0/1 배낭 문제로 알아보는 동적 계획법

정보컴퓨터공학부 201925111 김건호 PNU 스터디그룹 중간 보고서

서론

기존에 알고리즘 문제(PS)를 해결하면서, 시간 복잡도가 높은 경우에 대해 주로 `동적 계획법(Dynamic Programming)`을 사용해서 풀곤 했다. 하지만 `어떤 상태`에 대해 값을 `어떻게 저장해야 하는지` 설정하고, 어떻게 `점화식`을 세워야 할 지 어려움을 느꼈다. 이번 글에선 `언제 그리고 왜 동적 계획법을 쓰는가`에 대해 기술할 예정이다.

동적 계획법이란

`동적 계획법(Dynamic Programming, DP)`는 작은 하위 문제들을 통해, 복잡한 전체 문제를 해결하는 방식이다. 이를 위한 전제는, 문제의 최적 해결책이 하위 문제들의 최적 해결책으로 구성되어 있어야 한다. 이 명제는 전체 문제를 최적의 부분 문제로 나눌 수 있어야 함을 암시한다. 이는 상위 문제는 하위 문제에 의존함 또한 암시한다.

하위 문제를 통해 전체 문제를 해결

동적 계획법 : 하위 문제를 통해, 전체 문제를 해결하면 점이 무엇인가에 대해 생각해보자.

동적 계획법이 유리할 때는, `중복되는 계산 구조`가 있을 때 유리하다. 동적 계획법에서 해결한 부분 문제는 최적임을 의미하는데, `중복되는 계산 구조`를 발견한다면, 기존에 계산했던 값을 사용하면 되기 때문이다. 이는 해결한 부분 문제에 대해 정답을 `메모`했다가, 그대로 사용하는 방식과 유사하기 때문에 `메모이제이션(memoization)`이라고 부르기도 한다. `key`값은 현재 상태, `value`는 정답을 메모한다. 또한, 작은 문제에서부터 시작하기에, 문제의 크기가 줄어들고, 이는 `계산이 간단해짐`을 의미한다. 이렇게 찾은 최적의 부분 해들을 사용하여 `최적의 전체 해`를 얻을 수 있다.

동적 계획법(Dynamic Programming)을 사용해야 할 때

알고리즘 스터디에서 문제를 해결하며 느낀 바는 동적 계획법은 입력에 대해 지수 시간복잡도를 갖는 문제를, 입력에 대해 선형 시간복잡도로 변환할 수 있다는 것이다. 대표적인 문제,

`0/1 Knapsack problem(배낭 문제)`를 통해 알아보자.
[문제 링크 : 백준 12865(평범한 배낭)](https://www.acmicpc.net/problem/12865)
`0/1 배낭 문제`는 각 물건은 한 번만 선택할 수 있고, 주어진 물건들 중 일부를 선택하여 배낭에 넣을 때, 물건들의 무게 합이 배낭의 용량을 초과하지 않으면서 가치의 합을 최대화하는 문제이다. `물건을 n개, 가방이 버틸 수 있는 최고 무게를 w라고 하자.` 만약 문제를 `brute-force(완전 탐색)`으로 해결하려면 이는 입력에 대해 지수 시간복잡도를 갖는다. 왜냐하면 n개의 물건에 대해 전부 넣음과 넣지 않음을 계산하기 때문이다. 이는 O(2^n)으로 나타낼 수 있다.
<추가 설명>

a번째 물건을 n_a라고 하자. 첫 번째 물건에 대해 배낭에 넣거나 넣지 않거나 이다. 즉 2가지의 경우의 수를 갖는다. 2번째도 마찬가지다. Na번쨔 물건도 2가지 경우의 수를 갖는다. 이는 O(2^n)를 의미한다.
`이를 동적 계획법을 통해 해결해보자.` 본격적으로 문제 해결에 앞서, 현재 상태를 어떻게 정의해야 할까? 현재 상태를 잘 설정해야, 문제를 해결할 수 있다. 이 부분이 동적 계획법의 핵심이다. 그 다음으로, 설정한 상태를 토대로 전체로 도달할 수 있는 `점화식`을 세워야 한다.

탐욕법

>`탐욕법`은 눈 앞만 바라본다. 하지만 전체 조합을 고려해야 한다.

최고 값어치를 구하는 문제기에, 어떤 물건을 담았는지는 중요하지 않다. 현재 상태에서 하나 확정된 것은 현재 나의 배낭 무게이다. 또한, 현재 어떤 물건을 담았는지도 중요하다. 아래와 같은 예시를 살펴보자.
직관적으로 보았을 때, 4번 물건은 무조건 담아야 한다. 왜냐하면 무게 비율 가치가 제일 높기 때문이다. 그렇다면 이를
`탐욕적이게(greedy)`로 해결하면 될까? 정답은 `아니다`이다. 왜냐하면 아래 예시를 보자. `배낭의 용량을 9`이라고 하자.

물건	무게	가치	가치/무게 비율 (가치/무게)
1	3	4	1.33
2	4	5	1.25
3	5	7	1.4
4	2	3	1.5

`탐욕법`을 통해 문제를 해결하면 3번과 4번 물건을 넣을 것이다. 왜냐하면 가치/무게 비율이 제일 좋기 때문이다. 과연 이것이 정답일까? 즉, `10의 가치가 최대일까?` 정답은 `아니다.` 왜나하면, `1번 2번 4번` 물건을 넣는 것이 더 가치가 높기 때문이다. 즉, `탐욕적`으로 해결할 수 없다.

이러한 이유는, `탐욕법`은 각 물건을 부분적으로 담는 것에만 집중한다. 그러나, `0/1 배낭 문제`는 전체 물건을 한 번에 봐야 하는 조합적인 문제기 때문이다. 알고리즘이 지향하는 바가 완전히 다르기에 잘못된 접근이다.

0/1 배낭 문제 상태 설정

> `선택하지 않는 경우`와 `선택하는 경우`를 다 고려하며 상태를 설정해야 한다.

상태 설정을 해보자. 앞선 탐욕법의 실패에서 배낭 문제는 조합적인 문제임을 알았다. 일단 문제를 부분 문제로 나눠보자. 어떻게 나눌까? 일단 어떤 `물건를 보는가와 가방 무게`를 고려해야 한다. 하지만, `현재 어떤 물건`을 담았는지 상태를 다 기록하는 것은 결국 지수의 공간 복잡도를 가질 것이다. 왜냐하면 이는 경우의 수와 똑같은 문제기 때문이다.

배낭 문제에서 중요한 점은, 각 물건을 `선택하거나 선택하지 않는` 방식으로 해결할 수 있다는 것이다. 이때, `선택하지 않는 경우`와 `선택하는 경우`를 다 고려하며, `메모`를 잘 해보자. 현재 n번째 물건을 담았고, 가방의 무게를 Wn라고 하자. n+1번째 물건을 담는다면 가방의 무게는 Wn+Wn+1이다. 만약 담지 않는다면, 여전히 가방의 무게는 Wn일 것이다. 즉 물건을 순차적으로 고려하고, 다음 상태로 나아간다고 가정하면 물건의 번호와, 가방의 무게가 필요하다. 이를 `상태`로 정의해보자. 즉 메모를 기록할 자료 구조를 2차원 배열로 설정할 수 있게 된다. 배열 이름을 `dp`라고 하자.

```
dp[현재 몇 번째 물건을 고려하는지][현재 가방의 무게] = 최고 값어치
```

우리는 2차원 배열의 제일 끝을 보면 정답을 찾을 수 있게 되었다.

0/1 배낭 문제 점화식

> `선택하지 않는 경우`와 `선택하는 경우`를 다 고려하며 점화식을 세워보자. 조합적인 모든 것을 고려해야 한다. n번째 물건과 현재 가방의 무게를 w라고 하자. 다음 상태 즉, n+1번째 물건을 고려할 때, 넣거나, 넣지 않아도 된다. 즉 이를 점화식으로 써보자.

```
dp[n+1][w] = max(dp[n][w] , dp[n][w-(현재 보는 물건의 무게)] + (현
```

로 쓸 수 있다. dp[n][w]는 `n+1번째 물건을 넣지 않는다`이고, 후자는 `n+1번째 물건을 넣는 경우`이다. 이 때, `IndexOutOfBoundsException`을 조심하도록 해야 한다. 이제 이를 코드로 작성해보자.

소스 코드

`python3` 을 기준으로 작성하였다.

```python
import sys

    n,k=map(int,sys.stdin.readline().split()) #물건 수, 제한
    dp=[[0 for _ in range(k+1)] for _ in range(n+1)]
    answer=0
    for i in range(1,n+1) :
        w,v = map(int,sys.stdin.readline().split()) #weight,
        for j in range(1,k+1) :
            if j>=w :
                dp[i][j] = max(dp[i-1][j], dp[i-1][j-w] + v)
            else :
                dp[i][j] = dp[i-1][j]

    print(dp[n][k])
```

마무리하며

`동적 계획법`은 계산이 중복되는 구조에서 사용할 수 있다. 또한, `부분은 최적을 가져야 한다`는 원칙을 지켜야 한다. 또한 `지수 시간 복잡도`를 갖는 문제에서 `선형 시간 복잡도`로 변환할 수 있다. `0/1 배낭 문제` 같은 경우는 처음에 상태를 정의하는 것이 꽤나 까다로웠다. 특히, `현재 가방의 무게라는 것`이 처음엔 직관적으로 와닿지 않았다. `동적 계획법`문제의 알파이자 오메가는 `상태 정의와 저장하는 값`이라고 생각한다. 이를 잘못 정의하면, 잘못된 값들이 저장되어 전체 해결이 불가능해지기 때문이다. 또한, 계산된 값을 어떻게 저장하고 재사용할지 결정하는 것이 중요하다.

스터디 후기

알고리즘 스터디를 통해 여러 알고리즘 문제를 다루었다. 최장 부분 증가 수열(LIS), 최장 부분 공통 수열(LCS), 최단 경로 집중 공략, BFS, DFS 유형 분석, 투포인터 등등...

각자 공부해온 문제를 최대한 자세하게 설명하였다. 특히 필자는 `동적 계획법`을 주로 공부하였다.

동적 계획법을 통해 여러 복잡한 문제들을 효율적으로 해결할 수 있는 방법을 배웠고, 특히 문제의 **상태 정의**와 **중복 계산을 피하기 위한 값 저장**이 중요하다는 것을 깨달았다. 또한, 동적 계획법을 적용하는 문제들은 처음에는 어떻게 상태를 나누고, 어떤 방식으로 값을 저장할지 고민이 많았지만, 여러 문제를 풀어보며 점차적으로 직관이 생겼다.

스터디에서 다룬 `0/1 배낭 문제`와 같은 문제들은 상태 정의와 최적화 문제를 어떻게 분리하고, 중복 계산을 없애는지를 명확히 보여주는 좋은 예시였다. `동적 계획법`을 풀 때마다, 문제를 어

떻게 분할하고 최적의 해결 방법을 도출할지 고민하면서 점점 더 알고리즘적인 사고가 발전하는 것을 느꼈다.