```
#include <stdio.h>
#include <string.h>
#include <ctype.h>
#define MAX_WORDS 100
#define MAX_LEN 2
// Training Data (word-tag pairs)
const char* tagged_sentences[] = {
    "The/DT cat/NN sits/VBZ on/IN the/DT mat/NN",
    "A/DT dog/NN barks/VBZ loudly/RB",
    "Children/NNS play/VBP outside/RB"
};
typedef struct {
    char word[MAX_LEN];
    char tag[MAX_LEN];
} Token;
Token training_data[MAX_WORDS];
int token_count = 0
// Rule-based tagging
const char* rule_based(const char* word) {
    int len = strlen(word);
    if (strcmp(word, "the") == 0 || strcmp(word, "The") == 0 || strcmp(word, "a") == 0 || strcmp(word, "A")
== 0)
        return "DT";
    if (len > 2 && strcmp(&word[len - 2], "ly") == 0)
        return "RB";
    if (len > 1 && word[len - 1] == 's')
        return "NNS";
    return "NN";  // Default to noun
}
// Stochastic tagging (Unigram: most frequent tag per word)
const char* unigram_tagger(const char* word) {
    int max_count = 0;
    const char* best_tag = "NN";
    for (int i = 0; i < token_count; i++) {
        if (strcmp(training_data[i].word, word) == 0) {
            int count = 0;
```

```
        for (int j = 0; j < token_count; j++) {
            if (strcmp(training_data[j].word, word) == 0 && strcmp(training_data[j].tag, training_data[i].tag)
== 0)
                count++;
        }
        if (count > max_count) {
            max_count = count;
            best_tag = training_data[i].tag;
        }
    }
  }
  return best_tag;
}
// Split tagged training data into word/tag pairs
void process_training_data() {
  for (int i = 0; i < sizeof(tagged_sentences) / sizeof(tagged_sentences[0]); i++) {
    char line[200];
    strcpy(line, tagged_sentences[i]);
    char* token = strtok(line, " ");
    while (token != NULL && token_count < MAX_WORDS) {
      char* slash = strrchr(token, '/');
      if (slash) {
        *slash = '\0';
        strcpy(training_data[token_count].word, token);
        strcpy(training_data[token_count].tag, slash + 1);
        token_count++;
      }
      token = strtok(NULL, " ");
    }
  }
}
void tag_sentence(const char* sentence) {
  char copy[200];
  strcpy(copy, sentence);
  char* word = strtok(copy, " ");
```

```
    printf("\nOriginal: %s\n", sentence);
    printf("Rule-based tags:\n");
    while (word) {
        printf("%s/%s ", word, rule_based(word));
        word = strtok(NULL, " ");
    }

    strcpy(copy, sentence);
    word = strtok(copy, " ");
    printf("\nStochastic (Unigram) tags:\n");
    while (word) {
        printf("%s/%s ", word, unigram_tagger(word));
        word = strtok(NULL, " ");
    }
    printf("\n");
}

int main() {
    process_training_data();

    const char* test_sentence = "Children play loudly";
    tag_sentence(test_sentence);

    return 0;
}
```

4. Absolutely! Let's implement both **top-down** and **bottom-up parsing** using **Python NLTK (Natural Language Toolkit)**.

🧠 **What's the Difference?**

🔼 **Top-Down Parsing:**

- Starts with the **start symbol** (e.g., S) and tries to **rewrite it** to match the input sentence.

🔽 **Bottom-Up Parsing:**

- Starts with the **input words** and tries to build up the tree to the start symbol S.

✅ **Step-by-Step Implementation with NLTK**

1. Install NLTK (if not already installed)

```
pip install nltk
```

---

2. Python Code

```
import nltk
from nltk import CFG

# Define a simple grammar
grammar = CFG.fromstring("""
  S -> NP VP
  NP -> DT N | PRP
  VP -> V NP | V
  DT -> 'the' | 'a'
  N -> 'cat' | 'dog'
  V -> 'saw' | 'slept'
  PRP -> 'I'
""")

# Sample sentence
sentence = ['I', 'saw', 'a', 'dog']

# --- Top-Down Parser (Recursive Descent Parser) ---
print("Top-Down Parsing (Recursive Descent):")
rd_parser = nltk.RecursiveDescentParser(grammar)
for tree in rd_parser.parse(sentence):
    print(tree)
    tree.pretty_print()

# --- Bottom-Up Parser (Shift-Reduce Parser) ---
print("\nBottom-Up Parsing (Shift-Reduce):")
sr_parser = nltk.ShiftReduceParser(grammar)
for tree in sr_parser.parse(sentence):
    print(tree)
    tree.pretty_print()
```

✅ **Output (Example Tree):**

Top-Down Parsing (Recursive Descent):

```
(S
  (NP PRP I)
  (VP
    (V saw)
    (NP
      (DT a)
      (N dog))))
...
```

Bottom-Up Parsing (Shift-Reduce):

```
(S
  (NP PRP I)
  (VP
    (V saw)
    (NP
      (DT a)
      (N dog))))
...
```

Both parsers generate the same syntax tree, but **use different strategies** to reach it.

🧪 **Try It With Your Own Sentence!**

You can modify:

- The **grammar** (add Adj, PP, etc.)
- The **sentence** (e.g., ['the', 'cat', 'slept'])