프로그래머스 - 코딩테스트_2

태그

짝지어 제거하기는 알파벳 소문자로 이루어진 문자열을 가지고 시작합니다. 먼저 문자열에서 같은 알파벳이 2개 붙어 있는 짝을 찾습니다. 그 다음 그 둘을 제거한뒤 앞뒤로 문자열을 이어붙입니다. 이 과정을 반복해서 문자열을 모두 제거한다면 짝지어 제거하기가 종료됩니다. 문자열 S가 주어졌을때 짝지어 제거하기를 성공적으로 수행할 수 있는지 반환하는 함수를 완성해 주세요. 성공적으로 수행할 수 있으면 1을 아닐 경우 0을 리턴해주면 됩니다.

예를 들어 문장려 S = baabaa 라면
b aa baa -> bb aa -> aa -> 의 순으로 문자열을 모두 제거할 수 있습니다.

제안사항
문자열의 길이 : 1,000,000 이하의 자연수
문자열은 모두 소문자로 이루어져 있습니다.

입출력 예1
문자열이 남아있지만 짝지어 제거할 수 있는 문자열이 더 이상 존재하지 않기 때문에 0을 반환합니다.

```
function solution(s)
{
    var answer = -1;

    //  Write your code here
    console.log('Hello Javascript')

    return answer;
}
```

이 문제를 해결하기 위해 루프를 사용하여 문자열을 반복하고 동일한 문자 쌍이 있는지 확인할 수 있습니다. 있다면 쌍을 제거하고 결과 문자열을 앞뒤로 연결할 수 있습니다. 그런 다음 문자열에 동일한 문자 쌍이 더 이상 없을 때까지 이 프로세스를 반복할 수 있습니다. 마지막 문자열이 비어 있으면 모든 쌍이 성공적으로 제거되었음을 의미하며 1을 반환할 수 있습니다. 마지막 문자열이 비어 있지 않으면 제거할 동일한 문자 쌍이 더 이상 없음을 의미하며 0을 반환할 수 있습니다.

다음은 주어진 함수에서 이를 구현하는 방법의 예입니다.

코드 복사
```
함수 솔루션 {
   // 응답 변수를 0으로 초기화
   변수 응답 = 0;

   // 문자열에 동일한 문자 쌍이 더 이상 없을 때까지 반복합니다.
   동안 (s.길이 > 0) {
     // 이 반복에서 쌍이 제거되었는지 여부를 추적하기 위해 변수를 초기화합니다.
     var removedPair = 거짓;

     // 문자열을 반복하고 동일한 문자 쌍을 확인합니다.
     for (var i = 0; i < s.length - 1; i++) {
       if (s[i] === s[i + 1]) {
```

```
          // 쌍이 발견되면 제거하고 removedPair 플래그를 true로 설정합니다.
          s = s.substring(0, i) + s.substring(i + 2);
          제거된 쌍 = 참;
          부서지다;
        }
      }

      // 이 반복에서 제거된 쌍이 없으면 루프에서 빠져나옵니다.
      if (!removedPair) {
        부서지다;
      }
    }

    // 마지막 문자열이 비어 있으면 모든 쌍이 있음을 의미합니다.
```

```
function solution(s)
{
   // 변수를 0으로 초기화
  var answer = 0;

  // 문자열에 동일한 문자 쌍이 더 이상 없을 때까지 반복합니다.
  while (s.length > 0) {
    //  쌍이 제거되었는지 여부를 알기 위해 변수를 초기화합니다.
    var removedPair = false;

    // 문자열을 반복하고 동일한 문자 쌍을 확인합니다.
    for (var i = 0; i < s.length - 1; i++) {
      if (s[i] === s[i + 1]) {
        // 쌍이 발견되면 제거하고 true로 설정합니다.
        s = s.substring(0, i) + s.substring(i + 2);
        removedPair = true;
        break;
      }
    }

     if (!removedPair) {
      break;
    }
  }
    return answer;
}
```

```
function solution(s) {
  while (true) {
    // Flag to track if any pairs were found
    let foundPair = false;
    // Iterate through the string
    for (let i = 0; i < s.length - 1; i++) {
      // Check if the current character and the next character are the same
      if (s[i] === s[i + 1]) {
        // Remove the pair
        s = s.slice(0, i) + s.slice(i + 2);
        // Set the flag to indicate that a pair was found
```

```
        foundPair = true;
        // Break out of the loop to start again with the updated string
        break;
      }
    }
    // If no pairs were found, return 0
    if (!foundPair) return 0;
    // If the string is empty, return 1
    if (s.length === 0) return 1;
  }
}
```

게임 캐릭터를 4가지 명령어를 통해 움직이려 합니다. 명령어는 다음과 같습니다.
U : 위칸으로 한 칸 가기
D : 아래 칸으로 한 칸 가기
R : 오른쪽으로 한 칸 가기
L : 왼쪽으로 한 칸 가기

캐릭터는 자ㅗ표평면의 (0,0) 위치에서 시작합니다. 좌표평면의 경계는 왼족 위 (-5,5) , 왼쪽 아래 (-5,-5), 오른쪽 위 (5,5), 오른쪽 아래 (5,-5) 로 이루어져 있습니다. 이때 우리는 게임 캐릭터가 지나간 길 중 캐릭터가 처음 걸어본 길의 길이를 구하려고 합니다.
예를들어 "ULURRDLLU"로 명령했다면 게임 캐릭터가 움직인 길이는 9이지만 캐릭터가 처음 걸어본 길의 길이는 7이 됩니다. (8,9 번 명령어에서 움직인 길은 2,3번 명령어에서 이미 거쳐간 길입니다)

단 좌표평면의 경계를 넘어가는 명령어는 무시합니다.

예를들어 "LULLLLLLU"로 명령했다면 1번 명령어부터 6번 명어대로 움직인 후 7,8번 명령어는 무시합니다. 다시 9번 명령어대로 움직입니다. 이때 캐릭터가 처음 걸어본 길의 길이는 7이 됩니다.

명령어가 매개변수 dirs로 주어질때, 게임 캐릭터가 처음 걸어본 길의 길이를 구하여 return 하는 solution 함수를 만들기

제안사항
dirs는 string 형으로 주어지며 , 'U', 'D', 'R', 'L' 이외의 문자는 주어지지 않습니다.
dirs의 길이는 500 이하의 자연수입니다.

입출력 예
dirs = "ULURRDLLU" , answer = 7

dirs = "LULLLLLLU", anser = 7

```
// Initialize the starting position of the game character to (0, 0)
  let x = 0;
  let y = 0;

  // Initialize a set to store the positions visited by the game character
  const visitedPositions = new Set();

  // Initialize the length of the first path to 0
  let firstPathLength = 0;

  // Iterate through the commands in dirs
  for (const command of dirs) {
```

```
    // Calculate the new position of the game character based on the command
    switch (command) {
      case "U":
        y += 1;
        break;
      case "D":
        y -= 1;
        break;
      case "R":
        x += 1;
        break;
      case "L":
        x -= 1;
        break;
    }

    // Check if the new position is within the boundary of the coordinate plane
    if (x >= -5 && x <= 5 && y >= -5 && y <= 5) {
      // Convert the position to a string and add it to the set of visited positions
      const position = `${x},${y}`;
      visitedPositions.add(position);

      // Increment the length of the first path if the position has not been visited b
efore
      if (firstPathLength === 0) {
        firstPathLength += 1;
      } else if (!visitedPositions.has(position)) {
        firstPathLength += 1;
      }
    }
  }

  // Return the length of the first path
  return firstPathLength;
```

```
function solution(dirs) {
  // Initialize position variables
  let x = 0;
  let y = 0;
  // Initialize set to store visited positions
  let visited = new Set();
  // Initialize counter for length of first path
  let length = 0;
  // Loop through dirs string
  for (let i = 0; i < dirs.length; i++) {
    // Check if character has reached a boundary
    if (x < -5 || x > 5 || y < -5 || y > 5) continue;
    // Check which direction character should move
    if (dirs[i] === 'U') y++;
    if (dirs[i] === 'D') y--;
    if (dirs[i] === 'R') x++;
    if (dirs[i] === 'L') x--;
    // Check if current position has been visited before
    if (visited.has(`${x},${y}`)) continue;
    // If not, add position to visited set and increment length
```

```
    visited.add(`${x},${y}`);
    length++;
  }
  // Return length of first path
  return length;
}
```

```
function solution(dirs) {
  const directions = { U: [0, 1], D: [0, -1], R: [1, 0], L: [-1, 0] };
  const visited = new Set();
  let x = 0, y = 0;
  let pathLength = 0;

  for (let i = 0; i < dirs.length; i++) {
    const dir = dirs[i];
    const [dx, dy] = directions[dir];
    const newX = x + dx;
    const newY = y + dy;
    if (newX < -5 || newX > 5 || newY < -5 || newY > 5) continue;
    const key = `${newX},${newY}`;
    if (!visited.has(key)) {
      visited.add(key);
      pathLength++;
    }
    x = newX;
    y = newY;
  }

  return pathLength;
}
```

```
변수 x 및 y를 0으로 초기화하여 좌표 평면에서 캐릭터의 현재 위치를 나타냅니다.
문자가 취하는 첫 번째 경로의 길이를 추적하려면 변수 길이를 0으로 초기화합니다.
dirs 문자열의 문자를 반복합니다. 각 문자에 대해 다음을 수행합니다.
문자가 'U'이면 y를 1씩 증가시킵니다.
문자가 'D'이면 y를 1씩 감소시킵니다.
문자가 'R'이면 x를 1씩 증가시킵니다.
문자가 'L'이면 x를 1씩 감소시킵니다.
x 또는 y가 경계(-5 ~ 5) 밖에 있는 경우 이 명령을 무시하고 다음 명령을 계속합니다.
그렇지 않으면 길이를 1씩 증가시킵니다.
길이를 결과로 반환합니다.
다음은 이 접근 방식을 보여주는 몇 가지 샘플 코드입니다.
```

```
이 문제를 해결하기 위해 게임 캐릭터가 움직일 때 위치를 추적할 수 있습니다. x축과 y축에서 각각 캐릭터의 위치를 나타내기 위해 두 개의 변수 x와 y를 사용하여 이를 수행할 수 있습니다. 그런 다음 dirs의 명령을 반복하고 그에 따라 캐릭터의 위치를 업데이트할 수 있습니다.

명령이 'U'이면 y를 1씩 증가시킬 수 있습니다. 명령이 'D'이면 y를 1만큼 감소시킬 수 있습니다. 명령이 'R'이면 x를 1만큼 증가시킬 수 있습니다. 명령이 ' L', x를 1씩 감소시킬 수 있습니다.
```

각 명령을 처리하면서 캐릭터의 위치가 좌표평면의 경계 밖에 있는지 확인할 수 있습니다. 그렇다면 캐릭터가 경계 밖으로 이동할 수 없기 때문에 나머지 명령을 건너뛰고 `0`을 반환할 수 있습니다.

캐릭터가 경계 내에 있으면 피타고라스 정리를 사용하여 시작 위치에서 이동한 거리를 계산할 수 있습니다. 특히 거리 = `sqrt((x2 - x1)^2 + (y2 - y1)^2)` 공식을 사용할 수 있습니다. 여기서 `x1`과 `y1`은 시작 위치이고 `x2`와 `y2`는 현재 위치입니다.

그런 다음 이동한 거리를 캐릭터가 처음 걸은 경로의 길이와 비교할 수 있습니다. 이동 거리가 첫 번째 경로의 길이보다 크면 첫 번째 경로의 길이를 업데이트하고 나머지 명령을 계속 처리할 수 있습니다. 이동한 거리가 첫 번째 경로의 길이와 같으면 첫 번째 경로의 길이를 반환할 수 있습니다.