김범수

탐색

Search

By POSCAT

Contents

브루트 포스

- brute = 무식한, force = 힘
- 브루트 포스는 가능한 모든 경우의 수를 시도해보면서 문제를 해결하는 기법입니다.
- 브루트 포스를 사용하는 경우, 모든 경우의 수를 시도하기 때문에 비효율적이므로 많이 사용되지는 않는 기법입니다.
- O(k^N)꼴의 시간복잡도를 가지는 경우, N이 매우 작은 값이 아닌 한 TLE(time limit error)가 발생하므로, N의 범위를 확인하고 시간 복잡도를 통해 검증하는 과정이 중요합니다.
- 다음의 문제를 한 번 풀어봅시다.

브루트 포스

- https://www.acmicpc.net/problem/10819

문제

N개의 정수로 이루어진 배열 A가 주어진다. 이때, 배열에 들어있는 정수의 순서를 적절히 바꿔서 다음 식의 최댓값을 구하는 프로그램을 작성하시오.

$$|A[0] - A[1]| + |A[1] - A[2]| + \ldots + |A[N-2] - A[N-1]|$$

입력

첫째 줄에 N ($3 \leq N \leq 8$)이 주어진다. 둘째 줄에는 배열 A에 들어있는 정수가 주어진다. 배열에 들어있는 정수는 -100보다 크거나 같고, 100보다 작거나 같다.

출력

첫째 줄에 배열에 들어있는 수의 순서를 적절히 바꿔서 얻을 수 있는 식의 최댓값을 출력한다.

브루트 포스

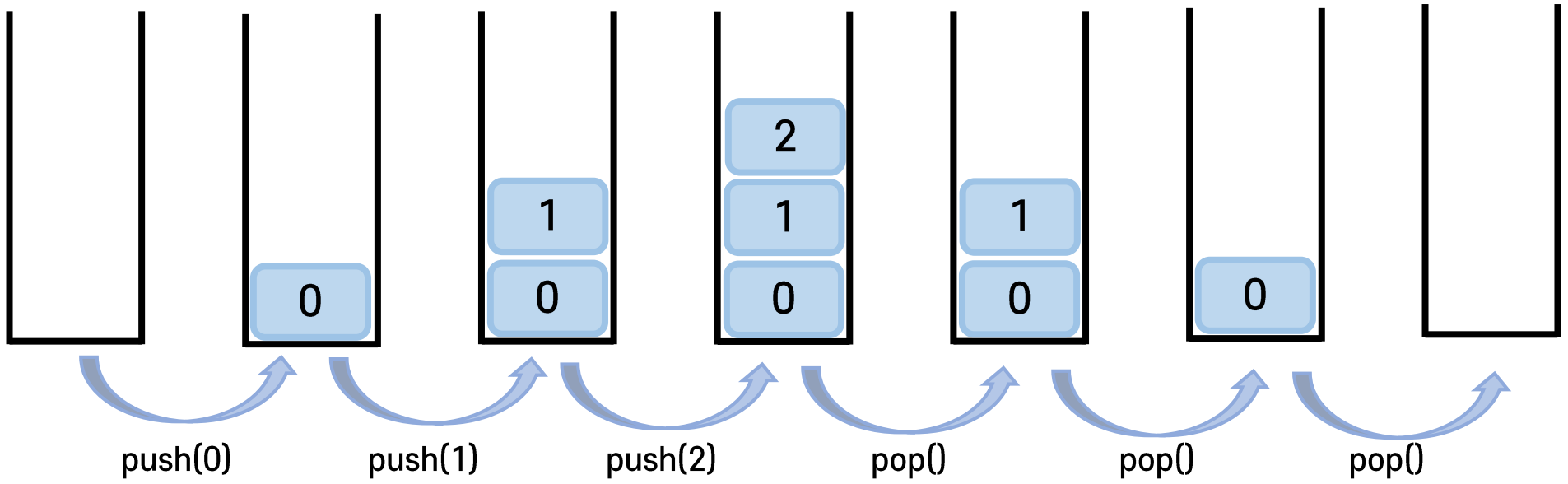
- c++에는 next_permutation이라는 함수가 존재합니다.
- 순열을 입력받아, 해당 순열을 다음 순열로 바꾸는 함수입니다.
- 예를 들어 {1, 2, 3}을 입력받으면, {1, 3, 2}로 변경합니다.
- {1, 2, 3} -> {1, 3, 2} -> {2, 1, 3} -> {2, 3, 1} -> {3, 1, 2} -> {3, 2, 1} -> {1, 2, 3}과 같은 순서로 변경하게 됩니다.
- 이 함수는 첫 순열, 즉 {1, 2, 3}으로 돌아오면 false, 아니라면 true를 반환합니다.
- 이 함수를 사용하여 주어진 문제를 간단하게 풀 수 있습니다.

stack과 queue

- 알고리즘에서는 다양한 자료구조가 존재합니다.
- 자료구조란 데이터를 효율적으로 관리하기 위한 데이터의 저장, 수정, 삭제 등을 하는 방식을 의미합니다.
- stack과 queue는 대표적인 자료구조 중 하나입니다.

stack과 queue

- stack은 쌓다 라는 사전적인 의미를 가지고 있습니다. stack 자료구조는 그 의미와 같이 데이터를 차곡차곡 쌓아올리는 형태로 저장하는 자료구조입니다.
- stack은 가장 마지막에 저장한 데이터부터 먼저 삭제할 수 있는 구조인 LIFO(Last in First out)의 구조를 가지고 있습니다.
- 아래 그림은 0, 1, 2라는 숫자를 저장하고 삭제하는 과정을 나타내는 그림입니다.

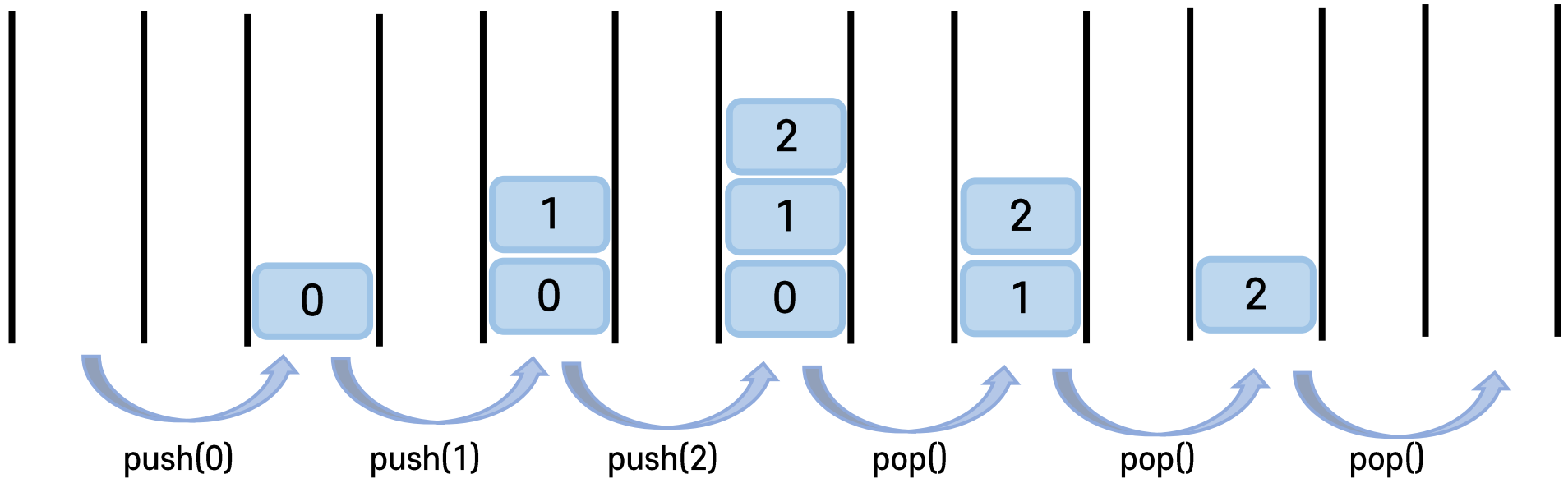


stack과 queue

- #include <stack>을 통해 이미 구현된 stack을 가져와 사용할 수 있습니다.
- stack <자료형> 이름; 의 형식으로 사용할 수 있습니다.
- push(x) 함수를 통해 x라는 데이터를 stack에 추가할 수 있습니다.
- pop() 함수를 통해 가장 위에 있는 데이터를 삭제할 수 있습니다.
- top() 함수를 통해 가장 위에 있는 데이터 값을 가져올 수 있습니다.
- size() 함수를 통해 stack에 저장된 데이터의 개수를 알 수 있습니다.
- empty() 함수를 통해 stack이 비었는지 아닌지 여부를 알 수 있습니다.
- 아래 링크의 문제를 통해 stack을 사용하는 법을 익혀봅시다.
- https://www.acmicpc.net/problem/10828

stack과 queue

- queue는 롤 큐를 잡는다 등, 우리가 평소에 사용하는 큐와 동일한 의미를 가지고 있습니다.
- stack은 queue와는 달리 가장 먼저 저장한 데이터를 가장 먼저 삭제하는 FIFO(First in First out)의 구조를 가집니다.
- 아래 그림은 0, 1, 2라는 숫자를 저장하고 삭제하는 과정을 나타내는 그림입니다.



stack과 queue

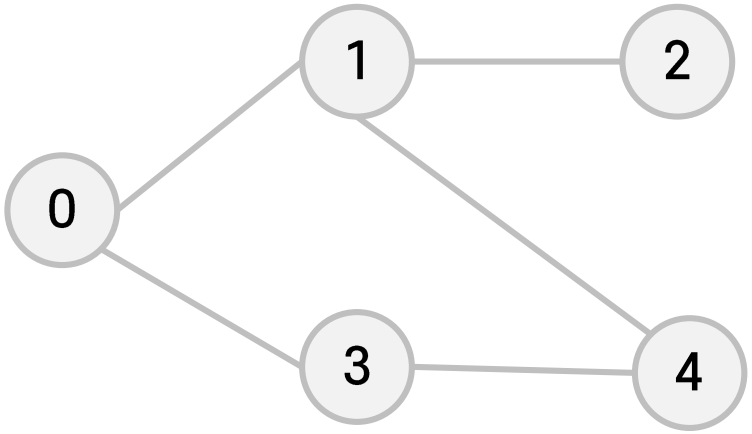
- #include <stack>을 통해 이미 구현된 queue를 가져와 사용할 수 있습니다.
- stack <자료형> 이름; 의 형식으로 사용할 수 있습니다.
- push(x) 함수를 통해 x라는 데이터를 맨 끝에 추가할 수 있습니다.
- pop() 함수를 통해 가장 앞에 있는 데이터를 삭제할 수 있습니다.
- front() 함수를 통해 가장 앞에 있는 데이터 값을 가져올 수 있습니다.
- size() 함수를 통해 queue에 저장된 데이터의 개수를 알 수 있습니다.
- empty() 함수를 통해 queue가 비었는지 아닌지 여부를 알 수 있습니다.
- 아래 링크의 문제를 통해 queue를 사용하는 법을 익혀봅시다.
- https://www.acmicpc.net/problem/10845

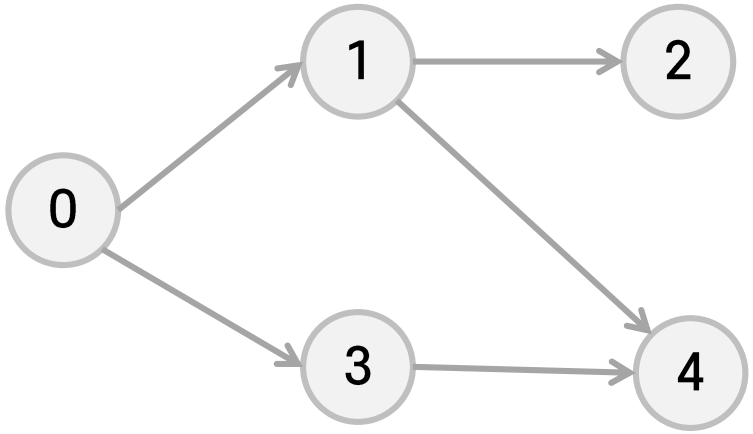
vector

- vector는 stack과 비슷한 구조를 가지고 있습니다. push_back(), pop_back() 함수를 통해 가장 마지막에 원소를 추가하고, 제거할 수 있습니다.
- stack과 queue는 각각 가장 마지막 또는 가장 처음 원소만 가져올 수 있습니다. 하지만 vector는 배열을 기반으로 작동하므로, 모든 원소에 바로 접근 할 수 있습니다.
- vector는 동적 배열, 즉 데이터의 개수에 따라 공간을 유동적으로 할당하는 배열입니다.
- 마찬가지로 vector 헤더에 구현되어 있으며, vector <int> v와 같은 형태로 사용할 수 있고, v[i]로 i번째 원소에 접근할 수 있습니다.

graph

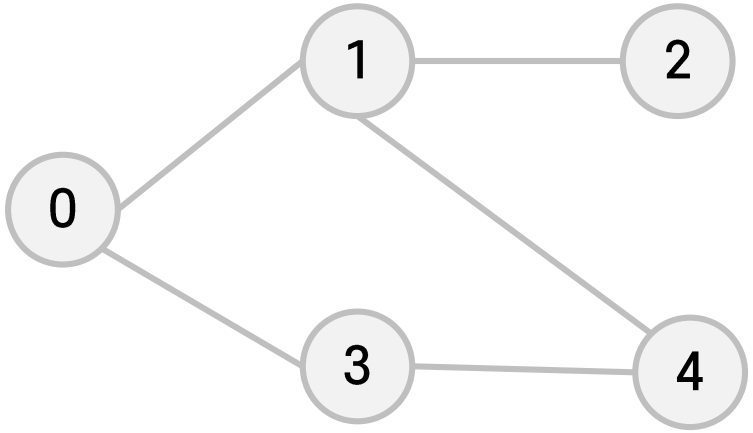
- 그래프(G, Graph)는 노드(V, vertex)와 노드를 연결하는 간선(E, Edge)으로 구성된 자료구조입니다.
- 예를 들어 지하철 노선도 등을 예로 들 수 있습니다. 각 지하철 정거장이 노드가 되고, 정거장과 정거장 사이의 길을 간선으로 표현할 수 있습니다.
- 간선은 방향이 없을 수도 있고, 방향이 존재할 수도 있습니다.
- 만약 그래프의 간선에 방향이 존재하지 않는다면 무향 그래프, 방향이 존재한다면 유향 그래프라고 합니다.





graph

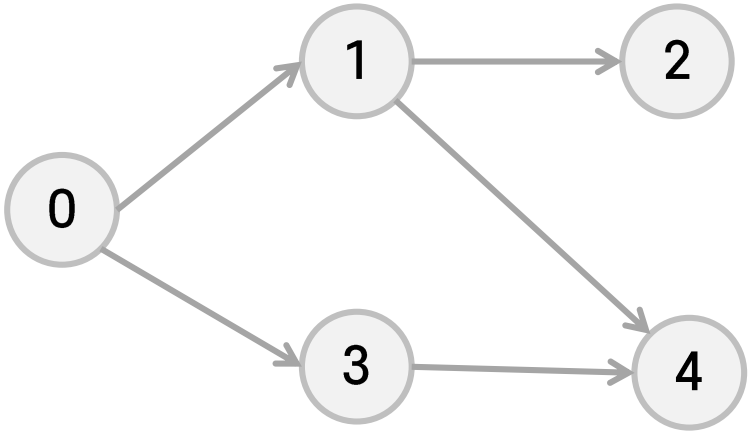
- 그래프를 저장하는 방식은 크게 인접 배열, 그리고 인접 리스트의 두 가지 방법이 존재합니다.
- 인접 배열은 노드의 개수를 V라고 한다면, V * V 크기의 배열에 간선이 존재하는 지 여부를 저장하는 방법입니다.
- 인접 배열을 adj[V][V]라 정의하면 adj[i][j]에는 node i에서 node j로 가는 간선이 존재하면 1, 아니면 0을 저장하게 됩니다.
- 무향 그래프의 경우, node i와 node j 사이에 간선이 존재하면, adj[i][j] = adj[j][i] = 1로 저장하게 됩니다.



$$\begin{pmatrix} 0 & 1 & 0 & 1 & 0 \\ 1 & 0 & 1 & 0 & 1 \\ 0 & 1 & 0 & 0 & 0 \\ 1 & 0 & 0 & 0 & 1 \\ 0 & 1 & 0 & 1 & 0 \end{pmatrix}$$

graph

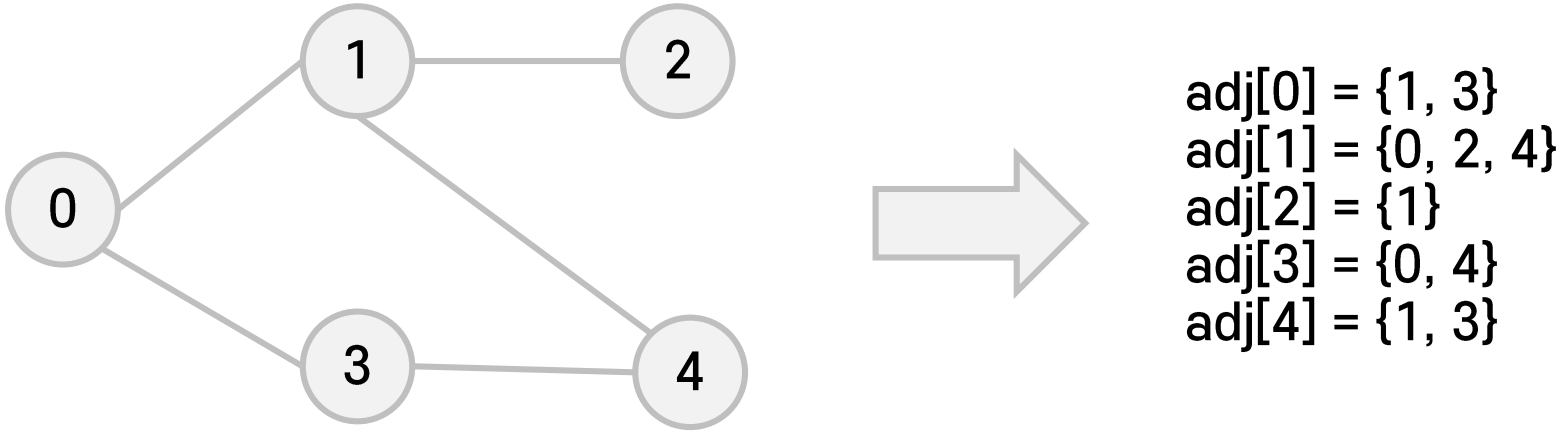
- 유향 그래프에서는 이와 달리 node i에서 node j 방향의 간선이 존재할 때 adj[i][j] = 1을 저장하게 됩니다.



$$\begin{pmatrix} 0 & 1 & 0 & 1 & 0 \\ 0 & 0 & 1 & 0 & 1 \\ 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 & 0 \end{pmatrix}$$

graph

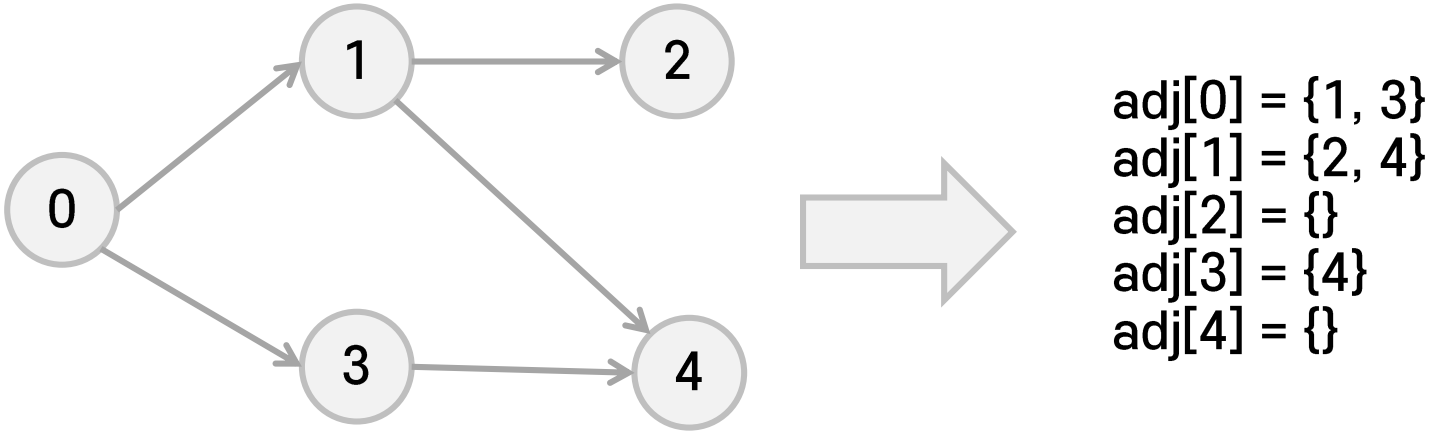
- 인접 리스트는 이와 달리 간선으로 연결된 node의 번호만 저장하여 graph를 표현합니다.
- 주로 c++에서는 vector를 사용하며, 총 node 개수 만큼의 vector를 사용합니다. (ex. vector <int> adj[V])
- 이 때 adj[i]에는 i와 연결된 노드의 번호를 저장하게 됩니다.
- 무향 그래프의 경우 node i와 node j 사이에 간선에 존재하면, adj[i]에는 j를, adj[j]에는 i를 저장합니다.



adj[0] = {1, 3}
adj[1] = {0, 2, 4}
adj[2] = {1}
adj[3] = {0, 4}
adj[4] = {1, 3}

graph

- 유향 그래프에서는 이와 달리 node i에서 node j 방향의 간선이 존재할 때 adj[i]에 j를 저장하게 됩니다.



graph

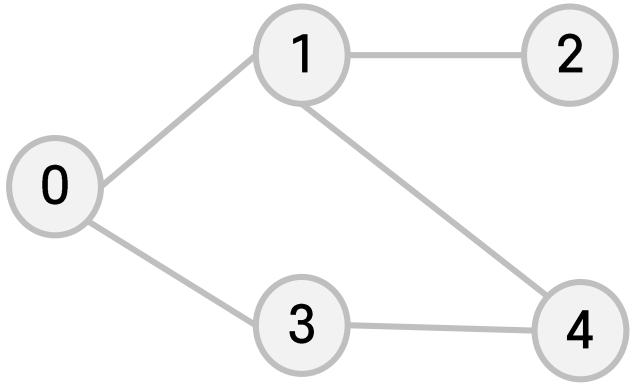
- 인접 배열과 인접 리스트, 두 방식의 장단점을 비교해봅시다.
- 먼저 사용되는 메모리의 크기를 분석해봅시다. 인접 배열은 V * V 크기의 배열을 사용하므로, V^2의 공간이 필요하지만, 인접 리스트는 간선이 존재할 때만 그 간선의 정보를 저장하므로, 모든 vector의 크기를 합하면 E의 공간만 사용됩니다.
- 만약 node i와 node j 사이에 간선이 존재하는 지 확인하고 싶다고 합시다. 인접 배열의 경우, 단순히 adj[i][j]가 1인지만 확인하므로 1번의 연산만 진행하면 되지만, 인접 리스트의 경우, adj[i] vector에 j가 존재하는 지 확인해야 하기 때문에 최악의 경우, E번의 연산을 진행하게 됩니다. (나중에 배울 이분 탐색 기법으로 log E로 줄일 수 있습니다.)

DFS

- 만약 그래프가 주어졌다면 우리는 그래프를 어떻게 탐색할 수 있을까요?
- 가장 단순한 방법으로는 시작점을 하나를 잡고, 그 주변을 돌아다니는 방법입니다.
- 이 때 돌아다니는 방식에 따라 DFS, BFS라고 불리게 됩니다.
- DFS는 간단하게 말하면, 일단 직진하면서 탐색하는 방법입니다.

DFS

- 먼저 0을 시작점으로 잡고 탐색을 해봅시다. 일단 0과 연결된 1로 이동해 봅니다. 그 후, 1에서는 1과 연결된 2로 이동해봅니다.
- 이제 2와 연결된 노드는 1밖에 없는데, 1은 이미 탐색했으므로 또 탐색할 장소가 없습니다. 따라서 1로 돌아가 봅니다.
- 1과 연결된 노드는 0, 2, 4가 있는데, 0, 2는 이미 탐색했으므로, 4로 이동해 봅니다. 그 후, 마찬가지로 4에서는 3으로 이동해 봅니다.
- 마찬가지로, 3과 연결된 노드는 0, 4밖에 없는데 둘 다 탐색했으므로, 또 탐색할 장소가 없어서 4로 돌아갑니다.
- 같은 원리로, 4에서 1로 돌아가고, 1에서 0으로 돌아갑니다.
- 0은 시작점이므로, 0에서 더 방문할 장소가 없다는 것은 탐색이 종료됨을 의미합니다.
- 이렇게 시작점 0에서 갈 수 있는 모든 노드를 방문하였습니다.



DFS

- 이처럼 그래프에서 최대한 끝까지 탐색해보고, 탐색할 곳이 없으면 다시 돌아가 다른 루트를 탐색하는 방법을 DFS라 합니다. DFS(Depth First Search)는 일단 깊이 탐색하는 것을 중요시 여기기 때문이 이와 같은 이름으로 불립니다.
- DFS는 어떻게 구현할 수 있을까요? (그래프는 인접 리스트로 저장했다고 가정합니다.)
- visited[V]라는 배열을 만들어 i번 노드에 방문했으면 visited[i]에 1을, 그렇지 않다면 0을 저장해서 i번 노드 방문 여부를 저장합니다.
- 그렇다면 재귀 함수를 이용해 모든 노드를 탐색할 수 있습니다.

DFS

Code Explanation

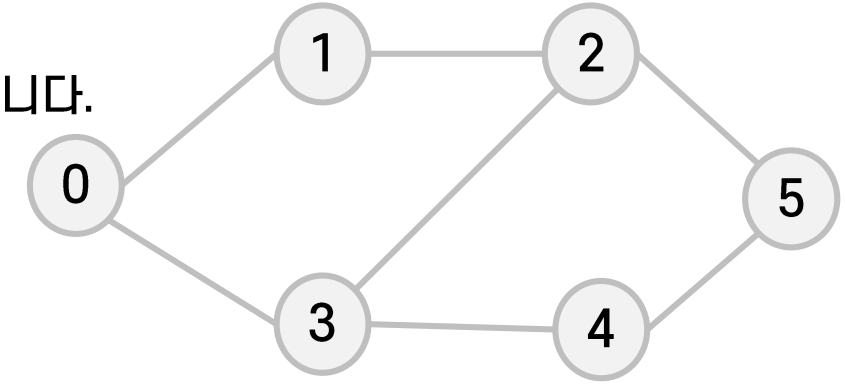
```
void DFS(int pos){
        visited[pos] = true; // pos 노드를 방문했다고 체크합니다.
        for(int i = 0; i < adj[pos].size(); i++){
                int next = adj[pos][i]; // next는 pos와 간선으로 연결된 노드입니다.
                if(!visited[next]){ // 만약 next를 아직 방문하지 않았다면
                        DFS(next); // next를 방문합니다.
                }
        }
}
```

DFS

- 만약 인접 리스트가 아닌 인접 배열로 저장을 했다면 어떻게 될까요? 인접 리스트를 사용하면 한 노드와 연결된 모든 노드를 볼 때, vector에 저장된 노드만 보면 되지만, 인접 배열을 사용하면 adj[pos][0] ~ adj[pos][V]까지 모든 노드를 봐야 하므로 간선의 수가 적다면 인접 리스트가 인접 배열보다 훨씬 효율적입니다.
- 위처럼 인접 배열보다는 인접 리스트가 더 효율적인 경우가 대부분이기 때문에 인접 리스트를 사용하여 그래프를 저장합니다.

BFS

- BFS는 DFS와 다르게 Breadth First Search, 즉 너비를 중요시하여 탐색합니다. BFS에서는 시작 노드와 가까운 노드부터 순차적으로 탐색하게 됩니다.
- 먼저 시작점 0을 탐색하여 1과 3이라는 노드를 찾고, 이 두 노드를 탐색할 목록에 넣습니다. (탐색 목록 : 1, 3)
- 탐색 목록의 가장 앞에 있는 1을 먼저 탐색하여 2라는 노드를 발견합니다. 이 2를 탐색 목록에 넣고, 1은 탐색을 진행했으므로 탐색 목록에서 제거합니다. (탐색 목록 : 3, 2)
- 탐색 목록의 가장 앞에 있는 3을 먼저 탐색하여 4라는 노드를 발견합니다. 마찬가지로 4를 탐색 목록에 넣고 3은 탐색 목록에서 제거합니다. (탐색 목록 : 2, 4)
- 같은 방법으로, 2에서 탐색을 진행하여 노드 4를 발견하고, 탐색 목록을 갱신합니다. (탐색 목록 : 4, 5)
- 4, 5에서는 더 이상 탐색할 노드가 없으므로 종료가 됩니다.



BFS

- BFS는 c++로 어떻게 구현할 수 있을까요? (그래프는 인접 리스트로 저장했다고 가정합니다.)
- 탐색 목록이 작동하는 방식을 보면, queue처럼 작동하는 것을 볼 수 있습니다.
- 따라서 BFS는 queue를 사용하여 구현하게 됩니다.

DFS

Code Explanation

```
void BFS(int pos){
        queue <int> q; // 탐색할 노드 목록을 저장할 queue를 정의합니다.
        q.push(start_pos); // 시작 노드를 queue에 추가합니다.
        visited[start_pos] = true; // start_pos를 탐색할 예정이라고 check합니다.
        while(!q.empty()){ // 더 이상 탐색할 노드가 없을 때까지 탐색을 진행합니다.
                int pos = q.front(); // 먼저 queue에서 탐색할 노드를 가져옵니다.
                q.pop(); // 탐색 목록에서 pos 노드를 제거합니다.
                for(int I = 0; I < adj[pos].size(); i++){
                int next = adj[pos][i]; // next는 pos와 간선으로 연결된 노드입니다.
                if(!visited[next]){ // 만약 next를 아직 탐색하지 않았고, 탐색할 예정도 없다면
                        q.push(next); // next를 탐색 목록에 추가합니다.
                }
        }
}
```