ALOHA
#3주차

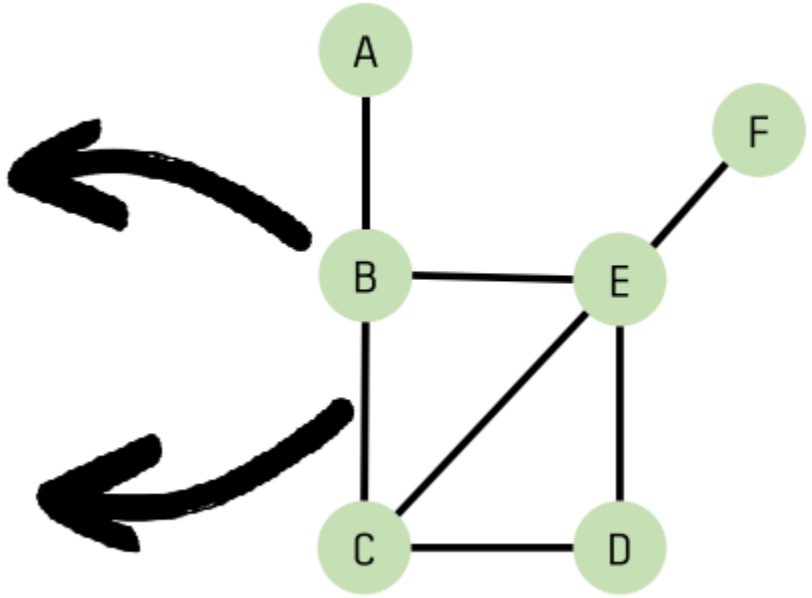
DFS, BFS, Backtracking

#CH.1

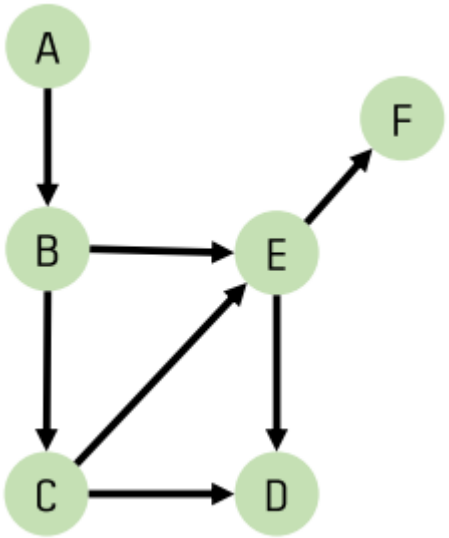
Graph

노드 (Node)
= 위치
알고리즘에서는 정점(vertex)

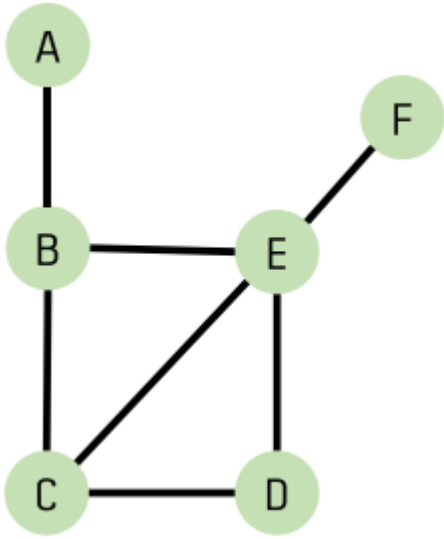
간선 (edge)
= 길
노드를 연결하는 선



Graph



Directed Graph
방향성 O



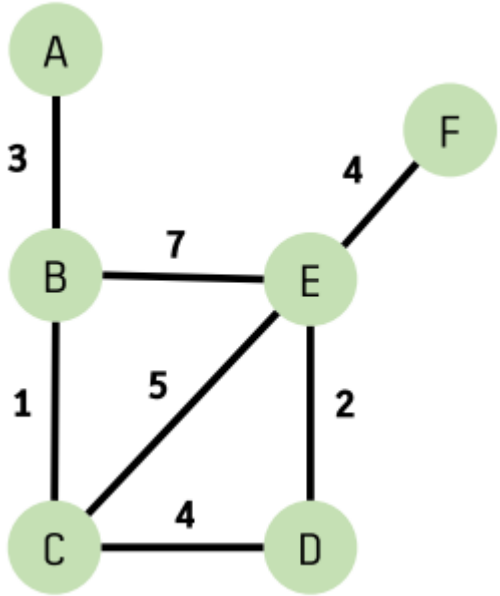
Undirected Graph
방향성 X

Graph

가중치 그래프

그래프의 간선에 비용이나 가중치가 할당된 그래프

ex) 도로를 지나가는데 드는 비용, 도로의 길이 등



Graph

그래프를 표현할 수 있는 2가지 방법

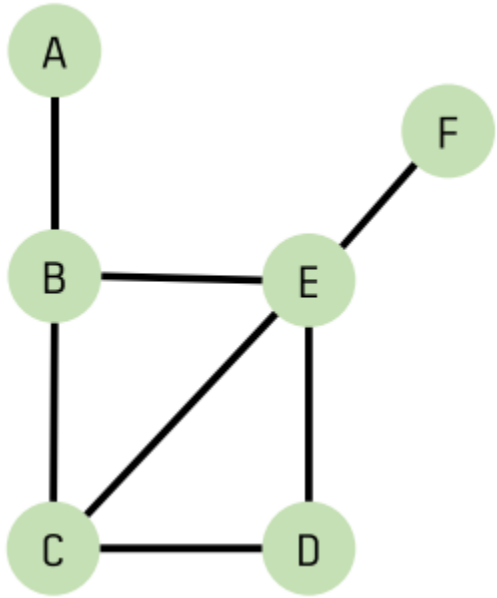
인접 리스트(연결리스트, 벡터)

인접 행렬 (배열)

Graph - 인접행렬

간선이 존재하면 “1”, 그렇지 않으면 “0”

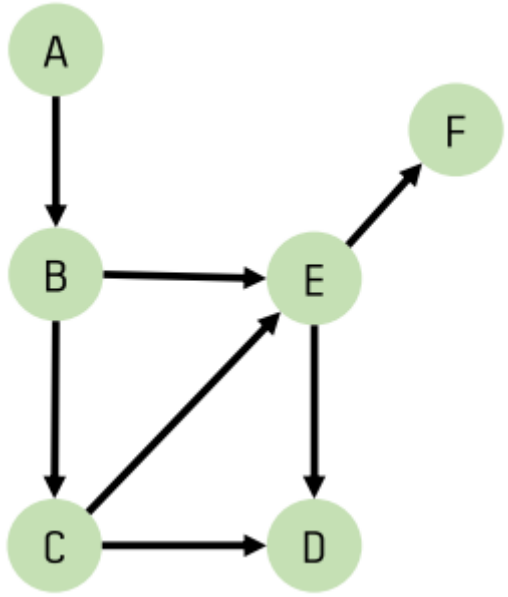
	A	B	C	D	E	F
A	0	1	0	0	0	0
B	1	0	1	0	1	0
C	0	1	0	1	1	0
D	0	0	1	0	1	0
E	0	1	1	1	0	1
F	0	0	0	0	1	0



Graph - 인접행렬

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

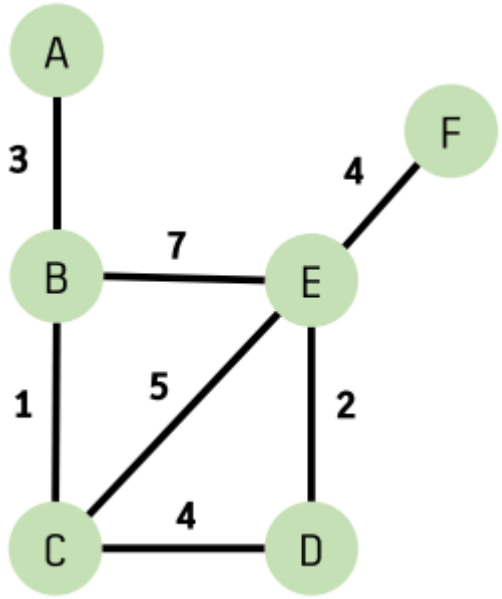


Graph - 인접행렬

(가중치가 존재하면)

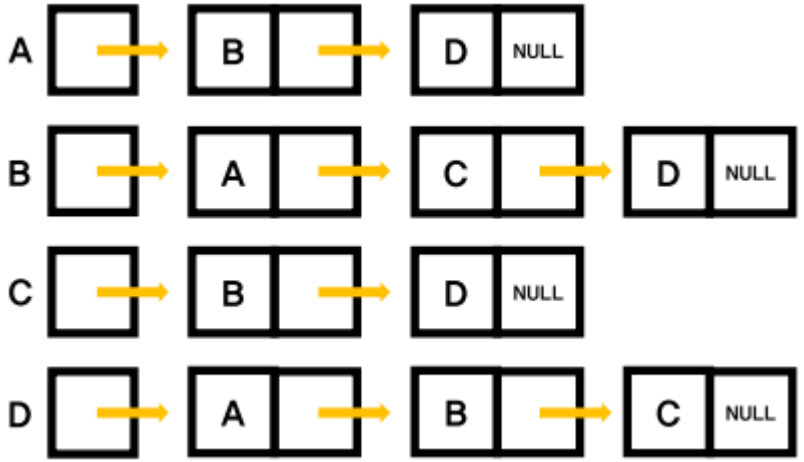
간선이 존재하면 "가중치", 그렇지 않으면 "0"

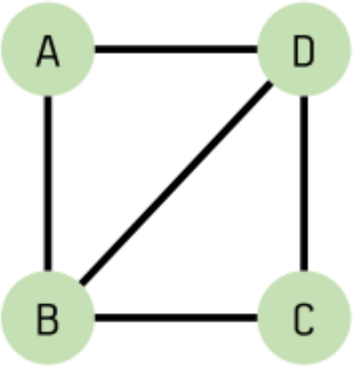
	A	B	C	D	E	F
A	0	3	0	0	0	0
B	3	0	1	0	7	0
C	0	1	0	4	5	0
D	0	0	4	0	2	0
E	0	7	5	2	0	4
F	0	0	0	0	4	0



Graph - 인접리스트(연결리스트)

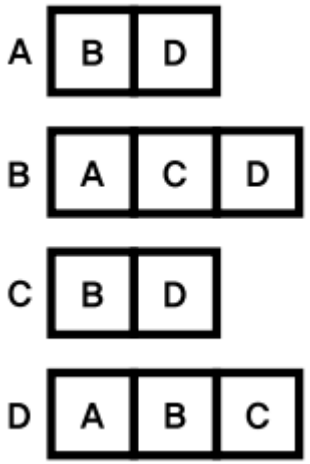
연결리스트를 사용할 수도 있지만,
알고리즘에서는 잘 사용하지 않는다. PASS!!

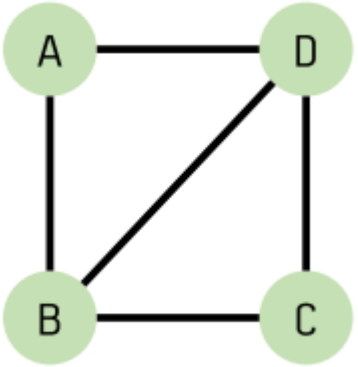




Graph - 인접리스트(벡터)

각각의 정점 벡터에 연결된 정점을 push_back

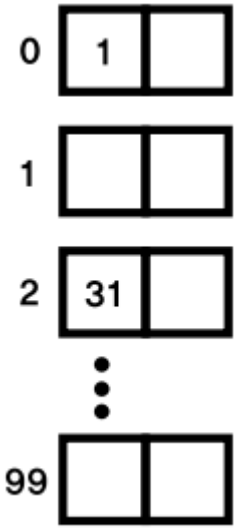




Graph - 인접리스트(벡터)

벡터 이용법?

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



Graph - 인접리스트(벡터)

그래프의 탐색

그래프의 모든 정점을 방문하는 것으로
DFS, BFS 등이 존재한다.

#CH.2

Depth First Search (깊이 우선 탐색)

DFS

개념

현재 정점에서 갈 수 있는 정점들까지 들어가며 탐색

한 방향으로 계속 갈 수 있을 때까지 탐색하다 막히면(더 이상 연결된 노드가 없다면)
가장 가까운 갈림길로 돌아와서 다른 방향으로 탐색을 진행

모든 정점을 방문하고 싶을 때 사용

BFS와 비교하면 코드는 **간단**하다.

DFS

특징

한 번 방문한 정점은 다시 방문하지 않는다

정점에 방문했는지 확인하는 배열을 만들어 구현하기도 함

재귀함수 혹은 스택을 이용하여 구현

표현 방식에 따른 시간복잡도의 차이

인접 행렬 : O(V^2)
인접 리스트 : O(V+E)
(V는 노드의 수 / E는 간선의 수)

DFS

간선(E)의 수가 적은 경우 인접 리스트를 사용하는 것이 유리하다!!

재귀함수 혹은 스택을 이용하여 구현

표현 방식에 따른 시간복잡도의 차이

인접 행렬 : O(V^2)
인접 리스트 : O(V+E)
(V는 노드의 수 / E는 간선의 수)

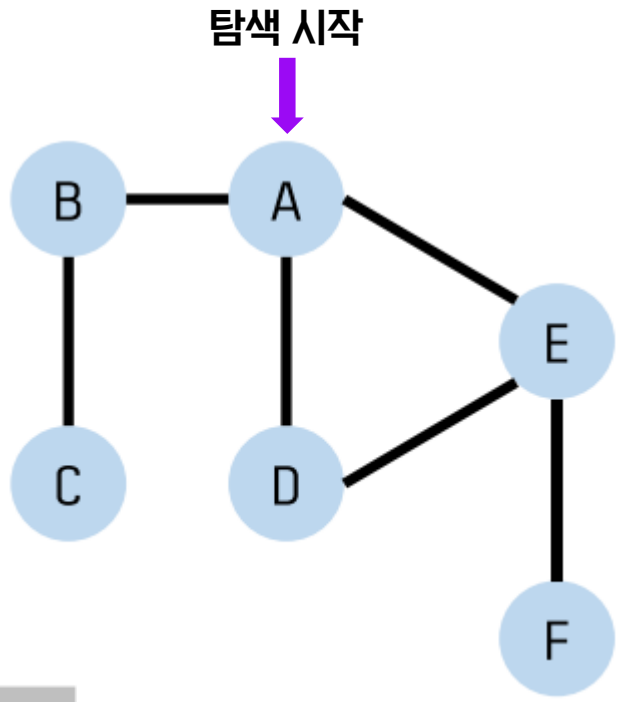
V^2 > V+E

DFS

원리

A ~ F 노드의 방문을 확인하는 visit 배열을 생성
정점의 개수 ≤ 배열 크기

그래프를 입력 받고, 탐색 시작할 정점 결정



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

now	

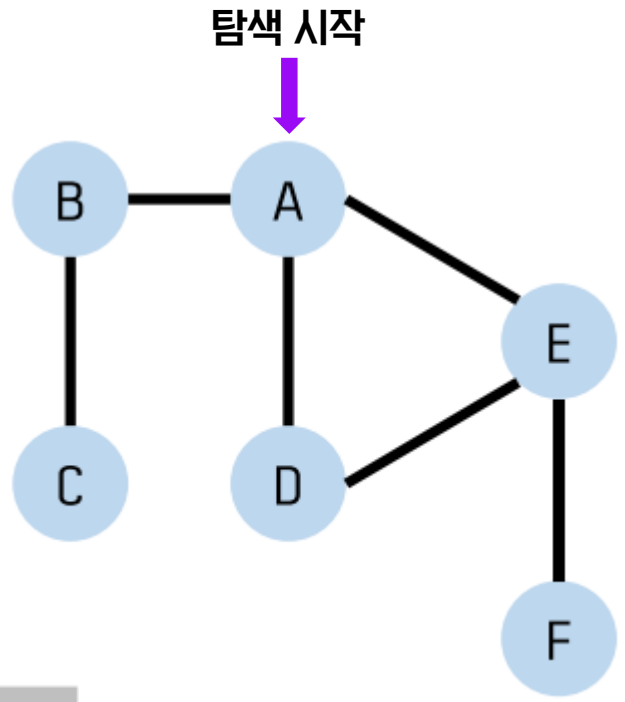
DFS

원리

A부터 탐색 시작

다른 정점부터 시작해도 상관 없다

그 이후 번호가 작은 정점부터 탐색



방문 순서

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

now	

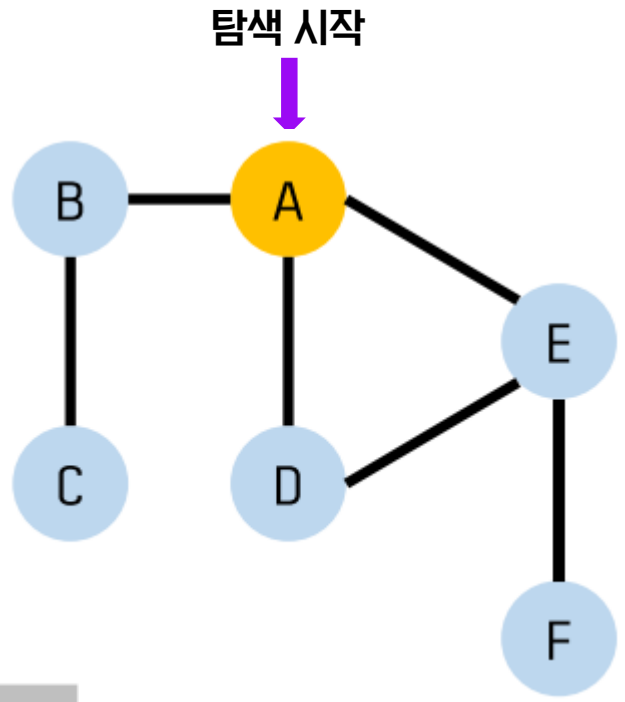
DFS

원리

A부터 탐색 시작

다른 정점부터 시작해도 상관 없다

그 이후 번호가 작은 정점부터 탐색



방문 순서

A

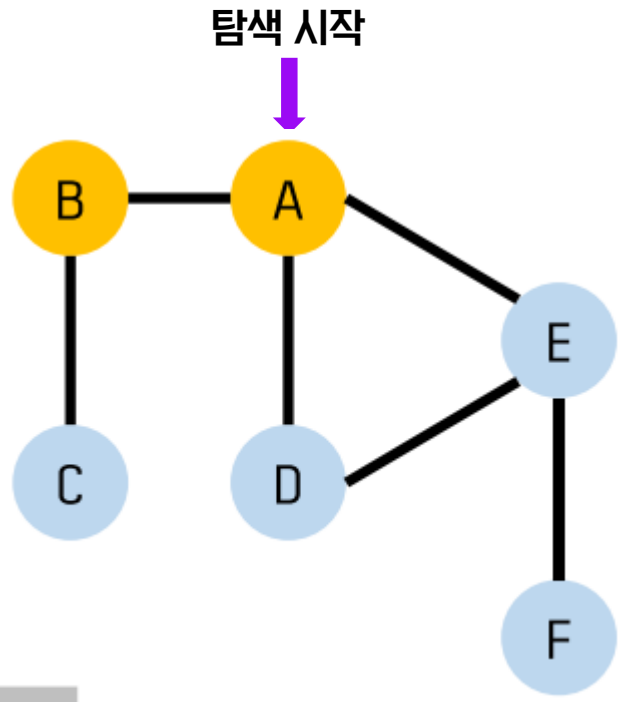
	A	B	C	D	E	F
visit	1	0	0	0	0	0

now	A

DFS

원리

그 이후 번호가 작은 정점부터 탐색



방문 순서

A → B

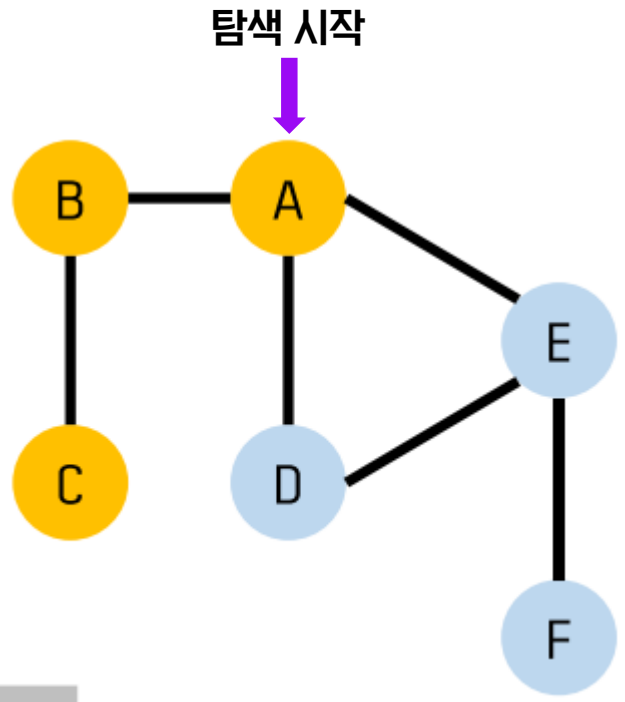
	A	B	C	D	E	F
visit	1	1	0	0	0	0

now	B

DFS

원리

그 이후 번호가 작은 정점부터 탐색



방문 순서

A → B → C

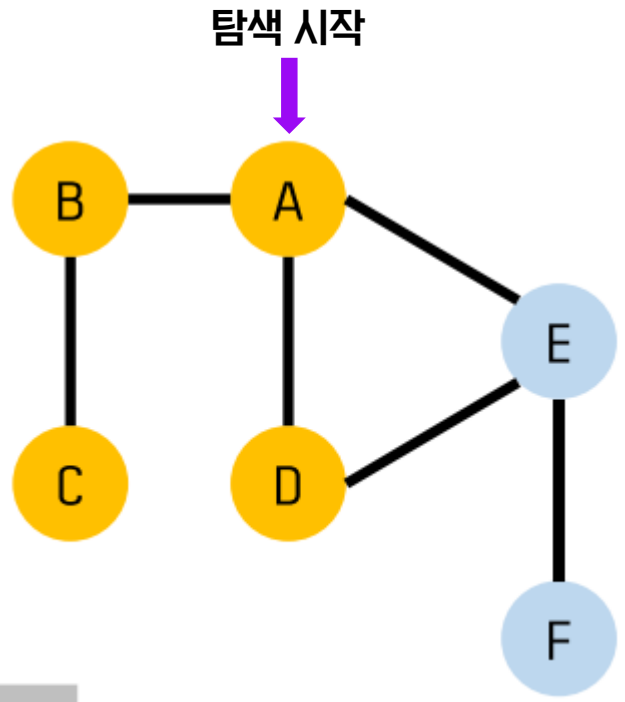
	A	B	C	D	E	F
visit	1	1	1	0	0	0

now	C

DFS

원리

C에서 더 갈 수 있는



방문 순서

A → B → C → D

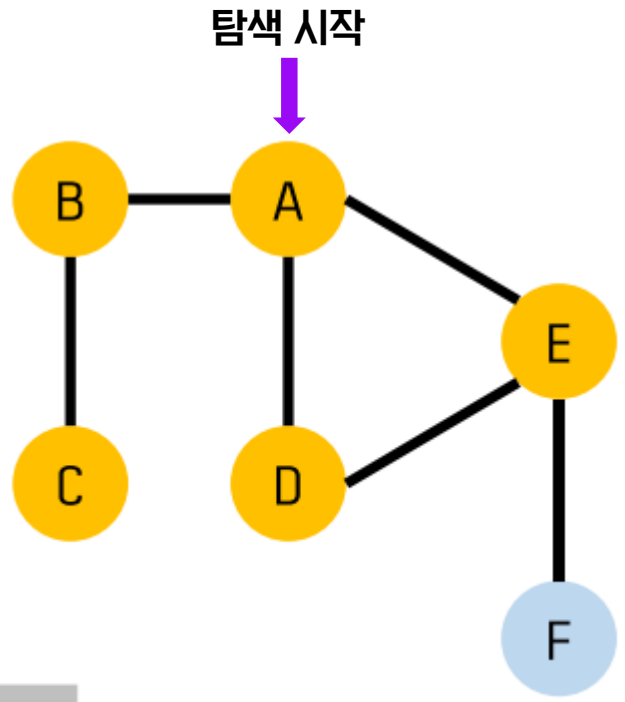
	A	B	C	D	E	F
visit	1	1	1	1	0	0

now	D

DFS

원리

그 이후 번호가 작은 정점부터 탐색



방문 순서

A → B → C → D → E

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

now	E

DFS

원리

그 이후 번호가 작은 정점부터 탐색

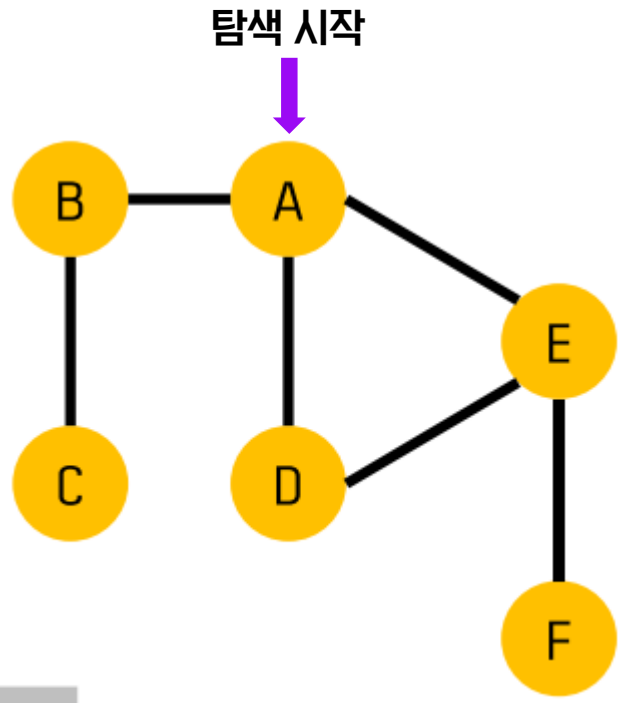
방문하지 않은 정점이 없으므로 탐색 종료

방문 순서

A → B → C → D → E → F

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

now	F



#CH.3

Breadth First Search (너비 우선 탐색)

BFS

개념

현재 정점에서 인접한 연결된 정점부터 탐색

시작 정점에 가까운 정점부터 넓게 탐색하여 먼 정점은 가장 나중에 방문

정점 간의 최단 경로 또는 임의의 경로를 찾을 때 사용

DFS와 비교하면 코드는 복잡하다.

BFS

특징

한 번 방문한 정점은 다시 방문하지 않는다

정점에 방문하였는지 확인하는 visit 배열을 만들어 구현

큐를 이용하여 구현

표현 방식에 따른 시간복잡도의 차이

인접 행렬 : O(V^2)
인접 리스트 : O(V+E)
(V는 노드의 수 / E는 간선의 수)

+ Dijkstra 알고리즘과 유사

BFS

간선(E)의 수가 적은 경우 인접 리스트를 사용하는 것이 유리하다!!

큐를 이용하여 구현

표현 방식에 따른 시간복잡도의 차이

인접 행렬 : O(V^2)
인접 리스트 : O(V+E)
(V는 노드의 수 / E는 간선의 수)

V^2 > V+E

+ Dijkstra 알고리즘과 유사

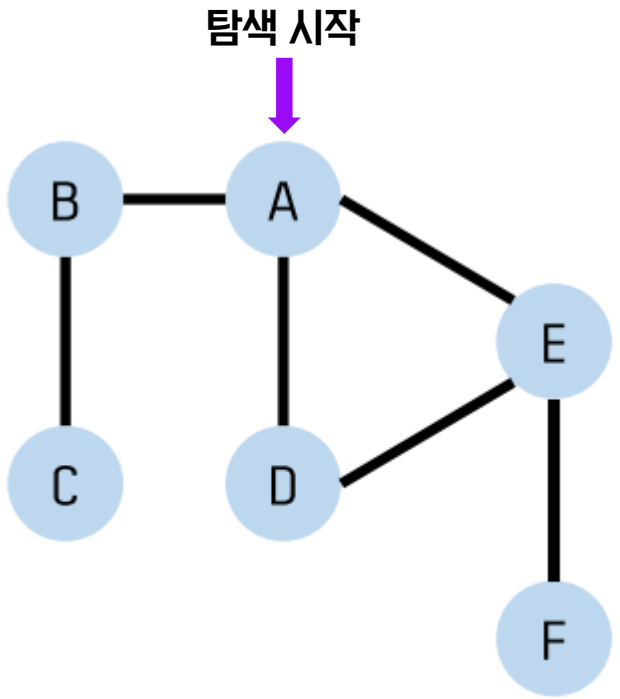
BFS

원리

A ~ F 노드의 방문을 확인하는 visit 배열을 생성

정점의 개수 ≤ 배열 크기

그래프를 입력 받고, 탐색 시작할 정점 결정

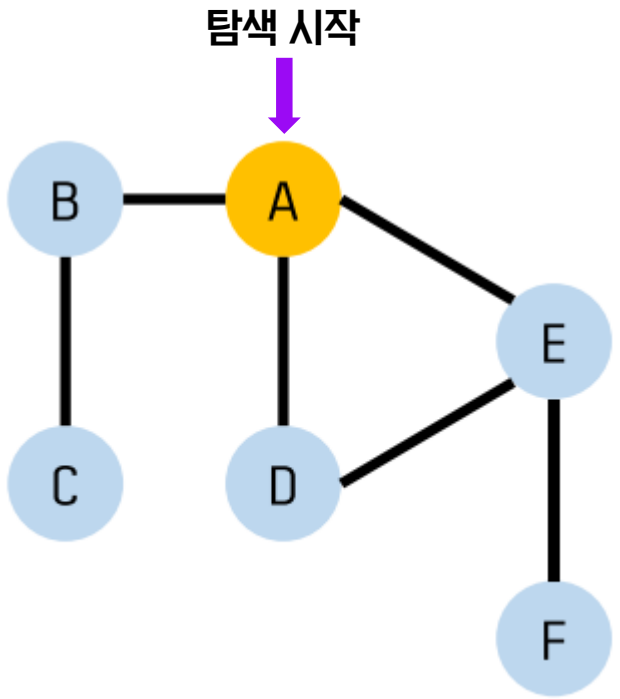


	A	B	C	D	E	F
visit	0	0	0	0	0	0
Queue						

BFS

원리

큐에 처음 탐색 시작하는 정점 입력



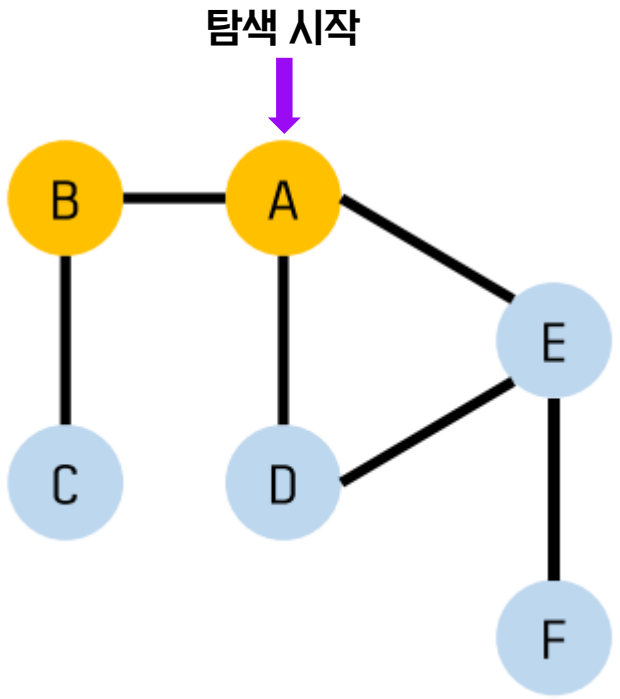
	A	B	C	D	E	F
visit	1	0	0	0	0	0
Queue	A					

BFS

원리

A와 인접한 방문하지 않은 정점을 큐에 삽입
큐의 순서대로 탐색한다

방문한 정점은 큐에서 POP



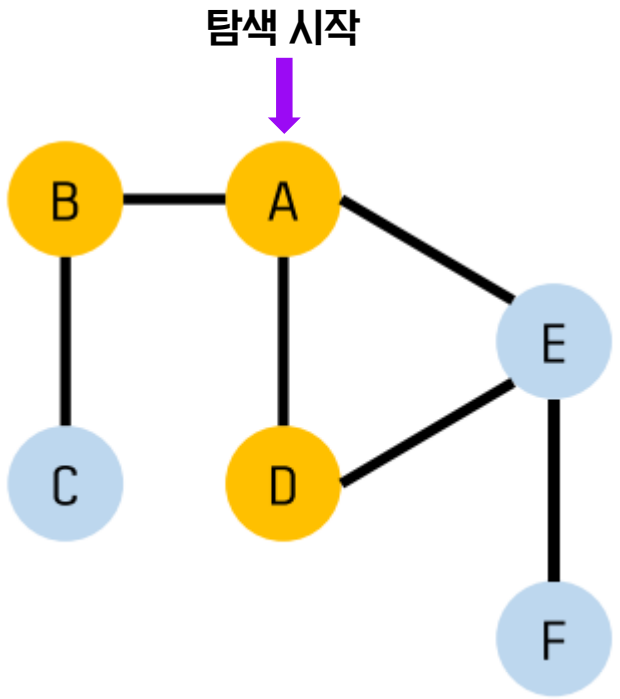
	A	B	C	D	E	F
visit	1	0	0	0	0	0
Queue	B	D	E			

BFS

원리

B와 인접한 방문하지 않은 정점을 큐에 삽입
큐의 순서대로 탐색한다

방문한 정점은 큐에서 POP



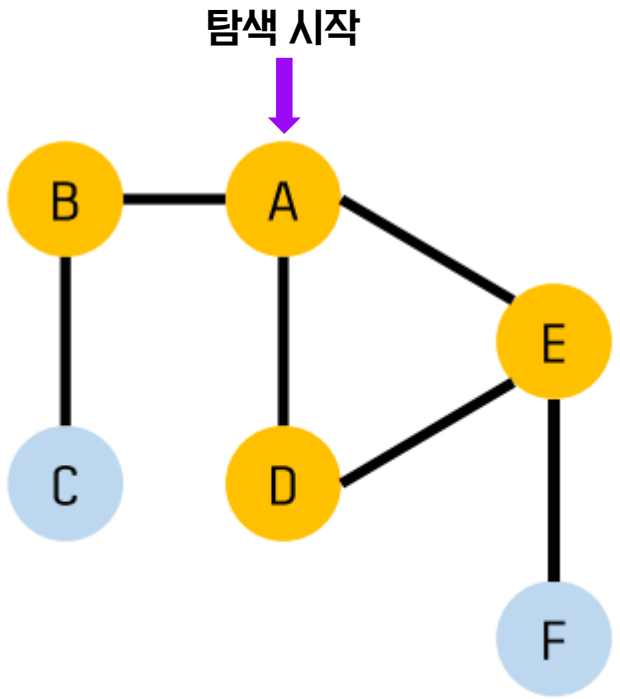
	A	B	C	D	E	F
visit	1	1	0	0	0	0
Queue	D	E	C			

BFS

원리

D와 인접한 방문하지 않은 정점을 큐에 삽입
큐의 순서대로 탐색한다

방문한 정점은 큐에서 POP



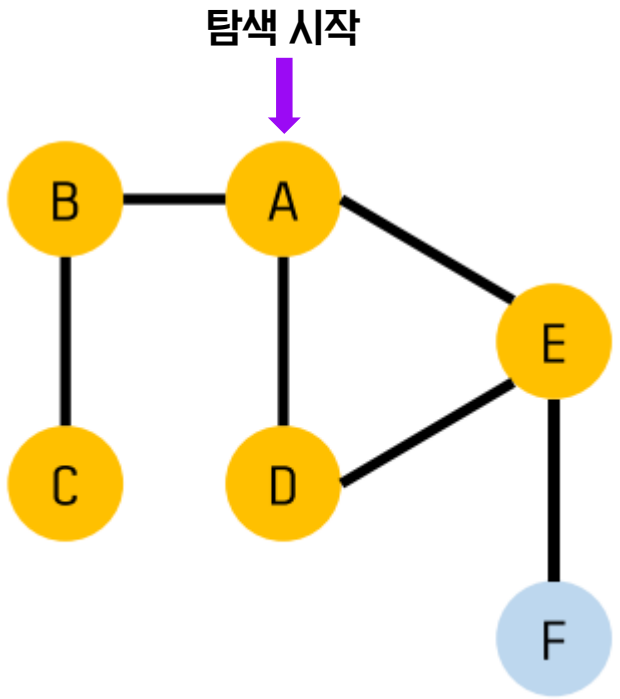
	A	B	C	D	E	F
visit	1	1	0	1	0	0
Queue	E	C	F			

BFS

원리

E와 인접한 방문하지 않은 정점을 큐에 삽입
큐의 순서대로 탐색한다

방문한 정점은 큐에서 POP



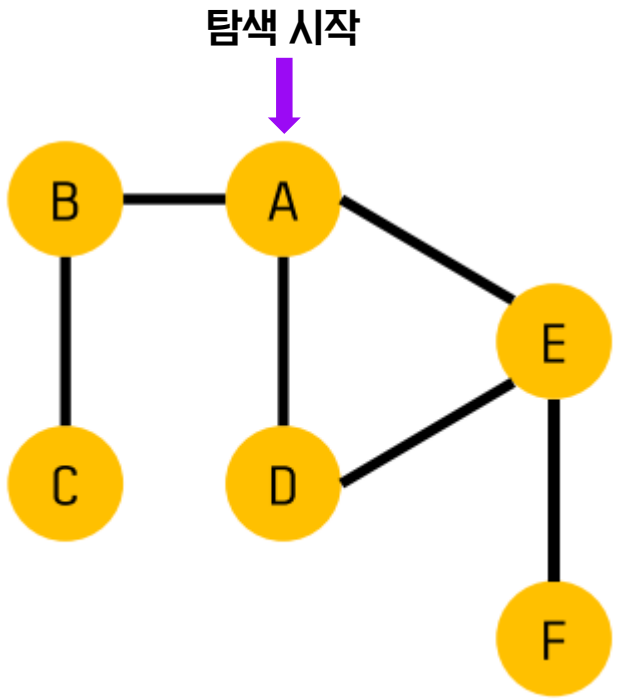
	A	B	C	D	E	F
visit	1	1	0	1	1	0
Queue	C	F				

BFS

원리

F와 인접한 방문하지 않은 정점을 큐에 삽입
큐의 순서대로 탐색한다

방문한 정점은 큐에서 POP

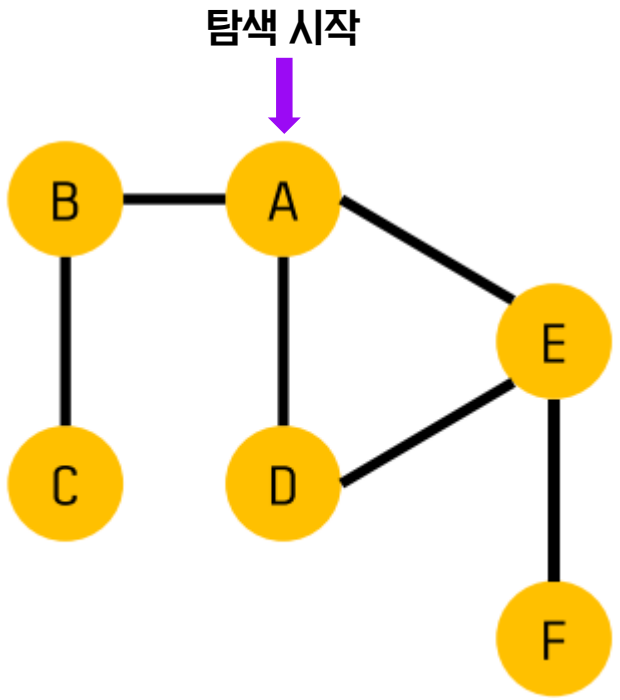


	A	B	C	D	E	F
visit	1	1	1	1	1	0
Queue	F					

BFS

원리

모든 정점을 방문했으면 탐색 종료



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

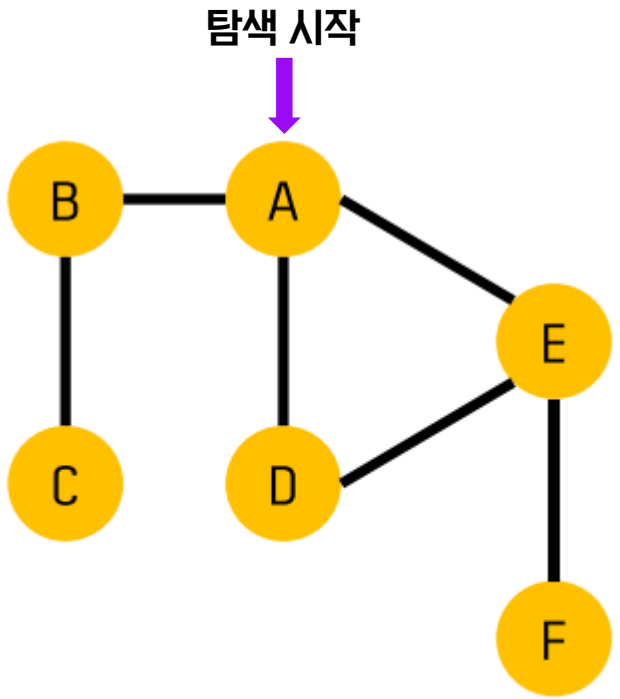
BFS

원리

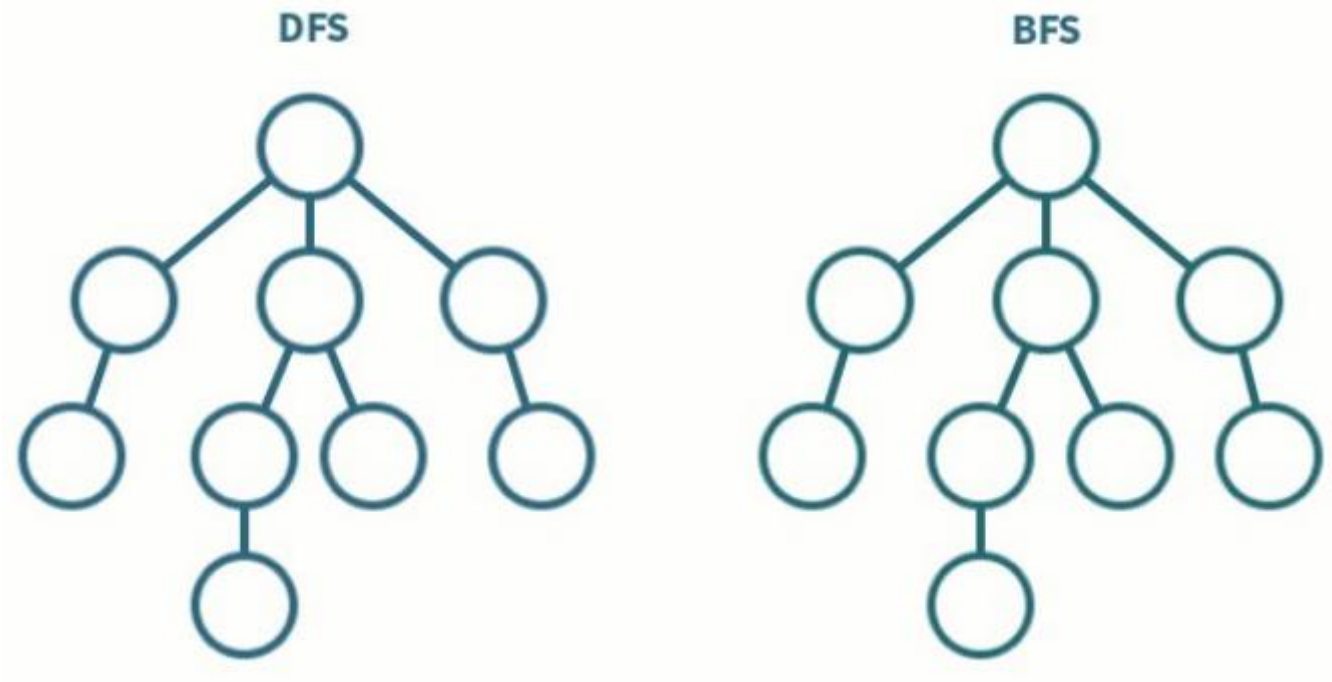
탐색 순서

A → B → D → E → C → F

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



한눈에 보는 DFS와 BFS의 탐색 차이



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

#CH.4

DFS, BFS의 구현

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
            }
        }

    }
}
```

BFS 구현 (벡터)

만약 visit_bfs를 정점을 pop할 때 바꾼다면?

- Queue에 여러 번 입력되어 중복 방문이 될 수 있다!!

풀어볼까유

#1260 DFS와 BFS

조규현

풀어볼까유

#11724

연결 요소의 개수

BFS, DFS 기초 문제

풀어볼까유

#2178 미로 찾기

뒤에 힌트!

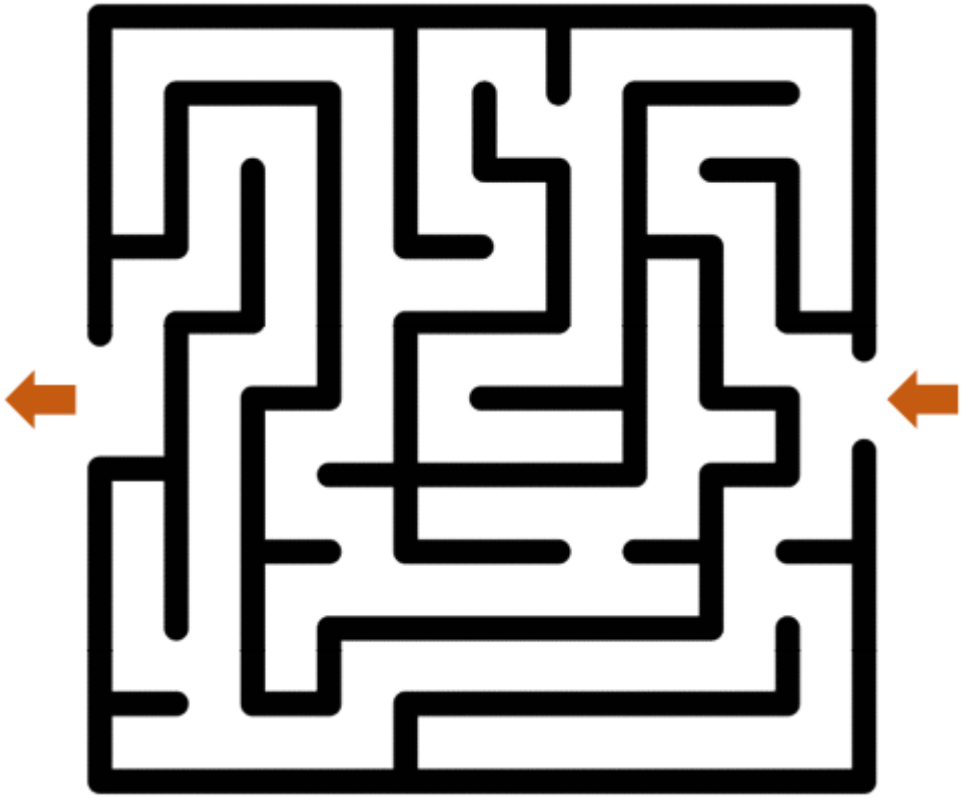
boj.kr/2178

정점 대신 (x, y) 좌표

BFS는 pair<int, int>를 사용하여 Queue 이용

지도이므로 배열로 받아서 탐색

visit은 미로의 모든 좌표 (2차원)



boj.kr/2178

한자리 수로 입력하기

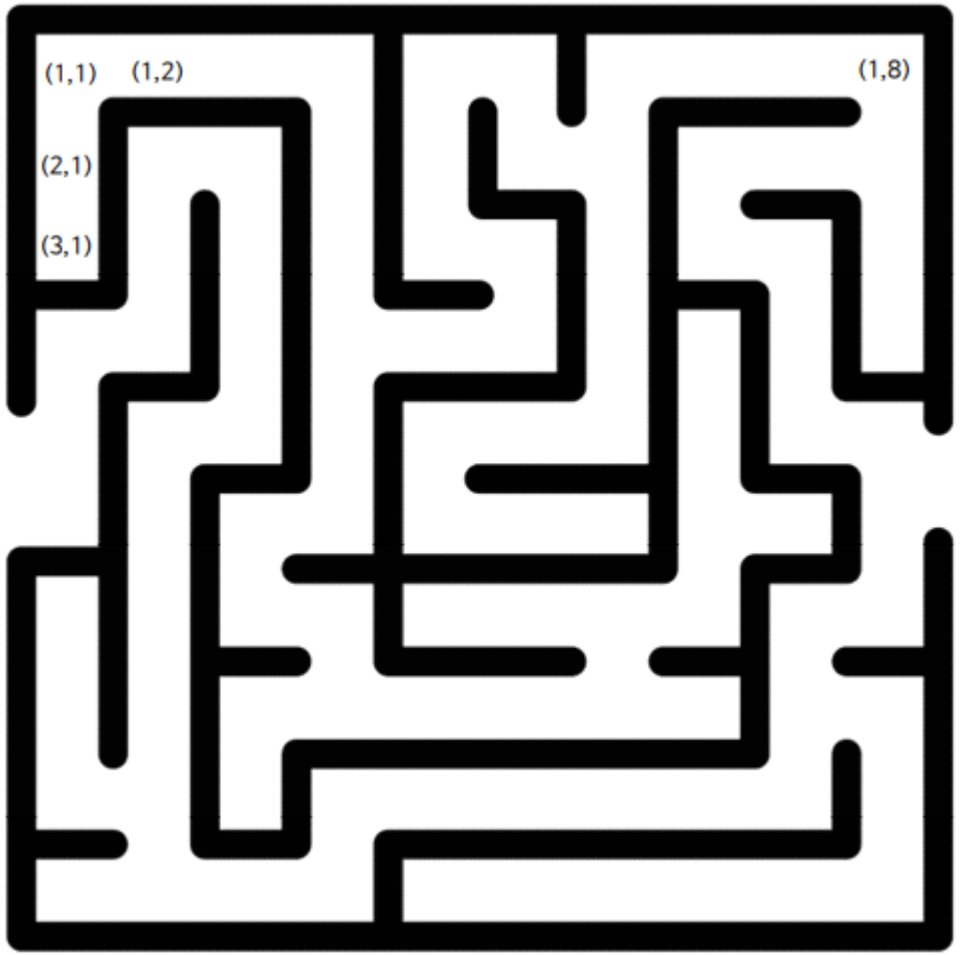
```
scanf("%1d", &MAP[i][j]);
```

최단거리를 구하기 위한 식

dist[x][y] 배열로 각 좌표까지
최단거리 나타내기

상, 하, 좌, 우 표현하기

```
int dx[4] = {-1,1,0,0};
int dy[4] = {0,0,1,-1};
```



boj.kr/2178

```
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?

boj.kr/2178

```
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?

boj.kr/2178

memset을 사용하여 배열을 -1로 초기화

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

-1	-1	-1	-1	-1	-1	-1	-1
-1	-1	-1	-1	-1	-1	-1	-1
-1	-1	-1	-1	-1	-1	-1	-1
-1	-1	-1	-1	-1	-1	-1	-1
-1	-1	-1	-1	-1	-1	-1	-1
-1	-1	-1	-1	-1	-1	-1	-1
-1	-1	-1	-1	-1	-1	-1	-1
-1	-1	-1	-1	-1	-1	-1	-1

boj.kr/2178

지도를 받고 BFS 탐색 실행

if문에 -1, 0이 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

boj.kr/2178

지도의 크기가 나와 있다면

if문에 지도의 크기보다 x, y값이 작다는 조건 추가

풀어볼까유

#2178 미로 찾기

이제는 할 수 있겠죠?

풀어볼까유

#1012 유기농 배추

비슷한 문제

풀어볼까유

#7576 토마토

DFS BFS 정석 문제

풀어볼까유

#2644 촌수계산

비슷한 문제

풀어볼까유

#2583 영역 구하기

비슷한 문제

풀어볼까유

#7562
나이트의 이동

dx, dy를 잘 정해보자!

#CH.5

BackTracking (feat.DFS)

BackTracking

개념

모든 경우의 수를 전부 고려하는 알고리즘
= ~~Brute-Force?~~

BUT

(조건을 만족하는) 모든 경우의 수를 전부
고려하는 트리 탐색 알고리즘

BackTracking

개념

(조건을 만족하는) 모든
경우의 수를 전부 고려하는
트리 탐색 알고리즘

최적해 (정답)

BackTracking

트리

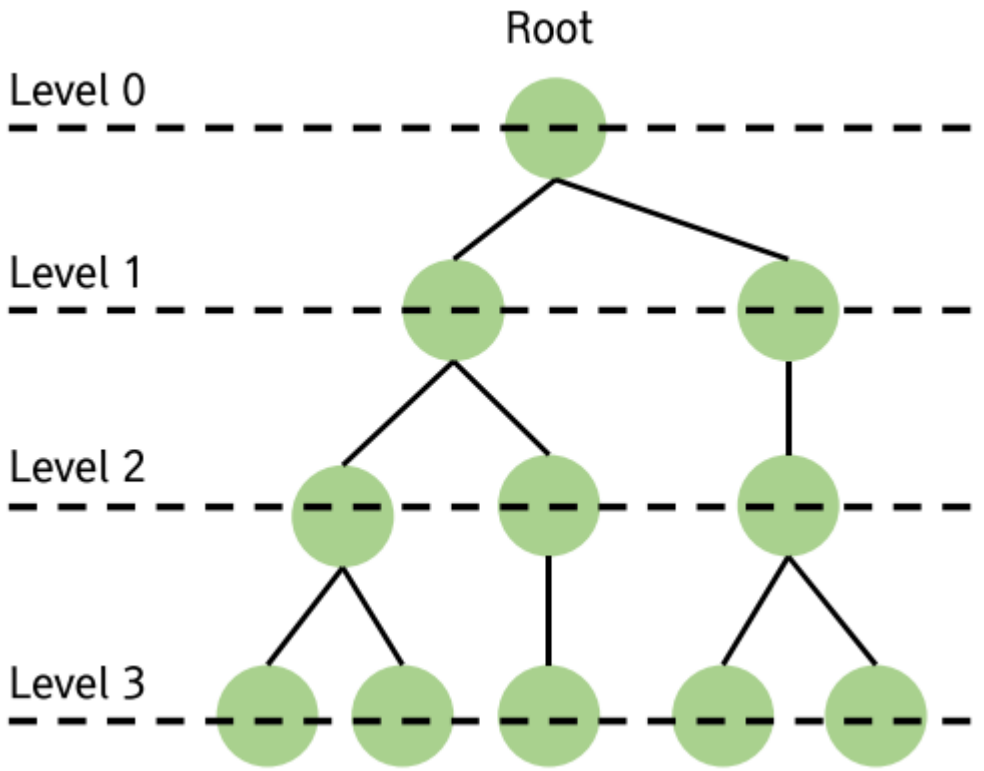
노드로 이루어진 자료구조

그래프의 일종

사이클이 없는 하나의 방향성이 있는 그래프

각 노드는 깊이를 가지고, Root 노드는 level 0

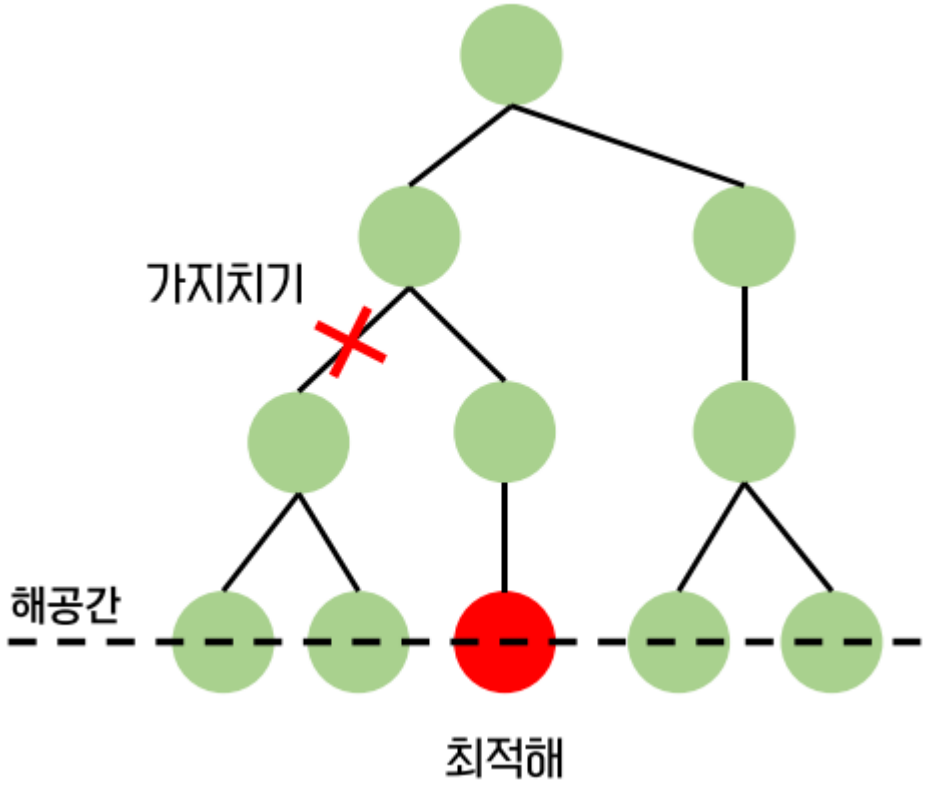
현재 노드와 연결된 레벨이 낮은 노드는
부모 노드라고 한다.



BackTracking

개념

최적해(=답)를 향해 가지를 뻗어나가며,
가능성이 없는 해(가지)는 제거하며
해 중에 최적 해를 탐색해나간다



BackTracking

질문

1. 해는 어떻게 정점으로 만드나요?
2. 트리는 어떻게 구성하나요?
3. 트리의 탐색은 어떻게 하나요?

BackTracking

1. 해는 어떻게 정점으로 만드나요?

해는 주로 Tuple을 사용해 만들지만, 다른 방법도 가능합니다

BackTracking

2. 트리는 어떻게 구성하나요?

Tuple의 개수만큼 트리의 깊이를 정합니다.

BackTracking

3. 트리의 탐색은 어떻게 하나요?

트리를 탐색하는 방법인 DFS를 사용합니다.

※ BFS를 사용하지 않는 이유
BFS: 같은 Level을 가지는 노드의 개수만큼 메모리 필요
DFS: 트리의 Level만큼 메모리가 필요

BFS의 공간복잡도가 DFS의 공간복잡도보다 매우 크므로 DFS를 사용

BackTracking

문제를 통해 원리를 공부해 봅시다!

오름차순으로 주어지는 이 집합의 부분집합의 원소들의 합이 sum이 되도록 만들어보세요

{5, 9, 11, 20}　　sum = 25

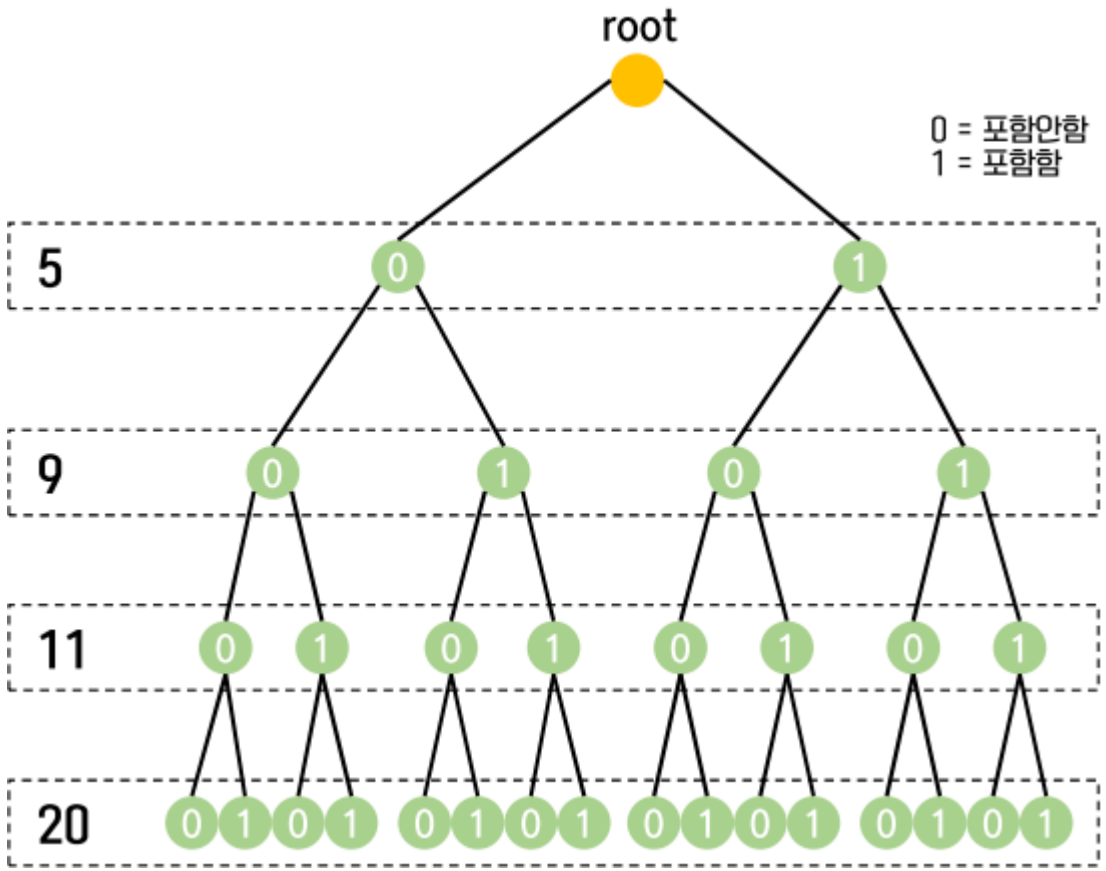
답 = {5, 9, 11}, {5, 20}

BackTracking

각 노드는 그 숫자의 유무

숫자의 개수가 4개이므로 깊이 = 4

Root부터 DFS 시작

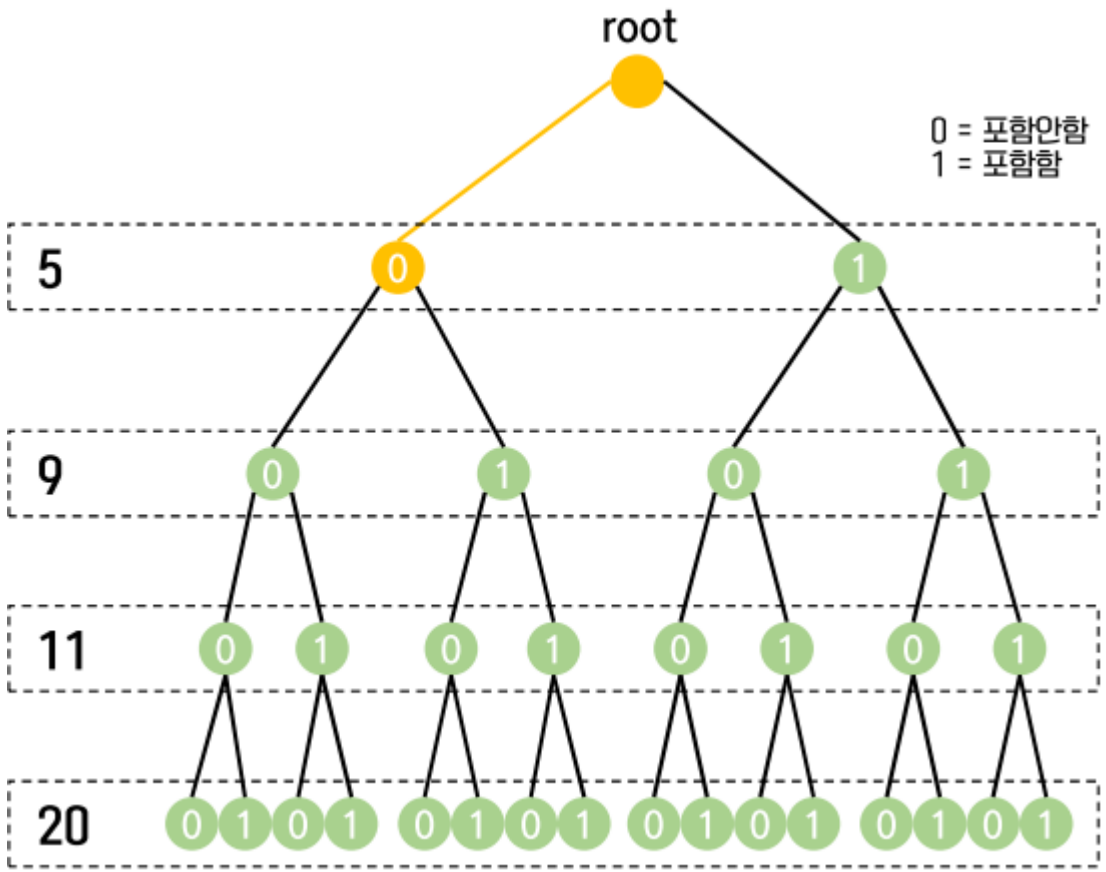


BackTracking

5를 포함하지 않으면 25보다 작나?
조건에 성립하므로 계속 탐색

순회할 때마다 조건과 비교

Tuple(0)

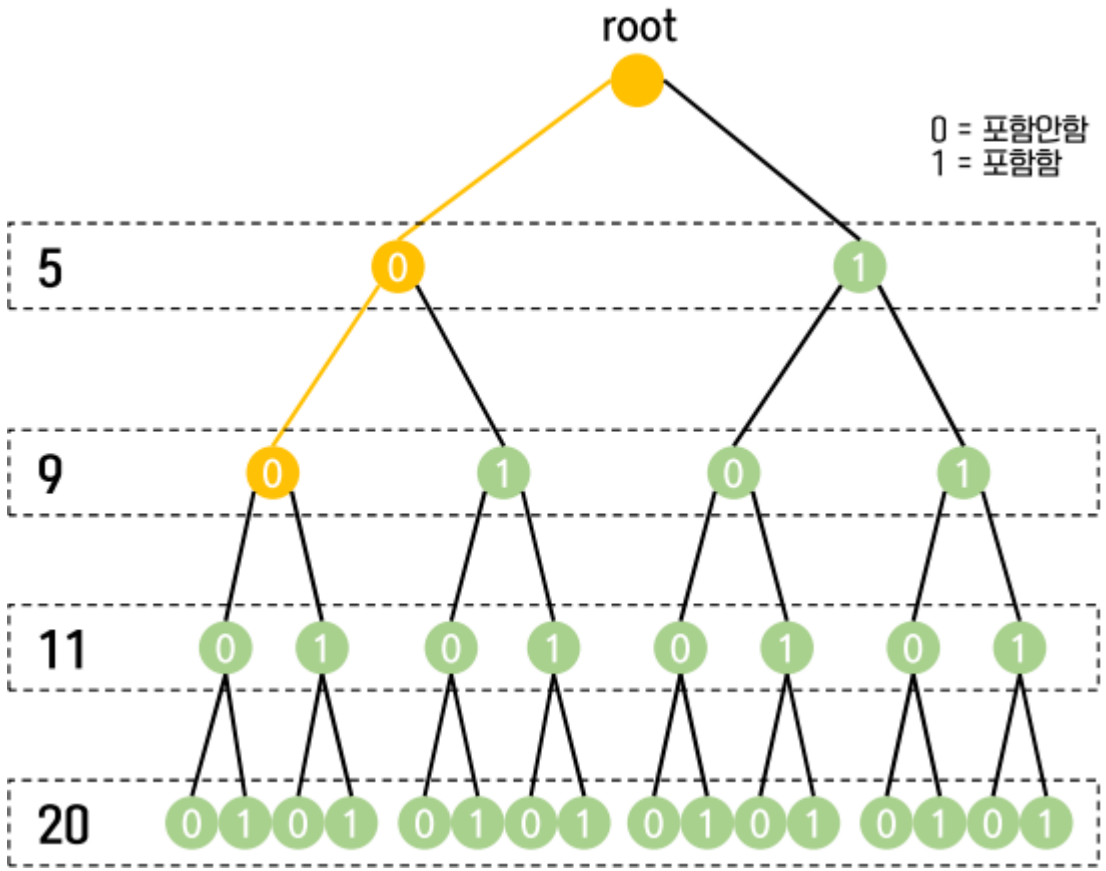


BackTracking

9를 포함하지 않으면 25보다 작나?
조건에 성립하므로 계속 탐색

순회할 때마다 조건과 비교

Tuple(0, 0)

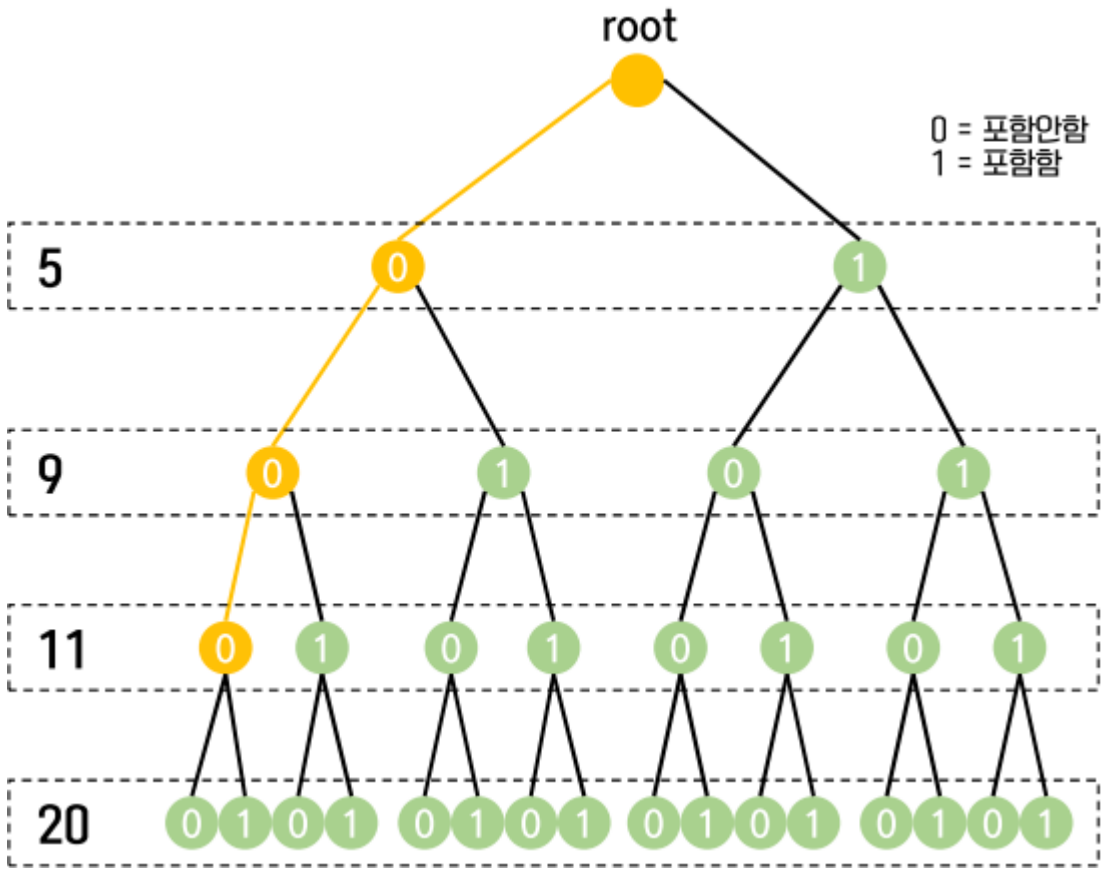


BackTracking

11를 포함하지 않으면 25보다 작나?
조건에 성립하므로 계속 탐색

순회할 때마다 조건과 비교

Tuple(0, 0, 0)



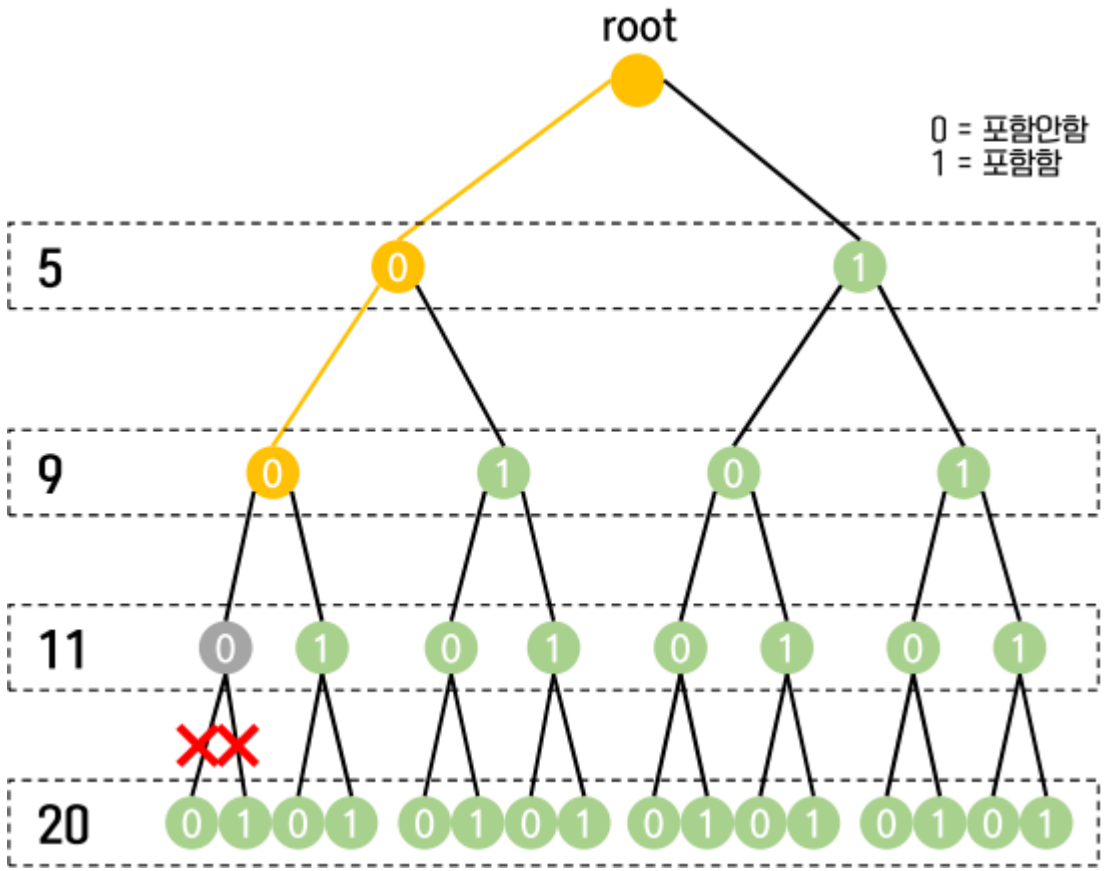
BackTracking

남은 Level의 합이 25이상인가?

성립하지 않으므로 부모 노드로 돌아감

순회할 때마다 조건과 비교

Tuple(0, 0)

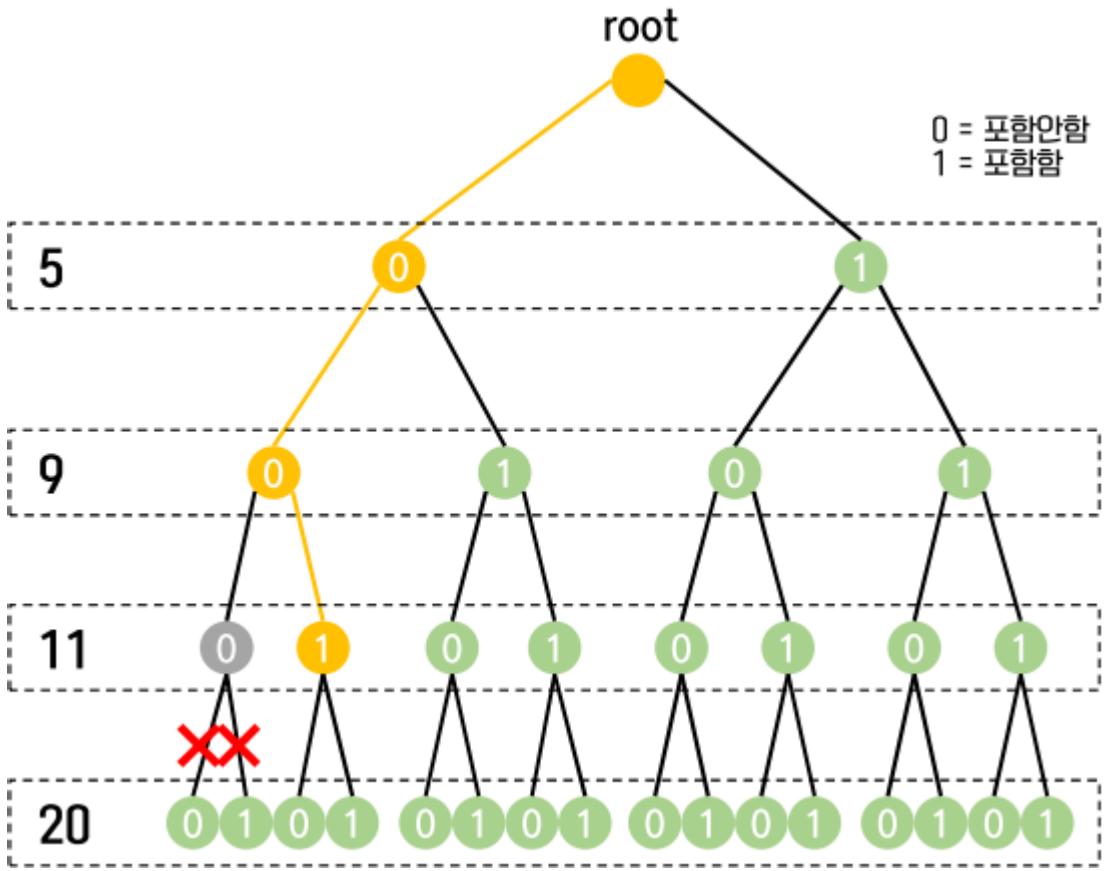


BackTracking

11를 포함해도 25보다 작나?
조건에 성립하므로 계속 순회

순회할 때마다 조건과 비교

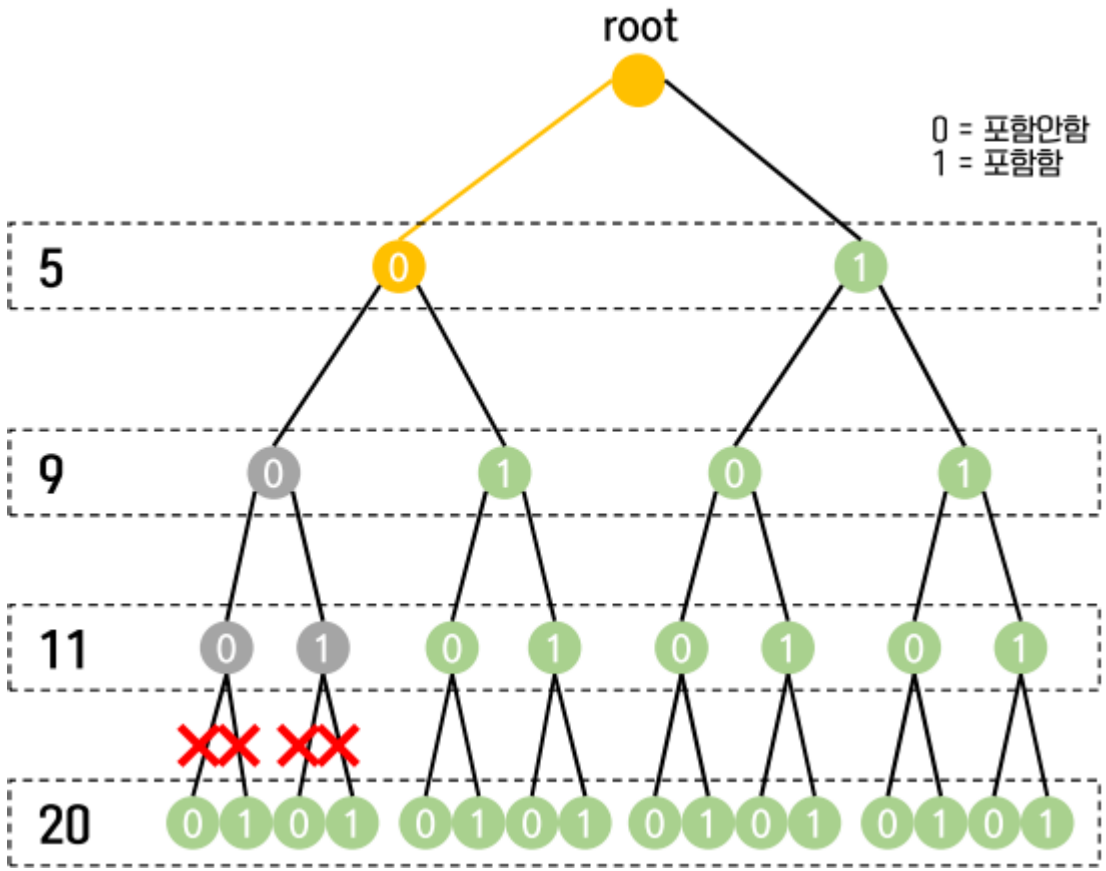
Tuple(0, 0, 1)



BackTracking

지금까지의 방식으로 계속 반복

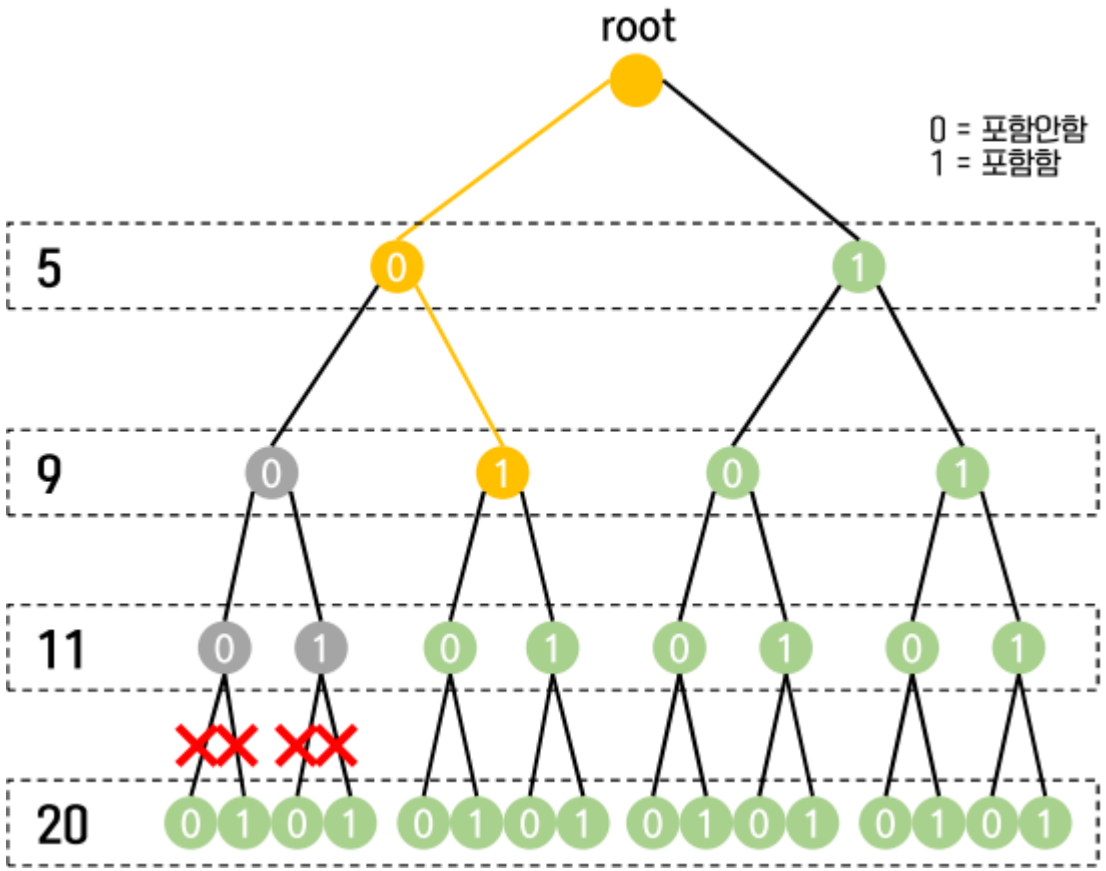
Tuple(0)



BackTracking

지금까지의 방식으로 계속 반복

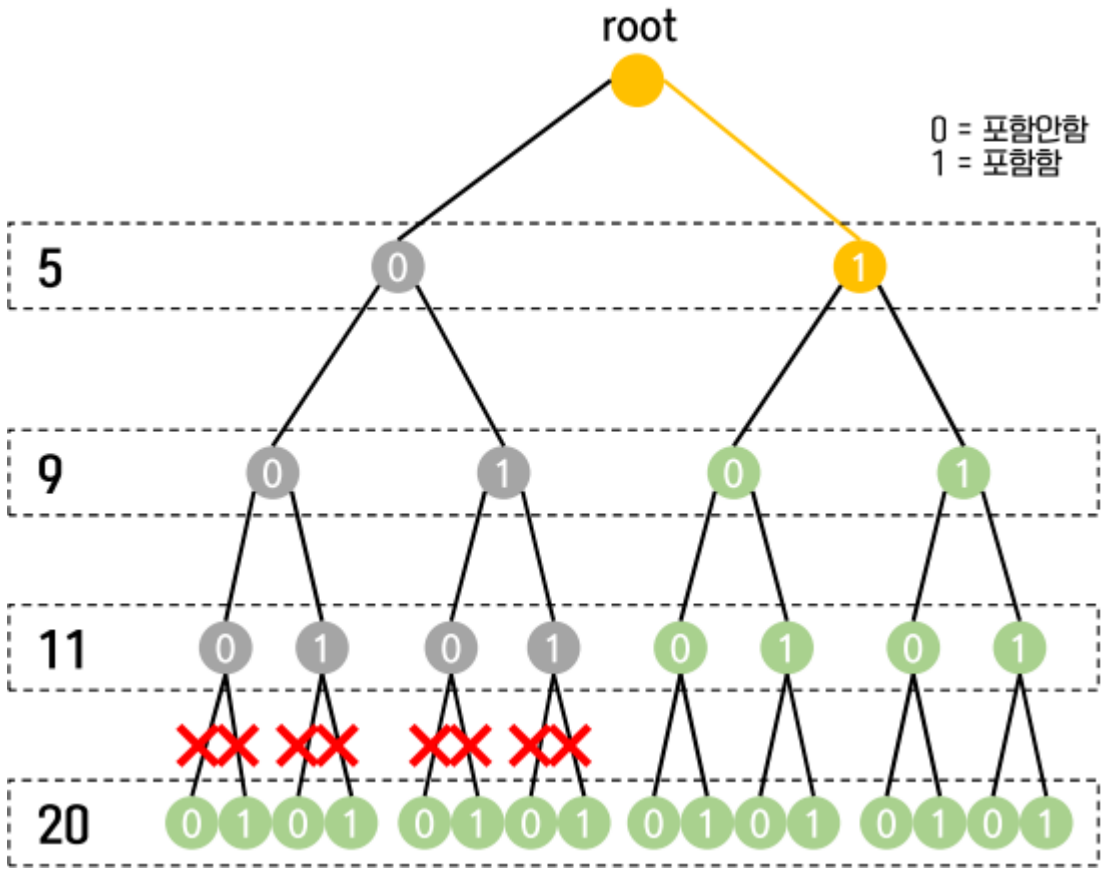
Tuple(0 ,1)



BackTracking

지금까지의 방식으로 계속 반복

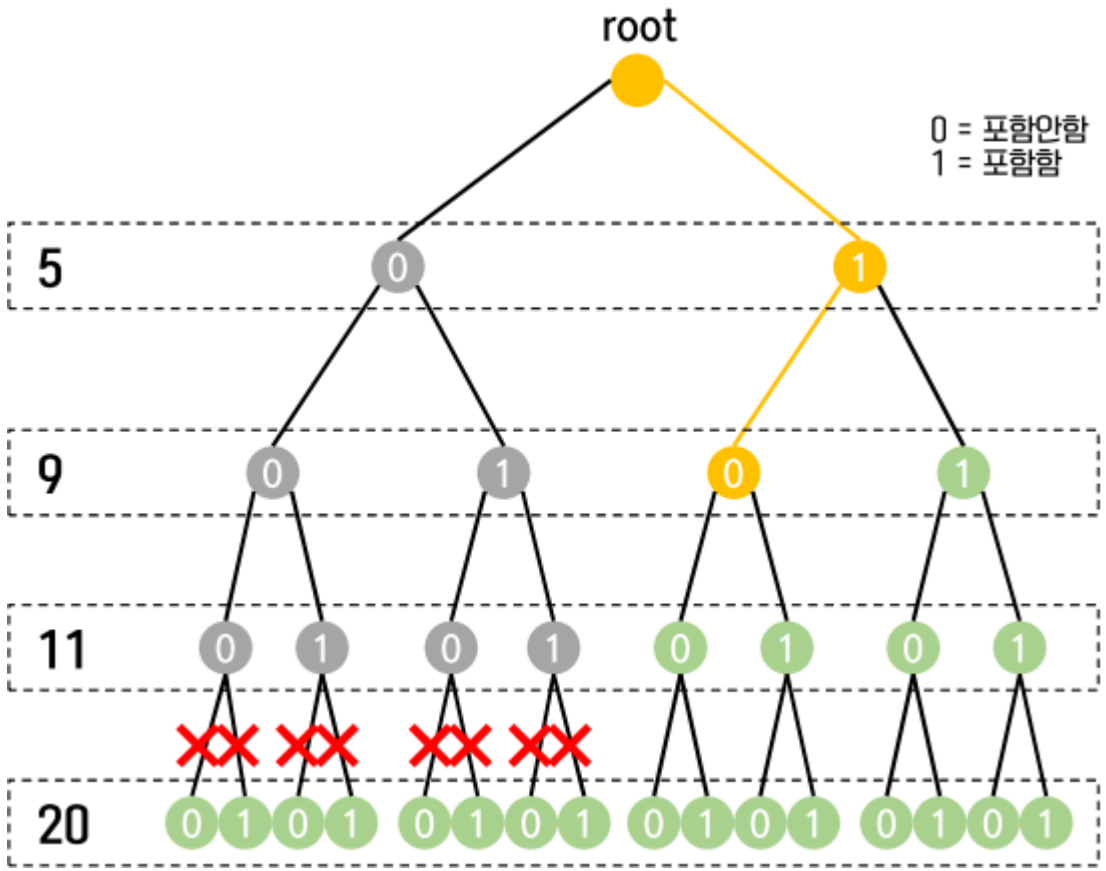
Tuple(1)



BackTracking

지금까지의 방식으로 계속 반복

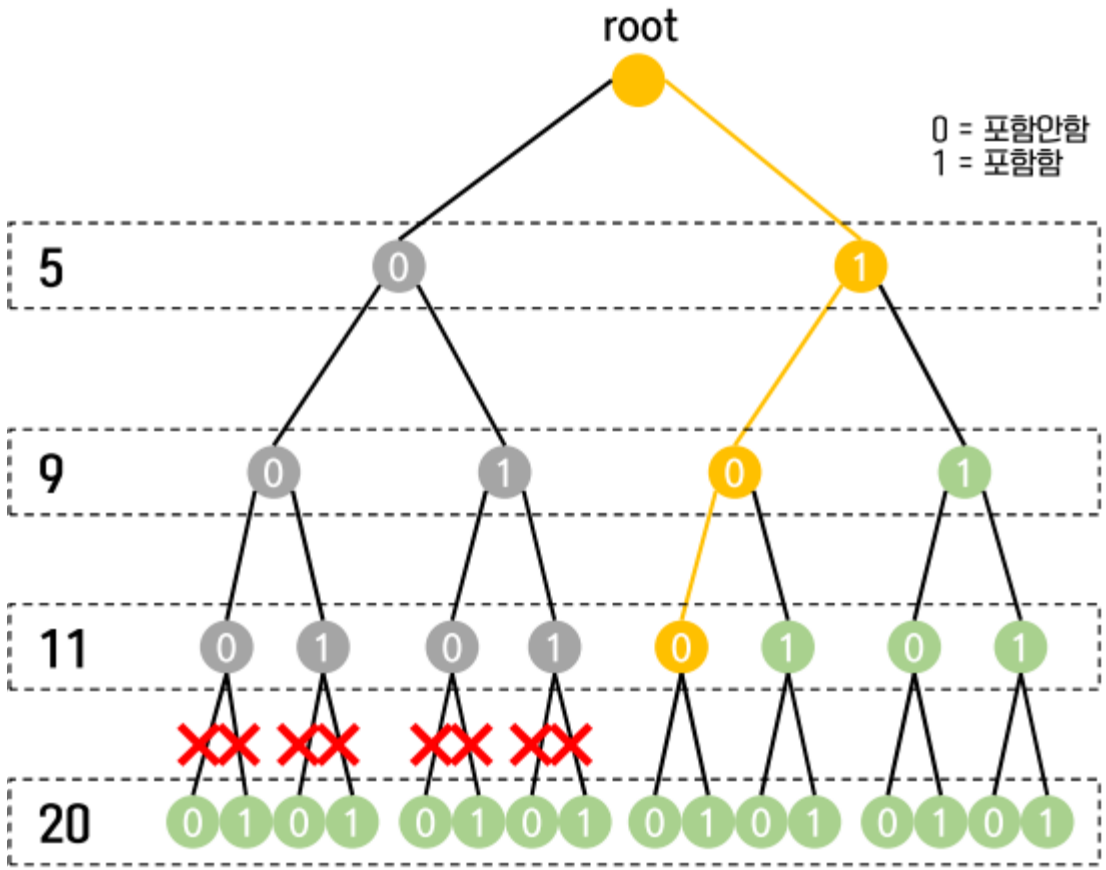
Tuple(1, 0)



BackTracking

지금까지의 방식으로 계속 반복

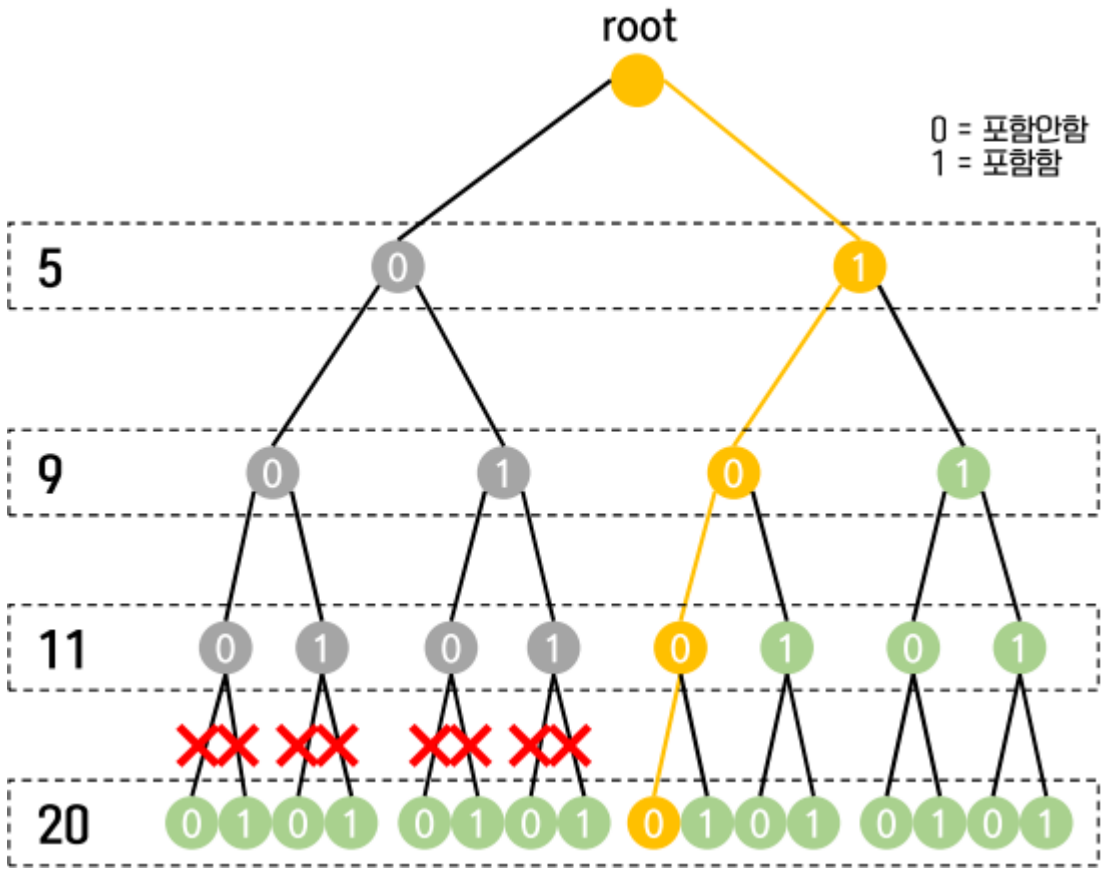
Tuple(1, 0, 0)



BackTracking

지금까지의 방식으로 계속 반복

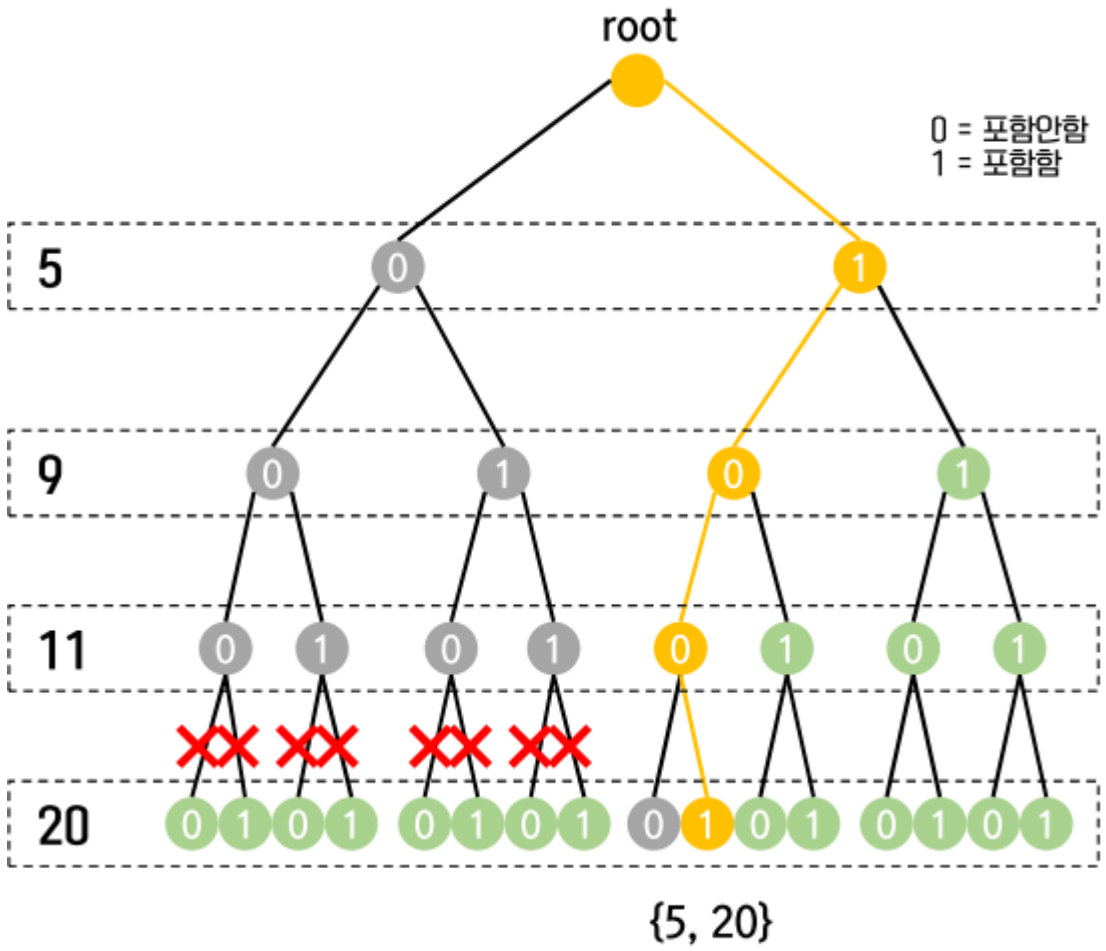
Tuple(1, 0, 0, 0)



BackTracking

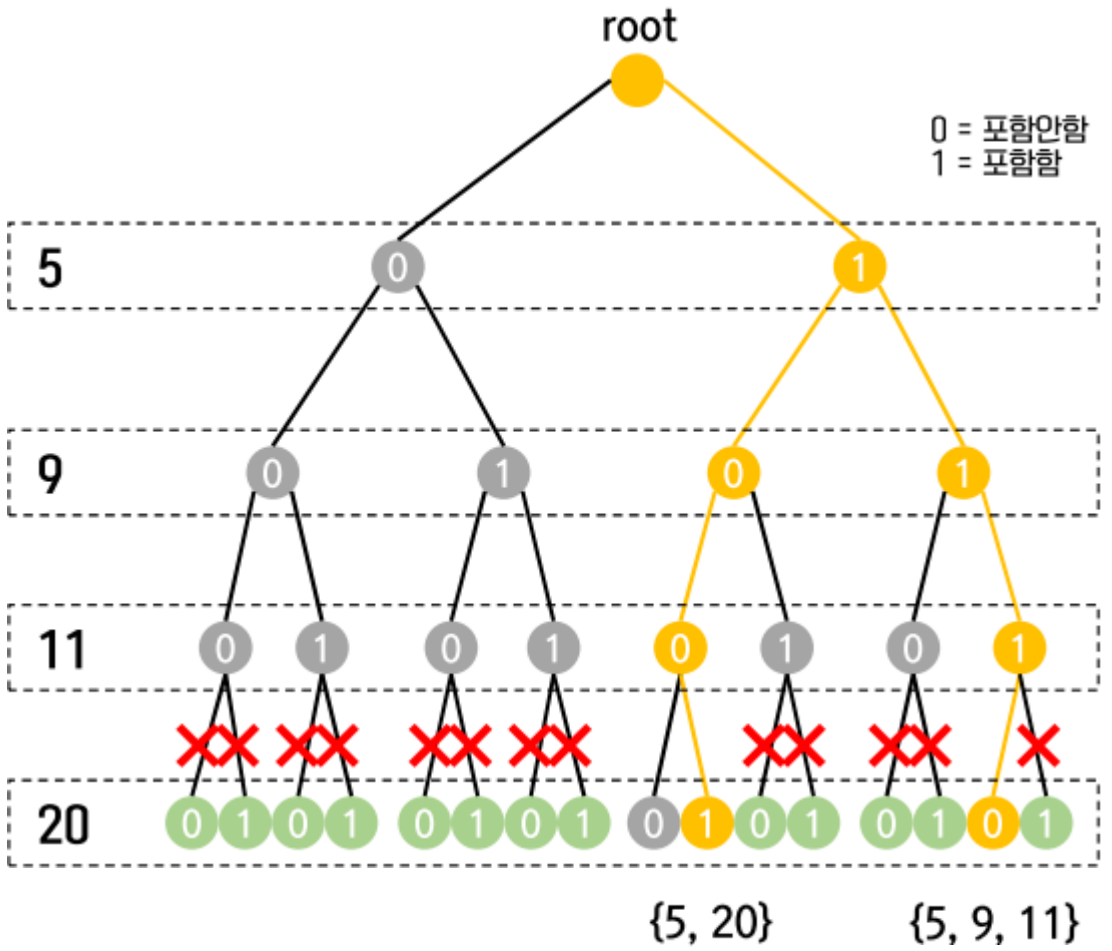
정답을 찾거나 **모든 노드가 죽으면**
탐색 종료

Tuple(1, 0, 0, 1)



BackTracking

모든 정답을 찾고 싶으면
모든 노드가 죽을 때까지 탐색



BackTracking

근데 트리의 노드와 간선을 모두 저장하면 메모리를 많이 사용하지 않을까?

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

BackTracking

첫 번째 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

첫 번째 방법 - 모든 해를 만들고 최적해인지 검사 (쉬움)

```
int solution[5];
// main 에서는 root부터 시작하므로 Backtracking(0)으로 시작
Ba                    depth                    깊이 확인
if                    //깊이
                      // 해인지
                  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
    }
}
```

해가 많이 존재할 때에는 어떻게?

BackTracking

두 번째 방법 – 현재 노드의 자식 노드들이 최적해일 수 있나 판단 (가지치기)

```
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
        // 지금까지의 합이 25이상이거나 지금부터 앞으로의 합이 25보다 작으면 넘어가지 말고 continue; 사용
        Backtracking(depth + 1); // 다음 자식노드로 이동함
    }
}
```

BackTracking

이렇듯 모든 해를 찾지 말고 가지치기를 하면서
탐색하는 것이 BackTracking의 핵심입니다!!

풀어볼까유

#1182
부분수열의 합

N <= 20에 유의

풀어볼까유

#1759

암호 만들기 (LIS)

(next_permutation)

풀어볼까유

#2661
좋은 수열

가지 치기

풀어볼까유

#1987 알파벳

가지 치기

풀어볼까유

#9663
N-Queen

백트래킹의 교과서

풀어볼까유

#9663
N-Queen

백트래킹의 교과서

풀어볼까유

#1799
비숍

N-Queen과 비슷?

풀어볼까유

#2580 스도쿠

스도쿠 빈칸 채우기

다음 시간에 만나요~