Preparazione alle Olimpiadi di Informatica

UNIVERSITÀ DELLA CALABRIA

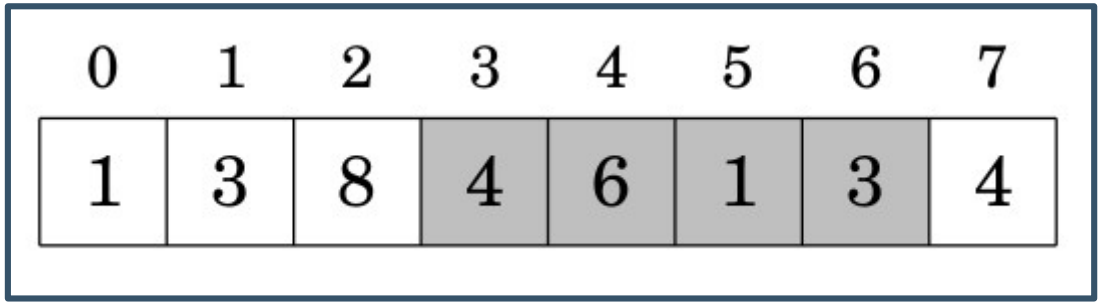
il Campus per eccellenza

Range queries

Mario Alviano

Range queries (1)

- Interrogazioni che coinvolgono intervalli di un array
  - $sum_q(a,b)$:= array[a] + … + array[b]
  - $min_q(a,b)$:= min(array[a], …, array[b])
  - $max_q(a,b)$:= max(array[a], …, array[b])



$sum_q(3,6) = 14$, $min_q(3,6) = 1$, $max_q(3,6) = 6$

Range queries (2)

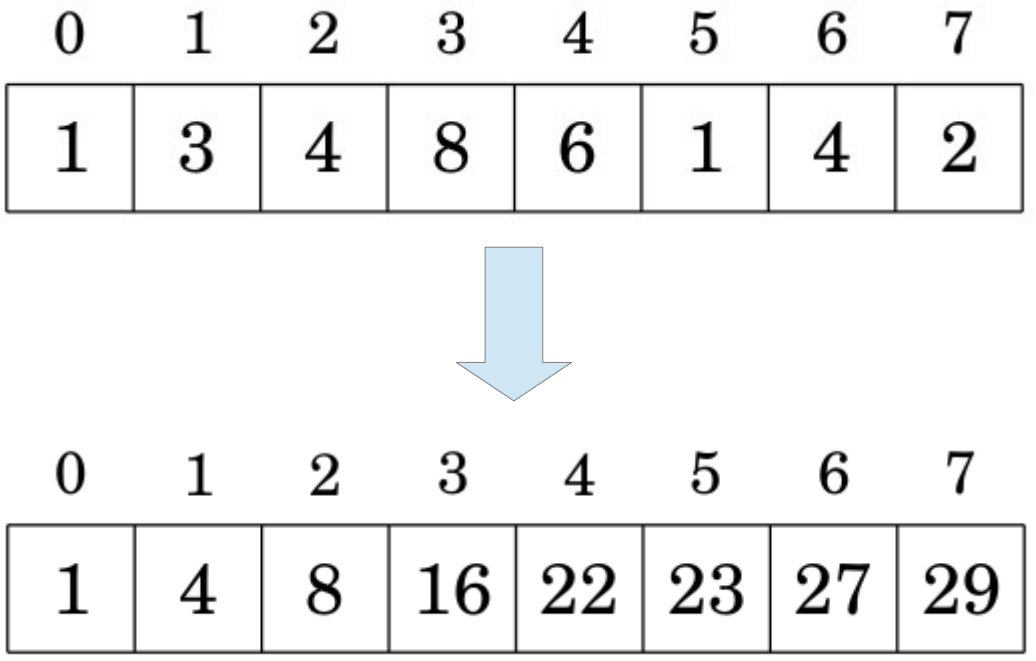
- Se dobbiamo rispondere a molte query, vogliamo evitare di iterare sull'intero range
- Query statiche
  - L'array non cambia
- Query dinamiche
  - L'array cambia

```
int sum(int a, int b) {
    int s = 0;
    for (int i = a; i <= b; i++) {
        s += array[i];
    }
    return s;
}
```

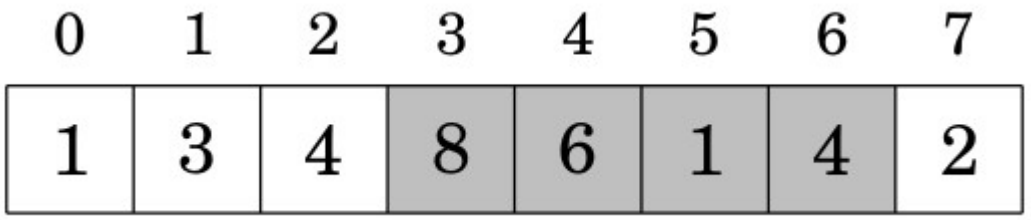
Vogliamo fare meglio di così!

Prefix Sum Array (1)

- Utile per rispondere a static sum queries
- Memorizza le somme dei prefissi dell'array
- Si costruisce in O(n)
- Le query sono processate in O(1)

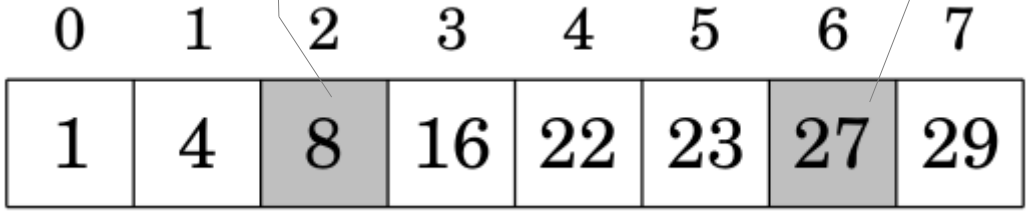


Prefix Sum Array (2)



$sum_q(3,6) = 8 + 6 + 1 + 4 = 19$

$sum_q(0,2)$ $sum_q(0,6)$



$sum_q(3,6) = sum_q(0,6) - sum_q(0,2)$
$= 27 - 8 = 19$

$$sum_q(a,b) = sum_q(0,b) - sum_q(0,a-1)$$

Prefix Sum Array (3)

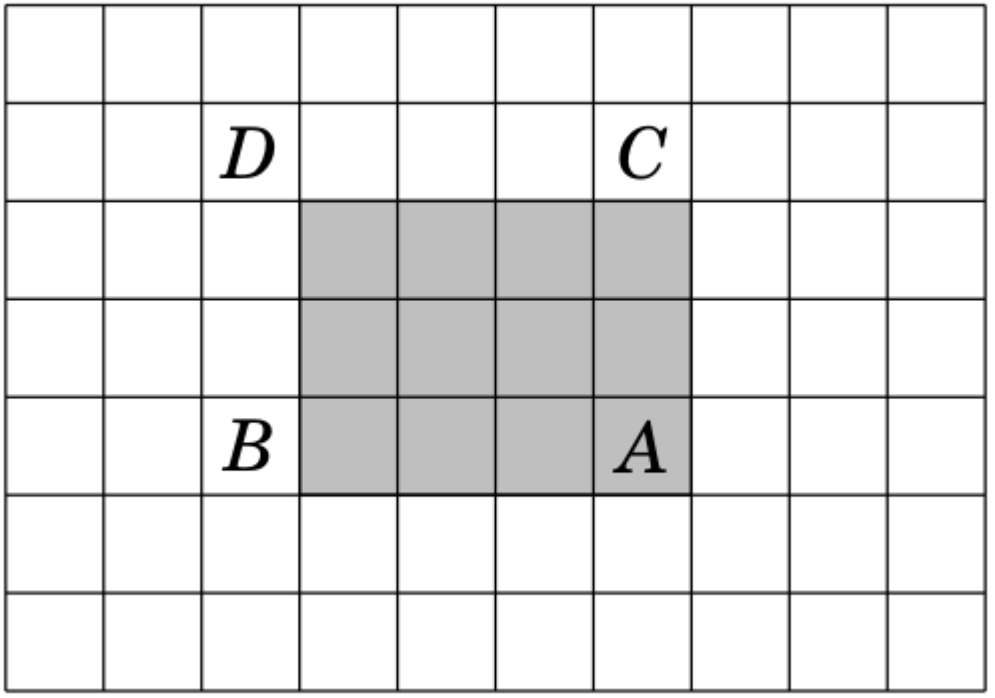
```
void prefix_sum_array(const vector<int>& array, vector<int>& prefix) {
    if(array.empty()) return;
    prefix.push_back(array[0]);
    for(int i = 1; i < array.size(); i++) prefix.push_back(prefix.back() + array[i]);
}
```

```
int sumq(const vector<int>& prefix, int a, int b) {
    return prefix[b] - (a > 0 ? prefix[a-1] : 0);
}
```

```
vector<int> array = {1, 3, 4, 8, 6, 1, 4, 2};

vector<int> prefix;
prefix_sum_array(array, prefix);

cout << sumq(prefix, 3, 6) << endl;
```

Prefix Sum Multidimensional Array (1)

- L’idea generalizza ad array multidimensionali
- Per ogni cella calcoliamo la somma degli elementi nel rettangolo fino all’angolo in alto a sinistra
- Per avere la somma del rettangolo ACDB sono sufficienti 3 somme



$$S(A) - S(B) - S(C) + S(D)$$

Prefix Sum Multidimensional Array (2)

```
typedef vector<vector<int>> grid;
typedef pair<int, int> cell;
```

```
void prefix_sum_array(const grid& array, grid& prefix) {
    for(int i = 0; i < array.size(); i++) {
        prefix.push_back(vector<int>());
        for(int j = 0; j < array[i].size(); j++) {
            prefix[i].push_back(array[i][j] +
                (i > 0 ? prefix[i-1][j] : 0) +
                (j > 0 ? prefix[i][j-1] : 0) +
                (i > 0 && j > 0 ? -prefix[i-1][j-1] : 0));
        }
    }
}
```

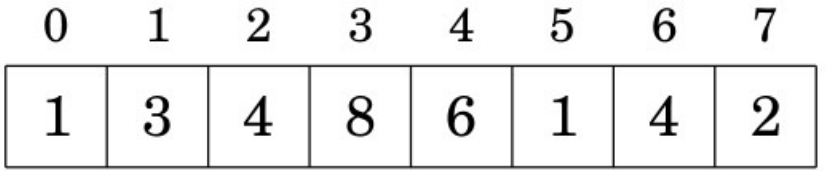
```
grid array = {
    {3, 7, 9, 2, 7},
    {9, 8, 3, 5, 5},
    {1, 7, 9, 8, 5},
    {3, 8, 6, 4, 10},
    {6, 3, 9, 7, 8}
};

grid prefix;
prefix_sum_array(array, prefix);

cout << sumq(prefix, {1,2}, {3,3}) << endl;
```

```
int sumq(const grid& prefix, cell a, cell b) {
    return prefix[b.first][b.second] - (a.second > 0 ? prefix[b.first][a.second-1] : 0)
        - (a.first > 0 ? prefix[a.first-1][b.second] : 0)
        + (a.first > 0 && a.second > 0 ? prefix[a.first-1][a.second-1] : 0);
}
```

Sparse Table (1)

- Utile per rispondere a static min/max queries
- Memorizza i valori di $\min_q(a,b)$ se $b - a + 1 = 2^k$ (per qualche k)
- Si costruisce in O(n log n)
- Le query sono processate in O(1)

Sparse Table (2)



a	b	$\min_q(a,b)$
0	0	1
1	1	3
2	2	4
3	3	8
4	4	6
5	5	1
6	6	4
7	7	2

a	b	$\min_q(a,b)$
0	1	1
1	2	3
2	3	4
3	4	6
4	5	1
5	6	1
6	7	2

a	b	$\min_q(a,b)$
0	3	1
1	4	3
2	5	1
3	6	1
4	7	1
0	7	1

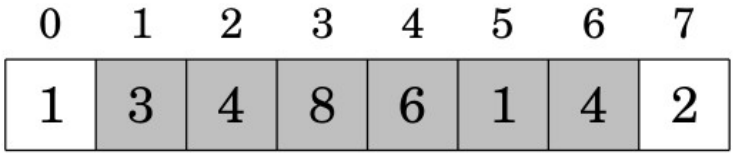
Ricorsione dal range più grande

Caso base: range di 1 elemento

$$\min_q(a,b) = \min(\min_q(a, a+w-1), \min_q(a+w, b))$$

$$w = (b-a+1)/2$$

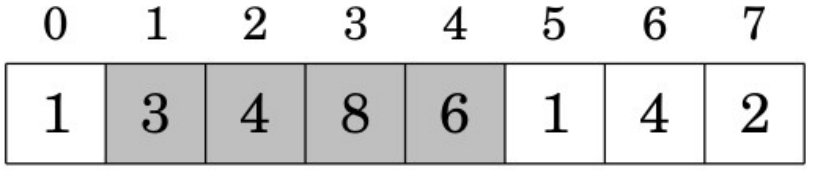
Sparse Table (3)



$$\min_q(a,b) = \min(\min_q(a,a+k-1), \min_q(b-k+1,b)).$$

b – a + 1 = 6
k = 4

k è la più grande potenza di 2
che non eccede b – a + 1



Ogni range è ottenuto
come unione di due range

Sparse Table (4)

```
typedef unordered_map<int, unordered_map<int, int>> sparse_table;
```

```
void min_table(const vector<int>& array, sparse_table& table) {
    for(int a = 0; a < array.size(); a++) table[a][a] = array[a];

    for(int w = 1; 2*w <= array.size(); w *= 2) {
        for(int a = 0; a + 2*w <= array.size(); a++) {
            int b = a + 2*w - 1;
            table[a][b] = min(table[a][a+w-1], table[a+w][b]);
        }
    }

}
```

```
int minq(sparse_table& table, int a, int b) {
    int k = log2(b - a + 1);
    return min(table[a][a+k-1], table[b-k+1][b]);
```

```
vector<int> array = {1, 3, 4, 8, 6, 1, 4, 2};

sparse_table table;
min_table(array, table);

cout << minq(table, 1, 6) << endl;
```

Binary Indexed Tree (1)

- Variante dinamica di prefix sum array
- Anche noto come Fenwick tree
- Sum queries in O(log n)
- Modifica di valori in O(log n)
- Solitamente implementato usando un array
- Indicizziamo da 1 per semplicità

Binary Indexed Tree (2)

p(k) := più grande potenza di 2 che divide k

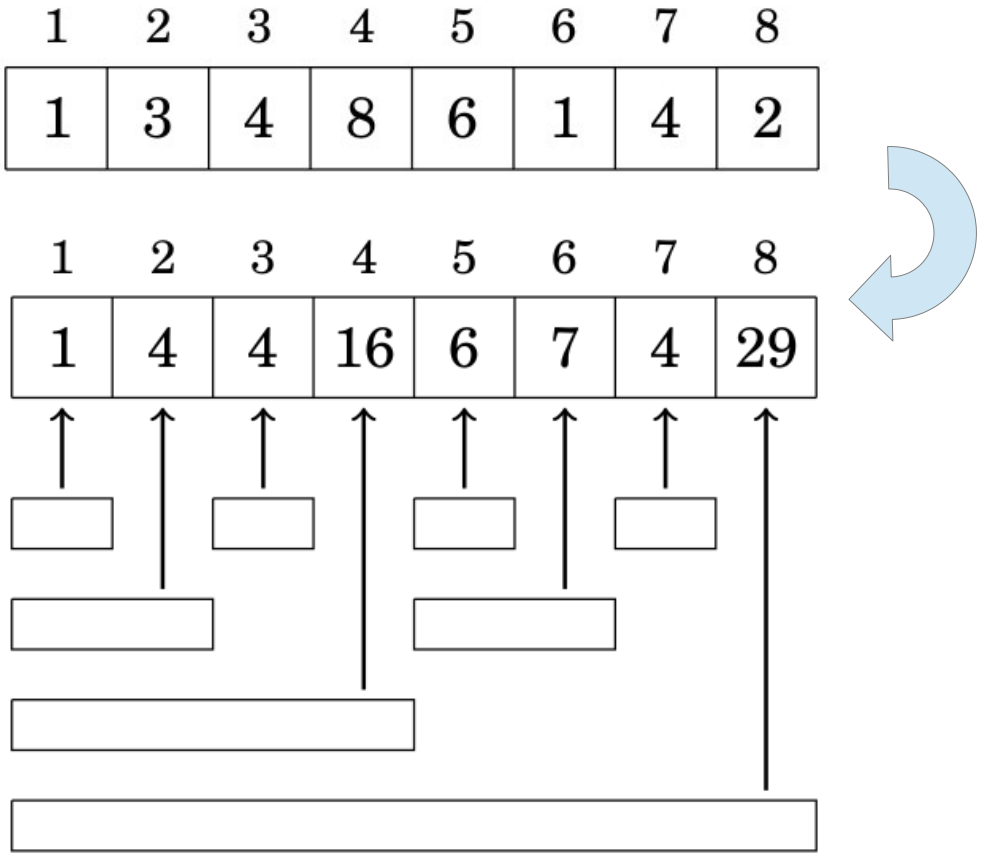
$$\texttt{tree}[k] = \text{sum}_q(k - p(k) + 1, k)$$

tree[k] := somma del range di dimensione p(k) che termina in posizione k

Esempio

p(6) = 2
tree[6] = $\text{sum}_q(5,6)$

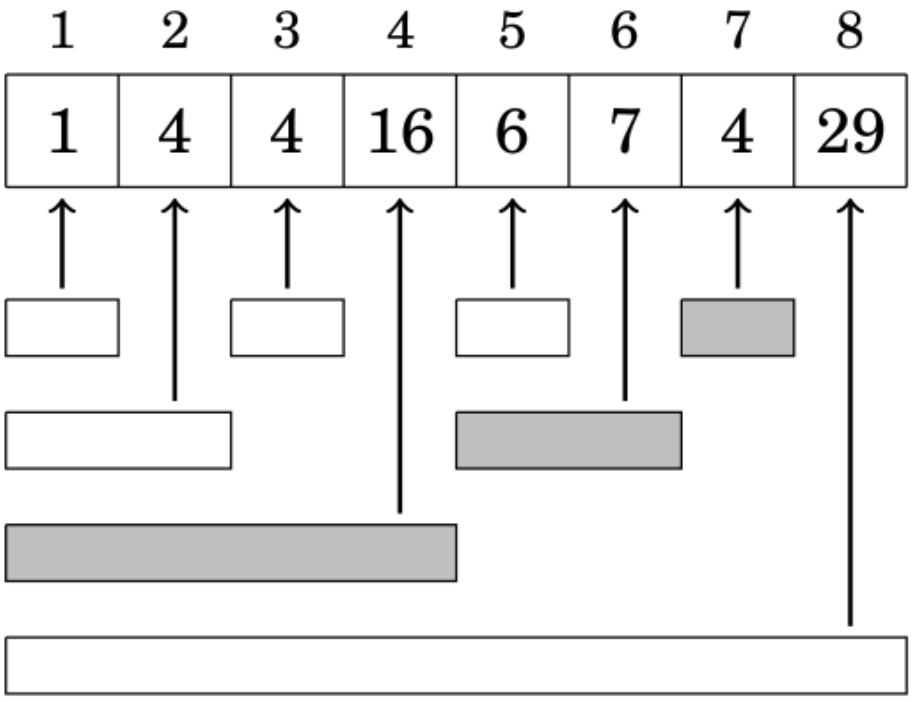
Binary Indexed Tree (3)



Ogni range [1,k] si ottiene come somma di O(log n) range memorizzati nel binary indexed tree

Binary Indexed Tree (4)

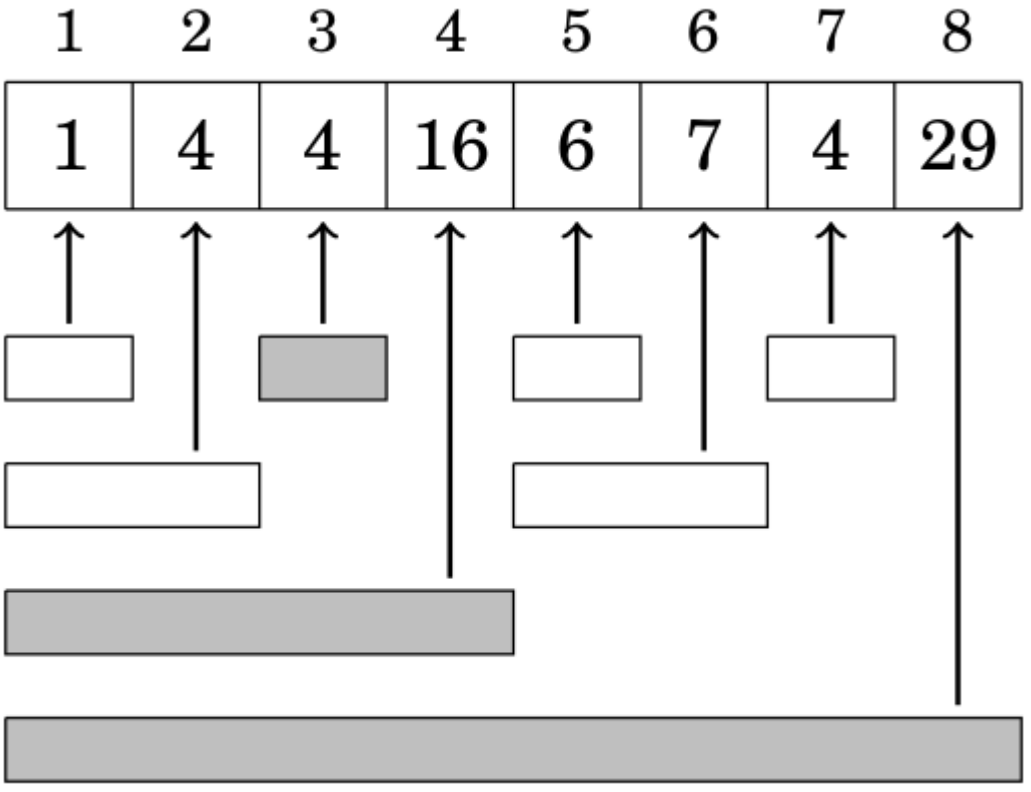
$$\text{sum}_q(1,7) = \text{sum}_q(1,4) + \text{sum}_q(5,6) + \text{sum}_q(7,7) = 16 + 7 + 4 = 27$$



Rispondiamo alle query come facevamo con il prefix sum array

$$\text{sum}_q(a,b) = \text{sum}_q(1,b) - \text{sum}_q(1,a-1)$$

Binary Indexed Tree (5)



Per cambiare un valore dell'array dobbiamo aggiornare O(log n) range

per cambiare array[3]

Binary Indexed Tree (6)

$$p(k) = k \& -k$$

```
void add(vector<int>& tree, int k, int val) {
    k++;
    while(k <= tree.size()) {
        tree[k-1] += val;
        k += k & -k;
    }
}
```

```
int sumq(const vector<int>& tree, int k) {
    k++;
    int res = 0;
    while(k >= 1) {
        res += tree[k-1];
        k -= k & -k;
    }
    return res;
}
```

```
int sumq(const vector<int>& tree, int a, int b) {
    return sumq(tree, b) - sumq(tree, a - 1);
}
```

```
void binary_indexed_tree(const vector<int>& array, vector<int>& tree) {
    tree.resize(array.size());
    for(int i = 0; i < array.size(); i++) add(tree, i, array[i]);
}
```

Segment Tree (1)

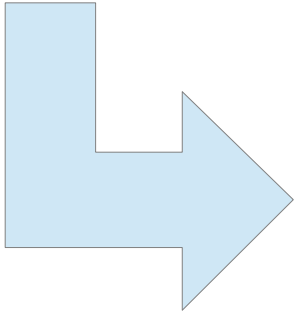
- Supporta range queries e aggiornamenti di elementi
- Lavora in O(log n)
- Richiede più memoria di un binary indexed tree
- L’implementazione è più articolata
- È un albero binario
  - Le foglie sono gli elementi dell’array
  - I nodi interni rappresentano range
- Per semplicità, assumiamo che il numero di elementi sia una potenza di 2

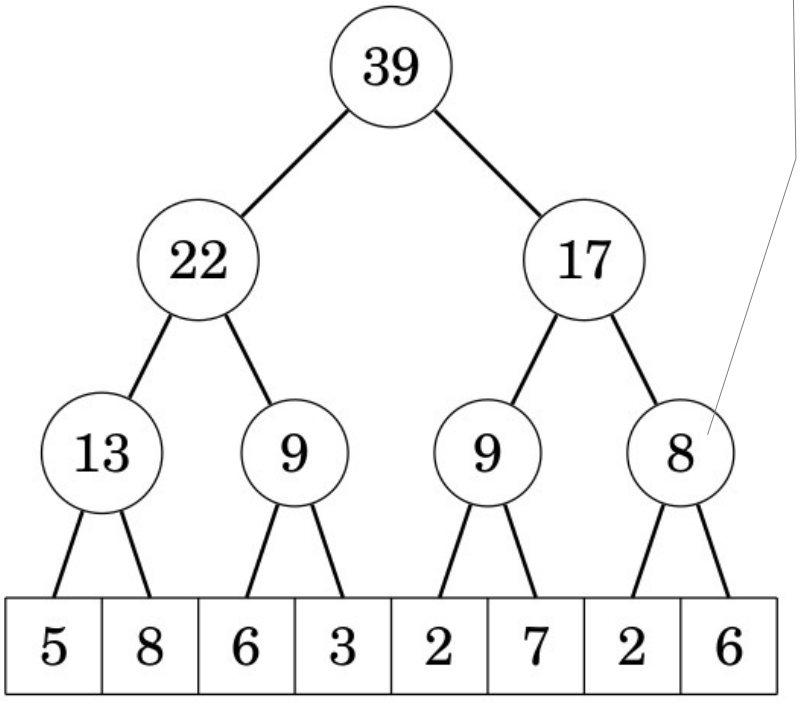
Segment Tree (2)

Somma dei figli

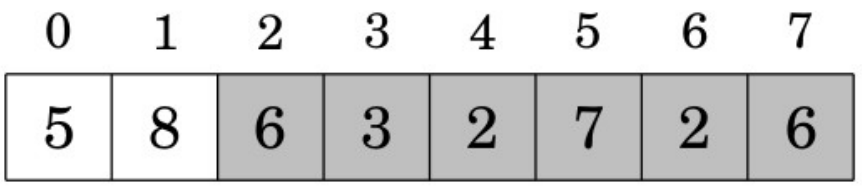
2 + 6 = 8

0	1	2	3	4	5	6	7
5	8	6	3	2	7	2	6



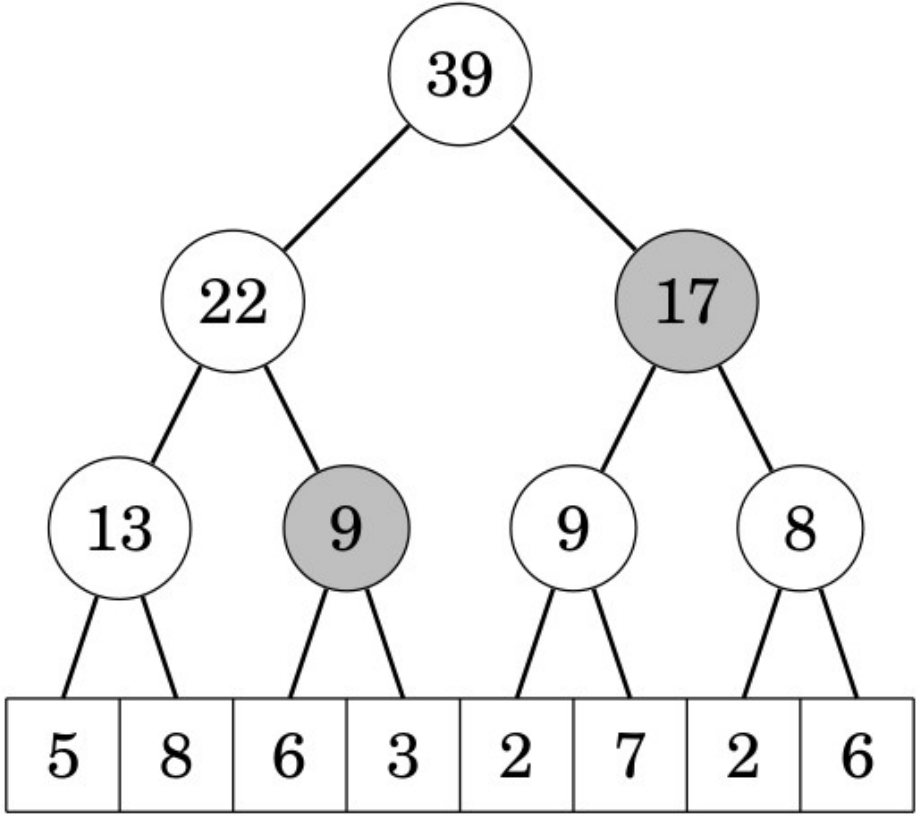


Segment Tree (3)

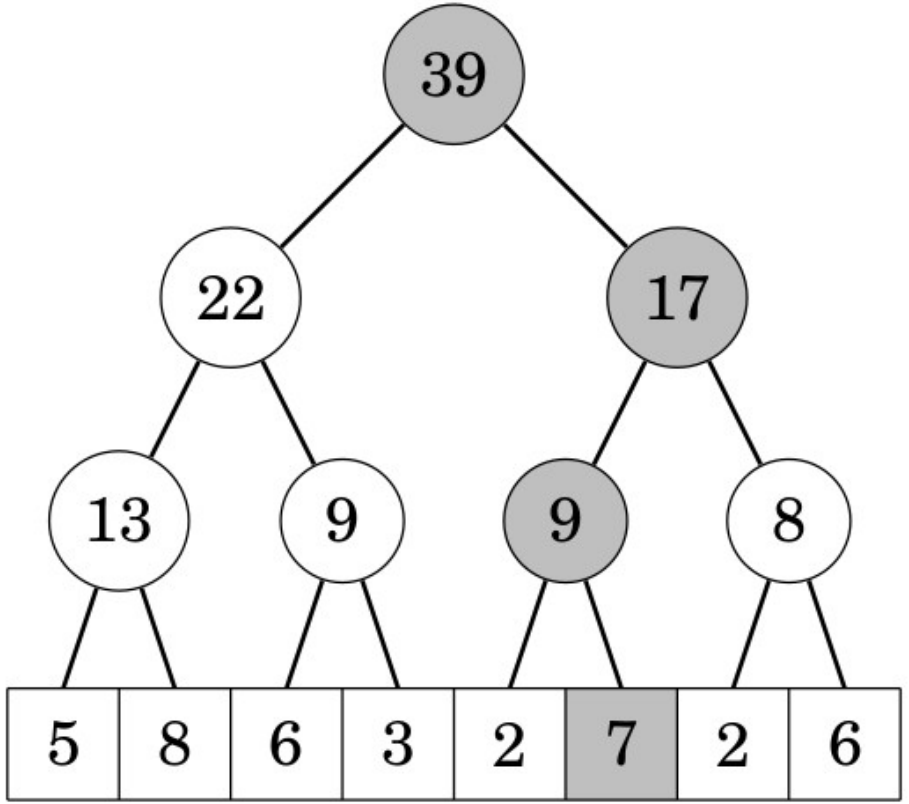


Ogni range è la somma di O(log n) nodi

$sum_q(2,7) = sum_q(2,3) + sum_q(4,7)$
$= 9 + 17 = 26$



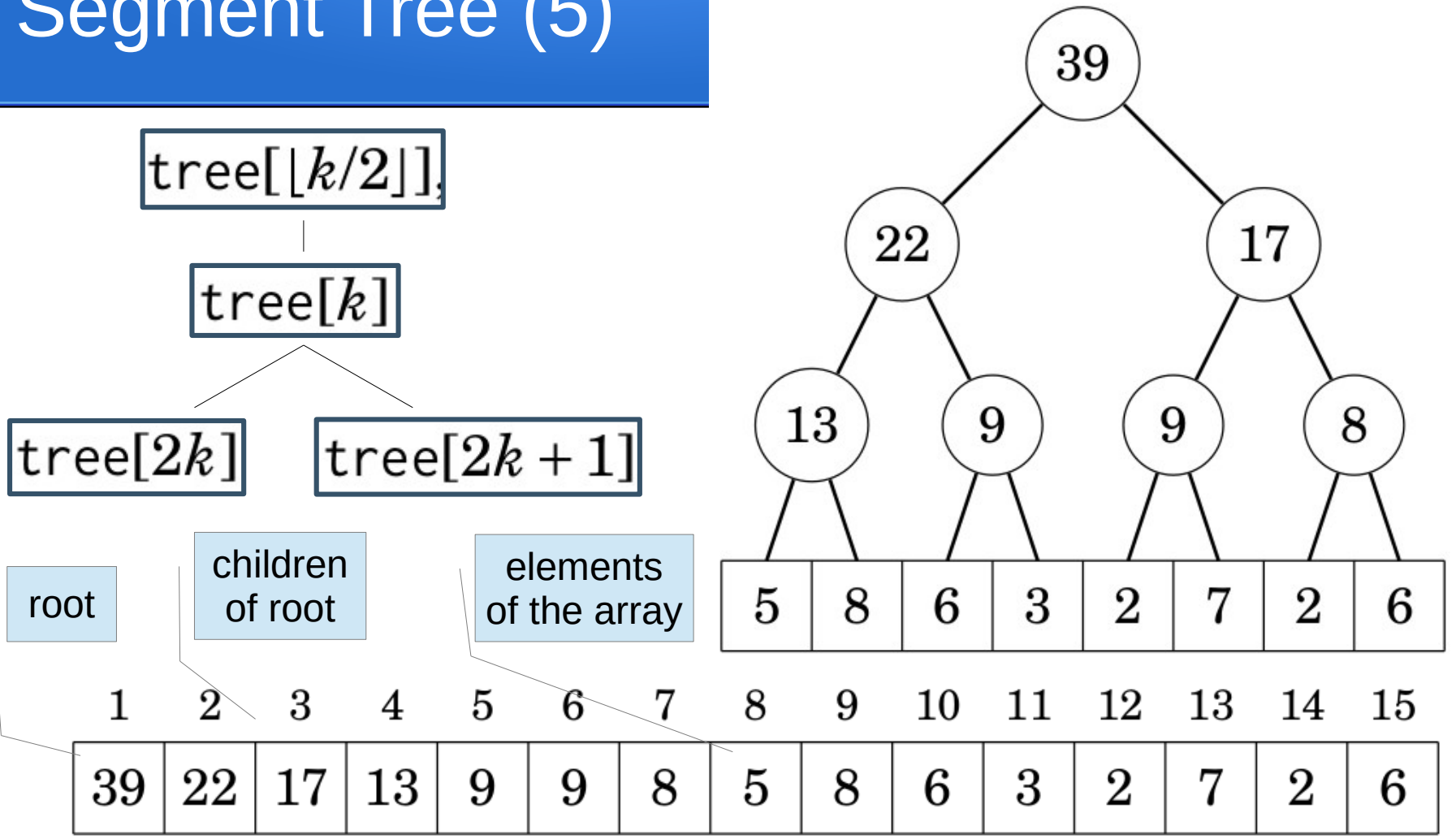
Segment Tree (4)



Per cambiare un elemento dell'array bisogna cambiare i nodi nel cammino dalla foglia associata fino alla radice

O(log n) nodi vengono modificati

Segment Tree (5)



Segment Tree (6)

```
void add(vector<int>& tree, int k, int val) {
    k += tree.size() / 2;
    tree[k] += val;
    for(k /= 2; k >= 1; k /= 2) tree[k] = tree[2*k] + tree[2*k+1];
}
```

```
void segment_tree(const vector<int>& array, vector<int>& tree) {
    tree.resize(array.size() * 2);
    for(int i = 0; i < array.size(); i++) add(tree, i, array[i]);
}
```

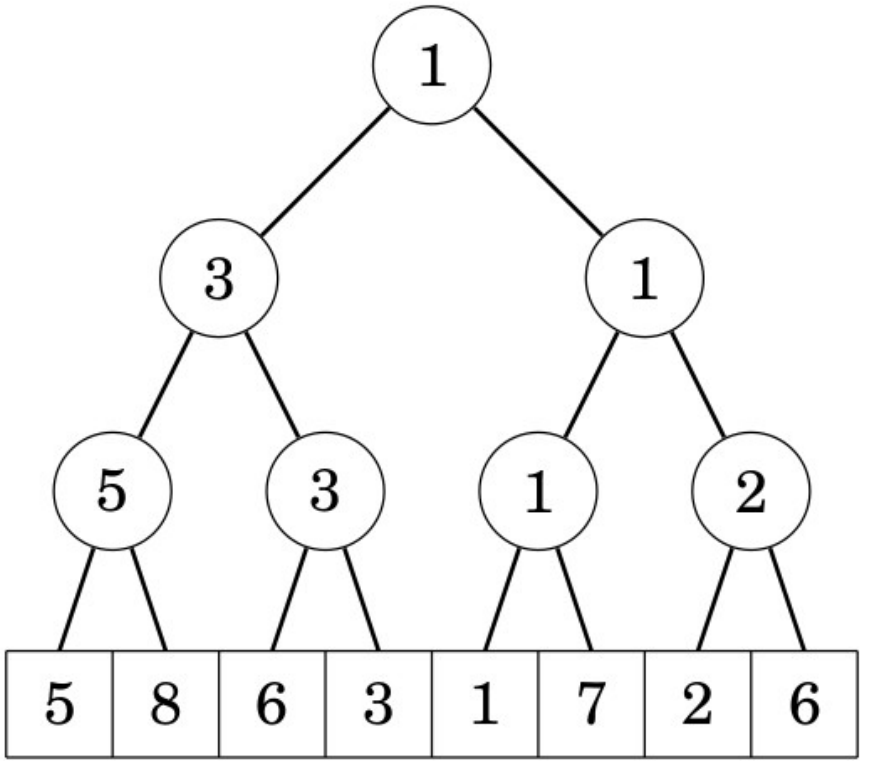
```
int sumq(const vector<int>& tree, int a, int b) {
    a += tree.size() / 2;
    b += tree.size() / 2;
    int res = 0;
    for(; a <= b; a /= 2, b /= 2) {
        if(a % 2 == 1) res += tree[a++];
        if(b % 2 == 0) res += tree[b--];
    }
    return res;
}
```

```
vector<int> array = {5, 8, 6, 3, 2, 7, 2, 6};

vector<int> tree;
segment_tree(array, tree);

cout << sumq(tree, 2, 7) << endl;
```

Segment Tree (7)

- In modo simile si ottengono altre range queries
  - minimo
  - massimo
  - massimo comune divisore
  - and, or, xor

segment tree per minimo



Segment Tree (8)

```
void set_value(vector<int>& tree, int k, int val) {
    k += tree.size() / 2;
    tree[k] = val;
    for(k /= 2; k >= 1; k /= 2) tree[k] = min(tree[2*k], tree[2*k+1]);
}
```

```
void segment_tree(const vector<int>& array, vector<int>& tree) {
    tree.resize(array.size() * 2);
    for(int i = 0; i < array.size(); i++) set_value(tree, i, array[i]);
}
```

segment tree
per minimo

```
int minq(const vector<int>& tree, int a, int b) {
    a += tree.size() / 2;
    b += tree.size() / 2;
    int res = INT_MAX;
    for(; a <= b; a /= 2, b /= 2) {
        if(a % 2 == 1) res = min(res, tree[a++]);
        if(b % 2 == 0) res = min(res, tree[b--]);
    }
    return res;
}
```

Segment Tree (9)

```
void set_value(vector<int>& tree, int k, int val) {
    k += tree.size() / 2;
    tree[k] = val;
    for(k /= 2; k >= 1; k /= 2) tree[k] = __gcd(tree[2*k], tree[2*k+1]);
}
```

```
void segment_tree(const vector<int>& array, vector<int>& tree) {
    tree.resize(array.size() * 2);
    for(int i = 0; i < array.size(); i++) set_value(tree, i, array[i]);
}
```

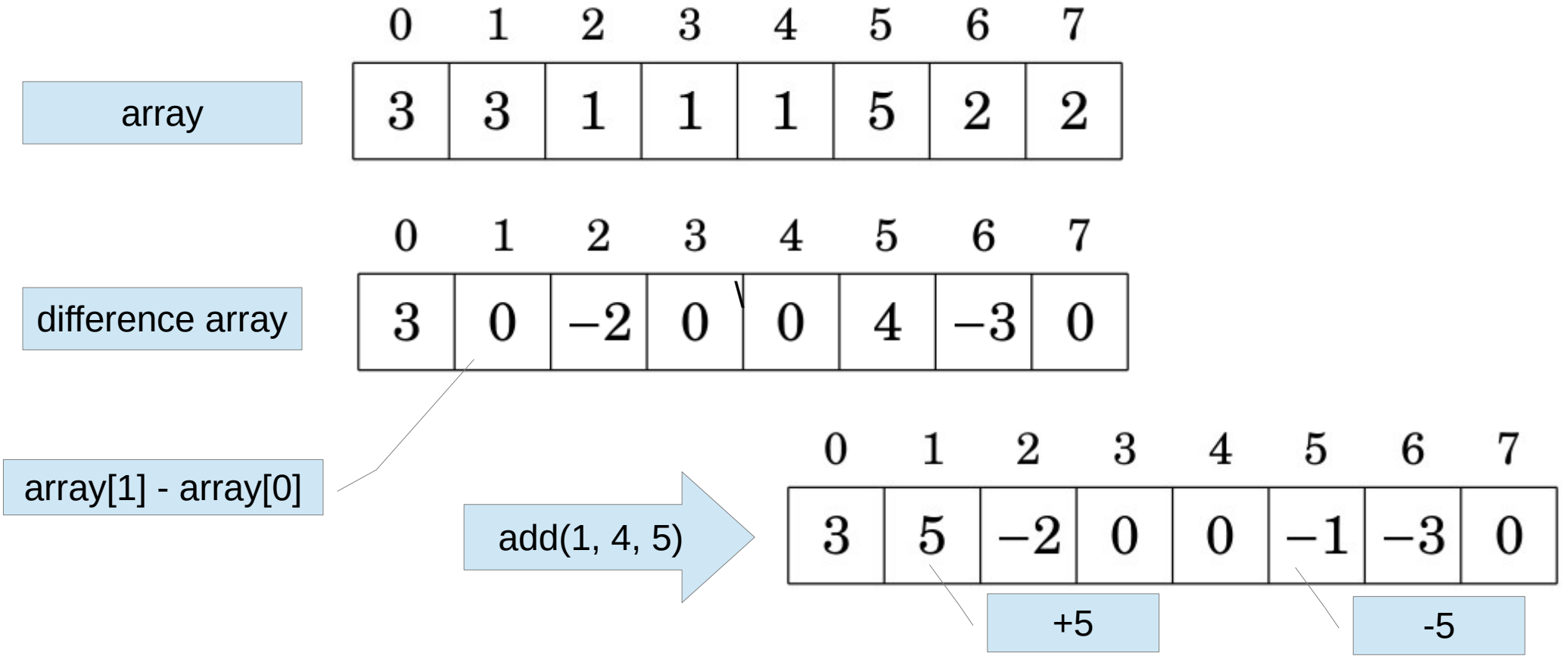
```
int gcdq(const vector<int>& tree, int a, int b) {
    a += tree.size() / 2;
    b += tree.size() / 2;
    int res = tree[a];
    for(; a <= b; a /= 2, b /= 2) {
        if(a % 2 == 1) res = __gcd(res, tree[a++]);
        if(b % 2 == 0) res = __gcd(res, tree[b--]);
    }
    return res;
}
```

segment tree
per massimo
comune divisore

Difference Array (1)

- Utile per range updates
- Memorizza le differenze fra elementi adiacenti
- Aggiungere una quantità a un range di elementi costa O(1)
- Determinare il valore di un elemento costa O(n)

Difference Array (2)



Difference Array (3)

```
void difference_array(const vector<int>& array, vector<int>& diff) {
    diff.resize(array.size());
    diff[0] = array[0];
    for(int i = 1; i < array.size(); i++) diff[i] = array[i] - array[i-1];
}
```

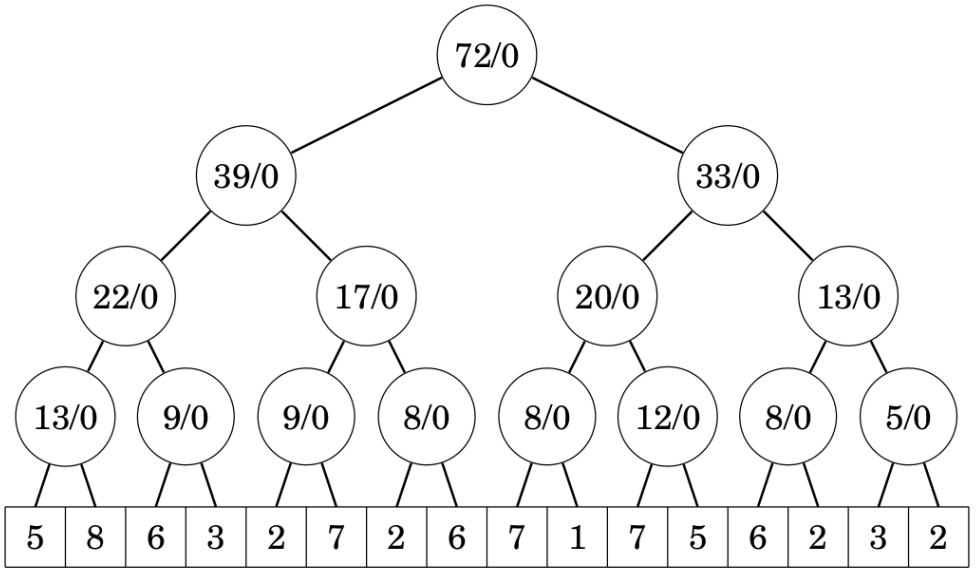
```
void add(vector<int>& diff, int a, int b, int val) {
    diff[a] += val;
    if(b+1 < diff.size()) diff[b+1] -= val;
}
```

```
int get_value(vector<int>& diff, int idx) {
    int res = 0;
    for(int i = 0; i <= idx; i++) res += diff[i];
    return res;
}
```

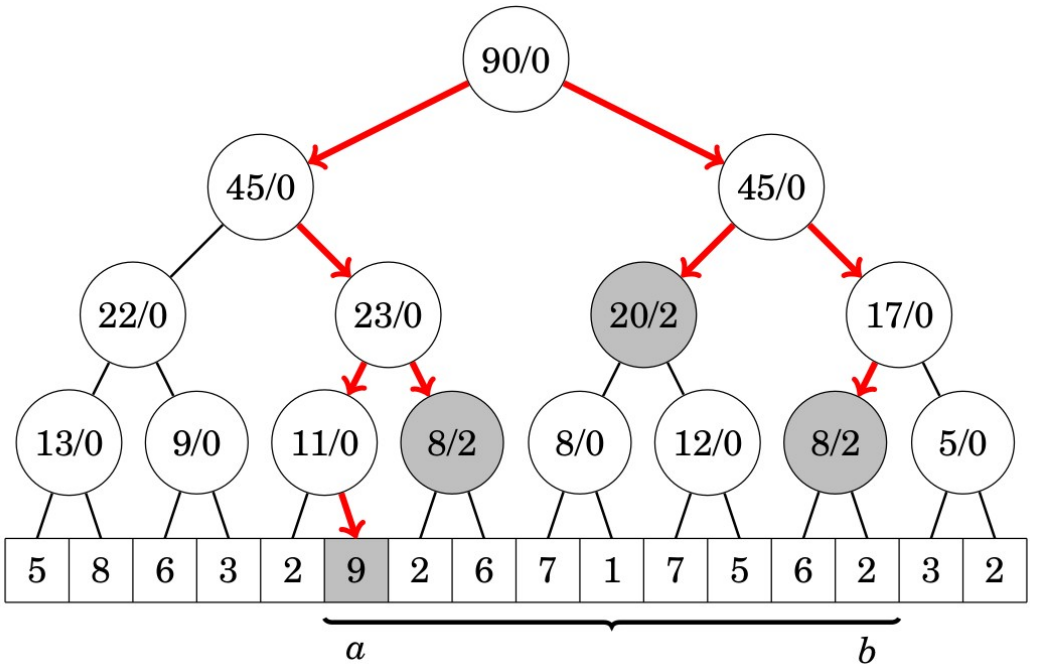
Lazy Segment Tree (1)

- Aggiunge **lazy propagation** al segment tree
- Utile per combinare range query e range update
- Ogni nodo può avere aggiornamenti non ancora propagati
- Vediamo lazy segment tree per sum range queries

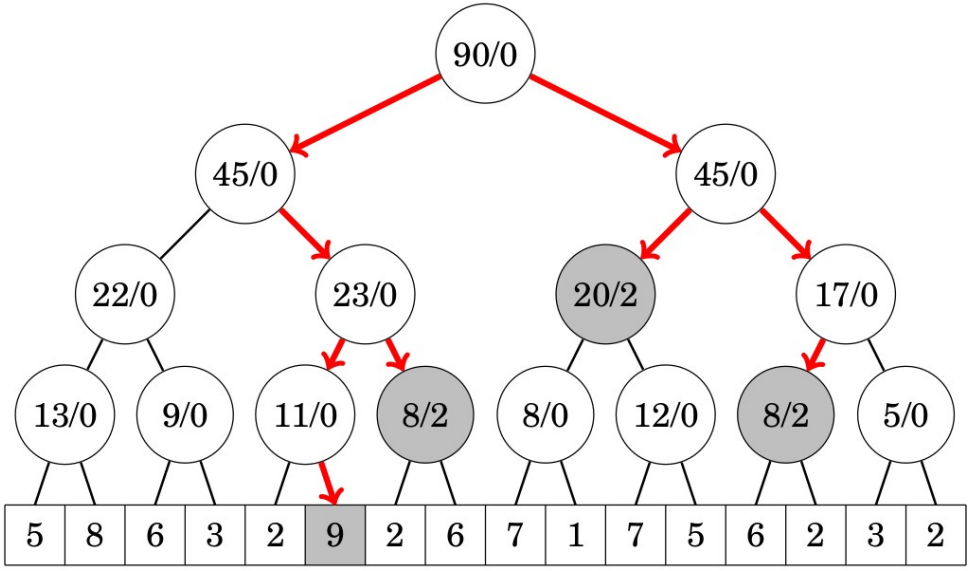
Lazy Segment Tree (2)



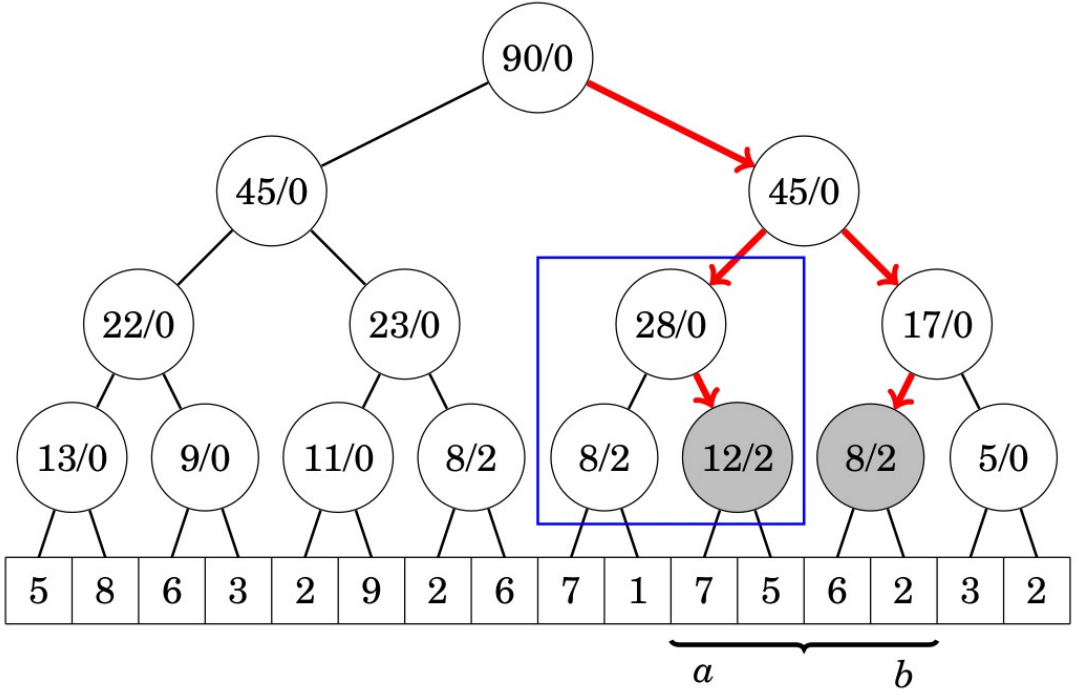
add(2,a,b)



Lazy Segment Tree (3)



$sum_q(a,b)$



Lazy Segment Tree (4)

```
void add(lazy_segment_tree& tree, int a, int b, int val) {
    add_util(tree, a, b, val, 1, 0, tree.size()/2 - 1);
}
```

```
void add_util(lazy_segment_tree& tree, int a, int b, int val, int node, int from, int to) {
    if(from > to || from > b || to < a) return;     // out of range

    // [from,to] is contained into [a,b]: update all elements (lazily)
    if(a <= from && to <= b) {
        tree[node].second += val;
        return;
    }

    lazy_propagation(tree, node, to - from + 1);

    // split [from,to] and repeat
    int mid = (from + to) / 2;
    add_util(tree, a, b, val, 2*node, from, mid);
    add_util(tree, a, b, val, 2*node + 1, mid + 1, to);
    lazy_propagation(tree, 2*node, mid - from + 1);
    lazy_propagation(tree, 2*node + 1, to - mid);
    tree[node].first = tree[2*node].first + tree[2*node + 1].first;
}
```

Lazy Segment Tree (5)

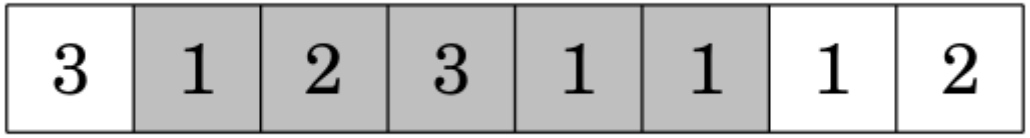
```
int sumq(lazy_segment_tree& tree, int a, int b) {
    return sumq_util(tree, a, b, 1, 0, tree.size()/2 - 1);
}
```

```
int sumq_util(lazy_segment_tree& tree, int a, int b, int node, int from, int to) {
    if(from > to || from > b || to < a) return 0;     // out of range

    lazy_propagation(tree, node, to - from + 1);

    // [from,to] is contained into [a,b]
    if(a <= from && to <= b) return tree[node].first;

    // split [from,to] and repeat
    int mid = (from + to) / 2;
    return sumq_util(tree, a, b, 2*node, from, mid) + sumq_util(tree, a, b, 2*node + 1, mid + 1, to);
}
```

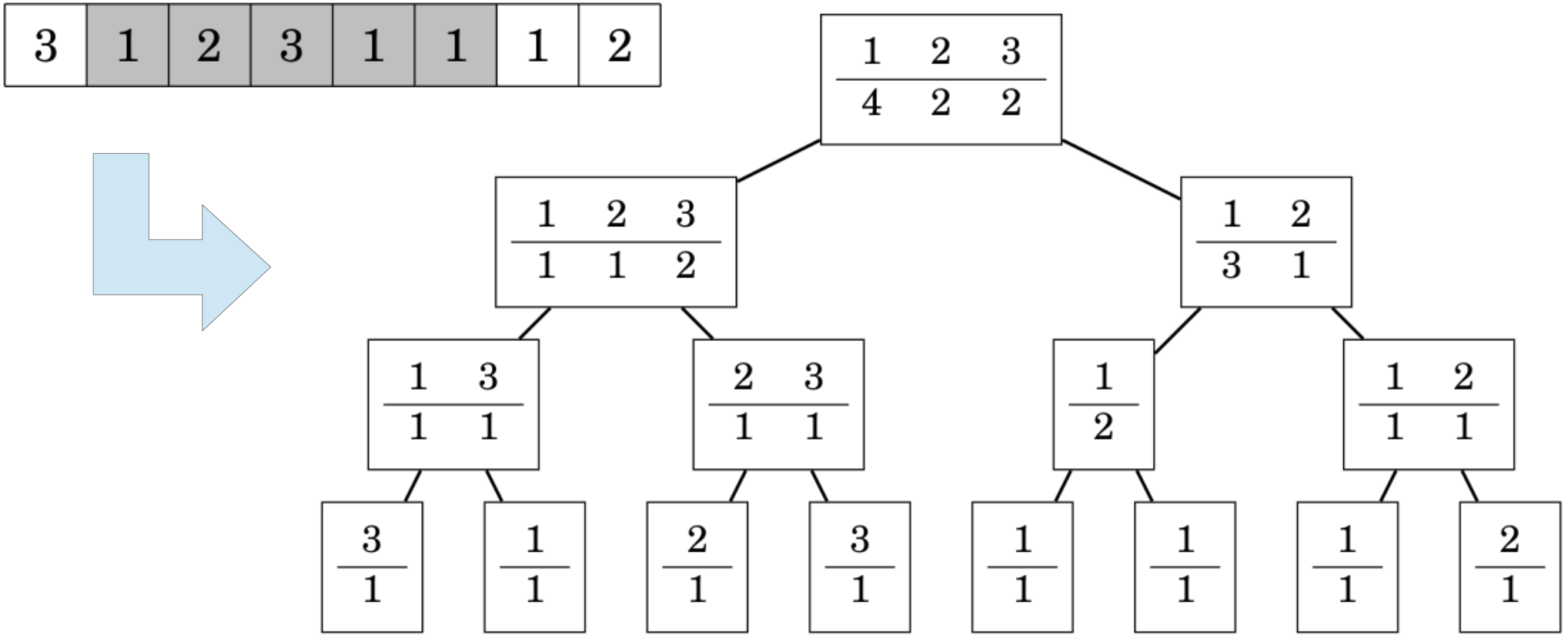
```
void lazy_propagation(lazy_segment_tree& tree, int node, int length) {
    if(tree[node].second == 0) return;
    if(length > 1) {
        tree[2*node].second += tree[node].second;
        tree[2*node + 1].second += tree[node].second;
    }
    tree[node].first += tree[node].second * length;
    tree[node].second = 0;
}
```

Segment Tree con strutture dati (1)

- Utili per contare il numero di elementi con una certa proprietà
- Ad esempio, quante volte compare 1 in un range?



Segment Tree con strutture dati (2)



Esercizi suggeriti

- Pile di mattoni (mattoni)
  - Usare difference array
- Wheel of Fortune (wheel)
  - Usare due segment tree, per minimo e massimo
  - Usare aritmetica modulare per individuare gli indici corretti
- Array Partition (partition)
  - Usare due lazy segment tree, per elementi fuori posto a sinistra e destra
- Flipping Coins (rangetree1)
  - Usare lazy segment tree e memorizzare il numero di **teste** in ogni nodo
- Multiples of 3 (rangetree2)
  - Usare lazy segment tree e memorizzare il numero di 0 e di 1 in ogni nodo
- Place (place)
  - Riordinare gli indici in modo che tutti i figli di un nodo occupino posizioni contigue
  - Usare lazy segment tree

Fine della lezione