Jiapeng Pei S01435611

Module 11 Problems

1. [12 pts] Given input alphabet {A, B, C, D}, construct the DFA for the string

   A B C D A B D

	0	**1**	**2**	**3**	**4**	**5**	**6**
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

   Show a trace of the KMP algorithm and your DFA on the target string

   A B C A B C D A B C D A B D

   - **A**BCABCDABCDABD

     ABCDABD

   - A**B**CABCDABCDABD

     A**B**CDABD

   - AB**C**ABCDABCDABD

     AB**C**DABD

   - ABC**A**BCDABCDABD

     ABCDABD

   - ABCA**B**CDABCDABD

     A**B**CDABD

   - ABCAB**C**DABCDABD

     AB**C**DABD

   - ABCABC**D**ABCDABD

     ABC**D**ABD

   - ABCABCD**A**BCDABD

     ABCD**A**BD

   - ABCABCDA**B**CDABD

     ABCDA**B**D

   - ABCABCDA**B**CDABD

     ABCDA**B**D

- ABCABCDAB**C**DABD

  AB**C**DABD
- ABCABCDABC**D**ABD

  ABC**D**ABD
- ABCABCDABCD**A**BD

  ABCD**A**BD
- ABCABCDABCDA**B**D

  ABCDA**B**D
- ABCABCDABCDAB**D**

  ABCDAB**D**

2. [12 pts] Regular expressions (or regexs) are often extended with convenience operations. Assuming the alphabet is the lower case letters a,b,c,d,e; write the regular expression for the following convenience operations:

   2.1 [3pts] Wildcard (usually the '.' character): will match any character in the alphabet0

   . => (a|b|c|d|e)

   2.2 [3pts] Kleene +: Like *, but the regular expression must have at least 1
   instance of the root regex

   (RE)+ => RE(RE)*

   2.3 [3pts] bounded closure: Match a finite set of concatenations.

   (RE){3, 5} => (RE RE RE | RE RE RE RE | RE RE RE RE RE)

   2.4 [3pts] character range:
   Example: a[b-d] matches ab, ac, or ad

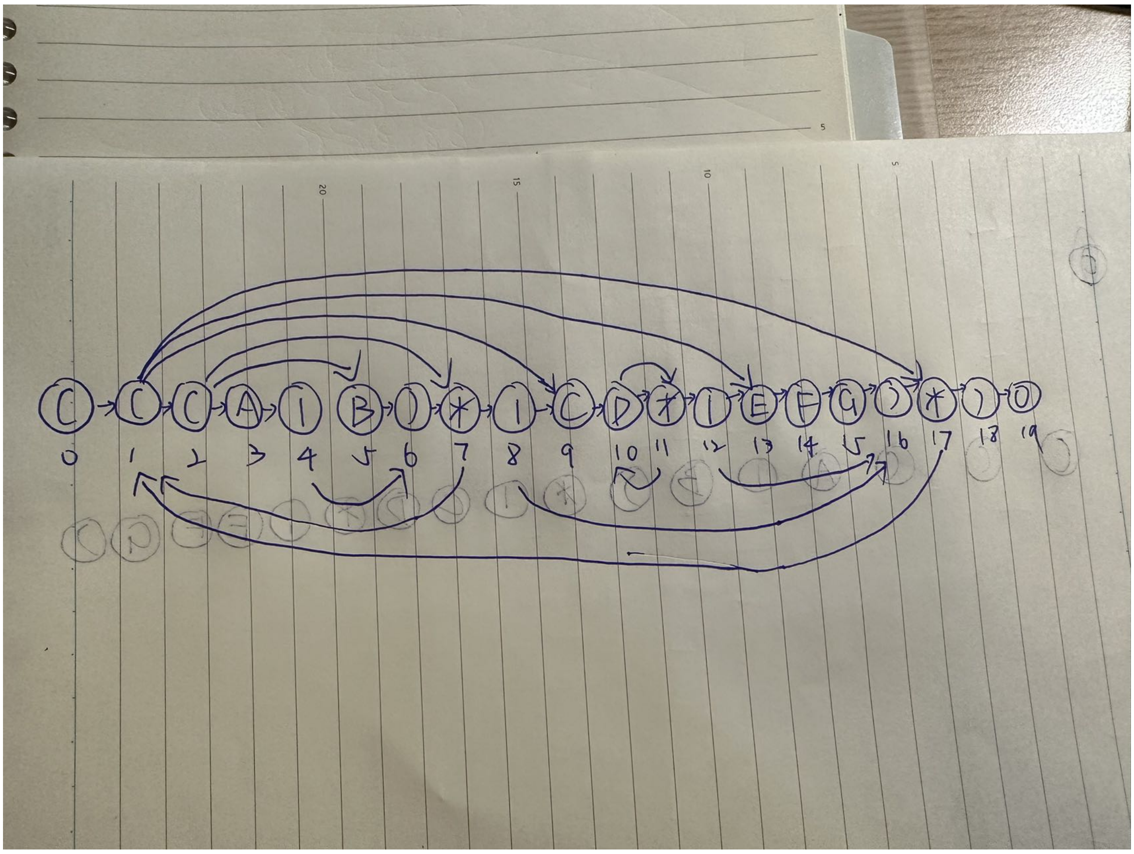
   RE[b-d] => RE(b|c|d)

   RE[a-e] => RE(a|b|c|d|e)

3. [18 pts] Given the regex over the alphabet {A,B,C,D,E,F,G}

   ((A | B)* | C D* | E F G)*

   3.1 [9 pts] Construct the NFA state machine and digraph of e-tranisitons for the given regex



3.2 [9 pts] Show the full set of state transitions when your NFA recognizer is applied to

A B B A C E F G E F G C A A B

- begin: 0, 1, 2, 3, 5, 7, 8, 9, 13, 17, 18, 19
- transition: 0-1-2-3-4-6-7-2-5-6-7-2-3-4-6-7-8-16-17-1-9-10-11-12-16-17-1-13-14-15-16-17-1-13-14-15-16-17-1-9-10-11-12-16-17-1-2-3-4-6-7-2-3-4-6-7-2-5-6-7-8-16-17-18-19

4. [14 pts] A contiguous subsequence of a list S is a subsequence made up of consecutive elements of S.
For instance, if S is

5, 15, −30, 10, −5, 40, 10

then

15, −30, 10

is a contiguous subsequence.

The subsequence

5, 15, 40

is **not** contiguous.

Give a linear-time algorithm for the following task:

Input: A list of numbers, $a1, a2, \ldots, an$
Output: A contiguous subsequence of maximum sum.
NOTE: A subsequence of length 0- has sum 0.

For the preceding example sequence,the answer would be

10, −5, 40, 10

with a sum of 55.

Hint: For each j ∈ {1, 2, . . . , n}, consider contiguous subsequences ending exactly at position j.

```
def findSubsequence(array, n):
    maxSum = float('-inf')
    curSum = 0
    left = 0
    right = 0
    begin = 0
    end = 0
    while right < n:
        curSum += array[right]
        right += 1
        if curSum > maxSum:
            maxSum = curSum
            begin, end = left, right
        if curSum <= 0:
            curSum = 0
            left = right

    return array[left:right]

print(findSubsequence([10, -5, 40, 10], 4)) # [10, -5, 40, 10]
print(findSubsequence([-5], 1)) # [-5]
```

5. [14 pts] You are given a string of n characters s[1 . . . n], which you believe to be a corrupted text document in which all punctuation and whitespace has vanished. A sample input is:

   itwasthebestoftimesitwastheworst

   You wish to reconstruct the document using a dictionary. The dictionary is available in the form of a Boolean function

   dict(w):
   for any string w, dict(w) = true if w is a valid word in the dictionary
   dict(w) returns false otherwise.

   5.1 [9 pts] Give a dynamic programming algorithm that determines whether the string s can be reconstituted as a string of valid words.
   The running tine should be no worse that $O(|s|^2)$, assuming a call to the dict function takes $O(1)$.

```
def validString(strDict, s):
    n = len(s)

    state = [False for i in range(n+1)]
    state[0] = True
    for i in range(1, n+1):
        for j in range(i):
            if state[j] and s[j:i] in strDict:
                state[i] = True
                break
    return state[n]

print(validString(['it', 'was', 'the', 'best', 'of', 'times', 'worst'],
'itwasthebestoftimesitwastheworst')) # True
```

5.2 [5 pts] In the event that the string is valid, make your algorithm output the corresponding sequence of words.

```
def validString(strDict, s):
    n = len(s)

    state = [False for i in range(n+1)]
    state[0] = True
    ret = []

    for i in range(n):
        for j in range(i+1, n+1):
            if state[i] and s[i:j] in strDict:
                state[j] = True
                ret.append(s[i: j])

    return ret

print(validString(['it', 'was', 'the', 'best', 'of', 'times', 'worst'],
'itwasthebestoftimesitwastheworst'))
# ['it', 'was', 'the', 'best', 'of', 'times', 'it', 'was', 'the', 'worst']
```

6. [16 pts] Best Matrix Multiply Order:

Let

$$A1\ A2\ A3\ A4\ \dots\ An$$

be a sequence of matrices that must be multiplied together.
Matrix Multiply is associative, so you may parenthesize the multiplication in whatever way works best.
Some reminders: An m x p matrix can be multiplied by a p x q matrix giving an m x q matrix.
The cost of multiplying the 2 matrices is m * p * q.

Example: Given the sequence

$$A1\ A2\ A3$$

where A1=10×10, A2=10×10, A3=10×1.
Choosing (A1 * A2) * A3 gives:

cost of A1×A2=10×10×10=1000
cost of(A1×A2)×A3=10×10×1
total cost = (1000) + 100 = 1100

On the other hand, choosing A1×(A2×A3)
cost of A2×A3=10×10×1=100
cost of A1×(A2×A3)=10×10×1=100
total cost = 100 + 100 = 200

So, 2nd choice is clearly best.

```
# dimArray[i-1] and dimArray[i] are the #rows and #cols of a matrix. For example:
# [10, 10, 10, 1] represents 3 matrix of size [10, 10], [10, 10] and [10, 1].

def findBestOrder(dimArray, n):
    inf = float('inf')

    # dp[i][j] donotes the min cost for multiplying from i to j
    row = [inf for i in range(n+1)]
    dp = [row for i in range(n+1)]
    for i in range(n+1):
        dp[i][i] = 0

    for k in range(2, n+1):
        for i in range(1, n-k+2):
            back = i + k - 1
            for j in range(i, back):
                dp[i][back] = min(dp[i][back], dimArray[i-1]*dimArray[j]*dimArray[back] + dp[i][j] + dp[j+1][back])

    return dp[1][n]

print(findBestOrder([10, 10, 10, 1], 3)) # 200
```

7. [14 pts] Minimum Edit Distance:

   You have at your disposal 3 character editing operations:

   1. d: delete a char
   2. i: insert a char
   3. c: change 1 char into another char

   You are given 2 strings s,t.
   Your problem is to always find the minimum # of operations to turn string s into string t.

   Example: s = 'cast', t='cats'

   Option 1: c s->t [pos 3]; c t->s [pos 4] has 2 operations
   Option 2: d a [pos 2]; c s a [pos 2];i s at the end. This has 3 operations.

   Option 1 is clearly better option 2.

```
def minDistance(word1, word2) {
    m = len(word1)
    n = len(word2)
    if m == 0 or n == 0:
        return max(m, n);
    row = [0 for i in range(n+1)]
    dp = [row for i in range(m+1)]

    for i in range(n+1):
        dp[0][i] = i
    for i in range(m+1):
        dp[i][0] = i
```

```python
    for i in range(1, m+1):
        for j in range(1, n+1):
            if word1[i - 1] == word2[j - 1]:
                dp[i][j] = dp[i-1][j-1]
            else:
                dp[i][j] = 1 + min(dp[i][j-1], dp[i-1][j], dp[i-1][j-1])

    return dp[m][n]
```