[5] 동적 프로그래밍

1) 동적 프로그래밍

□ 동적 프로그래밍(DP : dynamic programming) = 동적 계획법

- 작은 문제를 먼저 해결하여 그 결과를 테이블에 저장한 다음, 필요할 때마다 테이블에 저장된 값을 이용하여 상향식으로 해답을 구축 : Tabulation
- 동적 프로그래밍 알고리즘의 개발 절차
  - ✓ step 1 : 큰 문제에 대한 해와 작은 문제에 대한 해 사이의 재귀관계식 구하기
  - ✓ step 2 : 작은 문제의 해를 큰 문제의 해결에 사용
- 항상 최적값을 먼저 구하고 난 후, 최적해를 역으로 구축(traceback)해야 함

□ 동적 프로그래밍 vs 분할정복

- 공통점 : 문제를 더 작은 부분 문제로 분할하여 문제를 해결
- 분할정복 알고리즘
  - ✓ 부분 문제들을 재귀적으로 해결한 후, 그 결과들을 통합하여 원래의 문제를 해결
  - ✓ 하향식(top-down) 접근 방법
  - ✓ 부분 문제들이 서로 독립적인 문제로 분할될 때 유용
- 동적 프로그래밍
  - ✓ 작은 문제를 먼저 해결한 후 그 결과를 저장하여 더 큰 문제의 해결에 반복적으로 적용
  - ✓ 상향식(bottom-up) 접근 방법
  - ✓ 점진적(incremental) 해결법
  - ✓ 부분 문제들이 서로 독립적이지 않고, 부분 문제들이 다시 자신의 부분 문제를 공유할 때 사용

□ 피보나치 수열 : 분할정복 vs 동적 프로그래밍

- 피보나치 수열(Fibonacci sequence)

$$f(n) = \begin{cases} n & , n = 0, 1 \\ f(n-1) + f(n-2) & , n > 1 \end{cases}$$

- 분할정복 알고리즘 : 재귀관계식을 하향식으로 구현(재귀)

```
fib1(n)
  if (n==0 || n==1) return n
  return fib1(n-1) + fib1(n-2)
```

  - ✓ 사례의 원래 크기와 거의 같은 크기를 가진 2개의 사례로 분할됨 ⇒ 지수시간!
  - ✓ 재귀호출이 중복적으로 발생
- 동적 프로그래밍 알고리즘 : 재귀관계식을 상향식으로 구현(반복)

```
fib2(n)
  f[0...n] 공간 확보
  f[0] = 0
  f[1] = 1
  for (i=2; i<=n; i++)
    f[i] = f[i-1] + f[i-2]
  return f[n];
```

✓ $O(n)$ 시간 소요!

☞ Memoization : 하향식 DP 알고리즘

```
solve(n, f[])
  if f[n] != -1
    return f[n]
  f[n] = solve(n-1, f) + solve(n-2, f)
  return f[n]
fib3(n)
  f[0...n]을 –1로 모두 초기화
  f[0] = 0
  f[1] = 1
  return solve(n, f)
```

2) 동적 프로그래밍과 최적화 문제

□ 최적의 원칙(principle of optimality)

- 동적 프로그래밍으로 최적화 문제를 풀기 위해서는 최적의 원칙을 만족해야 함
- 큰 문제에 대한 최적해가 부분문제에 대한 최적해를 항상 포함하는 것을 일컬음
  - ✓ 최단경로문제 : 최단경로상의 모든 부분 경로들 역시 해당 정점들에 대하여 최적임
  - ✓ 최장경로문제 : 최장경로상의 부분 경로가 반드시 두 점상의 최장경로는 아님
- 증명 : cut-and-paste 테크닉

□ 행렬 경로 문제

- $n \times n$행렬의 최상단 맨 왼쪽 (1, 1)에서 시작하여 최하단 맨 오른쪽 (n, n)까지 오른쪽 혹은 아래쪽으로만 이동하여 도달하는 방법 중 이동 경로상의 점수 합의 최댓값 구하기
- 문제는 최적의 원칙을 만족하는가?
  - ✓ 최적 경로상의 지점은 해당 지점까지의 최적 경로
- 최적해에 대한 재귀 관계식
  - ✓ $m[i][j]$: 주어진 행렬에서 (i, j) 위치의 원소값
  - ✓ $c[i][j]$: (i, j)까지 최대 점수
  - ✓ (i, j)까지 도달하는 경로상의 최고점수는 $(i-1, j)$까지의 최고점수와 $(i, j-1)$까지의 최고점수 중 큰 것에 원소 (i, j)의 점수를 더한 값

$$c[i][j] = m[i][j] + \max(c[i][j-1], c[i-1][j]) \text{ if } i > 0, j > 0$$

  - ✓ 최대 점수 : $c[n][n]$

행렬 m[][]

	1	2	3	4
1	6	7	12	5
2	5	3	11	18
3	7	17	3	3
4	8	10	14	9

행렬 c[][]

0	0	0	0	0
0	6	13	25	30
0	11	16	36	54
0	18	35	39	57
0	26	45	59	68

- 코딩 주의사항
  - ✓ 크기 $(n+1)\times(n+1)$ 배열 생성, 0번째 행과 열 제로 패딩(zero-padding)
  - ✓ 만일 m[][]이 0-indexing으로 저장되어 있다면, m[i][j] ⇒ m[i-1][j-1]
- 시간복잡도
  - ✓ 가능한 모든 이동 경로를 따져 보는 방법(brute-force search) : 지수 시간
  - ✓ 분할정복법 : 재귀함수의 중복호출로 인하여 지수 시간
  - ✓ 이차원 배열 $c[\,][\,]$을 이용한 동적 프로그래밍 알고리즘의 실행시간 : $\Theta(n^2)$

3) 최장 공통 부분순서 문제

☐ 최장 공통 부분순서(LCS : Longest Common Subsequence) 문제

- 최장 공통 부분순서
  - ✓ 두 문자열에 공통으로 나타나는 부분순서 중 가장 긴 것
  - ✓ 예) <abcbdab>와 <bdcaba>의 최장 공통 부분순서
    - \- LCS : <bcba>, <bcab>, <bdab>
    - \- LCS의 길이 = 4
- LCS의 길이에 존재하는 최적 부분구조
  - ✓ 입력 : 두 문자열 $X_m = < x_1x_2 \cdots x_m >$, $Y_n = < y_1y_2 \cdots y_n >$
  - ✓ 출력 : 최장 공통 부분순서 $Z_k = < z_1z_2 \cdots z_k >$의 길이, k
  - ✓ $c[i][j]$: 문자열 $X_i = < x_1x_2 \cdots x_i >$와 $Y_j = < y_1y_2 \cdots y_j >$의 LCS의 길이
    - \- 만약 $x_i = y_j$이면, X_i과 Y_j의 LCS의 길이는 (X_{i-1}과 Y_{j-1}의 LCS의 길이) + 1
    - \- 만약 $x_i \neq y_j$이면, X_i과 Y_j의 LCS의 길이는 (X_i과 Y_{j-1}의 LCS의 길이)와 (X_{i-1}과 Y_j의 LCS의 길이) 중 더 큰 것

$$c[i][j] = \begin{cases} c[i-1][j-1] + 1 & \text{if } x_i = y_j \\ \max(c[i-1][j],\, c[i][j-1]) & \text{if } x_i \neq y_j \end{cases},\ \text{단 } i>0,\, j>0$$

  - ✓ LCS의 길이 : $c[m][n]$
- 코딩 주의사항
  - ✓ 크기 $(m+1)\times(n+1)$ 배열 생성, 0번째 행과 열 제로 패딩(zero-padding)
  - ✓ if x[i] == y[j] ⇒ if x[i-1] == y[j-1], 문자열의 0-indexing
- 시간복잡도 : $\Theta(m \cdot n)$
- 예제) X = "abdbdab", Y = "bdcaba" ⇒ LCS의 길이 = $c[7][6]$= 4

j / i		0	1	2	3	4	5	6
		-	b	d	c	a	b	a
0	-	0	0	0	0	0	0	0
1	a	0	0	0	0	1	1	1
2	b	0	1	1	1	1	2	2
3	c	0	1	1	2	2	2	2
4	b	0	1	1	2	2	3	3
5	d	0	1	2	2	2	3	3
6	a	0	1	2	2	3	3	4
7	b	0	1	2	2	3	4	4

- 최장 부분순서 구하기
  - ✓ 테이블 C를 역추적(traceback)하여 구하기

```
traceback(i, j)
  if(i==0 or j==0)
    return ""
  else if(X[i] == Y[j])
    return traceback(i-1, j-1) + X[i]  //문자열 연결
  else
    if(C[i][j-1] >= C[i-1][j])
      return traceback(i, j-1)
    else
      return traceback(i-1, j)
```

  - ✓ traceback(m, n)을 호출
  - ✓ 최장 부분순서의 길이는 유일하지만(unique), 그 길이를 갖는 최장 부분순서는 여러 개 존재할 수 있음
- 최장 부분순서 문제의 활용 사례(applications)
  - ✓ Bioinformatics : DNA 시퀀스의 염기(A,C,G,T) 서열 비교를 통한 유사성 판정
  - ✓ 유닉스 *diff* utility : 라인 기반의 파일 비교(file comparison) 유틸리티
  - ✓ Screen redisplay in “emacs” : 터미널에 가능한 적은 숫자의 문자를 업데이트
  - ✓ Revision control system(e.g. Git)

□ 편집거리 문제

- 편집거리는 어떤 문자열 $X_m = < x_1 x_2 \cdots x_m >$을 다른 문자열 $Y_n = < y_1 y_2 \cdots y_n >$로 변환하는데 필요한 최소 연산의 수
- 연산의 종류 : 삭제(deletion), 삽입(insertion), 변경(substitution)
- 두 개의 문자열의 상이성 척도 : Levenshtein 알고리즘
- $d[i][j]$: 문자열 $X_i = < x_1 x_2 \cdots x_i >$와 $Y_j = < y_1 y_2 \cdots y_j >$의 편집거리
  - ✓ 한쪽이 빈 문자열인 경우, 해당 문자열의 길이만큼 삽입 연산
  - ✓ 만약 $x_i = y_j$이면, 편집 연산이 필요 없으므로 $d[i][j]$는 $d[i-1][j-1]$과 같음
  - ✓ 만약 $x_i \neq y_j$이면,
    - x_i를 삭제하는 경우 : $d[i-1][j]+1$
    - y_j를 삽입하는 경우 : $d[i][j-1]+1$
    - x_i를 y_j로 변경하는 경우 : $d[i-1][j-1]+1$

$$d[i][j] = \begin{cases} i & \text{if } j=0 \\ j & \text{if } i=0 \\ d[i-1][j-1] & \text{if } i,j \neq 0 \text{ and } x_i = y_j \\ \min(d[i-1][j-1],\ d[i-1][j],\ d[i][j-1]) + 1 & \text{if } i,j \neq 0 \text{ and } x_i \neq y_j \end{cases}$$

- ✓ 편집거리 = $d[m][n]$

- 코딩 주의사항
  - ✓ 크기 $(m+1) \times (n+1)$ 배열 생성, 제로 패딩 안 함!
  - ✓ if x[i] == y[j] ⇒ if x[i-1] == y[j-1], 문자열의 0-indexing
- 예제) "MICROSOFT"와 "NCSOFT" 사이의 편집거리 = 4 (변경 1회, 삭제 3회)
- LCS는 deletion과 insertion만을 허용하는 편집거리 문제의 특별한 경우
  - ✓ (X_m, Y_n)의 편집거리 = $m+n$ − 2 * (X_m, Y_n)의 LCS 길이

4) 0-1 배낭 문제

□ 배낭 문제(knapsack problem)

- 배낭에 담을 수 있는 n개 아이템의 아이템별 무게와 가치가 주어 짐
  - ✓ $w[i]$: i번째 아이템의 무게
  - ✓ $p[i]$: i번째 아이템의 가치
  - ✓ C : 배낭에 넣을 수 있는 최대 무게(capacity)
- 목표 : 용량을 초과하지 않으면서, 배낭에 담는 아이템의 가치의 합이 최대가 되도록 담기
- 0-1 배낭 문제 vs 연속 배낭 문제
  - ✓ 0-1 : 아이템의 부분을 배낭에 넣는 것이 불가능 => 동적 프로그래밍
  - ✓ 연속 : 아이템의 부분을 배낭에 넣는 것이 가능 => 탐욕적 알고리즘

□ 0-1 배낭 문제 예제

- 3개의 아이템의 무게와 가치

	1	2	3
w[i]	5	10	20
p[i]	50	60	140

  - ✓ C = 30인 경우, 최대 가치 합 = 200 ⇐ 아이템 (2, 3) 무게 합 = 30
- 6개의 아이템의 무게와 가치

	1	2	3	4	5	6
w[i]	4	2	6	4	2	10
p[i]	7	10	6	7	5	4

  - ✓ C = 10인 경우, 최대 가치 합 = 24 ⇐ 아이템 (1, 2, 4) 무게 합 = 10
  - ✓ C = 17인 경우, 최대 가치 합 = 30 ⇐ 아이템 (1, 2, 3, 4) 무게 합 = 16

□ 0-1 배낭 문제를 푸는 동적 프로그래밍 알고리즘

- $r[i][c]$: 아이템 $(1,\ \ldots, i)$ 중에서 골라서 용량이 c인 배낭에 담을 수 있는 최대 가치
  - ✓ i번째 아이템을 선택하는 경우 : $r[i-1][c-w[i]]+p[i]$
  - ✓ i번째 아이템을 선택하지 않는 경우 : $r[i-1][c]$

$$r[i][c] = \begin{cases} r[i-1][c] & \text{if } c < w[i] \\ \max(r[i-1][c-w[i]]+p[i],\ r[i-1][c]) & \text{if } c \geq w[i] \end{cases},\ \text{단 } i>0, c>0$$

- 코딩 주의사항
  - ✓ 크기가 $(n+1)\times(C+1)$ 배열 생성, 첫 행과 첫 열 제로 패딩
  - ✓ w[]와 p[]의 0-indexing : r[i][c] = max(r[i-1][c-w[i]]+p[i], r[i-1][c])
    => r[i][c] = max(r[i-1][c-w[i-1]]+p[i-1], r[i-1][c])
- 최적값(=최대 가치) : $r[n][C]$
- 예제) 6개 아이템, $C=10$ => 최대 가치 = 24

i \ c	0	1	2	3	4	5	6	7	8	9	10
0	0	0	0	0	0	0	0	0	0	0	0
1	0	0	0	0	7	7	7	7	7	7	7
2	0	0	10	10	10	10	17	17	17	17	17
3	0	0	10	10	10	10	17	17	17	17	17
4	0	0	10	10	10	10	17	17	17	17	24
5	0	0	10	10	15	15	17	17	22	22	24
6	0	0	10	10	15	15	17	17	22	22	**24**

- 시간복잡도 : $O(n \cdot C)$
  - ✓ 주의 : 선형시간 아님!!!!!
  - ✓ n과 비교하여 C가 극도로 큰 경우에는 모든 부분집합을 고려하는 알고리즘보다 나쁨
- 0-1 배낭 문제는 대표적 NP-complete 문제