Ficha 22

Ordenamiento

1.] Algoritmos de ordenamiento. Clasificación tradicional.

Es claro que pueden existir muchos algoritmos diferentes para resolver el mismo problema, y el problema de la *ordenación de un arreglo* es quizás el caso emblemático de esa situación. Podemos afirmar que existen varias docenas de algoritmos diferentes para lograr que el contenido de un arreglo se modifique para dejarlo ordenado de menor a mayor, o de mayor a menor. Muchos de esos algoritmos están basados en ideas intuitivas muy simples, y otros se fundamentan en estrategias lógicas muy sutiles y no tan obvias a primera vista. En general, se suele llamar *algoritmo simples* o *algoritmos directos* a los del primer grupo, y *algoritmos compuestos* o *algoritmos mejorados* a los del segundo, aunque la diferencia real entre ambos no es sólo conceptual, sino que efectivamente existe una diferencia de rendimiento muy marcada en cuanto al tiempo que los algoritmos de cada grupo demoran en terminar de ordenar el arreglo [1]. Más adelante profundizaremos la cuestión del *análisis comparativo* del rendimiento de algoritmos, y justificaremos formalmente la diferencia entre los dos grupos, pero por ahora nos concentraremos sólo en la clasificación de los algoritmos y el funcionamiento de aquellos que estén al alcance de este curso.

Tradicionalmente, como dijimos, los algoritmos de ordenamiento se suelen clasificar en los dos grupos ya citados, y en cada grupo se encuentran los siguientes (aunque entienda: la tabla siguiente es sólo una forma clásica de presentar a los algoritmos más conocidos, pero estos no son todos los algoritmos que existen… que son varias docenas…) [1]:

Figura 1: Clasificación tradicional de los algoritmos de ordenamiento más comunes.

Algoritmos Simples o Directos	Algoritmos Compuestos o Mejorados
Intercambio Directo (Burbuja)	Método de Ordenamiento Rápido (*Quicksort*) [C. Hoare - 1960]
Selección Directa	Ordenamiento de Montículo (*Heapsort*) [J. Williams – 1964]
Inserción Directa	Ordenamiento por Incrementos Decrecientes (*Shellsort*) [D. Shell – 1959]

En principio, los algoritmos clasificados como *Simples* o *Directos* son algoritmos sencillos e intuitivos, pero de mal rendimiento si la cantidad *n* de elementos del arreglo es grande o muy grande, o incluso si lo que se desea es ordenar un archivo en disco. Aquí, *mal rendimiento* por ahora significa "demasiada demora esperada". En cambio, los métodos presentados como *Compuestos* o *Mejorados* tienen un muy buen rendimiento en comparación con los simples, y se aconsejan toda vez que el arreglo sea muy grande o se quiera ordenar un conjunto en memoria externa. Si el tamaño *n* del arreglo es pequeño (por ejemplo, no más de 100 o 150 elementos), los métodos simples siguen siendo una buena elección por cuanto su escasa eficiencia no es notable con conjuntos pequeños. Note que la idea final, es que cada algoritmo compuesto mostrado en esta tabla representa en realidad una mejora en el planteo del algoritmo simple de la misma fila, y por ello se los llama "algoritmos mejorados". Así, el algoritmo *Quicksort* es un replanteo del algoritmo de *intercambio directo* (más conocido como *ordenamiento de burbuja*) para eliminar los

elementos que lo hacen ineficiente. Paradójicamente, el *ordenamiento de burbuja* o *bubblesort* es en general el de *peor rendimiento* para ordenar un arreglo, pero ha dado lugar al *Quicksort*, que en general es el de *mejor rendimiento promedio* conocido.

2.] Funcionamiento de los Algoritmos de Ordenamiento Simples o Directos.

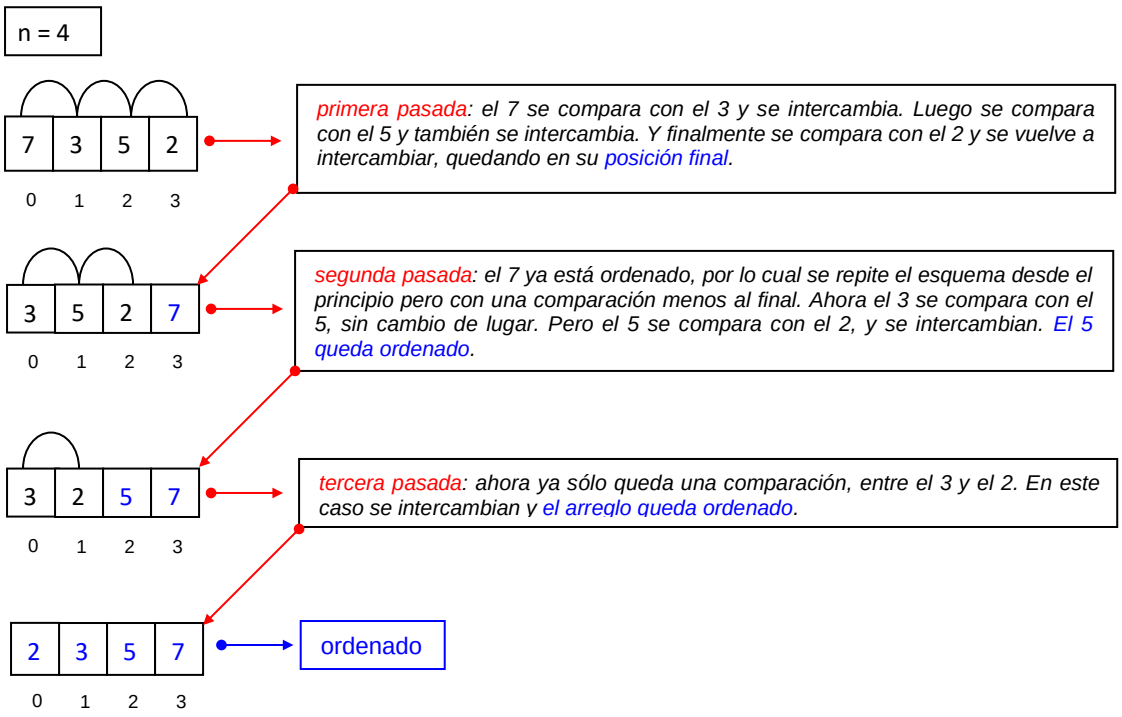
Haremos aquí una breve descripción operativa de las estrategias de funcionamiento de los algoritmos directos conocidos como *Intercambio Directo (o Burbuja o Bubblesort)* e *Inserción Directa*. Para una descripción del algoritmo de *Selección Directa*, revise la *Ficha 16.* Más adelante, en una Ficha posterior, mostraremos un análisis de rendimiento formal basado en calcular la cantidad de comparaciones que estos algoritmos realizan.

En todos los casos, suponemos un arreglo *v* de *n* elementos, y también suponemos que se pretende ordenar de menor a mayor. Hemos incluido un modelo llamado *test01.py* en el proyecto *[F22] Ordenamiento*, que acompaña a esta Ficha. En el mismo proyecto se incluye el módulo *ordenamiento.py*, que contiene todas las funciones que implementan estos algoritmos.

a.) Ordenamiento por *Intercambio Directo* (o *Bubblesort*):

La idea esencial del algoritmo es que cada elemento en cada casilla *v[i]* se compara con el elemento en *v[i+1]*. Si este último es menor, se intercambian los contenidos. Se usan dos ciclos anidados para conseguir que incluso los elementos pequeños ubicados muy atrás, puedan en algún momento llegar a sus posiciones al frente del arreglo. Gráficamente, supongamos que el tamaño del arreglo es *n = 4*. El algoritmo procede como se muestra en la *Figura 2*.

Figura 2: Funcionamiento general del *Ordenamiento por Intercambio Directo* (*Bubblesort*).



Puede verse que si el arreglo tiene *n* elementos, serán necesarias a lo sumo *n-1* pasadas para terminar de ordenarlo en el peor caso, y que en cada pasada se hace

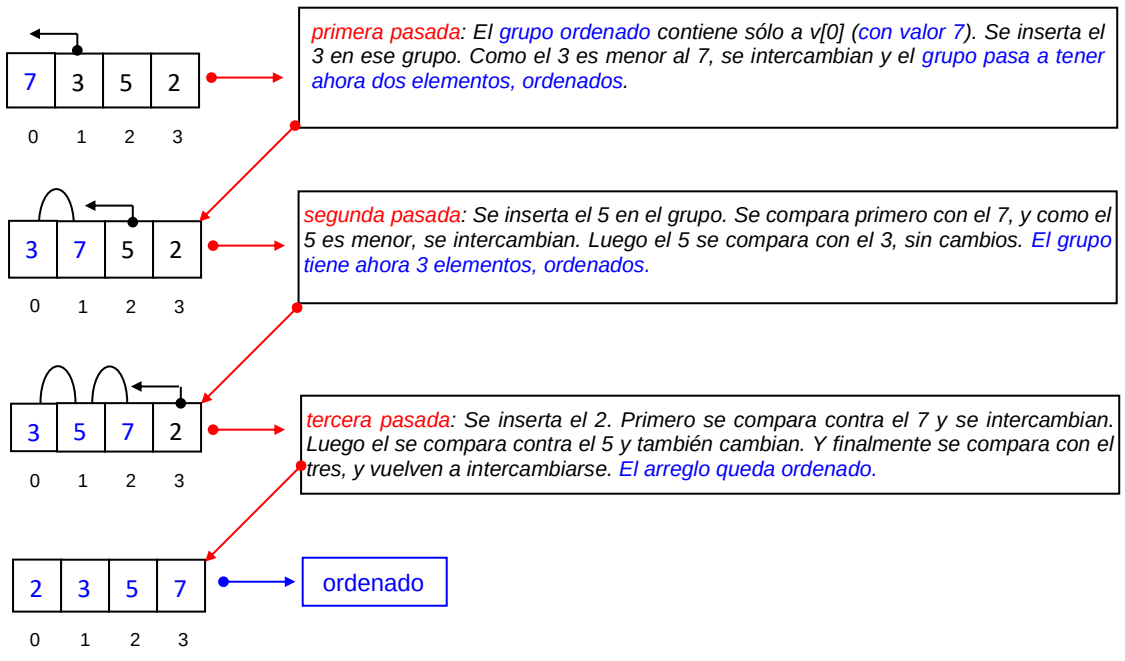
cada vez una comparación menos que en la anterior. Otro hecho notable es que el arreglo podría quedar ordenado *antes de la última pasada*. Por ejemplo, si el arreglo original hubiese sido [2 – 3 – 7 – 5], entonces en la primera pasada el 7 cambiaría con el 5 y dejaría al arreglo ordenado. Para detectar ese tipo de situaciones, en el algoritmo se agrega una variable a modo de *bandera de corte*: si se detecta que en una pasada no hubo ningún intercambio, el ciclo que controla la cantidad de pasadas se interrumpe antes de llegar a la pasada *n-1* y el ordenamiento se da por concluido. Se ha denominado *ordenamiento de burbuja* porque los elementos parecen *burbujear* en el arreglo a medida que se ordena... (☹)... La siguiente función del módulo *ordenamiento.py* (proyecto *[F22] Ordenamiento*) lo implementa:

```
def bubble_sort(v):
    n = len(v)
    for i in range(n-1):
        ordenado = True
        for j in range(n-i-1):
            if v[j] > v[j+1]:
                ordenado = False
                v[j], v[j+1] = v[j+1], v[j]
        if ordenado:
            break
```

b.) Ordenamiento por *Inserción Directa* (o *Inserción Simple*):

La idea es ahora distinta y más original. Se comienza suponiendo que el valor en la casilla *v[0]* conforma un subconjunto. Y como tiene un solo elemento, está ordenado. A ese subconjunto lo llamamos un *grupo ordenado* dentro del arreglo. Se toma *v[1]* y se trata de insertar su valor en el grupo ordenado. Si es menor que *v[0]* se intercambian, y si no, se dejan como estaban. En ambos casos, el grupo tiene ahora dos elementos y sigue ordenado. Se toma *v[2]* y se procede igual, comenzando la comparación contra *v[1]* (que es el mayor del grupo). Así, también hacen falta *n-1* pasadas, de forma que en cada pasada se inserta un nuevo valor al grupo. El algoritmo procede como se muestra en la *Figura 3*:

Figura 3: Funcionamiento general del Algoritmo de *Ordenamiento por Inserción Directa*.



La función que sigue (ver módulo *ordenamiento.py* del proyecto *[F22] Ordenamiento*) lo implementa:

```
def insertion_sort(v):
    n = len(v)
    for j in range(1, n):
        y = v[j]
        k = j - 1
        while k >= 0 and y < v[k]:
            v[k+1] = v[k]
            k -= 1
        v[k+1] = y
```

3.] Algoritmos de Ordenamiento Compuestos o Mejorados: El Algoritmo Shellsort.

Como dijimos, los algoritmos designados como *Compuestos* o *Mejorados* están basados en la idea de mejorar algunos aspectos que hacen que los algoritmos directos no tengan buen rendimiento cuando el tamaño del arreglo es grande o muy grande. De los tres algoritmos que hemos mencionado en la tabla de la *Figura 1* (*Quicksort*, *Heapsort* y *Shellsort*), analizaremos en esta sección sólo el mecanismo de funcionamiento de trabajo del *Shellsort*, que es el más simple en cuanto a su estructura y fundamentos.

El algoritmo *Quicksort* aplica una estrategia recursiva conocida como *Divide y Vencerás* y será analizado con detalle cuando se exponga esa estrategia en una Ficha posterior.

Finalmente, el algoritmo *Heapsort* se basa en el uso de una estructura de datos conocida como *Heap* (o *Grupo de Ordenamiento*) que escapa a los contenidos mínimos de este curso. No obstante, por si algún estudiante siente curiosidad por la forma general de funcionamiento de este algoritmo, lo hemos incluido en esta Ficha en forma de tema ***totalmente opcional*** (no será evaluado ni exigido en ninguna de las evaluaciones de ningún tipo previstas para esta asignatura, aunque alguna pregunta podría surgir en el Cuestionario asociado a esta Ficha).

De nuevo, suponemos en todos los casos un arreglo *v* de *n* elementos, y ordenamiento de menor a mayor.

a.) ***Ordenamiento de Shell* (*Shellsort* u *Ordenamiento por Incrementos Decrecientes*):**

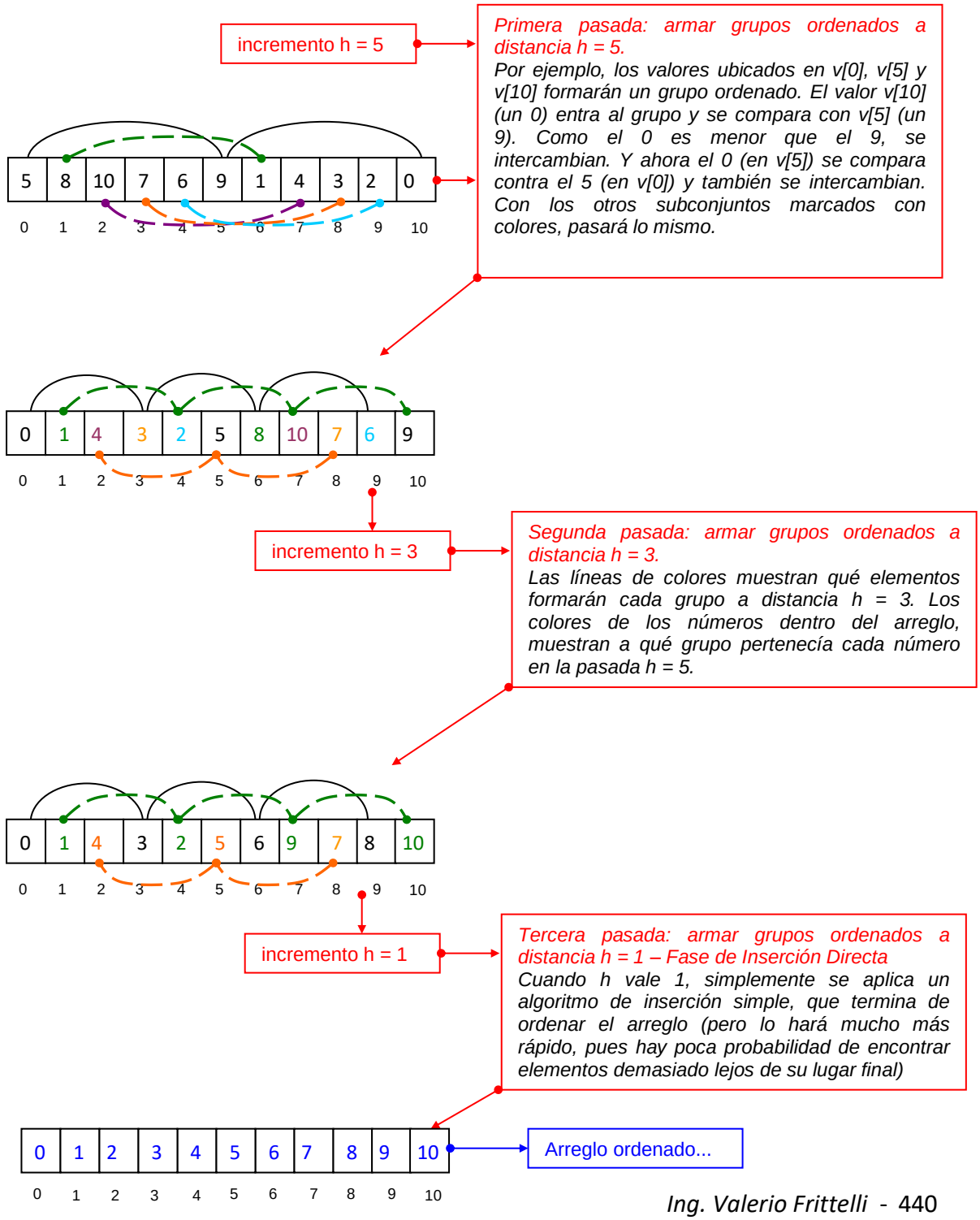
El algoritmo de ordenamiento por *Inserción Directa* es el más rápido de los métodos simples, pues aprovecha que un subconjunto del arreglo está ya ordenado y simplemente inserta nuevos valores en ese conjunto de forma que el subconjunto siga ordenado. El problema es que si llega a aparecer un elemento muy pequeño en el extremo derecho del arreglo, cuando el grupo ordenado de la izquierda ya contiene a casi todo el vector, la inserción de ese valor pequeño demorará demasiado, pues tendrá que compararse con casi todo el arreglo para llegar a su lugar final.

En 1959 un técnico de la *General Electric Company* llamado *Donald Shell*, publicó un algoritmo que mejoraba al de inserción directa y lo llamó *High-Speed Sorting Procedure*, aunque en honor a su creador se lo terminó llamando *Shellsort*. La idea es lanzar un proceso de ordenamiento por inserción, pero en lugar de hacer que cada valor que entra al grupo se compare con su vecino inmediato a la izquierda, se comience haciendo primero un reacomodamiento de forma que cada elemento del arreglo se compare contra elementos ubicados más lejos, a distancias mayores que

uno, y se intercambien elementos a esas distancias [2]. Luego, en pasadas sucesivas, las *distancias de comparación* se van acortando y repitiendo el proceso con elementos cada vez más cercanos unos a otros. De esta forma, se van armando grupos ordenados pero no a distancia uno, sino a distancia *h* tal que *h > 1*.

Finalmente, se termina tomando una distancia de comparación igual a uno, y en ese momento el algoritmo se convierte lisa y llanamente en una *Inserción Directa* para terminar de ordenar el arreglo. Las distancias de comparación se denominan en general *incrementos decrecientes*, y de allí el nombre con que también se conoce al método [2]. En la *Figura 4* mostramos la idea con un pequeño arreglo, suponiendo incrementos decrecientes de la forma [5 – 3 – 1].

Figura 4: Esquema general de funcionamiento del Algoritmo *Shellsort*.



No es simple elegir los valores de los incrementos decrecientes, y de esa elección depende muy fuertemente el rendimiento del algoritmo. En general, digamos que no es necesario que esos incrementos sean demasiados: suele bastar con una cantidad de distancias igual o menor al 10% del tamaño del arreglo, pero *debe asegurarse siempre que la última sea igual uno*, pues de lo contrario no hay garantía que el arreglo quede ordenado. Por otra parte, es de esperar que los valores elegidos como distancias de comparación *no sean todos múltiplos entre ellos*, pues si así fuera se estarían comparando siempre las mismas subsecuencias de elementos, sin mezclar nunca esas subsecuencias. Sin embargo, no es necesario que los valores de los incrementos decrecientes sean todos necesariamente primos. Es suficiente con garantizar valores que no sean todos múltiplos entre sí.

La función que sigue implementa el *algoritmo de Shell* [2]. Está incluida en el módulo *ordenamiento.py* del proyecto *[F22] Ordenamiento*.

```
def shell_sort(v):
    n = len(v)
    h = 1
    while h <= n // 9:
        h = 3*h + 1

    while h > 0:
        for j in range(h, n):
            y = v[j]
            k = j - h
            while k >= 0 and y < v[k]:
                v[k+h] = v[k]
                k -= h
            v[k+h] = y
        h //= 3
```

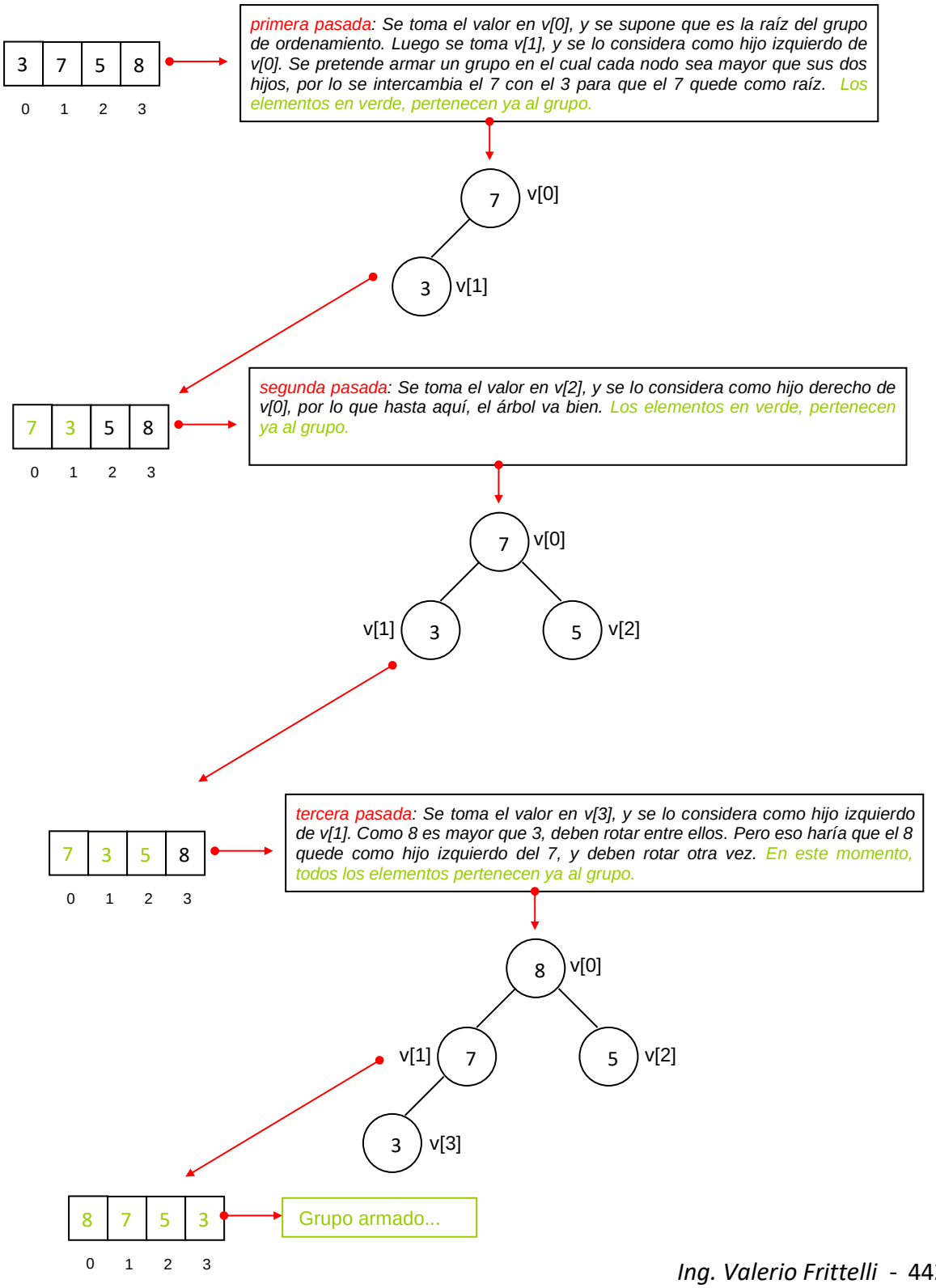
b.) ***El Algoritmo Heapsort.***

El método de ordenamiento por Selección Directa realiza n-1 pasadas, y el objetivo de cada una es seleccionar el menor de los elementos que sigan sin ordenar en el arreglo y trasladarlo a la casilla designada como pivot. El problema es que la búsqueda del menor en cada pasada se basa en un proceso secuencial, y exige demasiadas comparaciones. En 1964, un estudiante de Ciencias de la Computación, llamado J. Williams, publicó una mejora para el algoritmo de Selección Directa en Communicatons of the ACM, y llamó al mismo *Ordenamiento de Montículos* o *Heapsort* (en realidad, debería traducirse como Ordenamiento de Grupos de Ordenamiento, pero queda redundante…) Ese algoritmo reduce de manera drástica el tiempo de búsqueda o selección del menor en cada pasada, usando una estructura de datos conocida como *grupo de ordenamiento* (que no es otra cosa que una *cola de prioridades*, pero optimizada en velocidad: los elementos se insertan rápidamente, ordenados de alguna forma, pero cuando se extrae alguno, se extrae el menor [o el mayor, según las necesidades]) Igual que antes, al seleccionar el menor (o el mayor) se lo lleva a su lugar final del arreglo. Luego se extrae el siguiente menor (o mayor), se lo reubica, y así se prosigue hasta terminar de ordenar [1].

En esencia, un *grupo de ordenamiento* es un árbol *binario casi completo* (todos los nodos hasta el anteúltimo nivel tienen dos hijos, a excepción de los nodos más a la derecha en ese nivel que podrían no tener los dos hijos) en el cual el valor de cada nodo es mayor que el valor de sus dos hijos. La idea es comenzar con todos los

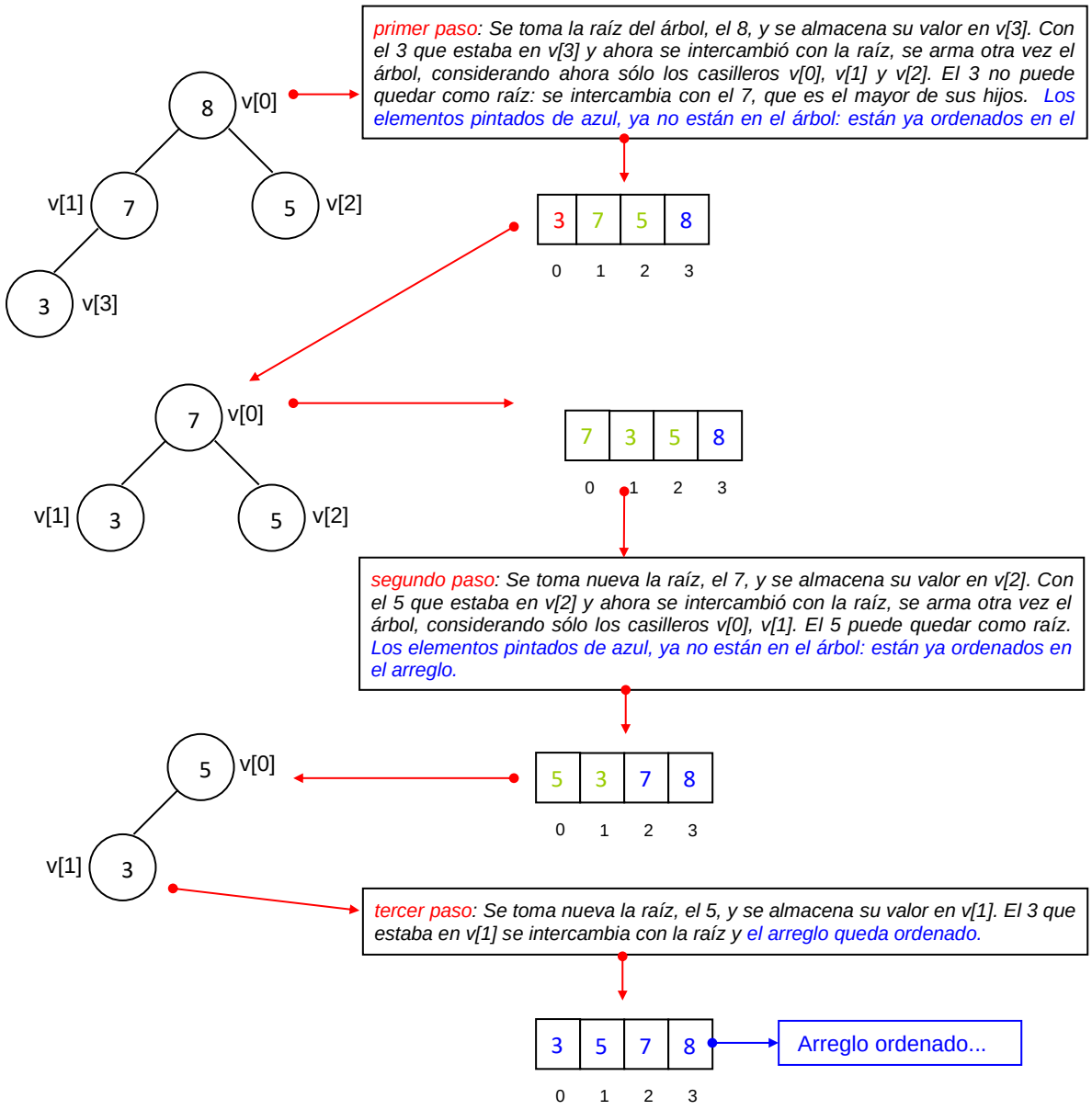
elementos del arreglo, y formar un árbol de este tipo, pero dentro del mismo arreglo (de otro modo, se usaría demasiado espacio adicional para ordenar...). Es simple implementar un árbol completo o casi completo en un arreglo: se ubica la raíz en el casillero v[0], y luego se hace que para nodo en el casillero v[i], su hijo izquierdo vaya en v[2*i + 1] y su hijo derecho v[2*i + 2]. Así, el algoritmo Heapsort primero lanza una fase de armado del grupo, en la cual los elementos del arreglo se reubican para simular el árbol. La Figura 5 muestra un esquema general del proceso de armado del grupo inicial para un pequeño arreglo de cuatro elementos.

Figura 5: Funcionamiento general del Algoritmo *Heapsort* - Fase 1: Armado del Grupo Inicial.



Una vez armado el grupo inicial[1], comienza la segunda fase del algoritmo: Se toma la raiz del árbol (que es el mayor del grupo), y se lleva al último casillero del arreglo (donde quedará ya ordenado). El valor que estaba en el último casillero se reinserta en el árbol (que ahora tiene un elemento menos), y se repite este mecanismo, llevando el nuevo mayor al anteúltimo casillero. Así se continúa, hasta que el árbol quede con sólo un elemento, que ya estará ordenado en su posición correcta del arreglo. Gráficamente, la segunda fase se desarrolla como se muestra en la *Figura 6*.

Figura 6: Funcionamiento general del Algoritmo *Heapsort* - Fase 2: Ordenamiento Final.



[1] El *Heapsort* usa un "grupo de datos" para favorecer el proceso de ordenamiento… Y eso nos refiere a la importancia del trabajo en grupos o en equipos para tareas de otra índole. En 1998, la película *Saving Private Ryan* (o *Rescatando al Soldado Ryan*) dirigida por *Steven Spielberg* y protagonizada por *Tom Hanks*, cuenta la dramática historia de un pelotón de ocho soldados que tenían la misión de entrar en territorio enemigo y rescatar a un único soldado (el tal Ryan…) en el contexto de la más feroz guerra que haya experimentado la humanidad: la Segunda Guerra Mundial. La película profundiza en la muy delgada línea que lleva a las personas a tomar decisiones aparentemente ridículas, pero fundamentadas en razones de ética, honor, compañerismo y respeto: ¿Qué tanto se justifica arriesgar las vidas de ocho hombres para salvar la de uno solo?

La función que implementa el algoritmo *Heapsort* es la siguiente, y está incluida en el módulo ordenamiento.py del proyecto *[F22] Ordenamiento* que acompaña a esta Ficha:

```
def heap_sort(v):
    # ordenamiento Heap Sort
    n = len(v)

    # Primera fase: crear el grupo inicial...
    for i in range(n):
        e = v[i]
        s = i
        f = (s - 1) // 2
        while s > 0 and v[f] < e:
            v[s] = v[f]
            s = f
            f = (s - 1) // 2
        v[s] = e

    # Segunda fase: Extraer la raiz, y reordenar el vector y el grupo...
    for i in range(n-1, 0, -1):
        valori = v[i]
        v[i] = v[0]
        f = 0
        if i == 1:
            s = -1
        else:
            s = 1
        if i > 2 and v[2] > v[1]:
            s = 2
        while s >= 0 and valori < v[s]:
            v[f] = v[s]
            f = s
            s = 2*f + 1
            if s + 1 <= i - 1 and v[s] < v[s+1]:
                s += 1
            if s > i - 1:
                s = -1
        v[f] = valori
```

A modo de cierre, y para tener una visión completa, compacta y comparativa de estos algoritmos, el proyecto *[F22] Ordenamiento* que acompaña a esta Ficha, contiene un modelo *test01.py* en el cual se puede seleccionar por medio un menú la técnica de ordenamiento a emplear para ordenar un vector. En ese programa hemos aplicado los seis algoritmos citados en la tabla de la *Figura 1*, incluyendo el *Heapsort* (que como se dijo, se agrega en esta Ficha a modo de tema opcional) y el *Quicksort* (que será presentado en una ficha posterior cuando se analice el tema de la estrategia *Divide y Vencerás*). El programa además, calcula el tiempo que cada algoritmo demora en terminar de ordenar el vector. Pruebe con valores grandes de *n*… y saque sus propias conclusiones. El programa completo es el siguiente (sin validar en cada opción si el arreglo está creado o no… dejamos ese control para los estudiantes):

```
import time
import ordenamiento

__author__ = 'Cátedra de AED'


def validate(inf):
    n = inf
```

```
    while n <= inf:
        n = int(input('Cantidad de elementos (> a ' + str(inf) + ' por favor): '))
        if n <= inf:
            print('Error: se pidio mayor a', inf, '... cargue de nuevo...')
    return n
def test():
    op = 0
    while op != 9:
        print('1. Generar el vector')
        print('2. Verificar orden')
        print('3. Ordenamiento por Selección Directa')
        print('4. Ordenamiento por Intercambio Directo (Burbuja)')
        print('5. Ordenamiento por Inserción Directa')
        print('6. Ordenamiento Heapsort')
        print('7. Ordenamiento Quicksort')
        print('8. Ordenamiento Shellsort')
        print('9. Salir')
        op = int(input('\t\tIngrese opción: '))
        if op == 1:
            print()
            n = validate(0)
            v = n * [0]
            ordenamiento.generate_random(v)
            print('Hecho... arreglo creado...')
            print()

        elif op == 2:
            print()
            if ordenamiento.check(v):
                print('Está ordenado...')
            else:
                print('No está ordenado...')
            print()

        elif op == 3:
            print()
            print('Ordenamiento: Selección Directa.')
            t1 = time.perf_counter()
            ordenamiento.selection_sort(v)
            t2 = time.perf_counter()
            tt = t2 - t1
            print('Hecho... Tiempo total insumido:', tt, 'segundos')
            print()

        elif op == 4:
            print()
            print('Ordenamiento: Intercambio Directo (Burbuja).')
            t1 = time.perf_counter()
            ordenamiento.bubble_sort(v)
            t2 = time.perf_counter()
            tt = t2 - t1
            print('Hecho... Tiempo total insumido:', tt, 'segundos')
            print()

        elif op == 5:
            print()
            print('Ordenamiento: Inserción Directa.')
            t1 = time.perf_counter()
            ordenamiento.insertion_sort(v)
            t2 = time.perf_counter()
            tt = t2 - t1
            print('Hecho... Tiempo total insumido:', tt, 'segundos')
            print()

        elif op == 6:
            print()
            print('Ordenamiento: Heap Sort.')
            t1 = time.perf_counter()
```

```
            ordenamiento.heap_sort(v)
            t2 = time.perf_counter()
            tt = t2 - t1
            print('Hecho... Tiempo total insumido:', tt, 'segundos')
            print()
        elif op == 7:
            print()
            print('Ordenamiento: Quick Sort.')
            t1 = time.perf_counter()
            ordenamiento.quick_sort(v)
            t2 = time.perf_counter()
            tt = t2 - t1
            print('Hecho... Tiempo total insumido:', tt, 'segundos')
            print()

        elif op == 8:
            print()
            print('Ordenamiento: Shell Sort.')
            t1 = time.perf_counter()
            ordenamiento.shell_sort(v)
            t2 = time.perf_counter()
            tt = t2 - t1
            print('Hecho... Tiempo total insumido:', tt, 'segundos')
            print()

# script principal...
if __name__ == '__main__':
    test()
```

Bibliografía

[1] Y. Langsam, M. Augenstein and A. Tenenbaum, Estructura de Datos con C y C++, México: Prentice Hall, 1997.

[2] R. Sedgewick, Algoritmos en C++, Reading: Addison Wesley - Díaz de Santos, 1995.

[3] Python Software Foundation, "Python Documentation," 2021. [Online]. Available: https://docs.python.org/3/.

[4] V. Frittelli, Algoritmos y Estructuras de Datos, Córdoba: Universitas, 2001.

[5] M. Pilgrim, "Dive Into Python - Python from novice to pro", Nueva York: Apress, 2004.