


Dynamic Programming

# Index	5
📅 CreatedAt	@September 28, 2022
👤 Person	A Ally Hyeseong Kim
🌟 Status	Done
🏷️ Tags	Dynamic Programming Java Python
📅 UpdatedAt	@September 28, 2022

References

파이썬 알고리즘 인터뷰

2021 세종도서 학술부문 선정작. 현업과 실무에 유용한 주요 알고리즘 이론을 깊숙이 이해하고, 파이썬의 핵심 기능과 문법까지 상세하게 이해할 수 있는 취업용 코딩 테스트를 위한 완벽 가이드다. 200여 개가 넘는...

 <https://www.aladin.co.kr/shop/wproduct.aspx?ItemId=2454958>
26



References

1. Dynamic Programming
2. 다이나믹 프로그래밍 방법론
3. Greedy Algorithm vs Dynamic Programming: Knapsack Problem

1. Dynamic Programming

Dynamic Programming Algorithm은 문제를 각각의 작은 문제로 나누어 해결한 결과를 저장해뒀다가 나중에 큰 문제의 결과와 합하여 풀이하는 알고리즘이다.

- **최적 부분 구조**를 갖고 있는 문제를 풀이할 수 있다.
 - **최적 부분 구조**: 문제의 최적 해결 방법이 부분 문제에 대한 최적 해결 방법으로 구성되는 경우

Algorithm	문제 특징	대표적인 문제
Dynamic Programming Algorithm	최적 부분 구조 중복된 하위 문제	0-1 배낭 문제 피보나치 수열 Dijkstra Algorithm
Greedy Algorithm	최적 부분 구조 탐욕 선택 속성	분할 가능 배낭 문제 Dijkstra Algorithm
Divide and Conquer	최적 부분 구조	Merge Sort Quick Sort

2. 다이나믹 프로그래밍 방법론

2.1. 상향식(Bottom-up)

상향식 방법은 더 작은 하위 문제부터 살펴본 다음, 작은 문제의 정답을 이용해 큰 문제의 정답을 풀어나간다. (**Tabulation**)

```
def fib(n):
    dp[0] = 0
    dp[1] = 1

    for i in range(2, n + 1):
        dp[i] = dp[i - 1] + dp[i - 2]

    return dp[n]
```

2.2. 하향식(Top-Down)

하향식 방법은 하위 문제에 대한 정답을 계산했는지 확인해가며 문제를 풀어나간다. (**Memoization**)

```
def fib(n):
    if n <= 1:
        return n

    if dp[n]:
        return dp[n]

    dp[n] = fib(n - 1) + fib(n - 2)

    return dp[n]
```

- 이미 풀어봤는지 확인하여 재사용하는 효율적인 방식이다.

3. Greedy Algorithm vs Dynamic Programming: Knapsack Problem

Knapsack Problem은 배낭에 담을 수 있는 무게의 최댓값(15kg)이 정해져 있을 때, 각 짐의 가치와 무게가 있는 짐을 배낭에 넣을 때 가격의 합이 최대가 되도록 짐을 고르는 방법을 찾는 문제이다.

- **0-1 Knapsack Problem** : 짐을 쪼갤 수 없는 경우 → 탐욕 선택 속성이 없고 중복된 하위 문제들 속성을 가지므로 **Dynamic Programming**으로 풀어야 한다.

```
def zero_one_knapsack(cargo):
    capacity = 15
    pack = []

    for i in range(len(cargo) + 1):
        pack.append([])
        for c in range(capacity + 1):
            if i == 0 and c == 0:
                pack[i].append(0)
            elif cargo[i - 1][1] <= c:
                pack[i].append(max(cargo[i - 1][0] + pack[i - 1][c - cargo[i - 1][1]], pack[i - 1][c]))
            else:
                pack[i].append(pack[i - 1][c])
```

```
return pack[-1][-1]
```

- `cargo = [(value, weight)]`