Open in Colab

Clase 19: Caso de Estudio III

Búsqueda y ordenamiento

En este caso de estudio se estudiarán algoritmos de búsqueda y ordenamiento utilizando estructuras mutables (listas de Python). En particular:

- Algoritmos de búsqueda:
  - Búsqueda Secuencial
  - Búsqueda Binaria
- Algoritmos de ordenamiento:
  - Mergesort
  - Bubblesort
  - Quicksort

+ Código + Texto

Búsqueda en una lista

Problema

- Lista (de Python) de números enteros
- Se desea buscar el índice en la lista donde se encuentra un valor en particular
  - Si hay varias ocurrencias de valor, se debe devolver el índice de la primera
  - Si el valor no está en el arreglo, se devuelve None (valor inválido para un índice)

Búsqueda Secuencial

La búsqueda secuencial es el algoritmo usado en clases para buscar sobre una lista. Dado una lista L y un valor x a buscar, seguimos el siguiente **algoritmo**:

- partiendo del primer elemento (indice i=0) comparamos el elemento en esa posición con el valor buscado x.
- si es igual, ¡lo encontramos! Retornamos el índice i.
- si no, incrementamos i en 1, esto es, seguimos buscando con el siguiente elemento.
- al final, si i sobrepasa el índice del mayor elemento de la lista, podemos decir que el valor buscado x **no está**.

Observación: Este algoritmo sirve para tanto para una lista ordenada, como una lista desordenada. Por ahora supondremos que la lista está *desordenada*.

```
# Las listas utilizadas en esta implementacion son listas de Python
# Los valores almacenados en las listas son numeros enteros

# Busqueda Secuencial

# busquedaSecuencial: list(int) int -> int o None
# devuelve el indice de la lista donde se encuentra valor haciendo una
# busqueda secuencial, o devuelve None si no esta
# ejemplo: busquedaSecuencial([258, 45, 99, 14, 102, 75, 128], 14) devuelve 3
def busquedaSecuencial(lista, valor):
    for indice in range(len(lista)):
        if lista[indice] == valor:
            # se encontro el valor
            return indice
    # valor no fue encontrado
    return None

# Test
lista = [258, 45, 99, 14, 102, 75, 128]
assert busquedaSecuencial(lista, 14) == 3
assert busquedaSecuencial(lista, 20) == None
```

```
lista = [258, 45, 99, 14, 102, 75, 128]
busquedaSecuencial(lista, 102)
```

¿Cuántas comparaciones debe realizar para buscar una lista con n valores?

- Mínimo = 1 (mejor caso)
- Máximo = n (peor caso)
- Promedio? Si uno busca el primer elemento de la lista, entonces haremos 1 comparación, si buscamos el segundo, haremos 2 comparaciones, y así hasta el último, donde hacemos n comparaciones. Si buscaremos n elementos en la lista y cada elemento que buscamos está en una posición al azar en la lista, entonces intuitivamente uno de ellos será el 1ero de la lista, otro será el segundo, etc. Entonces, si sumamos las comparaciones del 1ero, del 2do, etc. y lo dividimos por n calcularemos el número promedio de comparaciones:

  Promedio = $\frac{1+2+\ldots+n}{n} = \frac{1}{n}\sum_{i=1}^{n} i = \frac{n+1}{2}$

  - En promedio 500 comparaciones para buscar en $n = 1000$ valores
  - En promedio 500.000.000 comparaciones para buscar en $n = 1.000.000.000$ valores

Listas ordenadas: Si supiéramos que vamos a buscar siempre sobre una lista ordenada, cómo podríamos mejorar el algoritmo de búsqueda lineal para que se demore un poco menos en encontrarlo? Piense en (a) cómo modificaría el algoritmo, y (b) cuánto mejoraría.

Búsqueda Binaria

Si la lista sobre la cual buscamos está **ordenada** podemos mostrar un algoritmo mucho más eficiente (rápido), uno que toma menos comparaciones!

El algoritmo se denomina **búqueda binaria** y aquí daremos la versión recursiva:

- Si lista está vacía, retornar None (caso base)
- Sino:
    - Comparar x con elemento en la mitad de la lista
    - Si son iguales, devolver índice de mitad
    - Si x es menor, buscar recursivamente en primera mitad de la lista
    - Si x es mayor, buscar recursivamente en segunda mitad de la lista

EXPLICACIÓN: Puede ver una explicación detallada del algoritmo de búsqueda binaria en el siguiente video: ¿Cómo funciona la búsqueda binaria?

Función que implementa Búsqueda Binaria

```
# Busqueda Binaria

# busquedaBinariaRec: list(int) int int int -> int o None
# devuelve el indice de la lista donde se encuentra valor haciendo una
# busqueda binaria en el rango [ip,iu], o devuelve None si no esta
# Requiere que la lista este ordenada ascendentemente
# ejemplo: busquedaBinariaRec([14, 45, 75, 99, 102, 128, 258], 45, 0, 6) devuelve 1
def busquedaBinariaRec(listaOrdenada, valor, ip, iu):
    # Caso base: valor no fue encontrado
    if ip > iu:
        return None
    im = (ip + iu) // 2 # indice del medio
    # Caso base: valor esta al medio de la lista
    if listaOrdenada[im] == valor:
        return im
    # Llamados recursivos
    elif valor < listaOrdenada[im]:
        return busquedaBinariaRec(listaOrdenada, valor, ip, im - 1)
    else: # valor > listaOrdenada[im]
        return busquedaBinariaRec(listaOrdenada, valor, im + 1, iu)
```

```
# busquedaBinaria: list(int) int -> int o None
# Hace la primera invocacion a busquedaBinariaRec
def busquedaBinaria(listaOrdenada, valor):
    return busquedaBinariaRec(listaOrdenada, valor, 0, len(listaOrdenada) - 1)

# Test
lista = [14, 45, 75, 99, 102, 128, 258]
assert busquedaBinaria(lista, 45) == 1
assert busquedaBinaria(lista, 20) == None
```

```
listaOrdenada = [14, 45, 75, 99, 102, 128, 258]
busquedaBinaria(lista, 102)
```

Tiempo (rapidez) del algoritmo

La Búsqueda binaria es eficiente

- Si lista tiene n valores, búsqueda binaria realiza aproximadamente $\log n$ comparaciones
  - Si $n = 1.000.000.000$, búsqueda binaria realiza aproximadamente 30 comparaciones (*)
  - Mucho más rápido que búsqueda secuencial
- Si la única operación permitida para buscar es la comparación entre objetos, búsqueda binaria es óptima (lo mejor posible)

Observación matemática: ¿Por qué podemos decir que siempre toma $\log n$ comparaciones? En cada comparación se descarta *la mitad* del arreglo. Un buen desafío es demostrar que, si tiene un arreglo de n elementos, al descartar la mitad en cada paso, el número máximo de pasos siempre es $\log n$. Inténte demostrarlo!

Segunda Parte:

Ordenamiento (ordenar una lista)

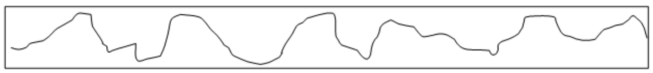
El Problema

Tenemos el siguiente problema. Queremos ordenar una lista de valores (por ejemplo, enteros):

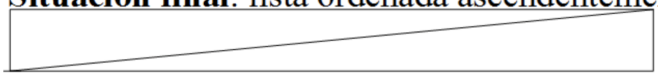
- los valores en la lista están desordenados
- queremos terminar con una lista con los valores ordenados.

Gráficamente:

Situación inicial: lista desordenada



Situación final: lista ordenada ascendentemente



Nota: Si bien mencionamos listas de enteros, puede ser listas de cualquier tipo de valor, basta que los elementos sean comparables entre sí. Por ejemplo, podrían ser *strings*, *floats*, etc. Por simplicidad, pensaremos en números enteros.

Los Algoritmos

Explicaremos 3 algoritmos distintos:

- Mergesort (o algoritmo de mezcla)
- Bubblesort
- Quicksort

Mergesort

Para entender el problema de ordenamiento y el algoritmo de Ordenamiento por Mezcla (*Mergesort*), debe ver el siguiente video.

El video plantea un desafío principal. Intente primero resolverlo por su cuenta (es OPCIONAL!) antes de ir a mirar la solución de más abajo.

Video MergeSort

Si logró hacer el desafío principal, intente ahora hacer el siguiente desafíos secundario OPCIONAL. No es difícil pero puede tomar harto tiempo pues requiere manejarse con archivos.

- Implemente una versión de mergesort que ordene palabras que lea desde un archivo de texto que contiene una palabra por línea. Suponga que el archivo es demasiado grande como para leerlo en memoria (varios de Gigabytes) por no puede usar listas de Python para almacenarlo. En vez, debe ir leyéndolo desde el archivo y ir escribiendo de inmediato lo que vaya ordenando en otros archivo. **Nota**: es posible usar listas de Python de tamaño fijo (2, 4, o 10 elementos, si necesita); el punto es que no puede leer todo el archivo a una lista de Python y luego hacer mergesort a la lista.

[ ] *3 celdas ocultas*

Bubblesort

Este es un algoritmo de ordenación más simple que mergesort pero más lento.

Idea: en cada una de las pasadas "hacer subir la burbuja (el mayor) al último lugar"

Algoritmo:

- Repetir $n-1$ veces(para los $n, n-1, \ldots, 2$ primeros elementos)
    - Recorrer todos los elementos (salvo el último)
    - Comparar elemento con siguiente
    - Intercambiar si están fuera de orden, es decir, si elemento es mayor que el siguiente

Intuición: Este algoritmo es lo que Ud. quizás haría para ordenar una fila de personas. Comenzando del inicio de la fila, compararía a la primera persona con el segunda persona. Si la primera es más alta, las intercambiaría de posición. Luego, compararía a la persona actualmente segunda con la tercera; si la segunda es más alta, los intercambiaría de posición. Etc. Y seguría así hasta el final. Note que estos intercambios ¡no dejan a todos ordenados en una primera pasada! Debe volver a revisarlos de nuevo a todos desde el comienzo, de a pares consecutivos (con el mismo procedimiento descrito), haciendo los intercambios que corresponda. Sólo cuando ha revisado a pares desde el inicio al final y NO ha hecho ningún intercambio, es que puede decir que la lista está ordenada.

```
# Bubblesort

# bubblesort: list(int) -> None
# efecto: ordena ascendemente la lista usando el algoritmo Bubblesort
# ejemplo: bubblesort([258, 45, 99, 14, 102, 75, 128]) modifica la lista
# y la deja en el estado [14, 45, 75, 99, 102, 128, 258]
def bubblesort(lista):
    k = len(lista)
    while k > 1:
        # hacer una pasada sobre lista[0], ..., lista[k - 1]
        for j in range(k-1):
            if lista[j] > lista[j + 1]:
                # intercambiar a[j] con a[j+1]
                # sintaxis para swap con listas de Python
                lista[j], lista[j + 1] = lista[j + 1], lista[j]
        # disminuir k
        k = k - 1

# Test
lista = [258, 45, 99, 14, 102, 75, 128]
bubblesort(lista)
assert lista == [14, 45, 75, 99, 102, 128, 258]
```

Entendiendo Bubblesort:

Intentaremos ordenar manualmente la siguiente lista, de menor a mayor, usando bubblesort:

```
["D","C","B","E","A"]
```

para eso mostraremos las distintas pasadas en una tabla, a continuación. En la tabla, cada línea muestra una "pasada" ordenando de a pares, mostrando cuáles elementos se comparan y si se intercambian o no.

Aquí usaremos la notación:

- "X" con "Y": significa que comparamos la letra X con la letra Y,
- Si aparece **"(si)"** significa que se hizo un un intercambio.
- Si aparece **"(no)"** es que se hizo un un intercambio.
- Al final de cada línea, se muestra cómo quedó la lista en esa pasada.

```
Lista original: ["D","C","B","E","A"]
```

Recorrido	1ra comp.	2da comp.	3ra comp.	4ta comp.	lista resultado al final de la pasada
1ro	"D" con "C":(si)	"D" con "B":(si)	"D" con "E": (no)	"E" con "A":(si)	["C","B","D","A","E"]
2do	"C" con "B":(si)	"C" con "D":(no)	"D" con "A": (si)	"D" con "E":(no)	["B","C","A","D","E"]
3ro	"B" con "C":(no)	"C" con "A":(si)	"C" con "D":(no)	"D" con "E":(no)	["B","A","C","D","E"]
4to	"B" con "A":(si)	"B" con "C":(no)	"C" con "D":(no)	"D" con "E":(no)	["A","B","C","D","E"]
4to (n)	"A" con "B":(no)	"B" con "C":(no)	"C" con "D":(no)	"D" con "E":(no)	["A","B","C","D","E"]

Quicksort

Este algoritmo es un algoritmo muy rápido, más rápido que Bubblesort pero algo más difícil de entender. Sin embargo es uno de los más usados en la práctica, porque

- es rápido, y
- ordena las listas "en el lugar", sin requerir crear otras listas (en su implementación más simple, Mergesort requiere usar otras listas).

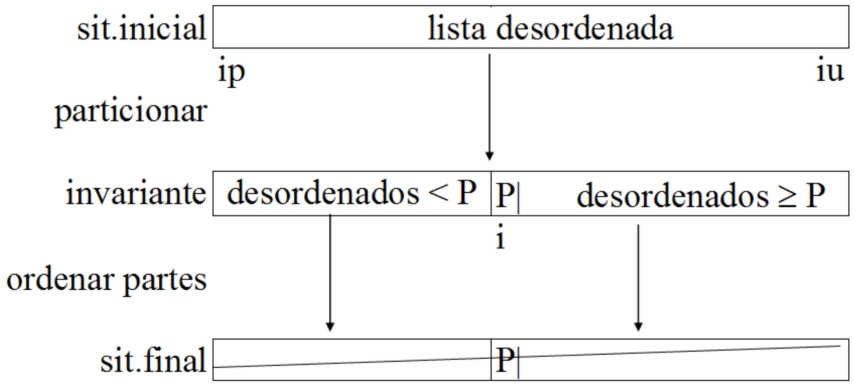
Introducción: Quicksort es un algoritmo recursivo de ordenamiento. Se basa en la siguiente idea:

1. Dada la lista desordenada, se escoge un elemento "pivote" dentro de la lista (por ejemplo, el que está al medio). Llamémoslo **P**.

2. El resto de la lista se compara con **P**. Los elementos *menores que **P*** se copian a la izquierda de **P**, y los elementos *mayores que **P*** se copian a la derecha de **P**.
3. Y luego se ordenan recursivamente las listas a la izquierda y derecha de **P**.

Nota: Al paso 2, donde se mueven menores y mayores con respecto a **P**, se le llama **particionar**.

Gráficamente:



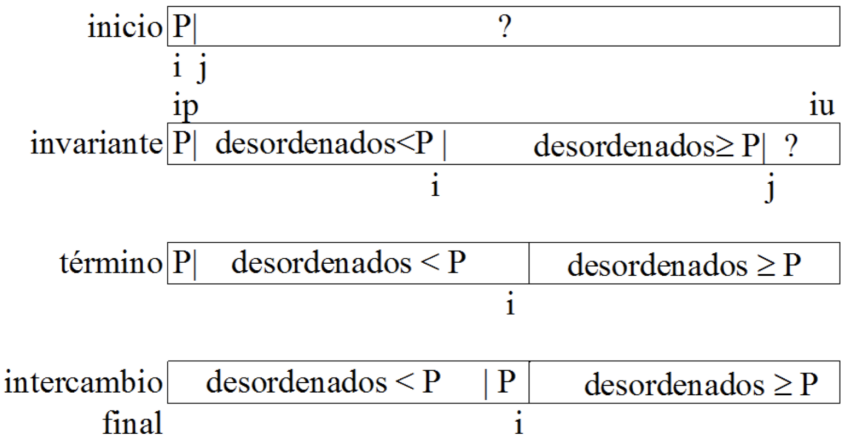
Detalles:

- P: se denomina “pivote”
    - P puede ser cualquier valor de la lista
- La recomendación es siempre escogerlo aleatoriamente entre los valores de la lista
- Otro método de selección: escoger tres valores aleatorios de la lista, el del medio de los tres se escoge como pivote

```
# qsort: list(int) int int -> None
# efecto: ordena ascendemente la lista usando el algoritmo Quicksort
# en el rango [ip,iu]
# ejemplo: qsort([258, 45, 99, 14, 102, 75, 128], 0, 6) modifica la lista
# y la deja en el estado [14, 45, 75, 99, 102, 128, 258]
def qsort(lista, ip, iu):
    # si ip >= iu no se hace nada
    if ip < iu:
        pivote = particionar(lista, ip, iu)
        qsort(lista, ip, pivote - 1)
        qsort(lista, pivote + 1, iu)

# Test
lista = [258, 45, 99, 14, 102, 75, 128]
qsort(lista, 0, 6)
assert lista == [14, 45, 75, 99, 102, 128, 258]
```

Particionar




```
# Quicksort

import random

# particionar: list(int) int int -> int
# escoge aleatoriamente un pivote y devuelve la posicion donde queda el pivote
# luego de particionar la lista en el rango [ip,iu]
# efecto: deja todos los valores menores que el pivote a la izquierda de este
# y todos los valores mayores que el pivote a la derecha de este
# ejemplo: particionar([258, 45, 99, 14, 102, 75, 128], 0, 6) y el pivote es 99
# entonces la lista queda como [45, 14, 75, 99, 258, 102, 128] y devuelve 3
def particionar(lista, ip, iu):
    # Pre-condicion: ip < iu
    assert ip < iu
    # se escoge pivote aleatoriamente
    pivote = random.randint(ip, iu)
    # se intercambia con el primero del segmento de lista
    lista[ip], lista[pivote] = lista[pivote], lista[ip]
    # se particiona usando el pivote
    i = ip
    for j in range(ip + 1, iu + 1):
        if lista[j] < lista[ip]: # recordar que pivote esta en ip
            i = i + 1
            lista[i], lista[j] = lista[j], lista[i]
    # lista[i] contiene el ultimo valor que es menor que el pivote
    lista[ip], lista[i] = lista[i], lista[ip]
    return i

# Test
lista = [258, 45, 99, 14, 102, 75, 128]
pivote = particionar(lista, 0, 6)
for indice in range(0, pivote - 1):
    assert lista[indice] < lista[pivote]
for indice in range(pivote + 1, len(lista)):
    assert lista[indice] > lista[pivote]
```

```
# quicksort: list(int) -> None
# efecto: ordena ascendemente la lista usando el algoritmo Quicksort
# ejemplo: quicksort([258, 45, 99, 14, 102, 75, 128]) modifica la lista
# y la deja en el estado [14, 45, 75, 99, 102, 128, 258]
def quicksort(lista):
    qsort(lista, 0, len(lista) - 1)

# Test
lista = [258, 45, 99, 14, 102, 75, 128]
quicksort(lista)
assert lista == [14, 45, 75, 99, 102, 128, 258]
lista = [20, 19, 18, 17, 16, 15, 14, 13, 12, 11, 10, 9, 8, 7, 6, 5, 4, 3, 2, 1]
quicksort(lista)
assert lista == [1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19, 20]
```

Eficiencia

- Tiempos de ordenamiento para una lista con n valores
    - Bubblesort: proporcional a n^2
    - Quicksort: proporcional a $n \log n$ en promedio (tiene un peor caso $n \hat{} 2$)
    - Mergesort: proporcional a $n \log n$ en promedio. Su peor caso es también $n \log n$ por lo que a veces es preferido a Quicksort, por ejemplo cuando las listas a ordenar no requieren de mucha memoria.

Eso es todo amigxs! Espero hayan disfrutado su caso de estudio :-D