완전탐색

(백트래킹, 상태트리와 CUT EDGE)-DFS(깊이우선탐색)

- **"스택"의 개념을 정확히 파악하고 있어야한다.**
- **재귀함수와 스택**
  - 재귀함수
    - 반복문의 효과
    - 다중 반복문을 사용할 때 코드의 유연성이 떨어지는 경우 사용
  - 스택 사용
    - 이때, 함수를 또 호출하면 **현재 위치의 정보(현재까지 실행된 위치)를** 스택의 top에 기록하고 새로 호출된 함수의 정보를 스택에 push함
- **재귀함수를 이용한 이진수 출력**
  - 함수 호출시 함수의 호출 정보들을 **스택**에 기록함
- **이진트리순회(DFS)**
  - **자식노드가 없다면 현재 노드 출력**
  - **전위순회: 부모노드 – 왼쪽 자식노드 – 오른쪽 자식노드**
  - **중위순회: 왼쪽 자식노드 - 부모노드 - 오른쪽 자식노드**
  - **후위순회: 왼쪽 자식노드 – 오른쪽 자식노드 – 부모노드**
  - 왼쪽 자식노드 값: 부모노드 값 x 2
  - 오른쪽 자식노드 값: 부모노드 값 x 2 + 1
- **부분집합구하기**
  - DFS(1)부터 시작 → 이진트리순회의 개념
  - 상태트리 구성
    - DFS의 인자를 **부분집합으로 사용하는 경우**와 **사용하지 않는 경우**로 나눈다.

- ◆ **사용하는 경우는 왼쪽 자식노드**로 **사용하지 않는 경우는 오른쪽 자식노드**로 한다.

- ■ **ch 리스트를 선언하여 부분집합 사용 여부를 파악**한다! → 스스로 생각해야 하는 부분

- 합이 같은 부분집합(DFS)

- ■ "부분집합 구하기" 문제를 응용한 문제
- ■ 서로 나뉜 두 부분집합의 합을 구하고 비교하여 같은 경우 YES를 출력하고 **exit(0) 함수를 통해 프로그램을 강제 종료 시킴**
- ■ 합이 서로 다른 경우(NO)를 확인하려면 모든 부분집합을 구해 보아야 한다. 이는 exit(0)함수가 실행되지 않는 것으로 구분이 가능하고 DFS함수의 return값이 None인 것으로 최종적으로 구분한다.
- ■ 강의 코드 – 전체 집합의 합에서 하나의 부분집합의 합을 빼서 이 값이 하나의 부분집합의 합과 같은지를 판단하여 두 부분집합의 합을 비교한다.
- ■ 시간 복잡도 줄이기 – **하나의 부분집합에서 원소들의 합이 전체 집합의 합의 절반을 넘어가는 경우** 두 부분집합의 합은 절대 같아질 수 없으므로 이런 경우 바로 return 해서 불필요한 연산을 줄인다.

- 잠깐지식(전역변수와 지역변수)

- ■ 전역 변수 – 모든 함수가 접근할 수 있음
- ■ 함수들은 자신 안에 선언된 변수가 **지역변수인지 먼저 확인**하고 아니라면 전역변수인지를 확인한다.
- ■ x = 3 → x라는 **변수가 생성**되고 3을 할당한다.

- 바둑이 승차 – Cut Edge Tech

- ■ 부분집합 구하기 문제 활용
- ■ timeout 에러 발생 → 시간복잡도 문제 해결 필요
  - ◆ 트럭에 태울 수 있는 무게 C보다 큰 경우는 바로 return해버린다. → 당연한 조건!
  - ◆ **ts: 현재까지 부분집합에 적용할지말지 이미 판단한 원소들의 합**
  - ◆ **total – ts: 앞으로 부분집합에 적용할지말지 판단해야할 원소들의 합**

◆ s(부분집합에 적용하기로 한 원소들의 합)에 total – ts를 더했을 때 현재까지 구한 최댓값(result)보다 작다면 **굳이 연산을 진행할 필요가 없으므로** 바로 return 해버린다.

- 중복순열 구하기(DFS)

■ 1 ~ n까지 숫자 중 m개를 중복을 허락하여 뽑는다.

■ 상태트리 구성이 핵심!

■ D(0), D(1), … 이 무엇을 의미하는지 생각해보자

■ D(0): 0번째 뽑았을 때 올 수 있는 숫자

■ D(1): 1번째 뽑았을 때 올 수 있는 숫자

- 동전 교환 – Cut Edge Tech

■ 중복순열 구하기와 같은 유형

■ 상태트리 구성이 핵심

■ D(0), D(1), … 이 무엇을 의미하는지 생각해보자

■ D(0): 동전의 개수가 0개

■ D(1): 동전의 개수가 1개

■ 동전을 뽑으면서 동시에 금액의 합도 같이 계산한다.

- 순열 구하기(DFS)

■ 중복순열과 같은 유형

■ 이미 뽑은 숫자는 다시 뽑으면 안되므로 "**ch리스트**"를 통해 관리한다.

◆ 숫자를 뽑으면 ch[] == 0인 숫자들만 뽑고 뽑았으면 ch[] = 1로 설정한다. 이후, 순열이 완성되어 함수가 종료되면 ch리스트를 다시 이전 상황으로 돌려놓아야 하므로 ch[] = 0으로 설정한다.

- 수열 추측하기(순열, 파스칼 응용)

■ **이항계수 문제**

◆ 이항계수 리스트 만들기 – **시간복잡도를 줄일 수 있음**

◆ 이항계수 계산 – 맨 첫 항(nC0)과 마지막 항(nCn)은 항상 1이다.

● nC0 x (1) + nC1 x (k^1) + nC2 x (k^2) + nC3 x (k^3) + … + nCn x (K^n)

◆ 이항계수 리스트를 만들지 못한다면 일일이 계속 더하여 최종 합을 계산하여야 한다. → 시간복잡도 증가

```
b = [1] * n
for i in range(1, n):
    b[i] = b[i - 1] * (n - i) // i
```

- 조합 구하기(DFS)

■ 중요!!!!! – 많은 문제에 응용됨!!!

■ DFS()의 인자로 s를 넣어준다. – s는 반복문의 시작점을 결정한다.

◆ 초기 s값은 1로 한다.

◆ DFS()가 재귀함수로 호출될 때마다 for문의 i값을 1씩 증가시킨 값 i+1을 DFS() 인자로 넣어준다.

◆ 따라서, DFS()가 재귀함수로 호출될 때마다 반복문의 시작점이 뒤로 밀리게 되므로 앞에서 뽑았던 숫자에는 접근하지 않게 된다.

- 수들의 조합(DFS)

■ 조합 구하기 문제와 같은 유형

■ DFS()의 인자로 s를 넣어줘서 반복문의 시작점을 조절하고 동시에 뽑은 숫자들의 합을 계산하기 위한 변수 Sum을 인자로 넣어준다.

- 라이브러리를 이용한 순열(수열 추측하기)

■ import itertools as it

■ for x in it.permutations(a, k):

◆ a는 주어진 숫자 리스트

◆ k는 a에서 k개 만큼 뽑아서 나열한다는 것을 의미

- 라이브러리를 이용한 조합

- import itertools as it
- for x in it.combinations(a, k):
  - a리스트에서 k개 만큼 뽑는다. – 나열X

- 인접행렬

- 그래프: 노드와 간선의 집합
- 방향그래프: 간선에 방향이 설정되어 있는 그래프
- 가중치 방향그래프: 방향그래프의 간선에 가중치가 있는 그래프
- 인접행렬 만들기 – 모든 원소가 0으로 초기화된 2차원 리스트 선언하기
  - g = [[0] * n for _ in range(n)]

- 경로 탐색(그래프 DFS)

- 인접행렬을 이용
- 방문한 노드는 다시 방문하면 안됨 – 그렇지 않으면, 무한 LOOP를 돌게됨
- 경로 직접 출력하기
  - path 리스트를 선언하고 노드를 방문할 때마다 path에 append한다.
  - 함수가 종료되어 이전 상황으로 돌아오면 앞에서 방문했던 노드를 pop해주어야 한다.