COMPLESSITA'

Quando si parla di complessità, si parla di complessità in tempo e complessità in spazio.

Complessità in tempo: si intende stimare il tempo impiegato dall'algoritmo; confrontare algoritmi diversi e ottimizzare le parti più importanti. Quando si parla di tempo ci si riferisce al "Wall-clock" time, ovvero il tempo effettivo utilizzato per eseguire l'algoritmo, il quale cambia a seconda della bravura del programmatore, dal linguaggio di programmazione usato, dal codice generato dal compilatore e dalla macchina usata.
Complessità in spazio: si intende stimare la dimensione dell'input. Quando si parla di spazio ci si riferisce al numero di elementi che costituiscono l'input (criterio di costo uniforme)

Come si calcola il tempo di esecuzione?

Per ogni linea di codice si calcola il numero di operazioni elementari che la sua esecuzione comporta.
Se una linea di codice viene ripetuta più volte (ad esempio perché contenuta nel corpo di un ciclo iterativo) durante l'esecuzione , è necessario calcolare quante volte viene eseguita e moltiplicare questo numero per il numero di operazioni elementari che essa comporta.
Infine si sommano i contributi delle linee e si cerca di calcolare la funzione risultante.

Per lo stesso input la complessità può variare in base all'algoritmo (nlogn 2^n n^2 esempio)

Confrontare due algoritmi significa verificare cosa accade al crescere della dimensione dell'input. Si confronta l'andamento asintotico delle funzioni del tempo di esecuzione. E questo tempo di esecuzione asintotico viene rappresentato tramite notazioni asintotiche.

Notazione asintotica O: limite superiore

Se f(n) = O(g(n)) è il tempo di calcolo richiesto da un algoritmo A, O(g(n)) è un limite superiore asintotico per la complessità in tempo di A.

O(g(n))= { f(n): esistono C>0 ed N tali che $0 \leq f(n) \leq Cg(n)$ per ogni $n \geq N$}

Notazione asintotica Ω: limite inferiore

f(n) = Ω (g(n)) è il tempo di calcolo richiesto dall'algoritmo A, allora Ω (g(n)) è un limite inferiore asintotico per la complessità in tempo di A.

Ω (g(n))= { f(n): esistono C>0 ed N tali che $f(n) \geq Cg(n) \geq 0$ per ogni $n \geq N$}

Notazione asintotica Θ: limite stretto

$f(n) =$ Θ (g(n)) è il tempo di calcolare richiesto dall'algoritmo A, allora Θ (g(n)) è il limite stretto asintotico per la complessità in tempo di A.

Θ (g(n))= O(g(n)) ∩ Ω (g(n)) = { f(n): esistono C1, C2 > 0 ed N tali che per ogni $n \geq N$
$0 \leq C1g(n) \leq f(n) \leq C2g(n)$}

Notazione asintotica o: limite superiore non asintoticamente stretto

f(n) = o(g(n)) è il tempo di calcolo richiesto dall'algoritmo A, allora o (g(n)) è un limite superiore non asintoticamente stretto per la complessità in tempo di A.

o(g(n))= { f(n): per qualsiasi costante c>0 esiste una costante n0>0 tale che $0 \leq f(n) < cg(n)$ per ogni $n \geq n0$}

Nella notazione o la funzione f(n) diventa insignificante rispetto a g(n) quando n tende all'infinito.

Notazione asintotica ω : limite inferiore non asintoticamente stretto

f(n) = ω (g(n)) è il tempo di calcolo richiesta dall'algoritmo A, allora ω (g(n)) è un limite inferiore non asintoticamente stretto per la complessità in tempo di A.

ω (g(n))= { f(n): per qualsiasi consente c>0, esiste una costante n0>0 tale che $f(n) > cg(n) \geq 0$ per ogni $n \geq n0$}

Si può esprimere la complessità tramite una relazione di ricorrenza che può essere risolta applicando uno dei tre seguenti modi:
- metodo di sostituzione
- metodo dell'albero di ricorsione
- metodo dell'esperto

Il metodo dell'albero di ricorsione:
in un albero di ricorsione ogni nodo rappresenta il costo di un singolo sottoproblema.
Sommiamo i costi all'interno di ogni livello dell'albero.
Sommiamo tutti i costi per livello per determinare il costo totale di tutti i livello della ricorsione

E' utile quando la ricorrenza descrive il tempo di esecuzione di un algoritmo Divide et Impera.
Utile per ottenere una buona ipotesi che poi viene verificata con il metodo di sostituzione.
L'albero di ricorsione può essere usato come prova diretta di una soluzione della ricorrenza.

Metodo e teorema dell'Esperto:
Fornisce direttamente le soluzioni asintotiche di molte ricorrenze del tipo: T(n)=aT(n/b)+f(n)

1) $T(n) = \Theta(n^{\log_b a})$ se $f(n) = O(n^{\log_b a - \varepsilon})$ per qualche costante $\varepsilon>0$
2) $T(n) = \Theta(n^{\log_b a} \cdot \log n)$ se $f(n) = \Theta(n^{\log_b a})$
3) $T(n) = \Theta(f(n))$ se $f(n) = \Omega(n^{\log_b a+\varepsilon})$ per qualche costante $\varepsilon>0$ ed esistono $c<1$ ed N tali che: $af(n/b) \leq cf(n)$ per ogni $n \geq N$

Strutture dati astratte

Dati: insieme di valori/oggetti
Operazioni: consentono di manipolare gli oggetti

Strutture dati: Organizzazione dei dati che consente di supportare le operazioni previste per un certo tipo di dato in modo efficiente.
Il tipo di dato astratto mi dice cosa vogliamo e la struttura dati mi dice come lo realizziamo.

Strutture dati che ci permettono di avere a che fare con insieme dinamici cioè che variano in base agli insiemi classici.
Nella matematica gli insiemi sono immutabili, ma nell'informatica sono dinamici in quanto possono crescere, ridursi o cambiare nel tempo.

Le operazioni definite a livello astratto sono quelle di inserimento, cancellazione e di verifica di appartenenza.

Insieme dinamico + operazioni inserimento, cancellazione o verifica di appartenenza = DIZIONARIO

Due tipologie di operazioni:
◊ operazioni che restituiscono informazioni sull'insieme;
◊ operazioni di modifica che cambiano l'insieme.

PILA (STACK)
◊ La pila è un tipo speciale di insieme dinamico in cui tutte le operazioni di inserimento e cancellazione avvengono alla stessa estremità (detta top)
◊ si segue il criterio LIFO: il primo elemento che può essere tolto della pila coincide con l'ultimo elemento introdotto.
◊ Per accedere all'elemento i-esimo si devono quindi togliere dalla pila tutti i successivi.

Operazioni :

- Stack-empty(S): restituisce true se S è vuoto; false altrimenti
- Push(S,x): aggiunge x come ultimo elemento di S
- Pop(S): toglie da S l'ultimo elemento e lo restituisce

CODA
◊ una coda è un tipo particolare di insieme dinamico nella quale gli elementi sono inseriti ad una estremità (posteriore) e sono estratti dall'altra
◊ La coda è quindi una lista FIFO

Operazioni:
- Enqueue(Q,x): aggiunge x come ultimo elemento di Q.
- Dequeue(Q): toglie da Q il primo elemento e lo restituisce

Rappresentazioni indicizzate -> i dati sono contenuti in array
Pro: vi è accesso diretto ai dati mediante indici
Contro: la dimensione è fissa

Rappresentazioni collegate -> i dati sono contenuti in record collegati tra loro mediante puntatori (liste)
Pro: dimensione è variabile
Contro: vi è accesso sequenziale ai dati

Una lista concatenata è un insieme di oggetti disposti in ordine lineare. L'ordine è determinato da un puntatore in ogni oggetto.
Una lista doppiamente concatenata è un oggetto con campo • chiave KEY • puntatore NEXT al prossimo livello • puntatore PREV all'elemento precedente.

Tipi di liste:
◊ singolarmente concatenata (manca PREV)
◊ doppiamente concatenata
◊ ordinata (testa è l'elemento minimo e la coda è l'elemento massimo)
◊ non ordinata
◊ circolare (il puntatore PREV della testa punta alla coda e il puntatore NEXT della coda punta alla testa)

Operazioni:
- List-search (L,k): restituisce un puntatore al primo elemento con valore k
- List-insert (L,x): inserisce l'elemento x davanti alla lista concatenata
- List-Delete (L,x) rimuove l'elemento x da L

Una sentinella è un elemento fittizio che consente di semplificare le condizioni al contorno.

Albero Radicato: insieme dinamico di elementi che hanno una relazione gerarchica.

ALGORITMI DI VISITA
Gli algoritmi di visita si distinguono in base al particolare ordine di accesso ai vari nodi:
In profondità (DFS)
In ampiezza (BFS)

Visite in profondità:
◊ Visita in preordine: si visita prima la radice e poi si effettuano le chiamate ricorsivi sul figlio sinistro e destro (radice + sottoalberi da sinistra a destra)
◊ Visita inordine (simmetrica): si effettua prima la chiamata ricorsiva sul figlio sinistro, poi si visita la radice ed infine si effettua la chiamata ricorsiva sul figlio destro.
◊ Visita in postordine: si effettuano prima le chiamate ricorsive sul figlio sinistro e destro e poi si visita la radice. (visito i sottoalberi da sinistra a destra e poi radice)

InsertionSort (n — n^2 — n^2)
L'algoritmo InsertionSort usa un approccio incrementale.
- Ordina il sotto-array A[1,…., j-1] e poi inserisce l'elemento A[j] in modo tale che A[1,…, j] sia ordinato.

Molti algoritmi sono ricorsivi -> questi algoritmi chiamano se stessi in modo ricorsivo una o più volte.
Gli algoritmi ricorsivi adottano la tecnica divide et impera.

•Divide et Impera

prevede tre passi ad ogni livello di ricorsione

- Divide: il problema viene diviso in un certo numero di sottoproblemi
- Impera: i sottoproblemi vengono risolti in modo ricorsivo (se i problemi sono piccoli si possono risolvere in maniera semplice)
- Combina: le soluzioni dei sottoproblemi vengono combinate per generare la soluzione del problema originale.

•MergeSort (nlogn — nlogn — nlogn)
– Se la sequenza da ordinare ha lunghezza 0 oppure 1, è già ordinata. Altrimenti:
– La sequenza viene divisa (divide) in due metà (se la sequenza contiene un numero dispari di elementi, viene divisa in due sottosequenze di cui la prima ha un elemento in più della seconda)
– Ognuna di queste sottosequenze viene ordinata, applicando ricorsivamente l'algoritmo(impera)
– Le due sottosequenze ordinate vengono fuse (combina). Per fare questo, si estrae ripetutamente il minimo delle due sottosequenze e lo si pone nella sequenza in uscita, che risulterà ordinata.

Si può usare InsertionSort per ordinare parti di array di dimensione minore di una certa costante k.

HeapSort O(nlogn) (Build-max-heap O(n))
• cos'è uno HEAP: Un array A[1..n] può essere interpretato come albero binario:
◊ A[1] è la radice,
◊ A[2i] e A[2i + 1] sono i figli di A[i]
◊ A[i/2] è il padre di A[i]

Proprietà dello heap:
• Max-Heap: per ogni nodo i diverso dalla radice A[parent(i)] ≥ A[i]
• Min-Heap: per ogni nodo i diverso dalla radice A[parent(i)] ≤ A[i]

Modifiche all'interno dell'array possono provocare la violazione della proprietà Max-Heap -> infatti A[1] può ritrovarsi più piccolo dei suoi figli.
- A[i] deve "scendere" in modo che il sottoalbero con radice di indice i diventi uno Max-Heap
- Per ripristinare la proprietà di Max-Heap viene usata la Max-Heapify O(logn)

Le code con priorità: strutture dati in cui è possibile immagazzinare degli oggetti x con una priorità (chiave) ed estrarli uno alla volta in ordine di priorità decrescente.

Max-Priorità - Min-Priorità

QuickSort (O(nlogn) — O(n^2) — O(n^2))
Si basa sulla partizione dell'array rispetto ad un suo elemento scelto come pivot. L'operazione viene quindi ripetuta sulle due parti così ottenute.
È basato sul paradigma Divide et Impera:

- Divide: partizione dell'array A[p...r] in due sottoarray A[p...q-1] e A[q+1...r] tale che ogni elemento in A[p... q-1] sia minore o uguale a A[q] che è minore o uguale agli elementi in A[q+1…r].

- Impera: ordina i due sottoarray A[p...q-1] e A[q+1...r] chiamando ricorsivamente QuickSort

- Combina: siccome i due array sono ordinati sul posto non occorre alcuna combinazione

La prestazione del QuickSort Dipende dal partizionamento, che a sua volta dipende da quali elementi vengono utilizzati nel partizionamento.

- Partizionamento bilanciato: l'algoritmo viene eseguito con la stessa velocità del MergeSort
- Partizionamento sbilanciato: l'algoritmo può essere asintoticamente lento quanto l'InsertionSort

CountingSort teta(n - n — n)
Abbiamo un array A con un range di elementi che va da 0 a k. Per poterlo ordinare utilizziamo un array B in cui metteremo la sequenza ordinata, e un array C ausiliario in cui metteremo il numero di occorrenze.

- Quando un algoritmo di ordinamento mantiene l'ordine iniziale tra gli elementi di valore uguale si dice che esso è stabile

RadixSort (O (kn) T(n,d,k)= teta(d(n+k))
Ho un array con al più d cifre in una cera base b. Per ordinare l'array si usa d volte un algoritmo di ordinamento stabile per ordinarej l'array rispetto a ciascuna delle d cifre partendo dalla meno significativa.
~~Assumiamo che i valori degli elementi dell'array siano interi rappresentabili con al più d cifre in una certa base b~~
~~- Es., interi di al più d = 5 cifre decimali, interi di d = 4 byte (cifre in base b = 256) o anche stringhe di d caratteri (b = 256)~~
~~- Per ordinare l'array si usa d volte un algoritmo di ordinamento stabile (es., CountingSort) per ordinare l'array rispetto a ciascuna delle d cifre partendo dalla meno significativa~~

BucketSort (Θ (n) —n — Θ (n^2)

- Assume che i valori da ordinare siano numeri reali in un intervallo semiaperto [*a*,*b*) che per semplicità di esposizione assumiamo sia l'intervallo [0,1)
- Per ordinare un array A[1..n] divide l'intervallo in n parti uguali e usa un array B[0..n-1] di liste (i bucket) mettendo in B[k] gli A[i] che cadono nella k-esima parte dell'intervallo
- Dopo di che riordina ciascuna lista e infine ricopia in A tutti gli elementi delle liste

HASHING

Ulteriore modo di rappresentare in maniera efficiente gli elementi di un insieme dinamico.

Tavole ad indirizzamento diretto: tecnica semplice di rappresentazione quando l'insieme U delle possibili chiavi è ragionevolmente piccolo.
Contro: ◊ occorre riservare memoria sufficiente per tante celle quante sono le possibili chiavi.
- ◊ se l'insieme U delle chiavi è troppo grande allora l'idirizzamento diretto è inutilizzabile.
- ◊ Se le chiavi necessarie sono soltanto una piccola frazione di U la maggior parte della memoria riservata risulta inutilizzata.

Per risolvere il problema dello spreco di memoria si utilizza la Tavola di Hash.

La tavola di hash richiede memoria pari al numero massimo di chiavi contemporaneamente presenti nel dizionario e quindi avrò n celle quante sono le chiavi.

In un tavola di hash da m celle ogni chiave viene memorizzata nella cella h(k) usando la funzione di hash: h : U -> {0…m-1}

Siccome |U| > m esisteranno molte coppie di chiavi distinte k1≠k2 tali che h(k1)=h(k2) -> collisione
Dunque diremo che per ogni k1≠k2 tali che h(k1)=h(k2) vi è una collisione tra le due chiavi k1 e k2.

Nessuna funzione hash può evitare collisioni, ma hanno il compito di minimizzare la probabilità di collisione e prevedere un meccanismo per poterle gestire.

Risoluzione delle collisioni con liste
Gli elementi con lo stesso valore Hash h' vengono memorizzati in una lista concatenata.

Hash uniforme semplice: quando la funzione h(k) distribuisce uniformemente le n chiavi tra le m liste. Ogni chiave da inserire nella tavola è estratta da U con la stessa probabilità 1/m delle altre chiavi di essere inserita in una qualsiasi delle m celle.

lunghezza media di una lista $\Theta(1+\alpha)$ $\alpha = n/m$

Tempo di ricerca di una chiave è sempre 1+a sia che la chiave ci sia o che non ci sia.

Una buona funzione hash deve soddisfare le proprietà dell'hashing uniforme.

Per creare una buona funzione hash bisogna seguire uno di questi metodi:

METODO DELLA DIVISIONE
Si basa sul resto della divisione per m: h(k)= k mod m
E' molto veloce però non funziona bene per ogni valore di m: m non deve essere potenza di 2

METODO DELLA MOLTIPLICAZIONE
h(k) = m(kA mod 1) dove A= costante tra 0 e 1
La scelta di m non è critica e funziona bene con tutti i valori di A anche se si preferisce il valore di

A = ($\sqrt{5}$ - 1)/2 è il valore migliore per minimizzare la possibilità di collisione.
Tuttavia è più lento del metodo della divisione.

Risoluzione delle collisioni con indirizzamento aperto
Nella tecnica di indirizzamento aperto si utilizza solo la tavola di hash e non strutture ausiliare (liste). Tutti gli elementi sono memorizzati nella tavola.
La funzione di hash non individua una singola cella ma una sequenza esaustiva di celle.

L'inserimento di un nuovo elemento avviene nella prima cella libera che si incontra nella sequenza.
La ricerca percorre la stessa sequenza di celle
La cancellazione della cella i non può essere fatta ponendo nil nella cella -> si assegna deleted.

La funzione hash è quindi una funzione h: U X {0,1…m-1} —> {0,1,…m-1} tale che per ogni chiave k la sequenza h(k,0), h(k,1), …., k(k,m-1) sia una permutazione di {0,1,…m-1}

La sequenza di ispezione è una lista ordinata degli slot esaminati, e può essere costruita con tre tecniche.

Ispezione lineare: la funzione h(k,i) si ottiene da una funzione hash ordinaria h'(k) ponendo h(k,i)= (h'(k)+i)modm. l'esplorazione inizia dalla cella h(k,0)= h'(k) e continua con le celle h'(k)+1, h'(k) +2… fino ad arrivare alla cella m-1 dopodiché si continua con le celle 0,1, ecc fino ad aver percorso circolarmente tutta la tabella.
E' un'ispezione molto facile ma ci consente di generare solo m permutazioni invece di m!.
Si tratta di un problema di addensamento primario, ovvero si formano lunghe file di celle occupate che aumentano il tempo medio di ricerca ed inoltre uno slot vuoto preceduto da i slot pieni ha probabilità (i+1)/m di essere riempito.

Ispezione quadratica: la funzione h(k,i) si ottiene da una funzione hash ordinaria ponendo h(k,i) = (h'(k)+c21^2 +c1i)modm. c1 c2 costanti dove c2≠0. I valori m,c1,c2 non possono essere qualsiasi ma devono essere scelti opportunamente in modo che la sequenza di ispezione percorra tutta la tavola.
Lo spazio tra la prima cella e la successiva aumenta però posso ancora avere una coincidenza di ispezione per due chiavi distinte e questo fenomeno è detto problema di addensamento secondario, meno grave del primo ma che mi concede sempre m permutazioni lasciando ancora un tempo di ricerca alto.

Hashing doppio: la funzione hash h(k,i) si ottiene da due funzioni hash ordinarie h1(k) e h2(k) ponendo h(k,i) = (h1(k) +ih2(k))modm. Per far si che la sequenza di ispezione percorra tutta la tavola il valore di h2(k) deve essere relativamente primo con m.
E' migliore rispetto alle altre due ispezioni perché mi consente di trovare m^2 permutazioni e mi riduce notevolmente i fenomeni di addensamento e rende il comportamento della funzione molto vicino a quella ideale dell'hash uniforme.

Complessità:
il numero atteso di ispezioni per:
- una ricerca senza successo è al massimo 1/(1-a)
- una ricerca con successo è al massimo 1/a log(1/(1-a))
- un inserimento è al massimo 1/(1-a)

Un albero binario di ricerca è un albero binario in cui il valore di ogni nodo è maggiore o uguale dei valori dei nodi del suo sottoalbero sinistro e minore o uguale dei valori dei nodi del suo sottoalbero destro.

Operazioni:
- Inorder-tree-walk(x): restituisce l'elenco ordinato delle chiavi presenti nell'albero (COMPLESSITA' n)
- Tree-search(x,k): restituisce il puntatore al nodo con chiave k (se esiste), oppure nil. (COMPLESSITA' O(h) dove h è l'altezza dell'albero)

- Tree-minimum(x): restituisce l'elemento con chiave minima. (COMPLESSITA' O(h) dove h è l'altezza dell'albero)

- Tree-maximum(x): restituisce l'elemento con chiave massima. (COMPLESSITA' O(h) dove h è l'altezza dell'albero)

◊ Tree-successor(x): restituisce il successore del nodo x. (COMPLESSITA' O(h) dove h è l'altezza dell'albero)

◊ Tree-predecessor(x): restituisce il predecessore del nodo x (COMPLESSITA' O(h) dove h è l'altezza dell'albero.)

◊ Tree-insert(T,z): inserisce il nuovo elemento z dell'albero binario T. (COMPLESSITA' O(h) dove h è l'altezza dell'albero.

◊ Tree-delate(T,z): cancella l'elemento z da T. (COMPLESSITA' O(h) dove h è l'altezza dell'albero)

E' possibile dimostrare che l'altezza media di un albero binario di ricerca è O(log n)

Cancellazione O(h) h altezza albero

Caso 1: il nodo v da cancellare non ha figli

Caso 2: il nodo v da cancellare ha un solo figlio

Caso 3: il nodo v da eliminare ha due figli

Occorrono però delle tecniche per mantenere bilanciato l'albero:
◊ alberi AVL
◊ alberi 2-3
◊ alberi rosso-neri

Il fattore di bilanciamento β(v) di un nodo v è la massima differenza di altezza fra i sottoalberi di v.
In questo caso si parla di bilanciamento in altezza.

ALBERI AVL

Gli alberi AVL sono alberi binari di ricerca bilanciati in altezza
- B(v) = altezza del sottoalbero sinistro di v - altezza del sottoalbero destro di v
- Per convenzione l'altezza di un albero vuoto è -1

Per essere un albero bilanciato il valore finale deve essere -1, 0 o 1.

OPERAZIONI SU ALBERI AVL

◊ La ricerca viene effettuata applicando la stessa procedura usata per gli alberi binari di ricerca.
◊ Inserimento e cancellazione possono sbilanciare l'albero.
◊ Il bilanciamento viene preservato attraverso opportune rotazioni.

RIBILANCIAMENTO

Si ricalcolano i fattori di bilanciamento solo nel ramo interessato dall'inserimento/cancellazione, dal basso verso l'alto.
Se appare un fattore di bilanciamento pari a +-2 occorre ribilanciare per mezzo di rotazioni.

- Si distinguono i seguenti casi:

- DD inserimento nel sottoalbero destro di un figlio destro
- SD inserimento nel sottoalbero destro di un figlio sinistro
- DS inserimento nel sottoalbero sinistro di un figlio destro
- SS inserimento nel sottoalbero sinistro di un figlio sinistro

Complessità operazioni su alberi AVL:

Tutte le operazioni hanno costo O(log n) poiché l'altezza dell'albero è O(log n) e ciascuna rotazione richiede solo tempo costante.

ALBERI ROSSO-NERI

Gli alberi rosso-neri sono alberi binari di ricerca in cui le operazioni Tree-Insert e Tree-Delate sono opportunamente modificate in modo tale da garantire un'altezza dell'albero h=O(log n)

◊ A tale scopo si aggiunge un bit per ogni nodo: il colore che può essere rosso o nero

•◊ Sono alberi binari di ricerca che soddisfano delle proprietà ulteriori:
1. La radice è nera
2. I nodi nil sono neri
3. se un nodo è rosso, allora entrambi i suoi figli sono neri
4. Ogni percorso da un nodo interno ad una foglia ha lo stesso numero di nodi neri.

Il numero di nodi neri lungo ogni percorso da un nodo v (escluso) ad una foglia è detto altezza nera di v indicata bh(v).
L'altezza nera di un albero RB è l'altezza nera della sua radice.

Le operazioni Search, Minimim, Maximum, Successor, Predecessor richiedono tempo O(log n)

Le operazioni Insert, Delete richiedono tempo O (log n) ma siccome esse modificano l'albero possono violare le proprietà degli alberi rosso-neri ed in tal caso occorre ripristinare tali proprietà. -> rotazioni

INSERIMENTO:

- ricerca della posizione usando la stessa procedura usata per gli alberi binari di ricerca
- coloriamo il nuovo nodo di rosso
- se l'albero risultante non soddisfa le quattro proprietà che lo caratterizzano le si devono risistemare. —> RB- Insert-Fixup ricolora i nodi ed effettua delle rotazioni

La 1 proprietà viene violata se il nodo inserito è la radice (quindi in un albero vuoto)
La 3 proprietà viene violata se il padre del nodo inserito è rosso.

RIBILANCIAMENTO:

CASO 1
x è il nodo inserito - rosso
w è lo zio di x - rosso

-> w e il padre di x cambiano colore e diventano neri
-> il nonno di x diventa rosso, se è radice rimane/diventa nero
-> si procede verso l'alto facendo del nonno di x il nuovo x se non soddisfa le proprietà.

CASO 2

x è il nodo inserito - rosso
w è lo zio di x - nero
se w è figlio sinistro allora anche x deve essere figlio sinistro, se invece w è figlio destro allora a sua volta anche x deve essere figlio destro

-> dobbiamo fare una rotazione per ricondurci al caso creando un percorso lineare e non più a “zig-zag”. Se x è figlio destro facciamo una rotazione a sinistra, se x invece è figlio sinistro facciamo una rotazione a destra.

CASO 3

x è il nodo inserito - rosso
w è lo zio di x - nero
se w è figlio destro allora x sarà figlio sinistro, se invece w è figlio sinistro allora x sarà figlio destro.

-> il padre di x diventa nero
-> il nonno di x diventa rosso
-> si effettua una rotazione sul padre di x: a destra se x è figlio sinistro, a sinistra se x è figlio destro.

CANCELLAZIONE:

- L’algoritmo di cancellazione per alberi rosso-neri è costruito sull’algoritmo di cancellazione per gli alberi binari di ricerca.
- Dopo la cancellazione si deve decidere se è necessario ribilanciare o meno
- se il nodo cancellato è rosso non sono necessarie operazioni di bilanciamento -> le altezze nere non sono cambiate; non sono stati creati nodi rossi consecutivi; la radice resta nera perché se il nodo cancellato è rosso allora non può essere la radice.

- Se il nodo cancellato è nero: -> ◊ possiamo violare la proprietà 1: la radice può essere un nodo rosso; ◊ possiamo violare la proprietà 3: se il genitore e uno dei figli del nodo cancellato erano rossi; ◊ abbiamo violato la proprietà 4: altezza nera cambiata

Se il nodo cancellato è nero con figlio rosso, bisogna cambiare il colore del figlio da rosso a nero.
Se il nodo cancellato è nero con figlio nero allora si può riscontrare uno di questi 4 casi:

CASO 1

x è il figlio del nodo cancellato - nero
w è il fratello di x - nero
w ha figli neri

-> w cambia colore in rosso
-> il padre di x, indipendentemente dal suo colore originale prende dal figlio il colore nero. Quindi se era rosso diventa nero, e se era nero diventa nero-nero.

CASO 2

x è il figlio del nodo cancellato - nero
w è il fratello di x - rosso
w ha figli neri

-> w cambia colore e diventa nero
-> il padre di x cambia colore e diventa rosso
-> si effettua una rotazione: se x è a destra allora si ruota a destra, se x è a sinistra allora si ruota a sinistra.

CASO 3

x è il figlio del nodo cancellato - nero
w è il fratello di x - nero
w ha il figlio della medesima posizione di x rosso: se x è un figlio sinistro allora il figlio sinistro di w è rosso, se x è figlio destro allora il figlio destro di w sarà rosso.

-> w cambia colore in rosso
-> il figlio rosso di w diventa nero
-> si effettua una rotazione su w per farlo diventare figlio (destro o sinistro a seconda della posizione) del suo stesso figlio.
In questo modo ci siamo ricondotti al CASO 4 -> l'albero non è ancor bilanciato.

CASO 4
x è figlio del nodo cancellato - nero
w è il fratello di x - nero
w ha il figlio nella posizione opposta di quella di x rosso: se x è figlio sinistro allora il figlio destro di w sarà rosso, se x invece è figlio destro allora il figlio sinistro di w sarà rosso.

-> il figlio rosso di w cambia colore in nero
-> w e il padre di x si invertono i colori
-> si effettua una rotazione su w in direzione di x: se x è a sinistra allora si ruota a sinistra, altrimenti a destra.

ALBERI 2-3 (tutte le operazioni hanno costo O(logn) poiché l'altezza dell'albero è O(logn)

•Un albero 2-3 è un insieme di nodi 2-3 che soddisfa la definizione di albero e che verifica le seguenti proprietà:
◊ le foglie sono tutte sullo stesso livello
◊ ogni nodo ha al più tre figli

Nodo 2-3: è un insieme ordinato contenente 2 chiavi k1 e k2 in ordine crescente e tre puntatori p1, p2, p3 tali che:
- p1 punta al sottoalbero contenente chiavi k tali che k<k1
- p2 punta al sottoalbero contenente chiavi k tali che k1<k<k2
- p3 punta al sottoalbero contente le chiavi k tali che k>k2.

Inserimento: per inserire una chiave devo scorrere l'albero come un normale albero di ricerca che presenta i nodi in ordine crescente e mi fermo quando trovo il nodo adatto alla mia chiave. Se il nodo ha una chiave sola allora aggiungo la mia nuova chiave; se il nodo ha gia due chiavi eseguo uno SPLIT —> tre chiavi in un nodo, spezzo il nodo in due con un processo ricorsivo ed inserisco la chiave a1 nel primo nodo, la chiave a3 nel secondo nodo. La chiave a2 va spostata nel nodo padre e se si presenta anche qui lo stesso problema si eseguo un ulteriore split. -> aumenta l'altezza dell'albero.

Cancellazione: Se si cancella una chiave ci si può trovare in una situazione con svariati nodi semivuoti: si esegue un MERGE ricompattando i nodi interessati in modo tale che ogni nodo abbia più chiavi possibili e facendo diminuire l'altezza dell'albero.

B-ALBERI

I B-alberi sono alberi bilanciati particolarmente adatti per memorizzare grandi quantità di dati in memoria secondaria (disco).

Sono simili agli alberi rosso-neri ma sono progettati per minimizzare il numero di accessi a disco.

Operazioni: Insert, Delate, Search + SPLIT e JOIN (MERGE)

I nodi dei B-Alberi possono contenere un numero n di chiavi ed avere n+1 figli con n≥1.

Un elevato grado di diramazioni riduce in modo drastico sia l'altezza dell'albero sia il numero di letture da disco necessarie per cercare la chiave.

•Proprietà:
Un B-albero di ordine m è un albero bilanciato che soddisfa le seguenti proprietà:

- ◊ ogni nodo contiene al più m-1 elementi.
- ◊ Ogni nodo contiene almeno m/2 (per eccesso) - 1 elementi, la radice può contenere anche un solo elemento.
- ◊ ogni nodo non foglia contenenti j elementi ha j+1 figli
- ◊ ogni nodo ha una struttura del tipo: p0(k1,r1)p1(k2,r2)p2…pj-1(kj,rj)pj j numero elementi del nodo
- ◊ k1….kj sono chiavi ordinate k1<…<kj
- ◊ nel nodo sono presenti j+1 riferimenti ai nodi figli p0,…pj e j riferimenti ai file dati r1…rj.

L'altezza di un B-albero è O(logt n) dello stesso ordine O (log2 n) degli alberi RB.
Più la base del logaritmo t è alta e più il b-albero diventa conveniente.

Ricerca: parte dalla radice, se non c'è si prosegue nel sottoalbero del nodo che può contenere l'elemento. Se il nodo corrente è un nodo foglia e l'elemento non viene trovato vuol dire che non è presente nell'albero.
Il costo della ricerca di un elemento è il numero di nodi letti (da 1 a h).

Inserimento: Si esegue la ricerca per verificare se l'elemento è già presente. Se la chiave che voglio inserire non è presente la inserisco in un nodo foglia: ◊ se la foglia non è piena si inserisce l'elemento.

- ◊ se la foglia è piena si esegue uno SPLIT —> inserisco la chiave nella posizione che gli spetta dopodiché suddivido il nodo in due portando nel nodo padre l'elemento centrale. -> altezza albero incrementa di una unità.

Il costo dell'operazione di inserimento sono: caso migliore, senza suddivisione h + 1 (si leggono h nodi e si scrive una foglia)
nel caso peggiore, suddivisione fino alla radice 3h+1 (si leggono h nodi e si scrivono 2h+1 nodi)

Cancellazione: Si esegue un'operazione di ricerca per individuare il nodo che contiene l'elemento da cancellare.

- ◊ se l'elemento da cancellare non si trova in un nodo foglia, lo rimpiazzo con il suo successore.

◊ se l'elemento si trova in un nodo foglia: - se la foglia non è troppo vuota cancello l'elemento; - se la foglia non soddisfa il Kmin inizia un processo di concatenazione o bilanciamento.

CONCATENAZIONE
Due nodi n e n1 possono essere concatenati tra loro se contengono un numero di chiavi inferiore al Kmax (m-1).
Rimane un puntatore (che puntava d n1) che non punta più e lo faccio sparire mettendo la chiave del nodo genitore in n.
La concatenazione si può propagare fino alla radice causandone l'eliminazione —> diminuisce l'altezza dell'albero.

BILANCIAMENTO
Se tra due nodi adiacenti non si può applicare il processo di concatenazione allora si distribuiscono gli elementi in modo bilanciato. Non influisce sulla struttura dell'albero ma sulle chiavi stesse.
Virtualmente unisco i due nodi n e n1 e la chiave del nodo genitore che presentano un numero eccessivo di chiavi. L'elemento mediano diventa una chiave del nodo padre mentre gli elementi restanti si suddividono in maniera equa e bilanciata nei due nodi di partenza.

Il costo dell'operazione di cancellazione sono: caso migliore, la cancellazione avviene in un nodo foglia e non sono necessarie operazioni di concatenazione e bilanciamento, h+1 (si leggono h nodi e si scrive una foglia)
nel caso peggiore, tutte le pagine del percorso devono essere concatenate tra loro ad eccezione delle prime due, il figlio della radice cosi come la radice vengono quindi modificati, 3h (si leggono 2h-1 nodi e se ne scrivono h+1)

Un B-albero è efficiente per la ricerca e la modifica dei singoli record, ma non è adatto alle elaborazioni di tipo sequenziale nell'ordine dei valori di chiave, né per la ricerca di valori di chiave compresi in un dato intervallo.

Per cercare più chiavi contemporaneamente in un determinato intervallo si usano i B+ALBERI

Proprietà:

◊ ogni nodo contiene al più m-1 elementi.
◊ Ogni nodo contiene almeno m/2 (per eccesso) - 1 elementi, la radice può contenere anche un solo elemento.
◊ ogni nodo non foglia contenenti j elementi ha j+1 figli
◊ ogni nodo foglia ha una struttura del tipo: (k1,r1)(k2,r2)…(kj,rj) j numero elementi del nodo
◊ k1….kj sono chiavi ordinate k1<…<kj
◊ nel nodo sono presenti j riferimenti ai file dati r1…rj.
◊ Ogni nodo foglia ha un puntatore al nodo foglia precedente e successivo
◊ Ogni nodo non foglia ha una struttura p0k1p1k2p2,..pj-1kjpj

B-Alberi	B+ -Alberi
Non c'è un collegamento tra i nodi foglia	I nodi foglia sono collegati tra loro in modo da formare una catena ordinata in base al valore delle chiavi
Le chiavi non si ripetono mai	E' possibile avere delle chiavi ripetute
I nodi intermedi usano due puntatori per ogni valore della chiave ki	Supporta efficientemente le interrogazioni basate su intervalli efficientemente
	La ricerca di una singola chiave è in media più costosa dato che è necessario raggiungere un nodo foglia per ottenere il puntatore ai dati.
Sono più convenienti per quanto riguarda l'occupazione in memoria dato che le chiavi non si ripetono	Sono i migliori per operazioni che richiedono il reperimento di record ordinati in base al valore della chiave o per intervalli di chiave

GRAFI

Un grafo G = (V,E) è costituito da un insieme di vertici V e un insieme di archi E ciascuno dei quali connette due vertici V detti estremi dell'arco.

Un grafo può essere:
- ◊ Orientato: quando vi è un ordine tra i due estremi degli archi. In questo caso il primo estremo si dice coda ed il secondo testa. (uv≠vu)
- ◊ Non orientato (uv=vu)
- ◊ Semplice: se non presenta cappi e non presenta archi con gli stessi estremi.
- ◊ Multigrafo: se ha archi con gli stessi estremi.

Il cappio è un arco con gli stessi estremi.

Il grado δ(v) di un vertice v è il numero di archi incidenti ad esso. Se il grafo è orientato δ(v) si suddivide in un grado entrante δ-(v) che è il numero di archi entranti in v ed un grado uscente δ+(v) che è il numero di archi uscenti da v.

Le componenti connesse di un grafo sono le classi di equivalenza dei suoi vertici rispetto alla relazione di raggiungibilità.
Un grafo non orientato si dice connesso se esiste almeno un cammino tra ogni coppia di vertici.

Le componenti fortemente connesse di un grafo orientato sono le classi di equivalenza dei suoi vertici rispetto alla relazione di mutua accessibilità.
Un grafo orientato si dice fortemente connesso se esiste almeno un cammino da ogni vertice u ad ogni altro vertice v.

RAPPRESENTAZIONE DEI GRAFI

Liste di adiacenza
Adj[u] è la lista dei nodi adiacenti ad u, ciò vuol dire che c'è un arco da u a v. Esistono liste di adiacenza per ogni nodo.

Non c'è modo veloce per determinare se un arco uv è presente nel grafo.
Quantità di memoria richiesta : O(|V| + |E|)

Matrice di adiacenza

Ho una matrice costituita da tante righe e tante colonne quanti sono i nodi, tant'è che lo spazio occupato è pari a |V| X |V|.
Si tratta di una matrice booleana composta da 0 e 1 per indicare la presenza o l'assenza di un collegamento ad un nodo.

Per determinare se un arco uv è presente nel grafo si richiede tempo costante dovuto ad un accesso diretto.
Quantità di memoria richiesta: O(|V|^2)

VISITA DEI GRAFI

Visita in ampiezza

Dato un grafo G=(V,E) ed un vertice particolare s € V detto sorgente, la visita in ampiezza (Breadh-first search) parte da s e visita sistematicamente il grafo per scoprire tutti i vertici che sono raggiungibili da s. Dopodiché calcola la distanza (lunghezza minima di un cammino dalla sorgente al vertice) di ogni vertice del grafo dalla sorgente. Produce inoltre un albero BF i cui rami sono cammini di lunghezza minima.

I nodi hanno tre colori:
◊ bianco - i vertici non ancora raggiunti
◊ grigio - i vertici che sto visitando e di cui non ho ancora visitato tutti i nodi adiacenti
◊ nero - i vertici che ho terminato di visitare.
I colori servono per tracciare il lavoro svolto cosi da non ripercorrere lo stesso cammino.

Uso la coda Q per tenere segnati i nodi grigi che devo finire di visitare.

Complessità: O(|V|+|E|)

Visita in profondità

Per questa tipologia di visita si parte da un nodo che non ha archi entranti e si segue un percorso toccando i vari nodi adiacenti fino ad arrivare ad un nodo che non ha archi uscenti. Se incontro nodi gia visitati, torno indietro e cambio percorso. E' possibile che ci siano dei nodi non visitati quindi prendo un nodo (nuova sorgente) ed inizio il percorso. Continuo questo procedimento fino a quando non ho finito di visitare tutto il grafo.

I puntatori al predecessore definiscono la foresta di ricerca in profondità.
I vertici possono essere:
◊ bianco - vertici non ancora raggiunti
◊ grigio - vertici scoperti
◊ nero - vertici la cui lista di adiacenze è stata completamente esplorata

Marcatempo: • d'inizio: indica con un valore numerico quando il nodo viene scoperto colorato di grigio.
• di fine: registra quando un vertice viene completato e colorato di nero.

Complessità: O(|V|+|E|)

Classificazione degli archi:

◊ archi d'albero: archi uv con v scoperto visitando le adiacenze di u
◊ archi all'indietro: archi uv con u=v; oppure v ascendente di u in un albero della foresta di ricerca in profondità.
◊ archi in avanti: archi uv con v discendente di u in un albero della foresta
◊ archi trasversali: archi uv in cui v e u appartengono a rami o alberi distinti della foresta.

•ORDINAMENTO TOPOLOGICO

Un ordinamento topologico di un grafo orientato aciclico (DAG) G = (V,E) è un ordinamento lineare dei suoi vertici tale che per ogni arco uv il vertice u precede il vertice v. Per transitività ne consegue che se v è raggiungibile da u allora u comprare prima di v nell'ordinamento.
Per determinare l'ordinamento topologico esistono due soluzioni:

◊ Soluzione diretta: Trovare ogni vertice che non ha alcun arco incidente in ingresso, stampare tale vertice e rimuoverlo insieme ai suoi archi, ripetere la procedura finche tutti i vertici risultano rimossi. Non si tratta di un ordine univoco, possono esistere più ordinamenti dello stesso grafo.

◊ Soluzione basata du DFS: Eseguo una visita in profondità di un grafo e segno i marcatempo di inizio e di fine. Una volta terminata la visita ordino i vari nodi partendo da quello con il marcatempo di fine più alto a quello più basso.

Complessità: O(|V|+|E|) come il DFS

Componenti fortemente connesse (CFC)

La visita in profondità si può usare anche per calcolare le componenti fortemente connesse di un grafo orientato.
—> Una componente fortemente connessa di un grafo orientato G=(V,E) è un insieme massimale di vertici U contenuti in V tale che per ogni u,v appartenenti ad U esiste un cammino da u a v e un cammino da v ad u.

3 fasi:
- si usa la visita in profondità in G per ordinare i vertici in ordine di tempo di completamento f decrescente
- si calcola il grafo trasposto Gt del grafo G
- si esegue una visita in profondità in Gt partendo dal tempo di fine maggiore calcolato nella prima fase.

Il grafo delle componenti fortemente connesse ha: come vertici le componenti fortemente connesse di G; un arco da cfc C ad una cfc C' se e solo se in G vi è un arco che connette un vertice di C ad un vertice di C'.

Albero di connessione minimo

Il problema dell'albero di connessione minimo si riferisce a determinare come interconnettere diversi elementi fra loro minimizzando certi vincoli sulle connessioni.

Bisogna associare un costo ai vari archi presenti.
•Si avrà un albero di copertura (spanning tree) cioè un albero con tutti i nodi del grafo originale che conterrà lo stesso numero di archi uguale al numero di nodi -1.

Un taglio (S,V\S) di un grafo non orientato G=(V,E) è una partizione di V in due sottoinsiemi disgiunti.
Un arco (u,v) attraversa il taglio se u appartiene ad S e v appartiene a V\S.

Algoritmo di Kruskal

Si individua un arco sicuro scegliendo un arco (u,v) di peso minimo tra tutti gli archi che connettono due alberi distinti della foresta.
- parto dall'arco di peso minore e vedo se collega nodi di alberi distinti.
- se collega nodi di alberi distinti allora lo traccio e proseguo con l'arco con il costo minore successivo, proseguendo questo procedimento fino a quando non ho toccato tutti i nodi del grafo.

Complessità: O(|E| log|E|) che in questo caso è anche uguale a O(|E| log|V|)

Algoritmo di Prim

L'algoritmo di Prim procede mantenendo in A un singolo albero da cui scegliamo un vertice per farlo radice r; l'albero crescerà fino a quando non ricoprirà tutti i nodi.
Ad ogni passo viene aggiunto un arco leggero che collega un nodo dell'albero con un nodo non ancora presente nell'albero.

Ogni nodo ha un puntatore al padre che l'ha preceduto
Si hanno i campi colore (bianco,grigio nero)
Ho un campo key che contiene il costo minimo dell'arco che connette un vertice dell'albero con un vertice esterno.

Complessità: O(|V| log|V| + |E| log|V|)

CAMMINI MINIMI DA SORGENTE UNICA

Il costo di un cammino p è la somma dei costi degli archi che lo costituiscono.

Ci sono 4 casi possibili:
- ◊ cammini minimi da un'unica sorgente a tutti gli altri vertici
- ◊ cammini minimi da ogni vertice ad un'unica destinazione
- ◊ cammini minimi da un'unica sorgente ad un'unica destinazione
- ◊ cammini minimi da ogni vertice ad ogni altro vertice

•Archi di peso negativo

La ricerca dei cammini minimi da una sorgente s può avere archi di peso negativo, ma non può avere cicli di costo negativo raggiungibili da s perché ad ogni iterazione del ciclo il peso continua a diminuire arrivando ad una soluzione non definita.

Un cammino minimi non può contenere cicli.
Un cammino minimo in un grafo G=(V,E) può avere al più |V|-1 archi.

Gli algoritmi per il problema dei cammini minimi usano la tecnica di rilassamento: il processo di rilassamento di un arco (u,v) consiste nel verificare se passando per u è possibile migliorare la stima del cammino minimo per v e in caso affermativo nell'aggiornare i valori del costo del percorso d[v] e del predecessore π[v].

Algoritmo di Bellman-Ford

Dato che non conosciamo i cammini minimi e non sappiamo da dove partire, poniamo un nodo senza archi entranti come sorgente s.
Inizialmente nella coda delle distanze e dei predecessori ci saranno tutti i nodi posti a d[v]=∞ π[v]=NILL tranne per il nodo S che avrà d[S]=0.
Una volta avvenuta l'inizializzazione si procede con la scansione di tutti gli archi applicando la tecnica di rilassamento per trovare il cammino minimo.
Saranno necessarie |V|-1 passate.

Complessità: O(|V|^2 |E|)

Algoritmo di Dijkstra

Risolve il problema dei cammini minimi a sorgente una in un G=(V,E) orientato e con pesi sugli archi non negativi.
Il peso e il predecessore di ogni nodo sono scritti e modificati in una tabella che tiene traccia di tutti gli spostamenti. Ha tante colonne quanti sono i vertici del grafo.
Il mio obiettivo è quello di andare a trovare un cammino minimo che mi soddisfi andando ad analizzare tutti i costi degli archi in modo che ogni nodo sia stato controllato.

Complessità: O(n^2) array con n elementi
O(|V|^3 + |V| |E|) se la coda è implementata con array
O(|V| |E| log|V|) se la coda è implementata con min-heap

CAMMINI MINIMI FRA TUTTE LE COPPIE

Gli algoritmi per determinare i cammini minimi tra tutte le coppie di vertici sono basati sulla rappresentazione del grafo tramite matrice di adiacenza composta dal peso che caratterizza l'arco stesso.
Avremo quindi una matrice D=(dij) di dimensione nXn dove dij è il peso del cammino minimi da i a j.
Inoltre si deve tenere traccia del predecessore attraverso una seconda matrice ∏ = (πij) dove πij è nil se non esiste cammino da i a j; o è il predecessore di j in qualche cammino da i.

Algoritmo di Floyd-Warshall

L'algoritmo Floyd-Warshall serve per trovare i cammini minimi tra tutte le coppie di vertici considerando i vertici intermedi di un cammino minimo.
Sia V={1,2…n} e consideriamo il sottoinsieme {1,2…k}, prendendo una coppia qualsiasi di nodi ij se la somma della distanza dik e dkj all'iterazione k è minore della distanza diretta dij all'iterazione k-1allora dik+dkj sarà la nuova distanza minima.

I nodi intermedi sono nodi all'interno di un cammino ad eccezione degli estremi.
Si tratta di una soluzione ricorsiva sulla matrice dij(k) dove k è il numero di visite che bisogna fare fino a quando k=n dove n è il numero di nodi. Quando modifico D(k)=(dij) devo modificare anche ∏(k)=(πij)

Complessità: Θ (n^3)

•Chiusura transitiva di un grafo orientato

Dato un grafo orientato G=(V,E) con l'insieme di vertici V={1..n} la chiusura transitiva di G permette di verificare se esiste un cammino da i a j per tutte le coppie di i,j appartenenti a V.
La chiusura transitiva di G viene definita con il grafo G*=(V,E*) con gli stesi nodi ma con gli archi E*={(i,j): esiste un cammino da i a j in G}

Complessità: Θ (n^3)

PROBLEMA DEL FLUSSO MASSIMO

Reti di flusso: Una rete di flusso è un grafo orientato G=(V,E) ai cui archi è associata una capacità c(u,v)≥0.
In una rete di flusso si distinguono: la sorgente s, e il pozzo t, rispettivamente l'inizio e la fine del grafo.
Ogni nodo è raggiungibile dalla sorgente e tutti i nodi possono raggiungere i pozzo.

Definizione formale di flusso: Sia G=(V,E) una rete di flusso con una funzione capacità c. Sia s la sorgente e t il pozzo.
Un flusso in G è una funzione f:V x V->R che soddisfa le seguenti proprietà:

- vincolo sulla capacità: f(u,v)≤c(u,v) —> il flusso non può superare la capacità.
- antisimmetria: f(u,v)=-f(v,u) —> dice che il flusso f(u,v) da u a v è l'esatto opposto del flusso f(v,u) da v a u.
- conservazione del flusso: per ogni u appartenente a V\{s,t}: la sommatoria v appartenente a V di f(u,v) = 0. Se c'è un flusso esso non si disperde uscendo dalla rete. La somma tra il flusso

uscente da s ed il flusso entrante da t deve essere nulla, perché ciò che entra deve necessariamente uscire.

La quantità f(u,v) può essere positiva, nulla o negativa e viene detta flusso da u a v.
Il valore di un flusso è definito come |f|= sommatoria v appartenente a V di f(s,v) ovvero il flusso totale che esce dalla sorgente s.

Il flusso entrante totale positivo in un nodo u: ∑(v app V) f(v,u)
Il flusso uscente totale positivo in un nodo u: ∑ (v app V) f(u,v)
Il flusso totale netto in un vertice è il flusso totale uscente positivo meno il flusso totale entrante positivo ed è uguale a 0: f(u,v) -f(v,u) =0

Il metodo di Ford-Fulkerson
Si basa su tre importanti idee:
◊ cammini aumentanti
◊ reti residue
◊ tagli

Si parte da f(u,v) =0 per ogni u e v e quindi si aumenta iterativamente il valore del flusso cercando un cammino dalla sorgente al pozzo lungo il quale sia possibile inviare ulteriore flusso.
Il procedimento termina quando non ci sono più cammini aumentanti.
Il cammino aumentante è dunque una iniezione di flusso lungo i vari archi che soddisfi la loro capienza.
Un cammino aumentante è semplicemente un cammino p da s a t nella rete residua. La capacità di un cammino aumentante p è la minima capacita degli archi p nella rete residua.

Una volta iniettato del flusso, la capacità dell'arco non sarà pari a quella che aveva in precedenza ma sarà minore tanto quanto il valore del flusso iniettato. La capacità rimanente è la capacità residua —> cf(u,v) = c(u,v) -f(u,v)

Se la capacità residue non è pari a 0 allora avrò uno sdoppiamento degli archi:
- arco in avanti con la capacità residua
- arco all'indietro con la rete residua —-> in un arco (u,v) la rete residua di v è il valore del flusso che gli ha passato u.

Il flusso è massimo se non ci sono più cammini aumentanti da s a t.
Per dimostrarlo si introduce il taglio, che va a tagliare in due il grafo in maniera tale che ci sia un insieme con la sorgente e un insieme con il pozzo.

Il taglio ha due proprietà:
- capacità del taglio: è la somma di tutte le capacità degli archi appartenenti all'insieme sorgente che vengono tagliati.
- flusso che attraversa il taglio: è la somma di tutte le capacità residue degli archi appartenenti all'insieme sorgente meno le capacità residue degli archi appartenenti ai nodi dell'insieme pozzo che vengono tagliati.

Procedimento algoritmo: mi domando se esiste un cammino che vada da s a t. Ne scelgo uno a caso. Una volta scelto il cammino si aggiunge il cammino aumentante degli archi toccati pari alla capacità minima trovata lungo il percorso.
Una volta aggiunti i cammini aumentanti sdoppio gli archi per vedere la capacità residua e la rete residua. A questo punto scelgo un altro percorso da s a t e svolgo gli stessi procedimenti.

Complessità: O(|E| • k) dove k è il numero di cammini aumentanti

Algoritmo di Edmonds-Karp

Applico una visita in ampiezza del metodo di Ford-Fulkerson per trovare il cammino minimo aumentante.

Complessità: O (|E|^2 |V|)

ABBINAMENTO MASSIMO NEI GRAFI BIPARTITI

Un grafo non orientato G=(V,E) si dice bipartito se i suoi vertici si possono ripartire in due sottoinsiemi S e D tali che ogni arco abbia estremi appartenenti a sottoinsiemi distinti.
Se xy è un arco diciamo che x e y si possono accoppiare
L'obiettivo è quello di trovare un insieme massimo di coppie distinte.

STRUTTURE DATI PER INSIEMI DISGIUNTI

Servono a mantenere una collezione S={S1,S2…Sk} di insiemi disgiunti.
Ogni insieme della collezione è individuato da un rappresentante che è un particolare elemento dell'insieme.
Operazioni:
Make-Set(x): aggiunge alla struttura dati un nuovo insieme contenente solo l'elemento x
Find-Set(x): ritorna il rappresentante dell'insieme che contiene x
Union(x,y): riunisce i due insiemi contenti x e y in un unico insieme.

-Rappresentazione con liste

Il modo più semplice di rappresentare una famiglia di insiemi disgiunti è usare una lista circolare per ogni insieme.
Ogni nodo ha un puntatore al nodo successivo ed un puntatore al nodo rappresentante.

I nodi hanno i campi:
- info: informazioni contenuta nel nodo
- rappr: il puntatore al rappresentante
- succ: il puntatore al successore

•Euristica dell'unione pesata

Consiste nel scegliere sempre la lista più corta per aggiornare i puntatori al rappresentante.
Per fare ciò occorre avere un campo lung nel rappresentate (che rimpiazza il campo rappr perché inutile nel rappresentate) che mi dice la lunghezza della lista. Infine per distinguere tutti gli altri nodi dal rappresentante si aggiunge ad ogni nodo un campo booleano rp.
Complessità Union: O(n2) n2 lunghezza della seconda lista.

-Rappresentazione con foreste

Una rappresentazione più efficiente si ottiene usando foreste di insiemi disgiunti dove ogni insieme viene rappresentato con un albero i cui nodi, oltre al campo info, hanno soltanto un campo parent che punta al padre.

•Euristica dell'unione per rango

Per ogni nodo x manteniamo un campo rank che è un limite superiore all'altezza del sottoalbero di radice x, ovvero il numero di archi nel cammino più lungo fra x e una foglia discendente.
Se dobbiamo unire due alberi, l'operazione union mette la radice dell'albero con rango minore come figlio de quella di rango maggiore.

•Euristica della compressione dei cammini

Quando effettuiamo una find-set(x) dobbiamo attraversare il cammino da x a radice, quindi possiamo approfittarne per far puntare alla radice dell'albero i puntatori al padre di tutti i nodi incontrati lungo il cammino. In questo modo le successive operazioni di find-set(x) suoi nodi di tale cammino risulteranno meno costose.

Ricorsione

E' ricorsivo quello che può essere definito in termini di se stesso. es. fattoriale, numeri di fibonacci…

•Si parla di ricorsione diretta quando una proceduta è espressa in termini di se stessa.

•Si parla di ricorsione indiretta quando una procedura p è espressa in termini di una procedura a sua volta definita direttamente o indirettamente in termini di p.

Una routine ricorsiva è costituita da: - base della ricorsione - passo ricorsivo

L'introduzione della ricorsione introduce il problema potenziale di ricorsioni infinite. In pratica, mentre un loop infinito effettivamente non ha mai termine, la ricorsione infinita prima o poi determina l'esaurimento della memoria disponibile per lo stack del processo che la esegue e la distruzione di quest'ultimo.

La ricorsione semplifica la struttura del programma però:
- in alcuni linguaggi le chiamate di procedura sono più costose delle operazioni di assegnamento.
- la ricorsione implica molto utilizzo di memoria.

Ogni procedura ricorsiva può essere tradotta in una procedura equivalente iterativa.

Algoritmo intrinsecamente ricorso: è quell'algoritmo la cui procedura iterativa deve necessariamente creare una pila in cui tenere traccia del valori delle variabili locali che l'algoritmo utilizza.

E' più conveniente utilizzare l'iterazione piuttosto della ricorsione

Tail Recursion: è una semplice tecnica per eliminare la ricorsione.
Una procedura p presenta tail recursion quando:
- ha un parametro x passato per valore;
- la sua ultima istruzione è una chiamata di p ricorsiva con valore y del parametro di chiamata

Si può eliminare la chiamata ricorsiva sostituendola con un assegnamento di y a x ed un goto alla prima istruzione della procedura.
- Eliminazione chiamate terminali: ci interessa il record di attivazione di q (nel caso in cui la procedura p era gia stata chiamata allora mettiamo il record di attivazione di p in q)
- Eliminazione chiamate terminali ricorsive: Si riscrive la procedure/funzione in modo che il ramo then sia quello che contiene la chiamata ricorsiva di coda ed il ramo else quello che contiene la base della ricorsione. La chiamata ricorsiva si può trasformare in un ciclo come segue:
  - l'istruzione if-then-else diventa una istruzione while
  - il test di if-then-else diventa il test del ciclo while
  - la sequenza di istruzioni del ramo then che precedono la chiamata ricorsiva diventa la prima parte del corpo del ciclo while
  - la chiamata ricorsiva si trasforma in un insieme di assegnamenti che in genere faranno uso di variabili temporanee; questi assegnamenti sono la parte finale del corpo del while.
  - le istruzioni del caso base diventano le istruzioni da eseguire all'uscita del ciclo while.