Algoritmi & Strutture Dati

Algoritmi di ordinamento

	Caso Migliore	Caso Medio	Caso Peggiore	In Loco	Stabile
Selection Sort	$\theta(n^2)$			SI	SI
Insertion Sort	$\theta(n)$	$\theta(n^2)$	$\theta(n^2)$	SI	SI
Merge Sort	$\theta(n \log n)$			NO	SI
Heap Sort	$\theta(n \log n)$			SI	NO
Quick Sort	$\theta(n \log n)$	$\theta(n \log n)$	$\theta(n^2)$	SI	NO

Notazione Asintotica

La notazione asintotica classifica il comportamento di una funzione per grandi valori di *n*.

Notazione O-grande

$O(g(n))$ rappresenta l'insieme delle funzioni **limitate superiormente** da $g(n)$

$$f(n) \in O(g(n)) \leftrightarrow esistono\ due\ costanti\ positive\ c\ ed\ n_0\ tali\ che\ per\ ogni\ n \geq n_0\ si\ verifica$$

$$0 \leq f(n) \leq c * g(n)$$

Proprietà transitiva

$$f(n) \in O(h(n))\ per\ qualche\ h(n) \in O(g(n))$$

Regola dei fattori costanti positivi

$$f(n) = d * h(n)\ per\ qualche\ h(n) \in O(g(n))\ e\ d > 0$$

Regola della somma

$$f(n) = h(n) + k(n)\ con\ h(n)\ e\ k(n) \in O(g(n))$$

Notazione Ω

$\Omega(g(n))$ rappresenta l'insieme delle funzioni **limitate inferiormente** da $g(n)$

$$f(n) \in \Omega(g(n)) \leftrightarrow esistono\ due\ costanti\ positive\ c\ e\ n_0\ tali\ che\ per\ ogni\ n \geq n_0\ si\ ha$$

$$0 \leq c * g(n) \leq f(n)$$

Vale la proprietà **riflessiva** $g(n) \in \Omega(g(n))$

Proprietà transitiva

$$f(n) \in \Omega(h(n))\ per\ qualche\ h(n) \in \Omega(g(n))$$

Regola dei fattori costanti positivi

$$f(n) = d * h(n)\ per\ qualche\ h(n) \in \Omega\big(g(n)\big)\ e\ d > 0$$

Regola della somma

$$f(n) = h(n) + k(n)\ con\ h(n)\ e\ k(n) \in \Omega(g(n))$$

Notazione θ

θ(g(n)) rappresenta l'insieme delle funzioni **limitate inferiormente e superiormente** da $g(n)$

$$f(n) \in \theta\big(g(n)\big) \leftrightarrow esistono\ tre\ costanti\ positive\ c_1, c_2\ ed\ n_0\ tali\ che\ per\ ogni\ n \geq n_0\ si\ verifica$$

$$0 \leq c_1 * g(n) \leq f(n) \leq c_2 * g(n)$$

Vale la proprietà **riflessiva** $g(n) \in \theta(g(n))$

Proprietà transitiva

$$f(n) \in \theta\big(h(n)\big)\ per\ qualche\ h(n) \in \theta(g(n))$$

Regola dei fattori costanti positivi

$$f(n) = d * h(n)\ per\ qualche\ h(n) \in \theta\big(g(n)\big)\ e\ d > 0$$

Regola della somma

$$f(n) = h(n) + k(n)\ con\ h(n)\ e\ k(n) \in \theta(g(n))$$

Proprietà simmetrica

$$f(n) \in \theta\big(g(n)\big) \leftrightarrow g(n) \in \theta(f(n))$$

Gerarchia

(dal più piccolo al più grande)

$$O(1) \longrightarrow O(\log n) \longrightarrow O(n) \longrightarrow O(n \log n) \longrightarrow O(n^2) \longrightarrow O(n^3) \longrightarrow \cdots \longrightarrow O(2^n)$$

$$\Omega(1) \longrightarrow \Omega(\log n) \longrightarrow \Omega(n) \longrightarrow \Omega(n \log n) \longrightarrow \Omega(n^2) \longrightarrow \Omega(n^3) \longrightarrow \cdots \longrightarrow \Omega(2^n)$$

$$\theta(1) \longrightarrow \theta(\log n) \longrightarrow \theta(n) \longrightarrow \theta(n \log n) \longrightarrow \theta(n^2) \longrightarrow \theta(n^3) \longrightarrow \cdots \longrightarrow \theta(2^n)$$

Alberi radicati

Un albero è un insieme è un insieme di nodi su cui è definita una relazione binaria, ad esempio " x è figlio di y" oppure "y è genitore di x".

Caratteristiche

- Ogni nodo ha un solo genitore, tranne la radice che non ha genitori
- Esiste un **cammino** diretto da ogni nodo alla radice
- Il numero di figli di un nodo è detto **grado**
- **Alberi binari** → Ogni nodo può avere solo un figlio destro o sinistro
- **Alberi di grado arbitrario** → Ogni nodo può avere quanti figli vuole

Alberi binari
Ogni nodo può avere solo un figlio destro o sinistro.

Un albero binario è **completo** se a ogni livello presenta tutti i nodi possibili.

Gli alberi possono essere rappresentati tramite **oggetti e riferimenti**, ovvero ogni nodo è un oggetto con i campi: parent, left, right, info.

Il campo root è un riferimento al nodo radice.

Alberi di grado arbitrario
Ogni nodo può avere quanti figli vuole.

Per rappresentarlo tramite oggetti e riferimenti si usa la struttura "**figlio sinistro, fratello destro**", dove ogni nodo ha i campi parent, left, right e info. I campi left e right rappresentano rispettivamente il figlio sinistro e il fratello destro.

Visite di alberi
Un albero può essere visitato ricorsivamente con diversi metodi:

- Visita in **preordine**
- Visita in **postordine**
- Visita **simmetrica**

Visita in preordine
Dopo aver processato un nodo, si processano i suoi figli.

Le operazioni sui nodi vengono effettuate **top-down**.

Visita in postordine
Prima vengono processati i figli, e poi il nodo.

Le operazioni sui nodi vengono effettuate **bottom-up**.

Visita simmetrica
È possibile effettuarla sono in caso di **albero binario**.

Si processa prima il figlio sinistro, poi il nodo e poi il figlio destro.

La visita simmetrica è particolarmente utile in un **albero binario di ricerca**, dove effettuandola posso produrre un ordinamento crescente dato che il figlio sinistro è sempre minore del figlio destro.

Tutte le visite hanno **costo** $\boldsymbol{\theta(n)}$.

Heap
Un heap è una struttura dati (array) che può rappresentare una **coda di priorità**.

Una coda di priorità è una **collezione di elementi** in cui ad ogni elemento è associato un valore di priorità, ed ogni valore di priorità definisce un ordinamento.

Nell'heap i valori sono in rapporto con la loro posizione nell'array.

Un heap può essere visto come un **albero binario quasi completo**, ovvero un albero binario dove l'ultimo livello può essere incompleto nella sua parte destra.
Quindi l'heap è un array che **codifica livello per livello** questo tipo di albero.

Dato un nodo *i*:

- I nodi figli si trovato in posizione *2i+1* e *2i+2*
- Il nodo genitore si trova in posizione $(i-1)/2$

In un **max-heap** l'elemento memorizzato nel nodo *i* ha valore maggiore o uguale degli elementi memorizzati nei suoi figli.

Proprietà

Se *h* è un heap che codifica un albero quasi-completo con *n* elementi, gli elementi da 0 a $(n/2)-1$ sono nodi interni.

Max-Heapify

Procedura che serve a mantenere le proprietà dell'heap su un sottoalbero radicato ad *i*.

Analizzo *i* e i suoi due figli, se uno dei due è più grande lo rimetto in ordine.

Il **tempo di esecuzione** è:

$$T(n) \leq T\left(\frac{2n}{3}\right) + \theta(1) \rightarrow T(n) = O(\log n)$$

Heapsort

Procedura di **ordinamento** che si avvale dell'heap.

Comincia costruendo un heap sull'array usando build-heap (tempo $O(n\,log\,n)$). Esegue un ordinamento sull'array, dove ad ogni chiamata ricorsiva ripristina le proprietà dell'heap (tempo $O(\log n)$ nel caso peggiore).
Complessivamente ha **complessità** nel caso migliore, peggiore e medio di:

$$\theta(n \log n)$$

Alberi binari di ricerca

Possono essere usati sia come **dizionari** (seach, insert, delete) che come **code di priorità** (minimum, maximum, predecessor, successor).

Proprietà

- In un abr il valore del figlio sinistro di un nodo *n* è sempre **minore** del valore del nodo *n*.
- In un abr il valore del figlio destro di un nodo *n* è sempre **maggiore** del valore del nodo *n*.

Negli abr l'inserimento, la cancellazione, la ricerca e il calcolo del minimo e del massimo hanno tutti **complessità** $\theta(h)$, dove *h* è la profondità dell'abr.

Nel **caso peggiore** (albero sbilanciato) h è $\theta(n)$.
Nel **caso migliore** (albero bilanciato) è h $\theta(\log n)$.
Con una visita simmetrica dell'abr si producono valori in ordine crescente.

Alberi rosso-neri

Un albero rosso-nero ha come scopo quello di mantenere **bilanciato** un albero binario di ricerca, in modo da avere un minor tempo di esecuzione e complessità per le operazioni su di esso.

Il **criterio di bilanciamento** è che la lunghezza del cammino più lungo tra la radice e un figlio *null* è al massimo due volte la lunghezza del cammino più corto.
Per gestire il bilanciamento si aggiunge una **sentinella** in modo che nessun nodo abbia solo figlio destro o sinistro, e si colora ogni nodo di rosso o di nero.

Caratteristiche

- Ogni nodo è **rosso** o **nero**
- La radice e la sentinella sono **nere**
- Se un nodo è **rosso** entrambi i suoi figli sono **neri**
- Tutti i **cammini** della radice alla sentinella contengono lo stesso numero di nodi **neri.**

Il cammino più lungo dalla radice alla sentinella corrisponde all'altezza + 1.
Gli alberi rosso-neri devono essere degli **abr**.
Non tutti gli abr possono essere alberi rosso-neri.
In un albero rosso-nero, **l'altezza** $h \in \theta(log\ n)$.

Quindi tutte le operazioni di **consultazione** (search, minimum, maximum) che di solito vengono eseguite in tempo Θ(n) su un abr, diventano eseguibili in $\boldsymbol{\theta(log\ n)}$ su un albero rosso-nero.

Anche le operazioni di **inserimento** e **cancellazione** di un nodo nell'albero rosso nero posso essere eseguite in tempo $\boldsymbol{\theta(log\ n)}$, ma devono essere abbinate a delle procedure che garantiscono la permanenza delle proprietà dell'albero rosso-nero.
Una procedura base di questo tipo è la **rotazione**, che viene eseguita in tempo costante ($\boldsymbol{\theta(1)}$) e non altera il colore dei nodi.

Complessità dei problemi

Un problema ammette infiniti algoritmi corretti.

- Un problema ha complessità temporale $O(f(n))$ se esiste un algoritmo che lo risolve che ha complessità temporale $O(f(n))$.
  Quindi $O(f(n))$ sono risorse sufficienti a risolvere il problema.
  In questo caso, $O(f(n))$ è un **limite superiore** alle risorse sufficienti a risolvere il problema, quindi potrebbe esistere un algoritmo che spende meno risorse per risolverlo.

- Un problema ha complessità temporale $\Omega(f(n))$ se esiste un algoritmo che lo risolve che ha complessità temporale $\Omega(f(n))$.
  Quindi $\Omega\big(f(n)\big)$ sono risorse sufficienti a risolvere il problema.
  In questo caso, $\Omega(f(n))$ è un **limite inferiore** alle risorse sufficienti per risolvere il problema,

quindi non è possibile che il problema possa essere risolto spendendo meno di $\Omega(f(n))$.
Dato che non è possibile considerare tutti gli algoritmi che possono risolvere un problema, non esiste un metodo preciso per determinare $\Omega(f(n))$.

- Un problema ha complessità temporale $\theta\big(f(n)\big)$ se ha contemporaneamente complessità temporale $O(f(n))$ e $\Omega(f(n))$.
  Limite inferiore e superiore coincidono, quindi $f(n)$ è la complessità intrinseca del problema
  Non è sempre possibile determinare $\theta\big(f(n)\big)$.

Alcuni esempi di problemi dalla complessità ignota sono il problema del commesso viaggiatore e, più in generale, tutti i problemi **NP-completi**.
Se si trovasse un algoritmo in grado di risolvere in tempo polinomiale un qualsiasi problema NP-completo, si potrebbero risolvere tutti i problemi NP-completi nello stesso tempo.
Per ora si ritiene, anche se non è mai stato dimostrato, che **non esiste** un algoritmo polinomiale in grado di risolvere i problemi NP-completi.

Il problema della ricerca

Nel problema della ricerca è nota una collezione di **coppie <chiave, valore>**. Un'istanza del problema può essere il valore di una chiave e la soluzione del problema il relativo valore.

Il problema della ricerca presenta diverse complessità nel caso peggiore, in base alla struttura dati utilizzata.

Liste	$O(n)$
Array non ordinati	$O(n)$
Array ordinati	$O(\log n)$
Alberi rosso-neri	$O(\log n)$

Possiamo dimostrare tramite un **albero di decisione** (ovvero un albero i cui nodi interni sono i vari confronti eseguiti dall'algoritmo e le foglie sono le possibili risposte) che nel caso peggiore una ricerca basata su confronti implica $\theta(\log n)$ confronti.

Allo stesso modo possiamo dimostrare un lower bound sugli **algoritmi di ordinamento per confronto**.
L'esecuzione di un algoritmo di ordinamento per confronto corrisponde alla discesa in un albero di decisione con $n!$ foglie, dove *n* è il numero di elementi da ordinare
Nel caso peggiore il numero di confronti (nodi interni nel cammino radice-foglia) è $\Omega(n \log n)$.

Quindi l'algoritmo **merge sort**, che ha complessità $\theta(n \log n)$, è un ottimo algoritmo di ordinamento per confronto.

Ordinamento in tempo lineare

In alcuni casi si possono ordinare elementi senza che vengano effettuati confronti tra i loro valori, se ad esempio si dispone di informazioni sulla distribuzione dei valori degli elementi.

Quindi si possono concepire algoritmi con **complessità** $\theta(n)$ nel caso peggiore, al posto di $\theta(n \log n)$ degli algoritmi di ordinamento visti fino ad ora.

Gli algoritmi seguenti suppongono solo valori interi in input.

Counting Sort

Ognuno degli *n* elementi in input è un **intero** nell'intervallo [1…k]

Quando $k \in O(n)$, allora l'ordinamento viene eseguito in tempo $O(n)$.

L'idea è di determinare il numero di elementi minori di x, per ogni x in input, ed in base a questo scegliere la posizione di x nell'array.
Ad esempio, se ci sono 17 elementi minori di x, metterò x nella posizione 18 dell'array.

- Si creano *k* contatori, uno per ogni possibile valore di *i*. (Tempo $\theta(n)$)
- Per ogni *i,* conto quanti sono gli elementi in input uguali ad *i.* (Tempo $\theta(n)$)
- Per ogni *i*, conto quanti sono gli elementi in input minori di *i.* (Tempo $\theta(n)$)
- Scorro l'array di input e posiziono gli elementi nel punto opportuno dell'array di output. (Tempo $\theta(n)$)

Proprietà

- L'algoritmo è **stabile**, ovvero due posizioni con la stessa chiave non vengono scambiate
- Non è un algoritmo **in loco**, ovvero l'output è memorizzato su un array diverso rispetto a quello di input
- Non è un algoritmo **online**, ovvero bisogna conoscere tutti gli elementi prima di iniziare ad ordinare
- Su un input generico ha **complessità** $\theta(n + k)$

Bucket Sort

- Creo una **lista** (bucket) per ogni *i* minore o uguale a *k.* (Tempo $\theta(n)$)
- Scorro l'array di input e metto l'elemento corrente nel bucket che corrisponde al suo valore. (Tempo $\theta(n)$)
- Svuoto i bucket mettendo i loro elementi nell'array di output. (Tempo $\theta(n)$)

Proprietà

- È un algoritmo **stabile**, ovvero due posizioni con la stessa chiave non vengono scambiate
- Non è un algoritmo **in loco**, ovvero l'output è memorizzato su un array diverso rispetto a quello di input
- Non è un algoritmo **online**, ovvero bisogna conoscere tutti gli elementi prima di iniziare ad ordinare
- Su un input generico ha **complessità** $\theta(n + k)$

Se invece si suppone che i numeri in input siano **arbitrari**, quindi non necessariamente interi, ed uniformemente distribuiti in un determinato intervallo, si possono generalizzare i due algoritmi appena visti.

Radix Sort

Assume che gli elementi da ordinare siano in una **sequenza** di *d* cifre (oppure *d* colonne) e che il valore di ogni cifra sia $O(n)$.

L'ordinamento deve essere fatto in base alla cifra più **significativa**. Vengono ordinate cifre a partire da quella più a destra della sequenza.

Dato che dobbiamo eseguire *d* ordinamenti, il **tempo complessivo** è $\theta(d(n+k))$, dove *k* è il valore massimo dei valori nelle celle.

Hash

Le **tabelle hash** sono strutture dati usate per mettere in corrispondenza una chiave con un valore.

Le **tabelle hash** realizzano degli array associativi, ovvero delle coppie <chiave,valore>, dove le operazioni consuete sono l'inserimento di una nuova coppia, la cancellazione di una specifica chiave e la ricerca del valore corrispondente ad una chiave.

Nelle tabelle hash inserimento, ricerca e cancellazione avvengono con tempo θ(n) nel **caso peggiore** e con tempo θ(1) nel **caso medio**. Quest'ultimo caso è quindi migliore rispetto agli alberi rosso-neri, dove era $\theta(\log n)$.

Si utilizza una struttura con un **singolo array**

Per realizzare una tabella hash devo definire due funzioni: **EQUAL** e **HASH**.

Si utilizza un array T per memorizzare i dati associati. La dimensione dell'array è m, che è molto minore della dimensione di tutte le chiavi K, ma molto vicina al numero di chiavi effettivamente utilizzate. Quindi viene definita una funzione hash che trasforma le chiavi di K interi nel range [0…m-1].

L'array T, che viene indicizzato ai dati associati tramite la **funzione HASH**, è chiamato **tabella hash**.

Caratteristiche della funzione HASH

- Deve essere **deterministica**, ovvero data una chiave k, dalla funzione calcolata in k avrò sempre lo stesso risultato. Se non fosse deterministica, dopo aver messo i valori nell'array non riesco più a risalire alla loro posizione.
- Deve essere calcolabile in tempo **costante**, rispetto agli elementi della tabella hash.
- Distribuisce le chiavi utilizzate in esecuzione in maniera **pseudocasuale** nell'intervallo [0…m-1]. Questo significa che, data una chiave $k \in K$, la probabilità che $HASH(k) = c$ deve essere la stessa per ogni casella $c \in [0 \dots m-1]$ di T.
  Quindi per chiavi simili vengono generati HASH diversi.

Dato che il codominio della funzione è molto più piccolo del dominio, è inevitabile che si generino delle **collisioni**. Queste collisioni vengono gestite tramite **liste di trabocco**. Quindi ogni elemento dell'array T è un riferimento al primo elemento della lista di trabocco, che è una lista semplicemente concatenata con tre campi: key, info e next.

Fattore di carico α : Rapporto tra *n* elementi memorizzati e *m* posizioni disponibili.
Se α<1 ci sono molte posizioni disponibili rispetto agli elementi memorizzati.
Se α>1 ci sono molti elementi da memorizzare rispetto alle posizioni disponibili.

Il **caso migliore** si ha quando la lista di trabocco è vuota, o ha un solo elemento, e si ha complessità θ(1).

Il **caso peggiore** si ha quando tutte le chiavi utilizzate corrispondo alla stessa posizione, quindi avrò complessità θ(n) perché devo eseguire una ricerca in una lista con n posizioni e calcolare EQUAL per n volte.

Nel **caso medio** si hanno due possibili scenari:

- La funzione HASH distribuisce uniformemente le chiavi, in questo caso le operazioni hanno complessità θ(α).
- La funzione HASH non dà garanzie rispetto alla distribuzione delle chiavi, in questo caso la complessità è la stessa del caso peggiore.

Potrebbe sembrare che α ∈ θ(n) data la definizione di α=n/m, ma in realtà *m* cambia al crescere di *n*, quindi quando il fattore di carico supera una certa soglia la dimensione *m* della tabella viene raddoppiata, quindi α viene dimezzato. Quindi il costo delle operazioni è ancora costante, ovvero $\theta(1)$.

Tipi di funzioni HASH

Esistono diversi tipi di funzioni HASH che variano in base alla tipologia di chiave (intera, stringa od oggetto arbitrario).

Nelle funzioni HASH per **interi**, esistono due metodi: **metodo della divisione** e **metodo della moltiplicazione**.

Nel **metodo della divisione** di usa il resto di una divisione intera: $h(k) = k \bmod m$. Il metodo è molto veloce ma le chiavi devono essere pseudocasuali.

Nel **metodo della moltiplicazione** si utilizza come hash la parte intera del prodotto
$h(k) = \lfloor m \cdot (k \cdot irr - \lfloor k \cdot irr \rfloor) \rfloor$, dove *irr* è un numero irrazionale in (0,1) e *m* è arbitrario.

Insiemi

Un insieme è una **collezione di elementi omogenei**.
Posso essere realizzati tramite tabelle hash in cui lo stesso elemento funge sia da valore che da chiave.

Tabelle Hash con indirizzamento aperto

Le **liste di trabocco**, usate nelle tabelle Hash per evitare le collisioni, utilizzano memoria aggiuntiva.
Nelle tabelle Hash con indirizzamento aperto, le collisioni vengono gestite senza ricorrere a memoria aggiuntiva.
Quindi, tutti gli elementi vengono memorizzati nella tabella stessa, non si usano più le liste di trabocco.

Il fattore di carico α non potrà mai superare 1, altrimenti viene generato un errore di overflow oppure un raddoppio della tabella

In caso di **collisione** con la chiave *k,* si ispezione la tabella per cercare una cella libera per *k*.
La tabella viene usata come una sequenza circolare, quindi dopo l'ultima posizione ci sarà la prima.
L'ordine con cui vengono scandite le celle dipende dal valore di k.

La sequenza di posizioni esaminate quando una chiave è stata **inserita** è la stessa di quando la medesima chiave viene **ricercata**. Se una cella candidata viene trovata vuota, la ricerca fallisce.

La **cancellazione** su tabelle hash con indirizzamento aperto è un'operazione critica, dato che la cancellazione dell'elemento potrebbe interrompere le sequenze di scansione.
Proprio per questo, le tabelle hash ad indirizzamento aperto tendono a non essere usate quando sono previste cancellazioni.

Si definiscono le seguenti funzioni di scansione:

Scansione Lineare

Funzione **facile** da implementare

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

Il primo tentativo viene eseguito su $h(k)$, successivamente su $h(k)+1$, poi $h(k)+2, h(k)+3$ ecc.

La scansione lineare è affetta dal fenomeno dell'**addensamento primario**: in sostanza si formano lunghe sequenze di celle occupate quando si vano ad inserire le chiavi e questo fa **aumentare il tempo della ricerca**

Scansione Quadratica

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

Il primo tentativo viene eseguito sempre su $h(k)$

La scelta di c_1 e c_2 è fondamentale per la correttezza della funzione.

La scansione quadratica è affetta dal fenomeno dell'**addensamento secondario**: in sostanza posso generare solo m sequenze diverse che differiscono solo per l'elemento iniziale.

Doppio Hashing

$$h(k,i) = (h_1(k) + i h_2(k))\ mod\ m$$

$h_1(k)$ e $h_2(k)$ sono due funzioni hash ausiliarie.
Il doppio hashing non ha fenomeni di addensamento primario o secondario.
Sono possibili $O(m^2)$ sequenze di scansione distinte che si distinguono per la cella iniziale e per il passo.

Grafi

Un **grafo orientato** (o **diretto**) è costituito da un insieme di **nodi** V e da un insieme di **archi** E.
Ogni arco è una coppia ordinata di nodi (u,v).

Un **cammino** è una sequenza di nodi per cui esistono degli archi che li collegano. Un cammino è detto **semplice** se tutti i suoi nodi sono distinti. Il numero di archi è la **lunghezza** del cammino.

Un **grafo non orientato** (o **non diretto**) l'insieme degli archi E è un insieme di coppie non ordinate, quindi l'arco non ha un orientamento e le coppie (u, v) e (v, u) rappresentano lo stesso arco.

I grafi possono essere rappresentati con **matrici di adiacenza**, **liste di adiacenza** oppure **oggetti e riferimenti**

Matrice di adiacenza

Si preferisce quando il grafo è particolarmente **denso**.

Si usa una matrice in cui l'elemento di posizione (i, j) segnala se esiste l'arco (i, j).

Occupa $\theta(n^2)$ spazio.

In caso di grafo non orientato, la matrice è **simmetrica**.

Liste di adiacenza

Si usa un array di **liste doppiamente concatenate**, dove nell'array si mette il riferimento al primo elemento della lista.

La lista di adiacenza di un nodo può essere lunga $O(m)$, i nodi possono essere $O(n)$.

In totale occupa $\theta(n) + \theta(m)$ spazio, e per percorrerla tutta ci si mette $O(n) + O(m)$.

Oggetti e Riferimenti

Ogni nodo ed ogni arco sono degli **oggetti**, al cui interno possono essere memorizzati identificatori, pesi, marcatori ecc.

Nodi e archi sono conservati in **liste doppiamente concatenate**.

Ogni nodo ha una lista dei suoi archi incidenti e ogni arco ha dei riferimenti ai suoi due estremi.

Archi diretti e indiretti vengono rappresentati allo stesso modo

Visite di un grafo

Nella visita di un grafo, i nodi non sono raggiunti in maniera casuale, ma in un ordine determinato dalla forma del grafo

Durante la visita, i nodi vanno marcati con un **marcatore** (ovvero un valore associato ad ogni nodo, che viene chiamato anche colore) per evitare di ciclare all'infinito.

Visita in ampiezza (BFS)

Nella **visita in ampiezza** si visitano i nodi seguendo un ordine imposto dalla loro **distanza**, ovvero prima vengono visitati i nodi più vicino, poi i più lontani.

Ogni nodo viene inserito ed estratto dalla coda una sola volta, ed ogni arco è considerato sia dal nodo di partenza che da quello di arrivo.
Quindi la **complessità** nel caso peggiore è $\theta(n + m)$.

Visita in profondità (DFS)

Nella **visita in profondità**, a partire da un nodo, si percorre tutto il grafo fino a che si trovano nodi non visitati. Quando tutti i vicini sono stati visitati si torna indietro verso il nodo di partenza per verificare che non ci siano nodi **adiacenti** non visitati

Su ogni nodo la visita viene lanciata una sola volta ed ogni arco è considerato sia dal nodo di partenza che dal nodo di arrivo.
Quindi la **complessità** nel caso peggiore è $\theta(n + m)$.

Algoritmi di ordinamento

Gli algoritmi di ordinamento si dividono in due tipi: **Greedy** & **Divide et Impera**

Greedy

Gli algoritmi greedy sono gli algoritmi che scelgono una soluzione in base all'opzione che al momento sembra la migliore, quindi più appetibile, ma in generale potrebbe non esserlo.
Un esempio di algoritmo greedy è il **selection sort**.

Divide et Impera

Gli algoritmi divide et impera riducono ricorsivamente il problema in tanti **sottoproblemi** di dimensione estremamente più piccola rispetto al problema originale. Il passo successivo è di trovare le soluzioni nelle istanze più piccole del problema e combinarle per tirare fuori la soluzione al problema originale.

Selection Sort

L'algoritmo si basa sul paradigma **greedy**.

Il selection sort basa il suo funzionamento su una **doppia visita** all'array: con la prima segue l'indice su cui al momento sta lavorando, con l'altra (annidata nella prima) cerca per ogni indice qual è il valore minore del sottoarray destro e lo scambia con l'indice corrente. Dovendo implementare la visita per ogni indice dell'array, la **complessità** sarà $\theta(n^2)$ sia nella versione iterativa (quella con due cicli *for* annidati) che in quella ricorsiva (un *for* che scorre l'indice corrente e una chiamata ricorsiva che trova e scambia i valori).

Selection sort **opera in loco** ed **è stabile**.

Insertion Sort

L'algoritmo Insertion Sort è un algoritmo **incrementale**.
L'idea è di mantenere un sottoinsieme ordinato di elementi. Si inserisce un elemento alla volta facendo crescere il sottoinsieme ordinato traslando tutti gli elementi per far posto al nuovo elemento.
La **complessità** è $\theta(n)$ nel caso migliore e $\theta(n^2)$ nel caso medio e peggiore.

Insertion Sort **opera in loco** ed **è stabile**.

Merge Sort

Merge Sort si basa sul paradigma **divide et impera**.

L'algoritmo divide ricorsivamente la sequenza in due sottosequenze più facili da ordinare, fino ad arrivare a dividere una sequenza di due numeri in due singoli numeri, da lì inizia la parte di **merge** dove si ordinano tutte le sottosequenza create fino ad arrivare alla sequenza completa ordinata.

Si può paragonare il suo procedimento alla creazione di un **albero binario completo**, dove i sottoproblemi da cui parte a combinare sono le foglie dell'albero. Tramite ciò è intuitivo come la sua

complessità sia data da lavoro di ordinamento sugli *n* nodi moltiplicato per l'altezza dell'albero (*log n*).
La **complessità** è quindi $\theta(n \log n)$.

Merge sort **non opera in loco** ed **è stabile**.

Heapsort

Comincia costruendo un **heap** sull'array usando build-heap (tempo $O(n \log n)$). Esegue un ordinamento sull'array, dove ad ogni chiamata ricorsiva ripristina le proprietà dell'heap (tempo $O(\log n)$ nel caso peggiore). In generale ha **complessità** nel caso migliore, peggiore e medio di:

$$\theta(n \log n)$$

Quick Sort

Quick sort si basa sul paradigma **divide et impera**.

L'array A[p ... r] viene **ripartito** (e risistemato) in due sottoarray non vuoti A[p ... q–1] e A[q+1 ... r], in modo che ogni elemento del primo sia minore o uguale ad A[q].
L'indice q viene calcolato dalla procedura di **partizionamento**.
Una volta combinati i due array, A è ordinato.

Il **caso peggiore** è quando la procedura di partizionamento elegge a pivot il valore massimo o minimo dell'array, dato che in quel caso i due sottoarray sono completamente sbilanciati, uno grande solo una casella in meno dell'array originale, e l'altro grande un'unica casella: quella del pivot.

Il **caso migliore** è quando la procedura di partizionamento elegge a pivot il valore mediano dell'array, in questo caso abbiamo due sottoarray perfettamente bilanciati.

La **complessità** è quindi $\theta(n^2)$ nel caso peggiore e $\theta(n \log n)$ nel caso medio e migliore.

Quick sort **opera in loco** ma **non è stabile**.