

2021-2 알고리즘

재귀(Recursion) II
: 문제 공간 탐색



한남대학교 컴퓨터공학과

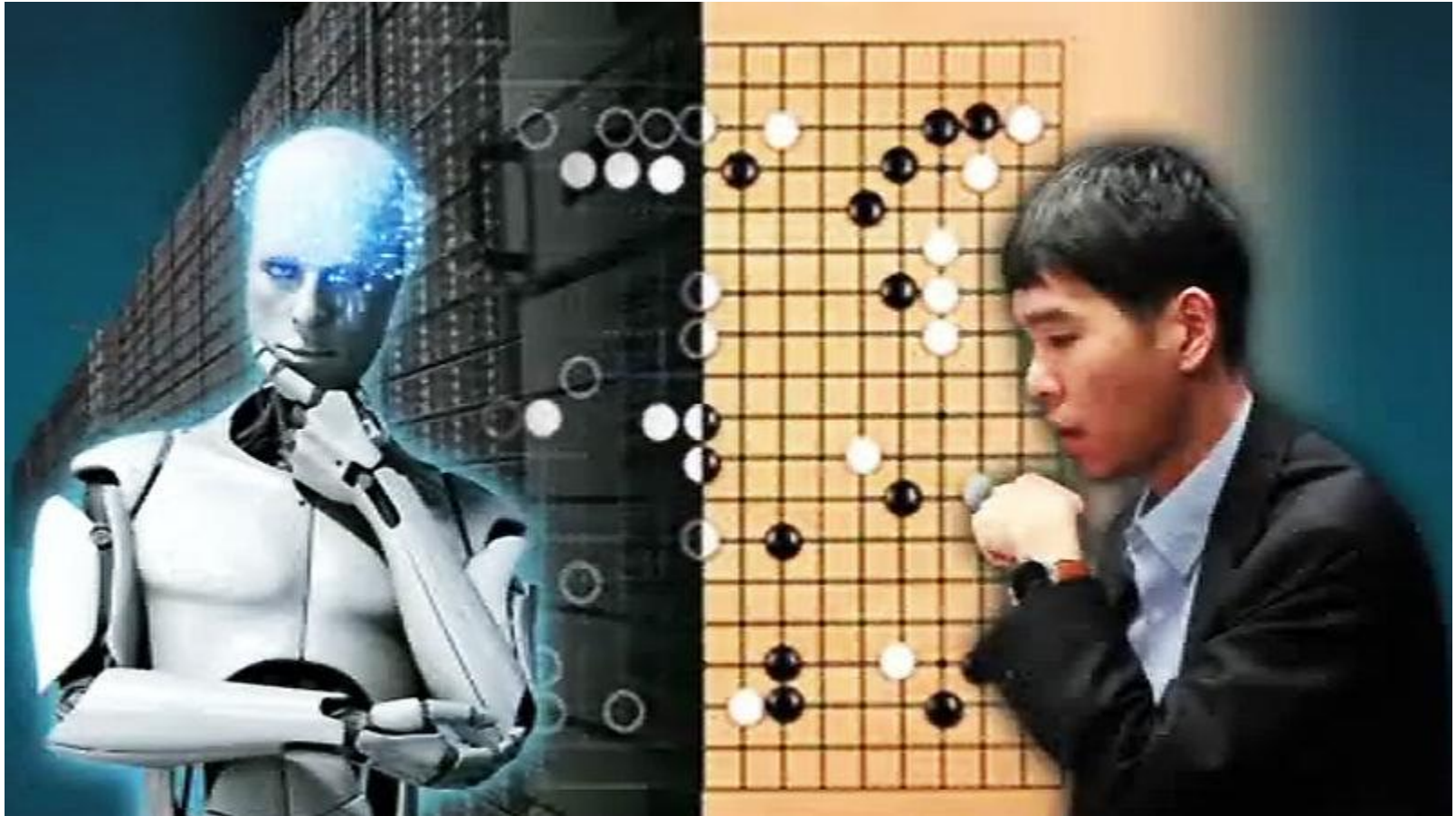
7장 구성

- 워밍업
- 완전 탐색 알고리즘(1)
- 완전 탐색 알고리즘(2)
- 백트래킹(1)
- 백트래킹(2)

워밍업



2016 세기의 대결



“알파고 바둑은 컴퓨터 무한자원 활용한 불공정 게임”

한겨레신문 2016.03.11, <https://www.hani.co.kr/arti/sports/baduk/734463.html>

f t talk link star print +



구체적으로는 알파고가 모든 경우의 수를 다 탐색하는 알고리즘인 브루트 포스(Brute force)를 일종의 '훈수꾼'으로 사용해 100% 승리할 수밖에 없다고 주장했다.



9일 오후 서울 종로구 포시즌스 호텔에서 열린 '구글 딥마인드 챌린지 매치'에서 구글 딥마인드가 개발한 인공지능 프로그램 '알파고'와 이세돌 9단(오른쪽)의 대국이 진행되고 있다. 구글 제공

IT전문 변호사 “알파고 무제한 훈수꾼 뒤”
“알파고가 하는 것 바둑 아냐...구글이 사과해야”

인공지능(AI) 알파고가 10일 이세돌 9단을 누르고 2승을 올린것과 관련해 정보통신(IT) 전문 변호사가 이 대국이 애초부터 불공정 게임이라며 이 9단의 필패를 예측한 것으로 드러나 눈길을 끌고 있다.

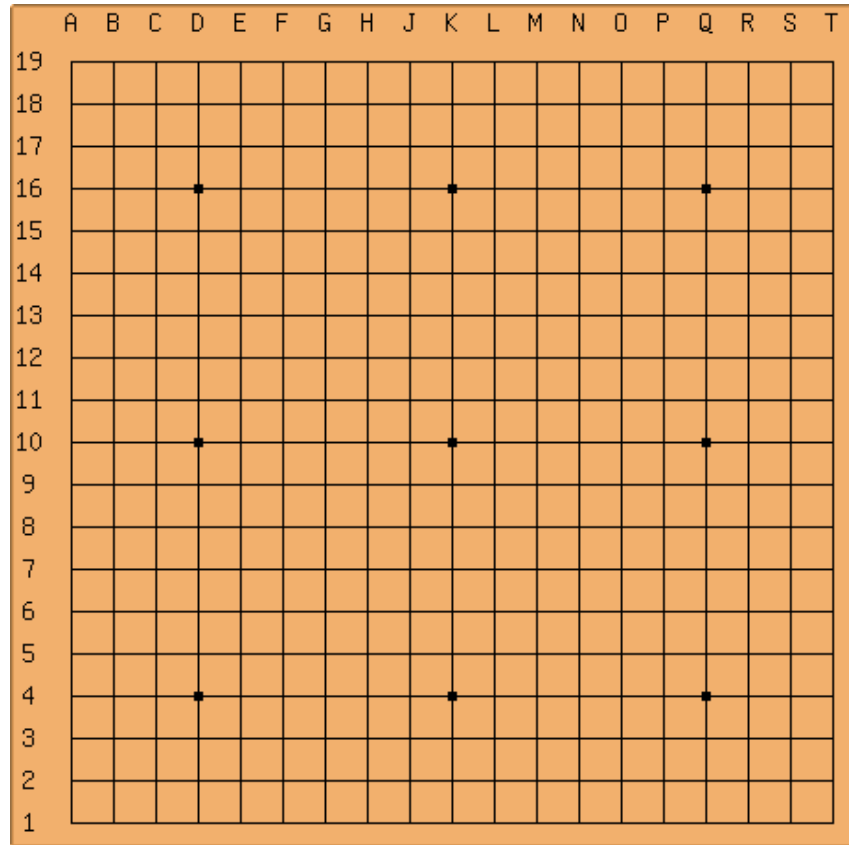
Brute Force?

- 후지쯔 사의 후가쿠(Fugaku)
 - 2021 현존 최고 성능의 슈퍼컴퓨터
 - 442페타플롭스(PF, petaflops, 초당 1000조 회 연산)



Brute Force?

- $19 \times 19 = 361$
- 경우의 수 $\leq 361!$



- 10⁷⁴² 년

완전 탐색 알고리즘(1)



완전 탐색(Exhaustive Search)

- **완전 탐색 알고리즘**

- 가능한 모든 해(**경우의 수**)를 탐색하는 알고리즘들을 가리킴
- 문제 공간에서 모든 경우의 수를 검사해 본 후 가장 좋은 해를 선택
- 단순무식한 방법Brute Force이라고도 부른다.
- 정해진 방법은 없고, 문제에 따라 여러 가지 알고리즘을 사용

- **완전 탐색 알고리즘의 의의**

- 어떤 문제에 대해, 더 좋은 알고리즘이 존재할 때도 많지만 좋은 알고리즘들도 기본적으로는 완전 탐색에서 출발한다.
- 완전 탐색은 문제 공간을 탐색해서 최적해를 찾기 위한 가장 기본적인 방법이며,
- 더 좋은 알고리즘을 찾기 위한 기초 작업이라고 할 수 있다.

Review: 재귀를 모르면 풀 수 없는 문제

풀기 어려운

- 문제1:

- 세 개의 문자 'a', 'b', 'c'를 한 번씩만 써서 만들 수 있는 모든 순열을 출력
- → 3중 반복문

```
abc  
acb  
bac  
bca  
cab  
cba
```

- 문제2:

- n 개의 문자 c_1, c_2, \dots, c_n 를 한 번씩만 써서 만들 수 있는 모든 순열을 출력
- → ???

완전 탐색: 사전식으로 나열하기

- 입력: n , n 개의 문자들의 집합
 - $C = \{c_1, c_2, \dots, c_n\}$
- 출력: C 에 속한 문자로 만들 수 있는 길이 n 인 모든 문자열
 - 중복 허용 characters can be used repeatedly
- 문제의 정의
- s_k : C 에 속한 문자로 만들 수 있는 길이 k 인 임의의 문자열
 - (가정) 길이 $(k-1)$ 인 문자열 s_{k-1} 이 주어질 때,
 - (관계) s_{k-1} 에 C 에 속한 문자 한 개를 더한다(s_k 의 개수는 C 의 크기와 같다).
 - (경계 조건) $S_0 = \emptyset$, $k \leq n$

완전 탐색: 사전식 나열

- 문제의 정의
- s_k : C에 속한 문자로 만들 수 있는 길이 k인 임의의 문자열
 - (가정) 길이 (k-1)인 문자열 s_{k-1} 이 주어질 때,
 - (관계) s_{k-1} 에 C에 속한 문자 한 개를 더한다(s_k 의 개수는 C의 크기와 같다).
 - (경계 조건) $S_0 = \emptyset, k \leq n$

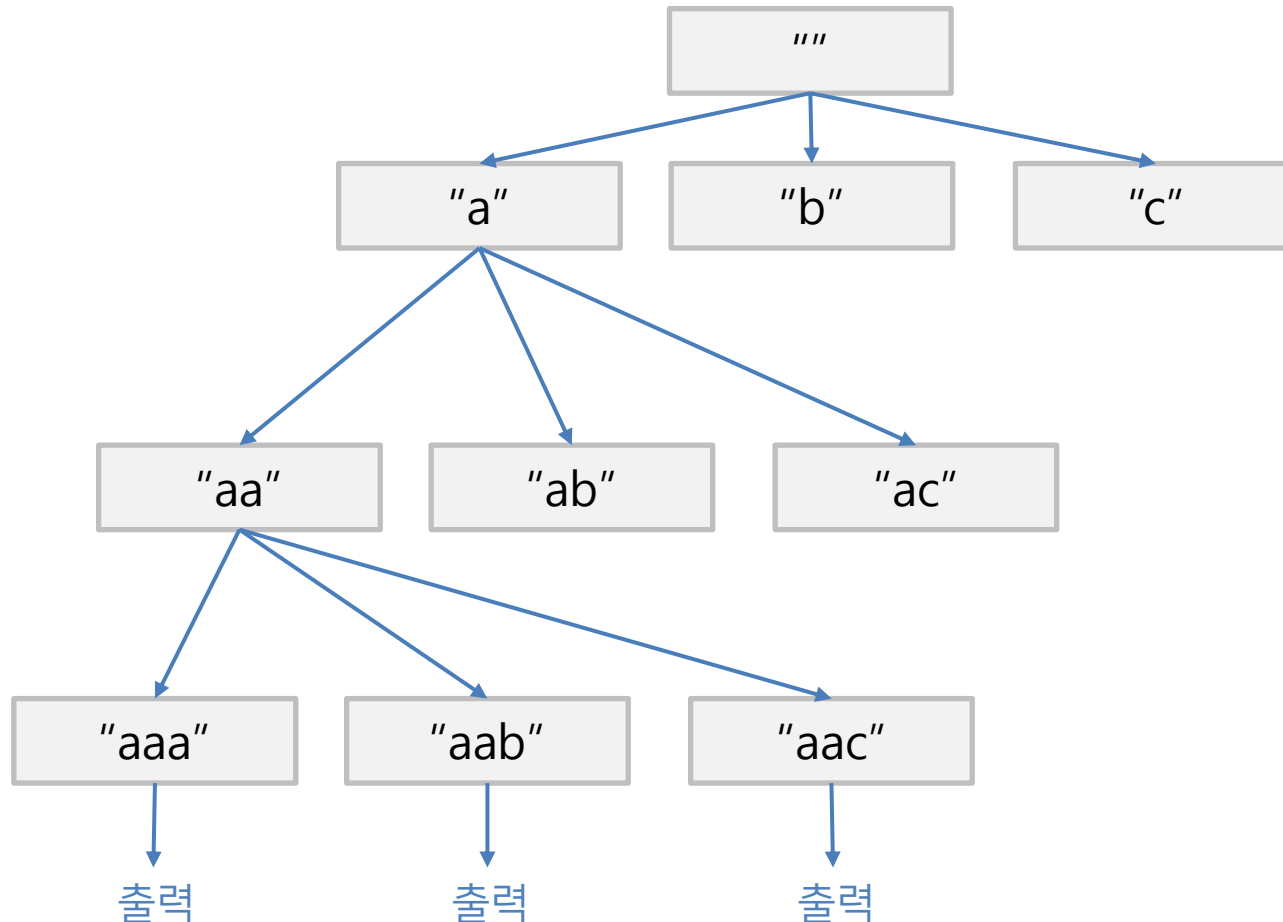
```
def enum_dict_order(s):  
    if len(s) == len(char_set):  
        print(s)  
    else:  
        for char in char_set:  
            enum_dict_order(s + char)
```

```
char_set = list("abc")  
enum_dict_order("")
```

```
aaa  
aab  
aac  
aba  
...  
ccb  
ccc
```

완전 탐색: 사전식 나열

- 문제 공간 탐색(Searching Problem Space)



완전 탐색: 사전식 나열

- 문자열 집합을 리턴 받아서 출력

```
def enum_dict_order(s):  
    if len(s) == len(char_set):  
        return [s]  
  
    result = []  
    for char in char_set:  
        strings = enum_dict_order(s + char)  
        result.extend(strings)  
  
    return result  
  
char_set = list("abc")  
result = enum_dict_order("")  
for s in result:  
    print(s)
```

```
aaa  
aab  
aac  
aba  
...  
ccb  
ccc
```

완전 탐색: 사전식 나열

- (참고) 문자열 집합으로 다루기
- 문제의 정의
 - $S(k) = C$ 에 소한 문자로 만들 수 있는 길이 k 인 모든 문자열들의 집합
 - Let $S(k)$ = Set of all strings of length k such that consist of the characters belong to C .
 - $S(k) =$
 - (가정) 길이 $(k-1)$ 인 모든 문자열 들의 집합 S_{k-1} 이 주어질 때,
 - (관계) S_{k-1} 에 모든 문자열에 대해,
 - C 에 속한 문자 한 개를 더해서 만들 수 있는 길이 k 인 문자열들의 집합
 - (경계 조건) $S_0 = \emptyset, k \leq n$

완전 탐색: 사전식 나열

- (참고) 문자열 집합으로 다루기

```
def enum_dict_order(given_strings, k):  
    if k == n:  
        return given_strings  
  
    new_strings = []  
    for string in given_strings: # 길이 k-1인 문자열들  
        for char in char_set:  
            new_string = string + char  
            new_strings.append(new_string)  
  
    return enum_dict_order(new_strings, k + 1)  
  
char_set = list("abc")  
n = len(char_set)  
all_strings = enum_dict_order([""], 0)  
for string in all_strings[1:]:  
    print(string)
```

```
aaa  
aab  
aac  
aba  
...  
ccb  
ccc
```

완전 탐색 알고리즘(2)

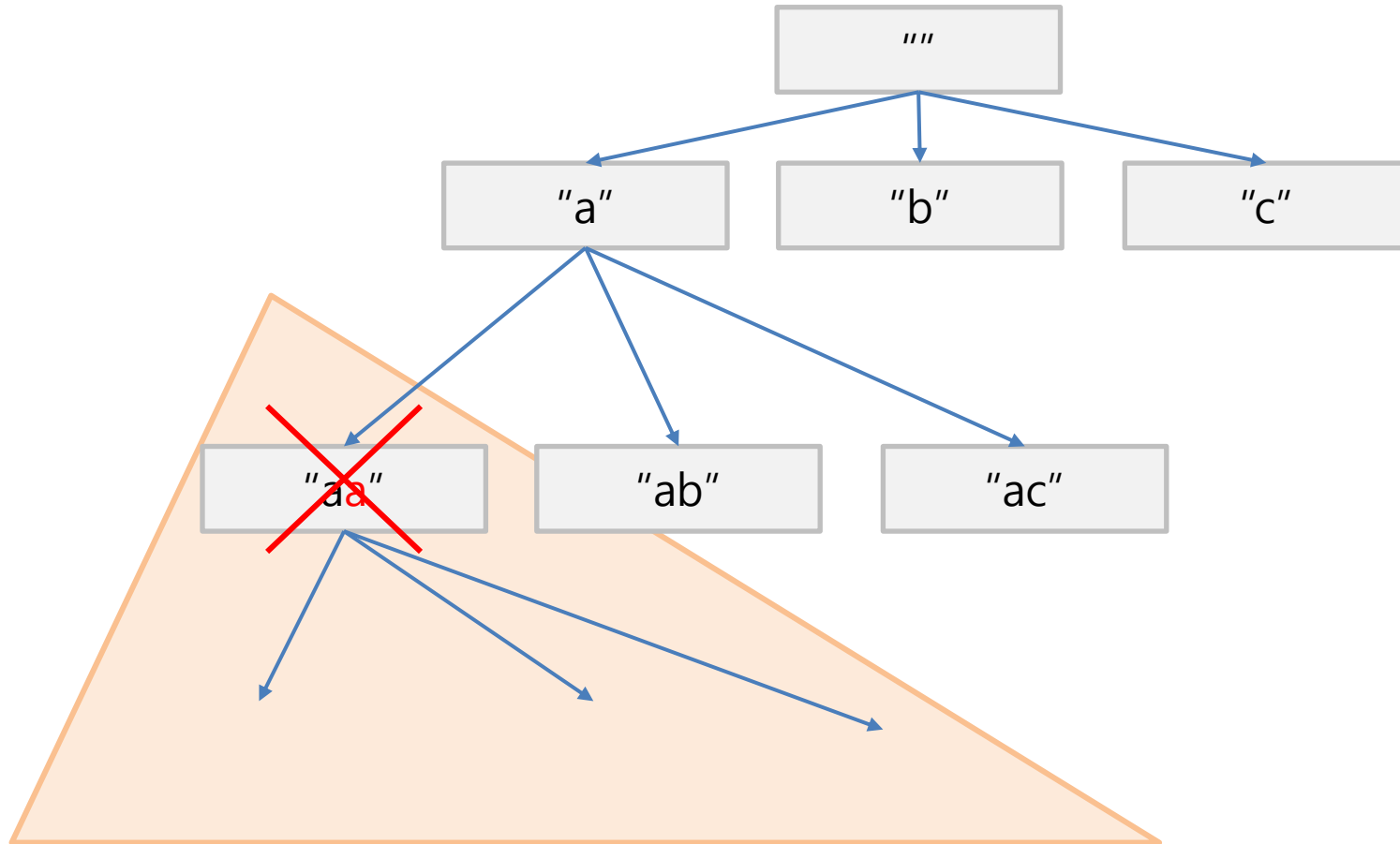


완전 탐색: 사전식 나열

- 입력: n , n 개의 문자들의 집합
 - $C = \{c_1, c_2, \dots, c_n\}$
- 출력: C 에 속한 문자를 **한번씩만 써서** 만들 수 있는 길이 n 인 모든 문자열
 - 중복 허용되지 않음. Characters can be used only once.

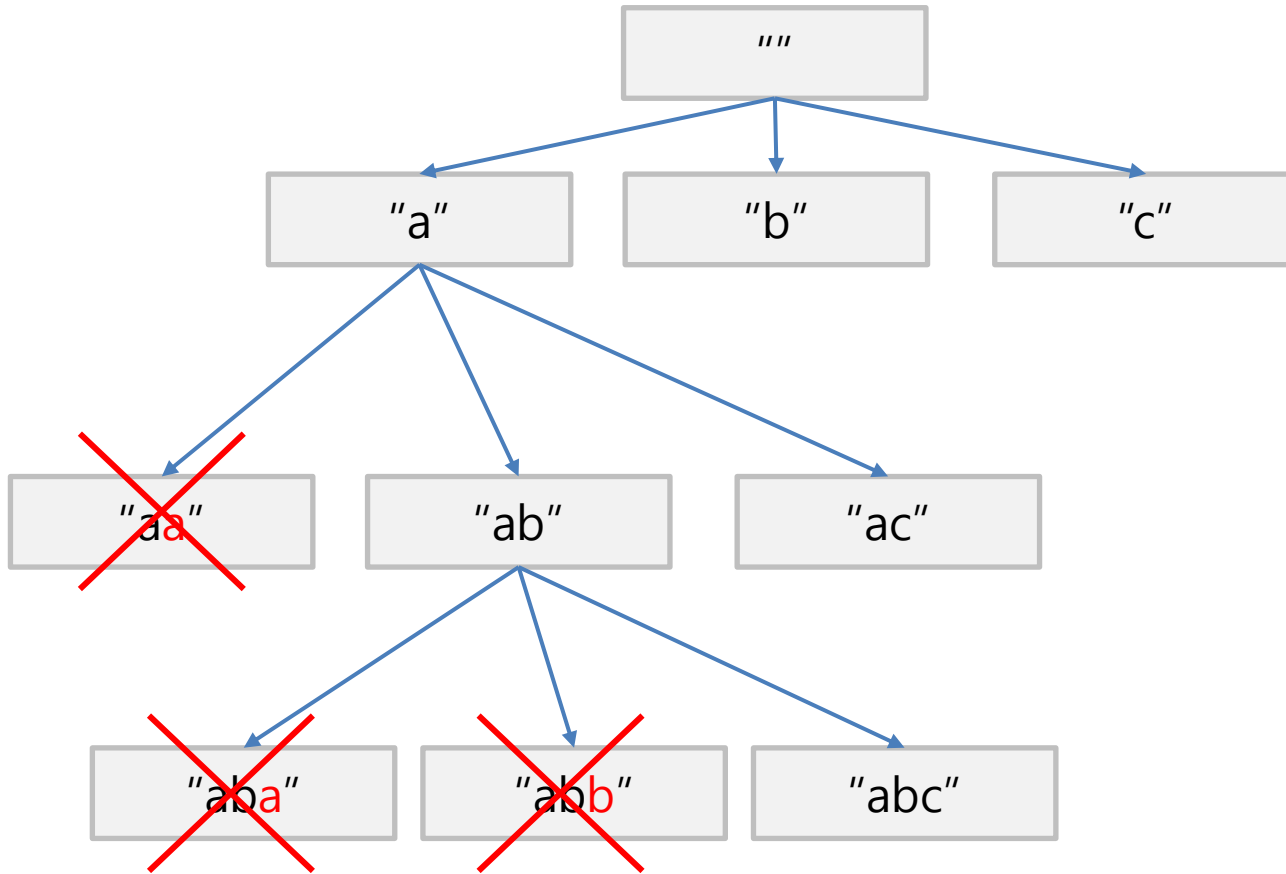
완전 탐색: 사전식 나열

- 문제 공간 탐색



완전 탐색: 사전식 나열

- 문제 공간 탐색



완전 탐색: 사전식 나열

- 입력: n , n 개의 문자들의 집합
 - $C = \{c_1, c_2, \dots, c_n\}$
- 출력: C 에 속한 문자를 **한번씩만 써서** 만들 수 있는 길이 n 인 모든 문자열

```
def enum_dict_order(s):  
    if len(s) == len(char_set):  
        print(s)  
    else:  
        for char in char_set:  
            if char not in s:  
                enum_dict_order(s + char)
```

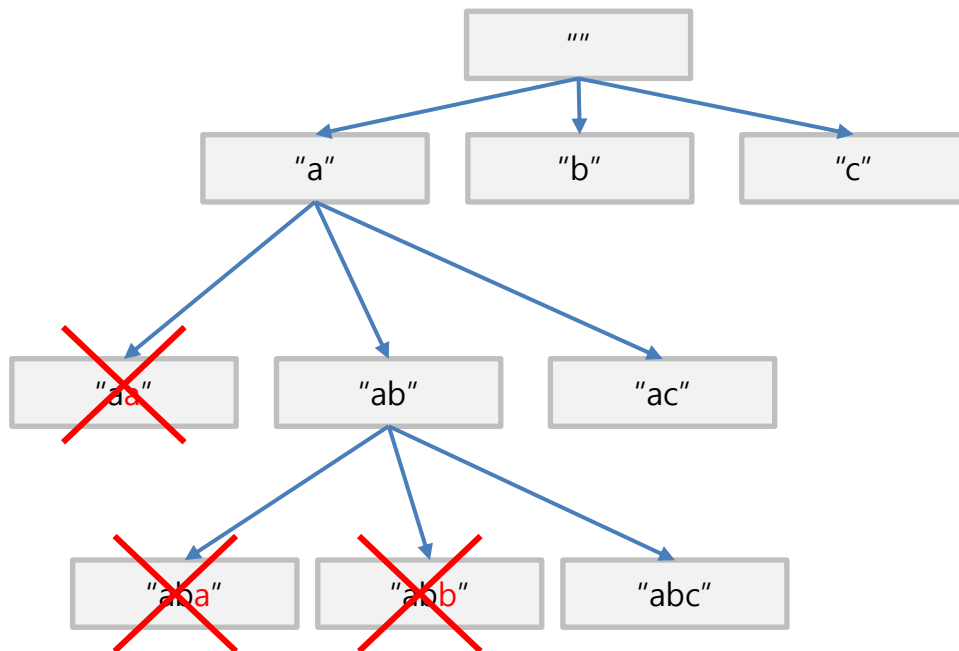
```
char_set = list("abc")  
enum_dict_order("")
```

```
abc  
acb  
bac  
bca  
cab  
cba
```

완전 탐색: 사전식 나열

- 가지치기(Pruning)

- 재귀 호출을 이용한 문제 공간 탐색에서
- (최적)해가 될 수 없는 공간의 탐색을 중단하는 기법




완전 탐색: 사전식 나열

- 한번 사용한 문자를 따로 기억하기

- 이 문제에서 당장은 필요 없지만 C로 구현하게 되면 따로 기억하는 게 좋음
- 이후 **그래프 탐색**에서 자주 사용하게 될 테크닉

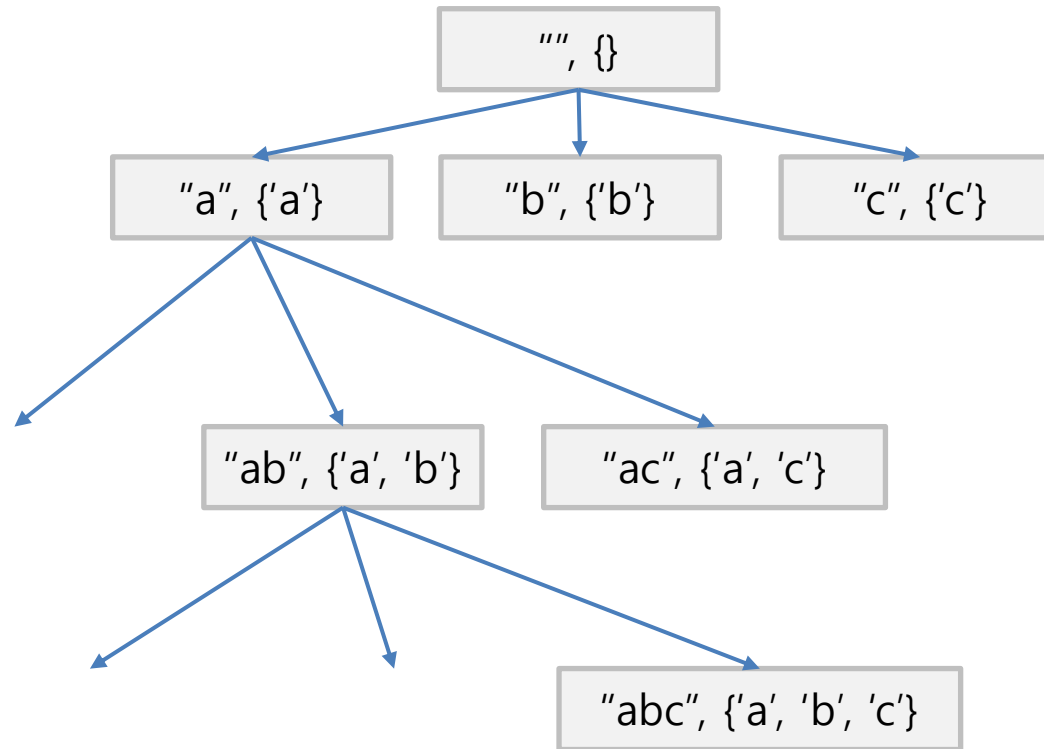
used_chars: 한번 사용한 문자들의 집합



```
def enum_dict_order(s, used_chars):  
    if len(s) == len(char_set):  
        print(s)  
        return  
  
    for char in char_set:  
        if char not in used_chars:  
            enum_dict_order(s + char, used_chars | {char})  
  
char_set = list("abc")  
enum_dict_order("", set())
```


완전 탐색: 사전식 나열

- 한번 사용한 문자를 따로 기억하기



- 연습문제
 - **used_chars** 대신 **unused_chars**를 기억하도록 수정해 보자.

연습문제

- **순열과 조합**
 - 배열(리스트)를 입력 받고, 배열의 원소들로 만들 수 있는
 - 순열(Permutation)과 조합(Combination)들을 출력해 보자.
- **(참고)**
 - `itertools.permutation()`
 - `itertools.combination()`

연습문제

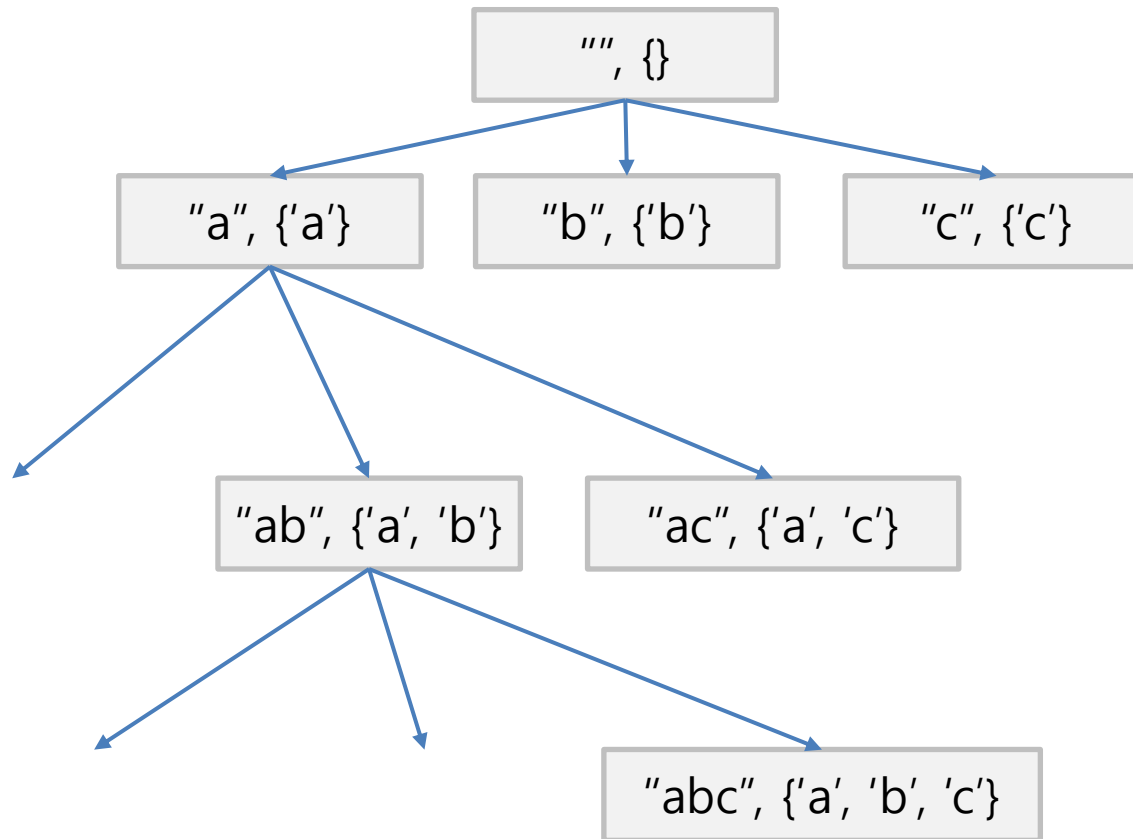
- 크기 n 인 배열, m , k 가 주어질 때,
- 배열의 원소 개를 더했을 때 m 에 가장 가까운 값을 구하라.
 - 예) $[10, 100, 24, 55, 99, 37], 70$
 - 예를 들어 현재까지 70에 가장 가까운 값이 79라면,
 - 원소들을 더한 값과 70의 차가 9 이상인 경우 탐색할 필요가 없음

백트래킹(1)



백트래킹을 배우기 전에...

- 한번 사용한 문자를 따로 기억하기
 - 앞의 코드에서 호출된 `enum_dict_orders()` 들은
 - `used_char`를 공유하지 않고 **각각 다른 `used_char` 집합을 가짐**



백트래킹을 배우기 전에...

- 앞의 코드를 아래와 같이 수정해 보자. 어떤 결과가 발생하는가?

```
def enum_dict_order(s, used_chars):  
    if len(s) == len(char_set):  
        print(s)  
        return  
  
    for char in char_set:  
        if char not in used_chars:  
            used_chars.add(char)  
            enum_dict_order(s + char, used_chars)  
  
char_set = list("abc")  
enum_dict_order("", set())
```

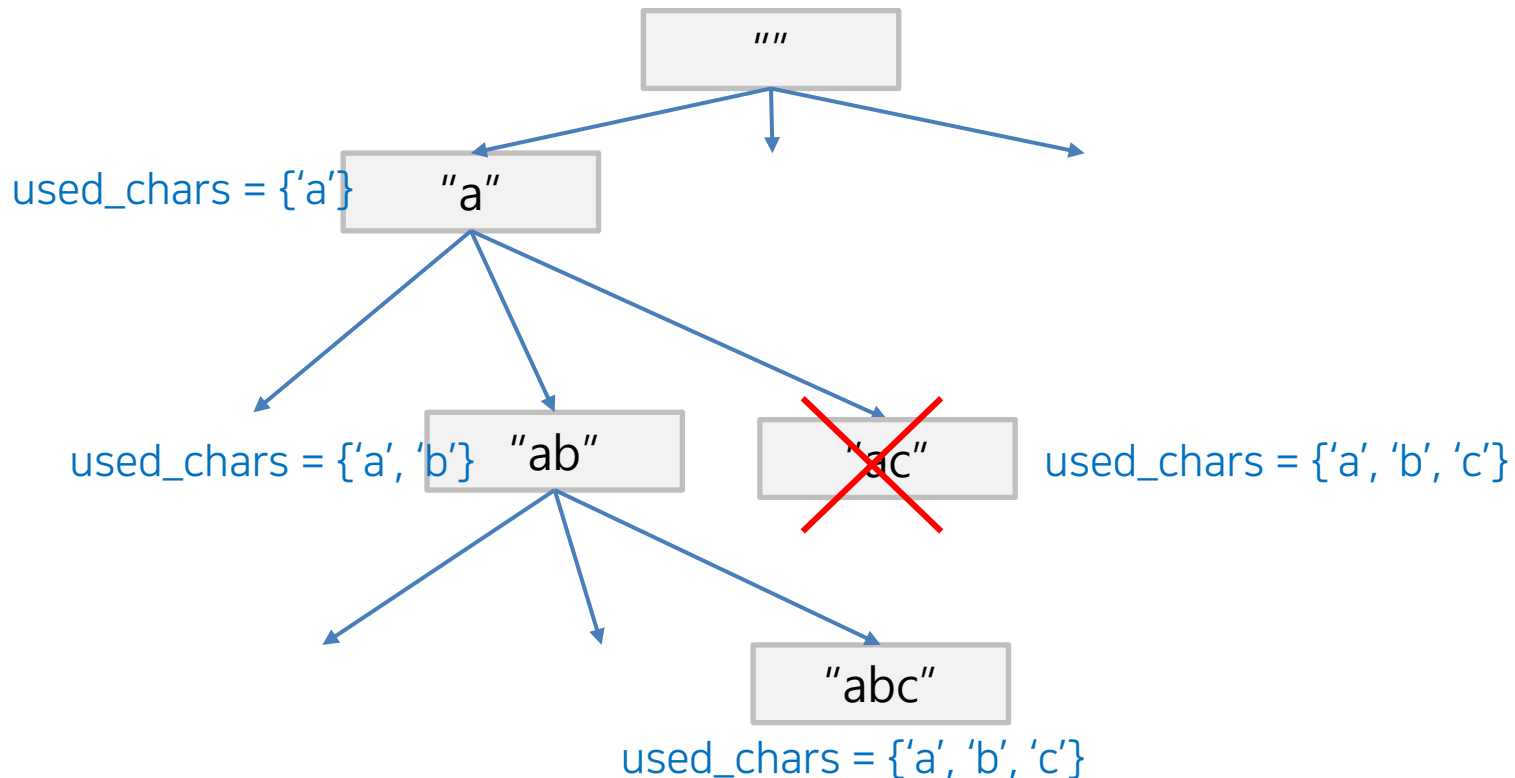
백트래킹을 배우기 전에...

- 조금 더 직관적인 코드

```
def enum_dict_order(s):  
    global used_chars  
  
    if len(s) == len(char_set):  
        print(s)  
        return  
  
    for char in char_set:  
        if char not in used_chars:  
            used_chars |= {char}  
            enum_dict_order(s + char)  
  
char_set = list("abc")  
used_chars = set()  
enum_dict_order("")
```

백트래킹을 배우기 전에...

- 하나의 `used_char` 집합이 프로그램 전체에 공유된다.
 - 아래 그림에서 `used_char`는 모두 하나의 객체를 가리킴
 - '객체를 복사하는가 참조하는가'의 차이



백트래킹을 배우기 전에...

• 복사 vs. 참조

- 재귀 호출에서 객체(구조체, 배열)를 복사해서 구현하면
- 호출된 함수(스택 프레임) 각각이 서로 다른 **상태(state)**를 가진다.

"" , { }

"ac" , { 'a' , 'c' }

"abc" , { 'a' , 'b' , 'c' }

- 재귀 호출의 특성 상, 이 모든 state들을 따로따로 기억하려면 엄청난 메모리 공간이 요구됨
- 재귀 호출 문제의 구현에 있어서 객체를 <복사 vs. 참조>하는 방법의 차이를 명확히 인지하고 활용할 수 있어야 함

백트래킹(Backtracking)

- 백트래킹(Backtracking)

- 프로그램 전체가 하나의 객체(상태)를 공유(전역 변수라고 생각하면 됨)
- 각각의 재귀 호출에서 필요한 상태 변화를 적용한 후,
- 거기에 따르는 작업이 끝나면 이전 상태로 되돌리는 roll-back 방법을 사용

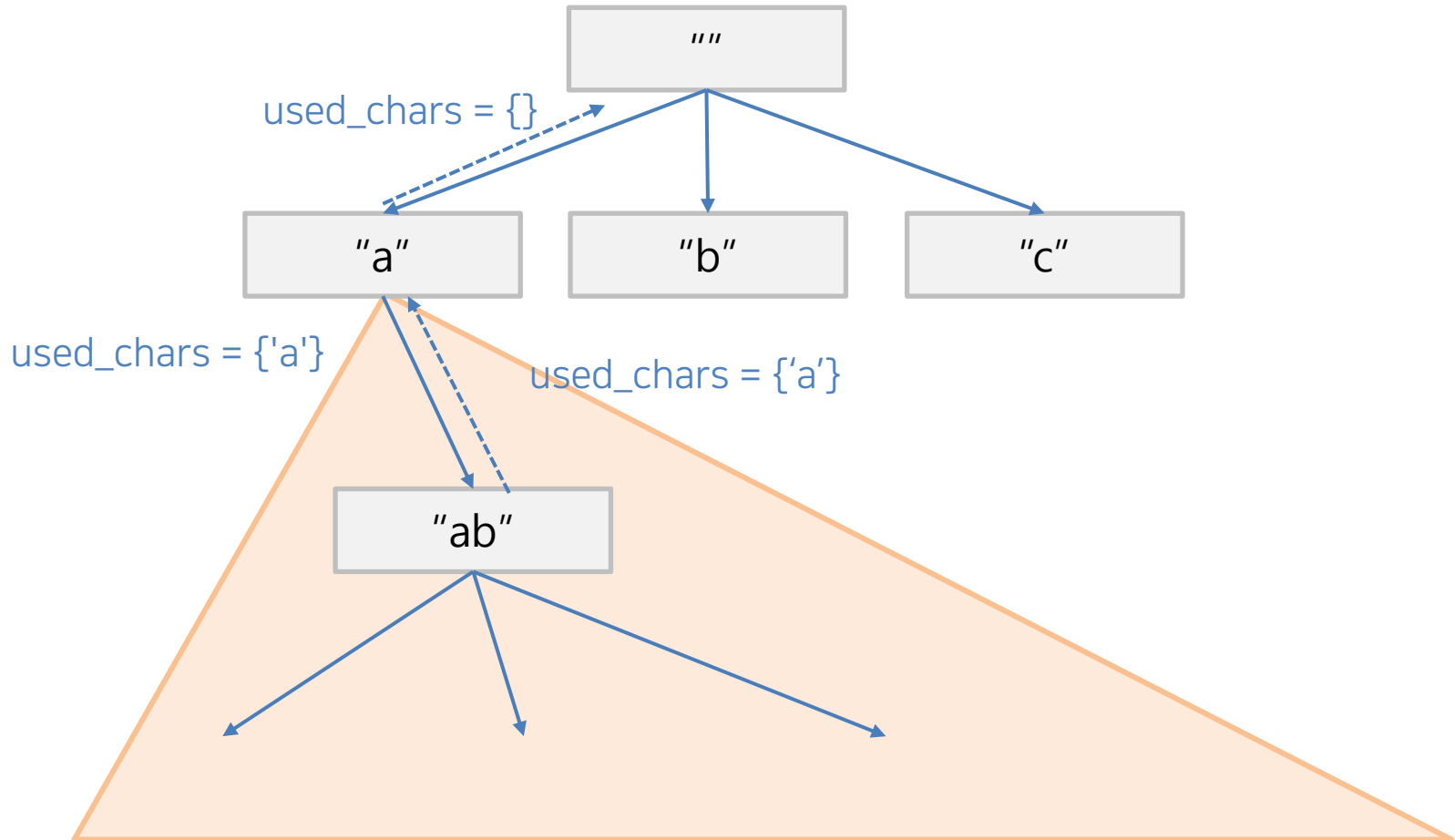
```
def enum_dict_order(s, used_chars):
    if len(s) == len(char_set):
        print(s)
        return

    for char in char_set:
        if char not in used_chars:
            used_chars.add(char)
            enum_dict_order(s + char, used_chars)
            used_chars.remove(char)

char_set = list("abc")
enum_dict_order("", set())
```

백트래킹(Backtracking)

- 함수 호출이 리턴된 후에는 전역 변수가 이전의 상태로 복구된다.

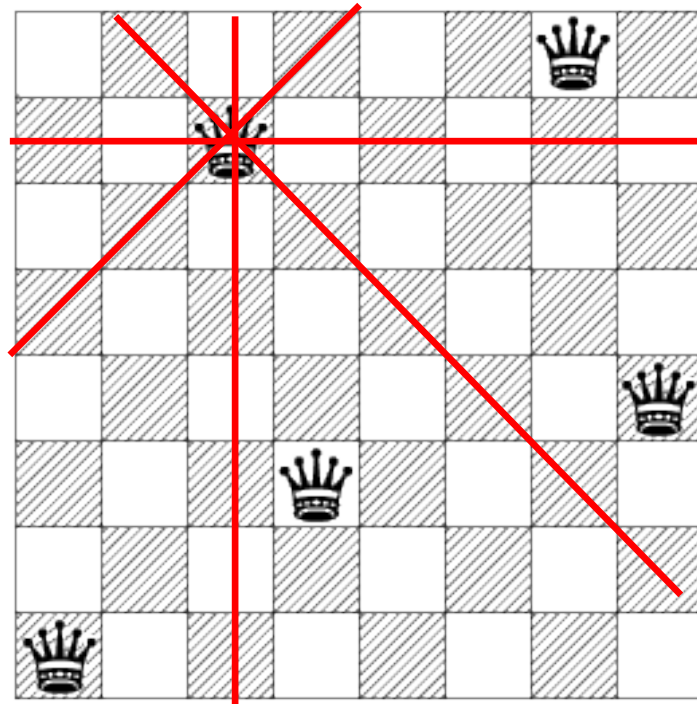


백트래킹(2)



8-Queen Problem

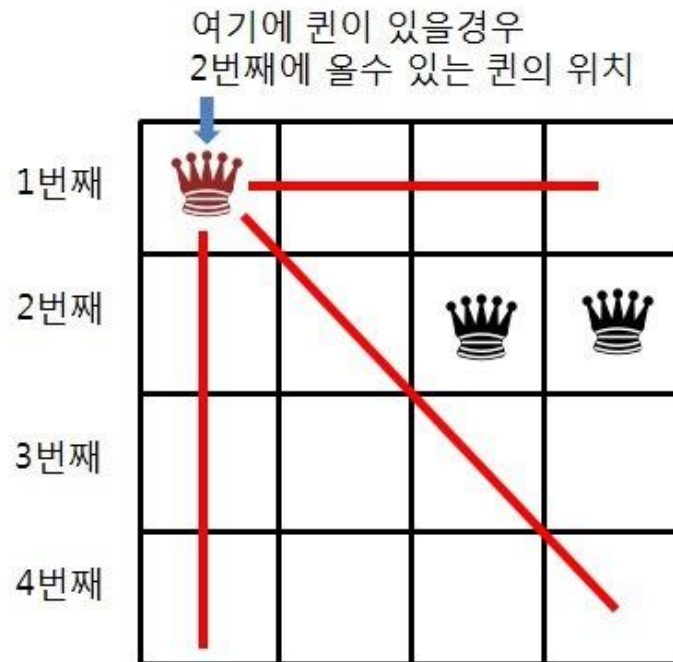
- 문제: 8x8 체스판에 퀸 8개를 배치하는 방법은 몇 가지인가?
- (또는 가능한 배치를 모두 출력)
 - Queen: 가로, 세로 대각선 8방향으로 거리 제한 없이 일직선 이동



8-Queen Problem

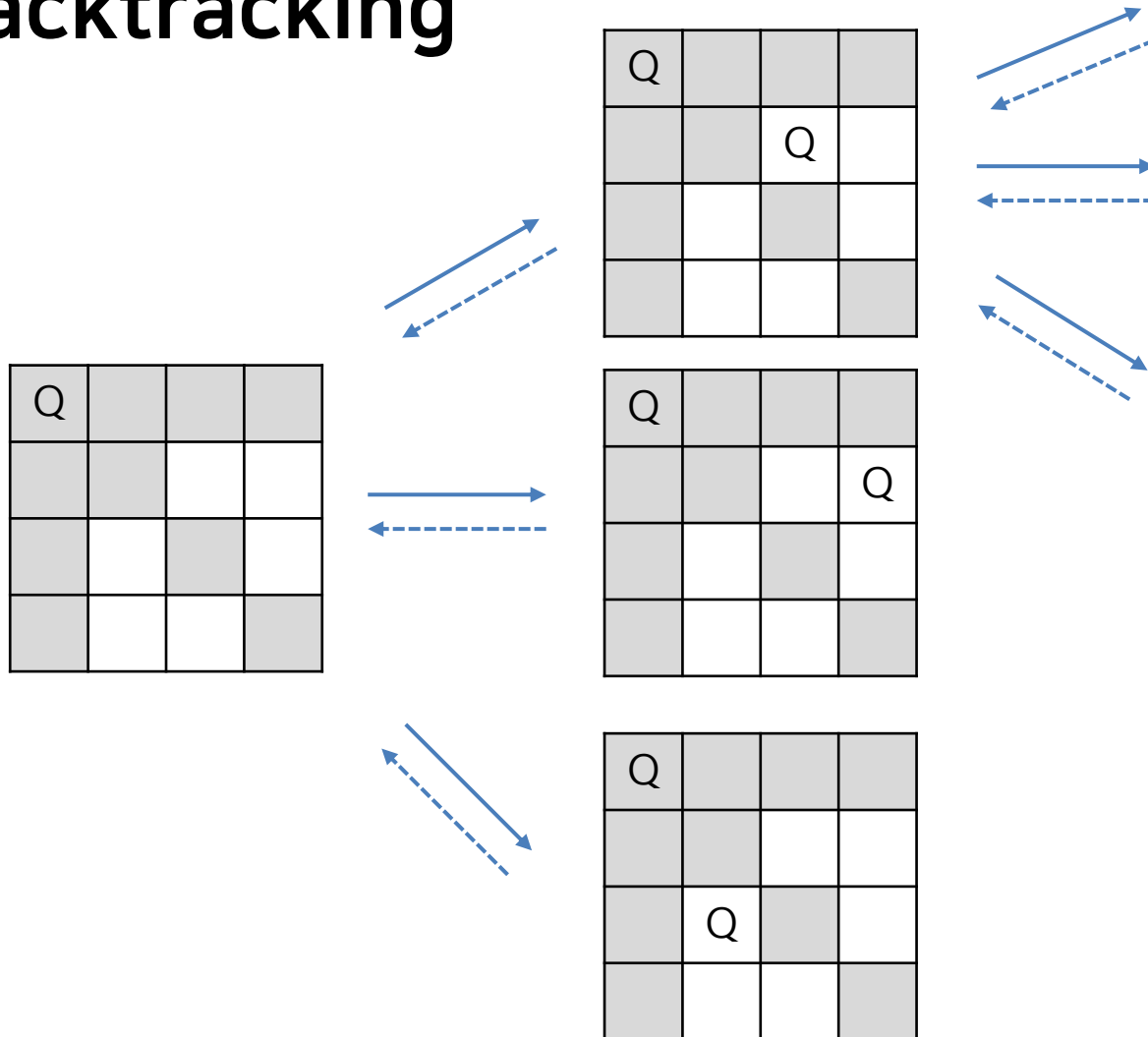
- 문제의 정의:

- (가정) 체스판에 $(n-1)$ 개 퀸이 배치된 상태
- (관계) n 번째 퀸을 놓을 수 있는 위치(들)



8-Queen Problem

- Backtracking



8-Queen Problem

- <Solution#1>

```
queens(n, board):
```

```
    n == 80이면 출력하고 return;
```

8x8 체스판을 탐색해서 퀸을 놓을 수 있는 자리 (x, y)를
찾는다(여러 개일 수도 있다).

각 (x, y)에 대해서

```
    board의 (x, y)에 퀸을 놓는다;
```

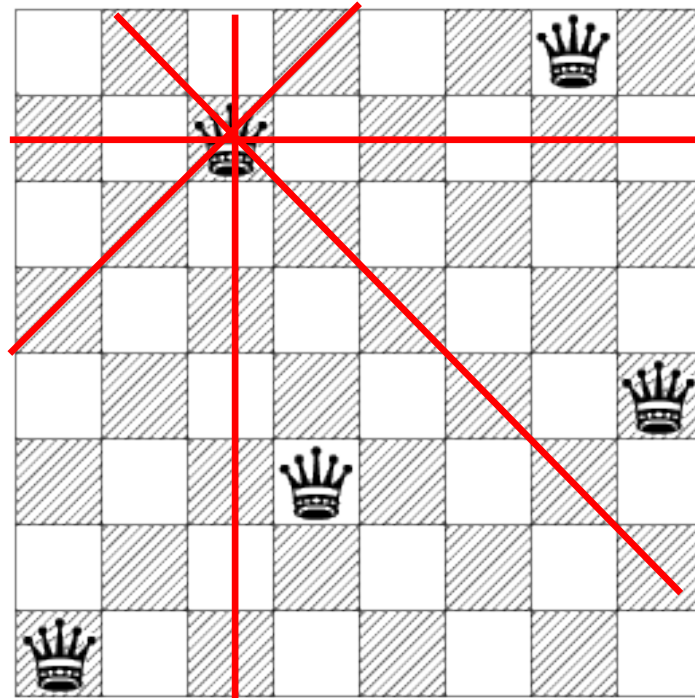
```
    eight_queen(n + 1, board)
```

```
    board의 (x, y)에서 퀸을 제거;    # backtracking
```


8-Queen Problem

- <Sub-Problem & Solution>

8x8 체스판의 상태가 주어질 때,
(x, y)에 퀸을 한 개 놓을 수 있는가? {
 (x, y)를 기준으로 8방향을 탐색한다;
 전부 비어 있으면 True, 아니면 False를 리턴;
}



8-Queen Problem

- <Sub-Problem & Solution> : Python code
 - 여러 가지 방법으로 구현할 수 있음

```
def placable(board, x, y):
    # 가로, 세로
    for i in range(8):
        if board[x][i] or board[i][y]:
            return False

    # xx, yy를 x, y에서부터 대각선 4방향으로 진행시킴
    for dx, dy in [(-1, -1), (1, 1), (1, -1), (-1, 1)]:
        xx = x + dx; yy = y + dy

        # 체스판 범위를 벗어나면 break
        while 0 <= xx < 8 and 0 <= yy < 8:
            # 다른 퀸이 발견되면 실패
            if board[xx][yy]:
                return False

            xx += dx; yy += dy

    return True
```

8-Queen Problem

- <Solution#2>
 - “퀸은 한 행(열)에 하나씩만 놓을 수 있다.”
 - Only 1 queen for each row(column).
 - (가정) 1..n행 중, row-1행까지 퀸이 배치된 상태
 - (관계) row행에서 퀸을 놓을 수 있는 자리들
 - 일종의 가지치기라고 볼 수 있음

```
queens(row, board):  
    if row > 8: 출력; return;
```

row행에서 퀸을 놓을 수 있는 자리 (x, y)들을 찾는다.

각 (x, y)에 대해서
board의 (x, y)에 퀸을 놓는다;
eight_queen(row + 1, board)
board의 (x, y)에서 퀸을 제거;

연습문제

- 1) Solution#1을 구현해 보자.
- 2) “퀸은 한 행(열)에 하나씩만 놓을 수 있다.”
 - 는 사실을 이용하면 8x8 체스판을 만드는 것보다 효율적인 표현이 가능하다.
 - 예를 들어 오른쪽과 같이 크기 8인 배열만 사용할 수도 있고,
 - 체스판을 그대로 만들고 ‘탐색’하지 않고 퀸의 좌표 값들을 계산하는 것만으로 배치 가능한지 확인할 수도 있다.
 - 작성한 코드에서 개선할 부분이 없는지 생각해 보자.

1행에서 퀸이 배치된 열 번호

2행에서 퀸이 배치된 열 번호

⋮

8행에서 퀸이 배치된 열 번호