Notas de Algoritmos

Ivan Litteri

Contents

1 Algoritmos de Ordenamiento

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

```

        while j < len(right):
            arr[k] = right[j]
            j += 1
            k += 1
```

1.1.2 Quick Sort

```
# Python program for implementation of Quicksort Sort

# This function takes last element as pivot, places
# the pivot element at its correct position in sorted
# array, and places all smaller (smaller than pivot)
# to left of pivot and all greater elements to right
# of pivot

def partition(arr, low, high):
    i = (low-1)         # index of smaller element
    pivot = arr[high]     # pivot

    for j in range(low, high):

        # If current element is smaller than or
        # equal to pivot
        if arr[j] <= pivot:

            # increment index of smaller element
            i = i+1
            arr[i], arr[j] = arr[j], arr[i]

    arr[i+1], arr[high] = arr[high], arr[i+1]
    return (i+1)

# The main function that implements QuickSort
# arr[] --> Array to be sorted,
# low  --> Starting index,
# high  --> Ending index

# Function to do Quick sort


def quickSort(arr, low, high):
    if len(arr) == 1:
        return arr
    if low < high:

        # pi is partitioning index, arr[p] is now
        # at right place
        pi = partition(arr, low, high)

        # Separately sort elements before
        # partition and after partition
        quickSort(arr, low, pi-1)
        quickSort(arr, pi+1, high)
```

1.1.3 Selection Sort

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

```python

        # Heapify the root.
        heapify(arr, n, largest)

# The main function to sort an array of given size

def heapSort(arr):
    n = len(arr)

    # Build a maxheap.
    for i in range(n//2 - 1, -1, -1):
        heapify(arr, n, i)

    # One by one extract elements
    for i in range(n-1, 0, -1):
        arr[i], arr[0] = arr[0], arr[i]  # swap
        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 $O(log(n))$, armar el heap cuesta $O(n)$. Entonces su complejidad es $O(n \cdot log(n))$.

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).

```python
# Python program for counting sort

# The main function that sort the given string arr[] in
# alphabetical order
def countSort(arr):

    # The output character array that will have sorted arr
    output = [0 for i in range(256)]

    # Create a count array to store count of inidividul
    # characters and initialize count array as 0
    count = [0 for i in range(256)]

    # For storing the resulting answer since the
    # string is immutable
    ans = ["" for _ in arr]

    # Store count of each character
    for i in arr:
        count[ord(i)] += 1

    # Change count[i] so that count[i] now contains actual
    # position of this character in output array
    for i in range(256):
        count[i] += count[i-1]

    # Build the output character array
    for i in range(len(arr)):
        output[count[ord(arr[i])]-1] = arr[i]
        count[ord(arr[i])] -= 1

    # Copy the output array to arr, so that arr now
    # 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
```

```python
  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).

```python
# 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])
```

```python

    # concatenate the result
    k = 0
    for i in range(slot_num):
        for j in range(len(aux[i])):
            arr[k] = aux[i][j]
            k += 1
    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

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

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 $O(1)$ (aprox.).

3.2.3 Algoritmo

```python
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 $O(V)$.
- Ordenar como mucho es $O(E\log(V))$.
- El for es mínimo $O(E)$, en el peor de los casos es $O(E\log(V))$.

- (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 $O(V \cdot E^2)$.

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 $O(n)$ 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 Jruskal 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 vpertice v de una componente fuertemente conexa hacia un vértice w de otra componente fuertemente conexa.