Estructuras de datos avanzadas I

Point-update y range-query

Marco Antonio Gómez Martín

Facultad de Informática - UCM

Estructuras de datos avanzadas

- Árboles de Fenwick (Binary indexed trees o Fenwick trees)
- Árboles de segmento / segment trees

Range Sum Query (RSQ)

Dado un vector de enteros, ¿cuál es la suma de `v[a..b]`?
¿Cómo lo plantearías si sabes que se realizarán varias consultas?

- Opción 1: recorrer el vector con un bucle, $<\mathcal{O}(1), \mathcal{O}(n)>$
- Opción 2: vector de sumas acumuladas, $<\mathcal{O}(n), \mathcal{O}(1)>$

Range Sum Query (RSQ)

Dado un vector de enteros, dos operaciones: dar la suma de `v[a..b]`, y cambiar el valor de `v[k]`.

- Opción 1: $<\mathcal{O}(1), \mathcal{O}(n), \mathcal{O}(1)>$
- Opción 2: $<\mathcal{O}(n), \mathcal{O}(1), \mathcal{O}(n)>$

Las operaciones anteriores se conocen como *point-update* (solo se cambia un valor) y *range-query* (se pregunta por la propiedad de un rango de valores).

Árboles de Fenwick

- Estructura de datos inventada en 1994 por Peter M. Fenwick.
- También conocidos como BIT (*binary indexed trees*).
- Útil para almacenar vectores de frecuencias acumulativas *dinámicas*.
- Aunque se llaman *árboles*, sirven para guardar vectores de enteros y obtener rápidamente la suma de un rango de valores.
- Para guardar n enteros, utilizan un vector de $n + 1$ enteros.
- Se llaman *árboles* por la forma en la que se distribuyen los datos (que no explicaremos...).
- Utiliza operaciones de bits sobre la posición a la que se quiere acceder.

Árboles de Fenwick

Operaciones:

- `init(n)`: inicializa la estructura para guardar n valores (referenciados entre 1 y n), todos ellos a cero. $\mathcal{O}(n)$
- `add(pos, val)`: incrementa en la posición pos la cantidad indicada $\mathcal{O}(log\ n)$ (*point-update*)
- `getSum(b)`: devuelve la suma de todos los valores hasta b $\mathcal{O}(log\ n)$ (*range-query*).

Gracias a `getSum(b)`, se puede sacar la suma entre a..b son la resta `getSum(b) - getSum(a-1)`.

Árboles de Fenwick

```
class FenwickTree {
  vector<int> ft;

public:
  FenwickTree(int n) { ft.assign(n+1, 0); }

  int getSum(int b) {
   int ret = 0;
   while (b) {
    ret += ft[b]; b -= (b & -b);
   }
   return ret;
  }

  void add(int pos, int val) {
   while (pos < ft.size()) {
    ft[pos] += val; pos += (pos & -pos);
   }
  }
```

Se pueden añadir tres métodos adicionales cómodos en algunos casos:

```
class FenwickTree {
public:

  [...]

  int getSum(int a, int b) {
    return getSum(b) - getSum(a - 1);
  }

  int getValue(int pos) {
   return getSum(pos) - getSum(pos - 1);
  }

  void setValue(int pos, int val) {
   add(pos, val - getValue(pos));
  }
};
```

Árboles de Fenwick

- La implementación anterior puede adaptarse fácilmente para otras operaciones distintas a la suma, aunque **es posible que únicamente sea posible el range-query con el intervalo [1..b]**.
- Esa limitación hace inservibles a los árboles de Fenwick en muchos problemas.
- Pero en muchos otros siguen siendo útiles, y se programan/teclean muy rápidamente.

Range Minimum Query (RMQ)

Dado un vector de enteros, ¿cuál es *el mínimo valor* de `v[a..b]`? ¿Si se realizan varias consultas?

- Opción 1: recorrer el vector con un bucle, $<\mathcal{O}(1), \mathcal{O}(n)>$
- Opción 2: ... ¡no hay forma de adaptar la idea de vector de sumas acumuladas!

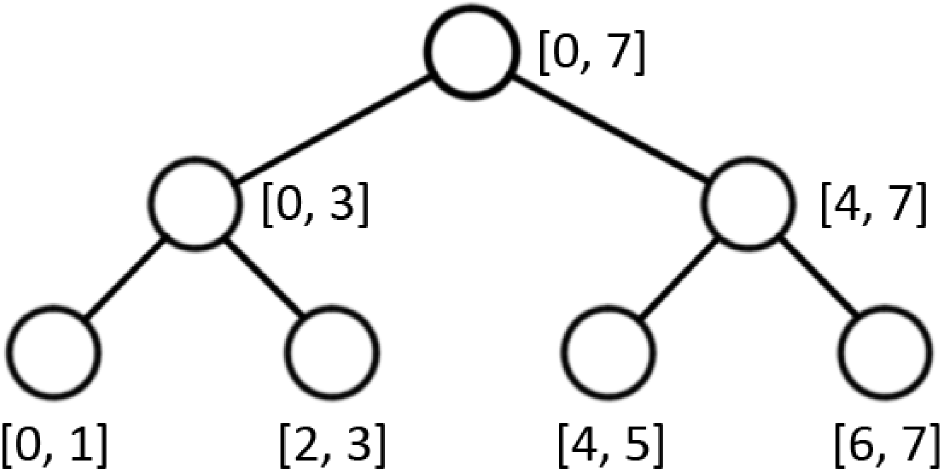
¿Y si en lugar del mínimo fuera el máximo? ¿Y si tuviéramos una operación más para *cambiar* alguno de los valores del vector?

Segment Trees

- Árbol binario para almacenar la información de un vector (normalmente de enteros).
- Hay una hoja por cada elemento del vector.
- Los nodos internos almacenan la información *agregada* de un segmento del vector (todas las hojas accesibles desde él).
- El hijo izquierdo y derecho de un nodo representa la mitad izquierda o derecha del segmento del padre.
    - La raíz el vector completo
    - El hijo izquierdo de la raíz, la primera mitad
    - El hijo derecho de la raíz, la segunda mitad.

Segment Trees

Ejemplo: primeros niveles del árbol para un vector de longitud 8:



[Se omite el nivel con las hojas]

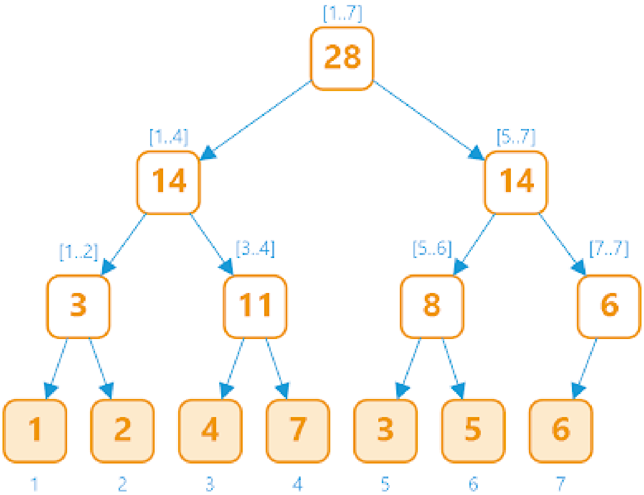
Segment Trees

Estructura más general que los árboles de Fenwick. Pueden usarse para RSQ, RMQ, ... pero son más largos de programar.

La información agregada puede ser:

- La suma de todos los valores en el intervalo. Para RSQ
- El mínimo/máximo de todos los valores del intervalo. Para RMQ
- ...

Segment Trees



Pseudo-código de una consulta

```
int query(nodo, a, b) { // query = rsq en este caso

    if el rango del nodo está incluido en [a..b]
       return nodo->val;
    if el rango del nodo está completamente
       fuera de [a..b]
       return 0; // Elemento neutro de la operación

    // Mejorable: el resultado podría afectar solo
    // a un hijo; podríamos ahorrarnos una de las
    // llamadas
    return query(nodo->iz, l, r) + // Operación
           query(nodo->dr, l, r);
}
```

Pseudo-código de un cambio

```
int update(nodo, pos, val) {

    if pos está fuera del rango del nodo
        return;
    if el rango del nodo es [pos..pos]
        nodo->val = val;
        return;

    // De uno de los dos estará fuera (¡mejorable!)
    update(nodo->iz, pos, val);
    update(nodo->dr, pos, val);

    nodo->val = nodo->iz->val + nodo->dr->val;
}
```

Implementación

- En lugar de nodos, se usa un array con los nodos guardados "por niveles".
- La raíz la colocamos en la posición 1.
- Dado un nodo i, el hijo izquierdo estará en $2 * i$ y el derecho en $2 * i + 1$.

<table>
<tr><td colspan="16">1: [0, 16)</td></tr>
<tr><td colspan="8">2: [0, 8)</td><td colspan="8">3: [8, 16)</td></tr>
<tr><td colspan="4">4: [0, 4)</td><td colspan="4">5: [4, 8)</td><td colspan="4">6: [8, 12)</td><td colspan="4">7: [12, 16)</td></tr>
<tr><td colspan="2">8: [0, 2)</td><td colspan="2">9: [2, 4)</td><td colspan="2">10: [4, 6)</td><td colspan="2">11: [6, 8)</td><td colspan="2">12: [8, 10)</td><td colspan="2">13: [10, 12)</td><td colspan="2">14: [12, 14)</td><td colspan="2">15: [14, 16)</td></tr>
<tr><td>16: 0</td><td>17: 1</td><td>18: 2</td><td>19: 3</td><td>20: 4</td><td>21: 5</td><td>22: 6</td><td>23: 7</td><td>24: 8</td><td>25: 9</td><td>26: 10</td><td>27: 11</td><td>28: 12</td><td>29: 13</td><td>30: 14</td><td>31: 15</td></tr>
</table>

Implementación

- No manejaremos nodos con valores e información del intervalo al que representan.
- Las funciones recursivas recibirán siempre al menos tres valores:
  - El id del nodo ($=$ posición que ocupa en el array).
  - El rango de [L,R] al que representa ese id.

Implementación

```
class SegmentTree {
  vector<int> st;
  int tam; // Número de hojas que manejamos
public:

  // Tamaño máximo que podremos guardar
  // (número de hojas).
  // Antes de las consultas, se necesita rellenar
  // con los datos iniciales usando build().
  SegmentTree(int maxN) {
   st.reserve(4 * maxN + 10);
  }

  [... Siguientes transparencias ...]
}
```

Implementación

```
int query(int a, int b) {
 return query(1, 0, tam-1, a, b);
}

int query(int vertex, int L, int R, int i, int j) {

  if (i > R || j < L) {
    return 0;
  }

  if (L >= i && R <= j)
    // Segmento completamente dentro de la consulta
    return st[vertex];

  int mitad = (L + R) / 2;

  return query(2*vertex, L, mitad, i, j) +
           query(2*vertex+1, mitad+1, R, i, j);
}
```

Implementación

```
void update(int pos, int newVal) {
 update(1, 0, tam-1, pos, newVal);
}
void update(int vertex, int l, int r,
            int pos, int newVal) {
 if ((pos < l) || (r < pos)) return;

 if (l == r) {
  st[vertex] = newVal;
  return;
 }

 int m = (l+r) / 2;
 if ((l <= pos) && (pos <= m))
  update(2 * vertex, l, m, pos, newVal);
 else
  update(2 * vertex + 1, m+1, r, pos, newVal);
 st[vertex] = st[2*vertex] + st[2*vertex + 1];
}
```

Implementación

```
void build(int *values, int n) {
 tam = n;
 build(values, 1, 0, n-1);
}

void build(int *values, int p, int l, int r) {
 if (l == r) {
  st[p] = values[l];
  return;
 }
 int m = (l+r)/2;
 build(values, 2*p, l, m);
 build(values, 2*p+1, m+1, r);
 st[p] = st[2*p]+st[2*p+1];
}
```

Ejercicio propuesto

12086 - Potentiometers