Ejercicio 15.- Se desea organizar una competición en la que participan k equipos, cada uno de los cuales cuenta con S_i seguidores, $1 \leq i \leq k$. Cada vez que se juega un partido, el equipo que gana sigue compitiendo, y el perdedor desaparece de la competición. Cada vez que un equipo pierde, todos sus seguidores pasan a serlo del equipo que los ha derrotado. A cada partido acuden todos los seguidores (actuales) de los dos equipos. Se pide:

> **1.-** Diseñar un algoritmo que organice los partidos (es decir, que decida quién debe jugar con quién) de forma que la suma de las asistencias a los mismos sea mínima. Demostrar que la solución computada es en efecto la que produce la mínima asistencia global.
>
> **2.-** Diseñar otro algoritmo que organice la competición de forma que ahora la suma de las asistencias a los partidos celebrados sea máxima. Demostrar que la solución computada es en efecto la que produce la máxima asistencia global.

1.-

La idea será emparejar cada vez a los equipos no eliminados que tengan una menor afición hasta ese momento. Esta solución es similar a la que se implementa para la solución del problema del juez *Lo que cuesta sumar.* El algoritmo es el siguiente:

```
// JUAN CARLOS LLAMAS NÚÑEZ 3º DG MARP

#include <iostream>
#include <fstream>
#include <vector>
#include "PriorityQueue.h"
#include <queue>
using namespace std;

bool gana(equipo e1, equipo e2);

struct equipo {
    int aficion;
};

queue<pair<equipo, equipo>> organizar(vector<equipo> const& equipos) {
    queue<pair<equipo, equipo>> partidos;
    PriorityQueue<equipo> pq;
    for (auto e : equipos)
        pq.push(e);
    while (pq.size() > 1) {
        equipo e1 = pq.top(); pq.pop();
        equipo e2 = pq.top(); pq.pop();
        partidos.push({ e1,e2 });
        if (gana(e1, e2)) {
            e1.aficion += e2.aficion;
            pq.push(e1);
        }
        else {
            e2.aficion += e1.aficion;
            pq.push(e2);
        }
    }
}
```

Hacemos notar que se presupone que la clase pública equipo dispone de un comparador menor y que indica si un equipo es menor que otro en base al número de aficionados que posee en un determinado instante de tiempo. Asimismo, presuponemos que la función gana que recibe como argumentos dos equipos devuelve un booleano que nos indica si el primero de ellos ha ganado al segundo.

2.-
Podemos hacer lo mismo con un comparador de equipos que ordene de mayor a menor y el mismo algoritmo sería válido. Sin embargo, en este caso bastaría que cada partido que se jugara fuera entre los dos equipos con mayor afición, ya que en la inserción en la cola de prioridad dentro del bucle colocará al equipo que gane el partido como el primero de la cola. Esto es así porque dicho equipo tendrá como número de aficionados la suma de los de las dos aficiones anteriores, que eran las dos más grandes. Por tanto, podemos ahorrarnos la cola de prioridad si implementamos el algoritmo de la siguiente manera:

```
// JUAN CARLOS LLAMAS NÚÑEZ 3º DG MARP

#include <iostream>
#include <fstream>
#include <vector>
#include "PriorityQueue.h"
#include <queue>
#include <algorithm>
using namespace std;

bool gana(equipo e1, equipo e2);

struct equipo {
    int aficion;
};

queue<pair<equipo, equipo>> organizar(vector<equipo> equipos) {
    queue<pair<equipo, equipo>> partidos;
    sort(equipos.begin(), equipos.end());
    int indGanador = 0;
    for (int i = 1; i < equipos.size(); i++) {
        partidos.push({ equipos[indGanador],equipos[i] });
        if (gana(equipos[indGanador], equipos[i])) {
            equipos[indGanador].aficion += equipos[i].aficion;
        }
        else {
            equipos[i].aficion += equipos[indGanador].aficion;
            indGanador = i;
        }
    }
    return partidos;
}
```

Asumimos que contamos con un comparador de equipos que hace que el vector, tras la función sort, quede ordenado por equipos de mayor a menor por número de aficionados.

Para demostrar que la solución computada es en efecto la que produce la mínima asistencia global, basta observar que el orden a seguir se puede representar mediante un árbol de Huffman, donde el número de veces que la afición de un equipo interviene en la suma (el total) es su profundidad en el árbol. Si n es el número de equipos, el coste total es entonces:

$$\mathrm{T} = \sum_{i=1}^{n} p_i \cdot l_i$$

Entonces el problema es totalmente análogo al de la codificación óptima de ficheros y, el algoritmo anterior es esencialmente el algoritmo voraz que resuelve dicho problema. Por tanto, tenemos que la solución producida por el algoritmo anterior es óptima y además se produce en coste O(*nlogn*) debido al uso de un montículo para almacenar las aficiones en cada paso.

En cuanto al apartado 2 veamos que, en efecto, la solución computada es la que produce una máxima asistencia global. Es claro que existe un número finito de formas de agrupar a los equipos de manera que los que pierdan desaparecen de la competición y los que ganen absorban la afición de los perdedores. Por tanto, existe una solución que maximiza la suma global de asistencia.

En primer lugar, si el número de equipos que participan en la competición es n, el número de partidos que se deben jugar para resolver la competición es exactamente n-1 porque después de cada partido se elimina exactamente un equipo (no se permiten empates).

Sea T_i = {E equipo | E sigue tras haberse jugado i partidos} con $0 \leq i \leq n-2$.

Sea C = (P_1, P_2, …, P_{n-1}) la tupla ordenada de cada uno de los partidos que se juegan en la competición donde cada P_i = (E, Ê) con E≠Ê en T_{i-1} con $1 \leq i \leq n-1$. Consideramos esta la solución óptima del problema.

Denotamos también por M_i^1 = *max*{ Afición(E) | E en T_{i-1}} que es la afición del equipo con mayor número de aficionados tras haberse jugado i-1 partidos. En caso de haber algún empate en número de aficionados (no se consideran empates entre equipos en los partidos disputados) se puede desempatar por quien tenga un menor índice. Análogamente sea M_i^2 = *max*{ Afición(E) | E en $T_{i-1}\backslash\{M_i^1\}$ } y desempatamos de la misma forma. Por tanto, $M_i^1 + M_i^2$ >= Asistencia(P_i).

Podemos considerar entonces C^*= (P_1^*, P_2^*, …, P_{n-1}^*) donde P_i^* = (E, Ê) con Afición(E) = M_i^1 y Afición(Ê) = M_i^2.

Se tiene entonces que:

$$\text{Asistencia}(\mathrm{C}) = \sum_{i=1}^{n-1} Asistencia(P_i) \leq \sum_{i=1}^{n-1} M_i^1 + M_i^2 = \sum_{i=1}^{n-1} Asistencia(P_i^*) = Asistencia(C^*)$$

Podemos observar que C^* es exactamente la solución computada por el algoritmo propuesto, ya que en cada paso se eligen los equipos con dos mayores aficiones. Esto está garantizado porque el vector está ordenado de mayor a menor, lo vamos recorriendo de izquierda a derecha y en cada paso se derrota a uno de los equipos de los que más aficionados tiene y su afición se trasvasa al otro.

Además, M^1_{i+1} = NuevaAfición(ganador(P_i^*)) = $M_i^1 + M_i^2$ ya que esta es la mayor afición de los equipos restantes. Por tanto, tras ordenar los equipos de mayor a menor es suficiente con llevar el índice del equipo que ha ganado el último partido, que será el que más aficionados tenga, y enfrentarlo con el siguiente de mayor afición, que será el siguiente en el recorrido del vector. Esto demuestra la corrección del algoritmo propuesto. El orden de complejidad es el de ordenar un vector de n elementos, es decir, O(*nlogn*).