Facultad de Ingeniería
UNER

Algoritmos y Estructuras de Datos

Informe del Trabajo Práctico N°1

Algoritmos de Ordenamiento e Implementación de Tipos Abstractos de Datos

Olivera Julieta, Sarmiento Augusto

Licenciatura en Bioinformática

Facultad de Ingeniería

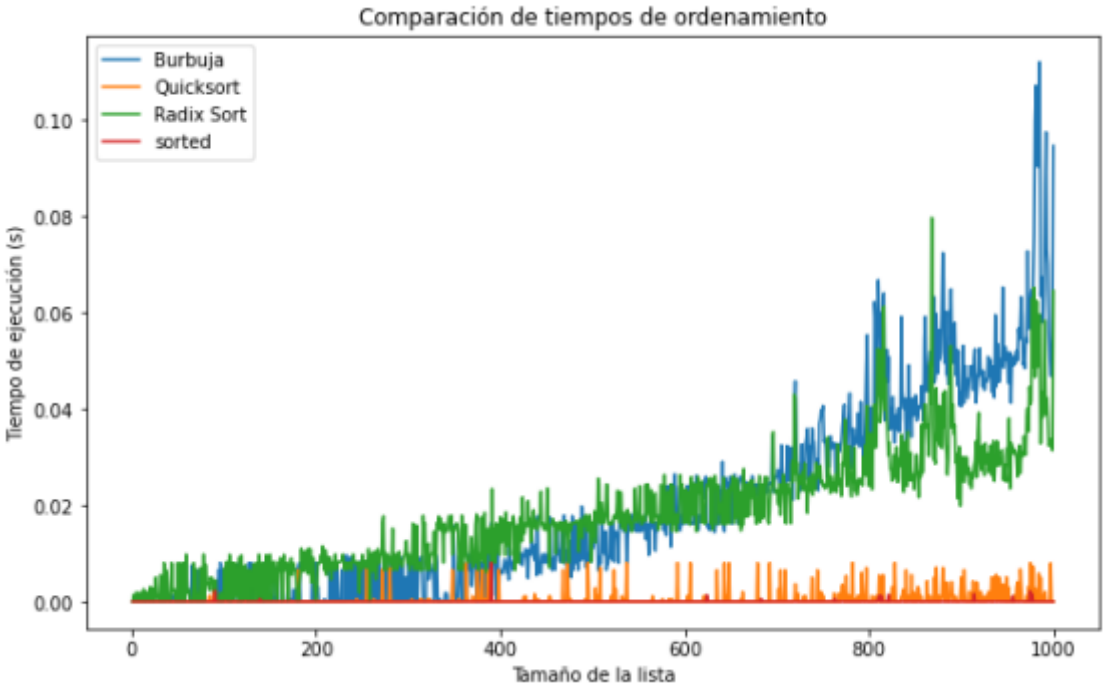
Universidad Nacional de Entre Ríos

Actividad 1

Para analizar los tiempos de ejecución de cada lista, se realizó un programa que compara las distintas 1000 listas utilizando los módulos time, random y matplotlib. Se intentó ingresar el código en el subdirector tests del directorio Actividad_1, pero nos encontramos con muchos problemas a la hora de utilizar la librería matplotlib, no se lograba instalar mediante el uso de pip ya que este último comando tampoco funcionaba, entonces se optó por correr el siguiente programa en un entorno diferente, definiendo previamente las funciones que se encuentran en el repositorio para los distintos algoritmos de ordenamiento (Spyder):

```
def medir_tiempos():
    tamaños = range(1, 1001)
    tiempos_burbuja = []
    tiempos_quicksort = []
    tiempos_radix = []
    tiempos_sorted = []

    for tamaño in tamaños:
        lista = [random.randint(10000, 99999) for _ in range(tamaño)]

        inicio = time.time()
        bubble_sort(lista.copy())
        tiempos_burbuja.append(time.time() - inicio)

        inicio = time.time()
        quick_sort(lista.copy())
        tiempos_quicksort.append(time.time() - inicio)

        inicio = time.time()
        radix_sort(lista.copy())
        tiempos_radix.append(time.time() - inicio)

        inicio = time.time()
        sorted(lista.copy())
        tiempos_sorted.append(time.time() - inicio)

    plt.figure(figsize=(10, 6))
    plt.plot(tamaños, tiempos_burbuja, label='Burbuja')
    plt.plot(tamaños, tiempos_quicksort, label='Quicksort')
    plt.plot(tamaños, tiempos_radix, label='Radix Sort')
    plt.plot(tamaños, tiempos_sorted, label='sorted')
    plt.xlabel('Tamaño de la lista')
    plt.ylabel('Tiempo de ejecución (s)')
    plt.title('Comparación de tiempos de ordenamiento')
    plt.legend()
    plt.show()

medir_tiempos()
```

Con la función se obtuvo la siguiente gráfica:



El algoritmo bubble sort tiene un orden de complejidad de O(n²). por cada pasada, el elemento de mayor tamaño tiene que llegar a su posición determinada “burbujeando” entre los demás.
El algoritmo quicksort tiene un orden de complejidad proporcional a O(n*log n), lo que se genera gracias a la subdivisión de listas. En el peor caso (si la lista ya está ordenada o semi-ordenada), el algoritmo puede presentar una complejidad de O(n²) debido al desbalance que quedaría en las listas.
En el algoritmo radix sort se tiene un orden de complejidad de O(n * d), siendo “d” el número de dígitos analizados. Este orden de complejidad es debido a que cada cada elemento debe procesarse en cada pasada.

El algoritmo sorted de python divide una lista en sublistas de tamaño óptimo y las ordena con un algoritmo de inserción (en algunas implementaciones de python se llegó a utilizar el algoritmo quicksort en este paso), para luego fusionarlas, lo que logra un orden de complejidad de O(n * log n).

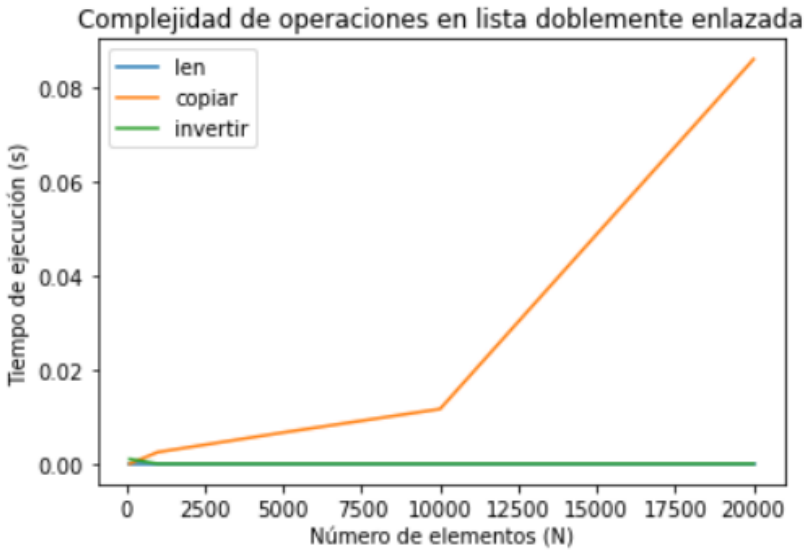
Actividad 2

Para la actividad se presentó el mismo problema para las gráficas, por lo tanto se decidió realizar las gráficas en otro IDE, las gráficas se obtuvieron con el siguiente código y utilizando las librerías time y matplotlib.pyplot, como así también nuestras propias listas doblemente enlazadas:

```
def medir_tiempo(func, *args, **kwargs):
    tiempo_inicio = time.time()
    resultado = func(*args, **kwargs)
    tiempo_fin = time.time()
    return resultado, tiempo_fin - tiempo_inicio
```

```
tamaños_de_listas = [100, 1000, 5000, 10000, 20000]
lista_tiempos_len = []
lista_tiempos_copiar = []
lista_tiempos_invertir = []

for i in tamaños_de_listas:
    dll = ListaDoblementeEnlazada()

    for j in range(i):
        dll.agregar_al_final(j)

    _, time_len = medir_tiempo(len, dll)
    copiar, tiempo_copiar = medir_tiempo(dll.copiar)
    dll.invertir()
    _, tiempo_invertir = medir_tiempo(len, dll)

    lista_tiempos_len.append(time_len)
    lista_tiempos_copiar.append(tiempo_copiar)
    lista_tiempos_invertir.append(tiempo_invertir)


plt.plot(tamaños_de_listas, lista_tiempos_len, label='len')
plt.plot(tamaños_de_listas, lista_tiempos_copiar, label='copiar')
plt.plot(tamaños_de_listas, lista_tiempos_invertir, label='invertir')
plt.xlabel('Número de elementos (N)')
plt.ylabel('Tiempo de ejecución (s)')
plt.title('Complejidad de operaciones en lista doblemente enlazada')
plt.legend()
plt.show()
```

La gráfica obtenida fue la siguiente:



Se puede observar que:

- Len tiene un orden de complejidad O(1), lo que es lógico ya que solo debe revisarse el atributo self.tamanio de las listas doblemente enlazadas
- Copiar tiene un comportamiento tendencialmente lineal, como fue requerido por la cátedra
- Invertir tiene una complejidad de O(1) debido a que el código propuesto por el grupo utiliza puntero, lo que evita un recorrido constante de las listas.

Las listas doblemente enlazadas se basan en el uso de Nodos, que también tuvieron que definirse previamente como una clase. Los Nodos tienen una carga útil y dos enlaces, uno a un nodo anterior y otro a un nodo siguiente, estos últimos dos campos pueden ser None.
Una lista doblemente enlazada estará originalmente vacía y se le pueden agregar elementos (en forma de objetos Nodo) con distintos métodos.
Las listas doblemente enlazadas tienen, a su vez, distintos atributos que resultan de importancia:

- cabeza: es el primer nodo de la lista, su anterior es None.
- cola: es el último nodo de la lista, su siguiente es None.
- tamanio: es igual a la cantidad de elementos de la lista (originalmente es 0).

Breve explicación de cada función:

- mostrar_lista(self): se realiza un print de cada elemento de la lista, printeando luego un espacio y pasando al siguiente. Si bien no se solicitó por la cátedra, resultó útil para corregir errores.
- agregar_al_final(self): se agrega un nodo al final de la lista, actualizando los atributos cola y tamanio de la misma. Si la lista está vacía, el nuevo nodo será también la cabeza de la lista.
- agregar_al_inicio(self): se agrega un nodo al inicio de la lista, actualizando los atributos cabeza y tamanio de la misma. Si la lista está vacía, el nuevo nodo será también la cola de la lista.
- está_vacia(self): se chequea el atributo cabeza de la lista, si el mismo no existe, indica que la lista está vacía.
- __len__(self): chequea el atributo tamanio y devuelve su valor.
- insertar(valor,posición): si la posición no se indica, simplemente se utiliza el método agregar_al_final() para cumplir con lo requerido por la cátedra. Si la posición está fuera del rango de la lista, se levanta un mensaje de error. Si la posición es 0, se utiliza el método agregar_al_inicio(). En cualquier otra situación, se crea un puntero para encontrar la posición deseada y se actualizan los valores de los atributos de los nodos de esa posición al agregar el nuevo. En cualquier situación se actualiza el atributo tamanio de la lista.
- extraer(posicion): Se analiza primero si la lista está vacía, avisando en caso de que lo esté. También levanta un error si está fuera del rango. Se busca la posición indicada con el puntero y se elimina el nodo en esa posición, devolviendo la carga útil del mismo como valor. Si no se indica una posición, se extrae el nodo de la cola.
- copiar(self): se crea una nueva lista vacía y un puntero sobre la lista pasada como argumento. Se copia el valor del nodo señalado por el puntero y se va agregando a la nueva lista, actualizando todos los atributos necesarios.
- invertir(self): este método hace uso de un puntero y va invirtiendo los nodos de la lista de uno por vez. Al finalizar, deben actualizarse los atributos cabeza y cola

puesto que estos, si bien se invirtieron, nunca cambiaron su contexto, esto es necesario para el buen funcionamiento de la lista.

- concatenar(self, otra_lista): se le suma a self cualquier otra lista doblemente enlazada. Se debe crear una copia de otra_lista para usar como lista auxiliar porque si no, las listas quedan enlazadas, y eso no es deseable. Se actualizan también los atributos de la primera lista.
- __add__(self, otra_lista): la diferencia con concatenar es que con este método sí se obtiene una nueva lista, haciendo uso de los métodos copiar() y concatenar().

Actividad 3:

Para la resolución de esta actividad como pilares se consideraron los códigos brindados por la cátedra y posteriormente se implementó la Clase Mazo, la cual representa una baraja de cartas mediante una lista doblemente enlazada, combinando palos con valores. Luego para facilitar la correcta ejecución de las acciones planteadas, se definió a este mazo como una lista en lugar de nodos interconectados.
Como resultado, el código informará si el ganador fue el Jugador 1, Jugador 2 o si hubo un empate.