Name: Jiajun Wang
UFID: 93187471
Email: wjj@ufl.edu

**Language: Java

**Compile:

```
-------------------------
javac MST.java
java -Xmx2G MST -r n d

n is number of nodes in the Graph
d is density of the Graph

-------------------------
javac MST.java
java -Xmx2G MST -s filename
---------------------------
Note: you should copy the file into the project folder.
```

**Structure:

```
1. default:
   MST.java                 //core class which concludes prim algorithm.
2. DataStructure
   FibonacciHeap.java       //Fibonacci Heap data structure
   FibonacciHeapNode.java   //util for Fibonacci Heap
   MinPQ.java               //util for random generating Graph.
3. Graph
   Edge.java
   EdgeWeightedGraph.java
4. Util
   GraphFactory.java        //util for random generating Graph
```

**Prototypes:

```
MST.java
//two types of algorithm: Simple and Fibonacci Heap
public static enum SchemType
{
    SIMPLE_SCHEME, F_HEAP_SCHEME;
}

//constructor of MST
public MST(EdgeWeightedGraph edgeWeightedGraph, SchemType type);
```

```
//Simple algorithm method.
private void SiPrim(EdgeWeightedGraph edgeWeightedGraph, int i);

//get min vertice from Simple structure
private void SiScan(EdgeWeightedGraph edgeWeightedGraph, int v);

//specific get min method, and called by SiScan
private int getMin(HashMap<Integer, Integer> arrayList);

//Fibonacci Heap algorithm
private void FiPrim(EdgeWeightedGraph edgeWeightedGraph, int s);

private void FiScan(EdgeWeightedGraph edgeWeightedGraph,
FibonacciHeapNode<Integer> node);

//print edges
public Iterable<Edge> edges();

//get total weight..
public int weight();

//check whether two algorithm has the same result…
private static boolean isEqual(MST mstFi, MST mstSi);

//CORE: Main method.
public static void main(String[] args);

FibonacciHeap.java:
//constructor
public FibonacciHeap()

//whether this heap is empty
public boolean isEmpty()

//reduce the weight value of this node
public void decreaseKey(FibonacciHeapNode<T> node, int key)

//insert a node into Heap
public void insert(FibonacciHeapNode<T> node, int key)


//check whether this heap contains this node..
```

```
public boolean contains(FibonacciHeapNode<T> heapNode)

//get min node from heap
public FibonacciHeapNode extractMin()

//add child nodes to root level
private FibonacciHeapNode addToRoot(FibonacciHeapNode node1,
FibonacciHeapNode node2)

//used in extractMin, combine the heap with same degree…
private void pairwiseCombine()

//used in pairwiseCombile(), combine heap nodes
private void reconnectHeap(HashMap<Integer, FibonacciHeapNode>
rootNodes)

//check childCut and cut ..
private void cascadingCut(FibonacciHeapNode y)

private void cut(FibonacciHeapNode x, FibonacciHeapNode y)

//meld to heap with same degree..
private FibonacciHeapNode meld(FibonacciHeapNode parent,
FibonacciHeapNode child)

FibonacciHeapNode.java
//constructor of FibonacciHeapNode
public FibonacciHeapNode(T index)

//get weight of this node
public final int getWeight()

//get index of this node
public final T getIndex()

Edge.java
//constructor
public Edge(int v, int w, int weight)

//constructor
public Edge(int v, int w)

//get weight of this edge
```

```
public int weight()

//get one point of this edge
public int either()

//get another one point of this edge..
public int other(int vertex)

//compare two edges..
public int compareTo(Edge edge)

//check equals
public boolean equals(Object e)

//print edge
public String toString()

EdgeWeightedGraph.java
//constructor
public EdgeWeightedGraph(int V)

//constructor
public EdgeWeightedGraph(int V, int E)

//constructor: read from file
public EdgeWeightedGraph(String fileName)

//get V
public int V()

//get Edge
public int E()

//add an edge to the graph
public void addEdge(Edge e)

//add an random edge to the graph
public void addEdge(int v, int w)

//get edges
public Iterable<Edge> adj(int v)
```

```
GraphFactory.java
//get a tree
public static EdgeWeightedGraph tree(int V)

//get a simple connect graph
public static EdgeWeightedGraph simple(int V, int E)

//get a simple connect graph with density
public static EdgeWeightedGraph simple(int V, double density)
```

**Summary:
Assumption:
For what I learned from class, Fibonacci Heap will run faster than Simple Array if the Graph is sparse. If the graph is dense, simple graph will run little faster than Fibonacci Heap.

Here is the Data using my code:

1000	Simple Array	Fibonacci Heap
10%	54	108
20%	121	187
30%	107	218
40%	88	331
50%	93	534
60%	98	673
70%	120	493
80%	228	577
90%	119	779

3000	Simple Array	Fibonacci Heap
10%	265	759
20%	624	1325
30%	1769	2020
40%	1024	2784
50%	626	4270
60%	3251	4364
70%	4041	5017
80%	886	5914
90%	1034	6658

5000	Simple Array	Fibonacci Heap
10%	1788	1830
20%	3240	3880
30%	1717	6215
40%	2920	8256
50%	7279	11313
60%	2119	13965
70%	2456	17067
80%	2898	19622
90%	6812	23146

**Result Analysis:
the reason why Simple array run faster than Fibonacci heap because I improved Simple array algorithm. For example, getMin() from Simple Array, I skip the unnecessary searching. This will reduce much time. Also for check contains(), I also reduce the time for searching.

Also, for Fibonacci Heap there are much reference. I think much reference changed will also causes the delay.

But for general way, it will work worse than Fibonacci Heap.