[스파르타코딩클럽] 알고보면 알기쉬운 알고리즘 - 4주차

매 주차 강의자료 시작에 PDF파일을 올려두었어요!

▼ PDF 파일

[수업 목표]

1. 트리, 힙의 개념과 활용법에 대해 배운다.
2. 그래프, BFS, DFS 에 대해 배워보자.
3. Dynamic Programming 의 개념과 그 필요성을 배워보자.

[목차]

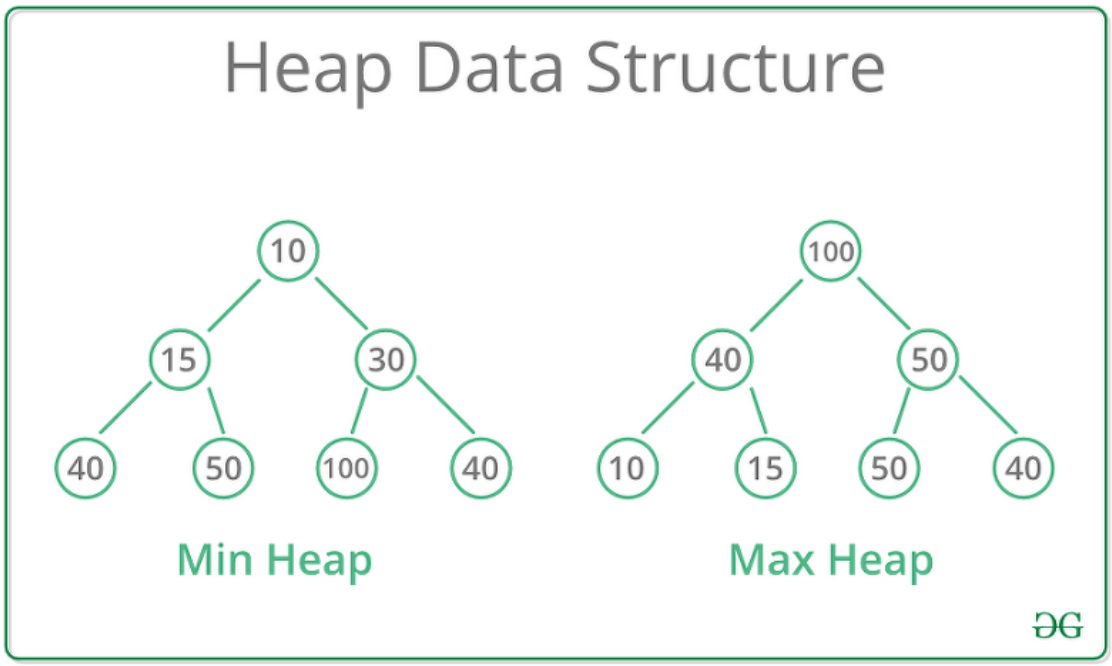
모든 토글을 열고 닫는 단축키
Windows : `Ctrl` + `alt` + `t`
Mac : `⌘` + `⌥` + `t`

---

01. 오늘 배울 것

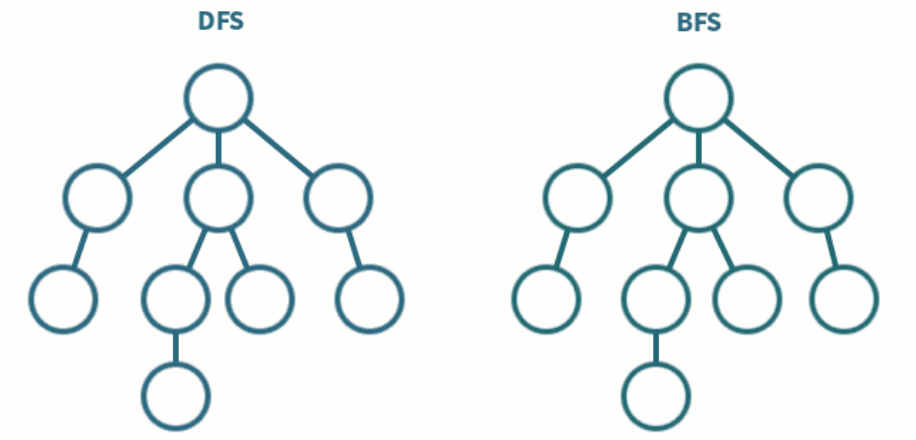
▼ 트리, 힙

트리와 힙이라는 자료구조에 대해 배워보겠습니다!
트리라는 자료구조를 이용하면 계층 구조의 데이터를 쉽게 표현할 수 있고,
힙이라는 자료구조를 이용하면 최댓값과 최솟값을 쉽게 뽑을 수 있습니다!



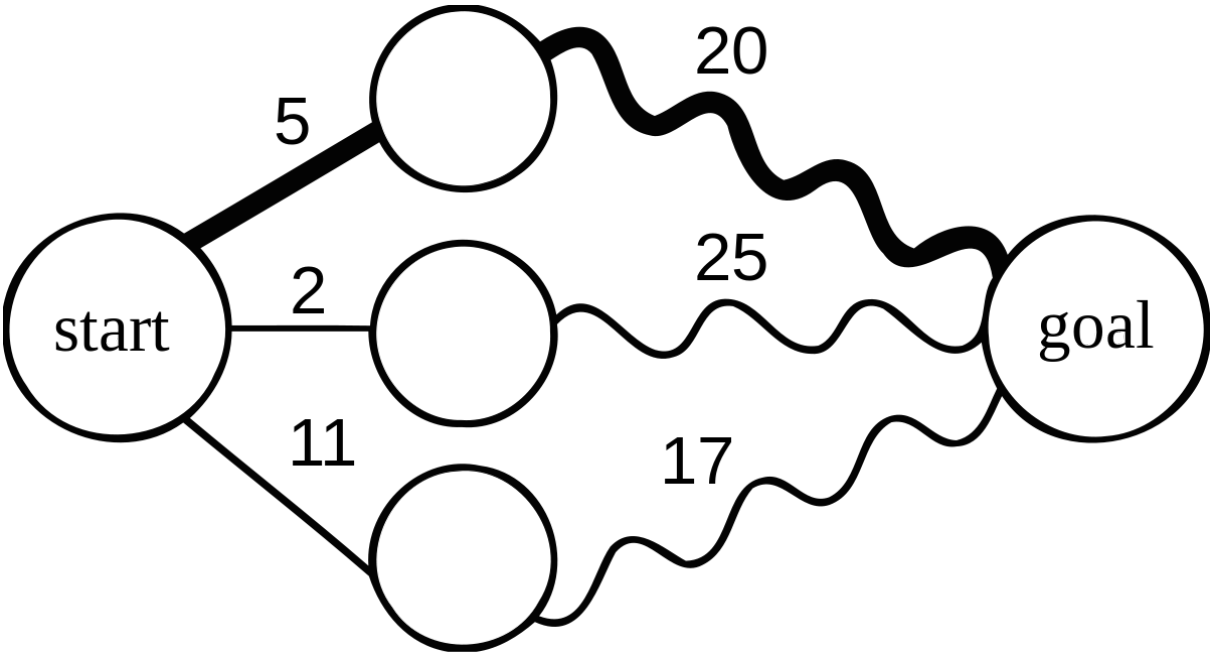
▼ BFS, DFS

👉 BFS 와 DFS 는
Breadth **F**irst **S**earch
Depth **F**irst **S**earch 의 약자로,
탐색의 순서를 깊이 우선으로 할 것인가, 너비 우선으로 할 것인가에 대한 방법입니다.
이런 탐색을 하는 이유와 장단점을 알아보시죠!



▼ Dynamic Programming

👉 Dynamic Programming,
동적 계획법은 부분 문제의 해를 통해 전체 문제를 해결하는 방법입니다!
DP 라고 줄여 말하기도 하는 이 방법론은 알고리즘 문제를 해결하는데 많이 사용되곤 합니다.
이따 더 자세하게 배워보겠습니다!



02. 트리 - 1

▼ 1) 트리란?

📖 뿌리와 가지로 구성되어 거꾸로 세워놓은 나무처럼 보이는 계층형 비선형 자료 구조.

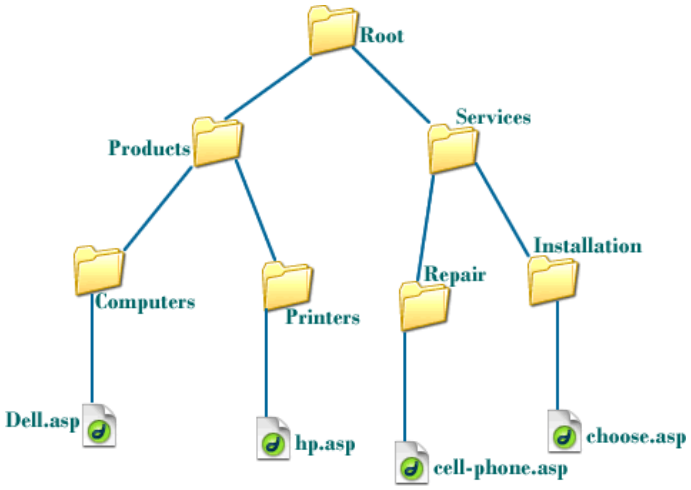
[네이버 지식백과] 트리 [tree] (천재학습백과 초등 소프트웨어 용어사전)

👉 앞서 보인 큐(Queue), 스택(Stack) 은 자료구조에서 선형 구조라고 합니다.
선형 구조란 자료를 구성하고 있는 데이터들이 순차적으로 나열시킨 형태를 의미합니다.

이번에 배울 트리는 바로 비선형 구조입니다!
비선형 구조는 선형구조와는 다르게 데이터가 계층적 혹은 망으로 구성되어있습니다.
선형구조와 비선형구조의 차이점은 형태뿐만 아니라 용도에서도 차이점이 많습니다!

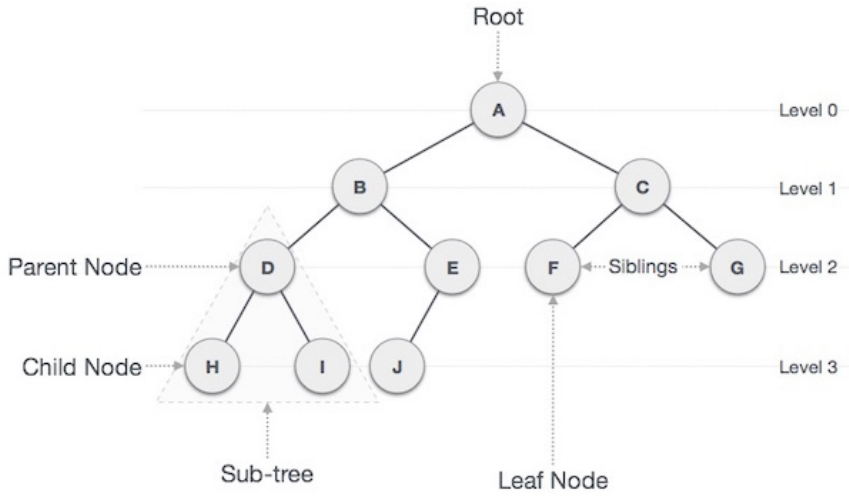
선형구조는 자료를 저장하고 꺼내는 것에 초점이 맞춰져 있고,
비선형구조는 표현에 초점이 맞춰져 있습니다.

* 아래 폴더 구조가 대표적인 트리의 형태입니다!



👉 트리는 이름에서부터 느껴지듯이 계층형 구조입니다!
위 아래가 구분되어 있습니다.
트리에서 나오는 용어들에 대해 언급하고 가겠습니다!

Node: 트리에서 데이터를 저장하는 기본 요소
Root Node: 트리 맨 위에 있는 노드
Level: 최상위 노드를 Level 0으로 하였을 때, 하위 Branch로 연결된 노드의 깊이를 나타냄
Parent Node: 어떤 노드의 상위 레벨에 연결된 노드
Child Node: 어떤 노드의 하위 레벨에 연결된 노드
Leaf Node(Terminal Node): Child Node가 하나도 없는 노드
Sibling: 동일한 Parent Node를 가진 노드
Depth: 트리에서 Node가 가질 수 있는 최대 Level



https://www.google.com/url?sa=i&url=https%3A%2F%2Fplanbs.tistory.com%2Fentry%2F%EC%9E%90%EB%A3%8C%EA%B5%AC%EC%A1%B0-Tree%EC%99%80-Tree%EC%9D%98-%ED%91%9C%ED%98%84-%EB%B0%A9%EC%8B%9D&psig=AOvVaw07pW3lco-Rf2znUBMf5UBI&ust=1603354258589000&source=images&cd=vfe&ved=0CAIQjRxqFwoTCKCdjpqexewCFQAAAAAdAAAAABAF

▼ 2) 트리의 종류

👉 트리는 이진 트리, 이진 탐색 트리, 균형 트리(AVL 트리, red-black 트리), 이진 힙(최대힙, 최소힙) 등 되게 다양한 트리가 있지만..! 저희는 이진 트리와 완전 이진 트리만 배워보겠습니다.

이진 트리(Binary Tree)의 특징은 바로
각 노드가 최대 두 개의 자식을 가진다는 것입니다.
막 하위의 노드가 4~5개 일 수 없습니다. 무조건 0, 1, 2 개만 있어야 합니다!

```
   o      Level 0
 o o o    Level 1
o  o  o   Level 2 # 이진 트리(X)

   o      Level 0
 o   o    Level 1
o o o     Level 2 # 이진 트리(O)
```

👉 완전 이진 트리(Complete Binary Tree)의 특징은 바로
노드를 삽입할 때 최하단 왼쪽 노드부터 차례대로 삽입해야 한다는 것 입니다!

```
   o      Level 0
 o   o    Level 1
  o o     Level 2  # -> 이진 트리 O 완전 이진 트리 X

   o      Level 0
 o   o    Level 1
o o o     Level 2  # -> 이진 트리 O 완전 이진 트리 O
```

03. 트리 - 2

▼ 3) 완전 이진 트리를 배열로 표현

👉 트리 구조를 표현하는 방법은
직접 클래스를 구현해서 사용하는 방법이 있고,
배열로 표현하는 방법이 있습니다.

엇, 트리 구조를 어떻게 배열에 저장할 수 있을까요?
바로 **완전 이진 트리**를 쓰는 경우에 사용할 수 있습니다!

완전 이진 트리는 왼쪽부터 데이터가 쌓이게 되는데,
이를 순서대로 배열에 쌓으면서 표현할 수 있습니다.

예를 들어 아래와 같은 완전 이진 트리를 배열에 표현한다면, 다음과 같습니다!

```
트리를 구현할 때는 편의성을 위해 0번째 인덱스는 사용되지 않습니다!
그래서 None 값을 배열에 넣고 시작합니다! [None]

      8      Level 0 -> [None, 8] 첫번째 레벨의 8을 넣고,
    6   3    Level 1 -> [None, 8, 6, 3] 다음 레벨인 6, 3을 넣고
   4 2 5     Level 2 -> [None, 8, 6, 3, 4, 2, 5] 다음 레벨인 4, 2, 5를 넣으면 됩니다!

자 그러면, [None 8, 6, 3, 4, 2, 5] 라는 배열이 되는데
다시 역으로 이 배열을 활용해서 트리 구조를 분석해보겠습니다.
다음과 같은 방법으로 트리 구조를 파악할 수 있습니다.

1. 현재 인덱스 * 2 -> 왼쪽 자식의 인덱스
2. 현재 인덱스 * 2 + 1 -> 오른쪽 자식의 인덱스
3. 현재 인덱스 // 2 -> 부모의 인덱스

예를 들어서 1번째 인덱스인 8의 왼쪽 자식은 6, 오른쪽 자식은 3 입니다.
그러면 1 * 2 = 2번째 인덱스! 6!
그러면 1 * 2 + 1 = 3번째 인덱스! 3! 입니다!
부모를 찾아보면, 3 // 2 = 1번째 인덱스 8 이므로 부모를 찾을 수 있습니다.

이를 다시 생각해보면
[None, 8, 6, 3, 4, 2, 5] 는
8 밑에 6, 3 이 있고, 6, 3 밑에 4, 2, 5가 있는 완전 이진 트리구나! 생각할 수 있습니다.
```

▼ 4) 완전 이진 트리의 높이

👉 트리의 높이(Height)는, 루트 노드부터 가장 아래 리프 노드까지의 길이 입니다!
예를 들어 다음과 같은 트리의 높이는 2라고 할 수 있습니다.

```
      o      Level 0  # 루트 노드
    o   o    Level 1
   o o o     Level 2  # 가장 아래 리프 노드

이 트리의 높이는 ? 2 - 0 = 2!
```

🧐 그러면 한 번, 각 레벨에 노드가 꽉~~ 차 있으면 몇 개가 있는지 잠깐 생각해보겠습니다.

아래 예시를 보면, 레벨을 k라고 한다면
각 레벨에 최대로 들어갈 수 있는 노드의 개수는
2^k 개수 임을 알 수 있습니다.

```
     1           Level 0 -> 1개
   2   3         Level 1 -> 2개
  4 5 6 7        Level 2 -> 4개
 8 9..... 14 15  Level 3 -> 8개
                 Level k -> 2^k 개
```

❓ 만약 높이가 h 인데 모든 노드가 꽉 차 있는 완전 이진 트리라면
모든 노드의 개수는 몇개일까요?

```
1 + 2^1 + 2^2 + 2^3 + 2^4 ..... 2^h

즉, 이를 수식으로 표현하면 2^h - 1 이 됩니다!
```

😎 즉, 높이가 h 일 때 최대 노드의 개수는 $2^h - 1$ 개 입니다.

그러면 이 때 최대 노드의 개수가 N이라면, h 는 뭘까요?

$2^h - 1 = N$
→
$h = log_2(N + 1)$

라고 할 수 있습니다!
즉! 완전 이진 트리 노드의 개수가 N일 때 최대 높이가 $log_2(N + 1)$ 이므로

이진 트리의 높이는 최대로 해봤자 $O(log(N))$ 이 구나! 라고 말씀하시면 됩니다.

04. 힙

▼ 4) 힙이란?

📖 힙은 데이터에서 최대값과 최소값을 빠르게 찾기 위해 고안된 완전 이진 트리(Complete Binary Tree)

💡 항상 최대의 값들이 필요한 연산이 있다면? 바로 힙을 쓰면 되겠죠!

그러면 힙을 구현하려면 어떻게 해야할까요?

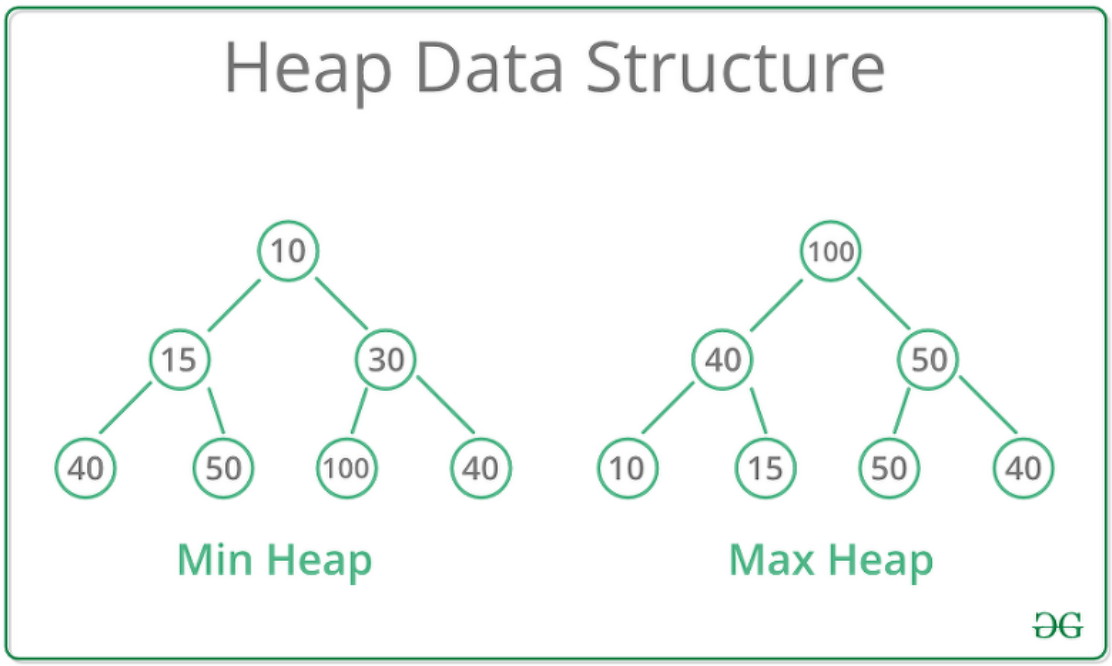
힙은 항상 큰 값이 상위 레벨에 있고 작은 값이 하위 레벨에 있도록 하는 자료구조입니다.
다시 말하면 부모 노드의 값이 자식 노드의 값보다 항상 커야 합니다.
그러면 가장 큰 값은 모든 자식보다 커야 하기 때문에 가장 위로 가겠죠!
따라서 최대의 값들을 빠르게 구할 수 있게 되는 것입니다.

예시를 들어보면 다음과 같습니다!

```
   8      Level 0
 6   3    Level 1
  2 1     Level 2  # -> 이진 트리 O 완전 이진 트리 X 이므로 힙이 아닙니다!

   8      Level 0
 6   3    Level 1  # -> 이진 트리 O 완전 이진 트리 O 인데 모든 부모 노드의 값이
4 2 1     Level 2  # 자식 노드보다 크니까 힙이 맞습니다!


   8      Level 0
 6   3    Level 1  # -> 이진 트리 O 완전 이진 트리 O 인데 모든 부모 노드의 값이
4 2 5     Level 2  # 자식 노드보다 크지 않아서 힙이 아닙니다..!
```

💡 참고로, 힙은 다음과 같이 최대값을 맨 위로 올릴수도 있고,
최솟값을 맨 위로 올릴 수도 있습니다!

최댓값이 맨 위인 힙을 Max 힙,
최솟값이 맨 위인 힙을 Min 힙이라고 합니다!



▼ 5) ✍️맥스 힙에 원소 추가

👉 힙의 규칙

`힙은 항상 큰 값이 상위 레벨에 있고 작은 값이 하위 레벨에 있어야 합니다.`

은 항상 지켜져야 합니다.

따라서, 원소를 추가하거나 삭제할때도 위의 규칙이 지켜져야 합니다!

원소를 추가할 때는 다음과 같이 하시면 됩니다.

1. 원소를 맨 마지막에 넣습니다.
2. 그리고 부모 노드와 비교합니다. 만약 더 크다면 자리를 바꿉니다.
3. 부모 노드보다 작거나 가장 위에 도달하지 않을 때까지 2. 과정을 반복합니다.

예시를 보겠습니다!

```
이 맥스 힙에서 9를 추가해보겠습니다!
      8      Level 0
    6   3    Level 1
   4 2 1     Level 2

1. 맨 마지막에 원소를 넣습니다.

      8      Level 0
    6   3    Level 1
   4 2 1 9   Level 2

2-1. 부모 노드와 비교합니다. 3보다 9가 더 크니까! 둘의 자리를 변경합니다.

      8      Level 0
    6   9    Level 1
   4 2 1 3   Level 2
```

```
      8      Level 0
    6   9    Level 1
  4 2 1 3    Level 2

2-2. 다시 부모 노드와 비교합니다. 8보다 9가 더 크니까! 둘의 자리를 변경합니다.

      8      Level 0
    6   9    Level 1
  4 2 1 3    Level 2

      9      Level 0
    6   8    Level 1
  4 2 1 3    Level 2

3. 가장 위에 도달했으므로 멈춥니다. 힙의 특성을 그대로 유지해 데이터를 삽입했습니다!

      9      Level 0
    6   8    Level 1
  4 2 1 3    Level 2
```

👉 위의 개념을 한 번 코드로 같이 작성하러 가보시죠!

기본 코드를 제공해드릴테니, 어렵지 않으실거에요! ㅎㅎ

❓ Q. 맥스 힙은 원소를 추가한 다음에도 맥스 힙의 규칙을 유지해야 한다.
맥스 힙에 원소를 추가하시오.

- 이 문제를 풀기 위해서는 어떻게 해야 할까요? 아래 코드를 복사 붙여넣기 하고 함수를 작성해보세요! 2분 정도 고민해 본 다음, 아래 방법들을 펼쳐 봅시다!

▼ [코드스니펫] 맥스 힙에 원소 추가

```
class MaxHeap:
    def __init__(self):
        self.items = [None]

    def insert(self, value):
        # 구현해보세요!
        return


max_heap = MaxHeap()
max_heap.insert(3)
max_heap.insert(4)
max_heap.insert(2)
max_heap.insert(9)
print(max_heap.items)  # [None, 9, 4, 2, 3] 가 출력되어야 합니다!
```

▼ 해결 방법

✅ 아까 생각했던 방법 그대로 코드로 옮기시면 됩니다!

우선 전체 배열에 값을 추가하신 다음에
그 원소의 인덱스인 len(self.items) - 1 부터 시작하면 됩니다!
append 함수로 맨 마지막에 넣었으니 맨 뒤 인덱스니까요!

그 인덱스부터 부모 노드의 인덱스의 노드와 값을 비교합니다.
만약 더 크다면 값을 교체하고 cur_index 에 parent_index 를 넣습니다.

이 과정을 cur_index 가 제일 꼭대기 칸, 1이 되기 전까지 반복하시면 됩니다!

```
...
    def insert(self, value):
        self.items.append(value)
        cur_index = len(self.items) - 1

        while cur_index > 1:  # cur_index 가 1이 되면 정상을 찍은거라 다른 것과 비교 안하셔도 됩니다!
            parent_index = cur_index // 2
            if self.items[parent_index] < self.items[cur_index]:
                self.items[parent_index], self.items[cur_index] = self.items[cur_index], self.items[parent_index]
                cur_index = parent_index
            else:
                break
...
```

▼ 시간 복잡도 분석

❓ 자 여러분, 방금 만든 insert 함수의 시간 복잡도는 과연 어떨까요?

✅ 한 번, 아까의 예시를 다시 보겠습니다.

```
이 맥스 힙에서 9를 추가해보겠습니다!
      8      Level 0
    6   3    Level 1
   4 2 1     Level 2

1. 맨 마지막에 원소를 넣습니다.

      8      Level 0
    6   3    Level 1
   4 2 1 9   Level 2

2-1. 부모 노드와 비교합니다. 3보다 9가 더 크니까! 둘의 자리를 변경합니다.

      8      Level 0
    6   3    Level 1
   4 2 1 9   Level 2

      8      Level 0
    6   9    Level 1
   4 2 1 3   Level 2

2-2. 다시 부모 노드와 비교합니다. 8보다 9가 더 크니까! 둘의 자리를 변경합니다.

      8      Level 0
    6   9    Level 1
   4 2 1 3   Level 2
```

```
      9      Level 0
    6   8    Level 1
   4 2 1 3   Level 2

3. 가장 위에 도달했으므로 멈춥니다. 힙의 특성을 그대로 유지해 데이터를 삽입했습니다!

      9      Level 0
    6   8    Level 1
   4 2 1 3   Level 2
```

✅ 이 설명을 보면, 결국 원소를 맨 밑에 넣어서 꼭대기까지 비교하면서 올리고 있습니다.

완전 이진트리의 최대 높이는 $O(log(N))$ 이라고 말씀 드렸었죠!
그러면, 반복하는 최대 횟수도 $O(log(N))$ 입니다.

즉! 맥스 힙의 원소 추가는 $O(log(N))$ 만큼의 시간 복잡도를 가진다고 분석할 수 있습니다.

▼ 6) ✍️맥스 힙의 원소 제거

👉 최대 힙에서 원소를 삭제하는 방법은 최댓값, 루트 노드를 삭제하는 것입니다!

스택과 같이 맨 위에 있는 원소만 제거할 수 있고, 다른 위치의 노드를 삭제할 수는 없습니다!

마찬가지로, 원소를 삭제할때도 힙의 규칙이 지켜져야 합니다!

아래와 같은 방법으로 하면 됩니다.

1. 루트 노드와 맨 끝에 있는 원소를 교체한다.
2. 맨 뒤에 있는 원소를 (원래 루트 노드)를 삭제한다.
3. 변경된 노드와 자식 노드들을 비교합니다.
두 자식 중 더 큰 자식과 비교해서 더 크다면 자리를 바꿉니다.
4. 자식 노드 둘 보다 부모 노드가 크거나 가장 바닥에 도달하지 않을 때까지 3. 과정을 반복합니다.
5. 2에서 제거한 원래 루트 노드를 반환합니다.

예시를 보겠습니다!

```
이 맥스 힙에서 원소를 제거해보겠습니다! (항상 맨 위의 루트 노드가 제거 됩니다.)
      8      Level 0
    7   6    Level 1
   2 5 4     Level 2

1. 루트 노드와 맨 끝에 있는 원소를 교체한다.

      8      Level 0
    7   6    Level 1
   2 5 4     Level 2

      4      Level 0
    7   6    Level 1
   2 5 8     Level 2

2. 맨 뒤에 있는 원소를 (원래 루트 노드)를 삭제한다. (이 값을 반환해줘야 합니다!)

      4      Level 0
    7   6    Level 1
```

```
   2 5 X      Level 2

3-1. 변경된 노드를 더 큰 자식 노드와 비교해야 합니다.
우선 부모와 왼쪽 자식을 비교합니다.
그리고 부모와 오른쪽 자식을 비교합니다.
그리고 부모 보다 큰 자식 중, 더 큰 자식과 변경해야 합니다.
왼쪽 자식인 7과 오른쪽 자식인 6 중에서 7이 더 크고, 부모인 4보다 크니까 둘의 자리를 변경합니다.

      4      Level 0
    7   6    Level 1
   2 5       Level 2

      7      Level 0
    4   6    Level 1
   2 5       Level 2

3-2. 다시 자식 노드와 비교합니다.
우선 부모와 왼쪽 자식을 비교합니다.
왼쪽 자식인 2는 부모인 4보다 작으니까, 제외 합니다.

      7      Level 0
    4   6    Level 1
   2 5       Level 2

이번에는 오른쪽 자식과 비교합니다. 5가 4보다 더 크니까 둘의 자리를 변경합니다.

      7      Level 0
    4   6    Level 1
   2 5       Level 2

      7      Level 0
    5   6    Level 1
   2 4       Level 2

4. 가장 아래 레벨에 도달했으므로 멈춥니다. 힙의 특성을 그대로 유지해 데이터를 삭제했습니다!

      7      Level 0
    5   6    Level 1
   2 4       Level 2

5. 그리고, 아까 제거한 원래 루트 노드, 8을 반환하면 됩니다!
```

👉 위의 개념을 한 번 코드로 같이 작성하러 가보시죠!

기본 코드를 제공해드릴테니, 어렵지 않으실거에요! ㅎㅎ

❓ Q. 맥스 힙은 원소를 제거한 다음에도 맥스 힙의 규칙을 유지해야 한다.
맥스 힙의 원소를 제거하시오.

- 이 문제를 풀기 위해서는 어떻게 해야 할까요? 아래 코드를 복사 붙여넣기 하고 함수를 작성해보세요! 2분 정도 고민해 본 다음, 아래 방법들을 펼쳐 봅시다!

▼ [코드스니펫] 맥스 힙의 원소 제거

```
class MaxHeap:
    def __init__(self):
        self.items = [None]

    def insert(self, value):
        self.items.append(value)
        cur_index = len(self.items) - 1

        while cur_index > 1:  # cur_index 가 1이 되면 정상을 찍은거라 다른 것과 비교 안하셔도 됩니다!
            parent_index = cur_index // 2
            if self.items[parent_index] < self.items[cur_index]:
```

```
                self.items[parent_index], self.items[cur_index] = self.items[cur_index], self.items[parent_index]
                cur_index = parent_index
            else:
                break

    def delete(self):
        # 구현해보세요!
        return 8  # 8 을 반환해야 합니다.


max_heap = MaxHeap()
max_heap.insert(8)
max_heap.insert(7)
max_heap.insert(6)
max_heap.insert(2)
max_heap.insert(5)
max_heap.insert(4)
print(max_heap.items)  # [None, 8, 7, 6, 2, 5, 4]
print(max_heap.delete())  # 8 을 반환해야 합니다!
print(max_heap.items)  # [None, 7, 5, 6, 2, 4]
```

▼ 해결 방법

아까 생각했던 방법 그대로 코드로 옮기시면 됩니다!

우선 루트 노드와 맨 끝에 있는 원소의 값을 바꿉니다!
이 때, 0번째 원소는 `None` 이 들어가 있으니
1 번째와 len(self.items) - 1 번째를 바꾸게 됩니다!

바꾸고 난 뒤, 맨 뒤의 원소를 뽑아 prev_max 변수에 저장합니다.

그리고 루트 원소의 인덱스인 1부터 비교를 시작하면 됩니다!
그 인덱스부터 자식 노드의 인덱스의 노드와 값을 비교합니다.

이 때, max_index 라는 변수에 현재 인덱스를 저장하고,

왼쪽 자식의 값과 비교합니다.
이 때, 자식의 인덱스가 배열의 사이즈보다 크지 않은지 확인해줘야 합니다!
만약 왼쪽 자식이 더 크다면 max_index에 left_child_index 를 넣습니다.

그리고 오른쪽 자식의 값과 비교합니다.
만약 오른쪽 자식이 더 크다면 max_index에 right_child_index 를 넣습니다.

그랬는데 max_index 가 현재 인덱스와 같다는 소리는?
부모 노드가 두 자식 보다 크다는 의미가 됩니다!

즉, 더 교체하지 않아도 된다는 의미이므로 반복문을 멈추고 나오면 됩니다.
이 과정을 계속 반복하시면 됩니다!

그리고 아~~까 저장해놨던 prev_max 를 반환하시면 됩니다!

```
...
    def delete(self):
        self.items[1], self.items[-1] = self.items[-1], self.items[1]
        prev_max = self.items.pop()
        cur_index = 1
```

```
        while cur_index <= len(self.items) - 1:
            left_child_index = cur_index * 2
            right_child_index = cur_index * 2 + 1
            max_index = cur_index

            if left_child_index <= len(self.items) - 1 and self.items[left_child_index] > self.items[max_index]:
                max_index = left_child_index

            if right_child_index <= len(self.items) - 1 and self.items[right_child_index] > self.items[max_index]:
                max_index = right_child_index

            if max_index == cur_index:
                break

            self.items[cur_index], self.items[max_index] = self.items[max_index], self.items[cur_index]
            cur_index = max_index

        return prev_max
...
```

▼ 시간 복잡도 분석

❓ 자 여러분, 방금 만든 delete 함수의 시간 복잡도는 과연 어떨까요?

✅ 한 번, 아까의 예시를 다시 보겠습니다.

```
이 맥스 힙에서 원소를 제거해보겠습니다! (항상 맨 위의 루트 노드가 제거 됩니다.)
      8      Level 0
    7   6    Level 1
   2 5 4     Level 2

1. 루트 노드와 맨 끝에 있는 원소를 교체한다.

      8      Level 0
    7   6    Level 1
   2 5 4     Level 2

      4      Level 0
    7   6    Level 1
   2 5 8     Level 2

2. 맨 뒤에 있는 원소를 (원래 루트 노드)를 삭제한다. (이 값을 반환해줘야 합니다!)

      4      Level 0
    7   6    Level 1
   2 5 X      Level 2

3-1. 변경된 노드를 더 큰 자식 노드와 비교해야 합니다.
우선 부모와 왼쪽 자식을 비교합니다.
그리고 부모와 오른쪽 자식을 비교합니다.
그리고 부모 보다 큰 자식 중, 더 큰 자식과 변경해야 합니다.
왼쪽 자식인 7과 오른쪽 자식인 6 중에서 7이 더 크고, 부모인 4보다 크니까 둘의 자리를 변경합니다.

      4      Level 0
    7   6    Level 1
   2 5       Level 2

      7      Level 0
    4   6    Level 1
   2 5       Level 2

3-2. 다시 자식 노드와 비교합니다.
우선 부모와 왼쪽 자식을 비교합니다.
왼쪽 자식인 2는 부모인 4보다 작으니까, 제외 합니다.
```

```
      7      Level 0
    4   6    Level 1
   2 5       Level 2

이번에는 오른쪽 자식과 비교합니다. 5가 4보다 더 크니까 둘의 자리를 변경합니다.

      7      Level 0
    4   6    Level 1
   2 5       Level 2

      7      Level 0
    5   6    Level 1
   2 4       Level 2

4. 가장 아래 레벨에 도달했으므로 멈춥니다. 힙의 특성을 그대로 유지해 데이터를 삭제했습니다!

      7      Level 0
    5   6    Level 1
   2 4       Level 2

5. 그리고, 아까 제거한 원래 루트 노드, 8을 반환하면 됩니다!
```

✅ 이 설명을 보면, 결국 원소를 맨 위에 올려서 바닥까지 비교하면서 내리고 있습니다.

완전 이진트리의 최대 높이는 $O(log(N))$ 이라고 말씀 드렸었죠!
그러면, 반복하는 최대 횟수도 $O(log(N))$ 입니다.

즉! 맥스 힙의 원소 삭제는 $O(log(N))$ 만큼의 시간 복잡도를 가진다고 분석할 수 있습니다.

05. 그래프

▼ 7) 그래프란?

📖 연결되어 있는 정점와 정점간의 관계를 표현할 수 있는 자료구조. [천재학습백과]

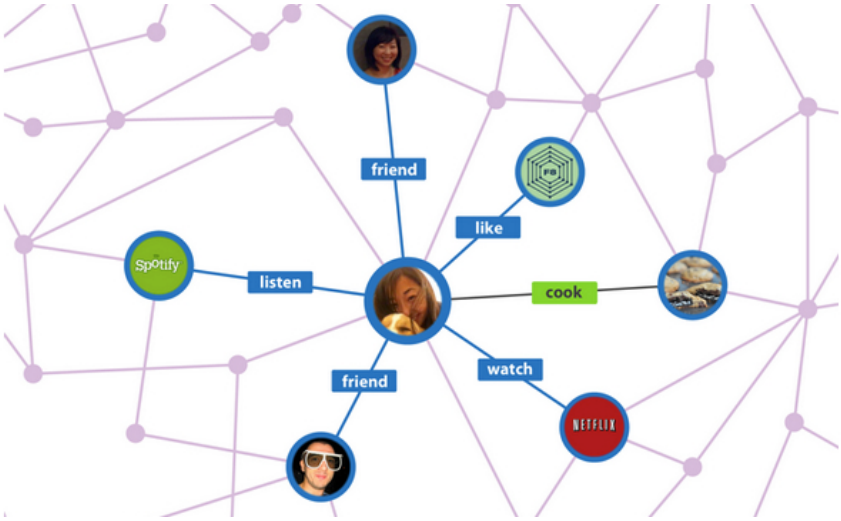
💡 저번에 트리를 배우면서 배웠던 "비선형 구조" 에 대해 기억 나시나요?

비선형 구조는 표현에 초점이 맞춰져 있다고 말씀 드렸는데,
선형구조는 자료를 저장하고 꺼내는 것에 초점이 맞춰져 있고,
비선형구조는 표현에 초점이 맞춰져 있습니다.

이번 자료구조인 그래프는 바로 **연결 관계에** 초점이 맞춰져 있습니다.

페이스북을 예시로 들어볼게요!

제가 친구 "제니"를 알고 있고, "로제"와 친합니다.
그리고 "로제"는 트와이스 "사나"를 안다고 하면,
저는 "사나"와 2촌 관계라고 말할 수 있겠죠!



👉 그래프에서 사용되는 용어들을 정리해드리면 다음과 같습니다!

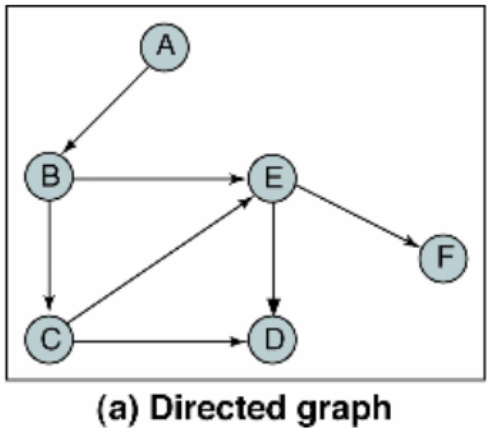
노드(Node): 연결 관계를 가진 각 데이터를 의미합니다. 정점(Vertex)이라고도 합니다.
간선(Edge): 노드 간의 관계를 표시한 선.
인접 노드(Adjacent Node): 간선으로 직접 연결된 노드(또는 정점)

예시를 한 번 들어보겠습니다!

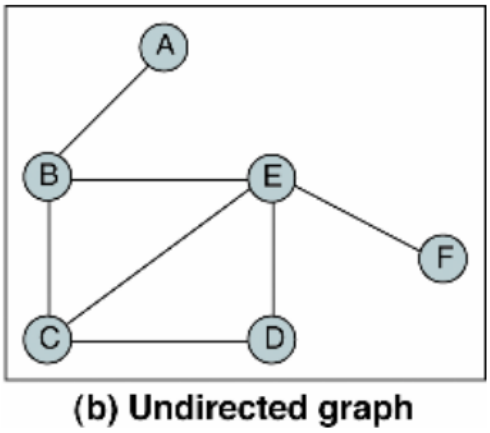
```
      로제 - 사나
       |
제니 - 르탄

르탄이는 연결 관계를 가진 데이터, 노드입니다!
르탄과 제니는 간선으로 연결되어 있습니다.
르탄과 로제는 인접 노드 입니다!
```

🧐 참고로 그래프는 유방향 그래프와 무방향 그래프 두가지가 있지만,
이번 강의에서는 **무방향 그래프**에 대해서만 배워보도록 하겠습니다!

유방향 그래프(Directed Graph): 방향이 있는 간선을 갖습니다. 간선은 단방향 관계를 나타내며, 각 간선은 한 방향으로만 진행할 수 있습니다.
무방향 그래프(Undirected Graph)는 방향이 없는 간선을 갖습니다



(a) Directed graph



(b) Undirected graph

▼ 8) 그래프의 표현 방법

👉 이런 그래프라는 개념을 컴퓨터에서 표현하는 방법은 두 가지 방법이 있습니다!

1) 인접 행렬(Adjacency Matrix): 2차원 배열로 그래프의 연결 관계를 표현
2) 인접 리스트(Adjacnecy List): 링크드 리스트로 그래프의 연결 관계를 표현

더 쉽게 표기하기 위해서 각 노드들에 번호를 매겨보겠습니다!

제니를 0, 르탄을 1, 로제를 2, 사나를 3 라고 하겠습니다.

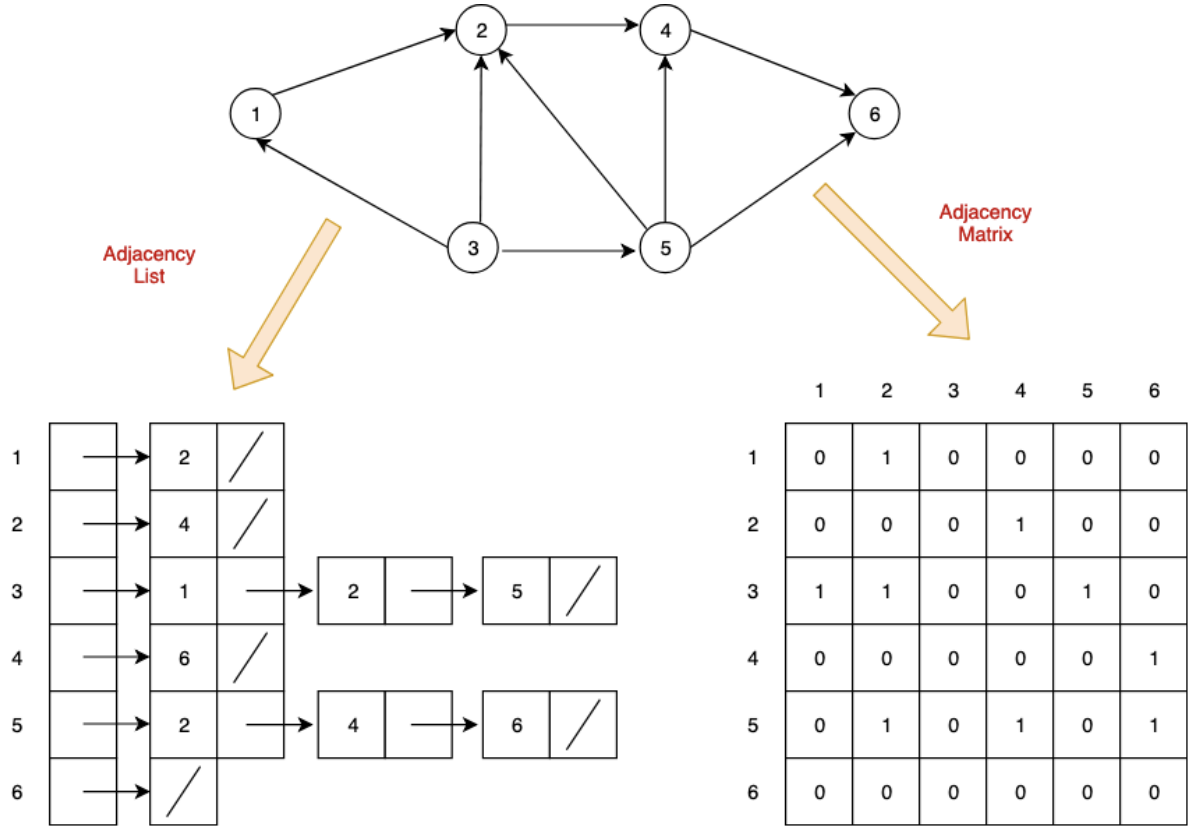
```
          2 - 3
          |
      0 - 1

1. 이를 인접 행렬, 2차원 배열로 나타내면 다음과 같습니다!
  0  1  2  3
0 X  O  X  X
1 O  X  O  X
2 X  O  X  O
3 X  X  O  X

이걸 배열로 표현하면 다음과 같습니다!
graph = [
    [False, True, False, False],
    [True, False, True, False],
    [False, True, False, True],
    [False, False, True, False]
]

2. 이번에는 인접 리스트로 표현해보겠습니다!
인접 리스트는 모든 노드에 연결된 노드에 대한 정보를 차례대로 다음과 같이 저장합니다.

0 -> 1
1 -> 0 -> 2
2 -> 1 -> 3
3 -> 2

이를 딕셔너리로 표현하면 다음과 같습니다!
graph = {
    0: [1],
    1: [0, 2]
    2: [1, 3]
    3: [2]
}
```

❓ 그러면 이 두 방식의 차이가 뭘까요?

👉 바로 시간 VS 공간 입니다!

인접 행렬으로 표현하면 즉각적으로 0과 1이 연결되었는지 여부를 바로 알 수 있습니다.
그러나, 모든 조합의 연결 여부를 저장해야 되기 때문에 $O(\text{노드}^2)$ 만큼의 공간을 사용해야 합니다.

인접 리스트로 표현하면 즉각적으로 연결되었는지 알 수 없고, 각 리스트를 돌아봐야 합니다.
따라서 연결되었는지 여부를 알기 위해서 최대 $O(\text{간선})$ 만큼의 시간을 사용해야 합니다.
대신 모든 조합의 연결 여부를 저장할 필요가 없으니 $O(\text{노드} + \text{간선})$ 만큼의 공간을 사용하면 됩니다.



06. DFS & BFS

▼ 9) DFS & BFS란?

📖 자료의 검색, 트리나 그래프를 탐색하는 방법. 한 노드를 시작으로 인접한 다른 노드를 재귀적으로 탐색해가고 **끝까지 탐색하면** 다시 위로 와서 다음을 탐색하여 검색한다. [컴퓨터인터넷IT용어대사전]

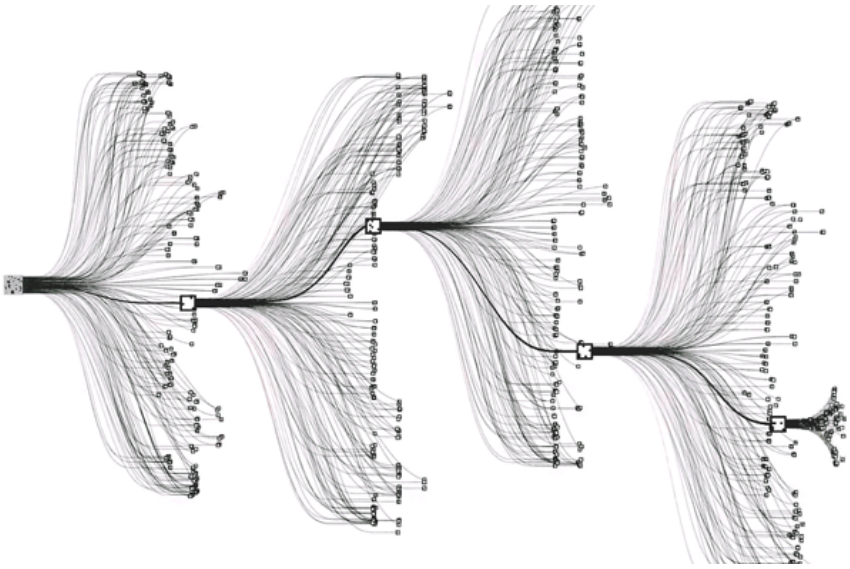
📖 한 노드를 시작으로 **인접한** 모든 정점들을 **우선 방문하는** 방법. 더 이상 방문하지 않은 정점이 없을 때까지 방문하지 않은 모든 정점들에 대해서도 넓이 우선 검색을 적용한다. [컴퓨터인터넷IT용어대사전]

👉 그 전에!
왜 DFS & BFS 를 배울까요?

정렬된 데이터를 이분 탐색하는 것처럼 아주 효율적인 방법이 있는 반면에,
모든 경우의 수를 **전부 탐색해야 하는** 경우도 있습니다.

대표적인 예시가 **알파고**입니다.
대국에서 발생하는 모든 수를 계산하고 예측해서 최적의 수를 계산해내기 위해
모든 수를 전부 탐색해야 합니다.

DFS 와 BFS 는 그 탐색하는 순서에서 차이가 있습니다.
DFS 는 끝까지 파고드는 것이고,
BFS 는 갈라진 모든 경우의 수를 탐색해보고 오는 것이 차이점입니다.



👉 DFS 는 끝까지 파고드는 것이라, 그래프의 최대 깊이 만큼의 공간을 요구합니다.
따라서 공간을 적게 씁니다. 그러나 최단 경로를 탐색하기 쉽지 않습니다.

BFS 는 최단 경로를 쉽게 찾을 수 있습니다! 모든 분기되는 수를 다 보고 올 수 있으니까요.
그러나, 모든 분기되는 수를 다 저장하다보니 공간을 많이 써야하고, 모든 걸 다 보고 오다보니 시간이 더 오래걸릴 수 있습니다.

이런 차이점들은 직접 구현해보면서 느끼러 가보죠!

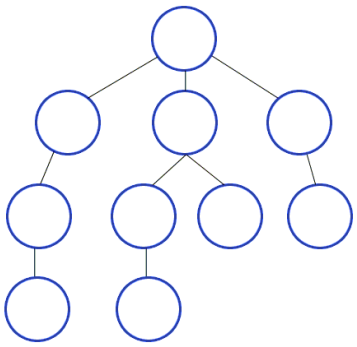
▼ 10) ✍️ DFS 구현해보기 - 재귀함수

👉 DFS는 Depth First Search 라고 했습니다!

갈 수 있는 만큼 **계속해서** 탐색하다가 **갈 수 없게 되면** 다른 방향으로 다시 탐색하는 구조입니다.
이 말만 들어서는 방법이 안 떠오르니까, 한 번 구체적으로 실행 과정을 적어보겠습니다!

- 노드를 방문하고 **깊이 우선으로 인접한** 노드를 방문한다.
- 또 그 노드를 방문해서 **깊이 우선으로 인접한** 노드를 방문한다.
- **만약 끝에** 도달했다면 리턴한다.

혹시 이 굵은 글자를 보면 뭔가 떠오르나요?
반복하다가.. 갈 수 없게 되면.. 탈출 조건..
그렇습니다 바로 재귀함수를 이용해서 구현할 수 있습니다!



👉 DFS 의 반복 방식은 방문하지 않은 원소를 계속해서 찾아가면 됩니다!

즉,
DFS(node) = node + DFS(node와 인접하지만 방문하지 않은 다른 node)
로 반복하면 됩니다.

그런데, 방문하지 않았다는 조건을 과연 어떻게 알 수 있을까요?
다 기록해놔야 알 수 있습니다.
이를 위해서 visited 라는 배열에 방문한 노드를 기록해두면 될 것 같습니다.

자 그러면
1. 루트 노드부터 시작한다.
2. 현재 방문한 노드를 visited 에 추가한다.
3. 현재 방문한 노드와 인접한 노드 중 방문하지 않은 노드에 방문한다.
4. 2부터 반복한다.

예시를 한 번 들어보겠습니다!

```
# 위의 그래프를 예시로 삼아서 인접 리스트 방식으로 표현했습니다!
graph = {
    1: [2, 5, 9],
    2: [1, 3],
    3: [2, 4],
    4: [3],
    5: [1, 6, 8],
    6: [5, 7],
    7: [6],
    8: [5],
    9: [1, 10],
    10: [9]
}
visited = [] # 방문한 걸 저장하기 위한 배열

1. 우선 탐색 시작 노드를 1로 잡겠습니다!

2. 현재 방문한 노드인 1을 visited 에 추가합니다. # visited -> [1]

3. 인접한 노드들인 [2, 5, 9] 에서 방문하지 않은 것들은 [2, 5, 9] 입니다. 2 에 방문합니다.

4. 현재 방문한 노드인 2을 visited 에 추가합니다. # visited -> [1, 2]

5. 인접한 노드들인 [1, 3] 에서 방문하지 않은 것들은 [3] 입니다. 3에 방문합니다.

6. 현재 방문한 노드인 3을 visited 에 추가합니다. # visited -> [1, 2, 3]

7. 인접한 노드들인 [2, 4] 에서 방문하지 않은 것들은 [4] 입니다. 4에 방문합니다.

8. 현재 방문한 노드인 4을 visited 에 추가합니다. # visited -> [1, 2, 3, 4]

9. 인접한 노드들인 [3] 에서 방문하지 않은 것들이 없습니다. 7로 돌아갑니다.

7. 인접한 노드들인 [2, 4] 에서 방문하지 않은 것들이 없습니다. 5로 돌아갑니다.

5. 인접한 노드들인 [1, 3] 에서 방문하지 않은 것들이 없습니다. 3로 돌아갑니다.

3. 인접한 노드들인 [2, 5, 9] 에서 방문하지 않은 것들은 [5, 9] 입니다. 5에 방문합니다.

10. 현재 방문한 노드인 5를 visited 에 추가합니다.  # visited -> [1, 2, 3, 4, 5]

11. 인접한 노드들인 [1, 6, 8] 에서 방문하지 않은 것들은 [6, 8] 입니다. 6에 방문합니다.

12. 현재 방문한 노드인 6를 visited 에 추가합니다.  # visited -> [1, 2, 3, 4, 5, 6]
```

```
13. 인접한 노드들인 [5, 7] 에서 방문하지 않은 것들은 [7] 입니다. 7에 방문합니다.

14. 현재 방문한 노드인 7를 visited 에 추가합니다.  # visited -> [1, 2, 3, 4, 5, 6, 7]

15. 인접한 노드들인 [6] 에서 방문하지 않은 것들이 없습니다. 11로 돌아갑니다.

11. 인접한 노드들인 [1, 6, 8] 에서 방문하지 않은 것들은 [8] 입니다. 8에 방문합니다.

16. 현재 방문한 노드인 8을 visited 에 추가합니다.  # visited -> [1, 2, 3, 4, 5, 6, 7, 8]

17. 인접한 노드들인 [5] 에서 방문하지 않은 것들이 없습니다. 11로 돌아갑니다.

11. 인접한 노드들인 [1, 6, 8] 에서 방문하지 않은 것들이 없습니다. 3으로 돌아갑니다.

3. 인접한 노드들인 [2, 5, 9] 에서 방문하지 않은 것들은 [9] 입니다. 9에 방문합니다.

18. 현재 방문한 노드인 9을 visited 에 추가합니다.  # visited -> [1, 2, 3, 4, 5, 6, 7, 8, 9]

19. 인접한 노드들인 [1, 10] 에서 방문하지 않은 것들은 [10] 입니다. 10에 방문합니다.

20. 현재 방문한 노드인 10을 visited 에 추가합니다.  # visited -> [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]

21. 인접한 노드들인 [9] 에서 방문하지 않은 것들이 없습니다. 19로 돌아갑니다.

19. 인접한 노드들인 [1, 10] 에서 방문하지 않은 것들이 없습니다. 3로 돌아갑니다.

3. 인접한 노드들인 [2, 5, 9] 에서 방문하지 않은 것들이 없습니다. 1로 돌아갑니다.

1. 끝났습니다.


자... 어마어마한 내용들이 있었는데요!
이 코드를 보면 어떤 순서대로 DFS 가 이루어지는 지 조금은 이해가 가실 것 같습니다.
모든 내용을 이해하실 필요는 없습니다 다만! 탐색의 순서와 느낌에 대해 이해가셨으면 좋겠습니다.
```

👉 위의 개념을 한 번 코드로 같이 작성하러 가보시죠!

기본 코드를 제공해드릴테니, 어렵지 않으실거에요! ㅎㅎ

❓ Q. 인접 리스트가 주어질 때, 모든 노드를 DFS 순서대로 방문하시오.

- 이 문제를 풀기 위해서는 어떻게 해야 할까요? 아래 코드를 복사 붙여넣기 하고 함수를 작성해보세요! 2분 정도 고민해 본 다음, 아래 방법들을 펼쳐 봅시다!

▼ **[코드스니펫] DFS 구현해보기 - 재귀함수**

```
# 위의 그래프를 예시로 삼아서 인접 리스트 방식으로 표현했습니다!
graph = {
    1: [2, 5, 9],
    2: [1, 3],
    3: [2, 4],
    4: [3],
    5: [1, 6, 8],
    6: [5, 7],
    7: [6],
    8: [5],
    9: [1, 10],
    10: [9]
}
visited = []


def dfs_recursion(adjacent_graph, cur_node, visited_array):
    # 구현해보세요!
    return
```

```
dfs_recursion(graph, 1, visited)  # 1 이 시작노드입니다!
print(visited)  # [1, 2, 3, 4, 5, 6, 7, 8, 9, 10] 이 출력되어야 합니다!
```

▼ 해결 방법

✅ 아까 생각했던 방법 그대로 코드로 옮기시면 됩니다!

1. 시작 노드인 1부터 탐색합니다!
2. 현재 방문한 노드를 visited_array 에 추가합니다!
3. 현재 방문한 노드와 인접한 노드 중 방문하지 않은 노드에 방문합니다!

현재 방문한 노드와 인접한 노드는 adjacent_graph[cur_node] 로 구할 수 있습니다!
방문하지 않은 걸 확인 하기 위해서는 visited_array 를 이용하시면 됩니다!

```
...
def dfs_recursion(adjacent_graph, cur_node, visited_array):
    visited_array.append(cur_node)
    for adjacent_node in adjacent_graph[cur_node]:
        if adjacent_node not in visited_array:
            dfs_recursion(adjacent_graph, adjacent_node, visited_array)
...
```

▼ 11) ✍️ DFS 구현해보기 - 스택

😢 DFS 를 재귀함수를 통해 잘 구현해보셨나요?

그.러.나.

재귀함수를 통해서는 무한정 깊어지는 노드가 있는 경우 에러가 생길 수 있습니다!

예전에 저희가 탈출조건 없이 반복시켰더니 `RecursionError` 라는
에러를 발생시켰던 것과 같이 DFS 의 깊이가 무한정 깊어지면 에러가 발생할 수 있기 때문입니다!

```
RecursionError: maximum recursion depth exceeded while calling a Python object

Process finished with exit code 1
```

☝️ 다른 방법으로 해결할 순 없을까요?

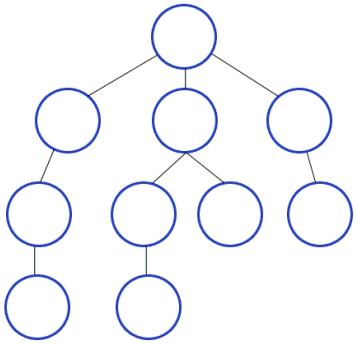
DFS 는 탐색하는 원소를 최대한 깊게 따라가야 합니다.
이걸 다시 말하면 인접한 노드 중 방문하지 않은 모든 노드들을 저장해두고,
가장 마지막에 넣은 노드들만 꺼내서 탐색하면 됩니다.

가장 마지막에 넣은 노드들..? → **스택**을 이용하면 DFS 를 재귀 없이 구현할 수 있습니다!

구현의 방법은 다음과 같습니다.
1. 루트 노드를 스택에 넣습니다.
2. 현재 스택의 노드를 빼서 visited 에 추가한다.
3. 현재 방문한 노드와 인접한 노드 중 방문하지 않은 노드를 스택에 추가한다.
4. 2부터 반복한다.
5. 스택이 비면 탐색을 종료한다.

그런데, 방문하지 않았다는 조건을 과연 어떻게 알 수 있을까요?
다 기록해놔야 알 수 있습니다.
이를 위해서 visited 라는 배열에 방문한 노드를 기록해두면 될 것 같습니다.

예시를 한 번 들어보겠습니다!



```
# 위의 그래프를 예시로 삼아서 인접 리스트 방식으로 표현했습니다!
graph = {
    1: [2, 5, 9],
    2: [1, 3],
    3: [2, 4],
    4: [3],
    5: [1, 6, 8],
    6: [5, 7],
    7: [6],
    8: [5],
    9: [1, 10],
    10: [9]
}
visited = [] # 방문한 걸 저장하기 위한 배열
stack = [1] # 시작 노드인 1을 넣어둔다.


1. 현재 스택에서 가장 마지막에 넣은 1을 빼서, visited 에 추가합니다.
# stack -> [] visited -> [1]

2. 인접한 노드들인 [2, 5, 9] 에서 방문하지 않은 것들인 [2, 5, 9]를 stack 에 추가합니다.
```

```
# stack -> [2, 5, 9] visited -> [1]

3. 현재 스택에서 가장 마지막에 넣은 9을 빼서, visited 에 추가합니다.
# stack -> [2, 5] visited -> [1, 9]

4. 인접한 노드들인 [1, 10] 에서 방문하지 않은 것들인 [10] 을 stack 에 추가합니다.
# stack -> [2, 5, 10] visited -> [1, 9]

5. 현재 스택에서 가장 마지막에 넣은 10을 빼서, visited 에 추가합니다.
# stack -> [2, 5] visited -> [1, 9, 10]

6. 인접한 노드들인 [9] 에서 방문하지 않은 노드들이 없으니 추가하지 않습니다.
# stack -> [2, 5] visited -> [1, 9, 10]

7. 현재 스택에서 가장 마지막에 넣은 5를 빼서, visited 에 추가합니다.
# stack -> [2] visited -> [1, 9, 10, 5]

8. 인접한 노드들인 [1, 6, 8] 에서 방문하지 않은 것들인 [6, 8]를 stack 에 추가합니다.
# stack -> [2, 6, 8] visited -> [1, 9, 10, 5]

9. 현재 스택에서 가장 마지막에 넣은 8를 을 빼서, visited 에 추가합니다.
# stack -> [2, 6] visited -> [1, 9, 10, 5, 8]

10. 인접한 노드들인 [5] 에서 방문하지 않은 노드들이 없으니 추가하지 않습니다.
# stack -> [2, 6] visited -> [1, 9, 10, 5, 8]

11. 현재 스택에서 가장 마지막에 넣은 6을 빼서, visited 에 추가합니다.
# stack -> [2] visited -> [1, 9, 10, 5, 8, 6]

12. 인접한 노드들인 [5, 7] 에서 방문하지 않은 것들인 [7] 을 stack 에 추가합니다.
# stack -> [2, 7] visited -> [1, 9, 10, 5, 8, 6]

13. 현재 스택에서 가장 마지막에 넣은 7를 을 빼서, visited 에 추가합니다.
# stack -> [2] visited -> [1, 9, 10, 5, 8, 6, 7]

14. 인접한 노드들인 [6] 에서 방문하지 않은 노드들이 없으니 추가하지 않습니다.
# stack -> [2] visited -> [1, 9, 10, 5, 8, 6, 7]

15. 현재 스택에서 가장 마지막에 넣은 2를 빼서, visited 에 추가합니다.
# stack -> [] visited -> [1, 9, 10, 5, 8, 6, 7, 2]

16. 인접한 노드들인 [1, 3] 에서 방문하지 않은 것들인 [3] 을 stack 에 추가합니다.
# stack -> [3] visited -> [1, 9, 10, 5, 8, 6, 7, 2]

17. 현재 스택에서 가장 마지막에 넣은 3을 빼서, visited 에 추가합니다.
# stack -> [] visited -> [1, 9, 10, 5, 8, 6, 7, 2, 3]

18. 인접한 노드들인 [2, 4] 에서 방문하지 않은 것들인 [4] 를 stack 에 추가합니다.
# stack -> [4] visited -> [1, 9, 10, 5, 8, 6, 7, 2, 3, 4]

19. 현재 스택에서 가장 마지막에 넣은 4를 빼서, visited 에 추가합니다.
# stack -> [] visited -> [1, 9, 10, 5, 8, 6, 7, 2, 3, 4]

20. 인접한 노드들인 [3] 에서 방문하지 않은 노드들이 없으니 추가하지 않습니다.

21. 현재 스택에서 꺼낼 것이 없습니다. DFS 가 끝났습니다.



여기서 조금 주의해야 할 점은, 아까 재귀로 구현한 방식과는 탐색하는 순서가 조금 다릅니다!

그러나, 가장 깊게 탐색하는 방식은 똑같습니다!

구현의 차이일뿐 개념은 동일하다는 점 이해하셨으면 좋겠습니다.

이 코드를 보면 어떤 순서대로 DFS 가 이루어지는 지 조금은 이해가 가실 것 같습니다.
```

👉 위의 개념을 한 번 코드로 같이 작성하러 가보시죠!

기본 코드를 제공해드릴테니, 어렵지 않으실거에요! ㅎㅎ

❓ Q. 인접 리스트가 주어질 때, 모든 노드를 DFS 순서대로 방문하시오.

- 이 문제를 풀기 위해서는 어떻게 해야 할까요? 아래 코드를 복사 붙여넣기 하고 함수를 작성해보세요! 2분 정도 고민해 본 다음, 아래 방법들을 펼쳐 봅시다!

▼ **[코드스니펫] DFS 구현해보기 - 스택**

```
# 위의 그래프를 예시로 삼아서 인접 리스트 방식으로 표현했습니다!
graph = {
    1: [2, 5, 9],
    2: [1, 3],
    3: [2, 4],
    4: [3],
    5: [1, 6, 8],
    6: [5, 7],
    7: [6],
    8: [5],
    9: [1, 10],
    10: [9]
}


def dfs_stack(adjacent_graph, start_node):
    # 구현해보세요!
    return


print(dfs_stack(graph, 1))  # 1 이 시작노드입니다!
# [1, 9, 10, 5, 8, 6, 7, 2, 3, 4] 이 출력되어야 합니다!
```

▼ 해결 방법

✅ 아까 생각했던 방법 그대로 코드로 옮기시면 됩니다!

1. 시작 노드를 스택에 넣습니다.
2. 현재 스택의 노드를 빼서 visited 에 추가한다.
3. 현재 방문한 노드와 인접한 노드 중 방문하지 않은 노드를 스택에 추가한다.

이 과정을 스택이 빌때까지 반복하면 됩니다!

현재 방문한 노드와 인접한 노드는 adjacent_node[current_node] 로 구할 수 있습니다!
방문하지 않은 걸 확인 하기 위해서는 visited 를 이용하시면 됩니다!

```
...
def dfs_stack(adjacent_graph, start_node):
    stack = [start_node]
    visited = []
    while stack:
        current_node = stack.pop()
        visited.append(current_node)
        for adjacent_node in adjacent_graph[current_node]:
            if adjacent_node not in visited:
                stack.append(adjacent_node)
    return visited
...
```

07. BFS

▼ 12) ✍️ BFS 구현해보기

👉 자 이번에는, BFS 를 구현해보도록 하겠습니다!

우선 DFS 와 BFS 의 차이점을 다시 곱씹어볼게요!

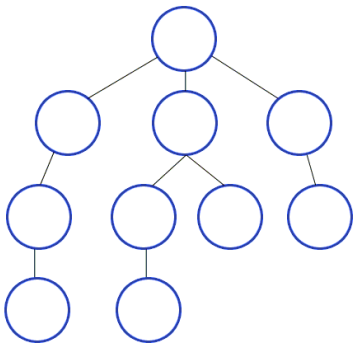
DFS 는 탐색하는 원소를 최대한 깊게 따라가야 합니다.
이를 구현하기 위해 인접한 노드 중 방문하지 않은 모든 노드들을 저장해두고,
가장 마지막에 넣은 노드를 꺼내서 탐색하면 됩니다. → 그래서 스택을 썼죠!

BFS 는 현재 인접한 노드 먼저 방문해야 합니다.
이걸 다시 말하면 인접한 노드 중 방문하지 않은 모든 노드들을 저장해두고,
가장 처음에 넣은 노드를 꺼내서 탐색하면 됩니다.

가장 처음에 넣은 노드들..? → **큐**를 이용하면 BFS 를 구현할 수 있습니다!

구현의 방법은 다음과 같습니다.
1. 루트 노드를 큐에 넣습니다.
2. 현재 큐의 노드를 빼서 visited 에 추가한다.
3. 현재 방문한 노드와 인접한 노드 중 방문하지 않은 노드를 큐에 추가한다.
4. 2부터 반복한다.
5. 큐가딕 비면 탐색을 종료한다.

예시를 한 번 들어보겠습니다!



```
# 위의 그래프를 예시로 삼아서 인접 리스트 방식으로 표현했습니다!
graph = {
    1: [2, 3, 4],
    2: [1, 5],
    3: [1, 6, 7],
    4: [1, 8],
    5: [2, 9],
    6: [3, 10],
    7: [3],
    8: [4],
    9: [5],
    10: [6]
}
```

```
visited = [] # 방문한 걸 저장하기 위한 배열
queue = [1] # 시작 노드인 1을 넣어둔다.

1. 현재 큐에서 가장 처음에 넣은 1을 빼서, visited 에 추가합니다.
# queue -> [] visited -> [1]

2. 인접한 노드들인 [2, 3, 4] 에서 방문하지 않은 것들인 [2, 3, 4]를 queue 에 추가합니다.
# queue -> [2, 3, 4] visited -> [1]

3. 현재 큐에서 가장 처음에 넣은 2을 빼서, visited 에 추가합니다.
# queue -> [3, 4] visited -> [1, 2]

4. 인접한 노드들인 [1, 5] 에서 방문하지 않은 것들인 [5]를 queue 에 추가합니다.
# queue -> [3, 4, 5] visited -> [1, 2]

5. 현재 큐에서 가장 처음에 넣은 3을 빼서, visited 에 추가합니다.
# queue -> [4, 5] visited -> [1, 2, 3]

6. 인접한 노드들인 [1, 6, 7] 에서 방문하지 않은 것들인 [6, 7]를 queue 에 추가합니다.
# queue -> [4, 5, 6, 7] visited -> [1, 2, 3]

7. 현재 큐에서 가장 처음에 넣은 4을 빼서, visited 에 추가합니다.
# queue -> [5, 6, 7] visited -> [1, 2, 3, 4]

8. 인접한 노드들인 [1, 8] 에서 방문하지 않은 것들인 [8]를 queue 에 추가합니다.
# queue -> [5, 6, 7, 8] visited -> [1, 2, 3, 4]

9. 현재 큐에서 가장 처음에 넣은 5을 빼서, visited 에 추가합니다.
# queue -> [6, 7, 8] visited -> [1, 2, 3, 4, 5]

10. 인접한 노드들인 [2, 9] 에서 방문하지 않은 것들인 [9]를 queue 에 추가합니다.
# queue -> [6, 7, 8, 9] visited -> [1, 2, 3, 4, 5]

11. 현재 큐에서 가장 처음에 넣은 6을 빼서, visited 에 추가합니다.
# queue -> [7, 8, 9] visited -> [1, 2, 3, 4, 5, 6]

12. 인접한 노드들인 [3, 10] 에서 방문하지 않은 것들인 [10]를 queue 에 추가합니다.
# queue -> [7, 8, 9, 10] visited -> [1, 2, 3, 4, 5, 6]

13. 현재 큐에서 가장 처음에 넣은 7을 빼서, visited 에 추가합니다.
# queue -> [8, 9, 10] visited -> [1, 2, 3, 4, 5, 6, 7]

14. 인접한 노드들인 [3] 에서 방문하지 않은 것들이 없으니 추가하지 않습니다.
# queue -> [8, 9, 10] visited -> [1, 2, 3, 4, 5, 6, 7]

15. 현재 큐에서 가장 처음에 넣은 8을 빼서, visited 에 추가합니다.
# queue -> [9, 10] visited -> [1, 2, 3, 4, 5, 6, 7, 8]

16. 인접한 노드들인 [4] 에서 방문하지 않은 것들이 없으니 추가하지 않습니다.
# queue -> [9, 10] visited -> [1, 2, 3, 4, 5, 6, 7, 8]

17. 현재 큐에서 가장 처음에 넣은 9을 빼서, visited 에 추가합니다.
# queue -> [10] visited -> [1, 2, 3, 4, 5, 6, 7, 8, 9]

18. 인접한 노드들인 [5] 에서 방문하지 않은 것들이 없으니 추가하지 않습니다.
# queue -> [10] visited -> [1, 2, 3, 4, 5, 6, 7, 8, 9]

19. 현재 큐에서 가장 처음에 넣은 10을 빼서, visited 에 추가합니다.
# queue -> [] visited -> [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]

20. 인접한 노드들인 [6] 에서 방문하지 않은 것들이 없으니 추가하지 않습니다.
# queue -> [] visited -> [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]

21. 현재 큐에서 꺼낼 것이 없습니다. BFS 가 끝났습니다.


자... 어마어마한 내용들이 있었는데요!
이 코드를 보면 어떤 순서대로 BFS 가 이루어지는 지 조금은 이해가 가실 것 같습니다.
모든 순서를 외우실 필요는 없습니다 다만! 탐색의 순서와 느낌에 대해 이해가셨으면 좋겠습니다.
```

👉 위의 개념을 한 번 코드로 같이 작성하러 가보시죠!

기본 코드를 제공해드릴테니, 어렵지 않으실거에요! ㅎㅎ

❓ Q. 인접 리스트가 주어질 때, 모든 노드를 BFS 순서대로 방문하시오.

- 이 문제를 풀기 위해서는 어떻게 해야 할까요? 아래 코드를 복사 붙여넣기 하고 함수를 작성해보세요! 2분 정도 고민해 본 다음, 아래 방법들을 펼쳐 봅시다!

▼ **[코드스니펫] BFS 구현해보기**

```
# 위의 그래프를 예시로 삼아서 인접 리스트 방식으로 표현했습니다!
graph = {
    1: [2, 3, 4],
    2: [1, 5],
    3: [1, 6, 7],
    4: [1, 8],
    5: [2, 9],
    6: [3, 10],
    7: [3],
    8: [4],
    9: [5],
    10: [6]
}


def bfs_queue(adj_graph, start_node):
    # 구현해보세요!
    return


print(bfs_queue(graph, 1))  # 1 이 시작노드입니다!
# [1, 2, 3, 4, 5, 6, 7, 8, 9, 10] 이 출력되어야 합니다!
```

▼ 해결 방법

✅ 아까 생각했던 방법 그대로 코드로 옮기시면 됩니다!

1. 시작 노드를 큐에 넣습니다.
2. 현재 큐의 노드를 빼서 visited 에 추가합니다.
3. 현재 방문한 노드와 인접한 노드 중 방문하지 않은 노드를 큐에 추가한다.

이 과정을 큐가 빌때까지 반복하면 됩니다!

현재 방문한 노드와 인접한 노드는 adjacent_node[current_node] 로 구할 수 있습니다!
방문하지 않은 걸 확인 하기 위해서는 visited 를 이용하시면 됩니다!

```
...
def bfs_queue(adj_graph, start_node):
    queue = [start_node]
    visited = []

    while queue:
        current_node = queue.pop(0)
        visited.append(current_node)
        for adjacent_node in adj_graph[current_node]:
```

```
            if adjacent_node not in visited:
                queue.append(adjacent_node)

    return visited
...
```

08. Dynamic Programming

▼ 13) ✍️ 피보나치 수열 - 재귀함수

👉 동적 계획법에 대해 설명 드리기 전에,
피보나치 문제를 풀어보면서 필요성을 느끼는 게 가장 좋을 것 같습니다!

그런데, 피보나치 수열이 뭘까요?

📖 수학에서, 피보나치 수(영어: Fibonacci numbers)는 첫째 및 둘째 항이 1이며 그 뒤의 모든 항은 바로 앞 두 항의 합인 수열이다. 처음 여섯 항은 각각 1, 1, 2, 3, 5, 8이다. [위키백과]

👉 즉, n번째 피보나치 수를 Fibo(n)라고 표현한다면,
Fibo(1) 은 1이고,
Fibo(2) 도 1입니다! (첫째 및 둘째 항을 1이라고 정했대요!)

Fibo(3) 부터는 이전 값과 이전 이전 값의 합입니다!
즉, Fibo(3) = Fibo(1) + Fibo(2) = 1 + 1 = 2 입니다.

그러면
Fibo(4) = Fibo(3) + Fibo(2) = 2 + 1 = 3
Fibo(5) = Fibo(4) + Fibo(3) = 3 + 2 = 5
Fibo(6) = Fibo(5) + Fibo(4) = 5 + 3 = 8 .....
Fibo(n) = Fibo(n - 1) + Fibo(n-2) 라고 표현할 수 있습니다!

어 그런데 비슷한 구조의 반복.. 뭐가 떠오르죠?

맞습니다! 바로 재귀함수로 풀 수 있습니다.

ㅎㅎ 한 번 구현 해보러 가시죠

❓ Q. 피보나치 수열의 20번째 수를 구하시오.

- 이 문제를 풀기 위해서는 어떻게 해야 할까요? 아래 코드를 복사 붙여넣기 하고 함수를 작성해보세요! 2분 정도 고민해 본 다음, 아래 방법들을 펼쳐 봅시다!

▼ **[코드스니펫] 피보나치 수열 - 재귀함수**

```
input = 20


def fibo_recursion(n):
    # 구현해보세요!
    return


print(fibo_recursion(input))  # 6765
```

▼ 해결 방법

✅ Fibo(n) = Fibo(n - 1) + Fibo(n-2) 라는 수식을 그대로 표현해주시면 됩니다!

그리고 피보나치의 탈출 조건은?
n 이 1이거나 2일 때 1을 반환해주시면 됩니다.

```
input = 20


def fibo_recursion(n):
    if n == 1 or n == 2:
        return 1
    return fibo_recursion(n - 1) + fibo_recursion(n - 2)


print(fibo_recursion(input))  # 6765
```

▼ 단점

👉 방금 만들어본 함수에서 input 을 한 번 100으로 바꿔보세요!

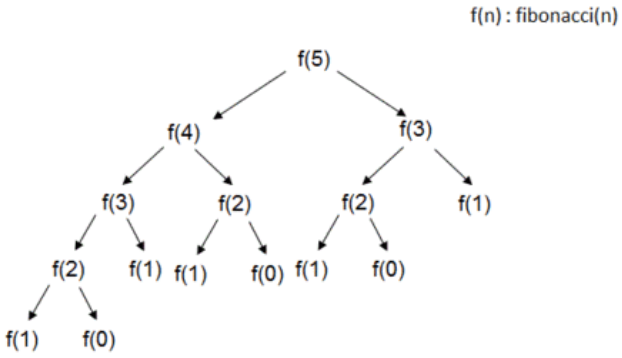
그러면 실행했을 때 연산이 너~~무 오래걸려서 값이 나오질 않습니다.

엄청 오래 걸리는 이유가 뭘까요?

✅ 한 번, 아까 만든 함수를 다시 생각해보겠습니다.

```
1. Fibo(3) 을 구한다고 치면,
Fibo(2) 와 Fibo(1) 을 더하면 됩니다.
이는 1, 1 이기 때문에 빠르게 반환됩니다.
-> 연산량 2번

2. Fibo(4) 를 구한다고 치면,
Fibo(3) 과 Fibo(2) 을 더하면 됩니다.
Fibo(3) 은? 1의 과정을 반복해야 합니다.
Fibo(2) 는 1이므로 빠르게 반환됩니다.
-> Fibo(3) 을 구하고 1 을 더합니다.
-> Fibo(3) 을 구하기 위해서는
  -> Fibo(2) 와 Fibo(1) 을 더하면 됩니다.
  -> 이는 1, 1 이기 때문에 빠르게 반환됩니다.
  -> 연산량 2번
-> 1.의 연산량 + 연산량 1번 = 연산량 3번

3. Fibo(5) 를 구한다고 치면,
Fibo(4) 과 Fibo(3) 을 더하면 됩니다.
```

```
Fibo(4) 은? 2.의 과정을 반복해야 합니다.
Fibo(3) 은? 1.의 과정을 반복해야 합니다.
-> Fibo(4) 을 구하고 Fibo(3)을 구합니다.
  -> Fibo(4) 을 구하기 위해서는               # Fibo(4) 를 구하기 위해서 Fibo(3)을 찾고
    -> Fibo(3) 을 구하고 1을 더해야 합니다.
    -> Fibo(3) 을 구하기 위해서는
      -> Fibo(2) 와 Fibo(1) 을 더하면 됩니다.
      -> 이는 1, 1 이기 때문에 빠르게 반환됩니다.
      -> 연산량 2번
  -> Fibo(3) 을 구하기 위해서는               # Fibo(5)를 구하기 위해 Fibo(3)을 찾습니다.
    -> Fibo(2) 와 Fibo(1)을 더하면 됩니다.
    -> Fibo(2) 와 Fibo(1) 을 더하면 됩니다.
    -> 이는 1, 1 이기 때문에 빠르게 반환됩니다.
    -> 연산량 2번
-> 2.의 연산량 + 1.의 연산량 = 연산량 5번

즉, 아래와 같은 그림처럼 했던 작업을 계속 다시 하게 됩니다.
이미 시켰던 똑같은 작업을 다른 곳에서 다시 새롭게 하고 있는겁니다.
이런 "삽질" 을 계속 시키지 않으려면, 이미 했던 일을 기록하고 있어야 합니다.
그 방법에 대해서 동적 계획법에서 배워 봅시다.
```



Recursion tree generated for computing 5th number of fibonacci sequence

▼ 14) 동적 계획법(Dynamic Programming)이란?

👉 르탄이는 매일 회사로 출근을 합니다.
그래서 출근하는 방법을 어떻게해야 가장 효율적인지를 알고 싶습니다.

집 - 봉천역 - 삼성역 - 코엑스 까지 걸어가는 길인데,

각각의 목적지까지 이동하는 방법은
지하철, 버스, 따릉이, 공유 킥보드가 있습니다.

그래서 다음과 같이 매일 실험해봤습니다.

1일 : 지하철(15분) - 지하철(20분) - 지하철(3분)
2일 : 지하철(15분) - 지하철(20분) - 버스(5분)
3일 : 지하철(15분) - 지하철(20분) - 따릉이(4분)
4일 : 지하철(15분) - 지하철(20분) - 공유 킥보드(6분)
5일 : 지하철(15분) - 버스(10분) - 지하철(3분)
6일 : 지하철(15분) - 버스(10분) - 버스(5분) .....

그런데, 꼭 이렇게 모든 경우의 수를 다 해봐야 할까요?

각 구간마다 수단이 얼마나 걸리는지만 알면 되지 않을까요?
또한, 이미 실험했던 시간은 다시 시도하지 않고 기록해두면 되지 않을까요?

이런 문제를 해결하기 위해서 동적 계획법(Dynamic Programming)이 탄생했습니다!

📖 동적 계획법(Dynamic Programming)이란 복잡한 문제를 간단한 여러 개의 문제로 나누어 푸는 방법을 말한다. 이것은 부분 문제 반복과 최적 부분 구조를 가지고 있는 알고리즘을 일반적인 방법에 비해 더욱 적은 시간 내에 풀 때 사용한다. [위키백과]

👉 동적 계획법은
여러 개의 하위 문제를 풀고 그 결과를 기록하고 이용해 문제를 해결하는 알고리즘입니다!

즉, 우리가 재귀함수를 풀어나갈 때 많이 했던 함수의 수식화를 시키면 됩니다.
`F(string) = F(string[1:n-2])` 라고 수식을 정의했던 것 처럼,
문제를 쪼개서 정의할 수 있으면 동적 계획법을 쓸 수 있습니다!

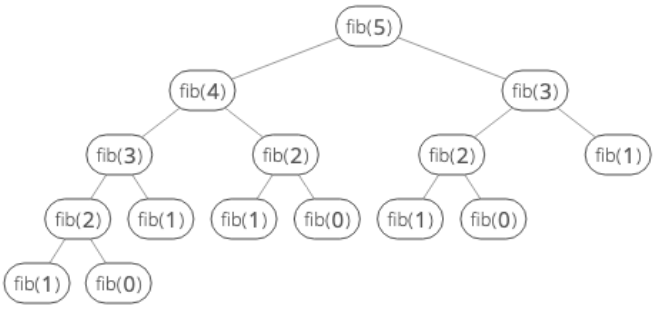
이처럼 문제를 반복해서 해결해 나가는 모습이 재귀 알고리즘과 닮아있습니다!
그러나 다른 점은, **그 결과를 기록하고 이용한다**는 점입니다!

여기서 용어정리 간단하게 해드리겠습니다!

결과를 기록하는 것을 **메모이제이션(Memoization)** 이라고 하고,
문제를 쪼갤 수 있는 구조를 **겹치는 부분 문제(Overlapping Subproblem)**라고 합니다!

아까 예시에서 설명 드렸던 부분에서
각 구간마다의 시간을 계산하면 최적의 시간을 구할 수 있는 것을 **겹치는 부분 문제,**
이미 실험했던 내용은 기록해두고 쓰면 된다는 것을 **메모이제이션** 이라고 생각하시면 됩니다!

즉, 겹치는 부분 문제일 경우 동적 계획법을 사용하면 되는데,
이 때 사용하는 방법은 메모이제이션을 이용하는구나! 라고 생각하시면 됩니다.



▼ 15) ✍️ 피보나치 수열 - 동적 계획법

😥 재귀함수로 피보나치 수열을 구현했을 때,
계속 똑같은 문제를 다시 푸는 바람에 목표를 달성하지 못했습니다!

이번에는, 새로 배운 동적 계획법의 **메모이제이션**이라는 방법을 통해서 해결해보도록 하겠습니다!

구현의 방법은 다음과 같습니다.
1. 메모용 데이터를 만든다. 처음 값인 Fibo(1), Fibo(2) 는 각각 1씩 넣어서 저장해둔다.
2. Fibo(n) 을 구할 때 만약 메모에 그 값이 있다면 바로 반환한다.
3. Fibo(n) 을 처음 구했다면 메모에 그 값을 기록한다.

예시를 한 번 들어보겠습니다.

```
memo = {
    1: 1, # Fibo(1) 의 값은 1이라고 기록해둔다.
    2: 1  # Fibo(2) 의 값은 1이라고 기록한다.
}

fibo(3) 을 찾아와라!

1. memo[3]이 있는지 본다.
2. 없으니까 fibo(2) + fibo(1) 을 구해야 한다.
3. 그러면 memo[2] 와 memo[1] 이 있는지 찾아본다.
4. 있으니까 그 값을 가져온 뒤 더해서 fibo(3) 을 만든다.
5. memo[3] 에 fibo(3) 을 기록한다.

fibo(4) 을 찾아와라!
1. memo[4]이 있는지 본다.
2. 없으니까 fibo(3) + fibo(2) 을 구해야 한다.
3. 그러면 memo[3] 와 memo[2] 이 있는지 찾아본다.
  4. memo[3] 이 없으니까 fibo(2) + fibo(1) 을 구해야 한다.
  5. 그러면 memo[2] 와 memo[1] 이 있는지 찾아본다.
  6. 있으니까 그 값을 가져온 뒤 더해서 fibo(3) 을 만든다.
  7. memo[3] 에 fibo(3) 을 기록한다.
8. memo[3] memo[2] 를 더해 fibo(4) 를 만들고 memo[4] 에 기록한다.

fibo(5) 을 찾아와라!
1. memo[5]이 있는지 본다.
2. 없으니까 fibo(4) + fibo(3) 을 구해야 한다.
3. 그러면 memo[4] 와 memo[3] 이 있는지 찾아본다.
  4. memo[4]가 없으니까 fibo(3) + fibo(2) 을 구해야 한다.
  5. 그러면 memo[3] 와 memo[2] 이 있는지 찾아본다.
    6. memo[3] 이 없으니까 fibo(2) + fibo(1) 을 구해야 한다.
    7. 그러면 memo[2] 와 memo[1] 이 있는지 찾아본다.
    8. 있으니까 그 값을 가져온 뒤 더해서 fibo(3) 을 만든다.
    9. memo[3] 에 fibo(3) 을 기록한다.
  10. memo[3] memo[2] 를 더해 fibo(4) 를 만들고 memo[4] 에 기록한다.
```

```
11. memo[3] 이 있으니까 그 값을 가져온다. (9에서 기록해놨다!!!!)
12. memo[4] 와 memo[3]을 더해 fibo(5) 를 만들고 memo[5] 에 기록한다.


이렇게 기록한 정보를 이용해서 부분 문제를 해결할 수 있었습니다!

어떻게 메모이제이션이 효율적으로 이용될 수 있는지 이해가 되셨길 바랍니다.
```

👉 위의 개념을 한 번 코드로 같이 작성하러 가보시죠!

기본 코드를 제공해드릴테니, 어렵지 않으실거에요! ㅎㅎ

❓ Q. 피보나치 수열의 100번째 수를 구하시오.

- 이 문제를 풀기 위해서는 어떻게 해야 할까요? 아래 코드를 복사 붙여넣기 하고 함수를 작성해보세요! 2분 정도 고민해 본 다음, 아래 방법들을 펼쳐 봅시다!

▼ [코드스니펫] 피보나치 수열 - 동적 계획법

```
input = 50

# memo 라는 변수에 Fibo(1)과 Fibo(2) 값을 저장해놨습니다!
memo = {
    1: 1,
    2: 1
}


def fibo_dynamic_programming(n, fibo_memo):
    # 구현해보세요!
    return


print(fibo_dynamic_programming(input, memo))
```

▼ 해결 방법

✅ 아까 생각했던 방법 그대로 코드로 옮기시면 됩니다!

1. 만약 메모에 그 값이 있다면 바로 반환한다.
2. 피보나치 값을 처음 구했다면 메모에 그 값을 기록한다.

메모는 fibo_memo 를 사용하면 됩니다!

아까 재귀 방식으로는 피보나치 100번째 숫자를 구하지 못했는데,
이번에는 엄청나게 빠르게 반환하는 걸 보실 수 있습니다!

이처럼, 다이나믹 프로그래밍은 성능적 향상 뿐만 아니라
부분 문제를 해결함으로써 전체 문제를 해결할 수 있게 됩니다!

```
input = 50

# memo 라는 변수에 Fibo(1)과 Fibo(2) 값을 저장해놨습니다!
```

```
memo = {
    1: 1,
    2: 1
}


def fibo_dynamic_programming(n, fibo_memo):
    if n in fibo_memo:
        return fibo_memo[n]

    nth_fibo = fibo_dynamic_programming(n - 1, fibo_memo) + fibo_dynamic_programming(n - 2, fibo_memo)
    fibo_memo[n] = nth_fibo
    return nth_fibo


print(fibo_dynamic_programming(input, memo))
```

9. 끝 & 숙제 설명

문제 직접 풀어보기!

▼ Q1. ✍️ 농심 라면 공장

❓ Q. 라면 공장에서는 하루에 밀가루를 1톤씩 사용합니다. 원래 밀가루를 공급받던 공장의 고장으로 앞으로 k일 이후에야 밀가루를 공급받을 수 있기 때문에 해외 공장에서 밀가루를 수입해야 합니다.

해외 공장에서는 향후 밀가루를 공급할 수 있는 날짜와 수량을 알려주었고, 라면 공장에서는 운송비를 줄이기 위해 최소한의 횟수로 밀가루를 공급받고 싶습니다.

현재 공장에 남아있는 밀가루 수량 stock, 밀가루 공급 일정(dates)과 해당 시점에 공급 가능한 밀가루 수량(supplies), 원래 공장으로부터 공급받을 수 있는 시점 k가 주어질 때, 밀가루가 떨어지지 않고 공장을 운영하기 위해서 최소한 몇 번 해외 공장으로부터 밀가루를 공급받아야 하는지를 반환 하시오.

dates[i]에는 i번째 공급 가능일이 들어있으며, supplies[i]에는 dates[i] 날짜에 공급 가능한 밀가루 수량이 들어있습니다.

```
stock = 4
dates = [4, 10, 15]
supplies = [20, 5, 10]
k = 30

# 다음과 같이 입력값이 들어온다면,
# 현재 재고가 4개 있습니다. 그리고 정상적으로 돌아오는 날은 30일까지입니다.
# 즉, 26개의 공급량을 사와야 합니다!
# 그러면 제일 최소한으로 26개를 가져오려면? supplies 에서 20, 10 을 가져오면 되겠죠?
# 그래서 이 경우의 최소 공급 횟수는 2 입니다!
```

- 이 문제를 풀기 위해서는 어떻게 해야 할까요? 아래 코드를 복사 붙여넣기 하고 함수를 작성해보세요! 2분 정도 고민해 본 다음, 아래 방법들을 펼쳐 봅시다!

▼ 👏 여기서 잠깐! Tip

- 최댓값, 최솟값 뽑는 방법

👉 우리는 `Heap` 이라는 자료 구조를 배웠습니다.
원할 때 언제든지 최솟값과 최댓값을 뽑을 수 있게 만든 자료구조였습니다!

그러면, 항상 힙 자료구조를 써줘야지 쓸 수 있을까요?
그렇지 않습니다! 미리 만들어놓은 라이브러리가 있습니다.
바로 `heapq` 모듈을 이용하면 됩니다.

우리가 만들었던 힙 구조를 유지하면서
원소를 추가하고 삭제하는 기능들을 다 구현해놨습니다! 다음 처럼 하시면 됩니다.

이렇게 heapq 를 이용해서 최솟값을 쉽게 구할 수 있습니다.

```
>>> import heapq # heapq 를 사용하기 위해서는 맨 위에 heapq 를 가져온다! 라는 구문을 써주셔야 합니다.

>>> heap = []
>>> heapq.heappush(heap, 4)
>>> heapq.heappush(heap, 1)
>>> heapq.heappush(heap, 7)
>>> heapq.heappush(heap, 3)
>>> print(heap)
[1, 3, 7, 4] # 힙의 형태를 유지한채로 원소가 넣어지기 때문에 heap[0]이 최솟값입니다!

>>> print(heapq.heappop(heap)) # 최솟값을 빼는 방법입니다.
1
>>> print(heap)
[3, 7, 4] # 마찬가지로 힙의 형태가 유지됩니다.
```

✅ 앗, 그런데 최댓값을 넣고 빼고 싶다면 어떻게 해야 할까요?

음수를 이용하면 됩니다.

-1 을 곱해서 넣고, 뺄 때도 -1을 곱한다면?
최솟값을 기준으로 정렬한다는 점을 이용해서 최댓값을 저장할 수 있습니다!

```
>>> import heapq # heapq 를 사용하기 위해서는 맨 위에 heapq 를 가져온다! 라는 구문을 써주셔야 합니다.

>>> heap = []
>>> heapq.heappush(heap, 4 * -1)
>>> heapq.heappush(heap, 1 * -1)
>>> heapq.heappush(heap, 7 * -1)
>>> heapq.heappush(heap, 3 * -1)
>>> print(heap)
[-7, -3, -4, -1] # 힙의 형태를 유지한채로 원소가 넣어지기 때문에 heap[0]이 최솟값입니다!

>>> print(heapq.heappop(heap) * - 1) # 최댓값을 빼는 방법입니다.
7
>>> print(heap)
[-4, -3, -1] # 마찬가지로 힙의 형태가 유지됩니다.
```

▼ [코드스니펫] 농심 라면 공장

```
import heapq
```

```
ramen_stock = 4
supply_dates = [4, 10, 15]
supply_supplies = [20, 5, 10]
supply_recover_k = 30


def get_minimum_count_of_overseas_supply(stock, dates, supplies, k):
    # 풀어보세요!
    return


print(get_minimum_count_of_overseas_supply(ramen_stock, supply_dates, supply_supplies, supply_recover_k))
```

▼ Q2. ✍️ 샤오미 로봇 청소기

❓ Q.

문제 설명

로봇 청소기가 주어졌을 때, 청소하는 영역의 개수를 구하는 프로그램을 작성하시오.

로봇 청소기가 있는 장소는 N×M 크기의 직사각형으로 나타낼 수 있으며, 1×1크기의 정사각형 칸으로 나누어져 있다. 각각의 칸은 벽 또는 빈 칸이다. 청소기는 바라보는 방향이 있으며, 이 방향은 동, 서, 남, 북중 하나이다. 지도의 각 칸은 (r, c)로 나타낼 수 있고, r은 북쪽으로부터 떨어진 칸의 개수, c는 서쪽으로 부터 떨어진 칸의 개수이다.

로봇 청소기는 다음과 같이 작동한다.

1. 현재 위치를 청소한다.
2. 현재 위치에서 현재 방향을 기준으로 왼쪽방향부터 차례대로 탐색을 진행한다.
   a. 왼쪽 방향에 아직 청소하지 않은 공간이 존재한다면, 그 방향으로 회전한 다음 한 칸을 전진하고 1번부터 진행한다.
   b. 왼쪽 방향에 청소할 공간이 없다면, 그 방향으로 회전하고 2번으로 돌아간다.
   c. 네 방향 모두 청소가 이미 되어있거나 벽인 경우에는, 바라보는 방향을 유지한 채로 한 칸 후진을 하고 2번으로 돌아간다.
   d. 네 방향 모두 청소가 이미 되어있거나 벽이면서, 뒤쪽 방향이 벽이라 후진도 할 수 없는 경우에는 작동을 멈춘다.

로봇 청소기는 이미 청소되어있는 칸을 또 청소하지 않으며, 벽을 통과할 수 없다.

입력 조건

로봇 청소기가 있는 칸의 좌표 (r, c)와 바라보는 방향 d가 주어진다. 이 때 d가 0인 경우에는 북쪽을, 1인 경우에는 동쪽을, 2인 경우에는 남쪽을, 3인 경우에는 서쪽을 바라보고 있는 것이다.

또한 청소하고자 하는 방의 지도를 2차원 배열로 주어진다.
빈 칸은 0, 벽은 1로 주어진다. 지도의 첫 행, 마지막 행, 첫 열, 마지막 열에 있는 모든 칸은 벽이다.

로봇 청소기가 있는 칸의 상태는 항상 빈 칸이라고 했을 때,
로봇 청소기가 청소하는 칸의 개수를 반환하시오.

```
r, c, d = 7, 4, 0
room_map = [
    [1, 1, 1, 1, 1, 1, 1, 1, 1, 1],
    [1, 0, 0, 0, 0, 0, 0, 0, 0, 1],
    [1, 0, 0, 0, 1, 1, 1, 1, 0, 1],
    [1, 0, 0, 1, 1, 0, 0, 0, 0, 1],
    [1, 0, 1, 1, 0, 0, 0, 0, 0, 1],
    [1, 0, 0, 0, 0, 0, 0, 0, 0, 1],
    [1, 0, 0, 0, 0, 0, 0, 1, 0, 1],
```

```
    [1, 0, 0, 0, 0, 0, 1, 1, 0, 1],
    [1, 0, 0, 0, 0, 0, 1, 1, 0, 1],
    [1, 0, 0, 0, 0, 0, 0, 0, 0, 1],
    [1, 1, 1, 1, 1, 1, 1, 1, 1, 1]
]
```

- 이 문제를 풀기 위해서는 어떻게 해야 할까요? 아래 코드를 복사 붙여넣기 하고 함수를 작성해보세요! 2분 정도 고민해 본 다음, 아래 방법들을 펼쳐 봅시다!

▼ **[코드스니펫] 샤오미 로봇 청소기**

```
current_r, current_c, current_d = 7, 4, 0
current_room_map = [
    [1, 1, 1, 1, 1, 1, 1, 1, 1, 1],
    [1, 0, 0, 0, 0, 0, 0, 0, 0, 1],
    [1, 0, 0, 0, 1, 1, 1, 1, 0, 1],
    [1, 0, 0, 1, 1, 0, 0, 0, 0, 1],
    [1, 0, 1, 1, 0, 0, 0, 0, 0, 1],
    [1, 0, 0, 0, 0, 0, 0, 0, 0, 1],
    [1, 0, 0, 0, 0, 0, 0, 1, 0, 1],
    [1, 0, 0, 0, 0, 0, 1, 1, 0, 1],
    [1, 0, 0, 0, 0, 0, 0, 1, 1, 0, 1],
    [1, 0, 0, 0, 0, 0, 0, 0, 0, 1],
    [1, 1, 1, 1, 1, 1, 1, 1, 1, 1]
]


def get_count_of_departments_cleaned_by_robot_vacuum(r, c, d, room_map):
    return


# 52 가 출력되어야 합니다!
print(get_count_of_departments_cleaned_by_robot_vacuum(current_r, current_c, current_d, current_room_map))
```

▼ Q3. ✍️ CGV 극장 좌석 자리 구하기

❓ Q. 극장의 좌석은 한 줄로 되어 있으며 왼쪽부터 차례대로 1번부터 N번까지 번호가 매겨져 있다.
공연을 보러 온 사람들은 자기의 입장권에 표시되어 있는 좌석에 앉아야 한다.

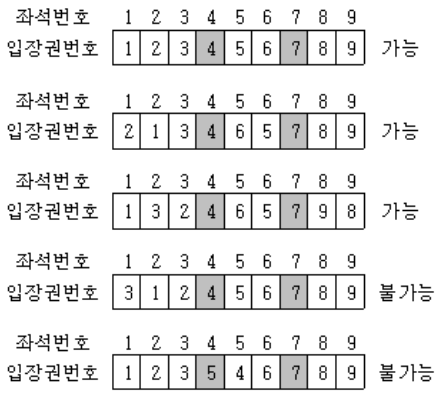
예를 들어서, 입장권에 5번이 쓰여 있으면 5번 좌석에 앉아야 한다.
단, 자기의 바로 왼쪽 좌석 또는 바로 오른쪽 좌석으로는 자리를 옮길 수 있다.

예를 들어서, 7번 입장권을 가진 사람은 7번 좌석은 물론이고,
6번 좌석이나 8번 좌석에도 앉을 수 있다.
그러나 5번 좌석이나 9번 좌석에는 앉을 수 없다.

그런데 이 극장에는 "VIP 회원"들이 있다.
이 사람들은 반드시 자기 좌석에만 앉아야 하며 옆 좌석으로 자리를 옮길 수 없다.

예를 들어서,
그림과 같이 좌석이 9개이고,
4번 좌석과 7번 좌석이 VIP석인 경우에 <123456789>는 물론 가능한 배치이다.
또한 <213465789> 와 <132465798> 도 가능한 배치이다.
그러나 <312456789> 와 <123546789> 는 허용되지 않는 배치 방법이다.

오늘 공연은 입장권이 매진되어 1번 좌석부터 N번 좌석까지 모든 좌석이 다 팔렸다.
총 좌석의 개수와 VIP 회원들의 좌석 번호들이 주어졌을 때,
사람들이 좌석에 앉는 서로 다른 방법의 가짓수를 반환하시오.




```
seat_count = 9
vip_seat_array = [4, 7]
```

- 이 문제를 풀기 위해서는 어떻게 해야 할까요? 아래 코드를 복사 붙여넣기 하고 함수를 작성해보세요! 2분 정도 고민해 본 다음, 아래 방법들을 펼쳐 봅시다!

▼ **[코드스니펫] CGV 극장 좌석 자리 구하기**

```
seat_count = 9
vip_seat_array = [4, 7]


def get_all_ways_of_theater_seat(total_count, fixed_seat_array):
```

```
    return


# 12가 출력되어야 합니다!
print(get_all_ways_of_theater_seat(seat_count, vip_seat_array))
```