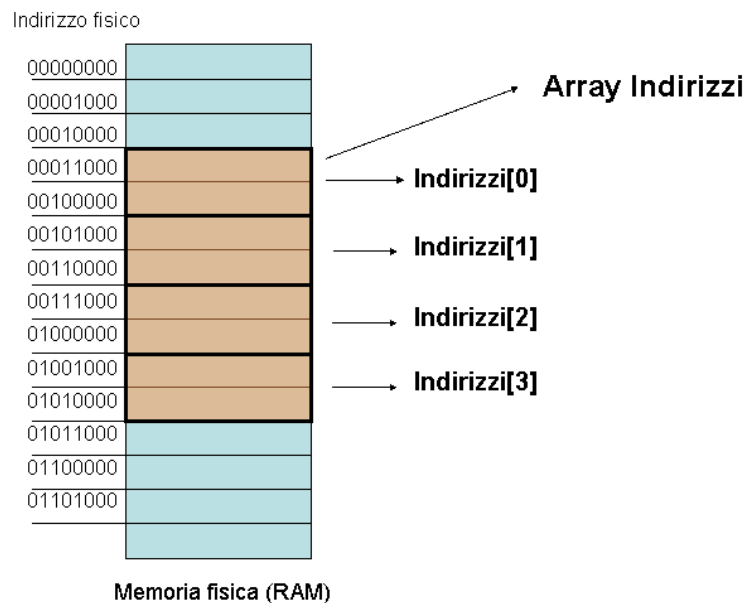


Esempi di algoritmi

Strutture dati contigue.

Normalmente gli algoritmi agiscono su insiemi di dati (strutture) complessi. La gestione dei dati può essere più o meno banale e dipende strettamente dalle proprietà della struttura che li contiene. Ad esempio, molte applicazioni possono memorizzare e gestire un numero elevato di informazioni che costituiscono delle banche dati. In particolare tali strutture possono contenere indirizzi dei clienti, codici di conti correnti bancari, numeri di telefono, ecc.

Normalmente informazioni tipo quelle descritte sopra vengono memorizzate in strutture dati dette *contigue*. Un esempio di struttura contigua è il classico *array* che esiste in tutti i linguaggi di programmazione. Un array viene memorizzato in memoria occupandone una zona contigua. La contiguità della struttura offre un duplice vantaggio: in primo luogo le operazioni di scrittura e lettura nell'array risultano più efficienti (minor tempo per accedere alle informazioni) e, in secondo luogo, l'array permette di accedere ai singoli elementi tramite indicizzazione facilitando lo scorrimento di tutta la struttura.



Nell'immagine viene rappresentata una struttura dati di tipo array allocata in memoria principale (RAM). La struttura dati chiamata *Indirizzi* contiene quattro elementi ognuno dei quali occupa 2byte (16 bit) ed è allocata in una memoria le cui righe sono costituite ciascuna da 1 byte (8bit). È possibile notare, quindi, che ogni elemento occupa due righe di memoria fisica. L'accesso ad un elemento della struttura avviene attraverso la notazione indicizzata *Indirizzi[i]* dove la variabile *i* rappresenta l'indice dell'array. Normalmente dato un array di dimensione *N* la variabile *i* è compresa nell'intervallo $[0, N-1]$. Questo perché, per convenzione, il primo elemento dell'array viene identificato con l'indice 0.

Ricerca di un valore in un elenco

Dato un elenco di valori (array) vogliamo valutare se al suo interno è presente o meno un determinato valore a noi noto che chiameremo *key*. La soluzione più immediata di questa classe di problemi consiste semplicemente nello scorrere l'intero elenco e confrontare ogni elemento con l'elemento *key*. L'algoritmo termina quando l'elemento viene trovato oppure una volta che tutti gli elementi dell'elenco sono stati valutati (ciò significa che siamo giunti alla fine dell'elenco senza trovare l'elemento cioè l'elemento non era presente).

Nella figura sottostante viene proposta un'implementazione in pseudocodice dell'algoritmo di ricerca.

```

1. Start
2. Dati[] ← input
3. Key ← input
4. N ← input
5. Trovato ← 0
6. i ← 0
7. while((i < N) && ! Trovato) do (
8.     if(Dati[i] == key) then (
9.         Trovato ← 1
10.    )
11.    i ← i+1
12. )end while
13. if(Trovato) then (
14.     Stampa "Elemento trovato"
15. ) else (
16.     Stampa "Elemento non trovato"
17. )
18. end

```

Inizializzazione delle variabili

Ciclo che scorre gli elementi dell'elenco

Stampa dei risultati

La prima parte del codice (righe 2-6) inizializza le variabili che verranno utilizzate dall'algoritmo. Gli elementi della variabile *Dati* di tipo array vengono prelevati dall'input primario attraverso la procedura *input*. Allo stesso modo vengono caricati i valori di *key* e di *N*. In questo caso la variabile *N* memorizza la dimensione dell'array *Dati* e servirà in seguito per scorrere tutti gli elementi. La variabile *i* verrà utilizzata come indice dell'array e pertanto viene inizialmente posta uguale a 0 (primo elemento dell'array). Infine, utilizziamo la variabile *Trovato* che servirà ad interrompere il ciclo nel momento in cui avremo riscontrato la presenza dell'elemento all'interno dell'array. Tale variabile viene inizialmente posta uguale a 0 (falso) indicando che l'elemento non è ancora stato trovato.

La parte vera e propria dell'algoritmo è rappresentata dal ciclo *while() do* che scorre tutto l'array e confronta gli elementi con l'elemento chiave. Notare che la condizione che controlla il ciclo è costituita da due diverse sotto-condizioni (*i < N*) e (*! Trovato*). Le due condizioni sono legate da un operatore booleano di tipo *AND* (&&) e pertanto perché il ciclo possa continuare è necessario che entrambe le condizioni siano vere. Questo fa sì che il ciclo continui fintanto che *i* non abbia raggiunto il valore di *N* e che la seconda condizione (*! Trovato*) sia uguale a 1 (vero).

Notare che nella seconda condizione la variabile *Trovato* viene negata dall'operatore (*!*) così che mentre *Trovato* vale 0 (non abbiamo ancora trovato l'elemento), (*! Trovato*) vale 1 (vero) ed il ciclo può continuare.

In altre parole il ciclo *while* smette di essere eseguito o nel caso in cui *i* abbia raggiunto il valore di *N* (siamo arrivati alla fine dell'array) oppure nel caso in cui l'elemento è stato trovato e la condizione (*! Trovato*) non risulta più vera.

Il corpo del ciclo *while* è costituito semplicemente dal confronto tra *i-esimo* elemento dell'array e la chiave (*Dati[i] == key*), e dall'aggiornamento delle variabili di controllo *i* e *Trovato*.

L'ultima parte dell'algoritmo è dedicata alla stampa del risultato ed effettua un controllo sulla variabile *Trovato* per decidere cosa stampare in output.

Tale algoritmo viene anche detto di **ricerca lineare** per il fatto che scorre linearmente tutti gli elementi dell'array.

Ricerca del valore minimo in un elenco di numeri

Il problema della ricerca del valore minimo (o massimo) in un array di numeri è un classico problema con cui ci si scontra facilmente nella programmazione degli elaboratori. In linea di principio, per risolvere questa classe di problemi, è necessario scorrere tutto l'array di numeri e confrontarne i valori a coppie di due. Da ogni confronto dobbiamo prendere il valore più piccolo (minimo) e memorizzarlo in una variabile temporanea per poterlo utilizzare al confronto successivo. Una volta raggiunta la fine dell'array nella variabile temporanea avremo memorizzato il valore minimo di tutto l'elenco.

Nella figura sottostante è riportato lo pseudocodice dell'algoritmo di ricerca del valore di minimo.

```

1. Start
2. Dati[] ← input
3. N ← input
4. i ← 0
5. min ← Dati[0]
6. while(i < N) do {
7.   If(Dati[i] < min) then {
8.     min ← Dati[i]
9.   }
10.  i ← i + 1
11. }end while
12. Stampa min
13. end

```

Inizializzazione delle variabili

Ciclo che scorre gli elementi dell'elenco

Stampa dei risultati

Nella fase di inizializzazione c'è da notare l'assegnamento alla variabile temporanea *min* del primo valore presente nell'array *Dati*. Tale assegnamento serve solo per dare un valore iniziale alla variabile *min* e poteva essere fatto con qualsiasi elemento dell'array (non necessariamente il primo).

Il ciclo *while* che costituisce l'algoritmo vero e proprio scorre l'intero array in quanto l'unica condizione di controllo è ($i < N$). All'interno del ciclo si valuta se l'*i-esimo* elemento dell'array risulta minore del valore memorizzato nella variabile temporanea *min* e in caso affermativo nella variabile *min* verrà memorizzato il nuovo valore inferiore a quello precedente. In questo modo, ogni volta che viene trovato un valore via via più piccolo questo viene salvato nella variabile temporanea *min* facendo sì che una volta raggiunta la fine dell'array questa contenga il valore più piccolo di tutti.

Da notare che, a differenza dell'algoritmo di ricerca di una chiave, questa volta è sempre necessario scorrere tutto il contenuto dell'array.

Problema dell'ordinamento dei dati

Spesso le strutture dati vengono mantenute “ordinate” secondo un ordine numerico crescente o decrescente o secondo un ordine alfabetico. Sono altresì possibili altri criteri di ordinamento che dipendono essenzialmente dalle particolarità dei dati che si trovano nelle strutture.

Strutture dati ordinate comportano alcuni vantaggi fra i quali il principale è senza dubbio la maggiore efficienza nel recuperare le informazioni. Si pensi ad esempio quanto sarebbe difficile trovare un numero telefonico nell'elenco di Pesaro se questo non fosse ordinato secondo l'ordine alfabetico. Data l'importanza dell'ordinamento delle strutture dati, sono stati descritti numerosi algoritmi in grado di manipolare strutture dati ordinandole secondo particolari criteri. Tali algoritmi vengono comunemente detti di **ordinamento**.

Algoritmi di ordinamento

Gli algoritmi di ordinamento si basano essenzialmente sul confronto e sullo spostamento dei valori contenuti nella struttura di partenza. I vari algoritmi si differenziano per le tecniche utilizzate e per la loro efficienza espressa in numero di confronti e spostamenti.

Possiamo in prima analisi suddividere gli algoritmi di ordinamento in:

- *Algoritmi di ordinamento sul posto*: classe di algoritmi che non si avvale di strutture dati ausiliarie per ordinare la struttura di partenza ma movimentano gli elementi all'interno della struttura stessa economizzando l'utilizzo della memoria (non c'è duplicazione dei dati)
- *Algoritmi di ordinamento non sul posto*: classe di algoritmi che partendo dalla struttura dati da ordinare ne costruisce una nuova parallela nella quale inserire gli elementi già ordinati (l'occupazione di memoria in questo caso raddoppia).

Un'ulteriore suddivisione può essere effettuata in base al fatto che alcuni algoritmi preservano la posizione relativa degli elementi uguali così come si trovavano nella struttura di partenza ed altri invece la invertono. I primi vengono detti *algoritmi di ordinamento stabili* mentre i secondi vengono detti *algoritmi di ordinamento non stabili*.

Algoritmo di ordinamento per ricerca del minimo (sul posto)

La struttura, inizialmente non ordinata, verrà ordinata partendo dalla ricerca del valore minimo che verrà spostato in posizione 0. La ricerca verrà poi ripetuta per gli elementi rimanenti (non ancora ordinati) trovando il nuovo valore di minimo che verrà spostato nella posizione 1 e così via fino all'ordinamento dell'intera struttura.

In altre parole la struttura dati di partenza verrà a trovarsi suddivisa in due sotto-strutture delle quali una sarà ordinata e l'altra ancora da ordinare. Man mano che l'algoritmo avvanzerà la struttura dati ordinata si espanderà a scapito di quella non ordinata che scomparirà al termine dell'algoritmo. Tale algoritmo viene comunemente detto **minSort**.

Nella figura sottostante viene riportato lo pseudocodice dell’algoritmo.



Da notare la presenza di due cicli *while do* “annidati”. Quello più esterno scorre tutti gli elementi dell’array e di volta in volta sposta il valore minimo nella posizione di destinazione. Il ciclo più interno cerca il valore minimo nella parte non ancora ordinata e lo fornisce al ciclo esterno. Per ogni passo del ciclo esterno il ciclo più interno scorre tutta la parte dell’array da ordinare. In linea di principio, dato un array di dimensione N l’algoritmo, nel caso peggiore, effettua $N \times N$ passi.

Strutture ricorsive

Le strutture ricorsive forniscono un'alternativa al paradigma iterativo per le strutture ripetitive di tipo *while() do*. In particolare, come abbiamo visto, le strutture iterative ripetono un gruppo di istruzioni un determinato numero di volte mentre la strutture ricorsive comportano la ripetizione dell'insieme di istruzioni come sotto-compiti eseguiti su dati parziali.

In altre parole, il paradigma ricorsivo prevede l'implementazione di procedure che vengono invocate su sotto-insiemi di dati.

Nella figura sottostante viene mostrato un esempio di utilizzo del paradigma ricorsivo.

```

1.  procedure Risolvi(problema p) (
2.    if(p e' semplice) then (
3.      risolvillo direttamente
4.    )else (
5.      suddividi p in sottoproblemi  $p_1, p_2, \dots, p_n$  risolvibili
6.      risolvi( $p_1$ ), risolvi( $p_2$ )... risolvi( $p_n$ )
7.      combina le soluzioni di  $p_1, p_2, \dots, p_n$  per avere quella di p
8.    )
9.  ) end procedure

```

In questo caso del tutto generale, la procedura *Risolvi* prende come argomento un problema e la prima cosa che fa' è controllare se tale problema è o meno risolvibile (se è abbastanza piccolo).

Nel caso in cui il problema non fosse risolvibile (perché troppo grande) la procedura lo suddivide in sottoproblemi più piccoli e invoca di nuovo sé stessa su questi nuovi problemi di dimensioni inferiori. Al termine la procedura combina le soluzioni dei sottoproblemi per ottenere la soluzione del problema di partenza.

Ricerca di un valore key in un elenco ordinato: implementazione ricorsiva.

Questa volta abbiamo a disposizione un elenco ordinato di valori e vogliamo valutare la presenza al suo interno del valore a noi noto (key). In linea di principio, non è necessario scorrere tutto l'elenco in quanto, essendo ordinato, sappiamo dove è meglio cercare. In particolare, si parte suddividendo l'elenco a metà e valutando in quale delle due parti dovrebbe trovarsi l'elemento (questo si ottiene semplicemente confrontando l'elemento centrale con il nostro elemento key).

Una volta identificata la parte in cui dovrebbe trovarsi l'elemento la si suddivide ulteriormente in due e la procedura continua fino a quando non si trova l'elemento. Tale algoritmo viene anche detto: di **ricerca binaria**.

Nella figura sottostante viene mostrato l'algoritmo in pseudocodice.

```

1.  procedure ricercaBin(Dati[], key, sx, dx)
2.      m ← (sx+dx)/2
3.      if(m < sx)then (
4.          Stampa "Elemento non presente"
5.          return -1
6.      )else if(key < Dati[m])then (
7.          return ricercaBin(Dati, key, sx, m-1)
8.      )else if(key > Dati[m])then (
9.          return ricercaBin(Dati, key, m+1, dx)
10.     )else (
11.         Stampa "elemento trovato"
12.         return m
13.     )
14. )

1.  start
2.  Key ← input
3.  Dati[] ← input
4.  N ← input
5.  Sx ← 0
6.  Dx ← N-1
7.  ricercaBin(Dati, key, Sx, Dx)
10. end

```

Nella figura si può notare l'unica invocazione alla procedura ricorsiva fatta dall'algoritmo (riga 7 blocco in basso). In questo esempio, in particolare, la procedura *ricercaBin* prende come argomenti l'array di dati, l'elemento *key* da ricercare e gli estremi che indicano in quale parte dell'array ricercare (individuazione di un sotto-problema). Si può notare come l'invocazione alla procedura nell'algoritmo (riga 7 blocco in basso) avvenga su tutto l'array, infatti, prima dell'invocazione vengono posti rispettivamente $Sx = 0$ e $Dx = N-1$. (Sx e Dx sono il limite destro e sinistro dell'array su cui ricercare l'elemento).

Notare che all'interno della procedura viene inizialmente individuata la metà dell'array rispetto agli estremi di ricerca ($m \leftarrow (sx + dx)/2$). In seguito viene confrontato il valore da ricercare *key* con il valore dell'array nel punto medio. Se *key* risulta maggiore di tale valore la ricerca continua a destra del valore *m* altrimenti avviene a sinistra. Si nota infatti che viene invocata di nuovo la procedura con parametri differenti. In particolare, nel caso in cui la ricerca debba continuare a destra del punto *m* si invoca la procedura con estremi di ricerca $Sx = m+1$ e $Dx = dx$. Nel caso si debba continuare la ricerca a sinistra, invece, l'invocazione avviene con $Sx = sx$ e $Dx = m-1$. In seguito a una di queste nuove invocazioni il flusso di esecuzione riparte dall'inizio della procedura determinando un'ulteriore suddivisione del campo di ricerca in due e così via.

Alla fine ci troveremo in una condizione in cui il campo di ricerca non è ulteriormente suddivisibile in quanto composto da un solo elemento. In questo caso l'algoritmo ha termine e, se l'elemento coincide con quello da ricercare (*key*) avremo trovato l'elemento, altrimenti potremmo dire che l'elemento non era presente nell'array.

Complessità.

Un problema è detto *indecidibile* se non esiste un algoritmo che lo risolve in tempo finito ed è detto decidibile se non è *indecidibile*

Gli algoritmi che risolvono lo stesso problema vengono confrontati tra di loro in termine di tempo (o passi) d'esecuzione.

Il numero di passi d'esecuzione di un algoritmo ne rappresenta la complessità in termini indipendenti dal tempo d'esecuzione di ogni passo. Facendo riferimento implicito ad un calcolatore elettronico che impiega un tempo costante ad eseguire le istruzioni elementari del proprio instruction set, si assume che ogni passo d'esecuzione richieda un tempo costante, e che quindi il numero di passi sia proporzionale al tempo di esecuzione totale. Poiché il numero di passi dipende generalmente dai dati d'ingresso, la complessità di un algoritmo non è espressa da un numero, ma dalla funzione che lega il numero di passi al valore dei dati d'ingresso.

Per convenzione, nell'esprimere la complessità di un algoritmo si trascurano sia le costanti moltiplicative che quelle additive, limitandosi a specificare il tipo di dipendenza funzionale dominante dai dati d'ingresso.

Tale funzione viene calcolata per il limite della dimensione dell'input che tende all'infinito e viene detta *complessità asintotica*.

- Classi di complessità asintotica:
 - $O(1)$: costante
 - $O(\text{Log}(n))$: logaritmica
 - $O(n)$: lineare
 - $O(n*\text{Log}(n))$: pseudologaritmica
 - $O(n^2)$: quadratica
 - $O(n^R)$ $R > 0$: polinomiale
 - $O(a^n)$ $a > 1$: esponenziale

Un problema decidibile è detto *computazionalmente trattabile* se esiste un algoritmo di complessità al più polinomiale che lo risolve. Algoritmi con complessità esponenziale non vengono trattati in quanto la velocità di esecuzione è talmente lenta da renderli inutilizzabili.

- Esempi:
 - Ricerca lineare: $O(n)$
 - Ricerca binaria: $O(\text{Log}2(n))$
 - Ricerca del min: $O(n)$
 - MinSort: $O(n^2)$
 - BubbleSort: $O(n^2)$
 - BidirectionalBubbleSort: $O(n^2)$
 - QuickSort: $O(n*\text{Log}2(n))$

Ref: J. Gleen Brookshear, "Informatica una panoramica generale"