Dynamic Programming

- **전체 문제에 대해서 가장 작은 1번 문제에 대한 답을 찾고 이것을 이용하여 2번 문제에 대한 답을 찾고 이것을 이용하여 3번 문제에 대한 답을 찾는다. 이것을 n번까지 반복하여 최종적으로 전체 문제에 대한 답을 구한다.**
- **DP문제는 예시의 첫 번째 원소부터 시작하여 각 원소를 마지막 항일 때로 가정하여 해를 찾는다. 이 과정을 주어진 데이터 전체까지 적용하여 일반화된 식을 찾는다.**
- **Bottom-Up & Top-Down**
  - **네트워크 선 자르기 문제(Bottom-Up 방식)**
    - **선의 길이 만큼에 해당하는 일차원 리스트 선언**
    - 직관적으로 알 수 있는 것은 직접 초기화를 해두고 풀이를 시작한다.
      - ex) 1번 문제에 대한 답, 2번 문제에 대한 답..,
    - ex) 길이가 7인 네트워크 선
      - 길이가 7인 일차원 리스트 dy를 선언
      - dy[1] = 1, dy[2] = 2, ...(직관적으로 알 수 있는 부분은 직접 초기화)
      - dy[3] = ?? -> 길이가 3인 선을 자르는 방법의 개수
        - 선의 가장 오른쪽을 1m로 자른 경우: 남은 부분은 2m -> dy[2] = 2이므로 총 2가지
        - 선의 가장 오른쪽을 2m로 자른 경우: 남은 부분은 1m -> dy[1] = 1이므로 총 1가지
        - 따라서, 총 3가지. dy[3] = 3
  - **네트워크 선 자르기 문제(Top-Down 방식)**
    - **선의 길이 만큼에 해당하는 일차원 or 이차원 리스트 선언**
    - 직관적으로 알 수 있는 것은 직접 초기화를 해두고 풀이를 시작한다.
    - memoization: **재귀 함수 호출**. 구한 값을 초기에 선언한 리스트에 저장하고 다음 값을 구할 때 재귀 함수를 호출하여 리스트에 저장된 값을 이용한다.

- 최대 부분 증가수열(LIS: Longest Increasing Subsequence)

- **ex) 5 3 7 8 6 2 9 4**
- 주어진 숫자들을 저장하고 있는 일차원 리스트 arr를 선언한다.
- 리스트의 첫 번째 원소부터 그 원소가 내가 만들고자 하는 증가수열의 마지막 항이라 가정하고 LIS를 생성한다.
  - 0번 원소(5)가 마지막 항인 경우: LIS – 5. 길이: 1
  - 1번 원소(3)가 마지막 항인 경우: LIS – 3. 길이: 1
  - 2번 원소(7)가 마지막 항인 경우: LIS – ㅁ 7

    ㅁ: 3일 때 – 3이 마지막 항인 경우의 가짓수는 1가지. 따라서, LIS의 길이: 2

    ㅁ: 5일 때 – 5가 마지막 항인 경우의 가짓수는 1가지. 따라서, LIS의 길이: 2
  - 3번 원소(8)가 마지막 항인 경우: LIS - ㅁ 8

    ㅁ: 7일 때 - 2가지. 따라서, LIS의 길이: 3

    ㅁ: 3일 때 - 1가지. 따라서, LIS의 길이: 2

    ㅁ: 5일 때 - 1가지. 따라서, LIS의 길이: 2
- 위 과정을 리스트의 마지막 원소까지 반복한다.
- **DP방식으로 해결하기(위 과정과 연결됨)**
  - 위에서 선언한 리스트 arr의 각 원소를 마지막 항으로 하는 최대 부분 증가수열의 길이를 저장하는 일차원 리스트 dy를 선언한다.
  - 직관적으로 알 수 있는 부분은 직접 초기화하고 시작한다. dy[1] = 1
  - dy[1] = 1, dy[2] = 1, dy[3] = 2, dy[4] = 3, dy[5] = 2 …
  - 위 과정을 주어진 데이터의 개수만큼 반복하여 리스트 dy 전체를 초기화한다.

■ 최대 선 구하기 문제 – 위와 동일

■ 가장 높은 탑 쌓기

- 밑면의 넓이가 큰 벽돌이 아래에 위치해야 한다.
- 주어진 데이터를 2차원 리스트 arr에 저장하는데 이때 밑면의 넓이에 대하여 내림차순으로 정렬한다. 벽돌의 무게만 고려하여 문제를 해결해 나가면 된다.

◆ 일차원 리스트 dy는 각 인덱스에 해당하는 벽돌이 꼭대기에 위치한 경우 최대 높이를 저장한다.

◆ **비교 대상이 여러 개인 경우 비교 대상을 줄이는 방법을 생각해본다. (ex 정렬)**

- 알리바바와 40인의 도둑(Bottom-Up)

■ 최단거리 이동(현재 위치에서 오른쪽 or 아래쪽으로만 이동할 수 있음)

■ **위에서와 같이 항상 마지막 항의 입장에서 생각하여 해를 구하고 일반화한다.**

■ 입력 데이터는 이차원 리스트 형태로 저장

■ **2차원 리스트 dy를 선언. 1차원 리스트로 선언하게 되면 사용할 수 있는 정보가 부족함.**

- 알리바바와 40인의 도둑(Top-Down)

■ 재귀 함수 DFS를 정의하여 구현

■ **memoization을 위해 2차원 리스트 dy를 선언(재귀를 사용하므로 memoization이 필요함)**

■ 재귀 함수 DFS를 바로 return하지 말고 dy에 값을 저장한 후에 return한다.

■ 최종적으로 구해야 하는 해에 대해서 상태 트리를 그려가면서 문제를 해결한다.

■

- Knapsack 알고리즘

■ 가방 문제

◆ **DP문제는 항상 일차원 또는 이차원 리스트 dy(dynamic table)를 모두 0으로 초기화하고 값을 계속 저장해 나간다.**

◆ **dy 리스트의 index와 각 원소가 무엇을 의미하는지 파악해야한다!**

◆ **자원이 무한개 존재하는 경우**

◆ dy 리스트의 index는 가방의 무게, 원소는 가방에 담을 수 있는 보석의 최대 가치를 의미한다.

◆ dy의 index j는 가방의 무게를 의미하므로 전체 가방 무게에서 보석의 무게를 빼

면서 남은 가방 무게를 파악하고 남은 가방 무게의 최대 가치(dy[j-w])와 현재 담은 보석의 가치를 계산한다.

- index i는 각 보석들을 반복문을 통해 방문하는 용도로 사용.
- 예시
  - 4 11 5 12 3 8 6 14 4 8
- 직관적으로 알 수 있는 것은 초기화한다.
  - 5 12 보석 1개만 있을 경우
    - dy[5] = dy[5 – 5] + 5g 보석의 가치 = dy[0] + 12 = 12
    - dy[6] = dy[6-5] + 5g 보석의 가치 = dy[1] + 12 = 12

      …

      dy[10] = dy[5] + 5g 보석의 가치 = 24
  - 3 8 보석을 추가할 때
    - dy[3] = dy[3 – 3] + 3g 보석의 가치 = 8
    - dy[4] = dy[4 – 3] + 3g 보석의 가치 = 8
    - **dy[5] = dy[5 – 3] + 3g 보석의 가치 = 8**
    - **dy[6] = dy[6 – 3] + 8 = 8 + 8 = 16 -> update!**
  - 최대 가치를 찾아야하므로 dy의 원소는 최댓값을 가져야한다. dy의 원소값이 새로 추가되는 값보다 크면 그대로 유지한다.
  - … 6 14 보석, 4 8 보석까지 위 과정을 반복한다.
  - dy의 가장 마지막 원소가 정답!

■ 동전 교환

- **자원이 무한개 존재하는 경우**
- 가방 문제와 동일한 알고리즘
- **dy리스트의 index와 원소가 무엇을 의미하는지 잘 생각하자!**

■ 최대 점수 구하기

- ◆ **한 유형당 한 개만 풀 수 있다. -> 자원이 유한개 존재하는 경우**
- ◆ dy(dynamic table)를 이차원 리스트로 선언한다. dy의 0행은 원소가 모두 0 -> **이전 문제에 대한 정보가** 필요하기 때문에!
- ◆ dy[i][j]: i – 문제 종류, j – 푸는데 걸리는 시간
- ◆ 다음 문제를 풀 때, 이전 문제를 풀었을 때 얻은 점수를 그대로 복사한다.
- ◆ But! 실제 대회에서는 dy를 이차원 리스트로 선언하지 않고 일차원으로 선언한다!.
  - 리스트의 크기가 커질수록 메모리가 많이 커지고 시간 복잡도가 증가함
  - *****일차원 리스트로 해결하는 방법이 존재함*** -> 실전!**
  - dy를 모두 0으로 초기화
  - dy를 앞에서부터 채워 나가면 중복되는 경우를 피할 수 없음! -> 뒤에서부터 채워 나간다!

■ 플로이드 와샬 알고리즘

◆ 그래프 최단거리

- 그래프의 모든 종점에서 모든 종점(자기 자신 제외)으로 가는데 드는 최소 비용 → **이차원 리스트로 선언하여 노드 간의 비용을 기록한다**.
- 이차원 리스트 dis 선언: dis[i][j] – i 노드에서 j 노드로 직접 이동할 수 있을 때 거리. 직접 갈 수 없으면 Max값으로 초기화
- **i 에서 j로 가는 최소 비용과 i에서 k를 거쳐 j로 가는 최소 비용 중 최솟값을 dis에 저장한다. -> 코드로는 3중 for문으로 구현**
  - k는 1번 노드에서 n번 노드까지 해당된다.
  - k가 n까지 증가하면서 이차원 리스트 dis를 계속 갱신한다.

◆ 회장 뽑기

- 그래프 최단거리 문제와 같은 알고리즘
- 두 회원이 친구 사이이면서 동시에 친구의 친구사이이면 두 사람은 친구 사이이다. == **최단 거리**를 의미

■ 위상 정렬(그래프)

- ◆ 일의 선후관계를 유지하면서 전체 일의 순서를 짜는 알고리즘
- ◆ 방향 그래프 - 인접 행렬로 표현
- ◆ 차수: 어떤 노드에 연결된 edge의 개수
- ◆ **진입 차수: 어떤 노드로 들어오는 edge의 개수**
  - 각 노드에 대해 진입 차수를 나타내는 일차원 리스트를 선언한다.
  - 리스트의 index는 작업의 번호
- ◆ **queue를 활용!**
  - 리스트의 원소가 0인 index들을(진입 차수가 0인 노드) queue에 모두 삽입!
  - queue에 삽입된 원소들을 하나씩 삭제하면서(작업을 마쳤다는 의미) 그 원소와 연결된 노드들의 진입 차수도 1씩 감소. 노드의 진입 차수가 0이 되는 순간 queue에 삽입!
  - 리스트의 원소가 모두 0이 될 때까지(모든 노드의 진입 차수가 0) 위 과정 반복