Ssafy_9W-2_Exhaustive-search

Ssafy_9W-2_Exhaustive-search

1. Iteration & Recursion

- 반복과 재귀는 유사한 작업 수행
- 반복은 수행하는 작업이 완료될 때 까지 계속 반복
  - 루프 (for, while 구조)
  - **코드를 n번 반복할 수 있다.**
- 재귀는 주어진 문제의 해를 구하기 위해 동일하면서 더 작은 문제의 해를 이용하는 방법
  - 하나의 큰 문제를 해결할 수 있는 (해결하기 쉬운) 더 작은 문제로 쪼개고 결과들을 결합한다.
  - **재귀 호출은 n중 반복문을 만들 수 있다.**

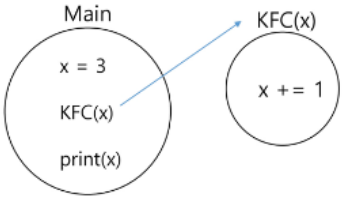
```
for a in range(1, 4):
    for b in range(1, 4):
        print(a, b)

'''
1 1
1 2
1 3
2 1
2 2
2 3
3 1
3 2
3 3
'''
```

- 위처럼 `1 1 ~ 3 3` 까지 출력할 수 있다.
- 하지만 N에 따라 출력해야 하는 범위가 달라진다면 (`111 ~ 333 if N = 3` or `1111 ~ 3333 if N = 4` etc.) 단순 for loop로는 구현이 어려워진다. 이때 아래와 같이 재귀로 구현할 수 있다.

```
path = []
N = 3

def run(lev):
    if lev == N:
        print(path)
        return

    for i in range(1, 4):
        path.append(i)
        run(lev+1)
        path.pop()
```

1.1. 재귀 특징

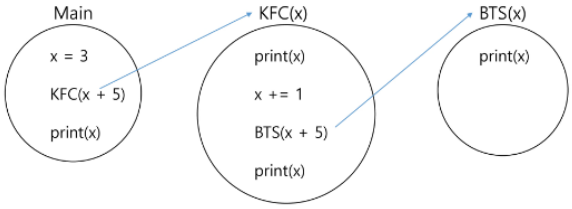
1.1.1. Call by value



- main 함수의 x와 kfc 함수의 x는 서로 다른 객체다.
- 즉, kfc 함수를 호출할 때, int 타입 객체를 전달하면 **값만 복사**[1]가 된다.
- 때문에 위 그림의 출력 결과는 4가 아닌 3이 된다.

```
def kfc(x):
    print(x)
    x += 1
    print(x)

x = 3
kfc(x+1)
print(x)
```

- kfc에 전달되는 x의 경우 값만 복사되어 전달되는 형태다.
- 따라서 위 코드의 실행 결과는 `4, 5, 3`이 출력된다.

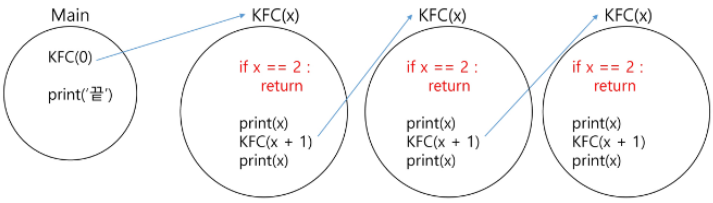


- 함수가 종료되면 main으로 돌아오는 게 아닌, **해당 함수를 호출했던 곳으로 돌아온다.**

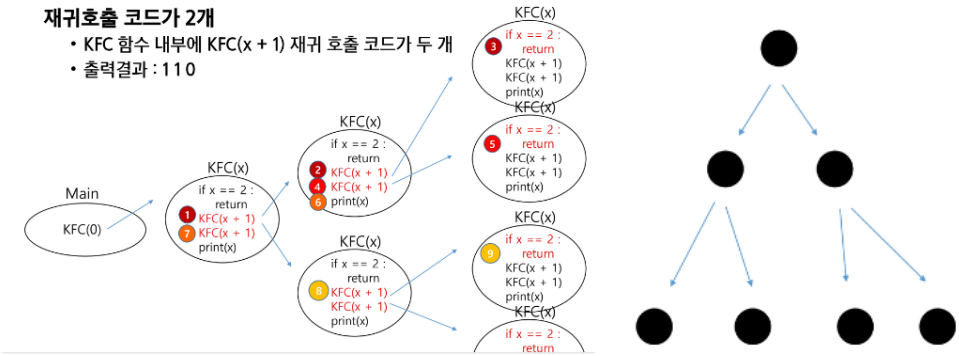
```
def bbq(x): # 9
    x += 10
    print(x) # 19

def kfc(x):
    print(x) # 4
    x += 3 # 7
    bbq(x+2)
    print(x) # 7

x = 3
kfc(x+1)
print(x) # 3
```

- 위 코드도 이전 코드와 동일하게 x의 값만 복사되어 전달되며, 함수 내에서의 x 값은 유지된다.

1.1.2. Base case



- 무한 재귀호출을 방지하는 방법
- 위 그림의 경우 x가 2가 되면 다음 재귀에 들어갈 수 없기 때문에 무한 재귀에 빠지지 않는다.



- 위처럼 재귀 호출 코드가 두 개(두 개의 branch)인 경우도 생각해볼 수 있다.

1.1.3. Level

```
def kfc(x):
    if x == 3:
        return
    kfc(x+1)
    kfc(x+1)
    kfc(x+1)
    kfc(x+1)
kfc(0)
```

- 위처럼 level 3, branch 4인 형태(재귀 호출이 다수인 경우)의 코드를 다음과 같이 표현할 수 있다.

```
def kfc(x):
    if x == 3:
        return

    for i in range(4):
        kfc(x+1)
kfc(0)
```

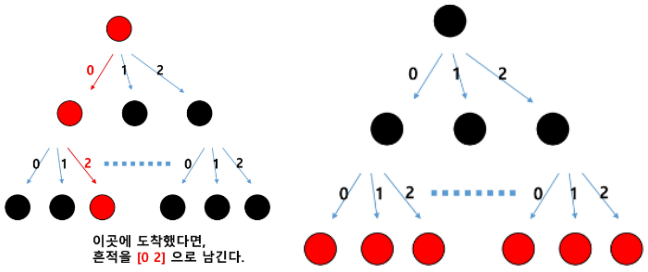
2. 순열

- 서로 다른 N 개에서, R 개를 중복 없이, 순서를 고려하여 나열하는 것
- 예:
  - [0] [1] [2] 로 구성된 3장의 카드가 다량 존재한다. 이 중 2장을 뽑아 순열을 나열하라.
  - [0 1] [0 2] [1 0] [1 2] [2 0] [2 1]

2.1. 중복 순열

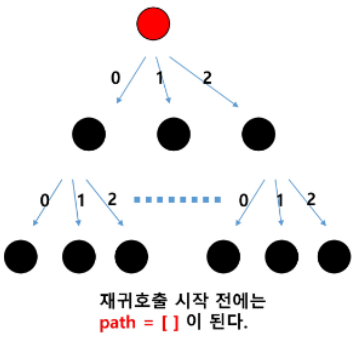
- 서로 다른 N 개에서, R개의 중복을 허용하고, 순서를 고려하여 나열하는 것
- 예:
  - [0] [1] [2] 로 구성된 3장의 카드가 다량 존재한다. 이 중 2장을 뽑아 중복순열을 나열하라.
  - [0 0] [0 1] [0 2] [1 0] [1 1] [1 2] [2 0] [2 1] [2 2]

2.1.1. 구현 원리



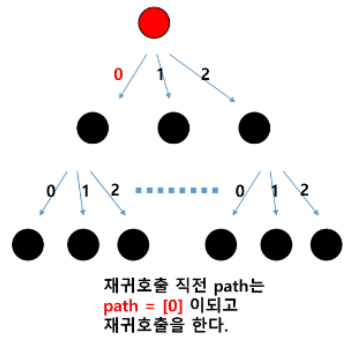
- (좌) 재귀 호출 마다, 이동 경로를 흔적으로 남긴다
- (우) 가장 마지막 레벨에 도착했을 때, 이동 경로를 출력한다.
- 결과는 [0 0] [0 1] [0 2] [1 0] [1 1] [1 2] [2 0] [2 1] [2 2]와 같다.

2.1.2. 코드



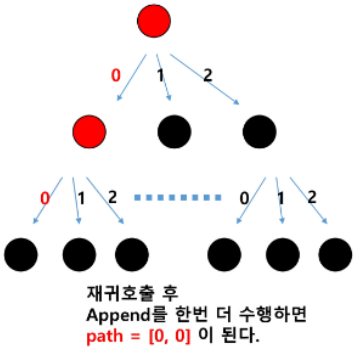

```
path = []

def kfc(x):
    if x == 2:
        return

    for i in range(3):
        kfc(x+1)

kfc(0)
```

- 먼저 path라는 전역 리스트를 만든다.
- 그리고 level 2, branch 3으로 동작하는 재귀 코드를 구현한다.

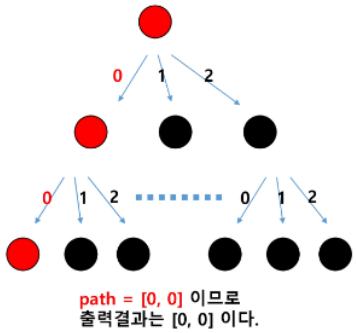



```
path = []

def kfc(x):
    if x == 2:
        return

    for i in range(3):
        path.append(i) #. point!
        kfc(x+1)

kfc(0)
```

- 재귀 호출 직전에 이동할 곳의 위치를 path 리스트에 기록한다.

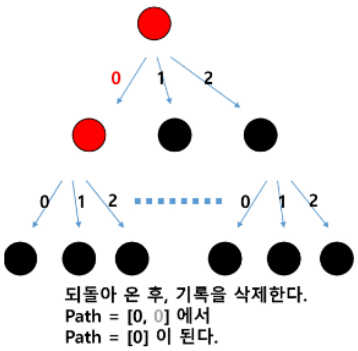


- 재귀 호출 후 다시 `path.append(i)`를 수행한다.




```
path = []

def kfc(x):
    if x == 2:
        print(path) #. point!
        return

    for i in range(3):
        path.append(i)
        kfc(x+1)

kfc(0)
```

- 바닥에 도착했으니, 출력하는 코드를 수행([0, 0])한다.



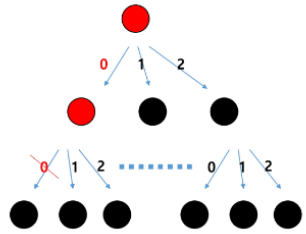

```
path = []

def kfc(x):
    if x == 2:
        print(path)
        return

    for i in range(3):
        path.append(i)
        kfc(x+1)
        path.pop() #. point!
```

```
kfc(0)
```

- 함수가 리턴되고, 돌아오면 기록을 삭제한다.
- 다음 for 문이 진행되고 i 는 1이 된다.
- 깊이에 도달했으니 출력([0, 1])한다.
- 최종적으로 [0 0] [0 1] [0 2] [1 0] [1 1] [1 2] [2 0] [2 1] [2 2]가 출력된다.

2.2. 순열

- 중복순열 코드를 작성한다.
- 중복을 제거하는 코드를 추가하면 순열 코드가 된다.
- 중복을 제거하는 원리는 전역 리스트를 사용해 이미 선택했던 숫자인지 아닌지 구분할 수 있다.
- 이를 used 혹은 visited 배열이라 한다.
- 예:



  - 재귀 호출 직전, 이미 선택한 숫자인지 검사하는 코드가 필요하다.

2.2.1. 구현

```
path = []
used = [False, False, False] #. point!

def kfc(x):
    if x == 2:
        print(path)
        return

    for i in range(3):
        if used[i] == True: continue
        used[i] = True #. point!
        path.append(i)
        kfc(x+1)
        path.pop()
        used[i] = False #. point!

kfc(0)
```

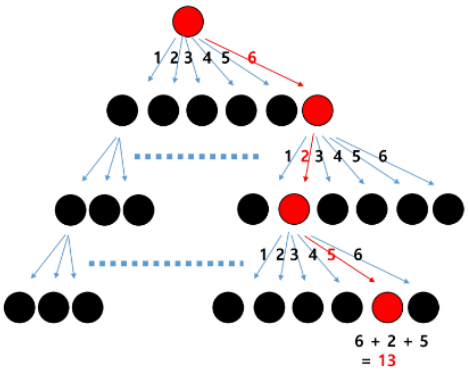
- 이미 사용한 코드를 구분하는 코드. 사용한 숫자일 경우 재귀호출을 생략함.
- 처음 사용하는 숫자라면 used에 기록, 처리가 끝났다면 기록을 지워준다.

3. 완전 탐색

- Brute-Force
- 모든 가능한 경우의 수를 모두 시도해, 정답을 찾아내는 알고리즘
- 예:
  - 자전거 열쇠 비밀번호 맞추기
    - 1111 ~ 9999 사이의 숫자 중 네 자리 수를 맞춰야 하는 경우
    - 4중 for loop으로 시도
  - 1 ~ 9로 이루어진 N자리 숫자를 맞추는 경우
    - 순열 코드(재귀)로 구현하여 모두 시도.

2.1. 주사위 눈금의 합

- 3개의 주사위를 던져 나올 수 있는 모든 경우에 대해, 합이 10 이하가 나오는 경우는 총 몇 가지 인가?
- 먼저 합을 출력하는 코드를 작성한다.

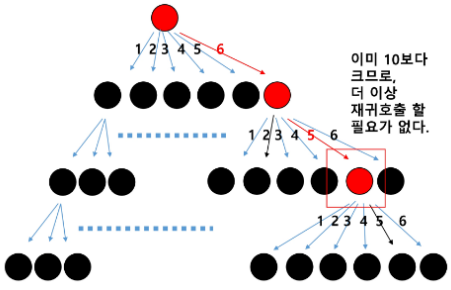



```
path = []

def kfc(x, sum):
    if x == 3:
        print(f'{path} = {sum}')
        return

    for i in range(1, 7):
        path.append(i)
        kfc(x+1, sum+i)
        path.pop()

kfc(0, 0)
```

- 재귀호출 때마다 선택한 값의 누적 합을 구한다.
- parameter에 sum을 추가해 구현한다.
  - sum = 지금까지의 합
  - i = 선택한 주사위 눈금
  - 재귀 호출 시 sum + i 값을 전달한다

```
path = []

def kfc(x, sum):
    if x == 3:
        if sum <= 10: #. point!
            print(f'{path} = {sum}')
        return

    for i in range(1, 7):
        path.append(i)
        kfc(x+1, sum+i)
        path.pop()

kfc(0, 0)
```

- 실제로는 모두 탐색하지만 출력은 하지 않는 방법.



- 위 경우 6 -> 5 이후는 10보다 크므로 재귀호출 할 필요가 없다.

```
path = []

def kfc(x, sum):
    if sum > 10: #. point!
        return

    if x == 3:
        print(f'{path} = {sum}')
        return

    for i in range(1, 7):
        path.append(i)
        kfc(x+1, sum+i)
        path.pop()

kfc(0, 0)
```

- 누적합이 10 넘어가는 순간 더 탐색할 필요 없으니 `return` 한다.

2.2. 연속 3장의 트럼프 카드

2.3. Baby-Jin

---

1. 이것과 비슷한 내용으로, Call by reference가 있다.↩