Apuntes OIFem 2020

Estructuras de Datos 2

Árboles de Segmentos

Los árboles de segmentos son estructuras para calcular funciones constructivas sobre un rango de valores dentro de una lista. Si un nodo guarda la función sobre los valores de los índices de l a r, su hijo izquierdo la guarda desde l hasta mid, el punto intermedio, y su hijo derecho la guarda desde $mid + 1$ hasta r. Vamos descendiendo de esta forma sobre el árbol hasta llegar a un punto donde l sea igual a r, momento en el que solo guardemos la función sobre un número en concreto y, por tanto, no nos haga falta seguir bajando.

La raíz se encuentra en el índice 1 y, para cada nodo con índice x, su hijo izquierdo tendrá el índice $2x$ y su hijo derecho el índice $2x + 1$. Para una lista de longitud N, hacer un par de cálculos matemáticos nos enseñará cómo, reservar un vector de longitud $4N$ es más que de sobra para guardar el árbol correspondiente.

Los árboles de segmentos tienen la función de ser dinámicos, es decir, en poco tiempo, podremos cambiar el valor de un índice de la lista y que el árbol refleje la función sobre la lista cambiada. En el addendum, se explica cómo hacer estos cambios de forma *vaga*.

Aquí incluyo el código con el ejemplo de la suma, pero se puede adaptar con facilidad a otras funciones. Tened en cuenta que `arbol` tiene que inicializarse a un vector de longitud $4N$ con valores 0 y que `lista` es la lista de números sobre la que hacemos los cálculos (también pueden ser listas que no sean de números).

Código- Árbol de segmentos

```
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 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
```

```
15          // hay que llamarla como (1, 0, N-1, arbol, lista) desde fuera de la
            ↪  función
16          if (L < R) {
17                  int mid = L + (R-L)/2;
18                  construirArbol(hijoIzquierdo(indiceNodo), L, mid, arbol, lista);
                    ↪  // construimos el subárbol izquierdo
19                  construirArbol(hijoDerecho(indiceNodo), mid + 1, R, arbol, lista);
                    ↪  // construimos el subárbol derecho
20                  arbol[indiceNodo] = funcion(arbol[hijoIzquierdo(indiceNodo)],
                    ↪  arbol[hijoDerecho(indiceNodo)]); // los juntamos
21          } else {
22                  arbol[indiceNodo] = lista[L]; // L = R -> subárbol de un solo
                    ↪  nodo: cogemos ese índice de la lista
23          }
24  }
25
26  int conseguirValor(int indiceNodo, int L, int R, int indicePrincipio, int
    ↪  indiceFinal, vector<int> & arbol) {
27          // encuentra el valor de la función sobre el rango [indicePrincipio,
            ↪  indiceFinal] de la lista original
28          // hay que llamarla como conseguirValor(1, 0, N-1, indicePrincipio,
            ↪  indiceFinal, arbol) desde fuera de la función
29          if (L >= indicePrincipio && R <= indiceFinal) {
30                  return arbol[indiceNodo]; // [L, R] está contenido en
                    ↪  [indicePrincipio, indiceFinal]
31          } else if (indicePrincipio > R || indiceFinal < L) {
32                  // No hay ningún índice en común entre [L, R] y [indicePrincipio,
                    ↪  indiceFinal]
33                  return -1; // si hay alguna posibilidad de una suma negativa,
                    ↪  tendremos que elegir otro valor, como menos infinito
34          } else {
35                  int mid = L + (R-L)/2; // encontramos el punto intermedio
36                  // conseguimos el valor de la función sobre la intersección entre
                    ↪  [L, mid] y [indicePrincipio, indiceFinal]
37                  int valorHijoIzquierdo = conseguirValor(hijoIzquierdo(indiceNodo),
                    ↪  L, mid, indicePrincipio, indiceFinal, arbol);
38                  // conseguimos el valor de la función sobre la intersección entre
                    ↪  [mid+1, R] y [indicePrincipio, indiceFinal]
39                  int valorHijoDerecho = conseguirValor(hijoDerecho(indiceNodo), mid
                    ↪  + 1, R, indicePrincipio, indiceFinal, arbol);
40                  if (valorHijoIzquierdo == -1)
41                          return valorHijoDerecho; // [mid+1, R] está contenido en
                            ↪  [indicePrincipio, indiceFinal] pero [L, mid] no
42                  else if (valorHijoDerecho == -1)
43                          return valorHijoIzquierdo; // [L, mid] está contenido en
                            ↪  [indicePrincipio, indiceFinal] pero [mid+1, R] no
44                  else
45                          return funcion(valorHijoIzquierdo, valorHijoDerecho); //
                            ↪  [indicePrincipio, indiceFinal] tiene parte en ambas
                            ↪  mitades
46          }
47  }
48
49  void cambio(int indiceNodo, int L, int R, int indiceCambiado, vector<int> & arbol,
    ↪  vector<int> & lista, int 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);
        cambio(hijoDerecho(indiceNodo), mid + 1, R, indiceCambiado, arbol, lista,
        ↪  nuevoValor);
        // recalculamos el valor sobre [L, R]
        arbol[indiceNodo] = funcion(arbol[hijoIzquierdo(indiceNodo)],
        ↪  arbol[hijoDerecho(indiceNodo)]);
    }
}
```

Propagación *vaga*

Imaginad que en el índice 0 de la lista contiene el valor 3 y lo cambiamos al valor 4. Esto significa que hay que cambiar una cantidad proporcional al logaritmo de N de nodos. Nada más hacer este cambio, devolvemos el valor de ese índice al 3 original. Hay que volver a cambiar todos esos nodos, dejándolos como al principio. La propagación vaga es una técnica que nos permite solo cambiar los valores de nodos cuando los vamos a usar, evitando hacer estos cambios absurdos. Consiste en dejar apuntados los cambios para más adelante y, al calcular la función sobre un nodo, mirar si hay un apunte que tener en cuenta, pasándolo a los hijos.

Los apuntes son a tiempo pasado, es decir, que hacemos un apunte al nodo x cuando hacemos un cambio al nodo x. Son una forma de recordar los cambios que hemos hecho al nodo x para cuando vayamos a usar los valores de sus hijos, haciéndoles estos cambios a ellos. Esto es lo que hace la función `propagar()`: hace los cambios a los hijos y hace los apuntes de estos cambios, posteriormente borrando el apunte de x al ya haber cambiado a los hijos.

Código- Máximo del Segmento

```
#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) {
```

```
14      return max(valorIzq, valorDer); // en este caso, queremos el máximo del rango
15  }
16
17  void construirArbol(int indiceNodo, int L, int R, vector<int> & arbol, vector<int>
    ↪  & lista) {
18      // construye el árbol de segmentos entre los índices [L, R] de la lista de
        ↪  forma recursiva
19      // hay que llamarla como (1, 0, N-1, arbol, lista) desde fuera de la función
20      if (L < R) {
21          int mid = L + (R-L)/2;
22          construirArbol(hijoIzquierdo(indiceNodo), L, mid, arbol, lista); //
            ↪  construimos el subárbol izquierdo
23          construirArbol(hijoDerecho(indiceNodo), mid + 1, R, arbol, lista); //
            ↪  construimos el subárbol derecho
24          arbol[indiceNodo] = funcion(arbol[hijoIzquierdo(indiceNodo)],
            ↪  arbol[hijoDerecho(indiceNodo)]); // los juntamos
25      } else {
26          arbol[indiceNodo] = lista[L]; // L = R -> subárbol de un solo nodo:
            ↪  cogemos ese índice de la lista
27      }
28  }
29
30  void propagar(int indiceNodo, int L, int R, vector<int> & arbol, vector<int> &
    ↪  vaga) {
31      if (L != R) {
32          arbol[hijoIzquierdo(indiceNodo)] += vaga[indiceNodo];
33          vaga[hijoIzquierdo(indiceNodo)] += vaga[indiceNodo];
34          arbol[hijoDerecho(indiceNodo)] += vaga[indiceNodo];
35          vaga[hijoDerecho(indiceNodo)] += vaga[indiceNodo];
36          vaga[indiceNodo] = 0;
37      } else vaga[indiceNodo] = 0;
38  }
39
40  int conseguirValor(int indiceNodo, int L, int R, int indicePrincipio, int
    ↪  indiceFinal, vector<int> & arbol, vector<int> & vaga) {
41      // encuentra el valor de la función sobre el rango [indicePrincipio,
        ↪  indiceFinal] de la lista original
42      // hay que llamarla como conseguirValor(1, 0, N-1, indicePrincipio,
        ↪  indiceFinal, arbol) desde fuera de la función
43      if (L >= indicePrincipio && R <= indiceFinal) {
44          propagar(indiceNodo, L, R, arbol, vaga);
45          return arbol[indiceNodo]; // [L, R] está contenido en [indicePrincipio,
            ↪  indiceFinal]
46      } else if (indicePrincipio > R || indiceFinal < L) {
47          // No hay ningún índice en común entre [L, R] y [indicePrincipio,
            ↪  indiceFinal]
48          return -1e9;
49      } else {
50          int mid = L + (R-L)/2; // encontramos el punto intermedio
51          propagar(indiceNodo, L, R, arbol, vaga);
52          // conseguimos el valor de la función sobre la intersección entre [L, mid]
            ↪  y [indicePrincipio, indiceFinal]
53          int valorHijoIzquierdo = conseguirValor(hijoIzquierdo(indiceNodo), L, mid,
            ↪  indicePrincipio, indiceFinal, arbol, vaga);
```

```
54          // conseguimos el valor de la función sobre la intersección entre [mid+1,
            ↪  R] y [indicePrincipio, indiceFinal]
55          int valorHijoDerecho = conseguirValor(hijoDerecho(indiceNodo), mid + 1, R,
            ↪  indicePrincipio, indiceFinal, arbol, vaga);
56          if (valorHijoIzquierdo == -1e9)
57              return valorHijoDerecho; // [mid+1, R] está contenido en
                ↪  [indicePrincipio, indiceFinal] pero [L, mid] no
58          else if (valorHijoDerecho == -1e9)
59              return valorHijoIzquierdo; // [L, mid] está contenido en
                ↪  [indicePrincipio, indiceFinal] pero [mid+1, R] no
60          else
61              return funcion(valorHijoIzquierdo, valorHijoDerecho); //
                ↪  [indicePrincipio, indiceFinal] tiene parte en ambas mitades
62      }
63  }
64
65  void sumar(int indiceNodo, int L, int R, int indInicio, int indFinal, int suma,
    ↪  vector<int> & vaga, vector<int> & arbol) {
66      if (indFinal < L || indInicio > R)
67          return; // no hay ningún índice en común entre [indInicio, indFinal] y [L,
            ↪  R]
68      if (L >= indInicio && R <= indFinal) {
69          // [L, R] está incluido en [indInicio, indFinal]: sumamos la suma al
            ↪  mínimo y lo dejamos guardado en vaga
70          vaga[indiceNodo] += suma;
71          arbol[indiceNodo] += suma;
72      } else {
73          // parte de [L, R] está incluido en [indInicio, indFinal]
74          propagar(indiceNodo, L, R, arbol, vaga); // propagamos los apuntes a los
            ↪  hijos
75          int mid = L + (R-L)/2;
76          // buscamos qué sección de [L, R] es la incluida y actualizamos
77          sumar(hijoIzquierdo(indiceNodo), L, mid, indInicio, indFinal, suma, vaga,
            ↪  arbol);
78          sumar(hijoDerecho(indiceNodo), mid+1, R, indInicio, indFinal, suma, vaga,
            ↪  arbol);
79          // calculamos el nuevo valor del nodo padre
80          arbol[indiceNodo] = funcion(arbol[hijoIzquierdo(indiceNodo)],
            ↪  arbol[hijoDerecho(indiceNodo)]);
81      }
82  }
83
84  int main() {
85      ios::sync_with_stdio(false);
86      cin.tie(NULL);
87      int T, N, Q, k, l, r, x;
88      cin >> T;
89      while(T--) {
90          cin >> N >> Q;
91          vector<int> lista(N);
92          for (int i = 0; i < N; i++)
93              cin >> lista[i];
94          vector<int> arbol(4*N, -1e9);
95          vector<int> vaga(4*N, 0);
96          construirArbol(1, 0, N-1, arbol, lista);
```

```
97          while(Q--) {
98              cin >> k >> l >> r;
99              l--;
100             r--;
101             if (k == 1) {
102                 cin >> x;
103                 sumar(1, 0, N-1, l, r, x, vaga, arbol);
104             } else {
105                 cout << conseguirValor(1, 0, N-1, l, r, arbol, vaga) << '\n';
106             }
107         }
108     }
109     return 0;
110 }
```