Estructuras de datos

Wilmer Emiro Castrillón Calderón

31 de julio de 2022

1. Tablas aditivas

Son estructuras de datos utilizadas para realizar operaciones acumuladas sobre un conjunto de datos estáticos en un rango específico, es decir, ejecutar una misma operación(como por ejemplo la suma) sobre un intervalo de datos, se asume que los datos en la estructura no van a cambiar. Estas estructuras también son conocidas como *Summed-area table* para el procesamiento de imágenes, a pesar de su nombre no necesariamente son exclusivas para operaciones de suma, pues la idea general es aplicable a otras operaciones.

Durante el cálculo de una misma operación sobre diferentes rangos se presenta superposición de problemas, las tablas aditivas son utilizadas para reducir la complejidad computacional aprovechando estas superposiciones utilizando programación dinámica.

Ejemplo inicial.

Dado un vector V = {5,2,8,2,4,3,1} encontrar para múltiples consultas la suma de todos los elementos en un rango [i,j], indexando desde 1, por ejemplo con el rango [1, 3] la suma es [5+2+8] = 15.

La solución trivial es hacer un ciclo recorriendo el vector entre el intervalo [i,j], en el peor de los casos se debe recorrer todo el vector, esto tiene una complejidad $O(n)$ puede que para una consulta sea aceptable, pero en casos grandes como por ejemplo un vector de tamaño 10^5 y una cantidad igualmente grande de consultas el tiempo de ejecución se hace muy alto, por lo tanto se hace necesario encontrar una mejor solución.

La operación suma tiene propiedades que nos pueden ayudar a resolver este problema de una forma mas eficiente:

1. La suma es asociativa es decir, se cumple: $a + (b + c) = (a + b) + c$, esto indica que sin importar la agrupación que se realice el resultado sera igual (no confundir con propiedad conmutativa).

2. La suma posee elemento neutro, es decir existe un β tal que $a + \beta = a$, en la suma $\beta = 0$.

3. La suma tiene operación inversa, es decir existe una operación que puede revertir la suma, la cual es la resta: si $a + b = c$ entonces $c - a = b$.

Considerando las anteriores propiedades el problema se puede trabajar desde otro enfoque, primero se puede definir $suma(x) = \sum_{k=1}^{x} V_k$, ahora tomando como ejemplo una consulta en el rango [3,5] del vector V, se puede utilizar $suma(2) = V_1 + V_2$ y $suma(5) = V_1+V_2+V_3+V_4+V_5$ para encontrar $V_3+V_4+V_5$, utilizando la propiedad asociativa se obtiene: $(V_1 + V_2) + (V_3 + V_4 + V_5) = (V_1 + V_2 + V_3 + V_4 + V_5)$ y usando la propiedad inversa se llega a: $(V_3+V_4+V_5) = (V_1+V_2+V_3+V_4+V_5)-(V_1+V_2)$ por lo tanto $(V_3+V_4+V_5) = suma(5)-suma(2)$. De esta manera el problema se puede generalizar como $\sum_{k=i}^{j} V_k = suma(j)-suma(i-1)$ cuando $i \neq 1$ y $suma(j)$ cuando $i = 1$.

Ahora pre-calculando $suma(x)$ se puede dar una solución inmediata a cada consulta, esto se puede resolver utilizando un enfoque básico de programación dinámica. Para encontrar $suma(x)$ se puede reescribir como: $suma(x) = V_x + suma(x-1)$ con caso base $suma(1) = V_1$ y por definición la operación acumulada sobre un conjunto vacío es igual al elemento neutro (esto significa que el indice cero tendrá el valor del elemento neutro), a partir de esto se puede obtener la siguiente solución en C++ con consultas indexando desde 1.

```
int V[] = {5,2,8,2,4,3,1}, memo[8];

void precalcular(){
    memo[0] = 0;
    for(int i = 0; i < 7; i++)
        memo[i+1] = V[i] + memo[i];
}

int consulta(int i, int j){ return memo[j] - memo[i-1]; }
```

De manera general las tablas aditivas son aplicables a cualquier operación que posea las tres propiedades descritas anteriormente: ser asociativa, tener elemento neutro y operación inversa, por ejemplo la suma, multiplicación o el operador de bits XOR.

Tablas aditivas en 2D

Las operaciones acumuladas no solo se pueden usar sobre una dimension, sino también sobre n-dimensiones, en estos casos se debe trabajar con el principio de inclusión-exclusión pues se debe considerar mejor las operaciones entre intervalos, si no tiene en cuenta este principio las soluciones contendrían elementos duplicados o faltantes, lo que produciría soluciones incorrectas.

Ejemplo: dada la matriz M encontrar para múltiples consultas la suma de todos los elemento en una submatriz $Q_{(i1,j1),(i2,j2)}$:

$$M = \begin{array}{|c|c|c|c|c|}\hline 1 & 9 & 6 & 3 & 7 \\ \hline 7 & 5 & 3 & 0 & 5 \\ \hline 0 & 7 & 6 & 5 & 3 \\ \hline 7 & 8 & 9 & 5 & 0 \\ \hline 9 & 5 & 3 & 7 & 8 \\ \hline \end{array}$$

En el intervalo $Q_{(2,2),(3,4)}$ el resultado es $5+3+0+7+6+5 = 26$.

En el caso de 1D se definió $suma(x) = \sum_{k=1}^{x} V_k$, ahora esta debe tener dos dimensiones, es decir, $suma(i,j)$ debe contener la suma de los elementos en la submatriz $Q_{(1,1),(i,j)}$, entonces ahora se definirá: $suma(i,j) = \sum_{k=1}^{i}\sum_{w=1}^{j} M_{k,w}$, mas sin embargo pre-calcular $suma(i,j)$ de

forma eficiente requiere de usar el principio de inclusión-exclusión, de manera trivial se puede calcular la primera fila como $suma(1,j) = \sum_{w=1}^{j} M_{1,w}$ y la primera columna como $suma(i,1) = \sum_{k=1}^{i} M_{k,1}$, en base a esto se puede calcular el resto de la matriz pero se debe tener algo de cuidado, por ejemplo si se toma $suma(2,2) = suma(1,2)+suma(2,1)+M_{2,2}$ se obtendría un resultado incorrecto pues se estaría realizando la siguiente operación: $(M_{1,1}+M_{1,2})+(M_{1,1},M_{2,1})+M_{2,2}$, se puede observar que el elemento $M_{1,1}$ se esta sumando dos veces, acá se aplica el principio de inclusión-exclusión: $|A| \cup |B| = |A| + |B| - |A \cap B|$ lo que significa que hace falta quitar la intersección, esta es $suma(1,1)$ entonce se puede generalizar como: $suma(i,j) = M_{i,j} + suma(i-1,j) + suma(i,j-1) - suma(i-1,j-1)$ cuando $i,j \neq 1$.

Una vez construido el pre-calculo es necesario realizar las consultas, se utilizara como ejemplo la consulta en el rango $Q_{(2,2),(3,4)}$, de igual manera se debe tener cuidado de no sumar un mismo intervalo mas de una vez, entonces para encontrar la suma en este intervalo se tomaría $suma(i2,j2)$ esta contendría $\sum_{k=1}^{i2}\sum_{w=1}^{j2} M_{k,w}$, esta tiene elemento adicionales como lo muestra la figura 1, al restarle $suma(i1-1,j2)$ se quitan algunos elementos (figura 2), al restarle $suma(i2,j1-1)$ pasaríamos a restar dos veces el intervalo $M_{(1,1),(i1-1,j1-1)}$ (figura 3), por lo tanto es necesario reponer lo faltante agregando $suma(i1-1,j1-1)$ (figura 4), de esta manera se puede generalizar la formula: $\sum_{k=i1}^{i2}\sum_{w=j1}^{j2} M_{k,w} = suma(i2,j2) - suma(i1-1,j2) - suma(i2,j1-1) + suma(i1-1,j1-1)$.

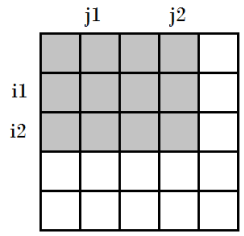


Figura 1

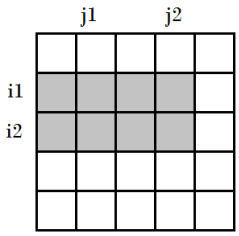


Figura 2

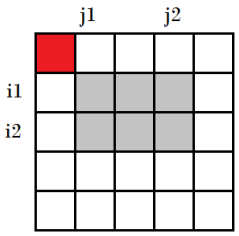


Figura 3

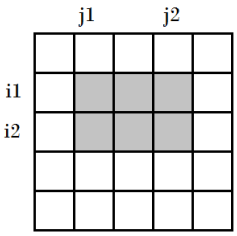


Figura 4

Para la solución en C++ la fila y columna cero de la tabla donde se guardara el pre-calculo se deberá llenar con el elemento neutro, el cero, luego se debe llenar el resto de la tabla siguiendo las formulas antes establecidas, este ejemplo es para consultas indexando desde 1.

```
1  void precalcular(){
2      memset(memo, 0, sizeof(memo));
3      for (int i = 1; i <= fila; i++)
4      for (int j = 1; j <= col; j++)
5          memo[i][j] = memo[i][j - 1] + memo[i - 1][j] +
6                          M[i - 1][j - 1] - memo[i - 1][j - 1];
7  }
8
9  int consulta(int i1, int j1, int i2, int j2){
10     return memo[i2][j2] - memo[i1-1][j2] -  memo[i2][j1-1] + memo[i1-1][j1-1];
11 }
```

Tablas aditivas en 3D

Las tablas aditivas se pueden generalizar para trabajar en n-dimensiones, mas sin embargo la dificultad de hacer los pre-cálculos y las consultas aumenta bastante, pues crece considerablemente la cantidad de operaciones a realizar. En el caso 3D igualmente se debe tener

cuidado con el principio de inclusión-exclusión, el pre-calculo se realizaría de la siguiente manera: sea $suma(i,j,k) = \sum_{x=1}^{i}\sum_{y=1}^{j}\sum_{z=1}^{k} V_{x,y,z}$, entonces $suma(i,j,k) = V_{i,j,k} + suma(i,j,k-1) + suma(i-1,j,k) + suma(i,j-1,k) - suma(i-1,j-1,k) - suma(i-1,j,k-1) - suma(i,j-1,k-1) + suma(i-1,j-1,k-1)$, y para las consultas: $\sum_{x=i1}^{i2}\sum_{y=j1}^{j2}\sum_{z=k1}^{k2} V_{x,y,z} = suma(i2,j2,k2) - suma(i2,j2,k1-1) - suma(i1-1,j2,k2) + suma(i1-1,j2,k1-1) - suma(i2,j1-1,k2) + suma(i2,j1-1,k1-1) + suma(i1-1,j1-1,k2) - suma(i1-1,j1-1,k1-1)$.

Conclusiones

Esta estructura de datos facilita encontrar el valor acumulado de una operación sobre un rango de valores estáticos, puede ser aplicada para operaciones que cumplan con las tres propiedades descritas anteriormente, la complejidad computacional de hacer el pre-calculo es lineal a la cantidad de elementos en la estructura, y las consultas se realizan en complejidad constante al realizar solamente operaciones directas sobre elementos en la tabla del pre-calculo. Si se requiere actualizar los valores de la tabla esto haría necesario actualizar el pre-calculo, esto representa que la operación de actualizar es lineal a la cantidad de elementos, pero esa complejidad no suele ser favorable, por lo tanto se recomienda utilizar solo en arreglos estáticos, y recurrir a otras estructuras como las Sparse table en el caso de arreglos dinámicos.

2. Disjoint set

Es una estructura de datos que permite agrupar elementos en conjuntos y consultar si dos elementos pertenecen a un mismo conjunto, contiene una estructura en forma de árbol pero con una implementacion poco compleja. La estructura consta de dos operaciones principales, $union(u,v)$ para unir los conjuntos que incluyen a los nodos u y v, y la operación $find(v)$ para indicar a cual conjunto pertenece un nodo v.

Cada nodo tendrá la información de su nodo padre, y cada conjunto tiene un nodo raíz. Entonces se tiene un arreglo lineal p donde $p[v]$ tiene el padre del nodo v, para los nodos raíz se tendrá que $p[v] = v$, como se muestra en la imagen siguiente, inicialmente cada nodo es un conjunto diferente entonces todo el arreglo debe iniciar con $p[v] = v$.

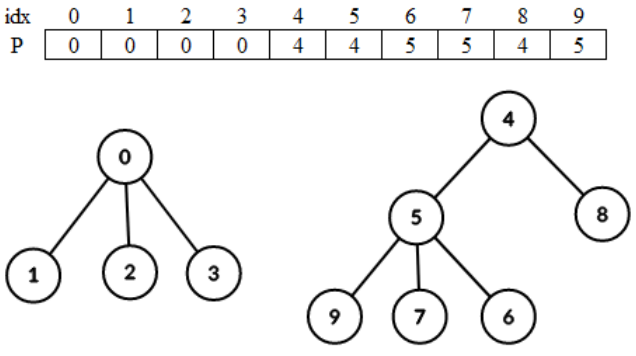


Figura 5: Disjoint set

Operación *Find*

La operación consiste en devolver el nodo raíz del conjunto al cual pertenece un nodo v, como cada nodo guarda el indice de su nodo padre, entonces solo se debe subir sobre el árbol hasta llegar al nodo raíz el cual se puede detectar con la condición $p[x] = x$.

Operación *Union*

Consiste en unir dos elementos u y v en un mismo conjunto, básicamente consiste en hacer que el nodo raíz del conjunto de u sea padre del nodo raíz del conjunto de v, de esta forma ambos nodos quedan dentro de un mismo árbol, si u y v ya se encuentran dentro en un mismo conjunto entonces no se hace nada.

```
struct union_find{
    int padre[100];

    void build(int n){
        for(int i = 0; i < n; i++) padre[i] = i;
    }

    int buscar(int v){//find
        if(v == padre[v]) return v;
        else return buscar(padre[v]);
    }

    void unir(int u, int v){//union
        u = buscar(u);  v = buscar(v);
        if(u != v) padre[u] = v;
    }

    bool MismoGrupo(int u, int v){
        return buscar(u) == buscar(v);
    }
};
```

Unión por rangos

Los disjoint set son estructuras flexibles que nos permiten realizar diferentes variantes para distintas tareas, una de ellas es la unión por rangos, la cual consiste en tener guardado en un arreglo la máxima profundidad que contiene cada nodo y nos permite durante la operación *Union* unir el árbol de menor rango a otro de mayor de rango. En la siguiente figura se observa un ejemplo de unir dos conjuntos con distinto rango, los conjuntos del nodo 9 y 8, sus raíces son 5 y 4 respectivamente, entonces como el nodo 4 tiene un rango mayor, se volverá el padre del nodo 5.

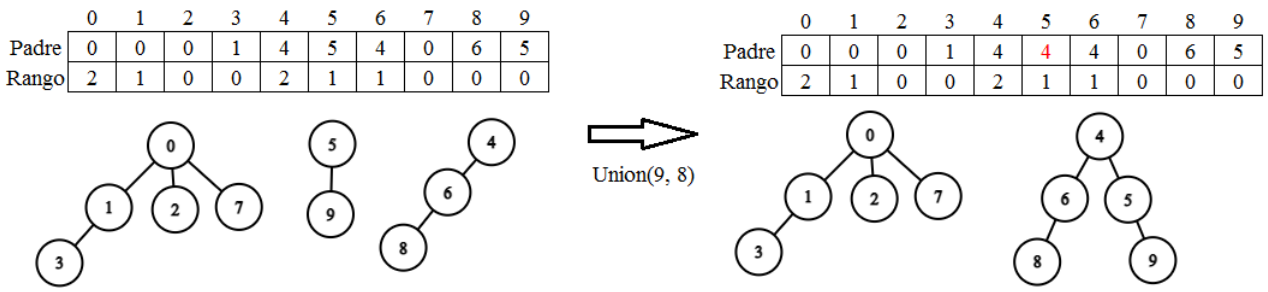


Figura 6: Unión por rangos

```cpp
int padre[100], rango[100];

void build(int n){
    for (int i = 0; i < n; i++){
        padre[i] = i;  rango[i] = 0;
    }
}

void unir(int u, int v){
    u = buscar(u);  v = buscar(v);
    if(u == v) return;
    if(rango[u] > rango[v]){
        padre[v] = u;
        return;
    }
    padre[u] = v;
    if(rango[v] == rango[u]) rango[v]++;
}
```

Compresión de caminos

Cuando se utiliza gran cantidad de nodos el disjoint set tradicional es ineficiente, debido a la posibilidad de formar arboles muy profundos, llevando a la operación $find(v)$ a tener complejidad $O(n)$ para cada búsqueda, existe una optimización la cual permite reducir la profundidad de los arboles, de tal forma que la máxima longitud de rangos sera 1, de esta forma se evita tener arboles profundos. La implementacion es sencilla y consiste que en cada búsqueda actualizar los nodos visitados como hijos directos del nodo raíz. La siguiente figura muestra como se vería el anterior ejemplo con compresión de caminos.

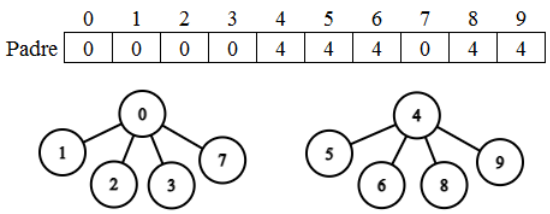


Figura 7

```cpp
int buscar(int v){
    if(v == padre[v]) return v;
    else return padre[v] = buscar(
        padre[v]);
}

void unir(int u, int v){ padre[buscar(
    u)] = buscar(v); }
```

Conclusiones

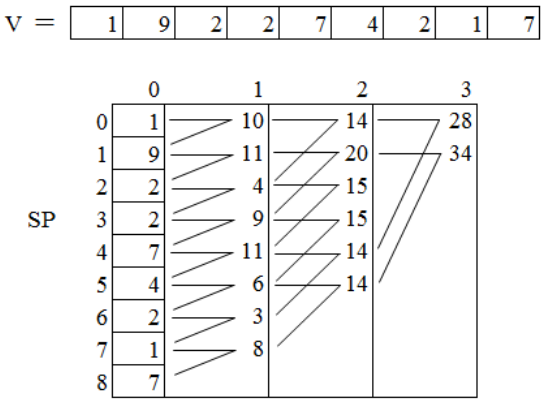
Esta estructura permite asociar elementos en conjuntos de forma eficiente, tiene múltiples aplicaciones como por ejemplo almacenar componentes conexas de grafos, es utilizada por otros algoritmos como el algoritmo de kruskal para la búsqueda del árbol de expansión mínima en grafos, también se puede almacenar información adicional sobre los conjuntos como su cantidad de nodos o almacenar para cada nodo la lista de sus descendientes, entre otras aplicaciones.

3. Sparse table

Es una estructura de datos para realizar operaciones sobre rangos en un conjunto datos, esta estructura utiliza un precálculo sobre los valores iniciales para poder realizar consultas en complejidad logarítmica, permite realizar operaciones que cumplen con la propiedad asociativa como la suma o la función mínimo(el valor mínimo entre dos valores), una de sus ventajas es su implementación corta, sin embargo esta estructura no permite actualizaciones, por lo que debe ser usada con datos estáticos.

Esta estructura aprovecha la propiedad de que todo número entero se puede representar de forma única como sumas de potencias de dos, consiste en operar rangos del arreglo que tengan longitudes de potencia de dos y guardarlos en una tabla de precálculo, se utiliza una matriz de n filas y $log_2(n)$ columnas, la primera columna contendrá el arreglo inicial, las demás columnas tendrán la operación acumulada en el rango $[i, i + 2^j)$ del arreglo inicial, para cada consulta se operan los valores de sólo algunos rangos, evitando que se operen todos los valores en el intervalo. Para la explicación de construcción y consulta se usará un vector V para los valores iniciales, una matriz SP para guardar el sparse table y la operación suma.

La construcción es corta y consiste de inicialmente llenar la primera columna con los valores iniciales $SP_{i,0} = V_i$, y para las demás columnas se aplica la siguiente fórmula $SP_{i,j} = SP_{i,j-1} + SP_{i+2^{j-1},j-1}$, con $1 \leq j \leq log_2(n)$ y $1 \leq i+2^j-1 < n$, al final cada posición de la matriz tendrá los siguientes valores $SP_{i,j} = \sum_{x=i}^{i+2^j-1} V_x$ con $0 \leq i < n$ y $0 \leq i + 2^j - 1 < n$. La construcción tiene una complejidad $O(n * log_2(n))$.




```
1 void construir(){
2     for(int i = 0; i < n; ++i)
3         SP[i][0] = V[i];
4
5     int x = log2(n);
6     for(int j = 1; j <= x; ++j)
7         for(int i = 0; i+(1<<j)-1 < n;
              ++i)
8             SP[i][j] = min(SP[i][j-1],
                  SP[i+(1<<(j-1))][j
                 -1]);
9 }
```

Figura 8

Para las consultas se debe tener en cuenta que $SP_{i,j}$ guarda la operación acumulada en el intervalo $[i, i + 2^j - 1]$, Entonces para obtener la operación acumulada en un rango $[L, R]$ se debe tomar la fila L y buscar el máximo j tal que $L + 2^j - 1 \leq R$ es decir, tomar el intervalo mas grande que no se salga del rango $[L, R]$, con esto ya se tiene el acumulado de una parte del intervalo, de tal forma que se puede actualizar el rango a $[L + 2^j, R]$ y repetir el proceso hasta encontrar el valor de todo el intervalo $[L, R]$; la complejidad es $O(log_2(n))$. Por ejemplo para calcular suma en el rango $[1, 6]$ se puede obtener con solo sumar dos valores de la tabla, el primero $SP_{1,2}$ el cual contiene la suma en el rango [1,4] y el segundo $SP_{5,1}$ el cual contiene la suma en el rango $[5, 6]$, entonces resultado es 26, como se muestra en la figura 9.

Optimización para operaciones idempotentes

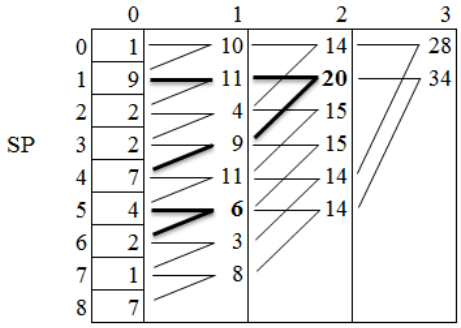


Figura 9

```
int consulta(int L, int R){
    int res = 0;
    while(L <= R){
        int j = log2(R-L+1);
        res += SP[L][j];
        L += 1<<j;
    }
    return res;
}
```

Las operaciones idempotentes son aquellas que se pueden aplicar múltiples veces y el resultado será el mismo que al aplicarse la primera vez, es decir $f(f(x)) = f(x)$, por ejemplo las funciones mínimo y máximo, para este tipo de casos se puede trabajar con intervalos superpuestos y la respuesta seguirá siendo igual, por ejemplo para encontrar el RMQ en un intervalo $[L, R]$ con $L < R$ se cumple que $RMQ(L, R) = min(RMQ(L, R-1), RMQ(L+1, R))$, entonces la consulta sobre el sparse table se puede simplificar para este tipo de funciones, de tal forma que solo dos intervalos son suficientes para encontrar la respuesta acumulada en cualquier rango, estos intervalos serán $[L, L + 2^j]$ y $[R - 2^j + 1, R]$ con $j = floor(log_2(R - L + 1))$ de esta forma la complejidad es $O(1)$.

```
int consulta_idempotentes(int L, int R){
    int j = log2(R-L+1);
    return min(SP[L][j], SP[R-(1<<j)+1][j]);
}
```

4. Fewnwick tree

También conocido como árbol binario indexado, es una estructura de datos no lineal la cual permite calcular operaciones acumuladas sobre rangos permitiendo actualizar el conjunto de datos, se basa en dos operaciones básicas, calcular la operación acumulada sobre los primeros i elementos, y actualizar un elemento del conjunto de datos, ambas se realizan en $O(log_2(n))$, el árbol requiere $O(n)$ de espacio, según la implementación utilizada se puede empezar a indexar desde 0 o 1, sin embargo ambas son equivalentes en cuanto a complejidad algorítmica.

La estructura consiste en almacenar en un arreglo lineal ft la operación acumulada sobre un intervalo del arreglo de datos A, de tal forma que al operar uno o múltiples elementos en ft se pueda obtener la operación acumulada del intervalo $[1, i]$, indexando desde 1, los intervalos del precalculo se toman basados en la representación binaria del indice, en la figura 10 se muestra un ejemplo utilizando la operación suma, para entender mejor se recomienda escribir los indices i en binario y comparar con los rangos sumados en ft_i.

La construcción del árbol consiste en empezar con ft vacío, es decir $\forall i \in [0, n] ft_i = 0$, posteriormente realizar n actualizaciones, una por cada elemento en A, entonces la construcción inicial tiene complejidad $O(n * log_2(n))$, la actualización consiste en agregar la diferencia entre el valor nuevo y el anterior, cada actualización consiste en tomar el indice i (indexando desde cero) e ir recorriendo sus bits comenzando desde el lsb (lowest significat bit), si el k-ésimo bit es cero significa que ft_{i+2^k} también debe ser actualizado (puede ser algo confuso al comienzo, se recomienda hacer el proceso a mano para que se pueda entender mejor), una forma mas

	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
A =	6	10	3	5	7	9	9	10	7	1	9	3	4	5	6	1
ft =	6	16	3	24	7	16	9	59	7	8	9	20	4	9	6	95

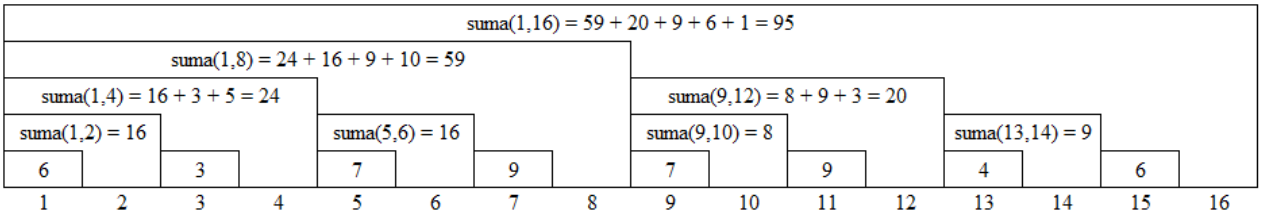


Figura 10: Fenwick tree

rápida de recorrer los bits que necesitamos es utilizando operaciones de bits, si en lugar de usar i utilizamos a $\sim i$, los bits que requerimos serán los que están en 1, comenzando desde el lsb, entonces se puede tomar el ultimo dígito en 1 usando la operación $i\& - i$ y posteriormente apagarlo para tomar el siguiente encendido, de esta forma se puede ir directamente a los bits que se necesitan, y por ultimo en lugar de actualizar ft_i se puede actualizar ft_{i+1} para que los indices empiecen en 1.

La operación de consulta, consiste en obtener la sumatoria hasta un indice i, esta operación tiene una idea similar a la de actualización, se empieza con el valor ft_i, posteriormente se actualiza i apagando su ultimo bit encendido y al resultado se le suma el nuevo ft_i, este proceso se repite mientras i sea mayor a 0.

```
int ft[MAX], n = 16;

struct fenwick_tree {//indexando desde 1

  fenwick_tree() {
    for(int i = 0; i < n; ++i) ft[i] = 0;
  }

  void update(int idx, int valor) {
    int x = ~idx, lsb = 0;

    while(lsb < n) {
      ft[idx + lsb + 1] += valor;
      lsb ^= x & -x;
      x = x ^ (x & -x);
    }
  }

  void construir(vector<int> &v) {
    for(int i = 0; i < v.size(); ++i) {
      update(i, v[i]);
    }
  }

  int query(int i) {
    int res = 0, lsb = 0;
```

```
27
28      while(i > 0) {
29        res += ft[i];
30        lsb = i & -i;
31        i = i ^ lsb;
32      }
33      return res;
34    }
35
36    int query(int i, int j) {
37      return query(j) - query(i - 1);
38    }
39  };
```