

Competitive Programming

Simone Stanganini

December 1, 2025

Contents

1	100 prisoners	3
2	Super clever algoritm	4
2.1	Duplicate Eleemnts in an array	4
2.1.1	log n passes	5
2.1.2	Destroy A	5
2.2	Floyds cycle finding	6
2.3	Majority Element	6
2.3.1	Boyer Moore Algorithm	7
2.3.2	Support Insertion/Deletion	7
2.4	Misra-Gries Heavy Hitter	9
3	Alice and bob	9
4	Fenwick Tree binary indexed tree, BIT	10
4.1	Dynamic Prefix Sums	10
4.2	Counting Inversion	11
4.3	Nested Segment	12
4.4	Update the Array	13
4.5	Dynamic Prefix Syms with range update	14
5	Segment Tree	15
5.1	BMQ with occurrency	16
5.2	problema piu greve	16
5.3	problema piu greveeeeeee	17
5.4	problema piu greveeeeeeeeeeeeeeee	17
5.5	Triplets	18

5.6	Range update with Lazy Propagation	18
5.7	Persistent Segment Tree	20
6	Mo's Algorithm	22
6.1	Problem meno difficult	22
6.2	A Difficult Problem	23
7	(STATIC) RMQ	24
7.0.1	Cartesian Tree	26
7.1	Colored Range Query Problem	28
7.2	Range Minimum Querys	28
8	Dynamic Programming	29
8.1	Fibonacci Numbers	29
8.2	Matrix Exponentiation	29
8.3	Rod Cutting	30
8.4	facebook: Robber	30
8.4.1	shortest/longest path computation on a DAG	30
8.4.2	Road cutting: ma con il DAG	31
8.4.3	Torniamo a faceboooooook	32
8.5	Snake and ladders	32
8.6	Minimum Cost Path	33
8.7	Longest Common Subsequence LCS	33
8.8	0/1 knapsack problem	34
8.9	Fractional knapsack	35
8.10	Puzzle exercise	35
8.11	Longest Increasing Subsequence (LIS)	36

1 100 prisoners

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

IMG MANCANTE
Screenshot_20251110_094931.png

C'è una stanza con 100 cassetti, ognuno contiene un numero ma non si conosce, ogni scienziato entra e può aprirne 50. Gli scienziati 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, coglioniiii.

Si usa una strategia adaptive, per aprire i drawer.

1. Scienziato 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 così.

IMG MANCANTE
Screenshot_20251110_103613.png

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

IMG MANCANTE
Screenshot_20251110_111744.png

IMG MANCANTE
Screenshot_20251110_112458.png

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 già, e poi gli elementi che mancano:

IMG MANCANTE
Screenshot_20251110_113544.png

quindi come si calcola? si fa:

IMG MANCANTE
Screenshot_20251110_113709.png

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

2 Super clever algoritmo

2.1 Duplicate Eleemnts in an array

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

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

IMG MANCANTE
Screenshot_20251110_114419.png

Complessità, $\Theta(n)$ tempo, ma anche $\Theta(n)$ space, quindi dobbiamo sistemarlo. Sembra impossibile perché già 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.

IMG MANCANTE
Screenshot_20251110_115155.png

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

IMG MANCANTE
Screenshot_20251110_115344.png

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

2.1.2 Destroy A

Ma se potessimo fare random access possiamo andare più 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.

IMG MANCANTE
Screenshot_20251110_120010.png

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, così da certificare il fatto che il ciclo esiste (lo sappiamo per cui secederà).
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 perché S fa m steps, mentre F fa $2m$ steps, si chiama a la lunghezza del prefisso prima dell'inizio del ciclo, mentre b quanto sono dentro al loop, per cui:



$$m = a + b$$

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

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 iterazione 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 rappresentazione 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 occorre almeno $\frac{n}{T} + 1$ volte, stiamo generalizzando il problema majority, con $T = 2$ è quello.

Si tiene un set K di T candidati e counters,



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 così così 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.

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

$$\text{RangeSum}(i, j) = \text{Sum}(j) - \text{Sum}(i - 1)$$

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:



Qui creiamo un albero per la prefix sum, solamente per le potenze di due, così 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.



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



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, ugualmente 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, così si capisce meglio.



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 iniziale, 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.



IMG MANCANTE
Screenshot_20251114_144043.png



IMG MANCANTE
Screenshot_20251114_144527.png

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

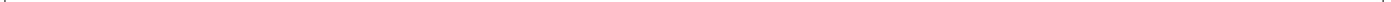


IMG MANCANTE
Screenshot_20251114_145327.png

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.



IMG MANCANTE
Screenshot_20251114_150246.png



IMG MANCANTE
Screenshot_20251114_150327.png

4.2 Counting Inversion

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

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

IMG MANCANTE
Screenshot_20251115_152154.png

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).

IMG MANCANTE
Screenshot_20251115_153936.png

FT based: si costruisce il FenwickTree dove in ogni posizione $F[j]$ ha la frequenza dell'elemento 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.

IMG MANCANTE
Screenshot_20251115_154448.png

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 sostituiscono i valori con il loro rank, tnato ci importa delle inversioni non dei valori effettivi, godo.

IMG MANCANTE
Screenshot_20251115_154532.png

4.3 Nested Segment

IMG MANCANTE
Screenshot_20251115_155008.png

IMG MANCANTE
Screenshot_20251115_161220.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?

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 così si toglie per gli altri, così si tratta sempre il segmento corrente come l'ultimo gasseeeeeeeeeeeeeeeeeeeee.

4.4 Update the Array

Dato un array di soli 0 vogliamo supportare 2 operazioni:

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



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 . Così 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 $j + 1$, così da cancellare quella propagazione



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



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 così 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.

IMG MANCANTE
Screenshot_20251117_104717.png

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

IMG MANCANTE
Screenshot_20251117_110140.png

5 Segment Tree

Stesso problema del FT, sempre la add ma un problema diverso per la range sum è più 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 descendente 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 già fatto

Quindi è linear o logaritmic? è linear nel nostro esempio, per cui come si fa? con questi partial overlaps diventa linear a forza di andare a sinistra e destra. Non ho capito però i discorsi del prof, e il running time è $\log n$ insomma, perché 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 perché lo ha spiegatooooooooooooooooooooo

5.2 problema più 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 $\sum_{j=1}^i A[j] \geq s$



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 perché 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 lì dentro, 6 è più piccolo di 15, allora si cerca 15-6 e infatti 13 è più grande allora gasse.

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

5.3 problema piu greveeeeeee

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



IMG MANCANTE
Screenshot_20251117_175458.png

Si tiene un bst in ogni nodo, così da cercare il successore 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 triplettie (number of ways of 3 values),

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

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 più 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, così 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 già vista, e comunque abbiamo visto anche per sum, ma è solo specifica per quello, mentre con ST possiamo fare molte più 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 mettiamo 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 perché 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 perché tanto va fatto anche nei nodi sotto, quindi ci ricordiamo solo il 20 nel lazy (secondo me uno fa confusione proofofofofofo). Nel nodo di destra come detto abbiamo PO, quindi si fa +20, e questo perché 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 perché 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? $\text{Sum}(3) = \text{RangeSum}(0,3)$, 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 dà la risposta, e si aggiunge nel lazy tree il pending update dei nodi sotto, così 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, così abbiamo tempo logaritmico. 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:

IMG MANCANTE
Screenshot_20251118_154518.png

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 val-



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.



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



ore 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.

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.

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 così 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 $A[i, \dots, j]$

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 occurrences

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 occurrences.

Abbiamo la seguente 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

Soluzione definitiva per questo tipo di trick:

22

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

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

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.

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

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

$K_e = \text{numero di occurences di } e \text{ dentro } A[l \dots r]$

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

IMG MANCANTE
Screenshot_20251118_184037.png

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:

IMG MANCANTE
Screenshot_20251119_153625.png

IMG MANCANTE
Screenshot_20251119_153701.png

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 algoritmhi.

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 $A[0, n - 1]$

- $\text{RMQ}(i, j) = \arg\min(A[i \dots j])$

IMG MANCANTE
Screenshot_20251119_155819.png

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, comprende entrambi, basta fare il minimo tra i due. si fa l'exploit del fatto che due range che si sovrappongono hanno la risposta di una query che li

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, così 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 godeeeeeeeeeeee

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 quello più a sinistra.

IDEA HO CAPITO, si prende sempre il minimo e da lì si splitta in due, e così via si prende sempre il minimo da sinistra, godoooooooo



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 più piccolo è 1, per cui è la nostra root ovviamente quello più a sinistra, allora dopo c'è 2, e sempre uno a destra come figlio, successivamente il piccolo a destra è sempre 1 e così via fino in fondo.

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



Quindi basta supportare LCA per rispondere alla query, ma noi abbiamo dei blocchi piccoli, si usa le balance parenthesis, quando si trova un nodo per la prima volta si mette



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 representation 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.

un parantesi aperti, quando si smette di visitarlo, quindi non ci passaremo più, 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 così via si ricostruisce, ma quindi servono **4n bit** perché dobbiamo anche capire se il figlio è a destra o a sinistra.

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 $A[i, j]$



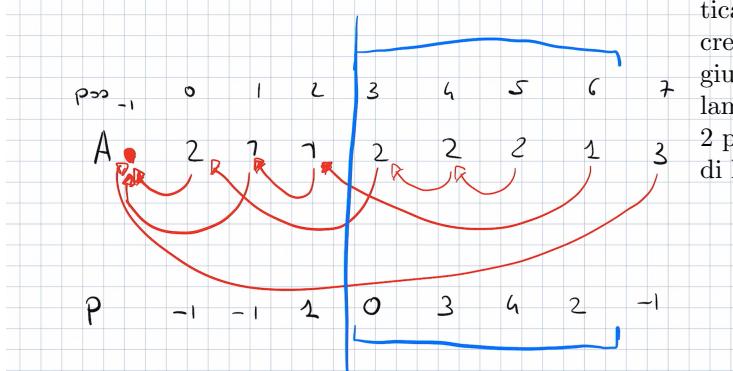
Prima soluzione: Segment tree con hashset, $\Theta(k \cdot \log n)$ e space $\Theta(n \log n)$

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, così sappiamo a chi si chiede. 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.

Solution space: $\Theta(n)$
query time : $\Theta(k)$

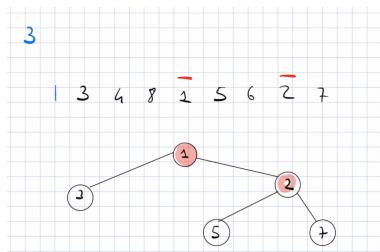


Quindi sapendo questa cosa, basta usare l'array P di pointer e cerchiamo in maniera ricorsiva i numeri più piccoli del min della range query. E ci si ferma quando si trova numeri più grossi di 3, perché tanto no freca, particolarmente si fanno RMQ nell'array P e ricorsivamente si crea il cartesian tree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

Bella merda ho scritto ma si capisce dai su.

7.2 Range Minimum Querys

Abbiamo un array e si chiede di reportare gli elementi più piccoli di un certo k :



Abbiamo l'array e sicuro dobbiamo riportare 1 e 2, si inizia e si chiede il minimo di tutto l'array, allora si fa una ricorsione a sinistra e destra e si continua così a chiedere il minimo, e ci si ferma ovviamente quando il minimo è più grande o uguale a k .

Quanto costa? si sta costruendo un cartesian tree quindi comunque $O(k)$

8 Dynamic Programming

8.1 Fibonacci Numbers

Riusiamo tutto quello che abbiamo già fatto godendo nel mentre.

```

 $F_0 = 0$ 
 $F_1 = 1$ 
 $F_n = F_{n-1} + F_{n-2}$ 

def Fib(n):
    if n == 0
        return 0
    if n == 1
        return 1
    return Fib(n-1) + Fib(n-2)

```

Si fa schifo questo algoritmo, ci teniamo le risposte in una tabella e si risponde continuamente.

```

M = [None] * 10
def FibMemo(n, M):
    if n == 0
        return 0
    if n == 1
        return 1
    if M[n] == None:
        M[n] = FibMemo(n-1, M) + FibMemo(n-2, M)
    return M[n]

```

TOP DOWN APPROAC: Ci teniamo la tabella M inizializzata a 0.

Possiamo anche avere un bottom down approach cercando di fare un algoritmo più veloce:

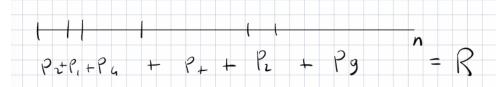
8.2 Matrix Exponentiation

Dobbiamo fare A^n , si fa divide and conquer, sappiamo che:

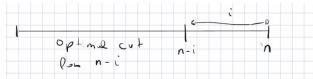
$$\begin{aligned} \text{How to compute } A^n? \\ \text{n even} & \quad A^n = A^{\frac{n}{2}} \cdot A^{\frac{n}{2}} \\ \text{n odd} & \quad A^n = A^{\frac{n}{2}} \cdot A^{\frac{n}{2}} \cdot A \end{aligned}$$

8.3 Rod Cutting

Abbiamo una barra di acciaio, di lunghezza n per un taglio di lunghezza $i \in [1, n]$ si prende P_i euros, e vogliamo massimizzare il guadagno, ovvero trovare i cuts che massimizzano la revenue totale.



Soluzione: quali sono i sub problems? solamente smaller input, quindi qui è : Optimal way to cut a rod of length K i n.

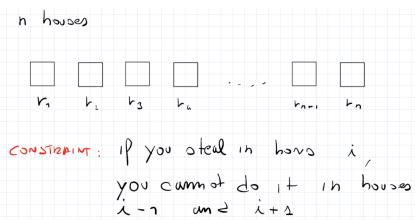


Combine: $R(n) = \max_{1 \leq i < n} \{R(n - 1) + P_i\}$ ovvero facciamo il sub problem e andiamo avanti restringendo i .

```
def rod(n):
    R = [0] * (n+1)
    for i in range(n+1):
        for l in range(i):
            R[i] = max(R[i], R[i-l] + P[l])
    O(n^2) time
```

Prima si inizializza la tabella dei total revenue, poi per ogni i a $n+1$ e per ogni l in che va fino ad i , si prende il massimo tra quello che c'è ora e toglie tutti gli i . $R[i-l] + P[l]$ ovvero tutti i possibili tagli da fare in quel pezzo di i .

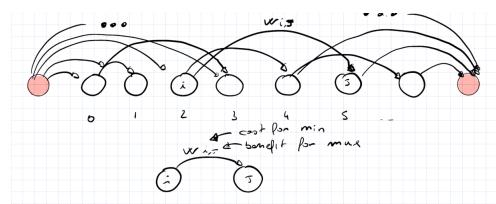
8.4 facebook: Robber



Ci serve un tool:

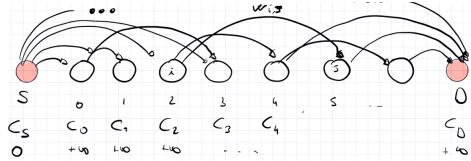
8.4.1 shortest/longest path computation on a DAG

Si definiscono dei nodi speciali, **Source** e **Destination**, si attacca a tutti la source e arrivano tutti alla destination, abbiamo la weight $w_{i,j}$ (il costo) e quindi vogliamo trovare il path dalla source alla destination con il massimo dei pesi oppure il minimo non importa.



Si usa l'algoritmo di dijkistra, abbiamo un heap e ci si mette U nodi dentro con il loro costo, e si estrae quello con costo minore, e si rilassano tutti gli edges che escono da quello, non ci ho capito una sega. costo numero di nodi + numero di archi per log n visto che si parla di heap.

Ma c'è un'altro modo, senza heap, perché se si processano i nodi fino ad arrivare al nodo i in ordine, gli altri non si cagano, questo perché in un DAG tutti gli archi che arrivano ad i si sono già processati, mentre quelli a destra di i ancora non si sono cagati, allora cosa si fa?



Si tiene il costo dello shortest path per arrivare a quel nodo c_i , inizialmente sono tutti ∞ , e step by step si computa lo shortest past da sinistra a destra. Per ogni vertex che si processa, si rilassano tutti i vertici.

Ovvero se i, j è un edge, allora il costo per andare a j , diventa il minimo tra il costo di c_j e $c_i + w_{i,j}$, e da sinistra a destra si fa questo calcolo e visto che le weight sono sempre positive, gasse. Traduzione: "Al nodo j conviene tenersi il costo che aveva prima, o arrivarci passando da me (i)?"

8.4.2 Road cutting: ma con il DAG

Si fa il dag con un nodo per ogni cut, e tutti i nodi sono connessi a tutti quelli dopo, e come revenue il guadagno da 1 a i è P_{i-1}

Di nuovo numero di edge più numero di vertex, quindi sempre quadratico in questo coso brutto.

Esempio da gemini:

Il problema consiste nel massimizzare il profitto tagliando un'asta di lunghezza $N = 4$. Modelliamo il problema come la ricerca del cammino a costo massimo (Longest Path) su un DAG, dove i nodi rappresentano i punti di taglio ($0 \rightarrow 4$) e gli archi rappresentano i pezzi venduti.

Dati di Input I prezzi per i tagli sono:

- Taglio da 1m: 1€
- Taglio da 2m: 5€ (Pezzo pregiato)
- Taglio da 3m: 8€
- Taglio da 4m: 9€

Logica dell'Algoritmo Poiché il grafo è un DAG, processiamo i nodi linearmente da sinistra a destra (da 0 a 4). Utilizziamo la formula di rilassamento per massimizzare il profitto:

$$DP[j] = \max(DP[j], DP[i] + P_{(j-i)})$$

In pratica, per ogni nodo, controlliamo se conviene arrivare lì estendendo un percorso precedente.

Esecuzione Step-by-Step

1. Inizializzazione al Nodo 0 (Source)

L'array dei profitti parte vuoto (0 a $-\infty$), eccetto la sorgente. Dal nodo 0 lanciamo gli archi verso tutti i nodi futuri:

- Verso 1: $0 + 1 = 1$
- Verso 2: $0 + 5 = 5$
- Verso 3: $0 + 8 = 8$
- Verso 4: $0 + 9 = 9$

Stato Array: [0, 1, 5, 8, 9]

2. Processiamo il Nodo 1 (Profitto attuale: 1)

Proviamo ad estendere il percorso partendo dal nodo 1:

- Verso 2: $1 + 1 = 2$. (Minore di 5, ignoro)
- Verso 3: $1 + 5 = 6$. (Minore di 8, ignoro)
- Verso 4: $1 + 8 = 9$. (Uguale a 9, ignoro)

Nessun miglioramento.

3. Processiamo il Nodo 2 (Profitto attuale: 5)

Qui avviene il rilassamento cruciale (il sorpasso):

- Verso 3: $5 + 1 = 6$. (Minore di 8, ignoro)
- Verso 4: $5 + 5 = 10$. (Maggiore di 9! **Aggiorno**)

Stato Array: [0, 1, 5, 8, 10]

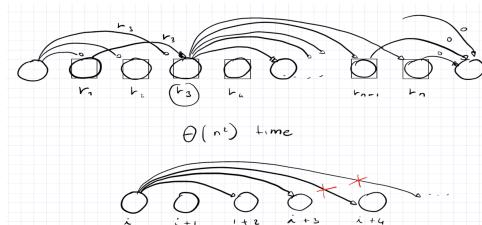
Conclusione

Arrivati alla fine, il profitto massimo accumulato al nodo 4 è **10€**.

Il percorso vincente sul DAG è: $0 \xrightarrow{+5} 2 \xrightarrow{+5} 4$, che corrisponde a due tagli da 2 metri ciascuno.

8.4.3 Torniamo a faceboooooook

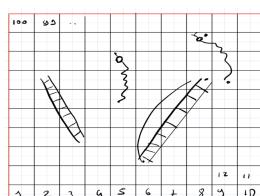
Come si fa con il dag? Se abbiamo un arco che arriva vuol dire che si deve rubare, ma non si puo andare nella casa dopo. sempre $\Theta(n^2)$ time



Consideriamo anche che possiamo togliere degli edge, questo perche non ha senso andare da i a $i + 4$, perche possiamo prima andare a $i + 2$ e poi si va a $i + 4$ in questa maniera abbiamo solamente 2 archi che escono da ogni nodo e quindi $\Theta(n)$

8.5 Snake and ladders

Abbiamo una matrice, si inizia dalla cella 1 e dobbiamo raggiungere 100, ci si muove con un dado, e ci sono snake che ti riportano indietro e ladder che ti portano avanti, trovare il minimo numero di tirate di dado per arrivare in fondo.

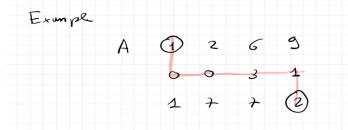


Si trasforma ogni cella in un nodo, e per ogni nodo abbiamo 6 archi (le facce del dado), e se si va su un ladder si va al suo nodo e via, e si computa lo shortest path, e via.

Running Time: $\Theta(n)$

8.6 Minimum Cost Path

A è una $n * m$ matrice con degli interi, l'obiettivo è computare il path di costo minimo dal top-left angolo al bottom right muovendoci solamente in basso e a destra.



8.7 Longest Common Subsequence LCS

Dato S_1 ($|S_1| = n$) e S_2 ($|S_2| = m$) trovare $LCS(S_1, S_2)$,

Given S_1 ($|S_1| = n$) and S_2 ($|S_2| = m$)
Find $LCS(S_1, S_2)$
 $S_1: A \text{ } B \text{ } C \text{ } B \text{ } D \text{ } A \text{ } B \quad n=7$
 $S_2: B \text{ } D \text{ } C \text{ } A \text{ } B \text{ } A \quad m=6$

Vogliamo $\Theta(n \cdot m)$ time.

$LCS(i, j)$ to denote the length of LCS between
 $S_1[1 \dots i]$ and $S_2[1 \dots j]$

Combine

$LCS(i, j) =$

CASE #1
$c_i = c_j$
$LCS(i, j) = 1 + LCS(i-1, j-1)$

CASE #2
$c_i \neq c_j$
$\max \{ LCS(i, j-1), LCS(i-1, j), LCS(i-1, j-1) \}$

Dobbiamo trovare il sub problema ovvero prendere il i prefissi delle due stringhe. E dobbiamo trovare come combinarli, ovvero aggiungere due simboli alle stringhe già processati, ovvero $c_1 == c_2$, se sono uguali gasse, se no stica.

Recurrence

$$LCS(i, j) = \begin{cases} 0 & \text{if } i=0 \text{ or } j=0 \\ 1 + LCS(i-1, j-1) & \text{if } S[i] == S[j] \\ \max \{ LCS(i, j-1), LCS(i-1, j), LCS(i-1, j-1) \} & \text{otherwise} \end{cases}$$

se sono lunghe 0 allora 0, se sono uguali allora gasse, e per gli altri casi uguali.

Ma questo approccio è top down, dobbiamo fare la ricorsione. mentre vorremmo buildare la tabella dall'inizio e pienare la tabella per riga e non per colonna.

Bottom-up Approach							
	1	2	3	4	5	6	
0	∅	B	BC	C	CA	CB	A
1	A						
2	B						
3	C						
4	B						
5	D						
6	A						
7	B						

$$\text{LCS}(i, j) = \begin{cases} 0 & \text{if } i = 0 \text{ or } j = 0 \\ 1 + \text{LCS}(i-1, j-1) & \text{if } S[i] = S[j] \\ \max\{\text{LCS}(i-1, j), \text{LCS}(i, j-1)\} & \text{otherwise} \end{cases}$$

Bottom-up Approach							
	1	2	3	4	5	6	
0	∅	B	BC	C	CA	CB	A
1	A	0	0	0	0	0	0
2	B	0	0	0	0	0	0
3	C	0	0	0	0	0	0
4	B	0	0	0	0	0	0
5	D	0	0	0	0	0	0
6	A	0	0	0	0	0	0
7	B	0	0	0	0	0	0

$$\text{LCS}(i, j) = \begin{cases} 0 & \text{if } i = 0 \text{ or } j = 0 \\ 1 + \text{LCS}(i-1, j-1) & \text{if } S[i] = S[j] \\ \max\{\text{LCS}(i-1, j), \text{LCS}(i, j-1)\} & \text{otherwise} \end{cases}$$

8.8 0/1 knapsack problem

0 si prende l'item e 1 no, e si deve prendere più item possibili per massimizzare il peso dello zaino. Abbiamo n items e abbiamo C capacity, ogni item i ha un valore v_i e una weight w_i , prendere un subset di item che massimizza il valore rimanendo nella C capacity.

Example		$C = 7$
weight	value	
1	1	$1 + 7 = 8$
3	4	$4 + 5 = 9$
4	5	
5	7	

Vogliamo $\Theta(n \cdot C)$ time.

Cerchiamo il problema per una capacity più piccola e meno item (ovvero si fa un ordering e si prende il prefix via via), e il combine è il seguente:

$$\text{Combine: } K(i, j) \text{ is an optimal solution for prefix up to } i \text{ item with capacity } j \leq C$$

Do we want to include item i ?

$$K(i, j) = \begin{cases} K(i-1, j-w_i) + v_i, & \text{yes} \\ K(i-1, j) & \text{no} \end{cases}$$

Abbiamo i item mentre prima ne avevamo $i-1$ item, e si vuole prendere l'item i ? No, ovvero si prende la soluzione prima, ma se si prende allora si prende la soluzione di prima + il suo valore, ma dobbiamo fargli spazio, quindi $j - w_i$.

recurrence? ovvero come si fa il base case? o il caso normale?

$$\text{Recurrence: } K(i, j) = \begin{cases} 0 & \text{if } i = \infty \text{ or } j = \infty \\ \max \left\{ \begin{array}{l} v_i + K(i-1, j-w_i) \\ K(i-1, j) \end{array} \right\} & \text{otherwise} \end{cases}$$

Così è facilissimo, si fa row wise, quindi bottom up, seguendo l'algoritmo gasse,

$K[x, \mathcal{S}]$

		w(x)								
w	v	x	0	1	2	3	4	5	6	7
1	0	0	0	0	0	0	0	0	0	0
1	1	1	0	1	2	3	4	5	6	7
3	4	2	0	2	4	5	5	5	5	5
4	5	3	0	2	4	4	5	6	6	6
5	7	4	0	2	4	5	7	7	7	7

per ogni cella si prende sempre il massimo valore possibile, e in caso si controlla se togliere l'ultimo item ha senso.

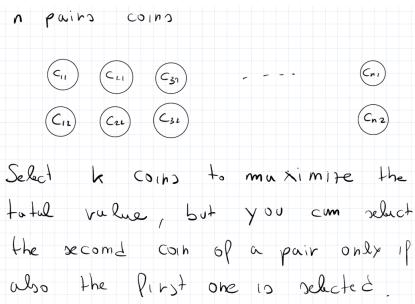
8.9 Fractional knapsack

You can select fraction of items
 $0 \leq p_i \leq 1$ costs $w_i \cdot p_i$
 and value $v_i \cdot p_i$

Si fa in $\Theta(n \log n)$ time, con un algoritmo greedy, per ogni item si fa il ratio $i = \frac{v_i}{w_i}$ value per unit of weight, si sortano gli item b decreasing ratio, così si massimizza il valore e si selezionano gli item fino a che non si finisce la capacity.

8.10 Puzzle exercise

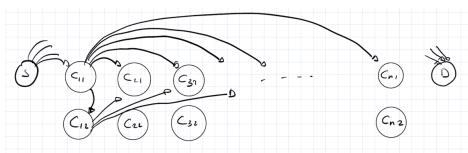
Abbiamo 2 pairs of coins, selezionare k coins per massimizzare il total value, ma si può selezionare il secondo coin solo se anche il primo è stato selezionato.



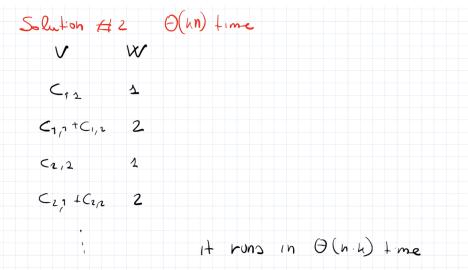
10	2	3
1	12	3
$k=1$	10	
$k=2$	$10 + 3 = 13$	

Con $k=1$ si prende 10 perché possiamo solo selezionare questo, 12 non si può prendere perché si deve prendere 1 prima.

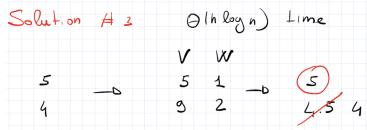
Comunque con $k=3$ non si estende le soluzioni già fatte, perché si prende 10 poi 1 e poi 12, quindi come si fa?????



Prima soluzione $\Theta(n^2)$, dag strategy e via , fake source e destination, e si connettono, ma dobbiamo fare lo shortest path di length k.



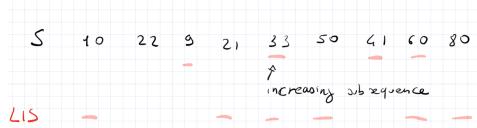
Seconda soluzione, fa $\Theta(nk)$ time , si fa un approach come il knapsack fractional, un coin ha weight 1, un coin e il suo pair ha weight 2, ma non ho capito.



terza soluzione: $\Theta(n \log n)$ si fa un greedy approach, e si usano gli item della seconda soluzione, si fa il ratio e si sortano ma dobbiamo stare attenti di non prendere il coin al primo livello.

8.11 Longest Increasing Subsequence (LIS)

Dato una sequenza S di n elementi, trovare LIS di S. Ovvero una sequenza di elementi che è increasing, ma vogliamo quella più lunga



Prima soluzione quadratica: si prende il sub problem, ovvero si prende solo il prefisso della sequenza, ma qui non serve perché dobbiamo ricordarci i numeri che compongono la LIS, quindi come si fa?

Si forza che LIS(i) finisce con $S[i]$ dove è l'ultimo elemento della LIS, così possiamo fare:

$$\text{LIS}(i) = \text{is the LIS of } S[0 \dots i] \\ \text{which ends with } S[i]$$

Recurrence

$$\text{LIS}(i) = 1 + \max \left\{ \begin{array}{l} \text{LIS}(j) \mid 1 \leq j \leq i-1 \\ \text{AND} \\ S[j] < S[i] \end{array} \right\}$$

La cosa brutta è che dobbiamo fare il check di tutte le LIS trovate finora, quindi gasse, quadratico.