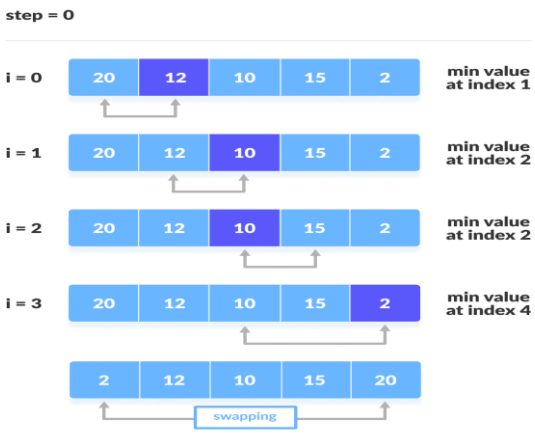
Selection Sort

Il Selection Sort e un algoritmo di ordinamento basato su confronti.

E' noto per la sua semplicità concettuale e per il fatto che effettua il numero minimo possibile di scambi tra elementi. E un algoritmo in-place (non richiede memoria aggiuntiva significativa) e non stabile nella sua forma classica. Il Selection Sort ordina un array **procedendo in questo modo**:



1.Si considera la prima posizione dell'array.

2. Si cerca il minimo tra tutti gli elementi a destra di quella posizione (inclusa).

3. Si scambia il minimo trovato con l'elemento nella posizione corrente.

4. Si ripete il processo per la posizione successiva, restringendo di volta in volta la parte dell'array da esaminare

ESEMPIO:

Supponiamo di applicare il selection sort al seguente array

A = {4, 2, 1, 3}

Ad ognuno dei passi dell'algoritmo, viene selezionato l'elemento più piccolo nella porzione di insieme ancora da ordinare, scambiandolo con l'elemento che si trova nella prima posizione di tale sottoinsieme.

i = 1 A = (4, 2,1, 3) → A = (1 | 2, 4, 3)

i = 2 A = (1 | 2, 4, 3) → A = (1, 2 | 4, 3)

i = 3 A = (1, 2, | 4,3) → A = (1, 2, 3 | 4)

i = 4 A = (1, 2, 3 | 4) → A = (1, 2, 3, 4)

Dopo 4 iterazioni, l'array sarà ordinato.

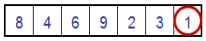
Per misurare le prestazioni dell'algoritmo (anche detta complessità) conteremo il numero di confronti.

Alla prima ripetizione del ciclo esterno vengono compiute n −1 operazioni con il ciclo interno; alla seconda ripetizione del ciclo esterno vengono eseguite n−2 operazioni con il ciclo interno, e così via fino a quando i = n.

Il numero di confronti totali sarà:

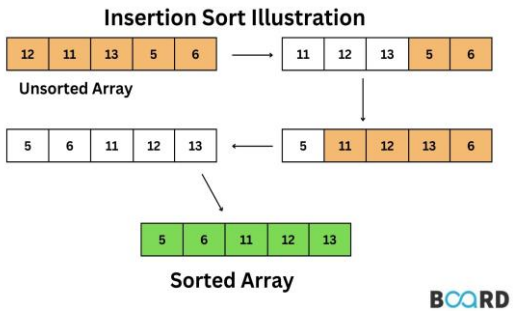
(N-1) + (N-2) + (N-3) + ... + 2 + 1 = N*(N-1)/2 = O(N^2)

Massima: O(n^2) - Ogni elemento deve essere confrontato con tutti gli altri.
Minima: O(n^2) - Anche nel caso migliore, ogni elemento deve essere confrontato con tutti.
Media: O(n^2) - In media, ogni elemento deve essere confrontato con tutti gli altri.
Spaziale: O(1) - Utilizza solo una quantità costante di spazio aggiuntivo.
Stabilità: Se due elementi hanno lo stesso valore, non verranno mai scambiati tra loro, e il loro ordine originale rimarrà invariato nell'array ordinato. Questo **NON** è il caso dell'selection sort.
Array ordinato-disordinato (Ulteriori considerazioni)
Da notare che l'algoritmo fa gli stessi confronti sia per un array disordinato, sia per un array già ordinato, ossia per il selection sort non esiste un caso migliore o peggiore ma fa sempre tutti i confronti.
Viene usato quando l'insieme da ordinare è composto da pochi elementi e quindi, anche se non è molto efficiente, ha il vantaggio di non essere troppo difficile da implementare.

SELECTION-SORT(*A*) n ← length[*A*] for j ← 1 to n - 1 do smallest ← j for i ← j + 1 to n do if *A*[i] < *A*[smallest] then smallest ← i exchange *A*[j] ↔ *A*[smallest] 	**Inizializzazione**: Si parte dal primo elemento dell'array non ordinato; **Ricerca del Minimo**: Si cerca il minimo elemento nell'array non ordinato; **Scambio**: Si scambia il minimo elemento trovato con l'elemento corrente in esame; **Ripetizione**: Si ripete il processo per ogni elemento successivo fino alla fine dell'array.

Insertion Sort

L'Insertion Sort e un algoritmo di ordinamento semplice e intuitivo. Simula il modo in cui una persona ordinerebbe un mazzo di carte in mano: ogni nuova carta viene inserita nella posizione corretta rispetto alle già ordinate. E' stabile, in-place, e particolarmente efficiente su array piccoli o quasi ordinati.



Esempio:

Vogliamo ordinare l'insieme A = {5, 3, 2, 1, 4}.

Iterazione 1. A = {5 | 2, 3, 1, 4} → A = {2, 5 | 3, 1, 4}

Iterazione 2. A = {2, 5 | 3, 1, 4} → A = {2,3, 5 | 1, 4}

Iterazione 3. A = {2, 3, 5 | 1, 4} → A = {1, 2, 3, 5 | 4}

Iterazione 4. A = {1, 2, 3, 5 | 4} → A = {1, 2, 3,4, 5}

Massima: O(n^2) - Ogni elemento deve essere confrontato con tutti gli altri.

Minima: O(n) - Nel caso migliore, l'array è già ordinato ogni elemento viene confrontato solo 1 volta.

Media: O(n^2) - In media, ogni elemento deve essere confrontato con metà degli altri elementi.

Spaziale: O(1) - Utilizza solo una quantità costante di spazio aggiuntivo.

Stabilità: Se due elementi hanno lo stesso valore, non verranno mai scambiati tra loro, e il loro ordine originale rimarrà invariato nell'array ordinato. Questo è il caso dell'insertion sort.

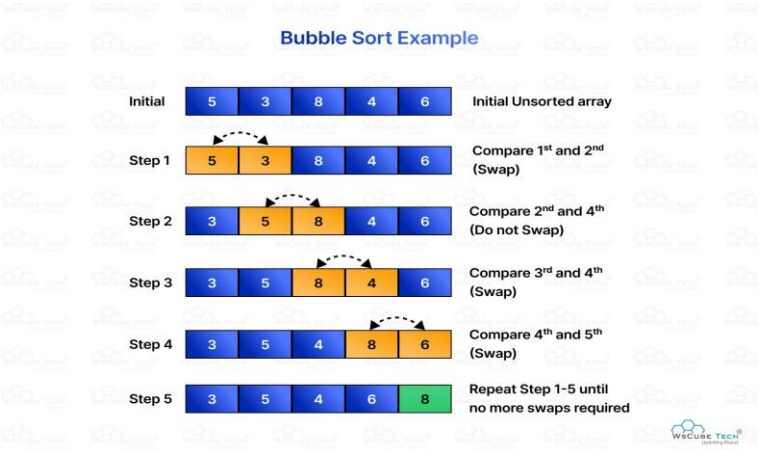
Array ordinato-disordinato (Ulteriori considerazioni)

Insertion Sort è più veloce 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.

INSERTION-SORT(A) 1 **for** $j = 2$ **to** $A.length$ 2 $key = A[j]$ 3 // Insert $A[j]$ into the sorted sequence $A[1..j-1]$. 4 $i = j - 1$ 5 **while** $i > 0$ and $A[i] > key$ 6 $A[i+1] = A[i]$ 7 $i = i - 1$ 8 $A[i+1] = key$	**Inizio**: L'algoritmo considera il primo elemento dell'array come già ordinato; **Iterazione**: A ogni passo successivo, prende il prossimo elemento e lo confronta con gli elementi già ordinati, spostandoli verso destra se sono maggiori dell'elemento in considerazione; **Inserimento**: L'elemento viene inserito nella posizione corretta; **Ripetizione**: Il processo viene ripetuto fino a che tutti gli elementi non sono ordinati.

Bubble Sort

L'algoritmo Bubble Sort è basato sullo scambio di elementi adiacenti. Ogni coppia di elementi adiacenti viene comparata e invertita di posizione, se il primo è più grande del secondo. Quando questa operazione è stata ripetuta per tutti gli elementi, si ha che l'elemento più grande si trova in fondo, a destra, della sequenza considerata.



Dopodiché, si ripete questa procedura sugli elementi rimanenti, togliendo elemento più grande trovato prima.

L'algoritmo, iterando più volte questi passaggi, fino a quando non ci saranno scambi. Allora l'array ottenuto sarà ordinato.

Deve il suo nome al modo in cui gli elementi vengono ordinati ossia quelli più piccoli "emergono" verso un'estremità della lista, così come fanno le bolle gassose in un bicchiere di acqua minerale; al contrario quelli più grandi "sprofondano" verso l'estremità opposta della sequenza.

Esempio

Abbiamo un array A = {3,5, 2, 4, 1} su cui applichiamo il bubble sort:

1. A = {3,5, 2, 4, 1} → A = {3,5,2, 4, 1} → A = {3, 2,5,4, 1} → A = {3, 2, 4,5,1} → A = {3, 2, 4, 1 | 5}
2. A = {3,2, 4, 1 | 5} → A = {2,3,4, 1 | 5} → A = {2, 3,4,1 | 5} → A = {2, 3, 1 | 4, 5}
3. A = {2,3, 1, | 4, 5} → A = {2,3,1 | 4, 5} → A = {2, 1 | 3, 4, 5}
4. A = {2,1 | 3, 4, 5} → A = {1, 2, 3, 4, 5}

Notare che, alla fine di ogni iterazione, l'elemento più grande del sottoinsieme ancora da ordinare, finisce in fondo alla sequenza, nella sua posizione definitivamente corretta, mentre la sottosequenza ancora in disordine si riduce di volta in volta di un elemento.

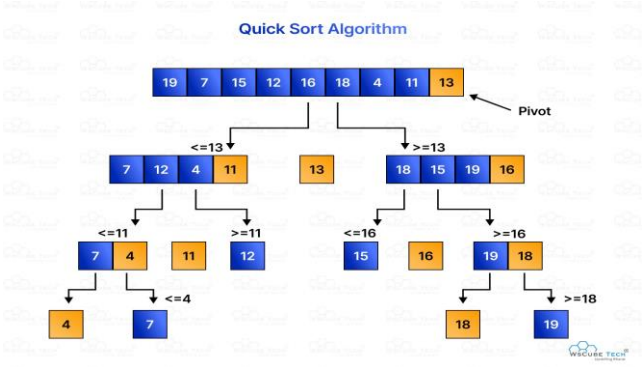
Massima: O(n^2) - Ogni elemento deve essere confrontato con tutti gli altri.
Minima: O(n) - Nel caso migliore, l'array è già ordinato e ogni elemento viene confrontato solo una volta.
Media: O(n^2) - In media, ogni elemento deve essere confrontato con metà degli altri elementi.
Spaziale: O(1) - Utilizza solo una quantità costante di spazio aggiuntivo.
Stabilità: Se due elementi hanno lo stesso valore, non verranno mai scambiati tra loro, e il loro ordine originale rimarrà invariato nell'array ordinato. Questo è il caso del bubble sort.
Array ordinato-disordinato (Ulteriori considerazioni)
Il Bubble Sort presenta un comportamento molto dipendente dall'ordine iniziale dell'array. Se l'array è già ordinato, l'algoritmo può essere leggermente ottimizzato utilizzando un flag che interrompe l'esecuzione quando non sono necessari scambi, riducendo così la complessità temporale a O(n) nel caso migliore. Tuttavia, anche in questo caso, il Bubble Sort esegue comunque una serie di confronti prima di determinare che l'array è ordinato. Se l'array è parzialmente ordinato, il comportamento è meno ottimizzato: anche se alcuni elementi sono già al posto giusto, l'algoritmo continua comunque a fare confronti e scambi.
In termini di efficienza, il Bubble Sort risulta ancora meno performante rispetto a algoritmi come l'Insertion Sort, che è in grado di gestire meglio gli array parzialmente ordinati. Infine, se l'array è completamente disordinato, il Bubble Sort esegue il massimo numero di scambi, con una complessità di O(n^2), rendendolo particolarmente inefficiente su grandi set di dati.
In generale, sebbene il Bubble Sort possa essere ottimizzato per array già ordinati, non è mai l'algoritmo migliore, soprattutto quando si lavora con array parzialmente ordinati o di dimensioni significative.

Pseudo code: **Bubble Sort(Array** $a[\,]$**)**

```
1. begin
2.    for i = 1 to n − 1
3.       for j = 1 to n − i
4.          if (a[j] > a[j + 1]) then
5.             Swap (a[j], a[j + 1])
8. end
```

Inizializzazione: Si parte dal primo elemento dell'array;
Confronto e Scambio: Si confrontano coppie di elementi adiacenti. Se un elemento è maggiore del suo successivo, i due vengono scambiati;
Ripetizione: Si ripete il processo per ogni elemento dell'array, riducendo ogni volta la lunghezza dell'array non ordinato di uno, poiché dopo ogni passaggio l'ultimo elemento è sicuramente ordinato.

Quick Sort



L'algoritmo Quick Sort, è forse il più famoso algoritmo di ordinamento, ed ha, nel caso medio, le migliori prestazioni tra gli algoritmi basati su confronto.
Adotta una strategia di tipo divide et impera, ossia divide il problema in tanti sotto problemi, fino a raggiungere dei sotto problemi facilmente risolvibili.
Questo approccio è conveniente solo se lo sforzo per ricomporre le soluzioni dei sotto problemi ed ottenere la soluzione del problema iniziale, è inferiore allo sforzo per risolvere il problema nella sua interezza.
In termini più tecnici abbiamo:
Scelta di un elemento dell'array, detto "pivot"
Suddivisione del vettore in due "sub-array"
Nel primo array tutti gli elementi minori o uguali al pivot, nel secondo quelli maggiori
Ordinare i due sub-array separatamente, ossia applicare ricorsivamente per ognuno i passi dal 1 al 4
Fondere i due sub-array ordinati insieme.
Riassumendo, abbiamo che l'ordinamento avviene mediante un algoritmo ricorsivo, che viene applicato a sotto-array sempre più piccoli, fino ad arrivare a sotto-array di pochi elementi, che possono essere risolte direttamente, senza altre chiamate ricorsive.
E' stato dimostrato che scegliendo in modo random il pivot, il Quick Sort ha una complessità media di O(n*log n) ma nel caso peggiore di O(n2).

L'ideale sarebbe però che tale risultato fosse raggiunto sempre: a ciò provvede il Merge Sort.

Massima: O(n^2) - Si verifica quando il pivot scelto è sempre il massimo o il minimo elemento.
Minima: O(n log n) - Si verifica quando il pivot divide sempre l'array in due metà uguali.
Media: O(n log n) - In media, l'algoritmo divide l'array in modo abbastanza equilibrato.
Spaziale: O(log n) - Utilizza spazio per la pila di chiamate ricorsive.
Stabilità: Quick Sort tipicamente scambia gli elementi in modo non controllato rispetto all'ordine originale. Ad esempio, se due elementi hanno lo stesso valore ma vengono separati da una partizione e poi scambiati durante la ricorsione, il loro ordine può cambiare.

Array ordinato-disordinato (Ulteriori considerazioni)

Il Quick Sort è un algoritmo di ordinamento che, sebbene molto efficiente nella maggior parte dei casi, mostra un comportamento variabile a seconda dell'ordine iniziale dell'array. Quando l'array è già ordinato o quasi ordinato, il Quick Sort può comportarsi nel peggiore dei modi, con una complessità temporale di O(n^2). Questo accade perché, scegliendo un pivot inefficace (come l'elemento più piccolo o più grande dell'array), si ottengono suddivisioni sbilanciate dell'array, aumentando il numero di operazioni necessarie per ordinare i dati.

Tuttavia, nel caso medio e nel caso peggiore, il Quick Sort ha una complessità di O(nlogn) ed è generalmente più veloce di altri algoritmi di ordinamento come il Merge Sort o l'Insertion Sort, grazie alla sua capacità di suddividere rapidamente l'array in parti più piccole. Quando l'array è disordinato, Quick Sort sfrutta al meglio il suo design basato sulla selezione del pivot per ottenere buone performance.

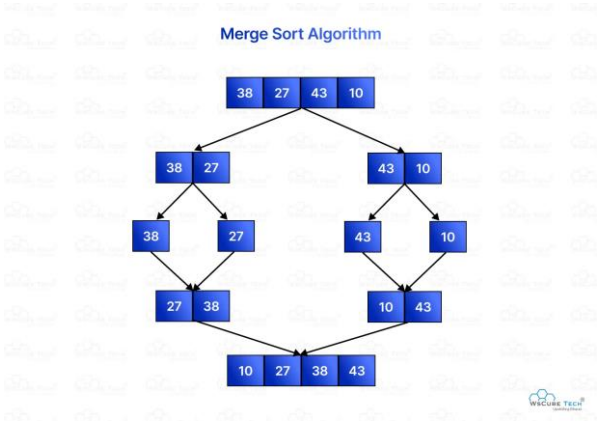
In conclusione, mentre il Quick Sort è estremamente veloce per array disordinati, la sua efficienza dipende fortemente dalla scelta del pivot e può degradare se l'array è già parzialmente ordinato o nel caso di dati con distribuzioni sfavorevoli.

Partizione(A,p,r) 1 $x \leftarrow A[r]$ 2 $i \leftarrow p-1$ 3 $j \leftarrow p$ 4 **while** j < r **do** 5 **if** $A[j] \leq x$ 6 **then** $i \leftarrow i+1$ 7 scambiare $A[i] \leftrightarrow A[j]$ 8 $j = j+1$ 9 scambiare $A[i+1] \leftrightarrow A[r]$ 10 **return** $i+1$	**Scelta del pivot**: Viene scelto un elemento dall'array, chiamato pivot. Diverse strategie possono essere utilizzate per scegliere il pivot (primo elemento, ultimo elemento, elemento medio, pivot casuale); **Partizionamento**: L'array viene riordinato in modo tale che tutti gli elementi minori del pivot siano posti alla sua sinistra e tutti gli elementi maggiori siano posti alla sua destra; **Ricorsione**: Il processo di partizionamento viene applicato ricorsivamente ai sotto array di elementi a sinistra e a destra del pivot.

Merge Sort

Il Merge Sort è molto simile al Quick Sort, ma l'elemento pivot, in questo caso, viene scelto sempre nel centro.

E' un algoritmo di ordinamento che si basa sulla strategia "**divide et impera**", ossia suddivide ricorsivamente l'array da ordinare in sottogruppi più piccoli, fino a ottenere singoli elementi che sono, per definizione, ordinati.



Successivamente, unisce questi singoli elementi in modo ordinato fino a ricostruire l'array originale, ma questa volta ordinato.

L'algoritmo è diviso in due principali metodi: MergeSort e Merge

Il metodo MergeSort è responsabile della divisione ricorsiva dell'array in due metà e della gestione della chiamata ricorsiva sui sottogruppi. Ecco i passaggi principali:

Dividere l'array: L'array viene diviso a metà. Se l'array ha più di un elemento, viene suddiviso ricorsivamente in due metà, fino a quando ogni sottogruppo contiene un solo elemento.

Ricorsività: Dopo aver diviso l'array, il MergeSort viene chiamato su ciascuna delle due metà, ripetendo la divisione fino a ottenere array di un singolo elemento.

Ritorno: La ricorsione termina quando un array ha un solo elemento (poiché un singolo elemento è già ordinato).

Il metodo Merge è il cuore dell'algoritmo, responsabile dell'unione delle due metà ordinate in un array singolo ordinato. Ecco come funziona:

Unire due metà: Quando l'array è suddiviso in sottogruppi di un singolo elemento, il metodo Merge inizia a combinare questi sottogruppi. Due sottogruppi ordinati vengono uniti confrontando il primo elemento di ciascun sottogruppo e inserendo l'elemento più piccolo nel nuovo array.

Continuare il confronto: Si continua a confrontare gli elementi rimanenti di ciascun sottogruppo e a inserirli nell'array risultante, fino a che uno dei sottogruppi non è vuoto.

Aggiungere i restanti elementi: Una volta che uno dei sottogruppi è vuoto, si copia il resto dell'altro sottogruppo direttamente nell'array finale.

Ritorno del risultato: Alla fine di questo processo, i due sottogruppi ordinati sono stati combinati in un array ordinato.

Massima: O(n log n) - L'array viene diviso a metà ad ogni passo e richiede tempo lineare per la combinazione.
Minima: O(n log n) - Anche nel caso migliore, l'array viene diviso e combinato allo stesso modo.
Media: O(n log n) - In media, l'algoritmo mantiene la stessa complessità grazie alla divisione e combinazione costante.
Spaziale: O(n) - Richiede spazio aggiuntivo per l'array temporaneo utilizzato durante il merge.
Stabilità: Se due elementi hanno lo stesso valore, non verranno mai scambiati tra loro, e il loro ordine originale rimarrà invariato nell'array ordinato. Questo **è** il caso del merge sort.

Array ordinato-disordinato (Ulteriori considerazioni)

Il Merge Sort è un algoritmo di ordinamento che si distingue per il suo comportamento altamente prevedibile, indipendentemente dall'ordine iniziale dell'array. Poiché si basa sulla tecnica "divide et impera", la sua complessità temporale è sempre O(nlog n), sia nel caso migliore, medio che peggiore. Anche se l'array è già ordinato, il Merge Sort esegue comunque il processo di divisione e fusione degli elementi, senza trarre vantaggio dalla disposizione iniziale dei dati. Questo lo rende meno efficiente rispetto ad algoritmi come l'Insertion Sort quando l'array è già parzialmente ordinato. Tuttavia, la sua stabilità e la sua prevedibilità lo rendono un'ottima scelta per situazioni in cui è necessaria una garanzia di prestazioni in tempo O(nlog n), anche su array di grandi dimensioni o disordinati. In generale, non è influenzato dal grado di ordinamento dell'array, ma la sua complessità spaziale di O(n) può essere un limite in contesti con risorse di memoria molto limitate.

Mergesort

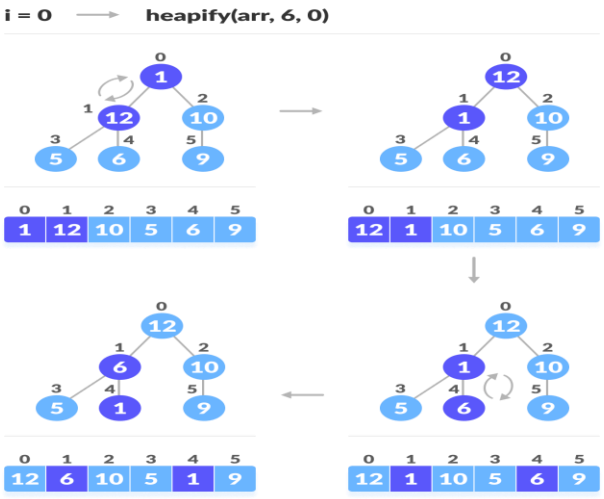
```
MergeSort(A):
    L <- mergeSort(A[1:n/2])
    R <- mergeSort(A[n/2 + 1 : n])
    return(merge(L,R))

merge(L,R):
    S <- Empty array of size m  // m = length of L + length of R
    i <- 1
    j <- 1
    for k = 1 to m
      if L[i] < R[j]:
        S[k] <- L[i]
        i <- i + 1
      else:
        S[k] <- R[j]
        j <- j + 1

    return(S)
```

Note: End cases missing in the pseudocode

La funzione MergeSort è ricorsiva e divide l'array in due metà, calcola l'indice medio. Successivamente chiama ricorsivamente sé stessa sulla metà sinistra e dopo chiama ricorsivamente a sua volta sé stessa sulla metà destra. Infine, unisce le due metà ordinate usando la funzione Merge.
La funzione Merge unisce due sotto array ordinati in un unico array ordinato, crea due array temporanei per contenere le due metà successivamente copia i dati negli array temporanei per poi confrontare gli elementi dei due temp e inserirli nell'array originale in ordine. Infine, copia eventuali elementi rimanenti.

Heap Sort

Il Heap Sort è un algoritmo di ordinamento che sfrutta la struttura dati heap, in particolare l'heap massimo, per ordinare un array. La fase fondamentale che consente di costruire e mantenere l'heap durante l'esecuzione di Heap Sort è il processo di



Cos'è un Heap?

Un heap è una struttura ad albero binario completa che soddisfa la proprietà dell'heap In un **max heap**, ogni nodo ha un valore maggiore o uguale a quello dei suoi figli. In un **min heap**, ogni nodo ha un valore minore o uguale a quello dei suoi figli.

Nel contesto di Heap Sort, lavoriamo con un max heap, in cui il valore del nodo radice (la parte superiore dell'heap) è il più grande. Questo permette di estrarre ripetutamente il massimo elemento dell'heap e posizionarlo correttamente alla fine dell'array.

Passaggi principali di Heap Sort con Heapify:

Heap Sort si svolge in due fasi principali: costruzione dell'heap e estrazione degli elementi.

Costruzione dell'Heap (**Heapify**)

Il primo passo consiste nel costruire l'heap a partire dall'array non ordinato. Questo avviene tramite l'operazione di heapify. L'operazione di heapify è utilizzata per mantenere la proprietà dell'heap (in questo caso la proprietà dell'heap massimo).

Heapify è applicato a partire dall'ultimo nodo che non è una foglia (questo è il nodo a indice n/2−1 nell'array). L'idea di base è quella di applicare heapify a ciascun nodo in modo che l'intero albero diventi un max heap.

Per un dato nodo, l'operazione heapify confronta il valore del nodo con i suoi figli. Se uno dei figli ha un valore maggiore del nodo, si scambia il valore del nodo con quello del figlio maggiore. Successivamente, l'operazione heapify viene chiamata ricorsivamente sul sottoalbero che è stato modificato, in modo da ripristinare la proprietà dell'heap in tutto l'albero.

Estrazione e Ordinamento

Una volta che l'heap è costruito, l'algoritmo esegue ripetutamente i seguenti passaggi:

Scambia la radice con l'ultimo elemento dell'array (la radice contiene l'elemento massimo).

Riduci la dimensione dell'heap di 1 (l'elemento massimo è ora nella sua posizione finale nell'array). Heapify la nuova radice per ripristinare la proprietà dell'heap. Poiché il massimo potrebbe essere stato spostato, bisogna "ripristinare" l'heap per assicurarsi che la struttura dell'heap massimo venga mantenuta. Ripeti i passaggi finché l'heap non è vuoto.

Massima: O(n log n) - La costruzione del heap e la rimozione del massimo elemento richiedono tempo logaritmico.
Minima: O(n log n) - Anche nel caso migliore, la costruzione del heap e la rimozione del massimo elemento richiedono tempo logaritmico.
Media: O(n log n) - In media, l'algoritmo mantiene la stessa complessità grazie alla struttura del heap.
Spaziale: O(1) - Utilizza solo una quantità costante di spazio aggiuntivo.

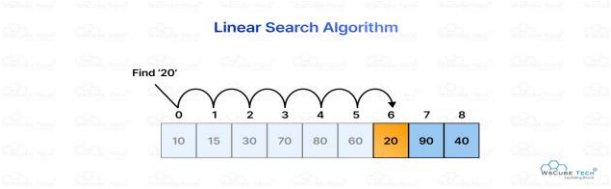
Array ordinato-disordinato (Ulteriori considerazioni)

L'Heap Sort è un algoritmo di ordinamento basato sulla struttura dati dell'heap, in particolare sull'heap massimo. La sua complessità temporale è sempre O(nlogn), sia nel caso migliore, medio che peggiore, indipendentemente dall'ordine iniziale dell'array. L'algoritmo funziona creando una struttura a heap a partire dall'array e poi estraendo ripetutamente il massimo elemento per ricostruire l'array ordinato. Poiché il processo di creazione dell'heap richiede un numero fisso di operazioni per ogni livello dell'heap, la complessità rimane O(nlogn).
Anche se l'array è già ordinato o parzialmente ordinato, l'Heap Sort non sfrutta questa disposizione, eseguendo sempre gli stessi passaggi per la costruzione dell'heap e il successivo ordinamento. Questo lo rende meno adattivo rispetto ad algoritmi come l'Insertion Sort, che può beneficiare di un array già ordinato. Inoltre, l'Heap Sort è un **algoritmo non stabile**, il che significa che non preserva l'ordine relativo degli elementi uguali. Sebbene non possieda la stessa efficienza di altri algoritmi adattivi su array parzialmente ordinati, la sua complessità di O(nlogn) lo rende particolarmente utile su grandi set di dati dove è necessario garantire prestazioni prevedibili e non si ha una forte necessità di stabilità.
Un altro vantaggio dell'Heap Sort è che opera in-place, cioè non richiede spazio aggiuntivo significativo, se non per l'array di input, rendendolo particolarmente vantaggioso in contesti con limitazioni di memoria. Tuttavia, la sua efficienza dipende fortemente dalla capacità di gestire l'heap in modo efficiente, e la sua stabilità mancante può essere un problema in alcune applicazioni.

MAX-HEAPIFY(A, i) 1 $l = \text{LEFT}(i)$ 2 $r = \text{RIGHT}(i)$ 3 **if** $l \leq A.heap\text{-}size$ and $A[l] > A[i]$ 4 $largest = l$ 5 **else** $largest = i$ 6 **if** $r \leq A.heap\text{-}size$ and $A[r] > A[largest]$ 7 $largest = r$ 8 **if** $largest \neq i$ 9 exchange $A[i]$ with $A[largest]$ 10 MAX-HEAPIFY$(A, largest)$	La funzione Max-heapify è la funzione principale, costruisce un heap massimo dall'array di input, poi estrae ripetutamente il massimo (radice) e lo mette alla fine dell'array. Confronta un nodo con i suoi figli e se necessario, scambia il nodo con il figlio più grande. Successivamente continua ricorsivamente se è stato effettuato uno scambio.

Linear Search

Linear Search è un algoritmo di ricerca semplice che controlla ogni elemento di un array uno per uno fino a trovare l'elemento desiderato o fino a quando tutti gli elementi sono stati controllati.

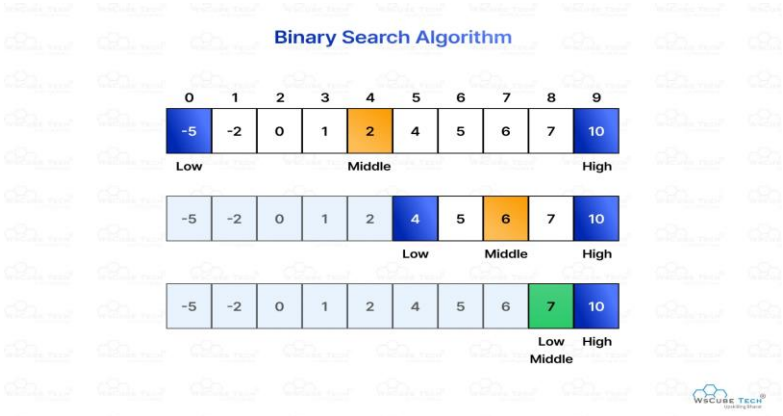


Massima: O(n) - L'elemento desiderato è l'ultimo o non è presente nell'array.
Minima: O(1) - L'elemento desiderato è il primo nell'array.
Media: O(n) - In media, l'elemento desiderato si trova a metà dell'array.
Spaziale: O(1) - Utilizza solo una quantità costante di spazio aggiuntivo.
(Ulteriori considerazioni)

```
1  def linear_search(arr, x):
2      for i in range(len(arr)):
3          if arr[i] == x:
4              return i
5      return -1
```

Il Linear Search è uno degli algoritmi di ricerca più semplici ed è utilizzato per trovare un elemento in una lista o in un array. La sua complessità temporale è sempre O(n), dove n è la dimensione dell'array, poiché l'algoritmo esplora ogni elemento dell'array uno per uno fino a trovare l'elemento cercato. Nel caso peggiore, se l'elemento non è presente o si trova all'ultimo posto dell'array, l'algoritmo dovrà esaminare tutti gli elementi. Sebbene semplice e facile da implementare, il Linear Search non è molto efficiente quando l'array è grande, poiché non sfrutta alcun tipo di informazione sull'ordinamento degli elementi. Nonostante ciò, il Linear Search è utile quando si lavora con array non ordinati o quando la lista è piccola, poiché non richiede alcuna preparazione preliminare dei dati (come l'ordinamento). Inoltre, **è un algoritmo stabile**, poiché non altera l'ordine degli elementi nell'array durante la ricerca. La sua semplicità e l'assenza di necessità di memoria aggiuntiva lo rendono ideale in contesti con risorse limitate, ma la sua efficienza ne limita l'uso in scenari in cui la velocità di ricerca è fondamentale.

Binary Search



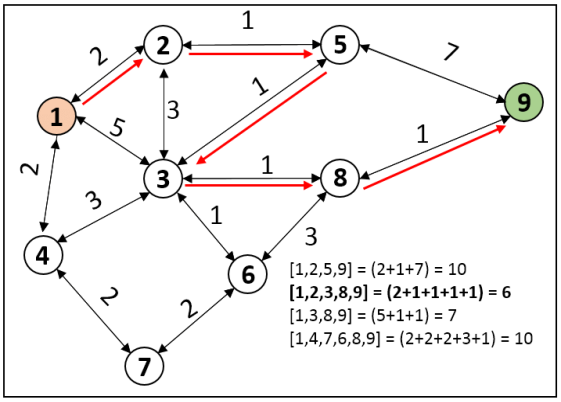
Binary Search è un algoritmo di ricerca efficiente che trova la posizione di un elemento in un array ordinato. Divide ripetutamente l'array a metà, confrontando l'elemento centrale con l'elemento desiderato, e restringe la ricerca alla metà appropriata fino a trovare l'elemento o determinare che non è presente.

Massima: O(log n) - L'array viene diviso a metà ad ogni passo.
Minima: O(1) - L'elemento desiderato è il primo elemento centrale.
Media: O(log n) - In media, l'array viene diviso a metà ad ogni passo.
Spaziale: O(1) - Utilizza solo una quantità costante di spazio aggiuntivo.
(Ulteriori considerazioni)

```python
def binary_search(arr, x):
    left, right = 0, len(arr) - 1
    while left <= right:
        mid = (left + right) // 2
        if arr[mid] == x:
            return mid
        elif arr[mid] < x:
            left = mid + 1
        else:
            right = mid - 1
    return -1
```

Il Binary Search è un algoritmo di ricerca efficiente utilizzato per trovare un elemento in un array ordinato. La sua complessità temporale è O(log n), che lo rende significativamente più veloce rispetto al Linear Search, soprattutto per array di grandi dimensioni. Il Binary Search funziona suddividendo ripetutamente l'array a metà, confrontando l'elemento centrale con il valore da cercare. Se il valore cercato è minore del valore centrale, l'algoritmo esamina la metà sinistra dell'array; se è maggiore, esamina la metà destra. Questa divisione continua fino a trovare l'elemento cercato o determinare che non è presente nell'array. Tuttavia, per poter utilizzare il Binary Search, l'array deve essere ordinato; se l'array non è ordinato, sarà necessario ordinarlo prima, il che potrebbe comportare una spesa computazionale aggiuntiva. Il Binary Search è anche un **algoritmo non stabile**, poiché non preserva l'ordine relativo degli elementi uguali, ma questa caratteristica non è rilevante in quanto non modifica effettivamente l'array durante la ricerca. Sebbene molto efficiente per array di grandi dimensioni, il Binary Search ha il limite di essere applicabile solo a dati ordinati, il che lo rende meno versatile rispetto a metodi come il Linear Search. Inoltre, il Binary Search è un algoritmo che opera in-place e non richiede memoria extra, a meno che non venga implementato in modo ricorsivo, nel qual caso può richiedere spazio per la pila di chiamate.

Dijkstra



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. Funziona mantenendo una lista dei nodi non visitati e aggiornando le distanze minime dai nodi sorgente ai nodi adiacenti. L'algoritmo di Dijkstra è un algoritmo di ricerca del cammino più breve che trova il cammino più breve da un nodo di partenza a tutti gli altri nodi in un grafo pesato, dove i pesi delle archi sono non negativi. La sua complessità temporale dipende dalla rappresentazione del grafo e dalla struttura dati utilizzata per implementare la coda di priorità. Con una coda di priorità implementata usando un heap binario, la complessità è O((V + E) * log V), dove V è il numero di nodi eE il numero di archi nel grafo. Se invece si utilizza una coda di priorità semplice, la complessità diventa O(V2), ma in generale l'uso dell'heap binario è il più efficiente.

L'algoritmo di Dijkstra funziona selezionando il nodo con la distanza più piccola, quindi esplorando i nodi adiacenti e aggiornando le distanze nel caso in cui si trovi un cammino più corto. Questo processo viene ripetuto fino a quando tutte le distanze minime dai nodi di partenza agli altri nodi non sono state determinate. Sebbene Dijkstra sia molto efficiente su grafi con pesi non negativi, non funziona correttamente se ci sono archi con pesi negativi, poiché non è in grado di gestire correttamente l'aggiornamento delle distanze in presenza di cicli negativi.

L'algoritmo di Dijkstra è non adattivo, il che significa che non beneficia di un grafo parzialmente risolto o ordinato. Ogni volta che viene eseguito, esplora tutti i possibili percorsi per determinare il cammino più breve. Dijkstra è anche un algoritmo greedy, in quanto prende decisioni ottimali locali (scegliere il nodo più vicino) sperando che queste portino a una soluzione globale ottimale. La sua complessità spaziale è O(V), poiché memorizza la distanza da ogni nodo al nodo di partenza.

L'algoritmo è ampiamente utilizzato in applicazioni come la navigazione di rete, le mappe geografiche, e l'analisi dei flussi in reti, poiché è in grado di fornire soluzioni ottimali in tempo relativamente breve, specialmente in grafi sparsi. Tuttavia, il suo comportamento dipende fortemente dalla struttura dei dati, e per grafi con pesi negativi o cicli negativi, si dovrebbero utilizzare algoritmi alternativi, come l'algoritmo di Bellman-Ford.

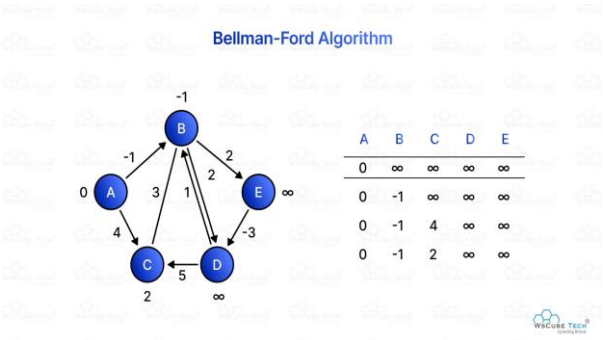
Massima: O((V + E) log V) - Dove V è il numero di nodi ed E è il numero di archi, dovuto all'uso di una coda di priorità.

Minima: O(V log V) - Nel caso migliore, ogni nodo viene visitato una volta.

Media: O((V + E) log V) - In media, l'algoritmo visita ogni nodo e arco una volta.

Spaziale: O(V + E) - La memoria necessaria per memorizzare le distanze e la coda di priorità.

Bellman-Ford



L'algoritmo di Bellman-Ford è utilizzato per trovare il percorso più breve da un nodo sorgente a tutti gli altri nodi in un grafo con pesi che possono essere negativi. Funziona rilassando ripetutamente i bordi del grafo per trovare percorsi più brevi, eseguendo questo processo per un numero di volte pari al numero di nodi meno uno. L'algoritmo di Bellman-Ford è un algoritmo di ricerca del cammino più breve che funziona su grafi con pesi negativi e consente di trovare il cammino più breve da un nodo di origine a tutti gli altri nodi in un grafo, sia che i pesi degli archi siano positivi che negativi. La sua complessità temporale è O(V·E), dove V è il numero di nodi e E è il numero di archi nel grafo. La complessità relativamente alta rispetto ad altri algoritmi come Dijkstra lo rende meno efficiente su grafi di grandi dimensioni, ma la sua capacità di gestire pesi negativi lo rende unico.

L'algoritmo di Bellman-Ford **funziona** iterando su tutti gli archi del grafo V - 1 volte, dove V è il numero di nodi. In ogni iterazione, per ogni arco (u,v) con peso w, l'algoritmo verifica se il cammino dal nodo di origine s a v passando per u è più corto rispetto al cammino precedentemente conosciuto. Se sì, aggiorna la distanza di v. Questo processo continua per V−1 iterazioni, poiché il cammino più lungo possibile in un grafo senza cicli negativi richiede al massimo V−1 archi. Se dopo V−1 iterazioni si può ancora aggiornare una distanza, significa che il grafo contiene un ciclo negativo, e l'algoritmo può rilevarlo.

L'algoritmo di Bellman-Ford è adattivo nel senso che, a differenza di Dijkstra, può rilevare la presenza di cicli negativi e gestirli, evitando risultati errati. Inoltre, è un algoritmo non greedy e dinamico, poiché esplora in modo iterativo tutte le possibilità di cammino. Tuttavia, rispetto a Dijkstra, ha una complessità temporale peggiore ed è meno efficiente per grafi molto grandi.

L'algoritmo di Bellman-Ford è molto utile in contesti in cui ci sono pesi negativi, come la modellizzazione di reti di flusso in cui i costi di alcuni archi possono essere negativi (ad esempio, quando ci sono guadagni associati al passaggio attraverso certi nodi). La sua complessità spaziale è O(V), poiché memorizza le distanze di tutti i nodi dal nodo di origine.

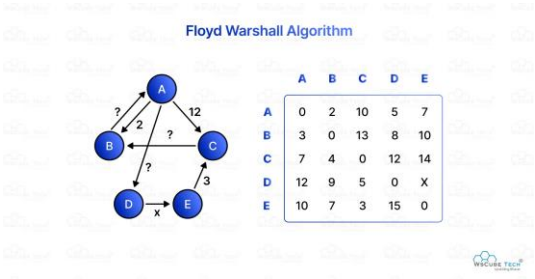
Massima: O(V * E) - Dove V è il numero di nodi ed E è il numero di archi, dovuto al rilassamento ripetuto di tutti i bordi.

Minima: O(V * E) - Anche nel caso migliore, l'algoritmo deve rilassare tutti i bordi.

Media: O(V * E) - In media, l'algoritmo rilassa tutti i bordi per ogni nodo.

Spaziale: O(V) - La memoria necessaria per memorizzare le distanze per ogni nodo.

Floyd-Warshall



L'algoritmo di Floyd-Warshall è un algoritmo di ricerca del cammino più breve che trova il cammino più breve tra tutti i coppie di nodi in un grafo ponderato, sia che i pesi degli archi siano positivi che negativi. Questo algoritmo è particolarmente utile quando si desidera determinare le distanze minime tra ogni coppia di nodi in un grafo, e la sua complessità temporale è O(V^3), dove V è il numero di nodi nel grafo. Sebbene la sua complessità cubica lo renda meno adatto per grafi di grandi dimensioni, è un algoritmo molto efficiente per grafi più piccoli o quando è necessario calcolare tutti i cammini più brevi tra coppie di nodi in un singolo passaggio.

L'algoritmo di Floyd-Warshall si basa su una tecnica di programmazione dinamica e funziona iterativamente per migliorare la soluzione per ogni coppia di nodi, considerando l'introduzione di nodi intermedi. L'idea di base è che verifica se esiste un cammino i→k→j che è più corto di quello direttamente da i a j. Questo processo continua per ogni possibile nodo intermedio k, e alla fine dell'algoritmo, la matrice risultante contiene le distanze minime tra tutte le coppie di nodi.

Il vantaggio principale dell'algoritmo di Floyd-Warshall è la sua capacità di gestire pesanti negativi: a differenza di Dijkstra o Bellman-Ford, che sono progettati per il calcolo dei cammini minimi da un singolo nodo, Floyd-Warshall è adatto a grafi completi e può determinare i cammini minimi per ogni coppia di nodi, anche quando ci sono pesi negativi. Tuttavia, se ci sono cicli negativi nel grafo, l'algoritmo di Floyd-Warshall è in grado di rilevarli, poiché le distanze minime tra un nodo e sé stesso diventano negative.

Nonostante la sua alta complessità temporale di O(V^3), che lo rende meno adatto per grafi molto grandi, l'algoritmo è adattivo in quanto non richiede una rappresentazione particolare del grafo (come un heap o una coda di priorità), ed è particolarmente utile quando si devono calcolare tutte le distanze minime tra le coppie di nodi senza dover eseguire più passaggi separati.

L'algoritmo di Floyd-Warshall è anche un algoritmo non stabile, poiché non preserva l'ordine originale degli archi nel grafo. Inoltre, la sua complessità spaziale è O(V^2), poiché memorizza una matrice delle distanze tra tutte le coppie di nodi.

In sintesi, l'algoritmo di Floyd-Warshall è ideale per calcolare i cammini più brevi tra tutte le coppie di nodi in grafi piccoli o medi, ma a causa della sua complessità cubica, non è adatto per grafi molto grandi. La sua capacità di gestire grafi con pesi negativi e rilevare cicli negativi lo rende particolarmente utile in applicazioni come la pianificazione di reti e l'analisi di flussi in grafi con pesi variabili.

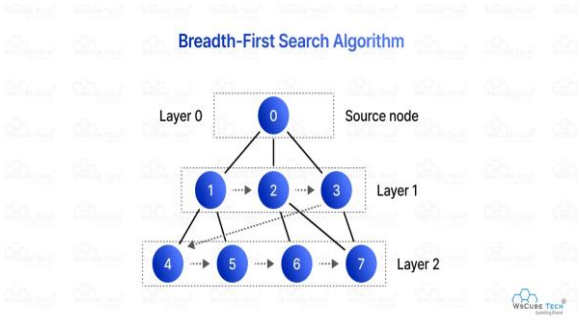
Massima: O(n^3) - L'algoritmo esegue tre cicli annidati su tutti i vertici.

Minima: O(n^3) - Anche nel caso migliore, l'algoritmo deve eseguire i tre cicli annidati.

Media: O(n^3) - In media, l'algoritmo mantiene la stessa complessità grazie ai tre cicli annidati.

Spaziale: O(n^2) - La memoria necessaria per memorizzare la matrice delle distanze.

Breadth-First Search (BFS)



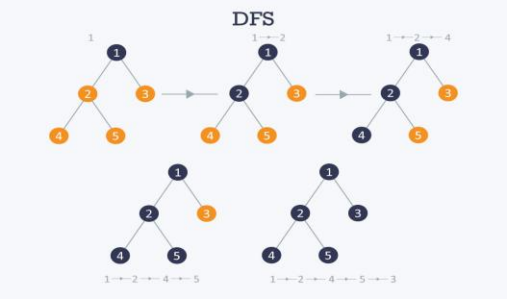
BFS (Breadth-First Search) è un algoritmo di traversamento di grafi che esplora i nodi livello per livello, partendo dal nodo iniziale e visitando prima tutti i nodi a distanza 1, poi quelli a distanza 2, e così via, fino a esplorare tutti i nodi raggiungibili. La sua **complessità temporale** è $O(V+E)$, dove V è il numero di nodi e E è il numero di archi nel grafo, poiché ogni nodo e ogni arco viene visitato al massimo una volta. BFS è implementato solitamente utilizzando una **coda** per garantire che i nodi vengano esplorati nell'ordine corretto.

Un aspetto fondamentale di BFS è che, essendo un algoritmo di esplorazione per livelli, è particolarmente adatto per **trovare il cammino più breve** in un grafo non ponderato. Poiché esplora prima i nodi più vicini al nodo iniziale, garantisce che il primo cammino trovato per ogni nodo sia il più breve. Questo lo rende ideale per applicazioni come la ricerca di percorsi più brevi in una rete, nelle mappe, o nella risoluzione di problemi di routing. Inoltre, BFS è **non orientato**, nel senso che non ha preferenze per la direzione degli archi: esplora il grafo considerando tutti gli archi uscenti da un nodo prima di passare ai nodi successivi.

Tuttavia, BFS ha alcune limitazioni. Sebbene sia molto utile per grafi **non ponderati**, non è altrettanto efficace su grafi ponderati, dove il cammino più breve potrebbe non seguire il percorso a livello (ad esempio, potrebbe esserci un arco lungo ma con un peso minore che porta a una soluzione più breve). In questi casi, algoritmi come **Dijkstra** sono più appropriati. Inoltre, mentre BFS è efficiente dal punto di vista della **complessità spaziale** (poiché richiede solo spazio per memorizzare i nodi da esplorare e quelli visitati), la quantità di memoria necessaria può crescere significativamente in grafi molto grandi o complessi.

In sintesi, **BFS** è un algoritmo estremamente utile per la ricerca di cammini minimi in grafi non ponderati e per applicazioni come la determinazione della connettività, la ricerca di soluzioni in problemi di rete, e la risoluzione di puzzle e giochi basati su grafi. La sua **semplicità**, **adattabilità** e la garanzia di trovare il cammino più breve lo rendono uno strumento fondamentale, ma la sua efficacia dipende fortemente dal tipo di grafo e dalla presenza di pesi sugli archi. **Se V è il numero di nodi (vertici). E è il numero di archi:** **Massima**: O(V + E) - L'algoritmo visita tutti i nodi e gli archi nel caso peggiore. **Minima**: O(V) - Anche nel caso migliore, l'algoritmo deve visitare tutti i nodi. **Media**: O(V + E) - L'algoritmo mantiene la stessa complessità grazie alla visita di nodi e archi. **Spaziale**: O(V) - La memoria necessaria per memorizzare i nodi visitati e la coda.

Depth-First Search (DFS)



DFS (Depth-First Search) è un algoritmo di traversamento di grafi che esplora il grafo seguendo i rami in profondità, partendo da un nodo di origine e visitando un nodo vicino finché non si raggiunge un nodo senza archi non ancora visitati. A quel punto, l'algoritmo torna indietro e esplora i nodi successivi. DFS può essere implementato utilizzando una **pila** o in modo ricorsivo, dove ogni chiamata ricorsiva rappresenta l'esplorazione di un nodo e dei suoi vicini. La **complessità temporale** di DFS è $O(V+E)$, dove V è il numero di nodi e E il numero di archi, poiché ogni nodo e ogni arco vengono visitati al massimo una volta.

DFS è particolarmente utile in molte applicazioni in cui è necessario esplorare un grafo in profondità o risolvere problemi che richiedono un'esplorazione esaustiva. Per esempio, viene usato per rilevare cicli in un grafo, per trovare componenti connesse in un grafo non orientato, o in algoritmi di **backtracking**, come nel caso della risoluzione di problemi come il Sudoku o il problema delle otto regine. Un'altra applicazione importante di DFS è nella **topological sorting** di un grafo diretto aciclico (DAG), in cui gli oggetti devono essere ordinati secondo una gerarchia di dipendenze.

Una delle principali **limitazioni** di DFS è che non garantisce di trovare il cammino più breve in un grafo, come invece fa BFS per grafi non ponderati. Inoltre, in grafi molto profondi, l'algoritmo può incorrere in problemi di **overflow della pila** a causa della ricorsione, soprattutto in linguaggi che non gestiscono bene la profondità della pila. Un altro svantaggio di DFS è che non è **adattivo** rispetto alla struttura del grafo, poiché esplora comunque tutti i rami fino in fondo, anche se potrebbe esserci una soluzione ottimale più vicina.

In sintesi, **DFS** è un algoritmo molto potente per l'esplorazione di grafi, risoluzione di problemi di ricerca in profondità, e applicazioni come il rilevamento di cicli, il backtracking e la topological sorting. Tuttavia, la sua mancanza di garanzia nel trovare il cammino più breve e la sua inefficienza in alcuni grafi molto profondi o ramificati lo rendono meno adatto in contesti in cui è importante trovare soluzioni ottimali o in cui la memoria è una risorsa limitata.
Se V è il numero di nodi (vertici). E è il numero di archi:
Massima: O(V + E) - L'algoritmo visita tutti i nodi e gli archi nel caso peggiore.
Minima: O(V) - Anche nel caso migliore, l'algoritmo deve visitare tutti i nodi.
Media: O(V + E) - In media, l'algoritmo mantiene la stessa complessità grazie alla visita di nodi e archi.
Spaziale: O(V) - La memoria necessaria per memorizzare i nodi visitati.

BFS VS DFS

BFS (Breadth-First Search) e **DFS (Depth-First Search)** sono entrambi algoritmi fondamentali di traversamento di grafi, ma differiscono significativamente nel modo in cui esplorano i nodi e nelle loro applicazioni.

Il **BFS** esplora il grafo livello per livello, partendo dal nodo iniziale e visitando tutti i nodi a distanza 1 prima di passare a quelli a distanza 2, e così via. La sua complessità temporale è $O(V+E)$, dove V è il numero di nodi e E è il numero di archi del grafo.

BFS è utilizzato principalmente per trovare il cammino più breve in un grafo non ponderato, poiché esplora tutte le possibili strade a una distanza fissa dal nodo iniziale prima di esplorare livelli più profondi. La struttura dati principale per implementare BFS è la **coda**, che consente di mantenere l'ordine di esplorazione dei nodi.

In contrasto, **DFS** esplora il grafo "profondamente", ossia va il più lontano possibile lungo ogni ramo prima di tornare indietro e considerare altri rami. DFS utilizza una **pila** (o ricorsione) per tenere traccia dei nodi da esplorare. La sua complessità temporale è anch'essa $O(V+E)$, ma DFS esplora i rami in modo diverso, non garantendo il cammino più breve, ma piuttosto esplorando un singolo cammino fino alla fine prima di fare marcia indietro. DFS è particolarmente utile per la ricerca di cicli nei grafi, la verifica di connettività e la ricerca di soluzioni in problemi di tipo "backtracking", come il problema delle otto regine.

Le **differenze principali** tra **BFS** e **DFS** riguardano il modo in cui esplorano il grafo e le situazioni in cui sono più adatti. **BFS** è più adatto quando si cerca il cammino più breve in un grafo non ponderato o quando è necessario esplorare i nodi a livello di distanza, come nella ricerca di percorsi in una rete. **DFS**, d'altra parte, è più adatto per esplorazioni in profondità, come quando si deve determinare la connettività di un grafo, rilevare cicli o risolvere problemi che richiedono esplorazioni exhaustive (come il backtracking).

Entrambi gli algoritmi sono **adattivi** in quanto esplorano il grafo in modo dinamico, ma DFS può essere meno efficiente in alcuni casi, come quando il grafo ha molti rami e il cammino desiderato si trova lontano nel grafo. Inoltre, **BFS** è garantito per trovare il cammino più breve in un grafo non ponderato, mentre DFS potrebbe non farlo.

In sintesi, **BFS** è preferito quando è necessaria una ricerca per livelli o il cammino più breve, mentre **DFS** è utile per esplorazioni profonde, rilevamento di cicli e risoluzione di problemi di backtracking. La scelta tra i due dipende dalle caratteristiche specifiche del problema e dal tipo di grafo su cui si sta lavorando.

Array

Un **array** è una struttura dati che memorizza una collezione di elementi dello stesso tipo in posizioni di memoria contigue. Gli array sono utilizzati per memorizzare dati in modo ordinato e permettono l'accesso rapido agli elementi tramite un indice.

```python
# Creazione di un array
array = [1, 2, 3, 4, 5]

# Accesso agli elementi
print(array[0])  # Output: 1
print(array[2])  # Output: 3

# Modifica di un elemento
array[1] = 10
print(array)  # Output: [1, 10, 3, 4, 5]
```

Gli array sono utili quando si ha bisogno di memorizzare e accedere a una sequenza di elementi in modo efficiente. Sono comunemente utilizzati in algoritmi che richiedono accesso casuale agli elementi, come la ricerca binaria.

L'accesso a un elemento è veloce, con complessità $O(1)$, ma le operazioni di **inserimento** e **cancellazione** possono essere più costose. Quando si inserisce un nuovo elemento in una posizione specifica, gli elementi successivi devono essere spostati di una posizione verso destra per fare spazio, comportando una complessità di $O(n)$. Analogamente, quando un elemento viene cancellato, gli elementi successivi devono essere spostati verso sinistra per colmare il vuoto, con la stessa complessità di $O(n)$. Queste operazioni possono risultare inefficienti, specialmente in array di grandi dimensioni, poiché richiedono lo spostamento di numerosi elementi.

Gli **array statici** hanno una dimensione fissa, mentre gli **array dinamici** possono adattarsi automaticamente, ma richiedono un ridimensionamento che può essere costoso. Sebbene offrano un rapido accesso e siano efficienti per operazioni su dati sequenziali, gli array sono meno flessibili rispetto ad altre strutture dati, come le liste, quando si tratta di modifiche frequenti alla loro dimensione.

Accesso: O(1) - L'accesso a un elemento tramite l'indice è costante.
Ricerca: O(n) - Nel caso peggiore, è necessario scorrere tutti gli elementi.
Inserimento: O(n) - Nel caso peggiore, è necessario spostare tutti gli elementi.
Cancellazione: O(n) - Nel caso peggiore, è necessario spostare tutti gli elementi.

Liste

```python
# Creazione di una lista
lista = [1, "ciao", 3.14, True]

# Accesso agli elementi
print(lista[0])  # Output: 1
print(lista[1])  # Output: ciao

# Modifica di un elemento
lista[1] = "salve"
print(lista)  # Output: [1, "salve", 3.14, True]

# Aggiunta di un elemento
lista.append("nuovo elemento")
print(lista)  # Output: [1, "salve", 3.14, True, "nuovo elemento"]

# Rimozione di un elemento
lista.remove(3.14)
print(lista)  # Output: [1, "salve", True, "nuovo elemento"]
```

Una **lista** è una struttura dati che memorizza una sequenza di elementi, ma a differenza degli array, gli elementi non sono memorizzati in posizioni contigue in memoria. Ogni elemento (nodo) contiene un riferimento al successivo (e, in alcune implementazioni, al precedente). Questo permette alle liste di crescere dinamicamente senza dover ridimensionare l'intera struttura. Le operazioni di **inserimento** e **cancellazione** sono generalmente più efficienti rispetto agli array. In particolare, l'inserimento o la cancellazione di un elemento in una lista non richiede lo spostamento di altri dati, ma solo il riallacciamento dei riferimenti. Tuttavia, l'inserimento o la cancellazione in una posizione specifica richiede comunque una ricerca dell'elemento, che ha una complessità di O(n) nel caso peggiore. L'accesso diretto agli elementi in una lista, al contrario, richiede di attraversare la lista a partire dalla testa (o dalla coda, nel caso di una lista doppia), comportando una complessità di o(n).

Rispetto agli **array**, le liste non sono limitate dalla dimensione fissa e possono adattarsi facilmente a nuove dimensioni. Tuttavia, a causa dell'accesso sequenziale agli elementi, le liste sono meno efficienti in operazioni che richiedono un accesso frequente e diretto agli elementi rispetto agli array. Le liste sono particolarmente utili in scenari in cui sono necessarie frequenti modifiche alla struttura (come inserimenti e cancellazioni), ma non sono ideali quando l'accesso rapido agli elementi è un requisito fondamentale.

Accesso: O(1) - L'accesso a un elemento tramite l'indice è costante.
Ricerca: O(n) - Nel caso peggiore, è necessario scorrere tutti gli elementi.
Inserimento: O(1) - L'aggiunta di un elemento alla fine della lista è costante.
Cancellazione: O(n) - Nel caso peggiore, è necessario spostare tutti gli elementi.

Stack

```python
# Creazione di uno stack utilizzando una lista
stack = []

# Aggiunta di elementi (push)
stack.append(1)
stack.append(2)
stack.append(3)
print(stack)  # Output: [1, 2, 3]

# Rimozione di elementi (pop)
print(stack.pop())  # Output: 3
print(stack.pop())  # Output: 2
print(stack)  # Output: [1]
```

Uno **stack** è una struttura dati che segue il principio **LIFO (Last In, First Out)**, ovvero l'ultimo elemento inserito è il primo ad essere estratto. Gli stack sono utilizzati per gestire dati in modo che l'accesso avvenga solo all'elemento più recente, come in un "contenitore" dove si aggiungono e rimuovono gli elementi in ordine inverso rispetto a quello in cui sono stati inseriti. Le operazioni principali su uno stack sono **push** (inserimento di un elemento) e **pop** (rimozione dell'elemento più recente). Entrambe le operazioni hanno una complessità temporale di $O(1)$, poiché l'accesso avviene solo all'elemento in cima allo stack, senza la necessità di scorrere gli altri elementi.

La **cancellazione** di un elemento dallo stack è altrettanto efficiente, poiché consiste solo nel rimuovere l'elemento in cima senza influire sugli altri dati. Tuttavia, uno **stack** non supporta l'accesso casuale agli elementi, come invece fanno array o liste, poiché l'accesso è limitato solo all'elemento superiore. Questo lo rende ideale per applicazioni come la gestione della profondità di una funzione in un linguaggio di programmazione (come nella gestione della chiamata di funzione o nell'algoritmo di backtracking).

Gli stack sono particolarmente utili in contesti in cui è necessario lavorare con i dati in ordine inverso, o quando è importante che l'elemento più recente venga trattato per primo. Tuttavia, la loro **limitazione** risiede nell'assenza di supporto per operazioni che richiedono l'accesso a più di un elemento alla volta, rendendoli meno adatti in scenari in cui è necessario un accesso casuale o a più dati simultaneamente.

In sintesi, uno **stack** è efficiente in scenari che richiedono operazioni rapide e controllate su un solo elemento alla volta, ma meno adatto quando si richiede accesso simultaneo o casuale agli elementi.

Accesso: O(n) - Nel caso peggiore, è necessario scorrere tutti gli elementi.
Ricerca: O(n) - Nel caso peggiore, è necessario scorrere tutti gli elementi.
Inserimento: O(1) - L'aggiunta di un elemento alla fine dello stack è costante.
Cancellazione: O(1) - La rimozione dell'ultimo elemento dello stack è costante.

Code

```python
from collections import deque

# Creazione di una coda
queue = deque()

# Aggiunta di elementi (enqueue)
queue.append(1)
queue.append(2)
queue.append(3)
print(queue)  # Output: deque([1, 2, 3])

# Rimozione di elementi (dequeue)
print(queue.popleft())  # Output: 1
print(queue.popleft())  # Output: 2
print(queue)  # Output: deque([3])
```

Una **coda** è una struttura dati che segue il principio **FIFO (First In, First Out)**, ovvero il primo elemento inserito è il primo ad essere estratto. Le operazioni principali su una coda sono **enqueue** (inserimento di un elemento) e **dequeue** (rimozione dell'elemento più vecchio, cioè quello in testa alla coda). Entrambe le operazioni sono efficienti, con una complessità temporale di O(1), poiché l'inserimento avviene alla fine della coda e la rimozione all'inizio, senza la necessità di spostare gli altri elementi.

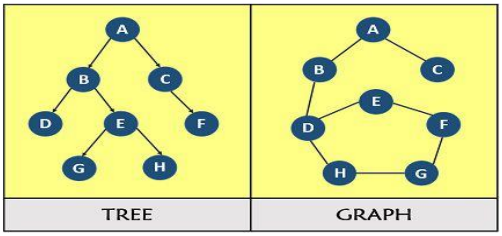
Le **code** sono particolarmente utili in scenari che richiedono l'elaborazione di dati in ordine di arrivo, come nella gestione delle richieste in un sistema operativo, nella stampa di documenti o nella gestione di eventi in un'applicazione. Una coda garantisce che gli elementi vengano trattati nell'ordine in cui sono stati ricevuti, rendendola ideale per questi tipi di applicazioni.

Rispetto ad altre strutture dati come gli **stack** o le **liste**, le code non permettono l'accesso diretto agli elementi. L'accesso è limitato solo all'elemento in testa per la rimozione e all'elemento in coda per l'inserimento, il che significa che non è possibile accedere agli elementi intermedi senza rimuovere quelli precedenti. Tuttavia, le code sono molto efficienti per operazioni di inserimento e rimozione sequenziale, mantenendo l'ordine di arrivo dei dati.

In sintesi, una **coda** è una struttura dati ottimale per gestire flussi di dati che devono essere elaborati in ordine di arrivo. Pur essendo efficiente nelle operazioni di inserimento e rimozione, presenta la limitazione di non consentire l'accesso diretto agli elementi intermedi, il che la rende meno adatta in scenari che richiedono l'elaborazione non sequenziale dei dati.

Accesso: O(n) - Nel caso peggiore, è necessario scorrere tutti gli elementi.
Ricerca: O(n) - Nel caso peggiore, è necessario scorrere tutti gli elementi.
Inserimento: O(1) - L'aggiunta di un elemento alla fine della coda è costante.
Cancellazione: O(1) - La rimozione del primo elemento della coda è costante.

Alberi



Un albero è una struttura dati gerarchica composta da nodi con un singolo nodo radice e zero o più sottoalberi. Gli alberi sono utili per rappresentare relazioni gerarchiche, come ad esempio in un sistema di file o un'organizzazione aziendale.
Un albero N-ario è un albero in cui ogni nodo può avere fino a N figli, diversamente da un albero binario che ha al massimo due figli. Questo tipo di albero è utile in scenari come la rappresentazione di strutture gerarchiche con più ramificazioni.
Un albero heap è un albero binario completo in cui i nodi rispettano una proprietà di ordinamento, per esempio in un heap massimo il valore di ogni nodo è maggiore o uguale a quello dei suoi figli. Gli heap sono spesso utilizzati per implementare strutture come le code di priorità.
Un albero bilanciato è un albero in cui la differenza di altezza tra i sottoalberi di un nodo non è troppo grande. Questo garantisce operazioni di ricerca, inserimento e cancellazione con una complessità logaritmica O(log n). Gli alberi bilanciati più comuni sono gli alberi AVL e Red-Black.
Un albero di ricerca binaria (BST) è un albero binario in cui, per ogni nodo, il valore dei suoi figli sinistro è minore del valore del nodo, mentre il valore dei suoi figli destro è maggiore. Questo permette di eseguire operazioni di ricerca, inserimento e cancellazione in tempo O(logN) nella media, ma in caso di albero non bilanciato la complessità può arrivare a O(n).

Alberi - Proprietà

Radice: Il nodo principale da cui partono tutti gli altri nodi.

Foglie: Nodi che non hanno figli.

Altezza: La lunghezza del percorso più lungo dalla radice a una foglia.

Profondità: La lunghezza del percorso dalla radice a un nodo specifico.

Livello: La distanza di un nodo dalla radice.

Sottoalbero: Un albero formato da un nodo e tutti i suoi discendenti.

Accesso: O(n) - Nel caso peggiore, è necessario visitare tutti i nodi.

Ricerca: O(n) - Nel caso peggiore, è necessario visitare tutti i nodi.

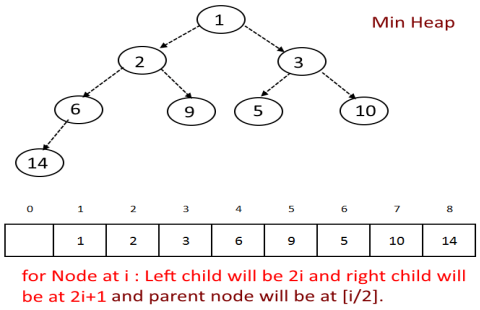
Inserimento: O(1) - L'aggiunta di un nodo è costante se si conosce il nodo genitore. **Cancellazione**: O(n) - Nel caso peggiore, è necessario visitare tutti i nodi per trovare il nodo da rimuovere.

Heap

Un **heap** è una struttura dati basata su un albero binario completo che soddisfa la proprietà di ordinamento. Esistono due tipi principali di heap: **min-heap** e **max-heap**.

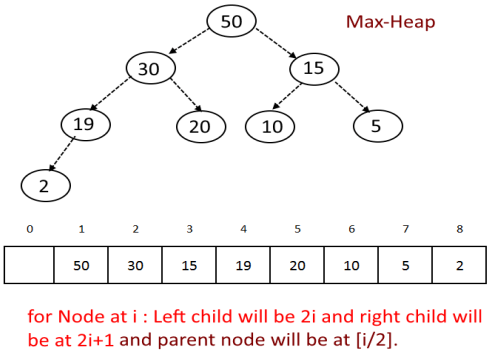
Min-Heap

Un **heap minimo** è un albero binario completo in cui ogni nodo ha un valore minore o uguale ai suoi figli. La radice dell'heap minimo contiene il valore più piccolo. Le operazioni di inserimento e rimozione del minimo sono analoghe a quelle dell'heap massimo, con una complessità di $O(\log n)$.



Max-Heap

Un **heap massimo** è un albero binario completo in cui ogni nodo ha un valore maggiore o uguale ai suoi figli. La radice dell'heap massimo contiene il valore più grande. In questo modo, l'elemento massimo è sempre facilmente accessibile in tempo $O(1)$, e le operazioni di inserimento e rimozione del massimo hanno una complessità $O(\log n)$, dove n è il numero di elementi.



Accesso: O(1) - L'accesso al minimo (min-heap) o al massimo (max-heap) è costante.
Inserimento: O(log n) - L'inserimento di un nuovo elemento richiede la riorganizzazione dell'heap.
Cancellazione: O(log n) - La rimozione del minimo (min-heap) o del massimo (max-heap) richiede la riorganizzazione dell'heap.

Gli **heap** sono molto utili per implementare strutture come **code di priorità**, dove si devono gestire gli elementi in base alla loro priorità (massima o minima). Grazie alla loro proprietà di ordinamento, le operazioni di inserimento e rimozione sono efficienti e garantiscono l'accesso rapido all'elemento con la priorità più alta (nell'heap massimo) o più bassa (nell'heap minimo). Tuttavia, gli heap non supportano l'accesso diretto a elementi arbitrari, come accade con gli array, e sono meno versatili in scenari che richiedono un accesso casuale agli elementi.

Hash Table

Una hash table (o tabella di hash) è una struttura dati che memorizza coppie chiave-valore, utilizzando una funzione di hash per mappare le chiavi a posizioni specifiche in una tabella (array). L'accesso a un elemento in una hash table avviene in tempo costante, O(1), nella maggior parte dei casi, grazie alla funzione di hash che calcola direttamente la posizione dell'elemento.

```python
# Creazione di una hash table utilizzando un dizionario
hash_table = {}

# Inserimento di coppie chiave-valore
hash_table["nome"] = "Alice"
hash_table["età"] = 25
hash_table["città"] = "Roma"

# Accesso ai valori tramite chiavi
print(hash_table["nome"])  # Output: Alice
print(hash_table["età"])   # Output: 25

# Modifica di un valore
hash_table["età"] = 26
print(hash_table)  # Output: {'nome': 'Alice', 'età': 26, 'città': 'Roma'}

# Cancellazione di una coppia chiave-valore
del hash_table["città"]
print(hash_table)  # Output: {'nome': 'Alice', 'età': 26}
```

La funzione di hash è una funzione che trasforma la chiave in un indice nell'array. Tuttavia, può verificarsi una collisione quando due chiavi diverse mappano alla stessa posizione. Le collisioni vengono gestite tramite varie tecniche, come:

Chaining: Ogni posizione nell'array contiene una lista (o una struttura simile) di tutti gli elementi che hanno la stessa posizione calcolata dalla funzione di hash.

Open addressing: In caso di collisione, la tabella cerca una nuova posizione libera secondo un determinato schema, come linear probing, quadratic probing, o double hashing.

Le operazioni principali di una hash table, come inserimento, ricerca e rimozione, sono in genere molto veloci, con una complessità di O(1), ma nel caso di molte collisioni, le operazioni potrebbero degradare a O(n).

Le hash table sono particolarmente utili per scenari che richiedono un accesso rapido ai dati tramite una chiave, come nelle applicazioni di ricerca di dati o nelle strutture di cache. Tuttavia, l'efficacia di una hash table dipende fortemente dalla qualità della funzione di hash e dalla gestione delle collisioni. Inoltre, le hash table non sono adatte quando l'ordine degli elementi è importante, poiché non mantengono alcuna sequenza tra gli elementi. **Accesso**: O(1) - L'accesso a un valore tramite la chiave è costante nel caso medio.

Ricerca: O(1) - La ricerca di un valore tramite la chiave è costante nel caso medio.

Inserimento: O(1) - L'inserimento di una nuova coppia chiave-valore è costante nel caso medio.

Cancellazione: O(1) - La cancellazione di una coppia chiave-valore è costante nel caso medio.

Hash Table - Gestione delle Collisioni

Quando due chiavi diverse producono lo stesso indice, le hash table utilizzano i bucket per gestire queste collisioni. Un bucket è una lista di coppie chiave-valore che condividono lo stesso indice. Utilizzano inoltre vari metodi per gestire le collisioni, ovvero situazioni in cui due chiavi diverse producono lo stesso indice:

Chaining

La **chaining** risolve le collisioni utilizzando liste collegate. Quando due chiavi producono lo stesso indice, entrambe vengono memorizzate in una lista collegata associata a quell'indice.

Open Addressing

L'**open addressing** risolve le collisioni cercando un'altra posizione libera nell'array. Esistono vari metodi di probing per trovare la posizione successiva disponibile.

Linear Probing

Il **linear probing** cerca la posizione successiva libera in modo sequenziale

Quadratic Probing

Il **quadratic probing** cerca la posizione successiva libera aumentando l'indice in modo quadratico.

Double Hashing

Il **double hashing** utilizza una seconda funzione di hash per determinare l'incremento dell'indice.

Versus tra i Sistemi di Memorizzazione: Utile per Esercizio di progettazione 4

StrutturaDati	Pro	Contro	Caso d'uso (Esempio Reale)
Heap	- Efficiente per implementare **code di priorità**. - Inserimento e rimozione del massimo/minimo in **O(log n)**. - Opera in **in-place**, senza necessità di memoria aggiuntiva.	Non adatto per la **ricerca arbitraria**. Non mantiene un ordine completo, solo parziale.	**Sistema ospedaliero**: Gestione di una **coda di priorità** per le emergenze, dove i pazienti con priorità più alta sono trattati prima. **Gestione risorse**: Pianificazione e schedulazione di processi con priorità diverse.
Stack	Operazioni **push** e **pop** in **O(1)**. Ottimo per **ricorsione** e **parsing** di espressioni.	Non adatto per **ricerca** di un elemento. Non mantiene l'ordine completo degli elementi.	**Sistemi di gestione documentale**: **Parsing** di espressioni matematiche o sintattiche in applicazioni come un compilatore. **Sistema ospedaliero**: Gestione delle **chiamate ricorsive** nei processi di analisi o diagnosi.
Queue	Operazioni **enqueue** e **dequeue** in **O(1)**. Adatta per **gestire processi e richieste** in un ordine FIFO.	Non ideale per **ricerca** di un elemento. Non mantiene ordine tra gli elementi.	**Magazzino**: **Gestione ordini** di prodotti in base all'ordine di arrivo (FIFO). **Call center**: Gestione delle **richieste in coda**, dove i primi a chiamare sono i primi a ricevere risposta.
Array	Accesso diretto agli elementi tramite indice in **O(1)**. Memorizzazione compatta senza overhead.	**Inserimento** e **cancellazione** lenti in **O(n)**. Dimensione fissa, non dinamica.	**Sistema di inventario magazzino**: Memorizzazione di un **insieme fisso** di prodotti con accesso rapido tramite codice prodotto. **Sistema di gestione biblioteca**: Memorizzazione dei **dati di libri** con accesso rapido tramite identificatore.
Liste (Linked List)	Inserimenti e cancellazioni rapidi in **O(1)**. - Dinamica e adatta per strutture dati di **dimensione variabile**.	La **ricerca** richiede **O(n)**. Maggiore overhead di memoria rispetto agli array.	**Sistema di gestione utenti**: Gestione di **elenco dinamico di prestiti di libri** in una biblioteca. **Sistemi ospedalieri**: Gestione di una **lista dinamica di pazienti** in attesa di trattamento.
Hash Table	Operazioni di **ricerca**, **inserimento** e **rimozione** in **O(1)** (nel caso ideale). - Ottima per memorizzare dati **unici** tramite una chiave.	Gestione delle **collisioni** può ridurre le prestazioni a **O(n)**. Non mantiene ordine tra gli elementi.	**Biblioteca digitale**: Memorizzazione di **dati unici** come **ISBN, titolo** e **autore** per ogni libro. **Sistema di gestione ospedaliera**: Memorizzazione e ricerca rapida dei **dati del paziente** tramite un **ID unico**.

Heap: Perfetto per implementare **code di priorità**, come la gestione delle emergenze in ospedali o la schedulazione di attività. Non è ottimale per la ricerca di dati specifici.

Stack: Utilizzato per **chiamate ricorsive** in compiti di diagnostica, parsing e gestione di espressioni. La sua utilità nelle strutture a LIFO è evidente in contesti come il parsing di sintassi o operazioni ricorsive.

Queue: Ottimo per **gestire ordini e richieste in FIFO** come in magazzini o call center. Efficiente nella gestione delle risorse in ordine di arrivo.

Array: Ideale per situazioni in cui si ha un set fisso di dati, come la gestione di prodotti in un magazzino o libri in una biblioteca, dove l'accesso rapido agli elementi è fondamentale.

Liste: Eccellenti per **strutture dinamiche**, come la gestione di utenti o prestiti in una biblioteca, dove l'inserimento e la rimozione rapidi sono richiesti.

Hash Table: Perfetta per applicazioni che richiedono **accesso rapido a dati unici**, come la gestione di libri in una biblioteca digitale o la gestione di pazienti tramite un ID unico.

Versus tra Algoritmi: Utile per Esercizio di progettazione 4

Algoritmo	Pro	Contro	Caso d'uso (Esempio Reale)
Insertion Sort	Efficiente per array **piccoli** o **quasi ordinati**. Facile da implementare. **Stabile**, preserva l'ordine degli elementi uguali.	Complessità di **O(n^2)** nel caso peggiore. Non scalabile per grandi dataset.	**Gestione ordini in un negozio**: Se gli ordini sono pochi e **quasi ordinati**, Insertion Sort è perfetto per aggiornamenti rapidi. **Gestione di piccole liste** in una **biblioteca digitale**.
Heap Sort	**O(n log n)** in tutti i casi. Eseguito **in-place** senza memoria extra. Funziona bene su grandi dataset e in **time complexity garantita**.	Non stabile, non preserva l'ordine degli elementi uguali. Più lento di altri algoritmi di ordinamento come Quick Sort in pratica.	**Gestione delle risorse in tempo reale**: Ordinamento di **priorità** per compiti o eventi, come in un sistema di **schedulazione di processi**. **Sistema di gestione di risorse in ospedali**: Ordinamento e gestione di **liste di attesa per interventi chirurgici**.
Selection Sort	Semplice da implementare. Funziona bene per **array di piccole dimensioni**.	**O(n^2)** nel caso peggiore. Non stabile, non preserva l'ordine degli elementi uguali. Inefficiente per grandi dataset.	**Ordinamento di piccole liste di libri** in una **biblioteca digitale**. **Schedulazione** in un **sistema di gestione risorse** con una quantità limitata di compiti.
Bubble Sort	Semplice da implementare. Funziona bene su piccole liste **quasi ordinate**.	Complessità di **O(n^2)** nel caso peggiore. Estremamente inefficiente per dataset di grandi dimensioni.	**Gestione di un elenco di libri** in una **biblioteca** dove gli elementi sono quasi ordinati e si deve fare un **piccolo aggiustamento**.
Quick Sort	Efficiente in media con complessità di **O(n log n)**. Funziona bene per grandi dataset. Eseguito **in-place** senza necessità di memoria extra.	Complessità nel caso peggiore di **O(n^2)**. Non stabile, non preserva l'ordine degli elementi uguali.	**Ordinamento di dati in un sistema bancario**: Gestione di enormi dataset per **ordini** o **transazioni**, dove la velocità è critica. **Gestione di libri in una biblioteca** con milioni di titoli.
Merge Sort	Complessità sempre **O(n log n)**, anche nel caso peggiore. Stabile, preserva l'ordine degli elementi uguali. Ottimale per grandi dataset.	Richiede spazio **O(n)** aggiuntivo per l'array temporaneo. Non è **in-place**.	**Ordinamento di cataloghi di grandi dimensioni** in una **biblioteca digitale**. **Gestione di dati in sistemi sanitari** dove la stabilità dell'ordinamento è cruciale.
Linear Search	Semplice da implementare. Non richiede ordinamento dei dati.	Complessità di **O(n)**, inefficiente su grandi dataset. Lento rispetto alla ricerca binaria per array ordinati.	**Sistema di gestione delle prenotazioni in un ospedale**: Ricerca di un **paziente** nella lista degli ospedalizzati. **Ricerca di libri per titolo o autore** in una **biblioteca digitale**.
Binary Search	Efficiente per **array ordinati**, complessità **O(log n)**. Molto veloce su grandi dataset.	Funziona solo su **array ordinati**. Richiede un array statico e ordinato.	**Ricerca rapida di libri per ISBN** in una **biblioteca digitale**. **Ricerca di un prodotto in un magazzino** dove gli articoli sono ordinati per codice prodotto.

Insertion Sort: Ottimo per piccoli array o array quasi ordinati, come quelli che si trovano in **sistemi di gestione di piccole liste** in una **biblioteca digitale**.

Selection Sort: Simile a Insertion Sort, ma meno efficiente in pratica, ideale per **piccole liste** come l'ordinamento di titoli in una biblioteca.

Bubble Sort: Facile da implementare e adatto per **piccole modifiche** in un array già parzialmente ordinato, come aggiornare rapidamente l'ordinamento dei libri in una **biblioteca**.

Quick Sort: Molto veloce per **grandi dataset**, ma rischia di peggiorare nei casi di **partizioni squilibrate**. Perfetto per sistemi come **banche**, dove i dati sono numerosi e l'ordinamento rapido è necessario.

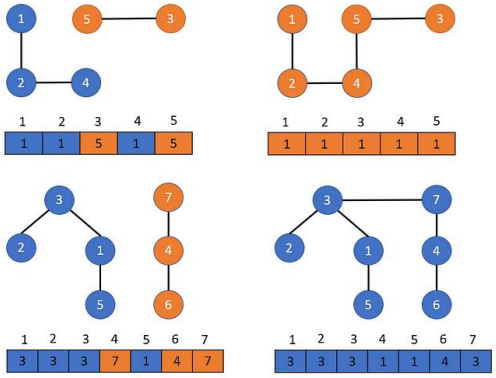
Merge Sort: Affidabile e **stabile**, perfetto per grandi set di dati e sistemi dove l'ordine degli elementi uguali è importante, come una **biblioteca digitale** con milioni di libri.

Heap Sort: Molto utile in scenari dove è necessario un ordinamento **affidabile e in-place** con una complessità garantita di **O(n log n)**. Ad esempio, la gestione delle **risorse e priorità** in **sistemi di schedulazione**.

Linear Search: Utile in scenari dove i dati non sono ordinati e la ricerca non è frequente, come cercare un libro per titolo o autore in una **biblioteca**.

Binary Search: Ideale per **ricerche veloci** in **array ordinati**, come trovare un prodotto in un **magazzino** o cercare un libro per ISBN in una **biblioteca digitale**.

Union-Find



La struttura dati **Union-Find** (o **Disjoint-Set**) è utilizzata per gestire una collezione di insiemi disgiunti e supporta due operazioni principali: **union** e **find**. Questa struttura è utile per determinare se due elementi appartengono allo stesso insieme e per unire due insiemi. La struttura Union-Find è comunemente utilizzata in algoritmi di grafi, come l'algoritmo di Kruskal per trovare l'albero di copertura minimo, e in problemi di connettività.

```python
class UnionFind:
    def __init__(self, n):
        self.parent = list(range(n))
        self.rank = [0] * n

    def find(self, u):
        if self.parent[u] != u:
            self.parent[u] = self.find(self.parent[u])  # Path compression
        return self.parent[u]

    def union(self, u, v):
        root_u = self.find(u)
        root_v = self.find(v)
        if root_u != root_v:
            if self.rank[root_u] > self.rank[root_v]:
                self.parent[root_v] = root_u
            elif self.rank[root_u] < self.rank[root_v]:
                self.parent[root_u] = root_v
            else:
                self.parent[root_v] = root_u
                self.rank[root_u] += 1

# Esempio di utilizzo
uf = UnionFind(5)
uf.union(0, 1)
uf.union(1, 2)
print(uf.find(0))  # Output: 0
print(uf.find(2))  # Output: 0
```

1. **Find**: Determina l'insieme a cui appartiene un elemento. Utilizza il path compression per ottimizzare la ricerca.
2. **Union**: Unisce due insiemi. Utilizza l'unione per rango per mantenere l'albero bilanciato.

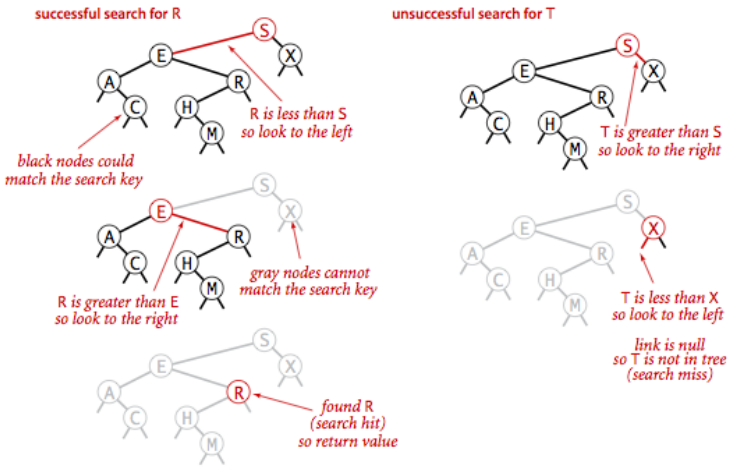
Complessità

- **Find**: O(α(n)) - Dove α(n) è la funzione inversa di Ackermann, estremamente lenta.
- **Union**: O(α(n)) - Dove α(n) è la funzione inversa di Ackermann, estremamente lenta.
- **Spaziale**: O(n) - La memoria necessaria per memorizzare i genitori e i ranghi.

La struttura Union-Find è estremamente efficiente e viene utilizzata in molti algoritmi di grafi e problemi di connettività

Binary Search Tree, BST

Un **Albero di Ricerca Binario** (Binary Search Tree, BST) è una struttura dati che permette di effettuare operazioni di ricerca, inserimento e cancellazione di elementi in modo efficiente. In un BST, per ogni nodo, i valori nel sottoalbero sinistro sono minori del valore del nodo, mentre i valori nel sottoalbero destro sono maggiori.



Successful (left) and unsuccessful (right) search in a BST

Operazioni BST

Ricerca

Migliore: O(log n) – quando possiamo dimezzare l'intervallo a ogni passo. Peggiore: O(n) – se l'albero è degenerato (es. una lista), visitiamo ogni nodo.

Inserimento

Migliore: O(log n) – stesso motivo della ricerca, se l'albero è bilanciato. Peggiore: O(n) – se tutti i nodi vanno solo a sinistra o destra.

Cancellazione

Migliore: O(log n) – vale se cancelliamo un nodo facilmente accessibile in un albero bilanciato. Peggiore: O(n) – se serve percorrere tutto l'albero per trovare e ristrutturare.

```
def insert(self, val):
    def _insert(node, val):
        if not node:
            return Node(val)
        if val < node.val:
            node.left = _insert(node.left, val)
        else:
            node.right = _insert(node.right, val)
        return node
    self.root = _insert(self.root, val)

def search(self, val):
    def _search(node, val):
        if not node or node.val == val:
            return node
        if val < node.val:
            return _search(node.left, val)
        return _search(node.right, val)
    return _search(self.root, val)
```

Complessità spaziale: O(n) – serve spazio per tutti i nodi.

BST - Albero AVL

Un **Albero AVL** è un tipo di BST bilanciato in cui la differenza di altezza tra i sottoalberi sinistro e destro di ogni nodo è al massimo 1. Questo bilanciamento garantisce che le operazioni di ricerca, inserimento e cancellazione abbiano una complessità temporale di O(log n). L'altezza Bilanciata si ha quando per ogni nodo di un albero AVL, l'altezza dei suoi due sottoalberi differisce al massimo di uno.

Rotazioni: sono operazioni fondamentali utilizzate per mantenere l'equilibrio degli alberi AVL durante le operazioni di inserimento e cancellazione.

Operazioni AVL

Rotazione Semplice a Sinistra

Rotazione Semplice a Destra

Rotazione Doppia Sinistra-Destra

Rotazione Doppia Destra-Sinistra

Inserimento (Insertion): L'inserimento in un albero AVL è simile a quello in un albero binario di ricerca, con l'aggiunta della necessità di riequilibrare l'albero se diventa sbilanciato.

Cancellazione (Deletion): La cancellazione in un albero AVL richiede un riequilibrio simile a quello dell'inserimento. Dopo la rimozione di un nodo, l'albero potrebbe diventare sbilanciato e richiedere delle rotazioni per mantenere l'equilibrio.

```python
class Nodo:
    def __init__(self, valore):
        self.valore = valore
        self.altezza = 0
        self.sinistro = None
        self.destro = None

def rotazioneSinistra(y):
    # Passo 1: Identificare i nodi coinvolti nella rotazione
    x = y.destro
    T2 = x.sinistro

    # Passo 2: Eseguire la rotazione
    x.sinistro = y
    y.destro = T2

    # Passo 3: Aggiornare le altezze
    aggiornaAltezza(y)
    aggiornaAltezza(x)

    # Ritorna il nuovo nodo radice
    return x

def aggiornaAltezza(nodo):
    if nodo:
        nodo.altezza = 1 + max(altezza(nodo.sinistro), altezza(nodo.destro))

def altezza(nodo):
    if nodo is None:
        return -1
    return nodo.altezza

def fattoreBilanciamento(nodo):
    if nodo is None:
        return 0
    return altezza(nodo.sinistro) - altezza(nodo.destro)
```

BST - Albero Rosso-Nero

Un **Albero Rosso-Nero** è un altro tipo di BST bilanciato che garantisce che il percorso più lungo dalla radice a una foglia non sia più del doppio del percorso più corto. Gli alberi rosso-neri utilizzano nodi colorati (rosso o nero) per mantenere il bilanciamento.

Complessità

BST (Binary Search Tree):

Ricerca, Inserimento, Cancellazione: **O(h)**, dove h è l'altezza dell'albero.

Se l'albero è bilanciato, l'altezza h è circa **O(log n)** e quindi le operazioni sono efficienti.

Nel caso peggiore, se l'albero è **completamente sbilanciato** (ad esempio, una lista collegata), l'altezza può diventare **O(n)**, e quindi anche le operazioni diventeranno **O(n)**.

Albero AVL:

Ricerca, Inserimento, Cancellazione: **O(log n)** grazie al bilanciamento.

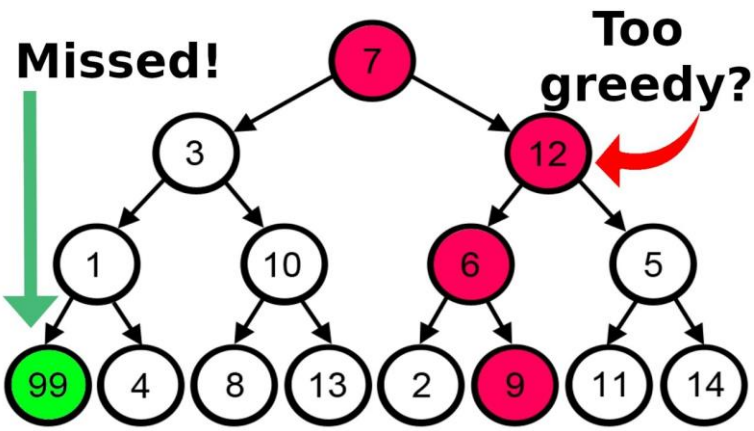
Un albero **AVL** è un albero binario di ricerca **auto-bilanciante**. La differenza di altezza tra il sottoalbero sinistro e destro di ogni nodo è al massimo 1, garantendo che l'albero sia sempre **bilanciato**. Pertanto, l'altezza è sempre **O(log n)**, e tutte le operazioni (ricerca, inserimento e cancellazione) hanno una complessità **O(log n)**.

Albero Rosso-Nero (Red-Black Tree):

Ricerca, Inserimento, Cancellazione: **O(log n)** grazie al bilanciamento.

Un albero **Rosso-Nero** è anche un albero binario di ricerca bilanciato, ma con alcune proprietà più rigide rispetto all'AVL. Questo bilanciamento garantisce che l'albero non diventi mai troppo sbilanciato, e l'altezza dell'albero è sempre **O(log n)**. Quindi le operazioni di ricerca, inserimento e cancellazione si eseguono in tempo **O(log n)**.

Greedy



L'algoritmo **Greedy** è una strategia che prende decisioni in modo **locale ottimale** in ogni passo, senza considerare gli effetti futuri delle scelte. In altre parole, l'algoritmo sceglie la migliore opzione possibile al momento, con l'aspettativa che, facendo scelte ottimali in ogni fase, si arriverà comunque alla soluzione ottimale complessiva.

Una delle principali caratteristiche di un algoritmo greedy è che ogni **decisione è irrevocabile**. Una volta fatta una scelta, non si torna indietro. Questo può sembrare controintuitivo, ma in certi problemi, come la divisione di risorse o la pianificazione di attività, questa strategia è efficace e conduce a buone soluzioni in tempi rapidi.

L'approccio greedy è **molto efficiente** in termini di tempo e risorse, poiché evita di esplorare tutte le possibili soluzioni. Tuttavia, la sua forza è anche la sua debolezza: non sempre porta alla soluzione ottimale globale, perché non considera l'intero scenario, ma si concentra su decisioni locali.

In definitiva, l'algoritmo greedy è utile in problemi dove le scelte locali ottimali si allineano con una soluzione globale ottimale, o dove una soluzione approssimativa è accettabile.

Greedy - Problema del Resto della Moneta (Coin Change Problem)

Il problema del resto della moneta consiste nel determinare il **numero minimo di monete** necessario per ottenere una somma S data, utilizzando una serie di monete con valori specifici. Questo problema può essere risolto utilizzando la programmazione dinamica, che consente di ottimizzare la ricerca della soluzione.

```python
def coin_change(coins, amount):
    coins.sort(reverse=True)  # Ordinamento delle monete in ordine decrescente
    result = []  # Lista per memorizzare le monete scelte
    for coin in coins:
        while amount >= coin:  # Finché la somma rimanente è maggiore o uguale alla moneta
            amount -= coin  # Sottrai la moneta dal resto da ottenere
            result.append(coin)  # Aggiungi la moneta alla lista del risultato
        if amount == 0:  # Se non resta denaro da ottenere, fermati
            break
    return result  # Restituisce la lista delle monete

# Esempio di utilizzo
coins = [1, 5, 10, 25]
amount = 63
print(coin_change(coins, amount))  # Output: [25, 25, 10, 1, 1, 1]
```

Greedy - Problema dello Zaino (Knapsack Problem)

Nel problema dello zaino, dato un insieme di oggetti con un peso e un valore associato, e una capacità massima dello zaino, l'obiettivo è determinare il massimo valore che può essere trasportato dallo zaino, scegliendo gli oggetti in modo tale da non superare la capacità.

```python
def knapsack(weights, values, capacity):
    items = sorted(zip(weights, values), key=lambda item: item[1] / item[0], reverse=True)
    total_value = 0
    for weight, value in items:
        if capacity >= weight:
            capacity -= weight
            total_value += value
        else:
            total_value += value * (capacity / weight)
            break
    return total_value
```

In questa versione del problema, usiamo un approccio **Greedy**. L'idea è di scegliere gli oggetti con il **rapporto valore/peso** più alto, cercando di massimizzare il valore totale per ogni unità di peso.

Greedy - Problema della Selezione delle Attività (Activity Selection Problem)

Nel problema della selezione delle attività, dato un insieme di attività con i relativi tempi di inizio e fine, l'obiettivo è selezionare il massimo numero di attività che non si sovrappongono. La strategia ottimale è quella di scegliere sempre l'attività che termina prima, così da lasciare spazio per quante più attività possibile.

```python
def activity_selection(start, end):
    activities = sorted(zip(start, end), key=lambda x: x[1])
    selected = []
    last_end = -1
    for s, e in activities:
        if s >= last_end:
            selected.append((s, e))
            last_end = e
    return selected
```

Minimum Spanning Tree, MST

Un Minimo Albero Ricoprente di un grafo non orientato connesso è un sottoinsieme degli archi che collega tutti i vertici del grafo senza cicli e con il peso totale minimo. Esistono due Algoritmi Greedy comunemente utilizzati per trovare il MST di un grafo: l'algoritmo di Kruskal e l'algoritmo di Prim.

Algoritmo di Kruskal per il Problema dell'Albero di Copertura Minima

L'algoritmo di Kruskal è un algoritmo utilizzato per trovare l'albero di copertura minima (MST) di un grafo con pesi. Utilizza la struttura dati **Union-Find** (Disjoint Set) per determinare se due vertici appartengono alla stessa componente connessa e per unire componenti connesse senza formare cicli

```
class DisjointSet:
    def __init__(self, vertices):
        self.parent = {}
        for v in vertices:
            self.parent[v] = v

    # 2) Funzione find
    def find(self, item):
        if self.parent[item] == item:
            return item
        else:
            self.parent[item] = self.find(self.parent[item])
            return self.parent[item]

    # 3) Funzione union
    def union(self, set1, set2):
        root1 = self.find(set1)
        root2 = self.find(set2)
        if root1 != root2:
            self.parent[root2] = root1

# 4) Algoritmo di Kruskal
def kruskal(graph):
    vertices = list(graph['vertices'])
    edges = sorted(graph['edges'], key=lambda edge: edge[2])  # Ordina gli spigoli in base al pe
    disjoint_set = DisjointSet(vertices)
    mst = []  # Albero di copertura minimo (MST)

    for edge in edges:
        u, v, weight = edge
        if disjoint_set.find(u) != disjoint_set.find(v):  # Se non sono nello stesso set
            disjoint_set.union(u, v)
            mst.append(edge)  # Aggiungi l'arco all'MST
        if len(mst) == len(vertices) - 1:  # Quando l'MST ha tutti i vertici
            break
    return mst
```

Algoritmo di Prim per l'Albero di Copertura Minima (MST)

L'algoritmo di Prim è un altro algoritmo che trova l'albero di copertura minimo (MST) di un grafo. A differenza di Kruskal, Prim cresce un albero partendo da un vertice e aggiungendo iterativamente l'arco di peso minimo che collega un vertice non ancora visitato.

```python
def prim(graph, start_vertex):
    mst = []  # Lista per memorizzare l'albero di copertura minimo (MST)
    visited = set()  # Insieme per tenere traccia dei vertici visitati
    min_heap = [(0, start_vertex, None)]  # Coda di priorità: (peso, vertice, arco)

    # 2) Ciclo per costruire l'MST
    while min_heap:
        weight, u, edge = heapq.heappop(min_heap)  # Estrai il vertice con il peso minimo
        if u not in visited:
            visited.add(u)  # Aggiungi il vertice all'insieme dei visitati
            if edge:  # Se l'arco è valido, aggiungilo all'MST
                mst.append(edge)
            for v, w in graph[u]:  # Esamina tutti i vicini del vertice u
                if v not in visited:
                    heapq.heappush(min_heap, (w, v, (u, v)))  # Aggiungi l'arco alla coda di prior
    return mst
```

L'algoritmo utilizza una struttura dati **heap** (coda di priorità) per scegliere l'arco di peso minimo in modo efficiente.

Reti di Flusso

Il problema delle reti di flusso riguarda il trasporto di una quantità di flusso attraverso una rete di nodi con capacità limitate sugli archi. L'obiettivo è determinare il flusso massimo che può essere trasportato da una sorgente a una destinazione nel rispetto delle capacità degli archi.

Componenti principali:

Nodi:

Sorgente (source): Il nodo da cui parte il flusso.

Destinazione (sink): Il nodo in cui il flusso deve arrivare.

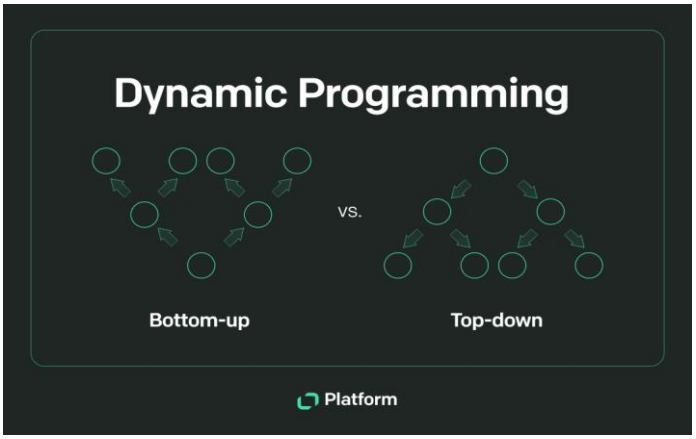
Nodi intermedi: I nodi che sono attraversati dal flusso.

Archi (Edge): Ogni arco ha una **capacità** che rappresenta la quantità massima di flusso che può passare attraverso di esso.

Flusso (Flow): La quantità di flusso che passa attraverso un arco, limitata dalla capacità dell'arco.

Programmazione dinamica

La **programmazione dinamica** è una tecnica di progettazione algoritmica utilizzata per risolvere problemi ottimizzabili attraverso la suddivisione in sottoproblemi sovrapposti.



L'idea centrale consiste nel calcolare ogni sottoproblema una sola volta e memorizzarne il risultato, evitando così di ricalcolarlo inutilmente. Questo approccio si applica quando un problema soddisfa due proprietà fondamentali: **ottimalità dei sottoproblemi** (una soluzione ottima del problema complessivo può essere costruita da soluzioni ottime dei sottoproblemi) e **sovrapposizione dei sottoproblemi** (gli stessi sottoproblemi si ripresentano più volte durante la risoluzione).

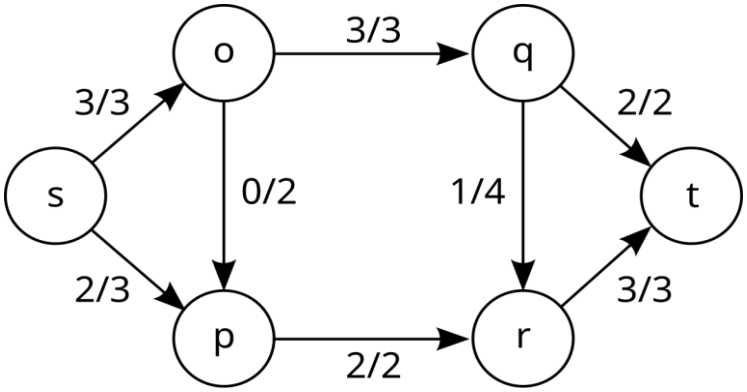
Esistono due modalità principali di implementazione: il **metodo top-down** (con memoization), che parte dal problema principale e risolve ricorsivamente i sottoproblemi salvando i risultati intermedi in una struttura come un dizionario o un array; e il **metodo bottom-up** (con tabulazione), che risolve iterativamente i sottoproblemi a partire dai casi base, riempiendo una tabella fino a ottenere la soluzione finale.

La programmazione dinamica è particolarmente efficace in problemi come il calcolo della serie di Fibonacci ottimizzata, il problema dello zaino (knapsack), il taglio delle aste, la pianificazione ottimale, e la ricerca di sottosequenze comuni (come nel longest common subsequence, LCS). Tuttavia, richiede una buona comprensione della struttura del problema per essere applicata correttamente, e può richiedere spazio aggiuntivo significativo per memorizzare i risultati parziali, soprattutto in implementazioni tabulari.

In generale, la programmazione dinamica rappresenta un potente strumento per ottenere soluzioni efficienti a problemi combinatori o decisionali complessi, laddove un approccio ingenuo avrebbe una complessità esponenziale.

EXTRA

Problema del flusso massimo



Il problema del flusso massimo riguarda il determinare la quantità massima di flusso che può essere trasportato da un nodo sorgente a un nodo destinazione in una rete di flusso, rispettando le capacità degli archi. Ogni arco ha una capacità che limita la quantità di flusso che può passarvi. Il problema è risolto utilizzando algoritmi che cercano di trovare il percorso più efficiente per trasportare il flusso, come l'algoritmo di Ford-Fulkerson o l'algoritmo di Edmonds-Karp.

Componenti principali:

1. **Nodi:**
    a. **Sorgente** (source): Il nodo da cui il flusso inizia.
    b. **Destinazione** (sink): Il nodo dove il flusso deve arrivare.
    c. **Nodi intermedi**: Altri nodi attraverso cui il flusso passa.
2. **Archi (Edge):**
    a. Ogni arco ha una **capacità** che rappresenta la quantità massima di flusso che può essere trasportata attraverso l'arco.
3. **Flusso (Flow):**
    a. La quantità di flusso che viene trasportata lungo un arco. Il flusso deve rispettare la capacità dell'arco.

Il problema è trovare il **flusso massimo** che può essere inviato dalla sorgente alla destinazione, data una rete di flusso con capacità sugli archi.

Metodo delle Reti Residue

Il metodo delle **reti residue** è un approccio fondamentale per risolvere il problema del **flusso massimo** nelle reti. In pratica, si tratta di un modo per **rappresentare le capacità rimanenti** di ciascun arco in una rete di flusso, aggiornando il flusso ogni volta che un percorso aumenta il flusso dalla sorgente alla destinazione.

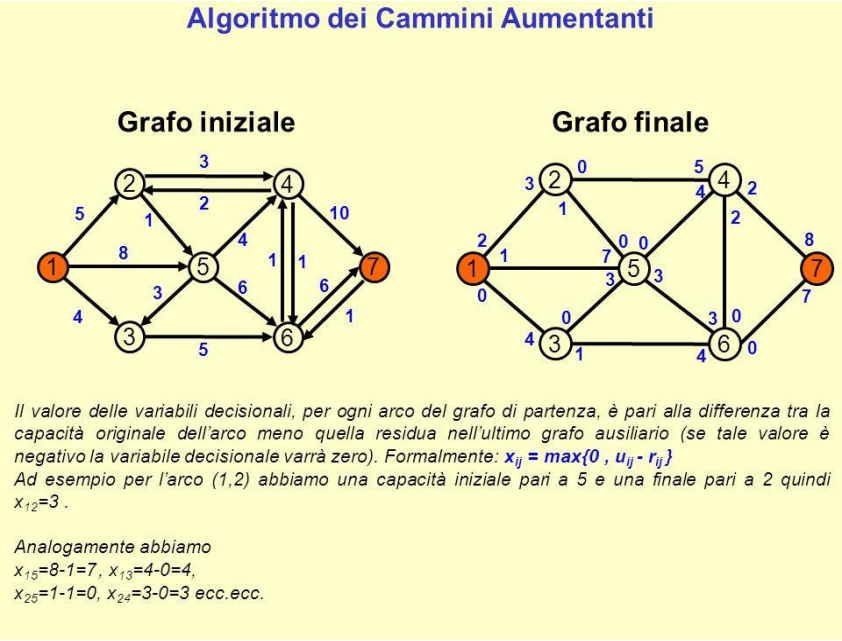
In un grafo di flusso, quando il flusso aumenta lungo un percorso, le capacità degli archi lungo il percorso vengono ridotte. Al contrario, la **rete residua** include sia le capacità residue per il flusso diretto (capacità rimanenti sugli archi) che per il flusso inverso (un arco inverso con capacità che rappresenta il flusso che può essere "ritirato" o ridotto).

1. Ogni arco ha una **capacità residua** che rappresenta quanto flusso può ancora passare attraverso quell'arco.
2. Ogni arco inverso ha una **capacità residua negativa** che rappresenta quanto flusso può essere "ritirato" dal nodo di destinazione verso la sorgente.

Il **flusso massimo** viene calcolato iterando sul grafo e aggiornando queste capacità residue fino a quando non ci sono più percorsi validi che possano trasportare flusso.

Metodo dei Cammini Aumentanti

Il **Metodo dei Cammini Aumentanti** è un approccio utilizzato per risolvere il problema del **flusso massimo** in una rete. Questo metodo si basa sull'identificazione di percorsi attraverso la rete che possono accogliere ulteriore flusso, incrementando progressivamente la quantità totale di flusso dalla sorgente alla destinazione.



1. **Inizializzazione del Flusso:** Si parte assegnando un flusso nullo a tutti gli archi della rete.
2. **Ricerca di un Cammino Aumentante:** Si individua un cammino dalla sorgente alla destinazione nel **grafo residuo**, dove la capacità residua di ogni arco è positiva. Questo cammino rappresenta un percorso attraverso il quale è possibile aggiungere flusso.
3. **Determinazione del Flusso Aumentante:** Si calcola la capacità residua minima lungo il cammino trovato; questa rappresenta la quantità massima di flusso che può essere aggiunta lungo quel percorso.
4. **Aggiornamento del Flusso e del Grafo Residuo:** Si incrementa il flusso lungo il cammino identificato e si aggiornano le capacità degli archi nel grafo residuo, riducendo le capacità lungo il cammino e aggiungendo archi inversi con le capacità corrispondenti.
5. **Ripetizione:** Si ripetono i passaggi precedenti finché non è più possibile trovare cammini aumentanti nel grafo residuo, indicando che è stato raggiunto il flusso massimo.

Rete Residua: Il concetto di rete residua è fondamentale in questo metodo, poiché consente di tracciare le capacità rimanenti e le possibili direzioni del flusso residuo, facilitando l'identificazione dei cammini aumentanti e l'aggiornamento delle capacità durante l'esecuzione dell'algoritmo

Algoritmo di Ford-Fulkerson

L'algoritmo di **Ford-Fulkerson** è una soluzione per il problema del flusso massimo in una rete. Si basa sul concetto di **cammini aumentanti** nella **rete residua**. Ogni volta che un cammino che può ospitare flusso viene trovato, l'algoritmo incrementa il flusso attraverso quel cammino, aggiornando le capacità degli archi nella rete. Il processo continua finché non esistono più cammini aumentanti che possano trasportare flusso.

```
Ford-Fulkerson(G, s, t):
  per ogni arco (u,v) in G:
    f[u,v] = 0  # Inizializza il flusso a zero su tutti gli archi

  mentre esiste un cammino p da s a t nella rete residua Gf:
    cf(p) = min{cf(u,v): (u,v) è in p}  # Trova il flusso massimo che può passare lungo p

    per ogni arco (u,v) in p:
      f[u,v] = f[u,v] + cf(p)  # Aggiorna il flusso lungo il cammino
      f[v,u] = f[v,u] - cf(p)  # Aggiungi flusso inverso

  return somma di f[s,v] per ogni v  # Somma i flussi dalla sorgente s a tutti gli altri vertici
```

Complessità:

Tempo:

Minima: O(E * V), nel caso in cui il flusso massimo è trovato in pochi passaggi.

Massima: O(E * V^2), nel caso peggiore, quando ogni iterazione richiede la ricerca di un cammino aumentante e il flusso massimo è grande.

Media: O(E * V^2), considerando che ogni iterazione potrebbe richiedere una ricerca completa del cammino.

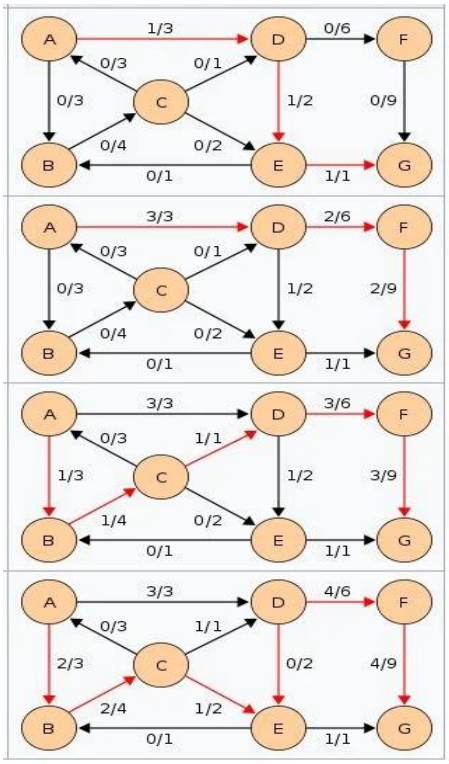
Spazio:

Minima: O(V + E), poiché dobbiamo mantenere un grafo e la lista dei percorsi.

Massima: O(V + E), per memorizzare la rete e i cammini trovati.

Media: O(V + E).

Algoritmo di Edmonds-Karp



L'algoritmo di **Edmonds-Karp** è una versione specifica dell'algoritmo di **Ford-Fulkerson** che utilizza una ricerca in **ampiezza (BFS)** per trovare il cammino aumentante. Questo approccio permette di garantire che l'algoritmo termini in un tempo polinomiale, con una complessità di $O(V * E^2)$, dove V è il numero di vertici e E è il numero di archi.

Tempo:

Minima: O(E * V), se ogni iterazione trova un cammino aumentante rapidamente.

Massima: $O(E * V^2)$, nel caso peggiore, poiché ogni iterazione comporta la ricerca di un cammino e l'aggiornamento del flusso.

Media: $O(E * V^2)$, in quanto l'algoritmo esplora i cammini in modo sistematico con BFS.

Spazio:

Minima e Massima: O(V + E), per memorizzare il grafo e il cammino.

```
Edmonds-Karp(G, s, t):
    inizializza f[u,v] = 0 per ogni arco (u,v) in G
    mentre esiste un cammino aumentante p da s a t nella rete residua Gf:
        cf(p) = min{cf(u,v): (u,v) è in p}
        per ogni arco (u,v) in p:
            f[u,v] += cf(p)
            f[v,u] -= cf(p)
    ritorna somma di f[s,v] per ogni v
```

Algoritmo di Dinic

L'**algoritmo di Dinic** è una versione avanzata dell'algoritmo di flusso massimo, che utilizza una combinazione di **livelli** e **flusso di blocco** per migliorare l'efficienza rispetto ad altri algoritmi di flusso massimo come **Ford-Fulkerson** o **Edmonds-Karp**. È più veloce nei casi di grafi densi, con una complessità di $O(V^2 * E)$ nel caso generale, ma in alcuni casi particolari (come grafi con larghezze costanti) può essere ancora più veloce.

L'algoritmo funziona con due fasi principali:

1. **Costruzione del livello:** Durante questa fase, si costruisce un grafo di livello dove i vertici sono etichettati in base alla distanza dalla sorgente.
2. **Flusso di blocco:** Viene trovato un flusso massimo utilizzando un algoritmo di flusso incrementale che esplora solo i cammini che rispettano i livelli stabiliti.

```
Dinic(G, s, t):
    costruisci la rete residua con capacità iniziali
    mentre esiste un cammino aumentante con livello:
        mentre esiste un flusso incrementale usando il cammino aumentante:
            aggiorna il flusso
    ritorna la somma del flusso
```

Tempo:

Minima: $O(E * \sqrt{V})$ (per grafi sparsi).

Massima: $O(V^2 * E)$ (per grafi densi).

Media: $O(V^2 * E)$, con una prestazione migliore rispetto a **Edmonds-Karp** nei grafi più densi.

Spazio: $O(V + E)$, per memorizzare la rete, il livello e i puntatori per ciascun nodo.

NP-Completezza

NP-completo è una classe di problemi decisionali che sono sia **in NP** che **NP-difficili**. Per un problema essere NP-completo, deve soddisfare due condizioni principali:

1. **In NP**: Un problema è in NP se una soluzione proposta per il problema può essere verificata in tempo polinomiale.
2. **NP-difficile**: Un problema è NP-difficile se ogni altro problema in NP può essere ridotto ad esso in tempo polinomiale.

In altre parole, se un algoritmo polinomiale esistesse per risolvere un problema NP-completo, allora esisterebbe un algoritmo polinomiale anche per **tutti** i problemi in NP, il che risolverebbe uno dei principali problemi non risolti nella teoria della complessità computazionale, ovvero **P ≠ NP**.

La **NP-completezza** ha un'importanza centrale nella teoria della complessità computazionale. Questi problemi sono considerati difficili da risolvere in modo efficiente (in tempo polinomiale), e attualmente non esiste una soluzione nota per risolverli in tempo polinomiale. Tuttavia, molti algoritmi approssimativi, euristici o di ricerca locale sono usati per trovare soluzioni "buone" in un tempo ragionevole.

In pratica, affrontare problemi NP-completi implica fare compromessi tra tempo di esecuzione e qualità della soluzione. Algoritmi di **approssimazione**, **branch and bound**, e **programmazione dinamica** sono tecniche utilizzate per trattare questi problemi quando una soluzione esatta non è fattibile.

Problema del soddisfacimento booleano (SAT): Determinare se esiste un'assegnazione delle variabili che soddisfa una formula booleana è il primo problema provato essere NP-completo (Teorema di Cook).

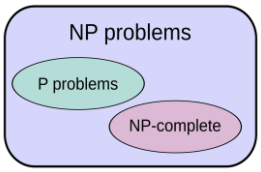
Problema del commesso viaggiatore (TSP): Trovare il cammino più corto che visita tutti i nodi esattamente una volta e ritorna al punto di partenza (versione decisionale: esiste un cammino di lunghezza ≤ k?).

Problema del clic (Clique Problem): Determinare se un grafo contiene un sotto grafo completo (clic) di dimensione almeno k.

Problema dello zaino (Knapsack Problem): Determinare se esiste una selezione di oggetti con peso totale ≤ W e valore totale ≥ V.

Problema della colorazione dei grafi (Graph Coloring): Determinare se è possibile colorare i vertici di un grafo con al più k colori tali che nessun arco connetta vertici dello stesso colore.

Classi di Complessità



In teoria della complessità computazionale, i problemi vengono classificati in classi di complessità a seconda delle risorse (tempo, spazio) necessarie per risolverli. Queste classi sono fondamentali per comprendere quanto "difficile" sia risolvere un determinato problema e come le soluzioni possono scalare con l'aumentare della dimensione del problema.

P (Tempo Polinomiale)

La classe P include tutti i problemi che possono essere risolti in tempo polinomiale rispetto alla dimensione dell'input. In altre parole, esistono algoritmi che risolvono questi problemi in tempo O(n^k), dove k è una costante e n è la dimensione dell'input.

Esempi: Ordinamento (ad esempio, quicksort, mergesort) - Ricerca binaria - Algoritmo di Dijkstra per il cammino minimo

Nota: I problemi in P sono considerati "facili" da risolvere, in quanto il tempo di esecuzione cresce in modo controllato all'aumentare della dimensione del problema.

NP (Non-deterministic Polynomial)

La classe NP include problemi per i quali una soluzione può essere verificata in tempo polinomiale. In altre parole, se ci viene fornita una soluzione, possiamo verificare rapidamente se è corretta, ma trovare la soluzione stessa potrebbe richiedere un tempo esponenziale. Un problema è in NP se è possibile verificare una soluzione in tempo polinomiale, ma non si sa se esiste un algoritmo che possa risolverlo in tempo polinomiale.

NP-Completo (NP-Complete)

Un problema è NP-completo se appartiene a NP e tutti gli altri problemi in NP possono essere ridotti a questo problema in tempo polinomiale. In altre parole, i problemi NP-completi sono i più difficili all'interno di NP: se si trovasse un algoritmo polinomiale per uno di essi, allora esisterebbe un algoritmo polinomiale per tutti i problemi NP.

Esempi: Problema del commesso viaggiatore (TSP) Problema della copertura del vertice I problemi NP-completi sono quelli per cui non si conosce una soluzione polinomiale, ma se ci viene fornita una soluzione, possiamo verificarla velocemente.

NP-Difficile (NP-Hard)

Un problema è NP-difficile se ogni problema in NP può essere ridotto a questo problema in tempo polinomiale. Non è necessario che il problema appartenga a NP (cioè che la sua soluzione possa essere verificata in tempo polinomiale), ma deve essere almeno altrettanto difficile da risolvere.

Esempi: Problema del cammino hamiltoniano (con o senza ciclo) - Problema della colorazione dei grafi

Un problema NP-difficile può essere in NP (se la soluzione è verificabile) o fuori NP (se la soluzione non è verificabile in tempo polinomiale).