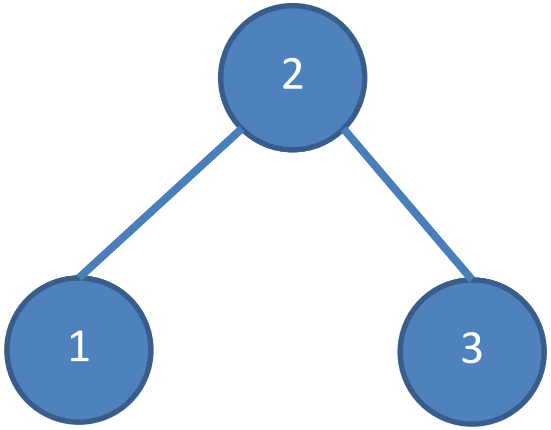
Minimum Spanning Tree

Union–Find Disjoint Sets

- Union–Find Disjoint Set เป็นโครงสร้างข้อมูลที่จำลองกลุ่มของ set ที่ไม่มีตัวซ้ำกันที่มีประสิทธิภาพ ใช้เวลาประมาณ O(1) ในการตัดสินว่า item สองตัวนั้นอยู่ใน set เดียวกันหรือไม่ และทำการรวม disjoint set สองอันเป็น set ที่ใหญ่ขึ้นได้อย่างรวดเร็ว
- โครงสร้างข้อมูลนี้สามารถถูกนำไปใช้ในปัญหาเกี่ยวกับการหา connected component ใน undirected graph
- เริ่มต้นแต่ละโหนดจะเป็น disjoint set แต่ละอัน หลังจากนั้นเราจะทำการรวมสองเส้นโหนดโดยใช้เส้นเชื่อม
- ดังนั้นเราสามารถทดสอบว่าสองโหนดอยู่ใน set เดียวกันได้ไม่ยาก

- Operation ที่ต้องการดังกล่าว ไม่ support จาก C++STL set เนื่องจากไม่ได้ออกแบบมาเพื่อการทำ disjoint set
- การใช้ vector ของ set และวนรอบแต่ละตัวเพื่อหาว่า set ไหนที่ item นั้นเป็นสมาชิกอยู่เสียค่าใช้จ่ายสูง C++STL set_union ไม่มีประสิทธิภาพ เวลารวมสอง set ใช้ linear time
- ดังนั้นเราจึงต้องการโครงสร้างข้อมูลที่มีประสิทธิภาพ นั้นคือ Union-Find Disjoint Sets

- แนวคิดของโครงสร้างข้อมูลนี้คือการเลือกตัวแทน 'parent' item เป็นตัวแทนของ set
- ถ้าเราแน่ใจว่าแต่ละ set ถูกแทนด้วย item เพียงตัวเดียว แล้วการตัดสินใจถ้า item นั้นอยู่ใน set หรือไม่จะทำได้ง่ายเลย นั่นคือ เราใช้ตัวแทน 'parent' item ในการระบุ set เลย



- ในการสร้าง Union–Find Disjoint Set จะสร้างโครงสร้าง tree ที่ disjoint sets นั้นอยู่ในรูปของต้นไม้หลายๆ ต้น(Forest of trees) โดยต้นไม้แต่ละต้นนั้นจะสอดคล้องกับ 1 disjoint set ทั้งนี้ root ของ tree จะถูกนำมาใช้เป็นตัวแทนของ set

- ดังนั้นตัวแทนที่เป็นตัวบ่งบอก set นั้นสามารถทำได้โดยท่องไปตามสายของ parent ไปยัง root ของ tree เนื่องจาก tree มีได้เพียง 1 root

- การจะทำให้มีประสิทธิภาพ เราจะเก็บ index ของ parent และความสูงของ tree ของแต่ละ set ไว้ (vi p และ vi rank ใน code ต่อไป) ทั้งนี้ vi คือ vector of integer p[i] เก็บ parent ของ item i

- ถ้า item i เป็นตัวแทนของ disjoint set แล้ว p[i]=i นั้นคือ self-loop

- rank[i] เก็บความสูงของ tree ที่มี root ที่ item i

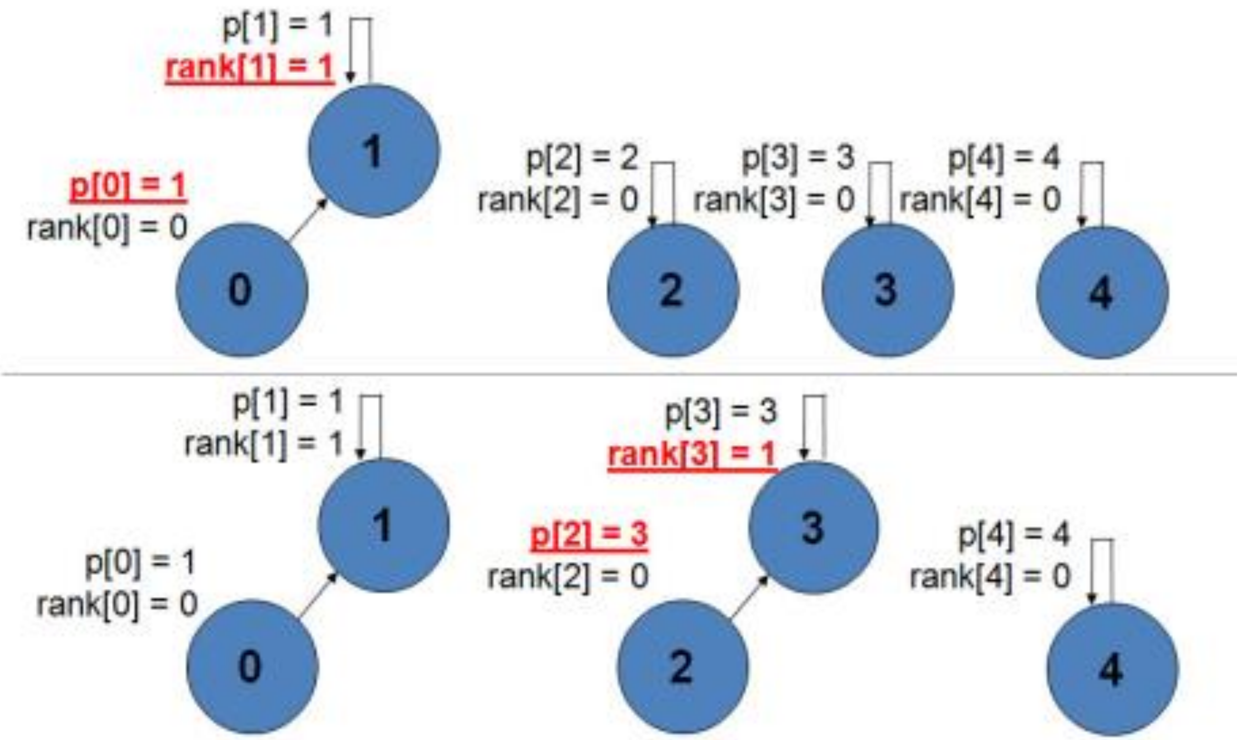
- สมมติว่าเรามี 5 disjoint sets {0,1,2,3,4} เริ่มต้นเราจะให้แต่ละ item เป็น disjoint set ด้วยตัวมันเองและให้ rank =0 และ parent ของแต่ละตัวเป็นตัวมันเอง

- ในการรวมสอง disjoint set เราจะกำหนดตัวแทน item(root) จากหนึ่งใน 2 disjoint set ให้เป็น parent ใหม่ของอีก disjoint set การทำเช่นนี้ทำให้การรวม tree 2 ต้นใน Union–Find Disjoint Sets มีประสิทธิภาพ

- unionSet(i,j) จะทำให้ทั้งสอง item i และ j มีตัวแทนตัวเดียวกันไม่ทางตรงก็ทางอ้อม

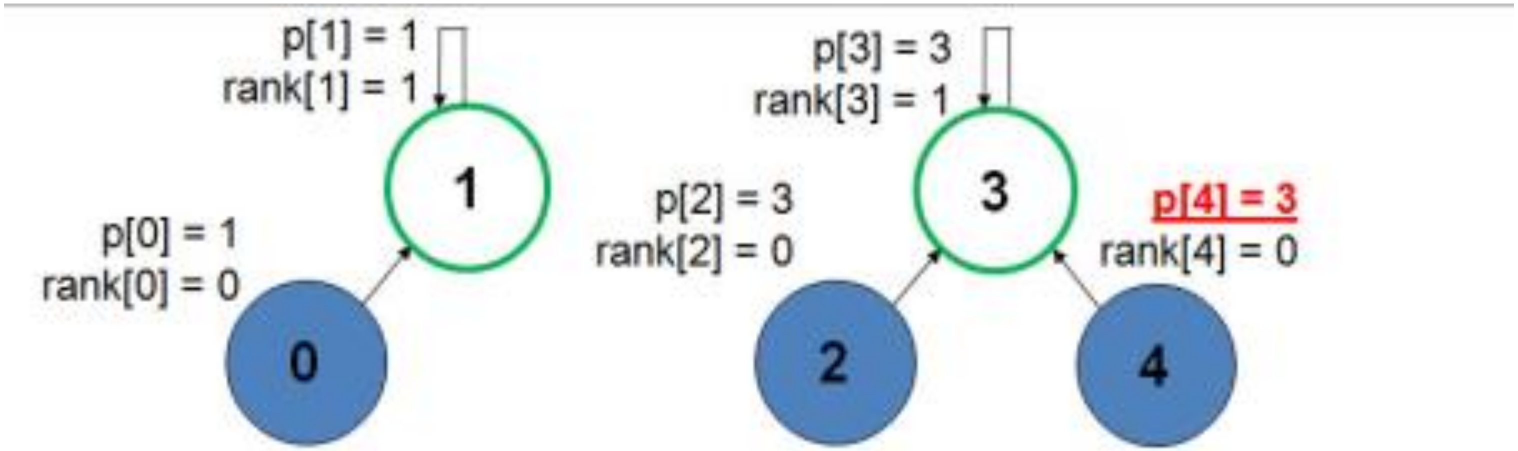
- การทำให้มีประสิทธิภาพ เราจะใช้ข้อมูลใน vi rank เพื่อกำหนด ตัวแทน item ของตัวที่มี rank สูงกว่าให้เป็น parent ใหม่ของอีกตัวที่มี rank ต่ำกว่า เพื่อทำให้ rank ของ tree ที่ได้ต่ำที่สุด

- ถ้าทั้งคู่มี rank เท่ากัน เราจะเลือกมาสักตัวหนึ่งให้เป็น parent ใหม่และเพิ่มค่าของ root's rank ทำแบบนี้เรียกว่า union by rank

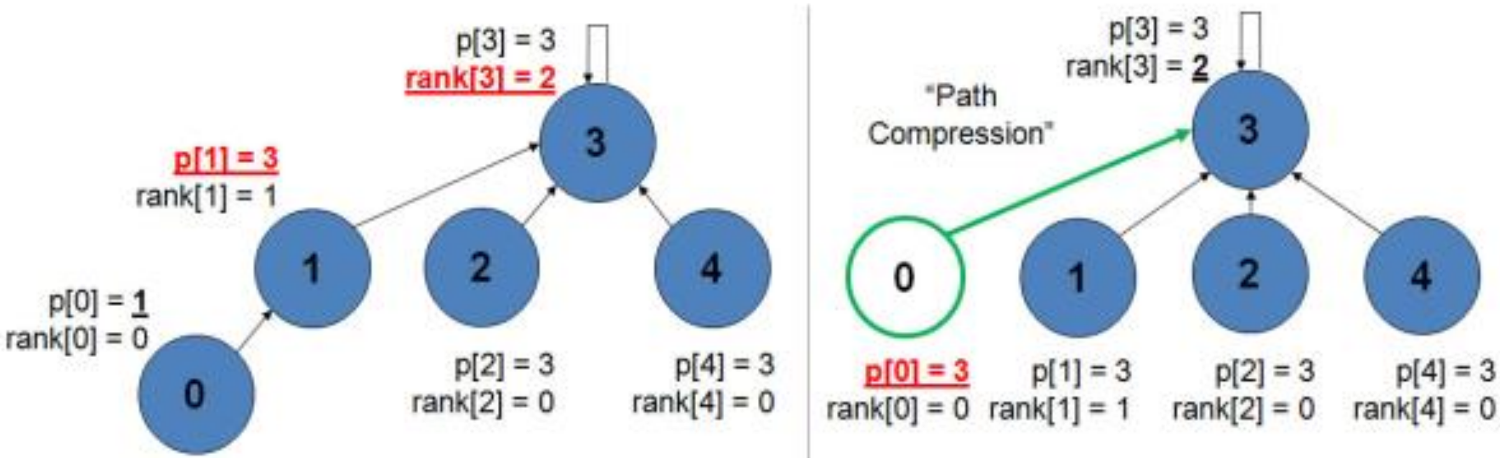


- รูปต่อไปเป็นตัวอย่างunionSet(0,1) โดยกำหนดให้ p[0] เป็น 1 และ rank[1] เป็น 1 และต่อไปเป็น unionSet(2,3) โดยกำหนด p[2] = 3 และ rank[3]=1

- เราจะสมมติว่าฟังก์ชัน findSet(i) จะเรียก findSet(p[i])แบบ recursive เพื่อหาตัวแทนของ set
- ในรูปต่อไป มีการเรียก unionSet(4,3) เรามี rank[findSet(4)] = rank[4]=0 ซึ่งน้อยกว่า rank[findset(3)] = rank[3]=1
- ดังนั้นเมื่อเรากำหนด p[4]=3 โดยไม่เปลี่ยนแปลงความสูงของ tree
- นี่คือการทำงานของ union by rank



- ที่นี้หากต้องการตรวจสอบว่าอยู่ set เดียวกันหรือไม่ isSameSet(0,4) ทำงานได้โดยไปเรียกหาว่า findSet(0) และ findSet(4) เป็นตัวเดียวกันหรือไม่
- ตัวอย่างเช่น findSet(4) = findSet(p[4])= findSet(3)=3



- มีเทคนิคที่ทำให้เพิ่มความเร็วของการ findSet นั่นคือ path compression

```cpp
#include <bits/stdc++.h>
using namespace std;
typedef vector<int> vi;

// Union-Find Disjoint Sets Library
// 1000 ปรับได้
vi pset(1000), setSize(1000);
int _numDisjointSets;
void initSet(int N) {
  setSize.assign(N, 1);
  _numDisjointSets = N;
  pset.assign(N, 0);
  for (int i = 0; i < N; i++)
          pset[i] = i;
}
```

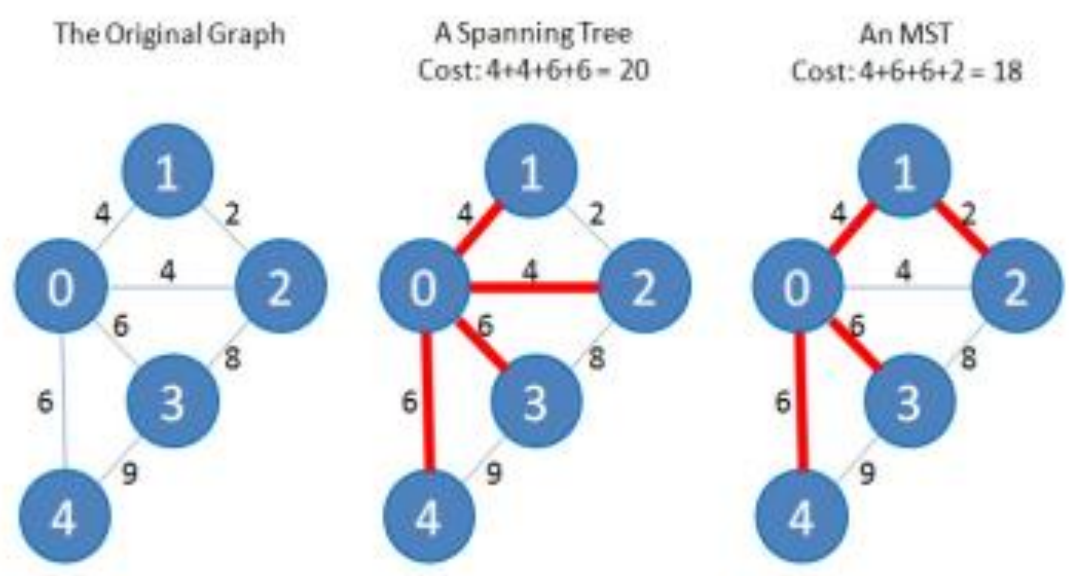
```
int findSet(int i) {
  return (pset[i] == i) ? i : (pset[i] = findSet(pset[i]));
}
bool isSameSet(int i, int j) {
  return findSet(i) == findSet(j);
}
void unionSet(int i, int j) {
  if (!isSameSet(i, j)) {
    _numDisjointSets--;
    setSize[findSet(j)] += setSize[findSet(i)];
    pset[findSet(i)] = findSet(j);
  }
}
int numDisjointSets() { return _numDisjointSets; }
int sizeOfSet(int i) { return setSize[findSet(i)]; }
```

```c
int main() {
  printf("Assume that there are 5 disjoint sets initially\n");
  initSet(5); // create 5 disjoint sets
  unionSet('A' - 'A', 'B' - 'A'); // unionSet(A, B)
  unionSet('A' - 'A', 'C' - 'A'); // unionSet(A, C)
  unionSet('D' - 'A', 'B' - 'A'); // unionSet(D, B)
  printf("findSet(A) = %d\n", findSet('A' - 'A'));
// will return 2 (the parent ID of 'A', 'B', 'C', 'D')
  printf("findSet(B) = %d\n", findSet('B' - 'A'));
// will return 2 (the parent ID of 'A', 'B', 'C', 'D')
  printf("findSet(C) = %d\n", findSet('C' - 'A'));
// will return 2 (the parent ID of 'A', 'B', 'C', 'D')
  printf("findSet(D) = %d\n", findSet('D' - 'A'));
// will return 2 (the parent ID of 'A', 'B', 'C', 'D')
  printf("findSet(E) = %d\n", findSet('E' - 'A'));
// will return 4 (the parent ID of 'E')
  printf("isSameSet(A, E) = %d\n", isSameSet('A' - 'A', 'E' - 'A'));
// will return 0 (false)
  printf("isSameSet(A, B) = %d\n", isSameSet('A' - 'A', 'B' - 'A'));
// will return 1 (true)
  return 0;
}
```

https://bit.ly/2HTYux1

Minimum Spanning Tree

- กำหนดให้ กราฟแบบมีน้ำหนัก G=(V,E) ที่ไม่มีทิศทางและเชื่อมกัน
- G เลือก subset ของเส้นเชื่อม E' ของ G ที่ทำให้กราฟ G ยังเชื่อมกันและน้ำหนักรวมของเส้นเชื่อมที่เลือก E' น้อยที่สุด



- ในการที่จะทำให้เงื่อนไขการเชื่อมกันสำเร็จนั้น เราต้องการอย่างน้อย V–1 เส้นเชื่อมเพื่อที่จะทำให้เกิดโครงสร้าง tree และ tree นี้จะต้องแผ่ทั่ว (span) ทุกโหนด V ใน G ถึงจะเป็น spanning tree

- มีหลาย spanning tree ใน G ทั้ง DFS และ BFS ก็สร้าง spanning tree ได้

- แต่ท่ามกลาง spanning tree ทั้งหมด จะมีบางต้น(อย่างน้อยหนึ่งต้น) ที่มีน้ำหนักรวมน้อยที่สุด

- ปัญหา Minimum Spanning Tree(MST) มีหลาย application
- ตัวอย่างเช่น เราสามารถจำลองปัญหาการสร้างถนนในเมืองที่ห่างไกล โดยใช้ MST ได้ โดยที่โหนดคือเมืองและเส้นเชื่อมคือถนนที่เราอาจจะสร้างเพื่อเชื่อมระหว่างเมืองสองเมือง ค่าใช้จ่ายในการสร้างถนนเพื่อเชื่อมเมือง i และเมือง j คือน้ำหนักของเส้นเชื่อม (i, j) MST ของกราฟนี้คือค่าใช้จ่ายต่ำสุดในการสร้างถนนเพื่อให้เชื่อมทุกเมือง
- MST สามารถแก้ได้ด้วยหลายๆ algorithm เช่น Prim's และ Kruskal's ทั้งคู่เป็น **Greedy algorithm**
- น้ำหนักของ MST ที่ได้จากสองอัลกอริทึมนั้นเป็นค่าเฉพาะนั่นคือ อาจมี MST หลายต้นแต่มีค่าน้ำหนักน้อยสุดค่าเดียว

Kruskal's Algorithm

- Joseph Bernard *Kruskal* Jr.'s algorithm เริ่มต้นจะเรียงเส้นเชื่อม E ตามน้ำหนักจากน้อยไปมาก สามารถทำได้ไม่ยากโดยการเก็บเส้นเชื่อมไว้ใน EdgeList data structure จากนั้น sort มันจากน้อยไปมาก Kruskal's algorithm จะลองเพิ่มเส้นเชื่อมเข้าไปใน MST ตราบเท่าที่การเพิ่มเส้นเชื่อมนั้นไม่ทำให้เกิด cycle

- ในการตรวจสอบ cycle นั้นทำได้โดยการใช้ Union–Find Disjoint Sets

- Code ค่อนข้างสั้นเพราะว่าเราแยกส่วน Union–Find Disjoint Sets ไปอีกฟังก์ชัน
- เวลาในการทำงานเป็น O(sorting+ trying to add each edge $\times$ cost of Union–Find operations) $= O(E \log E + E \times (\approx 1)) = O(E \log E) = O(E \log V^2)$
- $= O(2E \log V) = O(E \log V)$.

Union-Find Disjoint Sets

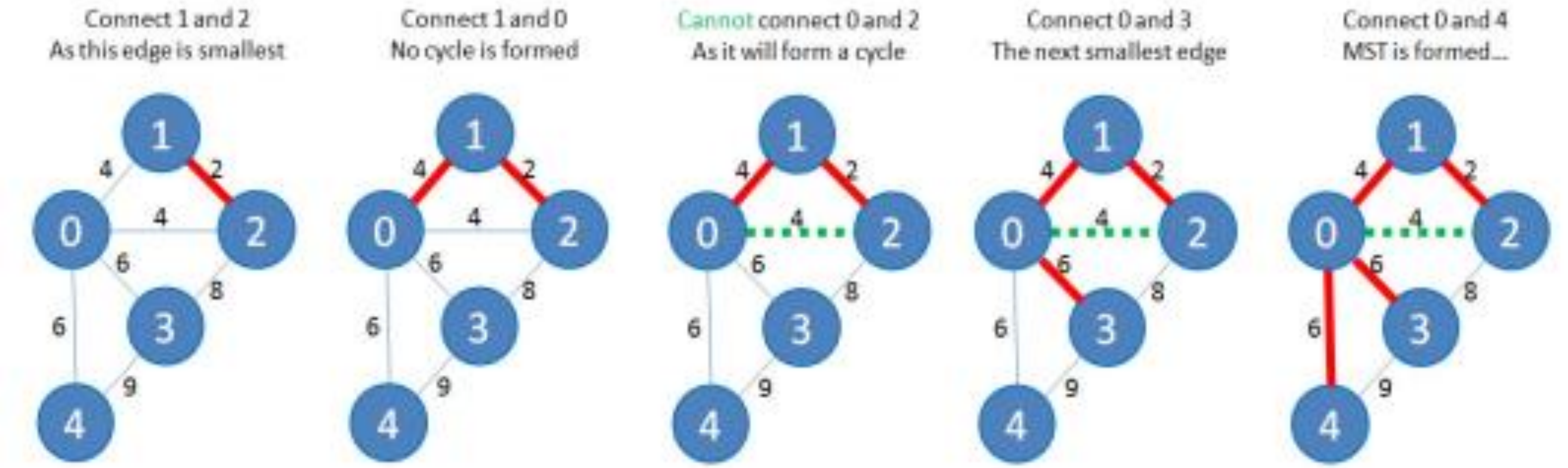
```cpp
#include <bits/stdc++.h>
using namespace std;
typedef pair<int, int> ii;
typedef vector<int> vi;
typedef vector<ii> vii;
vi pset(1000), setSize(1000);
int _numDisjointSets;
void initSet(int N) { setSize.assign(N, 1); _numDisjointSets = N;
pset.assign(N, 0); for (int i = 0; i < N; i++) pset[i] = i; }
int findSet(int i) { return (pset[i] == i) ? i : (pset[i] =
findSet(pset[i])); }
bool isSameSet(int i, int j) { return findSet(i) == findSet(j); }
void unionSet(int i, int j) {
  if (!isSameSet(i, j)) {
    _numDisjointSets--;
    setSize[findSet(j)] += setSize[findSet(i)];
    pset[findSet(i)] = findSet(j); } }
int numDisjointSets() { return _numDisjointSets; }
int sizeOfSet(int i) { return setSize[findSet(i)]; }
```

```
vector<vii> AdjList;
vi taken;                // global boolean flag to avoid cycle
int main() {
  int V, E, u, v, w;
  scanf("%d %d", &V, &E);
  AdjList.assign(V, vii());
  vector< pair<int, ii> > EdgeList;
// format: weight, two vertices of the edge
  for (int i = 0; i < E; i++) {
    scanf("%d %d %d", &u, &v, &w);    // read the triple: (a, b, w)
    EdgeList.push_back(make_pair(w, ii(u, v)));
// but store it as: (w, a, b)
    AdjList[u].push_back(ii(v, w));
    AdjList[v].push_back(ii(u, w));
  }
  sort(EdgeList.begin(), EdgeList.end());
// sort by edge weight in O(E log E)
```

```
  int mst_cost = 0; initSet(V); // all V are disjoint sets initially
  for (int i = 0; i < E; i++) { // for each edge, O(E)
    pair<int, ii> front = EdgeList[i];
    if (!isSameSet(front.second.first, front.second.second)) {
// if no cycle
      mst_cost += front.first;    // add the weight of e to MST
      unionSet(front.second.first, front.second.second);
// link endpoints
    }
  } // note: the runtime cost of UFDS is very light
  // note: the number of disjoint sets must eventually be one for a
valid MST
  printf("MST cost = %d (Kruskal's)\n", mst_cost);

  return 0;
}
```

https://bit.ly/2WuBk4t



5 7

0 1 4

0 2 4

0 3 6

0 4 6

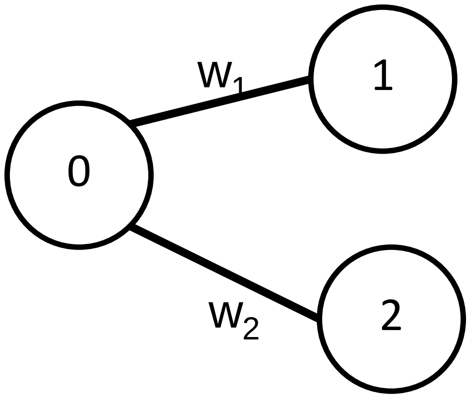
1 2 2

2 3 8

3 4 9

Prim's Algorithm

- Robert Clay Prim's algorithm นั้นเริ่มต้นเลือก start node มา (เพื่อความง่ายเราก็เลือกโหนด 0)



- หลังจากนั้น set ค่ามันว่าถูกใช้แล้ว (taken) และ enqueue ข้อมูลน้ำหนัก w และจุดปลายอีกฝั่ง u ของเส้นเชื่อม 0−>u ที่ยังไม่ถูก taken ลงใน priority queue

- ข้อมูลคู่นี้ถูกเรียงใน priority queue ตามน้ำหนักที่เพิ่มขึ้น และถ้าน้ำหนักเท่ากันเรียงตามหมายเลข node

- หลังจากนั้น Prim's algorithm จะเลือก pair(w,u) แบบ greedy จาก priority queue โดยเลือกจากน้ำหนัก w ที่น้อยก่อน ถ้าจุดปลายของเส้นเชื่อมนี้ไม่เคยถูกพบมาก่อน การทำเช่นนี้เพื่อป้องกันการเกิด cycle

- ถ้า pair(w,u) นั้นผ่านเงื่อนไข เราจะเพิ่ม น้ำหนักของ w ให้กับค่าของ MST และ u จะถูกมาร์คว่า taken และคู่ของ (w',v) ของแต่ละเส้นเชื่อม u->v ที่มีน้ำหนัก w' ที่เกิดกับ u ถูกเพิ่มลงไปใน priority queue ถ้า v ไม่เคย taken มาก่อน
- จะวนรอบทำไปจนกระทั่ง priority queue ว่าง
- code ยาวพอๆ กับ Kruskal และทำงานใน O(แต่ละเส้นเชื่อมทำงานครั้งเดียว x ค่าใช้จ่ายในการ dequeue/enqueue) =O(E logE) =O(ElogV)

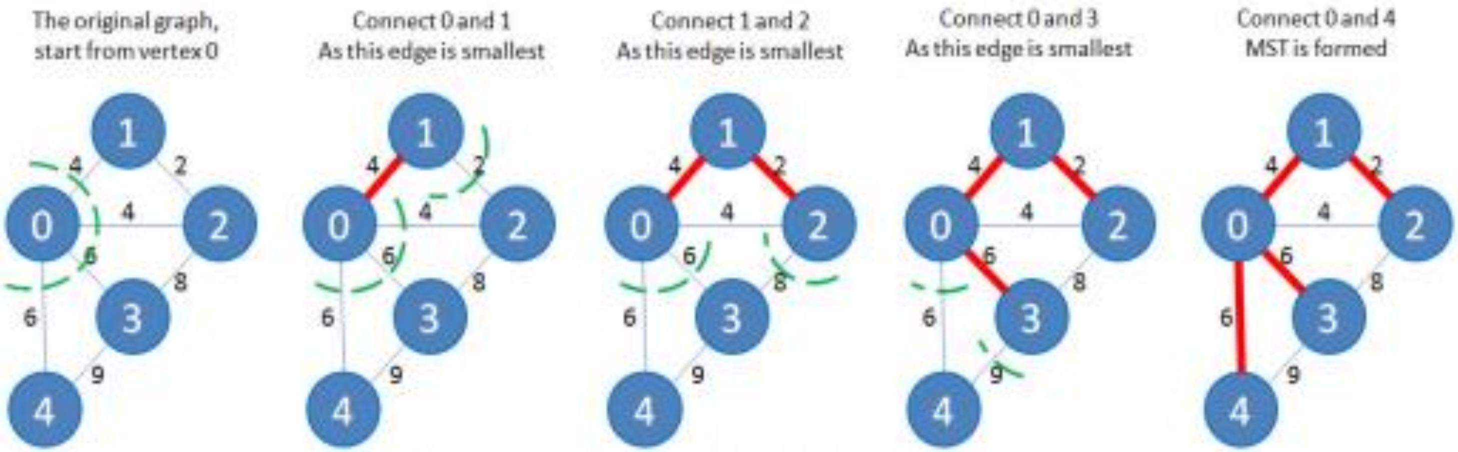
```
#include <bits/stdc++.h>
using namespace std;
typedef pair<int, int> ii;
typedef vector<int> vi;
typedef vector<ii> vii;
vector<vii> AdjList;
vi taken;                   // global boolean flag to avoid cycle
priority_queue<ii> pq;  // priority queue to help choose shorter edges
void process(int vtx) {
  taken[vtx] = 1;
  for (int j = 0; j < AdjList[vtx].size(); j++) {
    ii v = AdjList[vtx][j];
    if (!taken[v.first]) pq.push(ii(-v.second, -v.first));
  }
}
// sort by (inc) weight then by (inc) id by using -ve sign to reverse
order
```

```
int main() {
  int V, E, u, v, w;
  scanf("%d %d", &V, &E);
  AdjList.assign(V, vii());
  vector< pair<int, ii> > EdgeList;
//format: weight, two vertices of the edge

  for (int i = 0; i < E; i++) {
    scanf("%d %d %d", &u, &v, &w);
// read the triple: (a, b, w)
    EdgeList.push_back(make_pair(w, ii(u, v)));
// but store it as: (w, a, b)
    AdjList[u].push_back(ii(v, w));
    AdjList[v].push_back(ii(u, w));
  }
```

```
  int mst_cost = 0;
  // inside int main() --- assume the graph has been stored in AdjList
  taken.assign(V, 0);
  process(0);   //take vertex 0 and process all edges incident to vertex 0
  mst_cost = 0;
  while (!pq.empty()) { //repeat until V vertices (E = V-1 edges) are taken
    ii front = pq.top();
    pq.pop();
    u = -front.second;
    w = -front.first;       // negate the id and weight again
    if (!taken[u]){         // we have not connect this vertex yet
      mst_cost += w;
      process(u);
    }   // take u and process all edges incident to u
  }    // each edge is in pq only once!
  printf("MST cost = %d (Prim's)\n", mst_cost);
}
```

https://bit.ly/2HWWXXc


The original graph,
start from vertex 0
Connect 0 and 1
As this edge is smallest
Connect 1 and 2
As this edge is smallest
Connect 0 and 3
As this edge is smallest
Connect 0 and 4
MST is formed

ตัวอย่างของกราฟ

5 7

0 1 4

0 2 4

0 3 6

0 4 6

1 2 2

2 3 8

3 4 9