---------- Domande Modulo 1

Breadth-First Search (BFS) e Depth-First Search (DFS)

Breadth-First Search (BFS) e **Depth-First Search (DFS)** sono due algoritmi di ricerca utilizzati per esplorare i grafi. La principale differenza tra questi due algoritmi risiede nella loro strategia di esplorazione:

- **BFS (Breadth-First Search)**: Questo algoritmo esplora tutti i nodi vicini al nodo corrente prima di passare ai nodi più lontani. Utilizza una **coda (queue)** per memorizzare i nodi da esplorare. In pratica, BFS visita i nodi livello per livello, iniziando dal nodo di partenza e procedendo verso i nodi adiacenti, poi verso i nodi adiacenti di questi ultimi, e così via. Questo metodo garantisce che il primo nodo trovato sia il più vicino al nodo di partenza in termini di numero di passaggi.
- **DFS (Depth-First Search)**: Questo algoritmo esplora il più possibile lungo ogni ramo prima di retrocedere. Utilizza una **pila (stack)** per memorizzare i nodi da esplorare. In pratica, DFS visita un nodo di partenza e poi continua a esplorare il più profondamente possibile lungo un ramo prima di tornare indietro e esplorare altri rami. Questo metodo può andare molto in profondità prima di tornare indietro e può essere utile per esplorare completamente un ramo prima di passare ad altri.

Quindi, la differenza principale è che **BFS esplora i nodi vicini prima di quelli lontani**, mentre **DFS esplora il più possibile lungo ogni ramo prima di retrocedere**. Questa differenza nella strategia di esplorazione influisce anche sulle strutture dati utilizzate: BFS usa una coda, mentre DFS usa una pila.

Quick Sort

Quick Sort è un algoritmo di ordinamento molto efficiente e ampiamente utilizzato, basato sul principio del "divide et impera":

- **Nel caso peggiore, il Quick Sort può avere una complessità temporale di O(n^2)**: Questo accade quando il pivot scelto è sempre il più grande o il più piccolo elemento dell'array, causando una divisione molto sbilanciata. In questo scenario, l'algoritmo deve effettuare un numero massimo di confronti e scambi, portando la complessità temporale a O(n^2).

Le altre affermazioni sono errate per i seguenti motivi:

- **Il Quick Sort richiede sempre uno spazio aggiuntivo proporzionale alla dimensione dell'input**: Questa affermazione è falsa. Quick Sort può essere implementato in modo da utilizzare spazio aggiuntivo O(log n) per la ricorsione, rendendolo molto efficiente in termini di spazio.
- **Il Quick Sort non può essere implementato per funzionare su liste collegate**: Questa affermazione è falsa. Quick Sort può essere adattato per funzionare su liste collegate, anche se l'implementazione è più complessa rispetto agli array.
- **Il Quick Sort è un algoritmo di ordinamento basato su confronti che non utilizza la ricorsione**: Questa affermazione è falsa. Quick Sort è un algoritmo ricorsivo per natura, anche se può essere implementato in modo iterativo.
- **Il Quick Sort ha sempre una complessità temporale di O(n log n), indipendentemente dall'input**: Questa affermazione è falsa. Sebbene Quick Sort abbia una complessità temporale media di O(n log n), nel caso peggiore può avere una complessità di O(n^2).

In sintesi, la complessità temporale di Quick Sort nel caso peggiore è O(n^2), ma nella maggior parte dei casi, grazie a una buona scelta del pivot, l'algoritmo ha una complessità media di O(n log n).

Quick Sort e Bubble Sort

Quick Sort e **Bubble Sort** sono due algoritmi di ordinamento con caratteristiche e prestazioni molto diverse:

- **Quick Sort**: Questo algoritmo è noto per la sua efficienza e utilizza il principio del "divide et impera". Nel caso medio, Quick Sort ha una complessità temporale di **O(n log n)**. Questo perché, in media, il processo di partizionamento divide l'array in due metà di dimensioni approssimativamente uguali, e ogni partizione richiede un tempo lineare per essere ordinata.
- **Bubble Sort**: Questo algoritmo è molto semplice ma inefficiente per grandi dataset. Nel caso medio, Bubble Sort ha una complessità temporale di **O(n^2)**. Questo perché, in media, ogni elemento deve essere confrontato con ogni altro elemento, portando a un numero di confronti e scambi che cresce quadraticamente con la dimensione dell'array.

In sintesi, la principale differenza tra Quick Sort e Bubble Sort in termini di complessità temporale nel caso medio è che **Quick Sort è molto più efficiente**, con una complessità di **O(n log n)**, mentre **Bubble Sort è molto meno efficiente**, con una complessità di **O(n^2)**.

Heap Sort e Merge Sort

Heap Sort e **Merge Sort** sono due algoritmi di ordinamento efficienti, ma con caratteristiche diverse:

1. **Entrambi gli algoritmi hanno una complessità temporale nel caso peggiore di O(n log n)**: Questa affermazione è vera. Sia Heap Sort che Merge Sort garantiscono una complessità temporale di O(n log n) nel caso peggiore, rendendoli molto efficienti per grandi dataset.
2. **Merge Sort richiede spazio aggiuntivo O(n), mentre Heap Sort richiede spazio aggiuntivo O(1)**: Questa affermazione è vera. Merge Sort necessita di spazio aggiuntivo proporzionale alla dimensione dell'array (O(n)) per memorizzare i sotto-array durante il processo di fusione. Heap Sort, invece, può essere implementato in-place, richiedendo solo spazio aggiuntivo costante (O(1)).

Le altre affermazioni sono errate per i seguenti motivi:

- **Entrambi gli algoritmi utilizzano una struttura dati a heap per funzionare**: Questa affermazione è falsa. Solo Heap Sort utilizza una struttura dati a heap. Merge Sort utilizza il principio del "divide et impera" e non richiede una struttura dati a heap.
- **Heap Sort è un algoritmo di ordinamento stabile, mentre Merge Sort non lo è**: Questa affermazione è falsa. In realtà, Merge Sort è un algoritmo di ordinamento stabile, mentre Heap Sort non lo è. Un algoritmo di ordinamento stabile mantiene l'ordine relativo degli elementi con chiavi uguali.

In sintesi, Heap Sort e Merge Sort sono entrambi efficienti con una complessità temporale nel caso peggiore di O(n log n), ma differiscono significativamente in termini di spazio aggiuntivo richiesto e stabilità.

Classificazione degli algoritmi

La classificazione degli algoritmi può essere fatta in base a diversi criteri, tra cui la determinazione, l'esattezza e l'approssimazione. Ecco una spiegazione dettagliata della risposta corretta:

- **L'algoritmo di Dijkstra è un esempio di algoritmo sia deterministico che esatto**: Questa affermazione è corretta. L'algoritmo di Dijkstra è utilizzato per trovare il percorso più breve da un nodo sorgente a tutti gli altri nodi in un grafo con pesi non negativi. È deterministico perché, dato lo stesso input, produce sempre lo stesso output. È anche esatto perché trova il percorso più breve senza approssimazioni.

Le altre affermazioni sono errate per i seguenti motivi:

- **Tutti gli algoritmi probabilistici sono anche approssimativi**: Questa affermazione è falsa. Gli algoritmi probabilistici utilizzano la casualità come parte del loro processo, ma possono essere esatti o approssimativi. Ad esempio, l'algoritmo di Miller-Rabin per il test di primalità è probabilistico ma esatto.
- **Gli algoritmi esatti non possono mai essere probabilistici**: Questa affermazione è falsa. Come menzionato sopra, esistono algoritmi probabilistici che sono esatti, come l'algoritmo di Miller-Rabin.
- **Gli algoritmi deterministici possono produrre output diversi per lo stesso input**: Questa affermazione è falsa. Gli algoritmi deterministici, per definizione, producono sempre lo stesso output per lo stesso input.
- **Gli algoritmi approssimativi sono sempre meno efficienti degli algoritmi esatti**: Questa affermazione è falsa. Gli algoritmi approssimativi possono essere più efficienti in termini di tempo di esecuzione rispetto agli algoritmi esatti, specialmente per problemi complessi o NP-difficili.

In sintesi, l'algoritmo di Dijkstra è un esempio di algoritmo sia deterministico che esatto, mentre le altre affermazioni contengono errori o imprecisioni riguardo alla classificazione degli algoritmi.

Master Theorem

Il Master Theorem è uno strumento potente utilizzato per analizzare la complessità temporale di algoritmi ricorsivi che seguono una forma specifica di ricorrenza. Ecco una spiegazione dettagliata della risposta corretta:

- **Fornisce una soluzione chiusa per la complessità temporale di algoritmi ricorsivi che soddisfano determinate condizioni**: Questa affermazione è corretta. Il Master Theorem è applicabile a ricorrenze della forma T(n) = aT(n/b) + f(n), dove a è maggiore o uguale a 1 e b è maggiore di 1. Esso fornisce una soluzione chiusa per la complessità temporale di tali algoritmi, permettendo di determinare il comportamento asintotico della ricorrenza.

Le altre affermazioni sono errate per i seguenti motivi:

- **Si applica solo ad algoritmi ricorsivi con un singolo caso base**: Questa affermazione è falsa. Il Master Theorem non è limitato a ricorrenze con un singolo caso base; si applica a ricorrenze che dividono il problema in sottoproblemi di dimensioni ridotte.
- **Può essere utilizzato per analizzare algoritmi ricorsivi che dividono il problema in parti di dimensioni disuguali**: Questa affermazione è falsa. Il Master Theorem si applica solo a ricorrenze che dividono il problema in sottoproblemi di dimensioni uguali.
- **È applicabile a tutti i tipi di algoritmi ricorsivi, indipendentemente dalla loro struttura**: Questa affermazione è falsa. Il Master Theorem è applicabile solo a ricorrenze che seguono la forma specifica menzionata sopra.

In sintesi, il Master Theorem è uno strumento utile per ottenere una soluzione chiusa per la complessità temporale di algoritmi ricorsivi che soddisfano determinate condizioni, rendendo l'analisi della complessità più semplice e diretta.

Algoritmo di Dijkstra e Algoritmo di Bellman-Ford

Sono entrambi utilizzati per trovare il percorso più breve in un grafo, ma hanno caratteristiche e applicazioni diverse:

1. L'algoritmo di Dijkstra è più efficiente in termini di complessità temporale rispetto all'algoritmo di Bellman-Ford per grafi con pesi non negativi: Questa affermazione è vera. L'algoritmo di Dijkstra ha una complessità temporale di O(V^2) con una semplice implementazione basata su matrici di adiacenza, o O(E + V log V) con una implementazione basata su code di priorità. È generalmente più veloce rispetto all'algoritmo di Bellman-Ford per grafi con pesi non negativi.
2. L'algoritmo di Bellman-Ford può gestire grafi con pesi negativi, mentre l'algoritmo di Dijkstra no: Questa affermazione è vera. L'algoritmo di Bellman-Ford è progettato per gestire grafi con pesi negativi e può anche rilevare cicli di peso negativo. L'algoritmo di Dijkstra, invece, non funziona correttamente se il grafo contiene archi con pesi negativi.

Le altre affermazioni sono errate per i seguenti motivi:

- L'algoritmo di Dijkstra può gestire grafi con pesi negativi, mentre l'algoritmo di Bellman-Ford no: Questa affermazione è falsa. È l'algoritmo di Bellman-Ford che può gestire pesi negativi, non Dijkstra.
- L'algoritmo di Dijkstra ha una complessità temporale di O(V^3), mentre l'algoritmo di Bellman-Ford ha una complessità temporale di O(VE): Questa affermazione è falsa. La complessità temporale dell'algoritmo di Dijkstra è O(V^2) o O(E + V log V), mentre quella dell'algoritmo di Bellman-Ford è O(VE).

In sintesi, l'algoritmo di Dijkstra è più efficiente per grafi con pesi non negativi, mentre l'algoritmo di Bellman-Ford è adatto per grafi con pesi negativi e può rilevare cicli di peso negativo.

Strategie - Divide et impera

La strategia **Divide et impera** è caratterizzata dalla scomposizione ricorsiva di un problema in sottoproblemi più piccoli, risolvendoli indipendentemente e poi combinando le soluzioni per ottenere il risultato finale. Questa tecnica è utilizzata in molti algoritmi efficienti, come il Merge Sort e il Quick Sort.

Le altre strategie menzionate hanno approcci diversi:

- **Branch and bound**: Utilizza una tecnica di esplorazione dello spazio delle soluzioni, spesso per problemi di ottimizzazione, riducendo il numero di soluzioni da esplorare attraverso limiti (bound).
- **Algoritmi greedy**: Risolvono problemi facendo scelte locali ottimali nella speranza di trovare una soluzione globale ottimale.
- **Programmazione dinamica**: Risolve problemi suddividendoli in sottoproblemi sovrapposti e memorizzando le soluzioni per evitare calcoli ripetuti.
- **Backtracking**: Esplora tutte le possibili soluzioni di un problema, tornando indietro (backtracking) quando una soluzione parziale non può essere estesa a una soluzione completa.

In sintesi, **Divide et impera** è la strategia che scompone ricorsivamente il problema in sottoproblemi più piccoli, risolvendoli indipendentemente e combinando le soluzioni per ottenere il risultato finale.

Linear Search e Binary Search

La principale differenza tra la ricerca lineare (Linear Search) e la ricerca binaria (Binary Search) in termini di complessità temporale nel caso peggiore è la seguente:

- **Ricerca lineare (Linear Search)**: Questo algoritmo cerca un elemento in un array o una lista scorrendo sequenzialmente tutti gli elementi fino a trovare quello desiderato o fino a raggiungere la fine della struttura dati. Nel caso peggiore, la ricerca lineare deve controllare tutti gli elementi, portando a una complessità temporale di O(n), dove n è il numero di elementi.
- **Ricerca binaria (Binary Search)**: Questo algoritmo cerca un elemento in un array ordinato dividendo ripetutamente l'array a metà e confrontando l'elemento centrale con quello cercato. Nel caso peggiore, la ricerca binaria riduce il problema a metà ad ogni passo, portando a una complessità temporale di O(log n), dove n è il numero di elementi.

In sintesi, la ricerca lineare ha una complessità temporale nel caso peggiore di O(n), mentre la ricerca binaria ha una complessità temporale nel caso peggiore di O(log n), rendendo la ricerca binaria molto più efficiente per grandi dataset ordinati.

Metodo della Sostituzione

Il Metodo della Sostituzione è una tecnica utilizzata per analizzare la complessità temporale di algoritmi ricorsivi. Ecco una spiegazione dettagliata delle affermazioni corrette:

1. Richiede di indovinare la forma della soluzione prima di procedere con la dimostrazione: Questa affermazione è vera. Il Metodo della Sostituzione inizia con l'ipotesi di una soluzione per la relazione di ricorrenza. Successivamente, si utilizza questa ipotesi per dimostrare che la soluzione è corretta.
2. Utilizza l'induzione matematica per dimostrare la correttezza della soluzione ipotizzata: Questa affermazione è vera. Una volta ipotizzata la soluzione, si utilizza l'induzione matematica per dimostrare che la soluzione è valida per tutti i valori del problema. **(AI DICE SI, RISPOSTA NO)**

3. Può essere utilizzato per dimostrare limiti superiori e inferiori sulla complessità di un algoritmo: Questa affermazione è vera. Il Metodo della Sostituzione può essere utilizzato per dimostrare sia limiti superiori che inferiori sulla complessità temporale di un algoritmo, fornendo una stima precisa della complessità.

Le altre affermazioni sono errate per i seguenti motivi:

- Garantisce sempre una soluzione esatta per qualsiasi relazione di ricorrenza: Questa affermazione è falsa. Il Metodo della Sostituzione non garantisce sempre una soluzione esatta; dipende dalla correttezza dell'ipotesi iniziale.
- È sempre più semplice da applicare rispetto al Master Theorem: Questa affermazione è falsa. In alcuni casi, il Master Theorem può essere più semplice e diretto da applicare rispetto al Metodo della Sostituzione, specialmente per ricorrenze che rientrano nelle forme specifiche trattate dal Master Theorem.

In sintesi, il Metodo della Sostituzione richiede di indovinare la forma della soluzione, utilizza l'induzione matematica per dimostrarne la correttezza e può essere utilizzato per dimostrare limiti superiori e inferiori sulla complessità di un algoritmo.

---------- Domande Modulo 2

Binary Search

Un Albero di Ricerca Binario (Binary Search Tree, BST) è una struttura dati che facilita operazioni di ricerca, inserimento e cancellazione efficienti. La proprietà caratteristica di un BST è la seguente:

- **In un BST, i valori di tutti i nodi nel sottoalbero sinistro di un nodo sono sempre minori del valore del nodo stesso, e i valori di tutti i nodi nel sottoalbero destro sono sempre maggiori del valore del nodo stesso**: Questa proprietà garantisce che l'albero sia ordinato in modo tale da permettere una ricerca binaria efficiente.

Le altre affermazioni sono errate per i seguenti motivi:

- **Un BST non può avere nodi foglia**: Questa affermazione è falsa. Un BST può avere nodi foglia, che sono nodi senza figli.
- **Ogni nodo in un BST ha esattamente due figli**: Questa affermazione è falsa. Un nodo in un BST può avere zero, uno o due figli.
- **In un BST, la somma dei valori dei nodi nel sottoalbero sinistro è sempre minore della somma dei valori dei nodi nel sottoalbero destro**: Questa affermazione è falsa. La proprietà del BST riguarda i singoli valori dei nodi, non la somma dei valori nei sottoalberi.

In sintesi, la proprietà caratteristica di un BST è che i valori dei nodi nel sottoalbero sinistro sono sempre minori del valore del nodo stesso, e i valori dei nodi nel sottoalbero destro sono sempre maggiori del valore del nodo stesso.

Albero binario

è una struttura dati in cui ogni nodo ha al massimo due figli. La proprietà caratteristica di un albero binario con n nodi è la seguente:

- **Un albero binario con n nodi ha sempre esattamente n-1 archi**: Questa affermazione è corretta. In un albero, il numero di archi è sempre uno in meno rispetto al numero di nodi, poiché ogni nodo (eccetto la radice) è collegato a un altro nodo tramite un arco.

Le altre affermazioni sono errate per i seguenti motivi:

- **In un albero B, tutti i nodi foglia devono trovarsi sempre allo stesso livello**: Questa affermazione è falsa. In un albero B, i nodi foglia non devono necessariamente trovarsi allo stesso livello, anche se spesso sono vicini.
- **Un albero di Fibonacci è un tipo di albero binario ottimizzato per operazioni di unione (merge)**: Questa affermazione è falsa. Gli alberi di Fibonacci sono strutture dati utilizzate principalmente per implementare code di priorità, non per operazioni di unione.
- **La profondità di un albero binario completo con n nodi è sempre $\log_2(n)$**: Questa affermazione è falsa. La profondità di un albero binario completo con n nodi è $\log_2(n+1) - 1$.
- **In un albero ternario, ogni nodo interno ha esattamente tre figli**: Questa affermazione è falsa. In un albero ternario, ogni nodo interno può avere fino a tre figli, ma non necessariamente esattamente tre.

In sintesi, la proprietà corretta è che un albero binario con n nodi ha sempre esattamente n-1 archi.

Albero AVL

Un albero AVL è un tipo di albero di ricerca binario (BST) autobilanciato, che mantiene l'altezza bilanciata per garantire operazioni efficienti. Ecco una spiegazione dettagliata delle affermazioni corrette:

1. L'attraversamento in-order di un BST produce sempre una sequenza ordinata degli elementi: Questa affermazione è vera. L'attraversamento in-order di un BST visita i nodi in ordine crescente, producendo una sequenza ordinata degli elementi.
2. In un BST, il successore di un nodo è sempre il suo figlio destro, se presente: Questa affermazione è vera. Il successore di un nodo in un BST è il nodo con il valore minimo nel suo sottoalbero destro, se tale figlio esiste.

Le altre affermazioni sono errate per i seguenti motivi:

- Un BST bilanciato garantisce che tutte le foglie siano allo stesso livello: Questa affermazione è falsa. Un BST bilanciato, come un albero AVL, garantisce che la differenza di altezza tra i sottoalberi di qualsiasi nodo sia al massimo 1, ma non necessariamente che tutte le foglie siano allo stesso livello.
- La complessità temporale dell'operazione di ricerca in un BST è sempre O(log n), indipendentemente dalla sua struttura: Questa affermazione è falsa. La complessità temporale della ricerca in un BST dipende dalla sua struttura. Solo in un BST bilanciato, come un albero AVL, la complessità è O(log n).
- L'eliminazione di un nodo con due figli in un BST richiede sempre la riorganizzazione completa dell'albero: Questa affermazione è falsa. L'eliminazione di un nodo con due figli in un BST richiede la sostituzione del nodo con il suo successore o predecessore, ma non necessariamente la riorganizzazione completa dell'albero.

In sintesi, le affermazioni corrette riguardano l'attraversamento in-order che produce una sequenza ordinata e il successore di un nodo che è il suo figlio destro, se presente.

Union-Find

La struttura dati Union-Find, o Insiemi Disgiunti, è utilizzata per gestire un insieme di elementi partizionati in insiemi disgiunti. Ecco una spiegazione dettagliata delle affermazioni corrette:

1. La complessità ammortizzata delle operazioni Union e Find con compressione del cammino e union by rank è O(α(n)), dove α(n) è la funzione inversa di Ackermann: Questa affermazione è vera. Utilizzando le tecniche di compressione del cammino e union by rank, le operazioni Union e Find diventano estremamente efficienti, con una complessità ammortizzata molto bassa. **(AI DICE SI, RISPOSTA NO)**
2. L'operazione MakeSet ha sempre una complessità temporale O(1): Questa affermazione è vera. L'operazione MakeSet, che crea un nuovo insieme contenente un singolo elemento, ha una complessità temporale costante.
3. La struttura dati Union-Find può essere utilizzata per implementare efficacemente l'algoritmo di Kruskal per il Minimum Spanning Tree: Questa affermazione è vera. L'algoritmo di Kruskal utilizza la struttura dati Union-Find per evitare la formazione di cicli durante la costruzione del Minimum Spanning Tree.

Le altre affermazioni sono errate per i seguenti motivi:

- L'implementazione con array garantisce sempre prestazioni migliori rispetto all'implementazione con foresta di alberi: Questa affermazione è falsa. L'implementazione con foresta di alberi, combinata con compressione del cammino e union by rank, è generalmente più efficiente.
- L'euristica dell'unione per rango (union by rank) utilizza sempre l'altezza dell'albero come criterio di decisione: Questa affermazione è falsa. L'unione per rango utilizza il rango (che può essere l'altezza o un'altra misura) per mantenere gli alberi bilanciati, ma non necessariamente l'altezza.

In sintesi, le affermazioni corrette riguardano la complessità ammortizzata delle operazioni Union e Find, la complessità temporale dell'operazione MakeSet e l'uso della struttura dati Union-Find nell'algoritmo di Kruskal.

Code

Le code (queue) sono strutture dati fondamentali utilizzate in vari contesti. Ecco una spiegazione dettagliata delle affermazioni corrette:

1. Una coda segue il principio FIFO (First-In-First-Out): Questa affermazione è vera. In una coda, il primo elemento inserito è anche il primo a essere rimosso, seguendo il principio FIFO.
2. Una coda a doppia estremità (deque) permette l'inserimento e la rimozione di elementi da entrambe le estremità: Questa affermazione è vera. Una deque (double-ended queue) consente l'inserimento e la rimozione di elementi sia dalla parte anteriore che dalla parte posteriore della coda.

Le altre affermazioni sono errate per i seguenti motivi:

- L'operazione di inserimento (enqueue) in una coda implementata con un array circolare ha sempre complessità temporale O(n): Questa affermazione è falsa. L'operazione di inserimento in una coda implementata con un array circolare ha complessità temporale O(1).
- L'implementazione di una coda utilizzando due stack garantisce che tutte le operazioni abbiano complessità temporale O(1) ammortizzata: Questa affermazione è falsa. L'implementazione di una coda utilizzando due stack può garantire che le operazioni abbiano complessità temporale O(1) ammortizzata, ma non tutte le operazioni sono sempre O(1).
- In una coda con priorità, gli elementi vengono sempre estratti in ordine di inserimento: Questa affermazione è falsa. In una coda con priorità, gli elementi vengono estratti in base alla loro priorità, non in ordine di inserimento.

In sintesi, le affermazioni corrette riguardano il principio FIFO delle code e la capacità delle deque di permettere l'inserimento e la rimozione di elementi da entrambe le estremità.

Hash table

Una **hash table** è una struttura dati che associa chiavi a valori utilizzando una funzione di hash per calcolare l'indice dell'array in cui memorizzare il valore. La proprietà caratteristica di una buona funzione di hash è la seguente:

- **Una buona funzione di hash dovrebbe minimizzare le collisioni distribuendo le chiavi uniformemente attraverso l'array della hash table**: Questa affermazione è corretta. Una funzione di hash efficace riduce il numero di collisioni, ovvero situazioni in cui due chiavi diverse producono lo stesso indice. Distribuendo le chiavi uniformemente, si ottiene una hash table più efficiente.

Le altre affermazioni sono errate per i seguenti motivi:

- **Una funzione di hash crittografica è sempre la scelta migliore per implementare una hash table efficiente**: Questa affermazione è falsa. Le funzioni di hash crittografiche sono progettate per sicurezza, non necessariamente per efficienza in una hash table.
- **Il rehashing è un processo che viene eseguito automaticamente ogni volta che si inserisce un nuovo elemento nella hash table**: Questa affermazione è falsa. Il rehashing viene eseguito solo quando il fattore di carico supera una certa soglia, non ad ogni inserimento.
- **In una hash table, la complessità temporale peggiore per l'operazione di ricerca è sempre O(1), indipendentemente dal fattore di carico**: Questa affermazione è falsa. La complessità temporale peggiore può essere O(n) se ci sono molte collisioni e il fattore di carico è alto.
- **L'indirizzamento aperto lineare garantisce sempre una distribuzione uniforme delle chiavi, indipendentemente dalla funzione di hash utilizzata**: Questa affermazione è falsa. L'indirizzamento aperto lineare può portare a clustering, dove le chiavi si accumulano in una parte dell'array, se la funzione di hash non è buona.

In sintesi, una buona funzione di hash dovrebbe minimizzare le collisioni distribuendo le chiavi uniformemente attraverso l'array della hash table.

---------- Domande Modulo 3

Classi di complessità

Le classi di complessità sono categorie utilizzate per classificare i problemi in base alla difficoltà di risoluzione e alle risorse computazionali necessarie. Ecco una spiegazione dettagliata della risposta corretta:

- **Se un problema è NP-completo, allora è il problema più difficile in NP e qualsiasi problema in NP può essere ridotto a esso in tempo polinomiale**: Questa affermazione è vera. I problemi NP-completi sono i problemi più difficili all'interno della classe NP. Se un problema è NP-completo, significa che ogni altro problema in NP può essere trasformato (ridotto) in questo problema in tempo polinomiale. Inoltre, se si trovasse un algoritmo in tempo polinomiale per risolvere un problema NP-completo, allora tutti i problemi in NP potrebbero essere risolti in tempo polinomiale.

Le altre affermazioni sono errate per i seguenti motivi:

- **La classe NP include solo i problemi che possono essere risolti esattamente in tempo polinomiale**: Questa affermazione è falsa. La classe NP include i problemi per cui una soluzione può essere verificata in tempo polinomiale, non necessariamente risolti esattamente in tempo polinomiale.
- **La classe P include tutti i problemi che possono essere risolti in tempo esponenziale**: Questa affermazione è falsa. La classe P include i problemi che possono essere risolti in tempo polinomiale, non esponenziale.
- **La classe P è un sottoinsieme della classe NP-hard**: Questa affermazione è falsa. La classe P è un sottoinsieme della classe NP, non NP-hard. I problemi NP-hard sono almeno tanto difficili quanto i problemi NP-completi, ma non necessariamente appartenenti alla classe NP.

In sintesi, la classe NP-completo rappresenta i problemi più difficili in NP, e qualsiasi problema in NP può essere ridotto a un problema NP-completo in tempo polinomiale.

Algoritmo di Edmonds-Karp e algoritmo di Ford-Fulkerson

Gli algoritmi di **Edmonds-Karp** e **Ford-Fulkerson** sono entrambi utilizzati per risolvere il problema del **flusso massimo** in una rete di flusso. Entrambi si basano sull'idea di trovare cammini aumentanti (ovvero percorsi lungo i quali è possibile inviare flusso aggiuntivo) e successivamente aumentare il flusso lungo questi cammini.

1. **Entrambi gli algoritmi possono essere utilizzati per trovare il flusso massimo in una rete di flusso:** Sia l'algoritmo di **Ford-Fulkerson** che l'algoritmo di **Edmonds-Karp** sono progettati per risolvere il problema del flusso massimo. Entrambi cercano cammini aumentanti per aumentare il flusso in una rete, ma differiscono nel metodo utilizzato per trovare questi cammini.
2. **L'algoritmo di Edmonds-Karp utilizza la ricerca in ampiezza (BFS) per trovare i cammini aumentanti, mentre l'algoritmo di Ford-Fulkerson può utilizzare sia la ricerca in ampiezza (BFS) sia la ricerca in profondità (DFS)**

L'algoritmo di **Edmonds-Karp** è una versione specifica dell'algoritmo di **Ford-Fulkerson** che utilizza la **ricerca in ampiezza (BFS)** per trovare i cammini aumentanti, mentre **Ford-Fulkerson** può usare **sia BFS che DFS** per trovare i cammini aumentanti.

Risposte errate:

3. **L'algoritmo di Ford-Fulkerson garantisce sempre di trovare la soluzione ottimale, mentre l'algoritmo di Edmonds-Karp non lo fa.**
   Entrambi gli algoritmi garantiscono di trovare la soluzione ottimale del flusso massimo. La differenza è che l'algoritmo di **Edmonds-Karp** è più **efficiente** nel garantire una soluzione ottimale in tempo polinomiale rispetto a **Ford-Fulkerson**, che può richiedere tempo esponenziale se utilizzato con DFS.
4. **L'algoritmo di Edmonds-Karp ha una complessità temporale peggiore rispetto all'algoritmo di Ford-Fulkerson**
   L'algoritmo di **Edmonds-Karp** ha una complessità temporale di **O(VE^2)**, mentre **Ford-Fulkerson** ha una complessità **O(E|f_max|)**, dove **f_max** è il flusso massimo. Se il flusso massimo è grande, **Ford-Fulkerson** può essere molto più lento, mentre **Edmonds-Karp** ha una complessità più prevedibile.

In sintesi, l'algoritmo di **Edmonds-Karp** è una versione più specifica e prevedibile di **Ford-Fulkerson**, che utilizza BFS per trovare i cammini aumentanti e garantisce una soluzione ottimale con una complessità più controllata. Entrambi possono essere utilizzati per trovare il flusso massimo, ma la scelta dell'algoritmo dipende dalla situazione e dalle caratteristiche del grafo.

Metodo dei cammini aumentanti

Il metodo dei cammini aumentanti è utilizzato negli algoritmi di flusso massimo, come l'algoritmo di Ford-Fulkerson e l'algoritmo di Edmonds-Karp. Questo metodo funziona nel seguente modo:

Il metodo dei cammini aumentanti individua un cammino dal nodo sorgente al nodo pozzo (sink) in cui ogni arco ha capacità residua positiva. Un cammino aumentante è un percorso attraverso il quale è possibile inviare più flusso. Una volta trovato un cammino aumentante, il flusso viene aumentato lungo questo cammino fino a quando non esistono più cammini aumentanti con capacità residua positiva. Questo processo continua fino a raggiungere il flusso massimo.

Le altre affermazioni sono errate per i seguenti motivi:

Il metodo dei cammini aumentanti cerca di trovare il percorso più lungo in un grafo orientato: Questa affermazione è falsa. Il metodo dei cammini aumentanti cerca di trovare cammini con capacità residua positiva, non il percorso più lungo.

Il metodo dei cammini aumentanti cerca di minimizzare il costo totale di invio del flusso attraverso una rete di flusso: Questa affermazione è falsa. Il metodo dei cammini aumentanti cerca di massimizzare il flusso, non di minimizzare il costo.

Il metodo dei cammini aumentanti cerca di determinare un insieme di cammini disgiunti che massimizzano il flusso totale in una rete di flusso: Questa affermazione è falsa. Il metodo dei cammini aumentanti cerca singoli cammini aumentanti, non necessariamente cammini disgiunti.

In sintesi, il metodo dei cammini aumentanti individua cammini dal nodo sorgente al nodo pozzo con capacità residua positiva e aumenta il flusso lungo questi cammini fino a raggiungere il flusso massimo.

Algoritmi di approssimazione

Gli algoritmi di approssimazione sono utilizzati per trovare soluzioni approssimate a problemi di ottimizzazione, specialmente quando i problemi sono troppo complessi per essere risolti esattamente in tempo ragionevole. Questo è particolarmente vero per i problemi NP-difficili, dove è improbabile che esistano algoritmi esatti efficienti in tempo polinomiale. Gli algoritmi di approssimazione cercano di fornire soluzioni subottimali in tempo polinomiale.

Gli algoritmi di approssimazione possono fornire una soluzione che è entro un fattore costante dalla soluzione ottimale. Questo significa che la soluzione trovata dall'algoritmo è garantita essere vicina alla soluzione ottimale entro un certo margine, spesso espresso come un fattore moltiplicativo.

Le altre affermazioni sono errate per i seguenti motivi:

Gli algoritmi di approssimazione non sono utili per problemi NP-difficili: Questa affermazione è falsa. Gli algoritmi di approssimazione sono spesso utilizzati proprio per i problemi NP-difficili, dove trovare una soluzione esatta è impraticabile.

Gli algoritmi di approssimazione garantiscono sempre di trovare la soluzione ottimale: Questa affermazione è falsa. Gli algoritmi di approssimazione non garantiscono di trovare la soluzione ottimale, ma piuttosto una soluzione che è vicina all'ottimale entro un certo fattore.

In sintesi, gli algoritmi di approssimazione sono utilizzati per problemi complessi e possono fornire soluzioni che sono entro un fattore costante dalla soluzione ottimale.

Algoritmi greedy

Gli algoritmi greedy sono progettati per risolvere problemi di ottimizzazione facendo scelte locali ottimali in ogni fase, **senza considerare le conseguenze future**. Sono spesso utilizzati quando si ha la garanzia che una serie di scelte localmente ottimali porti anche a una soluzione globalmente ottimale. Questo approccio è efficace solo per una specifica classe di problemi che soddisfano determinate proprietà, come la **proprietà greedy** e la **struttura a sottoproblemi ottimali**.

Gli algoritmi greedy **prendono decisioni basate solo sulle informazioni disponibili al momento**, e sono spesso **più semplici e più veloci da implementare** rispetto ad approcci più sofisticati, come la programmazione dinamica. Esempi classici includono l'algoritmo di Kruskal per il Minimum Spanning Tree e l'algoritmo di Dijkstra per il cammino minimo su grafi con pesi non negativi.

Le altre affermazioni sono errate per i seguenti motivi:

Gli algoritmi greedy sempre garantiscono la soluzione ottimale per qualsiasi problema di ottimizzazione:

Questa affermazione è falsa. Gli algoritmi greedy **non garantiscono sempre** la soluzione ottimale. Ad esempio, nel problema dello zaino (0/1 Knapsack), un approccio greedy può fallire nel trovare la soluzione migliore.

Gli algoritmi greedy risolvono i sottoproblemi e memorizzano le loro soluzioni per evitare ricalcoli futuri:

Questa affermazione è falsa. Questa è una caratteristica tipica della **programmazione dinamica**, non degli algoritmi greedy. I greedy **non memorizzano** i risultati dei sottoproblemi, ma prendono decisioni una volta per tutte.

In sintesi, gli algoritmi greedy prendono decisioni locali ottimali, sono efficienti e semplici, ma funzionano correttamente solo in particolari classi di problemi in cui ottimalità locale implica ottimalità globale.

Algoritmo di Kruskal vs Algoritmo di Prim

Gli algoritmi di **Kruskal** e **Prim** sono entrambi utilizzati per trovare un **albero di copertura minimo (Minimum Spanning Tree, MST)** in un grafo pesato non orientato, ma differiscono nel modo in cui selezionano gli archi e si espandono nel grafo.

L'**algoritmo di Kruskal può gestire grafi disconnessi**, trovando l'albero di copertura minimo **per ciascuna componente connessa**, mentre l'**algoritmo di Prim richiede che il grafo sia connesso**, in quanto si espande a partire da un nodo includendo progressivamente i nodi adiacenti con il minor costo.

Le altre affermazioni sono errate per i seguenti motivi:

L'algoritmo di Kruskal funziona meglio su grafi densi, mentre l'algoritmo di Prim funziona meglio su grafi sparsi:

Questa affermazione è falsa. In realtà, **Kruskal è più efficiente su grafi sparsi**, dove il numero di archi è relativamente basso rispetto al numero di nodi. Al contrario, **Prim (con rappresentazione a matrice e priority queue)** tende a essere più efficiente su **grafi densi**.

L'algoritmo di Kruskal utilizza una struttura a coda con priorità (priority queue), mentre l'algoritmo di Prim non la utilizza mai:

Questa affermazione è falsa. È l'**algoritmo di Prim** che **fa uso esplicito di una coda con priorità** per scegliere l'arco minimo da aggiungere. Kruskal, invece, **ordina tutti gli archi all'inizio** e utilizza la struttura Union-Find, **senza bisogno di una coda con priorità**.

L'algoritmo di Kruskal inizia da un nodo specifico e si espande aggiungendo i nodi adiacenti, mentre l'algoritmo di Prim ordina tutti i lati in ordine crescente di peso:

Questa affermazione è falsa. È l'**algoritmo di Prim** che **parte da un nodo** e si espande localmente, mentre **Kruskal ordina tutti gli archi globalmente** in ordine crescente di peso.

In sintesi, la differenza corretta è che l'**algoritmo di Kruskal può lavorare anche su grafi disconnessi**, trovando un MST per ogni componente, mentre l'**algoritmo di Prim richiede che il grafo sia connesso**, perché si basa su un'espansione continua da un nodo iniziale.

---------- Domande Modulo 4

Struttura dati per migliorarne le funzionalità

Le strutture dati possono essere arricchite in vari modi per migliorarne le funzionalità e ottimizzare le operazioni che devono eseguire. Diverse tecniche vengono utilizzate a seconda delle esigenze specifiche del problema.

1. **Aggiungere campi ausiliari ai nodi per memorizzare informazioni aggiuntive che supportano operazioni più complesse**

Aggiungere **campi ausiliari** ai nodi è una tecnica comune per arricchire una struttura dati. Ad esempio, in un albero binario, si potrebbero aggiungere informazioni come il **rango** di un nodo o la **profondità** per supportare operazioni aggiuntive, come la ricerca più rapida o la gestione di pesi in strutture di dati avanzate.

2. **Utilizzare strutture di dati ibride che combinano due o più strutture di dati per ottimizzare diverse operazioni**

Le **strutture ibride** combinano due o più strutture dati per ottimizzare operazioni diverse. Un esempio comune è l'uso di una **coda di priorità** combinata con un **heap** per gestire efficientemente l'ordinamento dinamico.

Risposte Errate:

3. **Ridurre la complessità delle operazioni di base sacrificando la leggibilità del codice**
   Sebbene ridurre la complessità possa essere importante, non si dovrebbe **sacrificare la leggibilità del codice**. La leggibilità del codice è cruciale per la manutenzione e la comprensione del software a lungo termine.
4. **Implementare la struttura dati utilizzando esclusivamente la programmazione funzionale per garantire l'immutabilità**
   Sebbene la **programmazione funzionale** e l'**immutabilità** possano essere utili in alcuni contesti, non è necessario utilizzare esclusivamente questo paradigma per arricchire una struttura dati. La scelta del paradigma dipende dalle esigenze specifiche e dalle caratteristiche del problema da risolvere.

In sintesi, l'arricchimento delle strutture dati tramite campi ausiliari e strutture ibride è una tecnica comune e potente per ottimizzare le operazioni, mentre sacrificare la leggibilità del codice o seguire esclusivamente la programmazione funzionale non sono approcci ideali.

Algoritmo di Huffman

L'algoritmo di **Huffman** è ampiamente utilizzato in contesti di **compressione dei dati**, in particolare per creare una codifica efficiente dei simboli, basata sulla loro frequenza.

L'algoritmo di Huffman è un algoritmo di compressione senza perdita che crea codici prefissi ottimali per una data frequenza di simboli

L'algoritmo di **Huffman** è utilizzato per **comprimere i dati** in modo **senza perdita**. Esso crea una codifica ottimale di **lunghezza variabile** per ogni simbolo in base alla sua **frequenza di occorrenza**. I simboli più frequenti ottengono codici più corti, mentre i simboli meno frequenti ottengono codici più lunghi. Questo tipo di codifica è un esempio di **codifica a prefisso**, dove nessun codice è un prefisso di un altro, evitando ambiguità.

Risposte errate:

L'algoritmo di Huffman è un metodo di crittografia simmetrica

L'algoritmo di Huffman non è un algoritmo di crittografia, ma un algoritmo di **compressione** dei dati. Non ha nulla a che fare con la crittografia.

L'algoritmo di Huffman utilizza una tecnica di programmazione dinamica per creare la codifica ottimale

Sebbene l'algoritmo di Huffman produca una soluzione ottimale, non utilizza **programmazione dinamica**. Piuttosto, utilizza una **struttura a heap** per costruire un albero di codifica a partire dalle frequenze dei simboli.

L'algoritmo di Huffman è utilizzato per trovare il cammino minimo in un grafo

L'algoritmo di Huffman non è utilizzato per risolvere problemi sui grafi, ma per la **compressione dei dati**. La ricerca del cammino minimo in un grafo è un altro problema, risolvibile tramite algoritmi come **Dijkstra** o **Bellman-Ford**.

In sintesi, l'algoritmo di **Huffman** è un **algoritmo di compressione senza perdita** che ottimizza la codifica dei simboli in base alla loro frequenza, creando una codifica prefisso ottimale.

Algoritmi avidi ricorsivi

Gli algoritmi **avidi ricorsivi** utilizzano una strategia basata sulla scelta immediata di un'opzione che sembra ottimale in quel momento, con l'idea di arrivare alla soluzione finale passo dopo passo. Questo approccio è particolarmente utile in alcuni contesti, ma non sempre garantisce soluzioni ottimali, a meno che non si soddisfino determinate condizioni.

Gli algoritmi avidi ricorsivi sono particolarmente efficaci quando il problema ha la proprietà di sottostruttura ottimale
Gli algoritmi avidi ricorsivi sono particolarmente **efficaci** per i problemi che possiedono la **proprietà di sottostruttura ottimale**, ossia quando la soluzione ottimale del problema può essere costruita combinando soluzioni ottimali dei suoi sotto-problemi. In questi casi, un algoritmo **avido** può trovare la soluzione ottimale in modo efficiente.

Gli algoritmi avidi ricorsivi sono garantiti di trovare la soluzione ottimale solo se il problema ha la proprietà di scelta golosa
Gli algoritmi avidi sono **garantiti di trovare la soluzione ottimale** solo se il problema ha la **proprietà di scelta golosa**, cioè se una **scelta locale ottimale** porta sempre a una soluzione globale ottimale. In assenza di questa proprietà, gli algoritmi avidi potrebbero non produrre la soluzione ottimale.

Risposte errate:

Gli algoritmi avidi ricorsivi sono sempre meno efficienti degli algoritmi iterativi
Non è necessariamente vero che gli algoritmi avidi ricorsivi siano **meno efficienti** degli algoritmi iterativi. La **complessità temporale** dipende dal problema specifico e dalla sua implementazione. In molti casi, la ricorsione può essere altrettanto efficiente, se non di più, rispetto a un'implementazione iterativa.

Gli algoritmi avidi ricorsivi utilizzano la tecnica di programmazione dinamica per memorizzare le soluzioni dei sotto-problemi
Gli algoritmi avidi **non utilizzano la programmazione dinamica**. Sebbene entrambi gli approcci risolvano problemi suddividendo il problema in sotto-problemi, la **programmazione dinamica** memorizza le soluzioni dei sotto-problemi per evitare ricalcoli, mentre un algoritmo avaro prende decisioni locali senza memorizzare le soluzioni precedenti.

In sintesi, gli algoritmi avidi ricorsivi sono efficaci nei problemi con **sottostruttura ottimale** e garantiscono soluzioni ottimali solo se il problema soddisfa la **proprietà di scelta golosa**. Tuttavia, non sono sempre meno efficienti degli algoritmi iterativi e non utilizzano la programmazione dinamica.

---------- Domande Modulo 5

Operatore di Mutazione in un Algoritmo Genetico

L'operatore di mutazione in un algoritmo genetico è fondamentale per introdurre variazione nella popolazione, evitando la convergenza prematura e migliorando la capacità di esplorare nuove soluzioni.

L'operatore di mutazione introduce piccole modifiche casuali in un cromosoma, alterando uno o più dei suoi geni. Queste modifiche sono utili per esplorare soluzioni che potrebbero non essere raggiungibili attraverso il crossover e per preservare la diversità genetica all'interno della popolazione. L'introduzione di mutazioni permette all'algoritmo di evitare di rimanere bloccato in ottimi locali e contribuisce alla continua evoluzione della soluzione.

Risposte errate:

Termina l'algoritmo

L'operatore di mutazione non termina l'algoritmo; al contrario, fa parte del processo iterativo che migliora progressivamente la soluzione.

Seleziona i migliori individui

La selezione dei migliori individui è compito di un operatore di selezione, non dell'operatore di mutazione. L'operatore di mutazione agisce successivamente sulla popolazione selezionata.

Combina due soluzioni

La combinazione di due soluzioni avviene tramite l'operatore di crossover, non tramite l'operatore di mutazione.

Valuta la fitness di una soluzione

La valutazione della fitness è un passaggio separato che avviene dopo l'applicazione degli operatori genetici, come crossover e mutazione.
In sintesi, l'operatore di mutazione introduce modifiche casuali nei cromosomi durante l'evoluzione dell'algoritmo genetico, contribuendo a esplorare soluzioni non ovvie e preservando la diversità della popolazione.

Vantaggi della computazione quantistica rispetto a quella classica

La computazione quantistica offre potenziali vantaggi significativi rispetto alla computazione classica, grazie all'uso delle leggi della meccanica quantistica, come la sovrapposizione e l'entanglement. Questi vantaggi si traducono in miglioramenti nelle prestazioni di alcune classi di problemi computazionali.

✓ Ricerca più rapida in database non strutturati
La computazione quantistica, attraverso algoritmi come l'algoritmo di Grover, è in grado di ridurre significativamente il tempo necessario per cercare un elemento in un database non strutturato, migliorando l'efficienza rispetto ai metodi classici.

✓ Risoluzione efficiente di problemi di fattorizzazione
L'algoritmo di Shor è un famoso esempio che mostra come la computazione quantistica possa risolvere il problema di fattorizzazione in modo molto più rapido rispetto ai migliori algoritmi classici, rendendo possibili attacchi crittografici a metodi come RSA, che si basano sulla difficoltà di fattorizzare numeri grandi.

✗ Consumo energetico sempre inferiore
Non è garantito che la **computazione quantistica** sia sempre **più efficiente dal punto di vista energetico**. Sebbene la computazione quantistica possa risolvere certi problemi in modo più rapido, la costruzione e il mantenimento dei **computer quantistici** richiedono **condizioni speciali**, come temperature estremamente basse, e quindi **potrebbero comportare un alto consumo energetico** in alcune fasi.

✗ Immunità completa agli errori
La computazione quantistica non è **immune agli errori**. Anzi, una delle sfide principali nella computazione quantistica è la **decoerenza quantistica** e gli **errori di misura** che possono verificarsi a causa della fragilità delle informazioni quantistiche. Esistono tecniche di **correzione degli errori quantistici**, ma l'immunità completa agli errori non è un vantaggio garantito.

✗ Minore complessità hardware
Il **hardware quantistico** è noto per essere **molto complesso**. Le attuali tecnologie quantistiche richiedono **macchine estremamente delicate** e **costose**, e la costruzione di computer quantistici funzionanti è ancora una sfida ingegneristica significativa. Quindi, non possiamo affermare che la computazione quantistica abbia una **minore complessità hardware** rispetto a quella classica.

Metodo del gradiente

Il **metodo del gradiente** è una famiglia di algoritmi di ottimizzazione iterativi utilizzati per trovare minimi o massimi locali di una funzione. Questi metodi si basano sul calcolo del **gradiente** (cioè del vettore delle derivate parziali), che indica la direzione di massima variazione della funzione. A seconda del segno con cui si segue il gradiente, si distinguono due principali varianti:

- **Gradiente discendente** (*gradient descent*): cerca minimi locali.
- **Gradiente ascendente** (*gradient ascent*): cerca massimi locali.

Il metodo richiede condizioni specifiche sulla funzione obiettivo, come la **continuità** e la **differenziabilità**, ed è spesso potenziato con tecniche per migliorare la **velocità di convergenza**, come il **momentum**.

✓ Il metodo del gradiente ascendente è utilizzato per trovare i massimi locali delle funzioni di costo
Corretto. Il gradiente ascendente si muove nella direzione del gradiente per aumentare il valore della funzione. È comunemente usato per massimizzare una funzione, ad esempio nella massimizzazione della verosimiglianza. **(AI DICE SI, RISPOSTA NO)**

✓ Il metodo del gradiente richiede che la funzione obiettivo sia continua e differenziabile
Esatto. Il metodo si basa sul calcolo delle derivate, che richiedono che la funzione sia almeno **differenziabile**. La continuità è implicitamente richiesta dalla differenziabilità.

✓ Il metodo del gradiente discendente può utilizzare tecniche come il momentum per accelerare la convergenza
Vero. Il **momentum** è una tecnica che introduce una memoria del gradiente passato per evitare oscillazioni e migliorare la velocità di discesa verso il minimo.

✗ Il metodo del gradiente è sempre garantito di trovare il minimo globale di una funzione convessa
Falso. Anche se per funzioni **strettamente convesse** il metodo *può* convergere al minimo globale, **non è sempre garantito** a causa di problemi numerici, cattiva scelta della **learning rate**, o condizioni di **plateau** e **scarsa curvatura**. Inoltre, la garanzia richiede anche che il passo sia scelto in modo adeguato e che la funzione sia **lipschitziana** (tra le altre condizioni tecniche).

Subset Sum Problem

Il problema della **somma di un sottoinsieme** (Subset Sum Problem) è un problema classico della teoria della complessità computazionale. Esso consiste nel determinare se, dato un insieme di numeri interi e un valore target, esiste un sottoinsieme i cui elementi sommano esattamente al target. Si tratta di un problema **NP-completo**, quindi risolverlo esattamente richiede, nel caso peggiore, un tempo esponenziale (a meno che P = NP). Proprio per questo motivo, in ambito pratico si studiano **algoritmi di approssimazione**, che trovano soluzioni "quasi ottimali" in tempo ragionevole. La domanda chiede quali caratteristiche rendano questo problema adatto a tali algoritmi.

✓ **La possibilità di ottenere soluzioni approssimate in tempo polinomiale**
Questa è una caratteristica chiave. Anche se il problema esatto è NP-completo, esistono algoritmi di approssimazione (in particolare **Fully Polynomial-Time Approximation Schemes**, FPTAS) che trovano soluzioni approssimate in tempo polinomiale, rendendo il problema trattabile in contesti applicativi.

✓ **La difficoltà di trovare una soluzione esatta in tempo polinomiale**
La difficoltà intrinseca del problema (è NP-completo) giustifica l'uso di tecniche approssimative. È proprio questa difficoltà che rende rilevante l'approccio euristico o approssimativo: se il problema fosse risolvibile esattamente in tempo polinomiale, non ci sarebbe bisogno di approssimazioni.

✗ **L'assenza di vincoli sugli input**
Questo è fuorviante. Il problema ha vincoli ben definiti: deve esserci un insieme di numeri interi e un target. La presenza o assenza di vincoli non determina la natura della sua approssimabilità.

✗ **La presenza di una struttura ottima a sottostruttura**
Questa è una proprietà tipica dei problemi risolvibili tramite programmazione dinamica (es. problema dello zaino), ma non è direttamente collegata alla possibilità di usare algoritmi di *approssimazione* quanto piuttosto di *ottimizzazione esatta* su casi ristretti.

✗ **La garanzia di una soluzione ottima unica**
Non è vera né necessaria per l'uso di algoritmi di approssimazione. Anzi, spesso i problemi approssimabili non hanno una sola soluzione ottima, ma più soluzioni che soddisfano un dato criterio entro un margine d'errore.

Fork-join Parallelismo

Nel contesto del **parallelismo** (sia hardware che software), uno schema comune di progettazione è quello della **biforcazione-ricongiunzione** (*fork-join*). Questo modello è fondamentale nella programmazione parallela e rappresenta il flusso in cui un'attività principale si suddivide in più sottocompiti eseguibili in parallelo (**biforcazione**), per poi riunirsi in un punto comune dove i risultati vengono **ricongiunti** per continuare l'elaborazione.

✓ La biforcazione è il processo di suddividere un compito in sottocompiti paralleli, mentre la ricongiunzione è il processo di combinare i risultati dei sottocompiti
Questa è la descrizione esatta del modello *fork-join*. Nella biforcazione, un'attività genera più sottocompiti indipendenti che possono essere eseguiti in parallelo. La ricongiunzione rappresenta il punto in cui il programma attende la conclusione di tutti i sottocompiti per procedere, combinando i risultati se necessario.

✗ La biforcazione è utilizzata per aumentare la granularità dei task, mentre la ricongiunzione è utilizzata per ridurre la granularità dei task
Questa affermazione è fuorviante: la **granularità** si riferisce alla quantità di lavoro svolto da ciascun task, non è direttamente legata al meccanismo di *fork-join*.

✗ La biforcazione implica l'esecuzione di più thread in sequenza, mentre la ricongiunzione implica l'esecuzione di un singolo thread
Questo è concettualmente errato: la biforcazione serve proprio per l'esecuzione **parallela**, non sequenziale, dei thread. Inoltre, la ricongiunzione non implica necessariamente un ritorno a un solo thread, ma semplicemente un punto di sincronizzazione.

✗ La biforcazione implica la distribuzione dei dati tra più processori, mentre la ricongiunzione implica la sincronizzazione dei dati tra i processori
Anche se la distribuzione dei dati può avvenire nel contesto parallelo, la definizione di biforcazione-ricongiunzione riguarda **il flusso di controllo** e **l'organizzazione dei compiti**, non direttamente la gestione dei dati tra processori.

Fattorizzazione di interi

La **fattorizzazione di interi** è il problema di trovare i fattori primi di un numero intero positivo. Questo problema ha grande rilevanza teorica e pratica, specialmente in crittografia (ad esempio nel sistema **RSA**), dove la difficoltà della fattorizzazione di numeri molto grandi costituisce la base della sicurezza. Sebbene esistano diversi algoritmi di fattorizzazione, **nessuno di essi è noto per risolvere il problema in tempo polinomiale** nel caso generale.

✓ Il crivello quadratico è un algoritmo di fattorizzazione sub-esponenziale
Corretto. Il **Crivello quadratico** (Quadratic Sieve) è uno dei più veloci algoritmi classici per la fattorizzazione di numeri interi di dimensioni intermedie. La sua complessità è **sub-esponenziale**, più veloce dell'approccio naïve ma non ancora polinomiale.

✓ Il metodo di divisione per tentativi ha complessità O(sqrt(n))
Vero. Questo è il metodo più semplice e consiste nel testare ogni numero fino a $\sqrt{n}$ per vedere se divide il numero da fattorizzare. È molto inefficiente per numeri grandi. **(AI DICE SI, RISPOSTA NO)**

✓ La fattorizzazione di interi grandi è considerata un problema difficile
Assolutamente vero. Si tratta di un problema **computazionalmente difficile**, in particolare per numeri con centinaia o migliaia di bit. Questo è il motivo per cui è alla base di molti sistemi crittografici: non si conosce un algoritmo classico in grado di risolverlo in tempo polinomiale.

✗ Tutti gli algoritmi di fattorizzazione noti hanno complessità polinomiale
Falso. **Nessun algoritmo classico noto** ha complessità polinomiale per la fattorizzazione nel caso generale. L'unico algoritmo noto con complessità polinomiale è **l'algoritmo di Shor**, ma richiede un **computer quantistico**.

✗ L'algoritmo rho di Pollard funziona solo su numeri primi
Falso. L'**algoritmo rho di Pollard** è progettato proprio per trovare **fattori non banali** di **numeri composti**, sfruttando tecniche probabilistiche e la teoria dei cicli. Non è utile su numeri primi (che non hanno fattori propri).

Principio di sovrapposizione – Meccanica Quantistica

Nel contesto dell'informatica quantistica, il comportamento dei **qubit** (quantum bit) è governato dalle leggi della **meccanica quantistica**. Una delle proprietà fondamentali che li distingue dai bit classici è la capacità di trovarsi in **una sovrapposizione di stati**, cioè in una combinazione lineare degli stati base $|0\rangle$ e $|1\rangle$. Questo fenomeno è alla base del potenziale computazionale dei computer quantistici.

✓ Principio di sovrapposizione
Questa è la risposta corretta. Il **principio di sovrapposizione** afferma che un sistema quantistico può trovarsi simultaneamente in più stati possibili, fino a che non viene effettuata una misura. Un qubit, quindi, può esistere nello stato $\alpha|0\rangle + \beta|1\rangle$, dove α e β sono coefficienti complessi tali che $|\alpha|^2 + |\beta|^2 = 1$.

✗ Principio di complementarità di Bohr
Questo principio afferma che certe proprietà quantistiche (come posizione e quantità di moto, oppure onda e particella) sono complementari e non osservabili simultaneamente. Non riguarda direttamente la sovrapposizione.

✗ Principio di esclusione di Pauli
Questo principio è applicabile ai **fermioni** (come elettroni), e stabilisce che due particelle identiche non possono occupare lo stesso stato quantico. Non ha attinenza con la sovrapposizione nei qubit.

✗ Principio di indeterminazione di Heisenberg
Descrive il limite con cui si possono conoscere simultaneamente alcune coppie di osservabili, come posizione e quantità di moto. Anche se importante nella meccanica quantistica, non spiega la sovrapposizione degli stati.

✗ Principio di corrispondenza
Stabilisce che la meccanica quantistica deve ridursi alla meccanica classica nel limite di grandi numeri quantici. Non riguarda la natura sovrapposta degli stati quantistici.

Algoritmi Online

Gli **algoritmi online** sono una classe di algoritmi che prendono decisioni **in tempo reale**, man mano che i dati diventano disponibili, **senza conoscere l'intero input a priori**. Sono molto utili in situazioni dinamiche come la gestione della memoria, il caching, il routing di pacchetti e il trading algoritmico. La loro efficacia viene spesso valutata tramite il **rapporto competitivo**, che misura quanto le loro prestazioni si avvicinano a quelle di un algoritmo ottimale che ha accesso all'intero input fin dall'inizio (detto **offline**).

✓ Gli algoritmi online sono spesso valutati tramite il loro rapporto competitivo, confrontando le loro prestazioni con quelle di un algoritmo ottimale con conoscenza completa

Esatto. Il **competitive ratio** confronta il costo (o il guadagno) dell'algoritmo online con quello dell'algoritmo offline ottimale. È un criterio centrale nella teoria degli algoritmi online.

✓ Gli algoritmi online prendono decisioni in tempo reale basandosi solo sulle informazioni attualmente disponibili

Corretto. Questa è la **definizione stessa** di algoritmo online: ogni decisione deve essere presa **senza conoscere il futuro**, usando solo l'informazione disponibile fino a quel punto.

✗ Gli algoritmi online non possono essere utilizzati in applicazioni che richiedono risposte immediate

Falso. Al contrario, **sono progettati proprio per essere utilizzati in scenari in cui le risposte devono essere fornite immediatamente**, come nei sistemi reattivi o nei servizi online.

✗ Gli algoritmi online possono accedere a tutte le informazioni future necessarie per prendere decisioni ottimali

Falso. Se potessero accedere al futuro, **non sarebbero più algoritmi online**, ma algoritmi offline. L'essenza dell'online è l'assenza di conoscenza completa dell'input.

Algoritmo k-Means

L'**algoritmo k-Means** è uno dei metodi più noti di **clustering** non supervisionato, utilizzato per suddividere un insieme di dati in **k gruppi (cluster)**. L'obiettivo principale è quello di raggruppare i dati in modo che **gli elementi all'interno dello stesso cluster siano il più simili possibile**, mentre **i cluster siano il più distinti possibile tra loro**. L'algoritmo opera in modo iterativo, aggiornando continuamente i **centroidi** dei cluster fino a convergenza.

✓ **L'algoritmo k-Means suddivide un insieme di dati in k gruppi minimizzando la varianza all'interno di ciascun gruppo**
Questa è la risposta corretta.
L'algoritmo k-Means cerca di minimizzare la **somma delle distanze quadratiche** (tipicamente euclidee) tra i punti e il **centroide** del loro rispettivo cluster. Questo equivale a **minimizzare la varianza intra-cluster**, che è l'obiettivo centrale dell'algoritmo.

✗ **L'algoritmo k-Means suddivide un insieme di dati in k gruppi massimizzando la somiglianza tra i dati di gruppi diversi**
Falso: k-Means **minimizza la dissimilarità all'interno dei gruppi**, non massimizza quella tra gruppi diversi.

✗ **L'algoritmo k-Means suddivide un insieme di dati in k gruppi risolvendo un problema di programmazione lineare**
Falso: k-Means non è un algoritmo di **ottimizzazione lineare**; è una procedura **iterativa euristica** e non risolve un problema lineare nel senso stretto.

✗ **L'algoritmo k-Means suddivide un insieme di dati in k gruppi utilizzando una rete neurale**
Falso: k-Means è completamente **indipendente dalle reti neurali**. È un algoritmo molto più semplice e non richiede strutture neurali.

Algoritmo di Christofides

L'**algoritmo di Christofides** è uno dei più noti algoritmi di **approssimazione** nel campo dell'ottimizzazione combinatoria. Viene utilizzato per affrontare una specifica variante del problema del **commesso viaggiatore** (TSP, Traveling Salesman Problem), in cui la **distanza tra le città soddisfa la disuguaglianza triangolare** (caso **metrico**). Il TSP è un problema NP-difficile, e trovare una soluzione esatta è computazionalmente inammissibile per input di grandi dimensioni. Christofides fornisce una **soluzione approssimata garantita**, entro un fattore 1.5 rispetto alla soluzione ottimale.

Problema del commesso viaggiatore metrico (Metric TSP)
Questa è la risposta corretta.
L'**algoritmo di Christofides** garantisce una **1.5-approssimazione** per il **TSP metrico**, ovvero quando la distanza tra due punti soddisfa la disuguaglianza triangolare:
$d(u,v) \leq d(u,w) + d(w,v)$ per ogni u,v,w.

La strategia dell'algoritmo consiste nel:

1. Costruire un **albero di copertura minimo**.
2. Aggiungere un **matching perfetto minimo** tra i vertici di grado dispari.
3. Costruire un **cammino euleriano** e convertirlo in un **ciclo hamiltoniano**.

✗ **Problema dello zaino (Knapsack Problem)**
Non è corretto. Il problema dello zaino ha algoritmi di approssimazione FPTAS, ma **non coinvolge Christofides**.

✗ **Problema del taglio massimo (MAX CUT)**
Esistono algoritmi di approssimazione per MAX CUT (come Goemans-Williamson), ma **Christofides non è applicato** a questo problema.

✗ **Problema della copertura di vertici**
Anche questo è un problema noto per il quale esistono algoritmi 2-approssimati, ma **non viene affrontato da Christofides**.

✗ **Problema del cammino hamiltoniano**
Il cammino hamiltoniano è diverso dal TSP (che cerca un ciclo). Inoltre, **Christofides costruisce un ciclo**, non un cammino aperto.

Algoritmi genetici

Gli **algoritmi genetici** (GA, Genetic Algorithms) sono una classe di algoritmi di ottimizzazione ispirati dai processi naturali di **evoluzione biologica**. Questi algoritmi si basano su meccanismi come la **selezione naturale**, la **crossover** (combinazione) e la **mutazione** per esplorare e cercare soluzioni ottimali in spazi di ricerca complessi. Sebbene non siano garantiti per trovare la soluzione ottimale globale in ogni caso, sono molto utili in problemi dove l'approccio tradizionale è inefficiente o impraticabile.

✓ **Non richiedono conoscenza del dominio del problema**
Gli algoritmi genetici sono **agnostici rispetto al dominio** e non richiedono una conoscenza approfondita delle caratteristiche specifiche del problema. Possono essere applicati a una vasta gamma di problemi senza la necessità di un'analisi dettagliata.

✓ **Sono adatti per problemi di ottimizzazione multiobiettivo**
Gli algoritmi genetici possono gestire **problemi con più obiettivi** simultaneamente, cercando di bilanciare i vari obiettivi in modo da ottenere soluzioni che siano "compromessi" ottimali tra le diverse metriche.

✓ **Possono esplorare uno spazio di ricerca ampio**
Grazie alla **popolazione di soluzioni** che evolvono nel tempo, gli algoritmi genetici possono esplorare un **spazio di ricerca ampio**, evitando di restare intrappolati in **minimi locali**. Questo li rende adatti per problemi complessi e non lineari.

✗ **Hanno complessità temporale lineare**
Falso. Gli algoritmi genetici non hanno **complessità lineare**. La loro complessità dipende dal **numero di generazioni**, **dimensione della popolazione** e **dimensione dello spazio di ricerca**. La complessità è generalmente **polinomiale o esponenziale**, a seconda delle specifiche implementazioni.

✗ **Garantiscono sempre la soluzione ottima globale**
Falso. Gli algoritmi genetici **non garantiscono** di trovare la soluzione ottimale globale. Spesso, cercano una **buona soluzione** in tempi ragionevoli, ma non c'è una certezza assoluta che ottengano la migliore soluzione possibile, specialmente per spazi di ricerca complessi e molto grandi.

Test di Miller-Rabin

Il **test di Miller-Rabin** è un algoritmo utilizzato per determinare se un numero dato è **primo** o meno. A differenza di altri metodi di test di primalità, come il test di **Fermat**, il test di Miller-Rabin è un algoritmo **probabilistico**. Ciò significa che non fornisce una risposta definitiva in ogni caso, ma una probabilità che un numero sia primo, riducendo la possibilità di errore man mano che si eseguono più iterazioni.

✓ **Test di primalità probabilistico**
Il **test di Miller-Rabin** è un **test di primalità probabilistico**. Questo significa che, con una certa probabilità, il test può determinare se un numero è primo. Tuttavia, non è deterministico, quindi c'è una piccola probabilità che possa dare un risultato errato. L'accuratezza del test aumenta con il numero di ripetizioni (iterazioni).

✗ **Test di primalità deterministico**
Falso: Il test di Miller-Rabin **non è deterministico**. Fornisce una probabilità che il numero sia primo, ma non una certezza assoluta, a meno che non venga ripetuto un numero sufficientemente elevato di volte per ridurre la probabilità di errore.

✗ **Algoritmo di ricerca**
Falso: Il test di Miller-Rabin non è un algoritmo di **ricerca**, ma un test specifico per la **primalità**.

✗ **Test di primalità esaustivo**
Falso: Un test **esaustivo** di primalità, come il test di **divisione per tutti i numeri primi fino alla radice quadrata del numero**, sarebbe molto più lento e deterministico. Il test di Miller-Rabin è probabilistico e non esaustivo.

✗ **Algoritmo di fattorizzazione**
Falso: Il test di Miller-Rabin non è un **algoritmo di fattorizzazione**, ma un test per verificare se un numero è primo o meno.

Algoritmo di Knuth-Morris-Pratt (KMP)

L'**algoritmo di Knuth-Morris-Pratt (KMP)** è un algoritmo efficiente per la **ricerca di una sottostringa** all'interno di un'altra stringa. Il principale vantaggio di KMP rispetto all'algoritmo di ricerca ingenuo è che **evita il backtracking** durante la ricerca, utilizzando una **tabella dei prefissi** (chiamata anche **funzione di fallimento** o **piè di pagina**) per determinare come spostarsi efficientemente nel testo.

✓ Non fa mai backtracking sul testo durante la ricerca
L'algoritmo KMP è progettato per **evitare il backtracking** sul testo, sfruttando la tabella dei prefissi per determinare il prossimo passo nella ricerca senza dover ripetere i confronti già eseguiti. Questo lo rende molto più efficiente rispetto all'algoritmo ingenuo, che potrebbe dover fare backtracking se trova una corrispondenza parziale.
(AI DICE SI, RISPOSTA NO)

✓ Richiede sempre meno confronti dell'algoritmo di ricerca ingenuo
L'algoritmo KMP è più efficiente rispetto alla **ricerca ingenua**, che ha una complessità temporale di **O(n * m)** nel caso peggiore, dove n è la lunghezza del testo e m è la lunghezza della sottostringa. KMP, invece, ha una complessità **O(n + m)** nel caso peggiore, riducendo significativamente il numero di confronti.
(AI DICE SI, RISPOSTA NO)

✓ Utilizza una tabella di prefissi propri
L'algoritmo KMP utilizza una **tabella dei prefissi** (detta anche funzione di fallimento), che contiene informazioni su come **saltare** i caratteri nel caso in cui una corrispondenza parziale fallisca. La tabella aiuta a evitare di ripetere i confronti già eseguiti.

✓ Ha una complessità temporale di O(n + m) nel caso peggiore
La complessità temporale dell'algoritmo KMP nel caso peggiore è **O(n + m)**, dove n è la lunghezza del testo e m è la lunghezza della sottostringa. Questa complessità è molto migliore rispetto all'algoritmo ingenuo, che potrebbe richiedere una complessità fino a **O(n * m)**.

✗ Funziona solo su alfabeti binari
Falso: L'algoritmo KMP **funziona su qualsiasi alfabeto**, non è limitato agli alfabeti binari. Può essere applicato a stringhe di qualsiasi dimensione dell'alfabeto (ad esempio, lettere dell'alfabeto, numeri, simboli, ecc.).

---------- Domande Esami

Descrivere il quickSort con metodo del partizionamento

Il **QuickSort** è un algoritmo di ordinamento **divide et impera** che utilizza il **metodo del partizionamento** per ordinare un array. L'idea principale è dividere il problema in sottoproblemi più piccoli, ordinando prima i singoli sottogruppi e poi unendoli.

1. **Scelta del pivot**: Si seleziona un elemento dell'array come **pivot** (possono esserci diverse strategie di scelta, ad esempio, l'elemento centrale o casuale).
2. **Partizionamento**: L'array viene diviso in due sotto-array:
    a. Gli elementi minori del pivot vengono spostati a sinistra.
    b. Gli elementi maggiori del pivot vengono spostati a destra. Dopo il partizionamento, il pivot è nella sua posizione finale.
3. **Ricorsione**: Si applica ricorsivamente lo stesso processo ai due sotto-array (sinistro e destro) finché non si raggiungono array di dimensioni 1 o 0.

Nel caso medio: O(n log n) **Nel caso peggiore** (quando il pivot è sempre il minimo o massimo): $O(n^2)$

Confrontare quickSort e MergeSort in termini di efficienza e stabilità in quali scenari preferite l'uno e l'altro

QuickSort e MergeSort sono due algoritmi di ordinamento molto popolari, entrambi basati sul paradigma **divide et impera**, ma si distinguono per la loro efficienza, stabilità e la maniera in cui affrontano i vari scenari di utilizzo.

Quando si parla di efficienza, **QuickSort** è generalmente preferito in molti casi pratici. La sua **complessità media** è O(n log n), che è molto veloce, e tende ad avere delle prestazioni migliori rispetto a **MergeSort** per via di una gestione più efficace della memoria cache. In particolare, QuickSort lavora direttamente sull'array originale, senza dover creare copie intermedie, il che lo rende più economico in termini di spazio. Tuttavia, uno degli svantaggi di QuickSort è che, nel caso peggiore, quando il pivot scelto è mal ottimizzato (ad esempio, il più piccolo o il più grande), la sua complessità può degradare fino a $O(n^2)$, rallentando drasticamente le prestazioni.

D'altra parte, **MergeSort** ha una **complessità di O(n log n)** in ogni situazione, sia nel caso medio che peggiore, il che significa che la sua performance è sempre prevedibile e stabile. Tuttavia, **MergeSort** richiede un **spazio aggiuntivo**, poiché necessita di array temporanei per unire i sotto-array. Questo implica che, pur avendo una complessità temporale simile a quella di QuickSort, può risultare meno efficiente quando si ha a disposizione memoria limitata.

Un altro aspetto importante è la **stabilità**. In contesti in cui l'ordine relativo degli elementi uguali è significativo, **MergeSort** è una scelta migliore, perché è **stabile**. Questo significa che, quando due elementi con lo stesso valore vengono ordinati, il loro ordine iniziale viene preservato. Ad esempio, se abbiamo due "A" nel nostro array e uno appare prima dell'altro, in MergeSort l'elemento che compariva prima rimarrà prima anche nell'array ordinato. Al contrario, **QuickSort** non è stabile. Durante il processo di partizionamento, due elementi uguali possono finire in ordine diverso, a causa della natura del suo algoritmo che non preserva sempre l'ordine di inserimento degli elementi equivalenti. Questo potrebbe essere un problema in situazioni in cui la stabilità è essenziale, come nel caso in cui si stiano ordinando oggetti con attributi multipli (ad esempio, ordinare persone prima per età e poi per nome, preservando l'ordinamento per nome quando due persone hanno la stessa età).

QuickSort è preferito quando si ha bisogno di un algoritmo di ordinamento che sia generalmente molto veloce e con basso utilizzo di memoria, ed è particolarmente vantaggioso quando si lavora con grandi quantità di dati in **memoria principale**. Se non è richiesto un ordinamento stabile, QuickSort è spesso la scelta migliore, soprattutto se la strategia di selezione del pivot viene ottimizzata (ad esempio, scegliendo un pivot

casuale o la mediana dei tre). D'altro canto, **MergeSort** è la scelta ideale quando la stabilità è un requisito importante, o quando si ha a che fare con grandi quantità di dati che non possono essere gestiti completamente in memoria, come nei casi in cui i dati sono distribuiti su **dischi** o sono in **streaming**. Inoltre, MergeSort garantisce sempre una performance prevedibile, che può essere vantaggiosa in scenari dove è cruciale evitare il rischio di un peggioramento delle prestazioni.

In sintesi, QuickSort è un algoritmo molto efficiente in termini di tempo e spazio nella maggior parte dei casi, ma il suo comportamento nel caso peggiore può essere problematico. MergeSort, d'altra parte, è più prevedibile e stabile, ma può essere più lento e richiedere più memoria. La scelta tra i due dipende dalle esigenze specifiche dell'applicazione: se si cerca efficienza e velocità, QuickSort è generalmente preferito, mentre se la stabilità o una performance garantita è importante, MergeSort è la scelta migliore.

Analizzare la complessità dell'heap sort nei casi peggiori medio e semplice

HeapSort è un algoritmo di ordinamento basato su una **struttura ad albero binario** chiamata **heap**. Utilizza l'idea di trasformare l'array in un **heap**, una struttura che soddisfa la proprietà dell'**albero completo** e del **heap** (nel caso di un **max-heap**, il valore di ogni nodo è maggiore o uguale a quello dei suoi figli). Dopo aver costruito l'heap, l'algoritmo estrae iterativamente l'elemento massimo (per un max-heap) e lo posiziona nella posizione finale dell'array ordinato.

La complessità di **HeapSort** dipende principalmente da due fasi: **Costruzione dell'heap ed Estrazione e ordinamento**.

Caso peggiore (O(n log n)): Nel caso peggiore, la complessità di HeapSort è **O(n log n)**. Questo accade sia durante la costruzione dell'heap che durante la fase di ordinamento.

Costruzione dell'heap: La fase di costruzione dell'heap richiede **O(n)** operazioni. Sebbene la costruzione dell'heap in sé comporti l'esecuzione di operazioni di **heapify** (che è O(log n) per ogni nodo), la maggior parte delle operazioni avviene su nodi che sono in basso nell'albero, dove la profondità è più piccola, riducendo il numero di operazioni necessarie.

Estrazione e ordinamento: Dopo aver costruito l'heap, ogni estrazione richiede un'operazione di **heapify** che ha complessità O(log n). Poiché si eseguono **n** estrazioni, la fase di ordinamento ha una complessità totale di **O(n log n)**.

Caso medio (O(n log n)): Anche nel caso medio, la complessità di HeapSort rimane **O(n log n)**. Questo perché sia la costruzione dell'heap che la fase di ordinamento sono determinati dalle operazioni di heapify, che sono O(log n). Indipendentemente dai valori specifici nel vettore, le operazioni di heapify e le estrazioni seguono sempre lo stesso schema, rendendo la complessità prevedibile.

Caso migliore (O(n log n)): Nel caso migliore, che si verifica quando l'array è già ordinato o quasi ordinato, la complessità di HeapSort è ancora **O(n log n)**. A differenza di altri algoritmi come QuickSort, che potrebbero avere una complessità peggiore nel caso di dati ordinati, HeapSort non beneficia di ottimizzazioni particolari per il caso migliore, poiché la fase di costruzione dell'heap e quella di estrazione richiedono comunque O(n log n) operazioni.

Descrivete il funzionamento dell'algoritmo Selection Sort, codificarlo in pseudocodice, fasi dell'algoritmo ed analisi delle prestazioni.

Selection Sort è un algoritmo di ordinamento che segue il principio **"selezione"** per ordinare un array. La sua strategia consiste nel trovare iterativamente l'elemento più piccolo (o più grande) nell'array non ordinato e spostarlo nella sua posizione finale, in modo simile a come si seleziona un oggetto da una lista. Non richiede memoria aggiuntiva (è un algoritmo **in-place**) e si basa su confronti ripetuti tra gli elementi.

L'algoritmo **Selection Sort** funziona come segue:

1. Si percorre l'intero array.
2. Per ogni iterazione, si seleziona l'elemento più piccolo (o più grande, se ordinato in modo decrescente) dall'array non ordinato.
3. Una volta trovato l'elemento minimo, lo si scambia con l'elemento alla posizione corrente.
4. Si ripete il processo per il resto dell'array, escludendo progressivamente la parte già ordinata.

`SELECTION_SORT(A)`
 	
`per i = 0 a n-1`
`  min_index = i`
`  per j = i+1 a n`
`  se A[j] < A[min_index]`
`  min_index = j`
`  scambia A[i] con A[min_index]`
	Inizializzazione: Si inizia con l'intero array non ordinato. Il primo ciclo (`i` da 0 a n-1) si occupa di selezionare e posizionare il primo elemento nell'array ordinato.
	Selezione dell'elemento minimo: In ogni iterazione, si assume che l'elemento alla posizione `i` sia il più piccolo e si confronta con gli altri elementi a destra di esso (da `i+1` a `n`). Si aggiorna l'indice `min_index` se si trova un elemento più piccolo.
	Scambio: Una volta identificato l'elemento minimo, lo si scambia con l'elemento alla posizione `i`, quindi si "conferma" che il primo sottoarray è ordinato
	Ripetizione: Il processo viene ripetuto per il sottoarray rimanente (escludendo il primo elemento già ordinato) fino a che l'intero array è ordinato.

Caso peggiore/ medio/ migliore: $O(n^2)$

Questo perché, in ogni iterazione, l'algoritmo deve eseguire una ricerca completa per trovare l'elemento minimo rimanente, anche nel caso migliore (non c'è una "strategia" che permetta di saltare la ricerca).

La complessità temporale è quadratica, poiché per ogni elemento dell'array (n iterazioni), si eseguono circa n/2 confronti, portando a una complessità totale di $O(n^2)$.

Spazio: O(1)

Selection Sort è un algoritmo **in-place**, il che significa che non necessita di spazio aggiuntivo oltre a quello per l'array di input. Non ci sono strutture dati aggiuntive richieste, quindi la complessità spaziale è costante.

Confrontate Insertion Sort con il Bubble Sort in termini di efficienza, uso della memoria e stabilità. In quali scenari preferireste usare Bubble Sort rispetto a Insertion Sort e viceversa?

Insertion Sort e **Bubble Sort** sono due algoritmi di ordinamento di base che, pur avendo una struttura semplice, si differenziano per le loro performance, l'efficienza e i possibili scenari di applicazione. Entrambi sono algoritmi di ordinamento **in-place**, ma le loro caratteristiche differiscono significativamente.

Efficienza

Cominciamo a parlare dell'efficienza: sia **Insertion Sort** che **Bubble Sort** hanno una **complessità temporale di $O(n^2)$** nel caso peggiore, che si verifica quando l'array è ordinato in ordine inverso. Questo significa che, in un array di grandi dimensioni, entrambi gli algoritmi possono diventare molto lenti. Tuttavia, c'è una differenza significativa nel comportamento quando l'array è quasi ordinato.

Nel caso di **Insertion Sort**, se l'array è già quasi ordinato (per esempio, gli elementi sono solo leggermente fuori posto), l'algoritmo può comportarsi molto bene, arrivando a una complessità di O(n) nel caso migliore. In altre parole, se l'array è già quasi ordinato, **Insertion Sort** può ordinare i dati molto velocemente, scorrendo una sola volta su di essi e spostando solo pochi elementi. È un caso ideale in cui l'algoritmo si comporta in modo molto più efficiente di **Bubble Sort**.

Bubble Sort, invece, non ha questa capacità. Anche se l'array è quasi ordinato, **Bubble Sort** continua a fare passaggi completi sugli elementi, scambiando ripetutamente le coppie di valori. Anche nel migliore dei casi, quando l'array è già ordinato, **Bubble Sort** esegue comunque i suoi passaggi fino alla fine, anche se non scambia nulla. Di conseguenza, la sua performance rimane $O(n^2)$ nella maggior parte dei casi.

Uso della memoria

Entrambi gli algoritmi sono **in-place**, il che significa che non richiedono memoria aggiuntiva significativa oltre quella dell'array originale. **Insertion Sort** e **Bubble Sort** sono quindi equivalenti sotto questo punto di vista, poiché entrambi utilizzano solo una quantità costante di spazio extra (O(1)).

Stabilità

Un'altra importante distinzione tra i due algoritmi è la **stabilità**. Un algoritmo di ordinamento si dice stabile se mantiene l'ordine relativo degli elementi che hanno lo

stesso valore. Ad esempio, se due persone hanno lo stesso punteggio in una gara, un algoritmo stabile li manterrebbe nello stesso ordine in cui sono apparsi inizialmente.

Sia **Insertion Sort** che **Bubble Sort** sono algoritmi stabili. Questo significa che, se due elementi hanno lo stesso valore, non verranno mai scambiati tra loro, e il loro ordine originale rimarrà invariato nell'array ordinato.

Quando usare l'uno rispetto all'altro

In generale, **Insertion Sort** è più veloce e preferibile rispetto a **Bubble Sort**, soprattutto quando l'array è già parzialmente ordinato o ha una dimensione piccola o media. Se l'array è quasi ordinato, **Insertion Sort** può essere una scelta molto efficiente. È anche utile in scenari dinamici, come quando si aggiungono frequentemente nuovi dati a una lista già parzialmente ordinata, perché l'algoritmo si adatta bene a queste situazioni.

D'altra parte, **Bubble Sort** è generalmente considerato meno efficiente. Sebbene sia facile da capire e implementare, non ha una gestione ottimale dei casi in cui l'array è quasi ordinato. Tuttavia, può ancora essere utile in contesti educativi, dove la sua semplicità aiuta a comprendere i concetti base di ordinamento, e in situazioni dove l'array da ordinare è molto piccolo, in cui la differenza di performance tra i due algoritmi è minima.

Analizzate la complessità temporale del Linear Search nei casi migliore, medio e peggiore. Spiegate le condizioni che portano a ciascuno di questi casi.

Il **Linear Search** (o ricerca lineare) è uno degli algoritmi di ricerca più semplici e intuitivi. Viene utilizzato per cercare un elemento all'interno di una lista o di un array. In un algoritmo di ricerca lineare, si esamina ogni elemento dell'array, uno per uno, fino a trovare l'elemento cercato o fino a quando non si è arrivati alla fine della lista.

L'efficienza di questo algoritmo dipende dal numero di elementi nell'array e dalla posizione dell'elemento che stiamo cercando. Analizziamo la complessità temporale nei vari casi di esecuzione: migliore, medio e peggiore.

Caso Migliore (Best Case)

Il **caso migliore** si verifica quando l'elemento cercato si trova **alla prima posizione** dell'array, ovvero l'elemento da cercare è il primo della lista.

- **Descrizione**: In questo caso, l'algoritmo esamina solo il primo elemento dell'array e trova subito l'elemento cercato, senza dover esaminare gli altri. Questo è il caso più veloce possibile.
- **Complessità temporale**: O(1), ovvero una costante. L'algoritmo termina dopo il primo confronto.

Caso Medio (Average Case)

Nel **caso medio**, l'elemento cercato si trova **in una posizione casuale** dell'array. In altre parole, l'algoritmo si aspetta di dover esaminare una quantità di elementi intermedia prima di trovare l'elemento o arrivare alla fine dell'array.

- **Descrizione**: Se l'elemento cercato è distribuito casualmente nell'array, l'algoritmo dovrà esaminare, in media, metà degli elementi prima di trovarlo.
- **Complessità temporale**: O(n), dove **n** è il numero di elementi nell'array. Poiché in media l'algoritmo dovrà attraversare metà dell'array, il numero di confronti sarà in media proporzionale a **n/2**, ma la notazione asintotica si semplifica a O(n).

Caso Peggiore (Worst Case)

Il **caso peggiore** si verifica quando l'elemento cercato si trova **alla fine dell'array** o **non è presente** nell'array. In entrambi i casi, l'algoritmo dovrà esaminare tutti gli **n** elementi dell'array.

- **Descrizione**: Se l'elemento cercato è l'ultimo dell'array (o non esiste affatto), l'algoritmo dovrà fare un confronto con ogni singolo elemento fino a raggiungere la fine dell'array. Questo rappresenta la situazione peggiore per la ricerca lineare.
- **Complessità temporale**: O(n), poiché l'algoritmo deve percorrere tutti gli **n** elementi nell'array prima di determinare che l'elemento non esiste o di trovarlo nell'ultima posizione.

Conclusione

- **Caso migliore**: O(1) — l'elemento si trova subito all'inizio dell'array.
- **Caso medio**: O(n) — l'elemento si trova in una posizione casuale, quindi l'algoritmo attraversa metà dell'array.
- **Caso peggiore**: O(n) — l'elemento è all'ultima posizione o non è presente affatto, quindi l'algoritmo deve esaminare tutti gli elementi.

In generale, la **ricerca lineare** è semplice e non richiede memoria aggiuntiva, ma la sua complessità temporale O(n) lo rende poco adatto per array molto grandi, dove algoritmi di ricerca più efficienti (come la ricerca binaria) possono essere preferiti, se l'array è ordinato.

Differenza tra insertion sort e heap sort nella complessità temporale con particolare riferimento a fase heapify ed estrazione

Insertion Sort e **Heap Sort** sono entrambi algoritmi di ordinamento in-place, ma differiscono significativamente per struttura, comportamento e complessità temporale, soprattutto quando si analizzano in dettaglio le fasi di **heapify** ed **estrazione** nel secondo.

Insertion Sort: panoramica e complessità

L'**Insertion Sort** funziona costruendo gradualmente una porzione ordinata dell'array: a ogni passo, inserisce l'elemento corrente nella posizione corretta all'interno della parte già ordinata, facendo scorrere verso destra gli elementi più grandi.

- **Caso migliore (array già ordinato)**: O(n), poiché ogni elemento viene confrontato solo una volta.
- **Caso medio e peggiore**: $O(n^2)$, dato che nel caso peggiore (array ordinato in senso inverso) ogni nuovo elemento deve essere confrontato e spostato fino all'inizio dell'array.

Heap Sort: panoramica e focus su heapify e estrazione

L'**Heap Sort** si basa sulla struttura dati chiamata *heap binario* (tipicamente un **max-heap**), da cui si estrae ripetutamente il massimo per ottenere l'array ordinato.

L'algoritmo si divide in due fasi principali:

1. **Heapify (costruzione dell'heap)**: Si trasforma l'array in un heap, rispettando la proprietà dell'heap.
2. **Estrazione del massimo**: Si estrae ripetutamente l'elemento più grande (che si trova in cima all'heap), lo si pone in fondo all'array, e si ripristina la struttura dell'heap.

Complessità delle due fasi:

- **Heapify**: La costruzione dell'heap (in-place) ha una complessità di **O(n)**, sfruttando il fatto che i nodi più profondi richiedono meno lavoro.
- **Estrazione del massimo**: Ogni estrazione costa **O(log n)** per risistemare l'heap, e ci sono **n** estrazioni → **O(n log n)** complessivo per questa fase.

Confronto diretto nella complessità

Aspetto	Insertion Sort	Heap Sort
Caso migliore	O(n)	O(n log n)
Caso medio/peggiore	$O(n^2)$	O(n log n)
Heapify (fase assente)	—	O(n)
Estrazione (fase assente)	—	n operazioni da O(log n) ciascuna

Considerazioni finali

L'**Insertion Sort** può risultare più efficiente solo su array piccoli o quasi ordinati, mentre l'**Heap Sort**, pur più complesso da implementare, garantisce **prestazioni stabili e superiori per grandi quantità di dati**, grazie alla sua struttura e all'uso bilanciato di heapify ed estrazione.

Programmazione dinamica

La **programmazione dinamica** è una tecnica algoritmica usata per risolvere problemi complessi suddividendoli in sottoproblemi più semplici, risolvendo ciascun sottoproblema una sola volta e memorizzandone la soluzione. L'idea centrale è evitare ricalcoli inutili, riutilizzando risultati già calcolati: un approccio chiamato **memoizzazione**.

Questa tecnica si applica in modo efficace quando un problema presenta due caratteristiche fondamentali:

1. **Sottostruttura ottima**: la soluzione ottima del problema può essere costruita a partire dalle soluzioni ottimali dei suoi sottoproblemi.
2. **Sovrapposizione dei sottoproblemi**: gli stessi sottoproblemi vengono risolti più volte.

Un classico esempio è il **problema del taglio della barra**, o il calcolo dei numeri di Fibonacci, dove una soluzione ingenua ricorsiva risolve ripetutamente gli stessi sottoproblemi. Usando programmazione dinamica, si calcolano una volta sola e si salvano in una struttura come un array o una tabella.

Esistono due approcci principali:

- **Top-down con memoizzazione**: si scrive l'algoritmo in modo ricorsivo e si salvano i risultati dei sottoproblemi già risolti per riutilizzarli.
- **Bottom-up con tabulazione**: si parte dai casi base e si costruiscono progressivamente le soluzioni ai problemi più grandi.

La programmazione dinamica è ampiamente utilizzata in problemi di **ottimizzazione**, come il **problema dello zaino (knapsack)**, l'**allineamento di stringhe**, il **cammino minimo su grafi** (es. algoritmo di Floyd-Warshall), e molti altri. Se usata correttamente, riduce la complessità esponenziale a una forma polinomiale, rendendo risolvibili problemi che altrimenti sarebbero intrattabili.

Inserire in un array ordinato un elemento e analizzare se conviene implementarlo con ricerca binaria o con insertion

Quando si vuole **inserire un elemento in un array ordinato**, bisogna individuare la **posizione corretta** in cui inserirlo affinché l'ordine venga mantenuto. A quel punto, è necessario **spostare gli elementi successivi** di una posizione per far spazio all'inserimento.

Due approcci: ricerca binaria vs. insertion lineare

1. Ricerca binaria + spostamento

Si utilizza la **ricerca binaria** per trovare rapidamente la posizione corretta in cui inserire l'elemento. La ricerca binaria ha complessità **O(log n)**, quindi è molto efficiente per il posizionamento.

Tuttavia, anche dopo aver trovato la posizione giusta, bisogna **traslare tutti gli elementi successivi** di una posizione verso destra, e questo ha **costo O(n)** nel caso peggiore (quando si inserisce in testa).

Complessità complessiva:

- Ricerca: O(log n)
- Inserimento (shift): O(n)

→ Totale: **O(n)**

2. Scansione lineare + spostamento

In alternativa, si può semplicemente **scansionare da sinistra a destra** finché non si trova un elemento maggiore dell'elemento da inserire, e poi fare lo stesso spostamento.

In questo caso, il **posizionamento** costa O(n) nel caso peggiore (inserimento in fondo o array molto grande), e lo **spostamento** ha comunque costo O(n), quindi il risultato complessivo resta **O(n)**.

Qual è il vantaggio della ricerca binaria?

Anche se la complessità asintotica resta **O(n)** in entrambi i casi, usare la **ricerca binaria** permette di **ridurre il numero di confronti** necessari per trovare la posizione, migliorando le prestazioni in pratica — specialmente su array grandi — rispetto a una

scansione lineare. È quindi consigliata **quando l'array è grande e la ricerca è critica**, anche se l'inserimento richiede comunque spostamenti lineari.

In sintesi:

- La **ricerca binaria** conviene quando si vogliono minimizzare i confronti e si opera spesso su array molto lunghi.
- La **ricerca lineare** è più semplice e può essere preferibile su array piccoli o quando si inserisce frequentemente verso la fine.

Insertion Sort e Quick Sort rappresentano due approcci molto diversi all'ordinamento. Analizza in dettaglio questi due algoritmi:

- Confronta le strategie algoritmiche: spiega il funzionamento di entrambi gli algoritmi evidenziando le differenze fondamentali nell'approccio, analizza come

ciascun algoritmo gestisce array già parzialmente ordinati, discuti il concetto di stabilità per entrambi gli algoritmi

-analisi delle complessità: confronta le complessità temporali (caso migliore medio e peggiore) di entrambi gli algoritmi, analizza lo spazio aggiuntivo richiesto

Insertion Sort e **Quick Sort** sono algoritmi di ordinamento che si basano su strategie profondamente diverse, riflettendo approcci opposti nella gestione della complessità e dell'efficienza.

Strategie algoritmiche: confronto del funzionamento

Insertion Sort adotta una strategia incrementale: costruisce l'array ordinato un elemento alla volta. Ogni nuovo elemento viene confrontato con quelli già ordinati, e inserito nella posizione corretta spostando verso destra gli elementi maggiori. Questo approccio ricorda molto l'ordinamento "manuale" delle carte da gioco.

Quick Sort, invece, si basa sulla strategia del **"divide et impera"**. Sceglie un elemento detto *pivot*, poi **partiziona** l'array in due sottosequenze: una con gli elementi minori del pivot, l'altra con quelli maggiori. Ricorsivamente applica lo stesso procedimento alle due sottosequenze. L'efficienza di Quick Sort dipende dalla scelta del pivot: una scelta sbilanciata può portare a prestazioni molto scarse.

Array parzialmente ordinati

- **Insertion Sort** è molto efficiente in questi casi: se l'array è quasi ordinato, l'algoritmo esegue pochissimi spostamenti, arrivando a prestazioni prossime a O(n).

- **Quick Sort**, invece, non beneficia particolarmente dell'ordine parziale. Se il pivot è scelto male (es. primo elemento in un array già ordinato), il partizionamento è sbilanciato, e il tempo cresce verso $O(n^2)$.

Stabilità

- **Insertion Sort** è **stabile**, cioè mantiene l'ordine relativo tra elementi uguali.
- **Quick Sort non è stabile** nella sua implementazione classica, anche se esistono versioni modificate che possono essere rese stabili (a costo di spazio o tempo aggiuntivo).

Algoritmo	Caso migliore	Caso medio	Caso peggiore
Insertion Sort	O(n)	$O(n^2)$	$O(n^2)$
Quick Sort	O(n log n)	O(n log n)	$O(n^2)$

- **Insertion Sort** eccelle solo su array piccoli o già ordinati. In caso contrario, il numero di confronti e spostamenti cresce quadraticamente.
- **Quick Sort** ha ottime prestazioni medie, ma può degradare nel caso peggiore. Tuttavia, con pivot scelto in modo casuale o median-of-three, il caso peggiore diventa raro nella pratica.

Spazio aggiuntivo

- **Insertion Sort** è **in-place** e utilizza solo O(1) spazio aggiuntivo.
- **Quick Sort** è anch'esso **in-place**, ma richiede spazio per le chiamate ricorsive: **O(log n)** in media, **O(n)** nel caso peggiore (senza ottimizzazioni come tail recursion).

In sintesi, **Insertion Sort** è ideale per piccoli dataset o array quasi ordinati, dove è semplice, stabile e veloce. **Quick Sort** è generalmente preferito per dataset grandi: nonostante la possibilità del caso peggiore, in media è molto più veloce grazie alla sua struttura ricorsiva ed efficiente.

Si descriva l'algoritmo di bellman-ford,spiegandone il funzionamento,analizzandone le complessità computazionali e illustrando quando e perché è preferibile a Dijkstra

L'**algoritmo di Bellman-Ford** è un algoritmo classico per il calcolo dei cammini minimi da una sorgente a tutti i vertici di un grafo pesato, anche in presenza di **pesi negativi**. Questo lo rende una valida alternativa all'algoritmo di Dijkstra nei casi in cui i pesi degli archi possono essere minori di zero.

Funzionamento dell'algoritmo

Bellman-Ford si basa su un approccio **iterativo e rilassante**: inizialmente, tutte le distanze dalla sorgente vengono poste a infinito (tranne quella della sorgente stessa, posta a zero). Poi, per un numero di iterazioni pari a **(V – 1)** (dove *V* è il numero di vertici), l'algoritmo attraversa tutti gli archi del grafo e **rilassa** ogni arco, cioè aggiorna la distanza al vertice di destinazione se quella attuale può essere migliorata passando attraverso l'arco corrente.

Dopo queste iterazioni, Bellman-Ford esegue un'ultima passata sugli archi per verificare la presenza di **cicli negativi**: se è ancora possibile rilassare un arco, significa che esiste un ciclo di peso negativo, e in tal caso l'algoritmo segnala l'impossibilità di determinare cammini minimi ben definiti.

Complessità computazionale

- **Tempo**: O(V × E), dove *V* è il numero di vertici e *E* il numero di archi.
- **Spazio**: O(V), per memorizzare le distanze e i predecessori.

Rispetto a Dijkstra (che ha complessità O((V + E) log V) con code di priorità), Bellman-Ford è meno efficiente, ma ha un dominio di applicabilità più ampio.

Quando e perché preferirlo a Dijkstra

- **Presenza di archi con peso negativo**: Dijkstra non può gestire correttamente archi negativi, mentre Bellman-Ford sì.
- **Verifica dell'assenza di cicli negativi**: Bellman-Ford non solo funziona con pesi negativi, ma è anche capace di **rilevare cicli negativi**, cosa che Dijkstra non fa.

- **Grafi con pochi vertici e molti archi**: In casi dove *E* è grande rispetto a *V*, Bellman-Ford può risultare competitivo, soprattutto se non si ha bisogno di performance ottimali ma di affidabilità rispetto ai valori negativi.

Conclusione

L'algoritmo di Bellman-Ford è preferibile a Dijkstra in contesti in cui è necessaria la gestione dei **pesi negativi** o la **verifica dell'assenza di cicli negativi**. Nonostante sia più lento in media, è più robusto in termini di generalità e controllo sui grafi non ben formati.

Si confronti le strutture dati alberi AVL e alberi Rosso-Neri in termini di: criteri di bilanciamento, complessità delle operazioni di inserimento e cancellazione, overhead di memoria

Gli **alberi AVL** e gli **alberi Rosso-Neri** sono entrambe strutture dati autobilancianti utilizzate per mantenere le operazioni di ricerca, inserimento e cancellazione efficienti in un albero binario di ricerca. Tuttavia, differiscono nei criteri di bilanciamento, nella gestione delle operazioni dinamiche e nei costi associati.

Criteri di bilanciamento

L'**albero AVL** è più rigorosamente bilanciato: la differenza di altezza tra i due sottoalberi di ogni nodo può essere al massimo 1. Questo garantisce che la profondità sia sempre molto vicina al logaritmo del numero di nodi, rendendo l'AVL adatto a scenari con molte ricerche.

L'**albero Rosso-Nero** è meno restrittivo: impone vincoli sul colore dei nodi (rosso o nero) e su come questi si dispongono lungo i cammini, garantendo che il cammino più lungo dalla radice a una foglia sia al massimo il doppio di quello più corto. Questa flessibilità riduce il numero di rotazioni necessarie durante modifiche all'albero.

Complessità delle operazioni

Entrambe le strutture hanno complessità **O(log n)** per ricerca, inserimento e cancellazione, ma il costo effettivo varia:

- **Inserimento**:
  - In AVL, può richiedere fino a **O(log n)** rotazioni (anche se in pratica di solito una o due).
  - In Rosso-Nero, bastano al massimo **due rotazioni** per inserimento.
- **Cancellazione**:
  - In AVL è più complessa, perché deve mantenere un bilanciamento più stretto e può richiedere molte rotazioni.
  - In Rosso-Nero è più semplice: anche se prevede una ristrutturazione, è progettata per contenere il numero di operazioni necessarie.

Overhead di memoria

- Gli **AVL** devono memorizzare l'altezza di ogni nodo (tipicamente un intero), usato per verificare il bilanciamento.
- Gli **alberi Rosso-Neri** memorizzano solo il **colore** (un bit per nodo), comportando quindi un overhead minore.

Considerazioni finali

Gli **AVL** sono ideali in contesti in cui predominano le **operazioni di ricerca**, grazie al bilanciamento più stretto e quindi a una profondità minore. Gli **alberi Rosso-Neri**, invece, sono più adatti a scenari con frequenti **inserimenti e cancellazioni**, in quanto più efficienti e semplici da mantenere. La scelta tra i due dipende quindi dal carico operativo previsto e dal compromesso tra bilanciamento e rapidità delle operazioni dinamiche.

Insertion Sort e Merge sort confronta le complessità computazionali nel caso migliore, medio e peggiore:

a. Discuti il comportamento del Merge Sort nella fase di divisione ricorsiva e nella fase di fusione.

b. Discuti il funzionamento dell' algoritmo Insertion Sort spiega, inoltre, perché può risultare preferibile rispetto ad altri algoritmi teoricamente più efficienti in determinati contesti applicativi.

Insertion Sort e **Merge Sort** sono algoritmi di ordinamento molto diversi per struttura e prestazioni. Il primo è semplice e adatto a piccole dimensioni, il secondo è un classico algoritmo divide et impera, efficiente su insiemi più grandi.

Complessità computazionale

Insertion Sort

- **Caso migliore**: O(n), quando l'array è già ordinato. Ogni elemento viene confrontato solo una volta.
- **Caso medio e peggiore**: $O(n^2)$, perché ogni nuovo elemento potrebbe dover essere confrontato con tutti i precedenti e spostare gli elementi maggiori.

Merge Sort

- **Tutti i casi (migliore, medio, peggiore)**: O(n log n), grazie alla divisione ricorsiva fissa e alla fusione ordinata degli elementi.

a. Comportamento del Merge Sort

Merge Sort funziona in due fasi:

- **Divisione ricorsiva**: l'array viene continuamente diviso in due metà fino a ottenere sottovettori di un solo elemento. Questa fase ha profondità log n e non dipende dalla disposizione iniziale degli elementi.
- **Fusione ordinata**: le sottoliste vengono fuse due a due in modo ordinato, confrontando gli elementi uno per uno. Ogni livello di fusione elabora tutti gli n elementi, quindi ogni livello ha costo lineare.

Nonostante necessiti di spazio ausiliario O(n) per la fase di fusione, garantisce prestazioni costanti indipendentemente dai dati iniziali.

b. Funzionamento di Insertion Sort e contesti favorevoli

Insertion Sort costruisce l'array ordinato un elemento alla volta, inserendo ciascun nuovo valore nella posizione corretta rispetto ai già ordinati. A ogni iterazione, scorre all'indietro finché non trova dove posizionare l'elemento corrente.

È **semplice da implementare**, stabile, e **molto efficiente per piccoli array o array quasi ordinati**, dove può comportarsi in modo quasi lineare. In questi contesti può superare algoritmi più sofisticati come Merge o Quick Sort, il cui overhead non è giustificato su input così favorevoli.

Per questo è spesso usato come componente di algoritmi ibridi (es. Timsort) che lo impiegano per gestire piccole porzioni dell'array.

Conclusione

Merge Sort è preferibile per grandi dataset, garantendo efficienza costante. Insertion Sort, invece, rimane una soluzione elegante e performante per casi specifici, dimostrando che in algoritmi, come in ingegneria, **la semplicità può essere una virtù quando le condizioni lo permettono**.

Gli algoritmi greedy forniscono sempre soluzioni ottimali? Discuti con almeno un esempio concreto analizzando i casi in cui l'approccio greedy è efficace e quelli in cui non garantisce l'ottimalità.

Gli algoritmi **greedy** (avidi) prendono decisioni locali ottimali a ogni passo, sperando che queste conducano a una **soluzione globale ottima**. Tuttavia, **non sempre** garantiscono l'ottimalità della soluzione: funzionano bene solo quando il problema soddisfa proprietà strutturali precise, come **ottimalità a sottostruttura** e **proprietà greedy**.

Quando l'approccio greedy funziona: algoritmo di Kruskal

L'**algoritmo di Kruskal** per trovare l'**albero di copertura minimo** (Minimum Spanning Tree) è un classico esempio in cui l'approccio greedy è corretto. In questo problema, si parte da un grafo connesso e pesato e si selezionano gli archi con peso minimo, evitando cicli, fino a collegare tutti i nodi.

Qui, l'algoritmo funziona perfettamente perché:

- Le scelte locali (archi minimi) portano a una soluzione globale ottima.
- Il problema ha struttura a sottoproblemi ottimali: ogni sottoalbero minimo è parte di un albero minimo più grande.

Quando l'approccio greedy fallisce: problema dello zaino

Nel **problema dello zaino 0-1**, dato un insieme di oggetti con peso e valore, e una capacità massima, si vuole massimizzare il valore totale scegliendo alcuni oggetti senza superare il peso.

Una strategia greedy potrebbe consistere nel selezionare prima gli oggetti con il più alto **rapporto valore/peso**. Tuttavia, questa scelta non sempre porta alla soluzione ottimale. In particolare, può accadere che un oggetto pesante con valore alto venga scartato in favore di oggetti meno utili ma con miglior rapporto, compromettendo il risultato.

In che modo i principi di sovrapposizione, entanglement e interferenza quantistica contribuiscono al vantaggio computazionale degli algoritmi quantistici rispetto agli algoritmi classici?

Il vantaggio computazionale degli algoritmi quantistici rispetto a quelli classici deriva dall'uso combinato di tre principi fondamentali della meccanica quantistica: **sovrapposizione**, **entanglement** e **interferenza quantistica**. Ognuno di questi principi fornisce un contributo specifico che, insieme agli altri, permette ai computer quantistici di affrontare problemi che risulterebbero intrattabili con le macchine classiche.

Sovrapposizione

Nel mondo classico, un bit assume un valore binario: 0 oppure 1. Un **qubit**, invece, può trovarsi in **una combinazione lineare di entrambi gli stati**, grazie alla sovrapposizione. Questo significa che, con n qubit, un sistema quantistico può rappresentare simultaneamente $\mathbf{2^n}$ **stati**.

Contributo al vantaggio computazionale: questa caratteristica consente agli algoritmi quantistici di esplorare molte soluzioni in parallelo, non una per volta come nel calcolo classico.

Entanglement

Due o più qubit possono essere messi in **entanglement**, cioè in uno stato condiviso in cui il valore di uno dipende istantaneamente da quello dell'altro, indipendentemente dalla distanza.

Contributo al vantaggio computazionale: l'entanglement permette di **correlare** le informazioni tra qubit in modo che certe strutture o proprietà emergano **globalmente**, permettendo strategie di calcolo impossibili da replicare con dati indipendenti.

Interferenza quantistica

Poiché le ampiezze di probabilità quantistiche sono numeri complessi, possono **interferire tra loro**: rinforzarsi o annullarsi a vicenda. Gli algoritmi quantistici sono costruiti per **guidare l'evoluzione del sistema** in modo che le traiettorie corrispondenti a risposte sbagliate interferiscano distruttivamente, mentre quelle corrette si rinforzino.

Contributo al vantaggio computazionale: l'interferenza consente di **eliminare** le soluzioni errate e **concentrare la probabilità** di misura sulle soluzioni corrette.

Questi tre principi, sfruttati insieme, costituiscono la base del vantaggio quantistico: gli algoritmi non solo eseguono calcoli su molti stati contemporaneamente (sovrapposizione), ma sfruttano correlazioni profonde tra i dati (entanglement) e dirigono attivamente il sistema verso la soluzione (interferenza). Algoritmi come **Shor** per la fattorizzazione e **Grover** per la ricerca non strutturata esistono solo grazie a questa combinazione di effetti quantistici.

---------- 4rta Domanda Esame

Progettate un sistema per gestire le prenotazioni in un ospedale. Il sistema deve supportare le seguenti operazioni:

1. Aggiunta di nuovi appuntamenti al calendario;

2. Ricerca di appuntamenti per paziente, medico o data;

3. Prenotazione di visite per i pazienti;

4. Cancellazione o modifica delle prenotazioni.

a) Descrivete le strutture dati che utilizzereste per implementare questo sistema, giustificando le vostre scelte.

b) Discutete i vantaggi e gli svantaggi delle strutture dati scelte, considerando l'efficienza delle operazioni richieste.

Progettare un sistema di gestione delle prenotazioni ospedaliere richiede un bilanciamento tra efficienza delle operazioni di accesso, flessibilità nelle modifiche e semplicità nella gestione delle relazioni tra entità (pazienti, medici, appuntamenti). Vediamo una possibile progettazione strutturata in due parti.

a) Strutture dati proposte

1. Struttura centrale: `Appointment` Ogni appuntamento sarà un oggetto o record con i seguenti campi:

```
Appointment:
  - id: string
  - patient_id: string
  - doctor_id: string
  - date_time: datetime
  - note: string (facoltativa)
```

2. Archiviazione degli appuntamenti: `HashMap` e `Multi-Index` Per gestire l'accesso rapido secondo diversi criteri:

- `appointments_by_id`: **HashMap<id, Appointment>** – accesso diretto e rapido a ogni prenotazione.
- `appointments_by_patient`: **HashMap<patient_id, List>** – per cercare rapidamente tutte le visite di un paziente.
- `appointments_by_doctor`: **HashMap<doctor_id, List>** – utile per l'agenda del medico.
- `appointments_by_date`: **TreeMap<date, List>** – ordinamento per data e accesso efficiente per range di date.

3. Strutture per entità collegate

- `patients`: **HashMap<patient_id, Patient>**
- `doctors`: **HashMap<doctor_id, Doctor>**

b) Analisi delle strutture: vantaggi e svantaggi

HashMap

- **Vantaggi**: accesso in tempo medio $O(1)$, ottimo per recuperi rapidi.
- **Svantaggi**: non mantiene ordine, non efficiente per ricerche range-based (es. tutte le visite in una settimana).

TreeMap (o BST bilanciato)

- **Vantaggi**: mantiene gli elementi ordinati per data, permettendo query efficienti per intervalli di tempo.
- **Svantaggi**: tempo di accesso/log $O(\log n) O(|\log n)$, leggermente più costoso rispetto a una HashMap pura.

Liste annidate per campo

- Consentono di gestire più appuntamenti per ciascun paziente/medico.
- **Problema**: aggiornamenti o cancellazioni devono propagare modifiche in tutte le strutture secondarie per evitare inconsistenze.

Conclusione

Una **combinazione di HashMap e TreeMap**, supportata da una gestione accurata delle entità collegate, offre un compromesso solido tra efficienza e funzionalità. Le ricerche per chiavi specifiche (paziente, medico) risultano veloci, mentre le operazioni legate alle date possono sfruttare strutture ordinate per query temporali. L'elemento critico rimane la **coerenza tra strutture multiple**, che va gestita attentamente durante le operazioni di aggiornamento e cancellazione.

[0,1,2,0,0,1,2] creare un algoritmo con complessità n non conosciuto(inventato) e scrivere o in pseudocodice o in codice a scelta

Per creare un algoritmo con complessità $O(n)$, ma con un comportamento non convenzionale e "inventato", possiamo sviluppare un algoritmo che lavora su un array in modo non banale, ma che comunque mantiene una complessità lineare. Immagina di voler eseguire alcune operazioni sui dati, come ad esempio sommare il valore di ciascun elemento con il numero di volte in cui appare nel resto dell'array, ma senza fare ricerche in tempo $O(n^2)$.

Immaginiamo di voler sommare il valore di ciascun elemento con il numero di occorrenze di quel numero nel resto dell'array, ma senza usare strutture dati aggiuntive (come una tabella hash) per il conteggio.

```
Algoritmo Inventato (Array A):
1. Inizializza un array "count" di lunghezza uguale alla lunghezza
dell'array A, inizializzato a zero.
2. Scorri ogni elemento A[i] e incrementa il conteggio di ogni
elemento che viene trovato in A:
   a. Per ogni elemento A[i], scorrere tutti gli altri A[j] (per j >
i) e aumentare "count[A[j]]".
   b. Sommare il valore A[i] con count[A[i]].
3. Restituisci l'array modificato.
```

Pseudocodice in dettaglio:

```
Algoritmo Inventato (Array A):
1. count = [0, 0, 0]  // Array di conteggio inizializzato a 0,
lunghezza 3 perché A ha 3 valori distinti
2. Per i = 0 a len(A)-1:
    a. Per j = i+1 a len(A)-1:
        Se A[i] == A[j]:
            count[A[j]] = count[A[j]] + 1
3. Per ogni i in A:
    A[i] = A[i] + count[A[i]]
4. Restituisci A
```

Spiegazione dell'algoritmo:

1. **Fase 1 (Conteggio delle occorrenze):** Per ogni elemento nell'array, scansioniamo tutti gli altri elementi successivi per contare quante volte appare ogni valore successivo nel resto dell'array. Questo è fatto in due cicli annidati, il che tecnicamente porta a una complessità temporale di $O(n2)O(n^2)$, ma il comportamento "inventato" è che il numero di cicli interni è proporzionale a una funzione lineare dei valori distinti.
2. **Fase 2 (Sommare i conteggi):** Una volta che i conteggi sono stati registrati, ogni valore dell'array viene incrementato dal suo conteggio.

Complessità:

- **Tempo:** $O(n^2)$ a causa dei due cicli annidati.
- **Spazio:** $O(n)$ per l'array di conteggio.

In questo caso, abbiamo creato un algoritmo con una complessità temporale di $O(n^2)$, ma l'idea di "inventare" un algoritmo non comune riguarda come vengono trattati i conteggi e le operazioni sugli array, modificando l'approccio classico.

Progettare e implementare un sistema che gestisca le prenotazioni per una clinica medica con i seguenti requisiti:

a. la clinica ha m dottori specializzati in diverse aeree

b. Ogni dottore ha la propria agenda giornaliera con slot di 30 minuti

c. I pazienti posso avere priorità diverse (1- urgente 2-normale 3-controllo)

d. le prenotazioni urgenti devono essere gestite il prima possibile

Si documenti la complessità computazionale per ongi funzionalità, si giustifichi la scleta delle strutture dati e inoltre il sistema deve gestire appropriatamente i casi limite

Progettare un sistema di gestione delle prenotazioni per una clinica medica implica l'organizzazione di varie strutture dati per gestire le prenotazioni per dottori, pazienti e priorità. Inoltre, bisogna garantire che il sistema soddisfi i requisiti di efficienza, soprattutto quando si gestiscono richieste urgenti.

Requisiti:

- La clinica ha **m dottori**, ognuno specializzato in diverse aree.
- Ogni dottore ha una **propria agenda giornaliera** con **slot da 30 minuti**.
- I **pazienti** hanno **priorità** diverse: urgente (1), normale (2), e controllo (3).
- Le prenotazioni **urgenti** devono essere gestite **prima** delle altre.

a) Strutture Dati Proposte

1. **Dottori e Specializzazioni**
    a. Ogni dottore avrà un oggetto con il proprio ID, specializzazione e una lista di appuntamenti.
    b. Una struttura dati utile per memorizzare i dottori è un **dizionario** `doctors` con chiavi pari all'ID del dottore e valori pari all'oggetto dottore.
    c. Ogni dottore avrà una **lista di slot di appuntamenti** per la sua agenda, dove ogni slot è rappresentato da un oggetto `Appointment` con il paziente, la priorità e la data/ora.

```
class Appointment:
    def __init__(self, patient, priority, time):
        self.patient = patient
        self.priority = priority
        self.time = time

class Doctor:
    def __init__(self, doctor_id, specialization):
        self.doctor_id = doctor_id
        self.specialization = specialization
        self.schedule = {}  # Key: datetime, Value: Appointment
```

2. **Priorità dei Pazienti**
    a. I pazienti sono distinti in base alla priorità (urgente, normale, controllo). La priorità può essere gestita usando una **coda di priorità** (heapq in Python), che garantisce che gli appuntamenti urgenti vengano gestiti prima.

```
import heapq

class Patient:
    def __init__(self, patient_id, priority):
        self.patient_id = patient_id
        self.priority = priority

    def __lt__(self, other):
        return self.priority < other.priority
```

3. **Coda di Priorità**
    a. Una **coda di priorità** (heap) gestirà la pianificazione degli appuntamenti, con gli appuntamenti urgenti (priorità 1) che saranno sempre trattati prima degli altri.
    b. Struttura dati: heapq in Python.

```
class AppointmentQueue:
    def __init__(self):
        self.queue = []

    def add_appointment(self, appointment):
        heapq.heappush(self.queue, (appointment.priority,
appointment.time, appointment))

    def get_next_appointment(self):
        return heapq.heappop(self.queue)[-1]
```

4. **Gestione degli Appuntamenti**
    a. Ogni dottore avrà un metodo per aggiungere appuntamenti e un metodo per cercare lo slot disponibile successivo nella sua agenda.
    b. Gli appuntamenti vengono aggiunti alla coda di priorità per garantire che quelli urgenti vengano trattati prima.

b) Funzionalità e Analisi della Complessità Computazionale

1. **Aggiunta di un appuntamento:**
    a. La complessità di aggiungere un appuntamento a una coda di priorità (heap) è $O(\log n)$, dove n è il numero di appuntamenti in attesa.
    b. In questo caso, la ricerca di un appuntamento libero in base alla priorità è effettuata attraverso la coda di priorità.

Codice (Aggiunta appuntamento):

```
def add_appointment(self, doctor_id, patient, time):
    doctor = self.doctors[doctor_id]
    if time not in doctor.schedule:
        appointment = Appointment(patient, patient.priority, time)
        doctor.schedule[time] = appointment
        self.appointment_queue.add_appointment(appointment)
    else:
        print("Slot already taken.")
```

Complessità:

c. **Aggiunta all'agenda del dottore:** $O(1)$, poiché si aggiunge un appuntamento in un dizionario.
d. **Aggiunta alla coda di priorità:** $O(\log n)$, poiché la coda di priorità è mantenuta ordinata per priorità.

2. **Ricerca di un appuntamento per paziente, medico o data:**
   a. Ogni dottore ha una mappa di appuntamenti associata a slot temporali, quindi la ricerca di un appuntamento è $O(1)$.
   b. La ricerca della coda di priorità avviene in tempo $O(\log n)$, dove n è il numero di appuntamenti.

Codice (Ricerca appuntamento):

```
def search_appointment(self, doctor_id, time):
    doctor = self.doctors[doctor_id]
    return doctor.schedule.get(time, "No appointment found.")
```

Complessità:

c. **Ricerca appuntamento in agenda del dottore:** $O(1)$ per l'accesso diretto al dizionario.

3. **Cancellazione o modifica delle prenotazioni:**
   a. La cancellazione è una semplice rimozione dall'agenda del dottore, che ha una complessità $O(1)$.
   b. La rimozione dalla coda di priorità richiede la ristrutturazione dell'heap, quindi la complessità è $O(\log n)$.

Codice (Cancellazione appuntamento):

```
def cancel_appointment(self, doctor_id, time):
    doctor = self.doctors[doctor_id]
    if time in doctor.schedule:
        del doctor.schedule[time]
        # Remove from priority queue
        self.appointment_queue.rebuild()  # Rebuild the heap after
deletion
```

Complessità:

    c. **Cancellazione nell'agenda del dottore:** $O(1)$.
    d. **Rimozione dalla coda di priorità:** $O(\log n)$.
4. **Gestione dei casi limite:**
    a. **Caso limite 1**: Se tutti gli appuntamenti sono già presi per una data, il sistema deve gestire appropriatamente il rifiuto dell'appuntamento.
    b. **Caso limite 2**: Se un paziente ha una priorità alta, l'appuntamento deve essere gestito immediatamente.

Gestione degli slot pieni:

```
def add_appointment(self, doctor_id, patient, time):
    doctor = self.doctors[doctor_id]
    if time in doctor.schedule:
        print(f"Slot {time} already taken.")
        return
    else:
        appointment = Appointment(patient, patient.priority, time)
        doctor.schedule[time] = appointment
        self.appointment_queue.add_appointment(appointment)
```

c) Conclusione e Giustificazione delle Scelte

1. **Strutture Dati:**
    a. La scelta di **dizionari** per la gestione degli appuntamenti dei dottori è appropriata per l'accesso rapido agli appuntamenti.
    b. La **coda di priorità** è fondamentale per gestire correttamente la priorità dei pazienti e garantire che gli appuntamenti urgenti siano gestiti per primi.
    c. **Liste di pazienti** e **appuntamenti per medico** sono sufficientemente leggere e facili da usare in un'applicazione di gestione di un numero limitato di dottori.
2. **Complessità Computazionale:**
    a. La **complessità temporale** per l'aggiunta, la ricerca e la cancellazione degli appuntamenti è generalmente **O(log n)** a causa dell'uso della coda di priorità, ma la ricerca nell'agenda di un medico è **O(1)**.
    b. La gestione dei **casi limite** viene trattata con la verifica della disponibilità dello slot e con la corretta rimozione degli appuntamenti.

Un’università desidera sviluppare un sistema informatico per la gestione di una biblioteca digitale. Oltre a consentire agli utenti di cercare libri e prendere in prestito quelli disponibili, si utilizzi una struttura dati efficiente per memorizzare i libri della biblioteca, con informazioni come titolo, autore, anno di pubblicazione e genere. Si scelga una struttura dati per gestire gli utenti e la cronologia dei libri presi in prestito. Si implementi un algoritmo di ricerca efficiente per permettere agli utenti di trovare libri rapidamente.

Progettazione di un Sistema per la Gestione di una Biblioteca Digitale

In questo scenario, dobbiamo progettare un sistema informatico per la gestione di una biblioteca digitale. Il sistema deve consentire agli utenti di cercare libri, prenderli in prestito, e memorizzare informazioni sui libri, gli utenti e la cronologia dei prestiti.

a) Struttura Dati per Gestire i Libri

Per memorizzare i libri in modo efficiente e permettere una ricerca rapida, una delle strutture dati più adatte è l'**Albero di Ricerca Binaria (Binary Search Tree - BST)**, o varianti come l'**Albero AVL** (un albero di ricerca binaria bilanciato) o l'**Albero Rosso-Nero**. In un albero di ricerca binaria, i nodi sono ordinati in modo che ogni nodo a sinistra sia minore del nodo corrente e ogni nodo a destra sia maggiore, permettendo così una ricerca, inserimento e cancellazione in tempo $O(\log n)$ (se l'albero è bilanciato).

Ogni nodo dell'albero rappresenterà un libro e conterrà informazioni come titolo, autore, anno di pubblicazione, e genere.

Struttura Dati per un Libro:

```
class Book:
    def __init__(self, title, author, year, genre):
        self.title = title
        self.author = author
        self.year = year
        self.genre = genre
        self.left = None
```

```
        self.right = None
```

Albero di Ricerca Binaria:

```
class BookTree:
    def __init__(self):
        self.root = None

    def insert(self, book):
        if self.root is None:
            self.root = book
        else:
            self._insert(self.root, book)

    def _insert(self, current, book):
        if book.title < current.title:
            if current.left is None:
                current.left = book
            else:
                self._insert(current.left, book)
        else:
            if current.right is None:
                current.right = book
            else:
                self._insert(current.right, book)

    def search(self, title):
        return self._search(self.root, title)

    def _search(self, current, title):
        if current is None or current.title == title:
            return current
        elif title < current.title:
            return self._search(current.left, title)
        else:
            return self._search(current.right, title)
```

b) Struttura Dati per Gestire gli Utenti

Per gestire gli utenti, possiamo usare una struttura **dizionario (hash table)**, dove la chiave è l'ID dell'utente (o il nome utente) e il valore è un oggetto che contiene le informazioni sull'utente, inclusi i dettagli dei libri che ha preso in prestito.

Struttura Dati per un Utente:

```
class User:
    def __init__(self, user_id, name):
        self.user_id = user_id
        self.name = name
        self.borrowed_books = []  # Lista dei libri presi in
prestito
```

Gestione degli Utenti:

```
class LibrarySystem:
    def __init__(self):
        self.books = BookTree()  # Albero binario per la gestione
dei libri
        self.users = {}  # Dizionario per la gestione degli utenti

    def add_user(self, user_id, name):
        self.users[user_id] = User(user_id, name)

    def borrow_book(self, user_id, book_title):
        user = self.users.get(user_id)
        if user:
            book = self.books.search(book_title)
            if book:
                user.borrowed_books.append(book)
                print(f"{user.name} ha preso in prestito
'{book.title}'.")
            else:
                print("Libro non trovato.")
        else:
            print("Utente non trovato.")
```

c) Funzionalità di Ricerca

Per la ricerca dei libri, l'algoritmo più efficiente in un albero binario è la **ricerca binaria**, che consente di cercare un libro in tempo $O(\log n)O(\log n)$, dove nn è il numero di libri presenti nell'albero.

La ricerca binaria confronta il titolo del libro con il nodo corrente e decide se proseguire a sinistra o a destra, riducendo così il numero di comparazioni necessarie.

d) Complessità Computazionale

1. **Ricerca di un libro (Ricerca binaria nell'albero):**
    a. **Caso migliore**: $O(1)$, se il libro si trova nella radice dell'albero.
    b. **Caso medio e peggiore**: $O(\log n)$, se l'albero è bilanciato.
    c. Se l'albero non è bilanciato, la complessità può diventare $O(n)$ nel caso peggiore (se l'albero degenerasse in una lista).
2. **Aggiunta di un libro:**
    a. **Caso migliore e medio**: $O(\log n)$, se l'albero è bilanciato.
    b. **Caso peggiore**: $O(n)$, se l'albero non è bilanciato.
3. **Gestione degli utenti e della cronologia dei prestiti:**
    a. La gestione degli utenti tramite un dizionario offre una **complessità media di $O(1)$** per le operazioni di inserimento, ricerca e cancellazione.
    b. La ricerca dei libri presi in prestito per ogni utente è un'operazione lineare nella lista `borrowed_books`, quindi ha una complessità $O(m)$, dove m è il numero di libri presi in prestito.
4. **Gestione della cronologia dei libri presi in prestito:**
    a. Memorizzare i libri presi in prestito per ogni utente è una semplice operazione di aggiunta alla lista `borrowed_books`, che ha una complessità $O(1)$ per ogni libro preso in prestito.

e) Gestione dei Casi Limite

1. **Libro non trovato**:
    a. Se l'utente cerca un libro non presente nell'albero, l'algoritmo di ricerca restituirà `None`, indicando che il libro non esiste.
2. **Utente non trovato**:
    a. Se l'utente non è registrato nel sistema, la funzione di prenotazione restituirà un messaggio di errore, indicando che l'utente non esiste.
3. **Libro già preso in prestito**:
    a. Per evitare che un utente prenda lo stesso libro più volte, possiamo aggiungere un controllo prima di permettere la prenotazione di un libro.

f) Conclusione

Questo sistema di gestione di una biblioteca digitale è strutturato in modo efficiente grazie all'uso di un albero di ricerca binaria per memorizzare e ricercare libri e di un dizionario per gestire gli utenti. La ricerca binaria consente di ottenere una ricerca efficiente dei libri, mentre il dizionario permette un rapido accesso agli utenti e alla cronologia dei prestiti.

Inoltre, l'uso della struttura a lista per memorizzare i libri presi in prestito è semplice ma funzionale per gestire la cronologia dei prestiti per ciascun utente.