

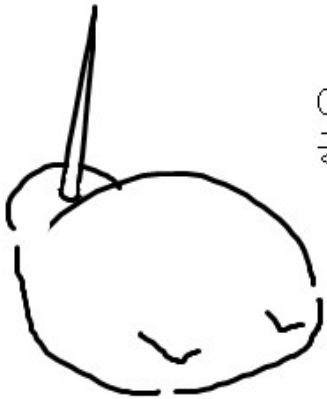
#3주차 알고리즘반

월요일이라 욱
한뎃바가지
하는중임



DFS, BFS, Backtracking

T. 김기현



아빠째 쇼
공부안할거임

공 부 하기 싫을땐



딱히 도움되진 않지만 이 포즈를 취해보세요

Graph(그래프)

3가지를 사용하는데 필요한 그래프에 대해 알아보자!

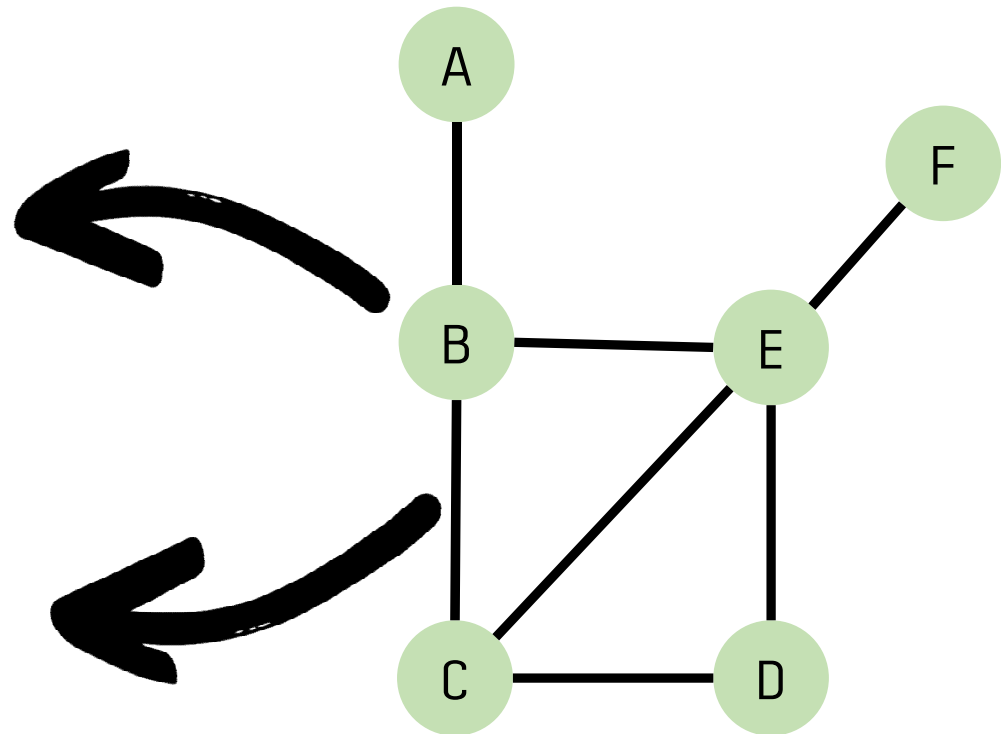
Graph(그래프)

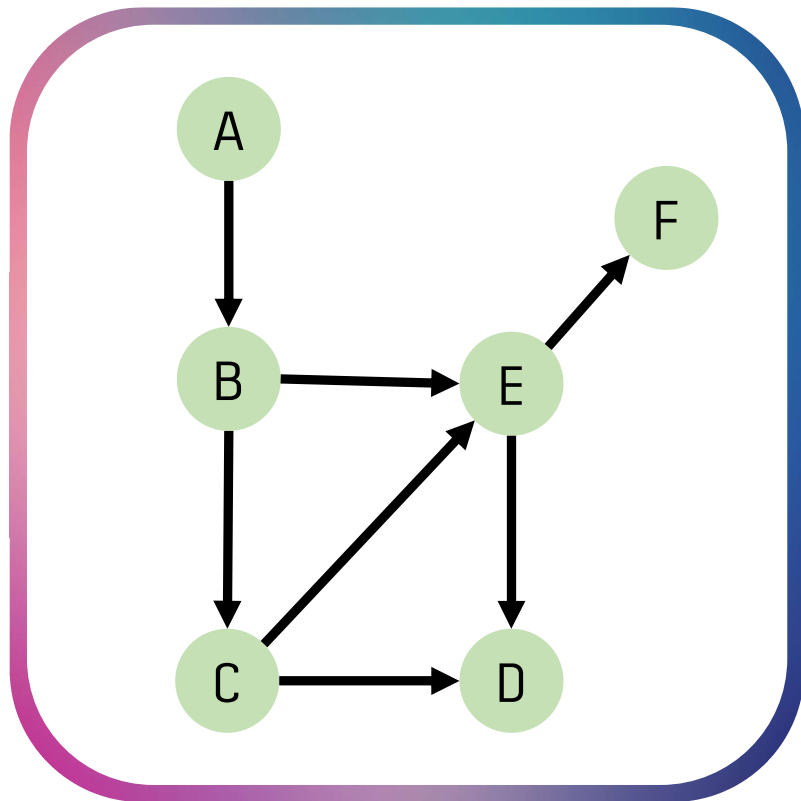
노드(N,node)

= 위치
알고리즘에서는 정점(vertex)

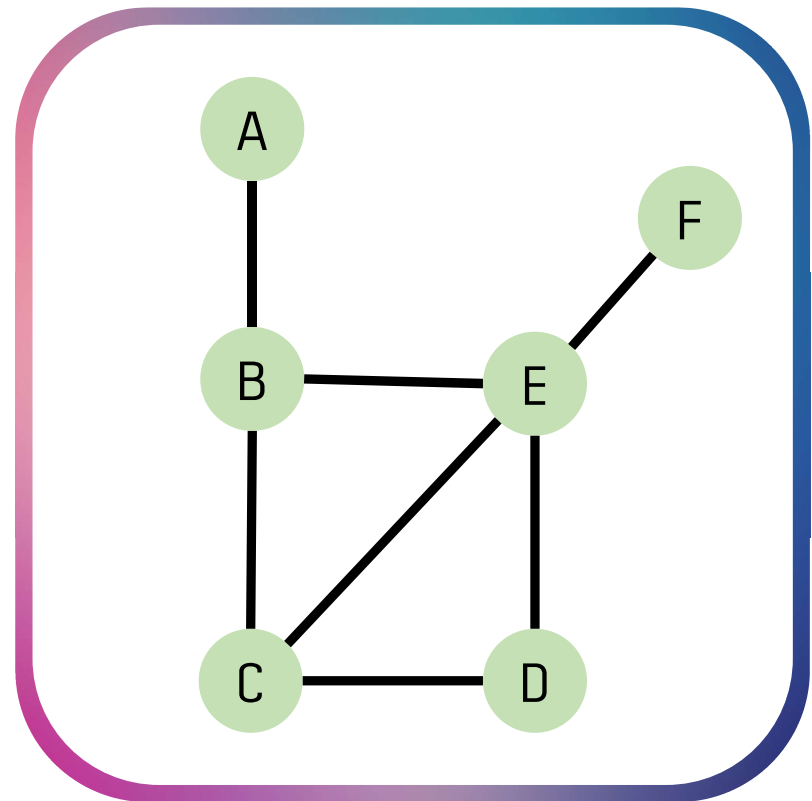
간선(E,edge)

노드들을 연결하는 선





Directed Graph
방향성 O

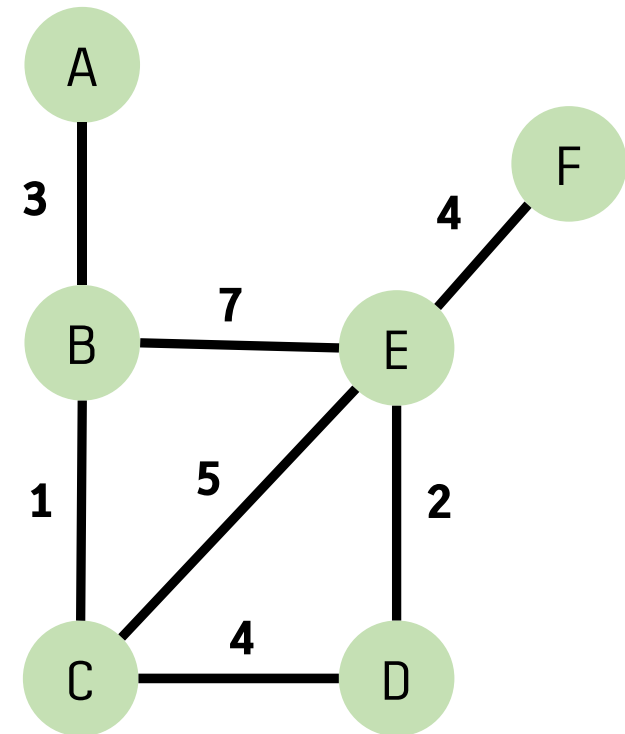


Undirected Graph
방향성 X

Graph(그래프)

가중치 그래프_Network

그래프의 간선에 **비용**이나 **가중치**가 할당된 그래프
Ex) 도로를 지나가는데 드는 비용, 도로의 길이 ...



Graph(그래프)

그래프를 표현할 수 있는 2가지 방법

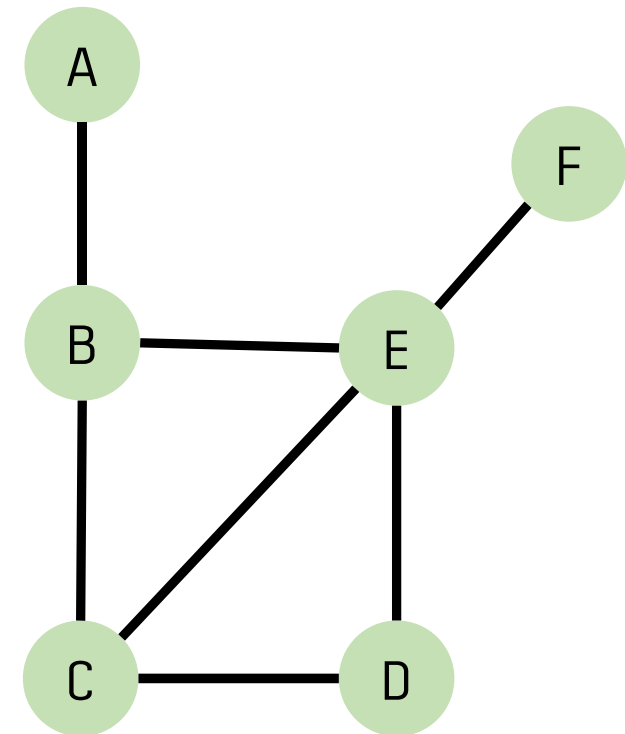
인접 행렬 (배열)

인접 리스트 (연결리스트, 벡터)

Graph_인접행렬 (배열)

간선이 존재하면 “1” 그렇지 않으면 “0”

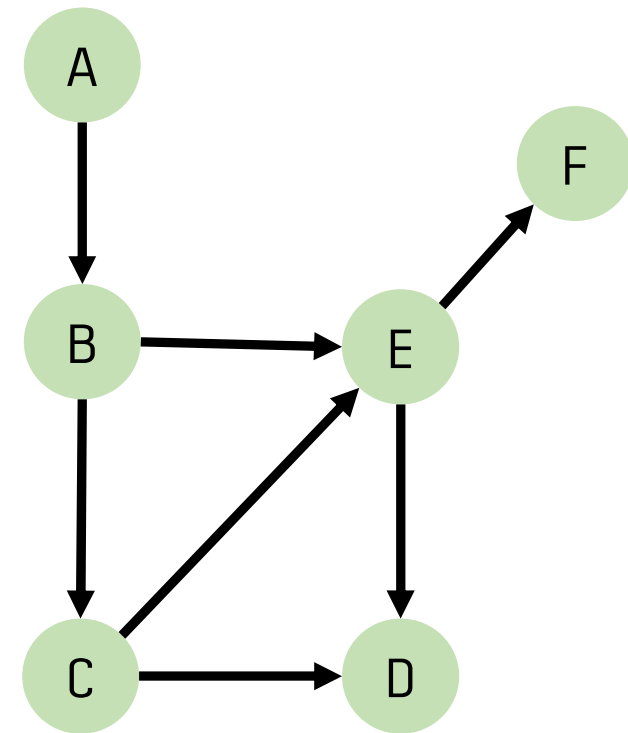
	A	B	C	D	E	F
A	0	1	0	0	0	0
B	1	0	1	0	1	0
C	0	1	0	1	1	0
D	0	0	1	0	1	0
E	0	1	1	1	0	1
F	0	0	0	0	1	0



Graph_인접행렬 (배열)

간선이 존재하면 “1” 그렇지 않으면 “0”

	A	B	C	D	E	F
A	0	1	0	0	0	0
B	0	0	1	0	1	0
C	0	0	0	1	1	0
D	0	0	0	0	0	0
E	0	0	0	1	0	1
F	0	0	0	0	0	0

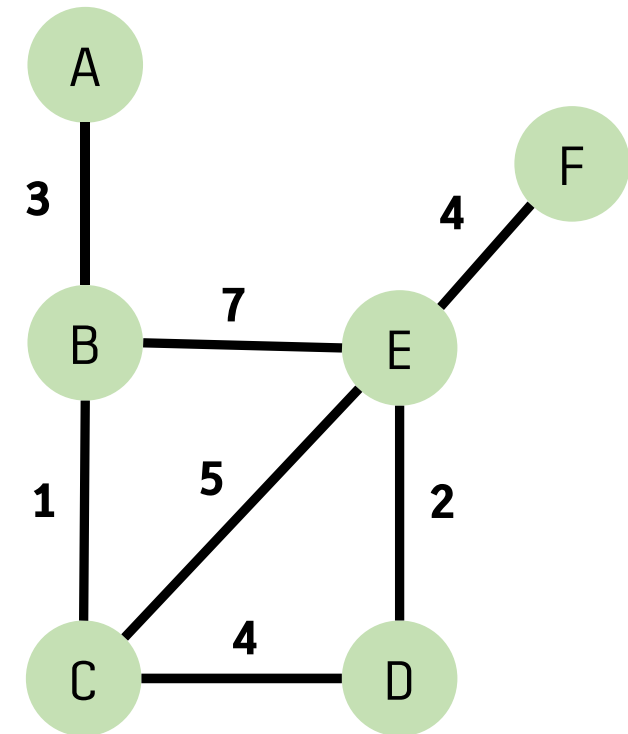


Graph_인접행렬 (배열)

가중치가 존재한다면..

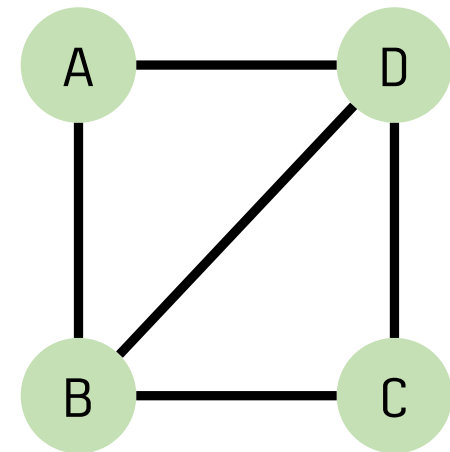
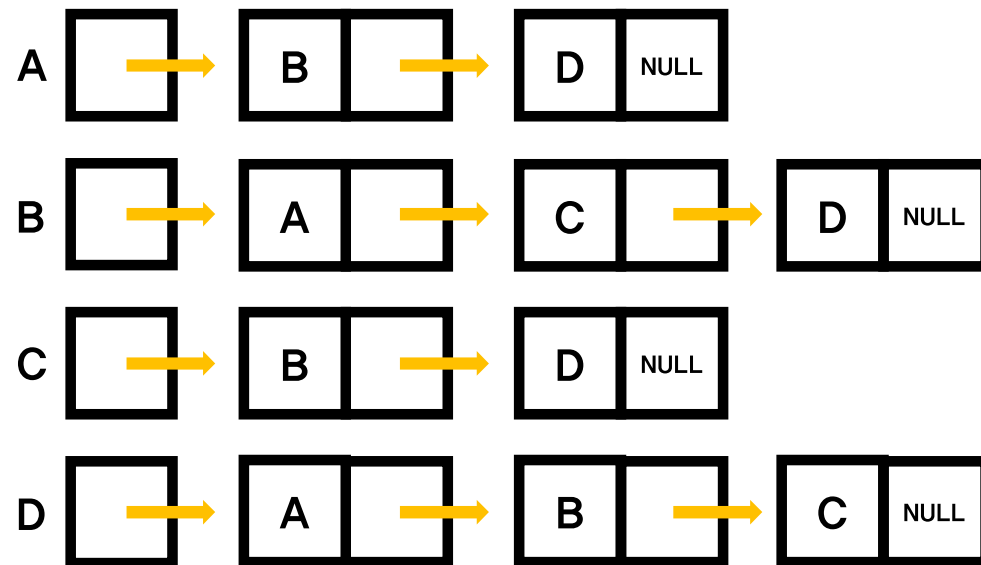
간선이 존재하면 “**가중치**” 그렇지 않으면 “0”

	A	B	C	D	E	F
A	0	3	0	0	0	0
B	3	0	1	0	7	0
C	0	1	0	4	5	0
D	0	0	4	0	2	0
E	0	7	5	2	0	4
F	0	0	0	0	4	0



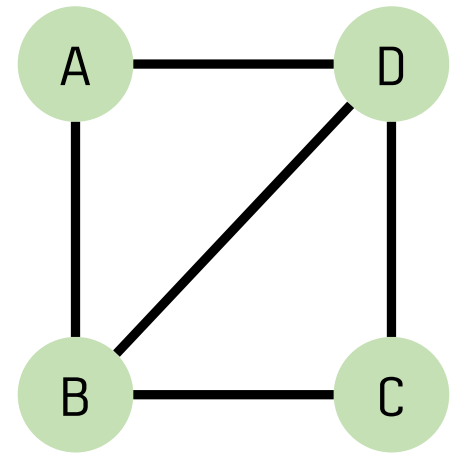
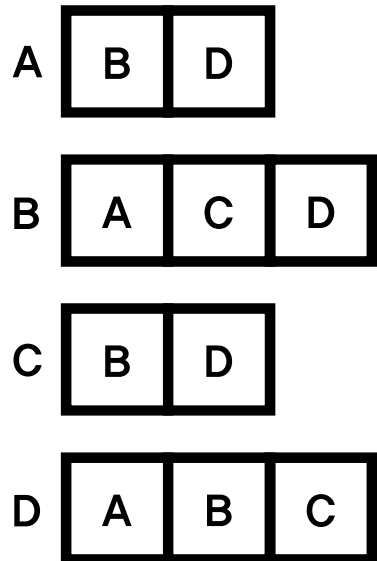
Graph_인접리스트 (연결리스트)

연결리스트를 사용하지만,
알고리즘에서는 잘 사용하지 않는다. **PASS**



Graph_인접리스트 (Vector)

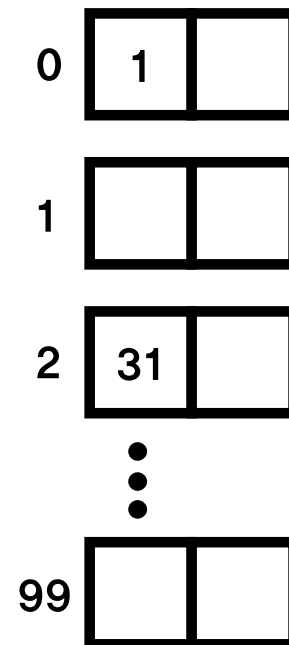
각각의 정점 벡터에 연결된 정점을 **push_back**



Graph_인접리스트 (Vector)

벡터 이용법?

```
1  #include<iostream>
2  [ #include<vector> // 벡터 include
3    using namespace std;
4  int main() {
5      vector<int> vec[100];
6      vec[0].push_back(1); // 0번째 배열의 벡터에 1을 push_back;
7      vec[2].push_back(31); // 2번째 배열의 벡터에 31을 push_back;
8  }
```



Graph(그래프)

그래프의 탐색

그래프의 모든 정점을 방문하는 것으로
DFS, BFS등이 존재한다



Depth-First Search (깊이 우선 탐색)

‘DFS’ 에 대해 알아보자!

DFS_개념

- 현재 정점에서 갈 수 있는 정점들까지 들어가며 탐색

→ 한 방향으로 계속 갈 수 있을 때까지 탐색하다 막히면, 가장 가까운 갈림길로 돌아와서 다른 방향으로 탐색을 진행하는 방법

- 모든 정점을 방문하고 싶을 때 사용
- BFS와 비교하면 코드는 간단하다

DFS_특징

1. 한번 방문한 정점은 다시 방문하지 않는다

→ 정점에 방문하였는지 확인하는 visit 배열을 만들어 구현

2. 재귀함수 또는 스택을 이용하여 구현

3. 표현 방식에 따른 시간복잡도

→ 인접 행렬 : $O(V^2)$

→ 인접 리스트 : $O(V+E)$

간선(E)이 적을 경우 **인접 리스트**를 사용하는 것이 유리하다!

2. 재귀함수 또는 스택을 이용하여 구현

3. 표현 방식에 따른 시간복잡도

$$V^2 > V+E$$

→ 인접 행렬 : $O(V^2)$

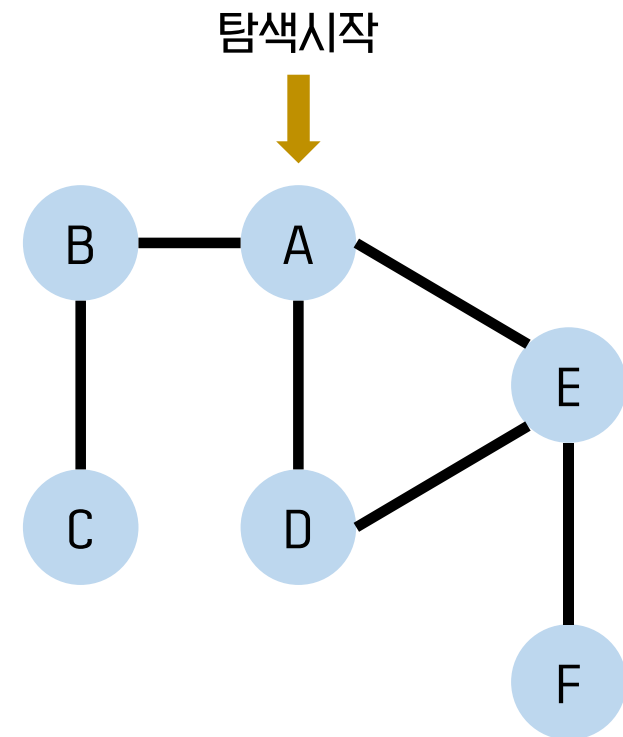
→ 인접 리스트 : $O(V+E)$

DFS_원리

- A ~ F 의 방문을 확인하는 visit배열 생성
→ 정점의 개수 ≤ 배열크기
- 그래프를 입력 받고, 탐색 시작할 정점 결정

	A	B	C	D	E	F
visit	0	0	0	0	0	0

now	
-----	--



DFS_원리

- A부터 탐색시작

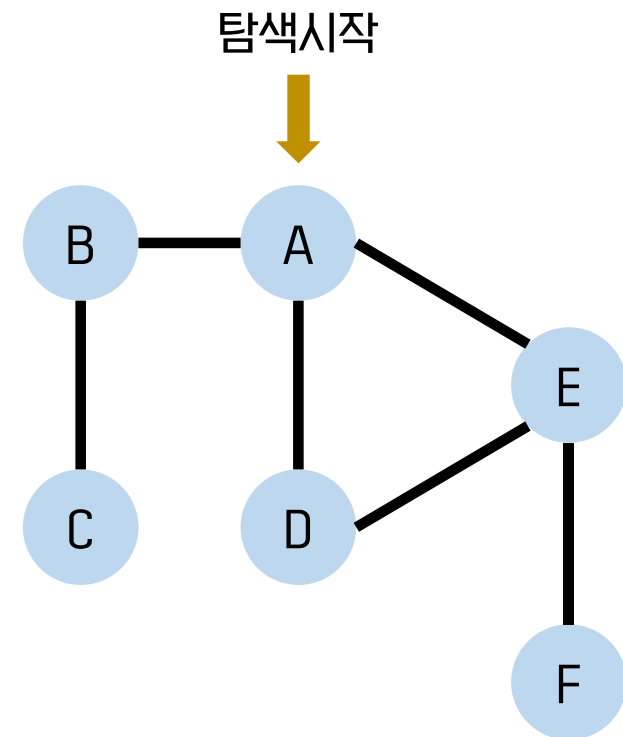
→ 다른 정점부터 시작해도 상관없다.

- 그 이후 번호가 작은 정점부터 탐색

방문 순서

	A	B	C	D	E	F
visit	0	0	0	0	0	0

now	
-----	--



DFS_원리

- A부터 탐색시작

→ 다른 정점부터 시작해도 상관없다.

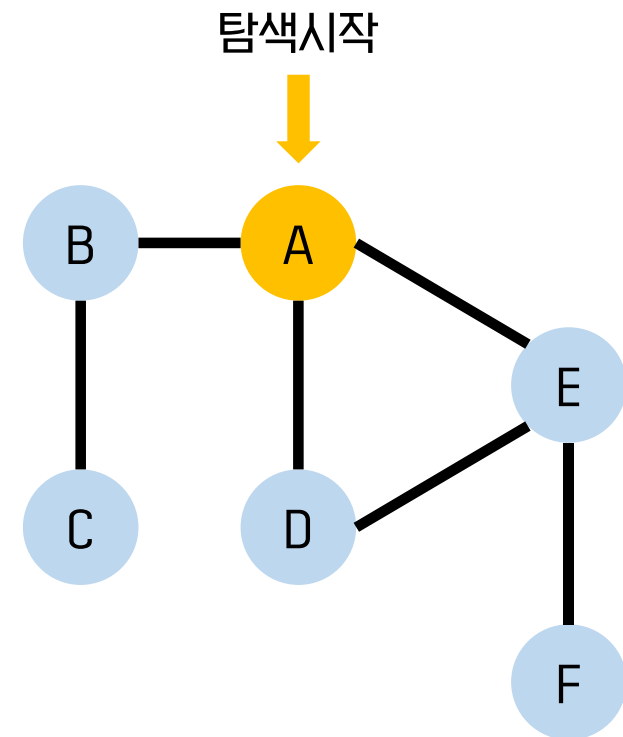
- 그 이후 번호가 작은 정점부터 탐색

방문 순서

A

	A	B	C	D	E	F
visit	1	0	0	0	0	0

now	A
-----	---



DFS_원리

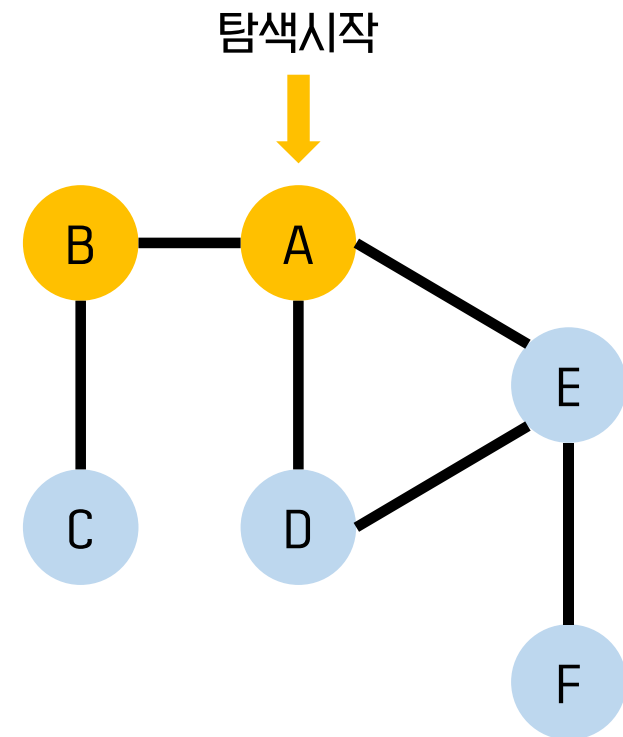
- 그 이후 번호가 작은 정점부터 탐색

방문 순서

A → B

	A	B	C	D	E	F
visit	1	1	0	0	0	0

now	B
-----	---



DFS_원리

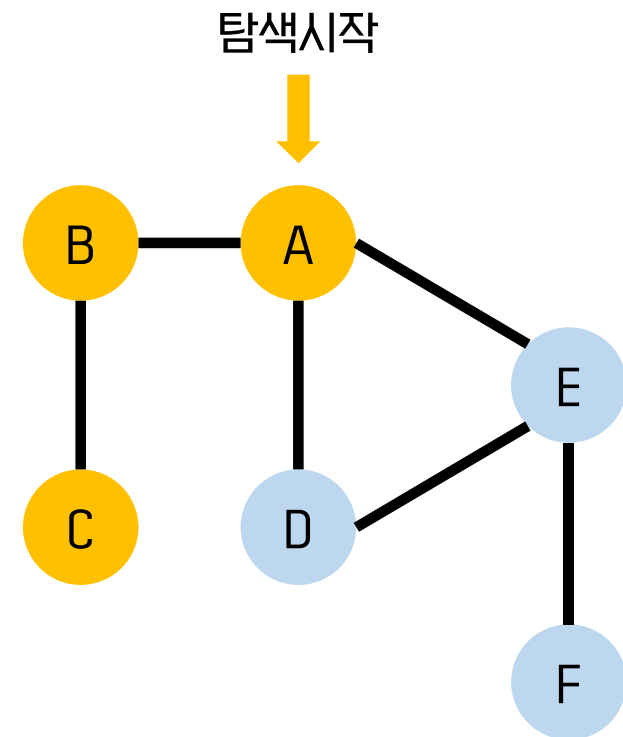
- 그 이후 번호가 작은 정점부터 탐색

방문 순서

A → B → C

	A	B	C	D	E	F
visit	1	1	1	0	0	0

now	C
-----	---



DFS_원리

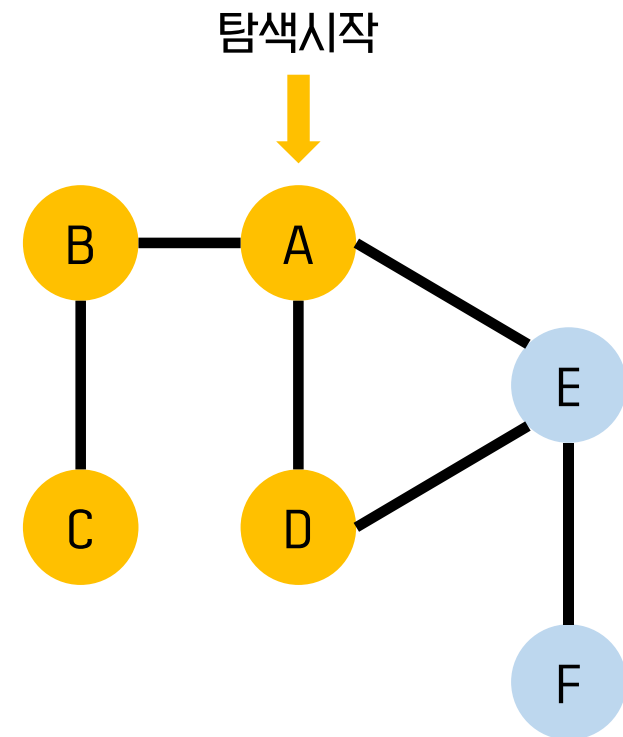
- C에서 더 갈수 있는 정점이 없으므로 돌아오기
→ 연결된 노드 중 방문하지 않은 정점이 있나 확인
- 그 이후 번호가 작은 정점부터 탐색

방문 순서

A → B → C → D

	A	B	C	D	E	F
visit	1	1	1	1	0	0

now	D
-----	---



DFS_원리

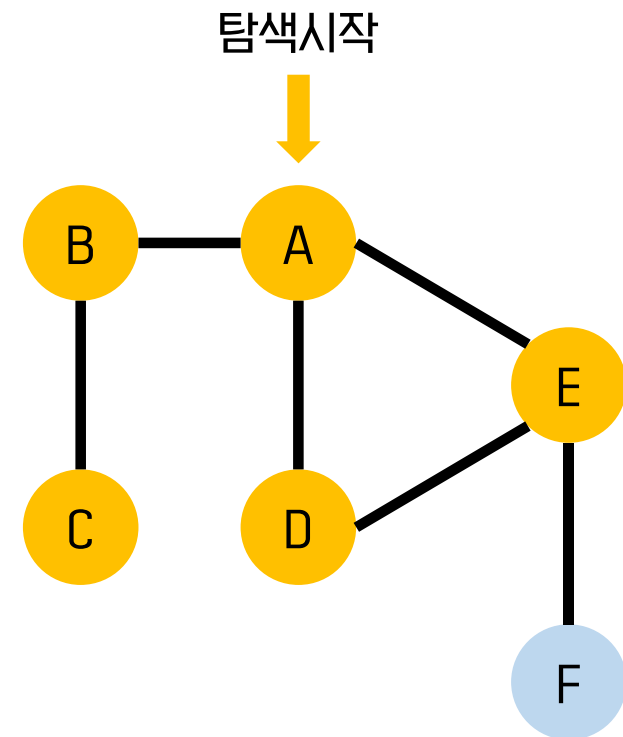
- 그 이후 번호가 작은 정점부터 탐색

방문 순서

$A \rightarrow B \rightarrow C \rightarrow D \rightarrow E$

	A	B	C	D	E	F
visit	1	1	1	1	1	0

now	E
-----	---



DFS_원리

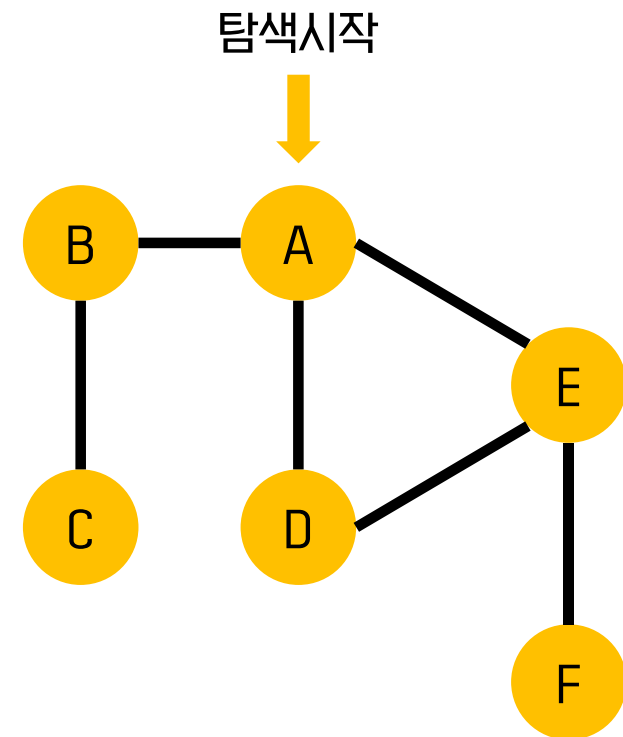
- 그 이후 번호가 작은 정점부터 탐색
- 방문하지 않은 정점이 없으므로 탐색 종료

방문 순서

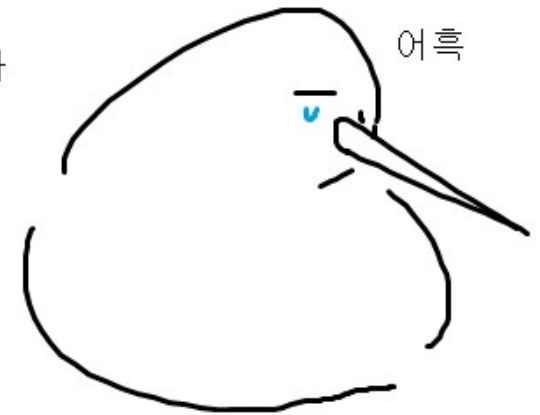
$A \rightarrow B \rightarrow C \rightarrow D \rightarrow E \rightarrow F$

	A	B	C	D	E	F
visit	1	1	1	1	1	1

now	F
-----	---



선생님 전
공부랑 안맞나봐
요,,,



Breadth-First Search (너비 우선 탐색)

‘BFS’ 에 대해 알아보자!

DFS_개념

- 현재 정점에서 인접한 연결된 정점부터 탐색
→ 시작 정점에 가까운 정점부터 넓게 탐색하여 먼 정점은 가장 나중에 방문
- 정점간의 **최단경로** 또는 **임의의 경로**를 찾을 때 사용
- BFS와 비교하면 **코드는 복잡**하다.

DFS_특징

1. 한번 방문한 정점은 다시 방문하지 않는다

→ 정점에 방문하였는지 확인하는 visit 배열을 만들어 구현

2. 큐를 이용하여 구현

3. 표현 방식에 따른 시간복잡도

→ 인접 행렬 : $O(V^2)$

→ 인접 리스트 : $O(V+E)$

+ Dijkstra 알고리즘과 유사

간선(E)이 적을 경우 **인접 리스트**를 사용하는 것이 유리하다!

2. **큐**를 이용하여 구현

3. 표현 방식에 따른 시간복잡도

$$V^2 > V+E$$

→ 인접 행렬 : $O(V^2)$

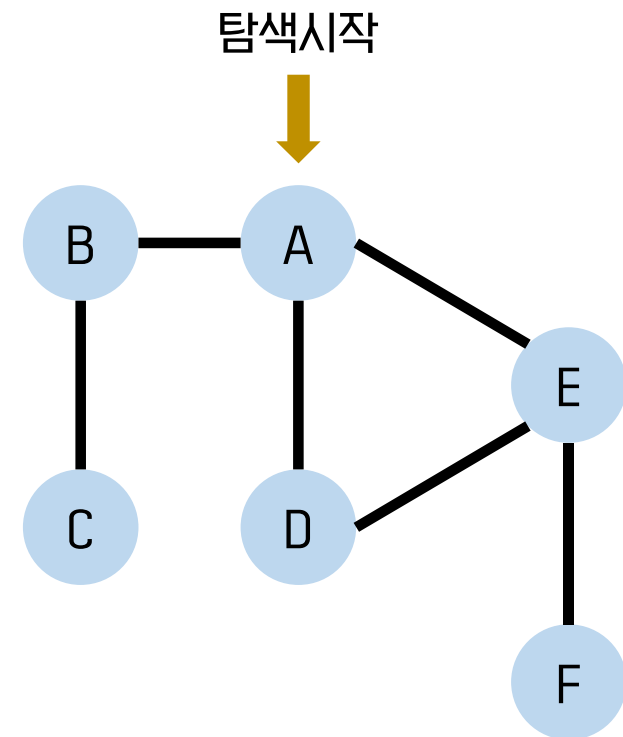
→ 인접 리스트 : $O(V+E)$

+ Dijkstra 알고리즘과 유사

BFS_원리

- A ~ F 의 방문을 확인하는 visit배열 생성
→ 정점의 개수 ≤ 배열크기
- 그래프를 입력 받고, 탐색 시작할 정점 결정

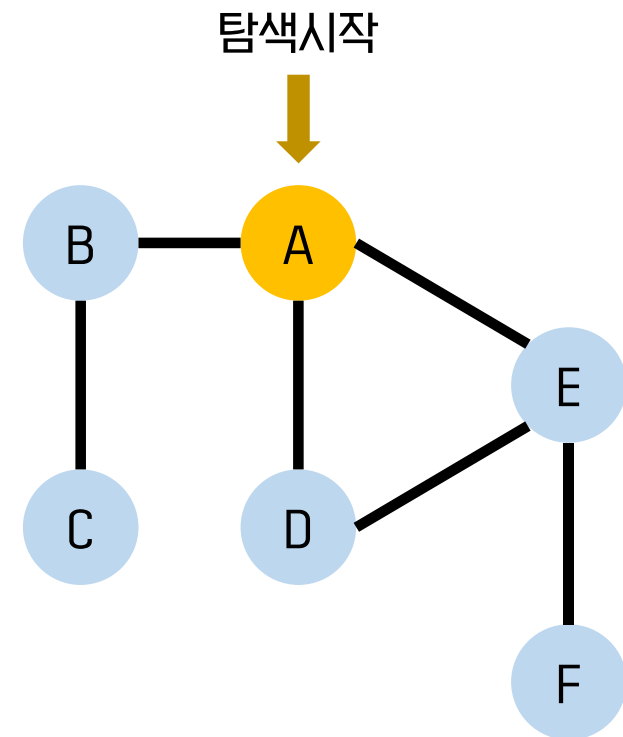
	A	B	C	D	E	F
visit	0	0	0	0	0	0
Queue						



BFS_원리

- 큐에 처음 탐색 시작하는 정점 입력

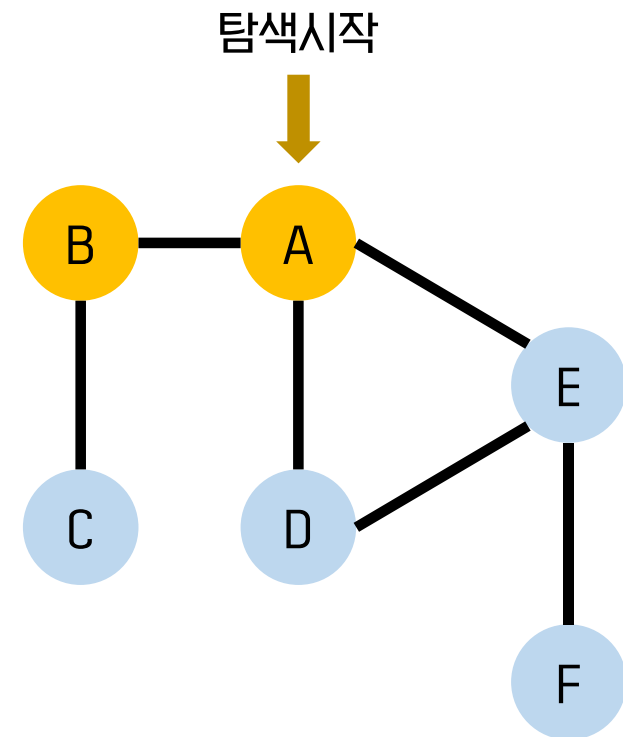
	A	B	C	D	E	F
visit	1	0	0	0	0	0
Queue	A					



BFS_원리

- A와 인접한 방문하지 않은 정점을 큐에 삽입
→ 큐의 순서대로 탐색한다
- 방문한 정점은 큐에서 POP

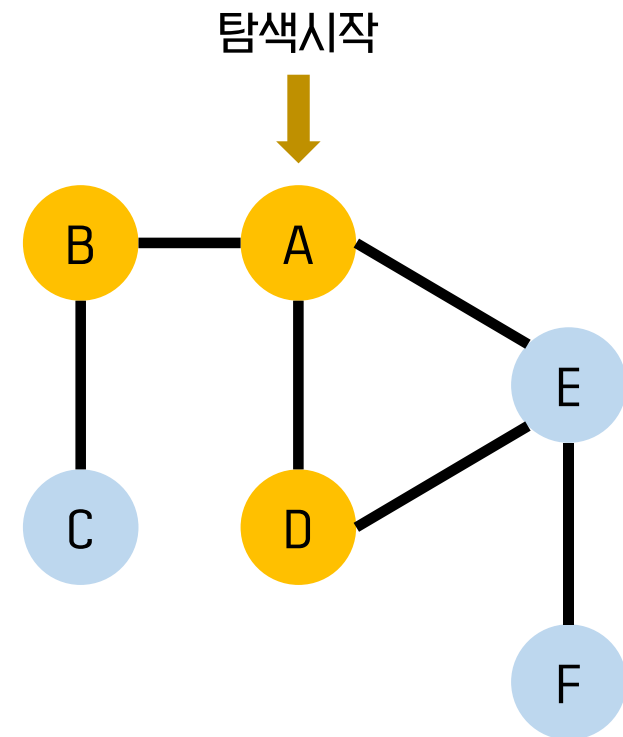
	A	B	C	D	E	F
visit	1	0	0	0	0	0
Queue	B	D	E			



BFS_원리

- B와 인접한 방문하지 않은 정점을 큐에 삽입
→ 큐의 순서대로 탐색한다
- 방문한 정점은 큐에서 POP

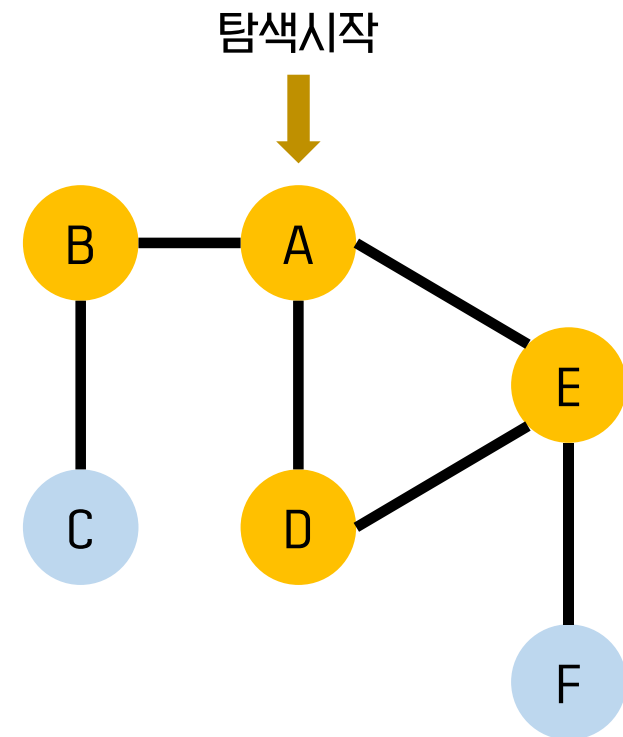
	A	B	C	D	E	F
visit	1	1	0	0	0	0
Queue	D	E	C			



BFS_원리

- D와 인접한 방문하지 않은 정점을 큐에 삽입
→ 큐의 순서대로 탐색한다
- 방문한 정점은 큐에서 POP

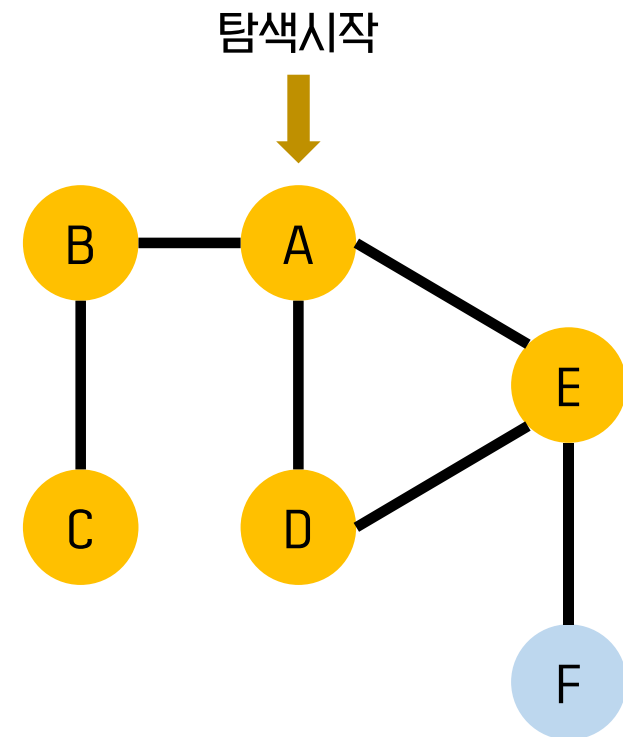
	A	B	C	D	E	F
visit	1	1	0	1	0	0
Queue	E	C	F			



BFS_원리

- E와 인접한 방문하지 않은 정점을 큐에 삽입
→ 큐의 순서대로 탐색한다
- 방문한 정점은 큐에서 POP

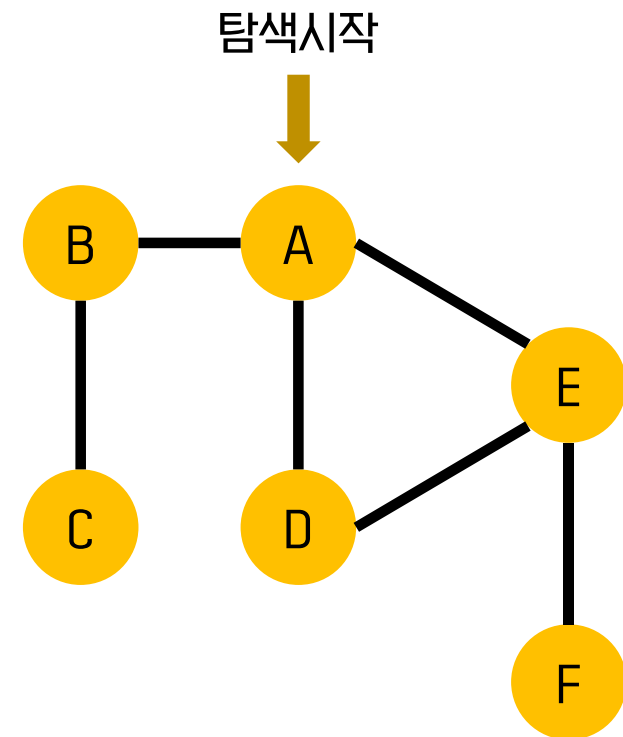
	A	B	C	D	E	F
visit	1	1	0	1	1	0
Queue	C	F				



BFS_원리

- F와 인접한 방문하지 않은 정점을 큐에 삽입
→ 큐의 순서대로 탐색한다
- 방문한 정점은 큐에서 POP

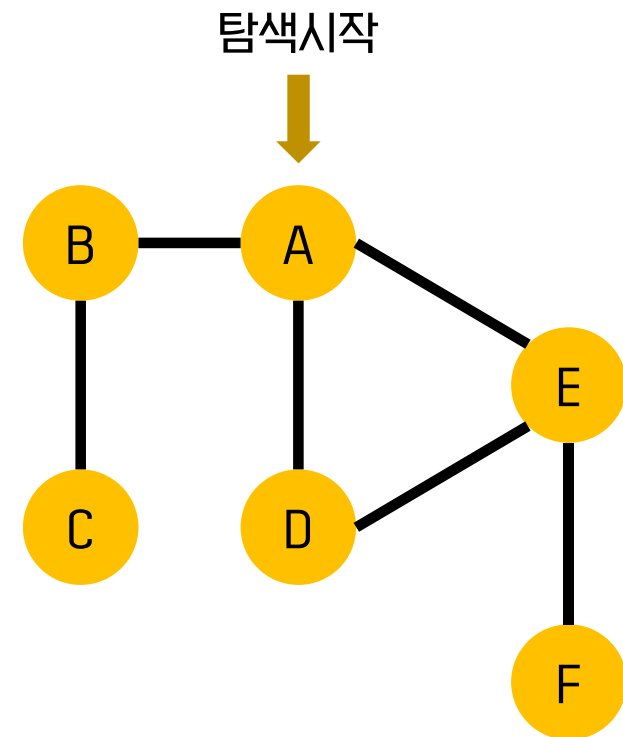
	A	B	C	D	E	F
visit	1	1	1	1	1	0
Queue	F					



BFS_원리

- 모든 정점을 방문했으면 탐색 종료

	A	B	C	D	E	F
visit	1	1	1	1	1	1
Queue						

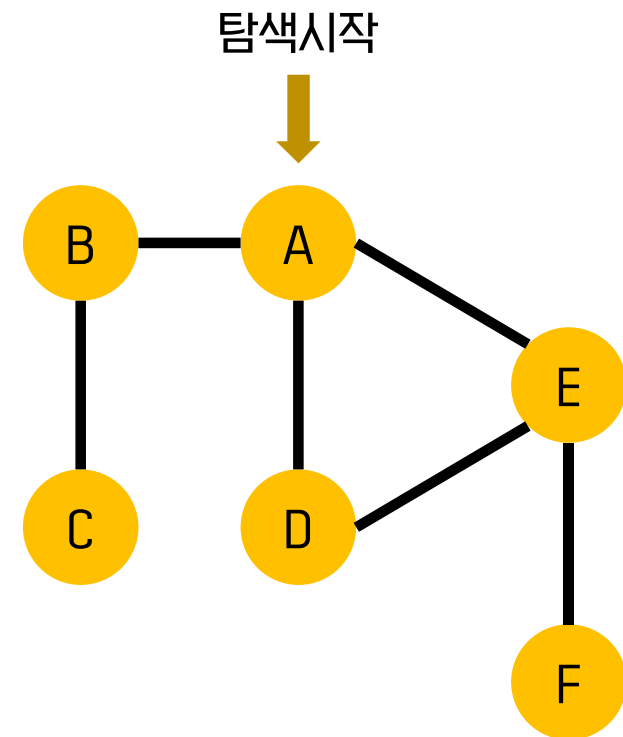


BFS_원리

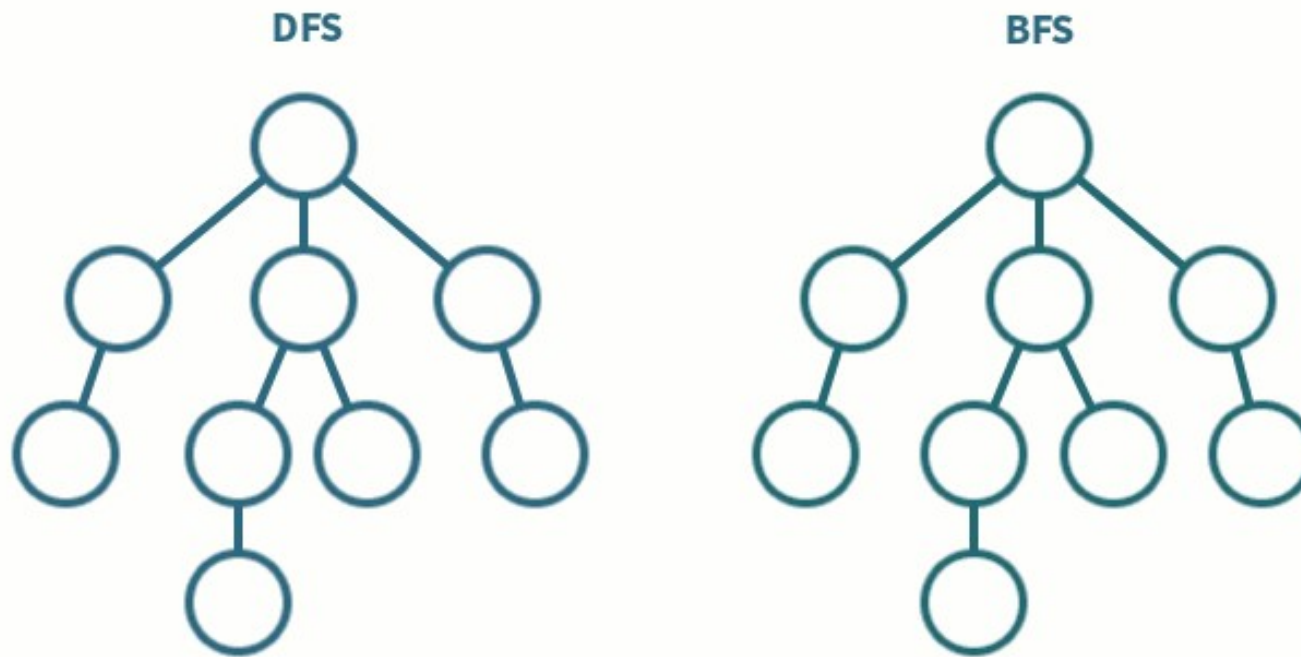
탐색 순서

$A \rightarrow B \rightarrow D \rightarrow E \rightarrow C \rightarrow F$

	A	B	C	D	E	F
visit	1	1	1	1	1	1
Queue						



한눈에 보는 DFS, BFS의 탐색차이



https://twpower.github.io/images/20180114_73/dfs-bfs-example.gif

일단 모르겠고 게
임 한 판만 하자



이제
이제

DFS, BFS 구현

DFS, BFS 하루 만에 뽀개기!

DFS_구현(벡터)

```
void dfs(int x) // main에서 처음 탐색할 정점 입력
{
    visit_dfs[x] = true; // 정점 x 를 방문했으므로 true
    printf("%d ", x); // 방문한 정점 x 를 출력
    for (int i = 0; i < vec[x].size(); i++) // 정점x에 연결된 정점만큼
        if (!visit_dfs[vec[x][i]]) // 방문하지 않은 정점이면
            dfs(vec[x][i]); // dfs 실행 (재귀함수이용)
}
```

BFS_구현(벡터)

```
void bfs()
{
    q.push(v); // v = 처음 시작할 정점
    visit_bfs[v] = true; // 방문했다고 표시
    while (!q.empty()) { // q 에 아무것도 없을때까지 (연결된 정점 X)
        int now = q.front(); // q의 첫번째 요소 = now
        q.pop(); // 방문한 정점은 pop
        printf("%d ", now); // 방문한 정점 출력
        for (int i = 0; i < vec[now].size(); i++) { // 현재 정점에 연결된 정점 개수만큼
            if (!visit_bfs[vec[now][i]]) { //방문하지 않았으면
                q.push(vec[now][i]); // 큐에 그 정점 push
                visit_bfs[vec[now][i]] = true; // 그 정점을 방문했다고 표시
            }
        }
    }
}
```

**만약 visit_bfs를 정점을 pop할 때
바꾼다면?**

- **Queue에 여러 번 입력되어 중복
방문이 될 수 있다!**

연습

1260번 - DFS와 BFS
11724번 - 연결 요소의 개수

오늘 숙제 안해두 내
일의 내가 하지 않을
까??



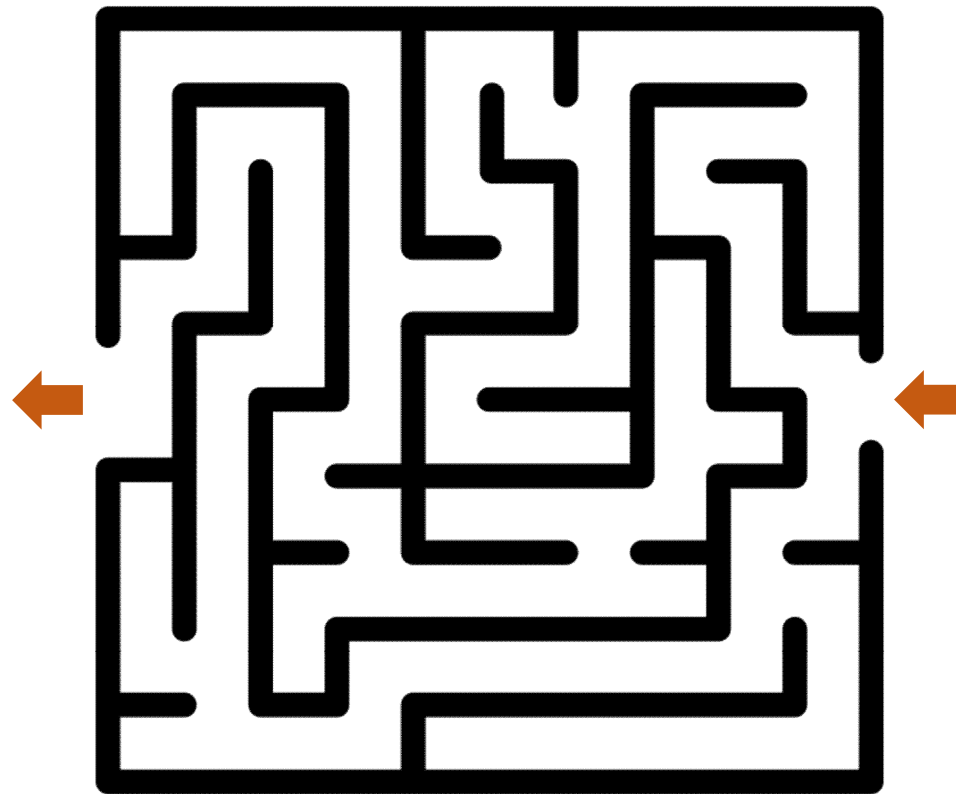
모야,,, 제
주말 돌려주세요;



시간순삭 미로찾기

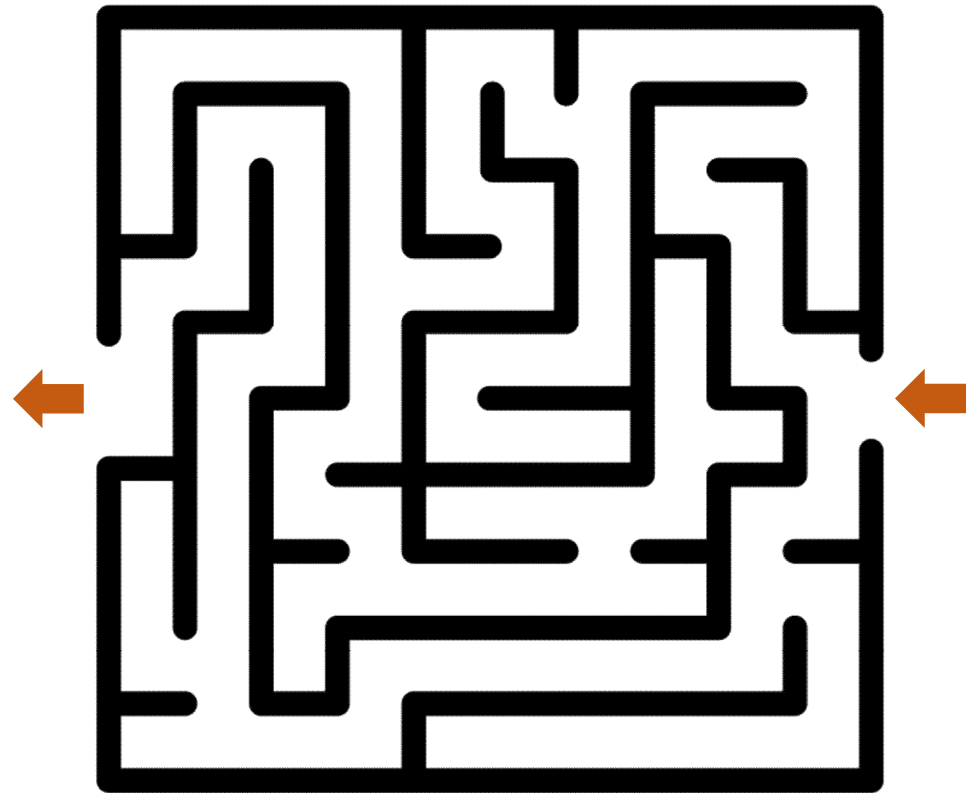
주말엔 벚꽃구경 말고 미로 구경하세요~

미로 찾기(2178번)



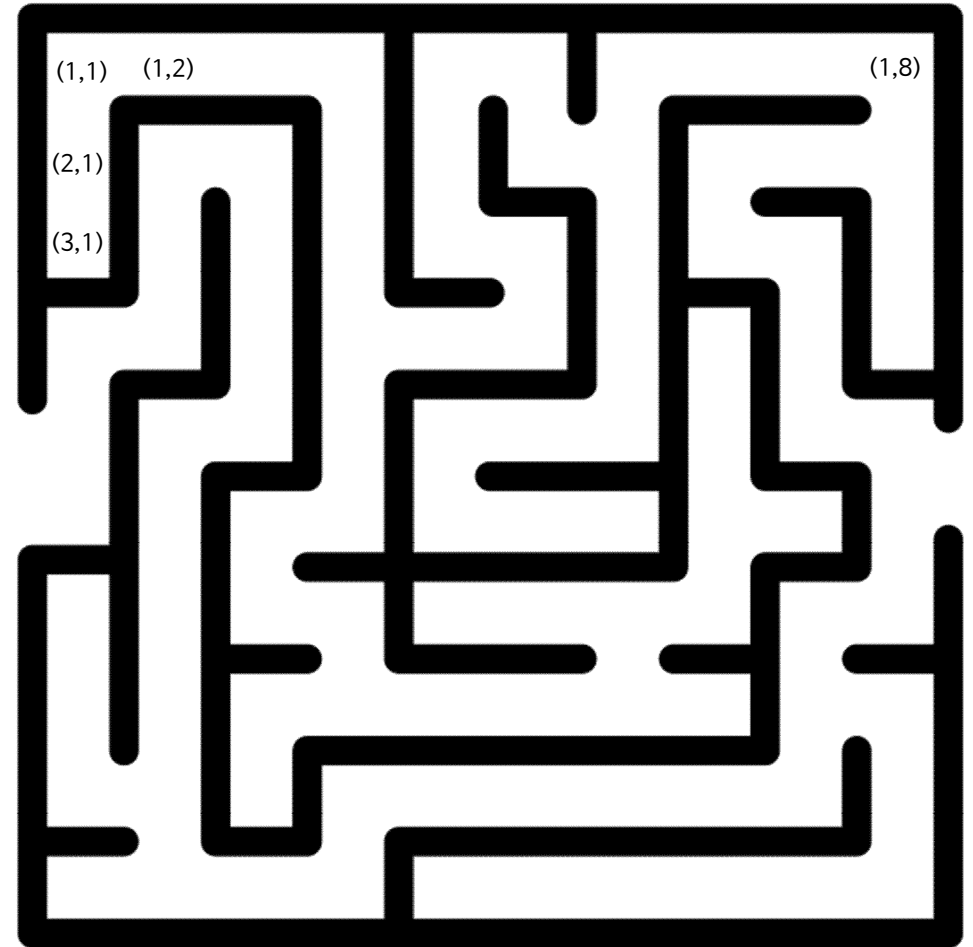
미로 찾기(2178번)_기본

- 정점 대신 (x,y) 좌표
- bfs는 `pair<int, int>` 사용 하여 queue 이용
- 지도이므로 배열로 받아서 탐색
- visit은 미로의 모든 좌표(2차원)



미로 찾기(2178번)_기본

- **한자리 수로 scanf 하기**
→ `scanf("%1d", &MAP[i][j])`
- **최단거리를 구하기 위한 식**
→ `dis[y][x]` 배열로 각 좌표까지
최단거리 구하기
- **상,하,좌,우를 나타내기**
→ `int dx[4] = {-1, 1, 0, 0}`
`int dy[4] = {0, 0, -1, 1}`



미로 찾기_DFS 이용

```
void dfs(int x, int y) { //x,y 좌표
    visit_dfs[y][x] = true; // 이차원 배열에서는 앞에가 y
    for (int i = 0; i < 4; i++) { // 상하좌우 확인
        int nx = x + dx[i]; // x값 변화
        int ny = y + dy[i]; // y값 변화
        if (visit_dfs[ny][nx] == false) { // 방문하지 않았다면 + 지도 안쪽인지 확인도 추가해야함!
            dfs(ny, nx); //dfs 실행
        }
    }
}
```

최단거리를 구하는 것이므로 bfs가 좋다

지도에서 벗어나는지 확인하는 법?

미로 찾기_BFS 이용

```
void bfs() {
    q.push({ 1,1 }); // 1,1 에서부터 미로탐색 시작
    while (!q.empty()) {
        int size = q.size(); // q의 값 = 현재 연결된 정점 개수
        int x = q.front().first; // x값
        int y = q.front().second; // y값
        q.pop();
        visit[x][y] = true; // 방문함 표시
        for (int i = 0; i < size; i++) {
            for (int j = 0; j < 4; j++) { // 상하좌우
                int nx = x + dx[j];
                int ny = y + dy[j];
                if (MAP[ny][nx] == 1 && dis[ny][nx] > dis[y][x] + 1) { // map 안쪽이고, 갈수 있는 길이고, 거리가 더 적으면
                    q.push({ nx,ny });
                    dis[ny][nx] = dis[y][x] + 1; // 걸린 거리 입력
                }
            }
        }
    }
}
```

dis 배열은 INF값으로 초기화

지도에서 벗어나는지 확인하는 법?

미로 찾기_memset

- **memset을 사용하여 배열을 -1로 초기화**

```
#include<string.h>
main() {
    memset(MAP, -1, sizeof(MAP));
}
```

[illegible]

미로 찾기 _memset

- 지도를 입력 받고 bfs 탐색 실행

→ if문에 -1, 0 이 아닌 조건을 추가

→ MAP은 (1,1) 부터 입력받기

-1	-1	-1	-1	-1	-1	-1	-1
-1	1	0	0	0	0	0	-1
-1	1	0	0	1	1	1	-1
-1	1	0	0	1	0	1	-1
-1	1	0	0	1	0	1	-1
-1	1	1	1	1	0	1	-1
-1	1	0	1	1	0	1	-1
-1	-1	-1	-1	-1	-1	-1	-1

미로 찾기_지도의 범위

- 지도의 크기가 나와있다면

→ if문에 지도의 크기보다 x, y 값이 작다는 조건 추가

연습

2178번 - 미로탐색_(최단거리)
1012번 - 유기농 배추
7576 - 토마토

내일은 내일의해가
떨것이다.



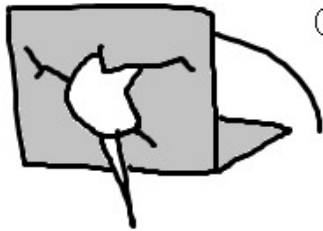
그리고 오늘 할 일들은 내일
의 내가 할것이다.

연습+++

2644번 - 촌수계산
2583번 - 영역 구하기
7562번 - 나이트의 이동

아니 과제를 이렇게
많이내쫓심면
어떡케요!!!!!!!!!!!!!!

노트본 우작꾼





머릿속에서 내적갈 등 하고잇음

BackTracking (feat.dfs)

마지막 'BackTracking' 에 대해 알아보자! (너무 많아..)

BackTracking_개념

모든 경우의 수를 전부 고려하는 알고리즘

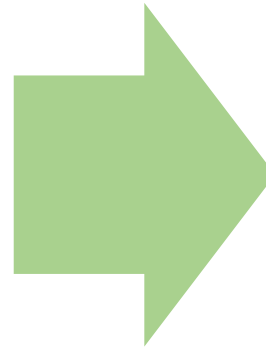
= ~~Brute-force?~~

BUT

**(조건을 충족하는) 모든 경우의 수를 전부
고려하는 트리 탐색 알고리즘**

BackTracking_개념

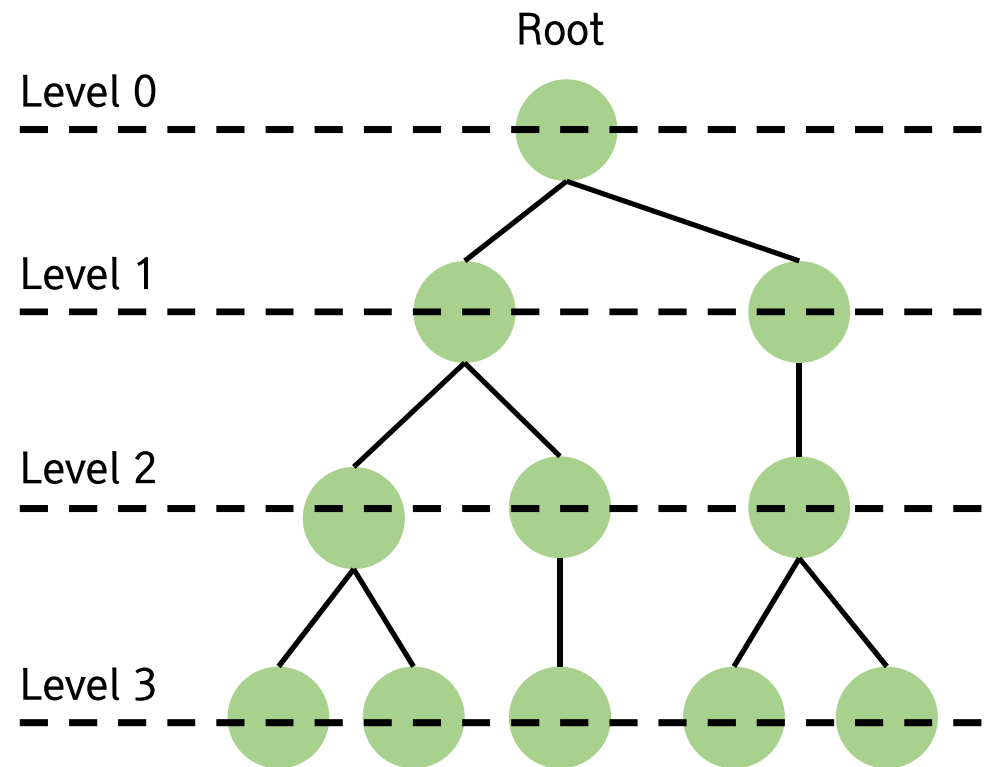
(조건을 충족하는) 모든
경우의 수를 전부 고려하
는 트리 탐색 알고리즘



최적해(정답)

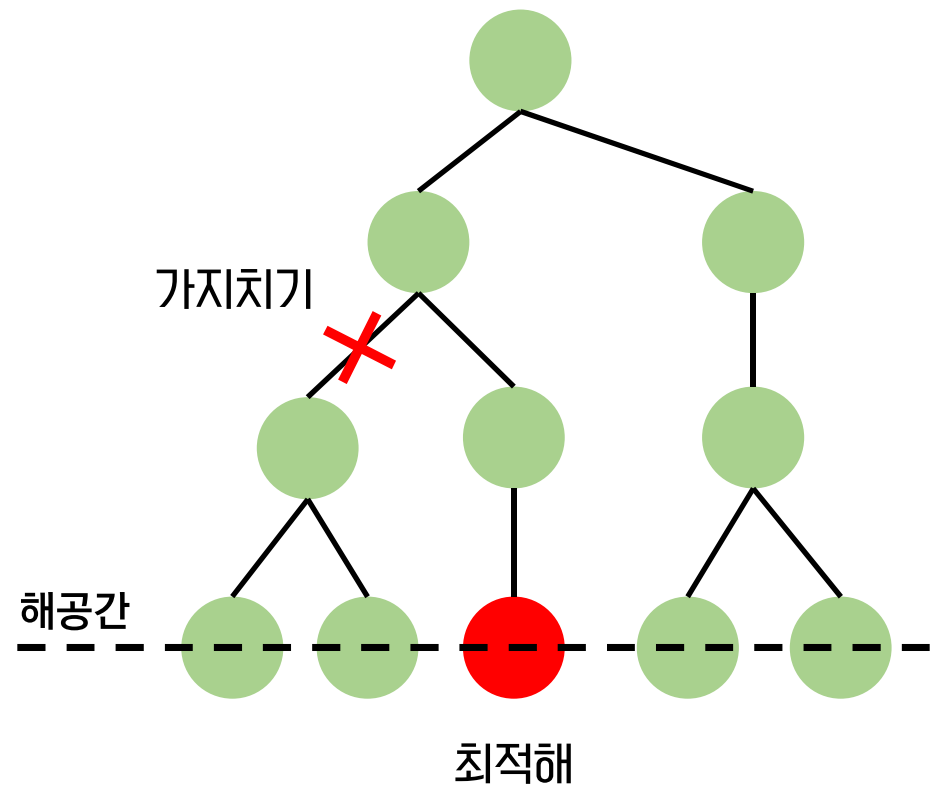
Tree_개념

- **노드**로 이루어진 자료구조
- **그래프**의 일종입니다
- 사이클이 **없**는 하나의 **방향성이 있**는 그래프
- 각 노드는 **깊이(level)**을 가지고 Root 노드는 level 0 이다
- 현재 노드와 연결된 레벨이 낮은 노드는 **부모 노드**라고 부른다



BackTracking_개념

최적해(답)을 향해 **가지를 뺏어나가며**,
가능성이 없는 해(가지)는 **제거**하며
해중에 최적 해를 탐색해나간다



BackTracking_질문

1. 해는 어떻게 **정점(노드)**으로 만드나요?
2. 트리는 어떻게 **구성**하나요?
3. 트리의 **순회(탐색)**은 어떻게 하나요?

질문

1. 해는 어떻게 정점(노드)으로 만드나요?

해는 주로 tuple을 사용해 만들지만, 다른 방법도 가능합니다

Ex) $(x_1, x_2, x_3, x_4 \dots, x_{n-1}, x_n)$

질문

2. 트리는 어떻게 구성하나요?

tuple의 개수_(n)만큼 트리의 깊이를 정합니다.

즉, x_n 까지 확정된 tuple은 깊이(Level)가 n 인 곳에 위치

질문

3. 트리의 순회(탐색)은 어떻게 하나요?

트리(그래프)를 탐색하는 법인 DFS를 사용합니다

- BFS를 사용하지 않는 이유

BFS = 같은 Level을 가지는 노드의 개수만큼 메모리가 필요

DFS = 트리의 Level만큼 메모리가 필요

노드의 개수가 K일 때 공간복잡도가

$BFS = O(K^N) > DFS = O(\log_2 N)$

이므로 DFS를 사용합니다

BackTracking_예시

문제를 통해 원리를 공부해 봅시다!

오른차순으로 주어지는 이 집합의 부분집합의 원소들의 합이 sum이 되도록 만들어보세요

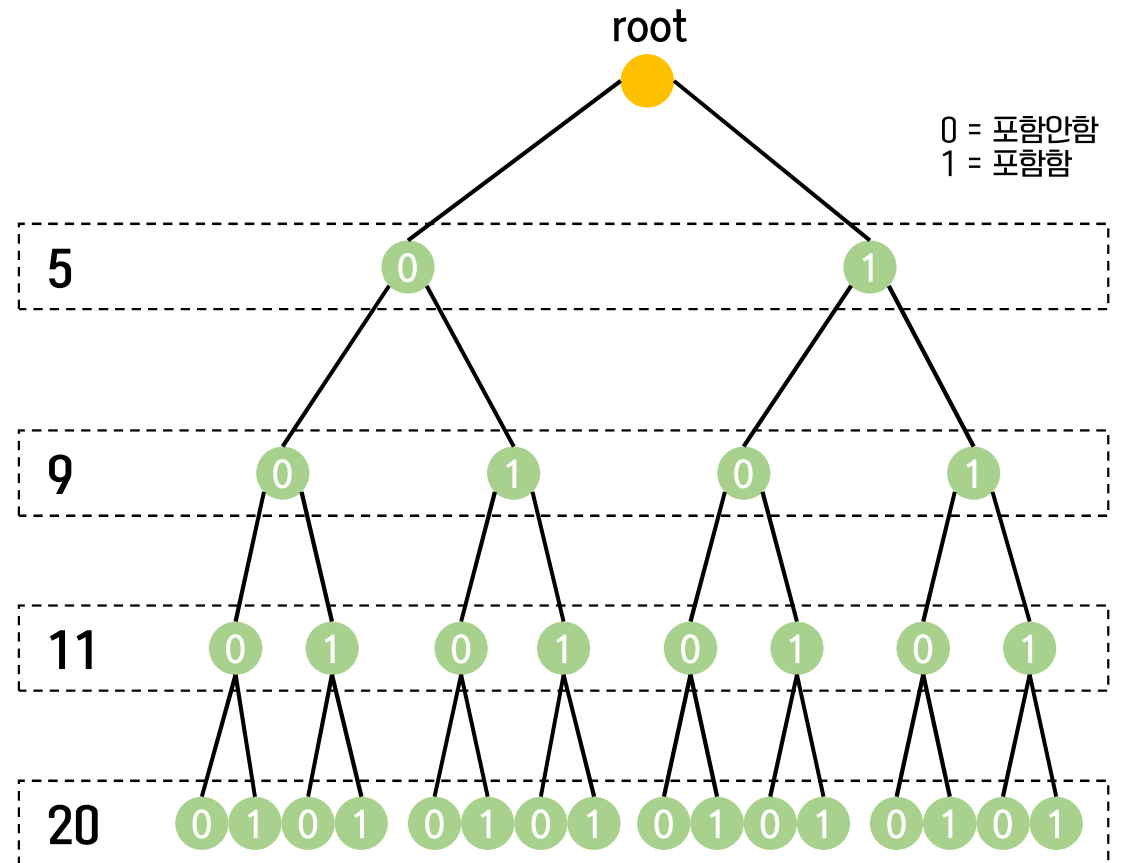
{5, 9, 11, 20}

sum = 25

답 = {5, 9, 11}, {5, 20}

BackTracking_예시

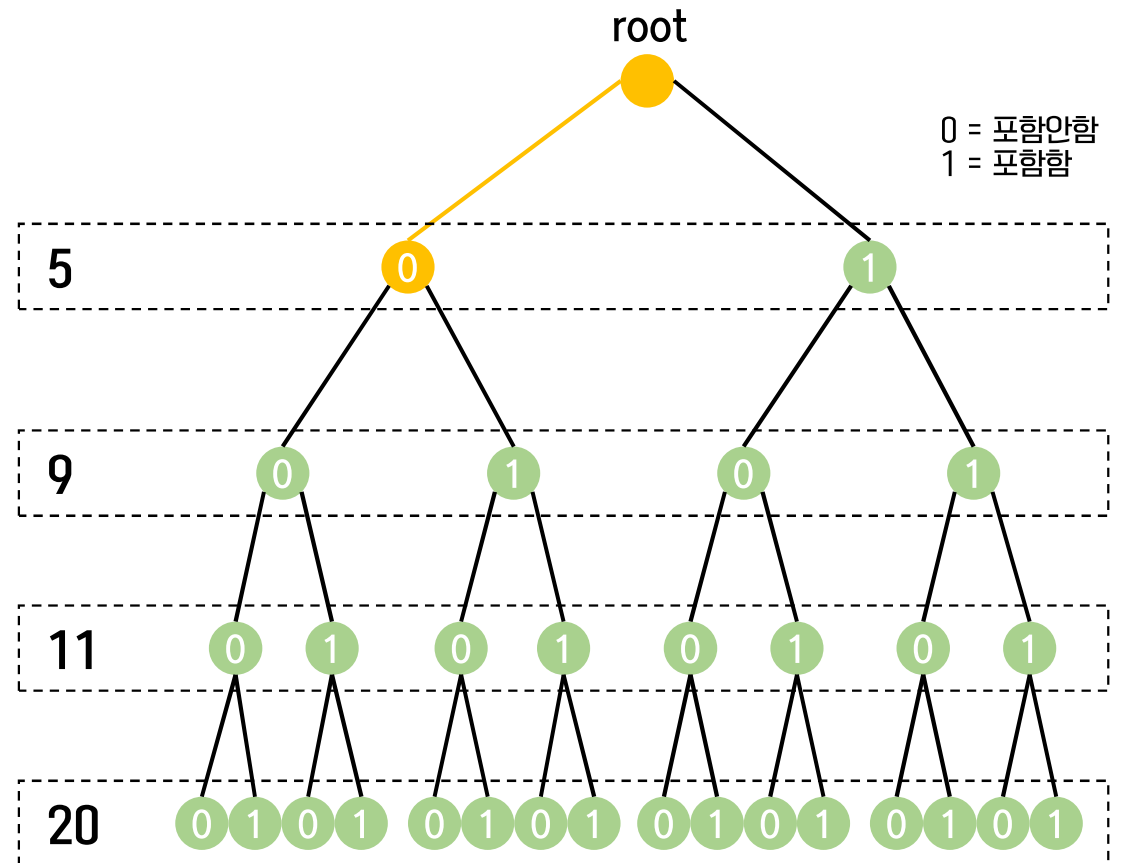
- 각 노드는 그 숫자의 유무
- 숫자의 개수가 4개이므로 깊이 = 4
- Root 부터 dfs 시작



BackTracking_예시

- 5를 포함하지 않으면 25보다 작나?
→ 조건에 성립하므로 계속 순회
- 순회 할 때마다 조건과 비교

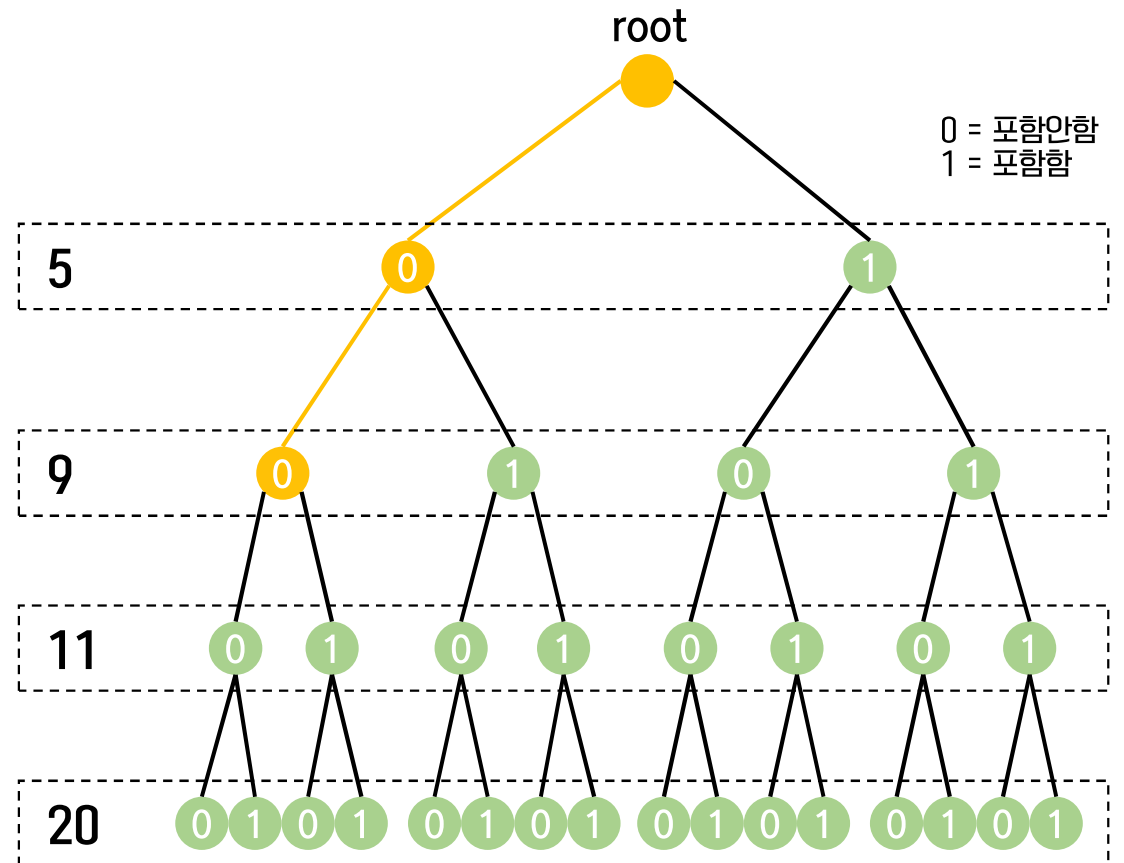
Tuple(0)



BackTracking_예시

- 9를 포함하지 않으면 25보다 작나?
→ 조건에 성립하므로 계속 순회
- 순회 할 때마다 조건과 비교

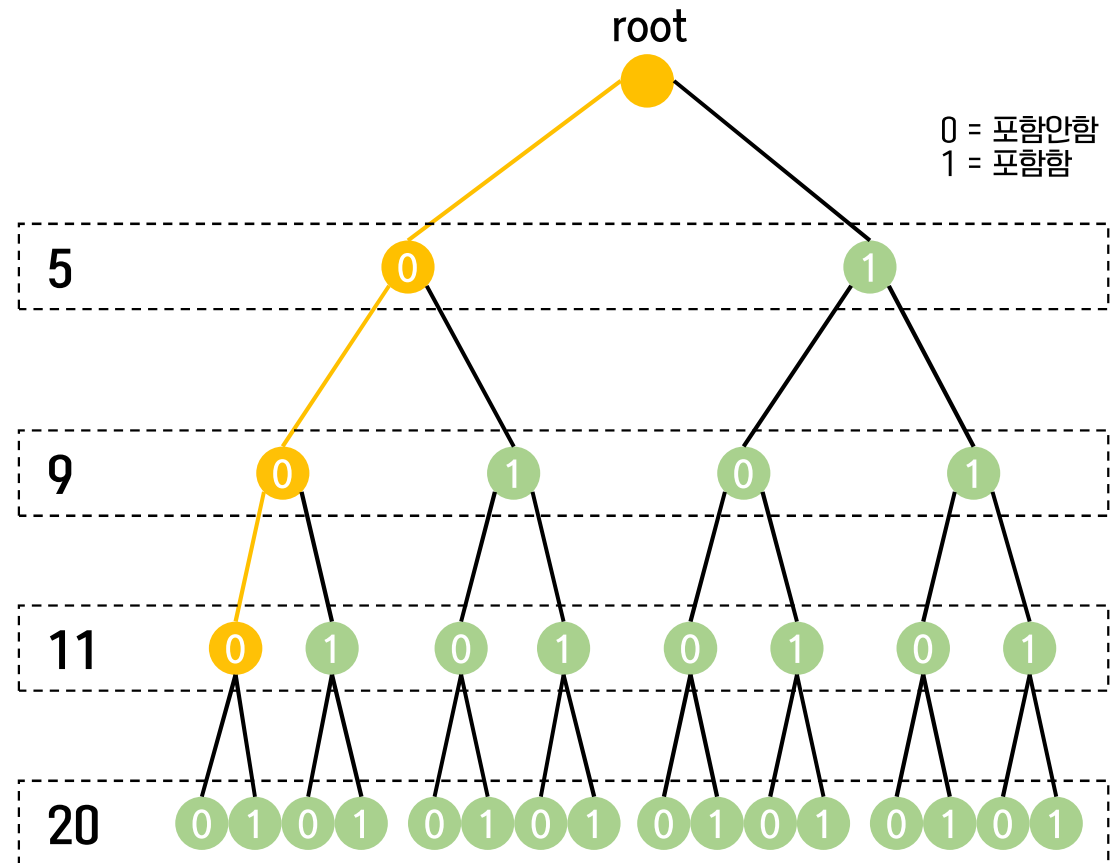
Tuple(0, 0)



BackTracking_예시

- 11를 포함하지 않으면 25보다 작나?
→ 조건에 성립하므로 계속 순회
- 순회 할 때마다 조건과 비교

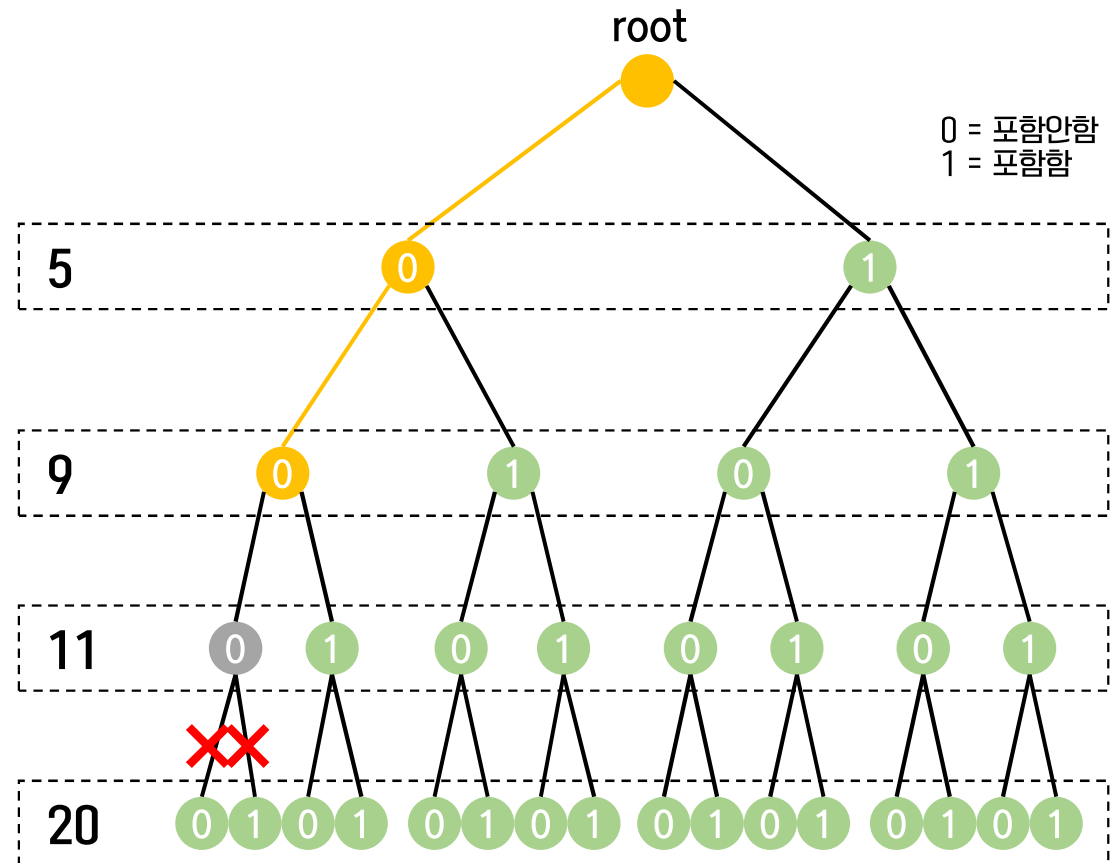
Tuple(0, 0, 0)



BackTracking_예시

- 남은 Level의 합이 250이상인가?
→ 성립하지 않으므로 부모 노드로 돌아감
- 순회 할 때마다 조건과 비교

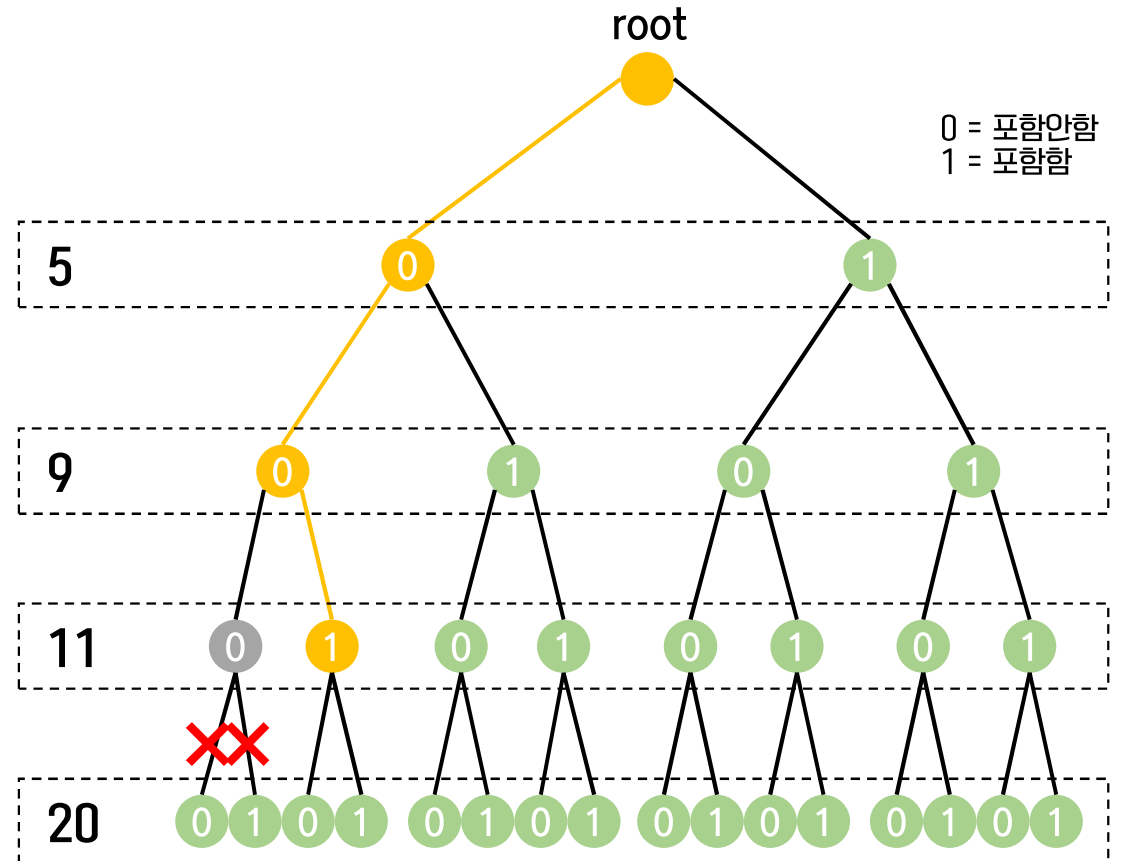
Tuple(0, 0)



BackTracking_예시

- 11을 포함해도 25보다 작은가?
→ 조건 성립하므로 계속 순회
- 순회 할 때마다 조건과 비교

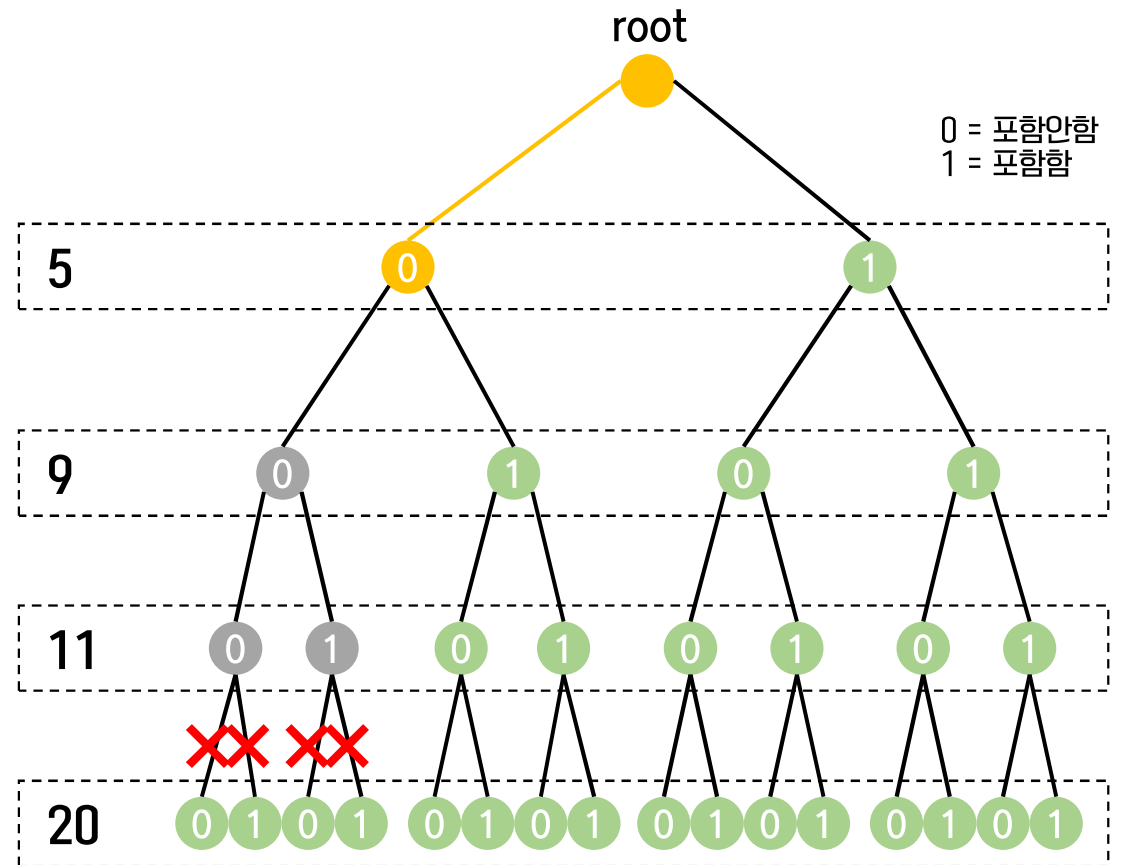
Tuple(0, 0, 1)



BackTracking_예시

- 지금 까지의 방식으로 계속 반복

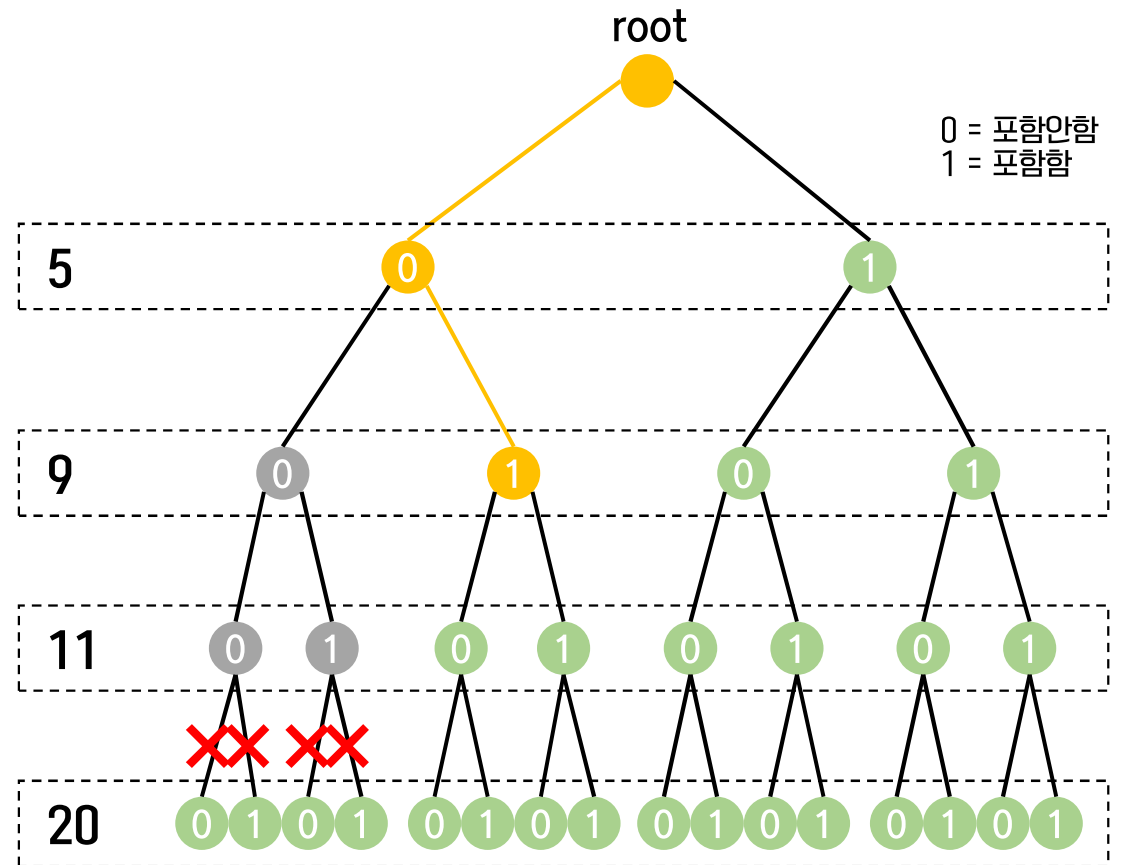
Tuple(0)



BackTracking_예시

- 지금 까지의 방식으로 계속 반복

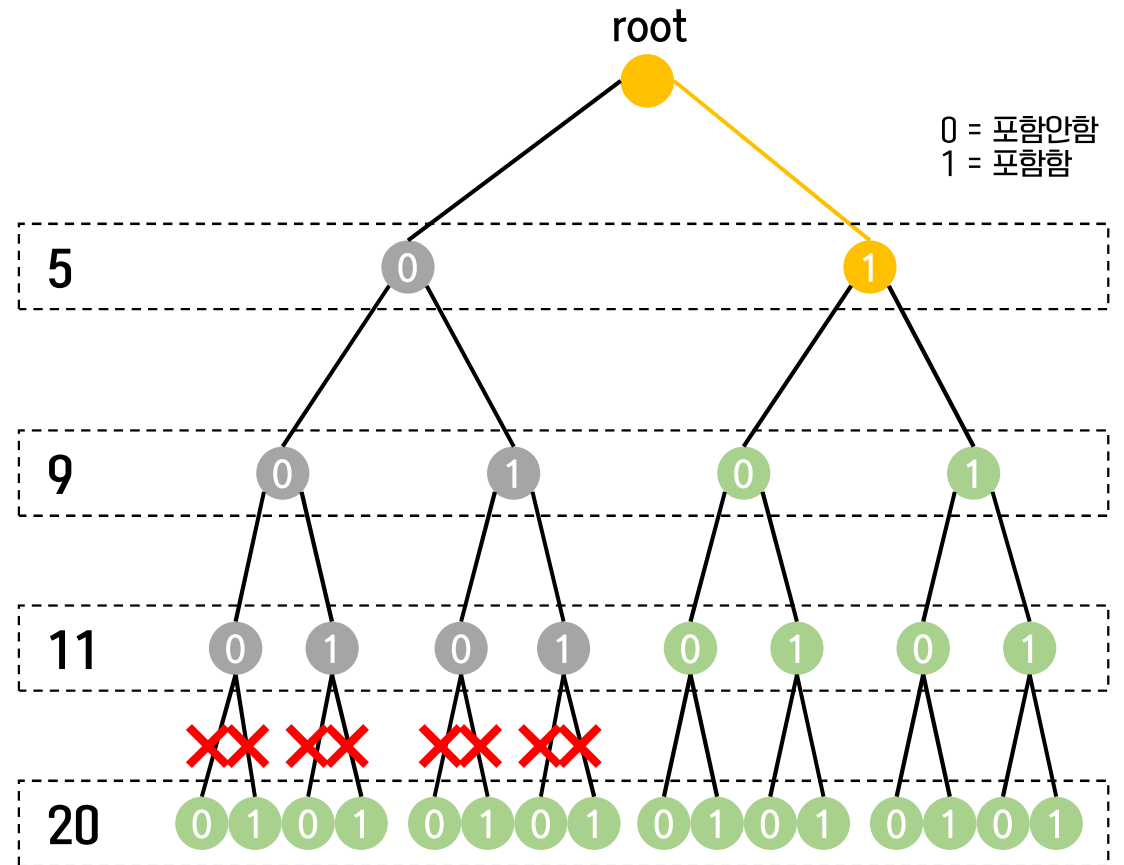
Tuple(0,1)



BackTracking_예시

- 지금 까지의 방식으로 계속 반복

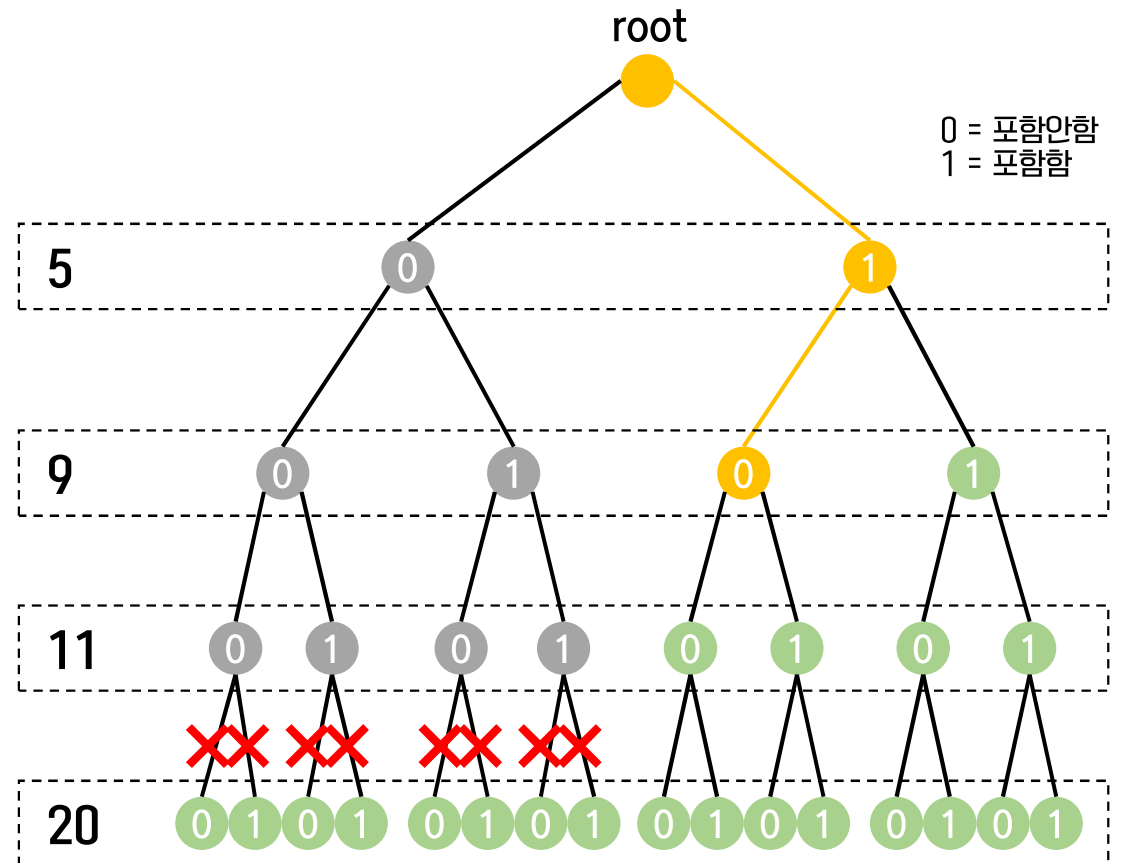
Tuple(1)



BackTracking_예시

- 지금 까지의 방식으로 계속 반복

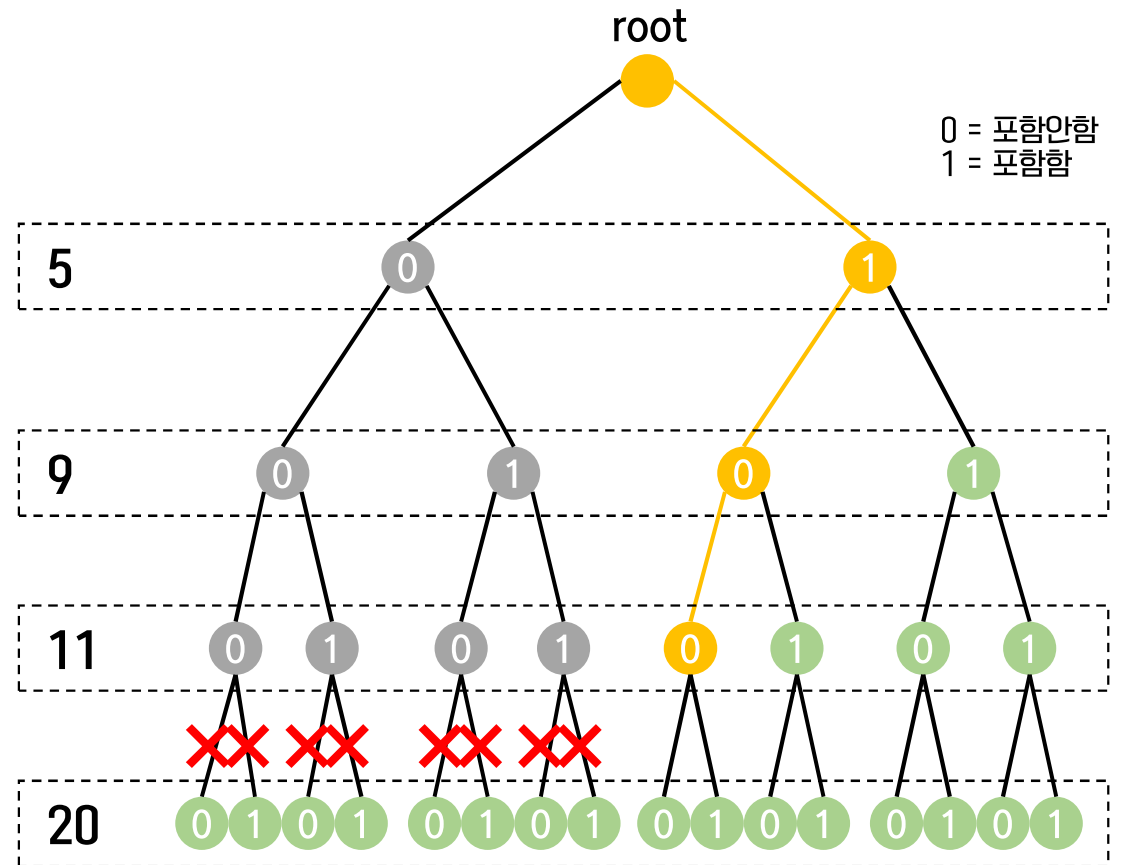
Tuple(1, 0)



BackTracking_예시

- 지금까지의 방식으로 계속 반복

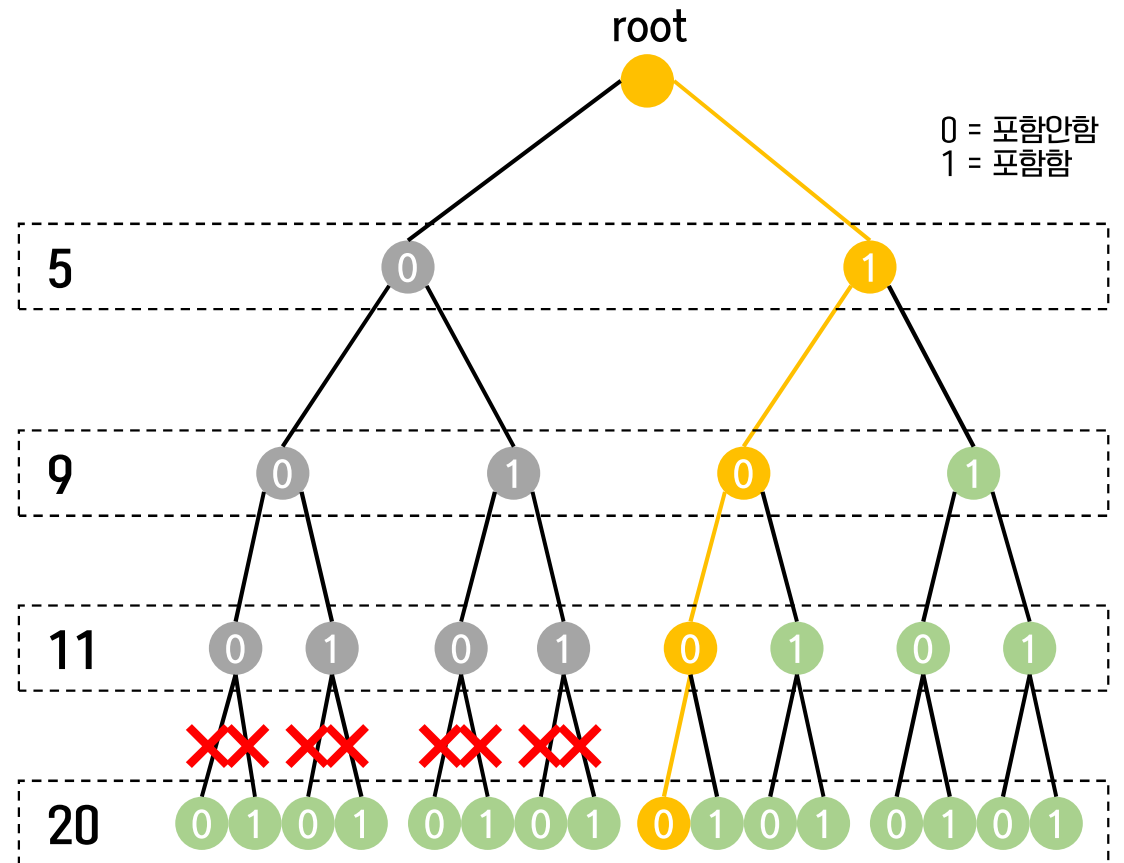
Tuple(1, 0, 0)



BackTracking_예시

- 지금까지의 방식으로 계속 반복

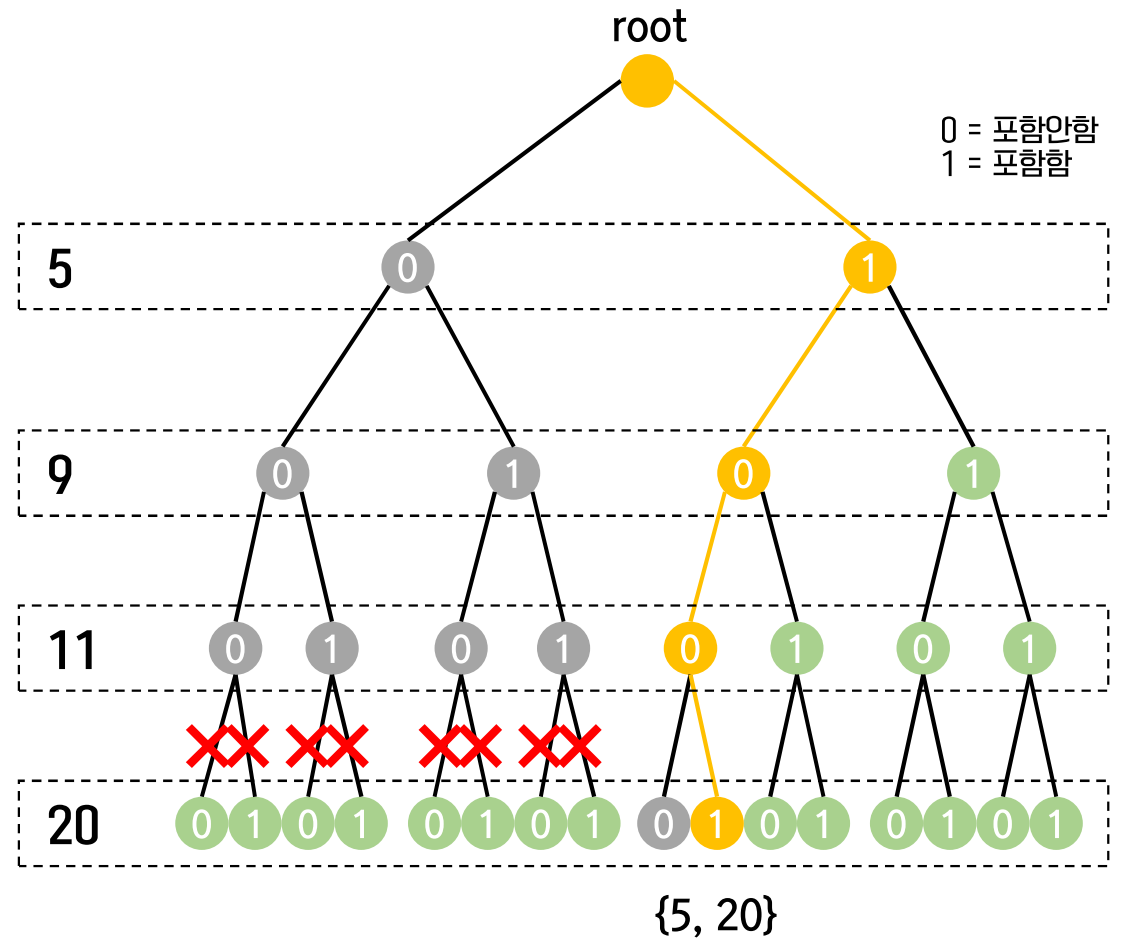
Tuple(1, 0, 0, 0)



BackTracking_여시

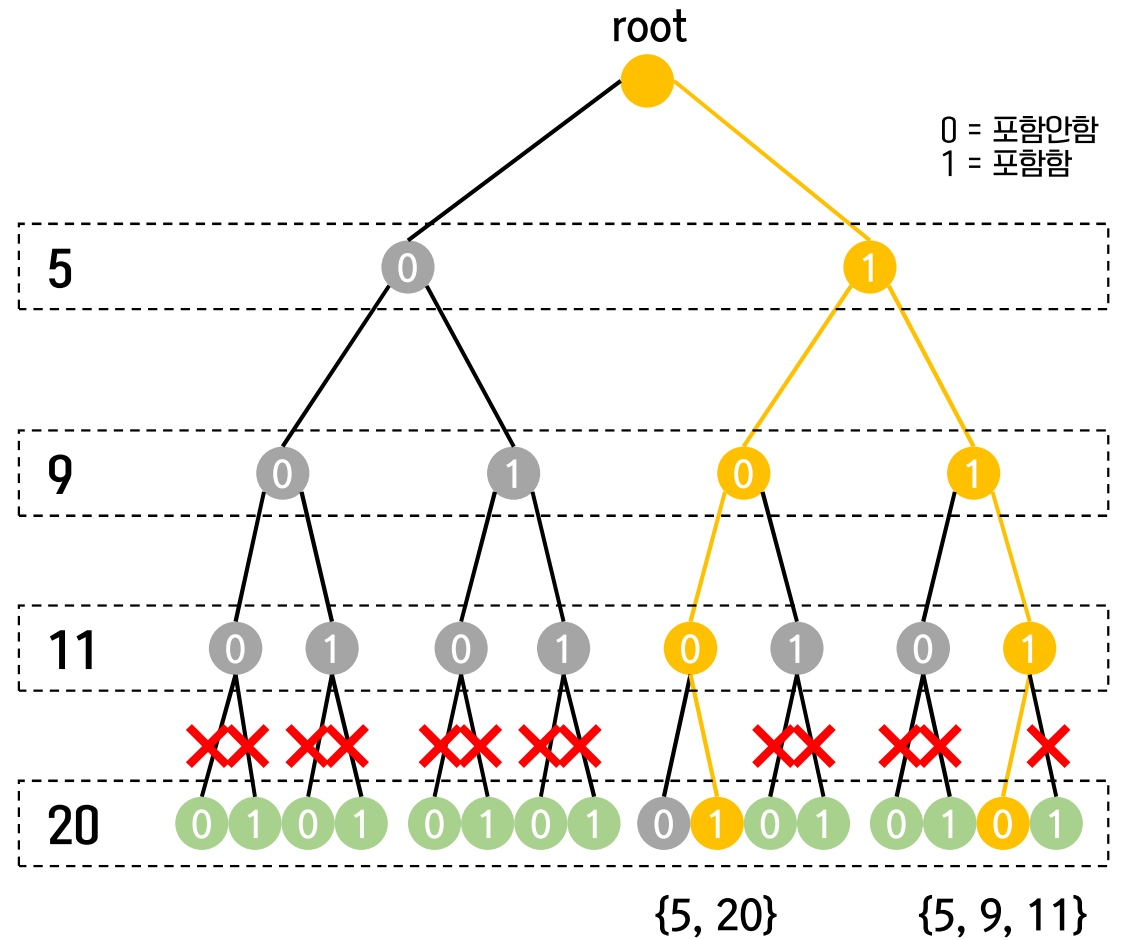
- 정답을 찾거나 모든 노드가 죽으면 탐색종료

Tuple(1, 0, 0, 1)



BackTracking_예시

- 모든 정답을 찾고 싶으면
모든 노드가 죽을 때 까지 탐색



BackTracking_구현

근데 트리의 노드와 간선을 모두 저장하면 메모리를 많이 쓰지 않을까?



우리는 해의 범위를 알고 있기 때문에 자식노드를 만들어 낼 수 있다.
간선과 노드는 저장하지 않고 순회를 하면서 만든다

BackTracking_구현

첫번째 방법 - 모든 해를 만들고 최적해인지 검사(쉬움)

```
int solution[5];  
// main 에서는 root부터 시작하므로 Backtracking(0)으로 시작  
int Backtracking(int depth) { //dfs방식으로 트리 순회, depth로 현재 깊이 확인  
    if (depth == 4) //깊이가 4일때 = 해일때  
        if (isOK()) // 최적해인지 검사  
            doSomething(); // 출력하거나 다른 결과수행  
    for (int i = 0; i <= 1; i++) { // 해가 완성x, 같은 깊이의 노드 개수만큼 트리를 만드는 구간  
        solution[depth] = i; // solution에 포함 되는지 안되는지 저장 or 다른 문제에서는 변형  
        Backtracking(depth + 1); // 다음 자식노드로 이동함  
    }  
}
```

BackTracking_구현

첫번째 방법 - 모든 해를 만들고 최적해인지 검사(쉬움)

해가 많이 존재할 때에는 어떻게?

```
int solution[5];  
if (depth == 4) // 깊이가 4일때 = 해를 찾았을 때  
    if (isOk()) // 최적해인지 검사  
        doSomething(); // 출력하거나 다른 결과수행  
for (int i = 0; i <= 1; i++) { // 해가 완성x, 같은 깊이의 노드 개수만큼  
    solution[depth] = i; // solution에 포함 되는지 안되는지 저장 or 다른 문제에서는 변형  
    Backtracking(depth + 1); // 다음 자식노드로 이동함  
}
```

BackTracking_구현

두번째 방법 - 현재 노드의 **자식노드들이 최적해**일 수 있나 판단(가지치기)

```
int solution[5];
// main에서는 root부터 시작하므로 Backtracking(0)으로 시작
int Backtracking(int depth) { //dfs방식으로 트리 순회, depth로 현재 깊이 확인
    if (depth == 4) //깊이가 4일때 = 해일때
        if (isOK()) // 최적해인지 검사
            doSomething(); // 출력하거나 다른 결과수행
    for (int i = 0; i <= 1; i++) { // 해가 완성x, 같은 깊이의 노드 개수만큼 트리를 만드는 구간
        solution[depth] = i; // solution에 포함 되는지 안되는지 저장 or 다른 문제에서는 변형
        // 지금까지의 합이 25이상이거나 지금부터 앞으로의 합이 25보다 작으면 넘어가지 말고 continue; 사용
        Backtracking(depth + 1); // 다음 자식노드로 이동함
    }
}
```

BackTracking_구현

이렇듯 모든 해를 찾지 말고 **가지치기**를 하면서 탐색
하는 것이 BackTracking의 핵심입니다!

연습

1182번 - 부분수열의 합
1759번 - 암호 만들기(LIS)
2661번 - 좋은 수열
1987 - 알파벳

할일 많음 + 움직이기 싫음 = 내일 해도 되겠지?



할일 많음+움직이기 싫음=내일해도 되겠지?

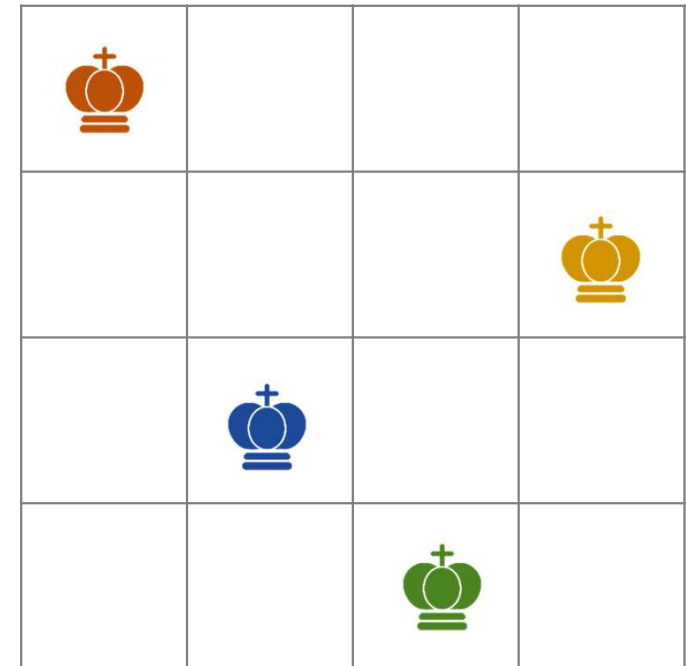


N-Queen_(9663번)

백트래킹의 교과서 문제








N-Queen

- 1. 같은 **행**에 존재하면 안 된다.
- 1. 같은 **열**에 존재하면 안 된다.
- 1. 같은 **대각선**에 존재하면 안 된다.



N-Queen

- 같은 행에 여왕이 2명 이상 올 수 없다?
→ x, y 축으로 나누어 생각해보자





1	2	3	4
			
			
			
			

N-Queen

- 확인은 어떻게 할까요?





→ 상하좌우는 x, y 좌표가 같은 것이 있는지 확인

→ 대각선은 x, y 좌표의 합과 차가 같은지 확인

1	2	3	4
			
			
			
			

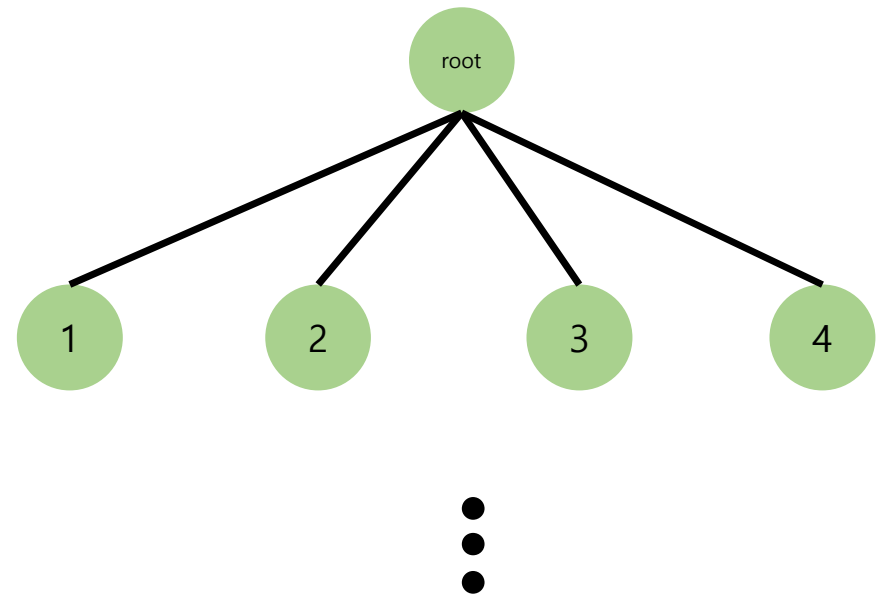
N-Queen

- 행 하나당 4개의 x좌표를 트리로 만들기

	1	2	3	4
Level 1 - - -		- - - - -	- - - - -	- - - - -
Level 2 - - -	- - - - -	- - - - -	- - - - -	
Level 3 - - -	- - - - -		- - - - -	- - - - -
Level 4 - - -	- - - - -	- - - - -		- - - - -

N-Queen

- 행 하나당 4개의 위치를 트리로 만들기



연습

9663 - N-Queen
1799번 - 비숍
2580번 - 스도쿠



다 보느라 수고하셨습니다 감사합니다

역시 나는 총명하구게
획적이구
최고라니깐!

