Capitolo 1

Introduzione

Definizione (problema computazionale)**.** Dato un dominio di input e un dominio di output, un *problema computazionale* è rappresentato dalla **relazione matematica** che associa un elemento del dominio in output ad ogni elemento del dominio di input.

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

Un esempio classico, ma ingannevole, di algoritmo è quello della preparazione di una ricetta: l'input sono gli ingredienti, l'esecutore è il cuoco, l'algoritmo è la ricetta e l'output è rappresentato dal piatto cucinato. È ingannevole perché non esiste un modello formale del cuoco, ossia qualcosa che descriva esattamente cosa un cuoco può fare e quali sono i passi elementari dell'algoritmo.

Algoritmi nella storia

Gli algoritmi sono molto antichi ed esistevano ancor prima del concetto di elaboratore.

Nè è un esempio il Papiro di Rhind o di Ahmes che mostra l'algoritmo del contadino per la moltiplicazione. Algoritmi di tipo numerico furono studiati da matematici babilonesi ed indiani. Esistono algoritmi in uso fino a tempi recenti che furono studiati dai matematici greci più di 2000 anni fa. Come ad esempio l'algoritmo di Euclide per il massimo comune divisore e gli algoritmi geometrici come il calcolo di tangenti, sezioni di angoli, etc.

Origine del nome

L'*origine del nome* ***algoritmo*** è dovuto al matematico, astronomo, astrologo e geografo Abu Abdullah Muhammad bin Musa **al-Khwarizmi**. Ha scritto un testo chiamato "Algoritmi de numero indorum", traduzione di un testo arabo ormai perso, che ha introdotto i numeri indiani (da noi comunemente detti numeri arabi) nel mondo occidentale.

Ha inoltre scritto un'altra opera "Al-Kitab al-muhtasar fi hisab **al-gabr** wa-l-muqabala" che è stata tradotta in latino come "Liber algebrae et almucabala" dalla quale ha avuto *origine il nome* ***algebra***.

1.1 Primi esempi di problemi

Per iniziare a comprendere ciò che intendiamo per problema computazionale iniziamo a vedere degli algoritmi che sono espressi in modo volutamente banale.

1.1.1 Minimo

Definizione del problema Il minimo di un insieme S è l'elemento di S che è minore o uguale ad ogni altro elemento di S. Possiamo esprimere matematicamente il problema nel seguente modo:

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

Partiamo da una soluzione semplice ma ingenua, che deriva dalla definizione del problema.

Algoritmo naïf Per trovare il minimo di un insieme, confronto ogni elemento con tutti gli altri; l'elemento che è minore di tutti è il minimo.

A questo punto ci accorgiamo dell'ambiguità della lingua italiana, infatti che cosa s'intende per "confronta ogni elemento con tutti gli altri" ? Non abbiamo un insieme di passi elementari ben specificati.

1.1.2 Ricerca dell'indice di un elemento

Definizione del problema Sia S una sequenza di dati $s_1, s_2, \ldots, s_n$ ordinati e distinti, ad esempio $s_1 > s_2 > \cdots > s_n$. Eseguire una ricerca della posizione di un dato valore v nell'insieme S consiste nel restituire un indice i tale che $1 \leqslant i \leqslant n$ (ossia che sia compreso fra 1 ed n, estremi inclusi), se v è presente nella posizione i, oppure 0, se v non vi è presente. Possiamo esprimere matematicamente il problema nel seguente modo:

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

Notiamo che, avendo caratterizzato la sequenza in modo tale che abbia *elementi distinti*, abbiamo semplificato il problema al caso in cui non ci sono ambiguità nella restituzione dell'indice.

Algoritmo naïf Per trovare il valore v nella sequenza S, confronto v con tutti gli elementi di S, in sequenza, e restituisco la posizione corrispondente; restituisco 0 se nessuno degli elementi corrisponde.

1.1.3 Problemi riscontrati

Le descrizioni degli algoritmi precedenti presentano diversi problemi:

- la **descrizione** dei problemi è stata fatta in linguaggio naturale, il quale è intrisicamente ambiguo, abbiamo quindi bisogno di un linguaggio più formale.
- come possiamo **valutare** se esistono algoritmi migliori di quelli proposti? Per farlo dobbiamo definire il concetto di "migliore".

1.1.4 Come descrivere un algoritmo

Come abbiamo già anticipato è necessario avere una descrizione il più possibile formale, ma senza soffermarsi sulle particolarità di un determinato linguaggio di programmazione. Per questo motivo nel seguito utilizzeremo lo "Pseudo-codice".

In alcuni casi daremo anche per scontato alcuni passaggi, ad esempio scriveremo "ordina gli elementi" e non sarà di nostro interesse sapere quale sarà lo specifico algoritmo di ordinamento utilizzato.

Esempi di Pseudo-codice sono i seguenti.

Implementazione naïf della ricerca del minimo

```
int min(int[] S, int n)
    from i ← 1 until n do // per ogni elemento del vettore
        bool isMin ← true // assumo di aver trovato il minimo
        from j ← 1 until n do // confronto l'elemento con tutti gli altri
            if i ≠ j and S[j] < S[i] then // se trovo un valore più piccolo
                isMin ← false // quell'indice non contiene l'elemento minimo
        // se dopo aver controllare ogni indice...
        if isMin then // ...trovo un valore che ha conservato il valore true
            return S[i] // l'elemento in posizione i-esima è il più piccolo
```

Nota che specificheremo sempre com'è fatto l'input, ossia passeremo in ingresso sia il vettore (S), sia la lunghezza del vettore passato (n).

Implementazione naïf della ricerca dell'indice di un elemento

```
int lookup(int[] S, int n, int v)
    from i ← 1 until n do // per tutti gli elementi del vettore
        if S[i] == v then // confronto tutti i valori con quello cercato
            return i // se lo trovo lo restituisco
    return 0 // altrimenti restituisco 0
```

I linguaggi di programmazione assumono che l'indice dei vettori parta da 0, ma per spiegarne alcuni è molto più semplice concentrarsi sugli indici che vanno da 1 a n e tralasciare questi dettagli implementativi.

Convenzioni dello pseudocodice

Di seguito sono riportate le convenzioni utilizzate sui lucidi e sugli appunti:

- l'assegnamento verrà indicato con $a \leftarrow b$;
- uno *swap* verrà indicato come $a \leftrightarrow b$, il quale corrisponde ai comandi $tmp \leftrightarrow a$; $a \leftarrow b$; $b \leftarrow tmp$;
- i vettori come $\mathbf{T}[\,]\ A = \mathsf{new}\ \mathbf{T}[1 \dots n]$;
- le matrici come $\mathbf{T}[\,][\,]\ A = \mathsf{new}\ \mathbf{T}[1 \dots n][1 \dots n]$;
- i tipi come **int**, **float**, **bool**, ...;
- i simboli logici come **and**, **or**, **not**;
- i simboli relazionali con $==$, $\neq$, $\leqslant$, $\geqslant$;
- i simboli matematici con $+$, $-$, $\cdot$, $/$, $\lfloor x \rfloor$, $\lceil x \rceil$, log, x^2, ...;
- talvolta scriveremo l'operatore if ternario come $\mathsf{iif}(condizione,\ v_1,\ v_2)$;
- **if** *condizione* **then** *istruzione*;
- **foreach** *elemento* $\in$ *insieme* **do** *istruzione*.

1.2 Come valutare un algoritmo

Quando valutiamo un algoritmo dobbiamo chiederci se risulta *corretto* ed *efficiente*.

Per quanto riguarda l'efficienza dobbiamo ancora stabilire come valutare se un programma è efficiente, e se lo è in assoluto. Nota che alcuni problemi non possono essere risolti in modo efficiente ma esistono soluzioni "ottime", ossia non è possibile essere più efficienti di così.

Per controllare la correttezza dobbiamo domandarci se il nostro algoritmo rispetta la relazione input-output del problema computazionale. Nota che alcuni problemi non possono essere risolti, mentre alcuni vengono risolti in modo approssimato.

1.2.1 Efficienza

Definizione (complessità di un algoritmo)**.** Analisi delle *risorse* impiegate da un algoritmo per risolvere un problema, in funzione della *dimensione* e della *tipologia* dell'input.

Le risorse si possono categorizzare in:

- **tempo**: ossia il tempo impiegato per completare l'algoritmo (definiremo più avanti come lo misureremo);

- **spazio**: la quantità di memoria utilizzata;
- **banda**: la quantità di bit spediti (interessante solo per gli algoritmi distribuiti).

Definizione di tempo

Il tempo effettivamente impiegato per eseguire un algoritmo dipende da troppi parametri come la bravura del programmatore, il linguaggio di programmazione utilizzato (C è più efficiente di Python), il codice generato dal compilatore, dalla velocità del processore e della memoria e dai processi attualmente in esecuzione sul sistema operativo.

È quindi necessario considerare una rappresentazione più astratta. Ad esempio potremmo considerare il *numero di operazioni "rilevanti"*, ovvero il numero di operazioni che caratterizzano lo scopo dell'algoritmo.

1.2.2 Primi esempi di calcolo della complessità

Nel caso della ricerca del minimo, l'operazione più rilevante è il numero di confronti di minoranza (<), nella caso della ricerca dell'indice invece è il numero di confronti di eguaglianza (==).

Implementazione naïf della ricerca del minimo

```
int min(int[] S, int n)
    from i ← 1 until n do // ciclo con n elementi
        bool isMin ← true
        from j ← 1 until n do // ciclo con n elementi
            if i ≠ j and S[j] < S[i] then // il confronto avviene solo se gli indici sono diversi
                isMin ← false
        if isMin then
            return S[i]
```

Calcolo della complessità Come calcoliamo la complessità? L'operazione rilevante all'interno di questo algoritmo è il confronto fra gli elementi agli indici i e j ($S[j] < S[i]$), la quale è ripetuta all'interno di due cicli annidati che scorrono gli n elementi del vettore, tranne quelli che hanno lo stesso indice. Quindi il numero di confronti totali per questa soluzione del problema del minimo è $n \cdot n - n = n^2 - n$. Si può fare meglio di così? Certo che sì!

Implementazione efficiente per la ricerca del minimo

```
int min(int[] S, int n)
    int min ← S[1] // minimo parziale
    from i ← 2 until n do // dal secondo elemento in poi...
        if S[i] < min then // ...confronto il minimo parziale con l'elemento corrente
            min ← S[i] // aggiorno il minimo parziale
    return min // restituisco il minimo trovato
```

Con questa implementazione effettuiamo $n - 1$ confronti perché il primo elemento non deve essere confrontato.

Questo algoritmo effettua n confronti ($S[i] == v$), uno per ogni elemento del vettore. Si può fare meglio di così?

Implementazione naïf della ricerca dell'indice di un elemento

```
int lookup(int[] S, int n, int v)
    from i ← 1 until n do
        if S[i] == v then
            return i

    return 0
```

Sfruttando il fatto che la *sequenza è ordinata* possiamo applicare la **ricerca binaria**. Prendiamo l'elemento centrale (di indice m) del sottovettore considerato: se l'elemento contenuto all'indice m è pari all'elemento cercato ($A[m] == v$) allora ho trovato il valore e lo restituisco al chiamante, altrimenti se l'elemento contenuto all'indice m è più piccolo dell'elemento cercato ($A[m] < v$) allora dovrò continuare la ricerca nella "metà di destra" ($m+1$, j), infine se l'elemento contenuto all'indice m è più grande dell'elemento cercato ($A[m] > v$) dovrò continuare la ricerca nella "metà di sinistra" (i, $m-1$). Nel caso sfortunato in cui l'elemento non esistesse all'interno del vettore gli indici si incroceranno, ed è questo che il caso base controlla all'inizio di ogni chiamata della funzione.

Algoritmo 1.2.1: Ricerca binaria all'interno di un vettore

```
// Effettua una ricerca binaria su un vettore di lunghezza arbitraria
binarySearch(ITEM[] A, ITEM v, int i, int j)
    if i > j then // se i cursori si incrociano
        return 0 // l'elemento non esiste
    else
        int m ← ⌊(i+j)/2⌋ // assegna la mediana
        if A[m] == v then // abbiamo trovato l'elemento
            return m // lo restitisco al chiamante
        else if A[m] < v then// se l'elemento cercato è più grande
            return binarySearch(A, v, m+1, j) // cerco nella seconda metà
        else // altrimenti
            return binarySearch(A, v, i, m-1) // cerco nella prima metà
```

Complessità Ad ogni passo il numero di elementi che considero viene dimezzato, quindi la complessità è logaritmica (più precisamente $\lceil \log \rceil$).

Nota. Tutte le volte che scriveremo un logaritmo sarà da intendere con base 2. Siamo informatici!

1.2.3 Correttezza

Per valutare la correttezza di un algoritmo possiamo utilizzare il concetto di invariante.

Definizione (invariante)**.** Condizione sempre vera *in un certo punto* del programma.

Definizione (invariante di ciclo)**.** Una condizione sempre vera all'inizio dell'iterazione di un ciclo.

Definizione (invariante di classe)**.** Una condizione sempre vera al termine dell'esecuzione di un metodo della classe.

Facciamo un esempio di invariante di ciclo, in quanto la utilizzeremo più avanti nella nostra trattazione.

Invariante di ciclo

Il concetto di **invariante di ciclo** ci aiuta a dimostrare la correttezza di un **algoritmo iterativo**. Distinguiamo tre fasi:

1. **inizializzazione** (caso base): la condizione è vera alla prima iterazione di un ciclo;
2. **conservazione** (passo induttivo): se la condizione è vera prima di un'iterazione del ciclo, allora rimane vera anche al termine (quindi prima della successiva iterazione);
3. **conclusione**: quando il ciclo termina, l'invariante deve rappresentare la "correttezza" dell'algoritmo.

Invariante di ciclo nella ricerca del minimo

Prendiamo ancora un'ultima volta come esempio l'algoritmo della ricerca del minimo.

L'invariante di ciclo per questo algoritmo è la seguente: all'inizio di ogni iterazione del ciclo, la variabile min contiene il minimo parziale degli elementi $S[1 \dots i-1]$. Quindi quando analizzo l'elemento i-esimo so che min contiene il minimo fra tutti gli elementi precendenti.

Inizializzazione (caso base): quando entriamo nel ciclo la variabile min contiene il minimo fra $S[1 \dots i-1]$ ed è vero poiché $i = 2$ quindi $i - 1 = 1$ e la variabile contiene il minimo fra il primo elemento ed il primo elemento. **Conservazione** (passo induttivo): nel momento in cui termino il ciclo la variabile min contiene il minimo parziale fra i ed n. **Conclusione**: al termine del ciclo i è pari a $n+1$, quindi la variabile min contiene il minimo parziale fra gli elementi compresi fra 1 e $i-1$, ma visto che i vale $n+1$, allora conterrà gli elementi il minimo fra gli elementi compresi fra $n+1-1 = n$. L'invariante di ciclo risulta quindi rispettata e rappresenta la correttezza del nostro algoritmo.

Questa analisi risulta eccessivamente dettagliata per un algoritmo così semplice, ma in futuro dovremo utilizzare l'invariante di ciclo per dimostrare algoritmi più complessi, è quindi di fondamentale importanza impararne i prìncipi.

Dimostrazione per induzione nella ricerca binaria

La **dimostrazione per induzione** è utile anche nel caso in cui si abbia a che fare con un **algoritmo ricorsivo**.

Infatti è possibile dimostrare la correttezza della ricerca binaria per induzione sulla dimensione n dell'input. Dove n è il numero di elementi passati alla funzione.

Il *caso base* si avvera quando non passiamo nessun elemento alla funzione ($n = 0$), ossia quando gli indici si sono incrociati ($i > j$) poiché non abbiamo trovato l'elemento cercato all'interno del vettore. Per *ipotesi* supponiamo che l'algoritmo sia corretto per tutti i valori n' più piccoli di n. Il *passo induttivo* è quindi costituito da tre casi:

1. trovo l'elemento e lo restituisco (il caso più semplice, $A[m] == v$);
2. l'elemento confrontato è più piccolo di quello che sto cercando ($A[m] < v$) e, in quanto i valori sono ordinati, l'elemento cercato si troverà sicuramente nella metà di destra delimitata dagli indici $m+1$ e j, i quali determinano una porzione del vettore n' più piccola del vettore di partenza, quindi l'algoritmo risulta corretto per induzione;
3. si ragiona in modo speculare al secondo caso con $A[m] > v$.

Conclusioni

Noi abbiamo analizzato soltato due aspetti degli algoritmi, ossia la loro *correttezza* ed *efficienza*. Ci sono tanti altri aspetti che si potrebbero guardare come ad esempio la semplicità, la modularità, la mantenibilità, l'espandibilità e la robustezza. I quali risultano però di secondaria importanza per un corso di algoritmi, e vengono trattati in modo approndito in un corso di ingegneria del software.

Alcune proprietà hanno un costo aggiuntivo in termini di prestazioni: ad esempio per scrivere codice modulare dobbiamo pagare il costo della gestione delle chiamate o per scrivere codice Java dobbiamo pagare il costo di interpretazione. Progettare algoritmi efficienti è quindi un prerequisito per poter pagare questo costo.