HLD e Centroid Decomposition

Lorenzo Ferrari, Francesco Lugli

Online, 23 febbraio 2024

Small to Large

Problema motivazionale

Problema (Distinct Colors)

È dato un albero con $N \leq 200000$ *nodi radicato in* 1*, ogni nodo ha un colore. Per ogni nodo, calcolare il numero di colori distinti nel suo sottoalbero.*

https://cses.fi/problemset/task/1139

Problema motivazionale

Problema (Distinct Colors)

È dato un albero con $N \leq 200000$ nodi radicato in 1, ogni nodo ha un colore. Per ogni nodo, calcolare il numero di colori distinti nel suo sottoalbero.

https://cses.fi/problemset/task/1139

Soluzione naive

- da ogni nodo v, parte una DFS

Problema motivazionale

Problema (Distinct Colors)

È dato un albero con $N \leq 200000$ nodi radicato in 1, ogni nodo ha un colore. Per ogni nodo, calcolare il numero di colori distinti nel suo sottoalbero.

https://cses.fi/problemset/task/1139

Soluzione naive

- da ogni nodo v, parte una DFS
- i colori dei nodi visitati vengono inseriti in un `std::set`

Problema motivazionale

Problema (Distinct Colors)

È dato un albero con $N \leq 200000$ *nodi radicato in* 1*, ogni nodo ha un colore. Per ogni nodo, calcolare il numero di colori distinti nel suo sottoalbero.*

https://cses.fi/problemset/task/1139

Soluzione naive

- da ogni nodo v, parte una DFS
- i colori dei nodi visitati vengono inseriti in un `std::set`
- la risposta per v è `set.size()`

Problema motivazionale

Problema (Distinct Colors)

È dato un albero con $N \leq 200000$ nodi radicato in 1, ogni nodo ha un colore. Per ogni nodo, calcolare il numero di colori distinti nel suo sottoalbero.

https://cses.fi/problemset/task/1139

Soluzione naive

- da ogni nodo v, parte una DFS
- i colori dei nodi visitati vengono inseriti in un `std::set`
- la risposta per v è `set.size()`

La complessità è $\mathcal{O}(N^2)$ o $\mathcal{O}(N^2 \log N)$: troppo lento.

Scendere a $\mathcal{O}(N \log^2 N)$

Ottimizzazione

- il set di un figlio v è sempre contenuto nel set del padre u
  - per costruire il set di u si può partire dal set di un figlio v e inserire uno a uno gli elementi dei set dei figli $\neq v$
- scegliendo come v ogni volta il figlio col set più grande, la complessità scende a $\mathcal{O}(N \log^2 N)$

Scendere a $\mathcal{O}(N \log^2 N)$

Ottimizzazione

- il set di un figlio v è sempre contenuto nel set del padre u
  - per costruire il set di u si può partire dal set di un figlio v e inserire uno a uno gli elementi dei set dei figli $\neq v$
- scegliendo come v ogni volta il figlio col set più grande, la complessità scende a $\mathcal{O}(N \log^2 N)$

Perché dovrebbe funzionare?

Contiamo il numero totale di operazioni `set.insert(...)`, per semplicità assumiamo di usare `std::multiset`

Scendere a $\mathcal{O}(N \log^2 N)$

Ottimizzazione

- il set di un figlio v è sempre contenuto nel set del padre u
  - per costruire il set di u si può partire dal set di un figlio v e inserire uno a uno gli elementi dei set dei figli $\neq v$
- scegliendo come v ogni volta il figlio col set più grande, la complessità scende a $\mathcal{O}(N \log^2 N)$

Perché dovrebbe funzionare?

Contiamo il numero totale di operazioni `set.insert(...)`, per semplicità assumiamo di usare `std::multiset`

Idea Ogni volta che un elemento è inserito nel set di un fratello, la dimensione del set di arrivo sarà *almeno il doppio* della dimensione del set di partenza.

Scendere a $\mathcal{O}(N \log^2 N)$

Ottimizzazione

- il set di un figlio v è sempre contenuto nel set del padre u
  - per costruire il set di u si può partire dal set di un figlio v e inserire uno a uno gli elementi dei set dei figli $\neq v$
- scegliendo come v ogni volta il figlio col set più grande, la complessità scende a $\mathcal{O}(N \log^2 N)$

Perché dovrebbe funzionare?

Contiamo il numero totale di operazioni `set.insert(...)`, per semplicità assumiamo di usare `std::multiset`

Idea Ogni volta che un elemento è inserito nel set di un fratello, la dimensione del set di arrivo sarà *almeno il doppio* della dimensione del set di partenza. Ma allora ogni elemento può essere spostato *al più* $\log_2 N$ volte!

Scendere a $\mathcal{O}(N \log^2 N)$

Ottimizzazione

- il set di un figlio v è sempre contenuto nel set del padre u
  - per costruire il set di u si può partire dal set di un figlio v e inserire uno a uno gli elementi dei set dei figli $\neq v$
- scegliendo come v ogni volta il figlio col set più grande, la complessità scende a $\mathcal{O}(N \log^2 N)$

Perché dovrebbe funzionare?

Contiamo il numero totale di operazioni `set.insert(...)`, per semplicità assumiamo di usare `std::multiset`

Idea Ogni volta che un elemento è inserito nel set di un fratello, la dimensione del set di arrivo sarà *almeno il doppio* della dimensione del set di partenza. Ma allora ogni elemento può essere spostato *al più* $\log_2 N$ volte!

$\implies$ il numero totale di insert è $\leq N \log_2 N$, la complessità è dunque $\mathcal{O}(N \log^2 N)$. $\square$

Small to Large

```
set<int> dfs(int v, int par) {
    set<int> set;
    for (int u : adj[v]) {
        if (u == par) continue;
        auto tmp = dfs(u, v);
        if (tmp.size() > set.size()) {  // importantissimo
            swap(tmp, set);             // O(1) per strutture nella STL
        }
        for (int c : tmp) {
            set.insert(c);
        }
    }
    set.insert(color[v]);
    answer[v] = set.size();

    return set;                         // O(1)
}
```

Heavy-Light Decomposition

Problema motivazionale

Problema (Path Queries II)

È dato un albero con $N \le 200'000$ *nodi numerati* $1, \ldots, N$. *Ogni nodo ha un valore* v_i. *Il tuo compito è processare* $Q \le 200'000$ *delle seguenti query:*

1. *cambia il valore del nodo* s *a* x*;*
2. *trova il massimo sul percorso dal nodo* a *al nodo* b.

https://cses.fi/problemset/task/1139

Problema motivazionale

Problema (Path Queries II)

È dato un albero con $N \leq 200'000$ *nodi numerati* $1, \ldots, N$. *Ogni nodo ha un valore* v_i. *Il tuo compito è processare* $Q \leq 200'000$ *delle seguenti query:*

1. *cambia il valore del nodo* s *a* x;
2. *trova il massimo sul percorso dal nodo* a *al nodo* b.

https://cses.fi/problemset/task/1139

Soluzione naive

- $\mathcal{O}(N)$ di preprocessing
- $\mathcal{O}(1)$ per update
- $\mathcal{O}(N)$ per query

HLD permette di passare a

- $\mathcal{O}(N \log N)$ di preprocessing
- $\mathcal{O}(\log N)$ per update
- $\mathcal{O}(\log^2 N)$ per query

Idea

L'idea è costruire un'opportuna struttura dati su alcuni cammini dell'albero.

[1] in particolare F cammini, con F numero di foglie.

Idea

L'idea è costruire un'opportuna struttura dati su alcuni cammini dell'albero.

Un modo (non efficiente) è considerare tutti cammini da una foglia alla radice e costruire su ognuno di essi un **segment tree**:

- ogni cammino semplice è unione di al più due sottocammini di tali cammini
- query e update in $\mathcal{O}(\log N)$

skill Issue preprocessing e memoria quadratiche.

[1] in particolare F cammini, con F numero di foglie.

Idea

L'idea è costruire un'opportuna struttura dati su alcuni cammini dell'albero.

Un modo (non efficiente) è considerare tutti cammini da una foglia alla radice e costruire su ognuno di essi un **segment tree**:

- ogni cammino semplice è unione di al più due sottocammini di tali cammini
- query e update in $\mathcal{O}(\log N)$

skill Issue preprocessing e memoria quadratiche.

Heavy-Light Decomposition risolve suddividendo l'albero in cammini **disgiunti**[1].

[1] in particolare F cammini, con F numero di foglie.

Idea

Definizione (archi pesanti e leggeri)

Un arco si dice *pesante* (“*heavy*”) se, detti u, v i suoi estremi con u padre di v, si ha[a]

$$size(v) \geq \max_{w \in children(u)} size(w)$$

e, se $size(v) = size(w)$, $v < w$. Un arco non pesante si dice *leggero* (“*light*”).

[a] con $size(v)$ intendiamo $size(subtree(v))$

I cammini che consideriamo sono quelli formati dagli archi pesanti, li chiameremo *heavy path*.

Idea

Definizione (archi pesanti e leggeri)

Un arco si dice *pesante* ("*heavy*") se, detti u, v i suoi estremi con u padre di v, si ha[a]

$$size(v) \geq \max_{w \in children(u)} size(w)$$

e, se $size(v) = size(w)$, $v < w$. Un arco non pesante si dice *leggero* ("*light*").

[a] con $size(v)$ intendiamo $size(subtree(v))$

I cammini che consideriamo sono quelli formati dagli archi pesanti, li chiameremo *heavy path*.

Animazione

Idea

Teorema

Sia v un nodo. Nel cammino da v alla radice ci sono al più $\log_2 N$ archi leggeri.

Idea

Teorema

Sia v un nodo. Nel cammino da v alla radice ci sono al più $\log_2 N$ archi leggeri.

Dimostrazione: sia u padre di v. Se l'arco (u, v) è leggero, allora

$$size(u) \geq 1 + 2size(v) > 2size(v).$$

Dunque, detto l il numero di archi leggeri da v alla radice, vale

$$N = size(root) > 2^l size(v) \geq 2^l,$$

quindi $l < \log_2 N$. □

Idea

Teorema

Sia v un nodo. Nel cammino da v alla radice ci sono al più $\log_2 N$ archi leggeri.

Dimostrazione: sia u padre di v. Se l'arco (u, v) è leggero, allora

$$size(u) \geq 1 + 2size(v) > 2size(v).$$

Dunque, detto l il numero di archi leggeri da v alla radice, vale

$$N = size(root) > 2^l size(v) \geq 2^l,$$

quindi $l < \log_2 N$. □

Il teorema dice che è possibile decomporre ogni cammino semplice in $\mathcal{O}(\log N)$ archi leggeri alternati a $\mathcal{O}(\log N)$ di heavy path (o sottocammini). Quindi basta costruire un segment tree su ciascun heavy path.

Implementazione

Prequisito: linearizzazione di alberi

Linearizziamo l'albero:

- facciamo partire una DFS dalla radice;
- ogni volta che entriamo in un nodo, lo mettiamo aggiungiamo a una lista.

La sequenza risultante ha la proprietà che $\forall v \; subtree(v)$ è un intervallo che inizia con v[a].

[a] in particolare l'intervallo $[in[v], out[v]]$, con $in[v]/out[v]$ tempo di entrata/uscita della DFS da v.

Implementazione

Prequisito: linearizzazione di alberi

Linearizziamo l'albero:

- facciamo partire una DFS dalla radice;
- ogni volta che entriamo in un nodo, lo mettiamo aggiungiamo a una lista.

La sequenza risultante ha la proprietà che $\forall v \; subtree(v)$ è un intervallo che inizia con v[a].

[a]in particolare l'intervallo $[in[v], out[v]]$, con $in[v]/out[v]$ tempo di entrata/uscita della DFS da v.

Implementazione

Possiamo fare la DFS visitando sempre per primo il figlio più pesante[a]. In questo modo **anche gli heavy path sono intervalli.**

[a]quello corrispondente all'arco pesante.

Implementazione

Prequisito: linearizzazione di alberi

Linearizziamo l'albero:

- facciamo partire una DFS dalla radice;
- ogni volta che entriamo in un nodo, lo mettiamo aggiungiamo a una lista.

La sequenza risultante ha la proprietà che $\forall v \; subtree(v)$ è un intervallo che inizia con v[a].

[a] in particolare l'intervallo $[in[v], out[v]]$, con $in[v]/out[v]$ tempo di entrata/uscita della DFS da v.

Implementazione

Possiamo fare la DFS visitando sempre per primo il figlio più pesante[a]. In questo modo **anche gli heavy path sono intervalli.**

[a] quello corrispondente all'arco pesante.

Ma allora basta tenere un unico Segment Tree per tutti gli heavy path!

Dettagli implementativi

Informazioni

- `par[v]`: padre di v;
- `pos[v]`: posizione di v nell'ordine DFS;
- `size[v]`: dimensione del subtree di v;
- `depth[v]`: profondità di v;
- `head[v]`: nodo in cima all'heavy path di v.

Dettagli implementativi

Informazioni

- `par[v]`: padre di v;
- `pos[v]`: posizione di v nell'ordine DFS;
- `size[v]`: dimensione del subtree di v;
- `depth[v]`: profondità di v;
- `head[v]`: nodo in cima all'heavy path di v.

Step dell'algoritmo

- prima DFS "dfs" per calcolare `size`, `depth` e spostare ogni heavy child all'inizio della corrispondente lista di adiacenza
  - $\forall v$ sortiamo il vettore `adj[v]` per `size` decrescente
- seconda DFS "decompose" per costruire il DFS-order e salvare `head`
  - per v heavy child di u, `head[v] = head[u]`
  - per tutti gli altri figli w, `head[w] = w`

Codice

- https://cp-algorithms.com/graph/hld.html
- https://cses.fi/paste/5c9d34a012f8941682483f/

Centroid Decomposition

Problema

Problema

Dato un albero con N nodi, contare il numero di percorsi distinti di lunghezza K

Problema

Problema

Dato un albero con N nodi, contare il numero di percorsi distinti di lunghezza K

Problema (più semplice)

Dato un albero con N nodi, contare il numero di percorsi distinti di lunghezza K che passano per il nodo c

Soluzione

Problema (più semplice)

Dato un albero con N nodi, contare il numero di percorsi distinti di lunghezza K che passano per il nodo c

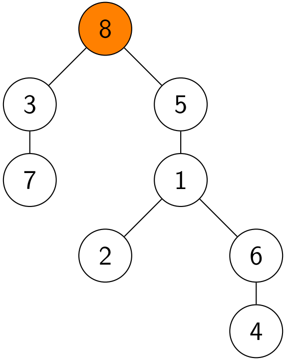
Soluzione

- numero i figli di c da 0 a $M - 1$
- per ogni figlio i conto i percorsi di lunghezza K che partono da i o da un suo discendente e arrivano al figlio j o a un suo discendente, con $j < i$
- per farlo, processo i sottoalberi in ordine e conto il numero di percorsi di lunghezza L per ogni $L \leq K$
- la complessità è $\mathcal{O}(N)$ perché visito ogni sottoalbero una sola volta

Soluzione

Esempio con $N = 8$ e $K = 3$.

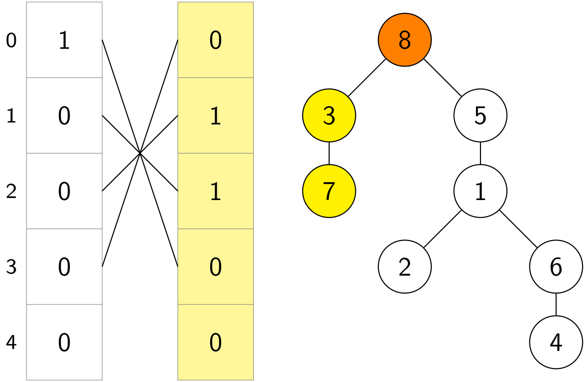
0	1
1	0
2	0
3	0
4	0



Soluzione

Esempio con $N = 8$ e $K = 3$.

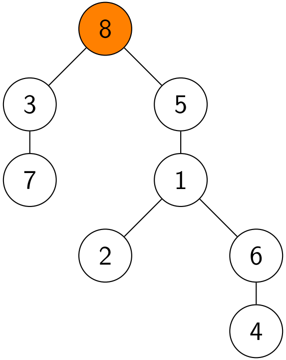
$$\text{ans} \mathrel{+}= 1 \times 0 + 0 \times 1 + 0 \times 1 + 0 \times 0$$



Soluzione

Esempio con $N = 8$ e $K = 3$.

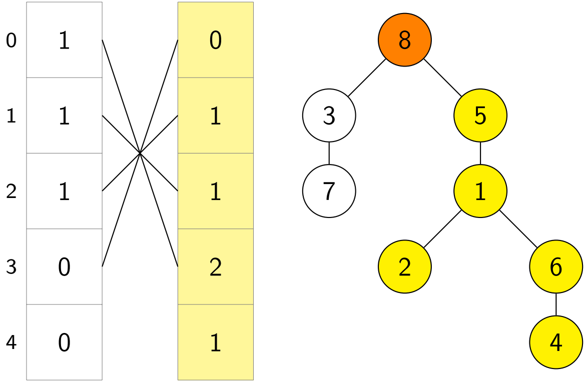
0	1
1	1
2	1
3	0
4	0



Soluzione

Esempio con $N = 8$ e $K = 3$.

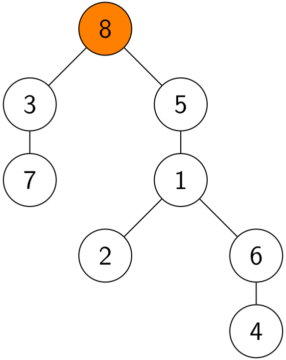
$$\text{ans} \mathrel{+}= 1 \times 2 + 1 \times 1 + 1 \times 1 + 0 \times 0$$



Soluzione

Esempio con $N = 8$ e $K = 3$.

0	1
1	2
2	2
3	2
4	1



Idea

Problema

Dato un albero con N nodi, contare il numero di percorsi distinti di lunghezza K

- Scelgo un nodo c a caso e conto i percorsi lunghi K che passano per c
- Elimino c e ripeto l'algoritmo sui restanti sottoalberi

Idea

Problema

Dato un albero con N nodi, contare il numero di percorsi distinti di lunghezza K

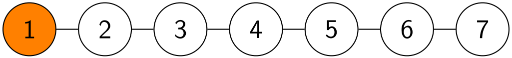
- Scelgo un nodo c a caso e conto i percorsi lunghi K che passano per c
- Elimino c e ripeto l'algoritmo sui restanti sottoalberi

Però la complessita è $\mathcal{O}(n^2)$ nel caso della linea

Caso Peggiore

$K = 3$, $N = 7$

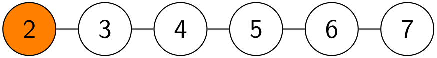
1 percorso che passa per il nodo 1



Caso Peggiore

$K = 3$, $N = 7$

1 percorso che passa per il nodo 2



Caso Peggiore

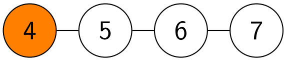
$K = 3$, $N = 7$

1 percorso che passa per il nodo 3

Caso Peggiore

$K = 3$, $N = 7$

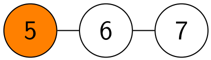
1 percorso che passa per il nodo 4



Caso Peggiore

$K = 3$, $N = 7$

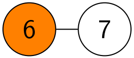
1 percorso che passa per il nodo 5



Caso Peggiore

$K = 3$, $N = 7$

Nessun percorso che passa per il nodo 6



Caso Peggiore

$K = 3$, $N = 7$

Nessun percorso che passa per il nodo 7

Idea

Problema

Dato un albero con N nodi, contare il numero di percorsi distinti di lunghezza K

- Scelgo un nodo c a caso e conto i percorsi lunghi K che passano per c
- Elimino c e ripeto l'algoritmo sui restanti sottoalberi

Però la complessita è $\mathcal{O}(n^2)$ nel caso della linea

Idea

Problema

Dato un albero con N nodi, contare il numero di percorsi distinti di lunghezza K

- Scelgo un nodo c a caso e conto i percorsi lunghi K che passano per c
- Elimino c e ripeto l'algoritmo sui restanti sottoalberi

Però la complessita è $\mathcal{O}(n^2)$ nel caso della linea

Posso scegliere c in maniera da ottenere sottoalberi di dimensione al massimo $\frac{N}{2}$.

Idea

Problema

Dato un albero con N nodi, contare il numero di percorsi distinti di lunghezza K

- Scelgo un nodo c a caso e conto i percorsi lunghi K che passano per c
- Elimino c e ripeto l'algoritmo sui restanti sottoalberi

Però la complessita è $\mathcal{O}(n^2)$ nel caso della linea

Posso scegliere c in maniera da ottenere sottoalberi di dimensione al massimo $\frac{N}{2}$.

In questo modo la complessità è $\mathcal{O}(N \log N)$.

Idea

Problema

Dato un albero con N nodi, contare il numero di percorsi distinti di lunghezza K

- Scelgo un nodo c a caso e conto i percorsi lunghi K che passano per c
- Elimino c e ripeto l'algoritmo sui restanti sottoalberi

Però la complessita è $\mathcal{O}(n^2)$ nel caso della linea

Posso scegliere c in maniera da ottenere sottoalberi di dimensione al massimo $\frac{N}{2}$.

In questo modo la complessità è $\mathcal{O}(N \log N)$.

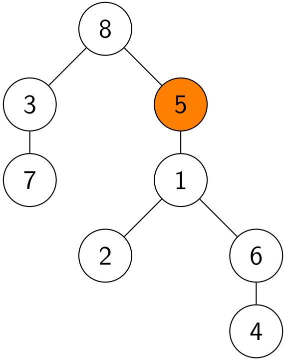
Il nodo c con questa proprietà è il **centroide** dell'albero.

Centroide

Definizione

Il centroide di un albero è un nodo tale che se l’albero viene radicato in esso, nessun sottoalbero ha dimensione maggiore di $\frac{N}{2}$

Centroide



Centroide

Definizione

Il centroide di un albero è un nodo tale che se l'albero viene radicato in esso, nessun sottoalbero ha dimensione maggiore di $\frac{N}{2}$

`find_centroid(u)`

- trova il centroide sapendo che è `u` o un suo discendente
- se ogni sottoalbero di `u` ha dimensione minore o uguale a $\frac{N}{2}$ allora `u` è il centroide
- altrimenti esiste un solo figlio `v` di `u` con dimensione maggiore di $\frac{N}{2}$, la risposta è `find_centroid(v)`.

Per trovare il centroide fisso una radice arbitraria `r` e calcolo `find_centroid(r)`

Implementazione

```
int calc_size(int u) {
  // calcola la dimensione del sottoalbero di u
  // chiama calc_size per i figli di u
  ...
}

int find_centroid(int u) {
  // trova il centroide sapendo che è u o un suo discendente
  for (int v : g[x])
    if (size[v] > N/2)
      return find_centroid(v);
  return u;
}
```

Centroid Decomposition

Centroid Decomposition

La **decomposizione in centroidi** di un albero è un nuovo albero definito in modo ricorsivo:

- la radice è il centroide c dell'albero originale
- i figli di c sono i **centroidi** dei sottoalberi ottenuti rimuovendo c dall'albero originale
- i sottoalberi dei figli di c sono le **decomposizioni in centroidi** dei loro sottoalberi nell'albero originale

L'albero risultante si può chiamare anche **albero dei centroidi**

Centroid Decomposition

Centroid Decomposition

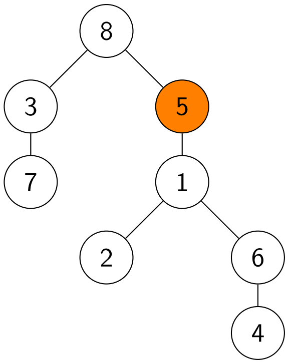
La **decomposizione in centroidi** di un albero è un nuovo albero definito in modo ricorsivo:

- la radice è il centroide c dell'albero originale
- i figli di c sono i **centroidi** dei sottoalberi ottenuti rimuovendo c dall'albero originale
- i sottoalberi dei figli di c sono le **decomposizioni in centroidi** dei loro sottoalberi nell'albero originale

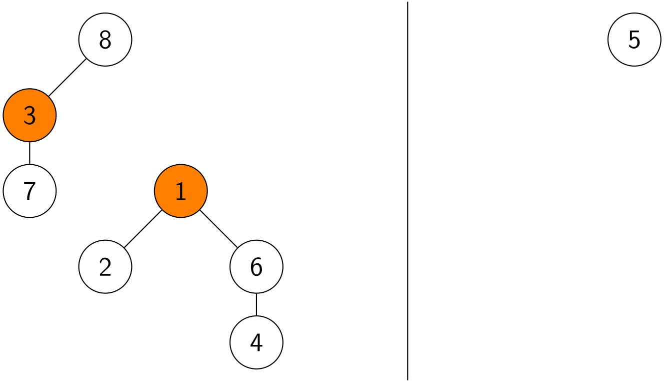
L'albero risultante si può chiamare anche **albero dei centroidi**

Si chiama componenente di un nodo u l'insieme composto da u e dai sui discendenti nell'albero dei centroidi.
Corrisponde anche al sottoalbero dell'albero di partenza di cui u è il centroide.

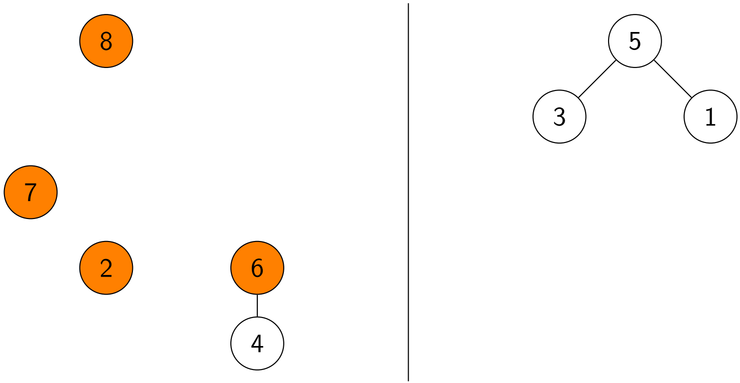
Centroid Decomposition



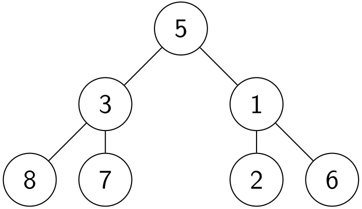
Centroid Decomposition



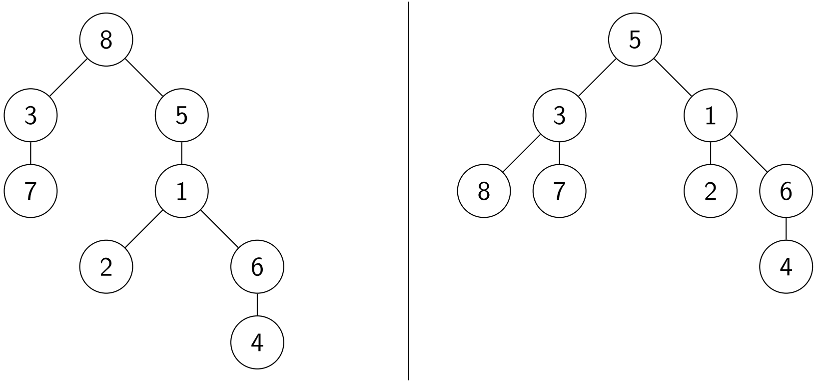
Centroid Decomposition



Centroid Decomposition



Centroid Decomposition



Proprietà

Teorema

L'altezza dell'albero dei centroidi è al più $\log N$

Dimostrazione:

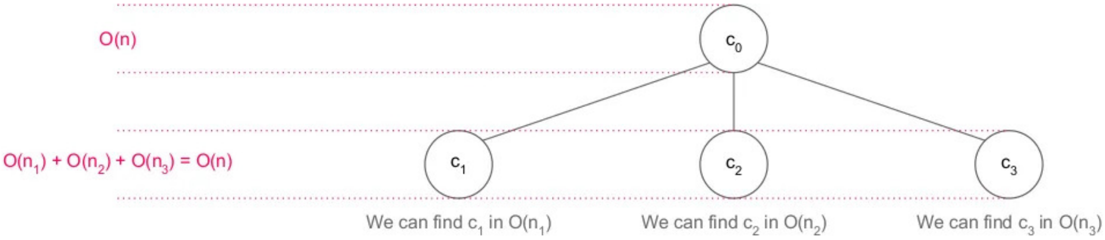
Sia `dim[u]` la dimensione della componente di cui `u` è il centroide.

Se `v` è padre di `u` allora vale `dim[v]` $\geq$ `dim[u]*2`.
(altrimenti v non sarebbe il centroide della sua componente)

Siccome in totale ci sono N nodi, ogni nodo ha al massimo $\log_2 N$ antenati.

Proprietà

La complessità per calcolare la centroid decomposition di un albero è $\mathcal{O}(N \log N)$, siccome nell'albero ci sono $\mathcal{O}(\log N)$ livelli, ciascuno con $\mathcal{O}(N)$ nodi



Proprietà

Teorema

Il percorso che collega i nodi a e b nell'albero originale passa per `lca(a,b)` nell'albero dei centroidi

Dimostrazione:

Ogni antenato di un nodo `u` nell'albero dei centroidi è centroide di una componente che contiene `u`.

Quindi `lca(a,b)` nell'albero dei centroidi è un nodo `c` la cui componente contiene sia `a` sia `b`.

Se il percorso fra `a` e `b` nell'albero originale non passasse per `c` esisterebbe una componente più piccola di quella di `c` che contiene sia `a` sia `b`.

In altre parole, siccome, rimuovendo `c`, `a` e `b` sono stati divisi in due sottoalberi diversi, il percorso da `a` a `b` deve passare per `c`.

Problema

Problema

Dato un albero di N nodi, inizialmente bianchi, supportare due tipi di query

1. Colora il nodo `u` di rosso
2. Trova la minima distanza dal nodo `u` a un nodo rosso

Problema

Problema

Dato un albero di N nodi, inizialmente bianchi, supportare due tipi di query

1. Colora il nodo `u` di rosso
2. Trova la minima distanza dal nodo `u` a un nodo rosso

Prima Idea

Implemento le query come vengono descritte: per rispondere alle query di tipo 2 eseguo una bfs dal nodo `u`

1. Imposto `colored[u] = true` — Complessità: $\mathcal{O}(1)$
2. Eseguo una bfs dal nodo `u` — Complessità: $\mathcal{O}(N)$

Problema

Problema

Dato un albero di N nodi, inizialmente bianchi, supportare due tipi di query

1. Colora il nodo `u` di rosso
2. Trova la minima distanza dal nodo `u` a un nodo rosso

Seconda Idea

Per ogni nodo `u` salvo la distanza `best[u]` dal nodo rosso più vicino

1. Eseguo una dfs dal nodo `u` per aggiornare `best` — Complessità: $\mathcal{O}(N)$
2. Restituisco `best[u]` — Complessità: $\mathcal{O}(1)$

Problema

Problema

Dato un albero di N nodi, inizialmente bianchi, supportare due tipi di query

1. Colora il nodo `u` di rosso
2. Trova la minima distanza dal nodo `u` a un nodo rosso

Terza Idea

Fisso la radice in `0`.
Per ogni nodo `u` salvo la distanza `best[u]` dal nodo rosso più vicino **nel suo sottoalbero**

1. Aggiorna `best[v]` per ogni `v` antenato di `u` — Complessità: $\mathcal{O}(H)$
2. Per ogni antenato `v` di `u` calcolo il valore `dist(u, v) + best[v]` e prendo il minore fra i risultati — Complessità: $\mathcal{O}(H)$

H è l'altezza dell'albero.
Nel caso peggiore $H = N$, entrambe le query hanno complessità $\mathcal{O}(N)$

Problema

Problema

Dato un albero di N nodi, inizialmente bianchi, supportare due tipi di query

1. Colora il nodo `u` di rosso
2. Trova la minima distanza dal nodo `u` a un nodo rosso

Idea Vincente

Calcolo la **centroid decomposition** dell'albero. Per ogni nodo `u` salvo la distanza `best[u]` dal nodo rosso più vicino **nella sua componente**

1. Aggiorno `best[v]` per ogni `v` antenato di `u` **nell'albero dei centroidi** — Complessità: $\mathcal{O}(\log N)$
2. Per ogni antenato `v` di `u` **nell'albero dei centroidi** calcolo `dist(u,v) + best[v]` e prendo il minore fra i risultati — Complessità: $\mathcal{O}(\log N)$

Attenzione `dist(u,v)` si riferisce all'albero originale!!