

# Introduction and Optimization Problems

---

John Guttag

MIT Department of Electrical Engineering and  
Computer Science

# 6.0002 선수 조건

---

- 파이썬을 활용한 객체 지향 프로그래밍 경험
  - 파이썬 3.5 사용 경험 선호
- 계산 복잡도 개념 숙지
- 간단한 알고리즘 숙지
- **6.0001 수강**

## Question 1

# 몇 가지 수업 전반적인 것들

## ■ 문제 세트

- 프로그래밍 문제의 구성 목표
  - 프로그래밍 능력 향상
  - 관념적 주제 습득에 도움

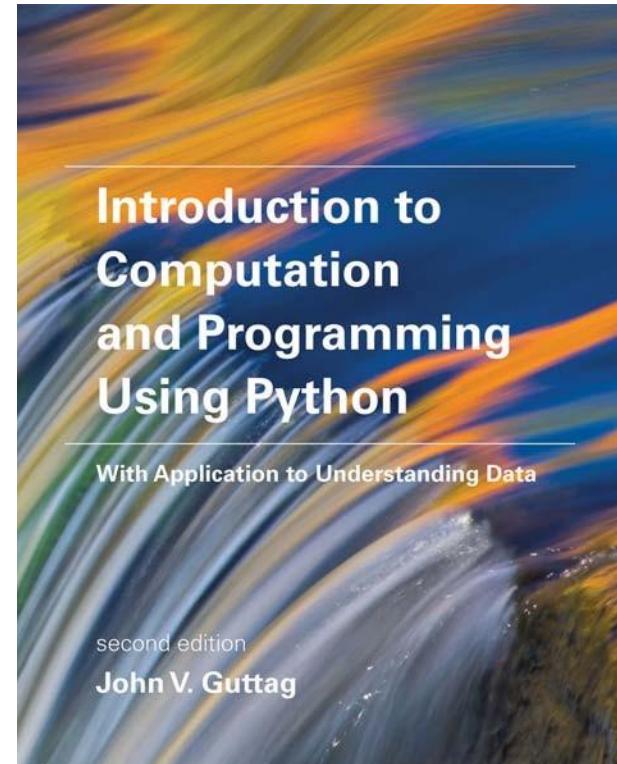
## ■ 간이 연습문제

- 간단한 프로그래밍 개념 습득을 위한 간단한 프로그래밍 문제들로 구성

## ■ 교재 읽기 과제

- 강의와 문제 세트에서 다루는 주제에 대한 추가적인 내용

## ■ 시험 : 위의 모든 내용을 포함



# 6.0001과의 비교?

---

- 프로그래밍 과제는 보다 쉬움
  - 프로그래밍 보다는 풀 수 있는 문제 중심
- 강의 내용은 보다 추상적
- 강의 속도는 보다 빠름
- 프로그래밍을 배우는 것은 적게, 대신 데이터 과학에 발을 들일 수 있게

# 프로그래밍 능력 연마

---

- 파이썬에 대한 추가적인 내용 약간
- 소프트웨어 공학
- 패키지 사용
- “카네기홀에 어떻게 가나요?”

# 계산 모델

- 우리가 사는 세계를 이해하는데 도움이 되는 계산
- 현재까지 일어난 일을 이해하거나 미래를 예측하는데 도움을 주는 실험적 장치



Images © sources unknown. All rights reserved. This content is excluded from our Creative Commons license. For more information, see <https://ocw.mit.edu/help/faq-fair-use>.

- 최적화 모델
- 통계적 모델
- 시뮬레이션 모델

# 오늘의 강의 관련 읽을거리

---

- Section 12.1
- Section 5.4 (람다 함수)

# 계산 모델

- 우리가 사는 세계를 이해하는데 도움이 되는 계산
- 현재까지 일어난 일을 이해하거나 미래를 예측하는데 도움을 주는 실험적 장치



Images © sources unknown. All rights reserved. This content is excluded from our Creative Commons license. For more information, see <https://ocw.mit.edu/help/faq-fair-use>.

## ■ 최적화 모델

- 통계적 모델
- 시뮬레이션 모델

# 최적화 모델이란?

- 목적 함수란 최대화하거나 최소화해야 하는 것;
  - 뉴욕에서 보스턴까지 여행하는 데 걸리는 시간을 최소화
- 제한 조건이란(없을 수도 있음) 반드시 지켜야 하는 것;
  - 100달러 이상 쓸 수 없음
  - 보스턴에 5시 전까지 도착해야 함



Images © sources unknown. All rights reserved. This content is excluded from our Creative Commons license. For more information, see <https://ocw.mit.edu/help/faq-fair-use>.

# 냅색 문제

---



Images © sources unknown. All rights reserved. This content is excluded from our Creative Commons license. For more information, see <https://ocw.mit.edu/help/faq-fair-use>.

# 냅색 문제

- 힘의 한계가 있어서, 들 수 있는 냅색 무게의 한계가 있다고 가정
- 들 수 있는 한 많은 물건을 들고 싶음
- 어떤 물건을 가져가고 어떤 물건을 남길 것인지 어떻게 결정할 것인가?
- 두 가지 종류
  - 0/1 냅색 문제
  - 연속 혹은 분수 냅색 문제



Images © sources unknown. All rights reserved. This content is excluded from our Creative Commons license. For more information, see <https://ocw.mit.edu/help/faq-fair-use>.

# 제가 가장 싫어하는 냅색 문제

---



Images © sources unknown. All rights reserved. This content is excluded from our Creative Commons license. For more information, see <https://ocw.mit.edu/help/faq-fair-use>.

# 0/1 냅색 문제, 공식화

---

- 각 물건들은 쌍으로 표현, <값, 무게>
- 냅색은 총 무게  $w$ 까지만 수용 가능
- 길이  $n$ 인 벡터  $L$ 은 가능한 물건들의 집합을 표현. 이 벡터의 각 원소들은 물건을 의미
- 길이  $n$ 인 벡터  $V$ 는 물건을 가져가는지 안 가져가는지를 나타냄. 만약  $V[i] = 1$ 이면, 물건  $I[i]$ 는 가져가는 것이고,  $V[i] = 0$ 이면, 물건  $I[i]$ 는 가져가지 않는 것을 의미

# 0/1 냅색 문제, 공식화

---

Find a  $V$  that maximizes

$$\sum_{i=0}^{n-1} V[i] * I[i].value$$

subject to the constraint that

$$\sum_{i=0}^{n-1} V[i] * I[i].weight \leq w$$

# 무차별 대입 알고리즘

---

- 1. 물건의 모든 가능한 조합을 나열함. 다시 말하면, 물건 집합의 모든 부분집합을 만듦. 이 집합을 **멱집합**이라고 함
- 2. 총합이 허용된 무게를 초과하는 조합을 모두 제거함
- 3. 남은 조합 중 가장 큰 값을 가지는 조합을 아무거나 하나 택함

# 그다지 실용적이진 않음

---

- 역집합이 얼마나 큰가?
- 상기
  - 길이  $n$ 인 벡터  $v$ 는 물건을 가져가는지 안 가져가는지를 나타냄. 만약  $v[i] = 1$ 이면, 물건  $I[i]$ 는 가져가는 것이고,  $v[i] = 0$ 이면, 물건  $I[i]$ 는 가져가지 않는 것을 의미
- $v$ 가 가질 수 있는 서로 다른 경우의 수가 얼마나 있는가?
  - $n$  비트로 표현될 수 있는 서로 다른 이진수의 개수
- 예를 들면, 100개의 물건들을 고를 수 있다면, 역집합의 크기는 얼마인가?
  - 1,267,650,600,228,229,401,496,703,205,376

## Question 2

# 그냥 바보 같은 짓을 한 걸까요?

---

- 유감스럽게도, 정답이 아님
- 0/1 낱색 문제는 본질적으로 지수적임
- 하지만 절망하지 말길

# 실용적인 대안으로써 탐욕 알고리즘

---

- 냅색이 가득차지 않은 동안에는  
    “가장 좋은” 가능한 물건을 냅색에 넣음
- 그러나 “가장 좋은”의 의미는 무엇일까?
  - 가장 가치있는
  - 최소 비용
  - 가치/단위의 비율이 가장 높은 것

# 예시

---

- 여러분이 식사를 하기 위해 식탁에 막 앉은 상황
- 여러 종류의 음식들을 얼마나 좋아하는지 모두 알고 있다고 가정. 예를 들어, 도넛을 사과보다 더 좋아함
- 하지만 넘지 말아야 할 칼로리 총량이 정해져 있음. 예를 들어, 750 칼로리 이상은 먹어서는 안 됨
- 어떤 걸 먹어야 할 지 결정하는 것도 일종의 낱색 문제

# 메뉴

---

Food	wine	beer	pizza	burger	fries	coke	apple	donut
Value	89	90	30	50	90	79	90	f0
calories	123	154	258	354	365	150	95	195

- 어떤 걸 주문할지 결정할 때 사용할 수 있는 프로그램을 살펴보자

# Food 클래스

---

```
class Food(object):
    def __init__(self, n, v, w): self.name = n
        self.value = v
        self.calories = w

    def getValue(self):      return self.value

    def getCost(self):       return self.calories

    def density(self):
        return self.getValue()/self.getCost()

    def __str__(self):
        return self.name + ': <' + str(self.value) \
            + ', ' + str(self.calories) + '>'
```

# 음식 메뉴 만들기

---

```
def buildMenu(names, values, calories):
    """names, values, calories lists of same length.
       name a list of strings
       values and calories lists of numbers
       returns list of Foods"""
    menu = []
    for i in range(len(values)):
        menu.append(Food(names[i], values[i],
                          calories[i]))
    return menu
```

# 유연한 탐욕 알고리즘의 시행

---

```
def greedy(items, maxCost, keyFunction): """Assumes items a list,  
maxCost >= 0,  
    keyFunction maps elements of items to numbers"""  
    itemsCopy = sorted(items, key = keyFunction, ←  
                      reverse = True)  
    result = []  
    totalValue, totalCost = 0.0, 0.0  
  
    for i in range(len(itemsCopy)): ←  
        if (totalCost+itemsCopy[i].getCost()) <= maxCost:  
            result.append(itemsCopy[i])  
            totalCost += itemsCopy[i].getCost() ← totalValue +=  
            itemsCopy[i].getValue()  
  
    return (result, totalValue)
```

# 알고리즘 효율성

---

```
def greedy(items, maxCost, keyFunction):           itemsCopy =  
    → sorted(items, key = keyFunction,  
            reverse = True)  
    result = []  
    totalValue, totalCost = 0.0, 0.0  
  
    for i in range(len(itemsCopy)): ←————  
        if (totalCost+itemsCopy[i].getCost()) <= maxCost:  
            result.append(itemsCopy[i])  
            totalCost += itemsCopy[i].getCost()  totalValue +=  
            itemsCopy[i].getValue()  
  
    return (result, totalValue)
```

## Question 3

# greedy 함수 사용

---

```
def testGreedy(items, constraint, keyFunction):
    taken, val = greedy(items, constraint, keyFunction)
    print('Total value of items taken =', val)
    for item in taken:      print('  ', item)
```

# greedy 함수 사용

---

```
def testGreedy(maxUnits):
    print('Use greedy by value to allocate', maxUnits, 'calories')
    testGreedy(foods, maxUnits, Food.getValue)      print('\nUse
greedy by cost to allocate', maxUnits,
    'calories')
testGreedy(foods, maxUnits,
           lambda x: 1/Food.getCost(x)) ←
print('\nUse greedy by density to allocate', maxUnits, 'calories')
testGreedy(foods, maxUnits, Food.density)
```

testGreedy(800)

?

# 람다 표현식

---

- 람다는 익명의 함수를 만드는데 사용
  - `lambda <idf, id2, ... idn>: <표현식>`
  - n개의 인자의 함수를 반환
- 여기서처럼, 매우 다루기 쉬움
- 매우 복잡한 람다 표현식을 쓰는 것도 가능
- 하지만 어떤 경우는 사용하지 말 것 –`def`를 대신 사용

# greedy 함수 사용

---

```
def testGreedy(foods, maxUnits):
    print('Use greedy by value to allocate', maxUnits,      'calories')
    testGreedy(foods, maxUnits, Food.getValue)            print("\nUse
    greedy by cost to allocate', maxUnits,
    'calories')
testGreedy(foods, maxUnits,
           lambda x: 1/Food.getCost(x))
print('\nUse greedy by density to allocate', maxUnits,
      'calories')
testGreedy(foods, maxUnits, Food.density)

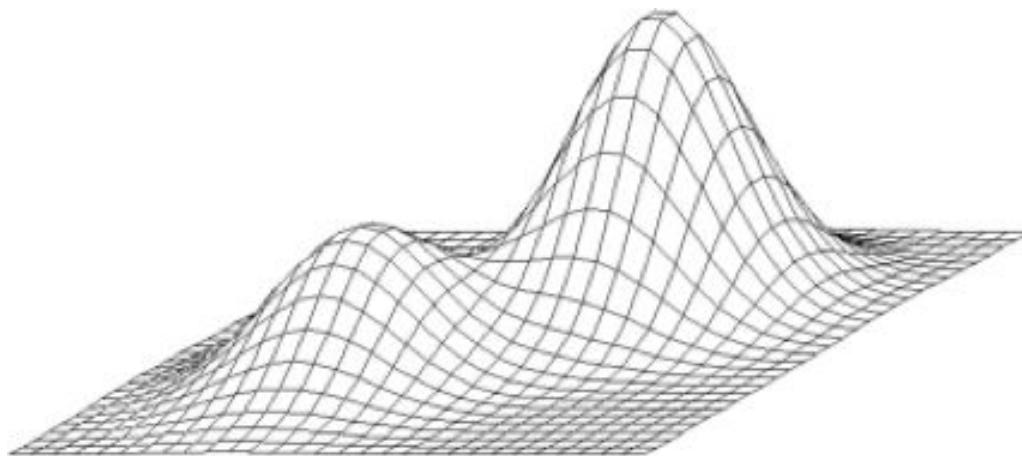
names = ['wine', 'beer', 'pizza', 'burger', 'fries',
         'cola', 'apple', 'donut', 'cake']      values =
[89,90,95,100,90,79,50,10]
calories = [123,154,258,354,365,150,95,195]
foods = buildMenu(names, values, calories) testGreedy(foods, 750)
```

Run code

# 왜 답이 다를까?

---

- 지역적으로 “최적의” 선택의 수열이 항상 전역적으로 최적의 해를 주는 것은 아님



- 밀도 기준 **greedy** 함수가 항상 최고의 선택일까?
  - `testGreedy(foods, 1000)` 를 시도해보자

# 탐욕 알고리즘의 장단점

---

- 시행하기 쉬움
  - 효율적인 계산
- 
- 하지만 항상 최적의 해를 주는 것은 아님
    - 근사가 얼마나 좋은지 조차 알지 못함
  - 다음 강의에서는 진짜 최적의 해를 찾는 방법을 알아볼 것

MIT OpenCourseWare

<https://ocw.mit.edu>

6.0002 Introduction to Computational Thinking and Data Science

Fall 2016

For information about citing these materials or our Terms of Use, visit: <https://ocw.mit.edu/terms>.