Esta actividad me emocionó mucho por el hecho de que íbamos a hacer uso de diferentes técnicas y algoritmos que hemos aprendido a lo largo del curso. Al principio batalle mucho porque no estaba encontrando solucionar un error, y luego terminó siendo que no había incluido una librería.

En esta ocasión se nos pide que creemos un grafo de tipo lista de adyacencia que tendrá una lista de direcciones IP y sus diferentes conexiones. Para esto primero se nos proporciona con 2 números:

```
1    13370 91910
```

Estos números se nos proporcionaron con la idea de facilitarnos el proceso de creación de vértices. Sin embargo, me tomé la molestia de realizar el programa sin esta ayuda. Lo primero que hice fue leer dicha línea y dividirla por espacios para así poder obtener el valor numero de líneas de vértices:

```
// Creamos variables
ifstream bitacora("bitacoraACT4_3.txt");
string ip, line, temp;
char delim = ' ';
int cont = 1;

// Obtenemos primera linea
getline(bitacora, line);
stringstream ss(line);

// Obtenemos el valor de # de Ip's
getline(ss, temp, delim);
int cantIp = stoi(temp);
```

De esta manera, ya se cuantas líneas me tengo que saltar para llegar a los accesos:

```
// Recorremos todos los lugares hasta llegar a las direcciones
for(int i = 0; i < cantIp; i++) {
    getline(bitacora, line);
}
```

Después de esto, podemos comenzar a crear nuestro grafo. La manera en la que nosotros tenemos definido el constructor de nuestro grafo es tomando por parámetro un vector de vectores de tipo ‘T’. Por lo que, si yo quería hacer uso de mi constructor, tenía que agregar

todos los valores a un vector y luego crear mi grafo. Si, esto hará que se tarde más mi código, pero tenía ganas de hacerlo sin la ayuda. Entonces, tenemos que observar cómo se nos proporcionan los accesos. A continuación pondré un ejemplo:

```
Sep 23 12:58:18 224.182.134.50:5073 80.169.79.65:1150 Failed password for illegal user root
```

Podemos ver entonces que si dividimos entonces por espacios el acceso, tendríamos la siguiente lista:

1. Mes
2. Dia
3. Hora/Minutos/Segundos
4. Ip de Origen/Puerto de Origen
5. Ip de Destino/Puerto de Destino
6. Razon

Si nosotros dividimos entonces este string por espacios, querremos leer solamente la línea 4 y 5. Sabemos también que los puertos siempre son de 4 números, por lo que para obtener nuestra dirección IP habrá que hacer un “substring” que ocupe desde el principio del string hasta su largo menos 5 (5 porque también tomamos en cuenta los dos puntos). Al saber todo esto, podemos escribir el siguiente ciclo:

```
// Ahora obtenemos los vertices y conexiones y los ponemos en una lista
while(getline(bitacora, line)) {
    stringstream ss(line);
    // Checamos todos los elementos de la linea
    while(getline(ss, temp, delim)) {
        // Solo queremos checar los ip, por lo que solo guardaremos valores cuando nuestro contador valga 4 y 5
        if(cont == 4) {
            ip = temp.substr(0, temp.length() - 5);
            tempList.push_back(ip);
        }
        if(cont == 5) {
            ip = temp.substr(0, temp.length() - 5);
            tempList.push_back(ip);
        }
        cont++;
    }
    // Introducimos vector a vector de vectores
    list.push_back(tempList);
    // Reseteamos vector y contador
    tempList.clear();
    cont = 1;
}
```

Este ciclo entonces nos creará un vector “list” que será un vector de vectores de tipo “string” que tendrá todas las direcciones IP de origen con su destino subsecuente.

El constructor funciona de la siguiente manera. Primero creará un vector de vértices en el cual introducirá todas las direcciones de IP únicas. Para terminar este primer paso, el codigo ordenará todas las IP dentro del vector de vertices:

```
// Creamos variables
int source = 0;
int target = 1;
int weight = 2;
vector<Edge<T>> vec;

// Creamos lista de vertices
for(vector<T> edge : list) {
    T temp = edge[source];
    int pos = findVertex(temp);
    if(pos < 0) {
        vertices.push_back(temp);
    }
    temp = edge[target];
    pos = findVertex(temp);
    if(pos < 0) {
        vertices.push_back(temp);
    }
}

sort(vertices.begin(), vertices.end());
```

Después, ya que el código conoce todas las direcciones, volver a recorrer la lista para obtener todas las direcciones de destino (Si, estamos recorriendo la lista de direcciones 2 veces, pero como mencione anteriormente, quería hacer yo el trabajo completo).

```
// Crear lista de adyacencias del size de la cantidad de vertices
vector<Edge<T>> temp;
vector<vector<Edge<T>>> tempList(vertices.size(), temp);
adjacents = tempList;

// Agregar los destinos a la lista de adyacencias
for(auto path : list) {
    int pos = findVertex(path[source]);
    Edge<T> edge;
    edge.vertex = path[target];
    edge.peso = 0;
    adjacents[pos].push_back(edge);
}
```

Para terminar, se nos pide que ordenemos nosotros nuestros datos por medio de un heap. Para este último paso, haré uso de 2 vectores nuevos. Uno que utilizaré para guardar todas las direcciones con sus adyacencias, y otro que utilizaré para que pueda mantener el vector de vértices ordenado de la misma manera. Esto se debe a que el vector de adyacencias utiliza posiciones con relación al vector de vértices, por lo que si simplemente ordenamos las adyacencias, no podríamos saber cual es cual. Por tanto, a la hora de hacer el heap, también estoy introduciendo valores a un vector de vértices nuevo que recorre igual que el heap.

```
// O(n^2)
// Algoritmo con el que haremos nuestro max Heap
template<class T>
void GrafosL<T> :: maxHeap() {

    // Despejamos vector de heap y vertices heap
    heap.clear();
    verticesHeap.clear();

    // Declaramos variable
    int pos, n;

    // Introducimos cada elemento a la lista
    for(int i = 0; i < adjacents.size(); i++) {
        // Metemos el valor de 'i' en nuestro heap
        heap.push_back(adjacents[i]);
        // Metemos el valor de vertices en 'i'
        verticesHeap.push_back(vertices[i]);
        // Posicion inicial sera el ultimo elmento del heap
        pos = heap.size() - 1;
        // Empazamos ciclo checando si el padre tiene menos accesos
        while(pos > 0 && heap[pos/2].size() < heap[pos].size()) {
            // En caso de tener menos accesos, hacemos un swap
            swap(heap[pos/2], heap[pos]);
            // Movemos valores del vertice
            swap(verticesHeap[pos/2], verticesHeap[pos]);
            // Recorremos
            pos = pos / 2;
        }
    }

}
```