Università di Trieste

Laurea in ingegneria elettronica e informatica

Enrico Piccin - Corso di Algoritmi e Strutture dati - Prof. Andrea Sgarro

Anno Accademico 2021/2022 - 2 Marzo 2022

Indice

2 Marzo 2022

1 Introduzione

Algoritmo è una parola molto antica, non connessa all'utilizzo e all'invenzione del calcolatore.
Alla base della teoria della computazione si pone il **Liber abaci** (tradotto "Libro della computazione"), scritto nel 1200 da Leonardo Bonacci, in contatto con la popolazione araba, in quanto mercante; egli è venuto a conoscenza della **numerazione araba**, introducendola in Occidente e spiegandola dettagliatamente all'interno del **Liber abaci**.
La numerazione araba è uno straordinario passo in avanti nella scienza, in quanto con essa viene introdotto il concetto di **notazione posizionale**, così come l'importanza del numero 0: i numeri non servono solamente per contare, come si pensava in precedenza, e per questo rinnegando il numero 0.
A Firenze, sempre negli stessi anni, ci fu una **protesta sindacale** contro l'innovazione tecnologica, contro questa nuova scoperta, facendo pressione affinché il governo abolisse il nuovo sistema di numerazione, in quanto avrebbe fatto perdere il posto di lavoro a tutti coloro che prima eseguivano difficili calcoli con la numerazione romana: tuttavia, tale proteste, com'é noto, possono rallentare il progresso, ma mai arrestarlo.
Leonardo Bonacci, nei suoi viaggi in Oriente, venne a conoscenza del **Liber abaci** di Al-Gorasmy, proveniente dalla Coresmia, ma che parlava persiano, da cui poi sarebbe stato tratto il nome **Algoritmo**, che letteralmente significa **procedimento di calcolo**.

ALGORITMO

Algoritmo significa letteralmente **procedimento di calcolo**. Tuttavia, bisogna chiarire che un algoritmo è un procedimento di calcolo non necessariamente numerico, ma molto più generale, che va ben al di là dei numeri.
Un altro importante elemento che contraddistingue l'algoritmo è la **meccanicità**, ovvero la sua esecuzione può essere affidata ad una macchina: ciò significa che un algoritmo non deve necessariamente essere meccanizzato, ma deve essere **meccanizzabile**; in altre parole, l'esecuzione (bada bene, l'esecuzione e non la sua ideazione) dell'algoritmo è completamente **stupida**.

Esempio 1: L'algoritmo della moltiplicazione è molto chiaro e semplice: basta solamente accedere ad una **base di dati** in cui sono memorizzati i prodotti elementari fra numeri molto piccoli, e quindi molto più semplici da trattare.
Una volta eseguite le operazioni di moltiplicazione tramite quanto esposto in precedenza, è necessario eseguire delle operazioni di addizione, che era necessario aver precedentemente memorizzato.
Ecco che quello che si è appena eseguito è un **procedimento di calcolo**.

Esempio 2: L'algoritmo della divisione fa sempre uso di una base di dati, nella quale devono essere memorizzati i risultati del prodotto del divisore con tutti i numeri decimali da 0 a 9 e confrontare ciascun prodotto con il termine da dividere per ottenere il quoziente.
Alternativamente, si sarebbe potuto creare un ciclo da 0 a 9 in cui per ogni indice si sarebbe dovuto verificare se questo fosse il fattore moltiplicativo corretto per ottenere la quantità giusta da sottrarre.

Osservazione: In ciascuno di tali esempi è essenziale la meccanicità del processo esecutivo, che appare evidente.

Quando si rappresentano delle quantità e, a maggior ragione, quando si effettuano dei calcoli, è fondamentale fissare una base di rappresentazione, da cui poi dipendono le cifre che si possono impiegare per la rappresentazione stessa.
La notazione posizionale permette anche di comprendere la rappresentazione di qualsiasi quantità con qualsiasi base, effettuando anche delle conversioni di base a seconda della maggiore o minore convenienza di rappresentazione.

Per esempio, volendo convertire una quantità rappresentata in base $\mathcal{B} = 7$ in una base $\mathcal{C} = 10$ si deve procedere come segue

$$(5203)_7 = 5 \cdot 7^3 + 2 \cdot 7^2 + 0 \cdot 7^1 + 3 \cdot 7^0 = 1715 + 98 + 0 + 3 = (1816)_{10}$$

Ovviamente le basi di rappresentazione sono almeno binarie, in quanto la **base unaria** non può, per ovvie ragioni, rappresentare alcuna quantità se non quella unica che viene permessa dalla base scelta, ossia lo 0.
Tuttavia, il processo inverso, atto a passare dalla rappresentazione di una quantità in base 10 ad una in base 3, non risulta essere così immediato.
Per cercare un algoritmmo che permette di effettuare tale conversione, si effettua un primo **passaggio controintuitivo** (che suggerisce, tuttavia, il corretto processo esecutivo), che prevede di rappresentare una quantità in base 10 in una quantità ancora in base 10, tramite un processo di divisioni successive. Si consideri, a tal proposito

$$(3412)_{10}$$

e si divida progessivamente tale numero per 10, come segue

3412	10
341	2
34	1
3	4
0	3

Leggendo, ora, i resti, al contrario si ottiene il numero cercato all'inizio. Se ora si prova a considerare un'altra base, come 3, l'operazione porta ad un risultato analogo

3412	3
1137	1
379	0
126	1
42	0
14	0
4	2
1	1
0	1

Per cui si è ottenuto

$$(3412)_{10} = (11200101)_3$$

Scegliendo la base 2 si ottiene, per esempio

241	2
120	1
60	0
30	0
15	0
7	1
3	1
1	1
0	1

Per cui si è ottenuto

$$(241)_{10} = (11110001)_2$$

Ovviamente la lunghezza di rappresentazione in base 2 prevede un numero di cifre pari a circa il triplo di quelle impiegate per rappresentare la medesima quantità in base 10, proprio perché

$$\log_2(10) \cong 3.3$$

Per passare da base 10 a base 100, le operazioni sono molto semplici

$$(375712)_{10} = [(37)(57)(12)]_{100}$$

usando come simboli

$$(00), (01), ..., (75), ..., (99)$$

Si consideri, ora la base 8 e si scriva un numero binario in base ottale:

$$(010101010)_2 = [(010)(101)(010)]_8 = (252)_8$$

Ancora una volta, le cifre impiegate per la rappresentazione sono state ridotte ad un terzo, sempre perché

$$\log_2(8) = 3$$

E se ora si volesse impiegare la base 16 si otterrebbe:

$$(010101010)_2 = [(1010)(1010)]_{16} = (\mathrm{AA})_{16}$$

Osservazione: Si consideri una lunghezza $l = 5$. Allora usando 5 cifre, non tutte nulle, in base 10, i numeri n che si possono rappresentare sono

$$10000 \leq n \leq 99999 \quad \equiv \quad 10^4 \leq n < 10^5 \quad \equiv \quad 10^{l-1} \leq n < 10^l$$

Da ciò si può estrapolare un risultato importante

$$\log_{10}\left(10^{l-1}\right) \leq \log_{10}(n) < \log_{10}\left(10^l\right) \quad \equiv \quad l - 1 \leq \log_{10}(n) < l$$

che è una relazione esatta. Tuttavia, approssimativamente, si può scrivere che

$$l_{10}(n) \cong \log_{10}(n)$$

3 Marzo 2022

Com'è noto, il matematico indiano **Ramanujan** ha affermato che la matematica esatta non rappresenta una base solida per la realtà, mentre la matematica vera è fatta di approssimazioni.
Hardy scoprì quanto fosse importante lo studio di **Ramanujan** e insieme a lui portò avanti la teoria dei numeri, una teoria **asintotica** che, come lui stesso affermava, non può essere esatta, ma fatta di approssimazioni.
Se, per esempio, si considera una quantità scritta in base 5, quale $n = (412)_5$

$$(412)_5 = 4 \cdot 5^2 + 1 \cdot 5^1 + 2 \cdot 5^0 = (107)_{10}$$

Se, ora, si fissa una lunghezza $l = 4$, una lunghezza rigida, senza considerare zeri in testa, si può capire che con 4 cifre si possono rappresentare, in base 5 numeri n nell'intervallo

$$1000 \leq n < 10000$$

ovvero tale per cui

$$5^{l-1} \leq n < 5^l$$

e ciò funziona con qualsiasi base, per cui, in generale, fissata una lunghezza l e una base $\mathcal{B}$ si ha che le quantità che possono essere rappresentate con l cifre, non tutte uguali a 0 è

$$\mathcal{B}^{l-1} \leq n < \mathcal{B}^l$$

Traducendo tale risultato tramite il logaritmo in base $\mathcal{B}$, sfruttando la crescenza in senso stretto della funzione logaritmica, si ottiene, equivalentemente

$$l - 1 \leq \log_{\mathcal{B}}(n) < l$$

in cui, ovviamente,

$$\log_{\mathcal{B}}(n) < l \leq \log_{\mathcal{B}}(n) + 1$$

che si può scrivere che

$$l_{\mathcal{B}}(n) \cong \log_{\mathcal{B}}(n)$$

Per cui l'**errore massimo** che si può commettere è di 1 cifra in base $\mathcal{B}$, nel caso peggiore, ma sarà sempre un po' maggiore del $\log_{\mathcal{B}}(n)$, per cui il logaritmo è una **sottostima della lunghezza**.
Tuttavia, nello spirito di Hardy, sarà utile anche scrivere che la lunghezza binaria di n è circa uguale al logaritmo binario di n, ovvero

$$\log_{\mathcal{B}}(n) \cong \log_{\mathcal{B}}(n)$$

in cui si può interpetare il $\log_{\mathcal{B}}(n)$ come una **lunghezza analogica**, mentre $l_{\mathcal{B}}(n)$ è una **lunghezza digitale**, in quanto **intera**, con precisione alla cifra (senza nulla in mezzo): in molti casi sarà più utile la lunghezza analogica di quella digitale, in quanto molto più precisa.
Grazie a questa formula è possibile capire facilmente come si alterano le lunghezze quando si effettua un cambiamento di base. Per esempio, si può osservare che

$$\log_2(10) \cong 3.38$$

per cui la lunghezza in base 2 è circa tre volte la lunghezza in base 10.

Esempio: Per trasformare un numero da base 10 in base 5 si deve procedere per divisioni successive per 5, considerando i resti (per questo si parla di *divisione intera*). Per esempio si ha che

$$10 \div 3 = 3 \text{ con resto di } 1$$

in cui, ovviamente, il resto r può essere

$$0 \leq r < D$$

con D divisore. Convertendo 32 da base 10 a base 5 ci si aspetta di ottenere un resto $0 \leq r \leq 4$.

1.1 Architettura dei calcolatori

Il calcolatore, naturalmente, si basa sulla logica binaria, ovvero opera impiegando la rappresentazione in base 2.
Il metodo più utilizzato per rappresentare caratteri diversi da quelli binari, tramite una codifica binaria, è il metodo ASCII (dall'inglese, American Standard Code For Information Interchange). Naturalmente, siccome la codifica tramite ASCII fa uso di soli 7 bit (sarebbero 8, ma un bit è riservato alla parità, per la rilevazione degli errori), il numero di n-uple binarie che si possono ottenere è 2^7, un numero certamente irrisorio per la rappresentazione di tutti i caratteri alfanumerici necessari per la comunicazione multilinugua.
In generale, fissata una lunghezza n, il numero di n-uple binarie distinte è, ovviamente, 2^n, che rappresenta una crescita esponenziale, praticamente infinita, anche se, ovviamente, in teoria sono un numero ben limitato.

Esempio: Si consideri una macchina calcolatrice che considera delle istruzioni di 9 bit come di seguito esposto:

NOME ISTRUZIONE	ISTRUZIONE
ADD	$010 \times \times \times \times \times \times$
PUNCH	$100 \times \times \times \times \times \times$

Tabella 1: Tabella di istruzioni operative per un calcolatore

Naturalmente, tale linguaggio è **Assembly**, ovvero un linguaggio molto simile al linguaggio macchina, che risulta particolarmente complesso da impiegare per lo sviluppo di software.

1.2 Diagramma di flusso

Si consideri il seguente **flowchart**, o **diagramma di flusso**:

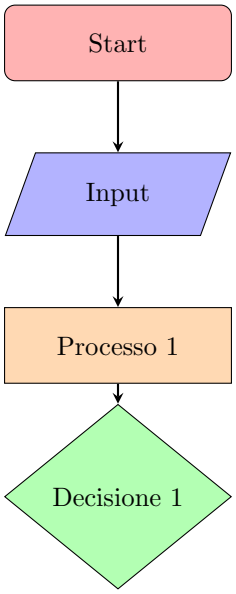

Figura 1: Diagramma di flusso

Tuttavia, tale tecnica di progettazione algoritmica è oramai superata, lasciando il posto allo **pseudocodice**, ossia un linguaggio di definizione delle istruzioni slegato da qualsiasi specifico linguaggio di programmazione di riferimento, che permette di esporre una serie di istruzioni esecutive molto simili a quelle di un programma vero e proprio.

2 Ordinamento (sorting)

Si espongono, di seguito, i principi di algoritmica dei più importanti algoritmi di ordinamento. Ciascuno di tali algoritmi prevede di effettuare l'ordinamento di n numeri forniti come input, in modo debolmente crescente, in caso di uguaglianza.

2.1 Bubble-Sort

Si espone di seguito l'algoritmo di ordinamento **bubble-sort** impiegando lo *pseudocodice*:

Algorithm 1 Bubble-sort

```
1: do the following n − 1 times
2:     point to the 1st element
3:     do the following n − 1 times
4:         compare with next
5:         if wrong order exchange
6:         point to the next
```

Naturalmente tale algoritmo è corretto e lo si può verificare immediatamente, considerando, per esempio, i seguenti 5 elementi, così ordinati:

2	3	1	4	5

Ovviamente il procedimento ci porta ad eseguire l'algoritmo 4 volte. Nella prima iterazione si ottiene

2	1	3	4	5

la seconda iterazione, invece, porta ad ottenere

1	2	3	4	5

mentre le ultime due iterazioni sono superflue. Si capisce facilmente che tale algoritmo non risulta essere pienamente efficiente, in quante alcune iterazioni potrebbero essere evitate, tramite un **flag**, per esempio. Allo stato attuale, il numero delle iterazioni da eseguire è

$$\#\text{iterazioni} = (n-1) \cdot (n-1)$$

Se, invece, si facesse in modo di evitare alcune iterazioni si avrebbe un numero di iterazioni

$$(n-1) \leq \#\text{iterazioni} \leq (n-1)^2$$

considerando $n - 1 \cong n$, e quindi $(n-1)^2 \cong n^2$ si può dire che la complessità del bubble-sort è **quadratica**, in quanto il numero delle iterazioni è n^2.

4 Marzo 2022

L'algoritmo bubble-sort non viene utilizzato, ad oggi, così come non si impiega lo pseudocodice in *pseudo-english* (da leggere psude-inglish). Esso è funzionale, ma non efficiente, in quanto la sua complessità è n^2.
Ecco che per definire un algoritmo di ordinamento non è necessaria solamente la sua funzionalità, ma anche l'efficienza.

2.2 Insertion-sort

L'insertion-sort è un algoritmo di ordinamento che prevede di considerare ciascuna quantità da ordinare ad una ad una e di effettuare un confronto solo quando ci sono dei cambiamenti.
La prima quantità è ovviamente già in ordine con se stessa. Se la seconda è più piccola della prima, si effettua uno scambio, per cui ora i primi due numeri sono ordinati. Si considera, ora, il terzo numero e se questo è più piccolo del secondo si effettua uno scambio e un nuovo confronto tra la seconda e la prima e così via.
Pertanto si effettuano tutti i confronti solamente quando si ha uno scambio delle quantità: questo comporta che il minimo numero di iterazioni è $n-1$, se n è il numero delle quantità da ordinare. Se, invece, tutte le quantità sono in disordine si effettua un numeri di iterazioni pari

$$1 + 2 + ... + (n-2) + (n-1) = \frac{n \cdot (n+1)}{2}$$

una formula molto semplice che Gauss determinò come segue, ovverosia scrivendo la somma dei numeri da 1 a k in ordine crescente e poi decrescente, come mostrato di seguito:

$$\begin{array}{ccccccccccc} 1 & + & 2 & + & 3 & + & ... & + & k-1 & + & k \\ k & + & k-1 & + & k-2 & + & ... & + & 2 & + & 1 \end{array}$$

essendo k numeri in ambedue le righe, sommando i due termini corrispondenti, uno sotto l'altro, si ottiene sempre $k+1$ che, sommato per k volte produce $k \cdot (k+1)$. Tuttavia, dal momento che tale quantità è il doppio di quella richiesta si ottiene

$$\frac{k \cdot (k+1)}{2}$$

Pertanto si ha che il numero di iterazioni dell'algoritmo *insertion-sort* è

$$n \cong n-1 \leq \#\text{iterazioni} \leq \frac{n \cdot (n-1)}{2} \cong n^2$$

che, in maniera approssimata è

$$n \leq \#\text{iterazioni} \leq n^2$$

pertanto, nel caso migliore, la **complessità è lineare**, mentre nel caso peggiore, la **complessità è quadratica**.

2.3 Pseudocodice

Per la scrittura dello pseudocodice si devono impiegare delle notazioni e dei simboli ben specifici, che di seguito vengono riportati.

2.3.1 Assegnazione

L'**assegnazione** viene indicata con il simbolo = (oppure := o $\leftarrow$), anche se l'assegnazione non è un'uguaglianza. Per esempio, la notazione

$$A = 3$$

significa che nella cella di memoria A viene inserito il valore 3. Analogamente, se si scrive

$$A = A + 1$$

significa che il valore presente nella cella di memoria A viene incrementato di 1 unità.

2.3.2 Condizione

La specifica della **condizione** avviene tramite l'istruzione **if**, secondo la notazione seguente:

if C **then**
    istruzioni
else
    istruzioni

2.3.3 Ciclo for

Il **ciclo for** è un'istruzione di ciclo in cui vengono indicate specificatamente le iterazioni che devono essere eseguite, secondo la notazione seguente

for $i = 0$ **to** n **do**

2.3.4 Ciclo while

Il **ciclo while** è un'istruzione di ciclo in cui si effettuano le istruzioni fintantoché la condizione specifcata è vera

while C **do**

Si consideri lo pseudocodice dell'algoritmo **INSERTION-SORT(A)**, esposto di seguito, dove **A** sta ad indicare **array**, ovvero un record di $length[A] = n$ valori da ordinare, già forniti in input. Di seguito si espone lo pseuodocodice, in cui si parte da 1 come posizione iniziale dell'array:

Algorithm 2 Insertion-sort

```
1: for j = 2 to length[A] = n
2:     do  key = A[j]
3:         ...
4:         i = j − 1
5:     while i > 0 ∧ A[i] > key
6:         do  A[i + 1] = A[i]
7:             i = i − 1
8:             A[i + 1] = key
```

Tale codice é concluso e il suo funzionamento può essere facilmente verificato come segue, considerando l'array A di lunghezza $length[A] = 6$:

5	2	4	6	1	3

Partendo con $j = 2$ si fissa $key = A[j] = 2$ e imponendo $i = 1$, si entra all'interno del ciclo *while*, in quanto $i > 0$ e $A[i] > key$ e si effettua l'istruzione $A[i + 1] = A[i]$ e $A[i + 1] = key$, trovandosi nella configurazione seguente

2	5	4	6	1	3

Terminato il ciclo while e il ciclo for si procede con $j = 3$ si fissa $key = A[j] = 4$ e imponendo $i = 2$, si entra all'interno del ciclo *while*, in quanto $i > 0$ e $A[i] > key$ e si effettua l'istruzione $A[i + 1] = A[i]$ e $A[i + 1] = key$, trovandosi nella configurazione seguente

2	4	5	6	1	3

Adesso, terminato il ciclo while e for, si considera $j = 4$, specificando $key = A[j] = 6$ e $i = 3$. In questo caso, tuttavia, non si entra nel ciclo while, in quanto $i > 0$, ma $A[i] < key$. Si procede direttamene con $j = 5$, $key = A[j] = 1$ e $i = 4$ e si entra nel ciclo while, compiendo tutte le iterazioni

2	4	5	1	6	3
2	4	1	5	6	3
2	1	4	5	6	3
1	2	4	5	6	3

e così via fino ad arrivare all'ordinamento finale

1	2	3	4	5	6

Ecco che, come si può vedere, tale algoritmo ha una complessità che, nel caso migliore, è **lineare** (n) e nel caso peggiore è **quadratica** (n^2).
Mentre si chiama **complessità tipica** la **complessità media**, ovvero la complessità dell'algoritmo nel caso intermedio. In questo caso la complessità tipica è n^2, che può essere calcolata, in maniera non propriamente corretta, come segue

$$\frac{n + n^2}{2} = n^2$$

Esiste, infine, anche una **complessità empirica**, basata sull'uso pratico dell'algoritmo: in particolare, l'algoritmo insertion-sort funziona particolarmente bene quando il **numero degli elementi da ordinare è ridicolmente basso**, il che potrebbe essere un controsenso; tuttavia, potrebbe essere particolarmente utile ricorrere all'ordinamento di pochi numeri all'interno di una procedura particolarmente complessa: ecco, allora, che l'utilizzo di insertion-sort diviene conveniente (cosa che non accade per bubble-sort).

Osservazione: È importante osservare che l'algoritmo di insertion-sort è un **algoritmo di ordinamento in loco**, ovvero tale per cui non si impiega un altro array per l'ordinamento, ma tutte le operazioni si effettuano sullo stesso array di partenza.

Osservazione: Quando si parla di algoritmica, non è possibile parlare di **completezza** senza parlare di **complessità**.

3 Grafi

Il **grafo** è una **struttura finita**. Gli elementi costituitvi di un grafo sono i **vertici** (o **nodi**) e gli **archi** (o **lati**) (dall'inglese *arcs* o *edges*). Per indicare i vertici si impiega la lettera v, mentre per indicare gli archi si usa la lettera ξ.
Un arco collega due vertici distinti che, per il momento, non è orientato, non rappresenta una freccia, in quanto si parla di **grafi semplici**, come illustrato di seguito:

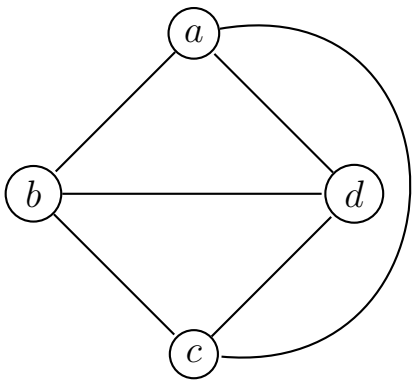

Figura 2: Esempio di grafo semplice

Il calcolo della copertura dei vertici prende il nome di **vertex cover** e prevede di definire il numero minimo di vertici essenziali per coprire tutti gli archi. L'ottimizzazione del grafo, in questo caso, prevede di determinare la copertura minima.
Si supponga di avere a disposizione k vertici (che si indica come $|v| = k$, in cui $|v|$ rappresenta la **cardinalità** dell'insieme dei vertici). Naturalmente, la copertura minima pari a 0 si ha quando i vertici del grafo sono tutti scollegati, ovvero non ci sono archi.
Il numero di archi in un grafo completo è pari a

$$\frac{|v| \cdot (|v| - 1)}{2}$$

per cui si potrebbe, in questo caso limite, affermare che la copertura minima sia k, ma se si elimina un nodo ancora la copertura sussiste, in quanto su ogni arco vi sarà sempre almeno un nodo coperto. Quindi si può affermare che la *vertex cover*, nel caso generale è compresa tra 0 e $k - 1$, con k numero dei vertici.

9 Marzo 2022

Il problema del **vertex cover**, ovvero di "ricoprimento dei vertici", è un proplema che riguarda la teoria dei grafi.
Gli elementi costituitvi di un grafo sono i **vertici** (o **nodi**, molto più raramente chiamati *punti*) e gli **archi** (dall'inglese *edges*, traducibili in **spigoli** o, più impropriamente, in *lati*), per il momento non orientati, che collegano due vertici, per il momento necessariamente distinti.
La notazione per indicare vertici e archi è la seguente

- L'insieme dei vertici si denota con v
- L'insieme degli archi si denota con ξ

Naturalmente sussiste la possibilità che in un grafo tutti i vertici siano sconnessi ed **isolati**, ovvero il numero degli archi è nullo, per cui si ottiene che $|\xi| = 0$.
Analogamente, volendo collegare tutti i nodi con un arco (ottenendo un **grafo completo**), si procede come seugue: partendo da un primo vertice se ne collega un secondo, necessariamente distinto; ma non volendo considerare ogni arco due volte, si divide per due, ottenendo

$$\frac{|v| \cdot (|v| - 1)}{2} \cong |v|^2$$

ottenendo

$$0 \leq |\xi| \leq \frac{|v| \cdot (|v| - 1)}{2}$$

considerando $|v|$ è la cardinalità, ossia il numero dei vertici considerati.
Il problema di **vertex cover** è un problema di ottimizzazione: ridurre il numero minimo di vertici tale per cui nel grafo ad ogni arco deve essere collegato almeno un vertice coperto. Per esempio, in un grafo dove ogni nodo è isolato, il numero minimo dei vertici è 0, in quanto non ci sono archi.
Nel caso di un **grafo completo**, come quello esposto di seguito

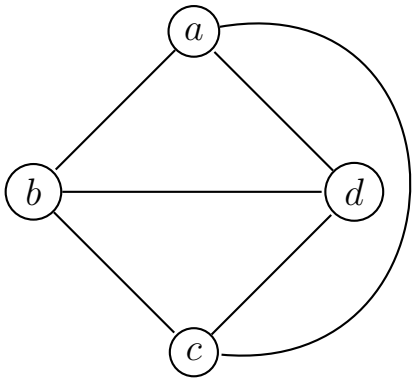

Figura 3: Esempio di grafo semplice completo

In questo caso la copertura minima non è pari a $|v|$, in quanto eliminando uno qualsiasi dei nodi ancora ad ogni arco sarà collegato almeno un nodo comperto. Pertanto si ha che

$$0 \leq \#nodi \leq |v| - 1$$

Per la risoluzione del **vertex cover** vi sono due algoritmi, di cui solo il secondo realmente efficace.
Si consideri il grafo seguete

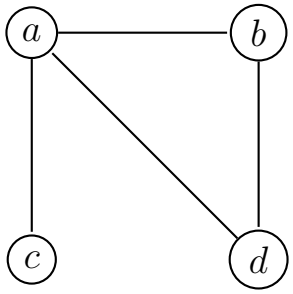

Figura 4: Esempio di grafo semplice

In cui, naturalmente, per coprire tutti gli archi sarà sufficientte considerare un poliziotto in a e uno in d (oppure in b), ottenendo

$\boxed{a}$	$\boxed{b}$	$\boxed{c}$	$\boxed{d}$
$\boxed{1}$	$\boxed{0}$	$\boxed{0}$	$\boxed{1}$

in cui con $\boxed{1}$ rappresenta la copertura del rispettivo vertice. Ecco che questa è una k-upla binaria di **peso** $w = 1+1 = 2$: un procedimento per verificare la corretta copertura prevede di considerare tutti gli archi e verificare per ciascuno che almeno ad un vertice collegato corrisponda 1; se un arco è collegato a due vertici 0, la copertura è scorretta.
Tuttavia,, la k-upla 1001 è anche una codifica binaria su 4 bit del numero $(9)_{10}$, che suggerisce la procedura di controllo seguente, definita a partire da k numero di vertici

Algorithm 3 Vertex-cover

```
1: for i = 0 to 2^k
2:     i → binario
3:     check(i)
```

In cui viene ottenuta, alla fine, la copertura di peso minore. Per eliminare alcune iterazioni, si potrebbe procedere

Algorithm 4 Vertex-cover

```
1: for w = 0 to k − 1
2:     for all
```

In cui, tuttavia, nel caso peggiore, si potrebbe procedere ad effettuare un numero di iterazioni pari a $2^k - 1$, che non è molto dissimile dal caso precedente, in cui si facevano inevitabilmente 2^k iterazioni.
Questo algoritmo, pertanto, pur essendo corretto, è ispirato al meccanismo dell'**exaustive search** (dall'inglese, ricerca esauriente), che prevede di controllare tutti gli archi al fine di verificarne la corretta copertura.
Il motivo per cui tale algoritmo è inutilizzabile è che presenta un numero di **iterazioni esponenziale** e, conseguentemente, una **complessità esponenziale**, la quale è intollerabile, dal momento che il numero delle iterazioni cresce esponenzialmente al variare dell'imput.
Di seguito si espone, invece, un nuovo algoritmo che risolve il problema del **vertex cover**; si considerino due nodi connessi da un arco e si eliminino tutti gli archi che incidono sul primo e sul secondo vertice e così via, fino ad esaurire tutti gli archi a disposizione: alla fine si ottiene un ricoprimento vero e proprio. Come di consueto, l'input non viene specificato a priori, ma il numero delle iterazioni per specificare l'input è $\cong |v| + |\xi|$. Lo pseudocodice si espone di seguito

Algorithm 5 Vertex-cover

```
C ← ∅
E' ← ξ(G)
while E' ≠ ∅
    do ... (u, v) in E
        C ← C ∪ {u, b}
        delete incidienti
return C
```

In cui il numero di iterazioni in cui tale algoritmo si impegna è, approssimativamente, pari a $\cong |v| + |\xi|$, ovvero si ha una **complesità lineare**: questa è un'ottima notizia, in quanto significa che tale algoritmo è velocissimo. L'unico problema è che esso è scorretto; si consideri il seguente grafo semplice:

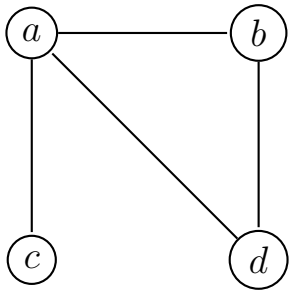

Figura 5: Esempio di scorrettezza dell'agoritmmo considerato

In questo caso, l'algoritmo, considerando per primo l'arco $a - b$, elimina tutti gli altri archi e produce un risultato corretto.
Tuttavia, se il primo arco considerato dall'algoritmo fosse $a - c$, l'algoritmo è costretto a considerare anche l'arco $b - d$, non producendo un risultato corretto.
Tale algoritmo prende il nome di **algoritmo approssimato**, in quanto non sempre produce un risultato corretto, ma mai disastroso.
Tuttavia, non esiste un algoritmo che sia corretto e anche accettabile dal punto di vista del numero delle iterazioni e quindi della complessità: pertanto ci si deve accontentare dell'algoritmo approssimato appena esposto.
Si assuma che il numero di nodi ottimali considerati sia $|\mathcal{O}|$ e il numero degli nodi effettivamente ottenuti sia $|\mathcal{P}|$, legati dalla seguente relazione

$$|\mathcal{O}| \leq |\mathcal{P}|$$

Ovviamente, per quanto visto con l'esempio precedente, tale disuguaglianza può anche essere stretta. Ovviamente, se ora si considerano gli archi privilegiati dall'algoritmo esposto $|\mathcal{A}|$ è

$$|\mathcal{A}| = \frac{|\mathcal{P}|}{2}$$

in quanto su ogni arco vi sono due nodi, per cui basta considerarne la metà Ora, il numero di nodi ottimale $|\mathcal{O}|$ è legato alla seguente relazione

$$|\mathcal{O}| \geq |\mathcal{A}|$$

per cui si ottiene che

$$\frac{|\mathcal{P}|}{2} \leq |\mathcal{O}| \leq |\mathcal{P}|$$

4 Algoritmi aritmetici

L'**algoritmo di Euclide** permette di calcolare il **Massimo Comune Divisore** (**M.C.D.**) tra due numeri, ma non procedendo alla fattorizzazione in fattori primi tra le due quantità considerate. Per esempio

$$12 = 3 \cdot 2 \cdot 2$$
$$12 = 3 \cdot 3$$

da cui si evince che

$$\text{MCD}(12, 9) = (12, 9) = 3$$

Tuttavia, non è possibile procedere attraverso la fattorizzazione per la risoluzione di tale problema, in quanto gli algoritmi per la fattorizzazione sono estremamente lenti. L'algoritmo di Euclide, invece, risolve tale problematica in maniera corretta, veloce ed efficiente, basandosi sul meccanismo della **ricorsività**.

La divisione può essere di due tipologie

1. Divisione esatta: $10 \div 3 = 3, \overline{3}$
2. Divisione intera: $10 \div 3 = 3$ con resto 1

10 Marzo 2022

Naturalmente, un algortimo non può essere applicato concretamente se ha una crescita esponenziale: esso è inutilizzabile, in quanto il tempo di risposta è troppo elevato per avere un impiego pratico.
La tabella seguente di Garied Jhonson, ne dà una fondamentale prova pratica, considerando un calcolatore le cui istruzioni durano $0,000001$ s per essere processate; naturalmente l'algoritmo considera input variabili, di lunghezza 10, 20, 30, 40, 50 e 60 e si supponga che la lunghezza dell'input determini in modo quanto più preciso e linearmente dipendente il numero delle operazioni che devono essere eseguite: se l'input ha lunghezza 60 significa che sono necessarie 60 operazione per terminare l'algoritmo e quindi $0,00006$ s.
Se, invece, la complessità dell'algoritmo è quadratica, allora ciò significa che se l'input è di lunghezza 60, allora il numero di operazioni sono $60 \cdot 60 = 3600$ e quindi il numero di secondi diviene $0,0036$ s.
Se la complessità è cubica, allora con lunghezza dell'input di 60 il numero di operazioni diviene $60 \cdot 60 \cdot 60$ e quindi il tempo impiegato è di $0,216$ s.
Con una complessità quintica, a 60 di lunghezza corrispondono 60^5 istruzioni e quindi 13 minuti di esecuzione.
Passando ad una complessità esponenziale, come 2^n, con lunghezza dell'imput pari a 60 si hanno 2^60 istruzioni e quindi un tempo esecutivo di 360 secoli. Passando appena a 3^n, con lunghezza dell'imput pari a 60 si hanno 3^60 istruzioni e quindi un tempo esecutivo di $1,3 \cdot 10^{13}$.

Osservazione: Si osservi che considerare un calcolatore 1000 volte più veloce può essere significativo se la complessità dell'algoritmo è polinomiale, ma è totalmente ininfluente se la complessità è esponenziale.

4.1 Algoritmo di Euclide

Si consideri il seguente algoritmo, noto come **algoritmo di euclide**, estremamente fulmineo:

Algorithm 6 Euclide

1: **begin**
2: $a, b = m, n$
3: **while** $b \neq 0$ **do** $a, b = b, a \bmod b$
4: mcd $= a$
5: **end**

In questo caso l'algoritmo considera come input **due numeri interi** m e n di cui ha senso determinare l'**m.c.d**, necessariamente interi, in quanto non si ha un controllo sintattico. Generalmente si considerano $m < n$, ma ciò è ininfluente, in quanto il programma provvede ad effettuare un cambiamento del loro ordine in automatico.
Nell'algoritmo vi sono delle assegnazioni composte, in cui

$$a, b = m, n$$

ovvero ad a si assegna il valore m, mentre a b si assegna il valore m. Così come in seguito si ha

$$a, b = b, a \bmod b$$

ovvero ad a si assegna il vecchio valore b, mentre a b si assegna il resto della divisione intera tra a e b. Alla fine del ciclo si ottiene che l'**m.c.d.** cercato è proprio a.
Si consideri, a tal proposito, il seguente esempio:

$$\boxed{12}\ \boxed{9}$$

Dopo il primo passo si ottiene

$$\boxed{9}\ \boxed{3}$$

ed infine

$$\boxed{3}\ \boxed{0}$$

ecco che nella prima cella si ha proprio l'm.c.d. cercato. Ora, tuttavia, bisogna verificare se tale algoritmo sia effettivamente corretto e che quella considerata non sia solo una combinazione; inoltre, per determinare la complessità dell'algoritmo è necessario considerare, essenzialmente, il numero di **iterazioni libere** del ciclo while, dal quale dipende la complessità dell'algoritmo.

Osservazione: Si osservi, innanzitutto, che il ciclo while si chiude visto che in posizione b sarà presente un resto, ovvero una quantità intera che progressivamente diminuisce fino a diventare 0. L'ultimo passaggio, naturalmente, è il più semplice, in quanto l'algoritmo, nell'ultima iterazione, ci si ritroverà sempre nella situazione desiderata:

$$\boxed{\alpha \cdot \text{m.c.d.}}\ \boxed{\text{m.c.d.}}$$

per cui nell'ultima iterazione si ha proprio l'm.c.d. che si sta cercando.
Per la dimallostrazione della correttezza dell'algoritmo, anche nelle fasi intermedie, si deve dimostrare essenzialmente che, a qualunque fase del processo esecutivo

$$\text{M.C.D.}(a, b) = \text{M.C.D.}(b, a \bmod b)$$

Per farlo, si parta dal caso inziale in cui $a = \alpha a'$ e $b = \alpha b'$, tale per cui

$$a = qb + r$$

in cui, ovviamente, si ha che

$$r = a - qb = \alpha \cdot (a' - qb')$$

in cui, ora

$$a = b = \alpha b' \quad \text{e} \quad b = r = \alpha \cdot (a' - qb')$$

... continua ...

Osservazione: Euclide era un alessandrino. Lamé, nel 1844, è da considerarsi il padre della **teoria della complessità**, mentre Reynaud, nel 1811 lo aveva già preceduto, ma si parla sempre della prima metà dell'800.
Si consideri il numero $n = 37855$, il quale viene progressivammente ridotto di **almeno** 10 volte ad ogni iterazione, portandolo progressivamente a 3785, 378, 37, 3 ed infine 0.
Pertanto, al più, il numero di iterazioni necessarie per annientare questo valore è pari alla lunghezza decimale del numero n (in questo caso pari a 5) che, con una buona dose di approssimazione può essere considerata

$$l_{10}(n) \cong \log_{10}(n)$$

Se, ora, il numero n considerato è scritto in binario e ad ogni iterazione viene annientato della metà, al più serviranno un numero di iterazioni pari alla lunghezza binaria del numero n considerato per annientarlo, ovvero

$$\#\text{iterazioni} \leq l_2(n) \cong \log_2(n)$$

Consierando l'algoritmo di Euclide e procedendo con i calcoli di **Lamé**, si considerino tre iterazioni del ciclo while, che producono i seguenti tre stati

$$\boxed{a}\ \boxed{b}$$

$$\boxed{b}\ \boxed{c}$$

$$\boxed{c}\ \boxed{d}$$

in cui, ovviamente,

$$b = qc + d$$

A parte il caso in cui i due numeri $a \leq b$, cui l'algoritmo provvede cambiandoli d'ordine, si ha sempre che $a > b$ e, quindi, il quoziente della divisione tra i due numeri è sempre almeno 1, quindi

$$b = qc + d \geq c + d \geq 2d$$

Pertanto, il numero di passi doppi é circa uguale al logaritmo in base 2 di n, quindi

$$\#\text{passi doppi} \cong \log_2(n) \longrightarrow \#\text{passi singoli} < 2\log_2(n)$$

pertanto, il numero di passi di cui si necessità per terminare il ciclo while è estremamente piccolo, anche nel caso in cui il numero n considerato è enormemente grande.

Osservazione: Un altro modo per descrivere l'algoritmo di Euclide è quello che riguarda la **ricorsività**, come mostrato di seguito

Algorithm 7 Euclide

```
1: Procedura Euclid(a,b)
2: if b = 0 then
3:     return a
4: esle
5:     return Euclid(b, a mod b)
```

4.2 Notazione *O* grande

Si considerno due funzione $f(x)$ e $g(x)$ infinite a $+\infty$, ovvero

$$\lim_{x\to+\infty} f(x) = +\infty \quad \text{e} \quad \lim_{x\to+\infty} g(x) = +\infty$$

Allora le due funzioni si rassomiglieranno per $x \to +\infty$ quando hanno lo stesso ordine di infinito, ovvero

$$\lim_{x\to+\infty} \frac{f(x)}{g(x)} = \alpha \in \mathbb{R}^+ - \{0\}$$

Tuttavia, taluna è una generalizzazione molto spartana, ma la notazione O grande lo è ancora di più. Si considerino, nuovamente, due funzioni $f(x)$ e $g(x)$ che debbono essere **definitivamente positive**, ovvero

$$\exists x_n : \forall x > x_n, f(x) > 0 \wedge g(x) > 0$$

In questo caso, ovviamente, affermare che $f(x)$ "assomiglia" a $g(x)$ significa affermare che $f(x)$ appartiene alla stessa classe di equivalenza di $g(x)$ (ovvero la classe delle funzioni che rassomigliano a $g(x)$), che si indica come segue

$$f(x) \in \Theta\left(g(x)\right)$$

che, generalmente, verrà denotato con

$$f(x) = \Theta\left(g(x)\right)$$

nonostante sia più propriamente corretto impiegare un segno di appartenza in luogo di uno di uguaglianza.

APPARTENENZA ALLA CLASSE DI EQUIVALENZA DI UNA FUNZIONE

Si dirà che $f(x)$ apparterrà alla stessa classe di equivalenza di $g(x)$ e si scriverà

$$f(x) = \Theta\left(g(x)\right)$$

se

$$\exists c_1 > 0, c_2 > 0, \overline{x} : c_1 \cdot g(x) \leq f(x) \leq c_2 \cdot g(x), \quad \forall x \geq \overline{x}$$

ovvero si riesce a descrivere ipoteticamente, con la funzione $g(x)$, una guaina all'interno della quale racchiudere la funzione $f(x)$.

Osservazione: Dalla definizione appena fornita, appare evidente come questa sia a tutti gli effetti una **relazione di equivalenza**, in quanto è soddisfatta la proprietà **riflessiva**

$$g(x) = \Theta\left(g(x)\right)$$

così come quella **simmetrica**

$$f(x) = \Theta\left(g(x)\right) \longrightarrow g(x) = \Theta\left(f(x)\right)$$

e infine quella **transitiva**

$$f(x) = \Theta\left(g(x)\right), g(x) = \Theta\left(h(x)\right) \longrightarrow f(x) = \Theta\left(h(x)\right)$$

Esempio: Si consideri la classe di equivalenza

$$\Theta\left(1\right)$$

Allora ad essa vi apparterranno tutte le costanti positive, così come le funzioni oscillanti che assumono valori positivi, come $f(x) = 2 + \sin(x)$, in quanto

$$2 \cdot 1 \leq f(x) \leq 3 \cdot 1, \quad \text{con } g(x) = 1$$

Osservazione: Si osservi che se due funzioni appartengono alla stessa classe di equivalenza, allora hanno lo stesso ordine di infinito. Tuttavia non è vero il contrario.
Si considerino, infatti, due funzioni $f(x)$ e $g(x)$ tali che

$$\lim \frac{f(x)}{g(x)} = \alpha$$

quindi definitivamente, per $x \geq \overline{x}$, si ha che

$$\alpha - \epsilon \leq \frac{f(x)}{g(x)} \leq \alpha + \epsilon$$

dal momento che dalla definizione di limite sia ha $\forall \epsilon > 0$, è possibile anche considerare $\epsilon = \frac{\alpha}{2}$, da cui

$$\frac{\alpha}{2} \leq \frac{f(x)}{g(x)} \leq \frac{\alpha}{2}$$

ma allora, moltiplicando ambo i membri per $g(x)$, essendo per definizione necessariamente positiva, la disuguaglianza si conserva diviene

$$\frac{\alpha}{2} \cdot g(x) \leq f(x) \leq \frac{\alpha}{2} \cdot g(x)$$

che corrisponde esattamente alla definizione di due funzioni che appartengono alla stessa classe di equivalenza. per cui, naturalmente, si è dimostrato che avere lo stesso ordine di infinito significa anche appartenere alla stessa classe di equivalenza, ma non è vero il contrario.