

Hoja 2: Ejercicio 10

Ramificación y poda

Diego Rodríguez Cubero

UCM

25 de febrero de 2026

Contenidos

- 1 Enunciado del problema
- 2 Idea general
- 3 Espacio de soluciones
- 4 Cota optimista
- 5 Cota pesimista
- 6 Algoritmo del ejercicio
- 7 Complejidad del algoritmo

Ejercicio 10

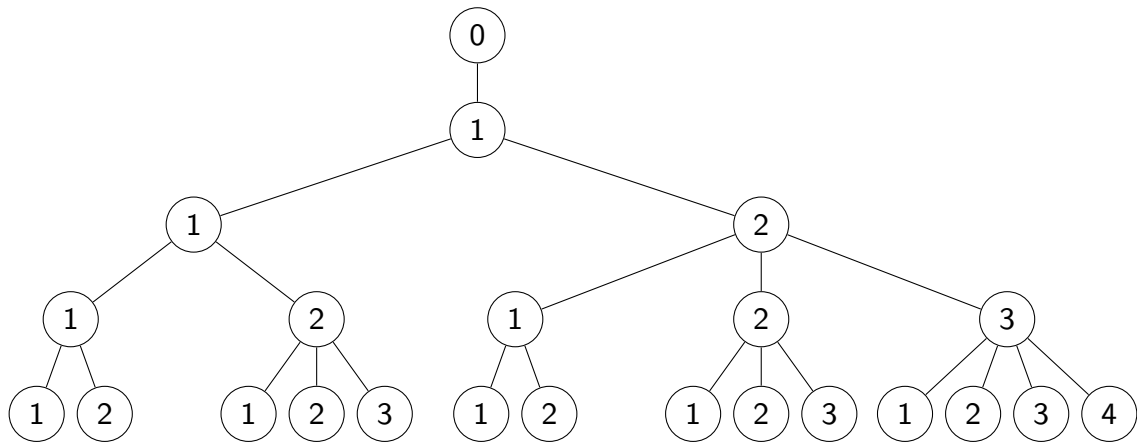
Una de las pruebas habituales del concurso Supervivientes es la construcción de una cabaña rudimentaria cuyo techo es una simple lona soportada por cuatro pilares. Los concursantes dispondrán de n de fragmentos de caña de longitudes enteras l_1, \dots, l_n ensamblando los cuales deberán obtener los cuatro 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.

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.

El espacio de soluciones se puede representar como un vector:

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

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



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), repartendolo 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 recorreremos 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     }
```


En este problema es claro ver que cualquier solución es factible, por mala que sea, por lo que para calcular una cota pesimista útil (siempre se podría usar 0, pero no nos serviría 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 recorreremos 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,
2      int k) {
3          vector<int> alturas_copia = alturas; // Copia de las alturas
4          actuales
5          // Igualamos
6          for (int i = k; i < longitudes.size(); i++) {
7              // Encontrar el pilar más bajo
8              int min_index = 0;
9              for (int j = 1; j < 4; j++) {
10                  if (alturas_copia[j] < alturas_copia[min_index]) {
11                      min_index = j;
12                  }
13              }
14              // Asignar el fragmento al pilar más bajo
15              alturas_copia[min_index] += longitudes[i];
16          }
17          return min(alturas);
18      }
```

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  
4     pilares  
5     int k; // Índice del fragmento a asignar  
6     int prioridad; // Altura del pilar más bajo en la asignación  
7     actual  
8     int cota_op; // Cota optimista para el nodo actual  
9 };
```

```

1 int ramificacion_y_poda(vector<int>& l, int n, int& mejor, vector<int>& mejor_asignacion) {
2     priority_queue<Nodo> pq; // Cola de prioridad para los nodos
3     Nodo raiz; // Inicialización del nodo raíz inicial
4     raiz.prioridad = 0;
5     raiz.cota_op = cota_optimista(l, raiz.alturas, 0);
6     mejor = cota_pesimista(l, raiz.alturas, 0);
7     pq.push(raiz, raiz.prioridad);
8     while (!pq.empty()) {
9         Nodo nodo = pq.top(); pq.pop();
10        if (nodo.cota_op <= mejor) continue; // poda
11        if (nodo.k == n) { // hoja
12            mejor = max(mejor, min(nodo.alturas)); mejor_asignacion = nodo.asignaciones; continue;
13        }
14        set<int> vistos; // evitar ramas simetricas
15        for (int j = 0; j < 4; j++) {
16            if (vistos.count(nodo.alturas[j])) continue;
17            vistos.insert(nodo.alturas[j]);
18            Nodo hijo; hijo.alturas = nodo.alturas;
19            hijo.alturas[j] += l[nodo.k]; hijo.k = nodo.k + 1;
20            hijo.prioridad = min(hijo.alturas[j], nodo.prioridad);
21            hijo.cota_op = cota_optimista(l, hijo.alturas, hijo.k);
22            if (hijo.cota_op > mejor) {
23                mejor = max(mejor, cota_pesimista(l, hijo.alturas, hijo.k));
24                pq.push(hijo, hijo.prioridad);
25            }
26        }
27    }
28    return mejor;
29 }
30

```

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 $O(n)$, aunque 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 $O(n)$, ya que en cada momento se almacenan las alturas de los pilares y las asignaciones de los fragmentos, lo que requiere espacio lineal en el número de fragmentos. Notese que si la mejor solución coincide con la cota pesimista inicial, se tendría que calcular la asignación exacta volviendo a realizar el proceso de asignación de los fragmentos al pilar más bajo, lo que se podría hacer en $O(n)$, por lo que el coste total del algoritmo en dicho caso no cambia.