BOJ15649 N과 M(1)

상태	시작 전

언제 DP? 언제 백트래킹?

- **DP**: 최댓값/최솟값/최단거리/경우의 *개수*처럼 "값 하나"가 목표일 때, 그리고 **겹치는 부분문제**가 있을 때.
  - 예) LIS, 배낭문제, 동전으로 만들 수 있는 경우의 수(개수만 출력).
- **백트래킹(DFS)**: **조건을 만족하는 모든 해를 "나열"**해야 할 때. 특히 "중복 없이 골라서 길이 M 수열을 **전부 출력**하라" 같은 문제.
  - 예) 순열/조합/부분집합 전부 출력, N-Queen 해 배치 전부 출력 등.

백트래킹 왜 꼭 필요할까?

1. **모든 순열을 빠짐없이 생성(완전성)**

- `for (x=1..N)`로 현재 단계에서 가능한 **모든 선택지를 하나씩** 시도합니다.
- 재귀로 내려가면 길이 `M`이 될 때마다 한 개의 수열이 완성돼요.

1. **중복 없는 순열 보장(정확성)**

- `if (!used[x])`가 **같은 숫자를 두 번 쓰는 것을 차단**합니다.
- 즉 `path`에는 항상 서로 다른 숫자만 들어가요.

1. **사전순 출력**

- `x=1 → 2 → ... → N` 순서로 시도하므로, 자연스럽게 사전순(작은 수 먼저)으로 결과가 나옵니다.

1. **되돌리기(backtrack)가 없으면? (필수성)**

- `path.pop_back(); used[x]=false;`를 하지 않으면 상태가 누적되어
  - 다음 가지에서 **이미 쓴 숫자가 계속 막히거나**,
  - `path`가 길게 남아 **잘못된(길이 초과/중복) 상태**로 탐색됩니다.
- 백트래킹은 **"한 선택을 시험해 보고, 끝나면 반드시 원상복구"**가 핵심 규칙입니다.

미니 추적 (N=3, M=2)

- 시작: `path=[]`, `used=[F,F,F,F]`
- x=1 선택 → `path=[1]` → 깊이로
  - x=1(사용 중) skip
  - x=2 선택 → `path=[1,2]` (출력) 복귀 → pop/unuse → `path=[1]`
  - x=3 선택 → `path=[1,3]` (출력) 복귀
- 복귀 후 1도 되돌림 → `path=[]`
- x=2 선택… 같은 방식으로 진행 → 결과:
  `1 2, 1 3, 2 1, 2 3, 3 1, 3 2`

불변식(Invariant)

- 호출 진입 시점마다
  - `path.size() == depth`
  - `used== true` ↔ `x ∈ path`
  - `path`는 중복 없음

    이 불변식이 "선택 → 탐색 → 되돌리기"로 항상 유지됩니다.

> 한마디로, 저 블록은 "현재 단계에서 가능한 수를 하나 고르고, 내려가서 모든 경우를 만들고, 다시 깨끗한 상태로 되돌아와 다음 수를 고르는" 백트래킹 프레임 그 자체예요.

이 문제(BOJ 15649)가 백트래킹인 이유

- 요구 사항이 **"길이가 M인 수열을 모두 출력"**입니다.

  → 출력 자체가 최대 P(N,M)=N!(N−M)!P(N, M) = \frac{N!}{(N-M)!}P(N,M)=(N−M)!N! 개로 **지수적**이에요.

  → DP는 "값 하나"를 빠르게 구할 때 빛나지만, **해를 전부 나열**해야 하면 어차피 출력량 만큼의 시간이 필요합니다.

  → 그러니 **결정 트리(1~N 중에서 다음 수를 고르는 선택지)** 를 따라가며, 사용한 수는 제외(visited)하고, 깊이가 M에 도달하면 출력하는 **백트래킹**이 가장 자연스럽습니다.

"DP로 못 하나요?"에 대한 답

- “경우의 **개수**”만 묻는 문제였다면 DP/조합론(P(N,M)P(N,M)P(N,M))으로 한 줄에 끝납니다.
- 하지만 “**모두 출력**”은 DP가 시간을 줄여줄 대상(최적값/개수)이 없습니다.
    - 비트마스크 DP 같은 걸로 “개수를 세는” 건 가능한데, **실제로 모든 수열을 뽑아내는 과정**은 결국 백트래킹과 같은 분기 탐색을 해야 합니다.

백트래킹으로 떠올리는 사고 절차 (암기 팁)

1. 문제에 **“모두 구하라/전부 출력”**이 보이면 → 백트래킹 후보.
2. 선택지가 단계적으로 쌓이는지 생각 → “길이 M, 각 단계에서 1..N 중 하나 선택”
3. 제약이 지역적이면(“중복 없이”) → `used` 배열로 가지치기.
4. **기저 조건**: 길이 M이 되면 출력.
5. 사전순이 필요하면 반복을 1→N 순으로.

한 줄 요약:

> “최적값·경우의 수(숫자 하나)”면 DP, “해 전부 나열”이면 백트래킹.
>
> 이 문제는 전형적인 **순열 나열** → **백트래킹이 정석**입니다.

단위 0. 문제 스펙 정리

- `1..N` 중에서 **중복 없이** 길이 `M`인 **순열** 전부 출력.
- 사전순(작은 수부터 시도)으로 탐색하면 자연스럽게 사전순 출력됨.

단위 1. 상태(State) 설계

- `path`: 현재까지 선택한 수열(스택처럼 사용)
- `used[1..N]`: 숫자 i를 이미 썼는지 여부
- `N, M`: 입력값

➡️ 이 상태만으로 “어디까지 골랐고, 무엇을 썼는지”를 표현합니다.

단위 2. 종료 조건 (Base case)

- `depth == M` 이면 **완성된 수열** ⇒ 출력하고 반환.
- 이게 재귀의 멈춤점이며, 한 개의 해를 기록하는 순간입니다.

단위 3. 후보 생성 & 제약 검사

- `for x in 1..N` : 현재 단계에서 사용할 **후보 숫자**를 순서대로 시도.
- `if (!used[x])` : **중복 금지** 제약을 검사(방문 안 한 숫자만 사용).

단위 4. 선택(Choose)

- `used= true` , `path.push_back(x)`

  현재 경로에 `x` 를 **반영**합니다. (상태 변경 ①, ②)

단위 5. 탐색(Explore)

- `dfs(depth + 1)`

  다음 깊이로 내려가 **남은 자리**를 채웁니다.

단위 6. 되돌리기(Unchoose, Backtrack)

- `path.pop_back()` , `used= false`

  방금 한 선택을 **원상복구**하고, 같은 레벨에서 **다른 후보**를 시도합니다.

  (백트래킹의 핵심: “해보고 → 되돌리기”)

단위 7. 입출력(I/O) 최적화

- 출력이 많으므로 C++에선 `ios::sync_with_stdio(false); cin.tie(nullptr);` 사용.
- 혹은 문자열 버퍼(`ostringstream` / `string`)에 모았다가 한 번에 출력.

전체 C++ 코드 (단위 주석 포함)

```
#include <bits/stdc++.h>
using namespace std;

int N, M;
vector<int> path;        // [단위1] 현재 경로
vector<bool> used;       // [단위1] 사용 여부

void dfs(int depth) {
    // [단위2] 종료 조건: 길이 M 완성
    if (depth == M) {
        for (int i = 0; i < M; ++i) {
            if (i) cout << ' ';
            cout << path[i];
        }
        cout << '\n';
        return;
    }

    // [단위3] 후보 생성: 1..N을 사전순으로 시도
    for (int x = 1; x <= N; ++x) {
        if (!used[x]) {                // 제약: 중복 금지
            // [단위4] 선택
            used[x] = true;
            path.push_back(x);

            // [단위5] 더 깊이 탐색
            dfs(depth + 1);

            // [단위6] 되돌리기
            path.pop_back();
            used[x] = false;
        }
    }
}

int main() {
    ios::sync_with_stdio(false);  // [단위7] I/O 최적화
    cin.tie(nullptr);
```

```
    cin >> N >> M;
    used.assign(N + 1, false);    // [단위1] 상태 초기화
    dfs(0);                       // 탐색 시작
    return 0;
}
```