고급소프트웨어실습 과제 9

> **학번: 20171665 / 이름: 이선호**

> 실습 3이 Greedy algorithm과 Dynamic programming 중 어디에 해당된다고 할 수 있는가? 이유를 설명하시오.

실습 3은 기본적으로 **동적계획법(Dynamic programming)**을 사용한 문제로 볼 수 있다. 동적계획법은 주어진 문제를 간단한 부분 문제로 나누어서 푸는 방법이며, 특히 메모이제이션(Memoization) 또는 태뷸레이션(Tabulation)을 통해 이전에 구한 부분 문제의 답을 메모리에 저장한다. 실습 3에서는 전체 수열의 어느 특정 위치부터 연속으로 현재 보고 있는 위치까지 더해온 합과 현재 위치에서 새롭게 연속 부분 수열을 택해 시작해가기 위해 고르는 현재 위치의 원소 값을 비교한다. 이 둘 중에서 큰 값을 현재 위치까지 봤을 때 가장 큰 연속 부분 수열의 합으로 메모이제이션을 통해 저장한다. 구체적으로 사용한 메모이제이션 배열의 정의를 쓰면 다음과 같다.

```
1   sum[i]: 처음부터 i번째 위치까지 봤을 때 가장 큰 연속 부분 수열의 합
```

매번 위치를 이동하면서 탐색할 때마다 가장 큰 연속 부분 수열의 합으로 올 수 있는 가장 큰 값을 찾아 업데이트하므로 매 단계마다 탐욕적으로 최적의 답을 선택한다는 점에서 탐욕 알고리즘(Greedy algorithm)으로 볼 수도 있지 않을까 하는 생각이 들 수 있지만, 이번 실습에서는 점화식을 세워서 매번 부분 문제의 답을 배열 등 메모리에 저장하는 메모이제이션을 통해 중복된 하위 문제의 답을 사용하여 최종적인 답을 찾아간다는 점에 주목한다면 동적계획법(Dynamic programmin)으로 보는 것이 바람직하다고 본다.

그러나 실습 3을 메모이제이션을 하지 않고 오로지 탐욕 알고리즘(Greed algorithm)으로만 풀 수도 있는데, 이는 직전까지 본 가장 큰 연속 부분 수열의 합이 음수이면서 동시에 현재 위치의 원소 값이 양수일 때는 현재까지 본 가장 큰 연속 부분 수열을 현재 위치에서 새롭게 시작하는 것이 유리하므로 그 연속 부분 수열의 합을 현재 원소의 값으로 반영한다. 이 경우 굳이 매 위치마다 가장 큰 연속 부분 수열의 합을 저장할 필요가 없으므로 동적계획법(Dynamic programming)이 아니다.

> 실습에서 다루지 않은 문제 중에 Dynamic programming을 사용했을 때 더 효율적으로 풀 수 있는 문제를 2개 들고, 알고리즘을 설명하시오. brute force 방식에 비하여 DP를 사용하는 경우 어느 정도 빨라지는지도 설명하시오.

팰린드롬 개수 구하기

[문제 링크] https://www.acmicpc.net/problem/14517

이 문제는 동적계획법(Dynamic programming)을 사용했을 때 더 효율적으로 해결할 수 있으며, 2017년 고려대학교 프로그래밍 대회의 G2번으로 출제되었던 문제이다. 또한 개인적으로 블로그에 풀이법을 정리하여 올린 적이 있다. (블로그 링크)

문제에서 문자열 S가 주어지는데, 이 부분 수열 중에서 팰린드롬이 될 수 있는 부분수열의 개수를 출력해야 하는데, 주의할 점은 연속된 부분 문자열이 아니라 전체 문자열에서 일부를 임의로 골라 순서는 유지한 채로 만든 수열이라는 것이다. 문자열 S의 최대 길이는 $1,000$이다. 동적 계획법 풀이를 사용하고자 메모이제이션을 위한 배열을 다음과 같이 정의할 수 있다.

```
dp[i][j]: i번째 문자에서 j번째 문자까지의 부분수열 중 팰린드롬을 만족하는 것의 개수
```

기본적으로 i번째에서 j번째 문자까지의 부분 수열인 팰린드롬의 개수는 i번째부터 $j-1$번째 문자까지의 부분 수열인 팰린드롬 개수와 $i+1$번째부터 j번째 문자까지의 부분수열인 팰린드롬의 개수를 더하여 구할 수 있다.

$$dp[i][j] = dp[i][j-1] + dp[i+1][j] \tag{1}$$

그러나 i번째 문자와 j번째 문자가 서로 같은지 다른지에 따라 합집합의 법칙에 의해 중복으로 구해지는 부분을 제외하는 부분이 달라진다.

i번째 문자와 j번째 문자가 서로 같은 경우에는 $i+1$번째에서 $j-1$번째 사이의 구간에서 나오는 부분수열 중 팰린드롬인 것의 양 끝에 rk각각 i번째 문자와 j번째 문자를 붙여서 새로운 팰린드롬을 만들 수 있다. 즉, `S[i][j] == S[j]`일 때, `dp[i + 1][j - 1]`로 구해지는 팰린드롬의 개수를 제외하지 않아도 된다. 또한 i번째 문자와 j번째 문자를 골라서 또 하나의 새로운 팰린드롬을 구할 수 있으므로 경우의 수에 1을 더해야 한다. 이를 식으로 정리하면 다음과 같다.

$$dp[i][j] = dp[i][j-1] + dp[i+1][j] + 1 \qquad \text{if} \quad s[i] = s[j] \tag{2}$$

그러나 i번째 문자와 j번째 문자가 서로 다른 경우에는 $i+1$번째에서 $j-1$번째 사이의 구간에서 나오는 부분수열 중 팰린드롬인 것의 양 끝에 rk각각 i번째 문자와 j번째 문자를 붙여서 새로운 팰린드롬을 만들 수 없으므로 이 중복된 경우의 수를 제외해 줘야 한다. 즉, `S[i][j] == S[j]`일 때, `dp[i + 1][j - 1]`로 구해지는 팰린드롬의 개수인 교집합 부분을 한 번 빼준다. 이를 식으로 정리하면 다음과 같다.

$$dp[i][j] = dp[i][j-1] + dp[i+1][j] - dp[i+1][j-1] \quad \text{if} \quad s[i] \neq s[j] \tag{3}$$

Brute Force 대신 Dynamic Programming 사용 시 장점

이 문제를 브루트 포스(Brute force)로 해결하려면 시간복잡도가 굉장히 커진다. 문제에서 주어질 수 있는 문자열 S의 최대 문자열의 길이가 $1,000$일 때, 각 위치의 원소를 부분 수열로 포함시키냐 그렇지 않느냐에 따라 부분 수열로 만들 수 있는 경우의 수는 2^{1000}이다. 그런데 이 많은 부분수열을 일일이 보면서 팰린드롬인지 확인할 수는 없다. 문자열 S의 길이가 l일 때 브루트 포스 사용 시 시간복잡도는 $O(2^l)$이다.

반면에 앞서 소개한 동적계획법 알고리즘을 사용하면 왼쪽 원소의 위치와 오른쪽 원소의 위치를 하나씩 이동해가면서 매번 구한 최적의 답을 배열로 저장하므로 중복되는 케이스를 재탐색할 필요가 없다. 그래서 시간복잡도는 $O(l^2)$이다.

환상의 듀엣

[문제 링크] https://www.acmicpc.net/problem/11570

이 문제도 동적계획법(Dynamic programming)을 사용했을 때 더 효율적으로 해결할 수 있으며, 2015년 서강대학교 프로그래밍 대회(SPC)의 Master G번으로 출제되었던 문제이다.. 또한 개인적으로 블로그에 풀이법을 정리하여 올린 적이 있다. (블로그 링크)

N개의 음이 순서대로 적혀있는 악보를 가지고 상덕이와 희원이라는 두 사람이 서로 음을 분담하여 번갈아가면서 노래를 부르려고 한다. 문제에서 두 사람이 서로 N개의 음을 분담했을 때, 각 사람마다 부르는 음 사이의 높이 차를 모두 합한 것이 그 사람이 노래를 할 때 힘든 정도로 정의했다. 이때, 두 사람의 힘든 정도의 합으로 나올 수 있는 최솟값을 구하는 것이 목표이다. N의 최댓값은 $2,000$이고, 동적 계획법 풀이를 사용하고자 메모이제이션을 위한 배열을 다음과 같이 정의할 수 있다.

```
dp[i][j]: 현재 i번째 음을 노래하려고 하는데 직전 음인 i-1번째 음을 노래하지 않은 사람이 j번째 음을 마지막으로 노래했을 때, 두 사람이 힘든 정도의 합의 최솟값
```

이때, 메모이제이션할 배열을 '직전 음을 노래하지 않은 사람이 j번째 음을 마지막으로 노래했을 때'로 정의하지 않고 '상덕이가 j번째 음을 마지막으로 노래했을 때'로 정의해버리면 문제 해결이 어려워질 수 있는데, 그 이유는 상덕이가 직전 음인 $i-1$번째 음을 노래했으면 희덕이가 마지막으로 노래한 음이 어떠한 음인지를 알 수 없다. 그래서 현재 음인 i번째 음을 희덕이가 노래할 경우의 두 사람의 힘든 정도 합의 최솟값을 구할 수 없다.

`a[i]`를 i번째로 들어 온 음의 높이라고 가정하면, `dp` 배열 원소 값을 구하기 위해서는 크게 두 가지 케이스로 분류하여 해결할 수 있다. 첫 번째로 직전 음인 $i-1$번째 음을 연주한 사람이 i번째 음도 연주하는 경우이다. 그러면 관계식을 다음과 같이 정의할 수 있다.

$$dp[i][j] = dp[i+1][j] + |a[i] - a[i-1]| \qquad (j < i-1) \tag{4}$$

두 번째로 j번째 음을 마지막으로 노래한 사람이 i번째 음을 연주하는 경우이다. 이때는 $i-1$번째 음을 노래한 사람이 $i+1$번째 음을 노래할 때 직전 음을 노래하지 않은 사람이 되므로 `dp` 배열의 정의에 의해 $dp[i+1][i-1]$를 업데이트 하게 된다. 이때의 관계식은 다음과 같이 정의할 수 있다.

$$dp[i][j] = dp[i+1][i-1] + |a[i] - a[j]| \qquad (j < i-1) \tag{5}$$

이 문제에서는 bottom-up 방식보다는 top-down 방식이 구현하기 수월하므로 종료 조건을 설정하여 재귀적으로 만드는 게 효율적이다. 그래서 관계식의 오른쪽 항에 dp 배열의 $i+1$ 번째에서 구한 원소 값이 더해지며, 이 항은 재귀적으로 함수로 들어가서 구해진다.

Brute Force 대신 Dynamic Programming 사용 시 장점

이 문제를 브루트포스로 해결하려면 N개의 음을 상덕이와 희원이가 분담할 수 있도록 두 개의 음의 집합으로 나눠야 한다. 모든 음은 두 개의 음의 집합 중 반드시 한 집합에 속해 있어야 하므로 이 두 개의 음의 집합을 구하는 데 있어서 2^N의 경우의 수를 고려해야 한다. 이는 지수적으로 해결해야 하므로 각 사람이 어떠한 음을 노래했는지를 직접 구하여 두 사람의 힘든 정도의 합의 최솟값을 구하는 것은 비효율적이다.

앞서 소개한 동적계획법으로 이 문제를 해결하면 i번째 원소를 가리키는 포인터와 j번째 원소를 가리키는 포인터를 이동해가며 재귀적으로 각 부분 문제의 값을 저장하고, 두 번 다시 부분 문제를 해결하지 않아 중복을 피할 수 있으므로 시간복잡도는 $O(N^2)$이다. 메모이제이션할 배열만 따로 관리한다면 브루트포스를 사용할 때인 $O(2^N)$보다 훨씬 적은 시간 내에 효율적으로 해결할 수 있다.