CS/ECE 374 B Fall 2019
**Homework 5 Problem 1**

Siyuan Chen (siyuanc2)
Megan Walsh (meganew2)
Jiashuo Tong (jtong8)

**Solution:**

(a) Describe a backtracking algorithm that decides whether an input string matches a parsed regular expression.

```python
# hw5p1a.py
def q1(curr_str, curr_regex):
    currstrlen = len(curr_str)
    if curr_regex == (''):
        # Match epsilon, O(1)
        if curr_str == '':
            return True
        return False
    elif curr_regex == (None):
        # Match nothing, O(1)
        return False
    elif curr_regex[0] == '.':
        # Concatenation. The algorithm tries to find a split point i such that curr_str[0:i] is accepted by LHS of regex and curr_str[i:] is accepted by RHS. Maximum n calls.
        for i in range(currstrlen+1):
            if q1(curr_str[0:i], curr_regex[1]) and q1(curr_str[i:], curr_regex[2]):
                return True
        return False
    elif curr_regex[0] == '+':
        # Union. The algorithm evaluates whether the current string is accepted by LHS. If false, continue evaluating RHS and return false if both LHS and RHS returns false.
        if q1(curr_str, curr_regex[1]) or q1(curr_str, curr_regex[2]):
            return True
        return False
    elif curr_regex[0] == '*':
        # Kleene star. The algorithm matches a longest possible portion of input string with the starred expression.
        if curr_str == '':
            return True
        else:
            for i in range(1, currstrlen+1):
                if q1(curr_str[currstrlen-i:], curr_regex[1]):
                    if q1(curr_str[:currstrlen-i], curr_regex):
                        return True
        return False
    elif (curr_regex[0] == '0') or (curr_regex[0] == '1'):
        # Matching single characters, O(1)
        if curr_str == curr_regex[0]:
            return True
        return False
    else:
```

```
        return False

if __name__ == "__main__":
    # Define initial condition of the 3 poles and disks
    regex = ('.',('+', ('0'), ('')), ('.', ('*', ('.', ('1'), ('.',
        ('*', ('1')), ('0')))), ('*', '1')))
    assert q1('', regex) == True
    assert q1('0', regex) == True
    assert q1('01', regex) == True
    assert q1('011011101', regex) == True
    assert q1('1', regex) == True
    assert q1('2', regex) == False
    assert q1('00', regex) == False
```

(b) Three recurrence relations govern the behavior of $+$, $\cdot$, and $*$ operations. For all other cases, The algorithm runs in constant time.

Union($+$):

$$\begin{aligned}
T_+(n) &\le \sum_{k=1}^{n}(T_+(k-1)+T_+(n-k))+O(1) \\
&= 2\sum_{k=0}^{n-1}T_+(k)+O(1) \\
T_+(n-1) &= 2\sum_{k=0}^{n-2}T_+(k)+O(1) \\
T_+(n)-T_+(n-1) &= 2T_+(n-1)+O(1) \\
T_+(n) &= 3T_+(n-1)+O(1)
\end{aligned}$$

Concatenation($\cdot$):

$$T_\cdot(n) \le 2T_\cdot(n)+O(1)$$

Kleene star($*$):

$$\begin{aligned}
T_*(n) &\le \sum_{k=1}^{n}(T_*(k-1)+T_*(n-k))+O(1) \\
&= 2\sum_{k=0}^{n-1}T_*(k)+O(1) \\
T_*(n-1) &= 2\sum_{k=0}^{n-2}T_*(k)+O(1) \\
T_*(n)-T_*(n-1) &= 2T_*(n-1)+O(1) \\
T_*(n) &= 3T_*(n-1)+O(1)
\end{aligned}$$

One can easily see that $T_+(n) = O(3^n)$ and $T_*(n) = O(3^n)$. Additionally, each $+$, $\cdot$, $*$ operations encountered reduces the remaining unprocessed length of the regular expression by 1.

The absolute worst case runtime occurs when the regular expression consists exclusively a series of nested Kleene stars, i.e. ((((((((0)*)*)*)*)*)*)*)*. Under this case, when the algorithm is given a string that is not accepted by the regex, i.e. 1111, each * operation will (after exploring all other available options) accept nothing and pass the entire string to its successor. Namely:

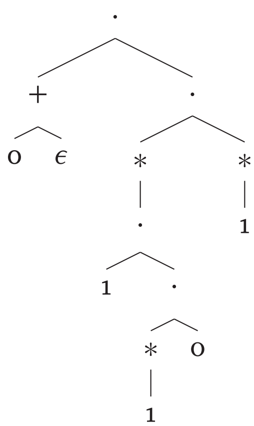
$$T(n,m) = T(n,m-1) + T_*(n)$$

Where $T_*(n) = O(3^n)$:

$$T(n,m) = T(n,m-1) + O(3^n)$$

Solving this recurrence then indicates that $T(n,m) = \Theta(m \cdot 3^n)$.

(c) The regular expression is parsed as follows:




One can easily see that the longest path from root to leaf consists of 6 $*$ or $\cdot$ operations. Therefore, in the worst case scenario where a string of length n, not acceptable by the regular expression, is fed to the algorithm, $T(n,m) = \Theta(m \cdot 3^n)$ where $n = 6$, or $T(n,m) = \Theta(6 \cdot 3^n)$.

■

CS/ECE 374 B Fall 2019 Siyuan Chen (siyuanc2)
**Homework 5 Problem 2** Megan Walsh (meganew2)
Jiashuo Tong (jtong8)

**Solution:** Skipped ■

CS/ECE 374 B Fall 2019

**Homework 5 Problem 3**

Siyuan Chen (siyuanc2)
Megan Walsh (meganew2)
Jiashuo Tong (jtong8)

**Solution:**

(a) Consider the Euclid gcd algorithm below:

```python
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)
```

One can notice that the absolute worst case complexity happens when of the two input values, $x$ or $y$, one is a large value while the other is 1. Without loss of generality, assume here that $y = 1$.

At each step, the program must calculate $x - y$, which takes $\Theta(\log x + \log y)$ time. As $y = 1$, $\log y = 0$. The recurrence relation can be written as:

$$T(x) = T(x-1) + \Theta(\log(x-1))$$

Solving the recurrence relation gives $T(x) = \Theta(\log(\Gamma(x+1))) = \Theta(\log(x!))$.

For any case other than $x = 1$ or $y = 1$, the recurrence call path will be shortened considerably and therefore the runtime complexity will not exceed $\Theta(\log(\max(x, y)!))$, or $\Theta(\log(x! + y!))$.

(b) For the mod gcd algorithm shown below:

```python
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)
```

The worst case that maximizes number of recursive call is when $x//y$ or $y//x$ always equal to 1. This happens when $x$ and $y$ are two consecutive Fibonacci numbers. Assume $x > y$ and let $x = Fib(N)$, $y = Fib(N-1)$.

One can see that `mod_gcd(Fib(N), Fib(N-1))` takes N steps before terminating. As Fibonacci numbers can be approximated by $Fib(n) \approx \frac{(\frac{1+\sqrt{5}}{2})^n}{\sqrt{5}}$, the algorithm under the worst case follows the following recurrence relation:

$$T(Fib(N)) = T(Fib(N-1)) + O(\log(Fib(N)) \cdot \log(Fib(N-1)))$$

Using the fact that $\frac{Fib(N)}{Fib(N-1)} \approx \frac{1+\sqrt{5}}{2}$, the recurrence relation can be rewritten as:

$$\begin{aligned}
T(n) &= T(n/a) + O(\log(n) \cdot \log(n/a)) \\
&= T(n/a) + O(\log(n) \cdot (\log(n) - \log(a))) \\
&\approx T(n/a) + O(\log^2(n))
\end{aligned}$$

where $a = \frac{1+\sqrt{5}}{2}$. Solving the recurrence gives $T(n) = \Theta(\log^3(n))$ where $n = \max(x, y)$, or the algorithm is $\Theta(\log^3(x + y))$.

(c) For the binary gcd algorithm shown below:

```python
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)
```

Let $T(x, y)$ be the worst case runtime complexity with input $x$ and $y$. The `binary_gcd` algorithm has multiple cases and we inspect the recurrence relation of each case below: Before entering any conditional cases, the algorithm calculates `x % 2` and `y % 2` once per each recursive call, which takes $O(1)$.

If `x % 2 == 0` and `y % 2 == 0`:

$$T(x, y) = T(x/2, y/2) + 2 \cdot O(1) + O(\log x) + O(\log y)$$

If `x % 2 == 0` and `y % 2 != 0`:

$$T(x, y) = T(x/2, y) + 2 \cdot O(1) + O(\log x)$$

If `x % 2 != 0` and `y % 2 == 0`:

$$T(x, y) = T(x, y/2) + 2 \cdot O(1) + O(\log y)$$

If `x % 2 != 0` and `y % 2 != 0` and `x > y`:

$$T(x, y) = T((x - y)/2, y) + 2 \cdot O(1) + O(\log x + \log y) + O(\log(x - y))$$

If `x % 2 != 0` and `y % 2 != 0` and `x <= y`:

$$T(x, y) = T(x, (y - x)/2) + 2 \cdot O(1) + O(\log x + \log y) + O(\log(y - x))$$

In the worst case scenario, similar to (1), let $y = 1$. Then, only the second and the forth case above are possible. Additionally, $\log y = 0$, $O(\log x + \log y) = O(\log x)$, and $O(\log(x - y)) = O(\log x)$. Then, the two cases can be simplified to:

If `x % 2 == 0` and `y == 1`:

$$T(x,y) = T(x/2,y) + 2 \cdot O(1) + O(\log x)$$

If `x % 2 != 0` and `y == 1`:

$$T(x,y) = T((x-1)/2,y) + 2 \cdot O(1) + O(\log x) + O(\log x)$$

We can then conclude that $T(x,y)$, or simply $T(x)$, satisfies the following recurrence relation:

$$T(x) \leq T(x/2) + 2 \cdot O(\log x)$$

Solving the recurrence relation gives $T(x) = \Theta(\log^2(x))$ or more generally, $T(x,y) = \Theta(\log^2(x+y))$.

■