Notas de Algoritmos y Programación II

Ivan Litteri

Índice

1. Algoritmos de Ordenamiento

Algoritmos	Complejidad Temporal (promedio)	Complejidad Espacial	Estable	Comparativo	In-Place
Merge Sort	$\Theta(n \cdot log(n))$	$\Theta(n)$	Treu	True	False
Quick Sort	$\Theta(n \cdot log(n))$	$\Theta(n \cdot log(n))$	False	True	True
Selection Sort	$\Theta(n^2)$	$\Theta(1)$	False	True	True
Insertion Sort	$\Theta(n^2)$	$\Theta(1)$	True	True	True
Bubble Sort	$\Theta(n^2)$	$\Theta(1)$	True	True	True
Radix Sort	$\Theta(n \cdot k)$	$\Theta(n + k)$	True	False	True or False
Bucket Sort	$\Theta(n + k)$	$\Theta(n)$	True	False	$\approxeq$ False
Counting Sort	$\Theta(n + k)$	$\Theta(k)$	True	False	False
Heapsort	$\Theta(n \cdot log(n))$	$\Theta(1)$	False	True	True

1.1. Algoritmos de Ordenamiento Comparativos

Se define un ordenamiento comparativo a cualquier algoritmo de ordenamiento que determina el orden de ordenamiento comparando pares de elementos. Se basan en comparar elementos para poder ordenarlos. Se tiene como precondición que los datos a ordenar sean comparables. Tiene como cota mínima $\Omega(n \cdot log(n))$ por lo tanto no puede ser mejor que esto.

1.1.1. Merge Sort

```python
# Python program for implementation of MergeSort
def mergeSort(arr):
    if len(arr) > 1:

         # Finding the mid of the array
        mid = len(arr)//2

        # Dividing the array elements
        left = arr[:mid]

        # into 2 halves
        right = arr[mid:]

        # Sorting the first half
        mergeSort(left)

        # Sorting the second half
        mergeSort(right)

        i = j = k = 0

        # Copy data to temp arrays left[] and right[]
        while i < len(left) and j < len(right):
            if left[i] < right[j]:
                arr[k] = left[i]
                i += 1
            else:
                arr[k] = right[j]
                j += 1
            k += 1

        # Checking if any element was left
        while i < len(left):
            arr[k] = left[i]
            i += 1
            k += 1
```

```
37
38          while j < len(right):
39              arr[k] = right[j]
40              j += 1
41              k += 1
```

1.1.2. Quick Sort

```
1  # Python program for implementation of Quicksort Sort
2
3  # This function takes last element as pivot, places
4  # the pivot element at its correct position in sorted
5  # array, and places all smaller (smaller than pivot)
6  # to left of pivot and all greater elements to right
7  # of pivot
8
9  def partition(arr, low, high):
10     i = (low-1)         # index of smaller element
11     pivot = arr[high]     # pivot
12
13     for j in range(low, high):
14
15         # If current element is smaller than or
16         # equal to pivot
17         if arr[j] <= pivot:
18
19             # increment index of smaller element
20             i = i+1
21             arr[i], arr[j] = arr[j], arr[i]
22
23     arr[i+1], arr[high] = arr[high], arr[i+1]
24     return (i+1)
25
26 # The main function that implements QuickSort
27 # arr[] --> Array to be sorted,
28 # low  --> Starting index,
29 # high  --> Ending index
30
31 # Function to do Quick sort
32
33
34 def quickSort(arr, low, high):
35     if len(arr) == 1:
36         return arr
37     if low < high:
38
39         # pi is partitioning index, arr[p] is now
40         # at right place
41         pi = partition(arr, low, high)
42
43         # Separately sort elements before
44         # partition and after partition
45         quickSort(arr, low, pi-1)
46         quickSort(arr, pi+1, high)
```

1.1.3. Selection Sort

```
1  def selection_sort(arr):
2      for i in range(len(arr)):
3          # Find the minimum element in remaining
4          # unsorted array
5          min_idx = i
6          for j in range(i+1, len(arr)):
7              if arr[min_idx] > arr[j]:
8                  min_idx = j
```

```
        # Swap the found minimum element with
        # the first element
        arr[i], arr[min_idx] = arr[min_idx], arr[i]
```

1.1.4. Insertion Sort

```
def insertionSort(arr):
    for i in range(1, len(arr)):
        up = arr[i]
        j = i - 1
        while j >= 0 and arr[j] > up:
            arr[j + 1] = arr[j]
            j -= 1
        arr[j + 1] = up
    return arr
```

1.1.5. Bubble Sort

```
# Python program for implementation of Bubble Sort

def bubbleSort(arr):
    n = len(arr)
    # Traverse through all array elements
    for i in range(n):
        # Last i elements are already in place
        for j in range(0, n-i-1):

            # traverse the array from 0 to n-i-1
            # Swap if the element found is greater
            # than the next element
            if arr[j] > arr[j+1] :
                arr[j], arr[j+1] = arr[j+1], arr[j]
```

1.1.6. Heapsort

```
# Python program for implementation of heap Sort

# To heapify subtree rooted at index i.
# n is size of heap

def leftChild(i):
    return 2 * i + 1

def rightChild(i):
    return 2 * i + 2

def heapify(arr, n, i):
    largest = i  # Initialize largest as root
    l = leftChild(i)
    r = rightChild(i)

    # See if left child of root exists and is
    # greater than root
    if l < n and arr[largest] < arr[l]:
        largest = l

    # See if right child of root exists and is
    # greater than root
    if r < n and arr[largest] < arr[r]:
        largest = r

    # Change root, if needed
    if largest != i:
        arr[i], arr[largest] = arr[largest], arr[i]  # swap
```

```
30
31          # Heapify the root.
32          heapify(arr, n, largest)
33
34  # The main function to sort an array of given size
35
36  def heapSort(arr):
37      n = len(arr)
38
39      # Build a maxheap.
40      for i in range(n//2 - 1, -1, -1):
41          heapify(arr, n, i)
42
43      # One by one extract elements
44      for i in range(n-1, 0, -1):
45          arr[i], arr[0] = arr[0], arr[i]  # swap
46          heapify(arr, i, 0)
```

Este algoritmo de ordenamiento no es estable ya que cuando se aplica *downheap*, como las cosas suben desde distintas ramas se pierde completamente el orden relativo; también si vamos a desencolar, el que estaba último pasa a estar primero y luego baja, y puede bajar por cualquier otra rama, etc. Es un algoritmo *in-place*.

La complejidad de heapify es $\Theta(log(n))$, armar el heap cuesta $\Theta(n)$. Entonces su complejidad es $\Theta(n \cdot log(n))$.

1.2. Algoritmos de Ordenamiento No Comparativos

1.2.1. Counting Sort

Nos va a permitir ordenar datos numéricos discretos que estén en un rango acotado y conocido (se debe poder obtener fácil el mínimo y el máximo).

```
1   # Python program for counting sort
2
3   # The main function that sort the given string arr[] in
4   # alphabetical order
5   def countSort(arr):
6
7       # The output character array that will have sorted arr
8       output = [0 for i in range(256)]
9
10      # Create a count array to store count of inidividul
11      # characters and initialize count array as 0
12      count = [0 for i in range(256)]
13
14      # For storing the resulting answer since the
15      # string is immutable
16      ans = ["" for _ in arr]
17
18      # Store count of each character
19      for i in arr:
20          count[ord(i)] += 1
21
22      # Change count[i] so that count[i] now contains actual
23      # position of this character in output array
24      for i in range(256):
25          count[i] += count[i-1]
26
27      # Build the output character array
28      for i in range(len(arr)):
29          output[count[ord(arr[i])]-1] = arr[i]
30          count[ord(arr[i])] -= 1
31
32      # Copy the output array to arr, so that arr now
33      # contains sorted characters
```

```
    for i in range(len(arr)):
        ans[i] = output[i]
    return ans
```

1.2.2. Radix Sort

- Se usa cuando queremos ordenar cosas por distintos criterios.
- Trabaja con elementos a ordenar que tengan varios dígitos o componentes.
- Utiliza un ordenamiento auxiliar que **tiene que ser estable**. Idealmente, que sea lineal.
- Cada elemento tiene que tener la misma cantidad de cifras, o muy similar.
- Ordena (utilizando el arreglo auxiliar) de la cifra **menos significativa** a la cifra **más significativa**.

Necesitamos que cada uno de estos dígitos, cifras o componentes, se ordene internamente con un ordenamiento estable auxiliar (si no es estable entonces no va a funcionar) generalmente *counting sort*.

Sirve para números que estén en cualquier base, arreglos, cadenas y cualquier cosa que tenga varias cifras de distinto valor. Pero todos los elementos deben tener la misma cantidad de cifras.

```
# Python program for implementation of Radix Sort
# A function to do counting sort of arr[] according to
# the digit represented by exp.

def countingSort(arr, exp1):

  n = len(arr)

  # The output array elements that will have sorted arr
  output = [0] * (n)

  # initialize count array as 0
  count = [0] * (10)

  # Store count of occurrences in count[]
  for i in range(0, n):
    index = (arr[i] / exp1)
    count[int(index % 10)] += 1

  # Change count[i] so that count[i] now contains actual
  # position of this digit in output array
  for i in range(1, 10):
    count[i] += count[i - 1]

  # Build the output array
  i = n - 1
  while i >= 0:
    index = (arr[i] / exp1)
    output[count[int(index % 10)] - 1] = arr[i]
    count[int(index % 10)] -= 1
    i -= 1

  # Copying the output array to arr[],
  # so that arr now contains sorted numbers
  i = 0
  for i in range(0, len(arr)):
    arr[i] = output[i]

# Method to do Radix Sort
def radixSort(arr):

  # Find the maximum number to know number of digits
```

```
  max1 = max(arr)

  # Do counting sort for every digit. Note that instead
  # of passing digit number, exp is passed. exp is 10^i
  # where i is current digit number
  exp = 1
  while max1 / exp > 0:
    countingSort(arr, exp)
    exp *= 10


    # Driver code
    arr = [170, 45, 75, 90, 802, 24, 2, 66]

    # Function Call
    radixSort(arr)

    for i in range(len(arr)):
        print(arr[i])
```

1.2.3. Bucket Sort

- En este caso queremos ordenar algo que puede no tener un rango enumerable (discreto).
- Debe ser conocida la distribución de los datos.
- Los datos debe n ser uniformemente distribuidos.
- Bastante útil si no podemos aplicar *counting sort* o *radix sort* (ej: números decimales).

Vamos a querer ordenar algo que no tenga un rango enumerable, pero algo que si tenga una distribución conocida. Inicialmente lo que queremos es que esté uniformemente distribuido. A veces no vamos a tener eso pero si vamos a saber como está distribuido y lo vamos a poder uniformizar. Por ejemplo podemos ordenar un arreglo de números con decimales infinitos que sabemos que están uniformemente distribuidos (con radix y counting sort no podemos).

```
# Python3 program to sort an array
# using bucket sort
def insertionSort(arr):
    for i in range(1, len(arr)):
        up = arr[i]
        j = i - 1
        while j >= 0 and arr[j] > up:
            arr[j + 1] = arr[j]
            j -= 1
        arr[j + 1] = up
    return arr

def bucketSort(arr):
    aux = []
    slot_num = 10 # 10 means 10 slots, each
                  # slot's size is 0.1
    for i in range(slot_num):
        aux.append([])

    # Put array elements in different buckets
    for j in arr:
        index_b = int(slot_num * j)
        aux[index_b].append(j)

    # Sort individual buckets
    for i in range(slot_num):
        aux[i] = insertionSort(aux[i])
```

```
28
29      # concatenate the result
30      k = 0
31      for i in range(slot_num):
32          for j in range(len(aux[i])):
33              arr[k] = aux[i][j]
34              k += 1
35      return arr
```

1.3. Propiedades de los Algoritmos de Ordenamiento

1.3.1. Estabilidad en algoritmos de ordenamiento

En un algoritmo de ordenamiento estable, los elementos que coinciden en su clave de ordenamiento aparecen, en el arreglo de salida, en el mismo orden relativo que en el arreglo original.

Un algoritmo de ordenamiento es estable cuando se asegura que el orden relativo de los elementos de *misma clave* (que son iguales para el ordenamiento) es idéntico a la salida que a la entrada.

1.3.2. Algoritmos de ordenamiento *in place*

Un algoritmo de ordenamiento es In-Place si ordena los elementos sobre el arreglo original. Un ejemplo de un algoritmo que no lo es, es *Merge Sort* ya que éste realiza una copia entera del arreglo para realizar el *merge*.

Un algoritmo de ordenamiento es in-place cuando ordena directamente sobre el arreglo original (utiliza $\Omega(1)$ de espacio adicional).

2. Grafos

2.1. Corte Mínimo

- El corte mínimo en una red es el peso total (mínimo) que necesitamos desconectar para que un grafo deje de estar conectado (conexo para grafos no dirigidos, déilmente conexo para dirigido).
- Si el grafo es no pesado corresponde a la cantidad de aristas (como considerar todos como peso = 1).
- Esto se aplica a cualquier tipo de grafo, pero para redes de flujo puede tener una ventaja.

2.1.1. Teorema *max flow min cut*

- Si el grafo corresponde a una red de flujo, entonces el corte mínimo tiene capacidad igual al flujo máximo.
- Va a suceder que la fuente y el sumidero se encuentren en sets opuestos.

2.2. Como obtener el corte mínimo

1. Agaramos nuestro grafo residual.
2. Vemos todos los vértices a los que llegamos desde la fuente.
3. Todas las aristas (del grafo original) que vayan de un vértice al que podamos llegar (en el residual) a uno que no (ídem), son parte del corte.

3. Algoritmos de Grafos

3.1. Prim

3.1.1. Paso a paso

1. Comienza en un vértice aleatorio, encola en un heap todas sus aristas.
2. El vértice origen queda como visitado.
3. Mientras el heap no está vacío, saca una arista.
4. Si ambos vértices de la arista fueron visitados, se descarta la arista (formaría un ciclo).
5. Caso contrario, se agrega la arista al árbol, y se encolan todas las aristas del nuevo vértice visitado.
6. Volvemos al paso 3.

3.1.2. Algoritmo

```
def mst_prim(grafo):
    # Elijo un vertice aleatorio
    v = grafo.vertice_aleatorio()
    # Creamos el conjunto de visitados
    visitados = set()
    visitados.add(v)
    # Creamos un heap
    q = Heap()
    # Creamos un arbol (mst)
    arbol = grafo_crear(grafo.obtener_vertices())
    # Por cada arista del origen, la encolamos.
    for w in grafo.adyacentes(v):
        q.encolar((v, w), grafo.peso_arista(v, w))
    # Mientras no este vacia la cola,
    while not q.esta_vacia():
        # desencolamos.
        v, w = q.desencolar()
        # Si el adyacente fue visitado, continuamos.
        if w in visitados:
            continue
        # Sino agregamos la arista al arbol.
        arbol.agregar_arista(v, w, grafo.peso_arista(v, w))
        visitados.add(w)
        # Encolamos los adyacentes al nuevo visitado.
        for u in grafo.adyacentes(w):
            if u not in visitados:
                q.encolar((w, u), grafo.peso_arista(w, u))

    return arbol
```

3.1.3. Complejidad

- Obtener el vértice aleatorio es $\Theta(1)$, $\Theta(V)$ en el peor de los casos.
- Crear el árbol a partir de los vértices deñ grafo original es $\Theta(V)$.
- Ver los adyacentes de un vértice en particular $\Theta(1)$ (despreciable).
- En el peor de los casos, encolamos y desencolamos todas las aristas del grafo. Entonces, costará $\Theta(E \cdot \log(V))$.

3.2. Kruskal

3.2.1. Paso a paso

1. Ordenamos las arista de menor a mayor.
2. Por cada arista (en ese orden), si los vértices no están en una misma componente conexa (dentro del árbol), agregamos la arista, y ahora ambos vértices están en la misma componente conexa.

3.2.2. TDA UnionFind (o clase de equivalencia)

- Por cada elemento tenemos una posición en un arreglo, que se autoreferencia.
- Cuándo unimos, simplemente hacemos que en vez de autoreferenciarse, uno de los dos pase a apuntar al otro.
- Para saber en qué clase de equivalencia está un elemento, preguntamos a quién referencia dicho elemento, hasta encontrar a uno que se autoreferencie.
- Para planchar la estructura, al hacer el paso anterior, una vez que encuentro a quien pertenece un elemento, hacemos que todo el camino directamente referencie al autoreferenciado (acortamos el camino).
- Funciona en $\Theta(1)$ (aprox.).

3.2.3. Algoritmo

```
def obtener_aristas(grafo):
    aristas = []
    visitados = set()
    for v in grafo.obtener_vertices():
        visitados.add(v)
        for w in grafo.adyacentes(v):
            if w not in visitados:
                aristas.append((v, w))
    return aristas

def mst_kruskal(grafo):
    # Creamos los conjuntos.
    conjuntos = UnionFind(grafo.obtener_vertices())
    # Ordenamos las aristas de menor a mayor
    aristas = sorted(obtener_aristas(grafo))
    arbol = grafo_crear(grafo.obtener_vertices())
    # Por cada arista (en ese orden),
    for a in aristas:
        v, w, peso = a
        # Si son iguales, salteamos.
        # Agregar esta arista generaria un ciclo.
        if conjuntos.find(v) == conjuntos.find(w):
            continue
        # Sino, agregamos la arista.
        arbol.agregar_arista(v, w, peso)
        # Unimos los conjuntos.
        conjuntos.union(v, w)
    return arbol
```

3.2.4. Complejidad

- Crear el conjunto a partir de los vértices del grafo es $\Theta(V)$.
- Ordenar como mucho es $\Theta(E \log(V))$.
- El for es mínimo $\Theta(E)$, en el peor de los casos es $\Theta(E \log(V))$.

- (Si tuviésemos más información de las aristas, podríamos usar un ordenamiento no comparativo y el Algoritmo de Kruskal podría ser más rápido que el de Prim.).

3.3. Ford-Fulkerson

3.3.1. Esquema General

```
def flujo(grafo, fuente, sumidero):
    Inicializamos el flujo por toda arista a 0
    Mientras haya un camino en la red residual:
        aumentamos el flujo acorde al camino
```

3.3.2. Paso a paso

- Inicialmente defino para cada arista el flujo en 0.
- Como necesito trabajar con la red residual lo que hago es copiar el grafo original. Inicialmente el grafo residual es igual al grafo original porque en el grafo residual tenemos 2 tipos de aristas, las mismas al las del grafo original (en el mismo sentido), pero con la capacidad restante, y por el otro lado, tenemos las aristas de retorno, inicialmente en 0.
- Mientras haya un camino en la red residual (obtener camino, mediante cualquier algoritmo). Esto es lo que define que en realidad este algoritmo es en realidad un método, ya que no exige un algoritmo específico para encontrar este camino.
- Para actualizar el flujo, hay que obtener la capacidad que tengo que actualizar, y eso es el mínimo peso de todas las aristas, porque no puede pasar más del mínimo de unidades para la arista de dicho peso.
- Luego itero todos los vértices para obtener las aristas, fijándome si el vértice en la posición `i` está entre los vértices del grafo original, ya que necesito saber si la arista con la que voy a trabajar es la arista original del grafo, o es la arista de retorno (del grafo residual).
- Para el primer caso, entonces queremos que vaya más flujo, entonces actualizamos el flujo de la arista actual, incrementándole el flujo mínimo obtenido anteriormente.
- Para el segundo caso, entonces para la arista de retorno (inversa), que si existe en el grafo, entonces tiene que volver flujo, para aumentar el flujo global, por lo tanto le disminuyo el flujo restándole el flujo mínimo obtenido anteriormente y actualizo en el sentido inverso.
- Para actualizar la red residual, lo primero que me tengo que fijar es cuánto es el peso anterior de lo que estamos actualizando, y si el peso anterior es igual a lo que estoy usando, la saco directamente, sino le resto el valor. Y si no tiene la arista recíproca, la agrego en el valor actual, sino le cambio el peso.
- Al final, devuelvo el flujo.

3.3.3. Algoritmo

```
def actualizar_grafo_residual(grafo_residual, u, v, valor):
    # Obtengo el peso anterior
    peso_anterior = grafo_residual.peso(u, v)
    # Si la arista tiene el mismo peso, entonces la saco.
    if peso_anterior == valor:
        grafo_residual.remover_arista(u, v)
    # Sino, le cambio el peso
    else:
        grafo_residual.cambiar_peso(u, v, peso_anterior - valor)
    # Si no existe la arista de retorno, la creo.
    if not grafo_residual.son_adyacentes(v, u):
        grafo_residual.agregar_arista(v, u, valor)
```

```
    # Sino, le cambio el peso.
    else:
        grafo_residual.cambiar_peso(v, u, peso_anterior + valor)

def flujo(grafo, s, t):
    # Inicializamos el flujo de cada arista en 0.
    flujo = {}
    for v in grafo:
        for w in grafo.adyacentes(v):
            flujo[(v, w)] = 0
    # Creamos una copia del grafo original.
    grafo_residual = copiar(grafo)
    # Mientras haya un camino en la red residual,
    while (camino = obtener_camino(grafo_residual, s, t)) is not None:
        # Actualizamos el flujo acorde al camino.
        capacidad_residual_camino = min_peso(grafo, camino)
        for i in range(1, len(camino)):
            # Si la arista pertenece al grafo original se trata de una arista
            # a la que le queremos aumentar el flujo.
            if camino[i] in grafo.adyacentes(camino[i-1]):
                flujo[(camino[i-1], camino[i])] += capacidad_residual_camino
                actualizar_grafo_residual(grafo_residual, camino[i-1], camino[i],
    capacidad_residual_camino)
            # Sino se trata de una arista de retorno, y lo queremos disminuir
            # el flujo para aumentarlo globalmente.
            else:
                flujo[(camino[i-1], camino[i])] -= capacidad_residual_camino
                actualizar_grafo_residual(grafo_residual, camino[i], camino[i-1],
    capacidad_residual_camino)
    # Devolvemos el flujo
    return flujo
```

3.3.4. Complejidad

- Todo depende de cómo elegimos buscar el camino.
- Puede demorar muchísimo, o ni siquiera converger.
- Teorema Edmonds-Karp: si se hace por BFS el algoritmo suempre funciona y es $\Theta(V \cdot E^2)$.

4. Preguntas

1. ¿Qué implica que un algoritmo de ordenamiento sea estable?
2. ¿Qué implica que un algoritmo de ordenamiento sea in-place?
3. ¿Qué implica que un algoritmo de ordenamiento sea comparativo? ¿En qué condiciones puede utilizarse?
4. ¿Es heapsort un algoritmo de ordenamiento estable?
5. ¿Es heapsort un algoritmo de ordenamiento in-place?
6. ¿Es heapsort un algoritmo de ordenamiento comparativo?
7. En un árbol binario, dado un nodo con dos hijos, explicar por qué su predecesor en el recorrido inorder no puede tener hijo derecho, y su sucesor (también, en el recorrido inorder) no puede tener hijo izquierdo.
8. Para implementar un TDA Cola de prioridad de minimos, nos proponen la siguiente solución: usar una estructura enlazada que mantendremos ordenada (con lo cuál, el mínimo es el primer elemento). ¿Es una buena solución en el caso general? Justificar. Comparar contra la implementación de cola de prioridad vista en clase.
9. ¿Cuál es la complejidad de heapsort? ¿es posible que conociendo información adicional logremos mejorar su complejidad?
10. ¿Es mergesort un algoritmo de ordenamiento estable?
11. ¿Es siempre mejor utilizar Counting Sort para ordenar un arreglo de números enteros por sobre utilizar un ordenamiento por Selección?
12. Definimos el siguiente algoritmo de División y Conquista que, esperamos, nos permita obtener el árbol de tendido mínimo de un Grafo. El algoritmo divide el conjunto de los vértices en dos, y llama recursivamente con un nuevo grafo sólo conformado por los vértices de una mitad, y las aristas que unen vértices de esa mitad. Llama para ambas mitades, y resuelve recursivamente el árbol de tendido mínimo de cada mitad (el caso base sería un grafo con un único vértice). Luego de resolver el árbol de tendido mínimo de ambas mitades, une ambos con la arista mínima que une los vértices de un lado con los del otro. Indicar y Justificar detalladamente cuál sería la complejidad de dicho algoritmo. ¿El algoritmo propuesto permite obtener el árbol de tendido mínimo? (suponer que el grafo es conexo). En caso de ser cierto, justificar, en caso de ser falso dar un contraejemplo.
13. Queremos comparar el heap d-ario (en el que cada nodo tiene hasta d hijos) con el heap binario común, que tiene hasta 2 hijos.
    a ¿Cómo se representa un heap d-ario en un arreglo? ¿Cómo calculo el padre de un nodo, o los d hinos de un nodo?
    b ¿Cuál es la altura de un heap d-ario con n elementos, en términos de n y d?
    c Dar una implementación eficiente de encolar (para un heap d-ario de máximos). Analizar en términos de n y d su eficiencia. Definir si tiene sentido la implementación de un heap d-ario por sobre uno binario.
14. Explicar cómo harías para implementar un TDA Pila utilizando internamente un TDA Heap. Indicar el orden de cada una de las primitivas.
15. Nos dan para elegir entre los siguientes 3 algoritmos para solucionar el mismo problema ¿Cuál elegirías? Justificar calculando el orden de los algoritmos:

a El algoritmo A reuelve el problema dividiéndolo en 5 subproblema de la mitad del tamaño, resolviendo cada subproblema de forma recursiva, y combinando las soluciones en tiempo real.

b El algoritmo B resuelve el problema dividiéndolo en 9 subproblemas de tamaño $\frac{n}{3}$, resolviendo cada subproblema de forma recursova y combinando las soluciones en tiempo cuadrático de n.

c El algoritmo C resuelve todos los problema de tamaño n eligiendo un subproblema de tamaño $n-1$ en tiempo $\Theta(n)$ luego resolviendo recursivamente ese subproblema.

5. Verdadero o Falso

1. a) Si en los ABBs con *misma* función de comparación se guardan los mismos elementos en *diferente* orden, tendrán el mismo inorder.
   b) Si dos ABBs *diferente* función de comparación se guardan los mismos elementos en el *mismo* orden, tendrán el mismo inorder.
   c) Si en dos ABBs con *misma* función de comparación se guardan los mismos elementos en *diferente* orden, tendrán el mismo preorder.

2. a) Se puede aplicar el algoritmo de Prim apra obtener el Árbol de Tendido Mínimo en un grafo con aristas de pesos negativos.
   b) Si en vez del árbol de tendido mínimo de un grafo buscáramos obtener el árbol de tendido máximo del mismo, podríamos modificar lévemente el Algoritmo de Kruskal para poder obtener dicho árbol.
   c) Si el grafo es no pesado, de todas formas los Algoritmos de Prim y Kruskal se pueden utilizar y son la mejor opción para obtener un árbol de tendido.

3. a) Dado un grafo G no dirigido con pesos positivos, y T un Árbol de tendido mínimo de dicho grafo, si se elevan al cuadrado los pesos de las aristas de G, T sigue siendo un árbol de tendido mínimo de G.
   b) Dado un grafo G no dirigido con pesos positivos, y P un camino de costo mínimo de v a todos los demás vértices de G, si se leevan al cuadrado los pesos de las aristas de G, P sigue siendo camino mínimo.

4. a) Siempre que se trabaje con datos discretos me conviene ordenar usando Counting Sort.
   b) Para que RadixSort funcione correctamente, el algoritmo auxiliar a utilziar debe ser in-place.
   c) EN Hpscotch hashing puede darse el caso que una búsqueda no sea en tiempo constante, según cómo sea la función de hashing.

5. Implementar un algoritmo que reciba un grafo dirigido y nos devuelva la cantidad de componentes débilmente conexas de este. Indicar y justificar la complejidad del algoritmo implementado.

   a) Para ordenar correctamente utilizando RadixSort, se debe ordenar los datos de la *cifra* menos significativa a la más significativa.
   b) BucketSort será lineal en tanto y en cuanto el algoritmo auxiliar para ordenar los buckets sea de tiempo lineal.
   c) En un AVL, tras una inserción, es encesario buscar en todo el árbol si se generó un desbalanceo.

6. a) Si queremos ordenar por año todos los sucesos importantes ocurridos en el mundo desde el año 0 hasta la actualidad, CountingSort es una buena alternativa. Consideremos que todos los años hay sucesos importantes, y en este en particular sólo faltó la invasión zombie (aunque todavía queda diciembre...).
   b) RadixSort logra una complejidad lineal (dadas las condiciones adecuadas) reduciendo la cantidad de comparaciones que realiza entre los datos.
   c) Un potencial problema de Hash & Displace es que requerimos de varias funciones de hashing, y ni así podemos estar seguros que todo funcionará (salvo que sean realmente muchas).

7. a) En un grafo bipartito no pueden haber ciclos con cantidad impar de vértices que lo compongan.
   b) En un árbol (grafo no dirigido, conexo y sin ciclos) todos los vértices con al menos dos adyacentes son puntos de articulación.

c) En un grafo dirigido, no existe camino de un vértice v de una componente fuertemente conexa hacia un vértice w de otra componente fuertemente conexa.

8. a) Al resolver el problema de maximización de flujo buscamos caminos entre la fuente y el sumidero en la red residual. Si en dicho camino se utiliza una arista que no existe en el grafo original, significa que no podemos considerar ese camino para aumentar el flujo total.

   b) En un grafo no dirigido con pesos negativos, podemos aplicar el algoritmo de Bellman-Ford para obtener el camino mínimo desde un vértice hacia todos los demás.

   c) Podemos calcular el Árbol de tendido mínimo de un grafo no dirigido que tenga pesos negativos.

9. a) Al resolver el problema de maximización de flujo buscamos caminos entre la fuente y el sumidero en la red residual. Si en dicho camino se utiliza una arista que no existe en el grafo original, significa que el flujo en la arista original (la del grafo original) se debe aumentar.

   b) No podemos resolver el problema de maximización de flujo si hay ciclos en el grafo.

   c) Si calculo el árbol de tendido mínimo de un grafo, y luego sobre ese árbol obtengo los caminos mínimos de un vértice a todos los demás, los caminos pueden coincidir con los caminos mínimos en el grafo original, pero no siempre sucederá. Aclaración: dar los ejemplos o contraejemplos que justifiquen/ejemplifiquen tu respuesta.

10. a) Al resolver el problema de maximización de flujo buscamos caminos entre la fuente y el sumidero en la red residual. Si en dicho camino se utiliza una arista que no existe en el grafo original, significa que el flujo en la arista original (la del grafo original) se debe disminuir, por lo que el flujo total también disminuye.

    b) Todo grafo dirigido con un vértice de grado de entrada 0 y un vértice de salida 0 es Red de Flujo.

    c) Siempre podemos calcular el camino mínimo desde un vértice en un grafo dirigido que tenga pesos negativos.

6. Resoluciones

6.1. Preguntas

1. Que un algoritmo de ordenamiento sea estable implica que los elementos que coinciden en su clave de ordenamiento aparecen, en el arreglo de salida, en el mismo orden relativo que en el arreglo original.

2. Que un algortimo de ordenamiento sea in-place implica que éste ordena directamente sobre el arreglo original (utiliza $\Omega(1)$ de espacio adicional).

3. Que un algoritmo de ordenamiento sea comparativo implica que éste determina el orden de ordenamiento comparando pares de elementos. Se basan en comparar elementos para poder ordenarlos. Se tiene como precondición que los datos a ordenar sean comparables. Tiene como cota mínima $\Omega(n \cdot log(n))$ por lo tanto no puede ser mejor que esto.

4. Heapsort no es estable ya que cuando se aplica *downheap*, como las cosas suben desde distintas ramas se pierde completamente el orden relativo; también si vamos a desencolar, el que estaba último pasa a estar primero y luego baja, y puede bajar por cualquier otra rama, etc.

5. Heapsort es in-place ya que *heapify*, modifica al arreglo original, y *downheap* también.

6. Heapsort es un algoritmo de ordenamiento comparativo ya compara los valores del arreglo para darle la propiedad de heap.

7. Dado un nodo con dos hijos, su predecesor en el recorrido inorder no puede tener hijo derecho, ya que si lo tuviese, éste no sería su predecesor en dicho orden de recorrido. Lo mismo para el sucesor en caso de tener hijo izquierdo. (Agregar ejemplo).

8. Ésta implementación no es una buena solución en el caso general. Para lograr desencolar en $\Theta(1)$ se sacrifica el costo de inserción, ya que pasará a costar $\Theta(n)$. Respecto a la implementación de la cola de prioridad vista en clase, la inserción en el heap cuesta $\Theta(\log(n))$ y en la estructura enlazada cuesta $\Theta(n)$ siendo n la antidad de elementos del arreglo. Y, el desencolado en el heap cuesta $\Theta(\log(n))$ y en la estructura enlazada cuesta $\Theta(1)$ siendo n el número de elementos del arreglo.

9. La complejidad de heapsort (*heapify* cuesta $\Theta(n)$ y *downheap* cuesta $\Theta(n \log(n))$) utilizando un algoritmo de ordenamiento comparativo es $\Theta(n \log(n))$. Teniendo información adicional sobre los elementos del arreglo, podemos llegar a utilizar un algoritmo de ordenamiento no comparativo y mejorar su complejidad.

10. Mergesort es un algorimo de ordenamiento estable.

11. No siempre es mejor utilizar Counting Sort para ordenar un arreglo de números enteros por sobre utilizar un ordenamiento por Selcción, porque si el rango de elementos en el arreglo es infinito, ya no nos sirve usar Counting Sort, pero siempre y cuando el rango sea finito y conocido, nos conviene usar Counting Sort.

12. El algoritmo propuesto no permite obtener el árbol de tendido mínimo, ya que va a generar un árbol distinto dependiendo de la distribución de los elementos en el arreglo de vértices inicial (y de peso total distinto) (Dar ejemplo con grafo K_5 con $V = \{A, B, C, D, E\}$, y $E = \{(A, B, 1), (A, C, 3), (A, D, 5), (A, E, 10), (B, C, 20), (B$ con el arreglo de vértices inicial $[A, B, C, D, E]$, y luego con el arreglo de vértices inicial $[A, E, D, B, C]$).
Y la complejidad es $\Theta(V)$ ya que obtener los vértices del grafo es $\Theta(V)$ y la función recursiva, por Teorema Maestro $\Theta(\log(V) + \log(n))$.

13. Calculo de la posición del padre:
$$padre = (pos - 1)/d$$
Calculo de la posición de un d hijo:
$$hijo = (pos \cdot d) + n \therefore 1 \leq n \leq d$$

14. Sin palabras.

15. *a*) Usando el Teorema Maestro:

$$T(n) = 5\left(\frac{n}{2}\right) + \Theta(n) \tag{1}$$

$$A = 5, B = 2, C = 1 \tag{2}$$

$$\log_B(A) > C \Rightarrow \Theta(n^{C \log_B(A)}) \tag{3}$$

$$\Theta(n^{\log_2(5)}) \approx \Theta(n^2) \tag{4}$$

b) Usando el Teorema Maestro:

$$T(n) = 9\left(\frac{n}{3}\right) + \Theta(n^2) \tag{5}$$

$$A = 9, B = 3, C = 2 \tag{6}$$

$$\log_B(A) = C \Rightarrow \Theta(n^C \log_B(n)) = \Theta(n^C \log(n)) \tag{7}$$

$$\Theta(n^2 \log(n)) \tag{8}$$

c) $\Theta(n^2)$

6.2. Verdadero o Falso

1. a) **TRUE**. Si la función de comparación es la misma, no importa en que orden se inserten los elementos, al tener la misma función de comparación dejará los árboles en el mismo orden relativo para el recorrido inorder.

   b) **FALSE**. Si la función de comparación es diferente en dos ABBs y se insertan los mismos elementos en el mismo orden, no tendran el mismo inorder ya que, no guardará los elementos en el mismo orden relativo a causa de la diferencia entre funciones de comparación.

   c) **FALSE**. Dado este caso, si en uno de los ABBs inserto un nodo ."A"primero, y en el otro inserto "B"primero, ya tengo un contraejemplo.

2. a) **TRUE**. No hay problema en hacer esto, funcionaría igual que teniendo todos los pesos positivos.

   b) **TRUE**. Si ordeno de mayor a menor las aristas debería poder obtener un Árbol de Tendido Máximo.

   c) **FALSE**. Usando BFS o DFS, con el diccionario de padres puedo armar un árbol de tendido, y con complejidad $\Theta(V + E)$ contra la complejidad $\Theta(E \log(V))$ de Kruskal y Prim.

3. a) **TRUE**. Como aclara que los pesos de las aristas son positivas, entonces al elevar dichos pesos al cuadrado el árbol de tendido mínimo es el mismo.

   b) **FALSE**. Contraejemplo: $G = (V, E), V = \{A, B, C\}, E = \{(A, B, 1), (B, C, 1), (A, C, 2)\}, P_{A-C} = [A, C]$ en G, pero $P_{A-C} = [A, B, C]$ en G^2.

4. a) **FALSE**. Que los valores sean discretos no es condición suficiente para utilizar Counting Sort, necesitaríamos saber además si el rango es acotado.

   b) **FALSE**. Una condición necesaria para que RadixSort funcione correctamente es que el algoritmo de ordenamiento auxiliar sea estable; la condición de ser in-place no es necesaria. Sólo se usa un algoritmo de ordenamiento auxiliar in-place cuando la memoria es algo a tener en cuenta.

   c) **FALSE**. Las búsquedas siempre son $\Theta(1)$ siempre que k tenga el valor que tenga. La función de hashing va a variar la complejidad de la resolución de colisiones.

5. a) **TRUE**. Es la forma en la que RadixSort ordena.

   b) **FALSE**. No es condición necesaria. Con que el algoritmo de ordenamiento auxiliar sea $\Theta(n \log(n))$, ya que se puede despreciar el crecimiento logarítmico con valores no tan grandes de B (cantidad de baldes) quedando $\Theta(n)$.

   c) . Sólo se necesita revisar el camino de inserción.

6. a) **TRUE**. Tenemos datos discretos en un rango acotado y conocido, siendo k el número de sucesos y n el rango (0 a 2021) entonces, Counting Sort es una buena opción, y la complejidad sería $\Theta(n + k)$ pero $k << n \Rightarrow \Theta(n)$. Además es estable.

   b) **FALSE**. RadixSort es un algoritmo de ordenamiento no comparativo. Para lograr una complejidad lineal $\Theta(d \cdot (n + k))$, siendo k el rango de los valores, d la cantidad de "dígitos", y n la cantidad de elementos. Si $d << \log(n) \wedge k << n \Rightarrow \Theta(n)$.

   c)

7. a) **TRUE**. Si un grafo contiene un ciclo de n vértices, o es un ciclo de n vértices, si n es par entonces si es bipartito porque puedo colorear los vértices del mismo con 2 colores, obteniendo dos conjuntos disjuntos de vértices de un mismo conjunto que no se relacionan entre sí. Usando como ejemplo Un ciclo de 2 vértices con aristas: $V = \{A, B\}, E = \{(A, B), (B, A)\}$ puedo usar un color para A y otro para B y obtener dos conjuntos disjuntos $color1 = \{A\}, color2 = \{B\}$. Usando como ejemplo un ciclo con n impar: $V = \{A, B, C\}, E = \{(A, B), (A, C), (B, C)\}$, si coloreo con 2 colores alternando en el ciclo (para verificar que sea bipartito), me encuentro con que hay 2 vértices de un color relacionándose entre sí, cosa que no cumple con la definición de grafo bipartito.

b) **TRUE**. En un árbol, todo vértice que tenga 2 adyacentes es punto de articulación, porque, es el único vértice por el cuál se puede llegar a sus adyacentes (por definición de árbol).

c) **FALSE**. En un grafo dirigido, puede no existir un camino de un vértice V a de una componente fuertemente conexa hacia un vértice W de otra componente fuertemente conexa, pero es no es cierto para todo par de vértices V, W. Contraejemplo: tengo una componente fuertemente conexa formada por vértices A, B, C y otra componente fuertemente conexa formada por E, F, G, existe una arista $(B- > E)$, gracias a esta arista, existe un camino desde un vértice V de la primera componente conexa a cualquier vértice V de la segunda componente conexa.

8. a) **FALSE**. La arista de la habla la consigna, es una arista de retorno (no pertenece al grafo original). La proposición .ªrista que no existe en el grafo original implica que no se puede considerar para aumentar el flujo total.es falsa, porque, la arista de retorno se tiene en cuenta para reducir el flujo de ese camino para aumentar el flujo total.

b) **TRUE**. El algoritmo de Bellman-Ford se puede aplicar en grafos (dirigidos como no dirigidos) que acepten pesos negativos, siempre y cuando no existan ciclos.

c) **TRUE**. Utilizando cualquier algoritmo para obtener el árbol de tendido mínimo podemos obtenerlo de un grafo dirigido. No hay impedimento alguno.

9. a) **FALSE**. Debe disminuir el flujo de la arista original, ya que si el algoritmo encontró una arista de estas características, se trata de una arista de retorno, entonces al actualizar el grafo, disminuirá el flujo de la arista original para aumentar el flujo total.

b) **FALSE**. Si hay ciclos de más de dos vértices no hay problema para resolver la maximización de flujo. Y si hay ciclo de 2 vértices (aristas antiparalelas), el problema puede ser resuelto agregando un vértice entre ellos, para convertirlo en un ciclo de 3 vértices.

c) **TRUE**. Es poco probable, pero puede suceder. No siempre es cierto ya que es posible que al armar el árbol de tendido mínimo me pierda algúna arista que signifique algún camino mínimo entre un vértice y otro. Ejemplo: $G = (V, E), V = \{A, B, C, D\}, E = \{(A, B, 2), (A, D, 1), (B, C, 1), (C, D, 1)\}$ utilizando el Algoritmo de Dijkstra para calcular el camino mínimo desde B hasta A obtenemos $P_{B-A} = [B, A, 2]$. Un árbol de tendido mínimo sería $T = (V', E')$, $V' = V$, $E' = \{(A, D, 1), (D, C, 1), (C, B, 1))\}$.

10. a) **FALSE**. La primera proposición es verdadera, la segunda también, pero la tercera no. Esto es, cuando el algoritmo encuentra una arista que no pertenece al grafo original, la arista paralela debe disminuir el flujo, pero no para disminuir así el flujo total del grafo, sino para aumentarlo.

b) **TRUE**. Se trata de la Red de Flujo trivial.

c) **FALSE**. Contraejemplo: si el grafo tiene ciclos negativos, $\nexists$ camino mínimo.

6.3. Finales

6.4. final-2020c2-1

Consigna

1. Implementar en C una primita `lista_t* lista_slice(const lista_t* original, size_t inicio, size_t fin)` que devuelva una **nueva** lista con los elementos de la original que se encuentran entre las posición `inicio` y `fin` (incluyendo el que se encuentra en la posición `inicio`, excluyendo el que se encuentra en la posición `fin`). Considerar la primera posición de la lista como la posición 0. En caso que `fin` sea menor o igual a `inicio`, devolver una lista vacía. En caso que `fin` sea mayor a la cantidad de elementos de la lista, simplemente agregar hasta el último. Si `inicio` es mayor a dicha última posición válida, devolver una lista vacía. Indicar y justificar la complejidad del algoritmo implementado. Ejemplos:

2. Implementar un algoritmo que determine si un grafo no dirigido es un grafo disperso. Se considera que un grafo es disperso si tiene menos del 5 % de las aristas totales que puede llegar a tener. Indicar y justificar el orden del algoritmo, si el grafo se encuentra implementado con matriz de adyacencia.

3. A un grafo pesado y no dirigido se le obtuvo un árbol de tendido mínimo. Sobre dicho árbol se quiere obtener caminos mínimos. Implementar un algoritmo que reciba dicho árbol y un vértice v y obtenga los caminos mínimos desde v hacia todos los demás vértices (sobre dicho grafo en forma de árbol, no el grafo original) en tiempo lineal de la cantidad de vértices y aristas. Justificar la complejidad del algoritmo implementado.

Resolción

```c
static void copy_slice(lista_t *original, lista_t *slice, size_t inicio, size_t fin)
{
    // Recorro desde el primero en la lista hasta que llegue al fin o al final
    // de la lista guardando los elementos en el rango, en el slice.
    nodo_t *actual = original->primero;
    size_t largo = original->largo;
    for (size_t pseudo_indice = 0; pseudo_indice < largo && pseudo_indice <= fin;
    pseudo_indice++)
    {
        if (pseudo_indice >= inicio && pseudo_indice <= fin)
        {
            lista_insertar_ultimo(slice, actual->dato);
        }
        actual = actual->proximo;
    }
}

lista_t *lista_slice(const lista_t *original, size_t inicio, size_t fin)
{
    // Si el fin o inicio no son posiciones validas devuelve NULL.
    if (fin <= inicio || inicio >= original->largo)
    {
        return NULL;
    }

    lista_t *slice;

    if ((slice = lista_crear()) == NULL)
    {
        return NULL;
    }

    copy_slice(original, slice, inicio, fin) // \Theta(n)

```

```
    // Devuelvo el slice.
    return slice;
}

/* Complejidad
    La mayor complejidad se la lleva la funcion auxiliar copy_slice que cuesta
en el peor caso \Theta(n) siendo n el largo de la lista original menos uno.
*/
```

```
def es_disperso(grafo) -> bool:
    no_aristas = 0
    no_vertices = 0
    for v in grafo:
        no_vertices += 1
        for _ in grafo.adyacentes(v):
            no_aristas += 0.5
    return (no_vertices*(no_vertices-1))/2 *.05 > no_aristas

''' Complejidad
    Recorrer cada vertice del grafo es \Theta(V), y para cada vertice, recorrer sus
adyacentes, si el grafo esta implementado con una matriz de adyacencia, es \Theta(V)
como mucho, y el for itera E veces, entonces la complejidad de la iteracion del
for interno es \Theta(V*(V+E))entonces la complejidad de es_disperso queda \Theta(V^2+V
    *E).
'''
```

3.

```
def bfs_shortest_path(graph: Graph, origin: str, destination: str) -> list:
    visited = set()
    vertices = Queue()
    vertices.enqueue(origin)
    visited.add(origin)
    paths = []
    path = [origin]
    while not vertices.is_empty():
        v = vertices.dequeue()
        if v == destination:
            break
        for w in graph.adjacents(v):
            if w not in visited:
                path.append(w)
                visited.add(w)
                vertices.enqueue(w)

    return path

def caminos_minimos_mst(mst: Graph, v: str) -> list:
    return [bfs_shortest_path(mst, v, w) for w in mst]
```

6.5. final-2019c2-4

6.5.1. Consigna

1. Implementar una función que dados dos arreglos de n y m elementos, respectivamente, devuelva si ambos arreglos tienen todos sus elementos diferentes entre sí en $\Theta(n + m)$. Justificar el orden de la función implementada.

2. Implementar un algoritmo que permita definir si un Grafo es un pseudo-bosque. Un Grafo es un pseudo-bosque si tiene a lo sumo un ciclo (considerar que el grafo podría no ser conexo). El algoritmo debe ejecutar de forma **lineal** tanto en la cantidad de vértices como de aristas. Justificar el orden del algoritmo implementado.

3. Implementar un algoritmo que permita ordenar un arreglo de n números cuyos valores van de 0 a $n^2 - 1$ en tiempo **lineal**. *Ayuda*: Para resolver este problema, no es conveniente ver los números necesariamente en base 10.

4. Implementar en C una función `lista_t* lista_map(lista_t*, void* (*f) (void*))` que, utilizando el **iterador interno** de la lista, permita crear **una nueva lista** con cada uno de los elementos de la recibida por parámetro, luego de haberles aplicado la función `f`. Pista: recordar que la primitiva `lista_iterar` recibe un puntero `void* extra` que puede ser un puntero a cualquier cosa. Puede ser un puntero a una estructura que guarde, por ejemplo, un puntero a función, entre otras cosas.

6.5.2. Resolción

```c
bool _arreglo_ordenado(int arr[], size_t inicio, size_t fin) {
    if (fin - inicio == 1)
        return arr[fin] >= arr[inicio];

    size_t medio = (inicio + fin) / 2;

    return _arreglo_ordenado(arr, inicio, medio) && _arreglo_ordenado(arr, medio, fin);
}

bool arreglo_ordenado(int arr[], size_t n) {
    return n <= 1 ? true : _arreglo_ordenado(arr, 0, n-1);
}
```

2.

3.

4.

6.6. final-2018c1-2

6.6.1. Consigna

1. Implementar un algoritmo que, dado un Grafo no dirigido, determine si tiene un ciclo, o no. Indicar el orden del algoritmo.

2. Se quiere implementar un TDA Diccionario con las siguientes primitivas: `obtener(x)` devuelve el valor de x en el diccionario; `insertar(x, y)` inserta en el diccionario la clave x con el valor y (entero); `borrar(x)` borra la entrada de x; `add(x,y)` le suma y al contenido de x; `add_all(y)` le suma y a todos los valores. Proponer una implementación donde todas las operaciones sean $\Theta(1)$. Justificar el orden de las operaciones.

3. Implementar un algoritmo que, dado un arreglo de n números enteros cuyos valores van de 0 a K (constante conocida), procese dichos números en tiempo $\Theta(n + K)$, devuelva alguna estructura que permita consultar cuántos valores ingresados están en el intervalo (A, B), en tiempo $\Theta(1)$. Explicar cómo se usaría dicha estructura para poder realizar tales consultas.

6.6.2. Resolción

```python
def dfs(grafo, origen, v, visitados: set):
    visitados.add(v)
    for w in grafo.adyacentes(v):
        if w not in visitados:
            return dfs(grafo, origen, w, visitados)
        if origen == w:
            return True
```

```
    return False

def tiene_ciclos(grafo):
    for v in grafo:
        if dfs(grafo, v, v, set()):
            return True
    return False
```

```
class Dict:
    def __init__(self):
        self.me = {}
        self.engania_pichanga = 0

    def add_all(self, y):
        self.engania_pichanga += y

    def obtener(self, x):
        return self.me[x] += self.engania_pichanga

    def borrar(self, x):
        if x in self.me:
            del(self.me[x])

    def insertar(self, x, y):
        self.me[x] = y

    def add(self, x, y):
        self.me[x] += y
```

3

```
def procesar_entrada(lista):
    d = {}
    i = 1
    for e in lista:
        d[e] = i
        i += 1
    return d

def contar_intervalo(entrada, a, b):
    d = procesar_entrada(entrada)
    return abs(d[a] - d[b])

def main():
    # f = [6, 1, 41, 3, 1, 2, 0]
    # counting_sort(f)
    f = [0, 1, 1, 2, 3, 6, 41]
    d =procesar_entrada(f)

if __name__ == '__main__':
    main()
```