Algoritmi e strutture dati

Il costo degli algoritmi non può essere misurato in base al tempo di esecuzione (diverso per ogni calcolatore). Meglio calcolare il costo in base all'input. Tuttavia generalmente dati due input uguali il tempo di esecuzione è infatti diverso per vari fattori architettonici hardware.

TEMPO DI ESECUZIONE -> detta anche COMPLESSITA' TEMPORALE, misura il tempo di un algoritmo in base al numero di linee di codice eseguite.

COMPLESSITA' SPAZIALE -> Per l'efficienza di un algoritmo non ci interessa soltanto il tempo di esecuzione ma anche lo SPAZIO DI MEMORIA che tale algoritmo richiede. In particolare si calcola come il numero di chiamate ai metodi attive contemporaneamente.

ALBERO DELLA RICORSIONE -> è una struttura ad albero che serve a rappresentare la ricorsione: ogni radice rappresenta una chiamata mentre i figli sono le sottochiamate che a loro volta sono chiamate principali e così via. In questo caso, se si assegna ad ogni nodo il numero di righe di codice eseguito, per calcolare il costo dell'albero basta procedere dal basso verso l'alto a sommare per ogni nodo il numero delle righe di codice dei figli e si avrà così il numero totale di righe eseguite.

NOTAZIONI ASINTOTICHE -> servono ad esprimere la tendenza approssimata di un algoritmo all'infinito utilizzando un'altra funzione ed esprimendo la relazione esistente. Esistono tre tipi:

- O grande -> f è O(g) se cresce meno di g
  Si può dimostrare che una funzione cresce meno di g se calcolando il limite a più infinito del rapporto f/g questo esce 0
- Omega -> f è Omega(g) se cresce di più rispetto a g
  Si può dimostrare che una funzione cresce di più di g se calcolando il limite a più infinito del rapporto f/g questo esce +infinito
- Theta -> f è Theta(g) se cresce esattamente come g
  Si può dimostrare che una funzione cresce esattamente come g se calcolando il limite a più infinito del rapporto f/g questo esce 1

Nel caso di somme si considera sempre il contributo più alto per il calcolo delle notazioni

METODO DELL'ITERAZIONE -> detto anche dello srotolamento, è un metodo che può essere utilizzato quando si ha una relazione di ricorrenza ovvero nel caso di funzioni ricorsive per cui si conosce qual è la relazione fra un passo e l'altro. In questo caso è dunque possibile applicare delle maggiorazioni o minorazioni per ottenere la funzione asintotica. Inoltre, tramite questo metodo si ottengono le costanti delle notazioni.

PROBLEMA -> un problema è costituito da input e output ed è risolvibile quando, dati degli ingressi, si riescono ad ottenere delle uscite ovvero dei risultati dall'esecuzione del problema. La funzione che porta da input ad output è proprio un algoritmo per cui se tale algoritmo non esiste allora il problema non è risolvibile.

ANALISI PER CASI -> quando viene dato un algoritmo si hanno diverse casistiche:

Caso peggiore -> caso in cui vengono eseguite più operazioni ovvero il costo di esecuzione è il più alto

Caso migliore -> caso in cui vengono eseguite meno operazioni ovvero il costo di esecuzione è il minore

Caso medio -> è dato dal valore atteso ovvero una media pesata e cioè dalla sommatoria dei costi di esecuzione per le rispettive probabilità di verificarsi.

OCCUPAZIONE DI MEMORIA -> Lo spazio di memoria occupato nelle chiamate ricorsive non è proporzionale alla ricorsione stessa ma allo spazio occupato dallo stack. In pratica per calcolare il costo spaziale si può ricorrere a valutare la grandezza dell'input

COMPLESSITA' AMMORTIZZATA -> tale complessità fa riferimento al costo di un algoritmo diviso per le operazioni che compie. Permette dunque di svolgere un'analisi media dell'andamento di un algoritmo che, se osservato nelle sue singole operazioni, potrebbe invece non dare ottimi risultati.

TIPOLOGIE DI ALGORITMI -> si possono dare degli attributi particolari agli algoritmi che seguono determinate proprietà Ad esempio:

- Algoritmo greedy o goloso -> è un algoritmo che cerca una soluzione compiendo un passo alla volta e scegliendo ciascuna volta una soluzione che è quella più appetibile.
- Algoritmo esatto -> fornisce un risultato esatto e cioè senza arrotondamenti
- Algoritmo ottimo -> fornisce la soluzione migliore o fra le migliori.

SCHEDULE DI ATTIVITA' -> consiste nell'avere un array in cui associa un tempo di partenza ad ogni attività contenuta. Si parla di schedule ottimo

ALGORITMO RANDOMIZZATO -> è un algoritmo in cui i dati di input sono dati con casualità in modo da valutare buone prestazioni in media ed evitare dunque che pochi casi ma pericolosi facciano sembrare tale algoritmo meno efficiente di quanto non sia. Il quicksort è ad esmepio un algoritmo randomizzato in quanto l'elemento pivotale è scelto sempre in maniera casuale.

COMPLESSITA' INTRINSECA DI UN PROBLEMA

I problemi sono spesso caratterizzati da dei costi di esecuzione che esistono sempre, indipendentemente dall'algoritmo che si vuole utilizzare. Si può parlare di

Limite superiore o upper bound -> quantità di tempo e memoria che sono sufficienti per la risoluzione nel caso peggiore. Se l'upper bound definisce una complessità del problema O(g(n)) allora esiste almeno un algoritmo che risolve il problema con la stessa complessità ovvero crescono al massimo come g.

Limite inferiore o lower bound -> quantità di tempo e memoria che sempre nel caso peggiore sono necessari ma non sufficienti alla risoluzione del problema. Se il lower bound (detto anche complessità intrinseca del problema) definisce una complessità pari a Omega(g(n)) allora tutti gli algoritmi che danno soluzione al problema nel caso peggiore hanno la stessa complessità ovvero crescono almeno come g.

ALGORITMI OTTIMALI -> gli algoritmi ottimali sono tali che se la complessità intrinseca del problema è pari a Omega(g(n)) allora essi hanno complessità O(g(n)) nel caso peggiore.

Metodi elementari per determinare la complessità intrinseca

TECNICA DELL'AVVERSARIO -> tale tecnica prevede di ipotizzare per assurdo che la complessità intrinseca di un problema sia minore di quanto in realtà ovvero si cerca se esiste un algoritmo con migliore complessità. Quando si giunge ad avere un output errato all'input dato (in quanto il numero di passi dell'ipotesi non sono sufficienti) allora ho dimostrato che la complessità non può essere minore di quanto affermato inizialmente.

ALBERI DI DECISIONE -> sono rappresentazioni di meccanismi decisionali in cui ogni nodo è un punto di confronto e i due figli rappresentano i due possibili esiti al confronto. Quando arrivo ad una foglia ho concluso il processo e sono arrivato ad una conclusione. In particolare è possibile ad esempio rappresentare la ricerca binaria di una stringa in un elenco di parole ordinate.

NUMERO MINIMO DI CONFRONTI -> è il numero minimo di confronti che un algoritmo basato su confronti deve fare per risolvere un problema. Per rappresentare i confronti si usano gli alberi decisionali e le foglie devono essere in numero maggiore o uguale alle possibili risposte del problema. Nel caso dell'ordinamento il costo sarà Omega(n logn) mentre nel caso della ricerca in un vettore ordinato esso sarà Omega(logn).

MODELLI

MODELLO DI COSTO UNIFORME -> le operazioni hanno tutte lo stesso costo (ciò semplifica il calcolo dei costi delle operazioni)

MODELLO DELL'OPERAZIONE DOMINANTE -> in presenza dell'iterazione di alcune operazioni all'interno ad esempio di un ciclo, si assume che, per il modello di costo uniforme, le operazioni al di fuori del ciclo abbiano costo unitario e quindi trascurabile mentre quelle all'interno sono appunto dette dominanti. In questo caso si può esprimere l'asintoticità tramite la notazione Theta

MODELLO A COSTO LOGARITMICO -> in questo modello si assume che il costo di esecuzione è proporzionale alla dimensione (in bit) dell'input e cioè al logaritmo dello stesso.

TECNICHE DI PROGETTAZIONE

DIVIDI ET IMPERA -> è una tecnica intrinsecamente ricorsiva che prevede la divisione del carico di lavoro in sottoparti dove ciascuna viene risolta in maniera indipendente e alla fine i risultati vengono riuniti insieme. Se i dati vengono suddivisi in parti uguali si parla di algoritmi di tipo 1 altrimenti di tipo 2.
Nel caso di problemi banali, è spesso più efficiente utilizzare una risoluzione iterativa piuttosto che ricorsiva tramite il dividi et impera.

TEOREMA DELLE RICORRENZE -> Per i problemi risolti col metodo del dividi et impera con algoritmi di tipo 1, si può applicare il cosiddetto teorema delle ricorrenze al fine di calcolare la complessità esatta degli algoritmi.

Il teorema afferma che, data una relazione di ricorrenza e la complessità dell'algoritmo dividi e combina pari a n^d ,la complessità dell'algoritmo è

- Theta($n^d \log_c n$) se $a/c^d = 1$
- Theta(n^d) se $a/c^d < 1$
- Theta($n^{d \log_c n}$) se $a/c^d > 1$

dove

- a è il numero di suddivisioni in cui si applica l'algoritmo contemporaneamente (ad esempio nel caso della ricerca binaria a=1 poiché cerco in una sola metà alla volta mentre nel caso del merge sort a=2 poiché analizzo due metà contemporaneamente)
- n/c è la dimensione delle suddivisioni
- d rappresenta il costo in notazione asintotica delle operazioni di suddivisione e scelta di quale sezione analizzare. Nel caso in cui serva una operazione allora d=0 come nel caso

della ricerca binaria mentre se servono n operazioni come nel caso del merge sort allora d=1

Applicando tale teorema inoltre si può verificare se effettivamente conviene applicare il metodo dividi et impera e con quali parametri.

Algortimo effettivo:

1) $aT(n-k)+b*n^d$

$$\text{Se a} = 1 \rightarrow O(n^{d+1})$$

$$\text{Se a} \geq 2 \rightarrow O(a^n * n^d)$$

2) $aT\left(\frac{n}{c}\right)+b*n^d$

$$\text{Se } \frac{a}{c^d} < 1 \rightarrow O(n^d)$$

$$\text{Se } \frac{a}{c^d} = 1 \rightarrow O(n^d * \log_b n)$$

$$\text{Se } \frac{a}{c^d} > 1 \rightarrow O(n^{\log_b a} * c)$$

dove:

a numero di sottoproblemi

c fattore dei divisione

b costante

d potenza di n

PROGRAMMAZIONE DINAMICA -> essa viene utilizzata quando i problemi non sono indipendenti fra di loro. Infatti, si poggia su una tabella in cui risultati parziali sono salvati e possono poi essere utilizzati quando vengono richiamati da altre parti dipendenti da essi. Un algortimo di questo genere è quello di Fibonacci

ELIMINAZIONE DELLA RICORSIONE IN CODA -> nel caso di funzioni ricorsive dove l'ultima azione eseguita è proprio una funzione ricorsiva, si ha che il rispettivo algoritmo risolutivo in veste iterativa risulta occupare molto meno spazio in memoria

TECNICA DEI CREDITI -> tale tecnica serve per calcolare il costo ammortizzato di una operazione e funziona prevedendo che alcune operazioni depositino dei crediti mentre altre le prelevino per cui se assegnamo c crediti allora la complessità finale sarà Theta(n). Tale metodo prevede che i crediti assegnati a ciascuna operazioni siano sempre un'approssimazione non per difetto e dunque il costo finale sarà una stima asintotica superiore.

STRUTTURE DATI

DIZIONARIO -> è una struttura che raccoglie coppie chiave-valore. La funzione di ricerca cerca il valore associato ad una determinata chiave.

ALBERO -> è una struttura gerarchica organizzata in nodi padre e nodi figli. I nodi alla fine sono detto foglie. L'altezza dell'albero è dato dal valore del massimo livello + 1 ovvero dal numero di nodi presenti sul cammino più lungo che va dalla radice ad un nodo foglia.

Nel caso in cui non vi sia un limite ai possibili figli di un nodo, possiamo collegare i nodi a delle liste di puntatori ai nodi figli.

Anche un nodo può essere visto come un albero di altezza 0 e con una radice e una foglia.

La complessità spaziale dell'iterazione sarà costante in quanto non dipende dall'input mentre il costo temporale è costante nel caso migliore o pari a Theta(n) nel caso peggiore, ovvero nel caso in cui si debbano scandire tutti i nodi e risultano essere tutti a null.

In un albero, si possono analizzare i nodi in diversi modi:

- Visita infissa o in ordine -> esso funziona visitando prima il sottoalbero sinistro, la radice e infine il sottoalbero destro. Dato un albero ordinato, tale ricerca restituisce i nodi in ordine crescente.
- Visita anticipata -> esso funziona visitando prima la radice, il sottoalbero sinistro e infine quello destro
  Costo temporale: Theta(n)
  Costo spaziale: Theta(n)
- Visita posticipata -> esso funziona visitando prima i sottoalberi e poi la radice
  Costo temporale: Theta(n)
  Costo spaziale: Theta(n)
- Visita a livelli -> si parte dalla radice e si visitano tutti i nodi che appartengono allo stesso livello prima di scendere al successivo. Per implementarla posso utilizzare una coda in cui inserisco i nodi di un livello e li estraggo poi una alla volta per esaminarne i figli.
  Costo temporale: Theta(n)
  Costo spaziale: Theta(n)

ALBERO BINARIO -> alberi in cui i nodi hanno grado compreso fra 0 e 2 ovvero possono avere uno, due o nessun figlio.

Un albero binario si dice DEGENERE se ogni nodo interno ha un solo figlio

ALBERO BINARIO DI RICERCA -> detti anche ABR sono alberi binari che risultano essere ordinati e permettono l'implementazione della ricerca binaria in quanto gode delle seguenti proprietà:

- Il sottoalbero sinistro di un nodo contiene soltanto i nodi con chiavi minori della chiave del nodo
- Il sottoalbero destro di un nodo contiene soltanto i nodi con chiavi maggiori della chiave del nodo
- Il sottoalbero destro e il sottoalbero sinistro devono essere entrambi due alberi binari di ricerca.

Inoltre ha complessità temporale pari a O(n) in tutti i casi (ricerca, inserimento, cancellazione, spaziale) nel caso peggiore. Nel caso migliore addirittura in corrispondenza di ricerca, inserimento e cancellazione è pari a O(h).

Un ABR con duplicati è un albero che presenta la stessa definizione di un albero binario ma possono essere presenti degli elementi duplicati per cui le proprietà possono essere rimodellate come segue:

- Il sottoalbero sinistro di un nodo contiene soltanto i nodi con chiavi non maggiori della chiave del nodo
- Il sottoalbero destro di un nodo contiene soltanto i nodi con chiavi non minori della chiave del nodo
- Il sottoalbero destro e il sottoalbero sinistro devono essere entrambi due alberi binari di ricerca.

BILANCIAMENTO -> un albero binario è bilanciato se la differenza fra il numero di nodi dei due sottoalberi destro e sinistro è al massimo pari a 1 ovvero i due sottoalberi devono differire per massimo un nodo. Tale differenza si chiama col nome di fattore di bilanciamento del nodo radice dei due sottoalberi.

Gli alberi bilanciati prevedono un costo di ricerca pari a O(log n) e altezza media pari a O(log n). Nel caso migliore la ricerca è pari a Theta(1) mentre in quello peggiore (se l'albero è degenere) sarà Theta(n). La complessità spaziale è Theta(1) in quanto iterativo e ha dunque numero di operazioni costanti.
Sia l'inserimento che l'eliminazione in un ABR hanno lo stesso costo

Per il ribilanciamento si hanno quattro possibili casi in base alla posizione del sottoalbero che sbilancia:

- T sottoalbero sinistro del figlio sinistro -> SS -> rotazione sinistra-sinistra
- T sottoalbero destro del figlio destro -> DD -> rotazione destra-destra
- T sottoalbero desto del figlio sinistro -> SD -> rotazione sinistra-destra
- T sottoalbero sinistro del figlio sinistro -> DS -> rotazione destra-sinistra

ALBERO AVL -> è un albero binario di ricerca che risulta però essere sempre bilanciato, ovvero il grado di sbilanciamento di ciascun nodo è al massimo 0 o 1 (mai 2).

ALBERO COMPLETO -> un albero si dice completo se tutti i suoi livelli sono pieni e tutti nodi tranne le foglie hanno due figli. Si dice quasi completo se è completo fatta eccezione per l'ultimo livello

CODA -> è una struttura di tipo FIFO ovvero first-in-first-out in quanto il primo elemento ad essere stato inserito è il primo ad essere eliminato.

CODEDE CON PRIORITA' -> sono strutture dove ogni elemento ha una propria priorità e il primo elemento ad essere prelevato è quello a maggiore priorità.

La priorità può essere di due tipi:

- Priorità fissate -> la priorità può assumere un valore finito in un intervallo dato.
  Essa può essere implementata come un array di code FIFO per cui in ogni cella vi è una coda di elementi con la stessa priorità. Per inserire un elemento con priorità k tale

elemento viene ad essere aggiunto nella cella di indice k dell'array. Per la rimozione invece si procede ad analizzare la coda che si trova all'indice dell'array che si vuole svuotare

- Priorità infinite -> la priorità può assumere un qualsiasi valore.
  In questo caso la realizzazione può avvenire tramite un array ordinato non circolare, una lista ordinata o un albero binario di ricerca di tipo AVL. Tuttavia, la migliore implementazione si ha con l'heap binario

L'ordinamento con code di priorità sfruttano la proprietà di una coda per ordinare gli elementi e spostarli poi in una struttura dati come un array. Esso prevede due fasi:

- inserire gli elementi nella coda (costo n inserimenti * costo inserimento)
- togliere gli elementi nella coda e ricopiarli nel vettore (costo n inserimenti * costo rimozione)

In base al tipo di coda usata si avrà:

- Heap -> il costo di inserimento è log(n) per entrambe le fasi . Dunque costo finale Theta(n*log(n))
- Lista ordinata -> il costo di inserimento è n nella prima fase . Dunque costo finale Theta(n*log(n))
- AVL -> il costo di inserimento è log(n) per entrambe le fasi. Dunque costo finale Theta(n*log(n))
- Coda a priorità fissata -> il costo di inserimento è 1 per entrambe le fasi. Dunque costo finale Theta(n)

La complessità spaziale sarà invece pari a Theta(n) perché si vanno ad occupare n celle della struttura su cui ci si appoggia

HEAP BINARIO -> l'heap binario è una formulazione più elastica dell'albero binario in quanto prevede che la relazione che esiste una sola regola quale quella che lega un padre e un figlio e che essa debba essere la stessa utilizzata in tutto l'albero fra padre e figlio; tale regola è appunto la priorità dell'heap. Dunque, diversamente dall'albero binario, non vi è un ordinamento specifico in tutto l'albero ma solo nei sottoalberi. Tale albero è anche esso caratterizzato da un bilanciamento costante come gli AVL e dunque non vi sono nodi che hanno grado di bilanciamento maggiore di 1.
Un heap decrescente (detto anche MAX HEAP) ha che il padre è sempre maggiore dei figli diretti e dunque anche dei nodi del sottoalbero. Tuttavia, non si può fare lo stesso ragionamento per quanto riguarda i livelli perché o le profondità in quanto l'unica regola valida è che il valore maggiore di un sottoalbero è sempre quello della radice.
La cancellazione di un elemento richiede un tempo Theta(log n) in quanto bisogna visitare tutto l'albero in altezza.

Per trasformare l'albero in array posso procedere associando ad ogni nodo l'indice che avrebbe in una visita per livelli. Così facendo, se un padre si trova in posizione x, allora avrebbe suo figlio sinistro in posizione 2z e suo figlio destro in posizione 2x+1.

UNION FIND -> sono strutture dati che si basano sul concetto di partizionamento di un insieme. Le operazioni principali sono

- Union -> unisce due insiemi e facendo in modo che tutti gli elementi puntino ad un solo nome

- Find -> è un'operazione di costo costante che restituisce il nome dell'insieme in cui l'elemento cercato si trova.
- Makeset -> inserisce un insieme nella struttura dati

Prese le componenti massimali di una foresta, esse sono una partizione dell'insieme dei nodi. Per verificare se si forma un ciclo basta invocare la find su due elementi e se essi appartengono a due sottoinsiemi diversi allora non vi sono cicli.

Tale struttura dati può essere implementata sia tramite il tipo QuickFind (dove il costo della find è costante) o di tipo QuickUnion (dove il costo della union è costante).

In entrambi i casi la radice contiene il nome dell'insieme mentre i figli sono gli elementi contenuti. In particolare si avrà:

- QUICKUNION -> prevede di unire due sottoinsiemi creando un nuovo nodo col nome del primo insieme e attaccandogli come figli i due insiemi da unire. In questo caso il costo è dunque costante perché bisogna solo aggiungere un nodo e due puntatori agli insiemi da unire.
- QUICKFIND -> prevede di unire due sottoinsiemi attaccando i figli del secondo alla radice del primo. In questo caso il costo è dunque proporzionale al numero di elementi da aggiungere al primo insieme e avrà costo quadratico ovvero Theta(n^2). Una possibile implementazione più efficiente consiste nel verificare quali dei due insiemi ha cardinalità (ovvero numero di elementi) maggiore; se è il primo si procede come sopra, mentre se è il secondo si procede allo stesso modo ma ricordandosi alla fine di sostituire il nome della radice con quello del primo insieme ottenendo così un costo che è O(n * log(n)).

GRAFI -> sono strutture in cui è data importanza alla relazione fra gli oggetti e agli archi che li collegano. Non esiste dunque una gerarchia come negli alberi.
Le relazioni possono essere bidirezionali e allora il grafo si dirà non orientato o monodirezionali per cui il grafo si dirà ORIENTATO per cui si può definire grafo una coppia di insiemi: uno è l'insieme dei nodi e l'altro è l'insieme delle relazioni. Nel caso della bidirezionalità, nell'insieme delle relazioni bisogna inserire non solo la coppia <a, b> ma anche la coppia <b, a>.

In base al numero di archi entranti o uscenti si possono anche definire il grado di entrata e quello di uscita di un nodo. Essi si definiscono come:

GRADO DI ENTRATA -> numero di archi entranti nel nodo. Ad esso si assimila il GRADO DI UN NODO

GRADO DI USCITA -> numero di archi uscenti dal nodo

ARCO INCIDENTE -> E' un arco che collega due nodi. Un arco che entra in b ed esce da a si definisce come <a, b>.

CAMMINO -> è una sequenza di nodi che, tramite l'orientamento dei propri archi, portano da un nodo a ad un nodo b.

RAGGIUNGIBILITA' TRA NODI -> due nodi si dicono raggiungibili se esiste un cammino (e dunque dei nodi collegati eventualmente con la giusta orientazione) che collega i nodi.

CICLO -> è un cammino che parte e arriva allo stesso nodo. Nel caso di grafo orientato bisogna rispettare l'orientamento mentre nel caso di grafi non orientati no. Inoltre, un cammino può essere anche con cicli quando non per forza parte e termina dallo stesso nodo ma comunque compie un

ciclo nel suo passaggio. Gli algoritmi che cercano i cammini senza cicli sono molto costosi (solitamente di costo esponenziale).

La verifica della presenza o meno di cicli all'interno dei grafi dipende se essi sono orientati o meno.

Per i grafi orientati vale la seguente PROPRIETA':
Se tutti i nodi hanno grado di entrata maggiore di zero allora il grafo è ciclico.

Per i grafi non orientati vale la seguente PROPRIETA':
Se esiste un nodo con grado di entrata zero e lo elimino insieme ai suoi archi, ottengo un nuovo grafico che se risulta essere ciclico comporta che anche il grafo non orientato di partenza era ciclico.

L'algoritmo di verifica sarà dunque il seguente:

- calcolo il grado di entrata di tutti i nodi
- se tutti i nodi hanno grado di entrata maggiore di zero allora vi è un ciclo
- altrimenti, elimino un nodo di grado zero
- se non rimangono più nodi posso concludere che non ci sono cicli, altrimenti itero il procedimento

La complessità temporale di tale algoritmo sarà Theta(n^2) mentre quello spaziale pari a Theta(n).

I grafi possono essere rappresentati attraverso diverse strutture:

- Grafo a lista d'archi -> si utilizza una struttura come una LinkedList o un ArrayList in cui ogni elemento è un arco ovvero una coppia di nodi. Tuttavia, tale metodo non è altamente efficiente in quanto è utile solo se voglio scorrere tutte le coppie.
- Matrice di adiacenza -> in tale matrice sono contrassegnate quelle caselle che sono individuate da due nodi collegati mentre i nodi che non presentano archi hanno la corrispettiva cella vuota. Dunque due archi sono collegati solo se la cella che individuano è contrassegnata. Il costo della rappresentazione è quadratico ovvero Theta(n^2).
  Per scorrere tutti gli archi ho sempre un costo Theta(n^2), per leggere gli archi uscenti da un nodo mi basta un costo Theta(n) e per vedere se esiste un arco fra due nodi ho un costo costante ovvero Theta(1).
- Lista di adiacenza -> tale meccanismo prevede di avere un array dove ciascuna cella ha come indice il valore del rispettivo nodo e contiene al suo interno una lista di tutti quei nodi che sono collegati in uscita alla cella a cui fa riferimento.
  Esso ha un costo di rappresentazione dato da n che è la lunghezza della struttura e m che è il numero di archi. Il costo sarà dunque Theta(n + m). Per scorrere tutti gli archi il costo sarà lo stesso.
- Matrice di incidenza -> rispetto alla matrice di adiacenza, non si rappresentano nodi-nodi ma nodi-archi e una cella viene segnata se esiste un arco di quel tipo per quel nodo.
- Lista di incidenza -> rispetto alla lista di adiacenza, le celle dell'array contengono dei nodi dove ciascuno dei quali fa riferimento ad una lista di archi.

I grafi, rispetto agli alberi, non presentano un punto di partenza specifico ma il nodo può essere scelto casualmente. Esistono due tipi di visite:

- Depth-first -> detta anche in profondità o a scandaglio, è un algoritmo simile alla visita anticipata sugli alberi ovvero, scelto il nodo di partenza, procedo a visitare quelli adiacenti. Per evitare un loop devo inserire un controllo per evitare di passare su un nodo già visitato.
  Complessità temporale (caso peggiore) : Nel caso della matrice è Theta(n^2) mentre nel caso della lista sarà dato dagli archi e dunque sarà Theta(m)
  Complessità temporale (caso migliore) : sarà pari a Theta(1) nel caso in cui non vi sono archi uscenti
  Complessità spaziale (caso peggiore) : Theta(n)
  Complessità spaziale (caso migliore) : Theta(1)
- Breadth-first -> detto anche in ampiezza o a ventaglio, visita i nodi adiacenti ma sullo stesso livello.
  Complessità temporale: Nel caso della matrice è Theta(n^2) mentre nel caso della lista sarà dato dagli archi e dunque sarà Theta(m)
  Complessità spaziale : Theta(n)

GRAFO CONNESSO -> un grafo non orientato è connesso quando posso definire un cammino per unire ciascuna coppia di nodi presenti. Per verificare la connessione si visitano tutti i nodi e verifico che almeno una volta vengano toccati tutti i nodi. Inoltre, un grafo può essere suddiviso in componenti ovvero sottografi che possono a loro volta essere connessi o meno.

COMPONENTE FORTEMENTE CONNESSA -> detta anche componente connessa massimale, è la componente di un grafo (ovvero un suo sottografo) connesso e tale che non esiste una componente più grande che risulta essere connessa. Dunque tutti i nodi sono connessi fra di loro.

GRAFO DEBOLMENTE CONNESSO -> un grafo è debolmente connesso se eliminando le orientazioni degli archi, esso continua ad essere connesso.

Inoltre, da un grafo connesso aciclico posso considerarlo come un albero. Per costruirlo devo creare una gerarchia “eleggendo” un nodo a radice e in base al nodo scelto ottengo alberi diversi.

GRAFI PESATI SUGLI ARCHI -> essi sono grafi dove gli archi presentano un penso ovvero hanno un loro eventuale costo che non è lo stesso per tutti gli archi e ciò influenza particolarmente l’efficienza dei cammini.
Grafi di questo tipo non sono più delle coppie di insiemi ma delle triple dove uno è l’insieme dei nodi, uno degli archi e infine la terza componente è una funzione che lega gli archi ai rispettivi pesi.
Per rappresentare un grafo del genere posso utilizzare le stesse strutture dei grafi normali apportando delle piccole modifiche:

- Matrice di adiacenza -> invece di contrassegnare o meno una casella che rappresenta un arco fra due nodi, posso inserirvi il peso del relativo arco.
- Lista di adiacenza -> ogni lista non contiene i nodi per cui esistono degli archi ma delle coppie di nodo e peso del relativo arco

Dato un grafo pesato si può parlare di COSTO DI UN CAMMINO come la somma dei costi di ciascun arco lungo un cammino percorso. Esisterà dunque un cammino di costo minimo che è proprio quel cammino che minimizza il costo dei percorsi esistenti ed il suo costo è detto distanza minima.

La distanza minima si può definire per tutti quei grafi che non hanno cicli di valore negativo e si chiama distanza di ordine n quando la distanza minima fra due nodi ha massimo n nodi intermedi nel cammino.

Un cammino si dice n-VINCOLATO se contiene nodi intermedi con indice minore di n.

CENTRALITA' DI UN NODO -> è il numero di cammini minimi in cui tale nodo compare come intermedio

ALBERO RICOPRENTE -> è un albero composto da tutti i vertici del grafo e da un sottoinsieme degli archi di partenza che servono a collegare tali vertici. Nel caso di un grafo non connesso, si può parlare di foresta di alberi ricoprenti.
Un albero ricoprente è detto di ALBERO RICOPRENTE MINIMO o anche di costo minimo se non esiste un altro albero ricoprente di costo minore.
Esistono diversi algoritmi che permettono di calcolare un albero ricoprente di costo minimo. Essi sono:

- ALGORITMO DI PRIM -> algoritmo per grafi non orientati e con pesi positivi. E' un tipo di algoritmo detto greedy o goloso perché compie piccoli passi e a ciascuno va a scegliere la soluzione più appetibile ovvero quella che in quel momento è la scelta più efficiente. esso non restituisce la migliore soluzione globale ma un albero ricoprente di costo minimo.
  Esso procede secondo i seguenti passi:
  - scelgo un nodo di partenza e lo aggiungo all'albero di output
  - scelgo uno dei nodi adiacenti secondo l'arco di costo minimo e l'aggiungo all'albero di output
  - itero il secondo processo fino a quando la cardinalità dell'albero di output non è la stessa dell'albero di partenza ovvero quando ho visitato e incluso tutti i nodi

  Il costo di tale algoritmo è pari a Theta(n^2) nel caso di una matrice di adiacenza e ad Theta(m) nel caso di una lista di adiacenza ma se uso l'heap posso ridurre tale cose a Theta(n * log(n))

- ALGORITMO DI KRUSKAL -> algoritmo per grafi non orientati e con pesi positivi. E' un tipo di algoritmo ottimo che può essere usato se esistono almeno due o più nodi. Esso procede come segue:
  - si scrivono in ordine crescente gli archi in base al proprio peso. L'ordine di archi con lo stesso peso è ininfluente
  - si sceglie un arco per volta dall'elenco sempre in maniera crescente e soprattutto evitando quegli archi che creano cicli
  - quando tutti i nodi sono stati collegati allora si è calcolato l'albero ricoprente minimo
- ALGORITMO DI DIJSTRA -> algoritmo per grafi orientati e non, ciclici e non, con pesi positivi. E' un algoritmo che similmente a quello di Prim calcola il costo del percorso nodo per nodo però risulta essere ancora migliore in quanto ad ogni iterazione aggiorna ogni nodo col valore che il percorso avrebbe se si andasse in quella direzione, permettendo dunque di scegliere sempre la scelta migliore. Esso segue i seguenti passaggi:
  - prende un nodo di partenza il cui valore sarà zero e aggiorna tutti i restanti nodi con i costi dei rami di collegamento
  - sceglie il ramo che porta ad un nodo di costo minore

        - riaggiorna tutti i nodi con le somme degli archi entranti possibili del percorso e ripete l'operazione
    - ALGORITMO DI FLOYD -> è un algoritmo per grafi pesati e orientati. Esso ha complessità cubica e procede fino a quando non ha raggiunto la distanza n-vincolata. Esso funziona basandosi su una matrice che contiene in ciascuna cella, individuata da due nodi, il valore del cammino che li collega e che può avere per nodi intermedi solo quelli di indice minore al valore del passo ovvero il percorso deve essere passo-vincolato. Dunque esso itera il processo per il numero del grafo verificando se ad ogni passo la distanza fra il nodo i e quello j è minore della somma fra la distanza fra i e il nodo di indice pari al passo e j e il nodo di indice pari al passo; in tal caso vuol dire che passare per il nodo di indice pari al passo è migliore e dunque si sostituisce tale risultato. Dopo tutte le iterazioni basta leggere la matrice per ottenere la distanza minima fra tutti i nodi dunque risulta restituire sistematicamente tutte le distanze di un grafo.

Lo spazio delle possibili soluzioni date da questi algoritmi si chiama FORESTA RICOPRENTE

PROBLEMA DELLA CRICCA SUI GRAFI -> tale problema riguarda la ricerca di sottografi completi dette cricche (ovvero grafi non orientati che presentano ciascun nodo collegato a tutti gli altri). Il più semplice algoritmo che risolve tale problema è quello di forza bruta ma impiega un tempo esponenziale per esaminare tutte le possibili combinazioni.

PROBLEMA DELLA COPERTURA CON NODI -> tale problema fa riferimento ad un insieme di copertura detto tale in quanto comprende tutti i vertici che hanno un arco di incidenza in un altro nodo. Trovare il minimo numero di archi per collegare tutti i nodi e almeno alla volta è lo scopo

PROBLEMA SUBSET SUM -> tale problema prevede di calcolare, dato un insieme, quali sono gli elementi da sommare per ottenere un numero obiettivo

TABELLA AD ACCESSO DIRETTO

Le tabelle ad accesso diretto sono dizionari memorizzati in degli array. Tramite alla chiave k si accede all'elemento v[k]. Ciò può causare un problema di efficienza nel caso di chiavi composte da molti caratteri in quanto ciò richiede un array molto grande.

FATTORE DI CARICO -> esso serve a misurare quanto è piena una tabella ad accesso diretto. Mi misura come il rapporto fra il numero di celle occupate e quelle totali. Grazie a questo parametro ci si può rendere conto in percentuale quanta memoria si sta sprecando.

TABELLA HASH

Le tabelle hash, anziché usare l'accesso diretto che richiede un grande spreco di memoria per chiavi molto lunghe, utilizza il meccanismo dell'hashing per cui, se la funzione di hashing è h(k), tramite la chiave k si accede all'elemento in posizione v[h(k)] ovvero non vi è corrispondenza univoca fra gli indici dell'array e le chiavi ma fra il risultato dell'hashing e gli indici degli array. Ciò permette di semplificare notevolmente chiavi molto lunghe.

HASHING -> consiste nell'applicare una funzione ad una chiave in ingresso ed ad utilizzare il risultato come indice per accedere al relativo dato nella tabella hash.

COLLISIONI -> diversi input possono restituire lo stesso hash e ciò causa una collisione nel momento dell'inserimento nella tabella in quanto si andrebbero a sovrascrivere. Vi sono due modi per risolvere le collisioni:

- liste di collisione -> ogni elemento della tabella punta ad una lista di oggetti che hanno lo stesso hash
- indirizzamento aperto -> se la cella rispettiva è occupata, ne vado a ricercare un'altra. Ciò si può fare sia con l'ispezione lineare che attraversa una per una le celle adiacenti fino ad individuarne una libera (che causa il fenomeno di agglomerazione cioè vi sono lunghi gruppi di celle consecutive occupate che rallentano la scansione) oppure tramite l'hashing doppio che consiste nel calcolare una nuova posizione come la somma fra la prima posizione ottenuta con la funzione di hash di partenza e il risultato di una seconda funzione hash diversa dalla prima applicata alla prima posizione ottenuta. Ciò permette di ridurre notevolmente

PROPRIETA' DI UNIFORMITA' SEMPLICE -> si ha quando una funzione hash è in grado di "distribuire" in modo uniforme gli oggetti in una tabella riducendo le collisioni.
In particolare, una funzione gode di tale proprietà se per ogni cella la probabilità che essa sia occupata è pari al reciproco della dimensione della tabella ovvero se ogni cella ha la stessa probabilità di essere occupata.

FUNZIONE HASH PERFETTA -> è una funzione iniettiva ovvero ad ogni input corrisponde un output diverso

HASHING ESTENDIBILE -> è una tecnica che prevede di travasare un array di una tabella hash in uno più grande quando quello è quasi pieno o in uno più piccolo se vi è troppo spreco di memoria. Tuttavia non conviene fare questa operazione per ogni elemento aggiunto o eliminato ma affidandosi ad una costante di incremento/decremento ciascuna volta.

PROPRIETA' INVARIANTE NELL'ESECUZIONE DELL'ALGORITMO -> proprietà che permette sempre di riuscire ad effettuare il travaso dell'array

ALGORITMI DI ORDINAMENTO

HEAPSORT IN PLACE -> dato un max heap (heap decrescente) esso procede iterando questi due passaggi:

- scambia la radice (che è il massimo valore) con il minimo dell'albero (che sarà una foglia) ed elimina tale foglia, ponendo tale valore nell'array da sinistra verso destra
- ristabilisce l'ordine

quando rimane un solo nodo, il suo valore viene messo in posizione zero nell'array e l'algoritmo termina.

COSTO: O(n * log(n)) ma risulta meno efficiente di algoritmi con prestazioni simili

COUNTING SORT -> tale algoritmo viene utilizzato quando si hanno degli elementi in un range determinato che si possono anche ripetere. Il procedimento è il seguente:

- si prende un array di supporto che ha tante celle quanto è il range dei possibili valori e si incrementa ciascuna cella in modo che faccia da contatore delle presenze di ciascun numero
  es. se il 2 compare 3 volte, la cella in posizione 2 conterrà il numero 3
- si calcolano le posizioni corrette dell'array ordinato in cui si andranno a posizionare i valori sommando la cella sinistra e quella destra e posizionando il risultato in quella destra
  es. se la cella 2 contiene 3 e la cella 3 contiene 1, la cella 3 conterrà 3+1=4

- procedo a scannerizzare nuovamente l'array di partenza e andando a posizionare il numero nella cella indicata dall'array di supporto per poi decrementare la cella che fa da indice
- ho ottenuto il nuovo array ordinato

COSTO: Theta(2^c + n) mentre nel caso spaziale è Theta(2^c * log(n)) perché il numero di cifre rappresentabili lo esprimo tramite una potenza del 2

RADIX SORT -> esso si basa sul confronto di cifra per cifra dei numeri e può essere utilizzato solo appunto con cifre. Il concetto è quello di dare priorità ad una cifra alla volta. Il procedimento è il seguente

- confronto le unità e ordino in base ad esse
- confronto le decine e ordino in base ad esse
  ….
- ottengo l'array ordinato

COSTO: tale algoritmo dipende dal numero di cifre che compone il numero più grande e che influenza quante volte viene ripetuto l'ordinamento. Dato un numero di cifre c, l'ordinamento temporale richiede Theta(c * n) mentre quello spaziale è meno efficiente perché richiede Theta(n + c*n)

QUICKSORT -> è un algoritmo basato sui confronti in quanto viene preso un elemento detto pivot e si confrontano tutti gli altri con esso. Il procedimento itera i seguenti punti:

- scelgo il pivot
- confronto gli elementi a partire dal primo: se sono minori vengono messi a sinistra del pivot, se sono maggiori a destra
- ottengo che ora il pivot si trova nella posizione corretta

COSTO: il caso peggiore si ha quando il pivot scelto è l'elemento maggiore o minore e in tale caso il costo dell'algoritmo è quadratico ovvero Theta(n^2)

Nel caso medio il costo si abbassa invece a pseudolineare ovvero O(n*log(n)).

La complessità spaziale è invece Theta(n) nel caso della presenza di ricorsione in coda mentre eliminandola si ottiene una complessità pari a Theta(log(n))

MERGE SORT -> è un algoritmo che sfrutta la tecnica del divide et impera. Esso procede in questo modo:

- divide l'array in due metà
- itera la divisione in due fino a raggiungere array formati da una singola cella (e dunque già ordinati)
- ordina le metà ottenute a partire dagli array formati da una singola cella e fonde gli array ordinati tramite dei confronti

L'algoritmo ha costo pseudolineare ovvero Theta(n*log(n)) sia nel caso medio che pessimo

SELECTION SORT -> esso funziona dividendo in due parti l'array, dove quella più a sinistra sarà quella ordinata e quella a destra da ordinare. Ad ogni iterazione si seleziona dalla parte non ancora ordinata il minimo che viene posto in coda alla parte ordinata. Alla fine si otterrà l'array ordinato.

L'algoritmo ha costo ottimo e peggiore sempre pari a Theta(n^2)

INSECTION SORT -> similmente al selection sort, divide l'array in due parti. Tuttavia, anziché prelevare il minimo della parte ordinata e inserirlo in coda, preleva un elemento a caso che viene inserito nella posizione corretta nella parte sinistra dell'array. Il caso ottimo si ha quando la sequenza è già ordinata per

cui il costo diventa lineare mentre nel caso in cui sia ordinato al contrario il costo di tale algoritmo diventa quadratico e cioè Theta(n^2).

COMPLESSITA' NEGLI ESERCIZI

- Se è un albero binario non bilanciato e mi richiede una condizione che deve essere sempre vera (o falsa) allora il caso migliore può ridursi a costante perché posso avere un ramo che presenta un nodo che non verifica (o verifica) la condizione.
- Se è un albero binario bilanciato devo tenere conto che anche nel caso migliore, se la richiesta è di analizzare ad esempio le foglie e trovarne almeno una che verifica o non verifica una condizione, devo comunque attraversare almeno una volta in lunghezza l'albero
- Se è un albero binario di ricerca (ABR) devo considerare la caratteristica dell'ordinamento per cui se mi richiede di determinare i nodi in un range o di ricercare un valore, nel caso migliore mi posso spostare in una metà alla volta dell'albero e quindi se non è bilanciato il caso migliore si ottiene con costo costante
- Se mi viene richiesta di valutare l'altezza, nel caso di un albero bilanciato esso sarà logaritmico altrimenti nel caso dell'albero degenere sarà lineare
- Se mi chiede che almeno una volta debba essere verificata una condizione, il caso migliore si ha quando verifico subito con la radice o i suoi figli mentre nel caso peggiore devo verificare tutti i nodi
- Quando si scrivono i metodi, è importante stare attenti a quando si devono verificare le foglie perché potrebbe non aggiornarsi un contatore. Dunque se devo contare le foglie conviene fare una return di 1 e sommare le funzioni ricorsive.

STRINGHE

DISTANZA DI LEVENSHTEIN -> detta anche distanza di edit, essa misura il numero di operazioni elementari che, date due stringhe, devono essere eseguite affinché siano uguali. Per operazioni elementari si intende:

- cancellazione di un carattere
- aggiunta di un carattere
- sostituzione di un carattere
- bonus: inversione di due caratteri consecutivi

Le operazioni eseguite non riguardano mai posizioni precedenti a quelle già elaborate

COSTO DI UNA SEQUENZA -> è il numero di operazioni elementari (vedi sopra) necessarie per realizzare una stringa.

BACKTRACKING

Tale tecnica viene utilizzata quando la risoluzione iterativa dei problemi risulta avere un costo esponenziale.

Un generico algoritmo di backtracking si compone di due parti:

- GENERATORE -> genera tutte le possibili soluzioni di un problema. Esso ha un approccio costruttivo ovvero costruisce pezzo per pezzo le possibili soluzioni

- CONTROLLORE -> controlla se la soluzione ottenuta è corretta. La complessità di implementazione di un controllore è Theta(n^k) ma si può ridurre tale complessità evitando di controllare determinate soluzioni.

In generale l'algoritmo ha la seguente struttura:

```
if(PrimaSoluzione)
1-      if (VerificaSoluzione)
            if(VerificaVincoli)
                andiamo al prossimo livello
            else
                ritorna al precedente livello
        ritorna in un livello
        if (ProssimaSoluzione)
            vai al passo 1
        else
            ritorna al precedente livello
```