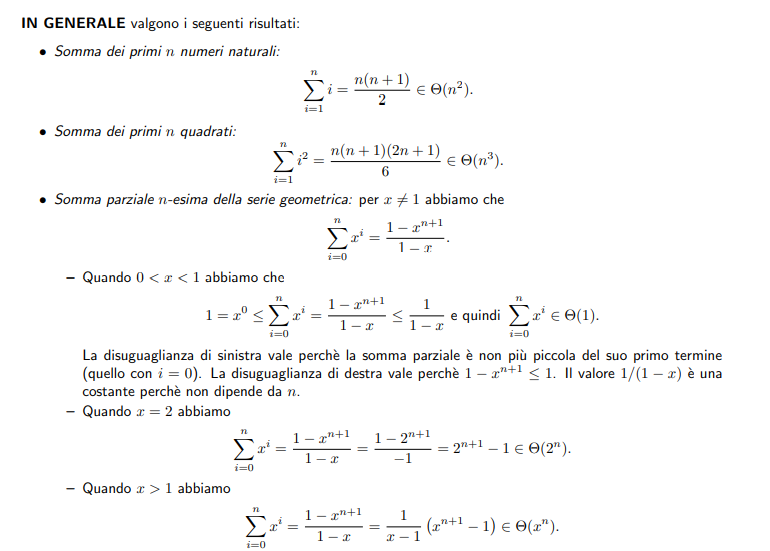
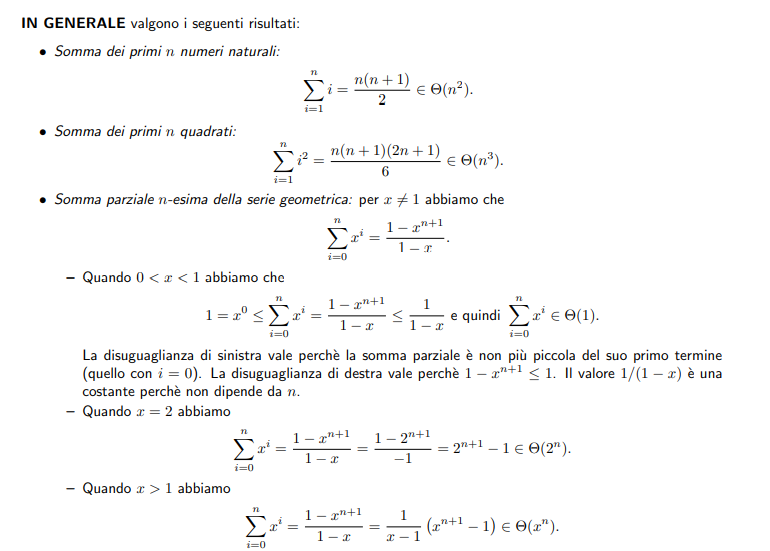
## Dire che valore stampa il codice (for annidati e while annidati)

* Se lo step è di 2, e l’argomento è 1, sopra alla sommatoria si scrive la metà del numero iniziale
* ⇒ nr di iterazioni
* for con while annidato: analizziamo cosa succede nel while al variare dell’indice del for (con i=0,i=1,...i=k, i=n) e determinare il numero di iterazioni

caso i=0 ⇒ nr di iterazioni → numero sopra alla sommatoria

caso i=n ⇒ nr di iterazioni → numero da cui parte i sotto la sommatoria





## Master Theorem (fornire un upper bound)

** **

* Determinare a b e d, sostituirli a logba e confrontare il risultato con d
* In base alla disuguaglianza riportare il risultato dal sistema a sinistra
* **HINT**:per ricordare il risultato pensare che il valore maggiore va messo ad esponente di n. Se sono uguali è una via di mezzo, quindi ndlog n

## Master Theorem (determinare α opportuno, se esiste)

* In base al costo computazionale T(n) fornito dall’esercizio, determinare il segno della disequazione.

## Ordinamento (Binary Search, dire indici confronto A[k] = key)

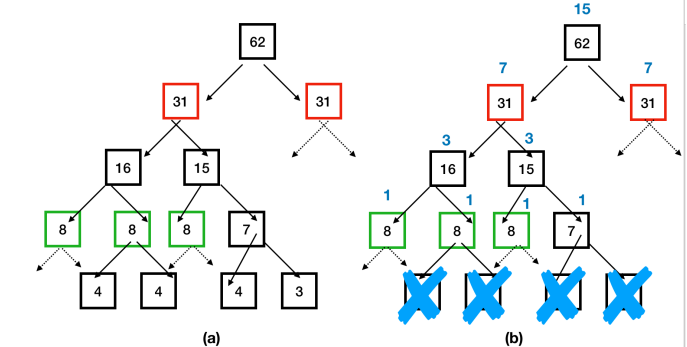
* Dati in ingresso array A e key (elemento da trovare)
* Eseguo operazioni della binary search, quindi trovo il k e faccio confronto.
* Se non ha esito positivo controllo se A[k] è minore o maggiore di key, se minore rieseguo operazioni dei punti prima sulla parte di sx (i=i j=k-1) altrimenti di dx
* continuo fino a quando non trovo effettivamente key oppure ho finito gli elementi nell’array su cui fare il confronto

## Ordinamento (MergeSort, dire sottosequenza su cui eseguo MergeSort alla i-esima chiamata ricorsiva compresa quella principale)

* In ingresso ho la sequenza A e il numero i (numero chiamata ricorsiva)
* Ricordando la struttura della funzione MergeSort alla chiamata principale prendo in considerazione l’intera sequenza
* successivamente faccio ricorsivamente MergeSort sulla parte di sx (i=i j=k) e di dx (i=k+1 j=j)
* continuo con la parte di sx fino a quando non ho terminato i valori e pian piano risalgo tra le varie chiamate ricorsive eseguendo la funzione sulle parti di destra (ovviemante se la parte di dx può essere divisa in ulteriori parti allora eseguirò la funzione su quella di sx e poi su quella di dx)
* termino quando raggiungo la chiamata ricorsiva richiesta (i) e dico su quale sequenza ha in input

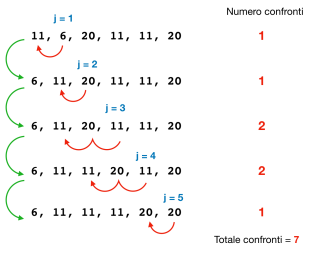
## Ordinamento (MergeSort che ordina sottosequenze di dim <= S usando InsertionSort, indicare numero di chiamate ricorsive)

* In input ho la lunghezza della sequenza e S
* divido dimensione sequenza per 2 fino a raggiungere sottosequenze di dimensione minore o uguale di S
* i blocchetti con numero <= S li elimino e risalendo l’albero conto il numero di blocchi non cancellati livello per livello + 1 per il padre, moltiplico il numero che ottengo (nell’esempio ho 7) \* 2 poiché l’altra metà dell’albero sarà identica e sommo 1 al risultato (radice dell’albero) → il risultato è quello che chiede l’esercizio



## Ordinamento (Insertion sort, determinare il nr di confronti)

* In input abbiamo la sequenza A
* j → iterazione i=j-1…-2….-3
* Finchè i ≥0 e A[i]>A[j] aumento il numero di confronti (e decremento i per il confronto successivo)
* Tengo traccia dei confronti di ogni iterazione fino a quando la sequenza non è ordinata
* ATTENZIONE: viene fatto il confronto anche quando A[i]>A[j]
* Alla fine sommo i confronti e il risultato è ciò che chiede l’esercizio



## Ordinamento (QuickSort, dire sequenza memorizzata in A dopo k chiamate ricorsive di Quicksort fermandosi prima della chiamata k + 1 e inclusa la chiamata principale, dire anche sequenza di pivot e delle q)

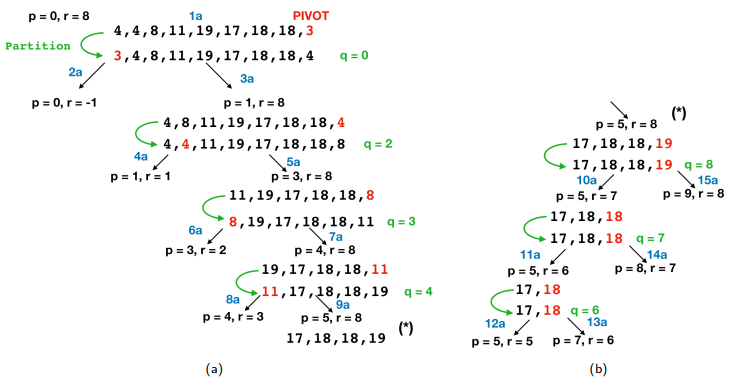
* **NB:**

Mettere la sequenza dopo il partition allineata con quella sopra

1. Scrivere tutti i numeri minori uguali al pivot nell’ordine con cui appaiono a sx del pivot
2. Scrivere il pivot
3. Se nella sequenza sopra, a destra della nuova posizione del pivot ci sono degli elementi maggiori di esso scriverli nelle stesse posizioni, lasciando eventualmente spazi vuoti
4. Negli spazi vuoti lasciati scrivere i numeri maggiori del pivot nell’ordine in cui apparivano alla sx di esso nella sequenza sopra

* Come pivot prendiamo sempre l’elemento più a sx della sequenza, la “q” invece indica la posizione del pivot dopo aver eseguito “partition” sulla sequenza stessa
* Per ogni chiamata riportare il valore di p e r. Sulla sottosequenza di sx p=p e r=q-1.

Sulla sottosequenza di dx p=q+1 e r=r



## Alberi (determinare cosa stampo se eseguo le varie visite sull’albero)

*DFS-pre* → Ordine di visita: radice - figlio sx - figlio dx

*DFS-in* → Ordine di visita: radice - figlio dx - figlio sx

*DFS-post* → Ordine di visita: figlio sx - figlio dx - radice

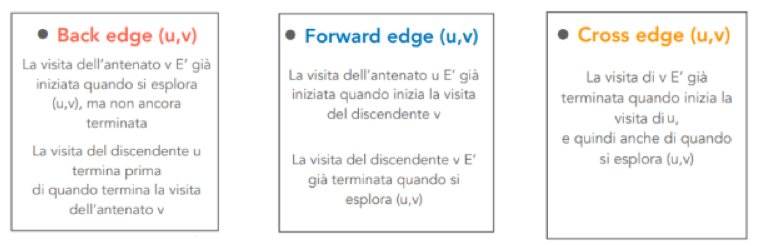
*BFS* → visito i livelli partendo dalla radice e andando verso il basso (stampo i nodi da sx verso dx)

## DFS e determinazione del tipo di arco

* Determinare l’albero di copertura del grafo dato, visitando i nodi in ordine crescente

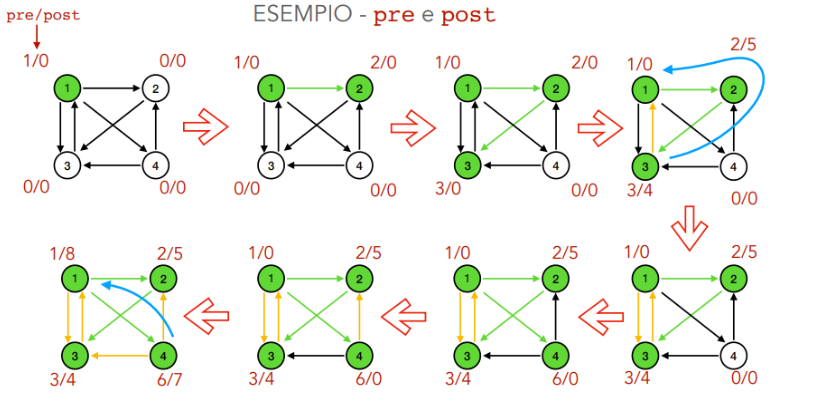
Se un arco forma un ciclo, con un nodo già visitato, non va preso

* Archi Tree: formano l’albero di copertura, partendo dal nodo sorgente sono tree tutti gli archi che vanno da un nodo la cui visita è iniziata verso un nodo non ancora raggiunto
* Archi Back: portano ad un antenato (tipo padre di padre), formano un ciclo
* Archi Forward: portano ad un discendente(tipo figli di figli)
* Arco Cross: portano né a un discendente ne a un antenato, collegano alberi di copertura diversi o due fratelli



## Grafi DFS (dire tempi di fine *post[1] … post[n]* al termine della procedura

* **NB** Gli archi che portano ad un nodo già scoperto non vanno considerati



## Cammini Minimi da sorgente singola Dijkstra (u → v)

* parto dal nodo che mi indica l’esercizio
* creo un vettore **prev** dove ad ogni nodo gli associo NULL, in ogni casella metterò i vari padri dei nodi mentre eseguo l’esercizio, resterà vuota la casella del nodo sorgente poiché non ha padre
* creo un vettore **dist** dove ad al nodo 1 il valore 0 e agli altri ထ
* partendo dal nodo iniziale scopro i nodi a cui è connesso in ordine lessicografico
* se la distanza del nodo u + il costo per raggiungere v è minore di quello presente in dist[v] lo sostituisco.
* aggiorno il vettore prev inserendo in posizione v l’identificativo del nodo u
* dopodichè faccio la stessa cosa a partire dal nodo con dist minore, ovvero priorità massima; in caso di dist uguale prendo quello lessograficamente più piccolo

**NB:** Se trovo un arco che mi porta ad un nodo con lo stesso costo con cui mi portava un arco già percorso non lo prendo

## Cammini Minimi da sorgente singola Bellman-Ford

* parto dal nodo che mi indica l’esercizio
* scrivo tutti gli archi in ordine lessicografico partendo dal nodo dato in input (solitamente 1), quindi tutti quelli che partono da 1, poi da 2 ecc….
* creo un vettore **prev** (array dei padri) dove in ogni casella metterò i vari padri dei nodi mentre eseguo l’esercizio, resterà vuota la casella del nodo sorgente poiché non ha padre
* creo una matrice **dist[]** dove sulle righe abbiamo i nodi che vanno da 1 a n (n → num nodi) e sulle colonne abbiamo le iterazioni (partono da 0 e man mano creerò altre colonne)
* inizializzo la prima colonna mettendo 0 in corrispondenza della riga del nodo src e mettendo +ထ nelle altre caselle corrispondenti agli altri nodi
* ora leggendo uno ad uno i vari archi scritti in precedenza **(u → v)** inizio ad eseguire l’algoritmo; dato che l’iterazione 0 è stata quella per inizializzare il tutto adesso parto da quella successiva, quindi scrivo nella colonna 1
* se il peso dell’arco (u → v) + costo per raggiungere u (**dist[u, iterazione - 1])** è minore di quello presente in **dist[v, iterazione - 1]** lo sostituisco
* aggiorno il vettore **prev** inserendo in posizione v l’identificativo del nodo u
* dopo aver finito gli archi controllo se due colonne della matrice sono uguali, se avviene allora mi fermo
* Altrimenti devo ripartire dal primo arco e rifare le operazioni scrivendo i risultati nell’iterazione (colonna) seguente

## Disjoint set (mostrare graficamente stato dell’insieme dopo le varie operazioni)

* Creo tanti nodi con self-loop quanti sono i makeset
* Le find non influenzano il risultato, si possono saltare
* le uniche operazioni che influenzano la rappresentazione sono le *union*
* le union sono composte da due parametri, che indicano l’id di uno dei nodi dell’albero a cui appartengono:

se i rispettivi alberi hanno la stessa altezza, quello con rappresentante più grande dei due sarà la radice dell’albero mentre il più piccolo suo figlio (ricorda → mantieni self loop in quello più grande) ⇒ l’altezza aumenterà di 1

altrimenti quello di altezza minore diventa figlio di quello con altezza maggiore

## Determinare MST con Kruskal

- ordino archi in modo lessicografico in base al peso (prima tutti gli archi di peso 1, poi 2 ecc..)

- partendo dal primo arco controllo se genera un ciclo, se non lo crea allora lo coloro altrimenti non lo considero

- faccio il punto precedente per ogni arco, oppure fino a quando tutti i nodi sono visitati

## Determinare MST con Prim

- parto dal nodo che mi indica l’esercizio

- creo un vettore **prev** dove ad ogni nodo gli associo NULL

- creo una coda **Q** composta da coppie di valori (nodo, costo)

- la coda parte con il nodo che ci dice l’esercizio associato ad un costo 0 e successivamente i nodi in ordine con costo associato +ထ

- partendo dal nodo iniziale verifico a quali altri nodi arriva e modifico il costo all’interno della coppia dei nodi corrispondenti in base al costo dell’arco, in più modifico anche il corrispettivo prev (modifico il nodo padre, quindi il nodo da cui si passe per raggiungere il nodo preso in considerazione)

- successivamente elimino il nodo iniziale dalla coda e rieseguo le stesse operazioni del punto precedente prendendo come nodo di partenza quello con priorità più alta (costo più basso).

- man mano che vado avanti mi devo ricordare di segnare sul grafo i nodi che ho visitato grazie alla coda e segnare l’arco che congiunge tale nodo al padre in modo da creare l’MST che chiede l’esercizio

- ATTENZIONE → non devo tenere conto dei costi precedenti a contrario di Dijkstra e Ford

## Codifica Huffman senza file

- Scriviamo le varie coppie (carattere, frequenza)

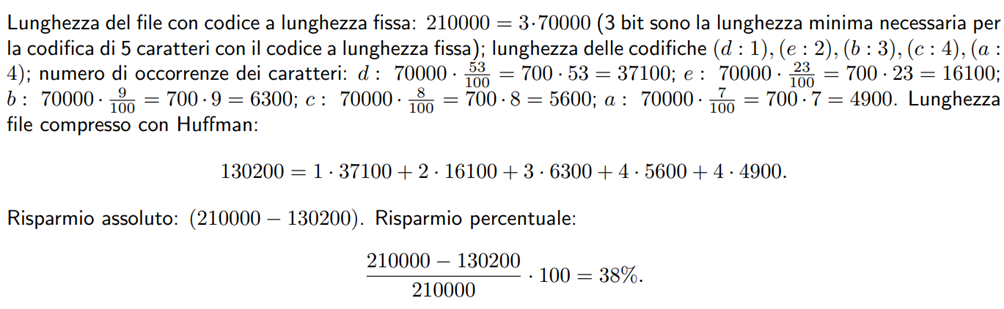
- Uniamo sotto la stessa radice i due caratteri con frequenza minore, la radice sarà composta dalla coppia ( -, freq1 + freq2)

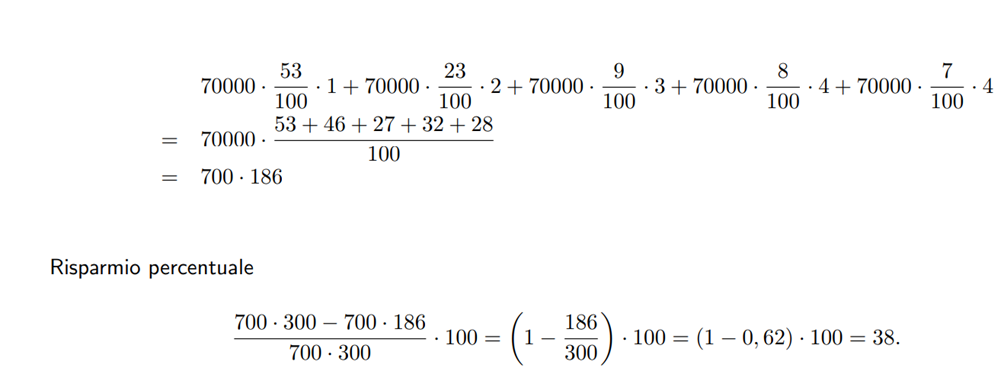
- Continuiamo procedimento fino ad arrivare ad avere solo un nodo del tipo ( - , somma di ogni freq.)

- Mettiamo degli zeri sugli archi che vanno dalla radice al nodo di sx e degli 1 sull’arco che va al nodo di dx

- Partendo dalla radice e raggiungendo il nodo con all’interno un carattere ci segniamo i vari 0 e 1 sugli archi, tale sequenza sarà la sua codifica

## Codifica Huffman con file

****

****



## Longest Increasing Subsequence

- Creo vettore L per contenere lunghezza delle sottosequenze

- Prendo ad ogni step il primo numero della sequenza A. che cancellerò da essa quando ho finito le operazioni da eseguirgli sopra

- Controllo andando verso sx quali numeri sono più piccoli del numero preso in considerazione

- Controllo il valore più alto contenuto in L nelle posizioni corrispondenti ai numeri più piccoli del numero scelto inizialmente

- Scrivo in L il valore più alto incrementato di 1

- Faccio così per tutti gli altri elementi della sequenza iniziale fino a completare L

## Zaino con ripetizione

- Creo vettore K che va da 0 fino a W (capacità che ci dice l’esercizio)

- Creo vettore Item per tenere traccia degli oggetti durante i calcoli

- Metto dentro le posizioni di K tanti 0 fino a quando il suo indice è < del minor peso dei vari oggetti, successivamente posso partire con i calcoli

- Per ogni indice di K devo eseguire la seguente formula e inserire il valore ottenuto al suo interno…



…e mi devo ricordare di aggiornare Item con il numero di oggetto corrispondente

- Dopo tutti i calcoli e dopo aver completato di sovrascrivere l’array K posso finalmente trovare gli oggetti ideali per lo zaino di capacità W

- Allora partendo dall’ultimo elemento del vettore K (indice = W) verifico l’oggetto corrispondente nell’array Item e faccio W = W-peso\_oggetto, il risultato sarà l’indice di K su cui rieseguire tale formula. Inoltre mi segno l’oggetto utilizzato

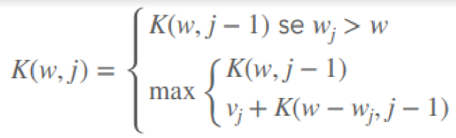
- Alla fine dei calcoli avrò una sequenza di oggetti utilizzati, essa è ciò che mi chiedeva inizialmente l’esercizio

## Zaino senza ripetizione

- Creo matrice W+1 x j+1

- Metto 0 in ogni casella della prima colonna e prima riga

- Inizio i calcoli, le caselle vanno riempite colonna per colonna utilizzando la seguente formula…



- Quando tutte le caselle sono state riempite allora prendo l’elemento nella casella in posizione (W, j)

- Tale valore è ciò che ci chiede l’esercizio

## Matrice delle distanze con Floyd-Warshall

- Creo matrice i x j (i righe, j colonne) che avrà k = 0

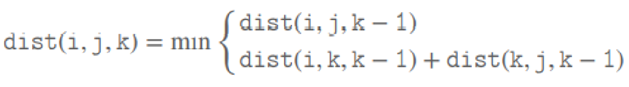
- Tali indici vanno da 1 fino al numero di nodi nel grafo

- La matrice la inizializzo con i pesi degli archi che vanno dal nodo in posizione i al nodo con posizione j

- Se l’arco non esiste metto +infinito altrimenti nel caso in cui i = j metto 0

- Dopo aver fatto ciò creo una nuova matrice (k = 1) partendo dalla prima, da essa prenderò la prima riga, la prima colonna e la diagonale

- Successivamente dovrò calcolare gli altri valori nelle posizioni vuote tramite la seguente formula…



K = numero della matrice

- Proseguo con altre matrici da cui prenderò dalla matrice precedente il numero di riga e colonna corrispondente a K e la diagonale

- Proseguo fino ad aver completato la matrice con il K che mi chiede l’esercizio

## Brute Force (string matching)

- Ad ogni iterazione prendo un carattere della sequenza e la confronto con il pattern

- Tengo traccia del numero di confronti, il confronto termina quando trovo un carattere che non “matcha” il pattern oppure l’esito di tutti i confronti mi permette di trovare completamente il pattern nella sequenza

- Termino di eseguire il brute-force quando nella sequenza non ho abbastanza caratteri per contenere l’eventuale pattern

- Sommo tutti i risultati dei confronti per avere il numero totale dei confronti

## Best prefix (string matching)

- Creo tabella con 3 colonne: prima colonna → j (va da 0 a numero caratteri pattern), seconda colonna → p[0…j-1] (scrivo andando verso il basso la sequenza con un carattere in più), terza colonna → s (numero di shift, è ciò che ci chiede l’esercizio)

- Per ogni caso di mismatch devo calcolare lo shift minimo e tale numero devo inserirlo nella corrispondente riga di s

- Vedi immagine seguente per procedimento…

