# Hoja 2: Ejercicio 10

## Ramificación y poda

Diego Rodríguez Cubero

UCM

2 de marzo de 2026

# Contenidos

# Enunciado del problema

## Ejercicio 10

Una de las pruebas habituales del concurso Supervivientes es la construcción de una cabaña rudimentaria cuyo techo es una simple lona soportada por cuatro pilares. Los concursantes dispondrán de $n$ de fragmentos de caña de longitudes enteras $l_1, \cdots, l_n$ ensamblando los cuales deberán obtener los cuatro pilares. El objetivo es que sus alturas queden razonablemente equilibradas y sean lo más altas posibles. Precisando, desarrollar el problema buscando maximizar el pilar más bajo de los cuatro.

# Idea general

Para resolver este problema, podemos utilizar una técnica de ramificación y poda. La idea es generar todas las posibles combinaciones de los fragmentos de caña para formar los cuatro pilares, y luego podar aquellas combinaciones que no cumplan con el objetivo de maximizar el pilar más bajo.

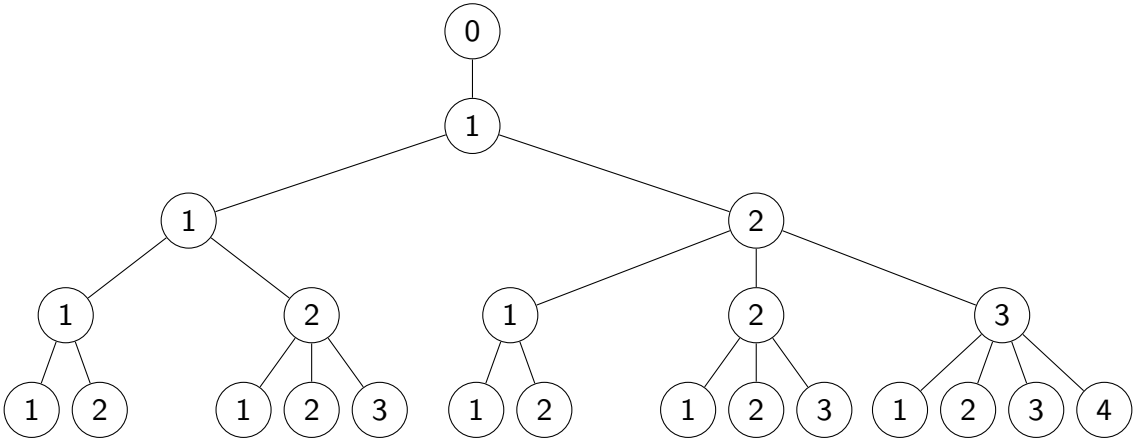
# Espacio de soluciones

El espacio de soluciones se puede representar como un vector:

$$(x_1, \cdots, x_n) : \qquad x_i \in \{1, 2, 3, 4\} \quad \forall i \in \{1, \cdots, n\}$$

donde $x_i$ representa la asignación del fragmento de caña $l_i$ a uno de los cuatro pilares. En forma de arbol se representaría de la siguiente manera, donde cada arista de la altura $i$ a un nodo de valor $k$ significa asignar $x_i = k$.

# Árbol



# Cota optimista

Para calcular la cota optimista, la idea sera ir dividiendo cada framgento (no se puede en el ejercicio, pero lo haremos para la cota), repartiendolo entre los cuatro pilares, y en el momento que todos tengan la misma altura, dividir cada fragmento restante entre los cuatro pilares, lo que nos dará una cota superior para la altura del pilar más bajo.
Este algoritmo tendrá un coste de $O(n)$, ya que recorremos los fragmentos restantes y en cada paso encontramos el pilar más bajo en tiempo constante.
Esto lo podemos hacer sumando todas las longitudes, primero igualamos los 2 pilares más bajos, luego los 3 pilares más bajos (2 de ellos ya estarán igualados), luego los 4 pilares (3 de ellos ya estarán igualados), y finalmente dividimos el resto entre los 4 pilares.

# Cota optimista - Código

El código para calcular la cota optimista podría ser el siguiente:

```
1         int cota_optimista {
2             int longitud_restante = sum(longitudes.begin() + k, longitudes.
      end());
3             vector<int> alturas_copia = alturas.sort(); // Copia de las
      alturas actuales
4             // Igualar los pilares más bajos
5             for (int i = 0; i < 3; i++) {
6                 int anadido = min(longitud_restante, (alturas_copia[i + 1] -
       alturas_copia[0]) * (i + 1));
7                 for (int j = 0; j <= i; j++) {
8                     alturas_copia[j] += anadido / (i + 1);
9                 }
10            }
11            return alturas_copia[0] + longitud_restante / 4;
12        }
13
```

# Cota pesimista

En este problema es claro ver que cualquier solucion es factible, por mala que sea, por lo que para calcular una cota pesimista útil (siempre se podria usar 0, pero no nos serviria para nada), podemos ir asignando los fragmentos de caña al pilar más bajo en cada paso, lo que nos dará una cota inferior para la altura del pilar más bajo.
Este algoritmo también tendrá un coste de $O(n)$, ya que recorremos los fragmentos restantes y en cada paso encontramos el pilar más bajo en tiempo constante.

# Cota pesimista - Código

El código para calcular la cota pesimista podría ser el siguiente:

```
1       int cota_pesimista(vector<int>& longitudes, vector<int>& alturas,
    int k) {
2           vector<int> alturas_copia = alturas; // Copia de las alturas
    actuales
3           // Igualamos
4           for (int i = k; i < longitudes.size(); i++) {
5               // Encontrar el pilar más bajo
6               int min_index = 0;
7               for (int j = 1; j < 4; j++) {
8                   if (alturas_copia[j] < alturas_copia[min_index]) {
9                       min_index = j;
10                  }
11              }
12              // Asignar el fragmento al pilar más bajo
13              alturas_copia[min_index] += longitudes[i];
14          }
15          return min(alturas);
16      }
17
```

# Explicación del algoritmo

Para hacer una correcta implementación del algoritmo de ramificación y poda, podemos definir el siguiente tipo "nodo"que guardara la información necesaria para cada nodo del árbol de búsqueda, a la vez que la prioridad (para tener acceso al nodo mas prometedor), que será la altura del pilar más bajo en la asignación actual:

```
1        struct Nodo {
2            vector<int> alturas; // Alturas actuales de los pilares
3            vector<int> asignaciones; // Asignación actual de fragmentos a
     pilares
4            int k; // Índice del fragmento a asignar
5            int prioridad; // Altura del pilar más bajo en la asignación
     actual (se puede cambiar por la cota optimista si se prefiere)
6            int cota_op; // Cota optimista para el nodo actual
7        };
8
```

```
int ramificacion_y_poda(vector<int>& l, int n, int& mejor, vector<int>& mejor_asignacion) {
    priority_queue<Nodo> pq; // Cola de prioridad para los nodos
    Nodo raiz; // Inicialización del nodo raíz inicial
    raiz.prioridad = 0;
    raiz.cota_op = cota_optimista(l, raiz.alturas, 0);
    mejor = cota_pesimista(l, raiz.alturas, 0);
    pq.push(raiz, raiz.prioridad);
    while (!pq.empty()) {
        Nodo nodo = pq.top(); pq.pop();
        if (nodo.cota_op <= mejor) continue; // poda
        if (nodo.k == n) { // hoja
            mejor = max(mejor, min(nodo.alturas)); mejor_asignacion = nodo.asignaciones; continue;
        }
        set<int> vistos; // evitar ramas simetricas
        for (int j = 0; j < 4; j++) {
            if (vistos.count(nodo.alturas[j])) continue;
            vistos.insert(nodo.alturas[j]);
            Nodo hijo; hijo.alturas = nodo.alturas;
            hijo.alturas[j] += l[nodo.k]; hijo.k = nodo.k + 1;
            hijo.prioridad = min(hijo.alturas[j], nodo.prioridad);
            hijo.cota_op = cota_optimista(l, hijo.alturas, hijo.k);
            if (hijo.cota_op > mejor) {
                mejor = max(mejor, cota_pesimista(l, hijo.alturas, hijo.k));
                pq.push(hijo, hijo.prioridad);
            }
        }
    }
    return mejor;
}
```

# Complejidad del algoritmo

La complejidad del algoritmo de ramificación y poda en el peor caso es $O(n4^n)$, ya que en el peor caso se exploran todas las combinaciones posibles de asignación de los fragmentos a los pilares, lo que da lugar a $4^n$ combinaciones, y para cada combinación se calcula la cota optimista y pesimista en $O(n)$, auqnue en la práctica, la poda puede reducir significativamente el número de combinaciones exploradas gracias a las podas que se realizan al comparar la cota optimista con la mejor solución encontrada hasta el momento.
El coste en espacio es $O(4^n)$, ya que en el peor caso se pueden almacenar todas las combinaciones posibles de asignación de los fragmentos a los pilares en la cola de prioridad.
Notese que si la mejor solucion coincide con la cota pesimista inicial, se tendria que calcular la asignacion exacta volviendo a realizar el proceso de asignacion de los fragmentos al pilar más bajo, lo que se podria hacer en $O(n)$, por lo que el coste total del algoritmo en dicho caso no cambia.