Tecniche square-root

Alessandro Bortolin

Volterra, 16 novembre 2022

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. ☺

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

- Non sempre è facile far stare una soluzione $\mathcal{O}(n\sqrt{n})$ nel tempo limite. ☹
- Alcune di queste tecniche sono **offline**. ☹

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

Tecniche

Quali sono le tecniche più comuni?

1. Square-root decomposition
2. Case processing
3. Batch processing
4. Algoritmo di Mo
5. Knapsack

Square-root decomposition

Square-root decomposition

La **square-root decomposition** è una tecnica che permette di creare 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 value

Problema

Sia dato un array di n elementi, devi eseguire q operazioni:

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

Possiamo risolverlo mediante un segment tree con lazy propagation, vediamo un'altra tecnica!

Minimum value

Soluzione senza modifiche

Dividiamo l'array in blocchi di $\sqrt{n}$ elementi ciascuno, per ciascun blocco calcoliamo il suo valore minimo.

Per rispondere a una richiesta:

- iteriamo su tutti i blocchi **totalmente inclusi** nell'intervallo e calcoliamo il valore minimo;
- iteriamo sui restanti valori singolarmente.

Per ogni richiesta iteriamo al massimo su $\mathcal{O}(\frac{n}{\sqrt{n}} + \sqrt{n}) = \mathcal{O}(\sqrt{n})$ valori.

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

Figura: esempio di richiesta con $l = 3$ e $r = 10$.

Minimum value

Soluzione con modifiche

Per ogni blocco memorizziamo due valori:

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

Per eseguire una modifica:

- 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 modifica con $l = 6$, $r = 13$ e $x = 2$.

Minimum value

Complessità

Per ogni modifica:

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

La complessità per ogni modifica è, come per le richieste, $\mathcal{O}(\frac{n}{\sqrt{n}} + \sqrt{n}) = \mathcal{O}(\sqrt{n})$.

Vasi 1

Problema

Ci sono n vasi disposti in linea, ciascun vaso ha un valore. Inizialmente i valori sono $0, 1, \ldots, n-1$. Devi eseguire q operazioni, ciascuna operazione può essere:

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

Soluzione

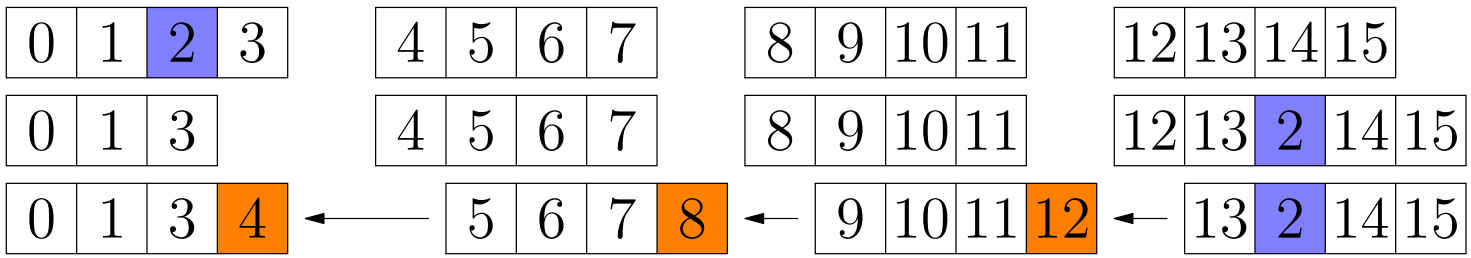
Dividiamo l'array in blocchi di $\sqrt{n}$ vasi e rappresentiamo ciascun blocco mediante una **deque**. Tutti i blocchi hanno la stessa dimensione, quindi sappiamo che il k-esimo elemento si troverà nel blocco $\left\lfloor \frac{k}{\sqrt{n}} \right\rfloor$ in posizione $k \mod \sqrt{n}$

Vasi 1

Soluzione

Per spostare un vaso dalla posizione x alla posizione y:

- rimuoviamo il vaso dal blocco originale;
- aggiungiamo il vaso nel nuovo blocco;
- per ogni blocco interamente compreso tra x e y:
  - se $x < y$, rimuoviamo il primo vaso del blocco e lo aggiungiamo in fondo al blocco precedente;
  - se $x > y$, rimuoviamo l'ultimo vaso del blocco e lo aggiungiamo all'inizio del blocco successivo.



Vasi 1

Complessità

- Rimuovere il vaso dal blocco originale ha complessità $\mathcal{O}(\sqrt{n})$.
- Aggiungere il vaso nel nuovo blocco ha complessità $\mathcal{O}(\sqrt{n})$.
- Aggiungere o rimuovere un vaso dalle estremità della deque può essere fatto in $\mathcal{O}(1)$ mediante le funzione della deque `pop_front`, `pop_back`, `push_front` e `push_back`.

Aggiungiamo o eliminiamo i vasi dalle estremità di un blocco al massimo $\sqrt{n}$ volte, la complessità per ogni modifica è quindi $\mathcal{O}(\sqrt{n})$.

Case processing

Case processing

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

Il primo algoritmo viene usato quando i valori sono “grandi” mentre il secondo viene usato quando i valori sono “piccoli“.

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 dello stesso colore.

Soluzione 1

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

Soluzione 2

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

Griglia colorata 1

Qual è la complessità dei due algoritmi?

Soluzione 1

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

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 $\mathcal{O}(n^3)$.

Remainder Problem (Codeforces 1207F)

Problema

Sia dato un array A di n interi, devi eseguire q operazioni:

- aggiungere x a A_y;
- stampare la somma di A_i per ogni i tale che $i \equiv y \mod x$.

Soluzione 1

Processiamo *normalmente* le operazioni, come descritto dal problema.
Aggiornare l'array ha complessità $\mathcal{O}(1)$, mentre calcolare la somma ha complessità $\mathcal{O}(n)$.

Soluzione 2

Precalcoliamo la risposta per ogni possibile coppia (x, y).
A ogni modifica dobbiamo aggiornare la risposta di n coppie (x, y), possiamo rispondere alle richieste in $\mathcal{O}(1)$.

Remainder Problem (Codeforces 1207F)

Nuova soluzione

Se $x > \sqrt{n}$ utilizziamo la soluzione 1, altrimenti utilizziamo la soluzione 2.

Precalcoliamo la risposta per le coppie (x, y) soltanto quando $x \leq \sqrt{n}$.
A ogni modifica, se $x \leq \sqrt{n}$ dobbiamo aggiornare la risposta di $\sqrt{n}$ coppie (x, y).

Per ogni richiesta:

- se $x \leq \sqrt{n}$ allora abbiamo già precalcolato il risultato;
- se $x > \sqrt{n}$ allora esiteranno al massimo $\frac{n}{\sqrt{n}}$ valori i tale che $i \equiv y \mod x$, possiamo calcolare la risposta in $\mathcal{O}(\sqrt{n})$.

Batch processing

Batch processing

Il *batch processing* è una tecnica che permette di eseguire una serie di operazioni di richieste e modifiche su una struttura dati in modo efficiente.

La tecnica consiste nel suddividere le operazioni in blocchi:

- alla fine di ogni blocco ricostruiamo la struttura dati applicando tutte le modifiche del blocco;
- per rispondere a una richiesta, controlliamo nella struttura dati, tenendo però conto delle modifiche che non sono state ancora applicate.

Griglia colorata 2

Problema

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

Soluzione 1

A 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, calcoliamo la distanza minima di ogni casella dalla casella nera più vicina;
- man mano che processiamo le caselle di un gruppo, le aggiungiamo in un buffer;
- a ogni turno, prima di aggiungere la casella al buffer, calcoliamo la sua distanza da ogni casella già presente nel buffer e confrontiamo il minimo 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 (semplificato)

Problema

Sia dato un albero di n nodi con radice 1, devi eseguire q operazioni:

- cambiare il padre di una foglia;
- calcolare la somma delle distanze dal nodo x di ogni discendente di x.

Piccioni in migrazione (semplificato)

Soluzione senza modifiche

Precalcoliamo alcuni valori:

- `deep`$[i]$: distanza di i dalla radice;
- `size`$[i]$: dimensione del sottoalbero di i;
- `sum`$[i]$: somma dei valori di `deep`$[j]$ per ogni j nel sottoalbero di i.

$$\sum_{i \in \text{subtree}(x)} \text{dist}(i, x) = \texttt{sum}[x] - \texttt{size}[x] \cdot \texttt{deep}[x]$$

Possiamo precalcolare questi valori mediante una DFS con complessità $\mathcal{O}(n)$ e rispondere a ogni richiesta con complessità $\mathcal{O}(1)$.

Piccioni in migrazione (semplificato)

Soluzione con modifiche

Dividiamo le operazioni in blocchi di $\sqrt{n}$, all'inizio di ogni blocco precalcoliamo i valori `deep`, `size` e `sum`. A ogni operazione:

- per spostare un nodo lo aggiungiamo semplicemente all'interno di un buffer;
- per rispondere a una richiesta su x, se x si trova all'interno del buffer, calcoliamo la risposta con una DFS;
- se x non è all'interno del buffer, calcoliamo la risposta utilizzando i valori precalcolati, poi per ogni nodo i nel buffer:
  - se la precedente posizione di i apparteneva al sottoalbero di x, sottraiamo $\texttt{deep}[i] - \texttt{deep}[x]$;
  - se la nuova posizione di i appartiene al sottoalbero di x, aggiungiamo $\texttt{deep}[i] - \texttt{deep}[x]$.

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

Piccioni in migrazione (l'originale)

Soluzione completa

La soluzione al problema semplificato può essere adattata al problema originale e permette di ottenere una complessità di $\mathcal{O}(q\sqrt{n \log n})$.

Vasi 1 (rivisitato)

Problema

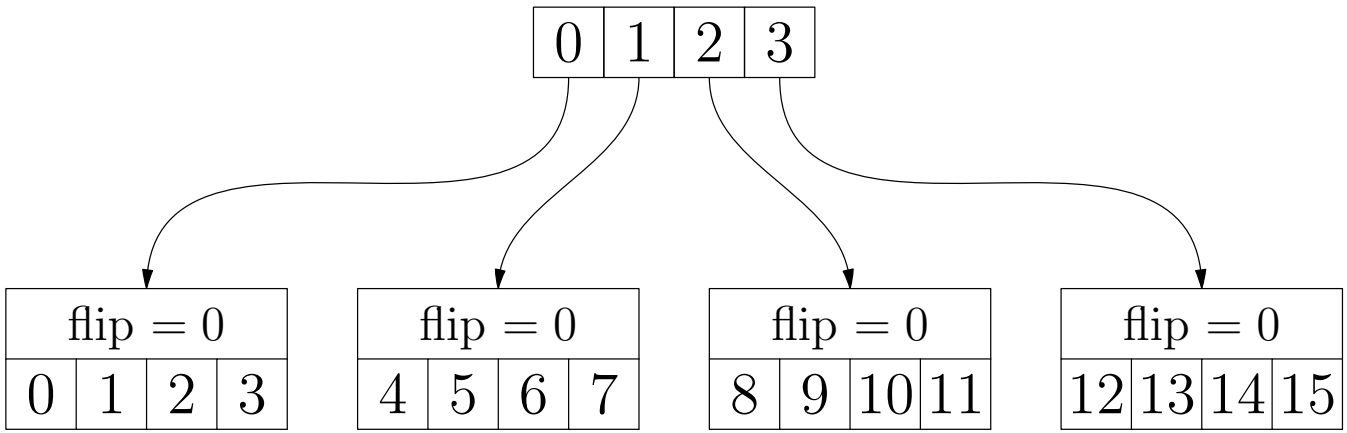
Ci sono n vasi disposti in linea, ciascun vaso ha un valore. Inizialmente i valori sono $0, 1, \ldots, n-1$. Devi eseguire q operazioni, ciascuna operazioni può essere:

- stampare il valore del k-esimo vaso;
- spostare il vaso in posizione x alla posizione y mantenendo l'ordine degli altri vasi;
- invertiamo l'ordine dei vasi nell'intervallo $[l, r]$.

Vasi 1 (rivisitato)

Soluzione

Dividiamo l'array in $\sqrt{n}$ blocchi, inizialmente di $\sqrt{n}$ elementi. Inoltre, memorizziamo se ciascun blocco è stato invertito con una flag `flip`.



Vasi 1 (rivisitato)

Soluzione

Durante le modifiche, i blocchi possono aumentare o diminuire di dimensione.

Per determinare la posizione di un vaso, scorriamo i blocchi e sommiamo la loro dimensione fino a trovare il blocco appartenente al vaso che cerchiamo.

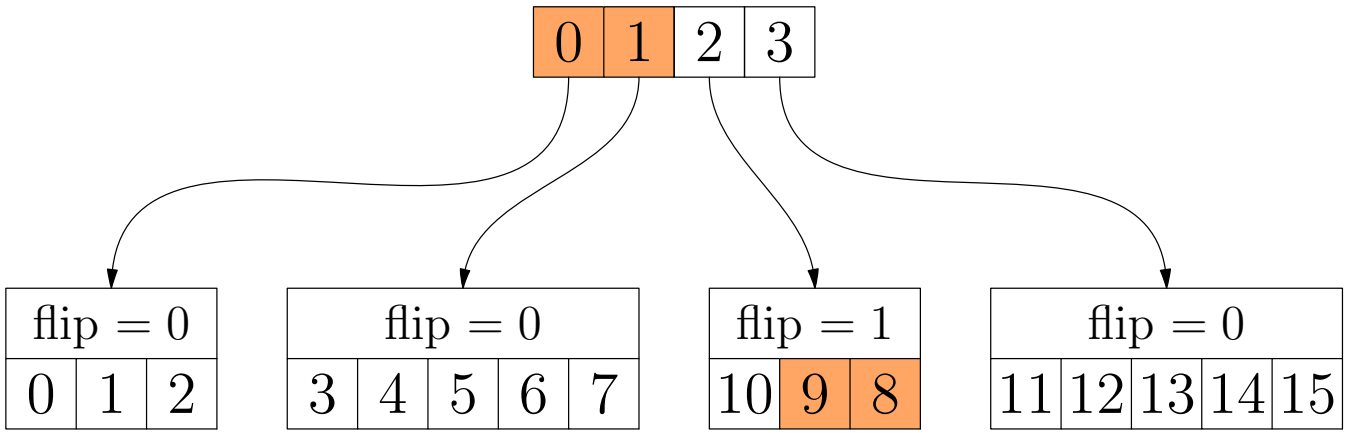


Figura: determinare il vaso in posizione 9.

Vasi 1 (rivisitato)

Soluzione

Per spostare un vaso:

- determiniamo i blocchi a cui appartiene il vaso prima e dopo lo spostamento;
- rimuoviamo il vaso dal blocco originale e lo aggiungiamo al nuovo blocco.

flip $= 0$			
0	1	2	3

flip $= 0$			
4	5	6	7

flip $= 0$			
8	9	10	11

flip $= 0$			
12	13	14	15

flip $= 0$		
0	1	3

flip $= 0$			
4	5	6	7

flip $= 0$			
8	9	10	11

flip $= 0$				
12	13	2	14	15

Figura: il vaso 2 è stato spostato in posizione 13.

Vasi 1 (rivisitato)

Soluzione

Per invertire l'ordine dei vasi nell'intervallo $[l, r]$:

- determiniamo i blocchi a cui appartengono i vasi l e r;
- dividiamo questi blocchi in due, in modo che il vaso l sia il primo del suo blocco e il vaso r sia l'ultimo del suo blocco;
- invertiamo l'ordine dei blocchi dell'intervallo e modifichiamo il valore delle flag `flip`.

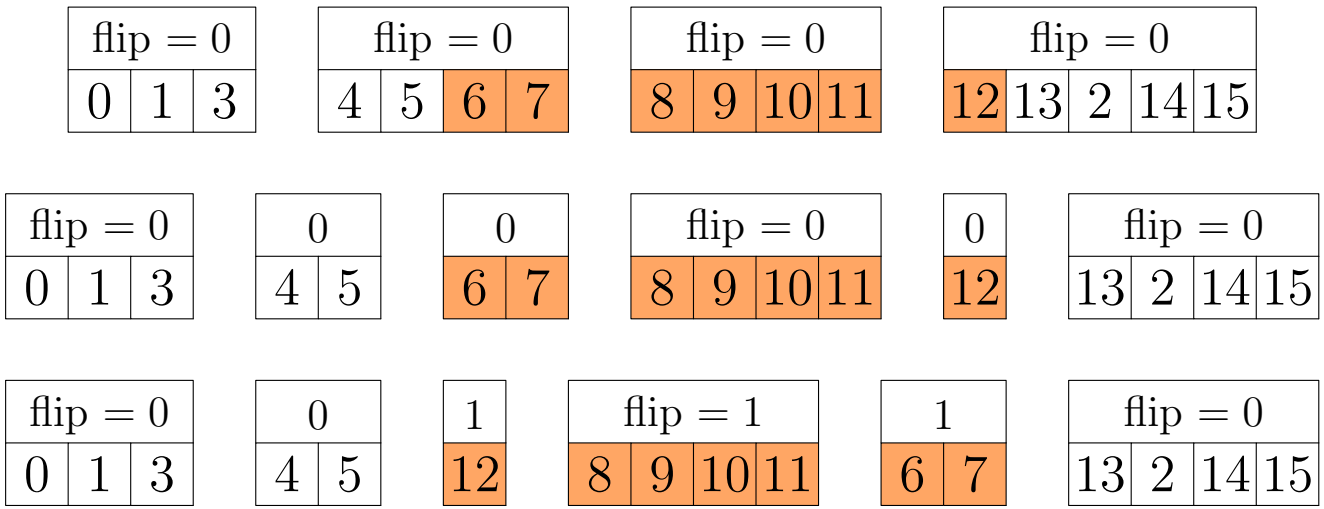


Figura: una query sull'intervallo $[5, 11]$.

Vasi 1 (rivisitato)

Soluzione

Infine, dopo ogni $\sqrt{n}$ operazioni, ricostruiamo tutti gli array.

In questo modo, in ogni istante ci saranno al massimo $\mathcal{O}(\sqrt{n})$ blocchi e ciascun blocco avrà al massimo $\mathcal{O}(\sqrt{n})$ vasi.

Algoritmo di Mo

Algoritmo di Mo

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

Il tipo di query deve essere **facilmente estendibile**, ovvero se conosco la risposta per l'intervallo $[l, r]$, posso calcolare velocemente la risposta per l'intervallo $[l, r + 1]$ e $[l, r - 1]$.

- somma ✓
- massimo comune divisore ✗
- massimo/minimo ✓
- frequenza dei valori ✓
- moda ✓

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, a ogni query l'intervallo corrente viene esteso o accorciato 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  }
```

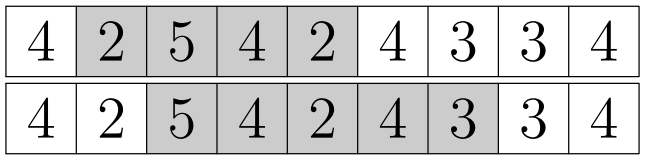


Figura: esempio in cui l'intervallo viene spostato.

Algoritmo di Mo

Complessità

A ogni query l'intervallo viene esteso o accorciato 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.

Algoritmo di Mo

Esempio

Le query sono: $(4,6)$, $(2,4)$, $(0,1)$, $(1,7)$, $(6,6)$, $(3,7)$.

$l = 0, r = 1$

$l = 2, r = 4$

$l = 1, r = 7$

$l = 4, r = 6$

$l = 3, r = 7$

$l = 6, r = 6$

0	1	2	3	4	5	6	7	8

Algoritmo di Mo

Complessità

- A 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 $\mathcal{O}\left((n+q)\sqrt{n}\right)$.

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 nei blocchi dispari in ordine decrescente. In questo modo, 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  }
```

Valori distinti

Problema

Sia dato un array di n elementi, devi eseguire q operazioni:

- calcolare il numero di valori distinti x nell'intervallo da l a r.

Condizioni

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

Valori distinti

Soluzione

Salviamo la frequenza degli elementi in un array, ogni volta che estendiamo o accorciamo l'intervallo, aumentiamo o diminuiamo la frequenza dei valori. Teniamo inoltre un variabile per contare il numero di valori distinti.

```
1   int arr[MAX_N];
2   int freq[MAX_VAL];
3   int uniq = 0;
4   void add(int pos) {
5       if (freq[arr[pos]] == 0) uniq++;
6       freq[arr[pos]]++;
7   }
8   void remove(int pos) {
9       freq[arr[pos]]--;
10      if (freq[arr[pos]] == 0) uniq--;
11  }
12  int get_answer() {
13      return uniq;
14  }
```

Algoritmo di Mo sugli alberi

Si può usare l'algoritmo di Mo sugli alberi?

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

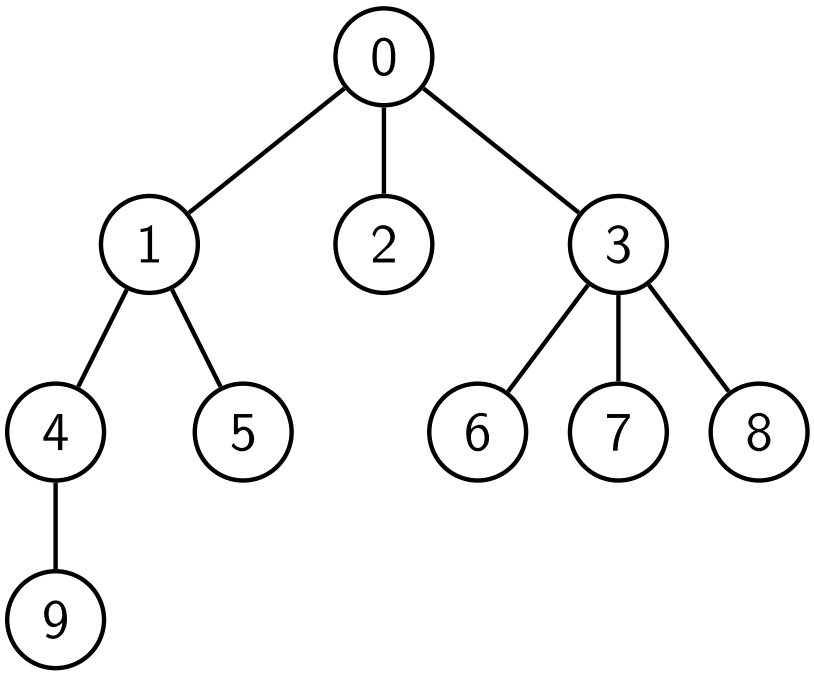
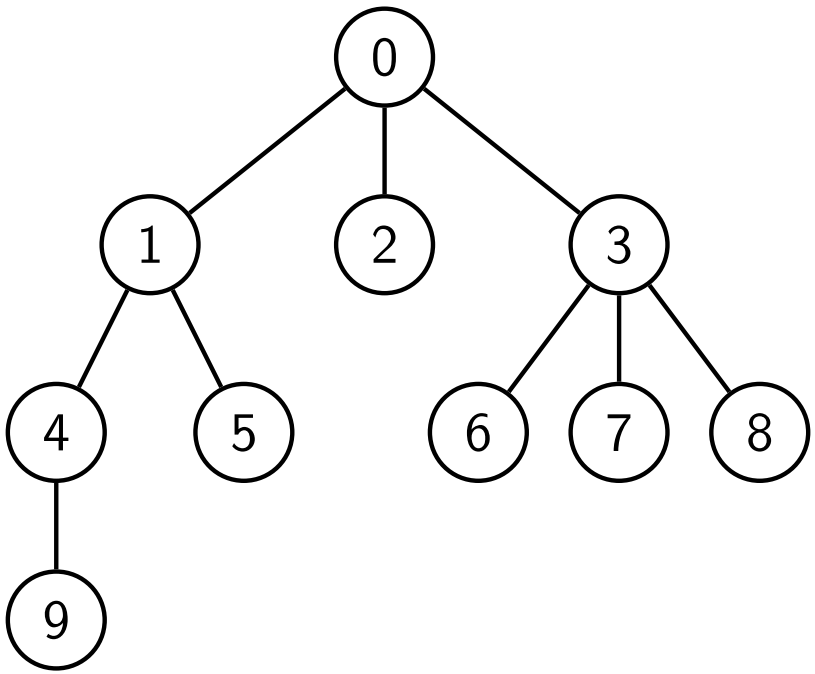


Figura: un albero.

Algoritmo di Mo sugli alberi

Possiamo convertire 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.

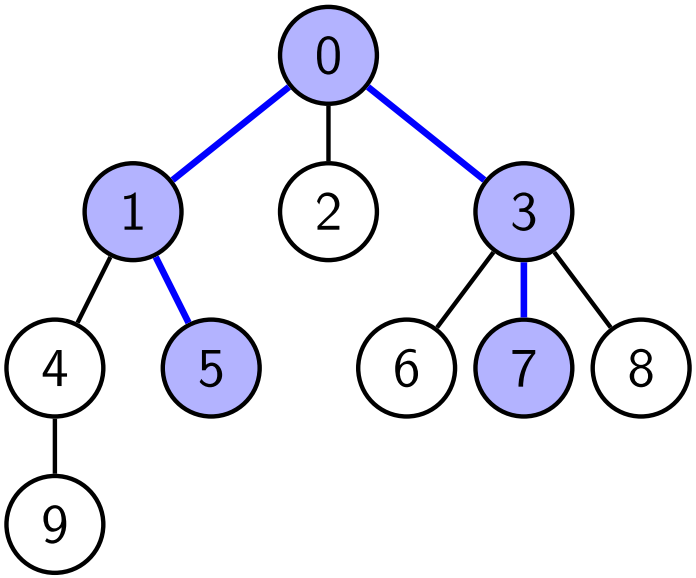


0	1	4	9	9	4	5	5	1	2	2
	3	6	6	7	7	8	8	3	0	

Figura: l'array prodotto dalla DFS.

Algoritmo di Mo sugli alberi

- La query di un percorso si traduce in una query tra la **seconda occorrenza** del primo nodo e la **prima occorrenza** del secondo nodo nell'array.
- In questo modo, vengono però contati anche nodi che non fanno necessariamente parte del percorso.
- È sufficiente eliminare i nodi che appaiono due volte nell'intervallo.
- Se nessuno dei due nodi è un antenato dell'altro, bisogna considerare separatamente il LCA dei due nodi.



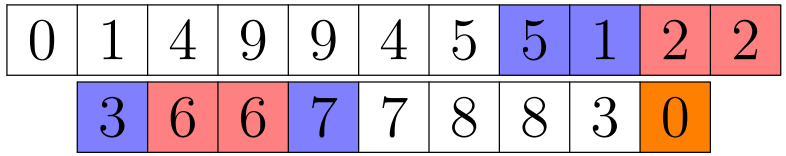


Figura: una query tra 5 e 7.

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       if (q.lca != q.l && q.lca != q.r)
8           toggle(q.lca);
9       answers[q.idx] = get_answer();
10      if (q.lca != q.l && q.lca != q.r)
11          toggle(q.lca);
12  }
```

```
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

Knapsack

Problema

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

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

Prima osservazione

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

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

Knapsack

Seconda osservazione

Dato un certo numero x, possiamo rappresentare x come:

$$1 + 2 + 4 + 8 + \cdots + 2^t + s = x$$

dove $s < 2^{t+1}$. Definiamo poi $I = \{1, 2, 4, 8, \ldots, 2^t, s\}$.
Notiamo che possiamo rappresentare tutti e soli i numeri tra 1 e x come somma di elementi distinti di I.
Ad esempio sia $x = 42$, otteniamo che $2^t = 16$, $s = 11$ e $I = \{1, 2, 4, 8, 16, 11\}$.

- $11 = 1 + 2 \quad + 8$
- $22 = \quad 2 + 4 \quad + 16$
- $32 = 1 \quad + 4 \quad + 16 + 11$

Notiamo poi che $|I| = \mathcal{O}(\log x)$

Knapsack

Soluzione

Possiamo modificare l'algoritmo tradizionale del knapsack processando gli oggetti di peso uguale contemporaneamente.

Sostituiamo tutti gli oggetti di peso i con nuovi oggetti di peso i, $2i$, $4i$, $\ldots$, $2^t \cdot i$, $s \cdot i$. Facendo così, riduciamo il numero di oggetti a $\mathcal{O}(\sqrt{w})$, utilizzando il classico knapsack, la complessità diventa $\mathcal{O}(w\sqrt{w})$.

$$\sum_{x=1}^{w} \log(\texttt{count}[x] + 1) \leq \sum_{r=0}^{\infty} \sqrt{\frac{w}{2^r}} = \mathcal{O}(\sqrt{w})$$

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

Gastronomic Event (SWERC 2022 K)

Problema

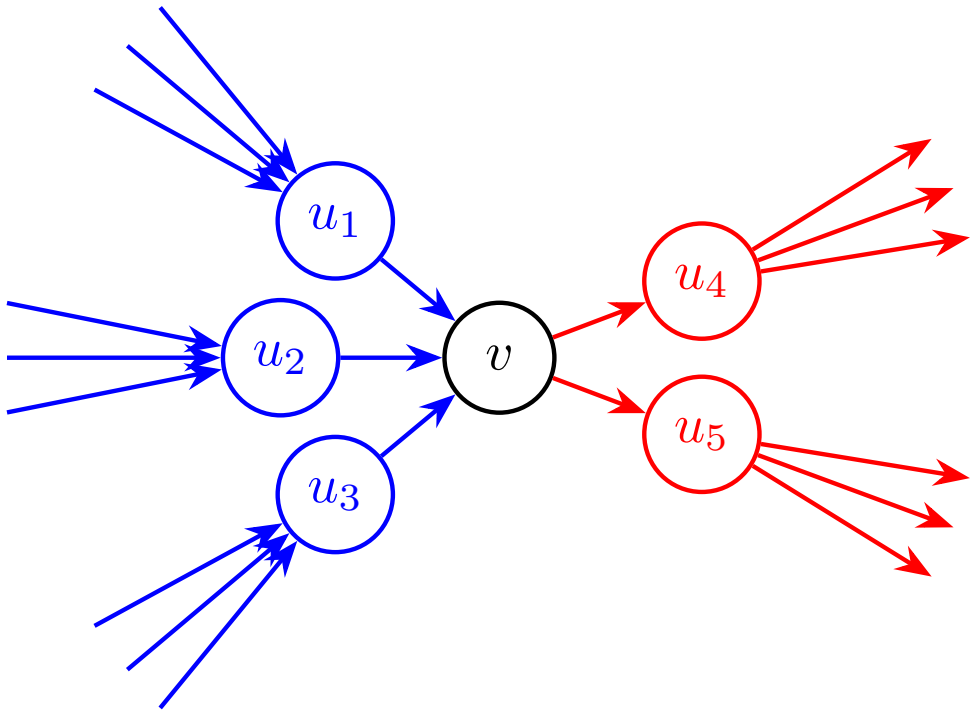
Dato un albero, devi orientare gli archi in modo da massimizzare il numero di percorsi semplici distinti.

Gastronomic Event (SWERC 2022 K)

Osservazione (non banale)

In una soluzione ottimale, non è mai presente un percorso in cui gli archi cambiano di direzione.

Di conseguenza, esiste al massimo un nodo *incrocio*, ovvero un nodo che ha più di un arco entrante e più di un arco uscente.

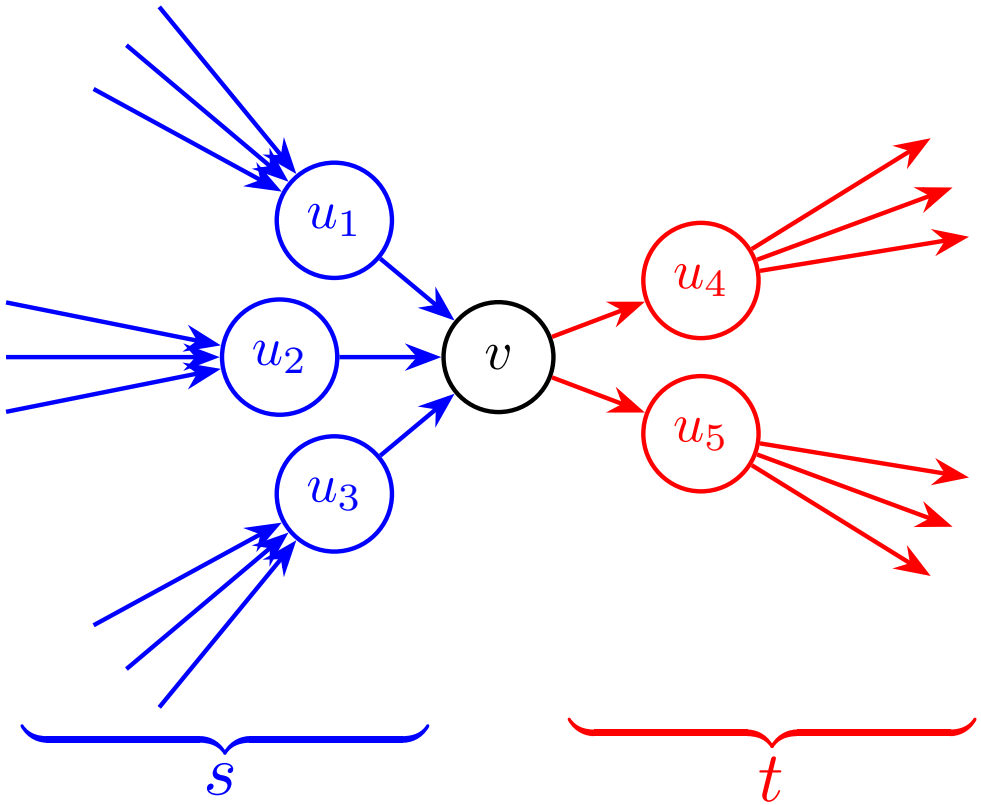


Gastronomic Event (SWERC 2022 K)

Soluzione

Fissato il nodo incrocio v, il numero di percorsi distinti è pari a:

$$n + \sum_{z=1}^{n} \text{dist}(z, v) + s \cdot t$$



Gastronomic Event (SWERC 2022 K)

Soluzione

Dobbiamo massimizzare:

$$n + \sum_{z=1}^{n} \text{dist}(z, v) + s \cdot t$$

$s \cdot t$ è massimizzato quando $s \approx t$:

- se v non è un centroide, allora è sempre conveniente mettere il sottoalbero più grande da un lato e tutti gli altri dall'altro lato;
- se v è un centroide eseguiamo un **knapsack** per provare tutte le possibilità:
  - la somma dei sottoalberi è $\mathcal{O}(n)$, quindi possiamo eseguire il knapsack con complessità $\mathcal{O}(n\sqrt{n})$;
  - ci sono al massimo due centroidi quindi eseguiamo il knapsack al più 2 volte.

Queste osservazioni ci permettono di ottenere una soluzione $\mathcal{O}(n\sqrt{n})$.

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 test case.
- Usare una potenza di 2 può essere, a volte, più efficiente.
- Fare diversi tentativi e aggiustare le costanti finché la soluzione non sta nei tempi.
- Un buon compromesso è:

```
constexpr int BLOCK = 512;
```

Esercizi

Problemi delle slide

- Vasi 1:
  https://training.olinfo.it/#/task/vasi/statement
- Remainder Problem:
  https://codeforces.com/problemset/problem/1207/F
- Piccioni in migrazione:
  https://training.olinfo.it/#/task/preoii_piccioni/statement
- Valori distinti:
  https://cses.fi/problemset/task/1734
- Gastronomic Event:
  https://codeforces.com/contest/1662/problem/G

Esercizi

Problemi aggiuntivi

- Serega and Fun:
  `https://codeforces.com/contest/455/problem/D`
- Propagation:
  `https://atcoder.jp/contests/abc219/tasks/abc219_g`
- Little Elephant and Array:
  `https://codeforces.com/contest/220/problem/B`
- Word Combinations:
  `https://cses.fi/problemset/task/1731`

Bibliografia


Animesh Fatehpuria.
Mo's Algorithm on Trees.
https://codeforces.com/blog/entry/43230, 2018.


Antti Laaksonen.
Guide to Competitive Programming.
Springer, 2017.


Ashley Khoo.
Knapsack, Subset Sum and the (max,+) Convolution.
https://codeforces.com/blog/entry/98663, 2021.


Benjamin Qi, Neo Wang.
Square Root Decomposition.
https://usaco.guide/plat/sqrt, 2021.


Kamil Debowski.
Square Root Techniques.
https://codeforces.com/blog/entry/96713, 2021.