Tecniche square-root

Alessandro Bortolin

Online, 18 aprile 2021

Introduzione

Perché cercare algoritmi $\mathcal{O}(n\sqrt{n})$?

- A volte la soluzione $\mathcal{O}(n \log n)$ non esiste oppure è troppo difficile. 
- Sono tecniche flessibili che si possono adattare a molti problemi. 
- Alcune di queste tecniche sono **offline**. 
- Non sempre è facile far stare le soluzioni nei tempi. 

In pratica un algoritmo $\mathcal{O}(n\sqrt{n})$ riesce a risolvere istanze fino a $n = 3 \cdot 10^5$.

Tecniche

Quali sono le tecniche più comuni?

1. Square-root decomposition
2. Specializzazione degli algoritmi
3. Algoritmo di Mo
4. Knapsack

Square-root decomposition

La **square-root decomposition** è una tecnica che permette di create strutture dati efficienti, dividendo l'array in **blocchi** di $\sqrt{n}$ elementi e memorizzando informazioni aggiuntive per ogni blocco.

Questa tecnica è in grado di gestire **range update** e **range query** in $\mathcal{O}(\sqrt{n})$.

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

Figura: esempio in cui ogni blocco contiene le informazioni sull'elemento minimo

Minimum query

Problema

Sia dato un array di n elementi e q query, ciascuna query può essere:

- aggiungere x a tutti gli elementi tra l e r;
- stampare l'elemento più piccolo tra l e r.

Possiamo risolvero mediante un segment tree, vediamo un'altra tecnica!

Minimum query

Soluzione senza update

Dividiamo l'array in blocchi di k elementi ciascuno, per ciascun blocco calcoliamo il suo valore minimo.
Per rispondere ad una query:

- iteriamo su tutti i blocci **totalmente inclusi** nel range e calcoliamo valore il minimo;
- iteriamo sui restanti valori singolarmente.

Per ogni query iteriamo al massimo su $\mathcal{O}(\frac{n}{k} + k)$ valori.

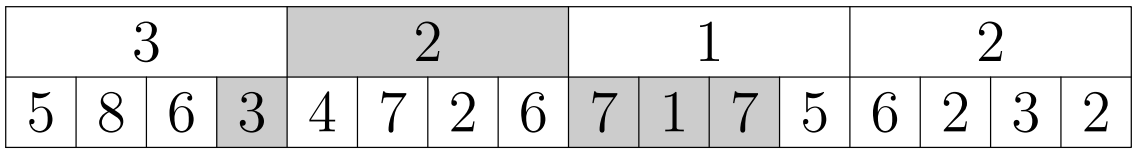


Figura: esempio di query con $l = 3$ e $r = 14$

Minimum query

Soluzione con update

Per ogni blocco memorizziamo due valori:

- `block_min[`i`]`: il valore minimo del blocco;
- `block_inc[`i`]`: l'incremento totale del blocco.

Per eseguire un update:

- se un blocco è interamente incluso nell'intervallo, aggiorniamo `block_inc[`i`]`;
- se un blocco è parzialmente incluso nell'intervallo, aggiorniamo singolarmente i valori e ricalcoliamo `block_min[`i`]`.

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

Figura: esempio di update con $l = 6$, $r = 13$ e $x = 2$

Minimum query

Complessità

Per ogni update:

- aggiorniamo `block_inc` al massimo $\frac{n}{k}$ volte;
- ricalcoliamo al massimo 2 valori di `block_min`.

La complessità per ogni update è, come per le query, $\mathcal{O}(\frac{n}{k} + k)$.
Il valore k che minimizza l'espressione è $\sqrt{n}$, quindi la complessità finale è $\mathcal{O}(\frac{n}{\sqrt{n}} + \sqrt{n}) = \mathcal{O}(\sqrt{n})$.

Vasi 1

Problema

Ci sono n vasi disposti in linea, ciascun vaso ha un valore. Inizialmente i valori sono $0, 1, \ldots, n-1$. Vengono date inoltre q query, ciascuna query può essere:

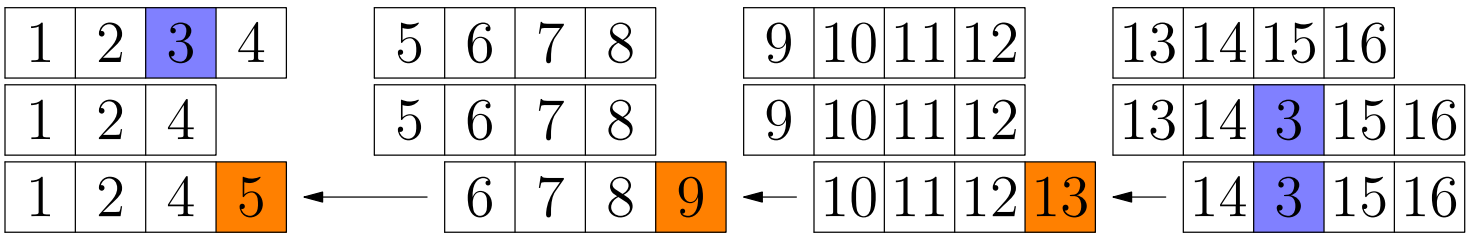
- stampare il valore del k-esimo vaso;
- spostare il vaso in posizione x alla posizione y mantenento l'ordine degli altri vasi.

Vasi 1

Soluzione

Dividiamo l'array in blocchi di $\sqrt{n}$ vasi, rappresentiamo ciascun blocco mediante una deque. Dato che tutti i blocchi hanno la stessa dimensione, possiamo determinare il k-esimo elemento in $\mathcal{O}(1)$.
Per spostare un vaso:

- rimuoviamo il vaso dal suo blocco;
- aggiungiamo il vaso al nuovo bloco;
- per ogni blocco in mezzo rimuoviamo il primo vaso e aggiungiamo in fondo alla deque il primo vaso del blocco successivo.



Vasi 1

Complessità

- Rimuovere e aggiungere un vaso da un blocco ha complessità $\mathcal{O}(\sqrt{n})$.
- Possiamo aggiungere o eliminare un vaso dalle estremità mediante le funzione della deque `pop_front`, `pop_back`, `push_front` e `push_back` con complessità $\mathcal{O}(1)$.

Eliminiamo e aggiungiamo un vaso dalle estremità di un blocco al massimo $\sqrt{n}$ volte, la complessità per ogni query è quindi $\mathcal{O}(\sqrt{n})$.

Specializzazione degli algoritmi

Alcuni problemi possono essere risolti efficientemente creando **due algoritmi** che risolvono determinate casistiche del problema.

Sebbene entrambi gli algoritmi possano essere usati singolarmente per risolvere il problema, **combinandoli** si ottiene un algoritmo più veloce ed efficiente.

Griglia colorata 1

Problema

Sia data un griglia $n \times n$ in cui in ogni casella contiene un colore rappresentato da un numero intero, determinare la minima distanza di Manhattan tra due caselle con lo stesso colore.

Soluzione 1

Per ogni colore t, iteriamo su tutte le coppie di caselle di tale colore e calcoliamo la distanza minima tra esse.
Per ogni colore la complessità è $\mathcal{O}(k^2)$ dove k è il numero di occorrenze di t.

Soluzione 2

Per ogni colore t, eseguiamo una BFS partendo da tutte le caselle di colore t.
Per ogni colore la complessità $\mathcal{O}(n^2)$.

Griglia colorata 1

Qual è la complessità dei due algoritmi?

Soluzione 1

Il caso peggiore è quando tutte le caselle hanno lo stesso colore, $k = n^2$ quindi la complessità è $\mathcal{O}(k^2) = \mathcal{O}(n^4)$.

Soluzione 2

Il caso peggiore è quando tutte le caselle hanno un colore diverso, la complessità è $\mathcal{O}(n^2 \cdot n^2) = \mathcal{O}(n^4)$.

Possiamo fare di meglio?

Griglia colorata 1

Osservazioni

- La soluzione 1 funziona bene quando un colore compare **poco** spesso.
- La soluzione 2 funziona bene quando un colore compare **molto** spesso.

Possiamo combinare i due algoritmi in un unico algoritmo più efficiente!

Nuova soluzione

- Utilizziamo la soluzione 1 quando un colore compare **al massimo** k volte.
- Utilizziamo la soluzione 2 quando un colore compare **almeno** k volte.

La complessità temporale è $\mathcal{O}(n^2 \cdot k + \frac{n^2}{k} \cdot n^2)$, il valore che minimizza l'espressione è $k = \sqrt{n^2} = n$, la complessità finale è $\mathcal{O}(n^3)$.

Griglia colorata 2

Problema

Sia data un griglia $n \times n$, inizialmente ogni casella è bianca. Ad ogni turno viene scelta una casella bianca e bisogna calcolare la minima distanza di Manhattan tra la casella ed una casella nera, al termine del turno la casella viene colorata di nero. Si ripete finché tutta la griglia è nera.

Soluzione 1

Ad ogni turno calcoliamo la distanza della casella da tutte le caselle nere. Calcolare la risposta ha complessità $\mathcal{O}(n^2)$, mentre aggiornare la griglia ha complessità $\mathcal{O}(1)$.

Soluzione 2

Per ogni casella memorizziamo la distanza minima da ogni casella nera, al termine di ogni turno aggiorniamo tutte le distanze minime. Calcolare la risposta ha complessità $\mathcal{O}(1)$, mentre aggiornare la griglia ha complessità $\mathcal{O}(n^2)$.

Griglia colorata 2

Possiamo combinare i due algoritmi in un unico algoritmo più efficiente!

Nuova soluzione

Processiamo i turni a gruppi di k:

- all'inizio di ogni gruppo di turni, calcolo la distanza minima di ogni casella, dalla casella nera più vicina;
- aggiungiamo le caselle che sono state colorate di nero nel gruppo corrente, ad un buffer;
- ad ogni turno calcoliamo la distanza da ogni casella nel buffer e la confrontiamo con il valore precalcolato all'inizio del gruppo.

La complessità temporale è $\mathcal{O}(\frac{n^2}{k} \cdot n^2 + n^2 \cdot k)$, il valore che minimizza l'espressione è $k = \sqrt{n^2} = n$, la complessità finale è $\mathcal{O}(n^3)$.

Piccioni in migrazione

Problema

Sia dato un albero di n nodi con radice 1, ogni nodo i ha un valore K_i.
Ci sono q query di due tipi:

- cambiare il padre di una foglia;
- dato un nodo x, calcolare la somma:

$$\sum_{i \in \text{subtree}(x)} (K_x - \text{dist}(i, x))$$

Piccioni in migrazione

Soluzione senza update

Precalcoliamo alcuni valori:

- `dist[`i`]`: distanza di i dalla radice;
- `size[`i`]`: dimensione del sotto-albero di i;
- `sum[`i`]`: somma dei valori di `dist[`j`]` per ogni j nel sotto-albero di i.

$$\begin{aligned}\sum_{i \in \text{subtree}(x)} (K_x - \text{dist}(i,x)) &= K_x \cdot \texttt{size[}x\texttt{]} - \sum_{i \in \text{subtree}(x)} \text{dist}(i,x) = \\ &= K_x \cdot \texttt{size[}x\texttt{]} - \texttt{sum[}i\texttt{]}\end{aligned}$$

Possiamo quindi precalcolare i valori mediante una DFS con complessità $\mathcal{O}(n)$ e rispondere ad ogni query con complessità $\mathcal{O}(1)$.

Piccioni in migrazione

Soluzione con update

Possiamo dividere le query in blocchi di $\sqrt{n}$, all'inizio di ogni blocco precalcoliamo i valori `dist`, `size` e `sum`. Per rispondere ad una query:

- per spostare un nodo lo aggiungiamo semplicemente all'interno di un buffer;
- per rispondere ad una query su x, calcoliamo la risposta utilizzando i valori precalcolati, poi per ogni nodo i nel buffer:
  - se la precedente posizione di i apparteneva al sotto-albero di x, sottraiamo $K_x - \texttt{dist}[i] + \texttt{dist}[x]$;
  - se la nuova posizione di i appartiene al sotto-albero di x, aggiungiamo $K_x - \texttt{dist}[i] + \texttt{dist}[x]$.

Complessità

Ricalcolare i valori con una DFS ha complessità $\mathcal{O}(n)$, in totale quindi $\mathcal{O}(q\sqrt{n})$.

Algoritmo di Mo

L'algoritmo di Mo è un algoritmo **offline** per processare query in un array **statico**, ovvero senza update. Ogni query viene fatta su un intervallo $[l, r)$

Il tipo di query deve essere, inoltre, **facilmente estendibile**, ovvero se conosco la risposta nell'intervallo $[l, r)$ posso calcolare la risposta in tempo breve per l'intervallo $[l, r + 1)$ e $[l, r - 1)$.

Dato che l'array è statico possiamo processare le query in **qualsiasi ordine.** Esiste un ordinamento con cui possiamo processare tutte le query in modo efficiente?

Algoritmo di Mo

L'algoritmo di Mo mantiene un intervallo con i valori già processati, ad ogni query l'intervallo corrente viene esteso o contratto nelle due direzioni fino a coincidere con il nuovo intervallo.

```
1 int cur_l = 0, cur_r = 0;
2 for (Query q : queries) {
3     while (cur_l > q.l) add(--cur_l);
4     while (cur_r < q.r) add(cur_r++);
5     while (cur_l < q.l) remove(cur_l++);
6     while (cur_r > q.r) remove(--cur_r);
7     answers[q.idx] = get_answer();
8 }
```

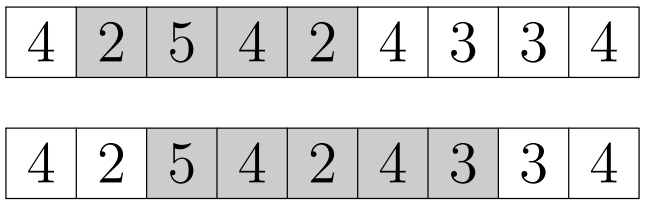


Figura: esempio in cui l'intervallo viene spostato

Algoritmo di Mo

Complessità

Ad ogni query l'intervallo viene esteso o contratto al più n volte, quindi la complessità è $\mathcal{O}(n \cdot q)$.

Possiamo fare di meglio?

Possiamo cambiare l'ordine con cui vengono eseguite le query in modo da minimizzare il numero di numeri di operazioni:

- dividiamo l'array in blocchi di $\sqrt{n}$ elementi e processiamo insieme le query che hanno l'estremo sinistro nello stesso blocco;
- se le query sono nello stesso blocco, le processiamo ordinate per l'estremo destro.

Nuova complessità

- Ad ogni nuova query l'estremo sinistro viene spostato al più $\sqrt{n}$ volte.
- Per ogni blocco di query, l'estremo destro viene spostato al più n volte.

La complessità totale è $\mathcal{O}(n\sqrt{n})$.

Algoritmo di Mo

Una query $[l_1, r_1)$ viene eseguita prima di un'altra query $[l_2, r_2)$ se:

- $\left\lfloor \frac{l_1}{\sqrt{n}} \right\rfloor < \left\lfloor \frac{l_2}{\sqrt{n}} \right\rfloor$ oppure,
- $\left\lfloor \frac{l_1}{\sqrt{n}} \right\rfloor = \left\lfloor \frac{l_2}{\sqrt{n}} \right\rfloor$ e $r_1 < r_2$.

```
1 bool compare(const Query& a, const Query& b) {
2     if (a.l / BLOCK != b.l / BLOCK) {
3         return a.l < b.l;
4     } else {
5         return a.r < b.r;
6     }
7 }
```

Algoritmo di Mo

Possiamo fare ancora di meglio?

Possiamo ordinare le query nei blocchi pari in ordine crescente e i blocchi dispari in ordine decrescente così all'inizio di ogni blocco l'estremo destro non deve tornare all'inizio dell'array.

```
1 bool compare(const Query& a, const Query& b) {
2     if (a.l / BLOCK != b.l / BLOCK) {
3         return a.l < b.l;
4     } else if (a.l / BLOCK % 2 == 0) {
5         return a.r < b.r;
6     } else {
7         return a.r > b.r;
8     }
9 }
```

Frequenza dei valori

Problema

Sia dato un array di n elementi e q query, in ciascuna query bisogna calcolare il numero di occorrenze del valore x tra l e r.

Condizioni

- ✓ Query offline
- ✓ Array statico
- ✓ Intervalli facilmente estendibili

Frequenza dei valori

Soluzione

Possiamo salvare la frequenza degli elementi in un array, ogni volta che estendiamo o contraiamo un elemento, aumentiamo o diminuiamo la frequenza dei valori. Per determinare la frequenza basta cercare il relativo valore nell'array.

```
1  int arr[MAX_N];
2  int freq[MAX_VAL];
3
4  void add(int pos) {
5      freq[arr[pos]]++;
6  }
7
8  void remove(int pos) {
9      freq[arr[pos]]--;
10 }
11
12 int get_answer(int val) {
13     return freq[val];
14 }
```

Algoritmo di Mo sugli alberi

Si può usare l'algoritmo di Mo sugli alberi?

Possiamo adattare l'algoritmo per eseguire query in percorsi in un albero.

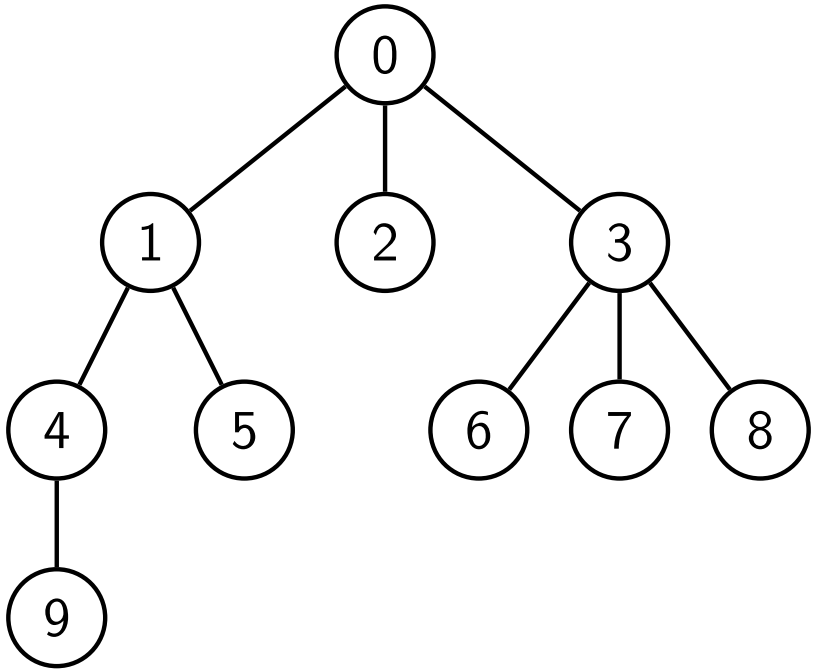
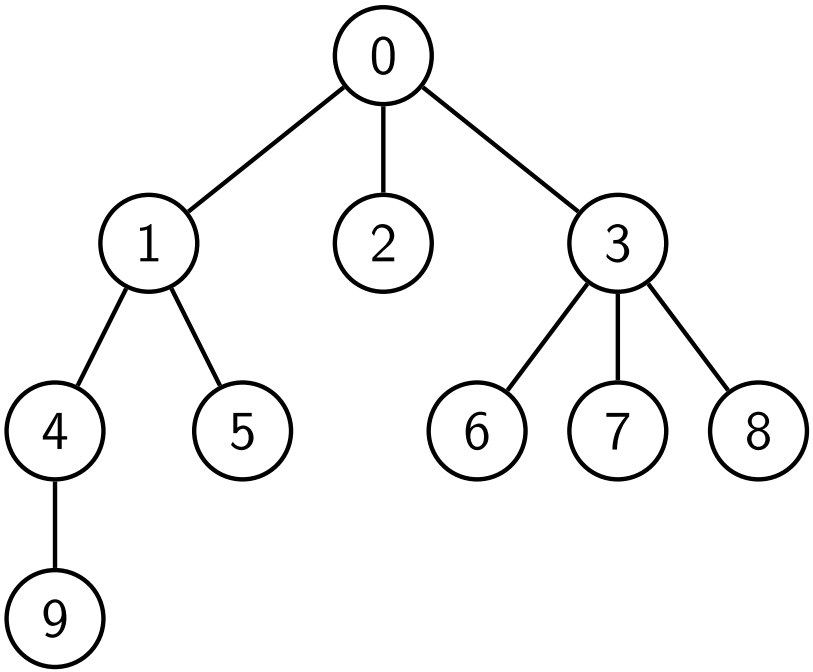


Figura: un albero

Algoritmo di Mo sugli alberi

Possiamo convertitre l'albero in un array mediante una DFS:

- aggiungo il nodo in fondo all'array;
- visito i figli;
- aggiungo nuovamente il nodo in fondo all'array.

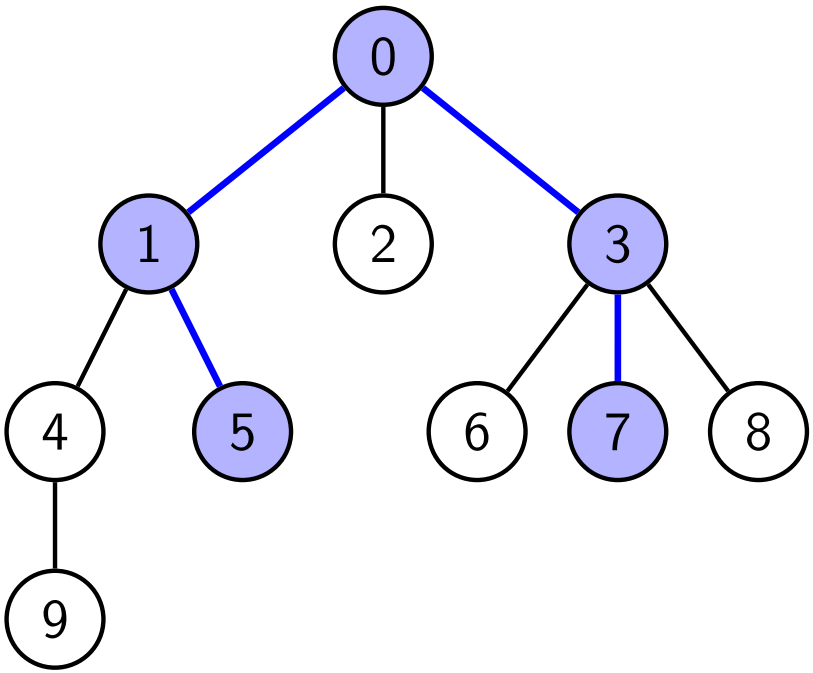


0	1	4	9	9	4	5	5	1	2	2
3	6	6	7	7	8	8	3	0		

Figura: l'array prodotto dalla DFS

Algoritmo di Mo sugli alberi

- Per fare una query su un percorso basta fare la query tra la seconda occorrenza del primo nodo e la prima occorrenza del secondo nodo nell'array.
- Vengono contati anche nodi che non fanno necessariamente parte del percorso.
- È sufficiente eliminare i nodi che appaiono due volte nell'intervallo.



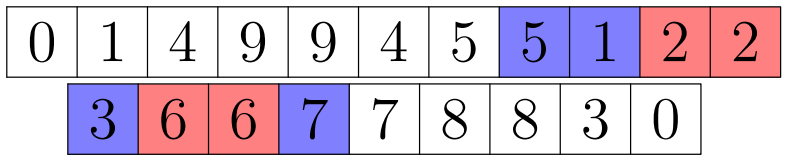
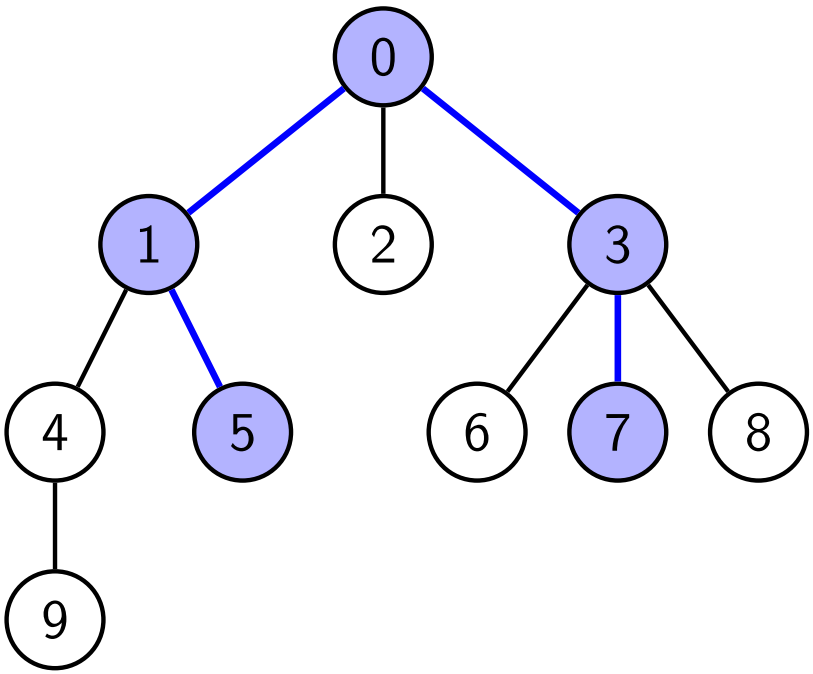


Figura: query tra 5 e 7

Algoritmo di Mo sugli alberi

Se nessuno dei due nodi è un antenato dell'altro, bisogna però considerare separatamente il LCA dei due nodi, che altrimenti sarebbe escluso dall'intervallo.



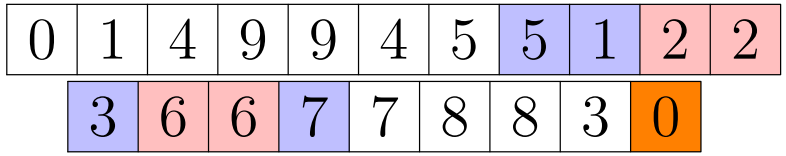


Figura: query tra 5 e 7

Algoritmo di Mo sugli alberi

Per eliminare i doppioni in modo efficiente basta memorizzare in un array globale se i valori sono già stati aggiungi nell'intervallo corrente.
Quando aggiungiamo o rimuoviamo un nuovo elemento, controlliamo se è già presente nell'intervallo corrente:

- se è presente, lo rimuoviamo;
- se non è presente, lo aggiungiamo.

```
1  int cur_l = 0, cur_r = 0;
2  for (Query q : queries) {
3      while (cur_l > q.l) toggle(--cur_l);
4      while (cur_r < q.r) toggle(cur_r++);
5      while (cur_l < q.l) toggle(cur_l++);
6      while (cur_r > q.r) toggle(--cur_r);
7      toggle(q.lca);
8      answers[q.idx] = get_answer();
9      toggle(q.lca);
10 }
```

```
1  bool inside[MAX_N];
2
3  void toggle(int pos) {
4      if (inside[pos]) {
5          inside[pos] = false;
6          remove(pos);
7      } else {
8          inside[pos] = true;
9          add(pos);
10     }
11 }
```

Knapsack

Problema

Dato un array di n elementi tale che la somma degli elementi sia uguale a w, determinare se esiste un subset di elementi tale la loro somma sia x.

Osservazione

Possiamo eseguire un knapsack con complessità $\mathcal{O}(nw)$.
Si può fare di meglio?

Knapsack

Osservazione

$$1 + 2 + \cdots + k \approx \frac{k^2}{2}$$

Quindi ci sono al più $\mathcal{O}(\sqrt{w})$ elementi distinti.

Soluzione

Possiamo modificare il knapsack tradizionale processando i pesi uguali contemporaneamente.
Sia $\texttt{cnt}[i]$ il numero di occorrenze di i, possiamo scrivere $\texttt{cnt}[i]$ come:

$$1 + 2 + 4 + 8 + \cdots + 2^t + s = \texttt{cnt}[i]$$

dove $s \leq 2^{t+1}$. In questo modo possiamo rappresentare $\texttt{cnt}[i]$ come somma di al massimo $\log n$ valori distini, inoltre ogni numero tra 1 e $\texttt{cnt}[i]$ può essere costruito come somma di tale valori.

Knapsack

Soluzione

Sostituiamo quindi le occorrenze di i con $i,\ 2 \cdot i,\ 4 \cdot i, \ldots, 2^t \cdot i,\ s \cdot i$. Adesso abbiamo ridotto i valori da n a $\sqrt{w}$, possiamo utilizzare il classico knapsack e la complessità finale è $\mathcal{O}(w\sqrt{w})$.

```
1  bitset<MAX_W> knapsack;
2
3  knapsack[0] = 1;
4  for (int x = 1; x <= W; x++) {
5      for (int k = 0; (1 << k) <= cnt[x]; k++) {
6          knapsack |= knapsack << (1 << k);
7          cnt[x] -= (1 << k);
8      }
9      knapsack |= knapsack << cnt[x];
10 }
```

Consigli

Aggiustare le costanti

- Spesso $\sqrt{n}$ non è sempre il valore ottimale da usare, tale valore può dipendere dal rapporto tra il costo di query e update e da altri fattori.
- Conviene utilizzare un valore statico anziché calcolare $\sqrt{n}$ per ogni testcase.
- Fare diversi tentativi e aggiustare le costanti finché la soluzione non sta nei tempi.