python
Programming

파이썬 프로그래밍 활용

2020-2
이 남 연

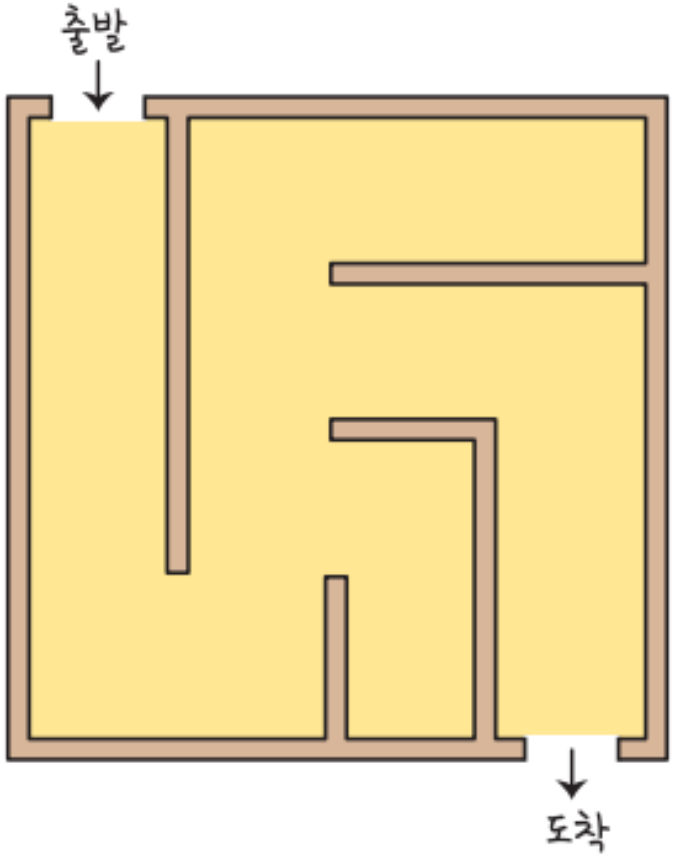
응용 문제

Chapter 8

문제 16: 미로 찾기 알고리즘

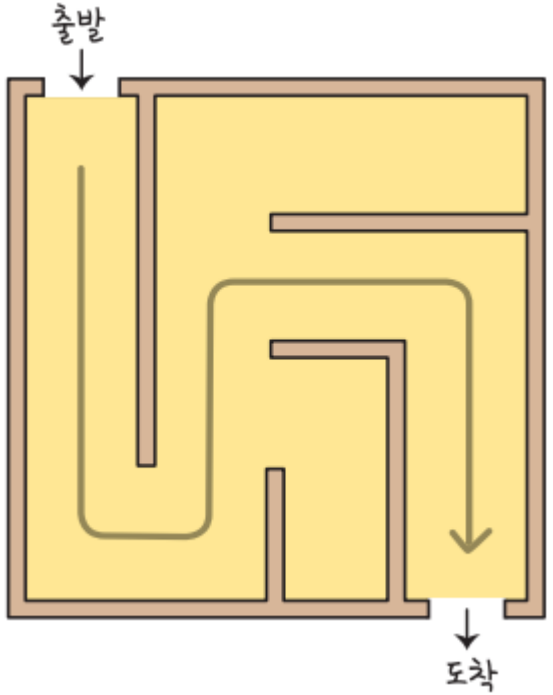
미로 찾기 알고리즘

- 문제 : 다음 그림과 같이 미로의 형태와 출발점과 도착점이 주어졌을 때 출발점에서 도착점까지 가기 위한 최단 경로를 찾는 알고리즘을 만들어 보세요.



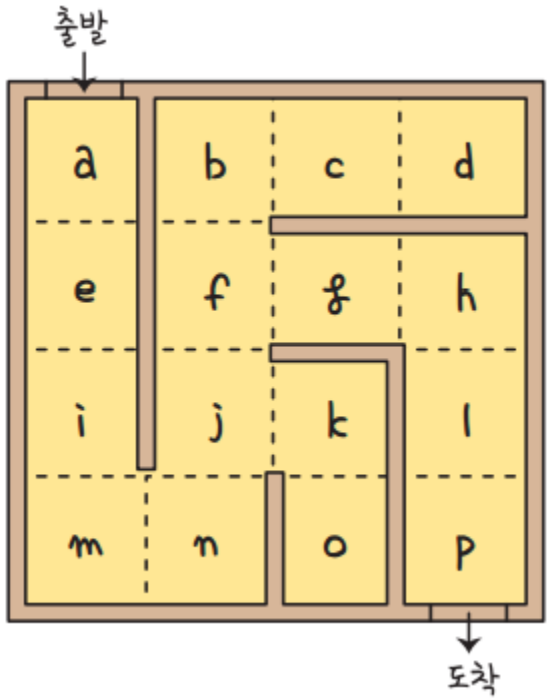
문제 분석과 모델링

- 주어진 미로는 굉장히 간단한 문제
- 이 문제를 컴퓨터에게 풀어 보라고 하려면 어떻게 해야 할까?
  - 사람에게는 쉽지만 컴퓨터에게 이 문제를 이해하고 풀게 하긴 어려움
  - 이때 필요한 것이 바로 '모델링(모형화)'
  - 모델링이란 주어진 현실의 문제를 정형화하거나 단순화하여 수학이나 컴퓨터 프로그램으로 쉽게 설명할 수 있도록 다시 표현하는 것
  - 모델링은 자연이나 사회현상을 사람의 언어로 표현한 문제를 컴퓨터가 쉽게 이해할 수 있도록 수학식이나 프로그래밍 언어로 번역하는 절차



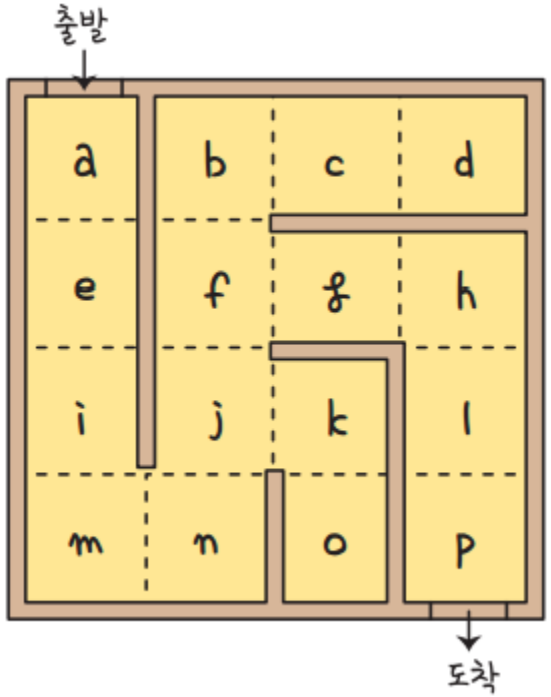
문제 분석과 모델링

- 그래프를 이용해 미로 찾기 문제를 단계별로 모델링해 보자
  - 일단 미로를 풀려면 미로 안의 공간을 정형화해야 함
  - 그림 16-1의 퍼즐은 4×4로 구성된 간단한 미로
  - 이동 가능한 위치를 각각의 구역으로 나누고, 구역마다 알파벳으로 이름을 붙이면 아래 그림과 같음



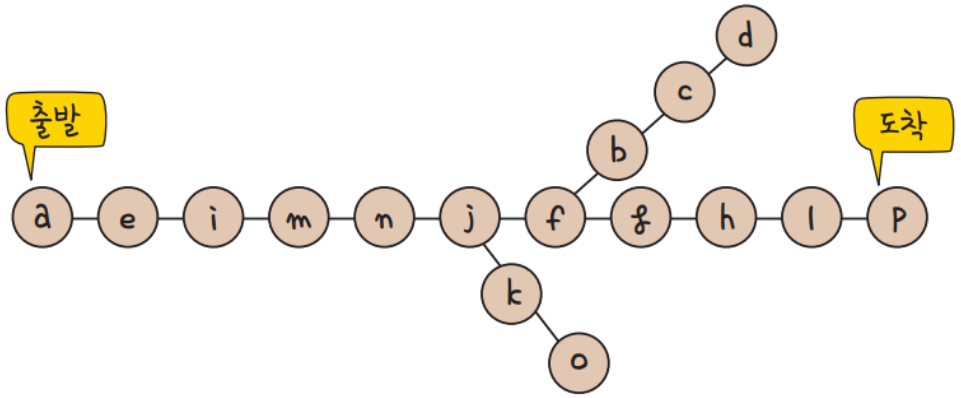
문제 분석과 모델링

- 모델(모형)을 이용해서 미로 찾기 문제와 정답을 다시 적어 보면 다음과 같이 표현할 수 있음
  - 출발점 a에서 시작하여 벽으로 막히지 않은 위치로 차례로 이동하여 도착점 p에 이르는 가장 짧은 경로를 구하고, 그 과정에서 지나간 위치의 이름을 출력
  - 정답: aeimnjfghlp



문제 분석과 모델링

- 최종 결과를 얻으려면 각 위치 사이의 관계를 컴퓨터에게 알려 줘야 하고 실제로 미로를 푸는 알고리즘도 만들어야 함
  - 이 문제는 문제 15에서 풀었던 그래프 탐색 문제와 같음
  - 위치 열여섯 개를 각각 꼭짓점으로 만들고, 각 위치에서 벽으로 막히지 않아 이동할 수 있는 이웃한 위치를 모두 선으로 연결



문제 분석과 모델링

- 그래프를 딕셔너리로 바꾸면 다음과 같음
  - 처음에 그림으로 주어졌던 미로 찾기 문제가 모델링을 통해 그래프가 되고, 그 그래프가 파이썬 언어가 이해할 수 있는 딕셔너리로 표현됨
  - 이제 그래프 탐색 알고리즘을 적용하여 출발점부터 도착점까지 탐색하는 프로그램을 만들어 보자

```
# 미로 정보
# 미로의 각 위치에 알파벳으로 이름을 지정
# 각 위치에서 한 번에 이동할 수 있는 모든 위치를 선으로 연결하여 그래프로 표현
maze = {
    'a': ['e'],
    'b': ['c', 'f'],
    'c': ['b', 'd'],
    'd': ['c'],
    'e': ['a', 'i'],
    'f': ['b', 'g', 'j'],
    'g': ['f', 'h'],
    'h': ['g', 'l'],
     'i': ['e', 'm'],
     'j': ['f', 'k', 'n'],
     'k': ['j', 'o'],
     'l': ['h', 'p'],
     'm': ['i', 'n'],
     'n': ['m', 'j'],
     'o': ['k'],
     'p': ['l']
}
```

미로찾기 알고리즘

미로찾기 알고리즘 코드

```python
# 미로 찾기 프로그램(그래프 탐색)
# 입력: 미로 정보 g, 출발점 start, 도착점 end
# 출력: 미로를 나가기 위한 이동 경로는 문자열, 나갈 수 없는 미로면 물음표("?")

def solve_maze(g, start, end):
    qu = []       # 기억 장소 1: 앞으로 처리해야 할 이동 경로를 큐에 저장
    done = set()  # 기억 장소 2: 이미 큐에 추가한 꼭짓점들을 집합에 기록(중복 방지)

    qu.append(start)  # 출발점을 큐에 넣고 시작
    done.add(start)   # 집합에도 추가

    while qu:           # 큐에 처리할 경로가 남아 있으면
        p = qu.pop(0)   # 큐에서 처리 대상을 꺼냄
        v = p[-1]     # 큐에 저장된 이동 경로의 마지막 문자가 현재 처리해야 할 꼭짓점
        if v == end:    # 처리해야 할 꼭짓점이 도착점이면(목적지 도착!)
            return p    # 지금까지의 전체 이동 경로를 돌려주고 종료
        for x in g[v]:  # 대상 꼭짓점에 연결된 꼭짓점들 중에
            if x not in done:     # 아직 큐에 추가된 적이 없는 꼭짓점을
                qu.append(p + x)  # 이동 경로에 새 꼭짓점으로 추가하여 큐에 저장하고
                done.add(x)       # 집합에도 추가함

    # 탐색을 마칠 때까지 도착점이 나오지 않으면 나갈 수 없는 미로임
    return "?"

```

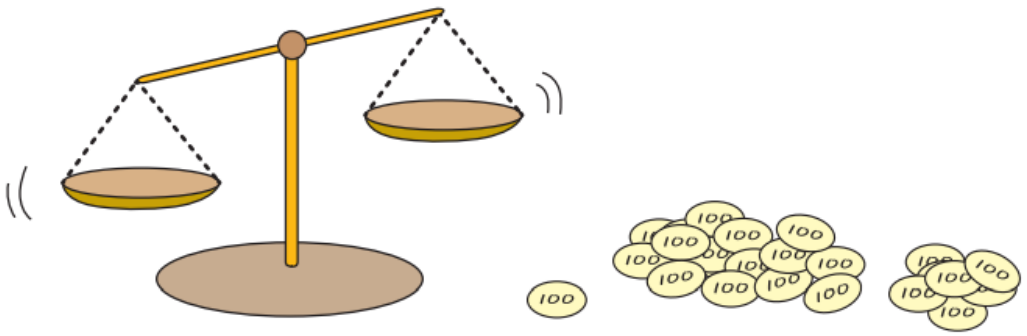
```python
# 미로 정보
# 미로의 각 위치에 알파벳으로 이름을 지정
# 각 위치에서 한 번에 이동할 수 있는 모든 위치를 선으로 연결하여 그래프로 표현
maze = {
    'a': ['e'],
    'b': ['c', 'f'],
    'c': ['b', 'd'],
    'd': ['c'],
    'e': ['a', 'i'],
    'f': ['b', 'g', 'j'],
    'g': ['f', 'h'],
    'h': ['g', 'l'],
    'i': ['e', 'm'],
    'j': ['f', 'k', 'n'],
    'k': ['j', 'o'],
    'l': ['h', 'p'],
    'm': ['i', 'n'],
    'n': ['m', 'j'],
    'o': ['k'],
    'p': ['l']
}
print(solve_maze(maze, 'a', 'p'))

```

```
aeimnjfghlp
```

문제 17 : 가짜 동전 찾기 알고리즘

가짜 동전 찾기

- 문제 : 겉보기에는 똑같은 동전이 n개 있습니다. 이 중에서 한 개는 싸고 가벼운 재료로 만들어진 '가짜 동전'입니다. 좌우 무게를 비교할 수 있는 양팔 저울을 이용해서 다른 동전보다 가벼운 가짜 동전을 찾아내는 알고리즘을 만들어 보세요.



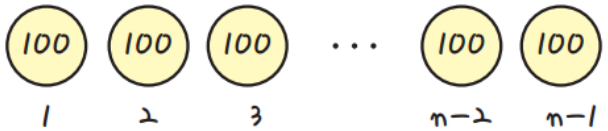
문제 분석과 모델링

문제 정리

- 동전 n개 중에는 무게가 적게 나가는 가짜 동전이 한 개 섞여 있음
- 무게를 숫자로 보여 주는 디지털 저울이 있다면 동전 무게를 차례대로 하나씩 재서 가짜 동전을 간단히 판별할 수 있음
- 하지만 우리가 사용할 저울은 양팔에 물건을 올리면 어느 쪽이 더 무거운지와 가벼운지만 알려 주는 '양팔 저울'

문제를 좀 더 정형화해 보자

- 동전 개수는 n개이므로 왼쪽에서 오른쪽으로 동전을 일렬로 나열한 다음 맨 왼쪽을 0번으로 하여 차례대로 번호를 붙여 보자
- 가장 오른쪽에 있는 동전은 n-1번이 됨



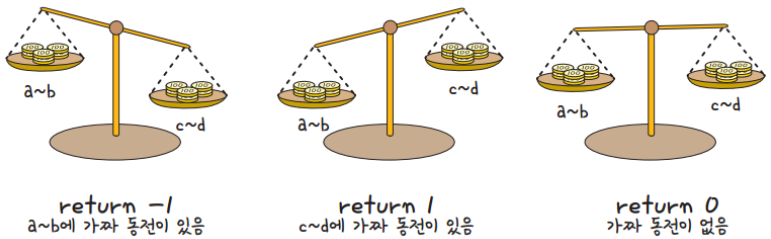
- 정리하면, 0번부터 n-1번까지 동전이 있고 이 중에 가짜 동전이 한 개 있는데, 우리가 만들 알고리즘은 양팔 저울로 저울질하여 가짜 동전의 위치 번호를 알아내는 것

문제 분석과 모델링

- ‘저울질’에 해당하는 기능을 프로그램으로 구현할 수 있는 weight 함수를 만들자

```
def weight(a, b, c, d):
```

- weight() 함수: a부터 b까지 동전을 양팔 저울의 왼쪽에, c부터 d까지 동전을 저울의 오른쪽에 올리고 저울질하는 함수
  - 이때 비교하는 동전의 무게는 같다고 가정(b-a=d-c)
- 이 함수 안에 변수 fake를 만들고 우리가 찾아야 할 가짜 동전의 위치를 저장
  - 가짜 동전 찾기 알고리즘은 fake 변수의 값을 직접 알 수는 없음 → weight() 함수를 호출해서 이 값을 찾아야만 함
- weight() 함수의 결괏값은 -1, 0, 1 세 가지 중 하나
  - a~b 쪽이 가볍다면 a와 b 사이에 가짜 동전이 있다는 뜻 → 결괏값 -1
  - c~d 쪽이 가볍다면 c와 d 사이에 가짜 동전이 있다는 뜻 → 결괏값 1
  - 양쪽 무게가 같다면 어느 쪽도 가짜 동전이 없다는 뜻 → 결괏값 0 → 이때는 저울에 올리지 않은 동전 중에 가짜 동전이 있다는 의미가 됨



가짜 동전 찾기 알고리즘

방법 ①: 하나씩 비교하기

- 무게가 적게 나가는 가짜 동전이 한 개만 있다고 했으므로 각 동전의 무게를 비교해서 가벼운 동전이 나온다면 그 동전이 바로 가짜 동전
- 0번 동전을 저울의 왼쪽에 올려놓고, 오른편에는 1번 동전부터 차례로 바꿔 가면서 저울질해 보면 가짜 동전을 쉽게 찾아낼 수 있음
- 0번과 1번 동전을 비교하는 첫 번째 저울질
  → 왼쪽이 가볍다면 0번 동전이, 오른쪽이 가볍다면 1번 동전이 가짜 동전
- 두 동전의 무게가 같다면 둘 다 가짜 동전이 아니므로 오른쪽에 2번 동전을 올리고 이 과정을 반복
- 이렇게 저울질을 하면 마지막 n-1번 동전이 가짜 동전일 경우(최악의 경우) 저울질을 n-1번 해야 가짜 동전을 찾아낼 수 있음

가짜 동전 찾기 알고리즘

- 가짜 동전을 찾는 알고리즘①

```python
# 주어진 동전 n개 중에 가짜 동전(fake)을 찾아내는 알고리즘
# 입력: 전체 동전 위치의 시작과 끝(0, n - 1)
# 출력: 가짜 동전의 위치 번호
# 무게 재기 함수

def weight(a, b, c, d):
    fake = 29  # 가짜 동전의 위치(알고리즘은 weight 함수를 이용하여 이 값을 맞혀야 함)
    if a <= fake and fake <= b:
        return -1
    if c <= fake and fake <= d:
        return 1
    return 0

# weight 함수(저울질)를 이용하여
# left와 right까지에 있는 가짜 동전의 위치를 찾아냄
def find_fakecoin(left, right):
    for i in range(left + 1, right + 1):  # left+1부터 right까지 반복
        # 가장 왼쪽 동전과 나머지 동전을 차례로 비교
        result = weight(left, left, i, i)
        if result == -1:   # left 동전이 가벼움(left 동전이 가짜)
            return left
        elif result == 1:  # i 동전이 가벼움(i 동전이 가짜)
            return i
        # 두 동전의 무게가 같으면 다음 동전으로

    # 모든 동전의 무게가 같으면 가짜 동전이 없는 예외 경우
    return -1

n = 100    # 전체 동전 개수
print(find_fakecoin(0, n - 1))
```

```
29
```

가짜 동전 찾기 알고리즘

방법 ②: 반씩 그룹으로 나누어 비교하기

- 주어진 동전을 절반씩 두 그룹으로 나눠서 양팔 저울에 올렸을 때 한쪽이 가볍다면 그 가벼운 쪽에 가짜 동전이 있다는 뜻
- 따라서 반대쪽에 있는 절반의 동전은 더는 생각할 필요가 없음
- 가벼운 쪽에 있는 동전만을 대상으로 다시 가짜 동전을 찾으면 됨
- 이렇게 하면 저울질 한 번으로 남은 동전 절반이 후보에서 탈락
- 저울질 한 번에 동전 한 개만 후보에서 탈락되던 방법 ①과 비교하면 필요한 저울질의 횟수가 눈에 띄게 줄어듦
- 남은 동전의 개수가 홀수일 경우
  - 가짜 동전 후보로 동전이 일곱 개 남아 있다고 가정해 보자
  - 7은 홀수이므로 동전 일곱 개를 똑같은 수의 두 그룹으로 나누는 것은 불가능
  - 하지만 세 개, 세 개, 나머지 한 개 이렇게 세 그룹으로 나눌 수는 있음
  - 세 그룹으로 나눈 후 개수가 세 개로 같은 두 그룹만 저울에 올려 보자
  - 왼쪽이 가볍다면 왼쪽에 올린 동전 세 개 중에 가짜 동전이 있을 것
  - 오른쪽이 가볍다면 오른쪽에 올린 동전 세 개 중에 가짜 동전이 있을 것
  - 두 그룹의 무게가 같다면 저울에 올리지 않은 나머지 동전 하나가 가짜 동전

가짜 동전 찾기 알고리즘

- 가짜 동전을 찾는 알고리즘②

```python
# 주어진 동전 n개 중에 가짜 동전(fake)을 찾아내는 알고리즘
# 입력: 전체 동전 위치의 시작과 끝(0, n - 1)
# 출력: 가짜 동전의 위치 번호

def weight(a, b, c, d):
    fake = 29  # 가짜 동전의 위치(알고리즘은 weight() 함수를 이용하여 이 값을 맞혀야 함)
    if a <= fake and fake <= b:
        return -1
    if c <= fake and fake <= d:
        return 1
    return 0

# weight() 함수(저울질)를 이용하여
# left와 right까지에 놓인 가짜 동전의 위치를 찾아냄
def find_fakecoin(left, right):
    # 종료 조건 : 가짜 동전이 있을 범위 안에 동전이 한 개뿐이면 그 동전이 가짜 동전임
    if left == right:
        return left
    # left와 right까지에 놓인 동전을 두 그룹(g1_left~g1_right, g2_left~g2_right)으로 나눔
    # 동전 수가 홀수면 두 그룹으로 나누고 한 개가 남음
    half = (right - left + 1) // 2
    g1_left = left
    g1_right = left + half - 1
    g2_left = left + half
    g2_right = g2_left + half - 1
    # 나눠진 두 그룹을 weight() 함수를 이용하여 저울질함
    result = weight(g1_left, g1_right, g2_left, g2_right)
    if result == -1:  # 그룹 1이 가벼움(가짜 동전이 이 그룹에 있음)
        return find_fakecoin(g1_left, g1_right)  # 그룹 1 범위를 재귀 호출로 다시 조사
    elif result == 1:  # 그룹 2가 가벼움(가짜 동전이 이 그룹에 있음)
        return find_fakecoin(g2_left, g2_right)  # 그룹 2 범위를 재귀 호출로 다시 조사
    else:  # 두 그룹의 무게가 같으면(나뉜 두 그룹 안에 가짜 동전이 없다면)
        return right  # 두 그룹으로 나뉘지 않고 남은 나머지 한 개의 동전이 가짜

n = 100    # 전체 동전 개수
print(find_fakecoin(0, n - 1))
```

29

알고리즘 분석

- 이 문제의 알고리즘 효율성을 '저울질 횟수'를 기준으로 생각해 보자

- 0번 동전과 나머지 동전을 일일이 비교하는 방법인 프로그램 알고리즘 1은 저울질이 최대 n-1번 필요
  → 계산 복잡도가 O(n)

- 동전 n개를 절반씩 나누어 후보를 좁히며 비교하는 방법인 프로그램 알고리즘 2→ 계산 복잡도가 O(log n)

알고리즘 분석

가짜 동전 문제와 순차/이분 탐색 알고리즘 비교

- 순차 탐색에서는 하나씩 비교하여 값을 찾아내므로 계산 복잡도가 O(n)
- 이분 탐색에서는 리스트가 이미 정렬되어 있다는 것을 전제로, 중간 값을 비교한 후 값이 있을 가능성이 없는 절반을 제외해 나가면서 값을 찾아내므로 계산 복잡도가 O(log n)
- 리스트 탐색 문제와 가짜 동전 문제는 겉으로 볼 때는 전혀 다른 문제로 보이지만, 잘 생각해 보면 구조가 비슷한 문제

문제 18 : 최대 수익 알고리즘

최대 수익 알고리즘

- 문제 : 어떤 주식에 대해 특정 기간 동안의 가격 변화가 주어졌을 때, 그 주식 한 주를 한 번 사고팔아 얻을 수 있는 최대 수익을 계산하는 알고리즘을 만들어 보세요.

- 주식을 거래해서 얻을 수 있는 최대 수익(이익)을 구하는 문제
  - 어떤 주식의 가격이 표 18-1과 같이 매일 변했다고 해 보자

날짜	주가(원)	날짜	주가(원)
6/1	10,300	6/8	8,300
6/2	9,600	6/9	9,500
6/3	9,800	6/10	9,800
6/4	8,200	6/11	10,200
6/5	7,800	6/12	9,500

  - 이 주식 한 주를 한 번 사고팔아 얻을 수 있는 최대 수익은 얼마일까?
  - 단, 손해가 나면 주식을 사고팔지 않아도 됨 → 최대 수익은 항상 0 이상

문제 분석과 모델링

- **주식 거래로 수익을 내는 가장 좋은 방법은 '가장 쌀 때 사서 가장 비쌀 때 파는 것'**
  - 얼핏 생각하면 주가(주식의 가격)의 최댓값에서 주가의 최솟값을 뺀 것으로 착각하기 쉬움
  - 앞의 표 을 예로 들면, 6월 1일의 주가 10,300원이 최댓값이고 6월 5일의 주가 7,800원이 최솟값
  - 하지만 아직 사지도 않은 주식을 6월 1일에 먼저 팔고 6월 5일에 주식을 살 수는 없으므로 단순히 최댓값과 최솟값을 구하는 것만으로는 올바른 답을 얻을 수 없음
- **주어진 자료를 모델링하여 파이썬 프로그램으로 만들어보자**
  - 우리에게 주어진 정보는 날짜와 주가 정보이지만 이 문제는 얻을 수 있는 최대 수익만 물어 보았으므로 정확한 날짜 정보는 없어도 상관없음
  - 따라서 정보를 단순화하여 각 날의 주식 가격만 뽑아 stock이라는 리스트로 만듦

```
stock = [10300, 9600, 9800, 8200, 7800, 8300, 9500, 9800, 10200, 9500]
```

  - 이제 이 리스트 값을 이용해서 얻을 수 있는 최대 수익을 계산해 보자!

최대수익 찾기 알고리즘

방법 ①: 가능한 모든 경우를 비교하기

- 가장 간단한 방법은 주식을 살 수 있는 모든 날과 팔 수 있는 모든 날의 주가를 비교해서 가장 큰 수익을 찾는 것
  - 예를 들어 첫째 날 10,300원에 주식을 샀다면 둘째 날 부터의 주식 가격인 9,600원, 9,800원 … 9,500원 중 하나로 주식을 팔 기회가 생김
  - 마찬가지로 둘째 날 9,600원에 주식을 샀다면 셋째 날 부터의 주식 가격인 9,800원, 8,200원 … 9,500원 중 하나로 주식을 팔 기회가 생김
- 이런 식으로 모든 경우를 비교해서 가장 큰 이익을 내는 경우를 찾으면 원하는 최대 수익을 계산할 수 있음
- 문제 3 동명이인 찾기에서 가능한 모든 사람을 비교하던 방식과 똑같음

```
# 리스트 안에 있는 n개 자료를 빠짐없이 한 번씩 비교하는 방법
for i in range(0, n - 1):
    for j in range(i + 1, n):
        # i와 j로 필요한 비교
```

- 이 경우 비교 횟수는 $\frac{n(n-1)}{2}$ 번이고 계산 복잡도는 $O(n^2)$이었음

최대수익 찾기 알고리즘

- 최대 수익을 구하는 알고리즘①

```python
# 주어진 주식 가격을 보고 얻을 수 있는 최대 수익을 구하는 알고리즘
# 입력: 주식 가격의 변화 값(리스트: prices)
# 출력: 한 주를 한 번 사고팔아 얻을 수 있는 최대 수익 값

def max_profit(prices):
    n = len(prices)
    max_profit = 0  # 최대 수익은 항상 0 이상의 값

    for i in range(0, n - 1):
        for j in range(i + 1, n):
            profit = prices[j] - prices[i]  # i날에 사서 j날에 팔았을 때 얻을 수 있는 수익
            if profit > max_profit:  # 이 수익이 지금까지 최대 수익보다 크면 값을 고침
                max_profit = profit

    return max_profit

stock = [10300, 9600, 9800, 8200, 7800, 8300, 9500, 9800, 10200, 9500]
print(max_profit(stock))

```

```
2400
```

최대수익 찾기 알고리즘

방법 ②: 한 번 반복으로 최대 수익 찾기

- 모든 경우를 비교하는 방법인 프로그램 1은 간단하고 직관적이지만, 불필요한 비교를 너무 많이 함
- 프로그램 1이 사는 날을 중심으로 생각한 것이라면 이번에는 파는 날을 중심으로 생각을 바꿔보자
  - 예를 들어 6월 10일에 9,800원을 받고 주식을 팔았다고 가정
  - 이때 얻을 수 있는 최고 수익은 6월 10일 이전에 가장 주가가 낮았던 날인 6월 5일에 7,800원에 산 경우이므로 2,000원
  - 만약 6월 11일 에 10,200원에 팔았다면, 6월 5일 7,800원과의 차이인 2,400원이 최대 수익
- 파는 날 기준, 이전 날들의 주가 중 최솟값만 알면 최대 수익을 쉽게 계산

최대수익 찾기 알고리즘

- 알고리즘

① 최대 수익을 저장하는 변수를 만들고 0을 저장

② 지금까지의 최저 주가를 저장하는 변수를 만들고 첫째 날의 주가를 기록

③ 둘째 날의 주가부터 마지막 날의 주가까지 반복

④ 반복하는 동안 그날의 주가에서 최저 주가를 뺀 값이 현재 최대 수익보다 크면 최대 수익 값을 그 값으로 고침

⑤ 그날의 주가가 최저 주가보다 낮으면 최저 주가 값을 그날의 주가로 고침

⑥ 처리할 날이 남았으면 4번 과정으로 돌아가 반복하고, 다 마쳤으면 최대 수익에 저장된 값을 결과값으로 돌려주고 종료

최대수익 찾기 알고리즘

- 최대 수익을 구하는 알고리즘②

```python
# 주어진 주식 가격을 보고 얻을 수 있는 최대 수익을 구하는 알고리즘
# 입력: 주식 가격의 변화 값(리스트: prices)
# 출력: 한 주를 한 번 사고팔아 얻을 수 있는 최대 수익 값

def max_profit(prices):
    n = len(prices)
    max_profit = 0          # 최대 수익은 항상 0 이상의 값
    min_price = prices[0]  # 첫째 날의 주가를 주가의 최솟값으로 기억

    for i in range(1, n):  # 1부터 n-1까지 반복
        profit = prices[i] - min_price  # 지금까지의 최솟값에 주식을 사서 i날에 팔 때의 수익
        if profit > max_profit:   # 이 수익이 지금까지 최대 수익보다 크면 값을 고침
            max_profit = profit
        if prices[i] < min_price: # i날 주가가 최솟값보다 작으면 값을 고침
            min_price = prices[i]

    return max_profit

stock = [10300, 9600, 9800, 8200, 7800, 8300, 9500, 9800, 10200, 9500]
print(max_profit(stock))

```

```
2400
```

알고리즘 분석

알고리즘 비교

- 첫 번째 알고리즘보다 두 번째 알고리즘이 더 효율적임
- 모든 경우를 비교한 첫 번째 알고리즘(프로그램 1)은 문제 3 동명이인 찾기와 비슷한 구조 → 계산 복잡도는 $O(n^2)$
- 리스트를 한 번 탐색하면서 최대 수익을 계산한 두 번째 알고리즘(프로그램 2)은 문제 2 최댓값 찾기와 비슷한 구조 → 계산 복잡도는 $O(n)$
- 입력 크기가 커질수록, 즉 더 많은 날의 주가가 입력으로 주어질수록 두 번째 알고리즘이 첫 번째 알고리즘보다 결과를 훨씬 빨리 낼 거라고 충분히 예상 가능

알고리즘 분석

- 최대 수익 문제를 두 가지 다른 방법으로 풀 때 걸리는 시간을 비교하는 프로그램을 만들어 보자

```python
# 최대 수익 문제를 푸는 두 알고리즘의 계산 속도 비교하기
# 최대 수익 문제를 O(n*n)과 O(n)으로 푸는 알고리즘을 각각 수행하여
# 걸린 시간을 출력/비교함

import time     # 시간 측정을 위한 time 모듈
import random  # 테스트 주가 생성을 위한 random 모듈

# 최대 수익: 느린 O(n*n) 알고리즘
def max_profit_slow(prices):
    n = len(prices)
    max_profit = 0
    for i in range(0, n - 1):
        for j in range(i + 1, n):
            profit = prices[j] - prices[i]
            if profit > max_profit:
                max_profit = profit

    return max_profit

# 최대 수익: 빠른 O(n) 알고리즘
def max_profit_fast(prices):
    n = len(prices)
    max_profit = 0
    min_price = prices[0]
    for i in range(1, n):
        profit = prices[i] - min_price
        if profit > max_profit:
            max_profit = profit
        if prices[i] < min_price:
            min_price = prices[i]

    return max_profit
```

```python
def test(n):
    # 테스트 자료 만들기(5000부터 20000까지의 난수를 주가로 사용)
    a = []
    for i in range(0, n):
        a.append(random.randint(5000, 20000))
    # 느린 O(n*n) 알고리즘 테스트
    start = time.time()       # 계산 시작 직전 시각을 기억
    mps = max_profit_slow(a)  # 계산 수행
    end = time.time()         # 계산 시작 직후 시각을 기억
    time_slow = end - start   # 두 시각을 빼면 계산에 걸린 시간
    # 빠른 O(n) 알고리즘 테스트
    start = time.time()       # 계산 시작 직전 시각을 기억
    mpf = max_profit_fast(a)  # 계산 수행
    end = time.time()         # 계산 시작 직후 시각을 기억
    time_fast = end - start   # 두 시각을 빼면 계산에 걸린 시간
    # 결과 출력: 계산 결과
    print(n, mps, mpf)  # 입력 크기, 각각 알고리즘이 계산한 최대 수익 값(같아야 함)
    m = 0  # 느린 알고리즘과 빠른 알고리즘의 수행 시간 비율을 저장할 변수
    if time_fast > 0:  # 컴퓨터 환경에 따라 빠른 알고리즘 시간이 0으로 측정될 수 있음(이럴 때는 0을 출력)
        m = time_slow / time_fast  # 느린 알고리즘 시간 / 빠른 알고리즘 시간
    # 입력 크기, 느린 알고리즘 수행 시간, 빠른 알고리즘 수행 시간, 느린 알고리즘 시간 / 빠른 알고리즘 시간
    # %d는 정수 출력, %.5f는 소수점 다섯 자리까지 출력을 의미
    print("%d %.5f %.5f %.2f" % (n, time_slow, time_fast, m))

test(100)
test(10000)
#test(100000)  # 수행 시간이 오래 걸리므로 일단 주석 처리
```

```
100 14345 14345
100 0.00039 0.00001 36.40
10000 14992 14992
10000 4.67046 0.00122 3838.03
```

알고리즘 분석

실행 결과

입력 크기 n	최대 수익	느린 알고리즘 수행 시간	빠른 알고리즘 수행 시간	느린 알고리즘 시간 / 빠른 알고리즘 시간
100	14658	0.00061초	0.00002초	40.87
10000	14996	6.09124초	0.00167초	3653.97
100000	15000	819.66065초	0.01953초	41969.70

※ Intel i7 2.7Ghz, macOS 10.12.3, Python 3.6.0 환경에서 테스트한 결과

- 실행 결과를 보면 입력 크기를 100으로 입력했을 때는 빠른 알고리즘과 느린 알고리즘의 계산 시간 차이가 40배 정도 남
- 그러다 입력 크기를 10,000과 100,000으로 입력했더니 차이가 3,700배와 42,000배로 급격히 벌어지는 것을 확인할 수 있음
- 입력 크기가 더 커진다면 두 알고리즘으로 답을 찾는 데 걸리는 시간의 격차는 훨씬 더 벌어질 것