Tarea I

Darío Vargas Montoya
Tecnológico de Costa Rica
Análisis de algoritmos
dariovm13@gmail.com

***Abstract*—En este documento se analizan el tiempo de ejecución de tres algoritmos diferentes programados en Python.**

***Index Terms*—python, algoritmos, ordenamiento, medición, empírica**

I. Introducción

En este documento se analizan el tiempo de ejecución de 3 algoritmos diferentes mediante la medición empírica.

II. Algoritmos utilizados

A. Quicksort

Quicksort es uno de los algoritmos de ordenamiento que históricamente ha sido conocido como un de los más rápidos conocidos en la práctica. Quicksort es un algoritmo recursivo de tipo ”divide y vencerás” y es fácil de implementar [1].

```
def quickSort(numbersList):
    high = []
    low = []
    pivotList = []

    if(len(numbersList) <= 1):
        return numbersList
    else:
        pivot = numbersList[0]
        for i in numbersList:
            if(i < pivot):
                low += [i]
            elif(i > pivot):
                high += [i]
            else:
                pivotList += [i]
        high = quickSort(high)
        low = quickSort(low)
    return low + pivotList + high
```

Listing 1. Quicksort

El anterior código se muestra el algoritmo quicksort adaptado de Nekto, 2017 [2]. Como se muestra en éste algoritmo se utilizan 3 listas las cuales contendrán los números mayores al pivote, los menores al pivote y por último una lista de los pivotes. Al iniciar el algoritmo se verifica si el tamaño de la lista de números es menor o igual a uno, de lo contrario, se selecciona el primer elemento de la lista como pivote y se inserta en la lista correspondiente los números mayores y menores al pivote para luego ser ordenados recursivamente. Por último, se combinan la listas de menores, la de pivotes y la de mayores para dar la lista ordenada como resultado.

B. Merge Sort

Merge Sort es uno de los algoritmos de ordenamiento más populares basados en el principio de divide y vencerás. Funciona de manera similar al Quicksort, divide la lista en dos partes y se llama a sí mismo para las dos mitades y luego fusiona las dos mitades ordenadas.

```
def mergesort(numbersList):
    arrayLenght = len(numbersList)
    if(arrayLenght <= 1):
        return numbersList
    else:
        middleIndex = arrayLenght//2
        leftList = mergesort(numbersList[:
    middleIndex])
        rightList = mergesort(numbersList[
    middleIndex:])
        return merge(leftList, rightList)

def merge(left, right):
    sortedArray = []
    while (len(left) > 0 and len(right) != 0):
        if(left[0] > right[0]):
            sortedArray += [right[0]]
            right = right[1:]
        else:
            sortedArray +=[left[0]]
            left = left[1:]

    while(len(left) > 0):
        sortedArray += [left[0]]
        left = left[1:]

    while(len(right) > 0):
        sortedArray += [right[0]]
        right = right[1:]

    return sortedArray
```

Listing 2. Merge Sort

El anterior algoritmo adaptado de Tutorialspoint [3] se muestra la clásica implementación del merge sort donde se parte el arreglo en dos parte para posteriormente llamar recursivamente las dos mitades para luego ser fusionadas.

C. BubbleSort

Bubble sort es uno de los algoritmos más fáciles de implementar, consiste en comparar pares de valores de arreglos, empezando en las posiciones 0 y 1. Si los valores comparados están en orden inverso, se intercambian. Bubble Sort funciona bien con arreglos pequeños, pero se vuelve demasiado ineficiente para arreglos más grandes [4].

```
def bubblesort(numbersList):
    arrayLenght = len(numbersList)
    for i in range(arrayLenght):
        swapped = True

        for j in range(0, arrayLenght -i - 1):
            if(numbersList[j] > numbersList[j + 1]):
                (numbersList[j],numbersList[j + 1])
    = (numbersList[j + 1], numbersList[j])
                swapped = False
        if swapped:
            break
    return numbersList
```

Listing 3. Bubble Sort

Como se observa en el Listing 3, se muestra el algoritmo de ordenamiento de burbuja adaptado de Programiz [5]. En ésta implementación primero recorre todo el arreglo, luego recorre el arreglo iniciando desde cero hasta el tamaño del arreglo menos la posición del elemento actual menos uno. Si el elemento actual es mayor al siguiente elemento se intercambian los elementos y finaliza hasta que no se hayan hecho intercambios en el arreglo.

III. Metodología

Para la presente documento se realizaron 10 ejecuciones de los 3 algoritmos de ordenamiento en tamaños de arreglo desde 16 hasta 32768, cabe recalcar que se utilizó para los 3 algoritmos el mismo arreglo desordenado y para la generación de los números aleatorios se utilizaban número desde 0 hasta el tamaño del arreglo.

IV. Análisis del tiempo de ejecución

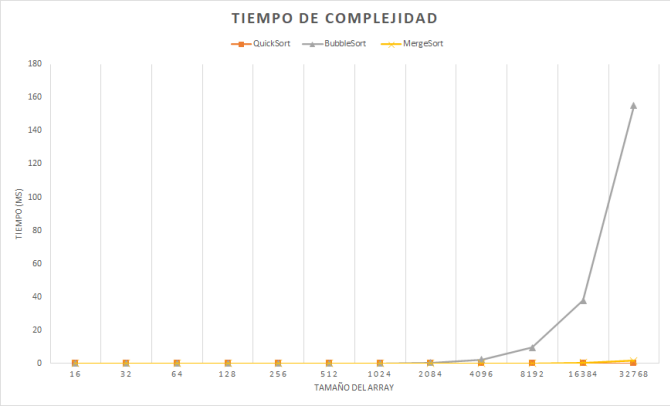

Fig. 1. Tiempo de ejecución de los algoritmos de ordenamiento.

En la figura 1 se muestra una gráfica con la complejidad de tiempo de los diferentes algoritmos de ordenamiento según el tamaño del arreglo. Como se observa, en los tamaños de arreglo que comprenden desde 16 hasta 2084 no existe una gran diferencia en la complejidad del tiempo, sin embargo en los arreglos de mayor tamaño, el ordenamiento burbuja empieza a tener tiempos de ejecución cada vez más ineficientes. Por otro lado, merge sort y bubblesort tienen tiempos de ejecución muy similares.

TABLE I
Tabla de datos

Tamaño	*Quicksort*	*Bubble sort*	*Merge sort*
16	2.444E-05	0.00059525	0.00180481
32	8.214E-05	0.00168903	0.00124266
64	0.00016011	0.00122472	0.00085244
128	0.00026646	0.00334474	0.00663579
256	0.00050311	0.01070566	0.00520997
512	0.00095783	0.03567177	0.00539754
1024	0.00234623	0.13982986	0.00707349
2084	0.00583197	0.62105387	0.02173501
4096	0.01168556	2.39952782	0.04764446
8192	0.02219983	9.48746835	0.13941344
16384	0.04253919	37.9237406	0.45777727
32768	0.09148755	155.217263	1.74680176

En la tabla 1 se muestran los datos del promedio de las ejecuciones realizadas.Como se puede observar, entre los tamaños de arreglos más pequeños se bubble sort y merge sort son muy parecidos en el tiempo de complejidad, sin embargo conforme aumenta el tamaño del arreglo, bubble sort tiene tiempos de ejecución cada vez más largos, ésto puede ser debido a que el método que utiliza para ordenar el arreglo sea mucho más ineficiente. Mientras por otro lado merge sort y quicksort que son algoritmos que utilizan el método de ”divide y venceras” son más rápidos.

References

[1] E. F. Figueroa, “Una máquina de turing para el algoritmo “quicksort”.”
[2] Nekto, “Quick sort recursion,” Website, Feb. 2017, acessed: 24-2-2021. [Online]. Available: https://stackoverflow.com/questions/42418398/quick-sort-recursion
[3] Tutorialspoint, “Data structures - merge sort algorithm,” acessed: 24-2-2021. [Online]. Available: http://bit.ly/2P9lD4i
[4] K. R. Irvine, *Lenguaje Ensamblador: para Computadoras Basadas en Intel, 5/ed.*, 5th ed., L. M. C. Castillo, Ed. Pearson Education, 2008.
[5] Programiz, “Bubble sort algorithm,” acessed: 24-2-2021. [Online]. Available: http://bit.ly/3kjna3g