무식하게 풀기(brute-force)란?

1. 완전 탐색(exhaustive search)이라 부르기도 하며 모든 경우의 수를 다 따져보는 알고리즘이다.
2. 가령, 해, 달, 별을 상징하는 패 세 장을 덮어 놓고 별 - 달 - 해 순으로 카드를 뽑는 순서를 출력하라 하면 간단하게 모든 카드를 하나씩 뒤져보며 그와 같은 순서가 나올 때까지 뒤져 보면 될 것이다.

시간 복잡도 연습

1. 완전 탐색은 시간 복잡도 연습하기에도 좋다. 모든 경우의 수를 다 따지기에 전부 계산을 하면 결국 그 수만큼 시간이 걸리기 때문이다. 또한, 반복문 중첩을 통해 경우의 수를 전부 고려하는 경우가 많아 단순히 반복의 수만으로 대략적인 시간 복잡도 계산이 가능하다.
2. 그렇기에 이번 완전 탐색에서는 문제 풀기와 동시에 시간 복잡도 계산 연습도 동시에 할 것이다.
3. 주먹구구식 연산 방법
    - 알고리즘 대회를 참여하거나 문제를 풀 때, 시간 초과로 답안이 틀렸다고 나온 걸 본 적이 있는가?
    - 위 사항을 겪었다면 그대는 일단, 주먹구구식 연산에 익숙해질 필요가 있다.
    - 대략, 1초당 1억번 연산을 한다 가정하고 계산을 해보자. 그럼 어느새 2초를 넘어가는 내 알고리즘을 볼 수 있을 것이다.

재귀 호출과 완전 탐색

1. 재귀는 큰 수부터 시작해 더는 쪼개지지 않을 때까지 나누고 나눈다. 즉, 반복문이 작은 것부터 시작해 점차 큰 영역을 정복해 나간다면 반대로 재귀는 큰 영역을 쪼개고 쪼개어 마지막의 작은 조각(기저 사례)에 도달할 때까지 나아간다.
2. 보통, 반복문과 달리 재귀는 평소 사람의 머릿속에서 이루어지는 연산 방식과 반대라 머리도 아프고 식 정립도 잘 안되지만, 완전 탐색을 통해 좀 더 쉽게 접할 수 있다. 왜냐하면 모든 경우의 수를 다 따지니까 결국 기저 사례도 쉽게 고안할 수 있기 때문이다.
3. 위 사항을 고려하여 완전 탐색 문항 중에서는 재귀로 풀라는 조건이 있는 문제도 나올 것이다.
4. 재귀는 성능도 반복문보다 떨어지는데 왜 연습하나요?
    - 단순히 괴롭히고 싶어서 그런 건 아니고 재귀를 잘만 이용하면 반복문 쓰는 시간보다 훨씬 코드 작성 시간이 줄어든다. 또한, 복잡한 계산이 필요할 수록 차라리 재귀로 다 쪼개고 컴퓨터가 내부 연산은 알아서 해주는 게 머리가 덜 아플 때가 놀랍게도 있다!
5. 재귀 호출을 이용한 완전 탐색 문제
    - n 개의 원소 중 m 개를 고르는 모든 조합을 찾는 알고리즘

```
// n : 전체 원소의 개수
// picked : 지금까지 고른 원소들의 번호
// toPick : 더 고를 원소의 개수
void pick(int n, vector<int>& picked, int toPick) {
        // 더 고를 원소가 없을 때, 고른 원소들을 출력
        if (toPick == 0) { printPicked(picked); return; }
        // 고를 수 있는 가장 작은 번호를 계산
        int smallest = picked.empty() ? 0 : picked.back + 1;
        // 이 단계에서 원소 하나를 고른다
        for (int next = smallest; next < n; next++) {
                picked.push_back(next);
                pick(n, picked, toPick - 1);
                picked.pop_back();
        }
}
```

- 보글 게임
  - 5×5의 알파벳 격자에 각각 알파벳을 넣고 원하는 문자 조합을 찾아내는 것
  - 조합은 연속된 문자로 이루어져 있으며 각 문자는 격자 간 연결되어 있어야 한다.
  - 왜 완전 탐색인가?
    - 원하는 문자 조합을 찾는다 : 무수한 경우의 수에서 답을 뽑아야 함
    - 연속된 격자로 이루어진다 : 인접칸을 뒤져가며 해당 조합이 있는 지 하나하나 손수 수를 다 따져봐야 함
  - 문제 분할
    - 격자 중에서 문자 조합의 가장 첫번째 문자와 일치하는 문자의 위치를 찾는다.
    - 해당 문자의 인접 격자를 하나씩 뒤지고 만약 두번째 문자와 일치하는 경우 다시 인접 격자를 찾는다.
    - 일치하는 문자가 없으면 다시 전으로 돌아가고 이전 행위를 반복한다.
  - 기저 사례
    - 현재 가리키고 있는 격자의 위치가 격자 범위(5x5) 바깥에 있으면 실패
    - 현재 가리키고 있는 격자의 문자가 원하는 문자가 아닌 경우 실패
    - 더는 검사할 문자가 없으면(문자 조합을 찾은 경우) 성공
  - 위의 내용을 고려하여 코드로 작성하면 아래와 같음

```
const int dx[8] = {-1, -1, -1, 1, 1, 1, 0, 0};
const int dy[8] = {-1, 0, 1, -1, 0, 1, -1, 1};
// 5x5의 보글 게임 판이 해당 위치에서 주어진 단어가 시작하는 지를 반환
bool hasWord(int y, int x, const string& word) {
        // 기저사례 1 : 현재 가리키고 있는 격자의 위치가 격자 범위 바깥에 있으면 실패
        if (!inRange(y, x)) return false;
        // 기저사례 2 : 현재 가리키고 있는 격자의 문자가 원하는 문자가 아니면 실패
        if (board[y][x] != word[0]) return false;
        // 기저사례 3 : 단어 길이가 1이면 성공
        if (word.size() == 1) return true;
        // 인접한 여덟 칸을 검사
        for (int direction = 0; direction < 8; ++direction) {
                int nextY = y + dy[direction];
                int nextX = x + dx[direction];
                // 다음 칸이 범위 안에 있는 지, 첫 글자가 일치하는 지 확인은 불필요
                if (hasWorld(nextY, nextX, word.substr(1))) return true;
        }
        return false;
}
```

- 시간 복잡도 분석
  - 마지막 글자에 도달하기 전에 주변의 모든 격자를 재귀 호출함
  - 각 격자 주변에는 최대 여덟 격자가 존재하고 탐색은 단어의 길이 N에 대해 N-1단계 진행함
  - 결과, 8^(N-1)이고 이를 시간복잡도로 나타내면 O(8^N)으로 수렴

재귀 호출과 부분 문제

1. 재귀 호출은 문제를 나누는 것에서 시작한다.
2. 보글 게임을 예시로 든 부분 문제
   - 현재 위치 (x, y)에서 단어의 첫 글자가 있는가?
   - 윗칸 (y - 1)(x)에서 시작해서 단어의 나머지 글자들을 찾을 수 있는가?
   - 좌측 윗칸 (y - 1)(x - 1)에서 시작해서 단어의 나머지 글자들을 찾을 수 있는가?

- ...
- 위 내용을 보듯이 살펴봐야 하는 격자와 단어 개수가 점진적으로 줄어든다. 이처럼 문제를 분할하여 지속적으로 동일한 계산법으로 계산이 이루어지면 재귀를 사용할 수 있다.

최적화 문제

1. 정의 : 문제의 답이 여러개인 경우, 해당 답안 중에서 가장 좋은 답안을 찾는 것
2. 예시
   - n 개의 사과 중에서 r개를 골라서 무게의 합을 최대화한다
   - n 개의 사과 중에서 r개를 골랐을 때, 개중에서 가장 무게가 큰 사과와 가장 무게가 작은 사과의 무게 차이를 최소화한다
3. 여행하는 외판원 문제(TSP)
   - 문제
     - 어떤 나라에 n(2<= n <= 10) 개의 도시가 있다. 한 영업 사원이 한 도시에서 출발해 다른 도시들을 전부 한 번 씩 방문한 뒤 시작 도시로 돌아오려고 한다.
     - 각 도시들은 전부 도로로 연결되어 있으며 해당 도로들을 이용하여 외판원이 돌 수 있는 가장 짧은 경로를 구하라
     - 연산 시간 : 1초 이내
   - 해결 방법
     - 경우의 수를 계산해보자. 모든 도시를 순서대로 나열하는 방법은 (n-1)!이다. 만약 10개 도시면 9!이며 이는 362,880이다. 어림짐작 연산으로 1초에 1억번이니 충분히 1초 이내로 계산 가능한 수준이다. 그러므로 완전 탐색을 적용해도 문제가 없다.
     - 재귀로 풀 수 있는가?
       - 부분 문제 : 현재 도시에서 다음 도시로 가면 방문해야 할 도시수가 줄어들고 줄어든 도시에서 동일한 계산으로 다음 도시를 찾아냄. 결국, 부분 문제가 성립하고 동일 계산식을 사용하므로 재귀가 가능함
     - 기저 사례
       - 모든 도시를 방문했을 때 시작 도시로 돌아가고 종료한다

```
int n; // 도시의 수
double dist[MAX][MAX]; // 두 도시 간의 거리를 저장하는 배열
// path : 지금까지 만든 경로
// visited  : 각 도시의 방문 여부
// currentLength : 지금까지 만든 경로의 길이
```

```
// 나머지 도시들을 모두 방문하는 경로들 중 가장 짧은 것의 길이를 반환한다
double shortestPath(vector<int>& path, vector<bool>& vistied, double
currentLength) {
        // 기저 사례 : 모든 도시들을 다 방문했을 때, 시작 도시로 돌아가고 종료함
        if (path.size() == n)
                return currentLength + dist[path[0]][path.back()];
        double ret = INF; // 매우 큰 값으로 초기화
        // 다음 방문할 도시들을 전부 시도해본다
        for (int next = 0; next < n; ++next) {
                if (vistied[next]) continue;
                int here = path.back();
                path.push_back(next);
                // 나머지 경로를 재귀 호출을 통해 완성하고 가장 짧은 경로의 길이를 얻는다
                double cand = shortestPath(path, visited, currentLength +
                dist[here][next]);
                ret = min(ret, cand);
                visited[next] = false;
                path.pop_back();
        }
        return ret;
}
```

그래서 언제 완전탐색을 쓰면 좋은데요?

1. 지수적으로 시간이 증가하더라도 절대적으로 구해야하는 후보의 가짓수가 적은 경우
    - 보글 게임을 예로 들면, 단어 길이가 적고 격자 개수가 적으면 결국 필요한 경우의 수가 줄어든다. 요구하는 시간과 메모리 확보에 문제가 없으면 굳이 머리를 쓰면서 더 효율적인 알고리즘 구상에 시간을 소모하느니 완전탐색으로 빨리 끝내고 다음 일로 넘어가는 게 효율적이다.
2. 효율적이 알고리즘이 떠오르지 않고 시간은 촉박한 경우
    - 시간이 부족한데 알고리즘을 떠올릴 시간이 어디 있는가? 내 인내심은 괜찮을 지 몰라도 상사의 인내심이 과연 기다려줄 지 알 수 없을 때는 일단 가장 간단한 완전탐색으로 해결하는 걸 시도하자. 그 다음에 만족하지 못하는 수치에 도달했을 때, 알고리즘을 다른 걸로 구성하는 게 적어도 내 정신건강에는 좋다.

3. 처음 보는 유형의 문제를 만났을 때
    - 일단, 완전탐색을 통해 풀어보고 거기서 생각을 확장하여 완전정복이 가능하니 DP나 그리디 등. 타 알고리즘을 떠올리는 게 문제 해결을 위한 빠른 출구가 될 수 있다.
    - 또한, 기계적인 코딩 테스트가 아닌 기술 면접에서 간단하게 낸 문제인 경우, 일단 완전탐색으로라도 시도해 보는 게 좋은 점수를 얻을 수 있다. 적어도 손 놓고 멍 때리다가 못하겠다고 던지는 것보다는 백 배 낫다.