Competitive Programming

Simone Stanganini

November 26, 2025

Contents

1 100 prisoners

Trump da un'ultima possibilità di freedom a 100 scenziati, trump da un numero da 1 a 100 ad ogni scenziato.

IMG MANCANTE
Screenshot_20251110_094931.png

C'è una stanza con 100 cassetti, ognuno contiene un numero ma non si conosce, ogni scenziato entra e può aprirne 50. Gli scenziati vincono se tutti trovano il loro numero. si comunicano all'inizio ma dopo non possono.

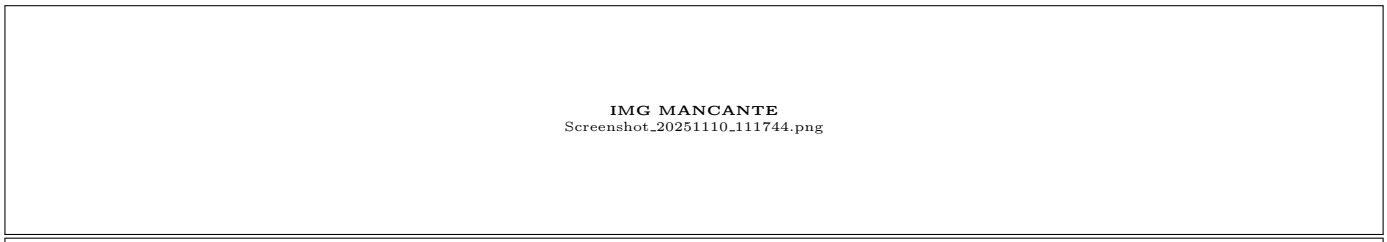
- Trivial Solution: nessuno si mette d'accordo e ognuno apre a caso. $\frac{1}{2}$ oppure il 50% di successo per il singolo ma $\frac{1}{2^{100}}$ che tutti vincono, coglioniiiii.

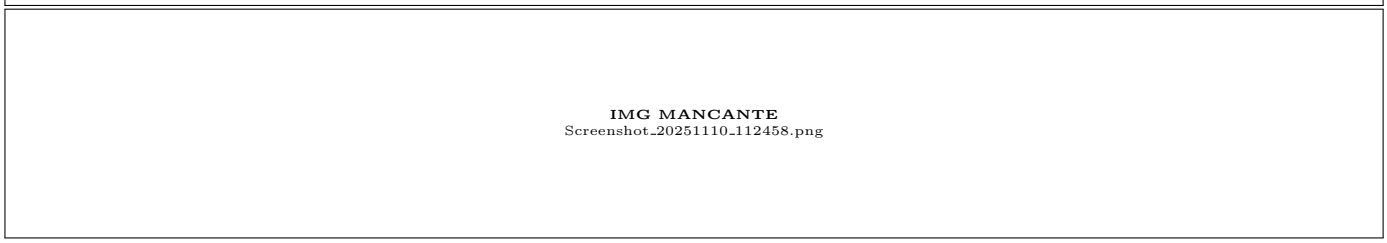
Si usa una strategia adaptive, per aprire i drawer.

1. Scenziato numero i apre il drawer i con numero d_i, se $d_i == i$ allora vince.
2. Se no apre il drawer d_i e continua cosi.

IMG MANCANTE
Screenshot_20251110_103613.png

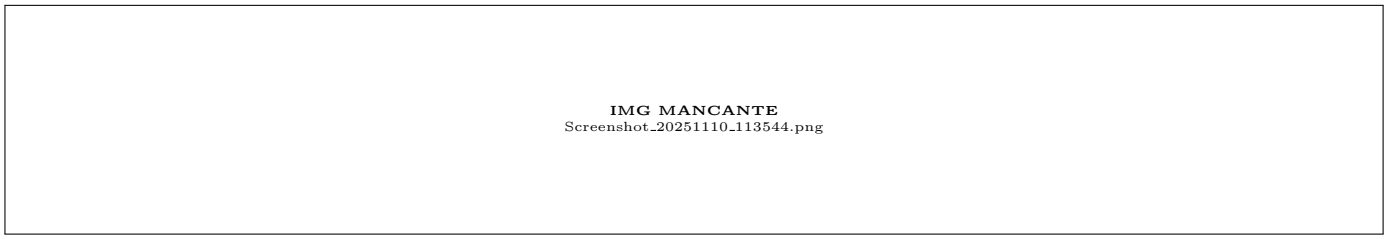
Ovviamente è un ciclo, quando lo percorriamo tutto si arriva alla soluzione, perche l'ultimo è il nostro, quindi per vincere dobbiamo prendere un ciclo minore di 50, quindi quale è questa probabilità?



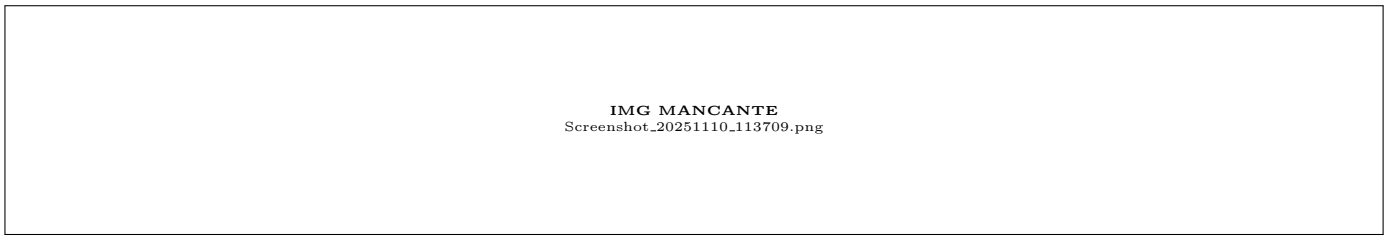


Si ma non esistono cicli in questi drawer, c'è solo un ciclo, cambia solo da che nodo ci entri.

Quindi l'equazione diventa tutte le permutazioni possibili, l'ordine che hanno visto che lo sappiamo gia, e poi gli elementi che mancano:



quindi come si calcola? si fa:



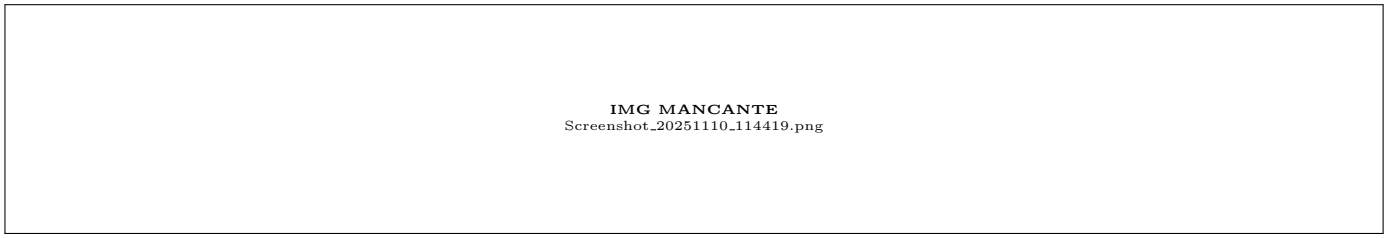
Vediamo che non è brutto adesso, rispetto alla soluzione trivial, abbiamo una probabilità piu grande yeeee.

2 Super clever algoritm

2.1 Duplicate Eleemnts in an array

Siamo un un array $A[0, ..., n]$ di n+1 numeri in $\{0, ..., n-1\}$, dobbiamo trovare almeno un elemento duplicato.

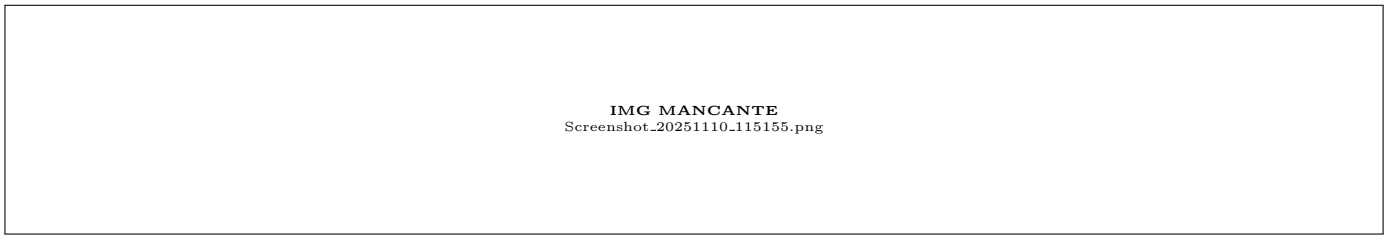
Si usa una bit map per vedere se abbiamo gia visto quel valore:



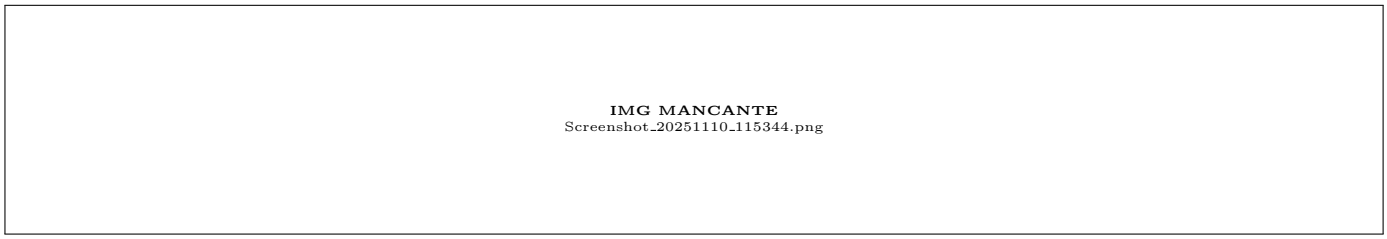
Complessità, $\Theta(n)$ tempo, ma anche $\Theta(n)$ space, quindi dobbiamo sistemarlo. Sembra impossibile perche gia usiamo solo 1 bit per risolvere questo problema.

2.1.1 log n passes

Si fanno $\Theta(\log n)$ passes su A, ovvero si ricostruisce l'array e (bitmap) bit per bit.



si guardano prima le cifre piu alte dei numeri, poi si guardano quelle intermedie and so on, cosi si fa lungo $\log n$ e, e ci settiamo solamente chi troviamo

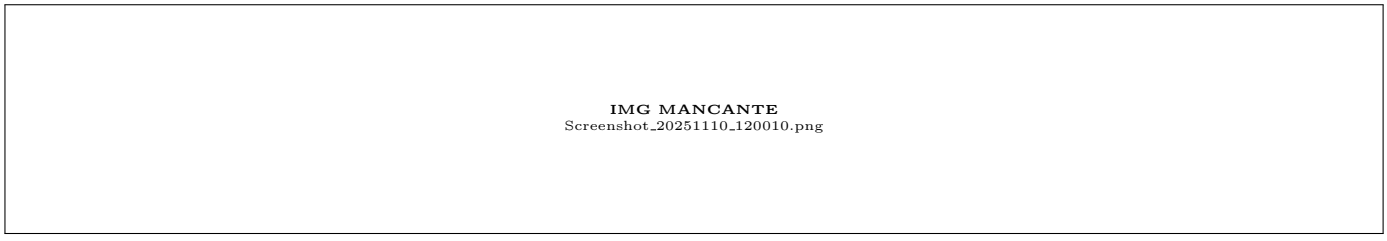


Greve in questa mnaiera ci troviamo i duplicati direttamente alla fine. Per ogni cifra si conta chi ne ha di piu, se troviamo piu 1 o piu zeri ce lo segniamo e si restringe via via il campo. Notiamo che è molto bellino ma solo se faccimao niente random access, se no siamo piu lenti di prima insomma.

2.1.2 Destroy A

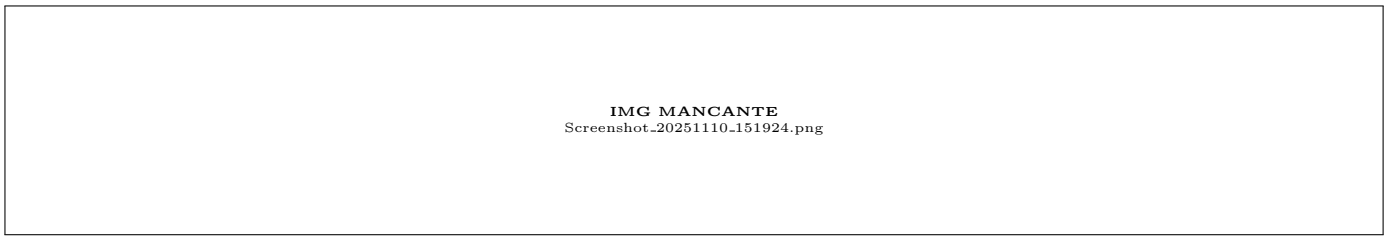
Ma se potessimo fare random access possiamo andare piu veloce? possiamo ma si distrugge A, quindi dobbiamo fare una copia in caso. Questa soluzione deriva dai 100 prisoners.

Si spostano i numeri nella loro posizione corretta, e quindi basta controllare la loro posizione una volta che si trovano (random access), e via.



2.2 Floyds cycle finding

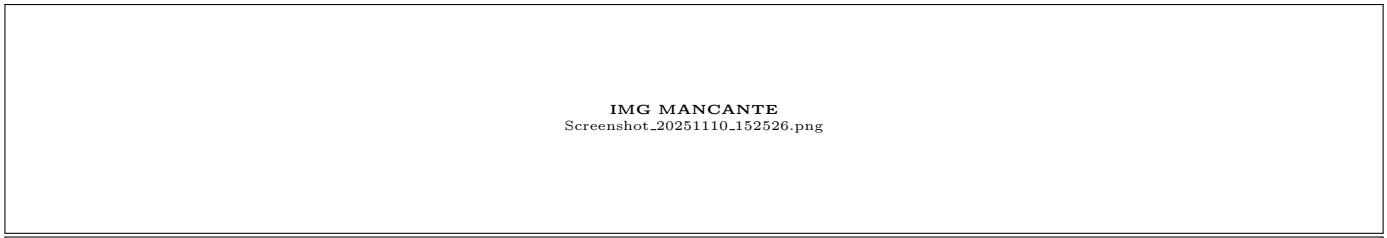
Trovare l'inizio di un ciclo in una lista in $\Theta(1)$ extra space, la lista è read only

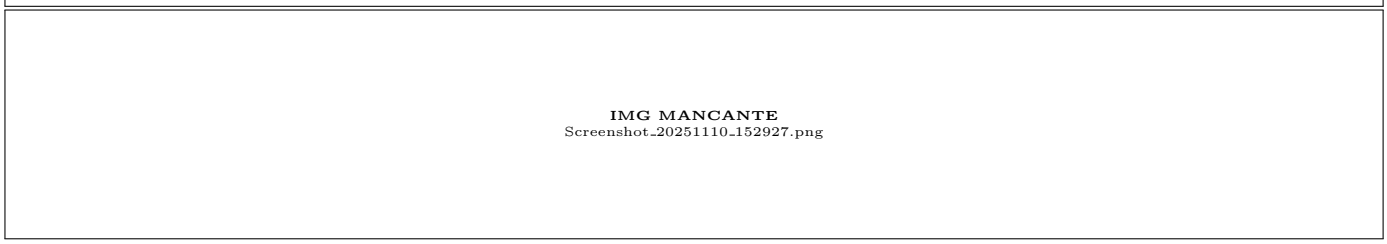


Si usano 2 puntatori, S Slow si muove solo 1 step, e F Fast che si muove 2 step by 2.

1. Prima si trovano nel ciclo, cosi da certificare il fatto che il ciclo esiste (lo sappiamo per cui seccederà).
2. Poi riportiamo F all'inizio e lo settiamo con speed 1, mentre intanto S continua.

MAGIA: Quando si ritroveranno, si ritroveranno all'inizio del ciclo. Questo perche S fa m steps, mentre F fa $2m$ steps, si chiama a la lunghezza del prefisso prima dell'inzio del ciclo, mentre b quanto sono dentro al loop, per cui:





$$m = a + b$$

$$2m = a + b + k * \text{length of the loop (=b)}$$
$$a = k * l - b$$

2.3 Majority Element

Dato un array $A[1, n]$ trovare se c'è, l'elemento che occorre almeno $\frac{n}{2} + 1$ volte, quindi la maggioranza.

Soluzioni:

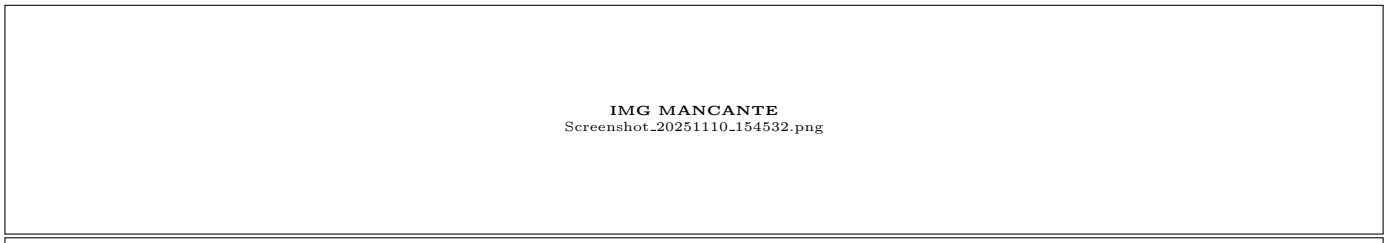
- Sort $\Theta(n \log n)$ time $\Theta(n)$ space
- hashmap $\Theta(n)*$ time $\Theta(n)$ space (l'hashmap è probabilistica dipende)

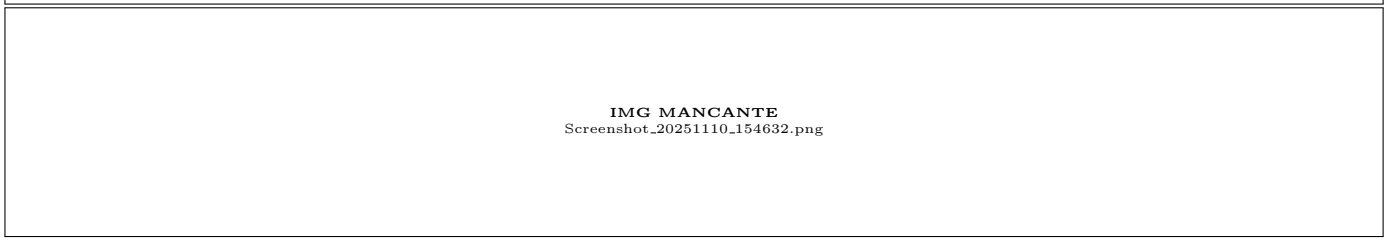
possiamo fare meglio in spazio? vogliamo tempo lineare $\Theta(n)$ e spazio costante$\Theta(1)$,

2.3.1 Boyer Moore Algorithm

Abbiamo un array.

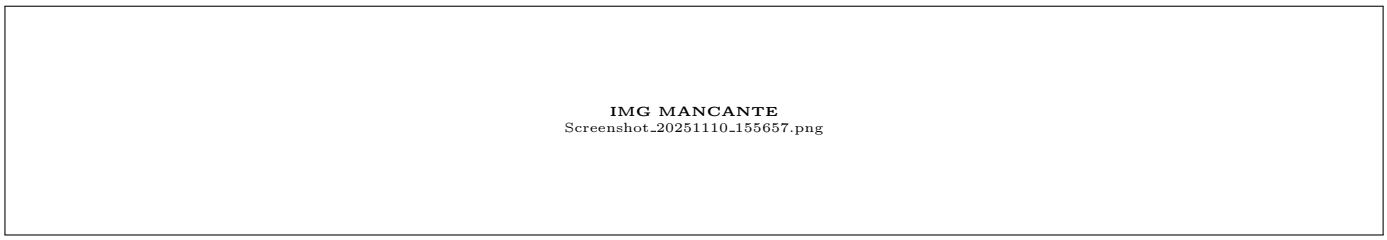
Si prende un candidato c e a un counter, si processa da sinistra a destra, e ad ogni itereazione abbiamo $A[i]$, e se $A[i]$ == c si aumenta il counter di 1, se no :



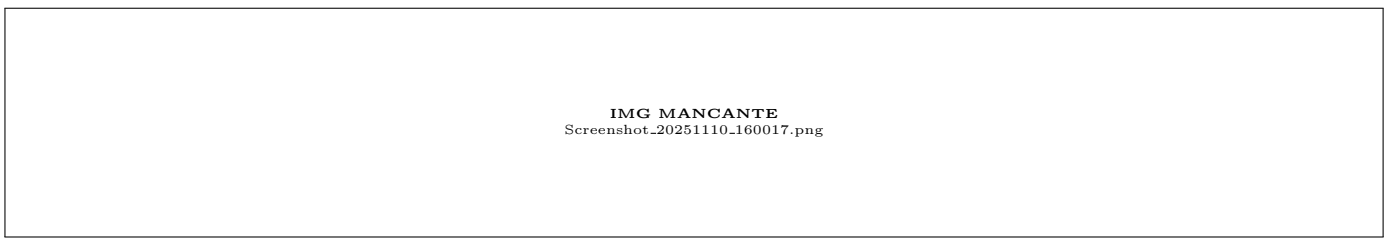


2.3.2 Support Insertion/Deletion

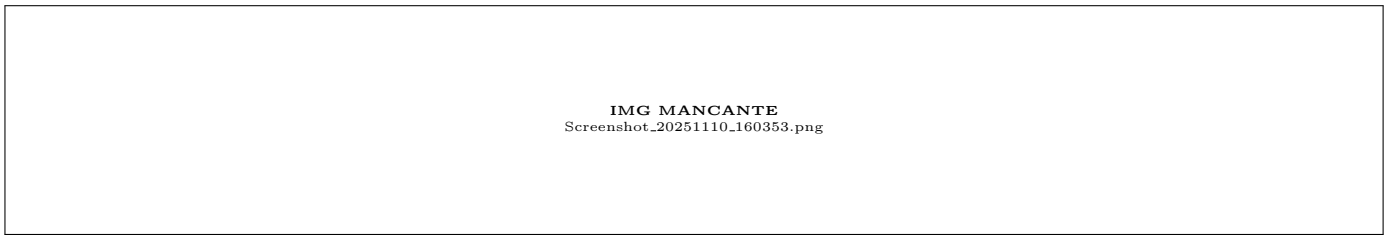
Abbiamo un array, si inserisce nella sequenza, oppure si fa la deletion, e dopo ogni singola operazione ci chiediamo quale è il majority. E vogliamo $\Theta(\log n)$ time e space.



Si fa la binary rappresentation per questa robetta, e si contano il numero di 0 e di 1.



bellissimi, adesso se facciamo un sacco di operazioni di delete e una insert, come si fa? basta che si fa la update dei nostri calcoli, una delete


IMG MANCANTE
Screenshot_20251110_160353.png

2.4 Misra-Gries Heavy Hitter

Dato un array$A[1, n]$, trovare l'elemento $T - 1$ che occore almeno $\frac{n}{T} + 1$ volte, stiamo generalizzando il problmea majority, con $T = 2$ è quello.

Si tiene un set K di T candidati e counters,

IMG MANCANTE
Screenshot_20251110_161014.png

Abbiamo $\Theta(T \cdot n)$ Time, godo

3 Alice and bob

Alice seleziona prima e Bob seleziona dopo, 2n numeri, Regola di Scelta: Un giocatore può prendere un numero solo da una delle due estremità della fila rimasta. Obiettivo: Ogni giocatore vuole massimizzare la somma totale dei numeri che ha raccolto. Alice vince se la sua somma finale (SA) è maggiore o uguale a quella di Bob (SB).

L'intuizione chiave: Scegliendo un numero da una posizione dispari (la 1°), Alice costringe Bob a scegliere tra due numeri che sono entrambi in posizioni pari. Qualunque numero Bob scelga, "scoprirà" un nuovo numero in posizione dispari per Alice.

Alice può continuare così per tutto il gioco, prendendo sempre un numero dal set "Dispari", e Bob sarà sempre costretto a prendere un numero dal set "Pari".

grazie gemini! insomma lei prima calcola quale è la somma vincente, se quella nei numeri pari o nei numeri dispari (posizioni negli array) e cosi cosi gasse.

4 Fenwick Tree binary indexed tree, BIT

4.1 Dynamic Prefix Sums

Dato un array $A[1, n]$ vorremmo supportare le seguenti operazioni: IN LOGARITMIC TIME.

- $\text{Sum}(i) = \sum_{k=1}^{i} A[k]$
- $\text{Add}(i, v) = A[i] += v$

$RangeSum(i, j) = Sum(j) - Sum(i - 1)$

Solution Prendiamo una versione rilassata del problema, la add rimane uguale, ma la sum si fa solo per le posizioni i che sono una potenza di due:

IMG MANCANTE
Screenshot_20251114_112943.png

Qui creaiamo un albero per la prefix sum, solamente per le potenze di due, cosi se ci chiedono la sum la sappiamo, e se chiedono la add, possiamo farlo, tipo add 2 in posizione 3, basta che aggiungiamo nel nodo 4 e 8 dell'albero 2, e si fa un tempo logaritmico.

IMG MANCANTE
Screenshot_20251114_113058.png

Adesso visto che facciamo questa cosa solo per le potenze di due, vediamo come tutti i nodi hanno solamente un bit settato a 1.

IMG MANCANTE
Screenshot_20251114_113838.png

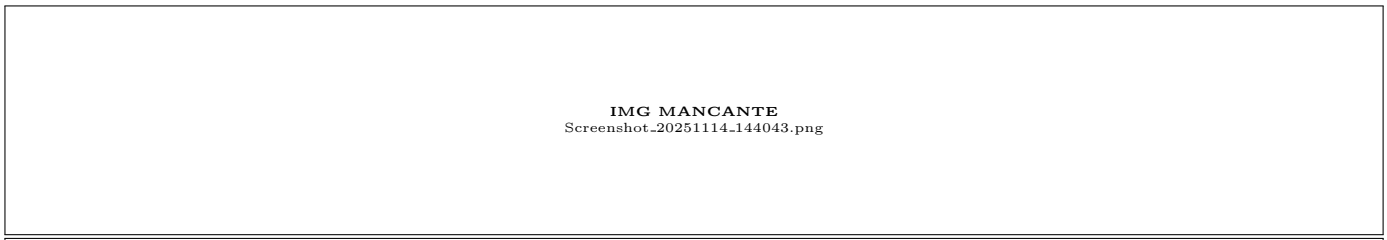
Qui si continua con 2 bit a 1, e vediamo come i figli vengono messi seguendo i bit del padre, il nodo 3 ha 2 bit dopo il nodo 2, uguale il 5 e il 6 hanno 2 bit dove il padre ne ha solo 1, mi so spiegato di merda ma si vede via. Praticamente il prefisso è del padre e il suffisso è del figlio, cosi si capisce meglio.

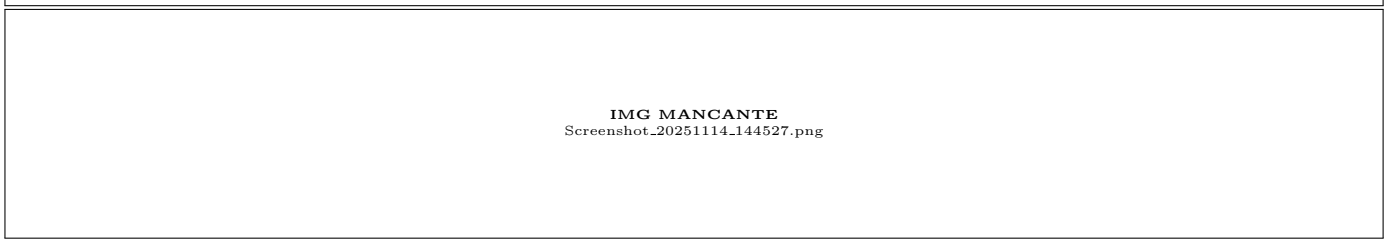
IMG MANCANTE
Screenshot_20251114_120123.png

Adesso cerchiamo di capire la regola, manca solo il nodo 7, dove va messo? si mette nel livello che ha quanti 1 ci sono nella sua rappresentazione binaria, 7 = `0111` quindi la terzo livello, e il suo padre chi sarà? 6 ovviamente, il suo prefisso è `011` e quindi suo figlio sarà per forza `0111`

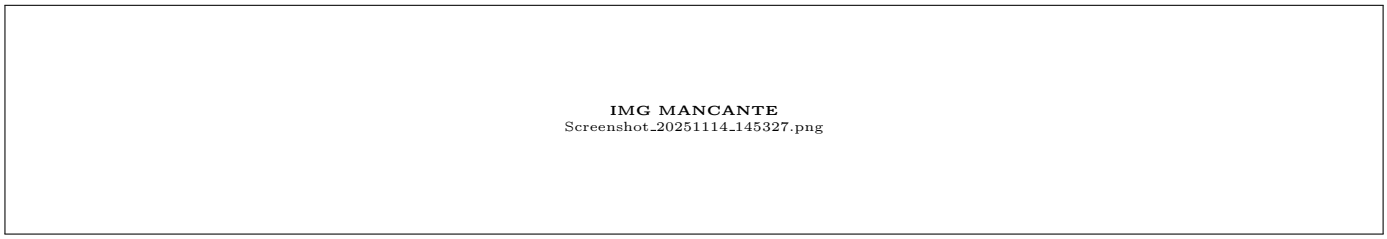
Quindi non serve una structure per il tree, non ci serve nemmeno l'array inziale, si tiene solo il nostro FenwickTree dentro l'array, tanto con sum si ricostruisce tutto, quindi 0 extra space.

COme si fa la sum? si parte dalla posizione chiesta ? quindi sum(3) si parte dal nodo tre dell'albero e si somma con i suoi genitori, quindi ci serve l'operazione di parent() e basta per fare la sum, quindi gasse.

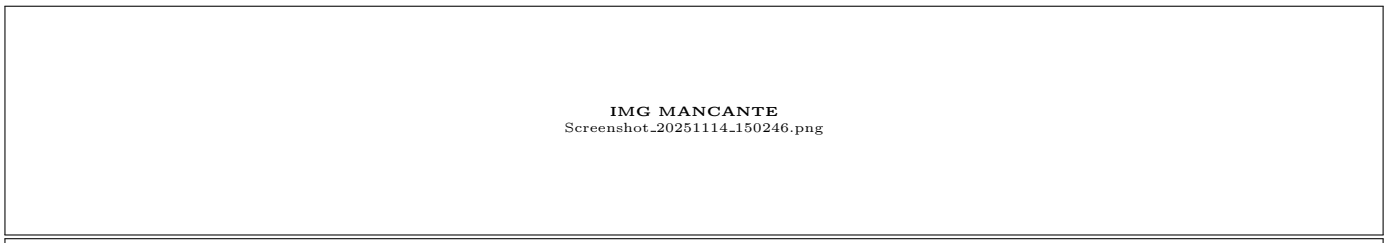


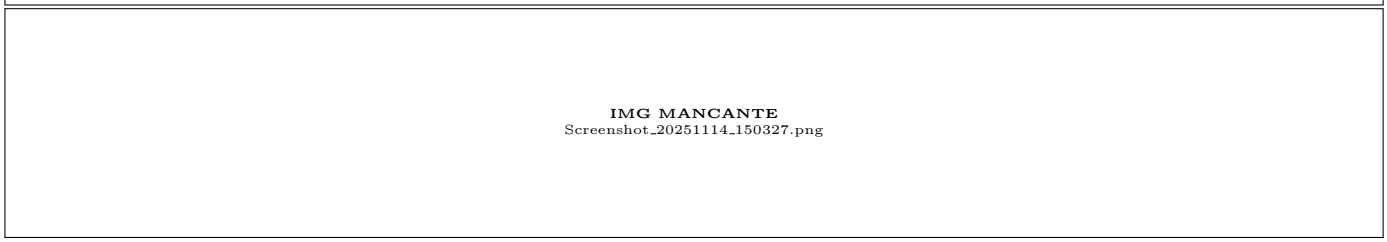


La cosa figa di qeusta struttura è che ha 0 extra space e il modo per calcolare il parent super easy. Quale è la sua **altezza**? $\log n$, questo perche sono tutte le potenze di 2. godo



Per fare la add è la solita storia, dobbiamo andare a destra e in alto, quindi gasse dobbiamo trovare il nodo a destra, ed è come il parent ma invece che togliere il trailing one (appunti vecchi gasse) si aggiunge.

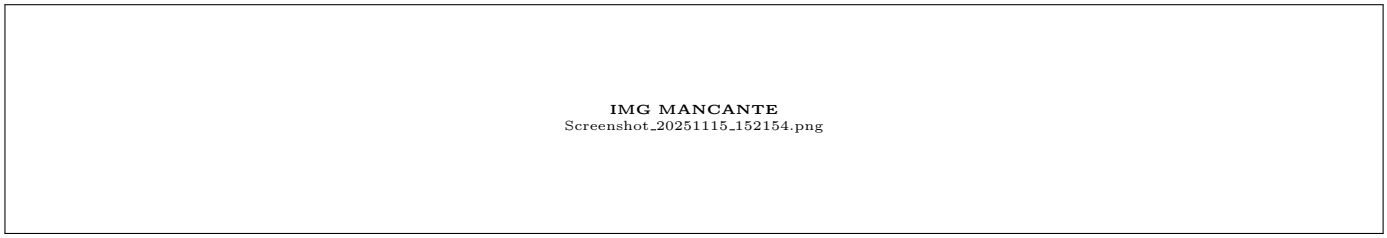




4.2 Counting Inversion

Dato $A[1n]$ di numeri poisitivi $\leq n$ contare il numero di inversions in A.

Una inversion è $i < j$ such that $A[i] > A[j]$.

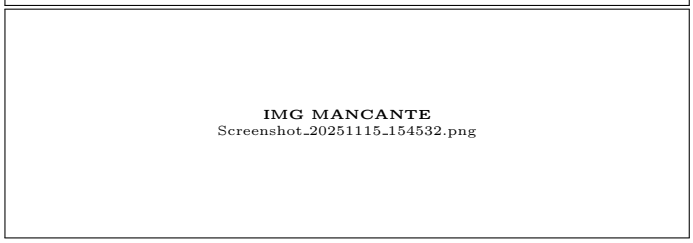


Questo conta quanto non è sortato l'array, ovviamente il bound è quando l'array è sortato in revers order.

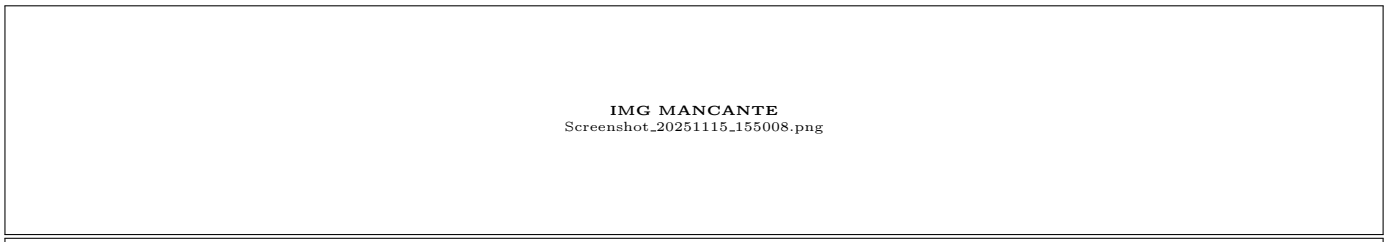
Soluzione trivial, fare insertion sort o contare gli steps, ma è $\Theta(n^2)$ time. C'è anche una soluzione mergesort based n logge enne ma non ci freca (basta contare le singole inversioni che fa il mergesort).

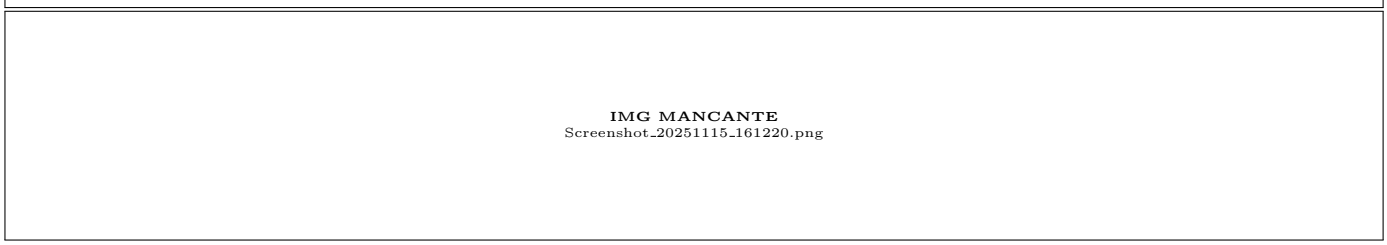
FT based: si costruisce il FenwickTree dove in ogni posizione $F[j]$ ha la frequenza dell'elemnto di $A[i] = j$, si fa uno scan da sinistra a destra di A, e si da l'update di questa bellissima cosa. Mentre popoliamo il FT è facilissimo vedere quante inversioni abbiamo all'interno di A, basta fare una rangesum dalla posizione dopo il nostro j fino alla fine del FT, quindi easy. come si vede dall'immagine si arriva a 1 e basta fare la somma di tutto l'array dopo e gasse. Quindi $\Theta(n \log n)$ time.

In caso il max value fosse troppo grosso, chiaramente non dobbiamo fare un FT enorme, ma basta remappare. Si sortano e diamo il rank, e si sostuiscono i valori con il loro rank, tnato ci importa delle inversion non dei valori effettivi, godo.



4.3 Nested Segment





IMG MANCANTE
Screenshot_20251115_162556.png

Facciamo un bel remap con l'inizio e la fine dei segmenti, successivamente si fa uno scan e si mette 1 nelle posizioni dove i segmenti iniziano (sinistra ovviamente diobe).

Quindi ora si fa la range sum dentro il range di un segmento, e infatti gasse si trova la risposta al nostro problema, ma comunque dobbiamo escludere delle cose, 7 e 9 sono di troppo, ma come si fa?

IMG MANCANTE
Screenshot_20251115_162857.png

Si inizia dall'ultimo segmento, che sicuramente va benissimo e si fa la sua range sum, e poi che si fa? siamo una struttura dinamica, allora si elimina il suo inizio, gasse cosi si toglie per gli altri, cosi si tratta sempre il segmento corrente come l'ultimo gasseeeeeeeeeeeeeeeeeeeeee.

4.4 Update the Array

Dato un array di soli 0 vogliamo supportare 2 operazioni:

- Access(i) = A[i]
- Update(i,j,v) = $\forall k \in [i, j]$ allora $A[k] += v$

IMG MANCANTE
Screenshot_20251115_172803.png

Si tiene V ovvero un FT, dove la posizione i è la prefix sum fino ad i di A, ovvero $\forall i A[i] = \sum_{k=1}^{i} V[k]$, praticamente la prefix sum di V è l'array A. Cosi se facciamo +1 in una posizione di V, questo si propaga direttamente in ogni posizione a destra, in questa maniera se si fa l'update, basta mettere $+v$ in posizione i e $-v$ in posizione $j + 1$, cosi da cancellare quella propagazione

IMG MANCANTE
Screenshot_20251115_173156.png

Ogni update si trasla con 2 add sul FT per funzionare, mentre un access è semplicemente una somma

IMG MANCANTE
Screenshot_20251115_173247.png

Una cosa che non avevo capito, il FT non è una scatola nera, fa solamente parent e moveright, serve solo a questo, in questo caso la add non propaga come abbiamo visto inizialmente per fare la dynamic prefix sum madonna bestia laida, quindi semplicemente si fa la add sul singolo nodo e gasse cosi quando facciamo la sum effettivamente è tutto aggiornato, godo. E' vero che in questo caso V (l'FT) è la prefix sum di A in ogni singolo i, godo, e per questo si fanno due add per le update.

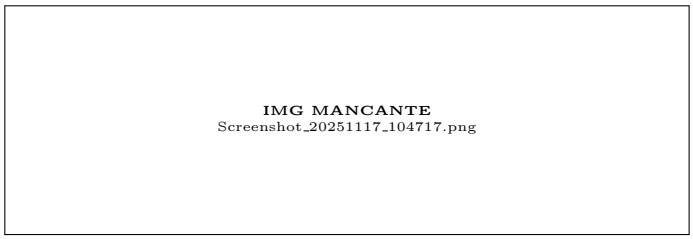
4.5 Dynamic Prefix Syms with range update

Dato A[1,n]

- rangeupdate(i,j,v)
- sum(i)
- access(i) = sum(i) - sum(i-1)

NOtare, la sum(i) non è come la $sum_v(i)$ del problema precedente, vogliamo proprio la sum di i in a quindi gasse.

Il sunto è che se gestiamo la rangeupdate come facciamo di solito, non abbiamo dei risultati corretti, in questo caso il prof dice moltiplichiamo per i la sum del FT, ma è corretto solo se si inizia sempre dall'inizio con la rangeupdate diobestia.

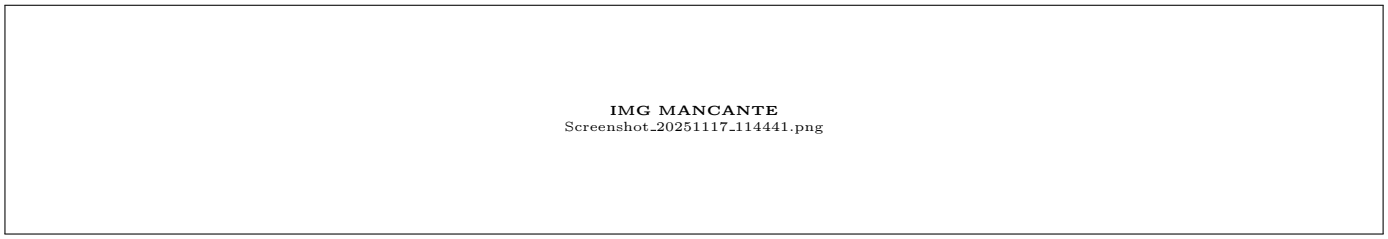


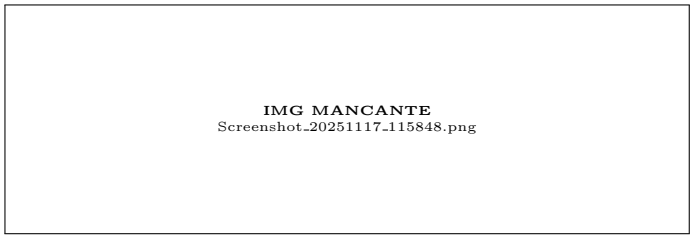
Dobbiamo usare 2 FT, il primo si usa come prima, stessa strategia, ma dobbiamo avere un secondo FT con le correzioni del primo.

5 Segment Tree

Stesso problema del FT, sempre la add ma un problema diverso per la range sum è piu generale, per esempio range min

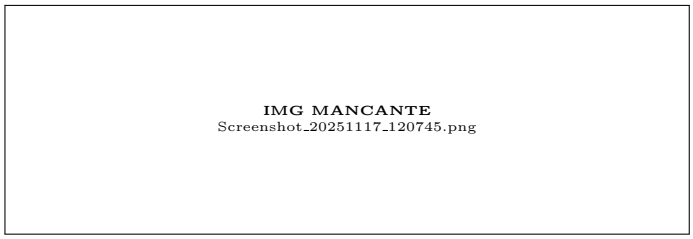
Ricordarsi: occupa il doppio dello spazio del FT, visto che FT è implicito





Si fa un binary tree sopra i valori, combinando le coppie delle posizioni, nei padri si mettono le sum dei figli, visot che la nostra operazione è quella (?),

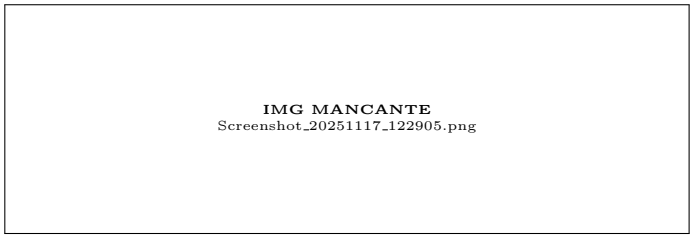
Praticamente il funzionamento è: ci teniamo nei nodi la risposta ai nodi sotto di noi.



Ora ci serve una update e un modo per rispondere al problema. E la update deve fare $\Theta(\log n)$ nodi.

Per la range sum come si fa? in questo caso abbiamo query range 2 e 6, e si fa la compare con i node range, e abbiamo 3 casi,

1. Total Overlap: quando il query range contiene totalmente il node range, quindi la risposta del nodo è richiesta per rispondere alla domanda richiesta. Si prende direttamente il valore e lo usiamo per la risposta
2. Partial Overlap: la risposta che cerchiamo è dentro in un qualche sub tree dentro il nostro nodo. Il nodo non ha la risposta ma solamente un suo descendent ha la risposta, quindi si deve continuare la ricorsione, in quale side? si va in tutte e due.
3. No overlap: nessun nodo ha questa risposta, per cui non si prende. Si lascia stare nessun overlap ci stoppiamo.



IMG MANCANTE
Screenshot_20251117_124650.png

E' un'algoritmo recursive che parte dal nodo, e ha tutti e 3 i casi, Si vede bene che si arriva a certi nodi dove abbiamo del partial overlap, e semplicemente seguiamo quello che abbiamo gia fatto

Quindi è linear o logaritmic? è linear nel nostro esempio, per cui come si fa? con questi partial overlaps diventa linear a forsa di andare a sinistra e destra. Non ho capito però i discorsi del prof, e il running time è log n insomma, perche si usa toccano almeno $4 \log n$ nodi.

Ora un'altro problema che questa struttura può risolvere:

IMG MANCANTE
Screenshot_20251117_130309.png

IMG MANCANTE
Screenshot_20251117_130309.png

5.1 BMQ with occurrency

IMG MANCANTE
Screenshot_20251117_165416-1.png

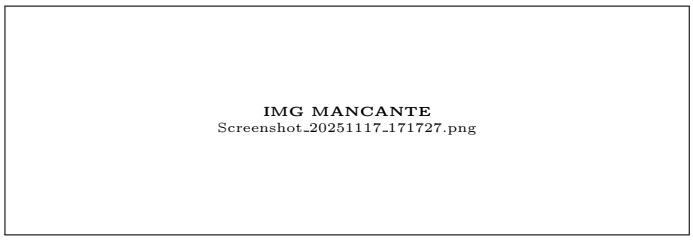
IMG MANCANTE
Screenshot_20251117_165854.png

Allora si crea l'abero per fare questo bellissimo algoritmo, se comunque dovessimo fare una update viene lo stesso, vabbe fa cagare sto problema perche lo ha spiegatooooooooooooooooo

5.2 problema piu greve

Array con NON NEGATIVE integer, vogliamo:

- add(i,v)
- Rangesum(i,j)
- search(s) : report the position i of the shortest prefix of A such that $\sum_{j=1}^{i} A[k] \geq s$

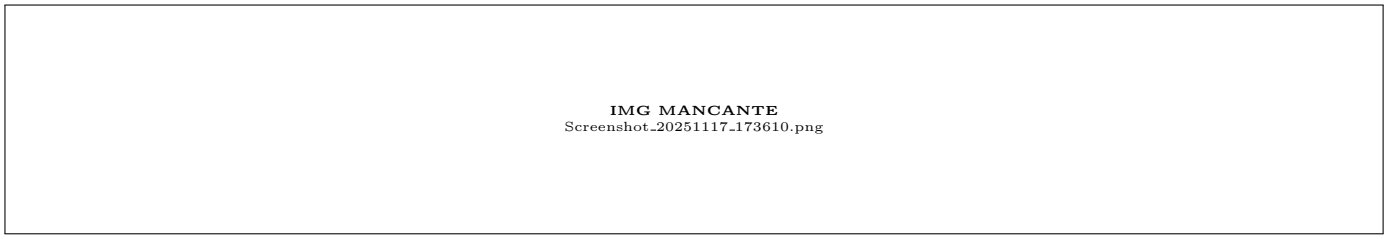


Trivial si cerca con una binary search ma viene una merdina, quindi che si fa, dobbiamo fare la ricerca di 23, si inizia dalla root, si dice ma 8 è troppo piccolo per arrivare a 23, quindi si va direttamente a destra perche si cerca 23-8, allora si cerca per 15 visto che togliamo il figlio di sinistra, allora 19 è maggiore o uguale a 15, allora si cerca li dentro, 6 è piu piccolo di 15, allora si cerca 15-6 e infatti 13 è piu grande allora gasse.

Quale è il running time? va in $\Theta \log n$

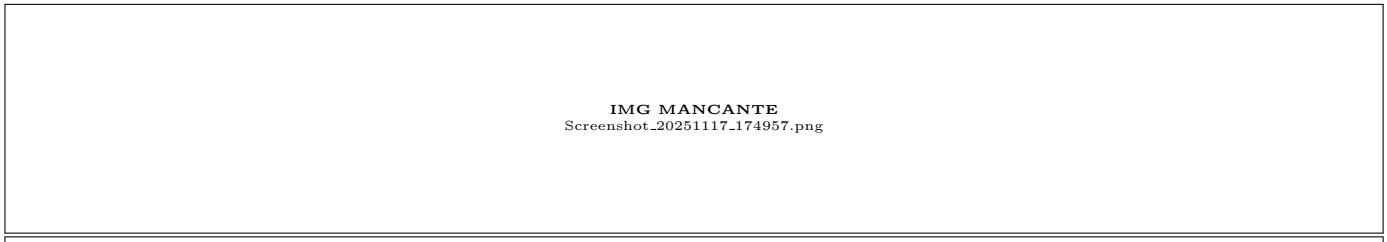
5.3 problema piu greveeeeee

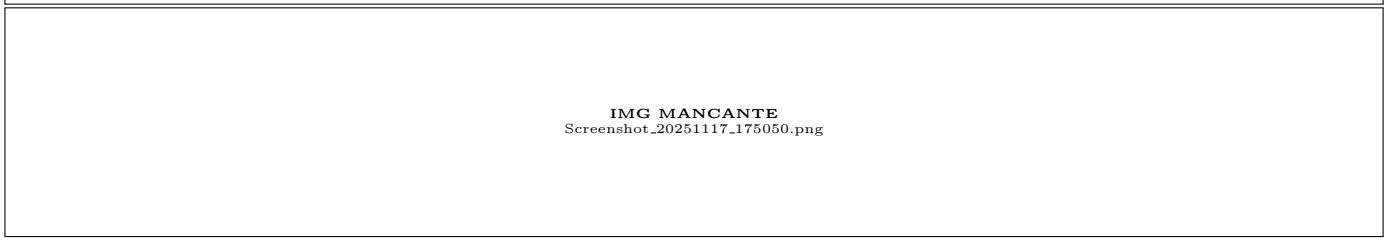
Dato A di interi:

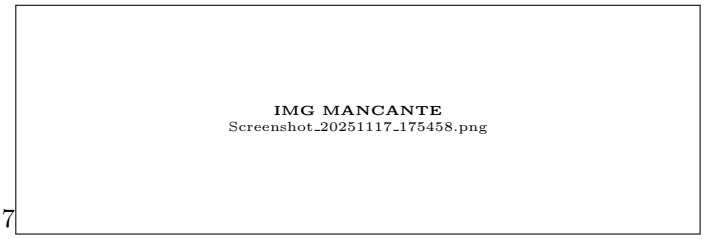


Notare che abbiamo $\Theta \log n$ mentre abbiamo $\Theta n \log n$ spazio per colpa dell'hashmap.

5.4 problema piu greveeeeeeeeeeeeeeee







Si tiene un bst in ogni nodo, cosi da cercare il successor nel range in $\log n$ time, Si computa il successore per tutti i total overlapping node, e si uniscono prendendo il minimo, e query time $\Theta(\log^n n)$, mentre space usage $\Theta(n \log n)$, query time can be improved,

5.5 Triplets

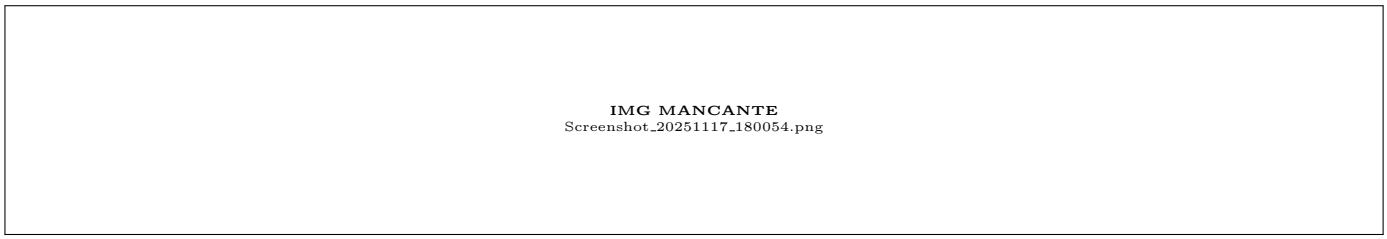
Usa segment tree o FT, per risolvere un problema, dato $A[0, n-1]$ di integer, e vogliamo contare nell'array il numero di triplìette (number of ways of 3 values),

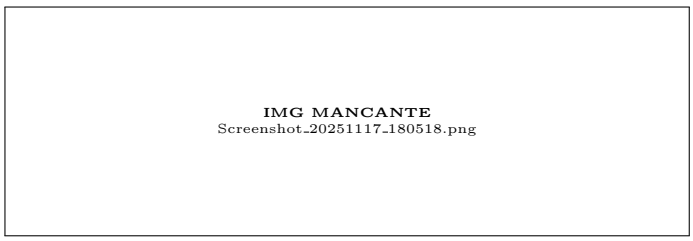
$$i, j, k \quad (0 \leq j < j < k < n)$$

such that

$$A[i] < A[j] < A[k]$$

Esempio:





Si processa l'array da sinistra a destra, e si contano il numero di occorrenze dei valori piu piccoli nel prefisso del FT, per 5 tipo c'è solamente un valore prima di 5, ovvero 3, mentre se la query fosse stata 7 allora sarebbe stato 4 la risposta, cosi si fa il k per ogni elemento godo, poi si fa questa cosa anche da destra a sinistra e si riconta.

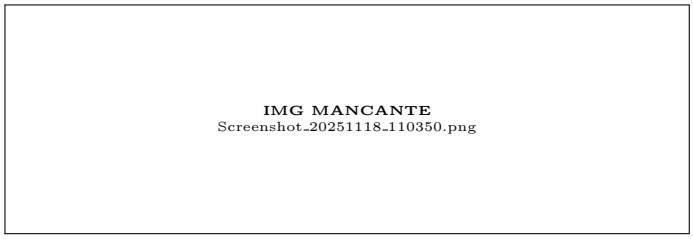
5.6 Range update with Lazy Propagation

Dato A[1,n]:

- Rangeupdate(i,j,v)
- Sum(i)

Ricordarsi che se non è sum ma è access possiamo usare FT con la soluzione gia vista, e comunque abbiamo visto anche per sum, ma è solo specifica per quello, mentre con ST possiamo fare molte piu operazioni.

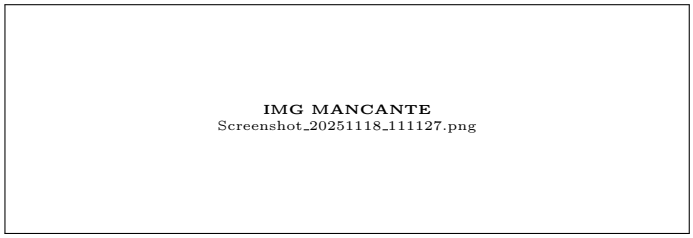
Soluzione: Si tengono due ST, che tiene l'operazione che vogliamo, quindi in questo caso la somma, ma se ne tiene un'altro il **Lazy Tree** dove ci mettimao gli update che ancora dobbiamo propagare, ovvero che facciamo, ma ancora non ci sono nell'altro ST.



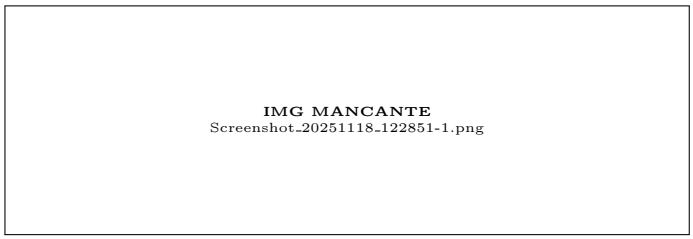
Facciamo la RangeUpdate(0,4,20), ogni volta che abbiamo una partial overlap si fa l'update dentro ST, si parte dalla root, abbiamo partial overlap, (il range è 0,4), quindi si aggiunge +100 alla root, questo perche abbiamo da fare l'update di 5 nodi, 20 * 5 = 100.

ricordiamo le regole:

- partial: si fa l'update su ST
- Total: si fa solamente l'update sul Lazy, solo se non è una leaf.

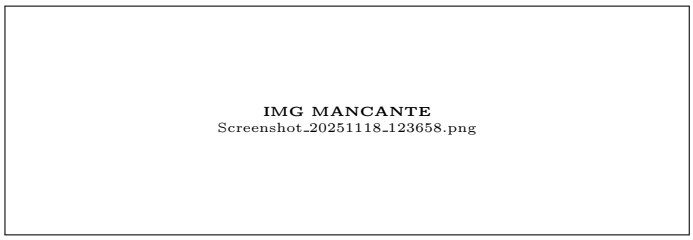


Ora dobbiamo scegliere tra destra e sinistra, a sinistra abbiamo total overlap, e destra abbiamo partial overlap, quindi sul nodo di sinistra si fa il +20 nel lazy (in realtà sarebbe 80) ma è implicito perche tanto va fatto anche nei nodi sotto, quindi ci ricordiamo solo il 20 nel lazy (secondo me uno fa confusione proofoofofofoofofo). Nel nodo di destra come detto abbiamo PO, quindi si fa +20, e questo perchè la porzione di intervallo è solamente di un nodo, quindi solamente +20.



Adesso sotto 21, abbiamo che 19 ha partial overlap, quindi si fa +20. Sotto 19 abbiamo 6 e 13, e 6 è un total overlap ma è una leaf quindi si scrive direttamente. A destra niente e anche a destra di quello sopra niente

Quale è il running time? come con sum, si toccano $\log n$ nodi, ma per ogni nodo quanto tempo? partial overlap è costant time, questo perche si toccano tutti i suoi nodi che servono, no overlap costant time, mentre total overlap? si mettono nel lazy tree quindi sempre costant time. Quidi $\Theta(\log n)$ time.

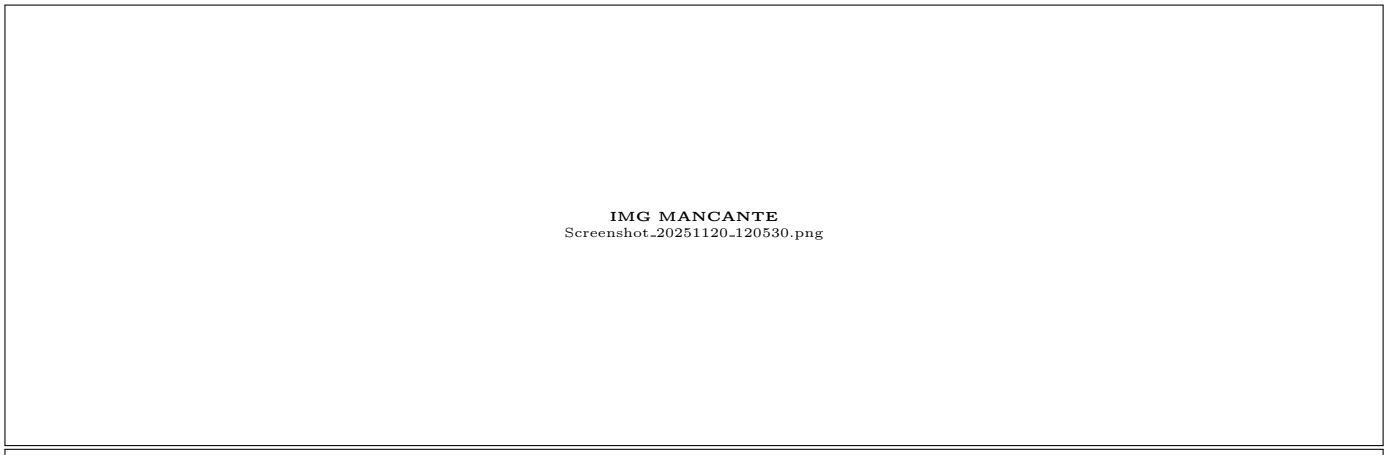


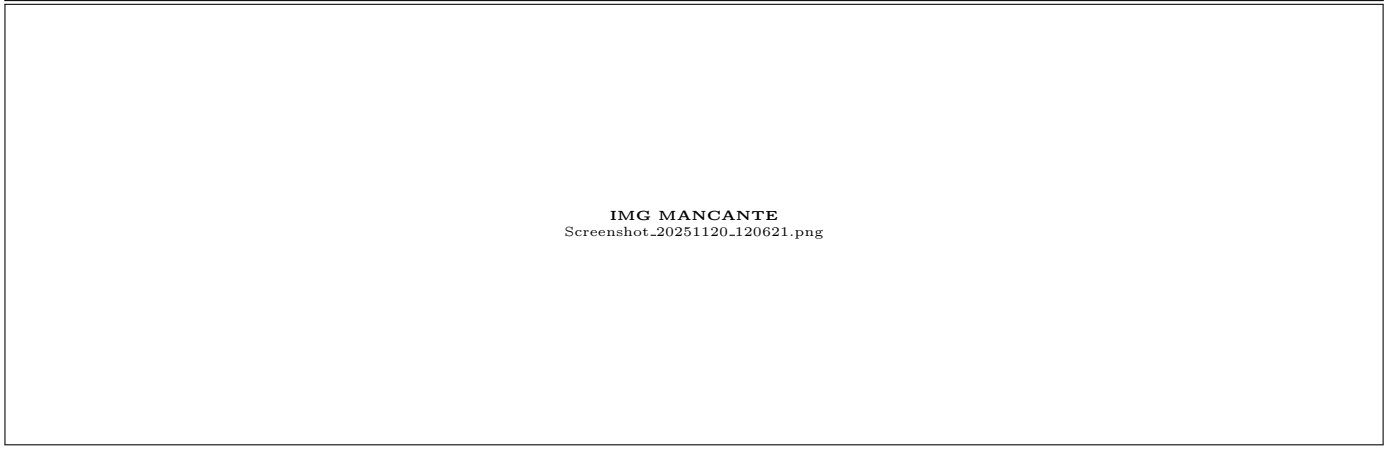
BELLISSIMOOOOOOOOOO

Ora come si fa la Sum? Sum(3) = RangeSum(0,3), si guarda per ogni nodo se nel lazy tree c'è un **pending update**, se c'è allora si aggiorna il nodo corrente che ci da la risposta, e si aggiunge nel lazy tree il pending update dei nodi sotto, cosi se mai dovessero servirci si facciamo l'update. Ora abbiamo la risposta alla nostra sum e via.

Quindi per un range update non si paga subito il costo ma si paga solo quando serve, cosi abbiamo tempo logaritmic. Anche la Sum va in $\Theta(\log n)$.

Altri appunti per cercare di capire meglio:





5.7 Persistent Segment Tree

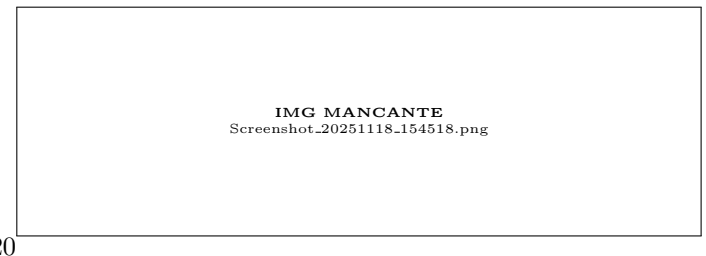
Vogliamo che la data structure sia persistent, ovvero abbiamo delle operazioni che fanno l'update della data structure, e certe che fanno le query, adesso però possiamo dare un timestamp con la query e vogliamo lo snapshot della struttura e via.

Dato $A[0, n-1]$

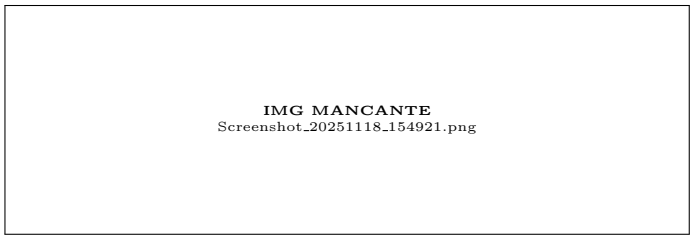
- Add(i,v)
- RangeSum(i,j,t) dove t è il timestamp

Facciamo piu soluzioni stupide:

- Facciamo una copia del ST dopo ogni update, pro: range sum in $\Theta(\log n)$ time ma cons: space usage $\Theta(n \cdot t)$ dopo t update, inoltre add operation prende $\Theta(n)$ time (perche si copia tutto il ST).
- Exploit: non ha senso copiare visto che non si cambia tutto, quindi abbiamo solamente n $\Theta(\log n)$ nodi dopo ogni update:



Si fa questa bella add, e vediamo che cambiamo questi nodi, allora per ogni nodo si tiene tutti i valori che hanno preso durante il tempo, quindi non ci teniamo solo un valore ma una lista di valori e di timestamp, in questo caso vediamo che 21 ha solo 21,0 visto che non è stato cambiato.



Adesso facciamo un'altra add, e vediamo che accanto 21 viene 41,2, visto che godo.

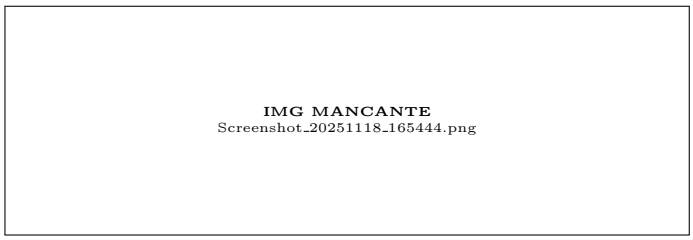
Pro: add prende $\Theta(\log n)$ e space $\Theta(n + t \cdot \log n)$ e range update prende

RangeSum(i,j,t) cerca per il predecessor di t in ogni lista dei Total OVerlapping Node, Time Complexity: $\Theta(\log n * \log t)$ ($\log t$ perche si fa una binary search dentro la lista)

Finalmente soluzione FIGA: Exploitiamo il fatto che tutti i $\log n$ nodi sono da una root-to-leaf path. Aggiungiamo un array. Si replica i nodi che cambiano e basta insomma.

in questo caso il nodo in posizione 3, ovvero 4 con 14, e dobbiamo cambiare anche il suo genitore, e si fanno proprio delle copie e si mettono al posto di quelli vecchi, ma comunque si utilizza l'array per puntare alla nuova root.

quindi è tutto come se fosse solamente un ST, semplicemente a seconda il timestamp cambia solamente lo starting point.



Se facciamo un'altra add semplicemente ci prendiamo i nodi sempre del timestamp diverso, e come si fa? il parent deve essere sempre the last one, quindi è easy. tanto per fare una ricerca si parte dalla root e quella si prende sempre con l'array.

- RangeSum prende $\Theta(\log n)$ time
- add prende $\Theta(\log n)$ time
- space usage prende $\Theta(n + t \log n)$ dopo t update

6 Mo's Algorithm

D'ora in poi si prenderanno un sacco di queryrange (ancora), Se è una sum query con update si puo' risolvere con un FT, mentre per le altre si possono risolvere con ST, ma deve essere possibili suddividere la query in sub query ed è cosi che si usa l'ST. MA NON PER TUTTEEEE

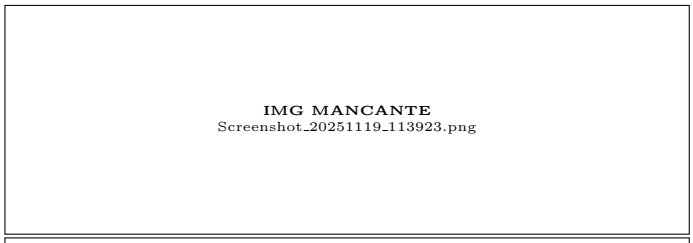
6.1 Problem meno difficult

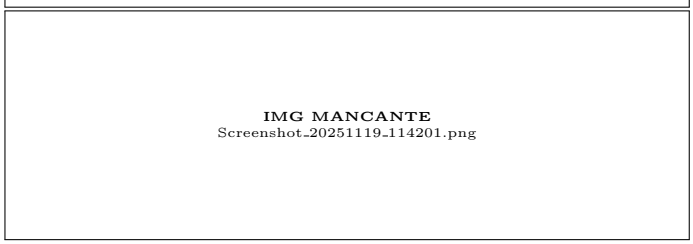
Dato $A[1, n]$ di colori (valori $\leq n$), e un set di Q di q queries (OFFLINE).

- 3 o piu (i,j): report # of distinct color that occur at least 3 times in $A[i, \ldots, j]$

Target time complexity: $\Theta((n+q)\sqrt{n})$ time

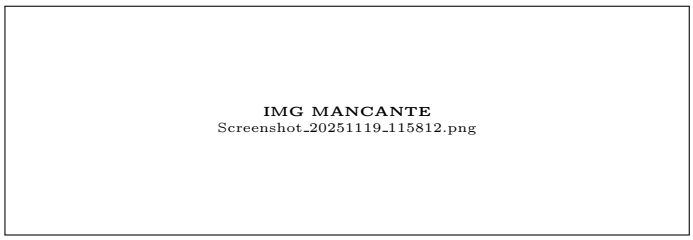
Facciamo una **trivial solution**: $\Theta(n \cdot q)$ time





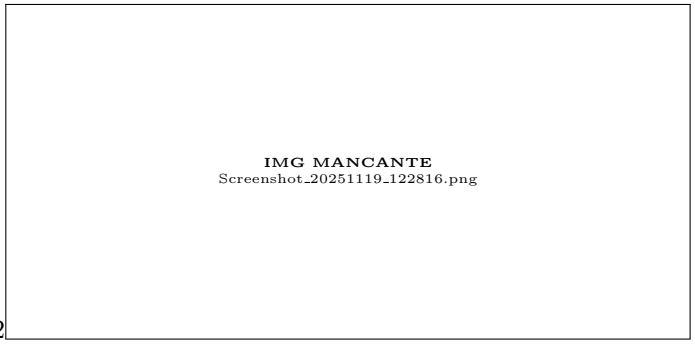
Si usa un array di counters per contare il numero di occurences

Adesso vorremmo usare la risposta alla prima query per dare una mano alla seconda, abbiamo un array di counters e abbiamo la variabile `Answer` che ci conta il numero di colori con almeno 3 occurences.



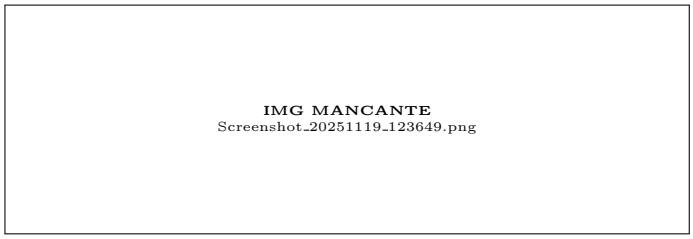
Abbiamo la seguene query, e sappiamo la sua risposta, se la prossima sarà un range diverso da quello della prima, si toglie semplicemente la parte che non ci interessa e via, ovviamente se non c'è overlap stiga

Soluzioe definitiva per questo tipo di trick:



Notare che abbiamo sempre: $\Theta(n \cdot q)$ time però l'ordine delle query serve, quindi che strategia si usa? vogliamo l'ordering per minimizzare il numero di elementi da togliere e da aggiungere, ma ancora non andrebbe a ridurre il running time.

Notare: esiste un ordering delle queries che permette un time complexity di $\Theta((n+q)\sqrt{n})$.

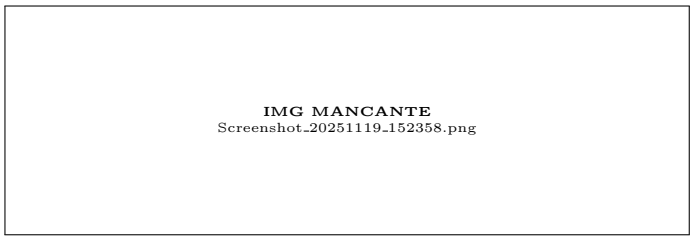


Si raggruppa le query dal loro left extreme (`cur_i`), in gruppi di size $\sqrt{n}$. (si fa il sort (i,j) by ($i/\sqrt{n}$,j))

In ogni gruppo si fa il sort del right extreme (`cur_j`), e si vanno a risolvere le queries.

Siamo in un gruppo abbiamo cur_i e curr_j, consideriamo `cur_j`, che alle brutte dentro il gruppo può arrivare fino a n. Quindi per ogni gruppo al massimo facciamo n "move", ovvero dobbiamo fare l'add n volte.

quindi overall $\Theta(n\sqrt{n})$ moves (o calls to add).



Invece quante moves facciamo in base a `cur_i`? siamo nella situazione dell'immagine, ovvero si distano al massimo di $\sqrt{n}$, quindi se nel gruppo ci sono b_i queries, allora si fanno al massimo $\Theta(b_i \cdot \sqrt{n})$ moves per gruppo i

Quinid quante in tutto? $\Theta(\sum_{i=1}^{\sqrt{n}} b_i \cdot \sqrt{n}) = \Theta(\sqrt{\cdot}q)$

Quindi in tutto $\Theta(n \cdot \sqrt{n} + q \cdot \sqrt{n}) = \Theta((n+q)\sqrt{n})$

6.2 A Difficult Problem

Dato un array A[0,n-1],

- Answer q queries (OFFLINE)

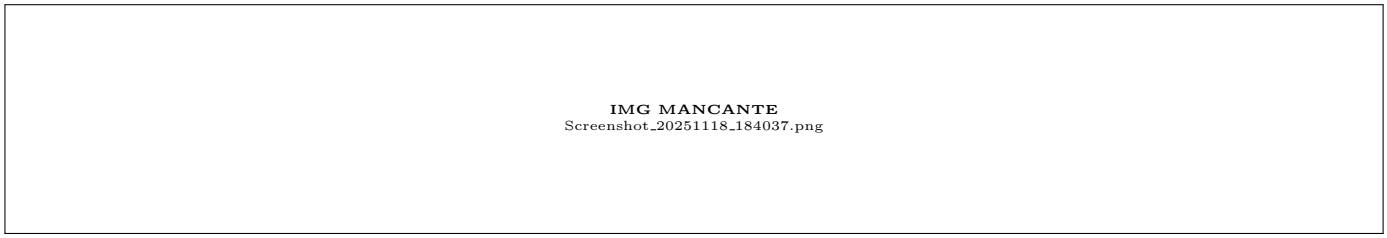
Online ovvero fare q queries una dopo l'altra e una risposta dopo l'altra, mentre OFFLINE abbiamo tutte le query in una volta e si deve dare la risposta a tutte le query insieme.

- Query(l,r) = $\sum_{e \in D(l,r)} e \cdot Ke^2$

D(l,r) = set of distinct element in $A[l \dots r]$,

Ke = numero di occurences di e dentro $A[l \dots r]$

quindi per ogni distinct element nel range dobbiamo computare quel lavoro, Ke è il numero di occurence dei distinct elemetn

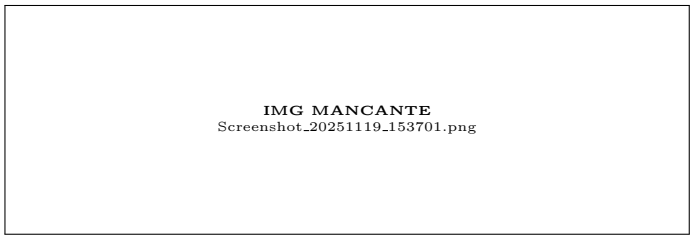


Target time complexity: $\Theta((n+q)\sqrt{n})$ time

Ora discutiamo della versione statica del problema.

(ORA SAPPIAMO COME FARE LA PARTE SOPRA ERA PRIMA GASSE)

Si fa come prima, ma semplicemente cambia la contribuzione all'interno di answer:



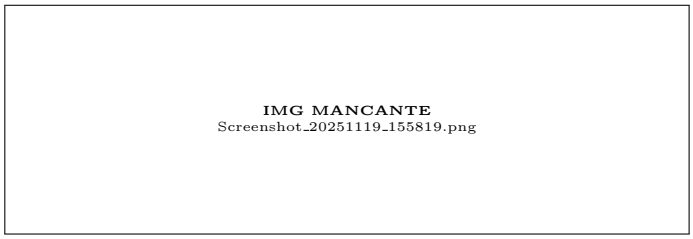
7 (STATIC) RMQ

Abbiamo fatto prefix sum static, che si risolve con un array, prefix sum dynamic che si risolve con FT, altre range queries che si risolvono con un ST, inoltre abbiamo fatto anche cose piu carine offline con mos algoritmh.

Range Minimum Queries, sono piu difficili di prefix sum, nel caso **dinamico** si possono risolvere con ST con tempo logaritmico, ma nel caso statico si possono risolvere in maniera lineare.

Dato $A[0, n-1]$

- RMQ(i,j) = argmin$(A[i \dots j])$



Target $\Theta(1)$ time e $\Theta(n)$ space (ST fa log n per query)

Prima soluzione brutta: $\Theta(1)$ time e $\Theta(n^2)$ space, si fa una tabella con tutte le possibili risposte e via gasse.

Seconda soluzione brutta: sparse array, $\Theta(1)$ time e $\Theta(n \log n)$ space, si fa l'exploit del fatto che due range che si overlappano hanno la risposta di una query che li comprende entrambi, basta fare il minimo tra i due.

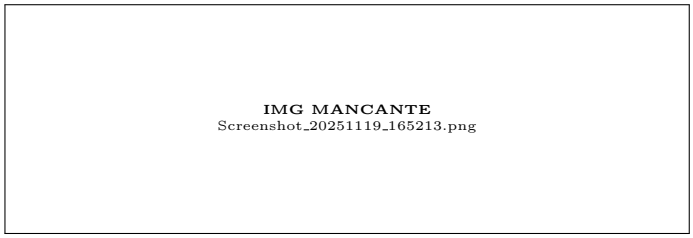
IMG MANCANTE
Screenshot_20251119_160356.png

Facciamo una tabella piu piccola, si tengono le risposte per $[i, i + 2^k]$ per ogni k, per la query si fa:

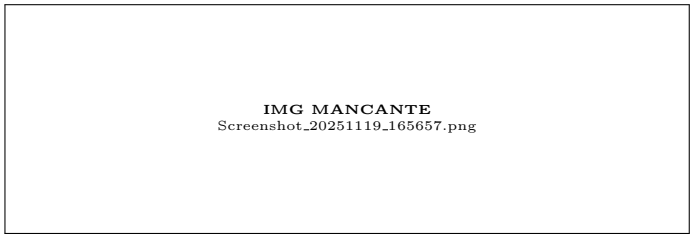
IMG MANCANTE
Screenshot_20251119_164341.png

IMG MANCANTE
Screenshot_20251119_164722.png

Soluzione 3: $\Theta(\log n)$ time e $\Theta(n)$ space



Creiamo dei blocchi dentro A lunghi $\log n$ e poi si tiene A' dove teniamo l'elemento minimo di ogni blocco, ovviamente A' è grande $\frac{n}{\log n}$ e si usa la soluzione di prima



questo è il modo per risolvere le query, cosi abbiamo tempo costante per il total overlap, mentre per calcolare il prefix (verde) e il suffix (rosso) della query che non sono direttamente su un blocco di A' si fa semplicemente uno scan e questo costa al massimo $\log n$ e infatti si godeeeeeeeeeee

ULTIMA SOLUZIONE: $\Theta(1)$ time e $\Theta(n)$ space Solo l'intuizione

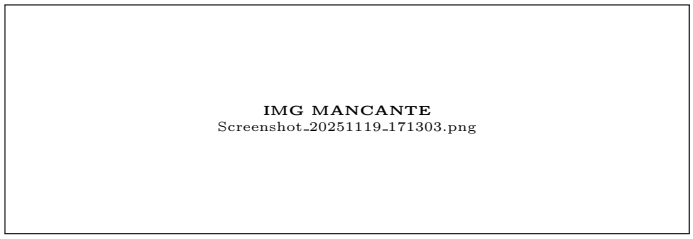
7.0.1 Cartesian Tree

2 regole:

- Ogni nodo ha sotto sempre uguale o maggiore,
- La posizione dei nodi nell'albero deve rispettare l'ordine originale dell'array.

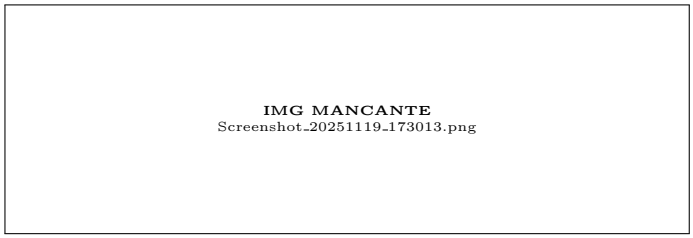
Se ci sono duplicati, allora si prende quelo piu a sinistra.

IDEA HO CAPITO, si prende sempre il minimo e da li si splitta in due, e cosi via si prende sempre il minimo da sinistra, godooooooo

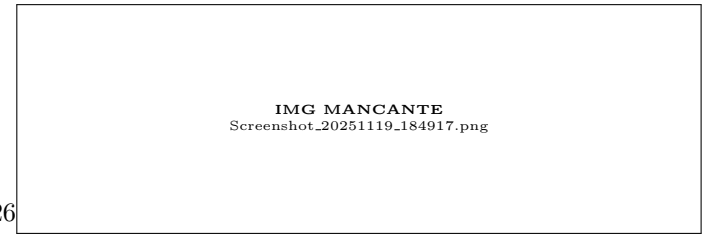


Si crea l'albero on top di A, ci mettiamo dentro il minimo ovviamente, la root è l'elemento minimo, si splitta l'array in due parti.

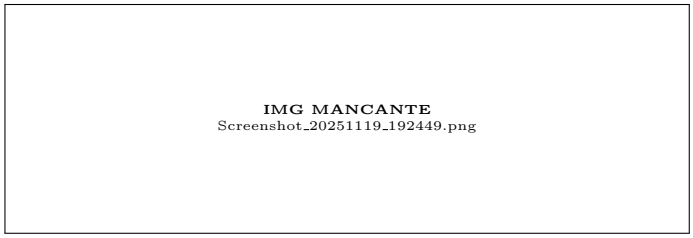
Il valore piu piccolo è 1, per cui è la nostra root ovviamente quello piu a sinistra, allora dopo c'è 2, e sempre uno a destra come figlio, successivamente il piccolo a destra è sempre 1 e cosi via fino infondo.



Quindi in questo caso RMQ(4,8) = 1, perche la risposta alla query è sempre la root di quel sub tree, ovvero LCA(4,8) lower common ancestor



Quindi basta supportare LCA per rispondere alla query, ma noi abbiamo dei blocchi piccoli, si usa le balance parantesis, quando si trova un nodo per la prima volta si mette un parantesi aperti, quando si smette di visitarlo, quindi non ci passaremo piu, ne metteremo una chiusa, in tutto sono `2n bits`



In questa maniera possiamo ricostruire tutto il tree, GODOOOOO, la prima parentesi è la root, e cosi via si ricostruisce, ma quindi servono `4n bit` perche dobbiamo anche capire se il figlio è a destra o a sinistra.

recap: come si fa questa cosa? si rappresenta ogni blocco dentro un cartesian tree e si trova il LCA con questi bit strani.

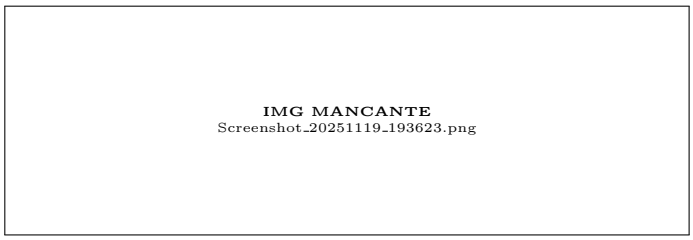
ROBA TEORICA NON TRIVIAL COME DICE IL PROF: Calcoliamo il tempo, mettiamo che stavolta si fanno i blocchi di $\frac{1}{8}\log n$ bits quindi il cartesian tree si rappresenta con $4 \cdot \frac{1}{8}\log n = \frac{1}{2}\log n$ bits. Quindi abbiamo $2^{\frac{1}{2}\log n}$ possible tree rappresentation ma questo è $2^{\frac{1}{2}\log n} = \sqrt{n}$.

Quindi insomma si fanno tutti i possibili tree (? TODO: chiedere) e abbiamo sempre la risposta in costant time.

7.1 Colored Range Query Problem

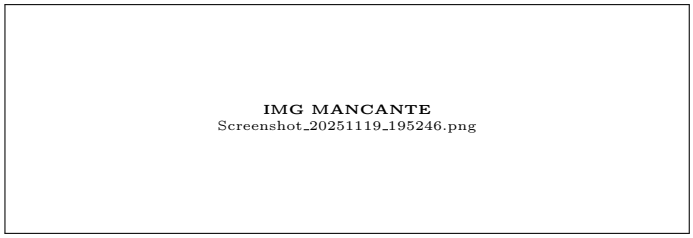
Dato $A[1, n]$ di colori (integer $\leq n$)

- Distinct (i,j) ritorna all distinct color in $A[i, j]$ in tempo proporzionale a $k = \#$ di distinct color in $A[i, j]$

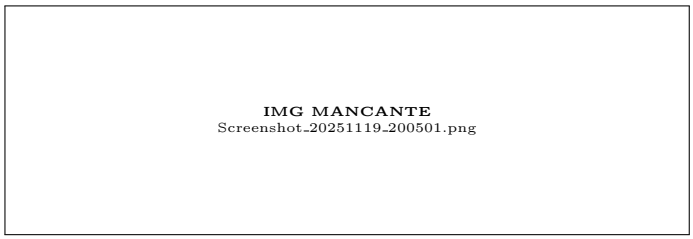


Prima soluzione: Segment tree con hashset, $\Theta(k \cdot \log n)$ e space $\Theta(n \log n)$

Bella soluzione: Space $\Theta(n)$ e query time $\Theta(k)$ quindi ottimaliiii



Abbiamo il nostro array, e per ogni valore: se è la prima occurence si connette al "PUNTO ROSSO" e se invece si trova una occurence si connette alla occurence prima, Poi ci teniamo un array dei pointer semplicemente, cosi sappiamo a chi si chiedeee Abbiamo il nostro query range che va a 3 a 6, e vediamo che ogni distinct element ha **esattamente una freccia che crossa la linea blu**, grazie punto rosso.



Quindi sapendo questa cosa, basta usare l'array P di pointer e cerchiamo in maniera ricorsiva i numeri piu piccoli del min della range query. E ci si ferma quando si trova numeri piu grossi di 3, perche tanto no freca, particamente si fanno RMQ nell'array P e ricorsivamente si crea il cartesian tree dei range, e non si va ricorsivamente giu, ma si va ricorsivamente a destra e a sinistra, e si fa solamente la ricorsione per elementi minori di 3, se ne perde 2 piu o meno per ognuno quindi siamo sempre nell'ordine di k.

Bella merda ho scritto ma si capisce dai su.