06 강

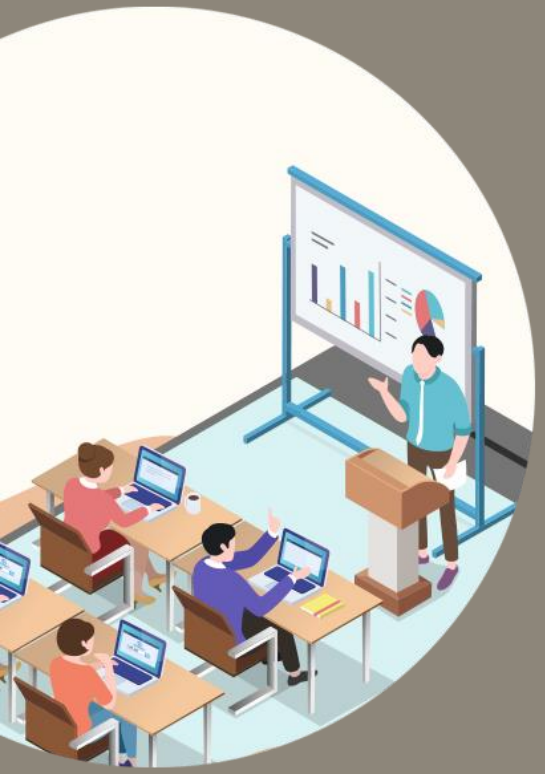
알고리즘과 자료구조

탐욕 알고리즘

서울과학기술대학교 신일훈 교수

한국방송통신대학교 프리임칼리지

학습목표



1. 탐욕 알고리즘의 개념을 이해한다.
2. 탐욕 알고리즘을 적용하여 거스름돈 구하는 알고리즘을 설계할 수 있다.
3. 탐욕 알고리즘을 적용하여 knapsack 문제를 해결하는 알고리즘을 설계할 수 있다.

01

탐욕 알고리즘 개념

1. 탐욕 알고리즘 개념

- 탐욕 (greedy) 전략 개념
  - 최적해를 찾기 위한 방법 중 하나 (brute-force, greedy, …)
  - brute-force는 모든 경우를 탐색하고 이 중 최적의 해를 찾음.
    - 최근접 쌍, 여러 도시 방문하는 최소 경로, knapsack 등
    - 모든 경우의 수가 너무 많은 경우,
      모든 경우를 탐색하는 시간이 오래 걸릴 수 있음

1. 탐욕 알고리즘 개념

❑ 탐욕 (greedy) 전략 개념

- greedy는 모든 경우를 고려하는 것이 아니라, 그 순간에 최적이라고 생각되는 것을 선택함.
  - 장기를 둘 때 몇 수 앞까지 보는 것이 아니라, 현 상태에서 최적이라고 판단되는 결정을 함.
  - 여러 도시를 방문해야 할 때, 이동 거리를 줄이기 위해 현재 위치에서 가장 가까운 도시부터 방문함 (전체 지도를 고려하지 않음..)
  - 결과적으로 global optimum이 아닌 local optimum (최적해의 근사값)을 도출할 가능성이 큼.
  - 장점은 brute-force보다 빠른 시간 안에 솔루션을 도출

02

거스름돈

2. 거스름돈

- 거스름돈으로 V원을 돌려줘야 할 때 동전 개수를 최소로 하는 솔루션을 구하시오.

  - 동전의 종류는 500, 100, 10, 1원이며,
    모든 동전은 무한히 사용할 수 있다고 가정.
  - 가령 580원을 거슬러야 한다면,
    500 * 1 + 10 * 8 = 총 9개의 동전 필요

    => 최적해

2. 거스름돈

- 거스름돈으로 V원을 돌려줘야 할 때 동전 개수를 최소로 하는 솔루션을 구하시오.
  - brute-force?
    - 580원을 만드는 모든 경우를 탐색하여 동전 개수를 최소화하는 솔루션 선택
    - 코딩도 쉽지 않고 시간도 오래 걸림
  - Greedy?

2. 거스름돈

- 거스름돈으로 V원을 돌려줘야 할 때 동전 개수를 최소로 하는 솔루션을 구하시오.

아이디어

동전종류: 500, 100, 10, 1

거스름돈: 580원

1. 남은 동전들 중, 액면가가 가장 높은 동전 선택.
2. 액면가*N이 잔액보다 작은 N을 구한다. (N이 해당 동전의 개수가 됨).
3. 잔액을 업데이트
4. 1-3번을 잔액이 0이 될 때까지 반복

2. 거스름돈

- 거스름돈으로 V원을 돌려줘야 할 때 동전 개수를 최소로 하는 솔루션을 구하시오.

의사코드

입력에 대한 가정: 동전 종류는 액면가 기준으로 내림차순으로 정렬되어 전달됨

예> coins = [500, 100, 10, 1]

2. 거스름돈

- 거스름돈으로 V원을 돌려줘야 할 때 동전 개수를 최소로 하는 솔루션을 구하시오.

의사코드

```
def min_coins_greedy(coins, change) :
    chosen = []  # 액면가 별로 선택된 동전 개수를 저장하는 리스트
    for coin in coins :
        count = change // coin # 몫을 구함
        chosen.append(count)
        change -= count * coin
    return chosen
```

2. 거스름돈

- 거스름돈으로 V원을 돌려줘야 할 때 동전 개수를 최소로 하는 솔루션을 구하시오.

의사코드

```
def min_coins_greedy(coins, change) :
    chosen = []  # 액면가 별로 선택된 동전 개수를 저장하는 리스트
    for coin in coins :
        count = change // coin # 몫을 구함
        chosen.append(count)
        change -= count * coin
    return chosen
```

최악 시간복잡도 ?

2. 거스름돈

- 거스름돈으로 V원을 돌려줘야 할 때 동전 개수를 최소로 하는 솔루션을 구하시오.

의사코드

```
def min_coins_greedy(coins, change) :
    chosen = []  # 액면가 별로 선택된 동전 개수를 저장하는 리스트
    for coin in coins :
        count = change // coin # 몫을 구함
        chosen.append(count)
        change -= count * coin
    return chosen
```

최악 시간복잡도 : O(N)

2. 거스름돈

- 거스름돈으로 V원을 돌려줘야 할 때 동전 개수를 최소로 하는 솔루션을 구하시오.

파이썬 코드

```
def min_coins_greedy(coins, change) :
    chosen = []
    for coin in coins :
        count = change // coin
        chosen.append(count)
        change -= count * coin
    return chosen
```

2. 거스름돈

- 거스름돈으로 V원을 돌려줘야 할 때 동전 개수를 최소로 하는 솔루션을 구하시오.

파이썬 코드

```
coins = [500, 100, 10, 1]
changes = 1534

print("잔돈: ", changes)
print("동전 종류", coins)
print("동전 개수", min_coins_greedy(coins, changes))
```

2. 거스름돈

- 거스름돈으로 V원을 돌려줘야 할 때 동전 개수를 최소로 하는 솔루션을 구하시오.

파이썬 코드 실행

```
In [98]: runfile('D:/data/재정/주식소스/stock/
result/untitled3.py', wdir='D:/data/재정/주식소스/
stock/result')
잔돈:  1534
동전 종류 [500, 100, 10, 1]
동전 개수 [3, 0, 3, 4]
```

2. 거스름돈

거스름돈 구하는 greedy 알고리즘

- 장점
  - 시간복잡도가 낮음 (모든 경우를 탐색하지 않기 때문)
  - local optimal solution을 찾을 수 있음
- 단점
  - global optimal solution을 못 찾을 수 있음

2. 거스름돈

- global optimal solution을 못 찾는 경우

파이썬 코드

```
coins = [60, 50, 5]
changes = 250

print("잔돈: ", changes)
print("동전 종류", coins)
print("동전 개수", min_coins_greedy(coins, changes))
```

2. 거스름돈

- global optimal solution을 못 찾는 경우

파이썬 코드 실행

```
In [105]: runfile('D:/data/재정/주식소스/stock/
result/untitled3.py', wdir='D:/data/재정/주식소스/
stock/result')
잔돈:  250
동전 종류 [60, 50, 5]
동전 개수 [4, 0, 2]
```

03

Knapsack

3. Knapsack

- 배낭에 넣을 수 있는 최대 무게: W, N개의 물건 (value , weight)
  배낭에 넣을 수 있으면서 가치의 총합을 최대로 하는 물건의 조합을 구하시오.

아이디어

names = ['A', 'B', 'C', 'D', 'E'], values = [10, 30, 20, 14, 23]

weights = [5, 8, 3, 7, 9], max_weight = 20

3. Knapsack

brute-force 아이디어

names = [‘A', ‘B', ‘C', ‘D', ‘E’], values = [10, 30, 20, 14, 23]
weights = [5, 8, 3, 7, 9], max_weight = 20

- 모든 경우의 수에 대해 가치의 총합과 무게의 총합을 각각 구하자.
  1. A
  2. A, B
  3. A, B, C
  4. A, B, C, D
  5. A, B, C, D, E
  6. …
- 무게의 총합이 최대 무게 이하인 경우들 중에서 가장 가치가 높은 경우를 선택한다.

3. Knapsack

greedy 아이디어1

names = [‘A', ‘B', ‘C', ‘D', ‘E’], values = [10, 30, 20, 14, 23]
weights = [5, 8, 3, 7, 9], max_weight = 20

- 무게 상관 없이, 가치가 가장 높은 물건부터 넣자 .
- 단, 해당 물건을 넣을 때 최대 무게를 초과하면
  이 물건은 넣지 않는다.

3. Knapsack

greedy 아이디어2

names = [‘A', ‘B', ‘C', ‘D', ‘E’], values = [10, 30, 20, 14, 23]
weights = [5, 8, 3, 7, 9], max_weight = 20

- 단위 무게 당 가치가 가장 높은 물건부터 넣자.
- 단, 해당 물건을 넣을 때 최대 무게를 초과하면
  이 물건은 넣지 않는다.

3. Knapsack

의사코드1 - value가 높은 물건부터 넣자

names = [‘A', ‘B', ‘C', ‘D', ‘E’], values = [10, 30, 20, 14, 23]
weights = [5, 8, 3, 7, 9], max_weight = 20

1. item 클래스 정의 : (name, value, weight)
   - (‘A’, 10, 5)
   - (‘B’, 30, 8),
   - …

3. Knapsack

의사코드1 - value가 높은 물건부터 넣자

names = [‘A', ‘B', ‘C', ‘D', ‘E’], values = [10, 30, 20, 14, 23]
weights = [5, 8, 3, 7, 9], max_weight = 20

2. items 리스트에 item 객체들을 추가
   - items = [(‘A’, 10, 5), (‘B’, 30, 8), (‘C’, 20, 3), (‘D’, 14, 7), (‘E’, 23, 9)]
3. items 리스트를 item의 value를 기준으로 내림차순 정렬

⇒ sorted_items = [(‘B’, 30, 8), (‘E’, 23, 9), (‘C’, 20, 3), (‘D’, 14, 7), (‘A’, 10, 5)]

3. Knapsack

의사코드1 - value가 높은 물건부터 넣자

```
cur_weight = 0
cur_value = 0
chosen_items = []
for item in sorted_items :
    if (cur_weight  + item. weight <= max_weight) :
        cur_value += item.value
        cur_weight += item.weight
        chosen_items.append(item)
return (chosen_items, cur_value, cur_weight)
```

3. Knapsack

의사코드1 - value가 높은 물건부터 넣자

names = ['A', 'B', 'C', 'D', 'E'], values = [10, 30, 20, 14, 23]
weights = [5, 8, 3, 7, 9], max_weight = 20

1. item 클래스 정의 : (name, value, weight)
   - ('A', 10, 5)
   - ('B', 30, 8),
   - ...

최악 시간복잡도 :
O(N)

3. Knapsack

의사코드1 - value가 높은 물건부터 넣자

names = ['A', 'B', 'C', 'D', 'E'], values = [10, 30, 20, 14, 23]
weights = [5, 8, 3, 7, 9], max_weight = 20

2. items 리스트에 item 객체들을 추가
   - items = [('A', 10, 5), ('B', 30, 8), ('C', 20, 3), ('D', 14, 7), ('E', 23, 9)]
3. items 리스트를 item의 value를 기준으로 내림차순 정렬

⇒ sorted_items = [('B', 30, 8), ('E', 23, 9), ('C', 20, 3), ('D', 14, 7), ('A', 10, 5)]

최악 시간복잡도 :
$O(N^2)$ 또는
$O(N\log N)$

3. Knapsack

의사코드1 - value가 높은 물건부터 넣자

```
cur_weight = 0
cur_value = 0
chosen_items = []
for item in sorted_items :
    if (cur_weight  + item. weight <= max_weight) :
        cur_value += item.value
        cur_weight += item.weight
        chosen_items.append(item)
return (chosen_items, cur_value, cur_weight)
```

최악 시간복잡도 :
O(N)

3. Knapsack

파이썬코드1 - value가 높은 물건부터 넣자

```
class Item(object):
    def __init__(self, name, value, weight):
        self.name = name
        self.value = value
        self.weight = weight
```

3. Knapsack

파이썬코드1 - value가 높은 물건부터 넣자

```
class Knapsack(object):
    def __init__(self, names, values, weights, max_weight):     #아이템들을생성함
        self.items = []
        self.max_weight = max_weight
        for i in range(len(names)) :
            item = Item(names[i], values[i], weights[i])
            self.items.append(item)
```

3. Knapsack

파이썬코드1 - value가 높은 물건부터 넣자

```
class Knapsack(object):
    def findBestCaseGreedy1(self):
        self.items.sort(reverse=True, key=lambda x:x.value)
        value = 0
        weight = 0
        chosen_items = []
```

3. Knapsack

파이썬코드1 - value가 높은 물건부터 넣자

```
def findBestCaseGreedy1(self):
    for item in self.items :
        if (weight + item.weight <= self.max_weight) :
            chosen_items.append(item.name)
            weight += item.weight
            value += item.value

    return (chosen_items, value, weight)
```

3. Knapsack

파이썬코드1 - value가 높은 물건부터 넣자

```
names = ['A', 'B', 'C', 'D', 'E', 'F', 'G', 'H']
values = [10, 30, 20, 14, 23, 11, 15, 18]
weights = [5, 8, 3, 7, 9, 2, 6, 1]
max_weight = 20
knapsack = Knapsack(names, values, weights, max_weight)
(chosen_items, value, weight) = knapsack.findBestCaseGreedy1()
print(chosen_items, value, weight)
```

3. Knapsack

파이썬코드1 실행 - value가 높은 물건부터 넣자

```
In [107]: runfile('D:/data/재정/주식소스/stock/result/
untitled3.py', wdir='D:/data/재정/주식소스/stock/result')
['B', 'E', 'C'] 73 20
```

3. Knapsack

의사코드2 - value/weight가 높은 물건부터 넣자

names = [‘A', ‘B', ‘C', ‘D', ‘E’], values = [10, 30, 20, 14, 23]
weights = [5, 8, 3, 7, 9], max_weight = 20

1. item 클래스 정의 : (name, value, weight)
2. items 리스트에 item 객체들을 추가
3. items 리스트를 item의 value/weight를 기준으로 내림차순 정렬

3. Knapsack

의사코드2 - value/weight가 높은 물건부터 넣자

```
cur_weight = 0
cur_value = 0
chosen_items = []
for item in sorted_items :
    if (cur_weight  + item. weight <= max_weight) :
        cur_value += item.value
        cur_weight += item.weight
        chosen_items.append(item)
return (chosen_items, cur_value, cur_weight)
```

3. Knapsack

파이썬 코드2 - value/weight가 높은 물건부터 넣자

```
class Item(object):
    def __init__(self, name, value, weight):
        self.name = name
        self.value = value
        self.weight = weight
```

3. Knapsack

파이썬 코드2 - value/weight가 높은 물건부터 넣자

```
class Knapsack(object):
    def __init__(self, names, values, weights, max_weight):     #아이템들을생성함
        self.items = []
        self.max_weight = max_weight
        for i in range(len(names)) :
            item = Item(names[i], values[i], weights[i])
            self.items.append(item)
```

3. Knapsack

파이썬 코드2 - value/weight가 높은 물건부터 넣자

```
class Knapsack(object):
    def findBestCaseGreedy2(self):
        self.items.sort(reverse=True, key=lambda x:(x.value/x.weight))
        value = 0
        weight = 0
        chosen_items = []
```

3. Knapsack

파이썬 코드2 - value/weight가 높은 물건부터 넣자

```
def findBestCaseGreedy2(self):
    for item in self.items :
        if (weight + item.weight <= self.max_weight) :
            chosen_items.append(item.name)
            weight += item.weight
            value += item.value

    return (chosen_items, value, weight)
```

3. Knapsack

파이썬 코드2 - value/weight가 높은 물건부터 넣자

```
names = ['A', 'B', 'C', 'D', 'E', 'F', 'G', 'H']
values = [10, 30, 20, 14, 23, 11, 15, 18]
weights = [5, 8, 3, 7, 9, 2, 6, 1]
max_weight = 20
knapsack = Knapsack(names, values, weights, max_weight)
(chosen_items, value, weight) = knapsack.findBestCaseGreedy1()
print(chosen_items, value, weight)
(chosen_items, value, weight) = knapsack.findBestCaseGreedy2()
print(chosen_items, value, weight)
```

3. Knapsack

파이썬 코드2 실행 - value/weight가 높은 물건부터 넣자

```
In [108]: runfile('D:/data/재정/주식소스/stock/result/
untitled3.py', wdir='D:/data/재정/주식소스/stock/result')
['B', 'E', 'C'] 73 20
['H', 'C', 'F', 'B', 'G'] 94 20
```

정리하기

- 탐욕(greedy) 알고리즘의 개념
- 탐욕 알고리즘을 활용한 거스름돈 문제 해결
- 탐욕 알고리즘을 활용한 knapsack 문제 해결

07강

다음 시간 안내▶▶▶

몬테카를로 시뮬레이션