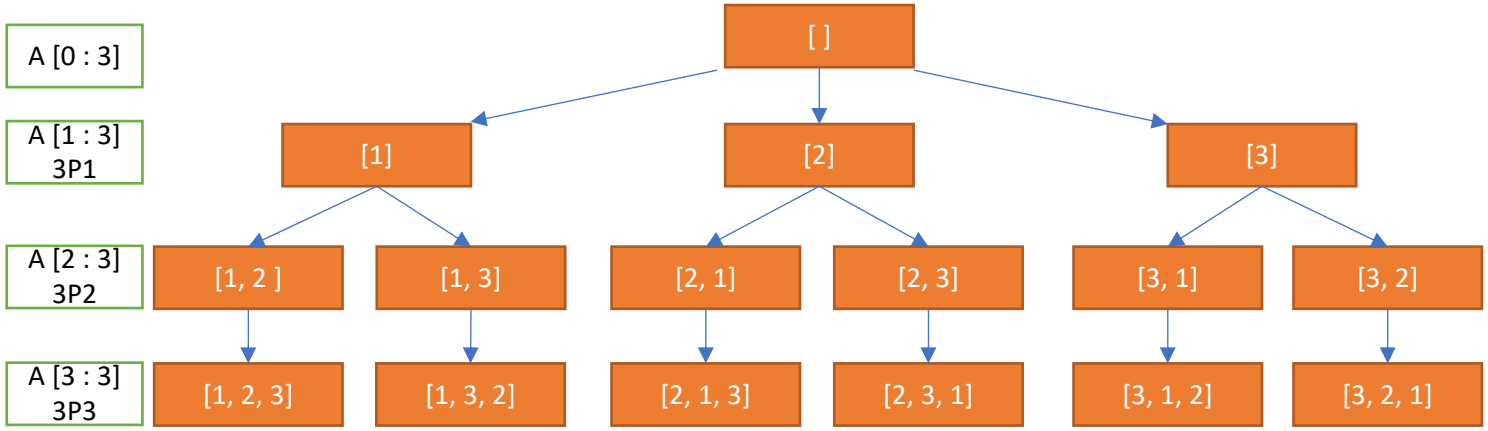
DFS 응용 문제들

순열

순열의 수식은 n!/(n-r)! 이다.
순서가 중요하므로 원소들이 모두 같아도 순서가 다르면 다른 것으로 간주한다.
모든 결과를 생성해야하는 문제일 경우,
결국, 가능한 모든 경우를 그래프 형태로 나열할 수 있는데, dfs를 사용해서 구현할 수 있다.
리스트 [1, 2, 3] 에서 가능한 모든 순열을 리턴하는 문제를 풀이하는 방법을 한번 보자.



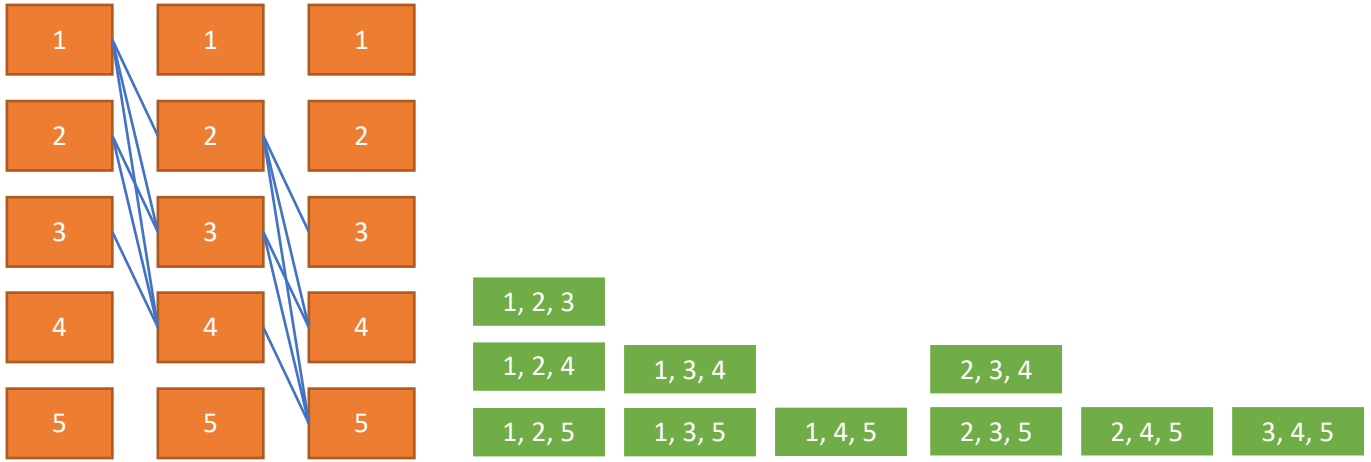
리프(leaf) 노드, A[3:3]의 모든 노드가 순열의 최종 결과다.
A[0:3]은 3개, A[1:3]은 2개, A[2:3]은 1개 순으로 이는 순열의 수식이기도 한(3x2x1) 형태이기도 하다.
위에서는 3! / 0!, 3!이 가능한 모든 갯수이다.
2개를 뽑을 경우는 A[2:3]으로 3!/(3-2)!, 3!이 되고, 1개를 뽑을 경우는 A[1:3]으로 3!/(3-1)!이 된다.

이전 값을 하나씩 덧붙여 계속 재귀 호출을 진행하다가 리프 노드에 도달한 경우, 즉, len(elements) == 0일 때 결과를 하나씩 담는다. 이렇게 순열 생성을 계속 재귀 호출하면 풀이를 완성할 수 있다

```python
#-*-coding:utf-8-*-
def permute(nums, k):
  results = []
  prev_elements = []
  n = len(nums)

  def dfs(elements, k):
    # 리프 노드일 때 결과 추가
    if len(elements) == n-k:
      results.append(prev_elements[:])
      print(prev_elements)
      return
    # 순열 생성 재귀 호출
    for e in elements:
      next_elements = elements[:]
      next_elements.remove(e)

      prev_elements.append(e)
      dfs(next_elements, k)
      prev_elements.pop()

  dfs(nums, k)
  return results

if __name__ == "__main__":
l1 = [1, 2, 3]
permute(l1, 2)
```

조합

조합의 갯수를 구하는 수식은 n!/ r*(n-r)! 이다.
순서가 중요하지 않으므로 원소들이 같으면 중복으로 여겨서 무시한다.
모든 결과를 생성해야하는 문제일 경우, 순열과 비슷하게 dfs를 사용해서
결국, 가능한 모든 경우를 그래프 형태로 나열할 수 있는데, 약간의 차이가 있다.
리스트 [1, 2, 3, 4, 5] 에서 3개의 조합을 리턴하는 문제를 풀이하는 방법을 한번 보자.



순열의 경우 자기 자신을 제외하고 모든 요소를 next_elements로 처리했으나,
이와 달리 조합의 경우 자기 자신뿐만 아니라 앞의 모든 요소를 배제하고
next_elements를 구성한다.

따라서 여기서는 그냥 elements 라는 이름으로 다음과 같이 처리한다.

1부터 순서대로 for문으로 반복하되, 재귀 호출할 때 넘겨주는 값은
자기 자신 이전의 값을 고정하여 넘긴다.
따라서 남아있는 값끼리 나머지 조합을 수행하게 되며, k=0이 되면 결과에 삽입한다.
마찬가지로 참조로 처리되지 않도록, 결과는 [:] 연산자로 값 자체를 복사해 추가한다.

순열 문제와 이 문제는 서로 비슷한 면이 있지만,
순열과 조합이 다르듯 구현 방식에도 차이가 있다.
무엇보다 이 문제는 모든 순열을 생성하는 이전 문제와 달리
K개의 조합만을 생성해야한다는 제약 조건이 추가된 문제다.
따라서 dfs()함수에서 k값을 별도로 전달받아 1씩 줄여나가며 재귀 호출하는
구조로,
K가 0이 되면 바로 빠져나가는 로직이 추가되어있다.

```python
#-*-coding:utf-8-*-
def combine(n: int, k: int):
  results = []

  def dfs(elements, start: int, k: int):
    if k == 0:
      results.append(elements[:])
      return
    # 자신 이전의 모든 값을 고정하여 재귀 호출
    for i in range(start, n+1):
      elements.append(i)
      dfs(elements, i+1, k-1)
      elements.pop()

  dfs([], 1, k)
  return results
```

조합(리스트)

```python
def combination_list(nums:list, k:int):
  results = []

  def dfs(elements, start:int, k:int):
    if k == 0:
      results.append(elements[:])
      return
    for i in range(start, len(nums)):
      elements.append(nums[i])
      dfs(elements, i+1, k-1)
      elements.pop()
  dfs([], 0, k)
return results
```

전화번호 문자 조합

2 ~ 9 까지의 숫자가 주어졌을 때 전화번호로 조합 가능한 모든 문자를 출력하라.
입력: "23"
출력: ["ad", "ae", "af", "bd", "be", "bf", "cd", "ce", "cf",]
→ 2는 abc, 3은 def가 가능하므로 각각 한문자씩 9개의 문자로 조합이 가능하다.

Digits는 입력 값이며, 각.자릿수에 해당하는 키판 배열을 DFS로 탐색하면 결과가 완성된다.
Digits는 입력 값이며, dic는 키판 배열이다. 입력 값을 자릿수로 쪼개어 반복하고, 숫자에 해당하는 모든 문자열을 반복하면서 마찬가지로 문자 단위로 재귀 탐색한다.

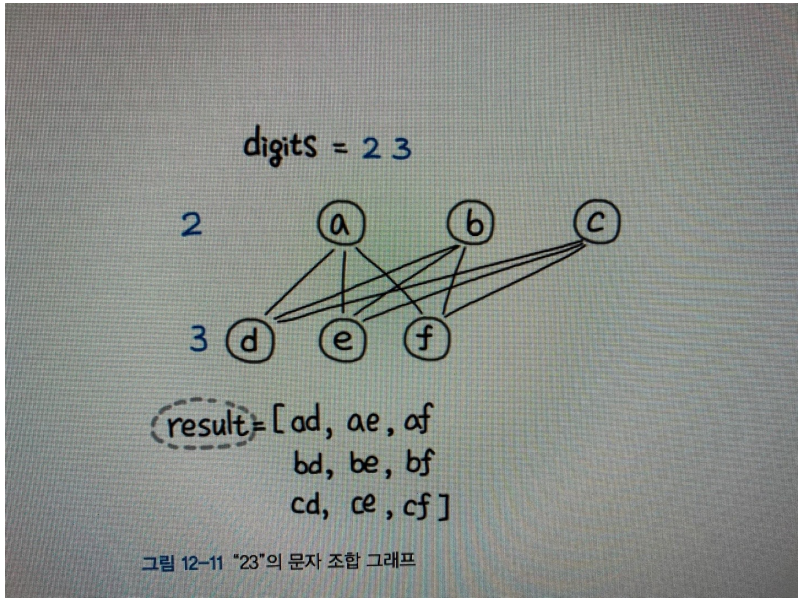


그림 12-11 "23"의 문자 조합 그래프

이 문제는 전체를 탐색하여 풀이할 수 있다.
항상 전체를 탐색해야하고 가지치기 등으로 최적화할 수 있는 문제는 아니기 때문에 결과는 비슷하다.
가능한 경우의 수를 위 그림과 같이 모두 조합하는 형태로 전체를 탐색한 후 백트래킹하면서 결과를 조합할 수 있다.

```python
#-*-coding:utf-8-*-
def letterCombination(digits:str) -> list[str]:
  def dfs(index, path):
    # 끝까지 탐색하면 백트래킹
    if len(path) == len(digits):
    result.append(path)
    return
    # 입력값 자릿수 단위 반복
    for i in range(index, len(digits)):# 2: slen("23")
    # 숫자에 해당하는 모든 문자열 반복
      for j in dic[digits[i]]:#a, b, c
        dfs(i+1, path+j)#a, b, c 각각에 해당하는 d, e, f를 조합.

  # 예외 처리
  if not digits:
  return []

  dic = {"2": "abc", "3": "def", "4": "ghi", "5": "jkl",
         "6": "mno", "7": "pqrs", "8": "tuv", "9": "wxyz"}
  result = []
  dfs(0, "")

return result

result = letterCombination("23")
print(result)
```

순열, 조합의 DFS, itertools 성능 비교

구현의 효율성, 성능을 위해 사용..
그러나 C++의 경우, 대부분의 문제에서 라이브러리를 이용한 조합을 사용할 경우, fail이 발생..
다시 한번 라이브러리로 테스트해보자..
메모리 사용량 때문인가??

조합은 Itertools의 함수의 결과가 튜플이라는 점.
대부분의 문제들은 리스트를 반환하도록 하므로 리스트로 반환해야하는 번거로움이 있다.

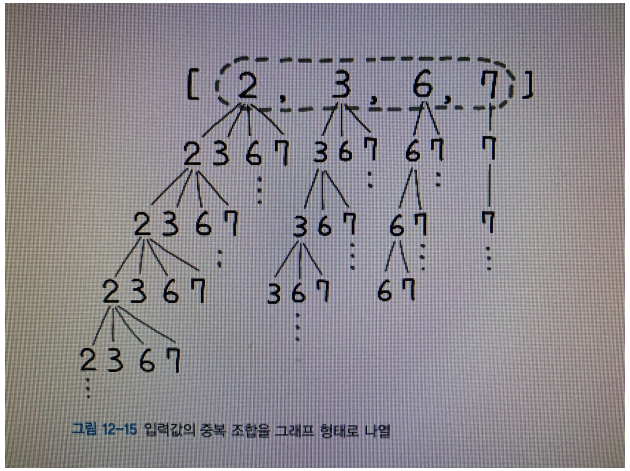
순열과 달리 조합의 경우, DFS와 모듈의 성능 차이가 꽤 큰 편이다.
모듈 자체의 성능이 좋은 것도 있지만 이해하기 쉽게 구현하려다 보니
DFS 풀이를 다소 비효율적으로 구현하기도 했다.
이 풀이와는 달리 k와 n을 뒤집어서 k-1을 재귀 호출하는 형태로 하면 탐색범위를 훨씬 더 좁혀 나갈 수 있다.
다만, 이 경우, 알고리즘을 직관적으로 이해하기 어려운 단점이 있다.

풀이	방식	실행 시간
1	DFS를 활용한 순열 생성	40ms
2	itertools	36ms

풀이	방식	실행 시간
1	DFS로 K개 조합 생성	536ms
2	itertools	76ms

DFS로 중복 조합 그래프 탐색

조합을 응용한 문제이다.
합 target을 만들 수 있는 모든 번호 조합을 찾는 문제인데, 앞서 순열 문제와 유사하게 DFS와 백트래킹으로 풀이할 수 있다.
간단히 구조를 그려보면 다음 그림과 같은 입력값의 중복 조합 그래프를 풀이하는 문제로 도식화 할 수 있다.
아래 그림보다는 다음 장의 dfs로 문자열 조합을 생성하는 문제의 그림을 참고하는게 더 좋을 듯함.



모든 중복 조합에서 찾아야 하기 때문에 이 그림과 같이 항상 부모의 값부터 시작하는 그래프로 구성할 수 있다.
만약 조합이 아니라 순열을 찾은 문제라면 자식노드는 항상 처음부터 시작해야해서 훨씬 더 많은 계산이 필요하다.
그러나 조합은 각각의 노드가 자기 자신부터 하위 원소까지의 나열로만 정리할 수 있다.
이 중복 조합 그래프를 DFS로 다음과 같이 탐색할 수 있다.

DFS로 재귀 호출하되, dfs()함수의 첫번째 파라미터는 합을 갱신해나갈 csum(candidates_sum의미) , 두번째 파라미터는 순서(자기 자신을 포함하는), 세번째 파라미터는 지금까지의 탐색경로로 정한다.
그런데 이 탐색 코드는 종료 조건이 없으며, 자기 자신을 포함하기 때문에 무한히 탐색하게 되기 때문에 다음과 같은 종료조건이 필요하다.

1. csum < 0: 목표값을 초과한 경우로 탐색을 종료한다.
2. Csum = 0: csum의 초기값은 target이며,
   따라서 csum의 0은 target과 일치하는 정답이므로
결과 리스트에 추가하고 탐색을 종료한다.
아래 구문에서 i가 아닌 0을 기입하면 순열로 풀이가 가능하다.

dfs(csum - candidates[i], 0, path+candidates[i])

```python
def combination(self, candidates:List[int], target:int)
-> List[List[int]]:
  result = []

  def dfs(csum, index, path):
    # 종료 조건
    if csum < 0:
      return
    if csum == 0:
      result.append(path)
      return
    # 자신부터 하위 원소까지의 나열 재귀 호출
    for i in range(index, len(candidates)):
      dfs(csum - candidates[i], i, path+candidates[i])
  dfs(target, 0, [])
  return result
```

```
combination(order, "", crs);
void combination(string src, string crs, int depth)
{
  if (crs.size() == depth) combi[crs]++;
  else
    for (int i = 0; i < src.size(); i++)
      combination(src.substr(i+1), crs+src[i], depth);
}
```