Q1

(i) source code:

```java
import java.util.PriorityQueue;

   /************************ Begin of Kruskal ***************************/
public class mst {
   public static void main(String[] args){
      int size[] = new int[]{10, 100, 500, 1000};
      long beginTime, endTime;

      int cnt = 20;

      for (int i = 0; i < size.length; i++) {
         mst mst = new mst();
         mst.init(size[i]);

         double minWeght = 0;
         long useTime = 0;

         for (int j = 0; j < cnt; j++) {
            beginTime = System.nanoTime();
            minWeght = minWeght + mst.kruskal();
            endTime = System.nanoTime();
            useTime += (endTime - beginTime);
         }

         System.out.println("kruskal -- Size : " + size[i] +
               "; Arerage Running Time : " + (useTime/cnt) +
               "; Arerage Weight : " + minWeght / cnt);

         minWeght = 0;
         useTime = 0;
         for (int j = 0; j < cnt; j++) {
            beginTime = System.nanoTime();
            minWeght = minWeght + mst.prim();
            endTime = System.nanoTime();
            useTime += (endTime - beginTime);
         }

         System.out.println("Prim   -- Size : " + size[i] +
               "; Arerage Running Time : " + (useTime/cnt) +
               "; Arerage Weight : " + minWeght / cnt);

         System.out.println();
```

```java
        }
    }

    private double[][] graph;
    private int N;
    private int[] father;

    class Edge implements Comparable<Edge>
    {
        int from;
        int to;
        double w;
        Edge next;
        public Edge(int from, int to, double w) {
            this.from = from;
            this.to = to;
            this.w = w;
        }
        public Edge(){}
        @Override
        public int compareTo(Edge o) {
            return w > o.w ? 1 : -1;
        }
    }

    //The initialization of the graph
    public void init(int n){
        N = n;
        graph = new double[n][n];
        for (int i = 0; i < n; i++) {
            for (int j = i + 1; j < n; j++) {
                graph[i][j] = graph[j][i] = Math.random();

            }
        }
    }

    /******************************** Begin of quick sort
***********************************/
    void quicksort(Edge arr[]){
        quicksort(arr, 0, arr.length -1);
    }

    void quicksort(Edge arr[], int left, int right) {
```

```
        int dp;
        if (left < right) {
            dp = partition(arr, left, right);
            quicksort(arr, left, dp - 1);
            quicksort(arr, dp + 1, right);
        }
    }

    int partition(Edge arr[], int left, int right) {
        Edge pivot = arr[left];
        while (left < right) {
            while (left < right && arr[right].w >= pivot.w)
                right--;
            if (left < right)
                arr[left++] = arr[right];
            while (left < right && arr[left].w <= pivot.w)
                left++;
            if (left < right)
                arr[right--] = arr[left];
        }
        arr[left] = pivot;
        return left;
    }
    /******************************** End of quick sort
*******************************/

    /************ Begin of Directed forest for disjoint sets
*************************/
    //Search
    public int find(int v)
    {
        if (father[v] != v)
        {
            father[v] = find(father[v]);
        }
        return father[v];
    }

    //Merge
    public double join(Edge e)
    {
        int x, y;
        x = find(e.from);
        y = find(e.to);
```

```java
        //System.out.print(e.from + ", " + e.to);

        if (x != y)
        {
            //System.out.println("true");
            father[x] = y;
            return e.w;
        }
        //System.out.println("false");
        return 0;
    }
    /*************** End of Directed forest for disjoint sets
*******************/

    public double kruskal()
    {
        double ans = 0;

        father = new int[N];
        for (int i = 0; i < N; i++) {
            father[i] = i;
        }

        Edge edges[] = new Edge[N*(N-1)/2];
        //init edge
        edges = new Edge[N*(N-1)/2];
        int k = 0;
        for (int i = 0; i < N; i++) {
            for (int j = i + 1; j < N; j++) {
                edges[k++] = new Edge(i, j, graph[i][j]);
            }
        }

        //sort the edge by weight
        quicksort(edges);

        for (int i = 0; i < edges.length; ++i)
        {
            ans += join(edges[i]);
        }

        return ans;
    }
```

```java
    /******************************* End of Kruskal
**********************************/

    /******************************* Begin of prim
***********************************/
    private Edge p[];

    public void addEdge(int from, int to, double w) {
        Edge e = new Edge();
        e.from = from;
        e.to = to;
        e.w = w;
        e.next = p[from].next;
        p[from].next = e;
    }

    public double prim() {
        p = new Edge[N];

        double dis[] = new double[N];
        boolean vis[] = new boolean[N];
        double ans = 0;

        for(int i = 0; i < N; i++) {
            p[i] = new Edge();
            dis[i] = Double.MAX_VALUE;
            vis[i] = false;
        }

        for (int i = 0; i < N; i++) {
            for (int j = 0; j < N; j++) {
                if (i != j) {
                    addEdge(i, j, graph[i][j]);
                }
            }
        }

        PriorityQueue<Edge> q = new PriorityQueue<Edge>();

        Edge edge = p[0].next;
        while(edge != null){
            q.add(edge);
            edge = edge.next;
        }
```

```
        vis[0] = true;

        while(!q.isEmpty()) {
            Edge t = q.poll();
            if(vis[t.to]){
                continue;
            }
            vis[t.to] = true;
            ans += t.w;

            edge = p[t.to].next;
            while(edge != null){
                if (!vis[edge.to]) {
                    q.add(edge);
                }
                edge = edge.next;
            }
        }

        return ans;
    }
}
    /******************************** End of prim
**********************************/
```

(ii) In my opinion, the value of L(n) should be approximately equal to $w(n-1)$, where w is the average weight of edges. However, actually, L(n) does not grow by following this formula because the weights of edges in an MST are always less than that of non-tree edges. And this situation causes that the growth of L(n) is increasingly slower. When n grows sharply, it can be observed that $L(n) \leq 1.25$.

(iii)

```
/Library/Java/JavaVirtualMachines/jdk1.8.0_91.jdk/Contents/Home/bin/java ...
kruskal -- Size : 10; Arerage Running Time : 127050; Arerage Weight : 1.0881411151356137
Prim    -- Size : 10; Arerage Running Time : 179507; Arerage Weight : 1.0881411151356137

kruskal -- Size : 100; Arerage Running Time : 1883166; Arerage Weight : 1.3515459275273471
Prim    -- Size : 100; Arerage Running Time : 3206522; Arerage Weight : 1.3515459275273474

kruskal -- Size : 500; Arerage Running Time : 57937065; Arerage Weight : 1.2246000766420306
Prim    -- Size : 500; Arerage Running Time : 96102922; Arerage Weight : 1.2246000766420313

kruskal -- Size : 1000; Arerage Running Time : 165086257; Arerage Weight : 1.216840944053906
Prim    -- Size : 1000; Arerage Running Time : 410017116; Arerage Weight : 1.2168409440539054


Process finished with exit code 0
```

(iv)

With the growth on the number of vertices n, the running time trends of Prim's algorithm grows slow while Kruskal's grows fast and faster than Prim's algorithm.

The reasons for this circumstance are following 4 points, which are also the difference between Prim's algorithm and Kruskal's algorithm:

1.Prim's algorithm initializes with a node, whereas Kruskal's algorithm initiates with an edge.
2.Prim's algorithms span from one node to another while Kruskal's algorithm select the edges in a way that the position of the edge is not based on the last step.
3.In prim's algorithm, graph must be a connected graph while the Kruskal's can function on disconnected graphs too.
4.Prim's algorithm has a time complexity of $O(V^2)$, and Kruskal's time complexity is $O(\log(V))$.