# Práctica 3

- El Problema de Selección
  - *QuickSelect*
  - *QuickSort*
- Programación Dinámica
  - El Problema de Dar Cambio en Dinero
  - El Problema de la Mochila

I-A. *QuickSelect* básico

Vamos a implementar el método *QuickSelect* para la determinación del elemento que ocupa la posición del índice *k* en una tabla, primero usando una función *split* que parta una tabla usando como pivote el primer elemento de la tabla.

1. Escribir una función
split(t: np.ndarray)-> Tuple[np.ndarray, int, np.ndarray]
que reparta los elementos de *t* entre dos arrays con los elementos menores y mayores que *t[0]* y devuelva una tupla con los elementos menores, el elemento *t[0]* y los elementos mayores.

2. Escribir una función
qsel(t: np.ndarray, k: int)-> Union[int, None]
que aplique de manera recursiva el algoritmo *QuickSelect* usando la función *split* anterior y devuelva el valor del elemento que ocuparía el índice *k* en una ordenación de *t* si ese elemento existe y *None* si no.

3. Escribir una función no recursiva
qsel_nr(t: np.ndarray, k: int)-> Union[int, None]
que elimine la recursión de cola de la función anterior.

```python
t = np.random.permutation(10)
shift = 0
for k in range(len(t)):
    print("QSEL: EJEMPLO CON 10 ELEMENTOS")
    print(t)
    print()
    print("Buscando rec:", k, "en:", t)
    print("Buscado  rec:", k, "Encontrado:", qsel(t, k+shift))
    print()
    print("Buscando nr :", k, "en:", t)
    print("Buscado  nr :", k, "Encontrado:", qsel_nr(t, k+shift))
    print()
```

```
QSEL: EJEMPLO CON 10 ELEMENTOS
[8 7 3 6 0 1 2 4 5 9]

Buscando rec: 0 en: [8 7 3 6 0 1 2 4 5 9]
qsel( t = [8 7 3 6 0 1 2 4 5 9] , k = 0 )
split [7, 3, 6, 0, 1, 2, 4, 5] 8 [9]
qsel( t = [7, 3, 6, 0, 1, 2, 4, 5] , k = 0 )
split [3, 6, 0, 1, 2, 4, 5] 7 []
qsel( t = [3, 6, 0, 1, 2, 4, 5] , k = 0 )
split [0, 1, 2] 3 [6, 4, 5]
qsel( t = [0, 1, 2] , k = 0 )
split [] 0 [1, 2]
0 enc-en-medio-del-split
Buscado  rec: 0 Encontrado: 0

Buscando nr : 0 en: [8 7 3 6 0 1 2 4 5 9]
qsel_nr(t = [8 7 3 6 0 1 2 4 5 9] , k = 0 )
split [7, 3, 6, 0, 1, 2, 4, 5] 8 [9]
split [3, 6, 0, 1, 2, 4, 5] 7 []
split [0, 1, 2] 3 [6, 4, 5]
split [] 0 [1, 2]
0 enc-en-medio-del-split
Buscado  nr : 0 Encontrado: 0
```

```python
t = np.random.permutation(10)
shift = 0
for k in range(len(t)):
    print("QSEL: EJEMPLO CON 10 ELEMENTOS")
    print(t)
    print()
    print("Buscando rec:", k, "en:", t)
    print("Buscado  rec:", k, "Encontrado:", qsel(t, k+shift))
    print()
    print("Buscando nr :", k, "en:", t)
    print("Buscado  nr :", k, "Encontrado:", qsel_nr(t, k+shift))
    print()
```

```
QSEL: EJEMPLO CON 10 ELEMENTOS
[8 7 3 6 0 1 2 4 5 9]

Buscando rec: 4 en: [8 7 3 6 0 1 2 4 5 9]
qsel( t = [8 7 3 6 0 1 2 4 5 9] , k = 4 )
split [7, 3, 6, 0, 1, 2, 4, 5] 8 [9]
qsel( t = [7, 3, 6, 0, 1, 2, 4, 5] , k = 4 )
split [3, 6, 0, 1, 2, 4, 5] 7 []
qsel( t = [3, 6, 0, 1, 2, 4, 5] , k = 4 )
split [0, 1, 2] 3 [6, 4, 5]
qsel( t = [6, 4, 5] , k = 0 )
split [4, 5] 6 []
qsel( t = [4, 5] , k = 0 )
split [] 4 [5]
4 enc-en-medio-del-split
Buscado  rec: 4 Encontrado: 4

Buscando nr : 4 en: [8 7 3 6 0 1 2 4 5 9]
qsel_nr(t = [8 7 3 6 0 1 2 4 5 9] , k = 4 )
split [7, 3, 6, 0, 1, 2, 4, 5] 8 [9]
split [3, 6, 0, 1, 2, 4, 5] 7 []
split [0, 1, 2] 3 [6, 4, 5]
split [4, 5] 6 []
split [] 4 [5]
4 enc-en-medio-del-split
Buscado  nr : 4 Encontrado: 4
```

```python
t = np.random.permutation(10)
shift = 0
for k in range(len(t)):
    print("QSEL: EJEMPLO CON 10 ELEMENTOS")
    print(t)
    print()
    print("Buscando rec:", k, "en:", t)
    print("Buscado  rec:", k, "Encontrado:", qsel(t, k+shift))
    print()
    print("Buscando nr :", k, "en:", t)
    print("Buscado  nr :", k, "Encontrado:", qsel_nr(t, k+shift))
    print()
```

```
QSEL: EJEMPLO CON 10 ELEMENTOS
[8 7 3 6 0 1 2 4 5 9]

Buscando rec: 7 en: [8 7 3 6 0 1 2 4 5 9]
qsel( t = [8 7 3 6 0 1 2 4 5 9] , k = 7 )
split [7, 3, 6, 0, 1, 2, 4, 5] 8 [9]
qsel( t = [7, 3, 6, 0, 1, 2, 4, 5] , k = 7 )
split [3, 6, 0, 1, 2, 4, 5] 7 []
7 enc-en-medio-del-split
Buscado  rec: 7 Encontrado: 7

Buscando nr : 7 en: [8 7 3 6 0 1 2 4 5 9]
qsel_nr(t = [8 7 3 6 0 1 2 4 5 9] , k = 7 )
split [7, 3, 6, 0, 1, 2, 4, 5] 8 [9]
split [3, 6, 0, 1, 2, 4, 5] 7 []
7 enc-en-medio-del-split
Buscado  nr : 7 Encontrado: 7
```

I-B. *QuickSelect_5*

Vamos a modificar la implementación anterior de *QuickSelect* con una selección de pivote mediante el procedimiento “mediana de medianas de cinco elementos”.

1. Escribir una función
split_pivot(t: np.ndarray, mid: int)-> Tuple[np.ndarray, int, np.ndarray]
que modifique la funcion *split* anterior de manera que use el valor *mid* para dividir *t* .

2. Escribir una función
pivot5(t: np.ndarray)-> int
que devuelve el “pivote 5” del array *t* de acuerdo al procedimiento “mediana de medianas de 5 elementos” y llamando a la función **qsel5_nr** que se define a continuación.

3. Escribir una función no recursiva
qsel5_nr(t: np.ndarray, k: int)-> Union[int, None]
que devuelve el elemento en el índice *k* de una ordenación de *t* utilizando la funciones *pivot5*, *split_pivot* anteriores.

```python
t = list(np.random.permutation(10).astype(int))
shift = 0
for k in range(len(t)):
    print("QSEL5: EJEMPLO CON 10 ELEMENTOS")
    print(t)
    print()
    print("Buscando:", k, "en:", t)
    print("Buscado:", k, "Encontrado:", qsel5_nr(t, k+shift))
    print()
```

```
QSEL5: EJEMPLO CON 10 ELEMENTOS
[1, 6, 2, 4, 3, 0, 9, 5, 7, 8]

Buscando: 0 en: [1, 6, 2, 4, 3, 0, 9, 5, 7, 8]
qsel5_nr(t = [1, 6, 2, 4, 3, 0, 9, 5, 7, 8] , k = 0 )
pivot5( [1 6 2 4 3 0 9 5 7 8] )
qsel5_nr(t = [3. 7.] , k = 1 )
7 enc-en-tupla-final: [3. 7.]
pivot5( [1 6 2 4 3 0 9 5 7 8] ) = 7
split [1 6 2 4 3 0 5] 7 [9 8]
pivot5( [1 6 2 4 3 0 5] )
qsel5_nr(t = [3.] , k = 0 )
3 enc-en-tupla-final: [3.]
pivot5( [1 6 2 4 3 0 5] ) = 3
split [1 2 0] 3 [6 4 5]
0 enc-en-tupla-final: [1 2 0]
Buscado: 0 Encontrado: 0
```

```python
t = list(np.random.permutation(10).astype(int))
shift = 0
for k in range(len(t)):
    print("QSEL5: EJEMPLO CON 10 ELEMENTOS")
    print(t)
    print()
    print("Buscando:", k, "en:", t)
    print("Buscado:", k, "Encontrado:", qsel5_nr(t, k+shift))
    print()
```

```
QSEL5: EJEMPLO CON 10 ELEMENTOS
[1, 6, 2, 4, 3, 0, 9, 5, 7, 8]

Buscando: 4 en: [1, 6, 2, 4, 3, 0, 9, 5, 7, 8]
qsel5_nr(t = [1, 6, 2, 4, 3, 0, 9, 5, 7, 8] , k = 4 )
pivot5( [1 6 2 4 3 0 9 5 7 8] )
qsel5_nr(t = [3. 7.] , k = 1 )
7 enc-en-tupla-final: [3. 7.]
pivot5( [1 6 2 4 3 0 9 5 7 8] ) = 7
split [1 6 2 4 3 0 5] 7 [9 8]
pivot5( [1 6 2 4 3 0 5] )
qsel5_nr(t = [3.] , k = 0 )
3 enc-en-tupla-final: [3.]
pivot5( [1 6 2 4 3 0 5] ) = 3
split [1 2 0] 3 [6 4 5]
4 enc-en-tupla-final: [6 4 5]
Buscado: 4 Encontrado: 4
```

```python
t = list(np.random.permutation(10).astype(int))
shift = 0
for k in range(len(t)):
    print("QSEL5: EJEMPLO CON 10 ELEMENTOS")
    print(t)
    print()
    print("Buscando:", k, "en:", t)
    print("Buscado:", k, "Encontrado:", qsel5_nr(t, k+shift))
    print()
```

```
QSEL5: EJEMPLO CON 10 ELEMENTOS
[1, 6, 2, 4, 3, 0, 9, 5, 7, 8]

Buscando: 7 en: [1, 6, 2, 4, 3, 0, 9, 5, 7, 8]
qsel5_nr(t = [1, 6, 2, 4, 3, 0, 9, 5, 7, 8] , k = 7 )
pivot5( [1 6 2 4 3 0 9 5 7 8] )
qsel5_nr(t = [3. 7.] , k = 1 )
7 enc-en-tupla-final: [3. 7.]
pivot5( [1 6 2 4 3 0 9 5 7 8] ) = 7
split [1 6 2 4 3 0 5] 7 [9 8]
7 enc-en-medio-del-split
Buscado: 7 Encontrado: 7
```

I-C. *QuickSort_5*

Finalmente, vamos a aplicar lo anterior a intentar obtener una versión de *QuickSort* de coste *O(N logN)* en el caso peor.

1. Escribir una función
qsort_5(t: np.ndarray)-> np.ndarray
que utilice las funciones anteriores *split_pivot*, *pivot_5* para devolver una ordenación de la tabla *t*.

NOTA: En las funciones anteriores, cuando la tabla tenga 5 o menos elementos, resolver el problema de selección ordenando dicha tabla con el método *np.sort* de *Numpy* y devolviendo el elemento que corresponda.

```python
n = 10
t = np.random.permutation(n).astype(int)
qsort5(t)
```

```
qsort5_nr( t = [7 5 1 9 4 6 3 0 8 2] )
pivot5( [7 5 1 9 4 6 3 0 8 2] )
qsel5_nr( t = [5. 3.] , k = 1 )
5 enc-en-tupla-final: [5. 3.]
pivot5( [7 5 1 9 4 6 3 0 8 2] ) = 5
split [1 4 3 0 2] 5 [7 9 6 8]
qsort5_nr( t = [1 4 3 0 2] )
pivot5( [1 4 3 0 2] )
pivot5( [1 4 3 0 2] ) = 2
split [1 0] 2 [4 3]
qsort5_nr( t = [1 0] )
pivot5( [1 0] )
pivot5( [1 0] ) = 1
split [0] 1 []
qsort5_nr( t = [1 0] ) = [0. 1.]
qsort5_nr( t = [4 3] )
pivot5( [4 3] )
pivot5( [4 3] ) = 4
split [3] 4 []
qsort5_nr( t = [4 3] ) = [3. 4.]
qsort5_nr( t = [1 4 3 0 2] ) = [0. 1. 2. 3. 4.]
qsort5_nr( t = [7 9 6 8] )
pivot5( [7 9 6 8] )
pivot5( [7 9 6 8] ) = 8
split [7 6] 8 [9]
qsort5_nr( t = [7 6] )
pivot5( [7 6] )
pivot5( [7 6] ) = 7
split [6] 7 []
qsort5_nr( t = [7 6] ) = [6. 7.]
qsort5_nr( t = [7 9 6 8] ) = [6. 7. 8. 9.]
qsort5_nr( t = [7 5 1 9 4 6 3 0 8 2] ) = [0. 1. 2. 3. 4. 5. 6. 7. 8. 9.]
array([0., 1., 2., 3., 4., 5., 6., 7., 8., 9.])
```

II-A. Dando cambio

Vamos a implementar el algoritmo de programación dinámica (PD) para hallar el número mínimo de monedas para dar cambio de una cierta cantidad para, a continuación, dar también la combinación óptima de monedas.

1. Escribir una función
change_pd(c: int, l_coins: List[int])-> np.ndarray
que devuelva la matriz generada por el algoritmo PD para obtener el número mínimo de monedas para dar cambio de una cantidad *c* con las monedas de la lista *l_coins* .

2. Escribir una función
optimal_change_pd(c: int, l_coins: List[int])-> Dict
que devuelva un *dict* con claves las monedas de la lista *l_coins* y valores el número de dichas monedas a usar para dar cambio óptimo de la cantidad *c* , haciendo un *backtracking* adecuado sobre la matriz PD para este problema a partir de la posición del valor óptimo.

```
optimal_change_pd( 17 , [1, 2, 5, 10] ):
change_pd( 17 , [1, 2, 5, 10] ):
change_matrix:
 [[ 0.  1.  2.  3.  4.  5.  6.  7.  8.  9. 10. 11. 12. 13. 14. 15. 16. 17.]
 [ 0.  1.  1.  2.  2.  3.  3.  4.  4.  5.  5.  6.  6.  7.  7.  8.  8.  9.]
 [ 0.  1.  1.  2.  2.  1.  2.  2.  3.  3.  2.  3.  3.  4.  4.  3.  4.  4.]
 [ 0.  1.  1.  2.  2.  1.  2.  2.  3.  3.  1.  2.  2.  3.  3.  2.  3.  3.]]
ch_m[ 3 , 17 ] =  3.0
ch_m[ 2 , 17 ] =  4.0 distinct: selected coin: d_ch[ 10 ] =  1 rest = 7
ch_m[ 2 , 7 ] =  2.0
ch_m[ 1 , 7 ] =  4.0 distinct: selected coin: d_ch[ 5 ] =  1 rest = 2
ch_m[ 1 , 2 ] =  1.0
ch_m[ 0 , 2 ] =  2.0 distinct: selected coin: d_ch[ 2 ] =  1 rest = 0
change_dict:
 {1: 0, 2: 1, 5: 1, 10: 1}
```

```
ch_matrix[i] = ch_matrix[i-1]
```

```
ch_matrix[i, cc] = min(ch_matrix[i, cc],
                       1+ch_matrix[i, cc-l_coins[i]])
```

```
c = 17
optimal_change_pd(c, l_coins)
```

```
optimal_change_pd( 17 , [1, 2, 5, 10] ):
change_pd( 17 , [1, 2, 5, 10] ):
change_matrix:
 [[ 0.  1.  2.  3.  4.  5.  6.  7.  8.  9. 10. 11. 12. 13. 14. 15. 16. 17.]
 [ 0.  1.  1.  2.  2.  3.  3.  4.  4.  5.  5.  6.  6.  7.  7.  8.  8.  9.]
 [ 0.  1.  1.  2.  2.  1.  2.  2.  3.  3.  2.  3.  3.  4.  4.  3.  4.  4.]
 [ 0.  1.  1.  2.  2.  1.  2.  2.  3.  3.  1.  2.  2.  3.  3.  2.  3.  3.]]
ch_m[ 3 , 17 ] =  3.0
ch_m[ 2 , 17 ] =  4.0 distinct: selected coin: d_ch[ 10 ] =  1 rest = 7
ch_m[ 2 , 7 ] =  2.0
ch_m[ 1 , 7 ] =  4.0 distinct: selected coin: d_ch[ 5 ] =  1 rest = 2
ch_m[ 1 , 2 ] =  1.0
ch_m[ 0 , 2 ] =  2.0 distinct: selected coin: d_ch[ 2 ] =  1 rest = 0
change_dict:
 {1: 0, 2: 1, 5: 1, 10: 1}
```

```
c = 37
optimal_change_pd(c, l_coins)
```

```
optimal_change_pd( 37 , [1, 2, 5, 10] ):
change_pd( 37 , [1, 2, 5, 10] ):
change_matrix:
 [[ 0.  1.  2.  3.  4.  5.  6.  7.  8.  9. 10. 11. 12. 13. 14. 15. 16. 17.
  18. 19. 20. 21. 22. 23. 24. 25. 26. 27. 28. 29. 30. 31. 32. 33. 34. 35.
  36. 37.]
 [ 0.  1.  1.  2.  2.  3.  3.  4.  4.  5.  5.  6.  6.  7.  7.  8.  8.  9.
   9. 10. 10. 11. 11. 12. 12. 13. 13. 14. 14. 15. 15. 16. 16. 17. 17. 18.
  18. 19.]
 [ 0.  1.  1.  2.  2.  1.  2.  2.  3.  3.  2.  3.  3.  4.  4.  3.  4.  4.
   5.  5.  4.  5.  5.  6.  6.  5.  6.  6.  7.  7.  6.  7.  7.  8.  8.  7.
   8.  8.]
 [ 0.  1.  1.  2.  2.  1.  2.  2.  3.  3.  1.  2.  2.  3.  3.  2.  3.  3.
   4.  4.  2.  3.  3.  4.  4.  3.  4.  4.  5.  5.  3.  4.  4.  5.  5.  4.
   5.  5.]]
ch_m[ 3 , 37 ] =  5.0
ch_m[ 2 , 37 ] =  8.0 distinct: selected coin: d_ch[ 10 ] =  1 rest = 27
ch_m[ 2 , 27 ] =  6.0 distinct: selected coin: d_ch[ 10 ] =  2 rest = 17
ch_m[ 2 , 17 ] =  4.0 distinct: selected coin: d_ch[ 10 ] =  3 rest = 7
ch_m[ 2 , 7 ] =  2.0
ch_m[ 1 , 7 ] =  4.0 distinct: selected coin: d_ch[ 5 ] =  1 rest = 2
ch_m[ 1 , 2 ] =  1.0
ch_m[ 0 , 2 ] =  2.0 distinct: selected coin: d_ch[ 2 ] =  1 rest = 0
change_dict:
 {1: 0, 2: 1, 5: 1, 10: 3}
```

NOTE: The value of the coin is subtracted as many times as possible (that is, as long as the amount to be exchanged is not exceeded using this coin).

II-B. El problema de la mochila

Vamos a implementar tanto el algoritmo codicioso como el de programación dinámica (PD) para resolver el problema de la mochila 0-1.

1. Escribir una función
knapsack_fract_greedy(l_weights: List[int], l_values: List[int], bound: int)-> int
que devuelva un *dict* con los pesos a tomar de cada elemento en la solución *greedy* del problema de la mochila
fraccionaria. (Usar para ello los enteros 0, 1, ... , como claves del *dict* .)

2. Escribir una función
knapsack_01_pd(l_weights: List[int], l_values: List[int], bound: int)-> int
que devuelva el valor óptimo de la mochila 0-1 obtenida mediante PD.

```python
l_weights = [4, 3, 5, 2]
l_values = [10, 40, 30, 20]
bound = 8

d_v_w = knapsack_fract_greedy(l_weights, l_values, bound=bound)
print("d_v_w =", d_v_w)
print("val_greedy knapsack fract:", val_greedy(l_weights, l_values, d_v_w))

d_v_w = knapsack_01_greedy(l_weights, l_values, bound=bound)
print("d_v_w =", d_v_w)
print("val_greedy knapsack 01:", val_greedy(l_weights, l_values, d_v_w))

d_w_v = optimal_knapsack_pd(l_weights, l_values, bound=bound)
print("val_pd knapsack pd:", val_pd(l_weights, l_values, d_w_v))
```

```
knapsack_fract_greedy:
relative values (values/weights): [ 2.5        13.33333333  6.         10.        ]
selection order by relative values: [1 3 2 0]
selected item 1 of weight 3 , remaining weight 5.0
selected item 3 of weight 2 , remaining weight 3.0
selected item 2 of weight 5 , in fraction 0.6 of weight 3.0 , remaining weight 0
not selected item 0 of weight 4 , remaining weight 0
d_v_w = {1: 3, 3: 2, 2: 3.0, 0: 0}
val_greedy knapsack fract: 78.0

knapsack_01_greedy:
relative values (values/weights): [ 2.5        13.33333333  6.         10.        ]
selection order by relative values: [1 3 2 0]
selected item 1 of weight 3 , remaining weight 5
selected item 3 of weight 2 , remaining weight 3
not selected item 2 of weight 5 , remaining weight 3
not selected item 0 of weight 4 , remaining weight 3
d_v_w = {1: 3, 3: 2, 2: 0, 0: 0}
val_greedy knapsack 01: 60.0
```

```
optimal_knapsack_pd( [4, 3, 5, 2] [10, 40, 30, 20] 8 ):
knapsack_01_pd:
0 4 10
1 3 40
2 5 30
3 2 20
```

v_matrix[i] = v_matrix[i-1]

```
v_matrix[i, b] = max(v_matrix[i, b],
                     l_values[i-1] + v_matrix[i-1, b-l_weights[i-1]])
```

Items
Weight
|
0: 4
1: 3
2: 5
3: 2

```
m:  0   1   2   3   4   5   6   7   8   <= Knapsack Weight
[ 0.  0.  0.  0.  0.  0.  0.  0.  0.]
[ 0.  0.  0.  0. 10. 10. 10. 10. 10.]
[ 0.  0.  0. 40. 40. 40. 40. 50. 50.]   <= Values (Benefit)
[ 0.  0.  0. 40. 40. 40. 40. 50. 70.]
[ 0.  0. 20. 40. 40. 60. 60. 60. 70.]]
m[ 4 , 8 ] =  70.0
m[ 3 , 8 ] =  70.0
m[ 2 , 8 ] =  50.0 distinct, selected item: d[ 5 ] =  1
m[ 2 , 3 ] =  40.0
m[ 1 , 3 ] =  0.0 distinct, selected item: d[ 3 ] =  1
m[ 1 , 0 ] =  0.0
d:
 {4: 0, 3: 1, 5: 1, 2: 0}
val_pd knapsack pd: 70
```

https://www.youtube.com/watch?v=IZHvQTx2bZ0