동적 계획법을 사용한 가장 긴 회문 부분 수열

가장 긴 회문 부분 시퀀스(LPS) 문제는 회문이기도 한 문자열의 가장 긴 부분 시퀀스를 찾는 것입니다.

문제는 찾는 문제와 다릅니다. 가장 긴 회문 부분 문자열. 부분 문자열과 달리, 하위 시퀀스 원래 문자열 내에서 연속적인 위치를 차지할 필요가 없습니다.

예를 들어 시퀀스를 고려하십시오. `ABBDCACB`.

```
The length of the longest palindromic subsequence is 5
The longest palindromic subsequence is BCACB
```

이 문제를 연습하세요

아이디어는 사용하는 것입니다 재귀 이 문제를 해결하기 위해. 아이디어는 문자열의 마지막 문자를 비교하는 것입니다. `X[i…j]` 첫 번째 캐릭터와 함께. 두 가지 가능성이 있습니다.

1. 문자열의 마지막 문자가 첫 번째 문자와 같으면 회문에 첫 번째 및 마지막 문자를 포함하고 나머지 부분 문자열에 대해 반복 `X[i+1, j-1]`.
2. 문자열의 마지막 문자가 첫 번째 문자와 다른 경우 두 값 중 최대값을 반환합니다.
   - 마지막 문자를 제거하고 나머지 부분 문자열에 대해 반복 `X[i, j-1]`.
   - 첫 번째 문자를 제거하고 나머지 부분 문자열에 대해 반복 `X[i+1, j]`.

이것은 시퀀스의 가장 긴 반복 부분 시퀀스의 길이를 찾는 데 다음과 같은 재귀 관계를 생성합니다. `X`:

```
            | 1                                 (if i = j)
LPS[i…j] =  | LPS[i+1…j-1] + 2                  (if X[i] = X[j])
            | max (LPS[i+1…j], LPS[i…j-1])      (if X[i] != X[j])
```

알고리즘은 C++, Java, Python에서 다음과 같이 구현할 수 있습니다. 솔루션은 시퀀스의 가장 긴 반복 부분 시퀀스의 길이를 찾습니다. `X` 위의 관계식을 재귀적으로 사용합니다.

C++ Java Python

```cpp
#include <iostream>
#include <string>
#include <algorithm>
using namespace std;

// 가장 긴 회문 부분 시퀀스의 길이를 찾는 함수
```

```
{
    // 기본 조건
    if (i > j) {
        return 0;
    }

    // 문자열 `X`에 하나의 문자만 있으면 회문입니다.
    if (i == j) {
        return 1;
    }

    // 문자열의 마지막 문자가 첫 번째 문자와 같으면
    if (X[i] == X[j])
    {
        // 회문에 첫 번째와 마지막 문자를 포함합니다.
        // 나머지 부분 문자열 `X[i+1, j-1]`에 대해 반복
        return findLongestPalindrome(X, i + 1, j - 1) + 2;
    }

    /*
      문자열의 마지막 문자가 첫 번째 문자와 다른 경우
        1. 마지막 문자를 제거하고 나머지 부분 문자열에 대해 반복
           `X[i,j-1]`
        2. 첫 번째 문자를 제거하고 나머지 부분 문자열에 대해 반복
           `X[i+1,j]`
    */

    // 두 값 중 최대값 반환
    return max(findLongestPalindrome(X, i, j - 1), findLongestPalindrome(X, i + 1, j));
}

int main()
{
    string X = "ABBDCACB";
    int n = X.length();

    cout << "The length of the longest palindromic subsequence is "
         << findLongestPalindrome(X, 0, n - 1);

    return 0;
}
```

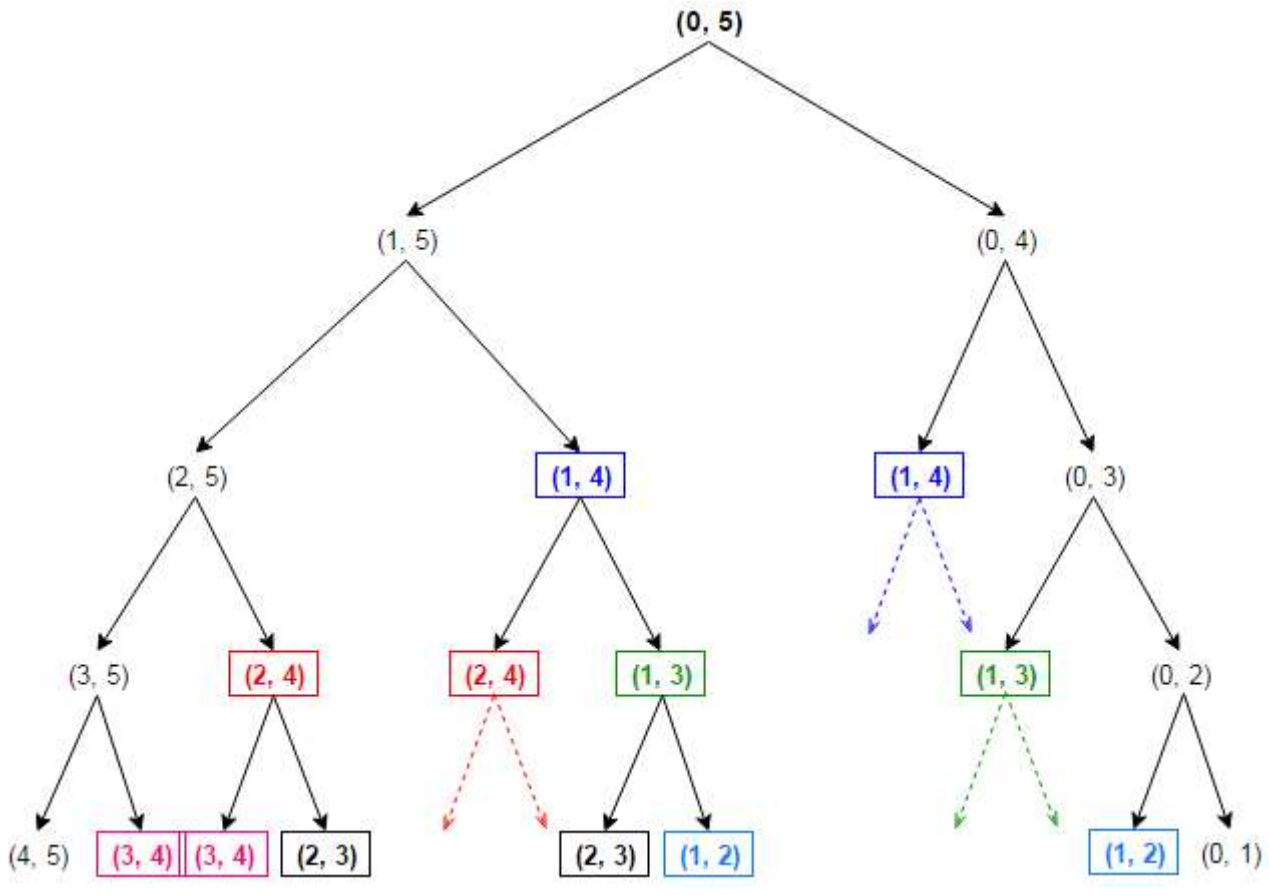
다운로드 코드 실행

산출:

```
The length of the longest palindromic subsequence is 5
```

위 솔루션의 최악의 경우 시간 복잡도는 기하급수적입니다. $O(2^n)$, 어디 `n` 입력 문자열의 길이입니다. 최악의 경우는 반복되는 문자가 없을 때 발생합니다. `X` (즉, LPS 길이는 1), 각 재귀 호출은 두 개의 재귀 호출로 끝납니다. 또한 호출 스택을 위한 추가 공간이 필요합니다.

LPS 문제는 최적의 하부구조 또한 전시 겹치는 하위 문제. 모든 고유한 문자를 갖는 길이 6의 시퀀스에 대한 재귀 트리를 고려해 보겠습니다. `ABCDEF` , LPS 길이가 1입니다.



보시다시피 동일한 하위 문제(동일한 색상으로 강조 표시됨)가 반복적으로 계산됩니다. 우리는 최적의 하위 구조와 중복되는 하위 문제가 있는 문제가 동적 프로그래밍에 의해 해결될 수 있다는 것을 알고 있습니다. 여기서 하위 문제 솔루션은 다음과 같습니다. *메모*계산된 것이 아니라 크기화된 것입니다. 이 방법은 C++, Java 및 Python에서 아래에 설명되어 있습니다.

C++ Java Python

```cpp
#include <iostream>
#include <string>
#include <unordered_map>
using namespace std;

// 가장 긴 회문 부분 시퀀스의 길이를 찾는 함수
// 하위 문자열 `X[i…j]`
int findLongestPalindrome(string X, int i, int j, auto &lookup)
{
    // 기본 조건
    if (i > j) {
        return 0;
    }

    // 문자열 `X`에 하나의 문자만 있으면 회문입니다.
    if (i == j) {
        return 1;
    }

    // 입력의 동적 요소에서 고유한 맵 키를 구성합니다.
    string key = to_string(i) + "|" + to_string(j);

    // 하위 문제가 처음 발견되면 해결하고
    // 결과를 맵에 저장
    if (lookup.find(key) == lookup.end())
    {
        /* 문자열의 마지막 문자가 첫 번째 문자와 같으면
           회문에 첫 번째와 마지막 문자를 포함하고 반복
           나머지 부분 문자열 `X[i+1, j-1]`에 대해 */

        if (X[i] == X[j]) {
            lookup[key] = findLongestPalindrome(X, i + 1, j - 1, lookup) + 2;
        }
        else {

            /* 문자열의 마지막 문자가 처음과 다른 경우
                캐릭터
                1. 마지막 문자를 제거하고 나머지 부분 문자열에 대해 반복
                   `X[i,j-1]`
                2. 첫 번째 문자를 제거하고 나머지 부분 문자열에 대해 반복
                   `X[i+1,j]`
                3. 두 값의 최대값 반환 */
```

```
        }
    }

    // 지도에서 하위 문제 솔루션을 반환합니다.
    return lookup[key];
}

int main()
{
    string X = "ABBDCACB";
    int n = X.length();

    // 하위 문제에 대한 솔루션을 저장할 맵 생성
    unordered_map<string, int> lookup;

    cout << "The length of the longest palindromic subsequence is " <<
        findLongestPalindrome(X, 0, n - 1, lookup);

    return 0;
}
```

다운로드 코드 실행

산출:

```
The length of the longest palindromic subsequence is 5
```

위 솔루션의 시간 복잡도는 $O(n^2)$ 그리고 요구한다 $O(n^2)$ 여분의 공간, 어디에 `n` 입력 문자열의 길이입니다.

그러나 위에서 논의한 두 가지 방법 모두 가장 긴 회문 부분 시퀀스 길이만 찾고 가장 긴 회문 부분 시퀀스 자체는 인쇄하지 않습니다.

Longest Palindromic 부분 시퀀스를 어떻게 인쇄할 수 있습니까?

가장 긴 회문 부분수열 문제는 최장 공통 부분 수열(LCS) 문제. 아이디어는 주어진 문자열의 LCS를 역으로 찾는 것입니다. 즉, `LCS(X, reverse(X))` 그리고 가장 긴 공통 부분 수열은 가장 긴 회문 부분 수열이 될 것입니다.

다음은 이를 시연하는 C++, Java 및 Python 프로그램입니다.

C++ Java Python

```
#include <iostream>
#include <string>
#include <algorithm>
using namespace std;

// 문자열 `X[0…m-1]` 및 `Y[0…n-1]`의 LCS를 찾는 함수
string findLongestPalindrome(string X, string Y, int m, int n, auto &lookup)
{
    // 시퀀스의 끝에 도달하면 빈 문자열을 반환합니다.
    if (m == 0 || n == 0) {
        return string("");
    }

    // `X`와 `Y`의 마지막 문자가 일치하는 경우
    if (X[m - 1] == Y[n - 1])
    {
        // 현재 문자(`X[m-1]` 또는 `Y[n-1]`)를 LCS에 추가합니다.
        // 하위 문자열 `X[0…m-2]` 및 `Y[0…n-2]`
        return findLongestPalindrome(X, Y, m - 1, n - 1, lookup) + X[m - 1];
    }

    // 그렇지 않으면 `X`와 `Y`의 마지막 문자가 다른 경우

    // 현재 셀의 맨 위 셀이 왼쪽보다 더 큰 값을 가지고 있는 경우
    // 셀, 문자열 `X`의 현재 문자를 삭제하고 LCS를 찾습니다.
    // 하위 문자열 `X[0…m-2]`, `Y[0…n-1]`
```

```
    }

    // 현재 셀의 왼쪽 셀이 위쪽보다 더 큰 값을 가지고 있는 경우
    // 셀, 문자열 `Y`의 현재 문자를 삭제하고 LCS를 찾습니다.
    // 하위 문자열 `X[0…m-1]`, `Y[0…n-2]`

    return findLongestPalindrome(X, Y, m, n - 1, lookup);
}

// 하위 문자열 `X[0…n-1]` 및 `Y[0…n-1]`의 LCS 길이를 찾는 함수
int LCSLength(string X, string Y, int n, auto &lookup)
{
    // 조회 테이블의 첫 번째 행과 첫 번째 열은 이미 0입니다.

    // 룩업 테이블을 상향식으로 채움
    for (int i = 1; i <= n; i++)
    {
        for (int j = 1; j <= n; j++)
        {
            // `X`와 `Y`의 현재 문자가 일치하는 경우
            if (X[i - 1] == Y[j - 1]) {
                lookup[i][j] = lookup[i - 1][j - 1] + 1;
            }
            // 그렇지 않으면 `X`와 `Y`의 현재 문자가 일치하지 않는 경우
            else {
                lookup[i][j] = max(lookup[i - 1][j], lookup[i][j - 1]);
            }
        }
    }

    return lookup[n][n];
}

int main()
{
    string X = "ABBDCACB";

    int m = X.length();

    // lookup[i][j]는 하위 문자열 `X[0…i-1]` 및 `Y[0…j-1]`의 LCS 길이를 저장합니다.
    vector<vector<int>> lookup(m + 1, vector<int>(m + 1));

    // 문자열 `Y`는 `X`의 반대입니다.
    string Y = X;
    reverse(Y.begin(), Y.end());

    // LCS를 사용하여 LPS의 길이를 찾습니다.
    cout << "The length of the longest palindromic subsequence is "
         << LCSLength(X, Y, m, lookup) << endl;

    // 조회 테이블을 사용하여 LPS를 인쇄합니다.
    cout << "The longest palindromic subsequence is "
         << findLongestPalindrome(X, Y, m, m, lookup);

    return 0;
}
```

다운로드 코드 실행

Output:

```
The length of the longest palindromic subsequence is 5
The longest palindromic subsequence is BCACB
```

위 솔루션의 시간 복잡도는 $O(n^2)$ 그리고 요구한다 $O(n^2)$ 여분의 공간, 어디에 `n` 입력 문자열의 길이입니다.

운동: 위의 재귀 하향식 버전에 대한 상향식 솔루션을 작성하십시오.

Dynamic Programming, String

Algorithm, Bottom-up, Medium, Microsoft, Recursive, Top-down