Greedy

# Index	1
CreatedAt	@September 28, 2022
Person	A Ally Hyeseong Kim
Status	In Progress
Tags	Greedy 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

https://www.aladin.co.kr/shop/wproduct.aspx?ItemId=245495826

References
1. Greedy Algorithm
2. Shortest Path Problem: Dijkstra Algorithm
3. Greedy Algorithm vs Dynamic Programming
4. Greedy Algorithm vs Dynamic Programming 1: Knapsack Problem
5. Greedy Algorithm vs Dynamic Programming 2: Coin-Change Problem

1. Greedy Algorithm

> `Greedy Algorithm`은 global optima를 찾기 위해 각 단계에서 local optima인 선택을 하는 heuristic 문제 해결 알고리즘이다.

- `Greedy Algorithm`을 적용할 수 있는 문제들은 `Greedy Choice Property`를 갖고 있는 `Optimal Substructure`이다.
  - `Greedy Choice Property`: 앞의 선택이 이후 선택에 영향을 주지 않는 것

    - `Optimal Substructure`: 최적 해결 방법이 부분 문제에 대한 최적 해결 방법으로 구성되는 경우

---

2. Shortest Path Problem: Dijkstra Algorithm

> `Shortest Path Problem`은 각 edge의 가중치 합이 최소가 되는 두 vertex(혹은 node) 사이의 경로를 찾는 문제이다.

- `Dijkstra Algorithm`은 항상 node 주변의 최단 경로만을 선택하는 대표적인 `Greedy Algorithm` 중 하나로 단순하고 실행 속도가 빠르다.
    - `Breath-First Search(BFS)`를 이용하여 node 주변을 탐색한다.

```
function Dijkstra(Graph, source):
    // 초기화
    dist[source] <- 0

    create vertex priority queue Q

    for each vertex v in Graph:
        if v != source
            // source에서 v까지 아직 모르는 길이
            dist[v] <- INFINITY
            // v의 이전 노드
            prev[v] <- UNDEFINED
        Q.add_with_priority(v, dist[v])

    // 메인 루프
    while Q is not empty:
        // 최고의 꼭짓점을 제거하고 반환
        u <- Q.extract_min()
        // Q에 여전히 남아 있는 v에 대해서만
        for each neighbor v of u:
            alt <- dist[u] + length(u, v)
            if alt < dist[v]
                dist[v] <- alt
                prev[v] <- u
                Q.decrease_priority(v, alt)

    return dist, prev
```

---

3. Greedy Algorithm vs Dynamic Programming

- `Greedy Algorithm`은 각 단계마다 local optima를 찾아 문제를 더 작게 줄여나간다.

- `Dynamic Programming`은 하위 문제에 대한 최적의 솔루션을 찾고 이 결과들을 결합한 정보를 바탕으로 global optima를 찾는다.

---

4. Greedy Algorithm vs Dynamic Programming 1: Knapsack Problem

> `Knapsack Problem`은 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.

- `Fractional Knapsack Problem`: 짐을 쪼갤 수 있는 경우 → `Greedy Algorithm`으로 풀이할 수 있다.

```
# cargo = [(가격, 무게), ...]
# 배낭에 담을 수 있는 무게 최대값: 15
def fractional_knapsack(cargo):
    capacity = 15
    pack = []
    # 단가(=가격/무게) 계산하여 정렬
    for c in cargo:
        pack.append((c[0] / c[1], c[0], c[1]))
    pack.sort(reverse=True)
    # 단가 순으로 Greedy Algorithm 이용
    total_value: float = 0
    for p in pack:
        if capacity >= p[2]:
            capacity -= p[2]
            total_value += p[1]
        else:
            fraction = capacity / p[2]
            total_value += p[1] * fraction
            break
    return total_value
```

- `0-1 Knapsack Problem`: 짐을 쪼갤 수 없는 경우 → `Dynamic Programming`으로 풀이해야 한다.

---

5. Greedy Algorithm vs Dynamic Programming 2: Coin-Change Problem

- 동전이 이전 액면의 배수 이상인 경우 → `Greedy Algorithm`으로 풀이할 수 있다.
  - ex. 10원, 50원, 100원 → 10원 * 16개 vs `100원 + 50원 + 10원`

- 동전이 이전 액면의 배수 이상이 아닌 경우 → `Dynamic Programming`으로 풀이해야 한다.
  - ex. 10원, 50원, 80원, 100원 → `80원 * 1개` vs 100원 + 50원 + 10원