

Notas de Algoritmos

Ivan Litteri

Algoritmos de Ordenamiento

Algoritmos	Complejidad Temporal	Complejidad Espacial	Estable	Comparativo	In-Place
Merge Sort	$O(n \cdot \log(n))$	$O(n)$	True	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

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 precondition 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.

Merge Sort

```
1 # Python program for implementation of MergeSort
2 def mergeSort(arr):
3     if len(arr) > 1:
4
5         # Finding the mid of the array
6         mid = len(arr)//2
7
8         # Dividing the array elements
9         left = arr[:mid]
10
11        # into 2 halves
12        right = arr[mid:]
13
14        # Sorting the first half
15        mergeSort(left)
16
17        # Sorting the second half
18        mergeSort(right)
19
20        i = j = k = 0
21
22        # Copy data to temp arrays left[] and right[]
23        while i < len(left) and j < len(right):
```

```

24         if left[i] < right[j]:
25             arr[k] = left[i]
26             i += 1
27         else:
28             arr[k] = right[j]
29             j += 1
30         k += 1
31
32     # Checking if any element was left
33     while i < len(left):
34         arr[k] = left[i]
35         i += 1
36         k += 1
37
38     while j < len(right):
39         arr[k] = right[j]
40         j += 1
41         k += 1

```

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)

```

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
9
10        # Swap the found minimum element with
11        # the first element
12        arr[i], arr[min_idx] = arr[min_idx], arr[i]

```

Insertion Sort

```

1 def insertionSort(arr):
2     for i in range(1, len(arr)):
3         up = arr[i]
4         j = i - 1
5         while j >= 0 and arr[j] > up:
6             arr[j + 1] = arr[j]
7             j -= 1
8         arr[j + 1] = up
9     return arr

```

Bubble Sort

```

1 # Python program for implementation of Bubble Sort
2
3 def bubbleSort(arr):
4     n = len(arr)
5     # Traverse through all array elements
6     for i in range(n):
7         # Last i elements are already in place
8         for j in range(0, n-i-1):
9
10            # traverse the array from 0 to n-i-1
11            # Swap if the element found is greater
12            # than the next element
13            if arr[j] > arr[j+1]:
14                arr[j], arr[j+1] = arr[j+1], arr[j]

```

Heapsort

```

1 # Python program for implementation of heap Sort
2
3 # To heapify subtree rooted at index i.
4 # n is size of heap
5
6 def leftChild(i):
7     return 2 * i + 1
8
9 def rightChild(i):
10    return 2 * i + 2
11
12 def heapify(arr, n, i):
13     largest = i # Initialize largest as root
14     l = leftChild(i)
15     r = rightChild(i)

```

```

16
17     # See if left child of root exists and is
18     # greater than root
19     if l < n and arr[largest] < arr[l]:
20         largest = l
21
22     # See if right child of root exists and is
23     # greater than root
24     if r < n and arr[largest] < arr[r]:
25         largest = r
26
27     # Change root, if needed
28     if largest != i:
29         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 $O(\log(n))$, armar el heap cuesta $O(n)$. Entonces su complejidad es $O(n \cdot \log(n))$.

Algoritmos de Ordenamiento No Comparativos

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 individual
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
34     for i in range(len(arr)):
35         ans[i] = output[i]
36     return ans

```

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.

```

1 # Python program for implementation of Radix Sort
2 # A function to do counting sort of arr[] according to
3 # the digit represented by exp.
4
5 def countingSort(arr, exp1):
6
7     n = len(arr)
8
9     # The output array elements that will have sorted arr
10    output = [0] * (n)
11
12    # initialize count array as 0
13    count = [0] * (10)
14
15    # Store count of occurrences in count[]
16    for i in range(0, n):
17        index = (arr[i] / exp1)
18        count[int(index % 10)] += 1
19
20    # Change count[i] so that count[i] now contains actual
21    # position of this digit in output array
22    for i in range(1, 10):
23        count[i] += count[i - 1]
24
25    # Build the output array
26    i = n - 1

```

```

27 while i >= 0:
28     index = (arr[i] / exp1)
29     output[count[int(index % 10)] - 1] = arr[i]
30     count[int(index % 10)] -= 1
31     i -= 1
32
33 # Copying the output array to arr[],
34 # so that arr now contains sorted numbers
35 i = 0
36 for i in range(0, len(arr)):
37     arr[i] = output[i]
38
39 # Method to do Radix Sort
40 def radixSort(arr):
41
42     # Find the maximum number to know number of digits
43     max1 = max(arr)
44
45     # Do counting sort for every digit. Note that instead
46     # of passing digit number, exp is passed. exp is 10^i
47     # where i is current digit number
48     exp = 1
49     while max1 / exp > 0:
50         countingSort(arr, exp)
51         exp *= 10
52
53
54     # Driver code
55     arr = [170, 45, 75, 90, 802, 24, 2, 66]
56
57     # Function Call
58     radixSort(arr)
59
60     for i in range(len(arr)):
61         print(arr[i])

```

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).

```

1 # Python3 program to sort an array
2 # using bucket sort
3 def insertionSort(arr):
4     for i in range(1, len(arr)):
5         up = arr[i]
6         j = i - 1
7         while j >= 0 and arr[j] > up:
8             arr[j + 1] = arr[j]
9             j -= 1
10        arr[j + 1] = up

```

```

11     return arr
12
13 def bucketSort(arr):
14     aux = []
15     slot_num = 10 # 10 means 10 slots, each
16                   # slot's size is 0.1
17     for i in range(slot_num):
18         aux.append([])
19
20     # Put array elements in different buckets
21     for j in arr:
22         index_b = int(slot_num * j)
23         aux[index_b].append(j)
24
25     # Sort individual buckets
26     for i in range(slot_num):
27         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

```

Propiedades de los Algoritmos de Ordenamiento

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.

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).

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 hijos 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 resuelve el problema dividiéndolo en 5 subproblemas 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 recursiva y combinando las soluciones en tiempo cuadrático de n .
 - c El algoritmo C resuelve todos los problemas de tamaño n eligiendo un subproblema de tamaño $n - 1$ en tiempo $O(n)$ luego resolviendo recursivamente ese subproblema.

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 para 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 levemente 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 elevan 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 utilizar 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 necesario 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 conexas hacia un vértice w de otra componente fuertemente conexas.