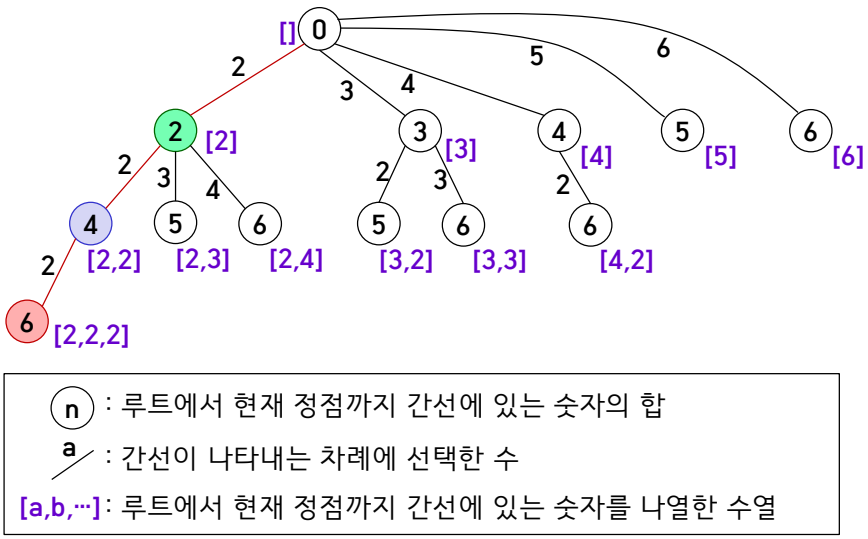
가능성 트리에 대한 탐색법과 이를 코드로 구현하는 방법

주어진 문제에 대한 여러 가능한 해 중 최적의 해를 찾아야 하는 경우, 먼저 시도해 볼 수 있는 방법은 **전체 solution space(모든 가능한 해들의 집합)를 탐색**하며 최적의 해를 찾아보는 것이다. 이러한 solution space는 많은 경우 **가능성 트리** 형태로 표현할 수 있다. 이 자료에서는 이렇게 가능성 트리 형태로 표현되는 solution space를 탐색하는 방법을 알아보고, 또한 이를 어떻게 코드로 구현하는지도 알아보겠다.

트리 형태의 solution space를 탐색하는 예제에서 시작해 보자. ① **자연수(≥2)로 이루어진 수열 중 ② 합이≤6인 모든 수열을 탐색**한다고 가정하겠다. [2, 2, 2]는 이 조건에 맞는 수열이다 - 수열을 이루는 모든 숫자가 ≥2이며, 이들의 합 2 + 2 + 2 ≤ 6 이기 때문이다. 이에 반해 [1, 2, 3]은 주어진 조건에 맞지 않아 탐색 대상이 아니다 - 수열의 첫 번째 숫자 1이 2보다 작아 조건 ①에 어긋나기 때문이다. 마찬가지로 [2, 2, 3]도 주어진 조건에 맞지 않아 탐색 조건이 아니다 - 수열을 이루는 숫자의 합 2 + 2 + 3 > 6이므로 조건 ②에 어긋나기 때문이다. 조건 ①~②를 모두 만족하는 경우를 찾기 위해서는 조건 ①을 만족하는 원소를 수열에 하나씩 더해보되 조건 ②에 어긋나면 마지막에 추가한 원소를 제외하고 다른 원소를 더해보는 방식으로 탐색할 수 있다. 이러한 방식에 따라 탐색한다면 [그림 1]과 같은 트리 형태로 solution space를 탐색하게 된다.



[그림 1] 가능성 트리 형태의 solution space 예

이 트리의 탐색은 루트에서 시작한다. 루트는 원소가 없는 빈 수열 []을 나타내는데 아직 선택한 숫자가 없는 상태이기 때문이다. 루트 바로 아래에는 2, 3, 4, 5, 6이 쓰인 간선 5개가 있는데 이들은 수열의 첫 번째 원소로 선택한 숫자들이다. 조건 ①에 따라 2부터 시작해 수열에 포함할 원소를 하나씩 선택해 가되, 조건 ②에 따라 합이 최대 6일 때까지만 선택한다. 이렇게 새로운 원소를 포함해 가는 과정을 [그림 1]과 같이 새로운 가지를 그려 나가는 형태로 표현할 수 있으며, 가지에 쓰인 숫자가 단계마다 포함한 원소를 나타낸다. 예를 들어 트리 가장 왼쪽 붉은 선 3개를 따라 내려간다는 것은 2, 2, 2를 차례로 선택했음을 의미하며, 따라서 수열 [2], [2,2], [2,2,2]를 차례로 탐색했다는 뜻이다. 정점에 있는 수는 루트에서 그 정점까지의 간선에 쓰인 숫자의 합이다. 예를 들어 tree 가장 왼쪽 아래의 붉은색 정점에 쓰인 수는 6인데, 2 + 2 + 2 = 6이기 때문이다. 조건 ②에 어긋나지 않도록 탐색하였으므로 트리의 모든 정점에 쓰인 수는 ≤6이다.

[Q] [그림 1]에서 본 것처럼 자연수(≥2)로 이루어진 수열 중 숫자의 합이≤6인 모든 수열을 탐색한다면 총 12개의 서로 다른 수열을 탐색하게 된다: [2], [2,2], [2,2,2], [2,3], [2,4], [3], [3,2], [3,3], [4], [4,2], [5], [6]. 만약 조건을 조금 바꾸어 자연수(≥2)로 이루어진 수열 중 숫자의 합이≤4인 모든 수열을 탐색한다면 총 몇 개의 서로 다른 수열을 탐색하게 되는가?

2개
4개
7개
12개

※ 이 자료에서 [Q]로 제시된 문제는 학습 후 풀이하는 온라인 퀴즈에 그대로 나오니 학습하면서 그때그때 문제를 풀어 두세요. 온라인 퀴즈에서 보기의 순서는 바뀔 수 있으니 유의하세요. (예: 보기 1이 보기 2가 되고, 보기 2가 보기 1이 될 수 있음)

[Q] 앞 문제에서처럼 자연수(≥2)로 이루어진 수열 중 숫자의 합이≤6인 모든 수열을 탐색한다고 가정하자. 여기에 조건을 하나 추가하여 수열의 각 숫자가 이전 숫자보다 큰 경우만 탐색한다고 하자. 즉 숫자가 증가하는 수열만 탐색하는 것이다.

예를 들어,
수열 [2,3]은 2 < 3이므로 이 조건을 만족하며
[3,2]는 3 > 2이므로 이 조건을 만족하지 않는다.

또한, 하나의 숫자로만 구성된 [2]와 같은 수열도 이 조건을 만족한다고 하자.

총 몇 개의 서로 다른 수열을 탐색하게 되는가?

2개
4개
7개
12개

지금까지 트리 형태의 solution space를 어떻게 탐색하는지 알아보았다. 이제 이 탐색법을 어떻게 코드로 구현할지 생각해 보겠다. 한 가지 방법은 함수의 재귀호출(recursion)을 사용하는 것이다. 아래의 [그림 2]는 [그림 1]의 트리를 재귀호출을 사용해 탐색하는 Python 코드를 보여준다.

※ 이 코드는 이번 시간 수업자료 FindAllSequence.py에 첨부되어 있다.

```
00 def findAllSequenceRecursion(maxSum, min):
01     def recur(currentSum, depth):
02         i = min
03         while currentSum+i <= maxSum:
```

```
04              sequence[depth] = i
05              print(sequence[0:depth+1])
06              recur(currentSum+i, depth+1)
07              i += 1
08
09      sequence = [0 for _ in range(maxSum)]
10      recur(0, 0)
```

[그림 2] 재귀호출을 사용해 가능성 트리 형태의 solution space를 탐색하는 코드

위 코드에서 보여주는 함수 findAllSequenceRecursion(maxSum, min)는 2개의 인자 maxSum과 min을 받아 **① 자연수(≥min)로 이루어진 수열 중 ② 합이≤maxSum인 모든 수열을 탐색**해 출력하는 기능을 한다. 이 함수를 호출하면 라인 09부터 실행되며, 그 이전의 라인 01~07은 이 함수가 내부적으로 호출하는 recur() 함수를 정의한다. 먼저 탐색한 수열을 임시 저장할 리스트인 sequence를 0으로 초기화하고 (라인 09), recur() 함수를 호출한다 (라인 10).

함수 recur(currentSum, depth)는 새로운 숫자를 선택하여 수열 마지막에 추가하는 함수이다. 2개의 인자 currentSum와 depth를 받는데, 각각 지금까지 수열에 추가한 수의 합과 추가한 원소의 갯수를 나타낸다. while 루프를 사용해 여러 가능한 수 i를 추가해 보는데 (라인 03~07) i = min에서 시작하므로 (라인 02) 조건 ①에 따라 숫자를 선택하게 되고 currentSum+i <= maxSum 조건을 만족할 때까지만 루프가 반복되므로 (라인 03) 조건 ②도 만족하는 숫자를 선택하게 된다. 이렇게 조건에 맞게 선택한 수는 sequence[depth]에 저장하고 (라인 04), 그때까지 찾은 수열을 출력한다 (라인 05). 한 숫자를 선택해 수열에 추가했다면 그 뒤에 추가할 숫자를 정하기 위해 같은 함수 recur()를 다시 호출, 즉 재귀 호출(recursion)한다 (라인 06). 함수 recur()는 수열로 선택한 숫자의 합이 ≤maxSum 조건을 만족하는 한 계속해서 재귀 호출되며 수열에 추가할 다음 숫자를 선택하게 되는데, 그러한 과정이 [그림 1]의 가능성 트리를 따라 깊이 내려갈 수 있을 때까지 탐색하는 DFS(Depth First Search)와 유사하다.

findAllSequenceRecursion(6,2)를 실행한 결과는 아래와 같다. 출력 결과를 [그림 1]과 비교해 일치함을 확인해 보자. 또한 DFS로 탐색하는 순서와도 유사함을 확인해 보자.

```
[2]
[2, 2]
[2, 2, 2]
[2, 3]
[2, 4]
[3]
[3, 2]
[3, 3]
[4]
[4, 2]
[5]
[6]
```

findAllSequenceRecursion() 함수 자체를 재귀 호출하지 않고 내부에 recur() 함수를 정의한 후 이를 재귀 호출한 이유는 사용자 편의성 때문이다. 즉, 사용자는 findAllSequenceRecursion() 함수의 인자인 maxSum과 min 값만 입력으로 전달하면 원하는 결과를 얻을 수 있다.

만약 반대로 내부에 recur() 함수를 정의하지 않고 findAllSequenceRecursion() 함수 자체를 재귀 호출하는 방식으로 구현했다면, findAllSequenceRecursion() 함수의 인자로 currentSum, depth, sequence 등도 추가되어야 한다. 그렇게 되면 사용자가 findAllSequenceRecursion() 함수를 호출하기 위해서는 currentSum, depth, sequence 등의 인자가 어떻게 사용되는지 이해하고 직접 초깃값을 입력으로 전달해야 하는데, 이것은 이 함수의 동작 원리를 잘 모르는 사용자에게는 불편하고 어렵다.

정리하면, recur()라는 내부 함수를 정의해 재귀호출에 사용함으로써, 재귀호출에 필요한 변수 초기화는 함수 내부에서 해주고, 함수를 호출하는 외부 사용자는 이해하기 쉬운 두 인자 maxSum과 min만 입력으로 전달하게 함으로써 더 편리하게 함수를 사용할 수 있다.

[Q] [그림 2]에 제시된 함수에 maxSum=4, min=2를 입력으로 주어 호출한다면, 총 몇 개의 서로 다른 수열을 출력하는가?

2개
4개
7개
12개

이제 지금까지 본 문제를 더 큰 maxSum 값에 대해 푼다고 가정하고 (예: maxSum=2000), [그림 2]의 코드를 더 효율적으로 구현하는 방법에 대해 알아보겠다. maxSum이 커질수록 [그림 2]와 같이 재귀호출을 사용한 구현에는 다음과 같은 문제가 발생한다.

① 함수를 연속해 재귀 호출하다 보면 stack overflow 오류(stack으로 사용할 수 있는 메모리 공간을 모두 사용해 더는 stack을 증가시킬 수 없는 경우 발생하는 오류)가 발생한다. 함수를 호출할 때마다 함수 수행에 필요한 정보를 저장할 공간을 메모리의 stack 영역에 마련하며 (예: 지역변수 값, 리턴 주소 등), 함수가 반환할 때 이 공간을 반환한다. maxSum이 클수록 더 깊은 depth까지 가능성 트리를 탐색해 가야 하는데, 이를 위해 함수를 수백~수천 회 연속으로 재귀 호출하다 보면 stack 영역의 메모리를 모두 사용하여 더는 함수를 재귀 호출할 수 없게 된다.

② 함수 호출이 너무 많이, 자주 수행되는 것은 성능에 좋지 않다. 함수를 호출할 때 stack에 함수 수행에 필요한 정보를 쌓았다가, 함수가 반환할 때 이를 삭제하는 과정은 기계어 level에서 보면 상당히 많은 수의 명령을 수행하는 작업이다. 따라서 성능을 개선해야 하는 상황이라면 함수 호출을 너무 많이, 자주 하는 것은 부담이 된다.

이처럼 재귀호출을 많이 했을 때 발생하는 문제를 완화하는 한 가지 방법은 재귀호출을 사용하는 대신 함수 호출과 stack 사용 과정을 직접 코드로 구현하는 것이다. 아래 [그림 3]은 [그림 2]의 코드를 재귀호출을 사용하지 않고 구현한 예를 보여준다.

```
00 def findAllSequenceNoRecursion(maxSum, min):
01     sequence = [0 for _ in range(maxSum)]
02     currentSum = [0 for _ in range(maxSum+1)] # currentSum in the stack
03     i = [0 for _ in range(maxSum+1)] # i in the stack
04     i[0] = min
05     depth = 0
06
07     while True:
08         while currentSum[depth] + i[depth] > maxSum: # 함수 반환 시뮬레이션하는 루프
09             depth -= 1
10             if depth < 0: return # 루트 이전까지 돌아가는 것은 더는 탐색할 수열 없음 의미
11         sequence[depth] = i[depth] # 이번에 선정한 숫자 i를 수열에 추가
12         print(sequence[0:depth+1]) # 선정한 수열 출력
13         currentSum[depth+1] = currentSum[depth] + i[depth]  # 다음 depth의 합 증가
14         i[depth] += 1    # 현재 depth로 반환 시 다음 값 사용하도록 i 값을 미리 증가시켜둠
15         depth += 1       # 다음 depth로 이동
16         i[depth] = min   # 새 depth에서는 i를 초깃값 min부터 시작
```

[그림 3] 재귀호출을 사용하지 않고 가능성 트리 형태의 solution space를 탐색하는 코드

함수 호출과 stack 사용 과정을 시뮬레이션하기 위해 stack 역할을 할 리스트 currentSum[]과 i[]를 사용한다. 이 리스트에는 함수 호출 전에 사용하던 지역변수 currentSum과 i값을 저장해 둠으로써 함수 반환 시 이 값들을 다시 사용할 수 있도록 한다. 특히 currentSum[k]와 i[k]는 stack의 depth k에서 사용하는 변수 currentSum과 i값으로 볼 수 있다.

라인 11~16은 함수 호출 과정을 시뮬레이션한다. 이번에 선정한 숫자 i를 수열 sequence[]에 저장하고 (라인 11), 그때까지 찾은 수열을 출력한다 (라인 12). 숫자 i가 수열에 추가되었으므로 숫자의 합 currentSum을 증가시키고 (라인 13), i를 다음에 사용할 +1된 값으로 미리 변경해둔 후 (라인 14), 다음 depth로 이동한다 (라인 15). 새로운 depth로 진입했으면 i를 min 값으로 초기화하여 이 값부터 선정하도록 한다 (라인 16).

라인 08~10은 함수 반환 과정을 시뮬레이션한다. 새로운 숫자 i를 수열에 추가했을 때 숫자의 합이 maxSum보다 커지면 조건에 어긋나므로 더는 새로운 숫자를 탐색하지 않고 depth를 1 감소시켜 이전 depth로 돌아간다. 하지만 바로 직전 depth로 돌아가더라도 다음 i 값을 선정하면 여전히 합이 maxSum 보다 클 수 있으므로 while 루프를 사용해 합이 ≤maxSum 조건을 만족하는 depth까지 돌아간다.

- 예를 들어 [그림 1]의 트리 가장 왼쪽 아래의 붉은색 정점을 탐색 중이며 (즉, [2,2,2]를 탐색함), maxSum=6, min=2라고 가정하자.
- 여기서 다음 depth로 이동하면 (트리에서 더 아래 가지로 내려감을 의미) 이번에 고려할 수열은 [2,2,2,2]가 되어 합이 6보다 커진다. 따라서 while 루프에서는 depth를 1 감소시켜 붉은색 정점으로 되돌아온다.
- 붉은색 정점에서는 기존에 [2,2,2]를 고려했으므로 이번에는 수열의 마지막 숫자를 1 증가시킨 다음 수열 [2,2,3]을 고려할 차례인데, 합이 6보다 크다. 따라서 while 루프에서는 depth를 다시 1 감소시켜 푸른색 정점으로 돌아온다.
- 푸른 색 정점에서는 기존에 [2,2]를 고려했으므로 이번에는 [2,3]을 고려할 차례인데, 합이 ≤maxSum

조건을 만족한다. 따라서 라인 08~10의 while 루프에서 빠져나와 라인 11~16의 함수 호출 과정으로 들어간다.

[그림 3]의 코드를 한 줄씩 손으로 시뮬레이션해 보며 동작 원리를 이해해 보자. 특히 [그림 2]의 재귀호출을 사용한 코드와 마찬가지로 maxSum=6, min=2라면 다음 순서로 수열을 탐색함을 확인하자: [2], [2,2], [2,2,2], [2,3], [2,4], [3], [3,2], [3,3], [4], [4,2], [5], [6].

[Q] [그림 3]에 제시된 함수에서 라인 16의 "i[depth] = min"을 "i[depth] = i[depth-1]"로 변경하고 maxSum=6, min=2를 인자로 호출하면 어떤 결과를 보게 되는가?

[2], [3]
[2], [2,2], [3], [4]
[2], [2,3], [2,4], [3], [4], [5], [6]
[2], [2,2], [2,2,2], [2,3], [2,4], [3], [3,2], [3,3], [4], [4,2], [5], [6]

지금까지 본 트리탐색 방법은 DFS 순서를 따랐으나, BFS 순서로 탐색하도록 할 수도 있다. 아래 [그림 4]가 그러한 코드를 보여준다. BFS는 queue와 loop을 사용하므로 재귀호출은 사용하지 않는다.

```
00 from queue import Queue
01 def findAllSequenceBFS(maxSum, min):
02     queue = Queue()
03     queue.put((0, None, None)) # (sum, current element, reference to parent tuple)
04     while queue.qsize() > 0:
05         v = queue.get()
06
07         # print to stdout
08         vUp = v
09         while vUp[1] != None:
10             print(vUp[1], end=' ') # print the element
11             vUp = vUp[2] # move to the parent
12         print()
13
14         e = min
15         while v[0]+e <= maxSum:
16             queue.put((v[0]+e, e, v))
17             e += 1
```

[그림 4] BFS 순서로 가능성 트리 형태의 solution space를 탐색하는 코드

BFS는 한 depth에서 탐색한 모든 정점을 queue에 저장한 후, queue에 저장한 정점을 차례로 꺼내며 다음 depth를 탐색하는 것을 반복한다. 이를 위해 먼저 루트를 queue에 넣어 초기화한다. (라인 03) 그리고 queue가 빌 때까지 while loop을 돌며 (라인 04) queue에 저장된 정점 v를 하나씩 꺼내어 (라인 05) v

아래로 탐색할 수 있는 모든 정점을 찾아 queue에 넣는다 (라인 14~17). 라인 07~12는 queue에서 꺼낸 정점을 출력하는 역할을 한다.

Queue에 정점 v를 저장할 때는 아래와 같이 3-tuple로 저장한다.

(지금까지 선택한 원소의 합, 직전에 선택한 원소, v의 parent 정점)

따라서 v[0]가 지금까지 선택한 원소의 합을 나타내고, v[1]은 직전에 선택한 원소를, v[2]는 v의 parent 정점에 대한 reference를 나타낸다. 예를 들어 [그림 1]의 붉은색 정점이 v라면 3-tuple은 (6, 2, 푸른색 정점)이 될 것이며, 푸른색 정점이 v라면 3-tuple은 (4, 2, 초록색 정점)이 된다. 루트가 v라면 3-tuple은 (0, None, None)이 되는데, 직전에 선택한 원소가 없고, parent도 없기 때문이다.

앞에서 본 3-tuple과 같이 각 정점에는 직전에 선택한 원소의 정보만 저장하며, 그전에 선택한 원소의 정보는 parent에 저장한다. 이렇게 하지 않고 정점마다 그때까지 선택한 모든 원소를 저장한다면 많은 저장 공간이 필요할 것이다. 대신 지금까지 선택한 모든 원소를 알고 싶다면 parent를 따라 루트까지 거슬러 올라가면 되는데, 이를 위해 3-tuple의 마지막 값으로 parent에 대한 reference를 저장한다. 라인 07~12는 이처럼 루트에 이를 때까지 (vUp[1] != None) parent를 따라 거슬러 올라가며 (vUp[2]) 지금까지 출력한 원소를 출력한다. 단 마지막에 선택한 원소부터 먼저 출력하므로 “3 2”가 출력되었다면 수열 [2, 3]을 의미하고, “2 3”이 출력되었다면 수열 [3, 2]를 의미함에 유의하자.

위 함수에 maxSum=6, min=2를 인자로 주어 실행하면 다음과 같은 출력을 볼 수 있다. [그림 1]의 트리를 BFS 순서로 탐색했으므로 DFS와 탐색한 순서는 다르지만, 결과적으로 찾아낸 수열의 집합은 12개의 같은 수열이다.

```
2
3
4
5
6
2 2
3 2   # 수열 [2, 3] 의미
4 2   # 수열 [2, 4] 의미
2 3   # 수열 [3, 2] 의미
3 3
2 4   # 수열 [4, 2] 의미
2 2 2
```