Consultas en rango

Problema

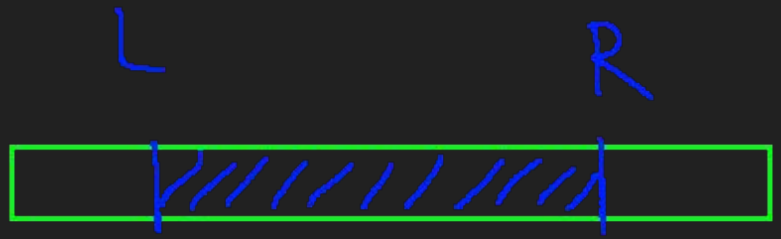
Queremos implementar una estructura de datos que, dado un array A de N elementos, soporte dos operaciones

- query(L, R): devuelve la suma A[L] + A[L+1] + … + A[R-1]
- update(P, X): modifica A tal que A[P] = X

Problema

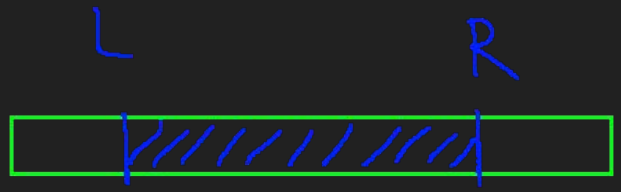
Queremos implementar una estructura de datos que, dado un array A de N elementos, soporte dos operaciones

- query(L, R): devuelve la suma A[L] + A[L+1] + … + A[R-1]
- update(P, X): modifica A tal que A[P] = X

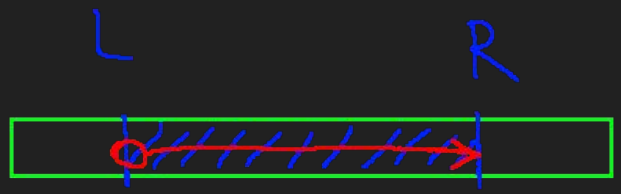


Implementación directa

Solución Naïve



Solución Naïve



Primera Solución

Tal como explica el problema, guardamos los datos en un array.

Para calcular una consulta, hacemos un for desde L a R.

Para actualizar una posición, la actualizamos en el array.

Primera Solución

Tal como explica el problema, guardamos los datos en un array.

Para calcular una consulta, hacemos un for desde L a R.

Para actualizar una posición, la actualizamos en el array.

```
int A[MAXN];

void init() {
}

ll query(int l, int r) {
    ll res = 0;
    for (int i = l; i < r; ++r)
        res += A[i];
    return res;
}

void update(int p, int x) {
    A[p] = x;
}
```

Primera Solución

Tal como explica el problema, guardamos los datos en un array.

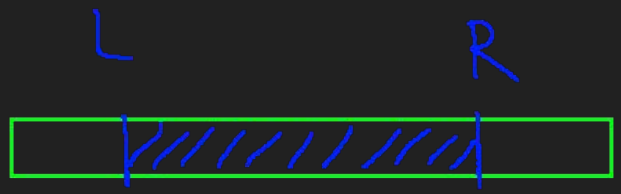
Para calcular una consulta, hacemos un for desde L a R. O(N)

Para actualizar una posición, la actualizamos en el array. O(1)

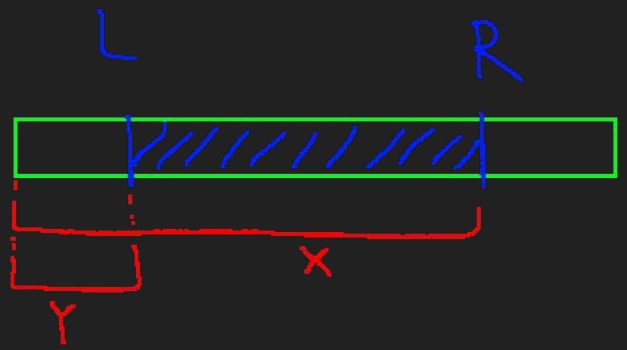
```
int A[MAXN];

void init() {
}

ll query(int l, int r) {
    ll res = 0;
    for (int i = l; i < r; ++r)
        res += A[i];
    return res;
}

void update(int p, int x) {
    A[p] = x;
}
```

Sumas Parciales

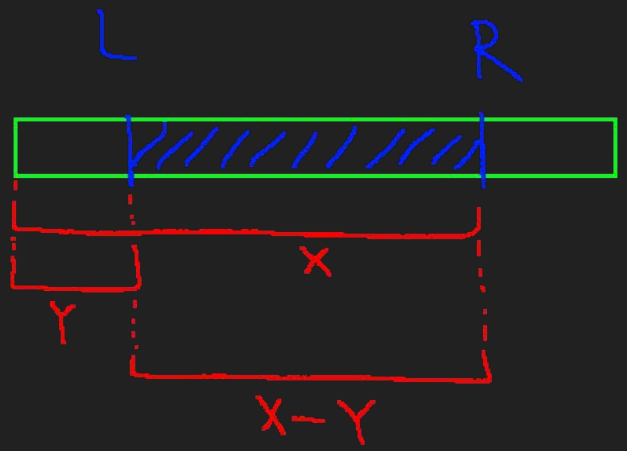
Sumas parciales



Sumas parciales



Sumas parciales



Segunda Solución

Damos vuelta un poco el problema. En vez de guardar los datos directamente, guardamos la suma desde el inicio del array hasta cada posición.

Para calcular una consulta, tomamos la diferencia de la suma hasta R con la suma hasta L.

Para actualizar una posición, re-calculamos todo lo que viene después en el array.

Segunda Solución

Damos vuelta un poco el problema. En vez de guardar los datos directamente, guardamos la suma desde el inicio del array hasta cada posición.

Para calcular una consulta, tomamos la diferencia de la suma hasta R con la suma hasta L.

Para actualizar una posición, re-calculamos todo lo que viene después en el array.

```
int A[MAXN];
ll  S[MAXN];

void init() {
    S[0] = 0;
    for (int i = 0; i < n; ++i)
        S[i+1] = A[i] + S[i];
}

ll query(int l, int r) {
    return S[r] - S[l];
}

void update(int p, int x) {
    A[p] = x;
    for (int i = p; i < n; ++i)
        S[i+1] = A[i] + S[i];
}
```

Segunda Solución

Damos vuelta un poco el problema. En vez de guardar los datos directamente, guardamos la suma desde el inicio del array hasta cada posición.

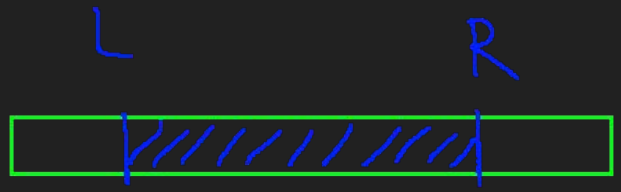
Para calcular una consulta, tomamos la diferencia de la suma hasta R con la suma hasta L. O(1)

Para actualizar una posición, re-calculamos todo lo que viene después en el array. O(N)

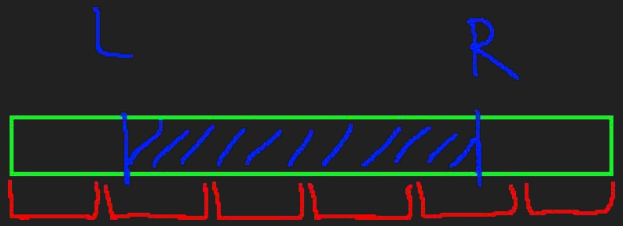
```
int A[MAXN];
ll  S[MAXN];

void init() {
    S[0] = 0;
    for (int i = 0; i < n; ++i)
        S[i+1] = A[i] + S[i];
}

ll query(int l, int r) {
    return S[r] - S[l];
}

void update(int p, int x) {
    A[p] = x;
    for (int i = p; i < n; ++i)
        S[i+1] = A[i] + S[i];
}
```

Bucketing

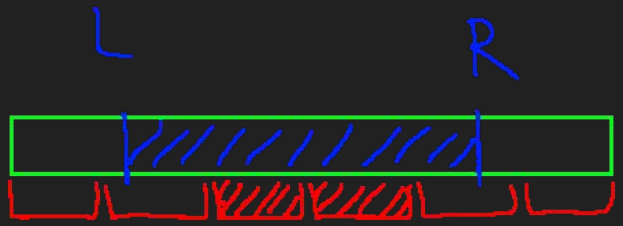
Bucketing



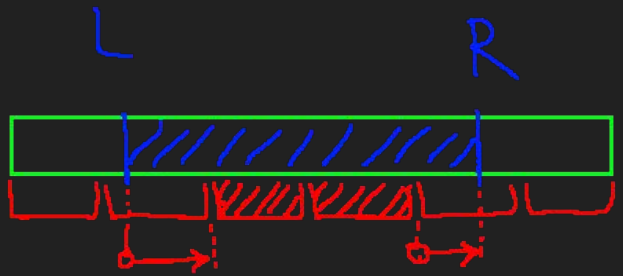
Bucketing



Bucketing



Bucketing



Bucketing

Cortamos el array en bloques de tamaño K.
Pre-calculamos la suma para cada bloque.

Suponemos que n es múltiplo de k.

Para cada índice `i`, su bloque es `i/k`;

Para cada bloque `b`, su primer índice es `b*k`.

Para cada query sumamos los bloques que abarca la query, más las puntas.

Para cada update actualizamos el valor y re-calculamos su bloque.

Bucketing

Cortamos el array en bloques de tamaño K.
Pre-calculamos la suma para cada bloque.

Suponemos que n es múltiplo de k.

Para cada índice `i`, su bloque es `i/k`;

Para cada bloque `b`, su primer índice es `b*k`.

Para cada query sumamos los bloques que abarca la query, más las puntas.

Para cada update actualizamos el valor y re-calculamos su bloque.

```
int A[MAXN];
ll  B[MAXK];

void init() {
    for (int i = 0; i < n; ++i) B[i/k] += A[i];
}

ll query(int l, int r) {
    ll res = 0;
    int lb = l/k, rb = r/k;
    // if (lb == rb) { for ... }
    for (int i = l; i < (lb+1)*k; ++i) res += A[i];
    for (int b = lb+1; b < rb; ++b)    res += B[b];
    for (int i = rb*k; i < r; ++i)     res += A[i];
    return res;
}

void update(int p, int x) {
    int b = p/k;
    A[p] = x;
    B[b] = 0;
    for (int i = 0; i < k; ++i) B[b] += A[b*k+i];
}
```

Bucketing

Cortamos el array en bloques de tamaño K.
Pre-calculamos la suma para cada bloque.

Suponemos que n es múltiplo de k.

Para cada índice `i`, su bloque es `i/k`;

Para cada bloque `b`, su primer índice es `b*k`.

Para cada query sumamos los bloques que abarca la query, más las puntas. O(N/K + K)

Para cada update actualizamos el valor y re-calculamos su bloque. O(K)

```
int A[MAXN];
ll  B[MAXK];

void init() {
    for (int i = 0; i < n; ++i) B[i/k] += A[i];
}

ll query(int l, int r) {
    ll res = 0;
    int lb = l/k, rb = r/k;
    // if (lb == rb) { for ... }
    for (int i = l; i < (lb+1)*k; ++i) res += A[i];
    for (int b = lb+1; b < rb; ++b)    res += B[b];
    for (int i = rb*k; i < r; ++i)     res += A[i];
    return res;
}

void update(int p, int x) {
    int b = p/k;
    A[p] = x;
    B[b] = 0;
    for (int i = 0; i < k; ++i) B[b] += A[b*k+i];
}
```

Bucketing

Tenemos queries en $O(N/K + K)$ y updates en $O(K)$. ¿Cómo elegimos K?

Bucketing

Tenemos queries en O(N/K + K) y updates en O(K). ¿Cómo elegimos K?

Depende de cada problema.

Pero notemos que si K es O(sqrt(N)) entonces ambas operaciones son O(sqrt(N)) asique suele ser una buena elección.

Muchas veces no hace falta que K sea justo sqrt(N), con que sea cercano a sqrt(MAXN) nos alcanza. (por ejemplo K=1024 cuando MAXN=10^6)

Resumen

Tenemos tres formas de resolver sumas en rango con distinta complejidad.

Estructura	init()	query(l,r)	update(p,x)
Array plano	O(1)	O(N)	O(1)
Tabla aditiva	O(N)	O(1)	O(N)
Bucketing	O(N)	O(sqrt(N))	O(sqrt(N))

El array plano es bueno con muchos updates y pocas queries, la tabla aditiva es buena en el caso contrario, y el bucketing es más balanceado.

Otro día veremos estructuras más complicadas con mejor complejidad.

Para pensar

- Vimos cómo resolver sumas en rango. ¿Podemos hacer productos en rango? ¿Si el array contiene un 0, cuáles técnicas andan y cuáles no?
- ¿Qué otras operaciones en rango podemos hacer aparte de la suma aplicando las ideas de hoy? ¿Se puede con todas las técnicas?
- ¿Qué pasa si en el bucketing también mantenemos una tabla aditiva en cada bucket? ¿Cómo queda la complejidad?
- ¿Qué pasa si en el bucketing mantenemos una tabla aditiva del array de buckets? ¿Cómo queda la complejidad?
- ¿Qué pasa si en el bucketing hacemos bucketing sobre el array de buckets? ¿Podemos obtener una cota mejor a sqrt(N) eligiendo bien los tamaños?

Problemas

- Static Range Sum Queries (CSES 1646)
- Range Xor Queries (CSES 1650)
- Static Range Minimum Queries (CSES 1647) testear TLE con sqrt
- Días Feriados (provincial OIA 2017)
- Forest Queries (CSES 1652) (*)
- Il Derby della Madonnina (Codeforces 1662L) (*) testear TLE con sqrt

(*) tiene una idea interesante aparte de lo visto en el taller