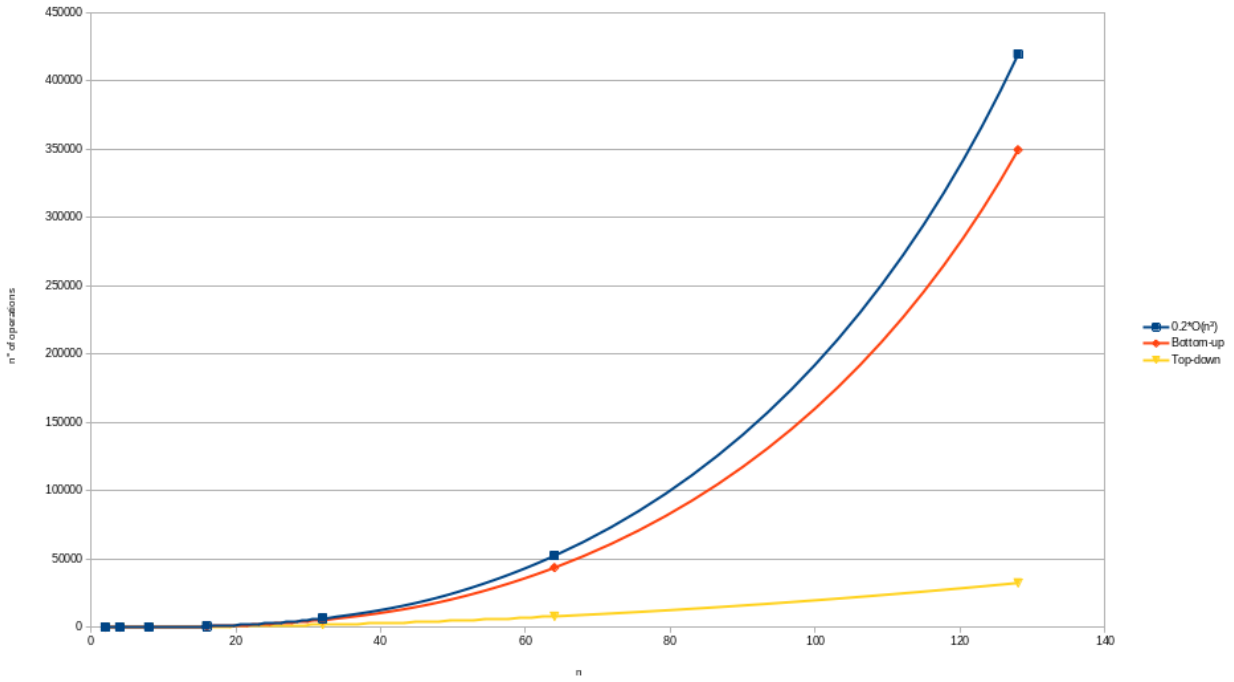

Figure 1: "(((...)))" - *The naive number of operations is too important to be on the graph*

As expected, the naive algorithm's complexity is far more important than the two other algorithms'. The top-down version is probably more efficient than the bottom-up version because, given the way it is implemented, it finds a solution very quickly and does not waste too much time with solutionless recursive calls, while the bottom-up version computes the parsing of every substring with every production rule no matter what.

$\frac{n}{2}$ pairs, i.e. '()()...'

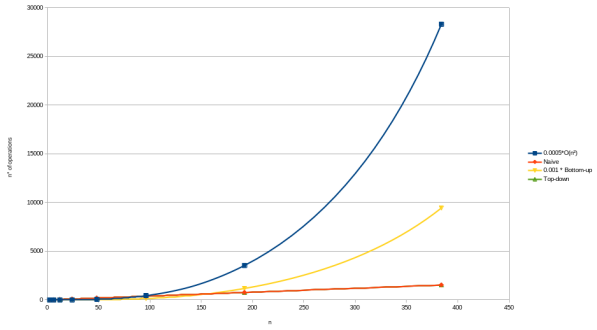

Figure 2: "()()..." Measured number of operations

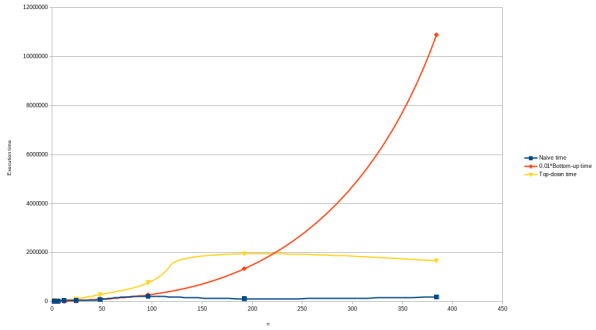

Figure 3: "()()..." Measured execution time

Here, both the naive and the top-down variants show a linear complexity, whilst the bottom-up variant has a quadratic complexity. This probably comes from the fact that the bottom-up variant performs the parsing for every possible substring while the others algorithms stop at the first positive recursive call.

For the naive and the top-down DP algorithms, if the first recursive call (i.e. $k = 0$) is positive each time, the cost is linear, since the recursion tree is equivalent to a basic for loop over the string. This is why the "number of operations" curves seem close to linear for the top-down and naive algorithms.

It is interesting to note that, in terms of effective computation time, the naive version is slightly faster than the top-down one. That is probably because it has no additional table management.

closing parenthesis then $\frac{n}{2} - 1$ pairs, i.e. ')()()...'

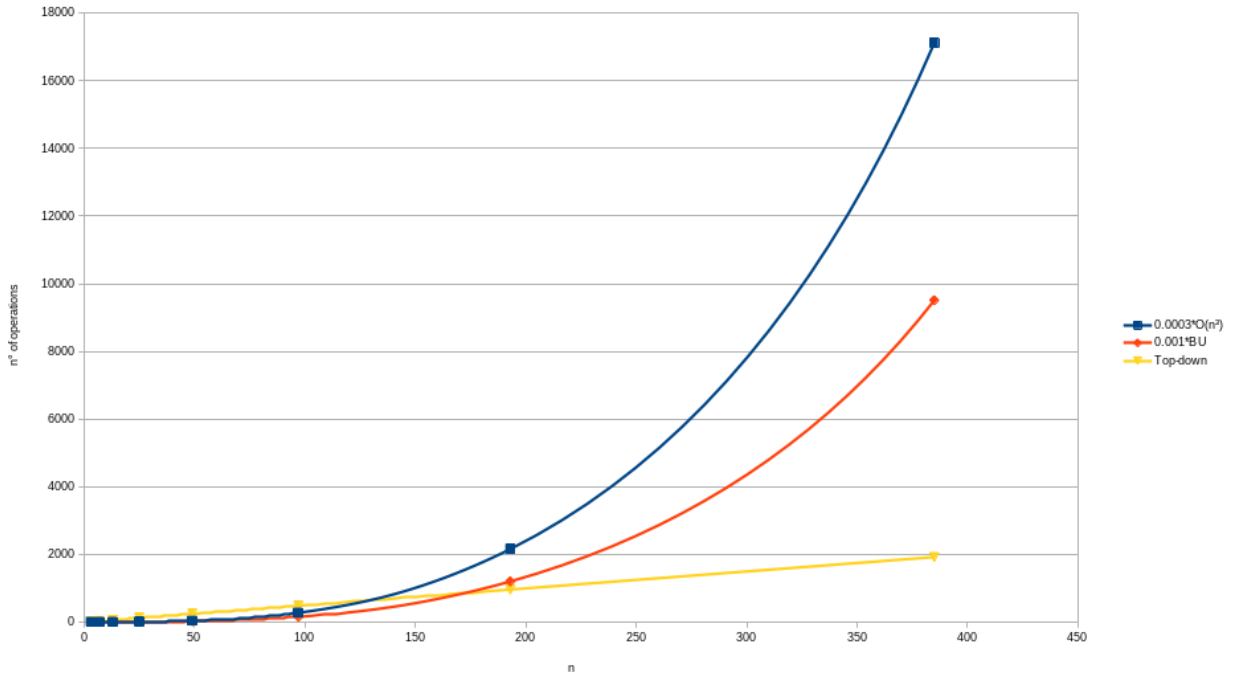

Figure 4: "()()..." - *The naive number of operations is too important to be on the graph*

Here, we obtain the same behavior as in the "$\frac{n}{2}$ opening followed by $\frac{n}{2}$ closing, i.e. (((...)))" case, but with an overall smaller numbers of operations. The top-down version's number of operations also seems close to linear again.

$\frac{n}{2} - 1$ **pairs, then opening parenthesis, i.e. '()()...('**

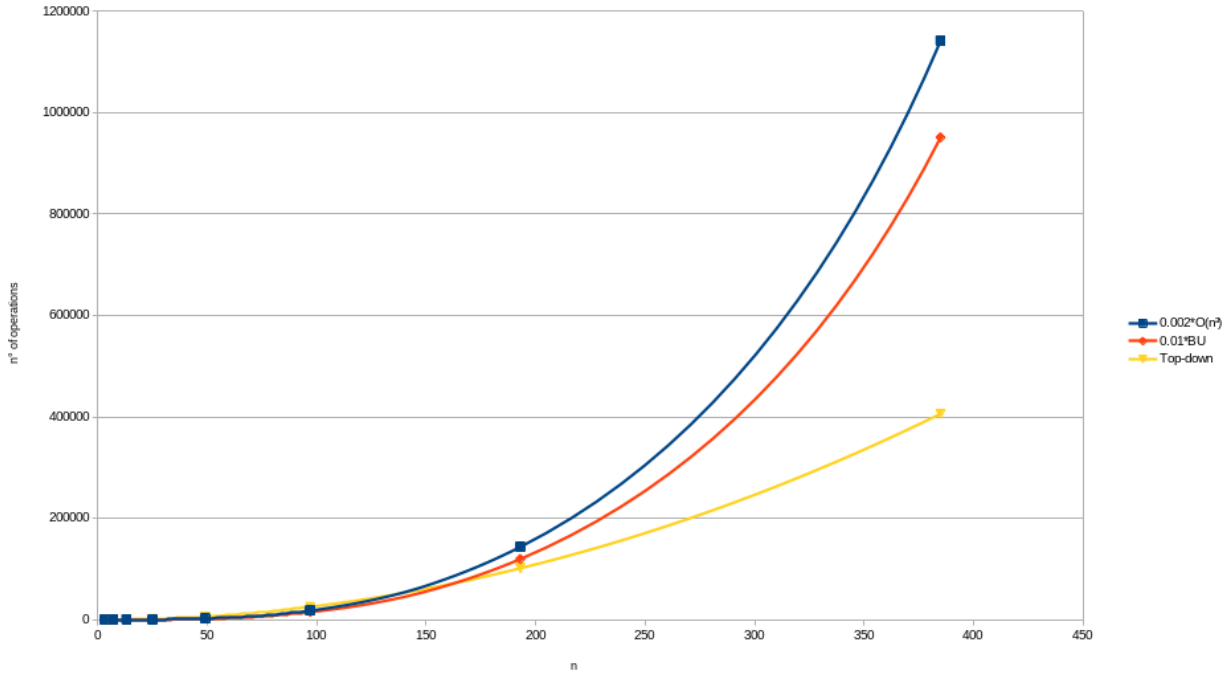

Figure 5: "()()...(" - *The naive number of operations is too important to be on the graph*

Here, we again have the same behavior as in the "$\frac{n}{2}$ followed by $\frac{n}{2}$ closing" case.

3.1.2 Stupid grammar

The grammar used for the next tests does not generate any word (i.e. $L(G) = \emptyset$) but the parsing algorithms can still be applied on it. Here is how it looks:

$$S \to ST|TS$$
$$T \to a$$

several a's, i.e. 'aa...'

The algorithms will end-up halting because, even though following the rules until getting a string is infinite (since we always have nonterminal symbols), in the top-down case, once the length of the input string is exceeded the computation is stopped, and in the bottom-up case, once the table is filled, there is no computations left to do. If the algorithms didn't end up halting, we would not be able to guarantee a $O(n^3)$ running time anyway.

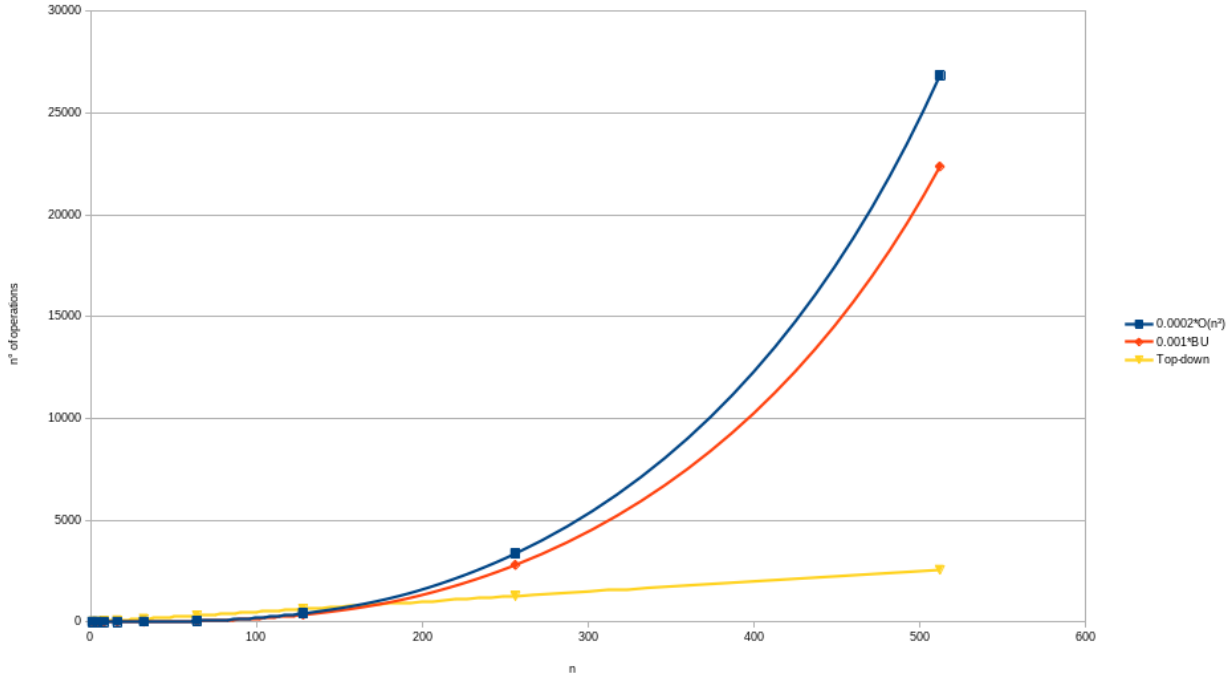

Figure 6: "aa..." - *The naive number of operations is too important to be on the graph*

The reason why the top-down version is faster than the bottom-up one is probably that all the possible computations are quickly stored in the table, which then allows to stop using recursive calls, whilst the bottom-up version does not have that kind of "memory".

4 Linear grammars

4.1 Turn it into Chomsky normal form

This is the easy solution: Transform each rule that does not match CNF into CNF rule:

$$A \to B\alpha \equiv A \to BC, C \to \alpha$$
$$A \to \alpha B \equiv A \to CB, C \to \alpha$$

Since each initial rule generates at most 2 CNF ones, the size of the generated CNF grammar is $O(2|G|) = O(|G|)$, $|G|$ being grammar's size. The transformation's time complexity is $O(|G|)$.

Since the CYK algorithm has a time complexity of $O(n^3|G|)$, and since $|G|$ can be considered unchanged after the transformation, the total parsing time complexity is $O(|G|) + O(n^3|G|) = O(n^3)$.

4.2 Adapting the CYK algorithm

It is possible to adapt the CYK algorithm to parse linear grammars, changing the way the global three-dimensional table is updated. The idea is that instead of testing all the possible string partitions, we check partitions of size $(1, n-1)$ for rules of form $A \to \alpha B$ and partitions of size $(n-1, 1)$ for rules of form $A \to B\alpha$.

Formally, in the bottom-up version, the table is now filled as follows:
Let $G = (N, \Sigma, P, S)$, N being the nonterminal symbols, Σ the terminal symbols, P the production rules, and S the starting symbol. let t a global three-dimensional table, s the parsed string and $n = |s|$.
$t[i,j] = \bigcup_{A \in N} \{A | A \to s[i]t[i-1, j+1] \in P$
$\vee\, A \to t[i-1, j]s[i+j-1] \in P\}$
for $i \in [1, n], j \in [1, n-i+1]$
If $S \in t[1, n] \quad s \in L(G)$, else $s \notin L(G)$.

Since the algorithm iterates over a n^2 sized array, and since the whole grammar is iterated for each cell, $T(n) = O(n^2|G|) = O(n^2) < O(n^3)$. It is hence worth it to adapt the CYK algorithm instead of turning G into CNF.

4.3 Empirical measurements

It is interesting to compare the two precedent methods to verify that both follow the expected behavior.

4.3.1 abc

The grammar used for the next tests only accepts strings of form $a^k b^l c^k \ \forall k \in \mathbb{N} \ \forall l \in \mathbb{N} \backslash 0$:

$$G = (N, \Sigma, P, A)$$
$$P = \left\{ \begin{array}{l} S \to Ac \\ S \to b \\ A \to aS \\ A \to aB \\ B \to bS \end{array} \right\}$$

Inputs from $L(G)$

Here are the measured time complexities for strings from $L(G)$.

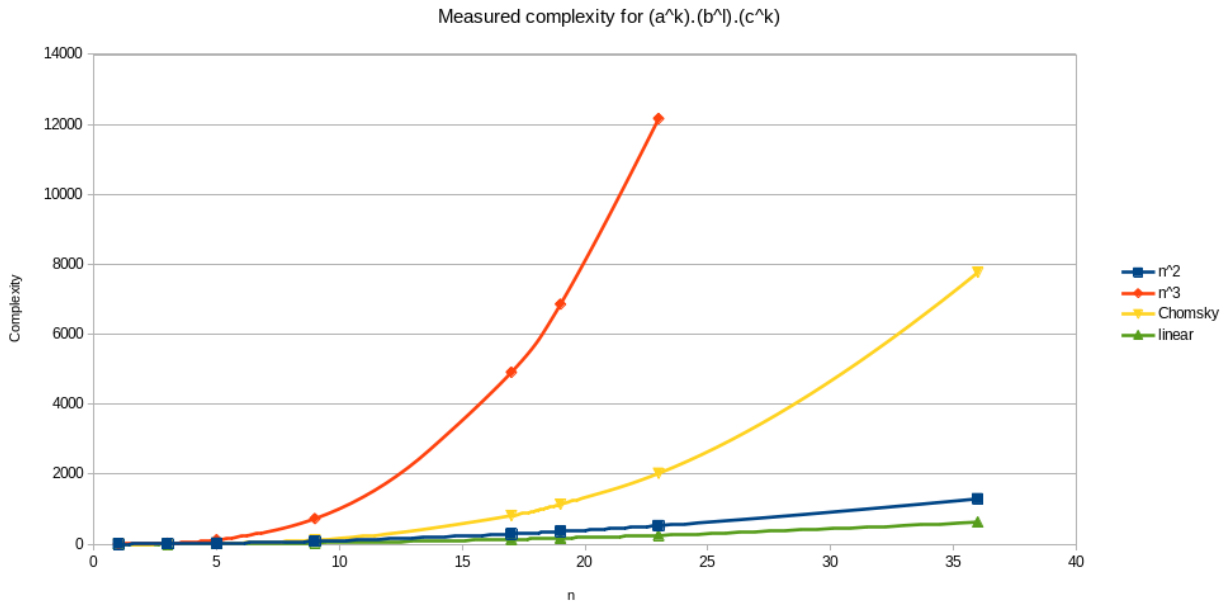

Figure 7: $a^k b^l c^k$

As expected, the adapted CYK algorithm's is of execution time $O(n^2)$ for the given inputs, while turning G into a CNF grammar, and then executing the CYK algorithm on the new CNF grammar is of execution time $O(n^3)$.

Inputs not from $L(G)$

Here are the measured time complexity for random strings not from $L(G)$, i.e. $\{\alpha = (a+b+c)^n | \alpha \notin L(G)\}$.

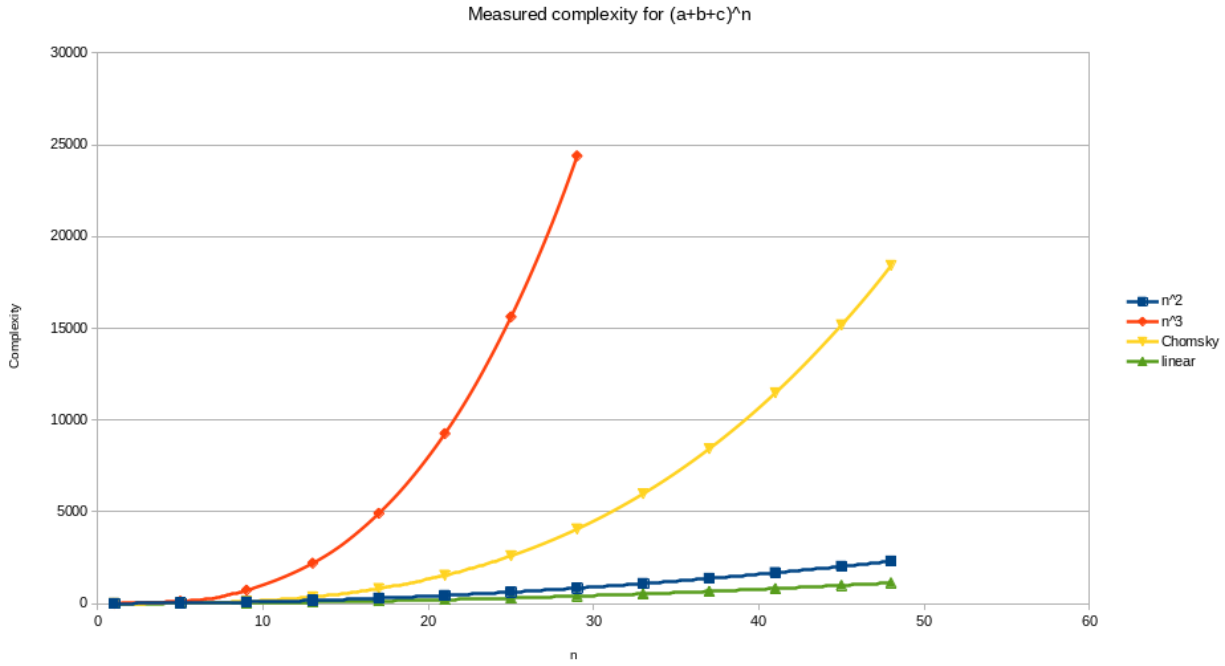

Figure 8: $(a+b+c)^n$

With wrong inputs, The measured complexities preserve the same behavior. It is hence indeed more efficient to adapt the CYK algorithm to linear grammars instead of translating linear grammars into CNF.