DISCLAIMER

Il materiale contenuto nel drive è stato raccolto e richiesto tramite autorizzazione ai ragazzi frequentanti il corso di studi di Informatica dell'Università degli Studi di Salerno. Gli appunti e gli esercizi nascono da un uso e consumo degli autori che li hanno creati e risistemati per tanto non ci assumiamo la responsabilità di eventuali mancanze o difetti all'interno del materiale pubblicato.

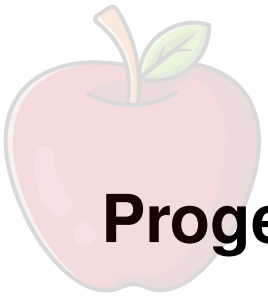
Il materiale sarà modificato aggiungendo il logo dell'associazione, in tal caso questo possa recare problemi ad alcuni autori di materiale pubblicato, tale persona può contattarci in privato ed elimineremo o modificheremo il materiale in base alle sue preferenze.

Ringraziamo eventuali segnalazioni di errori così da poter modificare e fornire il miglior materiale possibile a supporto degli studenti.

CoScienze
Associazione

UNIVERSITÀ DEGLI STUDI DI SALERNO

Corso di Laurea Triennale in informatica



Riassunto di

Progettazione di Algoritmi

Dario Mazza

Basato sulle lezioni di: *Prof. A. De Bonis*

Indice

Capitolo 1

Introduzione

In questo capitolo cominceremo con una discussione generale sui problemi computazionali e gli algoritmi che li risolvono, in particolare affronteremo il classico problema di *sorting*, proponendo due diversi algoritmi, che analizzeremo.

1.1 Algoritmi

Informalmente, un ***algoritmo*** è una qualsiasi procedura computazionale ben definita che prende in ***input*** uno o più valori e produce in ***output*** uno o più valori.

Un algoritmo può essere considerato anche uno strumento per risolvere un determinato ***problema computazionale***, quest'ultimo è una relazione tra input e output e l'algoritmo specifica la procedura computazionale da compiere affinché questa relazione sia rispettata.

Un algoritmo si definisce ***corretto*** se, per ogni istanza input, si ferma e produce un output. Un algoritmo scorretto potrebbe non fermarsi per qualche istanza input, o fermarsi producendo un output diverso da quello atteso.

È possibile progettare ***diversi algoritmi*** per risolvere uno stesso problema, infatti un particolare algoritmo può impiegare tempi diversi rispetto ad un altro, o usare spazi di memoria diversi. A tal proposito è utile avere un modo per ***confrontare tra loro diverse soluzioni*** per stabilire quale sia la migliore rispetto ad un certo criterio di efficienza. Inoltre è opportuno

valutare un algoritmo solo in base alle sue caratteristiche intrinseche e non a quelle del codice che lo implementa o della macchina su cui è eseguito.

Sulla base di ciò che è stato appena detto, si può intuire che il lavoro di un informatico non dovrebbe limitarsi al trovare soluzioni ai problemi, ma deve saper essere in grado anche di valutare le soluzioni in modo da stabilire se ci possono essere ***margini di miglioramento*** da un punto di vista prestazionale.

1.2 Efficienza di algoritmi

Un algoritmo è efficiente se, quando è implementato, viene eseguito velocemente su istanze reali.

Ci rendiamo conto che tale definizione è molto vaga, dato che non chiarisce dove viene eseguito l'algoritmo e quanto veloce deve essere la sua esecuzione. Inoltre non chiarisce cosa è un'istanza input reale e non fa capire come la velocità di esecuzione dell'algoritmo vari al crescere della dimensione dell'input.

Per una ***definizione concreta*** di efficienza occorre che ci:

- sia indipendente dal processore
- sia indipendente dal tipo di istanza
- dia una misura di come aumenta il tempo di esecuzione al crescere della dimensione dell'input

1.2.1 Algoritmo Brute Force

Per molti problemi non banali, esiste un naturale algoritmo di ***forza bruta*** che controlla ogni possibile soluzione. Tipicamente questo algoritmo impiega tempo 2^n per input di dimensione n, quindi un tempo esponenziale che ovviamente non è accettabile.

Esempio: Vogliamo ordinare in modo crescente un array di N elementi distinti. Soluzione con algoritmo di forza bruta:

permutiamo i numeri ogni volta in modo diverso, fino ad ottenere una permutazione ordinata.
NB: Nel caso pessimo avremo $N!$ permutazioni ($N! > 2^N \; per \; N > 3$)

1.2.2 Tempo polinomiale

Un'importante proprietà che vogliamo che sia soddisfatta per un algoritmo efficiente è la seguente:

Proprietà: Quando la dimensione dell'input raddoppia, l'algoritmo dovrebbe risultare più lento solo di un fattore costante c.

Convinciamoci che questa proprietà è soddisfatta per un algoritmo di ordinamento molto efficiente, ovvero il ***MergeSort***

MergeSort: Per definizione il MergeSort ha tempo $c \cdot n \cdot log_2 n$ con n corrispondente alla dimensione dell'input.

Verifichiamo che per $2n$ l'algoritmo ha tempo di esecuzione doppio rispetto a quanto detto in precedenza (più lento solo di un fattore costante c)

$$c \cdot n \cdot log_2 n \Rightarrow c \cdot 2n \cdot log_2(2n) = 2 \cdot c \cdot n \cdot (log_2 n + 1) \tag{1.1}$$

Sostituiamo $2n$ al posto di n, notiamo che per le proprietà dei logaritmi $log_2(2n)$ può essere scritto come $log_2 n + log_2 2$, quindi questa quantità sarà uguale a $log_2 n + 1$

$$2 \cdot c \cdot n \cdot (log_2 n + 1) = 2 \cdot c \cdot n \cdot log_2 n + 2 \cdot c \cdot n \tag{1.2}$$

Svolgiamo il prodotto e otteniamo l'equazione 1.2

$$2 \cdot c \cdot n \cdot log_2 n + 2 \cdot c \cdot n \leq 2 \cdot c \cdot n \cdot log_2 n + 2 \cdot c \cdot n \cdot log_2 n \tag{1.3}$$

Notiamo che sostituendo $2 \cdot c \cdot log_2 n$ al posto di $2 \cdot c \cdot n$ otterremo una quantità sicuramente maggiore o uguale al primo membro per ogni $n > 1$

$$2 \cdot c \cdot n \cdot log_2 n + 2 \cdot c \cdot n \cdot log_2 = 4 \cdot c \cdot n \cdot log_2 n \tag{1.4}$$

Infine sommando i termini simili deduciamo che raddoppiando la dimensione dell'input n, l'algoritmo rallenta solo di un fattore $c = 4$, quindi la **proprietà definita precedentemente è soddisfatta.**

Verifichiamo che questa proprietà valga anche per un algoritmo di forza bruta applicato ad un problema di ordinamento, che per una dimensione dell'input uguale a n impiega tempo $c \cdot (n-1) \cdot n!$

$$c \cdot (n-1) \cdot n! = c \cdot (2n-1) \cdot (2n)! \tag{1.5}$$

Sostituiamo $2n$ al posto di n otteniamo $c \cdot (2n-1) \cdot (2n)!$

$$c \cdot (2n-1) \cdot (2n)! = c \cdot (2n-1) \cdot (2n \cdot (2n-1) \cdot \ldots \cdot (n+1) \cdot n!) \tag{1.6}$$

Svolgiamo il fattoriale di $2n$ riportando tutti i termini fino a $n!$

$$c \cdot (2n-1) \cdot (2n \cdot (2n-1) \cdot \ldots \cdot (n+1) \cdot n!) > c \cdot 2 \cdot (n-1) \cdot n! \cdot n! \tag{1.7}$$

Adesso al posto di $(2n-1)$ sostituiamo $(2n-2)$ e raggruppiamo per 2, ottenendo $2 \cdot (n-1)$. Successivamente rimpiazziamo $2n \cdot (2n-1) \cdot \ldots \cdot (n+1)$ con $n!$: alla fine otterremo $c \cdot 2 \cdot (n-1) \cdot n! \cdot n!$ che è sicuramente una quantità maggiore del primo membro della *disequazione 1.7*

$$c \cdot 2 \cdot (n-1) \cdot n! \cdot n! = (2n!) \cdot c \cdot (n-1) \cdot n! = \tag{1.8}$$

Riscriviamo il secondo membro della *disequazione 1.7* in questo modo e come risultato avremo il tempo dell'algoritmo di forza bruta per input grande n moltiplicato per $(2n!)$, ***che non è una costante***. Quindi abbiamo dimostrato che l'algoritmo di forza brutta non soddisfa la proprietà ambita.

Di conseguenza possiamo dire che un algoritmo impiega tempo polinomiale se quando la dimensione dell'input raddoppia, l'algoritmo risulta più lento solo di un fattore costante c.

> In particolare diremo che un algoritmo ha tempo di esecuzione polinomiale se esistono ***due costanti*** $c > 0$ e $d > 0$ tali che su ciascun input di dimensione n, l'algoritmo sarà ***limitato superiormente dalla funzione*** $c \cdot n^d$

Se si passa da un input di dimensione n ad uno di dimensione $2n$ allora il tempo di esecuzione passa da $c \cdot n^d$ a $c \cdot (2n)^d = c \cdot 2^d \cdot n^d$, dove 2^d è una costante

1.2.3 Analisi del caso pessimo

Come è stato detto in precedenza, il tempo di esecuzione di algoritmo cresce al crescere della dimensione dell'input, ma per analizzare la qualità di un algoritmo quale input si deve prendere in considerazione? Tra i possibili input si può effettuare:

- Analisi del caso pessimo: consiste nell'ottenere un bound sul ***più grande tempo di esecuzione possibile*** per tutti gli input di una certa dimensione n.

- Analisi del caso medio: consiste nell'ottenere un bound al tempo di esecuzione su ***random*** in funzione di una distribuzione dell'input (richiede la conoscenza della distribuzione).

Supposto questo possiamo dare una ***nuova definizione di efficienza***:

> *Un algoritmo è efficiente se il suo tempo di esecuzione nel caso pessimo è polinomiale.*

Nonostante ciò, se considerassimo un algoritmo polinomiale con c e d molto grandi, i tempi di esecuzione sarebbero ***estremamente lenti***, ma per fortuna problemi che possono essere risolti con algoritmi che vengono eseguiti in tempo polinomiale hanno un tempo di esecuzione che ***cresce in maniera moderata*** (c e d molto piccoli).

In base a quanto detto finora ci sembra di capire che un algoritmo che ha tempo di esecuzione esponenziale non sia mai usabile per la sua scarsa efficienza, ma ciò non è propriamente vero. Come detto precedentemente, un algoritmo che ha tempo di esecuzione polinomiale, con le constanti c e d molto grandi, si comporta in maniera inefficiente. Analogamente ci sono algoritmi esponenziali che sono efficienti e usabili nella pratica poiché il ***caso pessimo si verifica molto raramente.***[1]

[1] Ad esempio l'algoritmo del simplesso è molto usato per problemi di programmazione lineare

Capitolo 2

Analisi della complessità

La tabella *2.1* ci fa comprendere che analizzare la complessità di un algoritmo è di rilevante importanza, dato che scegliere un algoritmo meno efficiente potrebbe comportare a tempo di esecuzione estremamente dilatati. **N.B:** la formazione del pianeta Terra risale a circa $4,54 \times 10^9$ anni fa.

	n	$nlog_2n$	n^2	n^3	1.5^n	2^n	$n!$
$n = 10$	< 1 sec	< 1 sec	< 1 sec	< 1 sec	< 1 sec	< 1 sec	4 sec
$n = 30$	< 1 sec	< 1 sec	< 1 sec	< 1 sec	< 1 sec	18 min	10^{25} years
$n = 50$	< 1 sec	< 1 sec	< 1 sec	< 1 sec	11 min	36 years	very long
$n = 100$	< 1 sec	< 1 sec	< 1 sec	1 sec	12 892 years	10^{17} years	very long
$n = 1\,000$	< 1 sec	< 1 sec	1 sec	18 min	very long	very long	very long
$n = 10\,000$	< 1 sec	< 1 sec	2 min	12 days	very long	very long	very long
$n = 100\,000$	< 1 sec	2 sec	3 hours	32 years	very long	very long	very long
$n = 1\,000\,000$	1 sec	20 sec	12 days	31 710 years	very long	very long	very long

Tabella 2.1: *La tabella riporta i tempi di esecuzione su input di dimensione crescente, per un processore che esegue un milione di istruzioni al secondo.*

2.1 Analisi Insertion-Sort

In questo capitolo inizieremo con l'analisi analitica degli algoritmi. In questo caso cominceremo ad analizzare la complessità computazionale dell'algoritmo di Insertion-Sort, codificato come segue in linguaggio C.

```
for(j=1;j<n;j++) //n e' la lunghezza di a
{
    elemDaIns=a[j];
    i=j-1;
    //cerca il posto per a[j]
    while(i >= 0 && a[i] > elemDaIns)
    {
        //shifto a destra gli elementi maggiori
        a[i+1]=a[i];
        i--;
    }
    a[i+1]=elemDaIns;
}
```

Supponendo che il primo elemento sia già ordinato, l'indice j indica l'elemento corrente che deve essere inserito in maniera ordinata nella porzione di array già ordinata ($a[0...j-1]$). L'indice j si sposta da sinistra a destra su tutto l'array: ad ogni iterazione del ciclo for l'elemento $a[j]$ è copiato in una variabile temporanea. Quindi cominciando dalla posizione $j-1$, gli elementi sono spostati ad uno ad uno di una posizione a destra finché non viene trovata una giusta posizione per $a[j]$.

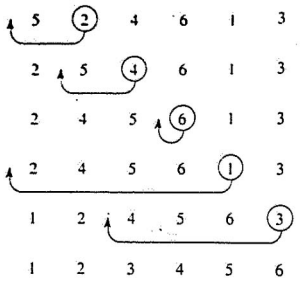


Figura 2.1: *L'elaborazione di Insertion-Sort sull'array A<5,2,4,6,1,3>. La posizione di indice j è indicata con un cerchio*

2.1.1 Calcolo del tempo di esecuzione

Analizziamo l'algoritmo passo passo, e cerchiamo di calcolare il tempo di esecuzione per ogni istruzione. Per comodità considereremo $j = 1$ come indice della prima posizione dell'array, e di conseguenza n come indice dell'ultima posizione. Inoltre, chiameremo t_j il numero di volte che il test del ciclo *while* verrà eseguito per la j-esima iterazione del ciclo *for* esterno.

INSERTION-SORT(A)		*costo*	*n° di volte*
1	**for** $j \leftarrow 2$ **to** *length*[A]	c_1	n
2	**do** *key* $\leftarrow A[j]$	c_2	$n-1$
3	▷ Si inserisce $A[j]$ nella		
	▷ sequenza ordinata $A[1\,..\,j-1]$	0	$n-1$
4	$i \leftarrow j-1$	c_4	$n-1$
5	**while** $i > 0$ e $A[i] >$ *key*	c_5	$\sum_{j=2}^{n} t_j$
6	**do** $A[i+1] \leftarrow A[i]$	c_6	$\sum_{j=2}^{n}(t_j-1)$
7	$i \leftarrow i-1$	c_7	$\sum_{j=2}^{n}(t_j-1)$
8	$A[i+1] \leftarrow$ *key*	c_8	$n-1$

Il tempo di esecuzione dell'algoritmo è la somma dei tempi di esecuzione delle singole istruzioni che lo compongono: un'istruzione impiegherà un tempo $c \cdot n$ dove c è il costo per eseguire una singola istruzione (dipendente dalla macchina) e n il numero di volte che verrà eseguita quest'ultima.

Per calcolare T_n, il tempo di esecuzione di Insertion-Sort, si sommano i prodotti delle colonne dei *costi* e delle *volte*, ottendendo:

$$\begin{aligned} T_n = & c_1 n + c_2(n-1) + c_4(n-1) + c_5 \sum_{j=2}^{n} t_j + c_6 \sum_{j=2}^{n}(t_j - 1) \\ & + c_7 \sum_{j=2}^{n}(t_j - 1) + c_8(n-1) \end{aligned} \tag{2.1}$$

Anche per input della stessa dimensione (in questo caso rappresentata dal numero di elementi dell'array) un algoritmo può dipendere da ***quale tipo*** di input è dato. Nel nostro esempio, il caso migliore è identificato dall'array già ordinato: per ogni $j = 2, 3 \ldots n$, si trova che $a[i] \leq elemDaIns$ ("key" in pseudocodice), con i che parte da $j - 1$.

Quindi nel ***caso migliore***, l'istruzione *while* sarà eseguita una sola volta, ossia $t_j = 1$ per $j = 2, 3 \dots n$, quindi avremo un tempo di esecuzione:

$$\begin{aligned} T_n &= c_1 n + c_2(n-1) + c_4(n-1) + c_5(n-1) + c_8(n-1) \\ &= (c_1 + c_2 + c_4 + c_5 + c_8)n - (c_2 + c_4 + c_5 + c_8) \end{aligned}$$

Quest'ultimo tempo di esecuzione può essere espresso come $an + b$, con a e b costanti dipendenti dai costi c_i, di conseguenza diremo che T_n in questo caso avrà tempo ***lineare***.

Contrariamente se l'array è ordinato in senso decrescente, allora si ha il *caso peggiore*. Infatti si dorvà confrontare ogni elemento $a[j]$ con ogni elemento presente nella sezione $a[1 \dots j-1]$ del vettore. Dunque avremo che $t_j = j$ per $j = 2, 3, \dots, n$. Ricordando che:

$$\sum_{j=1}^{n} j = \frac{n(n+1)}{2} \tag{2.2}$$

$$\sum_{j=1}^{n}(j-1) = \sum_{j=1}^{n} j - \sum_{j=1}^{n} 1 = \frac{n(n+1)}{2} - n = \frac{n(n-1)}{2} \tag{2.3}$$

Possiamo dedurre che le sommatorie presenti nell'equazione 2.1 si possono riscrivere in questo modo:

$$\sum_{j=2}^{n} j = \frac{n(n+1)}{2} - 1$$

$$\sum_{j=2}^{n}(j-1) = \frac{n(n-1)}{2}$$

Ne consegue che il tempo di esecuzione T_n sarà uguale a:

$$\begin{aligned} T_n = & c_1 n + c_2(n-1) + c_4(n-1) + c_5\left(\frac{n(n+1)}{2} - 1\right) + \\ & c_6\left(\frac{n(n-1)}{2}\right) + c_7\left(\frac{n(n-1)}{2}\right) + c_8(n-1) \end{aligned} \tag{2.4}$$

$$T_n = (\frac{c_5}{2} + \frac{c_6}{2} + \frac{c_7}{2})n^2 + (c_1 + c_2 + c_4 + \frac{c_5}{2} - \frac{c_6}{2} - \frac{c_7}{2} + c8)n + \\ - (c_2 + c_4 + c_5 + c_8) \tag{2.5}$$

Svolgendo i prodotti ed effettuando gli opportuni raccoglimenti otterremo l'equazione 2.5, che può essere espressa come an^2+bn+c, per costanti a, b, c che dipendono ancora dai costi di c_i; risulta quindi una ***funzione quadratica*** di n.

2.2 Ordine di grandezza

Nell'analizzare la complessità dell'algoritmo di Insertion-Sort abbiamo effettuato delle astrazioni per semplificarci il lavoro, infatti abbiamo ignorato il valore delle costanti c_i e a, b, c, che dipendono da molti fattori come ad esempio il tipo di macchina che esegue il programma.

Si può fare un'ulteriore operazione di astrazione considerando il ***tasso di crescita*** o ***ordine di grandezza*** del tempo di esecuzione. A tale scopo, si considera solo il termine principale ("dominante") di una formula per esempio an^2, dato che i termini di grado minore sono relativamente poco significativi per valori di n abbastanza grandi. Inoltre è possibile non considerare le costanti moltiplicative, poiché al crescere dell'input diventano sempre meno significative per il calcolo della complessità, quindi diremo che il tempo di esecuzione di Insertion-Sort ha ordine di grandezza n^2.

2.2.1 Notazione asintotica

Le notazioni usate per descrivere il tempo asintotico di esecuzione di un algoritmo sono definite in termini di funzioni il cui dominio è l'insieme dei numeri naturali $\mathbb{N} = \{0, 1, 2, \ldots\}$. Queste notazioni sono comode per descrivere la funzione $T(n)$ del tempo di esecuzione nel casi peggiore, definita solo su dimensioni intere dell'input.

Notazione O

> $T(n)$ è $O(f(n))$ se esistono due costanti $c > 0$ ed $n_o \geq 0$ tali che per per ogni $n \geq n_0$ si ha $0 \leq T(n) \leq c \cdot f(n)$

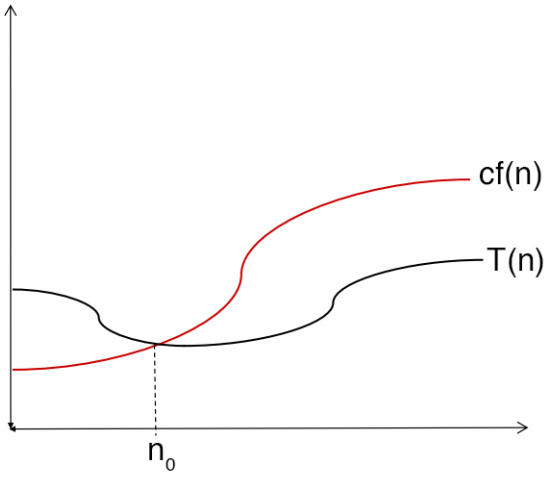


Figura 2.2: *Esempio di "Big O Notation"*

La notazione O si usa per dare un ***limite asintotico superiore*** ad una funzione a meno di un fattore costante: per tutti i valori $n \geq n_0$ il valore della funzione $T(n)$ coincide o si trova sotto $c \cdot f(n)$.

Usando la notazione O si può spesso descrivere il tempo di esecuzione di un algoritmo semplicemente analizzando la struttura complessiva dell'algoritmo; per esempio, per la struttura ciclica doppiamente annidata dell'algoritmo di Insertion-Sort, si ha un limite superiore $O(n^2)$ del tempo di esecuzione nel caso peggiore.

Notazione Ω

$T(n)$ è $\Omega(f(n))$ se esistono due costanti $c > 0$ ed $n_o \geq 0$ tali che per per ogni $n \geq n_0$ si ha $T(n) \geq c \cdot f(n) \geq 0$

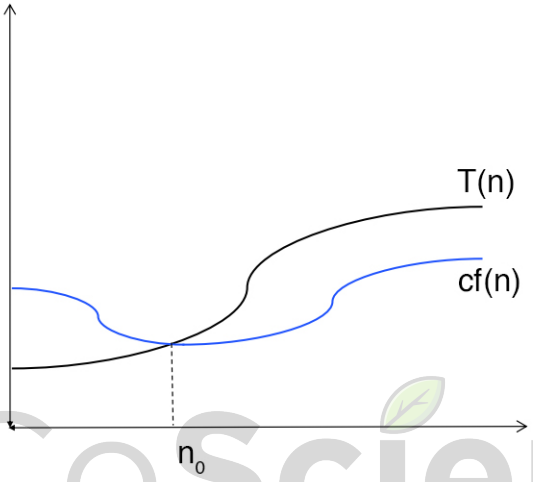


Figura 2.3: *Esempio di "Omega Notation"*

Così come la notazione O fornisce un limite asintotico superiore ad una funzione, la notazione Ω fornisce un ***limite asintotico inferiore***: per tutti i valori $n \geq n_0$ il valore della funzione $T(n)$ coincide o si trova sopra $c \cdot f(n)$

Notazione Θ

$T(n)$ è $\Theta(f(n))$ se $T(n)$ è sia $O(f(n))$ che $\Omega(f(n))$

Una funzione $T(n)$ appartiene all'insieme $\Theta(f(n))$ se esistono le costanti positive c' e c'' tali che $T(n)$ sia compresa tra $c' \cdot f(n)$ e $c'' \cdot f(n)$ per valori di n sufficientemente grandi.

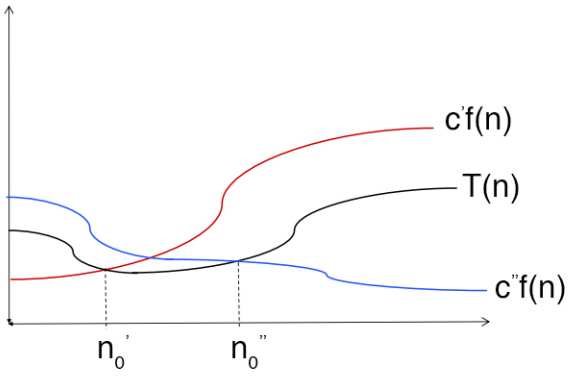


Figura 2.4: *Esempio di "Theta Notation"*

La definizione di $\Theta(f(n))$ richiede che $T(n)$ sia ***asintoticamente non negativa***, cioè richiede che $T(n)$ sia non negativa tutte le volte che n è sufficientemente grande. Analogamente anche la funzione $f(n)$ deve essere non negativa.

Si noti che $T(n) = \Theta(f(n))$ implica che $T(n) = O(f(n))$ in quanto la notazione Θ è una notazione più forte della notazione O (in termini insiemistici $\Theta(f(n)) \subseteq O(f(n))$).

2.3 Analisi asintotica di funzioni polinomiali

Sia $a_0 + a_1 n + \ldots + a_d n^d$ una funzione polinomiale $p(n)$, se $a_d > 0$ allora $p(n) = \Theta(n^d)$

Per dimostrare che $p(n) = \Theta(n^d)$, dobbiamo dimostrare sia che $p(n) = O(n^d)$ e sia che $p(n) = \Omega(n^d)$, a tal proposito divideremo la dimostrazione in due parti.

Dimostrazione $O(n^d)$ Dobbiamo trovare due costanti $c > 0$ e $n_0 \geq 0$ tali che $a_0 + a_1 n + \ldots + a_d n^d \leq c \cdot n^d$ per ogni $n \geq n_0$.

Dato che per ipotesi sappiamo che solo il coefficiente $a_d > 0$, di conseguenza possiamo scrivere la seguente disequazione:

$$a_0 + a_1 n + \ldots + a_d n^d \leq |a_0| + |a_1| n + |a_2| n^2 + \ldots + |a_d| n^d$$

La disequazione è corretta, poiché prendendo i valori assoluti di tutti i coefficienti avremo una quantità sicuramente maggiore o uguale del polinomio di partenza (ci possono essere coefficienti negativi che con il valore assoluto diventano positivi, e quindi maggiori).

$$|a_0| + |a_1| n + |a_2| n^2 + \ldots + |a_d| n^d \leq (|a_0| + |a_1| + \ldots + |a_d|) n^d$$

Anche in questo caso la disequazione è rispettata per ogni $n \geq 1$, poiché moltiplicando tutti i coefficienti per il termine di grado maggiore, otteniamo una quantità più grande.

Abbiamo trovato quindi le costanti $n_0 = 1$ e $c = (|a_0| + |a_1| + \ldots + |a_d|)$ tali che $a_0 + a_1 n + \ldots + a_d n^d \leq c \cdot n^d$ per ogni $n \geq n_0$, ossia abbiamo dimostrato che la nostra funzione polinomiale è $O(n^d)$.

Dimostrazione $\Omega(n^d)$ Dimostriamo che $a_0 + a_1 n + \ldots + a_d n^d$ è anche $\Omega(n^d)$:

$$a_d n^d + a_1 n + \ldots + a_0 \geq a_d n^d - (|a_0| + |a_1| n + \ldots + |a_{d-1}| n^{d-1}) \tag{2.6}$$

Il secondo membro è minore o uguale al primo dato che sottraiamo da $a_d n^d$ tutti i termini a_i presi con il valore assoluto, e quindi avremo una quantità minore. Poco fa abbiamo dimostrato che un polinomio di grado d è $O(n^d)$, ciò implica che $|a_0| + |a_1| n + \ldots + |a_{d-1}| n^{d-1} = O(n^{d-1})$, quindi esisteranno due costanti $c' > 0$ e $n'_0 \geq 0$ tali che:

$$|a_0| + |a_1| n + \ldots + |a_{d-1}| n^{d-1} \leq c' n^{d-1} \tag{2.7}$$

Quindi partendo dalla disequazione 2.6 possiamo scrivere la seguente disuguaglianza:

$$a_d n^d - (|a_0| + |a_1| n + \ldots + |a_{d-1}| n^{d-1}) \geq a_d n^d - c' n^{d-1} \tag{2.8}$$

Tale disequazione è verificata perché sapendo per la 2.8 che $c'n^{d-1} \geq |a_0| + |a_1|n + \ldots + |a_{d-1}|n^{d-1}$, stiamo sottraendo da $a_d n^d$ al secondo membro una quantità ***maggiore*** rispetto alla quantità sottratta da $a_d n^d$ al primo membro, di conseguenza il primo membro è maggiore o uguale al secondo.

Inoltre sfruttando la 2.6 possiamo affermare che il primo membro della 2.8 è minore o uguale a $a_d n^d + a_1 n + \ldots + a_0$, pertanto possiamo asserire che

$$a_0 + a_1 n + \ldots + a_d n^d \geq a_d n^d - c' n^{d-1} \tag{2.9}$$

Per dimostrare che $a_0 + a_1 n + \ldots a_d n^d = \Omega(n^d)$ dobbiamo trovare due costanti $n_0 \geq 0$ e $c > 0$ tali che $a_0 + a_1 n + \ldots a_d n^d \geq c n^d$ per ogni $n \geq n_0$.

Precedentemente abbiamo dimostrato che esistono due costanti $c' > 0$ e $n'_0 \geq 0$ tali che $a_0 + a_1 n + \ldots + a_d n^d \geq a_d n^d - c' n^{d-1}$ per ogni $n \geq n'_0$. Riscriviamo il secondo membro della disequazione in questo modo:

$$a_d n^d - c' n^{d-1} = \left(a_d - \frac{c'}{n}\right) n^d \tag{2.10}$$

Supponiamo di avere una costante $n''_0 > 0$ allora dato che $\left(a_d - \frac{c'}{n}\right)$ è crescente (al crescere di n, $\frac{c'}{n}$ diventa una quantità sempre più piccola) possiamo constatare che per ogni $n \geq n''_0$ avremo:

$$\left(a_d - \frac{c'}{n''_0}\right) \leq \left(a_d - \frac{c'}{n}\right) \tag{2.11}$$

Ridenominiamo il primo membro della disuguaglianza precedente in $c'' = \left(a_d - \frac{c'}{n''_0}\right)$, avremo quindi

$$c'' \leq \left(a_d - \frac{c'}{n}\right) \quad \text{per ogni} \quad n \geq n''_0 \tag{2.12}$$

Siccome $a_d > 0$ esisteranno certamente valori di n per cui $\left(a_d - \frac{c'}{n}\right) > 0$, troviamoli:

$$\left(a_d - \frac{c'}{n}\right) > 0 \Rightarrow n > \frac{c'}{a_d} \tag{2.13}$$

Quindi per ogni $n > \frac{c'}{a_d}$ si ha $\left(a_d - \frac{c'}{n}\right) > 0$.

Supposto questo possiamo porre la costante $n_0'' = \frac{2c'}{a^d}$ (rimaniamo un pò "larghi"), e di conseguenza possiamo trovare il valore della costante c'':

$$c'' = \left(a_d - \frac{c'}{n_0''}\right) = a_d - \frac{c'}{\frac{2c'}{a^d}} = a_d - \frac{a_d}{2} = \frac{a_d}{2}$$

In conclusione ***abbiamo trovato le due costanti*** $c'' > 0$ e $n_0'' \geq 0$ tali che $c'' \leq \left(a_d - \frac{c'}{n}\right)$ per ogni $n \geq n_0''$, e siccome

$$c'' \leq \left(a_d - \frac{c'}{n}\right) \Rightarrow c''n^d \leq \left(a_d - \frac{c'}{n}\right) n^d$$

possiamo sfruttare la disuguaglianza 2.9 per asserire che

$$a_0 + a_1 n + \ldots + a_d n^d \geq \left(a_d - \frac{c'}{n}\right) \geq c''n^d$$

per ogni $n \geq max\{n_0', n_0''\}$ dove $c'' = \frac{a_d}{2}$ e $n_0'' = \frac{2c'}{a_d}$.

2.3.1 Esempio svolto: analisi di un tempo quadratico

Coppia di punti più vicina Data una lista (o un array) di n punti nel piano $(x_1, y_1), \ldots, (x_n, y_n)$ vogliamo trovare la coppia di punti più vicina.

Soluzione $O(n^2)$ Calcoliamo la distanza tra tutte le possibili coppie di punti, in modo da stabilire quale sia la minore.

```
min ← (x1 - x2)² + (y1 - y2)²
for i = 1 to n {
   for j = i+1 to n {
      d ← (xi - xj)² + (yi - yj)²
      Se (d < min)
         min ← d
   }
}
```

Non c'è bisogno di estrarre la radice quadrata per effettuare i confronti tra le distanze.

Analisi Il for esterno verrà eseguito n volte più tutte le iterazioni del for interno, analizziamo il for interno

Chiamiamo t_i il numero di iterazioni del for interno all'i-esima iterazione del for esterno. In totale il for interno viene iterato $t_1 + t_2 + \ldots + t_n$, quindi $t_i = (n - i)$. Sommando t_i per tutte le iterazioni del for esterno si ha

$$\sum_{i=1}^{n}(n-i) = \frac{n(n-1)}{2}$$

Dunque, siccome la singola esecuzione del corpo del for interno richiede tempo pari ad una costante c, il tempo richiesto da tutte le iterazioni di tale for sarà $c\left(\frac{n(n-1)}{2}\right) = \Theta(n^2)$.

Quindi il tempo totale sarà $\Theta(n) + \Theta(n^2) = \Theta(n^2)$.

Capitolo 3

Proprietà notazione asintotica

3.1 Richiamo su alcune nozioni

Nei paragrafi seguenti saranno presenti alcuni richiami su nozioni matematiche che ci serviranno per effettuare dimostrazioni sulla complessità più avanti.

3.1.1 Richiamo sui logaritmi

1. $log_a x = \frac{log_b x}{log_b a}$
2. $log_a(xy) = log_a x + log_a y$
3. $log_a x^k = k \cdot log_a x$
4. $log_a x = \frac{1}{log_x a}$
5. $log_a(\frac{1}{x}) = -log_a x$
6. $log_a(\frac{x}{y}) = log_a x - log_a y$

3.1.2 Richiamo sulla parte intera

Parte intera inferiore

La parte intera inferiore di un numero x è denotata con $\lfloor x \rfloor$ ed è definita come quell'unico intero per cui vale

$$x - 1 < \lfloor x \rfloor \leq x$$

In altre parole , $\lfloor x \rfloor$ è il più grande intero minore o uguale di x.

Proprietà 1 L'intero più piccolo strettamente maggiore di x è $\lfloor x \rfloor + 1$.

Banalmente si può dimostrare dalla definizione di $\lfloor x \rfloor$, infatti da $x - 1 < \lfloor x \rfloor \leq x$, la prima disequazione implica $x < \lfloor x \rfloor + 1$, mentre la seconda $\lfloor x \rfloor \leq x$. Queste due disequazioni implicano la proprietà 1.

Proprietà 2 $\left\lfloor \frac{\lfloor \frac{a}{b} \rfloor}{c} \right\rfloor = \left\lfloor \frac{a}{bc} \right\rfloor$ per a, b, c interi con $b > 0$ e $c > 0$.

Parte intera superiore

La parte intera superiore di un numero x è denotata con $\lceil x \rceil$ ed è definita come quell'unico intero per cui vale che

$$x \leq \lceil x \rceil < x + 1$$

In altre parole $\lceil x \rceil$ è il più piccolo intero maggiore o uguale di x.

Proprietà 1 L'intero più piccolo strettamente minore di x è $\lceil x \rceil - 1$.

Banalmente si può dimostrare dalla definizione di $\lceil x \rceil$, infatti da $x \leq \lceil x \rceil < x + 1$, la seconda disequazione implica $\lceil x \rceil + 1 < x$, mentre la seconda $x \leq \lceil x \rceil$. Queste due disequazioni implicano la proprietà 1.

Proprietà 2 $\left\lceil \frac{\lceil \frac{a}{b} \rceil}{c} \right\rceil = \left\lceil \frac{a}{bc} \right\rceil$ per a, b, c interi con $b > 0$ e $c > 0$.

3.2 Regole per la notazione asintotica

1. $d(n) = O(f(n)) \Rightarrow ad(n) = O(f(n)), \forall a > 0$

   Es: $logn = O(n) \Rightarrow 7logn = O(n)$

2. $d(n) = O(f(n)), e(n) = O(g(n)) \Rightarrow d(n) + e(n) = O(f(n) + g(n))$

   Es: $logn = O(n), \sqrt{n} = O(n) \Rightarrow logn + \sqrt{n} = O(n)$

3. $d(n) = O(f(n)), e(n) = O(g(n)) \Rightarrow d(n)e(n) = O(f(n)g(n))$

   Es: $logn = O(\sqrt{n}), \sqrt{n} = O(\sqrt{n}) \Rightarrow \sqrt{n} \cdot logn = O(n)$

4. $d(n) = O(f(n)), f(n) = O(g(n)) \Rightarrow d(n) = O(g(n))$

   Es: $logn = O(\sqrt{n}), \sqrt{n} = O(n) \Rightarrow logn = O(n)$

5. $f(n) = a_d n^d + \ldots + a_1 n + a_0 \Rightarrow f(n) = O(n^d)$

   Es: $5n^7 + 6n^4 + 3n^3 + 100 = O(n^7)$

6. $n^x = O(a^n), \forall x > 0, a > 1$

   Es: $n^{100} = O(2^n)$

Le prime 5 proprietà mostrate sono valide anche nel caso in cui sostituissimo la O con Ω o Θ. Dimostriamone alcune...

Dimostrazione proprietà 1

$d(n) = O(f(n)), f(n) = O(g(n)) \Rightarrow d(n) = O(g(n))$

Se $d(n) = O(f(n))$ allora esisteranno due costanti $c' > 0$ e $n'_0 \geq 0$ tali che $d(n) \leq c' f(n)$ per ogni $n'_0 \geq 0$, moltiplicando entrambi i membri per una costante $a > 0$ otterremo

$$ad(n) \leq ac' f(n)$$

che è valida poiché moltiplicando entrambi i membri per una costante positiva il verso della disequazione rimane invariato. Abbiamo quindi trovato le costanti c e n_0 per cui $ad(n) = O(f(n))$, infatti basta porre $c = ac'$ ed $n_0 = n'_0$.

Dimostrazione proprietà 4 (transitività)

$d(n) = O(f(n)), f(n) = O(g(n)) \Rightarrow d(n) = O(g(n))$

Per ipotesi abbiamo che:

- $d(n) = O(f(n)) \Rightarrow$ esisteranno due costanti $c' > 0$ e $n'_0 \geq 0$ tali che $d(n) \leq c' f(n)$ per ogni $n'_0 \geq 0$.
- $f(n) = O(g(n)) \Rightarrow$ esisteranno due costanti $c'' > 0$ e $n''_0 \geq 0$ tali che $f(n) \leq c'' g(n)$ per ogni $n''_0 \geq 0$.

Considerando che $c' f(n) \leq c' c'' g(n)$, possiamo unire le due disuguaglianze come segue

$$d(n) \leq c' f(n) \leq c' c'' g(n) \qquad \forall n \geq n'_0, n''_0$$

Ponendo $c = c' c''$ ed $n_0 = max\{n'_0, n''_0\}$, possiamo affermare che $d(n) \leq cg(n)$ per ogni $n \geq n_0$ e ciò implica che $d(n) = O(g(n))$.

Dimostrazione proprietà 2 (additività)

$d(n) = O(f(n)), e(n) = O(g(n)) \Rightarrow d(n) + e(n) = O(f(n) + g(n))$

Per ipotesi abbiamo che:

- $d(n) = O(f(n)) \Rightarrow$ esisteranno due costanti $c' > 0$ e $n'_0 \geq 0$ tali che $d(n) \leq c'f(n)$ per ogni $n'_0 \geq 0$.
- $e(n) = O(g(n)) \Rightarrow$ esisteranno due costanti $c'' > 0$ e $n''_0 \geq 0$ tali che $e(n) \leq c''g(n)$ per ogni $n''_0 \geq 0$.

Considerando che $d(n)+e(n) \leq c'f(n)+c''g(n) \leq max\{c', c''\}f(n)+max\{c', c''\}g(n)$, è una disequazione valida perché prendendo il massimo tra le due costanti e moltiplicandolo alle due funzioni ottengo una quantità maggiore o uguale nel caso $c' = c''$. Alla luce di questo

$$max\{c', c''\}f(n) + max\{c', c''\}g(n) = max\{c', c''\}(f(n) + g(n)) \quad \forall n \geq n'_0, n''_0$$

Ponendo $c = max\{c', c''\}$ ed $n_0 = max\{n'_0, n''_0\}$, possiamo affermare che $d(n)+e(n) \leq c(f(n)+g(n))$ per ogni $n \geq n_0$ e ciò implica che $d(n)+e(n) = O(f(n)+g(n))$.

Espressione O	nome
$O(1)$	costante
$O(loglogn)$	$loglog$
$O(logn)$	logaritmico
$O(\sqrt[c]{n}), c > 1$	sublineare
$O(n)$	lineare
$O(nlogn)$	$nlogn$
$O(n^2)$	quadratico
$O(n^3)$	cubico
$O(n^k), k \geq 1$	polinomiale
$O(a^n), a > 1$	esponenziale

Tabella 3.1: La tabella riporta tutti i tempo di esecuzione di un algoritmo con il rispettivo nome

3.3 Analisi di funzioni logaritmiche

3.3.1 Bound asintotici per funzioni logaritmiche

Per quanto riguarda lo studio della complessità asintotica di funzioni logaritmiche, la base del logaritmo non influisce sull'analisi asintotica, infatti:

> $O(log_a n) = O(log_b n)$, $\Omega(log_a n) = \Omega(log_b n)$, $\Theta(log_a n) = \Theta(log_b n)$ per ogni costante $a > 0$ e $b > 0$.

Dimostrazione per O

Dalla proprietà 1 dei logaritmi si ha che $log_a n = \frac{log_b n}{log_b a}$, siccome banalmente si ha che $log_b n = O(log_b n)$ allora per la regola 1 delle notazioni asintotiche possiamo dire che

$$log_a n = \frac{1}{log_b a} \cdot log_b n = O(log_b n)$$

dove $\frac{1}{log_b a}$ è ovviamente una costante, analogamente si può dimostrare anche il contrario.

Di seguito è enunciata un'altra proprietà nota delle funzioni logaritmiche:

> Ogni funzione logaritmica $logn$ a prescindere dalla base sta sempre al di sotto, in termini di tempo, rispetto ad una funzione lineare. In poche parole $logn = O(n)$.

Dimostrazione per induzione

Dimostriamo per induzione che $log_2 n \leq (1) \cdot n$ per ogni $n \geq 1$.

Base induttiva: Dimostriamo che $log_2 n \leq n$ per $n = 1$, quindi $log_2 1 = 0 \leq 1$. **Vero**.

Passo induttivo: Supponiamo $log_2 n \leq n$ sia vera per un generico $n \geq 1$, dimostriamo che sia vera anche per $n + 1$.

Essendo il logaritmo una funzione crescente possiamo dire che $log_2(n+1) \leq log_2(2n)$, quindi sfruttando le proprietà dei logaritmi possiamo affermare che:

$$log_2(n + 1) \leq log_2(2n) = log_2 2 + log_2 n = 1 + log_2 n$$

Per ipotesi induttiva abbiamo che $log_2 n \leq n$ e quindi

$$1 + log_2 n \leq n + 1 \Rightarrow log_2(n + 1) \leq n + 1$$

Seguendo la catena di disuguaglianze abbiamo dimostrato che $log_2 n + 1 \leq n + 1$, di conseguenza abbiamo dedurre che $logn \leq cn$ per ogni $n \geq n_0$, dove $c = 1$ e $n_0 = 1$, e quindi $logn = O(n)$.

3.4 Tempo logaritmico

Tipicamente si ha un tempo logaritmico quando ogni passo effettuato dal logaritmo riduce di un fattore costante il numero di passi ancora da effettuare.

```
for(i=1;i<=n;i=i*2)
{
   printf("%d", i);
}
```

Il for in alto richiede tempo $\Theta(logn)$.

Dimostrazione

Il for termina quando i diventa maggiore di n, e ad ogni iterazione il valore di i raddoppia, di fatto dopo la k-esima iterazione $i = 2^k$.

Per sapere dopo quante iterazioni il ciclo termina, bisogna sapere ***il più piccolo*** k ***per cui*** $2^k > n$. In altre parole vogliamo un k tale che $2^k > n$ e $2^{k-1} \leq n$. Risolvendo le due disequazioni otteniamo:

$$2^k > n \iff k > log_2 n$$
$$2^{k-1} \leq n \iff k - 1 \leq log_2 n$$

Le due disuguaglianze ottenute implicano

$$log_2 n - 1 < k - 1 \leq log_2 n$$

Quindi per la definizione di ***parte intera inferiore*** possiamo riscrivere la precedente disequazione in $k - 1 = \lfloor log_2 n \rfloor \Rightarrow k = \lfloor log_2 n \rfloor + 1$. Di conseguenza deduciamo che il numero di iterazioni totali del ciclo è proprio $\lfloor log_2 n \rfloor + 1 = \Theta(n)$.

Ovviamente se invece di moltiplicare per due avessimo moltiplicato per una costante $c > 1$, essa sarebbe stata la base del logaritmo, ma come abbiamo dimostrato in precedenza la base del logaritmo è superflua ai fini della valutazione asintotica.

Inoltre, se invece di moltiplicare avessimo diviso per due il procedimento sarebbe stato analogo.

3.4.1 Analisi ricerca binaria

Dato un array a ordinato di n numeri ed un numero x, vogliamo determinare se x è in a.

```
binarySearch(a, n, x)
{
    l = 0;
    r = n;
    while(l<=r)
    {
        c = (l+r)/2;
        if(x = a[c])
            return true;
        if(x<a[c])
            r = c-1;
        else
            l = c+1;
    }
    return false;
}
```

Il while termina quando $l > r$, cioè quando il range $[l, r]$ è vuoto.

Inizialmente $[l, r] = [0, n-1]$ e quindi contiene n elementi, dopo la prima iterazione $[l, r]$ contiene al più $\lfloor \frac{n}{2} \rfloor$ elementi, dopo la seconda iterazione $[l, r]$ contiene al più $\lfloor \frac{n}{4} \rfloor$ elementi e così via. Possiamo dedurre quindi che alla k-esima iterazione $[l, r]$ contiene al più $\lfloor \frac{n}{2^k} \rfloor$ elementi. Per conoscere dopo quante iterazioni termina il ciclo, dobbiamo trovare il più piccolo intero per cui $\lfloor \frac{n}{2^k} \rfloor < 1$ e $\lfloor \frac{n}{2^{k-1}} \rfloor \geq 1$. Risolvendo le disequazioni otteniamo:

$$\left\lfloor \frac{n}{2^k} \right\rfloor < 1 \iff \frac{n}{2^k} < 1 \iff 2^k > n \iff k > log_2 n$$
$$\left\lfloor \frac{n}{2^{k-1}} \right\rfloor \geq 1 \iff \frac{n}{2^{k-1}} \geq 1 \iff 2^{k-1} \leq n \iff k - 1 \leq log_2 n$$

Le due disuguaglianze ottenute implicano

$$log_2 n - 1 < k - 1 \leq log_2 n$$

Quindi per la definizione di ***parte intera inferiore*** possiamo riscrivere la precedente disequazione in $k - 1 = \lfloor log_2 n \rfloor \Rightarrow k = \lfloor log_2 n \rfloor + 1$. Di conseguenza deduciamo che il numero di iterazioni totali del ciclo è proprio $\lfloor log_2 n \rfloor + 1 = \Theta(n)$.

3.5 Analisi di funzioni sublineari

```
j = 0;
i = 0;
while(i<=n)
{
    j++;
    i=i+j;
}
```

Il while termina quando i diventa maggiore di n. Alla k-esima iterazione al valore di i viene sommato $j = k$, per cui il valore di i è $\sum_{i=1}^{k} i = \frac{k(k+1)}{2}$, quindi affinché il while si interrompa è sufficiente che $\frac{k(k+1)}{2} > n$. Per semplicità osserviamo che $\frac{k^2}{2} \leq \frac{k(k+1)}{2}$ per cui se $\frac{k^2}{2} > n$ allora anche $\frac{k(k+1)}{2} > n$. Proseguiamo in questo modo:

$$\frac{k^2}{2} > n \iff k^2 > 2n \iff k > (2n)^{\frac{1}{2}}$$

Quindi k è il più piccolo intero maggiore di $(2n)^{\frac{1}{2}}$ per cui per la definizione di parte intera inferiore abbiamo che dopo $\lfloor (2n)^{\frac{1}{2}} \rfloor + 1 = O(\sqrt{n})$ iterazioni il while termina.

3.5.1 Logaritmi a confronto con polinomi e radici

Per ogni ***costante*** $x > 0$, $logn = O(n^x)$ (x può essere anche minore di 1).

Dimostrazione

Se $x \geq 1$ si ha $n \leq n^x$ per ogni $n \geq n_0$ e quindi $n = O(n^x)$. In precedenza abbiamo dimostrato che $logn = O(n)$, quindi sfruttando la proprietà transitiva abbiamo che $logn = O(n^x)$.

Adesso consideriamo il caso in cui $x < 1$. Vogliamo trovare le costanti $c > 0$ e $n_0 \geq 0$ tale che $logn \leq cn^x$ per ogni $n \geq n_0$.

Sappiamo che $log_2 m < m$ per ogni $m \geq 1$, ponendo $m = n^x$ con $n \geq 1$ si ha

$$log_2 n^x < n^x \Rightarrow x log_2 n < n^x$$

procediamo dividendo entrambi i membri per x, otterremo

$$log_2 n < \frac{1}{x} \cdot n^x$$

Abbiamo trovato le due costanti $c = \frac{1}{x}$ e $n_0 = 1$ e siccome la base del logaritmo può essere trascurata ai fini dell'analisi degli algoritmi abbiamo dimostrato che $logn = O(n^x)$ anche nel caso in cui $x < 1$.

Per ogni $x > 0$ e $b > 0$ ***costanti***, $(logn)^b = O(n^x)$.

Dimostrazione

Vogliamo trovare le costanti $c > 0$ e $n_0 \geq 0$ tali che $(logn)^b \leq cn^x$ per ogni $n \geq n_0$. Risolviamo $(logn)^b \leq cn^x$:

$$(logn)^b \leq cn^x \iff logn \leq (cn^x)^{\frac{1}{b}} = c^{\frac{1}{b}} n^{\frac{x}{b}}$$

Quindi ci basta trovare le costanti $c > 0$ e $n_0 \geq 0$ per cui $logn \leq c^{\frac{1}{b}} n^{\frac{x}{b}}$ per ogni $n \geq n_0$. Precedentemente abbiamo dimostrato che $logn = O(n^y)$ per ogni $y > 0$. Ciò è valido anche se poniamo $y = \frac{x}{b}$, quindi esistono due costanti $c' > 0$ e $n'_0 \geq 0$ tali che $logn = c' n^{\frac{x}{b}}$ per ogni $n \geq n'_0$.

A questo punto basta fissare $c' = c^{\frac{1}{b}}$ da cui $c = (c')^b$, e $n_0 = n'_0$.

Capitolo 4

Grafi

4.1 Grafi non direzionati

Un ***grafo non direzionato*** G è una coppia (V, E) dove V è un insieme finito ed E è una relazione binaria su V. L'insieme V è chiamato ***l'insieme dei vertici*** di G ed i suoi elementi sono chiamati ***vertici***. L'insieme E è chiamato ***l'insieme degli archi*** di G ed i suoi elementi sono chiamati ***archi***. Inoltre si indicano con $n = |V|$ e $m = |E|$, rispettivamente il numero di vertici e il numero di archi presenti nel grafo.

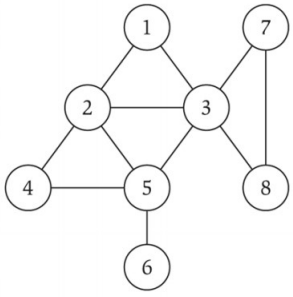


Figura 4.1: un grafo non direzionato.

Esempio di applicazione

Il grafo rappresentato nella figura sottostante rappresenta dei percorsi stradali, in particolare i nodi rappresentano gli incroci (ossia il punto in cui si intersecano più strade) e gli archi le strade, che in questo caso sono a doppio senso di circolazione dato che il grafo

non è direzionato. Inoltre notiamo i numeri presenti vicino agli archi, essi rappresentano il peso dell'arco, in questo caso rappresentano la lunghezza delle strade.

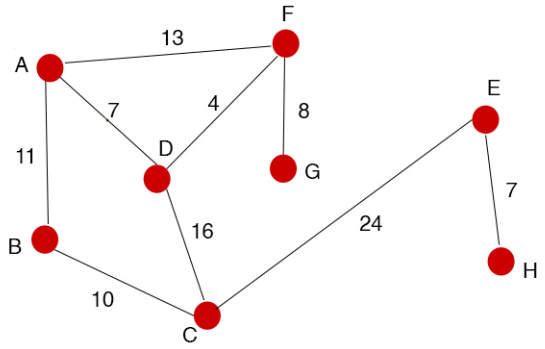


Figura 4.2: esempio di applicazione di un grafo non direzionato.

4.2 Grafo direzionato

Nei grafi direzionati, a differenza di quelli visti precedentemente gli archi hanno una direzione, quindi sono orientati. Si dice che l'arco $e = (u, v)$ lascia u (origine) ed entra in v (destinazione). Per esempio nella figura sottostante esiste l'arco $(1, 6)$ ma non l'arco $(6, 1)$.

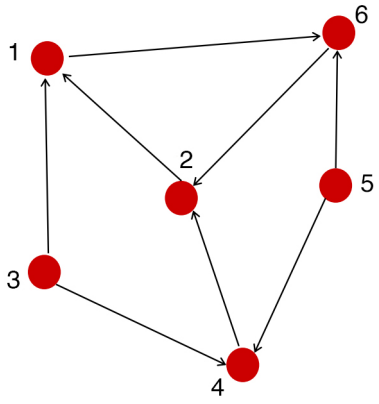


Figura 4.3: un grafo direzionato.

I grafi non direzionati possono essere considerati un caso particolare dei grafi direzionati in cui per ogni arco (u, v) c'è l'arco di direzione opposta (v, u).

Come nell'esempio precedente, la figura sottostante rappresenta dei percorsi stradali, ma a differenza di prima, le strade possono essere percorse soltanto nella direzione dell'arco.

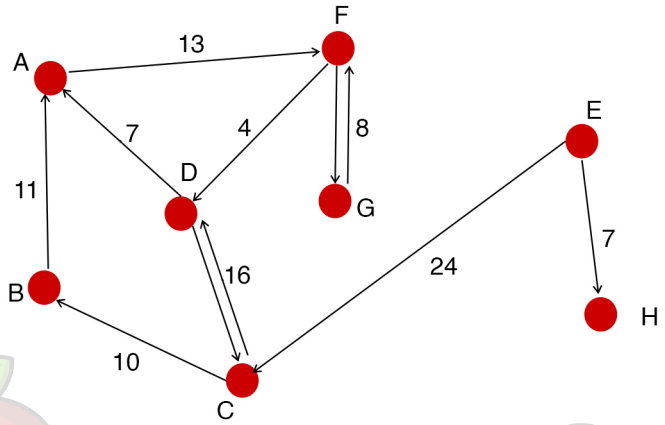


Figura 4.4: esempio di applicazione di un grafo direzionato.

4.3 Applicazioni dei grafi

I grafi hanno tantissime applicazioni nel mondo reale, per esempio:

- Rete di amicizia di un social network: gli utenti rappresentano i nodi, se due utenti diventano amici allora si crea un arco tra i due nodi;
- Google Maps: i nodi rappresentano le città, le intersezioni le strade e gli archi rappresentano la connessione diretta tra i nodi.
  - La rappresentazione tramite grafo permette di trovare il percorso più corto per andare da un posto ad un altro mediante un particolare algoritmo che analizzeremo in questi capitolo.
- World Wide Web: le pagine web sono i nodi e il link tra due pagine è un arco. Google utilizza questa rappresentazione per esplorare il World Wide Web.

Graph	Nodi	Archi
Trasporto	Intersezioni di strade	Strade
Trasporto	Aereoporti	Voli diretti
Comunicazione	Computer	Cavi fibra ottica
World Wide Web	Web page	hyperlink
Rete sociale	Persone	Relazioni
Catena alimentare	Specie	Predatore-preda
Scheduling	Task	Vincoli di precedenza
Circuiti	Gate	Wire

Tabella 4.1: Alcune applicazioni dei grafi

4.4 Terminologia

Consideriamo due nodi u e v di un grafo G connessi dall'arco $e = (u, v)$, si dice che:

- u e v sono adiacenti;
- u e v sono le estremità dell'arco e;
- l'arco (u, v) indice sui vertici u e v;
- u è un nodo vicino di v e viceversa;
- il grado di u (il numero di archi incidenti su u è indicato con $deg(u)$.

4.5 Calcolare il numero di archi di un grafo

4.5.1 Numero di archi di un grafo non direzionato

Sia m il numero di archi e n il numero di nodi di un grafo G, possiamo dire che:

> La somma di tutti i gradi dei nodi di G è $2m$: $\sum_{u \in G} deg(u) = 2m$

Dimostrazione Ciascun arco incide su due vertici e quindi viene contato due volte nella sommatoria dei gradi. L'arco (x,y) è contato sia in $deg(x)$ che in $deg(y)$ e per questo la somma di tutti i gradi dei nodi è 2 volte il numero dei archi.

Il numero m di archi di un grafo G non direzionato è al più $\frac{n(n-1)}{2}$.

Dimostrazione Il numero di coppie non ordinate distinte che si possono formare con n nodi è $\frac{n(n-1)}{2}$ perché posso scegliere il primo nodo dell'arco in n modi diversi, ma il secondo, dato che deve essere diverso dal primo nodo, può essere scelto in $n-1$ modi. Dimezziamo questa quantità dato che stiamo considerando un grafo non direzionato e quindi l'arco (u, v) è uguale all'arco (v, u).

4.5.2 Numero di archi di un grafo direzionato

Sia m il numero di archi e n il numero di nodi di un grafo G, possiamo dire che:

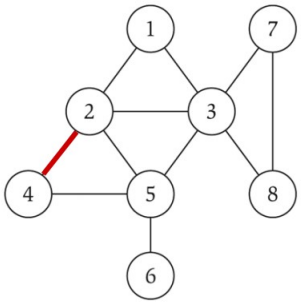
Il numero m di archi di G è al più n^2

Dimostrazione Il numero di coppie ordinate distinte che si possono formare con n nodi è n^2, infatti posso scegliere il primo nodo dell'arco in n modi diversi ma anche il secondo in n modi diversi (se ammettiamo archi con entrambe le estremità uguali).

4.6 Rappresentazioni di grafi

4.6.1 Matrice di adiacenza

Una matrice di adiacenza è una matrice A, $n \times n$ con $A_{uv} = 1$ se (u, v) è un arco. Nella matrice avremo due rappresentazioni di ciascun arco ed uno spazio proporzionale ad n^2. Inoltre per verificare se (u, v) è un arco basta accedere alle posizioni (u, v) della matrice, quindi richiede tempo $\Theta(1)$, mentre identificare tutti i possibili archi richiede ovviamente tempo $\Theta(n^2)$.



	1	2	3	4	5	6	7	8
1	0	1	1	0	0	0	0	0
2	1	0	1	1	1	0	0	0
3	1	1	0	0	1	0	1	1
4	0	1	0	0	1	0	0	0
5	0	1	1	1	0	1	0	0
6	0	0	0	0	1	0	0	0
7	0	0	1	0	0	0	0	1
8	0	0	1	0	0	0	1	0

Figura 4.5: Una matrice di adiacenza

4.6.2 Liste di adiacenza

Le liste di adiacenza sono array di liste in cui ogni lista è associata ad un nodo, in particolare ogni arco corrisponde ad un elemento della lista. Se l'arco (u, v) esiste allora la lista associata ad u contiene v e viceversa nel caso in cui si ha un grafo non direzionato. Lo spazio delle liste di adiacenza è proporzionale a $m + n$, infatti individuare tutti gli archi del grafo richiede tempo $O(deg(u))$. Inoltre controllare se (u, v) è un arco richiede tempo $O(deg(u))$.

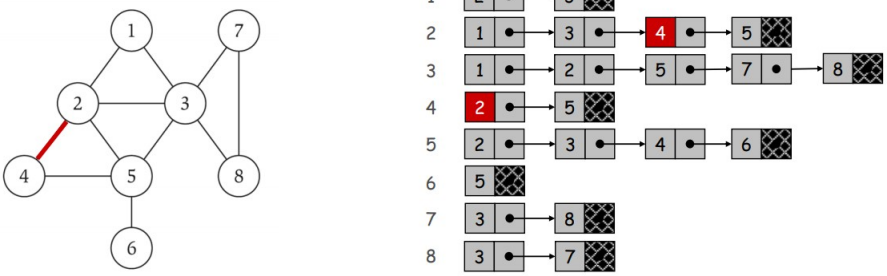


Figura 4.6: Una lista di adiacenza

4.7 Caratteristiche di un grafo

4.7.1 Percorsi e connettività

- Un percorso in un grafo non direzionato $G = (V, E)$ è una sequenza P di nodi $v_1, v_2, \ldots, v_{k-1}, v_k$ con la proprietà che ciascuna coppia di vertici consecutivi v_i, v_{i+1} è unita da un arco E.
- Un percorso è ***semplice*** se tutti i nodi sono distinti.
- un grafo non direzionato è ***connesso*** se per ogni coppia di nodi u e v, esiste un percorso tra u e v.

In alcuni casi può essere interessante scoprire il percorso più corto, cioè composto dal minimo numero di archi tra due nodi, ad esempio una rete di trasporti dove i nodi rappresentano gli aeroporti e gli archi i collegamenti diretti tra aeroporti.

4.7.2 Cicli

Un ciclo non è altro che un percorso che inizia e finisce nello stesso nodo. Formalmente diremo che:

- Un ciclo è un percorso $v_1, v_2, \ldots, v_{k-1}, v_k$ in cui $v_1 = v_k$, con $k > 2$.
- Un ciclo $v_1, v_2, \ldots, v_{k-1}, v_k$ è ***semplice*** se i primi $k - 1$ nodi del ciclo sono tutti distinti tra loro.

Esempio Nel grafo sottostante il percorso $1, 2, 4, 5, 3, 1$ rappresenta un ciclo semplice, mentre il percorso $1, 3, 7, 8, 3, 5, 2, 1$ rappresenta un ciclo non semplice.

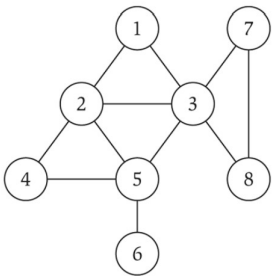


Figura 4.7: Un grafo ciclico

4.7.3 Alberi

Un grafo non direzionato è un ***albero (tree)*** se è connesso e non contiene cicli. In particolare:

Sia G un grafo direzionato con n nodi.

1. G è connesso.
2. G non contiene cicli.
3. G ha $n-1$ archi.

Ogni due delle seguenti affermazioni implica la restante informazione:

- 1 e $2 \implies 3$;
- 1 e $3 \implies 2$;
- 2 e $3 \implies 1$;

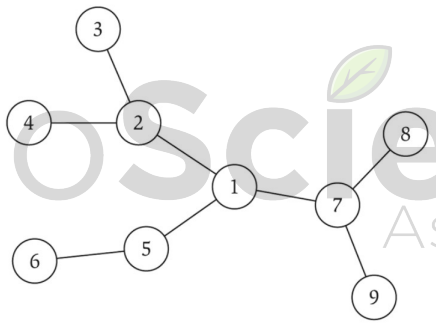


Figura 4.8: Un albero

Alberi con radice

Dato un albero T è possibile scegliere un nodo radice r in modo da considerare gli archi di T come orientati a partire da r.

Secondo questa struttura, un nodo v generico di T si dice:

- **Genitore:** il nodo w che precede v lungo il percorso da r a v (v viene detto figlio di w).
- **Antenato:** un qualsiasi nodo w che precede v lungo il percorso da r a v (v viene detto discendente di w).

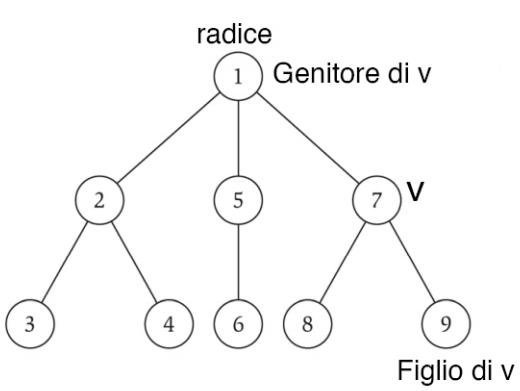


Figura 4.9: Lo stesso albero della figura 4.8, ma radicato in 1.

- **Foglia:** un qualsiasi nodo senza discendenti.

Gli alberi sono molto importanti in tanti ambiti, poiché permettono di rappresentare in maniera semplice ed efficace una struttura gerarchica. Quotidianamente usiamo questo tipo di struttura senza notarlo, ad esempio il ***File System*** del nostro computer.

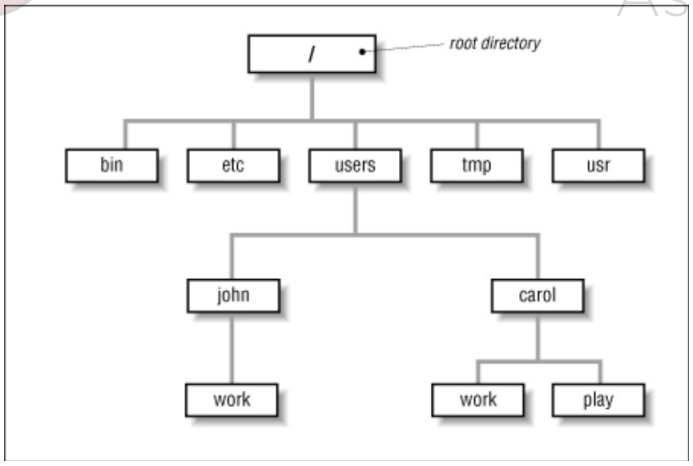


Figura 4.10: Il File System di un sistema operativo.

4.8 Visite di grafi

4.8.1 Connettività

Iniziamo con l'analizzare alcuni algoritmi applicabili ai grafi, in particolare l'algoritmo che stiamo per visionare permette di:

- **Risolvere il problema della connettività:** dati due nodi se t, esiste un percorso tra s e t?
- **Risolvere il problema del percorso più breve: dati due nodi se t, qual è la lunghezza del percorso più breve tra s e t?**
  - **Esempi di utilizzo di tale algoritmo:**
  - Attraversamento di un labirinto;
  - Minimo numero di scali in un viaggio aereo;
  - Minimo numero di dispositivi che devono essere attraversati dai dati di in una rete di comunicazione per andare dalla sorgente alla destinazione;
  - ecc...

4.8.2 Breadth First Search (visita in ampiezza)

L'idea di base di questo algoritmo consiste nell'esplorare il grafo a partire da un certo nodo detto ***sorgente***, e da questo raggiungerà tutti i nodi per cui esiste un percorso che li congiunge.

BFS Esplora il grafo a partire da una sorgente s muovendosi in tutte le possibili direzioni e visitando i nodi livello per livello (layer).

I layer sono descritti come nell'immagine 4.11, ossia in questo modo:

- $L_0 = \{s\}$;
- $L_1 =$ tutti i nodi adiacenti ad s;
- $L_2 =$ tutti i nodi che non appartengono né a L_0 né a L_1, e che sono uniti da una arco ad un nodo situato nel livello L_1;
- $L_i + 1 =$ tutti i nodi che non appartengono agli strati precedenti e che sono uniti da una arco ad un nodo situato nel livello L_i;

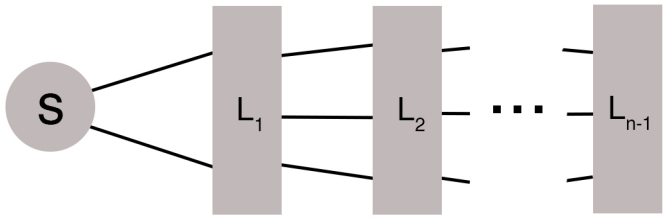


Figura 4.11: Livelli in BFS.

Per ogni i, il layer L_i consiste di tutti i nodi a distanza i dalla sorgente s. Inoltre, esiste un percorso da s a t se e solo se t appare in qualche livello.

Il teorema è facilmente dimostrabile per induzione e sostanzialmente afferma che:

- L_1 è il livello dei nodi a distanza 1 da s;
- L_2 è il livello dei nodi a distanza 2 da s;
  ...
- L_{n-1} è il livello dei nodi a distanza $n-1$ da s;

Pseudocodice BFS (schema dell'algoritmo)

```
1  L_0 = s;
2  for (i = 0; i ≤ n − 2; i++) do
3  |   L_{i+1} = ∅;
4  |   foreach nodo u ∈ L_i do
5  |   |   foreach nodo v adiacente ad u do
6  |   |   |   if v ∉ L_0, ..., L_{i+1} then
7  |   |   |   |   L_{i+1} = L_{i+1} ∪ v;
8  |   |   |   end
9  |   |   end
10 |   end
11 end
```

Esempio di esecuzione BFS

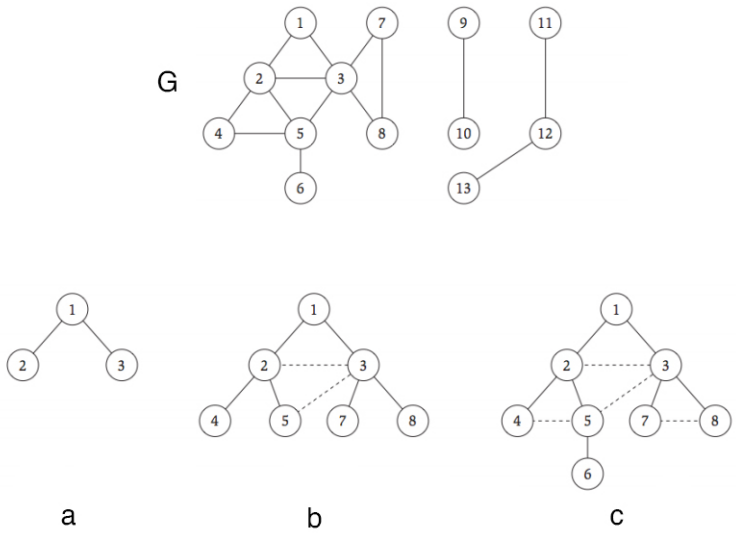


Figura 4.12: Esempio di esecuzione di BFS

Seguendo l'algoritmo appena scritto, deduciamo che i vari passaggi effettuati dall'algoritmo, considerando il nodo 1 come Livello 0 L_0, sono i seguenti:

a) $L_1 = \{2, 3\}$

b) $L_2 = \{4, 5, 7, 8\}$

c) $L_3 = \{6\}$

L'algoritmo riportato però ha un problema per quanto concerne lo studio della complessità, ossia non abbiamo un modo per capire se un nodo è stato già visitato in precedenza. Inoltre il tempo di esecuzione è fortemente dipendente da come è implementato il grafo e da come sono rappresentati gli insiemi L_i. Capiamo che è necessario aggiungere delle informazioni aggiuntive utili al fine di analizzare questo algoritmo.

L'algoritmo BFS dovrebbe produrre un ***albero*** che ha come radice s e come nodi, tutti i nodi del grafo raggiungibili da s.

Come ottenere l'albero Un nodo v viene visitato per la prima volta quando durante l'esame dei nodi adiacenti ad un certo vertice u di un certo livello L_i (linea 5). In questo momento oltre ad aggiungere v nel livello L_{i+1} (linea 7), aggiungiamo l'arco (u, v) e il nodo v all'albero.

Proprietà albero BFS

Si consideri un'esecuzione di BFS su $G = (V, E)$, e sia (x, y) un arco di G. I livelli di x e y differiscono al più di 1.

Dimostrazione Sia L_i il livello di x e L_j il livello di y. Supponiamo che x venga scoperto prima di y, cioè che $x \leq j$. Consideriamo il momento in cui l'algoritmo esamina i nodi adiacenti a x:

- **Caso 1:** il nodo y è già stato scoperto.

  Dato che per ipotesi il nodo y viene scoperto dopo x, avremo che il nodo y viene inserito:

  - nel livello i, dopo x, se y è adiacente a qualche nodo appartenente al livello $i - 1$.
  - nel livello $i + 1$ se è adiacente a qualche nodo nel livello i.

  Quindi si ha $j = i$ o $j = i + 1$.

- **Caso 2:** il nodo y non è stato scoperto.

  Siccome tra i nodi adiacenti di x c'è anche il nodo y, questo viene inserito nel livello L_{i+1}. Quindi in questo caso si ha $j = i + 1$.

Implementazione BFS Tree con liste di adiacenza

- Ciascun insieme L_i è rappresentato da una lista $L[i]$.
- Usiamo un array di booleani *Discovered* per indicare se un nodo è stato già visitato o meno.
- Durante l'esecuzione costruiamo anche l'albero BFS.

```
1  Discovered[s] = true e Discovered[v] = false per tutti gli altri v;
2  Inizializza L[0] in modo che contenga solo s;
3  Poni i = 0;
4  Inizializza il BFS tree T con un albero vuoto;
5  while i ≤ (n − 2) do
6      Inizializza L[i + 1] con una lista vuota;
7      foreach nodo u ∈ L[i] do
8          foreach nodo v adiacente ad u do
9              if Discovered[v] = false then
10                 Poni Discovered[v] = true;
11                 Aggiungi v alla lista L[i + 1];
12                 Aggiungi l'arco (u, v) all'albero T;
13             end
14         end
15     end
16     Poni i = i + 1;
17 end
```

Analizziamo l'algoritmo riga per riga:

- **Riga 1**: n volte.
- **Riga 2-4**: $O(1)$.
- **Riga 5**: n volte.
- **Riga 6**: $n - 1$ volte.
- **Riga 7**: sul totale di tutte le iterazioni del while al più n volte.
- **Riga 8-13**: sul totale di tutte le iterazioni del while al più $2m$ volte.
- **Riga 16**: $O(1)$.

Intuiamo che questo algoritmo richiede tempo $O(n + m)$ nel caso pessimo.

Implementazione BFS Tree con coda FIFO

L'algoritmo BFS si presta ad essere implementato con un coda, ogni volta che viene scoperto un nodo u, questo viene inserito nella coda. Successivamente vengono esaminati gli archi incidenti sul nodo presente in testa alla coda.

```
1  Inizializza la coda Q con una coda vuota;
2  Inizializza il BFS tree T con un albero vuoto;
3  Discovered[s] = true e Discovered[v] = false per tutti gli altri v;
4  Effettuiamo una enqueue(s) su Q;
5  while Q ≠ ∅ do
6      Estraiamo la head di Q con dequeue e poniamolo in u;
7      foreach nodo v adiacente ad u do
8          if Discovered[v] = false then
9              Poni Discovered[v] = true;
10             Aggiungi v in coda a Q con una enqueue;
11             Aggiungi l'arco (u, v) all'albero T;
12         end
13     end
14 end
```

Analizziamo l'algoritmo riga per riga:

- **Riga 1**: $O(1)$.
- **Riga 2**: $O(1)$.
- **Riga 3**: n volte.
- **Riga 4**: $O(1)$.
- **Riga 5**: nel caso pessimo verrà eseguito n volte
- **Riga 6**: $O(1)$.
- **Riga 7-13**: Nel caso pessimo per tutte le iterazioni del while verrà eseguito $2m$ volte.

Analogamente al caso precedente abbiamo una situazione in cui il tempo di esecuzion è proprio uguale ad $O(n + m)$.

4.8.3 Depth First Search (visita in profondità)

Un altro metodo naturale per trovare tutti i nodi raggiungibili da un certo nodo s in un grafo è la ***visita in profondità***, che simula il comportamento di una persona che esplora un labirinto di camere interconnesse. La persona parte dalla prima stanza (nodo s) e da lì si sposta in una delle camere accessibili effettuando lo stesso procedimento nella seconda camera e così via fin quando raggiunge un "vicolo cieco". A questo punto la persona torna indietro di una stanza e esplora le altre possibili stanze.

Quindi riassumendo, l'algoritmo dovrebbe comportarsi in questo modo:

- La visita parte da s, segue uno degli archi uscenti da s ed esplora il vertice v a cui conduce l'arco.
- Arrivato in v, se c'è un arco uscente da v che porta in un nodo w non ancora esplorato allora si procede esplorando w.

- Una volta in w si segue uno degli archi uscenti e così via fin quando non si raggiunge un nodo a partire dal quale non si può raggiungere alcun nodo non esplorato già in precedenza.

- Arrivati ad una situazione del genere l'algoritmo fa ***backtrack*** (torna indietro), finquando non arriva in un nodo a partire dal quale può visitare un altro nodo non ancora esplorato in precedenza.

Da come si può intuire dai passaggi appena descritti, questo algoritmo si presta ad una naturale soluzione ***ricorsiva***.

Pseudocodice DFS

```
1 Marca il nodo u come "Esplorato" e aggiungi u a R (insieme dei nodi esplorati);
2 foreach arco (u, v) incidente su u do
3     if v non è marcato con "Esplorato" then
4         Invoca una chiamata ricorsiva DFS(v);
5     end
6 end
```

Per analizzare un algoritmo ricorsivo occorre stabilire il costo della singola chiamata ricorsiva (ignorando le altre chiamate ricorsive al suo interno): ciascuna visita ricorsiva impiega tempo tempo $O(1 + deg(u))$.

Per sapere in totale il tempo richiesto dall'algoritmo dobbiamo stabilire il numero di chiamate ricorsive effettuate nel caso pessimo dall'algoritmo: in questo caso nel caso peggiore ci saranno tante chiamate ricorsive quante sono i nodi del grafo. Di conseguenza avremo:

$$\sum_{u \in G} O(1 + deg(u)) = O\left(\sum_{u \in G} 1 + \sum_{u \in G} deg(u)\right) = O(n + m)$$

Esempio di esecuzione DFS

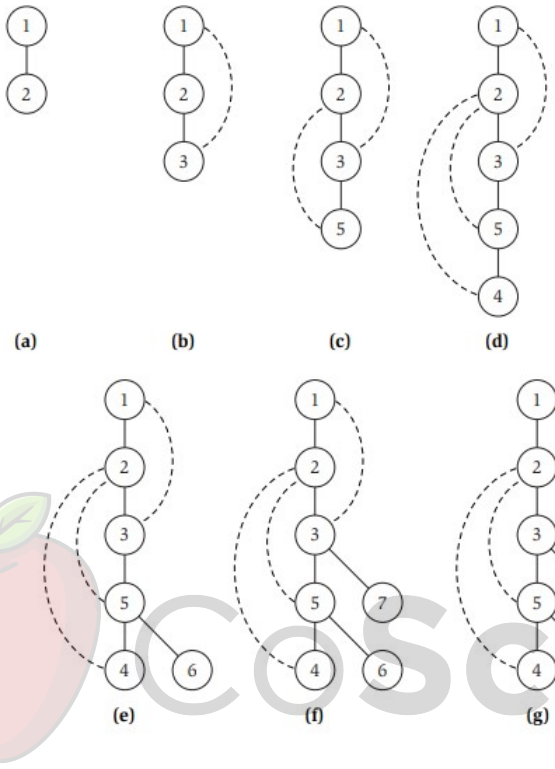


Figura 4.13: Esempio di esecuzione di DFS

Proprietà albero DFS

Proprietà 1

> Per una data chiamata ricorsiva $DFS(u)$, tutti i nodi che vengono etichettati come "Esplorati" tra l'inizio e la fine della chiamata $DFS(u)$ saranno discendenti di u nell'albero DFS.

Dimostrazione proprietà 1 Sia x un nodo esplorato tra l'inizio e la fine della chiamata $DFS(u)$. Per assurdo ***supponiamo che*** x ***non sia discendente di*** u, ovviamente u non potrà essere la radice poiché sarebbe antenato di tutti i nodi del grafo. Sia y il padre di x, quindi y sarà esplorato prima di x e dopo u altrimenti x sarebbe esplorato anch'esso prima di u. Ragioniamo analogamente anche per il nodo z ossia il padre di y

(z è esplorato prima di y e dopo u) e procediamo a ritroso nel grafo. Dato che tutte le chiamate sugli antenati di x avvengono dopo u (tra l'inizio e la fine di $DFS(u)$) compresa quella sulla radice dell'albero DFS, ma ciò è assurdo.

Proprietà 2

> Sia T un albero DFS e siano x e y due nodi di T collegati dall'arco (x, y) in G. Si ha che x e y. Si ha che x e y sono l'uno l'antenato dell'altro nell'albero T.

Dimostrazione proprietà 2

- **Caso (x, y) è in T:**

  In questo caso proprietà è banalmente vera.

- **Caso (x, y) non è in T:**

  Supponiamo che $DFS(x)$ venga invocata prima di $DFS(y)$, ciò vuol dire che alla chiamata $DFS(x)$ il nodo y è ancora etichettato come "Non Esplorato". La chiamata $DFS(x)$ esamina l'arco (x, y) ma per ipotesi non lo inserisce nell'albero T. Questo avviene solo se y è stato già etichettato come "Esplorato". Siccome all'inizio della chiamata $DFS(x)$ il nodo y era etichettato come "Non Esplorato", vuol dire che è stato esplorato tra l'inizio e la fine della chiamata di $DFS(x)$. Di conseguenza per la *proprietà 1* possiamo asserire che y è discendente di x.

Implementazione di DFS tramite stack

```
1  Poni Explored[s] = true ed Explored[v] = false per tutti i nodi v del grafo;
2  Inizializza lo stack S con s;
3  while S non è vuoto do
4  |  Metti in u il nodo al top di S;
5  |  if c'è un arco (u, v) incidente su u then
6  |  |  if Explored[v] = false then
7  |  |  |  Poni Explored[v] = true;
8  |  |  |  Inserisci v al top di S;
9  |  |  end
10 |  end
11 |  else
12 |  |  Effettua un pop su S;
13 |  end
14 end
```

Per implementare la *linea 6* in modo efficiente possiamo supporre di mantenere per ogni vertice u un puntatore corrispondente al prossimo nodo adiacente a u.

Il while richiede esattamente tempo $O(m)$ poiché lo stack si svuoterà dopo aver esaminato tutti i nodi adiacenti v per ogni nodo v esplorato. Considerato che l'istruzione alla *riga 1* impiega tempo $O(n)$ e tutte le altre istruzioni tempo costante, possiamo dire che questo algoritmo impiega tempo $O(n + m)$.

N.B Sia gli algoritmi per BFS che per DFS visionati valgono solo per i grafi non direzionati, per quelli direzionati si devono apportare alcune modifiche.

4.9 Componenti connesse

Una componente connessa di un grafo non direzionato è un ***sottoinsieme di vertici*** tale che per ogni coppia di vertici (u, v) della componente esiste un percorso tra u e v. Inoltre dato un nodo s diremo che esiste una ***componente connessa contenente*** s se esiste una componente formata da tutti i nodi raggiungibili da s.

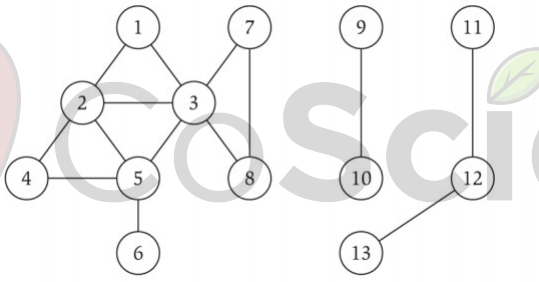


Figura 4.14: Il grafo nella figura è composto da 3 componenti connesse.

Come facciamo a trovare una componente connessa contenente un certo nodo s? Per trovare una componente connessa contente s si devono trovare tutti i nodi raggiungibili da s quindi possiamo usare l'algoritmo BFS o DFS utilizzando s come sorgente.

Effettuando le opportune modifiche è possibile usare BFS o DFS per trovare tutte le componenti connesse.

4.9.1 Insieme di tutte le componenti connesse

Per ogni due nodi s e t di un grafo, le loro componenti connesse o sono uguali o sono disgiunte.

Dimostrazione

- **Caso 1:** Esiste un percorso tra s e t.

  In questo caso ogni nodo u raggiungibile da s è anche raggiungibile da t e viceversa . Di conseguenza possiamo dire che le componenti connesse di s e t sono ***uguali*** poiché u è nella componente connessa di s se e solo se è anche in quella di t.

- **Caso 2:** Non esiste un percorso tra s e t.

  In questo caso non può esistere un nodo u raggiungibile sia da s che da t, di conseguenza i nodi s e t appartengono a due componenti connesse diverse.

Algoritmo per trovare tutte le componenti connesse tramite BFS

```
1 Per ogni nodo u di G setta Discovered[u] = false;
2 foreach nodo u di G do
3 |   if Discovered[u] = false then
4 |   |   BFS(u);
5 |   end
6 end
```

Affinché questo algoritmo funzioni si deve modificare l'algoritmo BFS in modo tale che nella fase di inizializzazione non vada a modificare i valori nell'array $Discovered$.
Analisi:

Indichiamo con k il numero di componenti connesse e con n_i e m_i rispettivamente il numero di nodi e di archi della componente i-esima. Il **foreach** verrà eseguito n volte, ma si entra all'interno dell'if solo k volte, poiché una volta eseguito BFS su un nodo, l'algoritmo setterà a $Discovered[v] = true$ per ogni nodo v raggiungibile da u (nella stessa componente connessa). Di conseguenza possiamo affermare che il tempo richiesto da tutte le visite BFS è

$$\sum_{i=1}^{k} O(n_i + m_i) = O\left(\sum_{i=1}^{k}(n_i + m_i)\right) = n + m$$

Poiché le componenti connesse diverse sono disgiunte tra di loro, ogni nodo sarà contato una sola volta, quindi la sommatoria è esattamente uguale a $n + m$. Quindi il tempo dell'algoritmo è proprio uguale a $O(n + m)$.

Alcune considerazioni

È possibile modificare l'algoritmo appena scritto in modo da poter marcare per ogni nodo la componente connessa a cui appartengono. A questo scopo si potrebbe usare un array che indica il numero della componente connessa a cui appartiene un singolo nodo (ad esempio $Component[u] = j$ se u appartiene alla componente j-esima).

A tal proposito per modificare l'algoritmo abbiamo bisogno di:

- Inizializzare un array $Component[]$ della taglia coincidente al numero di nodi del grafo;

- Modificare BFS in modo che prenda come parametro un valore che possa essere usato per marcare i nodi della stessa componente connessa ($Component[v] = k$ per ogni nodo v raggiunto dal BFS)

AllComponents(G):

```
1 Per ogni nodo u di G setta Discovered[u] = false;
2 Inizializza un contatore della componente connessa k = 1;
3 foreach nodo u di G do
4     if Discovered[u] = false then
5         BFS(u, k);
6         Incrementa k di 1;
7     end
8 end
```


BFS(u, k):

```
1  Inizializza L[0] in modo che contenga solo u;
2  Component[u] = k;
3  Poni i = 0;
4  Inizializza il BFS tree T con un albero vuoto;
5  while i ≤ (n − 2) do
6      Inizializza L[i + 1] con una lista vuota;
7      foreach nodo x ∈ L[i] do
8          foreach nodo v adiacente ad x do
9              if Discovered[v] = false then
10                 Poni Discovered[v] = true;
11                 Poni Component[v] = k;
12                 Aggiungi v alla lista L[i + 1];
13                 Aggiungi l'arco (x, v) all'albero T;
14             end
15         end
16     end
17     Poni i = i + 1;
18 end
```

4.10 Grafi bipartiti

Un grafo non direzionato è ***bipartito*** se è possibile partizionare l'insieme dei nodi in due sottoinsiemi ***disgiunti*** X e Y tali che ciascun arco del grafo abbia un'estremità in X e l'altra in Y. Per semplificare il concetto possiamo immaginare di colorare i nodi del grafo di due colori, come rosso e blu, in modo tale che ogni arco abbia un'estremità rossa e l'altra blu.

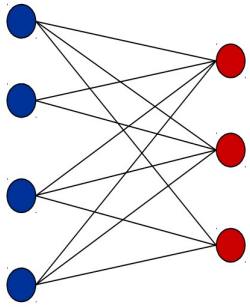
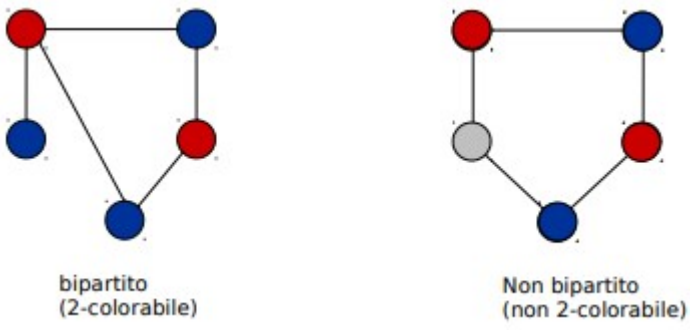


Figura 4.15: Un grafo bipartito

Lemma Se un grafo G è bipartito, non può contenere un ciclo dispari, in parole povere: G bipartito $\Rightarrow$ nessun ciclo dispari in G.



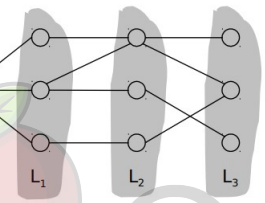
4.10.1 Determinare se un grafo è bipartito

Facciamo un piccolo richiamo sulla proprietà degli alberi BFS:

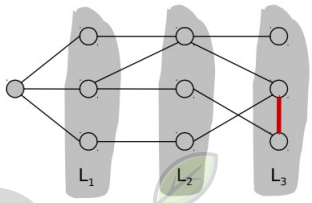
Proprietà BFS Si consideri un'esecuzione di BFS su $G = (V, E)$, e sia (x, y) un arco di G. I livelli di x e y differiscono al più di 1.

Facciamo una piccola osservazione: sia G un grafo connesso e siano $L_0, \ldots, L_k$ i livelli prodotti da un'esecuzione di BFS a partire da un nodo s. Possono verificarsi due casi:

- Nessun arco di G collega due nodi sullo stesso livello.
- Un arco di G collega due nodi sullo stesso livello.



Caso (i)

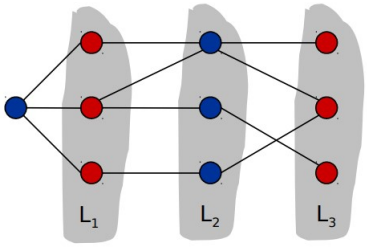


Caso (ii)

- Nel primo caso il grafo ***è bipartito***:

  Per la proprietà sulla distanza tra i livelli rammendata prima, si ha che se due nodi sono adiacenti allora o si trovano nello stesso livello o in livelli consecutivi.

  Poiché nel caso 1 non esistono nodi adiacenti appartenenti allo stesso livello, vuol dire che tutti nodi adiacenti fra loro si trovano in livelli consecutivi. Se provassi a marcare o colorare tutti i nodi appartenenti ad un livello dispari di rosso, e quelli appartenenti ad un livello pari di blu avrei una situazione in cui le estremità di ogni arco sono di colore diverso $\Rightarrow$ il grafo è bipartito.

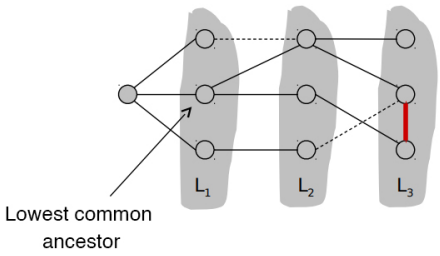


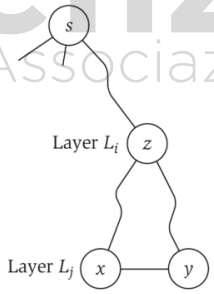
- Nel secondo caso il grafo ***non è bipartito***:

  Supponiamo che esista l'arco (x, y) tra due nodi x e y appartenenti ad uno stesso livello L_j. Indichiamo con z l'antenato comune più vicino a x e y, e indichiamo con L_i il suo livello. È possibile ottenere un ciclo dispari considerando il percorso formato dagli archi del BFS

  - da z a x ($j - i$ archi);
  - da z a y ($j - i$ archi);
  - e da x a y (un arco).

  In totale il ciclo conterrà $2(j - i) + 1$ archi, quindi avremo un ciclo dispari $\Rightarrow$ il grafo non è bipartito.





Algoritmo per determinare se un grafo è bipartito

Questo algoritmo ha userà una BFS modificata affinché possa determinare se il grafo sia bipartito. Utilizzeremo per lo scopo un array $Color[]$ che indica il colore del nodo:

BFS modificata:

```
1  Discovered[s] = true e Discovered[v] = false per tutti gli altri v;
2  Inizializza L[0] in modo che contenga solo s;
3  Poni i = 0;
4  Inizializza il BFS tree T con un albero vuoto;
5  Poni Color[s] = blue;
6  while i ≤ (n − 2) do
7      Inizializza L[i + 1] con una lista vuota;
8      foreach nodo u ∈ L[i] do
9          foreach nodo v adiacente ad u do
10             if Discovered[v] = false then
11                 Poni Discovered[v] = true;
12                 Aggiungi v alla lista L[i + 1];
13                 if i + 1 è dispari then
14                     Color[v] = red;
15                 end
16                 else
17                     Color[v] = blue;
18                 end
19                 Aggiungi l'arco (u, v) all'albero T;
20             end
21         end
22     end
23     Poni i = i + 1;
24 end
```

isBiparted:

```
1  BFS(s) su un nodo s arbitrario;
2  foreach arco (u, v) di G do
3      if Color[u] = Color[v] then
4          Restituisci false;
5      end
6  end
7  Restituisci true;
```

4.11 Visita di Grafi direzionati

Come abbiamo detto prima, gli algoritmi BFS e DFS scritti sono validi solamente per i grafi non direzionati. Per i grafi direzionati è importante effettuare delle modifiche in modo da considerare tutti i possibili percorsi tra i nodi del grafo.

4.11.1 Connettività forte

Quando abbiamo parlato di grafi non direzionati abbiamo introdotto il concetto di connettività, ci rendiamo conto che tale concetto non può essere applicato anche i grafi direzionati. A tal proposito introduciamo il concetto di ***connettività forte***:

- Dati due nodi u e v essi sono ***mutualmente raggiungibili*** se c'è un percorso (diretto) da u a v e un percorso (diretto) da v a u.
- Un grafo è ***fortemente connesso*** se ogni coppia di nodi + mutualmente raggiungibile.

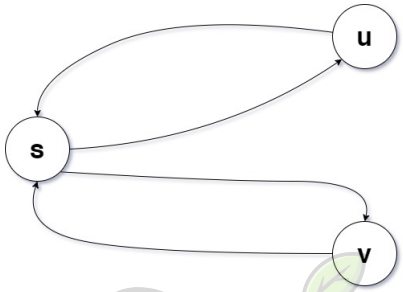


Figura 4.16: un grafo fortemente connesso

Algoritmo per la connettività forte

È possibile determinare se un grafo G è fortemente connesso in tempo $O(n+m)$ tramite BFS, in particolare basta eseguire i seguenti passaggi:

- Prendi un qualsiasi nodo s del grafo;
- Esegui la BFS con sorgente s nel grafo G;
- Crea il grafo G^{rev} inveertendo la direzione di ogni arco del grafo G;
- Esegui la BFS con sorgente s nel grafo G^{rev};
- Restituisci $true$ se e solo se tutti i nodi di G vengono raggiunti da entrambe le esecuzioni di BFS.

Il procedimento è corretto poiché la prima esecuzione trova i percorsi da s a tutti gli altri nodi e la seconda esecuzione trova i percorsi da tutti gli altri nodi ad s (i percorsi sono stati invertiti invertendo il grafo di partenza.

4.12 Grafi direzionati aciclici (DAG)

Un grafo direzionato aciclico (**DAG**) è un grafo direzionato che non contiene cicli direzionati. Sono molto usati per problemi reali poiché possono esprimere vincoli di precedenza o dipendenza: ad esempio un arco (v_i, v_j) indica che v_i deve precedere v_j o che v_j dipende da v_i.

Un esempio potrebbe essere il grafo delle propedeuticità degli esami.

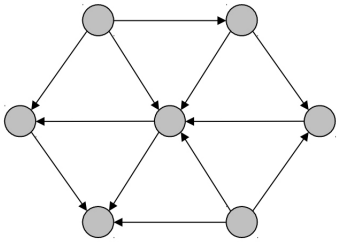


Figura 4.17: Un DAG

4.12.1 Ordine topologico

> Un ***ordinamento topologico*** di un grafo direzionato $G = (V, E)$ è un ordinamento dei nodi che lo compongono tali che per ogni arco v_i, v_j si ha che $i < j$, ossia i precede j nell'ordinamento. Quindi si deduce che il grafo avrà tutti gli archi che punteranno in avanti nell'ordinamento.

Ad esempio nel caso in cui un grafo direzionato G rappresenti le propedeuticità degli esami, un possibile ordinamento topologico potrebbe essere l'ordine in cui gli esami dovrebbero essere sostenuti dallo studente.

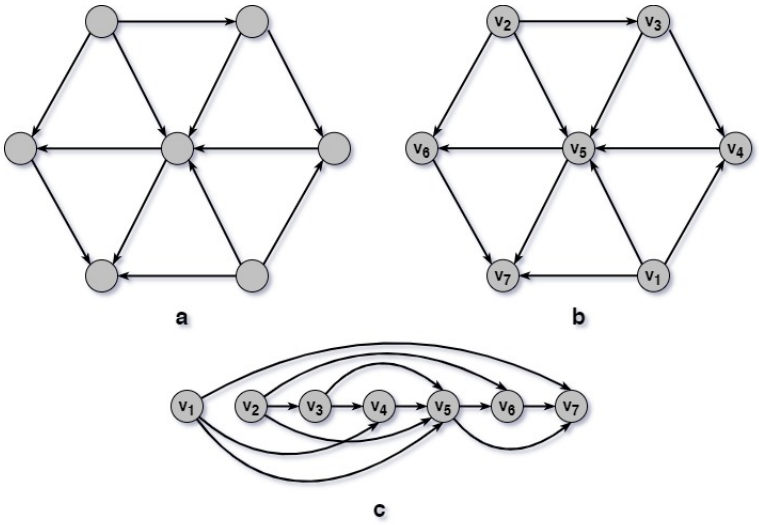
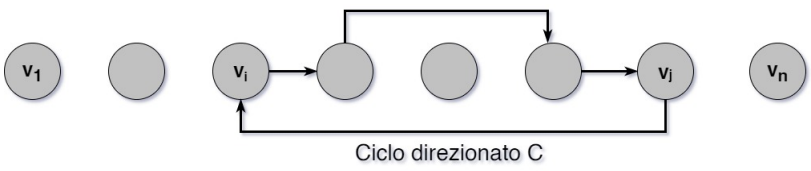


Figura 4.18: Un DAG con ordinamento topologico

Se un grafo direzionato G ha un ordine topologico allora è un ***DAG***.

Dimostrazione (per assurdo) Supponiamo che G sia un grafo direzionato e che abbia un ordinamento topologico $v_1, v_2, \ldots, v_n$. ***Per assurdo*** supponiamo che G non sia un DAG, ossia che abbia un ciclo C. Consideriamo che tra i nodi che appartengono al ciclo C, v_i sia il nodo con indice più piccolo e v_j il vertice che precede v_i all'interno del ciclo (ossia stiamo supponendo che esista l'arco (v_j, v_i)). Siccome il grafo G ha un ordinamento topologico e abbiamo l'arco (v_j, v_i), vorrà dire che per rispettare l'ordinamento $j < i$, ma ciò è assurdo dato che avevamo supposto che $i < j$.



Ci sono altri lemmi importanti dei DAG che adesso dimostreremo:

Se G è un DAG allora esisterà un nodo nel grafo senza nessun arco entrante.

Dimostrazione (per assurdo) Supponiamo che G sia un DAG e che ogni nodo di G abbia ***almeno un arco entrante***. Scegliamo un qualsiasi nodo v e cominciamo a seguire gli archi nel verso opposto alla loro direzione, ciò è possibile poiché abbiamo supposto che ogni nodo ha un arco entrante (v ha un arco entrante (u, v), u ha un arco entrante (x, u) e così via). Immaginiamo di procedere in questo modo per n archi, e quindi per $n+1$ nodi, questo vuol dire che esiste almeno un nodo w in G che viene percorso più di una volta, ossia ***deve esistere un ciclo direzionato*** che incomincia e finisce in w.

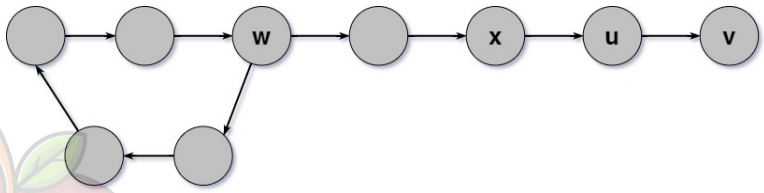


Figura 4.19: Un grafo in cui ogni nodo ha almeno un arco entrante

Se G è un DAG, allora G ha un ordinamento topologico.

Dimostrazione con induzione su n

- **Caso base:** vero banalmente se $n = 1$.
- **Passo induttivo:** Supponiamo che il lemma è vero per un DAG con un numero generico di nodi n, dimostriamo che ciò sia vero anche per $n + 1$.

  Dato un DAG con $n+1$ nodi, prendiamo un nodo senza archi entranti v e consideriamo il grafo $G - \{v\}$: questo sarà ancora un DAG in quanto cancellare un nodo non introduce cicli. Poiché $G-\{v\}$ è un DAG con n nodi allora, per ipotesi induttiva $G - \{v\}$ ha un ordinamento topologico. Riprendiamo il nodo v precedentemente rimosso e poniamolo all'inizio dell'ordinamento in modo che i suoi archi rispettino tale ordinamento: il nodo v non ha archi entranti per ipotesi, quindi il DAG risultante ha ancora un ordinamento topologico, di conseguenza abbiamo dimostrato che il lemma sia verò anche per un DAG con $n + 1$ nodi.

4.12.2 Algoritmi per l'ordinamento topologico

Sfruttando la dimostrazione per induzione scritta poc'anzi è possibile scrivere un algoritmo ricorsivo in grado di trovare l'ordinamento topologico in un DAG:
TopologicalOrder(G)

```
1 if esiste un nodo v senza archi entranti then
2 |   cancella v da G ottenendo G − {v};
3 |   L = TopologicalOrder(G − {v});
4 |   aggiungi v all'inizio di L;
5 |   return L;
6 end
7 else
8 |   return lista vuota;
9 end
```

Analisi:

- Supponiamo ottimisticamente che per ogni nodo esiste una variabile che indica il numero di archi entranti, l'if alla riga 1 implica un for iterato per ogni nodo del grafo: $O(n)$.
- Cancellare un nodo v da G richiede tempo $deg(v)$ poiché richiede tempo proporzionale al numero di archi uscenti da v.
- Ignorando le chiamate ricorsive, tutte le operazioni restanti richiedono tempo costante, quindi il costo di una singola chiamata ricorsiva richiede tempo $O(n + deg(v))$. L'algoritmo ricorsivo viene richiamato n volte quindi il tempo totale sarà $O(n^2)$. [1].

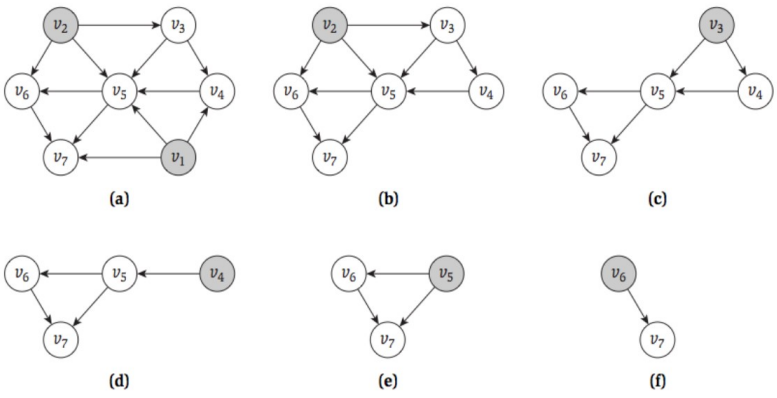


Figura 4.20: Esempio di esecuzione dell'algoritmo appena descritto

[1] $\sum_{u \in G} O(n + deg(u)) = O\left(\sum_{u \in G} n + \sum_{u \in G} deg(u)\right) = O(n^2 + m)$

Algoritmo in tempo lineare

L'algoritmo appena visionato richiede un tempo quadratico, questo tempo non è ottimale considerando soprattutto il caso in cui il grafo è sparso. Per ottenere un bound migliore occorre trovare un modo più efficiente per individuare un nodo senza archi entranti, evitando quindi di effettuare un ciclo lineare per ogni chiamata ricorsiva. Una possibile soluzione potrebbe essere la seguente:

- Un nodo w si dice attivo se non è stato ancora cancellato.
- Occorre memorizzare:
  - $count[w]$, ossia il numero di archi entranti in w provenienti da altri nodi attivi;
  - S, ossia l'insieme dei nodi attivi che non hanno archi entranti provenienti da altri nodi attivi.

Quindi supposto questo possiamo realizzare l'algoritmo in questo modo:

- **Fase di inizializzazione:** tempo $O(n+m)$ in quanto:
  - I valori di $count[w]$ vengono inizializzati e scandendo tutti gli archi e incrementati per ogni arco entrante in w, quindi basta scandire tutti gli archi una sola volta, e questo richiede tempo $O(m)$;
  - Inizialmente tutti i nodi sono attivi, quindi S conterrà tutti i nodi del grafo che non hanno archi entranti. Per scoprire i nodi che non hanno archi entranti basta scandire $count[]$ per ogni nodo del grafo quindi in tempo $O(n)$.
- **Fase di aggiornamento:** consiste nel trovare e cancellare il nodo v senza archi entranti. Per trovarlo basta prendere un qualsiasi nodo in S, per cancellare vR si devono effettuare le seguenti operazioni:
  1. *Cancellare v da S e da G:*
     Per cancellare v dal grafo G occorre tempo $O(deg(v))$ poiché si devono cancellare anche gli archi incidenti su v.
     Se rappresentassimo S con una linked list, e prendessimo ogni volta il nodo in testa alla lista, cancellare v da S richiede tempo costante $O(1)$.
  2. *Decrementare* $count[w]$:
     Si decrementa $count[w]$ per ogni arco (v,w) rimosso, se alla fine della rimozione di tutti gli archi di v, $count[w]=0$ (se il vertice w non avrà più archi entranti) occorre aggiungere w alla lista S, ciò richiede tempo costante $O(1)$ (supposto che inseriamo sempre in testa alla lista). L'operazione di inserimento in S dei nodi w riguarda nel **caso pessimo** tutti i nodi w adiacenti a v, quindi: $O\left(\sum_{i=1}^{deg(v)} 1\right) = O(deg(v))$.

  I passi 1. e 2. vengono eseguiti una volta per ogni nodo di G, quindi per effettuare tutti gli aggiornamenti, avremo complessità:

$$\sum_{u \in G} O(1) + \sum_{u \in G} O(deg(u)) = O(n+m)$$

Inizializzazione

```
1  count[v] = 0 per tutti i nodi v;
2  S = ∅;
3  L = ∅;
4  foreach arco (u, v) do
5  |  count[v] = count[v] + 1;
6  end
7  foreach nodo u in G do
8  |  if count[u] = 0 then
9  |  |  S = S ∪ u;
10 |  end
11 end
```

TopologicalOrder(G)

```
1  if S ≠ ∅ then
2  |  Scegli un nodo v da S;
3  |  Cancella v da S;
4  |  foreach arco (v, w) do
5  |  |  Cancella l'arco (v, w);
6  |  |  count[w] = count[w] − 1;
7  |  |  if count[w] = 0 then
8  |  |  |  S = S ∪ w;
9  |  |  end
10 |  end
11 |  L = TopolgicalOrder(G − {v});
12 |  aggiungi v all'inizio di L;
13 |  return L;
14 end
15 else
16 |  return lista vuota;
17 end
```

Capitolo 5

Algoritmi greedy

Generalmente gli algoritmi per la risoluzione di problemi di ottimizzazione sono costituiti da sequenze di passi elementari, in cui, ad ogni passo si presentano varie scelte alternative. Per alcuni problemi di ottimizzazione, le tecniche di programmazione dinamica per scegliere l'opzione migliore sono inutilmente onerose: è possibile ottenere lo stesso risultato con algoritmi molto più semplici ed efficienti. Una di queste strategie adottate è quella degli ***algoritmi greedy***, ossia una strategia che consiste nel prendere la decisione che, al momento, appare la migliore (***localmente ottima***), nella speranza di ottenere una soluzione ***globalmente ottima***.

5.1 Interval Scheduling

Un problema comune risolvibile in maniera ottimale con la tecnica greedy è quello dell'interval scheduling, che riceve in input un ***insieme di job*** ad ognuno dei quali è associato un intervallo, che ha inizio in un certo istante s_j fine in un istante f_j.

Supposto di aver a disposizione un unico calcolatore, l'obiettivo di questo problema è quello di eseguire più attività possibili.

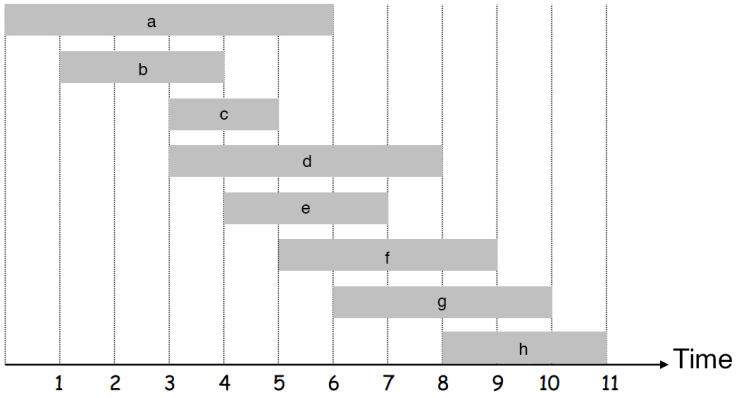
Ovviamente per scegliere l'ordine di esecuzione tra due job generici i e j, questi devono essere ***compatibili*** tra di loro, ossia se $f_i \leq s_j$ o $f_j \leq s_i$. Compreso questo possiamo riformulare l'obiettivo del nostro problema:

> Trovare un sottoinsieme di cardinalità massima di job a due a due compatibili.

Soluzione con algoritmo greedy

Se all'inizio scegliessimo a, poi potremmo scegliere o g o h, per un totale di 2 job. Se invece scegliessimo b e dopo e, poi potremmo scegliere anche h per un totale di 3 job, quindi qual è la strategia greedy da adottare?

Innanzi tutto lo schema greedy in questo caso considera i job in un certo ordine. Ad ogni passo viene esaminato il prossimo job nell'ordinamento e se questo è compatibile



con quelli scelti nei passi precedenti allora il job viene selezionato. Quindi stabilito questo, bisogna capire come ordinare i vari job. Ci sono varie opzioni:

- **Earliest Start Time**

  Considera i job in ordine crescente rispetto ai tempi di inizio s_j (Ad ogni passo si prova a prendere il job che inizia prima tra quelli ancora non esaminati).

  Problemi: Può accadere che il job che inizia per primo finisca dopo molti o, nel caso pessimo, tutti i job: *non è una buona soluzione*.

- **Shortest Interval**

  Considera i job in ordine crescente rispetto alle loro durate $f_i - s_i$ (Ad ogni passo si prova a prendere il job che dura meno tra quelli ancora non esaminati).

  Problemi: Può accadere che un job che dura meno di altri si sovrapponga a due job che non si sovrappongo tra di loro: *non è una buona soluzione*.

- **Fewest conflicts**

  Per ogni job, conta il numero c_j di job che sono in conflitto con lui e ordina in modo crescente rispetto al numero di conflitti (Ad ogni passo si prova a prendere il job che ha meno conflitti tra quelli ancora non esaminati).

Problemi: Può accadere che un job che genera meno conflitti di altri si sovrapponga a due job che non si sovrappongono tra di loro: *non è una buona soluzione*.

Nessuna delle strategie riportate è una strategia efficace: nel nostro caso la scelta vincente è la strategia ***Earliest Finish Time***, ossia quella che considera i job in ordine crescente rispetto ai tempi di fine f_j.

Algoritmo Earliest Finish Time:

- A corrisponde all'insieme di attività scelte;
- f indica il tempo di fine dell'ultimo job selezionato.

```
1  Ordina i job in ordine crescente in modo che f_1 ≤ f_2 ≤ ... ≤ f_n;
2  f = 0;
3  A = ∅;
4  for j che va da 1 a n do
5  |   if s_j ≥ f then
6  |   |   A = A ∪ {j};
7  |   |   f = j;
8  |   end
9  end
10 return A;
```


Analisi tempo di esecuzione:

- Costo ordinamento $O(nlogn)$ se supponiamo di utilizzare il *MergeSort*;
- Costo del for $O(n)$: mantenendo traccia del tempo di fine dell'ultimo job selezionato possiamo confrontarlo con il tempo d'inizio dei successivi job semplicemente verificando che $s_j \geq f$.

Abbiamo visto l'algoritmo greedy *Earliest Finish Time* etichettandolo come ottimo, perché? Dimostriamolo.

Dimostrazione

- Denotiamo con $i_1, i_2, \ldots, i_k$ l'insieme di job selezionati dall'algoritmo greedy in ordine crescente rispetto ai tempi di fine (come selezionati dall'algoritmo).
- Denotiamo con $j_1, j_2, \ldots j_m$ l'insieme di job in una soluzione considerata ottima, ordinati per comodità in ordine crescente rispetto ai tempi di fine.

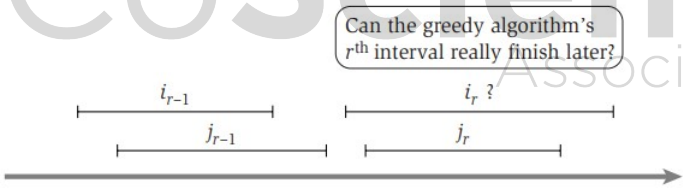
1. Dimostriamo per induzione che l'esecuzione dei job $i_1, i_2, \ldots, i_k$ termina non più tardi di quella dei job $j_1, j_2, \ldots, j_k$, ossia che per ogni indice $1 \leq r \leq k$ si ha che il tempo di fine di i_r è non più grande di quello di j_r.

   - **Passo base** $r = 1$

     Banalmente vero perché la prima scelta greedy seleziona l'attività con tempo di fine minore.

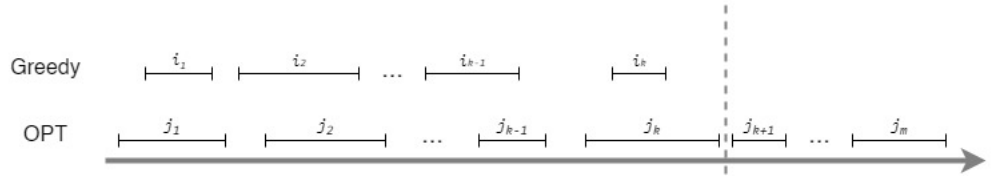
   - **Passo induttivo**

     Supponiamo che per un generico $r - 1$ il tempo di fine di i_{r-1} è non più grande di quello di j_{r-1}, in altri termini supponiamo che $f(i_{r-1}) \leq f(j_{r-1}))$, dimostriamo che ciò sia vero per ogni $r > 1$. Il job j_r sarà sicuramente compatibile con il job j_{r-1}, quindi avremo che $f(j_{r-1}) \leq s(j_r)$, applicando l'ipotesi induttiva possiamo dire che $f(i_{r-1}) \leq s(j_r)$, ossia che il job j_r è compatibile con il job i_{r-1}. In altri termini possiamo dire che al passo $r - 1$, il job j_r si trova nell'insieme di job compatibili con i_{r-1} e non ancora selezionati, insieme a i_r. Siccome l'algoritmo greedy seleziona i_r come prossimo job da eseguire, e per definizione tale algoritmo seleziona il job con il tempo di fine minore tra i vari job compatibili non ancora selezionati, vorrà dire che $f(i_r) \leq s(j_r)$ e che quindi i_r finisce non più tardi di j_r.



2. Usiamo ciò che abbiamo detto nel punto 1. per dimostrare che non è possibile che k sia minore di m: la soluzione greedy corrisponde alla soluzione ottima.

   Supponiamo per assurdo che la soluzione greedy non sia ottima e di conseguenza che $k < m$. Quindi la sequenza $j_1, j_2, \ldots, j_m$ conterrà almeno il job j_{k+1}. Come abbiamo dimostrato poco fa l'esecuzione della sequenza $i_1, i_2, \ldots, i_k$ termina non più tardi della sequenza $j_1, j_2, \ldots, j_k$, e si ha che i job $i_1, i_2, \ldots, i_k$ sono compatibili con $j_k + 1$, ma ciò è assurdo poiché la soluzione greedy dopo aver inserito i job $i_1, i_2, \ldots, i_k$, esamina le altre attività compatibili aventi tempo di fine più piccolo. Tra queste vi è anche j_{k+1} che non viene selezionato dall'algoritmo greedy, per cui è impossibile che l'algoritmo greedy inserisca solo k job, in altre parole è impossibile che $k < m$.



5.2 Partizionamento di intervalli

Un altro caso di studio per quanto concerne gli algoritmi greedy è quello del ***partizionamento di intrevalli***. In questo caso disponiamo di più risorse identiche tra loro e vogliamo che vengano svolte tutte le attività in modo tale da usare il minor numero di risorse e tenendo conto del fatto che due attività non possono usufruire della stessa risorsa allo stesso tempo.

Di conseguenza il nostro obiettivo corrisponde a far svolgere le n attività utilizzando il minor numero possibile di risorse e in modo che ciascuna risorsa utilizzata venga allocata ad al più un'attività alla volta.

Un esempio pratico potrebbe essere la divisione delle lezioni nelle varie aule di un'università. In questo caso:

- Le risorse sono rappresentate dalle aule e le attività dalle lezioni da svolgere.
- La lezione j comincia ad s_j e finisce a f_j.
- L'obiettivo è trovare il minor numero di aule che permetta di far svolgere tutte le lezioni in modo che non ci siano due lezioni che vengono svolte contemporaneamente nella stessa aula.

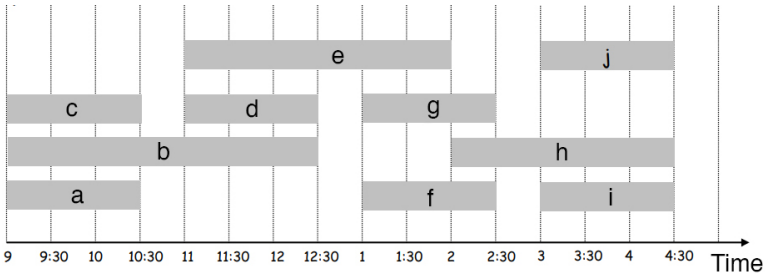


Figura 5.1: Divisione di 4 aule per 10 lezioni

Limite inferiore alla soluzione ottima

Immaginiamo di disporre gli intervalli lungo l'asse delle ordinate in modo che non si sovrapponga con nessun altro intervallo alla stessa altezza. Il numero massimo di intervalli intersecabili con una singola retta verticale che si muove lungo l'asse delle ascisse si

chiama ***profondità.*** Occorre notare che il numero di risorse necessarie (aule) non può essere minore della profondità, ossia $risorse \geq profondita$. In altre parole la profondità ***è un limite inferiore*** al numero di risorse necessarie.

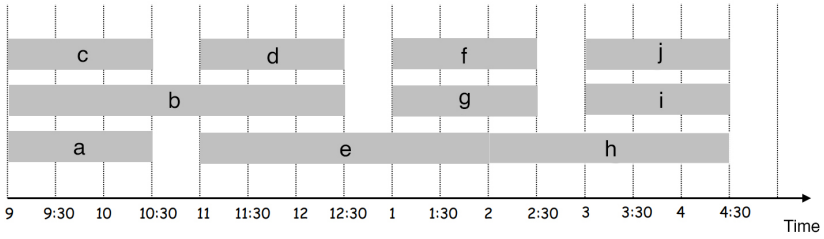


Figura 5.2: Lo stesso insieme di intervalli con profondità tre

È sempre possibile trovare uno schedule pari alla profondità dell'insieme di intervalli? Se così fosse allora il problema di partizionamento si ridurrebbe a constatare quanti intervalli si sovrappongono in un certe punto. L'algoritmo greedy potrebbe funzionare secondo il criterio di allocare una nuova risorsa solo se in un certo momento quelle già allocate sono tutte impegnate. Nel nostro esempio:

- Considera le lezione in ordine crescente dei tempi di inizio;
- Ogni volta che esamina una lezione controlla se può essere allocata in una delle aule già utilizzate per qualcuna delle lezioni già esaminate in precedenza. In caso contrario viene allocata una nuova aula.

Algoritmo greedy

```
1  Ordina gli intervalli in ordine crescente in modo che s_1 ≤ s_2 ≤ ... ≤ s_n;
2  d = 0;
3  for j = 1 che va da 1 a n do
4      if All'intervallo j può essere assegnata una risorsa v (già allocata) then
5          Assegna la risorsa v all'intervallo j;
6      end
7      else
8          Alloca una nuova risorsa d + 1;
9          Assegna la risorsa d + 1 all'intervallo j;
10         d = d + 1;
11     end
12 end
```

L'algoritmo appena scritto ha il problema di non poter essere analizzato nella riga 4: senza sapere altri dettagli implementativi il test nell'if non può essere analizzato. A tal proposito una soluzione potrebbe essere una Coda a priorità (*Min Priority Queue*) per tenere traccia delle risorse e le attività che ne usufruiscono, in particolare inseriremo nella

coda a priorità una entry composta dalla risorsa e una chiave rappresentata dal tempo di fine f_j dove j è l'ultima attività assegnata alla risorsa.

Ad ogni iterazione si estrarrà l'elemento con chiave minore, e quest'ultima si confronterà con il tempo di inizio della nuova attività. È è facile vedere che se si adotta come algoritmo di ordinamento un $MergeSort$ alla riga 1, l'intero algoritmo ***richiederà tempo*** $O(nlogn)$ poiché le operazioni di estrazione e confronto richiedono tempo costante.

```
1  Ordina gli intervalli in ordine crescente in modo che s_1 ≤ s_2 ≤ ... ≤ s_n;
2  d = 0;
3  Inizializza una coda a priorità Q = ∅;
4  for j = 1 che va da 1 a n do
5      if Rimuovo dalla coda l'entry (v, k_v) con chiave minore e k_v ≤ s_j then
6          Assegna la risorsa v all'intervallo j;
7          Inserisco nella coda l'entry (v, f_j);
8      end
9      else
10         Alloca una nuova risorsa d + 1;
11         Assegna la risorsa d + 1 all'intervallo j;
12         Inserisco nella coda l'entry (d + 1, f_j);
13         d = d + 1;
14     end
15 end
```


Dimostrazione della correttezza dell'algoritmo

Lemma Alla $j - esima$ iterazione del for, il valore di d in quella iterazione è pari alla profondità dell'insieme di intervalli $\{1, 2, \ldots, j\}$.

Dimostrazione Possiamo avere due casi:

1. Caso in cui alla $j - esima$ iterazione del for viene allocata una nuova risorsa.

   Quindi la risorsa d è stata allocata perché ciascuna delle altre $d - 1$ risorse già allocate è assegnata al tempo $s_j$ad un intervallo che non è ancora terminato. In altri termini $f_i > s_j$ per ciascuna attività i che sta impegnando una delle $d - 1$ risorse nell'istante di tempo s_j. Siccome abbiamo ordinato gli intervalli in ordine crescente rispetto al tempo di inizio, i $d - 1$ intervalli a cui sono assegnate le $d - 1$ risorse iniziano non più tardi di s_j, ossia $s_i \leq s_j < f_i$ per ciascuna attività i che sta impegnando una delle $d - 1$ risorse.

   Di conseguenza questi $d - 1$ intervalli e l'intervallo $[s_j, f_j]$ si sovrappongono all'istante di tempo s_j, e dato che la profondità per definizione è il massimo numero di intervalli intersecabili dalla retta verticale, quindi $d \leq$ profondità di $\{1, 2, \ldots, j\}$ nell'istante s_j. Abbiamo però osservato anche che in quel preciso istante di tempo $s_j \geq$ profondità di $\{1, 2, \ldots, j\}$, unendo le due osservazioni otteniamo che $d =$ profondità di $\{1, 2, \ldots, j\}$.

2. Caso in cui alla $j-esima$ iterazione del for non viene allocata una nuova risorsa.

   Sia j' l'ultima iterazione prima della $j-esima$ in cui viene allocata una nuova risorsa. Per quanto dimostrato in precedenza $d =$ profondità $\{1,2,\ldots j'\}$, e siccome $\{1,2,\ldots j'\}$ è contenuto in $\{1,2,\ldots j\}$, la profondità di $\{1,2,\ldots j'\} \leq$ profondità di $\{1,2,\ldots j\}$, di conseguenza $d \leq$ profondità di $\{1,2,\ldots j\}$.

   Come abbiamo osservato prima in un certo istante di tempo s_j, $d \geq$ profondità di $\{1,2,\ldots j\}$, e unendo le due affermazioni otteniamo proprio che $d =$ profondità di $\{1,2,\ldots j\}$.

L'algoritmo greedy usa esattamente un numero di risorse pari alla profondità dell'insieme di intervalli $\{1,2,\ldots,n\}$

Dimostrazione Per il lemma appena dimostrato, quando $j = n$ il numero di risorse allocate è uguale alla profondità dell'intervallo $\{1,2,\ldots,n\}$.

5.3 Minimizzazione dei ritardi

Discutiamo ora un problema di *scheduling* simile a quello con cui abbiamo iniziato il capitolo. Nonostante le somiglianze nella formulazione del problema e nell'algoritmo greedy per risolverlo, la dimostrazione che questo algoritmo è ottimale richiede un tipo di analisi più sofisticato.

Quindi consideriamo nuovamente uno scenario in cui si ha una singola risorsa in grado di elaborare una singola attività alla volta. Un certo job j richiederà t_j unità tempo ma deve essere terminato entro il tempo d_j (scadenza). Il ***ritardo*** del job j è definito come $\ell_j = max\{0, f_j - d_j\}$ dove f_j è il tempo di fine del job j. **L'obiettivo da raggiungere** è trovare uno scheduling di tutte le attività che ***minimizzi il ritardo massimo***.

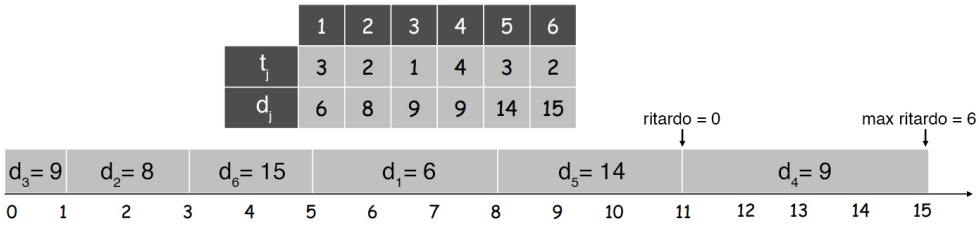


Figura 5.3: Esempio di scheduling con massimo ritardo uguale a 6

Anche in questo caso ci sono diverse possibilità per ordinare le varie attività:

- **Shortest processing time first**

  Considera i job in ordine crescente rispetto ai tempi di esecuzione t_j.

	1	2
t_j	1	10
d_j	100	10

  Controesempio: viene eseguito prima il job 1, di conseguenza il secondo job verrà eseguito all'istante di tempo 2 e terminerà all'istante di tempo 11, avendo quindi un ritardo totale pari a 1. Se avessimo scelto prima il job 2 avremmo avuto un ritardo pari a 0.

- **Smallest Slack**

  Considera i job in ordine crescente rispetto agli scarti $d_j - t_j$

	1	2
t_j	1	10
d_j	2	10

  Controesempio: viene eseguito prima il job 2 che impiega 10 istanti di tempo per terminare, quindi il job 1 sarà eseguito all'istante di tempo 10 e finirà all'istante 11 ritardando di $11-2=9$ istanti di tempo. Se avessimo scelto prima il job 1 avremmo avuto un ritardo massimo pari a 1

La soluzione greedy migliore in questo problema è rappresentata dalla strategia ***Earliest deadline first***, ossia quella che considera i job in ordine crescente rispetto ai tempi d_j nei quali devono essere ordinati (deadline).

Algoritmo greedy Earliest Deadline First

```
1 Ordina le attività in ordine crescente in modo che
    d_1 ≤ d_2 ≤ ... ≤ d_n;
2 t = 0;
3 for j che va da 1 a n do
4 |   Assegna il job j all'intervallo [t, t + t_j];
5 |   s_j = t;
6 |   f_j = t + t_j;
7 |   t = t + t_j;
8 end
```

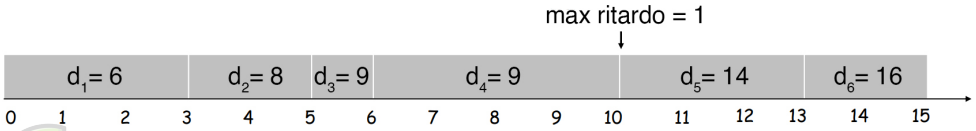


Figura 5.4: Lo stesso esempio di prima, con l'applicazione dell'algoritmo greedy

5.3.1 Ottimalità della soluzione greedy

La dimostrazione dell'ottimalità si basa sulle seguenti osservazioni che andremo poi a dimostrare:

1. La soluzione greedy ha le seguenti due proprietà:
   (a) Nessun ***idle time***: non ci sono momenti in cui la risorsa non è utilizzata.
   (b) Nessuna ***inversione***: se un job j ha scadenza maggiore di quella di un job i allora viene eseguito dopo i.
2. Tutte le soluzioni che hanno in comune le caratteristiche (a) e (b) con la soluzione greedy, hanno lo stesso ritardo massimo della soluzione greedy.
3. Ogni soluzione ottima può essere trasformata in un'altra soluzione ugualmente ottima per cui valgano le proprietà (a) e (b).

Notiamo che la proprietà 3 implica che esista una soluzione ottima che soddisfi che proprietà (a) e (b), ma per la proprietà 2 sappiamo che questa soluzione ha lo stesso ritardo massimo della soluzione greedy che quindi è a sua volta ottima.