

1 Introduzione

La parola **algoritmo**, un tempo utilizzata quasi esclusivamente da matematici e informatici, è oggi sempre più diffusa anche nel linguaggio comune.

Spesso viene impiegata in modo **vago o impreciso** per riferirsi a qualunque forma di automazione, decisione informatica o comportamento opaco dei **sistemi digitali**.

Oltre a definire genericamente sistemi informatici, capita spesso che il termine algoritmo venga **usato per qualunque descrizione** di un procedimento che **risolva un problema**. Dunque, diamo delle definizioni che colleghino i concetti di algoritmo e problema computazionale.

1.1 Problemi computazionali e algoritmi

Cos'è un problema computazionale?

Dati un dominio di input e un dominio di output, un **problema computazionale** è rappresentato dalla **relazione matematica** che associa ogni elemento del dominio in input ad uno o più elementi del dominio di output.

Esempio: Immaginiamo di dover seguire una ricetta. L'input è dato dagli ingredienti, l'output è dato dal piatto cucinato, mentre il sistema formale di calcolo è dato dal cuoco.

Cos'è un algoritmo?

Dato un problema computazionale, un algoritmo è un procedimento **effettivo** (di calcolo), espresso tramite un insieme di **passi elementari ben specificati** in un sistema formale di calcolo, che risolve il problema in un **tempo finito**.

L'espressione "**in un tempo finito**" implica che l'algoritmo deve terminare dopo un numero definito di passi, mentre "**passi elementari ben specificati**" si riferisce a operazioni descritte con precisione, realizzabili da un esecutore automatico.

1.1.1 Esempi di problemi computazionali e algoritmi

Per comprendere al meglio la differenza tra problema computazionale e algoritmo si considerino i seguenti problemi:

Problema del minimo

Il minimo di un insieme S è l'elemento di S che è minore o uguale ad ogni elemento di S .

$$\min(S) = a \Leftrightarrow \exists a \in S : \forall b \in S : a \leq b$$

Problema di ricerca

Sia $S = S_1, S_2, \dots, S_n$ una sequenza di dati ordinati e distinti, dove $S_1 \leq S_2 \leq \dots \leq S_n$. Eseguire una ricerca dalla posizione di un dato v in S consiste nel restituire l'indice corrispondente, se v è presente, oppure -1 se v non è presente.

$$lookup(S, v) = \begin{cases} i & \exists i \in 0, \dots, n - 1 : S_i = v \\ -1 & \text{altrimenti} \end{cases}$$

Dunque, esiste una distinzione ben precisa fra *il problema computazionale* che si vuole risolvere e gli *algoritmi* che lo risolvono.

Mentre un problema computazionale specifica quale relazione si desideri tra l'ingresso e l'uscita (cioè il risultato), l'algoritmo descrive la sequenza di azioni da eseguire per ottenere l'uscita desiderata a partire dall'ingresso.

In questo caso, dei possibili algoritmi che risolvono i problemi, sono i seguenti:

- **Problema del minimo:** Per trovare il minimo di un insieme, confronta ogni elemento con tutti gli altri; l'elemento che è minore di tutti è il minimo.
- **Problema di ricerca:** Per trovare un valore v nella sequenza S , confronta v con tutti gli elementi di S , in sequenza, e restituisci la posizione corrispondente; restituisci -1 se nessuno degli elementi corrisponde.

Gli algoritmi che vengono scelti per la risoluzione di un problema computazionale devono presentare **passi non ambigui** ed **eseguibili**, proprio per questo un generico algoritmo è caratterizzato dalle seguenti **proprietà**:

- **Atomicità:** l'algoritmo deve essere composto da passi elementari non scomponibili;
- **Non ambiguità:** l'algoritmo non deve generare errori;
- **Attualità:** l'algoritmo deve essere eseguibile;
- **Finitezza:** si intende che l'algoritmo deve terminare in un tempo finito;
- **Effettività:** l'algoritmo deve portare ad un risultato univoco.

1.1.2 Come valutare l'algoritmo scelto

Valutazione di un algoritmo

La valutazione di un algoritmo consiste nello stabilire se quest'ultimo risolve il problema in modo **efficiente** e **corretto**.

Per quanto riguarda il grado di **efficienza**:

- Alcuni problemi **non possono** essere risolti in modo **efficiente**, quindi non esistono algoritmi noti che risolvano il problema.
- Allo stesso modo, esistono soluzioni "ottime" che non possono essere portate ad un grado di efficienza maggiore proprio perché non è possibile essere più efficienti;
- Altri problemi presentano, banalmente, delle soluzioni che funzionano ma non sono particolarmente efficienti;

Invece, la **correttezza di un algoritmo** viene dimostrata tramite:

- Per quanto possibile, con una descrizione matematica;
- Oppure utilizzando una descrizione "informale".

N.B: alcuni problemi non possono essere risolti. Proprio perché alcuni problemi non possono essere risolti in maniera efficiente e corretta, vengono risolti in maniera approssimata.

1.1.3 Complessità di un algoritmo

La ricerca di un algoritmo efficiente e corretto, porta alla definizione di un ulteriore concetto, quello della "*complessità di un algoritmo*".

Complessità di un algoritmo

La **complessità di un algoritmo** è data dall'analisi delle risorse da esso impiegate per risolvere un problema, in funzione della dimensione e della tipologia dell'input.

Quando si parla di *risorse impiegate da un algoritmo*, ci si riferisce a:

- **Tempo**: ovvero il tempo impiegato per completare l'algoritmo.

Misurare il tempo di esecuzione di un algoritmo considerando il tempo di calcolo in senso assoluto (secondi, minuti, ecc.) rappresenta un **approccio errato** perché dipende da troppi parametri a seconda della piattaforma utilizzata per la misura:

- Frequenza del processore;
- Linguaggio di programmazione utilizzato;
- Ottimizzazione del compilatore utilizzato;
- Interazione tra memoria primaria (anche memorie cache) e secondaria;
- Velocità di trasmissione dati del bus;
- Processi attualmente in esecuzione;
- ecc ...

Proprio per questo motivo quando si parla di *tempo impiegato dall'algoritmo* ci si riferisce al **numero di operazioni rilevanti**, ovvero il numero di operazioni che caratterizzano lo scopo dell'algoritmo, come ad esempio, contare operazioni elementari come i confronti, che rappresentano una misura più stabile e indipendente dalla piattaforma (capitolo ??).

- **Spazio**: Quantità di memoria utilizzata;
- **Banda**: quantità di bit spediti.

1.2 Strutture dati

In un linguaggio di programmazione, un **dato** è un valore che una variabile può assumere.

Cos'è una struttura dati?

Una **struttura dati** è un modo per organizzare e memorizzare i dati sui supporti fisici.

Le strutture dati offrono il **vantaggio** di permettere una **buona organizzazione** dei dati permettendo così di **semplificare l'accesso** e la **modifica** degli stessi, influendo in modo diretto sull'efficienza dell'algoritmo.

Dunque, la caratteristica principale non è tanto il *tipo dei dati*, contenuti all'interno della struttura, ma il **modo in cui viene organizzata la collezione**.

Possiamo definire una struttura dati utilizzando **due elementi**:

- **Insieme di operatori**: ovvero l'insieme di operatori che permettono di manipolare la struttura (inserimenti, eliminazioni, ricerca, ordinamento, ecc...) ;
- **Modo sistematico di organizzare i dati**: indica il modo in cui i dati sono memorizzati e collegati tra loro (ad esempio, in modo sequenziale, a grafo o ad albero).

È importante notare che **non esiste una struttura dati adatta a qualsiasi compito**, quindi risulta utile considerare e conoscere i **vantaggi e svantaggi** di diverse strutture.

1.2.1 Tipologie di strutture dati

Le strutture dati possono essere classificate in base a diversi criteri:

- **Linearie e non linearie:** nelle strutture linearie gli elementi sono **disposti in sequenza** (ad esempio, array o liste), mentre nelle strutture non linearie ogni elemento **può collegarsi a più elementi** (ad esempio alberi o grafi);
- **Statiche e dinamiche:** nelle strutture statiche la **dimensione è fissa** (come ad esempio negli array), mentre nelle strutture dinamiche la dimensione **può variare nel tempo** (ad esempio liste collegate);
- **Omogenee e disomogenee:** le strutture omogenee presentano una collezione di **dati dello stesso tipo** (come ad esempio un array di interi), invece, le strutture disomogenee presentano collezioni di **dati di tipologie differenti**.

1.2.2 Le sequenze

Cos'è una sequenza?

Una **sequenza**, è una struttura dati **dinamica e lineare** che rappresenta una sequenza ordinata di valori, all'interno della quale uno stesso valore può **comparire anche più di una volta**.

L'**ordine** della collezione di dati all'interno della sequenza è **importante**, di tipo **posizionale**. Data una generica sequenza $S = S_1, S_2, \dots, S_n$ è possibile aggiungere o togliere elementi, mantenendo la struttura ordinata della sequenza: per indicare un generico elemento S_i della sequenza si utilizza il parametro pos_i , che rappresenta una **posizione logica** nella sequenza; dunque, è possibile accedere direttamente alla testa con pos_0 e alla coda con pos_n .

Alcuni esempi: di seguito verranno illustrate alcune operazioni effettuate sulle sequenze.

SEQUENCE

% Restituisce **true** se la sequenza è vuota

boolean isEmpty()

% Restituisce **true** se p è uguale a pos_0 oppure a pos_{n+1}

boolean finished(Pos p)

% Restituisce la posizione del primo elemento

Pos head()

% Restituisce la posizione dell'ultimo elemento

Pos tail()

% Restituisce la posizione dell'elemento che segue p

Pos next(Pos p)

% Restituisce la posizione dell'elemento che precede p

Pos prev(Pos p)

SEQUENCE (continua)

% Inserisce l'elemento v di tipo ITEM nella posizione p .
% Restituisce la posizione del nuovo elemento, che diviene il predecessore di
 p

Pos insert(Pos p , ITEM v)

% Rimuove l'elemento contenuto nella posizione p .
% Restituisce la posizione del successore di p ,
% che diviene successore del predecessore di p

Pos remove(Pos p)

% Legge l'elemento di tipo ITEM contenuto nella posizione p

ITEM read(Pos p)

% Scrive l'elemento v di tipo ITEM nella posizione p

write(Pos p , ITEM v)

Esempio di creazione di una sequenza il linguaggio C++

```
std::list<int> lista;  
lista.push_front(2);  
lista.push_front(1);  
lista.push_back(3);  
  
Result: [1, 2, 3]
```

1.2.3 Gli Insiemi

Cos'è un insieme?

Un **insieme** è una struttura dati **dinamica e non lineare** che memorizza una collezione **non ordinata** di elementi senza valori ripetuti.

A differenza delle sequenze, per ordinare un insieme, bisogna introdurre il concetto di **relazione d'ordine**. Infatti, nelle sequenze (come array o liste), l'ordine degli elementi non dipende **dal loro valore**, ma dalla **posizione che occupano**. Quindi non serve definire una "relazione d'ordine" tra gli elementi: la struttura stessa impone un ordine.

Gli insiemi, invece, sono per definizione una collezione di elementi unici ma senza un ordine posizionale. Quindi, in un **insieme**, l'ordinamento fra elementi è dato dall'**eventuale relazione d'ordine definita sul tipo degli elementi stessi**. Per questo motivo, se vogliamo ordinare gli elementi, è necessario specificare in che modo come avviene il confronto.

Esempio: nel caso degli **interi**, il confronto è immediato, poiché esiste un ordine naturale tra i valori numerici ($1 < 2 < 3 < \dots$). Per le **stringhe**, invece, l'ordinamento si basa sull'**ordine lessicografico**, ossia sul confronto dei caratteri secondo la sequenza alfabetica (ad esempio, "cane" precede "gatto" perché la lettera "c" viene prima della "g" nell'alfabeto).

Lavorando con gli insiemi, le operazioni ammesse sulle collezioni di dati sono le seguenti:

- **Operazioni base:** inserimento, cancellazione, verifica contenimento;
- **Operazioni di ordinamento:** massimo e minimo;
- **Operazioni di insiemistiche:** unione, intersezione, differenza;
- **Iteratori:** $\text{foreach } x \in S \text{ do}$

SET

% Restituisce la cardinalità dell'insieme
int = ize()

% Restituisce **true** se *x* è contenuto nell'insieme
boolean contains(ITEM *x*)

% Inserisce *x* nell'insieme, se non già presente
insert(ITEM *x*)

% Rimuove *x* dall'insieme, se presente
remove(ITEM *x*)

% Restituisce un nuovo insieme che è l'unione di *A* e *B*
SET union(SET *A*, SET *B*)

% Restituisce un nuovo insieme che è l'intersezione di *A* e *B*
SET intersection(SET *A*, SET *B*)

% Restituisce un nuovo insieme che è la differenza di *A* e *B*
SET difference(SET *A*, SET *B*)

Esempio di creazione di un insieme il linguaggio C++

```
std::set<std::string> frutta;
frutta.insert("mele");
frutta.insert("pere");
frutta.insert("banane");
frutta.insert("mele");
frutta.remove("mele");
Result: ["banane", "pere"]
```

1.2.4 I dizionari

Cos'è un dizionario?

Un **dizionario** è una struttura dati che rappresenta il concetto matematico di relazione univoca $R : D \rightarrow C$, o associazione chiave valore, dove:

- L'insieme *D* è il dominio (elementi detti chiavi);
- L'insieme *C* è il codominio (elementi detti valori)

Con questa tipologia di struttura dati, sono ammesse le seguenti operazioni:

- Ottenere il valore associato ad una particolare chiave (se presente), o **nil (null)** se assente;
- Inserire una nuova associazione chiave-valore, cancellando eventuali associazioni precedenti per la stessa chiave;
- Rimuovere unassociazione chiave-valore esistente;

DICTIONARY

% Restituisce il valore associato alla chiave *k* se presente, **nil**

altrimenti

ITEM lookup(ITEM *k*)

% Associa il valore *v* alla chiave *k*

insert(ITEM *k*, ITEM *v*)

% Rimuove l'associazione della chiave *k*

remove(ITEM *k*)

1.3 I puntatori

Cos'è un puntatore?

Un **puntatore** è una variabile che contiene un **indirizzo di memoria**. Spesso, l'indirizzo è la locazione di memoria di un'altra variabile.

N.B: un puntatore non può contenere un valore che non sia un indirizzo.

Gli usi tipici dei puntatori sono la creazione di strutture dati collegate come alberi e liste, la gestione di oggetti allocati dinamicamente e come parametri di funzioni.

Ogni puntatore ha un tipo associato. La differenza non sta nella rappresentazione dei puntatori, ma nel tipo dell'oggetto puntato. Le variabili dei puntatori devono essere dichiarate come tali:

```
int *p; //Puntatore a intero  
float *p; //Puntatore a float
```

Gli operatori speciali che vengono utilizzati con i puntatori sono * e &:

- & → è un operatore **unario** che restituisce l'**indirizzo di memoria** del suo operando;

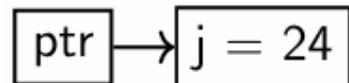
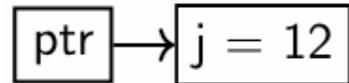
```
balptr = &balance //mette in balptr l'indirizzo di memoria di balance.
```

- * → è un operatore **unario** che restituisce il **valore** della variabile allocata all'indirizzo specificato dal suo operando.

```
value = *balptr //se balptr contiene l'indirizzo di balance, il valore della variabile balance viene messo in value.
```

Esempio: esercizio con puntatori

```
#include <iostream>  
int main() {  
    int j = 12;  
    int *ptr = &j;  
  
    cout << *ptr << endl;  
  
    j = 24;  
    cout << *ptr << endl;  
    cout << ptr << endl;  
    return 0;  
}
```



L'output del programma è il seguente:
12
24
0x7b03a928 (indirizzo di memoria)

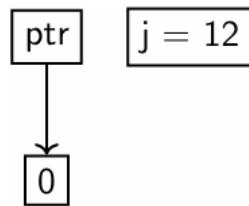
1.3.1 Puntatori nulli

Cos'è un puntatore?

Un puntatore può contenere il valore 0 a indicare che non punta ad alcun oggetto.
In questo caso viene detto **puntatore nullo**.

In questo caso, provando a stampare un puntatore nullo, si ottiene un errore.

```
#include <iostream>
int main() {
    int j = 12;
    int *ptr = 0;
    cout << *ptr << endl; // crash !
    return 0;
}
```



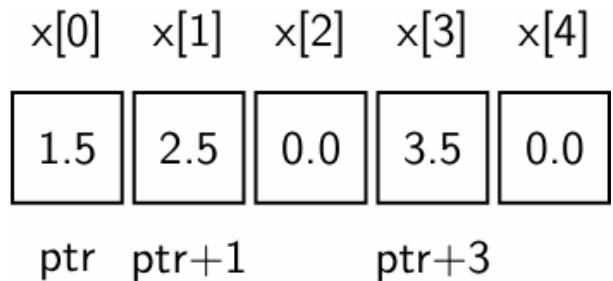
Segmentation violation (core dumped)!

Esempio: come i puntatori vengono utilizzati con gli array

```
//Alcune implementazioni di array in c++
int x[] = { 1, 2, 3, 4 };
char[] t = { 'C', 'i', 'a', 'o', '\0' };
char[] s = "Ciao";
int m[2][3] = { {11, 12, 13}, {21, 22, 23} };
```

```
int main() {
    float x[5];
    int j;

    for (j = 0; j < 5; j++) {
        x[j] = 0;
    }
    float *ptr = x;
    *ptr = 1.5; // x[0] = 1.5
    *(ptr+1) = 2.5; // x[1] = 2.5
    *(ptr+3) = 3.5; // x[3] = 3.5
}
```



Utilizzando gli array, non è necessario specificare che `ptr` punti ad un indirizzo, come fatto precedentemente (`*ptr = &j`), proprio perché `x` rappresenta di per sé l'indirizzo della prima posizione dell'array.