Estructuras de datos avanzadas II

Range-update

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

Point-update / range-query: ejemplo con RSQ

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

- Opción 1: recorrer el vector entre a y b con un bucle
  $<\mathcal{O}(1), \mathcal{O}(n), \mathcal{O}(1)>$
- Opción 2:Vector con sumas acumuladas
  $<\mathcal{O}(n), \mathcal{O}(1), \mathcal{O}(n)>$
- Opción 3: Fenwick o segment trees
  $<\mathcal{O}(n), \mathcal{O}(log\ n), \mathcal{O}(log\ n)>$

Range-update

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

- Opción 1: recorrer el vector entre a y b con un bucle
  $<\mathcal{O}(1), \mathcal{O}(n), \mathcal{O}(1)>$
- Opción 2:Vector con sumas acumuladas
  $<\mathcal{O}(n), \mathcal{O}(1), \mathcal{O}(n)>$
- Opción 3: Fenwick o segment trees
  $<\mathcal{O}(n), \mathcal{O}(log\ n), \mathcal{O}(log\ n)>$

¿Y si añadimos una operación para sumar K en *todos los elementos*, `v[a..b]`?

Range-update

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

- Opción 1: recorrer el vector entre a y b con un bucle
  $<\mathcal{O}(1), \mathcal{O}(n), \mathcal{O}(1), \mathcal{O}(n)>$
- Opción 2:Vector con sumas acumuladas
  $<\mathcal{O}(n), \mathcal{O}(1), \mathcal{O}(n)>$
- Opción 3: Fenwick o segment trees
  $<\mathcal{O}(n), \mathcal{O}(log\ n), \mathcal{O}(log\ n)>$

¿Y si añadimos una operación para sumar K en *todos los elementos*, `v[a..b]`?

Range-update

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

- Opción 1: recorrer el vector entre a y b con un bucle
  $<\mathcal{O}(1)$, $\mathcal{O}(n)$, $\mathcal{O}(1)$, $\mathcal{O}(n)>$
- Opción 2:Vector con sumas acumuladas
  $<\mathcal{O}(n)$, $\mathcal{O}(1)$, $\mathcal{O}(n)$, $\mathcal{O}(n)>$
- Opción 3: Fenwick o segment trees
  $<\mathcal{O}(n)$, $\mathcal{O}(log\ n)$, $\mathcal{O}(log\ n)>$

¿Y si añadimos una operación para sumar K en *todos los elementos*, `v[a..b]`?

Range-update

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

- Opción 1: recorrer el vector entre a y b con un bucle
  $<\mathcal{O}(1), \mathcal{O}(n), \mathcal{O}(1), \mathcal{O}(n)>$
- Opción 2:Vector con sumas acumuladas
  $<\mathcal{O}(n), \mathcal{O}(1), \mathcal{O}(n), \mathcal{O}(n)>$
- Opción 3: Fenwick o segment trees
  $<\mathcal{O}(n), \mathcal{O}(log\ n), \mathcal{O}(log\ n), \mathcal{O}(n\ log\ n)>$

¿Y si añadimos una operación para sumar K en *todos los elementos*, `v[a..b]`?

Range-update, point query con árboles de Fenwick

Operaciones:

- `init(n)`: inicializa la estructura para guardar n valores (referenciados entre 1 y n), todos ellos a cero.
- `add(a, b, val)`: incrementa las posiciones entre a y b la cantidad indicada (*range-update*)
- `getValue(a)`: devuelve el valor en la posición a (*point-query*).

Range-update, point query con árboles de Fenwick

Operaciones:

- `init(n)`: inicializa la estructura para guardar n valores (referenciados entre 1 y n), todos ellos a cero.
- `add(a, b, val)`: incrementa las posiciones entre a y b la cantidad indicada (*range-update*)
- `getValue(a)`: devuelve el valor en la posición a (*point-query*).

No hay que hacer grandes cambios:

- `add(a, b, val)` se traduce a `add(a, val)` + `add(b+1, -val)`
- `getValue(a)` se traduce a `getSum(a)`

Range-update, range query con 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(a, b, val)`: incrementa las posiciones entre a y b la cantidad indicada $\mathcal{O}(log\ n)$ (*range-update*)
- `getSum(a, b)`: devuelve la suma de los valores entre a y b $\mathcal{O}(log\ n)$ (*rangue-query*).

Range-update, range query con 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(a, b, val)`: incrementa las posiciones entre a y b la cantidad indicada $\mathcal{O}(log\ n)$ (*range-update*)
- `getSum(a, b)`: devuelve la suma de los valores entre a y b $\mathcal{O}(log\ n)$ (*rangue-query*).

No lo veremos, pero también se puede utilizando *dos* árboles de Fenwick.

Range-update con árboles de Fenwick

Adaptar la solución anterior del range-update para operaciones que no sean sumas es....

Range-update con árboles de Fenwick

Adaptar la solución anterior del range-update para operaciones que no sean sumas es...emocionante.

Range-update con árboles de Fenwick

Adaptar la solución anterior del range-update para operaciones que no sean sumas es...emocionante.

Los árboles de segmentos son mucho más versátiles.

Range-update con árboles de segmentos

Con árboles de segmentos se pueden implementar *varias* operaciones de actualización. Ejemplo (UVa 12436 - Rip Van Winkle's Code):

- `init(n)`: inicializa la estructura $\mathcal{O}(n)$
- `getSum(a, b)`: devuelve la suma de los valores entre a y b (permite de forma trivial tener `getSum(a)`).
- `incr(a, b)`: incrementa de forma creciente el segmento `[a..b]`.
- `decr(a,b)`: incrementa de forma decreciente.
- `set(a,b,c)`: asigna `c` a todos los elementos del intervalo.

Range-update con árboles de segmentos

Idea básica:

- Tener operaciones *diferidas*: la operación está aún pendiente de ejecutar en el nodo (y sus descendientes).
- Si la operación puede aplicarse al intervalo completo, se retrasa su ejecución en los hijos.
- En las consultas, si se llega a un nodo con una operación pendiente de aplicar, se aplica.

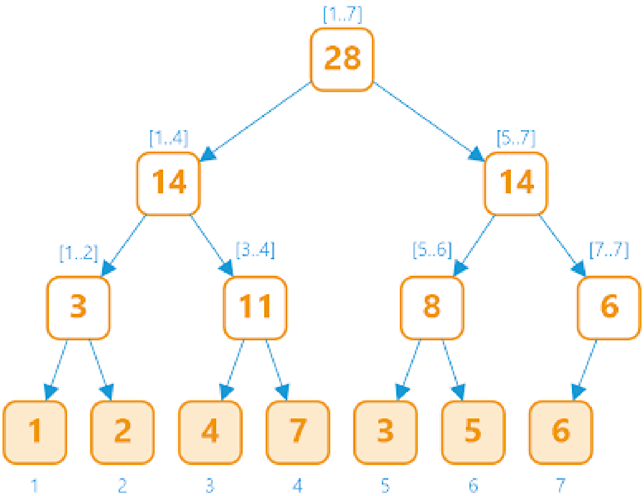
Range-update con árboles de segmentos

Idea básica:

- Tener operaciones *diferidas*: la operación está aún pendiente de ejecutar en el nodo (y sus descendientes).
- Si la operación puede aplicarse al intervalo completo, se retrasa su ejecución en los hijos.
- En las consultas, si se llega a un nodo con una operación pendiente de aplicar, se aplica.

Lazy update

Segment Trees



Pseudo-código de una actualización

```
void updateRange(nodo, a, b, op) {

   if hay una operación pendiente de ejecutar
      resuelve operación pendiente

   if el rango del nodo está incluido en [a..b]
      actualiza nodo actual
      marca la operación pendiente en los hijos

   if el rango del nodo no cae por completo en [a..b]
      # Mismo procedimiento que en pointUpdate
      actualización recursiva del rango en los hijos
      actualización del nodo en base a los hijos
}
```

Pseudo-código de una consulta

```
int query(nodo, a, b) {

    if hay una operación pendiente de ejecutar
       resuelve operación pendiente

    # Resto igual que sin ejecución perezosa

    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) <op> // Operación
           query(nodo->dr, l, r);
}
```

Implementación

- Además del vector con los datos, un vector con la operación pendiente de ejecutar en ese nodo.
- Método para establecer/mezclar la operación pendiente en un nodo (`setLazyUpdate(vertex, op)`).
- Método para resolver/ejecutar/consolidar la operación pendiente en el nodo (`pushLazyUpdate(vertex, op)`).
  - Llama al método anterior para dejar pendiente la operación en los hijos.

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) {

  pushLazyUpdate(vertex, L, R);

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

  if (i<=L && R <= j)
    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 updateRange(int vertex, int L, int R,
                 int a, int b, int op) {
  // Resolvemos posibles operaciones pendientes
  pushLazyUpdate(vertex, L, R);

  if ((b < L) || (R < a)) return;

  // ¿Intervalo afectado por completo?
  if ((a <= L) && (R <= b)) {
    // Nos aplicamos la operación y propagamos la
    // pereza a los hijos. Para evitar copiar/pegar,
    // lo hacemos aplicándonos la pereza, y luego
    // resolviéndola
    setLazyUpdate(vertex, op);
    pushLazyUpdate(vertex, L, R);
    return;
  }
```

Implementación

```
  [... continua ...]

 // Intervalo no afectado por completo. No podemos
 // ser perezosos. Aplicamos la operación en
  // los hijos
 int m = (L + R) / 2;
 updateRange(2*vertex, L, m, a, b, op);
 updateRange(2*vertex + 1, m+1, R, a, b, op);

 // Combinamos
 st[vertex] = st[2*vertex] + st[2*vertex + 1];
}
```

Implementación

Con árbol de segmentos para operaciones de sumas (equivalente al árbol de Fenwick):

- Añadimos un vector nuevo para almacenar el valor pendiente de sumar *a todos los elementos* del rango.
- El `rangeUpdate` recibe el valor que hay que sumar.

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

  [... sigue ...]
```

Implementación

```
[... continua ...]

void setLazyUpdate(int vertex, int value) {
  // Mezclamos
  // Importante +=: el nodo podría tener
  // otras operaciones pendientes anteriores
  lazy[vertex] += value;
}

[... sigue ...]
```

Implementación

```
  [... continua ...]

  void pushLazyUpdate(int vertex, int L, int R) {

    st[vertex] += (R - L + 1)*lazy[vertex];

    if (L != R) {
      // Tenemos hijos
      int m = (L + R) / 2;
      setLazyUpdate(2*vertex, lazy[vertex]);
      setLazyUpdate(2*vertex+1, lazy[vertex]);
    }
    lazy[vertex] = 0;
  }

  [... resto de operaciones ...]
}
```

Ejercicio propuesto

12769 - Kool Konstructions