#고급반 1주차

분리 집합 & 최소 스패닝 트리

<Disjoint Set & Minimum Spanning Tree>

#그래프와 트리

<Graphs & Trees> - 용어

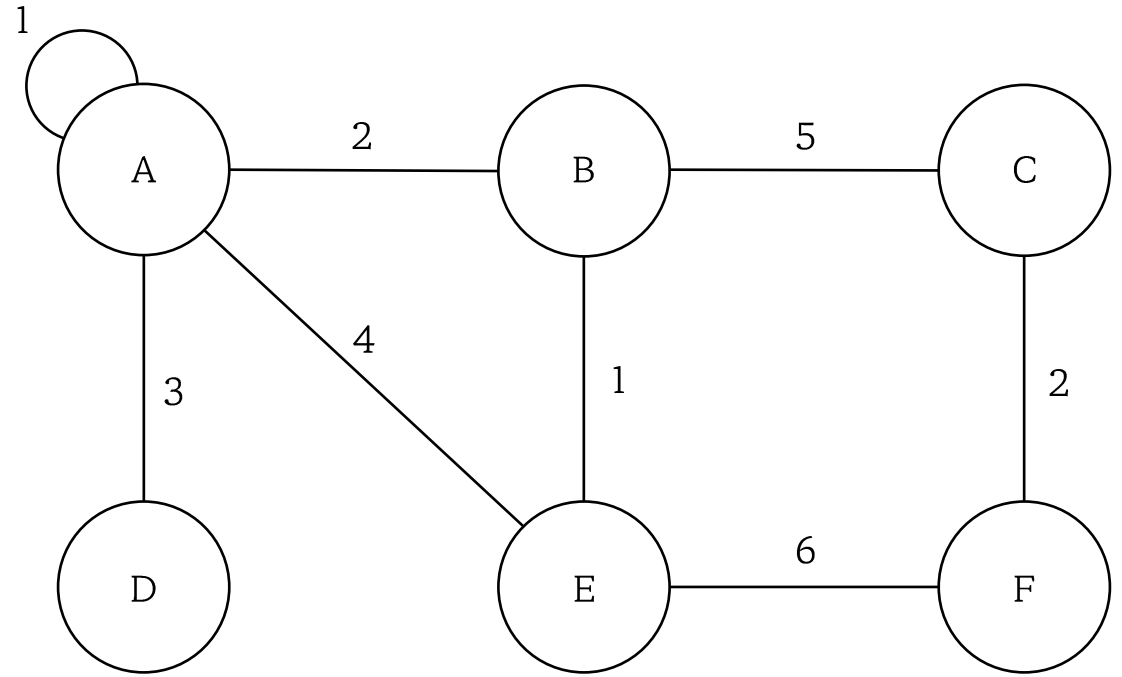
정점 : 그래프를 이루는 객체(꼭짓점)
vertices(nodes)

간선 : 정점을 연결하는 선(변)
edges

루프 : 시작과 끝이 같은 간선
loop

가중치 : 그래프의 간선에 붙는 수치
weight

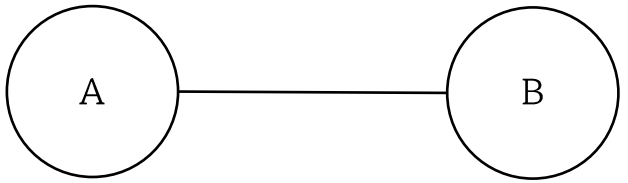
그래프 : 정점과 간선으로 이루어진 구조
graphs



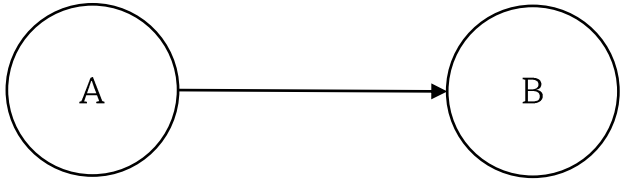
#그래프와 트리

<Graphs & Trees> - 용어

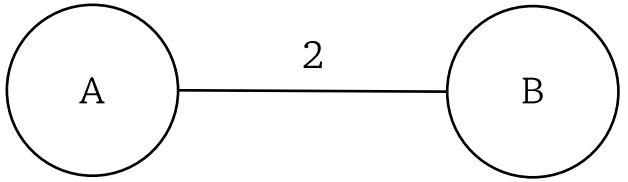
양방향 그래프 : 간선의 방향이 없는 그래프
bidirected graph(undirected graph)



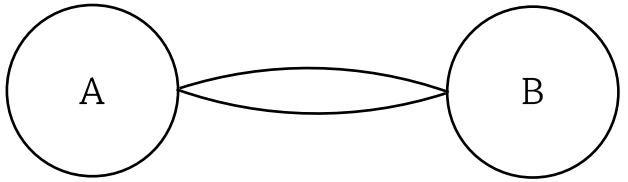
단방향 그래프 : 간선의 방향이 있는 그래프
directed graph(digraph)



가중치 그래프 : 간선에 가중치가 붙은 그래프
weighted graph



다중 그래프 : 두 정점을 잇는 간선이 여러 개일 수 있는 그래프
multigraph



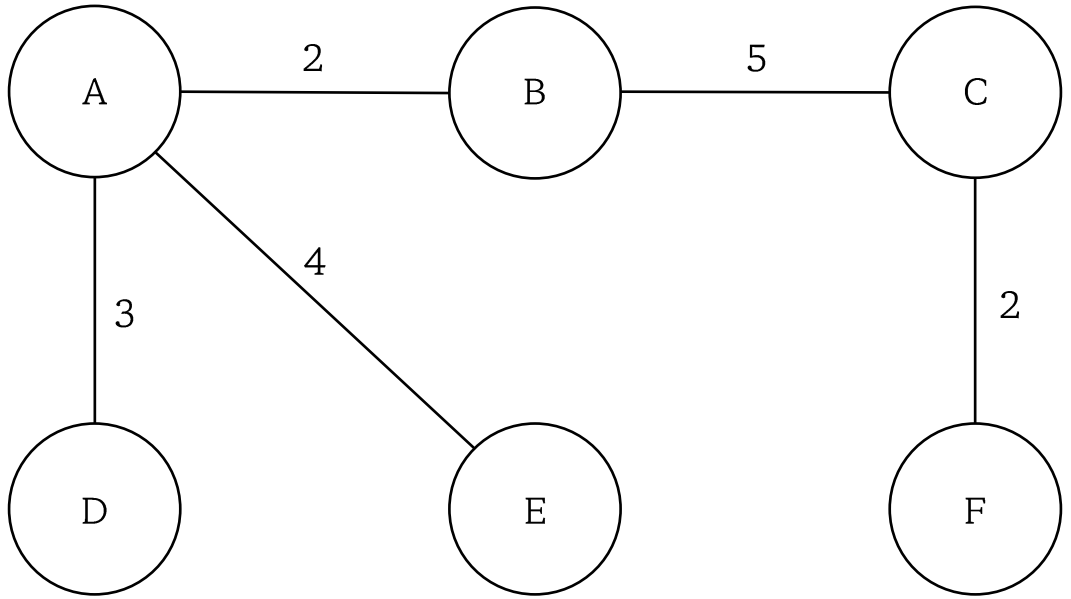
#그래프와 트리

<Graphs & Trees> - 용어

사이클 : 중복된 간선 없이 같은 정점으로 돌아오는 경로
path

트리 : 사이클이 없는 그래프
cycle

스패닝 트리 : 그래프 내 모든 정점을 포함하는 트리
spanning tree



#그래프와 트리

<Graphs & Trees> - 용어

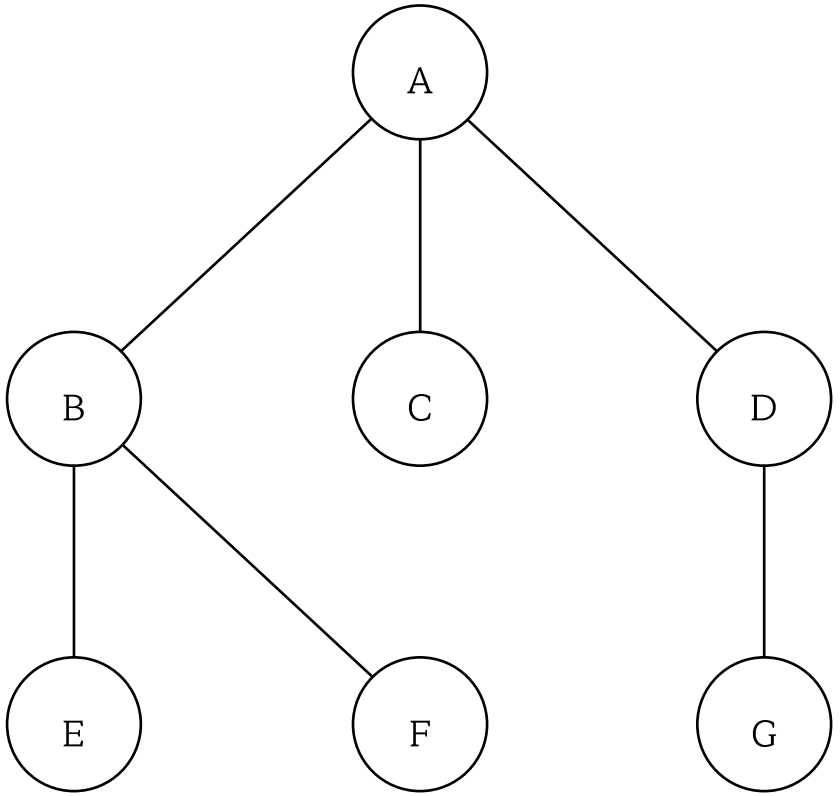
부모/자식 노드 : 어떤 노드의 상위/하위 노드
parent/child

루트 노드 : 부모 노드가 없는 최상위 노드
root node

단말 노드 : 자식 노드가 없는 최하위 노드
leaf node

내부 노드 : 루트 노드와 단말 노드 사이의 노드
internal node

형제 : 같은 부모를 가지는 노드
siblings



#분리 집합

<Disjoint set> - 소개

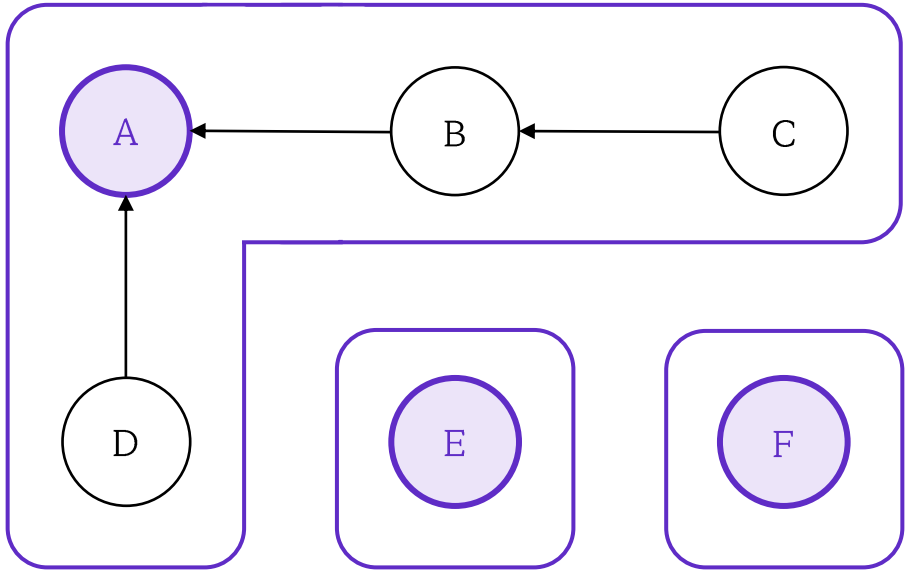
분리 집합 : 겹치는 부분이 없도록 모든 원소를 분리한 부분집합

Disjoint set

분리 집합을 구현하기 위해 두 함수가 필요합니다.

1. Union : 두 집합을 합치기
2. Find : 루트 노드 찾기

두 함수의 이름을 따서 Union-Find 알고리즘이라고 합니다.
하지만, Union에 Find가 필요하기 때문에 Find부터 알아볼 것입니다.



#분리 집합

<Disjoint set> - Find

Find : 어떤 노드의 루트 노드 찾기

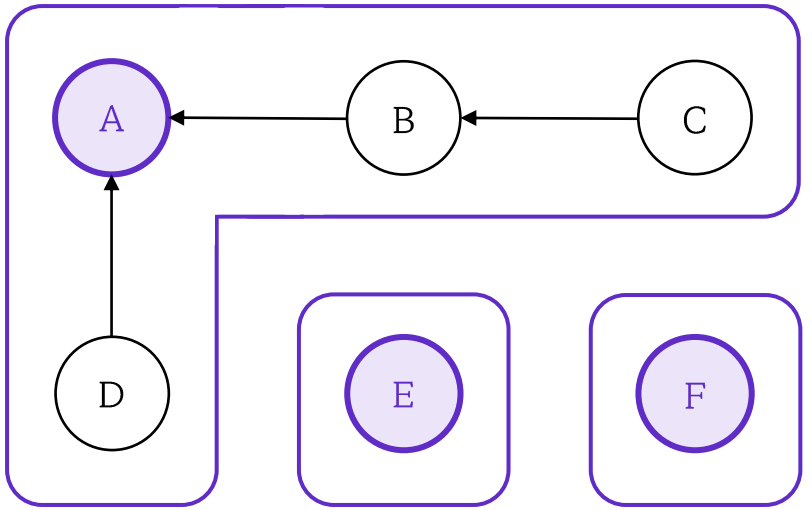
Finding root node

먼저, 배열을 하나 만들어야 합니다

배열 “Parent”는 부모 노드를 저장합니다.

만약 부모 노드가 없다면 자신을 저장합니다.

이 배열은 초기에는 모든 정점이 자기 자신을 저장해야 합니다.



<Parent>

Index	Value
A	A
B	A
C	B
D	A
E	E
F	F

#분리 집합

<Disjoint set> - Find

Find : 어떤 노드의 루트 노드 찾기

Finding root node

Find 함수는 재귀함수로 작성할 수 있습니다.

1. Index == Value

이 경우 그 노드가 루트 노드임을 의미합니다.

따라서, Index값을 return합니다.

2. Index != Value

이 경우 부모 노드인 Value값에서 다시 Find합니다.

#분리 집합

<Disjoint set> - Find

설명을 코드로 옮기면 다음과 같습니다.

init은 초기값을 지정하는 함수입니다.

find는 node의 루트 노드를 찾는 함수입니다.

```
const int NODE_CNT = 10000;

int parent[NODE_CNT];

void init() {
    for (int i = 0; i < NODE_CNT; i++) {
        parent[i] = i; // 모두 자기 자신을 저장
    }
}

int find(int node) {
    if (parent[node] == node) return node;
    else return find(parent[node]);
}
```

#분리 집합

<Disjoint set> - Union

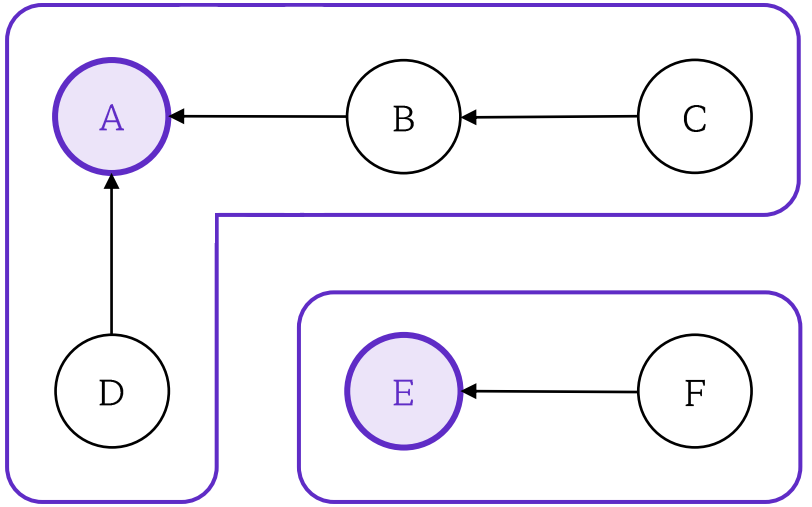
Union : 두 집합을 합치기

Merging two disjoint sets

F와 C를 합치는 상황을 살펴보겠습니다.

여러 가지 방법을 떠올릴 수 있지만,

가장 직관적인 방법부터 실행해보겠습니다.



<Parent>

Index	Value
A	A
B	A
C	B
D	A
E	E
F	E

#분리 집합

<Disjoint set> - Union

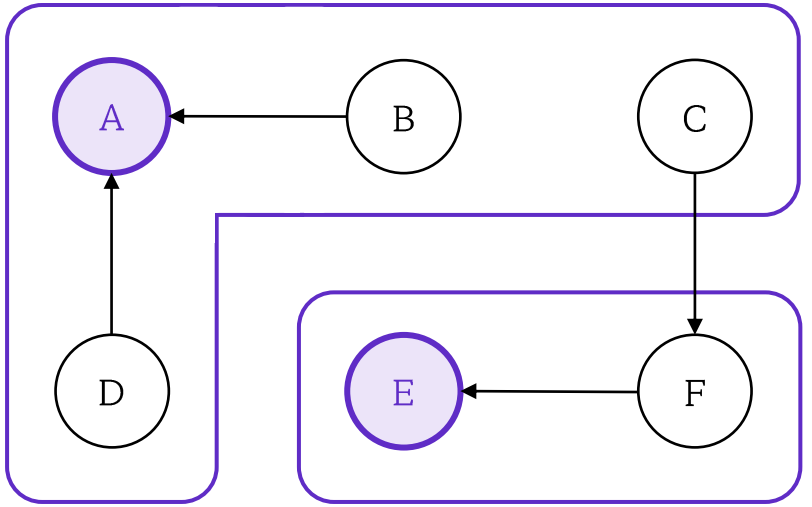
Union : 두 집합을 합치기

Merging two disjoint sets

1. parent[C] = F로 바꾸기

이 방법은 C-F를 연결해주지만,

C-B를 끊어서 ABCD에서 C가 없어집니다.



<Parent>

Index	Value
A	A
B	A
C	F
D	A
E	E
F	E

#분리 집합

<Disjoint set> - Union

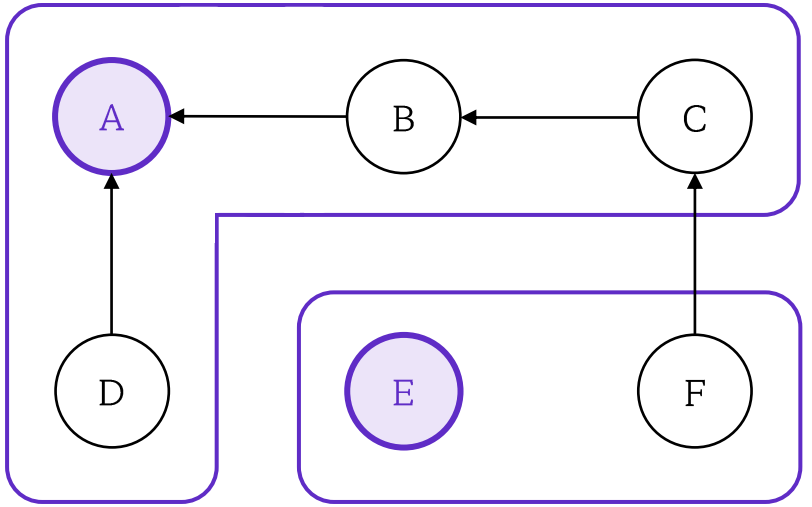
Union : 두 집합을 합치기

Merging two disjoint sets

2. parent[F] = C로 바꾸기

이 방법은 C-F를 연결해주지만,

E-F를 끊어서 EF에서 F가 없어집니다.



<Parent>

Index	Value
A	A
B	A
C	B
D	A
E	E
F	C

#분리 집합

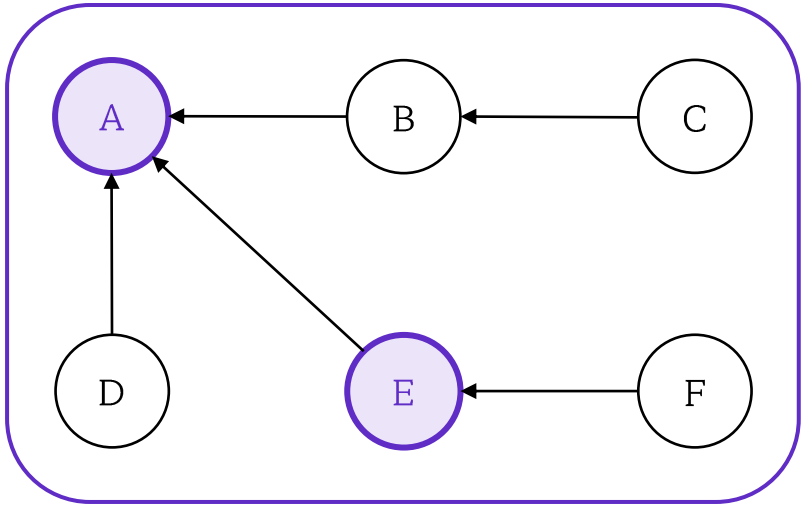
<Disjoint set> - Union

Union : 두 집합을 합치기

Merging two disjoint sets

3. 루트 노드끼리 연결하기

이 방법은 기존 자신을 저장하던
루트 노드의 정보를 변경하기 때문에
정상적으로 두 집합을 합칠 수 있습니다.



<Parent>

Index	Value
A	A
B	A
C	B
D	A
E	A
F	C

#분리 집합

<Disjoint set> - Union

설명을 코드로 옮기면 다음과 같습니다.

노드 x와 노드 y를 합치는 함수입니다.

만약, 루트 노드가 같다면, 바로 return합니다.

다르다면, 루트 노드끼리 연결합니다.

“union”은 예약어입니다.

따라서, 함수 이름으로 “merge”를 대신 사용했습니다.

```
void merge(int x, int y) {
    int ans_x = find(x);
    int ans_y = find(y);

    if (ans_x == ans_y) return;
    else parent[ans_x] = ans_y;
}
```

#분리 집합

<Disjoint set> - 최적화

최적화 : 시간복잡도 줄이기

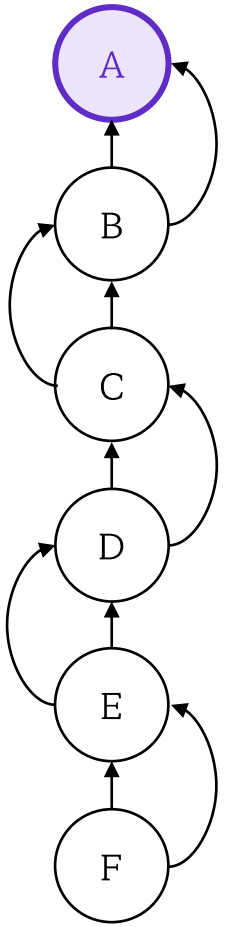
Optimization

오른쪽의 선형 그래프를 생각해봅시다.

Find(F)를 호출하면 함수가 총 5번 호출됩니다.

즉, (그래프의 깊이) - 1만큼 함수가 호출됩니다.

이런 상황에서, 속도를 더 빠르게 할 수 있을까요?



<Parent>

Index	Value
A	A
B	A
C	B
D	C
E	D
F	E

#분리 집합

<Disjoint set> - 최적화

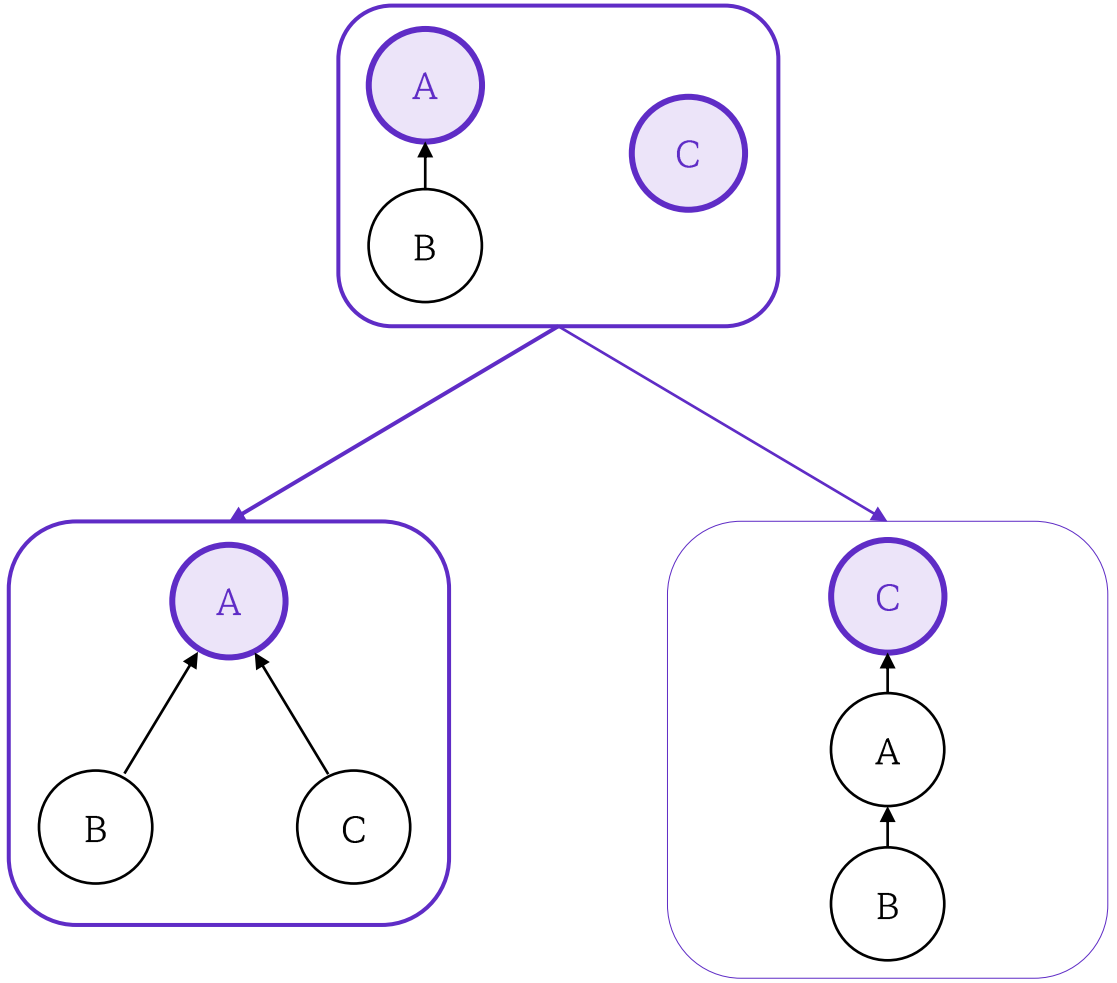
최적화 : 시간복잡도 줄이기

Optimization

오른쪽 그림에서 AB와 C를 합치는 방법은

깊이가 더 깊어지지 않는 왼쪽이 더 좋습니다.

따라서, 각각의 노드에서 트리의 깊이를 저장해봅시다.



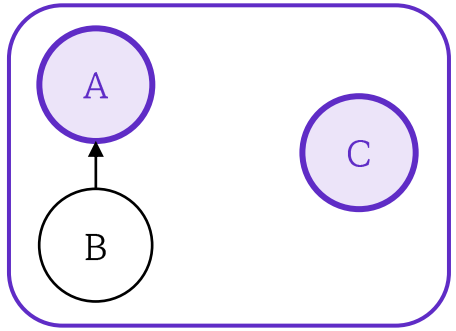
#분리 집합

<Disjoint set> - 최적화

최적화 : 시간복잡도 줄이기

Optimization

배열 이름은 “rank”라고 짓겠습니다.
합칠 때는 rank가 큰 쪽으로 합쳐야
깊이가 더 늘어나지 않습니다.
만약, rank가 같다면 합치는 순서는 상관이 없고
합친 쪽의 rank가 1 늘어납니다.
이렇게 각 노드의 rank를 저장하는 방식을
"Union by Rank"라고 합니다.



<rank>

Index	Value
A	2
B	1
C	1

#분리 집합

<Disjoint set> - 최적화

최적화 : 시간복잡도 줄이기
Optimization

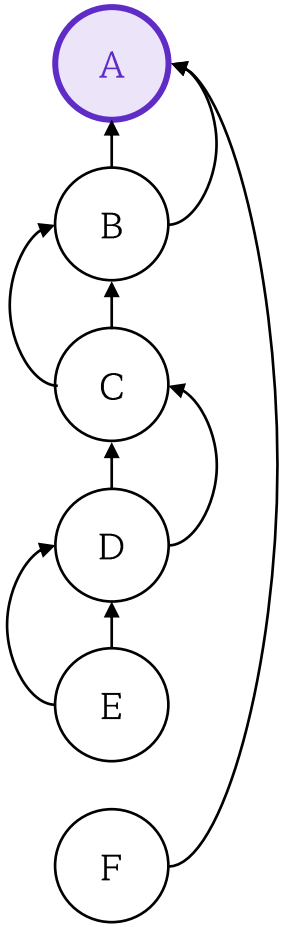
두 번째 방법은 “경로 압축”입니다.

Find 함수를 호출하면 루트 노드로

parent를 바꾸는 방법입니다.

오른쪽의 예시는 Find(F)를 호출한 경우입니다.

그러면, 어떻게 parent[F]에 A를 저장할 수 있을까요?



<Parent>

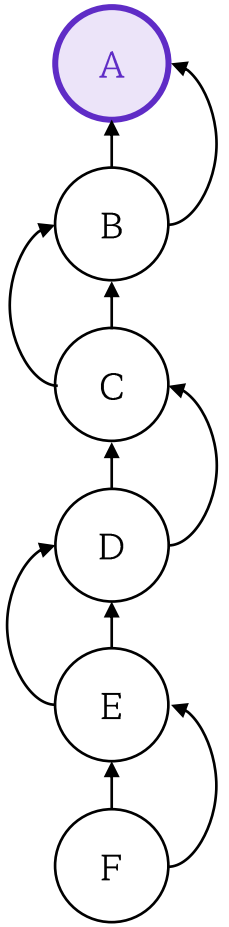
Index	Value
A	A
B	A
C	B
D	C
E	D
F	A?

#분리 집합

<Disjoint set> - 최적화

최적화 : 시간복잡도 줄이기
Optimization

Find(F)를 호출하면서,
"parent[F] = Find(E)"를 return합니다.
이때, 다음으로 Find(E)가 호출됩니다.
재귀적으로 Find(A)까지 호출되면, A를 return합니다.



<Stack>

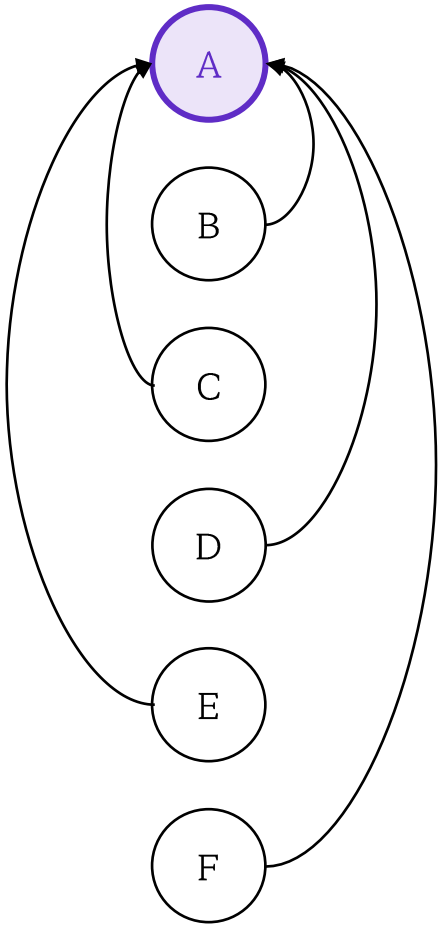
return A
parent[B] = Find(A)
parent[C] = Find(B)
parent[D] = Find(C)
parent[E] = Find(D)
parent[F] = Find(E)

#분리 집합

<Disjoint set> - 최적화

최적화 : 시간복잡도 줄이기
Optimization

이후, return된 A가 parent[B]의 값을 바꿉니다.
또한, Find(B)에서 A가 return되고,
parent[C]의 값을 바꿉니다.
이를 계속 반복합니다.



<Stack>

parent[B] = A
parent[C] = Find(B)
parent[D] = Find(C)
parent[E] = Find(D)
parent[F] = Find(E)

#분리 집합

<Disjoint set> - 최적화

최적화 : 시간복잡도 줄이기

Optimization

두 최적화를 코드로 옮기면 다음과 같습니다.

시간복잡도는 다음과 같습니다.

최적화 없는 코드 : $O(N)$

하나만 적용 : $O(\log N)$

둘 다 적용 : $O(\alpha(N))$

$O(\alpha(N))$에서 $\alpha(N)$은 akermann 함수의 역함수입니다.

"사실상 상수"라고만 이해해주시면 됩니다.

```
int depth[NODE_CNT];

int find(int node) {
    if (parent[node] == node) return node;
    else return parent[node] = find(parent[node]); //L-value 반환
}

void merge(int x, int y) {
    int ans_x = find(x);
    int ans_y = find(y);

    if (ans_x == ans_y) return;
    else {
        if (rank[x] > rank[y]) parent[ans_y] = ans_x;
        else if (rank[x] < rank[y]) parent[ans_x] = ans_y;
        else {
            parent[ans_y] = ans_x;
            rank[ans_x]++;
        }
    }
}
```

#분리 집합

<Disjoint set> - 연습 문제

집합의 표현(BOJ #1717)

<문제 설명>

- 초기에 n+1개의 집합이 있다.
- 이 집합에 합집합 연산을 한다.
- 합집합 연산 중간에 두 원소가 같은 집합인지 물어본다.
- 이때, 두 원소가 같은 집합에 있는지를 YES/NO로 출력한다.

<제약 조건>

- $1 \leq n \leq 1{,}000{,}000$
- $1 \leq m \leq 100{,}000$

#분리 집합

<Disjoint set> - 연습 문제

4 집합의 표현(BOJ #1717)

<문제 해설>

문제에서의 “합집합” 연산은 “Union”, “같은 집합에 속해있는지 확인”하는 연산은 “Find”를 의미합니다.

따라서, 기본적인 Disjoint Set을 구현한 후, 들어오는 입력에 맞게 Union 또는 Find를 실행하면 됩니다.

#분리 집합

<Disjoint set> - 연습 문제

두 연산을 구현하면 다음과 같습니다.

0이 들어오면 merge를 그대로 쓰면 됩니다.

1이 들어오면 find의 값을 비교해줍니다.

```
void query0(int a, int b) {
    merge(a, b);
    return;
}

void query1(int a, int b) {
    cout << ((find(a) == find(b)) ? "YES" : "NO");
    cout << '\n';
    return;
}
```

#최소 스패닝 트리

<Minimum Spanning Tree> - 소개

최소 스패닝 트리 : 가중치 그래프에서 간선의 가중치 합이 가장 작은 스패닝 트리

Minimum Spanning Tree(MST)

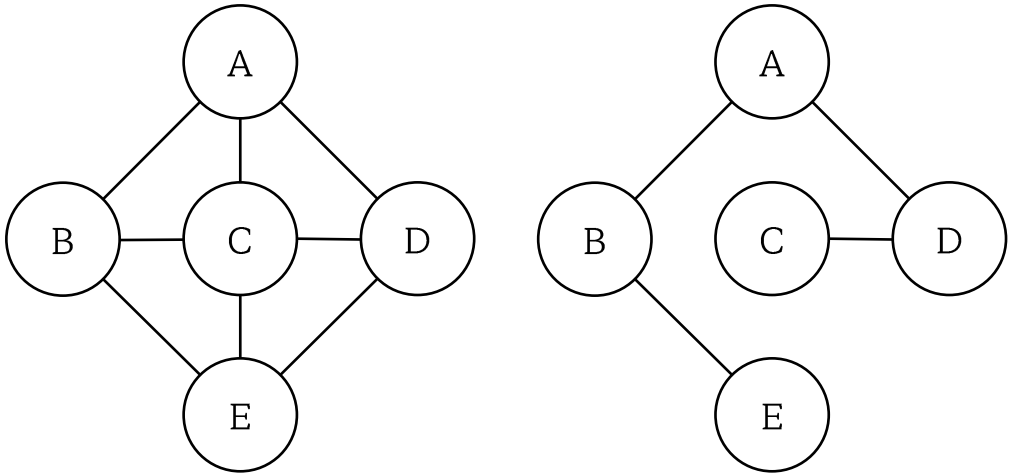
먼저, 스패닝 트리란 그래프의 모든 정점을 포함하는 트리입니다.

오른쪽의 예시에서는, A, B, C, D, E로 이루어진 그래프에서

A-C, B-C, C-E, E-D를 제거해 간선을 4개만 남겼습니다.

트리에서 간선의 개수는 (정점의 개수 – 1)개입니다.

따라서, 간선을 더 지우면 그래프가 분리될 것입니다.



#최소 스패닝 트리

<Minimum Spanning Tree> - 소개

최소 스패닝 트리 : 가중치 그래프에서 간선의 가중치 합이 가장 작은 스패닝 트리

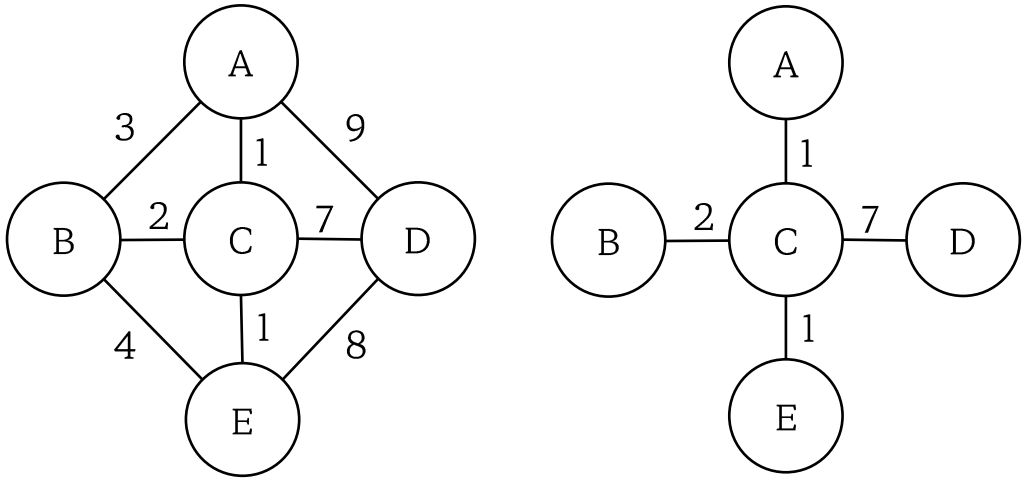
Minimum Spanning Tree(MST)

최소 스패닝 트리를 구하는 대표적인 두가지 알고리즘으로,

1. Kruskal's Algorithm
2. Prim's Algorithm

이 있습니다.

각각에 대해 알아보겠습니다.



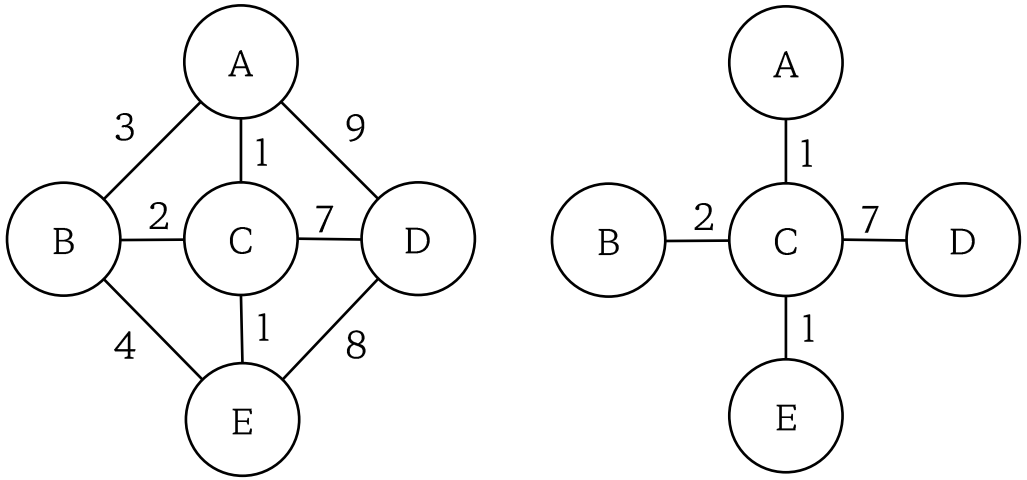
#최소 스패닝 트리

<Minimum Spanning Tree> - 소개

최소 스패닝 트리 : 가중치 그래프에서 간선의 가중치 합이 가장 작은 스패닝 트리

Minimum Spanning Tree(MST)

두 알고리즘은 공통적으로 그리디 알고리즘을 이용합니다.
정확히는, 두 알고리즘은 "최소가 되는 간선을 먼저 쓴다"
라는 아이디어를 사용합니다.
오른쪽의 MST 역시 1, 2, 1, 7 등 가중치가 작은 간선만을
활용하여 구성된 것을 확인할 수 있습니다.



#최소 스패닝 트리

<Minimum Spanning Tree> - Kruskal's Algorithm

최소 스패닝 트리 : 가중치 그래프에서 간선의 가중치 합이 가장 작은 스패닝 트리

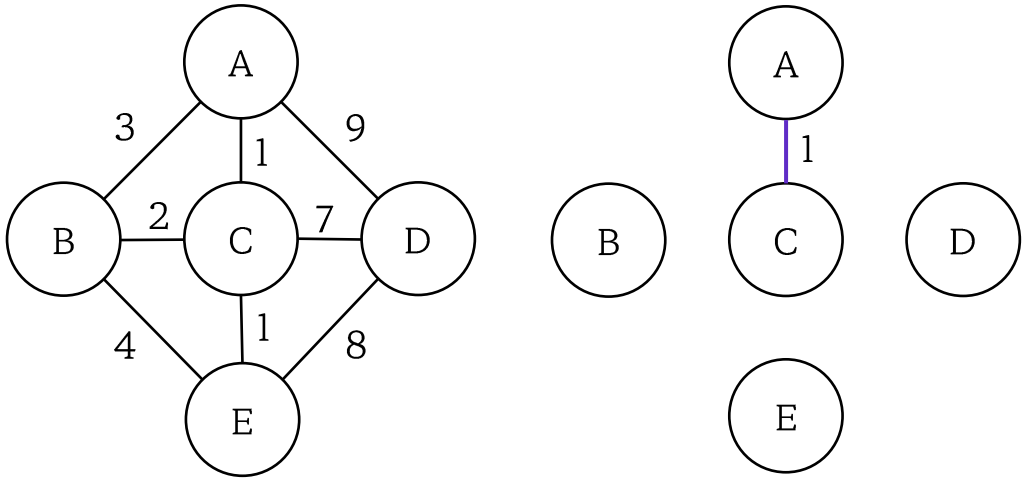
Minimum Spanning Tree(MST)

먼저, Kruskal's Algorithm부터 알아보겠습니다.

Kruskal's Algorithm은 다음의 과정을 거칩니다.

1. 남아있는 간선 중 가장 가중치가 작은 간선 선택
2. 그 간선이 의미가 있는 간선이라면 추가
3. 간선을 다 확인할 때까지 반복

여기에서 "의미가 있는 간선"이 무슨 의미일까요?



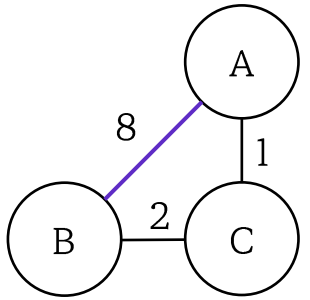
#최소 스패닝 트리

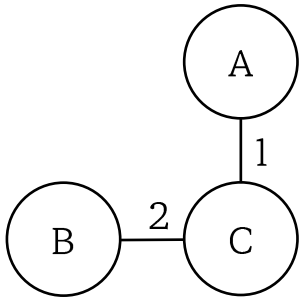
<Minimum Spanning Tree> - Kruskal's Algorithm

최소 스패닝 트리 : 가중치 그래프에서 간선의 가중치 합이 가장 작은 스패닝 트리

Minimum Spanning Tree(MST)

간선을 추가할 때, 이미 연결되어 있다면
간선을 더 추가할 필요가 없어집니다.
즉, 간선 추가 여부를 확인하기 위해
Disjoint Set을 사용해야 합니다.





#최소 스패닝 트리

<Minimum Spanning Tree> - Kruskal's Algorithm

최소 스패닝 트리 : 가중치 그래프에서 간선의 가중치 합이 가장 작은 스패닝 트리

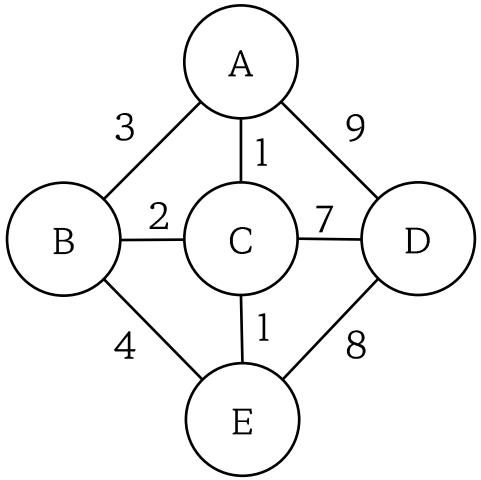
Minimum Spanning Tree(MST)

그러면, Kruskal's Algorithm을 이용하여

오른쪽 그래프의 MST를 구해보겠습니다.

먼저, 간선을 가중치에 따라 오름차순으로 정렬합니다.

가중치가 같은 간선끼리는 순서가 상관이 없습니다.



<Parent>

Edge	Weight
A - C	1
C - E	1
B - C	2
A - B	3
B - E	4
C - D	7
D - E	8
A - D	9

#최소 스패닝 트리

<Minimum Spanning Tree> - Kruskal's Algorithm

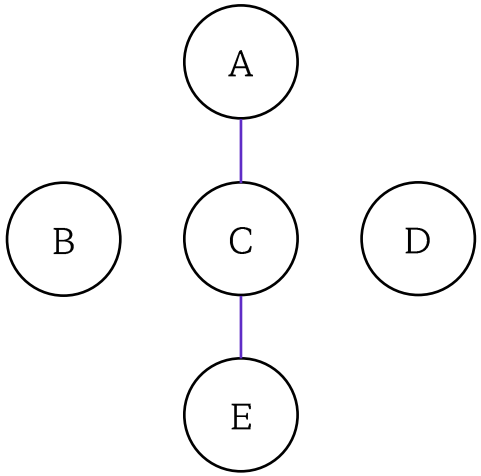
최소 스패닝 트리 : 가중치 그래프에서 간선의 가중치 합이 가장 작은 스패닝 트리

Minimum Spanning Tree(MST)

다음으로, 가중치가 작은 간선부터 추가합니다.

만약, 간선 양쪽의 두 정점이 이미 연결되어 있다면

간선을 추가하지 않습니다.



<Parent>

Edge	Weight
A - C	1
C - E	1
B - C	2
A - B	3
B - E	4
C - D	7
D - E	8
A - D	9

#최소 스패닝 트리

<Minimum Spanning Tree> - Kruskal's Algorithm

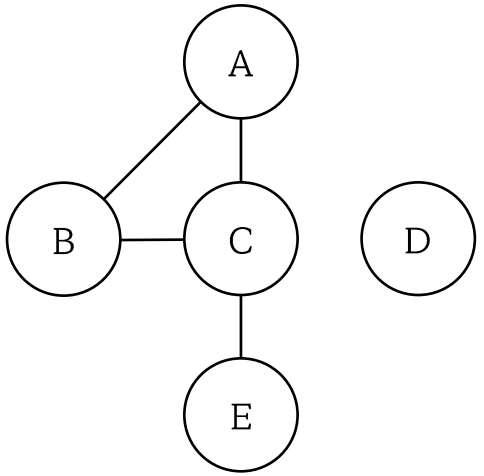
최소 스패닝 트리 : 가중치 그래프에서 간선의 가중치 합이 가장 작은 스패닝 트리

Minimum Spanning Tree(MST)

오른쪽은 A-B가 추가될 지 확인하는 경우입니다.

A, B는 A-C-B로 이미 연결되어 있습니다.

따라서, A-B를 추가하지 않습니다.



<Parent>

Edge	Weight
A - C	1
C - E	1
B - C	2
A - B	3
B - E	4
C - D	7
D - E	8
A - D	9

#최소 스패닝 트리

<Minimum Spanning Tree> - Kruskal's Algorithm

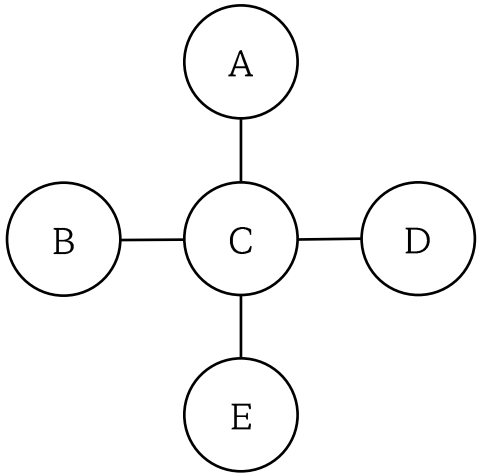
최소 스패닝 트리 : 가중치 그래프에서 간선의 가중치 합이 가장 작은 스패닝 트리

Minimum Spanning Tree(MST)

이 방법에 따라 A-B까지 확인한 후,

추가한 간선들만 모으게 되면 최소 스패닝 트리가 됩니다.

이제 구현 방법을 연습문제를 통해 알아보겠습니다.



<Parent>

Edge	Weight
A - C	1
C - E	1
B - C	2
A - B	3
B - E	4
C - D	7
D - E	8
A - D	9

#최소 스패닝 트리

<Minimum Spanning Tree> - Kruskal's Algorithm

최소 스패닝 트리(BOJ #1753)

<문제 설명>

- 그래프가 주어진다.
- 그래프에서 최소 스패닝 트리의 가중치를 구하기

<제약 조건>

- $1 \leq V \leq 10,000$
- $1 \leq E \leq 100,000$
- 답은 int 범위 내

#최소 스패닝 트리

<Minimum Spanning Tree> - Kruskal's Algorithm

먼저, union함수를 살짝 바꿔보겠습니다.

return값을 bool로 설정하고,

두 노드가 연결되어 있었다면 false,

그렇지 않았다면 true를 return하게 합니다.

```
bool merge(int x, int y) {
  int ans_x = find(x);
  int ans_y = find(y);
  if (ans_x == ans_y) return false;
  else {
    if (depth[x] > depth[y]) parent[ans_y] = ans_x;
    else if (depth[x] < depth[y]) parent[ans_x] = ans_y;
    else {
      parent[ans_y] = ans_x;
      depth[ans_x]++;
    }
    return true;
  }
}
```

#최소 스패닝 트리

<Minimum Spanning Tree> - Kruskal's Algorithm

다음으로, main함수입니다.

먼저, 간선을 입력받을 때,

가중치의 오름차순으로 정렬하여 저장하기 위해

priority_queue와 greater을 사용합니다.

저는 편의상 템플릿으로 priority_stack을 미리 만들어서 사용합니다.

```cpp
template<typename T>
using priority_stack = priority_queue<T, vector<T>, greater<T>>;

signed main() {
    ios_base::sync_with_stdio(0); init();
    cin.tie(NULL); cout.tie(NULL);

  int v; int e;
  cin >> v >> e;
  priority_stack<pair<int, pair<int, int>>> edges;
  while (e--) {
    int a, b, c;
    cin >> a >> b >> c;
    edges.push(mp(c, mp(a, b)));                    가중치부터 저장해줍니다
  }
  int ans = 0;
  while (!edges.empty()) {
    ans += edges.top().first *
      merge(edges.top().second.first,
        edges.top().second.second);
    edges.pop();
  }
  cout << ans;
}
```

#최소 스패닝 트리

<Minimum Spanning Tree> - Kruskal's Algorithm

그 다음, 간선을 하나씩 추가합니다.

추가를 위해서는 union 함수를 사용합니다.

만약 union함수가 true(1)을 return했다면,

답에 간선의 가중치를 추가합니다.

false(0)을 return했다면 추가하지 않습니다.

```cpp
template<typename T>
using priority_stack = priority_queue<T, vector<T>, greater<T>>;

signed main() {
    ios_base::sync_with_stdio(0); init();
    cin.tie(NULL); cout.tie(NULL);

  int v; int e;
  cin >> v >> e;
  priority_stack<pair<int, pair<int, int>>> edges;
  while (e--) {
    int a, b, c;
    cin >> a >> b >> c;
    edges.push(mp(c, mp(a, b)));
  }
  int ans = 0;
  while (!edges.empty()) {
    ans += edges.top().first *
      merge(edges.top().second.first,
        edges.top().second.second);
    edges.pop();
  }
  cout << ans;
}
```

true = 1, false = 0이므로
조건문 대신 곱해줘도 됩니다.

#최소 스패닝 트리

<Minimum Spanning Tree> - Prim's Algorithm

최소 스패닝 트리 : 가중치 그래프에서 간선의 가중치 합이 가장 작은 스패닝 트리

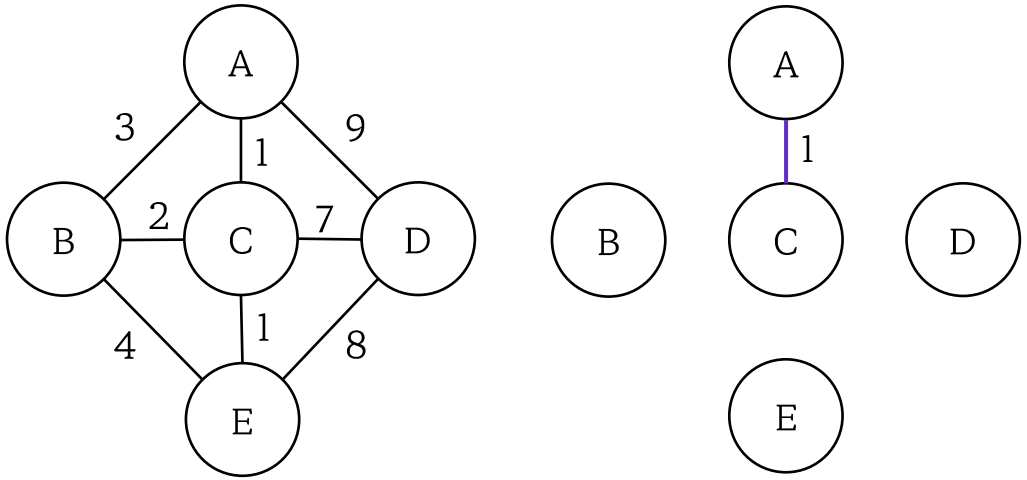
Minimum Spanning Tree(MST)

다음으로, Prim's Algorithm에 대해 알아보겠습니다.

Prim's Algorithm은 다음의 과정을 거칩니다.

1. 임의의 정점 추가
2. 이미 연결된 정점에서 벋어나간 다른 간선 확인
3. 확인한 간선 중 가장 작은 간선 추가
4. 트리를 이룰 때까지 2~3을 반복

앞서 보았던 Kruskal's Algorithm과 거의 비슷합니다.



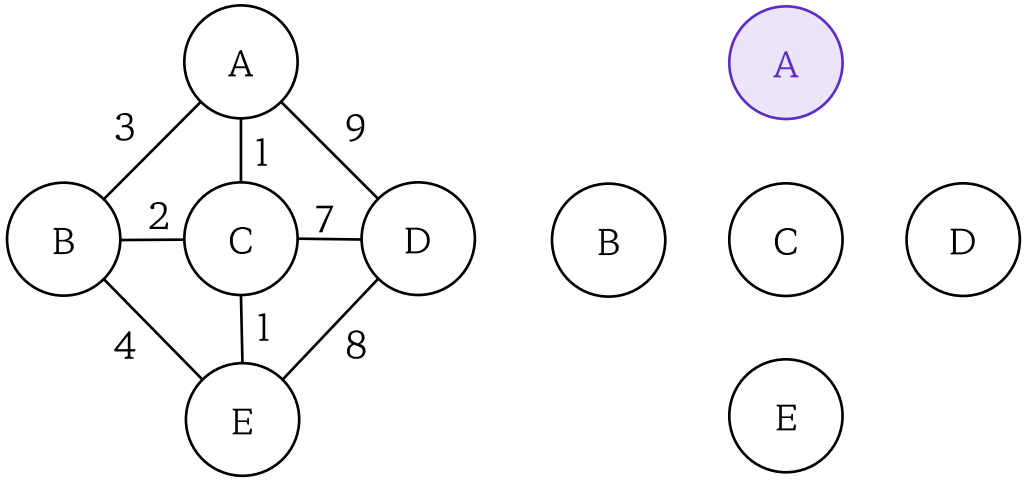
#최소 스패닝 트리

<Minimum Spanning Tree> - Prim’s Algorithm

최소 스패닝 트리 : 가중치 그래프에서 간선의 가중치 합이 가장 작은 스패닝 트리

Minimum Spanning Tree(MST)

그러면, Prim’s Algorithm을 이용하여
오른쪽 그래프의 MST를 구해보겠습니다.
먼저, 임의의 정점 A를 선택합니다.



#최소 스패닝 트리

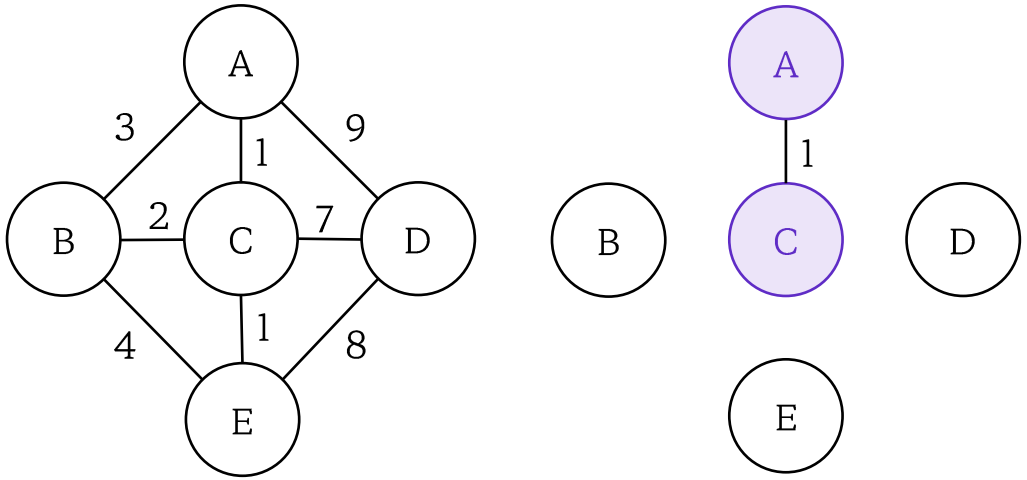
<Minimum Spanning Tree> - Prim's Algorithm

최소 스패닝 트리 : 가중치 그래프에서 간선의 가중치 합이 가장 작은 스패닝 트리

Minimum Spanning Tree(MST)

다음으로, A와 연결된 간선 중 가장 작은 간선인

A – C 간선을 추가합니다.



#최소 스패닝 트리

<Minimum Spanning Tree> - Prim's Algorithm

최소 스패닝 트리 : 가중치 그래프에서 간선의 가중치 합이 가장 작은 스패닝 트리

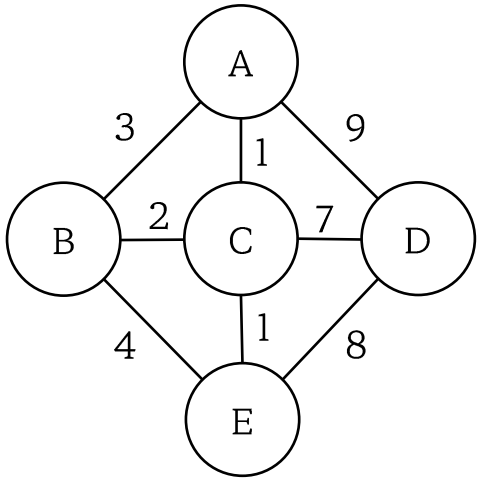
Minimum Spanning Tree(MST)

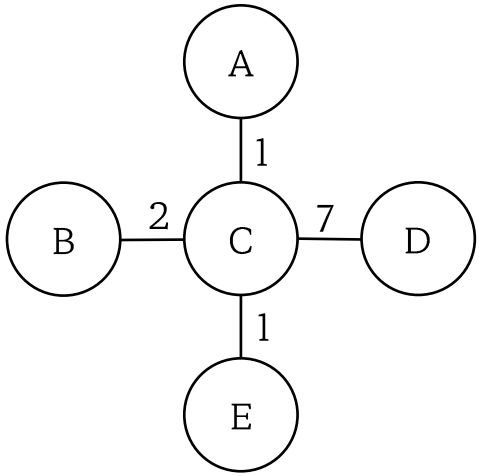
같은 과정을 반복하여 간선 3개를 더 추가하면,

A, B, C, D, E가 모두 연결된 트리가 완성됩니다.

이는 원래 그래프의 최소 스패닝 트리가 됩니다.

다시 같은 연습문제를 Prim's Algorithm으로 풀어봅시다!





#최소 스패닝 트리

<Minimum Spanning Tree> - Prim's Algorithm

Kruskal은 Disjoint Set을 이용했었습니다.

Prim은 Dijkstra와 비슷하게 구현합니다.

먼저, 입력을 인접 리스트로 저장합니다.

```
int v, e;
cin >> v >> e;
vector<pair<int, int>> edges[VERTEX_MAX + 1];
for(int i = 0; i < e; i++){
    int a, b, c;
    cin >> a >> b >> c;
    edges[a].push_back(mp(c, b));          mp = make_pair입니다
    edges[b].push_back(mp(c, a));
}
```

#최소 스패닝 트리

<Minimum Spanning Tree> - Prim's Algorithm

먼저, 1번 정점을 선택합니다.

priority_queue에서 first와 second는 각각

가중치와 현재 정점을 의미합니다.

만약 정점을 새로 방문했다면,

그 정점과 연결된 모든 간선의 정보를

priority_queue에 새로 추가하고

visited를 true로 놓습니다.

이를 모든 정점을 방문할 때까지 반복합니다.

```
int res = 0;
int visit_count = 0;
priority_stack<pair<int, int>> ps;
ps.push(mp(0, 1)); //1번 정점, 가중치 0
while(!ps.empty() && visit_count != v){
    pair<int, int> tmp = ps.top();
    ps.pop();
    if(!visited[tmp.second]){
        visited[tmp.second] = true;
        visit_count++;
        res += tmp.first;
        for(auto &p : edges[tmp.second]){
            ps.push(mp(p.first, p.second));
        }
    }
}
cout << res;
```

#최소 스패닝 트리

<Minimum Spanning Tree> - 심화문제

행성 터널(BOJ #2887)

<문제 설명>

- 3차원 좌표 N개가 주어진다.
- 이때, 이 좌표간 간선의 비용은 다음과 같이 구해진다.

  $\min(x_1 - x_2, y_1 - y_2, z_1 - z_2)$
- 모든 좌표를 연결하는 최소 비용 구하기

<제약 조건>

- $1 \leq N \leq 100{,}000$
- $10^{-9} \leq$ (좌표) $\leq 10^{9}$
- 같은 좌표는 주어지지 않는다.

#최소 스패닝 트리

<Minimum Spanning Tree> - 심화문제

행성 터널(BOJ #2887)

<문제 해설>

모든 좌표를 연결하는 최소 비용을 구해야 하므로, 최소 스패닝 트리의 가중치를 찾아야 합니다.

하지만, 주어진 좌표를 연결하는 모든 간선을 가지고 시작한다면 간선의 개수가 N^2개가 됩니다.

즉, N값이 최대 100,000인 이 문제에서는 모든 간선을 고려하기 어렵습니다.

#최소 스패닝 트리

<Minimum Spanning Tree> - 심화문제

행성 터널(BOJ #2887)

<문제 해설>

하지만, 이 문제는 좌표 간 가중치를 단순 거리가 아닌 각 좌표의 차의 최솟값으로 정의하게 됩니다.

즉, 한 점에서 가장 가중치가 작은 간선의 후보는 3개밖에 없습니다.

1. 그 점에서 x좌표 기준으로 가장 가까운 점과의 간선
2. 그 점에서 y좌표 기준으로 가장 가까운 점과의 간선
3. 그 점에서 z좌표 기준으로 가장 가까운 점과의 간선

따라서, x, y, z를 기준으로 각각 정렬한 다음, 3개의 간선을 추가하면 됩니다.

각각의 점마다 간선이 3개 그어지므로, 간선을 N^2개에서 $3N$개로 줄일 수 있습니다.

#연습 문제

<Problem Set> - Disjoint Set

2 제국 (BOJ #16402)

귀찮은 요소가 추가된 Disjoint Set

5 인간관계 (BOJ #20531)

분리 집합 + 조합론

3 교수님은 기다리지 않는다 (BOJ #3830)

감동실화 1

5 최고인 대장장이 토르비욘 (BOJ #13361)

앞의 3문제가 쉬운 고인물들을 위한 문제

#연습 문제

<Problem Set> - Minimum Spanning Tree

1 완전그래프의 최소 스패닝 트리 (BOJ #20390)

Prim을 쓰는 문제

2 복도 뚫기 (BOJ #9373)

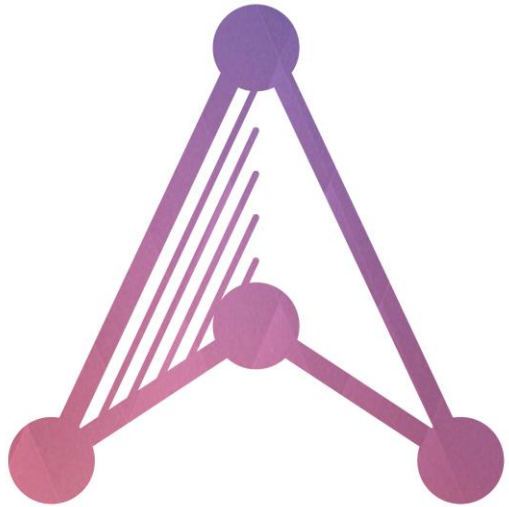
이게 왜 MST? 발상의 전환이 필요한 문제

5 조별과제 멈춰 (BOJ #23034)

감동실화 2

2 Fenced In(Platinum) (BOJ #11991)

고인물 전용 문제 2(Platinum)



ALOHA

The algorithm club.