Facultad de UNER Ingeniería

Trabajo N°1 Algoritmos y Estructura de Datos

Alumnos:
Anderson, Mateo.
Costa, Luciana.
Insaurralde, Rosario.

Profesor de comision: Juan Rizzato

Fecha límite 2 de mayo de 2025

PROBLEMA 1:

En este caso se presentan tres algoritmos de ordenamiento:

- Burbuja.
- Quicksort.
- Radix Sort.

Con los mismos necesitábamos evaluar su funcionamiento utilizando listas de números aleatorios de cinco dígitos, generando al menos 500 números. Debíamos comparar el tiempo de ejecución de cada algoritmo con listas de tamaño entre 1 y 1000 elementos y realizar una comparación con el ordenamiento sorted de python.

Ordenamiento burbuja:

Este tipo de ordenamiento (también denominado Bubble sort) nos ayuda a comparar elementos en una lista y los intercambia si están en un orden incorrecto. Este proceso se repite hasta que la lista esté completamente ordenada.

Análisis de complejidad

- En el peor caso $O(n^2)$ donde n es el número de elementos. Ocurre cuando la lista está ordenada en orden inverso.
- En el mejor de los casos O(n) cuando la lista ya está ordenada

```python
def burbuja(lista):
    for i in range(len(lista) - 1): #no definimos una variable con len(lista) para ocupar menos memoria
        for j in range(0, len(lista) - i - 1): #inicio, fin, paso (recorro desde el final hasta el principio, disminuyendo de a 1)
            if lista[j]>lista[j+1]: #si el numero en la posicion j es mayor al siguiente, sucede:
                lista[j], lista[j+1] = lista[j+1], lista[j] #intercambio las posiciones
```

Ordenamiento Quicksort

En este tipo de ordenamiento se elige un "pivote" y reorganiza la lista para que todos los elementos menores que el pivote estén a la izquierda y los mayores a la derecha.

Análisis de complejidad:

- En el peor de los casos $O(n^2)$. Ocurre cuando el pivote es el menor o el mayor número.
- En el mejor caso O(n log n.). Ocurre cuando el pivote divide la lista de forma equitativa.

```python
def quicksort(lista):
    if len(lista) <= 1: #si la lista tiene un solo elemento o esta vacia, devuelve la lista porque ya esta ordenada
        return lista
    else:
        pivote = lista[0] #elegimos el elemnto 0 porque si
        menores = [x for x in lista[1:] if x <= pivote] #Se está construyendo una nueva lista llamada menores que contiene todos los elementos de la
        mayores = [x for x in lista [1:] if x > pivote]
        return quicksort(menores) + [pivote] + quicksort(mayores) #devuelvo la lista completa uniendo todas las partes
```

Ordenamiento por Radix

Este tipo de ordenamiento (también llamado Radix Sort) es un algoritmo no comparativo, que ordena los números según los dígitos individuales comenzando desde el menos significativo al más significativo.

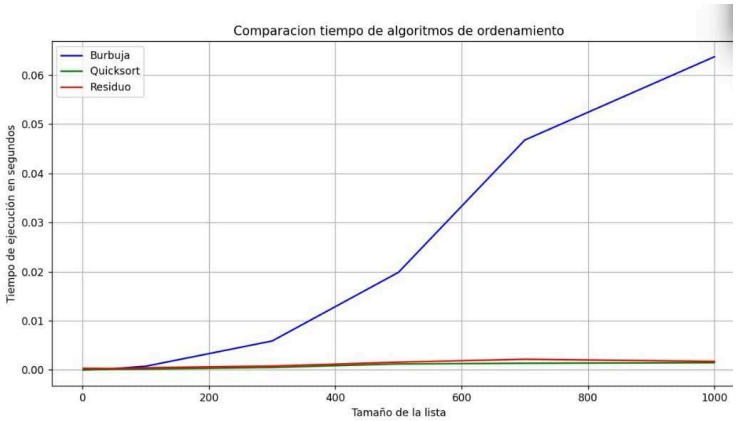
Análisis de complejidad:

- O(n.k) siendo n el el número de elementos y k el número de dígitos.
- Método muy eficiente en listas grandes

```python
def residuo(lista):
    maximo = max(lista) #busco el maximo para saber cuantos digitos recorremos
    exponente = 1 #empieza en 1
    while maximo // exponente > 0:
        conteo = [0] * 10 #conteo de 10 elementos
        salida = [0] * len(lista) #esta lista guarda elementos de la lista original almacenados

        for i in range(len(lista)):
            indice = lista[i] // exponente #divido el numero en la posicion i por el exponente
            conteo[indice % 10] += 1 #digito en la posicion y luego incremento el contador

        for i in range(1, 10):
            conteo[i] += conteo[i - 1] #acumulo valores en la lista de conteo

        i = len(lista) - 1 #recorro la lista desde el final hacia el inicio
        while i >= 0:
            indice = lista[i] // exponente #digito en la posicion i
            salida[conteo[indice % 10] - 1] = lista[i] #coloco el numero en la posicion correcta
            conteo[indice % 10] -= 1 #contador para el digito actual
            i -= 1 #muevo al siguiente elemento en la lista original

        lista = salida
        exponente *= 10 #paso a la siguiente posicion de digito

    return salida
```

Medición de Tiempos de Ejecución

Para medir los tiempos de ejecución, generamos listas de números aleatorios con un tamaño entre 1 y 1000, y ejecutamos cada uno de los algoritmos. Usamos la librería time de Python para medir el tiempo que tarda en ejecutarse cada algoritmo.



En el gráfico podemos ver los tiempos de ejecución de los tres algoritmos. Y como puede observarse, Bubble Sort es más lento a medida que aumenta el tamaño de la lista, mientras que Quicksort y Radix Sort muestran un mejor rendimiento.

Comparación con sorted

La función sorted() de Python utiliza Timsort, que es una combinación de merge sort y insertion sort. Su complejidad en el peor caso es O (n log n), lo que lo hace más eficiente que bubble sort en casi todos los casos. Además, es muy eficiente con listas parcialmente ordenadas debido a su uso de insertion sort.

Conclusión

A través de este análisis, se puede concluir lo siguiente:

- Radix Sort es más eficiente para grandes listas con números de longitud fija, ya que su tiempo de ejecución no depende tanto del tamaño de la lista, sino de los dígitos de los números.
- Quicksort es el algoritmo más eficiente en el promedio de casos, con una complejidad de O (n log n).
- Bubble Sort es el menos eficiente, especialmente para listas grandes, debido a su complejidad cuadrática $O(n^2)$.
- Comparando con sorted() de Python, podemos ver que Timsort, que utiliza una combinación de merge sort e insertion sort, es muy eficiente, especialmente para listas parcialmente ordenadas.

PROBLEMA 2

En esta parte del trabajo necesitábamos implementar una estructura conocida como lista doblemente enlazada. La misma se trata de una secuencia de nodos, donde cada nodo guarda un dato y tiene dos punteros: uno que apunta al siguiente nodo y otro que apunta al anterior. Esta estructura nos permite recorrer la lista en ambas direcciones, hacer inserciones y eliminaciones tanto al principio como al final de forma eficiente.

Para resolver esta consigna, implementamos las siguientes clases:

- Clase nodo:
  - Va a representar cada elemento de la lista.
  - Tiene tres atributos:
    - Dato: el valor que guarda el nodo
    - Siguiente: hace referencia al nodo que viene después.
    - Anterior: hace referencia al nodo que viene antes.

```
class nodo: #permite recorrer la lista en ambos sentidos
    def __init__(self, dato):
        self.dato = dato #almacena el dato
        self.anterior = None #puntero al numero anterior
        self.siguiente = None #puntero al numero siguiente
```

- Clase ListaDobleEnlazada:
  - Donde se gestiona la lista y sus operaciones.
  - Utiliza los siguientes metodos:
    - agregar_al_inicio(dato): agrega un nodo al principio de la lista.
    - agregar_al_final(dato): agrega un nodo al final al final de la lista.
    - Insetar (): inserta un nodo en una posición específica.
    - Extraer (): elimina el nodo en la posición dada y va a devolver su dato.
    - esta_vacia(): devuelve True si la lista está vacía.
    - __len__(): devuelve la cantidad de elementos en la lista.
    - Copiar(): nos va a dar una copia exacta de la lista original
    - Invertir(): invierte el orden de los nodos en la lista.
    - Concatenar(): agrega todos los elementos de otra lista al final de la lista actual.
    - __add__(): devuelve una lista combinada.
    - __iter__(): devuelve un generador que recorre los nodos uno a uno desde el inicio hasta el fin.

```
class ListaDobleEnlazada:
    def __init__(self):
        self.cabeza = None #aun no hay primer elemento
        self.cola = None #tampoco hay ultimo elemento aun
        self.tamanio = 0 #inicia en cero
```

PROBLEMA 3

Esta Parte del trabajo consistió en simular un juego de cartas denominado “Guerra”. Para esto implementamos una clase llamada mazo utilizando la estructura de la ListaDobleEnlazada Del ejercicio anterior. Con lo cual hicimos lo siguiente:

- poner_carta_arriba(self, carta): Se utiliza la función agregar_al_inicio(self, dato) de listas doblemente enlazadas con una carta ingresada a modo de dato.
- sacar_carta_arriba(self, mostrar=False): Se utiliza la función extraer(self, posicion), con la posicion fijada en cero, es decir la carta inicial. El parámetro “mostrar” debe ser “False” inicialmente, y se mostrará la carta extraída en caso de que se indique como “True” durante el juego.
- poner_carta_abajo(self, carta): Se utiliza la función agregar_al_final(self, dato) de listas doblemente enlazadas con una carta ingresada a modo de dato.
- __len__(self): se sobrecarga el método para poder implementarse sobre el mazo

Además, se creó la clase DequeEmptyError. La misma se despliega en caso de que se intente sacar una carta de un mazo sin elementos.

```
class DequeEmptyError(Exception):
    """El mazo está vacío."""
    pass  # Define una excepción personalizada para cuando el mazo está vacío
```