M4: Programming Assignment

Objective of this assignment:
Implement Dijkstra's algorithm

- USE THIS FILE AS THE STARTING DOCUMENT YOU WILL TURN IN. **DO NOT DELETE ANYTHING FROM THIS FILE:** JUST **INSERT** EACH ANSWER **RIGHT AFTER ITS QUESTION/PROMPT**.

- IF USING HAND WRITING (STRONGLY DISCOURAGED), **USE THIS FILE** BY CREATING SUFFICIENT SPACE AND WRITE IN YOUR ANSWERS.
- FAILING TO FOLLOW TURN IN DIRECTIONS /GUIDELINES WILL COST **A 30% PENALTY.**

What you need to do:

Complete the assigned these three tasks:
1) Implement **Dijkstra's algorithm**
2) Compile, execute and take a screenshot of a successful execution using a provided input graph called `graph.txt`
3) Compile, execute and take a screenshot of a successful execution using your own input graph called `myInputGraph.txt`

Objective of this assignment:

- Implement Dijkstra's algorithm

What you need to do:

1. **Ask questions if you have any doubt**
2. Implement Dijkstra's algorithm
3. Allow a user to provide a *graph* and a *source* vertex *s* as a text file.
4. Display the shortest path for *s* to every other vertex in the graph.
5. Output a file text describing the shortest path for s to every other vertex in the graph
6. **Ask questions if you have any doubt**

Objective:
The objective is to implement the Dijkstra's algorithm.

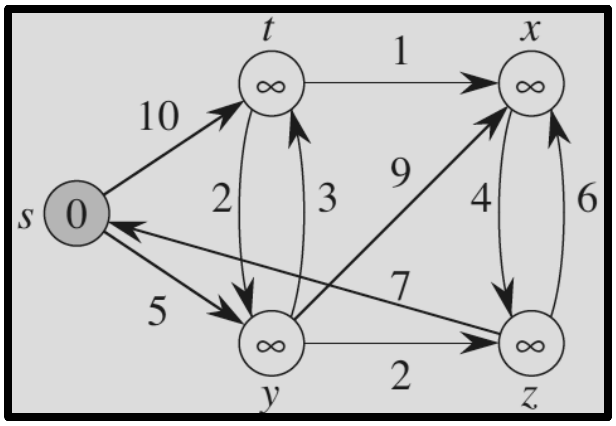
Input:
Your program must prompt the user to enter the name of an input file text. The input will be a text file describing the graph. For simplicity, vertices will be identified using only characters "a, b, ..., A, .., Z". The weights on the edges will be positive integers from 1 to 32,767.
A graph and the source *s* are provided in the text file on Canvas under the name *graph.txt* following this format:

```
s
t x,1 y,2
s t,10 y,5
y z,2 t,3 x,9
x z,4
z x,6 s,7
```

The first line provides the **source** vertex. For this example, Node `s` is the source. The remaining lines provide the adjacency list. The leftmost column is the list of vertices (here, vertices are s, t, y, x, z). For each vertex, the edges are provided as pairs `node, weight`. For example, Vertex `t` has two edges:
- the first edge is `x,1` meaning the edge (t,x) with weight 1
- the second edge is `y,2` meaning the edge (t,y) with weight 2

A node not connected to any other vertex will be listed with no edge(s).

The above input text file `graph.txt` (available on Canvas with this assignment) represents the graph below with Vertex `s` as the source:



Output:

If the Dijkstra's completes **successfully**, the output will provide the shortest path from the source to each vertex in the graph other than the source. You must display the output and write it to a text file under this format:

```
t: y t
y: y
x: y t x
z: y z
```

The first column is the list of nodes in the graph (except the source) in the order used in graph.txt. For each vertex, the shortest path is provided as a list of vertices describing the path. For example: the shortest path from `s` to `x` is the path `s y t x` (see output file above)

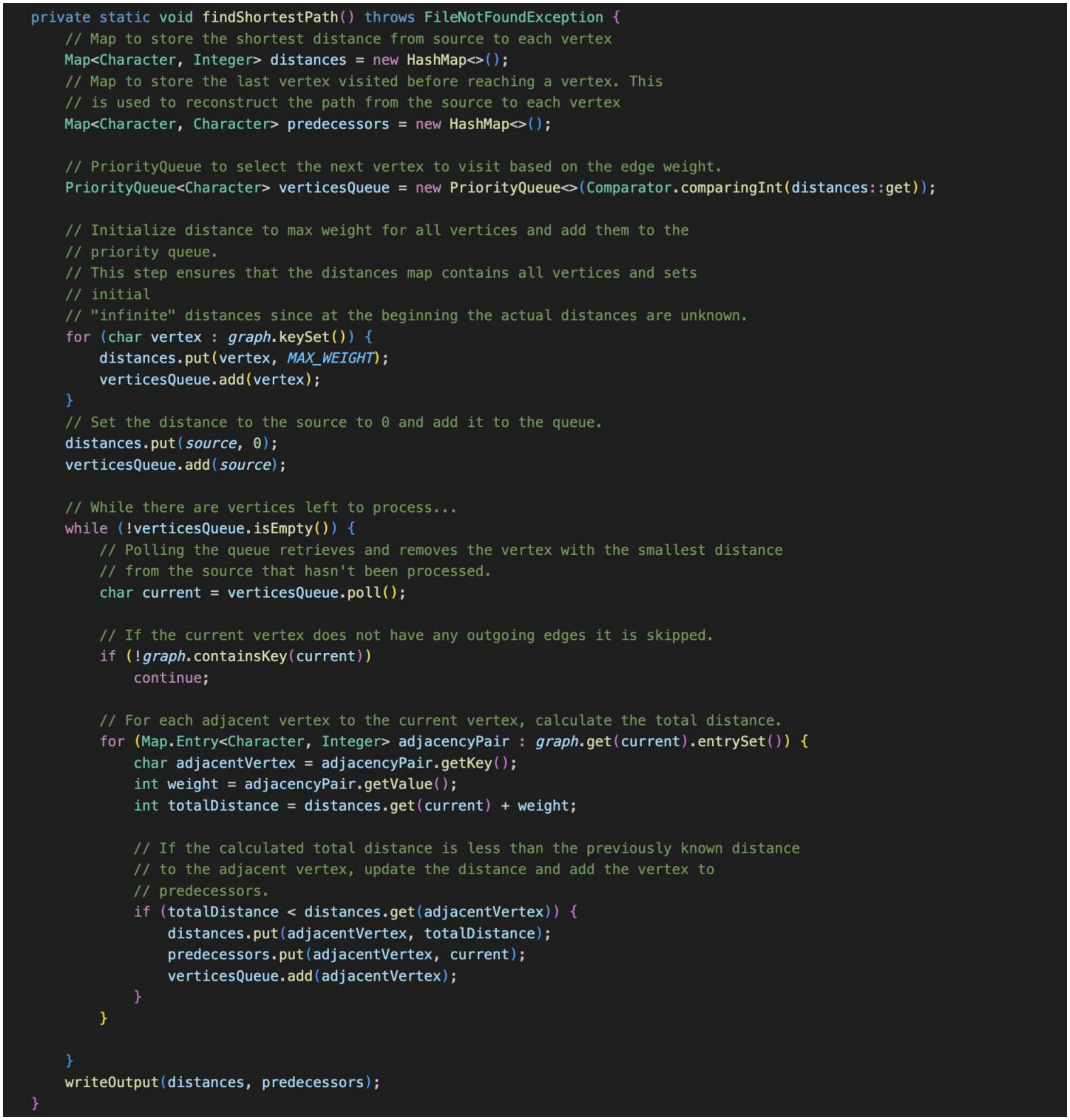
Programming

You can implement the Dijkstra's algorithm in your preferred language as long as it is already available on Engineering Tux machines. Insure that your program compiles and executes correctly on Tux machines.

Tasks:

Task I

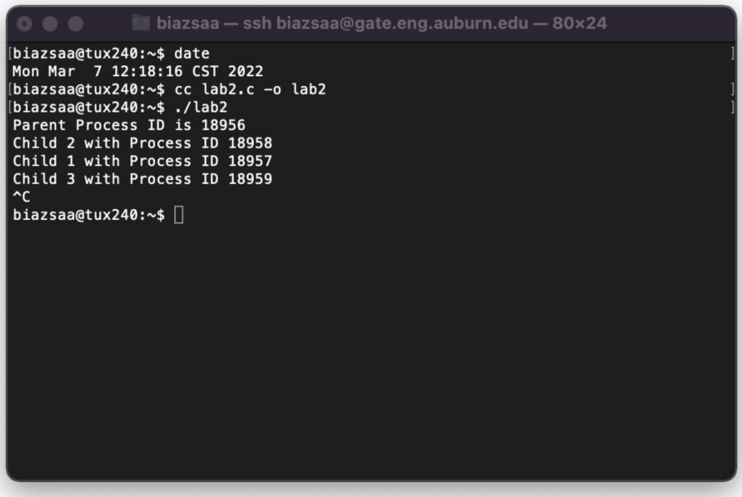
Implement the Dijkstra's algorithm



```java
private static void findShortestPath() throws FileNotFoundException {
    // Map to store the shortest distance from source to each vertex
    Map<Character, Integer> distances = new HashMap<>();
    // Map to store the last vertex visited before reaching a vertex. This
    // is used to reconstruct the path from the source to each vertex
    Map<Character, Character> predecessors = new HashMap<>();

    // PriorityQueue to select the next vertex to visit based on the edge weight.
    PriorityQueue<Character> verticesQueue = new PriorityQueue<>(Comparator.comparingInt(distances::get));

    // Initialize distance to max weight for all vertices and add them to the
    // priority queue.
    // This step ensures that the distances map contains all vertices and sets
    // initial
    // "infinite" distances since at the beginning the actual distances are unknown.
    for (char vertex : graph.keySet()) {
        distances.put(vertex, MAX_WEIGHT);
        verticesQueue.add(vertex);
    }
    // Set the distance to the source to 0 and add it to the queue.
    distances.put(source, 0);
    verticesQueue.add(source);

    // While there are vertices left to process...
    while (!verticesQueue.isEmpty()) {
        // Polling the queue retrieves and removes the vertex with the smallest distance
        // from the source that hasn't been processed.
        char current = verticesQueue.poll();

        // If the current vertex does not have any outgoing edges it is skipped.
        if (!graph.containsKey(current))
            continue;

        // For each adjacent vertex to the current vertex, calculate the total distance.
        for (Map.Entry<Character, Integer> adjacencyPair : graph.get(current).entrySet()) {
            char adjacentVertex = adjacencyPair.getKey();
            int weight = adjacencyPair.getValue();
            int totalDistance = distances.get(current) + weight;

            // If the calculated total distance is less than the previously known distance
            // to the adjacent vertex, update the distance and add the vertex to
            // predecessors.
            if (totalDistance < distances.get(adjacentVertex)) {
                distances.put(adjacentVertex, totalDistance);
                predecessors.put(adjacentVertex, current);
                verticesQueue.add(adjacentVertex);
            }
        }

    }
    writeOutput(distances, predecessors);
}
```

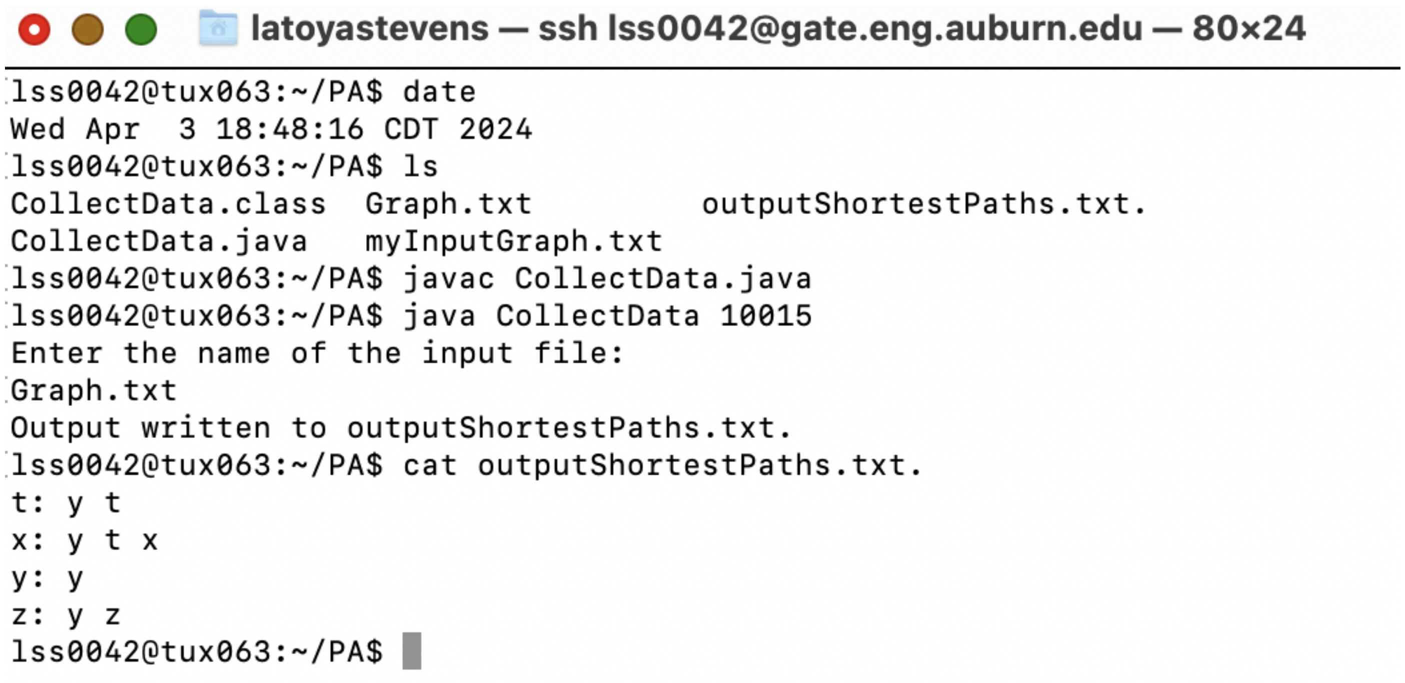
Task 2:

- Log on a Tux machine
- Type the date command
- compile your program
- Execute your program using the provided input file grah.txt.
- Display the shortest path under the provided format
- Take a readable screenshot showing the date, your username, the tux machine name, the compilation directions, the execution and the shortest path. The screenshot must be as readable as this template screenshot:



```
[biazsaa@tux240:~$ date
Mon Mar  7 12:18:16 CST 2022
[biazsaa@tux240:~$ cc lab2.c -o lab2
[biazsaa@tux240:~$ ./lab2
Parent Process ID is 18956
Child 2 with Process ID 18958
Child 1 with Process ID 18957
Child 3 with Process ID 18959
^C
biazsaa@tux240:~$
```

Insert your screenshot here

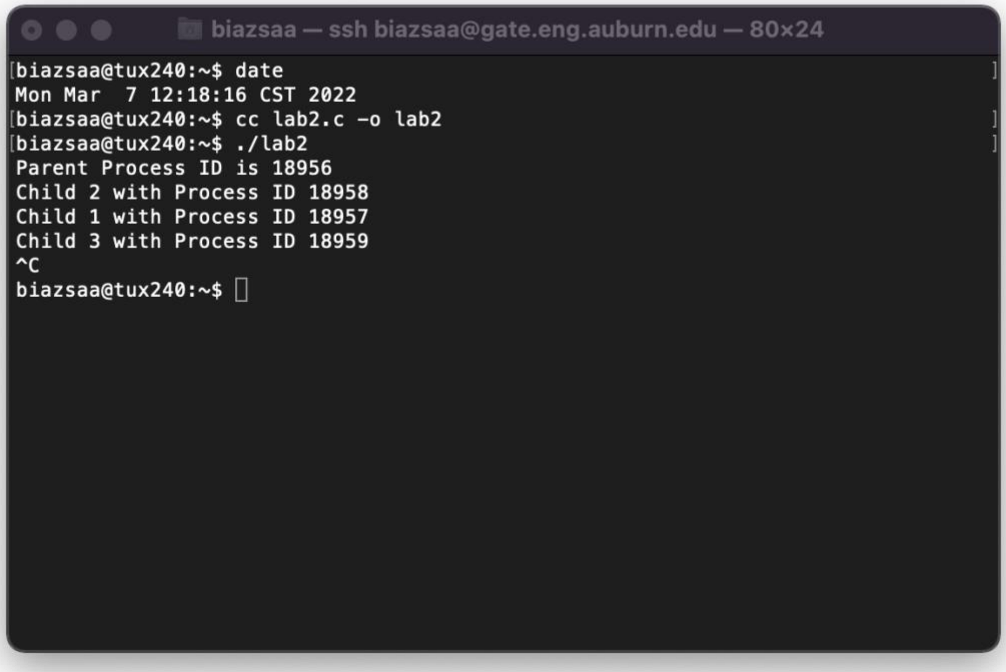


latoyastevens — ssh lss0042@gate.eng.auburn.edu — 80×24

```
lss0042@tux063:~/PA$ date
Wed Apr  3 18:48:16 CDT 2024
lss0042@tux063:~/PA$ ls
CollectData.class  Graph.txt         outputShortestPaths.txt.
CollectData.java   myInputGraph.txt
lss0042@tux063:~/PA$ javac CollectData.java
lss0042@tux063:~/PA$ java CollectData 10015
Enter the name of the input file:
Graph.txt
Output written to outputShortestPaths.txt.
lss0042@tux063:~/PA$ cat outputShortestPaths.txt.
t: y t
x: y t x
y: y
z: y z
lss0042@tux063:~/PA$ 
```

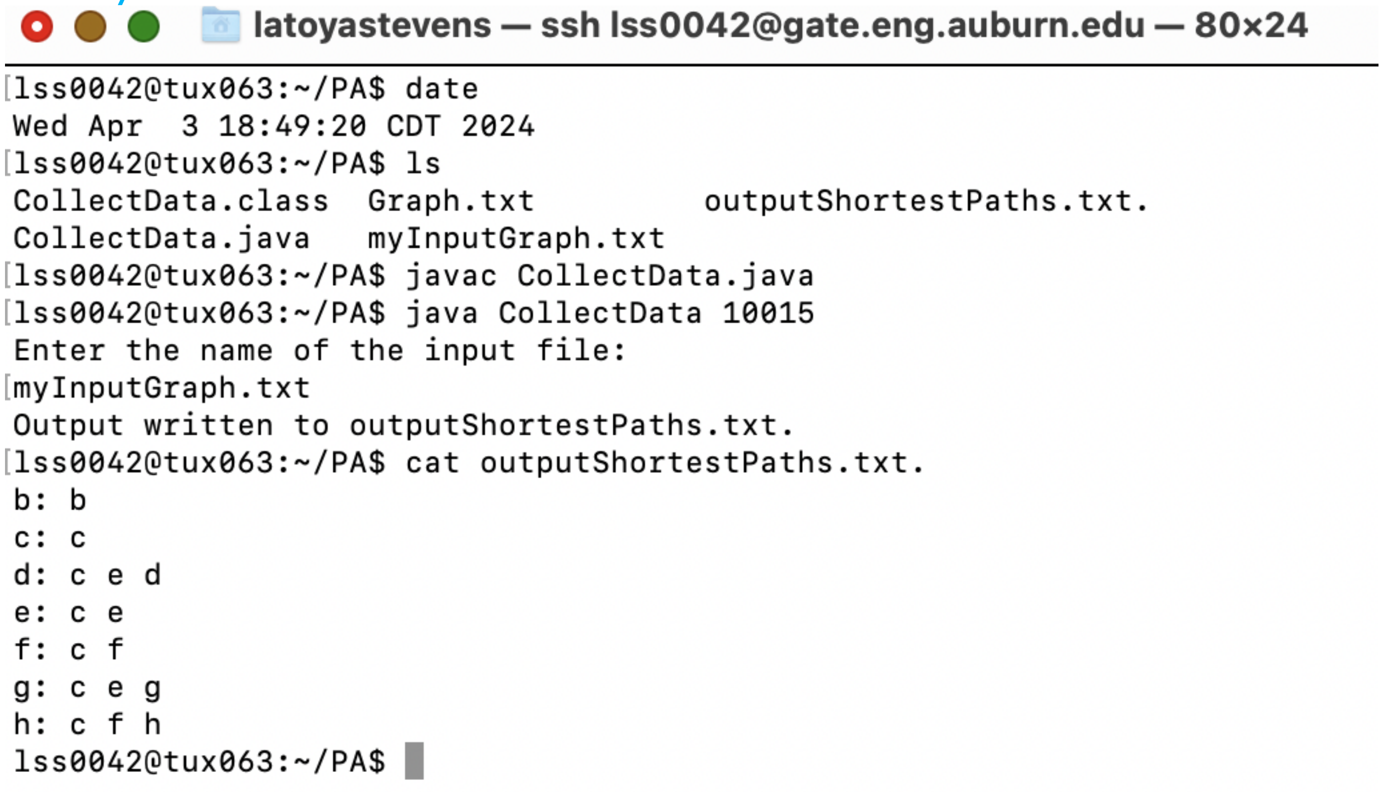
Task 3:

- Prepare your sample input graph file. This sample graph file (with .txt extension) must have at least 8 vertices and 16 edges. Call this file `myInputGraph.txt`. The output text file of your program must be called `outputShortestPaths.txt`,
- Log on a Tux machine
- Type the date command
- compile your program
- Execute your program using your input file `myInputGraph.txt`.
- Display the shortest path under the provided format
- Submit the files `myInputGraph.txt` and `outputShortestPaths.txt`.
- Take a readable screenshot showing the date, your username, the tux machine name, the compilation directions, the execution and the shortest path. The screenshot must be as readable as this template screenshot:



```
biazsaa — ssh biazsaa@gate.eng.auburn.edu — 80×24
biazsaa@tux240:~$ date
Mon Mar  7 12:18:16 CST 2022
biazsaa@tux240:~$ cc lab2.c -o lab2
biazsaa@tux240:~$ ./lab2
Parent Process ID is 18956
Child 2 with Process ID 18958
Child 1 with Process ID 18957
Child 3 with Process ID 18959
^C
biazsaa@tux240:~$
```

Insert your screenshot here



```
latoyastevens — ssh lss0042@gate.eng.auburn.edu — 80×24
lss0042@tux063:~/PA$ date
Wed Apr  3 18:49:20 CDT 2024
lss0042@tux063:~/PA$ ls
CollectData.class  Graph.txt          outputShortestPaths.txt.
CollectData.java   myInputGraph.txt
lss0042@tux063:~/PA$ javac CollectData.java
lss0042@tux063:~/PA$ java CollectData 10015
Enter the name of the input file:
myInputGraph.txt
Output written to outputShortestPaths.txt.
lss0042@tux063:~/PA$ cat outputShortestPaths.txt.
b: b
c: c
d: c e d
e: c e
f: c f
g: c e g
h: c f h
lss0042@tux063:~/PA$
```

Grading:

- The program compiles and executes correctly on a Tux machine (100% = 10% + 10% + 80%)
    - 5% for the input file `myInputGraph.txt`
    - 5% for the output file `outputShortestPaths.txt`
    - 30% for the program (working correctly with graph.txt). Without the screenshot, no credit will be awarded.
    - 30% for the program (working correctly with your sample `myInputGraph.txt`) Without the screenshot, no credit will be awarded.
    - 30% for the program (working correctly with a grading sample input file following the provided format). The grading sample input file will be prepared by the instructor to be used for grading.

Report

- This is this file with your inserted answers/screenshots.

What you need to turn in:

- Electronic copy of your source program (standalone/separately attached to assignment)
- Electronic copy of this report(standalone/separately attached to assignment). Submit this file as a Microsoft Word or PDF.
- A sample graph text file (with .txt extension) having at least 8 vertices and 16 edges. Call this file `myInputGraph.txt`,
- The output text file produced by your program. Call this file `outputShortestPaths.txt` ,

Grading

- See Points Distribution above