HW5

Xinhao Luo

Tuesday 12$^{\text{th}}$ May, 2020

1 DPV Problem 6.4

a) Sub-problem S(i): set true if string s[1..i] is a valid string, false otherwise.

```
Initialize S(1...len(s)) to false, S(0) = true.
for i from 1 to len(s):                                    // O(n)
    for j from 0 to i-1:                                // O(n)
        S(i) = S(i) or (S(j) and dict(s[j+1:i]))   // O(1)
return S(len(s))
```

The time complexity should be $O(n^2)$ at most. The algorithm starts from left to right and calculate all valid combinations of words, and once encounter overlapping, cached result will be used.

b) Assume we have the array stores the result of function S from part a)

```
if S(len(s)) is true:
    result = []
    i = len(s)
    j = i - 1
    while i >= 0:
        while S(i) == false or dict(s[j:i]) == false:
            i -= 1
        result.append(s[j:i])
        i = j - 1
        j = i - 1

    return result.reverse()
```

The time complexity should be $O(n^2)$

The result added from right to left each time. We only add word if two parts: current location until last location, and the last location until end are valid.

2 DPV Problem 6.8

1. Define a 2D-array, a[n][m], where a[i][j] means the longest common substring's length at index x_i and y_j.

2. Define findLargest(a) takes a 2D-array and return the largest value. The expected time complexity of this function should be O(mn).

```
Initialize array a[n][m] to 0
for i from 1 to n:
    for j from 1 to m:
        if x[i] = y[j]:
            a[i][j] = a[i-1][j-1] + 1 // O(1)

// The loop above should be O(nm) = O(mn)

return findLargest(a) // O(mn)
```

The time complexity should be O(nm) + O(mn) = O(mn)

Here we calculate every subset of string and store in the 2D-array. If we find a common letter, we will add 1 to the longest length and continue, otherwise keep the value as it is.

3 DPV Problem 6.13

a) Example: "2911"

With greedy, the first player will pick 2, which expose 9 to the second player

P1 2,1

P2 9,1

However, if the first player pick 1 first

P1 1,9

P2 2,1

No matter how player 2 pick, 9 will always be player 1

b) (a) Assume that the second player will also choose the optimal solution, which minimize what first player would have.

(b) Define a 2D-array that r[n][n], where r[i][j] means that the optimal sum the first player could have at this position, and its corresponded draw.

```
Initialize array r[n][n] as (0, front)
for i from n to 1:
    r[i][i] = (s[i], front)
    for j from i+1 to n:
        f = s[i] + min(r[i+2][j][0], r[i+1][j-1][0])
        b = s[j] + min(r[i][j-2][0], r[i+1][j-1][0])
        if f > b:
            r[i][j] = (f, front)
        else:
            r[i][j] = (b, back)
```

Because of the Double for-loop, time complexity should be $O(n^2)$

Each time, the result can be retrieved by using i, j, where i means how many card draw from front and j is from the back

```
return r[i][j][1]
```

In every step, we calculate the optimal strategy of the first player from the previous conditions in the table.

4 DPV Problem 6.22

Define 2D-array r[n][t], where r[i][j] is true if sum($a_1 ... a_i$) = j, false otherwise. If there is a integer x larger than integer

```
Initialize array r[n][t] = false, all r[i][0] = true
for i from 1 to n:
    r[i][a[i]] = true
    for j from 1 to t:
        r[i][j] = r[i-1][j] or r[i-1][j-a[i]]

// The loop above should be O(nt)

return r[n][t]
```

The time complexity is O(nt)

In every step, we mark if the current number can be reached by looking up the previous subset, or by adding a new number.