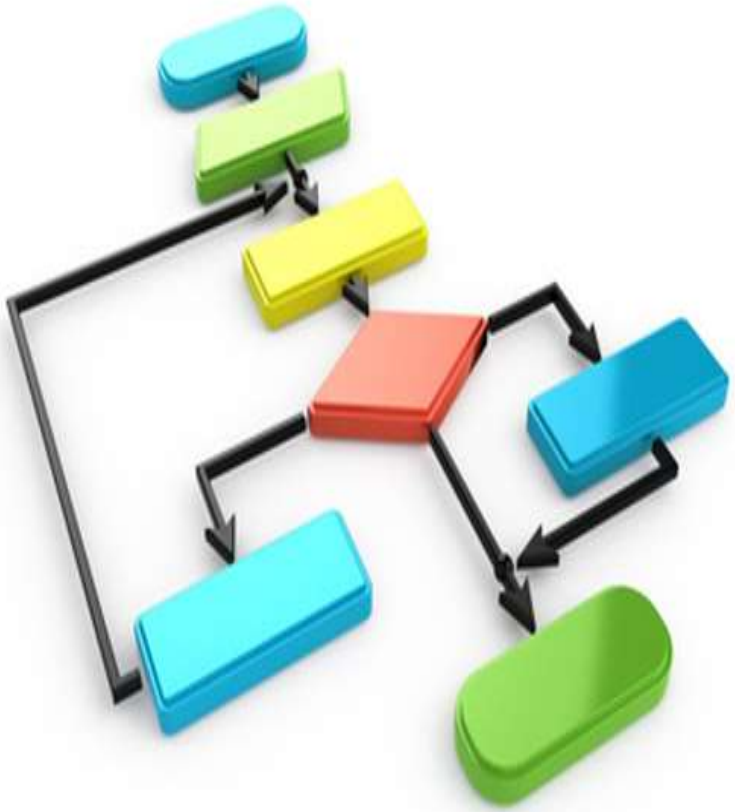
Práctica 1



ALGORITMIA Y ESTRUCTURAS DE DATOS AVANZADAS

Índice

00 Requisitos de entrega

Requisitos de entrega

01 Material a entregar

02 Corrección

01 Material a entregar

Crear una carpeta con el nombre p1NN donde **NN** indica el número de pareja y añadir en ella solo los siguientes archivos:

1. Un archivo Python p1NN.py con el código de las funciones desarrolladas en la práctica (no los scripts de medida de tiempo o dibujo de curvas) así como los imports estrictamente necesarios, a saber:

```
import numpy as np
from typing import List, Callable
```

Los nombres y parámetros de las funciones definidas en ellos deben ajustarse EXACTAMENTE a los utilizados en este documento.

Además, todas las definiciones de funciones deben incorporar los type hints adecuados.

01 Material a entregar

2. Un fichero p1NN.html con el resultado de aplicar el comando pdoc del paquete pdoc3 al módulo Python p1NN.py .

3. Un archivo p1NN.pdf con una breve memoria que contenga las respuestas a las cuestiones de la práctica en formato pdf.

En la memoria se identificará claramente el nombre de los estudiantes y el número de pareja. Si se añaden figuras o gráficos, DEBEN ESTAR SOBRE UN FONDO BLANCO.

Una vez que se hayan puesto estos archivos, comprimir esta carpeta en un archivo llamado p1NN.zip o llamado p1NN.7z .

No añadir ninguna estructura de subdirectorios a la carpeta.

La práctica no se corregirá hasta que el envío siga esta estructura.

02 Corrección

La corrección se hará en base a las siguientes tareas:

- La ejecución de un script que importará el módulo p1NN.py y comprobará la corrección de su código.

  La práctica no se corregirá mientras este script no se ejecute correctamente, penalizando las segundas presentaciones.

- La revisión de la documentación del código contenida en los ficheros html generados por pdoc. En particular, las docstrings deben ser escritas muy cuidadosamente.

  Además, el código Python debe estar formateado según el estándar PEP-8. Utilizar para ello un formateador como autopep8 o black.

- La revisión de una selección de las funciones de Python contenidas en el módulo p1NN.py .

- La revisión de la memoria con las respuestas a las cuestiones planteadas.

01 Python básico

Python básico

01 Midiendo tiempos con *%timeit*

02 Búsqueda binaria

03 Cuestiones

01 Midiendo tiempos con *%timeit*

Vamos a usar la orden mágica %timeit de IPython para medir tiempos de ejecución de pequeñas funciones, en nuestro caso, de multiplicación de matrices.

1. Escribir una función matrix_multiplication(m_1: np.ndarray, m_2: np.ndarray)-> np.ndarray que reciba dos matrices Numpy de dimensiones compatibles y devuelva otra matriz Numpy con su producto.

2. Usar el pequeño script inferior para medir los tiempos de ejecución de la función anterior sobre matrices aleatorias de dimensión creciente y guardarlos en una lista para poder analizarlos o procesarlos luego:

```
l_timings = []
for i in range(11):
    dim = 10+i
    m = ... np.random.uniform(0., 100, (dim, dim))
    timings = %timeit -o -n 10 -r 5 -q matrix_multiplication(m, m)
    l_timings.append([dim, timings.best])
```

Para ello cambiar los puntos suspensivos por una llamada adecuada al generador numpy uniform de números aleatorios uniformes que devuelva matrices aleatorias de dimensión dim x dim con valores aleatorios uniformes entre 0. y 1.

02 Búsqueda binaria

Vamos a programar y medir tiempos de ejecución de funciones recursivas e iterativas de búsqueda binaria.

1. Escribir una función rec_bb(t: List, f: int, l: int, key: int)-> int que reciba una lista, sus índices first, last primero y último, y una clave key , y aplique una versión recursiva de la búsqueda binaria para encontrar la posición de key entre first y last . Si no la encuentra, devolverá None .

2. Escribir una versión iterativa función bb(t: List, f: int, l: int, key: int)-> int de la función anterior con las mismas especificaciones.

3. Adecuar el script inferior para medir los tiempos de ejecución de las funciones anteriores, escogiendo valores adecuados de key para que el tiempo de ejecución sea máximo.

```
l_times = []
for i, size in enumerate(range(5, 15)):
    t = list(range(2**i * size))
    key = ...
    timings = %timeit -n 100 -r 10 -o -q ...(t, 0, len(t) - 1, key)
    l_times.append([len(t), timings.best])

times = np.array(l_times)
```

03 Cuestiones

```
from scipy.optimize import curve_fit

def tofit(x, a, b):
    #cambiar f por el ócdigo Python de la ófuncin de
    return a *f(x) + b

x = a_timings[: , 0]
#o (mas "pythonico"?) x = a_timings.T[0].T
y = np.array(a_timings)[: , 1]
y = y / y[0]    #normalizar timings

pars, _ = curve_fit(tofit, x, y)
```

1. ¿A qué función f se deberían ajustar los tiempos de multiplicación de la función de multiplicación de matrices? Usar este código para ajustar valores de la forma a*f(t)+b a los tiempos de ejecución de la función que hayamos guardado en el array Numpy timings , para a continuación dibujar tanto los tiempos como el ajuste calculado por la función, y comentar los resultados.

2. Calcular los tiempos de ejecución que se obtendrían usando la multiplicación de matrices a.dot(b) de Numpy y compararlos con los anteriores.

3. ¿Qué clave resultaría en el caso más costoso de la búsqueda binaria? Comparar los tiempos de ejecución de las versiones recursiva e iterativa de la búsqueda binaria en su caso más costoso y dibujarlos para unos tamaños de tabla adecuados. ¿Qué relación encuentras entre ambos tiempos? Argumentar gráficamente dicha relación.

02 Min Heaps

Min Heaps

01 Min Heaps sobre arrays de Numpy

02 Colas de prioridad sobre Min Heaps

03 Ordenando con Min Heaps

04 Cuestiones

01 Min Heaps sobre arrays de Numpy

Vamos a implementar sobre arrays de Numpy diversas funciones de trabajo con Min Heaps. Para ello supondremos que un min heap vacío se representa como None .

1. Escribir una función min_heapify(h: np.ndarray, i: int) que reciba un array h de Numpy y aplique la operación de heapify al elemento situado en la posición i .

2. Escribir una función insert_min_heap(h: np.ndarray, k: int)-> np.ndarray que inserte el entero k en el min heap contenido en h y devuelva el nuevo min heap.

3. Escribir una función create_min_heap(h: np.ndarray) que cree un min heap sobre el array de h de Numpy pasado como argumento.

02 Colas de prioridad sobre Min Heaps

Vamos a usar las funciones anteriores sobre min heaps para programar las primitivas de colas de prioridad suponiendo que las mismas son arrays de enteros en Numpy y donde el valor de cada elemento coincide con su prioridad. Suponemos también que los elementos con un valor de prioridad menor salen antes que los que tienen prioridad mayor.

1. Escribir una función pq_ini() que inicialice una cola de prioridad vacía.
2. Escribir una función pq_insert(h: np.ndarray, k: int)-> np.ndarray que inserte el elemento k en la cola de prioridad h y devuelva la nueva cola.
3. Escribir una función pq_remove(h: np.ndarray)-> np.ndarray que elimine el elemento con el menor valor de prioridad de h y devuelva dicho elemento y la nueva cola.

02 Ordenando con Min Heaps

Sabemos que en un Min Heap el elemento más pequeño está en su raíz. Esto sugiere el siguiente procedimiento para ordenar un array h:

1. Crear un min heap sobre h de manera in place.
2. Repetidamente:
    a) Extraer (por ejemplo, a una lista) el elemento en la raíz.
    b) Copiar el último elemento en ese momento del array que contiene el min heap a su primera posición.
    c) Rehacer el min heap aplicando heapify desde la raíz y sobre un array reducido donde se ignora el anterior último elemento.

Esto es, al aplicar heapify hay que ir reduciendo la parte del array que se considera. Por ejemplo, si en un momento dado el array h tiene k elementos, la llamada a min_heapify tras extraer la raíz y poner el último elemento en la raíz debe ser min_heapify(h[: k], 0) , y así sucesivamente.

1. Escribir una función min_heap_sort(h: np.ndarray)-> np.ndarray que implemente la idea anterior devolviendo un array con una ordenación del array inicial h.

03 Cuestiones

1. Analizar visualmente los tiempos de ejecución de nuestra función de creación de min heaps. Establecer razonadamente a qué función f se deberían ajustar dichos tiempos.

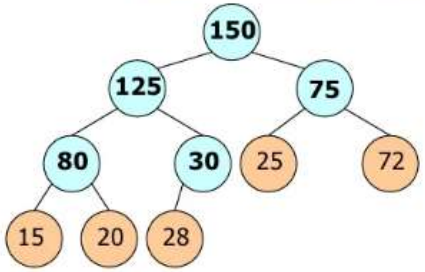
2. Dar razonadamente cuál debería ser el coste de nuestra función de ordenación mediante Min Heaps en función del tamaño del array.

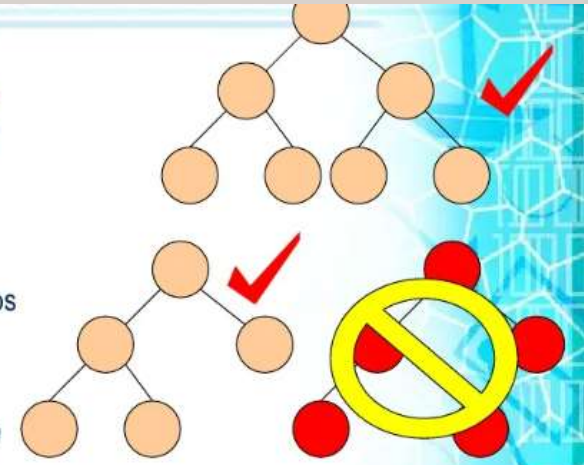
3. Analizar visualmente los tiempos de ejecución de nuestra función de ordenación mediante Min Heaps y comprobar que los mismos se ajustan a la función de coste del punto anterior.

03 Anexo Min Heaps

CONCEPTOS

- Arbol Binario Completo
  - Todos sus niveles estan completos
  - A excepción del ultimo nivel,
    - *Alli las hojas van apareciendo seguidas de izquierda a derecha*





- Arbol Parcialmente Ordenado
  - Es Binario Completo
  - Con propiedad de orden

Entre raiz e hijos, la raiz contiene el mayor(o menor) de todos

UTILIDAD DE UN HEAP

- Si el mayor valor esta siempre en la raiz
  - El heap presenta un cierto orden

    Al remover consecutivamente la raiz
    - *Vamos consiguiendo valores* ***ordenados***
- El heap se utiliza
  - Para el ordenamiento de elementos(HeapSort)
  - Para implementar colas de prioridad

QDesencolarMax, es retirar el valor de la raiz

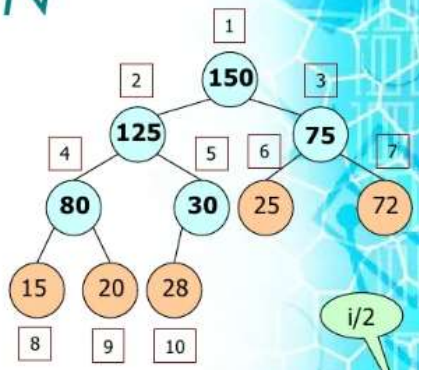
03 Anexo sobre Min Heaps

IMPLEMENTACION

- Un heap no admite "huecos",
  - C/nivel se va llenando de izq. A der
  - Hay una secuencia
- Podriamos numerar c/nodo
  - En el orden de llenado
- Si lo vemos asi, dado un índice

  Podemos conocer los indices de los hijos y el padre de un nodo

  Ejemplo: Del nodo 4, hijos 8 y 9, padre 2



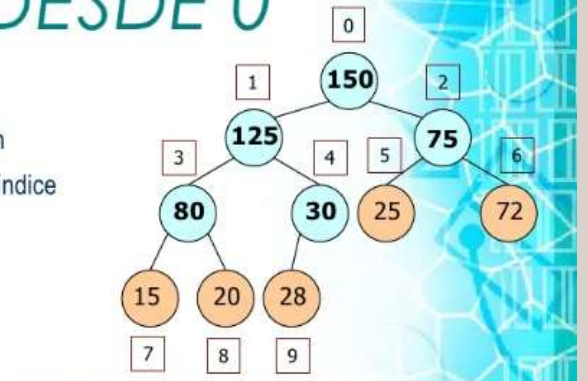
i	izq(i)	der(i)	padre(i)
1	2	3	-
2	4	5	1
3	6	7	1
4	8	9	2
5	10	-	2
6	-	-	3

CONTANDO DESDE 0

- Queremos usar un vector

  En c/elemento almacenar la información

  - Dirigirnos a hijos y padre calculando el índice respectivo
    - **Izq(i) = i*2**
    - **Der(i) = i*2+1**
    - **Padre(i) = i/2**
- Los vectores en C, empiezan desde 0
  - Cambia un poquito la regla
    - **Izq(i)** = (i+1)*2-1 = **i*2+1**
    - **Der(i)** = (i+1)*2 = **i*2+2**
    - **Padre(i)** = (i+1)/2-1 = **(i-1)/2**



150	125	75	80	30	25	72	15	20	28
0	1	2	3	4	5	6	7	8	9

03 Anexo sobre Min Heaps

REGLAS

- Vector V de tamaño efectivo n
  - V[0] es la raiz
  - Dado un nodo V[i]
    - *Si 2i+1 < n, V[2i+1] es el hijo izq*
    - *Si 2i+2 < n, V[2i+2] es el hijo der*
    - *Si i != 0, v[(i-1)/2] es el padre*
  - Si es heap

$$\forall i : 1 \le i < n : v[(i-1)/2] \ge v[i]$$

AJUSTAR
(min_heapify)

- Recobra la propiedad de orden
- Desde un nodo de índice pos

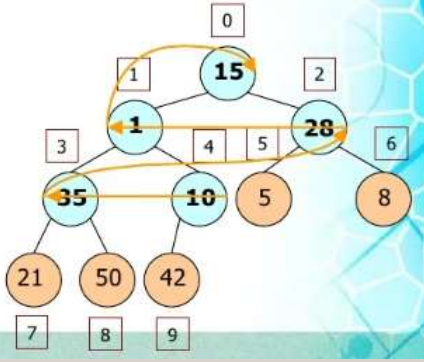
  Dado un índice pos, PosIzq y PosDer
  - Se compararan los tres para ver quien tiene el mayor
  - Si el mayor lo tiene algun hijo
    - *Intercambia*
    - *Al hacer esto, el sub-heap afectado puede perder su propiedad de Orden....*
    - *Ajustar el sub-heap afectado*

03 Anexo sobre Min Heaps

CONSTRUIR UN HEAP
(create_min_heap)

- La estructura de un heap
  - Puede almacenar un arreglo que no cumpla las propiedades de orden
- Ejemplo:

15	1	28	35	10	5	8	21	50	42
0	1	2	3	4	5	6	7	8	9

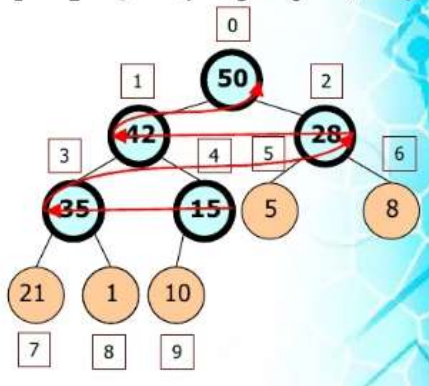


- Hay que arreglar la propiedad de orden
  - De cada raiz
  - Ajustar c/raiz
  - Desde la ultima a la primera

CONSTRUIR HEAP: EJEMPLO

50	42	28	35	15	5	8	21	1	10
0	1	2	3	4	5	6	7	8	9

- Ajustar el ultimo nodo raiz
  - Los nodos raiz comienzan desde 0 hasta n/2-1
- Al ajustar cada nodo
  - De atrás hacia delante
  - Nos aseguramos que los valores mas altos suban!



03 Anexo sobre Min Heaps

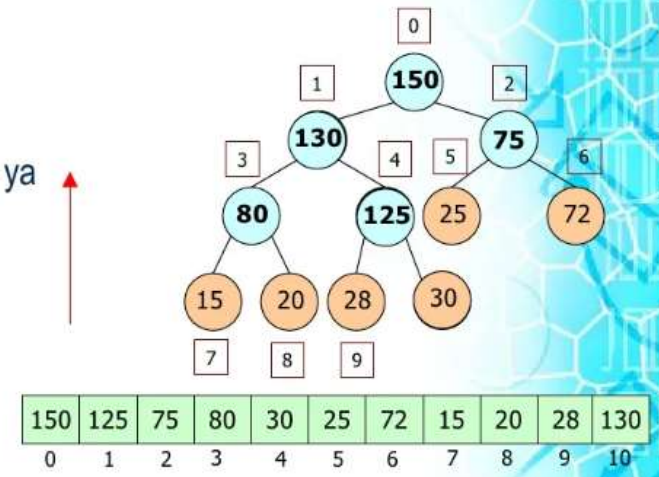
ENCOLAR
(insert_min_heap)

- U Al añadir un nuevo elemento el Heap
  - v DEBE conservar su propiedad de orden
- u Se añade al final del arreglo
- l El elemento empieza a subir a su posición ideal
  - e Comparando siempre con el padre
  - r Hasta que el valor insertado sea menor que el del padre

ENCOLAR: EJEMPLO

- Insertar al final
- l Subir el valor hasta que ya no sea necesario

Nuevo valor: 130



03 Anexo sobre Min Heaps

DESENCOLAR

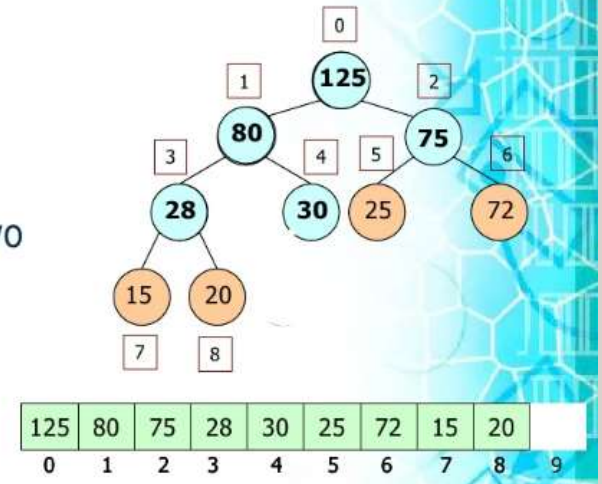
- Que importancia tiene la raiz en el heap?
  - t Es el mayor/menor elemento del mismo
  - r Sobre el resto de elementos no estamos seguros
  - e Pero de la raiz, es la mayor de todos

Desencolar es eliminar la raiz

- i Que valor se ubica en la nueva raiz?
- c El ultimo reemplaza al primero
- a El tamaño efectivo del heap cambia
- v Se ajusta desde la raiz
- l El arbol queda bien otra vez

DESENCOLAR: EJEMPLO

- P Intercambiar valores
  - o Raiz con ultimo
- o Aminorar tamaño efectivo
- e Ajustar arbol



03 Anexo sobre Min Heaps

HEAPSORT

- Uno de los usos de los heaps es ordenar
- Como?
  - Extraer el mayor elemento del heap(raiz)
    Ponerla al final de otro arreglo
    Repetir los dos ultimos pasos hasta que el Heap quede Vacio
- El arreglo quedara ordenado

HEAPSORT: EJEMPLO

- Desencolar y ponerlo al final del nuevo arreglo/lista
- Repetir hasta que el arbol quede vacio