완전탐색&시뮬레이션

week 8 백민희

목차

- 완전탐색 알고리즘 개념
- 브루트포스 기법
- 순열
- 비트마스크
- 시뮬레이션 개념

완전탐색 알고리즘 개념

- **완전탐색**은 간단히 가능한 **모든 경우의 수를 다 체크해서 정답을 찾는 방법**

- **고려사항**
- **1. 사용된 알고리즘이 적절한가? (문제를 해결할 수 있는가)**

- **2. 효율적으로 동작하는가?**
- (대부분 2번때문에 완전탐색 사용에 대한 제한이 따른다.)

- 완전탐색 기법으로 문제 풀기 전에
고려해야 할 사항들

- 1) 해결하고자 하는 문제의 가능한 경우의 수를 대략적으로 계산한다.
  2) 가능한 모든 방법을 다 고려한다.
  3) 실제 답을 구할 수 있는지 적용한다.

가능한 모든 방법을 다 고려한다

① Brute Force 기법
② 순열(Permutation)
③ 재귀 호출
④ 비트마스크 - 2진수 표현 기법을 활용하는 방법
⑤ BFS, DFS를 활용하는 방법

브루트포스 기법

- 이 방법은 반복 / 조건문을 통해 가능한 모든 방법을 단순히 찾는 경우를 의미한다.

- 예를 들어, 위의 자물쇠 암호를 찾는 경우와 같이 모든 경우를 다 참조하는 경우가 그러하다.

순열

- 순열이란 n 개의 값 중에서 r 개의 숫자를 모든 순서대로 뽑는 경우를 말합니다.

- 예를 들어 [1, 2, 3] 이라는 3 개의 배열에서 2 개의 숫자를 뽑는 경우는
- [1, 2]
- [1, 3]
- [2, 1]
- [2, 3]
- [3, 1]
- [3, 2]

이렇게 6 개가 됩니다.

Visited 배열을 이용한 순열

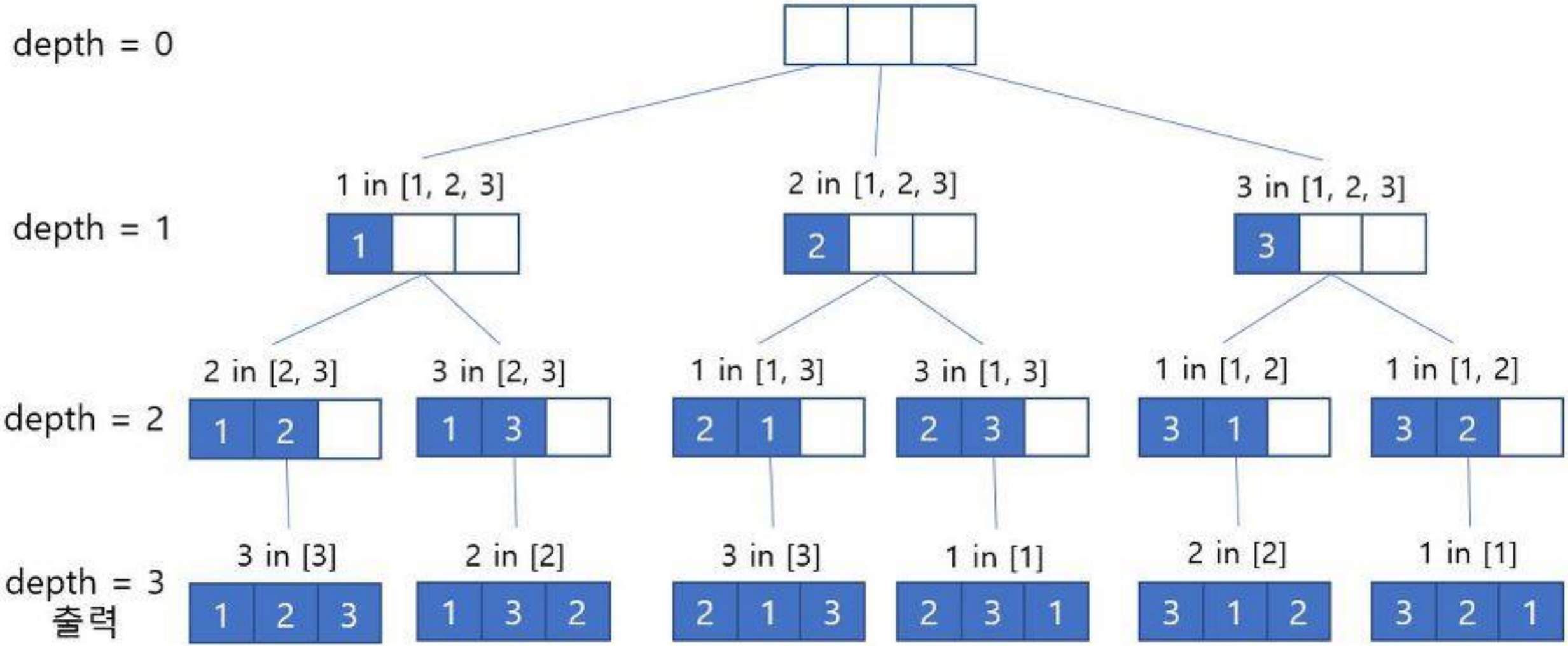
변수	설명
arr	r 개를 뽑기 위한 n 개의 값
output	뽑힌 r 개의 값
visited	중복해서 뽑지 않기 위해 체크하는 값

DFS를 돌면서 모든 인덱스를 방문하여 output 에 값을 넣습니다.

이미 들어간 값은 visited 값을 true 로 바꾸어 중복하여 넣지 않도록 합니다.

depth 값은 output 에 들어간 숫자의 길이라고 생각하시면 됩니다.

depth 의 값이 r 만큼 되면 output 에 들어있는 값을 출력하면 됩니다.


depth = 0
depth = 1
depth = 2
depth = 3
출력
1 in [1, 2, 3]
2 in [1, 2, 3]
3 in [1, 2, 3]
1
2
3
2 in [2, 3]
3 in [2, 3]
1 in [1, 3]
3 in [1, 3]
1 in [1, 2]
1 in [1, 2]
1 2
1 3
2 1
2 3
3 1
3 2
3 in [3]
2 in [2]
3 in [3]
1 in [1]
2 in [2]
1 in [1]
1 2 3
1 3 2
2 1 3
2 3 1
3 1 2
3 2 1

소스 코드

```java
// 순서를 지키면서 n 개중에서 r 개를 뽑는 경우
// 사용 예시: perm(arr, output, visited, 0, n, 3);
static void perm(int[] arr, int[] output, boolean[] visited, int depth,
    if (depth == r) {
        print(output, r);
        return;
    }

    for (int i=0; i<n; i++) {
        if (visited[i] != true) { //방문 안 되어있으면
            visited[i] = true; //방문처리함
            output[depth] = arr[i];
            perm(arr, output, visited, depth + 1, n, r);
            output[depth] = 0; // 이 줄은 없어도 됨
            visited[i] = false;;
            //재귀를 반복햇을 때 중복된 인덱스를 방문하지 않기 위해서 설정함
        }
    }
}
```

비트마스크

- 비트마스크란 비트(bit) 연산을 통해서 부분 집합을 표현하는 방법을 의미

- And 연산(&) : 둘 다 1이면 1
- OR 연산(|) : 둘 중 1개만 1이면 1
- NOT 연산(~) : 1이면 0, 0이면 1
- XOR 연산(^) : 둘의 관계가 다르면 1, 같으면 0
- Shift 연산(<<, >>) : A << B라고 한다면 A를 좌측으로 B 비트만큼 미는 것이다.

A	B	A & B	A | B	~A	A ^ B
0	0	0	0	1	0
0	1	0	1	1	1
1	0	0	1	0	1
1	1	1	1	0	0

- 모든 숫자는 2진수로 표현될 수 있기 때문에 2진수를 통해 비트연산을 수행할 수 있다.
- 13=1101(2)
- 72=1001000(2)
- 그러면 13의 경우에는 부족한 앞의 3자리를 0으로 채우고 비트연산을 수행할 수 있다. 그 결과는 다음과 같다.

```
  0 0 0 1 1 0 1        0 0 0 1 1 0 1        0 0 0 1 1 0 1
& 1 0 0 1 0 0 0      | 1 0 0 1 0 0 0      ^ 1 0 0 1 0 0 0
-------------------  -------------------  -------------------
  0 0 0 1 0 0 0        1 0 0 1 1 0 1        1 0 0 0 1 0 1
```

Shift 연산

- <<, >> 로 표현하며 **해당 방향으로 원 비트를 특정 값만큼 밀어버린다는 개념으로 이해**하면 된다.

- 1 << 3 이라고 한다면 어떨까? 1은 이진수로 1(2) 인데, 이를 3칸 좌측으로 밀어버리므로 1000(2)가 된다.

- >> 는 우측으로 밀어버리는 것인데, 우측으로 밀어버리면 버려지는 것들은 어떻게 할까? 그냥 삭제되는 것이다.
- 예를 들어서 10 >> 2 라고 한다면 10은 이진수로 1010(2)1010(2) 이므로 2칸 밀면 10(2)10(2)가 되는 것이다.

시간복잡도

- **순열 시간복잡도는 O(N x (N-1)!)이라서 O(N!)**
- **비트 연산의 시간복잡도는 내부적으로 상수 시간 정도로 처리가 되어 O(1)**

시뮬레이션(Simulation)

- 일련의 명령에 따라서 개체를 차례대로 이동시키는 것
- 풀이를 떠올리는 것은 쉽지만 소스코드로 옮기기 어려운 문제

- **종류**
- 알고리즘은 간단한데 코드가 지나칠 만큼 길어지는 문제
- 실수 연산을 다루고 특정 소수점 자리까지 출력해야 하는 문제
- 문자열을 특정한 기준에 따라서 끊어 처리해야 하는 문제
- 적절한 라이브러리를 찾아서 사용해야 하는 문제
- **특징**
- 일반적으로 2차원 공간을 다루는 문제가 많이 나온다
- 2차원 공간을 다루기 위해 행렬(Matrix) 개념을 사용
- 이차원 공간에서의 방향 벡터가 자주 나옴