Greedy Algorithm

greedy (탐욕적 기법)

- 최적해를 찾기 위한 방법 중 하나 (brute-force, monte-carlo, …)
- brute-force는 모든 경우를 탐색하고 이 중 최적의 해를 찾음.
  - 최대값 찾기 문제, knapsack, 최근접 쌍 등
  - 단점은 시간이 오래 걸릴 수 있음

greedy (탐욕적 기법)

- greedy는 모든 경우를 고려하는 것이 아니라, 어떤 결정을 해야 할 때
  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

문제 1

- 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

문제 1

- brute-force?
  - 580원을 만드는 모든 경우를 탐색하여 동전 개수를 최소화하는 솔루션 선택
  - 코딩도 쉽지 않고 시간도 오래 걸림
- greedy?

문제1

- brute-force?
  - 580원을 만드는 모든 경우를 탐색하여 동전 개수를 최소화하는 솔루션 선택
  - 코딩도 쉽지 않고 시간도 오래 걸림
- greedy?
  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

문제 1

- 파이썬 코드

  - 동전 종류는 액면가 기준으로 내림차순으로 정렬되어 전달됨

```
def min_coins_greedy(coins, left) :
    # coins 리스트에는 동전 종류가 액면가 기준으로 정렬되어 전달됨
    # coins = [500, 100, 10, 1]

    ...
```

문제 1

- 파이썬 코드

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

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

문제2

- 각각 무게가 $weight_i$, 가치가 val_i인 n개의 물건들을 배낭에 넣으려고 한다. 배낭에 넣을 수 있는 최대 무게는 W이다. 배낭에 넣은 물건들의 가치를 최대로 하는 물건의 조합을 구하시오.
  - *greedy1:*
  - *greedy2:*
  - 여러분의 방법?

문제2

- 각각 무게가 $weight_i$, 가치가 val_i인 n개의 물건들을 배낭에 넣으려고 한다. 배낭에 넣을 수 있는 최대 무게는 W이다. 배낭에 넣은 물건들의 가치를 최대로 하는 물건의 조합을 구하시오.
  - *greedy1*: 무게 상관 없이, 가치가 가장 높은 물건부터 넣자
  - *greedy2*: 단위 무게 당 가치가 가장 높은 물건부터 넣자
  - 여러분의 방법?
  -

문제2

- 파이썬 코드 – 기본 코드

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

문제2

- 파이썬 코드 – 기본 코드

```
class Knapsack(object):
    def __init__(self, names, values, weights, max_weight):      # 아이템들을 생성함
        self.items = []
        self.max_weight = max_weight
        self.max_value = 0
        self.opt_case = 0

        for i in range(len(names)) :
            item = Item(names[i], values[i], weights[i])
            self.items.append(item)
```

문제2

- 파이썬 코드 – 기본 코드

```
def printItems(self):
    for item in self.items:
        print(item.name, item.value, item.weight)
```

문제2

- 파이썬 코드 – 기본 코드

```
names = ['0', '1', '2', '3', '4']
values = [10, 30, 20, 14, 23]
weights = [5, 8, 3, 7, 9]
max_weight = 20

knapsack = Knapsack(names, values, weights, max_weight)
```

문제2

- 파이썬 코드 – 2단계
  - 기본 코드에서, 아이템들은 정렬되어 있지 않음
  - *greedy1*을 구현하기 위해서는 아이템들을 가치 기준으로 정렬하는 것이 필요
  - *greedy2*를 구현하기 위해서는 아이템들을 단위 무게 당 가치 기준으로 정렬하는 것이 필요
  - 기본 코드의 *items* 리스트를 각각의 기준으로 어떻게 정렬할까?

파이썬 리스트 정렬

- sort() 함수
  - 숫자는 오름차순으로 기본 정렬
  - 원본 리스트가 정렬된 형태로 수정됨

```
mylist = [7, 5, 3, 9]

mylist.sort()

print(mylist)          # [3, 5, 7, 9] 출력
```

파이썬 리스트 정렬

- sort() 함수
  - 내림 차순 정렬을 위해서는 인자를 전달
  - 원본 리스트가 정렬된 형태로 수정됨

```
mylist = [7, 5, 3, 9]

mylist.sort(reverse=True)

print(mylist)          # [9, 7, 5, 3] 출력
```

파이썬 리스트 정렬

- sort() 함수
  - 문제2에서는 item의 value를 기준으로 정렬해야 함.
  - 이를 위해 key를 lambda 함수 형태로 전달

```
self.items.sort(reverse=True, key=lambda x:x.value) # item의 value를 기준으로 정렬
self.printItems()
self.items.sort(reverse=True, key=lambda x:(x.value/x.weight)) # item의 단위 무게 당 value를 기준으로 정렬
self.printItems()
```

파이썬 리스트 정렬

- sorted() 함수

  - sort() 함수와 사용법은 유사함. 차이점은 원본은 수정되지 않고, 정렬된 별도의 리스트가 반환됨

```
mylist = [7, 5, 3, 9]

sorted_list = sorted(mylist)

print(mylist)          # [7, 5, 3, 9] 출력

print(sorted_list)     # [3, 5, 7, 9] 출력
```

문제2

- 파이썬 코드 – greedy1 함수 추가

```
def findBestCaseGreedy1(self):
        value = 0
        weight = 0
        best_chosen_items = []


        ...
        return (best_chosen_items, self.max_value)
```

문제2

- 파이썬 코드 – greedy2 함수 추가

```
def findBestCaseGreedy2(self):
        value = 0
        weight = 0
        best_chosen_items = []


        ...
        return (best_chosen_items, self.max_value)
```

문제2

- 파이썬 코드 – test 코드 수정

```
names = ['0', '1', '2', '3', '4', '5', '6', '7']
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, max_value) = knapsack.findBestCaseGreedy1()
print(chosen_items, max_value)

(chosen_items, max_value) = knapsack.findBestCaseGreedy2()
print(chosen_items, max_value)
```

문제2

- 추가 고려 사항
  - greedy는 최적의 해를 구할 수 있나? 여러 케이스를 만들어서 최적의 해를 찾는지 brute-force와 비교해 보자.
  - item들이 가루로 되어 있어서 일부분만을 배낭에 채울 수 있다면 greedy는 최적의 해를 찾을 수 있을까?
    - 금가루, 은가루, 동가루, …

Summary

- greedy
  - 최적해를 찾기 위한 방법 중 하나 (brute-force, monte-carlo, …)
  - 모든 경우를 고려하는 것이 아니라, 어떤 결정을 해야 할 때 그 순간에 최적이라고 생각되는 것을 선택함.
  - 결과적으로 global optimum이 아닌 local optimum (최적해의 근사값)을 도출하게 됨.
  - 장점은 brute-force보다 빠른 시간 안에 솔루션을 도출

Greedy 적용 문제

- 최소 비용 신장 트리
- 최단 경로