# 완전 탐색과
# 백트래킹

경북대학교 전현승
dogdriip@gmail.com

# Table of contents


- 0x00 완전 탐색 : 그냥 다 해보기
- 0x01 백트래킹?

## 0x00

# 완전 탐색 : 그냥 다 해보기

# 완전 탐색

- **완전 탐색 (Brute-force)** : 모든 경우의 수를 다 시도해보는 것!

- **이론적으로 가능한 모든 경우의 수를 일일이 다 탐색해보는 것**
- 따라서 실패할 일이 없다.
- 당연히 하나씩 다 대입해보면서 풀면 **정확도 100%**

- 그렇다면 완탐은 최강의 문제풀이 방법인가?

# 완전 탐색

- 그렇다면 완탐은 최강의 문제풀이 방법인가?
- 꼭 그렇지만은 않다

- 굉장히 쉬운 방법임과 동시에 **시간 복잡도는 최악**
  - 모든 경우를 다 시도해보는 방법이므로 당연히…
- 문제의 제한을 잘 보고, 시간 복잡도 분석을 했을 때,
  제한시간 내에 돌아갈 정도의 시간 복잡도라면 시도해볼만 한 방법

- Tip : 모든 문제풀이의 기초 방법
  - 문제를 마주했을 때, *'모든 경우의 수를 다 따져볼 수 있는가?'*를 먼저 생각해보면 좋다

# 일곱 난쟁이



- 아홉 난쟁이의 키가 주어졌을 때, 합이 100이 되는 일곱 난쟁이 찾기
- → 9개의 자연수가 주어졌을 때, 합이 100이 되는 7개의 자연수 찾기

- *'모든 경우의 수를 다 따져볼 수 있는가?'*

# 일곱 난쟁이

## #2309 – 일곱 난쟁이

- 9개 중 순서에 상관없이 7개 뽑는 경우의 수 : $\binom{9}{7} = 36 = O(1)$
- 그냥 9명 중에서 7명 고르는 경우의 수를 다 해 보면 된다!

- 구현은 7중 반복문 또는 재귀함수 등으로
- 9명 중에서 고르지 않을 2명을 선택하는 식으로 구현하면 2중 반복문

# 일곱 난쟁이 – 반복문 구현

#2309 – 일곱 난쟁이

- 2중 반복문

```
sort(h, h + 9);

for (int i = 0; i < 9; i++) {
    for (int j = i + 1; j < 9; j++) {
        if (sum - h[i] - h[j] == 100) {
            // 전체 합에서 두 개 뺐더니 100이면 답을 찾은 것
            // 나중에 출력할 때 두 개만 빼면 된다
            print[i] = print[j] = false;
        }
    }
}
```

# 일곱 난쟁이 – 재귀함수 구현



- 재귀함수 구현 – main() 부분

```
vector<int> h(9);
vector<int> picked;
bool visited[9];
int sum;

int main() {
    for (int i = 0; i < 9; i++) {
        cin >> h[i];
    }

    solution(7);

    return 0;
}
```

# 일곱 난쟁이 – 재귀함수 구현

#2309 – 일곱 난쟁이

- solution(int to_pick) : 고를 게 to_pick개 남은 상태에서 앞에서부터 선택

```
void solution(int to_pick) {
    // [여기에 기저 케이스 부분 입력]

    for (int i = 0; i < 9; i++) {
        if (!visited[i]) {
            picked.push_back(num[i]);
            visited[i] = true;
            sum += num[i];
            solution(to_pick - 1);
            picked.pop_back();
            visited[i] = false;
            sum -= num[i];
        }
    }
}
```

# 일곱 난쟁이 – 재귀함수 구현

## #2309 – 일곱 난쟁이

- 아직 고르지 않았다면, picked에 넣어주고, 골랐다고 표시해주고, sum도 그때그때 관리

```
void solution(int to_pick) {
    // [여기에 기저 케이스 부분 입력]

    for (int i = 0; i < 9; i++) {
        if (!visited[i]) {
            picked.push_back(num[i]);
            visited[i] = true;
            sum += num[i];
            solution(to_pick - 1);
            picked.pop_back();
            visited[i] = false;
            sum -= num[i];
        }
    }
}
```

# 일곱 난쟁이 – 재귀함수 구현

#2309 – 일곱 난쟁이

- 골라야 할 개수를 하나 줄여서 다시 재귀

```
void solution(int to_pick) {
    // [여기에 기저 케이스 부분 입력]

    for (int i = 0; i < 9; i++) {
        if (!visited[i]) {
            picked.push_back(num[i]);
            visited[i] = true;
            sum += num[i];
            solution(to_pick - 1);
            picked.pop_back();
            visited[i] = false;
            sum -= num[i];
        }
    }
}
```

# 일곱 난쟁이 – 재귀함수 구현

#2309 – 일곱 난쟁이

- 재귀가 끝나고 돌아왔다면 다시 원래대로 돌려주는 작업이 필요

```
void solution(int to_pick) {
    // [여기에 기저 케이스 부분 입력]

    for (int i = 0; i < 9; i++) {
        if (!visited[i]) {
            picked.push_back(num[i]);
            visited[i] = true;
            sum += num[i];
            solution(to_pick - 1);
            picked.pop_back();
            visited[i] = false;
            sum -= num[i];
        }
    }
}
```

# 일곱 난쟁이 – 재귀함수 구현



- 재귀함수의 **기저 케이스 (Base case)** : 재귀함수가 더 깊게 들어갈 수 없는, 최소 케이스

```
void solution(int to_pick) {
    if (to_pick == 0) {
        if (sum == 100) {
            sort(picked.begin(), picked.end());
            for (int it : picked) {
                cout << it << '\n';
            }
            exit(0);
        } else {
            return;
        }
    }

    // ...
}
```

# 일곱 난쟁이 – 재귀함수 구현

#2309 – 일곱 난쟁이

- 더 이상 고를 게 없는 경우 (7개를 모두 고른 경우)가 기저 케이스가 된다

```
Void solution(int to_pick) {
    if (to_pick == 0) {
        if (sum == 100) {
            sort(picked.begin(), picked.end());
            for (int it : picked) {
                cout << it << '\n';
            }
            exit(0);
        } else {
            return;
        }
    }

    // ...
}
```

# 일곱 난쟁이 – 재귀함수 구현

## #2309 – 일곱 난쟁이

- 합이 100이 된다면 골랐던 것들을 정렬해서 출력 후 종료, 아니면 그냥 리턴

```
void solution(int to_pick) {
    if (to_pick == 0) {
        if (sum == 100) {
            sort(picked.begin(), picked.end());
            for (int it : picked) {
                cout << it << '\n';
            }
            exit(0);
        } else {
            return;
        }
    }

    // ...
}
```

# 치킨치킨치킨

## #16439 – 치킨치킨치킨

- 고리 회원 N명이 있고, 치킨 종류는 M종류가 있다.
- 회원마다 각 치킨의 선호도가 있고,
  한 사람의 만족도는 시킨 치킨들 중 선호도가 가장 큰 값으로 결정된다.
- 치킨을 최대 세 가지 종류만 시키려 한다. 회원들의 만족도 합의 최대는?

# 치킨치킨치킨



- 예를 들어 N = 4, M = 6이면, 입력 형식은 다음과 같이 주어진다

1번 사람의 3번 치킨에 대한 선호도

| | | |
|---|---|---|
| 1번 사람 | → | 1 2 3 4 5 6 |
| 2번 사람 | → | 6 5 4 3 2 1 |
| 3번 사람 | → | 3 2 7 9 2 5 |
| 4번 사람 | → | 4 5 6 3 2 1 |

4번 사람의 5번 치킨에 대한 선호도

# 치킨치킨치킨

## #16439 – 치킨치킨치킨

- 치킨을 선택하면, 각 사람의 만족도는 각 행 중 최댓값으로 결정된다

| | | | | | | | | |
|---|---|---|---|---|---|---|---|---|
| 1번 사람 → | 1 | 2 | 3 | 4 | 5 | 6 | → 만족도 | 4 |
| 2번 사람 → | 6 | 5 | 4 | 3 | 2 | 1 | → 만족도 | 6 |
| 3번 사람 → | 3 | 2 | 7 | 9 | 2 | 5 | → 만족도 | 9 |
| 4번 사람 → | 4 | 5 | 6 | 3 | 2 | 1 | → 만족도 | 5 |

= 만족도 합 24

# 치킨치킨치킨



- 어떻게 풀까? 하고 문제 제한을 다시 봤더니
- 고리 회원의 수 $N$ $(1 \le N \le 30)$, 치킨 종류의 수 $M$ $(3 \le M \le 30)$
- *'모든 경우의 수를 다 따져볼 수 있는가?'*

- 치킨은 무조건 3종류 시키는 게 이득이므로
- 치킨 M종류 중 3종류 선택 : $\binom{M}{3} = O(M^3)$
- 선택한 치킨들에 대해 모든 회원들의 만족도 계산 : $O(N)$
- 넉넉하게 모든 경우의 수를 다 돌려볼 수 있다

# 치킨치킨치킨

#16439 – 치킨치킨치킨

- 뭐 이런 식으로, 모든 치킨에 대해 3개씩 골라보면 된다

| 1 | 2 | 3 | 4 | 5 | 6 |
|---|---|---|---|---|---|
| 6 | 5 | 4 | 3 | 2 | 1 |
| 3 | 2 | 7 | 9 | 2 | 5 |
| 4 | 5 | 6 | 3 | 2 | 1 |

| 1 | 2 | 3 | 4 | 5 | 6 |
|---|---|---|---|---|---|
| 6 | 5 | 4 | 3 | 2 | 1 |
| 3 | 2 | 7 | 9 | 2 | 5 |
| 4 | 5 | 6 | 3 | 2 | 1 |

| 1 | 2 | 3 | 4 | 5 | 6 |
|---|---|---|---|---|---|
| 6 | 5 | 4 | 3 | 2 | 1 |
| 3 | 2 | 7 | 9 | 2 | 5 |
| 4 | 5 | 6 | 3 | 2 | 1 |

| 1 | 2 | 3 | 4 | 5 | 6 |
|---|---|---|---|---|---|
| 6 | 5 | 4 | 3 | 2 | 1 |
| 3 | 2 | 7 | 9 | 2 | 5 |
| 4 | 5 | 6 | 3 | 2 | 1 |

| 1 | 2 | 3 | 4 | 5 | 6 |
|---|---|---|---|---|---|
| 6 | 5 | 4 | 3 | 2 | 1 |
| 3 | 2 | 7 | 9 | 2 | 5 |
| 4 | 5 | 6 | 3 | 2 | 1 |

| 1 | 2 | 3 | 4 | 5 | 6 |
|---|---|---|---|---|---|
| 6 | 5 | 4 | 3 | 2 | 1 |
| 3 | 2 | 7 | 9 | 2 | 5 |
| 4 | 5 | 6 | 3 | 2 | 1 |

| 1 | 2 | 3 | 4 | 5 | 6 |
|---|---|---|---|---|---|
| 6 | 5 | 4 | 3 | 2 | 1 |
| 3 | 2 | 7 | 9 | 2 | 5 |
| 4 | 5 | 6 | 3 | 2 | 1 |

. . .

# 치킨치킨치킨 – 구현

#16439 – 치킨치킨치킨

- 입력받고 재귀함수 bf() 호출

```
int n, m, ma[31][31], max_sum;
Vector<int> picked;

int main() {
    cin >> n >> m;
    for (int i = 0; i < n; i++) {
        for (int j = 0; j < m; j++) {
            cin >> ma[i][j];
        }
    }

    bf(); cout << max_sum << '\n';

    return 0;
}
```

# 치킨치킨치킨 – 구현

#16439 – 치킨치킨치킨

- 재귀함수에서 고르고, 재귀 돌리고, 다시 골랐던 걸 빼는 부분

```
void bf() {
    // 기저 케이스 부분

    for (int i = (picked.empty() ? 0 : picked.back() + 1); i < m; i++) {
        picked.push_back(i);
        bf();
        picked.pop_back();
    }
}
```

# 치킨치킨치킨 – 구현

#16439 – 치킨치킨치킨

- 지금까지 고른 게 없으면 인덱스 0부터, 골랐던 게 있으면 맨 마지막 원소 다음 원소부터

```
void bf() {
    // 기저 케이스 부분

    for (int i = (picked.empty() ? 0 : picked.back() + 1); i < m; i++) {
        picked.push_back(i);
        bf();
        picked.pop_back();
    }
}
```

# 치킨치킨치킨 – 구현

## #16439 – 치킨치킨치킨

- 기저 케이스 : 고른 개수가 3이면, 즉 다 골랐으면, 만족도 합을 구해서 최댓값 갱신

```
void bf() {
    if (picked.size() == 3) {
        int sum = 0;
        for (int i = 0; i < n; i++) {
            sum += max({ma[i][picked[0]], ma[i][picked[1]], ma[i][picked[2]]});
        }

        max_sum = max(max_sum, sum);
        return;
    }

    // ...
}
```

# 치킨치킨치킨 – 구현

#16439 – 치킨치킨치킨

- 지금 구현한 방식처럼, 재귀함수에 인자를 넣지 않는 식으로 구현해도 된다

```
void bf() {
    if (picked.size() == 3) {
        int sum = 0;
        for (int i = 0; i < n; i++) {
            sum += max({ma[i][picked[0]], ma[i][picked[1]], ma[i][picked[2]]});
        }

        max_sum = max(max_sum, sum);
        return;
    }

    // ...
}
```

## 0x01

# 백트래킹?

# 백트래킹?

- **백트래킹 (Backtracking, 퇴각 검색)** : 음… 완전 탐색의 한 종류라고 생각하면 된다.
- 일반적인 완전 탐색처럼 **무식하게 모든 경우의 수를 찾는 것이 아니라,**
  **가지치기**라는 개념을 도입한 완전 탐색 (정도로 생각하면 될 듯)

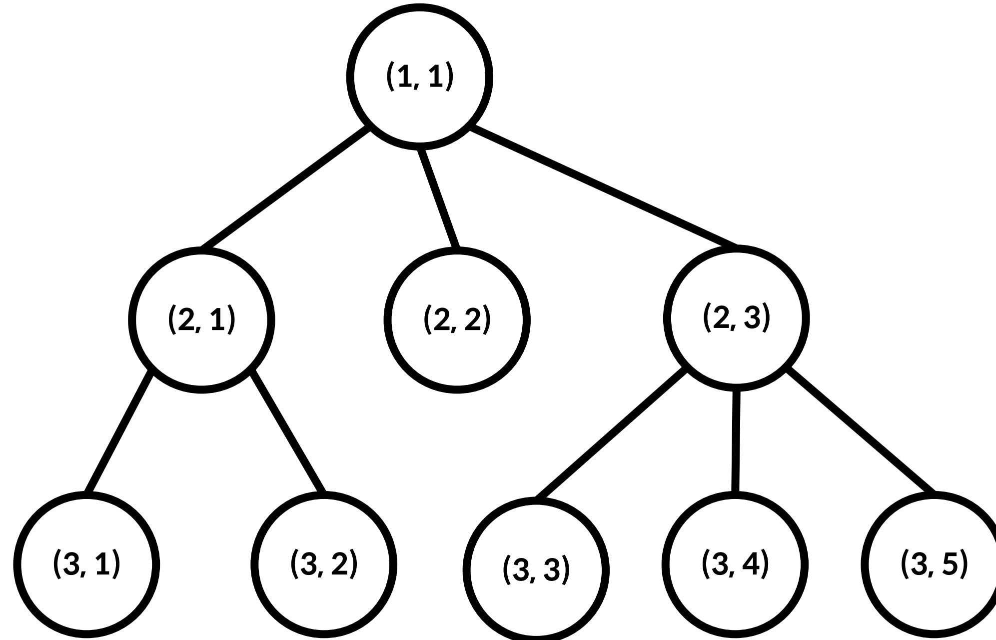
- 가지치기?

# 백트래킹의 핵심 : 가지치기

- **가지치기 (Pruning)** : 백트래킹의 핵심
- 모든 경우를 탐색해보지 않고,
  **가망이 없는 (== 유망하지 않은) 경우를 만나면**
  **더 탐색하지 않고 바로 윗 단계로 올라간다.**
- 탐색하다가 가망이 없어 보인다면
  더 이상 깊게 들어가지 않고,
  바로 탐색을 종료하고 위로 올라가서 다른 경우를 탐색한다.

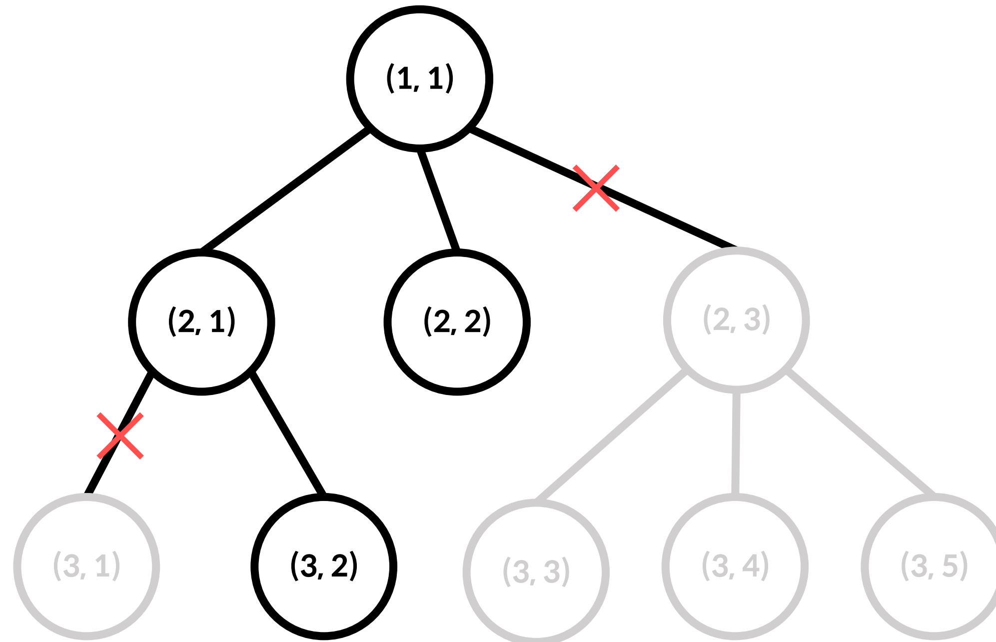
- '유망하지 않은 **경우를 배제**함으로써 **풀이 시간 단축**'

# 백트래킹의 핵심 : 가지치기




- 완전 탐색에서 가능한 상태를 모두 트리로 나타냈을 때 이러한 상태가 된다고 하면

# 백트래킹의 핵심 : 가지치기




- 아예 가망이 없어서 더 탐색해봐야 손해인 노드는 더 탐색하지 않는 것

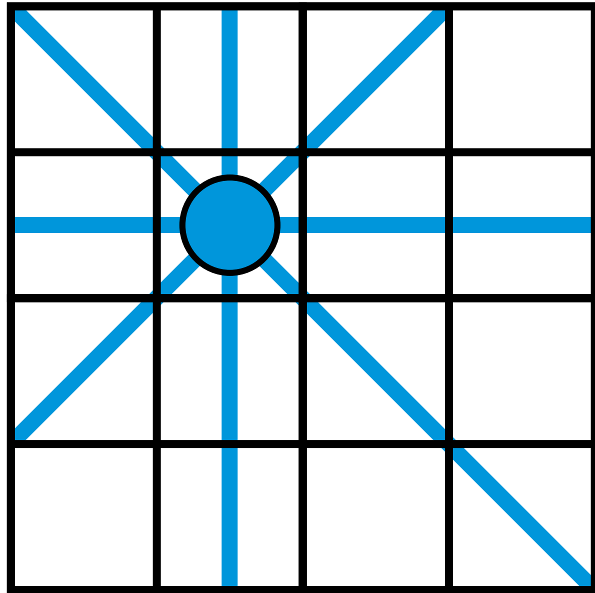
# 백트래킹 대표 문제 – N-Queen

#9663 N-Queen

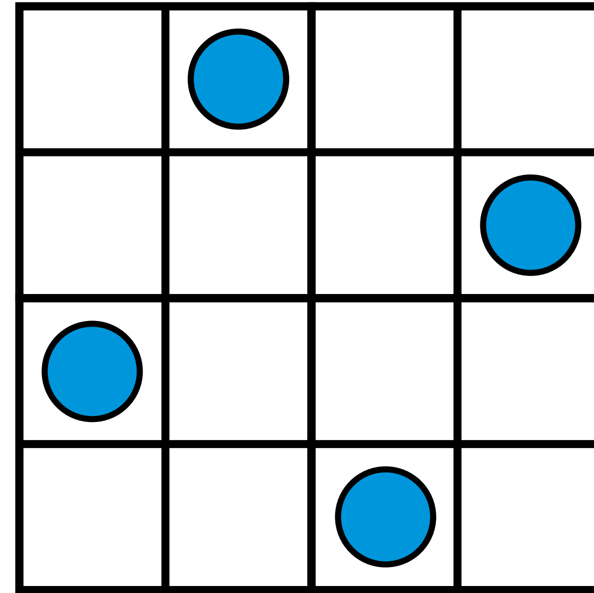
- 대표적인 백트래킹 문제
- N-Queen Problem : N×N 크기의 체스판에 퀸 N개를 서로 공격할 수 없게 놓는 것.
- N이 주어지면, 이러한 경우의 수를 구하여라.

# 백트래킹 대표 문제 – N-Queen

**#9663 N-Queen**



퀸의 공격 범위 예시



조건에 맞게 4×4 체스판에
4개의 퀸을 배치한 경우 중 하나

# 백트래킹 대표 문제 – N-Queen

## #9663 N-Queen

- 어떻게 풀 수 있을까?
- 우선, *'모든 경우의 수를 다 따져볼 수 있는가?'*
- 그냥 아무렇게나 N개의 퀸을 놓아보고, 되는지 판단하고, … 이를 반복하면?

- 문제의 제한 : $1 \leq N \leq 15$

# 백트래킹 대표 문제 – N-Queen

## #9663 N-Queen

- 처음에 빈 N×N 체스판 위에 놓을 수 있는 퀸의 위치 수 : $N^2$
- 그 다음에 놓을 수 있는 퀸의 위치 수 : $N^2 - 1$
- 그 다음 : $N^2 - 2$
- …
- 총 N개의 퀸을 놓아보는 모든 경우의 수 ≈ $\left(N^2\right)^N = N^{2N}$

- 이렇게 모든 경우를 다 따져보면 $O(N^{2N})$
- N이 최대 15까지 입력될 수 있기 때문에 이 방법으로는 불가능하다.

# 백트래킹 대표 문제 – N-Queen

#9663 N-Queen

- 저렇게 무식하게 하지 말고, 경우의 수를 조금씩 줄여보자.

- 일단 한 행에 퀸은 하나밖에 올 수 없음이 자명하다.
- 각 행에 퀸을 하나씩만 놓아보면 어떨까?

# 백트래킹 대표 문제 – N-Queen

#9663 N-Queen

- 첫 행에 놓을 수 있는 퀸의 위치 수 : $N$
- 둘째 행에 놓을 수 있는 퀸의 위치 수 : $N$
- $\cdots$
- N개의 행에 각각 하나씩, 총 N개의 퀸을 놓는 경우의 수 : $N^N$

- 이것도 안 된다.

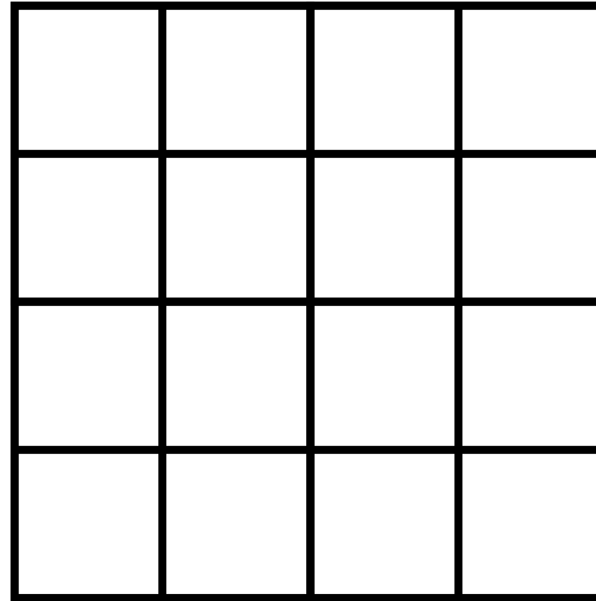
# 백트래킹 대표 문제 – N-Queen

#9663 N-Queen

- 여기서 가지치기를 도입해 보자!
- 완전 탐색을 하되, **가망이 있는 경우인지 판단하면서 진행하다가, 가망이 없는 경우 바로 탐색을 종료**하는 식으로 진행하자.
- 즉, 중간에 어딘가 잘못 놓은 곳이 있어서,
  아무리 다른 곳에 퀸을 놓아도 정답이 되지 않을 경우,
  그냥 상위 단계로 올라와서 아까 놨던 퀸을 다시 놓아보는 것.

# 백트래킹 대표 문제 – N-Queen

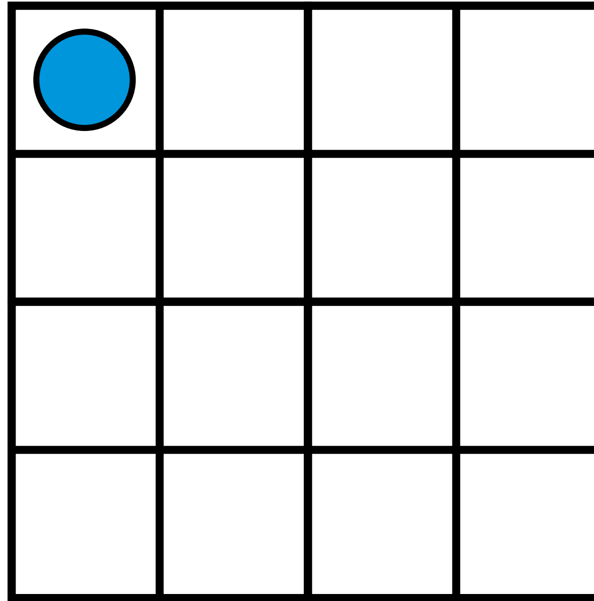
#9663 N-Queen



- 초기 체스판 상태

# 백트래킹 대표 문제 – N-Queen

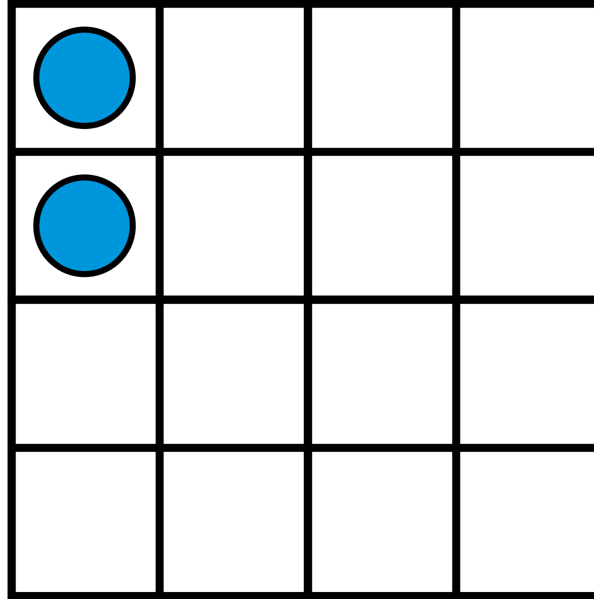
#9663 N-Queen



- 일단 1행에서, 1열에 한번 놓아보자

# 백트래킹 대표 문제 – N-Queen

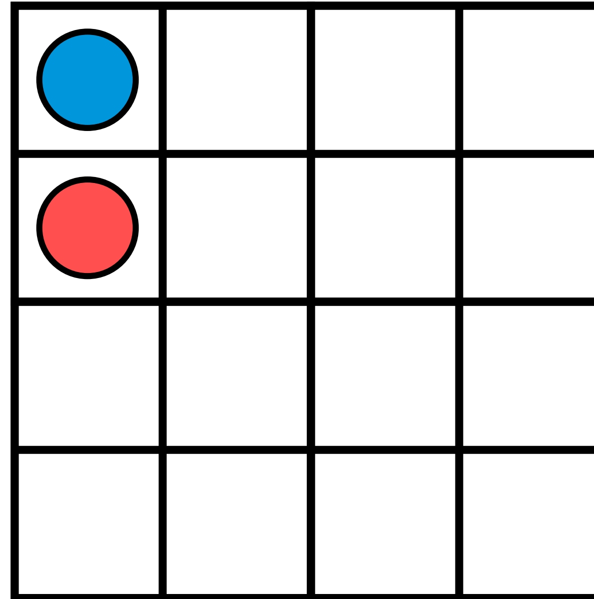
#9663 N-Queen



- 그 다음 2행으로 가서, 1열에 놓아보면?

# 백트래킹 대표 문제 – N-Queen

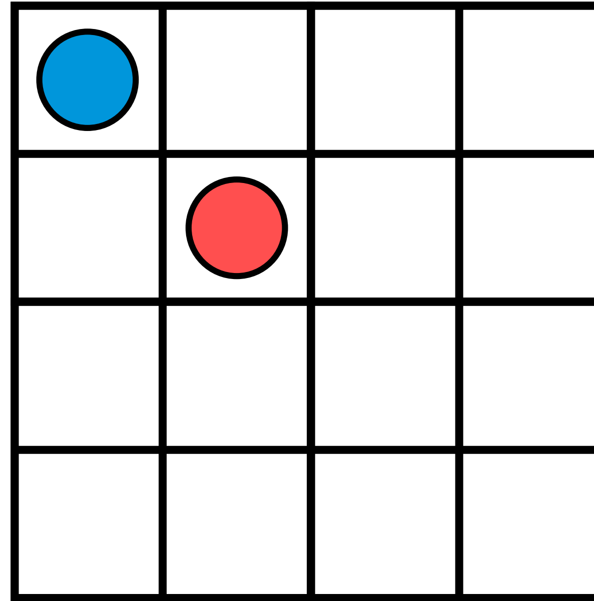
#9663 N-Queen



- 애초에 조건에 위배되므로, 남은 두 행을 어떻게 채우든 이 상태라면 정답이 나올 수 없다.
- 더 탐색해봐도 가망이 없는 경우이다

# 백트래킹 대표 문제 – N-Queen

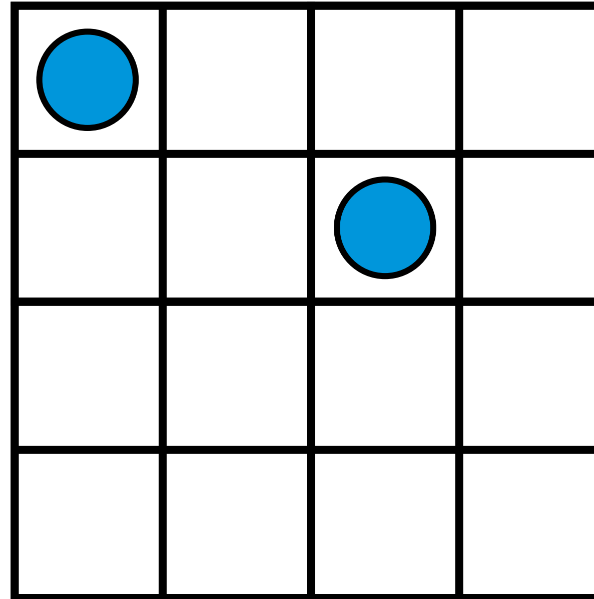
#9663 N-Queen



- 2행에 놓았던 퀸을 빼고, 이번에는 2열에 놓아보자.
- 근데 이래도 안 된다

# 백트래킹 대표 문제 – N-Queen

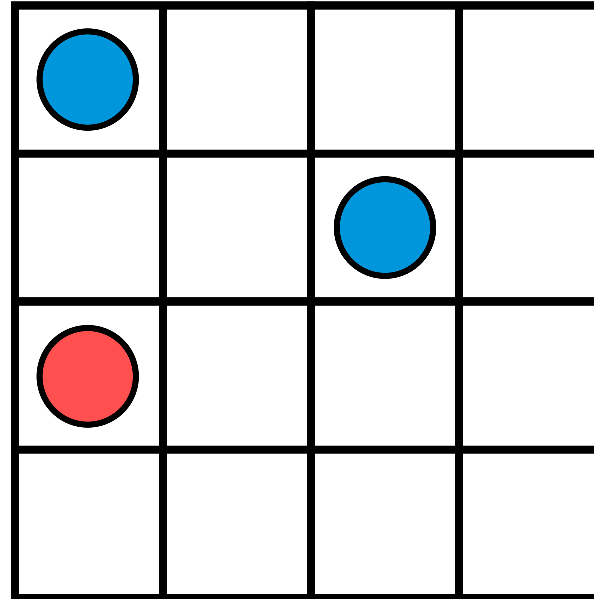
#9663 N-Queen



- 다시 빼고, 3열에 놓아보자
- 이번에는 된다.

# 백트래킹 대표 문제 – N-Queen

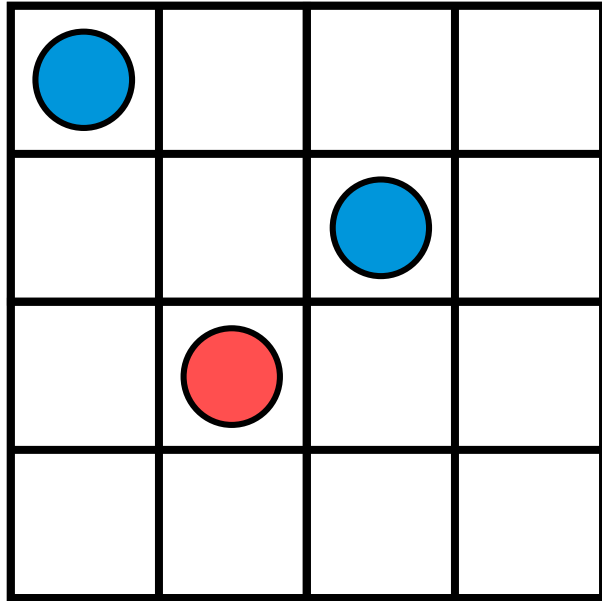
#9663 N-Queen

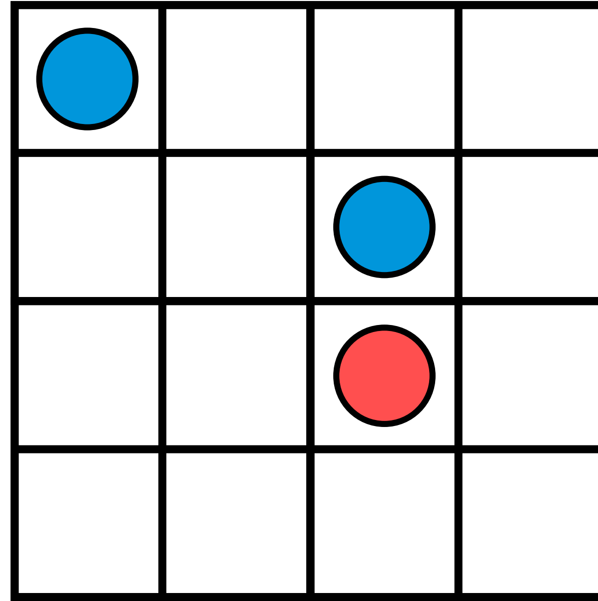


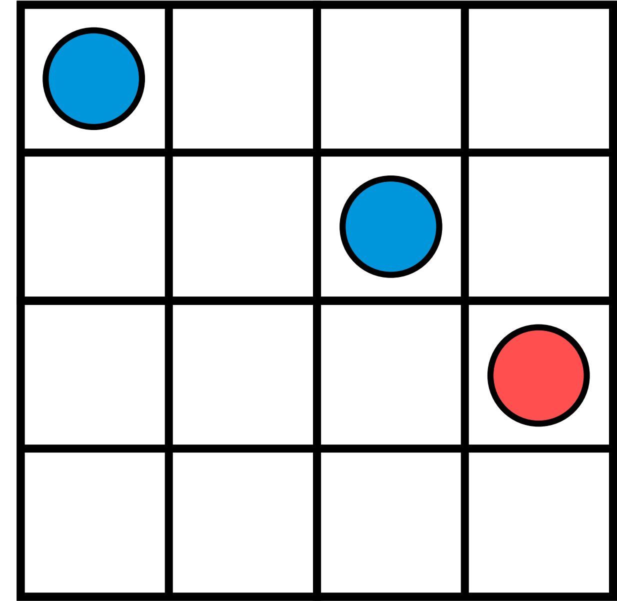
- 3행으로 가서, 또 1열부터 놓아보자.
- 일단 1열은 안 된다

# 백트래킹 대표 문제 – N-Queen

#9663 N-Queen



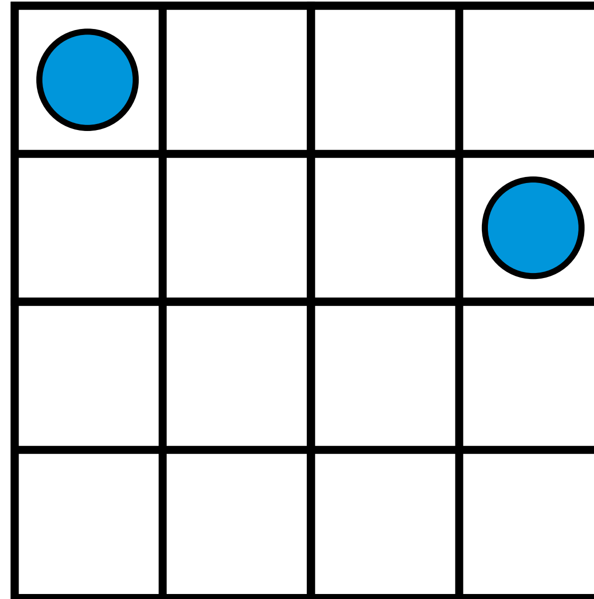




- 2, 3, 4열 모두 놓을 수 없다.
- 더 이상 조건을 만족하면서 진행할 수 없는 경우이다. 상위 레벨로 되돌아가자.

# 백트래킹 대표 문제 – N-Queen

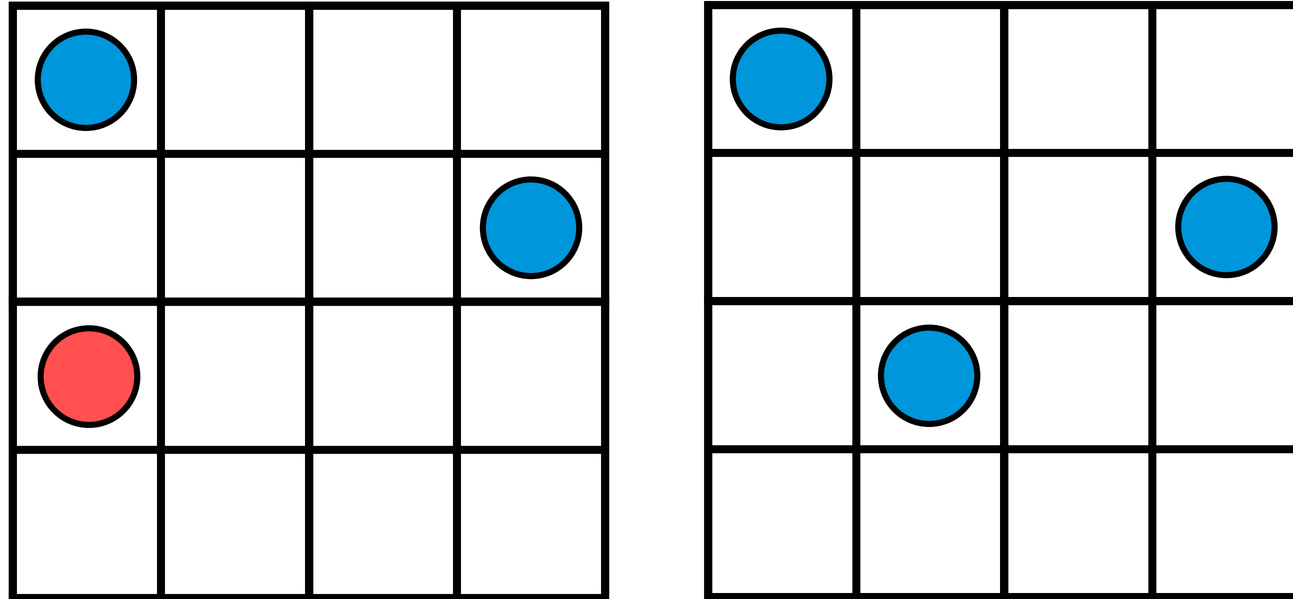
#9663 N-Queen



- 결국 2행의 퀸이 4열까지 왔다
- 3행을 또 채워보자

# 백트래킹 대표 문제 – N-Queen

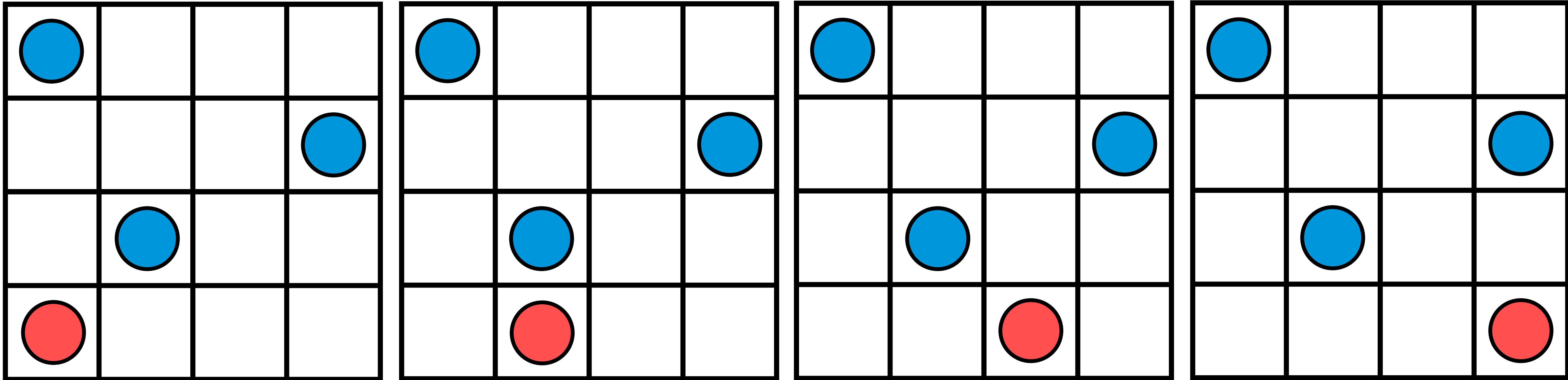
#9663 N-Queen



- 3행의 1열은 안 되고, 2열은 된다.
- 2열에 놓아보고 마지막 4행으로 넘어가자

# 백트래킹 대표 문제 – N-Queen

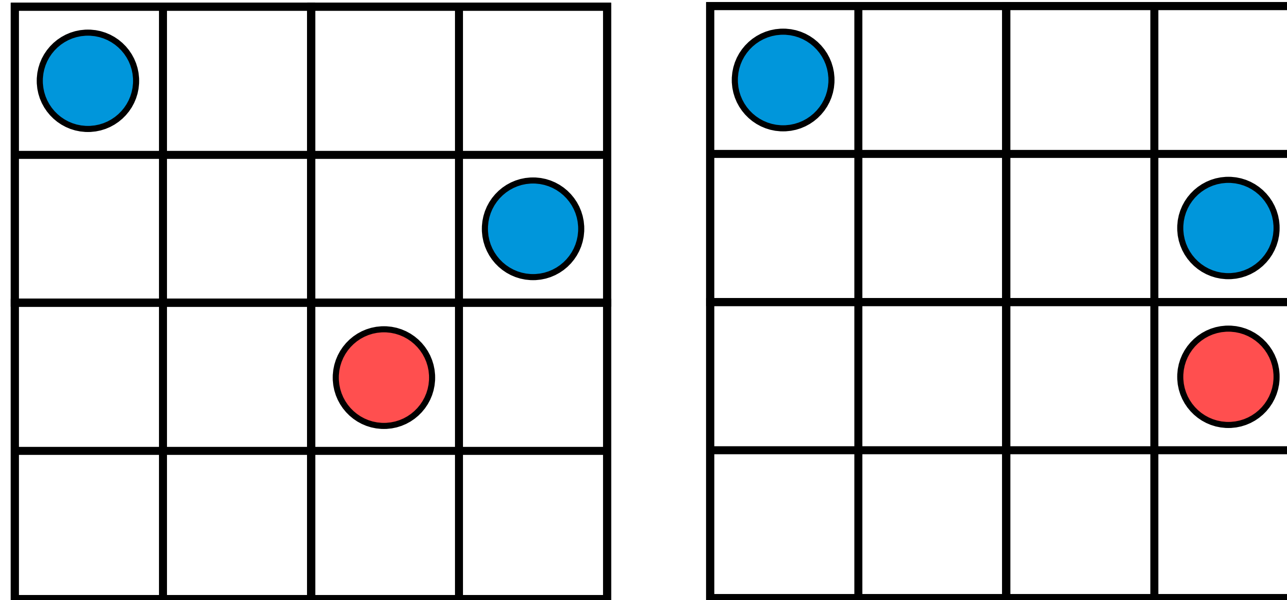
#9663 N-Queen



- 그런데, 1~3행을 저렇게 채워두면 4행에는 어느 곳에도 퀸을 놓을 수 없다.
- 잘못 놓은 경우이다. 다시 3행으로 돌아가자.

# 백트래킹 대표 문제 – N-Queen

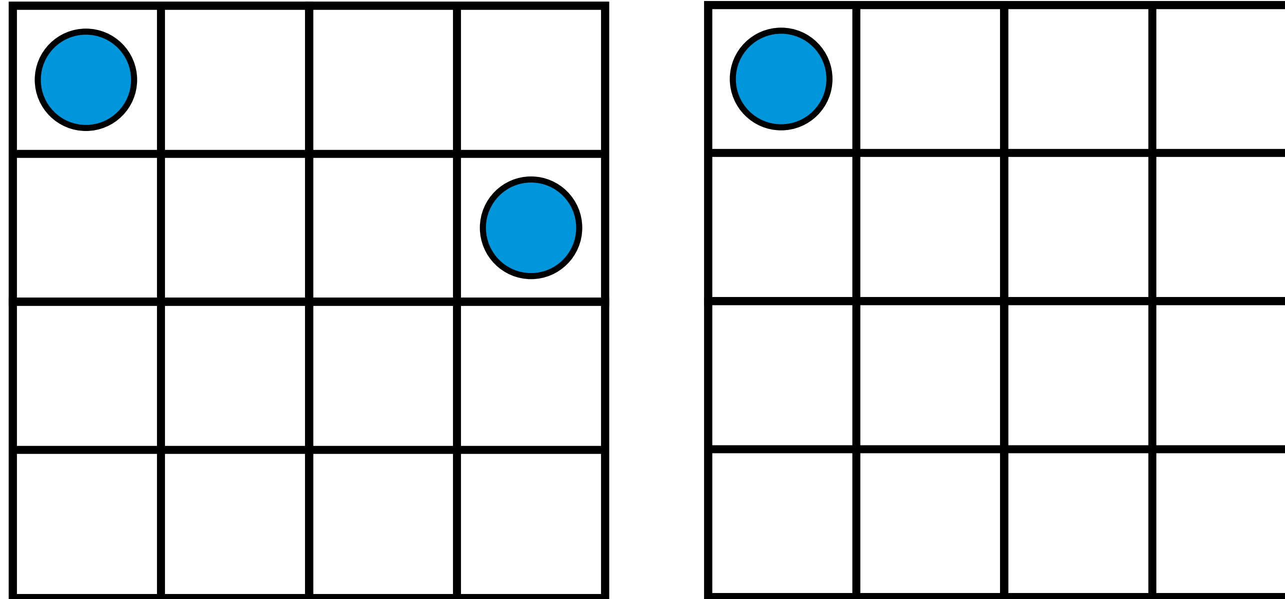
#9663 N-Queen



- 3행의 2열에 마지막으로 놓았으니, 3열부터 다시 놓을 수 있는지 체크해 보자
- 그런데 3, 4열 둘 다 놓을 수 없다. 다시 2행으로 돌아가자.

# 백트래킹 대표 문제 – N-Queen

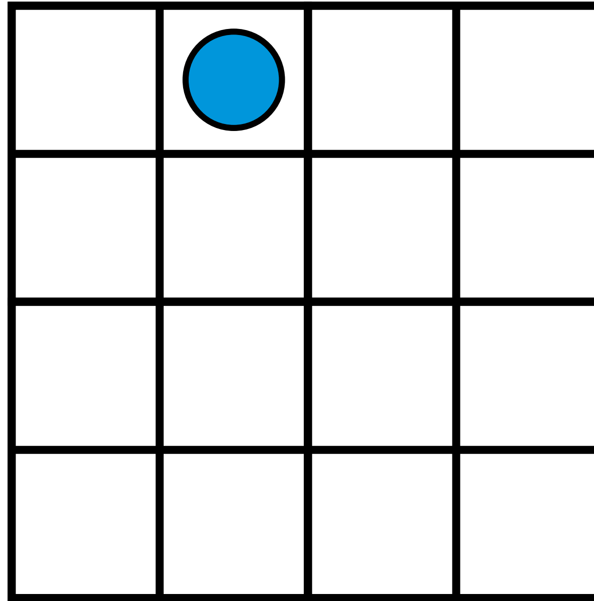
#9663 N-Queen



- 그런데 2행은 이미 4열까지 왔기 때문에, 즉 가능한 경우의 수를 모두 봤기 때문에, 다시 1행으로 돌아가야 한다.

# 백트래킹 대표 문제 – N-Queen

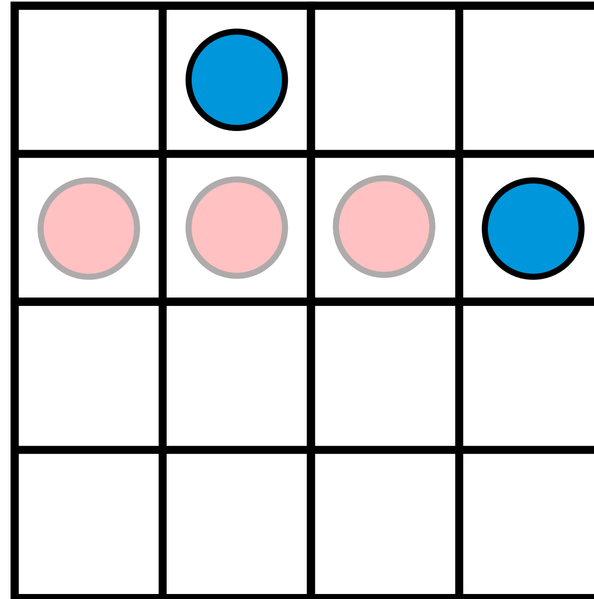
#9663 N-Queen



- 이제 1행의 퀸을 옮겨서, 2열에 놓아보고 다시 2~4행을 채워보자

# 백트래킹 대표 문제 – N-Queen

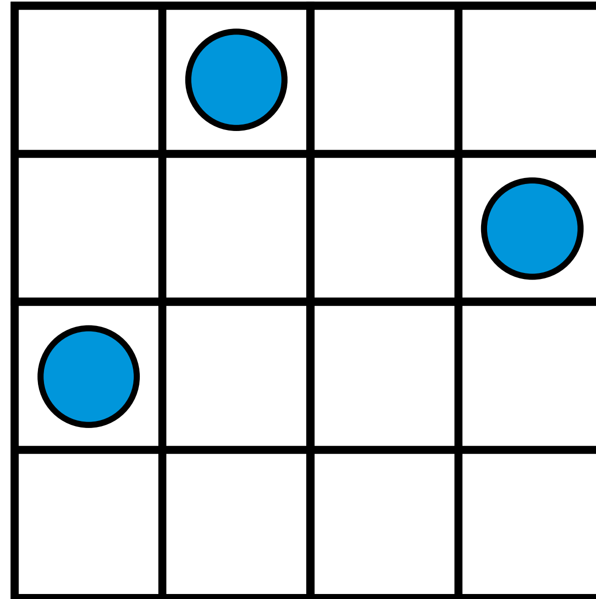
#9663 N-Queen



- 2행에서는 4열에만 놓을 수 있다

# 백트래킹 대표 문제 – N-Queen

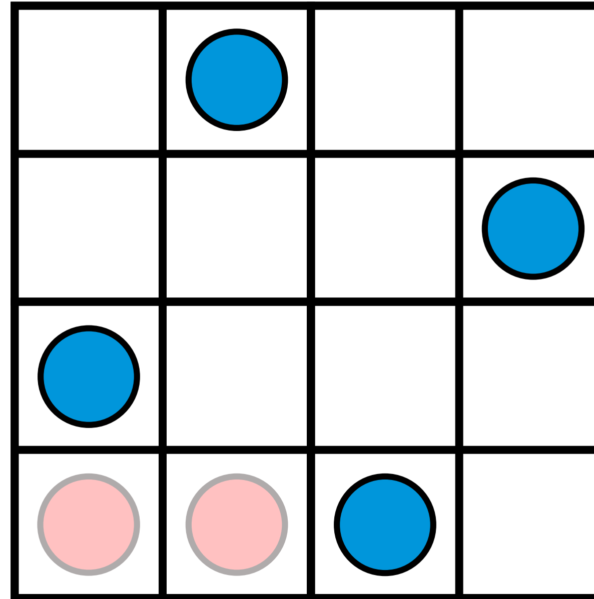
#9663 N-Queen



- 3행에서는 1열이 되니까 일단 놓아보고

# 백트래킹 대표 문제 – N-Queen

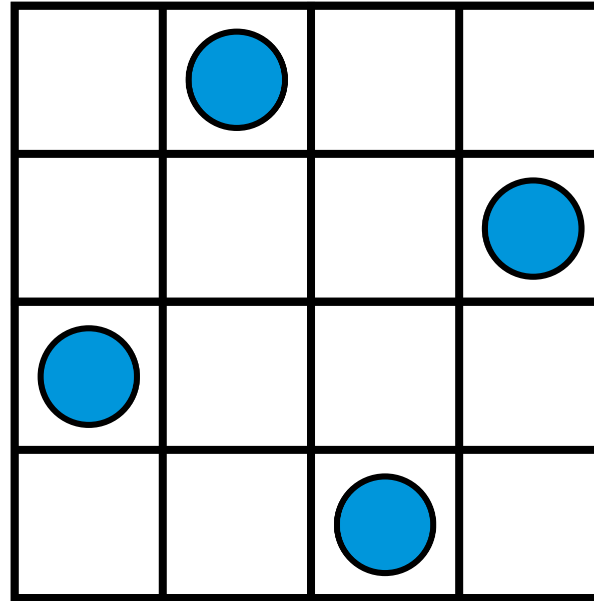
## #9663 N-Queen



- 4행에는 1, 2열은 안 되고, 3열에 놓을 수 있다.

# 백트래킹 대표 문제 – N-Queen

#9663 N-Queen



- 조건을 충족시키면서 마지막 행까지 도달했으므로,
  ‘체스판 위에 퀸 N개 놓는 경우의 수’ 하나를 찾은 것!

# 백트래킹 대표 문제 – N-Queen

## #9663 N-Queen

- 이런 식으로, 조건을 만족시키면서,
  조건에 맞지 않는 경우의 수는 배제하면서 완전 탐색을 진행하는 방식이
  백트래킹(가지치기)이다.

# N-Queen – 구현

#9663 N-Queen

- main() 부분

```
int n;
int row[15];  // row[x] : x번째 행에 퀸이 몇 번째 열에 있는지
int cnt;  // 경우의 수

int main() {
    cin >> n;

    // solution(x) : x번째 행에 퀸을 놓아보는 함수
    // 0번째 행부터 시작
    solution(0);
    cout << cnt << '\n';

    return 0;
}
```

# N-Queen – 구현

#9663 N-Queen

- solution() 부분

```
// solution(x) : x번째 행에 퀸을 놓아보는 함수
void solution(int x) {
    // 기저 케이스 부분

    // x행의 0열~(n-1)열까지 보면서 퀸을 놓아보고, 조건을 만족한다면 계속 진행
    for (int i = 0; i < n; i++) {
        row[x] = i;
        if (check(x)) {
            solution(x + 1);
        }
    }
}
```

# N-Queen – 구현

#9663 N-Queen

- solution() 부분 – 퀸을 놓아보고, 조건을 만족해야만 다음 solution(x+1) 호출

```
// solution(x) : x번째 행에 퀸을 놓아보는 함수
void solution(int x) {
    // 기저 케이스 부분

    // x행의 0열~(n-1)열까지 보면서 퀸을 놓아보고, 조건을 만족한다면 계속 진행
    for (int i = 0; i < n; i++) {
        row[x] = i;
        if (check(x)) {
            solution(x + 1);
        }
    }
}
```

# N-Queen – 구현

#9663 N-Queen

- solution() 부분 – 기저 케이스

```
void solution(int x) {
    if (x == n) {
        // 체스판 맨 밑 row까지 왔으면, 하나의 방법을 찾은 것
        // 경우의 수를 하나 추가하고 return
        cnt++;
        return;
    }

    // ...
}
```

# N-Queen – 구현

#9663 N-Queen

- solution(n)을 호출했다는 것은 조건을 만족하며 마지막 행까지 채웠다는 것

```
void solution(int x) {
    if (x == n) {
        // 체스판 맨 밑 row까지 왔으면, 하나의 방법을 찾은 것
        // 경우의 수를 하나 추가하고 return
        cnt++;
        return;
    }

    // ...
}
```

# N-Queen – 구현

#9663 N-Queen

- check(x) : 현재 x행까지 채운 상태일 때, 지금까지 채운 체스판이 조건에 맞는지 확인

```
bool check(int x) {
    for (int i = 0; i < x; i++) {
        if (row[x] == row[i] || x - i == abs(row[x] - row[i])) {
            return false;
        }
    }

    return true;
}
```

# N-Queen – 구현

#9663 N-Queen

- 세로로 겹치는 퀸이 있는지, 대각선으로 겹치는 퀸이 있는지 확인

```
bool check(int x) {
    for (int i = 0; i < x; i++) {
        if (row[x] == row[i] || x - i == abs(row[x] - row[i])) {
            return false;
        }
    }

    return true;
}
```

# Practice

#2231 분해합

#1526 가장 큰 금민수

#1018 체스판 다시 칠하기

#1065 한수

#14500 테트로미노

#6603 로또

#1759 암호 만들기

#1987 알파벳

#15684 사다리 조작

#2661 좋은수열

#2239 스도쿠