Note per il corso di

Analisi e Progettazione di Algoritmi*
Primo modulo

a.a. 21/22

Elena Zucca

23 marzo 2022

Indice

*Queste note sono in gran parte elaborate a partire da materiale didattico del professor Elio Giovannetti, che ringrazio infinitamente. Gli esempi e gli esercizi servono ad acquisire familiarità con le tecniche descritte, ma *non* saranno oggetto di domande all'orale.

1 Analisi della correttezza e complessità degli algoritmi

1.1 Notazioni asintotiche

Def. 1.1 Una funzione $g(n)$ appartiene all'insieme $O(f(n))$ (si dice anche $g(n)$ è $O(f(n))$) se esistono due costanti $c > 0$ e $n_0 \geq 0$ tali che $g(n) \leq cf(n)$ per ogni $n \geq n_0$, ossia, da un certo punto in poi, g sta sotto una funzione "multipla" di f ("cresce al più come f").

Una funzione $g(n)$ appartiene all'insieme $\Omega(f(n))$ (si dice anche $g(n)$ è $\Omega(f(n))$) se esistono due costanti $c > 0$ e $n_0 \geq 0$ tali che $g(n) \geq cf(n)$ per ogni $n \geq n_0$, ossia, da un certo punto in poi, f sta sopra una funzione "sottomultipla" di f ("cresce almeno come f").

Una funzione $g(n)$ appartiene all'insieme $\Theta(f(n))$ (si dice anche $g(n)$ è $\Theta(f(n))$) se esistono tre costanti $c_1, c_2 > 0$ e $n_0 \geq 0$ tali che $c_1 f(n) \leq g(n) \leq c_2 f(n)$ per ogni $n \geq n_0$, ossia, da un certo punto in poi, f è compresa tra un "multiplo" di f e un "sottomultiplo" di f ("cresce come f").

Le espressioni "multiplo" e "sottomultiplo" sono un abuso di linguaggio, giustificato dalla seguente osservazione: nella definizione di O è significativo il fatto che c possa essere maggiore di uno, mentre nella definizione di Ω è significativo che c possa essere minore di uno.

Per comodità si usa anche scrivere $f(n) = O(g(n))$, e analogamente per le altre notazioni. Occorre però ricordare che si tratta di un abuso di notazione, poiché il simbolo $=$ in questo caso non denota un'uguaglianza, ma l'appartenenza a un insieme.

Proprietà della notazione asintotica

Transitiva
$f(n)$ è $O(g(n))$ e $g(n)$ è $O(h(n))$ implica $f(n)$ è $O(h(n))$
$f(n)$ è $\Omega(g(n))$ e $g(n)$ è $\Omega(h(n))$ implica $f(n)$ è $\Omega(h(n))$
$f(n)$ è $\Theta(g(n))$ e $g(n)$ è $\Theta(h(n))$ implica $f(n)$ è $\Theta(h(n))$

Riflessiva

$f(n)$ è $O(f(n))$
$f(n)$ è $\Omega(f(n))$
$f(n)$ è $\Theta(f(n))$

Simmetrica $f(n)$ è $\Theta(g(n))$ se e solo se $g(n)$ è $\Theta(f(n))$

Simmetrica trasposta $f(n)$ è $O(g(n))$ se e solo se $g(n)$ è $\Omega(f(n))$

Somma $f(n) + g(n)$ è $O(\max\{f(n), g(n)\})$, analogamente per Ω e Θ

Comportamenti notevoli Trattabili

$\Theta(1)$	costante
$\Theta(\log n)$	logaritmico
$\Theta(\log n)^k$	poli-logaritmico
$\Theta(n^{1/2})$	radice quadrata
$\Theta(n)$	lineare
$\Theta(n \log n)$	pseudolineare
$\Theta(n^2)$	quadratico
$\Theta(n^3)$	cubico

Intrattabili

$\Theta(2^n)$	esponenziale
$\Theta(n!)$	fattoriale
$\Theta(n^n)$	esponenziale in base n

1.2 Complessità di algoritmi e problemi

In generale sia il numero di passi elementari che lo spazio richiesto da un algoritmo dipendono dalla dimensione n dell'input. Tuttavia, in generale non sono *funzioni* di n, perché, a parità di dimensione, input diversi possono richiedere un tempo di calcolo e un'occupazione di memoria molto diversi (per esempio, l'algoritmo di ordinamento insertion sort effettua $\Theta(n)$ confronti nel caso di una sequenza ordinata, mentre in altri casi ne effettua $\Theta(n^2)$). Scrivere $T(n)$ ed $S(n)$ sarebbe quindi ambiguo: occorre

distinguere, per ogni n, *caso migliore*, *caso peggiore* e *caso medio*. Formalmente, se i è un input, siano $t(i)$ e $s(i)$ il tempo di esecuzione e lo spazio di memoria necessario [1] per l'esecuzione dell'algoritmo per l'input i. Le seguenti funzioni risultano allora ben definite:

complessità temporale del caso peggiore $T_{worst}(n) = \max\{t(i) \mid i \text{ ha dimensione } n\}$

complessità temporale del caso migliore $T_{best}(n) = \min\{t(i) \mid i \text{ ha dimensione } n\}$

complessità temporale del caso medio $T_{avg}(n) = \text{avg}\{t(i) \mid i \text{ ha dimensione } n\}$

Queste tre funzioni hanno di solito andamenti asintotici ben definiti. Analoghe definizioni si possono dare per la complessità spaziale.

Per il caso medio, per ogni n si considerano tutti i possibili input di dimensione n, siano $i_1, \ldots i_N$, e si fa la media aritmetica dei tempi:

$$T_{avg}(n) = \frac{t(i_1)+\ldots+t(i_N)}{N}$$

Ovviamente, se i possibili casi di input non sono equiprobabili, ma si presentano con probabilità $p_1, \ldots, p_N$ con $p_1 + \ldots + p_N = 1$, la media deve essere una media pesata:

$$T_{avg}(n) = p_1 \cdot t(i_1) + \ldots + p_N \cdot t(i_N)$$

Vi sono quindi nove possibili affermazioni sulla complessità di un algoritmo, non tutte però indipendenti tra loro nè ugualmente interessanti. In particolare, ovviamente si ha che se $T_{worst}(n) = O(f(n))$, allora anche $T_{avg}(n) = O(f(n))$ e $T_{best}(n) = O(f(n))$, e simmetricamente se $T_{best}(n) = \Omega(f(n))$, allora anche $T_{avg}(n) = \Omega(f(n))$ e $T_{worst}(n) = \Omega(f(n))$.

Sono quindi giustificate le seguenti definizioni:

- *un algoritmo ha complessità* $O(f(n))$ se $T_{worst}(n) = O(f(n))$. Ossia, $f(n)$ è una delimitazione superiore del tempo di calcolo: al crescere di n il tempo di calcolo non cresce più di $f(n)$, qualunque sia l'input.
- *un algoritmo ha complessità* $\Omega(f(n))$ se $T_{best}(n) = \Omega(f(n))$. Ossia, $f(n)$ è una delimitazione inferiore del tempo di calcolo: al crescere di n il tempo di calcolo cresce almeno come $f(n)$, qualunque sia l'input.
- *un algoritmo ha complessità* $\Theta(f(n))$ se ha complessità $O(f(n))$ e $\Omega(f(n))$. Ossia, $f(n)$ è una delimitazione sia inferiore che superiore del tempo di calcolo, qualunque sia l'input.

Per esempio, nel caso di insertion sort, il tempo di esecuzione nel caso peggiore è $\Theta(n^2)$, quindi anche $O(n^2)$, quindi, per definizione, il tempo di esecuzione dell'algoritmo è $O(n^2)$. Il tempo nel caso migliore è $\Theta(n)$, quindi anche $\Omega(n)$ quindi, per definizione, il tempo di esecuzione dell'algoritmo è $\Omega(n)$. D'altra parte, il tempo di esecuzione dell'algoritmo non è nè $\Theta(n^2)$ nè $\Theta(n)$: per ogni dimensione n vi sono sia input per cui il tempo è proporzionale a n, sia input per cui è proporzionale a n^2.

Il caso migliore è di solito scarsamente interessante, poiché riguarda input molto particolari; un'eccezione è per esempio il caso di insertion sort discusso sopra, in cui il caso migliore (sequenza ordinata) è significativo. Il caso medio è significativo solo se la distribuzione delle probabilità riflette la situazione reale di uso dell'algoritmo. La determinazione di tale distribuzione può essere difficile, inoltre in applicazioni critiche l'efficienza nel caso medio può non essere una garanzia sufficiente; un'eccezione è per esempio quicksort, in cui il caso medio è interessante. Il caso peggiore è la complessità che si studia di solito, poiché è l'unica che fornisce la garanzia che in ogni caso il tempo di esecuzione non sarà maggiore di un certo tempo prevedibile.

Vediamo brevemente come viene analizzata la complessità degli algoritmi *randomizzati*.

Un algoritmo randomizzato è un algoritmo la cui computazione dipende da una o più scelte casuali, quindi dato un input *non* fornisce sempre lo stesso output. Possiamo schematizzare un algoritmo randomizzato nel modo seguente:

```
random_alg(i)
  c = random(i)
  return alg(i,c)
```

Il vantaggio si ha tipicamente nel caso di algoritmi deterministici che risultano inefficienti su alcune istanze.

Abbiamo definito sopra la complessità temporale del caso medio di un algoritmo deterministico:

$$T_{avg}(n) = \text{avg}\{t(i) \mid i \text{ ha dimensione } n\}$$

[1] In aggiunta allo spazio occupato dall'input stesso.

Calcoliamo quindi la media (pesata) dei tempi di computazione su tutti gli input della medesima dimensione: dipende dalla distribuzione di probabilità dell'input.

Nel caso di un algoritmo randomizzato, possiamo calcolare la media (pesata) dei possibili tempi di computazione per *uno stesso input* (*expected running time*): dipende dalla distibuzione di probabilità della variabile casuale.

$$T_{exp}(i) = \mathbb{E}[t(i, c)]$$

La complessità che interessa è il tempo atteso peggiore (*worst case expected running time*):

$$T_{exp\_worst}(n) = \max\{\, T_{exp}(i) \mid i \text{ ha dimensione } n\}$$

Per esempio, quicksort ha complessità $O(n^2)$ nel caso peggiore, $O(n \log n)$ nel caso medio. L'algoritmo risulta quindi inefficiente su alcune istanze. Quicksort randomizzato ha tempo atteso nel caso peggiore $O(n \log n)$. Un'istanza che sia un caso peggiore richiede sempre tempo $O(n^2)$ con l'algoritmo deterministico; con la versione randomizzata può richiedere $O(n^2)$ per qualche esecuzione, ma *in media* richiede $O(n \log n)$.

Nel confronto precedente abbiamo assunto che l'algoritmo randomizzato fornisca sempre il risultato corretto (algoritmi Las Vegas). Alcuni tipi di algoritmi randomizzati (Monte Carlo) possono anche fornire un risultato errato (forniscono comunque il risultato giusto con probabilità > 0). In questo caso la ripetizione dell'esecuzione permette di limitare la probabilità di sbagliare. Il vantaggio si ha tipicamente in casi in cui l'algoritmo deterministico è molto inefficiente.

Introduciamo ora la nozione di complessità *dei problemi*.

delimitazione superiore *Un problema ha complessità* $O(f(n))$ se esiste un algoritmo di complessità $O(f(n))$ che lo risolve. Ossia, è possibile risolvere il problema in un tempo che cresca non più di $f(n)$.

delimitazione inferiore *Un problema ha complessità* $\Omega(f(n))$ se tutti i possibili algoritmi risolventi hanno complessità $\Omega(f(n))$. Ossia, non è possibile risolvere il problema in un tempo che cresca meno di $f(n)$.

Quindi, per trovare una delimitazione superiore $f(n)$ alla complessità di un problema, è sufficiente (e necessario) trovare *un* algoritmo che risolva il problema in un tempo $O(f(n))$, mentre, per trovare una delimitazione inferiore $g(n)$, è necessario dimostrare che *qualunque* possibile algoritmo deve impiegare un tempo $\Omega(g(n))$. Non basta quindi che tutti gli algoritmi conosciuti abbiano complessità $\Omega(g(n))$.

Un problema algoritmico, in un dato momento storico, può essere *aperto* (o *con gap algoritmico*) o *chiuso*. Ovviamente, un problema aperto può venire chiuso, ma non viceversa. Un problema è chiuso se si conoscono limite superiore e inferiore coincidenti, ossia:

- esiste un algoritmo risolvente di complessità $O(f(n))$
- si è dimostrato che qualunque algoritmo risolvente deve avere complessità $\Omega(f(n))$, ossia non può esistere un algoritmo di complessità inferiore a $\Omega(f(n))$.

Si è quindi dimostrato che l'algoritmo risolvente è *ottimo*. Saranno possibili solo miglioramenti marginali, per esempio per una costante additiva o moltiplicativa.

Un problema è aperto se (tutte le) delimitazioni inferiori e superiori differiscono, ossia:

- il miglior algoritmo risolvente noto ha complessità $O(f(n))$, per esempio $O(n^2)$
- si è dimostrato che qualunque algoritmo risolvente deve avere complessità $\Omega(g(n))$, con $g \neq f$, per esempio $\Omega(n)$. In altri termini, si sa risolvere il problema in un tempo $f(n)$, per esempio n^2, e si sa che non lo si può risolvere in un tempo migliore di $g(n)$, dove $g(n)$ "cresce meno" di $f(n)$, per esempio n.

Per esempio, supponendo di conoscere come algoritmi di ordinamento solo selection sort e insertion sort, poiché entrambi sono quadratici nel caso peggiore, si può affermare che la complessità del problema dell'ordinamento ha limite superiore $O(n^2)$. Inoltre, dato che un algoritmo di ordinamento ha necessariamente complessità almeno lineare in quanto deve esaminare tutti gli elementi, si può affermare che la complessità del problema dell'ordinamento ha limite inferiore $\Omega(n)$. In base solo a queste informazioni, il problema dell'ordinamento avrebbe un gap algoritmico. Un gap algoritmico può essere chiuso in due modi:

- dal di sopra: si trova un algoritmo migliore, abbassando così il limite superiore; se si trova un algoritmo di complessità coincidente con il limite inferiore, tale algoritmo è ottimo e il problema è chiuso
- dal di sotto: si riesce a dimostrare un limite inferiore più alto; se questo coincide con la complessità dell?algoritmo migliore esistente, si è dimostrato che l'algoritmo è ottimo e il problema è chiuso.

I due modi non sono mutuamente esclusivi, ossia può succedere che si riesca a trovare un algoritmo migliore, e contemporaneamente si riesca a dimostrare un limite inferiore più alto.

Per esempio, nel caso del problema dell'ordinamento, entrambi i limiti possono essere spostati, infatti:

- si conoscono algoritmi migliori, come mergesort, che ha complessità $O(n \log n)$, si può quindi affermare che il problema ha limite superiore $O(n \log n)$
- si può dimostrare che il problema, a certe condizioni (ordinamento per soli confronti), ha limite inferiore $\Omega(n \log n)$.

In base a queste considerazioni si può concludere che il problema dell'ordinamento per confronti è chiuso.

Nella sezione 4 considereremo il *problema della soddisfacibilità* di una formula booleana. Per esso, esiste un ovvio algoritmo di complessità esponenziale (l'algoritmo banale che prova tutte le possibili combinazioni di valori di verità), si ha un ovvio limite inferiore polinomiale, e:

- nessuno è riuscito a trovare un algoritmo polinomiale
- nessuno è riuscito a dimostrare che un algoritmo polinomiale non può esistere.

Vi sono parecchi problemi come questo, per i quali gli unici algoritmi risolventi noti sono esponenziali, e si sospetta fortemente, ma non si è dimostrato, che non esistano algoritmi migliori. Un altro esempio famoso è quello del commesso viaggiatore. Si tratta in generale di problemi per i quali, se si possiede la soluzione (per esempio un'assegnazione di valori di verità che rende soddisfacibile la formula booleana), verificare che essa è corretta è facile, cioè può essere fatto in modo efficiente. Si tratta dei problemi NP-completi, nozione che formalizzeremo nella sezione 4. Per altri problemi algoritmici, come, per esempio, le torri di Hanoi, gli unici algoritmi risolventi sono esponenziali, e si è dimostrato che non possono esistere algoritmi migliori. Infine, non tutti i problemi sono risolubili: per alcuni si può dimostrare che non esiste un algoritmo che risolve il problema (questo è argomento del corso di Teoria degli Automi e Calcolabilità). Per esempio, il problema di determinare, dati due programmi, se essi sono equivalenti, cioè se per qualunque input producono lo stesso output, oppure il problema dell'arresto (dato un programma e un input per tale programma, determinare se il programma, per quell'input, termina).

1.3 Correttezza di algoritmi ricorsivi

È basata sul principio di induzione, di cui casi particolari molto usati sono il principio di induzione aritmetica e quello di induzione aritmetica completa (anche detta induzione aritmetica forte), ma che più in generale si applica a qualunque insieme definito induttivamente. Vediamo cosa significa.

Def. 1.2 [Definizione induttiva] Una *definizione induttiva* R è un insieme di *regole* $\frac{Pr}{c}$, dove Pr è un insieme detto delle *premesse* e c è un elemento[2], detto la *conseguenza* della regola. L'insieme *definito induttivamente* $\mathcal{I}$ è *il più piccolo* tra gli insiemi X *chiusi rispetto a* R, ossia tali che, per ogni regola $\frac{Pr}{c} \in R$, se $Pr \subseteq X$ allora $c \in X$.

In altre parole, un insieme definito induttivamente è un insieme i cui elementi sono *tutti e soli* quelli che si ottengono applicando ripetutamente un insieme di regole, a partire da quelle con premessa vuota che costituiscono la base della definizione induttiva.

Una definizione induttiva è in genere data attraverso una qualche descrizione finita "a parole", per esempio:

> L'insieme dei numeri pari è il più piccolo insieme tale che (oppure: è l'insieme definito induttivamente da):
>
> - 0 è un numero pari,
> - se n è un numero pari, allora $n + 2$ è un numero pari.

Questa descrizione corrisponde all'insieme (infinito) di regole: $\frac{\emptyset}{0} \cup \{\frac{n}{n+2} \mid n \in \mathbb{N}\}$.

Esempi tipici di insiemi definiti induttivamente sono i naturali, le liste, gli alberi, gli alberi binari. Quando un insieme è definito induttivamente è possibile provarne delle proprietà *per induzione*, ossia utilizzando il principio dato sotto.

Prop. 1.3 [Principio di induzione (forma generale)] Sia R una definizione induttiva, $\mathcal{I}$ l'insieme da essa definito, e P un predicato su U con $\mathcal{I} \subseteq U$. Se

Base $P(c)$ vale per ogni $\frac{\emptyset}{c} \in R$

Passo induttivo ($P(d)$ vale per ogni $d \in Pr$) implica che valga $P(c)$ per ogni $\frac{Pr}{c} \in R, Pr \neq \emptyset$

[2] Si usa talvolta fissare l'insieme *universo* U che contiene tutte le premesse e conseguenze delle regole, tuttavia questo può essere ricavato a posteriori dalle regole e quindi omesso dalla definizione.

allora $P(d)$ vale per ogni $d \in \mathcal{I}$.

Dimostrazione Poniamo $C = \{d \mid P(d) \text{ vero}\}$ e osserviamo che la condizione $(\star)$ può essere scritta nella forma equivalente:

$Pr \subseteq C$ implica $c \in C$.

Allora è chiaro che C risulta chiuso rispetto a R, quindi $\mathcal{I} \subseteq C$, quindi $P(d)$ vale per ogni $d \in \mathcal{I}$. □

Sia R una definizione induttiva, $\mathcal{I} \subseteq U$, e P un predicato su U.

Vediamo alcuni casi particolari del principio di induzione.

Prop. 1.4 [Principio di induzione aritmetica] Sia P un predicato sui numeri naturali tale che

Base vale $P(0)$

Passo induttivo se vale $P(n)$ allora vale $P(n+1)$,

allora $P(n)$ vale per ogni $n \in \mathbb{N}$.

Dimostrazione $\mathbb{N}$ può essere visto come l'insieme definito induttivamente da

- $0 \in \mathbb{N}$
- se $n \in \mathbb{N}$ allora $n+1 \in \mathbb{N}$.

Quindi la proposizione è un caso particolare del principio di induzione. □

Vediamo due esempi di applicazione che utilizzeremo spesso nel seguito.

Esempio 1.5 [Serie aritmetica] Proviamo per induzione aritmetica che $\sum_{i=1}^{n} i = \frac{n(n+1)}{2}$, per tutti gli $n \in \mathbb{N}$.

Base $P(0)$: $0 = 0$

Passo induttivo Assumiamo $P(n)$ e dimostriamo $P(n+1)$:

$$\begin{aligned} \sum_{i=1}^{n+1} i = & \sum_{i=1}^{n} i + (n+1) \\ = & \frac{n(n+1)}{2} + (n+1) \quad \text{(per ipotesi induttiva)} \\ = & \frac{(n+1)(n+2)}{2} \end{aligned}$$

Esempio 1.6 [Serie geometrica] Proviamo per induzione aritmetica che $\sum_{i=0}^{n} q^i = \frac{q^{n+1}-1}{q-1}$, per tutti gli $n \in \mathbb{N}$.

Base $P(0)$: $q^0 = 1 = \frac{q^1-1}{q-1}$

Passo induttivo Assumiamo $P(n)$ e dimostriamo $P(n+1)$:

$$\begin{aligned} \sum_{i=0}^{n+1} q^i &= \sum_{i=0}^{n} q^i + q^{n+1} \\ &= \frac{q^{n+1}-1}{q-1} + q^{n+1} \qquad \text{(per ipotesi induttiva)} \\ &= \frac{q^{n+1}-1+q^{n+2}-q^{n+1}}{q-1} \\ &= \frac{q^{n+2}-1}{q-1} \end{aligned}$$

Prop. 1.7 [Principio di induzione aritmetica completa (o forte)] Sia P un predicato sui numeri naturali tale che

Base vale $P(0)$

Passo induttivo se vale $P(m)$ per ogni $m < n$ allora vale $P(n)$,

allora $P(n)$ vale per ogni $n \in \mathbb{N}$.

Dimostrazione $\mathbb{N}$ può essere visto come l'insieme definito induttivamente da

- $0 \in \mathbb{N}$
- se $\{m \mid m < n\} \subseteq \mathbb{N}$ allora $n \in \mathbb{N}$.

Quindi la proposizione è un caso particolare del principio di induzione. □

Esempio 1.8 Un altro esempio: possiamo dare una definizione induttiva degli alberi binari (con nodi in N) :

- l'albero vuoto è un albero binario,
- se L e R sono alberi binari e $x \in N$, allora la tripla (L, x, R) è un albero binario.

Di conseguenza, è possibile provare proprietà degli alberi binari con il principio di induzione, che prende la seguente forma:

Sia P un predicato sugli alberi binari tale che

- P vale sull'albero vuoto,
- se P vale su L e R, allora vale su (L, x, R),

allora P vale su tutti gli alberi binari.

Esercizio 1.9

- Definire induttivamente l'altezza di un albero binario.
- Provare per induzione sulla definizione di albero binario che il numero di nodi di un albero binario di altezza h è compreso tra $h + 1$ e $2^{h+1} - 1$.

Se abbiamo un algoritmo ricorsivo il cui input varia in un insieme definito induttivamente, e definito a sua volta in modo induttivo (ossia ricalcando la definizione induttiva dell'insieme), è facile provare la correttezza dell'algoritmo con il principio di induzione.

Per esempio nel caso di un algoritmo sui numeri naturali:

- se l'algoritmo è corretto per 0
- e assumendo che sia corretto per n, è corretto per $n + 1$,

allora è corretto su qualunque numero naturale. Analogamente si può applicare il principio di induzione arimetica forte o il principio di induzione per alberi, liste, etc.

Inoltre, il principio di induzione non solo ci consente di provare a posteriori la correttezza di un algoritmo, ma fornisce anche una guida per il suo sviluppo. Per esempio, nel caso dell'induzione aritmetica:

- si scrive l'algoritmo corretto per 0
- se l'argomento è $n > 0$, si assume che l'algoritmo sappia operare correttamente su $n - 1$, e fidandosi di tale assunzione si scrive l'algoritmo corretto per operare su n.

Esempio 1.10 [Correttezza di ricerca binaria ricorsiva] La correttezza del seguente algoritmo di ricerca binaria ricorsiva:

```
binary_search(x,a)// a[0..n-1]
        return binary_search(x,a,0,n-1)

binary_search(x,a,inf,sup)
  if (inf <= sup)
    mid = (inf + sup)/2
    if (x < a[mid]) return binary_search(x,a,inf,mid-1)
    else if (x > a[mid]) return binary_search(x,a,mid+1,sup)
    else return true
  return false
```

si basa sull'induzione aritmetica forte, in quanto le chiamate ricorsive sono effettuate su sequenze di dimensione strettamente minore di quella iniziale. Si tratta di un algoritmo *divide-et-impera*.

1.4 Correttezza di algoritmi iterativi

Come si scrive un algoritmo iterativo "in modo che sia corretto"?

```
//Pre: precondizione = quello che sappiamo essere vero all'inizio
while (B)
   C
//Post: postcondizione = quello che vogliamo sia vero alla fine
```

L'algoritmo è corretto se, assumendo che valga Pre all'inizio, possiamo concludere che vale $Post$ alla fine.
Cos'è un'invariante di ciclo? qualunque condizione Inv tale che:

```
//Pre: Inv ∧ B
C
//Post: Inv
```

cioè, se devo eseguire il corpo del ciclo e la condizione vale, allora vale anche dopo averlo eseguito.
Quando un'invariante serve a provare la correttezza?

- Inv vale all'inizio
- se valgono insieme Inv e $\neg\ B$ e allora vale $Post$.

Infatti, in questo caso è facile vedere per induzione aritmetica sul numero di iterazioni che, assumendo che il comando termini, alla fine vale $Post$. Come si prova la terminazione? trovando una quantità t (funzione di terminazione) tale che, se devo eseguire il corpo, quindi se valgono Inv e B:

- viene decrementata eseguendo il corpo
- è limitata inferiormente.

Esempio 1.11 [Esempio "didattico"]

```
//Pre: x = x0 ∧ y = y0 ∧ x ≥ 0
while (x != 0)
  x = x-1
  y = y+1
//Post: y = x0 + y0
```

Ci sono molte invarianti, per esempio $\mathtt{x} \geq 0$, $\mathtt{x} \leq x_0$, $\mathtt{y} \geq y_0$, `true`, `false`, ma quella che mi serve è $\mathtt{x}+\mathtt{y} = x_0+y_0 \wedge \mathtt{x} \geq 0$.

```
//Pre: x = x0 ∧ y = y0 ∧ x ≥ 0
while (x!=0) //Inv: x + y = x0 + y0 ∧ x ≥ 0
  x = x-1
  y = y+1
//Post: y = x0 + y0
```

Infatti:

- vale banalmente all'inizio
- se vale insieme a $\mathtt{x} = 0$, allora vale la postcondizione.
- se vale insieme a $\mathtt{x} \neq 0$ prima di eseguire il corpo, allora vale dopo

La funzione di terminazione che mi serve è il valore di $\mathtt{x}$, infatti:

- viene decrementato a ogni passo
- assumendo che valga Inv (quindi $\mathtt{x} \geq 0$), è limitato inferiormente da 0.

Metodologia: l'invariante si ottiene in genere "indebolendo" la postcondizione, in modo da modellare il "risultato parziale al passo generico". L'invariante non serve solo per provare la correttezza a posteriori, ma soprattutto come guida allo sviluppo dell'algoritmo. In molti algoritmi iterativi, inclusi una buona parte di quelli che vedremo, l'idea iniziale è "una buona invariante". Una volta trovata la proprietà invariante, si scrivono nell'ordine: il corpo del ciclo in modo che produca un avvicinamento alla soluzione, la condizione di controllo in modo che in uscita si abbia la postcondizione, l'inizializzazione in modo da garantire che l'invariante valga inizialmente. Vedremo esempi di questa metodologia nella sezione 1.6.

1.5 Esempi di progettazione e analisi di algoritmi ricorsivi

Esempio 1.12 [Torri di Hanoi] Problema delle torri di Hanoi: si hanno tre pioli verticali (sinistro, centrale, destro), e n dischi forati, di diametri tutti diversi, che si possono infilare nei pioli. All'inizio tutti i dischi sono su un piolo in ordine decrescente di diametro. Scopo del gioco: spostare tutti i dischi su un altro piolo, eventualmente utilizzando il piolo rimanente, senza violare le seguenti regole:

- si può spostare un solo disco per volta, cioè quello più in alto nel piolo di partenza
- non si può mettere un disco sopra un altro di diametro inferiore.

Problema: il gioco è risolubile per qualunque numero n di dischi? E se lo è, quale è per ogni n la sequenza di mosse che lo risolve?

Teorema 1.13 Il gioco è risolubile per qualsiasi numero n di dischi.

Dimostrazione Per induzione aritmetica su n.

Base Il gioco è risolubile per $n = 1$. Ovvio, perché un solo disco può essere spostato con una sola mossa da un piolo a un altro.

Passo induttivo Assumendo (ipotesi induttiva) che il problema sia risolubile per $n-1$, proviamo che è risolubile per n. Infatti, per ipotesi induttiva è possibile spostare gli $n-1$ dischi superiori dal piolo di partenza (sia *from*) al piolo ausialiario (sia *aux*). A questo punto è possibile spostare il disco più grande da *from* al polo di arrivo (sia *to*). Infine, sempre per ipotesi induttiva è possibile spostare gli $n-1$ dischi superiori da *aux* a *to*. □

La dimostrazione per induzione della risolubilità del problema fornisce direttamente un algoritmo ricorsivo:

```
hanoi(n, from, aux, to)
      if (n = 1) move(from, to)
      else
        hanoi(n-1, from, to, aux)
        move(orig, dest)
        hanoi(n-1, aux, from, to)
```

Analisi della complessità: il tempo di calcolo è evidentemente proporzionale al numero di mosse.

- Per $n = 1$ si ha una mossa, quindi $T(1) = 1$.
- Per $n > 1$ occorrono:
  - $T(n-1)$ per mosse per spostare gli $n-1$ dischi da *from* ad *aux*
  - 1 mossa per spostare il disco più grande da *from* a *to*
  - $T(n-1)$ mosse per spostare gli $n-1$ dischi da *aux* a *to*.

Si ha quindi la seguente *relazione di ricorrenza*:

$T(1) = 1$
$T(n) = 1 + 2\,T(n-1)$, per $n > 1$.

Una tecnica che spesso consente di individuare la soluzione di una relazione di ricorrenza consiste nell'espandere successivamente le chiamate ricorsive nella definizione di $T(n)$, come esemplificato sotto:

$T(n) = 1 + 2\,T(n-1) =$
$1 + 2(2\,T(n-2) + 1) =$
$2^0 + 2^1 + 2^2\,T(n-2) =$
$\ldots =$
$2^0 + \ldots + 2^i\,T(n-i) = \ldots =$ (notando che l'ultimo termine si ha per $i = n-1$)
$2^0 + \ldots + 2^{n-1}\,T(n-(n-1)) = 2^0 + \ldots + 2^{n-1} =$ (vedi esempio 1.6) $2^n - 1$

Un modo alternativo di ottenere lo stesso risultato è espandendo "per livelli" (dell'albero di ricorsione), nel modo illustrato sotto:

livello 0: $1 = 2^0$ mossa
livello 1: $2 = 2^1$ mosse
livello 2: 2^2 mosse
...
livello i: 2^i mosse
... (notando, analogamente a sopra, che l'ultimo livello che fornisce un contributo è quello $n-1$)
livello $n-1$: 2^{n-1} mosse

Sommando il contributo dei vari livelli si ottiene la sommatoria precedente.

Una volta individuata la soluzione in uno dei due modi, si può verificarne rigorosamente la correttezza per induzione aritmetica:

Base $T(1) = 2^1 - 1$ vero per definizione.

Passo induttivo $T(n+1) = 2\,T(n) + 1 =$ (hp.ind)$= 2(2^n - 1) + 1 = 2^{n+1} - 1$

Si ha quindi che $T(n) = \Theta(2^n)$, mentre la complessità in spazio corrisponde all'altezza dell'albero di ricorsione, ossia alla massima profondità dello stack, quindi è $S(n) = \Theta(n)$.

Esercizio 1.14

1. Provare che questo numero di mosse è minimo, quindi il gioco delle Torri di Hanoi è intrattabile: il numero di mosse cresce esponenzialmente al crescere del numero dei dischi.
2. (non banale) Dare un algoritmo iterativo e provarne la correttezza (si ottiene la stessa sequenza di mosse).

Esempio 1.15 [Complessità di ricerca binaria ricorsiva] La relazione di ricorrenza per la ricerca binaria ricorsiva è la seguente (conviene esprimere n come 2^k):

$T(2^0) = 1$
$T(2^k) = 1 + T(2^{k-1})$, per $k > 0$.

Utilizzando la tecnica per sostituzioni successive si ha:

$$T(2^k) = 1 + T(2^{k-1}) = 1 + 1 + T(2^{k-2}) = \ldots = 1 + \ldots + 1 \ (k+1 \text{ volte})$$

quindi $T(n) =)(\log n)$.

Esempio 1.16 [Quicksort] Schema astratto dell'algoritmo, anch'esso di tipo divide-et-impera:

```
quicksort(s)
   if (|s| > 1)
     togli un elemento p da s (per esempio il primo)
     partiziona gli altri elementi di s in due parti:
       una sequenza s1 contenente tutti gli elementi < p
       una sequenza s2 contenente tutti gli elementi ≥ p
       forma la sequenza s1 p s2
     quicksort(s1)
     quicksort(s2)
```

L'elemento *p* è detto *pivot* o *perno* della partizione. È facile provare la correttezza dell'algoritmo per induzione aritmetica completa sulla lunghezza n della sequenza *s*:

Base Una sequenza di lunghezza zero o uno è ordinata, e infatti l'algoritmo non fa nulla.

Passo Dopo la partizione si ha:

elementi di $s_1 < p \leq$ elementi di s_2

Per ipotesi induttiva, l'algoritmo ordina correttamente sequenze di lunghezza $< n$. Le sequenze s_1 ed s_2 hanno certamente lunghezza $< n$ perché è stato tolto *p*, quindi dopo le due chiamate ricorsive s_1 ed s_2 sono ordinate; allora la sequenza $s_1\ p\ s_2$ è chiaramente ordinata.

Complessità temporale: assumiamo di poter effettuare la partizione in tempo lineare e senza usare strutture ausiliarie (vi sono diversi algoritmi concreti).

$T(1) = \Theta(1)$
$T(n) = \Theta(n) + T(r) + T(n-r-1)$ per $n > 1$

dove $\Theta(n)$ è il tempo necessario per effettuare la partizione, $T(r)$ il tempo necessario per ordinare s_1 di lunghezza r, $T(n-r-1)$ il tempo necessario per ordinare s_2 di lunghezza $n-r-1$, ed $r \geq 0$ è diverso per ogni chiamata ricorsiva.

Il caso migliore si avrà quando, a ogni chiamata ricorsiva, le due parti s_1 ed s_2 risultano di uguale lunghezza (partizione sempre bilanciata). In questo caso avremo

$T(1) = \Theta(1)$
$T(n) = \Theta(n) + T(\frac{n}{2}) + T(\frac{n}{2})$, per $n > 1$.

Questa relazione di ricorrenza è la stessa del mergesort ed ha soluzione $T(n) = \Theta(n \log n)$ (provarlo per esercizio).

Il caso peggiore si avrà quando, a ogni chiamata ricorsiva, le due parti s_1 ed s_2 risultano una di lunghezza 0 e l'altra di lunghezza $n-1$ (partizione sempre sbilanciata al massimo). In questo caso avremo:

$T(1) = \Theta(1)$
$T(n) = \Theta(n) + T(n-1)$ per $n > 1$.

che ha soluzione $T(n) = \Theta(n^2)$, infatti espandendo si ha

$T(n) = n + (n-1) + ... + 1 = n(n+1)/2$

come si può verificare facilmente per induzione aritmetica.

Si può provare che si ha anche

$T_{avg}(n) = \Theta(n \log n)$

1.6 Esempi di progettazione e analisi di algoritmi iterativi

Esempio 1.17 [Ricerca binaria iterativa] Sia la sequenza rappresentata con un array `a` con indici $0..n-1$. Idea: si confronta l'elemento da cercare con l'elemento centrale dell'array, a seconda del risultato ci si restringe a considerare solo la prima o la seconda metà, e così via. Idea per l'invariante: al passo generico il valore da cercare x, se presente nell'array, si trova nella porzione di array compresa fra due indici `inf` e `sup`, formalmente:

$\texttt{x} \in \texttt{a}[0..n-1] \Rightarrow \texttt{x} \in \texttt{a}[\texttt{inf}..\texttt{sup}]$

Guidati da questa invariante:

Passo confronto con x l'elemento centrale `a[mid]` della porzione `a[inf..sup]`, sono possibili tre casi:

- `x` $<$ `a[mid]`: `x`, se c'è, si trova nella porzione `a[inf..mid` -1`]`, quindi `sup = mid-1`
- `x` $>$ `a[mid]`: `x`, se c'è, si trova nella porzione `a[mid` $+1$`..sup]`, quindi `inf = mid+1`
- `x` $=$ `a[mid]`: ho trovato `x`, fine!

Condizione di controllo la porzione di array su cui effettuare la ricerca non deve essere vuota, quindi `inf` $\leq$ `sup`

Inizializzazione inizialmente la porzione di array su cui effettuare la ricerca è l'intero array, quindi `inf` $= 0$ e `sup` $= n-1$.

Algoritmo completo:

```
binary_search(x,a)
  inf = 0; sup = n-1 //Pre: inf = 0 ∧ sup = n − 1
  while (inf <= sup) //Inv: x ∈ a[0..n − 1] ⇒ x ∈ a[inf..sup]
    mid = (inf + sup)/2
    if (x < a[mid]) sup = mid-1
    else if (x > a[mid]) inf = mid+1
    else return true
  //Post: x ∉ a[0..n − 1]
  return false
```

Vediamo ora un altro esempio in cui il ruolo dell'invariante è ancora più significativo.

Esempio 1.18 [Bandiera nazionale olandese] Si ha una sequenza di n elementi rossi, bianchi e blu. Vogliamo modificare la sequenza in modo da avere prima tutti gli elementi rossi, poi tutti i bianchi, poi tutti i blu. Le operazioni che possiamo effettuare sulla sequenza sono:

- `swap(i,j)` scambia di posto due elementi, per $1 \leq \mathtt{i}, \mathtt{j} \leq n$
- `colour(i)` restituisce `Red`, `White`, `Blue` per $1 \leq \mathtt{i} \leq n$

Vogliamo inoltre esaminare ogni elemento *una volta sola*.

Indichiamo con $Red(\mathtt{i}, \mathtt{j})$ il predicato "gli elementi da `i` a `j` compresi sono tutti rossi" e analogamente $White(\mathtt{i}, \mathtt{j})$, $Blue(\mathtt{i}, \mathtt{j})$. Siano inoltre `U`, `W` e `B` gli indici del primo elemento ignoto, bianco e blu. La postcondizione sarà quindi:

$$Red(1, \mathtt{W}-1) \wedge White(\mathtt{W}, \mathtt{B}-1) \wedge Blue(\mathtt{B}, n)$$

e l'invariante:

$$Red(1, \mathtt{U}-1) \wedge White(\mathtt{W}, \mathtt{B}-1) \wedge Blue(\mathtt{B}, n) \wedge \mathtt{U} \leq \mathtt{W}$$

(l'ultima condizione nell'invariante è aggiunta per chiarezza).

Algoritmo completo:

```
//Pre: U = 1 ∧ W = B = n + 1
while (U < W) //Inv: Red(1, U − 1) ∧ White(W, B − 1) ∧ Blue(B, n) ∧ U ≤ W
    switch (Colour(W-1)
      case Red : swap(U, W-1); U = U + 1
      case White : W = W - 1
      case Blue : swap(W-1, B-1); W = W-1; B = B-1
//Post: Red(1, W − 1) ∧ White(W, B − 1) ∧ Blue(B, n)
```

Prova di correttezza:

- Inv vale all'inizio banalmente
- $Inv \wedge \mathtt{U} \geq \mathtt{W}$ implica $Post$ perché is ha $\mathtt{U} = \mathtt{W}$
- se vale Inv e $\mathtt{U} < \mathtt{W}$ eseguendo il corpo del ciclo vale ancora Inv: si vede per casi
- come funzione di terminazione possiamo prendere $\mathtt{W} - \mathtt{U}$, infatti decresce eseguendo il corpo del ciclo in ognuno dei casi, e se vale la condizione di controllo è limitata inferiormente.

2 Grafi

2.1 Introduzione e terminologia

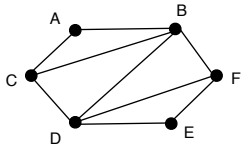
Def. 2.1 [Grafo] Un *grafo (orientato)* è una coppia $G = (V, E)$ dove:

- V è un insieme i cui elementi sono detti *nodi* o *vertici*
- E è un insieme di *archi* (*edges*), dove un arco è una coppia di nodi detti *estremi* dell'arco.

In un grafo *non orientato* gli archi sono coppie non ordinate, ossia (u, v) e (v, u) denotano lo stesso arco.

Un arco da un nodo in se stesso è detto *cappio*, e un grafo senza cappi è detto *semplice*. La definizione di grafo si può generalizzare a quella di *multigrafo*, in cui gli archi sono un multiinsieme. Nel caso di grafi orientati, il primo elemento della coppia è detto *nodo uscente* o *coda*, il secondo *nodo entrante* o *testa*.

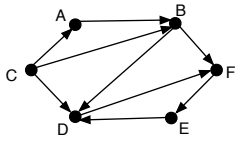
Dato il seguente grafo non orientato:



- l'arco (A, B) è *incidente* sui nodi A e B,

- i nodi A e B sono *adiacenti*, A è *adiacente* a B, B è *adiacente* ad A,
- i nodi adiacenti a un nodo A si chiamano anche i *vicini* di A,
- il *grado* $\delta(u)$ di un nodo *u* è il numero di archi incidenti sul nodo, per esempio $\delta(B) = 4$.

Dato il seguente grafo orientato:



- l'arco (A, B) è *incidente* sui nodi A e B, *uscente* da A, *entrante* in B,
- il nodo B è *adiacente* ad A, ma A non è adiacente a B,
- i nodi adiacenti a un nodo A si chiamano anche i *vicini* di A,
- il *grado* $\delta(u)$ di un nodo *u* è il numero di archi incidenti sul nodo, per esempio $\delta(B) = 4$,
- il *grado uscente* (*outdegree*) $\delta_{out}(u)$ di un nodo *u* è il numero di archi uscenti dal nodo, per esempio $\delta_{out}(B) = 2$,
- il *grado entrante* (*indegree*) $\delta_{in}(u)$ di un nodo *u* è il numero di archi entranti nel nodo, per esempio $\delta_{in}(B) = 2$.

Dato un grafo $G = (V, E)$, con n nodi ed m archi, si hanno le seguenti ovvie proprietà:

- se G è non orientato
  - la somma dei gradi dei nodi è il doppio del numero degli archi: $\sum_{u \in V} \delta(u) = 2m$,
  - m è al massimo il numero di tutte le possibili coppie non ordinate di nodi, ossia $n + (n-1) + \ldots + 1 = n(n+1)/2$, quindi $m = O(n^2)$.
- se G è orientato:
  - la somma dei gradi uscenti dei nodi e la somma dei gradi entranti dei nodi sono uguali al numero degli archi: $\sum_{u \in V} \delta_{out}(u) = \sum_{u \in V} \delta_{in}(u) = m$, quindi anche in questo caso $\sum_{u \in V} \delta(u) = 2m$,
  - m è al massimo il numero di tutte le possibili coppie ordinate di nodi, ossia n^2, quindi $m = O(n^2)$.

Naturalmente un grafo dove vi sia un arco per ogni coppia di nodi è un caso limite molto particolare e poco interessante, in genere si avrà m minore o molto minore del limite superiore. Quindi la complessità degli algoritmi sui grafi viene di solito espressa in funzione dei due parametri n e m ossia come $T(n, m)$, anche se m è in ogni caso $O(n^2)$.

Un grafo in cui il numero degli archi sia dell'ordine di n^2 si dice grafo *denso*.

In un grafo G:

- un *cammino* (*path*) è una sequenza di nodi $u_0, \ldots, u_n$ con $n \geq 0$ e, per ogni $i \in 0..n-1$, u_{i+1} adiacente a u_i, ossia (u_i, u_{i+1}) è un arco,
- nel caso di un grafo non orientato si trova anche il termine *catena* invece di cammino,
- gli archi (u_i, u_{i+1}) si dicono *contenuti* nel cammino,
- $n - 1$, ossia il numero degli archi, è la *lunghezza* del cammino,
- un cammino è *degenere* o *nullo* se è costituito da un solo nodo, ossia ha lunghezza 0,
- è *semplice* se i nodi sono distinti, tranne al più il primo e l'ultimo,
- in un grafo orientato, un cammino non nullo forma un *ciclo* se il primo nodo coincide con l'ultimo,
- in un grafo non orientato, un cammino (catena) di lunghezza ≥ 3 forma un *ciclo* o *circuito* (semplice) se il primo nodo coincide con l'ultimo e tutti gli altri nodi sono distinti,

- *v* è *raggiungibile* da *u* se esiste un cammino da *u* a *v*.

Un grafo G è *aciclico* se non vi sono cicli in G. Un grafo orientato aciclico è anche detto DAG (*directed acyclic graph*). Un grafo non orientato si dice *connesso* se ogni nodo è raggiungibile da ogni altro. Un grafo orientato si dice *fortemente connesso* se ogni nodo è raggiungibile da ogni altro, *debolmente connesso* se il grafo non orientato corrispondente è connesso. Un grafo connesso avente n nodi deve avere almeno $n - 1$ archi.

Un *sottografo* di $G = (V, E)$ è un grafo ottenuto da G non considerando alcuni archi o alcuni nodi insieme agli archi incidenti su di essi. Il sottografo *indotto da un sottoinsieme* V' *di nodi* è il sottografo di G costituito dai nodi di V' e dagli archi di G che connettono tali nodi.

Un *albero libero* è un grafo non orientato connesso aciclico. Se in un albero libero si fissa un nodo *u* come radice, si ottiene un albero nel senso usuale (ossia, *radicato*), avente *u* come radice. Graficamente si può pensare di "appendere" il grafo a un qualunque nodo, e si ottiene sempre un albero.

Dato un grafo non orientato e connesso G, un *albero ricoprente* (*spanning tree*) di G è un sottografo di G che contiene tutti i nodi ed è un albero libero. Ossia, è un albero che connette tutti i nodi del grafo usando archi del grafo, ha quindi n nodi ed $n - 1$ archi. Analogamente, dato un grafo non orientato (eventualmente non connesso) G, si chiama *foresta* ricoprente di G un sottografo di G che contiene tutti i nodi ed è una foresta libera. Si vede facilmente che, dato un grafo non orientato, esiste sempre una foresta ricoprente di G, un albero ricoprente se G è connesso.

Rappresentazione di grafi

Lista di archi Si memorizza l'insieme dei nodi e la lista degli archi, spazio totale $O(n + m)$. Molte operazioni richiedono di scorrere l'intera lista.

operazione	**tempo di esecuzione**
grado di un nodo	$O(m)$
nodi adiacenti	$O(m)$
esiste arco	$O(m)$
aggiungi nodo	$O(1)$
aggiungi arco	$O(1)$
elimina nodo	$O(m)$
elimina arco	$O(m)$

Liste di adiacenza Per ogni nodo si memorizza la lista dei nodi adiacenti. Si hanno n liste, di lunghezza totale $2m$ per grafo non orientato, m per grafo orientato, quindi complessità spaziale $O(n + m)$. Diventa più semplice trovare gli adiacenti di un nodo, operazione cruciale in molte applicazioni, mentre il test di verifica dell'esistenza di un arco non è molto efficiente. Inoltre nel caso di grafo non orientato ogni arco è rappresentato due volte, quindi si ha ridondanza e quindi il problema di mantenere la coerenza dell'informazione.

operazione	**tempo di esecuzione**
grado di *u*	$O(\delta(u))$
nodi adiacenti a *u*	$O(\delta(u))$
esiste arco (u, v)	$\min(\delta(u), \delta(v))$
aggiungi nodo	$O(1)$
aggiungi arco	$O(1)$
elimina nodo	$O(m)$
elimina arco (u, v)	$O(\delta(u) + \delta(v))$

Matrice di adiacenza assumendo corrispondenza tra nodi e $1..n$, matrice quadrata M di dimensione n a valori booleani (oppure $0, 1$), dove $M_{i,j}$ vero se e solo se esiste l'arco (u_i, u_j). Richiede più spazio, $O(n^2)$, ma la verifica della presenza di un arco è in tempo costante. Per contro, trovare gli adiacenti diventa più costoso (intera riga), e nel caso di grafi non orientati si ha ridondanza (matrice simmetrica). Il tempo per l'aggiunta ed eliminazione di un nodo tiene conto della riallocazione.

operazione	**tempo di esecuzione**
grado di un nodo	$O(n)$
nodi adiacenti	$O(n)$
esiste arco	$O(1)$
aggiungi nodo	$O(n^2)$
aggiungi arco	$O(1)$
elimina nodo	$O(n^2)$
elimina arco	$O(1)$

La rappresentazione generalmente più conveniente è quella con liste di adiacenza, in particolare per grafi *sparsi* ossia con un numero di archi m molto minore di n^2. La rappresentazione con matrice di adiacenza può essere preferibile quando il grafo è *denso* (m prossimo a n^2) o quando è importante controllare in modo efficiente se esiste un arco tra due vertici dati.

Entrambe le rappresentazioni sono facilmente adattabili ai grafi *pesati*, ossia dove ogni arco ha un *peso* (*costo*) associato.

2.2 Visite

Per visitare un grafo si possono usare algoritmi simili a quelli per gli alberi. Tuttavia, poiché un nodo è raggiungibile in più modi, bisogna "marcare" i nodi, in modo che se si raggiunge un nodo già visitato non lo si visiti di nuovo. Le operazioni eseguite al passo di visita dipendono, come sempre, dal particolare algoritmo.

L'insieme dei nodi viene partizionato in tre insiemi, tradizionalmente indicati con i colori bianco, grigio, e nero:

bianco inesplorato, ossia non ancora toccato dall'algoritmo

grigio aperto, ossia toccato dall'algoritmo (visita iniziata)

nero chiuso (visita conclusa)

Le visite iterative usano un insieme F detto *frangia* da cui vengono via via estratti i nodi da visitare. A seconda della struttura dati impiegata per la frangia si ottengono diversi tipi di visita: analogamente a quanto succede per gli alberi, la visita in ampiezza utilizza una coda, la visita in profondità una pila, mentre gli algoritmi di Dijkstra e Prim che vedremo successivamente utilizzano una coda a priorità.

Consideriamo per semplicità la visita del sottografo connesso a partire da un nodo. Ossia, dati un grafo G e un nodo di partenza s, la visita di tutti i nodi raggiungibili da s. Costruisce implicitamente l'*albero di visita* T, cioè l'albero di radice s in cui il genitore di un nodo *u* è l'altro estremo *v* dell'arco (v, u) percorrendo il quale si è arrivati a *u*. La costruzione dell'albero di visita può essere resa esplicita nell'algoritmo stesso, per esempio memorizzando per ogni nodo il suo genitore. I nodi nell'albero di visita corrente sono tutti grigi o neri. Si tratta di un albero ricoprente. Si generalizza a *visita completa* che costruisce una foresta ricoprente dell'intero grafo.

Visita in ampiezza (breadth-first) La frangia è una coda `Q`.

```
BFS(G,s) //visita nodi di G raggiungibili a partire da s
  for each (u nodo in G) marca u come bianco; parent[s] = null
  Q = coda vuota
  Q.add(s); marca s come grigio;
  while (Q non vuota)
    u = Q.remove() //u non nero
    visita u
    for each ((u,v) arco in G)
          if (v bianco)
            marca v come grigio; Q.add(v); parent[v]=u
    marca u come nero
```

Osservazioni: la coda mantiene l'ordine in cui i nodi sono trovati dall'algoritmo, ogni nodo entra in coda una volta sola. Invariante del ciclo: nodi grigi = nodi in F = nodi i cui archi uscenti non sono ancora stati esaminati; nodi nell'albero = nodi neri e grigi. In realtà in questo caso non è necessario distinguere tra nodi grigi e neri ma lo facciamo per chiarezza. L'algoritmo dato sopra costruisce il *sottografo dei predecessori*:

- i nodi sono tutti quelli raggiungibili da s, ossia tutti gli u tali che `parent[u]`$\neq$ `null`, più s stesso
- gli archi sono gli u, v tali che `parent[v]`=*u* e v è raggiungibile.

Nella visita in ampiezza, il predecessore (padre) di un nodo viene deciso nel momento in cui il nodo viene incontrato.

Si vede facilmente che il sottografo dei predecessori risulta essere effettivamente un albero (detto *albero di visita in ampiezza*) o *albero BFS*, in quanto è connesso e il numero di archi è uguale al numero di nodi raggiungibili meno uno. Inoltre, ogni nodo è il più vicino possibile alla radice, ossia, la visita calcola la *distanza* (lunghezza minima di un cammino) dalla radice a ogni nodo.

Visita in ampiezza e cammini minimi: definiamo $d(u)$ la distanza di *u* da s, ossia la lunghezza minima di un cammino da s a *u*.

Il livello di un nodo *u* nell'albero di visita è uguale alla lunghezza minima di un cammino da s a *u*, cioè `level`$(u) = d(u)$.

Visita in profondità (depth-first) La frangia è una pila `S`.

```
DFS(G,s) //visita nodi di G raggiungibili a partire da s
  for each (u nodo in G) marca u come bianco; parent[s] = null
  S = pila vuota
  S.push(s); marca s come grigio;
  while (S non vuota)
    u = S.pop()
    if (u non nero)
      visita u
      for each ((u,v) arco in G)
        if (v bianco)  marca v come grigio; S.push(v); parent[v]= u
        else if (v grigio) S.push(v); parent[v]= u //modifica il padre
      marca u come nero
```

Osservazioni: si segue un cammino nel grafo finché possibile. Un nodo può essere inserito nella pila anche se grigio, quindi più volte, nel caso peggiore tante volte quanti sono i suoi archi entranti. Il padre viene modificato ogni volta. Può essere estratto dalla pila un nodo nero.

Complessità della visita completa: assumiamo n = numero nodi, m = numero archi, e che la marcatura e le operazioni di inserimento, eliminazione e modifica delle strutture dati siano fattibili in tempo costante. Nota: assunzioni valide per pila e coda, potrebbero non esserlo con strutture dati diverse, per esempio coda a priorità per algoritmi di Dijkstra e Prim.

- marcature dei nodi: $O(n)$
- inserimenti in F e T, modifiche di F e T, estrazioni da F: ogni nodo viene inserito una prima volta in F e T, poi eventualmente F e T vengono aggiornati al più m volte: $O(n+m)$
- esplorazione archi incidenti eseguita per ogni nodo, quindi:
  - lista di archi: $O(n \cdot m)$
  - liste di adiacenza: $O(n+m)$ perché ogni lista di adiacenza viene scandita una volta sola
  - matrice di adiacenza: $O(n^2)$

Visita in profondità ricorsiva La visita in profondità si può anche realizzare molto semplicemente in modo ricorsivo, analogo all'algoritmo ricorsivo per la visita preorder sugli alberi. Occorre tuttavia marcare i nodi come visitati. Diamo l'algoritmo per la visita completa (ossia, anche di un grafo non connesso) e una marcatura "nero" inutile ai fini dell'algoritmo che evidenzia la fine della visita.

```
DFS(G)
  for each (u nodo in G) marca u come bianco; parent[u]=null
  for each (u nodo in G) if (u bianco) DFS(G,u)

DFS(G,u)
  //inizio visita
  visita u; marca u come grigio
  for each ((u,v) arco in G)
    if (v bianco)
      parent[v]=u
      DFS(G,v)
  //marca u come nero
  //fine visita
```

Nella visita in profondità, in modo del tutto analogo a quanto visto per la visita in ampiezza, viene costruita una *foresta* DFS.

2.3 Cammini minimi in un grafo pesato: algoritmo di Dijkstra

Def. 2.2 [Grafo pesato] Un grafo *pesato* è un grafo $G = (V, E)$ dove a ogni arco è associato un *peso* o *costo* attraverso una funzione $c\_ : E \to \mathbb{R}$. Il costo di un cammino da s a t è la somma dei pesi o costi degli archi che compongono il cammino. Un cammino da s a t si dice *minimo* se ha peso minimo (detto *distanza*) fra tutti i cammini da s a t.

Naturalmente può esistere più di un cammino minimo. Se non esistono archi con peso negativo, un cammino minimo fra due nodi connessi esiste sempre, altrimenti può non esistere.

Problema: dato un grafo orientato pesato G, con pesi non negativi (che a seconda dell'applicazione possono rappresentare distanze, tempi di percorrenza, costi di attività, etc.), e dati un nodo di partenza s e un nodo di arrivo t, trovare un cammino minimo da s a t. Generalizzazione: dati un grafo orientato pesato G con pesi non negativi e un nodo di partenza s, trovare per ogni nodo *u* del grafo il cammino minimo da s a *u*.

Esempi di applicazione: navigatore satellitare (trovare l'itinerario più corto, o più veloce, da un luogo a un altro su una mappa stradale); trovare il percorso di durata minima fra due stazioni di una rete ferroviaria o di una rete di trasporto pubblico urbano; protocollo di routing OSPF (Open Shortest Path First) usato in Internet per trovare la migliore connessione da ogni nodo della rete a tutte le possibili destinazioni.

Algoritmo di Dijkstra (1959), idea: è una visita in ampiezza in cui a ogni passo:

- per i nodi già visitati si ha la distanza e l'albero T di cammini minimi
- per ogni nodo non ancora visitato si ha distanza provvisoria = lunghezza minima di un cammino in T più un arco
- si estrae dalla coda un nodo a distanza provvisoria minima (che risulta essere la distanza)
- si aggiornano le distanze provvisorie dei nodi adiacenti al nodo estratto tenendo conto del nuovo arco in T

Dato che ogni volta si estrae un minimo, conviene rappresentare l'insieme dei nodi ancora da visitare come coda a priorità. L'uso di una coda a priorità realizzata come heap invece che come semplice array o lista fu introdotto da Johnson nel 1977; tale versione dell'algoritmo è quindi talvolta chiamata algoritmo di Johnson.

Per non dover trattare a parte il caso di nodi non in coda, conviene inserire all'inizio in coda tutti i nodi assegnando loro come distanza provvisoria "infinito".

```
Djikstra(G,s)
  for each (u nodo in G)  dist[u] = ∞ // tutti i nodi sono bianchi
  parent[s] = null; dist[s] = 0 // s diventa grigio
  Q = heap vuoto
  for each (u nodo in G)  Q.add(u,dist[u])
  while (Q non vuota)
      u = Q.getMin() //estraggo nodo a distanza provvisoria minima, u diventa nero
      for each ((u,v) arco in G) //v diventa o resta grigio
          if (dist[u] + c_{u,v} < dist[v]) // se v nero falso
            parent[v] = u; dist[v] = dist[u] + c_{u,v}
            Q.changePriority(v, dist[v]) //moveUp
```

Correttezza Sia $d(u)$ la distanza (lunghezza di un cammino minimo) da s a *u*. L'invariante è composta di due parti:

1. per ogni nodo *u* non in Q, ossia "nero"
   $\texttt{dist}[u] = d(u)$
2. per ogni nodo u in Q
   $\texttt{dist}[u] = d_{\setminus Q}(u)$ = lunghezza di un cammino minimo da s a *u* i cui nodi, tranne *u*, non sono in Q, ossia sono nodi "neri".

Convenzione: se questo cammino non esiste, ossia *u* non è raggiungibile da s solo attraverso nodi neri (*u* è "bianco"), $d_{\setminus Q}(u) = \infty$.

L'invariante vale all'inizio:

1. vale banalmente perché non ci sono nodi neri (tutti i nodi sono in coda)
2. vale banalmente perché la distanza è per tutti infinito, tranne che per s per cui vale 0.

L'invariante si mantiene.

1. Viene estratto dalla coda il nodo *u* con $\texttt{dist}[u]$ minima, quindi, per l'invariante (2), tale che esiste un cammino π da s a *u* minimo tra quelli costituiti da tutti nodi neri tranne l'ultimo. Tale cammino è allora anche il minimo in assoluto fra tutti i cammini da s a *u*. Infatti, supponiamo per assurdo che esista un cammino da s a *u* di costo minore di quello di π. Questo cammino deve necessariamente contenere nodi non neri, altrimenti avrebbe costo maggiore o uguale a quello di π. Sia *w* il primo di tali nodi non neri. Il cammino quindi è della forma $\pi_1\pi_2$ con π_1 cammino da s a *w* e π_2 cammino da *w* a *u*.

Ma il cammino π_1, essendo formato solo da nodi neri tranne l'ultimo, ha costo maggiore o uguale a quello di π, e quindi a maggior ragione $\pi_1\pi_2$ ha costo maggiore o uguale a quello di π.

Quindi, aggiungendo il nodo *u* estratto dalla coda all'insieme dei nodi neri, l'invariante (1) vale ancora, ossia, è ancora vero che per tutti i nodi neri, compreso il nuovo nodo nero *u*, è stato trovato il cammino minimo.

2. Poiché l'insieme dei nodi neri risulta modificato per l'aggiunta di *u*, occorre però ripristinare l'invariante (2): per ogni nodo *v* in `Q`, `dist`[*v*] deve essere la lunghezza del minimo fra i cammini da s a *v* i cui nodi sono tutti, eccetto *v* neri; ma adesso fra i nodi neri c'è anche *u*, quindi bisogna controllare se per qualche nodo *v* adiacente a *u* il cammino da s a *v* passante per *u* è più corto del cammino trovato precedentemente, e in tal caso aggiornare `dist`[*v*] e `parent`[*v*].

Postcondizione: al termine dell'esecuzione dell'algoritmo la coda è vuota, quindi tutti i nodi sono neri. Poiché vale l'invariante (1), per ogni nodo è stato trovato il cammino minimo da s.

Nota: per dimostrare che il ciclo mantiene l'invariante (1), che è quello che interessa, è necessario dimostrare che il ciclo mantiene anche l'invariante (2).

Complessità Se la coda a priorità è realizzata come sequenza non ordinata, ogni inserimento in coda o modifica della priorità ha complessità $O(1)$, ma l'estrazione del minimo ha complessità $O(n)$, quindi la complessità dell'algoritmo è $O(n^2)$. Analogamente se la coda a priorità è realizzata come sequenza ordinata. Nella versione di Johnson, se n e m sono il numero rispettivamente dei nodi e degli archi, si ha:

- inizializzazione di tutti i nodi come bianchi: $O(n)$
- n estrazioni dallo heap: $O(n \log n)$
- ciclo interno: ogni arco viene percorso una volta, e per ogni nodo adiacente si ha eventuale `moveUp` nello heap, quindi $O(m \log n)$

Complessivamente si ha quindi $O((m + n) \log n)$. Si noti che se il grafo è denso, cioè $m = O(n^2)$, la complessità diventa $O(n^2 \log n)$, quindi peggiore della versione originale quadratica.

2.4 Algoritmo di Prim

Def. 2.3 [Minimo albero ricoprente] Dato un grafo G connesso, non orientato e pesato, un *minimo albero ricoprente*[3] di G è un albero ricoprente di G in cui la somma dei pesi degli archi è minima.

Un minimo albero ricoprente di G è quindi un sottografo di G tale che:

- sia un albero libero, ossia connesso e aciclico
- contenga tutti i nodi di G
- la somma dei pesi degli archi sia minima.

Esempi di applicazione: costruzione di reti di computer, linee telefoniche, rete elettrica, etc.

Idea: simile a Dijkstra, ma si prende ogni volta, fra tutti i nodi adiacenti a quelli per cui si è già trovato il minimo (neri), quello connesso a un nodo nero dall'arco di costo minimo, cioè si cerca il nodo "più vicino" all'albero già costruito (poi, come in Dijkstra, si aggiornano gli altri nodi).

```
Prim(G,s)
  for each (u nodo in G) marca u come non visitato //necessario
  for each (u nodo in G) dist[u] = ∞
  parent[s] = null; dist[s] = 0
  Q = heap vuoto
  for each (u nodo in G) Q.add(u, dist[u])
  while(Q non vuota)
    u = Q.getMin() //estraggo nodo a minima distanza dai neri
    marca u come visitato (nero)
    for each ((u,v) arco in G)
      if (v non visitato  &&  c_u,v < dist[v] )
        parent[v] = u; dist[v] = c_u,v
        Q.changePriority(v,dist[v]) //moveUp
```

[3] In inglese *minimum spanning tree*.

Nota: a differenza di quanto accade in Dijkstra, occorre un controllo esplicito che i nodi adiacenti al nodo *u* estratto dalla coda non siano già stati visitati, perché non è detto che per un nodo *v* già visitato il test `c_u,v < dist[v]` sia falso.

Correttezza Sia T l'albero di nodi neri (non in `Q`) corrente. L'invariante è composta di due parti:

1. $T \subseteq MST$ per qualche MST minimo albero ricoprente di G
2. per ogni nodo $u \neq s$ in `Q`
   `dist`$[u]$ = costo minimo di un arco che collega *u* a un nodo nero.

(se questo arco non esiste consideriamo il costo minimo $= \infty$).
L'invariante vale all'inizio:

1. vale banalmente perché T è vuoto (non ci sono nodi neri)
2. vale banalmente perché non ci sono nodi neri e la distanza è per tutti infinito, tranne che per s.

L'invariante si mantiene.

1. Viene estratto dalla coda un nodo *u* con `dist`$[u]$ minima, cioè connesso a un nodo nero y da un arco di costo minimo tra tutti quelli che "attraversano la frontiera", cioè uniscono un nodo nero a un nodo non nero. L'arco (y, u) diventa quindi parte dell'albero di nodi neri corrente. Dimostriamo che aggiungendo tale arco si ha ancora un sottoalbero di un minimo albero ricoprente di G.

   Supponiamo per assurdo che aggiungendo (y, u) a T l'albero ottenuto non appartenga a nessun minimo albero ricoprente di G. Per l'invariante (1), $T \subseteq MST$ per un certo MST minimo albero ricoprente di G. MST, essendo un albero ricoprente, per definizione è un sottografo connesso che connette tutti i nodi di G: quindi in MST, se non c'è l'arco (y, u), ci deve essere un altro cammino da y a *u*. Questo cammino dovrà contenere un (primo) arco (x, z) che "attraversa la frontiera", ossia connette un nodo di T a un nodo non di T.

   Poiché MST è un albero, quindi ogni coppia di nodi è connessa da un unico cammino, se eliminiamo l'arco (x, z) otteniamo un grafo non connesso, costituito da due alberi (sottografi connessi aciclici), siano T_1 e T_2. Quindi se consideriamo il sottografo formato da T_1, T_2 e l'arco (y, u) otteniamo:

   - nuovamente un albero
   - che contiene tutti i nodi di G, quindi è un albero ricoprente
   - in cui la somma dei pesi degli archi è minore o uguale di quella di MST, in quanto differisce da quest'ultimo solo per l'arco (y, u) al posto dell'arco (x, z), e $c_{y,u} \leq c_{x,z}$ dato che $c_{y,u}$ è il minimo costo di un arco da un nodo nero a un nodo non nero.

   Quindi è un minimo albero ricoprente che contiene l'arco (y, u), contro l'ipotesi per assurdo.

2. L'insieme dei nodi neri risulta modificato per l'aggiunta di *u*, quindi occorre ripristinare l'invariante (2), controllando se per qualche nodo *v* non nero e adiacente a *u* l'arco (u, v) ha costo minore del precedente arco che univa *v* a un nodo nero, e in tal caso aggiornare l'arco e la distanza.

Postcondizione: all'uscita dal ciclo tutti i nodi sono neri, quindi T connette tutti i nodi, cioè è un albero ricoprente di G. Per l'invariante (1), $T \subseteq MST$ per qualche MST minimo albero ricoprente, quindi $T = MST$.

Come in Dijkstra, si noti che per dimostrare che il ciclo mantiene l'invariante (1) è necessario dimostrare che il ciclo mantiene anche l'invariante (2).

Nota: L'algoritmo di Prim genera un albero radicato di radice s, ma il minimo albero ricoprente è per definizione un albero non radicato, quindi è possibile ottenere lo stesso a partire da nodi diversi. In particolare si può dimostrare che se i pesi degli archi sono tutti distinti il minimo albero ricoprente è unico.

L'analisi della complessità è esattamente la stessa dell'algoritmo di Dikstra, quindi $O((n + m) \log n)$.

2.5 Algoritmo di Kruskal

Anche questo algoritmo risolve il problema del minimo albero ricoprente. In particolare, il minimo albero ricoprente viene costruito mantenendo una foresta alla quale si aggiunge a ogni iterazione un nuovo arco che unisce due sottoalberi distinti. Quindi l'algoritmo di Kruskal, a differenza di quello di Prim, non costruisce l'albero a partire da un nodo scelto come radice, ma come un insieme di archi che alla fine risulta essere un grafo connesso aciclico, quindi un albero libero.

Algoritmo astratto:

```
Kruskal(G)
  s = sequenza archi di G in ordine di costo crescente
  T = foresta formata dai nodi di G e nessun arco
  while (s non vuota)
    estrai da s il primo elemento (u,v)
    if (u,v non connessi in T) T = T + (u,v)
  return T
```

Nella sequenza non si fa nessun reinserimento o variazione di ordine quindi non è una coda ma una semplice sequenza ordinata, realizzabile per esempio con un array.

Ottimizzazione: un albero di n nodi ha $n-1$ archi, quindi si può interrompere il ciclo dopo aver aggiunto all'albero $n-1$ archi.

```
Kruskal(G)
  s = sequenza archi di G in ordine di costo crescente
  T = foresta formata dai nodi di G e nessun arco
  counter = 0
  while (counter < n-1)
    estrai da s il primo elemento (u,v)
    if (u,v non connessi in T) T = T + (u,v)
    counter++
  return T
```

Correttezza Simile a Prim. Invariante: $T \subseteq MST$ per qualche MST minimo albero ricoprente di G.

All'inizio vale banalmente perché in T non ci sono archi.

Si mantiene, infatti: si sceglie un arco (u, v) di costo minimo tra quelli non ancora inseriti in T.

- Se *u* e *v* appartengono a uno stesso albero nella foresta T, aggiungendo l'arco (u, v) si avrebbe un ciclo, quindi esso viene correttamente scartato, T rimane uguale e l'invariante vale ancora.
- Se *u* e *v* appartengono a due alberi distinti nella foresta T, dimostriamo che aggiungendo l'arco (u, v) a T si ha ancora una foresta contenuta in un minimo albero ricoprente. Per assurdo supponiamo che non sia così. Per l'invariante, tutti gli archi di T appartengono a un minimo albero ricoprente di G, sia MST. Se in MST non c'è l'arco (u, v), ci deve essere un altro cammino che connette *u* a *v*. Tale cammino deve contenere un arco (x, y) che connette un nodo x dell'albero T_u che contiene *u* con un nodo y che non appartiene a tale albero, dato che *v* non vi appartiene. Tale arco non può appartenere alla foresta corrente T, perché in tal caso sarebbe un arco di T_u. Quindi tale arco è ancora in *s*, quindi $c_{u,v} \leq c_{x,y}$.

  Se eliminiamo l'arco (x, y) da MST, otteniamo due alberi non connessi fra loro, e quindi se aggiungiamo a questi due alberi l'arco (u, v) otteniamo, analogamente a Prim:

  - nuovamente un albero
  - che contiene tutti i nodi di G, quindi è un albero ricoprente
  - in cui la somma dei pesi degli archi è minore o uguale di quella di MST, in quanto differisce da quest'ultimo solo per l'arco (u, v) al posto dell'arco (x, y), e $c_{u,v} \leq c_{x,y}$.

  Quindi è un minimo albero ricoprente che contiene l'arco (u, v), contro l'ipotesi per assurdo.

Postcondizione: all'uscita dal ciclo si ha necessariamente un unico albero, perché l'algoritmo ha esaminato tutti gli archi di G unendo ogni volta due alberi di T se non ancora connessi (ossia, ha inserito $n-1$ archi, come si vede direttamente nella versione ottimizzata). Quindi alla fine T è un unico albero, sottoalbero di un minimo albero ricoprente di G contenente tutti i nodi di G, quindi è un minimo albero ricoprente di G.

Complessità algoritmo astratto: il problema è controllare se due nodi sono già connessi. Farlo in modo banale con una visita dei due alberi richiede tempo $O(n)$ nel caso peggiore, quindi si ha $O(m \cdot n)$ (infatti il controllo viene fatto nel caso peggiore su tutti gli archi). Per migliorare l'efficienza possiamo implementare la foresta con una struttura detta *union-find*.

Strutture union-find Servono a rappresentare una collezione di insiemi disgiunti sulla quale siano possibili le seguenti operazioni:

- `makeSet(a)` aggiunge l'insieme costituito dal solo elemento `a` (singleton)

- `union(A,B)` sostituisce gli insiemi `A` e `B` con la loro unione
- `find(a)` restituisce l'insieme che contiene l'elemento `a`.

Nell'algoritmo di Kruskal, gli insiemi sono le componenti connesse trovate finora, inizialmente costruite con `makeSet` (ogni nodo una componente a sé); per vedere se due nodi sono già nella stessa componente si usa `find`; per fondere due componenti connesse di usa `union`.

Supponiamo di identificare ogni insieme con un suo elemento (se pensiamo gli insiemi come classi di equivalenza, un suo rappresentante). Si ha quindi:

- `makeSet(a)` aggiunge l'insieme costituito dal solo elemento `a` (singleton)
- `union(a,b)` sostituisce gli insiemi (rappresentati da) `a` e `b` con (un rappresentante del)la loro unione
- `find(a)` restituisce (il rappresentante del)l'insieme che contiene l'elemento `a`.

Idea generale per l'implementazione: ogni insieme è un albero la cui radice è il rappresentante. Si usano alberi con numero arbitrario di figli, in genere difficili da implementare, ma in questo caso interessa solo risalire al padre. La collezione di insiemi è quindi una foresta. Basandosi su questa idea generale, è possibile dare diverse rappresentazioni con diverse caratteristiche di efficienza. Vediamo prima due rappresentazioni elementari, assumendo che n sia il numero di `makeSet`.

Gli alberi *QuickFind* permettono di eseguire rapidamente la `find`. Sono alberi di altezza *uno*.

- `makeSet(a)` aggiunge un nuovo albero con due nodi, radice `a` e figlio `a`: $O(1)$
- `union(a,b)` rende la radice dell'albero che contiene `a` padre di tutti i nodi dell'albero che contiene `b`: $O(n)$
- `find(a)` restituisce il padre di `a`: $O(1)$

Gli alberi *QuickUnion* permettono di eseguire rapidamente la `union`. Sono alberi di altezza arbitraria.

- `makeSet(a)` aggiunge un nuovo albero con un unico nodo `a`: $O(1)$
- `union(a,b)` tra rappresentanti rende `a` padre di `b`: $O(1)$
- `union(a,b)` generica, richiede prima una `find`: $O(n)$
- `find(a)` risale la catena dei padri di `a`: $O(n)$

L'operazione più costosa è la `find`, perché l'altezza degli alberi può crescere senza controllo. Si consideri per esempio:

```
makeSet(1)  ... makeSet(n)
union(n-1,n)
...
union(1,2)
```

Per mantenere gli alberi bilanciati, si può effettuare l'unione scegliendo sempre come radice del nuovo insieme quella dell'insieme di cardinalità maggiore, cioè la radice dell'albero con meno nodi diventa figlio della radice dell'altro (*union-by-size*).

Teorema 2.4 Con la union-by-size, l'altezza di ogni albero della struttura union-find è al più logaritmica nel numero di nodi dell'albero.

Dimostrazione Proviamo che $h \leq \log|T|$, ossia $2^h \leq |T|$, per ogni albero T di altezza h, è una proprietà invariante della struttura union-find.

- vale all'inizio: la proprietà vale banalmente per la struttura vuota
- vale dopo una `makeSet`: si ha un nuovo albero T con un solo nodo, per esso si ha quindi banalmente $2^h = |T|$
- vale dopo una `union`, infatti, siano T e T' i due alberi che vengono fusi, di altezza h e h' rispettivamente, con $2^h \leq |T|$, $2^{h'} \leq |T'|$ e, per esempio, $|T'| \leq |T|$:
  - se $h'+1 \leq h$, si ottiene un nuovo albero di altezza h e numero di nodi $|T|+|T'|$, e $2^h \leq |T| < |T|+|T'|$
  - se $h'+1 > h$, si ottiene un nuovo albero di altezza $h'+1$ e numero di nodi $|T|+|T'|$, e $2^{h'+1} \leq 2|T'| \leq |T|+|T'|$. □

Si ha quindi:

- `union` tra rappresentanti: $O(1)$
- `union` generica: $O(\log n)$
- `find`: $O(\log n)$

Una tecnica alternativa a quella dell'unione per dimensione, detta *union-by-rank*, è la seguente: l'unione viene effettuata scegliendo come radice del nuovo albero quella dell'albero di altezza maggiore, cioè la radice dell'albero meno alto diventa figlio della radice di quello più alto. La tecnica si chiama *union-by-rank* piuttosto che union-by-height perchè può essere generalizzata scegliendo in base non all'altezza effettiva ma a un suo *maggiorante*, vedi dopo.

Teorema 2.5 Con la union-by-rank, l'altezza di ogni albero della struttura union-find è al più logaritmica nel numero di nodi dell'albero.

Nota implementativa: ovviamente, perché le operazioni abbiano la complessità indicata sopra, per ogni albero della foresta union-find sarà necessario memorizzare l'informazione sul suo numero di nodi o altezza, in modo che quando si effettua la `union` il numero di nodi o l'altezza dell'albero unione possano essere calcolati in tempo costante a partire da quelli dei due argomenti. Per esempio, si può aggiungere un campo `size` a ogni nodo, che sarà rilevante solo per i nodi radice.

Un'altra tecnica per migliorare l'efficienza è fare in modo che la `find` durante la sua esecuzione tenda a diminuire l'altezza dell'albero, così da rendere più veloci le operazioni successive *path compression*: la `find` rende figli della radice tutti i nodi che incontra nel suo percorso di risalita dal nodo alla radice. Si noti che, dato che la `union` tra elementi generici consiste di due operazioni di `find` seguite dall'unione vera e propria, introducendo la compressione dei cammini nella `find` la introduciamo automaticamente anche nella `union`.

Nota implementativa: possiamo utilizzare insieme senza problemi la union-by-size e la compressione dei cammini, in quanto quest'ultima *non* modifica il numero di nodi degli alberi che costituiscono la foresta union-find. In altri termini, in un'implementazione in cui si aggiunga un campo `size` a ogni nodo, durante la compressione non è necessario aggiornare i campi `size` dei nodi coinvolti. Nel caso della union-by-rank, invece, la compressione dei cammini può modificare l'altezza dell'albero cui appartiene il nodo argomento della `find`. Tuttavia, anche in questo caso è possibile evitare di aggiornare l'informazione, interpretandola non più come altezza dell'albero, ma come una quantità, chiamata *rank*, che sappiamo essere *maggiore o uguale* dell'altezza dell'albero. L'unione viene effettuata in base al rank piuttosto che all'altezza, e, come nel caso dell'altezza, se i due argomenti dell'unione hanno lo stesso rank r l'albero unione deve avere rank $r + 1$, per garantire che il rank sia un maggiorante dell'altezza dell'albero.

Il vantaggio della compressione dei cammini si ha quando si effettua una sequenza di operazioni, non può quindi venire misurato dalla complessità di una singola operazione. Si considera quindi la nozione di *complessità ammortizzata.* Si tratta di una nozione utile in tutti i casi in cui il tempo di calcolo impiegato da un algoritmo che realizzi un'operazione su una struttura dati può dipendere, oltre che dalla dimensione n della struttura, dalle operazioni (dello stesso o di altro genere) effettuate prima.

Si può dimostrare che la complessità per una sequenza di n `makeSet`, m `find` e $n - 1$ `union` è $O((n + m) \log^* n)$, dove $\log^*$ è una funzione dalla crescita lentissima, in pratica costante da un certo punto in poi.

```
Kruskal(G)
  s = sequenza archi di G in ordine di costo crescente
  T = foresta formata dai nodi di G e nessun arco
  UF = struttura union-find vuota
  for each (u nodo in G) UF.makeSet(u)
  while (s non vuota)
    estrai da s il primo elemento (u,v)
      if (UF.union_by_need(u,v))
      // restituisce vero ed esegue union delle radici se UF.find(u) ≠ UF.find(v)
      // falso altrimenti
        T = T + (u,v)
  return T
```

Si noti che si ha una corrispondenza biunivoca tra gli alberi della foresta corrente T e gli insiemi della struttura union-find: due nodi appartengono a uno stesso albero in T se e solo se appartengono a uno stesso insieme nella struttura union-find.

Complessità dell'algoritmo di Kruskal nella versione con union-find:

- ordinamento degli archi: $O(m \log m)$
- n `makeSet` , $2m$ `find` e $n - 1$ `union`: "quasi" $O(n + m)$.

Essendo il grafo connesso si ha $m \geq n - 1$, quindi $O(n + m) = O(m)$, quindi complessivamente si ha $O(m \log m)$.

2.6 Ordinamento topologico

È facile vedere che in un grafo orientato aciclico (DAG) la relazione sull'insieme dei nodi "*v* è raggiungibile da *u*" è un ordine parziale, vedi definizione A.1. Un ordine totale, vedi definizione A.2, che "raffina" questo ordine parziale si chiama ordine topologico.

Def. 2.6 [Ordine topologico] Un *ordine topologico* su un grafo orientato aciclico $G = (V, E)$ è un ordine totale (stretto) $<$ su V tale che, per ogni $u, v \in V$:

$$(u, v) \in E \Rightarrow u < v$$

Problema: dato un DAG trovare un ordine topologico. È un problema rilevante per scheduling di job, realizzazione di diagramma PERT delle attività, ordine di valutazione delle caselle in un foglio di calcolo, ordine delle attività specificate da un makefile, etc.

Si vede facilmente che possono esistere diversi ordini topologici per lo stesso grafo.

Def. 2.7 In un grafo orientato definiamo un nodo *sorgente (source)* se non ha archi entranti, *pozzo (sink)* se non ha archi uscenti.

È facile vedere che in un grafo orientato aciclico esistono sempre almeno un nodo pozzo e un nodo sorgente: infatti, se per assurdo così non fosse a partire da un nodo qualunque si potrebbe sempre costruire un ciclo. Quindi, esiste sempre un ordine topologico: infatti esiste sempre un nodo sorgente, eliminando questo dal grafo con tutti i suoi archi uscenti esiste sempre un nodo sorgente nel grafo restante, e così via.

Questa osservazione porta direttamente a un ovvio algoritmo astratto. Per evitare di modificare il grafo e trovare in tempo costante, a ogni passo, un nodo sorgente, possiamo memorizzare per ogni nodo il suo indegree. Invece di rimuovere un arco (u, v) si decrementa il valore di indegree per il nodo *v*. Quando il valore di indegree per un nodo diventa zero, lo si inserisce in un insieme di nodi sorgente da cui si estrae ogni volta il nodo successivo da inserire nell'ordine topologico.

Si ottiene quindi il seguente algoritmo (n numero nodi, m numero archi):

```
topologicalsort(G)
  S = insieme vuoto
  Ord = sequenza vuota
  for each (u nodo in G) indegree[u] = indegree di u //m passi
  for each (u nodo in G) if (indegree[u] = 0) S.add(u) //n passi
  while (S non vuoto) // in tutto m passi
    u = S.remove()
    Ord.add(u) //aggiunge in fondo
    for each ((u,v) arco in G)
      indegree[v]--
      if (indegree[v]=0) S.add(v)
  return Ord
```

La complessità è quindi $O(n + m)$.

Vediamo ora un algoritmo alternativo che consiste essenzialmente in una visita in profondità. Indichiamo esplicitamente nell'algoritmo il tempo di inizio e fine visita utilizzando dei timestamp (per semplicità assumiamo un contatore `time` globale).

```
DFS(G)
  for each (u nodo in G) marca u come bianco; parent[u]=null
  time = 0
  for each (u nodo in G) if (u bianco) DFS(G,u)

DFS(G,u,T)
  time++; start[u] = time //inizio visita
  visita u; marca u come grigio
  for each ((u,v) arco in G)
    if (v bianco)
      parent[v]=u
      DFS(G,v)
  time++; end[u] = time //marca u come nero
  //fine visita
```

Proprietà della visita: se G è aciclico, per ogni (u, v), in qualunque visita in profondità di G si ha:

$\texttt{end}[v] < \texttt{end}[u]$

Prova (semi-formale):

- se visito prima *u*, allora durante la visita di *u* trovo l'arco (u, v) e *v* è bianco, quindi viene effettuata la chiamata `DFS(G,v)`, ossia inizia la visita di *v*, quindi dovrà essere necessariamente $\texttt{end}[v] < \texttt{end}[u]$
- se visito prima *v*, allora durante la visita di *v* non posso trovare un cammino da *v* a *u*, essendo G aciclico, quindi devo completare la visita di *v* prima di iniziare quella di *u*, quindi a maggior ragione si ha $\texttt{end}[v] < \texttt{end}[u]$.

Quindi, l'ordine dei nodi di G secondo il tempo di fine visita in una visita in profondità è l'inverso di un ordine topologico. Di conseguenza, per ottenere un ordine topologico dei nodi di un DAG è sufficiente effettuare una visita in profondità, inserendo a ogni fine visita il nodo in una sequenza ordinata. La complessità è semplicemente quella della visita, quindi $O(n + m)$.

2.7 Componenti fortemente connesse

Def. 2.8 In un grafo orientato G, due nodi *u* e *v* si dicono *mutuamente raggiungibili*, o *fortemente connessi*, se ognuno dei due è raggiungibile dall'altro, ossia se esistono un cammino da *u* a *v* e un cammino da *v* a *u* (si può dimostrare che questo avviene se e solo se *u* e *v* appartengono allo stesso ciclo).

È immediato vedere che la relazione di mutua raggiungibilità o connessione forte, che indichiamo con $\leftrightarrow$, è una relazione di equivalenza, vedi definizione A.3.

Def. 2.9 [Componente fortemente connessa] In un grafo orientato G, le componenti fortemente connesse sono le classi di equivalenza della relazione $\leftrightarrow$, ossia i sottografi massimali di G i cui nodi sono tutti fortemente connessi tra loro.

Passando al quoziente rispetto all'equivalenza $\leftrightarrow$, si ottiene un grafo $G^{\leftrightarrow}$ detto *grafo quoziente* in cui i nodi sono le componenti fortemente connesse di G ed esiste un arco $(\mathcal{C}, \mathcal{C}')$ se e solo se esiste un arco da un nodo in $\mathcal{C}$ a un nodo in $\mathcal{C}'$.
È chiaro che il grafo quoziente risulta essere aciclico, quindi su di esso esiste un ordine topologico.

Def. 2.10 Una componente fortemente connessa di un grafo orientato è una *sorgente* o un *pozzo* se è, rispettivamente, un nodo sorgente o pozzo nel grafo quoziente.

In una visita in profondità di un grafo G, il tempo di fine visita $end(\mathcal{C})$ di una componente fortemente connessa $\mathcal{C}$ è il massimo dei tempi di fine visita dei nodi appartenenti a $\mathcal{C}$.

Lemma 2.11 Se $\mathcal{C}$ e $\mathcal{C}'$ sono due componenti fortemente connesse del grafo G, ed esiste un arco da (un nodo di) $\mathcal{C}$ a (un nodo di) $\mathcal{C}'$, allora in qualunque visita in profondità di G si ha:

$$end(\mathcal{C}') < end(\mathcal{C})$$

Dimostrazione (semi-formale) Si hanno due casi.

- Il primo nodo visitato di $\mathcal{C}$ e $\mathcal{C}'$ è in $\mathcal{C}$, sia *u*. Allora devono essere visitati tutti i nodi di $\mathcal{C}$ e $\mathcal{C}'$ prima di terminare la visita di *u*, quindi:

  $$end(\mathcal{C}') < end(u) = end(\mathcal{C})$$

- Il primo nodo visitato di $\mathcal{C}$ e $\mathcal{C}'$ è in $\mathcal{C}'$. Allora, dato che non può esistere un cammino da $\mathcal{C}'$ a $\mathcal{C}$, la visita attraversa tutti i nodi di $\mathcal{C}'$ prima di qualunque nodo di $\mathcal{C}$, ossia la visita dei nodi di $\mathcal{C}$ inizierà dopo, e terminerà anche dopo, la visita dei nodi di $\mathcal{C}'$, quindi:

  $$end(\mathcal{C}') < end(\mathcal{C}).$$ □

Teorema 2.12 In una visita in profondità di un grafo orientato il nodo avente il massimo tempo di fine visita appartiene a una componente fortemente connessa sorgente.

Dimostrazione Sia *u* il nodo avente il massimo tempo di fine visita. Se la componente fortemente connessa $\mathcal{C}$ cui appartiene *u* non fosse una sorgente, ci sarebbe un arco da un altra componente fortemente connessa $\mathcal{C}'$ a $\mathcal{C}$. Allora, per il lemma precedente, avremmo $end(\mathcal{C}) < end(\mathcal{C}')$, e quindi $end(u)$ non sarebbe il massimo tempo di fine visita. □

Teorema 2.13 In una visita in profondità di un grafo orientato la visita di un nodo appartenente a una componente fortemente connessa pozzo $\mathcal{C}$ visita tutti e soli i nodi di $\mathcal{C}$.

Dimostrazione Ovvio, perché tutti i nodi raggiungibili vengono visitati e non c'è nessun arco uscente da una componente fortemente connessa pozzo. □

Si noti che non vale la proprietà simmetrica di quella del Teorema 2.12, ossia il nodo avente il minimo tempo di fine visita *non* appartiene necessariamente a una componente fortemente connessa pozzo. Non è quindi possibile con una visita in profondità di un grafo trovare le componenti fortemente connesse pozzo.

Osservazione: una componente fortemente connessa sorgente in un grafo orientato G è una componente fortemente connessa pozzo nel *grafo trasposto* G^{T}, cioè nel grafo che si ottiene da G invertendo l'orientamento degli archi.

Dalle precedenti proprietà è facilmente ricavabile un algoritmo per trovare le componenti fortemente connesse di un grafo G:

- si effettua una visita in profondità inserendo i nodi in una sequenza `Ord` in ordine di fine visita
- si estrae via via da `Ord` l'ultimo nodo *u*, per il Teorema 2.12 *u* si trova in una componente fortemente connessa sorgente nel grafo ottenuto da G non considerando le componenti fortemente connesse precedenti
- per trovare tutti i nodi di tale componente fortemente connessa, che per l'osservazione sopra è una componente fortemente connessa pozzo nel grafo trasposto, per il Teorema 2.13 basta effettuare una visita dei nodi (non ancora visitati) raggiungibili da *u* nel grafo trasposto G^{T}
- non è necessario durante le visite modificare i grafi G e G^{T} perché basta, come al solito, marcare i nodi visitati.

Più concretamente:

```
SCC(G)
  DFS(G,Ord) //aggiunge i nodo visitati a Ord in ordine di fine visita
  //si noti che non occorre calcolare i tempi di fine visita
  G^T = grafo trasposto di G
  Ord^↔ = sequenza vuota //ordine topologico delle c.f.c.
  while (Ord non vuota)
    u = ultimo nodo non visitato in Ord //si trova in una c.f.c. sorgente
    C = insieme di nodi vuoto
    DFS(G^T,u,C) //aggiunge nodi visitati in C
    Ord^↔.add(C) //aggiunge in fondo
  return Ord^↔
```

La sequenza delle componenti fortemente connesse ottenute è in ordine topologico, perché ogni volta si individua una componente fortemente connessa a cui arriva un arco da una delle precedenti. Se il grafo è aciclico le componenti fortemente connesse sono i singoli nodi, quindi l'algoritmo restituisce semplicemente un ordine topologico dei nodi.

Complessità:

- visita in profondità del grafo: $O(n+m)$
- generazione del grafo trasposto: $O(n+m)$
- successive visite del grafo trasposto: $O(n+m)$

quindi complessivamente $O(n+m)$.

3 Programmazione dinamica

3.1 Introduzione

Introduciamo la tecnica attraverso l'esempio dei numeri di Fibonacci.

$$\begin{aligned} fib_0 &= 0 \\ fib_1 &= 1 \\ fib_{i+1} &= fib_i + fib_{i-1} \end{aligned}$$

L'ovvio algoritmo ricorsivo per calcolarli ha relazione di ricorrenza:

$$T(n) = T(n-1) + T(n-2) + \Theta(1)$$

Quindi, dato che ovviamente $T(n+1) > T(n)$ per $n > 2$,

$T(n) > 2\,T(n-2) + 1$

Risolviamo:

$T(n) > 2(2\,T(n-4)+1)+1 = 2^2\,T(n-4) + 2^1 + 2^0 > \ldots > 2^i\,T(n-2i) + 2^{i-1} + \ldots + 2^1 + 2^0$
(ultimo termine per $i = k$ se $n = 2k$)
$> 2^k + \ldots + 2^0 = 2^{k+1} - 1 = 2^{\frac{n}{2}+1} - 1 = \Theta(\sqrt{2}^n)$.

L'albero delle chiamate ricorsive è infatti simile a quello per le torri di Hanoi ma con $n/2$ livelli.

Si ha quindi un algoritmo esponenziale. Infatti, l'algoritmo ricorsivo ricalcola inutilmente i risultati parziali, come risulta evidente provando a scrivere l'albero delle chiamate.

Tuttavia, è banale scrivere un algoritmo iterativo lineare con la tecnica della *programmazione dinamica*:

```
Fibonacci(n)
  fib = array con indici 0..n-1
  fib[0] = 0
  fib[1] = 1
  for (i=2; i < n; i++) fib[i] = fib[i-1]+ fib[i-2]
  return fib[n-1]
```

Invariante: `fib[0..i-1]` contiene i primi `i` numeri di Fibonacci. Esercizio: dare algoritmo ricorsivo equivalente.

Questo primo esempio illustra le caratteristiche della programmazione dinamica[4]. Come nell'approccio divide et impera, si ricava la soluzione di un problema componendo opportunamente le soluzioni di sottoproblemi più piccoli (la correttezza è quindi sempre provata per induzione aritmetica completa sulla dimensione dei problemi). Tuttavia, mentre l'approccio divide et impera procede in modo top-down, la programmazione dinamica procede in modo bottom-up. Ossia, per prima cosa si risolvono i sottoproblemi base e a partire da questi si risolvono i successivi fino ad arrivare a quello richiesto, memorizzando i risultati intermedi. In questo modo, nel caso in cui un sottoproblema venga utilizzato per risolvere molti problemi di livello superiore, è possibile calcolarne la soluzione una volta sola, e quindi l'approccio risulta vantaggioso.

3.2 Longest common subsequence

Consideriamo ora il problema della *più lunga sottosequenza comune* (*longest common subsequence* o *LCS*). Una *sottosequenza* di una sequenza s è una sequenza di elementi di s presi nello stesso ordine, ossia è individuata da una sequenza di indici di s.

Problema: date due sequenze trovare una sottosequenza comune di lunghezza massima (individuata da una sequenza di coppie di indici). Esempi reali di tale problema si trovano in biologia (trovare la più lunga sottosequenza comune a due sequenze di DNA), nell'ambito della sicurezza informatica (individuare, in un log costituito da una sequenza di comandi, le sottosequenze che indicano la presenza di un possibile attacco al sistema), etc.

L'algoritmo ingenuo ("brute force"), date le due sequenze s_1 ed s_2, di lunghezza m ed n rispettivamente, genera tutte le sottosequenze di s_1, e per ognuna di esse controlla se questa è anche sottosequenza di s_2, tenendo traccia della più lunga trovata. Ogni sottosequenza di s_1 corrisponde a un sottoinsieme degli indici di s_1, quindi esse sono 2^m. Per ognuna occorrono nel caso peggiore n passi per controllare se è anche sottosequenza di s_2, la complessità è quindi $n \times 2^m$.

Una formulazione induttiva della soluzione può essere data nel modo seguente, dove $X[1..m]$ e $Y[1..n]$ sono le due sequenze e $LCS(i,j)$ indica una sottosequenza comune di lunghezza massima tra i prefissi $X[1..i]$ e $Y[1..j]$.

$LCS(0,j) = [\,]$ per $0 \le j \le n$
$LCS(i,0) = [\,]$ per $0 \le i \le m$
per $i \neq 0, j \neq 0$:
$LCS(i,j) = LCS(i-1,j-1) \cdot (i,j)$ se $X(i) = Y(j)$
$LCS(i,j) = max(LCS(i-1,j), LCS(i,j-1))$ altrimenti.

Infatti, come base della definizione, la LCS di due sequenze di cui una è vuota risulta vuota. Il passo induttivo corrisponde al caso in cui entrambi i prefissi sono non vuoti (si noti che in questo caso la dimensione del problema è una coppia di numeri e consideriamo l'ordinamento prodotto). Supponendo che le tre sequenze $LCS(i,j-1)$, $LCS(i-1,j)$, $LCS(i-1,j-1)$ siano note, vogliamo calcolare $LCS(i,j)$. Occorre esaminare $X(i)$ e $Y(j)$ e considerare due casi.

- Se $X(i) = Y(j)$, tutte le sottosequenze comuni di $X[1..i]$ e $Y[1..j]$ sono tutte quelle di $X[1..i-1]$ e $Y[1..j-1]$, e tutte quelle ottenute aggiungendo in fondo a una di queste la coppia (i,j). È quindi facile vedere che $LCS(i,j) = LCS(i-1,j-1) \cdot (i,j)$.

[4] Si noti che il termine "programmazione" qui è nel senso di *pianificazione*, come per la programmazione lineare.

- Se $X(i) \neq Y(j)$, le sottosequenze comuni di $X[1..i]$ e $Y[1..j]$ non possono includere la coppia (i, j). Sono quindi sottosequenze comuni di $X[1..i-1]$ e $Y[1..j]$, oppure sottosequenze comuni di $X[1..i]$ e $Y[1..j-1]$, quindi *LCS*(i, j) è la più lunga fra le due sequenze *LCS*$(i-1, j)$ e *LCS*$(i, j-1)$.

La relazione di ricorrenza, considerando il caso peggiore, è la seguente:

$$T(n, m) = T(n-1, m) + T(n, m-1) + 1$$

Considerando la somma $k = n + m$ si ha $T(k) = 2T(k-1) + 1$, che è la relazione di ricorrenza delle torri di Hanoi. Si ha quindi un algoritmo esponenziale, si può dare un algoritmo migliore con la tecnica della programmazione dinamica. Infatti, dato che ogni problema di dimensione inferiore è utilizzato nella risoluzione di più problemi di dimensione superiore, è meglio per efficienza procedere bottom-up anziché top-down, memorizzando via via le soluzioni dei problemi.

Si costruisce una matrice `LCS` con m+1 righe ed n+1 colonne. In base alla definizione induttiva data sopra, la prima riga e la prima colonna vanno riempite con la sequenza vuota, di lunghezza 0. Si può poi procedere riga per riga (o colonna per colonna), l'ultima casella, cioè quella nell'angolo a destra in basso, conterrà la soluzione. In ogni casella non occorre memorizzare tutta la LCS ma basta mettere: se si ha lo stesso carattere sulla riga e sulla colonna, una "freccia diagonale", aumentando di 1 la lunghezza rispetto alla casella puntata dalla freccia; se sulla riga e sulla colonna ci sono due caratteri diversi, una freccia verso la cella di lunghezza maggiore fra la contigua sopra e la contigua a sinistra (se hanno uguale lunghezza si sceglie per esempio quella sopra). La lunghezza è la stessa di quella della cella puntata dalla freccia. La LCS corrispondente a ciascuna casella si ottiene, dall'ultimo elemento al primo, seguendo le frecce a partire dalla casella. Si noti che non è quindi necessario memorizzare i caratteri nelle celle, mentre la freccia è necessaria per ricostruire all'indietro la LCS, e la lunghezza è necessaria per calcolare le successive celle contigue. Se si è interessati solo alla lunghezza della LCS, ma non alla effettiva sequenza, le frecce non servono, e basta costruire la matrice delle lunghezze.

Vediamo un esempio:

		A	T	C	B	A	B
	0	0	0	0	0	0	0
B	0	0↑	0↑	0↑	↖ 1	← 1	↖ 1
A	0	↖ 1	← 1	← 1	↑ 1	↖ 2	← 2
C	0	↑ 1	↑ 1	↖ 2	← 2	↑ 2	↑ 2
A	0	↖ 1	↑ 1	↑ 2	↑ 2	↖ 3	← 3
T	0	↑ 1	↖ 2	↑ 2	↑ 2	↑ 3	↑ 3
B	0	↑ 1	↑ 2	↑ 2	↖ 3	↑ 3	↖ 4
A	0	↑ 1	↑ 2	↑ 2	↑ 3	↖ 4	↑ 4

Abbiamo evidenziato le caselle corrispondenti alla LCS `ACAB`, di lunghezza 4, che può essere ricostruita a partire dall'ultima casella, e a un'altra LCS `ACBA`.

Si noti che per calcolare `LCS(i,j)` basta conoscere tre caselle contigue (`LCS(i-1,j-1)`, `LCS(i-1,j)`, `LCS(i,j-1)`) e viceversa ogni casella può essere utilizzata per calcolarne altre tre.

Implementazione: come già detto, se si memorizza l'intera matrice, non è necessario inserire in `i,j` l'intera sequenza *LCS*(i, j): basta memorizzarne solo la lunghezza e il riferimento alla LCS utilizzata per trovarla, lo spazio necessario è quindi $O(mn)$. Usiamo per semplicità due matrici: la matrice `L` delle lunghezze (interi) e la matrice `R` dei riferimenti (coppie di indici, oppure tre valori convenzionali).

```
for (i = 0; i <= m; i++) L[i,0] = 0
for (j = 0; j <= n; j++) L[0,j] = 0

for (i = 1; i <= m; i++)
  for (j = 1; j <= n; j++)
    if (X[i] = Y[j])
      L[i,j] = L[i-1,j-1] + 1; R[i,j] = ↖
    else if (L[i,j-1] > L[i-1,j])
        L[i,j] = L[i,j-1; R[i,j] = ←
    else L[i,j] = L[i-1,j; R[i,j] = ↑
```

Come già osservato, per ricostruire la LCS si parte da `(m,n)` e si va all'indietro seguendo le frecce, in corrispondenza di ogni freccia diagonale si ha un elemento della sequenza (scrivere l'algoritmo per esercizio).

La complessità temporale è $T(m, n) = \Theta(mn)$ (costruzione di matrici $m \times n$). Assumendo come al solito $m \sim n$, l'algoritmo è quadratico: molto meglio dell'algoritmo ingenuo esponenziale.

La complessità spaziale è anch'essa $S(m,n) = \Theta(mn)$ (memorizzazione di matrici $m \times n$). Ottimizzazioni dello spazio: la prima riga e la prima colonna della matrice `R` non contengono alcuna informazione, si potrebbe quindi in questo caso usare una matrice $m \times n$. Si è preferito usare una matrice $m+1 \times n+1$ per avere gli stessi indici della matrice `L`. Alternativamente, ma complicando un po' il codice, si potrebbe usare anche per `L` una matrice $m \times n$, poiché la prima riga e la prima colonna sono sempre costituite da tutti zeri. Infine, se serve solo la lunghezza, non occorre conservare l'intera matrice ma in ogni momento solo una porzione: $\Theta(m+n)$ o anche $\Theta(min(m,n))$ (verificare per esercizio). Con queste ottimizzazioni si ottiene quindi una complessità spaziale lineare.

3.3 Algoritmo di Floyd e Warshall

Problema: dato grafo orientato pesato trovare cammini minimi tra tutte le coppie di nodi. Sono ammessi costi negativi ma non cicli di costo negativo. Per semplicità identifichiamo i nodi con i numeri $1..n$.

Idea: chiamiamo k-vincolato un cammino che passa solo per nodi in $1..k$ (esclusi gli estremi), per $k \leq n$, e indichiamo con $d^k(x,y)$ la *distanza k-vincolata tra x e y*, cioè la lunghezza minima di un cammino k-vincolato (come al solito, ∞ se tale cammino non esiste).

È facile vedere che (dato che non vi sono cicli di costo negativo) se esiste un cammino da x a y, esiste un cammino minimo semplice (ossia, senza nodi ripetuti). Possiamo quindi considerare solo i cammini semplici.

Possiamo esprimere d^k in funzione di d^{k-1} nel modo seguente:

$$d^k(x,y) = min\{d^{k-1}(x,y), d^{k-1}(x,k) + d^{k-1}(k,y)\}$$

Infatti, dato un cammino minimo (semplice) k-vincolato da x a y, si hanno due casi:

- non passa per k, quindi è anche un cammino minimo $k-1$-vincolato
- passa per k, quindi (essendo semplice) è composto da un cammino $k-1$-vincolato da x a k, e da un cammino $k-1$-vincolato da k a y.

Inoltre ovviamente

$$d^0(x,y) = \begin{cases} 0 & \text{se } x = y \\ c_{x,y} & \text{se } x \neq y \text{ ed esiste l'arco } (x,y) \\ \infty & \text{altrimenti} \end{cases}$$

Diamo un algoritmo di programmazione dinamica. Utilizziamo $n+1$ matrici $n \times n$: $\mathtt{D}_0, \ldots, \mathtt{D}_n$.

```
FloydWarshall(G)
  for each (x,y : nodi in G)
    D^0[x,y] = 0 se x=y, c_{x,y} se x ≠ y ed esiste arco (x,y), ∞ altrimenti
  for (k=1; k <= n; k++)
    for (x,y : nodi in G) D^k[x,y] = D^{k-1}[x,y]
    if (D^{k-1}[x,k] + D^{k-1}[k,y] < D^k[x,y]) D^k[x,y] = D^{k-1}[x,k] + D^{k-1}[k,y]
  return D^n
```

Questo algoritmo ha chiaramente complessità temporale e spaziale cubica. Tuttavia, è possibile utilizzare uno spazio quadratico anziché cubico (ossia, utilizzare un'unica matrice `D` anziché costruire in sequenza $n+1$ matrici. Infatti, quando si aggiorna (eventualmente) il valore di $\mathtt{D}^k\mathtt{[x,y]}$ con l'espressione $\mathtt{D}^{k-1}\mathtt{[x,k]} + \mathtt{D}^{k-1}\mathtt{[k,y]}$, utilizzando un'unica matrice potrebbe capitare che questi valori siano già stati aggiornati, cioè siano $\mathtt{D}^k\mathtt{[x,k]}$ e $\mathtt{D}^k\mathtt{[k,y]}$. Ma questo è ininfluente in quanto questi valori aggiornati sono necessariamente uguali ai precedenti, dato che stiamo calcolando la distanza vincolata da un nodo x a k (e da k a un nodo y), e quindi aggiungere il nodo k tra quelli utilizzabili non cambia le cose. Formalmente:

$d^k(x,k) = min\{d^{k-1}(x,k), d^{k-1}(x,k) + d^{k-1}(k,k)\} = d^{k-1}(x,k)$, e analogamente $d^k(k,y)$

Se vogliamo ottenere, oltre alla distanza, anche il cammino minimo, occorre calcolare anche la *matrice dei predecessori*, dove:

$\pi_{xy} = \mathtt{null}$ se $x = y$ oppure non esiste un cammino da x a y, altrimenti è il predecessore di y in un cammino minimo da x a y.

L'algoritmo modificato in questo modo è il seguente:

```
FloydWarshall(G)
  for each (x,y : nodi in G)
    D[x,y] = 0 se x=y, c_{x,y} se x ≠ y ed esiste arco (x,y), ∞ altrimenti
    Π[x,y] = x se x ≠ y ed esiste arco (x,y), null altrimenti
  for (k=1; k <= n; k++)
    for (x,y : nodi in G)
      if (D[x,k] + D[k,y] < D[x,y])
        D[x,y] = D[x,k] + D[k,y]
        Π[x,y] = Π[k,y]
return D,Π
```

La matrice dei predecessori finale individua il sottografo dei cammini minimi. [manca disegno] In particolare, il sottografo indotto dalla riga x della matrice Π sarà l'albero dei cammini minimi con radice x. [manca disegno] Formalmente, definiamo il *sottografo $G_{\Pi,x}$ dei predecessori* per x nel modo seguente:

- i nodi sono tutti i nodi raggiungibili da x, ossia gli y tali che $\pi_{xy} \neq \texttt{null}$, più x stesso
- gli archi sono tutti quelli da un predecessore a un nodo raggiungibile, ossia della forma (π_{xy}, y) con $\pi_{xy} \neq \texttt{null}$.

Dato un albero di cammini minimi $G_{\Pi,x}$, possiamo ottenere un cammino minimo da x a y nel modo seguente:

```
shortest_path(Π,x,y)
  if (x=y) return x
  else if (Π[x,y] = null) return ... [non esiste cammino]
  else return shortest_path(Π,x,Π[x,y]) · y
```

4 Teoria della NP-completezza

Significa "Non deterministic Polinomial time" perché la classe NP fu originariamente studiata nel contesto degli algoritmi non deterministici. Noi utilizzeremo una nozione equivalente più semplice, quella di verifica.

I problemi vengono suddivisi in termini di complessità computazionale in due classi principali:

- quelli risolvibili con un algoritmo polinomiale ($T(n) = O(n^k)$) sono considerati *trattabili* (facili)
- quelli per cui non esiste un algoritmo polinomiale ($T(n) = \Omega(k^n)$) *non trattabili* (difficili).

Questo per considerazioni di ordine "pratico":

- solitamente le complessità $O(n^k)$ hanno k piccoli, e possono essere ulteriormente migliorate
- per diversi modelli di calcolo, un problema che può essere risolto in tempo polinomiale in un modello, può esserlo anche in un altro
- la classe dei problemi risolvibili in tempo polinomiale ha interessanti proprietà di chiusura, in quanto i polinomi sono chiusi per addizione, moltiplicazione e composizione. Quindi per esempio se l'output di un algoritmo polinomiale è utilizzato come input per un altro l'algoritmo composto è polinomiale.

Ci sono anche problemi dei quali non si conosce la complessità (problemi aperti), tipicamente, non si è ancora riusciti a dimostrare che $T(n) = O(n^k)$ oppure $T(n) = \Omega(k^n)$.

Informalmente, la classe P è quella dei problemi *risolvibili* in tempo polinomiale, la classe NP è quella dei problemi *verificabili* in tempo polinomiale. Risolvere un problema significa che data una sua istanza ne fornisco una soluzione. Verificare un problema significa, informalmente, che data una sua istanza e una possibile soluzione, so controllare se questa risolve effettivamente l'istanza del problema. È intuitivamente chiaro che $\mathsf{P} \subseteq \mathsf{NP}$, ossia che se so risolvere un problema in tempo polinomiale so anche verificarlo in tempo polinomiale. Quello che non si sa è se $\mathsf{P} = \mathsf{NP}$, oppure $\mathsf{P} \subset \mathsf{NP}$, ossia se ci sono problemi verificabili in tempo polinomiale che non sono risolvibili in tempo polinomiale.

All'interno della classe NP vi è una classe di problemi "più difficili" di tutti gli altri. Questa è la classe dei problemi NP-completi (NP-C). Sappiamo risolvere questi problemi solo in tempo esponenziale, ma non possiamo escludere che esistano algoritmi in tempo polinomiale per risolverli. I problemi NP-C godono di un'importante proprietà: se si scoprisse un algoritmo che risolve uno di questi problemi in tempo polinomiale, tutti i problemi di NP-C (e come conseguenza tutti quelli di NP) sarebbero risolvibili in tempo polinomiale. Si dimostrerebbe quindi che $\mathsf{P} = \mathsf{NP}$.

Nel seguito:

- formalizziamo la nozione di problema
- mostriamo come astrarre dal particolare linguaggio usato per descrivere il problema
- formalizziamo le classi P, NP, NP-C
- troviamo un (primo) problema in NP-C
- descriviamo altri problemi in NP-C.

4.1 Problemi astratti e concreti e classe P

Def. 4.1 Un *problema (astratto)* è una relazione $\mathcal{P} \subseteq I \times S$, dove I è l'insieme degli *input* (o *istanze* del problema) e S è l'insieme delle (possibili) *soluzioni*.

In generale per ogni istanza la soluzione può non essere unica (per esempio può esserci più di un cammino minimo).

Def. 4.2 Un *problema (astratto) di decisione* $\mathcal{P}$ è un problema (astratto) in cui ogni input ha come soluzione vero oppure falso, ossia $\mathcal{P}\colon I \to \{T, F\}$.

Problemi di ricerca sono quelli in cui si cerca una soluzione. *Problemi di ottimizzazione* sono quelli in cui ci sono diverse soluzioni e se ne cerca una che sia "ottima" (per esempio, il minimo albero ricoprente, oppure il cammino minimo). La teoria della complessità computazionale si concentra sui problemi di decisione, in quanto in genere, per ogni problema di altro tipo $\mathcal{P}$, è possibile dare un problema di decisione $\mathcal{P}_d$ tale che risolvendo il primo si sappia risolvere il secondo. Nel caso di un problema di ricerca si restituisce vero se e solo se una soluzione esiste, nel caso di un problema di ottimizzazione si pone una limitazione al valore da ottimizzare. Per esempio: trovare il cammino minimo tra una coppia di nodi in un grafo è un problema di ottimizzazione. Un problema di decisione collegato è il seguente: dati due nodi determinare se esiste un cammino lungo al più k. Infatti se abbiamo un algoritmo per risolvere il primo problema, un algoritmo che risolve il secondo si ottiene eseguendo il primo e poi confrontando il valore minimo ottenuto con k. In altri termini il problema di decisione $\mathcal{P}_d$ è "più facile" del problema originale $\mathcal{P}$, quindi se proviamo che $\mathcal{P}_d$ è "difficile" indirettamente proviamo che lo è anche $\mathcal{P}$.

Da ora in poi quindi parleremo semplicemente di "problema" intendendo problema di decisione.

Ai fini di questo corso, non ci interessa fissare un particolare formalismo (per esempio i programmi in un certo linguaggio) per esprimere gli algoritmi, quindi un algoritmo A sarà semplicemente una funzione (in generale parziale, dato che l'algoritmo potrebbe non terminare per qualche input).

Dato un problema $\mathcal{P}\colon I \to \{T, F\}$, diciamo che *un algoritmo A risolve $\mathcal{P}$* se, per ogni input $i \in I$, $A(i) = \mathcal{P}(i)$. Un problema $\mathcal{P}$ è nella classe $\mathsf{Time}(f(n))$ se e solo se esiste un algoritmo di complessità temporale $O(f(n))$ che lo risolve, dove n è la dimensione dell'input. Analogamente, $\mathcal{P}$ è nella classe $\mathsf{Space}(f(n))$ se e solo se esiste un algoritmo di complessità spaziale $O(f(n))$ che lo risolve. Sia $\mathsf{P} = \bigcup_{k\geq 0} \mathsf{Time}(n^k)$, $\mathsf{PSpace} = \bigcup_{k\geq 0} \mathsf{Space}(n^k)$, $\mathsf{ExpTime} = \bigcup_{k\geq 0} \mathsf{Time}(2^{n^k})$.

È facile vedere che $\mathsf{P} \subseteq \mathsf{PSpace}$ in quanto un algoritmo che impiega un tempo polinomiale può accedere al più a un numero polinomiale di locazioni di memoria. Si può provare inoltre che $\mathsf{PSpace} \subseteq \mathsf{ExpTime}$ (intuitivamente, assumendo per semplicità le locazioni di memoria binarie, n^c locazioni di memoria possono trovarsi in al più 2^{n^c} stati diversi). Non si sa se queste inclusioni siano strette (sono problemi aperti). Sappiamo invece che $\mathsf{P} \subset \mathsf{ExpTime}$, ossia che esistono problemi che possono essere risolti in tempo esponenziale ma non polinomiale, quindi provabilmente intrattabili (per esempio, le torri di Hanoi).

Notiamo che la definizione precedente è semi-formale, perché non essendoci un formato preciso per l'input parliamo genericamente di "dimensione".

Per essere più precisi, possiamo uniformare tutti i possibili input codificandoli come stringhe binarie.

Def. 4.3 Un *problema concreto* $\mathcal{P}$ è un problema il cui insieme di istanze è l'insieme delle stringhe binarie, ossia $\mathcal{P}\colon \{0,1\}^\star \to \{T, F\}$.

È equivalente considerare qualunque alfabeto con cardinalità almeno 2. Un problema astratto $\mathcal{P}$ può essere rappresentato in modo concreto tramite una *codifica*, ossia una funzione iniettiva:

$$c\colon I \to \{0,1\}^\star$$

Il problema concreto $c(\mathcal{P})$ è definito da $c(\mathcal{P})(x) = T$ se e solo se $x = c(i)$ e $\mathcal{P}(i) = T$, ossia assumiamo convenzionalmente che la soluzione sia falso sulle stringhe che non sono codifica di nessun input.

4.2 Classe NP

Introduciamo ora il significato della classe NP con degli esempi.

Consideriamo le formule su un insieme di variabili definite dalla seguente sintassi:

$$\begin{array}{lll} x & ::= & \ldots \\ \phi & ::= & x \mid \overline{\phi} \mid \phi_1 \wedge \phi_2 \mid \phi_1 \vee \phi_2 \mid \exists x.\phi \mid \forall x.\phi \end{array}$$

In particolare, le *formule in forma normale congiuntiva* (CNF) e le *formule quantificate* sono definite nel modo seguente:

l	$::=$	$x \mid \overline{x}$	letterale
c	$::=$	$l_1 \vee \ldots \vee l_n$	clausola
ϕ_{CNF}	$::=$	$c_1 \wedge \ldots \wedge c_n$	formula in CNF
ϕ_Q	$::=$	$\phi_{CNF} \mid \exists x.\phi_Q \mid \forall x.\phi_Q$	formula quantificata

Problema della soddisfacibilità SAT: data una formula in forma normale congiuntiva, determinare se esiste un'assegnazione di valori di verità alle variabili che la renda vera.

Problema delle formule quantificate: data una formula quantificata, determinare se è vera.

Entrambi i problemi possono essere facilmente risolti in tempo esponenziale (possiamo prendere come dimensione dell'istanza del problema il numero di variabili).

```
eval(phi)
  return eval(phi,empty)

eval(exists x.phi,env)
  return eval(phi,env.add(x,true)) or eval(phi,env.add(x,false))

eval(forall x.phi,env) =
  return eval(phi,env.add(x,true) and eval(phi,env.add(x,false))

...

sat(phi) //con variabili x_1 ... x_n
  return eval(exists x_1. ... exist x_n.phi)
```

Anzi sono in PSpace in quanto usano un numero lineare di locazioni di memoria.

Questo esempio mostra come spesso un algoritmo di decisione generi, in caso positivo, una "prova", detta *certificato*, che dimostra la verità della proprietà da verificare, per esempio, nel caso della soddisfacibilità, l'assegnazione di valori alle variabili che rende vera la formula. Mentre nel caso della soddisfacibilità verificare la validità di un certificato è facile, nel caso delle formule quantificate il "certificato" stesso consta di un numero esponenziale di assegnazioni di valori di verità a variabili. Questo suggerisce l'idea di usare come classificazione la difficoltà di *verificare* se un certificato è valido per un problema. Informalmente, definiamo NP la classe dei problemi che ammettono certificati verificabili in tempo polinomiale. Per esempio, dato che possiamo verificare in tempo polinomiale se un'assegnazione di valori alle variabili rende vera una formula in forma normale congiuntiva, il problema della soddisfacibilità è nella classe NP. Non possiamo dire altrettanto per il problema delle formule quantificate: non è noto se sia in NP, ma si congettura che non lo sia.

Un altro esempio: problema del *ciclo hamiltoniano* in un grafo non orientato. È un ciclo semplice che contiene ogni nodo (quindi passa esattamente una volta per ogni nodo). È facile vedere che questo problema è in NP (un certificato è un ciclo), mentre non è noto un algoritmo polinomiale che lo risolva.

Def. 4.4 Un *algoritmo di verifica* per un problema (astratto) $\mathcal{P} \subseteq I$ è un algoritmo $A\colon I \times C \to \{T, F\}$, dove C è un insieme di *certificati*, e $A(x, y) = T$ per qualche y se e solo se $x \in \mathcal{P}$.

Nel caso dei problemi concreti, si ha $I = C = \{0, 1\}^\star$.

Quindi un algoritmo verifica un problema se per ogni istanza con risposta positiva esiste un certificato valido, e non esiste per stringhe che non vi appartengono. Per esempio nel caso del grafo hamiltoniano se non lo è non è possibile esibire un certificato.

La classe NP è quindi la classe dei problemi che possono essere verificati da un algoritmo polinomiale. Più precisamente:

Def. 4.5 Un problema $\mathcal{P}$ è nella classe NP se e solo se esistono un algoritmo di verifica polinomiale A e una costante $k > 0$ tali che

$$\mathcal{P} = \{x \mid \exists y.A(x, y) = T, |y| = O(|x|^k)\}$$

```
sat_ver(phi,env)
  return eval(phi,env)
```

Equivalentemente, si può anche definire NP come la classe dei problemi per cui esiste un algorimo polinomiale *non deterministico* che li risolve, tale cioè che se la risposta è positiva ci sia almeno una possibile computazione che dia risposta positiva.

```
sat_non_det(phi) //con variabili x_1 ... x_n
  b_1 = true or b_1 = false
  ...
  b_n = true or b_n = false
  return eval(phi, x_1 -> b_1, ..., x_n -> b_n)
```

Legame con la verifica in tempo polinomiale (informalmente): un algoritmo non deterministico può sempre essere visto come composto di due fasi (vedi esempio sopra):

- una prima fase non deterministica che costruisce un certificato
- una seconda fase deterministica che controlla se questo è un certificato valido.

È facile verificare che $\mathsf{P} \subseteq \mathsf{NP}$, perché un algoritmo deterministico è un caso particolare di algoritmo non deterministico, oppure equivalentemente è un caso particolare di algoritmo di verifica in cui si ignora il certificato. Inoltre, la fase deterministica di verifica può essere eseguita in tempo polinomiale solo se il certificato ha dimensione polinomiale, quindi $\mathsf{NP} \subseteq \mathsf{PSpace}$. Per entrambe le inclusioni non si sa se siano proprie.

4.3 Problemi NP-completi

I problemi *NP-completi* sono i "più difficili" tra i problemi in NP. Ossia, se per qualcuno di questi problemi si trovasse un algoritmo polinomiale, se ne troverebbe uno per tutti quelli in NP, e quindi si dimostrerebbe che $\mathsf{P} = \mathsf{NP}$.

Infatti ognuno dei problemi in NP è *riducibile* a un problema NP-completo, nel senso formalizzato sotto.

Def. 4.6 Dati due problemi concreti $\mathcal{P}_1$ e $\mathcal{P}_2$, diciamo che $\mathcal{P}_1$ *è riducibile polinomialmente a* $\mathcal{P}_2$, e scriviamo $\mathcal{P}_1 \leq_{\mathsf{P}} \mathcal{P}_2$, se esiste una funzione $f\colon \{0,1\}^\star \to \{0,1\}^\star$, detta *funzione di riduzione*, calcolabile in tempo polinomiale, tale che, per ogni $x \in \{0,1\}^\star$, $x \in \mathcal{P}_1$ se e solo se $f(x) \in \mathcal{P}_2$.

Per esempio, il problema della soddisfacibilità è polinomialmente riducibile a quello delle formule quantificate.

Lemma 4.7 Dati due problemi concreti $\mathcal{P}_1$ e $\mathcal{P}_2$ tali che $\mathcal{P}_1 \leq_{\mathsf{P}} \mathcal{P}_2$, se $\mathcal{P}_2 \in \mathsf{P}$ allora anche $\mathcal{P}_1 \in \mathsf{P}$.

Dimostrazione Dato un algoritmo in tempo polinomiale A_2 che risolve $\mathcal{P}_2$, un algoritmo in tempo polinomiale A_1 che risolve $\mathcal{P}_1$ può essere ottenuto eseguendo, a partire da x, prima l'algoritmo in tempo polinomiale che calcola $f(x)$, poi $A_2(f(x))$. L'algoritmo A_1 risulta polinomiale. L'algoritmo A_1 risolve $\mathcal{P}_1$ in quanto $A_1(x) = A_2(f(x)) = \mathcal{P}_2(f(x)) = \mathcal{P}_1(x)$. □

Def. 4.8 Un problema concreto $\mathcal{P}$ si dice *NP-arduo* o *NP-difficile* (in inglese *NP-hard*) se per ogni problema $\mathcal{Q} \in \mathsf{NP}$ si ha $\mathcal{Q} \leq_{\mathsf{P}} \mathcal{P}$, si dice *NP-completo* se appartiene alla classe NP ed è NP-arduo. Indichiamo con NP-C la classe dei problemi NP-completi.

Nota: esistono problemi NP-ardui ma non appartenenti alla classe NP. Le nozioni di *hard* e *completo* si possono dare in generale relativamente a una qualunque classe di problemi.

Teorema 4.9 Se un qualunque problema NP-completo appartiene alla classe P, allora si ha $\mathsf{P} = \mathsf{NP}$, o, equivalentemente, se è $\mathsf{P} \neq \mathsf{NP}$, quindi esiste almeno un problema in NP non risolvibile in tempo polinomiale, allora nessun problema NP-completo è risolvibile in tempo polinomiale.

Dimostrazione Ovvio in base al precedente lemma. □

Quindi, per risolvere il problema aperto $\mathsf{P} \stackrel{?}{=} \mathsf{NP}$, basterebbe, per un qualunque problema in NP-C, trovare un algoritmo polinomiale o provare che non ne esiste uno.

Per dimostrare che un problema è in NP basta mostrare un algoritmo polinomiale non deterministico che lo risolve, o un algoritmo polinomiale che lo verifica. Come possiamo invece dimostrare che un problema è NP-completo? Una volta dimostrato che almeno un problema, sia $\mathcal{P}$, è NP-completo, possiamo provare che un altro lo è mostrando la riducibilità di $\mathcal{P}$ a questo

problema, infatti in questo modo anche il nuovo problema risulta NP-completo per la transitività della relazione di riducibilità. Bisogna quindi trovare un "primo" problema di cui dimostrare la NP-completezza.

Nota metodologica: provare che un problema è NP-completo è utile, perché in tal caso è opportuno cercare un algoritmo approssimato o ridursi a un sottoproblema per cui si possa trovare un algoritmo polinomiale.

Storicamente, il primo problema di cui è stata provata la NP-completezza (da Stephen Cook nel 1971) è il problema della soddisfacibilità.

Teorema 4.10 [Teorema di Cook] SAT è NP-completo.

Non diamo la dimostrazione del teorema di Cook. L'idea è la seguente: si definisce un algoritmo che, dato un problema $\mathcal{P} \in \mathsf{NP}$ e un input x per $\mathcal{P}$, costruisce una formula in forma normale congiuntiva che descrive un algoritmo non deterministico per $\mathcal{P}$, ossia la formula è soddisfacibile se e solo se l'algoritmo restituisce T.

Come anticipato, in genere le dimostrazioni di NP-completezza non si fanno in questo modo diretto, ma trovando un altro problema NP-completo che si riduca a quello del quale vogliamo provare la NP-completezza. Vediamo degli esempi.

Def. 4.11 Data una formula in 3CNF, ossia in CNF e tale che ogni clausola sia la disgiunzione di esattamente tre letterali, il *problema della 3-soddisfacibilità* $3SAT$ richiede di verificare se esiste un'assegnazione di valori di verità alle variabili che rende la formula vera.

È un caso particolare del problema di soddisfacibilità, in cui il numero di letterali in ogni clausola è esattamente tre. Nonostante questa limitazione si può provare che il problema rimane NP-completo. Questo problema è interessante perché è più facile da ridurre in un altro rispetto alla forma generale.

Teorema 4.12 $3SAT$ è NP-completo.

Dimostrazione L'appartenenza a NP è ovvia essendo un caso particolare di SAT. Per dimostrare che è NP-arduo, diamo una riduzione di SAT in $3SAT$, ossia mostriamo che ogni formula CNF può essere trasformata in una formula 3CNF. Anzitutto, scegliamo tre variabili nuove y_1, y_2, y_3 e aggiungiamo alla formula CNF sette nuove clausole che sono tutte le possibili combinazioni dei corrispodenti letterali tranne $\overline{y_1} \vee \overline{y_2} \vee \overline{y_3}$, in modo tale che l'unica assegnazione di valori a queste variabili che rende vera la loro congiunzione sia $y_1 = y_2 = y_3 = T$, ossia:

$$y_1 \vee y_2 \vee y_3,\ y_1 \vee y_2 \vee \overline{y_3},\ y_1 \vee \overline{y_2} \vee y_3,\ y_1 \vee \overline{y_2} \vee \overline{y_3},\ \overline{y_1} \vee y_2 \vee y_3,\ \overline{y_1} \vee y_2 \vee \overline{y_3},\ \overline{y_1} \vee \overline{y_2} \vee y_3.$$

Poi trasformiamo ogni clausola c nella formula in una congiunzione di clausole ognuna di esattamente tre letterali.

- Se $c = l$, si sostituisce c con $l \vee \overline{y_1} \vee \overline{y_2}$.
- Se $c = l_1 \vee l_2$, si sostituisce c con $l_1 \vee l_2 \vee \overline{y_1}$.
- Se $c = l_1 \vee l_2 \vee c'$ con c' disgiunzione di almeno un letterale:
  - se c' è un solo letterale non si fa nulla
  - altrimenti, si introduce una nuova variabile y_c, e si sostituisce c con

    $$y_c \vee c',\ l_1 \vee l_2 \vee \overline{y_c},\ \overline{l_1} \vee y_c \vee \overline{y_1},\ \overline{l_2} \vee y_c \vee \overline{y_1}.$$

    Le ultime tre clausole implicano che $l_1 \vee l_2$ è equivalente a y_c, quindi c è equivalente a $y_c \vee c'$. A questo punto, si riapplica induttivamente il procedimento a $y_c \vee c'$.

Per costruzione la congiunzione di tutte le nuove clausole è equivalente alla formula di partenza, e la trasformazione può essere effettuata in tempo polinomiale. Quindi $SAT \leq_{\mathsf{P}} 3SAT$. □

Si può provare che numerosi altri problemi sono NP-completi usando il problema della 3-soddisfacibilità. Vediamo due esempi: il problema della clique e il problema dell'insieme indipendente.

Def. 4.13 Una *clique* o *cricca* in un grafo non orientato $G = (V, E)$ è un insieme $V' \subseteq V$ di nodi tale che per ogni coppia di essi esiste l'arco che li collega, ossia il sottografo indotto da V' è completo. La *dimensione* di una clique è il numero dei suoi nodi. Il *problema della clique* richiede di trovare una clique di dimensione massima in un grafo. Il corrispondente problema di decisione richiede di determinare se nel grafo esiste una clique di dimensione k.

L'algoritmo banale consiste nell'esaminare tutti i possibili sottoinsiemi di nodi di dimensione k.

Teorema 4.14 Il problema della clique è NP-completo.

Dimostrazione Un certificato per il problema della clique è un sottoinsieme dei nodi di dimensione k. È facile verificare polinomialmente se il sottoinsieme è una clique, e questo prova che il problema è in NP. Per dimostrare che il problema è NP-arduo, diamo una riduzione di $3SAT$ in questo problema nel modo seguente. Data una formula 3CNF ϕ formata di k clausole, costruiamo il grafo G_ϕ nel modo seguente:

- un nodo per ogni occorrenza di letterale in una clausola, quindi $3k$ nodi
- un arco tra due occorrenze di letterali se sono in due clausole diverse e non sono uno la negazione dell'altro.

Mostriamo che ϕ è soddisfacibile se e solo se G_ϕ contiene una clique di k nodi.

$\Rightarrow$ Se ϕ è soddisfacibile, significa che esiste un'assegnazione di valori alle variabili tale che almeno un letterale per ogni clausola risulta vero. Consideriamo i nodi di G_ϕ corrispondenti ai letterali scelti. Esiste un arco che collega ogni coppia di questi nodi, perché sono in clausole diverse e non sono uno la negazione dell'altro, altrimenti non potrebbero essere contemporaneamente veri. Abbiamo quindi trovato una clique di dimensione k.

$\Leftarrow$ Se G_ϕ contiene una clique di k nodi, dato che non vi sono archi tra nodi che rappresentano letterali nella stessa clausola, sappiamo che la clique contiene esattamente un nodo per ogni clausola, siano $l_1, \ldots, l_k$ i corrispondenti letterali. Scegliamo un'assegnazione di valori alle variabili tale che $l_1, \ldots, l_k$ risultino veri, ciò è possibile in quanto sappiamo che tra di essi non vi sono due letterali che sono uno la negazione dell'altro, altrimenti non vi sarebbe un arco. Con questa assegnazione ogni clausola è soddisfatta, e quindi l'intera formula. □

Def. 4.15 Un *insieme indipendente* in un grafo $G = (V, E)$ è un insieme $V' \subseteq V$ di nodi tale che per ogni coppia di essi non esiste l'arco che li collega, ossia il sottografo indotto da V' non ha archi. La *dimensione* di un insieme indipendente è il numero dei suoi nodi. Il *problema dell'insieme indipendente* richiede di trovare un insieme indipendente di dimensione massima in un grafo. Il corrispondente problema di decisione richiede di determinare se nel grafo esiste un insieme indipendente di dimensione k.

Teorema 4.16 Il problema dell'insieme indipendente è NP-completo.

Dimostrazione Un certificato per il problema dell'insieme indipendente è un sottoinsieme dei nodi di dimensione k. È facile verificare polinomialmente se il sottoinsieme è un insieme indipendente, e questo prova che il problema è in NP. Per dimostrare che il problema è NP-arduo, definiamo una riduzione dal problema della clique nel modo seguente. Dato un grafo G consideriamo il grafo $\overline{G}$ che ha lo stesso insieme di nodi e tale che, per ogni coppia di nodi (u, v), in $\overline{G}$ esiste l'arco (u, v) se e solo se in G non esiste. È immediato vedere che un insieme di nodi definisce una clique in G se e solo se definisce un insieme indipendente in G'. □

Si conoscono ormai moltissimi problemi NP-completi, tra questi citiamo:

- commesso viaggiatore: dato un grafo non orientato completo pesato, trovare un ciclo hamiltoniano di costo minimo (nella versione di decisione stabilire se esiste un ciclo hamiltoniano di costo al più k)
- zaino: dato uno "zaino" con una certa capacità ed n oggetti a cui sono associati dei pesi e dei profitti, trovare il sottoinsieme di profitto massimo che sia possibile mettere nello zaino (nella versione di decisione trovare se esiste un sottoinsieme di profitto $\geq k$)
- copertura di vertici: dato un grafo non orientato $G = (V, E)$ trovare una *copertura di vertici* (*vertex cover*) di dimensione minima per G, ossia un insieme $V' \subseteq V$ di nodi tale che per ogni arco (u, v) in G almeno un estremo appartenga a V' (nella versione di decisione stabilire se esiste una copertura di vertici di dimensione k)
- colorazione: dato un grafo $G = (V, E)$ trovare il minimo k per cui esiste una *k-colorazione* dei nodi di G, ossia una funzione $c\colon V \to 1..k$ tale che $c(u) \neq c(v)$ se $(u, v) \in E$ (nella versione di decisione stabilire se esiste una k-colorazione).

A Relazioni d'ordine e di equivalenza

Def. A.1 [Ordine parziale] Un *ordine parziale* su un insieme X è una relazione $\leq$ su X che sia:

riflessiva $x \leq x$

antisimmetrica se $x \leq y$ e $y \leq x$ allora $x = y$

transitiva se $x \leq y$ e $y \leq z$ allora $x \leq z$.

Un *ordine parziale stretto* su X è una relazione $<$ su X che sia:

antiriflessiva $x \not< x$,

asimmetrica se $x < y$, allora $y \not< x$,

transitiva se $x < y$ e $y < z$ allora $x < z$.

Dato un ordine parziale $\leq$ a partire da questo possiamo definire il corrispondente ordine parziale stretto: $x < y$ se e solo se $x \leq y$ e $x \neq y$, e viceversa dato un ordine parziale stretto $<$ possiamo definire il corrispondente ordine parziale: $x \leq y$ se e solo se $x < y$ oppure $x = y$. Quindi è indifferente dare l'uno o l'altro.

Def. A.2 [Ordine totale] Un *ordine totale* su un insieme X è una relazione $\leq$ su X che sia un ordine parziale e inoltre sia:

totale $x \leq y$ oppure $y \leq x$.

Analogamente è possibile definire un *ordine totale stretto*. Un esempio tipico di ordine totale è l'ordinamento usuale sui numeri naturali, interi o reali, mentre un esempio tipico di ordine parziale è la relazione tra nodo padre e nodo figlio in un albero (ci sono nodi non confrontabili).

Def. A.3 [Equivalenza] Un'*equivalenza* su un insieme X è una relazione $\sim$ su X che sia:

riflessiva $x \sim x$

simmetrica se $x \sim y$ allora $y \sim x$

transitiva se $x \sim y$ e $y \sim z$ allora $x \sim z$.

La *classe di equivalenza* di $x \in X$ è l'insieme di tutti gli elementi equivalenti a x. Un elemento che appartiene a una certa classe di equivalenza si chiama anche *rappresentante* di tale classe. L'insieme delle classi di equivalenza è detto *insieme quoziente di X rispetto a $\sim$*.