1) Consider the following Corpus of three sentences a) There is a big garden.
b) Children play in a garden
c) They play inside beautiful garden
Calculate P for the sentence “They play in a big Garden” assuming a bi-gram language model.

```
public class Exp1 {
    public static int countBigrams(String[] corpus, String word1, String word2) {
        int count = 0;
        for (String sentence : corpus) {
            String[] words = sentence.split(" ");
            for (int i = 0; i < words.length - 1; i++) {
               if (word1.equalsIgnoreCase(words[i]) && word2.equalsIgnoreCase(words[i + 1]))
{
                    count++;
                }
            }
        }
        return count;
    }
    public static int countUnigrams(String[] corpus, String word) {
        int count = 0;
        for (String sentence : corpus) {
            for (String w : sentence.split(" ")) {
                if (word.equalsIgnoreCase(w)) {
                    count++;
                }
            }
        }
        return count;
    }
    public static void main(String[] args) {
        String[] corpus = {
                "There is a big garden",
                "Children play in a garden",
                "They play inside beautiful garden"
        };
        String[] testWords = "They play in a big Garden".split(" ");
        double probability = 1.0;
        int corpusSize = 9;

        for (int i = 0; i < testWords.length - 1; i++) {
            int bigramCount = countBigrams(corpus, testWords[i], testWords[i + 1]);
            int unigramCount = countUnigrams(corpus, testWords[i]);
            System.out.printf("Bigram count of ('%s', '%s'): %d%n", testWords[i], testWords[i
+ 1], bigramCount);
            System.out.printf("Unigram count of '%s': %d%n", testWords[i], unigramCount);
            probability *= (double) bigramCount / (unigramCount + corpusSize);
        }
```

```
        System.out.printf("Probability: %.8f%n", probability);
    }
}
```

```
java -cp /tmp/f1NuvOrqLf/Exp1
Bigram count of ('They', 'play'): 1
Unigram count of 'They': 1
Bigram count of ('play', 'in'): 1
Unigram count of 'play': 2
Bigram count of ('in', 'a'): 1
Unigram count of 'in': 1
Bigram count of ('a', 'big'): 1
Unigram count of 'a': 2
Bigram count of ('big', 'Garden'): 1
Unigram count of 'big': 1
Probability: 0.00000826

=== Code Execution Successful ===
```

2) Find the bigram count for the given corpus. Apply Laplace smoothing and find the bigram probabilities after add-one smoothing (up to 4 decimal places)

```
public class Exp2 {
    public static int countBigrams(String[] corpus, String word1, String word2) {
        int count = 0;
        for (String sentence : corpus) {
            String[] words = sentence.split(" ");
            for (int i = 0; i < words.length - 1; i++) {
                if (word1.equalsIgnoreCase(words[i]) && word2.equalsIgnoreCase(words[i + 1]))
{
                    count++;
                }
            }
        }
        return count;
    }
    public static int countUnigrams(String[] corpus, String word) {
        int count = 0;
        for (String sentence : corpus) {
            for (String w : sentence.split(" ")) {
                if (word.equalsIgnoreCase(w)) {
                    count++;
                }
            }
        }
        return count;
    }
    public static void main(String[] args) {
        String[] corpus = {
                "There is a big garden",
                "Children play in the garden",
                "They play inside beautiful garden"
        };
        String[] testWords = "They play in a big garden".split(" ");
        double probability = 1.0;
        int vocabularySize = 9;
        for (int i = 0; i < testWords.length - 1; i++) {
            int bigramCount = countBigrams(corpus, testWords[i], testWords[i + 1]);
            int unigramCount = countUnigrams(corpus, testWords[i]);
            double smoothedProbability = (double) (bigramCount + 1) / (unigramCount +
vocabularySize);
            System.out.printf("Bigram count of ('%s', '%s'): %d%n", testWords[i], testWords[i
+ 1], bigramCount);
            System.out.printf("Unigram count of '%s': %d%n", testWords[i], unigramCount);
            System.out.printf("Smoothed probability of ('%s', '%s'): %.4f%n", testWords[i],
testWords[i + 1], smoothedProbability);
```

```
            probability *= smoothedProbability;
        }
        System.out.printf("Final Probability: %.8f%n", probability);
    }
}
```

```
java -cp /tmp/FGVmLrtMF4/Exp2
Bigram count of ('They', 'play'): 1
Unigram count of 'They': 1
Smoothed probability of ('They', 'play'): 0.2000
Bigram count of ('play', 'in'): 1
Unigram count of 'play': 2
Smoothed probability of ('play', 'in'): 0.1818
Bigram count of ('in', 'a'): 0
Unigram count of 'in': 1
Smoothed probability of ('in', 'a'): 0.1000
Bigram count of ('a', 'big'): 1
Unigram count of 'a': 1
Smoothed probability of ('a', 'big'): 0.2000
Bigram count of ('big', 'garden'): 1
Unigram count of 'big': 1
Smoothed probability of ('big', 'garden'): 0.2000
Final Probability: 0.00014545

=== Code Execution Successful ===
```

```
# 4) Implement top-down and bottom-up parsing using python NLTK

import nltk
from nltk import CFG, ChartParser, RecursiveDescentParser,tree
grammar = CFG.fromstring("""
    S -> NP VP
    VP -> V NP | V NP PP
    PP -> P NP
    V -> "saw" | "ate" | "walked"
    NP -> "Rahil" | "Bob" | Det N | Det N PP | N
    Det -> "a" | "an" | "the" | "my" | "his"
    N -> "dog" | "cat" | "telescope" | "park" | "Moon" | "terrace"
    P -> "in" | "on" | "by" | "with" | "from"
""")

# Get user input for the sentence
sentence = input("Enter a sentence: ").split()

# Function to parse and display trees
def parse_and_display(parser, sentence, parse_type):
    print(f"{parse_type} Parsing:")
    trees = list(parser.parse(sentence))
    if not trees:
        print("No parse trees found.")
    else:
        for tree in trees:
            tree.pretty_print()
            tree.draw()

# Bottom-Up Parsing
parse_and_display(ChartParser(grammar), sentence, "Bottom-Up")
# Top-Down Parsing
parse_and_display(RecursiveDescentParser(grammar), sentence, "Top-Down")
```

```
Enter a sentence:  the cat saw the dog
Bottom-Up Parsing:
              S
      ________|___
     |            VP
     |         ___|___
     NP       |       NP
  ___|___     |    ___|___
Det      N    V  Det      N
 |       |    |   |       |
the     cat  saw the     dog

Top-Down Parsing:
              S
      ________|___
     |            VP
     |         ___|___
     NP       |       NP
  ___|___     |    ___|___
Det      N    V  Det      N
 |       |    |   |       |
the     cat  saw the     dog
```

5) Given the following short movie reviews, each labeled with a genre, either comedy or Action: a) fun, couple, love, love : comedy
b) fast, furious, shoot : action
c) couple, fly, fast, fun, fun :comedy
d) furious, shoot, shoot, fun :action
e) fly, fast, shoot, love :action
and a new document D: fast, couple, shoot, fly compute the most likely class for D. Assume a naive Bayes classifier and use add-1 smoothing for the likelihoods.

```
from sklearn.feature_extraction.text import CountVectorizer
from sklearn.naive_bayes import MultinomialNB
from sklearn.pipeline import make_pipeline
# Data
reviews = [
    ("fun, couple, love, love", "comedy"),
    ("fast, furious, shoot", "action"),
    ("couple, fly, fast, fun, fun", "comedy"),
    ("furious, shoot, shoot, fun", "action"),
    ("fly, fast, shoot, love", "action")
]
D = "fast, couple, shoot, fly"
# Prepare data
texts, labels = zip(*reviews)
# Create and fit the model
model = make_pipeline(CountVectorizer(tokenizer=lambda x: x.split(', ')), MultinomialNB())
model.fit(texts, labels)
# Predict the class and probabilities
predicted_class = model.predict([D])[0]
probabilities = model.predict_proba([D])[0]
# Output results
print(f"Predicted class for '{D}': {predicted_class}")
print(f"Class probabilities: {dict(zip(model.classes_, probabilities))}")
```

```
Predicted class for 'fast, couple, shoot, fly': action
Class probabilities: {'action': 0.7006979608594499, 'comedy': 0.2993020391405504}
/usr/local/lib/python3.10/dist-packages/sklearn/feature_extraction/text.py:528: UserWarning: The parameter 'token_pattern' will not be used since 'tokenizer' is not None'
  warnings.warn(
```

6) The dataset contains following 5 documents:

D1: "Shipment of gold damaged in a fire"

D2: "Delivery of silver arrived in a silver truck"

D3: "Shipment of gold arrived in a truck"

D4: “Purchased silver and gold arrived in a wooden truck”

D5: “The arrival of gold and silver shipment is delayed.”

Find the top two relevant documents for the query document with the content “gold silver truck " using the vector space model.

Use the following similarity measure and analyze the result:

a) Euclidean distance

b) Manhattan distance

c) Cosine similarity

```
import numpy as np
from sklearn.feature_extraction.text import CountVectorizer
from scipy.spatial.distance import euclidean, cityblock
from sklearn.metrics.pairwise import cosine_similarity
# Documents
docs = [
    "Shipment of gold damaged in a fire",
    "Delivery of silver arrived in a silver truck",
    "Shipment of gold arrived in a truck",
    "Purchased silver and gold arrived in a wooden truck",
    "The arrival of gold and silver shipment is delayed."
]
# User input for query
query = input("Enter a query: ")
# Vectorize
vec = CountVectorizer(stop_words="english")
X = vec.fit_transform(docs + [query]).toarray()
doc_vecs, qry_vec = X[:-1], X[-1]
# Compute distances and similarities
euclidean_dists = [euclidean(doc, qry_vec) for doc in doc_vecs]
manhattan_dists = [cityblock(doc, qry_vec) for doc in doc_vecs]
cosine_sims = cosine_similarity(doc_vecs, qry_vec.reshape(1, -1)).flatten()
# Rankings
top_2_euclidean = np.argsort(euclidean_dists)[:2] + 1
top_2_manhattan = np.argsort(manhattan_dists)[:2] + 1
top_2_cosine = np.argsort(-cosine_sims)[:2] + 1
print("Euclidean Distances:", euclidean_dists)
print("Manhattan Distances:", manhattan_dists)
print("Cosine Similarities:", cosine_sims)
print("\nTop 2 docs (Euclidean):", top_2_euclidean)
print("Top 2 docs (Manhattan):", top_2_manhattan)
print("Top 2 docs (Cosine):", top_2_cosine)
```

```
Enter a query: the arrival of gold
Euclidean Distances: [1.7320508075688772, 3.0, 2.0, 2.449489742783178, 1.7320508075688772]
Manhattan Distances: [3, 7, 4, 6, 3]
Cosine Similarities: [0.40824829 0.         0.35355339 0.28867513 0.63245553]

Top 2 docs (Euclidean): [1 5]
Top 2 docs (Manhattan): [1 5]
Top 2 docs (Cosine): [5 1]
```

8) Extract Synonyms and Antonyms for a given word using WordNet.

```
import nltk
from nltk.corpus import wordnet
nltk.download('wordnet')
def get_synonyms_antonyms(word):
    synonyms, antonyms = set(), set()
    for syn in wordnet.synsets(word):
        for lemma in syn.lemmas():
            synonyms.add(lemma.name())
            antonyms.update(ant.name() for ant in lemma.antonyms())
    return synonyms, antonyms
word = input("Enter a word: ")
synonyms, antonyms = get_synonyms_antonyms(word)
print(f"Synonyms: {', '.join(synonyms)}")
print(f"Antonyms: {', '.join(antonyms)}")
```

```
[nltk_data] Downloading package wordnet to /root/nltk_data...
[nltk_data]   Package wordnet is already up-to-date!
Enter a word: fat
Synonyms: fatty_tissue, fatten_up, fertile, avoirdupois, blubber, fatten_out, flesh_out, fat, plump, rich, adipose_tissue, fatten, fatty, fill_out, plump_out, productive, juicy, fatnes
Antonyms: nonfat, thin, leanness
```