[I submit this pdf instead of the writeup.txt because its easier to edit.]

QUESTION 1:

```
jiacheng@ada:~/CS320/lab7/11$ javac Example.java
jiacheng@ada:~/CS320/lab7/11$ java Example 2000000000
2000000000 > -294967296
jiacheng@ada:~/CS320/lab7/11$ g++ -O0 -o example example.cpp
jiacheng@ada:~/CS320/lab7/11$ ./example 2000000000
2000000000 > -294967296
jiacheng@ada:~/CS320/lab7/11$ g++ -O2 -o example example.cpp
jiacheng@ada:~/CS320/lab7/11$ ./example 2000000000
2000000000 <= -294967296
```

Sudocode: of if statement

```
if(a<=b) -> a <= b
    else -> a > b
```

Explanation: because of int b = a + 2000000000; when a = 2000000000, b should be 4000000000. In 32 bits system the memory range is (-214748648, 214748647). However b has been overloaded. It has 2 way to display the relationship of a and b. The result is listed above, The -O0 flag complier provide a > b, and -O2 flag compiler provide a <=b. Its means -O0 is not fix the mathematical solution which the sudocode if statement shows above. When b gets value of 4000000000, its already beyond the range, and the value will be altered, the value is -294967296 but the compile time. The compiler will provide the result base on this value of b. In -O2 compiler. The UB is ignored, and the error is optimized. The compiler will provide “wraps around” method to deal with overflow. The -O2 compiler translates a mathematical trustable program to interpreter, and provide more mathematical correction solution.

QUESTION 2:

1 For arrays, prepending is much slower than appending.

```
=======================
TWO SETs & actual timing:
jiacheng@ada:~/CS320/lab8/04X$ time -p java IntSequenceTimer
array 10000 5000 0 500
real 0.73
user 0.39
sys 0.21

jiacheng@ada:~/CS320/lab8/04X$ time -p java IntSequenceTimer
array 10000 0 5000 500
real 29.06
user 28.90
sys 0.25
```

This times 10000 repetitions of a test in which we append 5000 integers, prepend 0 integers, and read 5000 integers from the middle of the sequence. Comparing with times 10000 repetitions of a test in which we append 0 integers, prepend 5000 integers, and read 500 integers from the middle of the sequence. Both for array

Expectation: I expect the time of prepending should be 5 times(or more) slower than time of appending. Because array is hard to pre-appending than appending.

Analysis: True. The time shows above, prepending time is 41X which more than !0X.

2 For lists, prepending and appending are roughly the same speed.

```
=======================
TWO SETs & actual timing:
jiacheng@ada:~/CS320/lab8/04X$ time -p java IntSequenceTimer
list 10000 0 5000 500
real 23.27
user 23.18
sys 0.26

jiacheng@ada:~/CS320/lab8/04X$ time -p java IntSequenceTimer
list 10000 5000 0 500
real 23.56
user 23.43
sys 0.26
```

This times 10000 repetitions of a test in which we append 0 integers, prepend 5000 integers, and read 5000 integers from the middle of the sequence. Comparing with times 10000 repetitions of a test in which we append 5000 integers, prepend 0 integers, and read 500 integers from the middle of the sequence. Both for list.

Expectation: I expect the time of prepending should be roughly same time of appending. Because it's have no much different for list to prepending and and appending.

Analysis: True. The time shows above, both appending and prepending show roughly same time around 23s.

3 Prepending is much faster in the list implementation than in the array implementation.

```
=======================
TWO SETs & actual timing:
jiacheng@ada:~/CS320/lab8/04X$ time -p java IntSequenceTimer
list 10000 0 5000 500
real 25.14
user 24.85
sys 0.22
```

```
jiacheng@ada:~/CS320/lab8/04X$ time -p java IntSequenceTimer
array 10000 0 5000 500
real 29.72
user 29.42
sys 0.30
```

This times 10000 repetitions of a test in which we append 0 integers, prepend 5000 integers, and read 5000 integers from the middle of the sequence for list. Comparing with times 10000 repetitions of a test in which we append 0 integers, prepend 5000 integers, and read 500 integers from the middle of the sequence for array.

Expectation: My expectation is list prepending is faster than array prepending. Because list do prepending easier than array.

Analysis: False. The time shows above, list is not <u>much</u> faster than array. Because the time of list is 4s faster which only 1.2 times faster than array. It's not satisfy 5 times and more in lab8 statement.

4 Appending is roughly the same speed for both implementations.

```
=======================
TWO SETs & actual timing:
jiacheng@ada:~/CS320/lab8/04X$ time -p java IntSequenceTimer
array 10000 5000 0 500
real 0.62
user 0.45
sys 0.27

jiacheng@ada:~/CS320/lab8/04X$ time -p java IntSequenceTimer
list 10000 5000 0 500
real 26.73
user 25.74
sys 0.24
```

This times 10000 repetitions of a test in which we append 5000 integers, prepend 0 integers, and read 5000 integers from the middle of the sequence for array. Comparing with times 10000 repetitions of a test in which we append 5000 integers, prepend 0 integers, and read 500 integers from the middle of the sequence for list.

Expectation: My expectation is array appending is faster than list appending. Because array do appending easier than array.

Analysis: False. The time shows above, list is 43 times slower than array when do appending. It's not roughly same.

QUESTION 3:

a)

AdjSetGraph's constructor -> AdjSetDirectedGraph's constructor

```
public void addEdge(V v1, V v2) {
  Set<V> adjs1 = adjacents(v1);   // Check that v1 is in the 
graph
  Set<V> adjs2 = adjacents(v2);   // Check that v2 is in the 
graph
  adjs1.add(v2);                  // Add an edge from v1 to v2
}
```

The biggest different part between is in addEdge function. Due to in AdjSetGrpah is represents undirected graphs. Java has add an edge from v2 to v1 (if they are distinct) which is adjs2.add(1). But, in AdjSetDirectedGraph will represent directed graph. This means that adding an edge from vertex v1 to v2 does not automatically imply that there will also be an edge from v2 to v1.

I have changed the constructor's name to AdjSetDirectedGrpah, and addEdge function.

b) I have updated the implementation of emptyDirectedGraph() in GraphUtils.java to invoke the constructor for AdjSetDirectedGraph. Which return new AdjSetDirectedGraph<V>() rather than AdjSetGraph<V>()

```
static <V> Graph<V> emptyDirectedGraph() {
  return new AdjSetDirectedGraph<V>();
}
```

c) I have completed the implementation of the flip() method in GraphUtils.java. The function construct a flipped version of the directed graph that is passed in as an input.

```
public static <V> Graph<V> flip(Graph<V> g) {
   Graph<V> flipped = emptyDirectedGraph();
   for(V v: g.vertices())
   {
      flipped.addVertex(v);
   }
   for(V v: g.vertices())
   {
     for(V w: g.neighbors(v)){
       flipped.addEdge(w,v);
     }
   }
   return flipped; }
```

For test, I change the code in SccTest.java

```
System.out.print(GraphUtils.dumpGraph(g)); to
System.out.print(GraphUtils.dumpGraph(GraphUtils.flip(g)));
```

Which the result will be flipped g.

```
[original]
a: b
b: c d
c: b
d: e
e: f
f: d g
g:

[after flipped]
a:
b: a c
c: b
d: b f
e: d
f: e
g: f
```

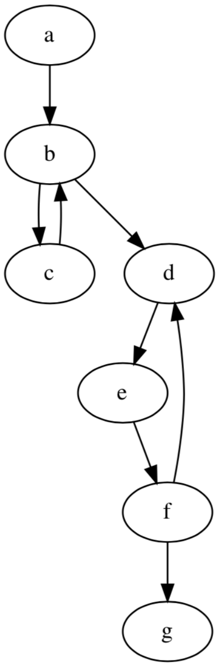
Its easy to see for example a->b become b->a, e->f become f->e. The graph is flipped.

d) I Completed the implementation of the scc() method in GraphUtils.java.

```
public static <V> List<List<V>> scc(Graph<V> g) {
  List<V> finished;
  finished = dfs(g);
  System.out.print(“DFS order: " + finished);
  Collections.reverse(finished);
  Graph<V> flipped;
  flipped = flip(g);
  Set<V> visited = new HashSet<V>();
  List<List<V>> sccs = new LinkedList<List<V>>();

  for(V v:finished){
    if(visited.contains(v) == false){
       List<V>scc = new LinkedList<V>();
       visitscc(v, scc, visited, flipped);
       sccs.add(scc);
    }
  }
  return sccs;}

public static <V> void visitscc(V v, List<V> scc, Set<V> visited,
Graph<V> flipped){
      visited.add(v);
      scc.add(v);
      for(V w:flipped.neighbors(v)){
    if(!visited.contains(w)){
     visitscc(w, scc, visited, flipped);
    }
      }
   }

public static <V> List<V> dfs(Graph<V> g){
     Set<V> visited = new HashSet<V>();
     List<V> finished = new LinkedList<V>();
     for(V v:g.vertices()){
          visit(v,visited,g,finished);
}
return finished; }

public static <V> void visit(V v, Set<V> visited, Graph<V> g, List<V>
finished){
     if(!visited.contains(v)){
          visited.add(v);
          for(V w: g.neighbors(v)){
               visit(w, visited, g, finished);
          }
     finished.add(v);}
}
```

Beside scc() function, I also created visitscc(), dfs(), and visit() function to help scc() find strongly connected component.

For test, I use SccTest.java which already have implemented code. And I use this command: dot -Tpdf -o graph.pdf graph.dot to generate graph.pdf and graph.dot.
I got “a” -> “b”; “a” -> “o”; “b” -> “c”; “b” -> “d”; “c” -> “b”; “d” -> “e”; “e” -> “f”; “f” -> “d”; “d” -> “g”; in graph.dot and graph.pdf shows below.

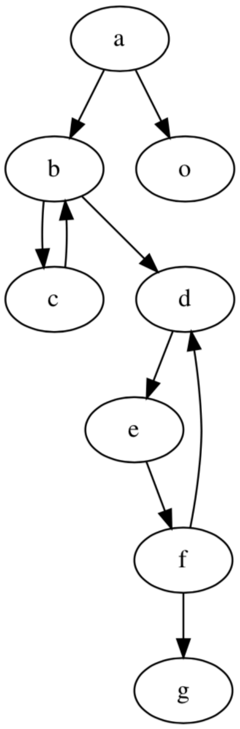


```
Vertex count = 7
a: b
b: c d
c: b
d: e
e: f
f: d g
g:
DFS order: [c, g, f, e, d, b, a]
-----
Number of strongly connected components = 4
strongly connected component: a
strongly connected component: b c
strongly connected component: d f e
strongly connected component: g
-----
```

There are 7 vertexes and 4 strongly connected components which are [a], [bc], [dfe]and [g].

After that I edited code in SccTest to add one more strongly connected component. I added g.addVertex(“o"); g.addEdge("a", “o"); to add one more vertex o and one edge form a to o. I got new result in graph.dot which appear one more line “a” -> “o”; in digraph. I got new graph which appear one more vertex o and one edge form a to o.

My expectation is vertex count become to 8 and strongly connected components become to 5 which one [o] SCC is added.



```
Vertex count = 8
a: b o
b: c d
c: b
d: e
e: f
f: d g
g:
o:
DFS order: [c, g, f, e, d, b, o, a]
-----
Number of strongly connected components = 5
strongly connected component: a
strongly connected component: o
strongly connected component: b c
strongly connected component: d f e
strongly connected component: g
-----
```

As what I expect, there are 8 vertexes and 5 strongly connected components which are [a], [o], [bc], [dfe]and [g].