백준 9177번 - 단어 섞기
문제
입력
출력
예제 입력1
예제 출력1
출처
알고리즘 분류
접근 방법
소스코드

백준 9177번 - 단어 섞기

시간제한	메모리 제한	제출	정답	맞은 사람	정답 비율
1초	256MB	3064	809	578	27.709%

문제

세 개의 단어가 주어졌을때, 꿍은 첫 번째 단어와 두 번째 단어를 섞어서 세 번째 단어를 만들 수 있는지 궁금해졌다. 첫 번째와 두 번째 단어는 마음대로 섞어도 되지만 원래의 순서는 섞여서는 안 된다. 다음과 같은 경우를 생각해보자.

- 첫 번째 단어 : cat
- 두 번째 단어 : tree
- 세 번째 단어 : tcraete

보면 알 수 있듯이, 첫 번째 단어와 두 번째 단어를 서로 섞어서 세 번째 단어를 만들 수 있다. 아래와 같이 두 번째 예를 들어보자.

- 첫 번째 단어 : cat
- 두 번째 단어 : tree
- 세 번째 단어 : catrtee

이 경우 역시 가능하다. 그렇다면 "cat"과 "tree"로 "cttaree"를 형성하는건 불가능하다는걸 눈치챘을 것이다.

입력

입력의 첫 번째 줄에는 1부터 1000까지의 양의 정수 하나가 주어지며 데이터 집합의 개수를 뜻한다. 각 데이터집합의 처리과정은 동일하다고 하자. 각 데이터집합에 대해, 세 개의 단어로 이루어져 있으며 공백으로 구분된다. 모든 단어는 대문자 또는 소문자로만 구성되어 있다. 세 번째 단어의 길이는 항상 첫 번째 단어와 두 번째 단어의 길이의 합이며 첫 번째 단어와 두 번째 단어의 길이는 1~200이다.

출력

각 데이터집합에 대해 다음과 같이 출력하라.

만약 첫 번째 단어와 두 번째 단어로 세 번째 단어를 형성할 수 있다면

```
Data set n: yes
```

과 같이 출력하고 만약 아니라면

```
Data set n: no
```

과 같이 출력하라. 물론 n은 데이터집합의 순번으로 바뀌어야 한다. 아래의 예제 출력을 참고하라.

예제 입력1

```
3
cat tree tcraete
cat tree catrtee
cat tree cttaree
```

예제 출력1

```
Data set 1: yes
Data set 2: yes
Data set 3: no
```

출처

출처

알고리즘 분류

- 다이나믹 프로그래밍
- 그래프 이론
- 문자열
- 그래프 탐색

접근 방법

Naive하게 먼저 접근을 해보자.

각 테스트케이스에서 입력된 값들을 `word[0], word[1], word[2]` 라고 정의한다.

문제의 핵심은 아래와 같다.

1. `word[0]`의 모든 문자들이 `word[2]`에 오름차순으로 매핑되는가?
2. `word[1]`의 모든 문자들이 `word[2]`에 오름차순으로 매핑되는가?

위 2가지 조건을 모두 만족할 경우에만, "yes"를 출력하는 것이다.

1번 조건을 고려해보자면

`word[2]`의 i번째 인덱스의 문자가 `word[0]`의 j번째 문자에 대응된다면 `word[2]`의 $i+1$번째 인덱스부터

다시 `word[0]`의 $j+1$번째 문자를 매핑하는 방식으로 진행하는 것이다.

2번 조건을 고려해보자면

`word[2]`의 i번째 인덱스의 문자가 `word[1]`의 j번째 문자에 대응된다면 `word[2]`의 $i+1$번째 인덱스부터

다시 `word[1]`의 $j+1$번째 문자를 매핑하는 방식으로 진행하는 것이다.

`word[2]`의 i번째 문자를 `word[0]`과 `word[1]`이 선택하는 경우는 $\{(0,1),(1,0)\}$ 임이 자명하다.

(두 문자가 동일한 index에 매핑될 수 없으므로)

최종적으로 백트래킹을 통하여 $O(2^N)$로 Naive하게 접근하여 작은 범위에 대하여 답을 구할 수 있다. 하지만 문제의 조건 상 N은 최대 400이므로 해당 방법으로는 문제를 해결할 수 없다.

하지만, 우리는 Naive한 접근 방법에서 통찰을 얻을 수 있다.

개선을 위하여 불필요한 탐색을 제거하는 방법으로 문제풀이를 개선해보자.

`word[2]`의 i번째 문자까지 `word[0]`에서 j번째 문자까지 매핑이 되었고 `word[1]`에서 k번째 문자까지 매핑이 되어 끝까지 탐색을 진행하였을 때(즉, `word[2]`의 끝까지 살펴보았을 때) `word[0]`의 모든 문자가 오름차순으로 매핑되고 `word[1]`의 모든 문자가 오름차순으로 매핑되는 경우가 존재한다고 가정해보자.

위 상태를 $dp[i][j][k]$로 정의하자.

그럴 경우 서로 다른 2개의 탐색 지점(u, v)에서 탐색을 진행하여 동일한 state인 $dp[i][j][k]$ 에 도달하였을 때 뒷 부분의 탐색은 이전에 탐색한 경로와 동일하므로 이것을 메모이제이션 해 둔다면 추가적인 탐색 없이도 u, v 에서도 $dp[i][j][k]$와 동일하게 매핑 여부를 $O(1)$에 알 수 있다.

테이블을 $dp[i][j][k]$ 로 정의하게 될 경우, 최악의 경우 $i*j*k$ 크기의 테이블을 모두 채워야하고 이에 필요한 최대 연산량은 400 * 200 * 200 으로 대략 10^7 이다. 테스트 케이스의 수는 최대 10^3 이므로, 전체 문제를 해결하기 위해 최대 10^{10}의 시간이 소요된다.

이를 해결하기 위해 테이블을 좀 더 개선해야할 필요가 있다.

단순하게 차원을 낮춰 테이블을 아래와 같이 정의해보자.

$dp[i][j]$:= `word[0][i]까지 매핑되고 word[1][j]까지 매핑된다고 하였을 때, 해당 경로로 탐색을 진행할 때 word[2]에 모든 문자가 매핑될 수 있는가?`

문제 해결에는 아무런 지장이 없는 것을 알 수 있다.

이렇게 최적화하므로서 최악의 경우 $i * j$ 크기의 테이블을 채우는데 필요한 최대 연산량을 대략 10^4 정도로 줄일 수 있다.

소스코드

```cpp
#define FASTIO cin.tie(0)->sync_with_stdio(false), cout.tie(0)
////////////////////////////////////////////////////////////////
#include <bits/stdc++.h>
using namespace std;
string word[3];
int dp[201][201];
int solve(int i, int j, int k){
    int &ret = dp[j][k];
    if(ret != -1)
        return ret;
    if(i == word[2].size()){
        if(j == word[0].size() && k == word[1].size())
            return ret = 1;
        return ret = 0;
    }
    ret = 0;
    // 현재 i번째 word[3]까지 word[0] 가 j번째, word[1]이 k번째 까지 매칭되었을 때
    // 해당 단어를 끝까지 완성할 수 있는가?
    if(word[2][i] == word[0][j])
        ret |= solve(i+1, j+1, k);
    if(!ret && word[2][i] == word[1][k])
        ret |= solve(i+1, j, k+1);
    return ret;
}
int main(void){
    FASTIO;
////////////////////////////////////////////////////////////////
    int T;
    cin >> T;
    for(int i=1; i<=T; i++){
        memset(dp, -1, sizeof(dp));

        cin >> word[0] >> word[1] >> word[2];
        solve(0, 0, 0);
        cout << "Data set " << i << ": ";
        if(dp[word[0].size()][word[1].size()] == -1)
            cout << "no\n";
        else
            cout << "yes\n";
    }
```

```
41      return 0;
42  }
```