

Query su array: strategie e strutture dati

Gabriele Farina e Francesco Vercellesi

27 luglio 2024

Indice

1	Introduzione	2
1.1	Informazioni facilmente unibili (IFU)	2
1.2	Massimo sottoarray	2
2	Prefix/Suffix precomputations	3
2.1	L'idea	3
3	Fenwick Trees	4
3.1	Introduzione	4
3.2	Struttura interna	5
3.3	Query	5
3.4	Update	6
3.5	Alcuni commenti intermedi	8
3.6	♣ Numero di inversioni	9
3.7	♣ Massima sottosequenza crescente (LIS)	10
3.8	Range updates	12
4	Sparse Tables	13
4.1	(Static) Range Min Query (RMQ)	13
4.2	Struttura interna	14
4.3	Costruzione	15
4.4	Query	15



1 Introduzione

Mantenere ed estrarre velocemente informazioni da un array è una richiesta che capita molto spesso. Molti sottoproblemi si riducono a questo tipo di attività: dalla programmazione dinamica agli algoritmi su grafi, saper determinare velocemente certe grandezze (ad esempio un minimo o una media) circa una certa porzione di dati può essere un passo cruciale.

1.1 Informazioni facilmente unibili (IFU)

Una certa quantità circa un array è “facilmente unibile” quando, una volta separato l’array in due parti, essa può essere facilmente determinata utilizzando un approccio *divide et impera*.

Alcuni esempi di IFU:

- Somma, prodotto, xor, min, gcd (massimo comun divisore). Ad esempio, la somma degli elementi di un array si calcola facilmente conoscendo la somma degli elementi delle sue due metà.
- Numero di elementi con valore $\leq x$. Infatti, il numero di elementi con valore $\leq x$ è pari alla somma del numero di elementi con valore $\leq x$ nelle due metà dell’array originale.
- Valore del massimo sottoarray, come mostrato nella prossima sezione.

1.2 Massimo sottoarray

Supponiamo di avere un array $a = \langle a_1, a_2, \dots, a_{2n} \rangle$ di $2n$ elementi, e di voler calcolare il valore del massimo sottoarray utilizzando un approccio *divide et impera*¹. Indicata con $a^l = \langle a_1, \dots, a_n \rangle$ la metà sinistra, e con $a^r = \langle a_{n+1}, \dots, a_{2n} \rangle$ la metà destra dell’array, è facile convincersi che il valore $\text{maxsub}(a)$ del massimo sottoarray di a è dato da

$$\text{maxsub}(a) = \max \left\{ \begin{array}{c} \text{maxsub}(a^l), \\ \text{maxsub}(a^r), \\ \text{maxsuff}(a^l) + \text{maxpref}(a^r) \end{array} \right\},$$

dove maxsuff e maxpref rappresentano rispettivamente il valore del massimo suffisso e prefisso dell’argomento.

¹Esistono diversi algoritmi efficienti per il problema del massimo sottoarray, tra cui ad esempio l’algoritmo di Kadane. Tuttavia, siamo qui interessati a mostrare che l’informazione del massimo sottoarray è facilmente unibile, e per questo stiamo cercando una caratterizzazione *divide et impera*.

Al fine di caratterizzare completamente l'algoritmo, è necessario mostrare che anche maxsuff e maxpref sono IFU. D'altra parte, indicata con sum la funzione somma degli elementi di un array², vale:

$$\begin{aligned}\text{maxpref}(a) &= \max \left\{ \begin{array}{c} \text{maxpref}(a^l), \\ \text{sum}(a^l) + \text{maxpref}(a^r) \end{array} \right\}, \\ \text{maxsuff}(a) &= \max \left\{ \begin{array}{c} \text{maxsuff}(a^r), \\ \text{maxsuff}(a^l) + \text{sum}(a^r) \end{array} \right\}.\end{aligned}$$

2 Prefix/Suffix precomputations

Introduciamo il seguente problema:

Sia $a = \langle a_1, \dots, a_n \rangle$ un array di n interi. Vogliamo supportare, il più velocemente possibile, la seguente query:

Dati l e r , quanto vale $\Sigma(l, r) = a_l + a_{l+1} + \dots + a_r$?

2.1 L'idea

La soluzione più semplice del problema ricalcola ogni volta il valore di Σ scorrendo gli elementi da l a r di a mantenendo un accumulatore per la risposta. Al caso peggiore questo approccio ha un costo computazionale pari a $\mathcal{O}(n)$ per query. Possiamo fare molto meglio.

Ad esempio, possiamo notare che esistono al massimo $\Theta(n^2)$ valori diversi di Σ . Questo porta ad un algoritmo ammortizzato che comincia ad essere vantaggioso quando il numero di query è molto elevato.

La soluzione più efficiente tuttavia passa dal fatto che, dal momento che gli elementi del vettore a non cambiano, è conveniente precalcolare certe informazioni su a che ci permettono di calcolare Σ più efficientemente. In particolare, introduciamo il vettore s così definito:

$$s_0 = 0, \quad s_1 = a_1, \quad \dots, \quad s_n = a_1 + a_2 + \dots + a_n.$$

È allora evidente che

$$\Sigma(l, r) = a_l + a_{l+1} + \dots + a_r = s_r - s_{l-1}.$$

Il vettore s è detto *vettore delle somme prefisse*, poiché contiene, per ogni i , la somma del prefisso di lunghezza i di a . È possibile costruire s usando $\mathcal{O}(n)$

²Si noti che sum è una IFU.

operazioni. Il costo di rispondere ad ogni query Σ è costante (è sufficiente fare due accessi alla memoria e una sottrazione).

3 Fenwick Trees

Introduciamo ora un'estensione del problema precedente:

Sia $a = \langle a_1, \dots, a_n \rangle$ un array di n interi. Vogliamo supportare, il più velocemente possibile, le seguenti operazioni:

- Dati l e r , quanto vale $\Sigma(l, r) = a_l + a_{l+1} + \dots + a_r$?
- Dato un indice j ed un intero δ , assegnare $a_j \leftarrow a_j + \delta$.

Si noti che la seconda operazione è del tutto equivalente a dare un indice j ed un valore v , e assegnare $a_j \leftarrow v$.

È importante notare che la soluzione di prima, basata sull'idea delle somme prefisse, non si sposa bene con l'operazione di update introdotta. Infatti, mentre il calcolo di Σ risulta ancora $\mathcal{O}(1)$, l'assegnamento $a_j \leftarrow a_j + \delta$ potrebbe, nel caso peggiore, comportare la necessità di ricalcolare daccapo l'intero vettore s delle somme prefisse, con un costo associato pari a $\mathcal{O}(n)$.

Introdurremo ora una struttura dati, i *Fenwick trees*, capace di supportare entrambe le query in tempo $\mathcal{O}(\log n)$.

3.1 Introduzione

A differenza di quanto il nome suggerisce, è più facile pensare ai Fenwick trees come array piuttosto che come alberi. L'idea alla base dei Fenwick tree è trovare un modo efficiente di suddividere l'array iniziale in parti, sfruttando l'ipotesi di IFU.

Dato un array lungo n , la scomposizione alla base dei Fenwick tree è tale che vengono individuate n parti e che ogni prefisso è unione di $\mathcal{O}(\log n)$ parti. Allo stesso modo, nell'aggiornare un valore dell'array si rende necessario aggiornare $\mathcal{O}(\log n)$ parti. Assumendo che l'unione di due parti e l'aggiornamento di una singola parte siano operazioni eseguibili in tempo costante, si deduce che ogni query e ogni update costa $\mathcal{O}(\log n)$ tempo. L'implementazione di un Fenwick tree è molto semplice: richiede circa 15 righe di codice.

3.2 Struttura interna

Dato l'array $a = \langle a_1, a_2, \dots, a_n \rangle$ di dimensione n , il Fenwick Tree associato ad a , per l'operazione di somma, è un secondo array t di dimensione n , tale per cui

$$t_k = a_{\tilde{k}+1} + \dots + a_k,$$

dove $\tilde{k} = k - \text{lsb}(k)$ è il valore di k privato del suo bit meno significativo (lsb, dall'inglese *least significant bit*). Un esempio è mostrato in Figura 1.

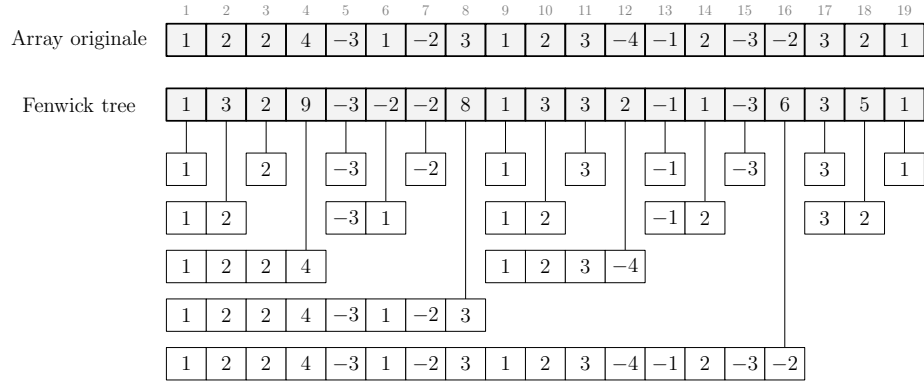


Figura 1: Struttura di un Fenwick tree.

3.3 Query

È facile determinare la somma del prefisso $a_1, \dots, a_k$ in tempo $\mathcal{O}(\log k) = \mathcal{O}(\log n)$. Nell'esempio di Figura 2 sotto k vale 13.

Infatti, per trovare la somma degli elementi da 1 a 13, operiamo così, aggiungendo un bit alla volta:

- somma degli elementi da 0 (00000₂) escluso a 8 (01000₂) incluso. Questo valore coincide con $t_8 = 8$.
- somma degli elementi da 8 (01000₂) escluso a 12 (01100₂) incluso. Questo valore coincide con $t_{12} = 2$.
- somma degli elementi da 12 (01100₂) escluso a 13 (01101₂) incluso. Questo valore coincide con $t_{13} = -1$.

In generale, dal momento che ad ogni passaggio stiamo aggiungendo un bit all'estremo superiore dell'intervallo di valori da sommare, il numero di operazioni

richieste per calcolare la somma è pari al numero di bit settati di k , e quindi $\mathcal{O}(\log k)$.

È comodo per l'implementazione notare che gli indici di t che andiamo a sommare sono quelli dati dalla sequenza:

$$\begin{aligned} i_1 &= k \\ i_2 &= i_1 - \text{lsb}(i_1) \\ i_3 &= i_2 - \text{lsb}(i_2) \\ &\dots \end{aligned}$$

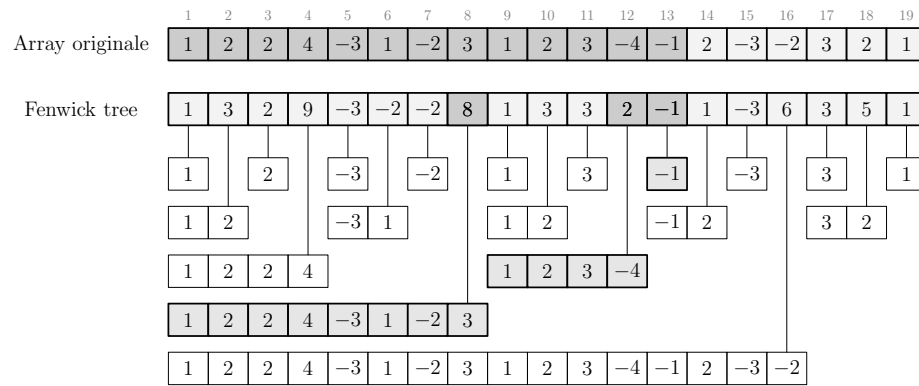


Figura 2: Query in un Fenwick tree.

■ Implementazione

```
long long sum(int k) {
    long long res = 0;

    // (i & -i) == lsb(i)
    for (int i = k; i > 0; i -= (i & -i)) {
        res += ft[i];
    }

    return res;
}
```

3.4 Update

Supponiamo di aver modificato il valore in posizione j dell'array, aggiungendogli una quantità δ , positiva o negativa. Per semplicità riferiamoci all'esempio di Figura 1, avendo posto $j = 5, \delta = 4$.

Array originale

1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19
1	2	2	4	1	1	-2	3	1	2	3	-4	-1	2	-3	-2	3	2	1

Fenwick tree

1	3	2	9	1	2	-2	12	1	3	3	2	-1	1	-3	10	3	5	1
1		2		1		-2		1		3		-1		-3		3		1
1	2			1	1			1	2			-1	2			3	2	
1	2	2	4					1	2	3	-4							
1	2	2	4	1	1	-2	3											
1	2	2	4	1	1	-2	3	1	2	3	-4	-1	2	-3	-2			

Nel caso di Figura 3, è necessario aggiornare l'array t in questo modo:

$$t_5 \leftarrow t_5 + 4, \quad t_6 \leftarrow t_6 + 4, \quad t_8 \leftarrow t_8 + 4, \quad t_{16} \leftarrow t_{16} + 4.$$

$$\begin{array}{lll} i_1 \leftarrow j & \longrightarrow & t_{i_1} \leftarrow t_{i_1} + \delta \\ i_2 \leftarrow i_1 + \text{lsb}(i_1) & \longrightarrow & t_{i_2} \leftarrow t_{i_2} + \delta \\ i_3 \leftarrow i_2 + \text{lsb}(i_2) & \longrightarrow & t_{i_3} \leftarrow t_{i_3} + \delta \\ & \vdots & \end{array}$$

■ Implementazione

```
void update(int k, int delta) {  
    // (i & -i) == lsb(i)  
    for (int i = k; i < ft.size(); i += (i & -i)) {  
        ft[i] += delta;  
    }  
}
```

☞ Attenzione: la struttura mostrata è 1-based, chiamare `update(0,  $\delta$ )` causa un loop infinito.

3.5 Alcuni commenti intermedi

1. La struttura del Fenwick tree si estende in modo naturale quando, invece della somma, si è interessati ad altre operazioni **associative** e **invertibili**. L'invertibilità è un tassello *fondamentale* per poter ottenere informazioni su sottoarray che non sono prefissi o suffissi in tempo logaritmico nella dimensione dell'array.
2. Non è necessario scrivere codice ad-hoc per inizializzare un Fenwick tree a partire da un vettore già riempito: per creare il Fenwick tree corrispondente basta chiamare ripetutamente la funzione `update` per inserire uno ad uno i valori del vettore nel Fenwick tree.
3. È facile rendere la struttura 0-based rimappando silenziosamente gli indici (notate le istruzioni `i = k + 1` nei cicli `for`):

```
long long sum(int k) {
    long long res = 0;

    for (int i = k + 1; i > 0; i -= (i & -i)) {
        res += ft[i];
    }

    return res;
}

void update(int k, int delta) {
    for (int i = k + 1; i < ft.size(); i += (i & -i)) {
        ft[i] += delta;
    }
}
```

☞ Attenzione: il vettore `ft` deve essere comunque di dimensione $n + 1$.

3.6 ♣ Numero di inversioni

Prima di continuare, fermiamoci a contemplare un problema classico, ovvero quello del conteggio del numero di inversioni in un array.

Data una permutazione $a = \langle a_1, a_2, \dots, a_n \rangle$ degli interi $1, \dots, n$, una *inversione* è definita come una coppia di indici (i, j) con $i < j$, per cui vale la condizione $a_i > a_j$.

Data la permutazione, contare il numero di inversioni in tempo $\mathcal{O}(n \log n)$.

★ Fissiamo un elemento della coppia

Quando un problema ci chiede di enumerare delle coppie, una idea **estremamente generale e ricorrente** consiste nel fissare uno degli elementi della coppia e provare a vedere quanto velocemente si è in grado di determinare le posizioni possibili per l'altro estremo.

In questo caso scegliamo di fissare il secondo elemento j (il caso con i è esattamente simmetrico): siamo ora interessati a contare per quanti $i < j$ accade che $a_i > a_j$. Notiamo che una volta fissato il valore j anche il valore di a_j è fissato. In altre parole, fissato j , il problema si riduce a determinare, all'interno del suffisso $a_1, a_2, \dots, a_{j-1}$, quanti sono i valori inferiori ad a_j .

★ Costruiamo induttivamente la risposta

Supponiamo di possedere una struttura dati in grado di supportare queste operazioni:

- Aggiungere un dato numero nel range $[1, n]$ alla collezione
- Dato v , contare quanti numeri sono maggiori di v .

Una tale struttura ci permetterebbe di risolvere induttivamente la risposta, usando il seguente algoritmo:

```
long long conta_inversioni(vector<int> a) {
    int inversioni = 0;
    for (int j = 0; j < n; j++) {
        inversioni += struttura.quanti_maggiori(a[j]);
        struttura.aggiungi_numero(a[j]);
    }
    return inversioni;
}
```

La struttura in realtà non è nient'altro che un Fenwick tree. Consideriamo infatti un array c , dove c_i contiene quanti numeri di valore i sono stati inseriti nella struttura.

La chiamata a `aggiungi_numero(v)` si riduce allora a `update(v,  $\delta = 1$ )`, mentre `quanti_magiori(v)` si riduce a `sum(v + 1, n)`.

3.7 ♣ Massima sottosequenza crescente (LIS)

Studiamo un altro problema classico, ovvero quello della ricerca della massima sottosequenza comune (LIS, dall'inglese *Longest Increasing Subsequence*).

Dato un array a di n valori nel range $[1, n]$, trovare la lunghezza della sottosequenza strettamente crescente (non necessariamente contigua) più lunga.

Dato l'array, trovare la risposta in $\mathcal{O}(n \log n)$.

★ Fissiamo un estremo della sottosequenza

In modo del tutto analogo a quanto visto prima, fissiamo un estremo della risposta. In questo caso, fissato j siamo interessati a conoscere la lunghezza di una LIS che termina con a_j .

Indicata con $\text{lis}(i)$ la lunghezza di ogni LIS massima che termina in a_i , sappiamo che

$$\text{lis}(j) = 1 + \max_{i < j \wedge a_i < a_j} \text{lis}(i)$$

(nel caso in cui non esista nessun i che rispetti la condizione, il max vale 0).

★ Costruiamo induttivamente la risposta

Supponiamo di possedere un dizionario che supporti le due operazioni:

- Inserire una coppia (chiave, valore).
- Restituire il massimo dei valori aventi chiave minore di v .

Il problema in effetti è semplicemente una versione più avanzata di quanto accadeva nel caso delle inversioni, in cui chiave e valore coincidevano. Non dobbiamo dimenticare che anche in questo caso le chiavi appartengono all'intervallo $[1, n]$.

Una tale struttura ci permetterebbe di risolvere induttivamente la risposta, usando il seguente algoritmo:

```

int longest_increasing_subsequence(vector<int> a) {
    int lis = 0;
    for (int j = 0; j < n; j++) {
        int lunghezza = 1 + struttura.max_chiavi_minori(a[j]);
        lis = std::max(lis, lunghezza);
        struttura.inserisci(a[j], lunghezza);
    }
    return lis;
}

```

La struttura dati si ottiene considerando un Fenwick tree costruito rispetto alla funzione massimo: l'inserimento coincide con l'operazione **update**(chiave, valore), mentre la ricerca del massimo corrisponde al metodo che prima avevamo chiamato **sum**.

```

int query(int k) {
    int res = 0;

    for (int i = k; i > 0; i -= (i & -i)) {
        res = max(res, ft[i]);
    }

    return res;
}

void update(int k, int new_val) {
    for (int i = k; i < ft.size(); i += (i & -i)) {
        ft[i] = max(ft[i], new_val);
    }
}

```

3.8 Range updates

Esiste un “trucco” che è bene conoscere, in quanto viene comodo meno raramente di quanto possa sembrare.

Supponiamo di dover risolvere il problema:

Sia $a = \langle a_1, \dots, a_n \rangle$ un array di n interi. Vogliamo supportare, il più velocemente possibile, le seguenti operazioni:

- Dato un indice j , quanto vale a_j ?
- Dati due indici $l \leq r$ ed un intero δ , assegnare $a_j \leftarrow a_j + \delta$ per tutti gli indici j compresi tra l e r inclusi.

I Fenwick tree supportano entrambe le operazioni in tempo $\mathcal{O}(\log n)$. La chiave di volta risiede nel non mantenere a , bensì la sua “derivata discreta”, ovvero il vettore d delle differenze finite di a :

$$d_i \triangleq \begin{cases} a_1 & \text{se } i = 1 \\ a_i - a_{i-1} & \text{se } i > 1 \end{cases}.$$

Un esempio di vettore delle differenze finite è mostrato in Figura 4.

	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19
Array originale	1	2	2	1	1	1	1	-2	2	2	2	2	2	2	3	3	4	4	1
Differenze finite	1	1	0	-1	0	0	0	-3	4	0	0	0	0	0	1	0	1	0	-3

Figura 4: Esempio di vettore delle differenze finite.

Questa rappresentazione ha la proprietà che

$$a_k = d_1 + \dots + d_k \quad \forall k \in \{1, \dots, n\}.$$

Inoltre, come mostrato in Figura 5, aggiungere una quantità δ a $a_l, \dots, a_r$ si traduce su d nelle due operazioni elementari:

$$d_l \leftarrow d_l + \delta, \quad d_{r+1} \leftarrow d_{r+1} - \delta \quad (\text{ammesso che } r+1 \leq n).$$

È quindi sufficiente costruire il Fenwick tree associato a d e mantenere quello.

	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19
Array originale	1	2	2	1	1	2	2	-1	3	3	3	3	2	2	3	3	4	4	1
Differenze finite	1	1	0	-1	0	1	0	-3	4	0	0	0	-1	0	1	0	1	0	-3

Figura 5: Operazioni sul vettore delle differenze finite ($l = 6, r = 12, \delta = 1$).

4 Sparse Tables

Consideriamo una categoria speciale di IFU che non sono invertibili (e quindi non possono essere maneggiate efficacemente dai Fenwick tree): le IFU **idempotenti**.

Si tratta di tutte quelle informazioni tali che possono essere ricostruite anche a partire da suddivisioni in parti con intersezione non nulla. Alcuni esempi includono:

- min, max
- gcd (massimo comun divisore).

In particolare, il problema di trovare il minimo in sottoarray di un array statico è un problema particolarmente studiato data la sua connessione al problema del minimo antenato comune (LCA, dall'inglese *Lowest Common Ancestor*) negli alberi.

Le sparse tables sono strutture dati molto semplici in grado di risolvere il problema statico (quindi senza aggiornamento dei valori) di trovare l'informazione idempotente richiesta in tempo costante a valle di un preprocessing tendenzialmente $\mathcal{O}(n \log n)$, con n la dimensione dell'array di partenza.

4.1 (Static) Range Min Query (RMQ)

Consideriamo il seguente problema, detto “problema RMQ” dall'inglese *Range Min Query*:

Sia $a = \langle a_1, a_2, \dots, a_n \rangle$ un array di n interi. Vogliamo supportare, il più velocemente possibile, la seguente query:

Dati l e r , quanto vale $m(l, r) = \min\{a_l, a_{l+1}, \dots, a_r\}$?

Si noti che nonostante l'apparente somiglianza con il problema della somma statica studiato in precedenza, in questo caso non è possibile utilizzare delle precomputazioni legate a prefissi e suffissi, nè i Fenwick trees. Sorprendente-

mente, è possibile ideare una struttura dati, la Sparse Table appunto, in grado di risolvere questo problema in tempo costante per ogni query.

4.2 Struttura interna

Siano $a_1, \dots, a_n$ gli n valori dell'array. La sparse table associata a questi è una matrice. Nel caso della Sparse Table costruita per la funzione $\min$, alla riga h , in posizione j è memorizzata la quantità

$$t_{h,j} = \min\{a_j, a_{j+1}, \dots, a_{j+2^h-1}\}.$$

$h = 4$	-4	-4	-4	-4																	
$h = 3$	-3	-3	-3	-3	-3	-4	-4	-4	-4	-4	-4	-4									
$h = 2$	-1	1	-3	-3	-3	-3	-2	-2	1	-4	-4	-4	-4	-3	-3	-3					
$h = 1$	-1	1	2	2	-3	-3	-2	-2	1	1	2	-4	-4	-1	-3	-3	-2	2			
$h = 0$	-1	1	2	2	4	-3	1	-2	3	1	2	3	-4	-1	2	-3	-2	3	2		
Array originale	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19		
	-1	1	2	2	4	-3	1	-2	3	1	2	3	-4	-1	2	-3	-2	3	2		

Figura 6: Esempio di Sparse Table.

Si noti che per $h = 0$ la riga della matrice coincide con l'array originale. Inoltre, il numero di posizioni sensate sono circa $n \log n$: $\log n$ righe da circa n valori ognuna.

4.3 Costruzione

Costruire una sparse table è particolarmente semplice:

- La riga 0 coincide con l'array originale.
- Per le righe oltre la 0 vale la relazione $t_{h,j} = \min\{t_{h-1,j}, t_{h-1,j+2^{h-1}}\}$.

```
vector<vector<int>> build_sparse_table(vector<int> a) {
    vector<vector<int>> t;
    t.push_back(a);

    for (int h = 1; (1 << h) <= a.size(); h++) {
        int len = 1 << h;
        t.push_back(vector<int>(a.size() - len + 1));

        for (int j = 0; j < t[h].size(); j++) {
            t[h][j] = min(t[h - 1][j], t[h - 1][j + len / 2]);
        }
    }

    return t;
}
```

4.4 Query

Per trovare in tempo logaritmico il minimo degli elementi dalla posizione l alla posizione r inclusa, troviamo il più grande h tale che $2^h \leq r - l + 1$. La risposta alla query è allora

$$\min\{t_{h,l}, t_{h,r-2^h+1}\}.$$

```
int query(const vector<vector<int>> &t, int l, int r) {
    int h = 0;
    while ((1 << (h + 1)) <= (r - l + 1)) {
        h++;
    }

    return min(t[h][l], t[h][r + 1 - (1 << h)]);
}
```

Per risparmiare il fattore logaritmico associato alla ricerca del valore di h , possiamo precalcolare per ogni possibile valore di $r - l + 1$ il valore di h associato.