10.1 서론

동적 계획법 (Dynamic Programming ; DP) → 알고리즘 설계의 주요 패러다임

↳ 프로그래밍 (계획법) - "일련의 선택들"

↳ 동적 (Dynamic) - 이러한 선택이 미리 결정되는 것이 아니라 현재 상태에 좌우

⇒ 동적 계획법을 통해 지수 시간이 소요되는 계산을 다항 시간으로 줄일 수 있음

↳ 앞 장에서는 하나의 문제를 풀기 위한 알고리즘을 다루었지만 여기서는 다양한 문제를 동적 계획법으로 바꾸는 것에 초점

하향식 (Top-down) 설계 - 문제를 해결하기 위한 자연스러운 방법

↳ 전반적인 것을 먼저 생각한 후 상세한 것을 추가 ⇒ 상위 레벨을 보다 작은 문제로 분할 (주로 재귀 이용)

↳ 재귀는 좋은 매커니즘이지만 잘 제어하지 못하면 아주 비효율적 (대표적으로 피보나치 수)

〈예시〉 피보나치 수

피보나치를 재귀로 구현하면 트리 형태가 됨 ⇒ 수행시간은 $\Omega(2^{\frac{n}{2}})$ → 지수시간 소요

배열을 사용하여 다음과 같이 바꾸면 상수 시간 내에 가능

$f(0) = 0 \quad f(1) = 1$

for $(i=2,\ i \le n,\ i{+}{+})$

$f(i) = f(i-1) + f(i-2)$ → 이전의 결과값들을 저장

동적 계획법 - 크기가 작은 부분문제들의 결과를 테이블에 저장 ⇒ 큰 문제의 해를 구할 때 이를 이용 (많은 재귀를 대치)

10.2 부분문제 그래프와 순행

〈정의 10.1〉 부분문제 그래프 (subproblem graph)

어떤 문제에 대한 재귀 알고리즘 A

A에 대한 부분문제 그래프 - 유향 그래프

↳ 정점 : 입력

예시 : 입력 I에 대해 알고리즘 A를 수행할 때 어떤 입력 J에 대한 재귀 호출을 하면

$I \rightarrow J$

P가 알고리즘 A의 input A(P)의 부분문제 그래프 - A에 대한 부분문제 그래프의 일부분. P로부터 도달 가능

알고리즘 A가 항상 종료 ⇒ A의 부분문제 그래프에 사이클 X ⇒ 사이클이 없는 유향 그래프 (DAG) 알고리즘이 이용됨

DFS에서 정점 방문을 나타내기 위해 색을 이용 → 색을 할당하지 않는 탐색 : 비기억 그래프 운행
↳ 비기억 그래프 운행 : 사이클이 없는 그래프에서 모든 경로를 운행 ← 재귀 호출의 방식 (비효율적)

해를 구하고자 하는 문제 I로 부터 부분 문제들 $J_1, J_2, \cdots, J_n$로 가는 에지 존재 ⇒ 부분문제의 해부터 구해야 함
⇒ 부분 문제 그래프 → 종속 문제 그래프로 볼 수 있음.
↳ 해결되어야 하는 부분문제들의 순서 구하고 해가 나중에 이용될 경우를 위해 저장 ⇒ 부분문제는 한번만 해결하면 됨.

동적 계획법의 본질 : 부분문제 그래프에 대한 역위상 순서 구하고 이 순서에 따라 부분 문제 해결.
이 해를 다른 부분문제의 해를 구하는데 이용하도록 저장

DFS를 이용하여 역위상 순서를 계산하는 알고리즘을 이용하면 쉽게 해결 (역위상 순서 계산과 해 구하기가 동시에 가능)
↳ DFS는 그 자체가 재귀 ⇒ 동적 계획법도 기존의 A와 유사한 구조를 가지게 됨.
↳ 이미 방문했던 정점을 가지 않는 것이 핵심.

<정의 10.2> 재귀 알고리즘의 동적계획법.
재귀 알고리즘 A의 동적 계획법 : DP(A)로 표기.
DP(A) : 해결하고자 하는 문제 P에 대해 A(P)에 대한 부분문제 그래프에서 깊이 우선 탐색을 하는 프로시저.
↳ 부분문제들의 해는 사전식 구조에 저장 → 해를 기억하는 과정 : memo-ization

재귀 알고리즘 A에 다음의 문장을 삽입하여 DP(A)로 변환 가능.
해결하고자 하는 문제를 P, 이에 대한 부분문제를 Q라 할 때,
① 부분 문제 Q에 대한 재귀호출을 하기전에 Q에 대한 해가 있는지 검사.
a. Q에 대한 해가 없으면 재귀호출 시행 (새로운 깊이 우선 탐색 시작)
b. 해가 있으면 그 해를 사용, 재귀호출 X. (이미 발견된 정점)
② P에 대한 해를 반환하기 이전에 P에 대한 해 저장

이 알고리즘에서는 부분문제에 대한 사이클이 있으면 안됨.
↳ DFS에는 사이클 운행을 방지할 수 있지만 이 알고리즘은 불가능

도달 가능한 부분 문제들의 개수 = 사전식 구조의 크기 ⇒ 효율적인 DP 설계와 분석에 있어 매우 중요.

<예시> DP (fib)

```
fibWrap (n)  → 포장기
    Dict soln = create (n)
    return fibDP (soln, n)

fibDP (soln, k)
    int fib, f1, f2
        if (k < 2)
            fib = k
        if (member (soln, k) = false)
            f1 = fibDP (soln, k-1)
        else
            f1 = retrieve (soln, k-1)
        if (member (soln, k-2) = false)
            f2 = fibDP (soln, k-2)
        else
            f2 = retrieve (soln, k-2)
    fib = f1 + f2
    store (soln, k, fib)
    return fib
```

동적 계획법 → DFS 관점에서 보면 시간복잡도 분석에 유용

10.3 연속된 행렬의 곱셈

연속된 행렬의 곱셈순서 → 동적 계획법의 전형적인 예제

↳ 여기서는 곱셈 순서를 구하는 방법 그 자체보다 동적 계획법에 의한 알고리즘 개발 원칙을 보는 것이 목표

⇒ 새로운 문제를 풀때 동적 계획법이 언제 좋은 전략인지를 아는 것이 중요.

- 행렬의 곱셈 순서 문제

두 개 이상의 행렬을 곱할 때 가장 좋은 행렬의 곱셈 순서를 결정

$p \times q$ 행렬과 $q \times r$ 행렬의 곱 ⇒ pqr 번의 원소 곱셈 필요

↳ (행렬의 곱은 순서에 관계없이 같은 결과 (결합 법칙 성립)
행렬을 곱하는 순서에 따라 수행 시간에서 큰 차이가 남

<예제> 행렬의 다양한 곱셈 순서

$A_1(30\times1) \times A_2(1\times40) \times A_3(40\times10) \times A_4(10\times25)$ 일 때 순서에 따라

$((A_1A_2)A_3)A_4 = 30\cdot1\cdot40 + 30\cdot40\cdot10 + 30\cdot10\cdot25 = 20,700$

$A_1(A_2(A_3A_4)) = 40\cdot10\cdot25 + 1\cdot40\cdot25 + 30\cdot1\cdot25 = 11,750$

$(A_1A_2)(A_3A_4) = 30\cdot1\cdot40 + 40\cdot10\cdot25 + 30\cdot40\cdot25 = 41,200$

$A_1((A_2A_3)A_4) = 1\cdot40\cdot10 + 1\cdot10\cdot25 + 30\cdot1\cdot25 = 1,400$

$A_1, A_2, \cdots A_n$ 인 n개의 행렬, 각각의 차원이 $d_{i-1} \times d_i$ $(1 \le i \le n)$ 일 때 어떤 순서로 곱해야 최소 비용?

↳ 두 행렬 A_k와 A_{k+1}의 곱을 k번째 행렬곱

- 욕심쟁이 전략

가장 작은 비용의 행렬 곱을 계속 선택 → 수정된 차원에서 계속 반복

↳ 특수한 경우에는 잘 작동하지만 일반적인 해를 구하지는 못함.

↳ 일반적으로 동적 계획법은 욕심쟁이 방법보다 시간이 많이 걸리므로 greedy가 안될 때 고려

- 동적 계획법에 의한 해

행렬들의 최소 비용 곱 순서를 구하는 재귀 알고리즘을 먼저 개발

↳ 처음 행렬곱을 선택한 후 남아있는 문제들에 대한 최적의 해를 재귀적으로 탐색

첫 번째 행렬곱 i를 모든 위치에서 탐색 후 최적 해 선정 (뒤로 가면서 수행하므로 퇴각 검색 알고리즘)

행렬의 차원 $d_0, d_1, \cdots, d_n$은 배열 $d:m$에 저장

부분 문제의 식별 — 남아있는 행렬들의 차원을 인덱스로 하는 수열에 저장.

↳ 초기 인덱스 수열: $0 \sim n$. ⇒ 수열의 처음과 마지막을 제외한 모든 인덱스는 두 행렬의 곱.

수열의 첫 번째 행렬 곱: i ⇒ 남아있는 문제의 인덱스 수열: $0, \cdots, i-1, i+1, \cdots, n$

↳ i번째 행렬 곱의 결과

```
mmTry1(dim, len, seq)
    if(len < 3)
        bestCost = 0
    else
        bestCost = ∞
        for(i=1, i≤len-1, i++)
            C= seq(i) 번째 행렬곱 비용
            newSeq = i번째 원소가 삭제된 seq
            b = mmTry(dim, len-1, newSeq)
            bestCost = min(bestCost, b+c)
    return bestCost
```

이 알고리즘의 수행 시간에 대한 순환방정식

$T(n) = (n-1)T(n-1) + n$

$= \theta((n-1)!)$

↳ 너무 오래 걸림

동적 계획법 설계 ⇒ 부분 문제 그래프부터 분석

↳ 인덱스 수열 : 처음에는 연속하는 수열. 시간이 지날수록 파편화

⇒ 연속하지 않는 수열을 지정할 수 있는 간단한 방법이 없음

⇒ 초기 수열로 부터 나오는 부분 문제도 매우 많아 효율적 탐색 어려움

동적 계획법 알고리즘 설계 원칙 - 부분문제의 식별자는 간단해야 함.

↳ 식별자의 수는 부분문제 그래프의 최대 크기에 의해 결정

사전식 구조의 크기는 식별자의 수에 종속.

처음에 수행할 행렬곱이 아니라 제일 마지막에 수행할 행렬곱을 고르면 그 부분문제는?

↳ ① 차원의 인덱스가 $0, \sim i$ 인 행렬 $A_1, \cdots, A_i$의 곱 ⇒ $A_0 \cdots A_i = B_1(d_0, d_i)$

② 차원의 인덱스가 $i \sim n$ 인 행렬 $A_{i+1} \sim, A_n$의 곱 ⇒ $A_{i+1} \cdots A_n = B_2(d_i, d_n)$

⇒ 마지막 단계로 B_1, B_2 곱 ⇒ $B_3(d_0, d_n)$ cost (d_0, d_i, d_n)

↳ 각 부분문제는 $(0, i)$, (i, n)으로 식별 가능

부분문제 (i, n)에서 마지막 연산 k ⇒ (i, k), (k, n)으로 구분 가능 ⇒ 서로 다른 부분문제의 수: $\theta(n^2)$

```
mmTry2(dim, low, high)
    if(high - low = 1)
        bestCost = 0
    else
        bestCost = ∞
        for(k = low+1, k ≤ high-1, k++)
            a = mmTry2(dim, low, k)
            b = mmTry2(dim, k, high)
            c = dim(low) × dim(k) × dim(high)
            bestCost = min(bestCost, a+b+c)
    return bestCost
```

→ 모든 k에 대한 비용 검사

퇴각 검색 알고리즘 : 2^n의 시간 소요

ㄴ 이걸 DP로 바꾸면

```
mmTry2DP(dim, low, high, cost)
    if(high - low = 1)
        bestCost = 0
    else
        for(k = low+1, k ≤ high-1, k++)
            if(member(low, k) = fase)
                a = mmTry2DP(dim, low, k, cost)
            else
                a = retrieve(cost, low, k)
            if(member(k, high) = false)
                b = mmTry2DP(dim, k, high, cost)
            else
                b = retrieve(cost, k, high)
        c = dim(low) × dim(k) × dim(high)
        bestCost = min(bestCost, a+b+c)
    store(cost, low, high, bestCost)
    return bestCost
```

→ $n+1 \times n+1$ 행렬로 비용 저장

k 저장하기 위한 방법 필요

역위상순서에 따라 필요한 부분문제의 해를

미리 구해놓으면 더 효율적

<알고리즘 10.1> 최적 행렬 곱셈 순서

Input: 차원 $d_0, \cdots, d_n$을 저장하는 배열 dim, 행렬의 수 n

Output: multOrder 전달 후 계산, 최적 비용.

```
float matrixOrder(int[] dim, int n, int[] multOrder)
    int [][] last = new int[n+1][n+1]
    float [][] cost new float[n+1][n+1]
    int low, high, k, bestlast.
    float bestCost

    for(low=n-1, low≥0, low--)
        for(high=low+1, high≤n, high++)
            // 부분 문제 (low, high)의 해 계산 → cost[low][high], last[low][high]에 저장
            if(high-low ==1)
                bestCost = 0
                bestlast = -1
            else
                bestCost = ∞
            for(k=low+1, k≤high-1, k++)
                float a= cost[low][k]
                float b = cost[k][high]
                float c = multcost(dim[low], dim[k], dim[high])
                if(a+b+c < bestCost)
                    bestCost = a+b+c
                    bestlast = k
            cost[low][high] = bestCost
            last[low][high] = bestlast                → θ(n^3) 시간 복도에 해결
            extractOrderWrap(n, last, multOrder)
            return cost[0][n]
```

〈알고리즘 10.2〉 최적 행렬 곱셈 순서 추출.

Input: n, 10.1의 last

Output: 곱셈 순서 multorder

```
int multorderNext
extractmultorderWrap(n, last, multorder)
        multorderNext = 0
        extractorder(0, n, last, multorder)

extractorder(low, high, last, multorder)
        int k.
        if(high-low >1)
            k= last(low)(high)
            extractorder(low, k, last, multorder)
            extractorder(k, high, last, multorder)
            multorder(multorderNext) = k
            multorderNext ++.
```

10.4 최적 이진 탐색 트리의 구성

주어진 키들에 대한 탐색 빈도 수를 알 때 최적 이진 탐색 트리의 구성

BST → 각 노드의 키: 왼쪽 부트리보다는 크고, 오른쪽보다 작거나 같다. (중순위 운행 ⇒ 키가 증가하는 순서대로)

ㄴ BST T에서 키 k를 찾는데 걸리는 시간: C_i, k_i가 탐색될 확률이 P_i일 때, 평균 탐색 시간

$$A(T) = \sum_{i=1}^{n} P_i C_i$$

모든 키의 확률이 비슷하다면 최대한 균형잡힌 것이 유리.

ㄴ 그렇지 않을 때는 어떻게 배치?

탐색 빈도 수가 큰 값의 레벨을 최대한 작게 해야 유리 → 루트는 오히려 더 비효율적

key $k_1, k_2, \cdots, k_n$에 대해 (순서대로 정렬됨) k_k를 선택

⇒ 행렬곱 순서 찾기와 같다.

〈정의 10.3〉

1. 부분문제 (low, high)에 대해 루트가 k_r 일때, 최소 탐색 비용 $A(low, high, r)$
2. 부분문제 (low, high)에서 모든 가능한 루트에 대한 최소탐색비용 : $A(low, high)$
3. $P(low, high) = P_{low} + \cdots + P_{high}$로 정의 ⇒ 가중치

k_{low}, k_{high}에 대한 BST T의 가중탐색 비용 : W

T가 다른 트리의 부트리로서 T의 루트 깊이가 1 ⇒ T에 대한 가중 탐색 비용 ; $W + P(low, high)$

↳ T로 들어오는 모든 탐색은 T가 독립적인 트리일 때보다 비교를 한번 더 함.

T에 속한 키를 탐색할 확률 : $P(low, high)$

이를 이용해 재귀적으로 분할하면

$$A(low, high, r) = P_r + P(low, r-1) + A(low, r-1) + P(r+1, high) + A(r+1, high)$$

$$= P(low, high) + A(low, r-1) + A(r+1, high)$$

$$A(low, high) = \min\{A(low, high, r) \mid low \le r \le high\}$$

〈알고리즘 10.3〉 최적 이진 탐색 트리

input : 각 키에 대한 확률 $P_1, P_2, \cdots, P_n$을 저장한 배열 Prob, 키의 개수 n

output : $(n+2) \times (n+1)$ 이차원 배열 Cost와 root 받아서 계산. 첫 번째 인덱스 0은 제외

Cost(low)(high) : 키에 대한 최소 탐색 가중 비용   root(low)(high) : 키에 대한 루트

```
optimalBST(Prob, n, Cost, root)
    for(low = n+1, low ≥ 1, low--)
        for(high = low-1, high ≤ n, high++)
            bestChoice(Prob, Cost, root, low, high)
    return Cost
```

```
bestChoice (Prob, cost, root, low, high)   ← 부분문제
    if (high < low)
        bestchoice = 0
        bestroot = -1
    else
        bestChoice = ∞
    for (r = low, r ≤ high, r++)
        rCost = p(low, high) + cost(low)(r-1) + cost(r-1)(high)
        if (rCost < bestCost)
            bestCost = rCost
            bestroot = r
    cost(low)(high) = bestCost
    root(low)(high) = bestroot
    return
```

10.5 단어들의 행 분리

연속하는 일련의 단어들이 주어졌을 때 하나의 단락을 이루는 여러 개의 행들로 분리하는 문제.

↳ 모든 행에 삽입되는 여분의 공백 최소화 ⇒ 컴퓨터로 자동화된 식자에서 아주 중요

행 분리 문제: 연속하는 n개의 단어들의 길이 $w_1, \sim w_n$과 행의 폭 W

↳ 각 w_i는 단어의 끝에 하나의 공백을 포함한 길이로 가정

행의 폭 W는 행의 끝에 여분 공백 하나가 있는 것으로 간주

단어 $i \sim j$까지 하나의 행에 배치 ⇒ $w_i + \cdots + w_j \le W$

↳ 이 때 행에 있는 여분 공백 수 $X = W - (w_i + \cdots + w_j)$

앞에서 봤던 문제들과 매우 유사
- 재귀 호출을 하기전에 해가 사전에 있는지 검사
- 프로시저로 부터 복귀하기 전에 지금 해 저장

```
lineBreak(w, W, i, n, L)
  if(w_i + ... + w_j ≤ W)
      행 L에 모든 단어를 배치, 벌점 0
  else
      다음을 만족하는 모든 k>0에 대한 kPenalty의 최소값을 penalty로 둔다.
      w_i + ... + w_j ≤ W, X = W - (w_i + ... + w_{i+k-1})이고
      kPenalty = lineCost(X) + lineBreak(w, W, i+k, n, L+1)
      k_min = 위의 penalty가 가장 작은 k.
      단어 i부터 단어 i+k_min-1까지 행 L에 배치
  return penalty
```

10.6 동적 계획법을 이용한 알고리즘 개발

동적 계획법의 본질 : 부분 문제의 해를 구해서 이를 테이블에 저장 ← 이 부분문제의 해가 필요하면 다시 계산하지 않고 저장된 해 사용

동적 계획법 해의 개발

1. 재귀적 알고리즘과 같이 문제를 Top-down으로 해결하는 것이 유용 → 작은 문제의 해를 안다는 가정 하에, 큰 문제 해결
2. 작은 문제들의 해를 저장하여 반복 줄이는 것이 가능 ⇒ 이 해들을 저장하기 위한 사전식 구조 정의
3. 동적 계획법 프로시저의 시간 복잡도 → 부분 문제 그래프 깊이우선 탐색에 의해 부분문제의 수 / 에지의 수로 분석 가능
4. 사전식 구조에 대한 자료구조 결정
5. 가능하면 부분문제 그래프 분석하여 사전식 구조 엔트리 계산 최적화 ⇒ 역위상 순서
6. 사전식 구조에서 문제의 해를 얻는 방법 정의