알고리즘 HW #2

컴퓨터공학부
2013-11431 정현진

1번 문제

m부터 n까지의 문자열에서 가장 긴 substring의 palindrome의 길이를 C(m,n)이라고 하자. 그러면 C(m,n)는 경우가 두 가지로 나눠지는데, 각각 다음과 같다. 이 때 i번째 위치의 문자열의 값은 S_i라고 하자. ($S = \langle s_m s_{m+1} s_{m+2} \cdots s_i \cdots s_n \rangle$)

1. i와 j의 문자열이 서로 같은 경우, 즉 S_i == S_j
2. j와 j의 문자열이 서로 다른 경우, 즉 S_i != S_j

첫 번째 경우를 생각해보면, C(i,j)는 양 쪽 끝이 같기 때문에 안 쪽 substring의 최적해, 즉 C(i+1,j-1)의 값보다 2가 큰 것을 알 수 있다. 두 번째 경우는 왼쪽 substring의 최적해인 C(i, j-1)와 오른쪽 substring의 최적해인 C(i+1, j)의 최적해 중 큰 값을 사용하면 된다. 이를 식으로 정리해 보면

1. C(i,j) = C(i+1, j-1) + 2 if S_i == S_j
2. C(i,j) = max{C(i, j-1), C(i+1, j)} if S_i != S_j

따라서 두 경우 모두 자신의 최적해를 구할 때 더 작은 문제의 최적해를 사용하므로 최적 부분구조(optimal substructure)를 가지고 있는 것을 알 수 있다. 또한 위의 식은 알고리즘을 재귀적으로 나타낸 식이 된다.

이제 이 알고리즘을 dynamic programming으로 작성해보면,

```
LongestPalindrome(1, j)
// 문자열 S(i,j) = {S_i, S_i+1, ... , S_j}의 가장 긴 palindrome의 길이 구하기.
// 이 때 문자열의 시작부터 끝까지의 최댓값을 구하는 것이므로 시작점은 1으로 둔다.
{
  for m <- j to 1
    for n <- m to j
      // for문을 이렇게 돌리는건 행렬의 대각선 끝 지점(j,j)부터 올라오면서 최종적으로
      // (1, j)까지 구하기 위함이다.
      if(m == n)                // base case 1 (길이 1짜리 string)
        C[m, n] = 1;
      else if(m + 1 == n)       // base case 2 (길이 2짜리 string)
        if(S[m] == S[n])
          C[m, n] = 2;
        else
          C[m, n] = 0;
      else
      {
        if(S[m] == S[n])                // Case 1
          C[m, n] = C[m+1, n-1] + 2;
        else            // Case 2
          C[m, n] = max{C[m+1, n], C[m, n-1]};
      }
  return C[1, j];
}
```

위와 같이 나타낼 수 있다.

알고리즘의 수행시간을 구해보자. 이 알고리즘은 for문을 이중으로 돌면서 바깥쪽은 j부터 0까지 j번 돌고 그 안쪽은 순서대로 1, 2, 3, …, j번 반복한다. 그리고 for문 안쪽은 단지 값을 넣어주므로 상수 시간이 소요되므로 무시해도 좋다. 따라서 수행 시간은 1 + 2 + 3 + 4 + … + j 가 되고, 이는 $\frac{j(j+1)}{2}$ 라는 식임을 알 수 있다.

이를 theta 식으로 나타내면 $\Theta(n^2)$이 된다. 따라서 이 알고리즘의 수행 시간은 $\Theta(n^2)$이다.

2번 문제

우선 놀이판에서 한 열에 표시할 수 있는 경우의 수를 알아보면, ○, △, ×를 놓을 수 있는데 이들이 행을 바꿔가며 놓일 수 있는 경우의 수는 3!이므로 총 6개의 패턴이 한 열에 발생할 수 있다.
그리고 가로로 이웃하는 두 칸에는 같은 표시가 나타나면 안 되는 제약 조건이 있으므로, 임의의 열의 각 패턴마다 양립할 수 있는 이전 열의 패턴의 수는 2개이다.

n열 중 1열부터 i열까지 표시를 하는 경우에 1열부터 i열까지의 최댓값을 생각해보자. 그러면 다음과 같이 총 6개의 경우가 나눠진다.

1. i열을 ○, △, × 순서로 표시하는 경우의 최고점.
2. i열을 ○, ×, △ 순서로 표시하는 경우의 최고점.
3. i열을 △, ○, × 순서로 표시하는 경우의 최고점.
4. i열을 △, ×, ○ 순서로 표시하는 경우의 최고점.
5. i열을 ×, △, ○ 순서로 표시하는 경우의 최고점.
6. i열을 ×, ○, △ 순서로 표시하는 경우의 최고점.

그리고 각각의 경우는 양립되는 2가지의 i-1열의 패턴이 있다.
1~6번 표시의 순서를 각각 패턴 1~6이라고 봤을 때, i-1열에 양립 가능한 패턴들을 정리해보면 다음과 같다.

1. 패턴4 or 패턴6
2. 패턴3 or 패턴5
3. 패턴2 or 패턴5
4. 패턴1 or 패턴6
5. 패턴2 or 패턴3
6. 패턴1 or 패턴4

예를 들어, i열에서 1번 패턴인 경우 1번 패턴의 최고점은 i-1열이 패턴4인 경우의 최고점과 패턴6인 경우의 최고점 중 더 큰 값과 i열에서 패턴1의 값을 더한 값이 된다. 또한 다른 패턴 또한 모두 마찬가지로 이전 열의 최적해를 사용한다.

이와 같이 최적해를 구하기 위해 자신보다 하나 작은 문제의 최적해를 사용하는 것을 알 수 있다. 따라서 이 구조는 최적 부분구조(optimal substructure)를 가지고 있음을 알 수 있다. 이 때 i번째 열까지의 최적해는 6가지 패턴 중 최댓값을 선택하면 된다.

이를 재귀적인 식으로 나타내보자. 이 때 C(i,p)는 i열이 패턴 p로 놓일 경우의 최고점수, W(i,p)는 i열이 패턴 p로 놓일 경우에 i열의 점수합이라고 할 때, 다음과 같은 식으로 나타낼 수 있다.

$$C(i,p) = W(1,p) \qquad \text{if } i == 1$$
$$C(i,p) = \max\{C(i-1,p_0),\ C(i-1,p_1)\} + W(i,p) \qquad \text{if } i > 1$$

이 때 p_0과 p_1은 i열의 패턴 p와 양립 가능한 2가지 패턴이다. 그리고 최종적으로 n열의 최고점수는 C(n,1)부터 C(n,6)까지의 n열의 6가지 패턴의 최고점수 중 최댓값을 선택하면 된다.

이제 이 알고리즘을 dynamic programming으로 작성해보면,

```
1.  MaxSum(n)
2.  // 1열부터 n열까지 표시를 할 때의 최고 점수
3.  {
4.    for p <- 1 to 6       // 첫 번째 열은 base case
5.      c[1,p] = w[1,p];
6.
7.    for i <- 2 to n
8.      for p <- 1 to 6     // 총 패턴 6개
9.        // p_0과 p_1은 패턴 p에 양립할 수 있는 2개의 패턴.
10.       c[i,p] = max{c[i-1,p_0], c[i-1,p_1]} + w[i,p];
11.
12.   // n번째 열의 6가지의 패턴의 최고 점수들 중 최댓값을 반환.
13.   return max{c[n,1], c[n,2], c[n,3], c[n,4], c[n,5], c[n,6]};
14. }
```

위의 그림처럼 나타낼 수 있다. 구체적으로 알고리즘을 적으면 각 열의 패턴마다 w를 구하는 과정도 필요하고 각 패턴별로 양립하는 이전 패턴 2가지씩을 6개 패턴별로 모두 나눠 구해야 하지만 모두 적으면 너무 길어져서 생략했다.

이제 이 알고리즘의 complexity가 O(n)임을 증명해보자. 우선 알고리즘에 생략된 부분 중 w를 구하는 과정은 1열부터 n열까지 각 열마다 패턴 6개의 경우의 합을 구해 저장하면 되므로 6n번의 수행이 필요하다. 그리고 4~5번째 줄은 상수 시간이 소요된다. 7줄부터의 for문을 살펴보면, 바깥쪽의 for문은 2부터 n까지 n-1번 반복되고, 안쪽의 for문은 6번의 반복만 필요하므로, 7~9줄의 for문은 총 6(n-1)의 수행이 필요하다. for문 안쪽은 단지 상수시간이 소요되므로 무시해도 된다.
알고리즘의 모든 부분이 최대 c*n의 형태의 수행을 한다는 것을 볼 수 있다. 따라서 이 알고리즘의 complexity는 O(n)을 만족한다.

3번 문제

이 문제는 수업 시간에 배운 Longest Common Subsequence(LCS)를 구하는 알고리즘을 이용했다. 우선 주어진 배열을 작은 값부터 큰 값까지 정렬을 한 뒤, 정렬한 배열과 기존 배열의 LCS의 길이를 구하면 longest monotonic increasing subsequence(LMIS)의 길이를 구할 수 있다. 즉 문제가 두 문자열 사이의 LCS를 구하는 문제로 바뀌게 된다. 이 때 LMIS의 길이를 구하는 데 사용한 원소들의 위치를 기억해 놓으면, 문제에서 요구한 답인 LMIS 배열을 구할 수 있다.

LCS의 길이를 구하는 문제를 살펴보면, 두 문자열 X_m = $\langle x_1 x_2 x_3 \cdots x_m \rangle$과 Y_n = $\langle y_1 y_2 y_3 \cdots y_n \rangle$이 있다고 할 때 두 가지의 경우로 나눌 수 있다. $LCS(X_m, Y_n)$는 X_m과 Y_n 사이의 LCS의 길이를 나타낸다.

1. $LCS(X_m, Y_n)$ = $LCS(X_{m-1}, Y_{n-1})$ + 1 if x_m == y_n
2. $LCS(X_m, Y_n)$ = max{$LCS(X_{m-1}, Y_n)$, $LCS(X_m, Y_{n-1})$} if x_m != y_n

위 식에서 최적해, 즉 LCS의 길이를 구하기 위해 자신보다 작은 문제의 최적해를 사용하는 것을 볼 수 있다. 따라서 이 LCS의 길이는 구하는 문제는 최적 부분구조(optimal substructure)를 가지고 있다.

이제 LCS의 길이를 구하는 부분을 포함해 dynamic programming을 작성하면, 다음과 같다.

```
1.  LMIS(X_n)
2.  // 배열 X_n = [x_1, x_2, ... , x_n]의 Longest Monotonic Increasing Subsequence 구하기
3.  // S_n은 정렬된 배열, L은 LCS의 길이를 담은 배열, P는 원소의 위치를 담은 배열, C는 정답을 담은 LMIS 배열.
4.  {
5.    S_n = Sorting(X_n);   // 배열 X_n을 ascending order로 정렬.
6.
7.    for i <- 0 to n       // base case. 길이가 0인 문자열과의 LCS의 길이는 0.
8.      L[i, 0] = 0;
9.      L[0, i] = 0;
10.     P[0, i] = 'x';
11.     P[i, 0] = 'x';
12.
13.   for i <- 1 to n       // 배열 S_n과 X_n 모두 길이가 n
14.     for j <- 1 to n
15.       if(S[i] == X[j])
16.         L[i,j] = L[i-1,j-1] + 1;
17.         P[i,j] = 'd';   // diagonal, 대각선 위로 가라는 뜻
18.       else
19.           if(L[i-1,j] >= L[i,j-1])
20.             L[i,j] = L[i-1,j];
21.             P[i,j] = 'u';       // up, 위로 가라는 뜻
22.           else
23.             L[i,j] = L[i,j-1];
24.             P[i,j] = 'l';       // left, 왼쪽으로 가라는 듯
25.
26.   length = L[n,n];      // LCS의 길이
27.   C[length];            // LCS의 길이만큼만 선언
28.
29.   i, j = n;
30.   k = length;
31.
32.   // n,n부터 대각선으로 이동할 때만 C 배열에 추가.
33.   while(P[i,j] != 'x')
34.     if(P[i,j] == 'd')
35.       C[k--] = X[j]; // 혹은 S[i]를 줘도 됨. 어차피 같기 때문.
36.       i--;
37.       j--;
38.     else if(P[i,j] == 'l')
39.       j--;
40.     else
41.       i--;
42.
43.   return C;
44. }
```

알고리즘을 살펴보자. 5줄은 주어진 배열을 sorting하는 과정이다. 이 과정의 수행 시간은 종류에 따라 다르지만 bubble sort 등 간단한 알고리즘을 사용해도 O(n^2)이라는 복잡도를 얻을 수 있다.

7~11줄은 base case를 초기화하는 부분으로 수행시간은 n이 된다.

13~24줄의 for문은 LCS의 길이를 구하고 위치를 기억해놓는 부분이다. for문 안쪽은 모두 상수

시간이 걸리고, 바깥쪽의 for문과 안쪽의 for문 모두 n번씩 반복을 하므로 n^2번의 수행을 하는 것을 볼 수 있다.
26~30줄은 필요한 변수를 선언하는 과정으로 모두 상수 시간이 걸린다.
33~41줄의 while문은 LMIS 배열을 구하는 과정으로 (n,n)부터 i 혹은 j가 0이 되는 순간까지 수행한다. 코드를 살펴보면 모든 케이스에 i 혹은 j가 1이 줄거나 또는 i, j가 모두 1씩 줄어든다. i나 j가 무조건 1씩 줄기 때문에 최악의 경우인 i와 j가 번갈아 줄어드는 경우에도 최대 2n번 이상의 수행을 하지 않는다.
알고리즘의 모든 부분을 살펴봤을 때, 수행시간은 O(n^2)이 되는 것을 알 수 있다.

4번 문제

임의의 줄이 단어 i부터 시작해 j로 끝난다고 하자. 그러면 이 줄에서 필요한 공간과 남는 공간은 각각

1. 필요한 공간: $\sum_{k=i}^{j} w_k + (j-i)$ 단, 공간은 80을 넘지 않아야 함.
2. 남는 공간: $80-(j-i)-\sum_{k=i}^{j} w_k$

임을 알 수 있다.
이 상황에서 최적 부분구조(optimal substructure)를 생각해보면,
C(i)를 최적해, 즉 W_i의 길이를 가진 i번째 단어를 마지막으로 임의의 줄이 끝나는 경우의 여백의 "제곱합"의 최솟값이라고 하자.
그리고 이전 줄이 k번째 단어로 끝났다고 가정하면, i번째 단어가 있는 줄은 k+1번째 단어부터 i번째까지 단어가 나열되어 있을 것이다. 그리고 C(i)를 구해 보면

C(i) = $\min(C(k)+(80-(i-k-1)-\sum_{j=k+1}^{i} w_j)^2)$

이 때 k는 $(i-k-1)+\sum_{j=i+1}^{i} w_j \leqq 80$ 의 조건을 만족해야 한다.

위와 같은 식으로 나타나는데, 최적해를 구하기 위해 더 작은 문제의 최적해의 값을 사용하고 있는 것을 볼 수 있다. 따라서 C(i)를 구하는 문제는 최적 부분구조(optimal substructure)를 가지고 있음을 알 수 있다.
이 알고리즘을 dynamic programming으로 작성해보았다.

```
1.  MinSquareSum(words)
2.  // 1번째 단어부터 n번째 단어까지 있을 때, W[i]는 i번째 단어의 길이.
3.  // 단어 배치를 통해 각 줄의 여백의 제곱합의 최솟값을 구하는 문제.
4.  // S[i]는 i번째 까지의 단어 길이의 합, C[i]는 최소여백제곱합
5.  {
6.    S[1] = W[1];
7.    for i <- 2 to n       // S를 구함.
8.      S[n] = S[i-1] + W[n];
9.
10.   for i <- 1 to n
11.     if(i - 1 + (S[i] - S[1]) <= 80)     // 첫 줄의 경우.
12.       if(i == n)        // n번째까지 단어를 배치해도 첫 줄일 경우
13.         return 0;       // 마지막 줄의 공백은 제외하므로 0을 리턴.
14.       C[i] = (80 - (i - 1) - (S[i] - S[1]))^2;
15.     else
16.       k = i-1;
17.       min = C[k] + (80 - (i - k - 1) - (S[i] - S[k+1]))^2;
18.       while(i - k - 1 + (S[i] - S[k+1]) <= 80) // k를 줄여나가며 최솟값 탐색.
19.         if(C[k] + (80 - (i - k - 1) - (S[i] - S[k+1]))^2 < min)
20.           min = C[k] + (80 - (i - k - 1) - (S[i] - S[k+1]))^2;
21.         k--;
22.       C[i] = min;
23.
24.   k = n-1;
25.   min = C[k];
26.
27.   // 마지막 줄의 여백은 제외하므로 n-1번째 줄까지의 최솟값 반환.
28.   while(n - k - 1 + (S[n] - S[k+1]) <= 80)
29.     if( C[k] < min )
30.       min = C[k];
31.     k--;
32.
33.   return min;
34. }
```

알고리즘을 살펴보자. 우선 6~8줄은 i번째 단어까지의 길이 합을 구한 것으로, 수행시간은 n이 걸린다. 이는 10~22줄과 28~31줄의 반복문을 돌면서 W_i들의 합을 중복해서 구하지 않고(그럴 경우 overhead가 크다) O(1)의 수행시간으로 구하기 위해서 만들어 놓았다.
그 뒤 10~22줄은 바깥쪽의 for문이 n번 돌고 있다. 11~14줄의 if문의 경우 1번째부터 i번째까지의 필요한 공간이 80보다 작은 경우, 즉 첫 번째 줄인 경우의 케이스이다. 이 경우에는 이전 줄의 값이 필요 없으므로 바로 대입한다. 또한 마지막 줄의 공백은 신경 쓰지 않기 때문에

n번째까지 루프를 모두 돌아도 첫 번째 줄일 경우 바로 0을 리턴한다.
15~22줄은 다음 줄부터의 경우로, k의 값을 i-1부터 줄여나가며 최적해를 찾는다. 이 때 18~21줄의 while문의 수행시간을 생각해보자. C(i)를 구하기 위해서 살펴볼 k의 개수 중 최악의 경우는 이전에 길이 1짜리인 단어가 연속해서 있을 경우로, 이 경우 봐야 할 k의 개수가 40개이므로 최악의 경우에도 상수시간 안에 하나의 C(i)의 값을 찾을 수 있는 것을 알 수 있다. 따라서 for문 안의 부분들은 모두 상수 시간의 수행 시간이 걸리고, 10~22줄의 총 수행 시간은 O(n)임을 알 수 있다.
그리고 마지막 줄의 여백을 제외한 값을 리턴하기 위해 28~31줄의 while문이 수행되는데, 이 구간에서도 마찬가지로 k의 갯수는 최대 상수개이기 때문에 28~31줄의 수행 시간은 상수 시간이 걸린다는 것을 알 수 있다.

알고리즘의 부분들을 모두 살펴본 결과 수행 시간은 O(n), 즉 linear time에 수행된다는 것을 알 수 있다.

5번 문제

이 문제는 한 행에서는 한 방향으로만 갈 수 있다는 사실을 이용했다.
(i,j)에서의 가장 점수가 높은 경로를 C(i,j)라고 하고, 그 점에서의 값을 A(i,j)라고 할 때 알고리즘은 다음과 같이 나타낼 수 있다.

$$
\begin{aligned}
C(i,j) &= A(1,\ j) + C(1,\ j-1) && \text{if } i == 1 \\
&\quad \max\{L(i,\ j),\ R(i,\ j)\} && \text{if } i > 1
\end{aligned}
$$

$$
\begin{aligned}
\text{s. t.} \quad L(i,\ j) &= A(i,\ 1) + C(i-1,\ 1) && \text{if } j == 1 \\
&\quad A(i,\ j) + \max\{C(i-1,\ j),\ L(i,\ j-1)\} && \text{if } j > 1 \\
R(i,\ j) &= A(i,\ n) + C(i-1,\ n) && \text{if } j == n \\
&\quad A(i,\ j) + \max\{C(i-1,\ j),\ R(i,\ j+1)\} && \text{jf } j < n
\end{aligned}
$$

우선 i가 1일 경우를 살펴보면, (1,1)의 점부터 시작해 오른쪽으로 가는 단방향밖에 없으므로 C는 j-1까지의 합에 자신의 값을 더한 값이 된다.
i가 1보다 큰 경우를 보면, C는 L과 R의 최댓값이라고 적어 놨는데, L과 R은 일종의 부분적인 최적해라고 볼 수 있다.
L은 자신의 왼쪽 방향으로부터의 최적해를 구하며, R은 자신의 오른쪽 방향으로부터의 최적해를 구한다. L의 경우 각각의 점에서 왼쪽으로부터 오는 경로와 위로부터 오는 경로 중 더 큰 값을 택하고, 반대로 R의 경우 오른쪽으로부터 오는 경로와 위로부터 오는 경로 중 더 큰 값을 택해 최적해를 구한다.
그리고 각각 L, R을 구한 뒤 왼쪽으로부터 최적해(L)와 오른쪽으로부터의 최적해(R)의 값 중 더 큰 것을 택해 전체적인 최적해를 구하게 된다.
이처럼 자신의 최적해를 구하기 위해 더 작은 문제의 최적해를 이용하는 것을 볼 수 있으므로 이 문제는 최적 부분구조(optimal substructure)를 가지고 있음을 알 수 있다.

알고리즘을 dynamic programming을 이용해 작성해보면,

```
1.  MaxSumofPath{A}
2.  // A[1][1] ~ A[n][n] 행렬에서 (1,1)부터 (n,n)까지의 경로 중 최댓값.
3.  {
4.    C[1][1] = A[1][1];                    // 1,1 초기화
5.    for i <- 2 to n                       // 첫번째 행 초기화
6.      C[1][i] = C[1][i-1] + A[1][i];
7.
8.    for i <- 2 to n
9.      L[i][1] = A[i][1] + C[i-1][1];      // L의 가장 왼쪽 초기화
10.     R[i][n] = A[i][n] + C[i-1][n];      // R의 가장 오른쪽 초기화
11.     for j <- 2 to n     // L을 구함
12.       L[i][j] = A[i][j] + max{L[i][j-1], C[i-1][j]};
13.     for j <- n-1 to 1   // R을 구함
14.       R[i][j] = A[i][j] + max{R[i][j+1], C[i-1][j]};
15.     for j <- 1 to n     // 이제 C를 구함.
16.       C[i][j] = max{L[i][j], R[i][j]};
17.
18.   return C[n][n];
19. }
```

위의 그림과 같다.

알고리즘을 살펴보자. 우선 4~6줄에서는 1행의 최적해를 지정해주었다. for문이 n-1번 반복되므로 4~6줄의 수행 시간은 n-1, O(n) 이다.
그 뒤 8~16줄의 코드는 L, R, C를 2행부터 n행까지 내려가면서 차례대로 구하는 코드이다. 우선 L과 R의 base case를 초기화 해주었다. 이 부분은 상수 시간이 소요된다. 그 뒤 11~12줄, 13~14줄, 15~16줄에서 for문이 각각 n-1번, n-1번, n번의 반복을 하며 값을 구한다. 즉 3번의 for문을 수행하므로 11~16줄까지의 수행 시간은 3n-2이라고 할 수 있다. 바깥쪽의 for문이 n-1번 수행되므로 8~16줄의 전체 수행 시간은 $(3n-2)(n-1)$, 즉 O(n^2)이 된다.
알고리즘의 모든 부분을 살펴본 결과, 이 알고리즘의 수행 시간은 O(n^2)임을 알 수 있다.