

Solution:

(a) Suppose that the input string is w and the parsed regular expression is r , the algorithm decides whether $w \in L(r)$.

Before we solve this problem, consider the text segmentation problem in Notes 2.5.

The problem requires us to segment a string into English words, for example, segment this string:

BLUESTEMUNITROBOTHEARTHANDSATURNSPIN

The general strategy is: *select the first output word, and recursively segment the rest of the input string.*

So the algorithm for text segmentation is:

```
SplitTable(A[1...n]):
1 : if n = 0:
2 :     return TRUE
3 : for i ← 1 to n:
4 :     if IsWord(A[1...i]):
5 :         if SplitTable(A[i + 1...n]):
6 :             return TRUE
7 : return FALSE
```

The problem of checking whether the input string w belongs to the language $L(r)$ can also be solved by backtracing, the first step is to *recursively devide the string into two parts, namely the prefix $w[1...i]$ and suffix $w[i + 1...n]$.*

Why can we first divide the string into two parts and then solve the problem? Here is my reason:
The parsed regular expression is in essence a tuple, we denote $r[i]$ as the i^{th} element in this tuple.
Regardless of the regular expression for the empty language, there are four cases to consider:

- if $r[0] = (a)(a \in \Sigma)/(\epsilon)$, the tuple is (a) or (ϵ) .
- if $r[0] = \cdot$, the tuple represents the concatenation of two regular expressions, namely the 1^{st} and 2^{nd} elements in the tuple, $r[1] \cdot r[2]$, the tuple is $(\cdot, r[1], r[2])$.
- if $r[0] = +$, the tuple represents the union of two regular expressions, namely the 1^{st} and 2^{nd} elements in the tuple, $r[1] + r[2]$, the tuple is $(+, r[1], r[2])$.
- if $r[0] = *$, the tuple represents the Kleene closure of a regular expression, namely the 1^{st} element in the tuple, $r[1]^*$, the tuple is $(*, r[1])$.

We first take $r[0]$ and then divide the string by the following rules:

- if $r[0] = (a)(a \in \Sigma)/(\epsilon)$, this is a base case where the string could no longer be divided, it has already been divided into a character or an empty string, the character should be a in order to match the regular expression.

- if $r[0] = \cdot$, the prefix $w[1...i]$ should match $r[1]$ and the suffix $w[i + 1...n]$ should match $r[2]$.
- if $r[0] = +$, we do not split the string and we check if the string matches $r[1]$ or if the string matches $r[2]$.
- if $r[0] = *$, both the prefix $w[1...i]$ and suffix $w[i + 1...n]$ should match $r[1]$.

So the algorithm for problem (a) is:

```
MatchRegex( $w[1...n]$ ,  $r$ ):
1 : if  $r[0] = \cdot$ :
2 :   for  $i \leftarrow 1$  to  $n$ :
3 :     return MatchRegex( $w[1...i]$ ,  $r[1]$ ) AND MatchRegex( $w[i + 1...n]$ ,  $r[2]$ )
4 : else if  $r[0] = +$ :
5 :   return MatchRegex( $w[1...n]$ ,  $r[1]$ ) OR MatchRegex( $w[1...n]$ ,  $r[2]$ ):
6 : else if  $r[0] = *$ :
7 :   for  $i \leftarrow 1$  to  $n$ :
8 :     return MatchRegex( $w[1...i]$ ,  $r[1]$ ) AND MatchRegex( $w[i + 1...n]$ ,  $r[1]$ ):
9 : else:
10:  return  $w = r[0]$ 
```

Furthermore, line 3, line 5 and line 8 can be modified to reduce the number of recursive calls as:

```
MatchRegex( $w[1...n]$ ,  $r$ ):
1 : if  $r[0] = \cdot$ :
2 :   for  $i \leftarrow 1$  to  $n$ :
3 :     if MatchRegex( $w[1...i]$ ,  $r[1]$ ):
4 :       return MatchRegex( $w[i + 1...n]$ ,  $r[2]$ )
5 :     return FALSE
6 : else if  $r[0] = +$ :
7 :   if MatchRegex( $w[1...n]$ ,  $r[1]$ ):
8 :     return TRUE
9 :   return MatchRegex( $w[1...n]$ ,  $r[2]$ )
10: else if  $r[0] = *$ :
11:   for  $i \leftarrow 1$  to  $n$ :
12:     if MatchRegex( $w[1...i]$ ,  $r[1]$ ):
13:       return MatchRegex( $w[i + 1...n]$ ,  $r[1]$ )
14:     return FALSE
15: else: # base case happens when the string could no longer be divided
16:   return  $w = r[0]$     # return TRUE if the character matches  $a$  ( $a \in \Sigma$  or  $\epsilon$ )
```

(b) The time complexity is:

$$\Theta(2^{mn})$$

Denote $T(n, m)$ as the runtime if the size of the string is n and the size of the grammar is m

- consider \cdot , $T(n, m)$ is bounded by checking all possibilities of splits:

$$T(n, 1) \leq 2n$$

$$T(n, m) = \Theta((2n)^m)$$

- consider $+$, $T(n, m)$ is bounded by checking both $r[1]$ and $r[2]$:

$$T(n, m) \leq T(n, m - 1) + T(n, m - 1) + \Theta(1)$$

$$T(n, m) = \Theta(2^m)$$

- consider $*$, $T(n, m)$ is bounded by every character except the last can match the Kleen star:

$$T(n, 1) \leq 2^n$$

$$T(n, m) = \Theta(2^{nm})$$

When there are only Kleene stars, the worst-case runtime is bounded by $\Theta(2^{mn})$.

(c) Just see m as a constant in (b), we find that $+$ takes constant time, $*$ takes $\Theta(2^n)$ time, and \cdot takes some polynomial time because $T(n, m) = \Theta((2n)^m)$.

The time complexity is $\Theta(2^n \cdot n^c)$, where c is a constant.

■

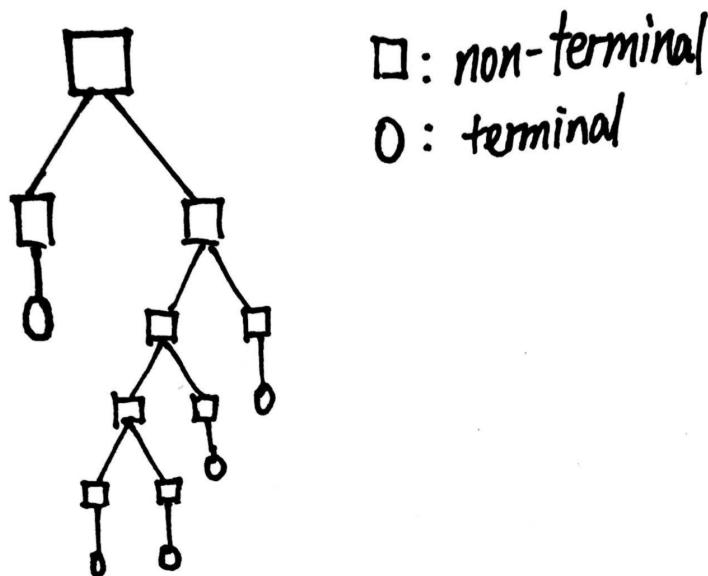
Solution:

(a) CNF grammar is defined as:

$$\begin{aligned} X &\rightarrow YZ \\ X &\rightarrow a \\ S &\rightarrow \epsilon \end{aligned}$$

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

We can see the process of deriving the string under CNF as a binary tree, where a square represents a non-terminal and a circle represents a terminal.



We need both a string $w[1...n]$ (also assume we know its length) and a start non-terminal X to check if the string can be generated by a CNF grammar. If $X \rightarrow YZ$, denote Y as $X[0]$ and Z as $X[1]$. The algorithm is as follows:

CheckCNF($w[1...n]$, X):

- 1 : **if** $n = 0$: # base case if the string is empty, return TRUE if X derives ϵ
- 2 : return $X \rightarrow \epsilon$
- 3 : **else if** $n = 1$: # base case if the string is a character, return TRUE if X derives w
- 4 : return $X \rightarrow w$
- 5 : **else** : # X is an internal non-terminal node in the tree
- 6 : $Y \leftarrow X[0]$

```

7 :   Z ← X[1]
8 :   for Y,Z in {Y, Z}:      # check two possible non-terminal derivations
9 :       for i ← 1 to n - 1: # check every prefix and its corresponding suffix
10:          if CheckCNF(w[1...i], Y):
11:              return CheckCNF(w[i...n], Z)
12:      return FALSE

```

(b) m is the size of the grammar and n is the size of the input string.

Because every non-terminal can transit either to two non-terminals or one character, the maximum number of production rules is bounded by $\Theta(m^2)$.

Suppose to check a string of length n takes $T(n)$ time, since we iterate through every possible prefix and its corresponding suffix, we have:

$$T(n) = [T(1) + T(n-1)] + [T(2) + T(n-2)] + \dots = 2\sum_1^{n-1} T(i) = S_{n-1}$$

The total runtime is:

$$2m^2\sum_1^{n-1} T(i)$$

Since $T(n) = 2m^2S_{n-1}$ and $S_n - S_{n-1} = 2m^2T(n) = 2m^2S_{n-1}$, we have:

$$S_n = (2m^2 + 1)S_{n-1} = \dots = (2m^2 + 1)^{n-1}S_1$$

Since S_1 is bounded by $\Theta(1)$, $T(n)$ is therefore bounded by:

$$\Theta((2m^2 + 1)^{n-1})$$

Or simpler,

$$\Theta(m^n)$$

■

Solution:

(a) The Euclidean algorithm to find GCD by subtraction is:

```
EuclidGCD(x, y):
1 : if  $x = y$ :
2 :     return  $x$ 
3 : else if  $x > y$ :
4 :     return EuclidGCD( $x - y, y$ )
5 : else
6 :     return EuclidGCD( $x, y - x$ )
```

If we consider every subtraction takes $\Theta(1)$ time first, denote $x + y = n$, the worst case happens when one of x or y is 1 (such like $\text{EuclidGCD}(x, 1)$), we take linear steps to get the GCD, 1, so in this case the worst-case time complexity is $\Theta(n)$.

Now every subtraction takes $\Theta(\log xy) = \Theta(\log xy)$ time, the worst case also happens when one of x or y is 1. Since $x + y = n$, the maximal value of $\Theta(\log xy)$ will be realized when xy is maximized, that is, when $x = y = \frac{n}{2}$, the corresponding $\Theta(\log xy) = \Theta(\log \frac{n^2}{4}) = 2\Theta(\log \frac{n}{4})$, the asymptotic bound is $\Theta(\log n)$. Because the value of $\Theta(\log xy)$ when one of x or y is 1 is also bounded by $\Theta(\log n)$, so we conclude that each subtraction takes $\Theta(\log n)$ time. Therefore, just like the case where each subtraction takes $\Theta(1)$ time, only the number of function calls influences time complexity, the worst case still happens when one of x or y is 1, and the worst-case time complexity is:

$$\Theta((x + y)\log xy) = \Theta(n\log n)$$

(b) The Euclidean algorithm to find GCD using mod operator is:

```
ModGCD(x, y):
1 : if  $y = 0$ :
2 :     return  $x$ 
3 : else if  $x > y$ :
4 :     return ModGCD( $y, x \% y$ )      # return  $y$  when  $x \% y = 0$ 
5 : else
6 :     return ModGCD( $x, y \% x$ )      # return  $x$  when  $y \% x = 0$ 
```

By observing the function structure, we can rewrite an equivalent function as:

```
ModGCDSimplified(x, y):
1 : if  $x \% y = 0$ :
2 :     return  $y$ 
3 : else:
4 :     return ModGCDSimplified( $y, x \% y$ )
```

Line 1 in ModGCDSimplified means that if x can be divided by y , the GCD of x and y is y , while line 4 and line 6 in ModGCD just take a step further (see my comments).

Denote (x_i, y_i) as the value of (x, y) pair where ModGCDSimplified performs i steps,

- for $i = 1$, $y_1 = 0$,
- for $i = 2$, $y \geq 1$,
- for $i > 2$, $(x_{k+1}, y_{k+1}) \rightarrow (x_k, y_k) \rightarrow (x_{k-1}, y_{k-1})$, we have:

$$\begin{aligned}x_k &= y_{k+1}, \\x_{k-1} &= y_k, \\y_{k-1} &= x_k \bmod y_k,\end{aligned}$$

Therefore $x_k = q \cdot y_k + y_{k-1}$ for some $q \geq 1$, so:

$$y_{k+1} \geq y_k + y_{k-1}$$

$$y_k \geq \text{Fibonacci}_k$$

Where Fibonacci_k represents the k^{th} Fibonacci number, which is approximated by:

$$\text{Fibonacci}_n \approx \frac{1}{\sqrt{5}} \left(\frac{1+\sqrt{5}}{2} \right)^n$$

The number of recursive calls is therefore logarithmic based on the sum of x and y , bounded by $\Theta(\log(x+y))$, since every mod operation takes $\Theta(\log x \log y)$ time, the time complexity is:

$$\Theta(\log(x+y) \log x \log y) = \Theta((\log x)^3)$$

(c) The binary Euclidean algorithm to find GCD is:

```
BinaryGCD(x, y):
1 : if x = y:
2 :   return x
3 : evenx ← x%2 = 0      # " $\leftarrow$ " is the assignment operator, " $=$ " means equal
4 : eveny ← y%2 = 0      # line 3 and line 4 assign TRUE if x % 2 / y % 2 equals 0
5 : if evenx and eveny:
6 :   return 2 · BinaryGCD(x//2, y//2)      # "//" forces integer division
7 : else if evenx:
8 :   return BinaryGCD(x//2, y)
9 : else if eveny:
10:  return BinaryGCD(x, y//2)
11 : else if x > y:
12 :   return BinaryGCD((x - y)//2, y)
13 : else:
14 :   return BinaryGCD(x, (y - x)//2)
```

Let n denote the total number of bits needed to represent both x and y , $n = \log(x \cdot y)$. We can see that $x - y$ takes $\Theta(\log x + \log y)$ time, which is equivalent to $\Theta(n)$ time; $x//2$ takes $\Theta(\log x)$ time, which is also bounded by $\Theta(n)$.

There are 3 cases to consider:

- if both x and y are even, we will remove two binary digits with constant time, let $T(n)$ denotes the total time we take if the total number of bits needed to represent both x and y is n :

$$T(n) = T(n-2) + \Theta(1)$$

- if one of x or y is even:

$$T(n) = T(n-1) + \Theta(1)$$

- if both x and y are not even, we need to delete m ($m = 1$ is the worst case) bits using linear steps, therefore,

$$T(n) = T(n-1) + \Theta(n)$$

The third case is the worst case, the time complexity is therefore:

$$\Theta(n^2) = \Theta(\log(xy)^2)$$

■