Ejercicio 3

Debemos encontrar los valores mínimo y máximo de un vector realizando comparaciones con como máximo 3/2 veces los valores.

Si solo tuviéramos que buscar el máximo o el mínimo del vector tendríamos que recorrer este realizando al menos una comparación con cada número de modo que obtendríamos una complejidad O(n)

Podríamos buscar con el algoritmo mencionado primero el máximo del vector y posteriormente el mínimo, pero entonces estaríamos realizando 2n comparaciones.

- Para reducir el número de comparaciones evaluamos los elementos en posiciones simétricas respecto del centro del vector intercambiándolos de modo que a un lado del vector queden los menores de la comparación y al otro los mayores. Con esto habremos realizado 1/2 n comparaciones y nos habremos asegurado que de la mitad a un lado del vector estará el máximo de este y en el otro lado estará el mínimo.

  Sobre la mitad que contiene al máximo realizamos una búsqueda secuencial para encontrarlo de modo que realizamos 1/2 n comparaciones.

  Hacemos lo mismo con la mitad que contiene el mínimo realizando otras 1/2 n comparaciones.

Finalmente habremos encontrado el máximo y el mínimo del vector mediante un algoritmo voraz realizando tan solo 3/2 n comparaciones.

Función principal:

```
/**
 De un array de datos se nos proporciona su valor máximo y mínimo.
 Se comparan solo 3/2 veces todos los datos.
 */
func findMinMax(of data: inout [Int]) -> (min :Int,max :Int){
    let half = data.count/2
    for index in 0..<half{                          //Se realizan comparaciones con 1/2 de los valores
        let opposite_index = data.count-1-index
        if (data[index] > data[opposite_index]){
            data.swapAt(index, opposite_index)
        }
    }//En la primera mitad del array aseguramos que está el valor mínimo y en la segurnda el valor máximo.
    //Se realizan comparaciones con 1/2 de los valores
    let min = findBest(of : data, using : {d1, d2 in d1 < d2}, from: 0, to: half)
    //Se realizan comparaciones con 1/2 de los valores
    let max = findBest(of : data, using : {d1, d2 in d1 > d2}, from: half, to: data.count)
    return (min,max)
}
```

Función auxiliar:

```
/**
 De un array de datos se nos proporciona su mejor valor para la función de optimización proporcionada entre el rango indicado.
 Se comparan todos los datos en el rango indicado.
 */
func findBest(of values: [Int], using comparator : (Int,Int)->Bool, from start_point : Int = 0, to end_point : Int) -> Int {
    var best = values.first!
    for index in start_point ..< end_point{
        let value = values[index]
        if (comparator(value, best)){
            best = value
        }
    }
    return best
}
```

Ejercicio 4

Demos buscar el camino mínimo en un grafo no dirigido utilizando el algoritmo de Prim.

Demos recoger todos los pasos que da el algoritmo para resolver el problema. Para ello se utilizan las siguientes estructuras de datos que se rellenan durante la ejecución del algoritmo.

La información recogida en dichas estructuras es hasta cierto punto redundante, pero se utilizan pues lo que se pretende demostrar es que se ha comprendido el funcionamiento del algoritmo y que se sabe qué valor contiene cada variable durante su ejecución y lo que dicho valor representa.

- Matriz nxn en la que se indican los valores de las aristas que forman el camino mínimo.
- Vector de tamaño n cuyos elementos indican:
  - Valor de la nueva arista introducida
  - Nodo desde el que se insertó la arista
  - Nuevo nodo visible tras insertar la arista.

El algoritmo de Prim genera el camino mínimo a partir de un nodo inicial añadiendo en cada iteración el nodo al que desde el inicial se puede llegar formando un camino mínimo.
En cada iteración se añade un nuevo componente válido de modo que el algoritmo es voraz. Esta nueva arista genera una estructura de árbol respecto de las otras ya que se añade a un nodo ya visitado haciendo a otro nuevo visible.

```
func findPath (in data :[[Int]]) -> (graph: [[Int]],order: [arista]) {
    var nodes = data
    var selected_graph = [[Int]].init(repeating: [Int].init(repeating: 0, count: nodes.first!.count), count: nodes.count)
    var selected_order = [arista]()
    var near = [Int].init(repeating: 0, count: nodes.count)
    var distance = [Int].init(nodes.map(){$0.first!}) //Iniciamos el vector distancia al primer valor de cada columna
    for i in 0..<nodes.count-1 {
        var min = max
        var min_index = 0
        //Buscamos el mínimo del vector distancia
        for j in 0..<distance.count {
            if distance[j] < min{
                min = distance[j]
                min_index = j
            }
        }
        //Rellenamos los valores de la salida
        selected_graph[near[min_index]][min_index]=distance[min_index]
        selected_graph[min_index][near[min_index]]=distance[min_index]
        selected_order.append(arista.init(x: near[min_index], y: min_index, distance: distance[min_index]))
        //Marcamos el nodo como tomado para evitar bucles
        nodes[near[min_index]][min_index]=max
        distance[min_index]=max
        //Actualizamos el valor distancia poniendo el mínimo entre el valor que ya tenía para cada columna compardo
        //con el que hay en el grafo en la fila en la que se ha encontrado el valor mínimo
        for j in 0..<nodes[i].count {
            if nodes[j][min_index] < distance[j]{
                distance[j] = nodes[j][min_index]
                near [j] = min_index
            }
        }
    }
    return (selected_graph,selected_order)
}
```

Ejercicio 6

Debemos implementar un algoritmo voraz para unir las escaleras de la forma más eficiente posible.
Para ser voraz nuestro algoritmo deberá en cada iteración anadir una nueva parte válida a la solución de modo que sea la más eficiente de entre las existentes en el momento en el que se añada.
Analizando el problema vemos que el modo más eficiente de resolverlo es tomando en cada iteración las dos escaleras de menor tamaño de entre las posibles.
Para implementar el algoritmo partiremos de una lista ordenada de escaleras donde tomaremos las dos primeras. La nueva escalera resultado de la unión será inserta de nuevo en la lista de modo que esta siga siendo ordenada tras la inserción.

```
var nueva_escalera : Int
var escalera_1 : Int
var escalera_2 : Int
var insert_index : Int
var time = 0;
for _ in escaleras{
    if escaleras.count > 2 {
        //Los dos primeras escaleras del vector son los que proporcionan un tiempo mínimo ya que son las más pequeñas
        escalera_1 = escaleras.remove(at: 0)
        escalera_2 = escaleras.remove(at: 0)
        nueva_escalera = escalera_1 + escalera_2
        time += nueva_escalera;
        insert_index = 0;
        //Colocamos la nueva escalera ordenada dentro del vector
        while insert_index < escaleras.count && (escaleras[insert_index] < nueva_escalera){
            insert_index += 1
        }
        escaleras.insert(nueva_escalera, at: insert_index)
        //Mostramos datos
        print ("Added: \(nueva_escalera) <-- \(escalera_1) + \(escalera_2), at: \(insert_index)")
        print ("time: \(time)")
        print (escaleras)
    } else {//Con los dos últimos valores ya no es necesario ejecutar el algoritmo, los tomamos en orden
        for escalera_impar in escaleras{
            print ("Added: \(escalera_impar)")
            time += escalera_impar;
            escaleras.remove(at: 0)
        }
    }
}
print ("Total time: \(time)")
```