Answer to Problem Set 11

- Student Name: Gaole Dai
- Student ID: S01435587

Problem 1

Code to construct DFA is shown below:

```
def mkDFA(pat,alpha):
    dfa = [dict(zip(alpha,
                    [0 for _ in alpha]))
           for _ in pat]
    dfa[0][pat[0]] = 1
    x = 0
    for j,c in enumerate(pat[1:]):
        dfa[j+1] = dfa[x]
        dfa[j+1][c] = j+2
        x = dfa[x][c]
    return dfa
```

Apply the code, we could construct the DFA as below:

	j	0	1	2	3	4	5	6
pat		A	B	C	D	A	B	D
A		1	1	1	1	5	1	1
B		0	2	0	0	0	6	0
C		0	0	3	0	0	0	3
D		0	0	0	4	0	0	7

- j = 0, Remembering state X = dfa[A, 0] = 0
- j = 1, Remembering state X = dfa[B, 0] = 0
- j = 3, Remembering state X = dfa[C, 0] = 0
- j = 4, Remembering state X = dfa[D, 0] = 0
- j = 5, Remembering state X = dfa[A, 0] = 1
- j = 6, Remembering state X = dfa[B, 1] = 2

Trace of KMP and DFA is shown below:

idx	input	cur_state	go to state
0	A	0	1
1	B	1	2
2	C	2	3
3	A	3	1
4	B	1	2
5	C	2	3
6	D	3	4
7	A	4	5
8	B	5	6
9	C	6	3
10	D	3	4
11	A	4	5
12	B	5	6
13	D	6	7

Problem 2

2.1

$Regex = (a|b|c|d|e)$

which denotes that with symbol `|`, any character would be matched.

2.2

$Regex = (RE)(RE)*$

Pattern `(RE)+  ==> (RE)(RE)*` indicates that there will be at least 1 root regex and `*` other regex.

For example, `(RE)=(ac)`, then `Regex = (ac)(ac)*`

2.3

$$(RE)\{x,y\} = (\underbrace{RERERE\cdots RE}_{(x-1)\ number\ of\ RE})(RE|RERE|RERERE\cdots|\underbrace{RERERE\cdots RE}_{(y-x+1)\ number\ of\ RE})$$

Instance of the root regex example: $(ab)\{3,5\} = (abab)(ab|abab|ababab)$

2.4

$(RE)[RE_x - RE_y] = (RERE_x|\cdots|RERE_y)$

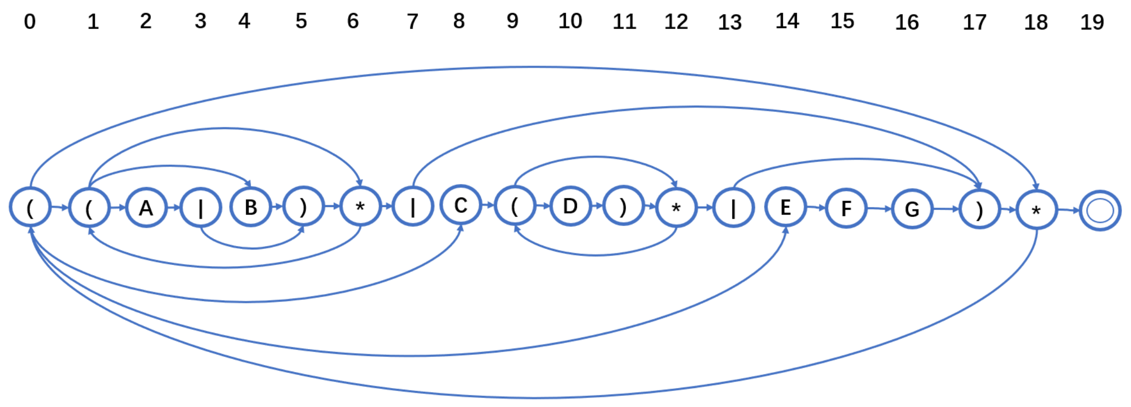
Instance of the root regex example: $(a)[b-d] = (ab|ac|ad)$

Problem 3

3.1

I added the bracket among `D` ==> $((A|B)*|C(D)*|EFG)*$

The NFA state machine is constructed as below:



3.2

As numbered the state in the figure 3.1, the state transaction is
$0 \to 1 \to 2 \to 3 \to 5 \to 6 \to 1 \to 4 \to 5 \to 6 \to 1 \to 4 \to 5 \to 6 \to 1 \to 2 \to 3 \to 5 \to 6 \to 7 \to 17 \to 18 \to 0 \to 8 \to 9 \to 12 \to 13 \to 1$
$\to 0 \to 14 \to 15 \to 16 \to 17 \to 18 \to 0 \to 8 \to 9 \to 12 \to 13 \to 17 \to 18 \to 0 \to 1 \to 2 \to 3 \to 5 \to 6 \to 1 \to 2 \to 3 \to 5 \to$
$6 \to 1 \to 4 \to 5 \to 6 \to 7 \to 17 \to 18 \to 19$

State transaction:

Input A: $0 \to 1 \to 2 \to 3 \to 5 \to 6$

Input B: $\to 1 \to 4 \to 5 \to 6$

Input B: $\to 1 \to 4 \to 5 \to 6$

Input A: $\to 1 \to 2 \to 3 \to 5 \to 6$

Input C: $\to 7 \to 17 \to 18 \to 0 \to 8 \to 9 \to 12 \to 13 \to 17 \to 18$

Input E: $\to 0 \to 14$

Input F: $\to 15$

Input G: $\to 16 \to 17 \to 18$

Input C: $\to 0 \to 8 \to 9 \to 12 \to 13 \to 17 \to 18$

Input A: $\to 0 \to 1 \to 2 \to 3 \to 5 \to 6$

Input A: $\to 1 \to 2 \to 3 \to 5 \to 6$

Input B: $\to 1 \to 4 \to 5 \to 6 \to 7 \to 17 \to 18 \to 19$

Problem 4

The algorithm is shown below:

```
def maxSubArray(nums: List[int]) -> int:
    # Initialize our variables using the first element.
    cur_subarray = max_subarray = nums[0]

    # Start with the 2nd element since we already used the first one.
    for num in nums[1:]:
        # If cur_subarray is negative, throw it away. Otherwise, keep adding to it.
        cur_subarray = max(num, cur_subarray + num)
        max_subarray = max(max_subarray, cur_subarray)

        return max_subarray
```

Apply Dynamic Programming, Kadane's Algorithm to conquer this problem. Any subarray whose sum is *positive* is worth keeping. therefore, we could store the an integer variable `curr_subarray` and add the values of each element to it. When it becomes negative, we reset it to 0, which indicates the empty array. Since the it only iterate one time of the array, the time complexity is $\mathcal{O}(n)$.

Problem 5

5.1

The python algorithm to determine if the string is valid for split is:

```
def valStrSplit(str) -> bool:
  N = len(str)
  dp = [False] * (N + 1)
  dp[0] = True
  for i in range(1, N):
    for j in range(i, 1, -1):
      if dict(str[i : j]) and dp[i-1]:
        dp[i] = True
  return dp[0]
```

The algorithm gives the $\mathcal{O}(|S|^2)$ running time, since it has nested for loops over the string, and the $dict(w)$ function uses $\mathcal{O}(1)$. $dp[n]$ is the array with boolean to store whether the substring is dictionary word, return the first element which determines if the whole string could be splited.

5.2

The python algorithm to print the substring is:

```
def outputSeq(str):
  N = len(str)
  dp = [False] * (N + 1)
  idxArr = [0] * (N + 1)
  dp[0] = True
  for i in range(1, N):
    for j in range(i, 1, -1):
      if dict(str[i : j]) and dp[i-1]:
        dp[i] = True
        idxArr[i] = j
  i = 0
  while i < N:
     print(str[idxArr[i]+1 : i] + " ")
        i = idxArr[i]
```

We have another variable which store the substring index of $dp[n]$ in $idxArr[n]$ while iterating the string, and print out with the index of the substring using while loop. The time complexity is still $\mathcal{O}(n^2)$.

Problem 6

```
def matrixMul(m, n):
    # Init two 2d-array for storing matrix multiply order and memo with 0
    dp = [ [0]*n for i in range(n)]
    trace = [ [0]*n for i in range(n)]

    for L in range (2, n):
      for i in range(1, n - L + 1):
        j = i + L - 1
        dp[i][j] = sys.maxsize
        for k in range (i, j - 1):
            # tmp = cost/scalar multiplications
            tmp = dp[i][k] + dp[k + 1][j] + m[i - 1] * m[k] * m[j]
            if tmp < dp[i][j]:
              dp[i][j] = tmp

              # Each entry bracket[i,j]=k shows
              # where to split the product arr
              # i,i+1....j for the minimum cost.
              trace[i][j] = k
    print("The min operation for matrix multiply is " + dp[1][n-1])
    return trace

def printOrderBracket(i, j, n, path):
    name = 'A'
    # if only one matrix left in current segment
    if i==j:
        print(name)
        name += 1
        return
    print("(")
    # Recursively put brackets around subexpression
    # from i to path[i][j]
    printOrderBracket(i, path[i][j], n, path)

    # Recursively put brackets around subexpression
    # from bracket[i][j] + 1 to j.
    printOrderBracket(path[i][j] + 1, j, n, path)

    print(")")
```

```
testM = [40, 20, 30, 10, 30]
N = len(testArr)
path = matrixMul(testM, N)
printOrderBracket(1, N-1, N, path)

==> The min operation for matrix multiply is 26000
==> ((A(BC))D)
```

The total number of operations needed to multiply all the matrices is the sum of the three factors, which is the left subsequence from index `i` to `j`, the product of these dimensions is $ri \times cj$ and the right subsequence from index `j+1` to `k`, the product has dimensions $cj \times ck$. The minimization of these numbers in the subsquence could be shown in recurrence relation:

$$dp[i,j] = \begin{cases} 0 & \text{if } i = j \\ \min\limits_{i \leq j < k} [dp[i,j] + dp[j+1,k] + (m_i \times m_j \times m_k)] & \text{if } i < j \end{cases}$$

The time complexity of the DP method is $\mathcal{O}(N^3)$.

Problem 7

```
    def minDistance(self, word1, word2):

        word1Len = len(word1)
        word2Len = len(word2)

        # if one of the strings is empty
        if word1Len * word2Len == 0:
            return word1Len + word2Len

        # array to store the convertion history
        d = [ [0] * (word2Len + 1) for _ in range(word1Len + 1)]

        # init boundaries
        for i in range(word1Len + 1):
            d[i][0] = i
        for j in range(word2Len + 1):
            d[0][j] = j

        # DP compution
        for i in range(1, word1Len + 1):
            for j in range(1, word2Len + 1):
                left = d[i - 1][j] + 1
                down = d[i][j - 1] + 1
                left_down = d[i - 1][j - 1]
                if word1[i - 1] != word2[j - 1]:
                    left_down += 1
                d[i][j] = min(left, down, left_down)
        return d[word1Len][word2Len]
```

We store the values for characters in a matrix $dp[n][n]$ and use the values instead of computing them again, and in the matrix, we store the minimum value of operations required for `i` characters of `word1` and `j` characters of `word2` in $dp[i, j]$.

If both characters are same the edit distance, then remains the same, otherwise the minimum of current edit distance +1.

The recurrence relation is:

$$dp[i, j] = \min \begin{cases} dp(i-1, j) + 1 & \text{if deletion} \\ dp(i, j-1) + 1 & \text{if insertion} \\ dp(i-1, j-1) + \begin{cases} 0 & \text{if } s[i] = t[j] \\ 1 & \text{if } s[i] \neq t[j] \end{cases} & \text{if replacement} \end{cases}$$

The time complexity and space complexity is $\mathcal{O}(n * m)$.