Homework 5

CS/ECE 374B

Due 8 p.m. on Tuesday, October 15

- Note that backtracking algorithms are not in general efficient, especially in the worst-case. Your goal in this homework is to come up with a correct algorithm, not an efficient one.
- Whenever we ask for runtime complexity, we are looking for the asymptotic worst-case complexity. For full credit you should give a tight bound ($\Theta(\cdot)$ instead of $O(\cdot)$)
- Algorithms may be written in pseudocode or Python. In either case, your goal is to communicate to the grader how your algorithm works; hard to follow or messy algorithms will get less credit. Explain any complex steps in your algorithm.

1. Regular expressions are frequently implemented using backtracking. In this question, you will be given a parsed regular expression, represented as a collection of nested tuples. Below is the format of the parsed regular expression, shown in both mathematical notation for pseudocode and in Python code:

Regex	**Tuple**	**Python**
$\emptyset$	$(\emptyset)$	`(None)`
ϵ	(ϵ)	`('')`
$a(\in \Sigma)$	(a)	`('a')`
$r_1 r_2$	$(\cdot, r_1, r_2)$	`('.',r1,r2)`
$r_1 + r_2$	$(+, r_1, r_2)$	`('+',r1,r2)`
$r1^*$	$(*, r_1)$	`('*',r1)`

For example, the regex $(0+\epsilon)(11^*0)^*1^*$ would be represented as

$$(\cdot, (+, (0), (\epsilon)), (\cdot, (*, (\cdot, (1), (\cdot, (*, (1)), (0)))), (*, 1)))$$

or:

```
('.',
  ('+',
    ('0'),
    ('')),
  ('.',
    ('*',
      ('.',
        ('1'),
        ('.',
          ('*',('1')),
          ('0')))),
    ('*','1')))
```

(4) (a) Describe a backtracking algorithm that decides whether an input string matches a parsed regular expression. **Do not** implement any sort of finite automaton!

(3) (b) Calculate the runtime complexity of your algorithm in terms of the length of the input n and the length of the regular expression m. Justify your answer.

(3) (c) For the fixed regular expression example above, calculate the runtime complexity of your algorithm in terms of the length of the input n.

2. Another place where backtracking algorithms are implemented is in parsing grammars.

(5) (a) Implement a backtracking algorithm to see if an input string can be generated by a grammar written in Chmosky Normal Form. Recall that in a CNF grammar, all productions have one of the following forms:

$$X \to YZ$$
$$X \to a$$
$$S \to \epsilon$$

where X, Y, Z are non-terminals, a is a terminal, and S is the start non-terminal.

(5) (b) Calculate the runtime complexity of your parser in terms of the length of the input n and the size of the grammar m

(0) (c) ~~Modify your algorithm (if necessary) to work with a generic context-free grammar and calculate the runtime complexity of your modified algorithm. Note: you may just do this part and skip the previous ones, but the earlier parts are a little simpler.~~

3. On the last homework we used GCD to find weak cryptographic keys. We will now consider how the GCD is computed.

(3) (a) A common algorithm, due to Euclid, is implemented as follows:

```
def euclid_gcd(x, y):
    if x == y:
        return x
    elif x > y:
        return euclid_gcd(x-y,y)
    else:
        return euclid_gcd(x,y-x)
```

Analyze the runtime complexity of this algorithm in terms of x and y. Assume that subtraction takes $\Theta(\log x + \log y)$ time

(4) (b) What is commonly used today is an improvement over the algorithm that uses the mod operator:

```
def mod_gcd(x,y):
    if y == 0:
        return x
    elif x > y:
        return mod_gcd(y, x % y)
    else:
        return mod_gcd(x, y % x)
```

Analyze the runtime complexity of this algorithm, given that computing x%y takes $\Theta(\log x \log y)$ time.

(3) (c) Here is another variation:

```
def binary_gcd(x,y):
    if x == y:
        return x
    evenx = (x % 2 == 0)
    eveny = (y % 2 == 0)
    if evenx and eveny:
        # a//b forces integer division
        return 2*binary_gcd(x//2, y//2)
    elif evenx:
```

```
        return binary_gcd(x//2,y)
    elif eveny:
        return binary_gcd(x,y//2)
    elif x > y:
        return binary_gcd((x-y)//2,y)
    else:
        return binary_gcd(x,(y-x)//2)
```

Analyze the runtime complexity of this algorithm, given that computing x−y takes $\Theta(\log x + \log y)$ time, computing x%2 takes constant time (why?), and computing x//2 takes $\Theta(\log x)$ time.

Solved Question

(5) 4. (a) In this question you should implement a backtracking algorithm to see if a string x is accepted by an NFA N. Your NFA will be given to you as input in the following form:

- δ is a dictionary mapping the current state and an input character to a set of states
- The starting state and the set of accepting states

Assume that the NFA has no ϵ-transitions.

Solution:

```
def nfa_accepts(x, delta, s, A, n=0):
    """ returns True if NFA represented by 'delta', started in state 's',
    ends in an accepting state (in 'A') on input 'x[n:]'.

    Call with 's' being the starting state of the NFA and 'n=0'. """
    # base case
    if n == len(x):
        # Accept iff s is an accepting state
        return (s in A)
    # iterate over all non-deterministic transitions from s on x[n]
    for next_state in delta[(s,x[n])]:
        if nfa_accepts(x, delta, next_state, A, n+1):
            return True
    return False
```

■

Rubric:

- +1 for correct base case(s)
- +1 for enumerating all potential next 'moves'
- +1 for checking constraint on moves (implicit in this solution)
- +1 for stopping iteration once a solution is found
- +1 for correct recursive call
- -1 for other mistakes

(b) Analyze the runtime of your algorithm in terms of n, the length of the input string, and m, the size of your NFA.

Solution: When we call nfa_accepts it makes at most $|Q|$ recursive calls (where Q is the set of states in the NFA), each of which reduces the length of the input to be seen by 1, giving the maximum recursion depth of n. Assuming s **in** A runs in constant time, the algorithm is $O(|Q|^n)$.

The worst-case occurs when $|\delta(q,c)| = |Q|$, i.e., $\delta(q,c) = Q$ for all q, and $A = \emptyset$, ensuring all $|Q|^n$ possibilities get explored. This NFA takes $\Theta(|Q|^2)$ space to represent in the usual set notation, so $|Q| = \Theta(m^{\frac{1}{2}})$, giving us a runtime of $\Theta(m^{\frac{n}{2}})$. ■

Rubric:

- +2 for finding upper bound on execution
- +2 for finding lower bound
- -1 for either of the above bounds not being tight
- +1 for considering space to represent worst-case example