Programmazione Algoritmi e Computabilità

2024-2025

Introduzione

Il corso si prefigge l'obiettivo di fornire conoscenze, sia teoriche che pratiche, per la progettazione e implementazione di un'applicazione *software*, mediante un processo di sviluppo *agile* e *API-led*. Si punterà, su piccola scala, al design di algoritmi efficienti e computabili, con le relative strutture dati. Questo primo obiettivo sarà raggiunto studiando la teoria della complessità e della computabilità algoritmica. Su una scala maggiore, si studierà l'architettura *software*. Questo secondo obiettivo sarà raggiunto impiegando il linguaggio Java, abbinato a *middleware*, *framework* e librerie, insieme a *tool* legati all'IDE Eclipse. # Complessità degli algoritmi

Riprendiamo i concetti di analisi matematica, come la notazione asintotica, per stabilire l'appartenenza del tempo di esecuzione alle classi di complessità. La funzione $f(n)$ rappresenta il tempo di esecuzione o l'occupazione di risorse di un algoritmo su un input di dimensione n. Utilizzeremo questa funzione come astrazione in informatica per nascondere complessità irrilevanti.

La notazione O ("o grande") rappresenta un limite superiore, mentre Ω rappresenta un limite inferiore. La notazione Θ indica limiti sia superiori che inferiori esatti. Le prime due notazioni sono generalmente più facili da trovare.

La notazione O è la più comunemente definita. Una funzione $f(n)$ appartiene alla classe $O(g(n))$ se esistono due costanti $c > 0$ e $n_0 > 0$ tali che $f(n) \leq cg(n)$ per ogni $n \geq n_0$. In altre parole, a partire da n_0, la curva analizzata sta sempre al di sotto di $cg(n)$.

La notazione Ω è utilizzata per indicare un limite inferiore. Una funzione $f(n)$ appartiene alla classe $\Omega(g(n))$ se esistono due costanti $c > 0$e $n_0 \geq 0$ tali che $f(n) \geq cg(n)$ per ogni $n \geq n_0$.

La notazione Θ è utilizzata per indicare limiti sia superiori che inferiori esatti. Una funzione $f(n)$ appartiene alla classe $\Theta(g(n))$ se esistono tre costanti \$c_1, c_2 > 0 \$e $n_0 \geq 0$ tali che $c_1 g(n) \leq f(n) \leq c_2 g(n)$ per ogni $n \geq n_0$.

È importante notare che stiamo facendo un abuso di notazione. In realtà, dovremmo usare il simbolo $\in$ invece di $=$ per indicare l'appartenenza alle classi di complessità. Ad esempio, dovremmo scrivere $f(n) \in O(g(n))$ per indicare che O è un limite superiore asintotico per la funzione appartenente alla sua classe.

Teorema: date due funzioni $f(n)$ e $g(n)$, $f(n) \in \Theta(g(n))$ se e solo se $f(n =\in O(g(n))$ e $f(n) = \Theta(g(n))$. (ricontrollare dalle slide)

Ad esempio, $O(n^2)$ e $\Omega(n^2)$ sono limiti inferiori e superiori per qualsiasi polinomio quadratico, aggiustando adeguatamente i parametri del polinomio. Vale anche $\Theta(n^2)$. Un polinomio quadratico è anche $O(n^3)$ ma non $\Theta(n^3)$. Anche quando calcoliamo O, ci conviene stare vicini al sup di Θ.

Proprietà: - *transitiva*: vale per tutte e tre le notazioni. Si dimostra immediatamente applicando la definizione - *riflessiva*: vale per tutte e tre. Ogni funzione è limite inferiore e superiore per sé stessa. - *simmetria*: vale solo per Θ. Se una funzione è nello stesso ordine di un'altra, allora l'altra è nello stesso ordine della prima. - per le altre due vale la simmetria transposta. Il limite superiore per una diventa limite inferiore per l'altra.

Le notazioni di O, Θ e Ω sembrano simili agli ordinamenti totali dei numeri. Non vale però la tricotomia (cioè che debba valere per forza minore, uguale o maggiore). Le funzioni oscillanti rendono difficile il confronto asintotico.

Regole di semplificazione: 1. posso ignorare le costanti moltiplicative nell'ordine 2. la somma di due funzioni avrà O somma tra le O dei due membri (casi di somma: sequenze, *branching*. Per il branching si conta il costo del ramo più lento. Per le *subroutine*, si somma il loro costo) 3. il prodotto di funzioni avrà O pari al prodotto degli O dei membri (si applica ai cicli, in base all'O del numero di iterazioni e all'O del corpo)

In caso di dubbi sull'ordine, si applica il teorema dei limiti. Una funzione è dello stesso O di un'altra se il limite all'infinito del loro rapporto è zero. Notare che non vale l'inverso. L'inverso parziale è che il limite del loro rapporto sia finito se esiste. Se il rapporto è finito, allora il numeratore è Ω del denominatore. Non vale l'inverso, posso solo dire che se esiste allora è una quantità positiva. Se il limite del rapporto è finito positivo, allora vale Θ. Se il limite non esiste, non sono asintoticamente confrontabili.

Limiti temporali

Un problema computazionale ha complessità $O(f(n))$ *(upper bound)* se esiste un algoritmo per la sua risoluzione con complessità $O(f(n))$. Un problema computazionale ha complessità $\Omega(f(n))$ *(lower bound)* se ogni algoritmo per la sua risoluzione ha complessità $\Omega(f(n))$. Riuscendo a dimostrare che un problema ha delimitazione inferiore $\Omega(f(n))$, e trovando un algoritmo che ha delimitazione superiore $O(f(n))$, allora si ottiene, a meno di costanti, un algoritmo risolutivo ottimale.

Per la ricerca del minimo in insiemi non ordinati di n elementi la complessità inferiore è $\Omega(n)$ perché è necessario scandire almeno una volta l'intero insieme. Un algoritmo $O(n)$ è dunque ottimo. Il *mergesort* ha complessità $n \log n$ per un problema $n \log n$ ed è dunque ottimale. Il *bubblesort* è subottimale perché ha complessità n^2.

La dimensione dell'*input* non è il solo criterio in base al quale valutare l'esecuzione degli algoritmi. Distinguiamo caso migliore, peggiore, medio. Dato tempo(I) il tempo di esecuzione dell'algoritmo sull'istanza I, abbiamo

$$T_{worst}(n) = \max_{\text{istanze } I \text{ di dimensione } n} \{\text{tempo}(I)\}$$

$$T_{best}(n) = \max_{\text{istanze } I \text{ di dimensione } n} \{\text{tempo}(I)\}$$

$$T_{avg}(n) = \sum_{\text{istanze } I \text{ di dimensione } n} \{\mathcal{P}(I)\ \text{tempo}(I)\}$$

dove $\mathcal{P}(I)$ è la distribuzione di probabilità dei casi.

Per la ricerca sequenziale (cerca lungo l'intera lista, fermati alla prima istanza trovata), il tempo migliore è 1 (elemento cercato in testa), il peggiore è n (elemento in ultima posizione), media $(n+1)/2$. Questo vale se assumiamo che le istanze siano equidistribuite e che la probabilità di ogni elemento sia $1/n$. Applicando la formula di prima esce la probabilità del caso medio (serie aritmetica).

Spesso il calcolo del caso medio è molto complesso e, in molti casi, si scopre che il caso medio è molto vicino al caso peggiore. Pertanto, è spesso conveniente determinare il limite superiore nel caso peggiore.

La ricerca binaria per liste ordinate si basa su tre indici: inferiore, medio, superiore. Ad ogni passo si confronta l'elemento all'indice medio con il *target* della ricerca. In caso di differenza, restringiamo la ricerca alla metà corrispondente di lista. In un numero finito di passi l'algoritmo collassa su di un solo elemento. Se esso non è quello cercato, la ricerca fallisce. In alcuni casi può capitare che gli indici si incrocino, e anche in questo caso si può dedurre che l'elemento cercato non sia presente in lista. Il tempo migliore è costante (l'elemento centrale è quello cercato, e viene trovato subito): $\Theta(1)$. Il tempo peggiore si ha quando l'elemento cercato è l'ultimo considerato, o non c'è. Dato che si procede per dimezzamenti della lista, il numero di passi è pari $\log_2 n$ del numero di elementi. Il caso medio è $\log n - 1 + \frac{1}{n}$, simile al caso peggiore.

Strutture dati

Si analizzeranno ora le strutture dati nell'ottica dell'analisi degli algoritmi. Si introduca innanzitutto il concetto di *Abstract Data Type*. Esso risponde alla domanda *"che cosa?"*, nel senso che definisce la semantica, le operazioni correlate, gli ingressi, le uscite e i vincoli legati al tipo di dati in questione. La struttura dati vera e propria, invece, è una concretizzazione dell'ADT, e risponde alla domanda *"come?"*, fornendo i dettagli implementativi. Nella programmazione a oggetti, gli ADT corrispondono alle interfacce, mentre i tipi concreti corrispondono alle classi. Per fornire un esempio, definiamo astrattamente un dizionario. I dati sono un insieme di coppie chiave / valore. Le operazioni consentite sono l'inserimento (aggiunta di una nuova coppia all'insieme), la cancellazione (rimozione di una coppia dall'insieme data la sua chiave) e ricerca (recuperare una coppia dall'insieme data la chiave). Volendo essere più precisi, potremmo anche definire il tipo di ritorno di ogni operazione, compreso il valore restituito in caso di errori. In Java, la definizione prende la forma seguente:

```
public interface Dizionario {
    public void insert(Object e, Comparable k);
    public void delete(Comparable k);
    public Object search(Comparable k);
}
```

Questa implementazione è non generica (usiamo `Object` e la versione non generica di `Comparable`). Volendo sfruttare i tipi generici, invece:

```
public interface Dizionario <E, K extends Comparable <? Super k >> {
    public void insert(E e, K k);
    public void delete(K k);
```

```
    public E search(K k);
}
```

Rappresentazioni indicizzate e collegate

Le tecniche di rappresentazione dei dati possono essere divise in due categorie: *rappresentazioni indicizzate* e *rappresentazioni collegate*. Il primo raggruppamento memorizza i dati in aree contigue di memoria, come in un *array*. Questo permette di accedere direttamente ai dati tramite un indice. Di contro, però, la dimensione è fissa, causando spreco di spazio dovuto alla frammentazione interna, da mitigare con riallocazioni frequenti che richiedono un tempo lineare. Il secondo tipo utilizza invece *record* spazialmente separati, ma collegati logicamente da puntatori. L'effettiva compattezza e lo spazio occupato dipendono dalla specifica implementazione. Esempi di struttura collegata sono la lista semplice, la lista doppiamente collegata e la lista circolare doppiamente collegata. Le rappresentazioni collegate hanno dimensioni variabili, veramente dinamiche. Le aggiunte e le rimozioni sono effettuate a tempo costante. L'accesso dei dati invece richiede un tempo medio lineare, perché è necessariamente sequenziale.

Riprendiamo l'esempio relativo all'implementazione del dizionario. Volendolo costruire come struttura dati collegata, possiamo basarlo sull'array. L'inserimento di una nuova coppia richiede la riallocazione dell'array. Si alloca un nuovo spazio di memoria con lo spazio per un elemento in più. Successivamente lo si riempie copiando il contenuto della versione precedente dell'array finché non si raggiunge l'elemento precedente, in ordine di chiave, rispetto all'elemento da inserire. Si inserisce infine l'elemento nuovo, e dopo di esso tutti gli elementi preesistenti che lo seguono in ordine di chiave. La cancellazione richiede di trovare prima l'indice dell'elemento da cancellare, e in seguito di riallocare l'array con un posto in meno, saltando nella copia l'elemento da scartare. Entrambe le operazioni di inserimento e cancellazione hanno tempo lineare. Dato che l'array è ordinato, la ricerca può essere effettuata in tempo logaritmico sfruttando l'algoritmo di ricerca binaria. Un'implementazione collegata del dizionario può essere generata utilizzando una lista collegata. Le nuove coppie si inseriscono in testa, a tempo costante. L'operazione, infatti, richiede l'aggiornamento di un puntatore (quello della testa, se la lista è vuota) o due puntatori (testa e riferimento al successivo, se la lista non è vuota). L'eliminazione è a tempo lineare perché richiede di scandire sequenzialmente la lista per trovare l'elemento desiderato, e di riscrivere i puntatori per collegare tra loro gli elementi precedente e successivo ad esso. Anche la ricerca è sequenziale e dunque a tempo lineare.

Definiamo la struttura dati astratta per una pila. Si tratta di una semplice sequenza di elementi, e la particolarità sta nella politica d'accesso LIFO (*Last In, First Out*). Possiamo definire operazioni per verificare se sia vuota o meno, per inserire ed eliminare elementi dalla cima, o per osservare tale cima in sola lettura. Anche la pila, come il dizionario, può essere implementata come struttura indicizzata o collegata. Una possibile implementazione indicizzata in Java prende la forma seguente:

```
public interface Pila {
    private int maxSize;
    private int top;
    private Object[] listArray;
}

class LPila implements Pila {
    private Record top;
    private int size;
    ...
}
```

La struttura dati astratta di una coda deve permettere due operazioni (*enqueue*, "accoda", e *dequeue*, "rimuovi"), e di accedere in sola lettura all'estremo di interesse. La coda può anche essere bilaterale (nota come *deque*, ovvero *Double-Ended QUEue*), e permettere accodamento e scodamento di elementi da entrambi gli estremi. L'implementazione indicizzata di una coda può essere effettuata tenendo due indici, *front* e *rear*, per il primo e ultimo elemento della coda all'interno del vettore. Dato che inserimenti e cancellazioni possono lasciare spazi vuoi sia in testa che in coda, conviene interpretare l'array come una struttura circolare e leggerlo "a orologio" lungo un verso stabilito. Questo riduce la necessità di riallocazioni dovute al ricompattamento dei dati. A livello implementativo l'effetto può essere ottenuto usando l'aritmetica modulare nel calcolo degli indici. In questa versione circolare è possibile distinguere tra coda piena e vuota soltanto se è presente almeno uno spazio vuoto tra il primo e l'ultimo elemento. Non è possibile infatti dedurre dagli indici *front* e *rear* se l'array sia completamente pieno o completamente vuoto, perché in entrambi i casi la loro posizione si sovrappone. L'implementazione collegata per le code e le pile è molto semplice da gestire. Tutte le operazioni sono a tempo costante perché si opera sempre e solo sugli elementi in testa (o eventualmente in coda). ## Alberi

Un albero radicato è una coppia $T = (N, A)$ costituita da un insieme N di *nodi* e da un insieme A di coppie di nodi, dette *archi*. I dati sono contenuti nei nodi. Le relazioni gerarchiche tra dati sono rappresentate dagli archi che li collegano. Chiamiamo *grado* di un nodo il suo numero di figli. Chiamiamo *foglie* i nodi senza figli. Chiamiamo *cammino* la sequenza di nodi che collega un nodo antenato ad un nodo discendente tramite sole relazioni padre-figlio. La *lunghezza* di un cammino corrisponde al numero di archi attraversati. Il cammino tra la radice ed un qualsiasi nodo è univoco. La lunghezza di tale cammino è detta *profondità* del nodo. L'*altezza* dell'albero è la lunghezza del cammino più lungo possibile al suo interno. Un albero $k-ario$ completo è un albero in cui tutte le foglie hanno la stessa profondità e tutti i nodi interni hanno grado k. Esso ha k^h foglie a profondità h, e $\frac{k^h-1}{k-1}$ nodi interni. Un albero binario completo ha dunque $2^h - 1$ nodi interni e 2^h foglie.

Rappresentiamo il tipo di dato astratto per un albero:

```
tipo Albero:
dati:
    un insieme di nodi (di tipo nodo) e un insieme di archi
operazioni:
    numNodi() -> intero
        restituisce il numero di nodi presenti nell'albero
    grado(nodo v) -> intero
        restituisce il numero di figli del nodo v
    padre(nodo v) -> nodo
        restituisce il padre del nodo v dell'albero, o null se
        v è la radice
    figli(nodo v) -> (nodo, nodo, ..., nodo)
        restituisce, uno dopo l'altro, i figli del nodo v
    aggiungiNodo(nodo u) -> nodo
        inserisce un nuovo nodo v come figlio di u nell'albero
        e lo restituisce. Se v è il primo nodo ad essere
        inserito nell'albero, esso diventa la radice (e u viene
        ignorato).
    aggiungiSottoalbero(Albero a, nodo u)
        inserisce nell'albero il sottoalbero a in modo che la
        radice di a diventi figlia di u.
    rimuoviSottoalbero(nodo v) -> Albero
        stacca e restituisce l'intero sottoalbero radicato in
        v. L'operazione cancella dall'albero il nodo v e tutti
        i suoi discendenti.
```

Le rappresentazioni concrete degli alberi sono sempre collegate. Per alberi nei quali ogni nodo abbia un numero limitato di figli, possiamo definire un numero limitato di puntatori ai figli. Quando il numero di figli necessita di essere arbitrario, si possono invece tenere un puntatore al primo figlio e un puntatore al fratello successivo.

Gli alberi più usati sono gli alberi binari. Di essi possiamo dare una definizione ricorsiva. L'insieme vuoto $\emptyset$ è un albero binario. Se T_s e T_d sono alberi binari ed r è un *nodo* allora la terna ordinata (r, T_s, T_d) è un albero binario. In memoria ogni nodo ha un puntatore al padre e due puntatori ai figli. Gli alberi binari si possono utilizzare, tra le altre cose, per rappresentare espressioni matematiche e per costruire classificatori. Definiamo *algoritmi di visita* gli algoritmi che consentono l'accesso sistematico ai nodi e agli archi di un albero. Gli algoritmi di visita si distinguono in base all'ordine di accesso ai nodi. Analizziamo innanzitutto un algoritmo di visita generico:

```
algoritmo visitaGenerica(nodo r)
    S <- {r}
    while (S != vuoto) do
        estrai un nodo u da S
        visita il nodo u
        S <- S unito {figli di u}
```

Esso richiede tempo $O(n)$ per visitare un albero con n nodi a partire dalla radice. Un possibile algoritmo non generico è la visita in profondità (DF), che parte dalla radice e visita i nodi di figlio in figlio fino al raggiungimento di una foglia. Risale poi al primo antenato con figli non visitati, e ripete la visita verso le foglie a partire da quei figli. L'algoritmo ha tre varianti. La prima, detta detta *preordine* o *ordine anticipato*, visita prima il nodo, poi i sottoalberi sinistro e destro. La seconda, detta *inordine* o *ordine simmetrico*, visita prima il sottoalbero sinistro, poi il nodo base, poi il sottoalbero destro. L'ultima, detta *postordine* o *ordine posticipato*, visita prima il sottoalbero sinistro, poi il destro, e infine il nodo.

```
// preordine iterativo
```

```
algoritmo visitaDFS(nodo r)
    Pila S
    S.push(r)
    while(not S.isEmpty()) do
        u <- S.pop()
        if(u != null) then
            visita il nodo u
            S.push(figlio destro di u)
            S.push(figlio sinistro di u)

// preordine ricorsivo
algoritmo visitaDFSRicorsiva(nodo r)
    if(r = null) then return
    visita il nodo r
    visitaDFSRicorsiva(figlio sinistro di r)
    visitaDFSRicorsiva(figlio destro di r)
```

Un algoritmo alternativo è la *visita in ampiezza* (BFS). Partendo dalla radice, l'algoritmo procede visitando i nodi per livelli successivi. Un nodo ad un dato livello è visitabile solo se tutti i nodi del livello superiore sono stati visitati.

```
// versione iterativa
algoritmo visitaBFS(nodo r)
    Coda C
    C.enqueue(r)
    while(not C.isEmpty()) do
        u <- C.dequeue()
        if(u != null) then
            visita il nodo u
            C.enqueue(figlio sinistro di u)
            C.enqueue(figlio destro di u)
```

Alberi di ricerca

Un *albero binario di ricerca* (*binary search tree*, BST) è un albero binario che soddisfa le seguenti proprietà:

1. ogni nodo v contiene un elemento $elem(v)$ cui è associata una $chiave(v)$ presa da un dominio totalmente ordinato
2. le chiavi nel sottoalbero sinistro di v sono minori o uguali alla $chiave(v)$
3. le chiavi nel sottoalbero destro di v sono maggiori o uguali alla $chiave(v)$

Gli alberi binari di ricerca sono utilizzati per implementare con efficienza i *dizionari*, definiti come collezioni di coppie chiave/valore. In questo caso, si chiede che le chiavi da ordinare tramite l'albero siano univoche. In casi d'uso che non richiedano chiavi univoche, possono esserci varie altre regole per ordinare gli elementi.

Su un albero binario generico il costo della ricerca è $O(n)$ indipendentemente dalla strategia applicata, che sia in profondità o in ampiezza. Se l'albero è di ricerca, possiamo partire dalla radice e decidere in che verso muoverci sfruttando l'ordinamento delle chiavi. L'algoritmo di ricerca, formalmente, è:

```
algoritmo search(chiave k) -> elem
    v <- radice di T
    while (v != null) do
        if (k = chiave(v)) then return elem(v)
        else if (k < chiave(v)) then v <- figlio sinistro di v
        else v <- figlio destro di v
    return null
```

Il caso peggiore si verifica quando la chiave desiderata corrisponde ad una foglia a massima profondità, o quando essa è introvabile. In tal caso, il numero di passi è pari all'altezza dell'albero, e la complessità è $O(h)$, dove h è l'altezza dell'albero. Possiamo mettere in relazione l'altezza h dell'albero con il suo numero n di elementi, per ottenere una stima di complessità in funzione di quest'ultimo. Il caso peggiore corrisponde alla ricerca sull'albero degenere linearizzato, nel quale ogni elemento ha un solo figlio. In questo caso l'albero degenera in una lista collegata con $h = n - 1$ elementi, e la ricerca ha dunque complessità $O(n)$ perché richiede la scansione lineare. L'altezza è $h = \Theta(n)$. Il caso migliore è un albero completo bilanciato, in cui ogni elemento ha entrambi i figli,

tranne le foglie, e in cui i sottoalberi destro e sinistro di ogni elemento hanno lo stesso numero di membri. In tal caso, l'altezza dell'albero è $h = \Theta(\log(n))$.

Valendo l'ordinamento gerarchico delle chiavi, gli elementi di chiave minima e massima saranno rispettivamente il più a sinistra e il più a destra dell'albero. L'algoritmo per trovare il minimo a partire dalla radice deve procedere verso sinistra fino a una foglia. L'algoritmo per trovare il massimo a partire dalla radice deve procedere verso destra fino a una foglia.

```
algoritmo max(nodo u) -> nodo
    v <- u
    while(figlio destro di v != null) do
        v <- figlio destro di v
    return v

algoritmo min(nodo u) -> nodo
    v <- u
    while(figlio sinistro di v != null) do
        v <- figlio sinistro di v
    return v
```

Chiamiamo *predecessore* di un nodo u in un BST il nodo v nell'albero di chiave massima minore o uguale a $chiave(u)$. Chiamiamo *successore* di un nodo u in un BST il nodo v nell'albero di chiave minima maggiore o uguale a $chiave(u)$. Algoritmicamente, il successore è il minimo del sottoalbero destro, se esiste. Se il sottoalbero destro non esiste, si risale l'albero fino ad un antenato che sia figlio sinistro del proprio nodo padre. Tale nodo padre sarà l'elemento sul cui sottoalbero destro sarà possibile trovare il successore dell'elemento desiderato. Il predecessore è il massimo del sottoalbero sinistro, se esiste. Se il sottoalbero sinistro non esiste, si risale lungo l'albero fino a trovare un antenato che sia figlio destro del proprio nodo padre, e si cerca il predecessore dell'elemento desiderato nel sottoalbero sinistro dell'antenato. Questo algoritmo richiede ovviamente di salvare un riferimento al padre in ogni nodo dell'albero.

```
algoritmo pred(nodo u) -> nodo
    if (u ha un figlio sinistro sin(u)) then
        return max(sin(u))
    while (parent(u) != null e u è figlio sinistro di suo padre) do
        u <- parent(u)
    return parent(u)
```

L'inserimento in un albero binario di ricerca va effettuato mantenendo la proprietà di ricerca. L'operazione di inserimento è inizialmente simile a quella di ricerca, perché risulta necessario trovare il punto dove inserire ordinatamente il nuovo elemento. Questi è sempre aggiunto come foglia, nel modo meno invasivo possibile. Anche l'inserimento, come la ricerca, ha complessità $O(h)$. La visita in profondità in ordine simmetrico riflette l'ordinamento delle chiavi.

Cancellare mantenendo la proprietà di ricerca è più complesso. Se l'elemento da eliminare è una foglia, può essere rimosso direttamente senza bisogno di altre considerazioni. Se l'elemento da eliminare ha un solo figlio, attacchiamo il figlio dell'elemento da eliminare al padre dell'elemento da eliminare, al posto dell'eliminato. Se l'elemento da eliminare ha due figli, deve essere sostituito con il suo predecessore, e quest'ultimo va poi eliminato secondo le regole dei casi precedenti. Questa procedura funziona perché, per definizione, il predecessore non ha figli destri. In questo modo, la sua eliminazione ricade necessariamente in uno dei primi due casi. La procedura può essere effettuata anche utilizzando l'elemento successore al posto del predecessore.

L'efficienza $O(h)$, come spiegato in precedenza si traduce, in termini di n, in $O(n)$ nel caso peggiore per alberi molto sbilanciati, e in $O(\log(n))$ nel caso peggiore per alberi bilanciati. A tal fine, definiamo il *fattore di bilanciamento* $\beta(v) = |h(\text{sinistro}(v)) - h(\text{destro}(v))|$. Definiamo un albero *bilanciato in altezza* se $\forall v$ vale $\beta(v) \leq 1$, *completo* se $\forall v\ \beta(v) = 0$ e degenerato in lista se $\beta(v) = h$. Esiste una categoria di alberi, chiamata AVL, che comprende alberi binari bilanciati in altezza, che si auto-aggiustano per rotazione. La rotazione consiste nello spostare la posizione gerarchica di nodi senza però alterare gli archi. Ad esempio, un nodo figlio con un solo figlio può essere promosso a nodo con due figli (il figlio originale e il padre) alzandolo di un livello, e abbassando il padre. La rotazione ha costo $O(1)$. L'inserimento con rotazione richiede di calcolare i fattori di bilanciamento lungo il percorso tra la radice e il nuovo nodo inserito. Successivamente, in caso di criticità solitamente riconoscibili da uno sbilanciamento sopra la soglia di 2, si effettua una singola rotazione nel verso opportuno. Strategie di bilanciamento alternative alla rotazione comprendono lo spostamento e la fusione. Gli alberi rosso-neri utilizzano entrambe queste strategie.

B-Alberi

I B-alberi sono alberi bilanciati utilizzati per memorizzare grandi quantità di dati su disco. Sono particolarmente efficaci per le basi di dati, poiché sono progettati per minimizzare il numero di accessi necessari. Le operazioni fondamentali che possono essere eseguite su un B-albero includono l'inserimento, la cancellazione e la ricerca di dati. Inoltre, per mantenere l'albero bilanciato, vengono eseguite operazioni di bilanciamento come la divisione e l'unione dei nodi.

Ogni nodo di un B-albero contiene più chiavi. I nodi possono contenere fino a n chiavi e avere $n+1$ figli. Una lettura su disco $DiskRead(x)$ o una scrittura $DiskWrite(x)$ richiede un tempo di accesso proporzionale alla quantità di dati, nell'ordine dei millisecondi, il che è molto più lento rispetto a un'operazione su CPU. Utilizzare un albero molto ramificato permette di ridurre l'altezza dell'albero e, di conseguenza, il numero di accessi alla memoria. Ad esempio, se $n + 1 = 1000$, è possibile contenere $10^9 - 1$ chiavi in un albero di altezza 2.

Un B-albero T è un albero con radice $root[T]$ che soddisfa le seguenti proprietà:

1. Ogni nodo x contiene vari campi, tra cui:
   - $n[x]$: il numero di chiavi presenti nel nodo.
   - $n[x] + 1$: il numero di figli del nodo.
   - Altri campi specifici che possono essere recuperati dalle slide.
2. Se il nodo non è una foglia, contiene anche i puntatori ai suoi figli.
3. Le $n(x)$ chiavi di un nodo interno separano gli intervalli contenenti le chiavi dei sottoalberi, rispettando la proprietà degli intervalli.
4. Tutte le foglie si trovano allo stesso livello h, corrispondente all'altezza dell'albero.
5. Il numero di chiavi in un nodo è limitato da una costante t, detta grado minimo dell'albero:
   - Ogni nodo, eccetto la radice, ha almeno $t - 1$ chiavi e t figli:
   $$n[x] \geq t - 1$$
   .
   - Se l'albero non è vuoto, la radice contiene almeno una chiave; se la radice non è una foglia, ha almeno due figli.
   - Un nodo può contenere al massimo $2t-1$ chiavi, nel qual caso è considerato pieno e deve essere suddiviso.

Indicando con k_j una qualsiasi chiave memorizzata nel sottoalbero $C_j[x]$, la proprietà dei sottointervalli ci dice che, per $j = 1, \ldots, n[x] + 1$:

$$k_j \leq key_i[x] \leq k_{i+1}, \quad i = 1, \ldots, n[x]$$

I B-alberi minimi, con $t = 2$, sono noti come alberi 2-3-4.

Ogni B-albero con grado minimo t che contiene N chiavi avrà un'altezza h che sarà al massimo $\log_t \frac{N+1}{2}$, assumendo per convenzione $h = -1$ per gli alberi vuoti.

Dimostrazione:

- Passiamo alla versione esponenziale della disuguaglianza: $h \leq \log_t \frac{N+1}{2} \rightarrow N \geq 2t^h - 1$.
- Se l'albero è vuoto, $h = -1$ e $N = 0 \geq 2t^{-1} - 1$ (caso base).
- Supponiamo che l'albero non sia vuoto, con radice r e $h \geq 0$. Sia m_i il numero di nodi al livello i-esimo. Allora:
  - $m_0 = 1$
  - $m_1 = n[root] + 1 \geq 2$ (figli della radice che ha almeno una chiave)
  - $m_i \geq tm_{i-1}$ per $i > 1$, quindi $m_i \geq t^{i-1}m_1 \geq 2t^{i-1}$ (ogni nodo ha almeno t figli)

Quindi, per le chiavi:

$$N = \sum_x n[x]$$

$$= n[root] + \sum_{i=1}^{h} \sum_{x \ di \ livello \ i} n[x]$$

$$\geq 1 + \sum_{i=1}^{h} 2t^{i-1}(t-1)$$

(che è una serie geometrica)

$$= 1 + 2(t-1)\frac{t^h - 1}{t-1}$$

$$= 1 + 2(t^h - 1) = 2t^h - 1.$$

L'altezza di un B-albero è $O(\log_t N)$, dello stesso ordine di grandezza di $O(\log_2 N)$ degli alberi binari, ma per $50 \leq t \leq 2000$ la notazione nasconde un fattore di riduzione compreso tra 5 e 11.

Operazioni Elementari e Convenzioni

Le operazioni elementari su un B-albero seguono alcune convenzioni fondamentali:

- la radice dell'albero è sempre mantenuta in memoria;
- i nodi passati come parametri alle funzioni sono sempre letti dalla memoria.

Operazioni Definite

Le operazioni principali definite per un B-albero includono:

- costruttore di un albero vuoto (`BTree`): inizializza un B-albero vuoto
- `search`: cerca una chiave specifica all'interno dell'albero
- `insert`: inserisce una nuova chiave nell'albero
- `delete`: rimuove una chiave esistente dall'albero

Procedure Ausiliarie

Per supportare le operazioni principali, vengono utilizzate tre procedure ausiliarie:

- `SearchSubtree`: Cerca una chiave all'interno di un sottoalbero specifico.
- `SplitChild`: Divide un nodo figlio pieno in due nodi, ridistribuendo le chiavi e aggiornando i puntatori ai figli.
- `InsertNonfull`: Inserisce una chiave in un nodo che non è pieno, mantenendo l'albero bilanciato.

Queste procedure ausiliarie sono essenziali per garantire che l'albero rimanga bilanciato e che le operazioni di inserimento e cancellazione siano efficienti.

Implementazioni e complessità

Il costruttore `BTree`:

```
BTree(t)
    root[T] <- nil
```

ha complessità $O(1)$.

L'operazione di ricerca `Search`:

```
Search(T,k)
    if root[T] = nul then return nil
    else return SearchSubtree(root[T], k)
```

si basa sull'operazione ausiliaria `SearchSubTree`:

```
SearchSubtree(x, k)
    i <- 1
    while i<= n[x] and k>key_i[x] do
        i <- i+1 // ricerca dell'indice i tale che k <= key_i [x]
                 // invariante: proprietà degli intervalli
        if i <= n[x] and k = key_i [x] return x, i // successo
    else if leaf[x] return nil // ricerca senza successo
    else // ricerca ricorsiva nel sottoalbero c_i[x]
        DiskRead(c_i[x])
        return SearchSubtree(c_i[x], k)
```

che prende un nodo come argomento. Il costo complessivo va calcolato sommando il costo ricorsivo della discesa al costo additivo della ricerca sequenziale o binaria sulle chiavi del nodo considerato. La versione binaria prende la seguente forma:

```
SearchSubtree(x, k)
    i <- 1, j <- n[x]+1
    while i < j do
        if k <= key_floor((i+j)/2) [x] then j <- floor((i+j)/2)
        else i <- floor((i+j)/2) + 1
    if leaf[x] return nil // ricerca senza successo
    else // ricerca ricorsiva nel sottoalbero c_i[x]
        DiskRead(c_i[x])
        return SearchSubtree(c_i[x], k)
```

Il numero di *DiskRead* è, al più, uguale all'altezza h dell'albero, ed è quindi $O(h) = O(\log_t N)$ con N chiavi nel B-albero. Il tempo T di CPU per la ricerca è $T = O(th) = O(t \log_t N)$. Usando la ricerca dicotomica, il costo è $O(\log th)$ e complessivamente $O(\log t \log_t N) = O(\log N)$.

L'inserimento di una chiave in un B-albero segue un processo specifico. Non si aggiunge mai ai nodi interni. Le chiavi vengono inserite solo nelle foglie. Questo perché l'inserimento nei nodi interni creerebbe sottoalberi, complicando la gestione dell'albero. Una chiave viene inserita solo in una foglia. Se la foglia non è piena, l'inserimento avviene direttamente. Se la foglia è piena, il nodo viene diviso. Se la foglia è piena, viene divisa in due nodi più piccoli. L'elemento centrale viene spostato nel nodo padre, mantenendo l'ordinamento delle chiavi. La complessità dell'inserimento è $O(h)$, dove h è l'altezza dell'albero. Questo è dovuto al fatto che ogni operazione di inserimento richiede un numero costante di accessi al disco (DiskRead e DiskWrite) tra due chiamate consecutive di `insertNonFull`. Consideriamo, ad esempio, un B-albero con grado minimo $t = 4$. In questo caso, il numero di chiavi in un nodo è compreso tra 3 e 7 (limite inferiore $t - 1$, limite superiore $2t - 1$). Se la radice è piena, si crea una nuova radice e si sposta la vecchia radice come sua figlia. Il nodo figlio viene quindi diviso in due nodi più piccoli, spostando l'elemento centrale nella nuova radice. Questo processo mantiene l'ordinamento delle chiavi, poiché le chiavi sono già ordinate.

La cancellazione di una chiave può lasciare un nodo con un numero di chiavi inferiore al minimo. In questo caso, si uniscono due nodi fratelli, inserendo come intermezzo la chiave intermedia presa dal padre. Questa operazione è speculare alla divisione (*split*). Le operazioni di inserimento e cancellazione in un B-albero sono progettate per mantenere l'albero bilanciato e ordinato, garantendo così un accesso efficiente ai dati. La complessità di queste operazioni è $O(h)$, dove h è l'altezza dell'albero, grazie a un numero costante di accessi al disco tra le chiamate consecutive delle procedure ausiliarie.

Tabelle hash

Un'altra struttura dati concreta per il tipo astratto `Dizionario` è la *tabella hash* o *tavola ad accesso diretto*. L'idea alla base delle tabelle *hash* è mappare direttamente la chiave all'indice di un *array*. Questo è semplice se le chiavi sono univoche e numeriche, purché non superino le dimensioni dell'*array*. I costi associati all'accesso a diverse strutture dati variano: una lista e un albero di ricerca non bilanciato hanno un costo di $O(n)$, mentre un albero di ricerca bilanciato ha un costo di $O(\log n)$. Le tabelle hash, invece, offrono un costo di $O(1)$.

Di seguito è riportato lo pseudocodice dell'implementazione di una tabella *hash*.

```
classe TavolaAccessoDiretto implementa Dizionario:
dati:
    un array v di dimensione m>=n in cui v[k] = elem se c'è un elemento con
    chiave n nel dizionario, e v[k] = null altrimenti.
    Le chiavi k devono essere nell'intervallo [0, m-1].

operazioni:
    insert(elem e, chiave k)
        v[k] <- e
    delete(chiave k)
        v[k] <- null
    search(chiave k) -> elem
        return v[k]
```

Tutte e tre le operazioni hanno complessità temporale lineare: $T(n) = O(1)$. Il fattore di carico α è definito come il rapporto tra il numero di elementi n e la capacità m della struttura dati:

$$\alpha = \frac{n}{m}$$

La *funzione hash* h mappa un dominio totalmente ordinato U (che può includere chiavi non numeriche) a un intervallo $[0, m-1]$. L'elemento con chiave k viene posizionato in $v[h(k)]$. Un problema comune è la *gestione delle*

collisioni: è difficile trovare una funzione *hash* h che sia veramente univoca (*hash perfetta*), dove $u \neq v$ implica $h(u) \neq h(v)$. Questo è possibile solo se il numero di elementi in U è minore o uguale a m, permettendo a h di essere iniettiva. Solitamente, la funzione *hash* è suriettiva. Una collisione si verifica quando si inserisce un elemento la cui chiave ha un valore *hash* uguale a quello di un elemento già memorizzato. Per ridurre le collisioni, è possibile limitare l'insieme delle chiavi.

Per ottenere un dizionario con un tempo di accesso veramente $O(1)$, è necessario disporre di una funzione *hash* che sia *perfetta*, ovvero che mappi ogni chiave a un indice univoco senza collisioni, e calcolabile in tempo $O(1)$. Se queste condizioni non sono soddisfatte, è importante che la funzione *hash* garantisca almeno l'*uniformità semplice*, ovvero che ogni elemento abbia la stessa probabilità di causare una collisione.

Le funzioni *hash* crittografiche sono progettate per essere *unidirezionali*, il che significa che è difficile risalire al messaggio originale a partire dal valore *hash*. Queste funzioni sono resistenti alle collisioni, alla *preimmagine* (è difficile trovare un messaggio che corrisponda a un dato valore hash) e alla *seconda preimmagine* (è difficile trovare due messaggi diversi che producano lo stesso valore *hash*). # Algoritmi di ordinamento

Gli algoritmi di ordinamento possono essere classificati in base alla loro complessità temporale e al tipo di operazioni che utilizzano. Tipicamente, gli algoritmi di ordinamento hanno una complessità di $O(n^2)$ o $O(n \log n)$ quando sono basati sul confronto. Esistono anche algoritmi di ordinamento che non si basano sul confronto e che possono raggiungere una complessità di $O(n)$.

MergeSort

MergeSort è un algoritmo di ordinamento che utilizza la strategia *divide et impera*. L'idea è scomporre il problema iniziale in sottoproblemi più piccoli, risolvere tali sottoproblemi ricorsivamente e poi unire le soluzioni. In pratica, l'algoritmo dimezza l'*array*, applica l'algoritmo ai sottoarray e poi fonde le sottosequenze ordinate. L'algoritmo continua a dividere finché non rimangono solo *array* di 1 o 2 elementi. Ad esempio, partendo dall'array `[52804719326]`, la divisione procede come segue:

```
[528047] [19326]
[528] [047]
[52] [8]
[5] [2]
```

Una volta che l'*array* è stato completamente suddiviso, inizia la fase di fusione (*merge*). Ad esempio, partendo dai sottoarray `5 2`, la fusione procede come segue:

```
[25] [8]
[258]
```

La fusione avviene scorrendo le due liste con due indici, confrontando le coppie di elementi nelle due liste e scegliendo quale elemento mettere per primo nell'*array* fuso. Questa procedura funziona correttamente solo se gli *array* di partenza sono già ordinati.

Ecco lo pseudocodice per l'algoritmo *MergeSort* operando su un array di dimensione n:

```
MergeSort(A, p, r)
    if p < r then
        q <- floor((p + r) / 2)
        MergeSort(A, p, q)    // [n/2] elementi
        MergeSort(A, q + 1, r)  // [n/2] elementi
        Merge(A, p, q, r)
```

La procedura `Merge` funziona come segue:

1. Estrai ripetutamente il minimo tra gli elementi dei sottoarray `A[p..q]` e `A[q+1..r]` e copialo in un array di output `C` fino a quando uno dei due sottoarray non si svuota.
2. Copia gli elementi rimasti dal sottoarray non svuotato in `C`.

L'algoritmo MergeSort non opera in loco, poiché richiede un array di output `C`. La complessità temporale della procedura `Merge` è lineare, ovvero $\Theta(n)$, dove $n = r - p + 1$.

```
merge(A,p,q,r)
    n_1 <- q - p + 1
    n_2 <- r - q
    for i <- 1 to n_1 do
        L[i] <- A[p+i-1]
    for j <- 1 to n_2 do
```

```
        R[j] <- A[q+j]
    L[n_1 + 1] <- R[n_2 + 1] <- infinity
    i <- j <- 1
    for k <- p to r do
        if L[i] <= R[j] then
            A[k] <- L[i]
            i <- i + 1
        else
            A[k] <- R[j
```

I costi del *merge* possono essere suddivisi in tre componenti principali. L'assegnamento di q è lineare, mentre la divisione risulta poco costosa. Il passo più oneroso è rappresentato dalla fusione.

Il numero di confronti può essere espresso dalla seguente equazione:

$$T(n) = d(n) + 2 \cdot T\left(\frac{n}{2}\right) + c(n)$$

dove d rappresenta il costo di divisione, che è $\Theta(1)$, e $c(n)$ rappresenta il costo di fusione, che è $\Theta(n)$. Di conseguenza, il costo totale rientra nella classe $\Theta(n)$.

Applicando il teorema Master con $a = 2$ e $b = 2$, si ottiene

$$T(n) = \Theta(n \log n).$$

Pertanto, il *MergeSort* risulta ottimale dal punto di vista temporale, ma non lo è dal punto di vista della memoria, poiché non può essere eseguito in loco.

QuickSort

QuickSort è un algoritmo di ordinamento che divide il problema in due sottoproblemi scegliendo un elemento *pivot* e separando gli elementi maggiori e minori rispetto a questo pivot.

La versione non *in loco* di QuickSort funziona come segue:

1. Scegli un elemento x (il *pivot*) dall'array A.
2. Partiziona A rispetto a x calcolando due sotto-array:
   - A_1 contiene tutti gli elementi di A che sono minori o uguali a x.
   - A_2 contiene tutti gli elementi di A che sono maggiori di x.
3. Se A_1 contiene più di un elemento, applica QuickSort a A_1.
4. Se A_2 contiene più di un elemento, applica QuickSort a A_2.
5. Copia la concatenazione di A_1 e A_2 in A.

La partizione *in loco* di QuickSort avviene scorrendo l'array da sinistra verso destra e da destra verso sinistra. Durante questa scansione, ci si ferma su un elemento maggiore del pivot quando si scorre da sinistra verso destra, e su un elemento minore del pivot quando si scorre da destra verso sinistra. A questo punto, si scambiano i due elementi e si continua finché gli indici non si incrociano. Una volta che gli indici si incrociano, si pongono i due sotto-array a sinistra e a destra del pivot.

Per implementare QuickSort in loco, è necessario definire una funzione ausiliaria chiamata `Partition(A, i, f)`:

1. $x = A[i]$ (partiziona $A[i..f]$ intorno al pivot $A[i]$).
2. Inizializza inf $= i$ e sup $= f + 1$.
3. Esegui un ciclo infinito:
   - Incrementa inf finché inf $\leq f$ e A[inf] $\leq x$.
   - Decrementa sup finché A[sup] $> x$.
   - Se inf $<$ sup, scambia A[inf] e A[sup].
   - Altrimenti, esci dal ciclo.
4. Scambia $A[i]$ e A[sup].
5. Restituisci sup.

Alla riga 5, la condizione è sufficiente perché l'indice non uscirà mai dall'array senza prima incontrare il pivot e bloccarsi. La riga 8 posiziona il pivot al centro. La riga 9 restituisce il nuovo pivot. Il tempo di esecuzione di QuickSort è $\Theta(n)$ ad ogni iterazione, poiché vengono effettuati $n - 1$ confronti tra gli elementi e il pivot.

Ora possiamo scrivere l'implementazione completa di QuickSort:

```
QuickSort(A, i, f)
    if (i >= f) then return
```

```
m = Partition(A, i, f)
QuickSort(A, i, m-1)
QuickSort(A, m+1, f)
```

Questa versione di QuickSort non è stabile, poiché l'albero delle chiamate ricorsive può essere sbilanciato a seconda della distribuzione dell'array di origine rispetto al pivot. Una possibile ottimizzazione è la scelta dell'elemento mediano come pivot iniziale.

Caso peggiore

Nel caso peggiore, il pivot è il minimo o il massimo dell'*array* (*array* ordinato direttamente o inversamente). Il numero di confronti è:

$$T(n) = \Theta(n) +$$

Utilizzando il metodo dell'*albero della ricorsione*, sommiamo i nodi di tutti i livelli e notiamo che sono:

$$T(n) = \sum_{i=2}^{n} i + \not{\pi} = \Theta(n^2)$$

Caso migliore

Nel caso migliore, l'albero di ricorsione è perfettamente bilanciato (due sottoalberi di dimensione non maggiore di $n/2$), e l'algoritmo ha un costo coincidente a quello del *MergeSort* (pseudolineare in n):

$$T(n) = \Theta(n \log n)$$

Caso medio

Per ottenere un'approssimazione del caso medio di *QuickSort*, possiamo scegliere un *pivot* dall'*array* in modo casuale. Questo riduce la possibilità di un comportamento peggiore del previsto. Possiamo ulteriormente migliorare la scelta del *pivot* selezionando il mediano di un sottoinsieme di elementi estratti casualmente.

Nel caso di una sola estrazione, ogni elemento ha una probabilità di $\frac{1}{n}$ di essere scelto come pivot. Il numero C di confronti nel caso atteso è dato da:

$$C(n) = \sum_{a=0}^{n-1} \frac{1}{n} \left[n - 1 + C(a) + C(n - a - 1) \right] = n - 1 + \sum_{a=0}^{n-1} \frac{2}{n} C(a)$$

dove a e $(n - a - 1)$ sono le dimensioni dei sottoproblemi risolti ricorsivamente. $C(a)$ e $C(n - a - 1)$ danno luogo alla stessa sommatoria perché entrambe le chiamate alternano partizioni buone e cattive. La relazione di ricorrenza $C(n) = n - 1 + \sum_{a=0}^{n-1} \frac{2}{n} C(a)$ ha soluzione $C(n) \leq 2n \log n$, quindi:

$$T(n) = O(n \log n)$$

Non possiamo usare Θ perché la soluzione è ottenuta per integrazione e non è dunque possibile ottenere un limite stretto.

Java implementa *QuickSort*, *MergeSort* e *HeapSort* come funzioni nelle classi `Arrays` e `Collections`. Queste implementazioni sono ottimizzate per garantire prestazioni efficienti e affidabili in una vasta gamma di scenari.

Algoritmi lineari

Gli algoritmi di ordinamento che funzionano in tempo lineare in n hanno condizioni particolari di applicabilità. Tra questi, troviamo l'IntegerSort (o CountingSort), il BucketSort e il RadixSort.

IntegerSort

L'IntegerSort funziona esclusivamente su vettori X di numeri interi di cui siano noti il minimo e il massimo. I valori devono essere compresi nell'intervallo $[1, k]$. L'algoritmo costruisce un array di appoggio Y di k elementi, inizializzato a zero. Scorrendo X, per ogni occorrenza del numero n, incrementa di uno l'n-esima cella di Y. Al termine dello scorrimento, ricostruisce X copiando al suo interno gli indici di Y, ognuno un numero di volte pari al numero contenuto nella rispettiva cella.

Osserviamo lo pseudocodice:

```
IntegerSort(X, k)
1. sia Y un array di dimensione k
2. for i = 1 to k
3.    Y[i] = 0
4. for j = 1 to n
5.    Y[X[j]] = Y[X[j]] + 1
6. k = 1
7. for i = 1 to k
8.    while Y[i] > 0
9.       X[k] = i
10.      Y[i] = Y[i] - 1
11.      k = k + 1
```

Analizziamo la complessità temporale dell'algoritmo: - Il tempo per inizializzare Y a zero è $O(k)$. - Il tempo per contare gli indici è $O(n)$. - Il ciclo esterno della ricostruzione è compatto e viene eseguito k volte. - I cicli interni sono indefiniti, ma le iterazioni totali saranno n.

In sintesi, l'algoritmo ha una complessità temporale di $O(k + n)$, che si approssima a $O(n)$ se $k \leq n$. Non essendo basato sul confronto, non ha il limite inferiore $\Omega(n \log n)$.

Stabilità degli algoritmi

Un algoritmo è definito *stabile* se preserva l'ordine iniziale tra elementi con la stessa chiave. La stabilità è particolarmente utile per ordinamenti su più chiavi. Applicando algoritmi stabili, è possibile ordinare prima per una chiave e poi per un'altra; se l'algoritmo non è stabile, gli elementi rimarranno ordinati solo per l'ultima chiave applicata.

BucketSort

BucketSort è un algoritmo utilizzato per ordinare *array* con chiavi intere. Funziona inserendo l'intero *record* nella posizione corrispondente alla sua chiave in un nuovo *array* ordinato. Questo metodo è particolarmente efficace quando le chiavi sono distribuite uniformemente. Per ordinare n *record* con chiavi intere nell'intervallo $1, k]$, consideriamo un esempio pratico: ordinare n record con campi come nome, cognome, anno di nascita, matricola, ecc. Si potrebbe desiderare di ordinare i record per matricola o per anno di nascita. L'*input* del problema è costituito da n record mantenuti in un array. Ogni elemento dell'*array* è un record che include un *campo chiave* (rispetto al quale si desidera ordinare) e altri campi associati alla chiave (*informazioni satellite*). Per risolvere questo problema, è sufficiente mantenere un *array* di liste, anziché di contatori, e operare in modo simile all'algoritmo IntegerSort. La lista $Y[x]$conterrà gli elementi con chiave uguale a x. Infine, è necessario concatenare le liste per ottenere l'ordinamento desiderato.

BucketSort (X, k) 1. Sia Y un array di dimensione k 2. for $i = 1$ to k `do` $Y[i]$ = lista vuota `// Inizializzazione` 3. `for` $j = 1$ `to` n `do` 4. `if` $(key(X[j]) \in 1, k)$ `then` errore 5. `else` appendi il record $X[j]$ alla lista $Y[key(X[j])]$ 6. `for` $i = 1$ to k `do // Ordine crescente a livello di bucket` 7. copia l'ordinamento in X gli elementi della lista $Y[i]$

Un algoritmo è definito stabile se preserva l'ordine iniziale tra elementi con la stessa chiave. Il BucketSort è reso stabile appendendo gli elementi di Xin coda alla lista appropriata $Y[i]$. Una domanda pertinente è se gli altri algoritmi di ordinamento che abbiamo visto finora siano stabili. Tra questi, possiamo considerare il CountingSort, il Mergesort e il Quicksort.

Per ordinare $n$interi nell'intervallo $[1, k]$utilizzando il BucketSort, sono necessarie $O(\log_b k)$passate. Ciascuna passata richiede un tempo di $O(n + b)$. Utilizzando la relazione $\log_2 k = \log_n k \log_2 n$, il tempo complessivo dell'algoritmo è $O((n + b) \log_b k)$.

Se $b = O(n)$, il tempo di esecuzione diventa $O(n \log_b k) = O(n \log k)$. Se $b = O(1)$, il tempo di esecuzione è $O(n \log k) = O(n \log n)$. L'algoritmo raggiunge un tempo lineare se $k = O(n^c)$, dove cè una costante. In media, il tempo è S, ovvero $O(n)$ se $k = O(n)$.

RadixSort

RadixSort è un algoritmo di ordinamento che opera in modo inverso rispetto ai metodi tradizionali, ordinando gli elementi numerici a partire dalla cifra meno significativa fino a quella più significativa. Questo processo viene eseguito cifra per cifra, applicando il BucketSort a ciascuna cifra. RadixSort trova applicazione in vari campi, tra cui il controllo del plagio e la biologia molecolare computazionale, dove è utilizzato per identificare la sottostringa ripetuta più lunga. Il RadixSort rappresenta i valori in una certa base $b$ed esegue una serie di BucketSort sulle cifre in modo bottom-up, partendo dalla cifra meno significativa verso quella più significativa.

L'algoritmo RadixSort può essere descritto come segue:

RadixSort(A, 0) = for i = 0 to t-1 do - Esegui un algoritmo di BucketSort lineare per ordinare gli elementi di A sulla cifra i.

La correttezza del RadixSort si basa su due principi fondamentali. Se x e y hanno una diversa i-esima cifra, la i-esima passata di BucketSort li ordina correttamente. Se x e y hanno la stessa i-esima cifra, la proprietà di stabilità del BucketSort li mantiene ordinati correttamente. Dopo la t-esima passata di BucketSort, i numeri sono correttamente ordinati rispetto alle $t$cifre meno significative.

Esempio

Supponiamo di voler ordinare $n = 10^6$numeri da 32 bit. Come scegliere la base b? - $n = 10^6$ è compreso tra 2^{19} e 2^{20}. - Scegliendo $b = 2^{16} = 65536$, si ha: - $k = 2^{16}/1 = 65536$ con $c = 2$. - Sono sufficienti 2 passate di BucketSort. - Ogni passata richiede tempo lineare $O(n)$.

Usi

Il sorter IBM 802 è un esempio storico di applicazione del RadixSort. Questo dispositivo era utilizzato per ordinare le schede perforate da 12x80 per il linguaggio FORTRAN. Il RadixSort trova applicazione in vari altri campi, tra cui l'ordinamento delle stringhe, l'indicizzazione dei testi, il controllo di copiature e plagio, e la biologia molecolare computazionale. L'identificazione di geni, ad esempio, si può effettuare cercando la sottostringa ripetuta più lunga in una stringa di Ncaratteri rappresentanti le basi azotate. Queste applicazioni sfruttano l'efficienza e la stabilità del RadixSort per gestire grandi volumi di dati in modo efficiente.

Segue lo pseudocodice dell'algoritmo RadixSort, utilizzando come base per il bucketSort a valore t. La t-esima cifra più significativa di BucketSort considera come chiave la t-esima cifra meno significativa della rappresentazione decimale del numero.

```
procedura bucketSort(array A di n interi, interi b e t)
  sia T un array di dimensione b
  for i = 1 to n do
    c = estrai cifra t-esima di A[i] //Inizializzazione
    T[c] = T[c] + 1 // rappresentazione in base b
  for i = 1 to b do
    for j = 1 to T[i] do
      copia ordinatamente in A gli elementi della lista Y[i]
  fine procedura

algorithmo radixSort(array A di n interi)
  while (cifra t-esima di A non è 0)
    bucketSort(A, 10, t)
  t = t - 1
```

Riepilogo e confronto

BucketSort e RadixSort sono generalizzazioni del CountingSort. Quest'ultimo, in particolare, assume che gli elementi siano compresi nell'intervallo $[1, k]$, ed è particolarmente utile quando k è piccolo, ovvero dell'ordine di $O(n)$. Se ogni *bucket* ha dimensione 1, si ottiene il CountingSort. Il BucketSort, d'altra parte, presuppone la conoscenza della distribuzione dell'ingresso. Nel caso peggiore, il BucketSort ha una complessità di $\Theta(n^2)$, mentre nel caso medio la sua complessità è $\Theta(n)$. Il RadixSort, invece, assume che gli interi siano composti da t cifre, con ogni cifra in base b. Questo algoritmo è particolarmente utile quando i numeri da ordinare sono molti, t è costante e piccolo. È importante notare che nessuno di questi algoritmi esegue l'ordinamento in loco.

Grafi

Una struttura dati non lineare e non gerarchica è rappresentata dai *grafi*. Un grafo è costituito da un insieme di *vertici*, o *nodi*, e da un insieme di *archi* o *spigoli* che collegano questi vertici. I grafi possono essere *orientati* o non *orientati*. In un grafo orientato, gli archi hanno una direzione specifica, mentre in un grafo non orientato gli archi sono bidirezionali. Nel primo caso la relazione non è simmetrica, nel secondo lo è. Un *cappio* è un arco i cui estremi coincidono. Un tipo particolare di grafo è il *multigrafo*, che ammette la presenza di più archi tra la stessa coppia di nodi.

Un grafo orientato è definito *semplice* se non presenta due archi con gli stessi estremi. In caso contrario, si parla di multigrafo. A meno che non sia specificato diversamente, assumeremo sempre che un grafo sia semplice. Due vertici collegati da uno spigolo sono detti *adiacenti*, e tale spigolo è detto incidente ai due vertici. Il *grado* di un vertice è il numero di archi incidenti ad esso. La somma dei gradi di tutti i vertici è pari al doppio del numero di archi nel grafo. Nei grafi orientati, la relazione di adiacenza segue l'orientamento: il vertice da cui l'arco esce è adiacente al vertice in cui l'arco entra. Di conseguenza, il grado di un vertice è dato dalla somma del numero di archi entranti e uscenti. A livello complessivo del grafo, il grado totale di archi entranti è uguale al grado totale uscente, ed entrambi sono pari al numero di archi totali. Come nel caso dei grafi non orientati, il grado complessivo è pari al doppio degli archi totali.

Un *cammino semplice* è un percorso che attraversa vertici distinti in un grafo. Il cammino più corto tra due nodi è detto distanza. Un ciclo semplice è un percorso chiuso che non si ripete. Un cappio è un ciclo semplice di lunghezza unitaria.

Componenti connesse

Una *componente connessa* di un grafo G è un insieme massimale di vertici $U \subseteq V$ tale che per ogni coppia di vertici in U esiste un cammino che li collega in G. Un grafo non orientato $G = (V, E)$ è connesso se esiste almeno un cammino tra ogni coppia di vertici, o equivalentemente, se esiste un'unica componente connessa. È facile verificare che la relazione tra vertici "essere raggiungibile da" è una *relazione di equivalenza*, poiché gode delle proprietà riflessiva, simmetrica e transitiva. Di conseguenza, le componenti connesse di un grafo non orientato sono le classi di equivalenza dei suoi vertici rispetto alla relazione di raggiungibilità.

Una *componente fortemente connessa* di un grafo orientato G è un insieme massimale di vertici $U \subseteq V$ tale che, per ogni coppia di vertici u e v in U, esiste in G un cammino orientato che collega u a v e un cammino orientato che collega v a u. Un grafo orientato $G = (V, E)$ è fortemente connesso se esiste almeno un cammino orientato tra ogni coppia di vertici. Le componenti fortemente connesse rappresentano le classi di equivalenza dei vertici rispetto alla relazione di equivalenza definita dalla connettività forte.

Gli *alberi* sono casi particolari di grafi. Un albero è un grafo non orientato, connesso e aciclico. Inoltre, un albero ha esattamente $|V| - 1$ archi, come dimostrabile.

Un DAG (*Directed Acyclic Graph*) è un grafo orientato ed aciclico.

Implementazione

Una possibile interfaccia per un grafo in Java:

```
interface Grafo {
    public int n();
    public int m();
    public int grado(Vertice v);
    public Arco[] archi(Vertice v);
    public Arco sonodi(Vertice v);
    public void aggiungiArco(Vertice x, Vertice y);
    public void aggiungiVertice(Vertice v);
    public void cancellaArco(Arco e);
    public void cancellaVertice(Vertice v);
    // fine...
}
```

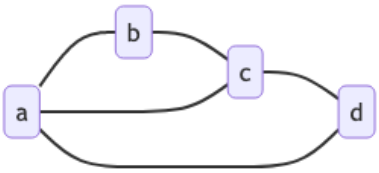
Ecco alcune delle più note e utilizzate librerie per la gestione dei grafi nel linguaggio Java:

- **JGraphT** https://jgraph.github.io/
  - Per gli algoritmi!
  - Usa Jgraph come UI
- **JDSL** (Data Structures Library in Java) http://128.148.32.110/cgc/jdsl/

- **Annas** http://code.google.com/p/annas/
- **JGraphX** - Java Swing graph visualization library https://github.com/jgraph/jgraphx
  - Utili come debugging dei vostri algoritmi
- Altre API alternative: Jung, yworks, prefuse.org e BFG

Linearizzazione dei grafi

Un grafo può essere rappresentato in memoria in diversi modi. Prendiamo ad esempio il seguente grafo:



Lista di archi

Una lista di archi per un grafo è un elenco di tutte le coppie di vertici collegati da un arco. Nell'esempio, la lista di archi è:

$$\begin{bmatrix}(a,b) & (c,a) & (b,c) & (c,d) & (a,d)\end{bmatrix}$$

I costi associati alla rappresentazione mediante lista di archi sono rappresentati nella seguente tabella:

Operazione	Tempo di esecuzione
`grado(v)`	$O(m)$
`archiIncidenti(v)`	$O(m)$
`sonoAdiacenti(r, y)`	$O(m)$
`aggiungiVertice(v)`	$O(1)$
`aggiungiArco(r, y)`	$O(1)$
`rimuoviVertice(v)`	$O(m)$
`rimuoviArco(e)`	$O(m)$

Liste di adiacenza e di incidenza

Una lista di incidenza per un grafo è una rappresentazione che associa a ogni vertice una lista dei lati incidenti. Una lista di adiacenza è una rappresentazione che associa a ogni vertice una lista dei vertici adiacenti. Nell'esempio utilizzato, le liste di adiacenza e di incidenza assumono le seguenti forme:

lista di adiacenza

$$\begin{bmatrix} a \\ b \\ c \\ d \end{bmatrix} \begin{array}{cccccc} \to & b & \to & d & \to & c \\ \to & c & \to & a & & \\ \to & a & \to & d & \to & b \\ \to & c & \to & a & & \end{array}$$

lista di incidenza

$$\begin{bmatrix} a \\ b \\ c \\ d \end{bmatrix} \begin{array}{cccccc} \to & 0 & \to & 4 & \to & 1 \\ \to & 2 & \to & 0 & & \\ \to & 1 & \to & 3 & \to & 2 \\ \to & 4 & \to & 3 & & \end{array} \quad \textit{dove} \quad \begin{array}{cc} 0 & \begin{bmatrix}(a,b)\end{bmatrix} \\ 1 & \begin{bmatrix}(c,a)\end{bmatrix} \\ 2 & \begin{bmatrix}(b,c)\end{bmatrix} \\ 3 & \begin{bmatrix}(c,d)\end{bmatrix} \\ 4 & \begin{bmatrix}(a,d)\end{bmatrix} \end{array}$$

I costi associati alla rappresentazione mediante liste di adiacenza e di incidenza sono rappresentati nella seguente tabella:

Operazione	Tempo di esecuzione
`grado(v)`	$O(\delta(v))$
`archiIncidenti(v)`	$O(\delta(v))$
`sonoAdiacenti(x, y)`	$O(\min\{\delta(x), \delta(y)\})$
`aggiungiVertice(v)`	$O(1)$
`aggiungiArco(x, y)`	$O(1)$
`rimuoviVertice(v)`	$O(n)$
`rimuoviArco(e = (x, y))`	$O(\delta(x) + \delta(y))$

Matrice di adiacenza

Una matrice di adiacenza per un grafo è una matrice quadrata dove l'elemento in posizione (i, j) è 1 se c'è un arco da i a j, altrimenti è 0. La matrice di adiacenza per l'esempio corrente è riportata di seguito:

	a	b	c	d
a	0	1	1	1
b	1	0	1	0
c	1	1	0	1
d	1	0	1	0

I costi associati alla rappresentazione mediante matrice di adiacenza sono rappresentati nella seguente tabella:

Operazione	Tempo di esecuzione
`grado(v)`	$O(n)$
`archiIncidenti(v)`	$O(n)$
`sonoAdiacenti(x, y)`	$O(1)$
`aggiungiVertice(v)`	$O(n^2)$
`aggiungiArco(x, y)`	$O(1)$
`rimuoviVertice(v)`	$O(n^2)$
`rimuoviArco(e)`	$O(1)$

Matrice di incidenza

Una matrice di incidenza per un grafo è una matrice che indica quali vertici sono connessi da ciascun arco. La matrice di incidenza per l'esempio corrente è riportata di seguito:

	(a, b)	(c, a)	(b, c)	(c, d)	(a, d)
a	1	1	0	0	1
b	1	0	1	0	0
c	0	1	1	1	0
d	0	0	0	1	1

I costi associati alla rappresentazione mediante matrice di incidenza sono rappresentati nella seguente tabella:

Operazione	Tempo di esecuzione
`grado(v)`	$O(m)$
`archiIncidenti(v)`	$O(m)$
`sonoAdiacenti(x, y)`	$O(m)$
`aggiungiVertice(v)`	$O(nm)$
`aggiungiArco(x, y)`	$O(nm)$
`rimuoviVertice(v)`	$O(nm)$
`rimuoviArco(e)`	$O(n)$

Grafi orientati

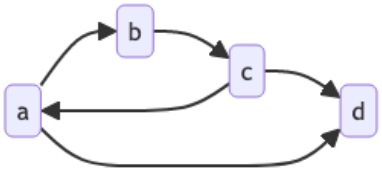


Figura 1: diagram

La lista degli archi rimane formalmente invariata nel caso di grafi orientati. È importante prestare attenzione a riportare, per ogni coppia, il nodo da cui l'arco esce come primo elemento della coppia, e il nodo in cui l'arco entra come secondo elemento:

$$\big[(a, b) \quad (c, a) \quad (b, c) \quad (c, d) \quad (a, d)\big]$$

Nelle liste di adiacenza e di incidenza e nelle matrici di adiacenza per grafi orientati si riportano soltanto gli archi uscenti:

lista di adiacenza

$$\begin{bmatrix} a \\ b \\ c \\ d \end{bmatrix} \begin{array}{ccccc} \rightarrow & b & \rightarrow & d \\ \rightarrow & c & & \\ \rightarrow & a & \rightarrow & d \\ & & & \end{array}$$

lista di incidenza

$$\begin{bmatrix} a \\ b \\ c \\ d \end{bmatrix} \begin{array}{ccccc} \rightarrow & 0 & \rightarrow & 4 \\ \rightarrow & 2 & & \\ \rightarrow & 1 & \rightarrow & 3 \\ & & & \end{array} \quad \textit{dove} \quad \begin{array}{cc} 0 & (a,b) \\ 1 & (c,a) \\ 2 & (b,c) \\ 3 & (c,d) \\ 4 & (a,d) \end{array}$$

matrice di adiacenza

	a	b	c	d
a	0	1	1	1
b	1	0	1	0
c	1	1	0	1
d	1	0	1	0

Nelle matrici di incidenza per grafi orientati si riporta con il valore 1 l'incidenza con arco uscente, -1 l'incidenza con arco entrante:

	(a,b)	(c,a)	(b,c)	(c,d)	(a,d)
a	1	-1	0	0	1
b	-1	0	1	0	0
c	0	1	-1	1	0
d	0	0	0	-1	-1

Operazioni di visita

Una *visita* (o *attraversamento*) di un grafo G consente di esaminare i nodi e gli archi di G in modo sistematico, partendo da un vertice sorgente s. Questo problema è fondamentale in molte applicazioni e può essere affrontato con diversi tipi di visite, ciascuna con proprietà specifiche. In particolare, le visite più comuni sono la visita in ampiezza (BFS, breath first search) e la visita in profondità (DFS, depth first search).

Un vertice viene *marcato* quando viene incontrato per la prima volta; questa marcatura può essere mantenuta tramite un vettore di bit di marcatura. La visita genera un albero di copertura T del grafo, radicato in s. Un insieme di vertici $F \subseteq T$ mantiene la frangia di T. Se un vertice v è in F, significa che è aperto e che esistono archi incidenti su v non ancora esaminati. Se un vertice v è in $F - F$, significa che è chiuso e che tutti gli archi incidenti su v sono stati esaminati. Se la frangia F è implementata come coda, si ottiene la *visita in ampiezza* (BFS). Se invece la frangia F è implementata come pila, si ottiene la *visita in profondità* (DFS).

Il tempo di esecuzione della visita dipende dalla struttura dati utilizzata. Se si utilizza una lista di archi, il costo è $O(mn)$, poiché l'operazione `archiIncidenti(v)` richiede l'esame di tutti gli archi del grafo per ogni vertice v. Se si utilizza una lista di adiacenza o di incidenza, il costo è $O(m+n)$, dato che la somma delle lunghezze delle liste di adiacenza per tutti i vertici è $2m$. Se si utilizza una matrice di adiacenza, il costo è $O(n^2)$, poiché l'operazione `archiIncidenti(v)` richiede la scansione di una riga e quindi un tempo $O(n)$ per ogni vertice v. Infine, se si utilizza una matrice di incidenza, il costo è $O(mn)$, poiché l'operazione `archiIncidenti(v)` richiede $O(m)$ per ogni vertice v.

Analizziamo l'algoritmo per la visita in ampiezza:

1. Inizializza tutti i vertici come non marcati.
2. Scegli un vertice s non marcato.
3. Marca il vertice s.
4. Inserisci s in una coda.
5. Finché la coda non è vuota:
    1. Estrai un vertice v dalla coda.
    2. Se v è non marcato:
        1. Marca v.
        2. Inserisci tutti i vicini di v nella coda.
6. Se v è un nodo di radice:
    1. Aggiungi v alla coda F.
    2. Imposta s come padre di v nell'albero T.
7. Restituisci l'albero T.

Gli archi incidenti in v possono essere esaminati in qualsiasi ordine. Nell'albero BFS, ogni vertice si trova il più vicino possibile alla radice s. La visita in ampiezza gode inoltre delle seguenti proprietà: - Per ogni nodo v, il livello di v nell'albero BFS è pari alla distanza di v dalla sorgente s. - Per ogni arco (u, v) di un grafo non orientato, gli estremi u e v appartengono allo stesso livello o a livelli consecutivi dell'albero BFS. - Se il grafo è orientato, possono esistere archi (u, v) che attraversano all'indietro più di un livello.

La visita in profondità (DFS) è un algoritmo che esplora i vertici di un grafo in modo ricorsivo, partendo da un vertice sorgente s. La variabile t è utile per segnare i marcatempo di inizio e fine della visita. Possiamo dividere l'algoritmo in due componenti, seguendo la logica ricorsiva del *divide et impera:*

- procedura `visitaDFSRicorsiva(vertice v, albero T)`:
  1. Marca e visita il vertice v: `inizio[v] ← t++`.
  2. Per ogni arco (v, w):
     1. Se w non è marcato:
        1. Chiama ricorsivamente `visitaDFSRicorsiva(w, T)`.
        2. Aggiungi l'arco (v, w) all'albero T.
- algoritmo `visitaDFS(vertice s) → albero vuoto`:
  1. Chiama `visitaDFSRicorsiva(s, T)`.
  2. Restituisci l'albero T.

La complessità dell'algoritmo è $O(n + m)$, dove n è il numero di vertici e m è il numero di archi. La procedura `visitaDFSRicorsiva` è chiamata esattamente una volta per ogni vertice, e il ciclo interno viene eseguito $n(n+1)/2$ volte, portando a un tempo di esecuzione complessivo di $n(n + 1)/2$. La visita DFS gode inoltre delle seguenti proprietà: - Per ogni arco (u, v) di un grafo non orientato: - (u, v) è un arco dell'albero DFS, oppure - i nodi u e v sono l'uno discendente/antenato dell'altro, oppure - (u, v) è un arco in avanti o un arco all'indietro.

- Per ogni arco (u, v) di un grafo orientato:
  - (u, v) è un arco dell'albero DFS, oppure
  - i nodi u e v sono l'uno discendente/antenato dell'altro, oppure
  - (u, v) è un arco trasversale a sinistra, ovvero il vertice v è in un sottoalbero visitato precedentemente rispetto a u.

Riepilogo

Un grafo è una struttura composta da vertici e archi che li collegano. La terminologia include concetti come vertici, archi, grafo orientato e non orientato. Esistono diversi modi per rappresentare i grafi nella memoria di un calcolatore, come liste di adiacenza, liste di incidenza, matrici di adiacenza e matrici di incidenza. La scelta della rappresentazione può influenzare significativamente i tempi di esecuzione degli algoritmi sui grafi, come dimostrato nelle visite in ampiezza e in profondità. Gli algoritmi di visita, come la visita in ampiezza (BFS) e la visita in profondità (DFS), sono fondamentali per esplorare i grafi in modo sistematico. La visita in ampiezza esamina i vertici livello per livello, mentre la visita in profondità esplora i vertici in modo ricorsivo, seguendo i percorsi più profondi possibili. ### Problema dell'oracolo

Definire un algoritmo che, dato un array X di n interi nell'intervallo $[1, k]$, processa X in modo da poter rispondere a domande del tipo: quanti interi di X cadono nell'intervallo $[a, b]$? Questo deve essere possibile per qualsiasi valore di a e b, anche se non appartengono a X.

L'algoritmo deve richiedere un tempo di pre-processamento di $O(n + k)$ per costruire l'oracolo (*array* Y), mentre l'oracolo deve poter rispondere alle domande in tempo $O(1)$.

Idea 1: Variante dell'IntegerSort

Questa soluzione consiste nel sostruire un *array* Y di dimensione k in tempo $O(n + k)$, dove $Y[i]$ rappresenta il numero di elementi di X che sono minori o uguali a i.

Esempio:

```
X = 5 1 6 8 6
Y = 1 0 0 0 1 2 0 1 (output della fase 1 di IntegerSort)
Y = 1 1 1 1 2 4 4 5 (output della fase di pre-processamento dell'oracolo)
```

Per ottenere l'oracolo, per ogni elemento dell'*array* del CountingSort, sommiamo il valore della posizione precedente.

In pseudocodice:

```
CostruisciOracolo(X, k)
1. Sia Y un array di dimensione k
2. Per i = 1 a k, imposta Y[i] = 0
```

```
3. Per i = 1 a n, incrementa Y[X[i]]
4. Per i = 2 a k, imposta Y[i] = Y[i] + Y[i-1]
5. Restituisci Y
```

```
InterrogaOracolo(Y, k, a, b)
1. Se b > k, imposta b = k
2. Se a <= 1, restituisci Y[b]
   Altrimenti, restituisci (Y[b] - Y[a-1])
```

Interrogazione $(0, 8) = \mathrm{Y}(8) = 5$.

Altre tecniche algoritmiche

Il paradigma *divide et impera* è applicabile solo se il numero di sottoproblemi da risolvere ricorsivamente è polinomiale rispetto alla dimensione dell'*input* e se i sottoproblemi sono divisibili e indipendenti. Per questo motivo sono state sviluppate altre tecniche algoritmiche, che

Programmazione dinamica

La *programmazione dinamica* è una tecnica *bottom-up*. Si identificano sottoproblemi elementari e si risolvono. Si costruisce una tabella delle soluzioni parziali. Dato che non c'è assunzione di indipendenza tra sottoproblemi, alcuni di essi potrebbero essere ricorrenti e dunque la soluzione elaborata per la prima istanza può essere riutilizzata quando nuove istanze si ripresentano. Questa tecnica può risultare onerosa perché è completamente esaustiva sui sottoproblemi.

Esempio: Fibonacci

La serie di Fibonacci ha la seguente definizione ricorsiva: - $F(1) = 1$ - $F(n),\ n \geq 1 = n + F(n-1)$

L'algoritmo in versione *divide et impera* è il seguente:

```
algoritmo fibonacci1(intero n) -> intero
    se n = 0 o n = 1 allora restituisci 1
    altrimenti restituisci fibonacci1(n-1) + fibonacci1(n-2)
```

Il tempo di esecuzione è $T(n) = O(2^n)$ perché ogni problema crea due sottoproblemi. Costruendo un albero di ricorsione possiamo notare che molti dei sottoproblemi sono ricorrenti.

Costruiamo un algoritmo bottom-up con la programmazione dinamica:

```
algoritmo fibonacci2(intero n) -> intero
    sia Fib un array di n interi (indicizzato 1..n)
    Fib[1] <- Fib[2] <- 1
    per i = 3 a n fare
        Fib[i] <- Fib[i-1] + Fib[i-2]
    restituisci Fib[n]
```

Distanza tra due stringhe

Definiamo *edit distance* il numero minimo di modifiche elementari per trasformare una stringa in un'altra. È utilizzata nei correttori ortografici per sostituire una parola non esistente nel dizionario con la più vicina memorizzata.

Da `presto` a `risotto` servirebbero 13 operazioni "in place". Possiamo ottimizzare procedendo così:

Azione	Costo	Stringa ottenuta
Inserisco P	1	`P|RISOTTO`
Mantengo R	0	`PR|ISOTTO`
Sostituisco I con E	1	`PRE|SOTTO`
Mantengo S	0	`PRES|OTTO`
Cancello O	1	`PRES|TTO`
Mantengo T	0	`PREST|TO`
Cancello T	1	`PREST|O`
Mantengo O	0	`PRESTO|`

Approccio formale

Denotiamo con $\delta(X, Y)$ la distanza tra X e Y. Definiamo X_i il prefisso di X (ovvero l'i-esimo carattere incluso per i compreso tra 0 e m)

Riduciamo il problema di calcolare $\delta(X, Y)$ al calcolo del $\delta(X_i, Y_j)$ per ogni i, j tali che $0 \leq i \leq m$ \$ e $0 \leq j \leq n$.

Manteniamo le soluzioni parziali in una tabella D di dimensione $(m + 1) \times (n + 1)$.

Inizializzazione della tabella

Alcune soluzioni interessano le stringhe nulle: - $\delta(X_0, Y_j) = j$ partendo dalla stringa vuota X_0, basta inserire uno a uno i j caratteri di Y_j. - $\delta(X_i, Y_0) = i$ partendo da X_i, basta rimuovere uno ad uno gli i caratteri per ottenere Y_0 Queste soluzioni sono rispettivamente memorizzate nella prima riga e prima colonna della tabella D.

Il costo $\delta(X_i, Y_j)$ è ignoto per $i \geq 1$ e $j \geq 1$. Se $x_i = y_j$, il minimo costo per trasformare X_i in Y_j è uguale al minimo costo per trasformare X_{i-1} in Y_{j-1}:

$$D[i, j] = D[i - 1, j - 1]$$

Se invece $x_i \neq y_j$, distinguiamo in base all'ultima operazione usata per trasformare X_i in Y_j in una sequenza ottima di operazioni. Il minimo costo per trasformare X_i in Y_j, relativo all'operazione `inserisci($i_j)`, è uguale al minimo costo per trasformare X_i in Y_{j-1} più 1 per inserire il carattere y_j:

$$D[i, j] = 1 + D[i, j - 1]$$

Il minimo costo per trasformare X_i in Y_j, relativo all'operazione `cancella($i_j)`, è uguale al minimo costo per trasformare X_{i-1} in Y_j più 1 per la cancellazione del carattere x_i

$$D[i, j] = 1 + D[i - 1, j]$$

Il minimo costo per trasformare X_i in Y_j, relativo all'operazione `sostituisci(`(x_i, y_j)`)`, è uguale al minimo costo per trasformare X_{i-1} in Y_{j-1} più 1 per sostituire il carattere x_i con y_j

$$D[i, j] = 1 + D[i - 1, j - i]$$

In conclusione, per $i \geq 1$ e $j \geq 1$:

$$D[i, j] = \begin{cases} D[i - 1, j - 1], & \text{se } x_i = y_i \\ 1 + \min\{D[i, j - 1], D[i - 1, j], D[i - 1, j - 1]\} & \text{altrimenti} \end{cases}$$

Di seguito è riportata la tabella D costruita dall'algoritmo. In grassetto sono indicate due sequenze di operazioni che permettono di ottenere la distanza tra le stringhe.

	P	**R**	**E**	**S**	**T**	**O**	
	0	**1**	2	3	4	5	6
R	1	1	**1**	2	3	4	5
I	2	2	2	**2**	3	4	5
S	3	3	3	3	**2**	3	4
O	4	4	4	4	**3**	3	3
T	5	5	5	5	**4**	**3**	4
T	6	6	6	6	5	**4**	4
O	7	7	7	7	6	5	**4**

L'algoritmo può essere formalizzato in pseudocodice nel seguente modo:

```
algoritmo distanzaStringhe(stringa X, stringa Y) -> intero
    matrice D di (m+1) x (n+1) interi
    for i = 0 to m do D[i,0] <- j
    for j = 1 to n do D[0,j] <- i
    for i = 1 to m do
        for j = 1 to n do
            if(x_i != y_j) then
                D[i,j] <- 1 + min{D[i,j-1],D[i-1,j],D[i-1,j-1]}
            else D[i,j] <- D[i-1,j-1]
```

```
    return D[m,n]
```

Il tempo di esecuzione e l'occupazione memoria sono $\Theta(m \cdot n)$.

Le soluzioni di programmazione dinamica sono semplici da analizzare perché usano cicli compatti, ma sono pesanti da elaborare per lo stesso motivo.

Esempio: resto del distributore

Un distributore di bibite contiene al suo interno n monete i cui valori (interi positivi) sono ripetutamente scelti con probabilità $p_1, p_2, ..., p_n$. Si consideri il problema di decidere se esiste una moneta r (un intero positivo) utilizzando un opportuno sottoinsieme delle n monete a disposizione.

1. Descrivere un algoritmo efficiente per decidere se il problema ammette una soluzione oppure no.
2. Determinare il costo computazionale dell'algoritmo descritto al punto 1.
3. Modificare l'algoritmo di cui al punto 1 per determinare se quali nono le monete da erogare per produrre il resto R. Descrivere il costo computazionale dell'algoritmo modificato in modo qualitativo.

Si definisca la matrice booleana $M[1..n, 0..R]$ tale che $M[i, r]$ è `true` se e solo se esiste un sottoinsieme delle prime i monete di valore complessivo uguale a r. Come caso base, se $r = 1$ possiamo solo scegliere se usare la prima moneta oppure no. In tal caso possiamo erogare solamente un resto pari a zero (cioè usando la moneta) oppure pari al valore della moneta, $c[1]$. Quindi per ogni $r = 0, ..., R$ otteniamo:

$$M[1, r] = \begin{cases} true & \text{se } r = 0 \text{ oppure } r = c[1] \\ false & \text{altrimenti} \end{cases}$$

Il problema si riduce ora a trovare se c'è una moneta r tale che $M[n, R]$ sia `true`. Il calcolo della matrice $M[i, r]$ è effettuato in tempo $O(nR)$.

L'algoritmo può essere formalizzato nel modo seguente:

```
algoritmo RESTO( array c[1..n] di int, int R ) → bool
    array M[1..n, 0..R] di bool;
    // inizializza M[1, r]
    for r:=0 to R do
        if ( r == 0 || r == c[1] ) then
            M[1, r] := true;
        else
            M[1, r] := false;
        endif
    endfor
    // calcola i restanti elementi della tabella
    for i := 2 to n do
        for r := 0 to R do
            if ( r >= c[i] ) then
                M[i, r] := M[i-1, r] || M[i-1, r-c[i]];
            else
                M[i, r] := M[i-1, r];
            endif
        endfor
    endfor
    return M[n, R];
```

Per determinare il monte da usare, calcoliamo una ulteriore matrice booleana $U[i, r]$ della stessa dimensione di M: $U[i, r]$ è `true` se e solo se si può essere la moneta i-esima per erogare il resto r.

```
algorithmo DeterminaMoneta(i, n, int, int, R) -> bool
    array U[1..n, 0..R] di bool;
    array M[1..n, 0..R] di bool;
    // inizializza M[1, r] e U[1, r]
    for i from 1 to n do
        for r from 0 to R do
            if (i == c[i]) then
                M[1, r] := true;
                U[1, r] := true;
            else if (M[i, r] == 0) then
```

```
            M[1, r] := true;
            U[1, r] := false;
        else
            M[1, r] := false;
            U[1, r] := false;
        endif
    endfor
    // calcola i restanti elementi delle tabelle
    for i:=2 to n do
        for r := 0 to R do
            if ( r >= c[i] ) then
                M[i, r] := M[i-1, r] || M[i-1, r-c[i] ];
                U[i, r] := M[i-1, r-c[i] ];
            else
                M[i, r] := M[i-1, r];
                U[i, r] := false; // non usiamo la moneta i-esima
            endif
        endfor
    endfor

    // costruzione della soluzione (monete selezionate)
    if ( M[n, R] == true ) then
        int i := n; int r := R;
        while ( r > 0 ) do
            if ( U[i, r] == true ) then
                print "uso la moneta" i;
                r := r - c[i];
            endif
            i := i - 1;
        endwhile
    else print "nessuna soluzione";
    endif
    return M[n, R];
```

La programmazione *greedy* non sempre conduce ad una soluzione ottima ma, se lo fa, è da preferire alla programmazione dinamica perché più efficiente. Quest'ultima è onerosa per definizione, dovendo costruire le soluzioni a tutti i sottoproblemi, anche quelli non necessari alla costruzione della soluzione.

Esercizio del palinsesto

Un'emittente televisiva deve organizzare il palinsesto di una giornata di 24 ore (1440 minuti). L'emittente dispone di una lista non vuota di durate in minuti degli n programmi disponibili.

1. scrivere un algoritmo di programmazione dinamica che restituisce `true` se e solo se esiste un sottoinsieme degli n programmi disponibili la cui durata complessiva sia esattamente di 1440 minuti
2. determinare il costo computazionale di tale algoritmo
3. raffinare l'algoritmo per individuare anche i programmi che fanno parte della soluzione

Definiamo una matrice $B[1..n, 0, .., 1440]$ tale che $B[i, j] = true$ se e solo se esiste un sottoinsieme dei primi i programmi $1, ..., i$ la cui durata complessiva sia esattamente j. Controlleremo insomma di poter coprire tutti i possibili intervalli di tempo interi da 0 minuti a 24 ore. Se $i = 1$ abbiamo che $B[i, j] = true$ se e solo se $j = d[1]$ oppure $j = 0$. Per ogni $i = 2, ..., n, j = 0, ...1440$, gli altri elementi della matrice sono definiti in questo modo:

$$B[i, j] = \begin{cases} B[i-1, j] \vee B[i-1, j-d[i]] & \text{se } j \geq d[i] \\ B[i-1, j] & \text{altimenti} \end{cases}$$

$B[i-1, j]$ significa coprire j minuti con $i-1$ sottoprogrammi (il programma i-esimo non viene usato) mentre $B[i-1, j-d[i]]$ significa sottrarre a j la durata del programma i-esimo, che viene considerato. Rappresentiamo lo pseudocodice dell'algoritmo soluzione al punto 1:

```
Algoritmo PALINSESTO(array d[1..n] di int) -> bool
    array B[1..n, 0..1440] di bool;

// Inizializzazione di B[1, j]
for j := 0 to 1440 do
```

```
    if ( j == 0 || j == d[1] ) then
        B[1, j] := true;
    else
        B[1, j] := false;
    endif
endfor

// Riempimento della matrice B
for i := 2 to n do
    for j := 0 to 1440 do
        if ( j >= d[i] ) then
            B[i, j] := B[i-1, j] || B[i-1, j-d[i]];
        else
            B[i, j] := B[i-1, j];
        endif
    endfor
endfor

return B[n, 1440];
```

L'algoritmo calcola tutti i valori $B[i, j]$ e restituisce $B[n, 1440]$. Le operazioni nei cicli sono lineari e sono ripetute 1440 volte che è una costante. L'algoritmo è dunque $\Theta(n)$.

Ecco una possibile soluzione del terzo punto. Dobbiamo tenere traccia dei programmi che vengono selezionati per ottenere la durata complessiva di 1440 minuti. Per fare questo, possiamo utilizzare una matrice ausiliaria `S[1..n, 0..1440]` che indica se il programma `i` è stato incluso per ottenere la durata `j`. In particolare, `S[i, j]` sarà `true` se il programma `i` è incluso nella soluzione per ottenere la durata `j`, altrimenti sarà `false`.

```
Algoritmo PALINSESTO(array d[1..n] di int) -> (bool, array di int)
    array B[1..n, 0..1440] di bool;
    array S[1..n, 0..1440] di bool
    array soluzione[1..n] di int;
    int count = 0;

// Inizializzazione di B[1, j]
for j := 0 to 1440 do
    if ( j == 0 || j == d[1] ) then
        B[1, j] := true;
        S[1, j] := (j == d[1]); // Se j == d[1], allora il programma 1 è incluso
    else
        B[1, j] := false;
        S[1, j] := false;
    endif
endfor

// Riempimento delle matrici B e S
for i := 2 to n do
    for j := 0 to 1440 do
        if ( j >= d[i] ) then
            if ( B[i-1, j] ) then
                B[i, j] := true;
                S[i, j] := false; // Il programma i non è incluso
            else if ( B[i-1, j-d[i]] ) then
                B[i, j] := true;
                S[i, j] := true; // Il programma i è incluso
            else
                B[i, j] := false;
                S[i, j] := false;
            endif
        else
            B[i, j] := B[i-1, j];
            S[i, j] := false;
        endif
```

```
    endfor
endfor

// Se esiste una soluzione, ricostruisci i programmi inclusi
if ( B[n, 1440] ) then
    int j = 1440;
    for i := n to 1 do
        if ( S[i, j] ) then
            count += 1;
            soluzione[count] := i;
            j -= d[i];
        endif
    endfor
    return (true, soluzione[1..count]);
else
    return (false, []);
endif
```

Il costo computazionale dell'algoritmo rimane $\Theta(n \cdot 1440)$, poiché dobbiamo riempire le matrici `B` e `S` per ogni combinazione di `i` e `j`. La ricostruzione della soluzione ha un costo lineare rispetto al numero di programmi, quindi non influisce significativamente sul costo complessivo.

Esercizio dell'ascensore

Un gruppo di $n > 0$ persone deve salire su un ascensore che può sostenere un peso massimo di C kg. Indichiamo con $p[1], \ldots, p[n]$ i pesi in kg delle n persone. Il vettore non è ordinato e i pesi sono numeri interi.

1. scrivere un algoritmo di programmazione dinamica che restituisca il numero massimo di persone che possono salire contemporaneamente senza superare la capacità C dell'ascensore
2. determinare il costo computazionale di tale algoritmo
3. raffinare l'algoritmo per individuare anche le persone che faranno parte del sottogruppo che sale

Costruiamo una matrice $N[i, c]$ i cui elementi indicano il massimo numero di persone scelte tra le prime i che è possibile caricare in un ascensore avente portata massima di c kg per $i = 1, ..., n$ $c = 0, ..., C$. Per una sola persona si ha che $N[1, c]$ è uguale a 1 se e solo se $c \geq p[1]$, quindi

$$N[1, c] = \Big\{ 1 sec \geq p[1] 0 altrimenti$$

Nel caso generico:

$$N[i, c] = \begin{cases} \max\{N[i-1, c],\ N[i-1, c - P[i]] + 1\} & \text{se } c \geq p[i] \\ N[i-1, c] & \text{altrimenti} \end{cases}$$

$N[i-1, c]$ significa scartare la persona i-esima, $N[i-1, c-P[i]] + 1$ significa accettarla sull'ascensore.

La tabella si compila in tempo $\Theta(nC)$ e il risultato finale è scritto nella cella $N[n, C]$.

Strategia *greedy*

Si tratta di una strategia euristica *top-down* per risolvere problemi di ottimizzazione. L'idea è scegliere soluzioni ottime ai sottoproblemi sperando di arrivare a una soluzione globalmente ottima. Per dimostrare l'ottimalità della soluzione greedy bisogna dimostrare che:

- la soluzione ottima è composta di soluzioni ottime in tutti i suoi sottoproblemi (sottostruttura ottima)
- la scelta ottima localmente non pregiudica il raggiungimento dell'ottimo globale (proprietà della scelta golosa)

Costruiamo la strategia a partire da un esempio. Immaginiamo di voler trovare il cammino più corto per andare da Napoli a Torino. Gli ingredienti sono:

- l'insieme dei candidati (le città da cui passare)
- l'insieme dei candidati già esaminati
- una funzione obiettivo da minimizzare o massimizzare (la lunghezza del cammino Napoli-Torino e tre funzioni:
- `ammissibile`: verifica se un insieme di candidati rappresenta una soluzione (ovvero se un insieme di città è un cammino da Napoli a Torino)
- `ottimo`: verifica se un insieme di candidati è soluzione ottima (se è il cammino Napoli-Torino più breve)
- `seleziona`: indica quale dei candidati non ancora esaminati è al momento il più promettente

```
algoritmo paradigmaGreedy(insieme di candidati C) -> soluzione
    S <- ∅
    while ((not ottimo(S))) and (C != ∅)) do
        x <- seleziona(C)
        C <- C - {x}
        if (ammissibile(S U {x})) then S <- S U {x}
    if (ottimo(S)) then return S
    else errore("non ho trovato soluzioni")
```

L'algoritmo viene detto *greedy* perché sceglie sempre il candidato più promettente.

Problema di sequenziamento

Un *server* (CPU, *web server* o anche un operatore umano) deve servire $n = cost.$ clienti. Il servizio richiesto dal cliente i-esimo ha tempo di erogazione t_i e tempo d'attesa $T(i)$. Vogliamo minimizzare il tempo di attesa medio:

$$T_{avg} = \frac{T}{n} = \frac{1}{n}\sum_{i=1}^{n} T(i)$$

e quindi

$$T = \sum_{i=1}^{n} T(i)$$

Supponiamo di avere 3 clienti: - $t_1 = 50\ msec$ - $t_2 = 100\ msec$ - $t_3 = 3\ msec$

Osserviamo i 6 possibili ordinamenti:

Ordine	T
1 2 3	50 + (50+100) + (50 + 100 +3) msec = 353 msec
1 3 2	50 + (50+3) è (50 + 3 + 100) msec = 265 msec
2 1 3	100 + (100 + 50) + (100 + 50 + 3) msec = 403 msec
2 3 1	100 + (100 + 3) + (100 + 3 + 50) msec = 356 msec
3 1 2	3 + (3 + 50) + (3 + 50 + 100) msec = 209 msec
3 2 1	3 + (3 + 100) + (3 + 100 + 50) msec = 259 msec

Il seguente algoritmo genera l'ordine di servizio in maniera incrementale secondo una strategia *greedy*. Supponiamo di aver deciso di sequenziare i clienti: $i_1, i_2, \ldots, i_m$. Se adesso decidiamo di servire il cliente j, il tempo totale di servizio diventa

$$t_{i1} + t_{i2} + ... + t_{im} + t_j.$$

La scelta *greedy* consiste, ad ogni passo j, nel servire la richiesta più breve tra quelle rimanenti. Lo pseudocodice della soluzione assume dunque la forma seguente:

```
algoritmo sequenziamento(array C) -> soluzione
    S = {}
    C = tempi (durata) t_j per j=1..n dei servizi richiesti
    Ordina C in modo non decrescente
    for j=1 to n do
        S = S U {j}
    return S
```

Il ciclo `for`contiene un inserimento a tempo costante quindi complessivamente è a costo lineare nel numero di clienti. L'ordinamento di *preprocessing* ha costo $n \log n$. Complessivamente è quest'ultimo il costo a prevalere nell'algoritmo, quindi complessivamente `sequenziamento` ha costo $O(n \log n)$.

Esempio: problema del resto con algoritmo greedy

Una strategia *greedy* non garantisce sempre l'ottimalità della soluzione prodotta. Prendiamo, ad esempio, il problema di un distributore automatico che deve restituire un certo resto R utilizzando il minor numero di monete possibile. Supponiamo di avere a disposizione monete da 1, 5, 10, 20 e 50 centesimi di euro.

Il problema con una strategia greedy può essere descritto come segue: l'insieme C dei candidati è costituito da un insieme finito di monete da 1, 5, 10, 20 e 50 centesimi di euro, non ordinati. La funzione $val(S)$ rappresenta il numero di monete nella soluzione S. La funzione $cost(x)$ indica il numero di monete nella soluzione S necessarie per restituire il resto R. La funzione $scelte(x)$ rappresenta la scelta di monete che minimizza il valore di x. La funzione ottima è quella che minimizza il valore delle monete scelte in modo da ottenere esattamente il resto R.

È riportato di seguito lo pseudocodice della soluzione:

```
algoritmo distributoreResto(C(resto,R) : soluzione/soluzione
C -> monete contenute nel vettore di distribuzione
R -> resto da restituire
while ((valore(S) != R) and (C != 0)) do
x = moneta di valore più elevato in C
C = C - x;
if (valore(S + {x}) <= R) then S = S U {x}
if (valore(S) = R) then return S come resto esatto
else return S come resto parziale
endwhile
```

Non sempre l'algoritmo distributoreResto è in grado di restituire il resto esatto. Osserviamo un esempio di funzionamento corretto: - $C = (50, 50, 20)$ ed $R = 70 : S = (50, 20)$, $valore(S) = 70 = R$ Si riporti invece un esempio di funzionamento subottimale: - $C = (50, 20, 20, 20, 5)$ ed $R = 65 : S = (50, 5)$, $valore(S) = 55$ L'errore sta al primo passo: viene fatta la scelta sbagliata $x = 50$ che non può mai essere disfatta; non utilizzando la moneta da 50, potremmo restituire il resto esatto

Problema dell'ascensore con strategia greedy

Un gruppo di $n > 0$ persone deve salire su un ascensore che può sostenere un peso massimo di C kg. Indichiamo con $p[i] \dots p[n]$ i pesi (in kg) delle n persone. I vettori non sono ordinati e i pesi sono numeri interi. È necessario sviluppare un algoritmo greedy che restituisca il numero massimo di persone che possono salire contemporaneamente senza superare la capacità C dell'ascensore. Successivamente, è importante determinare il costo computazionale dell'algoritmo descritto al punto 1. Infine, è utile raffinare l'algoritmo del punto 1 per individuare anche le persone che fanno parte del sottogruppo che sale.

Per la selezione globale, si deve considerare che le persone che possono entrare non devono superare la portata C dell'ascensore. È riportato di seguito lo pseudocodice per la soluzione:

```
Algoritmo AscensoreGlobale(int C, array p[1..n] di int ) : int
OrdinaPresenti(p);
while (n && p[1] <= C) do
C = C - p[1];
n = n - 1;
endwhile
return C;
```

Il costo dell'algoritmo è dominato dal costo dell'operazione di ordinamento, che è $O(n \log n)$ utilizzando un algoritmo ottimale che impiega un ordinamento per confronto. Se si considera un algoritmo che sfrutta il fatto che i pesi siano interi, il costo diventa lineare, ovvero $O(n)$.

Problema dei cammini minimi

Il *costo di cammino minimo* da un vertice u ad un vertice v è definito nel seguente modo:

$$\delta(u,v) = \begin{cases} \min\{W(p)\} & \text{se esistono cammini } p \text{ da } u \text{ a } v \\ \infty & \text{altrimenti} \end{cases}$$

Un *cammino minimo* da u a v è un cammino p da u a v di costo

$$W(p) = \delta(u,v).$$

Si definisce *problema dei cammini minimi* il calcolo dei cammini minimi. Esistono quattro versioni del problema:

1. determinare i cammini minimi da un'unica sorgente a tutti gli altri vertici
2. determinare i cammini minimi da ogni vertice ad un'unica destinazione
3. determinare i cammini minimi da un'unica sorgente ad un'unica destinazione
4. determinare i cammini minimi tra tutte le coppie di vertici

Si determinerà soltanto la soluzione della prima formulazione del problema. La seconda si ricava dal primo calcolandone il *grafo specchio*. La terza si può risolvere usando la soluzione della prima, perché non si conosce alcun algoritmo asintoticamente migliore. Anche la quarta quarta si può risolvere usando la soluzione della prima per ogni vertice del grafo, ma in genere è risolvibile più efficientemente in altri modi.

In alcuni casi il costo degli archi può essere negativo. Questo diventa un problema per l'algoritmo dei cammini minimi quando nel grafo si formano cicli il cui costo complessivo sia negativo. Se u è un vertice raggiungibile da s con un cammino p passante per un vertice v di un ciclo negativo allora esistono cammini da s a u di costi sempre minori e il costo di cammino minimo $\delta(s,u)$ non è definito. In questo caso poniamo $\delta(s,u) = -\infty$.

Esempio

Vogliamo calcolare i costi per muoversi da s agli altri nodi. diagram Il grafo superiore presenta un ciclo negativo di costo -3. Anche quello inferiore ha costo negativo ma non è raggiungibile da s.

In genere ci interessa calcolare non solo i costi dei cammini minimi dalla sorgente s ad ogni vertice del grafo ma anche i cammini minimi stessi. Siccome i cammini minimi hanno sottostruttura ottima possiamo rappresentarli aumentando ogni vertice con un puntatore `p[v]` che punta al vertice precedente in un cammino minimo da s a v.

Lemma: Limite superiore per i costi di cammino minimo

Per ogni arco (u,v) vale la diseguaglianza:

$$\delta(s,v) \leq \delta(s,u) + W(u,v).$$

Dimostrazione:

- se u non è raggiungibile da s allora: $\delta(s,u) = \infty e \delta(s,v) \leq \infty + W(u,v)$ banalmente
- se u è raggiungibile da s allora $\delta(s,u) + W(u,v)$ è il costo di un cammino da s a v ed è quindi maggiore o uguale di $\delta(s,v)$.

Corollario: scomposizione dei costi di cammino minimo

Se p è un cammino minimo da s ad un vertice v diverso da s ed u è il vertice che precede v nel cammino allora

$$\delta(s,v) = \delta(s,u) + W(u,v).$$

La dimostrazione è conseguenza della sottostruttura ottima: $\delta(s,v) = W(p) = \delta(s,u) + W(u,v)$.

Tecnica del rilassamento

Gli algoritmi che studieremo per il problema dei cammini minimi usano la tecnica del rilassamento. Aggiungiamo ad ogni vertice v del grafo un campo `d[v]` che rappresenta una stima di cammino minimo: durante tutta l'esecuzione dell'algoritmo un limite superiore per $\delta(s,v)$ mentre alla fine è proprio uguale a $\delta(s,v)$.

L'inizializzazione dei campi `p[v]` e `d[v]` è la stessa per tutti gli algoritmi:

```
Inizializza(G, s, d, p)    // G grafo pesato sugli archi
    for ogni v in V[G] do
        p[v] <- nil
        d[v] <- infty
    d[s] <- 0
```

Il *rilassamento di un arco* (u, v) consiste nel controllare se è possibile migliorare il cammino finora trovato per v (e quindi la stima `d[v]`) allungando il cammino trovato per u con l'arco (u, v).

```
Rilassa(G, u, v, d, p, W)
    if d[v] > d[u] + W(u,v) then
        d[v] <- d[u] + W(u,v)
        p[v] <- u
```

Lemma 1: effetto del rilassamento

Dopo aver eseguito `Rilassa(G,u,v)` vale la diseguaglianza

$$d[v] \leq d[u] + W(u, v),$$

ovvero le stime `d[v]` sono monotone non crescenti.

Dimostrazione

Se $d[v] > d[u] + W(u, v)$ prima del rilassamento, viene posto $d[v] = d[u] + W(u, v)$. Se invece vale la disuguaglianza $d[v] \leq d[u] + W(u, v)$ prima del rilassamento, non viene fatto nulla e quindi tale condizione rimane verificata anche dopo il passaggio di rilassamento.

Lemma 2: nivariante del rilassamento

Dopo l'inizializzazione per ogni vertice v vale la disuguaglianza

$$d[v] \leq \delta(s, v),$$

che rimane verificata anche dopo un numero qualsiasi di rilassamenti. Inoltre, se a un certo punto $d[v] = \delta(s, v)$, il suo valore non può più variare.

Dimostrazione

Dopo l'inizializzazione, $d[s] = 0 \leq \delta(s, s)$ (in quali casi vale il $>$ stretto?) e per ogni altro vertice $d[v] = \infty \geq \delta(s, v)$. Se $d[v]$ non viene modificata durante l'esecuzione di `Rilassa(G,u,v)` la disuguaglianza resta ancora vera. Se $d[v]$ viene invece modificata, allora $d[v] = d[u] + W(u, v)$. Siccome $d[u]$ non è stata modificata vale la disuguaglianza $d[u] \geq \delta(s, u)$ e quindi per il limite superiore dei costi di cammino minimo:

$$d[v] \geq \delta(s, u) + W(u, v) \geq \delta(s, v)$$

Infine, dato che il valore di $d[v]$ può soltanto diminuire e $d[v] \geq \delta(s, v)$ se $d[v] = \delta(s, v)$ allora il suo valore non può più variare.

Lemma 3: correttezza di $d[v]$ per vertici non raggiungibili

Dopo l'inizializzazione per ogni vertice v non raggiungibile da s vale la disuguaglianza $d[v] = \delta(s, v)$, ed essa rimane vera anche dopo un numero qualsiasi di rilassamenti.

Dimostrazione

Dopo l'inizializzazione $d[v] = \infty$ per ogni vertice diverso da s, se v non è raggiungibile da s allora $\delta(s, v) = \infty = d[v]$ e per l'invariante del rilassamento $d[v]$ non può più cambiare.

Lemma 4: estensione della correttezza di $d[v]$ per vertici raggiungibili

Se (u, v) è l'ultimo arco di un cammino minimo da s a v e $d[u] = \delta(s, u)$ prima di eseguire il rilassamento dell'arco (u, v), allora dopo il rilassamento $d[v] = \delta(s, v)$.

Dimostrazione

Dopo il rilassamento $d[v] \geq \delta(s, u) + w(u, v)$, siccome (u, v) è l'ultimo arco di un cammino minimo:

$$\delta(s, v) = \delta(s, u) + w(u, v)$$

e quindi

$$d[v] \leq \delta(s, v)$$

Per l'invariante del rilassamento $\delta(s, v)$ è anche un limite inferiore di $d[v]$, per cui vale necessariamente l'uguaglianza:

$$d[v] = \delta(s, v).$$

Possiamo concludere che qualsiasi algoritmo che esegua l'inizializzazione ed una sequenza di rilassamenti per cui alla fine $d[v] = \delta(s, v)$ per ogni vertice v calcola correttamente i cammini minimi. Esistono due algoritmi classici di questo tipo, l'algoritmo di Dijkstra l'algoritmo di Bellman e Ford. L'algoritmo di Dijkstra richiede che i pesi degli archi non siano negativi mentre quello di Bellman-Ford funziona anche nel caso generale.

Algoritmo di Dijkstra

L'algoritmo di Dijkstra è una procedura *greedy* che risolve il problema dei cammini minimi da una singola sorgente per pesi non negativi. Si può adattare anche a pesi negativi normalizzando la funzione di somma dei pesi. Utilizza un insieme S di vertici i cui pesi dei percorsi minimi sono già stati determinati. L'algoritmo seleziona a turno il vertice u in $V \setminus S$ col minimo valore $d[u]$, inserisce u in S, e rilassa tutti gli archi uscenti da u.

Segue lo pseudocodice dell'algoritmo:

```
Dijkstra(G,s,d,p,W)
    Inizializza(G,s,d,p)
    S = []
    Q = V(G)   // Coda (di priorità)
    while ( Q != [] )
        u = extract_Min(Q) //scelta greeedy
        S = S U {u}
            for each vertice v adiacente a u
                relax(u,v,d,p,W)

relax(u,v,d,p,W)
    if d[v] > d[u] + w(u,v)
    then decrease_key(d[v], d[u] + w(u,v))
        p[v] = u
```

Esempio di esecuzione

Primo passo: scegliamo s come vertice sorgente.

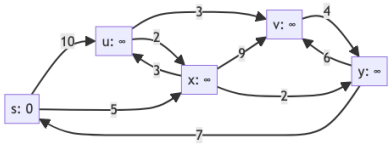


Figura 2: diagram

Secondo passo: s viene estratto da Q ed i vertici adiacenti x ed u vengono «rilassati» (le frecce rosse indicano i predecessori nel cammino minimo).

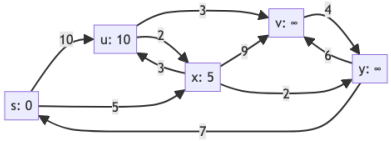


Figura 3: diagram

Terzo passo: x viene estratto da Q e i vertici adiacenti u, v e y vengono «rilassati»

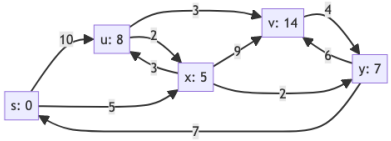


Figura 4: diagram

Quarto passo: y viene estratto da Q ed il vertice adiacente v viene «rilassato»

Quinto passo: u viene estratto da Q ed il vertice adiacente v viene «rilassato»

Sesto passo: v viene estratto da Q. La lista dei predecessori ora definisce il cammino minimo da s per ogni nodo

Tempo di esecuzione Il tempo di esecuzione dipende dall'implementazione della coda di priorità. Differenti implementazioni danno differenti costi per le operazioni sulla coda. Ad esempio, `extract_Min` viene eseguita $O(|V|)$

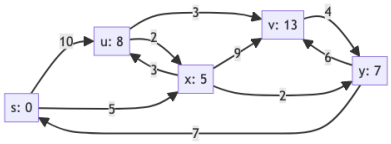


Figura 5: diagram

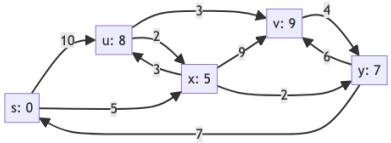


Figura 6: diagram

volte, mentre `relax(decrease_key)` viene eseguita $O(|E|)$ volte. Il tempo totale è dunque

$$|V|T_{extract-Min} + |E|T_{decrease\_key}.$$

Confrontiamo l'uso di array o di heap binari:

Coda a priorità	$T_{extract-Min}$	$T_{decrease\_key}$	Tempo totale
Array	$O(\|\|V\|\|)$	$O(1)$	$O(\|\|V\|\|^2)$
Heap binario	$O(\log \|\|V\|\|)$	$O(\log \|\|V\|\|)$	$O((\|\|V\|\| + \|\|E\|\|) \log \|\|V\|\|)$

Se G è denso, è preferibile l'*array*.

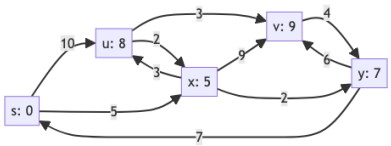


Figura 7: diagram

Sviluppo agile e API-lead

I quattro fattori essenziali studiati dall'ingegneria del software sono:

1. tempo
2. qualità
3. risorse
4. ambito (il più facile da controllare)

I metodi di sviluppo possono dividersi in tre principali categorie: *waterfall* (pianificato, lineare), *spiral* (pianificato, iterativo) e *agile* (non pianificato, *test-driven*).

Sviluppo *agile*

Nella metodologia *agile* lo sviluppo è visto come fortemente comunicativo e legato alle persone più che ai processi. La documentazione è vista come meno importante rispetto alle *release* incrementali di *software* funzionante. È forte l'impiego di *best practice* quali la programmazione in coppia, lo sviluppo basato sui test e l'integrazione continua, a scapito di processi pesanti e rigorosi.

I quattro punti principali del *Manifesto Agile* (Beck et al. 2001): sono

1. individuals and interactions over processes and tools
2. working software over comprehensive documentation
3. customer collaboration over contract negotiation
4. responding to change over following a plan

Per esteso, il manifesto si espande in dodici principi:

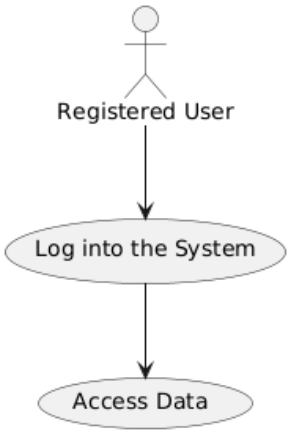
1. soddisfazione del cliente attraverso la consegna precoce e continua di *software* funzionante
2. adattamento a requisiti in cambiamento durante il processo di sviluppo
3. consegna frequente di *software* funzionante
4. collaborazione giornaliera tra *stakeholder* aziendali e sviluppatori durante il progetto
5. supportare, fidarsi di, e motivare le persone coinvolte
6. abilitare interazioni faccia a faccia
7. il *software* funzionante è la misura principale di progresso
8. le procedure agili promuovono lo sviluppo sostenibile
9. l'eccellenza tecnica e la buona progettazione migliorano l'agilità
10. semplicità - sviluppa solo quanto basta per portare a termine il lavoro adesso
11. le migliori architetture, i migliori requisiti e progetti emergono da squadre auto-organizzate
12. auto-miglioramento: le squadre riflettono regolarmente su come essere più efficaci

Pianificazione (*planning game)*

La pianificazione si basa sull'utilizzo delle *user stories*, ovvero una rappresentazione dei casi d'uso che segua lo schema seguente:

> As a `user`, I `want to`, `so that`.

Le *user stories* possono essere trasformate in casi d'uso da schematizzare regolarmente tramite un diagramma UML dei casi d'uso. Ad esempio, la storia utente: `as a registered user, I want to log into the system to access my data.` diventa:



È possibile costruire infine una struttura che divida lo sviluppo in *epic* che raccolgono *task* legati alle storie:

Figura 8: diagram

Testing

Il *testing* segue la filosofia del *test-driven development*: si parte dalla costruzione dei test, con *tutte* le funzionalità da implementare. Tutti i test inizialmente falliscono per mancanza del metodo o dell'unità da testare, ma agiscono in questo modo da promemoria e guida per lo sviluppo. I test stessi possono essere interpretati come una forma di documentazione.

System metaphor

Il concetto di *system metaphor* nello sviluppo agile è un modo per descrivere e capire l'architettura e il progetto di un sistema software utilizzando un'analogia semplice e condivisa. Questa metafora serve come linguaggio comune per la squadra di sviluppo, gli *stakeholder* e gli utenti, aiutando tutti a capire la struttura del sistema, dei componenti e delle interazioni tra parti. La metafora semplifica i concetti, guida il progetto, ne facilita l'uniformita e ne favorisce l'evoluzione. Tutti sono coinvolti e interessati nello sviluppo del sistema. È importante che l'architettura di base e l'intero sistema siano chiari a tutti, e che tutti se ne sentano responsabili. L'architettura non è una *pipeline* in *silos* come nello sviluppo tradizionale, ma una *tavola rotonda* dove i ruoli precedentemente rappresentati dai *silos* (*product owner*, *project manager*, *designer*, sviluppatore, *tester*) convivono parallelamente in un *core team*, e possono perfino scambiarsi i ruoli nel corso del tempo.

Programmazione in coppia

Il pair programming nella metodologia agile consiste nella collaborazione di due programmatori: il *driver*, che scrive il codice, e il *navigator*, che revisiona e guida il processo. Questo approccio migliora la concentrazione, riduce gli errori e aumenta la qualità del codice grazie alla continua revisione e al confronto costante tra i due sviluppatori.

Timeboxing

Il *timeboxing* nella metodologia *agile* consiste nella divisione del tempo in intervalli di durata prefissata, alla fine dei quali ci si ferma, indipendentemente dal risultato ottenuto. Questo approccio è ampiamente utilizzato in metodologie come *Scrum* ed *eXtreme Programming*, specialmente per attività come il *brainstorming*. Un esempio pratico è la *tecnica del pomodoro*, che prevede 25 minuti di lavoro seguiti da 5 minuti di pausa. Il *timeboxing* aiuta a mantenere la concentrazione, a gestire meglio il tempo e a ridurre la procrastinazione, migliorando così l'efficienza e la produttività del team.

Continuous integration

La *Continuous Integration* (CI) è una pratica fondamentale nella metodologia *agile* che mira a evitare il cosiddetto "*big bang disaster*", ovvero la situazione in cui nulla compila e tutti i test falliscono durante la fase di integrazione

finale. Questo approccio prevede che ogni modifica al codice venga automaticamente testata e compilata su un *server* collegato al repository remoto. In questo modo, eventuali problemi vengono identificati e risolti tempestivamente, riducendo il rischio di errori critici in fase di rilascio.

DevOps è una metodologia che promuove la collaborazione e l'integrazione tra il team di sviluppo, il team di garanzia qualità e il team di operazioni di deployment. Questo approccio mira a ridurre i *silos* organizzativi e a migliorare la comunicazione e la cooperazione tra i vari team, accelerando così il ciclo di vita dello sviluppo software. Automatizzando il processo di rilascio come parte della CI, si garantisce che il codice sia sempre pronto per la produzione, migliorando così l'efficienza e la qualità del software.

Un altro aspetto cruciale della CI è l'*Infrastructure-as-Code* (*IaC*), che consiste nell'utilizzo di codice per gestire e configurare gli strumenti di management della configurazione. Questo approccio permette di automatizzare ulteriormente il processo di integrazione continua, rendendo la gestione delle infrastrutture più efficiente e meno soggetta a errori umani. In sintesi, la CI non solo migliora la qualità del codice, ma anche la collaborazione tra i team e l'efficienza operativa.

Problemi con le metodologie agili

Le metodologie agili, sebbene offrano numerosi vantaggi, presentano anche alcune sfide significative. Uno dei principali problemi è la mancanza di obiettivi chiaramente definiti, che può portare al fenomeno del *feature creep*, ovvero l'aggiunta continua di nuove funzionalità che allunga indefinitamente il progetto. Questo può rendere difficile determinare quando un progetto è effettivamente completato. Inoltre, le metodologie agili richiedono un cliente singolo e altamente coinvolto, disposto a partecipare attivamente al processo di sviluppo. Questo livello di impegno non è sempre facile da ottenere, specialmente in progetti complessi o con clienti multipli. Un altro aspetto critico è l'utilizzo di team piccoli, che possono essere più agili ma anche più vulnerabili a problemi di risorse umane, come la perdita di membri chiave del team. Un'altra problematica legata al personale è la difficile, seppure possibile, scalabilità del metodo *agile* a team di grandi dimensioni. Infine, l'integrazione delle tecniche agili spesso avviene in modo "all-or-nothing", con una forte codipendenza tra le varie pratiche. Questo significa che l'adozione parziale delle metodologie agili può non portare ai benefici attesi, rendendo necessaria una transizione completa per ottenere risultati significativi.

Prodotti e progetti

Tradizionalmente, il lavoro nello sviluppo software si concentrava su progetti ad-hoc commissionati da singoli clienti. Questo approccio comportava la creazione di soluzioni personalizzate per soddisfare esigenze specifiche. Nel corso degli anni è emersa una tendenza verso lo sviluppo di prodotti generici, vendibili su un mercato più ampio. All'interno delle aziende spesso c'è personale dedicato che cerca opportunità per creare prodotti a valore aggiunto. Questo processo può includere il riutilizzo di componenti già sviluppate in passato, ottimizzando così le risorse e riducendo i tempi di sviluppo. L'economia delle API (Application Programming Interfaces) ha ulteriormente trasformato il mercato, permettendo l'uso di servizi in abbonamento o a consumo. Questo modello consente alle aziende di offrire funzionalità avanzate e scalabili, facilitando l'integrazione con altre piattaforme e servizi.

La *Software Product Line* (SPL) è un approccio allo sviluppo *software* che si concentra sulla creazione di una famiglia di prodotti simili, i quali condividono una base comune di componenti riutilizzabili. Questo metodo permette di ridurre i costi e i tempi di sviluppo, migliorando al contempo la qualità e la coerenza dei prodotti.

Modello di esecuzione

In passato, i software venivano eseguiti principalmente in locale, o al massimo sul server del cliente, con aggiornamenti periodici installati manualmente. Questo approccio richiedeva un intervento diretto per mantenere il software aggiornato e funzionante. È in seguito emerso il modello di *esecuzione ibrida*, in cui alcune funzionalità aggiuntive vengono servite direttamente dal *server* del venditore. Questo approccio combina l'esecuzione locale con l'accesso a servizi remoti, offrendo maggiore flessibilità e scalabilità. Il modello *Software as a Service* (*SaaS*) rappresenta un ulteriore passo avanti. In questo caso, il software viene eseguito interamente sul server del venditore e accessibile tramite *web app*. Questo approccio elimina la necessità di installazioni locali e aggiornamenti manuali, garantendo che gli utenti abbiano sempre accesso alla versione più recente del software.

Product software management

Le responsabilità del *product manager* includono la ricerca di nuove opportunità di mercato, la vendita ai clienti e la costruzione di un business model sostenibile. Questa figura è inoltre responsabile della gestione della customer experience, assicurandosi che i prodotti soddisfino le esigenze degli utenti e offrano un valore significativo. Questi compiti devono essere svolti tenendo conto delle limitazioni tecniche, bilanciando le aspettative dei clienti con le capacità tecnologiche dell'azienda.

API

La definizione di *Application Programming Interface* (*API*) è stata introdotta da Cotton e Greatorex nel 1968 come *endpoint* esposti tramite i quali è possibile accedere a un servizio. Inizialmente, le API erano principalmente utilizzate per facilitare l'interazione tra diversi componenti *software* all'interno di un unico sistema. Negli ultimi anni, le API sono diventate esse stesse prodotti a tutti gli effetti, producibili, vendibili e componibili. Questo cambiamento ha portato alla nascita dell'*economia delle API*, in cui le aziende offrono accesso a funzionalità e servizi tramite API, spesso in modalità di abbonamento o a consumo. Le API sono ora considerate componenti fondamentali per l'innovazione e la scalabilità, permettendo alle aziende di integrare facilmente nuove funzionalità e di creare soluzioni composite che rispondono a esigenze specifiche del mercato.

AMMD (Agile Model Driven Development)

AMMD (*Agile Model Driven Development*) è una versione agile del *Model-Driven Development* (*MDD*), che combina i principi dell'agilità con la modellazione guidata dai modelli. Come molti altri metodi agili, AMMD è iterativo e incrementale, permettendo di adattarsi rapidamente ai cambiamenti e alle esigenze del progetto. Il principio base di AMMD è quello di affiancare una modellazione semplice e sufficiente alla scrittura del codice, garantendo che il modello venga aggiornato in tempo reale durante le iterazioni dello sviluppo.

Uno degli aspetti distintivi di AMMD rispetto ad altre tecniche simili è che non richiede l'uso di un linguaggio di modellazione particolare. Questo permette ai team di sviluppo di scegliere gli strumenti e i linguaggi di modellazione più adatti alle loro esigenze specifiche, senza essere vincolati da imposizioni rigide.

Di seguito è riportato uno schema delle principali categorie di modelli utilizzate nell'ambito AMMD, divise per categoria.

Figura 9: diagram

Le principali tipologie di modellazione utilizzate in AMMD includono modellazione strutturale, modellazione dell'uso e modellazione architetturale. La modellazione strutturale si concentra sulla definizione delle interfacce e delle strutture dati del sistema. Un esempio comune è il diagramma delle interfacce, che descrive come i diversi componenti del sistema interagiscono tra loro. Questo tipo di modellazione è fondamentale per garantire che il sistema sia ben strutturato e facilmente mantenibile. La modellazione dell'uso si focalizza su come gli utenti interagiranno con il sistema. Include la creazione di casi d'uso, diagrammi di sequenza e altri artefatti che descrivono i flussi di lavoro e le interazioni tra utenti e sistema. La modellazione dell'uso è cruciale per assicurare che il sistema soddisfi le esigenze degli utenti finali. La modellazione architetturale, infine, si occupa della definizione dell'architettura complessiva del sistema. Include la creazione di diagrammi di componenti, diagrammi di distribuzione e altri artefatti che descrivono come i vari componenti del sistema sono organizzati e come comunicano tra loro. La modellazione architetturale è essenziale per garantire che il sistema sia scalabile, performante e sicuro.

Iterazioni

L'iterazione iniziale, denominata *envisioning*, si concentra sulla concettualizzazione della visione del prodotto e della struttura iniziale. Questa fase dura solitamente pochi giorni. È fondamentale scegliere fin da subito i framework, poiché da essi derivano alcune scelte architetturali essenziali. La prima iterazione successiva all'*envisioning* comprende diverse attività: la modellazione, che richiede ore di lavoro; il model storming, che si svolge in pochi minuti; e lo sviluppo guidato dai test, che occupa diverse ore. Inoltre, viene incluso il *testing* investigativo. Le iterazioni seguenti adottano lo stesso modello della prima. Al termine di ciascuna iterazione, è possibile effettuare una revisione opzionale con il cliente.

Il seguente schema rappresenta il processo di *test-driven development*:

Il *testing investigativo* si focalizza sulla ricerca di *bug* in situazioni estreme, svolgendosi in un'unica sessione intensiva. Questo tipo di *testing* è cruciale per identificare problemi che potrebbero emergere in condizioni operative inusuali.

I requisiti vengono ordinati in una *pila dei requisiti* in base all'urgenza della loro implementazione, permettendo di gestire le priorità in modo efficace. Questo approccio garantisce che i requisiti più critici vengano affrontati per primi, assicurando un progresso coerente e mirato.

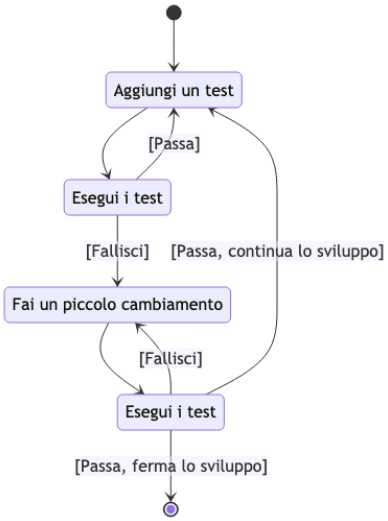


Figura 10: diagram

La *matrice di tracciabilità* dei componenti è una tabella che registra la data e le informazioni relative all'inserimento di ciascun requisito. Questa matrice fornisce una panoramica dettagliata dello stato dei requisiti, facilitando il monitoraggio e la gestione del progetto. Si registrano il nome del requisito; un suo identificativo; il tipo (funzionale / non funzionale); i gradi di priorità, criticità e rischio; la data di inserimento e una breve descrizione. La tabella seguente rappresenta un esempio di matrice di tracciabilità dei requisiti.

nome requisito	id requisito	tipo	priorità	criticità	rischio	data	descrizione	fonte	implementazione	requisito padre	requisiti figli
acquisto prodotti	1	funzionale	alta	alta	basso	22/06/2005	deve permettere ai clienti di fare acquisti online	verbale riunione 22/06/2005			2,3,4,5
registrazione	2	funzionale	alta	alta	alto	22/06/2005	deve permettere al cliente nuovo di registrarsi	verbale riunione 22/06/2005		1	

L'utilizzo della matrice è considerato una *best practice* perché garantisce la tracciabilità sia in avanti che all'indietro. Questo consente di seguire l'evoluzione dei requisiti nel tempo, facilitando la gestione e il controllo del progetto.

Nell'ambito dell'*envisioning* architetturale, è fortemente consigliato produrre un diagramma di *deployment* UML. Questo diagramma fornisce una rappresentazione visiva della distribuzione del sistema sull'*hardware*, aiutando a comprendere meglio la struttura e le interazioni tra i vari componenti.

Casi d'uso

I *casi d'uso* devono essere formulati nel modo più astratto possibile, evitando qualsiasi riferimento alla parte implementativa. Questo approccio permette di concentrarsi sulle interazioni e sui requisiti funzionali senza essere influenzati dalle specifiche tecniche di realizzazione. Possono essere rappresentati come *user story*, con un diagramma formale UML, con una specifica scritta o con più tecniche contemporaneamente.

Esempio CoCoMe

L'esempio CoCoME (Common Component Modeling Example) è un caso di studio ampiamente utilizzato per illustrare la realizzazione di componenti software. È spesso utilizzato come esempio didattico. CoCoME modella un sistema di gestione per un supermercato, coprendo vari aspetti come la gestione degli inventari, le vendite e le operazioni di cassa. Questo esempio è particolarmente utile per dimostrare come i componenti software possano essere progettati e integrati in un sistema complesso. CoCoME adotta un approccio modulare. Il sistema viene suddiviso in componenti distinti che interagiscono tra loro. Ogni componente è responsabile di una specifica funzionalità. Esempi di funzionalità sono la gestione dei prodotti, il processo di vendita e l'interfaccia utente. La modularità facilita la manutenzione e l'aggiornamento del sistema. I componenti possono essere sviluppati, testati e aggiornati indipendentemente gli uni dagli altri. Un aspetto fondamentale di CoCoME è l'uso di interfacce ben definite per la comunicazione tra i componenti. Queste interfacce garantiscono che i componenti possano interagire in modo coerente e prevedibile. Ciò riduce il rischio di errori e migliora l'affidabilità del sistema. Inoltre, l'uso di interfacce standardizzate facilita l'integrazione di nuovi componenti o la sostituzione di componenti esistenti, senza dover modificare l'intero sistema.

Gli standard ANSI/IEEE 1471:2000 e ISO/IEC/IEEE 42010:2011 forniscono una definizione ufficiale dell'architettura *software*, stabilendo linee guida e *best practice* per la progettazione e la documentazione dei sistemi software. Questi *standard* sono ampiamente riconosciuti e utilizzati nella pratica professionale per garantire la qualità e l'efficienza dei sistemi software.

Nell'ingegneria del *software* tradizionale, il processo di sviluppo comporta un passaggio da un singolo problema a molteplici soluzioni possibili, esplorando l'intero spazio delle soluzioni per trovare quella ottimale. Al contrario, l'approccio moderno all'ingegneria del *software* prevede, data una specifica problematica, la selezione tra un numero più ristretto di architetture di riferimento, che vengono poi adattate per creare implementazioni specifiche. Questo metodo semplifica il processo decisionale e permette di concentrarsi su soluzioni già validate e ottimizzate. Un altro principio fondamentale nell'architettura software è quello delle *few interfaces*, che mira a minimizzare il numero totale di comunicazioni tra i moduli. Ogni modulo dovrebbe comunicare con il minor numero possibile di altri moduli, idealmente avvicinandosi al minimo teorico di $n - 1$ collegamenti tra n moduli. Ridurre le interazioni tra i moduli non solo semplifica la struttura del sistema, ma migliora anche la manutenibilità e la gestione delle dipendenze, rendendo il sistema più robusto e meno suscettibile agli errori. Il seguente grafico rappresenta un esempio di interconnessione minimizzata ed un esempio di interconnessione eccessiva:

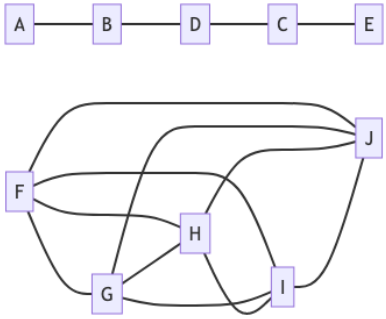


Figura 11: diagram

La *separazione delle responsabilità* consiste nel dividere un'applicazione in funzionalità distinte con la minima sovrapposizione possibile. Questo approccio mira a minimizzare i punti di interazione tra le diverse parti del sistema, ottenendo così un'alta coesione e un basso accoppiamento. Il *principio di singola responsabilità* afferma che ogni componente o modulo deve avere responsabilità solo per una specifica funzionalità o per un'aggregazione di funzionalità strettamente correlate. Questo principio promuove la modularità e la manutenibilità del sistema, rendendo più semplice l'identificazione e la risoluzione dei problemi. Il *principio di minima conoscenza* stabilisce che un componente non deve conoscere i dettagli interni di altri componenti, ma solo la loro interfaccia. Procedere in questo modo riduce la dipendenza tra i componenti e facilita la modifica e l'aggiornamento del sistema senza effetti collaterali indesiderati. Il principio *Don't repeat yourself* (*DRY*) sottolinea l'importanza di evitare la duplicazione di funzionalità tra componenti diversi. Questo approccio mira a ridurre la ridondanza e a migliorare la manutenibilità del codice, rendendo più facile apportare modifiche e aggiornamenti. Il principio *minimize upfront design* è ampiamente applicato nel *design agile*. Questo principio suggerisce di progettare solo ciò che è necessario in quel momento, evitando di investire tempo e risorse in dettagli che potrebbero non essere immediatamente rilevanti. Questo approccio permette di concentrarsi sulle esigenze attuali e di adattarsi rapidamente ai cambiamenti, promuovendo flessibilità e efficienza nello sviluppo del software.

Nella progettazione del *software* si distinguono tre diversi livelli di *pattern*. Gli *idiomi*, o *pattern* a livello di codice, riguardano le convenzioni utilizzate direttamente nel codice sorgente per migliorare la leggibilità e l'efficienza. I *design pattern*, a livello di componente, forniscono soluzioni per problemi comuni nella progettazione, promuovendo la riutilizzabilità e la modularità. Gli *architectural pattern*, a livello di sistema, definiscono la struttura complessiva del sistema, come la stratificazione, che suddivide il sistema in livelli con responsabilità ben definite, facilitando la gestione della complessità.

Il *modello 4+1* è un *framework* di progettazione del *software* che integra quattro diverse prospettive, unite da scenari. La *vista logica* o *funzionale* descrive i requisiti funzionali del sistema e i servizi che esso offre, utilizzando un modello statico che include componenti, classi e interazioni tra componenti. La *vista dei processi*, invece, è un modello dinamico che riguarda l'esecuzione delle componenti, includendo processi, *thread* e interazioni tra entità attive, oltre ai requisiti non funzionali sul comportamento del sistema. La *vista di sviluppo*, o di *implementazione*, utilizza diagrammi statici per guidare il *team* di sviluppo, mostrando unità compilative, *package*, componenti e la distribuzione in unità logiche. La *vista fisica* descrive come le componenti software create si distribuiscono sull'hardware. Infine, gli *scenari* modellano i casi d'uso e i requisiti funzionali, fornendo una visione integrata delle diverse prospettive. Per rappresentare la vista logica si utilizzano diagrammi dei componenti, delle classi e dell'attività. La vista dei processi è rappresentata da diagrammi della macchina a stati, oggetti, componenti,

sequenza e attività. La vista di sviluppo include diagrammi delle classi, dei *package*, sequenza e macchina a stati. La vista fisica è rappresentata dal diagramma di *deployment*. Gli scenari utilizzano diagrammi dei casi d'uso, sequenza e attività per fornire una visione completa delle diverse prospettive del sistema. [comment]: appunti pessimi riassunta con Mistral. Da rivedere assolutamente.

Un componente software incapsula funzionalità e dati, ed è tipicamente specifico per un determinato dominio applicativo. Nella specifica dei componenti, si ragiona in termini di interfacce: interfacce fornite, interfacce richieste (dipendenze) e contratti (obblighi pubblici), che includono precondizioni, postcondizioni e invarianze. L'implementazione di un componente consiste in una struttura di oggetti realizzati (istanze di classi) e algoritmi che implementano la funzionalità dichiarata nella specifica del componente.

Un diagramma dei componenti specifica le interfacce richieste e fornite. Ad esempio, un componente chiamato `ComponentA` può richiedere un'interfaccia chiamata `InterfaceB` e fornire un'interfaccia chiamata `InterfaceA`.

```
@startuml
component ComponentA
ComponentA ---( InterfaceB : Requires
InterfaceA -- ComponentA : Provides
@enduml
```

Le relazioni tra componenti possono essere di diversi tipi:

- *is-part-of*: indica l'appartenenza di un componente a un altro.
- *uses*: indica l'invocazione di un componente da parte di un altro.
- *is-located-with*: indica che due componenti fanno parte dello stesso supercomponente.
- *shares-data-with*: indica che due componenti condividono la memoria.

Un connettore collega due componenti e può assumere diverse forme, come chiamate di procedura, memoria condivisa, metodi remoti, client-server, passaggio di messaggi, accesso a database e multicast asincrono. I connettori possono anche includere wrapper e adattatori. I protocolli definiscono le regole per l'ordinamento delle operazioni di un'interfaccia. Infine, è importante distinguere tra *tier* e *layer*. I *tier* rappresentano livelli fisici, mentre i *layer* rappresentano livelli logici.

Stili architettonici

Uno stile architettonico rappresenta una raccolta denominata di decisioni di design architettonico che sono applicabili in un determinato contesto di sviluppo. Queste decisioni vincolano il design architettonico a un sistema specifico all'interno di quel contesto, promuovendo qualità benefiche in ogni sistema risultante. Gli stili architettonici si manifestano attraverso modelli organizzativi ricorrenti e idiomi, riflettendo una comprensione condivisa e consolidata delle forme di design comuni. Essi sono considerati un segno distintivo di un campo ingegneristico maturo. Secondo Shaw e Garlan, uno stile architettonico è un'astrazione delle caratteristiche ricorrenti di composizione e interazione in un insieme di architetture. Medvidovic e Taylor concordano su questa visione, sottolineando l'importanza degli stili architettonici nel fornire una struttura coerente e riconoscibile ai progetti.

Le architetture basate su oggetti e chiamate di metodi possono operare localmente o remotamente. Nel caso di interazioni remote, viene utilizzato un meccanismo di comunicazione inter-processo, come ad esempio le chiamate di procedura remota (RPC) e l'invocazione di metodi remoti (RMI). Questo stile di interazione è sincrono, il che significa che ogni chiamata di metodo attende il completamento dell'operazione prima di procedere con l'esecuzione successiva.

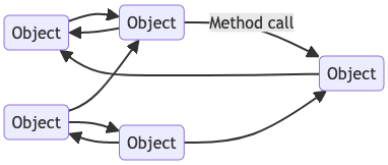


Figura 12: diagram

Un meccanismo di comunicazione inter-processo può essere descritto attraverso due dimensioni fondamentali: la dimensione spaziale e la dimensione temporale. La dimensione spaziale si occupa di due aspetti principali. Il primo è la natura dell'interazione, che può essere *uno-a-uno* o *uno-a-molti*. Il secondo aspetto riguarda il modo in cui le parti interagenti vengono a conoscenza l'una dell'altra. La dimensione temporale, invece, si concentra sul tipo di interazione, che può essere sincrona o asincrona. La combinazione di queste due dimensioni determina diversi stili di interazione e, di conseguenza, diversi stili architettonici, come mostrato in tabella.

	Uno-a-uno	Uno-a-molti
Sincrono	Richiesta/risposta	-
Asincrono	Notifica	Pubblica/iscriviti
Asincrono	Richiesta risposta asincrona	Pubblica/risposte asincrone

Stili di base

Esistono diversi stili architettonici di base, tra cui il modello *Pipe-and-Filter*, l'architettura multilivello, il modello *client-server* e le sue varianti, il modello *broker*, il modello MVC, l'architettura orientata ai messaggi, l'architettura orientata ai servizi e l'architettura esagonale (nota anche come architettura porte e adattatori, spesso utilizzata per i microservizi).

Pipe-and-filter

L'architettura *Pipe-and-Filter* prevede che un flusso di dati, in un formato relativamente semplice, venga passato attraverso una serie di processi, ciascuno dei quali lo trasforma in qualche modo. I dati vengono costantemente alimentati nella *pipeline* e i processi lavorano in modo concorrente. Questo tipo di architettura è estremamente flessibile, poiché quasi tutti i componenti possono essere rimossi o sostituiti, e nuovi componenti possono essere inseriti facilmente.

Una *pipeline* di Big Data su *cloud* segue un modello a cinque stadi. Inizia con un *data lake* che contiene tutti i dati in formati come *stream*, BLOBs (*Binary Large Objects*), CLOBs (*Character Large Objects*) o *file*, ad esempio file JSON. Un esempio di tecnologie *open source* utilizzate in questo contesto è l'ecosistema Apache Spark, che consente di passare dal *data lake* alla *data warehouse* fino all'analisi dei dati. Un esempio pratico è la piattaforma *ENEA PELL smart city* per l'illuminazione pubblica.

L'architettura *Pipe-and-Filter* e i suoi principi di *design* si basano su diversi concetti chiave. Il primo è il principio *divide et impera*, dove i processi separati possono essere progettati indipendentemente. Questo aumenta la coesione funzionale dei processi e riduce l'accoppiamento, poiché ciascun processo ha un solo input e un solo output. I componenti della *pipeline* sono spesso buone astrazioni, nascondendo i dettagli interni e aumentando la riutilizzabilità. I processi possono essere utilizzati in molteplici contesti e spesso è possibile trovare componenti riutilizzabili da inserire nel pipeline. Il sistema è progettato per essere flessibile e testabile, rendendo facile testare i singoli processi. Infine, il *design* difensivo implica un rigoroso controllo degli *input* di ciascun componente, o l'uso del *design by contract*.

Architettura multilivello

L'*architettura multilivello* consente di costruire sistemi complessi sovrapponendo strati a livelli crescenti di astrazione. Ogni livello comunica esclusivamente con il livello immediatamente sottostante e fornisce un'interfaccia ben definita, ovvero un insieme di servizi, al livello immediatamente superiore.

I principi di *design* dell'architettura multilivello si basano su diversi concetti fondamentali. Il principio *divide et impera* permette di progettare i livelli indipendentemente. Questo aumenta la coesione all'interno di ciascun livello. I livelli inferiori, se ben progettati, non hanno conoscenza dei livelli superiori. L'unica connessione tra i livelli avviene tramite l'API, e questo riduce l'accoppiamento. Tale approccio aumenta l'astrazione, poiché non è necessario conoscere i dettagli di implementazione dei livelli inferiori. I livelli inferiori possono essere progettati in modo generico, aumentando la riutilizzabilità. Spesso è possibile riutilizzare livelli costruiti da altri che forniscono i servizi necessari.

L'architettura multilivello offre grande flessibilità. Permette di aggiungere nuove funzionalità basate su servizi di livello inferiore o di sostituire livelli di livello superiore. Isolando i componenti in livelli separati, il sistema diventa più resistente all'obsolescenza. Tutte le funzionalità dipendenti possono essere isolate in uno dei livelli inferiori, favorendo la portabilità. I livelli possono essere testati indipendentemente, facilitando la verifica del sistema. Infine, le API dei livelli sono punti naturali per implementare controlli rigorosi delle asserzioni, promuovendo un design difensivo.

Architettura client-server

L'architettura *client-server* e le sue varianti rappresentano un modello di architettura distribuita in cui almeno un componente assume il ruolo di *server*, che attende connessioni e le gestisce una volta stabilite. Almeno un altro componente assume il ruolo di *client*, iniziando le connessioni per ottenere un servizio. I *server* non conoscono il numero né l'identità dei *client*, mentre i *client* conoscono l'identità del server. Un'ulteriore estensione di questo modello è il pattern *peer-to-peer*, dove il sistema è composto da varie componenti *software* distribuiti su diversi

host, creando un sistema decentralizzato. Un esempio di questa architettura è un *motore di ricerca web*. I *pattern* architetturali coinvolti in questo caso includono una combinazione di livelli, *pipe* e filtri, e *client-server*.

L'architettura distribuita e i suoi principi di *design* si basano su una serie di concetti chiave. Il principio *divide et impera* implica la suddivisione del sistema in processi *client* e *server*, permettendo lo sviluppo indipendente di ciascuno. Il *server* può fornire un servizio coeso ai *client*, aumentando la coesione del sistema. Generalmente esiste un solo canale di comunicazione che scambia messaggi semplici. Questo riduce l'accoppiamento. I componenti distribuiti separati sono spesso buone astrazioni, che nascondono i dettagli interni e aumentano la riutilizzabilità. È spesso possibile trovare *framework* adatti per costruire buoni sistemi distribuiti.

I sistemi distribuiti possono essere facilmente riconfigurati aggiungendo *server* o *client* aggiuntivi, aumentando la flessibilità. La portabilità è migliorata poiché è possibile scrivere *client* per nuove piattaforme senza dover portare il *server*. La testabilità è facilitata poiché *client* e server possono essere testati indipendentemente. Infine, un *design* difensivo implica l'implementazione di controlli rigorosi nel codice di gestione dei messaggi, garantendo la robustezza del sistema.

Architettura MVC

L'architettura *Model-View-Controller* (MVC) è un *pattern* architetturale utilizzato per separare il livello dell'interfaccia utente dagli altri componenti del sistema. Il *modello* contiene le classi sottostanti, le cui istanze sono visualizzate e manipolate. La *vista* contiene gli oggetti utilizzati per rendere l'aspetto dei dati del modello nell'interfaccia utente. Il *controllore* contiene gli oggetti che gestiscono e controllano l'interazione dell'utente con la vista e il modello. Il *pattern* di design *Observable* viene normalmente utilizzato per separare il modello dalla vista.

Un esempio dell'architettura MVC per l'interfaccia utente prevede interazioni triangolari. Nel modello passivo, la vista "tira" i risultati dal controllore. Nel modello attivo, il modello "spinge" i cambiamenti di stato alla vista attraverso un pattern *Observer*, noto anche come *Publish/Subscribe*.

I principi di design dell'architettura MVC includono il modello *divide et impera*, nel quale i tre componenti possono essere progettati in modo relativamente indipendente. Si aumenta così facendo la coesione dei componenti, che è più forte rispetto a una situazione in cui la vista e il controllore fossero combinati in un unico livello dell'interfaccia utente. La riduzione dell'accoppiamento è ottenuta minimizzando i canali di comunicazione tra i tre componenti. L'aumento del riuso è favorito dall'utilizzo estensivo di componenti riutilizzabili per vari tipi di controlli dell'interfaccia utente. La flessibilità è migliorata poiché è generalmente facile modificare l'interfaccia utente cambiando la vista, il controllore o entrambi. La testabilità è facilitata poiché è possibile testare l'applicazione separatamente dall'interfaccia utente.

Confrontando l'architettura MVC con l'architettura a tre livelli, quest'ultima è concettualmente lineare: una regola fondamentale è che il livello *client* non comunica mai direttamente con il livello dati. In un modello a tre livelli, tutte le comunicazioni devono passare attraverso il livello intermedio. Al contrario, l'architettura MVC è triangolare: la vista invia aggiornamenti al controllore, il controllore aggiorna il modello, e la vista riceve aggiornamenti direttamente dal modello (*push*) o dal controllore (*pull*).

Architettura Broker

Il pattern architetturale *Broker* è utilizzato per strutturare sistemi distribuiti con componenti disaccoppiati. I *server* pubblicano le loro capacità, ovvero i servizi e le caratteristiche, a un *broker*. I *client* richiedono un servizio al *broker*, che poi reindirizza il *client* a un servizio appropriato dal suo registro. Questo pattern si basa su connettori di invocazione remota, permettendo a un oggetto di chiamare i metodi di un altro oggetto senza sapere che quest'ultimo è situato remotamente. Esempi di tecnologie di implementazione includono lo standard aperto OMG CORBA (*Common Object Request Broker Architecture*), RPC (*Remote Procedure Call*), Java RMI (*Remote Method Invocation*) e gRPC (*google Remote Procedure Calls*) per invocazioni di servizi remoti. Quest'ultimo si distingue per l'uso di servizi anziché oggetti, messaggi o riferimenti.

Un esempio di questo pattern è l'RMI in Java, dove la JVM lato *client* comunica con la JVM lato *server*, che svolge il ruolo di *broker*. Un altro esempio è gRPC, un'infrastruttura RPC *open source* e versatile sviluppata da Google per connettere servizi. gRPC utilizza HTTP/2 per il trasporto e *Protocol Buffers* come linguaggio di definizione delle interfacce (IDL). Offre funzionalità come autenticazione, *streaming* bidirezionale, controllo del flusso, *binding* bloccanti o non bloccanti, cancellazione e *timeout*.

I principi di *design* dell'architettura Broker includono *divide et impera*, permettendo di progettare gli oggetti remoti in modo indipendente. Questo aumenta la riusabilità, poiché è spesso possibile progettare gli oggetti remoti in modo che possano essere utilizzati anche da altri sistemi. Inoltre, è possibile riutilizzare oggetti remoti creati da altri. La flessibilità è garantita dalla possibilità di aggiornare i *broker* quando necessario, o di far comunicare il *proxy* con diversi oggetti remoti. La portabilità è migliorata poiché è possibile scrivere *client* per nuove piattaforme

mantenendo l'accesso ai *broker* e agli oggetti remoti su altre piattaforme. Infine, un design difensivo implica l'implementazione di controlli rigorosi delle asserzioni negli oggetti remoti.

Architettura ad elaborazione di transazioni

Il *pattern* architetturale di *elaborazione delle transazioni* prevede che un processo legga una serie di ingressi uno alla volta. Ogni ingresso descrive una transazione, ovvero un comando che tipicamente comporta una modifica ai dati memorizzati dal sistema. Un componente chiamato *dispatcher delle transazioni* decide come gestire ciascuna transazione. Il *dispatcher* invia una chiamata di procedura o un messaggio a uno dei componenti responsabili dell'elaborazione della transazione.

I principi di *design* dell'architettura di elaborazione delle transazioni includono *divide et impera*, dove i gestori delle transazioni rappresentano suddivisioni del sistema che possono essere assegnate a ingegneri del *software* separati. I gestori delle transazioni sono unità naturalmente coese, e separare il *dispatcher* dai gestori tende a ridurre l'accoppiamento. La flessibilità è garantita dalla possibilità di aggiungere facilmente nuovi gestori delle transazioni. Un *design* difensivo implica l'implementazione di controlli rigorosi delle asserzioni in ciascun gestore delle transazioni e/o nel *dispatcher*.

Architettura orientata ai messaggi

Il pattern architetturale *orientato ai messaggi* prevede che diversi sottosistemi comunichino e collaborino esclusivamente scambiando messaggi, anche quando il destinatario non è disponibile. Questo modello è noto anche come *Message-Oriented Middleware* (MOM). I mittenti e i ricevitori devono conoscere solo i formati dei messaggi, e le applicazioni che comunicano non devono essere disponibili contemporaneamente, grazie al modello asincrono di pubblicazione/sottoscrizione. I messaggi possono essere resi persistenti. Esempi di tecnologie includono il *Java Message Service* (JMS) di Java EE, che permette alle applicazioni Java di scambiare messaggi, *Google Cloud Messaging* (GCM), sostituito da *Google Firebase Cloud Messaging* (FCM), *Roscore* per i sistemi basati su *Robot Operating System* (ROS) e lo standard ISO MQTT (*MQ Telemetry Transport o Message Queue Telemetry Transport*) su TCP/IP.

Nello stile di interazione *publish-subscribe*, i messaggi sono inviati da un componente (il publisher) attraverso canali virtuali (topics) a cui altri componenti software interessati possono sottoscriversi (subscribers). Un esempio di questa architettura è *Google Cloud Messaging* (GCM), ora sostituito da Google Firebase Cloud Messaging (FCM). I principi di *design* dell'architettura orientata ai messaggi includono *divide et impera*, dove l'applicazione è composta da componenti software isolati. La riduzione dell'accoppiamento è ottenuta poiché i componenti sono collegati in modo lasco, condividendo solo messaggi in formati di scambio dati. L'aumento dell'astrazione è garantito dalla semplicità di manipolazione dei formati dei messaggi prescritti, nascondendo i dettagli dell'applicazione dietro il sistema di messaggistica. La riusabilità è migliorata se i formati dei messaggi sono flessibili. I componenti possono essere riutilizzati finché il nuovo sistema aderisce ai formati dei messaggi proposti. La flessibilità è garantita dalla possibilità di aggiornare o potenziare facilmente la funzionalità di un sistema orientato ai messaggi aggiungendo o sostituendo componenti. La testabilità è migliorata poiché ogni componente può essere testato indipendentemente. Un *design* difensivo consiste semplicemente nel validare tutti i messaggi ricevuti prima di elaborarli.

Architettura esagonale

L'*architettura esagonale*, nota anche come *architettura porte e adattatori*, rappresenta un'alternativa all'architettura a strati ed è alla base dell'*architettura a microservizi*. In questo modello, il sistema è suddiviso in diverse componenti debolmente accoppiate, rappresentate come esagoni. Queste componenti sono connesse attraverso porte, che sono API astratte, e adattatori, che fungono da collante tra le porte e il mondo esterno. Gli adattatori permettono l'interazione attraverso una porta utilizzando una specifica tecnologia di comunicazione o connettore. È possibile avere più adattatori per una singola porta. Ad esempio, i dati possono essere forniti da un utente attraverso un'interfaccia grafica (GUI), un'interfaccia a riga di comando o un *controller* API. Un esempio di questa architettura è rappresentato da due applicazioni esagonali che comunicano tramite REST.

[comment] : lezione 27 nov

Validazione dell'architettura software

L'architettura software deve essere documentata perché racchiude tutte le decisioni prese durante la fase di progettazione e le proprietà del sistema. Inoltre, è fondamentale validare l'architettura per assicurarsi che soddisfi i requisiti iniziali.

La *validazione dell'architettura* consiste nel verificare che essa raggiunga determinati livelli di qualità. Una valutazione comparativa di diverse architetture, volta a individuarne i pregi e i difetti, è particolarmente utile nelle

fasi iniziali della progettazione. Questo processo aiuta a identificare l'architettura migliore da cui partire. Esistono diverse tipologie di valutazioni. Le validazioni qualitative si basano sull'esperienza e sulla conoscenza del dominio. Le validazioni quantitative, invece, si basano su metriche matematiche. È possibile anche validare l'architettura in modo dinamico attraverso simulazioni, per comprendere la resistenza al carico e altre proprietà simili. Un'altra tecnica di validazione è basata sugli scenari. Alcune qualità del sistema sono osservabili durante l'esecuzione, come le prestazioni, la sicurezza e la disponibilità. Altre proprietà, invece, non sono osservabili durante l'esecuzione, come la manutenibilità, la modificabilità, la portabilità, la riutilizzabilità, l'integrabilità e la testabilità.

L'uso di componenti a grana grossa può rendere il sistema più performante, ma può compromettere la sua manutenibilità. Allo stesso modo, la ridondanza dei dati può aumentare la disponibilità dei sistemi distribuiti, ma può ridurre la loro sicurezza. È quindi essenziale trovare un equilibrio tra questi aspetti, noto come *quality trade-off*. Per migliorare un attributo di qualità specifico, è possibile adottare *tattiche*, meccanismi o *pattern* architetturali che mirano a migliorare la qualità del sistema. Ad esempio, esistono tattiche per le prestazioni che possono essere applicate per ridurre la latenza del sistema. Queste tattiche vengono selezionate in base agli obiettivi di design per la metrica in questione, come le prestazioni. Gli obiettivi di design principali sono limitare la necessità di accesso alle risorse e gestire tali risorse in modo più veloce. Esistono due famiglie di soluzioni, una per ciascun obiettivo, e da queste si scelgono le migliori tattiche da applicare.

L'affidabilità di un sistema può essere misurata matematicamente attraverso l'intervallo di *uptime*, ovvero la probabilità che un componente o il sistema, attivo all'istante 0, sia ancora attivo all'istante t. La disponibilità, invece, amplia il concetto di affidabilità includendo anche l'idea di recupero, ovvero la capacità del sistema di autoripararsi. Un modo per misurare l'affidabilità è espresso dalla formula $R(1000) = 90\%$, che indica una probabilità del 90% di trovare il sistema ancora attivo dopo 1000 ore dall'avvio. Un altro modo di misurare è il "numero di 9", come mostrato in tabella:

Uptime %	**Downtime %**	**Downtime annuale**	**Downtime settimanale**
90%	10%	36.5 giorni	16:48 ore
99%	1%	3.65 giorni	1:41 oreminuti
99.9%	0.1%	8:45 minuti	10:05 minuti
99.99%	0.01%	52:30 minuti	1 minuto
99.999%	0.001%	5:15 minuti	6 secondi
99.9999%	0.0001%	31.5 secondi	0.6 secondi

Più formalmente:

- X è l'istante di fallimento di un componente
- $F(t)$ è la sua distribuzione cumulata
- $R(t)$ è la probabilità che il componente, attivo all'istante 0, stia ancora funzionando all'istante t

Allora

$$R(t) = P(X > t) = 1 - F(t)$$

e $f(t)\mathrm{d}t$ è la probabilità di fallire nell'intervallo $(t, t + \mathrm{d}t)$. Allora

$$F(t) = \int_0^t f(t)\mathrm{d}t$$

e

$$R(t) = 1 - F(t) = 1 - \int_0^t f(t)\mathrm{d}t = \int_t^\infty f(t)\mathrm{d}t.$$

Se $F(t)$ è esponenziale è disponibile un'approssimazione:

$$F(t) = P(X \leq t) = 1 - e^{\frac{-t}{MTTF}} \cong \frac{t}{MTTF} \quad (\text{per } \frac{t}{MTTF} \ll 1)$$

Dove $MTTF$ è il *Minimum Time To Failure*, determinabile sperimentalmente.

Un *Reliability Block Diagram* rappresenta lo stato del sistema. Nelle connessioni seriali, è necessario che tutte le componenti funzionino correttamente per mantenere il sistema operativo. Al contrario, nelle connessioni parallele, il sistema è più resistente, poiché è sufficiente che almeno una componente sia operativa. A partire dal diagramma, è possibile calcolare l'affidabilità del sistema utilizzando specifiche formule matematiche. L'affidabilità di una connessione seriale è data dal prodotto dell'affidabilità delle sue componenti, espressa come ΠR_i. L'affidabilità di una connessione parallela, invece, è data da $1 - \Pi(1 - R_i)$. In presenza di componenti gerarchici, il calcolo

dell'affidabilità può essere effettuato in modo *bottom-up*, partendo dalle componenti di livello più basso e risalendo fino a quelle di livello superiore.

La *disponibilità* è invece definita come

$$A = \frac{MTTF}{MTTF + MTTR}$$

dove $MTTR$ è il *Minimum Time To Repair*.

Modello a componenti di Android

Il framework di componenti di Android si articola in quattro elementi principali: *activity* e *fragment*, servizi, fornitori di contenuto e ricevitori *broadcast*. Il file *manifest*, un documento XML, descrive l'architettura dell'applicazione.

L'interazione tra i componenti avviene tramite eventi, rappresentati da oggetti *Intent*. Android supporta tre stili architetturali: MVC, MVP e MVVM. Gli ultimi due, MVP e MVVM, mirano a disaccoppiare la vista dal modello. In Android, la vista è rappresentata dalle *activity*, che corrispondono alle schermate dell'applicazione. Nelle versioni più recenti di Android, sono stati introdotti il *ViewModel* e il *LiveData* per facilitare l'implementazione del modello MVVM.

Un'*activity* visualizza la schermata utente e gestisce le interazioni con essa. Può essere visualizzata in una finestra a schermo intero o sovrapponibile. Un'applicazione Android è composta da un insieme di *activity* separate e disaccoppiate, con una *main activity* che può chiamare altre *activity*. Le *activity* attraversano diversi stati: *created*, *started* (visibile), *resumed* (visibile), *paused* (parzialmente visibile), *stopped* (nascosta) e *destroyed*. Per modificare il comportamento del sistema degli stati, è necessario sovrascrivere i metodi dell'*Activity*. I *fragment* sono utilizzati per creare interfacce utente dinamiche che si adattano alle dimensioni dello schermo. Possono essere attaccati e staccati e, sebbene non siano salvati nel *manifest*, hanno un ciclo di vita simile a quello delle *view*.

I servizi sono componenti autonomi senza interfaccia utente, che funzionano come controllori. Eseguono attività di lunga durata senza interazioni dirette con l'utente e rimangono in esecuzione in background, a meno che non ci sia carenza di memoria o risorse. Spesso operano in IPC e possono servire più applicazioni contemporaneamente. I servizi attraversano gli stati *create*, *bind*, *unbind* e *destroy*. I *fornitori di contenuto* consentono la condivisione di dati tra applicazioni, come rubrica, immagini, audio e video. Offrono operazioni per gestire i dati su database.

Le interazioni tra i componenti avvengono tramite oggetti *Intent*, che rappresentano richieste di eseguire un'azione. Gli *Intent* possono essere espliciti, specificando il nome completo della classe della componente da eseguire all'interno della stessa applicazione, o impliciti, specificando solo l'azione e lasciando ad Android il compito di determinare quale componente deve eseguirla. I *layout* disponibili includono il layout lineare e il layout a tabella.