PROGRAMMAZIONE II

Liste: Una lista è una sequenza di elementi non indicizzabile dove ogni elemento contiene un campo valore e un puntatore all'elemento successivo. Sulle liste è possibile la cancellazione che ha complessità da O(1) a O(n) e tre tipi d inserimento:

-**inserimento in testa:** l'elemento viene sempre inserito in testa alla lista, questa procedura ha complessità O(1);

-**inserimento in coda:** l'elemento viene inserimento sempre come ultimo elemento della lista, questa procedura ha complessità O(n) poiché ogni volta bisogna scorrere tutta la struttura;

-**inserimento in mezzo:** l'elemento viene inserito in modo ordinato nella lista, bisogna quindi scorrere tutta la struttura fino a trovare il giusto posto per l'elemento da inserire, la complessità di questa procedura va da O(1) a O(n).

Esistono diversi tipi di lista, i più usati sono la linked list, la double linked list e la lista circolare.

Inserimento ricorsivo: avremo due metodi, uno si occuperà soltanto di richiamare la procedura sulla testa solo la prima volta e il secondo sarà una procedura con 3 casi base ovvero:

-quando il nodo passato iter è NULL effettuo inserimento in **lista vuota**;

-quando il nodo iter passato punta alla **testa** e il suo valore è minore rispetto alla testa, effettuo un **inserimento in testa**;

-quando l'iteratore non è NULL e il valore del suo successivo è minore del valore che voglio inserire, effettuo una **chiamata ricorsiva** su iter->succ;

-altrimenti, se non rientro in nessuno dei casi precedenti, farò **inserimento in coda**.

Pila: Una pila è una struttura dati elementare basata sulla politica **LIFO**. Essa è accessibile soltanto dalla testa tramite un puntatore che viene aggiornato sempre all'elemento più alto nella pila. Sulla pila è possibile effettuare soltanto due operazioni:

-**push**: permette di inserire un elemento in testa alla pila, aggiornando il puntatore al primo elemento;

-**pop**: permette di estrarre l'elemento in testa alla pila e di visualizzarlo, inoltre, aggiorna la testa all'elemento immediatamente sottostante a quello estratto.

Coda: La coda è una struttura dati elementare basata sulla politica **FIFO**. Essa è accessibile sia dalla testa che dalla coda e permette l'inserimento e l'estrazione in ordine degli elementi. Sulla coda è possibile effettuare soltanto due operazioni:

-**enqueue**: permette di inserire un nuovo elemento in coda alla coda;

-**dequeue**: permette di estrarre l'elemento in testa alla coda, inoltre sposta il campo testa indietro di una posizione.

Spesso l'implementazione della coda avviene tramite liste o tramite **array circolari**, questi ultimi sono array che vengono indicizzati in base all'operazione modulo sulla lunghezza. Questo particolare tipo di array viene utilizzato per evitare che una volta effettuato il dequeue degli elementi le prime posizioni rimangano vuote, in questo modo, inizio e fine dell'array sono solo virtuali e non rispettano gli indici 0 e n-1.

Albero: un albero è un particolare **grafo orientato** che ha soltanto un nodo in entrata k nodi uscenti. Se un nodo non ha archi uscenti si dice **foglia** e se non ha archi entranti si dice **radice**. Poiché in un albero non ci possono essere nodi con due o più archi entranti, per ogni albero esiste una sola radice, inoltre la **direzione degli archi** è univoca e verso il basso. I nodi che non sono ne radice ne foglia si dicono **nodi interni**, inoltre, il nodo da cui parte un arco si dice **padre** e quello in cui arriva si dice **figlio**, due nodi figli dello stesso padre si dicono **fratelli** ma da un fratello non posso raggiungere l'altro.

Definizione ricorsiva: Da ogni nodo non foglia di un albero si dirama un sottoalbero, quindi si intuisce la natura ricorsiva di questa struttura. Dato un nodo, i nodi che appartengono allo stesso sottoalbero si dicono **discendenti** e i nodi che si trovano nel suo percorso fino alla radice si dicono **ascendenti**. Abbiamo quindi che:

Un albero è un insieme di nodi che:

-o è vuoto (non ha nodi);

-oppure ha un nodo chiamato radice da cui discendono zero o più sottoalberi, essi stessi alberi.

Il **livello di un nodo** è la sua distanza dalla radice. **La radice ha livello 0**, i suoi figli 1 eccetera. I fratelli sono tutti sullo stesso livello ma non tutti i nodi dello stesso livello possono essere fratelli. La distanza di un nuovo nodo, sarà uguale alla distanza del nodo precedente +1.

La **profondità di un albero** è la lunghezza del cammino più lungo dalla radice ad una foglia. Comunque ogni nodo ha la propria profondità e si calcola a partire dal campo padre a salire.

Un albero di profondità h si dice **equilibrato** o **bilanciato** se, dato k numero massimo di figli che un nodo può avere, ogni nodo interno, ha esattamente k figli. Un albero è quindi bilanciato se inoltre tutte le foglie si trovano sullo stesso livello.

ALBERO BINARIO

Un albero si dice binario se ogni suo nodo può ammettere al più due figli, figlio destro e figlio sinistro. Ad ogni livello n, un albero binario può contenere al più 2^n nodi. Il numero totale di nodi di un albero di profondità n è al massimo $\mathbf{2^{n+1}-1}$ e il livello n+1 avrà il doppio dei nodi rispetto al livello n.

ALBERO BINARIO DI RICERCA (BST)

I BST sono alberi in cui vengono memorizzati **dati confrontabili** che godono di una **proprietà fondamentale:**

-Dato x nodo generico, se y è un nodo nel sottoalbero sinistro di x, allora y < x, altrimenti y si troverà a destra.

Il criterio viene applicato ad ogni sotto albero in **modo ricorsivo**, scendendo via via nella struttura.

VISITE DI UN BST

Esistono principalmente due strategie notevoli:

-**PreOrder**: visita prima la radice, poi il sottoalbero sinistro e infine il sottoalbero destro in

modo ricorsivo fino a quando non trova nodi NULL;

-**InOrder**: visita prima il sottoalbero sinistro, poi la radice e infine il sottoalbero destro.

Questa procedura, se applicata ad un BST, restituisce i valori in **ordine crescente**.

-**PostOrder**: visita prima il sottoalbero sinistro, poi il destro ed infine la radice.

Ovviamente per esaminare tutto l'albero le procedure vanno applicate alla radice dell'albero.

InOrder iterativo: è necessario usare uno **stack** per il **backtracking** che memorizza elementi di tipo Nodo.

-imposto un Nodo current alla root e un flag a false;

-fino a quando il flag è false, se current non è NULL, push(current) e sposto current a sx, altrimenti, se lo stack è vuoto metto il flag a true, altrimenti, faccio il pop dello stack, stampo il suo valore e sposto current a dx.

INSERIMENTO IN UN BST

I nuovi elementi di un BST vengono sempre inseriti come nuove **foglie**, mantenendo la proprietà fondamentale dei BST.

Ogni volta che voglio inserire un nuovo elemento, posso cercare scendendo nella struttura il posto giusto in cui inserire il nuovo valore, scegliendo ad ogni biforcazione se scendere nel sottoalbero destro o in quello inserito. Quando non ci sarà più nessun nodo sottostante, avrò trovato il posto giusto per la nuova foglia.

Nella procedura di inserimento è importante anche valutare il **caso base** in cui l'albero è vuoto e il primo elemento che sarà inserito diventerà radice.

COMPLESSITA'

La procedura non è lineare perché non devo scorrere tutto l'albero per inserire l'elemento ma basta scorrerne una parte, farò quindi un numero di passi minore o uguale alla profondità dell'albero stesso.

Questa procedura è molto efficiente in caso di alberi ben bilanciati e con una buona topologia, in questo caso la procedura andrebbe a **O(log n**), altrimenti, in caso di alberi sbilanciati o degeneri, la procedura potrebbe richiedere anche tempo **O(n)** con n altezza dell'albero.

RICERCA IN UN BST

L'idea è quella di visitare non tutto l'albero ma soltanto gli elementi che potrebbero contenere il valore che stiamo cercando. Se l'elemento non è presente nell'albero andremo oltre le foglie a valori NULL.

COMPLESSITA'

La complessità della procedura di ricerca è pari alla profondità dell'albero h. Per un albero a n elementi la complessità varia da **log n** a **n**, a seconda della topologia.

MINIMO E MASSIMO DI UN BST

Il minimo di un BST è **l'elemento più a sinistra** dell'albero mentre il massimo è l'elemento più a destra di esso. La complessità di ricerca dipende dall'altezza dell'albero**: T(n) = O(h).**

CANCELLAZIONE DI UN NODO DI UN BST

La cancellazione di un nodo è la procedura più complessa da applicare su un BST in quanto bisogna mantenere l'invariante fondamentale. La cancellazione di un nodo si divide in 3 sotto casi:

-**CASO 1**: l'elemento è una foglia (non ha figli);

-**CASO 2:** l'elemento ha un solo figlio (destro o sinistro);

-**CASO 3:** l'elemento ha entrambi i figli;

-**3A**: il successore del nodo da eliminare figlio destro diretto;

-**3B**: il successore del nodo da eliminare non è figlio diretto ma minimo del suo sottoalbero destro.

CASO 1: bisognerà semplicemente cancellare la foglia e sistemare il puntatore del padre, in modo che punti a NULL;
CASO 2: basterà cancellare il nodo e far prendere all'unico figlio il posto del padre, sistemando i puntatori di padre e figli.

CASO 3: bisogna cercare il successore y del sottoalbero destro del nodo da cancellare z.

-se y è figlio destro di z, semplicemente y prenderà il posto z;

-se y non è figlio destro di z ma solo un **discendente**, bisognerà prima trapiantare y e y->right, poi collegare i figli destri di z a y, trapiantare z e y e infine collegare i figli sinistri di z a y.

OSS: il successore non può avere figli sinistri in quanto sarà il minimo del sottoalbero.

La codifica della cancellazione si compone di due procedure:

-**Trapianta**: permette di spostare sottoalberi rimpiazzando un sottoalbero di radice u con un sottoalbero di radice v. Trapianta sposta anche sottoalberi vuoti;

-**Cancella**: esamina i casi, chiama trapianta e modifica gli opportuni puntatori.

SUCCESSORE DI UN BST

Per il calcolo del successore abbiamo due casi da considerare:

-il sottoalbero destro di x **non è vuoto**: torniamo il minimo del sottoalbero destro;

-il sottoalbero destro **è vuoto**: dobbiamo ricercare il minimo antenato di x il cui figlio sinistro è anch'esso antenato di x (ogni nodo è antenato di se stesso).

Grafo: si dice grafo un insieme di archi e nodi **G = {V, E}** in cui gli archi possono essere direzionati, se hanno un verso, oppure no, se si possono percorrere in entrambi i versi,

V è l'insieme dei **vertici** ed **E** è l'insieme degli **archi**.

Dato un arco (u,v), se G è **direzionato**, l'arco u si dice **uscente** mentre l'arco v si dice **entrante**. Se (u,v) appartiene ad E, v è adiacente a u.

Nei grafi non direzionati, E consiste di coppie non ordinate di nodi e **l'adiacenza è simmetrica**, ovvero vale per entrambi i versi.

DEFINIZIONI:

-**Grado di un nodo**: numero di archi entranti (non direzionato);

numero di archi entranti + numero di archi uscenti (direzionato).

-**Cammino**: di lunghezza k se da u a v => v_0, v_1, ..., v_k => u=v_0 e v = v_k.

-Un nodo v è **raggiungibile** da u se esiste un cammino da u a v.

-Un cammino è **semplice** se tutti i vertici in esso sono distinti.

-Dato un cammino da v_0 a v_k, un **sottocammino** è la sequenza di vertici di un cammino v_i, … v_j con 0<= i <= j <=k.

-Un ciclo si dice **semplice** se tutti i suoi nodi sono distinti.

-Un grafo senza cicli si dice **aciclico**.

-Un grafo non direzionato si dice **connesso** se ogni coppia di vertici è unita da un cammino.

-In un grafo possiamo definire come **classi di equivalenza** tutti i nodi connessi tra di loro, le classi vengono indicate come **componenti connesse**.

-Un grafo non direzionato è **connesso** se ha un'unica componente connessa.

-Un grafo è **fortemente connesso** per ogni coppia di vertici (u,v) se esiste un cammino da u a v e uno da v a u.

-Le componenti fortemente connesse sono determinate dalla relazione "**sono mutuamente raggiungibili".**

-G' = {V', E'} è un **sottografo** di G se V' è sottoinsieme di V e E' è sottoinsieme di E.

-Un grafo non direzionato è **completo** se ogni coppia di vertici è adiacente.

RAPPRESENTAZIONE DI UN GRAFO

Esistono due metodi principali per la rappresentazione di un grafo:

-**Liste di adiacenza**: utile per rappresentare grafi sparsi con pochi archi, la complessità è O(|V|+|E|).

-**Matrici di adiacenza**: utile con grafi molto denti, richiede sempre $O(|V|^2)$.

VISITA DI UN GRAFO

Avremo bisogno di algoritmi più sofisticati per la visita dei grafi, poiché, a differenza degli alberi, i cui nodi sono tutti raggiungibili, in un grafo potrebbero esserci cicli o archi e nodi non raggiungibili. Per la visita si utilizzano due algoritmi fondamentali:

-**Ampiezza**: partendo da un nodo centrale, mi espando prendendo cerchi concentrici sempre con raggio maggiore.

-**Profondità**: partendo da un nodo esamino i discendenti fissando punti di backtracking.

BREADTH FIRST SEARCH (BFS)

Dato un vertice s, esploriamo il grafo per scoprire ogni vertice v raggiungibile da s. Implicitamente, quest'algoritmo calcola la distanza di ogni v da s e inoltre, produce un **Breadth first tree** (**BFT**), ovvero un albero in cui il cammino da s a v rappresenta lo **shortest path** tra i due nodi del grafo.

Ogni nodo assumerà un **colore** diverso che indicherà il suo stato in quel preciso momento dell'esecuzione dell'algoritmo. Inizialmente i nodi saranno tutti colorati di **bianco**, quando un nodo viene scoperto per la prima volta però, esso si colora di **grigio** e viene inserito in una **coda di supporto**. Quando un nodo esce dalla coda di supporto per essere esaminato, esso diventa **nero**, in modo da non essere più inserito in coda ed evitare loop.

I nodi grigi quindi, rappresentano il **punto di frontiera** tra ciò che è stato scoperto e ciò che non lo è stato ancora; i nodi neri invece rappresentano i nodi già esaminati e possono essere adiacenti solo a nodi non bianchi.

La visita terminerà con tutti i nodi neri e la coda di supporto vuota. Alla fine avremo costruito quindi l'**albero dei cammini minimi** in quanto ogni nodo tranne uno (il primo), avrà un solo arco entrante ed n archi uscenti.

L'unica **problematica** della BFS è che quando il grafo è composto da più di una componente connessa, la procedura esaminerà soltanto la componente contenente il nodo da cui essa è partita, ignorando le altre in quanto irraggiungibili.

COMPLESSITA'

La fase di inizializzazione costa ordine di |V| + il costo del while. Il ciclo while dipende dalla dimensione delle liste di adiacenza. La somma delle liste è 2|E| nel caso di grafi non orientati e |E| per quelli orientati. La complessità totale è quindi **O(|V|+|E|).** La stessa procedura implementata con una matrice avrà invece sempre complessità **O($|V|^2$).**

BREADTH FIRST TREE (BFT)

La procedura BFS costruisce un albero chiamato **grafo dei predecessori** in cui ad ogni nodo è associato un predecessore.

-$V_p = \{ v \in V : p[v] != NULL\}$

-$E_p = \{(p[v], p) \in E : v \in V_p, v != s\}$

Gli archi $\in E_p$ Sono chiamati **Tree-Edges.**

PRINT PATH

Supponendo di avere già un BST possiamo ottenere i cammini minimi tra la root e un nodo.

La procedura in modo ricorsivo richiama se stessa passando come parametro il padre del nodo appena esaminato.

La complessità di questa procedura dipende dalla lunghezza del cammino.

DEPTH FIRST SEARCH (DFS)

Il grafo viene visitato in profondità piuttosto che in ampiezza.

-Gli archi vengono esplorati a partire da un nodo v che:

-sia stato scoperto più di recente;

-abbia ancora archi uscenti non esplorati.

-Quando gli archi uscenti di v terminano, si fa **backtracking**, esplorando altri eventuali archi uscenti dal nodo precedente a v.

-Il processo è ripetuto fin quando vi sono nodi da esplorare.

DEPTH FIRST FOREST (DFF)

Come per il BFS, anche con la DFS si viene a creare un grafo dei predecessori G_p dove:

-$V_p = V$;

-$E_p = \{(p[v], v)$ in E : $v \in V$, $p[v] != NULL\}$

G_p non è più un albero ma una foresta in cui ogni albero rappresenta una componente connessa del grafo.

TIMESTAMPS

la DFA marca **temporalmente** ogni vertice visitato. Ogni vertice ha due **etichette**, quella di entrata e quella di uscita, queste etichette vengono salvate in due vettori:

-**d[v]:** registra quando il nodo è stato scoperto (bianco -> grigio).

-**f[v]:** registra quando la ricerca finisce di esaminare la lista di adiacenza di v (grigio-> nero).

Per le timestamps vale una **proprietà fondamentale**, ovvero, per ogni $v \in V \Rightarrow d[v] < f[v]$.

ALGORITMO

L'algoritmo si compone di due procedure, una iterativa e una ricorsiva, qui non esiste un vettore delle distanze ma una variabile temporale che si aggiorna ad ogni visita e permette di assegnare timestamps di uscita e ingresso a ogni nodo.

DFS: inizializza i nodi a bianco e il vettore dei precedenti a NULL, inoltre setta il tempo a 0;

Per ogni nodo dell'insieme E:

se il nodo è bianco, chiama la procedura ricorsiva (formando implicitamente backtracking), altrimenti incrementa un ciclo for che analizza l'elemento successivo. Questo for serve a non trascurare nessun nodo, anche se non esistono percorsi verso di esso, questo algoritmo visita quindi tutto il grafo anche se composto da diverse componenti connesse.

DFS Visit: imposta il colore del nodo su cui è stata chiamata la procedura come grigio, inoltre setta anche il suo timestamp di entrata a timestamp attuale+1.

Per ogni elemento della lista di adiacenza del nodo su cui è stata chiamata la visit:

se il nodo è bianco, chiama la funzione ricorsivamente sul nodo che si sta esaminando. Alla fine delle chiamate ricorsive (quando non ci saranno più nodi nella lista di adiacenza), grazie al backtracking, torniamo il controllo al chiamante e coloriamo il nodo di nero, impostandone anche il suo timestamp di uscita a quello attuale+1. Proprio per questo motivo vale la proprietà fondamentale dei timestamps.

OSS: con più componenti connesse, avremo più alberi appartenenti alla DFF, il numero di alberi sarà uguali al numero di componenti connesse, inoltre, in base alla scelta del nodo di partenza su cui viene applicata la DFS, si genereranno diversi alberi, a partire dallo stesso grafo.

COMPLESSITA'

La fase di inizializzazione ha complessità $O(|V|)$.

Per ogni vertice eseguiamo il for e controlliamo se il nodo è bianco, in tal caso eseguiremo la visit, quindi, indipendentemente da dove, la chiamata alla visit viene fatta una volta sola per ogni nodo.

Nella visit, il costo del for dipende dalla lunghezza della lista di adiacenza di ogni nodo, quindi il numero di passi totali è proporzionale alla somma della dimensione di tutte le liste di adiacenza, quindi, per grafi direzionati sarà O(|E|).

Nel **caso pessimo**, la complessità della DFS è uguale a quella della BFS, ovvero **O(|V|+|E|).**

Nel **caso migliore** invece, (grafo senza archi), |E| = 0, quindi O(|V|+|E|) = **O(|V|).**

Considerando una matrice il costo sarà a prescindere **$O(|V|^2)$.**

CLASSIFICAZIONE DEGLI ARCHI

Tree edges: (u, v) è un tree edgesse v è scoperto per la prima volta quando si è esplorato l'arco (u, v). Essi sono gli archi che collegano i nodi dell'albero.

Back edges: (u, v) collega u ad un antenato v (non diretto) nel DFT.

Forward edges: (u, v) collega u con un discendente v (molto più avanti) nel DFT.

Cross edges: tutti gli altri tipi di archi, come quelli tra alberi.

TOPOLOGICAL SORT

Un'applicazione del DFS è **l'ordinamento topologico** che può essere effettuato solo su un grafo diretto e **aciclico** (senza back edges).

Riusciamo a capire che un grafo è **ciclico** se nella lista di adiacenza di un nodo bianco, troviamo un nodo grigio, ciò significa che noi stiamo obbligatoriamente venendo da lui, in quanto il nodo grigio è ancora in esplorazione (sarebbe stato nero altrimenti), ciò significa che noi siamo antenati del nodo grigio **(pi fozza),** inoltre, dato che il nodo grigio è nella lista di adiacenza del nodo bianco, sappiamo che esiste un arco dal nodo bianco a quello grigio. In conclusione, dal nodo grigio è possibile raggiungere il nodo bianco e vicersa, ciò conferma l'**esistenza di un ciclo all'interno del grafo**.

Il topological sort non è un ordinamento classico basato su dei valori ma un "**appiattimento**" del grafo, infatti esso può essere visto come un **ordinamento dei vertici su una linea orizzontale**.

Dallo stesso grafo, in base al risultato della DFS, posso estrapolare **diversi topological sort** che però godono tutti della stessa **invariante**:

- **se esiste l'arco (u, v) $\in V_p$ allora u viene prima d v nel sorting.**

Siano dati i nodi u e v, e le timestamp di ingresso d[] e le timestamp di uscita f[], abbiamo che v è un **discendete** di u se d[u] < d[v] < f[v] <f[u].

Per **implementare** il TS facimente è sufficiente **impilare** tutti i nodi ogni volta che gli viene assegnato il timestamps di uscita, essendo esso il minore, basterà poi stampare le pop

Gli elementi vengono ordinati in modo descrescente rispetto al timestamps di uscita applicato dalla DFS.

La **complessità** del topological sort dipende da quella del DFS dato che l'ordinamento può essere fatto con l'inserimento in testa ad una lista che ha complessità O(1).

Possono esistere minimo 0 topological sort per lo stesso grafo, ovvero quando il grafo è ciclico, invece possono esistere fino a |V|! topological sort quando E è vuoto, quindi non esistono archi.

COMPONENTI FORTEMENTE CONNESSE

La DFS permette di **decomporre** un grafo nelle sue componenti fortemente connesse.

Una **proprietà fondamentale** è che le **timestamps** di una componente, sono tutte **minori** rispetto alle timestamps di un'altra, possiamo quindi organizzare i nodi sulla base di esse.

Per stabilire l'algoritmo, oltre alla DFS, servirà stabile il **grafo trasposto** (tramite matrice trasposta), ovvero un grafo dove l'arco (u, v) diventa (v, u). **Ovviamente G e G^T avranno le stesse componenti connesse**.

COMPLESSITA'

La complessità del DFS è O(|V|+|E|), qui lo chiamiamo due volte su dei grafi che hanno gli stessi n nodi, il terzo passo ha invece costo costante. Il problema è dunque la trasposizione del grafo che con liste costa O(|V|+|E|) mentre con matrici $O(|V|^2)$.

Asintoticamente quindi la procedura ha complessità O(|V|+|E|) per le liste e $O(|V|^2)$ per le matrici.

PSEUDOCODICE

DFS(G) (impila i nodi esaminati)

Calcolo G^T

DFS G^T (secondo la pila creata da DFS)

Stampa i vertici di ogni albero creato.

Il calcolo del grafo trasposto serve a capire se una componente è connessa o meno.

Un grafo è ciclico quando scorrendo la lista di un nodo non ancora esplorato trovo un nodo grigio.

ALGORITMI DI ORDINAMENTO E DI RICERCA

Gli algoritmi di ordinamento elementari sono una tipologia di algoritmi **basata sul confronto** e lo scambio in-place degli elementi di una struttura dati.

SWAP SORT
Confronta il prima elemento con tuti i successivi e fa uno scambio se l'ordine non è corretto,

Alla fine della prima iterazione, in prima posizione avremo l'elemento più piccolo della'array e dopo n iterazioni le prime n celle saranno ordinate. A ogni iterazione partiremo dalla cella successiva poiché quella precedente è già ordinata.

COMPLESSITA'

Quest'algoritmo ha complessità **$O(n^2)$** nel caso peggiore, è **in-place** e non esiste un caso migliore in quando l'algoritmo farà comunque sempre gli stessi passi.

INSERTION SORT

Ogni volta che ricevo un elemento lo ordino nella prima metà dell'array, facendo scorrere il resto con un ciclo for.

Useremo un while per inserire i nuovi elementi, ovvero, scorrerò l'array al contrario fino a quando non troverò un elemento minore che andrò a posizionare davanti all'ultimo elemento da ordinare.

A ogni ripetizione gli elementi saranno spostati tutti di una posizione in avanti.

COMPLESSITA'

Il while potrebbe avere complessità pari ad un ciclo for nel caso in cui dovessi inserire l'elemento nella prima posizione ogni volta, quindi avrei **$O(n^2)$.**

Se l'array fosse già ordinato però, la **guardia** del while non scatterebbe abbassando la complessità a **O(n).**

BUBBLE SORT

Per ogni iterazione, si confrontano coppie di elementi adiacenti e si scambiano i valori quando il primo elemento è maggiore del secondo. Alla fine di ogni iterazione, l'elemento più grande dell'array sarà risalito all'ultima posizione disponibile e il controllo si effettuerà su porzioni dell'array più piccole.

COMPLESSITA'

Nel bubble sort pure la complessità è sempre **$O(n^2)$** poiché non esistono controlli

E' possibile integrare una **sentinella** che si attiva se durante la prima iterazione non avviene nessuno scambio, limitando il secondo for, in questo modo, in caso di array ordinato, la complessità del bubble sort scenderà a **O(n).**

Gli algoritmi di ordinamento basati su confronti hanno complessità $\Omega(n \log n)$ sempre.

MERGESORT

Questo algoritmo è basato sul paradigma **Divide et Impera**, secondo il quale un problema di grandi dimensioni può essere scomposto in problemi più facilmente risolvibili, per poi unire le soluzioni. Esso si divide in tre fasi:

-**Decomposizione**: se la sottosequenza contiene almeno due elementi, essa viene divisa in 2;

-**Ricorsione**: riordina ricorsivamente ogni sottosequenza;

-**Ricombinazione**: Le soluzioni delle sottosequenze ordinate vengono ordinate in un'unica sequenza.

COMPLESSITA'

Il mergesort ha complessità **n log n**, la procedura di **merge** ha complessità **O(n)** e non esistono casi migliori. (dimostrazione)

Inoltre il mergesort non è una procedura inplace inquanto la procedura merge crea un array di supporto, portando la **complessità spaziale a O(n).**

QUICKSORT

L'idea è quella di dividere gli n elementi in **3 partizioni**: la sinistra, in centro (pivot) e la destra, lo scopo è quello di ordinare tutti gli elementi più piccoli del **pivot** alla sua sinistra e quelli più grandi alla sua destra. Effettuando la procedura ricorsivamente ordineremo array sempre più piccoli fino ad ordinare l'intero array.

Il pivot potrà essere scelto randomicamente oppure come ultimo elemento dell'array.

La procedura implementata consiste nello scorrere l'array alla ricerca di elementi minori del pivot, e spostarli chiamando un indice i solo in caso di bisogno, l'ultimo passo consterà nel portare il pivot al centro dell'array.

COMPLESSITA'

Si ipotizza che scegliendo sempre il pivot migliore la complessità sia **n log n,** esistono però dei casi peggiori come la **scelta sempre estremamente sbilanciata del pivot** oppure un **array ordinato al contrario**, questi due casi portano la complessità del quicksort a **$O(n^2)$.**

Inoltre, il quicksort **effettua dalle log n a n chiamate ricorsive**, in base alla scelta del pivot. Nel caso di scelta sempre sbilanciata del pivot, l'albero risulterà degenere e sbilanciato solo verso una direzione. Il numero di chiamate corrisponde all'**altezza dell'albero di ricorsione.**

RICERCA SEQUENZIALE

Quest'algoritmo confronta tutto l'array per la ricerca di una chiave e restituisce l'indice se la trova, altrimenti -1. Ovviamente la complessità di quest'algoritmo è **O(n).**

RICERCA BINARIA

Quest'algoritmo è applicabile solo se l'array è già ordinato, e, partendo dal centro del vettore, si sposterà a destra o a sinistra a seconda del peso della chiave, l'algoritmo si fermerà quando l'indice destro e sinistro saranno uguali, in quel caso se l'elemento puntato sarà quello cercato, ritornerà la posizione, altrimenti -1.

COMPLESSITA'

L'algoritmo ha complessità log n poiché opera sempre su metà array, si dimostra che **log n = Ω(log n)** .

SELECTION SORT RICORSIVO

```
Void selection( int*a, int n){
For(int j=1;j<n;j++)
   If(a[0] > a[j]     scambia(a[0], a[j];
}
For(int i=1;i<n;i++){
Selection(a+1, n-i);
}
```

Questa procedura sfrutta l'aritmetica dei puntatori.

CALCOLO DEL MCD CON RESTI SUCCESSIVI

-n se n<m

-MCD(n,m) se m<n

-MCD(n,m%n) negli altri casi

Quest'algoritmo ha complessità phi^k

TORRI DI HANOI

```
void Hanoi(iniziale, centrale, finale, n){
   if(n==1)    cout<<"muovo il disco"<<iniziale<<"a "<<finale<<endl;
   else{
      Hanoi(iniziale, centrale, finale, n-1);
      cout<<"muovo il disco"<<n<<"dal piatto"<<iniziale<<"a quello"<<finale<<endl;
      Hanoi(centrale,iniziale, finale, n-1);
   }
}
```

Quest'algoritmo ha complessità 2^n-1.

ALGORITMI PER LOG E ELEVAMENTI A POTENZA

```
int root(int a){
   int i=0;
   int s=0;
   while(i < a){
      s = i*i*i;
      if(s == a)   return i;
      i++;
   }
   return -1;
}

int log(int base, int argomento){
   int conta = 0;
   int pow = 1;
   while(pow < argomento){
      pow *= base;
      conta++;
   }
   if(pow != argomento)    return -1;
   else    return conta;
}
```