Soluciones OIFem 2020
Entrenamiento 10

Máximo del segmento

Teoría

- Árboles de segmentos
- Propagación vaga

Solución

Usando el código de los apuntes, podemos resolverlo con facilidad. Basta con darnos cuenta de que la función del árbol es el máximo y que si se suma un valor a todos los nodos de un segmento, su máximo será incrementado por exactamente este valor.

Código

C++

```cpp
#include <iostream>
#include <vector>
using namespace std;

int hijoIzquierdo(int p) {
    return p << 1; // forma rápida de calcular 2p
}

int hijoDerecho(int p) {
    return (p << 1)+1; // forma rápida de calcular 2p + 1
}

int funcion(int valorIzq, int valorDer) {
    return max(valorIzq, valorDer); // esta funcion es la que hay que cambiar si queremos un árbol de segmentos que no calcule sumas, sino otra función
}

void construirArbol(int indiceNodo, int L, int R, vector<int> & arbol, vector<int> & lista) {
    // construye el árbol de segmentos entre los índices [L, R] de la lista de forma recursiva
    // hay que llamarla como (1, 0, N-1, arbol, lista) desde fuera de la función
    if (L < R) {
```

```
        int mid = L + (R-L)/2;
        construirArbol(hijoIzquierdo(indiceNodo), L, mid, arbol, lista); //
        ↪ construimos el subárbol izquierdo
        construirArbol(hijoDerecho(indiceNodo), mid + 1, R, arbol, lista); //
        ↪ construimos el subárbol derecho
        arbol[indiceNodo] = funcion(arbol[hijoIzquierdo(indiceNodo)],
        ↪ arbol[hijoDerecho(indiceNodo)]); // los juntamos
    } else {
        arbol[indiceNodo] = lista[L]; // L = R -> subárbol de un solo nodo:
        ↪ cogemos ese índice de la lista
    }
}

void propagar(int indiceNodo, int L, int R, vector<int> & arbol, vector<int> &
↪ vaga) {
    if (L != R) {
        arbol[hijoIzquierdo(indiceNodo)] += vaga[indiceNodo];
        vaga[hijoIzquierdo(indiceNodo)] += vaga[indiceNodo];
        arbol[hijoDerecho(indiceNodo)] += vaga[indiceNodo];
        vaga[hijoDerecho(indiceNodo)] += vaga[indiceNodo];
        vaga[indiceNodo] = 0;
    } else vaga[indiceNodo] = 0;
}

int conseguirValor(int indiceNodo, int L, int R, int indicePrincipio, int
↪ indiceFinal, vector<int> & arbol, vector<int> & vaga) {
    // encuentra el valor de la función sobre el rango [indicePrincipio,
    ↪ indiceFinal] de la lista original
    // hay que llamarla como conseguirValor(1, 0, N-1, indicePrincipio,
    ↪ indiceFinal, arbol) desde fuera de la función
    if (L >= indicePrincipio && R <= indiceFinal) {
        // indicePrincipio <= L <= R <= indiceFinal
        propagar(indiceNodo, L, R, arbol, vaga);
        return arbol[indiceNodo]; // [L, R] está contenido en [indicePrincipio,
        ↪ indiceFinal]
    } else if (indicePrincipio > R || indiceFinal < L) {
        // No hay ningún índice en común entre [L, R] y [indicePrincipio,
        ↪ indiceFinal]
        return -1e9; // si hay alguna posibilidad de una suma negativa, tendremos
        ↪ que elegir otro valor, como menos infinito
    } else {
        propagar(indiceNodo, L, R, arbol, vaga);
        int mid = L + (R-L)/2; // encontramos el punto intermedio
        // conseguimos el valor de la función sobre la intersección entre [L, mid]
        ↪ y [indicePrincipio, indiceFinal]
        int valorHijoIzquierdo = conseguirValor(hijoIzquierdo(indiceNodo), L, mid,
        ↪ indicePrincipio, indiceFinal, arbol, vaga);
        // conseguimos el valor de la función sobre la intersección entre [mid+1,
        ↪ R] y [indicePrincipio, indiceFinal]
        int valorHijoDerecho = conseguirValor(hijoDerecho(indiceNodo), mid + 1, R,
        ↪ indicePrincipio, indiceFinal, arbol, vaga);
        if (valorHijoIzquierdo == -1e9)
            return valorHijoDerecho; // [mid+1, R] está contenido en
            ↪ [indicePrincipio, indiceFinal] pero [L, mid] no
        else if (valorHijoDerecho == -1e9)
```

```
60              return valorHijoIzquierdo; // [L, mid] está contenido en
                ↪  [indicePrincipio, indiceFinal] pero [mid+1, R] no
61          else
62              return funcion(valorHijoIzquierdo, valorHijoDerecho); //
                ↪  [indicePrincipio, indiceFinal] tiene parte en ambas mitades
63      }
64  }
65
66  void sumar(int indiceNodo, int L, int R, int indInicio, int indFinal, int suma,
    ↪  vector<int> & vaga, vector<int> & arbol) {
67      if (indFinal < L || indInicio > R)
68          return;
69      else if (L >= indInicio && R <= indFinal) {
70          vaga[indiceNodo] += suma;
71          arbol[indiceNodo] += suma;
72      } else {
73          propagar(indiceNodo, L, R, arbol, vaga);
74          int mid = L + (R-L)/2;
75          sumar(hijoIzquierdo(indiceNodo), L, mid, indInicio, indFinal, suma, vaga,
            ↪  arbol);
76          sumar(hijoDerecho(indiceNodo), mid+1, R, indInicio, indFinal, suma, vaga,
            ↪  arbol);
77          arbol[indiceNodo] = funcion(arbol[hijoIzquierdo(indiceNodo)],
            ↪  arbol[hijoDerecho(indiceNodo)]);
78      }
79  }
80
81  int main() {
82      ios::sync_with_stdio(false);
83      cin.tie(NULL);
84      int T, N, Q, k, l, r, x;
85      cin >> T;
86      while(T--) {
87          cin >> N >> Q;
88          vector<int> lista(N);
89          for (int i = 0; i < N; i++)
90              cin >> lista[i];
91          vector<int> arbol(4*N, -1e9);
92          vector<int> vaga(4*N, 0);
93          construirArbol(1, 0, N-1, arbol, lista);
94          while(Q--) {
95              cin >> k >> l >> r;
96              l--;
97              r--;
98              if (k == 1) {
99                  cin >> x;
100                 sumar(1, 0, N-1, l, r, x, vaga, arbol);
101             } else {
102                 cout << conseguirValor(1, 0, N-1, l, r, arbol, vaga) << '\n';
103             }
104         }
105     }
106     return 0;
107 }
```

Dynamic Range Sum Queries

Teoría

- Árboles de segmentos

Solución

Este era un árbol con función suma, con la peculiaridad de que podía haber overflow y había que cambiar las funciones aprendidas para que usaran `long long int`.

Código

C++

```cpp
#include <vector>
#include <iostream>
using namespace std;
typedef long long int ll;

int hijoIzquierdo(int p) {
        return p << 1; // forma rápida de calcular 2p
}

int hijoDerecho(int p) {
        return (p << 1)+1; // forma rápida de calcular 2p + 1
}

ll funcion(ll valorIzq, ll valorDer) {
        return valorIzq + valorDer;
}

void construirArbol(int indiceNodo, int L, int R, vector<ll> & arbol, vector<ll> & lista) {
        // construye el árbol de segmentos entre los índices [L, R] de la lista de forma recursiva
        // hay que llamarla como (1, 0, N-1, arbol, lista) desde fuera de la función
        if (L < R) {
                int mid = L + (R-L)/2;
                construirArbol(hijoIzquierdo(indiceNodo), L, mid, arbol, lista); // construimos el subárbol izquierdo
                construirArbol(hijoDerecho(indiceNodo), mid + 1, R, arbol, lista); // construimos el subárbol derecho
                arbol[indiceNodo] = funcion(arbol[hijoIzquierdo(indiceNodo)], arbol[hijoDerecho(indiceNodo)]); // los juntamos
        } else {
                arbol[indiceNodo] = lista[L]; // L = R -> subárbol de un solo nodo: cogemos ese índice de la lista
        }
}

```

```cpp
ll conseguirValor(int indiceNodo, int L, int R, int indicePrincipio, int indiceFinal, vector<ll> & arbol) {
        // encuentra el valor de la función sobre el rango [indicePrincipio, indiceFinal] de la lista original
        // hay que llamarla como conseguirValor(1, 0, N-1, indicePrincipio, indiceFinal, arbol) desde fuera de la función
        if (L >= indicePrincipio && R <= indiceFinal) {
                return arbol[indiceNodo]; // [L, R] está contenido en [indicePrincipio, indiceFinal]
        } else if (indicePrincipio > R || indiceFinal < L) {
                // No hay ningún índice en común entre [L, R] y [indicePrincipio, indiceFinal]
                return -1; // si hay alguna posibilidad de una suma negativa, tendremos que elegir otro valor, como menos infinito
        } else {
                int mid = L + (R-L)/2; // encontramos el punto intermedio
                // conseguimos el valor de la función sobre la intersección entre [L, mid] y [indicePrincipio, indiceFinal]
                ll valorHijoIzquierdo = conseguirValor(hijoIzquierdo(indiceNodo), L, mid, indicePrincipio, indiceFinal, arbol);
                // conseguimos el valor de la función sobre la intersección entre [mid+1, R] y [indicePrincipio, indiceFinal]
                ll valorHijoDerecho = conseguirValor(hijoDerecho(indiceNodo), mid + 1, R, indicePrincipio, indiceFinal, arbol);
                if (valorHijoIzquierdo == -1)
                        return valorHijoDerecho; // [mid+1, R] está contenido en [indicePrincipio, indiceFinal] pero [L, mid] no
                else if (valorHijoDerecho == -1)
                        return valorHijoIzquierdo; // [L, mid] está contenido en [indicePrincipio, indiceFinal] pero [mid+1, R] no
                else
                        return funcion(valorHijoIzquierdo, valorHijoDerecho); // [indicePrincipio, indiceFinal] tiene parte en ambas mitades
        }
}

void cambio(int indiceNodo, int L, int R, int indiceCambiado, vector<ll> & arbol, vector<ll> & lista, ll nuevoValor) {
    // cambia el índice indiceCambiado de la lista original a nuevoValor
    // hay que llamarla como cambio(1, 0, N-1, indiceCambiado, arbol, lista, nuevoValor) desde fuera de la función
    if (indiceCambiado < L || indiceCambiado > R)
        return; // indiceCambiado no está en [L, R]
    if (L == R) {
        // hemos llegado a la hoja del árbol que corresponde a indiceCambiado
        arbol[indiceNodo] = lista[indiceCambiado] = nuevoValor;
    } else {
        // estamos en un intervalo que contiene indiceCambiado
        int mid = L + (R-L)/2; // encontramos el punto intermedio
        // recorremos los dos subárboles, encontrando cuál de ellos contiene el índiceCambiado
        cambio(hijoIzquierdo(indiceNodo), L, mid, indiceCambiado, arbol, lista, nuevoValor);
```

```
67          cambio(hijoDerecho(indiceNodo), mid + 1, R, indiceCambiado, arbol, lista,
            ↪  nuevoValor);
68          // recalculamos el valor sobre [L, R]
69          arbol[indiceNodo] = funcion(arbol[hijoIzquierdo(indiceNodo)],
            ↪  arbol[hijoDerecho(indiceNodo)]);
70      }
71  }
72
73  int main() {
74          ios::sync_with_stdio(false);
75          cin.tie(NULL);
76          int n, q;
77          cin >> n >> q;
78          vector<ll> nums(n);
79          for (int i = 0; i < n; i++)
80                  cin >> nums[i];
81          vector<ll> arbol(4*n, 0);
82          construirArbol(1, 0, n-1, arbol, nums);
83          while(q--) {
84                  int tipo;
85                  cin >> tipo;
86                  if (tipo == 1) {
87                          int k, u;
88                          cin >> k >> u;
89                          cambio(1, 0, n-1, k-1, arbol, nums, u);
90                  } else {
91                          int a, b;
92                          cin >> a >> b;
93                          cout << conseguirValor(1, 0, n-1, a-1, b-1, arbol) <<
                            ↪  '\n';
94                  }
95          }
96      return 0;
97  }
```

Dynamic Range Minimum Queries

Teoría

- Árboles de segmentos

Solución

Las observaciones de este problema son similares a las del primero, solo que la función a usar es el mínimo.

Código

C++

```cpp
#include <vector>
#include <iostream>
using namespace std;

int hijoIzquierdo(int p) {
        return p << 1; // forma rápida de calcular 2p
}

int hijoDerecho(int p) {
        return (p << 1)+1; // forma rápida de calcular 2p + 1
}

int funcion(int valorIzq, int valorDer) {
        return min(valorIzq, valorDer);
}

void construirArbol(int indiceNodo, int L, int R, vector<int> & arbol, vector<int>
↪  & lista) {
        // construye el árbol de segmentos entre los índices [L, R] de la lista de
        ↪  forma recursiva
        // hay que llamarla como (1, 0, N-1, arbol, lista) desde fuera de la
        ↪  función
        if (L < R) {
                int mid = L + (R-L)/2;
                construirArbol(hijoIzquierdo(indiceNodo), L, mid, arbol, lista);
                ↪  // construimos el subárbol izquierdo
                construirArbol(hijoDerecho(indiceNodo), mid + 1, R, arbol, lista);
                ↪  // construimos el subárbol derecho
                arbol[indiceNodo] = funcion(arbol[hijoIzquierdo(indiceNodo)],
                ↪  arbol[hijoDerecho(indiceNodo)]); // los juntamos
        } else {
                arbol[indiceNodo] = lista[L]; // L = R -> subárbol de un solo
                ↪  nodo: cogemos ese índice de la lista
        }
}

int conseguirValor(int indiceNodo, int L, int R, int indicePrincipio, int
↪  indiceFinal, vector<int> & arbol) {
```

```
31          // encuentra el valor de la función sobre el rango [indicePrincipio,
            ↪  indiceFinal] de la lista original
32          // hay que llamarla como conseguirValor(1, 0, N-1, indicePrincipio,
            ↪  indiceFinal, arbol) desde fuera de la función
33          if (L >= indicePrincipio && R <= indiceFinal) {
34                  return arbol[indiceNodo]; // [L, R] está contenido en
                    ↪  [indicePrincipio, indiceFinal]
35          } else if (indicePrincipio > R || indiceFinal < L) {
36                  // No hay ningún índice en común entre [L, R] y [indicePrincipio,
                    ↪  indiceFinal]
37                  return -1; // si hay alguna posibilidad de una suma negativa,
                    ↪  tendremos que elegir otro valor, como menos infinito
38          } else {
39                  int mid = L + (R-L)/2; // encontramos el punto intermedio
40                  // conseguimos el valor de la función sobre la intersección entre
                    ↪  [L, mid] y [indicePrincipio, indiceFinal]
41                  int valorHijoIzquierdo = conseguirValor(hijoIzquierdo(indiceNodo),
                    ↪  L, mid, indicePrincipio, indiceFinal, arbol);
42                  // conseguimos el valor de la función sobre la intersección entre
                    ↪  [mid+1, R] y [indicePrincipio, indiceFinal]
43                  int valorHijoDerecho = conseguirValor(hijoDerecho(indiceNodo), mid
                    ↪  + 1, R, indicePrincipio, indiceFinal, arbol);
44                  if (valorHijoIzquierdo == -1)
45                          return valorHijoDerecho; // [mid+1, R] está contenido en
                            ↪  [indicePrincipio, indiceFinal] pero [L, mid] no
46                  else if (valorHijoDerecho == -1)
47                          return valorHijoIzquierdo; // [L, mid] está contenido en
                            ↪  [indicePrincipio, indiceFinal] pero [mid+1, R] no
48                  else
49                          return funcion(valorHijoIzquierdo, valorHijoDerecho); //
                            ↪  [indicePrincipio, indiceFinal] tiene parte en ambas
                            ↪  mitades
50          }
51  }
52
53  void cambio(int indiceNodo, int L, int R, int indiceCambiado, vector<int> & arbol,
    ↪  vector<int> & lista, int nuevoValor) {
54      // cambia el índice indiceCambiado de la lista original a nuevoValor
55      // hay que llamarla como cambio(1, 0, N-1, indiceCambiado, arbol, lista,
        ↪  nuevoValor) desde fuera de la función
56      if (indiceCambiado < L || indiceCambiado > R)
57          return; // indiceCambiado no está en [L, R]
58      if (L == R) {
59          // hemos llegado a la hoja del árbol que corresponde a indiceCambiado
60          arbol[indiceNodo] = lista[indiceCambiado] = nuevoValor;
61      } else {
62          // estamos en un intervalo que contiene indiceCambiado
63          int mid = L + (R-L)/2; // encontramos el punto intermedio
64          // recorremos los dos subárboles, encontrando cuál de ellos contiene el
            ↪  índiceCambiado
65          cambio(hijoIzquierdo(indiceNodo), L, mid, indiceCambiado, arbol, lista,
            ↪  nuevoValor);
66          cambio(hijoDerecho(indiceNodo), mid + 1, R, indiceCambiado, arbol, lista,
            ↪  nuevoValor);
67          // recalculamos el valor sobre [L, R]
```

```
68          arbol[indiceNodo] = funcion(arbol[hijoIzquierdo(indiceNodo)],
            ↪  arbol[hijoDerecho(indiceNodo)]);
69      }
70  }
71
72  int main() {
73          ios::sync_with_stdio(false);
74          cin.tie(NULL);
75          int n, q;
76          cin >> n >> q;
77          vector<int> nums(n);
78          for (int i = 0; i < n; i++)
79                  cin >> nums[i];
80          vector<int> arbol(4*n, 0);
81          construirArbol(1, 0, n-1, arbol, nums);
82          while(q--) {
83                  int tipo;
84                  cin >> tipo;
85                  if (tipo == 1) {
86                          int k, u;
87                          cin >> k >> u;
88                          cambio(1, 0, n-1, k-1, arbol, nums, u);
89                  } else {
90                          int a, b;
91                          cin >> a >> b;
92                          cout << conseguirValor(1, 0, n-1, a-1, b-1, arbol) <<
                            ↪  '\n';
93                  }
94          }
95      return 0;
96  }
```

Arranging Shoes

Teoría

- Árboles de segmentos
- Algoritmos voraces

Solución

Este problema, el primero de la IOI de 2019, se resolvía ordenando los zapatos de forma voraz. Para obtener los 100 puntos, había que hacerlo usando un árbol de segmentos o de Fenwick, en vez de la solución naïve cuadrática. Las soluciones oficiales explican en detalle este algoritmo.

Trata de programar la solución descrita en el pdf del link antes de leer el código que la implementa.

Street Lamps

Teoría

- Árboles de Fenwick

Solución

Este problema es una modificación sobre los árboles de Fenwick. Antes de ver la solución, lee el addendum si no lo has hecho todavía, y trata de resolverlo o de obtener puntos parciales.

Puedes encontrar soluciones oficiales en C++ para todos los problemas de esa competición **aquí**.

Salary Queries

Teoría

- Árboles de segmentos
- Bucket sort

Solución

En vez de usar un nodo en el árbol por empleado, que nos daría MLE, los guardaremos en cubos. Es decir, partiremos el rango de salarios posible en una serie de cubos/cajones y meteremos cada salario en el cubo correspondiente. Así, podremos responder queries mucho más rápido, ya que trataremos a cada cubo como un solo valor de cara al árbol de segmentos, entrando solo en los cubos correspondientes cuando cambiamos un salario.

Código

C++

```cpp
#include <vector>
#include <iostream>
#include <map>
#include <unordered_map>
#include <cstring>
using namespace std;

const int tamanoCubos = 10000, numeroDeCubos = 1e9/tamanoCubos;
int N, sal[200000], arbol[4*numeroDeCubos], frecuencia[numeroDeCubos];
map<int, map<int, int>> buck;

int cuboAlQuePertenece(int A) {
        if (A%tamanoCubos == 0) return A/tamanoCubos - 1;
        else return A/tamanoCubos;
}

int hijoIzquierdo(int p) {
        return p << 1;
}

int hijoDerecho(int p) {
        return (p << 1)+1;
}

void construirArbol(int indiceNodo, int L, int R) {
        if (L < R) {
                int mid = L + (R-L)/2;
                construirArbol(hijoIzquierdo(indiceNodo), L, mid);
                construirArbol(hijoDerecho(indiceNodo), mid + 1, R);
                arbol[indiceNodo] = arbol[hijoIzquierdo(indiceNodo)] +
                ↪  arbol[hijoDerecho(indiceNodo)];
        } else {
                arbol[indiceNodo] = frecuencia[L];
        }
```

```
}

void init() {
        memset(frecuencia, 0, sizeof(frecuencia));
        int b;
        for (int i = 0; i < N; i++) {
                b = cuboAlQuePertenece(sal[i]);
                frecuencia[b]++;
                // guardamos cada salario dentro del cubo al que corresponde como un número entre 1 y el tamaño del cubo
                if (sal[i] % tamanoCubos)
                        buck[b][sal[i]%tamanoCubos]++;
                else
                        buck[b][tamanoCubos]++;
        }
        construirArbol(1, 0, numeroDeCubos-1);
}

int conseguirValor(int indiceNodo, int L, int R, int indicePrincipio, int indiceFinal) {
        if (indicePrincipio > R || indiceFinal < L)
                return 0;
        else if (L >= indicePrincipio && R <= indiceFinal)
                return arbol[indiceNodo];
        else {
                int mid = L + (R-L)/2;
                int valorHijoIzquierdo = conseguirValor(hijoIzquierdo(indiceNodo), L, mid, indicePrincipio, indiceFinal);
                int valorHijoDerecho = conseguirValor(hijoDerecho(indiceNodo), mid + 1, R, indicePrincipio, indiceFinal);
                return valorHijoIzquierdo + valorHijoDerecho;
        }
}

int intervaloDerechaCubo(int B, int lo) {
        // [lo, final del cubo]
        if (buck.find(B) == buck.end() || lo > buck[B].rbegin()->first) return 0;
        auto it = buck[B].lower_bound(lo);
        int ans = 0;
        while(it != buck[B].end()) {
                ans += it->second;
                it++;
        }
        return ans;
}

int intervaloIzquierdaCubo(int B, int hi) {
        // [principio del cubo, hi]
        if (buck.find(B) == buck.end() || hi < buck[B].begin()->first) return 0;
        auto it = buck[B].lower_bound(hi);
        if (it->first > hi) it--;
        int ans = 0;
        while(it != buck[B].begin()) {
                ans += it->second;
                it--;
```

```
85          }
86          return ans + buck[B].begin()->second;
87  }
88
89  int intervaloDentroDeUnCubo(int B, int lo, int hi) {
90          if (buck.find(B) == buck.end() || lo > buck[B].rbegin()->first) return 0;
91          auto it = buck[B].lower_bound(lo);
92          int ans = 0;
93          while(it != buck[B].end() && it->first <= hi) {
94                  ans += it->second;
95                  it++;
96          }
97          return ans;
98  }
99
100 int salariosEnElIntervalo(int lo, int hi) {
101         int posA = cuboAlQuePertenece(lo), posB = cuboAlQuePertenece(hi), numA,
            ↪  numB;
102         // número de A dentro del cubo
103         if (lo % tamanoCubos != 0)
104                 numA = lo % tamanoCubos;
105         else
106                 numA = tamanoCubos;
107         // número de B dentro del cubo
108         if (hi % tamanoCubos != 0)
109                 numB = hi % tamanoCubos;
110         else
111                 numB = tamanoCubos;
112         if (posA == posB)
113                 return intervaloDentroDeUnCubo(posA, numA, numB); // pertenecen al
                    ↪  mismo cubo
114         else if (posA == posB-1)
115                 return intervaloDerechaCubo(posA, numA) +
                    ↪  intervaloIzquierdaCubo(posB, numB); // son de cubos contiguos:
                    ↪  no hay cubos intermedios que sumar
116         else
117                 return conseguirValor(1, 0, numeroDeCubos-1, posA+1, posB-1) +
                    ↪  intervaloDerechaCubo(posA, numA) +
                    ↪  intervaloIzquierdaCubo(posB, numB);
118                 // tenemos cubos intermedios que sumar
119 }
120
121 void cambio(int indiceNodo, int L, int R, int indiceCambiado, int indiceNuevo) {
122     if (indiceNuevo < L || indiceCambiado > R || (indiceCambiado < L && R <
        ↪  indiceNuevo))
123         return;
124     if (L == R) {
125         arbol[indiceNodo] = frecuencia[L];
126     } else {
127         int mid = L + (R-L)/2;
128         cambio(hijoIzquierdo(indiceNodo), L, mid, indiceCambiado, indiceNuevo);
129         cambio(hijoDerecho(indiceNodo), mid + 1, R, indiceCambiado, indiceNuevo);
130         arbol[indiceNodo] = arbol[hijoIzquierdo(indiceNodo)] +
            ↪  arbol[hijoDerecho(indiceNodo)];
131     }
```

```
132 }
133
134 void actualizarSalario(int posicion, int nuevoSalario) {
135         int numA, numB, posA = cuboAlQuePertenece(sal[posicion]), posB =
        ↪  cuboAlQuePertenece(nuevoSalario);
136         // quitamos el viejo salario
137         if (sal[posicion] % tamanoCubos == 0)
138                 numA = tamanoCubos;
139         else
140                 numA = sal[posicion]%tamanoCubos;
141         // añadimos el nuevo salario
142         if (nuevoSalario % tamanoCubos == 0)
143                 numB = tamanoCubos;
144         else
145                 numB = nuevoSalario % tamanoCubos;
146         // si están en el mismo cubo, es fácil
147         if (posA == posB) {
148                 buck[posA][numA]--;
149                 buck[posA][numB]++;
150                 sal[posicion] = nuevoSalario;
151                 return;
152         }
153         // si no, hacemos unos cambios más
154         frecuencia[posA]--;
155         frecuencia[posB]++;
156         buck[posA][numA]--;
157         buck[posB][numB]++;
158         sal[posicion] = nuevoSalario;
159         if (posA > posB)
160                 swap(posA, posB);
161         cambio(1, 0, numeroDeCubos-1, posA, posB);
162 }
163
164 int main() {
165         ios::sync_with_stdio(false);
166         cin.tie(NULL);
167         int Q, a, b;
168         char c;
169         cin >> N >> Q;
170         for (int i = 0; i < N; i++) cin >> sal[i];
171         init();
172         while(Q--) {
173                 cin >> c >> a >> b;
174                 if (c == '?')
175                         cout << salariosEnElIntervalo(a, b) << "\n";
176                 else
177                         actualizarSalario(a-1, b);
178         }
179         return 0;
180 }
```