EDA - Práctica Calificada
Reynaldo Cusi Ascencio

# Problem 1 - 5 points

Write a function that takes in the head of a Singly Linked List that contains a loop (in other words, the list's tail node points to some node in the list instead of None / null . The function should return the node (the actual node--not just its value) from which the loop originates in constant space.

Each LinkedList node has an integer value as well as a next node pointing to the next node in the list.

**Sample Input**

```
head = 0 -> 1 -> 2 -> 3 -> 4 -> 5 -> 6 // the head node with value 0
                           ^         v
                           9 <- 8 <- 7
```

**Sample Output**

```
// the node with value 4
v
4 -> 5 -> 6
^         v
9 <- 8 <- 7
```

```cpp
#include <iostream>

using namespace std;

// Clase nodo para la lista enlazada
struct NodoLista
{
  int value;
  NodoLista *next = NULL;
  NodoLista(int value)
  {
    this->value = value;
  }
};

/**
 *  Función para imprimir una lista enlazada, entra en un
 *  bucle infinito si la lista contiene ciclos
 * */
void print_list(NodoLista *head)
{
  NodoLista *it = head;
  while (it)
  {
```

```
    cout << it->value << endl;
    it = it->next;
  }
}

/**
 *  Función que busca un ciclo en una lista enlazada,
 *  si contiene un ciclo devuelve un puntero al nodo en el que
 *  el ciclo empieza, caso contrario retorna nulo
 * */
NodoLista* find_cycle(NodoLista *head)
{
  /***
   *  Para ejecutar este algoritmo en espacio de memoria constante,
   *  no podemos utilizar una estructura adicional para ir guardando
   *  los nodos que ya visitamos, por lo que recurriremos al
   *  algoritmo de Floyd para búsqueda de ciclos
   *
   *  El algoritmo de Floyd consta de 2 punteros que recorren la lista enlazada,
   *  un puntero avanzará más rápido que el otro, si lo vemos como una analogía
   *  el puntero más rápido será el conejo y el puntero más lento será una tortuga
   *
   *  La idea de que un puntero avance más rápido que el otro, implica que si existe
      algún ciclo en la lista enlazada, estos punteros en algún momento tendrán que
encontrarse
   * */
  NodoLista *turtle = head;
  NodoLista *rabbit = head->next;

  /***
   *  Entonces tenemos 2 condiciones de parada, si turtle == rabbit, osea que ambos
   *  punteros se encontraron, el otro caso sería que el conejo no tenga
   *  un nodo siguiente, lo que indicaría que la lista enlazada no contiene ciclos
   * */
  while (turtle != rabbit && rabbit->next)
  {
    /**
     *  Aca se representa la idea de un puntero más rápido que el otro,
     *  pues la tortuga solo avanza un nodo, mientras que el conejo
     *  avanza 2 nodos
     * */
    turtle = turtle->next;
    rabbit = rabbit->next->next;
  }

  /**
   *  Si al terminar de avanzar con los punteros, estos son iguales,
   *  entonces para hallar donde empezó este ciclo tendremos que mover
   *  a la tortuga a la posición inicial de la lista, osea al head, y
   *  el conejo lo haremos avanzar 1 paso, ya que antes lo inicializamos
```

```
   *  con 1 paso mas adelante que la tortuga
   * */
  if (turtle == rabbit)
  {
    turtle = head;
    rabbit = rabbit->next;
    /**
     *  Después de mover a la tortuga al inicio, haremos que el conejo
     *  y la tortuga avancen 1 nodo a la vez, finalmente el nodo en el que
     *  se encuentren indicará el inicio del ciclo, esto tiene una demostración
     *  matemática de por qué funciona
     * */
    while (turtle != rabbit)
    {
      turtle = turtle->next;
      rabbit = rabbit->next;
    }
    // Finalmente se retorna el nodo en el que se encontraron
    return turtle;
  }
  else
    return NULL;
}

int main()
{
  // Creamos nuestra lista enlazada
  NodoLista NodoLista0 = NodoLista(0);
  NodoLista NodoLista1 = NodoLista(1);
  NodoLista NodoLista2 = NodoLista(2);
  NodoLista NodoLista3 = NodoLista(3);
  NodoLista NodoLista4 = NodoLista(4);
  NodoLista NodoLista5 = NodoLista(5);
  NodoLista NodoLista6 = NodoLista(6);
  NodoLista NodoLista7 = NodoLista(7);
  NodoLista NodoLista8 = NodoLista(8);
  NodoLista NodoLista9 = NodoLista(9);

  NodoLista0.next = &NodoLista1;
  NodoLista1.next = &NodoLista2;
  NodoLista2.next = &NodoLista3;
  NodoLista3.next = &NodoLista4;
  NodoLista4.next = &NodoLista5;
  NodoLista5.next = &NodoLista6;
  NodoLista6.next = &NodoLista7;
  NodoLista7.next = &NodoLista8;
  NodoLista8.next = &NodoLista9;
  NodoLista9.next = &NodoLista4;

  //print_list(&NodoLista0);  //Recorrido de lista sin control, genera bucle
```

```
  cout << "Existe bucle ?" << endl;
  if(find_cycle(&NodoLista0)){
    cout<<"Si existe: "<<endl;
    cout<<find_cycle(&NodoLista0)->value<<endl;
  }else{
    cout<<"no cycle found"<<endl;
  }
  return 0;
}
```

**Análisis:**

El algoritmo utilizado en este ejercicio consta de una complejidad de ejecución O(n). Para que el puntero más rápido se encuentre con el más lento tiene que recorrer el ciclo un número k constante de veces, por lo que
las iteraciones serían k*c, donde c es la longitud del ciclo. En consecuencia el número total de iteraciones es x + k*c, donde x es la longitud desde el inicio hasta el comienzo de ciclo.

## Problem 3 - 5 points

You're given a non-empty array of positive integers where each integer represents the maximum number of steps you can take forward in the array. For example, if the element at index `1` is `3`, you can go from index `1` to index `2`, `3`, or `4`.

Write a function that returns the minimum number of jumps needed to reach the final index. Note that jumping from index `i` to index `i + x` always constitutes one jump, no matter how large `x` is.

**Sample Input**

```
array = [3, 4, 2, 1, 2, 3, 7, 1, 1, 1, 3]
```

**Sample Output**

```
4 // 3 --> (4 or 2) --> (2 or 3) --> 7 -->3
```

```cpp
#include <iostream>

using namespace std;

// Definimos una constante que representa el infinito
#define INF 100000

/**
 *  Para resolver este problema haremos uso de programación dinámica,
 *  por lo que ocuparemos una estructura adicional para guardar las soluciones,
 *  en este caso basta con un array que indique el número mínimo de saltos necesarios
 *  para llegar desde la posición del índice hasta el final,
 *  por ejemplo, minimo_saltos[n-1] siempre será 0, pues ya se encuentra en el final
 * */
const int n = 11;
int minimo_saltos[n] = {0};

/**
 *  En esta funcion devolveremos el número mínimo de saltos necesarios para
 *  llegar al final desde la posición inicial
 * */
int obtiene_minimo_saltos(int *array)
{
  /**
   *  Entonces, cómo es programación dinámica tendremos que llenar nuestro arreglo
   *  de soluciones(minimo_saltos), lo llenaremos de derecha a izquierda porque
   *  ya sabemos cuántos saltos se necesitan para llegar desde el final
   *  hasta el final, osea 0
   * */
  for (int i = n - 2; i >= 0; i--)
  {
    int minimum = INF;
    /**
```

```
     *  En cada iteración del ciclo hallaremos el valor de minimo_saltos[i],
     *  para lo cual verificaremos el número mínimo de saltos de las posiciones
     *  a las que podemos llegar desde nuestra posición actual, estamos en la posición i,
     *  por lo que las posiciones a las que podemos llegar son array[i],
     *  la condición (i+j+1<n) solo sirve para evitar que nos salgamos de los límites
     *  del arreglo
     * */
    for (int j = 0; j < array[i] && (i + j + 1 < n); j++)
    {
      /**
       *  si el número mínimo de saltos de la posición i+j+1 es menor al mínimo de
saltos,
       *  actualizamos el valor mínimo
       * */
      if (minimo_saltos[i + j + 1] < minimum)
      {
        minimum = minimo_saltos[i + j + 1];
      }
    }

    /**
     *  una vez encontramos el número mínimo de saltos para llegar al final desde
     *  la posición y, simplemente le asignamos el valor de 1 + minimum,
     *  se le suma 1 porque tenemos que contar el salto que da para llegar a la posición
     *  donde se encuentra el mínimo
     * */
    minimo_saltos[i] = 1 + minimum;
  }

  /**
   *  Finalmente, como ya tenemos el número mínimo de saltos para llegar al final
   *  desde cualquier posición, simplemente retornamos el número mínimo de saltos
   *  del primer elemento
   * */
  return minimo_saltos[0];
}

int main()
{
  // Creamos el array de entrada
  int array[n] = {3, 4, 2, 1, 2, 3, 7, 1, 1, 1, 3};
  cout << obtiene_minimo_saltos(array) << endl;
  return 0;
}
```

**Análisis:**

Este ejercicio fue resuelto utilizando programación dinámica, si intentamos resolver con fuerza bruta la complejidad será de O(n^n), ya que se tendría que inspeccionar para cada elemento del

array todos sus posibles caminos hasta el final. En el caso de la implementación de programación dinámica, la complejidad de ejecución sería de O(n^2), pues en un arreglo adicional vamos guardando el número
de saltos mínimos para llegar hasta el final, entonces para llenar el arreglo que contiene los saltos mínimos tenemos que recorrer el arreglo una vez, y para cada elemento del arreglo tenemos que buscar el número mínimo de saltos de las posiciones a las que podemos llegar, por lo que en el peor de los casos sería O(n^2)

# Problem 4 - 7 points

---

Write a function that takes in a non-empty list of non-empty sorted arrays of integers and returns a merged list of all of those arrays.
The integers in the merged list should be in sorted order.

## Sample Input

```
arrays = [
  [1, 5, 9, 21],
  [-1, 0],
  [-124, 81, 121],
  [3, 6, 12, 20, 150],
]
```

## Sample Output

```
[-124, -1, 0, 1, 3, 5, 6, 9, 12, 20, 21, 81, 121,150]
```

## Hints

- If you were given just two sorted lists of numbers in real life, what steps would you take to merge them into a single sorted list? Apply the same process to k sorted lists. The first element
- in each array is the smallest element in the respective array; to find the first element to add to the final sorted list, pick the smallest integer out of all of the smallest elements. Once you've found the smallest integer, move one position forward in the array that it came from and continue applying this logic until you run out of elements.
  The approach described in Hint #2 involves repeatedly finding the smallest of k elements, since
- there are k arrays. Doing so can be naively implemented using a simple loop through the k relevant elements, which results in an O(k)-time operation. Can you speed up this operation by using a specific data structure that lends itself to quickly finding the minimum value in a set of values.
- Follow the approach described in Hint #2, using a Min Heap to store the k smallest elements at any given point in your algorithm.

```cpp
#include <iostream>
#include <vector>
#include <stack>

using namespace std;

/**
 *  Esta función permite unir 2 arreglos que ya están ordenados en 1 arreglo
```

```
 *  ordenado de manera lineal
 * */
vector<int> merge(vector<int> a, vector<int> b)
{
  /**
   *  Utilizamos 2 índices para recorrer ambos arreglos
   *  al mismo tiempo
   * */
  int i, j;
  i = j = 0;

  // en este vector result guardaremos el arreglo ordenado
  vector<int> result;

  /**
   *  Avanzaremos los índices i y j por sus respectivos arreglos,
   *  y hallaremos cual es menor, si a[i] es menor que b[j], entonces
   *  agregamos a[i] a result e incrementamos el valor de i, la misma
   *  idea se cumpliria para b[j]
   * */
  while (i < a.size() && j < b.size())
  {
    if (a[i] < b[j])
      result.push_back(a[i++]);
    else
      result.push_back(b[j++]);
  }

  /**
   *  Uno 2 iteradores terminará antes
   *  su arreglo antes que el otro, por lo que simplemente tendremos
   *  que agregar los elementos que faltan del iterador que aún no
   *  termino de recorrer su arreglo
   * */
  while (i < a.size())
    result.push_back(a[i++]);

  while (j < b.size())
    result.push_back(b[j++]);

  return result;
}

/**
 *  sort_arrays sera nuestra funcion recursiva que actuara como
 *  la función recursiva del merge sort, la idea es dividir los
 *  k arreglos en 2 mitades k/2 arreglos, estas mitades serán left y right,
 *  una vez tenemos las mitades llamamos a la función de manera recursiva,
 *  pero le pasamos los arreglos de la izquierda y la derecha a la función merge,
 *  que es la que se encarga de juntarlos
```

```
 * */
vector<int> sort_arrays(vector<vector<int>> arrays)
{
  /**
   *  las funciones recursivas siempre necesitan condiciones de parada,
   *  caso contrario se ejecutarán infinitamente, en este caso
   *  la condición de parada está dada cuando el arreglo de arreglos
   *  solo contiene 1 arreglo, lo que indicaría que ya está ordenado,
   *  la otra condición de parada es cuando contiene 2 arreglos, en este caso
   *  tendremos que unir estos 2 arreglos con la función merge()
   * */
  if (arrays.size() == 1)
    return arrays[0];

  if (arrays.size() == 2)
    return merge(arrays[0], arrays[1]);

  /**
   *  en esta parte se crean los arreglos de arreglos para la mitad izquierda
   *  y la mitad derecha
   * */
  vector<vector<int>> left;
  vector<vector<int>> right;

  // Agregamos los arreglos a su respectiva mitad a la que pertenecen
  for (int i = 0; i < arrays.size(); i++)
    if (i < arrays.size() / 2)
      left.push_back(arrays[i]);
    else
      right.push_back(arrays[i]);

  return merge(sort_arrays(left), sort_arrays(right));
}

int main()
{
  // creamos el arreglo de arreglos
  vector<vector<int>> arrays;

  arrays.push_back({1, 5, 9, 21});
  arrays.push_back({-1, 0});
  arrays.push_back({-124, 81, 121});
  arrays.push_back({-3, 6, 2, 20, 150});

  vector<int> sorted = sort_arrays(arrays);
  for (int i = 0; i < sorted.size(); i++)
    cout<<sorted[i]<<" ";
  cout<<endl;
}
```

**Análisis:**

En este ejercicio se utilizó una implementación basada en el merge sort, simplemente se acomodó el algoritmo para que acepte un arreglo de arreglos de números ordenados, por lo tanto, para unir 2 arreglos se tiene una complejidad de O(n), y vamos uniendo por mitades, entonces la altura del árbol sería de lg(n), por lo que, en total la complejidad de ejecución sería de O(n.lg(n))