1) Introduzione

Prima di qualunque altra considerazione è importante capire cosa sia l'informatica. Si può facilmente osservare che al giorno d'oggi l'informatica permea la nostra vita quotidiana, sia quando essa è direttamente percepibile (ad esempio mentre navighiamo in Internet utilizzando un pc) sia quando è invisibile ("embedded", ossia per così dire "immersa" dentro un'automobile, un elettrodomestico, un telefono).

Senza l'informatica, la società contemporanea -così come la conosciamo- semplicemente non potrebbe continuare ad esistere. Non solo, ma ormai l'informatica permea anche apparecchiature dalle quali dipendono le vite delle persone (aeroplani di linea, apparecchiature mediche) che potrebbero essere messe in pericolo da malfunzionamenti derivanti da errori informatici per cui, oltre ad essere pervasiva, essa riveste un ruolo di assoluta importanza.

L'informatica viene spesso, ed a torto, considerata una mera attività pratica, per svolgere la quale è sufficiente un approccio dilettantistico e per cui non è necessaria una vera professionalità. Nulla di più inesatto: in realtà l'informatica è una disciplina scientifica. Ma non può essere considerata una sorta di "scienza dei calcolatori", poiché i calcolatori (o elaboratori) ne sono solo uno strumento: l'informatico può anche lavorare solamente con carta e penna. In realtà, l'informatica -intesa come disciplina scientifica- non coincide con alcuna delle sue applicazioni.

Dal dizionario Devoto-Oli della lingua italiana: "***L'informatica è la scienza che consente di ordinare, trattare e trasmettere l'informazione attraverso l'elaborazione elettronica...***"

In questa definizione le parole chiave sono due: *informazione* e *scienza*.

In effetti, la pervasività dell'informatica deriva dal fatto che in qualunque tipo di attività umana è necessario gestire qualche tipo di informazione, che va memorizzata ed elaborata, entrambe attività rigorose e sistematiche, ed è per questo che si richiede un approccio scientifico.

1.1 Algoritmi

La definizione di informatica proposta dall'ACM (Association for Computing Machinery), una delle principali organizzazioni scientifiche di informatici di tutto il mondo, è la seguente: ***"L'informatica è la scienza degli algoritmi che descrivono e trasformano l'informazione: la loro teoria, analisi, progetto, efficienza, realizzazione e applicazione."***

Iniziamo dunque con l'introdurre un concetto fondamentale, centrale per l'informatica, quello di ***algoritmo***.

Un algoritmo è "***una sequenza di comandi <u>elementari</u> ed <u>univoci</u> che terminano in un <u>tempo finito</u> ed operano su <u>strutture dati</u>***".

Un comando è *elementare* quando non può essere scomposto in comandi più semplici; è *univoco* quando può essere interpretato in un solo modo.

Ad esempio, domandiamoci se la seguente ricetta per fare un uovo al tegamino sia o no un algoritmo:

1. Rompere un uovo in padella.
2. Cuocere l'uovo.

Il passo 1. non è né elementare né univoco in quanto, dopo aver rotto l'uovo (cosa che già si può fare in molti modi, non tutti utili allo scopo), si deve separare il guscio dal resto.

Il passo 2. non è elementare: l'attività di cottura si può scomporre nell'accensione del fornello, nell'aggiunta del condimento e del sale, prevede il controllo della cottura, eccetera.

Se un algoritmo è veramente tale, e quindi è ben specificato, chi (o ciò che) lo esegue non ha bisogno di pensare, deve solo eseguire con precisione i passi elencati nell'algoritmo, nella sequenza in cui appaiono.

E infatti un calcolatore non pensa, ma esegue pedissequamente tutte le operazioni elencate negli algoritmi progettati (e quindi pensati) da un essere umano. Se si verifica un errore e il risultato è sbagliato, l'errore non è del calcolatore ma di chi ha progettato l'algoritmo.

Nei prossimi paragrafi approfondiremo cosa significa che i comandi di un algoritmo operano su una struttura dati e che devono terminare in un tempo finito.

1.2 Strutture dati

Per risolvere i problemi abbiamo, ovviamente, bisogno di gestire i relativi dati. A tal fine dovremo definire le opportune ***strutture dati***, che sono gli strumenti necessari per organizzare e memorizzare i dati veri e propri, semplificandone l'accesso e la modifica.

Non esiste una struttura dati che sia adeguata per ogni problema, per cui è necessario conoscere proprietà, vantaggi e svantaggi delle principali strutture dati in modo da poter scegliere di volta in volta quale sia quella più adatta (o più facilmente adattabile) al problema.

Come vedremo più avanti nel corso, il progetto o la scelta della struttura dati da adottare nella soluzione di un problema è un aspetto fondamentale per la risoluzione del problema stesso, al pari del progetto dell'algoritmo stesso. Per questa ragione, gli algoritmi e le strutture dati fondamentali vengono sempre studiati e illustrati assieme.

1.3 Efficienza

E' evidente che, affinché un algoritmo sia utilizzabile, deve concludersi e produrre il suo output entro un tempo "ragionevole". Un aspetto fondamentale che va affrontato nello studio degli algoritmi è la loro efficienza, cioè la quantificazione delle loro esigenze in termini di ***tempo*** e di ***spazio***, ossia tempo di esecuzione e quantità di memoria richiesta.

Questo perché:

- I calcolatori sono molto veloci, ma non infinitamente veloci;
- La memoria è economica e abbondante, ma non è né gratuita né illimitata.

Nel corso illustreremo il concetto di ***costo computazionale*** degli algoritmi in termini di numero di operazioni elementari e quantità di spazio di memoria necessario in funzione della dimensione dell'input.

Per comprendere l'importanza del costo computazionale di un algoritmo, in particolare per quanto riguarda il tempo di esecuzione richiesto, facciamo un semplice esempio (nota: nell'esempio, e nel resto di queste dispense, il termine *log n* denota $lg_2 n$, cioè il logaritmo in base 2 di *n*):

- Il calcolatore V (veloce) è in grado di effettuare 10^9 operazioni al secondo;
- Il calcolatore L (lento) è in grado di effettuare 10^7 operazioni al secondo;
- Il problema da risolvere è quello di ordinare $n=10^6$ numeri interi;
- L'algoritmo IS (Insertion Sort) richiede $2n^2$ operazioni;
- L'algoritmo MS (Merge Sort) richiede *50 n log n* operazioni.

Come chiariremo meglio in seguito, MS risulta più efficiente di IS. Domandiamoci se la maggiore velocità del calcolatore V riesce a bilanciare la minore efficienza dell'algoritmo IS, confrontando il tempo di esecuzione di IS sul calcolatore V con quello di MS sul calcolatore L.

Tempo di V(IS) = $\frac{2(10^6)^2 istruzioni}{10^9 istruzioni\ al\ secondo} = 2000\ secondi = 33\ minuti$

Tempo di L(MS) = $\frac{50*10^6 \log 10^6\ istruzioni}{10^7 istruzioni\ al\ secondo} = 100\ secondi = 1{,}5\ minuti$

Come si vede, la risposta è no.

Se poi supponiamo di aumentare la dimensione dell'input, portandola a 10^7 numeri interi, il divario aumenta:

Tempo di V(IS) = $\frac{2(10^7)^2 istruzioni}{10^9 istruzioni\ al\ secondo} = 2\ giorni$

Tempo di L(MS) = $\frac{50*10^7 \log 10^7\ istruzioni}{10^7 istruzioni\ al\ secondo} = 20\ minuti$

Questo ci fa capire come, indipendentemente dall'aumento di velocità dei calcolatori prodotto dagli avanzamenti tecnologici, l'efficienza degli algoritmi sia un fattore di importanza cruciale.

1.4 Problem solving e problemi computazionali

Il ***problem solving*** è un'attività per raggiungere una soluzione a partire da una situazione iniziale. Si tratta quindi di un'attività creativa, di natura essenzialmente progettuale, ed in questo risiede la sua difficoltà.

L'approccio al problem solving segue in genere i seguenti passi:

- *analisi del problema:* lettura approfondita della situazione iniziale, comprensione ed identificazione del problema;
- *esplorazione degli approcci possibili:* identificazione delle metodologie di soluzione tra i metodi noti;
- *selezione di un approccio:* scelta dell'approccio migliore;
- *definizione dell'algoritmo risolutivo:* identificazione dei dati e progettazione della sequenza di passi elementari da applicare su di essi;
- *riflessione critica:* a problema risolto, ripensamento delle fasi della soluzione proposta per identificare eventuali criticità e possibili migliorie.

I problemi oggetto del problem solving possono variare nei campi più disparati. In questo contesto, qui restringiamo la nostra attenzione ai ***problemi computazionali***, problemi cioè che richiedono di descrivere in modo automatico una specifica relazione tra un insieme di valori in ***input*** e il corrispondente insieme di valori in ***output***.

Esempio 1.1

- Problema computazionale: ordinare n numeri dal più piccolo al più grande.
- Input (anche detto ***istanza del problema***): sequenza di n numeri $a_1, a_2, \ldots, a_n$;
- Output: permutazione $a'_1, a'_2, \ldots, a'_n$ della sequenza di input tale che $a'_1 \leq a'_2, \ldots, \leq a'_n$.

Un algoritmo è ***corretto*** se, per ogni istanza di un problema computazionale, termina producendo l'output corretto. In tal caso diremo che ***l'algoritmo risolve il problema***.

Tra i vari tipi di problemi computazionali, evidenziamo i seguenti:

- *problemi di decisione:* sono quei problemi per cui la risposta ad ogni istanza può essere VERO o FALSO (esempio: dato un intero positivo *x, x* è un numero primo?)

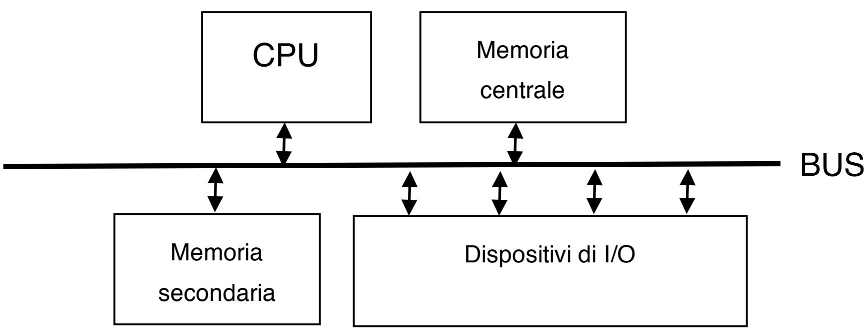
- *problemi di ricerca:* sono quei problemi per cui la risposta è più articolata di un valore booleano e presuppone che essa risponda alle richieste del problema (esempio: dato un intero positivo *x,* determinare un divisore primo di *x* non banale)
- *problemi di enumerazione:* sono quei problemi in cui si chiede di elencare TUTTE le soluzioni ammissibili (esempio: dato un intero positivo *x,* determinare tutti i divisori primi di *x*)
- *problemi di verifica:* sono quei problemi in cui l'input è costituito da un'istanza ed una possibile soluzione del problema (certificato), si chiede di verificare se il certificato è davvero una soluzione ammissibile (esempio: dato un intero positivo *x,* ed un certificato *y,* verificare che *y* sia un divisore primo di *x* non banale)
- *problemi di ottimizzazione:* sono quei problemi per cui è definita anche una funzione di costo/vantaggio e, tra tutte le soluzioni, bisogna selezionare quella con costo/vantaggio minimo/massimo (esempio: dato un intero positivo *x,* determinare il massimo divisore primo di *x*).

1.5 Modello del calcolatore

E' necessario definire un modello astratto di calcolatore per poter approfondire il problema del costo computazionale degli algoritmi.

Possiamo modellare il calcolatore come un apparato costituito di quattro tipi di unità funzionali:

- processore (CPU, Central processing Unit);
- memoria centrale (RAM, Random Access memory);
- memoria secondaria (o memoria di massa);
- dispositivi di input/output (tastiera, schermo, stampante, ecc.).



1.5.1 Memoria

Concentriamoci sulla RAM (random access memory). Essa può essere vista come una lunga sequenza di componenti elementari, ognuna delle quali contiene una unità di informazione, il ***bit*** (abbreviazione di ***binary digit***) che assume solo i valori zero e uno). Ciascuna di queste componenti è anche detta ***cella di memoria***.

Parole di memoria

Un gruppo di 8 bits è detto ***byte***. I bytes sono, a loro volta, aggregati fra loro in strutture, dette ***registri di memoria*** o ***parole di memoria***, caratterizzate dal fatto che su di esse il processore è in grado di operare (in lettura o in scrittura) con un'unica operazione.

Nei calcolatori attuali, una parola di memoria può essere costituita da 4 bytes (calcolatori a 32 bits) o 8 bytes (calcolatori a 64 bits).

Aspetti importanti sono i seguenti:

- ciascuna parola di memoria ha un ***indirizzo***;
- gli indirizzi corrispondono alla posizione dei bytes nella sequenza;
- gli indirizzi sono numeri interi e non sono memorizzati da nessuna parte, ma sono determinati dall'ordinamento consecutivo delle celle stesse;
- solo i bytes e non i singoli bits sono indirizzabili (cioè accessibili da parte della CPU).

Il termine ***random access memory*** usato per la memoria centrale convoglia un concetto molto importante: ***il tempo di accesso ad una qualunque parola di memoria è sempre lo stesso***, indipendentemente dalla posizione (e quindi dall'indirizzo) della parola. Le operazioni che il processore può effettuare su una parola di memoria sono la ***lettura*** (che preleva il contenuto corrente della parola) e la ***scrittura*** (che memorizza un contenuto nella parola, sovrascrivendo quello precedente).

Il tempo di accesso, molto ridotto, oggi è dell'ordine dei nanosecondi (10^{-9} secondi).

Gli indirizzi dei bytes sono numeri interi, quindi possono essere codificati in binario. Poiché per rappresentare un numero intero *n* è necessario un numero di bit pari a *log n* (che a sua volta è un numero intero se *n* è una potenza di 2, come è sempre il caso del numero di celle di memoria nei calcolatori), il numero di bytes esistenti (e quindi le dimensioni della RAM) determina il numero minimo di bit necessari a rappresentare gli indirizzi per poter accedere alle celle stesse.

Viceversa, il numero di bits utilizzati per rappresentare gli indirizzi di memoria determina il numero massimo di bytes indirizzabili. Questo numero viene chiamato ***spazio di indirizzamento***. Ad esempio, con indirizzi a 32 bits non si possono indirizzare più di 2^{32} celle di memoria, cioè circa quattro miliardi, nemmeno se il calcolatore ne contenesse molte di più.

La parola di memoria è l'unità massima sulla quale è possibile operare mediante un'unica operazione (istruzione).

Per eseguire un'operazione si deve specificare l'indirizzo della parola di memoria su cui si vuole operare, che coincide con l'indirizzo del primo byte facente parte della parola stessa. Di conseguenza gli indirizzi delle parole di memoria sono numeri interi multipli di 4 o di 8, a seconda dei casi. Gli indirizzi dei bytes intermedi in generale non si possono usare. Perciò, in queste situazioni, di norma il tentativo di accedere a una cella di memoria il cui indirizzo non è un multiplo di 4 (o di 8) produce un errore (ad es. "*illegal memory access*") che causa l'immediata terminazione del programma in esecuzione.

La memoria centrale, d'altro canto, è piuttosto costosa, e perde il suo contenuto quando viene a mancare l'alimentazione elettrica. Questa caratteristica si indica col termine ***volatilità***.

Memoria secondaria

Per superare il problema della volatilità si deve fare ricorso a un tipo diverso di memoria, la **memoria secondaria** (o **memoria di massa**). Essa ha le seguenti caratteristiche:

- conserva il contenuto (programmi e dati) anche in assenza di alimentazione elettrica;
- è molto più lenta della memoria centrale;
- è molto più abbondante della memoria centrale (arriva anche ai terabyte, 10^{12} byte);
- è meno costosa della memoria centrale.

1.5.2 Random access machine

Per poter valutare l'efficienza di un algoritmo è necessario analizzarlo, cioè quantificare le risorse che esso richiede per la sua esecuzione, senza che tale analisi sia influenzata da una specifica tecnologia che, inevitabilmente, col tempo diviene superata.

Esiste un modello teorico chiamato ***random access machine*** (***modello RAM***) che riflette l'organizzazione della memoria sopra descritta indipendentemente dalle caratteristiche tecniche di uno specifico calcolatore reale. La random access machine è quindi una macchina astratta, la cui validità e potenza concettuale risiede nel fatto che non diventa obsoleta con il progredire della tecnologia.

Il modello RAM ha queste caratteristiche:

- esiste un ***singolo processore***, che esegue le operazioni sequenzialmente, una dopo l'altra (non è possibile eseguire più operazioni contemporaneamente);
- esiste un insieme di ***operazioni elementari***, l'esecuzione di ciascuna delle quali richiede per definizione un ***tempo costante***. Esempi di tali operazioni sono: le operazioni artimetiche, le letture, le scritture, il salto condizionato, ecc.;
- esiste un limite alla dimensione di ogni valore memorizzato ed al numero complessivo di valori utilizzati. In particolare, il massimo valore rappresentabile in memoria non può superare 2 elevato alla dimensione della parola (32 o 64).

Criterio della misura di costo uniforme

Sia d la dimensione della parola. Se è soddisfatta l'ipotesi che ogni dato in input sia minore di 2^d, ciascuna operazione elementare sui dati del problema verrà eseguita in un tempo costante. In tal caso si parla di **misura di costo uniforme**.

Tale criterio non è sempre realistico perché, se un dato del problema è più grande, esso deve comunque essere memorizzato, ed in tal caso si useranno più parole di memoria. Di conseguenza, anche le operazioni elementari su di esso dovranno essere reiterate per tutte le parole di memoria che lo contengono, e quindi richiederanno un tempo che non è più costante.

Criterio della misura di costo logaritmico

Questo criterio, perfettamente realistico, risolve il problema sopra esposto assumendo che il costo delle operazioni elementari sia funzione della dimensione degli operandi (ossia dei dati). Poiché il numero di bits necessari per memorizzare un valore n è proporzionale a $log\ n$, si parla di **misura di costo logaritmico**.

Essa però implica rilevanti complicazioni nei calcoli dell'efficienza di un algoritmo, per cui sia in letteratura che nella pratica si sceglie di usare la misura di costo uniforme che si rivela adatta alla maggior parte dei problemi reali.

Analizziamo informalmente un semplicissimo programma applicando entrambi i criteri al fine di evidenziarne le differenze. Il programma consiste di un ciclo, reiterato n volte, che calcola il valore 2^n:

```
x ← 1;
for i = 1 to n do
     x ← x*2;
```

Con la misura di costo uniforme si vede facilmente che il tempo di esecuzione totale è proporzionale ad n, poiché si tratta di un ciclo eseguito n volte nel quale, ad ogni iterazione, si compiono due operazioni, ciascuna delle quali ha costo unitario: l'incremento del contatore e il calcolo del nuovo valore di x.

Con la misura di costo logaritmico le cose diventano subito più complicate perché sia l'incremento di i che il raddoppio di x non hanno più costo costante. In particolare, per ogni singola iterazione il costo complessivo è dato da:

- *log x = log 2^i^ = i* per il calcolo del nuovo valore di *x*;
- *log i*, per l'incremento del contatore.

Il costo totale diviene dunque:

$$\sum_{i=1}^{n}(i+\log i)$$

Ora, considerando che:

$$\sum_{i=1}^{n}(i+\log i) \geq \sum_{i=1}^{n} i = \frac{n(n+1)}{2}$$

e che:

$$\sum_{i=1}^{n}(i+\log i) \leq \sum_{i=1}^{n} 2i = n(n+1)$$

si vede che il tempo di esecuzione totale è proporzionale ad n^2.

Nel seguito, anche se non espressamente evidenziato, useremo sempre il criterio della misura di costo uniforme.