15 - Paradigma generativo

Paradigma generativo: recap

Dal paradigma generativo scaturiscono tecniche di progetto di algoritmi che generano direttamente la soluzione, anziché selezionarla tra gli elementi già presenti nello spazio di ricerca.
In questo contesto, il concetto di spazio di ricerca viene gestito in modo specifico:

- Viene considerato **esclusivamente in fase di progetto** dell'algoritmo.
- Questa analisi preliminare è finalizzata a caratterizzare le soluzioni del problema e a definire una **strategia risolutiva diretta** per ogni istanza, evitando l'esplorazione durante l'esecuzione.

Appartengono a questo paradigma due tecniche fondamentali di progettazione algoritmica:

- La tecnica **greedy**.
- La tecnica **divide-et-impera**.

Tecnica Greedy

La tecnica Greedy viene usata soprattutto per i problemi di ottimizzazione, cioè quando devi trovare la soluzione migliore possibile (come il massimo guadagno o il minimo costo).
Il funzionamento è intuitivo: l'algoritmo non trova la soluzione completa in un colpo solo, ma la **costruisce passo dopo passo**, aggiungendo un pezzo alla volta.

Il funzionamento si basa rigorosamente su due principi fondamentali:

- **Scelta Ottima Locale:** In ogni stadio i, per definire la componente i-esima della soluzione, si seleziona il valore che appare "migliore" tra quelli ammissibili, valutandolo rispetto a un criterio di preferenza fissato a priori.
- **Irrevocabilità:** Una volta compiuta la scelta per la componente i-esima, si procede a determinare le componenti successive. Non è previsto alcun meccanismo per rivedere o modificare le decisioni già prese; l'algoritmo non torna mai sui suoi passi (*no backtracking*).

Algoritmo greedy: struttura procedurale

L'implementazione di un algoritmo greedy presuppone che si acquisisca la rappresentazione di una specifica istanza del problema e che si disponga di un metodo per organizzare la costruzione di un elemento dello spazio di ricerca in stadi successivi.

L'algoritmo viene strutturato in questo modo:

- Si imposta l'indice dello stadio corrente i a 1 e si inizializza la struttura z che conterrà la soluzione in fase di costruzione.
- Si determina l'insieme A costituito dai valori ammissibili per la componente i-esima di z. Se tale struttura A non è vuoto, si procede scegliendo al suo interno l'elemento migliore rispetto al criterio di preferenza fissato.
- Si verifica se lo stadio i-esimo corrente corrisponde all'ultimo stadio del processo. Se la condizione è verificata, l'algoritmo termina e restituisce $o(z)$ come risultato finale.
- Qualora non si sia raggiunto l'ultimo stadio, si incrementa l'indice i di 1 e si ritorna al passo di selezione (punto 2) per determinare la componente successiva.

Esempio Applicativo: Il Problema dello Zaino (Knapsack Problem)

Il problema dello zaino rappresenta un esempio di ottimizzazione combinatoria. Lo scenario prevede la disponibilità di un budjet limitato B e la necessità di selezionare un sottoinsieme di n possibili investimenti (o oggetti) al fine di massimizzare la rendita totale.

Ogni singolo investimento i da due parametri fondamentali:

- un profitto P_i
- Un costo C_i

Il problema viene formalizzato ricevendo in input $2n + 1$ interi positivi (i vettori dei profitti P, dei costi C e il budget B). L'obbiettivo è determinare n valori per le variabili decisionali $X_1, X_2, \ldots, X_n$ tali che:

1. $X_i \in \{0, 1\}$ per ogni $1 \leq i \leq n$. Il valore 1 indica l'oggetto selezionato, 0 altrimenti.
2. Si vuole rendere massima la somma dei profitti degli oggetti selezionati:

$$\sum_{i=1}^{n} P_i X_i \ (da\ massimizzare)$$

3. La somma massima sei costi degli oggetti selezionati non deve superare il budget disponibile B:

$$\sum_{i=1}^{n} C_i X_i \leq B$$

Si consideri un'istanza specifica del problema caratterizzato da un budget totale $B = 5n$ e da quattro elementi disponibili. Per applicare efficacemente la logica greedy, è necessario valutare ogni oggetto attraverso un criterio che sintetizzi la convenienze dell'investimento:

i **(Oggetto)**	P_i **(Profitto)**	C_i **(Costo)**	P_i/C_i **(Rapporto)**
1	6	2	3
2	4	1	4

i (Oggetto)	P_i (Profitto)	C_i (Costo)	P_i/C_i (Rapporto)
3	7	2	3.5
4	10	5	2

Il criterio di base adottato per ordinare le variabili decisionali consiste nel valutare simultaneamente sia il costo che il profitto, utilizzando la formula P_i/C_i. Questo rapporto esprime, di fatto, la "densità" di valore dell'oggetto.

Una volta effettuato l'ordinamento degli oggetti in ordine decrescente rispetto a tale rapporto, si procede alla selezione sequenziale degli stessi, includendoli nella soluzione fintanto che risultano compatibili con i vincoli di budget residuo.

Applicando il metodo all'istanza $B = 5$, gli oggetti vengono scelti nei seguente ordine di priorità:

- Oggetto 2 (Rapporto massimo pari a 4).
- Oggetto 3 (Rapporto pari a 3.5).
- Oggetto 1 (Rapporto pari a 3).
  Al termine del processo, la somma dei costi degli oggetti selezionati (1+2+2) risulta pari a 5, esaurendo esattamente il budget disponibile.

Metodo

La formalizzazione dell'algoritmo prevede la definizione di una funzione specifica, denominata `zainogreedy`, progettata per accettare in ingresso i parametri che descrivono l'istanza del problema.

La funzione richiede i seguenti argomenti:

- Il numero n di oggetti disponibili, i quali vengono denotati e indicizzati tramite numeri interi compresi nell'intervento da 0 a $n - 1$.
- Il vettore P, contiene i valori dei profitti associati a ciascun oggetto.
- Il vettore C, contente i valori dei costi associati a ciascun oggetto.
- Il valore del budget B, che rappresenta il vincolo di capacità massima dello zaino.

Il risultato dell'elaborazione viene restituito tramite un vettore soluzione X.
La semantica di tale vettore è binaria:

- La i-esima componente vale 1 se l'oggetto i è stato selezionato per essere incluso nella soluzione.
- La componente vale 0 in caso contrario.

Per decidere l'ordine di scelta, la funzione usa una lista di appoggio (chiamata V) dove mette in fila gli oggetti.

Li ordina dal migliore al peggiore, basandosi su quanto conviene ogni oggetto (cioè sul rapporto Profitto/Costo P_1/C_1) .

Attenzione: quando spostiamo gli oggetti per ordinarli, rischiamo di non capire più chi è chi. Per non perdere il filo, ogni elemento di questa lista salva **due informazioni** :

1. Il **valore** del rapporto P_1/C_1 associato all'oggetto.
2. **L'indice** originale dell'oggetto, memorizzato in un apposito campo, che ne permette l'identificazione univoca indipendentemente dalla nuova posizione assunta dopo l'ordinamento.

Terminata la fase di ordinamento, l'algoritmo procede alla costruzione della soluzione iterando sul vettore V:

1. La funzione scandisce il vettore V elemento per elemento.
2. Per ogni passo i, vene analizzato l'oggetto corrispondente a $V(i)$.
3. La decisione di includere l'oggetto nella soluzione finale viene presa verificando la sua compatibilità con il vincolo del budget. Se l'inserimento dell'oggetto non viola il vincolo di capacità residua, esso viene aggiunto alla soluzione; in caso contrario viene scartato senza possibilità di riconsiderazione futura.

Tecnica greedy: concetti base

Un algoritmo greedy costruisce la soluzione attraverso una sequenza di decisioni. Il principio è in ogni piccolo passo viene intrapresa l'azione che al momento appare migliore (ottimo locale).

È fondamentale notare che questa strategia è di natura **euristica**: di per sè, non garantisce sempre il raggiungimento della soluzione ottima globale.

Per determinare se un algoritmo greedy è effettivamente in grado di trovare la soluzione ottima per uno specifico problema di ottimizzazione, è necessario verificare la sussistenza di due proprietà teoriche strettamente correlate:

- La proprietà della scelta greedy.
- La sottostruttura ottima.

1 Proprietà della scelta greedy: Questa proprietà assicura che è possibile ottenere una soluzione ottima globale prendendo esclusivamente decisioni che costituiscono ottimo locali. Per convalidare questa proprietà, occorre dimostrare formalmente che una soluzione ottima del problema può essere modificata in modo tale da includere la prima scelta greedy. Questa inclusione deve permettere di ridurre il problema originale a un sottoproblema più piccolo ma della stessa tipologia.

2 Sottostruttura ottima: Un problema possiede queste proprietà se la soluzione ottimale globale è costruita combinando le soluzioni ottimali dei suoi sottoproblemi. In altri termini, se

si analizza la soluzione finale migliore, si riscontra che essa è formata, a sua volta, dalle scelte migliori possibili per ogni singola parte del problema.

Le due proprietà sono interdipendenti: per dimostrare che l'effettuazione di una scelta greedy riduce validamente il problema a un sottoproblema minore dello stesso tipo, è necessario provare che la soluzione ottima globale è composta dalle soluzioni ottime dei sottoproblemi generati.

Esempio: problema di selezione di attività

Questo problema riguarda l'assegnamento efficiente di una risorsa condivisa a un insieme di attività in competizione tra loro.

Sia $S = \{1, 2, \ldots, n\}$ un insieme costituito da n attività che necessitano di utilizzare una determinata risorsa. La natura della risorsa impone un vincolo di esclusività: essa non può essere utilizzata contemporaneamente da più attività.

Ogni generica attività k è definita temporaneamente da due parametri:

1. Tempo di inizio (attivazione): I_k
2. Tempo di fine (conclusione): F_k vale sempre la condizione $I_k \leq F_k$

Definizione di compatibilità: Due attività distinte k e j sono definite compatibili se i rispettivi intervalli temporali di esecuzione $[I_k, F_k]$ e $[I_j, F_j]$ non si sovrappongono.

Il problema di individuare un sottoinsieme di S che contenga il massimo numero di attività mutuamente compatibili.

La risoluzione del problema attraverso l'approccio greedy prevede una fase preliminare di organizzazione dei dati seguita da una procedura di selezione iterativa.

Fase 1: Ordinamento (preprocessing): Le attività in ingresso vengono ordinate in modo crescente rispetto al loro tempo di fine. Si ottiene così una sequenza ordinata tale che:

$$F_1 \leq F_2 \leq F_3 \leq \cdots \leq F_n$$

Se si considerano le attività $1, \ldots, n$ relative a un'istanza, lo spazio di ricerca è costituito dall'insieme di tutti i possibili sottoinsieme di $\{1, \ldots, n\}$. Tuttavia, l'approccio generativo evita di esplorare questo spazio esaustivamente.

Fase 2: Selezione: Ad ogni passo, la scelta greedy consiste nel selezionare l'attività k che, tra quelle ancora disponibili, rilascia per prima la risorsa condivisa. Grazie all'ordinamento preliminare, ciò corrisponde a scegliere l'attività disponibile con il minor tempo di fine F_k.

Fase 3: Aggiornamento: Una volta selezionata l'attività k e aggiunta alla soluzione, si procede all'aggiornamento dell'insieme delle attività disponibili. Vengono eliminate dell'insieme tutte le attività incompatibili con k, ovvero quelle il cui intervallo di esecuzione si

sovrappone a quello di k (attività che richiedono la risorsa nel lasso di tempo compreso tra I_k e F_k). Si ripete quindi il processo con le attività rimanenti.

```
GREEDY (I e F: VETTORI; n: INTEGER)
Variabile A: INSIEME     //CONTIENE LE ATTIVITÀ

    CREAINSIEME (A)
    INSERISCI (1,A)
    j <- 1
    for k=2 to n do
        if Ik >= Fj then //controlla sovrapposizione
         INSERISCI (k,A)
        j <- k
```

Qui si assume che i vettori siano già ordinati secondo i tempi di fine.

Esempio: il problema del percorso più breve in un grafo

Un'altra applicazione fondamentale dell'approccio generatico riguarda i grafi.

Sia $G = (N, A)$ un grafo orientato con archi etichettati da valori interi positivi (pesi).
Dato un nodo $r \in N$, $\forall$ nodo $u \in N$, 'obbiettivo è trovare la lunghezza del percorso più breve per connettere r a un qualsiasi nodo u.

La strategia risolutiva per il problema del percorso più breve si basa sull'idea di calcolare la lunghezza dei cammini minimi dal nodo sorgente r verso tutti gli altri nodi, procedendo rigorosamente in **ordine crescente** di lunghezza.

Per implementare questo metodo sono necessarie due strutture fondamentali:

- Un insieme S: contiene i nodi per i quali, in un dato istante, è già stata calcolata in via definitiva la lunghezza del cammino minimo da r.
- Un vettore $dist$: avente una dimensione pari al numero totale dei nodi. Il valore $dist(i)$ indica la lunghezza del percorso minimo per andare dalla sorgente r al nodo i, calcolato utilizzando come tappe intermedie esclusivamente i nodi già presenti nell'insieme S.

Questa procedura si fonda sull'ipotesi che le distanze (i pesi degli archi) siano rappresentate da interi positivi.

La base dell'algoritmo risiede nella decisione di quale nuovo cammino generare ad ogni passo.
Si osserva che, se il prossimo cammino minimo da consolidare è quello verso un nodo u, allora tutti i nodi intermedi di tale percorso devono già appartenere all'insieme S.

Dimostrazione per assurdo: Se un nodo k appartenente al cammino verso u non fosse in S, esisterebbe un cammino da r a k di lunghezza inferiore a quella del cammino per u (dato che i pesi sono positivi).
Questo contraddirebbe l'ipotesi che il prossimo cammino minimo da generare sia proprio quello per u.

Sulla base dell'osservazione precedente, la selezione del nodo u e della sua distanza minima avviene analizzando il vettore $dist$:

1. **Selezione:** Si individua il nodo u non appartenente a S che possiede il valore minimo in $dist(i)$.
2. **Inserimento:** Il nodo u viene inserito nell'insieme S.
3. **Aggiornamento:** Si ricalcolano le distanze per i nodi z adiacenti a u che non sono ancora in S. Se per un nodo z, connesso a u tramite un arco con peso e, si verifica che la distanza passando per u è vantaggiosa, ovvero:

   $$dist(u) + e < dist(z)$$

   allora si aggiorna il valore:

   $$dist(z) \leftarrow dist(u) + e$$

   Al termine dell'esecuzione, viene generato un **albero di copertura** T, radicato in r, che include un cammino da r verso ogni altro nodo del grafo.
   Tale albero può essere rappresentato mediante un vettore dei padri.

Una soluzione ammissibile T è considerata ottima se e solo se soddisfa la **condizione di Bellman**:

- Per ogni arco (i, j) appartenente all'albero T:

  $$dist(i) + C_{ij} = dist(j)$$

- Per ogni arco (i, j) non appartenente all'albero T:

  $$dist(i) + C_{ij} \geq (j)$$

  Nota: Nel caso più generale in cui i pesi degli archi possano assumere valori negativi, questo algoritmo non è applicabile e si deve ricorrere all'algoritmo di Bellman-Ford.

```
CAMMINIMINIMI (G: GRAFO, r: NODO)
    CREAINSIEME(S)
    T(r) <- 0
    DIST(r) <- 0

    // Inizializzazione
    for k = 1 to n do
```

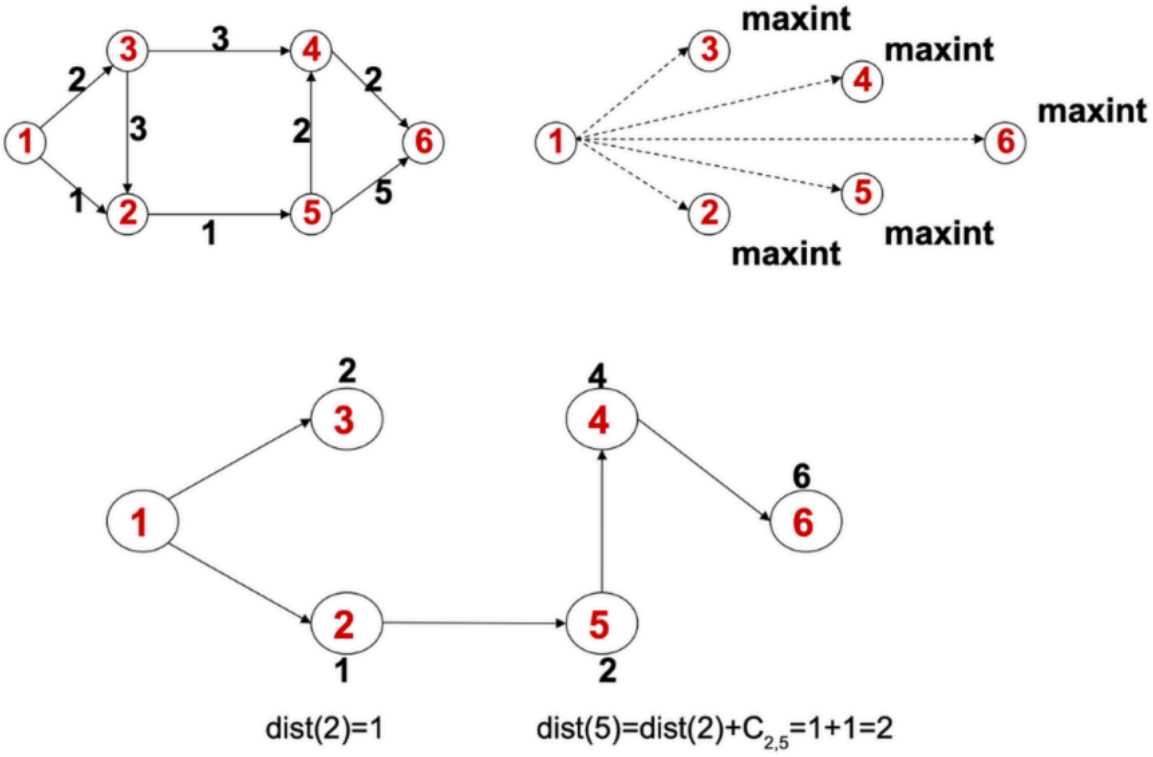
```
        if k != r then
            T(k) <- r
            DIST(k) <- MAXINT

    INSERISCI(r, S)

    // Ciclo principale
    while not INSIEMEVUOTO(S) do
        i <- LEGGI(S)
        CANCELLA(i, S)

        // A(i) è l'insieme di adiacenza del nodo i
        for j in A(i) do
            if DIST(i) + C[i,j] < DIST(j) then
                T(j) <- i
                DIST(j) <- DIST(i) + C[i,j]

                if not APPARTIENE(j, S) then
                    INSERISCI(j, S)
```



Calcolo di aggiornamento per mostrare la logica della disuguaglianza $DIST(i) + C_{ij} < DIST(j)$.

- Viene preso in esame l'arco che va dal nodo 2 al nodo 5.

- Essendo il nodo 2 già consolidato con una distanza $dist(2) = 1$, l'algoritmo verifica se è possibile migliorare la distanza verso il nodo 5.
- Il calcolo:

$$dist(5) = dist(2) + C_{2,5} = 1 + 1 = 2$$

- Poiché il nuovo valore (2) è minore della distanza precedente (che era `MAXINT` o un valore superiore), il vettore delle distanze viene aggiornato.

Algoritmo di Dijkstra

L'algoritmo generico visto precedentemente diventa l'algoritmo di Dijkstra (1959) se si specifica la natura della struttura dati S.
Affinchè l'algoritmo sia efficiente e corrisponda alla versione di Dijkstra, la struttura S deve essere implementata come una coda con priorità.

In questo contesto le operazioni generiche di lettura e cancellazione vengono sostituite dagli operatori algebrici delle code con priorità:

- **Min:** individua l'elemento con la priorità più alta (in questo caso, il valore numerico basso).
- **Cancellamin:** estrae e rimuove l'elemento minimo della struttura.

Ad ogni iterazione del ciclo principale, l'algoritmo estrae dalla coda S il nodo che ha la priorità minima. Poichè la priorità è la distanza, ciò significa che si seleziona sempre il nodo più vicino a r tra quelli non ancora consolidati, rispettando perfettamente il principio greedy.

Tecnica Divide et impera

Tecnica che deriva dall'idea di determinare la strategia risolutiva di un problema facendo ricorso al **principio di decomposiozione induttiva**.

Per poter applicare efficacemente questo approccio, è necessario che il problema soddisfi quattro requisiti strutturali:

- Deve esistere una relazione di ordinamento sulle istanze del problema, basata sulla dimensione dell'input.
- Si deve disporre di un metodo di risoluzione diretto per tutte le istanze che non superano una prefissata dimensione limite.
- Deve esistere un meccanismo per suddividere i dati di ingresso in diverse parti. Ciascuna parte deve avere una dimensione minore rispetto a quella originaria e deve rappresentare l'input di una nuova istanza dello stesso problema.
- Deve esistere un meccanismo per comporre le soluzioni ottenute dalle istanze individuate con la suddivisione, al fine di ottenere la soluzione finale per l'istanza originaria.

L'esecuzione di un algoritmo basato su questa tecnica segue un flusso logico ricorsivo standardizzato:

- **Verifica della Dimensione (Caso Base):** Si controlla se la dimensione dell'input è inferiore a un certo valore soglia k. In tal caso, non si procede oltre con la suddivisione, ma si utilizza un metodo diretto per calcolare e restituire il risultato immediato.
- **Suddivisione (Divide):** Se la dimensione supera la soglia, si divide l'input in più parti. Ciascuna di queste parti deve avere una dimensione rigorosamente inferiore rispetto all'input originario.
- **Ricorsione:** Si esegue ricorsivamente l'algoritmo su ciascuno degli input parziali individuati al passo precedente.
- **Composizione (Impera):** Si ricompongono i risultati parziali ottenuti dalle chiamate ricorsive per ottenere il risultato finale relativo all'istanza originaria.

Le applicazioni più note e diffuse di questa tecnica si riscontrano negli algoritmi di ordinamento, in particolare nel **Natural-Merge-Sort** e nel **Quicksort**.

Esempio: Il problema del minimo e massimo simultanei

In alcune specifiche applicazioni sorge la necessità di individuare il minimo e il massimo all'interno di un insieme di n elementi simultaneamente.

Un esempio tipico si riscontra nei programmi di grafica che devono rappresentare in scala un insieme di dati coordinate (x, y).
In questo scenario, è necessario determinare sia il minimo che il massimo di ogni coordinata per dimensionare correttamente il grafico.

Esistono due modi per affrontare il problema, con costi computazionali diversi:

- Cercando il minimo e il massimo in modo indipendente (scandendo l'array due volte o con due variabili separate senza interazione), sarà necessario un totale di $2(n-1)$ confronti.
- Mantenendo aggiornati gli elementi minimo e massimo via via incontrati e confrontando i due elementi della coppia in input tra loro prima di confrontarli con i globali, sono sufficienti $3(n/2)$ confronti. Questo approccio riduce il numero di operazioni richieste rispetto al metodo indipendente.

L'algoritmo del massimo e minimo simultaneo

L'applicazione della tecnica divide-et-impera per la ricerca simultanea degli estremi prevede una procedura ricorsiva che scompone il problema fino a renderlo banale.

L'algoritmo segue questi passi logici:

1. Se la dimensione del vettore (o della porzione considerata) non supera i 2 elementi, si calcolano direttamente il minimo e il massimo mediante un unico confronto.

2. In caso contrario (dimensione > 2):
    - Si divide il vettore in due sotto-vettori di dimensione uguale (o quasi uguale).
    - Si calcolano ricorsivamente:
        - Il minimo min_1 e il massimo max_1 del primo sottovettore.
        - Il minimo min_2 e il massimo max_2 del secondo sottovettore.
    - Si determina il minimo e il massimo del vettore complessivo confrontando i risultati parziali: min_1 con min_2 per il minimo globale, e max_1 con max_2 per il massimo globale.

La funzione `MAXMIN` accetta in ingresso il vettore S e gli indici x (inizio) e y (fine) che delimitano la porzione da analizzare.

```
VECT MAXMIN(S:VECT, x:INT, y:INT)

    // Caso Base: 1 o 2 elementi
    if (y - x) <= 1 then
        return { MAX(S[x], S[y]), MIN(S[x], S[y]) }

    // Passo Ricorsivo
    else
        // Calcolo del punto mediano per la divisione
        middle = floor((x + y) / 2)

        // Chiamate ricorsive sulle due metà
        (max1, min1) = MAXMIN(S, x, middle)
        (max2, min2) = MAXMIN(S, middle + 1, y)

        // Composizione dei risultati
        return ( MAX(max1, max2), MIN(min1, min2) )
```

Quicksort

La versione ricorsiva del Quicksort è un algoritmo di ordinamento basato sulla tecnica **divide-et-impera**.
L'idea centrale dell'algoritmo è la selezione di un elemento specifico all'interno del vettore, denominato **pivot**.
Una volta scelto il pivot, gli altri elementi vengono sistemati attorno ad esso seguendo una logica di partizionamento:

- Tutti gli elementi **più piccoli** del pivot vengono spostati alla sua sinistra.
- Tutti gli elementi **più grandi** del pivot vengono spostati alla sua destra.

Questa operazione genera due partizioni distinte.

Proprietà della Partizione e Ricorsione
Un aspetto fondamentale è che **non è necessario** che le due partizioni siano ordinate al loro interno in questa fase. L'ordinamento globale emerge dalla ricorsione: la procedura `quicksort` viene infatti richiamata ricorsivamente sulle due partizioni appena create. Il processo ricorsivo termina quando si raggiunge il **passo base**, che coincide con una partizione di lunghezza uno.

La funzione `QUICCKSORT` accetta il vettore A e gli indici p e q. Se la partizione contiene più di un elemento ($p < q$), si procede alla divisione e alle chiamate ricorsive.

```
QUICKSORT(A: VECT, p: INT, q: INT)
    if (p < q)
        pivot = PARTIZIONA(A, p, q);
        QUICKSORT(A, p, pivot-1);
        QUICKSORT(A, pivot, q);
```

La funzione `PARTIZIONA` riorganizza gli elementi e restituisce l'indice di separazione.
In questa versione, il pivot inizializzato come il primo elemento ($A[i]$).

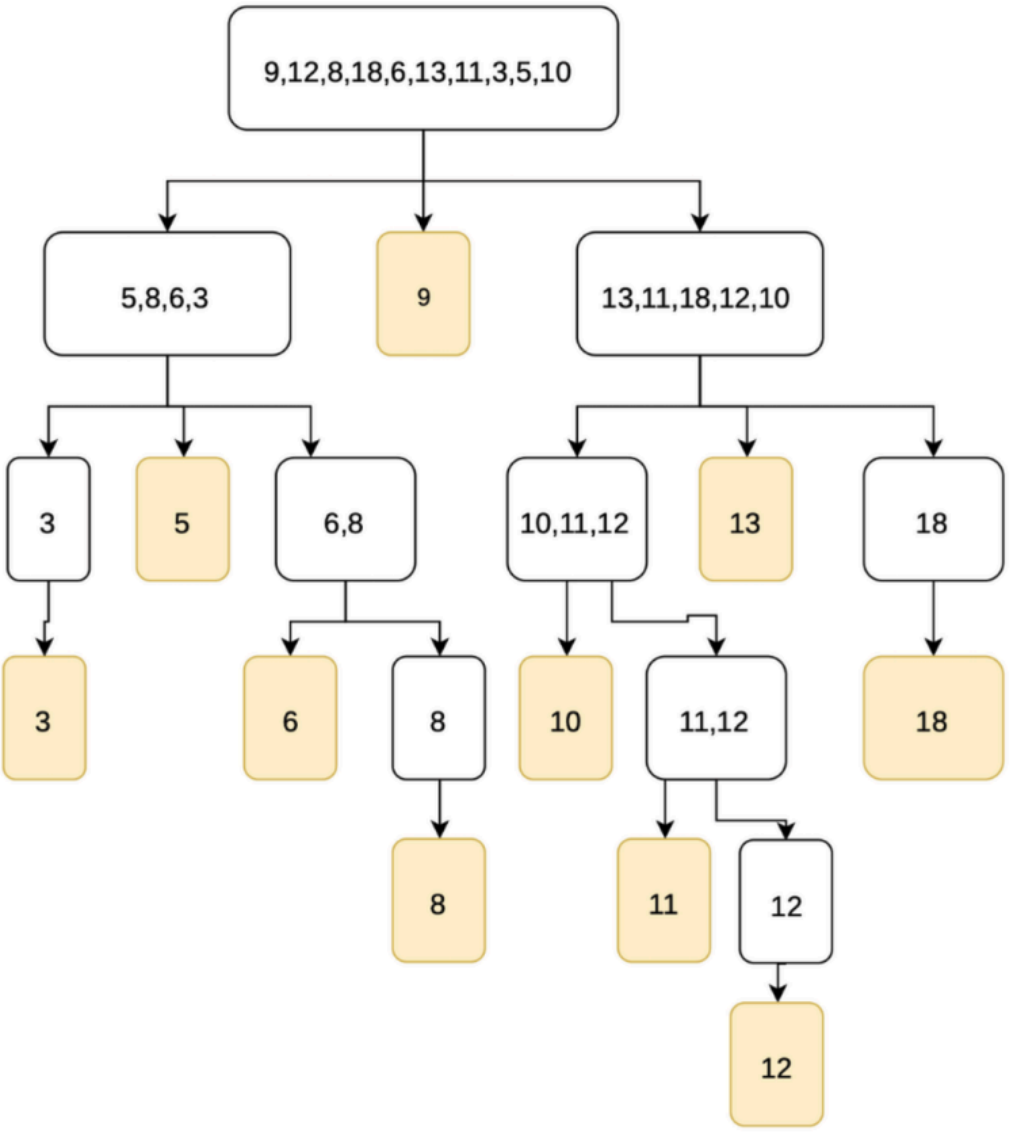
```
PARTIZIONA(A: VECT, p: INT, q: INT)
    i <- p
    j <- q
    pivot <- A[i]

    while i <= j
        // Scorre da destra verso sinistra cercando elementi <= pivot
        while A[j] > pivot
            j <- j - 1

        // Scorre da sinistra verso destra cercando elementi >= pivot
        while A[i] < pivot
            i <- i + 1

        // Se gli indici non si sono incrociati, scambia gli elementi
        if i <= j
            SCAMBIA(A[i], A[j])
            i <- i + 1
            j <- j - 1
```

```
    return i
```



[3, 5, 6, 8, 9, 10, 11, 12, 13, 18]

Analisi dell'esempio:
Prima partizione (radice):

- Viene selezionato l'elemento pivot iniziale. Il pivot è il 9.
- Il vettore viene partizionato rispetto al 9:
  - Nel ramo di sinistra finiscono tutti gli elementi minori: 5, 8, 6, 3.
  - nel ramo di destra finiscono gli elementi maggiori: 13, 11, 18, 12, 10.
- In questo momento, il numero 9 ha trovato la sua collocazione definitiva nell'ordinamento finale.

L'algoritmo viene riapplicato autonomamente sui due sotto-vettori generati:

- Dal gruppo 5, 8, 6, 3 (ramo sinistro) viene scelto 5 come pivot. Si separano il 3 (minore) e il gruppo 6, 8(maggiori).
- Dal gruppo 13, 11, 18, 12, 10 (ramo destro) viene scelto 13 come pivot. Si separano il gruppo 10, 11, 12 e il 18.

La ricorsione prosegue fino a isolare i singoli elementi che rappresentano le foglie dell'albero (3, 6, 8, 10, 12, 18).

Quicksort complesità

L'algoritmo Quicksort è caratterizzato da una complessità computazionale considerata ottima per il problema dell'ordianemnto, pari a $O(n \log n)$.
Tuttavia, le prestazioni dipendono dalla struttura dei dati di ingresso e dalla scelta del pivot.

Scenari:

- **Caso medio:** in condizioni normali, l'algoritmo mantiene la sua efficienza ottimale di $O(n \log n)$.
- **Caso pessimo:** esiste una possibilità che l'algoritmo degradi fino a una complessità quadratica di $O(n^2)$.

La situazione di inefficienza massima si verifica tipicamente quando gli elementi del vettore sono già ordinati e si adotta la strategia base di selezionare sempre il primo elemento come pivot.
In questo scenario, il partizionamento risulta estremamente sbilanciato (una partiziona vuota e l'altra con $n-1$ elementi), costringendo l'algoritmo a effettuare un numero eccessivo di chiamate ricorsive senza dividere efficacemente il problema.

L'algoritmo può essere reso più efficace modificando il criterio di selezione del pivot. La tecnica prevede di scegliere l'elemento pivot calcolando il mediano tra glie elementi specifici del vettore.

- Il primo elemento $A[1]$.
- L'ultimo elemento $A[n]$.
- L'elemento centrale $A\left[\frac{n}{2}\right]$.
  Una volta individuato il mediano, si effettua uno scambio con il primo elemento del vettore prima della chiamata alla procedura `PARTIZIONA`. Questo riduce la probabilità di incappare nel caso pessimo.

La tecnica divide-et-impera: conclusioni

Questa tecnica si configura come un metodo **generativo** che sfrutta la **decomposizione induttiva** per determinare la strategia di risoluzione del problema.

L'efficacia complessiva di questo approccio si manifesta attraverso due aspetti principali:

1. L'uso dell'induzione consente di progettare algoritmi che risultano spesso **semplici e intuitivi**. Solitamente, gli algoritmi derivanti da questa tecnica offrono prestazioni superiori rispetto ad altre metodologie.
2. L'efficienza degli algoritmi divide-et-impera non è casuale, ma dipende matematicamente da tre parametri specifici :
   - a: il numero di sottoproblemi generati dalla suddivisione.
   - b: la dimensione dei dati in ingresso ai singoli sottoproblemi (quanto sono più piccoli rispetto all'originale).
   - $F(n)$: il costo computazionale necessario per effettuare le operazioni di scomposizione del problema iniziale e di successiva composizione dei risultati parziali.