

Sources:

Problem A

1. <https://stackoverflow.com/questions/10577422/create-palindrome-from-existing-string-by-removing-characters#10577540>
2. <https://www.geeksforgeeks.org/longest-common-subsequence-dp-using-memoization/>

Problem B

3. <https://leetcode.com/problems/wildcard-matching/solution/>
4. <https://stackoverflow.com/questions/49695477/removing-specific-duplicated-characters-from-a-string-in-python>

Problem A.

Approach 1 : Brute force

For Approach 1 I used the brute force recursive solution of the Longest Common Subsequence (LCS) problem that was discussed in *Exploration: Dynamic Programming - Longest Common Subsequence Problem*. I used LCS because it allows you to compute the longest common subsequence of the original string and its mirror, or inverted string. Thus in *checkPalindrome()*, the length of the original string is stored and then the original string, the inverted original string, and the lengths of the strings are passed to the helper function *checkPalindrome_1_helper()* to compute the length of the LCS. The helper function then returns the length of the max subsequence *k* to *check_Palindrome_1()*. If the length of the max subsequence is less than the length of the original string minus *k*, then it is not possible to make a palindrome by removing at most *k* letters.

```
checkPalindrome_1_helper(s1, s2, m, n):  
    # base case  
    if m < 0 or n < 0:  
        return 0  
  
    # characters match  
    else if s1[m] = s2[n]:  
        return 1 + checkPalindrome_1_helper(s1, s2, m-1, n-1)  
  
    # characters do not match  
    else:  
        return max(checkPalindrome_1_helper(s1, s2, m-1, n), checkPalindrome_1_helper(s1, s2, m, n-1))
```

```
checkPalindrome_1(string, k):  
    # get length of max substring of split string  
    m = n = len(string)  
    max_substring = checkPalindrome_1_helper(string, string[::-1], m-1, n-1)  
  
    # check that the palindrome is >= the length of the string - k  
    if max_substring >= (n - k):  
        return True  
    else:  
        return False
```

Time complexity: $O(2^n)$, where *n* is the length of the original string

Problem A.

Approach 2 : Dynamic Programming

For Approach 2 I chose a top-down dynamic programming solution of LCS discussed in *Exploration: Dynamic Programming - Longest Common Subsequence Problem*. As with the brute force solution of Approach 1, the LCS is computed for the original string and its mirror. The difference in Approach 2, however, is that the solution to each subproblem is stored in a 2d array `memo[][]` so that it does not have to be recomputed. This transforms the time complexity of the solution from exponential to polynomial. The result, or the length of the longest common subsequence, is stored in the bottom right corner of `memo[][]` and returned to `checkPalindrome_2()`, so that it can be computed if the k palindrome is valid.

```
def checkPalindrome_2_helper(s1, s2, m, n, memo):
    # base case
    if m == 0 or n == 0:
        return 0

    # if sub-problem is already stored
    if memo[m-1][n-1] != -1:
        return memo[m-1][n-1]

    # characters match
    if s1[m-1] == s2[n-1]:
        memo[m-1][n-1] = 1 + checkPalindrome_2_helper(s1, s2, m-1, n-1, memo)
        return memo[m-1][n-1]

    # characters do not match
    else:
        memo[m-1][n-1] = max(checkPalindrome_2_helper(s1, s2, m-1, n, memo), checkPalindrome_2_helper(s1, s2, m, n-1, memo))
        return memo[m-1][n-1]
```

```
def checkPalindrome_2(string, k):
    # create 2d array to store results
    m = n = len(string)
    memo = [[-1 for x in range(n+1)] for x in range(m+1)]

    # get length of max substring of split string
    max_substring = checkPalindrome_2_helper(string, string[::-1], m, n, memo)

    if max_substring >= (n - k):
        return True
    else:
        return False
```

Time complexity: $O(n^2)$, where n is the length of the original string

(Bonus) Does your Approach 2 have improved time complexity compared to Approach 1, if yes compare the time complexities.

Yes, the brute force approach for Approach 1 had a time complexity of $O(2^n)$, whereas the dynamic programming approach used for Approach 2 had a reduced time complexity of $O(n^2)$. Note that for both approaches n is the length of the original string.

(Bonus) How did you use the techniques learned in this course to solve this problem?

Name the algorithm techniques/strategies that you have used.

I relied on the top-down dynamic programming approach that was used to solve the Longest Common Subsequence problem discussed in the lecture: *Exploration: Dynamic Programming - Longest Common Subsequence Problem*. A 2d array, `memo[][]`, stores the solution to each subproblem as it is solved, so that it does not have to be recomputed. This reduces the time complexity from

Problem B.*Pattern Matching : Dynamic Programming*

This problem also reminded me of the dynamic programming solution of LCS discussed in *Exploration: Dynamic Programming - Longest Common Subsequence Problem*. Again, I implemented a bottom-up solution where the solved results of the subproblems are stored in a 2d array *memo*[][]. Because of the special characters '*' and '?', the recursive formula is slightly different from the recursive formula of LCS.

$$\text{patternmatch}[s, p] = \begin{cases} \text{False} & \text{if } s = "" \text{ or } p = "" \\ \text{True} & \text{if } s = p \\ & \text{if } p = '*' \\ \text{patternmatch}[s[1:], p[1:]] & \text{if } s = p \text{ or if } p = '?' \\ \text{patternmatch}[s, p[1:]] \text{ or } \text{patternmatch}[s[1:], p] & \text{if } s \text{ and } p \text{ do not match and } p = '*' \end{cases}$$

```
def patternmatch(string, p):
    # remove any duplicate * from pattern
    pattern = remove_dup_char(p, '*')

    # memo to store calculated values
    m = len(string)
    n = len(pattern)
    memo = [[0 for x in range(n + 1)] for x in range(m + 1)]

    # process bottom-up
    for i in range(1, m + 1):
        for j in range(1, n + 1):
            # if the entire pattern is *, then True
            if pattern[0:] == '*':
                return True
            # sub-string & sub-pattern match, or remaining pattern is *
            elif string[i - 1] == pattern[j - 1] or pattern[j:] == '*':
                memo[i][j] = True
            # if 1st char of string & pattern equal, keep processing remainder
            elif string[i - 1] == pattern[j - 1] or pattern[j - 1] == '?':
                if i == 1 and j == 1:
                    memo[i][j] = True
                else:
                    memo[i][j] = memo[i - 1][j - 1]
            # else if p != s and p = *
            elif pattern[j - 1] == '*':
                memo[i][j] = memo[i][j - 1] or memo[i - 1][j]
            # if none of the cases apply, then there is no match
            else:
                memo[i][j] = False

    return memo[i][j]
```

```
def remove_dup_char(p, to_remove):
    pattern = []
    repeat_char = False
    for i in range(len(p)):
        if (i == 0) or (p != to_remove) or (p[i-1] != to_remove):
            pattern.append(p[i])

    return pattern
```

Problem B.

Pattern Matching : Dynamic Programming

Time complexity: $O(m*n)$, where m is the length of the original string and n is the length of the original pattern.

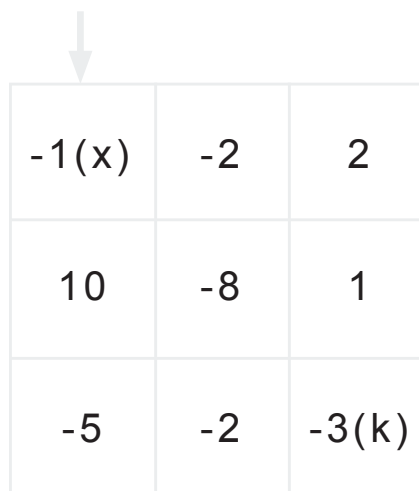
(Bonus) How did you use the techniques learned in this course to solve this problem? Name the algorithm techniques/strategies used.

I relied on Dynamic Programming to create an efficient bottom-up solution with the use of a 2d array *memo[][]* to store the results of computed subproblems. This reduced the time complexity from exponential to $O(m*n)$. Since this problem was very similar to LCS, this helped me a lot in figuring out the recursive formulas for the subproblems. The normal character matching conditions were the same, I only needed to consider how the '*' had the same as the condition when the characters do not match.

Problem C.

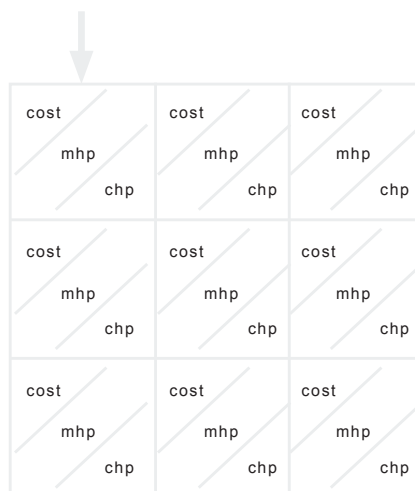
Maze : Greedy Solution

I chose to implement a greedy solution because we need to find an *optimal* solution to the maze problem. In this case, we are trying to find the *minimum* number of health points required for a path from the start of the maze to the end of the maze. We are only allowed to move to the right or below, not in the diagonal direction, so at each step we consider the cumulative cost of minimum health points from the previous cells to the left and above. We then choose the previous cell with the minimum health points, or if the minimum health points are the same for both cells, then we choose the cell with the higher total of current health points. If the previous cell is out of bounds of the 2d array, then the value is ∞ . We then adjust our current cumulative totals for the current cell and proceed to process the next cell. At the end of the maze, the minimum health points value, mhp, is the solution to the maze.



-1(x)	-2	2
10	-8	1
-5	-2	-3(k)

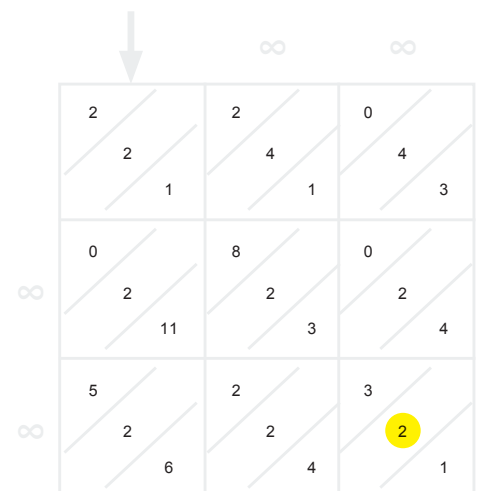
example maze



cost mhp chp	cost mhp chp	cost mhp chp
cost mhp chp	cost mhp chp	cost mhp chp
cost mhp chp	cost mhp chp	cost mhp chp

storing results of previous steps

cost: the cost for each step
if cell is negative = abs(cell), else = 0
*starting cell costs an extra point
mhp: cumulative cost of health points
chp: running total of current health points



2 2 1	2 4 1	0 4 3
0 2 11	8 2 3	0 2 4
5 2 6	2 2 4	3 2 1

greedy strategy applied to example maze

result is stored in mhp in bottom right cell

Time complexity: $O(m*n)$, where maze is a 2d array of size $m \times n$.

(Bonus) How did you use the techniques learned in this course to solve this problem? Name the algorithm techniques/strategies used.

I considered using greedy, backtracking and dynamic programming solutions to solve the maze problem. Ultimately I chose a greedy solution because at each step of the maze you can choose the local optimum value to find the global optimal solution, minimal health points, for the problem. I did not use a dynamic programming solution, because I did not need to access previously computed values for multiple results, only for the two previous cells. I did not use a backtracking approach because it had a much larger time complexity. I also considered using Dijkstra's shortest path algorithm, but this did not work well because the maze contained negative weights.

Problem C.

Maze : Greedy Solution

```
def getTesla(maze):
    # create two tables to store results: mhp = minimum health pts, chp = current health pts
    mhp = chp = [[0] * len(maze[0]) for i in range(len(maze))]

    # fill mhp bottom-up
    for i in range(0, len(maze)):
        for j in range(0, len(maze[0])):
            # base case
            if i == 0 and j == 0:
                # negative step
                if maze[0][0] < 0:
                    mhp[0][0] = abs(maze[0][0]) + 1
                    chp[0][0] = maze[0][0] + mhp[0][0]

                # positive step
                else:
                    mhp[0][0] = 0
                    chp[0][0] = maze[0][0]
                    continue

            # fill out of bounds cells with values
            mhp_left = mhp_above = infinity
            chp_left = chp_above = 0
            # if cell to left is valid
            if j != 0:
                mhp_left = mhp[i][j-1]
                chp_left = chp[i][j-1]
            # if cell above is valid
            if i != 0:
                mhp_above = mhp[i-1][j]
                chp_above = chp[i-1][j]

            # select best mhp from above or left cell: if mhp is equal, then choose cell with best chp
            mhp_prev = chp_prev = 0
            if mhp_above < mhp_left:
                mhp_prev = mhp_above
                chp_prev = chp_above
            elif mhp_above > mhp_left:
                mhp_prev = mhp_left
                chp_prev = chp_left
            elif mhp_above == mhp_left:
                if chp_above > chp_left:
                    mhp_prev = mhp_above
                    chp_prev = chp_above
                else:
                    mhp_prev = mhp_left
                    chp_prev = chp_left

            # negative step
            if maze[i][j] < 0:
                hp_cost = abs(maze[i][j])
                if chp_prev > hp_cost:
                    mhp[i][j] = mhp_prev
                    chp[i][j] = chp_prev - hp_cost
                else:
                    mhp[i][j] = mhp_prev + hp_cost
                    chp[i][j] = 1

            # positive step
            else:
                mhp[i][j] = mhp_prev
                chp[i][j] = chp_prev + maze[i][j]

    return mhp[len(maze)-1][len(maze[0])-1]
```