1. INTRODUCCIÓN TEÓRICA.

1.1. **Montículos**, **propiedades**:

El nodo i es el padre de $\begin{cases} 2 * i \\ 2 * i + 1 \end{cases}$

El nodo i es el hijo del nodo i div 2 (o $i \div 2$).

En resumen, para los vectores sería:

Hijos con respecto a padres $\begin{cases} T[i] \geq T[2 * i] \\ T[i] \geq T[2 * i + 1] \end{cases}$

Padres con respecto a hijos. $T[i] \leq T[i \ div \ 2]$

- Hay dos posibles algoritmos para crear un montículo.

1. El primer algoritmo es:

```
procedimiento crear-montículo-lento(T[1..n])
   para i ←2 hasta n hacer flotar(T[1..i]),i) fpara
fprocedimiento
```

- el algoritmo iterativo de flotar una posición es:

```
procedimiento flotar(T[1..n],i)
   k ←i;
   repetir
      j ←k;
      si j > 1 y T[j ÷ 2] < T[k] entonces
         k ←j ÷ 2
         intercambiar T[j] y T[k]
         /* si j = k, entonces el nodo ha llegado a su posición final*/
   hasta que j = k
fprocedimiento
```

2. El segundo algoritmo es el siguiente:

```
procedimiento crear-montículo(T[1..n])
   para i ←[n/2] hasta 1 hacer hundir(T,i) fpara
fprocedimiento
```

- el algoritmo iterativo para *hundir* un elemento del montículo es: MÁS RÁPIDO

```
procedimiento hundir(T[1..n],i)
   k ←i;
   repetir
      j ←k;
      /* Buscar el hijo mayor del nodo j */
      si 2 * j ≤ n y T[2 * j] > T[k] entonces k ←2 * j
      si 2 * j ≤ n y T[2 * j] > T[k] entonces k ←2 * j + 1
      intercambiar T[j] y T[k]
      /* si j = k, entonces el nodo ha llegado a su posición final */
   hasta que j = k
fprocedimiento
```

- El primero es ineficiente con respecto al segundo, hace más operaciones de *flotar* y aunque tiene coste lineal la constante multiplicativa es mayor, por tanto, el algoritmo segundo es más eficiente.

2. EJERCICIOS.

2.1. Explicar cómo funciona el algoritmo que dota a un vector la propiedad del montículo en tiempo lineal (no es necesario demostrar que ese es el coste). Tomar como ejemplo el vector $V = [4,3,7,7,10,1]$.

2.2. Dado el siguiente montículo $[10,6,3,5,2,3,2]$ se pide insertar el valor 6 describiendo toda la secuencia de cambios en el mismo.

2.3. Dado un montículo T[1..n], programar completamente en pseudocódigo una función recursiva flotar(T,i) para flotar el elemento de la posición i del vector T. Explicar cómo usar esta función para insertar un elemento en el montículo.

2.4. Programar en pseudocódigo un algoritmo recursivo para la operación de hundir un elemento en un montículo.

2.5. Implementar una versión recursiva de una función que tome un vector y le dé estructura de montículo.

2.6. Sea T[1. .12] una matriz tal que T[i]=i para todo $i \leq 12$. Crear un montículo en tiempo lineal, especificando cada paso y mostrando en todo momento el estado de la matriz T.

2.7. Dado m = [2,6,4,12,7,4,4]. Comprobar si es o no montículo de mínimos. Si no lo es, programar una función para convertirlo en montículo y aplicarlo a 'm'. Si no, programar una función de añadir elemento mediante la función "flotar" y aplicarlo al valor 3. En ambos casos, escribir el montículo resultante y detallar todos los pasos.

2.8. Sea T[1. . n] con k elementos (k < n) un montículo de mínimos. Se pide programar una función **recursiva** "flotar" que dado un nuevo elemento T[k+1] restaure la propiedad del montículo en T.

2.9. Estudiar la complejidad del algoritmo de ordenación por Selección por la llamada al procedimiento, especificado a continuación, `Selection (a,1,n)`.

- El procedimiento Selección puede ser implementado como sigue:

```
procedimiento  Selection (a:vector; primero,ultimo: int);
   para i=primero hasta ultimo-1 hacer
         posmin = PosMinimo(a,i,ultimo);
         Intercambia(a, i, posmin);
   fpara
fprocedimiento Selection
```

➢ En el algoritmo anterior se utiliza una función **`PosMinimo`** que calcula la posición del elemento mínimo de un subvector :

```
Int función PosMinimo (a:vector;primero,ultimo:int);
/* devuelve la posición del mínimo elemento de a[primero..ultimo] */
      pmin=primero;
   para i=primero+1 hasta ultimo hacer
         si a[i] < a[pmin] entonces
               pmin = i
         fsi;
      fpara
   return pmin;
ffuncion PosMinimo;
```

➢ También se utiliza el procedimiento **`Intercambia`** para intercambiar dos elementos de un vector:

```
función Intercambia (a:vector ; i , j :int );
/* intercambia a[i]  con a[j]  */
      aux = a[i] ;
   a[i] = a[j] ;
      a[j] = aux;
ffuncion Intercambia;
```

2.10. Ejercicio de examen, septiembre_19.

- Se tiene el algoritmo de ordenación siguiente:

```
procedimiento TerciosSort (A:vector; I,J: int);
   K ←((J-I+1)/3);
   si A(I) > A(J) entonces Intercambia(A(I),A(J)) fsi;
   si J-I+1 <= 2 entonces return fsi;
   TerciosSort (A, I, J-K);
   TerciosSort (A, I+K, J);
   TerciosSort (A, I, J-K);
fprocedimiento TerciosSort;
```

- Se utiliza el procedimiento **Intercambia** para intercambiar dos elementos de un vector:

```
procedimiento Intercambia (a:vector ; i , j :int );
/* intercambia a[i] con a[j] */
   aux = a[i];
    a[i] = a[j];
   a[j] = aux;
fprocedimiento Intercambia;
```

Se pide:

a. (2 puntos). Estudiar la complejidad del algoritmo TerciosSort por la llamada al procedimiento TerciosSort(A,1,n).

b. (0.5 punto). Comparar la eficiencia del algoritmo propuesto con los de ordenación por Inserción y por Quicksort.

NOTAS:

- Se puede utilizar valores de n de la forma $(3/2)^m$
- $\log_3 2 = 0.63092$
- $\log_b a = \log_c a / \log_c b$

2.11. Algoritmo Recursivo para la Búsqueda Ternaria.

El algoritmo de "búsqueda ternaria" realiza una búsqueda de un elemento en un vector ordenado.
La función compara el elemento a buscar "clave" con el que ocupa la posición n/3 y si este es menor que el elemento a buscar se vuelve a comparar con el que ocupa la posición 2n/3. En caso de no coincidir ninguno con el elemento buscado se busca recursivamente en el subvector correspondiente de tamaño 1/3 del original.

a. Escribir un algoritmo para la búsqueda ternaria.
b. Calcular la complejidad del algoritmo propuesto.
c. Comparar el algoritmo propuesto con el de búsqueda binaria.