Notas de Algoritmos

Ivan Litteri

Algoritmos de Ordenamiento

Algoritmos	Complejidad Temporal	Complejidad Espacial	Estable	Comparativo	In-Place
Merge Sort	$O(n \cdot log(n))$	$O(n)$	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	
Bucket Sort	$O(n^2)$	$O(n)$	True	False	
Counting Sort	$O(n + k)$	$O(k)$	True	False	True
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 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.

Merge Sort

```
# 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
```

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
```

```python
        quickSort(arr, low, pi-1)
        quickSort(arr, pi+1, high)
```

Selection Sort

```python
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]
```

Insertion Sort

```python
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
```

Bubble Sort

```python
# 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]
```

Heapsort

```python
# 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)
```

```python

    # 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

        # 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 $O(n \cdot log(n))$.

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).

```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
```

```python
    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
```

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
# 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
```

```python
  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])
```

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
```

```python
    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])

    # 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
```

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).