Submission instruction:

Students will work on a programming assignment using an online system. The student will receive an invite to an online coding environment (Vocareum). Student should submit the homework using the online environment ONLY. Submissions of other kind will not be accepted. Late submission date and time are taken seriously. Be aware of your clock! Do not submit after the final deadline. Failure to follow instructions will result in penalties.

Due Date: March 23rd

- Late homework: you lose 20% of the homework's grade per 24-hour period that you are late. Beware, the penalty grows very fast: grade = points * (1 – n * 0.2) where n is the number of days late (n=0 if submitted on time, n=1 is submitted between 1 second and 24h late, etc).
- Grade review/adjustment: Requests will be considered up to 2 weeks after the grade is released. After that, it will be too late and requests for grading review will be denied.
- Homework assignments are to be solved individually.
- You are welcome to discuss class material in review groups, but do not discuss how to solve the homework.

Grading Policy

- Your source code must be written in Python.
- Your code must compile and execute on Vocareum with python 2.7.
- Your programs will be tested on test cases which would be different from the ones provided in input files.
- The source code must be properly commented.
- Your program should run in a reasonable time.
- You will use Vocareum.com to submit your code. Please check "Student Help.pdf" in the Homework-2 folder for instructions on how to use the system.

1. Overview:

A fundamental obstacle in logic-based AI is the difficulty of solving the satisfiability problem (usually referred to as SAT). SAT problems are statements in propositional logic, written in conjunctive normal form (CNF). CNF sentences consist of *variables*, *literals*, and *clauses*. A variable (A, for example) can be assigned true or false, a literal is a variable or its negation (A or NOT A). In a CNF sentence, a clause consists of a disjunction of literals (literals joined together by OR operation). A sentence is a conjunction of clauses (clauses joined together by AND operation).

Sentence Example: (A ∨ ¬B) ^ (B ∨ C ∨ D) ^ (¬A ∨ C)

The SAT problem asks if any assignment of truth values to variables exists that makes a CNF logical sentence true. SAT problems are very difficult to solve. SAT is NP-complete, and therefore no known algorithm can solve it in less than exponential amount of time. If a sentence has n variables, it will take $O(2^n)$ operations to discover an assignment in the worst case – equivalent to checking every possible assignment.

2. Representing Propositional Sentences:

The standard notation for propositional logic is not the easiest for computers to process. Sure, we can find unicode characters for all of special symbols, but we probably do not want to operate directly on unicode-encoded strings for automated reasoning. Instead, we can represent sentences in propositional logic using nested lists.

Traditional notation:

¬R ^ B ⇒ W

List representation (Python-style):

["implies", ["and", ["not", "R"], "B"], "W"]

This is convenient, especially if your program is reading strings from stdin or from a file. In Python, you can convert a string that is formatted to look like a list into an actual list by calling the *eval* function on it.

```
>>> mystring = '["implies", ["and", ["not", "R"], "B"], "W"]'
>>> mylist = eval(mystring)
>>> mylist
['implies', ['and', ['not', 'R'], 'B'], 'W']
>>> len(mystring)
44
>>> len(mylist)
3
```

A variable can be represented as a string.

"R"

A sentence is a list, where the element in the first position is string that denotes a logical connective, and all remaining elements are either variables or sentences.

["not", "R"]

["and", ["or", "P", "Q", "R"], ["not", "S"]]

["iff", "S", ["and", "Q", "R"]]

In this format, we write propositional variables as strings beginning with an uppercase letter, and connectives in lowercase. There are only five connectives, so the first element of EVERY list must be one of the following strings:

"not" negation
"and" conjunction
"or" disjunction
"implies" implication
"iff" biconditional implication

3. Problem-1: CNF-converter [50 Points]

To create a SAT solver, your input has to be in CNF. In this problem, you have to create a program, *CNFconverter.py*, which would convert any propositional logic sentence into its equivalent CNF sentence.

Input & Output format:

The input to the program would be a file consisting propositional sentences represented in lists as explained in Section 2. Input file "sentences.txt" is provided. The number *n* in the first line indicates the number of input propositional sentences in the file. The next *n* lines contain one propositional sentence per line. You can assume that the variables are one character strings i.e. "A", "B"..."Z".

sentences.txt

```
5
["implies", ["and", ["not", "R"], "B"], "W"]
["not", ["implies", ["implies", ["or", "P", ["not", "Q"]], "R"], ["and", "P", "R"]]]
["or", ["not", ["implies", "P", "Q"]], ["implies", "R", "P"]]
["not", ["not", ["not", ["not", ["not", "P"]]]]]
["or", ["and", "A", "B"], ["and", "C", "D"]]
```

Figure 1: Input format for Problem-1

The command to run your program would be in the following format:

python CNFconverter.py –i inputfilename

The output of the program would be the CNF sentences equivalent to the input propositional sentence. The output should be in the same format as described in Section 2, i.e. the list representation. The output file must be named "sentences_CNF.txt" containing the *n* lines where each line contains the equivalent CNF sentence for each input propositional sentence provided in the input file. Make sure that the output of your program can be converted into a nested list using the *eval* function as described in Section 2.

sentences_CNF.txt

```
1  ["or", "R", ["not", "B"], "W"]
2  ["and", ["or", ["not", "P"], "R"], ["or", "Q", "R"], ["or", ["not", "P"], ["not", "R"]]]
3  ["and", ["or", "P", ["not", "R"]], ["or", ["not", "Q"], ["not", "R"], "P"]]
4  ["not", "P"]
5  ["and", ["or", "C", "A"], ["or", "D", "A"], ["or", "C", "B"], ["or", "D", "B"]]
6
```

Figure 2: Output format for Problem-1

Notes:

- Order of the literals does not matter. For example, ["and", "A", "B"] and ["and", "B", "A"] are equal and you can output any of them.
- Eliminate all the duplicates from conjunctions and disjunctions. ["and", "A", "A"] is wrong and should be replaced with "A". Note that this also extends to duplicate clauses. For example, ["and", "A", ["or", "C", "D"], ["or", "D", "C"]] should be replaced with ["and", "A", ["or", "C", "D"]].
- Some inputs might lead to something like ["and", "A", ["not", "A"], ...]. If you have something like this in your output, that is fine. You don't have to remove it as long as the output is logically correct and in a CNF format.
- Assume that the inputs are always logically correct, i.e. "iff" and "implies" always have 2 parameters, "not" has 1 parameter, "and" and "or" have 2 or more parameters.

4. Problem-2: SAT-solver [50 Points]

In this problem you need to build a boolean satisfiability solver that takes a set of variables and connectives in CNF and returns either a satisfying assignment that would make the CNF sentence true or determines that no satisfying assignment is possible. In particular, we are asking you to create a program, *DPLL.py* , which would implement the DPLL algorithm.

Input & Output format:

The input to the program would be a file consisting propositional sentences in CNF format represented in lists as explained in Section 2. Input file "CNF_sentences.txt" is provided. The number *n* in the first line indicates the number of input propositional sentences in the file. The next *n* lines contain one propositional sentence in CNF format per line. You can assume that the variables are one character strings i.e. "A", "B"..."Z".

CNF_sentences.txt

```
1  5
2  ["or", "R", ["not", "B"], "W"]
3  ["and", "A", ["not", "A"]]
4  ["not", "P"]
5  ["and", ["or", "P", ["not", "R"]], ["or", ["not", "Q"], ["not", "R"], "P"]]
6  ["and", "A", ["or", "B", "C"], ["or", "B", "D"]]
7
```

Figure 3: Input format for Problem-2

The command to run your program would be in the following format:

python DPLL.py –i inputfilename

The output of the program should be a file, named "CNF_satisfiability.txt", containing *n* lines each containing a python list, where *n* is the number of

CNF sentences in the input file. The first element in the python list should be either *true* or *false* indicating the boolean satisfiability of that particular input sentence. If the first element is *true* then the list must contain *m* additional elements, where *m* is the number of variables in the input sentence. For each variable, the corresponding element in the list should indicate whether that variable is *true* or *false* to satisfy the boolean satisfiability of the input sentence as shown in the output format in Figure 4.

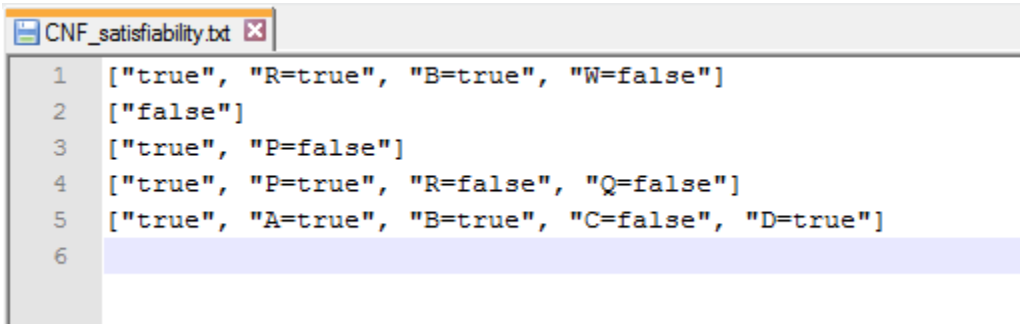

Figure 4: Output format for Problem-2

Notes:

- Order of variables in your output does not matter.
- There can be many possible boolean assignments to the variables to satisfy the boolean satisfiability of a CNF sentence. Your output can be anyone of those possibilities. For example, for input sentence ["or", "R", ["not", "B"], "W"]; ["true", "R=false", "B=true", "W=true"] is also correct.