# Proyecto 1

| | |
|---|---|
| Fecha | @April 4, 2022 |
| Fuente | |
| Material | |
| Tags | |

Por **Rodríguez Pérez Axel**

Licenciatura en Ciencias Genómicas, Escuela Nacional de Estudios Superiores Unidad Juriquilla

# Complejidad de algoritmos

## TAREA 1. Ordenando un vector

Diseña dos algoritmos e impleméntalos (en python) que ordenen un vector de tamaño $n$ de menor a mayor. El algoritmo debe incluir el llenado del vector ya sea vía teclado o leyendo los datos de un archivo y debe incluir la impresión de resultado, ya sea a pantalla o como parte de un archivo.

### Código de primer algoritmo

```
# Primer algoritmo que ordena un vector de menor a mayor

# INPUT DE LOS DATOS DEL VECTOR
vector = []  # Creacion de un vector vacio
tam_vect = int(input(f"Inserte la longitud del vector: "))  # Tamaño del vector

# Insercion de los datos del vector
for i in range(tam_vect):
    elemento = int(input(f"Inserte el elemento {i} del vector: \n"))  # i-esimo elemento del vector
    vector.append(elemento)  # Se agrega al vector

# ORDENAMIENTO DEL VECTOR
for i in range(1, tam_vect):
    actual = vector[i]  # tomamos el i-esimo elemento del vector
    j = i  # j es una var. temporal

    # Desplazamiento de los elementos
    while j > 0 and vector[j-1] > actual:  # mientras el elemento anterior a 'actual' (i-esimo elemento)
        # sea mayor que este
        vector[j] = vector[j-1]  # recorremos hacia adelante ese elemento (el anterior a 'actual')
        j = j-1  # se le resta 1 a 'j' debido a que ahora este será el índice del recorrido hacia atras
        # elemento 'actual'

    # Insertar el elemento en su lugar
    vector[j] = actual  # El desplazado elemento 'actual' toma su nuevo lugar en el j-esimo indice del vector
print(vector)
```

### Complejidad de primer algoritmo

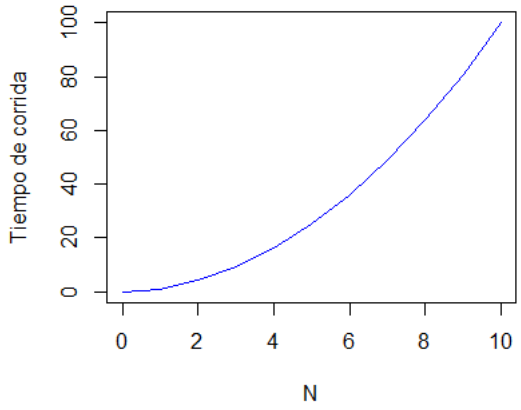
En este caso, nos referimos como $n$ a `tam_vect`. Se puede dividir el código en dos bloques:

1. El primero, con un ciclo for dependiente de `tam_vector`.
2. El segundo, dos ciclos anidados, el primero dependiente de `tam_vect` y el segundo de `j`.

Tomamos el bloque dos, ya que por contener dos ciclos anidados, es el más complejo.

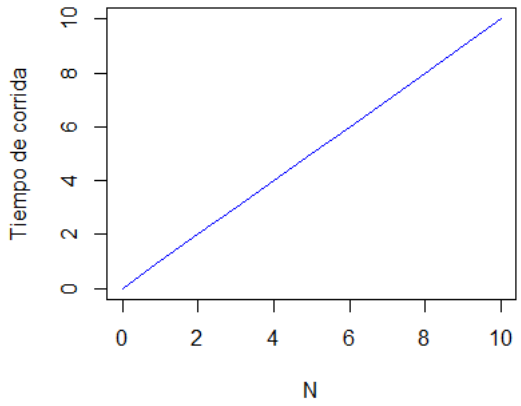
**$O$ (O grande):**

- En el caso desfavorable, es decir, con los datos ordenados a la inversa, se necesita realizar $(n-1)+(n-2)+(n-3)+...+1$ comparaciones e intercambios, o lo que es igual a $\frac{n^2-n}{2}$.
- Por lo tanto, $O(n^2)$, es decir, tiene una O grande cuadrática.




**$\Omega$ (Omega grande):**

- En el caso más favorable, es decir, si recibiéramos el vector con los datos ya ordenados, el algoritmos solo realiza $n$ comparaciones (`tam_vect` comparaciones).
- Por tanto, $\Omega(n)$, es decir, una omega grande lineal.




## Código de segundo algoritmo

```
# Algoritmo que obtiene el camino más corto entre dos nodos en una red no dirigida

import numpy as np  # Se importa la libreria numpy
import pandas as pd  # Se importa la libreria pandas
from queue import Queue  # Se importa la funcion Queue del modulo queue

# INPUT DE TABLA DE ADYACENCIA
nodos = int(input(f"Inserte el numero de nodos de la red: "))  # Numero de nodos de la red
ad_table = np.random.random((nodos, nodos))  # tabla de adyacencia
ad_table = pd.DataFrame(ad_table)
for i in range(nodos):
    for j in range(nodos):  # Para cada columna i de cada fila j
        if ad_table[j][i] != ad_table[i][j]:  # Si los valores de la tabla de adyacencia de la fila
            # j--columna i son diferentes a las de la fila i--columna j
            conex = int(input(f"El nodo {i} tiene una conexion con el nodo {j}? "))  # Input de si
            # existe conecion entre ambos nodos
```

```
            # Al ser una matriz no dirigida, las celdas j-i e i-j deben tener el mismo valor
            ad_table[j][i] = conex
            ad_table[i][j] = conex
        else:
            if j == i:  # Checamos si el nodo j tiene una concexion consigo msimo
                conex = int(input(f"El nodo {i} tiene una conexion con el nodo {j}? "))
                ad_table[j][i] = conex
                ad_table[i][j] = conex

# Creacion de la lista de adyacencia
adj_list2 = {}  # 'ad_list2' sera un diccionario en el que se almacenara la
# informacion de los vecinos de cada uno de los nodos del grafo
for nodo in range(nodos):
    vecinos2 = []  # 'vecinos2' almacena una lista con todos los vecinos del nodo 'nodo'
    for i in range(nodos):
        if ad_table[nodo][i] == 1:  # Si en la tabla de adyacencia se encuentra un 1, quiere
            # decir que el nodo 'nodo' y el nodo 'i' son vecinos
            vecinos2.append(i)  # Por eso se agrega al nodo 'i' a la lista de vecinos
    adj_list2[nodo] = vecinos2  # 'nodo' es la llave para los valores de la lista 'vecinos2',
    # que contienen los vecinos de ese mismo nodo

# PROCESO

# INICIALIZACION DE TODOS LOS ARREGLOS REQUERIDOS

visitados = {}  # Para tener registro de todos los nodos visitados
distancias = {}  # Diccionario de distancias
vecino = {}  # Diccionario de los vecinos de cada nodo
camino_general = []  # El camino que se recorre del nodo de inicio hasta el ultimo vertice

cola = Queue()  # Inicializamos una cola vacia

for nod in adj_list2.keys():  # Para cada uno de los nodos
    visitados[nod] = 0  # Asumimos que no lo hemos visitado (no hemos pasado por el),
    vecino[nod] = None  # Que no tiene vecinos
    distancias[nod] = -1  # Y que su distancia es de -1. Esto solo para tener un orden

nodo_inicio = int(input(f"Ingrese el nodo del que se parte: "))
visitados[nodo_inicio] = 1  # Puesto que es el nodo inicial, se dice que ya lo recorrimos
distancias[nodo_inicio] = 0  # La distancia de un nodo a si mismo es 0
cola.put(nodo_inicio)  # Agregamos el nodo de inicio a la cola

while not cola.empty():  # Mientras la cola no este vacia
    actual = cola.get()  # Removemos el elemento mas a la izquierda de la cola
    camino_general.append(actual)

    for v in adj_list2[actual]:  # Para cada uno de los vecinos de actual
        if not visitados[v]:  # Checamos si el vertice 'v' (que es vecino de 'actual') ha sido
            # visitado anteriormente o no (si ya lo recorrimos o no)
            visitados[v] = 1  # Si no ha sido visitado, lo marcamos ahora como visitado
            vecino[v] = actual
            distancias[v] = distancias[actual] + 1  # Esto quiere decir que la distancia de 'v'
            # al nodo de inicio sera la distancia del nodo 'actual' al mismo vertice mas uno (ya que
            # 'v' es vecino de 'actual' y por ende esta un nivel mas alejado de 'nodo_inicio')
            cola.put(v)  # Una vez visitado 'v', lo agregamos a la cola


# El camino mas corto del nodo de inicio hacia el nodo objetivo
nodo_ob = int(input(f"Ingrese al que se quiere llegar: "))  # Nodo objetivo
camino = []  # Vector que almacenara la ruta para llegar de nodo_i a nodo_ob
while nodo_ob is not None:
    camino.append(nodo_ob)
    nodo_ob = vecino[nodo_ob]
    # Esto quiere decir que iremos de reversa. Es decir, comenzamos desde el nodo al que queremos
    # llegar, checamos sus vecinos y agregamos

camino.reverse()  # Como el vector estaba al reves, lo volteamos para que quede en orden.

print(f"El camino mas corto es {camino}")
```

## Complejidad del segundo algoritmo

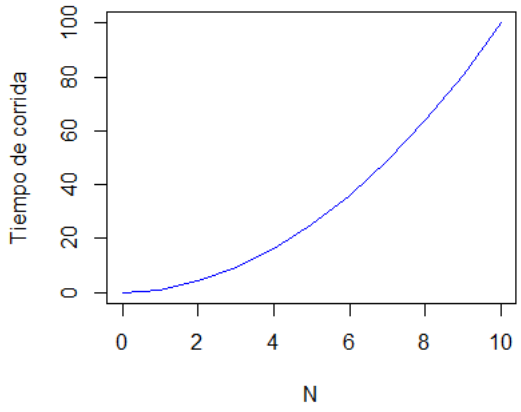
En este código, de igual forma nos tomamos a `tam_vect` como $n$. Se puede dividir el código en dos bloques:

1. El primero, el llenado del vector que se va a ordenar, con un ciclo for dependiente de `tam_vect`.
2. El segundo, el ordenamiento del vector, que cuenta con dos ciclos anidados, y un condicional dentro del ciclo.

Tomamos el el bloque dos, ya que por los dos ciclos anidados, es el más complejo.

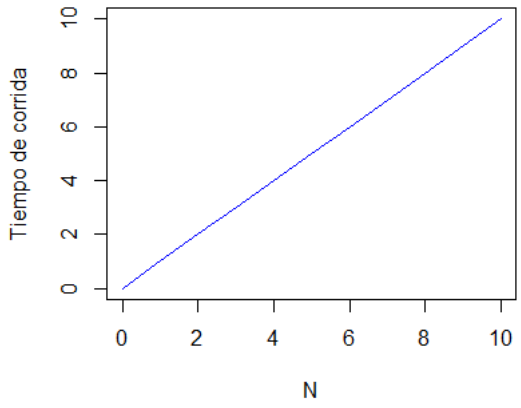
**$O$ (O grande):**

- En el caso menos óptimo, es decir, que el vector esté acomodado de mayor a menor, se necesita realizar $n$ veces $n$ comparaciones e intercambios, es decir $n^2$.
- Por tanto, $O(n^2)$, es decir, tiene una O grande cuadrática.




**$\Omega$ (Omega grande):**

- En el caso más favorable, es decir, si recibiéramos el vector con los datos ya ordenados, el algoritmos solo realiza $n$ comparaciones, ya que nunca entró a las sentencias del condicional, por lo que la bandera `orden` nunca cambió su valor y no fue necesario entrar en el ciclo `while` de nuevo.
- Por tanto, $\Omega(n)$, es decir, una omega grande lineal.




## Discusión

Si bien es verdad que el valor de Theta grande y O grande de ambos algoritmos es el mismo, el tiempo de corrida del primer algoritmo sería considerablemente más rápido que el del segundo, ya que mientras que este tiene una $O(n) = \frac{n^2-n}{2}$, el segundo tiene una O grande de solo $n^2$.

Es decir que, si por ejemplo, tuviéramos un $n$ de tamaño 2,000, el primer algoritmo, en el caso menos favorable, tardaría $\frac{2{,}000^2-2{,}000}{2} = 1,999,000$ milisegundos en correr, mientras que el caso del segundo sería de $2{,}000^2$ = 4,000,000 milisegundos.

## TAREA 2. Obteniendo el camino más corto

Diseña dos algoritmos e impleméntalos (en python) que obtengan el camino más corto entre dos nodos en una red no dirigida. El algoritmo debe incluir el llenado de la matriz de adyacencia correspondiente y la impresión del resultado a pantalla.

### Código de primer algoritmo

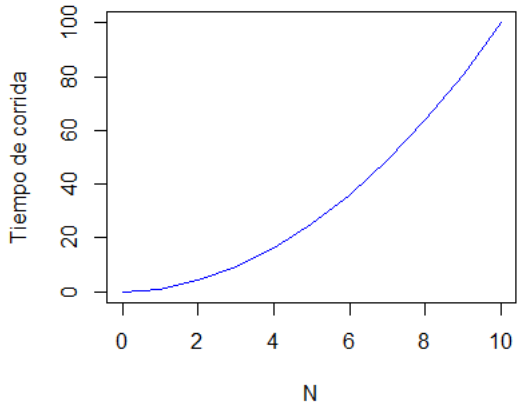
```
# Segundo algoritmo que ordena un vector de menor a mayor

# INPUT DE LOS DATOS DEL VECTOR
vector = []  # Creacion de un vector vacio
tam_vect = int(input(f"Inserte la longitud del vector: "))  # Tamaño del vector

# Insercion de los datos del vector
for i in range(tam_vect):
    elemento = int(input(f"Inserte el elemento {i} del vector: \n"))  # i-esimo elemento del vector
    vector.append(elemento)  # Se agrega al vector

# ORDENAMIENTO DEL VECTOR
orden = 1  # Bandera encendida (igual a 1) cuando siga siendo necesario un ordenamiento
tope = 0  # numero de elementos desordenados - 1. Funciona como tope
while orden == 1:  # Mientras el vector siga desordenado
    orden = 0  # Asumimos que esta ordenado
    tope = tope + 1
    for i in range(0, tam_vect - tope):  # Para cada i-esimo elemento en los indices de 0 al tope del vector - tope
        if vector[i] > vector[i + 1]:  # Si el i-esimo elemento del vector es mayor a su contiguo
            orden = 1  # El vector esta desordenado

            # Intercambiamos los dos elementos
            mayor = vector[i]  # Almacenamos el elemento de mayor valor
            menor = vector[i+1]  # Almacenamos el elemento de mayor valor
            vector[i+1] = mayor  # Se recorre el elemento mayor una posicion
            vector[i] = menor  # El elemento menor retrocede una posicion

print(vector)
```

### Complejidad de primer algoritmo

Se puede dividir el código en 5 bloques. El bloque de mayor importancia en cuanto a complejidad es aquel en el que se realiza el input de la tabla de adyacencia, ya que estos son dos ciclos for anidados a los que forzosamente se debe entrar. En este caso, $n$ representa el numero de nodos, es decir `nodos`.

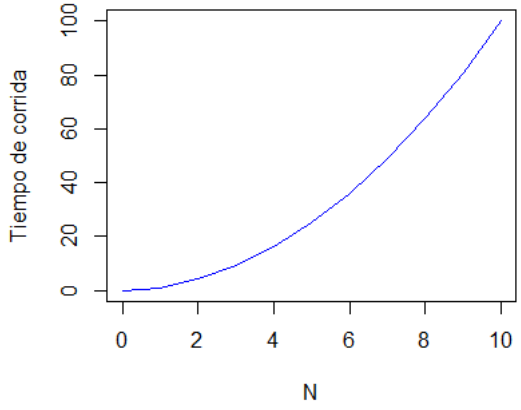
**$O$ (O grande):**

- En el caso menos óptimo, se tienen que ingresar $n + (n-1) + (n-2) + (n-3) + ... + 1$ datos a la matriz, ya que esas son las veces que estamos pidiendo información acerca de los vecinos del nodo $k$. Esto debido a que, al ser una matriz no dirigida, no se necesita preguntar dos veces las conexiones de $k$.
- Entonces, se tiene que hacemos $\frac{n^2-n}{2}$ acciones, lo que significa que $O$ grande es $n^2$.




**$\Omega$ (Omega grande):**

- En el caso más óptimo, se cuenta con una $\Omega$ grande de $n^2$, de igual forma, ya que uno de los inputs del algoritmo es justamente el llenado de la tabla de adyacencia.




### Código de segundo algoritmo

```
# Algoritmo que obtiene el camino más corto entre dos nodos en una red no dirigida

import numpy as np  # Se importa la libreria numpy
import pandas as pd  # Se importa la libreria pandas
from queue import Queue  # Se importa la funcion Queue del modulo queue

# INPUT DE TABLA DE ADYACENCIA
nodos = int(input(f"Inserte el numero de nodos de la red: "))  # Numero de nodos de la red
ad_table = np.random.random((nodos, nodos))  # tabla de adyacencia
ad_table = pd.DataFrame(ad_table)
for i in range(nodos):
    for j in range(nodos):  # Para cada columna i de cada fila j
        if ad_table[j][i] != ad_table[i][j]:  # Si los valores de la tabla de adyacencia de la fila
            # j--columna i son diferentes a las de la fila i--columna j
            conex = int(input(f"El nodo {i} tiene una conexion con el nodo {j}? "))  # Input de si
            # existe conecion entre ambos nodos

            # Al ser una matriz no dirigida, las celdas j-i e i-j deben tener el mismo valor
```

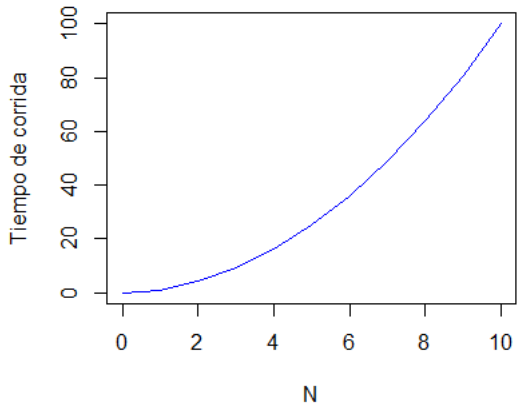
```
            ad_table[j][i] = conex
            ad_table[i][j] = conex
        else:
            if j == i:  # Si estamos checando si el nodo j tiene una concexion consigo msimo
                conex = int(input(f"El nodo {i} tiene una conexion con el nodo {j}? "))
                ad_table[j][i] = conex
                ad_table[i][j] = conex

# Creacion de la lista de adyacencia
adj_list2 = {}  # 'ad_list2' sera un diccionario en el que se almacenara la
# informacion de los vecinos de cada uno de los nodos del grafo
for nodo in range(nodos):
    vecinos2 = []  # 'vecinos2' almacena una lista con todos los vecinos del nodo 'nodo'
    for i in range(nodos):
        if ad_table[nodo][i] == 1:  # Si en la tabla de adyacencia se encuentra un 1, quiere
            # decir que el nodo 'nodo' y el nodo 'i' son vecinos
            vecinos2.append(i)  # Por eso se agrega al nodo 'i' a la lista de vecinos
    adj_list2[nodo] = vecinos2  # 'nodo' es la llave para los valores de la lista 'vecinos2',
    # que contienen los vecinos de ese mismo nodo

# PROCESO

# INICIALIZACION DE TODOS LOS ARREGLOS REQUERIDOS

visitados = {}  # Para tener registro de todos los nodos visitados
distancias = {}  # Diccionario de distancias
vecino = {}  # Diccionario de los vecinos de cada nodo
camino_general = []  # El camino que se recorre del nodo de inicio hasta el ultimo vertice

cola = Queue()  # Inicializamos una cola vacia

for nod in adj_list2.keys():  # Para cada uno de los nodos
    visitados[nod] = 0  # Asumimos que no lo hemos visitado (no hemos pasado por el),
    vecino[nod] = None  # Que no tiene vecinos
    distancias[nod] = -1  # Y que su distancia es de -1. Esto solo para tener un orden

nodo_inicio = int(input(f"Ingrese el nodo del que se parte: "))
visitados[nodo_inicio] = 1  # Puesto que es el nodo inicial, se dice que ya lo recorrimos
distancias[nodo_inicio] = 0  # La distancia de un nodo a si mismo es 0
cola.put(nodo_inicio)  # Agregamos el nodo de inicio a la cola

while not cola.empty():  # Mientras la cola no este vacia
    actual = cola.get()  # Removemos el elemento mas a la izquierda de la cola
    camino_general.append(actual)

    for v in adj_list2[actual]:  # Para cada uno de los vecinos de actual
        if not visitados[v]:  # Checamos si el vertice 'v' (que es vecino de 'actual') ha sido
            # visitado anteriormente o no (si ya lo recorrimos o no)
            visitados[v] = 1  # Si no ha sido visitado, lo marcamos ahora como visitado
            vecino[v] = actual
            distancias[v] = distancias[actual] + 1  # Esto quiere decir que la distancia de 'v'
            # al nodo de inicio sera la distancia del nodo 'actual' al mismo vertice mas uno (ya que
            # 'v' es vecino de 'actual' y por ende esta un nivel mas alejado de 'nodo_inicio')
            cola.put(v)  # Una vez visitado 'v', lo agregamos a la cola


# El camino mas corto del nodo de inicio hacia el nodo objetivo
nodo_ob = int(input(f"Ingrese al que se quiere llegar: "))  # Nodo objetivo
camino = []  # Vector que almacenara la ruta para llegar de nodo_i a nodo_ob
while nodo_ob is not None:
    camino.append(nodo_ob)
    nodo_ob = vecino[nodo_ob]
    # Esto quiere decir que iremos de reversa. Es decir, comenzamos desde el nodo al que queremos
    # llegar, checamos sus vecinos y agregamos

camino.reverse()  # Como el vector estaba al reves, lo volteamos para que quede en orden.

print(f"El camino mas corto es {camino}")
```

### Complejidad de primer algoritmo

Este código es muy parecido al anterior: se puede dividir en 5 bloques. El que tiene un término mayor sería, de nuevo, el bloque correspondiente al del llenado de la tabla de adyacencia. Recordemos que $n$ representa el número de nodos del grafo, es decir, `nodos`.

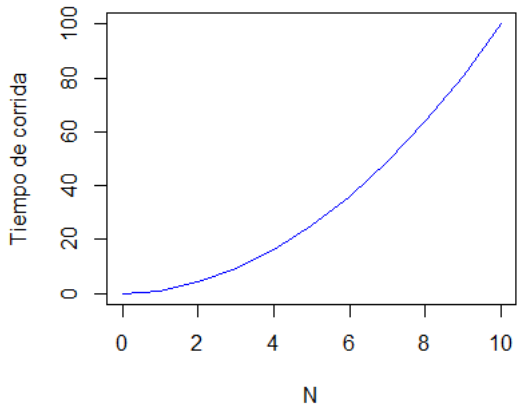
**$O$ (O grande):**

- En el caso menos óptimo, se tiene una $O$ grande de $n^2$, ya que tendremos que, en principio, se tendría que recorrer $n$ veces los $n$ nodos para verificar con cuáles nodos tiene conexión cada respectivo nodo.




**$\Omega$ (Omega grande):**

- En el caso más óptimo, se cuenta con una $\Omega$ grande de $n^2$, de igual forma.




### Discusión

Ambos códigos parecen tener una eficiencia similar, puesto que en ambos códigos el término de más relevante es cuadrado. La única diferencia radica en su sencillez, más que en su complejidad, para lo cual, el código 2 requiere de estructuras menos complejas para su correcta ejecución.

# Punteros

## TAREA 3. Ordenando un vector con memoria dinámica y punteros

```
// Algoritmo que que ordena un vector de menor a mayor
// Rodriguez Perez Axel
```

```
#include<iostream>  /* libreria (archivo de cabecera) iostream contiene codigo que permite a un programa C++ mostrar
la salida en la pantalla y leer la entrada del teclado */
#include<new>
#include <typeinfo>


using namespace std;  // Utilizamos el espacio de trabajo estandar

int main() {
  int i, j, z;  // Variables auxiliares
  int actual;  // Var. temporal que se utiliza para evaluar un elemento
  int tam_vect;  // Var. que almacena el tamanyo del vector
  int *p2;  // Puntero que apunta a elementos de vector
  int *p1;  // Puntero que apunta hacia 'actual'
  int *p3;  // Puntero que apunta hacia z-esimo elemento del vector
  int *vector;  // vector a ordenar


  // INPUT DEL VECTOR
  cout<<"Introduzca el tamanyo del vector: ";
  cin>> tam_vect; // tamanyo del vector

  vector = new (nothrow) int [tam_vect];  // Creacion de vector dinamico

  // Verificar que sea un puntero valido
  if (vector == nullptr){
    cout << "Error: no hay memoria suficiente";
  }else{
    p2 = vector;
      for(z=0; z<tam_vect; z++){
      cout<<"Ingrese el valor del elemento " <<z<<": ";
      p3 = &vector[z];
          cin>>*p3; // Input del z-esimoelemento del vector
      }
    }

  // ORDENAMIENTO DEL VECTOR
  for(i=1; i<tam_vect; i++){ // Para cada elemento del vector
    actual = vector[i]; // el valor a evaluar sera el i-esimo elemento del vector
    p1 = &actual;  // p1 apunta hacia la ubicacion de variable 'actual'
    for(j=i; j > 0 && vector[j-1] > *p1; j--){ // Para el elemento evaluado 'i', si j
    // mayor a cero Y el elemento anterior a 'i'('j') menor a 'actual'
        vector[j] = vector[j-1];  // cambiar j a la posicion de de 'actual'
      }
  vector[j] = *p1; // cambiar a actual a la posicion donde antes estaba el elemento 'j'.
    }

  // IMPRESION DE VECTOR ORDENADO
  for(i=0; i<tam_vect; i++){
    cout<<*p2<<endl;
      p2++;
    }

  //LIBERAR MEMORIA
  delete[] vector;
}
```

## TAREA 4. Obteniendo el camino más corto con memoria dinámica y punteros

```
#include <iostream>
#include<bits/stdc++.h>
#include<time.h> // Libreria para generar numeros al azar
#include <vector>

using namespace std;  // Utilizamos el espacio de trabajo estandar

vector<int>lista_adyacencia;

int main() {
  int nodos, nodo_i, nodo_ob, s;
    int i, j;
    int conex, n_conexiones=0;

    srand(time(NULL));  // Para poder obtener numeros aleatorios

    cout<<"Inserte el numero de nodos de la red: ";
```

```
  cin>>nodos;

  // Input de la tabla de adyacencia
  int ad_table[nodos][nodos]; // ad_table es una variable con 'nodos' columnas con datos de tipo int

  //Declaracion de la lista de adyacencia
  vector<int> ad_list[nodos];

  int *p1;  // Puntero que apunta hacia la direccion en memoria del elemento en la fila i, columna j
  int *p2; // Puntero que apunta hacia el elemento de la i-esima fila y j-esima columna de la matriz de adyacencia
  int *p3; // Puntero que apunta hacia el elemento de la j-esima fila y i-esima columna de la matriz de adyacencia

  // LLENADO DE MATRIZ CON NUMEROS ALEATORIOS
  // Se requiere de una matriz con numero aleatorios para no tener que preguntar dos
// veces por la conexion entre dos nodos que con anticipacion ya sabemos tienen una conexion.
// Esto debido a que es un grafo no dirigido
for(i=0; i<nodos; i++){
    for(j=0; j<nodos; j++){
      p1 = &ad_table[i][j];
      *p1 = 1+rand() %1000; // el valor del elemento de la fila i, columna j tendra un
      // valor aleatorio desde 0 a 1000
    }
  }


// PROCESO
for(i=0; i<nodos; i++){
  for(j=0; j<nodos; j++){
    if(ad_table[i][j] != ad_table[j][i]){  //Si los valores de la tabla de adyacencia de la fila
          // j--columna i son diferentes a las de la fila i--columna j
      cout<<"El nodo " <<i<< " tiene una conexion con el nodo " <<j<< "? ";
      cin>>conex; // Input de si existe conecion entre ambos nodos
      p2 = &ad_table[i][j];
      p3 = &ad_table[j][i];
      // Al ser una matriz no dirigida, las celdas j-i e i-j deben tener el mismo valor
      *p2 = conex;
      *p3 = conex;
      if(conex == 1){
        n_conexiones = n_conexiones+1;
        ad_list[i].push_back(j); // Anyadimos a la lista de adyacencia a 'j' en la posicion 'i'
        ad_list[j].push_back(i); // // Anyadimos a la lista de adyacencia a 'i' en la posicion 'j'
      }
    }else{
      if(j==i){ // Checamos si el nodo j tiene una concexion consigo msimo
        cout<<"El nodo " <<i<< " tiene una conexion con el nodo " <<j<< "? ";
        cin>>conex;
        p2 = &ad_table[i][j];
        p3 = &ad_table[j][i];
        *p2 = conex;
        *p3 = conex;
        if(conex == 1){
          n_conexiones = n_conexiones+1;
          ad_list[i].push_back(j);
          ad_list[j].push_back(i);
        }
      }
    }
  }
}


// INICIALIZACION DE TODOS LOS ARREGLOS REQUERIDOS
 int visitados[nodos]= {0}; // vector de tamanyo nodos con puros valores igual a 0,
 // indicando que no se ha visitado ningun nodo
 int *p4; //Puntero que apunta hacia la direccion en memoria de el indice 'nodo_i'

 int vecinos[nodos]; // Vector 'vecinos' para los vecinos de cada nodo. Se utiliza para encontrar el camino mas corto
 int *p5; // Puntero que apunta hacia la direccion en memoria de vecinos[nodo_i]

cout<<"Inserte el nodo de donde se parte: ";
cin>>nodo_i; // Nodo inicial
int *p6;
p6 = &nodo_i; // p6 es un puntero que apunta hacia el nodo inicial

int *p7;  // Puntero que se utiliza para apuntar hacia el u-esimo
// valor del vector visitados
int *p8;  // Puntero que se utiliza para apuntar hacia el u-esimo
// valor del vector vecinos

p4 = &visitados[nodo_i];
```

```
  *p4 = 1;  // mediante la desreferenciacion de p4 le asigamos 1 a el nodo_i-esimo
  // valor del vector 'visitados'. Esto porque se marca como visitado el nodo fuente
  p5 = &vecinos[nodo_i];
  *p5 = -1;  // mediante la desreferenciacion de p5 le asignamos -1 al nodo_i-esimo valor del vector vecinos,
  // debido a que el nodo base no parte hacia ningun lado, es decir, se toma como si no tuviera parientes.

  queue<int> cola;  // Creamos una cola
  cola.push(*p6);  // Incluimos al nodo inicial al inicio de la cola
  while(!cola.empty()){  // mientras la cola no este vacia
    int elemento = cola.front();  // 'v' va a ser el elemento que tomamos del principio de la cola
    cola.pop();  // lo removemos de la cola
    for(int u: ad_list[elemento]){  // Para cada uno de los vecinos de 'elemento'
      p7 = &visitados[u];
      p8 = &vecinos[u];
      if (!*p7){ // Si no ha sido visitado
        *p7 = 1; // Lo marcamos ahora como visitado
        cola.push(u);  // Ahora incluimos u a la cola
        *p8 = elemento;  // uno de los vecinos del nodo 'u' va a ser 'elemento'
      }
    }
  }

  // ENCONTRAR EL CAMINO MAS CORTO
  cout<<"Inserte el nodo al que se quiere llegar: ";
  cin>>nodo_ob;

  int actual = nodo_ob; // El nodo que empezaremos evaluando es el nodo objetivo

  int *camino;  // puntero que utilizamos como vector
  int *p12;  // p12 apunta hacia la direccion en memoria del f-esimo valor del vector vecinos

  camino = new(nothrow) int[nodos]; // En el peor de los casos, el camino mas corto sera igual al numero de nodos de la red
  int f = 0;  // tamanyo final del vector 'camino'
  if(camino == nullptr){
    cout << "Error: no hay memoria suficiente";
  }else{
    while (actual != -1){  //mientras que el nodo que estemos verificando sea distinto de -1. Es decir, mientras que no sea el nodo inicial
      p12 = &camino[f];
      *p12 = actual;  // uno de los vecinos del f-esimo elemento es actual
      actual = vecinos[actual];  // actual es uno de los vecinos de actual
      f++;
    }
  }


  // REVERSIR EL ARREGLO 'camino'
    int camino2[f];
    int *p13;  // Puntero que apunta a los valores de los vectores 'camino' y 'camino2'

    p13 = &camino[0];
    for (i = 0; i < f; i++){
        p13++; // Obtenemos el ultimo valor ddel vector 'camino'
    }
    p13--; // decrementamos 'p13'
    for ( i = 0; i < f; i++) {
        camino2[i] = *p13;  // El ultimo valor de 'camino' lo hacemos el primero de 'camino2';
    // el penultimo de 'camino', el segundo de 'camino2' y asi sucesivamente
        p13--;
    }
    p13 = &camino2[0];
    for ( i = 0; i < f; i++){
        camino[i] = *p13;  // Nos regresa 'camino' ahora ordenado
        p13++;
    }

  // IMPRESION A PANTALLA DE CAMINO MAS CORTO
    p13 = &camino[0]; // p13 guarda la direccion del primer valor de 'camino'
    for ( i = 0; i < f; i++){
        cout <<" " << *p13 <<endl;  // Imprime cada elemento del camino
        p13++;
    }

  // LIBERAR MEMORIA
  delete[] camino;
}
```