Módulo 2 - Capítulo 5

Unidad 27

Quiz 1

¿Cuántas operaciones de intercambio se han ejecutado en el proceso de ordenamiento de burbujas siguiente?

```
def bubble(S):
    n = len(S)
    exchange_counter = 0

    for _ in range(n):
        print(S)
        for j in range(n - 1):
            if S[j] > S[j + 1]:
                exchange_counter += 1
                print('\t>>> exchange_counter:', exchange_counter)
                S[j], S[j + 1] = S[j + 1], S[j]

    print('\n>>> Final exchange_counter:', exchange_counter)

S = [50, 30, 40, 10, 20]
bubble(S)
```

```
[50, 30, 40, 10, 20]
    >>> exchange_counter: 1
    >>> exchange_counter: 2
    >>> exchange_counter: 3
    >>> exchange_counter: 4
[30, 40, 10, 20, 50]
    >>> exchange_counter: 5
    >>> exchange_counter: 6
[30, 10, 20, 40, 50]
    >>> exchange_counter: 7
    >>> exchange_counter: 8
[10, 20, 30, 40, 50]
[10, 20, 30, 40, 50]

>>> Final exchange_counter: 8
```

Quiz 2

¿Cuántas operaciones de comparación se han ejecuado en el proceso de ordenamiento por inserción que se muestra a continuación?

```python
def insertion_sort(S):
    n = len(S)
    comparation_counter = 0

    for i in range(1, n):
        print(S)
        x = S[i]
        j = i - 1

        while j >= 0 and S[j] > x:
            S[j + 1] = S[j]
            j -= 1
            comparation_counter += 1
            print('\t>>> comparation_counter:', comparation_counter)

        S[j + 1] = x

    print('\n>>> Final comparation_counter:', comparation_counter)


S = [50, 30, 40, 10, 20]
insertion_sort(S)
```

```
[50, 30, 40, 10, 20]
    >>> comparation_counter: 1
[30, 50, 40, 10, 20]
    >>> comparation_counter: 2
[30, 40, 50, 10, 20]
    >>> comparation_counter: 3
    >>> comparation_counter: 4
    >>> comparation_counter: 5
[10, 30, 40, 50, 20]
    >>> comparation_counter: 6
    >>> comparation_counter: 7
    >>> comparation_counter: 8

>>> Final comparation_counter: 8
```

Quiz 3

Dadas dos palabras, escribe un algoritmo que determine si estas dos cadenas son anagramas. Un anagrama es una palabra que se forma reordenando las letras de otra palabra utilizando todas las letras originales exactamente una vez. (Por ejemplo, `LISTEN` y `SILENT` son anagramas).

```python
def convert_to_ascci(S: str):
    return [ ord(c) for c in S ]
```

```
def insertion_sort(S):
    n = len(S)

    for i in range(1, n):
        x = S[i]
        j = i - 1

        while j >= 0 and S[j] > x:
            S[j + 1] = S[j]
            j -= 1
        S[j + 1] = x

    return S

def is_anagram(a, b, sort_fun = insertion_sort):
    a = sort_fun(convert_to_ascci(a.lower()))
    b = sort_fun(convert_to_ascci(b.lower()))

    print("¿Las cadenas son anagramas? =>", a == b)


a = "lIsTeN"
b = "SiLeNt"

is_anagram(a, b)
```

```
¿Las cadenas son anagramas? => True
```

```
a = input("Ingrese la cadena a: ")
b = input("Ingrese la cadena b: ")

is_anagram(a, b)
```

```
¿Las cadenas son anagramas? => True
```

Quiz 4

El uso de un algoritmo de clasificación permite determinar fácilmente si se trata de un anagrama o no.

- Cree una función que evalúe los anagramas utilizando la función `sorted()` incorporada de python
- Modificar la función `selection_sort()` para crear una función que determine los anagramas
- Modificar la función `insertion_sort()` para crear una función que determine los anagramas

```

```

```
def selection_sort(S):
    n = len(S)
    for i in range(n - 1):
        smallest = i
        for j in range(i + 1, n):
            if S[j] < S[smallest]:
                smallest = j
        S[i], S[smallest] = S[smallest], S[i]
    return S
```

```
def insertion_sort(S):
    n = len(S)
    for i in range(1, n):
        x = S[i]
        j = i - 1
        while j >= 0 and S[j] > x:
            S[j + 1] = S[j]
            j -= 1
        S[j + 1] = x
    return S
```

```
a = "lIsTeN"
b = "SiLeNt"

is_anagram(a, b, sort_fun=sorted)
is_anagram(a, b, sort_fun=selection_sort)
is_anagram(a, b, sort_fun=insertion_sort)

a = "lIsTeN"
b = "SiLeNcE"

is_anagram(a, b, sort_fun=sorted)
is_anagram(a, b, sort_fun=selection_sort)
is_anagram(a, b, sort_fun=insertion_sort)
```

```
¿Las cadenas son anagramas? => True
¿Las cadenas son anagramas? => True
¿Las cadenas son anagramas? => True
¿Las cadenas son anagramas? => False
¿Las cadenas son anagramas? => False
¿Las cadenas son anagramas? => False
```

Unidad 28

Quiz 1

¿Cuántas veces se ejecutó la función `merge_sort()` en el proceso de ordenamiento por mezcla que se muestra a continuación?

```
def merge(S, low, mid, high):
    R = []
    i, j = low, mid + 1

    for _ in range(low, high+1):
        if i > mid:
            R.append(S[j])
            j += 1
        elif j > high:
            R.append(S[i])
            i += 1
        elif S[i] < S[j]:
            R.append(S[i])
            i += 1
        else:
            R.append(S[j])
            j += 1
    for k in range(len(R)):
        S[low] = R[k]
        low += 1

execution_counter = 0

def merge_sort(S, low, high):
    global execution_counter
    execution_counter += 1
    print('\t>>> S:', S)
    print('\t>>> execution_counter:', execution_counter)

    if low < high:
        mid = (low + high) // 2
        merge_sort(S, low, mid)
        merge_sort(S, mid + 1, high)
        merge(S, low, mid, high)

def evaluate_execution(S):
    global execution_counter
    execution_counter = 0

    merge_sort(S, 0, len(S) - 1)
    print('>>> S:', S)
    print('>>> Final execution_counter:', execution_counter)
```

```
S = [6, 2, 11, 7, 5, 4, 8, 16, 10, 3]
evaluate_execution(S)
```

```
    >>> S: [6, 2, 11, 7, 5, 4, 8, 16, 10, 3]
    >>> execution_counter: 1
    >>> S: [6, 2, 11, 7, 5, 4, 8, 16, 10, 3]
    >>> execution_counter: 2
    >>> S: [6, 2, 11, 7, 5, 4, 8, 16, 10, 3]
    >>> execution_counter: 3
    >>> S: [6, 2, 11, 7, 5, 4, 8, 16, 10, 3]
    >>> execution_counter: 4
    >>> S: [6, 2, 11, 7, 5, 4, 8, 16, 10, 3]
    >>> execution_counter: 5
    >>> S: [6, 2, 11, 7, 5, 4, 8, 16, 10, 3]
    >>> execution_counter: 6
    >>> S: [2, 6, 11, 7, 5, 4, 8, 16, 10, 3]
    >>> execution_counter: 7
    >>> S: [2, 6, 11, 7, 5, 4, 8, 16, 10, 3]
    >>> execution_counter: 8
    >>> S: [2, 6, 11, 7, 5, 4, 8, 16, 10, 3]
    >>> execution_counter: 9
    >>> S: [2, 6, 11, 7, 5, 4, 8, 16, 10, 3]
    >>> execution_counter: 10
    >>> S: [2, 5, 6, 7, 11, 4, 8, 16, 10, 3]
    >>> execution_counter: 11
    >>> S: [2, 5, 6, 7, 11, 4, 8, 16, 10, 3]
    >>> execution_counter: 12
    >>> S: [2, 5, 6, 7, 11, 4, 8, 16, 10, 3]
    >>> execution_counter: 13
    >>> S: [2, 5, 6, 7, 11, 4, 8, 16, 10, 3]
    >>> execution_counter: 14
    >>> S: [2, 5, 6, 7, 11, 4, 8, 16, 10, 3]
    >>> execution_counter: 15
    >>> S: [2, 5, 6, 7, 11, 4, 8, 16, 10, 3]
    >>> execution_counter: 16
    >>> S: [2, 5, 6, 7, 11, 4, 8, 16, 10, 3]
    >>> execution_counter: 17
    >>> S: [2, 5, 6, 7, 11, 4, 8, 16, 10, 3]
    >>> execution_counter: 18
    >>> S: [2, 5, 6, 7, 11, 4, 8, 16, 10, 3]
    >>> execution_counter: 19
>>> S: [2, 3, 4, 5, 6, 7, 8, 10, 11, 16]
>>> Final execution_counter: 19
```

```
S = [6, 2, 11, 7, 5, 4, 8, 16, 10, 3, 1, 12, 9]
evaluate_execution(S)
```

```
>>> S: [6, 2, 11, 7, 5, 4, 8, 16, 10, 3, 1, 12, 9]
>>> execution_counter: 1
>>> S: [6, 2, 11, 7, 5, 4, 8, 16, 10, 3, 1, 12, 9]
>>> execution_counter: 2
>>> S: [6, 2, 11, 7, 5, 4, 8, 16, 10, 3, 1, 12, 9]
>>> execution_counter: 3
>>> S: [6, 2, 11, 7, 5, 4, 8, 16, 10, 3, 1, 12, 9]
>>> execution_counter: 4
>>> S: [6, 2, 11, 7, 5, 4, 8, 16, 10, 3, 1, 12, 9]
>>> execution_counter: 5
>>> S: [6, 2, 11, 7, 5, 4, 8, 16, 10, 3, 1, 12, 9]
>>> execution_counter: 6
>>> S: [2, 6, 11, 7, 5, 4, 8, 16, 10, 3, 1, 12, 9]
>>> execution_counter: 7
>>> S: [2, 6, 11, 7, 5, 4, 8, 16, 10, 3, 1, 12, 9]
>>> execution_counter: 8
>>> S: [2, 6, 11, 7, 5, 4, 8, 16, 10, 3, 1, 12, 9]
>>> execution_counter: 9
>>> S: [2, 6, 7, 11, 5, 4, 8, 16, 10, 3, 1, 12, 9]
>>> execution_counter: 10
>>> S: [2, 6, 7, 11, 5, 4, 8, 16, 10, 3, 1, 12, 9]
>>> execution_counter: 11
>>> S: [2, 6, 7, 11, 5, 4, 8, 16, 10, 3, 1, 12, 9]
>>> execution_counter: 12
>>> S: [2, 6, 7, 11, 5, 4, 8, 16, 10, 3, 1, 12, 9]
>>> execution_counter: 13
>>> S: [2, 6, 7, 11, 4, 5, 8, 16, 10, 3, 1, 12, 9]
>>> execution_counter: 14
>>> S: [2, 4, 5, 6, 7, 8, 11, 16, 10, 3, 1, 12, 9]
>>> execution_counter: 15
>>> S: [2, 4, 5, 6, 7, 8, 11, 16, 10, 3, 1, 12, 9]
>>> execution_counter: 16
>>> S: [2, 4, 5, 6, 7, 8, 11, 16, 10, 3, 1, 12, 9]
>>> execution_counter: 17
>>> S: [2, 4, 5, 6, 7, 8, 11, 16, 10, 3, 1, 12, 9]
>>> execution_counter: 18
>>> S: [2, 4, 5, 6, 7, 8, 11, 16, 10, 3, 1, 12, 9]
>>> execution_counter: 19
>>> S: [2, 4, 5, 6, 7, 8, 11, 10, 16, 3, 1, 12, 9]
>>> execution_counter: 20
>>> S: [2, 4, 5, 6, 7, 8, 11, 3, 10, 16, 1, 12, 9]
>>> execution_counter: 21
>>> S: [2, 4, 5, 6, 7, 8, 11, 3, 10, 16, 1, 12, 9]
>>> execution_counter: 22
>>> S: [2, 4, 5, 6, 7, 8, 11, 3, 10, 16, 1, 12, 9]
>>> execution_counter: 23
>>> S: [2, 4, 5, 6, 7, 8, 11, 3, 10, 16, 1, 12, 9]
>>> execution_counter: 24
>>> S: [2, 4, 5, 6, 7, 8, 11, 3, 10, 16, 1, 12, 9]
```

```
    >>> execution_counter: 25
>>> S: [1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 16]
>>> Final execution_counter: 25
```

Quiz 2

Dadas N listas ordenadas como entrada, escribe un programa que las fusione en una lista ordenada

```
def multiple_merge(numbers_list):
    if len(numbers_list) == 1:
        return numbers_list[0]
    else:
        mid = len(numbers_list) // 2
        left = multiple_merge(numbers_list[:mid])
        right = multiple_merge(numbers_list[mid:])
        return merge(left, right)


def merge(left, right):
    merged_list = []
    i = j = 0
    while i < len(left) and j < len(right):
        if left[i] <= right[j]:
            merged_list.append(left[i])
            i += 1
        else:
            merged_list.append(right[j])
            j += 1
    merged_list += left[i:]
    merged_list += right[j:]
    return merged_list
```

```
N = int(input("Ingrese el número de listas: "))
numbers_list = []

for i in range(N):
    numbers = list(map(int, input("Ingrese una lista de números: ").split()))
    print(numbers)

    numbers_list.append(numbers)

ordered_list = multiple_merge(numbers_list)
print("Fusionada dentro: ", ordered_list)
```

```
[9, 8, 7]
[6, 5, 4]
```

```
[3, 2, 1]
Fusionada dentro:  [3, 2, 1, 6, 5, 4, 9, 8, 7]
```

Unidad 29

Quiz 1

Dada la siguiente lista, escriba la salida después de ejecutar la función `partition()`

```
def partition(S, low, high):
    pivot = S[low]
    left, right = low + 1, high

    while left < right:
        while left <= right and S[left] <= pivot:
            left += 1
        while left <= right and S[right] >= pivot:
            right -= 1

        if left < right:
            S[left], S[right] = S[right], S[left]

    pivot_point = right
    S[low], S[pivot_point] = S[pivot_point], S[low]

    return pivot_point

def quick_sort(S, low, high):
    if low < high:
        pivot_point = partition(S, low, high)
        print('\t>>> pivot_point:', pivot_point)
        quick_sort(S, low, pivot_point - 1)
        quick_sort(S, pivot_point + 1, high)
```

```
S = [15, 10, 12, 20, 25, 13, 22]
quick_sort(S, 0, len(S) - 1)
print(S)
```

```
    >>> pivot_point: 3
    >>> pivot_point: 2
    >>> pivot_point: 1
    >>> pivot_point: 6
    >>> pivot_point: 5
[10, 12, 13, 15, 20, 22, 25]
```

Quiz 2

Escriba un algoritmo que encuentre el K° elemento mayor, dados N elementos desordenados.

Después de resolver el problema con los dos métodos anteriores, analiza qué algoritmo es más eficiente.

- Puede devolver el elemento K° después de utilizar la función de ordenamiento.
- Puede utilizar la función partition() para realizar una llamada recursiva hasta que el pivote sea el K° elemento.

```
from random import randint
import time

random_numbers = 0
k = 3

def measure_time(func):
    global random_numbers
    random_numbers = [randint(0, 10_000) for _ in range(10_000)]
    def measure_function(*args, **kwargs):
        start = time.time()
        nums = func(*args, **kwargs)
        k_number = nums[-k]
        end = time.time()
        print('>>> Execution time:', end - start)
        print('>>> k_number:', k_number)
        return k_number
    return measure_function
```

Buble Sort

```
@measure_time
def bubble(S):
    n = len(S)

    for _ in range(n):
        for j in range(n - 1):
            if S[j] > S[j + 1]:
                S[j], S[j + 1] = S[j + 1], S[j]

    return S

bubble(random_numbers)
```

```
>>> Execution time: 8.513029098510742
>>> k_number: 9998
9998
```

Insertion Sort

```
@measure_time
def insertion_sort(S):
    n = len(S)

    for i in range(1, n):
        x = S[i]
        j = i - 1

        while j >= 0 and S[j] > x:
            S[j + 1] = S[j]
            j -= 1

        S[j + 1] = x

    return S

insertion_sort(random_numbers)
```

```
>>> Execution time: 2.000645160675049
>>> k_number: 10000
10000
```

Selection Sort

```
@measure_time
def selection_sort(S):
    n = len(S)
    for i in range(n - 1):
        smallest = i
        for j in range(i + 1, n):
            if S[j] < S[smallest]:
                smallest = j
        S[i], S[smallest] = S[smallest], S[i]
    return S

selection_sort(random_numbers)
```

```
>>> Execution time: 1.975259780883789
>>> k_number: 9997
9997
```

Merge Sort

```
def merge(S, low, mid, high):
    R = []
    i, j = low, mid + 1

    for _ in range(low, high+1):
        if i > mid:
            R.append(S[j]); j += 1
        elif j > high:
            R.append(S[i]); i += 1
        elif S[i] < S[j]:
            R.append(S[i]); i += 1
        else:
            R.append(S[j]); j += 1
    for k in range(len(R)):
        S[low] = R[k]; low += 1

def merge_sort(S, low, high):
    if low < high:
        mid = (low + high) // 2
        merge_sort(S, low, mid)
        merge_sort(S, mid + 1, high)
        merge(S, low, mid, high)
    return S

@measure_time
def exe_merge_sort():
    return merge_sort(random_numbers, 0, len(random_numbers) - 1)

exe_merge_sort()
```

```
>>> Execution time: 0.028768539428710938
>>> k_number: 9998
9998
```

Quick Sort

```
def partition(S, low, high):
    pivot = S[low]
    left, right = low + 1, high

    while left < right:
        while left <= right and S[left] <= pivot:
            left += 1
        while left <= right and S[right] >= pivot:
            right -= 1
```

```
        if left < right:
            S[left], S[right] = S[right], S[left]

    pivot_point = right
    S[low], S[pivot_point] = S[pivot_point], S[low]

    return pivot_point


def quick_sort(S, low, high):
    if low < high:
        pivot_point = partition(S, low, high)
        quick_sort(S, low, pivot_point - 1)
        quick_sort(S, pivot_point + 1, high)

    return S


@measure_time
def exe_quick_sort():
    return quick_sort(random_numbers, 0, len(random_numbers) - 1)


exe_quick_sort()
```

```
>>> Execution time: 0.013513326644897461
>>> k_number: 9999
9999
```

Despues de comparar los tiempos de ejecución en los diversos métodos de ordenamiento, he determinado que el mejor método para un arreglo con 10.000 de números aleatorios es el método `quick_sort()` con un tiempo de ejecución equivalente a `0.0135` segundos.