Algorithms Analysis by Joshua Luo

Palindrome.py

Determine if a given string contains a palindrome when at most k characters are removed from the original string.

Ex:
string: “abcdeba”; k = 2
Output: True
Explanation: By removing characters “de” from the string, it becomes a 2-palindrome string “abcba”.

Approach 1: Brute Force

Starting with the original string, we will determine every possible string resulting from removing 0 to len(string) characters from the string. Whenever the current working string is the appropriate length (having 0 to k characters removed), we will check if it is a palindrome. The boolean result will be logically OR'd with the returned boolean variable(initialized as False) so that any found k-palindrome, which would have a True check result, will make the returned boolean variable True.

Pseudocode:

```
def checkPalindrome_1(string, k, string):
    return helper_1(string, k)

def helper_1(string, k, cur_string):
    if len(cur_string) == 0:
        return False

    is_palindrome = False
    if len(string) - k <= len(cur_string <= len(string):
        is_palindrome = is_palindrome or cur_string == cur_string[::-1]

    for i in range(len(cur_string)):
        next_string = cur_string[0:i] + cur_string[i+1:len(cur_string)]
```

```
            next_string_is_palindrome = helper_1(string, k, next_string)
            is_palindrome = is_palindrome or next_string_is_palindrome

    return is_palindrome
```

Time complexity: O(n!)

Starting with the original string of n letters as our working string, we have n choices for letters to remove. Then for each resulting working string after removing a letter, there are now n-1 choices of letters to remove, then n-2, etc., until the string is empty. This means that beginning with the first call, the for loop will run n times, then n-1 times, then n-2 times, etc. which is n*(n-1)*(n-2) …, which is O(n!).

Approach 2: Top-down Memoization

If we draw a tree to represent each possible step and option of removing a letter, we will find that there are repeating substring subproblems because you can reach the same substring by removing the same letters but in a different order. We can use memoization to store each unique substring which can be referred to for a known solution before further exploring letters to remove. The result of the working substring will first be logically OR'd with the palindrome check of its substrings so that any found k-palindrome can carry a True boolean through to be returned.

Pseudocode:

```
def checkPalindrome_2(string, k):
    memo = {}
    helper_2(string, k, string, memo)
    return memo[string]

def helper_2(string, k, cur_string, memo):
    if cur_string in memo:
        return memo[cur_string]

    has_k_paindrome = False
    if len(string) - k <= len(cur_string <= len(string):
        has_k_palindrome = has_k_palindrome or cur_string == cur_string[::-1]
```

```
    for i in range(len(cur_string)):
        next_string = cur_string[0:i] + cur_string[i+1:len(cur_string)]
        has_k_palindrome = has_k_palindrome or helper_1(string, k, next_string)

    memo[cur_string] = has_k_palindrome
    return is_palindrome
```

Time complexity: $O(2^n)$

Comparison:

Approach 2 uses the dynamic programming method of top-down memoization to eliminate repetitive subproblems. Repetitive subproblems appear when the same letters are removed but in different orders, resulting in the same substring subproblem. Whereas a brute force algorithm would completely process the exact same subproblems where a given substring is identical to one previously encountered, memoization holds the result determined from the previous encounter, and uses a constant look-up time to get the answer instead of processing that subproblems and any further identical subproblems.

Patternmatch.py

Determine if the given string matches a given string pattern, which can have character symbols “?” for any single character and “*” for any sequence (including empty) of characters.

Ex:
string: “abcde”; pattern: “*a?c*”
Output: True

Approach:

We will use a 2-D matrix with the dimensions of the length of the string and the length of the pattern to compare whether the string matches the pattern. We then fill out the matrix by determining if a string of a certain length matches a pattern of a certain length. For any given index, we can actually determine if the substring of the string and pattern so far are a match depending on whether their individual index values match and the result from previous substrings. The possible cases of comparing strings up to a certain index are 1) if the letters at

given indexes of the string and pattern are the same or the pattern's index value is "?", then the result is whether the substring with indexes - 1 for both matches or not; 2) if the pattern's index value is "*", then the result is either of the index - 1 value of the string with the current index value of the pattern (the * represents 1 or more letters) OR the current index value of the string with the index - 1 value of the pattern (the * represents 0 letters); or 3) if the current index values of the string and pattern don't match, then the result is False. Once the matrix is filled out, the last cell, which is whether the entire string matches the entire pattern, will hold whether the string matches the pattern.

Pseudocode:

```
def patternMatch(string, p):
    m = len(string)
    n = len(p)
    cache = [[None for x in range(n+1)] for x in range(m+1)]

    for i in range(m+1):
        for j in range(n+1):
            if i == 0 and j == 0:
                cache[i][j] = True
            elif j == 0:
                cache[i][j] = False
            elif i == 0 and p[j - 1] == "*":
                cache[i][j] = cache[i][j - 1]
            elif p[j - 1] == "?" or (string[i - 1] == p[ - 1j]):
                cache[i][j] = cache[i - 1][j - 1]
            elif p[j] == "*":
                cache[i][j] = cache[i - 1][j] or cache[i][j - 1]
            else:
                cache[i][j] = False

    return cache[m][n]
```

Time complexity: O(m * n) where m and n are the lengths of the string and pattern

This solution uses dynamic programming with a bottom-up longest common substring algorithm. Starting with an empty string and pattern, we incrementally add every possible combination of letters to the working string and the pattern and determine if that's a match depending on whether the newest added letter to each is a match and the match result before those letters were added. When the matrix is completely filled, the last cell will hold whether the complete string and the complete pattern are a match.

GetTesla.py

Mr. X is in a maze. Tesla's latest model car's keys are in the bottom right corner of the maze. Mr. X is located at the top left corner of the room. Mr. X can move only right or down in a single step.

At each step there is an energy booster (positive integers) or an energy exhausting space (negative integers) or neutral space (0). As Mr. X enters that space, he can gain or lose health. The first room and the last room would also impact Mr. X's energy levels.

Mr. X starts with some health points, and he may lose or gain health points as he enters into different spaces in the maze.

Find the initial health points needed for Mr. X to be able to get the keys. The function returns the lowest possible starting health points value.

-1 (X)	-2	2
10	-8	1
-5	-2	-3 (K)

The minimum health points needed are 2. X would follow the path

down->down->right->right

Sample Input: M: [[-1, -2, 2], [10, -8, 1], [-5, -2, -3]]; Output: 2

Approach:

We will make a matrix of the same size as the given matrix M, which will store the minimum energy required to reach the last space from the current space. As we work from the last space we can move up or left, the energy cost for the next valid and/up or left space is the sum of the current space and the next space. There are two possible cases depending how the sum affects Mr X's energy level: 1) the sum makes the total energy cost zero or positive, which means we can effectively make the total cost 0 from this space; or 2) the sum makes the total energy cost negative, which means this is the amount of energy loss we must overcome. When we fill out all the spaces, we will have a value at our starting space for how much it will cost to reach the last space from the starting space, and our minimum starting energy will be the absolute value of that plus 1 (so Mr. X's energy level stays 1 or more).

Pseudocode:

```
def getTesla(M):
    m = len(min_energy_map)
    n = len(min_energy_map[0])
    min_energy_map = [[None for x in range(n)] for x in range(m)]
    min_energy_map[m - 1][n - 1] = M[m - 1][n - 1]
    for i in range(m - 1, -1, -1):
        for j in range(n - 1, -1, -1):
            if i != 0 or j != 0:
                if i - 1 >= 0:
                    cost_above = min_energy_map[i][j] + M[i - 1][j]
                    if cost_above > 0:
                        cost_above = 0
                    min_energy_map[i - 1][j] = cost_above
                if j - 1 >= 0:
                    cost_left = min_energy_map[i][j] + M[i][j - 1]
                    if cost_left >= 0:
                        cost_left = 0
                    if i < m - 1:
                        min_energy_map[i][j - 1] = max(cost_left, min_energy_map[i][j - 1])
                    else:
                        min_energy_map[i][j - 1] = cost_left
```

```
    return abs(min_energy_map[0][0]) + 1
```

Time complexity: O(m * n) where M is a matrix with dimensions m by n spaces

My solution uses bottom-up dynamic programming to determine the cost of reaching the final space when starting from a space. Beginning with the smallest maze of a single space (the last space of the input M), it will cost the value of that space. From there, we have two possible spaces that Mr. X can move from: the left or the top, and we can calculate the amount of energy it takes to reach the final space from those spaces. We continue calculating these values until we fill our energy cost matrix, after which the real starting space now holds how much energy it will take to reach the end. Then the solution is the absolute value of the starting tile's energy cost plus 1 because Mr. X must remain at 1 health or above.