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!

Solution: The pseudocode for the backtracking algorithms

```
1:  function MATCH(s, r)
2:      if r[0] = None then
3:          return False
4:      else if r[0] = ' ' then
5:          return s == ε
6:      else if r[0] = a(∈ Σ) then
7:          return s == a
8:      n = len(s)
9:      if r[0] = "·" then
10:         for i = 0 → n do
11:             if MATCH(s[0 : i], r[1]) and MATCH(s[i : n], r[2]) then
12:                 return True
13:             else
14:                 return False
15:     else if r[0] = "+" then
16:         return MATCH(s, r[1]) or MATCH(s, r[2])
17:     else if r[0] = "∗" then
18:         if s = ε then
19:             return True
20:         for i = 1 → n do
21:             if MATCH(s[0 : i], r[1]) and MATCH(s[i : n], r) then
22:                 return True
23:             else
24:                 return False
```

■

(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.

Solution: Consider a operating on an n-character string. Let $T^+(n)$, $T^\cdot(n)$, and $T^*(n)$ be the time it takes to match a regular expression that has a union, concatenation, and Kleene star as the top-level operator. We can express these in terms of $T_1(n)$ and $T_2(n)$, which we will use to represent the time it takes to match r_1 and r_2 (in case of union and concatenation):

$$T^+(n) \leq T_1(n) + T_2(n)$$

$$T^\cdot(n) \leq \sum_{i=0}^{n} T_1(i) + T_2(n-i)$$

$$T^*(n) \leq \sum_{i=1}^{n} T_1(i) + T^*(n-i)$$

First, observe that these are inequalities. The backtracking algorithm will stop if $T_1(i)$ is not (star or concatenation) / is (union) a match. Second, if these were equalities, then Kleene star has the potential for exponential growth:

$$T^*(n) - T^*(n-1)T_1(n) + T^*(n-1)$$

$$T^*(n) = 2T^*(n-1) + T_1(n)$$

To construct worst-case behavior, we therefore need to create a Kleene star such that the match to the first part of the string always succeeds and the match to the last part always fails. One such expression is $(0^*)^*$, applied to the string $0^{n-1}1$, producing $\Theta(2^n)$ exponential behavior.

You can create a slightly longer evaluation time by creating, e.g., $0^*(0^*)^*$. Then we will have:

$$T(n) = \sum_{i=0}^{n} T_{0^*}(i) + T_{(0^*)^*}(n-i)$$

Note that the sum is dominanted by the second component. Since $\sum_{i=0}^{n} 2^n = 2^{n+1} - 1$ this does not change the exponential bound, but repeating this process m times will results in $\Theta(2^{n+m})$ execution time. ■

(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.

Solution: The expression $(0+\epsilon)(11^*0)^*1^*$ has three concatenation groups. Note that the first group only adds $O(n)$ time since it will fail to match any string longer than length 1 and therefore not try to match the remaining string.

To ensure that the second concatenation group $(11^*0)^*$ gets called as many times as possible, we can make the third group 1^* not match by, e.g., using the input string $1^{n-2}00$. Then the algorithm will try to split the string into n possible prefixes, and try to match $(11^*0)^*$ to every prefix. To do that, in turn, it will compute n prefixes again and try to match them against 11^*0 (taking linear time), failing except for the last iteration.

So the entire match process will take $\Theta(n^2)$ ■

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.

Solution:

```
# example grammar
productions = [ ('X', ['Y','Z']), ('X', ['a']), ('S', []) ]

def matches(s, G, n):
    """ returns true if string `s` matches nonterminal `n` in the grammar `G` """
    # iterate over all productions
    for lhs, rhs in G:
        # only consider the productions with n as the LHS
        if lhs == n:
            if len(rhs) == 0:
                # epsilon production
                if len(s) == 0:
                    return True
            elif len(rhs) == 1:
                # production deriving terminal
                if len(s) == 1 and s[0] == rhs[0]:
                    return True
            else:
                # split the string into two
                # non-empty parts and try to match them
                # to the two non-terminals in the RHS
                for i in range(1,len(s)):
                    if matches(s[:i], G, rhs[0]) and
                        matches(s[i:], G, rhs[1]):
                        return True
    # no productions match
    return False
```

■

(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

Solution: First consider a grammar $S \rightarrow SS|0$. If we are trying to match the string $0^{n-1}1$, it is easy to see that the time taken will satisfy the recurrence:

$$T(n) = \sum_{i=1}^{n} T(n-i) + O(n)$$

following an analysis similar to the previous question, giving us a runtime of $\Theta(2^n)$

We can further increase the runtime by using the grammar $S \rightarrow SS|AA|0$, $A \rightarrow AA|SS|0$. Then we will have

$$T(n) = 2\sum_{i=1}^{n} T(n-i) + O(n)$$

resulting in $\Theta(3^n)$ runtime. We can keep doing this by adding yet more symbols; note that this will make the length of the grammar grow quadratically, however. If we state that a grammar that follows this pattern with k different non-terminals will take αk^2 space, the runtime complexity will be $\Theta((m/\alpha)^{n/2})$ time.

(note that the dynamic programming version of this algorithm, also known as the CYK algorithm, runs in $\Theta(n^3)$ time.) ■

(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.

Solution: When we call euclid_gcd function, it makes at most one recursive call, each of which reduce the length of input by at least 1. So the depth of the recursive tree is at most max(x, y) = $\Theta(x+y)$. Since each recursion takes $\Theta(logx + logy)$ time to do the subtraction, the total run time for the algorithm is $O((x+y)(logx + logy))$.

The worst case occurs when $y = 1$ or $x = 1$, then the function will call itself x or y times. And since each recursion takes $\Theta(logx + logy)$ time to do the subtraction, the total run time for this case is $(x+y)(logx + logy)$.

We now know that all problems take at most $(x+y)(logx + logy)$ (Upper bound), and some problem takes at least $(x+y)(logx + logy)$ (Lower bound). We can conclude that there's a tight bound $\Theta((x+y)(logx + logy))$.

■

(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.

Solution: We want to calculate the maximum depth of the recursion tree, in order to calculate the upper bound. And we can prove that the depth is bounded by $log(x+y)$ by the following.

Denote (x_i, y_i) as the value of x and y for which the above algorithm performs i steps (the depth of the recursion tree is i). Assume $x_i \geq y_i$, we can show that

1. for one step, $y_1 = 0$,
2. for two steps, $y_2 \geq 1$,
3. for more steps, assume we have three consecutive steps (x_{k+1}, y_{k+1}), (x_k, y_k), (x_{k-1}, y_{k-1}), then $x_k = y_{k+1}$, $x_{k-1} = y_k$, $y_{k-1} = x_k mod y_k$, so $x_k = q * y_k + y_{k-1} = y_{k+1}$ for some $q \geq 1$. Thus, we can conclude that $y_{k+1} \geq y_k + y_{k-1}$.

We can see that y_{k+1} should be larger than the $Fibonacci_k$. The closed form of a Fibonacci number $f_i = \frac{1}{\sqrt{5}}((\frac{1+\sqrt{5}}{2})^i - (\frac{1-\sqrt{5}}{2})^i)$. Take the log on both sides we can get, $i \approx log(f_i)$, which means $i \leq log(y)$, so depth of the recursion tree is bounded by $log(y)$ if $x \geq y$, and bounded by $log(x)$ if $y \geq x$. So in general, it is bounded by $log(x+y)$. Since computing x mod y takes $\Theta(\log x \log y)$ time, the total run time for the algorithm is $O(log(x+y)(logxlogy))$.

The worst case would be x and y are continuous Fibonacci number f_i, f_{i-1}. Assume $x > y$, then x mod $y = f_{i-2}$. For each recursion call, $x_i \% y_i$ would be the smaller Fibonacci number. So we can conclude that the depth of the recursion tree is i, which is $log(x+y)$. Since computing $x\%y$ takes $\Theta(\log x \log y)$ time, the total run time for the worst case is $O(log(x+y)(logxlogy))$.

We now know that all problems take at most $log(x+y)(logxlogy)$ (Upper bound), and some problem takes at least $log(x+y)(logxlogy)$ (Lower bound). We can conclude that there's a tight bound $\Theta(log(x+y)(logxlogy))$.

■

(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.

Solution: We want to calculate the maximum depth of the recursion tree, in order to calculate the upper bound. And we can prove that the depth is bounded by log(xy) = logx + logy by the following.

Denote (x_i, y_i) as the value of x and y for which the above algorithm performs i steps (the depth of the recursion tree is i). We can show that $x_{i+1}y_{i+1} \geq 2x_i y_i$ since in each recursive call, at least one of the parameter is halved. Thus, the height of the recursion tree is bounded by O(log(xy)) = O(logx + logy). The runtime for each reversive call is also O(logx + logy) since:

1. both x and y are even, x//2 takes Θ(logx) and y//2 takes Θ(logy), so binary_gcd(x//2, y//2) takes Θ(logx + logy) time.
2. x or y is even, it takes Θ(logx + logy) time.
3. both x and y are odd, the minus operation takes Θ(logx + logy) time, and // operation takes less time than that. So the operation time is Θ(logx + logy).

The total run time for the algorithm is $O(\log(xy)(\log x + \log y)) = O(((\log x + \log y)^2)$.

Since all numbers would lead to $O((\log x + \log y)^2)$ runtime, there must be a worst case which lead to this runtime.

We now know that all problems take at most $(\log x + \log y)^2$ (Upper bound), and some problem takes at least $(\log x + \log y)^2$ (Lower bound). We can conclude that there's a tight bound $\Theta((log\,x + log\,y)^2)$.

■

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