Report

General

To run any python file, make an appropriate input file and follow the following command.

```
python3 q<no>.py <input_json> <output_json>
```

Question 1: Regex to NFA

Step 0

Add concatenation operator (`.`) to RE. We will add a `.` between *c1* and *c2* when the following condition is met.

```
def check_concat(self, c1: str, c2: str):
        if c1.isalnum() or c1 == ')' or c1 == '*':
            if c2.isalnum() or c2 == '(':
                return True
        return False
```

Step 1

Convert Infix RE to Postfix RE as that will be easier to solve. The following is the code for the same:

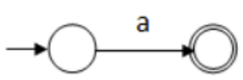
```
def infix_to_posfix_re(self, re: str):
        postfix_re = []
        stack = []
        for c in re:
            if c.isalnum():
                postfix_re.append(c)
            elif c == '(':
                stack.append(c)
            elif c == ')':
                while stack[-1] != '(':
                    postfix_re.append(stack.pop())
                stack.pop() # '('
            else:
                while len(stack) > 0 and stack[-1] != '(' and\
                                precedenceOf(stack[-1]) >= precedenceOf(c):
                    postfix_re.append(stack.pop())
                stack.append(c)

        while len(stack) > 0:
            postfix_re.append(stack.pop())
        return ''.join(postfix_re)
```

Step 2

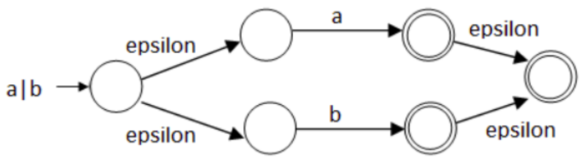
Convert the postfix regular expression into an NFA using a stack, where each element on the stack is a NFA. The input expression is scanned from left to right.

When this process is completed, the stack contains exactly one NFA. The constructed NFA's is based on the inductive rules below.

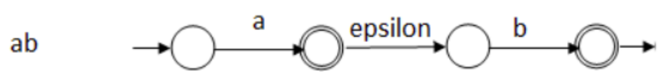
- **Rule 1:** Make NFA that accepts a single character. Add two states and label one of them as start and other as finish. Draw an arrow from start state to finish with given character as edge re.



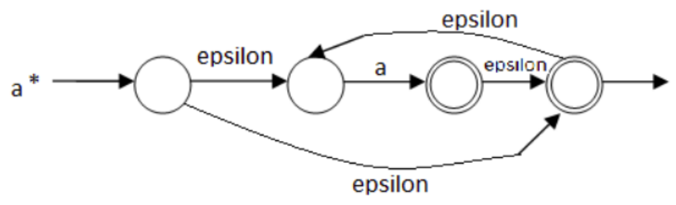
- **Rule 2:** Given NFA1 that accepts regular expression r1 and NFA2 that accepts regular expression r2 then make NFA3 that accepts r1 + r2. Add a new start state s and make a ε-transition from this state to the start states of NFA1 and NFA2. Add a new final state f and make a ε-transition to this state from each of the final states of NFA1 and NFA2.



- **Rule 3:** Given NFA1 that accepts regular expression r1 and NFA2 that accepts regular expression r2 then make NFA3 that accepts r1∘ r2. Add a ε-transition from the final state of r1 to the start state of r2. The start state of NFA3 is the start state of NFA1 and the final state of NFA3 is the final state of NFA2. You will have to think about it but I do not think you will have multiple final states in NFA1.



- **Rule 4:** Given NFA1 that accepts regular expression r then make a NFA2 that accepts r*. Add a new start state s and make a ε-transition from this state to the start state of NFA1. Make a ε-transition from the final state of F1 to the new start state s. The final states of NFA1 are no longer final and s is the final state of NFA2.



The following is a rough code for the same

```
while (not end of postfix expression) {
    c = next character in postfix expression;
    if (c == ’.’) {
        nFA2 = pop();
        nFA1 = pop();
        push(NFA that accepts the concatenation of L(nFA1) followed by L(nFA2));
    } else if (c == ’+’) {
        nFA2 = pop();
        nFA1 = pop();
        push(NFA that accepts L(nFA1) | L(nFA2));
    } else if (c == ’*’) {
        nFA = pop();
        push(NFA that accepts L(nFA) star);
    } else{
        push(NFA that accepts a single character c);
    }
}
```

Question 2: NFA to DFA

Algorithm

Given an NFA $N = (Q, \Sigma, \delta, q_0, F)$, which recognises a language *A*, an equivalent DFA *M* can be defined as: $M = (Q', \Sigma, \delta', q_0', F')$
where,

$$Q' = \mathcal{P}(Q)$$

$$\delta'(R, a) = \bigcup_{r \in R} \delta(r, a)$$

$$q_0' = \{q_0\}$$

$$F' = \{R \in Q' | R \cap F \neq \phi\}$$

Implementation

Initialise *states*, *letters* and *start_states* of the DFA from NFA.

```
def initialize_from_nfa(self, nfa: NFA):
        # 2**len states of dfa
        # state(i) of dfa: List of states whose position is a set bit in i.
        self.states = []
        for idx in range(2**len(nfa.states)):
            curr_state = []
            for pos in range(len(nfa.states)):
                if (idx >> pos) & 1:
                    curr_state.append(nfa.states[pos])
            self.states.append(curr_state)
        # same as nfa
        self.letters = nfa.letters
        # start_state: state(i) in dfa where set bit in i are nfa start_states.
        self.start_states = [nfa.start_states]
```

make the new transition function for dfa

```
dfa.transition_function = []
for from_states in dfa.states:
    for ch in dfa.letters:
        to_states = set()
        for from_state in from_states:
            if (from_state, ch) in nfa.move:
                to_states = to_states | nfa.move[(from_state, ch)]
         dfa.transition_function.append([from_states, ch, list(to_states)])
```

make the new final_states list for dfa

```
dfa.final_states = []
for curr_states in dfa.states:
    for state in curr_states:
        if state in nfa.final_states:
            dfa.final_states.append(curr_states)
            break
```

Question 3: DFA to Regex

We can do this in two steps:

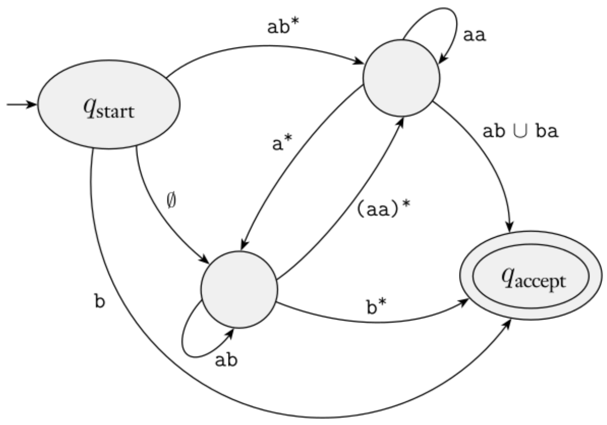
1. Converting DFA to *generalized nondeterministic finite automaton*, GNFA.
2. Reducing States of GNFA

Step 1: DFA ⇒ GNFA

GNFAs always have a special form that meets the following conditions.
• The start state has transition arrows going to every other state but no arrows coming in from any other state.

• There is only a single accept state, and it has arrows coming in from every other state but no arrows going to any other state. Furthermore, the accept state is not the same as the start state.
• Except for the start and accept states, one arrow goes from every state to every other state and also from each state to itself. The regex on arrows can be ϕ if there is no transition between the states.

Example:



Code for the same:

```
def make_gnfa(self):
        self.trans = {}
        # initliaze with phi
        for i in self.states:
            self.trans[i] = {}
            for j in self.states:
                self.trans[i][j] = 'ϕ'
        # add re to edge
        for _from, ch, to in self.transition_function:
            if self.trans[_from][to] == 'ϕ':
                self.trans[_from][to] = ch
            else:
                self.trans[_from][to] += f'+{ch}'

        self.gnfa_start = 'Q#' + str(len(self.states))
        self.gnfa_final = 'Q#' + str(len(self.states) + 1)
        self.trans[self.gnfa_start] = {}
        self.trans[self.gnfa_final] = {}
        self.trans[self.gnfa_start][self.gnfa_final] = 'ϕ'
        for state in self.states:
            if state in self.start_states:
                self.trans[self.gnfa_start][state] = '$'
            else:
                self.trans[self.gnfa_start][state] = 'ϕ'

            if state in self.final_states:
                self.trans[state][self.gnfa_final] = '$'
            else:
                self.trans[state][self.gnfa_final] = 'ϕ'
```

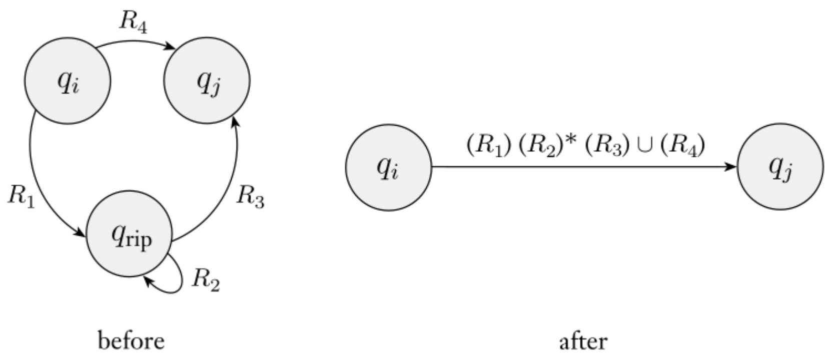
Step 2: Reducing states of GNFA

Remove all the states except the start and finish state. Now, the RE on the arrow from start to finish is the resultant RE for this DFA.

Lets say we are removing q_{rip}. We do so by the following approach:

1. Find the list of predecessors and successors of *rip*.
2. For each (i, j) pair where i is a predecessor and j is a successor, update arrow between them by the following rules.
    1. q_i goes to q_{rip} with an arrow labeled R_1,
    2. q_{rip} goes to q_{rip} with an arrow labeled R_2,
    3. q_{rip} goes to q_j with an arrow labeled R_3,
    4. q_i goes to q_j with an arrow labeled R_4,

    then in the new machine, the arrow from q_i to q_j gets the label $(R_1)(R_2)*(R_3) + (R_4)$



Question 4: DFA Minimization

I did it using Myphill-Nerode algorithm. The working of the algorithm is as follows:

1. Remove all the unreachable states by doing a BFS/DFS.
2. Draw a table for all pairs of states (Q_i, Q_j) not necessarily connected directly [All are unmarked initially]
3. Consider every state pair (Q_i, Q_j) in the DFA where Q_i ∈ F and Q_j ∉ F or vice versa and mark them. [Here F is the set of final states]
4. If there is an unmarked pair (Q_i, Q_j), mark it if the pair {δ(Q_i, A), δ(Q_j, A)} is marked for some input alphabet. Repeat this step until we cannot mark anymore states.

5. Combine all the unmarked pair (Q_i, Q_j) and make them a single state in the reduced DFA.