一、 程式摘要

1. **邏輯/原理**

求「Minimum Spanning Tree」，我們分別以下列兩種演算法實作。

✓ **Kruskal**

以蒐集 edge 的方是找出最小 weight 且不形成 cycle 的情況下來建立 MST。首先建立 edge 與 vertex 的 set，前者儲存所有 edge 及其 weigth，後者儲存各個 vertex 的 parent 及 child 關係。我們需先將 edge set 依 weight 由小到大排序，再從 edge set 中一一取出兩端的 vertex，利用 FIND_ROOT_VERTEX method 去判斷兩 vertex 是否屬於同一個 set 以及有無 cycle 存在，如果不是則可加入 MST 中，並利用 UNION_VERTEX method 將兩 vertex 合併為一個新 set，接著再從 edge set 中的取出下一條 edge 重複步驟，最後即可形成 MST。

✓ **Prim**

先建立幾組 set（edge：儲存所有 edge 及其 weight；parent：儲存各個 vertex 的 parent 及 child 關係；key：edge 的 weight，但會存在 vertex 上，表示到達該 vertex 的成本；visited 的表示是否已在 MST 中）。決定起始 vertex u，找出與 u 相連 edge 之 weight，判斷從 u 經由 edge 到達 v 需要多少成本，若成本小於原 key[v]中的成本，則更新 key[v]，同時也代表兩 vertex 可以有一個 parent and child 的關係，因此也記錄到 parent set 中；之後再利用 EXTRACT_MIN method 選出 key set 中值最小的 vertex 放入 MST，並重複前述步驟，找出與此 vertex 相連的所有 edge 做判斷，以此概念逐漸向外擴張，最終即可蒐集完 MST。

2. **語言**

兩種方式皆以 C 語言實作。

二、 程式內容說明

1. 程式註解

✓ **Kruskal**

```
#include <stdio.h>
#include <stdlib.h>

// 宣告常數
#define MAX_VERTICES_NUMBER 1000
#define MAX_EDGES_NUMBER 10000

// 裝填 edge 相關屬性的結構
// 包含兩個 vertex 及其權重
```

```c
struct edge {
    int vertex_one;
    int vertex_two;
    int weight;
};

// 裝填各種 set 的結構
struct set_blob {
    // 記錄 vertex
    // 索引為 vertex number
    // 對應到的值為其 parent
    int vertex_set[MAX_VERTICES_NUMBER];
    // 記錄 vertex 的合併的優先權，越大的應在越上層
    // 索引為 vertex number
    // 對應到的值為其權重
    int rank_set[MAX_VERTICES_NUMBER];
    // 記錄 edge
    struct edge edge_set[MAX_EDGES_NUMBER];
};

// 裝填結果的結構
struct mst_result {
    struct edge edge_set[MAX_EDGES_NUMBER];
    int edge_size;
    int option_num;
    int sum;
};

// 宣告 set 集合的全域變數
struct set_blob sb;

// 找出最大值
int find_max(int a, int b) {
    return (a > b) ? a : b;
}

// 找出最小值
int find_min(int a, int b) {
    return (a < b) ? a : b;
}

// qsort 所需比較 edge 結構大小(以 weight 為基準)的 funciton
```

```c
int edge_compare(const void *a, const void *b) {
	struct edge *edge_a = (struct edge *)a;
	struct edge *edge_b = (struct edge *)b;
	return edge_a->weight - edge_b->weight;
}

// 清空 set 的 function
void clear_set(int vertex_num, int edge_num) {
	for (int i = 0; i < vertex_num; i++) {
		sb.vertex_set[i] = 0;
		sb.rank_set[i] = 0;
	}

	for (int i = 0; i < edge_num; i++) {
		sb.edge_set[i].vertex_one = 0;
		sb.edge_set[i].vertex_two = 0;
		sb.edge_set[i].weight = 0;
	}
}

// 建立 edge relation
void add_edge(int index, int vertex_one, int vertex_two, int weight) {
	sb.edge_set[index].vertex_one = vertex_one;
	sb.edge_set[index].vertex_two = vertex_two;
	sb.edge_set[index].weight = weight;
}

// 建立 vertex
void make_vertex(int vertex) {
	sb.vertex_set[vertex] = vertex;
}

// 找出 vertex 的 root
int find_root_vertex(int vertex) {
	int parent = sb.vertex_set[vertex];
	if (parent != vertex) {
		sb.vertex_set[vertex] = find_root_vertex(parent);
	}
	return sb.vertex_set[vertex];
}

// 合併兩棵 MST
```

```
void union_vertex(int vertex_one, int vertex_two) {
	int vertex_one_rank = sb.rank_set[vertex_one];
	int vertex_two_rank = sb.rank_set[vertex_two];
	if (vertex_one_rank < vertex_two_rank) {
		sb.vertex_set[vertex_one] = vertex_two;
	} else {
		sb.vertex_set[vertex_two] = vertex_one;
		if (vertex_one_rank == vertex_two_rank) {
			sb.rank_set[vertex_one] += 1;
		}
	}
}

// 找出 MST 的 function
struct mst_result kruskal(int vertex_num, int edge_num) {

	// 初始化裝填結果的結構
	struct mst_result result;
	result.edge_size = 0;
	result.sum = 0;

	// 逐一建立 vertex
	for (int i = 0; i < vertex_num; i++) {
		make_vertex(i);
	}

	// 將 edge 依照 weight 由小到大進行排序
	qsort(sb.edge_set, edge_num, sizeof(struct edge), edge_compare);

	// 逐一蒐集 MST 的 edge
	for (int i = 0; i < edge_num; i++) {
		int vertex_one = sb.edge_set[i].vertex_one;
		int vertex_two = sb.edge_set[i].vertex_two;
		int vertex_one_root = find_root_vertex(vertex_one);
		int vertex_two_root = find_root_vertex(vertex_two);
		if (vertex_one_root != vertex_two_root) {
			// 裝填結果
			int weight = sb.edge_set[i].weight;
			int index = result.edge_size;
			result.edge_set[index].vertex_one = find_min(vertex_one, vertex_two);
			result.edge_set[index].vertex_two = find_max(vertex_one, vertex_two);
			result.edge_set[index].weight = weight;
```

```
                result.edge_size += 1;
                result.sum += weight;
                // 進行合併
                union_vertex(vertex_one_root, vertex_two_root);
            }
        }
        // 回傳結果
        return result;
}

// main funciton
int main() {
        // 輸入測資筆數
        int input_num;
        scanf("%i", &input_num);

        // 宣告多個裝填結果的結構
        struct mst_result result[input_num];

        // 輸入測資內容
        for (int i = 0; i < input_num; i++) {

            // 輸入 vertex、edge 及 option
            int vertex_num;
            int edge_num;
            int option_num;
            scanf("%d %d %d", &vertex_num, &edge_num, &option_num);

            // 建立 edge
            for (int j = 0; j < edge_num; j++) {
                int vertex_one;
                int vertex_two;
                int weight;
                scanf("%d %d %d", &vertex_one, &vertex_two, &weight);
                add_edge(j, vertex_one, vertex_two, weight);
            }

            // 呼叫運算 function
            result[i] = kruskal(vertex_num, edge_num);
            result[i].option_num = option_num;
            // 清空 set
            clear_set(vertex_num, edge_num);
```

```
        }

        // 輸出結果
        for (int i = 0; i < input_num; i++) {
            if (result[i].option_num == 1) {
                for (int j = 0; j < result[i].edge_size; j++) {
                    printf("%d %d\n", result[i].edge_set[j].vertex_one,
result[i].edge_set[j].vertex_two);
                }
            }
            printf("%d\n", result[i].sum);
        }
    }
```

✓ **Prim**

```
#include <stdio.h>
#include <limits.h>

// 宣告常數
#define MAX_VERTICES_NUMBER 1001
#define MAX_EDGES_NUMBER 10001
#define TRUE 1
#define FALSE 0
#define START 0
#define DEFAULT_VALUE -1

// 裝填各種 set 的結構
struct set_blob {
    // 記錄 edge 的 weight，將之暫存在 vertex 上，表示到達該 vertex 的成本
    int key_set[MAX_VERTICES_NUMBER];
    // 記錄最後在 MST 中 各個 vertex 的 parent 與 child 關係
    int parent_set[MAX_VERTICES_NUMBER];
    // 記錄哪些 vertex 已經在 MST 中
    int visited_set[MAX_VERTICES_NUMBER];
    // 記錄 edge relation
    // 對應到的值為 weight
    int edge_set[MAX_VERTICES_NUMBER][MAX_VERTICES_NUMBER];
};

// 裝填 edge 的結構
// 用於結果的輸出
struct edge {
```

```
        int vertex_one;
        int vertex_two;
};

// 裝填結果的結構
struct mst_result {
        struct edge edge_set[MAX_EDGES_NUMBER];
        int edge_size;
        int option_num;
        int sum;
};

// 宣告 set 集合的全域變數
struct set_blob sb;

// 找出最大值
int find_max(int a, int b) {
        return (a > b) ? a : b;
}

// 找出最小值
int find_min(int a, int b) {
        return (a < b) ? a : b;
}

// 初始化 set 的 funciton
void init_set() {
        for (int i = 0; i < MAX_VERTICES_NUMBER; i++) {
                for (int j = 0; j < MAX_VERTICES_NUMBER; j++) {
                        sb.edge_set[i][j] = DEFAULT_VALUE;
                }
                sb.key_set[i] = (i == START) ? DEFAULT_VALUE : INT_MAX;
                sb.parent_set[i] = DEFAULT_VALUE;
                sb.visited_set[i] = FALSE;
        }
}

// 建立 edge relation
void add_edge(int vertex_one, int vertex_two, int weight) {
        if (sb.edge_set[vertex_one][vertex_two] == DEFAULT_VALUE) {
                sb.edge_set[vertex_one][vertex_two] = weight;
                sb.edge_set[vertex_two][vertex_one] = weight;
```

```
        } else {
            if (weight < sb.edge_set[vertex_one][vertex_two]) {
                sb.edge_set[vertex_one][vertex_two] = weight;
                sb.edge_set[vertex_two][vertex_one] = weight;
            }
        }
}

// 在尚未放進 MST 的 vertex 中，取出有最小 key 值的 vertex
int extract_min(int vertex_num) {
        int min = INT_MAX;
        int vertex = DEFAULT_VALUE;
        for (int i = 0; i < vertex_num; i++) {
            if (sb.visited_set[i] == FALSE && sb.key_set[i] < min) {
                min = sb.key_set[i];
                vertex = i;
            }
        }
        return vertex;
}

// 找出 MST 的 function
struct mst_result prim(int vertex_num) {

        // 初始化裝填結果的結構
        struct mst_result result;
        result.edge_size = 0;
        result.sum = 0;

        // 逐一將 vertex 放入 MST
        for (int i = 0; i < vertex_num; i++) {
            int vertex = extract_min(vertex_num);
            // 放進 MST
            sb.visited_set[vertex] = TRUE;

            // 裝填結果
            // 跳過起點
            if (vertex != START) {
                int index = result.edge_size;
                int parent = sb.parent_set[vertex];
                int weight = sb.edge_set[vertex][parent];
                result.edge_set[index].vertex_one = find_min(vertex, parent);
```

```
                result.edge_set[index].vertex_two = find_max(vertex, parent);
                result.edge_size += 1;
                result.sum += weight;
            }
            for (int j = 0; j < vertex_num; j++) {
                if (sb.visited_set[j] == FALSE && sb.edge_set[vertex][j] != DEFAULT_VALUE &&
sb.edge_set[vertex][j] < sb.key_set[j]) {
                    sb.parent_set[j] = vertex;
                    sb.key_set[j] = sb.edge_set[vertex][j];
                }
            }
        }
        // 回傳結果
        return result;
    }

    // main funciton
    int main() {
        // 輸入測資筆數
        int input_num;
        scanf("%i", &input_num);

        // 宣告多個裝填結果的結構
        struct mst_result result[input_num];

        // 輸入測資內容
        for (int i = 0; i < input_num; i++) {

            // 輸入 vertex、edge 及 option
            int vertex_num;
            int edge_num;
            int option_num;
            scanf("%d %d %d", &vertex_num, &edge_num, &option_num);

            // 初始化 set
            init_set();

            // 建立 edge
            for (int j = 0; j < edge_num; j++) {
                int vertex_one;
                int vertex_two;
                int weight;
```

```
                scanf("%d %d %d", &vertex_one, &vertex_two, &weight);
                add_edge(vertex_one, vertex_two, weight);
            }

            // 呼叫運算 function
            result[i] = prim(vertex_num);
            result[i].option_num = option_num;
        }
        // 輸出結果
        for (int i = 0; i < input_num; i++) {
            if (result[i].option_num == 1) {
                for (int j = 0; j < result[i].edge_size; j++) {
                    printf("%d %d\n", result[i].edge_set[j].vertex_one,
result[i].edge_set[j].vertex_two);
                }
            }
            printf("%d\n", result[i].sum);
        }
    }
```

2. 圖解

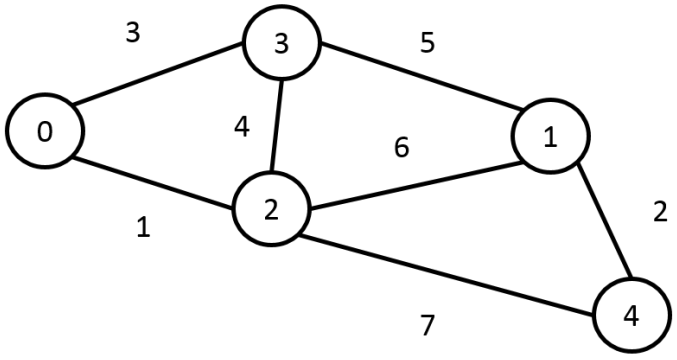
✓ **Kruskal**

給定一組 edge set（如下圖），u 及 v 代表 vertex，以 weight 為基準進行排序，並依下列步驟說明找出 MST 的過程。

(a)

u	0	1	3	3	1	2	4
v	2	4	0	2	3	1	2
weight	1	2	3	4	5	6	7

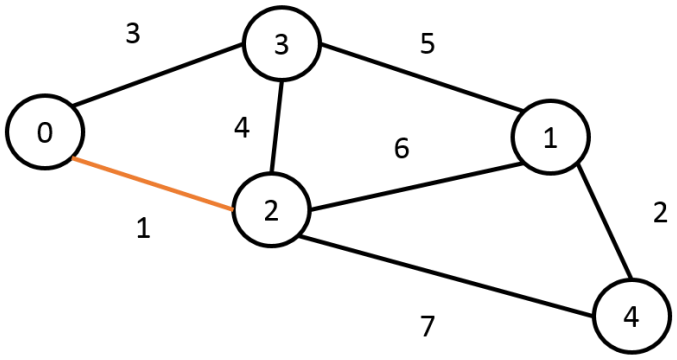
	0	1	2	3	4
vertex_set	0	1	2	3	4
rank_set	0	0	0	0	0



(b)

u	0	1	3	3	1	2	4
v	2	4	0	2	3	1	2
weight	1	2	3	4	5	6	7

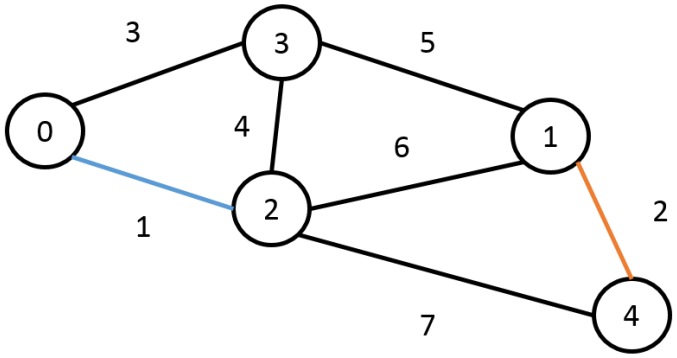
	0	1	2	3	4
vertex_set	0	1	0	3	4
rank_set	1	0	0	0	0



(c)

u	0	1	3	3	1	2	4
v	2	4	0	2	3	1	2
weight	1	2	3	4	5	6	7

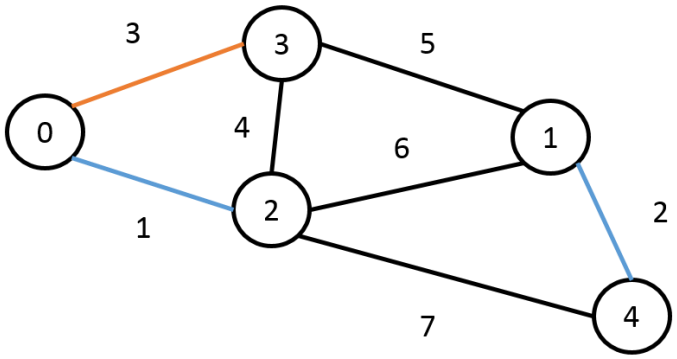
	0	1	2	3	4
vertex_set	0	1	0	3	1
rank_set	1	1	0	0	0



(d)

u	0	1	3	3	1	2	4
v	2	4	0	2	3	1	2
weight	1	2	3	4	5	6	7

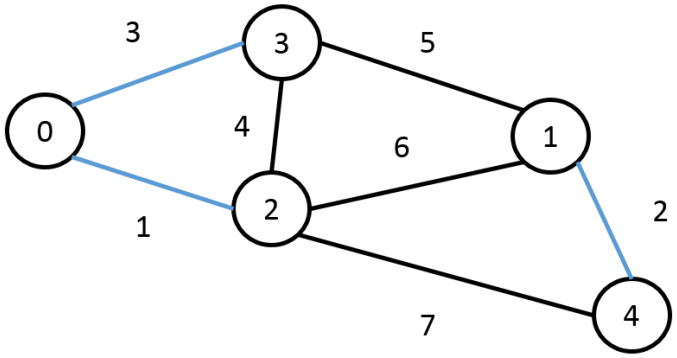
	0	1	2	3	4
vertex_set	0	1	0	0	1
rank_set	1	1	0	0	0



(e)

u	0	1	3	3	1	2	4
v	2	4	0	2	3	1	2
weight	1	2	3	4	5	6	7

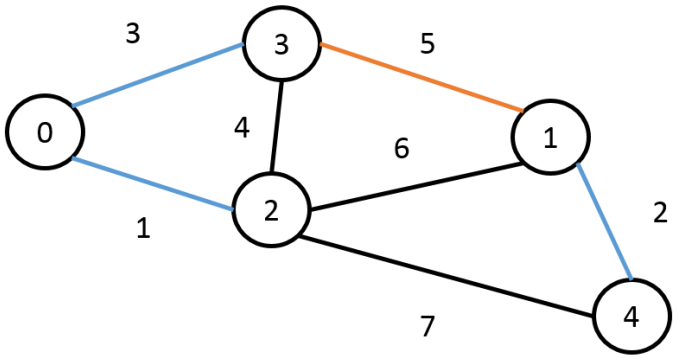
	0	1	2	3	4
vertex_set	0	1	0	0	1
rank_set	1	1	0	0	0



(f)

u	0	1	3	3	1	2	4
v	2	4	0	2	3	1	2
weight	1	2	3	4	5	6	7

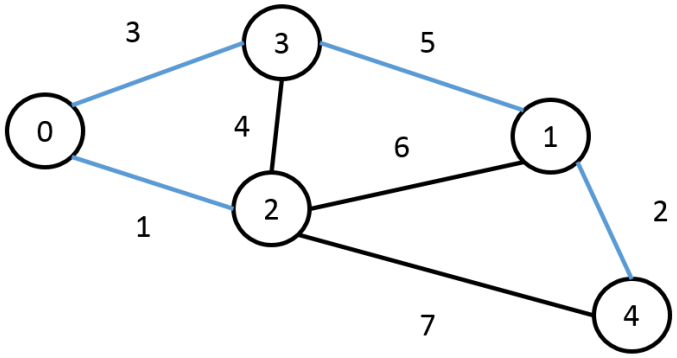
	0	1	2	3	4
vertex_set	1	1	0	0	1
rank_set	1	2	0	0	0



(g)

u	0	1	3	3	1	2	4
v	2	4	0	2	3	1	2
weight	1	2	3	4	5	6	7

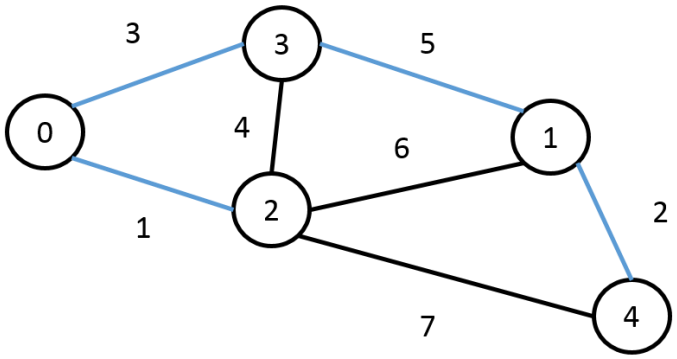
	0	1	2	3	4
vertex_set	1	1	1	0	1
rank_set	1	2	0	0	0



(h)

u	0	1	3	3	1	2	4
v	2	4	0	2	3	1	2
weight	1	2	3	4	5	6	7

	0	1	2	3	4
vertex_set	1	1	1	0	1
rank_set	1	2	0	0	0



✓ **Prim**

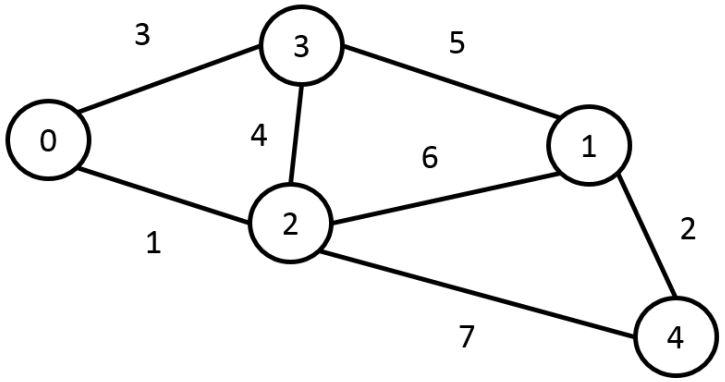
建立一個 5 乘 5 的 edge matrix，用來表示兩 vertex 所構成的 edge 其對應到的 weight；-1 則

表示不存在該 edge，依據此表以下列步驟說明蒐集 MST 的過程。

	0	1	2	3	4
0	-1	-1	1	3	-1
1	-1	-1	6	5	2
2	1	6	-1	4	-1
3	3	5	4	-1	-1
4	-1	2	-1	-1	-1

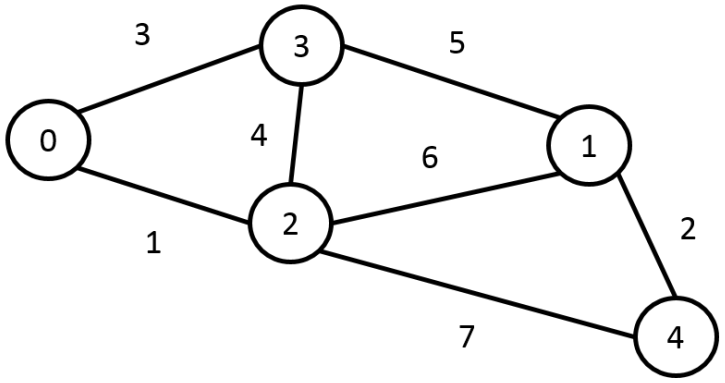
(a)

	0	1	2	3	4
parent_set	-1	-1	-1	-1	-1
key_set	-1	MAX	MAX	MAX	MAX
visited_set	F	F	F	F	F



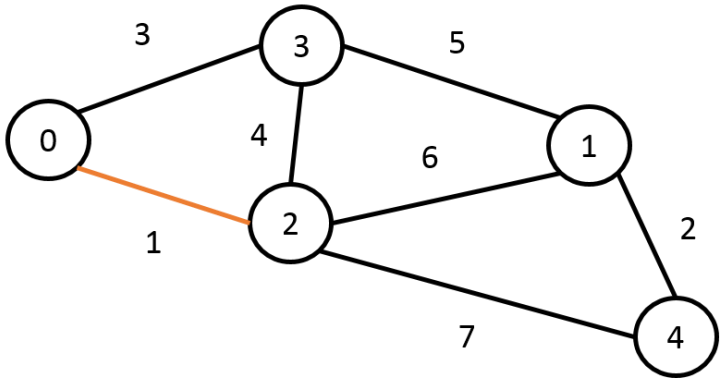
(b)

	0	1	2	3	4
parent_set	-1	-1	0	0	-1
key_set	-1	MAX	1	3	MAX
visited_set	T	F	F	F	F



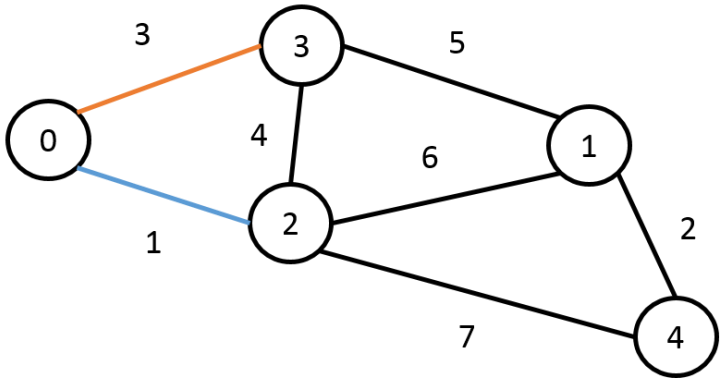
(c)

	0	1	2	3	4
parent_set	-1	2	0	0	-1
key_set	-1	6	1	3	MAX
visited_set	T	F	T	F	F



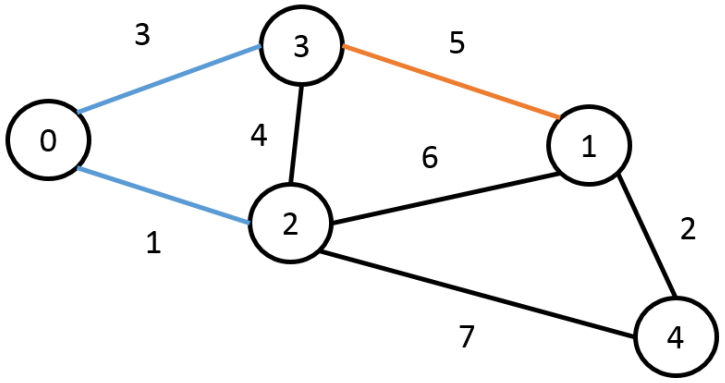
(d)

	0	1	2	3	4
parent_set	-1	3	0	0	-1
key_set	-1	5	1	3	MAX
visited_set	T	F	T	T	F



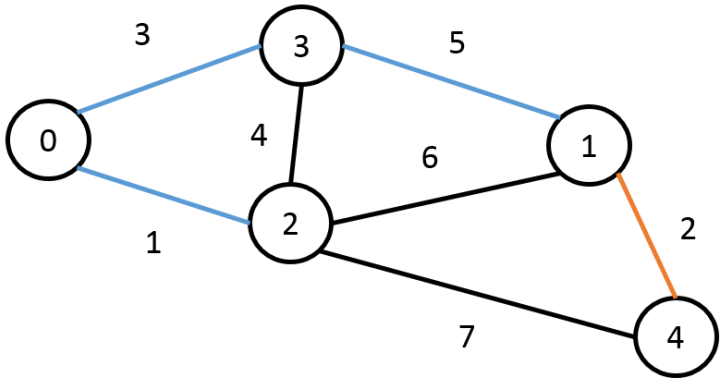
(e)

	0	1	2	3	4
parent_set	-1	3	0	0	1
key_set	-1	5	1	3	2
visited_set	T	T	T	T	F



(f)

	0	1	2	3	4
parent_set	-1	3	0	0	1
key_set	-1	5	1	3	2
visited_set	T	T	T	T	T



3. 虛擬碼

✓ Kruskal

```
// 建立 vertex
MAKE_VERTEX(v) {
    // 先將自己與 parent 設為一樣
    vertex_set[v] = v;
}

// 找出 vertex 的 root
FIND_ROOT_VERTEX(v) {
    // 若還沒找到 root
    if (vertex_set[v] != v)
        // 繼續往上找
        // 並且將 parent 更新，該 vertex 最終才會順利指向 root
        vertex_set[v] = FIND_ROOT_VERTEX(vertex_set[v])
    // 回傳 parent
    return vertex_set[v]
}

// 合併兩棵 MST
UNION_VERTEX(u, v) {
    // rank 越大表示該 vertex 底下有越多 vertex
    if (rank_set[u] < rank_set[v])
        // 將 rank 大的當上層
```

```
            vertex_set[u] = v
        else
            vertex_set[v] = u
            // 若一樣大
            if (rank_set[u] == rank_set[v])
                // 則挑選其一累加 rank
                rank_set[u]++

// 找出 MST
KRUSKAL(V, E) {

    // 逐一建立 vertex
    for i = 0 to V
        make_vertex(i)

    // 將 edge 依照 weight 由小到大進行排序
    QSORT(edge_set, edge_compare)

    // 逐一蒐集 MST 的 edge
    for i = 0 to E
      // 判斷兩 vertex u 與 v 是否屬於同一個 set
        int u_root = FIND_ROOT_VERTEX(u)
        int v_root = FIND_ROOT_VERTEX(v)
        if (u_root != v_root)
            // 將 u 與 v 合併成一個新的 set
            UNION_VERTEX(u_root, vertex_two_root)
}
```

✓ **Prim**

```
// 取出有最小 key 值的 vertex
EXTRACT_MIN(V)
    // 給予初始值
    int min = INT_MAX
    int v = DEFAULT_VALUE
    // 迭代尋找最小 key 值
    for i = 0 to V
        // 在尚未放進 MST 的 vertex 中
        // 找最小 key 值，即最小 weight
        if (visited_set[i] == FALSE and key_set[i] < min)
            // 發現更小的則取代掉
            min = key_set[i]
            // 記錄該 vertex
```

```
                v = i
        // 回傳 vertex
        return v

// 找出 MST
PRIM(V) {

        // 逐一將 vertex 放入 MST
        // 從 vertex 0 開始找
        for i = 0 to V
                // 取出與目標 vertex 擁有最小 weight 的 vertex
                int u = EXTRACT_MIN(V)
                // 將此 vertex u 放入 MST 中
                visited_set[u] = TRUE
                // 找出所有與 u 相連的 edge
                for v = 0 to V
                        // 將 v 的 key 與所有相連 edge 之 weight 作比較
                        // 若 weight 較小
                        // 表示從 u 經由此 edge 到達 v，需要此 weight 的成本
                        // 反之，若 weight 較大，則不更新
                        // 表示從其他 vertex 到達 v 的成本會更小
                        if (visited_set[v] == FALSE
                                and edge_set[u][v] != DEFAULT_VALUE
                                and edge_set[u][v] < key_set[v])
                                // 更新 v 的 parent 為 u
                                parent_set[v] = u
                                // 更新 v 的 key 值為該 weight
                                key_set[v] = edge_set[u][v]
```