Python Programming

(26) 배낭 문제(Knapsack Problem)
+ 동적 계획법(Dynamic Programming)

배낭 문제

- **담아서 짊어질 수 있는 무게에 한계가 있는 배낭에 물건들을 담아 값의 총합이 최대가 되게 하려면 어떻게 하면 될까? (최대한 비싸게 채워 넣으려면?)**
- **물건 목록 예**

	값	무게
시계	175	10
액자	90	9
라디오	20	4
꽃병	50	2
책	10	1
컴퓨터	200	20

배낭 문제

- **물품의 클래스의 예**

```
class Item:
    def __init__(self, n, v, w):
        self.name = n
        self.value = v
        self.weight = w
    def getName(self):
        return self.name
    def getValue(self):
        return self.value
    def getWeight(self):
        return self.weight
    def __str__(self):
        return '<{0}: {1}, {2}>'.format(self.name,self.value,self.weight)
    def density(self):
        return self.value/self.weight
```

배낭 문제

- **분할 가능 배낭 문제와 짐을 쪼갤 수 없는 0/1 배낭 문제 등으로 나뉜다.**
- **배낭 문제의 수학적 정의**
  - 각 물품의 가치를 v_i, 무게를 w_i 라고 하고, 배낭에 넣을 수 있는 최대 무게를 W 라고 하면,

    $\sum w_i x_i \leq W$ 이라는 조건에서 $\sum v_i x_i$ 을 최대로 만드는 것이다.

  - 단, 0/1 배낭 문제의 경우 x_i 는 0 또는 1이고 분할가능 배낭 문제의 경우 0부터 1사이의 실수
  - 그 밖에 x_i 가 1보다 큰 수가 허용되는 배낭 문제도 있음.

Greedy Algorithm

- **한 가지 기준을 정하고 그 기준에 따라 제일 좋은 것부터 차례대로 담는 것이다.**
  - 기준에 따라 목록을 정렬(sort)하는 것이 필요하다.

- **배낭 문제에서 기준의 예**
  - 값(가치 또는 가격)
  - 무게의 역수
  - 무게 대비 가치: 분할 가능 배낭 문제에서는 항상 최적의 조합을 찾을 수 있게 해준다.

- **0/1 배낭 문제에서는 어떤 기준을 정해도 최적의 조합을 찾는다는 보장을 못한다.**

정렬(Sort)

- **파이썬에서는 내장 함수 sorted와 리스트 타입의 sort 메쏘드를 이용할 수 있다. (직접 만드는 것도 가능하다.)**
- **메쏘드 sort**
  - 파라미터 key: 함수 객체. 정렬 기준을 정함. 숫자일 경우 생략 가능.
  - 파라미터 reverse: True면 큰 것부터. 생략 또는 False면 작은 것부터
  - 사용 예: ['b', 'd' , 'c'].sort(key=str.lower) → ['b', 'c' , 'd']
- **함수 sorted**
  - 리스트 뿐만 아니라 iterable 타입은 다 가능한데, 결과는 새로 만든 리스트를 반환
  - 사용 예: dic = {'b':8, 'd':7, 'c':5}; newlist = sorted(dic, key=lambda k:dic[k]) → ['c' , 'd', 'b']

정렬(Sort)

- **Greedy algorithm을 배낭 문제에 적용할 때 sorted 함수 사용 예**
  - 물품들의 리스트를 itemList라고 하면,
  - 값이 기준인 경우

```
sortedList = sorted(itemList, key = Item.getValue, reverse = True)
```

  - 무게의 역수가 기준인 경우

```
sortedList = sorted(itemList, key = lambda i:i.getWeight())
```

  - 무게 대비 가치가 기준인 경우

```
sortedList = sorted(itemList, key = Item.density, reverse = True)
```

최적해(Optimal Solution)

- **가능한 해 중에서 가장 좋은 해를 말함**
- **Local optima(국부 최적 또는 극값)**
  - 특정 영역 안에서 최적, 곧 기준이 되는 값이 최대나 최소
  - 0/1 배낭 문제에 greedy algorithm을 적용하면 특정 조건 아래에서 최적인 조합을 찾는 셈이므로 local optimum을 얻게 된다.
- **Global optima**
  - 일반적인 의미의 최적해
  - 모든 영역에서 가장 좋은 해 (배낭 문제의 경우 가격 합이 최대)
  - 분할 가능 배낭 문제에 greedy algorithm을 적용하면 global optimum을 얻을 수 있다.

무차별 대입(Brute-Force)

- **단순 직진 방식**
- **묻지도 따지지도 않고 가능한 해들을 처음부터 끝까지 모두 대입해서 판단하는 방법**
- **0/1 배낭 문제의 경우**
  1. 가능한 모든 품목 조합을 나열(= 어떤 집합의 모든 부분집합을 구하는 작업과 같음)
  2. 제한 무게 총합을 넘기는 조합들을 배제
  3. 남는 조합들의 가격 합을 각각 계산하여 최대값이 되는 조합을 찾음

  ➢ Computational complexity = $O(n \cdot 2^n)$

  ❖ 최적해를 보장하지만, 계산 시간을 많이 소모해야 한다.

무차별 대입(Brute-Force)

- **모든 부분 집합을 리스트로 생성하는 함수**

```
def genPowerset(L):
    powerset = []
    for i in range(2**len(L)):
        binStr = getBinaryRep(i, len(L))
        subset = []
        for j in range(len(L)):
            if binStr[j] == '1':
                subset += [L[j]]
        powerset += [subset]
    return powerset
```

```
def getBinaryRep(n, numDigits):
    result = ''
    while n >0:
        result = str(n%2) + result
        n = n//2
    if len(result) > numDigits:
        raise ValueError('not enough digits')
    for i in range(numDigits - len(result)):
        result = '0' + result
    return result
```

참고

- **모든 부분 집합을 리스트로 생성하는 함수**

```
def genPowerset(L):
    powerset = []
    for i in range(2**len(L)):
        binStr = getBinaryRep(i, len(L))
        subset = []
        for j in range(len(L)):
            if binStr[j] == '1':
                subset += [L[j]]
        powerset += [subset]
    return powerset
```

```
def getBinaryRep(n, numDigits):
    result = bin(n)
    result = result[2:]
    for i in range(numDigits - len(result)):
        result = '0' + result
    return result
```

무차별 대입(Brute-Force)

- **배낭 문제에 맞춰 개량한 경우**

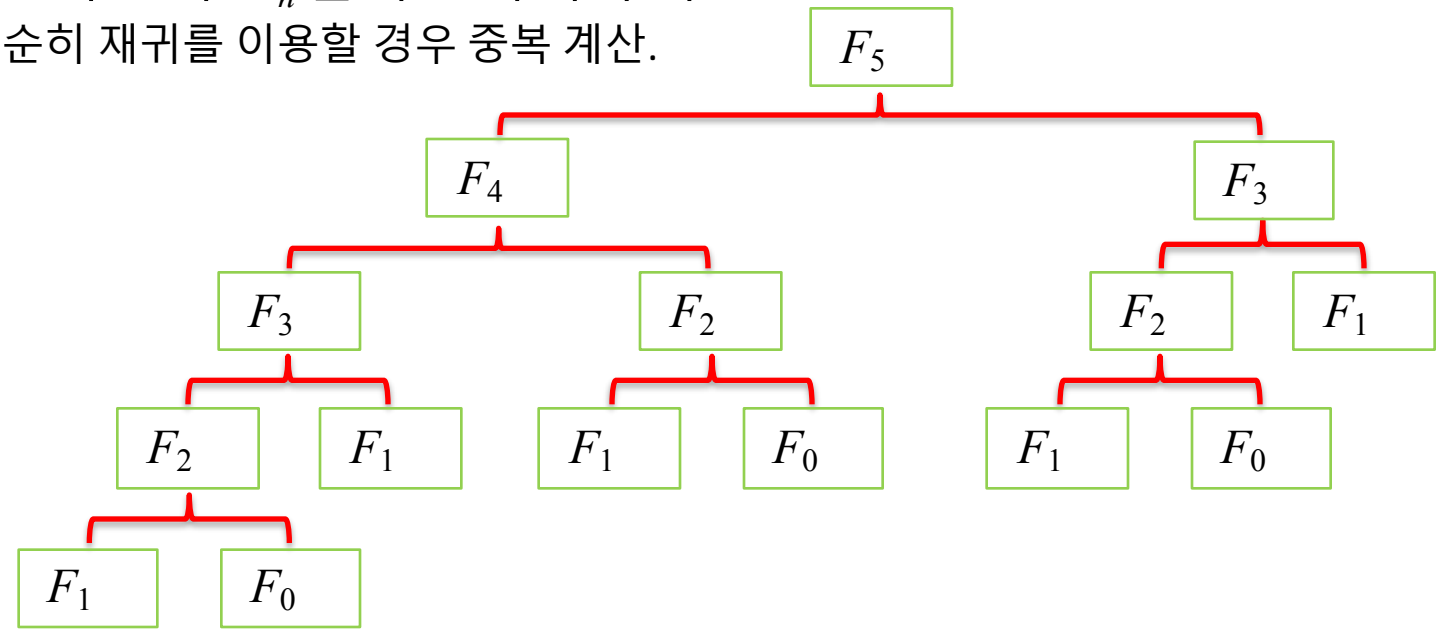
```
def genPowerset(L,  maxWeight):
    powerset = []
    for i in range(2**len(L)):
        binStr = getBinaryRep(i, len(L))
        subset = []
        totalWeight = 0
        for j in range(len(L)):
            if binStr[j] == '1':
                subset += [L[j]]
                totalWeight += L[j].getWeight()
            if totalWeight > maxWeight:  break
        if totalWeight <= maxWeight:
            powerset += [subset]
    return powerset
```

동적 계획법(Dynamic Programming)

- **Bellman이 고안한 문제 풀이법**
- **복잡한 문제를 쉬운 문제들로 나눠서 재귀적으로 푸는데, 부분 문제들끼리 해를 공유해 중복 계산을 막으면서 본 문제의 해를 구하는 것이다.**
- **두 가지 특성을 가진 문제에 적용할 수 있다.**
  - 최적 부분 구조(optimal substructure): 부분 문제들의 최적해를 조합하면 본 문제의 최적해를 구할 수 있는 구조
  - 부분 문제 반복(overlapping subproblems): 최적해를 얻기 위해서는, 같은 부분 문제를 반복해서 풀어야 하는 것, 또는 부분 문제의 해가 반복되어 사용되어야 하는 것

동적 계획법(Dynamic Programming)

- **피보나치 수열 문제**
  - 재귀(recursion)를 이용하는 경우 동적 계획법을 적용할 수 있다.
  - 부분 문제들: $F_n = F_{n-1} + F_{n-2}, \quad n = 2,3,4, \ldots, N$
  - 최적 부분 구조: F_n 들을 조합하면 F_N 을 얻을 수 있다.
  - 부분 문제 반복: F_n 들이 반복해서 나옴
    - 단순히 재귀를 이용할 경우 중복 계산.



동적 계획법(Dynamic Programming)

- **피보나치 수열 문제**
  - 동적계획법을 적용한 코드
    - 메모이제이션(memorization)을 통해 해를 공유

```
def dp_fib(n, memo = {}):
    """ Dynamic programming version Fibonacci """
    if n == 0 or n == 1:
        return n
    result = memo.get(n)
    if result == None:
        result = dp_fib(n-1,memo) + dp_fib(n-2,memo)
        memo[n] = result
    return result
```

동적 계획법(Dynamic Programming)

- **0/1 배낭 문제**
  - 본 문제: 각 물품의 가치를 v_i, 무게를 w_i 라고 하고, 배낭에 넣을 수 있는 최대 무게를 W 라고 하면,

    $\sum w_i x_i \le W$ 이라는 조건에서 $\sum v_i x_i$ 를 최대로 만드는 것이다.
  - 부분 문제들: j-1번째까지 x_i 들이 각각 0인지 1인지 결정한 상태에서 x_j가 0인지 1인지 결정하는 것. (j-1번째까지 x_i 들의 조합이 다르면 다른 문제)
  - 최적 부분 구조: 본 문제의 최적해는 부분 문제들의 최적해들 중 하나가 된다. (주의: 본 문제의 최적해까지 최적해들만 거친다는 뜻이 아님. 최적해들만 거치는 것은 greedy algorithm이므로 본 문제의 최적해에 이르지 못할 수 있음.)
  - 부분 문제 반복: j-1번째까지 x_i 들을 결정한 상태에서 무게의 합이 같은 문제들은 사실상 같은 문제

동적 계획법(Dynamic Programming)

- **0/1 배낭 문제**
  - 몇 번째까지 결정했는지 알려 주는 숫자 *j*-1(또는 *N*-*j*+1)과 무게 합(또는 추가가능한 무게 - 제한 무게에서 무게 합을 뺀 것)에 대한 메모이제이션을 하면 중복 문제 풀이를 방지할 수 있다.
  - Computational complexity: pseudo-polynomial
    - Polynomial ≈ Pseudo-polynomial < Exponential
    - 물품 목록이 길어질수록 무게 합이 같은 조합이 나올 확률이 커지기 때문

# of items	# of selected items	# of calls
4	4	31
8	6	337
16	9	1493
32	12	3650
64	19	8707
128	27	18306
256	40	36675

Guttag 책 표 13.8

동적 계획법에 의한 배낭 문제 풀이

```
def fastMaxVal(toConsider, avail, memo = {}):
    """ toConsider: 남은 물품 목록, avail: 더 넣을 수 있는 무게 """
    remLen = len(toConsider) # N – j + 1
    # 메모에 이미 존재한다면
    if (remLen, avail) in memo:
        return memo[(remLen, avail)]
    # 더 이상 넣을 수 없는 경우
    elif remLen == 0 or avail == 0
        result = (0, ())
    # j번째 물품을 넣으면 무게 제한을 초과하는 경우
    elif toConsider[0].getWeight() > avail:
        result = fastMaxVal(toConsider[1:], avail, memo)
```

동적 계획법에 의한 배낭 문제 풀이

```
else:
    nextItem = toConsider[0]  # j번째 물품
    # 넣는 경우
    withVal, withToTake = \
            fastMaxVal(toConsider[1:], avail – nextItem.getWeight(), memo)
    withVal += nextItem.getValue()
    # 넣지 않는 경우
    withoutVal, withoutToTake = fastMaxVal(toConsider[1:], avail, memo)
    # 어느 쪽이 더 좋은지 판단
    if withVal > withoutVal:
        result = (withVal, withToTake + (nextItem,))
    else:
        result = (withoutVal, withoutToTake)
memo[(remLen, avail)] = result
return result
```

아주 긴 물품 리스트로 테스트하기

```
import random as rnd

def buildManyItems(numItems, maxVal, maxWeight):
    items = []
    for i in range(numItems):
        items += [Item(str(i),rnd.randint(1,maxVal),rnd.randint(1,maxWeight))]
    return items

def bigTest(numItems):
    items = buildManyItems(numItems, 30, 30)
    val, taken = fastMaxVal(items, 100)
    print('Items Taken')
    for item in taken:
        print(item)
    print('Total value = ', val)
```

참고: 아주 긴 물품 리스트로 테스트하기

```
import time

def bigTest(numItems):
    items = buildManyItems(numItems, 30, 30)
    stime = time.process_time()
    val, taken = fastMaxVal(items, 100)
    etime = time.process_time()
    print('Items Taken')
    for item in taken:
        print(item)
    print('Total value = ', val, ' Elapsed time = ', etime - stime)
```