Optimization Problems,

John Guttag

MIT Department of Electrical Engineering and Computer Science

오늘 강의 관련 읽을거리

- Chapter 13

탐욕 알고리즘의 장단점

- 시행하기 쉬움
- 효율적인 계산

- 하지만 항상 최적의 해를 주는 것은 아님
  - 근사가 얼마나 좋은지조차 알지 못함

Question 1

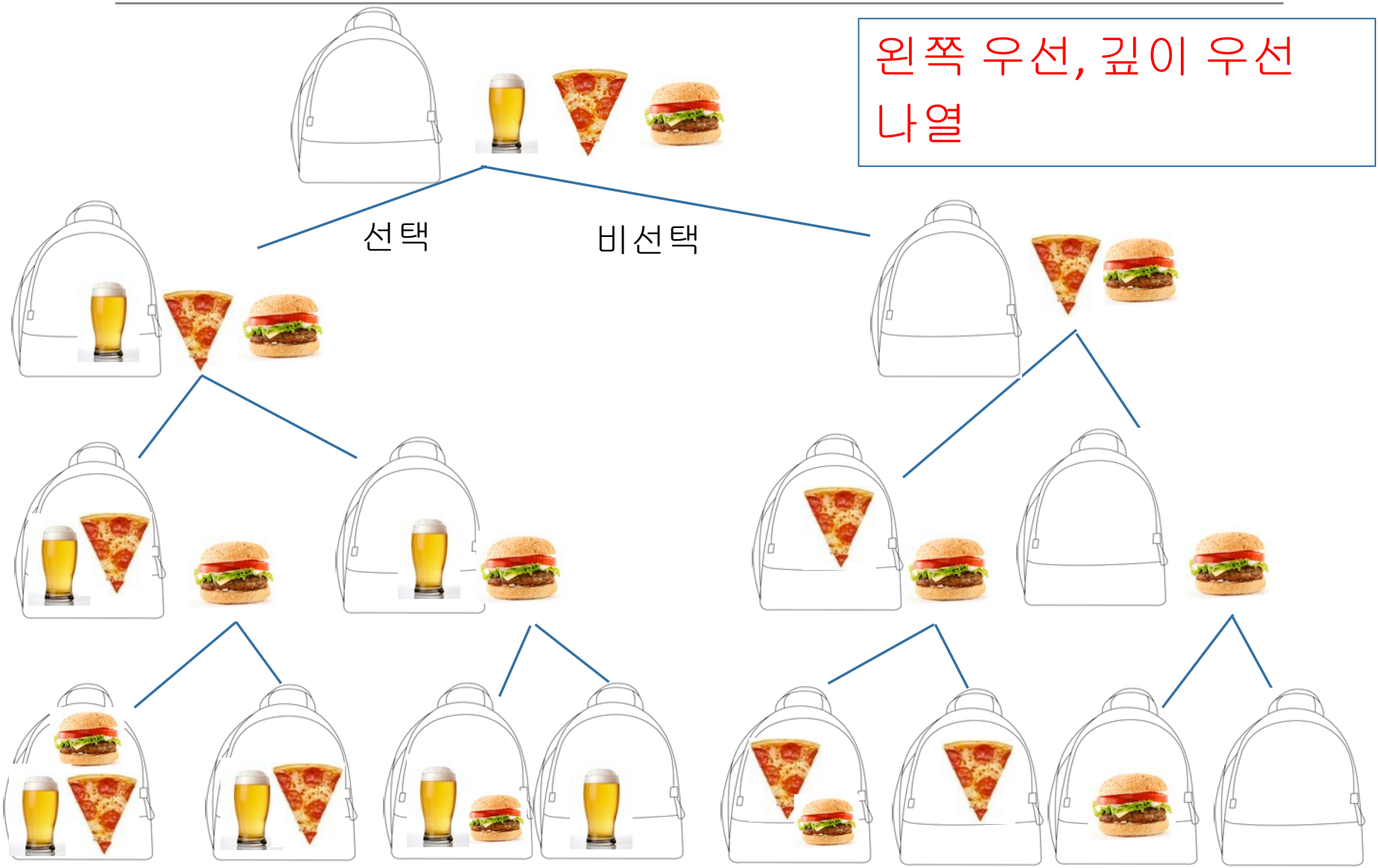
무차별 대입 알고리즘

- 1. 물건의 모든 가능한 조합을 나열함
- 2. 총합이 허용된 무게를 초과하는 조합을 모두 제거함
- 3. 남은 조합 중 가장 큰 값을 가지는 조합을 아무거나 하나 택함

탐색 트리 구현

- 트리는 뿌리에서 시작해서 위에서 아래로 만들어짐
- 첫 번째 원소는 고려해야하는 물건 중 선택
  - 냅색에 그 물건을 위한 공간이 있으면, 노드는 물건을 선택한 결과를 반영하여 만들어짐. 관례적으로, 왼쪽 자식으로 그림
  - 선택하지 않는 결과도 탐색함. 이는 오른쪽 자식임
- 이 과정을 단말 노드가 아닌 자식에게 재귀적으로 적용
- 최종적으로, 제한조건을 만족하는 노드 중 가장 큰 값을 가진 노드를 선택

탐색 트리 경우의 수 나열



왼쪽 우선, 깊이 우선 나열

계산 복잡도

- 시간은 만들어진 노드 수와 연관
- 레벨의 수는 선택할 수 있는 물건의 개수
- 레벨 i에서 노드의 수는 2^i
- 따라서, n개의 물건이 있을 때 노드의 개수는
  - $\sum i=0\uparrow i=n 2\uparrow i$
  - I.e., O($2\uparrow n+1$)
- 분명한 최적화 : 제한조건을 어기는 트리는 탐색하지 않음 (e.g., 너무 많은 칼로리)
  - 복잡도는 불변
- 무차별 대입법은 쓸모없는 것을 뜻할까?
  - 한 번 봅시다

결정 트리 구현 헤더

```
def maxVal(toConsider, avail):
    """Assumes toConsider a list of items,          avail a
                    weight
       Returns a tuple of the total value of a
            solution to 0/1 knapsack problem and
            the items of that solution"""
```

`toConsider`. 아직 고려하지 않은 트리의 상단 노드에 있는 물건 리스트(재귀 호출에서 이전 호출과 연관)

`avail`. 남은 여유 공간의 양

maxVal 함수의 본체 (주석 생략)

```
if toConsider == [] or avail == 0:        result = (0, ())
    elif toConsider[0].getUnits() > avail:
        result = maxVal(toConsider[1:], avail)        else:
        nextItem = toConsider[0]
        withVal, withToTake = maxVal(toConsider[1:],
                                     avail - nextItem.getUnits())  withVal +=
        nextItem.getValue()
        withoutVal, withoutToTake = maxVal(toConsider[1:], avail)
    if withVal > withoutVal:
            result = (withVal, withToTake + (nextItem,))          else:
            result = (withoutVal, withoutToTake)    return result
```

실제로 탐색 트리를 만드는 것은 아님

지역변수 result는 지금까지 탐색한 것 중 최고의 해를 기록

첫 번째 강의에서 다룬 예제

- 750 칼로리 예산 하에서, 다음 메뉴 중 최적의 음식 세트를 선택

Food	wine	beer	pizza	burger	fries	coke	apple	donut
Value	89	90	30	50	90	79	90	10
calories	123	154	258	354	365	150	95	195

탐색 트리는 잘 작동

- 더 좋은 답을 줌
- 빨리 끝남
- 하지만 2^8 은 큰 수가 아님
  - 선택할 수 있는 메뉴가 훨씬 많을 때 어떤 일이 일어나는지 확인해봐야 함

더 큰 예 실행 코드

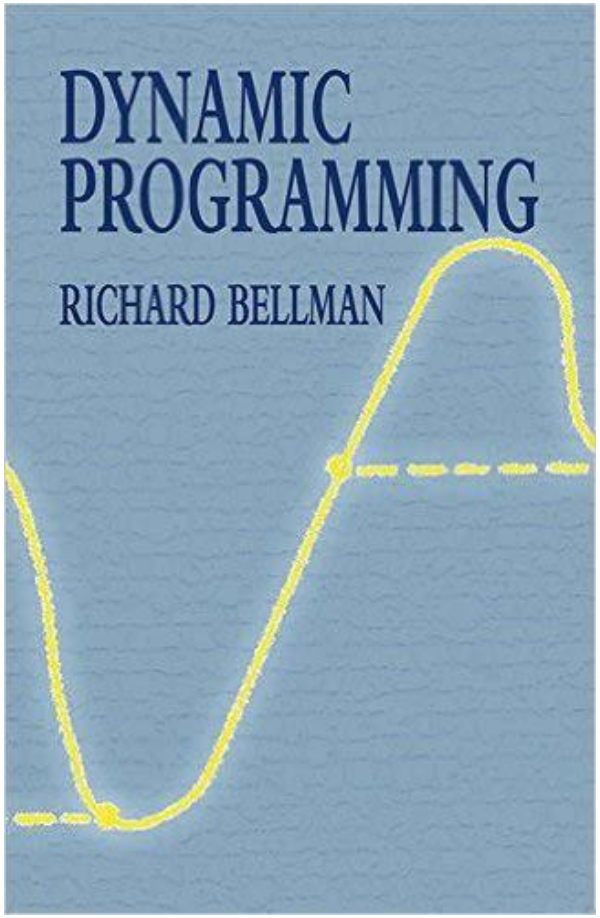
```
import random

def buildLargeMenu(numItems, maxVal, maxCost):  items = []
    for i in range(numItems):    items.append(Food(str(i),
                                 random.randint(1, maxVal),
                                 random.randint(1, maxCost)))
    return items


for numItems in (5,10,15,20,25,30,35,40,45,50,55,60):
    items = buildLargeMenu(numItems, 90, 250)
    testMaxVal(items, 750, False)
```

가망이 없는가?

- 이론적으론, 그렇습니다
- 실생활에선, 아닙니다
- 동적 프로그래밍이 해결책



동적 프로그래밍이란?

때때로 이름은 그냥 이름입니다

"1950년대는 수학 연구에 좋은 해가 아니었습니다... 저는 윌슨과 공군을 제가 실제로 수학을 하고 있다는 사실로부터 보호하기 위해 무언가 해야 한다고 느꼈습니다...
'어떤 제목, 어떤 이름을 선택해야 할까?... 부정적인 의미로 동적이라는 단어를 사용하는 것은 불가능하다. 부정적 의미를 줄 수 있는 조합을 생각해보자.' 하지만 불가능했습니다. 따라서 동적 프로그래밍이 좋은 이름이라고 생각했습니다.
이는 하원의원조차도 반대할 수 없는 것이었습니다. 따라서 저는 그것을 제 활동을 위한 우산으로 사용했습니다."

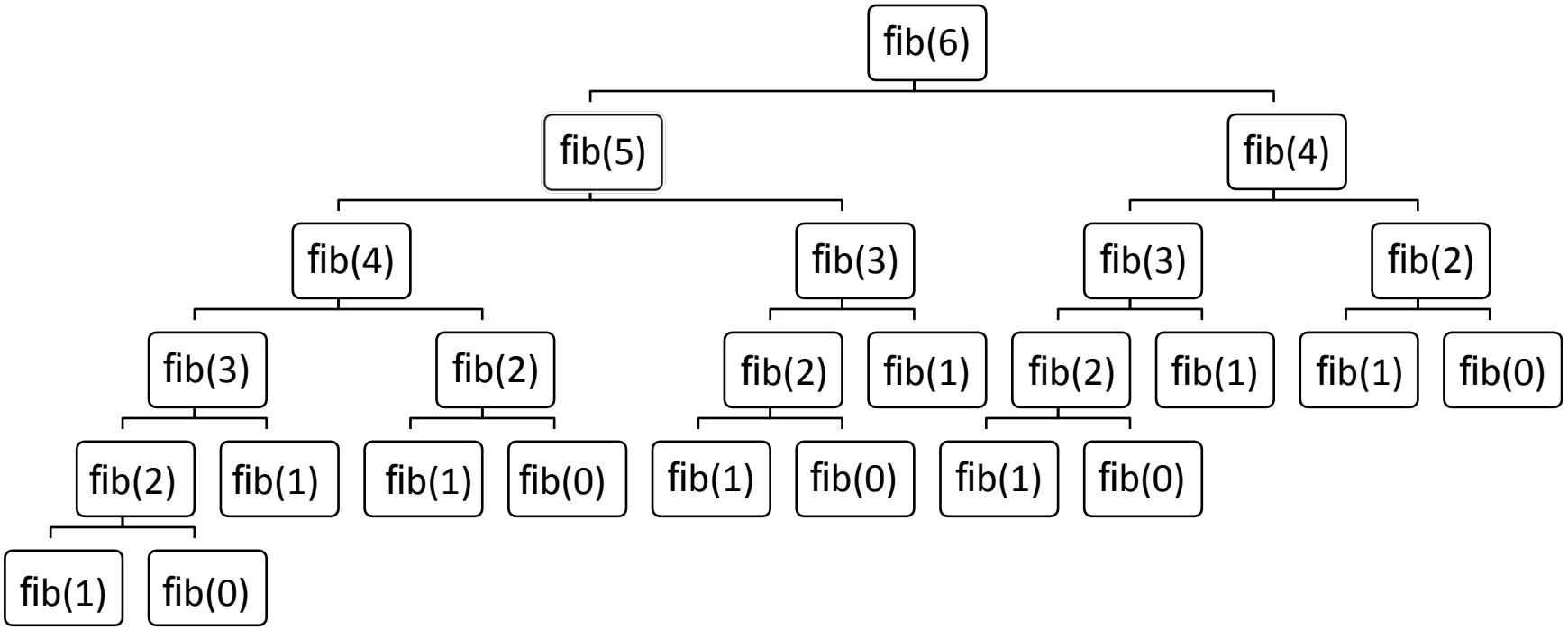
-- Richard Bellman

피보나치의 재귀적 구현

```
def fib(n):
    if n == 0 or n == 1:  return 1
    else:
        return fib(n - 1) + fib(n - 2)
```

fib(120) = 8,670,007,398,507,948,658,051,921

Fibonacci(6) = 13 을 위한 재귀 호출 트리



반복 작업은 분명히 좋지 않은 생각

- 시간을 공간과 맞바꿈
- 테이블을 만들어 지금까지 한 것을 기록
  - **fib(x)**를 계산하기 전에, **fib(x)**가 테이블에 이미 저장되어있는지 확인
    - 있다면, 값을 가져옴
    - 없다면, 계산하고 테이블에 저장
  - 메모이제이션이라고 불림

메모를 사용해 피보나치 계산

```
def fastFib(n, memo = {}):
        """Assumes n is an int >= 0, memo used only by
              recursive calls
            Returns Fibonacci of n"""        if n == 0 or n == 1:
              return 1   try:
              return memo[n]  except KeyError:
              result = fastFib(n-1, memo) +\  fastFib(n-2, memo)
              memo[n] = result  return result
```

어디서 쓸 수 있을까?

- 최적 부분 구조 : 전역 최적 해를 지역 부분 문제에서 최적 해를 결합함으로써 얻을 수 있는 문제
  - For x > 1, fib(x) = fib(x - 1) + fib(x – 2)

- 중복 부분 문제 : 최적 해를 구할 때 같은 문제를 여러 번 풀어야 하는 문제
  - fib(x)를 한 번 계산하거나 여러 번 계산하거나

0/1 냅색 문제에서는 어떨까?

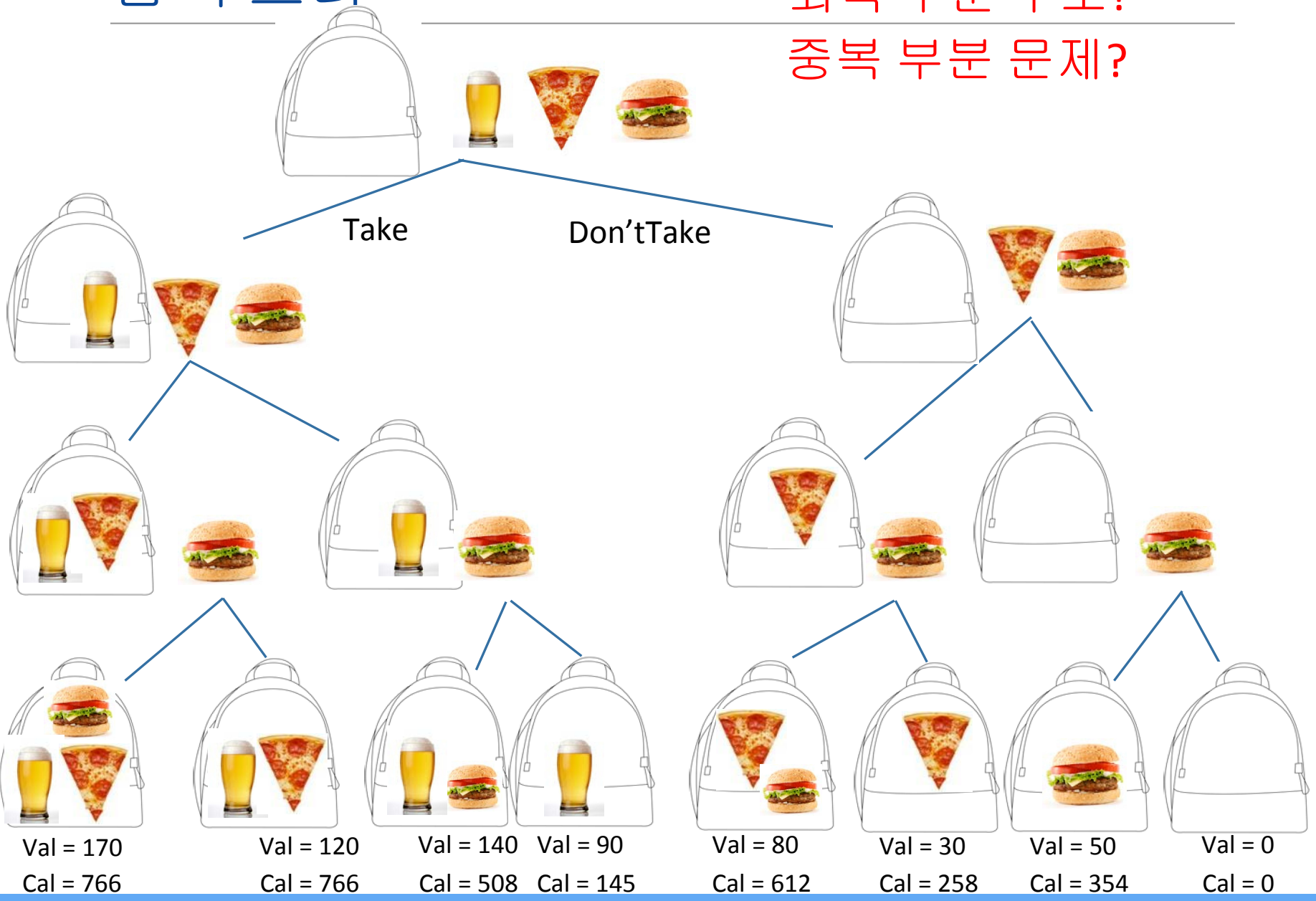
- 이 조건들이 성립할까?

Questions 2 and 3

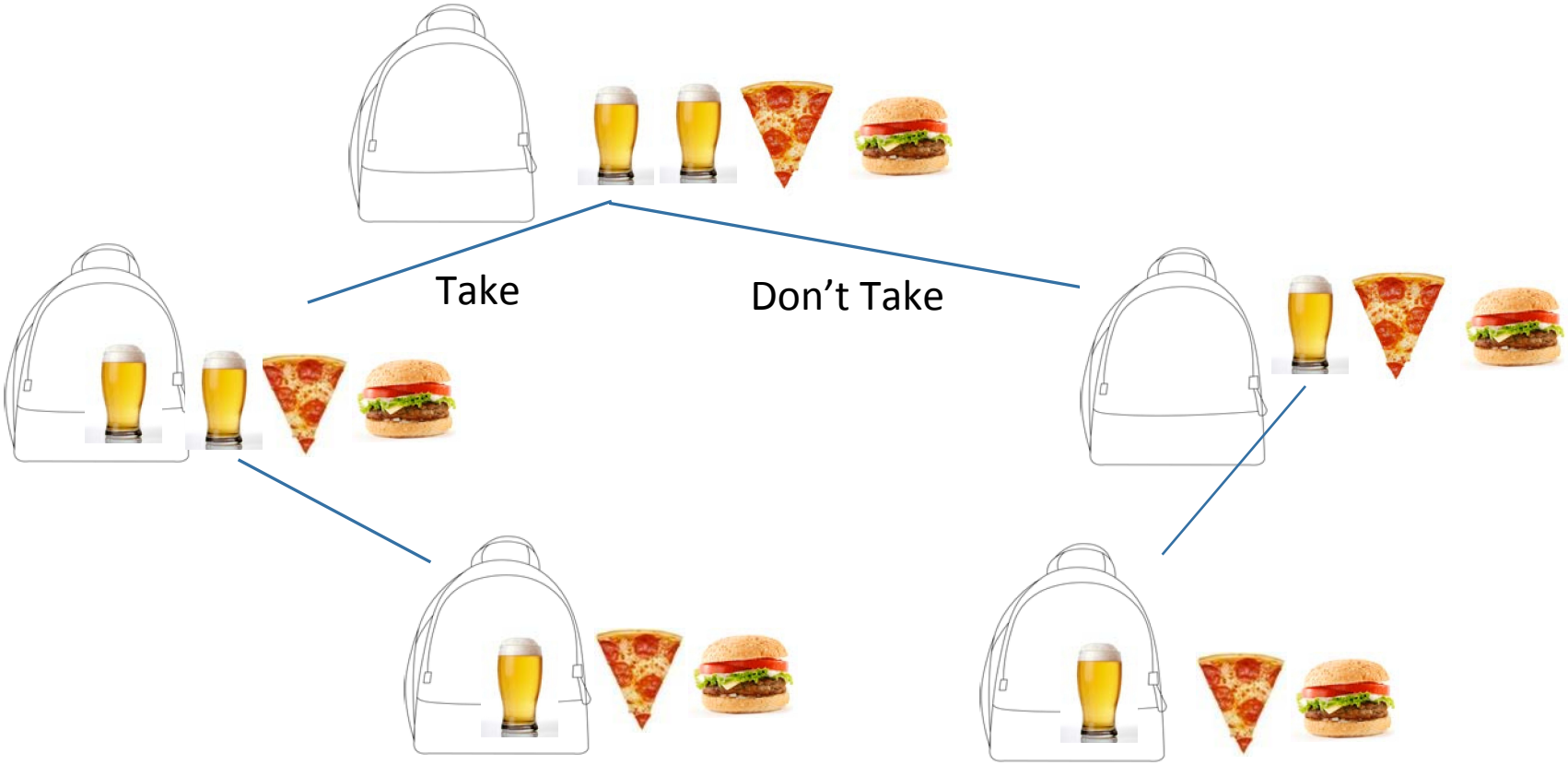
탐색 트리

최적 부분 구조?

중복 부분 문제?



다른 메뉴

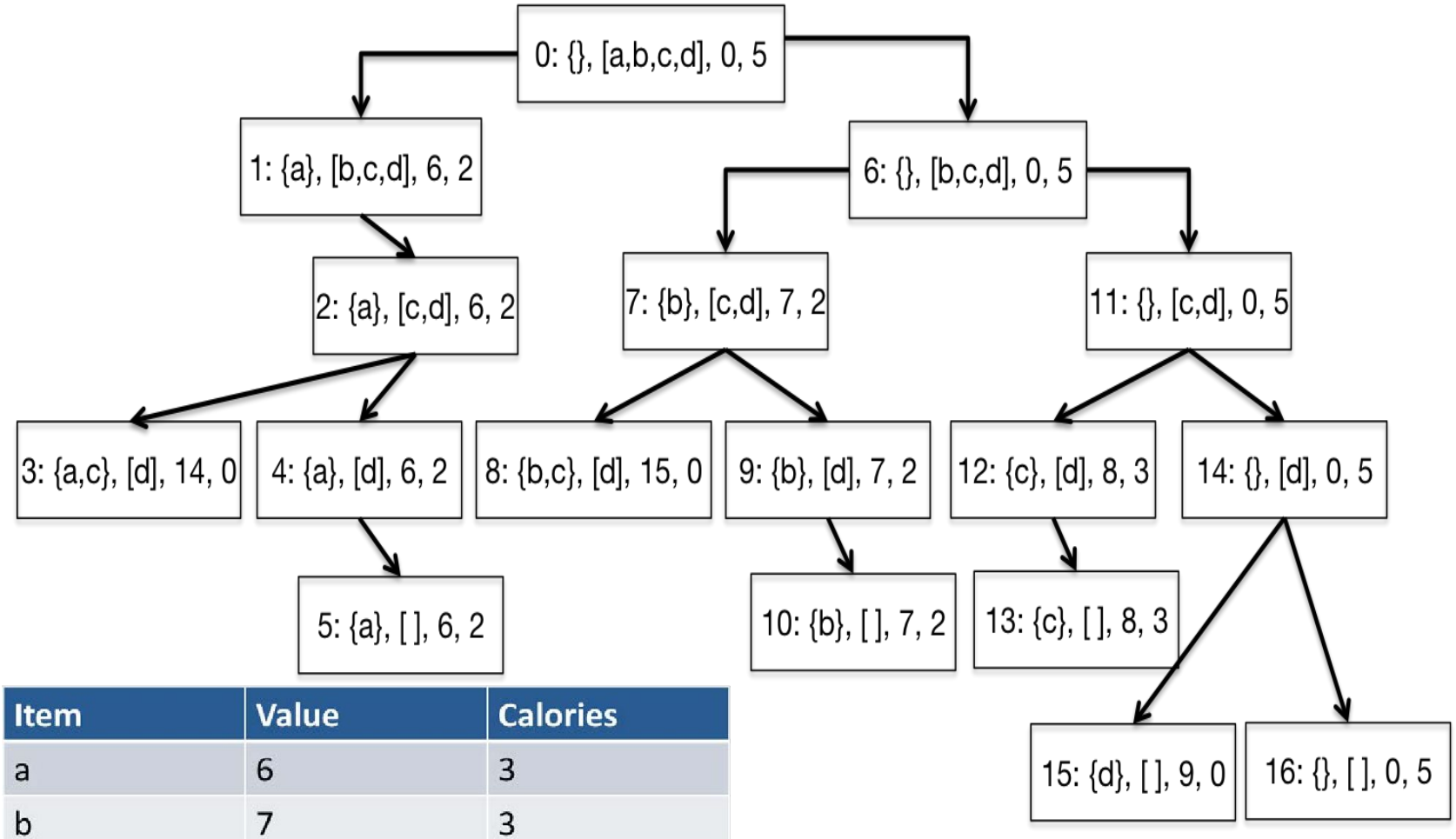


중복 물건을 가질 필요도 없음

Item	Value	Calories
a	6	3
b	7	3
c	8	2
d	9	5

탐색 트리

- 각 노드 = <선택한 것, 남은 것, 값, 남은 칼로리>

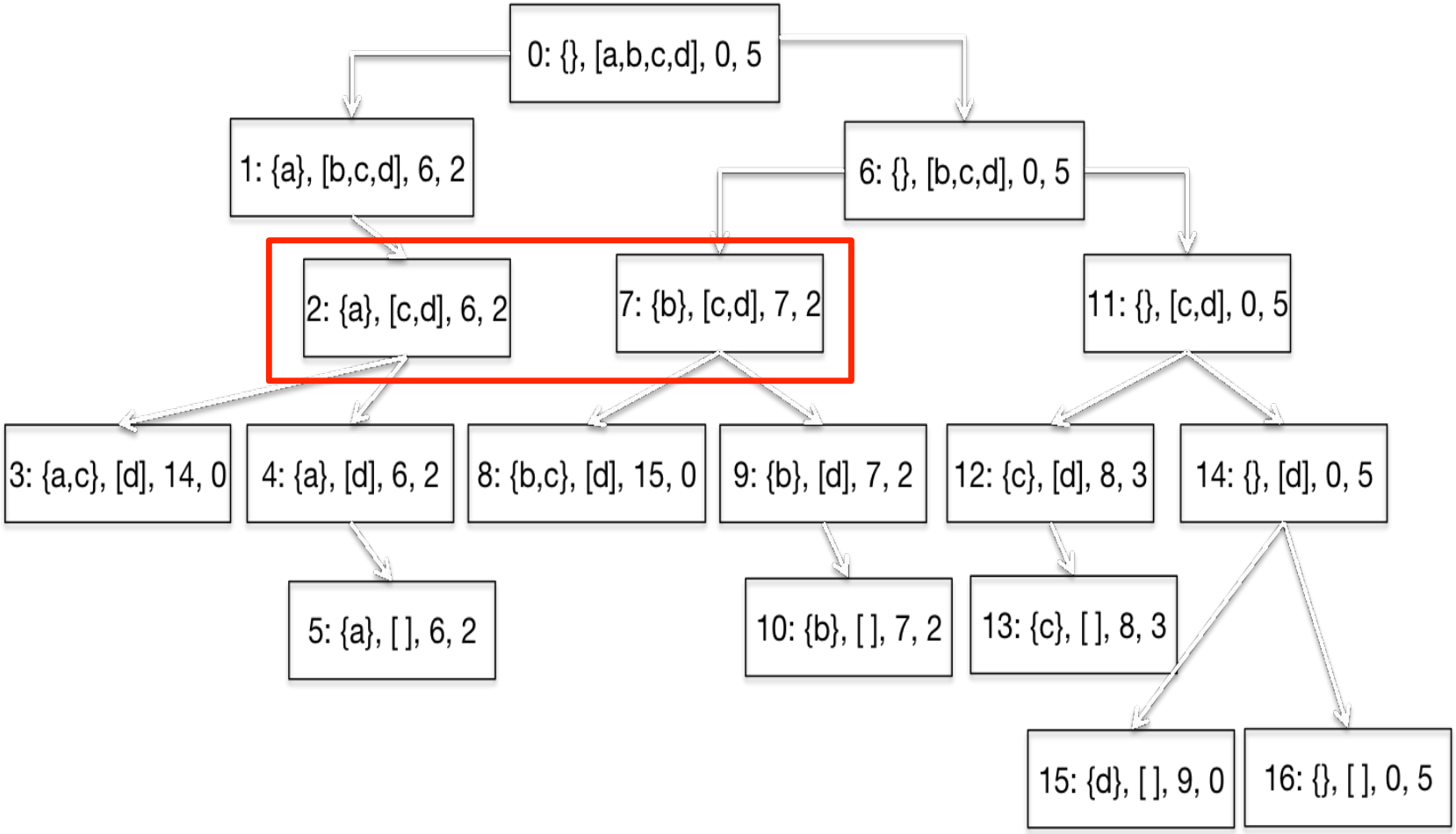


Item	Value	Calories
a	6	3
b	7	3
c	8	2
d	9	5

각 노드에서 어떤 문제를 푸는가?

- 남은 무게가 주어졌을 때, 남은 물건 중 값을 최대화하는 것을 선택
- 이전에 선택한 물건, 혹은 그 물건들의 값은 현재 노드에서의 선택과 무관

중복 부분 문제


0: {}, [a,b,c,d], 0, 5
1: {a}, [b,c,d], 6, 2
6: {}, [b,c,d], 0, 5
2: {a}, [c,d], 6, 2
7: {b}, [c,d], 7, 2
11: {}, [c,d], 0, 5
3: {a,c}, [d], 14, 0
4: {a}, [d], 6, 2
8: {b,c}, [d], 15, 0
9: {b}, [d], 7, 2
12: {c}, [d], 8, 3
14: {}, [d], 0, 5
5: {a}, [ ], 6, 2
10: {b}, [ ], 7, 2
13: {c}, [ ], 8, 3
15: {d}, [ ], 9, 0
16: {}, [ ], 0, 5

메모를 쓰기 위한 maxVal 수정

- 세 번째 인수로 memo를 추가
  - def fastMaxVal(toConsider, avail, memo = {}):
- 메모의 핵심은 튜플
  - (고려해야 하는 남은 물건, 가능한 무게)
  - 고려해야 하는 남은 물건은 다음과 같이 표현

    len(toConsider)
- 함수의 몸체가 첫 번째로 하는 일은 주어진 가능한 무게에서 물건의 최적의 선택이 이미 메모에 있는지 확인
- 함수의 몸체가 마지막으로 하는 일은 메모를 갱신

퍼포먼스

len(items)	2**len(items)	Number of calls
2	4	7
4	16	25
8	256	427
16	65,536	5,191
32	4,294,967,296	22,701
64	18,446,744,073,709,551,616	42,569
128	Big	83,319
256	Really Big	176,614
512	Ridiculously big	351,230
1024	Absolutely huge	703,802

왜 이렇게 되나요?

- 문제는 지수적
- 우리가 우주의 법칙을 바꾼 걸까요?
- 동적 프로그래밍이 기적일까요?
- 아닙니다, 하지만 계산 복잡도는 미묘합니다
- fastMaxVal 의 실행 시간은 서로 다른 쌍 <toConsider, avail> 에 의해 결정됨
  - toConsider 이 가능한 값은 len(items)에 의해 제한됨
  - avail 이 가능한 값은 다소 특징짓기 어려움
    - 서로 다른 무게의 합에 의해 제한됨
  - 읽기 숙제에서 자세한 내용을 다룸

강의 1 - 2의 요약

- 실제로 중요한 많은 문제들은 최적화 문제로 쓸 수 있음
- 탐욕 알고리즘은 대부분 적당한 답을 줌(최적은 아닐 수 있음)
- 최적 해를 찾는 것은 거의 지수적으로 어려움
- 하지만 동적 프로그래밍은 최적화 문제의 부분 문제에서 좋은 퍼포먼스를 줌- 최적 부분 구조와 중복 부분 문제를 가지는 경우
  - 해는 항상 옳음
  - 옳은 순환 하에서 빠름

“롤-오버” 최적화 문제

점수 = ((60 – (a+b+c+d+e))*F + a*ps1 + b*ps2 + c*ps3 + d*ps4 + e*ps5

목표:

F, ps1, ps2, ps3, ps4, ps5 가 주어졌을 때,
점수를 최대화 하는 a, b, c, d, e 를 찾으시오

제한 조건:

a, b, c, d, e = 10 또는 0
a + b + c + d + e ≥ 20

6.0002 Introduction to Computational Thinking and Data Science
Fall 2016