

RICORSIONE

Per applicare la ricorsione devo scomporre il problema in più problemi identici a quello di partenza che siano più piccoli

PROBLEMA DI OTTIMIZZAZIONE

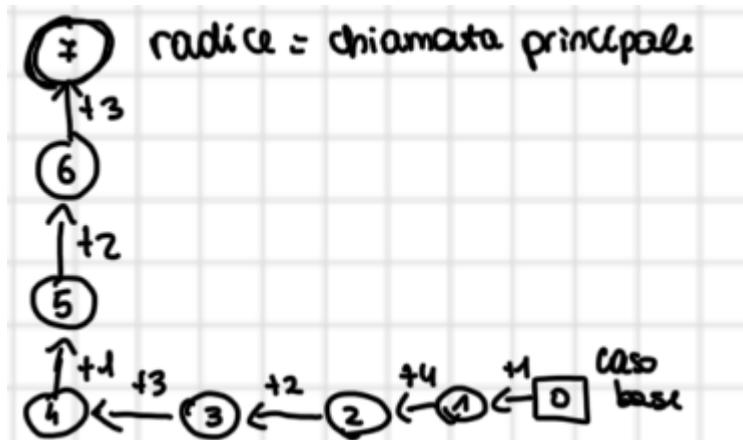
L'ottimizzazione è una caratteristica del problema che esiste a prescindere dalla soluzione.

Un problema si dice di ottimizzazione quando questo ha più soluzioni ma non tutte sono uguali e la capacità di prendere la soluzione con valore di **bontà** più alto (o più basso) in base a ciò che chiede il problema stesso

Bontà = punteggio assegnato alle soluzioni per scegliere poi la migliore

ALBERO DI RICORSIONE

E' un albero dove ogni nodo corrisponde ad una chiamata ricorsiva



Aspetti negativi della ricorsione

- La ricorsione occupa memoria per le tante chiamate
- Algoritmo iterativo più veloce a livello pratico perché non deve gestire chiamate nella memoria

EQUAZIONE DI RICORRENZA

Describe il tempo di esecuzione di un algoritmo ricorsivo

$T(n)$ dove T è il tempo impiegato in base all'input n

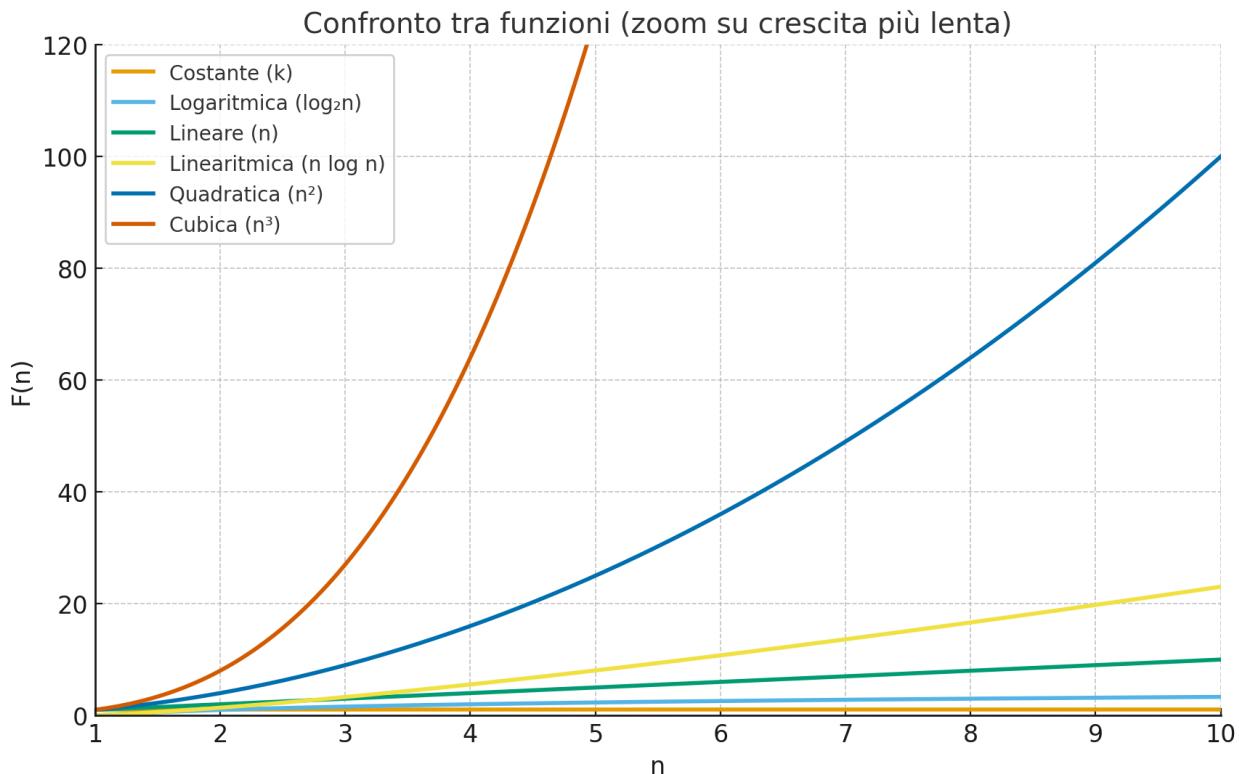
ANALISI DEL COSTO DI UN ALGORITMO

Analisi asintotica – > comportamento dell'algoritmo con valori molto grandi

Confronto tra funzioni

1. $F(n) = k | 0 < k < \infty$ **costante**

2. $F(n) = n$ cresce **linearmente** all'input dato
3. $F(n) = \log_2 n$ cresce meno rispetto alla lineare
4. $F(n) = n \log n$ funzione **linearitmica** cresce un po' di più della funzione lineare
 - funzioni polinomiali
5. $F(n^2)$ funzione **quadratica**
6. $F(n^3)$ funzione **cubica** e così via...
 - funzione esponenziale
7. $F(n) = k^n$
8. $F(n) = n!$



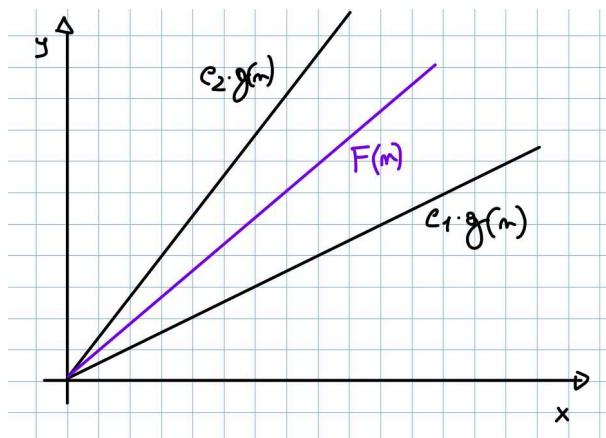
COME CAPIRE LA CLASSE DI UNA FUNZIONE

Se la funzione è $F(n) = n^3 + 7n^4 + 2n \log^3 n + 3$
 confrontiamo i termini e da li capiamo quello con la grandezza maggiore:
 in questo caso quello che cresce più velocemente è $7n^4$ quindi l'ordine di grandezza è n^4

Le classi principali sono 3 $\Theta(g(n))$, $O(g(n))$, $\Omega(g(n))$

1. $\Theta(g(n)) \rightarrow$ theta di g di n
 è la classe di tutte le funzioni che hanno un comportamento asintotico simile a g(n),

$$\Theta(g(n)) = \{F(n) : \exists c_1, c_2, n_0 | 0 \leq c_1 g(n) \leq F(n) \leq c_2 g(n) \forall n \geq n_0\}$$

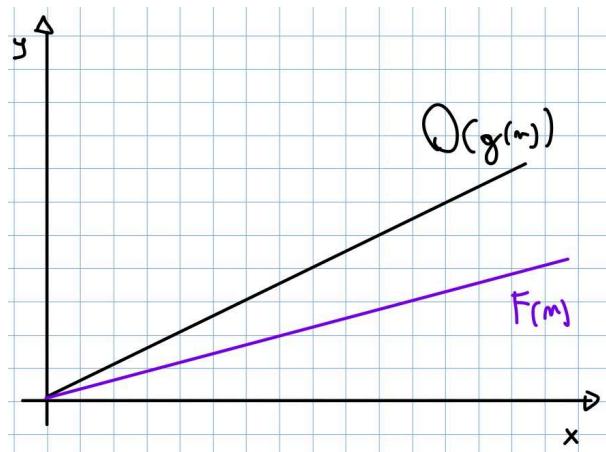


$F(n)$ sta sempre in mezzo o al massimo uguale a $c_1 g(n)$ e $c_2 g(n)$

2. $O(g(n))$

è la classe che limita superiormente il comportamento di una funzione: ovvero la funzione non andrà mai sopra $O(n)$

$$O(g(n)) = \{F(n) : \exists c, n_0 | 0 \leq F(n) \leq c g(n) \forall n \geq n_0\}$$

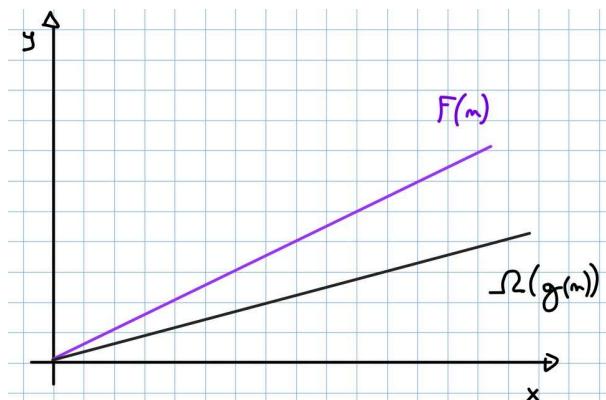


$F(n)$ sta sempre sotto o al massimo uguale alla funzione $O(g(n))$

3. $\Omega(g(n))$

è la classe che limita inferiormente il comportamento di una funzione: ovvero la funzione non andrà mai sotto $\Omega(g(n))$

$$\Omega(g(n)) = \{F(n) : \exists c, n_0 | 0 \leq c g(n) \leq F(n) \forall n \geq n_0\}$$



$F(n)$ sta sempre sopra o al massimo uguale alla funzione $\Omega(g(n))$

esiste anche o piccolo ed è uguale a O solo che la funzione si comporta sempre meglio e mai uguale

Nella realtà...

Tuttavia, nella pratica, gli algoritmi raramente operano nei loro casi estremi. Un algoritmo descritto come $O(n^2)$ potrebbe, in molti scenari reali, comportarsi quasi sempre come un algoritmo $O(n \log n)$, se le condizioni medie del problema lo favoriscono. Viceversa, un algoritmo con complessità $\Omega(n)$ nel caso migliore potrebbe raramente riuscire a raggiungere quel limite, perché le situazioni ottimali sono poco frequenti.

Nell'analisi asintotica degli algoritmi, è prassi comune trascurare i fattori costanti e i termini di ordine inferiore. Questo approccio consente di concentrarsi sulla crescita dominante della funzione di costo al crescere della dimensione dell'input n . Tuttavia, nella pratica quotidiana — soprattutto per input di piccole o medie dimensioni — queste costanti possono fare una differenza sostanziale.

Infatti magari un algoritmo con una complessità peggiore potrebbe essere più efficiente (per input medi o piccoli) di un altro con complessità migliore

In sintesi, possiamo dire che:

- le notazioni asintotiche descrivono il comportamento a lungo termine;
- i fattori costanti e i termini minori influenzano le prestazioni nel breve termine;
- un algoritmo asintoticamente migliore può essere più lento di un altro per una vasta gamma di input reali.

RISOLUZIONE EQUAZIONI DI RICORRENZA

Se indichiamo con $T(n)$ il costo dell'algoritmo su un problema di dimensione n , possiamo esprimere questo comportamento nel modo seguente:

$$T(n) = aT\left(\frac{n}{b}\right) + f(n)$$

Dove a rappresenta il numero di sottoproblemi prodotti, b il fattore di riduzione della dimensione e $f(n)$ il costo complessivo delle operazioni non ricorsive, ossia di tutte quelle attività che non comportano nuove chiamate, come la divisione del problema, la fusione dei risultati o eventuali operazioni preliminari

Non tutti gli algoritmi ricorsivi possono essere descritti mediante l'equazione soprastante:

(es. quickSort)

Questi esempi mostrano che, quando la riduzione del problema non è uniforme, non è possibile applicare direttamente formule standard come il teorema Master, e occorre analizzare la ricorrenza caso per caso. Tuttavia, i principi generali rimangono gli stessi:

è sempre possibile visualizzare la struttura delle chiamate come un albero, contare i nodi generati e sommare i costi dei livelli per ottenere il comportamento complessivo dell'algoritmo

Il metodo dell'albero di ricorsione

L'idea è costruire un albero in cui ogni nodo rappresenta una singola invocazione dell'algoritmo ed ogni arco corrisponda ad una chiamata ricorsiva. In questo modo possiamo capire passo dopo passo come si distribuisce il lavoro complessivo nei diversi livelli di ricorsione

Immaginiamo di avere un algoritmo descritto da:

$$T(n) = aT\left(\frac{n}{b}\right) + f(n)$$

Il metodo dell'albero di ricorsione consiste nel rappresentare questa relazione sotto forma di un albero in cui:

- la *radice* dell'albero corrisponde al problema iniziale di dimensione n ;
- i *nodi interni* rappresentano le chiamate ricorsive generate a ciascun livello;
- ogni nodo è etichettato con il costo del lavoro locale $f(n_i)$, dove n_i è la dimensione del sottoproblema corrispondente;
- i *figli di un nodo* rappresentano le chiamate generate da quel nodo, ognuna di dimensione ridotta di un fattore b .

La costruzione di questo albero permette di scomporre la ricorsione in livelli. Il primo livello contiene un solo nodo (il problema originale), il secondo livello contiene a nodi (uno per ciascun sottoproblema), il terzo livello ne contiene a^2 , e così via. A ogni livello k dell'albero il numero di nodi è a^k , e la dimensione dei sottoproblemi è ridotta a $\frac{n}{b^k}$. Il costo totale associato al livello k può quindi essere scritto come:

$$C^k = a^k f\left(\frac{n}{b^k}\right)$$

cioè come il numero di nodi di quel livello moltiplicato per il costo del lavoro svolto in ciascun nodo.

L'idea è quella di fare la somma dei costi dei singoli livelli dell'albero

$$T(n) = \sum_{k=0}^L C_k = \sum_{k=0}^L a^k f\left(\frac{n}{b^k}\right)$$

dove:

- L rappresenta la profondità dell'albero

Questo approccio fornisce non solo un modo intuitivo per stimare la crescita di $T(n)$, ma anche una rappresentazione visiva dell'andamento del lavoro. Osservando come varia il costo dei livelli successivi, è possibile capire se il lavoro complessivo

è dominato dai livelli più alti (quelli vicini alla radice), dai livelli intermedi o da quelli più profondi dell'albero

Abbiamo 3 possibili casistiche che corrispondono ai 3 casi del teorema master

1. Quando la somma è dominata dai primi livelli, il costo totale è determinato dal lavoro iniziale $f(n)$
2. Quando tutti i livelli hanno lo stesso ordine di grandezza, il costo cresce come il numero dei livelli
3. Quando i livelli inferiori diventano progressivamente più costosi, il termine dominante si sposta verso il fondo dell'albero

COME APPLICARE QUESTO METODO

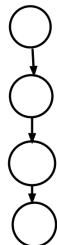
Esempio ricerca binaria

Consideriamo l'equazione:

$$T(n) = T\left(\frac{n}{2}\right) + 1$$

Ogni chiamata ricorsiva genera una sola chiamata su metà dell'input, e il lavoro aggiuntivo (calcolo dell'indice medio e confronto) è costante. Possiamo immaginare l'albero della ricorsione come una catena di chiamate successive, in cui ogni nodo produce un unico figlio di dimensione dimezzata, è una dimensione che ogni livello si dimezza quindi la profondità dell'albero è pari a $\log_2 n$. Il costo totale si ottiene sommando il lavoro di tutti i livelli

Albero:



$$T(n) = 1 + 1 + 1 + \dots + 1$$

dove il numero dei termini 1 è pari a $\log_2 n + 1$. Da qui risulta immediatamente che la complessità è $T(n) = O(\log n)$

In questo caso, il costo per livello è costante e l'albero ha profondità logaritmica: il risultato è quindi una crescita proporzionale al numero dei livelli

Il metodo della sostituzione

Consiste nel formulare un'ipotesi sulla forma asintotica della soluzione e nel dimostrare che tale ipotesi è corretta attraverso un ragionamento induttivo.

Si parte dall'equazione di ricorrenza e, osservando la struttura del problema, si tenta di "indovinare" la crescita di $T(n)$, ad esempio di $O(n)$ $O(n^2)$ ecc... . Una volta formulata una ipotesi la si sostituisce nell'equazione e si verifica se l'uguaglianza (o

disuguaglianza) risulta soddisfatta per valori sufficientemente grandi di n . Se l'ipotesi risulta coerente, viene così confermata; altrimenti, la si modifica finché non produce una forma valida.

Questo metodo è chiamato "della sostituzione" perché prevede di sostituire l'ipotesi nella ricorrenza, semplificarla e controllare che il risultato sia coerente.

Si hanno quindi 3 fasi da seguire:

1. Si formula un'ipotesi sul comportamento asintotico di $T(n)$. Spesso l'ipotesi deriva dall'intuizione fornita da metodi più intuitivi come l'albero di ricorsione
2. Si sostituisce questa ipotesi all'interno dell'equazione di ricorrenza e si verifica se l'uguaglianza o disuguaglianza risulta soddisfatta
3. Se necessario, si aggiusta l'ipotesi (ad esempio includendo una costante o un termine logaritmico) fino a ottenere un'espressione coerente

COME APPLICARE QUESTO METODO

Esempio ricerca binaria

Consideriamo l'equazione:

$$T(n) = T\left(\frac{n}{2}\right) + 1$$

L'intuizione suggerisce una crescita logaritmica. Supponiamo quindi $T(n) \leq c \log_2 n$, sostituendolo nell'equazione otteniamo:

$$T(n) \leq c \log_2 \frac{n}{2} + 1 = c(\log_2 n - 1) + 1 = c \log_2 n - c + 1.$$

Affinché la disuguaglianza $T(n) \leq c \log_2 n$ sia rispettata, è sufficiente che $-c + 1 \leq 0$ cioè $c \geq 1$. Anche in questo caso la nostra ipotesi è coerente $T(n) = O(\log n)$

TEOREMA MASTER (Importante)

Dopo aver visto il metodo dell'albero di ricorsione, che offre una rappresentazione intuitiva della struttura del costo, e il metodo della sostituzione, che consente una verifica formale dell'ipotesi asintotica, possiamo introdurre un terzo strumento, spesso più diretto ed efficace: il Teorema Master. Questo risultato fornisce una regola generale per determinare in modo sistematico l'ordine di grandezza di molte equazioni di ricorrenza della forma

$$T(n) = aT\left(\frac{n}{b}\right) + f(n)$$

dove:

- $a \geq 1$ il numero di sottoproblemi in cui viene suddiviso il problema di dimensione n
- $b > 1$ il fattore di riduzione della dimensione di ciascun sottoproblema.

- $f(n)$ il costo del lavoro non ricorsivo, ossia il tempo a suddividere il problema e a combinare i risultati

Enunciato del teorema master

Sia $T(n) = aT(n/b) + f(n)$ con $a \geq 1$, $b > 1$, e $f(n)$ una funzione positiva. Allora valgono i seguenti casi:

1. se esiste una costante $\varepsilon > 0$ tale che:

$$f(n) = O(n^{\log_b a - \varepsilon})$$

cioè il lavoro non ricorsivo è asintoticamente più piccolo del lavoro interno alla ricorsione, allora

$$T(n) = \Theta(n^{\log_b a})$$

In questo caso domina il costo generato dalla parte ricorsiva dell'algoritmo (le chiamate interne)

2. (generalizzato) se (con $k \geq 0$)

$$f(n) = \Theta(n^{\log_b a} \log^k n)$$

ossia il lavoro non ricorsivo ha lo stesso ordine di grandezza del lavoro ricorsivo (a meno di un fattore logaritmico), allora

$$T(n) = \Theta(n^{\log_b a} \log^{k+1} n)$$

In questo caso, tutti i livelli dell'albero di ricorsione contribuiscono in modo equivalente al costo totale, e la moltiplicazione per un fattore $\log^k n$ nel termine $f(n)$ si traduce in un incremento di un ordine logaritmico nel costo complessivo

3. se esiste una costante $\varepsilon > 0$ tale che:

$$f(n) = \Omega(n^{\log_b a + \varepsilon})$$

cioè il lavoro non ricorsivo cresce più velocemente del lavoro interno, e se inoltre è verificata una condizione di regolarità (detta condizione di dominanza)

$a f(\frac{n}{b}) \leq c f(n)$ per una costante $c < 1$ e n sufficientemente grande, allora

$$T(n) = \Theta(f(n))$$

In questo caso, la parte ricorsiva diventa trascurabile rispetto al lavoro non ricorsivo. Esempio tipico: una ricorrenza come $T(n) = 2T(n/2) + n^2$, dove il termine n^2 domina.

L'idea in breve

L'idea del Teorema Master è confrontare la funzione $f(n)$ — che misura il lavoro esterno alla ricorsione — con la quantità $n^{\log_b a}$, che rappresenta il costo totale del lavoro

ricorsivo. A seconda di quale dei due termini cresce più rapidamente, si individuano tre comportamenti distinti.

In sostanza, $n^{\log_b a}$ descrive quanto "grande" diventa l'albero della ricorsione, mentre $f(n)$ misura il costo aggiuntivo sostenuto a ciascun livello.

Il comportamento finale di $T(n)$ dipende da quale di queste due componenti cresce più rapidamente:

- Se $f(n)$ cresce molto meno di $n^{\log_b a}$, il termine ricorsivo domina
- Se cresce molto di più, prevale il termine non ricorsivo $f(n)$
- Se le due funzioni hanno crescita simile, i contributi si equilibrano e il costo totale si distribuisce tra tutti i livelli.

Il parametro ε viene introdotto proprio per formalizzare questa differenza di crescita: esso rappresenta una "distanza esponenziale" tra le due funzioni.

Quando si scrive, ad esempio, $f(n) = O(n^{\log_b a - \varepsilon})$, si intende che $f(n)$ cresce in modo sensibilmente più lento rispetto a $n^{\log_b a}$, tanto da risultare inferiore di un intero fattore polinomiale. Allo stesso modo, $f(n) = \Omega(n^{\log_b a + \varepsilon})$

Come capire in che caso mi trovo

Un modo semplice per orientarsi è il seguente: se tra $f(n)$ e $n^{\log_b a}$ compare una differenza di potenze, anche minima, questa differenza è sufficiente a stabilire il caso corretto del teorema. Al contrario, quando le due funzioni sono dello stesso ordine, ma $f(n)$ contiene un termine moltiplicativo in $\log n$ o una funzione molto vicina alla crescita polinomiale, il problema rientra nel caso "intermedio"

Esempio 1

Consideriamo la ricorrenza $T(n) = 2T(n/2) + n$. Qui $a = 2$, $b = 2$ e quindi $n^{\log_b a} = n$. ($\log_2 2 = 1$) Poiché $f(n) = n$ ha la stessa crescita, non esiste un $\varepsilon > 0$ tale che $f(n)$ sia né più piccolo né più grande di un fattore polinomiale rispetto a $n^{\log_b a}$: ci troviamo dunque nel caso intermedio, e la soluzione è $T(n) = \Theta(n \log n)$

Esempio 2

Consideriamo invece $T(n) = 2T(n/2) + n^2$. In questo caso $n^{\log_b a} = n$, ma $f(n) = n^2$ cresce più rapidamente di un intero fattore polinomiale, cioè $f(n) = \Omega(n^1 + \varepsilon)$ con $\varepsilon = 1$. Qui il termine non ricorsivo domina e la soluzione è $T(n) = \Theta(n^2)$

Applicazione del Teorema Master

Esempio ricerca binaria

La ricerca binaria è descritta dalla ricorrenza $T(n) = T(n/2) + 1$

In questo caso $a = 1$, $b = 2$ e $f(n) = 1$. Calcoliamo il termine di riferimento $n^{\log_b a}$: poiché $\log_2 1 = 0$, si ottiene $n^{\log_2 1} = n^0 = 1$. Confrontiamo ora $f(n)$ con questo valore:

$$f(n) = 1 = \Theta(1) = \Theta(n^{\log_2 1})$$

Siamo dunque nel secondo caso del Teorema Master, quello in cui $f(n)$ ha lo stesso ordine di grandezza del termine ricorsivo.

Applicando la formula corrispondente, otteniamo:

$$T(n) = \Theta(n^{\log_2 1} \log n) = \Theta(\log n)$$

In ogni passo della ricerca binaria, il problema viene dimezzato, ma il lavoro svolto ad ogni livello (una sola comparazione) è costante. Poiché ci sono $\log_2 n$ livelli fino a ridurre il problema a un singolo elemento, il costo totale cresce in modo logaritmico. Il Teorema Master, in questo caso, conferma in modo immediato ciò che l'intuizione suggerisce: ogni livello contribuisce in modo uniforme, e il numero di livelli determina la crescita complessiva

(altri 2 esempi per gli altri 2 casi nel pdf 4 a pagina 20, dopo ci sono anche altre eq. di ricorrenza risolte con i 3 metodi spiegati [4 Risoluzione delle Equazioni di Ricorrenza.pdf](#))

STRUTTURE DATI

HEAP

Per implementare una struttura dati astratta : Coda con priorità (lifo)

- A parità di valore si considera il tempo di arrivo
- Si considera la chiave (valore più piccola)

0	1	2	3	4	5
5	1	2	3	9	4

le chiavi sono i valori dentro l'array, mai la posizione

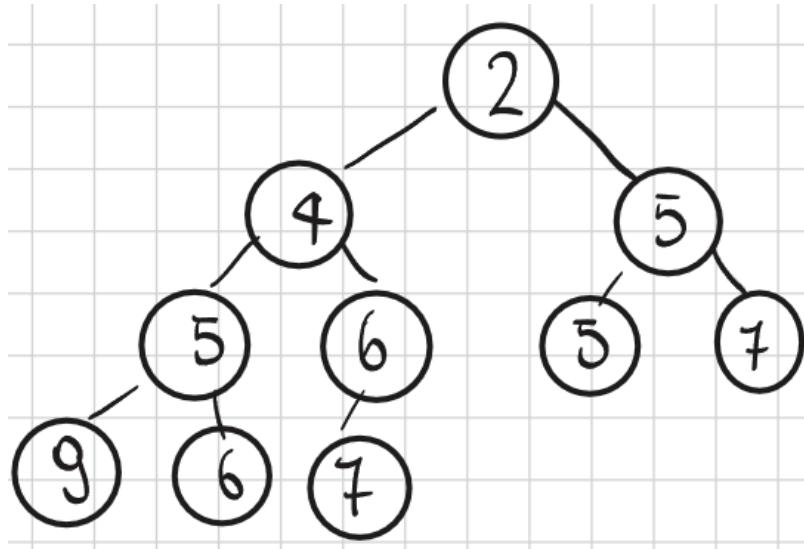
Ovviamente per essere efficiente bisogna che abbia un costo basso

	<u>INS</u>	<u>EXTR</u>	<u>DECRL</u>	<u>MIN</u>	OPERAZIONI
ARRAY	$\Theta(1)$	$\Theta(n)$	$\Theta(1)$	$\Theta(n)$	
ARRAY min	$\Theta(n)$	$\Theta(n)$	$\Theta(n)$	$\Theta(1)$	
B BST	$\Theta(\log(n))$	$\Theta(\log(n))$	$\Theta(\log(n))$	$\Theta(\log(n))$	

1. array disordinato
2. array ordinato
3. BST

Definizione HEAP

Struttura dati non lineare. E' un albero binario dove ogni nodo ha al più due figli. È posizionale, ha un figlio sinistro e uno destro, in base a come lo considero (min heap o max heap) posso avere i figli che siano tutti minori o uguali del padre(max heap) oppure tutti maggiori uguali del padre(min heap). È completo. Tutti i livelli sono pieni. L'ultimo livello può essere non pieno ma solo se i nodi sono allineati da sinistra verso destra



Può essere rappresentato come un albero ma molte volte si sceglie di rappresentarlo come un array, quando aggiungo o rimuovo un elemento bisogna chiamare **heapfy** che risistema la struttura dati.

HEAPFY

Questa procedura è molto importante perché serve a ripristinare la struttura dello heap dopo una modifica

Ecco un esempio in pseudocodice
questo se lo rappresento come un albero

HEAPIFY(x) → Realizza un heap

$l \leftarrow x.\text{left}$

$r \leftarrow x.\text{right}$

IF ($l \neq \text{NULL}$ AND $\text{key}[l] < \text{key}[x]$)

then $\text{min} \leftarrow l$

IF ($r \neq \text{NULL}$ AND $\text{key}[r] < \text{key}[\text{min}]$)

then $\text{min} \leftarrow r$

IF $\text{min} \neq x$ THEN

SWAP ($\text{key}[x]$, $\text{key}[\text{min}]$)

HEAPIFY (min)

Questo se lo rappresento come un array (questo è il min heap, esiste la versione per il max heap), Questo codice ha complessità $O(\log n)$ perché l'heap è un albero binario completo e l'altezza è $\log n$

1. **heapfy(H,i)**

2. `l = left(i)`

3. `r = right(i)`

4. `min = i`

5. `if(l ≤ heapsize and H[l] < H[min]) then`

6. `min = l`

7. `if(r ≤ heapsize and H[r] < H[min]) then`

8. `min = r`

9. `if (min ≠ i) then`

10. `swap(H,i,min)`

11. `heapfy(H,min)`

riga 1 → definizione della funzione, prende in input l'array H e il nodo (indice nell'array) che stiamo prendendo in considerazione

riga 2 → ad I assegniamo l'indice del figlio sinistro di i

riga 3 → ad r assegniamo l'indice del figlio destro di i

riga 4 → assegniamo un minimo temporaneo = ad i (nodo corrente)

riga 5-8 → confronto: controlla se l è dentro l'array e quindi se esiste e se l è più piccolo del minimo temporaneo esso diventa il nuovo minimo, la stessa cosa avviene per r

riga 9-10 → se il minimo non è i allora il nodo i e il nuovo minimo si scambiano,

riga 11 → chiamo heapfy sul nuovo minimo

(heapsize è il numero di nodi contenuti nello heap, e la dimensione dell'array)

CREARE UN HEAP (come array)

- parto da un array vuoto e inserisco gli elementi (non può essere chiamato ancora heap perché non è ordinato nella maniera corretta)

- Uso la procedura *build-minHeap*

```
build-minHeap()
```

```
for i = n/2 down to 1 do heapfy(H,i)
```

- **n** → è il **numero di elementi** presenti nell'array (la dimensione dell'heap).

Esempio: se l'array ha 10 valori, allora **n = 10**.

- **H** → è l'**array** che contiene gli elementi da sistemare in un heap.

È l'array su cui lavorano **Heapify** e **BuildHeap**.

- **i** → è l'**indice del nodo** che stiamo "heapificando" in quel momento.

Il ciclo parte da **n/2** (cioè dall'ultimo nodo che ha almeno un figlio) e va fino a **1** (la radice).

questo perché da $n/2 + 1$ iniziano le foglie dell'albero, che non hanno figli

per fare un max heap chiameremo *build-maxHeap* e cambierà la funzione heapfy in modo da mettere i nodi con valore più alto in cima, quindi al contrario rispetto a min-heap

Modifica di un valore dentro l'heap

Per fare questo si usa una funzione chiamata updateKey che cambia il valore della chiave del nodo, in questo caso è una decreaseKey su un min heap :

1. **DECREASE-KEY(H, i, k)**

2.

```
`H[i] = k // Aggiorno la chiave all'indice i`
```

3.

```
`while (i > 1 and H[parent(i)] > H[i]) do // Controllo indice > 1 invece di NULL`
```

4. `swap(H, i, parent(i))`

5. `i = parent(i)`

riga 1 → definizione della funzione, prende in input l'heap, il nodo da cambiare e il valore nuovo

riga 2 → aggiorno la chiave

riga 3-5 → finché i non è la root e il padre di i è maggiore di i scambia il padre con il figlio e assegna il nuovo padre

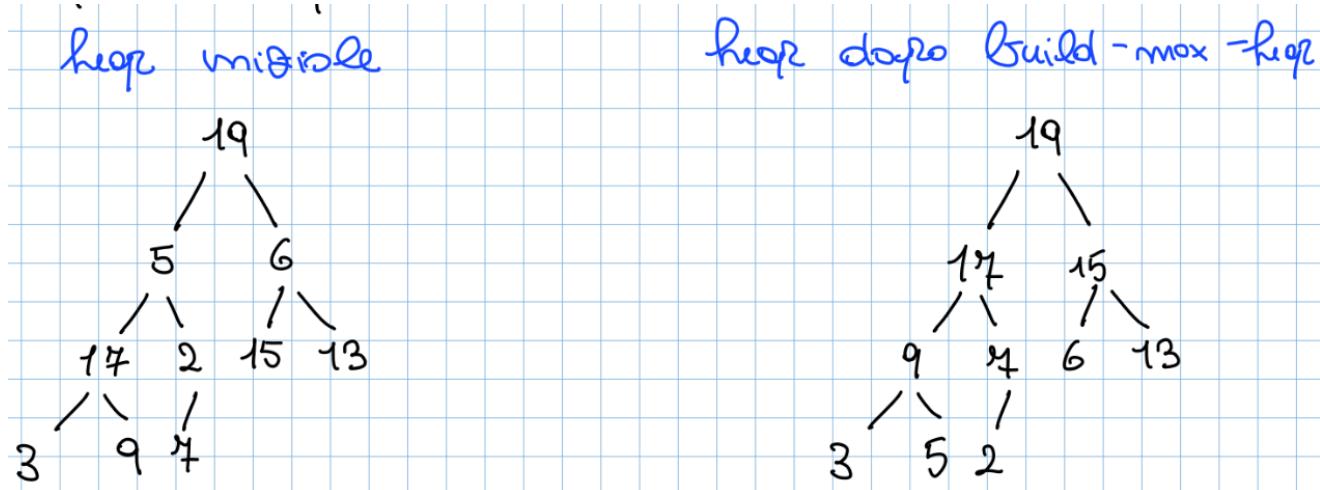
La struttura heap ci semplifica la vita quando parliamo di velocità e complessità nei problemi di ordinamento

Ad esempio nel **selection sort** normale con array il problema maggiore è trovare ogni volta il massimo il che ci porta ad avere una complessità $O(n^2)$

Possiamo invece usare una struttura heap che migliora la complessità, usando comunque un array

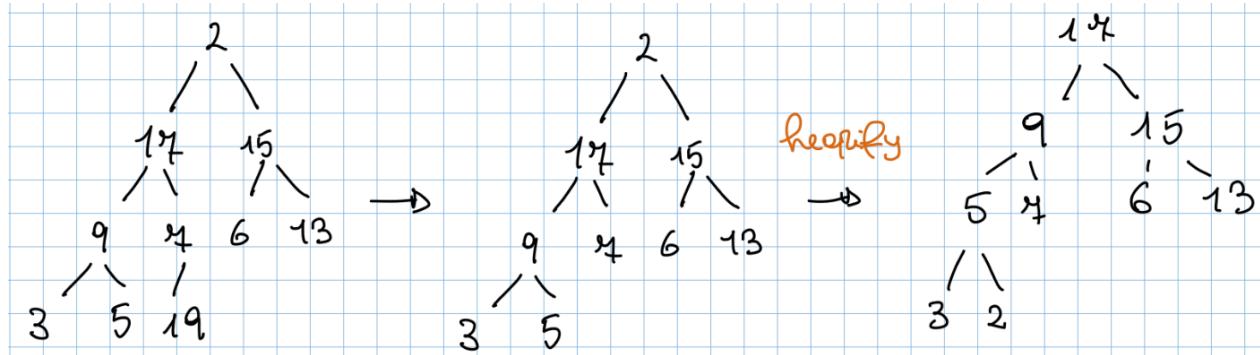
Ecco quindi il procedimento per creare un selection sort con un heap:

(Viene rappresentato come un albero ma nella realtà è un array)



1. Parto da un array disordinato (sinistra) e chiamo **build-max-heap** che mi va a creare un max-heap in cui ho i figli sempre minori uguali del padre
2. Per prendere il massimo scambio il massimo (19) che è la root con il minimo (2) che è l'ultimo nodo, quindi prendo il nuovo ultimo nodo e chiamo **heapfy** che mi va a

riordinare di nuovo il max-heap



3. Posiziono il massimo (19) alla fine dell'array e ripeto questa procedura fino a quando l'array non è ordinato (ovviamente scalo la dimensione su cui opero di 1 ogni volta che posiziono un massimo)

Da questa idea di usare un heap come struttura dati nasce l'**heap-sort**

HEAP SORT

1. **heapSort(A, n)**
2. `buildMaxHeap(A, n)`
3. `for i = 0 to n-1 do`
4. `extractMax(A)`

riga 1 → definizione della funzione, prende in input l'array disordinato e la dimensione

riga 2 → richiama la funzione buildMaxHeap

riga 3-4 per ogni elemento dell'array (max heap) ne estrae il massimo

1. **extractMax(A)**
2. `Swap(A, 0, n-1)`
3. `n = n-1`
4. `heapfy(A, 0)`

riga 1 → definizione della funzione, prende in input l'array

riga 2 → scambia il massimo (root) con l'ultimo elemento dell'array (che sarà una foglia e quindi un elemento piccolo)

riga 3 → riduce la dimensione dell'array perché il massimo è già stato ordinato

riga 4 → chiama heapfy per risistemare l'array e passa il primo elemento perché è quello che va risistemato

Questa procedura ha complessità $O(n \log n)$ molto simile al mergeSort che è $\Theta(n \log n)$ e consuma anche meno memoria perché qui lavoriamo con un singolo array a differenza del mergeSort

Definizione

E' dimostrato che un algoritmo di ordinamento che fa confronti non potrà mai scendere sotto $\Theta(n \log n)$ come complessità

Ma non tutti gli algoritmi di ordinamento hanno bisogno di fare comparazioni

Algoritmi di ordinamento che non necessitano di confronti

Ne vedremo 2:

COUNTING SORT

Costruiamo un array A:

`A = [4,3,5,3,5,3,2,3,4]`

adesso creo un secondo array chiamato C, questo array avrà l' ultimo indice = all'elemento massimo di A + 1, quindi nel nostro caso avrà dimensione 7 ($\max(A) = 5 + 1$ ma dato che inizio da 0 avrà dimensione 7), lo inizializzo a 0:

`C = [0,0,0,0,0,0,0]`

`0 1 2 3 4 5 6`

L'ultimo indice è l'indice k che ci da la dimensione

Con questo array C, conterò le occorrenze dei numeri presenti in A, nel nostro caso:

`C = [0,0,1,4,2,2,0]`

`0 1 2 3 4 5 6`

infatti il 2 compare una volta, il 3 quattro volte ecc...

Quindi `C[i]` = numero di occorrenze dell' elemento i in A

Adesso modifichiamo l'array C rendendolo C', in cui in `C[i]` andiamo a sommare il numero `C[i]+C[i-1]`

Quindi sommiamo l'elemento corrente con il precedente e lo posizioniamo nell'elemento corrente

In pseudocodice stiamo facendo: `for i = 0 to k do (C[i] = C[i] + C[i-1])`

Otterremo così l'array C modificato in C':

`C' = [0,0,1,5,7,9,9]`

`0 1 2 3 4 5 6`

In questo modo all'interno di C' avremo il numero di elementi in A minori o uguali ad i infatti quanti elementi ci sono minori o uguali a 2 nell'array A? 1 (solo il 2) quanti elementi ci sono minori o uguali a 3 nell'array A? 5 (i quattro 3 e il 2) ecc...

Adesso creiamo un ultimo array B della stessa dimensione di A

B = [0,0,0,0,0,0,0,0,0]

0 1 2 3 4 5 6 7 8

Fatto questo si scorre l'array A al contrario, quindi partendo dalla fine, questo serve a mantenere la stabilità, poi:

Per ogni elemento i (elemento corrente dentro A), si posiziona i in B[C'[i]-1] e successivamente si decrementa: C'[i] di 1, perché avendo posizionato l'elemento i ne ho 1 in meno da posizionare quindi devo scalare di 1 il numero degli elementi i: se C'[3] = 5 ho 5 tre da posizionare, dopo aver posizionato il primo 3 C'[3] = 4 e così via

Ecco lo pseudocodice del countingSort

```

1. CountinSort(A,n){
2.   `k = max(A)`
3.   `C = new Array(k+1)`
4.   `for i = 0 to k do C[i] = 0`
5.   `for i = 0 to n-1 do C[A[i]] = C[A[i]]+1`
6.   `for i = 1 to k do C[i] = C[i] + C[i-1]`
7.   `for i = n-1 down to 0 do`
8.     `B[C[A[i]]-1] = A[i]`
9.     `C[A[i]] = C[A[i]]-1`
10. }
```

La complessità del counting sort è $O(n + k)$ dove n è il numero di elementi presenti in A e k è il MAX(A)

RADIX SORT

L'idea generale del radix sort è ordinare l'array in base alle cifre di ogni elemento, ovvero si ordina (generalmente) partendo dalla cifra meno significativa e ad ogni passata si va fino a quella più significativa

Faccio un esempio visivo:

A = [329,457,657,839,436,720,355]

329 720 720 329
 457 355 329 355
 657 436 436 436
 839 457 839 457
 436 657 355 657
 720 329 457 720
 355 839 657 839

Partendo dall'array iniziale, ho ordinato prima rispetto alla terza cifra (seconda colonna) poi rispetto alla seconda (terza colonna) e infine rispetto alla terza (quarta colonna), così facendo ho ordinato l'array

Pseudocodice

1. **RadixSort(A,n,h)**
2. `for i = 0 to h = 1 do`
3. `countingSort(A,n,h)`

dove A è l'array, n la dimensione e h sono le cifre dei numeri all'interno (ovviamente devono avere tutti lo stesso numero di cifre)

La formula per trovare il numero in base alla cifra che voglio (unità,decine,ecc...) è:

$$\left(\frac{A[i]}{10^h} \right) \% 10$$

esempio: con 10^0 cioè la cifra delle unità

$$\left(\frac{345}{1} \right) \% 10$$

$$345 \% 10 = 5$$

La complessità del radix sort è $O(n)$

TABELLE AD INDIRIZZAMENTO DIRETTO

Queste tabelle funzionano in maniera molto simile all' array C del counting sort, ovvero un array in cui si salva il numero di occorrenze per ogni numero, all'indice che rappresenta quel numero, infatti per queste tabelle supponiamo di avere un insieme di numeri:

$$S = \{1,3,3,5,5,6,7\}$$

salviamo il numero di occorrenze di questi numeri all'interno di un array T che ha dimensione pari al massimo+1 dell'insieme S

$$T = [0,1,0,2,0,2,1,1]$$

$$0 \ 1 \ 2 \ 3 \ 4 \ 5 \ 6 \ 7$$

Infatti nell'insieme S lo 0 compare zero volte l'1 una volta, il 3 due volte e così via
 E' molto facile da capire e da implementare infatti in pseudocodice le operazioni sono:
 Inserimento:

1. **insert(T, k)**
2. `T[k] = T[k]+1`

Cancellazione:

1. **delete(T, k)**
2. `if T[k] = 0 then T[k] = 0`
3. `else T[k] = T[k] -1`

Ricerca:

1. **search(T, k)**
2. `if(T[k] ≥ 1) return 1`
3. `return`

dove T è l'array e k l'elemento

Questa struttura è molto semplice ma NON è efficiente se la differenza tra min e max nell'insieme è grande, infatti se avessi un insieme S = {1,2,3,300,5000} avrei 5000+1 caselle di cui la maggior parte vuota

Sarebbe ottimo avere una struttura con le caratteristiche di una tabella ad indirizzamento diretto ma che consumi il giusto numero di celle

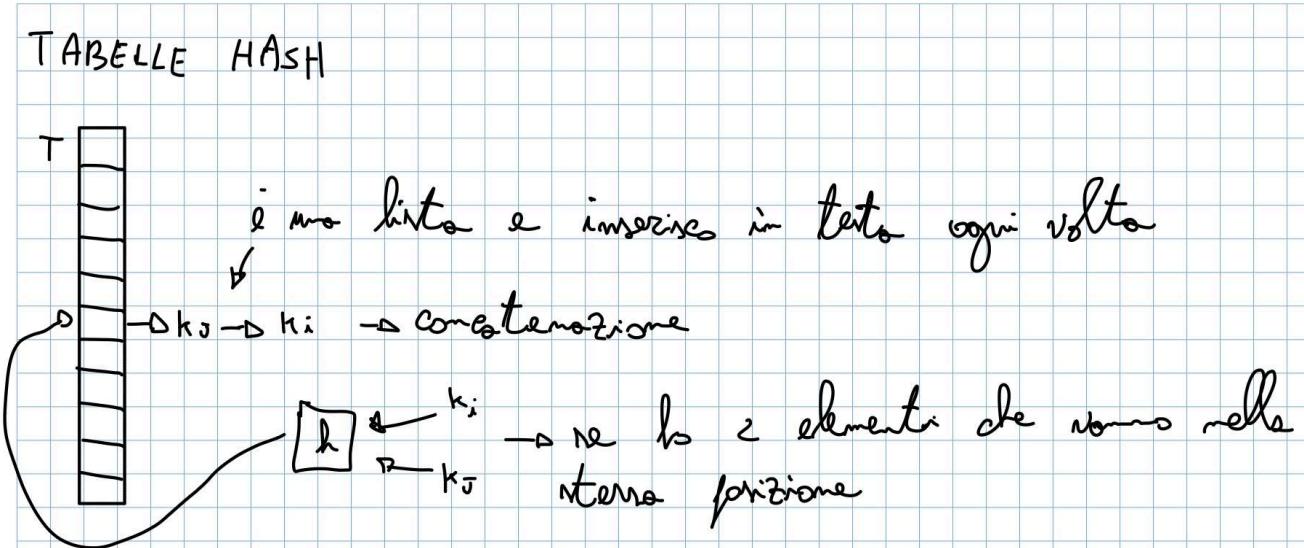
TABELLE HASH

Queste tabelle funzionano in maniera simile a quelle ad indirizzamento diretto, solo che utilizzano una funzione (chiamata hash) che ne determina la posizione, in questa maniera eliminano il problema di creare array giganteschi per inserire valori altrettanto grandi, vediamo dopo come funziona, per ora gestiamo le **collisioni**.

Le collisioni possono avvenire se ho 2 valori uguali da inserire (o in generale la funzione produce lo stesso risultato e quindi lo stesso indice), la funzione hash verrà eseguita 2 volte sui 2 valori e quindi determinerà la stessa posizione per entrambi i valori, questo problema viene gestito implementando una lista in ogni indice dell'array, in questa maniera se 2 valori devono andare nello stesso indice avremo in quell'indice una lista che conserverà i valori

Se K_i e K_j sono due chiavi e h è la funzione hash, la collisione si verifica quando $h(K_i) = h(K_j)$

Rappresento questa struttura così:



Lo pseudocodice delle funzioni con cui gestire la lista è questo:

1. **insert(T, k)**
2. `listInsert(k, T[h(k)])`

1. **search(T, k)**
2. `return listSearch(k, T[h(k)])`

Il **caso pessimo** è quando tutti gli elementi collidono e quindi ho una lista concatenata normalissima e quindi perdo le proprietà di una tabella hash

Per capire quanto la tabella sia piena nasce la misura: **fattore di carico**

$$a = \frac{n}{m}$$

Dove n è il numero di celle occupate

e m è la lunghezza dell'array T (ovvero quante celle ho in totale)

Il fattore di carico varia tra 0 e 1 (0 = vuota 1 = piena) se supero l'1 vuol dire che ogni cella è piena e si sono verificate collisioni

Ogni elemento k ha la stessa possibilità di essere inserito nella posizione i , ovvero $\frac{1}{m}$

La complessità delle funzioni di ricerca nel caso di successo e di insuccesso è $O(1 + a)$, l'1 è simbolico perché controlliamo la lista di elementi per capire se è alla fine, in caso accederà ad un null (+1)

Adesso cerchiamo di capire quali sono i *principali metodi per creare una funzione hash*:

Metodo della divisione

E' il metodo più usato per definire h perché è abbastanza semplice da implementare.

Abbiamo k che sono le chiavi da inserire, e m è il numero di celle dell'array

$$h(k) = k \bmod m$$

Esempio:

`T = [0,0,0,0,0,0,0,0,0]`

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

I numeri da inserire sono: 50,53,67

$50 \bmod 10 = 0 \rightarrow$ indice 0 nell'array

$53 \bmod 10 = 3 \rightarrow$ indice 3 nell'array

$67 \bmod 10 = 7 \rightarrow$ indice 7 nell'array

`T = [50,0,0,53,0,0,0,67,0,0]`

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

(nell'array non inserisco direttamente il numero ma un puntatore che punta ad una lista che contiene i numeri, 3 liste da un elemento in questo caso)

Scegliere bene il modulo è importante, ad esempio se un numero k fosse molto grande e scegliessi un m troppo piccolo avrei un array piccolo con liste enormi

Esempio:

$$8012 \bmod 5 = 2$$

$$8012 \bmod 5 = 2$$

già con 2 numeri diversi ho ottenuto la stessa posizione nell'array quindi li ho fatti collidere

Metodo della moltiplicazione

Abbiamo k che sono le chiavi da inserire, e m è il numero di celle dell'array

Scegliamo un A compreso tra 0 e 1 (0 e 1 esclusi) allora:

$$h(k) = \lfloor m * ((k * A) \bmod 1) \rfloor$$

- Il mod 1 serve ad escludere la parte decimale della moltiplicazione $k * A$
- $0 < k * A < k \rightarrow$ ricorda che sarà per forza minore di k perché sto moltiplicando per un numero A che essendo compreso tra 0 e 1 va a diminuire la quantità k
- $0 \leq m * ((k * A) \bmod 1) < m$
- Di tutta la moltiplicazione ne prendo il floor per escludere la parte intera

INDIRIZZAMENTO APERTO (non si usano liste per le collisioni)

Qui a differenza di prima non utilizziamo una lista per concatenare le chiavi che vanno in collisione (da notare che il fattore di carico non supererà mai 1), quindi questa tabella hash può riempirsi al punto tale da non accettare più valori per l'inserimento, questo permette di velocizzare le operazioni di ricerca.

Per indirizzare le chiavi esaminiamo le celle della tabella (*ispezione*) finché non troviamo una cella vuota per l'inserimento, anziché ispezionare sempre partendo dalla cella 0 fino a quella $m-1$, esaminiamo in base alla chiave da inserire, estendiamo quindi la funzione hash in modo da prendere un secondo input, ovvero un numero che parte da 0 e che indica il tentativo di inserimento, questo numero i obbliga la funzione hash a dare un risultato diverso ogni volta (visto che ogni volta che l'inserimento fallisce i viene incrementata)

$h(k) \rightarrow h(k, i)$: dove k è l'elemento da inserire e i è il tentativo da cui partire per esaminare

$(0, \dots, m-1)$

E' importante quindi garantire che $h(k, i) \neq h(k, j)$ se $i \neq j$

Esempio:

1. primo tentativo ($i = 0$)

$h(3, 0) = 5 \rightarrow 5$ è l'indice nell'array ovvero la posizione in cui deve andare il 3

2. Secondo tentativo ($i = 1$)

Se la posizione numero 5 era occupata i si incrementa e si passa al prossimo tentativo

$h(3, 1) = 6$ se la posizione 6 è libera il 3 viene inserito altrimenti si continua così fino alla fine dell'array o alla prima posizione vuota

Con questo tipo di implementazione le funzioni diventano:

1. **insert(T, k)**

2. `i = 0`

3. `while(i < m and T[i] != null) do`

4. `i = i + 1`

5. `if (i < m) then T[i] = k`

1. **search(T, k)**

2. `i = 0`

3. `while(i < m and T[i] != null) do`

```

4.     `if(T[i] = k) return true`

5.     `i = i + 1`

6.     `return false`
```

Dove m è il numero di celle e n è il numero di elementi

Entrambe queste funzioni hanno complessità $O(n)$

In generale quindi l'indirizzamento aperto, quando si verificano collisioni non concatena con una lista ma scorre le posizioni successive fin quando se ne presenta una disponibile per l'inserimento, per farlo segue delle tecniche, ne abbiamo 3

HASHING

1. Ispezione lineare

Per farlo seguiamo la formula

$$h(k, i) = (h'(k) + i) \mod m$$

dove $h(k)$ è la funzione hash completa che gestisce anche le collisioni

$h'(k)$ è un'altra funzione hash (ausiliaria) che serve per costruire $h(k)$

In pratica quindi $h'(k)$ è la funzione hash che mi indica il punto da cui iniziare l'ispezione lineare, quindi a livello pratico inizia dalla posizione $h'(k)$ e scorre le celle in cerca di una libera fino ad arrivare a quella che la precede (il modulo serve per tornare all'inizio dell'array quando si trova alla fine di quest'ultimo)

Problemi:

Questa metodologia è semplice da implementare ma presenta un problema: ovvero gli elementi k sono "attratti" dalle parti dell'array con massa più grande:

Glomerazione primaria

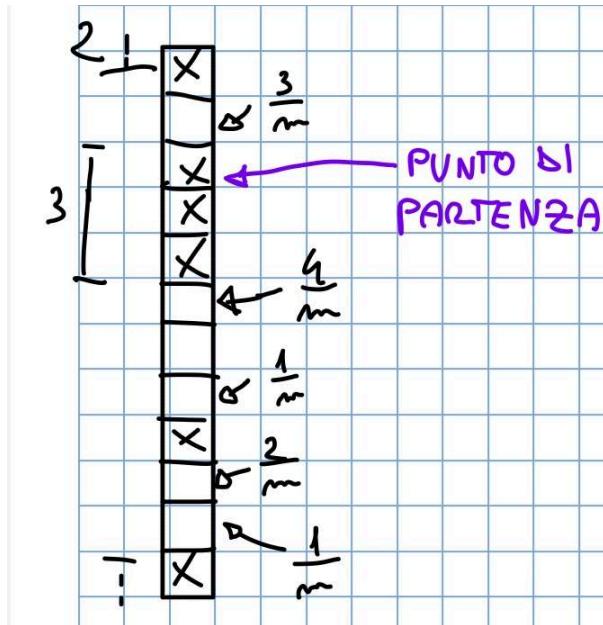
Partiamo da un array vuoto, ogni elemento k ha probabilità $\frac{1}{m}$ (m = numero totale di celle) di finire in una determinata cella, ma se il mio array non è vuoto ma ho ad esempio delle zone più occupate rispetto ad altre:

`T = [1,0,1,1,1,0,0,0,1,0,0,1]` 0 cella vuota, 1 cella piena

`0 1 2 3 4 5 6 7 8 9 10 11`

Supponiamo che la nostra funzione $h'(k)$ ci abbia dato come punto di partenza l'indice numero 2 che è occupato, così come lo sono anche l'indice 3 e 4, l'elemento $h'(k)$ aveva probabilità $\frac{1}{m}$ di finire all'indice 2 ma dato che è occupato la funzione $h(k)$ scorrerà all'indice successivo, questo comporta un aumento delle probabilità a $\frac{2}{m}$ di finire all'indice 3 perché le probabilità si sommano, anche il 3 è occupato quindi va al 4 (la probabilità è $\frac{3}{m}$) e dato che anche il 4 è occupato andrà ancora avanti all'indice 5 che è il primo libero, la probabilità totale adesso è $\frac{4}{m}$.

Abbiamo capito perché le parti più massicce tendono ad ingrandirsi di ancora di più attirando i nuovi elementi, in pratica ad ogni tentativo in cui la cella è occupata la probabilità di finire alla prossima cella aumenta di 1



come si può vedere dal disegno l'elemento k ha probabilità $\frac{1}{m}$ di finire in una cella vuota e se capita in una cella piena ad ogni spostamento la probabilità aumenta

2. Ispezione quadratica

segue la formula

$$h(k, i) = (h'(k) + c_1 i + c_2 i^2) \mod m$$

dove c_1 e c_2 sono diversi da 0 e sono delle costanti

In questo caso la glomerazione primaria non c'è ma invece c'è la secondaria che è molto più contenuta

3. Hashing doppio

si basa sulla formula

$$h(k) = (h'(k) + i h''(k)) \mod m$$

ALBERI ROSSO-NERI

Introduzione

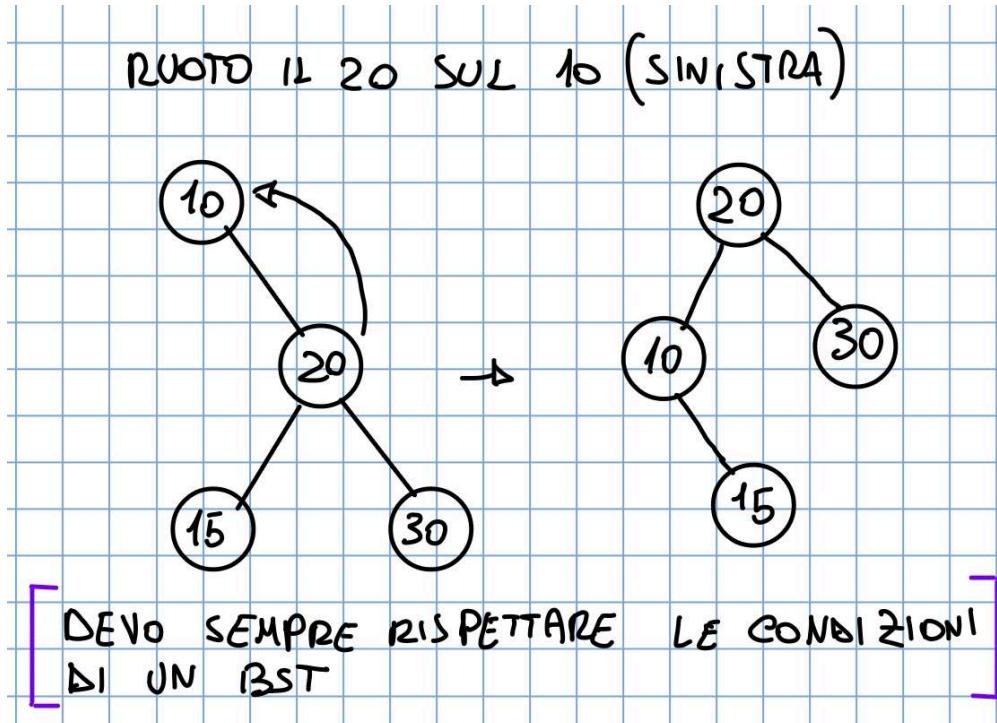
Un albero per dirsi bilanciato deve avere come obiettivo quello di (almeno asintoticamente) fare lo stesso lavoro sia nella parte di destra che nella parte di sinistra
Prima di parlare degli alberi rosso-neri dobbiamo prima capire il concetto di rotazione in un albero bilanciato (bst)

ROTAZIONE

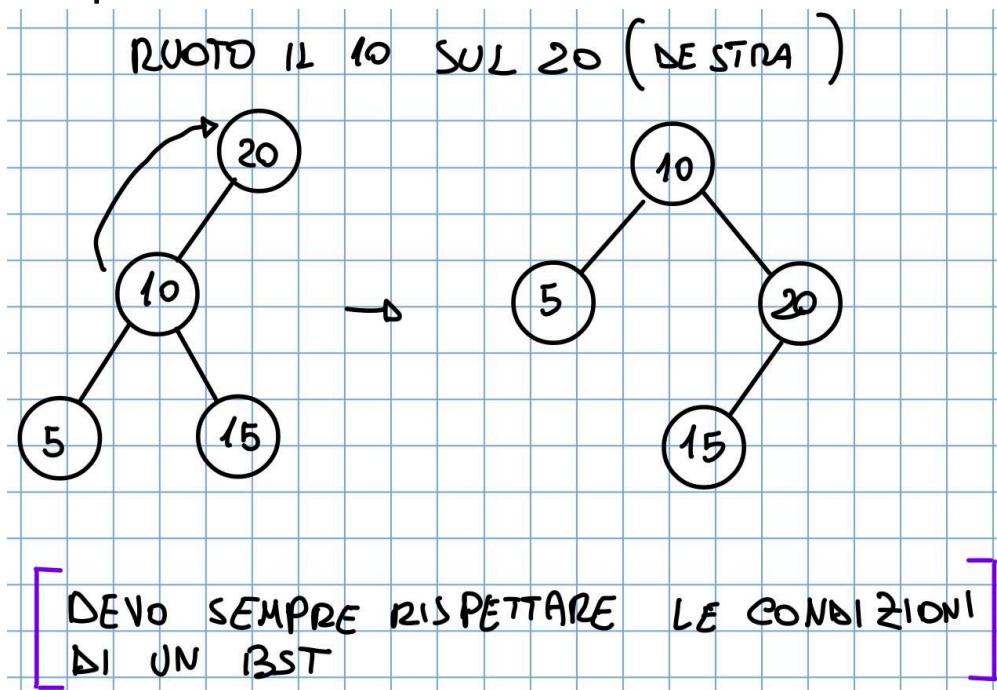
Abbiamo 2 tipologie di rotazione: rotazione a destra e rotazione a sinistra.
E' abbastanza semplice ma bisogna prestare attenzione:

Posso ruotare solo se il nodo su cui devo ruotare ha un figlio: se ruoto a sinistra deve avere un figlio destro, se invece devo ruotare a destra deve avere un figlio sinistro

Esempio rotazione sinistra



Esempio rotazione destra



In questo modo abbiamo modificato la struttura dell'albero ma non "il suo significato". E' questo il motivo per cui queste rotazioni sono importanti, servono quindi a ribilanciare l'albero.

Adesso parliamo degli alberi rosso-neri

ALBERI ROSSO-NERI -> REGOLE

Gli alberi rosso neri hanno 5 proprietà fondamentali:

1. Ogni nodo è rosso o nero
2. La root è sempre nera
3. Le foglie sono sempre nere

- Rilassiamo questa regola :

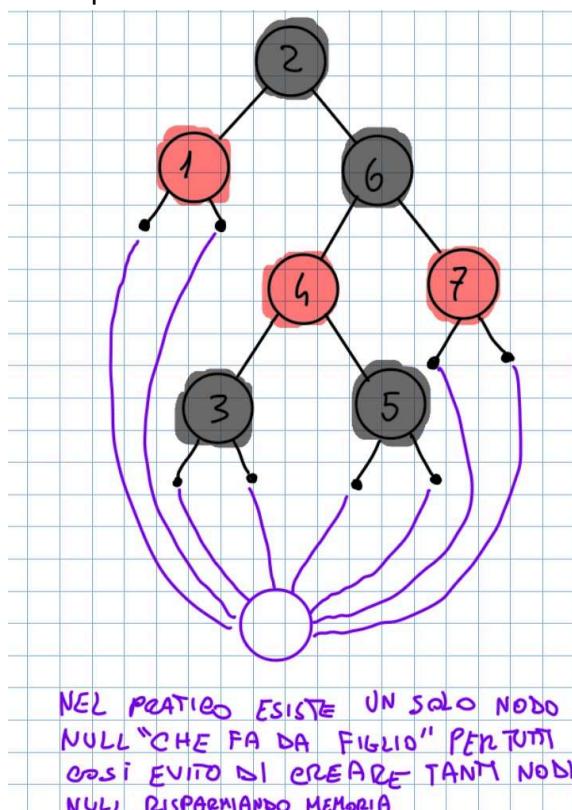
A livello sia pratico che teorico considero le foglie come NULL, così farà risultare questi nodi NULL come nodi neri, essendo questi ultimi foglie la proprietà è rispettata, aumento l'altezza dell'albero di 1 livello ma a livello asintotico non cambia molto

4. Un nodo rosso ha sempre figli neri
5. Un qualsiasi cammino da un nodo ad una foglia incontra sempre lo stesso numero di nodi neri

Nota bene: ci sono vari modi per colorare uno stesso albero, l'importante è rispettare le proprietà

Queste proprietà possono essere rispettate solo se l'albero è bilanciato.

Esempio di albero rosso nero



(Un albero rosso nero che è solo colorato di nero può comunque rispettare le proprietà, perché non è obbligatorio che ci siano nodi rossi)

Altezza nera

Denotiamo con $bh(x)$ l'altezza nera che è il numero di nodi neri partendo da un nodo x (senza contarlo) e arrivando ad una foglia, ad esempio se da x a una foglia incontro 3 nodi neri $bh(x) = 3$

Un albero rosso nero deve essere bilanciato per poter rispettare le proprietà, ma nella realtà difficilmente ci saranno alberi perfettamente bilanciati, l'importante è che lo siano almeno abbastanza da far rispettare le proprietà: in un albero rosso-nero, se denotiamo *l'altezza* (altezza normale non altezza nera) *minima* = k allora *l'altezza massima* di un'altra parte dell'albero può essere al più $2k + 1$, più di così non posso sbilanciare l'albero altrimenti non rispetterei più le 5 proprietà

Lemma

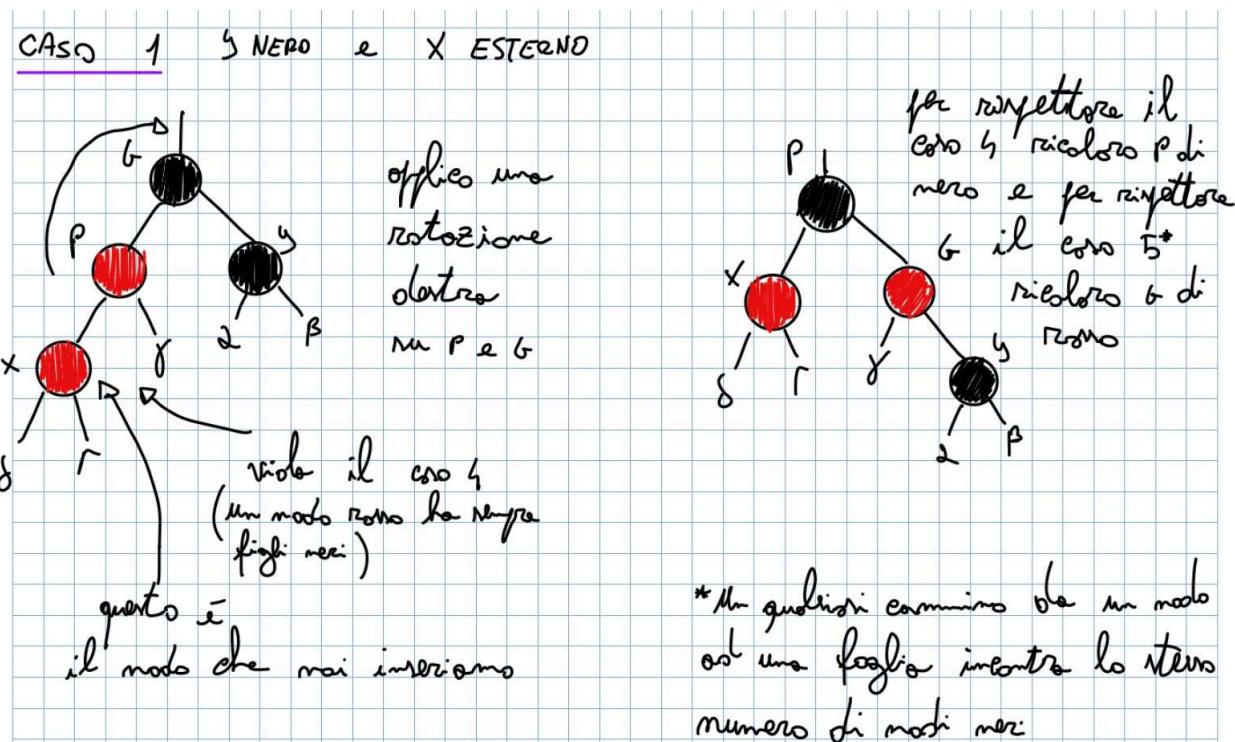
L'altezza massima di un albero rosso-nero con n nodi interni è $2\log(n + 1)$
(dimostrazione sul libro a pagina 257 [Introduzione Agli Algoritmi](#))

INSERIMENTO IN UN ALBERO ROSSO NERO

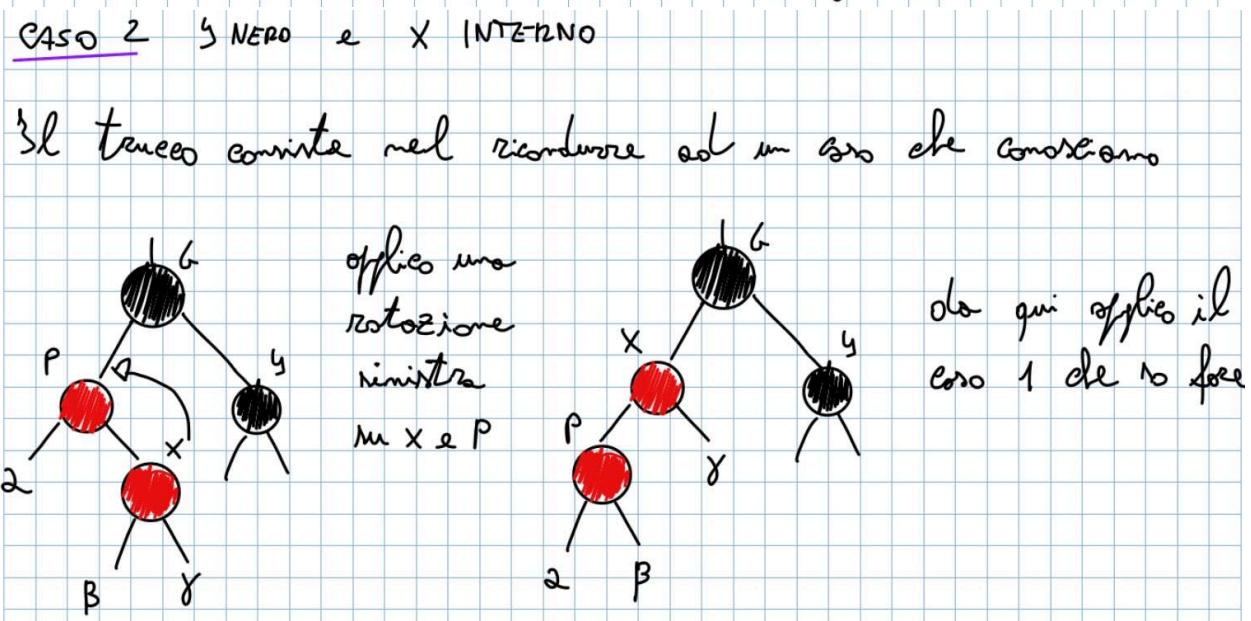
Il nodo inserito è sempre rosso così diminuiamo i problemi.

In totale abbiamo **3 casi**:

(Spiegherò i casi con l'albero d'esempio usato a lezione)



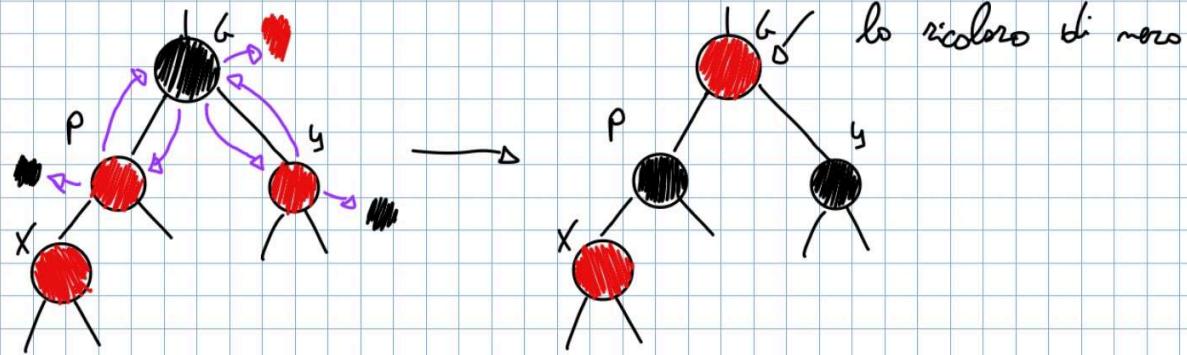
1.



2.

CASO 3 y Rosso

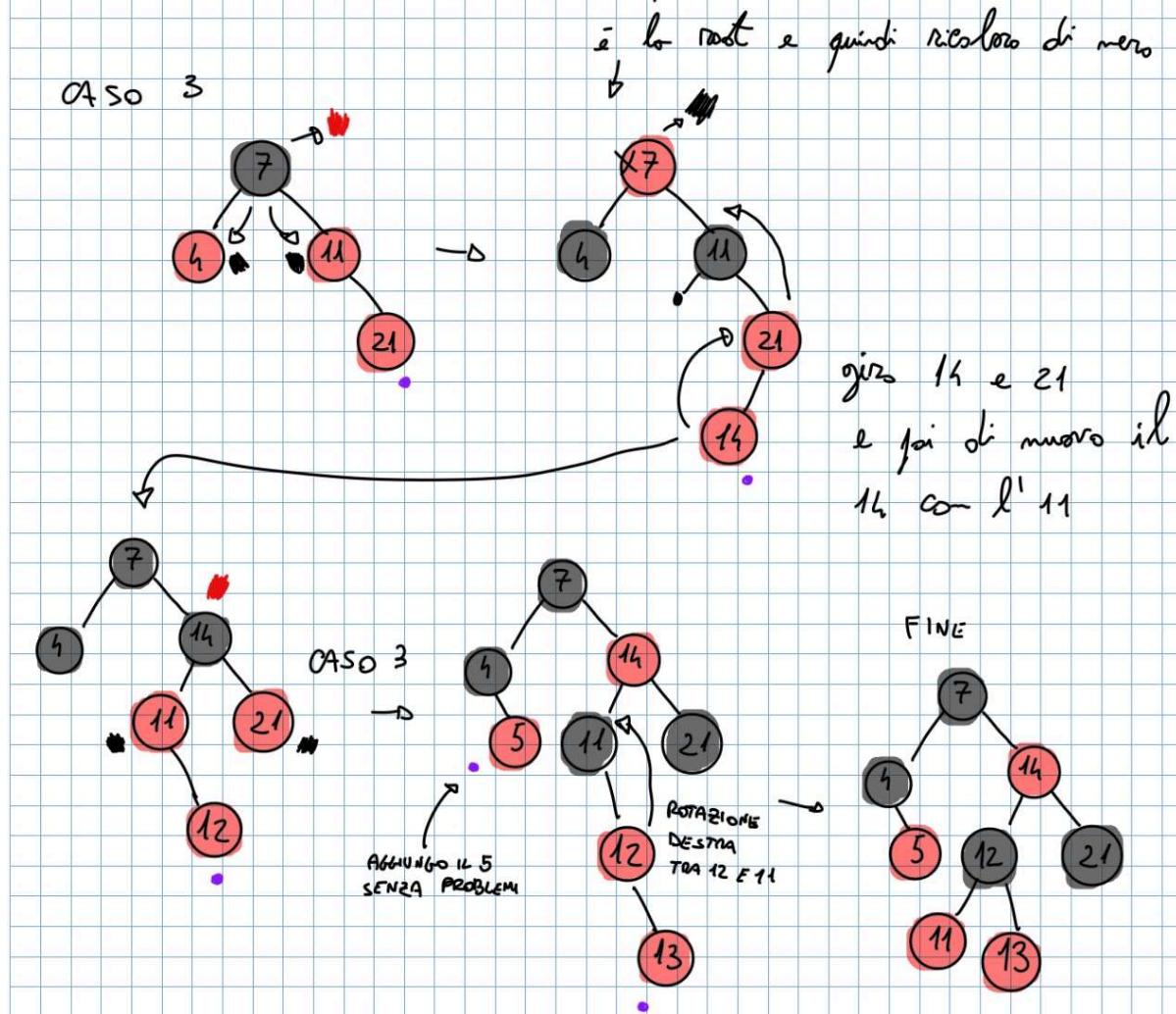
In questo caso il padre di y (g) è nero e il colore nero si sposta a ragionare il rosso dei figli.



3.

Adesso vediamo un esempio

ESEMPIO (EVENZIATORE = COLORE CHE AVEVA PRIMA DI EVENTUALI MODIFICHE, COLORE ACCANTO CON LA PENNA = COLORE DOPO LE MODIFICHE; '•' = NODO DA AGGIUNGERE)

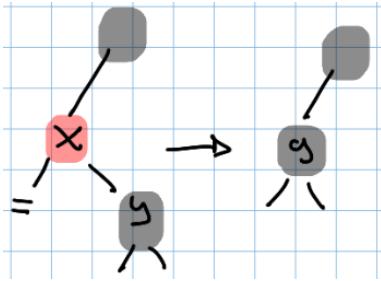


(Altri esempi sono visibili negli appunti 9 e 10)

CANCELLAZIONE

La cancellazione è un'operazione abbastanza complessa, iniziamo col dire che

l'eliminazione di *un nodo rosso non comporta problematiche*, basta eseguire una normale cancellazione di un nodo in un albero,



mentre per quanto riguarda la *cancellazione di un nodo nero così facendo altero sempre il numero di nodi neri in un cammino (regola 5)* quindi va gestito attentamente:

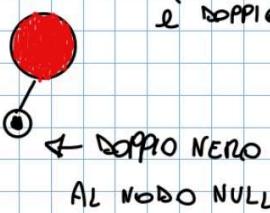
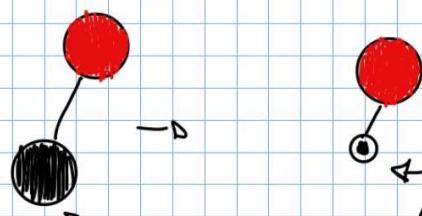
Abbiamo 2 casi di rimozione, in cui tolgo un nodo nero e il suo colore va gestito dandolo ad un altro nodo nero

CASO 1

padre rosso e figlio nero

elimino e il nodo null (proprietà 3 rilasciata)

e' doppio nero

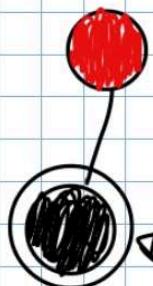


dovrò gestire il doppio nero

1.

CASO 2

padre e figlio neri



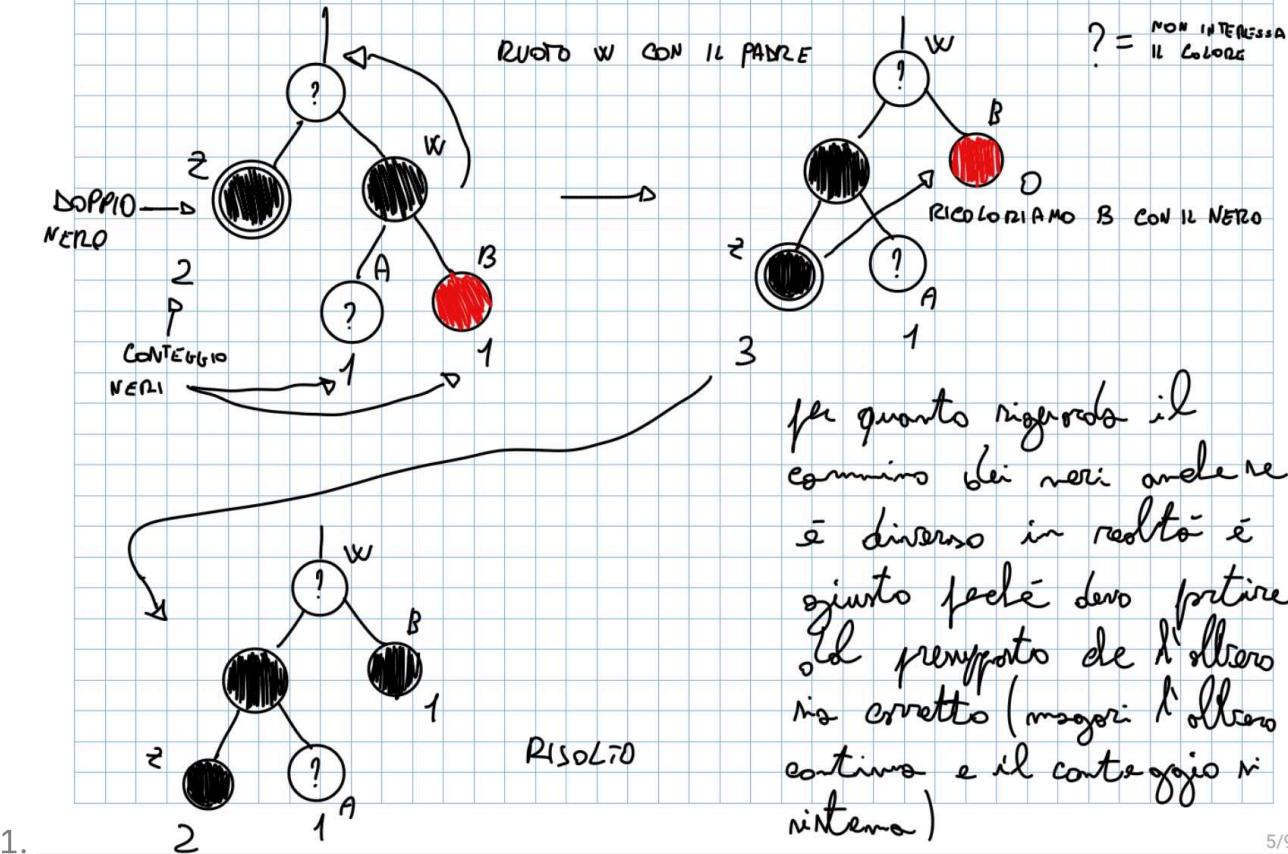
dovrò gestire il doppio nero

2.

Il nodo doppio nero è un nodo che contiene un flag che dice che lui è nero e contiene anche il colore di un altro nodo nero eliminato, questo doppio nero va gestito e ci sono 5 casi: 3 casi normali e 2 varianti dei due primi casi:

Prima di iniziare ricorda che quando si ruota si scambiano anche i colori
Userò le lettere per indicare i nodi

CASO 1 W NERO CON ALMENO 1 FIGLIO ROSSO

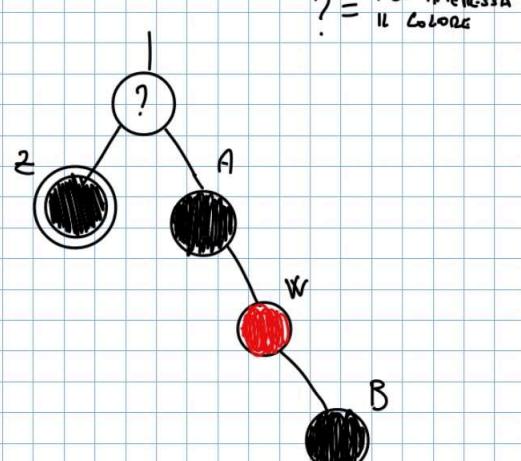
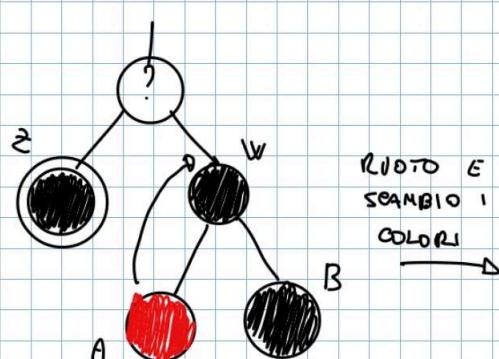


1.

VARIANTE CASO 1

B È NERO

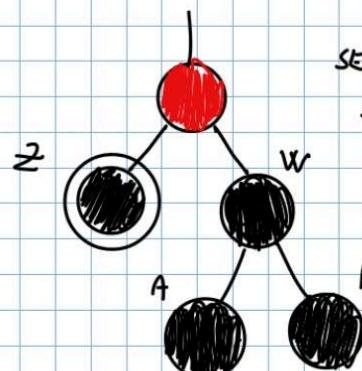
2.



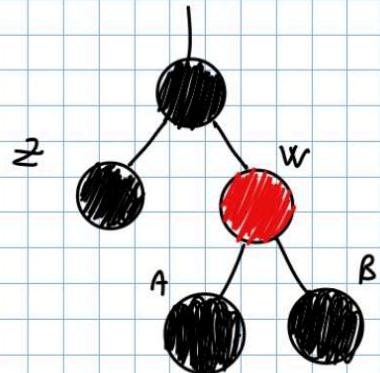
ADESSO MI RITROVO DI NUONO NEL CASO 1 NORMALE

CASO 2

W NERO CON 2 FIGLI NERI



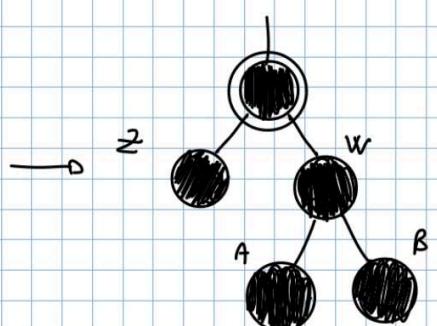
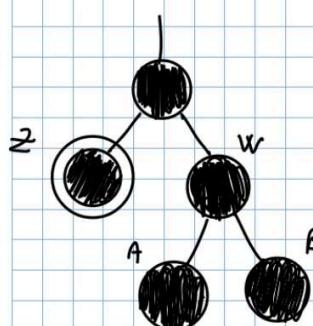
SE IL PADRE E' ROSSO
SCAMBIO I LORO COLORI
E HO FINITO



3.

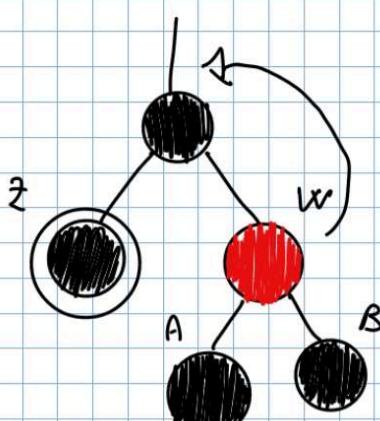
VARIANTE CASO 2

TUTTO NERO

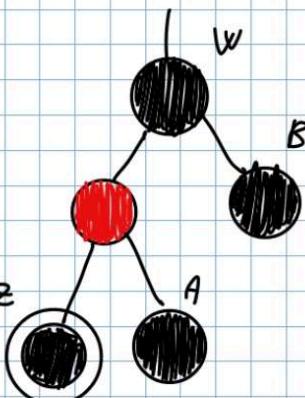


lo metto nella speranza
che altri modi risolvono
il problema
ne scrivo alla root,
la root elimina il
doppio nero e fa uno
altezza nera

4.

CASO 3 W E' ROSSO E NON HA FIGLIO NERO

ROTAZIONE A
SINISTRA



ho risolto col m' albo dei cr.

o 1 o 2

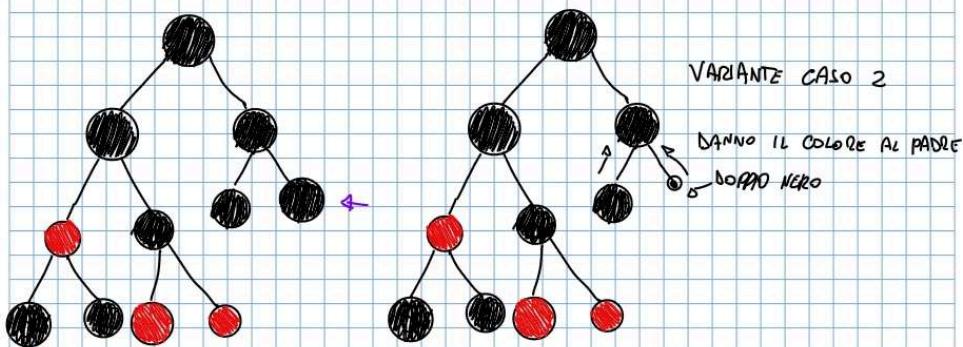
5.

Vediamo adesso 2 esempi: uno è l'eliminazione di vari nodi, l'altro è l'eliminazione del minimo

(Per capirli adeguatamente bisogna rifarli passo passo e usare questi per vedere se si è fatto bene)

ELIMINAZIONE NODO

→ = NODO DA PRIMVERE



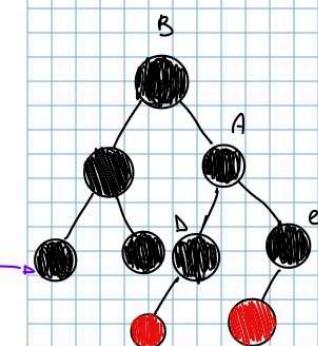
RUSTY
CON A

(VARIANTA
CASO 1 ? VERIFICARE)

R

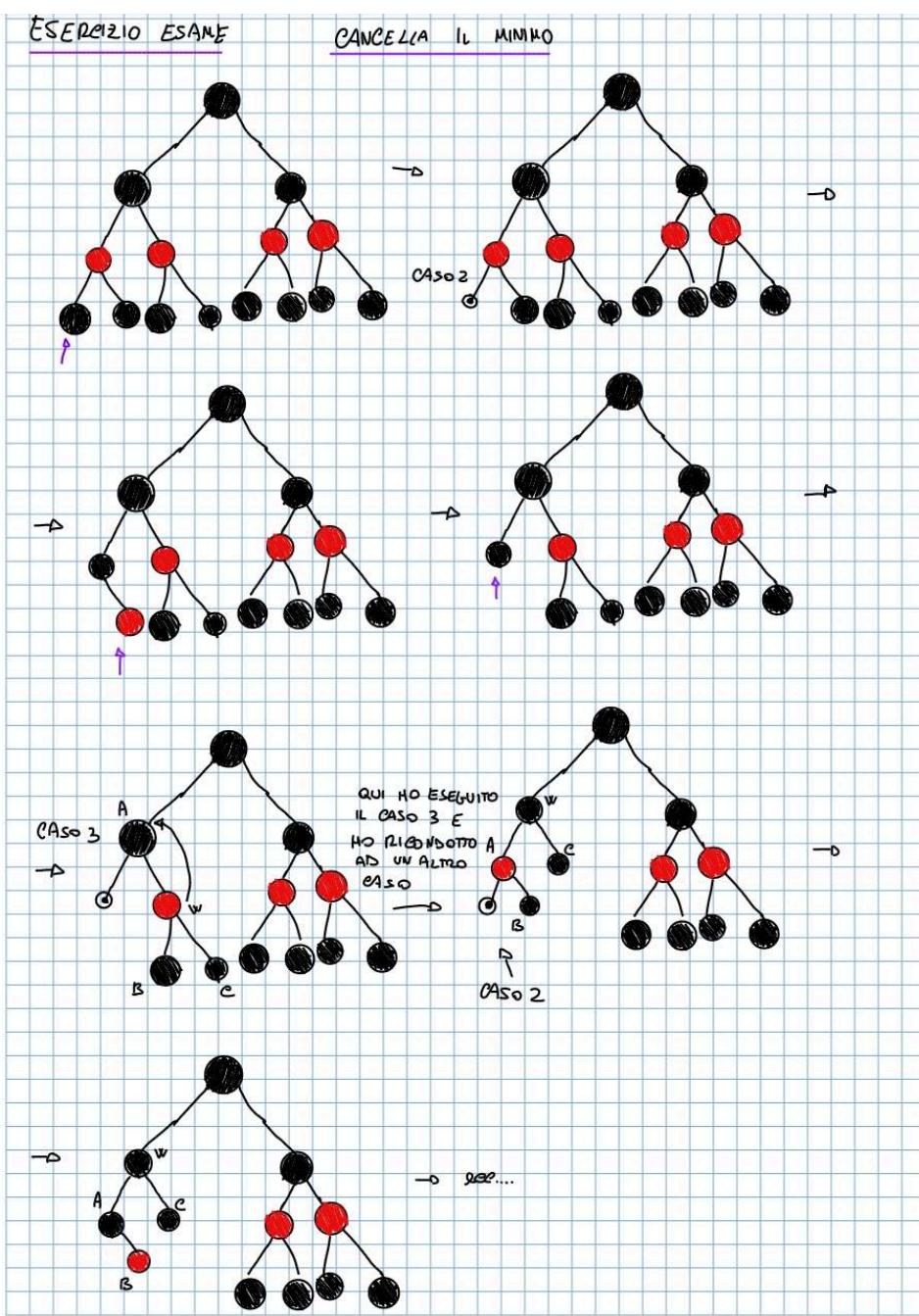
IL D PPI O NERO
D1 C LO DO AD
E
(con il conteggio
dei mesi è
interrato)

IL ROSSO NON DA
PROBLEMI

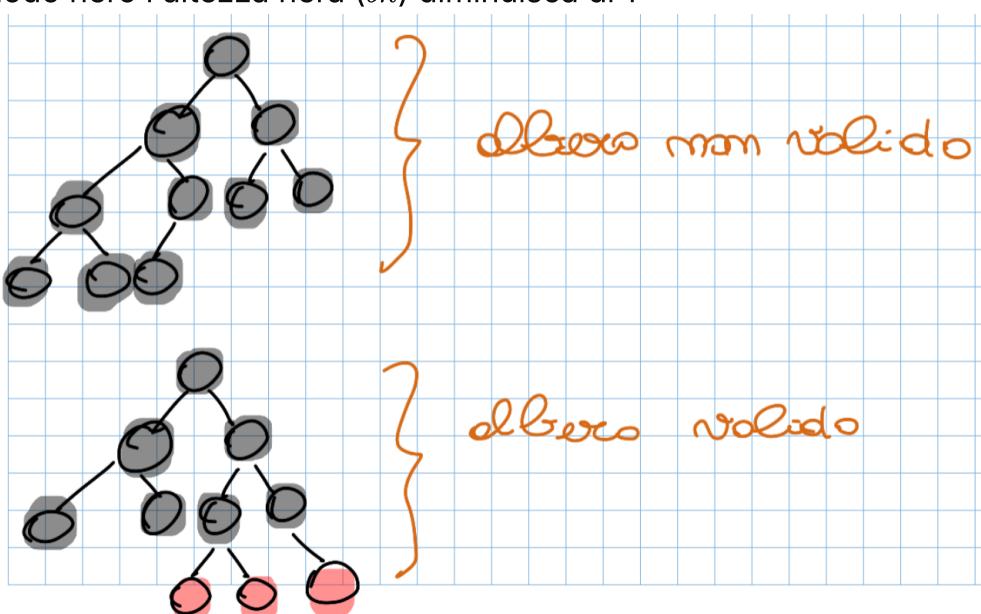


VADIANZE

1.

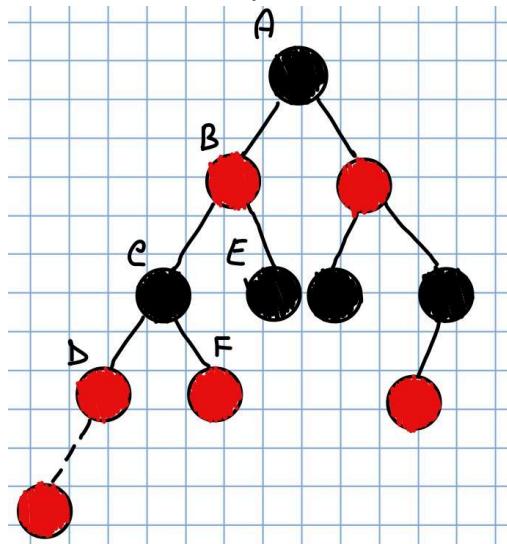


Fornire un esempio di albero rosso nero con 10 nodi tale che dopo aver eliminato un nodo nero l'altezza nera (hh) diminuisca di 1:



Il primo albero non è valido perché è un albero sbagliato, dato che non rispetta la regola 5 (il cammino dei neri uguale per tutti), e quindi giusto il secondo albero perché tiene conto della regola per avere una determinata altezza nera (in questo caso 3) che è $N_{neri} = 2^{bh} - 1 = 7$, infatti ci sono 7 nodi neri e gli ultimi 3 rossi che non influiscono sul conteggio dell'altezza nera, 7 è quindi il numero minimo di nodi per avere un'altezza nera di 3 e se ne venisse eliminato anche solo 1 l'albero dovrebbe ribilanciarsi diminuendo per forza il numero dell'altezza nera

Fornire un esempio di albero rosso nero in cui l'inserimento aumenta l'altezza nera



Il nodo tratteggiato che stiamo inserendo è ovviamente di colore rosso, questo porta ad una reazione a catena in cui si parte dal caso 3 dell'inserimento (ovvero i figli D e F scambiano il colore con il padre C), in questo modo C diventa rosso, qui non posso riapplicare il caso 3 perché E è nero quindi va applicata una rotazione e in questo modo riesco a propagare il "conflitto" fino alla radice dove viene espulso, in questo modo aumenta l'altezza nera

PROGRAMMAZIONE DINAMICA

Partiamo con il dire cosa è un problema di ottimizzazione:

Definizione

Un problema si dice di ottimizzazione quando esistono varie soluzioni ma solo alcune di queste sono le migliori, per capire qual è la migliore ad ogni soluzione applichiamo la funzione bontà che è definita così: $f : S \rightarrow R$, dove S è l'insieme delle soluzioni e R i numeri reali

Ad esempio nel problema dei cammini minimi nei grafi se parto da un nodo A potrei poter scegliere 2 percorsi diversi per arrivare a C: A-B-C o A-D-B-C, normalmente il primo cammino è la soluzione migliore, ma non è sempre così, il grado di bontà assegnato ad ogni soluzione è diverso in base al problema che si affronta

Adesso parliamo della programmazione dinamica

PROGRAMMAZIONE DINAMICA

La programmazione divide et impera divide il problema in *sotto problemi indipendenti* che va a risolvere ricorsivamente e poi unisce le soluzioni per risolvere il problema più grande, la programmazione dinamica invece è usata quando i *sotto problemi non sono indipendenti* (ovvero quando essi si ripetono più volte), in questo contesto un algoritmo divide et impera spreca molto tempo risolvendo dei casi che ha già risolto, mentre un algoritmo dinamico memorizza il risultato del sotto problema riconducendolo quindi ad un caso base, in questa maniera quando il sotto problema si ripresenterà verrà risolto immediatamente (quindi ogni sotto problema viene risolto una sola volta e salvato il risultato in una tabella).

La programmazione dinamica tipicamente si applica ai problemi di ottimizzazione
Il processo di costruzione di un algoritmo di programmazione dinamica può essere suddiviso in *4 fasi*:

1. Strutturazione

Gode della sottostruttura ottima? (la soluzione ottima del problema grande si ottiene combinando soluzioni ottime dei sottoproblemi)

Sovrapposizione dei sottoproblemi: gli stessi sottoproblemi compaiono più volte?
posso usare approccio ricorsivo?
se sì...

2. Definire una funzione ricorsiva per il calcolo della soluzione ottima

3. Costruire una procedura bottom-up (dai problemi piccoli a quelli più grandi) per il calcolo di una soluzione ottima

4. Opzionale - costruzione di una soluzione ottima

Vediamo dei problemi di ottimizzazione:

Problema del rod-cut

Abbiamo un' azienda che acquista barre di acciaio e le taglia in porzioni per poi rivenderle, si vuole sapere qual è il taglio ottimale per una barra di lunghezza i in modo da massimizzarne il ricavo:

lunghezza i	1	2	3	4	5	6	7	8	9	10
prezzo p_i	1	5	8	9	10	17	17	20	24	30

Ad esempio qual è il taglio ottimale per una barra lunga 4?

- Non tagliarla → guadagno = 9
- Tagliarla in 2 pezzi → guadagno = 10
- Tagliarla in un pezzo singolo e uno da 3 → guadagno = 9
- Tagliarla in 2 pezzi singoli e uno da 2 → guadagno = 7

Chiaramente la soluzione ottimale per il nostro problema è la seconda opzione che fa ricavare 10

Cerchiamo di risolverlo seguendo le 4 fasi della programmazione dinamica:

Fase 1 -Capire se ha sottostruttura ottima-

Strutturazione, il problema gode della sottostruttura ottima?

Possiamo dire che il problema $P(n)$ ha soluzione $S(n)$, la lunghezza totale della barra è n , possiamo quindi suddividere il problema in più sottoproblemi più piccoli, infatti la barra viene tagliata a lunghezza k avremo così la barra divisa in 2 una di lunghezza k e il resto sarà il totale n meno l'altra parte ovvero k , quindi il problema diventa:

$$P(n) = P(k) + P(n - k)$$

con $k < n$, $n - k < n$ e $1 \leq k \leq n$

Una soluzione ottima al problema generale è composta da soluzioni ottime di sottoproblemi?

Noi sappiamo che una soluzione $S(n) = S(k) + S(n - k)$, avremo che:

$S^*(n) = S(k) + S^*(n - k)$ dove $S^*(n)$ è una soluzione "buona" e $S^*(n - k)$ è una soluzione perfetta al sottoproblema $n - k$ ma ci manca la soluzione ottima di k , infatti: $S^{\#}(n) = S^*(k) + S^*(n - k)$, quindi abbiamo che $S^{\#}(n)$ è la soluzione ottima al problema perché è composta da soluzioni ottime degli altri sottoproblemi

Quindi: $S^*(n) \leq S^{\#}(n)$

Fase 2 -Definizione funzione ricorsiva-

Definiamo una funzione ricorsiva per il calcolo della soluzione ottima

$$r(i) = \begin{cases} 0 & \text{se } i = 0 \\ \max_{1 \leq k \leq i} (r(k) + r(i - k)) & \text{se } i \geq 1 \end{cases}$$

Dove i è la lunghezza della barra e k è il punto in cui taglio la barra

Questa funzione max calcola i valori di k da 1 fino ad i e poi sceglie il massimo (in pratica calcola la formula per ognuno, e alla fine tiene solo il risultato più grande, ricordo che k è dove la barra viene tagliata, quindi è come se provasse a tagliare in tutti i modi possibili e poi tiene solo il taglio che restituisce il ricavo massimo)

Fase 3 -Definizione di una procedura per il calcolo della soluzione ottima-

Come dice la fase 3 dobbiamo costruire una procedura bottom-up per il calcolo della soluzione ottima

Dato che si ripetono gli stessi sottoproblemi (come dice la prog. dinamica) usiamo un array per memorizzare il risultato di questi sottoproblemi in modo da risolverli una volta sola

Andrò spiegare nel dettaglio ogni riga della funzione

Abbiamo 2 array:

R[n] → qui andremo a mettere i risultati dei problemi già esaminati

P[i] → questo array contiene il ricavo in base alla lunghezza

1. Rod-cut(n)

```

2. `if n = 0 then return 0`  

3. `for i = 1 to n do`  

4.   `m = P[i]`  

5.   `for k = 1 to i - 1 do`  

6.     `if R[k] + R[i-k] > m then`  

7.       `m = R[k] + R[i-k]`  

8.   `R[i] = m`
```

riga 1 → alla funzione passo la lunghezza della sbarra (n)

riga 2 → questo è il caso base in cui la lunghezza è 0

riga 3 → è il ciclo principale che gestisce la funzione in maniera bottom-up, infatti parte dal basso (1) fino alla lunghezza della sbarra (n)

riga 4 → qui si inizializza m (che è il massimo provvisorio) con il prezzo della barra di lunghezza i , ovvero il prezzo della barra senza essere tagliata (che comunque potrebbe essere la soluzione ottimale)

riga 5 → questo ciclo identifica tutte le possibili posizioni in cui tagliare la barra (parte da 1 e si ferma a $i - 1$, perché i sarebbe la barra non tagliata, che abbiamo già tenuto in considerazione nella riga precedente)

riga 6 → qui ci chiediamo: la somma dei valori di questi prezzi ($R[k] + R[i-k]$) è maggiore di m ? Se si allora questa somma diventerà il mio nuovo massimo, e ricomincia il ciclo, ogni volta che la condizione nell'if è vera aggiornerà il nuovo massimo

riga 7 → aggiorna il massimo

riga 8 → finito il ciclo m conterrà il valore massimo per quella lunghezza della barra, il risultato viene salvato in R alla lunghezza i

Questa procedura ha complessità $O(n^2)$

Fase 4 -Costruzione di una soluzione ottima-

Costruzione di una soluzione ottima

Useremo praticamente la funzione del passo precedente a cui aggiungiamo un array k per segnare dove abbiamo tagliato la sbarra in corrispondenza del ricavo massimo (prima trovavamo solo il ricavo massimo) e creeremo una funzione print

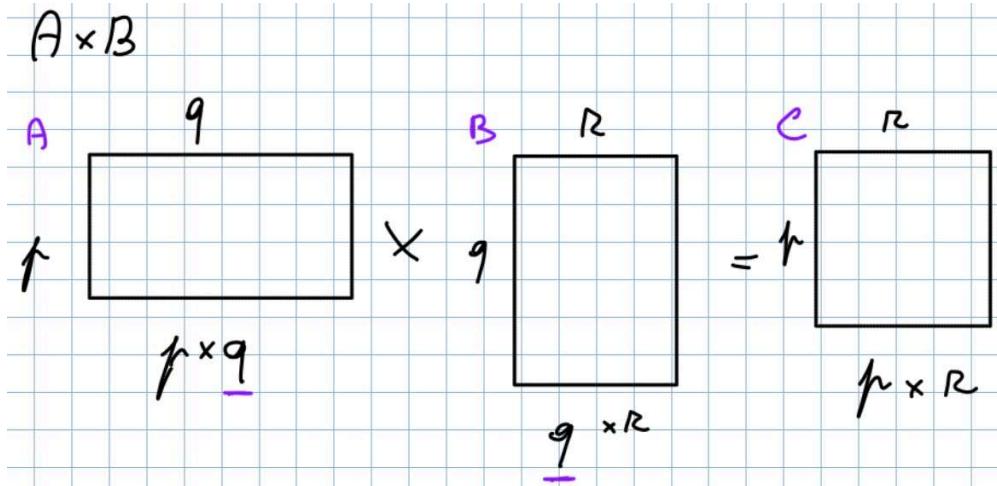
```
Rod-cut(n)
k = newArray(n)
if n = 0 then return 0
for i = 1 to n do
m = P[i]
k[i] = i
for k = 1 to i - 1 do
if R[k] + R[i-k] > m then
m = R[k] + R[i-k]
k[i] = k
R[i] = m
```

Le uniche aggiunte sono appunto questo array che salva in posizione i (la lunghezza della barra) il punto k in cui è stata tagliata

```
print-cut(n,k)
if k[n] = n then
print(n)
else
print-cut(k[n],k)
print-cut(n-k[n],k)
```

Problema della moltiplicazione tra matrici

In algebra due matrici si possono moltiplicare se il numero di colonne della prima è uguale al numero di righe della seconda e la matrice risultante avrà come righe il numero di righe della prima e come colonne il numero di colonne della seconda.



Pensiamo quindi ad una funzione che moltiplichi 2 matrici:

1. `matrix-multiply(A,B,p,q,r)`
2. ``C = newMatrix(p,r)``

3. `for i = 1 to p do`
4. `for j = 1 to r do`
5. `C[i,j] = 0`
6. `For k = 1 to q do`
7. `C[i,j] = C[i,j] + A[i,k]xB[k,j]`
8. `return C`

Questa funzione non tiene conto della *parentesizzazione*, quindi funziona ma è poco efficiente

Il numero totale di operazioni svolte durante una moltiplicazione tra matrici è dato da: $p \times q \times r$

ovvero righe della prima(p), colonne della prima e righe della seconda(q) e colonne della seconda(r)

Cosa è la parentesizzazione?

Le matrici godono della proprietà associativa, quindi in sostanza: cambiando l'ordine delle moltiplicazioni il risultato finale non cambia, ma cambia il numero di operazioni che facciamo per moltiplicarle, ad esempio:

$\begin{pmatrix} A & \times & B \\ 10 \times 100 & & 100 \times 5 \end{pmatrix} \times C \quad 5 \times 50$ <p style="text-align: center;">↓</p> $AB \quad 10 \times 100 \times 5 = 5000$ <p style="text-align: center;">↓</p> $ABC \quad 10 \times 5 \times 50 = 2500$ <p style="text-align: center;">↓</p> $\text{TOT OPERAZIONI} = 7500$	$A \quad 10 \times 100 \quad \times \begin{pmatrix} B & \times & C \\ 100 \times 5 & & 5 \times 50 \end{pmatrix}$ <p style="text-align: center;">↓</p> $BC \quad 100 \times 5 \times 50 = 25000$ <p style="text-align: center;">↓</p> $BCA \quad 10 \times 100 \times 50 = 50000$ <p style="text-align: center;">↓</p> $\text{TOT OPERAZIONI} = 75000$
---	--

10 volte di più

Come possiamo vedere il numero di operazioni è significativamente diverso tra i 2 modi di moltiplicare, possiamo perciò dire che la parentesizzazione è il modo di aggregare le moltiplicazioni. A noi ovviamente interessa quella che ci fa fare meno moltiplicazioni

Fase 1 -Capire se ha sottostruttura ottima-

la generica matrice i ha righe

A_i

$P_{i-1} \times P_i$

$P_{i-1} \times P_i$

questo è la moltiplicazione delle matrici dalla fine all'ultima

diciamo la moltiplicazione, moltiplicando da A_1 fino ad A_k e da A_{k+1} fino a A_m

$A_{1,n} = A_{1,k} \times A_{k+1,n}$

numero di moltiplicazioni
1 + $A_{1,k} \times A_{k+1,n}$

$S_{1,n}^* = S_{1,k}^* + S_{k+1,n}^* + P_0 P_k P_n$

la soluzione ottima è la moltiplicazione tra la sol. ottima della matrice $A_{1,k} \times A_{k+1,n}$

$S_{1,n}^*$ è il numero di moltiplicazioni che devo fare

In pratica parentesizziamo nella maniera migliore per ottenere la sol. ottima

Fase 2 -Definizione funzione ricorsiva-

$$S_{1,n} = \begin{cases} \emptyset & \text{se } n = 1 \\ \min_{1 \leq k < n} (S_{1,k} + S_{k+1,n} + P_0 P_k P_n) & \text{se } n > 1 \end{cases}$$

In sostanza il caso base restituisce insieme vuoto perché se la matrice è una non fa la moltiplicazione mentre se le matrici sono più di una allora: per ogni k che va da 1 fino a $n - 1$ calcola la somma della soluzione $S_{1,k}$ e $S_{k+1,n}$ e somma anche $P_0 P_k P_n$ che è il numero di moltiplicazioni che si fanno, di tutti questi calcoli sceglie il minimo (in poche parole controlla ogni singola **parentesizzazione** che si può fare, spostando la k a mano a mano, e sceglie il minimo tra tutti i calcoli fatti, ovvero la **parentesizzazione** migliore) Generalizzando:

$$S_{i,j} = \begin{cases} \emptyset & \text{se } i = j \\ \min_{i \leq k < j} (S_{i,k} + S_{k+1,j} + p_{i-1} p_k p_j) & \text{se } i < j \end{cases}$$

Fase 3 -Definizione di una procedura per il calcolo della soluzione ottima-

Il problema grande se risolto con una procedura ricorsiva pura (divide et impera) porterebbe ad avere molti sottoproblemi uguali, quindi devo usare la memorizzazione, non attraverso un normale array ma attraverso una matrice. Utilizziamo un approccio bottom-up

Data una matrice S:

		5 →							
			1	2	3	4	5	6	7
↓	i	0	7
	2	0	6
3		0	5
4			0	4
5				0	3
6					0	.	.	.	2
7						0	.	.	1

- Le righe i indicano l'inizio della catena di matrici che prendo in considerazione, mentre le colonne j indicano la fine
- $S[i, j]$ è il costo minimo per moltiplicare le matrici da A_i fino ad A_j
Esempio: la cella alla riga 2 e colonna 5 ($S[2, 5]$), lì dentro scriveremo il costo minimo per fare $A_2 \cdot A_3 \cdot A_4 \cdot A_5$
- La diagonale principale è composta da 0 perché la catena è formata da una sola matrice e quindi il costo è 0
- Pattern di risoluzione: la matrice non si riempie a caso ma dato che prende in considerazione catene che vanno dalla matrice i che ha indice più piccolo della matrice j che ha indice più grande ($i < j$) la matrice si riempirà solo sopra la diagonale e si riempirà seguendo la lunghezza l della catena:
 - $l = 1$ (Lunghezza 1): Sono le singole matrici (A_1, A_2, \dots). Costo 0. È la diagonale principale che abbiamo già riempito.
 - $l = 2$ (Lunghezza 2): Ora calcoliamo il costo per moltiplicare coppie di matrici vicine: $(A_1 A_2), (A_2 A_3), (A_3 A_4) \dots$ Questi valori vanno nella diagonale subito sopra quella degli zeri.
 - $l = 3$ (Lunghezza 3): Ora calcoliamo catene di 3 matrici: $(A_1 A_2 A_3), (A_2 A_3 A_4) \dots$
Per calcolare il costo di $A_1 \dots A_3$, l'algoritmo guarda i calcoli fatti al passo precedente (lunghezza 2) che sono già scritti nella tabella. Non deve ricalcolarli
 - ...e così via fino a $l = n$
- La soluzione ottima è quella posta nell'ultima casella

	Da Sapere						
1 →	$A_{1,1}$	$A_{2,2}$	$A_{3,3}$	$A_{4,4}$	$A_{5,5}$	$A_{6,6}$	$A_{7,7}$
2 →	$A_{1,2}$	$A_{2,3}$	$A_{3,4}$	$A_{4,5}$	$A_{5,6}$	$A_{6,7}$	
3 →	$A_{1,3}$	$A_{2,4}$	$A_{3,5}$	$A_{4,6}$	$A_{5,7}$		
Pattern di risoluzione	4 →	$A_{1,4}$	$A_{2,5}$	$A_{3,6}$	$A_{4,7}$		
	5 →	$A_{1,5}$	$A_{2,6}$	$A_{3,7}$			
	6 →	$A_{1,6}$	$A_{2,7}$				
	7 →	$A_{1,7}$					

Inizia da 1 e finisce a 7 con la soluzione ottima

Scriviamo la procedura e descriverò ogni riga

```

1. matrix-chain-order(p,n) (mco)
2.   `S = newMatrix(n,n)`
3.   `for i = 1 to n do S[i,i] = 0`
4.   `for l = 2 to n do`
5.     `for i = 1 to n-l-1 do`
6.       `j = i + l - 1`
7.       `S[i,j] = +∞`
8.       `for k = 1 to j-1 do`
9.         `if S[i,j] > S[i,k] + S[k+1,j] + p[i-1]* p[k]* p[j]  then`
10.           `S[i,j] = S[i,k] + S[k+1,j] + p[i-1]* p[k]* p[j]`
11. return S[1,n]

```

riga 1 → dichiarazione della funzione che prende in input un vettore p con le dimensioni delle matrici e il numero di matrici (n)

riga 2 → creazione della matrice che contiene le soluzioni (S)

riga 3 → inizializza a 0 la diagonale della matrice S che indica il costo nullo

riga 4 → questo for esterno indica la lunghezza (risolve prima tutte le catene di 2 matrici poi quelle da 3, 4, 5 ecc...)

riga 5 → questo for indica la *i* da dove si parte ovvero l'inizio della catena

riga 6 → calcola l'indice di fine della catena

riga 7 → inizializziamo a $+\infty$, perché dobbiamo trovare un valore molto basse quindi lo inizializziamo ad un numero altissimo così che non possa essere scambiato per il minimo

riga 8 → questo for è quello che parentesizza la catena, quindi prova tutti i possibili k, ad esempio per $A_1 \dots A_4$, prova a tagliare dopo la prima matrice ($A_1|A_2A_3A_4$), dopo la seconda ($A_1A_2|A_3A_4$)

riga 9 → controlla se il taglio effettuato al punto k costo meno di quello trovato fino a quel momento, è composto da 3 parti: $S[i, k]$: Costo ottimale della parte sinistra (già calcolato e salvato in memoria), $S[k+1, j]$: Costo ottimale della parte destra (già calcolato e salvato),

$p[i-1]*p[k]*p[j]$: Costo per moltiplicare le due matrici risultanti

riga 10 → se il nuovo valore è più piccolo del precedente aggiorna il nuovo minimo

riga 11 → ritorna il minimo assoluto (che è soluzione ottima e si troverà nella cella all'angolo in alto a destra)

Questa procedura ha complessità $O(n^3)$

Questo algoritmo calcola il *costo minimo della soluzione* ma non abbiamo modo di sapere quale sia l'ordine effettivo delle matrici da moltiplicare.

Fase 4 -Costruzione di una soluzione ottima-

Molti algoritmi richiedono di trovare il valore della soluzione ottima, quindi ci potremmo fermare al terzo step, in questo caso però abbiamo bisogno anche ricostruire la parentesizzazione della soluzione ottima in modo da poter effettuare la vera e propria moltiplicazione, da questa esigenza nasce lo step successivo, solitamente opzionale.

```

matrix-chain-order(p,n) (mco)
S = newMatrix(n,n)
D = newMatrix(n,n)
for i = 1 to n do S[i,i] = 0
for l = 2 to n do
for i = 1 to n-l+1 do
j = i + l - 1
S[i,j] = +∞
for k = 1 to j-1 do
if S[i,j] > S[i,k] + S[k+1,j] + p[i-1]* p[k]* p[j] then
S[i,j] = S[i,k] + S[k+1,j] + p[i-1]* p[k]* p[j]
D[i,j] = k
return S[1,n]

print-chain(D,i,j)
if i = j then
print(A_i)
else
k = D[i,j]

```

```

print "("
print-chain(D,i,k)
print-chain(D,k+1,j)
print ")"

```

Mentre l'algoritmo che è rimasto essenzialmente lo stesso si occupa di esplorare e trovare tutti i possibili modi di parentesizzare la catena, la matrice $D[i, j]$ salva esattamente quale indice k ha prodotto la parentesizzazione migliore. La funzione print-chain serve per stampare la parentesizzazione migliore in base ai parametri passati.

Problema della distanza di editing tra 2 stringhe

Questo problema analizza la distanza di editing tra 2 stringhe, ovvero la "differenza" tra le 2, ad esempio casa e chiesa hanno una distanza di editing di 4 perché la differenza è "hie" + "a"

In una stringa possiamo apportare 3 operazioni:

1. Inserimento di un carattere → casa → casta → distanza pari a 1
2. Cancellazione di un carattere → casta → casa → distanza pari a 1
3. Sostituzione di un carattere → casta → vasta → distanza pari a 1

Esistono altre operazioni come lo swap, l'inversione ecc... ma queste 3 sono le più importanti

Come passo da casa a chiese?

casa → chasa → chesa → chiesa → chiese

Ho effettuato 3 inserimenti e 1 sostituzione

Avrei potuto anche cancellare e riscrivere la parola ma avrebbe preso più tempo, e infatti quando la stringa è piccola posso ignorare l'efficienza ma quando la stringa diventa molto grande devo ottimizzare

Fase 1 - Sottostruttura ottima-

Vogliamo trasformare la stringa X in Y e sappiamo che:

$X_i = X_1, X_2, X_3 \dots X_i$ e $Y_i = Y_1, Y_2, Y_3 \dots Y_j$

$|X| = n$, $|Y| = m$

sappiamo anche che: $X_i = X_{i-1}X_i$ e la stessa cosa per Y.

Per capire se il problema gode della sottostruttura ottima dobbiamo analizzare 2 casi:

1. $X_i = Y_j \rightarrow X_{i-1}Y_{j-1}$

Se l'ultimo carattere di X e di Y è uguale il problema si riduce di 1

2. $X_i \neq Y_j$

In questo caso devo scegliere quale delle 3 operazioni fare (inserimento, cancellazione, sostituzione):

- $ED(X_{i-1}, Y_{j-1}) +$ sostituzione di X_i con Y_j
- $ED(X_{i-1}, Y_j) +$ cancellazione di X_i

- $ED(X_i, Y_{j-1}) + \text{inserimento di } Y_j \text{ alla fine di } X$
 ED sta per una funzione che calcola la distanza di editing

Fase 2 -Definizione funzione ricorsiva-

$$ED(i, j) = \begin{cases} i & \text{se } j = 0 \\ j & \text{se } i = 0 \\ ED(i - 1, j - 1) & \text{se } X[i] = Y[j] \quad (\text{l'ultimo car-}) \\ \min \left(\underbrace{ED(i - 1, j)}_{\text{cancellazione}}, \underbrace{ED(i, j - 1)}_{\text{inserimento}}, \underbrace{ED(i - 1, j - 1)}_{\text{sostituzione}} \right) + 1 & \text{se } X[i] \neq Y[j] \end{cases}$$

scelgo il costo minimo tra le 3 operazioni

CASI BASE
(quando 1 delle 2)

i e j nei casi base indicano una delle 2 stringhe vuote, quindi se devo trasformare la stringa i nella stringa j o viceversa dovrò fare i operazioni o j operazioni (per ricopiarla)

Analizzando una possibile funzione ricorsiva pura (top-down) ci accorgiamo che ci sono vari sottoproblemi che si ripetono, perciò usare questo approccio è inefficiente

Fase 3 -Definizione di una procedura per il calcolo della soluzione ottima-

Useremo una matrice per rappresentare la distanza di editing in cui ogni numero rappresenta la distanza di editing tra le 2 stringhe

1. **EDT(x, y, n, m)**
2. `ED = new matrix(n + 1, m + 1)`
3. `for i = 0 to n do`
4. `ED[i, 0] = i`
5. `for j = 1 to m do`
6. `ED[0, j] = j`
7. `for i = 1 to n do`
8. `for j = 1 to m do`

```

9.      `if (x[i] = y[j]) then`  

10.     `ED[i, j] = ED[i - 1, j - 1]`  

11.     `else`  

12.     `ED[i, j] = min(ED[i, j - 1], ED[i - 1, j], ED[i - 1, j -  

    1]) + 1`  

13.     `return ED[n, m]`  


```

riga 1 → Definizione della funzione che prende in input le stringhe x e y con la loro dimensione

riga 2 → crea una matrice di dimensione $n+1$ e $m+1$ (j sono le colonne i le righe)

riga 3-6 → rappresentano i casi base in cui una o tutte e 2 le stringhe sono vuote

riga 7-8 → questi 2 cicli for servono a scorrere la matrice

riga 9 → l'if controlla se siamo nel caso in cui i 2 ultimi caratteri siano uguali (secondo caso nella funzione ricorsiva) e...

riga 10 → se l'if è vero copia il numero della casella precedente (diagonale) in quella corrente perché i 2 caratteri nelle stringhe sono uguali quindi ricopio la distanza di editing perché non è cambiata

riga 11-12 → altrimenti (se i 2 caratteri sono diversi) all'interno della cella corrente mette il numero minimo tra le celle vicine che rappresentano inserimento cancellazione e sostituzione, al numero nella cella aggiungo 1 perché ho fatto un operazione in più

riga 13 → restituisce il numero scritto nell'ultima casella in basso a sinistra che rappresenta la soluzione ottima

La complessità temporale e di memoria di questa funzione è $O(n \times m)$

Esempio con la matrice fatta a lezione:

	C	A	S	A	
-	0	1	2	3	4
C	0 1	0 1	2	3	4
H	2 2	1 1	1 1	2 2	3
I	3 3	2 2	2 2	2 2	3
E	4 4	3 3	3 3	3 3	3
S	5 5	4 4	4 4	3 3	1
E	6	5	5	4	1

Fase 4 -Costruzione di una soluzione ottima-

```

Print-EDT(ed, x, y, n, m)
i = n
j = m
while (ed[i, j] > 0) do
if (x[i] = y[j]) then
i = i - 1
j = j - 1
else
e = min(ed[i, j - 1], ed[i - 1, j], ed[i - 1, j - 1])
if (e = ed[i, j - 1]) then
j = j - 1 → stampiamo ins(y[j])
else if (e = ed[i - 1, j]) then
i = i - 1 → stampiamo canc(x[i])
else
i = i - 1 → stampiamo sost(x[i],y[j])
j = j - 1

```

Trovare la sottostringa più lunga di tutte (veloce)

Questa è una sottostringa: X = a~~gator~~ Y = b~~gatop~~

Fase 1 -Sottostruttura-

Il problema gode di una sottostruttura ottima perché se i caratteri sono uguali è necessario ridurre pian piano l'indice del carattere in cui ci troviamo ($\text{LCS}(x_{i-1}, y_{j-1}) + 1$)

Fase 2 -Definizione di una funzione ricorsiva-

$$\text{LCS}(i, j) = \begin{cases} 0 & \text{se } i = 0 \text{ oppure } j = 0 \\ 0 & \text{se } X[i] \neq Y[j] \\ \text{LCS}(i - 1, j - 1) + 1 & \text{se } X[i] = Y[j] \end{cases}$$

Fase 3 -Costruzione di una procedura per il calcolo dell *longest common substring*-

Scrivo una procedura ottimizzata che usa solo 2 array anziché una matrice

```

1. LCS(X, Y, n, m)

2.   `LCS_i-1 = newArray(m)`

3.   `LCS_i = newArray(n)`

4.   `for j = 0 to m do` 

5.     `LCS_i-1[j] = 0` 

6.   `for i = 1 to n do` 

7.     `LCS_i[0] = 0` 

8.     `for j = 1 to m do` 

9.       `LCS_i[j] = Ø` 

10.      `if X[i] = Y[j] then` 

11.        `LCS_i[j] = LCS_i-1[j-1]+1` 

12.      `if M < LCS_i[j] then` 

13.        `M = LCS_i[j]` 

14.   `LCS_i-1 = LCS_i`
```

riga 1 → dichiarazione della funzione che prende in input le 2 stringhe e le dimensioni

riga 2-3 → dato che è una versione ottimizzata non creo una matrice ma uso solo 2 array per memorizzare la riga precedente (*i-1*) e la riga corrente

riga 4-5 → inizializza l'array della riga precedente a 0 perché prima di iniziare la sottostringa comune è sempre vuota

riga 6 → inizia un ciclo che scorre la stringa X dall'inizio alla fine

riga 7 → gestisce il caso limite in cui non si può avere una corrispondenza prima del primo carattere

riga 8 → per ogni carattere della stringa X scorre tutta la stringa Y

riga 9 → inizializza a 0 (o insieme vuoto) perché a differenza delle "sottosequenze", nelle "sottostringhe" se due caratteri non sono uguali, la catena si spezza e la conta deve ripartire da zero

riga 10-11 → controlla se il carattere è uguale tra X e Y e se si allora prende il valore della riga precedente gli aggiunge 1 e lo mette come valore della riga corrente (perché ha trovato una corrispondenza)

riga 12-13 → confronta il valore di M (massima sequenza trovata) con sostanzialmente la nuova sequenza che ha appena trovato e se è maggiore di M, M viene aggiornato come nuovo massimo

riga 14 → una volta che è stata controllata la riga "corrente" questa diventerà la riga precedente

Vediamo un esempio:

	6	6	C	C	A	T
0	0	0	0	0	0	0
1	0	0	0	0	1	1
T	0	0	0	0	0	0
G	0	1	1	0	0	0
C	0	0	0	2	1	0
C	0	0	0	1	3	0
A	0	0	0	0	0	4
T	0	0	0	0	0	0

La sottostringa andrà a formarsi sulla diagonale, ogni volta che trova un carattere corrispondente prende il valore della diagonale nella cella precedente e gli somma 1 (perché ha appena trovato un carattere corrispondente), la matrice viene analizzata

scorrendo le colonne sulle righe in questo modo: inizia dalla **G** e confronta con ogni carattere di **ATGCCAT**, G-A scrive 0, G-T scrive 0, G-G scrive 1 perché il carattere combacia poi scorre fino alla fine senza trovare altri caratteri uguali, poi passa all'altra G che fa la stessa cosa, poi passa alla **C** che scrive sempre 0 fin quando non trova la prima C dell'altra stringa, quindi prende il valore di prima dalla diagonale (1) e gli somma 1 così diventa 2, abbiamo appena trovato la sottostringa più lunga fino ad adesso, poi continua con gli altri caratteri finché non finisce.

Adesso vediamone una variante chiamata **longest common subsequence**

Longest common subsequence

La differenza con le sottostringhe è che possono includere dei caratteri nel mezzo:

X = ACGAAT

Y = CCATAG

E' comunque un problema simile a quello della sottostringa infatti la funzione ricorsiva è simile:

$$LCS(i, j) = \begin{cases} 0 & \text{se } i = 0 \text{ oppure } j = 0 \\ LCS(i - 1, j - 1) + 1 & \text{se } x[i] = y[j] \\ \max(LCS(i - 1, j), LCS(i, j - 1)) & \text{se } x[i] \neq y[j] \end{cases}$$

Scelgo quindi il max tra cancellazione e inserimento

ALGORITMI GREEDY (GOLOSI)

Per alcuni problemi di ottimizzazione è uno spreco utilizzare la programmazione dinamica andando a fare le migliori scelte possibili, spesso conviene affidarsi ad algoritmi più semplici chiamati algoritmi golosi o greedy. Un algoritmo greedy fa sempre la scelta migliore in quel determinato momento ovvero fa una scelta *localmente ottima* nella speranza che essa porterà ad una *soluzione globalmente ottima*. Gli algoritmi golosi possono essere applicati ad una vasta quantità di problemi di ottimizzazione, ne vedremo alcuni:

Compressione di Huffman

Questo è un problema che riguarda la compressione dei dati all'interno di un file per far risultare il file con una dimensione minore.

Supponiamo di avere un testo T composto da 100 caratteri:

Il nostro alfabeto Σ :

- a = 20 → numero di volte che si ripete il carattere
- b = 7
- c = 3
- d = 10
- e = 15
- f = 1

- g = 30
- h = 4
- i = 5
- l = 5

Supponiamo che in questa configurazione ogni carattere occupi 10 bit di spazio, quindi per il testo (100 caratteri) occuperà : $100 \times 10 = 1000$ bit, per testi piccoli non è un problema ma con testi molto lunghi lo spazio occupato diventa un problema

La soluzione a questo potrebbe essere assegnare più bit ai caratteri meno frequenti e meno bit a quelli più frequenti in modo da risparmiare spazio a discapito però di una codifica e decodifica più impegnative.

Ma bisogna stare attenti a usare bene i *prefissi*, perché ogni carattere deve iniziare con una sequenza di bit che sia univoca in modo che non ci siano ambiguità quando il computer dovrà decodificarli:

Ad esempio:

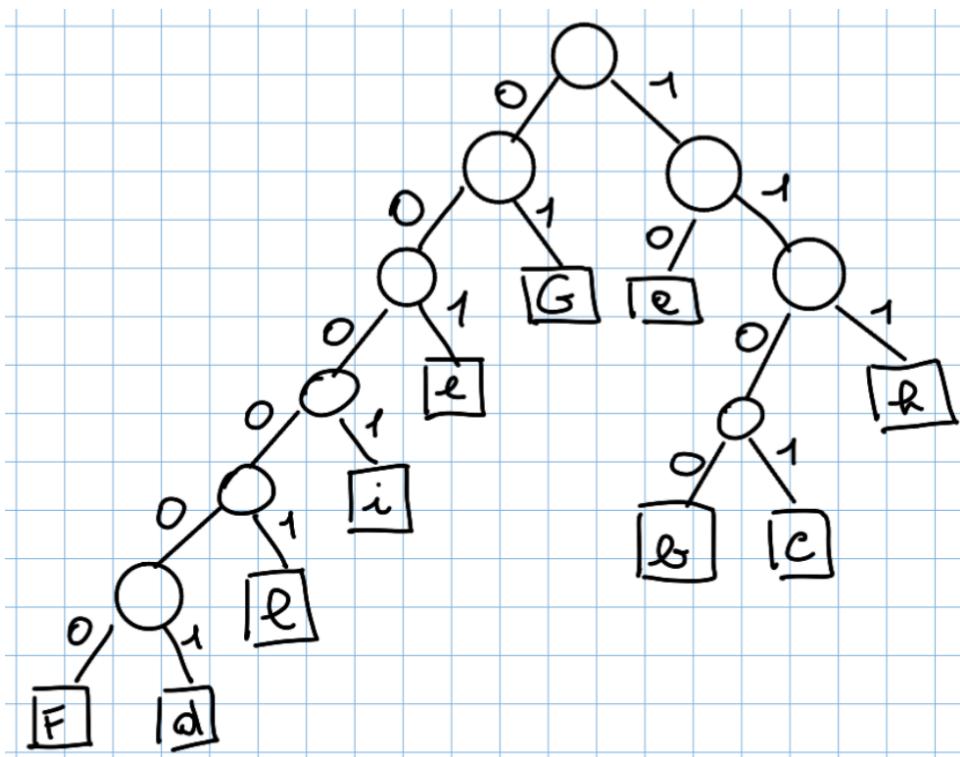
Carattere	Codifica Semplice	Codifica con prefissi
a = 20	0000	10
b = 7	0001	1100
c = 3	0010	1101
d = 10	0011	000001
e = 15	0100	001
f = 1	0101	000000
g = 30	0110	01
h = 4	0111	111
i = 5	1000	0001
l = 5	1001	00001
Testo = 100	400 Bit	298 bit

Utilizzando questa codifica siamo riusciti ad abbassare di molto lo spazio utilizzato, anche se è stato difficile codificare tutto stando attento ai prefissi.

Questa codifica con i prefissi non è sempre migliore di quella a 4 bit perché se ho un carattere come ad esempio la f o la d occupo 6 bit anziché 4, il guadagno sta nel fatto che queste lettere appaiono di meno rispetto alla g ad esempio e quindi visto che la g occupa solo 2 bit nel complesso di un testo vario ci vado a guadagnare memoria

Huffman ci da una mano a creare una codifica con i prefissi utilizzando un albero

Questo è l'albero della nostra codifica con prefissi:



Per leggerlo si parte dalla root e si decide: a = 10 quindi vado a destra da lì scendo a sinistra con lo 0 e ho trovato la a

L'obiettivo è quindi trovare l'albero che codifica meglio i nostri caratteri

$\forall c \in \Sigma$ abbiamo che: $f(c)$ = frequenza, $d_T(c)$ = profondità di quel carattere nell'albero

Per valutare l'albero usiamo questa funzione:

$$b(T) = \sum_{c \in \Sigma} f(c) * d_T(c)$$

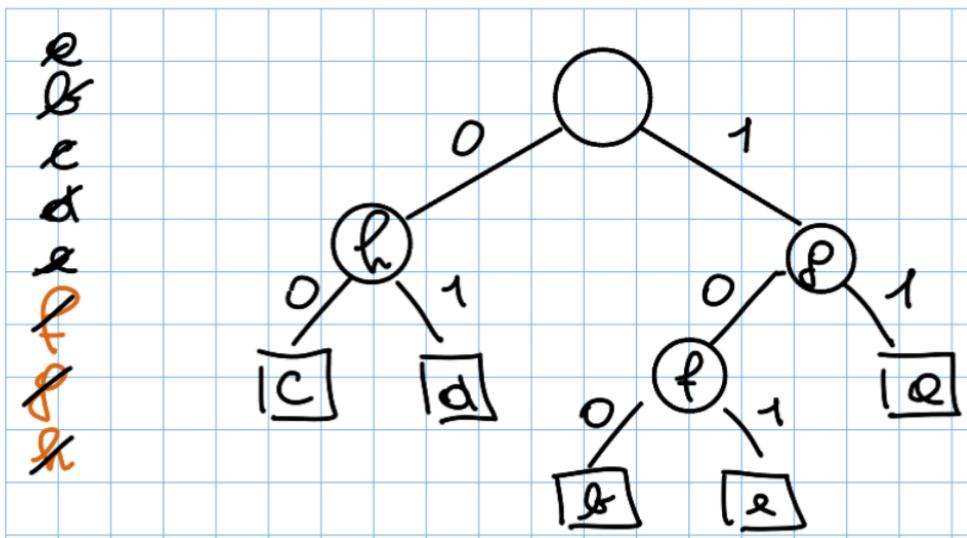
Il valore di bontà $b(T)$ è dato dalla **sommatoria**, per ogni carattere c appartenente all'alfabeto Σ del prodotto tra la **frequenza** $f(c)$ del carattere e la **profondità nell'albero** $d_T(c)$ (che sarebbe la lunghezza del codice del carattere)

Dobbiamo trovare una strategia che minimizza $b(T)$ (algoritmo di Huffman)

Lo faremo ricorsivamente utilizzando una strategia greedy

L'algoritmo di Huffman parte dalle foglie, quindi :

- **Prende i due caratteri che appaiono meno volte** di tutti.
- **Li unisce**: Li fonde insieme creando un nuovo nodo (il padre dei 2 caratteri) che pesa quanto la somma dei due.
- **Li allontana**: Dato che li ha uniti subito, questi due caratteri finiranno nel punto più profondo dell'albero. Significa che avranno il codice più lungo.
- **Ripete**: Ora considera il nuovo gruppo (padre e i 2 figli) come se fosse un singolo oggetto e ricomincia dal punto 2



Unisce C e D in H, C e D avranno codice più lungo rispetto ad H fa le stesse cose con gli altri caratteri, dato che unisce 2 caratteri in 1 l'albero finisce quando non ci sono più caratteri, e arriva alla root

Dimostrazione sotto struttura ottima

Definizioni iniziali:

- Σ : Insieme dei caratteri (la soluzione a questo è T)
- Consideriamo due nodi a e b (le foglie con frequenza minima) e il loro genitore z.
- $\Sigma' = \Sigma - \{a, b\} \cup \{z\}$ (la soluzione a questo è T')

Relazioni tra frequenze e profondità:

$$1. f(z) = f(a) + f(b)$$

- La frequenza del nodo padre è la somma delle frequenze dei figli.

$$2. d_T(a) = d_{T'}(z) + 1 \text{ e } d_T(b) = d_{T'}(z) + 1$$

- La profondità di a/b nell'albero originale (T) è la profondità di z nell'albero ridotto (T') più 1. **Costruzione di una relazione tra T e T'**: Il costo dell'albero T, denotato come $b(T)$, è dato, come detto prima, dalla somma delle frequenze per le profondità: $b(T) = \sum_{c \in \Sigma} f(c) * d_T(c)$

3. Espandendo la sommatoria per mettere in relazione T con T':

$$b(T) = [\sum_{c \in \Sigma'} f(c) * d_{T'}(c)] - f(z) \cdot d_{T'}(z) + f(a)d_T(a) + f(b)d_T(b)$$

Nota: Il termine tra parentesi quadre è $b(T')$. Sottraiamo il contributo di z (che è in T' ma non è foglia in T) e aggiungiamo i contributi di a e b, così abbiamo ottenuto $b(t)$, e l'uguaglianza è vera.

4. Sostituendo le relazioni di profondità e frequenza ($f(z) = f(a) + f(b)$):

$$b(T) = b(T') - (f(a) + f(b))d_{T'}(z) + f(a)(d_{T'}(z) + 1) + f(b)(d_{T'}(z) + 1)$$

5. Svolgendo i calcoli, i termini con $d_{T'}(z)$ si cancellano: $= f(a) + f(b)$

6. Quindi la relazione fondamentale è: $b(T) = b(T') + f(a) + f(b)$

Suppongo che la sottostruttura ottima non esista (dimostrazione per assurdo)

- **Ipotesi per assurdo:** Supponiamo che T **non** sia l'albero ottimo per Σ (e quindi anche che T' non sia l'albero ottimo per Σ'). Di conseguenza, deve esistere un albero T_{opt} con costo strettamente inferiore a T : $b(T_{opt}) < b(T)$
- **Costruzione dell'albero ridotto:** Prendiamo T_{opt} e uniamo le foglie a e b nel padre z . Otteniamo un nuovo albero T'_{opt} valido per l'alfabeto ridotto Σ' . Il costo di questo albero ridotto è:

$$b(T'_{opt}) = b(T_{opt}) - (f(a) + f(b))$$

- **Sviluppo algebrico:** Riprendiamo la diseguaglianza del punto 1: $b(T_{opt}) < b(T)$
Sottraiamo a entrambi i membri la quantità costante $(f(a) + f(b))$:
 $b(T_{opt}) - (f(a) + f(b)) < b(T) - (f(a) + f(b))$ Sostituiamo i termini con le definizioni dei costi ridotti ($b(T'_{opt})$ e $b(T')$):

$$\underbrace{b(T_{opt}) - (f(a) + f(b))}_{b(T'_{opt})} < \underbrace{b(T) - (f(a) + f(b))}_{b(T')}$$

Otteniamo infine:

$$b(T'_{opt}) < b(T')$$

Abbiamo dimostrato l'esistenza di un albero T'_{opt} con costo inferiore a T' .
Questo **contraddice l'ipotesi iniziale** che T' fosse l'albero ottimo per Σ' .

Adesso vediamo lo pseudocodice di questa procedura:

L'algoritmo di Huffman ha complessità $O(n \log n)$ dove n è la cardinalità di Σ

1. **Huffman(Σ, f)**
2. **Q = newPriorityQueue**
3. **foreach c ∈ Σ do**
4. `x = newNode(c)`
5. `insert x in Q`
6. **for i = 1 to |Σ| - 1 do**
7. `x = extractMin(Q)`
8. `y = extractMin(Q)`

9. `z = newNode`

10. **f(z) = f(x) +f(y)**

11. **left(z) = x**

12. **right(z) = y**

13. **insert z in Q**

riga 1 → definizione della funzione, Σ è l'alfabeto, f invece le frequenze di ogni carattere

riga 2 → crea una coda con priorità, che servirà a mantenere in cima i valori di frequenze più basse

riga 3-5 → inizia un ciclo che scorre ogni carattere, crea un nodo x dove viene inserito il carattere e poi lo inserisce nella coda con priorità

riga 6 → questo è il ciclo principale, che verrà eseguito $n - 1$ volte dove n è il numero di caratteri

riga 7-8 → estrae dalla coda Q i 2 nodi che hanno frequenza più bassa (sono i nodi da combinare)

riga 9 → crea il nuovo nodo z

riga 10 → calcola la frequenza combinata degli altri 2 nodi

riga 11-12 → assegna a x e y come figli destro e sinistro a z

riga 13 → reinserisce z nella coda, e ricomincia, da questo momento z è un nodo combinato e verrà trattato come un altro carattere con una propria frequenza

Alla fine del ciclo, nella coda Q rimarrà un solo nodo: la radice dell'intero albero di Huffman. Percorrendo l'albero dalla radice alle foglie si ottengono i codici binari per ogni carattere.

GRAFI E CAMMINI MINIMI

Prima di introdurre il prossimo problema risolto con approccio greedy, ripassiamo i grafi:

$$G = (V, E)$$

$V = \{v_1, v_2, v_3, \dots, v_n\}$ → questi sono i nodi o vertici del grafo $\rightarrow |V| = n$

$E \subseteq \{(a_i, a_j) : i, j \in V\}$ → questi sono gli archi che collegano 2 nodi

Tipo	Relazione degli Archi	Rappresentazione Visiva
Direzionario	Coppia ordinata (u, v)	Frecce \rightarrow
Non Direzionato	Insieme non ordinato $\{u, v\}$	Linee –
Ordinato	Segue una sequenza logica	Nodi in fila (es. A, B, C)

Percorsi

Dato un percorso $P = < u_1, u_2, \dots, u_k >$ è un cammino se:

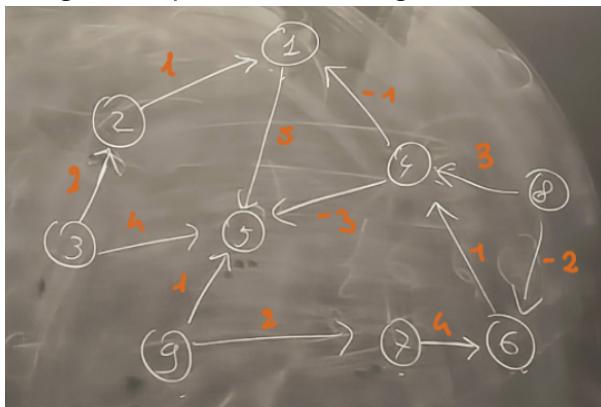
1. $u_i \in V, \forall 1 \leq i \leq k \rightarrow$ ogni elemento del cammino deve essere un vertice
 2. $(u_i, u_{i+1}) \in E \quad \forall 1 \leq i < k \rightarrow$ per ogni coppia di nodi deve esistere un arco che li collega
- Un cammino semplice è un cammino che passa 1 sola volta da ogni nodo (aciclico)

Relazioni e grafi

Se dobbiamo rappresentare una relazione con un grafo possiamo identificarla con un arco tra 2 nodi, come scegliamo se usare un arco orientato o non orientato? Dipende dalla relazione che devo seguire

Grafi pesati

Un grafo è pesato se ad ogni arco diamo un peso, utilizzando la funzione: $W : E \rightarrow R$



Trovare usi in cui gli archi abbiano un peso negativo è difficile ma esistono, la maggior parte però hanno peso positivo

Calcolare il peso di un percorso

Usiamo questa formula:

$$W(p) = \sum_{i=1}^{k-1} W(u_i, u_{i+1})$$

La sommatoria ci dice di sommare il peso di tutti gli archi del cammino partendo dal primo arco fino all'ultimo, mentre $W(u_i, u_{i+1})$ rappresenta il peso del singolo arco, quindi si deve sommare il peso di ogni arco

DOMANDA ESAME

Che cosa è un ordinamento topologico di un grafo?

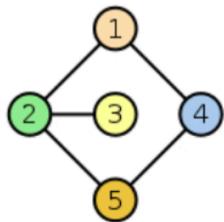
- Un ordinamento topologico di un grafo è un ordinamento lineare dei nodi in modo che ci sia una determinata relazione, se $\exists (u, v) \in E$ tale che $u < v$. Ad esempio: A-F-D-G-C-E-B, questo potrebbe essere un esempio di ordinamento topologico di un grafo
- In un grafo possono esserci più ordinamenti topologici.
- Se il grafo ha un ciclo non posso fare ordinamenti topologici

Definizione Componenti Connesse

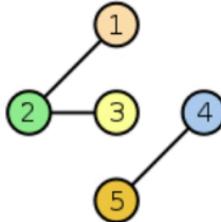
Sia $G = (V, E)$ un grafo e sia $V = v_1 \cup v_2 \cup \dots \cup v_k$ la partizione indotta dalla relazione di connessione tra i nodi. Sia $G = \{V_i, E_i\}$ il sottografo indotto da V_i per ogni $i = 1 \dots k$. Tali

sottografi si chiamano componenti connesse di G .

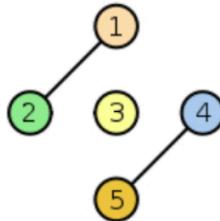
Esempi:



Grafo con 1 comp. connessa



Grafo con 2 comp. connesse



Grafo con 3 comp. connesse

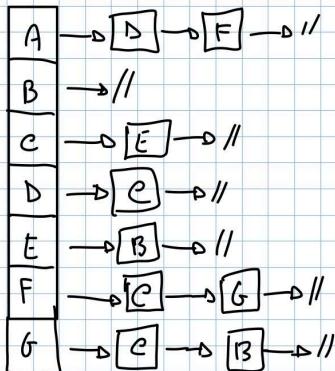
Definizione Componenti Fortemente Connesse

In un grafo $G = (V, E)$ se ho una componente连通的 che sia un ciclo allora ho una componente fortemente connessa

Come rappresentiamo i grafi?

Abbiamo 2 modi: *liste di adiacenza* o *matrici di adiacenza*

LISTE DI ADIACENZA



$O(\sqrt{v + e})$

MATRICE DI ADIACENZA

	A	B	C	D	E	F	G
A	-	1	1	1	1	1	-
B	-	-	-	-	-	-	-
C	-	-	-	-	1	-	-
D	-	-	1	-	-	-	-
E	-	1	-	-	-	-	-
F	1	-	1	-	-	-	1
G	1	1	2	-	-	-	1

$O(\sqrt{v^2 + e})$

BFS → Breadth-First Search → Ricerca in ampiezza

L'idea alla base della BFS è partire da un nodo S e esplorare i nodi a distanza 1 poi 2 poi 3 e così via...

Per sapere se siamo già passati da un nodo usiamo i colori, ovvero delle variabili che cambiano valore (come per gli alberi rosso neri)

Usiamo:

- grigio quando visito il nodo
- nero dopo la visita
- bianco non visitato

Usiamo $d[v]$ che indica la distanza di un nodo v da s

Vediamo lo pseudocodice della procedura BFS che ha complessità $O(V + E)$ se uso la lista di adiacenza oppure $O(V^2)$ se uso la matrice di adiacenza

1. **BFS(V, s)**
2. `for each $v \in V$ do`
3. `color[v] = W` \rightarrow $W = \text{white}$
4. `d[v] = +\infty`
5. `d[s] = 0` \rightarrow la distanza tra s e se stesso è 0
6. `Q = {}` \rightarrow coda vuota
7. `enqueue(s)`
8. `color[s] = G` \rightarrow $G = \text{grey}$
9. `while Q ≠ {} do`
10. `v = dequeue(Q)`
11. `for each $u \in \text{Adj}(v)$ do`
12. `if color[u] = W then`
13. `d[u] = d[v]+1`
14. `color[u] = G`
15. `enqueue(u)`
16. `color[v] = B` \rightarrow $B = \text{black}$

riga 1 → dichiarazione della funzione, prende in input l'insieme dei vertici e il vertice di inizio(s)

riga 2-4 → scorre ogni nodo e setta il colore a bianco per inizializzarli, e la distanza dei vertici ad infinito

riga 5 → imposta la distanza della sorgente (s) da se stessa che è 0

riga 6 → inizializza una coda vuota

riga 7 → inserisce il nodo di partenza nella coda, è il primo nodo da analizzare

riga 8 → colora il vertice (s) di grigio perché sta venendo analizzato

riga 9 → inizia il while principale della funzione, questo while continua finché la coda non è vuota, se la coda è vuota vuol dire che abbiamo analizzato tutti i nodi

riga 10 → estrae il primo nodo dalla coda, quindi lo analizziamo

riga 11 → questo ciclo for scorre tutti i vicini di v (che chiama u)

riga 12 → controlla se il vicino (u) è bianco (non visitato)

riga 13 → qui costruiamo i livelli di distanza, perché la distanza di u da s(sorgente) è la distanza di v +1

riga 14 → adesso u è stato scoperto quindi lo coloriamo di grigio

riga 15 → aggiungiamo u alla coda così che possa essere successivamente analizzato

riga 16 → imposta il colore di v a nero perché abbiamo finito di analizzarlo

Una visita BFS genera un albero BFS

DFS → Depth-First Search → Ricerca in Profondità

Funzione naturalmente ricorsiva, viene usato per trovare le componenti fortemente connesse.

Anche qui vale la regola:

- █ grigio quando visito il nodo
- █ nero dopo la visita
- □ bianco non visitato

Iniziamo dallo pseudo codice della procedura

1. **DFS(G, S)**
2. `for each v ∈ V do`
3. `color[v] = W`
4. `π[v] = NULL` → questo è il padre del nodo corrente
5. `T = 0` → tempo

```

6.   `for each v ∈ V do`  

7.     `if color[v] = W then`  

8.       `DFS-VISIT(v)`  


```

riga 1 → definizione della funzione, prende in input il grafo G e il nodo sorgente S

riga 2-4 → per ogni nodo v li colora di bianco e setta il loro padre a NULL

riga 5 → inizializza il tempo di visita a 0 perché non abbiamo ancora iniziato

riga 6-8 → per ogni nodo v controlla se il colore è bianco (non visitato) e se si chiama DFS-VISIT

Complessità $O(V)$

```

1. DFS-VISIT(v)  

2.   `d[v] = T`  

3.   `T = T+1`  

4.   `color[v] = G`  

5.   `for each u ∈ Adj(v) do`  

6.     `if color[u] = W then`  

7.       `π[u] = v`  

8.       `color[u] = B`  

9.       `F[u] = T`  

10.      `T = T+1`  


```

riga 1 → definizione della funzione, prende in input il nodo v

riga 2 → inizializza **d[v]** (array che tiene traccia del tempo di inizio visita) con il tempo T

riga 3 → aumenta il tempo di 1 perché stiamo andando avanti

riga 4 → colora di grigio il nodo perché lo stiamo esaminando

riga 5-7 → per ogni nodo u che è vicino di v controlla se u è bianco (non visitato) e in caso imposta come padre di u il nodo v

riga 8-9 → a questo punto con v abbiamo finito e lo coloriamo di nero e impostiamo il valore di T come tempo di fine visita in $F[v]$

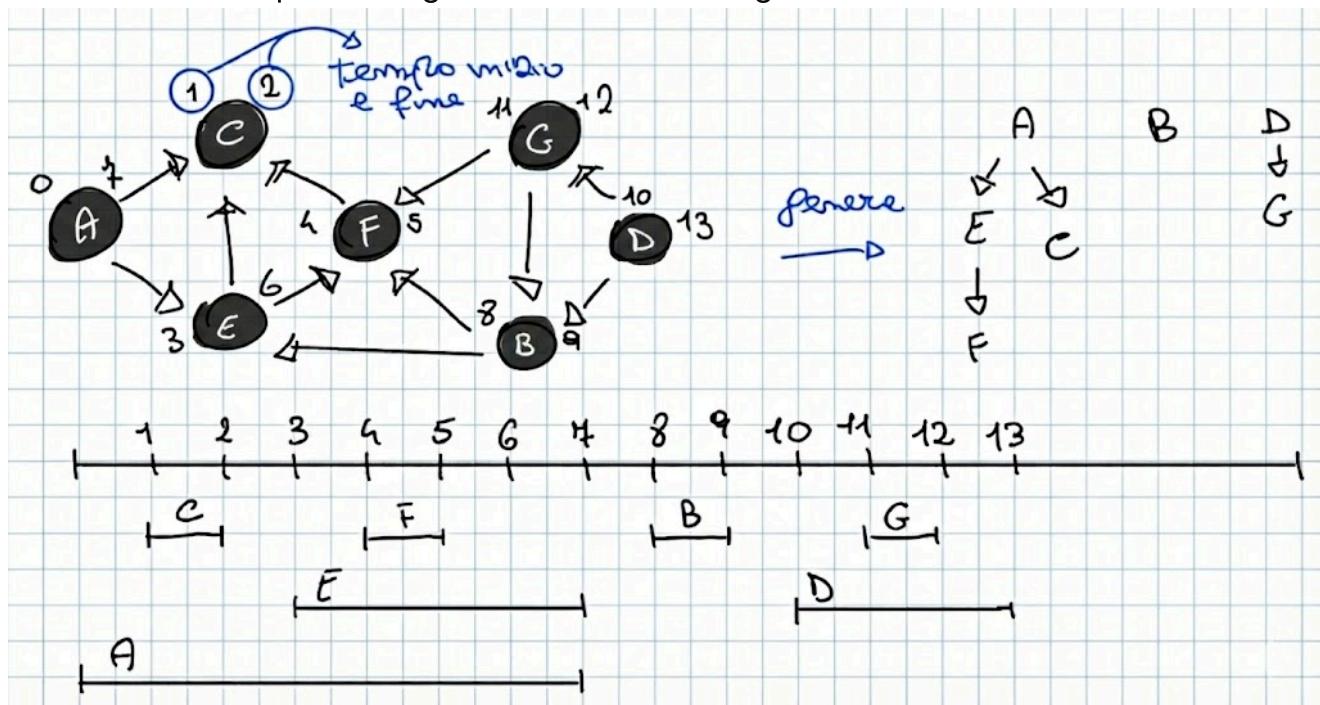
riga 10 → aumentiamo il tempo T di 1 perché stiamo andando avanti

Complessità $O(E)$

Complessità totale delle 2 procedure = $O(V + E)$

La chiamata DFS genera un albero DFS

Vediamo un esempio su un grafo e l'albero che si genera



Come possiamo vedere si inizia dal nodo A che ha anche il tempo di inizio e fine, come scritto in ogni nodo e nella timeline, accanto possiamo vedere l'albero DFS che si genera in base a come sono esplorati i nodi

Ordinamenti topologici

Etichettiamo gli archi inserendo:

T : archi consecutivi nell'albero DFS

I : archi all'indietro

F : archi in avanti → cioè archi che collegano A ad F ad esempio (cioè collegano ad un discendente non diretto)

C : archi di attraversamento

Tutte queste informazione ci servono a fare un ordinamento topologico del grafo usando la DFS

Ad esempio sull'esercizio di prima:

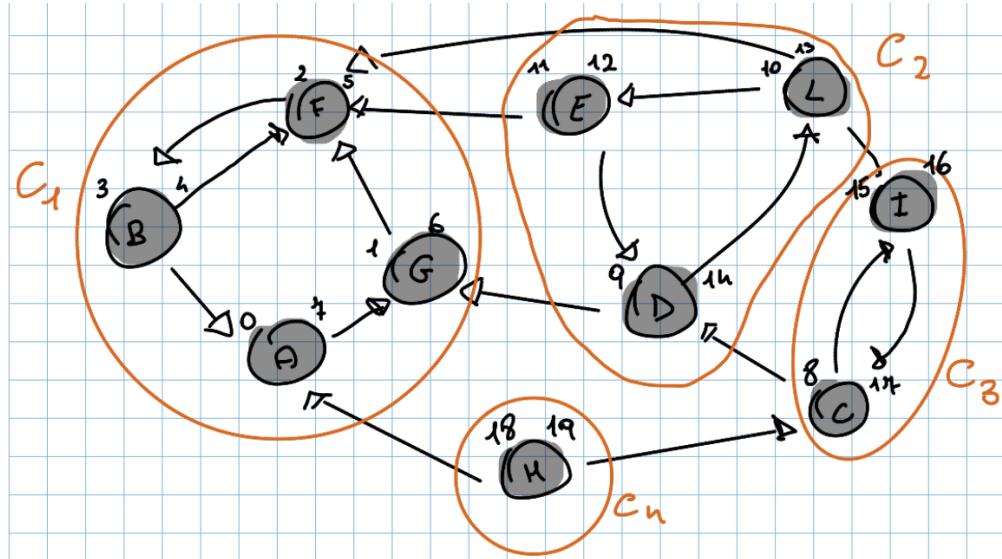
Ordinamento topologico manuale → A,D,G,B,E,F,C

Ordinamento topologico rispetto al tempo di fine visita → D,G,B,A,E,F,C

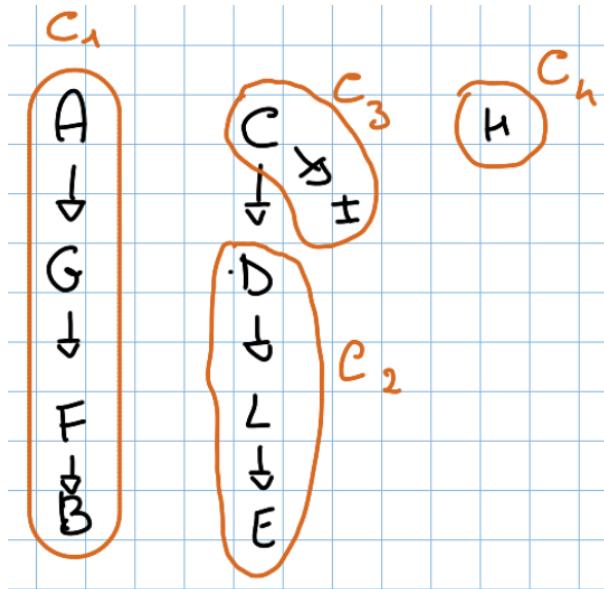
Quindi possiamo usare la DFS sia per l'ordinamento topologico sia per verificare l'aciclicità (nella parte in cui scorre tutti i vicini di v se trova nodi grigi vuol dire che c'è un ciclo)

Cicli DFS

Avendo delle componenti fortemente connesse, possiamo suddividere il grafo in **macronodi** ciascun macronodo è una componente fortemente connessa



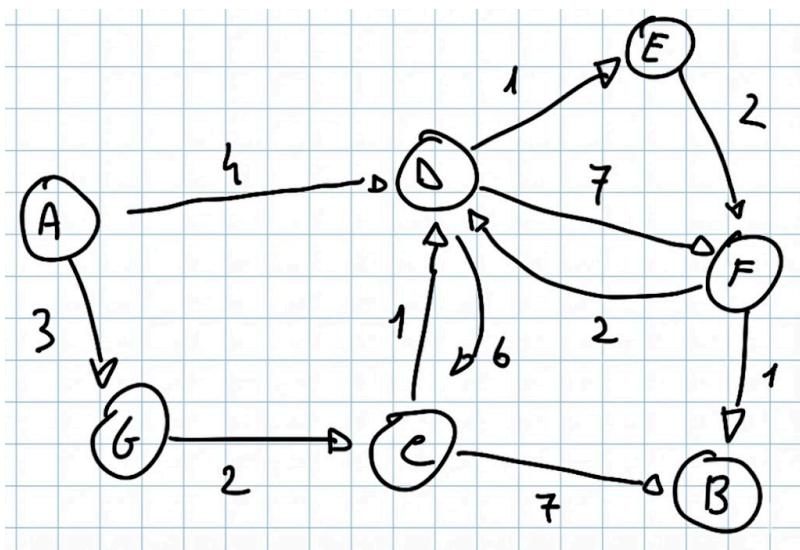
Chiamando la DFS su questo grafo avremo questo albero



Quindi abbiamo un numero di alberi in base a quante sono le componenti fortemente connesse.

Problema dei cammini minimi

Negli esempi useremo grafi orientati ma vale la stessa cosa per quelli non orientati
Abbiamo un grafo pesato e diversi percorsi tra cui scegliere, vogliamo scegliere il percorso meno dispendioso



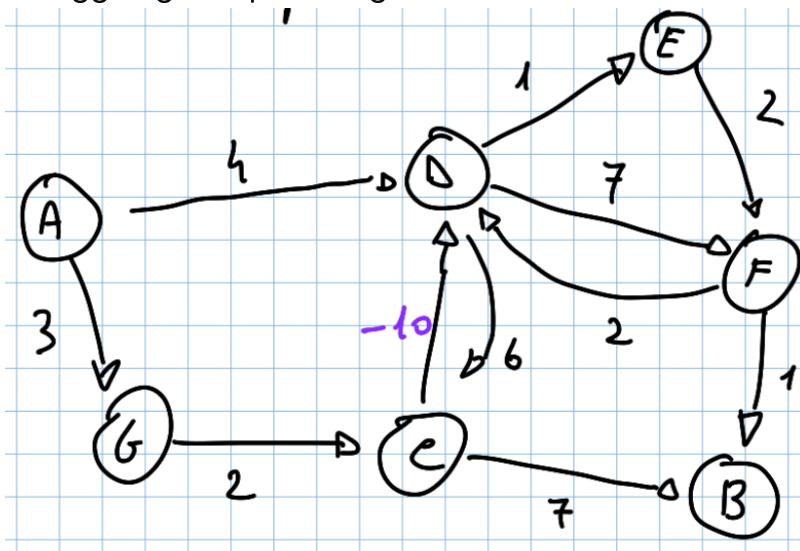
Possiamo percorrere diversi cammini:

$\langle A, G, C, B \rangle \rightarrow \text{costo} = 12$

$\langle A, D, E, F, B \rangle \rightarrow \text{costo} = 17$

$\langle A, D, E, F, B \rangle \rightarrow \text{costo} = 8 \rightarrow$ anche se è il più lungo è quello che costa meno

Se aggiungo un peso negativo la situazione cambia:



$\langle A, G, C, D, E, F, B \rangle \rightarrow \text{costo} = -1$

Se esiste un peso negativo dove c'è un ciclo il problema non ha soluzione, perché non esiste un cammino minimo

La nostra soluzione può essere solo positiva

Cammini minimi

Abbiamo 4 tipi di cammini minimi

1. **Problema dei cammini minimi da sorgente unica(single source)**: dato un grafo vogliamo trovare un cammino minimo che va da un dato vertice sorgente $s \in V$ a ciascun vertice $v \in V$

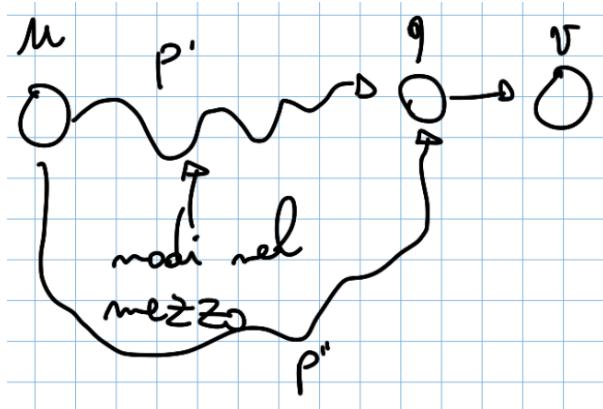
2. *Problema dei cammini minimi con destinazione unica(single destination)*: trovare un cammino minimo da ciascun vertice v a un dato vertice t destinazione.
(Invertendo la direzione di ciascun arco nel grafo lo possiamo ricondurre al primo caso)
3. *Problema dei cammini minimi fra tutte le coppie di vertici(all-pairs)*: trovare un cammino minimo da u a v per ogni coppia di vertici.
4. *Problema del cammino minimo per una coppia di vertici(single pair)*: trovare un cammino minimo da u a v , è una variante del primo problema.

Affronteremo solo il primo e il terzo, iniziamo con il primo problema, **Single-Source-Shortest-Path**

Dimostrazione cammino minimo e sottostruttura ottima

Dimostrazione per assurdo:

Immaginiamo un cammino minimo P' dal nodo u al nodo q e al nodo v che è la destinazione (q è il nodo prima della destinazione)



avremo che: $P'(u \rightsquigarrow q) + w(q, v) = (u, v)$

supponiamo di avere anche un secondo cammino ancora più piccolo:

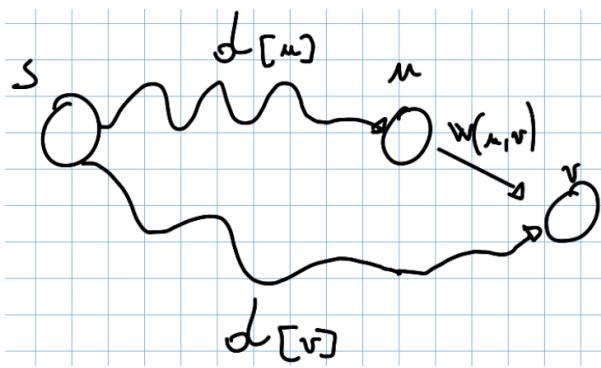
$P''(u \rightsquigarrow q) < P'(u \rightsquigarrow q)$ ma:

$P''(u \rightsquigarrow q) + w(q, v) < P'(u \rightsquigarrow q) + w(q, v) = \delta(u, v)$ che è una contraddizione, quindi si evince che P'' non può essere un cammino minimo e che un qualsiasi sotto cammino del cammino minimo è anch'esso minimo

Nel codice per tenere traccia della *stima del cammino minimo* dalla sorgente s a v useremo un array, $d[v]$ che sarà inizialmente inizializzato a $+\infty$ perché non ho modo di sapere quale sia il camminino minimo prima di iniziare a cercarlo, e successivamente dopo vari *aggiornamenti* con dei cammini minimi temporanei all'interno di $d[v]$ ci sarà il vero cammino minimo

Questi *aggiornamenti* sono dei **relax(u, v)** degli archi

Cosa è il relax di un arco?



Ho $d[u]$ e $d[v]$ e mi faccio questa domanda:

if ($d[u] + w(u,v) < d[v]$) **then** $d[v] = d[u] + w(u,v)$ ovvero, il percorso da S a u + il pezzo per arrivare da u a v è più piccolo del percorso da S a v che ho trovato precedentemente? se si allora assegno il nuovo cammino minimo all' array $d[v]$
Questo è il relax di un arco

Vediamo lo pseudocodice della funzione *Generic-single source shortest path che trova un cammino minimo*, ma a differenze dell'algoritmo di Bellman-Ford se nel grafo c'è un ciclo negativo questo algoritmo non funziona più

1. **Generic-SSSP(G,S)**

2. `for each $v \in V$ do`

3. ` $d[v] = +\infty$ `

4. ` $d[S] = 0$ `

5. `while $\exists (u,v) \in E : d[u] + w(u,v) < d[v]$ do`

6. `RELAX(u,v)`

7. `return d`

riga 1 → definizione della funzione che prende in input il grafo e il nodo S che è la sorgente

riga 2-4 → per ogni nodo v imposta la stima della distanza da S a + infinito e la distanza da S a 0

riga 5-6 → questo è pezzo principale della procedura, continuerà a chiamare la funzione **RELAX(u,v)** finché esisterà un arco da u a v tale che $d[u] + w(u,v)$ sia minore di $d[v]$

riga 7 → ritorna la distanza cioè il cammino minimo

Questo algoritmo ha complessità $O(2^v)$ nel caso peggiore

Proprietà dei cammini minimi

1. Disuguaglianza triangolare

Per qualsiasi arco $(u, v) \in E$, si ha $\delta(s, v) \leq \delta(s, u) + w(u, v)$

2. Limite superiore

Per tutti i vertici $v \in V$, si ha sempre $d[v] \geq \delta(s, v)$ e, una volta che il limite superiore $d[v]$ assume il valore $\delta(s, v)$, esso non cambia più

3. Convergenza

Se ho un cammino del tipo: $s \rightsquigarrow u \rightarrow v$ da un punto $d[u] = \delta(s, u)$ sono sicuro che rilassando $d[v] = \delta(s, v)$

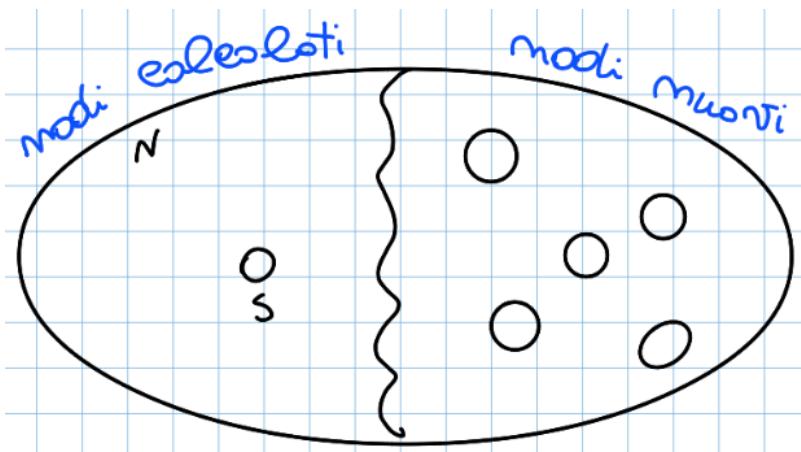
DAG → Directed Acyclic Graph

Rilassando gli archi di un DAG (Directed Acyclic Graph) pesato $G = (V, E)$ secondo un ordine topologico dei suoi vertici è possibile calcolare i cammini minimi da una sorgente unica nel tempo $\Theta(V + E)$. L'algoritmo inizia ordinando topologicamente il DAG, se esiste un cammino dal vertice u al vertice v , allora u precede v nell'ordine topologico. Effettuiamo un passaggio sui vertici secondo l'ordine topologico. Durante l'elaborazione vengono rilassati tutti gli archi che escono dal vertice

1. DAG-SHORTEST-PATHS(G, s)

2. `foreach $v \in V$ do`
3. ` $d[v] = +\infty$ `
4. `esegui DFS per calcolare il tempo di fine visita $F[v]$ `
5. `for each $v \in V$ do` \rightarrow in ordine topologico
6. `for each u in $\text{Adj}[v]$ do`
7. `RELAX(u, v)`

Risolviamo il problema nel caso in cui ci siano cicli negativi usando Bellman-Ford



Questa è la configurazione iniziale prima di iniziare l'algoritmo, man mano che eseguo le relax i nodi si spostano dall'insieme dei nodi nuovi a quello dei calcolati, solo in questa direzione, normalmente l'insieme dei nodi calcolati all'inizio contiene solo la sorgente S, ma nel caso in cui nel grafo ci siano nodi isolati questi si troveranno già qui, solo che noi lo scopriremo alla fine dell'algoritmo (si trovano lì perché da soli costituiscono un cammino minimo)

Questo algoritmo rilassa ogni arco del grafo $V-1$ volte

$V - 1$ perché è la lunghezza massima di un cammino minimo in un grafo

Vediamone lo Pseudocodice, questo algoritmo ha complessità $O(V * E)$ se si usano liste di adiacenza e $O(V^3)$ se si usano matrici di adiacenza

1. **Bellman-Ford(G, s, w)**
2. `d = newArray(len(v))`
3. `for each v ∈ V do`
4. `d[v] = +∞`
5. `π[v] = NULL` → padre nodo corrente
6. `d[s] = 0`
7. `for i = 0 to V-1 do` → $V-1$ → è questa la lunghezza massima di un cammino minimo
8. `for each (u,v) ∈ E do`

```

9.     `relax(u,v,w)`

10.    for each (u,v) in E do → ultimo relax che (V in totale) così da trovare cicli negativi

11.        `if(d[v] > d[u]+w(u,v)) then`


12.            `return FALSE`


13.    return TRUE

```

riga 1 → dichiarazione della funzione, prende in input il grafo G, la sorgente s e un array con i pesi w

riga 2 → creo un array che ha lunghezza pari al numero di vertici

riga 3-5 → per ogni nodo v inizializza la stima della distanza a +infinito e il padre di ogni nodo a NULL

riga 6 → imposta la distanza da s a 0

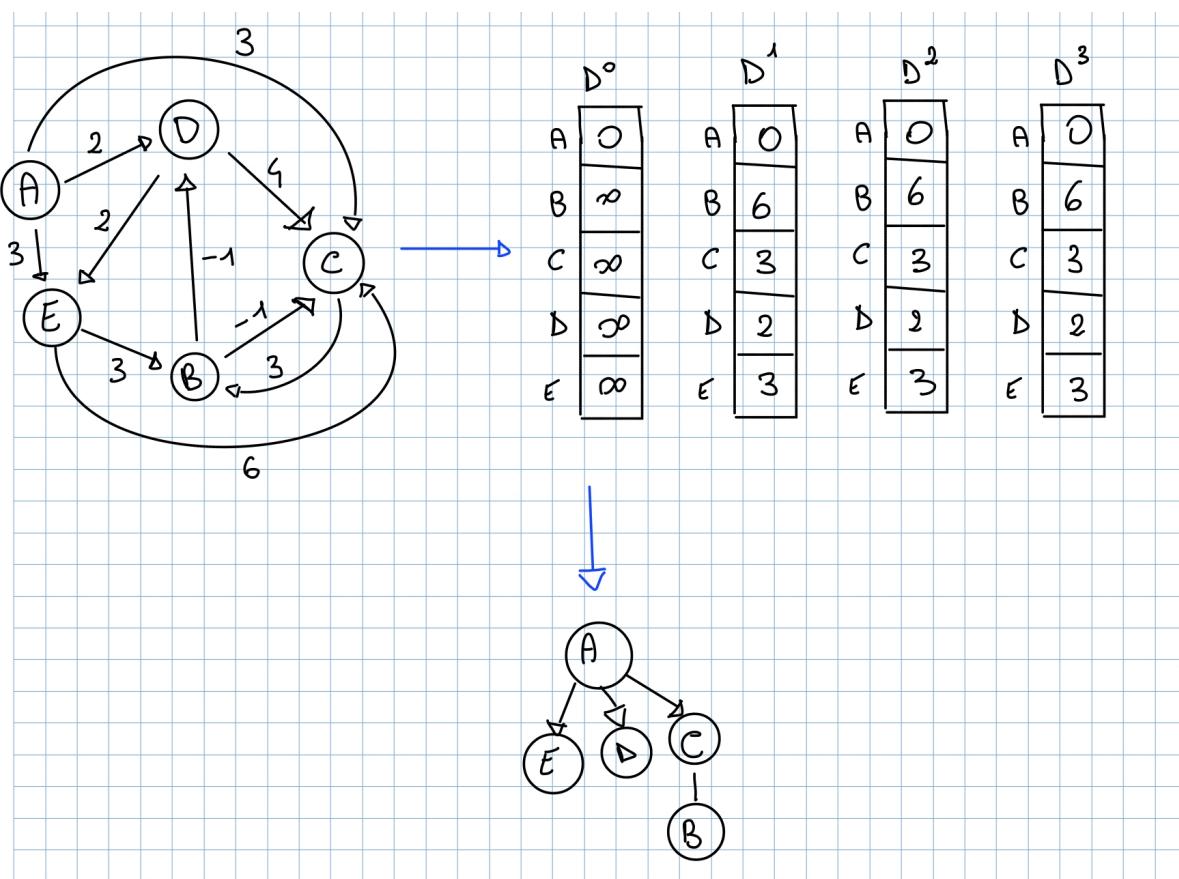
riga 7-9 → questa è la parte principale, ogni arco viene rilassato V-1 volte (V-1 perché è la lunghezza massima di un cammino minimo) (*chiede all'esame*)

riga 10-12 → rilassa un ultima volta, che sarebbe una volta in più del necessario perché se effettivamente funziona e gli archi si rilassano allora c'è un ciclo negativo e la funzione ritorna FALSE non essendoci un cammino minimo

riga 13 → se non si sono rilassati allora il grafo non ha cicli negativi quindi ritorna TRUE e abbiamo così il nostro cammino minimo

All'esame potrebbe esserci un applicazione di questo tipo in cui passo passo eseguiamo l'algoritmo

Si inizia dal nodo A e dobbiamo trovare il cammino minimo per arrivare a ogni nodo:



Caratteristica	Algoritmo Generico (Generic-SSSP)	Bellman-Ford
Logica	"Rilassa finché puoi": sceglie archi a caso finché trova qualcosa da migliorare.	"Sistematica": rilassa tutti gli archi del grafo per esattamente V - 1 volte .
Cicli Negativi	Loop Infinito : continua a rilassare gli archi del ciclo senza mai fermarsi.	Rilevamento : esegue un controllo finale (la V-esima volta) per segnalare se esistono cicli negativi.
Terminazione	Non garantita se il grafo contiene cicli di peso negativo raggiungibili.	Garantita sempre, perché il numero di iterazioni è fissato.
Complessità	Potenzialmente esponenziale $O(2^V)$ nel caso peggiore.	Determinata: $O(V \cdot E)$ con liste di adiacenza o $O(V^3)$ con matrice.

Algoritmo di Dijkstra

Questo algoritmo lavora solo con pesi positivi, se incontra un peso o un loop negativo fallisce

Vediamo lo pseudocodice, ha complessità $O((V + E)\log V)$

1. **Dijkstra(G, s, w)**
2. `d = newArray(len(V))`
3. `for each v ∈ V do`

4. `d[v] = +∞`
5. `π[v] = NULL` → padre del nodo corrente
6. `d[s] = 0`
7. `Q = buildmin-heap(v)`
8. `while Q ≠ ∅ do` → qua dentro voglio prendere il nodo v con stima più piccola in $v-s$
9. `v = extractMin(Q)`
10. `for each u in Adj[v] do`
11. `if d[u] > d[v] + w(v,u) then`
12. `decreaseKey(Q, u, d[v]+w(v,u))`

riga 1 → definizione della funzione, prende in input il grafo G , la sorgente s e l'array dei pesi w

riga 2 → crea un array d di lunghezza pari al numero di nodi, che conterrà la stima

riga 3-5 → per ogni nodo v imposta la distanza dalla sorgente a + infinito, e il padre di ogni nodo a NULL

riga 6 → imposta la distanza da s (sorgente) a 0

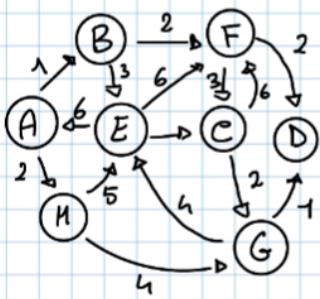
riga 7 → in Q crea un min-heap

riga 8 → ciclo principale che viene eseguito finché Q non è vuoto

riga 9 → passo greedy, estrae il nodo con la stima $d[v]$ più piccola fino a quel momento

riga 10-12 → per ogni nodo vicino di v (v è il nodo con la stima minima estratto prima) facciamo un relax, ovvero controlliamo se la distanza $d[u]$ è maggiore di $d[v] + w(v,u)$, se è così allora abbiamo trovare un nuovo cammino minimo, e aggiorniamo la coda Q (essendo un min-heap possiamo cambiare la priorità nella coda usando `decreaseKey`)

Applicazione di Dijkstra, potrebbe chiederla all'esame



	d^0_A	d^1_A	d^2_B	d^3_B	d^4_B	d^5_E	d^6_E	d^7_E
A	0	0	0	0	0	0	0	0
B	∞	1	1	1	1	1	1	1
C	∞	∞	∞	∞	6	5	5	5
D	∞	∞	∞	∞	5	5	5	5
E	∞	∞	4	4	4	4	4	4
F	∞	∞	3	3	3	3	3	3
G	∞	∞	6	6	6	6	6	6
H	∞	2	2	2	2	2	2	2

↙ Promi

Considerato pre
m Comincia

Vediamo adesso il 3 problema **All-Pairs-Shortest-Path (APSP)**

$\delta(u, v)$ = al cammino minimo tra u e v per qualsiasi u, v appartenenti a V

$\delta(v, v) = 0$

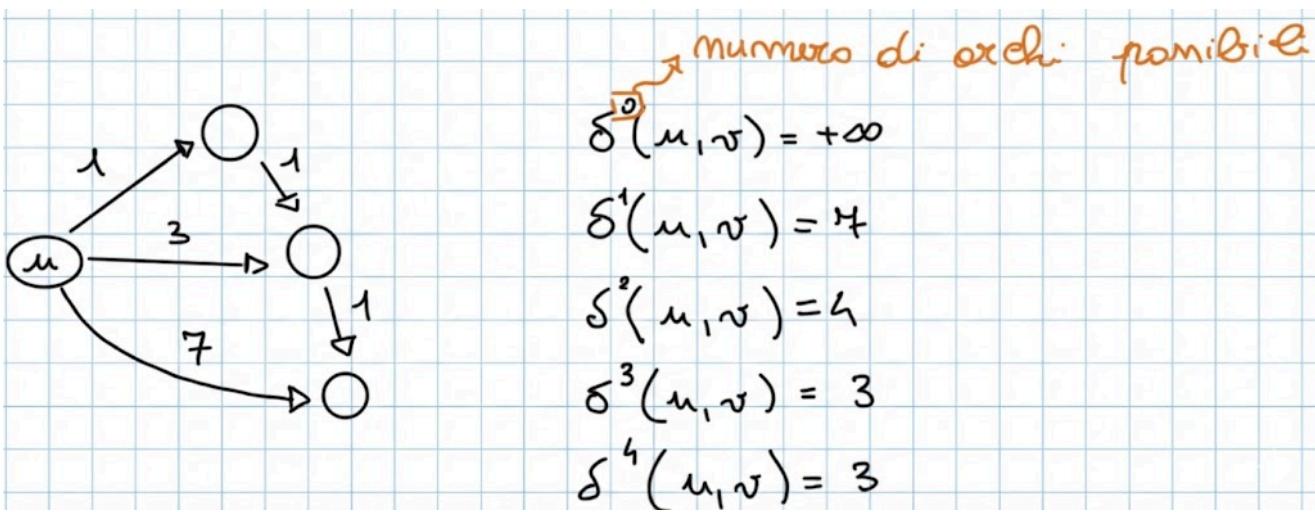
Noi potremmo usare gli algoritmi visti per il SSSP eseguirli V volte e ottenere i cammini minimi per tutte le coppie, ma è conveniente? Vediamo:

Algoritmo	SSSP	APSP
Bellman-Ford	$O(V^3)$	$O(V^4)$
Dijkstra	$O(V^2 \log V)$	$O(V^3 \log V)$
DAG	$O(V^2)$	$O(V^3)$

Come si vede aumentiamo di un ordine di grandezza la complessità, riusciamo a fare meglio di così? Creiamo un nuovo algoritmo

Useremo la programmazione dinamica (salteremo il punto 1 sottostruttura ottima perché è già stato dimostrato)

Identifichiamo la dimensione del problema attraverso: *Il numero di archi che possono essere presenti in un cammino minimo, cioè la lunghezza massima di un cammino minimo* $\rightarrow d$



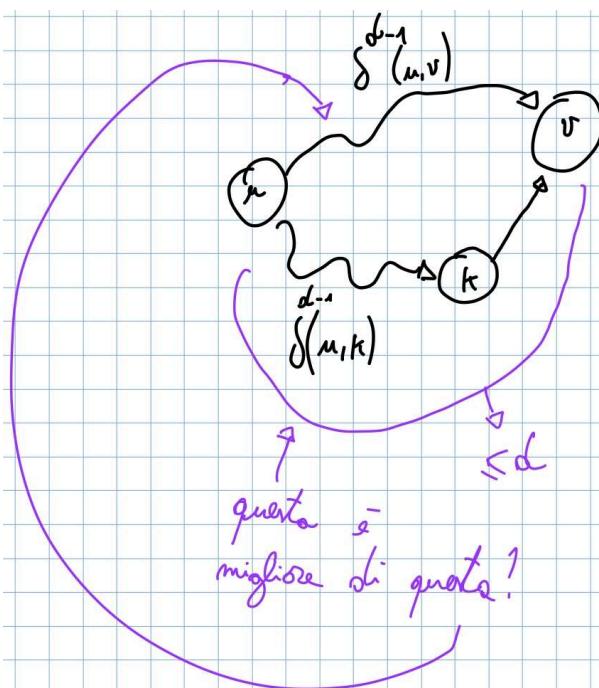
Se possiamo usare 0 archi la distanza tra u e v è infinita, se ne possiamo usare solo 1 allora è 7 e così via...

Quindi partiremo da $d = 0$ fino a scendere piano piano verso $d = V - 1$, usando una matrice di adiacenza W

prendiamo quindi questo come caso base:

$$W[i, j] = \begin{cases} 0 & \text{se } i = j \\ \text{peso}(i, j) & \text{se } (i, j) \in E \\ +\infty & \text{se } (i, j) \notin E, i \neq j \end{cases}$$

Adesso useremo un concetto simile alla relax di un arco:



ovvero devo quindi capire se il cammino $\delta^{d-1}(u, v)$ è migliore o peggiore del cammino $\delta^{d-1}(u, k) + w(k, v)$

quindi per qualsiasi k che trovo:

```
if(delta_d-1(u, k) + w(k, v) < delta_d-1(u, v)) then
    delta_d(u, v) = delta_d-1(u, k) + w(k, v)
else
    delta_d(u, v) = delta_d-1(u, v)
```

$$l^d(i, j) = \begin{cases} w[i, j] & \text{se } d = 1 \\ \min_{1 \leq k \leq n} (l^{d-1}(i, k) + w(k, j)) & \text{se } d > 1 \end{cases}$$

dove:

$l^d(i, j)$ è la matrice di adiacenza dei cammini minimi tra le coppie

$w(k, j)$ è il peso dell'arco da k a j

controlliamo il minimo un totale di n volte ($n = |V|$) usando k come contatore, inizialmente è 1 poi 2 fino ad n

Vediamo lo pseudocodice, con complessità $O(V^3)$ il primo e $O(V)$ il secondo che insieme fanno $O(V^4)$

Questa funzione calcola i percorsi con **un arco in più** rispetto a prima

```

1. extend-APSP(l^d, W)
2. `l^{d+1} = newMatrix(n, n)`
3. `for i = 1 to n do`
4.   `for j = 1 to n do`
5.     `l^{d+1}[i, j] = +∞`
6.   `for k = 1 to n do`
7.     `if l^d[i, k] + w(k, j) < l^{d+1}[i, k] then`
8.       `l^{d+1}[i, j] = l^d[i, k] + w(k, j)`
```

riga 1 → definizione della funzione, prende in input la matrice l^d che contiene i cammini minimi fino a quel punto trovati, e W che è la matrice di adiacenza con i pesi degli archi
riga 2 → crea una nuova matrice vuota dove scriveremo i risultati per $d+1$ (ovvero i cammini minimi nuovi scoperti usando un arco in più di prima)

riga 3-4 → fa 2 for per scorrere la matrice

riga 5 → inizializziamo tutta la matrice a +infinito

riga 6 → in questo ciclo proviamo tutti i possibili nodi intermedi k (sarebbe la parte del minimo nella formula)

riga 7-8 → prende il costo calcolato precedentemente da i a k , gli somma il peso da k a j e controlla se è minore del costo da i a k nella matrice con un arco in più, se si allora nella matrice nuova scriveremo il nuovo cammino minimo

questo codice va eseguito diverse volte partendo da $d = 2$ fino a $V-1$:

Questa ripete il compito finché non hanno coperto tutte le lunghezze possibili

1. APSP(W)

2. `l^1 = W`

3. `for d = 2 to V-1 do`

4. `l^d = extend-APSP(l^{d-1}, W)`

5. `return l^{V-1}`

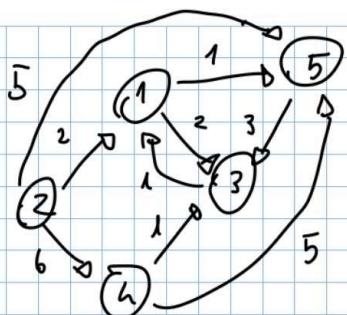
riga 1 → definizione della funzione, prende in input la matrice con i pesi degli archi

riga 2 → questo è il caso base, in cui si può usare un solo arco e quindi i cammini minimi sono semplicemente il peso di quegli archi

riga 3-4 → questo ciclo inizia da $d = 2$ perché $d = 1$ lo abbiamo già calcolato e si ferma a $V-1$ perché $V-1$ è la lunghezza massima di un cammino minimo, poi chiama per la matrice di indice d la funzione di prima passando la matrice precedente e la matrice dei pesi

riga 5 → ritorna l'ultima matrice che contiene i cammini minimi per ogni coppia di nodi

Applichiamo questo algoritmo:



venno fatte 4 matrici tot = $4 \times V-1$

$W = l^1$

1 2 3 4 5

1	0	∞	2	∞	5
2	2	0	∞	6	∞
3	1	0	0	9	∞
4	∞	∞	1	0	5
5	9	∞	3	∞	0

l^2 1 2 3 4 5

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

l^3 1 2 3 4 5

1	0				
2		0			6
3			0		
4				0	3
5					0

Questo algoritmo ha una complessità troppo alta, come lo miglioriamo?

Anziché creare sempre matrici nuove ne possiamo creare una e per le altre moltiplichiamo sempre la stessa, in maniera ottimizzata...

$l^1 = W$
 $l^2 = W * W$
 salto l^3
 $l^4 = l^2 * l^2$
 $l^8 = l^4 * l^4$
 e così via...

la nostra funzione può quindi diventare:

$$l^d(i, j) = \begin{cases} w[i, j] & \text{se } d = 1 \\ \min_{1 \leq k \leq n} (l^{d/2}(i, k) + l^{d/2}(k, j)) & \text{altrimenti} \end{cases}$$

1. **fast-APSP(W)**

2. `l^1 = W`

3. `d = 1`

4. `while d < V-1 do`

5. `l^2d = extend-APSP(l^d, l^d)`

6. `d = 2d`

7. `return l^d`

Inserendo un iterazione in più capiamo se ci sono cicli negativi

riga 1 → definizione della funzione, prende in input la matrice dei pesi W

riga 2 → caso base

riga 3 → impostiamo $d = 1$

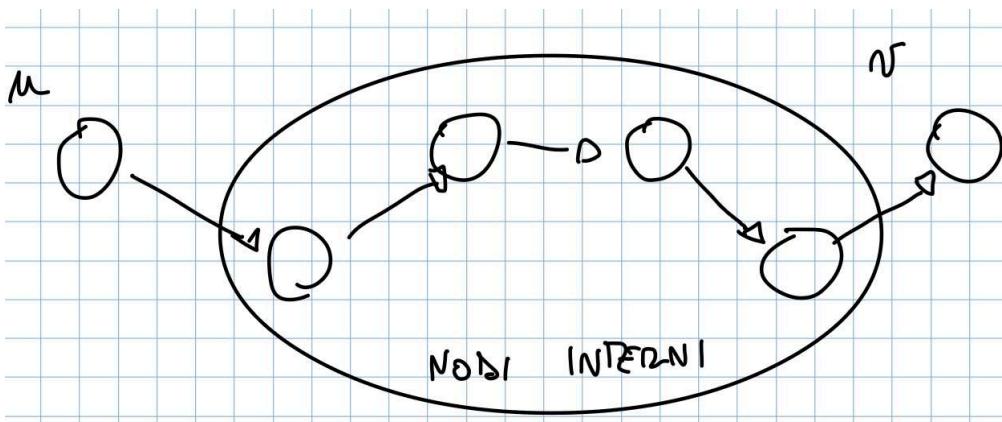
riga 4-6 → continuiamo a ciclare finché la lunghezza di d è minore di $V-1$ perché $V-1$ è la lunghezza massima di un cammino minimo, poi per la matrice $2d$ chiama la funzione passando l^d e l^d in questo modo non trova i percorsi di prima + un arco ma trova "percorso di lunghezza d + percorso di lunghezza d " in un colpo solo, quindi otteniamo un percorso di lunghezza $2d$, aggiorniamo $d = 2d$

riga 7 → ritorna l^d cioè l'ultima matrice calcolata

così facendo otteniamo una complessità di $O(v^3 \log V)$ come l'algoritmo di Dijkstra ma con la differenza che possiamo usare archi di peso negativo

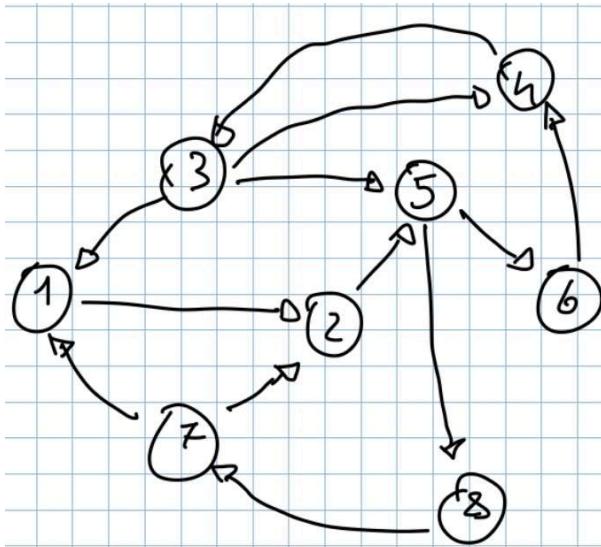
Ma resta comunque abbastanza alta la complessità, si può abbassare ancora usando l'algoritmo di Floyd-Warshall

Floyd-Warshall



Qui si usa un ragionamento diverso, ovvero anziché aumentare di 1 gli archi tutte le volte, aumentiamo di 1 i nodi visitabili, all'inizio avremo quindi

$V_0 = \{0\}$, $V_1 = \{v_1\}$, $V_2 = \{v_1, v_2\}$ ecc... fino a $V_k = \{v_1, v_2, \dots, v_k\}$, quindi **k sono i nodi interni da cui posso passare**, immaginiamo questo grafo:



$\delta^0(i, j) \rightarrow$ posso andare da 1 a 2 perché non passo da nodi interni, da 3 a 8 no
 $\delta^1(i, j) \rightarrow$ posso andare da 7 a 2 perché posso passare da 1 nodo interno

Usiamo una matrice $D^k[i, j] = \delta^k(i, j)$ per memorizzare un cammino minimo

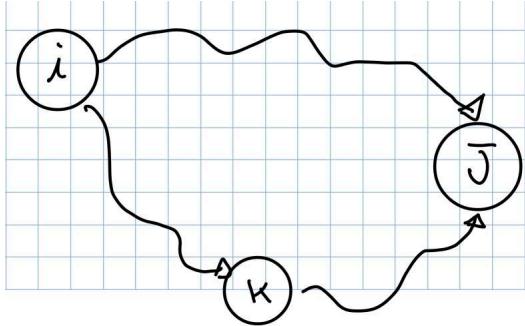
La nostra funzione ricorsiva è quindi:

$$D^k[i, j] = \begin{cases} w[i, j] & \text{se } k = 0 \quad \text{caso base con } V^0 = \{\} \\ \min_{\substack{\text{non ho bisogno di un for} \\ \text{che scorre tutti i nodi perché } k \text{ è definito}}} (D^{k-1}[i, j], D^{k-1}[i, k] + D^{k-1}[k, j]) & \text{se } k > 0 \end{cases}$$

dove:

$D^{k-1}[i, j]$ corrisponde alla soluzione del passo precedente

nel minimo in pratica controllo se:



il cammino da $i-k-k-j$ è più breve del cammino $i-j$

Prima di scrivere lo pseudocodice, per aiutarci a creare l'albero dei cammini minimi terremo conto del predecessore di un nodo, questa procedura ha complessità $O(V^3)$ esattamente come bellman-ford per risolvere l'SSSP

1. **Floyd-Warshall(W)**

2. ` $D^0 = W;$ `

3. ` $\pi[i,j] = \dots$ ` $i \rightarrow$ se (i,j) è un arco, altrimenti NULL

4. `for $k = 1$ to n do`

5. ` $D^k = \text{newMatrix}(n,n)$ `

6. `for $i = 1$ to n do`

7. `for $j = 1$ to n do`

8. ` $D^k[i,j] = D^{k-1}[i,j]$ `

9. ` $\pi^k[i,j] = \pi^{k-1}[i,j]$ `

10. `if $D^k[i,j] > D^{k-1}[i,k] + D^{k-1}[k,j]$ then`

11. ` $D^k[i,j] = D^{k-1}[i,k] + D^{k-1}[k,j]$ `

12. ` $\pi^k[i,j] = \pi^{k-1}[k,j]$ `

13. **return D^k**

riga 1 → definizione della funzione, prende in input la matrice dei pesi W

riga 2 → caso base, non posso esplorare nodi interni quindi resta il semplice peso dell'arco

riga 3 → Inizializza la matrice dei predecessori (π). Questa serve per ricostruire il percorso, per creare l'albero

riga 4 → il ciclo più importante, la k rappresenta la quantità di nodi interni che possiamo usare

riga 5 → prepara una nuova matrice, (nella realtà anziché creare k matrici ne usiamo solo 2 perché ci serve la corrente e la precedente, quando finisco con la corrente la faccio diventare la precedente e la precedente di prima la riuso come nuova matrice corrente)

riga 6-7 → for per scorrere la matrice

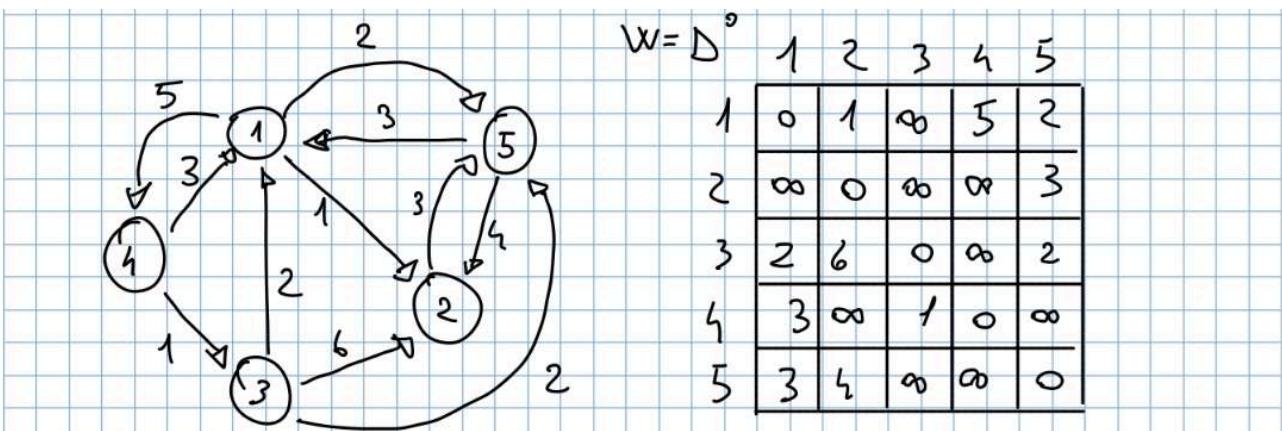
riga 8-9 → qui ipotizziamo che il percorso trovato precedentemente sia quello migliore

riga 10-12 → qui ci chiediamo se il percorso attuale i-j sia più lungo di quello i-k + k-j, se si allora il percorso i-k + k-j diventa il nuovo percorso minimo i-j, e aggiorniamo anche la matrice dei predecessori

riga 13 → ritorniamo la matrice dei cammini minimi

Questo algoritmo avendo una complessità di $O(V^3)$ è il migliore per risolvere il problema APSP

Esegiamolo


 D^1

	1	2	3	4	5
1	0	1	∞	5	2
2	∞	0	∞	0	3
3	2	3	0	7	2
4	3	4	1	0	5
5	3	4	∞	8	0

 D^2

	1	2	3	4	5
1	0	1	∞	5	2
2	∞	0	∞	∞	3
3	2	3	0	7	2
4	3	4	1	0	5
5	3	4	∞	8	0

la colonna e
la riga con lo
stesso elemento
della D^n si
rispondono

 D^3

	1	2	3	4	5
1	0	1	∞	5	2
2	∞	0	∞	∞	3
3	2	3	0	7	2
4	3	4	1	0	3
5	3	4	∞	8	0

 D^4

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

 D^5

	1	2	3	4	5
1	0	1	6	5	2
2	6	0	8	11	3
3	2	3	0	7	2
4	3	4	1	0	3
5	3	4	9	8	0

$$D^5[i,j] = \delta^5(i,j)$$