BFS & DFS
백트래킹

목차

BFS 개념 및 원리
구현 및 동작 과정

DFS 개념 및 원리
구현 및 동작 과정

백트래킹 개념 및 원리
기본문제 및 심화문제 소개

BFS

너비 우선 탐색(BFS, Breadth-First Search)

너비 우선 탐색이란

루트 노드(혹은 다른 임의의 노드)에서 시작해서 인접한 노드를 먼저 탐색하는 방법

- 시작 정점으로부터 가까운 정점을 먼저 방문하고 멀리 떨어져 있는 정점을 나중에 방문하는 순회 방법이다.
- 즉, 깊게(deep) 탐색하기 전에 넓게(wide) 탐색하는 것이다.
- 사용하는 경우: **두 노드 사이의 최단 경로** 혹은 **임의의 경로를 찾고 싶을 때** 이 방법을 선택한다.
  - Ex) 지구상에 존재하는 모든 친구 관계를 그래프로 표현한 후 Ash와 Vanessa 사이에 존재하는 경로를 찾는 경우
  - 깊이 우선 탐색의 경우 - 모든 친구 관계를 다 살펴봐야 할지도 모른다.
  - 너비 우선 탐색의 경우 - Ash와 가까운 관계부터 탐색
- 너비 우선 탐색(BFS)이 깊이 우선 탐색(DFS)보다 좀 더 복잡하다.

BFS

너비 우선 탐색(BFS)의 특징

- 직관적이지 않은 면이 있다.
  - BFS는 시작 노드에서 시작해서 거리에 따라 단계별로 탐색한다고 볼 수 있다.
- BFS는 **재귀적으로 동작하지 않는다.**
- 이 알고리즘을 구현할 때 가장 큰 차이점은, 그래프 탐색의 경우 **어떤 노드를 방문했었는지 여부를 반드시 검사** 해야 한다는 것이다.
  - 이를 검사하지 않을 경우 무한루프에 빠질 위험이 있다.
- BFS는 방문한 노드들을 차례로 저장한 후 꺼낼 수 있는 자료 구조인 **큐(Queue)를 사용한다.**
  - 즉, **선입선출(FIFO)** 원칙으로 탐색
  - 일반적으로 큐를 이용해서 반복적 형태로 구현하는 것이 가장 잘 동작한다.
- 'Prim', 'Dijkstra' 알고리즘과 유사하다.

BFS

너비 우선 탐색(BFS)의 시간 복잡도

- 인접 리스트로 표현된 그래프: O(N+E)
- 인접 행렬로 표현된 그래프: O(N^2)
- 깊이 우선 탐색(DFS)과 마찬가지로 그래프 내에 적은 숫자의 간선만을 가지는 **희소 그래프 (Sparse Graph)** 의 경우 인접 행렬보다 인접 리스트를 사용하는 것이 유리하다.

DFS

깊이 우선 탐색(DFS, Depth-First Search)

깊이 우선 탐색이란

루트 노드(혹은 다른 임의의 노드)에서 시작해서 다음 분기(branch)로 넘어가기 전에 해당 분기를 완벽하게 탐색하는 방법

- 미로를 탐색할 때 한 방향으로 갈 수 있을 때까지 계속 가다가 더 이상 갈 수 없게 되면 다시 가장 가까운 갈림길로 돌아와서 이곳으로부터 다른 방향으로 다시 탐색을 진행하는 방법과 유사하다.
- 즉, 넓게(wide) 탐색하기 전에 깊게(deep) 탐색하는 것이다.
- 사용하는 경우: **모든 노드를 방문** 하고자 하는 경우에 이 방법을 선택한다.
- 깊이 우선 탐색(DFS)이 너비 우선 탐색(BFS)보다 좀 더 간단하다.
- 단순 검색 속도 자체는 너비 우선 탐색(BFS)에 비해서 느리다.

DFS

깊이 우선 탐색(DFS)의 특징

- 자기 자신을 호출하는 **순환 알고리즘의 형태** 를 가지고 있다.
- 전위 순회(Pre-Order Traversals)를 포함한 다른 형태의 트리 순회는 모두 DFS의 한 종류이다.
- 이 알고리즘을 구현할 때 가장 큰 차이점은, 그래프 탐색의 경우 **어떤 노드를 방문했었는지 여부를 반드시 검사** 해야 한다는 것이다.
  - 이를 검사하지 않을 경우 무한루프에 빠질 위험이 있다.

DFS

너비 우선 탐색(BFS)의 시간 복잡도

- 인접 리스트로 표현된 그래프: O(N+E)
- 인접 행렬로 표현된 그래프: O(N^2)
- 깊이 우선 탐색(DFS)과 마찬가지로 그래프 내에 적은 숫자의 간선만을 가지는 **희소 그래프 (Sparse Graph)** 의 경우 인접 행렬보다 인접 리스트를 사용하는 것이 유리하다.

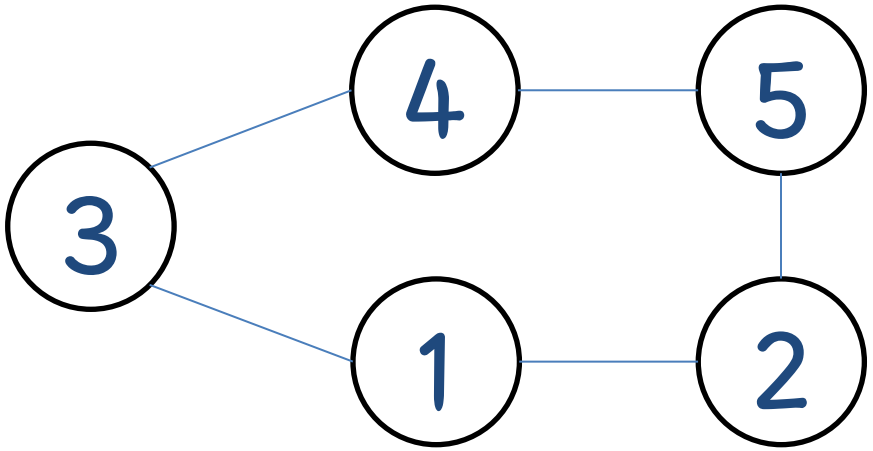
BFS & DFS

```
static StringBuilder sb = new StringBuilder();
static int N, M, S; // N : 정점의 개수, M = 간선의 개수, S = 탐색 시작 정점 번호
static ArrayList<Integer>[] list; // 각 정점 별 간선 정보를 저장하는 리스트
static boolean[] visited; // 정점의 방문 여부를 저장하는 배열

public static void main(String[] args) throws IOException {
    BufferedReader br = new BufferedReader(new InputStreamReader(System.in));
    StringTokenizer st = new StringTokenizer(br.readLine());

    N = Integer.parseInt(st.nextToken());
    M = Integer.parseInt(st.nextToken());
    S = Integer.parseInt(st.nextToken());

    // 리스트 초기화
    list = new ArrayList[N+1];
    for(int i = 1; i <= N; i++) {
        list[i] = new ArrayList<>();
    }

    visited = new boolean[N+1];

    for(int i = 0; i < M; i++) {
        st = new StringTokenizer(br.readLine());
        int cur = Integer.parseInt(st.nextToken());
        int next = Integer.parseInt(st.nextToken());
        // 양방향 그래프 간선정보 저장
        list[cur].add(next);
        list[next].add(cur);
    }

    dfs(S); // 시작정점을 매개변수로 넣어줌
    sb.append("\n");
    bfs(S); // 시작정점을 매개변수로 넣어줌

    System.out.println(sb.toString());
```

변수 선언부 & main 함수

BFS & DFS

```java
static void dfs(int node) { // node는 현재 방문중인 정점
    sb.append(node + " ");
    visited[node] = true; // 현재 정점을 방문처리하고
    // 연결된 다른 정점들의 방문을 고려한다
    for(int i = 0; i < list[node].size(); i++) { // list[node] : 현재 정점과 연결된 다른 정점들의 리스트
        // 만약 연결된 정점이 방문한 적이 없다면 방문
        if(!visited[list[node].get(i)]) {
            dfs(list[node].get(i));
            // 해당 정점을 방문하여 갈 수 있는 모든 경로를 탐색하고 나왔기 때문에 방문 처리를 false
            visited[node] = false;
        }
    }
}
```



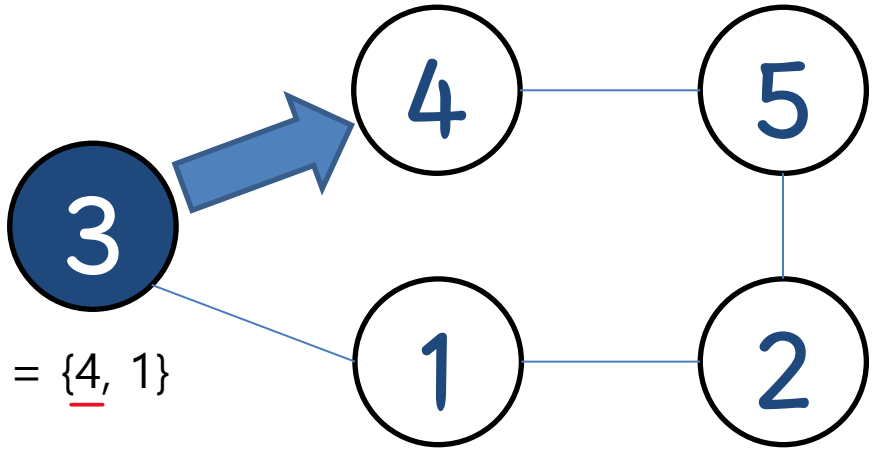
Visited

1 2 3 4 5

Output

BFS & DFS

```java
static void dfs(int node) { // node는 현재 방문중인 정점
    sb.append(node + " ");
    visited[node] = true; // 현재 정점을 방문처리하고
    // 연결된 다른 정점들의 방문을 고려한다
    for(int i = 0; i < list[node].size(); i++) { // list[node] : 현재 정점과 연결된 다른 정점들의 리스트
        // 만약 연결된 정점이 방문한 적이 없다면 방문
        if(!visited[list[node].get(i)]) {
            dfs(list[node].get(i));
            // 해당 정점을 방문하여 갈 수 있는 모든 경로를 탐색하고 나왔기 때문에 방문 처리를 false
            visited[node] = false;
        }
    }
}
```



list[3] = {4, 1}

Visited

1 2 ~~3~~ 4 5

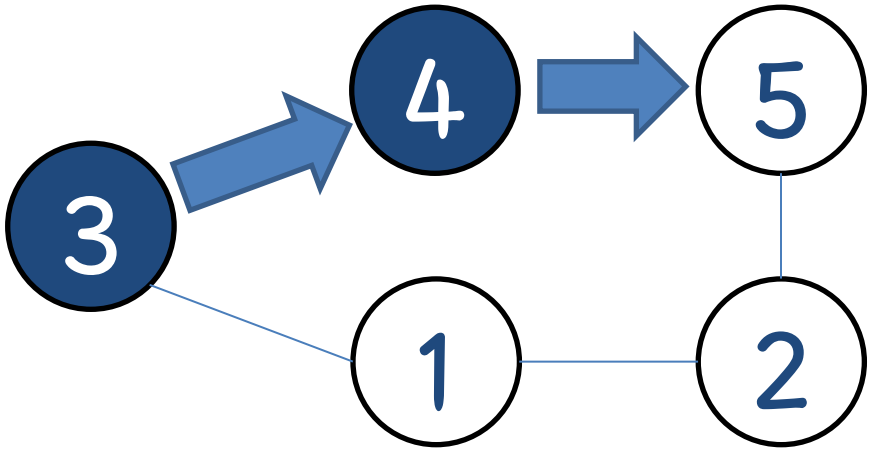
Output

3

BFS & DFS

```java
static void dfs(int node) { // node는 현재 방문중인 정점
    sb.append(node + " ");
    visited[node] = true; // 현재 정점을 방문처리하고
    // 연결된 다른 정점들의 방문을 고려한다
    for(int i = 0; i < list[node].size(); i++) { // list[node] : 현재 정점과 연결된 다른 정점들의 리스트
        // 만약 연결된 정점이 방문한 적이 없다면 방문
        if(!visited[list[node].get(i)]) {
            dfs(list[node].get(i));
            // 해당 정점을 방문하여 갈 수 있는 모든 경로를 탐색하고 나왔기 때문에 방문 처리를 false
            visited[node] = false;
        }
    }
}
```

list[4] = {5, ~~3~~}



Visited

1 2 ~~3~~ ~~4~~ 5

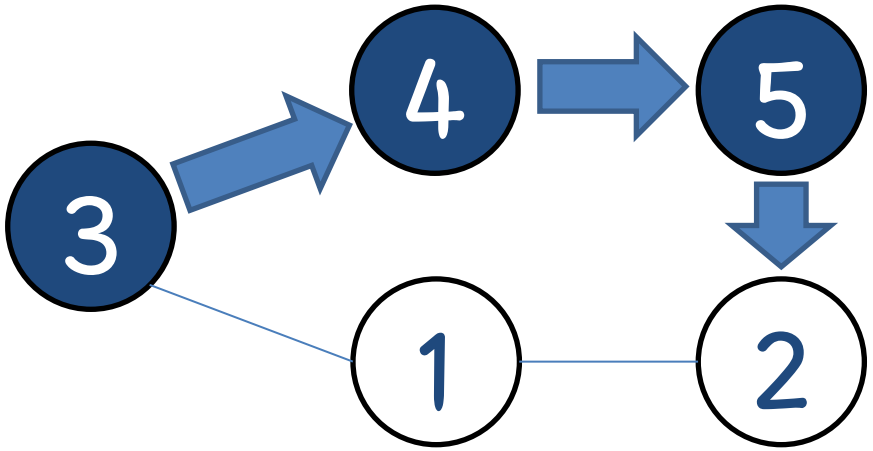
Output

3 4

BFS & DFS

```java
static void dfs(int node) { // node는 현재 방문중인 정점
    sb.append(node + " ");
    visited[node] = true; // 현재 정점을 방문처리하고
    // 연결된 다른 정점들의 방문을 고려한다
    for(int i = 0; i < list[node].size(); i++) { // list[node] : 현재 정점과 연결된 다른 정점들의 리스트
        // 만약 연결된 정점이 방문한 적이 없다면 방문
        if(!visited[list[node].get(i)]) {
            dfs(list[node].get(i));
            // 해당 정점을 방문하여 갈 수 있는 모든 경로를 탐색하고 나왔기 때문에 방문 처리를 false
            visited[node] = false;
        }
    }
}
```

list[5] = {~~4~~, 2}



Visited

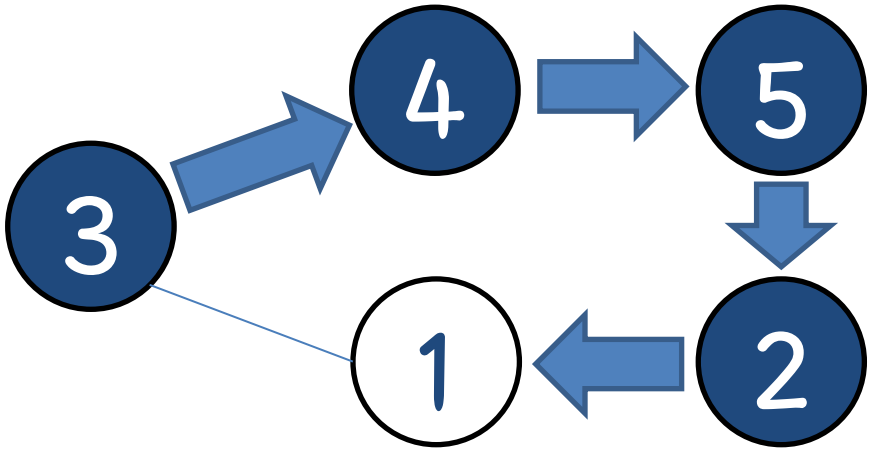
1 2 ~~3~~ ~~4~~ ~~5~~

Output

3 4 5

BFS & DFS

```java
static void dfs(int node) { // node는 현재 방문중인 정점
    sb.append(node + " ");
    visited[node] = true; // 현재 정점을 방문처리하고
    // 연결된 다른 정점들의 방문을 고려한다
    for(int i = 0; i < list[node].size(); i++) { // list[node] : 현재 정점과 연결된 다른 정점들의 리스트
        // 만약 연결된 정점이 방문한 적이 없다면 방문
        if(!visited[list[node].get(i)]) {
            dfs(list[node].get(i));
            // 해당 정점을 방문하여 갈 수 있는 모든 경로를 탐색하고 나왔기 때문에 방문 처리를 false
            visited[node] = false;
        }
    }
}
```



list[2] = {~~5~~, 1}

Visited

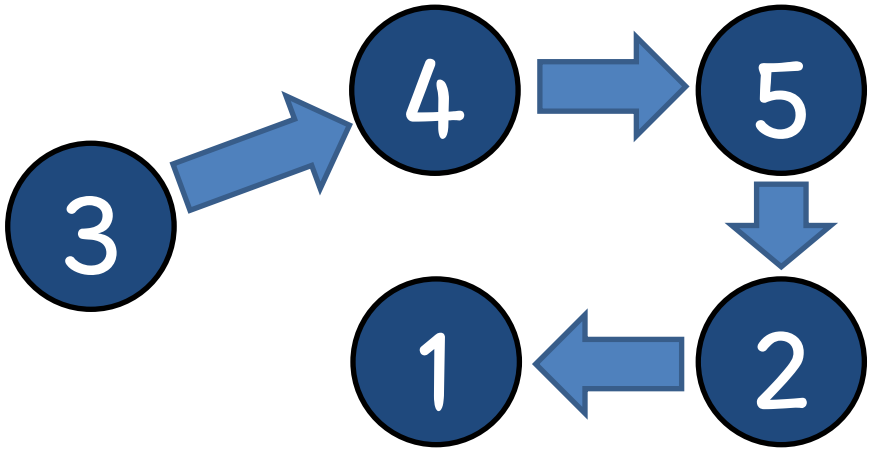
1 ~~2~~ ~~3~~ ~~4~~ ~~5~~

Output

3 4 5 2

BFS & DFS

```
static void dfs(int node) { // node는 현재 방문중인 정점
    sb.append(node + " ");
    visited[node] = true; // 현재 정점을 방문처리하고
    // 연결된 다른 정점들의 방문을 고려한다
    for(int i = 0; i < list[node].size(); i++) { // list[node] : 현재 정점과 연결된 다른 정점들의 리스트
        // 만약 연결된 정점이 방문한 적이 없다면 방문
        if(!visited[list[node].get(i)]) {
            dfs(list[node].get(i));
            // 해당 정점을 방문하여 갈 수 있는 모든 경로를 탐색하고 나왔기 때문에 방문 처리를 false
            visited[node] = false;
        }
    }
}
```



list[1] = {~~2~~, ~~3~~}

Visited

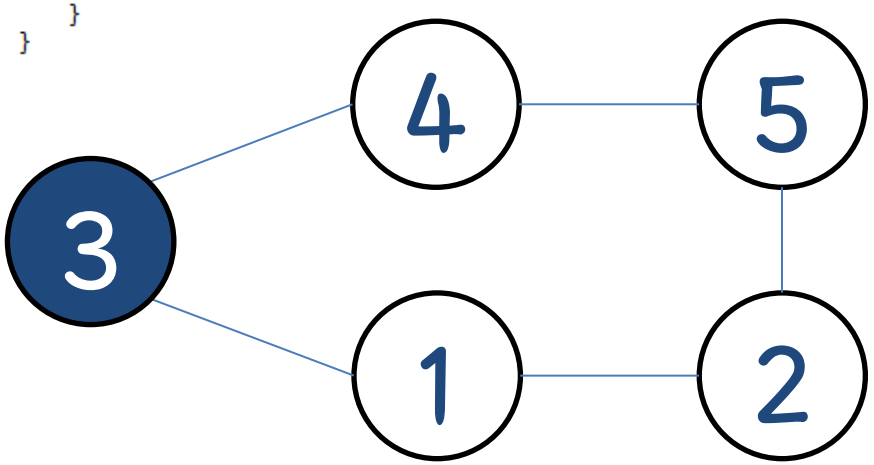
~~1~~ ~~2~~ ~~3~~ ~~4~~ ~~5~~

Output

3 4 5 2 1

BFS & DFS

```java
public static void bfs(int start) { // start는 매개변수로 넣어 준 시작 정점
    // 다음으로 방문할 정점의 정보를 넣어줄 queue를 선언
    Queue<Integer> q = new ArrayDeque<>();
    // 시작 정점을 넣어준다.
    q.add(start);
    // 방문 정보를 저장하는 배열을 초기화하고 시작 정점을 방문 처리
    visited = new boolean[N+1];
    visited[start] = true;

    // 큐가 빌 때까지 = 즉, 더 이상 방문할 정점이 없을 때까지
    while(!q.isEmpty()) {
        // 큐의 맨 앞의 요소(제일 먼저 넣은 정점)을 꺼낸다.
        int node = q.poll();
        sb.append(node + " ");
        for(int i = 0; i < list[node].size(); i++) { // list[node] : 현재 정점과 연결된 다른 정점들의 리스트
            // 만약 연결된 정점이 방문한 적이 없다면
            if(!visited[list[node].get(i)]) {
                // 해당 노드를 방문처리하고
                visited[list[node].get(i)] = true;
                // 큐에 저장하여 다음 방문지로 결정 ~^^~
                q.offer(list[node].get(i));
            }
        }
    }
}
```



Queue

3

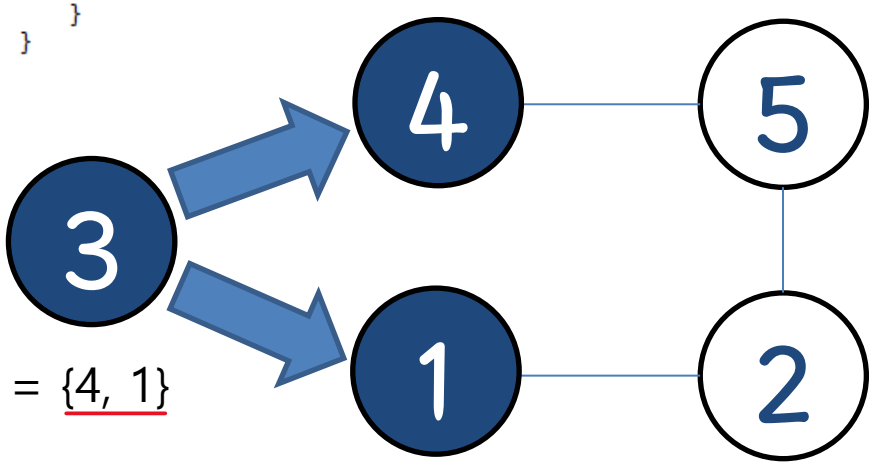
Visited

1 2 ~~3~~ 4 5

Output

BFS & DFS

```java
public static void bfs(int start) { // start는 매개변수로 넣어 준 시작 정점
    // 다음으로 방문할 정점의 정보를 넣어줄 queue를 선언
    Queue<Integer> q = new ArrayDeque<>();
    // 시작 정점을 넣어준다.
    q.add(start);
    // 방문 정보를 저장하는 배열을 초기화하고 시작 정점을 방문 처리
    visited = new boolean[N+1];
    visited[start] = true;

    // 큐가 빌 때까지 = 즉, 더 이상 방문할 정점이 없을 때까지
    while(!q.isEmpty()) {
        // 큐의 맨 앞의 요소(제일 먼저 넣은 정점)을 꺼낸다.
        int node = q.poll();
        sb.append(node + " ");
        for(int i = 0; i < list[node].size(); i++) { // list[node] : 현재 정점과 연결된 다른 정점들의 리스트
            // 만약 연결된 정점이 방문한 적이 없다면
            if(!visited[list[node].get(i)]) {
                // 해당 노드를 방문처리하고
                visited[list[node].get(i)] = true;
                // 큐에 저장하여 다음 방문지로 결정 ~^^~
                q.offer(list[node].get(i));
            }
        }
    }
}
```



list[3] = {4, 1}

Queue

4 1

Visited

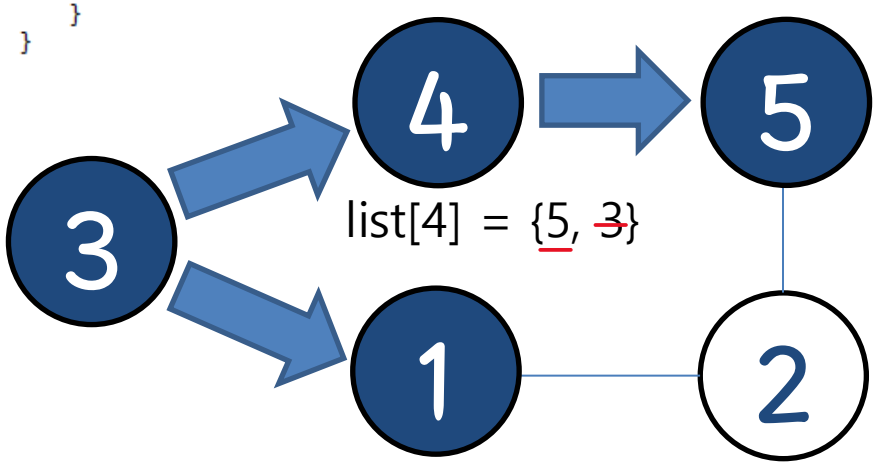
~~1~~ 2 ~~3~~ ~~4~~ 5

Output

3

BFS & DFS

```java
public static void bfs(int start) { // start는 매개변수로 넣어 준 시작 정점
    // 다음으로 방문할 정점의 정보를 넣어줄 queue를 선언
    Queue<Integer> q = new ArrayDeque<>();
    // 시작 정점을 넣어준다.
    q.add(start);
    // 방문 정보를 저장하는 배열을 초기화하고 시작 정점을 방문 처리
    visited = new boolean[N+1];
    visited[start] = true;

    // 큐가 빌 때까지 = 즉, 더 이상 방문할 정점이 없을 때까지
    while(!q.isEmpty()) {
        // 큐의 맨 앞의 요소(제일 먼저 넣은 정점)을 꺼낸다.
        int node = q.poll();
        sb.append(node + " ");
        for(int i = 0; i < list[node].size(); i++) { // list[node] : 현재 정점과 연결된 다른 정점들의 리스트
            // 만약 연결된 정점이 방문한 적이 없다면
            if(!visited[list[node].get(i)]) {
                // 해당 노드를 방문처리하고
                visited[list[node].get(i)] = true;
                // 큐에 저장하여 다음 방문지로 결정 ~^^~
                q.offer(list[node].get(i));
            }
        }
    }
}
```



Queue

1 5

Visited

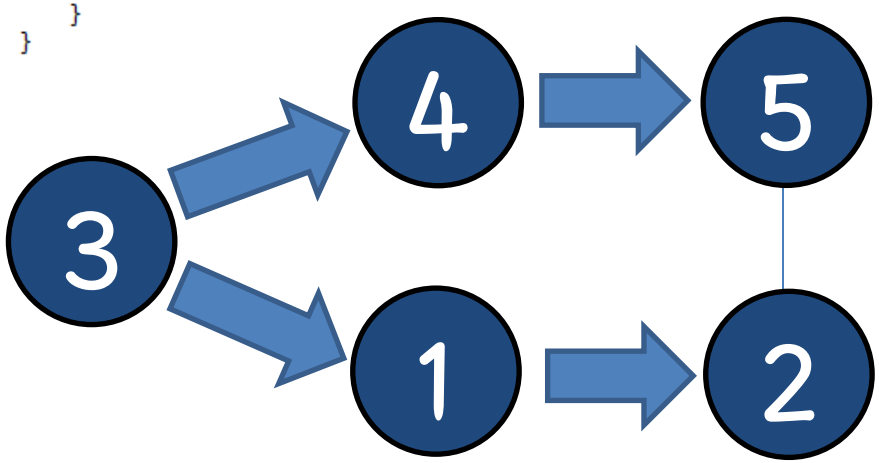
~~1~~ 2 ~~3~~ ~~4~~ ~~5~~

Output

3 4

BFS & DFS

```java
public static void bfs(int start) { // start는 매개변수로 넣어 준 시작 정점
    // 다음으로 방문할 정점의 정보를 넣어줄 queue를 선언
    Queue<Integer> q = new ArrayDeque<>();
    // 시작 정점을 넣어준다.
    q.add(start);
    // 방문 정보를 저장하는 배열을 초기화하고 시작 정점을 방문 처리
    visited = new boolean[N+1];
    visited[start] = true;

    // 큐가 빌 때까지 = 즉, 더 이상 방문할 정점이 없을 때까지
    while(!q.isEmpty()) {
        // 큐의 맨 앞의 요소(제일 먼저 넣은 정점)을 꺼낸다.
        int node = q.poll();
        sb.append(node + " ");
        for(int i = 0; i < list[node].size(); i++) { // list[node] : 현재 정점과 연결된 다른 정점들의 리스트
            // 만약 연결된 정점이 방문한 적이 없다면
            if(!visited[list[node].get(i)]) {
                // 해당 노드를 방문처리하고
                visited[list[node].get(i)] = true;
                // 큐에 저장하여 다음 방문지로 결정 ~^^~
                q.offer(list[node].get(i));
            }
        }
    }
}
```



list[1] = {2, ~~3~~}

Queue

5 2

Visited

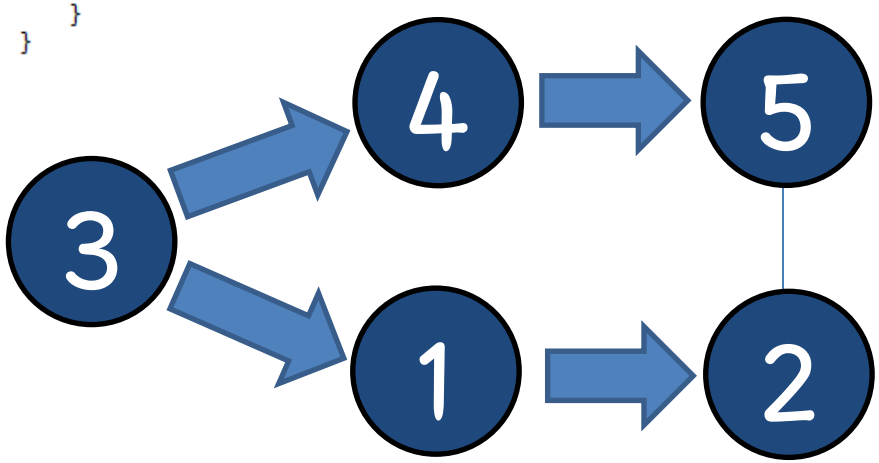
~~1~~ ~~2~~ ~~3~~ ~~4~~ ~~5~~

Output

3 4 1

BFS & DFS

```java
public static void bfs(int start) { // start는 매개변수로 넣어 준 시작 정점
    // 다음으로 방문할 정점의 정보를 넣어줄 queue를 선언
    Queue<Integer> q = new ArrayDeque<>();
    // 시작 정점을 넣어준다.
    q.add(start);
    // 방문 정보를 저장하는 배열을 초기화하고 시작 정점을 방문 처리
    visited = new boolean[N+1];
    visited[start] = true;

    // 큐가 빌 때까지 = 즉, 더 이상 방문할 정점이 없을 때까지
    while(!q.isEmpty()) {
        // 큐의 맨 앞의 요소(제일 먼저 넣은 정점)을 꺼낸다.
        int node = q.poll();
        sb.append(node + " ");
        for(int i = 0; i < list[node].size(); i++) { // list[node] : 현재 정점과 연결된 다른 정점들의 리스트
            // 만약 연결된 정점이 방문한 적이 없다면
            if(!visited[list[node].get(i)]) {
                // 해당 노드를 방문처리하고
                visited[list[node].get(i)] = true;
                // 큐에 저장하여 다음 방문지로 결정 ~^^~
                q.offer(list[node].get(i));
            }
        }
    }
}
```



list[5] = {~~4~~, ~~2~~}

Queue

2

Visited

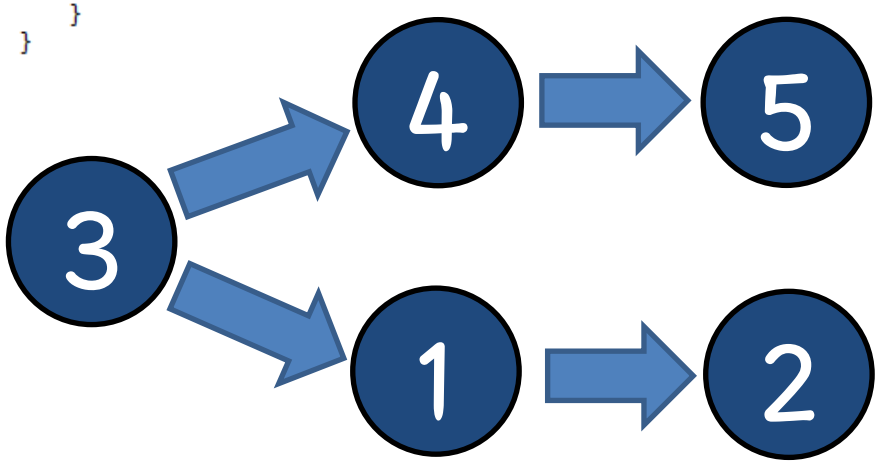
~~1~~ ~~2~~ ~~3~~ ~~4~~ ~~5~~

Output

3 4 1 5

BFS & DFS

```java
public static void bfs(int start) { // start는 매개변수로 넣어 준 시작 정점
    // 다음으로 방문할 정점의 정보를 넣어줄 queue를 선언
    Queue<Integer> q = new ArrayDeque<>();
    // 시작 정점을 넣어준다.
    q.add(start);
    // 방문 정보를 저장하는 배열을 초기화하고 시작 정점을 방문 처리
    visited = new boolean[N+1];
    visited[start] = true;

    // 큐가 빌 때까지 = 즉, 더 이상 방문할 정점이 없을 때까지
    while(!q.isEmpty()) {
        // 큐의 맨 앞의 요소(제일 먼저 넣은 정점)을 꺼낸다.
        int node = q.poll();
        sb.append(node + " ");
        for(int i = 0; i < list[node].size(); i++) { // list[node] : 현재 정점과 연결된 다른 정점들의 리스트
            // 만약 연결된 정점이 방문한 적이 없다면
            if(!visited[list[node].get(i)]) {
                // 해당 노드를 방문처리하고
                visited[list[node].get(i)] = true;
                // 큐에 저장하여 다음 방문지로 결정 ~^^~
                q.offer(list[node].get(i));
            }
        }
    }
}
```



Queue

Visited

~~1~~ ~~2~~ ~~3~~ ~~4~~ ~~5~~

Output

3 4 1 5 2

list[2] = {~~5~~, ~~1~~}

백트래킹

- **백트래킹(Backtracking)**

  해를 찾아가는 도중, 지금의 경로가 해가 될 것 같지 않으면 그 경로를 더이상 가지 않고 되돌아갑니다.

  즉, 코딩에서는 반복문의 횟수까지 줄일 수 있으므로 효율적입니다.

  이를 **가지치기**라고 하는데, 불필요한 부분을 쳐내고 최대한 올바른 쪽으로 간다는 의미입니다.

  일반적으로, 불필요한 경로를 조기에 차단할 수 있게 되어 경우의 수가 줄어들지만, 만약 N!의 경우의 수를 가진 문제에서 최악의 경우에는 여전히 지수함수 시간을 필요로 하므로 처리가 불가능 할 수도 있습니다. **가지치기를 얼마나 잘하느냐에 따라 효율성이 결정되게 됩니다.**

- 정리하자면, 백트래킹은 모든 가능한 경우의 수 중에서 **특정한 조건을 만족하는 경우만 살펴보는 것**입니다.
- 즉 **답이 될 만한지 판단하고 그렇지 않으면 그 부분까지 탐색하는 것을 하지 않고 가지치기 하는 것**을 백트래킹이라고 생각하면 됩니다.
- 주로 문제 풀이에서는 **DFS 등으로 모든 경우의 수를 탐색하는 과정**에서, **조건문 등을 걸어 답이 절대로 될 수 없는 상황을 정의하고, 그러한 상황일 경우에는 탐색을 중지시킨 뒤 그 이전으로 돌아가서 다시 다른 경우를 탐색하게끔** 구현할 수 있습니다.

백트래킹

암호 만들기 성공

☆

골드 V

시간 제한	메모리 제한	제출	정답	맞힌 사람	정답 비율
2 초	128 MB	43285	20404	14154	44.814%

문제

바로 어제 최백준 조교가 방 열쇠를 주머니에 넣은 채 깜빡하고 서울로 가 버리는 황당한 상황에 직면한 조교들은, 702호에 새로운 보안 시스템을 설치하기로 하였다. 이 보안 시스템은 열쇠가 아닌 암호로 동작하게 되어 있는 시스템이다.

암호는 서로 다른 L개의 알파벳 소문자들로 구성되며 최소 한 개의 모음(a, e, i, o, u)과 최소 두 개의 자음으로 구성되어 있다고 알려져 있다. 또한 정렬된 문자열을 선호하는 조교들의 성향으로 미루어 보아 암호를 이루는 알파벳이 암호에서 증가하는 순서로 배열되었을 것이라고 추측된다. 즉, abc는 가능성이 있는 암호이지만 bac는 그렇지 않다.

새 보안 시스템에서 조교들이 암호로 사용했을 법한 문자의 종류는 C가지가 있다고 한다. 이 알파벳을 입수한 민식, 영식 형제는 조교들의 방에 침투하기 위해 암호를 추측해 보려고 한다. C개의 문자들이 모두 주어졌을 때, 가능성 있는 암호들을 모두 구하는 프로그램을 작성하시오.

입력

첫째 줄에 두 정수 L, C가 주어진다. ($3 \le L \le C \le 15$) 다음 줄에는 C개의 문자들이 공백으로 구분되어 주어진다. 주어지는 문자들은 알파벳 소문자이며, 중복되는 것은 없다.

출력

각 줄에 하나씩, 사전식으로 가능성 있는 암호를 모두 출력한다.

백트래킹

```
public static void make(int cnt, int start, int vowel, int consonant) {
    if(vowel == L - 1 || consonant == L) return;

    if(cnt == L) {
        for(int i = 0; i < choosed.length; i++) {
            sb.append(choosed[i]);
        }
        sb.append("\n");
        return;
    }

    for(int i = start; i < pos.length; i++) {
        choosed[cnt] = pos[i];
        if(pos[i] == 'a' || pos[i] == 'e' || pos[i] == 'i'  || pos[i] == 'o'  || pos[i] == 'u') {
            make(cnt + 1, i + 1, vowel + 1, consonant);
        }
        else {
            make(cnt + 1, i + 1, vowel, consonant + 1);
        }
    }
}
```

기본&심화문제

기본&심화 문제

문제 번호	제목	정보	맞은 사람	제출	정답 비율
2667	1 단지번호붙이기		29265	109435	40.283%
2210	2 숫자판 점프		3213	5446	74.860%
2529	2 부등호		5691	15874	51.349%
15270	3 친구 팰린드롬		75	486	28.302%
2206	4 벽 부수고 이동하기		12006	78617	22.822%
1941	3 소문난 칠공주		2023	6363	48.041%
15684	4 사다리 조작		5595	44995	21.553%


THANKS