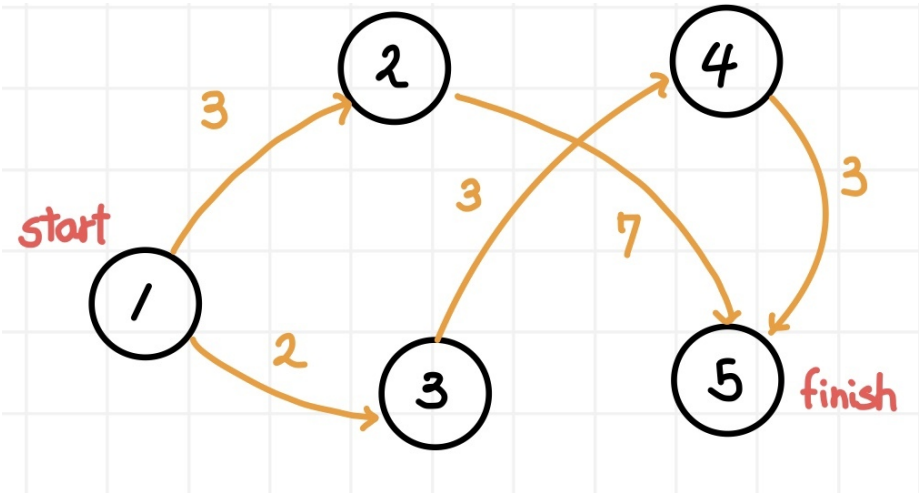
다익스트라 알고리즘 - Djikstra

길찾기 알고리즘 중 기초

다익스트라 알고리즘은 **길찾기 알고리즘**의 기초라고 생각합니다. 조건적인 부분들도 그렇고 이후 개념을 확장해 A* 알고리즘으로 넘어갈 수도 있습니다. 다익스트라 알고리즘은 **한 지점에서 다른 지점으로 가는 경로**를 찾는게 목적입니다.

다익스트라 알고리즘의 동작



위 그림에서 시작 지점이 1이고 도착 지점이 5라고 생각하고 알고리즘을 실행해 보겠습니다.

1. 일단 **시작 지점부터 각 지점까지의 거리를 저장할 배열을 초기화**해줍니다.

1	2	3	4	5
INF	INF	INF	INF	INF

2. 시작 지점의 비용을 0으로 계산해 줍니다. (시작 → 시작은 비용이 0이니)

1	2	3	4	5
0	INF	INF	INF	INF

3. 시작 지점과 연결된 지점을 방문합니다. (1→2, 1→3)

1	2	3	4	5
0	3	2	INF	INF

4. 방문한 지점은 2, 3번입니다. 이후 이 지점들과 연결된 지점들을 탐색합니다

   (2 → 5, 3 → 4)

1	2	3	4	5
0	3	2	5 (2 + 3)	10 (3 + 7)

   4번 지점의 경우 1 → 3 → 4 이므로 비용이 2 + 3 = 5가 되며 5번 지점은 1 → 2 → 5로 3 + 7 = 10이 됩니다.

5. 마지막으로 4번 지점과 연결된 곳을 탐색하고 종료합니다.

1	2	3	4	5
0	3	2	5	8 (5 + 3)

   여기서 기존의 5번 지점까지 가는데 비용은 10 (1 → 2 → 5)이었지만 4번을 거쳐 가는 방식이 더 빨랐으므로 (1 → 3 → 4 → 5 경로가 비용이 8) 더 빠른 경로의 비용으로 업데이트 해줍니다!

코드로 보는 다익스트라

처음 등장한 다익스트라는 시간복잡도가 $O(n^2)$이었습니다. (여기서 n은 정점)

```
int distFromStart[MAX];
bool visit[MAX];

void Djikstra(int start){
  // 시작 노드와 연결된 모든 정점들의 거리를 비교해 업데이트
  for(int i = 1; i <= 정점의 수; i++)
    distFromStart[i] = edge[start][i];

  // 시작 노드 방문 처리
  visit[start] = true;

  for(정점의 수만큼 반복)
  {
    int 다음 정점 = (방문한 정점들 중 가장 빠른 정점 탐색하는 함수);
    여기서 다음 정점에 대한 distFromStart 배열 수정
  }
}
```

(평소 잘 사용하지 않는 코드라 의사코드로 표현했습니다)

일단 지금 코드의 경우 제곱승으로 동작하는 함수라 생각보다 빠르지 않습니다. 처음 알고리즘이 고안되었을때는 이런 형태를 하고 있었는데 이후 `최소 힙`을 사용해 이 시간복잡도를 $O((V+E)logV)$로 줄인 형태가 되었다고 합니다.

> 여기서 V는 vertex(=정점의수)이며 E는 edge(=간선의 수)

C++의 경우 `queue` STL에 있는 `priority_queue<T> (우선순위 큐)`를 사용해서 구현하게 됩니다.

> 💡 **우선순위 큐** : 기본적으로 **최대 힙**으로 동작하는 자료구조

```
#include <iostream>
#include <queue>
#include <vector>
using namespace std;

// arr은 문제의 시작 지점이 인덱스로 가는데까지 비용을 저장
int arr[100];

// 입력에서 인덱스가 출발지가 되고 <도착지, 비용> 순으로 받아오게 됩니다.
vector<pair<int, int>> v[100];

void Djikstra(int start){
    // 큐에는 <시간, 출발지> 순으로 데이터가 들어가게 됩니다.
    // 때문에 우선순위 큐는 시간에 따라서 우선순위를 만들어 줍니다.
    priority_queue<pair<int, int>> q;

    arr[start] = 0;
    q.push({0, start});

    while(!q.empty()){
        int cur = q.top().second;
        int time = -q.top().first;
        q.pop();

        // 혹시 현재 비용이 이전의 비용보다 크다면(불필요한 값) 이번 정점은 탐색 끝
        if(time > arr[cur])
            continue;

        for(int i = 0; i < v[cur].size(); i++){
            int nextCur = v[cur][i].first;
            int nextTime = v[cur][i].second;

            if(arr[nextCur] > time + nextTime){
                arr[nextCur] = time + nextTime;
                q.push({-arr[nextCur], nextCur});
            }
        }
    }
}
```

이 형태라면 이전에 알아본 `BFS` 알고리즘과 비슷한 형태로 동작하며 시간도 빠르고 구현해야할 함수도 줄어들게 됩니다.

여기서 주목할 것은 우선순위 큐에서 **참조할때와 삽입할 때** 입니다. 위 알고리즘에서 핵심은 **'가장 짧은 곳'**을 탐색하기 위함인데 `우선순위 큐`는 기본적으로 '최대 힙'이기 때문에 '가장 비용이 큰 지점'이 나오게 됩니다. 때문에 부호를 바꿔서 넣음으로서 '최소 힙'으로 동작하게 하는 것이죠.

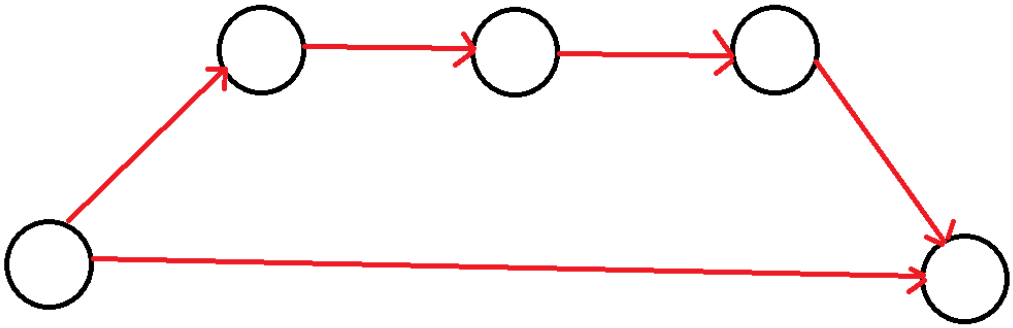
혹시나 위 - 방식이 마음에 들지 않는다면

```
priority_queue<pair<int, int>, vector<pair<int, int>>, greater<pair<int, int>> q;
```

로 하면 오름차순 정렬이 됩니다. (작은 값이 top에 있는 최소힙)

다익스트라 알고리즘의 특성

- 다익스트라 알고리즘은 **음의 가중치가 없을때** 사용하게 됩니다.
  - 음의 가중치가 있다면 사이클(무한루프)이 형성되거나 정확한 값이 나올 수 없게 됩니다.
  - '그럼 제일 작은 가중치가 1이 되도록 다 같은 값을 더해주면 되는거 아닌가?' 라고 생각할 수있는데 이 방법도 안됩니다.



    이런 상황에서 예외가 나오게 되니 참고하시면 됩니다.

    (모든 경우가 아닌 음의 가중치가 있으면 값이 나오지 않을 **가능성이 있다** 로 이해하시면 좋습니다)
  - 만약 음의 가중치가 있는 경우 `벨만-포드 알고리즘`을 사용해 풀이하게 됩니다.
- 시작지점 1개에 대해서 판별하는 알고리즘입니다.
  - 만일 `from 모든 지점 to 모든 지점` 이 필요하다면 알고리즘을 노드 수만큼 반복하면 됩니다.
  - 또는 `플로이드-와샬 알고리즘`이 있는데 시간 복잡도는 $O(n^3)$이며 비슷한 시간에, 짧은 코드로 탐색할 수 있습니다.

추천 문제

14284번: 간선 이어가기 2

정점 n개, 0개의 간선으로 이루어진 무방향 그래프가 주어진다. 그리고 m개의 가중치 간선의 정보가 있는 간선리스트가 주어진다. 간선리스트에 있는 간선 하나씩 그래프에 추가해 나갈 것이다. 이때, 특정 정점 s와 t가 연결

https://www.acmicpc.net/problem/14284

5972번: 택배 배송

농부 현서에게는 지도가 있습니다. N (1 <= N <= 50,000) 개의 헛간과, 소들의 길인 M (1 <= M <= 50,000) 개의 양방향 길이 그려져 있고, 각각의 길은 C_i (0 <= C_i <= 1,000) 마리의 소가 있습니다. 소들의 길은 두 개의

https://www.acmicpc.net/problem/5972

14938번: 서강그라운드

예은이는 요즘 가장 인기가 있는 게임 서강그라운드를 즐기고 있다. 서강그라운드는 여러 지역중 하나의 지역에 낙하산을 타고 낙하하여, 그 지역에 떨어져 있는 아이템들을 이용해 서바이벌을 하는 게임이다. 서강그라운드

https://www.acmicpc.net/problem/14938