Taller ProgComp: Track EDD

Fenwick Tree

Gabriel Carmona Tabja

Universidad Técnica Federico Santa María,
Università di Pisa

May 20, 2024

Part I

Range Query con Update

RSQ - Range Sum Query

Contextualización

Tenemos un arreglo de tamaño *N*, este arreglo posee números. Dentro de este programa existen dos tipos de consultas:

1. Determinar el valor de la suma dentro de un rango *i* y *j*, donde $i <= j$
2. Actualizar el valor que se encuentra en la posición *i*

RSQ - RANGE SUM QUERY

Contextualización

Tenemos un arreglo de tamaño N, este arreglo posee números. Dentro de este programa existen dos tipos de consultas:

1. Determinar el valor de la suma dentro de un rango i y j, donde $i <= j$
2. Actualizar el valor que se encuentra en la posición i

Opciones

- Prefix Sum: query $O(1)$, update $O(n)$ y memoria n words

RSQ - Range Sum Query

Contextualización

Tenemos un arreglo de tamaño N, este arreglo posee números. Dentro de este programa existen dos tipos de consultas:

1. Determinar el valor de la suma dentro de un rango i y j, donde $i <= j$
2. Actualizar el valor que se encuentra en la posición i

Opciones

- Prefix Sum: query $O(1)$, update $O(n)$ y memoria n words
- SQRT Decomp: query $O(\sqrt{n})$, update $O(\sqrt{n})$ y memoria $\sqrt{n} + \sqrt{n} \cdot \sqrt{n}$ words

RSQ - Range Sum Query

Contextualización

Tenemos un arreglo de tamaño N, este arreglo posee números. Dentro de este programa existen dos tipos de consultas:

1. Determinar el valor de la suma dentro de un rango i y j, donde $i <= j$
2. Actualizar el valor que se encuentra en la posición i

Opciones

- Prefix Sum: query $O(1)$, update $O(n)$ y memoria n words
- SQRT Decomp: query $O(\sqrt{n})$, update $O(\sqrt{n})$ y memoria $\sqrt{n} + \sqrt{n} \cdot \sqrt{n}$ words
- Segment Tree: query $O(\log n)$, update $O(\log n)$ y memoria $4n$ words (se puede hacer en $2n$)

Part II

Fenwick Tree

Fenwick Tree

Definición

Fenwick Tree (o BIT) es un *Binary Indexed Tree* descrito por Peter M. Fenwick en 1994.
Caracteristicas:

- Calcular una función en un rango l a r en $O(\log n)$
- Actualizar un valor en $O(\log n)$
- Require n words

Fenwick Tree

Definición

Fenwick Tree (o BIT) es un *Binary Indexed Tree* descrito por Peter M. Fenwick en 1994.
Caracteristicas:

- Calcular una función en un rango l a r en $O(\log n)$
- Actualizar un valor en $O(\log n)$
- Require n words

Implementación

- Almacenar un arreglo T de tamaño n

FENWICK TREE

Definición

Fenwick Tree (o BIT) es un *Binary Indexed Tree* descrito por Peter M. Fenwick en 1994.
Caracteristicas:

- Calcular una función en un rango l a r en $O(\log n)$
- Actualizar un valor en $O(\log n)$
- Require n words

Implementación

- Almacenar un arreglo T de tamaño n
- Definir una función $g(i)$, dónde $0 \leq g(i) \leq i$

Fenwick Tree

Definición

Fenwick Tree (o BIT) es un *Binary Indexed Tree* descrito por Peter M. Fenwick en 1994.
Caracteristicas:

- Calcular una función en un rango l a r en $O(\log n)$
- Actualizar un valor en $O(\log n)$
- Require n words

Implementación

- Almacenar un arreglo T de tamaño n
- Definir una función $g(i)$, dónde $0 \leq g(i) \leq i$
- En la posición i, se almacenará el resultado de f entre $g(i)$ e i

FUNCIÓN $g(i)$

Definición

$g(i)$ reemplaza todos los 1 finales a 0.

FUNCIÓN $g(i)$

Definición

$g(i)$ reemplaza todos los 1 finales a 0.

Ejemplos

- $g(11) = g(1011_2) = 1000_2 = 8$
- $g(13) = g(1101_2) = 1100_2 = 12$
- $g(21) = g(10101_2) = 10100_2 = 20$

FUNCIÓN $g(i)$

Definición

$g(i)$ reemplaza todos los 1 finales a 0.

Ejemplos

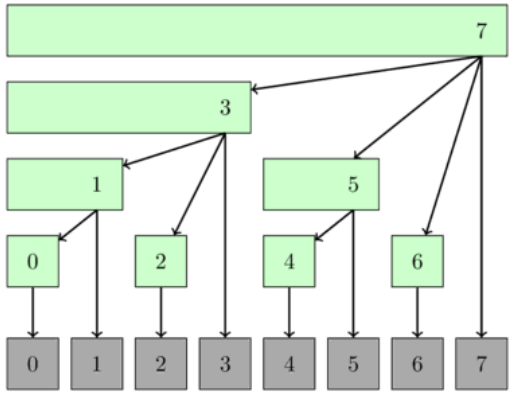
- $g(11) = g(1011_2) = 1000_2 = 8$
- $g(13) = g(1101_2) = 1100_2 = 12$
- $g(21) = g(10101_2) = 10100_2 = 20$

Fomula

$g(i) = i\&(i+1)$

EJEMPLO VISUAL

Si el arreglo fuera de tamaño 8, el árbol resultado sería:



RESOLVER OPERACIONES

Query en f

Para una query entre 0 e i:

1. Se añade la respuesta del rango $g(i)$ e i

RESOLVER OPERACIONES

Query en f

Para una query entre 0 e i:

1. Se añade la respuesta del rango $g(i)$ e i
2. Luego, salta al rango $g(g(i)-1)$ y $g(i)-1$

Resolver operaciones

Query en f

Para una query entre 0 e i:

1. Se añade la respuesta del rango $g(i)$ e i
2. Luego, salta al rango $g(g(i) - 1)$ y $g(i) - 1$
3. Añade el resultado en la respuesta

RESOLVER OPERACIONES

Query en f

Para una query entre 0 e i:

1. Se añade la respuesta del rango $g(i)$ e i
2. Luego, salta al rango $g(g(i) - 1)$ y $g(i) - 1$
3. Añade el resultado en la respuesta
4. Se repite el punto 2 hasta que $g(i) - 1 < 0$

Update en *i*

Actualizar en i, afecta a todos los rangos tales que $g(j) \leq i \leq j$

Resolver operaciones

Query en f

Para una query entre 0 e i:

1. Se añade la respuesta del rango $g(i)$ e i
2. Luego, salta al rango $g(g(i) - 1)$ y $g(i) - 1$
3. Añade el resultado en la respuesta
4. Se repite el punto 2 hasta que $g(i) - 1 < 0$

Update en *i*

Actualizar en i, afecta a todos los rangos tales que $g(j) \leq i \leq j$
¿Cómo hacemos esto?

RESOLVER OPERACIONES

Query en f

Para una query entre 0 e i:

1. Se añade la respuesta del rango $g(i)$ e i
2. Luego, salta al rango $g(g(i) - 1)$ y $g(i) - 1$
3. Añade el resultado en la respuesta
4. Se repite el punto 2 hasta que $g(i) - 1 < 0$

Update en i

Actualizar en i, afecta a todos los rangos tales que $g(j) \leq i \leq j$
¿Cómo hacemos esto?
Usando una función $h(i)$ tal que $h(i) = i|(i + 1)$

Código

```
struct fenwick_tree {
  vector <int> bit; int n;
  fenwick_tree(int n): n(n) { bit.assign(n, 0); }
  fenwick_tree(vector <int> &a): fenwick_tree(a.size()) {
    for (size_t i = 0; i < a.size(); i++)
      add(i, a[i]);
  }
  int sum(int r) {
    int ret = 0;
    for (; r >= 0; r = (r & (r + 1)) - 1)
      ret += bit[r];
    return ret;
  }
  int sum(int l, int r) {
    return sum(r) - sum(l - 1);
  }
  void add(int idx, int delta) {
    for (; idx < n; idx = idx | (idx + 1))
      bit[idx] += delta;
  }
};
```

CÓDIGO

```cpp
struct fenwick_tree {
  vector <int> bit; int n;
  fenwick_tree(int n): n(n) { bit.assign(n, 0); }
  fenwick_tree(vector <int> &a): fenwick_tree(a.size()) {
    for (size_t i = 0; i < a.size(); i++)
      add(i, a[i]);
  }
  int sum(int r) {
    int ret = 0;
    for (; r >= 0; r = (r & (r + 1)) - 1)
      ret += bit[r];
    return ret;
  }
  int sum(int l, int r) {
    return sum(r) - sum(l - 1);
  }
  void add(int idx, int delta) {
    for (; idx < n; idx = idx | (idx + 1))
      bit[idx] += delta;
  }
};
```

OJO: Fenwick Tree solo permite operaciones que tienen inverso.

Part III

Fenwick Tree 2D

Ahora en 2D

Extensión

Extender Fenwick Tree para permitir query y update en una grilla, es directo, dado que solo debemos añadirle la dimensión extra.

```
struct fenwick_tree_2d {
  int N, M;
  vector < vector < int >> bit;
  fenwick_tree_2d(int N, int M): N(N), M(M) {
    BIT.assign(N + 1, vector < int > (M + 1, 0));
  }
  void update(int x, int y, int v) {
    for (int i = x; i <= N; i += (i & -i))
      for (int j = y; j <= M; j += (j & -j))
        BIT[i][j] += v;
  }
  int sum(int x, int y) {
    int s = 0;
    for (int i = x; i > 0; i -= (i & -i))
      for (int j = y; j > 0; j -= (j & -j))
        s += BIT[i][j];
    return s;
  }
  int query(int x1, int y1, int x2, int y2) {
    return sum(x2, y2) - sum(x2, y1 - 1) - sum(x1 - 1, y2) + sum(x1 - 1, y1 - 1);
  }
};
```

References I